FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

OOP14W

computer languages

Skriptum zu 185.A01

Objektorientierte Programmiertechniken

im Wintersemester 2014/2015

Franz Puntigam
Andreas Krall

Technische Universität Wien
Institut für Computersprachen

Inhaltsverzeichnis

Vorwort

Objektorientierte Programmiertechniken (OOP) ist eine Vorlesung mit Übung an der TU Wien. In den Studienplänen der Informatik- und Wirtschaftsinformatik-Bachelorstudien bildet OOP einen Teil des Moduls *Programmierparadigmen* oder *Software Engineering und Projektmanagement.* Diese Titel verraten schon viel über die Lehrveranstaltung: Es geht um ein Kennenlernen der wichtigsten Prinzipien und Techniken bei der Programmierung im objektorientierten Paradigma aus dem Blickwinkel des Software-Engineering. Als Programmiersprache wird Java verwendet.

Von Teilnehmern wird erwartet, dass Sie schon in Java programmieren können, die wichtigsten Algorithmen und Datenstrukturen kennen und Wissen über die objektorientierte Modellierung mitbringen. Auch Sprachkonzepte wie Interfaces, Subtyping, Klassen und Vererbung sollten in Ihren Grundzügen schon bekannt sein. Darauf aufbauend werden unter Anderem folgende Themen behandelt:

- Übersicht über Programmierparadigmen
- Ziele und Eigenschaften des objektorientierten Paradigmas
- Gestaltung von Objektschnittstellen und Untertypbeziehungen
- Umgang mit statischer und dynamischer Typinformation
- Einsatz einiger gängiger Software-Entwurfsmuster

Viel Erfolg bei der Teilnahme an der Lehrveranstaltung!

Franz Puntigam
Andreas Krall

`www.complang.tuwien.ac.at/franz/objektorientiert.html`

1 Paradigmen der Programmierung

Der Begriff *Paradigma* hat eine ganze Reihe unterschiedlicher Bedeutungen. Im Zusammenhang mit der Programmierung versteht man darunter eine bestimmte Denkweise oder Art der Weltanschauung. Entsprechend entwickelt man Programme in einem gewissen Stil. Nicht jeder individuelle Programmierstil ist gleich ein eigenes Paradigma, sondern nur solche Stile, die sich grundlegend voneinander unterscheiden. 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.

Häufig unterscheidet man das auf Maschinenbefehlen aufbauende *imperative* vom auf formalen Modellen beruhenden *deklarativen* Paradigma. Das imperative Paradigma unterteilt man nach der vorherrschenden Programmaufteilung in das *prozedurale* und *objektorientierte* Paradigma, das deklarative nach dem formalen Modell in das *funktionale* und *logikorientierte* Paradigma. Allerdings ist diese einfache Einteilung kaum strukturiert und lässt weniger etablierte Paradigmen unbeachtet.

Man kann auf unterschiedliche Weise Struktur in die Paradigmenvielfalt bringen. Beispielsweise würde sich ein historischer Rückblick gut zur Klärung der Frage eignen, warum bestimmte frühe Paradigmen gescheitert sind und andere bisher erfolgreich waren. Eine Strukturierung nach Anwendungsbereichen, in denen gewisse Paradigmen vorherrschen, würde pragmatische Gesichtspunkte in den Mittelpunkt stellen. Wir versuchen hier die Paradigmen nach den vorrangig betrachteten Aspekten einzuteilen, obwohl sich dabei ein nicht immer ganz orthogonales Gefüge ergibt. Auf diese Weise lässt sich der Gestaltungsspielraum am besten abschätzen.

1.1 Berechnungsmodell und Programmstruktur

Hinter jedem Programmierparadigma steckt ein Berechnungsmodell. Berechnungsmodelle haben immer einen formalen Hintergrund. Sie müssen in sich konsistent und in der Regel Turing-vollständig 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. Mühsam gewonnene Erfahrungen sind es auch, die zur langsamen Weiterentwicklung von Paradigmen führen. So kommen bestimmte Strukturen in die Programme, die im ursprünglichen Formalismus nicht vorhanden waren. Häufig sind gerade diese Strukturen von entscheidender Bedeutung, nicht so sehr die ursprünglichen Eigenschaften der Formalismen.

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 ganz 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 wohl *primitiv-rekursive Funktionen,* die von einer vorgegebenen Menge einfacher Funktionen ausgehen und daraus durch Komposition und Rekursion neue Funktionen bilden, so wie es Programmierer(inne)n geläufig ist. Primitiv-rekursive Funktionen können vieles berechnen, aber nicht alles, was berechenbar ist. Turing-Vollständigkeit erreicht man durch *μ-rekursive Funktionen,* wo der μ-Operator angewandt auf partielle Funktionen das kleinste aller möglichen Ergebnisse liefert. Der ebenfalls Turing-vollständige λ-Kalkül (von dem es mehrere Varianten gibt) wurde unabhängig von primitiv- und μ-rekursiven Funktionen entwickelt und beschreibt Funktionen ohne Notwendigkeit für Rekursion.

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 war und ist vor allem die Möglichkeit, neue Funktionen einfach aus bestehenden Funktionen zusammenzusetzen. Dazu braucht man keine Theorie. Frühe imperative Programmiersprachen wie Algol enthielten dennoch λ-Ausdrücke. Aber in der damaligen Form zusammen mit imperativen Befehlen 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 viel später wurden Lisp-Dialekte um λ-Ausdrücke erweitert. Moderne funktionale Sprachen enthalten selbstverständlich λ-Ausdrücke. Erst in jüngster Zeit werden λ-Ausdrücke 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 mächtige Teilmenge der Prädikatenlogik, die Basis eines Turing-vollständigen Berechnungsmodells bilden. Seit etwa 1965 kann man mittels *Resolution* und *Unifikation* mehr oder weniger automatisch Beweise über Horn-Klauseln ableiten [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 Anwendern und Programmierern, die teilweise 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 ursprünglichen Ziele wurden nichteinmal annähernd erreicht. Heute spielt die logikorientierte Programmierung (im engeren Sinn) nur mehr eine untergeordnete Rolle. Ihr Einfluss ist dennoch sehr stark. So verwenden wir heute fast ausschließlich relationale Datenbanken mit logikorientierten Abfragesprachen.

Constraint-Programmierung: Ein Zweig der Programmiersprachen der 5. Generation verwendete Einschränkungen auf Variablen wie etwa „$x < 5$“ zusätzlich zu solchen wie „A oder B ist wahr“. Fortschrittliche Beweistechniken können solche *Constraints* vergleichsweise ef-

fizient auflösen. Dieser Zweig hat sich verselbständigt. Constraints sind auch mit funktionalen und imperativen Sprachen kombinierbar, nicht nur mit logikorientierten. Heute verwendet man zum Auflösen von Constraints vorwiegend fertige Bibliotheken, die fast überall eingebunden werden können.

Temporale Logik und Petri-Netze: In temporaler Logik kann man in logischen Ausdrücken recht einfach zeitliche Abhängigkeiten abbilden. Beispielsweise kann man festlegen, 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 es darum geht, irgendwelche von der Zeit abhängigen Aussagen oder Ereignisse formal zu beschreiben, etwa die Synchronisation in nebenläufigen Programmen oder die Steuerung von Maschinen.

In etwa denselben Bereichen verwendet man auch verschiedene Arten von *Petri-Netzen,* 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 *Prozessen* (vergleichbar mit Threads) in nebenläufigen Systemen. Die bekanntesten Prozesskalküle sind

CSP (Communicating Sequential Processes) [16] und *π-Kalkül* [30]. Primitive Operationen gibt es ausschließlich für das Senden und Empfangen von Daten, und primitive Operationen sind durch Hintereinanderausführung, Parallelausführung sowie alternative Ausführung miteinander kombinierbar. Während man im λ-Kalkül Turing-Vollständigkeit durch die Übergabe beliebiger Argumente bei Funktionsaufrufen erreicht, 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 vor allem zur Beschreibung *reaktiver* Systeme, die auf Ereignisse in der Umgebung reagieren, wann immer diese auftreten. Endlosberechnungen im λ-Kalkül führen zu undefinierten Ergebnissen. Dagegen sind Endlosberechnungen im π-Kalkül wohldefiniert, aber es können Deadlocks auftreten.

Automaten: Die klassische Automatentheorie wurde in der Frühzeit der Informatik entwickelt. Man unterscheidet Automaten unterschiedlicher Komplexität, die gleichzeitig verschiedene Arten von Grammatiken und entsprechende Sprachklassen darstellen [17]. Obwohl Grammatiken nur Syntax beschreiben, sind die mächtigsten unter ihnen Turing-vollständig. 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 hatten 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, ganz primitiven arithmetischen Operationen und bedingten Anweisungen nur entweder eine While-Schleife oder eine Goto-Anweisung (Sprung an eine beliebige andere Programmstelle) gibt. Diese beiden Sprachen sind Turing-vollständig. Im Gegensatz dazu ist die *LOOP*-Sprache, in der es statt While bzw. Goto eine Schleife mit einer vorgegebenen Anzahl an Iterationen gibt, nur äqui-

valent 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 Praktische Realisierung

Beim Programmieren denkt man 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 nicht selten 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 man verwendet zusätzliche Sprachkonzepte zur Verbesserung der Kombinierbarkeit.

Konsistenz: Programmiersprachen und -paradigmen müssen zahlreiche Erwartungen hinsichtlich verschiedenster Aspekte erfüllen. Ein ein-

ziger Formalismus reicht als Grundlage dafür nicht aus. Meist muss man mehrere sprachliche Konzepte – etwa solche zur Beschreibung von Algorithmen und andere zur Beschreibung von Datenstrukturen – zu einer Einheit verschmelzen. 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. Aus diesem Grund wird man kaum ein zufriedenstellendes Ergebnis erhalten, wenn man einfach nur die beste formale Grundlage für jeden Aspekt wählt und alle entsprechenden Formalismen zusammenfügt. Ein riesiges Konglomerat mit zahlreichen Widersprüchen würde entstehen. Gute Sprachen und Paradigmen kommen mit wenigen konsistenten Konzepten aus. Im Umkehrschluss bedeuted 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 versteht man unter Abstraktion nicht mehr nur die Abstraktion über das System, sondern abstrakte Sichtweisen vieler weiterer Aspekte. Man kann Abstraktionen von fast allem entwickeln und in Programmen darstellen. Im Laufe der historischen Entwicklung von Programmierparadigmen hat der erreichbare Abstraktionsgrad ständig zugenommen und ist heute sehr hoch. In den Abschnitten 1.2 und 1.3 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. Viele Programmierer(innen) bevorzugen systemnahe Paradigmen aus diesem Grund und weil sie ihr eigenes System gut kennen und nutzen wollen. Unverzichtbar ist Systemnähe dann, wenn Hardwarekomponenten direkt angesprochen werden müssen. Gelegentlich wird aus wirtschaftlichen Gründen eine starke

Betriebssystemabhängigkeit auf Kosten der Portabilität gewünscht. In anderen Bereichen möchte man dagegen aus Sicherheits- und Portabilitätsgründen vor direkten Zugriffen auf das System geschützt sein. Ein Paradigma, das überall passt, wird es kaum geben können.

Unterstützung: Das ansonsten beste Paradigma hat keinen Wert, wenn entsprechende Programmiersprachen, Entwicklungswerkzeuge sowie vorgefertigte Programmteile fehlen. So manches ursprünglich nur mittelmäßige Paradigma hat sich wegen guter Unterstützung trotzdem durchgesetzt und wurde im Laufe der Zeit verbessert. Es sind nicht nur inhärente Eigenschaften, die den Erfolg eines Paradigmas ausmachen. Ohne Personen, die an den Erfolg glauben und viel Zeit und Geld in die Entwicklung investieren, kann sich kein Paradigma durchsetzen. Nicht selten sind Kleinigkeiten und Zufälle für den Erfolg ausschlaggebend. Beispielsweise hat die beinahe schon tot geglaubte Programmiersprache Smalltalk zum Jahrtausendwechsel plötzlich eine Renaissance erlebt, nur weil zufällig keine ganzen Zahlen mit beschränktem Wertebereich unterstützt wurden und daher keine Überläufe möglich waren. Manchmal ist auch eine starke Unterstützung durch einflussreiche Unternehmen essentiell. Ein Beispiel dafür ist die große Verbreitung von Java und C#.

Beharrungsvermögen: Obwohl Softwareentwickler(innen) mit ständigen Innovationen leben, sind sie größeren Änderungen der Programmierparadigmen gegenüber kaum aufgeschlossen. 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. Man spricht von *Killerapplikationen,* also erfolgreichen Programmen, die den Erfolg einer Technik oder eines Paradigmas so deutlich aufzeigen, dass sie das Potential dazu haben, althergebrachte Techniken oder Paradigmen in diesem Bereich zu ersetzen. 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. Man kann 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, sodass man mit hoher Wahrscheinlichkeit davon ausgehen kann, dass Sie es auch in absehbarer Zukunft noch sein werden.

Die wichtigsten Werkzeuge der Softwareentwickler 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. Solche technischen Entwicklungen beeinflussen Programmierparadigmen. Sprachkonzepte, die sich früher nicht durchgesetzt haben, können wegen neuer Implementierungstechniken plötzlich erfolgreich sein.

1.1.3 Evolution und Struktur im Kleinen

Widersprüchliche Ziele. Paradigmen entwickeln sich auch ohne Paradigmenwechsel langsam, aber stetig weiter. Die Entwicklung verläuft meist wellenförmig, selten geradlinig. Eine Triebfeder dafür sind immer wieder neue praktische Erfahrungen im Spannungsfeld zwischen zum Teil widersprüchlichen Forderungen. So ist es etwa unmöglich, folgende Forderungen gleichzeitig und in vollem Umfang zu erfüllen:

- Flexibilität und Ausdruckskraft sollen in kurzen Texten 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.

Die althergebrachte dynamische Programmierung bevorzugt den ersten gegenüber dem zweiten Punkt, während die althergebrachte statische Programmierung den zweiten gegenüber dem ersten bevorzugt. Viele jüngere, hauptsächlich (aber nicht nur) objektorientierte Sprachen erzielen einen hohen Grad an Flexibilität und Ausdruckskraft ebenso wie Lesbarkeit und Sicherheit, aber häufig nur auf Kosten der einfachen Verständlichkeit – dritter Punkt. Obwohl es auf längere Sicht eine Tendenz in Richtung Vernachlässigung des dritten Punkts gibt, werden neue Sprachen fast immer mit verbesserter Einfachheit und Verständlichkeit beworben. Bei

Erfolg verschwindet diese Zielsetzung aber rasch aus dem Blickfeld. Man kann auch erkennen, dass statische und dynamische Programmierung zwar praktisch schon seit Beginn der Informatik zusammen existieren, aber gelegentlich Pendelbewegungen einmal hin zu eher statischer und dann wieder hin zu eher dynamischer Programmierung stattfinden.

Strukturierte Programmierung. Selten kann sich in diesem Spannungsfeld eine Idee so eindeutig durchsetzen, dass nach einer gewissen Zeit alle Pendelbewegungen aufhören. Das Paradebeispiel dafür ist die *strukturierte Programmierung,* die mehr Struktur in die prozedurale Programmierung bringt. Jedes Programm bzw. 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, Rekursion oder Ähnliches)

Heute erscheinen diese Kontrollstrukturen nicht nur in der prozeduralen Programmierung als so selbstverständlich, dass man sich kaum mehr etwas anderes vorstellen kann. Nur durch die genaue Ausformung dieser Kontrollstrukturen ergeben sich noch Unterschiede. Das mächtige und früher allgegenwärtige Goto, das einen Gegenpol zur strukturierten Programmierung bildet, wird heute fast gar nicht mehr verwendet.

Ein wesentliches Ziel der strukturierten Programmierung besteht darin, dass jede Kontrollstruktur nur je einen wohldefinierten Einstiegs- und Ausstiegspunkt hat. Das erleichtert das Verfolgen des Programmpfads. Man erhöht also die Lesbarkeit (und damit auch die Sicherheit) auf Kosten einer etwas verringerten Flexibilität und Ausdruckskraft. Es ist nicht selbstverständlich, dass jemand freiwillig zugunsten der Lesbarkeit auf Flexibilität verzichtet. In diesem Fall hat man mit der verbesserten Lesbarkeit so eindeutig positive (und mit Goto so negative) Erfahrungen gemacht, dass schließlich sogar eingefleischte Freiheitsfanatiker die Einschränkungen in Kauf genommen haben. Bessere Lesbarkeit bedeutet in diesem Fall auch bessere Kombinierbarkeit: Es ist stets klar, wie man die Kontrollstrukturen aneinanderfügen kann. Damit wird das Programmieren einfacher. Man gewinnt gleichzeitig aus praktischer Sicht wieder etwas an Flexibilität, weil man die vielfältigen Kombinationsmöglichkeiten besser überblicken und nutzen kann, während die theoretische Mächtigkeit von Goto wegen fehlender Übersicht praktisch nur zu einem kleinen Teil genutzt wurde.

In der funktionalen Programmierung sind Seiteneffekte unerwünscht. Jedoch wären Schleifen ohne Seiteneffekte ebenso sinnlos wie die Hintereinanderausführung, die ohne Zuweisungen ja nichts bewirkt. Statt Schleifen verwendet man Rekursion und statt Hintereinanderausführung eine Ineinanderschachtelung von Funktionsaufrufen. Im Prinzip nutzt daher auch die funktionale Programmierung eine Variante der strukturierten Programmierung aufbauend auf Sequenz, Auswahl und Wiederholung. Auch die logikorientierte Programmierung beruht auf diesem Muster.

Seiteneffekte und Querverbindungen. Ein Problem der imperativen Programmierung wird durch struktrurierte Programmierung nicht gelöst: Programmfortschritte erzielt man über Seiteneffekte, vor allem durch Zuweisung neuer Werte an Variablen (sowie Ein- und Ausgaben, die letztlich wieder über Zuweisungen realisiert sind). Das wäre kein Problem, könnte man beim Programmieren alle Veränderungen der Programmzustände durch Zuweisungen überblicken. Leider ist so mancher Seiteneffekt gut versteckt. Nehmen wir als Beispiel einen Prozeduraufruf $f(x)$, der die Zahlen im Array x sortiert. Vielleicht hat f weitere uns unbekannte Seiteneffekte, sammelt etwa statistische Daten über die sortierten Zahlen. Während Aufrufe wie $f(x); f(y)$ mit verschiedenen Arrays so wie gedacht funktionieren, würde eine Aufrufsequenz $f(x); f(x); f(y)$ durch das semantisch vermeintlich irrelevante doppelte Sortieren von x die statistischen Daten verfälschen. Diese Daten sind unerwartet mit x und y verbunden. Auswirkungen versteckter Seiteneffekte und entsprechender Querverbindungen sind in der Praxis oft gravierend.

Es gibt mehrere Ansätze um versteckte Seiteneffekte in den Griff zu bekommen. Wichtig ist die korrekte und vollständige Dokumentation. Einen darüber hinausgehenden recht radikalen Ansatz verfolgen deklarative Paradigmen, einen eher gemäßigten objektorientierte Paradigmen.

Die deklarative Programmierung strebt *referentielle Transparenz* als Eigenschaft aller Ausdrücke im Programm an. Ein Ausdruck ist referentiell transparent, wenn er durch seinen Wert ersetzt werden kann, ohne die Semantik des Programms dadurch zu ändern. Man kann den Ausdruck auch durch einen anderen, ebenfalls referentiell transparenten Ausdruck desselben Werts ersetzen. Beispielsweise lässt sich $3+4$ durch 7 oder $14/2$ ersetzen, ohne die Semantik zu ändern. Auch $f(x)+f(x)$ lässt sich durch $2 * f(x)$ ersetzen, obwohl die Bedeutung der Funktion f unbekannt ist. Jedoch darf f keine Seiteneffekte haben und nicht von Variablen abhän-

gen, deren Werte sich im Laufe der Zeit ändern könnten. Variablenwerte sind nur mittels Seiteneffekten änderbar. Daher erreicht man referentielle Transparenz durch gänzlichen Verzicht auf Seiteneffekte. Seiteneffektfreiheit ist Mittel zum Zweck, nicht das Ziel. Referentielle Transparenz ist das Ziel. Man kann Ausdrücke vereinfachen, ohne Angst vor versteckten Querverbindungen und dadurch verursachten Fehlern haben zu müssen. Ohne referentielle Transparenz dürfte man $f(x)+f(x)$ nicht durch $2*f(x)$ ersetzen, weder gedanklich noch tatsächlich.

Ein- und Ausgaben sind Seiteneffekte, ohne die auch die deklarative Programmierung nicht auskommt. Dieses Problem hat man in neueren funktionalen Sprachen trickreich gelöst: Man erlaubt Ein- und Ausgaben nur gut sichtbar ganz oben in der Aufrufhierarchie und verbannt sie aus allen anderen Funktionen. Auf diese Weise ist referentielle Transparenz in allen Programmteilen gewährleistet. In älteren Sprachen oder in Paradigmen, wo dieser Ansatz (noch) nicht funktioniert, verzichtet man im Zusammenhang mit Ein- und Ausgabe auf referentielle Transparenz.

Auch imperative Paradigmen unterstützen Funktionen und erlauben bis zu einem gewissen Grad einen funktionalen Programmierstil. Wo es einfach geht setzt man überall, auch in der imperativen Programmierung auf referentielle Transparenz. So sind primitive Operationen wie $+$, $-$, $*$ und $/$ fast durchwegs referentiell transparent, aber nicht nur diese.

In der objektorientierten Programmierung geht man, wo es keine referentielle Transparenz gibt, mit Seiteneffekten ganz offensiv um: Man nimmt stets an, dass es entsprechende Querverbindungen gibt. Trotz vieler Querverbindungen kann man sie überschaubar halten, indem man sie lokal auf einzelne Objekte beschränkt. Objekte stellen in diesem Zusammenhang ein Strukturierungsmittel dar um Prozeduren und Variablen entstrechend der Querverbindungen zu Einheiten zusammenzufassen. Mit Prozeduren alleine geht das nicht, da Aufrufhierarchien von Natur aus anders strukturiert sind als Querverbindungen – daher das „Quer“ im Namen. Objektorientierte Programme lassen Zustandsänderungen auf der Ebene einzelner Objekte verständlich werden, ohne (wie in der prozeduralen Programmierung) den gesamten Programmzustand überblicken zu müssen.

First-Class-Entities. Funktionen 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 *First-Class-Entities* sind. Das bedeutet, dass Funktionen wie nor-

male Daten verwendet werden. Man kann Funktionen zur Laufzeit erzeugen (z.B. λ-Ausdrücke), in Variablen ablegen, als Argumente an andere Funktionen übergeben und als Ergebnisse von Funktionen zurückbekommen. Methoden in Java sind dagegen keine First-Class-Entities. Man kann sie zwar aufrufen, aber nicht an Variablen zuweisen oder als Argumente übergeben. Objekt-ähnliche Konzepte gibt es ebenso in vielen Paradigmen, z.B. in Form von Modulen (siehe Abschnitt 1.2), aber im Wesentlichen nur in der objektorientierten Programmierung sind Objekte als First-Class-Entities wie alle anderen Daten erzeug- und verwendbar. Offensichtlich bekommt so manches Konzept erst durch die gleiche Behandlung wie alle anderen Daten jene überragende Bedeutung, die notwendig ist, um ein Paradigma darauf aufzubauen.

First-Class-Entities sind häufig viel komplizierter als vergleichbare Konzepte, weil die uneingeschränkte Verwendbarkeit eine große Zahl an zu berücksichtigenden Sonderfällen nach sich zieht. Diesen Aufwand möchte man zugunsten von Einfachheit und Effizienz vermeiden. Man akzeptiert ihn nur, wenn die uneingeschränkte Verwendbarkeit echte Vorteile bringt.

Betrachten wir beispielsweise Funktionen. Solange wir nur rekursive Aufrufe (und vordefinierte Schleifen) für Wiederholungen verwenden, bieten Funktionen als First-Class-Entities keine Vorteile. Die Verwendung entspricht der von primitiv-rekursiven Funktionen. Aber *Funktionen höherer Ordnung* (also Funktionen mit Funktionen als Parametern) bringen eine neue Dimension in die Programmierung. Sie sind wie Kontrollstrukturen verwendbar, etwa zur Realisierung eigener Arten von Schleifen. So erhält auch der λ-Kalkül seine Ausdrucksstärke. Ein häufiger Gebrauch von Funktionen höherer Ordnung führt zu einem eigenen Paradigma, der *applikativen Programmierung,* bei der man quasi Schablonen von Programmteilen schreibt, die dann durch Übergabe von Funktionen (zum Füllen der Löcher in den Schablonen) ausführbar werden. In diesem Stil lassen sich sehr kompakte und trotzdem verständliche Programme schreiben. Funktionen erster (= nicht-höherer) Ordnung eignen sich dafür nicht.

Erst die First-Class-Eigenschaft macht Objekte vielseitig verwendbar. Spezialisierte Objekte (z.B. Iteratoren) kann man wie Kontrollstrukturen verwenden und dadurch etwas mit Funktionen höherer Ordnung Vergleichbares erreichen. Im Detail ergeben sich aber große Unterschiede; ein Konzept kann das andere nicht vollständig ersetzen. Daher gibt es Bemühungen, Funktionen bzw. Methoden höherer Ordnung auch in der objektorientierten Programmierung besser zu unterstützen. Das bedeutet jedoch nicht, dass diese Methoden auch referentiell transparent sein müssen.

1.2 Programmorganisation

Im Zusammenhang mit Berechnungsmodellen und einfachen Programmstrukturen spricht man vom *Programmieren im Kleinen.* Bei der *Programmierung im Großen* geht es um die Zerlegung großer Programme in überschaubare Einheiten. Man beschäftigt sich nicht mit einzelnen Variablen, Typen, Funktionen, etc., sondern mit größeren Gruppen davon und den Beziehungen zwischen diesen Gruppen. Das wesentliche Ziel ist die *Modularisierung* von Programmen. Sie soll so erfolgen, dass größtmögliche Flexibilität und Wartbarkeit über einen langen Zeitraum erzielt wird.

1.2.1 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 gegeneinander austauschbar sind. Wenn man etwas genauer hinsieht, muss man verschiedene Formen von Modularisierungseinheiten unterscheiden.

Modul: Darunter versteht man 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 *Linker* oder erst zur Laufzeit zum ausführbaren Programm verbunden. Wenn einzelne Module geändert werden, braucht man 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 sogar 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 versteht man 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 verwendet. Schnittstellen unterscheiden klar zwischen Modulinhalten, die von anderen Modulen 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 gewissen Regeln entsprechende Zugriffsmöglichkeit von außen bestehen muss. Änderungen privater Modulinhalte wirken sich 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, muss man in Modulen häufig auch angeben, 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. Derselbe Name kann in einem anderen Modul (also in einem anderen Namensraum) eine ganz andere Bedeutung haben. Beim Programmieren kann man Namen in anderen Modulen ignorieren. Allerdings kann es vorkommen, dass man aus unterschiedlichen Modulen unterschiedliche Inhalte desselben Names importieren muss. Solche Namenskonflikte kann man durch *Umbenennung* während des Importierens oder durch *Qualifikation* des Namens (darunter versteht man das Voranstellen des Modulnamens vor den importierten Namen) auflösen.

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 von Modulen widerspricht. In der Praxis ist die gemeinsame Übersetzung voneinander zyklisch abhängiger Module manchmal dennoch erlaubt. Wesentliche Vorteile ergeben sich aber nur bei getrennter Übersetzung.

Zyklen in den Abhängigkeiten lassen sich auflösen, indem man Module in je zwei getrennte Module aufteilt, 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 die Implementierungen nicht direkt auf andere Implementierungen sondern auf entsprechende Schnittstellen zugreifen, sind danach die Implementierungen getrennt voneinander übersetzbar.

Objekt. Im Gegensatz zu Modulen sind Objekte keine Übersetzungseinheiten und werden in der Regel 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 nennt man *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 kann man eine solche Abstraktion auch mit Modulen erreichen, aber durch zusätzliche Verwendungszwecke nur eingeschränkt.

Anders als Module sind Objekte praktisch immer First-Class-Entities. 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 demselben 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 denselben Zustand und dasselbe Verhalten haben, auch wenn sie nicht identisch sind. In diesem Fall ist ein Objekt eine *Kopie* des anderen.

Es gibt eine breite Palette an Möglichkeiten zur Festlegung der Details. So ist Data-Hiding praktisch 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 man kommt ganz ohne Klassen und ähnliche Konzepte aus. Meist verwendet man 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 derselben Klasse haben gleiches Verhalten. Objekte unterschiedlicher Klassen verhalten sich unterschiedlich, sodass nur Objekte derselben Klasse gleich oder identisch sein können. Der Begriff Klasse kommt von dieser Klassifizierung anhand des Verhaltens.

Meist ist es möglich, Klassen von anderen Klassen abzuleiten und dabei Methoden zu erben. Auf eine bestimmte Art angewandt (siehe Abschnitte 1.2.3 und 1.3) 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 derselben Klasse gleich ist, kümmert man sich darum fast immer auf der Klassenebene. Allerdings sind die Kriterien häufig aufgelockert. Beispielsweise können auch andere Objekte derselben Klasse auf private Inhalte zugreifen, und es gibt mehrere Stufen der Sichtbarkeit. Dadurch, dass man neue Objekte nur über Klassen (oder durch Kopieren) erzeugt, kann man trotz größerer Flexibilität dennoch die Ziele des Data-Hiding erreichen.

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. Sowohl 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.

Die Einbindung von Komponenten in Programme ist aufwendiger als die von Modulen, da dabei 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 (manchmal unnötig aufwendig erscheinende) Vorgehensweise ermöglicht zyklische Abhängigkeiten zwischen eingebundenen Komponenten.

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

Der klassische Lösungsansatz besteht darin, die globale Namensverwaltung ganz den Softwareentwicklern zu überlassen. Beim Anwenden von Werkzeugen (z.B. Compilern) muss man alle Dateien anführen, welche die zu bearbeitenden Modularisierungseinheiten enthalten. Das einzige Hilfsmittel sind oft standardmäßig vorgegebene Verzeichnisse, in denen automatisch nach verwendeten Modul- oder Klassennamen gesucht wird.

Etwas fortgeschrittener sind Modularisierungseinheiten, die man Namensräume nennt. Sie fassen mehrere Modularisierungseinheiten zu einer Einheit zusammen, ohne die getrennte Übersetzbarkeit zu zerstören. Meist entstehen dabei hierarchische Strukturen, die Verzeichnisstrukturen ähneln und oft tatsächlich auf Verzeichnisse abgebildet werden. 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 verwendet man 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.

1.2.2 Parametrisierung

Ein altbekanntes Schlüsselkonzept zur Steigerung der Flexibilität von Modularisierungseinheiten ist deren Parametrisierung. Darunter versteht man im weitesten Sinne, dass man in der Implementierung der Modularisierungseinheiten Löcher lässt, die man erst später füllt. Ein Beispiel dafür haben wir schon betrachtet: Aus einem Modul wird eine Komponente, wenn man zunächst offen lässt, von welchen anderen Komponenten etwas importiert wird; erst beim Zusammensetzen des Systems werden diese Komponenten bestimmt. Im Allgemeinen kann man beliebige Teile offen lassen, und das Befüllen der Löcher kann zu unterschiedlichen Zeitpunkten aus verschiedenen Quellen erfolgen.

Befüllen zur Laufzeit. Am einfachsten ist es, wenn das Befüllen der Löcher erst zur Laufzeit erfolgt und die dazu verwendeten Daten First-Class-Entities sind. In diesem Fall kann man die Löcher durch einfache Variablen darstellen. 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. Man kann die Werte auf unterschiedliche Weise zu den Variablen bringen, unter anderem so:

Konstruktor: Beim Erzeugen eines Objekts wird ein Konstruktor ausgeführt, der die Objektvariablen initialisiert. Diesen Konstruktor versieht man mit formalen Parametern. An der Stelle, an der die Ob-

jekterzeugung veranlasst wird, übergibt man Werte als aktuelle Parameter an den Konstruktor, die dann zur Initialisierung verwendet werden. Das ist die häufigste und einfachste Form der Parametrisierung, nicht nur in objektorientierten Sprachen.

Initializierungsmethode: In einigen Fällen sind Konstruktoren nicht verwendbar, beispielsweise wenn Objekte durch Kopieren erzeugt werden oder zwei zu erzeugende Objekte voneinander abhängen; man kann ja das später erzeugte Objekt nicht an den Konstruktor des zuerst erzeugten Objekts übergeben. Solche Probleme kann man durch Methoden lösen, 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: Man kann Werte an zentralen Stellen (etwa in globalen Variablen oder als Konstanten) ablegen, 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 kann man direkt auf die Variablen oder Konstanten zugreifen oder Methoden verwenden. Bei Klassen kann man 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 auch zur *Dependency-Injection.* Dabei überträgt man die Verantwortung für das Erzeugen und Initialisieren von Objekten an eine zentrale Stelle (z.B. eine Klasse), von der aus man die Abhängigkeiten zwischen den Objekten überblicken und steuern kann.

Generizität. Unter Generizität versteht man eine Form der Parametrisierung, bei der Löcher (zumindest konzeptuell) bereits zur Übersetzungszeit befüllt werden. Daher können alle Arten von Modularisierungseinheiten generisch sein (also Löcher enthalten, die mittels Generizität befüllt werden), aber beispielsweise auch Funktionen und Ähnliches. Das, womit die Löcher gefüllt werden sollen, wird zunächst durch generische Parameter bezeichnet. In allen Löchern, die mit Demselben befüllt werden sollen, stehen auch dieselben generischen Parameter. Später, aber noch vor der

Programmausführung werden die Parameter durch das Einzufüllende ersetzt. Bevorzugt stehen generische Parameter nicht für gewöhnliche Werte, sondern für Konzepte, die keine First-Class-Entities sind. Häufig sind das Typen. In diesem Fall nennt man generische Parameter auch *Typparameter*. Generizität ist also vorwiegend für solche Fälle gedacht, wo das Befüllen der Löcher zur Laufzeit nicht funktioniert.

Grundsätzlich ist Generizität sehr einfach und schon lange erprobt. In der Praxis ergeben sich aber viele Schwierigkeiten. So muss der Compiler mehrere Varianten desselben Cods verwalten, den generischen Quellcode sowie eine oder mehrere durch Füllen der Löcher daraus generierte Variante(n). Fehlermeldungen, die sich auf generierten Code beziehen, kann man kaum verständlich ausdrücken. Oft braucht man Einschränkungen auf den generischen Parametern, das heißt, das dafür Einzufüllende muss bestimmte Bedingungen erfüllen. Da es sich nicht um First-Class-Entities handelt, lassen sich manche derartige Einschränkungen nur schwer ausdrücken. Besonders schwierig wird es, wenn für mehrere Vorkommen desselben generischen Parameters verschiedene Einschränkungen gelten.

