Skriptum zu

Objektorientierte Programmierung

Wintersemester 2002/2003

Franz Puntigam
Technische Universität Wien
Institut für Computersprachen
http://www.complang.tuwien.ac.at/franz/objektorientiert.html

Inhaltsverzeichnis

1 Grundlagen und Ziele **1**
- 1.1 Konzepte objektorientierter Programmierung 2
  - 1.1.1 Objekte 2
  - 1.1.2 Klassen 5
  - 1.1.3 Polymorphismus 8
  - 1.1.4 Vererbung 11
- 1.2 Qualität in der Programmierung 13
  - 1.2.1 Qualität von Programmen 14
  - 1.2.2 Effizienz der Programmerstellung und Wartung . . 17
  - 1.2.3 Qualität von Programmiersprachen 19
- 1.3 Rezept für gute Programme 20
  - 1.3.1 Zusammenhalt und Kopplung 21
  - 1.3.2 Wiederverwendung 24
  - 1.3.3 Entwurfsmuster 26
- 1.4 Paradigmen der Programmierung 28
  - 1.4.1 Imperative Programmierung 28
  - 1.4.2 Deklarative Programmierung 30
  - 1.4.3 Paradigmen für Modularisierungseinheiten 31

2 Enthaltender Polymorphismus und Vererbung **33**
- 2.1 Das Ersetzbarkeitsprinzip 33
  - 2.1.1 Untertypen und Schnittstellen 34
  - 2.1.2 Untertypen und Codewiederverwendung 38
  - 2.1.3 Dynamisches Binden 41
- 2.2 Ersetzbarkeit und Objektverhalten 44
  - 2.2.1 Client-Server-Beziehungen 45
  - 2.2.2 Untertypen und Verhalten 50
  - 2.2.3 Abstrakte Klassen 54

2.3 Vererbung versus Ersetzbarkeit 56
2.3.1 Reale Welt versus Vererbung versus Ersetzbarkeit . 56
2.3.2 Vererbung und Codewiederverwendung 58
2.4 Klassen und Vererbung in Java 61
2.4.1 Klassen in Java 61
2.4.2 Vererbung in Java 65
2.4.3 Zugriffskontrolle in Java 67
2.4.4 Schnittstellen in Java 71

3 Generizität und Ad-hoc-Polymorphismus 75
3.1 Generizität 75
3.1.1 Wozu Generizität? 76
3.1.2 Einfache Generizität in GJ 77
3.1.3 F-gebundene Generizität in GJ 81
3.2 Verwendung von Generizität im Allgemeinen 85
3.2.1 Richtlinien für die Verwendung von Generizität . . 86
3.2.2 Arten der Generizität 90
3.3 Typabfragen und Typumwandlung 94
3.3.1 Verwendung dynamischer Typinformation 94
3.3.2 Typumwandlungen und Generizität 98
3.3.3 Kovariante Probleme 102
3.4 Überladen versus Multimethoden 106
3.4.1 Unterschiede zwischen Überladen und Multimethoden 107
3.4.2 Simulation von Multimethoden 110
3.5 Ausnahmebehandlung 113
3.5.1 Ausnahmebehandlung in Java 113
3.5.2 Einsatz von Ausnahmebehandlungen 117

4 Softwareentwurfsmuster 121
4.1 Erzeugende Entwurfsmuster 122
4.1.1 Factory Method 122
4.1.2 Prototype 125
4.1.3 Singleton 128
4.2 Strukturelle Entwurfsmuster 131
4.2.1 Decorator 131
4.2.2 Proxy 134
4.3 Entwurfsmuster für Verhalten 136
4.3.1 Iterator 136
4.3.2 Template Method 139

Vorwort

Die Lehrveranstaltung „Objektorientierte Programmierung“ ist eine Vorlesung mit Laborübung im Umfang von zwei Semesterwochenstunden an der Technischen Universität Wien. Unter anderem werden folgende Themenbereiche der objektorientierten Programmierung anhand von Java (erweitert um Generizität) behandelt:

- Datenabstraktion
- Polymorphismus
- Vererbung und Untertyprelationen
- Klassenhierarchien
- Generizität
- Objektschnittstellen
- Implementierung gängiger Entwurfsmuster

Die TeilnehmerInnen an der Lehrveranstaltung sollen einen Überblick über die wichtigsten Konzepte objektorientierter Programmiersprachen bekommen und lernen, diese Konzepte so einzusetzen, dass qualitativ möglichst hochwertige und gut wartbare Software entsteht. Grundlegende Kenntnisse zumindest einer Programmiersprache werden vorausgesetzt. Java-Kenntnisse werden nicht vorausgesetzt, sind aber hilfreich.

Das erste Kapitel dieses Skriptums

- führt grundlegende objektorientierte Programmierkonzepte ein,
- gibt einen Überblick über die Qualität in der Programmierung,
- vermittelt erste Hinweise darauf, mit welchen Problemen man in der objektorientierten Programmierung rechnen muss und wie man diese umgehen oder lösen kann,

- und klassifiziert Programmiersprachen anhand ihrer Paradigmen, um eine Einordnung der objektorientierten Sprachen in die Vielfalt an Programmiersprachen zu erleichtern.

Das zweite Kapitel beschäftigt sich mit dem in der objektorientierten Programmierung besonders wichtigen Themenkomplex des enthaltenden Polymorphismus zusammen mit Klassenhierarchien, Untertyprelationen und Vererbung. Vor allem das Ersetzbarkeitsprinzip wird ausführlich behandelt. Eine Beschreibung der Umsetzung dieser Konzepte in Java rundet das zweite Kapitel ab.

Das dritte Kapitel ist neben weiteren Formen des Polymorphismus vor allem der Generizität gewidmet. Es werden Programmiertechniken vorgestellt, die entsprechende Problemstellungen auch bei fehlender Sprachunterstützung für Generizität, kovariante Spezialisierungen und mehrfaches dynamisches Binden lösen können. Auch Ausnahmebehandlungen werden im dritten Kapitel kurz angesprochen.

Das letzte Kapitel stellt häufig verwendete Entwurfsmuster vor. Nebenbei werden einige Tipps und Tricks in der objektorientierten Programmierung gegeben.

Die Lehrveranstaltung soll einen Überblick über Konzepte der objektorientierten Programmierung, Zusammenhänge zwischen ihnen, mögliche Schwierigkeiten sowie Ansätze zu deren Beseitigung vermitteln. Keinesfalls soll sie als Java-Kurs verstanden werden. Insbesondere die umfangreichen Klassenbibliotheken, die in der Java-Programmierung Verwendung finden, werden nicht behandelt. Informationen finden an der Java-Programmierung interessierte LeserInnen unter anderem im world wide web, zum Beispiel unter `http://java.sun.com`.

Viel Erfolg bei der Teilnahme an der Lehrveranstaltung!