Annotationen. Annotationen sind optionale Parameter, die man zu unterschiedlichsten Sprachkonstrukten hinzufügen kann. Ein Beispiel dafür ist die Annotation überschriebener Methoden in Java mittels `@Override`. Annotationen werden von Werkzeugen verwendet oder aber auch 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 einfach ignorieren. Diese Form der Annotationen wirkt sich statisch, also zur Übersetzungszeit aus. Man kann Annotationen meist auch dynamisch, also zur Laufzeit abfragen. 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. Sie 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 Löcher, 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

setzt man Annotationen oft in Situationen ein, wo Informationen nicht nur von lokaler Bedeutung sind, sondern auch System-Werkzeuge (wie einen Compiler) oder das Betriebssystem steuern oder zumindest beeinflussen.

Aspekte. Auch in der aspektorientierten Programmierung kommt man in der Regel ohne Spezifikation von Löchern im Programm aus. Stattdessen fügt man zu einem bestehenden Programm von außen neue *Aspekte* hinzu. 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 entspechend der Aspekte. Meist geschieht dies vor der Übersetzung des Programms, manche Aspect-Weaver erledigen diese Aufgabe erst zur Laufzeit. Beispielsweise kann man über Aspekte recht einfach erreichen, dass bestimmte Aktionen im Programm zuverlässig in einer Log-Datei protokolliert werden, ohne den Source-Code des Programms dafür ändern zu müssen. Die Aspekte lassen sich vor der Compilation des Programms leicht austauschen oder weglassen, sodass man auf einfache Weise ganz unterschiedliches Programmverhalten erreichen kann. Über Aspekte kann man etwa die Generierung von bestimmter Debug-Information veranlassen und diese auch leicht wieder wegnehmen.

Bestimmte Aufgaben lassen sich durch Aspekte überzeugend rasch und einfach lösen. Für andere Aufgaben sind Aspekte dagegen kaum geeignet. Ein Problem besteht darin, dass die Bestimmung der betroffenen Punkte im Programm gelegentlich Wissen über so manches Implementierungsdetail voraussetzt. Wenn sich solche Details ändern, müssen auch die Aspekte entsprechend geändert 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 Löcher sich ändern, dann muss sich auch das ändern, was zum Befüllen der Löcher verwendet wird. Solche notwendigen Änderungen behindern die Wartung gewaltig. Vor allem muss man für die Änderungen alle Stellen kennen,

an denen eine Modularisierungseinheit verwendet wird. Bei Modularisierungseinheiten, die in vielen unterschiedlichen Programmen über die ganze Welt verteilt zum Einsatz kommen, ist das so gut wie unmöglich. Nachträgliche Änderungen der Löcher in solchen Modularisierungseinheiten sind dadurch praktisch kaum durchführbar.

1.2.3 Ersetzbarkeit

Eine Möglichkeit zur praxistauglichen nachträglichen 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 keinerlei Änderungen an Stellen nach sich zieht, an denen A (bzw. nach der 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 man sich, von außen betrachtet, von A und B erwartet. 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 dasselbe 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. Man kann entsprechende Schnittstellen auf verschiedene Weise spezifizieren:

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. Man nennt eine solche Schnittstelle *Signatur* der Modularisierungseinheit. In Kapitel 2 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 man sich hinsichtlich der Ersetzbarkeit jedoch nur auf Signaturen verlässt, 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 man statt des Inhalts von A den entsprechenden Inhalt von B verwendet. Dennoch verlässt man sich eher auf Signaturen als ganz auf Ersetzbarkeit zu verzichten.

Abstraktion realer Welt: Neben der Signatur beschreibt man Schnittstellen durch Namen und informelle Texte, 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, und 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 Einzelfällen 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*) und wie – vor allem 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 informal und nicht präzise. Vor allem komplexere Vorbedingungen stehen nicht selten in Konflikt zu Data-Hiding, weil sie von Programmzuständen abhängen, die eigentlich nach au-

ßen 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. Genaugenommen 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. Generell dürfen die Protokolle nicht beliebig komplex sein, da die Konsistenz sonst nicht mehr entscheidbar wäre. 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. Man versucht stets, Programme so zu gestalten, dass jeder Programmteil möglichst problemlos durch einen anderen Programmteil ersetzbar ist. Einerseits achtet man auf Ersetzbarkeit von Objekten innerhalb eines Programms. 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.

In der objektorientierten Programmierung verwendet man als Basis für Schnittstellenbeschreibungen meist Abstraktionen, in jüngerer Zeit häufig gepaart mit Zusicherungen. Abstraktionen werden dabei über Klassen realisiert. Ersetzbarkeit wird nur dann als gegeben angesehen, wenn es auch eine Klassenableitung gibt. Programmierer(innen) haben sich an die doppelte Bedeutung von Ableitungen – für Vererbung und Ersetzbarkeit – gewöhnt und betrachten sie als zusammengehörig, obwohl Vererbung aufgrund der Definition nichts mit Ersetzbarkeit zu tun hat.

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.

1.3 Typisierung

Praktisch alle aktuellen Programmiersprachen verwenden Typen, deren Bedeutung stetig zunimmt. Und das, obwohl Typen Einschränkungen darstellen: Man kann nicht beliebige Werte, Ausdrücke, etc. verwenden, sondern nur solche, die vorgegebenen Typen enstprechen. Als Gegenleistung erhält man verbesserte Planbarkeit – weil man früher weiß, womit man es zu tun hat – und bestimmte Formen der Abstraktion. Beides hat einen positiven Einfluss auf die Lesbarkeit und Zuverlässigkeit von Programmen.

1.3.1 Typkonsistenz, Verständlichkeit und Planbarkeit

Berechnungsmodelle kennen nur je eine Art von Werten, z.B. nur Zahlen, Symbole oder Terme. In Programmiersprachen unterscheidet man jedoch mehrere Arten von Werten, z.B. ganze Zahlen, Fließkommazahlen und Zeichenketten. Typen helfen bei der Klassifizierung dieser Werte. Viele Operationen sind nur für Werte bestimmter Typen definiert. Beispielsweise kann man nur ganze Zahlen oder Fließkommazahlen miteinander multiplizieren, aber keine Zeichenketten. Wenn die Typen der Operanden mit der Operation zusammenpassen, sind die Typen *konsistent.* Andernfalls tritt ein *Typfehler* auf. Ganz ohne Typen kommt man nur aus, wenn alle Operationen auf alle Operanden sinnvoll anwendbar sind, was praktisch nie der Fall ist. Daher gibt es in jeder Programmiersprache Typen.

Typprüfungen. Es gibt jedoch erhebliche Unterschiede zwischen Sprachen hinsichtlich des Umgangs mit Typen. Nur wenige sehr hardwarenahe Sprachen prüfen die Konsistenz von Typen gar nicht, etwa Assembler und Forth; bei einem Typfehler werden Bit-Muster einfach falsch interpretiert. Wesentlich mehr Sprachen prüfen die Typkonsistenz in vielen, aber nicht allen Fällen; etwa bleiben in C Type-Casts und Array-Grenzen unüberprüft. Die Mehrheit der Sprachen prüft Typkonsistenz vollständig.

Sprachen unterscheiden sich erheblich hinsichtlich des Zeitpunkts, an dem Typen bekannt sind und überprüft werden. In dynamischen Spra-

chen ergeben sich Typen erst zur Laufzeit und werden erst bei der Anwendung einer Operation (dynamisch) überprüft. In statischen Sprachen hat bereits der Compiler viel Information über Typen und nutzt sie für statische Prüfungen. Statische Typprüfungen haben einen Vorteil gegenüber dynamischen: Man kann sicher sein, dass zur Laufzeit keine entsprechenden Typfehler auftreten. Bei dynamischer Prüfung muss man immer mit Typfehlern rechnen. Allerdings ist die statisch verfügbare Information begrenzt. Das heißt, man muss manche Aspekte (etwa Array-Grenzen) dynamisch prüfen, oder man schränkt die Flexibilität der Sprache so weit ein, dass statische Prüfungen immer ausreichen. In den letzten Jahrzehnten wurden große Fortschritte bei der statischen Typanalyse erzielt, sodass es heute hinreichend flexible Sprachen ausschließlich auf der Basis statischer Typprüfungen gibt, z.B. Haskell. Viele Sprachen wie Java bevorzugen dennoch einen Mix aus dynamischer und statischer Prüfung.

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 ganz bestimmter, eher leicht auffindbarer Fehler. Manchmal hört man die Meinung, dynamische 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 dynamischen Sprachen üblicherweise nicht gibt) das *Lesen und Verstehen* der Programme wesentlich 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 bei statischer Typprüfung müssen nicht alle Typen explizit hingeschrieben sein. Viele Typen kann ein Compiler aus der Programmstruktur herleiten; man spricht von *Typinferenz.* Beispielsweise braucht man in Haskell keinen einzigen Typ hinzuschreiben, obwohl der Compiler alle Typen statisch auf Basis von Typinferenz prüft. Zur Verbesserung der Lesbarkeit kann und soll man Typen dennoch explizit anschreiben. Bis zu einem gewissen Grad erhöhen aber auch schon statische Typprüfungen alleine die Verständlichkeit, obwohl keine Typen dastehen. Das hat mit der geringeren Flexibilität zu tun: Man darf keine so komplexen Programmstrukturen verwenden, dass der Compiler bei der Prüfung der Typkonsistenz überfordert wäre. Einfachere Programmstrukturen sind nicht nur für Compiler, sondern auch für Menschen leichter verständlich.

Typen und Entscheidungsprozesse. Berechnungsmodelle beschreiben ausschließlich die Programmausführung. Alle bedeutenden Entscheidungen hinsichtlich der Programmstruktur und des Programmablaufs werden jedoch schon vorher, bei der Programmerstellung getroffen. Hier ist eine Grobeinteilung 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.
- Zum Zeitpunkt der *Erstellung von Übersetzungseinheiten* werden die meisten wichtigen Entscheidungen getroffen, auf die man beim Programmieren Einfluss hat. Hierzu braucht man viel Flexibilität.
- Manche wichtigen 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 kann man die *Initialisierungsphase* von der *eigentlichen Programmausführung* 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 geht man ebenso wie bei der Programmerstellung fix davon aus, 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, geht man an dieser Stelle davon aus, dass die Variable ein Objekt dieses Typs referenziert. Vor dem Ablegen einer Zahl in der Variablen wird ein `Integer`-Objekt erzeugt und beim Auslesen wieder die Zahl extrahiert. Entsprechenden Code 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.

Diese Beispiele zeigen Folgendes: Je früher Entscheidungen getroffen werden, desto weniger ist zur Laufzeit zu tun und desto weniger Programmcode (Fallunterscheidungen) braucht man. Ohne statisch geprüfte Typen wäre mehrfacher Aufwand nötig: Sogar wenn man weiß, dass die Variable eine ganze Zahl enthält, muss der Compiler eine beliebige Referenz annehmen und zur Laufzeit eine dynamische Typprüfung durchführen.

Frühe Entscheidungen haben noch einen anderen Vorteil: Typfehler und gelegentlich auch andere indirekt 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 man effektiver reagieren als auf Fehler, die während der eigentlichen Programmausführung auftreten. Auch bei dynamischer Typprüfung kann man versuchen, mögliche Typfehler schon in der Initialisierungsphase zu erkennen. Dazu gibt es die Möglichkeit, den Typ eines Wertes abzufragen.

Planbarkeit. Letztendlich erleichtern frühe Entscheidungen die Planbarkeit weiterer Schritte. Wenn man weiß, dass eine Variable vom Typ `int` ist, braucht man kaum mehr Überlegungen darüber anzustellen, welche Werte in der Variablen enthalten sein könnten. Statt auf Spekulationen baut man auf Wissen auf. Um sich auf einen Typ festzulegen muss man voraussehen (also planen), wie bestimmte Programmteile im fertigen Pro-

gramm verwendet werden. Man wird zuerst jene Typen festlegen, bei denen man kaum Zweifel an der künftigen Verwendung hat. Frühe Entscheidungen sind daher oft recht 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 entsprechend 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 kann man Ähnliches auch in dynamischen Sprachen erreichen. Sehr häufig findet man in dynamischen Programmen Variablennamen, welche die Art der darin enthaltenen Werte treffend beschreiben. Damit bezweckt man nicht nur bessere Lesbarkeit, sondern möchte auch getroffene Entscheidungen festhalten, auf denen weitere Entscheidungen aufbauen. Das funktioniert gut, solange man sich strikt an entsprechende Konventionen hält. Dazu braucht es viel Programmierdisziplin. Gelegentlich geht die Programmierdisziplin 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 im Programm zu finden. Nicht selten behält man aus Bequemlichkeit bestehende Namen bei, obwohl die durch die Namen gegebenen Informationen nicht mehr stimmen.

1.3.2 Abstraktion

Wie wir in Abschnitt 1.2.3 gesehen haben, stellen Abstraktionen der realen Welt eine in der Praxis wichtige Grundlage für die Ersetzbarkeit und damit für die Wartbarkeit von Software dar. Typen spielen eine entscheidende Rolle im Umgang mit Abstraktionen.

Abstrakte Datentypen. Der Begriff *abstrakter Datentyp* bezieht sich in erster Linie auf die Trennung der Innenansicht von der Außenansicht einer Modularisierungseinheit (Data-Hiding) zusammen mit Kapselung. 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. Implizit steht hinter jeder Modularisierungseinheit ein nicht näher beschriebenes (also abstraktes) Konzept, das im Idealfall eine Analogie in der realen Welt hat. Private und nach außen sichtbare Inhalte zusammen müssen diesem Konzept entsprechen. Solange das Konzept erhalten bleibt, kann man private Inhalte problemlos ändern, ohne dabei die Verwendbarkeit der Modularisierungseinheit zu beeinträchtigen.

Ein abstrakter Datentyp ist im Wesentlichen eine Schnittstelle einer Modularisierungseinheit. Man kann die Signatur der Modularisierungseinheit als Schnittstelle bzw. Typ betrachten. In diesem Fall spricht man von einem *strukturellen Typ,* weil der Typ der Modularisierungseinheit nur von den Namen, Parametertypen und Ergebnistypen der nach außen sichtbaren Inhalte abhängt – quasi von der nach außen sichtbaren Struktur. Wenn unterschiedliche Modularisierungseinheiten dieselbe Signatur haben, dann haben sie auch denselben strukturellen Typ.

Abstraktion bedeutet, dass der Typ einer Modularisierungseinheit gedanklich für ein Konzept aus der realen Welt steht. Strukturelle Typen eignen sich nur beschränkt zur Abstraktion, weil unterschiedliche Modularisierungseinheiten auch zufällig dieselbe Signatur haben können, derselbe Typ also für unterschiedliche Konzepte stehen kann. Mit viel Programmierdisziplin lässt sich dieses Manko beseitigen: Man braucht nur jeder Modularisierungseinheit einen eigentlich nicht verwendeten Inhalt mitzugeben, dessen Name das Konzept dahinter beschreibt. Damit werden gleiche Signaturen für unterschiedliche Konzepte ausgeschlossen. Wegen dieser stets vorhandenen Möglichkeit zur Abstraktion werden in der Theorie überwiegend strukturelle Typen betrachtet.

In der Praxis ist die Abstraktion von überragender Bedeutung. Man denkt beim Programmieren hauptsächlich abstrakt in Konzepten, nur selten an Signaturen. Daher verwenden Programmiersprachen zum Großteil *nominale Typen* statt struktureller Typen. Jeder nominale Typ hat, wie der Begriff schon andeutet, neben einer Signatur auch einen eindeutigen Namen. Beispielsweise entspricht der Typ eines Objekts (in Java, C#, C++, ...) dem Namen der Klasse, von der das Objekt erzeugt wurde. Natürlich haben alle Objekte desselben nominalen Typs auch dieselbe (in der Klasse beschriebene) Signatur, aber entscheidend ist der gemeinsame Klassenname. Wenn zwei Klassen genau dieselbe Signatur beschreiben, so haben sie dennoch unterschiedliche Namen und entsprechen daher unter-

schiedlichen nominalen Typen. Die Konzepte dahinter können sich unterscheiden. Zwei nominale Typen sind äquivalent wenn sie denselben Namen haben, zwei strukturelle Typen wenn sie dieselbe Struktur haben.

Obwohl hinter abstrakten Datentypen oft komplexe Konzepte stecken, können auch elementare Typen wie `int` als abstrakte Datentypen betrachtet werden. Diese Typen abstrahieren über Implementierungsdetails, etwa die genaue Repräsentation in der Maschine.

Untertypen. Untertypbeziehungen werden 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 1.2.3) gibt es also keine Untertypen.

Wie wir in Kapitel 2 sehen werden, sind Untertypbeziehungen durch das Ersetzbarkeitsprinzip für strukturelle Typen ganz eindeutig definiert. Die Theorie lässt keinen Spielraum: Sind zwei strukturelle Typen gegeben, kann man durch Anwendung fixer Regeln automatisch ermitteln, ob ein Typ Untertyp des anderen ist [2].

Für nominale Typen reichen einfache Regeln nicht aus. Abstrakte und daher den Regeln nicht zugängliche Konzepte lassen sich ja nicht automatisch vergleichen. In der Praxis muss man beim Programmieren explizit hinschreiben, welcher Typ Untertyp von welchem anderen ist. Meist verwendet man dazu abgeleitete Klassen: Wird eine Klasse U von einer Klasse T abgeleitet, so nimmt man 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. Man soll U nur dann von T ableiten, wenn das Konzept hinter T durch das Konzept hinter U vollständig ersetzbar ist, sodass statt einem Objekt 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.

So wie in Abschnitt 1.2.3 beschrieben ist es durchaus möglich, Zusicherungen in die Entscheidung von Untertypbeziehungen einzubeziehen. Einige wenige Programmiersprachen wie Eiffel [28] machen das. 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 man Untertypbeziehungen explizit hinschreiben und damit die Verantwortung für die Kompatibilität der Konzepte übernehmen muss. Daher betrachten die meisten Programmiersprachen Zusicherungen nicht als zu den Typen gehörig. Trotzdem empfiehlt es sich, beim Programmieren die wichtigsten Zusicherungen zumindest in Form von Kommentaren hinzuschreiben und dadurch die gröbsten Fehler bei Untertypbeziehungen zu verhindern.

Obwohl die Regeln zur Entscheidung von Untertypbeziehungen struktureller Typen recht einfach sind, implizieren sie doch eine bedeutende Einschränkung, die auch für nominale Typen gilt: Typen von Funktions- bzw. Methoden-Parametern dürfen in Untertypen nicht stärker werden. Enthält ein Obertyp `T` z.B. eine Methode `boolean compare(T x)`, kann sie im Untertyp `U` nicht durch `boolean compare(U x)` überschrieben sein. Derartiges wird in der Praxis häufig benötigt. Aber es gibt keine Möglichkeit, entsprechende Typen statisch zu prüfen. Dynamische Prüfungen sind natürlich möglich. Aus demselben Grund sind auch Wertebereichseinschränkungen im Allgemeinen nicht statisch prüfbar; das bedeutet z.B., dass `int` nicht als Untertyp von `long` betrachtet werden kann, obwohl jede Zahl in `int` auch in `long` vorkommt.

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

Einfache Generizität ist leicht zu verstehen und auch vom Compiler leicht handzuhaben. Die Komplexität steigt jedoch rasch an, wenn man Einschränkungen auf Typparametern berücksichtigt, die vor allem (aber nicht nur) in der objektorientierten Programmierung benötigt werden. 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 und C# eingesetzt. *Higher-Order-Subtyping,* auch *Matching* genannt [1], geht einen eher direkten Weg und beschreibt Einschränkungen über Untertyp-ähnliche Beziehungen, die wegen Unterschieden in Details aber keine Untertypbeziehungen

sind. 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 statischen objektorientierten Sprachen Generizität. Jedoch ist Generizität hinischtlich der Wartung prinzipiell kein vollwertiger Ersatz für Ersetzbarkeit.

Wie wir in Kapitel 3 sehen werden gibt es auch mehrere Möglichkeiten, wie der Compiler generische Programme in nichtgenerische umformt. Es gibt Unterschiede im Umfang und Spezialisierungsgrad des generierten Codes, in der im generischen Programm verfügbaren Information und im Zeitpunkt, zu dem diese Information verwendbar ist.

1.3.3 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; in den Kapiteln 2 und 3 werden wir uns näher damit beschäftigen. Im Folgenden betrachten wir kurz einige weitere Bereiche, teilweise mit für Uneingeweihte überraschenden Möglichkeiten und Grenzen.

Rekursive Datenstrukturen. Viele Datenstrukturen wie Listen und Bäume sind in ihrer Größe unbeschränkt. Nicht selten spricht man von potentiell unendlichen Strukturen. Aber tatsächlich unendlich große Strukturen sind im Computer niemals darstellbar. Wenn man in Typen versucht, rekursive Datenstrukturen als möglicherweise unendlich zu betrachten, wird man scheitern, weil Typprüfungen in eine Endlosschleife geraten.

Der einzig sinnvolle Weg zur Beschreibung unbeschränkter Strukturen führt über eine induktive Konstruktion: Man beginnt mit einer endlichen Menge M_0, die nur einfache Werte enthält, z.B. $M_0 = \{\texttt{end}\}$ wenn man nur einen einzigen Wert `end` braucht. Dann beschreibt man, wie man über endlich viele Möglichkeiten aus einer Menge M_i die Menge M_{i+1} $(i \geq 0)$ generieren kann, wobei M_{i+1} zumindest alle Elemente von M_i enthält, z.B. $M_{i+1} = M_i \cup \{\texttt{elem}(n, x) \mid n \in \text{Int}; x \in M_i\}$. Die meist unendliche Menge $M = \bigcup_{i=0}^{\infty} M_i$, also die Vereinigung aller M_i, enthält dann alle beschriebenen Strukturen. 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 werden

Typen von Datenstrukturen tatsächlich auf diese Weise konstruiert. Der Typ `Lst`, dessen Werte die Elemente der oben beschriebenen Menge sind, wird in Haskell so definiert (wobei „`|`“ Alternativen trennt):

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

Aus der Syntax geht auf den ersten Blick hervor, dass `Lst` ein rekursiver Typ ist, weil `Lst` 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: Man kann z.B. Listen konstruieren, in denen alle Listenelemente vom selben Typ sind oder in denen sich die Typen der Listenelemente zyklisch wiederholen, aber man kann keine Listen konstruieren, in denen alle Listenelemente unterschiedliche Typen haben. Die Art der Konstruktion ist nicht willkürlich, sondern unterscheidet handhabbare Strukturen von solchen, in denen Typen nicht in endlicher Zeit prüfbar sind. In der Praxis braucht man sicher nicht mehr als derart konstruierbar ist.

Man muss klar zwischen M_0 (nicht-rekursiv) und der Konstruktion aller M_i mit $i > 0$ (rekursiv) unterscheiden, wobei M_0 nicht leer sein darf. Diese Eigenschaft nennt man *Fundiertheit.* In Haskell muss es in jeder Typdefinition zumindest eine nicht-rekursive Alternative (z.B. `end` in `Lst`) 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`, und Klassen beschreiben die M_i mit $i > 0$. Da statt einem Objekt immer auch `null` verwendet werden kann, ist Fundiertheit immer gegeben. Der Nachteil dieser Vereinfachung liegt auf der Hand: Man muss beim Programmieren stets damit rechnen, statt einem Objekt nur `null` zu erhalten, auch wenn man es gar nicht mit rekursiven Datenstrukturen zu tun hat.

Typinferenz. Die Techniken hinter der Typinferenz sind heute weitgehend ausgereift, und für bestimmte Aufgaben wird Typinferenz sehr breit eingesetzt, auch in Java, C# und C++. Ein technisches Problem ist bis heute jedoch ungelöst: Typinferenz funktioniert nicht, wenn gleichzeitig (also an derselben Stelle im Programm) 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 sehr lokal und für wenige Sprachkonstrukte in objektorientierten Sprachen.

Im Gegensatz zu Untertypen verträgt sich Generizität sehr gut mit Typinferenz. Das gilt auch für Einschränkungen auf Typparametern, trotz großer Ähnlichkeiten der Einschränkungen zu Untertypen.

Typinferenz erspart das Anschreiben von Typen, ändert aber nichts an der durch statische Typprüfungen reduzierten Flexibilität. Fehlende Typangaben dienen auch nicht der Dokumentation. Aber gerade im lokalen Bereich vermeidet Typinferenz das mehrfache Hinschreiben desselben 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 formalen 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 Wertes 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 man z.B. weiß, 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 formalen Parameter. Erst in jüngster Zeit beginnt man, das Propagieren beliebiger solcher Information in Programmiersprachen zu unterstützen.

Typen werden in der Regel als unveränderlich angesehen: Eine Variable, deren deklarierter Typ `int` ist, enthält immer eine entsprechende ganze Zahl, auch nachdem man der Variablen einen neuen Wert zugewiesen hat. Auch beliebige Information kann man so handhaben: 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. Man muss irgendwie festlegen, welche Werte die gewünschten Eigenschaften besitzen. Z.B. 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 möchte man 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. Mit solcher Information kann man umgehen, indem man das Propagieren kontrolliert: Man kann die Information zwar weitergeben, etwa vom Argument zum formalen Parameter, aber nicht duplizieren; das Argument hätte die Information durch die Weitergabe an den Parameter verloren. Derartiges funktioniert gut, wenn man Typen nicht als unveränderlich, sondern als zustandsbehaftet ansieht; je nach Zustand des Typs ist die Information vorhanden oder nicht. Solche Typen gibt es heute in experimentellen Sprachen. Diese Typen sind sehr mächtig, weil sie auch Eigenschaften ausdrücken können, die für die Synchronisation nebenläufiger Programme benötigt werden (z.B. Prozesstypen).

Zusammenfassend kann man sagen, dass man mit statisch geprüften Typen heute viel realisieren kann, was noch vor wenigen Jahren als gänzlich unmöglich erschien. Umgekehrt kennt man aber auch unerwartete Einschränkungen bei statischen Typprüfungen, die es notwendig machen, manche Aspekte erst zur Laufzeit zu prüfen.

1.4 Objektorientierte Programmierung

Wenden wir uns nun verstärkt der objektorientierten Programmierung zu. Der Inhalt dieses Abschnitts sollte großteils schon aus anderen Lehrveranstaltungen bekannt sein. Hier wird also eine kurze Wiederholung gegeben, vor allem um die verwendete Terminologie wieder in Erinnerung zu rufen.

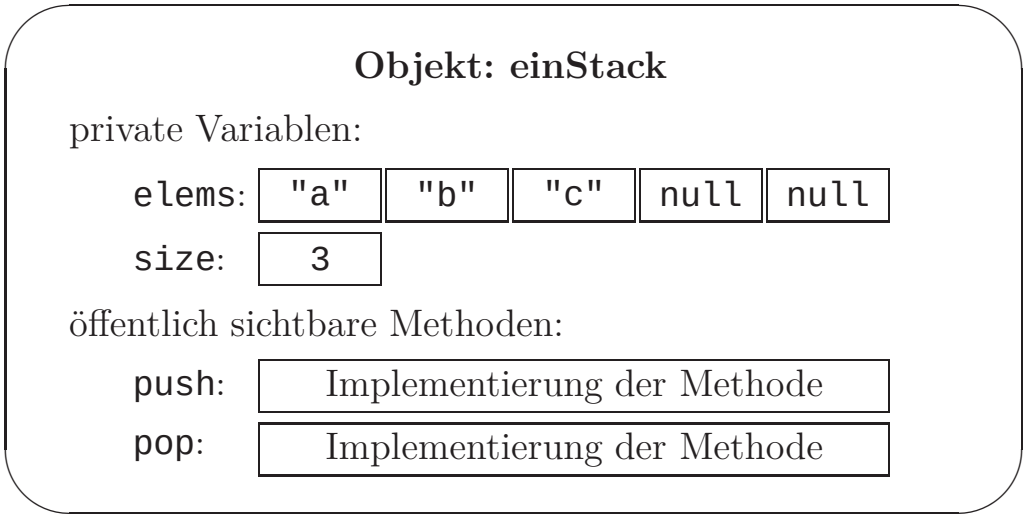
1.4.1 Basiskonzepte

Die objektorientierte Programmierung will Softwareentwicklungsprozesse unterstützen, die auf inkrementelle Verfeinerung aufbauen. Gerade bei diesen Entwicklungsprozessen spielt die leichte Wartbarkeit eine große Rolle. Im Wesentlichen geben objektorientierte Sprachen Entwickler(inne)n Werkzeuge in die Hand, die sie zum Schreiben leicht wiederverwendbarer und änderbarer Software brauchen.

Objekt. Ein Objekt ist eine grundlegende Modularisierungseinheit in der Ausführung eines Programms – siehe Abschnitt 1.2.1. Zur Laufzeit besteht

die Software aus einer Menge von Objekten, die einander teilweise kennen und untereinander *Nachrichten* (*Messages*) austauschen.

Die folgende Abbildung veranschaulicht 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. Zurzeit sind drei Einträge vorhanden.

Das Zusammenfügen von Daten und Methoden zu einer Einheit nennt man *Kapselung* (*Encapsulation*). Daten und Methoden im Objekt sind untrennbar miteinander verbunden: Die Methoden benötigen die Daten zur Erfüllung ihrer Aufgaben, und die genaue Bedeutung der Daten ist nur den Methoden des Objekts bekannt. Methoden und Daten sollen zueinander in einer engen logischen Beziehung stehen. Eine gut durchdachte Kapselung ist ein wichtiges Qualitätsmerkmal.

Jedes Objekt besitzt folgende Eigenschaften [35]:

Identität (Identity): Ein Objekt ist durch seine unveränderliche Identität eindeutig gekennzeichnet. Über seine Identität kann man das Objekt ansprechen, ihm also eine Nachricht schicken. Vereinfacht kann man sich die Identität als die Adresse des Objekts im Speicher vorstellen. Dies ist aber nur eine Vereinfachung, da die Identität erhalten bleibt, wenn sich die Adresse ändert – zum Beispiel beim Verschieben des Objekts bei der Garbage-Collection oder beim Auslagern in eine Datenbank. Jedenfalls gilt: Gleichzeitig durch zwei Namen bezeichnete Objekte sind *identisch* (*identical*) wenn sie am selben Speicherplatz liegen, es sich also um nur ein Objekt mit zwei Namen handelt.

Zustand (State): Der Zustand setzt sich aus den Werten der Variablen im Objekt zusammen. Er ist meist änderbar. In obigem Beispiel ändert sich der Zustand durch Zuweisung neuer Werte an `elems` und `size`. Zwei Objekte sind *gleich* (*equal*) wenn sie denselben Zustand und dasselbe Verhalten haben. Objekte können auch gleich sein, wenn sie nicht identisch sind; 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 verwendet man viele Varianten des Begriffs Gleichheit. Oft werden nur manche – nicht alle – Variableninhalte in den Vergleich einbezogen. Gelegentlich betrachtet man nur identische Objekte als gleich.

Verhalten (Behavior): Das Verhalten eines 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 und die Deklarationen der Objektvariablen 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 das Verhalten nicht jedes Detail widerspiegelt. Statt der Implementierung haben wir beim Programmieren nur eine abstrakte Vorstellung im Kopf, die viele Details offen lässt. Beispielsweise gibt die Methodenbeschreibung

> `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 Abstraktionen ganz selbstverständlich und viel 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, häufig ergeben sich durch Verwendung nominaler Typen benannte Abstraktionen – siehe Abschnitt 1.3.2. Über Zusicherungen kann man das Verhalten beliebig genau beschreiben. 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 braucht man nur eine Schnittstelle kennen, nicht aber 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 sieht man 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. Man nennt 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.

Anmerkung: Man sagt auch, 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 der „Schnittstelle“ wenn wir eine beliebige meinen.

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

In einer objektorientierten Sprache mit Klassen schreibt man beim Programmieren hauptsächlich Klassen. Objekte werden nur zur Laufzeit durch Verwendung von Konstruktoren oder durch Clonen erzeugt.

Ein Stack könnte in Java 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 if
    // not empty; otherwise pop returns null
    public String pop() {
        if (size > 0)
            return elems[--size];
        return null;
    }
}
```

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 Kommentare verlassen und nicht die Implementierung betrachten müssen. Außerdem können Details der Implementierung problemlos nachträglich geändert werden, solange die unveränderten Kommentare die geänderte Implementierung treffend beschreiben.

1.4.2 Polymorphismus

Das Wort *polymorph* kommt aus dem Griechischen und heißt „vielgestaltig“. Im Zusammenhang mit Programmiersprachen spricht man von *Polymorphismus,* wenn eine Variable oder Methode gleichzeitig mehrere Typen haben kann. Ein formaler Parameter einer polymorphen Methode kann an Argumente von mehr als nur einem Typ gebunden werden. Objektorientierte Sprachen sind polymorph. Im Gegensatz dazu sind konventionelle statisch typisierte Sprachen wie C und Pascal *monomorph*: Jede Variable oder Funktion hat einen eindeutigen Typ.

Man kann verschiedene Arten des Polymorphismus unterscheiden [8]:

- Polymorphismus
  - universeller Polymorphismus
    - Generizität
    - Untertypen
  - 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 erzielt man Gleichförmigkeit durch gemeinsame Schnittstellen für unterschiedliche Objekte. Überladene Methoden müssen, abgesehen vom gemeinsamen Namen, keinerlei strukturelle Ähnlichkeiten besitzen und sind daher ad-hoc-polymorph. Auch bei Typumwandlungen, insbesondere bei Casts auf elementaren Typen wie von `int` auf `double`, gibt es keine gemeinsame Struktur; `int` und `double` sind intern ja ganz unterschiedlich dargestellt. Casts auf Referenztypen fallen dagegen eher in die Kategorie Untertypen. Wir werden uns in Kapitel 2 mit Untertypen und in Kapitel 3 mit den anderen Arten des Polymorphismus näher beschäftigen.

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

In einer objektorientierten Sprache hat eine Variable (oder ein formaler Parameter) gleichzeitig folgende Typen:

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

Dynamischer Typ: Das ist der *spezifischste Typ,* den der in der Variable 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 setzt man unter anderem für dynamisches Binden ein.

Statischer Typ: Dieser Typ wird vom Compiler (statisch) ermittelt und liegt irgendwo zwischen deklariertem und dynamischem Typ. In vielen Fällen ordnet der Compiler derselben Variablen 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 daher nicht vor.

Eine Konsequenz aus der Verwendung von Untertypen ist *dynamisches Binden:* Wenn man eine Nachricht an ein Objekt schickt, kennt der Compiler nur den statischen und deklarierten Typ des Empfängers der Nachricht. Die Methode soll aber entsprechend dem dynamischen Typ 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 man sich zur Laufzeit die Bestimmung der passenden Methode erspart. Durch mögliche Optimierungen ist statisches Binden etwas effizienter als dynamisches. Dennoch ist 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 Programmierer(inne)n Schreibaufwand. Außerdem werden einige Programmänderungen vereinfacht, da sich Änderungen von Klassen auf alle davon abgeleiteten Klassen auswirken.

In populären objektorientierten Programmiersprachen können bei der Vererbung Unterklassen im Vergleich zu Oberklassen aber 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 durch neue Methoden überschrieben, die jene in der Oberklasse ersetzen. Meist gibt es eine Möglichkeit, von der Unterklasse aus auf überschriebene Methoden der Oberklasse zuzugreifen.

Diese beiden Änderungsmöglichkeiten sind beliebig kombinierbar.

In Sprachen wie Java, C# und C++ 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, wie oben beschrieben, 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 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. 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 findet man zum Beispiel in C++. Neben Klassen gibt es in Java *Interfaces*, auf denen es auch Mehrfachvererbung gibt. Solche Interfaces sind Schnittstellen, denen andere Eigenschaften von Klassen fehlen.

Wir verwenden in diesem Skriptum für das Sprachkonstrukt in Java den englischen Begriff, während wir mit dem gleichbedeutenden deutschen Begriff Schnittstellen im Allgemeinen bezeichnen.

1.4.3 Vorgehensweisen in der Programmierung

Faktorisierung. Zusammengehörige Eigenschaften und Aspekte des Programms sollen zu Einheiten zusammengefasst werden. In Analogie zur Zerlegung eines Polynoms in seine Koeffizienten nennt man die Zerlegung eines Programms in Einheiten mit zusammengehörigen Eigenschaften *Faktorisierung* (*Factoring*). Wenn zum Beispiel mehrere Stellen in einem Programm aus denselben Sequenzen von Befehlen bestehen, soll man diese Stellen durch Aufrufe einer Methode ersetzen, die genau diese Befehle ausführt. Gute Faktorisierung führt dazu, dass zur Änderung aller dieser Stellen auf die gleiche Art und Weise eine einzige Änderung der Methode ausreicht. Bei schlechter Faktorisierung hätten alle Programmstellen gefunden und einzeln geändert werden müssen. Gute Faktorisierung verbessert auch die Lesbarkeit des Programms, beispielsweise dadurch, dass die Methode einen Namen bekommt, der ihre Bedeutung widerspiegelt.

Objekte dienen durch Kapselung zusammengehöriger Eigenschaften in erster Linie der Faktorisierung. Durch Zusammenfügen von Daten mit Methoden hat man mehr Freiheiten zur Faktorisierung als in anderen Paradigmen, bei denen Daten und Funktionen klar getrennt sind.

> **Faustregel:** Gute Faktorisierung kann die Wartbarkeit eines Programms wesentlich erhöhen.

Die objektorientierte Programmierung bietet viele Möglichkeiten zur Faktorisierung und erleichtert damit das Finden einer guten Zerlegung. Bei Weitem nicht jede Faktorisierung ist gut. Es ist unsere Aufgabe beim Programmieren die Qualität von Faktorisierungen zu beurteilen.

Die Lesbarkeit eines Programms kann man erhöhen, indem man es so in Objekte zerlegt, wie es der Erfahrung in der realen Welt entspricht. Das heißt, Software-Objekte sollen die reale Welt simulieren, soweit dies zur Erfüllung der Aufgaben sinnvoll erscheint. Namen für Software-Objekte sollen den üblichen Bezeichnungen realer Objekte entsprechen. Dadurch ist das Programm einfacher lesbar, vorausgesetzt alle Personen haben annähernd dieselben Vorstellungen über die reale Welt. Man darf die Simulation aber nicht zu weit treiben. Vor allem soll man keine Eigenschaften

der realen Welt simulieren, die für die entwickelte Software bedeutungslos sind. Die Einfachheit ist wichtiger.

> **Faustregel:** Man soll die reale Welt simulieren, aber nur so weit, dass die Komplexität dadurch nicht erhöht wird.

Entwicklungsprozesse. Die Qualität eines Programms wird von teilweise widersprüchlichen Faktoren bestimmt. Sie machen es schwer, qualitativ hochwertige Programme zu schreiben. Einflussgrößen sind zu Beginn der Entwicklung nicht bekannt oder nicht kontrollierbar. Zum Beispiel weiß man oft nicht genau, welche Eigenschaften des Programms gebraucht werden. Erfahrungen kann man erst sammeln, wenn das Programm existiert.

Traditonell wird Software nach dem *Wasserfallmodell* entwickelt, in dem die üblichen Entwicklungsschritte (Analyse, Design, Implementierung, Verifikation und Validierung) 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 verwendet man für größere Projekte eher *zyklische Prozesse,* bei denen man die einzelnen Schritte ständig auf jeweils einem anderen Teil der gesamten Aufgabe wiederholt. So erhält man schon recht früh Feedback und kann auf geänderte Anforderungen reagieren. 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.

> **Faustregel:** Zyklische Prozesse verkraften Anforderungsänderungen besser, aber Zeit und Kosten sind schwer planbar.

Erfahrung. In den vergangenen Jahrzehnten hat sich in der Programmierung ein umfangreicher Erfahrungsschatz darüber entwickelt, mit welchen Problemen man in Zukunft rechnen muss, wenn man eine Aufgabe auf eine bestimmte Art löst. Gute Programmierer(innen) setzen diese Erfahrungen gezielt ein. Eine Garantie für den Erfolg eines Projekts gibt es natürlich trotzdem nicht, aber die Wahrscheinlichkeit dafür steigt.

Gerade in der objektorientierten Programmierung ist der gezielte Einsatz von Erfahrungen essenziell. Objektorientierte Sprachen bieten viele

unterschiedliche Möglichkeiten zur Lösung von Aufgaben. Jede Möglichkeit hat andere Eigenschaften. Mit ausreichend Erfahrung wird man jene Möglichkeit wählen, deren Eigenschaften später am ehesten hilfreich sind. Mit wenig Erfahrung wählt man einfach nur die Möglichkeit, die man zuerst entdeckt. Damit verzichtet man auf einen wichtigen Vorteil. Generell kann man sagen, dass die objektorientierte Programmierung durch erfahrene Leute derzeit wahrscheinlich das erfolgversprechendste Programmierparadigma darstellt, andererseits aber Gelegenheitsprogrammierer(innen) und noch unerfahrene Softwareentwickler(innen) oft überfordert.

Zusammenhalt und Kopplung. Bei der Softwareentwicklung müssen wir in jeder Phase wissen, wie wir vorgehen müssen um möglichst hochwertige Software zu produzieren. Vor allem eine gute Faktorisierung ist ein entscheidendes Kriterium, aber leider erst gegen Ende der Entwicklung beurteilbar. Daher gibt es 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 Entwickler(innen) dieser Klasse zuständig.

Klassen-Zusammenhalt (Class-Cohesion): Darunter versteht man den *Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse.* Dieser Grad ist zwar nicht einfach messbar, oft aber intuitiv einfach fassbar. Der Zusammenhalt ist hoch, wenn alle Variablen und Methoden eng zusammenarbeiten und durch den Namen der Klasse gut beschrieben sind. Das heißt, einer Klasse mit hohem Zusammenhalt fehlt etwas Wichtiges, wenn man beliebige Variablen oder Methoden entfernt. Außerdem wird der Zusammenhalt niedriger, wenn man die Klasse sinnändernd umbenennt.

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.

Faustregel: 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.

Faustregel: Die Objekt-Kopplung soll schwach sein.

Schwache Kopplung deutet auf gute Kapselung hin, bei der Objekte möglichst unabhängig voneinander sind. Dadurch beeinflussen Programmänderungen wahrscheinlich weniger Objekte unnötig. Beeinflussungen durch unvermeidbare Abhängigkeiten zwischen Objekten sind unumgänglich.

Klassen-Zusammenhalt und Objekt-Kopplung stehen in einer engen Beziehung zueinander. Wenn der Klassen-Zusammenhalt hoch ist, dann ist oft die Objekt-Kopplung schwach und umgekehrt. Da Menschen auch dann sehr 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 Entwickler(innen) sich zwischen mehreren Alternativen zu entscheiden haben, können Klassen-Zusammenhalt und Objekt-Kopplung der einzelnen Alternativen einen wichtigen Beitrag zur Entscheidungsfindung liefern. Erwartete Klassen-Zusammenhalte und Objekt-Kopplungen der Alternativen lassen sich im direkten Vergleich einigermaßen sicher prognostizieren. Klassen-Zusammenhalt und Objekt-Kopplung sind Faktoren in der Bewertung von Alternativen. In manchen Fällen können jedoch andere Faktoren ausschlaggebend sein.

Auch mit viel Erfahrung wird man kaum auf Anhieb einen optimalen Entwurf liefern können. Normalerweise muss die Faktorisierung einige Male geändert werden; man spricht 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 zu werden brauchen. Man muss nicht von Anfang an einen optimalen Entwurf haben, sondern nur alle nötigen Refaktorisierungen durchführen, bevor sich Probleme über das ganze Programm ausbreiten. Natürlich darf man nicht so häufig refaktorisieren, dass bei der inhaltlichen Programmentwicklung kein Fortschritt mehr erkennbar ist.

> **Faustregel:** Ein vernünftiges Maß rechtzeitiger Refaktorisierungen führt häufig zu gut faktorisierten Programmen.

1.4.4 Wiederverwendung und Paradigmenwahl

Es ist sinnvoll, bewährte Software so oft wie möglich wiederzuverwenden. Das spart Entwicklungsaufwand. Wir müssen zwischen verschiedenen Arten der Wiederverwendung unterscheiden:

Programme: Die meisten Programme werden im Hinblick darauf entwickelt, dass sie häufig (wieder)verwendet werden. Dadurch zahlt es sich erst aus, einen großen Aufwand in die Entwicklung zu stecken.

Daten: Auch Daten in Datenbanken und Dateien werden in vielen Fällen häufig wiederverwendet. Nicht selten haben Daten eine längere Lebensdauer als die Programme, die sie benötigen oder manipulieren.

Erfahrungen: Häufig unterschätzt wird die Wiederverwendung von Konzepten und Ideen in Form von Erfahrungen. Diese können zwischen sehr unterschiedlichen Projekten ausgetauscht werden.

Code: Viele Konzepte von Programmiersprachen, wie zum Beispiel Untertypen, Vererbung und Generizität, wurden im Hinblick auf die Wiederverwendung von Code entwickelt. Man kann mehrere Arten der Codewiederverwendung unterscheiden:

Bibliotheken: Einige Klassen in Klassenbibliotheken werden sehr häufig (wieder)verwendet. Allerdings kommen nur wenige, relativ einfache Klassen für die Aufnahme in Bibliotheken in Frage. Komplexere Klassen sind dafür meist zu stark projektbezogen.

Projektinterne Wiederverwendung: Hochspezialisierte Programmteile sind nur innerhalb eines Projekts in unterschiedlichen Programmversionen wiederverwendbar. Wegen der Komplexität erspart bereits eine einzige Wiederverwendung viel Arbeit.

Programminterne Wiederverwendung: Code in einem Programm kann zu unterschiedlichen Zwecken oft wiederholt ausgeführt werden. Durch den Einsatz eines Programmteils in mehreren Aufgaben wird das Programm einfacher und leichter wartbar.

Die Erfahrung zeigt, dass durch objektorientierte Programmierung tatsächlich Code-Wiederverwendung erzielbar ist. Kosteneinsparungen ergeben sich aber normalerweise nur

- mit ausreichend Erfahrung um die Möglichkeiten gut zu nutzen
- und wenn viel Zeit in die Wiederverwendbarkeit investiert wird.

Mit wenig Erfahrung investiert man oft zu wenig, zu viel oder an falscher Stelle in die Wiederverwendbarkeit. Fehlentscheidungen rächen sich durch lange Entwicklungszeiten oder sogar das Scheitern eines Projekts. Im Zweifel soll man anfangs eher weniger in die Wiederverwendbarkeit investieren, diese Investitionen aber nachholen, sobald sich ein Bedarf ergibt.

> **Faustregel:** Code-Wiederverwendung erfordert beträchtliche Investitionen in die Wiederverwendbarkeit. Man soll diese tätigen, wenn ein tatsächlicher Bedarf absehbar ist.

Die erhoffte Wiederverwendung von Code ist wahrscheinlich der wichtigste Grund für den Erfolg der objektorientierten Programmierung. Vor allem Untertypen und Ersetzbarkeit in den Händen erfahrener Entwickler(innen) sind ausschlaggebend.

In vielen Programmierparadigmen stehen Algorithmen zentral im Mittelpunkt, etwa in der prozeduralen, funktionalen und logikorientierten Programmierung. Für die objektorientierte Programmierung gilt das nicht. Hier steht die Datenabstraktion im Mittelpunkt, aber Algorithmen müssen unter Umständen aufwendig auf mehrere Objekte verteilt werden. Das

kann den Entwicklungsaufwand von Algorithmen erhöhen und deren Verständlichkeit verringern. Diese Unterschiede haben einen Einfluss darauf, wofür die Paradigmen gut geeignet sind.

> **Faustregel:** Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.

Wenn die Wiederverwendung und Wartbarkeit von untergeordneter Bedeutung ist, beispielsweise weil man die Software nur einmal einsetzt, kann die objektorientierte Programmierung ihre Vorteile nicht ausspielen. Diese kommen erst durch einen langen Einsatz- und Wartungszeitraum zum Tragen. In anderen Paradigmen kann man wesentlich rascher programmieren, da man sich nicht um die Wiederverwendbarkeit kümmern muss. Soll Software aber über einen längeren Zeitraum gewartet werden, zahlt es sich aus, durch objektorientierte Programmierung etwas mehr in die Entwicklung zu investieren. Sonst zahlt es sich nicht aus.

1.5 Wiederholungsfragen

Folgende Fragen sollen beim Erarbeiten des Stoffes helfen. Sie stellen keine vollständige Aufzählung möglicher Prüfungsfragen dar.

1. Was versteht man unter einem Programmierparadigma?
2. Wozu dient ein Berechnungsmodell?
3. Welche Berechnungsmodelle werden in Programmierparadigmen verwendet, und welche charakteristischen Eigenschaften haben sie?
4. Welche Eigenschaften von Berechnungsmodellen sind für deren Erfolg häufig (mit)bestimmend?
5. Im Spannungsfeld welcher widersprüchlichen Ziele befinden sich Programmierparadigmen? Wie äußert sich dieses Spannungsfeld?
6. Was ist die strukturierte Programmierung? Wozu dient sie?
7. Wie gehen unterschiedliche Paradigmen mit Seiteneffekten um?

8. Was bedeutet referentielle Transparenz, und wo findet man referentielle Transparenz?

9. Wieso passt referentielle Transparenz nicht gut mit Ein- und Ausgabe zusammen, und wie kann man das Dilemma lösen?

10. Welchen Zusammenhang gibt es zwischen Seiteneffekten und der objektorientierten Programmierung?

11. Was sind First-Class-Entities? Welche Gründe sprechen für deren Verwendung, welche dagegen?

12. Was haben Funktionen höherer Ordnung mit einem applikativen Programmierstil zu tun?

13. Welche Modularisierungseinheiten gibt es, was sind ihre charakteristischen Eigenschaften, und wodurch unterscheiden sie sich?

14. Welche Bedeutung haben Schnittstellen für Modularisierungseinheiten? Warum unterscheidet man zwischen von außen zugreifbaren und privaten Inhalten?

15. Was ist und wozu dient ein Namensraum?

16. Warum können Module nicht zyklisch voneinander abhängen, Komponenten aber schon?

17. Was versteht man unter Datenabstraktion, Kapselung und Data-Hiding?

18. Warum und inwiefern ist die Einbindung von Komponenten komplizierter als die von Modulen?

19. Wie kann man globale Namen verwalten und damit Namenskonflikte verhindern?

20. Was versteht man unter Parametrisierung? Wann kann das Befüllen von „Löchern“ durch welche Techniken erfolgen?

21. Warum braucht man zur Parametrisierung in der Objekterzeugung neben Konstruktoren gelegentlich auch Initialisierungsmethoden?

22. Welche Vor- und Nachteile hat die zentrale Ablage von Werten zum Zweck der Parametrisierung?

23. Was unterscheidet Generizität von den verschiedenen Formen der Parametrisierung zur Laufzeit?

24. Was sind Annotationen und wozu kann man sie verwenden? Wodurch unterscheiden sie sich von Generizität?

25. Was versteht man unter aspektorientierter Programmierung?

26. Wodurch unterscheiden sich die verschiedenen Formen der Parametrisierung von der Ersetzbarkeit, und warum ist die Ersetzbarkeit in der objektorientierten Programmierung von so zentraler Bedeutung?

27. Wann ist A durch B ersetzbar?

28. Wodurch kann festgelegt sein, ob A durch B ersetzbar ist?

29. Was ist die Signatur einer Modularisierungseinheit?

30. Wie verhält sich die Signatur einer Modularisierungseinheit zur Abstraktion, die durch diese Modularisierungseinheit gebildet wird?

31. Was sind Zusicherungen, und welche Rolle spielen sie für Modularisierungseinheiten?

32. Wann sind Typen miteinander konsistent, und was sind Typfehler?

33. Wie schränken Typen die Flexibilität ein, und warum verwendet man Typen trotzdem?

34. Welche Gründe sprechen für den Einsatz statischer Typprüfungen, welche dagegen?

35. Was versteht man unter Typinferenz? Welche Gründe sprechen für bzw. gegen deren Einsatz?

36. Zu welchen Zeitpunkten können Entscheidungen getroffen werden (Typen und Entscheidungsprozesse)?

37. Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?

38. Wie beeinflussen Typen die Planbarkeit weiterer Schritte?

39. Was ist ein abstrakter Datentyp?

40. Was unterscheidet strukturelle von nominalen Typen?

41. Warum verwenden wir in Programmiersprachen meist nominale Typen, in theoretischen Modellen aber hauptsächlich strukturelle?

42. Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?

43. Warum kann ein Compiler ohne Unterstützung durch Programmierer(innen) nicht entscheiden, ob ein nominaler Typ Untertyp eines anderen nominalen Typs ist?

44. Erklären Sie Einschränkungen bei Untertypbeziehungen zusammen mit statischer Typprüfung.

45. In welchem Zusammenhang verwendet man Higher-Order-Subtyping und F-gebundene Generizität?

46. Wie konstruiert man rekursive Datenstrukturen?

47. Was versteht man unter Fundiertheit rekursiver Datenstrukturen? Welche Ansätze dazu kann man unterscheiden?

48. Warum wird Typinferenz in objektorientierten Sprachen meist nur lokal beschränkt eingesetzt?

49. Wie können statisch geprüfte Typen beliebige Eigenschaften von Werten propagieren?

50. Erklären Sie folgende Begriffe:
    - Objekt, Klasse, Vererbung
    - Identität, Zustand, Verhalten, Schnittstelle
    - deklarierter, statischer und dynamischer Typ
    - Faktorisierung, Refaktorisierung
    - Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung

51. Welche Arten von Polymorphismus unterscheidet man? Welche davon sind in der objektorientierten Programmierung wichtig? Warum?

52. Wann sind zwei gleiche Objekte identisch und wann sind zwei identische Objekte gleich?

53. Sind Datenabstraktion, Datenkapselung und Data-Hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?

54. Was besagt das Ersetzbarkeitsprinzip? (Häufige Prüfungsfrage!)

55. Warum ist Ersetzbarkeit in der objektorientierten Programmierung so wichtig (mehrere Gründe)?

56. Wann und warum ist gute Wartbarkeit wichtig?

57. Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung? Welche Vorteile kann man sich davon erwarten, dass diese Faustregeln erfüllt sind?

58. Welche Arten von Software kann man wiederverwenden, und welche Rolle spielt jede davon in der Softwareentwicklung?

59. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?

60. Wofür ist die objektorientierte Programmierung gut geeignet, und wofür ist sie nicht gut geeignet?

2 Untertypen und Vererbung

In Abschnitt 2.1 untersuchen wir Grundlagen von Untertypbeziehungen. In Abschnitt 2.2 gehen wir auf wichtige Aspekte des Objektverhaltens ein, die man bei der Verwendung von Untertypen beachten muss. Danach betrachten wir in Abschnitt 2.3 die Vererbung im Zusammenhang mit direkter Codewiederverwendung. Schließlich beleuchten wir in Abschnitt 2.4 einige Details zu Klassen und Interfaces in Java.

2.1 Das 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.

Per Definition ist ein Objekt eines Untertyps überall verwendbar, wo ein Objekt eines Obertyps erwartet wird. Insbesondere benötigt man das Ersetzbarkeitsprinzip für

- den Aufruf einer Methode mit einem Argument, dessen Typ ein Untertyp des Typs des entsprechenden formalen Parameters ist
- und für die Zuweisung eines Objekts an eine Variable, wobei der Typ des Objekts ein Untertyp des deklarierten Typs der Variable ist.

Beide Fälle kommen in der objektorientierten Programmierung häufig vor.

2.1.1 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. Aber 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 Abschnitt 1.3.2.

Man kann Untertypbeziehungen auch ohne Vererbung nur auf der Basis struktureller Typen definieren. Das macht man hauptsächlich in der Theorie, um formal schwer nachvollziehbare, auf Intuition beruhende Abstraktionen unberücksichtigt lassen zu können. Manche Sprachen wie Modula-3 bieten neben nominalen auch strukturelle Typen an. Dynamische Sprachen wie Objective-C, Smalltalk [13] und Ruby betrachten Klassen als getrennt von Typen, sodass sich strukturelle Typen ergeben. Der Umgang damit wird seit Kurzem häufig als „Duck-Typing“ bezeichnet.

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 spezielle 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.

Begründung: Auf eine Konstante kann außerhalb des Objekts nur lesend zugegriffen werden. Wenn man die Konstante in einem Objekt vom Typ T liest, erwartet man sich, dass man ein Ergebnis vom Typ A erhält. Diese Erwartung soll auch erfüllt sein, wenn das Objekt vom Typ U ist, wenn also ein Objekt von U verwendet wird, wo ein Objekt von T erwartet wird. Aufgrund der Bedingung gibt es im Objekt vom Typ U eine entsprechende Konstante vom Typ B. Da B ein Untertyp von A sein muss, ist die Erwartung immer erfüllt.

- Für jede 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. Wenn man eine Variable eines Objekts vom Typ T außerhalb des Objekts schreibt, erwartet man sich, dass man der Variablen jedes Objekt vom Typ A zuweisen 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. Wenn man lesende und schreibende Zugriffe gemeinsam betrachtet, 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 formalen Parameter beider Methoden gleich ist
  - und der deklarierte Typ jeden formalen Parameters in U ein Obertyp des deklarierten Typs des entsprechenden formalen Parameters in T ist (gilt nur für Eingangsparameter).

  Begründung: Für die Ergebnistypen der Methoden gilt dasselbe wie für Typen von Konstanten beziehungsweise lesende Zugriffe auf Variablen: Der Aufrufer einer Methode möchte ein Ergebnis des in T versprochenen Ergebnistyps bekommen, auch wenn die entsprechende Methode in U ausgeführt wird. Für die Typen der formalen Parameter gilt dasselbe 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 Parametertypen in U nur Obertypen der Parametertypen in T sein.

Diese Beziehung für Parametertypen gilt für Sprachen wie Java, in denen Argumente nur vom Aufrufer an die aufgerufene Methode übergeben werden, sogenannte *Eingangsparameter.* In einigen Sprachen wie z.B. Ada können über Parameter Objekte auch von der aufgerufenen Methode an den Aufrufer zurückgegeben werden (*Ausgangsparameter*), also in dieselbe Richtung wie Ergebnisse von Methoden. Für die Typen solcher Parameter gelten dieselben Bedingungen wie für Ergebnistypen. In vielen Sprachen ist es auch möglich, dass über ein und denselben Parameter ein Argument an die Methode übergeben und von dieser ein anderes Argument an den Aufrufer zurückgegeben wird (*Durchgangsparameter*). Die deklarierten Typen solcher Parameter müssen in U und T äquivalent sein.

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.* Man unterscheidet folgende Arten der Varianz:

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

Kontravarianz: Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des Elements im Obertyp. Zum Beispiel sind deklarierte Typen von formalen Eingangsparametern 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. Zum Beispiel sind deklarierte Typen von Variablen und Durchgangsparametern invariant. Die betrachteten in den Typen enthaltenen Elementtypen variieren nicht.

Möglichkeiten und Grenzen: Ein Beispiel in einer Java-ähnlichen Sprache:

```
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.

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. Man kommt kaum in Versuchung sie zu brechen. Nur eine Bedingung, nämlich die geforderte Kontravarianz formaler Parametertypen, möchte man 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 ein formaler Parametertyp stets gleich der Klasse ist, in der die Methode definiert ist, heißt *binäre Methode.* Die Eigenschaft *binär* bezieht sich darauf, dass der Name der Klasse in der Methode mindestens zweimal vorkommt – einmal als Typ von `this` und mindestens einmal als Typ eines formalen 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 3.1.3 kennenlernen.)

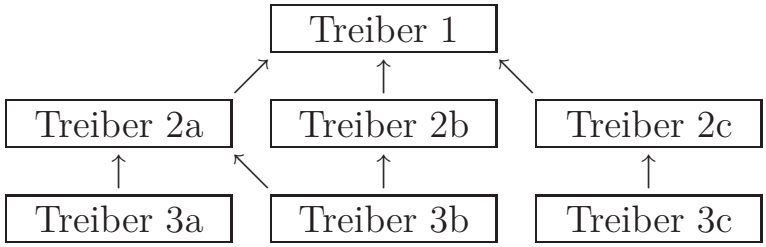
Untertypbeziehungen sind in Java nicht nur aufgrund nominaler Typen stärker eingeschränkt als durch obige Bedingungen notwendig wäre. In Java sind alle Typen invariant, abgesehen von kovarianten Ergebnistypen ab Version 1.5. Der Grund dafür liegt darin, dass bei Zugriffen auf Variablen und Konstanten 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 formalen Parameter unterschieden werden, wäre es schwierig, überladene Methoden von Methoden mit kontravariant veränderten Typen auseinanderzuhalten.

2.1.2 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 gar 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.

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 natürlich auch 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:



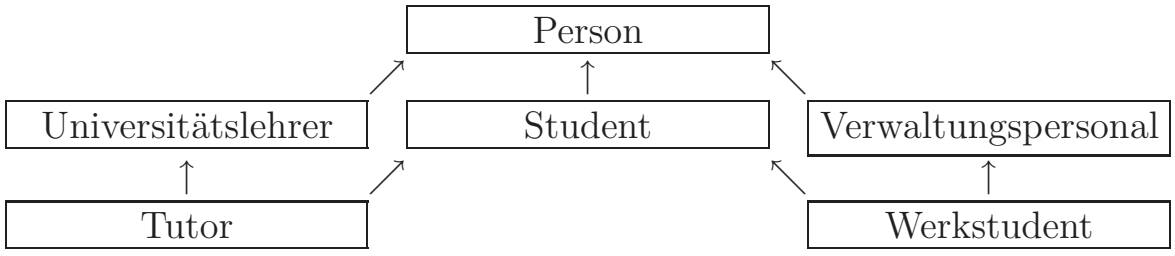
An diesem Bild fällt die Version 3b auf: Sie vereinigt die zwei inkompatiblen Vorgängerversionen 2a und 2b. Ein Untertyp kann mehrere Obertypen haben, die zueinander in keiner Untertypbeziehung stehen. Das ist ein Beispiel für Mehrfachvererbung, während in den anderen Fällen nur Einfachvererbung nötig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

> **Faustregel:** Man soll 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 2.1.1 beschriebenen Bedingungen erfüllt sind. Im Wesentlichen kann die Schnittstelle also nur erweitert werden. Wenn die Aufteilung eines Programms in einzelne Objekte gut ist, bleiben Schnittstellen normalerweise recht stabil.

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

Das, was in obigem Beispiel für verschiedene Versionen einer Klasse funktioniert, kann man genauso gut innerhalb eines einzigen Programms nutzen, wie wir an einem modifizierten Beispiel sehen. Wir wollen ein Programm zur Verwaltung der Personen an einer Universität entwickeln. Die dafür verwendete Klassenstruktur könnte so aussehen:



Entsprechend diesen Strukturen sind Tutor(inn)en sowohl Universitätslehrer(innen) als auch Student(inn)en, und Werkstudent(inn)en an der Universität gehören zum Verwaltungspersonal und sind Student(inn)en. Wir benötigen im Programm eine Komponente, die Serienbriefe – Einladungen zu Veranstaltungen, etc. – an alle Personen adressiert. Für das Erstellen einer Anschrift benötigt man nur Informationen aus der Klasse `Person`.

Die entsprechende Methode braucht nicht zwischen verschiedenen Arten von Personen zu 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(inn)en oder Werkstudent(inn)en sind. Für dieses Beispiel müssen in Java ebenso Interfaces verwendet werden.

> **Faustregel:** Man soll auf Ersetzbarkeit achten um interne Codewiederverwendung im Programm zu erzielen.

Solche Klassenstrukturen können helfen, Auswirkungen nötiger Programmänderungen lokal zu halten. Wenn man eine Klasse, zum Beispiel `Student`, ändert, bleiben andere Klassen, die nicht von `Student` erben, unberührt. Anhand der Klassenstruktur ist leicht erkennbar, welche Klassen von der Änderung betroffen sein können. 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 Klassen selbst betreffen, sondern auch alle Programmstellen, die auf Objekte der Typen `Student`, `Tutor` oder `Werkstudent` zugreifen. Aber Programmteile, die auf Objekte von `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:** Man soll auf Ersetzbarkeit achten um Programmänderungen lokal zu halten.

Falls bei der nötigen Programmänderung alle Schnittstellen der Klasse unverändert bleiben, betrifft die Änderung keine Programmstellen, an denen `Student` und dessen Unterklassen verwendet werden. Lediglich diese Klassen selbst sind betroffen. Auch daran kann man 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 Klassenstruktur 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 man möglichst nur von solchen

Klassen ableiten soll, 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. Man soll nur Untertypen von stabilen Obertypen bilden.

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

> **Faustregel:** Man soll Parametertypen vorausschauend und möglichst allgemein wählen.

Trotz der Wichtigkeit stabiler Schnittstellen darf man nicht bereits zu früh zu viel Zeit in den detaillierten Entwurf der Schnittstellen investieren. Zu Beginn hat man häufig noch nicht genug Information um stabile Schnittstellen zu erhalten. Schnittstellen werden trotz guter Planung oft erst nach einigen Refakorisierungsschritten stabil.

2.1.3 Dynamisches Binden

Bei Verwendung von Untertypen kann der dynamische Typ einer Variablen oder eines Parameters ein Untertyp des deklarierten Typs sein. Eine als `Person` deklarierte Variable kann etwa ein Objekt von `Werkstudent` enthalten. Meist 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 formalen 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 muss man vorsichtig sein: Man macht 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 Art auf konventionelle Weise über eine ganze Zahl bestimmt ist, in die Standardausgabe schreibt:

```
public void gibAnredeAus(int art, String name) {
    switch(art) {
        case  1: System.out.print("S.g. Frau " + name);
                 break;
        case  2: System.out.print("S.g. Herr " + name);
                 break;
        default: System.out.print(name);
    }
}
```

In der objektorientierten Programmierung wird man die Art der Anrede eher durch die Klassenstruktur zusammen mit dem Namen beschreiben:

```
public class Adressat {
    protected String name;
    public void gibAnredeAus() {
        System.out.print(name);
    }
    ... // Konstruktoren und weitere Methoden
}
public class WeiblicherAdressat extends Adressat {
    public void gibAnredeAus() {
        System.out.print ("S.g. Frau " + name);
    }
}
public class MaennlicherAdressat extends Adressat {
    public void gibAnredeAus() {
        System.out.print ("S.g. Herr " + name);
    }
}
```

Durch dynamisches Binden wird automatisch die gewünschte Variante von `gibAnredeAus()` aufgerufen. Statt einer `switch`-Anweisung wird also dynamisches Binden verwendet. Ein Vorteil des objektorientierten Ansatzes ist die bessere Lesbarkeit. Man weiß anhand der Namen, wofür bestimmte Unterklassen von `Adressat` stehen. Die Zahlen 1 oder 2 bieten diese Information nicht. Außerdem ist die Art der Anrede mit dem auszugebenden Namen verknüpft, wodurch man im Programm stets nur ein Objekt von `Adressat` anstatt einer ganzen Zahl und einem String verwalten muss. Ein anderer Vorteil des objektorientierten Ansatzes ist besonders wichtig: Wenn sich herausstellt, dass neben „Frau“ und „Herr“ noch weitere Arten von Anreden, etwa „Firma“, benötigt werden, kann man diese leicht durch Hinzufügen weiterer Klassen einführen. Es sind keine zusätzlichen Änderungen nötig. Insbesondere bleiben die Methodenaufrufe unverändert.

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 Anredeart 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 `Adressat` und ihre Unterklassen konzentriert ist. Es bleibt auch dann alles konzentriert, wenn zu `gibAnredeAus()` weitere Methoden hinzukommen.

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

2.2 Ersetzbarkeit und Objektverhalten

In Abschnitt 2.1 haben wir Bedingungen kennengelernt, 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.

2.2.1 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. Man spricht von *Client-Server-Beziehungen* zwischen Objekten. Einerseits sieht man ein Objekt als einen *Server*, der anderen Objekten seine Dienste zur Verfügung stellt. Andererseits ist ein Objekt ein *Client*, der Dienste anderer Objekte in Anspruch nimmt. Die meisten Objekte spielen gleichzeitig die Rollen von Server und Client.

Für die Ersetzbarkeit von Objekten sind Client-Server-Beziehungen bedeutend. Man kann ein Objekt gegen ein anderes austauschen, wenn das neue 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]:

> 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 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 Variable des Servers 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 Objektzustandes. 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 auf dem Sparbuch durchgeführt werden. Eine Invariante impliziert eine Nachbedingung auf jeder Methode des Servers.

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 Variablen, 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 kann man auch die Reihenfolge von Methodenaufrufen einschränken. Beispielsweise darf man eine Methode namens `initialize` in jedem Objekt nur einmal aufrufen, und 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.

Da History-Constraints noch nicht etabliert und die Regeln dahinter weniger einheitlich sind, werden sie oft nicht als Möglichkeit zur Gestaltung des Softwarevertrags wahrgenommen. Dennoch steckt hinter ihnen 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 in neueren Versionen, die sich aber nur beschränkt zur Beschreibung von Verträgen eignen. Sogar in Eiffel sind viele sinnvolle Zusicherungen nicht direkt ausdrückbar. In diesen Fällen kann und soll man Zusicherungen als Kommentare in den Programmcode schreiben und händisch überprüfen. Umgekehrt soll man fast jeden Kommentar als Zusicherung lesen.