Franz Puntigam

`franz@complang.tuwien.ac.at`
`http://www.complang.tuwien.ac.at/franz/objektorientiert.html`

Kapitel 1

Grundlagen und Ziele

Immer mehr Unternehmen der Softwarebranche steigen auf objektorientierte Programmierung um. Ein großer Teil der SoftwareentwicklerInnen verwendet derzeit bereits Methoden der objektorientierten Programmierung. Dabei stellt sich die Frage, welche Vorteile die objektorientierte Programmierung gegenüber anderen Paradigmen bietet oder zumindest erwarten lässt, die den umfangreichen Einsatz in der Praxis rechtfertigen. Solche erhofften Vorteile sowie mögliche Gefahren wollen wir in diesem Kapitel betrachten. Die Stellung der objektorientierten Programmierung unter der Vielzahl existierender Programmierparadigmen wollen wir durch eine Klassifizierung der Paradigmen veranschaulichen. Außerdem soll das Kapitel einen ersten Überblick über die später im Detail behandelten Themen geben und nebenbei einige wichtige Begriffe einführen.

In Abschnitt 1.1 werden die wichtigsten Konzepte objektorientierter Programmiersprachen angesprochen. Diese Konzepte werden in den folgenden Kapiteln genauer behandelt.

In Abschnitt 1.2 beschäftigen wir uns damit, welche Ziele durch die Programmierung im Allgemeinen erreicht werden sollen, was gute Programmierung von schlechter Programmierung unterscheidet und wie Programmiersprachen die Qualität beeinflussen können.

In Abschnitt 1.3 werden wir untersuchen, wie man gute objektorientierte Programme schreibt und welche Schwierigkeiten dabei zu überwinden sind.

Abschnitt 1.4 gibt eine Klassifizierung von Programmiersprachen anhand ihrer üblichen Verwendungen. Diese Klassifizierung soll Zusammenhänge mit anderen Paradigmen aufzeigen und helfen, den Begriff der objektorientierten Programmierung abzugrenzen.

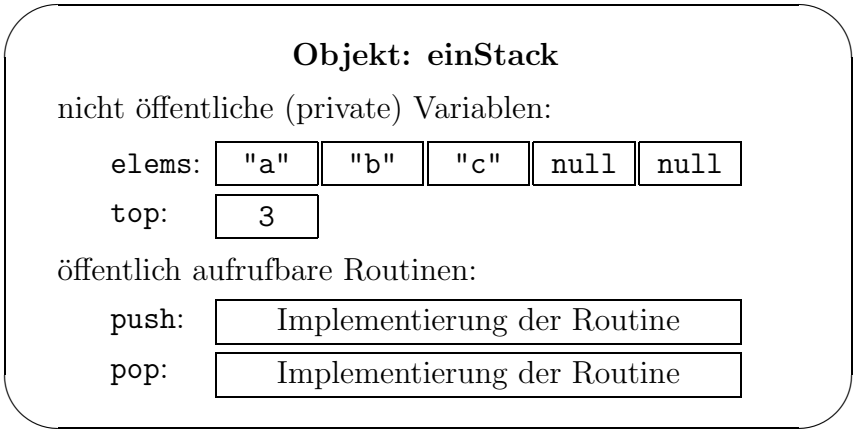
1.1 Konzepte objektorientierter Programmierung

Wir wollen zunächst die objektorientierte Programmierung und ihre Konzepte betrachten. Das wichtigste Ziel der objektorientierten Programmierung ist, den EntwicklerInnen alle Werkzeuge zu geben, die sie brauchen, um qualitativ hochwertige Programme zu schreiben. Dabei sollen vor allem Softwareentwicklungsprozesse, die auf inkrementelle Verfeinerung aufbauen, unterstützt werden. Gerade bei diesen Entwicklungsprozessen spielt die leichte Wartbarkeit der Programme eine große Rolle. Im Wesentlichen will die objektorientierte Programmierung also die leichte Änderbarkeit von Programmen unterstützen.

1.1.1 Objekte

Das wichtigste Konzept der objektorientierten Programmierung ist, wie der Name schon sagt, das des *Objekts*. Ein Objekt ist eine grundlegende Einheit in der Ausführung eines Programms. Zur Laufzeit besteht die Software aus einer Menge von Objekten, die einander (teilweise) kennen und untereinander *Nachrichten* (messages) austauschen. Man kann ein Objekt am ehesten als eine *Kapsel* verstehen, die zusammen gehörende Variablen und Routinen enthält. Zusammen genommen beschreiben die Variablen und Routinen eine Einheit in der Software. Von außen soll man auf das Objekt nur zugreifen, indem man ihm eine Nachricht schickt, das heißt, eine nach außen sichtbare Routine des Objekts aufruft.

Die folgende Abbildung veranschaulicht ein Objekt:



Dieses Objekt mit der Funktionalität eines Stacks kapselt zwei Variablen und zwei Routinen zu einer Einheit. Die Routinen sind von außerhalb

des Objekts aufrufbar. Auf die Variablen kann nur durch diese beiden Routinen zugegriffen werden. Eine Variable enthält ein Array mit dem Inhalt des Stacks, eine andere die aktuelle Anzahl der Elemente im Stack. Das Array kann höchstens fünf Stackelemente halten. Zur Zeit sind drei Einträge vorhanden.

Das Zusammenfügen von Daten und Routinen zu einer Einheit nennt man *Kapselung* (encapsulation). Daten und Routinen in einem Objekt sind untrennbar miteinander verbunden: Die Routinen benötigen die Daten zur Erfüllung ihrer Aufgaben, und die genaue Bedeutung der Daten ist oft nur den Routinen des Objekts bekannt. Routinen und Daten stehen zueinander in einem engen logischen Zusammenhang. In Abschnitt 1.2 werden wir sehen, dass eine gut durchdachte Kapselung von Daten und Routinen ein wichtiges Qualitätsmerkmal objektorientierter Programme ist. In Abschnitt 1.3 werden wir Faustregeln zur Unterstützung der Suche nach geeigneten Kapselungen kennen lernen. In Abschnitt 1.4 werden wir feststellen, dass die Kapselung von Daten und Routinen zu Objekten ein entscheidendes Kriterium zur Abgrenzung der objektorientierten Programmierung von anderen Programmierparadigmen ist.

Jedes Objekt besitzt folgende Eigenschaften:

Identität (identity): Seine Identität kennzeichnet ein Objekt eindeutig. Sie ist unveränderlich. 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 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 in der Regel änderbar. In obigem Beispiel ändert sich der Zustand durch Zuweisungen neuer Werte an die Variablen `elems` und `top`. 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 kann durch Zustandsänderungen nicht verloren gehen.