(für Interessierte)

Anmerkungen wie diese gehören nicht zum Prüfungsstoff. Folgendes Beispiel in Eiffel veranschaulicht Zusicherungen:

```
class KONTO feature {ANY}

    guthaben: Integer;
    ueberziehungsrahmen: Integer;

    einzahlen (summe: Integer) is
        require summe >= 0
        do guthaben := guthaben + summe
        ensure guthaben = old guthaben + summe
    end; -{}- einzahlen

    abheben (summe: Integer) is
        require summe >= 0;
                guthaben + ueberziehungsrahmen >= summe
        do guthaben := guthaben - summe
        ensure guthaben = old guthaben - summe
    end; -{}- abheben

    invariant guthaben >= -ueberziehungsrahmen
end -{}- class KONTO
```

Zu jeder Methode kann man vor der eigentlichen Implementierung (`do`-Klausel) eine Vorbedingung (`require`-Klausel) und nach der Implementierung eine Nachbedingung (`ensure`-Klausel) angeben. 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. Wird eine Zusicherung zu Nein ausgewertet, erfolgt eine Ausnahmebehandlung oder Fehlermeldung. In Nachbedingungen ist die Bezugnahme auf Variablen- und Parameterwerte zum Zeitpunkt des Methodenaufrufs erlaubt. So bezeichnet `old guthaben` den Wert von `guthaben` zum Zeitpunkt des Methodenaufrufs.

Diese Klasse sollte bis auf einige syntaktische Details selbsterklärend sein. Die Klausel `feature {ANY}` besagt, dass die danach folgenden Variablendeklarationen und Methodendefinitionen überall im Programm sichtbar sind. Nach dem Schlüsselwort `end` und einem (in unserem Fall leeren) Kommentar kann zur besseren Lesbarkeit der Name der Methode oder der Klasse folgen.

Hier ist ein Java-Beispiel für Kommentare als Zusicherungen:

```
public class Konto {
    public long guthaben;
    public long ueberziehungsrahmen;
    // guthaben >= -ueberziehungsrahmen

    // einzahlen addiert summe zu guthaben; summe >= 0
    public void einzahlen (long summe) {
        guthaben = guthaben + summe;
    }

    // abheben zieht summe von guthaben ab;
    // summe >= 0; guthaben+ueberziehungsrahmen >= summe
    public void abheben (long summe) {
        guthaben = guthaben - summe;
    }
}
```

Beachten Sie, dass Kommentare in der Praxis (so wie in diesem Beispiel) keine expliziten Aussagen darüber enthalten, ob und wenn Ja, um welche Arten von Zusicherungen es sich dabei handelt. Solche Informationen kann man aus dem Kontext herauslesen. 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 „Einzahlen addiert Summe zu Guthaben“ bezieht sich darauf, wie die Ausführung einer bestimmten Methode den Objektzustand verändert. Das kann nur eine Nachbedingung sein. Nachbedingungen lesen sich häufig wie Beschreibungen dessen, was eine Methode tut. Aber die Aussage „summe ≥ 0“ bezieht sich auf eine erwartete Eigenschaft eines Parameters und ist daher eine Vorbedingung auf `einzahlen`. Mit derselben Begründung ist „Abheben zieht Summe von Guthaben ab“ eine Nachbedingung und sind „summe ≥ 0“ und „guthaben+ueberziehungsrahmen $\geq$ summe“ Vorbedingungen auf `abheben`.

Nebenbei bemerkt sollen Geldbeträge wegen möglicher Rundungsfehler niemals durch Fließkommazahlen 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 2.1 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 2.2.2), 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 (= Kommentar) geändert wird.

> **Faustregel:** Zusicherungen sollen stabil bleiben. Das ist für Zusicherungen in Typen 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 2.2.2. 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 in der Signatur steht.

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

Zusicherungen bieten umfangreiche Möglichkeiten zur Gestaltung der Client-Server-Beziehungen. Aus Gründen der Wartbarkeit soll man Zusicherungen aber nur dort einsetzen, wo tatsächlich Informationen benötigt werden, die über jene in der Signatur hinausgehen. Insbesondere soll man Zusicherungen so einsetzen, dass der Klassenzusammenhalt maximiert und die Objektkopplung minimiert wird. In obigem Konto-Beispiel wäre es wahrscheinlich besser, die Vorbedingung, dass abheben den Überziehungsrahmen nicht überschreiten darf, wegzulassen und dafür die Einhaltung der Bedingung direkt in der Implementierung von abheben durch eine `if`-Anweisung zu überprüfen. Dann ist nicht mehr 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. Einen äußerst unerwünschten Effekt erzielt man, wenn man Zusicherungen (= nötige Kommentare) einfach aus Bequemlichkeit nicht in den Programmcode schreibt, der Client aber trotzdem bestimmte Eigenschaften vom Server erwartet (oder umgekehrt), also beispielsweise implizit voraussetzt, dass eine Einzahlung den Kontostand erhöht. In diesem Fall hat man die Abhängigkeiten zwischen Client und Server nur versteckt. Wegen der Abhängigkeiten können Programmänderungen zu unerwarteten Fehlern führen, die man nur schwer findet, da die Abhängigkeiten nicht offensichtlich sind. Es sollen daher alle Zusicherungen explizit im Programmcode stehen. Andererseits sollen Client und Server aber so unabhängig wie möglich bleiben.

> **Faustregel:** Alle benötigten Zusicherungen sollen (explizit als Kommentare oder zumindest durch sprechende Namen impliziert) 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 ist es gut, 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.

2.2.2 Untertypen und Verhalten

Zusicherungen, die zu Typen gehören, müssen auch bei der Verwendung von 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 2.1 kennengelernt haben, müssen folgende Bedingungen gelten, damit ein Typ U Untertyp eines Typs T ist [24]:

Vorbedingung: Jede Vorbedingung auf einer Methode in T muss eine Vorbedingung auf der entsprechenden Methode in U implizieren. Das heißt, 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 muss die Vorbedingung in U automatisch erfüllt sein, wenn sie in T erfüllt ist. Wenn Vorbedingungen in U aus T übernommen werden, können sie mittels Oder-Verknüpfungen schwächer werden. Ist die Vorbedingung in T zum Beispiel „$x > 0$“, kann die Vorbedingung in U auch „$x > 0$ oder $x = 0$“, also abgekürzt „$x \geq 0$“ lauten.

Nachbedingung: Jede Nachbedingung auf einer Methode in U muss eine Nachbedingung 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 muss eine Nachbedingung in T

automatisch erfüllt sein, wenn ihre Entsprechung in U erfüllt ist. Wenn Nachbedingungen in U aus T übernommen werden, können sie mittels Und-Verknüpfungen stärker werden. Lautet die Nachbedingung in T beispielsweise „`result` > 0“, kann sie in U auch „`result` > 0 und `result` > 2“, also „`result` > 2“ sein.

Invariante: Jede Invariante in U muss eine Invariante 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 Invariante in T muss automatisch erfüllt sein, wenn sie in U erfüllt ist. Wenn Invarianten in U aus T übernommen werden, können sie, wie Nachbedingungen, mittels Und-Verknüpfungen stärker werden. Dieser Zusammenhang mit Nachbedingungen ist notwendig, da Invarianten entsprechende Nachbedingungen auf allen Methoden des Typs implizieren.

Die Begründung geht davon aus, dass Objektvariablen nicht durch andere Objekte 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 Variablen 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 möglichst nicht durch andere Objekte verändert werden.

Server-kontrollierter History-Constraint: Dafür gilt im Prinzip dasselbe 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 im Prinzip dasselbe wie für Vorbedingungen, jedoch bezogen auf Einschränkungen in der Reihenfolge von Methodenaufrufen. Eine Reihenfolge von Methodenaufrufen nennt man *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. Im Allgemeinen müssen wir bei der Überprüfung Client-kontrollierter History-Constraints 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 Client kommen.

Im Prinzip lassen sich obige Bedingungen auch formal überprüfen. In Programmiersprachen wie Eiffel, in denen Zusicherungen formal definiert sind, wird das tatsächlich 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 natürlich keine formale Überprüfung möglich. Daher müssen Programmierer(innen) alle nötigen Überprüfungen per Hand durchführen.

Im Einzelnen muss sichergestellt werden, dass

- obige Bedingungen für Untertypbeziehungen eingehalten werden,
- die Implementierungen der Server die Nachbedingungen, Invarianten und Server-kontrollierten History-Constraints erfüllen und nichts voraussetzen, was nicht in Vorbedingungen, Invarianten und History-Constraints (beider Arten) festgelegt ist
- und Clients die Vorbedingungen und Client-kontrollierten History-Constraints der Aufrufe erfüllen und nichts voraussetzen, was nicht durch Nachbedingungen, Invarianten und History-Constraints zugesichert wird.

Es kann sehr aufwendig sein, alle solchen Überprüfungen vorzunehmen. Einfacher geht es, wenn man während der Codeerstellung und bei Änderungen stets an die einzuhaltenden Bedingungen denkt, die Überprüfungen also nebenbei erfolgen. Wichtig ist darauf zu achten, dass die Zusicherungen unmissverständlich formuliert sind. Nach Änderung einer Zusicherung ist die Überprüfung besonders schwierig, und 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:

```
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 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 man den Inhalt dieser Beschreibungen von Methoden genauer betrachtet, sieht man, dass es sich dabei um Nachbedingungen handelt. Da Nachbedingungen festlegenen, 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` darf man sich nicht dazu verleiten lassen, generell starke Einschränkungen in Zusicherungen zu verwenden. Solche Einschränkungen erschweren die Wartung (siehe Abschnitt 2.2.1). 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
        ...;
    }
}
```

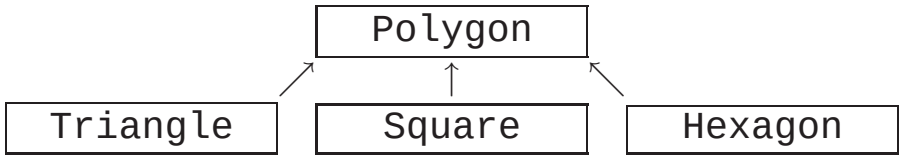
Aber `SetWithoutDelete` können wir, wie der Klassenname schon sagt, nicht um eine Methode `delete` erweitern. Jede vernünftige Nachbedingung von `delete` muss in Konflikt zum History-Constraint stehen. Man darf nicht zu früh festlegen, dass es kein `delete` gibt, nur weil man es gerade nicht braucht. Invarianten wie in `SetWithoutDelete` soll man nur verwenden, wenn man wirklich darauf angewiesen ist. Andernfalls verbaut man sich Wiederverwendungsmöglichkeiten.

Kommentare als Zusicherungen setzen voraus, dass man Untertypbeziehungen explizit deklariert, also nominale Typen verwendet. Damit bringt man den Compiler dazu, beliebige weitere Bedingungen (gegenüber den in Abschnitt 2.1.1) für eine Untertypbeziehung vorauszusetzen. Beispielsweise muss man 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 könnte ein Objekt von `Set` auch verwendet werden, wo eines von `SetWithoutDelete` erwartet wird. Es soll auch keine Untertypbeziehung zwischen `SetWithoutDelete` und `SetWithDelete` bestehen, obwohl dafür alle Bedingungen aus Abschnitt 2.1.1 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 auch Codevererbung sinnvoll.

2.2.3 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 benötigt man oft nur eine der Aufgaben, 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 erfüllen einen ähnlichen Zweck.

Nehmen wir als Beispiel folgende Klassenstruktur:



Jede Unterklasse von `Polygon` beschreibt ein z. B. am Bildschirm darstellbares Vieleck mit einer bestimmten Anzahl von Ecken. `Polygon` selbst beschreibt keine bestimmte Anzahl von Ecken, sondern fasst nur die Menge aller Vielecke zusammen. Wenn man eine Liste unterschiedlicher Vielecke benötigt, wird man 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 man kann Unterklassen von `Polygon` ableiten. Jede Unterklasse muss eine Methode `draw` enthalten, da diese Methode in `Polygon` deklariert ist. Genaugenommen ist `draw` als abstrakte Methode deklariert; das heißt, es ist keine Implementierung von `draw` angegeben, sondern nur dessen Schnittstelle mit einer kurzen Beschreibung – einer Zusicherung als Kommentar. In abstrakten Klassen brauchen wir keine Implementierungen für Methoden anzugeben, da die Methoden ohnehin nicht ausgeführt

werden; 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 brauchen abstrakte Methoden nicht zu überschreiben. Neben 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, die keine Implementierungen enthalten, sind eher stabil als andere Klassen. Zur Verbesserung der Wartbarkeit soll man neue Klassen vor allem von stabilen Klassen ableiten. Außerdem soll man möglichst stabile Typen für formale Parameter und Variablen verwenden. Da es leichter ist, abstrakte Klassen ohne Implementierungen stabil zu halten, ist man gut beraten, hauptsächlich solche Klassen für Parameter- und Variablentypen zu verwenden.

> **Faustregel:** Es ist empfehlenswert, als Obertypen und Parametertypen hauptsächlich abstrakte Klassen (ohne Implementierungen) und Interfaces zu verwenden.

Vor allem 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. Diesen Konflikt kann man leicht lösen, indem man für die Typen der Parameter nur abstrakte Klassen verwendet. Es ist ja leicht, zu jeder konkreten Klasse eine oder mehrere abstrakte Klassen als Oberklassen zu schreiben, die die benötigten Bedingungen möglichst genau angeben. Damit werden unnötige Abhängigkeiten vermieden.

2.3 Vererbung versus Ersetzbarkeit

Vererbung ist im Grunde sehr einfach: Von einer Oberklasse wird scheinbar, aber meist nicht wirklich, eine Kopie angelegt, die durch Erweitern und Überschreiben abgeändert wird. Die resultierende Klasse ist die Unterklasse. Wenn man nur Vererbung betrachtet und Einschränkungen durch Untertypbeziehungen ignoriert, 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 unterscheiden und dann die Bedeutungen dieser Beziehungen für die Codewiederverwendung untersuchen.

2.3.1 Reale Welt, Vererbung, Ersetzbarkeit

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

Untertypen: Diese Beziehung, die auf dem Ersetzbarkeitsprinzip beruht, haben wir bereits untersucht.

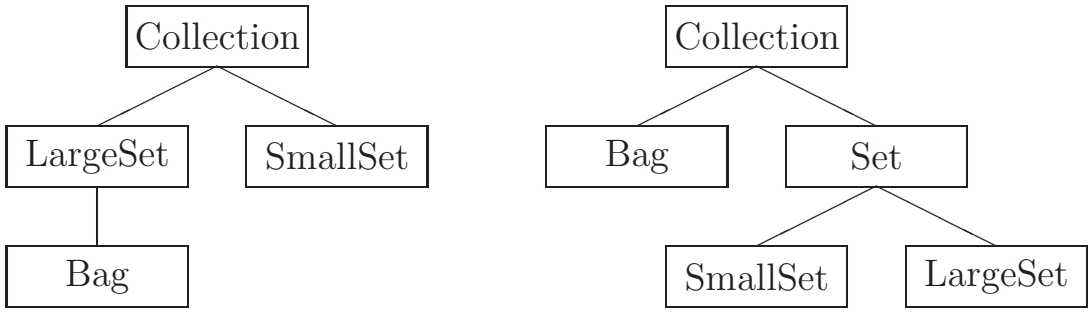
Vererbung: Dabei entsteht eine neue Klasse durch Abänderung einer bestehenden Klasse. Es ist nicht nötig, aber wünschenswert, dass dabei Code aus der Oberklasse in der Unterklasse direkt wiederverwendet wird. Für reine Vererbung ist das Ersetzbarkeitsprinzip irrelevant.

Is-a-Beziehung: Schon 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 „ein Student ist eine 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 Kriterien sind in Java reine Vererbungs- von Untertypbeziehungen nicht unterscheidbar.

Man kann leicht erkennen, ob reine Vererbungs- oder Untertypbeziehungen angestrebt werden. Betrachten wir dazu ein Beispiel:



reine Vererbungsbeziehung Untertypbeziehung

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. 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` dieselbe Bedeutung haben sollen. Wir wollen im Programmcode nur selten zwischen `LargeSet` und `SmallSet` unterscheiden. `Bag` und `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 letztendlich von Details und beabsichtigten Verwendungen abhängen.

2.3.2 Vererbung und Codewiederverwendung

Manchmal kann man durch reine Vererbungsbeziehungen, die Untertypbeziehungen unberücksichtigt lassen, einen höheren Grad an direkter Codewiederverwendung erreichen als wenn man bei der Softwareentwicklung Untertypbeziehungen anstrebt. 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 man trotzdem ein Objekt einer Unterklasse statt dem einer Oberklasse verwendet, wird sich ein unerwünschtes Programmverhalten zeigen (Fehler im Programm). Verzichtet man 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 gewinnt man zwar durch die reine Vererbung bessere direkte Codewiederverwendung in kleinem Umfang, tauscht diese aber gegen viele Möglichkeiten für die indirekte Codewiederverwendung in großem Umfang, die nur durch die Ersetzbarkeit gegeben sind.

> **Faustregel:** Wiederverwendung durch das Ersetzbarkeitsprinzip 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. Die soll sich Untertypbeziehungen unterordnen. Im Allgemeinen soll es keine Vererbungsbeziehung geben, die nicht auch eine Untertypbeziehung ist, bei der also alle Zusicherungen kompatibel sind.

Wie die Erfahrung zeigt, vergessen Programmieranfänger 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.

Man soll aber nicht gleich von vornherein 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 kann man bereits in der abstrakten Klasse `Set` implementieren, von der `SmallSet` und `LargeSet` erben. 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 desselben Codes, 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 muss man alle gleichen oder ähnlichen Programmteile gleichzeitig ändern, wenn sich die Anforderungen ändern. Gerade dabei kann Vererbung sehr hilfreich sein.

> **Faustregel:** Auch reine Vererbung kann sich positiv auf die Wartbarkeit auswirken.

Es kommt vor, dass nicht alle solchen 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. 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; ... }
    }
}
```

Die Methode `foo` muss gänzlich neu geschrieben werden, obwohl der Unterschied minimal ist. Man muss ja immer eine ganze Methode überschreiben, nicht nur eine Anweisung der Methode. Eine Aufspaltung von `foo` in mehrere Methoden kann helfen:

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

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

Das ist eine Anwendung der *Template-Method* (siehe Abschnitt 5.3.3). Man braucht nur mehr einen Teil der Methode zu überschreiben. Solche Techniken setzen aber voraus, dass man bereits beim Schreiben der Klasse `A` sehr klare Vorstellungen davon hat, 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 kann man auch durch zusätzliche Parameter beschreiben und nach außen sichtbare Methoden nur zum Setzen der Parameter verwenden:

```
public class A {
    public void foo() { fooY(1); }
    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` braucht nur ein Argument an `fooY` zu 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)

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.

2.3.3 Fehlervermeidung

Im Zusammenhang mit Untertypen und Vererbung passieren immer wieder schwere Programmierfehler, die sich erst später nach Programmänderungen äußern. Häufig ist Programmierer(inne)n gar nicht bewusst, dass sie etwas falsch machen. Die Fehler fallen ja bei einfachem Testen nicht gleich 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 man von einem Objekt eines Untertyps anderes Verhalten erwartet 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 verliert man bald den Überblick.

Die Einhaltung folgender Regel sollte fast alle Probleme mit falschen Untertypbeziehungen vermeiden:

> **Faustregel:** Man muss beim Programmieren *stets* prüfen, ob sich ein Objekt tatsächlich *immer* so verhält, wie in *jedem* Typ des Objekts beschrieben.

Leider enthält diese Regel einige Schwierigkeiten, nämlich die Wörter „stets“, „immer“ und „jedem“. Man muss also sehr häufig recht komplexe und umfangreiche Prüfungen vornehmen. Das geht nur, wenn man während des Programmierens ein recht genaues Modell des Objektverhaltens und der Verhaltensbeschreibungen (also der Typen) im Kopf hat, und das für alle Objekte und Typen, mit denen man sich gerade beschäftigt. Dazu ist höchste Konzentration erforderlich. Es reicht ein Moment der Unachtsamkeit oder Ermüdung, und schon ist ein schwerwiegender, aber kaum zu entdeckender Fehler eingebaut. Man muss sich sehr tief in die Verhaltensbeschreibungen einarbeiten um keine Details zu übersehen. Es ist viel Übung erforderlich, bis man sich die notwendigen Überprüfungen so gut eingeprägt hat, dass sie im Unterbewusstsein während des Programmierens quasi automatisch ablaufen. Zusammengefasst kann man folgende Ratschläge geben:

- Gründlich in vorgegebene Verhaltensbeschreibungen einarbeiten.
- Auf eine Umgebung achten, in der man sich gut konzentrieren kann.
- Bei Ermüdung unverzüglich eine Pause einlegen.
- Viel selbst programmieren, auch komplizierte Aufgaben selbst lösen.

Allerdings werden diese Ratschläge dadurch relativiert, dass man fast immer unter Zeitdruck arbeiten muss. 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 sollte man sich, so gut es geht, an diese Ratschläge halten. Dann lernt man nicht nur wie man die inhaltlichen Fehler ausmerzt, sondern auch wie man mit dem Zeitdruck umgeht.

Ein bewusster und gut durchdachter 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 man ständig im Kopf haben muss. Außerdem wird man eher auf Widersprüche aufmerksam, wenn die widersprüchlichen Kommentare in derselben Klasse oder im selben Interface 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 2.3.1. Ein falsches oder noch nicht ganz verinnerlichtes Verständnis liegt mit hoher Wahrscheinlichkeit vor, wenn man es als schwierig empfindet, 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 man sich so daran gewöhnt hat, dass man die Problematik kaum mehr sieht. Das Umlernen ist alles andere als einfach. Aber es ist notwendig, da das falsche Verständnis langfristig zu schwerwiegenden Fehlern führen kann.

Eine andere häufige Fehlerursache ist das Übersehen von scheinbaren Nebensächlichkeiten. Aufgrund der wichtigsten Bedingungen hätten wir zwar eine Untertypbeziehung, aber einige kleine, auf den ersten Blick unwichtige Bedingungen zerstören diese Beziehung, ohne dass es uns auffällt. Diesem Problem kann man nur mit mehr Aufmerksamkeit begegnen. Beispielsweise können die störenden Bedingungen von Programmteilen kommen, die wir nur für das Testen brauchen, und die wir in unseren Überlegungen daher nicht näher betrachtet haben. Aber diese Bedingungen können 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. Die Aufzählung solcher Beispiele, in denen man mehr auf Nebensächlichkeiten achten sollte, lässt sich noch endlos weiterführen.

Manchmal ist aber gerade die Konzentration auf Nebensächlichkeiten schuld an falschen Untertypbeziehungen. Wenn man etwa zu sehr an einfache Änderbarkeit durch Vererbung denkt, übersieht man vielleicht wesentliche Kriterien für Untertypbeziehungen. Man braucht schon etwas Erfahrung, um sich stets auf das Wichtigste zu konzentrieren.

2.4 Exkurs: 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. Erfahrene Java-Programmierer mögen verzeihen, dass es zur Erreichung dieses Ziels notwendig ist, einige scheinbar ganz triviale Sprachkonstrukte zu erklären.

2.4.1 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 sollte man sich an diese Konventionen halten, auch wenn sie der Compiler nicht überprüft. 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 formalen 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 `this` wie den Namen einer Variable verwendet. Tatsächlich bezeichnet `this` immer das aktuelle Objekt der Klasse. 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 formalen Parameter `r` des Konstruktors zu unterscheiden. Wie in diesem Beispiel können formale Parameter (oder lokale Variablen) Variablen im aktuellen Objekt der Klasse verdecken, die denselben Namen haben. Über `this` kann man dennoch auf die Objektvariablen zugreifen. Wie im zweiten Konstruktor gezeigt, kann man `this` immer verwenden, auch wenn es gar nicht nötig ist. Außerdem benötigt man `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 Kopien der Objektvariablen.

Manchmal benötigt man Variablen, die nicht zu einem bestimmten Objekt einer Klasse gehören, sondern zur Klasse selbst. Solche *Klassenvariablen* kann man in Java einfach durch Voranstellen des Schlüsselwortes `static` deklarieren. Klassenvariablen stehen nicht in den Objekten der Klasse, sondern in der Klasse selbst. Auf eine in einer Klasse `A` deklarierte Klassenvariable `x` kann man durch `A.x` zugreifen. Auf Objektvariablen kann man hingegen nur über ein Objekt der Klasse zugreifen, 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, sollte man vermeiden Klassenvariablen als Variablen zu sehen, die allen Objekten einer Klasse gemeinsam gehören, da diese Sichtweise längerfristig zu unklaren Verantwortlichkeiten und damit zu Konflikten führt. Von einer nichtstatischen Methode aus sollte man auf eine Klassenvariable nur mit derselben Vorsicht zugreifen, mit der man auf Variablen eines anderen Objekts zugreift – 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 man darf nicht auf Objektvariablen zugreifen. 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, sollte man 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 formalen 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.

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 kann man private Variablen und Methoden aus der Umgebung verwenden. Es gibt zwei Arten geschachtelter Klassen:

Statische geschachtelte Klassen: Diese 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 geschachtelten Klassen sind Objektvariablen und nichtstatische Methoden ganz normal zugreifbar. Objekte von `EnclosingClass.StaticNestedClass` werden 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 umschließende Klasse (`EnclosingClass`) können in `InnerClass` uneingeschränkt verwendet werden, auch private. Innere Klassen dürfen jedoch keine statischen Methoden und keine statischen geschachtelten 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` eine Variable 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. Beim Einsatz ist jedoch zu bedenken, dass immer eine sehr starke Kopplung zwischen geschachtelten und umgebenden Klassen bzw. ihren Objekten besteht. Geschachtelte Klassen sollen nur verwendet werden, wenn auch alternative Implementierungsmöglichkeiten ähnlich starke Objekt-Kopplungen ergeben würden.

In Java sind auch Klassen Objekte, nämlich als Instanzen der vordefinierten Klasse `Class`. Klassenvariablen, statische Methoden und statisch geschachtelte Klassen verwendet man am besten nur dann, wenn man sie als Variablen, Methoden bzw. Klassen eines Objekts von `Class` betrachtet. In allen anderen Fällen sollte man Objektvariablen, nichtstatische Methoden und innere Klassen bevorzugen.

2.4.2 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. Sonst hätten wir eine Endlosrekursion.

Eine Variable in der Unterklasse kann denselben 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 kann man in der Unterklasse direkt durch ihren Namen ansprechen. Die Variable in der Oberklasse, von der die Unterklasse direkt abgeleitet ist, kann man über `super` ansprechen. 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, kann man durch Typumwandlungen ansprechen. Z.B. ist `((Oberklasse)this).v` die Variable, die in `Oberklasse` den Namen `v` hat. Eine verdeckte Klassenvariable kann man leicht über den Klassennamen ansprechen, beispielsweise durch `Oberklasse.v`.

Überschriebene nichtstatische Methoden aus der Oberklasse, von der direkt abgeleitet wird, kann man über `super` ansprechen, wie wir in Abschnitt 2.3 gesehen haben. Mittels Typumwandlung kann man überschriebene Methoden aus Oberklassen aber niemals ansprechen: Eine Typumwandlung ändert nur den deklarierten Typ. Auf Grund von dynamischem Binden hat der deklarierte Typ (abgesehen von der Auswahl überladener Methoden, siehe Kapitel 3) keinen Einfluss auf die Methode, die ausgeführt wird. Dynamisches Binden macht den Effekt der Typumwandlung wieder rückgängig. Statische Methoden aus Oberklassen kann man wie Klassenvariablen durch Voranstellen des Klassennamens ansprechen.

Es soll noch einmal betont werden, dass eine Methode der Unterklasse eine der Oberklasse nur überschreibt, wenn Name, Parameteranzahl und Parametertypen gleich sind. 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.

In Java gibt es eine Möglichkeit zu verhindern, dass eine Methode in einer Unterklasse überschrieben wird. Das Überschreiben ist unmöglich, wenn die Methode mit dem Schlüsselwort `final` definiert wurde. Da ein Überschreiben nicht möglich ist, werden solche Methoden durch statisches Binden aufgerufen. Dadurch erfolgt der Aufruf (meist unmerklich) schneller als durch dynamisches Binden. Trotzdem soll man `final` im Normalfall eher nicht verwenden, da das Verbieten des Überschreibens die Wartbarkeit vermindern kann. Nicht überschreibbare Methoden sind für spezielle Fälle vorgesehen, in denen man das Überschreiben aus Sicherheitsgründen, z. B. zur Vermeidung der Umgehung einer Passwortabfrage, verbieten will.

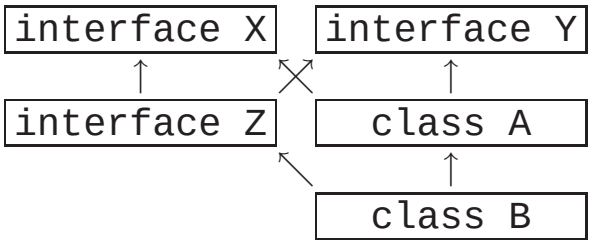
> **Faustregel:** Methoden sollen nur in Spezialfällen als `final` deklariert sein.

Man kann auch ganze Klassen mit dem Schlüsselwort `final` versehen. Solche Klassen haben keine Unterklassen. Dadurch ist es auch nicht möglich, die Methoden der Klassen zu überschreiben. In manchen objektorientierten Programmierstilen verwendet man 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 aufwendig 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 formalen Parametern und Variablen vermeidbar. In diesem Fall kann die konsequente Verwendung von `final` Klassen für alle nicht-abstrakten Klassen vorteilhaft sein. Damit erhält man einen Programmierstil ähnlich dem von Sather (siehe Abschnitt 2.3.2).

Interfaces sind in Java eingeschränkte abstrakte Klassen, die (im Gegenzug zu den Einschränkungen) Mehrfachvererbung unterstützen. Interfaces unterscheiden sich von normalen abstrakten Klassen wie folgt:

- Beginnen mit `interface` statt `abstract class`.
- Alle Methoden sind abstrakt, Modifier `abstract` nicht notwendig.
- Enthält außer `static-final`-Konstanten keine Variablen.
- In allen anderen Fällen sind die Modifier `static`, `final`, `private` und `protected` verboten.
- Methoden und Konstanten sind immer `public`, auch ohne Modifier.
- Nach `extends` können mehrere, durch Komma getrennte Namen von Interfaces, aber nicht von Klassen stehen (Mehrfachvererbung).

Auch Klassen können von mehreren Interfaces erben:




```
public interface X {
    static final double PI = 3.14159;
    double fooX();
}
public interface Y {
    double fooY();
}
public interface Z extends X, Y {
    double fooZ();
}
public class A implements X, Y {
    protected double factor = 2.0;
    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(); }
    public double fooZ() { return factor / fooX(); }
}
```

Interface `Z` erbt von `X` und `Y`. Somit enthält `Z` die Konstante `PI` sowie die Methoden `fooX`, `fooY` und `fooZ`. 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, beziehungsweise die Klasse die durch die Interfaces spezifizierten Schnittstellen implementiert. In einer Klassendefinition kann nach `extends` nur eine Klasse stehen.

Interfaces sind als Typen verwendbar. In dieser Hinsicht unterscheiden sie sich nicht von Klassen. Wie für abstrakte Klassen ohne Implementierungen gilt die Faustregel, dass Interfaces stabiler sind als Klassen mit Implementierungen. Durch Mehrfachvererbung sind sie oft flexibler einsetzbar als abstrakte Klassen. Daher sollten Interfaces immer verwendet werden, wo dies möglich ist, das heißt, wo die oben genannten Einschränkungen zu keinen Nachteilen führen.

Faustregel: Interfaces sind abstrakten Klassen vorzuziehen.

Interfaces dienen fast ausschließlich der Festlegung von Untertypbeziehungen, die das Ersetzbarkeitsprinzip erfüllen. Reine Vererbung ist mit Interfaces nicht sinnvoll. Das ist ein weiterer Grund, warum Interfaces abstrakten Klassen vorzuziehen sind: Man kommt nicht so leicht in Versuchung, sich auf Vererbung anstatt auf Untertypbeziehungen zu konzentrieren. Wie bei Klassen gilt auch bei Interfaces, dass die entsprechenden Typen gemachte Zusicherungen einschließen. Man sollte also stets Zusicherungen hinschreiben und die Kompatibilität der Zusicherungen händisch überprüfen. In Interfaces sind Zusicherungen sogar noch wichtiger als in Klassen, da das Verhalten von Methoden höchstens aus gut gewählten Methodennamen, aber nicht aus einer Implementierung abgeleitet werden kann. Außerdem suchen erfahrene Programmierer 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.

2.4.3 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. Der Name des Verzeichnisses

ist der Paketname. Während der Softwareentwicklung steht der Quellcode einer Klasse meist im selben Verzeichnis wie die compilierte Klasse. Auch die Datei, die den Quellcode enthält, hat denselben Namen wie die Klasse, aber mit der Endung `.java`. Es ist auch möglich, dass eine Quellcodedatei mehrere Klassen enthält. Von diesen Klassen darf nur eine, nämlich 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 Paketes enthalten, in dem die Namen definiert sind, außer wenn sie im selben Paket definiert sind. Diese Namen sind relativ zu einer Basis, dem *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
```

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 Dateien 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;
```

kann man `foo` noch einfacher durch „`AClass.foo();`“ aufrufen. Häufig möchte man 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 Paketes bezeichnet, zu dem die Klasse in der Quelldatei gehört. Ist eine solche Zeile in der Quelldatei vorhanden, muss der Aufruf von `javac` zur Compilation der Datei oder `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 Quelldatei oder compilierte Datei in einem anderen Verzeichnis steht, lässt sie sich nicht compilieren beziehungsweise 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 verwendet 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 demselben Namen verdeckt sind. Einheiten, die mit dem Schlüsselwort `private` definiert wurden, sind sonst nirgends sichtbar. Sie werden auch nicht vererbt.

(für Interessierte)

Genau genommen stimmt es 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 durch 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, dass `private` Einheiten nicht vererbt werden, da es beim Programmieren meist diesen Anschein hat, auch wenn es aus der Sicht des Compilers nicht stimmt.

Einheiten, die mit dem Schlüsselwort `public` definiert wurden, sind dagegen überall sichtbar und werden vererbt. Man kann

```
myclasses.examples.test.AClass.foo();
```

aufrufen, wenn sowohl die Klasse `AClass` als auch die statische Methode `foo` mit dem vorangestellten Schlüsselwort `public` definiert wurden. In allen anderen Fällen darf man `foo` nicht aufrufen.

Neben diesen beiden Extremfällen gibt es noch zwei weitere Möglichkeiten zur Steuerung der Sichtbarkeit. Bei diesen Möglichkeiten sind Ein-

heiten zwar im selben 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. Sie sind im selben Paket sichtbar, sonst aber nirgends. Einheiten mit Default-Sichtbarkeit werden in Unterklassen nur geerbt, wenn die Unterklassen im selben 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.