Verhalten (behavior): Das Verhalten eines Objekts beschreibt, wie sich das Objekt beim Empfang einer Nachricht verhält, das heißt, was das Objekt beim Aufruf einer entsprechenden Routine macht. Routinen, die beim Empfang von Nachrichten ausgeführt werden, nennt man häufig *Methoden* (methods). Das Verhalten ist von der Nachricht – also dem Methodennamen zusammen mit den *aktuellen Parametern*, auch *Argumente* der Nachricht genannt –, der entsprechenden aufgerufenen Methode und dem Zustand des Objekts abhängig. In obigem Beispiel wird die Methode `push` beim Empfang der Nachricht `push("d")` das Argument `"d"` in den Stack einfügen, und `pop` beim Empfang von `pop()` ein Element entfernen und an den Absender der Nachricht zurück geben.

Diese drei Eigenschaften sind zur Definition des Begriffs Objekt unbedingt notwendig. Wir verlangen aber noch eine weitere Eigenschaft:

Schnittstelle (interface): Schnittstellen eines Objekts beschreiben das Verhalten des Objekts, sofern dies für Zugriffe von außen notwendig ist. Ein Objekt kann mehrere Schnittstellen haben, die das Objekt aus den Sichtweisen unterschiedlicher Verwendungen beschreiben. Meist enthalten die Schnittstellen nur die Köpfe der überall aufrufbaren Routinen. Manchmal enthalten sie auch Konstanten. Die Schnittstellen stehen in engem Zusammenhang mit den Typen des Objekts.

Häufig verwendet man ein Objekt als *black box* oder *grey box*; das heißt, der Inhalt des Objekts ist von außen zum Großteil nicht sichtbar. Nur das, was in den Schnittstellen beschrieben ist, ist von außen sichtbar. Die Schnittstellen dienen dazu, den Inhalt des Objekts von dessen verschiedenen Außenansichten klar zu trennen. ProgrammiererInnen, die ein Objekt verwenden wollen, brauchen nur eine Schnittstelle des Objekts kennen, nicht aber dessen Inhalt. Man spricht daher von *data hiding*. Kapselung zusammen mit data hiding heißt *Datenabstraktion*, da die Daten in einem Objekt nicht mehr direkt sichtbar und manipulierbar, sondern abstrakt sind. Im Beispiel sieht man die Daten des Objekts nicht als Array von Elementen zusammen mit der Anzahl der gültigen Einträge im Array, sondern als abstrakten Stack, der über zwei Methoden zugreifbar und manipulierbar ist. Diese Abstraktion bleibt unverändert, wenn wir das Array gegen eine andere Datenstruktur, sagen wir eine Liste austauschen. Durch diese Eigenschaft sind Datenabstraktionen sehr hilfreich bei

der Wartung: Details von Objekten sind änderbar, ohne deren Außenansichten und damit deren Verwendungen zu beeinflussen.

1.1.2 Klassen

Objektorientierte Sprachen beinhalten ein Klassenkonzept. Jedes Objekt gehört zu genau einer Klasse, die die Struktur des Objekts beschreibt. Außerdem beschreibt die Klasse *Konstruktoren* (constructors), das sind Routinen zur Erzeugung bzw. Initialisierung neuer Objekte. Alle Objekte, die zur Klasse gehören, wurden durch Konstruktoren dieser Klasse erzeugt. Man nennt diese Objekte *Instanzen* der Klasse. (Anmerkung: Man sagt manchmal, ein Objekt gehöre zu mehreren Klassen, der spezifischsten Klasse und deren Oberklassen; wir verstehen im Skriptum unter der Klasse eines Objekts immer dessen spezifischste Klasse.)

Alle Instanzen einer Klasse haben nicht nur dieselben Methoden – also dasselbe Verhalten –, sondern auch dieselben Schnittstellen und dieselben Typen. Aber zwei unterschiedliche Instanzen der gleichen Klasse haben immer unterschiedliche Identitäten und unterschiedliche Variablen – genauer: Instanzvariablen –, obwohl diese Variablen gleiche Namen und Typen tragen. Auch die Objektzustände können sich unterscheiden.

In einer objektorientierten Programmiersprache mit Klassen schreiben ProgrammiererInnen hauptsächlich Klassen. Objekte werden nur zur Laufzeit durch Verwendung von Konstruktoren erzeugt. Oft gibt es in diesen Sprachen gar keine Möglichkeit, Objekte direkt auszuprogrammieren.

Neben der Struktur und Initialisierung ihrer Instanzen beschreiben Klassen Schnittstellen. Diese legen fest, was für wen zugreifbar ist.

Ein kleines Beispiel in Java soll demonstrieren, wie Klassen aussehen:

```
class Stack {
    private String[] elems;
    private int top = 0;
    public Stack (int sz) {
        elems = new String[sz];
    }
    public void push (String elem) {
        if (top < elems.length) {
            elems[top] = elem;
            top = top + 1;
        }
    }
```

```
    public String pop() {
        if (top > 0) {
            top = top - 1;
            return elems[top];
        }
        else
            return null;
    }
}
```

Folgende Beispielserklärung ist für Leser gedacht, die noch nicht genug Erfahrung mit Java gesammelt haben. Erfahrene ProgrammiererInnen in Java können solche speziell gekennzeichneten Textstellen überspringen.

(Anmerkungen zu Java)

Jede Instanz der Klasse `Stack` enthält die Variablen `elems` vom Typ `String[]` (Array von Zeichenketten) sowie `top` vom Typ `int` – ganze Zahl von -2^{31} bis $2^{31} - 1$. Alle Variablen sind mit `private` deklariert, also nur in Instanzen von `Stack` sichtbar. Jede Instanz unterstützt `push` und `pop`. Beide Methoden sind `public`, also überall sichtbar, wo eine Instanz von `Stack` bekannt ist. Der Ergebnistyp `void` bedeutet, dass `push` kein Ergebnis zurück gibt. Der formale Parameter `elem` von `push` ist vom Typ `String`. Die Methode `pop` liefert ein Ergebnis vom Typ `String`, hat aber keine formalen Parameter – ausgedrückt durch ein leeres Klammernpaar. Daneben gibt es einen Konstruktor. Syntaktisch sieht ein Konstruktor wie eine Methode aus, abgesehen davon, dass der Name immer gleich dem Namen der Klasse ist und kein Ergebnistyp angegeben wird. Der Konstruktor im Beispiel ist `public`, also überall sichtbar.

Neue Objekte werden durch den Operator `new` erzeugt und durch den Aufruf eines Konstruktors initialisiert. Zum Beispiel erzeugt `new Stack(5)` eine neue Instanz von `Stack` mit neuen Variablen `elems` und `top` und ruft den Konstruktor in `Stack` auf, wobei der formale Parameter `sz` an `5` gebunden ist. Bei der Ausführung des Konstruktors wird durch `new String[sz]` eine neue Instanz eines Arrays von Zeichenketten erzeugt. Im Array finden 5 Zeichenketten Platz. Dieses Array wird an die Variable `elems` zugewiesen. Die Variable `top` wurde bereits zu Beginn mit 0 initialisiert.

In Java sind auch Arrays gewöhnliche Objekte, allerdings mit einer speziellen Syntax durch Verwendung eckiger Klammern. Am Anfang enthält jeder Array-Eintrag `null`; das bedeutet, dass der Array-Eintrag mit keinem Objekt belegt ist. Jede Variable in Java, die ein Objekt enthalten kann, kann stattdessen auch `null` enthalten.

Ein Aufruf von `push` stellt fest, ob es im Array noch einen freien Eintrag gibt, also `top` kleiner als `elems.length` (die Größe des Arrays `elems`) ist. In diesem Fall wird der Parameter als neues Element in das Array eingetragen und `top` erhöht; andernfalls bleibt der Zustand unverändert. Ein Aufruf von `pop` dekrementiert `top` um 1 und liefert durch eine `return`-Anweisung den Array-Eintrag an der Position `top` zurück, falls `top` größer als 0 ist; sonst liefert die Methode den Wert `null` – kein Objekt – zurück.

Da jede Instanz von `Stack` ihre eigenen Variablen hat, stellt sich die Frage, zu welcher Instanz von `Stack` die Variablen gehören, auf die die Methoden zugreifen. In der Klasse selbst steht nirgends, welches Objekt das ist. Die Instanz von `Stack`, die dabei verwendet wird, ist im Aufruf der Methode eindeutig festgelegt, wie wir in folgendem Beispiel sehen können:

```
class StackTest {
    public static void main (String[] args) {
        Stack s = new Stack(5);
        int i = 0;
        while (i < args.length) {
            s.push(args[i]);
            i = i + 1;
        }
        while (i > 0) {
            i = i - 1;
            System.out.println(s.pop());
        }
    }
}
```

(Anmerkungen zu Java)

Die Klasse `StackTest` definiert nur eine Methode `main`. Diese Methode wird automatisch aufgerufen, wenn `StackTest` als Java-Programm verwendet wird. Die Methode ist `public` und `static`; das heißt, sie ist überall sichtbar und hängt nicht von irgendwelchen Variablen einer Instanz von `StackTest` ab. Daher können wir `main` überall aufrufen, ohne eine Instanz von `StackTest` zu benötigen. Das ist erforderlich, da es beim Start des Programms ja noch keine Instanzen der Klasse gibt. Die Methode hat ein Array von Zeichenketten als Parameter. Beim Programmstart enthält dieses Array die Argumente (command line arguments), die im Programmaufruf angegeben werden. Nachdem `StackTest` und `Stack` durch

```
javac Stack.java StackTest.java
```

übersetzt wurden, können wir das Programm zum Beispiel so aufrufen:

```
java StackTest a b c
```

Damit ist `args` ein Array von drei Zeichenketten – `"a"`, `"b"` und `"c"`.

Die Methode `main` hat zwei lokale Variablen. Die Variable `s` wird mit einer neuen Instanz von `Stack` initialisiert und `i` mit 0. Die erste Schleife wird für jede Zeichenkette in `args` einmal durchlaufen. Der Ausdruck `args.length` bezeichnet die Variable `length` im Objekt `args`, die in unserem Fall die Anzahl der Elemente im Array `args` angibt. In jedem Schleifendurchlauf wird die Nachricht `push(args[i])` an das Objekt

`s` gesendet; es wird also `push` in `s` mit der Zeichenkette `args[i]` als Argument aufgerufen. Bei der Ausführung von `push` ist bekannt, dass die Variablen des Objekts `s` zu verwenden sind. Die zweite Schleife wird gleich oft durchlaufen wie die erste. Die Anweisung `System.out.println(s.pop())` gibt das oberste Element am Stack auf die Standardausgabe – normalerweise das Terminal – aus und entfernt dieses Element vom Stack. Im Detail passiert Folgendes: `System` ist eine im Java-System vorgegebene Klasse, die eine statische Variable `out` enthält. Eine mit `static` deklarierte Variable unterscheidet sich von einer Instanzvariable – ohne `static` deklariert – dadurch, dass sie nicht zu einer Instanz der Klasse gehört, sondern zur Klasse selbst. Daher brauchen wir auch keine Instanz von `System` angeben, um auf die Variable zuzugreifen, sondern die Klasse selbst. Die Variable `System.out` enthält ein Objekt, den output stream für die Standardausgabe. In diesem Objekt wird die Methode `println` aufgerufen, die eine Zeile mit dem Argument in den output stream schreibt. Als Argument wird der Methode das Ergebnis eines Aufrufs von `pop` in `s` übergeben. Nach einem Programmaufruf `java StackTest a b c` werden am Bildschirm folgende drei Zeilen ausgegeben:

```
c
b
a
```

Was ausgegeben wird, wenn der Programmaufruf mehr als 5 Argumente enthält, kann sich der Leser selbst überlegen – oder ausprobieren.

1.1.3 Polymorphismus

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

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

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

Statischer Typ: Der statische Typ wird vom Compiler (statisch) ermittelt. Dieser Typ kann spezieller sein als der deklarierte Typ. In vielen Fällen ordnet der Compiler ein und derselben Variablen an verschiedenen Stellen verschiedene statische Typen zu. Solche Typen werden

beispielsweise für die statische Typüberprüfung verwendet. Es hängt von der Qualität des Compilers ab, wie speziell der statische Typ ist. In Sprachdefinitionen kommen statische Typen daher nicht vor.

Dynamischer Typ: Das ist der spezifischste Typ, den der in der Variable gespeicherte Wert tatsächlich hat. Dynamische Typen sind oft spezieller als statische Typen und können sich mit jeder Zuweisung eines Wertes an eine Variable ändern. Dem Compiler sind dynamische Typen nur in dem Spezialfall bekannt, in dem dynamische und statische Typen einander stets entsprechen. Dynamische Typen werden unter anderem für die Typüberprüfung zur Laufzeit verwendet.

Man kann verschiedene Arten von Polymorphismus unterscheiden:

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

Beim universellen Polymorphismus haben die Typen, die zueinander in Beziehung stehen, eine gleichförmige Struktur. Diese Gleichförmigkeit ist beim Ad-hoc-Polymorphismus nicht nötig.

Generizität (genericity): Generizität wird auch als *parametrischer Polymorphismus* bezeichnet, weil die Gleichförmigkeit durch Typparameter erreicht wird. Das heisst, Ausdrücke können Parameter enthalten, für die Typen eingesetzt werden. Zum Beispiel kann im Ausdruck `List<a>` der Typparameter `a` durch den Typ `int` ersetzt werden. Das Ergebnis der Ersetzung, `List<int>`, ist der (generierte) Typ einer Liste von ganzen Zahlen. Ein Ausdruck mit freien Typparametern bezeichnet die Menge aller Ausdrücke, die durch Einsetzen von Typen generiert werden können. Anders gesagt, Typparameter werden implizit als universell über die Menge aller Typen quantifizierte Variablen betrachtet. Daher wird Generizität dem *universellen Polymorphismus* zugerechnet. Wir werden uns in Kapitel 3 mit Generizität in der objektorientierten Programmierung beschäftigen.