Für weniger geübte Java-Programmierer(innen) ist es gar nicht leicht, stets die richtigen Sichtbarkeitseigenschaften zu wählen. Hier sind einige Ratschläge, die diese Wahl erleichtern sollen:

- Alle Methoden, Konstruktoren und Konstanten (und in ganz seltenen Fällen auch Variablen – ist aber verpönt), die man bei der Verwendung der Klasse oder von Objekten der Klasse benötigt, sollen `public` sein.

- Man verwendet `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, verwendet man am besten `protected`. Man sollte `protected` Variablen weitgehend meiden und stattdessen in Unterklassen nur über `protected` Methoden indirekt darauf zugreifen. Generell ist `protected` nur einzusetzen, wenn es in Unterklassen einen echten Bedarf dafür gibt.

- 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. Man verwendet dafür am besten Default-Sichtbarkeit. Außer bei einem echten Bedarf an enger Zusammenarbeit zwischen Klassen sollte man `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 kann man in der nach außen sichtbaren Schnittstelle eines Objekts immer auch eine Methode zum Abfragen des aktuellen Wertes („Getter“) und eine zum Setzen des Wertes („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 Objekt-Kopplung und niedrigen Klassen-Zusammenhalt und damit auf eine schlechte Faktorisierung des Programms hin. Refaktorisierung ist angesagt.

> **Faustregel:** Methoden zum direkten Setzen bzw. Abfragen von Variablenwerten sind im Normalfall zu vermeiden.

Wenn unklar ist, wo etwas sichtbar sein soll, verwendet man zu Beginn die am stärksten eingeschränkte Variante. Erst wenn sich herausstellt, dass eine weniger restriktive Variante nötig ist, erlaubt man weitere Zugriffe. Es ist viel einfacher, Restriktionen der Sichtbarkeit aufzuheben, als neue Einschränkungen einzuführen. Aber auch beim Ausweiten der Sichtbarkeit können sich Probleme ergeben.

(für Interessierte)

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, kaum 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. Softwareentwickler(innen) 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.

2.5 Wiederholungsfragen

1. In welchen Formen (mindestens zwei) kann man durch das Ersetzbarkeitsprinzip Wiederverwendung erzielen?

2. Wann ist ein struktureller Typ Untertyp eines anderen strukturellen Typs? Welche Regeln müssen dabei erfüllt sein? Welche zusätzliche Bedingungen gelten für nominale Typen bzw. in Java? (Hinweis: Häufige Prüfungsfrage!)

3. Sind die in Punkt 2 angeschnittenen Bedingungen (sowie das, was Compiler prüfen können) hinreichend, damit das Ersetzbarkeitsprinzip erfüllt ist? Wenn nicht, was muss noch beachtet werden?

4. Was bedeutet Ko-, Kontra- und Invarianz, und für welche Typen in einer Klasse trifft welcher dieser Begriffe zu?

5. Was sind binäre Methoden, und welche Schwierigkeiten verursachen sie hinsichtlich der Ersetzbarkeit?

6. Wie soll man Typen formaler Parameter wählen um gute Wartbarkeit zu erzielen?

7. Warum ist dynamisches Binden gegenüber `switch`- oder geschachtelten `if`-Anweisungen zu bevorzugen?

8. Welche Rolle spielt dynamisches Binden für die Ersetzbarkeit und Wartbarkeit?

9. Welche Arten von Zusicherungen werden unterschieden, und wer ist für die Einhaltung verantwortlich? (Hinweis: Häufige Prüfungsfrage!)

10. Wie müssen sich Zusicherungen in Unter- und Obertypen zueinander verhalten, damit das Ersetzbarkeitsprinzip erfüllt ist? Warum? (Hinweis: Häufige Prüfungsfrage!)

11. Warum sollen Signaturen und Typen stabil bleiben? Wo ist Stabilität besonders wichtig?

12. Was ist im Zusammenhang mit allgemein zugänglichen (= möglicherweise nicht nur innerhalb des Objekts geschriebenen) Variablen und Invarianten zu beachten?

13. Wie genau sollen Zusicherungen spezifiziert sein?

14. Wozu dienen abstrakte Klassen und abstrakte Methoden? Wo und wie soll man abstrakte Klassen einsetzen?

15. Ist Vererbung dasselbe wie das Ersetzbarkeitsprinzip? Wenn Nein, wo liegen die Unterschiede?

16. Worauf kommt es zur Erzielung von Codewiederverwendung eher an – auf Vererbung oder Ersetzbarkeit? Warum?

17. Was bedeuten folgende Begriffe in Java?
    - Objektvariable, Klassenvariable, statische Methode
    - Static-Initializer
    - geschachtelte und innere Klasse
    - `final` Klasse und `final` Methode
    - Paket

18. Wo gibt es in Java Mehrfachvererbung, wo Einfachvererbung?

19. Welche Arten von `import`-Deklarationen kann man in Java unterscheiden? Wozu dienen sie?

20. Wozu benötigt man eine `package`-Anweisung?

21. Welche Möglichkeiten zur Spezifikation der Sichtbarkeit gibt es in Java, und wann soll man welche Möglichkeit wählen?

22. Wodurch unterscheiden sich Interfaces in Java von abstrakten Klassen? Wann soll man Interfaces verwenden? Wann sind abstrakte Klassen besser geeignet?

3 Generizität und Ad-hoc-Polymorphismus

In Kapitel 2 haben wir uns mit Untertypen beschäftigt. Nun werden wir alle weiteren Arten von Polymorphismus in objektorientierten Sprachen betrachten. Die Abschnitte 3.1 und 3.2 sind der Generizität und ihrer Verwendung gewidmet. Zum besseren Verständnis behandeln wir in Abschnitt 3.3 eine Alternative zur Generizität, die auf dynamischen Typvergleichen und Typumwandlungen beruht. In Abschnitt 3.4 werden wir uns Unterschiede zwischen Überladen und mehrfachem dynamischem Binden durch Multimethoden vor Augen führen. Dabei werden wir Möglichkeiten aufzeigen, mehrfaches dynamisches Binden in Sprachen zu verwenden, die nur einfaches dynamisches Binden bereitstellen.

3.1 Generizität

Generische Klassen, Typen und Methoden enthalten Parameter, für die Typen eingesetzt werden. Andere Arten generischer Parameter unterstützt Java nicht. Daher nennt man 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.

3.1.1 Wozu Generizität?

An Stelle expliziter Typen werden im Programm Typparameter verwendet. Das sind einfach nur Namen, die später 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, in denen alle Listenelemente vom Typ `String` sind. Bald stellt sich jedoch heraus, dass wir

auch Listen mit Elementen vom Typ `Integer` sowie solche mit Elementen vom Typ `Student` brauchen. Da der existierende Code nur mit Zeichenketten umgehen kann, müssen wir zwei neue Varianten 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 tatsächlich existierender Typ, sondern ein Typparameter. Den Code für `String`-, `Integer`- und `Student`-Listen könnte man daraus erzeugen, indem man alle Vorkommen von `Element` durch diese Typnamen ersetzt.

Auf den ersten Blick schaut es so aus, als ob man denselben Effekt auch erzielen könnte, wenn man im ursprünglichen `String`-Listencode alle Vorkommen von `String` durch `Integer` beziehungsweise `Student` ersetzt. 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. Aus diesem Grund wählt man einen neutralen Namen wie `Element`, der im Code in keiner anderen Bedeutung vorkommt. Ein Programmstück kann auch mehrere Typparameter unterschiedlicher Bedeutung enthalten.

Natürlich kann man sich Schreibaufwand ersparen, wenn man eine Kopie eines Programmstücks anfertigt und darin alle Vorkommen eines Typparameters mit Hilfe eines Texteditors durch einen Typ ersetzt. Aber dieser einfache Ansatz bereitet Probleme bei der Wartung: Nötige Änderungen des kopierten Programmstücks müssen in allen Kopien gemacht werden, was einen erheblichen Aufwand verursachen kann. Leichter geht es, wenn das Programmstück nur einmal existiert. Das ist einer der Gründe für den Einsatz von Generizität: Man schreibt ein Programmstück nur einmal und kennzeichnet Typparameter als solche. Statt einer Kopie verwendet man 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 gar keine Kopien der Programmstücke, sondern kann durch Typumwandlungen (Casts) ein und denselben Code für mehrere Zwecke – etwa Listen mit Elementen unterschiedlicher Typen – verwenden. Generizität erspart damit nicht nur Schreibarbeit, sondern kann das übersetzte Programm auch kürzer und überschaubarer machen.

3.1.2 Einfache Generizität in Java

Generische Klassen und Interfaces haben ein oder mehrere Typparameter, die in spitze 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 bezeichnet `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`.

Die generische Klasse `List<A>` in Abbildung 3.1 implementiert das Interface `Collection<A>`. Diese Klasse enthält die beiden inneren Klassen `Node` und `ListIter`, deren Objekte als Listenknoten beziehungsweise Iteratoren Verwendung finden. Der Typparameter `A` ist auch in diesen beiden Klassen sichtbar und wie ein Klassen- oder Interfacename verwendbar. Im Beispiel hat `List` nur einen Default-Konstruktor. Explizite Konstruktoren hätten wie üblich die Syntax `List(...){...}`, das heißt, es werden keine Typparameter für `List` angegeben.

Das Programmstück in Abbildung 3.2 zeigt den Umgang mit generischen Klassen. 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 dieselbe Funktionalität wie

```
public class List<A> implements Collection<A> {
    private class Node {
        private A elem;           // element in node
        private Node next = null; // next node in list
        private Node(A elem) {
            this.elem = elem;
        }
    }
    private Node head = null;  // first node of list
    private Node tail = null;  // last list node

    private class ListIter implements Iterator<A> {
        private Node p = head; // iterator position

        public A next() {      // get next list element
            if (p == null)
                return null;
            A elem = p.elem;
            p = p.next;
            return elem;
        }
        public boolean hasNext() {  // more elements?
            return p != null;
        }
    }
    public void add(A x) {     // add element to list
        if (head == null)
            tail = head = new Node(x);
        else
            tail = tail.next = new Node(x);
    }
    public Iterator<A> iterator() { // new list iter.
        return new ListIter();
    }
}
```

Abbildung 3.1: Generische Listenklasse

```
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();
    }
}
```

Abbildung 3.2: Verwendung generischer Listen

elementare Typen. Ein Nachteil ist der im Vergleich zu elementaren Werten weniger effizientere 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.

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 miteinander kompatibel.

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 erkannt und als Fehler gemeldet. Wer den Umgang mit Collections, Listen und ähnlichen Datenstrukturen ohne Unterstützung durch Generizität gewohnt ist, kennt die Probleme mangelnder statischer Typ-

sicherheit, bei der Typfehler (in Form von Typkonvertierungsfehlern) erst zur Laufzeit auftreten. Generizität kann solche dynamischen Typfehler beseitigen und gleichzeitig die Lesbarkeit von Programmen verbessern.

Nicht nur Klassen und Interfaces können generisch sein, sondern auch Methoden, wie das nächste Beispiel zeigt:

```
public interface Comparator<A> {
    int compare(A x, A y);  // result <  0 if x <  y
                            // result == 0 if x == y
                            // result >  0 if x >  y
}

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;
    }
}
```

Die Methode `compare` in `Comparator<A>` vergleicht zwei Objekte desselben 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>` kann man erkennen, 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 man die für Typparameter zu verwendenden Typen nicht explizit angeben muss:

```
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.

Zum Abschluss sei hier noch ein Beispiel für die Implementierung eines sehr einfachen Komparators gezeigt:

```
class IntComparator implements Comparator<Integer> {
    public int compare(Integer x, Integer y) {
        return x.intValue() - y.intValue();
    }
}
```

Aufgrund von Autounboxing kann man die dritte Zeile auch einfach durch `return x - y;` ersetzen. Ein Komparator für Zeichenketten wird zwar etwas komplizierter, aber nach demselben Schema aufgebaut sein.

3.1.3 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 dieser Typen bestimmte Methoden oder Variablen haben.

Schranken. Über manche Typparameter benötigt man mehr Information um auf Objekte der entsprechenden Typen zugreifen zu können. Gebundene Typparameter liefern diese Information: In Java kann man für jeden

Typparameter eine Klasse und beliebig viele Interfaces als *Schranken* angeben. 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. Man kann Instanzen des Typparameters wie Instanzen der Schranken verwenden:

```
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. implementierten. 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);  // res. <  0 if this <  that
                            // res. == 0 if this == that
                            // res. >  0 if this >  that
}
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;
    }
}
```

Die Klasse `Integer` im Beispiel ist eine vereinfachte Form der in Java standardmäßig vorhandenen Klasse gleichen Namens. `Integer` wird von `Comparable<Integer>` abgeleitet. Der Name der Klasse kommt also 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 wird nach dem formalen Modell, in dem solche Konzepte untersucht wurden, *F-gebundene Generizität* genannt [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` von `X` abgeleitet ist oder umgekehrt. Natürlich gibt es die expliziten Untertypbeziehungen, wie beispielsweise die zwischen `Integer` und `Comparable<Integer>`. Man kann Klassen wie üblich ableiten:

```
class MyList<A> extends List<List<A>> { ... }
```

Dann ist `MyList<String>` ein Untertyp von `List<List<String>>`, aber `MyList<X>` ist kein Untertyp von `List<Y>` wenn `Y` möglicherweise ungleich `List<X>` ist. Die Annahme impliziter Untertypbeziehungen ist ein häufiger Anfängerfehler. Man muss stets bedenken, dass es weder in Java noch in irgendeiner anderen Sprache, die auf F-gebundener Generizität beruht, sichere implizite Untertypbeziehungen dieser Art gibt.

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>` und `List<Square>` (entsprechend dem Beispiel in Abschnitt 2.2.3). 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 nur beim Schreiben auftauchen. 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 kann man `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 2.1.1). 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 Wildcards auch ohne Schranken verwendet werden können. 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 Mensch kann man nach komplizierten Über-

legungen zwar nachvollziehen, wie die Richtung mit den Klammerebenen zusammenhängt, aber intuitiv ist das nicht. Maschinen können damit einfacher umgehen. Am besten verlässt man sich dabei auf den Compiler, der zuverlässig warnt, wenn man einen falschen Wildcard verwendet.

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 dient manchmal als Begründung 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, ähnlich wie die Verwendung von Arrays in obigem Beispiel, in einigen Fällen nicht statisch typsicher wenn `A` ein Typparameter ist. In solchen Fällen liefert der Compiler eine Warnung. Weitere Einschränkungen der Generizität in Java gibt es bei expliziten Typkonvertierungen und dynamischen Typvergleichen, wie wir später sehen werden.

Die Kompatibilität von älteren zu neueren Java-Versionen hat noch einen kleinen Nachteil: Programme führen zur Laufzeit Typprüfungen durch, obwohl der Compiler bereits zur Übersetzungszeit zugesichert hat, dass solche Fehler gar nicht auftreten können. Daher ist mit der Verwendung von Generizität ein (sehr kleiner) Verlust an Laufzeiteffizienz verbunden. In anderen Programmiersprachen hat Generizität überhaupt keinen negativen Einfluss auf die Laufzeit.

3.2 Verwendung von Generizität im Allgemeinen

Wir wollen nun betrachten, wie man Generizität in der Praxis einsetzt. Abschnitt 3.2.1 gibt einige allgemeine Ratschläge, in welchen Fällen sich die Verwendung auszahlt. In Abschnitt 3.2.2 werden wir uns mit möglichen Übersetzungen generischer Klassen beschäftigen und einige Alternativen zur Generizität vorstellen, um ein etwas umfassenderes Bild davon zu bekommen, was Generizität leisten kann.

3.2.1 Richtlinien für die Verwendung von Generizität

Generell ist der Einsatz von Generizität immer sinnvoll, wenn er 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. Man soll 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. Praktisch 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 wird, ist nicht immer gleich zu erkennen ob ein Typ als Elementtyp angesehen werden soll oder nicht. Die Lesbarkeit und Laufzeiteffizienz wird durch die Verwendung von Generizität kaum oder überhaupt nicht beeinträchtigt. Es zahlt sich daher aus, Generizität bereits frühzeitig zu verwenden.

Es kann passieren, dass man die Sinnhaftigkeit von Typparametern erst spät erkennt. In diesen Fällen soll man das Programm gleich refaktorisieren, also die Klasse oder Methode mit Typparametern versehen. Ein Hinauszögern der Refaktorisierung führt leicht zu unnötigem Code.

Üblicher Programmcode enthält nur relativ wenige generische Containerklassen. Der Grund dafür liegt einfach darin, dass die meisten Programmierumgebungen mit umfangreichen Bibliotheken ausgestattet sind, welche die am häufigsten verwendeten, immer wieder gleich strukturierten

Klassen und Methoden bereits enthalten. Man braucht diese Klassen und Methoden also nur zu verwenden statt sie neu schreiben zu müssen.

> **Faustregel:** Klassen und Methoden in Bibliotheken sollten generisch sein.

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, nicht-generischen Java-Versionen verwendet werden, und generisches Java kann mit nicht-generischen Klassen umgehen.

Abfangen erwarteter Änderungen. Mittels Generizität kann man erwartete Programmänderungen vereinfachen. Das gilt vor allem auch für Typen von formalen Parametern, die sich entsprechend dem Ersetzbarkeitsprinzip nicht beliebig ändern dürfen. Man soll von Anfang an Typparameter für die Typen formaler Parameter verwenden, wenn man erwartet, dass sich diese Typen im Laufe der Zeit ändern. Das gilt auch dann, wenn es sich nicht offensichtlich um Elementtypen in Containerklassen handelt. 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:** Man soll Typparameter als Typen formaler Parameter verwenden, wenn Änderungen der Parametertypen absehbar sind.

Beispielsweise schreiben wir eine Klasse, die Konten an einem Bankinstitut repräsentiert. Nehmen wir an, unsere Bank kennt derzeit nur Konten über Euro-Beträge. Trotzdem ist es sinnvoll, sich beim Erstellen der Klasse nicht gleich auf Euro-Konten festzulegen, sondern für die Währung einen Typparameter zu verwenden, für den neben einem Euro-Typ auch ein US-Dollar-Typ eingesetzt werden kann (falls die Währung typrelevant ist). Bei Einsatz derselben Software in einem anderen Land erleichtert dies die Programmänderung: Bestehender Code, der sich nur auf Typparameter bezieht, bleibt unverändert, und nur die Teile, die sich auf eine bestimmte Währung beziehen, müssen verändert werden. Von einer Währung erwarten wir, dass sie bestimmte Eigenschaften erfüllt. Wir werden im Konto

daher gebundene Typparameter verwenden, und alle Währungen werden Untertypen eines bestimmten Typs, sagen wir `Currency`, sein.

Ein Konto ist auch eine Art von Container. Es geht darum, dass dies nicht von vorneherein sichtbar sein muss. Die erwartete Änderung eines Parametertyps, die von einem anderen Programmteil ausgeht, lässt vermuten, dass es sich um einen Container handeln könnte.

Untertyprelationen und Generizität sind in Java eng 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. Man soll stets überlegen, ob man eine Aufgabe besser durch Ersetzbarkeit, durch Generizität, oder (häufig sinnvoll) eine Kombination aus beiden Konzepten löst.

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 verhindern.

Verwendbarkeit. Generizität und Untertypbeziehungen sind oft gegeneinander austauschbar. Das heißt, man kann 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 man an folgenden zwei Beispielen sieht:

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 denselben Typ haben. Solche Listen sind *homogen*. Ohne Generizität ist es nicht möglich, eine solche Klasse zu schreiben. Zwar kann man auch ohne Generizität Listen erzeugen, die Elemente beliebiger Typen enthalten können, aber es ist nicht

statisch sichergestellt, dass alle Elemente in der Liste denselben Typ haben. Daher kann man 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 kann man 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 man mit Generizität oder Untertypbeziehungen alleine nicht machen kann. Sie zeigen damit auf, in welchen Fällen man Generizität bzw. Untertypen zur Erreichung des Ziels unbedingt braucht.

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, sollte man 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 kommen Programmierer(innen) manchmal auf die Idee, stets Generizität einzusetzen, aber dynamisches Binden nur dort zuzulassen, wo es unumgänglich ist. Da Generizität und Untertypbeziehungen oft gegeneinander austauschbar sind, kann man das im Prinzip machen. 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:** Man soll Effizienzüberlegungen in der Entscheidung, ob man Generizität oder Untertypbeziehungen einsetzt, beiseitelassen.

Solche Optimierungen auf der untersten Ebene sind nur etwas für Experten, die Details ihrer Compiler und ihrer Hardware sehr gut kennen, und auch dann sind die Optimierungen nicht portabel. Viel wichtiger ist es, auf die Einfachheit und Verständlichkeit zu achten. Wenn Effizienz entscheidend ist, muss man vor allem die Effizienz der Algorithmen betrachten.

Natürlichkeit. Häufig bekommt man auf die Frage, ob man in einer bestimmten Situation Generizität oder Subtyping einsetzen soll, die Antwort, dass der *natürlichere* Mechanismus am besten geeignet sei. Für erfahrene Leute ist diese Antwort 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 Erfahrung sieht man kaum, was natürlicher ist. Es zahlt sich in jedem Fall aus genau zu überlegen, was man mit Generizität erreichen will und erreichen kann. Wenn man sich zwischen Generizität und Subtyping entscheiden soll, ist es angebracht, auch eine Kombination von Generizität und Subtyping ins Auge zu fassen. Erst wenn diese Überlegungen zu keinem eindeutigen Ziel führen entscheidet man sich für die natürlichere Alternative.

3.2.2 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 3.3 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 3.1.1 argumentiert. Dem Nachteil einer größeren Anzahl übersetzter Klassen und Methoden 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 brauchen keine Typumwandlungen und damit zusammenhängende Überprüfungen durchgeführt zu werden. Außerdem sind auf jede übersetzte Klasse eigene Optimierungen anwendbar, die von den Typen abhängen. Daher haben Programme bei heterogener Übersetzung etwas bessere Laufzeiteffizienz. Heterogene Übersetzung wird beispielsweise für *Templates* in C++ verwendet.

Große Unterschiede zwischen Programmiersprachen gibt es im Zusammenhang mit gebundener Generizität. Java und Eiffel verlangen, dass Programmierer(innen) eine Schranke vorgeben und nur Untertypen der Schranke den Typparameter ersetzen können. Programmierer(innen) müssen die geeigneten Klassenhierarchien erstellen. Vor allem bei Verwendung von Typen aus vorgefertigten Bibliotheken, deren Untertypbeziehungen zueinander nicht mehr ohne Weiteres im Nachhinein festlegbar sind, ist das ein bedeutender Nachteil. In Java wird dieser Nachteil dadurch vermindert, dass mehrere Interfaces als Schranken angegeben werden können.

Durch die heterogene Übersetzung von Templates brauchen Programmierer(innen) in C++ keine Schranken anzugeben, 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. Unterschiedliche Typen, die einen Typparameter ersetzen, brauchen keinen gemeinsamen Obertyp zu haben. Schranken auf Typparametern sind jedoch hilfreich, indem sie Client-Code vom Server-Code entkoppeln, sodass Clients keine versteckten Details des Servers kennen müssen. In C++ gibt es derzeit nichts Enstprechendes. 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)

Folgendes Beispiel zeigt einen Ausschnitt aus einer generischen Klasse in C++:

```
template<class 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<class 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. Für künftige Standards werden Wege gesucht, solche Eigenschaften unter Beibehaltung von Templates auszudrücken.

Eine weitere flexible Variante für den Umgang mit gebundenen Typparametern bietet Ada: Als Schranke gibt man Eigenschaften an, welche die Typen, die Typparameter ersetzen, erfüllen müssen. Beispielsweise wird angegeben, dass eine Methode mit bestimmten Parametern nötig ist. Auf den ersten Blick ist diese Variante genauso flexibel wie Templates in C++. Jedoch kann man bei jeder Verwendung einer generischen Einheit getrennt angeben, welche Methode eines Typs für eine bestimmte Eigenschaft verwendet werden soll. Methoden werden wie Typen als generische Parameter behandelt. Ada hat aber den Nachteil, dass generische Methoden nicht einfach aufgerufen werden können, sondern zuvor daraus nicht-generische Methoden abgeleitet werden müssen. Gründe dafür haben nichts mit der Spezifikation von Schranken zu tun.

(für Interessierte)

Eine generische Funktion in Ada [19] 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.

In Java kann man Ähnliches wie in Ada durch Komparatoren (siehe Abschnitt 3.1.2) auch erzielen, jedoch nur mit hohem Aufwand.

3.3 Typabfragen und Typumwandlungen

Prozedurale und funktionale Programmiersprachen unterscheiden streng zwischen Typinformationen im Programm, die nur dem Compiler zum Zeitpunkt der Übersetzung zur Verfügung stehen, und dynamischen Programminformationen, die während der Programmausführung verwendet werden können. Es gibt in diesen Sprachen keine dynamische Typinformation. Im Gegensatz dazu wird in objektorientierten Programmiersprachen dynamische Typinformation 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 einer bestimmten Typs ist, sowie zur überprüften Umwandlung des deklarierten Typs. Wir wollen nun den Umgang mit dynamischer Typinformation untersuchen. In Abschnitt 3.3.1 finden sich allgemeine Hinweise dazu. In den nächsten beiden Abschnitten werden spezifische Probleme durch Verwendung dynamischer Typinformation gelöst: Abschnitt 3.3.2 behandelt simulierte Generizität und Abschnitt 3.3.3 kovariante Probleme.

3.3.1 Verwendung dynamischer Typinformation

Jedes Objekt hat eine Methode `getClass`, welche die Klasse des Objekts als Ergebnis zurückgibt. Diese Methode bietet die direkteste Möglichkeit des Zugriffs auf den dynamischen Typ. Davon wird aber eher selten Gebrauch gemacht, da diese Typinformation an Programmstellen, an denen

die Klasse eines Objekts nicht ohnehin bekannt ist, kaum benötigt wird. In Spezialfällen ist `getClass` jedoch sehr wertvoll.

Etwas häufiger möchte man 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`, `Studienassistent` oder `Werkstudent` ist (entsprechend der Typhierarchie aus Abschnitt 2.1.2). Die Abfrage liefert `false` wenn `p` gleich `null` oder von einem anderen dynamischen Typ ist. Solche dynamischen Typabfragen können, wie alle Vergleichsoperationen, an beliebigen Programmstellen stehen, an denen Boolesche Ausdrücke erlaubt sind.

Mittels Typabfragen lässt sich zwar der Typ eines Objektes zur Laufzeit bestimmen, aber Typabfragen reichen nicht aus, um Eigenschaften des dynamisch ermittelten Typs zu nutzen. 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 – siehe Abschnitt 3.3.3) Beispiel modifiziert ein Beispiel aus Abschnitt 2.1.1:

```
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 herum sind weitere Klammern nötig, damit klar ist, dass der Typ von `p` umgewandelt werden soll, nicht der Typ des Ergebnisses von `p.z == z` wie in `(Point3D)p.z == z`.

Verbesserungen von `Point2D` und `Point3D` folgen in Abschnitt 3.3.3[1]. 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 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 muss man sehr vorsichtig sein. Man kann sich in solchen Fällen nicht mehr auf die Intuition verlassen, die man sich im Laufe der Zeit in der objektorientierten Programmierung angeeignet hat. 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. Bei der Typumwandlung erfolgt eine Überprüfung, die sicherstellt, dass das Objekt den gewünschten Typ hat. Sind die Bedingungen nicht erfüllt, erfolgt eine Ausnahmebehandlung.

Dynamische Typabfragen und Typumwandlungen sind sehr mächtige Werkzeuge. Man kann damit einiges machen, was sonst nicht oder nur

[1] Versuchen Sie Fehler in dieser Lösung selbst zu finden. Die Verwendung von dynamischer Typinformation und 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 man den Fehler finden, indem man Zusicherungen auf `Point3D` und `Point2D` explizit macht.

sehr umständlich machbar wäre. Allerdings kann die falsche Verwendung von dynamischen Typabfragen und Typumwandlungen 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 man sich vermutlich nach wie vor auf statische Typsicherheit verlässt.

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 allgemein bekannt, lassen sich `switch`- und `if`-Anweisungen durch dynamisches Binden ersetzen. Das soll man 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 `doSomethingOfAnyType`, die Un-

terklassen `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 (recht aufwendige) Lösung kann man parallel zur unveränderbaren Klassenhierarchie eine gleich strukturierte Hierarchie aufbauen, deren Klassen (Wrapper-Klassen) die zusätzlichen Methoden beschreiben. Typabfragen ermöglichen eine einfachere Lösung.

- 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.

- Der deklarierte Typ von `x` kann sehr viele Untertypen haben. Wenn `doSomething` nicht in einer gemeinsamen Oberklasse in der Bedeutung von `doSomethingOfAnyType` implementierbar ist, beispielsweise weil der deklarierte Typ von `x` ein Interface ist, muss `doSomething` in vielen Klassen auf gleiche Weise implementiert werden. Das bedeutet einen Mehraufwand für die Wartung. Der Grund dafür liegt in der fehlenden Unterstützung der Mehrfachvererbung in Java. Durch Refaktorisierung und Verwendung geeigneter Entwurfsmuster lassen sich diese Probleme abschwächen oder vermeiden. Typabfragen ermöglichen eine einfache Lösung.

> **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 Fließkommazahl wird beispielsweise durch Runden gegen Null eine ganze Zahl. Dabei kann Information verlorengehen, 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.

3.3.2 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 3.2.2 gesehen haben:

- Alle Ausdrücke in spitzen Klammern werden weggelassen.
- Jedes andere 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 – siehe weiter unten.

Für das Beispiel aus Abschnitt 3.1.2 wird folgender Code generiert:

```
public interface Collection {
    void add(Object elem);
    Iterator iterator();
}
public interface Iterator {
    Object next();
    boolean hasNext();
}
```

```
public class List implements Collection {
    private class Node {
        private Object elem;
        private Node next = null;
        private Node(Object elem) {
            this.elem = elem;
        }
    }
    private Node head = null;
    private Node tail = null;

    private class ListIter implements Iterator {
        private Node p = head;

        public Object next() {
            if (p == null)
                return null;
            Object elem = p.elem;
            p = p.next;
            return elem;
        }
        public boolean hasNext() {
            return p != null;
        }
    }
    public void add(Object x) {
        if (head == null)
            tail = head = new Node(x);
        else
            tail = tail.next = new Node(x);
    }
    public Iterator iterator() {
        return new ListIter();
    }
}
```

In diesem übersetzten Code wurden keine Typumwandlungen eingeführt, da nirgends ein Typparameter durch einen Typ ersetzt wurde. Solche Typumwandlungen kommen beispielsweise durch die Übersetzung des Codes in Abbildung 3.2 ins Spiel:

```
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` sind eigentlich unnötig, entsprechen aber dem Schema. Sie 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. Hier haben diese Typumwandlungen keinerlei Bedeutung, da sie weder die deklarierten Typen ändern noch überladene Methoden vorhanden sind. 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 man sie auch selbst ohne Unterstützung durch den Compiler durchführen kann. Das ist vermutlich ein Grund dafür, dass Java ursprünglich keine Generizität unterstützte. 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 Ausnahmebehandlungen 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 liegt also in erster Linie in der besseren Lesbarkeit und höheren Typsicherheit.

Viele 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, und 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:** Man soll nur sichere Formen der Typumwandlung 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 man das Programmstück so schreibt, als ob man Generizität verwenden würde, dieses Programmstück händisch auf mögliche Typfehler, die bei Verwendung von Generizität zu Tage treten, untersucht und dann die homogene Übersetzung durchführt.

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 zur Auswahl zwischen überladenen Methoden manchmal sinnvoll – 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 sicherer Formen der Typumwandungen impliziert, dass es einen sinnvollen Programmzweig geben muss, der im Falle des Scheiterns des Typvergleichs ausgeführt wird. Wenn man in einem alternativen Zweig einfach nur eine Ausnahmebehandlung anstösst, kann man nicht mehr von einer sicheren Typumwandlung sprechen. Leider erweisen sich gerade falsche Typannahmen in alternativen Zweigen zu Typabfragen als häufige Fehlerquelle.

> **Faustregel:** Bei Zutreffen des zweiten Punktes ist besonders darauf zu achten, dass alle Annahmen im alternativen Zweig (bei Scheitern des Typvergleichs) in Zusicherungen stehen.

Außerdem gibt es oft mehrere alternative Zweige, die sich in geschachtelten `if`-Anweisungen zeigen. Aufgrund der damit verbundenen Wartungsprobleme sollte man auf solche Lösungen verzichten.

Bei Zutreffen des dritten Punktes 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 kann die Intuition manchmal in die Irre führen, 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 macht man hier?
}
List<String> bad(Object o) {
    if (o instanceof List<String>)      // error
        return (List<String>)o;         // error
    else { ... }          // Was macht man 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 Ausnahmebehandlung 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 that) {
        if (!(that instanceof List))
            return false;
        Iterator<A> xi = this.iterator();
        Iterator yi = ((List)that).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>` generische Klassen, `List` und `Iterator` sind entsprechende übersetzte Klassen. Die übersetzten Klassen nennt man *Raw-Types.* Man spricht 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 man bei Typdeklarationen die spitzen Klammern anzugeben vergisst. Da solche Typen vom Compiler als Raw-Types missverstanden werden, erfolgt keine Typüberprüfung der Generizität, und man hält das falsche Programm für korrekt. Daher ist es besonders wichtig, stets auf die richtige Verwendung der spitzen Klammern zu achten.

3.3.3 Kovariante Probleme

In Abschnitt 2.1 haben wir gesehen, dass Typen von Eingangsparametern nur kontravariant sein können. Kovariante Eingangsparametertypen verletzen das Ersetzbarkeitsprinzip. In der Praxis wünscht man sich manchmal gerade kovariante Eingangsparametertypen. Entsprechende Aufgabenstellungen nennt man *kovariante Probleme.* Zur Lösung kovarianter Probleme bieten sich dynamische Typabfragen und Typumwandlungen an, wie folgendes Beispiel zeigt:

```
abstract class Futter { ... }
class Gras extends Futter { ... }
class Fleisch extends Futter { ... }
abstract class Tier {
    public abstract void friss(Futter x);
}
class Rind extends Tier {
    public void friss(Gras x) { ... }
    public void friss(Futter x) {
        if (x instanceof Gras)
            friss((Gras)x);
        else
            erhoeheWahrscheinlichkeitFuerBSE();
    }
}
class Tiger extends Tier {
    public void friss(Fleisch x) { ... }
    public void friss(Futter x) {
        if (x instanceof Fleisch)
            friss((Fleisch)x);
        else
            fletscheZaehne();
    }
}
```

In `Rind` und `Tiger` ist `friss` überladen, es gibt also mehrere Methoden desselben Namens. Die Methoden mit `Futter` als Parametertyp überschreiben jene aus `Tier`, während die anderen nicht in `Tier` vorkommen. Es wird die Methode ausgeführt, deren formaler Parametertyp der spezifischste Obertyp des *deklarierten* Argumenttyps ist. Im Beispiel wird der gewünschte deklarierte Argumenttyp durch Typumwandlung bestimmt.