Enthaltender Polymorphismus (inclusion polymorphism): Diese Art, auch *subtyping* genannt, spielt in der objektorientierten Pro-

grammierung eine wichtige Rolle. Angenommen, der Typ `Person` hat die *Untertypen* (subtypes) `Student` und `Angestellter`. Dann ist jedes Objekt vom Typ `Student` oder `Angestellter` auch ein Objekt vom Typ `Person`. An eine Routine mit einem formalen Parameter vom Typ `Person` kann daher auch ein Argument vom Typ `Student` oder `Angestellter` übergeben werden. Die Menge der Objekte vom Typ `Person` enthält alle Objekte der Typen `Student` und `Angestellter`. Die Routine akzeptiert alle Argumente vom Typ t, wobei t eine universell über `Person` und dessen Untertypen quantifizierte Variable ist. Daher zählt auch enthaltender Polymorphismus zum universellen Polymorphismus.

Eine Schnittstelle eines Objekts entspricht im Wesentlichen einem Typ des Objekts. Wenn die Schnittstelle beziehungsweise der Typ eine Methode beschreibt, dann müssen auch alle Schnittstellen, die Untertypen des Typs entsprechen, eine dazu kompatible Methode beschreiben. Eine Methode ist kompatibel, wenn sie überall dort verwendbar ist, wo die ursprüngliche Methode erwartet wird. Diese Einschränkung kann man zur Definition von enthaltendem Polymorphismus durch das *Ersetzbarkeitsprinzip* verwenden:

> Ein Typ U ist ein Untertyp eines Typs T (bzw. T ist ein Obertyp von U) wenn eine Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird.

Der Compiler sieht nur den statischen Typ einer Variablen oder eines formalen Parameters. Der dynamische Typ wird erst während der Ausführung festgelegt. Daher kann der Compiler auch nicht immer feststellen, welche konkrete Methode des in der Variable enthaltenen Objekts gegebenenfalls ausgeführt werden muss, da ja nur die Schnittstelle bekannt ist. Die Methode kann manchmal erst während der Programmausführung festgelegt werden. Dies ist als *dynamisches Binden* (dynamic binding) bekannt. *Statisches Binden* (static binding) bedeutet, dass bereits der Compiler die auszuführende Methode festlegt. In der objektorientierten Programmierung ist man im Zusammenhang mit enthaltendem Polymorphismus auf dynamisches Binden angewiesen. Enthaltenden Polymorphismus und dynamisches Binden werden wir in Kapitel 2 behandeln.

Überladen (overloading): Eine Routine heißt *ad-hoc-polymorph*, wenn sie Argumente mehrerer unterschiedlicher Typen, die in keiner Rela-

tion zueinander stehen müssen, akzeptiert und sich für jeden dieser Typen anders verhalten kann. Beim Überladen bezeichnet ein und derselbe Name verschiedene Routinen, die sich durch die deklarierten Typen ihrer formalen Parameter unterscheiden. Die deklarierten Typen der übergebenen Argumente entscheiden, welche Routine ausgeführt wird. Überladen dient häufig nur der syntaktischen Vereinfachung, da für Operationen mit ähnlicher Funktionalität nur ein gemeinsamer Name vorgesehen werden braucht. Zum Beispiel bezeichnet „/“ sowohl die ganzzahlige Division als auch die Division von Fließkommazahlen, obwohl diese Funktionen sich im Detail sehr stark voneinander unterscheiden. Überladen ist nicht spezifisch für die objektorientierte Programmierung.

Typumwandlung (type coercion): Typumwandlung ist eine semantische Operation. Sie dient zur Umwandlung eines Wertes in ein Argument eines Typs, der von einer Routine erwartet wird. Zum Beispiel wird in C jede Instanz von `char` oder `short` bei der Argumentübergabe in eine Instanz von `int` umgewandelt, wenn der Parametertyp `int` ist. Sprachen wie C++ definieren durch diffizile Regeln, wie Typen umgewandelt werden, wenn zwischen mehreren überladenen Routinen gewählt werden kann. Auch die Typumwandlung ist nicht spezifisch für die objektorientierte Programmierung.

1.1.4 Vererbung

Die *Vererbung* (inheritance) in der objektorientierten Programmierung ermöglicht es, neue Klassen aus bereits existierenden Klassen abzuleiten. Dabei werden nur die Unterschiede zwischen der *abgeleiteten Klasse* (derived class) und der entsprechenden *Basisklasse* (base class), von der abgeleitet wird, angegeben. Die abgeleitete Klasse heißt auch *Unterklasse* (subclass), die Basisklasse *Oberklasse* (superclass). Vererbung erspart ProgrammiererInnen Schreibaufwand. Außerdem werden einige Programmänderungen vereinfacht, da sich Änderungen von Klassen auf alle davon abgeleiteten Klassen auswirken.

In allen 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 Routinen der Oberklasse zuzugreifen.

Diese beiden Änderungsmöglichkeiten sind beliebig kombinierbar.

Im nächsten Beispiel leiten wir, ausgedrückt durch `extends`, eine Klasse `CounterStack` aus `Stack` (in Unterabschnitt 1.1.2) ab:

```
class CounterStack extends Stack {
    private int counter;
    public CounterStack (int sz, int c) {
        super(sz);
        counter = c;
    }
    public void push (String elem) {
        counter = counter + 1;
        super.push(elem);
    }
    public void count() {
        push (Integer.toString(counter));
    }
}
```

(Anmerkungen zu Java)

Der Konstruktor für `CounterStack` ruft durch `super(sz)` den Konstruktor der Oberklasse `Stack` auf und leitet ihm das erste Argument weiter. Das zweite Argument wird zur Initialisierung von `counter` verwendet. Das Wort `super` in einer Unterklasse bezeichnet im Allgemeinen eine entsprechende fiktive Instanz der Oberklasse. Die Methode `push` ist überschrieben. Die neue Methode erhöht `counter` und ruft anschließend die überschriebende Methode auf. Die Methode `pop` ist nicht überschrieben, wird also von `Stack` geerbt. `CounterStack` erweitert `Stack` um `count`. Diese Methode wandelt den Wert von `counter` in eine Zeichenkette um und fügt sie in den Stack ein.

Beim Aufruf von `push` steht kein Objekt, in dem die Methode auszuführen ist. In solchen Fällen beziehen sich Methodenaufrufe immer auf das aktuelle Objekt, also das Objekt, in dem `count` aufgerufen wurde. Das aktuelle Objekt wird durch `this` bezeichnet. Damit ist `push(...)` eine abgekürzte Schreibweise für `this.push(...)`.

In Programmiersprachen wie Java besteht ein enger Zusammenhang zwischen Vererbung und enthaltendem Polymorphismus: Eine Instanz einer Unterklasse kann, zumindest soweit es vom Java-Compiler überprüfbar

ist, überall verwendet werden, wo eine Instanz einer Oberklasse erwartet wird. Änderungsmöglichkeiten bei der Vererbung sind, wie oben beschrieben, eingeschränkt, um die Ersetzbarkeit von Instanzen der Oberklasse durch Instanzen der Unterklasse zu ermöglichen. Es besteht eine direkte Relation zwischen Klassen und Typen: Die Klasse eines Objekts ist gleichzeitig der speziellste Typ des Objekts. Dadurch entspricht eine Vererbungsbeziehung einer Untertypbeziehung. Im Beispiel ist `CounterStack` ein Untertyp von `Stack`. Eine Instanz von `CounterStack` kann überall verwendet werden, wo eine Instanz von `Stack` erwartet wird. Jede Variable vom Typ `Stack` kann auch eine Instanz vom Typ `CounterStack` enthalten.

In Java wird dynamisches Binden verwendet, wie man leicht an der Ausführung folgender Methode sehen kann:

```
public void foo (Stack s) { s.push("foo"); }
```

Bei einem Aufruf von `foo` wird zuerst der spezifischste dynamische Typ, das ist die Klasse von `s` (beziehungsweise vom Argument, das für den formalen Parameter `s` verwendet wird) festgestellt. Ist dies `CounterStack`, wird `push` in `CounterStack` aufgerufen, sonst in `Stack`. Für den Aufruf von `push` wird also nicht automatisch der deklarierte Typ `Stack` verwendet, sondern der dynamische Typ, der oft erst zur Laufzeit bekannt ist.

In Java ist die Vererbung zwischen Klassen auf *Einfachvererbung* (single inheritence) beschränkt; das heißt, jede Unterklasse wird von genau einer anderen Klasse abgeleitet. Die Verallgemeinerung dazu ist *Mehrfachvererbung* (multiple inheritence), wobei jede Klasse von mehreren unterschiedlichen Oberklassen abgeleitet sein kann. Mehrfachvererbung auf Klassen wird zum Beispiel in C++ unterstützt. Neben Klassen gibt es in Java *Interfaces*, auf denen es auch Mehrfachvererbung gibt. Solche Interfaces sind aber nur Schnittstellen, denen andere Eigenschaften von Klassen fehlen, wie wir in Abschnitt 2.4 sehen werden. (Zur Unterscheidung verwenden wir in diesem Skriptum für das Sprachkonstrukt in Java den englischen Begriff, während wir mit dem gleichbedeutenden deutschen Begriff Schnittstellen im Allgemeinen bezeichnen.)

1.2 Qualität in der Programmierung

Die Qualität in der Programmierung gliedert sich in zwei Bereiche:

- die Qualität der erzeugten Programme
- sowie die Effizienz der Erstellung und Wartung der Programme.

Nur wenn die Qualität beider Bereiche zufriedenstellend ist, kann man brauchbare Ergebnisse erwarten. Diese beiden Bereiche sind eng ineinander verflochten: Ein qualitativ hochwertiges Programm erleichtert die Wartung, und eine effiziente Programmerstellung lässt im Idealfall mehr Zeit zur Verbesserung des Programms.

Wir betrachten zunächst die Qualität der Programme und anschließend die Effizienz der Programmerstellung und Wartung. Danach beschäftigen wir uns kurz mit der Qualität von Programmiersprachen.

1.2.1 Qualität von Programmen

Bei der Qualität eines Programms unterscheiden wir zwischen

- der Brauchbarkeit des Programms,
- der Zuverlässigkeit des Programms
- und der Wartbarkeit des Programms.

Die Brauchbarkeit durch die AnwenderInnen steht natürlich an erster Stelle. Nur wenn die AnwenderInnen ihre tatsächlichen Aufgaben mit dem Programm zufriedenstellend lösen können, hat es für die AnwenderInnen einen Wert. Für SoftwareentwicklerInnen ist ein Softwareprojekt in der Regel nur dann (sehr) erfolgreich, wenn der Wert des entwickelten Programms aus Sicht der Benutzer die Entwicklungskosten (stark) übersteigt.

Folgende Faktoren beeinflussen die Brauchbarkeit:

Zweckerfüllung: Die AnwenderInnen möchten mit einem Programm eine gegebene Klasse von Aufgaben lösen. Das Programm erfüllt seinen Zweck nur dann, wenn es genau die Aufgaben lösen kann, für die es eingesetzt wird. Features – das sind Eigenschaften – eines Programms, die AnwenderInnen nicht brauchen, haben keinen Einfluss auf die Zweckerfüllung. Allerdings können nicht benötigte Eigenschaften die Brauchbarkeit durch größeren Resourcenbedarf und schlechtere Bedienbarkeit negativ beeinflussen.

Bedienbarkeit: Die Bedienbarkeit besagt, wie effizient Aufgaben mit Hilfe des Programms lösbar sind und wie hoch der Einlernaufwand ist. Die Bedienbarkeit ist gut, wenn vor allem für häufig zu lösende Aufgaben möglichst wenige Arbeitsschritte nötig sind, keine unerwartet langen Wartezeiten entstehen und zur Bedienung keine besonderen Schulungen notwendig sind. Oft hängt die Bedienbarkeit aber auch von den Gewohnheiten und Erfahrungen der AnwenderInnen ab.

Effizienz des Programmablaufs: Jeder Programmablauf benötigt Resourcen wie Rechenzeit, Hauptspeicher, Plattenspeicher und Netzwerkbandbreite. Ein Programm, das sparsamer mit solchen Resourcen umgeht, hat eine höhere Qualität als ein weniger sparsames. Das gilt auch dann, wenn Computer in der Regel über ausreichend Resourcen verfügen, denn wenn das Programm zusammen mit anderen Anwendungen läuft, können die Resourcen trotzdem knapp werden. Das sparsamere Programm ist unter Umständen auch gleichzeitig mit anderen resourcenverbrauchenden Anwendungen nutzbar.

Neben der Brauchbarkeit ist auch die *Zuverlässigkeit* für die ProgrammanwenderInnen sehr wichtig. Das Programm soll nicht nur manchmal brauchbar sein, sondern die AnwenderInnen sollen sich darauf verlassen können. Fehlerhafte Ergebnisse und Programmabstürze sollen nicht vorkommen. Natürlich ist die geforderte Zuverlässigkeit in hohem Maße von der Art der Anwendung abhängig. Für die Software im Sicherheitssystem eines Kernkraftwerks wird ein weitaus höherer Grad an Zuverlässigkeit gefordert als für ein Textverarbeitungssystem. Absolute Zuverlässigkeit kann aber nie garantiert werden. Da die Zuverlässigkeit ein bedeutender Kostenfaktor ist, gibt man sich bei nicht sicherheitskritischen Anwendungen mit wesentlich geringerer Zuverlässigkeit zufrieden, als erreichbar ist.