Es ist ganz natürlich, `Gras` und `Fleisch` als Untertypen von `Futter` anzusehen. `Gras` und `Fleisch` sind offensichtlich einander ausschließende Spezialisierungen von `Futter`. Ebenso sind `Rind` und `Tiger` Spezialisierungen von `Tier`. Es entspricht der praktischen Erfahrung, dass Tiere im Allgemeinen Futter fressen, Rinder aber nur Gras und Tiger nur Fleisch. Als Parametertyp der Methode `friss` wünscht man sich daher in `Tier` `Futter`, in `Rind` `Gras` und in `Tiger` `Fleisch`.

Genau diese Beziehungen in der realen Welt sind aber nicht typsicher darstellbar. Zur Lösung des Problems bietet sich eine erweiterte Sicht der Beziehungen in der realen Welt an: Auch einem Rind kann man Fleisch und einem Tiger Gras zum Fressen anbieten. Wenn man das macht, muss man aber mit unerwünschten Reaktionen der Tiere rechnen. Obiges Programmstück beschreibt entsprechendes Verhalten: Wenn dem Tier geeignetes Futter angeboten wird, erledigen die überladenen Methoden `friss` mit den Parametertypen `Gras` beziehungsweise `Fleisch` die Aufgaben. Sonst führen die überschriebenen Methoden `friss` mit dem Parametertyp `Futter` Aktionen aus, die vermutlich nicht erwünscht sind.

Durch Umschreiben des Programms kann man zwar Typabfragen und Typumwandlungen vermeiden, aber die unerwünschten Aktionen bei kovarianten Problemen bleiben erhalten. Die einzige Möglichkeit besteht darin, kovariante Probleme zu vermeiden. Beispielsweise reicht es, `friss` aus `Tier` zu entfernen. Dann kann man zwar `friss` nur mehr mit Futter der richtigen Art in `Rind` und `Tiger` aufrufen, aber man kann Tiere nur mehr füttern, wenn man die Art der Tiere und des Futters genau kennt.

Faustregel: Kovariante Probleme soll man vermeiden.

(für Interessierte)

Kovariante Probleme treten in der Praxis so häufig auf, dass einige Programmiersprachen teilweise Lösungen dafür anbieten. Zunächst betrachten wir Eiffel: In dieser Sprache sind kovariante Eingangsparametertypen durchwegs erlaubt. Wenn die Klasse `Tier` die Methode `friss` mit dem Parametertyp `Futter` enthält, können die überschriebenen Methoden in den Klassen `Rind` und `Tiger` die Parametertypen `Gras` und `Fleisch` haben. Dies ermöglicht eine natürliche Modellierung kovarianter Probleme. Weil dadurch aber 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

`friss` mit einer falschen Futterart aufgerufen, kommt es zu einer Ausnahmebehandlung zur Laufzeit. Tatsächlich ergibt sich dadurch derselbe Effekt, als ob man in Java ohne vorhergehende Überprüfung den Typ des Arguments von `friss` auf die gewünschte Futterart umwandeln würde. Von einer echten Lösung des Problems kann man daher nicht sprechen.

Einen anderen Ansatz bieten *virtuelle Typen,* die derzeit in keiner gängigen Programmiersprache verwendet werden [21, 18]. Man kann virtuelle Typen als geschachtelte Klassen wie in Java ansehen, die jedoch, anders als in Java, in Unterklassen überschreibbar sind. Die beiden Klassenhierarchien mit `Tier` und `Futter` als Wurzeln werden eng verknüpft: `Futter` ist in `Tier` enthalten. In `Rind` ist `Futter` mit einer neuen Klasse überschrieben, welche die Funktionalität von `Gras` aufweist, und `Futter` in `Tiger` mit einer Klasse der Funktionalität von `Fleisch`. Statt `Gras` und `Fleisch` schreibt man dann `Rind.Futter` und `Tiger.Futter`. Der Typ `Futter` des Parameters von `friss` bezieht sich immer auf den lokal gültigen Namen, in `Rind` also auf `Rind.Futter`. Noch immer muss man `friss` in `Rind` mit einem Argument vom Typ `Rind.Futter` und in `Tiger` mit einem Argument vom Typ `Tiger.Futter` aufrufen; die Art des Tieres muss also mit der Art des Futters übereinstimmen, und der Compiler muss die Übereinstimmung überprüfen können. Aber wenn `Tier` (und daher auch `Rind` und `Tiger`) eine Methode hat, die ein Objekt vom Typ `Futter` als Ergebnis liefert, kann man das Ergebnis eines solchen Methodenaufrufs als Argument eines Aufrufs von `friss` in demselben Objekt verwenden. Dabei braucht man die Art des Tieres nicht zu kennen und ist trotzdem vor Typfehlern sicher. Eine ähnliche Methode kann man durch kovariante Ergebnistypen auch in Java schreiben; sie liefert in `Rind` ein Ergebnis vom Typ `Gras` und in `Tiger` eines vom Typ `Fleisch`. Aber in Java kann man 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 inhaltliche Vorteil virtueller Typen. Ein weiterer Vorteil könnte auf der psychologischen Ebene liegen, da der Umgang damit sehr natürlich wirkt, sodass tiefgehende Schwierigkeiten im Umgang mit kovarianten Problemen nicht in den Vordergrund treten. Die Praxisrelevanz dieser Vorteile ist derzeit mangels Erfahrungen allerdings kaum abschätzbar.

Binäre Methoden. Einen häufig vorkommenden Spezialfall kovarianter Probleme stellen binäre Methoden dar. Wie in Abschnitt 2.1 eingeführt, hat eine binäre Methode mindestens einen formalen Parameter, dessen

Typ stets gleich der Klasse ist, welche die Methode enthält. Im Prinzip kann man binäre Methoden auf dieselbe Weise behandeln wie alle anderen kovarianten Probleme. Das heißt, man könnte (wie in Abschnitt 3.3.1) dynamische Typabfragen verwenden um den dynamischen Parametertyp zu bestimmen. Das ist aber problematisch, wie wir gleich sehen werden. Hier ist eine weitere, 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 x == that.x && y == that.y;
    }
}
class Point3D extends Point {
    private int x, y, z;
    protected boolean uncheckedEqual(Point p) {
        Point3D that = (Point3D)p;
        return x==that.x && y==that.y && z==that.z;
    }
}
```

Anders als in allen 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 direkt 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 3.3.1 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 3.3.1 ist falsch, da ein Aufruf von `equal` in `Point2D` mit einem Argument vom Typ `Point3D` als Ergebnis `true` liefern kann.

(für Interessierte)

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 Ausnahmebehandlung eingeleitet. Methoden wie `equal` in obigem Beispiel sind damit sehr einfach programmierbar. Falls die zu vergleichenden Objekte unterschiedliche Typen haben, kommt es zu einer Ausnahmebehandlung, die an geeigneten Stellen abgefangen werden kann.

3.4 Überladen versus 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 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 spricht man 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 3.4.1 werden wir uns die Unterschiede deshalb deutlich vor Augen führen. In Abschnitt 3.4.2 werden wir sehen, dass man Multimethoden auch in Sprachen wie Java recht einfach simulieren kann.

3.4.1 Deklarierte versus dynamische Argumenttypen

Folgendes Beispiel soll klar machen, 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 3.3.3:

```
Rind   rind = new Rind();
Futter gras = new Gras();
rind.friss(gras);          // Rind.friss(Futter x)
rind.friss((Gras)gras);    // Rind.friss(Gras x)
```

Wegen dynamischem Binden werden die Methoden `friss` auf jeden Fall in der Klasse `Rind` ausgeführt, unabhängig davon, ob `rind` als `Tier` oder `Rind` deklariert ist. Der Methodenaufruf in der dritten Zeile führt die überladene Methode mit dem Parameter vom Typ `Futter` aus, da `gras` mit dem Typ `Futter` deklariert ist. Für die Methodenauswahl ist es unerheblich, dass `gras` tatsächlich ein Objekt von `Gras` enthält; der dynamische Typ von `gras` ist `Gras`, da `gras` direkt vor dem Methodenaufruf mit einem Objekt von `Gras` initialisiert wird. Es zählt aber nur der deklarierte Typ. Der Methodenaufruf in der vierten Zeile führt die überladene Methode mit dem Parameter vom Typ `Gras` aus, weil der deklarierte Typ von `gras` wegen der Typumwandlung an dieser Stelle `Gras` ist. Typumwandlungen ändern ja den deklarierten Typ eines Ausdrucks.

Häufig weiß man in solchen Fällen, dass `futter` ein Objekt von `Gras` enthält und nimmt an, dass die Methode mit dem Parameter vom Typ `Gras` gewählt wird. Diese Annahme ist aber falsch! Es wird stets der deklarierte Typ verwendet, auch wenn man den dynamischen Typ kennt.

Nehmen wir an, die erste Zeile des Beispiels sehe so aus:

```
Tier rind = new Rind();
```

Wegen dynamischen Bindens würde `friss` weiterhin in `Rind` 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 `Tier` gesucht, nicht in `Rind`. In `Tier` ist `friss` nicht überladen, sondern es gibt nur eine Methode mit einem Parameter vom Typ `Futter`. Daher wird in `Rind` auf jeden Fall die Methode mit dem Parameter vom Typ `Futter` ausgeführt, unabhängig davon, ob der deklarierte Typ des Arguments `Futter` oder (nach einem Cast) `Gras` 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 `friss` in `Rind` 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 `Gras` ist, wird im Endeffekt immer die Methode mit dem Parametertyp `Gras` aufgerufen. Es ist empfehlenswert, Überladen nur so zu verwenden.

> **Faustregel:** Man soll Überladen nur so verwenden, dass es keine Rolle spielt, ob bei der Methodenauswahl deklarierte oder dynamische Typen der Argumente verwendet werden.

Für je zwei überladene Methoden gleicher Parameteranzahl

- soll es zumindest eine Parameterposition geben, an der sich die Typen der Parameter unterscheiden, nicht in Untertyprelation zueinander stehen und auch keinen gemeinsamen Untertyp haben,
- oder alle Parametertypen der einen Methode sollen Obertypen der Parametertypen der anderen Methode sein, und bei Aufruf der einen Methode soll nichts anderes gemacht werden, als auf die andere Methode zu verzweigen, falls die entsprechenden dynamischen Typen der Argumente dies erlauben.

Unter diesen Bedingungen ist die strikte Unterscheidung zwischen deklarierten und dynamischen Typen bei der Methodenauswahl nicht wichtig.

Das Problem der Verwechslung von dynamischen und deklarierten Typen könnte man nachhaltig lösen, indem man zur Methodenauswahl generell die dynamischen Typen aller Argumente verwendet. Statt überladener Methoden hätte man dann Multimethoden. Würde Java Multimethoden unterstützen, könnte man die Klasse `Rind` im Beispiel aus Abschnitt 3.3.3 kürzer und ohne dynamische Typabfragen und -umwandlungen schreiben:

```
class Rind extends Tier {
    public void friss(Gras x) { ... }
    public void friss(Futter x) {
        erhoeheWahrscheinlichkeitFuerBSE();
    }
} // Achtung: In Java ist diese Lösung falsch !!
```

Die Abfrage, ob `x` den dynamischen Typ `Gras` hat, hätte man sich erspart, da `friss` mit dem Parametertyp `Futter` bei Multimethoden nur aufgerufen wird, wenn der dynamische Argumenttyp nicht `Gras` 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 `friss` 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 `Gras` ist. Die Multimethodenvariante von `friss` kann sogar effizienter sein als die Variante mit Überladen, wenn der statische Typ des Arguments ein Untertyp von `Gras` ist, nicht jedoch der deklarierte Typ. Die Laufzeiteffizienz ist daher kaum ein Grund für fehlende Multimethoden in einer Programmiersprache.

Unter der höheren Komplexität der Methodenauswahl versteht man etwas anderes als die Laufzeiteffizienz: Für Programmierer(innen) 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 `friss` mit einem Argument vom Typ `Gras` (oder einem Untertyp davon) aufrufen, sind die Parametertypen beider Methoden mit dem Argumenttyp kompatibel. Da `Gras` spezieller ist als `Futter`, wird die Methode mit dem Parametertyp `Gras` ausgeführt. Diese Regel ist für die Methodenauswahl aber nicht hinreichend wenn Multimethoden mehrere Parameter haben, wie folgendes Beispiel zeigt:

```
public void frissDoppelt(Futter x, Gras y) { ... }
public void frissDoppelt(Gras x, Futter y) { ... }
```

Mit einem Aufruf von `frissDoppelt` mit zwei Argumenten vom Typ `Gras` 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 [20]) 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. Programmierer(innen) müssen eine zusätzliche Methode

```
public void frissDoppelt(Gras x, Gras 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 Programmierer(innen) eher Multimethoden oder eher Überladen haben wollen, bleibt offen.

In Java kommt es zu Fehlern, wenn man unbewusst Überladen statt Überschreiben verwenden, wenn man also eine Methode überschreiben will, die überschreibende Methode sich aber in den Parametertypen von der zu überschreibenden Methode unterscheidet. Beispielsweise werden in C++ die häufigsten derartigen Probleme vom Compiler erkannt, weil es Einschränkungen beim Überladen ererbter Methoden gibt. Nicht so in Java. Man muss speziell darauf achten, dass Parametertypen der überschreibenden und überschriebenen Methode wirklich gleich sind. Nur Ergebnistypen dürfen ab Java 1.5 kovariant verschieden sein.

3.4.2 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 ist es nicht schwer, mehrfaches dynamisches Binden durch wiederholtes einfaches Binden zu simulieren. Wir nutzen mehrfaches dynamisches Binden für das Beispiel aus Abschnitt 3.3.3 und eliminieren damit dynamische Typabfragen und Typumwandlungen:

```
public abstract class Tier {
    public abstract void friss(Futter futter);
}
public class Rind extends Tier {
    public void friss(Futter futter) {
        futter.vonRindGefressen(this);
    }
}
public class Tiger extends Tier {
    public void friss(Futter futter) {
        futter.vonTigerGefressen(this);
    }
}
public abstract class Futter {
    abstract void vonRindGefressen(Rind rind);
    abstract void vonTigerGefressen(Tiger tiger);
}
public class Gras extends Futter {
    void vonRindGefressen(Rind rind) { ... }
    void vonTigerGefressen(Tiger tiger) {
        tiger.fletscheZaehne();
    }
}
public class Fleisch extends Futter {
    void vonRindGefressen(Rind rind) {
        rind.erhoeheWahrscheinlichkeitFuerBSE();
    }
    void vonTigerGefressen(Tiger tiger) { ... }
}
```

Die Methoden `friss` in `Rind` und `Tiger` rufen Methoden in `Futter` 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 `Gras` und `Fleisch` gibt es nicht nur *eine* entsprechende Methode, sondern je eine für Objekte von `Rind` und `Tiger`. Bei einem Aufruf von `tier.friss(futter)` wird zweimal dynamisch gebunden. Das erste dynamische Binden unterscheidet zwischen Objekten von `Rind` und `Tiger` und spiegelt sich im Aufruf von `vonRindGefressen` und

`vonTigerGefressen` wider. Ein zweites dynamisches Binden unterscheidet zwischen Objekten von `Gras` und `Fleisch`. In den Unterklassen von `Futter` sind insgesamt vier Methoden implementiert, die alle möglichen Kombinationen von Tierarten mit Futterarten abdecken.

Statt `vonRindGefressen` und `vonTigerGefressen` hätten wir auch einen gemeinsamen Namen wählen können (= Überladen), da sich die Typen der formalen Parameter eindeutig unterscheiden.

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 man mehrfaches dynamisches Binden durch wiederholtes einfaches dynamisches Binden ersetzen kann. Bei Multimethoden mit mehreren Parametern muss entsprechend oft dynamisch gebunden werden. Sobald man den Übersetzungsschritt verstanden hat, kann man ihn ohne große intellektuelle Anstrengungen für vielfaches dynamisches Binden durchführen.

Diese Lösung kann auch dadurch erzeugt worden sein, dass in der ursprünglichen Lösung aus Abschnitt 3.3.3 `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 in unserem Beispiel entspricht dem *Visitor-Pattern*, einem klassischen Entwurfsmuster – siehe Kapitel 5. Klassen wie `Futter` nennt man *Visitorklassen,* die darin enthaltenen Methoden wie `vonRindGefressen` sind *Visitormethoden,* und Klassen wie `Tier` heißen *Elementklassen.* Visitor- und Elementklassen sind oft gegeneinander austauschbar. Beispielsweise könnten die eigentlichen Implementierungen (Visitormethoden) in den Tier-Klassen stehen, die nur 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 $(i = 1 \ldots k)$ 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 $i = 2 \ldots k-1$ gleichzeitig Element- und Visitorklassen sind.) Für $k = 4$ und $n_1, \ldots, n_4 = 10$ 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 muss man 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 könnte man statt `vonRindGefressen` eine Methode `vonVegetarierGefressen` 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. Man kann 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.

Lösungen mit Multimethoden oder dynamischen Typabfragen haben meist den Vorteil, dass die Anzahl der nötigen Methoden deutlich kleiner bleibt. Dies trifft besonders dann zu, wenn die Multimethode aus einigen speziellen Methoden mit uneinheitlicher Struktur der formalen Parametertypen und ganz wenigen allgemeinen Methoden, die den großen Rest behandeln, auskommt. Bei Verwendung dynamischer Typabfragen ist in diesen Fällen der große Rest in wenigen `else`-Zweigen versteckt.

3.5 Wiederholungsfragen

1. Was ist Generizität? Wozu verwendet man Generizität?

2. Was ist gebundene Generizität? Was kann man mit Schranken auf Typparametern machen, das ohne Schranken nicht geht?

3. In welchen Fällen soll man Generizität einsetzen, in welchen nicht?

4. Was bedeutet statische Typsicherheit in Zusammenhang mit Generizität, dynamischen Typabfragen und Typumwandlungen?
5. Welche Arten von Generizität kann man hinsichtlich ihrer Übersetzung und ihrem Umgang mit Schranken unterscheiden? Welche Art wird in Java verwendet, und wie flexibel ist diese Lösung?
6. Was sind (gebundene) Wildcards als Typen in Java? Wozu kann man sie verwenden?
7. Wie kann man Generizität simulieren? Worauf verzichtet man, wenn man Generizität nur simuliert?
8. Was wird bei der heterogenen bzw. homogenen Übersetzung von Generizität genau gemacht?
9. Was muss der Java-Compiler überprüfen um sicher zu sein, dass durch Generizität keine Laufzeitfehler entstehen?
10. Welche Möglichkeiten für dynamische Typabfragen gibt es in Java, und wie funktionieren sie genau?
11. Was wird bei einer Typumwandlung in Java umgewandelt – der deklarierte, dynamische oder statische Typ? Warum?
12. Welche Gefahren bestehen bei Typumwandlungen?
13. Wie kann man dynamische Typabfragen und Typumwandlungen vermeiden? In welchen Fällen kann das schwierig sein?
14. Welche Arten von Typumwandlungen sind sicher? Warum?
15. Was sind kovariante Probleme und binäre Methoden? Wie kann man mit ihnen umgehen oder sie vermeiden?
16. Wie unterscheidet sich Überschreiben von Überladen, und was sind Multimethoden?
17. Wie kann man Multimethoden simulieren? Welche Probleme können dabei auftreten?
18. Was ist das Visitor-Entwurfsmuster?
19. Wodurch ist Überladen problematisch, und in welchen Fällen ergeben sich kaum Probleme?

4 Kreuz und quer

Nicht immer folgen Berechnungen den linear einfach verständlichen Prinzipien der strukturierten Programmierung und nicht alle Gegebenheiten lassen sich so orthogonal in Modularisierungseinheiten faktorisieren wie wir uns das wünschen. Manchmal können wir kreuz und quer verlaufende Berechnungen, Verbindungen zwischen den Daten sowie Programmänderungen nicht vermeiden. Darum geht es in diesem Kapitel. Wir beschäftigen uns mit einigen ausgewählten Programmierkonzepten, die uns dabei helfen auch in solchen Situationen strukturiert vorzugehen.

Der Schwerpunkt liegt auf Konzepten, die in der Java-Programmierung zum Einsatz kommen. Ausnahmebehandlungen als ein häufig verwendetes Konzept zum Durchbrechen des üblichen Programmablaufs betrachten wir in Abschnitt 4.1. Auch die nebenläufige Programmierung schafft durch die notwendige Synchronisation zahlreiche Querverbindungen, die nur schwer in den Griff zu bekommen sind, wie wir in Abschnitt 4.1.2 sehen werden. Annotationen, die eine Parametrisierung auch in aus Sicht der Wartung schwierigen Situationen erlauben, behandeln wir in Abschnitt 4.3. Schließlich beschäftigen wir uns mit der aspektorientierten Programmierung in AspectJ, einem Programmierstil und Werkzeug speziell für den Umgang mit Querverbindungen, in Abschnitt 4.4.

4.1 Ausnahmebehandlung

Ausnahmebehandlungen dienen vor allem dem Umgang mit unerwünschten Programmzuständen. Zum Beispiel werden in Java Ausnahmebehandlungen 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. In Abschnitt 4.1.1 gehen wir auf Ausnahmebehandlungen in Java ein und geben danach in Abschnitt 4.1.2 einige Hinweise auf den sinnvollen Einsatz von Ausnahmebehandlungen.

4.1.1 Ausnahmebehandlung in Java

Eigentlich sollten Teilnehmer an „Objektorientierte Programmiertechniken“ mit den Grundlagen der Ausnahmebehandlung in Java schon vertraut sein. Da es diesbezüglich immer wieder Schwierigkeiten gibt und um Missverständnissen hinsichtlich der Terminologie vorzubeugen sind hier die wichtigsten Fakten (nocheinmal) zusammengestellt.

Ausnahmen sind in Java gewöhnliche Objekte, die über spezielle Mechanismen als Ausnahmen verwendet werden. Alle Objekte von `Throwable` sind dafür verwendbar. Praktisch verwendet man 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 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 überprüfbare, meist vom System vorgegebene Ausnahmen. 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 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 wird. 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 formalen 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 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. Man weiß daher nicht, welche Befehle bereits ausgeführt wurden und ob die Werte in den Variablen konsistent sind. Man muss `catch`-Klauseln so schreiben, dass sie mögliche Inkonsistenzen beseitigen. Variablen, die in einem `try`-Block deklariert wurden, sind in entsprechenden `catch`-Blöcken nicht sicht- und verwendbar.

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 aber weggelassen werden, wie das folgende Beispiel zeigt:

```
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.

4.1.2 Einsatz von Ausnahmebehandlungen

Ausnahmen werden in folgenden Fällen eingesetzt:

Unvorhergesehene Programmabbrüche: Wird eine Ausnahme nicht abgefangen, kommt es zu einem Programmabbruch. Die entsprechende 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 möglichst 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 Programmfluss 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 soll man Ausnahmen und Ausnahmebehandlungen nur in echten Ausnahmesituationen und sparsam einsetzen.

Bei Auftreten einer Ausnahme wird der normale Programmfluss durch eine Ausnahmebehandlung ersetzt. Während der normale Programmfluss 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, das viele Ausnahmebehandlungen enthält, ist daher oft nur 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:** Man soll 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 kann man vermeiden, indem man die Ausnahme, dass `x` gleich `null` ist, abfängt:

```
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. wenn die Implementierung von `getNext` geändert wird.

> **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(Exception x) { ... }
```

Die zweite Variante funktioniert natürlich nur wenn `T1` bis `Tn` Unterklassen von `Exception` sind. 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 sollten 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 würde ebenso nicht-lokale Abhängigkeiten im Programm erzeugen.

Man kann darüber diskutieren, ob es vernünftiger ist, statt überprüfter Zusicherungen Untertypen von `RuntimeException` zu verwenden. Dadurch könnte man `throws`-Klauseln vermeiden, mit denen man gerade in den Fällen, in denen Zusicherungen besonders sinnvoll sind, nur schwer umgehen kann. Vorteile bieten überprüfte Zusicherungen ja vor allem in Fällen, in denen man Zusicherungen vermeiden soll. Aus diesem Grund gibt es überprüfte Zusicherungen im Wesentlichen nur (mehr) in Java.

4.2 Nebenläufige Programmierung

Bisher sind wir implizit davon ausgegangen, dass jedes Programm schrittweise, ein Befehl nach dem anderen, in *einem Thread* ausgeführt wird. Java bietet, wie die meisten anderen Programmiersprachen auch, minimale Unterstützung für die *nebenläufige Programmierung,* bei der *mehrere Threads* gleichzeitig nebeneinander laufen und Befehle aus verschiedenen Threads entweder (auf mehreren Prozessor-Kernen) tatsächlich oder (auf nur einem Prozessor-Kern) scheinbar gleichzeitig ausgeführt werden.

Die Programmierung wird durch nebenläufige Threads aufwendiger, da wir gelegentlich neue Threads erzeugen und kontrollieren müssen, vor allem aber, da wir nebenläufige Threads *synchronisieren* müssen um zu verhindern, dass durch gleichzeitige Zugriffe die aus Variablen gelesenen und in Variablen geschriebenen Werte inkonsistent werden.

4.2.1 Thread-Erzeugung und Synchonisation in Java

Folgendes Beispiel soll ein Synchronisationsproblem demonstrieren:

```
public class Zaehler {
    private int i = 0, j = 0;
    public void schnipp() { i++; j++; }
}
```

Die Variablen `i` und `j` sollten stets die gleichen Werte enthalten. Wenn wir jedoch in mehreren nebenläufigen Threads `schnipp` in demselben Objekt von `Zaehler` 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 `schnipp` 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.

Die Ausführungen von `schnipp` lassen sich in Java sehr einfach synchronisieren, indem wir `schnipp` als `synchronized` definieren:

```
public synchronized void schnipp() { i++; j++; }
```

In jedem Objekt wird zu jedem Zeitpunkt höchstens eine `synchronized` Methode ausgeführt. Wenn mehrere Threads `schnipp` annähernd gleichzeitig aufrufen, werden alle bis auf einen Thread solange blockiert, bis dieser eine aus `schnipp` zurückkehrt. Dann darf der nächste Thread `schnipp` ausführen, und so weiter. Die oben beschriebenen Synchronisationsprobleme sind damit beseitigt. Die Methode wird *atomar*, also wie eine nicht weiter in Einzelteile zerlegbar Einheit ausgeführt.

So wie `schnipp` sollen alle Methoden, die auf Objekt- oder Klassenvariablen zugreifen, in nebenläufigen Programmen oder Programmteilen als `synchronized` definiert sein, um Inkonsistenzen bei Variablenzugriffen zu verhindern. Das gilt vor allem für ändernde Zugriffe wie in obigem 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).

Synchronisierte Methoden sollen nur kurz laufen, da sie sonst die Wahrscheinlichkeit für das Blockieren anderer Threads sowie die durchschnittliche Dauer der Blockaden erhöhen. Überlegungen zur Synchronisation sind aufwendig. 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 der Forderung nach kurz laufenden Methoden. 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 schnipp() {
    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 `schnipp` 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, brauchen sich rekursive Aufrufe nicht mehr um Synchronisation zu kümmern.

Zugriffe auf eine Variable, die als `volatile` deklariert ist, sind seit Java 5 atomar. Mit Hilfe der Klasse `AtomicInteger` könnte man obiges Beispiel auch ohne synchronisierte Blöcke und ohne Locking schreiben.

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 Druckertreiber {
    private boolean online = false;
    public synchronized void drucke (String s) {
        while (!online) {
            try { wait(); }
            catch(InterruptedException ex) { return; }
        }
        ... // schicke s zum Drucker
    }
    public synchronized void onOff() {
        online = !online;
        if (online) notifyAll();
    }
    ...
}
```

Die Methode `drucke` stellt sicher, dass `online` den Wert `true` hat, bevor das Argument an den Drucker weitergeleitet wird. Falls `online` nicht `true` ist, wird `wait` aufgerufen. Diese in `Object` vordefinierte Methode blockiert den aktuellen Thread so lange, bis dieser explizit wieder

aufgeweckt wird, oder mit einem entsprechenden Argument für eine bestimmte Zeit. Die Überprüfung der Bedingung erfolgt in einer Schleife, da auch 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. Entsprechend dem Java-Standard ist immer, auch ohne Grund damit zu rechnen, dass ein Thread aus dem Wartezustand aufwacht. Daher erfolgen Überprüfungen solcher Bedingungen fast immer in Schleifen. Ebenso ist es fast immer notwendig, die Ausnahme `InterruptedException` abzufangen, die vom System bei vorzeitiger Beendigung des wartenden Threads ausgelöst wird.

Die Methoden `wait`, `notify` und `notifyAll` können nur innerhalb einer synchronisierten Methode oder eines synchronisierten Blocks ausgeführt werden. Bei Ausführung von `wait` wird der aktive Thread in eine Warteliste gehängt, die dem aktuellen Objekt gehört, welches den Lock hält. Der Lock wird daraufhin freigegeben, damit andere Threads auf das Objekt zugreifen können. Über `notifyAll` werden alle Threads in der Warteliste des aktuellen Objekts wieder aufgeweckt. Nach Freigabe des Locks wird ein aufgeweckter Thread nach dem anderen weiter ausgeführt, wobei während der Ausführung der Lock auf den entsprechenden Thread gesetzt ist. Die Methode `notify` ähnelt `notifyAll`, jedoch wird nur *ein* Thread aufgeweckt, falls einer vorhanden ist. Es kann passieren, dass `notify` nichts bewirkt, weil alle Threads kurzfristig aufgeweckt waren, aber unmittelbar danach wieder in die Warteliste kommen. Daher ist `notify` mit Vorsicht zu genießen, also eigentlich zu meiden.

Objekte der folgenden Klasse erzeugen nach Aufruf von `run` in einer Endlosschleife immer wieder neue Zeichenketten und schicken diese an den im Konstruktor festgelegten Druckertreiber:

```
public class Produzent implements Runnable {
    private Druckertreiber t;
    public Produzent(Druckertreiber t) { this.t = t; }
    public void run() {
        String s = ....
        for (;;) {
            ...          // produziere neuen Wert in s
            t.drucke(s); // schicke s an Druckertreiber
        }
    }
}
```

Das vordefinierte Interface `Runnable` spezifiziert nur `run`. Objekte von Klassen wie `Produzent`, die `Runnable` implementieren, können wie in folgendem Codestück zur Erzeugung neuer Threads verwendet werden:

```
Druckertreiber t = new Druckertreiber(...);
for (int i = 0; i < 10; i++) {
    Produzent p = new Produzent(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`, und der Aufruf von `start` bewirkt die Ausführung von `p.run()` im neuen Thread. Im Beispiel produzieren zehn Objekte von `Produzent` 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 „deprecated“ sind und nicht mehr verwendet werden sollten.

4.2.2 Nebenläufigkeit in der Praxis

In der Praxis verwendet man die grundlegenden Sprachkonzepte der Nebenläufigkeit nicht sehr häufig. Gründe dafür sind einerseits prinzipielle Schwierigkeiten im Umgang mit Nebenläufigkeit, andererseits aber auch eine Reihe vorgefertigter Lösungen für die häufigsten Aufgaben, die man nur mehr zu verwenden braucht. Damit wird die Entwicklung nebenläufiger Programme auf eine höhere Ebene verschoben.

Die wichtigsten vorgefertigten Lösungen findet man in den Java-Paketen `java.util.concurrent` sowie `java.util.concurrent.atomic` und `java.util.concurrent.locks`. Zum Teil handelt es sich dabei um gut durchdachte und effiziente Implementierungen von Programmteilen, die man mit entsprechendem Wissen selbst schreiben könnte, zum Teil (vor allem in `java.util.concurrent.atomic`) aber auch um Lösungen, welche heute übliche Hardwareunterstützung für Parallelität nutzbar macht. Hinter diesen Lösungen stecken bekannte Verfahren im Umgang mit Nebenläufigkeit und Parallelität, die in anderen Lehrveranstaltungen thematisiert werden. Wir wollen hier nur exemplarisch einige wenige Möglichkeiten aufzeigen.

Aufgaben und Threads. Vor allem aus funktionalen Programmen kennt man 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 man das Ergebnis ganz normal aus der Variablen auslesen. Wenn man von der Variablen liest 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 heute oft verschiedenste Aufgaben (*Tasks*) erledigt werden, die unabhängig voneinander irgendwann, aber möglichst effizient ablaufen sollen. Die Darstellung einzelner Elemente in einem Webbrowser ist ein Beispiel dafür. Aus Effizienzgründen ist es nicht immer 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 in regelmäßigen Abständen wiederholte Ausführen bestimmter Aufgaben.

Thread-sichere Datenstrukturen. Eine Reihe von Klassen im Java-Paket `java.util.concurrent` stellt synchronisierte Varianten üblicher Datenstrukturen dar. Zum Beispiel 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. Auf Objekte von `HashMap` darf dagegen nicht gleichzeitig von mehreren Threads aus zuge-

griffen werden (würde zu Fehlern führen), aber bei nur einem Thread ist `HashMap` natürlich effizienter. Es gibt noch weitere Varianten: So 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 älteren Java-Versionen gibt es statt `HashMap` nur die Klasse `Hashtable`. Diese Klasse wird heute selten verwendet, da `HashMap` einen etwas größeren Funktionsumfang hat und ohne Nebenläufigkeit effizienter ist. Allerdings ist `Hashtable` von Haus aus synchronisiert und bietet bei Nebenläufigkeit ähnliche Effizienz wie ein Objekt von `HashMap`, das über `synchronizedMap` synchronisiert wurde. Aus praktischer Sicht sind oft die kleinen Unterschiede im Funktionsumfang ausschlaggebend dafür, welche Klasse wir einsetzen. Beispielsweise kann man in einem Objekt von `HashMap` auch den Wert `null` ablegen, in einem Objekt von `Hashtable` aber nicht. Die Klasse `ConcurrentHashMap` ist 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 zugreifen. Durch Methoden wie `synchronizedMap` und `synchronizedList` in der Klasse `Collections` können diese Datenstrukturen aber mit Synchronisation ausgestattet werden, sodass sie auch in einer nebenläufigen Umgebung einsetzbar sind. Allerdings sind diese Datenstrukturen bei gleichzeitigen Zugriffen durch viele Threads nur wenig effizient. Vor allem sind Iteratoren über diesen Datenstrukturen bei Nebenläufigkeit nicht robust, das heißt, nach Änderungen der Datenstrukturen funktionieren sie nicht mehr vernünftig. In `java.util.concurrent` findet man Varianten dieser Datenstrukturen, die auch bei gleichzeitigen Zugriffen durch viele Threads noch effizient sind und etwas robustere Iteratoren bieten. Allerdings unterscheiden sich die nebenläufigen Datenstrukturen in vielen Details von den Varianten aus `java.util`. Darin spiegelt sich die Tatsache wider, dass man in der nebenläufigen und parallelen Programmierung zum Teil andere Datenstrukturen und Algorithmen braucht als in der sequentiellen Programmierung. Wer effiziente nebenläufige Programme schreiben will, muss also auf jeden Fall umlernen, auch wenn nur vorgefertigte Programmteile zum Einsatz kommen.

Vorgehensweise. Bei der Suche nach passenden Zerlegungen einer großen Aufgabe in nebenläufig ausführbare Teilaufgaben lässt man sich meist von der Unabhängigkeit der Teilaufgaben voneinander leiten. Wenn die Teilaufgaben nicht voneinander abhängen, ist beispielsweise `Executor` sinnvoll mit bestmöglicher Effizienz einsetzbar. Die Teilaufgaben sollen möglichst nicht auf gemeinsame Daten zugreifen müssen.

Oft lässt sich dieses Ziel nicht ganz erreichen. Dann muss man dafür sorgen, dass es möglichst wenige gleichzeitige Zugriffe auf gemeinsame Daten gibt, vor allem möglichst wenige gleichzeitige Schreibzugriffe. Klassen wie `ConcurrentHashMap` können in diesem Fall einen Ausweg bieten. Allerdings muss dabei sichergestellt sein, dass die gemeinsamen Daten keine Einschränkungen in der Ausführungsreihenfolge der Teilaufgaben bedingen, die Daten also nicht auf komplexe Weise voneinander abhängen. Man weiß ja nicht, wann genau welche Teilaufgabe ausgeführt wird.

Sind Abhängigkeiten in der Ausführungsreihenfolge unvermeidlich, wird es schwierig. Man kann versuchen, Struktur in die Einschränkungen bezüglich der Ausführungsreihenfolge zu bringen, sodass beispielsweise Klassen wie `Phaser` einsetzbar werden, die es erlauben, Teilaufgaben in mehreren Phasen auszuführen. Sobald man jedoch auf solche Formen der Synchronisation angewiesen ist, kann man nicht mehr so einfach die Zuteilung der Teilaufgaben an Threads einem `Executor` überlassen, sondern muss sich selbst darum kümmern. Jeder Versuch der Kontrolle der Ausführungsreihenfolge lässt den Schwierigkeitsgrad der Lösung rasant ansteigen. Auch wenn man die zahlreichen Synchronisationsmöglichkeiten auf höherer Ebene nützt, erreicht man bald eine Komplexität, die nahe an der Komplexität liegt, die man bei reiner Nutzung der primitiven Sprachkonstrukte der Nebenläufigkeit hätte. Ganz kann man heute auf die primitiven Sprachkonstrukte trotz zahlreicher und gut durchdachter vorgefertigter Lösungen noch nicht verzichten.

Ein Ausweg steht sehr oft offen: Wenn es zu schwierig wird bestimmte Aufgaben nebenläufig zu lösen, kann man sie noch immer sequentiell lösen. Obwohl es stark vereinfachend klingt, liegt im Kern dieser Aussage viel Potential für eine gute Zerlegung einer Aufgabe in Teilaufgaben. Wenn die nötige Synchronisation zu aufwendig wäre, würde die nebenläufige Ausführung der Teilaufgaben weder hinsichtlich der Nutzung paralleler Hardware noch hinsichtlich der Einfachheit der gesamten Programmstruktur irgendwelche Vorteile bringen. Es zahlt sich aus, in einigen Bereichen auf Nebenläufigkeit zu verzichten. Dafür kann man an anderen Stellen im Programm die Möglichkeiten der Nebenläufigkeit umso effektiver nutzen.

4.2.3 Synchronisation und Objektorientiertheit

In der nebenläufigen Programmierung muss man stets auf Synchronisation achten, auch wenn man vorgefertigte Lösungen verwendet. Beispielsweise muss man vermeiden, dass zwei möglicherweise nebenläufig ausgeführte Tasks auf gemeinsame Daten in einer nicht Thread-sicheren Datenstruktur zugreifen. Man hat zwar versucht, über die Namensgebung Struktur in die Thread-Sicherheit von Klassen zu bringen, aber ganz ist das nicht gelungen. Beispielsweise sind `Vector` und `Hashtable` sicher, die ähnlichen Klassen `LinkedList` und `HashMap` aber nicht. Beim Programmieren muss man daher sehr viel Sorgfalt walten lassen.

Auch zu viel Synchronisation macht sich negativ bemerkbar: Die gleichzeitige Ausführung von Threads wird verhindert und die Laufzeit des Programms möglicherweise verlängert. In Extremfällen wird die Ausführung so stark verzögert, dass überhaupt kein Fortschritt mehr möglich ist. Gefürchtet sind *Deadlocks,* das sind zyklische Abhängigkeiten zwischen zwei oder mehr Threads: Beispielsweise möchte ein Thread p, der bereits den Lock auf ein Object x hält, auch den Lock auf ein anderes Objekt y und wartet darauf, dass ein anderer Thread q den Lock auf y freigibt. Wenn q zufällig auf das Freiwerden des Locks auf x wartet bevor jener auf y freigegeben werden kann, befinden sich p und q in einem Deadlock und warten ewig aufeinander. Übliche Techniken zur Vermeidung von Deadlocks bestehen in der Verhinderung solcher Zyklen durch eine lineare Anordnung aller Objekte im System; Locks dürfen nur in dieser Reihenfolge angefordert werden. Wenn x in der Anordnung vor y steht, ist zwar Thread p erlaubt, nicht jedoch q, da ein Thread, der bereits einen Lock auf y hält, keinen Lock auf x mehr anfordern darf. Leider ist eine lineare Anordnung in der Praxis viel einschränkender als man oft glaubt: Dadurch werden alle Arten von zyklischen Strukturen verhindert, bei deren Abarbeitung Synchronisation nötig sein könnte. Oft nimmt man für solche Strukturen die Gefahr von Deadlocks in Kauf. Neben Deadlocks gibt es eine Reihe weiterer Ursachen für unerwünschte (vorübergehende oder dauerhafte) gegenseitige Behinderungen von Threads durch Synchronisation. Eigenschaften, die die Abwesenheit solcher unerwünschter gegenseitiger Behinderungen betreffen (neben Deadlocks beispielsweise auch Livelocks und Starvation), nennt man zusammengefasst *Liveness-Properties.*

Manchmal wird empfohlen, Liveness-Properties wie geforderte nichtfunktionale Eigenschaften eines Programms zu behandeln. Das bedeutet, dass man sich beim Schreiben des Programms zunächst nicht darum küm-

mert, sondern erst durch ausgiebiges Testen Verletzungen dieser Eigenschaften zu finden versucht. Treten beim Testen keine Probleme (mehr) auf, nimmt man die Eigenschaften als erfüllt an.

Heute gibt es Werkzeuge (auch für Java), die in der Lage sind, bestimmte Eigenschaften von Programmen zu beweisen. So kann man oft auch beweisen, dass keine Deadlocks auftreten. Leider können diese Werkzeuge nicht alles. Da Livelocks und Starvation viele unterschiedliche Ursachen und Auswirkungen haben können, gibt es dafür auch keine klaren formalen Definitionen, sodass sie formalen Beweisen kaum zugänglich sind. Daher bleibt nur das Testen.

Vorgefertigten Lösungen für die nebenläufige Programmierung beruhen zum Großteil auf bekannten Techniken, die nicht oder kaum anfällig für Verletzungen der Liveness-Properties sind. Genau deswegen sollte man sie bevorzugen, auch wenn sie die Programmstruktur einschränken. In den vorgefertigten Implementierungen hat man sich natürlich sehr bemüht, solche Probleme zu vermeiden. 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 ganz unterschiedlicher Programmteile. Man muss also das gesamte Programm betrachten, nicht nur kleine Teile. Am besten folgt man Empfehlungen, die aus Erfahrungen resultieren. Empfehlungen zur nebenläufigen Java-Programmierung im herkömmlichen Stil sind in [23] zu finden, aktuellere Programmiertechniken in [12].

Objektorientierte Konzepte. Das von Java unterstützte Basiskonzept der nebenläufigen Programmierung, das *Monitor-Konzept,* ist schon recht alt [15, 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 man sich selbst kümmern muss. Vorgefertigte Lösungen haben an dieser Problematik im Grunde nichts geändert. Daher ist es auch heute noch schwierig, gute nebenläufige objektorientierte Programme zu schreiben.

Um Synchronisation in Untertypbeziehungen einzubeziehen muss man vor allem Client-kontrollierte History-Constraints berücksichtigen. Synchronisation bewirkt ja Einschränkungen auf der Reihenfolge, in der Methoden abgearbeitet werden, und 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. Und 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 die Bedingungen zu halten. Dort wo komplexe Synchronisation notwendig ist, sollte man deswegen unter den heutigen Gegebenheiten auf Ersetzbarkeit verzichten. Die wichtigste Empfehlung bei der Planung der Synchronisation ist daher, diese so lokal wie möglich zu halten und so einfach wie möglich zu gestalten.

Abhängigkeiten, die durch die notwendige Synchronisation in die Software eingeführt werden, stehen auch der direkten Wiederverwendung von Code, also 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 schlicht und einfach daraus, dass man bei der Faktorisierung der Software nach objektorientierten Gesichtspunkten anders vorgehen muss als bei der Zerlegung von Aufgaben in nebenläufige Teilaufgaben. Wenn man die objektorientierten Gesichtspunkte in den Mittelpunkt stellt, ergeben sich häufig so starke Abhängigkeiten zwischen den Teilaufgaben, dass Nebenläufigkeit kaum sinnvoll einsetzbar ist. Andererseits führt eine Zerlegung nach nebenläufigen Gesichtspunkten leicht zu sehr niedrigem Klassenzusammenhalt und längerfristig hohem Wartungsaufwand wegen nötiger Refaktorisierungen. Nur selten ist eine Zerlegung nach beiden Gesichtspunkten gut.

Wie die Terminologie zeigt – Nachrichten werden gesendet und Empfangen – war die objektorientierte Programmierung ursprünglich (in den 1970er- und 1980er-Jahren) durchaus mit Konzepten der verteilten und damit auch nebenläufigen Programmierung kompatibel. Man betrachtete Objekte als Prozesse oder Threads, die mit anderen Objekten tatsächlich durch das Senden und Empfangen von Nachrichten Informationen austauschten. Bekannt ist das *Actor-Modell* [3, 14] als Basis für die Kom-

munikation zwischen Objekten und deren Synchronisation. Auf damaliger Hardware hat sich ein derartiges Modell jedoch noch nicht effizent genug implementieren lassen, sodass die Semantik solange vereinfacht wurde, bis daraus ein Vorläufer des heute bekannten Objektmodells ohne Nebenläufigkeit entstanden ist. Später, als man doch Nebenläufigkeit benötigte, hat man einfach die Konzepte aus der prozeduralen Programmierung übernommen. Viele Probleme, die sich aus dieser Entscheidung ergeben, wurden erst in jüngerer Zeit offensichtlich. Das ist ein Grund dafür, warum in den letzten Jahren das Actor-Modell plötzlich wieder an Popularität gewann und heute Unterstützung für die Actor-Programmierung in vielen Programmiersprachen, auch unter Java, angeboten wird. Die meisten derartigen Lösungsansätze sind jedoch noch nicht ausgereift und effizient genug für einen Einsatz im größeren Maßstab. Aber diese Entwicklung zeigt, dass die nebenläufige objektorientierte Programmierung, wie sie heute meist betrieben wird, noch lange nicht der Weisheit letzter Schluss ist. In diesem Bereich dürfen wir in absehbarer Zukunft noch viel erwarten.

4.3 Annotationen und Reflexion

Die Idee hinter Annotationen ist einfach: Man versieht unterschiedliche Programmteile mit Markierungen, die man Annotationen nennt, und das Laufzeitsystem sowie Entwicklungswerkzeuge (zur Übersetzungszeit) 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 erscheint, so komplex sind Details der Umsetzung. Einerseits sollte das Hinzufügen von Annotationen zu Java die Syntax nicht allzusehr 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 4.3.1 untersuchen wir, wie diese Details in Java gelöst wurden, und in Abschnitt 4.3.2 betrachten wir einige Anwendungsbeispiele.

4.3.1 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 hat man die Syntax von Interface-Definitionen übernommen und leicht 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 kann man zu Klassen, Interfaces und Enums hinzufügen um auf Korrekturen hinzuweisen. Das könnte so aussehen:

```
@BugFix(who="Kaspar", date="1.10.2013", level=3,
        bug="class unnecessary and maybe harmful",
        fix="content of class body removed")
public class Buggy { }
```

Die einzelnen Einträge in der Definition der Annotation beschreiben also 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 immer 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. Eine syntaktische Vereinfachung erlaubt das Weglassen des Namens 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 kann man 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 das API 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 genauso 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.

Weiters sind in der Definition einer Annotation noch 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 einer Unterklasse als annotiert gilt.

Man kann 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 braucht man bei der Verwendung kein Argument für `comment` angeben. Das ist besonders dann sinnvoll, wenn

die Zeile erst später hinzugefügt wird, das heißt, wenn `@BugFix` schon vorher verwendet wurde. Alle diese Stellen der Verwendung braucht man durch die Default-Belegung nicht unbedingt zu ändern.

Verwendung zur Laufzeit. Wurde über `@Retention(RUNTIME)` festgelegt, dass eine Art von Annotationen 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();
}
```

Man kann 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 Klasse eines Objekts als Instanz der Klasse `Class` ermittelt, so enthält die statische Variable `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 man genau weiß, auf welche Annotation man zugreifen möchte. Durch `getAnnotations` kann man alle Annotationen der Klasse gleichzeitig auslesen, ohne deren genaue 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 kann man meist auch nur dann weiterarbeiten, wenn man weiß, welche Annotationen man haben möchte. Auf die Datenfelder kann man ja nur zugreifen, wenn man vorher einen Cast auf den richtigen Typ der Annotation macht.

Die Technik, mit der man zur Laufzeit auf Annotationen zugreifen kann, nennt man *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 man sich die Details der Klasse ansehen kann. So wie man über `getAnnotation` und `getAnnotations` Informationen über Annotationen bekommt, kann man sich 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. Andere 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. Diese Klasse bietet ebenso Methoden um Annotationen und viel anderes Nützliches abzufragen. Man kann etwa die Typen der formalen Parameter und des Ergebnisses sowie Annotationen auf den formalen Parametern abfragen. Über `invoke` kann man die Methode auch aufrufen. Dabei muss man den Empfänger der Nachricht sowie passende Argumente bereitstellen. Wenn der Empfänger oder ein Argument einen unpassenden Typ hat, oder wenn der Aufruf der Methode wegen eingeschränkter Sichtbarkeit gar nicht erlaubt ist, bekommt man eine Ausnahme. Abgesehen davon, dass die Überprüfungen, die üblicherweise der Compiler macht, erst zur Laufzeit passieren, hat ein solcher Aufruf dieselbe Semantik wie ein normaler Methodenaufruf. Es wird auch dynamisch gebunden.

Ähnliches gilt für Informationen über Variablen in Objekten des Typs `Field`. Man kann über Reflexion die Werte sichtbarer Variablen lesen und schreiben, und natürlich kann man auch Annotationen abfragen.

Reflexion ist in der Programmierung eine äußerst mächtige Technik. Man kann sehr flexibel fast alles zur Laufzeit entscheiden, was üblicher-

weise schon beim Programmieren 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 man diese Freiheit auf unkontrollierte Weise nutzt.

4.3.2 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 verwendet man 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 verwendet man `@Override`. Statt dieser Annotation wäre auch ein Modifier sinnvoll gewesen, aber aufgrund der geschichtlichen Entwicklung hat sich eine Annotation angeboten. Man kann jede Annotation als Modifier sehen, den ein (Pre-)Compiler oder das Laufzeitsystem versteht.

Manchmal stolpert man über eine `@Deprecated`-Annotation, mit der Programmelemente, die man nicht mehr verwenden sollte, gekennzeichnet werden. 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, sollte man von der Verwendung solcher Annotationen Abstand nehmen. Warnungen haben ja einen Grund, den man nicht vernachlässigen sollte. Es kommt gar nicht so selten vor, dass man eine solche Annotation „vorübergehend“ in den Code schreibt, weil man sich erst später um ein Problem kümmern möchte, und dann darauf vergisst. Auch wenn

man genau weiß, 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.

Reflexion. Reflexion ist eine Variante der *Metaprogrammierung,* 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. Man weiß, dass man mit diesen Techniken in speziellen Fällen sehr viel erreichen kann. Man kennt aber auch die Gefahren. Daher versucht man solche Techniken in der tagtäglichen Programmierung zu vermeiden und greift nur darauf zurück, wenn man keine einfachere Lösung findet.

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 Parametern 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 man vielleicht nicht rechnet, ganz zu schweigen von der Gefahr, dass wir nicht wissen, was die mit `invoke` aufgerufenen Methoden machen. Möglicherweise ist die Methode nicht `public`, verlangt (andere) Argumente, oder existiert gar nicht. In diesen Fällen werden Ausnahmen geworfen, die wir irgendwie abfangen 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 könnte man 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 Oberflächen 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“, deren Werte von außen zugreifbar sind. Solche Properties führt man ein, indem man entsprechende Setter- bzw. Getter-Methoden schreibt. 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 Oberfläche 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“ geht man ähnlich vor.

In seltenen Fällen benötigt man Information, die nicht leicht über Reflexion verfügbar ist. Dafür gibt es z.B. die `@ConstructorProperties`-Annotation: In übersetzten Klassen kann man kaum feststellen, welche Parameter eines Konstruktors welcher Property entsprechen. 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.

In der Java-EE (Enterprise-Edition) verwendet man 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 verwendet man `@TransactionAttribute`

um festzulegen, ob eine Methode innerhalb einer Transaktion zu verwenden ist. Mittels `@Stateful` bzw. `@Stateless` legt man neben weiteren Eigenschaften fest, ob eine „Session-Bean“ – ein für die Dauer einer Sitzung existierendes Objekt – zustandsbehaftet sein soll oder nicht. Mit EJB muss man sich schon intensiv auseinandersetzen, bevor man es sinnvoll verwenden kann. Über Annotationen ergibt sich beinahe schon eine eigene Sprache innerhalb von Java.

4.4 Aspektorientierte Programmierung

Ein wesentliches Designziel bei der Programmierung ist die Aufteilung der Funktionalität eines Programms auf einzelne Module (*separation of concerns*). In der objektorientierten Programmierung funktioniert die Aufteilung gut für die Kernfunktionalitäten (*core concerns*), die sich leicht in eine Klasse packen lassen. Oft gibt es aber Anforderungen, die alle Bereiche eines Programms betreffen (*cross-cutting concerns*, Querschnittsfunktionalitäten). Die aspektorientierte Programmierung kapselt Verhalten, das mehrere Klassen betrifft, in Aspekten. Ein Aspekt beschreibt dabei an einer Stelle sowohl eine Funktionalität, als auch alle Stellen im Programm, an denen diese Funktionalität angewendet werden soll.

Die Idee der aspektorientierten Programmierung soll anhand des Beispiels einer Bankensoftware erläutert werden. Die Anforderungen enthalten Funktionalitäten für die Verwaltung von Konten, Geldtransfers, Krediten, Wertpapieren und Ähnlichem. Bei diesen Kernfunktionalitäten ist die Modularisierung im Großen und Ganzen offensichtlich. Weiters wird eine sichere Zugriffskontrolle für alle Komponenten der Software gefordert. Die Grundfunktionalität der Zugriffskontrolle kann einfach in einem Modul zusammengefasst werden, die Aufrufe dieser Grundfunktionen werden sich aber ungekapselt über das gesamte restliche Programm verteilen. Die Zugriffskontrolle ist ein *Cross-Cutting-Concern.* In konventioneller objektorientierter Programmierung verhindern Cross-Cutting-Concerns eine saubere Modularisierung, die Verständlichkeit leidet, Wiederverwendbarkeit und einfache Wartung werden unmöglich gemacht. Ein mehrdimensionales Problem muss eindimensional gelöst werden.

Die aspektorientierte Programmierung bietet eine Lösung. Wir verwenden AspectJ (`www.eclipse.org/aspectj`), es gibt aber für fast alle anderen Programmiersprachen ebenso aspektorientierte Erweiterungen (z.B. AspectL für Lisp und LOOM.NET für C#). In AspectJ kapselt ein

Aspekt vollständig alle Teile von Cross-Cutting-Concerns, die Implementierung der Funktionalität und Spezifikationen der Stellen, wo diese angewendet werden soll. Dazu werden in AspectJ folgende Begriffe verwendet:

Join-Point Ein Join-Point ist eine identifizierbare Stelle während einer Programmausführung wie z.B. der Aufruf einer Methode oder der Zugriff auf ein Objekt.

Pointcut Ein Pointcut ist ein Programmkonstrukt, das einen Join-Point auswählt und kontextabhängige Information dazu sammelt, z.B. die Argumente eines Methodenaufrufs oder das Zielobjekt.

Advice Ein Advice ist jener Programmcode, der vor (`before()`), um (`around()`) oder nach (`after()`) dem Join-Point ausgeführt wird.

Aspect Ein Aspekt ist wie eine Klasse das zentrale Element in AspectJ. Ein Aspekt enthält alle Deklarationen, Methoden, Pointcuts und Advices.

Anhand des folgenden Programmstückes soll gezeigt werden, welche Stellen in einem Programm Join-Points sein können.

```
01                public class Test{
02 Methodenausf.     public static void main(String[]a) {
03 Konstruktoraufruf    Point pt1 = new Point(0,0);
04 Methodenaufruf       pt1.incrXY(3,6);
05                    }
06                }
07
08 Klasseninit    public class Point {
09 Objektinit        private int x;
10                   private int y;
11                   public Point(int x, int y) {
12 Feldzugr.(write)     this.x = x;
13                      this.y = y;
14                   }
15                   public void incrXY(int dx, int dy){
16 Feldzugr.(read)      x = this.x + dx;
17                      y =+ dy;
18                   }
19                }
```

In Zeile `02` ist ein möglicher Join-Point die Ausführung des Beginns der Methode `main`, in Zeile `03` der Aufruf des Konstruktors `Point` und in Zeile `12` der schreibende Zugriff auf des Feld `this.x`.

In einem Pointcut werden Join-Points definiert. In einem Advice können anonyme Pointcuts definiert werden, meistens werden aber benannte Join-Points verwendet. Die Syntax sieht folgendermaßen aus:

[*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 Argumente Untertyp von EventListener sind.

```
public pointcut AddListener() :
   call(* javax..*.add*Listener(EventListener+))
```

Bei der Signatur steht ein

- `*` für eine beliebige Anzahl von Zeichen außer einem `.`,
- `..` für eine beliebige Anzahl jedes beliebigen Zeichens,
- `+` für jeden Untertyp eines Typs,
- `!` (Negation) für alle Join-Points außer dem Spezifizierten,
- `||` für die Vereinigungsmenge von Join-Points und
- `&&` für die Durchschnittsmenge von Join-Points.

Es gibt mehrere Arten von Pointcut-Typen:

`execution(MethodSignature)` Ausführung einer Methode

`call(MethodSignature)` Aufruf einer Methode

`execution(ConstructorSignature)` Ausführung eines Konstruktors

`call(ConstructorSignature)` Aufruf eines Konstruktors

`get(FieldSignature)` lesender Feldzugriff

`set(FieldSignature)` schreibender Feldzugriff

staticinitialization(TypeSignature) Initialisierung einer Klasse

preinitialization(ConstructorSignature) erster Schritt der Initialisierung eines Objekts

initialization(ConstructorSignature) Initialisierung eines Objekts

handler(TypeSignature) Ausführung einer Ausnahmenbehandlung

Folgende Pointcuts sind kontrolflussbasiert:

cflow(Pointcut) alle dem Kontrollfluss entsprechenden Join-Points eines Pointcuts inklusive des äußersten

cflowbelow(Pointcut) alle Join-Points innerhalb eines Pointcuts, die dem Kontrollfluss entsprechen

Folgende Pointcuts sind sichtbarkeitsbasiert:

within(Typepattern) alle Join-Points innerhalb des lexikalischen Sichtbereichs einer Klasse oder eines Aspekts

withincode(Method/ConstructorSignature) alle Join-Points im lexikalischen Sichtbereich der Methode oder des Konstruktors

Der Pointcut `set private float Account._balance` beschreibt alle schreibenden Zugriffe auf ein privates Feld vom Typ `float` der Klasse `Account` mit dem Namen `_balance`.

Der folgende Pointcut

```
call(* java.io.PrintStream.print*(..)) &&
        !within(TraceAspect)
```

schließt alle Pointcuts innerhalb von `TraceAspect` von der spezifizierten Menge der Pointcuts (alle Aufrufe von Methoden aus `PrintStream` die mit `print` beginnen) aus.

In einem *Advice* wird angegeben, welche Anweisungen an den ausgewählten Join-Points ausgeführt werden können. Ein Advice hat die Form

`before() :` *Pointcut* {*Programmcode*}

In diesem Fall wird der Programmcode vor dem Pointcut ausgeführt. Wird das Schlüsselwort `after()` verwendet, wird der Programmcode nach dem

Pointcut ausgeführt. Mit dem Schlüsselwort `about()` kann Programmcode um den Pointcut ausgeführt werden. Dabei kann der originale Programmcode z.B. komplett umgangen werden oder mit anderen Argumenten ausgeführt werden.

Im folgenden Programmstück wird ein Advice mit einem anonymen Pointcut und einem mit Namen versehenen Pointcut gezeigt:

```
before() : call(* Account.*(..)) {Benutzer überprüfen}

pointcut connectionOperation(Connection connection) :
   call(* Connection.*(..) throws SQLException);
before(Connection connection) :
   connectionOperation(connection) {
      System.out.println("Operation auf " + connection);
   }
```

Der Advice `{Benutzer überprüfen}` wird vor jedem Aufruf einer Methode mit beliebigem Ergebnistyp und beliebiger Signatur der Klasse Account ausgeführt. Der zweite Advice zeigt wie Information über ein Argument eines Pointcuts und eines Advices weitergereicht werden kann. Der Pointcut `connectionOperation` beschreibt dabei alle Aufrufe von Methoden der Klasse `Connection`, die eine `SQLException` Ausnahme werfen können.

Ein Aspekt fasst Deklarationen, Pointcuts und Advices zusammen. Er sieht genau so wie ein Klasse aus, nur das Schlüsselwort `class` ist durch das Schlüsselwort `aspect` ersetzt:

```
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. Dabei wird in einem `before` *advice* die Einrücktiefe erhöht, in einem `after` *advice* die Einrücktiefe wieder erniedrigt und die Art des Join-Points ausgegeben. `thisJoinPoint` ist dabei ein Zeiger auf das Join-Point-Objekt mit allen Informationen über einen Join-Point. Wird dieser Aspekt auf das vorher beschriebene Testprogramm Point angewendet, wird folgende Ausgabe erzeugt.

```
Before: staticinitialization(Test.<clinit>)
After: staticinitialization(Test.<clinit>)
Before: execution(void Test.main(String[]))
  Before: call(Point(int, int))
    Before: staticinitialization(Point.<clinit>)
  After: staticinitialization(Point.<clinit>)
    Before: preinitialization(Point(int, int))
  After: preinitialization(Point(int, int))
    Before: initialization(Point(int, int))
      Before: execution(Point(int, int))
        Before: set(int Point.x)
      After: set(int Point.x)
        Before: set(int Point.y)
      After: set(int Point.y)
    After: execution(Point(int, int))
  After: initialization(Point(int, int))
After: call(Point(int, int))
  Before: call(void Point.incrXY(int, int))
    Before: execution(void Point.incrXY(int, int))
      Before: get(int Point.x)
    After: get(int Point.x)
      Before: set(int Point.x)
```