Oft lebt ein Programm nur so lange es weiterentwickelt wird. Sobald die Entwicklung und Weiterentwicklung – einschließlich der laufenden Korrekturen von Fehlern und Anpassungen an sich ändernde Bedingungen, das heisst *Wartung* (maintenance) – abgeschlossen ist, kann ein Programm kaum mehr verkauft werden, und die AnwenderInnen beginnen, auf andere Programme umzusteigen. Daraus kann man erkennen, dass gerade bei erfolgreicher Software, die oft über einen langen Zeitraum verwendet wird – also einen langen *Lebenszyklus* hat –, die Kosten für die Wartung einen erheblichen Teil der gesamten Kosten ausmachen. Man schätzt, dass die Wartungskosten bis zu 70 % – bei sehr erfolgreicher Software sogar mehr – der Gesamtkosten ausmachen. Die gute *Wartbarkeit* eines Programms kann die Gesamtkosten also erheblich reduzieren.

Es gibt große Unterschiede in der Wartbarkeit von Programmen. Sie beziehen sich darauf, wie leicht Programme geändert werden können. Folgende Faktoren spielen eine Rolle:

Einfachheit: Ein einfaches Programm ist einfacher verständlich und daher auch einfacher änderbar als ein kompliziertes. Deswegen soll die Komplexität des Programms immer so klein wie möglich bleiben.

Lesbarkeit: Die Lesbarkeit ist gut, wenn es für ProgrammiererInnen leicht ist, durch Lesen des Programms die Logik im Programm zu verstehen und eventuell vorkommende Fehler oder andere zu ändernde Stellen zu entdecken. Die Lesbarkeit hängt zu einem guten Teil vom Programmierstil ab, aber auch von der Programmiersprache.

Lokalität: Der Effekt jeder Programmänderung soll auf einen kleinen Programmteil beschränkt bleiben. Dadurch wird vermieden, dass eine Änderung Programmteile beeinflusst, die auf den ersten Blick gar nichts mit der Änderung zu tun haben. Nicht-lokale beziehungsweise globale Effekte der Änderung – z. B. ein eingefügter Prozeduraufruf überschreibt den Wert einer globalen Variable – werden von ProgrammiererInnen oft nicht gleich erkannt und führen zu Fehlern.

Faktorisierung: Zusammengehörige Eigenschaften und Aspekte des Programms sollen zu Einheiten zusammengefasst werden. In Analogie zur Zerlegung eines Polynoms in ihre Koeffizienten nennt man den Vorgang des Auffindens und Zusammenfassens zusammengehöriger Eigenschaften in einem Programm *Faktorisierung* (factoring). Wenn zum Beispiel mehrere Stellen in einem Programm aus genau denselben Sequenzen von Befehlen bestehen, soll man diese Stellen durch Aufrufe einer Prozedur ersetzen, die genau diese Befehle ausführt. Die Faktorisierung führt dazu, dass zur Änderung aller dieser Stellen auf die gleiche Art und Weise eine einzige Änderung der Prozedur ausreicht. Ohne diese Faktorisierung hätten alle Programmstellen gefunden und einzeln geändert werden müssen, um denselben Effekt zu erreichen. Die Faktorisierung verbessert auch die Lesbarkeit des Programms, wenn die Prozedur einen Namen bekommt, der ihre Bedeutung widerspiegelt.

Objekte dienen durch Kapselung zusammengehöriger Eigenschaften in erster Linie der Faktorisierung des Programms. Durch Zusammenfügen von Daten mit Routinen haben ProgrammiererInnen mehr Freiheiten zur Faktorisierung als in der prozeduralen Programmierung, bei der Daten prinzipiell von Prozeduren getrennt sind (siehe Abschnitt 1.4). Gute Faktorisierung kann die Wartbarkeit eines Programms wesentlich erhöhen.

Zur Klarstellung: Die objektorientierte Programmierung bietet mehr Möglichkeiten zur Faktorisierung als andere Paradigmen und erleichtert damit ProgrammiererInnen, eine für das Problem geeignete Zerlegung in einzelne Komponenten zu finden. Aber die Faktorisierung eines Pro-

gramms erfolgt auf keinen Fall automatisch so, dass alle Zerlegungen in Objekte gut sind. Es ist die Aufgabe der ProgrammiererInnen, gute Zerlegungen von schlechten zu unterscheiden.

Die Lesbarkeit eines objektorientierten 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. Vor allem Namen für Software-Objekte sollen den üblichen Bezeichnungen realer Objekte entsprechen. Dadurch ist das Programm einfacher lesbar, da stets die Analogie zur realen Welt besteht, vorausgesetzt alle EntwicklerInnen haben annähernd dieselben Vorstellungen über die reale Welt. Man darf die Simulation aber nicht zu weit treiben. Die Einfachheit ist wichtiger.

1.2.2 Effizienz der Programmerstellung und Wartung

Die große Zahl der Faktoren, die die Qualität eines Programms bestimmen, machen es ProgrammiererInnen schwer, qualitativ hochwertige Programme zu schreiben. Dazu kommt noch das Problem, dass viele Einflussgrößen zu Beginn der Entwicklung noch nicht bekannt sind. Einige davon sind von SoftwareentwicklerInnen nicht kontrollierbar. Zum Beispiel wissen AnwenderInnen oft nicht genau, welche Eigenschaften des Programms sie zur Lösung ihrer Aufgaben tatsächlich brauchen. Erfahrungen mit dem Programm können sie ja erst sammeln, wenn das Programm existiert.

Ein typischer Softwareentwicklungsprozess umfasst folgende Schritte:

Analyse (analysis): Die Aufgabe, die durch die zu entwickelnde Software gelöst werden soll, wird analysiert. Das Ergebnis, das ist die *Anforderungsdokumentation*, beschreibt die Anforderungen an die Software – was die Software tun soll.

Entwurf (design): Ausgehend von dieser Anforderungsdokumentation wird in der Entwurfsphase das Programm entworfen. Die *Entwurfsdokumentation* beschreibt, wie die Anforderungen erfüllt werden sollen.

Implementierung (implementation): Der Entwurf wird in ein Programm umgesetzt.

Verifikation (verification) und Validierung (validation): Die Verifikation ist die Überprüfung, ob das Programm die in der Anforderungsdokumentation beschriebenen Anforderungen erfüllt. Validie-

rung ist die Überprüfung, wie gut das Programm die Aufgaben der AnwenderInnen tatsächlich löst und ob die Qualität des Programms dessen Weiterentwicklung rechtfertigt.

Im traditionellen *Wasserfallmodell* werden diese Schritte in der gegebenen Reihenfolge durchgeführt, gefolgt von einem Schritt für die Wartung. Solche Softwareentwicklungsprozesse haben den Nachteil, dass die Validierung erst sehr spät erfolgt. Es können also bereits recht hohe Kosten angefallen sein, bevor festgestellt werden kann, ob die entwickelte Software für die AnwenderInnen überhaupt brauchbar ist. Das Risiko ist groß. Bei kleinen Projekten und in Fällen, in denen die Anforderungen sehr klar sind, kann das Wasserfallmodell durchaus vorteilhaft sein.