```
    After: set(int Point.x)
      Before: set(int Point.y)
    After: set(int Point.y)
  After: execution(void Point.incrXY(int, int))
After: call(void Point.incrXY(int, int))
After: execution(void Test.main(String[]))
```

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).

Da AspectJ die Syntax von Java erweitert, können AspectJ-Programme nicht mit `javac` übersetzt werden, sondern werden mit dem Compiler `ajc` übersetzt. Zu beachten dabei ist, dass Klassen nicht einzeln übersetzt werden können, sondern alle Klassen auf einmal übersetzt werden müssen.

4.5 Wiederholungsfragen

1. Wie werden Ausnahmebehandlungen in Java unterstützt?
2. Wie sind Ausnahmen in Untertypbeziehungen zu berücksichtigen?
3. Wozu kann man Ausnahmen verwenden? Wozu soll man sie verwenden, wozu nicht?
4. Durch welche Sprachkonzepte unterstützt Java die nebenläufige Programmierung? Wozu dienen diese Sprachkonzepte?
5. Wozu brauchen wir Synchronisation? Welche Granularität sollen wir dafür wählen?
6. Zu welchen Problemen kann Synchronisation führen, und was kann man dagegen tun?
7. Was ist aspektorientierte Programmierung? Wann setze ich sie sinnvoll ein.
8. Was bedeutet *separation of concerns*?
9. Was sind *core concerns*, was *cross cutting concerns*?
10. Was sind Join-Points, Pointcuts, Advices und Aspekte?

11. An welchen Programmpunkten können sich Join-Points befinden?

5 Software-Entwurfsmuster

Entwurfsmuster (*Design-Patterns*) dienen der Wiederverwendung kollektiver Erfahrung. Wir wollen exemplarisch einige häufig verwendete Entwurfsmuster betrachten. Da das Thema der Lehrveranstaltung objektorientierte Programmiertechniken sind, konzentrieren wir uns dabei auf Implementierungsaspekte und erwähnen andere in der Praxis wichtige Aspekte nur am Rande. Die Idee der Entwurfsmuster gründet sich im Wesentlichen auf ein weithin bekanntes Buch (Gang-of-Four-Buch), das allen Interessierten empfohlen wird [11]:

> 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 im Skriptum und in der Lehrveranstaltung nur einen kleinen Teil der im Buch beschriebenen und in der Praxis häufig eingesetzten Entwurfsmuster.

5.1 Grundsätzliches

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 muss man zwar den Code neu schreiben, kann dabei aber auf bestehende Erfahrungen zurückgreifen.

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. Software-Entwurfsmuster sind das bekannteste Hilfsmittel in diesem Bereich. Heute kann man davon ausgehen, dass sogut wie jede(r) Software-Entwickler(in) die wichtigsten Entwurfsmuster kennt und in der täglichen Arbeit häufig darauf zurückgreift.

Entwurfsmuster geben im Softwareentwurf immer wieder auftauchenden Problemstellungen und deren Lösungen Namen, damit man einfacher darüber sprechen kann. Außerdem beschreiben Entwurfsmuster, welche Eigenschaften man sich von bestimmten Lösungen erwarten kann. Wer einen ganzen Katalog möglicher Lösungen für eine Aufgabe entweder in schriftlicher Form oder nur abstrakt vor Augen hat, kann gezielt jene Lösung auswählen, deren Eigenschaften der zu entwickelnden Software am ehesten entgegenkommen. Kaum eine Lösung wird nur gute Eigenschaften haben. Häufig wählt man daher jene Lösung, deren Nachteile man am ehesten für akzeptabel hält.

5.1.1 Bestandteile von Entwurfsmustern

Jedes Entwurfsmuster besteht hauptsächlich aus diesen vier Elementen:

Name: Der Name ist wichtig, damit man in einem einzigen Begriff ein Problem, dessen Lösung und Konsequenzen daraus ausdrücken kann. Damit wird der Softwareentwurf auf eine höhere Ebene verlagert; man braucht nicht mehr jedes Detail einzeln anzusprechen. Es ist gar nicht leicht, geeignete Namen für Entwurfsmuster zu finden, die jede(r) Entwickler(in) mit dem Entwurfsmuster assoziiert. Solche Namen müssen sich im Laufe der Zeit gegenüber anderen durchsetzen.

Beispielsweise haben wir in Abschnitt 3.4.2 ein Entwurfsmuster namens *Visitor-Pattern* kennengelernt. 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 auch alle anderen wichtigen Entwurfsmuster) in der Fachliteratur eingehend analysiert, mit ähnlichen Techniken verglichen und schließlich auch häufig angezweifelt. Heute kennt man mehrere ganz unterschiedliche Varianten, auf die wir hier nicht näher eingehen. Gerade diese

umfangreiche Diskussion hat den Begriff erst wirklich etabliert. Genaugenommen verbindet man mit dem Begriff nicht mehr nur eine einzige, eng umrissene Technik, sondern eine ganze Fülle ähnlicher Lösungsansätze in einem Anwendungsbereich, deren Eigenschaften man sehr gut kennt.

Problemstellung: Das ist die Beschreibung des Problems zusammen mit dessen Umfeld. Daraus geht hervor, unter welchen Bedingungen das Entwurfsmuster überhaupt anwendbar ist. Bevor man ein Entwurfsmuster in Betracht zieht, muss man sich überlegen, ob die zu lösende Aufgabe mit dieser Beschreibung übereinstimmt.

Beispielsweise empfiehlt sich das Visitor-Pattern 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 muss man sich schon recht intensiv mit der Problemstellung und den Einsatzmöglichkeiten beschäftigen, bevor man ein Gefühl dafür bekommt, in welchen Situationen die Verwendung eines Entwurfsmusters angebracht ist. Als Anfänger kennt man oft nur eine oder einige wenige Einsatzmöglichkeiten. Durch den praktischen Einsatz, aber auch durch theoretische Überlegungen und einschlägige Literatur lernt man im Laufe der Zeit immer weitere Einsatzmöglichkeiten kennen, bis man das gesamte Einsatzgebiet abschätzen kann.

Lösung: Das ist die Beschreibung einer bestimmten Lösung der Problemstellung. Diese Beschreibung ist allgemein gehalten, damit sie leicht an unterschiedliche Situationen angepasst werden kann. Sie soll jene Einzelheiten enthalten, die zu den beschriebenen Konsequenzen führen, aber nicht mehr.

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. Man kann 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 oder Komponente besser geeignet, die wir vorgefertigt in ein Programm einbinden könnten.

Konsequenzen: Das ist eine Liste von Eigenschaften der Lösung. Man kann sie als eine Liste von Vor- und Nachteilen der Lösung betrachten, muss dabei aber aufpassen, da ein und dieselbe Eigenschaft in manchen Situationen einen Vorteil darstellt, in anderen einen Nachteil und in wieder anderen irrelevant ist.

Eine der wichtigsten Eigenschaften (der betrachteten Version) des Visitor-Patterns besteht darin, unerwünschte dynamische Typabfragen und Typumwandlungen durch dynamisches Binden zu ersetzen. Damit kann man die Wartbarkeit verbessern. Man kann 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 dieses Entwurfsmuster 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 bestimmte Konsequenzen von ganz bestimmten Implementierungsdetails ab.

Entwurfsmuster scheinen die Lösung vieler Probleme zu sein, da man nur mehr aus einem Katalog von Mustern zu wählen braucht, um eine ideale Lösung für ein Problem zu finden. Tatsächlich lassen sich Entwurfsmuster häufig so miteinander kombinieren, dass man alle gewünschten Eigenschaften erhält. Leider führt der exzessive Einsatz von Entwurfsmustern oft zu einem unerwünschten Effekt: Das entstehende 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. Man sollen also genau abwägen, ob es sich im Einzelfall auszahlt, eine bestimmte Eigenschaft auf Kosten der Programmkomplexität zu erzielen. Die Softwareentwicklung bleibt also 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.

5.1.2 Iterator als Beispiel

Iteratoren haben wir schon in Kapitel 3 kennengelernt und auch schon in der Praxis eingesetzt. Genau deswegen eignet sich dieses Entwurfsmuster gut dazu, es als Beispiel für den Aufbau von Entwurfsmustern im Allgemeinen etwas näher zu betrachten.

Name und Problemstellung lassen sich ganz 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.

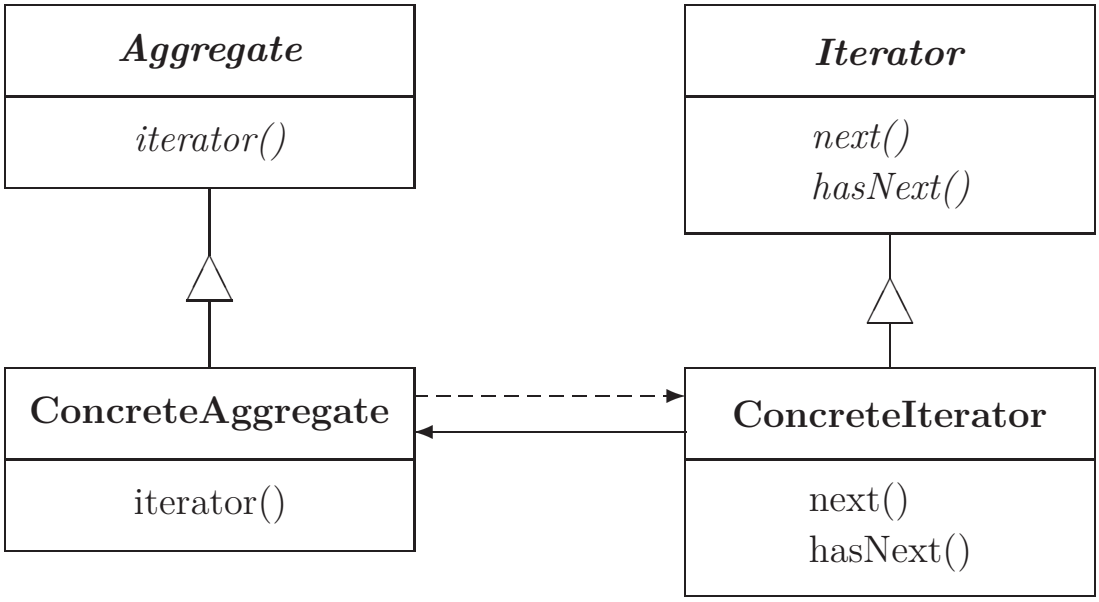
Dieses Entwurfsmuster ist verwendbar, um

- auf den Inhalt 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, das heißt, um polymorphe Iterationen zu unterstützen.

Bei der Beschreibung der meisten Entwurfsmuster folgt hier ein typisches kleines Anwendungsbeispiel. Da wir mit solchen Beispielen schon vertraut sind, verzichten wir ausnahmsweise darauf.

Die Struktur von Entwurfsmustern, hier konkret Iterator, sieht wie in der folgenden Grafik aus. Wir werden Klassen als Kästchen darstellen, die die Namen der Klassen in Fettschrift enthalten. Durch einen waagrechten Strich getrennt können auch Namen von Methoden (mit 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. Namen im Programmcode, der ein Entwurfsmuster implementiert, können sich natürlich von den Namen in der Grafik unterscheiden. Die Namen in der Grafik helfen nur dem intuitiven Verständnis der Struktur und ermöglichen deren Erklärung (bzw. Diskussionen über das Entwurfsmuster). Daher sollte man diese Namen kennen, auch wenn man im Programmcode andere Namen verwendet.



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“ definiert eine Schnittstelle für die Erzeugung eines neuen Iterators. 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 das Aggregat zusätzlich vereinfachen. Beispielsweise enthält das in den Java-Standardbibliotheken vordefinierte Interface `Iterator` auch eine Methode `remove` um das aktuelle Element aus dem Aggregat zu entfernen.

- Auf ein und demselben Aggregat können gleichzeitig mehrere Abarbeitungen stattfinden, da jeder Iterator selbst den aktuellen Abarbeitungszustand verwaltet. Gelegentlich findet man Iterator-ähnlichen Code, in dem der Abarbeitungszustand im Aggregat und nicht im Iterator verwaltet wird. Solcher Code widerspricht dem Iterator-Pattern, da zwingend gefordert ist, dass auf demselben Aggregat mehrere gleichzeitige Abarbeitungen möglich sind.

Es gibt zahlreiche Möglichkeiten zur Implementierung von Iteratoren. Hier sind einige Anmerkungen zu Implementierungsvarianten:

- Man kann zwischen internen und externen Iteratoren unterscheiden. Interne Iteratoren kontrollieren selbst, wann die nächste Iteration erfolgt, bei externen Iteratoren bestimmen die Anwender, wann sie das nächste Element abarbeiten möchten. Alle Beispiele zu Iteratoren, die wir bis jetzt betrachtet haben, sind externe Iteratoren, bei denen Anwender in einer Schleife nach dem jeweils nächsten Element fragen. Ein interner Iterator enthält die Schleife selbst. Der Anwender übergibt dem Iterator eine Routine, die vom Iterator auf allen Elementen ausgeführt 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 schwierig. Andererseits sind interne Iteratoren oft einfacher zu verwenden, da eine Anwendung die Logik für die Iterationen (also die Schleife) nicht braucht.

Interne Iterationen spielen vor allem in der funktionalen Programmierung eine große Rolle, da es dort gute Unterstützung für die dynamische Erzeugung und Übergabe von Routinen (in diesem Fall Funktionen) an Iteratoren gibt, andererseits aber externe Schleifen nur umständlich zu realisieren sind. In der objektorientierten Programmierung werden hauptsächlich externe Iteratoren eingesetzt.

- Oft ist es schwierig, externe Iteratoren auf Sammlungen von Elementen zu verwenden, wenn diese Elemente zueinander in komplexen Beziehungen stehen. Durch die sequentielle Abarbeitung geht die Struktur dieser Beziehungen verloren. Beispielsweise erkennt man 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. Beispielsweise können wir die Methode `max` in `CollectionOps2` (siehe Abschnitt 3.1.3) als internen Iterator betrachten, der eine durch das Argument spezifizierte Methode – in diesem Fall einen Vergleich – auf die Elemente des Aggregats anwendet.

- Der Algorithmus zum Durchwandern eines Aggregats muss nicht immer im Iterator selbst definiert sein. Auch das Aggregat kann den Algorithmus bereitstellen und den Iterator nur dazu benützen, eine Referenz auf das nächste Element zu speichern. 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 kann man Iteratoren durch innere Klassen in Aggregaten definieren, wie zum Beispiel den Iterator in der Klasse List (siehe Abschnitt 3.3.2). 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 einfache Lösung dieses Problems besteht darin, das Aggregat bei der Erzeugung eines Iterators zu kopieren. Meist ist diese Lösung aber viel zu aufwendig. Ein *robuster Iterator* erreicht dasselbe Ziel, ohne das ganze Aggregat zu kopieren. Es erfordert viel Erfahrung und Aufwand, robuste Iteratoren zu schreiben. Die Detailprobleme hängen stark von der Art des Aggregats ab.

- Aus Gründen der Allgemeinheit ist es oft praktisch, Iteratoren auch auf leeren Aggregaten bereitzustellen. In einer Anwendung braucht man die Schleife nur so lange auszuführen, so lange es Elemente gibt – bei leeren Aggregaten daher nie – ohne eine eigene Behandlung für den Spezialfall zu brauchen.

5.2 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 1.2.2. Wir betrachten drei recht einfache erzeugenden Entwurfsmuster: Factory-Method, Prototype und Singleton. Diese Entwurfsmuster wurden gewählt, da sie zeigen, dass man oft mit relativ einfachen Programmiertechniken die in Programmiersprachen vorgegebenen Möglichkeiten erweitern kann.

5.2.1 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 kann man sich 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 eines zu erzeugenden Objekts zur Übersetzungszeit oft nicht bekannt, sodass ein einfaches `new` nicht ausreicht:

```
public abstract class Document { ... }
public class Text extends Document { ... }
...   // classes Picture, Video, ...
public abstract class DocCreator {
    protected abstract Document create();
}
public class TextCreator extends DocCreator {
    protected 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:

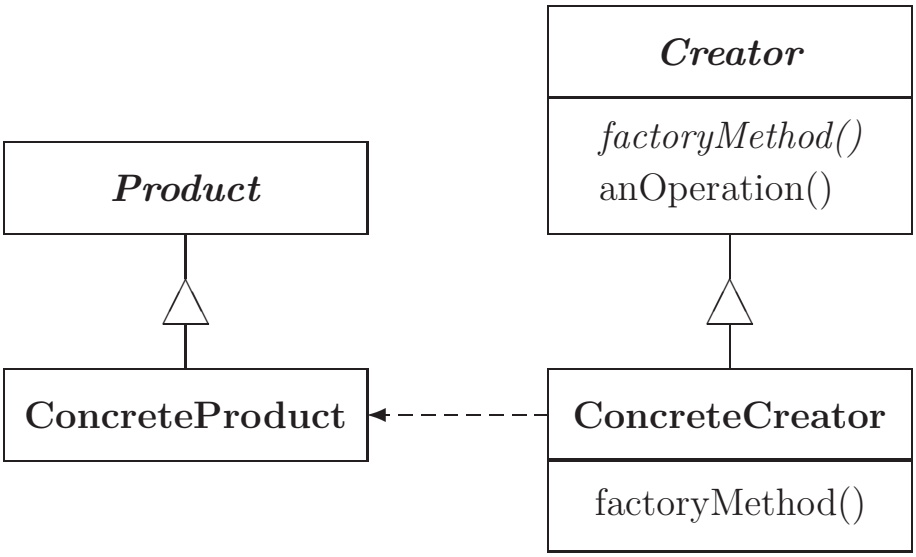
- Argumente von allgemeinem Interesse (nicht spezifisch für bestimmte Dokumente) können an `newDoc` übergeben und über `create` an den (nicht näher bekannten) Konstruktor weitergeleitet werden.
- Man kann Argumente an zentraler Stelle ablegen. Eine Variable wie `c` in `NewDocManager` eignet sich dafür 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, kann man 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 neue Objekte erzeugen soll, deren Klasse aber nicht selbst 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 man das Wissen, an welche Unterklasse delegiert wird, lokal halten möchte;
- 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 Factory-Method 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 Factory-Method. Ausführungen dieser Methode erzeugen neue Objekte von „ConcreteProduct“.

Factory-Methods haben unter anderem folgende Eigenschaften:

- Sie bieten Anknüpfungspunkte (Hooks) für Unterklassen. 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 Klassenhierarchien, die Creator-Hierarchie mit der Product-Hierarchie. Beispielsweise ist die Klassenstruktur

bestehend aus `Document`, `Text`, etc. äquivalent zu der, die von den Klassen `DocCreator`, `TextCreator`, etc. gebildet wird. Dies kann unter anderem bei kovarianten Problemen hilfreich sein. Beispielsweise erzeugt eine Methode `generiereFutter` in der Klasse `Tier` nicht direkt Futter einer bestimmten Art, sondern liefert in der Unterklasse `Rind` ein neues Objekt von `Gras` und in `Tiger` eines von `Fleisch` zurück. Meist sind parallele Klassenhierarchien (mit vielen Klassen) aber unerwünscht.

Zur Implementierung dieses Entwurfsmusters kann man die Factory-Method in „Creator“ entweder als abstrakte Methode realisieren, oder eine Default-Implementierung dafür vorgeben. Im ersten Fall braucht „Creator“ keine Klasse kennen, die als „ConcreteProduct“ verwendbar ist, dafür sind alle konkreten Unterklassen gezwungen, die Factory-Method zu implementieren. Im zweiten Fall kann man „Creator“ selbst zu einer konkreten Klasse machen, gibt aber Unterklassen von „Creator“ die Möglichkeit, die Factory-Method zu überschreiben.

Es ist manchmal sinnvoll, der Factory-Method Parameter mitzugeben, die bestimmen, welche Art von Produkt erzeugt werden soll. In diesem Fall bietet die Möglichkeit des Überschreibens noch mehr Flexibilität.

Hier ist eine Anwendung von Factory-Methods 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 manchmal in der Notwendigkeit, viele Unterklassen von „Creator“ zu erzeugen, die nur `new` mit einem bestimmten „ConcreteProduct“ aufrufen. Zumindest in Java gibt es keine Möglichkeit, diese vielen Klassen zu vermeiden. In C++ könnte man die Zahl der Klassen durch Verwendung von Templates klein halten.

5.2.2 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.

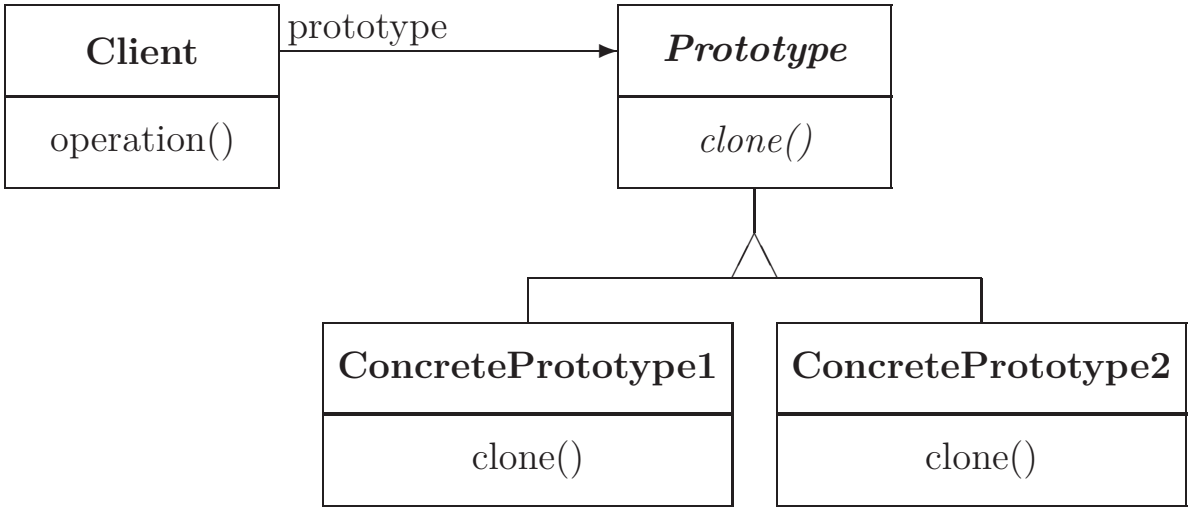
Zum Beispiel kann man in einem System, in dem verschiedene Arten von Polygonen wie Dreiecke und Rechtecke vorkommen, ein neues Polygon durch Kopieren eines bestehenden Polygons erzeugen. Das neue Polygon hat dieselbe Klasse wie das Polygon, von dem die Kopie erstellt wurde. An der Stelle im Programm, an der der Kopiervorgang aufgerufen wird (sagen wir in einem Zeichenprogramm), braucht diese Klasse nicht bekannt zu sein. Das neue Polygon kann etwa 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 zu erzeugen, die einer parallelen Hierarchie von „Product“-Klassen entspricht (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.

Prototypes 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 brauchen nicht geändert zu 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` definiert. 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 Ausnahmebehandlung aus.

Eine Implementierung von `clone` zur Erzeugung tiefer Kopien kann sehr komplex sein. Das Hauptproblem stellen dabei zyklische Referenzen dar. Wenn `clone` einfach nur rekursiv auf zyklische Strukturen angewandt wird, erhält man eine Endlosschleife, die zum Programmabbruch aus Speichermangel führt. Wie solche zyklischen Referenzen aufgelöst werden sollen, hängt im Wesentlichen von der Anwendung ab. 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.

Prototypen sind vor allem in statischen Sprachen wie C++ und Java sinnvoll. In eher dynamischen Sprachen wie Smalltalk und Objective C wird ähnliche Funktionalität bereits direkt von der Sprache unterstützt. Dieses Entwurfsmuster ist in die sehr dynamische objektorientierte Sprache *Self* [34] fest eingebaut und bildet dort die einzige Möglichkeit zur Erzeugung neuer Objekte. Es gibt in Self keine Klassen, sondern nur Objekte, die als Prototypen verwendbar sind.

5.2.3 Singleton

Das Entwurfsmuster *Singleton* sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globalen Zugriff auf dieses Objekt.

Es gibt zahlreiche Anwendungsmöglichkeiten für dieses Entwurfsmuster. 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. Es ist besser, die Klasse selbst für die Verwaltung ihrer einzigen Objekte verantwortlich zu 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 Einfachheit dieses Entwurfsmusters verzichten wir auf eine grafische Darstellung. Ein Singleton besteht nur aus einer gleichnamigen Klasse mit einer statischen Methode „instance“, welche das einzige Objekt der Klasse zurückgibt. Obwohl die Erklärung so einfach ist, sind einige Probleme bei der Implementation schwierig zu lösen, wie später kurz angeschnitten wird.

Singletons haben unter anderem folgende Eigenschaften:

- Sie erlauben den kontrollierten Zugriff auf das einzige Objekt.

- Sie vermeiden durch Verzicht auf globale Variablen unnötige Namen und weitere unangenehme Eigenschaften globaler Variablen.
- Sie unterstützen Vererbung.
- Sie verhindern, dass irgendwo Instanzen außerhalb der Kontrolle der Klasse erzeugt werden.
- Sie erlauben auch mehrere Instanzen. Man kann 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.
- Sie sind flexibler als statische Methoden, da statische Methoden kaum Änderungen erlauben und dynamisches Binden nicht unterstützen.

Einfache Implementierungen dieses Entwurfsmusters bereiten keinerlei Schwierigkeiten, wie folgendes Beispiel zeigt:

```
public class Singleton {
    private static Singleton singleton;
    protected Singleton() {
        // soll nicht von außen aufgerufen werden
        singleton = null;
    }
    public static Singleton instance() {
        if (singleton == null)
            singleton = new Singleton();
        return singleton;
    }
}
```

Man benötigt häufig Singletons, für die mehrere Implementierungen zur Verfügung stehen. Das heißt, die Klasse `Singleton` hat Unterklassen. Beispielsweise gibt es mehrere Implementierungen für Drucker-Spooler, im System sollte trotzdem immer nur ein Drucker-Spooler aktiv sein. Das soll von `Singleton` auch dann garantiert werden, wenn Programmierer(innen) eine Auswahl zwischen den Alternativen treffen können.

Überraschenderweise ist die Implementierung eines solchen Singletons gar nicht mehr einfach. Die folgende Lösung ist noch am einfachsten, wenn auch vielleicht nicht ganz zufriedenstellend:

```
public class Singleton {
    private static Singleton singleton = null;
    protected 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 {
    protected SingletonA() { ... }
}
public class SingletonB extends Singleton {
    protected SingletonB() { ... }
}
```

Nur der erste Aufruf von `instance` wählt die zu verwendende Alternative. Nach Erzeugung des Objekts hat `kind` keinerlei Bedeutung.

Um die feste Verdrahtung der Alternativen in `Singleton` zu vermeiden kann man `instance` in den Untertypen implementieren:

```
public class Singleton {
    protected static Singleton singleton = null;
    protected Singleton() { ... }
    public static Singleton instance() {
        if(singleton==null) singleton = new Singleton();
        return singleton;
    }
}
public class SingletonA extends Singleton {
    protected SingletonA() { ... }
    public static Singleton instance() {
        if(singleton==null) singleton = new SingletonA();
        return singleton;
    }
}
```

Man kann einfach zwischen den Alternativen wählen, indem man den ersten Aufruf von `instance` in der entsprechenden Klasse durchführt. Alle weiteren Aufrufe geben stets das im ersten Aufruf erzeugte Objekt zurück. Allerdings ist nicht mehr die Klasse `Singleton` alleine für die Existenz nur eines Objekts verantwortlich, sondern es müssen alle Unterklassen mitspielen und `instance` entsprechend implementieren.

Es gibt einige weitere Lösungen für dieses Problem, die aber alle ihre eigenen Nachteile haben. Beispielsweise kann man sicherstellen, dass höchstens eine Klasse mit einer alternativen Implementierung geladen wird. Dieser Ansatz ist nicht sehr flexibel. Mehr Flexibilität erhält man, wenn man, ähnlich wie in der ersten Lösung, die Auswahl zwischen Alternativen in der Implementierung von `instance` in `Singleton` durchführt, die gewünschte Alternative statt in einer `switch`-Anweisung aber durch einen Zugriff auf eine kleine Datenbank findet. Allerdings kann der Eintrag neuer Alternativen in die Datenbank problematisch sein, da dies nicht in der Verantwortung von `Singleton` liegt.

5.3 Entwurfsmuster für Struktur und Verhalten

Wir wollen Decorator und Proxy als zwei einfache Vertreter häufig gebrauchter struktureller Entwurfsmuster betrachten, also von Entwurfsmustern, welche die Programmstruktur beeinflussen. Diese Muster können ähnlich aufgebaut sein, unterscheiden sich aber in ihrer Verwendung und ihren Eigenschaften. Als Entwurfsmuster zur Beschreibung des Verhaltens haben wir schon den Iterator und (in groben Zügen) das Visitor-Pattern betrachtet. Als weiteres Verhaltens-Muster wollen wir uns die Template-Method anschauen.

5.3.1 Decorator

Das Entwurfsmuster *Decorator*, auch *Wrapper* genannt, gibt Objekten dynamisch zusätzliche Verantwortlichkeiten (siehe Abschnitt 1.4.3). Decorators stellen eine flexible Alternative zur Vererbung bereit.

Manchmal möchte man einzelnen Objekten zusätzliche Verantwortlichkeiten geben, nicht aber der ganzen Klasse. Zum Beispiel möchte man 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 auch wieder weggenommen wird:

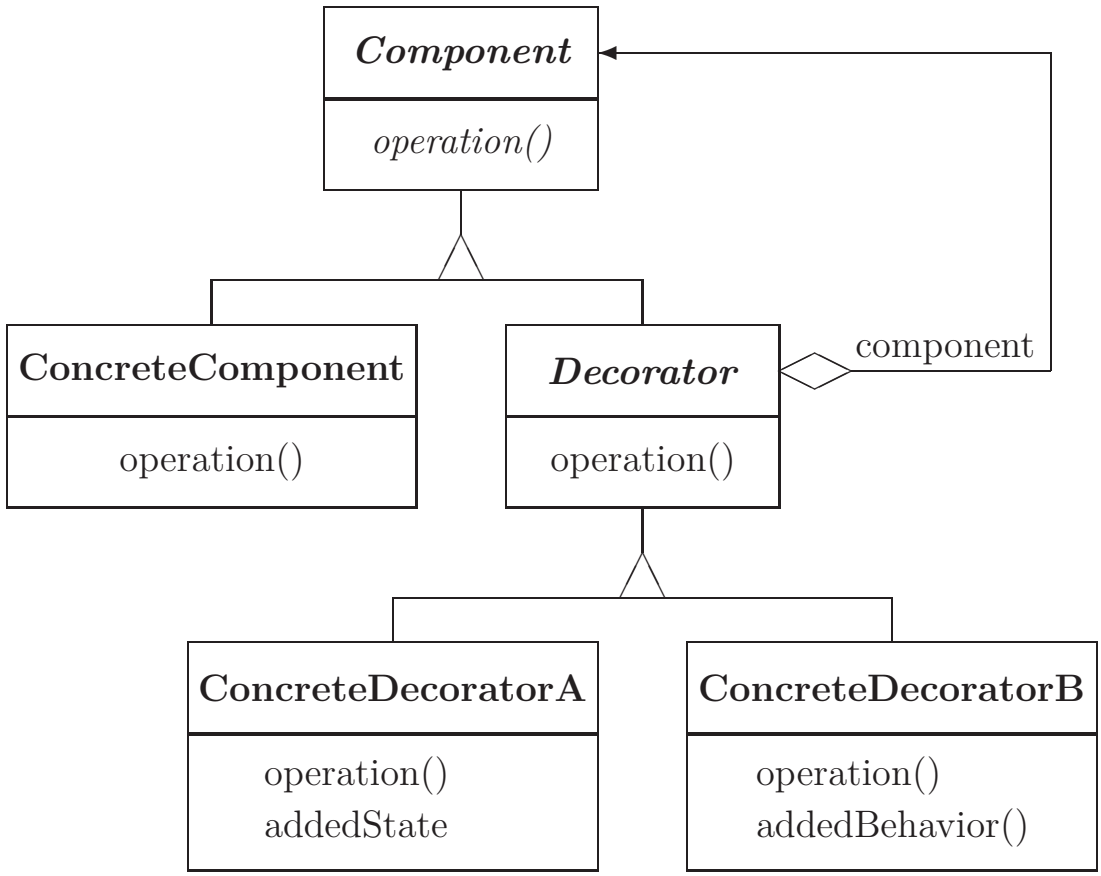
```
public interface Window {
    void show(String text);
}
public class WindowImpl implements Window {
    public void show(String text) { ... }
}
public abstract class WinDecorator implements Window {
    protected Window win;
    public void show(String text) { win.show(text); }
}
public class ScrollBar 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 in einem speziellen Fall keine Vererbung unterstützt (beispielsweise bei final Klassen).

Das Entwurfsmuster hat folgende Struktur, wobei der Pfeil mit einem Kästchen für Aggregation (also eine Referenz auf ein Objekt, dessen Bestandteil das die Referenz enthaltende Objekt ist) steht:



Die abstrakte Klasse beziehungsweise 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 stati-

scher 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 Klassenhierarchie 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 soll man sich 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 man die abstrakte Klasse „Decorator“ weglassen und statt dessen die konkrete Klasse verwenden. 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 braucht diese Methoden nur einmal in der abstrakten Klasse zu 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.

5.3.2 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. Man erzeugt das eigentliche Objekt erst, wenn es wirklich gebraucht wird. Stattdessen verwendet man in der Zwischenzeit einen Platzhalter, der erst bei Bedarf durch das eigentliche Objekt ersetzt wird. Falls nie auf die Daten zugegriffen wird, erspart man sich 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 im Wesentlichen 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. Da die Erzeugung eines Objekts aufwendig sein kann, wird sie so lange verzögert, bis es wirklich einen Bedarf dafür gibt.

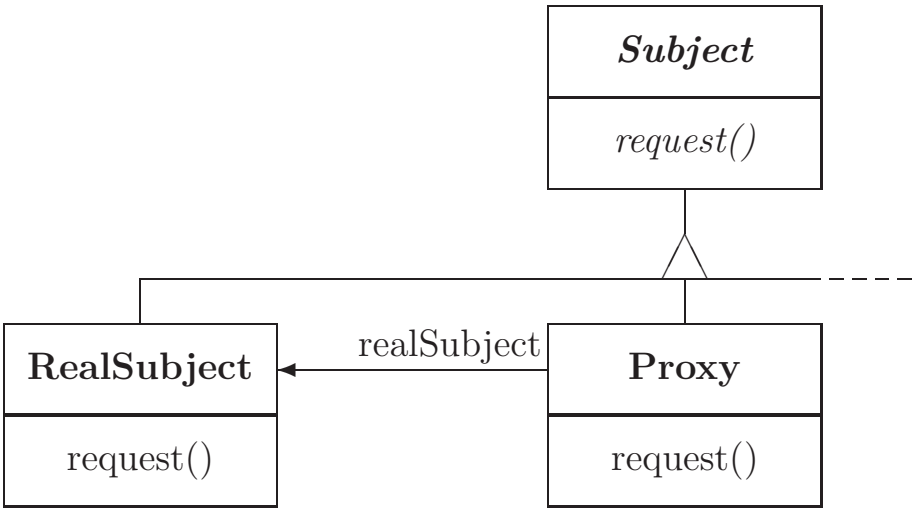
Protection-Proxies kontrollieren Zugriffe auf Objekte. Derartige Proxies sind sinnvoll, wenn Objekte je nach Zugreifer oder Situation unterschiedliche Zugriffsrechte haben sollen.

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“ definiert die 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 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 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 muss man beachten, wie man auf ein Objekt zeigt, das in einem anderen Namensraum liegt oder noch gar nicht existiert. Für nicht existierende Objekte könnte man zum Beispiel `null` verwenden und für Objekte in einer Datei den Dateinamen.

Ein Proxy kann dieselbe 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.

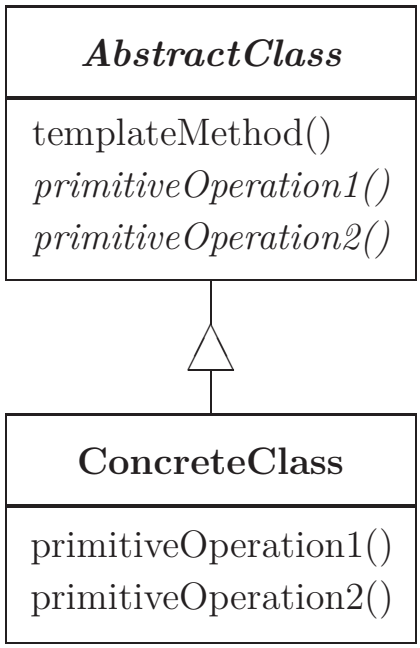
5.3.3 Template-Method

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* aufrufen und nur das Überschreiben dieser Hooks in Unterklassen ermöglichen.

Die Struktur dieses Entwurfsmusters ist recht einfach:



Die (meist abstrakte) Klasse „AbstractClass“ definiert (abstrakte) primitive Operationen und implementiert das Grundgerüst des Algorithmus, das die primitiven Operationen aufruft. Die Klasse „ConcreteClass“ implementiert die primitiven Operationen.

Template-Methods haben unter anderem folgende Eigenschaften:

- Sie stellen eine fundamentale Technik zur direkten Wiederverwendung von Programmcode dar (siehe Beispiele in Abschnitt 2.3.2). Sie sind vor allem in Klassenbibliotheken und Frameworks sinnvoll, weil sie ein Mittel sind, um gemeinsames Verhalten zu faktorisieren.
- Sie führen zu einer umgekehrten Kontrollstruktur, die manchmal als *Hollywood-Prinzip* bezeichnet wird („Don't call us, we'll call you"). Die Oberklasse ruft die Methoden der Unterklasse auf – nicht wie in den meisten Fällen umgekehrt.
- Sie rufen meist nur eine von mehreren Arten von Operationen auf:
  - konkrete Operationen in „ConcreteClass";
  - konkrete Operationen in „AbstractClass", also Operationen, die ganz allgemein auch für Unterklassen sinnvoll sind;
  - abstrakte primitive Operationen, die einzelne Schritte im Algorithmus ausführen;
  - Factory-Methods;
  - Hooks, das sind Operationen mit in „AbstractClass" definiertem Default-Verhalten, das bei Bedarf in Unterklassen überschrieben oder erweitert werden kann; oft besteht das Default-Verhalten darin, nichts zu tun.

  Es ist wichtig, dass genau spezifiziert ist, welche Operationen Hooks (dürfen überschrieben werden) und welche abstrakt sind (müssen überschrieben werden). Für die effektive Wiederverwendung ist es wichtig, dass man weiß, welche Operationen dafür vorgesehen sind, in Unterklassen überschrieben zu werden. Alle Operationen, bei denen dies Sinn macht, sollen Hooks sein, da es beim Überschreiben anderer Operationen leicht zu Fehlern kommt.

Die primitiven Operationen, die von der Template-Methode aufgerufen werden, sind in der Regel `protected` Methoden, damit sie nicht in unerwünschten Zusammenhängen aufrufbar sind. Primitive Operationen, die überschrieben werden müssen, sind als `abstract` deklariert. Die Template-Methode selbst, also die Methode, die den Algorithmus implementiert, soll nicht überschrieben werden. Sie kann `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".

5.4 Wiederholungsfragen

1. Erklären Sie folgende Entwurfsmuster und beschreiben Sie jeweils das Anwendungsgebiet, die Struktur, die Eigenschaften und wichtige Details der Implementierung:
   - Decorator
   - Factory-Method
   - Iterator
   - Prototype
   - Proxy
   - Singleton
   - Template-Method
   - Visitor (siehe Abschnitt 3.4.2)
2. Welche Arten von Iteratoren gibt es, und wofür sind sie geeignet?
3. Wie wirkt sich die Verwendung eines Iterators auf die Schnittstelle des entsprechenden Aggregats aus?
4. Inwiefern können geschachtelte Klassen bei der Implementierung von Iteratoren hilfreich sein?
5. Was ist ein robuster Iterator? Wozu braucht man Robustheit?
6. Wird die Anzahl der benötigten Klassen im System bei Verwendung von Factory-Method, Prototype, Decorator und Proxy (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?
7. Wird die Anzahl der benötigten Objekte im System bei Verwendung von Factory-Method, Prototype, Decorator und Proxy (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?
8. Vergleichen Sie Factory-Method mit Prototype. Wann stellt welches Entwurfsmuster die bessere Lösung dar? Warum?
9. Wo liegen die Probleme in der Implementierung eines so einfachen Entwurfsmusters wie Singleton?
10. Welche Unterschiede und Ähnlichkeiten gibt es zwischen Decorator und Proxy?

11. Welche Probleme kann es beim Erzeugen von Kopien im Prototype geben? Was unterscheidet flache Kopien von tiefen?

12. Für welche Arten von Problemen ist Decorator gut geeignet, für welche weniger? (Oberfläche versus Inhalt)

13. Kann man mehrere Decorators bzw. Proxies hintereinander verketten? Wozu kann so etwas gut sein?

14. Was unterscheidet Hooks von abstrakten Methoden?

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] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[12] Brian Goetz. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[13] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[14] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceeding IJCAI'73*, 1973.

[15] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[16] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[17] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, 1969.

[18] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Information and Computation*, 175(1):34–49, 2002.

[19] ISO/IEC 8652:1995. Annotated ada reference manual. Intermetrics, Inc., 1995.

[20] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA, 1989.

[21] 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.

[22] Wilf LaLonde and John Pugh. Subclassing $\neq$ subtyping $\neq$ 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.