Heute verwendet man eher *zyklische Softwareentwicklungsprozesse*. Dabei werden die oben genannten Schritte in einem Zyklus wiederholt ausgeführt. Zuerst wird nur ein kleiner, aber wesentlicher Teil der durchzuführenden Aufgabe analysiert und ein entsprechendes Programm entworfen, implementiert, verifiziert und validiert. Im nächsten Zyklus wird das Programm erweitert, wobei die Erfahrungen mit dem ersten Programm in die Analyse und den Entwurf einfließen. Diese Zyklen werden fortgesetzt, solange das Programm lebt, also auch zur Wartung. In der Praxis werden die Zyklen und die einzelnen Schritte in den Zyklen jedoch meist nicht streng in der beschriebenen Reihenfolge durchgeführt, sondern häufig überlappend. Das heißt, es wird gleichzeitig analysiert, entworfen, implementiert und überprüft. Wenn man mit einem kleinen Teil des Programms beginnt und das Programm schrittweise ausweitet, spricht man auch von *schrittweiser Verfeinerung*. Die Vorteile solcher Entwicklungsprozesse liegen auf der Hand: Man kann bereits recht früh auf Erfahrungen mit dem Programm zurück greifen, und die Gefahr, dass unter hohem Aufwand im Endeffekt nicht gebrauchte Eigenschaften in das Programm eingebaut werden, ist kleiner. Insgesamt ist das Risiko also kleiner, aber der Fortschritt eines Softwareprojekts ist nur schwer planbar.

In der Praxis eingesetzte Softwareentwicklungsprozesse unterscheiden sich im Detail stark voneinander. Praktisch jedes Unternehmen hat seine eigenen Standards dafür. Alles vom starren Wasserfallmodell bis zu sehr dynamischen zyklischen Prozessen kommt vor.

Qualitätsunterschiede zwischen einzelnen Softwareentwicklungsprozessen sind kaum greifbar, da viele Faktoren mitspielen und es nur wenige vergleichbare Daten gibt. Zum Beispiel hängt die Qualität eines bestimmten Prozesses von der Art der Softwareprojekte ebenso ab wie von der inter-

nen Unternehmenskultur – Organisationsstrukturen, Art der Mitarbeiter, etc. – und der Art der Zusammenarbeit mit Kunden und Partnern.

Jedes Unternehmen ist natürlich bestrebt, die eigenen Entwicklungsprozesse zu verbessern. Sobald irgendwo ein kleines Problem auftaucht, wird es gelöst. Gerade solche oft durchgeführten kleinen Anpassungen führen schließlich zu einem konkurrenzfähigen Softwareentwicklungsprozess. Generell gilt, dass nur ein gut an die tatsächlichen Gegebenheiten angepasster Prozess von hoher Qualität ist. In der Regel funktioniert es nicht, wenn ein Unternehmen einen Softwareentwicklungsprozess von einem anderen Unternehmen übernimmt, ohne ihn an die eigenen Gegebenheiten anzupassen.

Einige Kriterien für gute Softwareentwicklung sind nicht spezifisch für die Softwarebranche. Zum Beispiel ist die ständige Kontrolle, ob die eingesetzten Prozesse geeignet sind, ebenso wichtig wie ein gutes Gesprächsklima unter Mitarbeitern; das heißt, dass effizient Informationen ausgetauscht werden, nicht dass ständig getratscht wird. Mitarbeiter sollen genau wissen, was sie zu tun haben und worauf es ankommt.

1.2.3 Qualität von Programmiersprachen

In erster Linie hängt die Qualität von Programmen und die Effizienz der Programmerstellung und Wartung von den Mitarbeitern und Managementstrukturen ab. Aber auch die Qualität der eingesetzten Programmierwerkzeuge und die Programmiersprache spielen eine nicht zu unterschätzende Rolle. Eine Programmiersprache soll es den ProgrammiererInnen leicht machen, gute Programme zu schreiben. Dabei soll die Sprache den Softwareentwicklungsprozess möglichst gut unterstützen. Das sind einige allgemeine Qualitätsmerkmale von Programmiersprachen:

Zuverlässigkeit: Die Wahrscheinlichkeit des Auftretens eines Fehlers soll klein sein, und sogar wenn ein Fehler auftritt, sollen AnwenderInnen noch immer so weit wie möglich bedient werden – *Fehlertoleranz.* Sprachen unterstützen die Zuverlässigkeit durch folgende Kriterien:

Schreibbarkeit: Ein Programm soll so ausgedrückt werden können, wie es dem Problem entspricht. Details und Tricks einer Sprache sollen die ProgrammiererInnen nicht von der Lösung des eigentlichen Problems abhalten. Je leichter sich ProgrammiererInnen auf das Problemlösen konzentrieren kann, desto weniger fehleranfällig ist das Programm.

Lesbarkeit: Natürlich hängt auch die Lesbarkeit eines Programms stark von der Programmiersprache ab. Die Lesbarkeit ist wichtig für die Wartbarkeit. Gute Lesbarkeit hilft beim Entdecken von Fehlern. Ausdrucksstarke Sprachen können die Schreibbarkeit und Lesbarkeit verbessern helfen.

Einfachheit: Eine einfache Sprache ist einfach handhabbar und ermöglicht ProgrammiererInnen, Algorithmen so einfach auszudrücken, dass keine Zweifel an deren Korrektheit entstehen.

Sicherheit: Die Programmiersprache soll keine Eigenschaften haben, hinter denen gefährliche Fallen versteckt sind. Zum Beispiel ist bekannt, dass die Verwendung von nicht überprüften Typumwandlungen gefährlich ist und daher nicht unterstützt werden soll; zumindest sollen gute Alternativen dafür angeboten werden. Leider sind sichere Programmierspracheigenschaften oft weniger ausdrucksstark und flexibel als gefährliche.

Robustheit: Programmiersprachen sind robust, wenn sie Möglichkeiten zum Umgang mit unvorhergesehenen Ereignissen anbieten – z. B. arithmetischer Überlauf, Datei lässt sich nicht öffnen, etc. Das Verhalten des Systems wird dadurch auch beim Auftreten unerwarteter Ereignisse vorhersehbar.

Wartbarkeit der Software: Software soll wartbar, also möglichst einfach änderbar sein. Programmiersprachen unterstützen die Wartbarkeit vor allem durch Lesbarkeit und Einfachheit, aber auch durch Sprachunterstützung für Faktorisierung und Lokalität.

Effizienz: Nicht nur Compiler und andere Werkzeuge, sondern auch die damit erzeugten Programme sollen effizient sein. Programmiersprachen unterscheiden sich recht stark in diesen Punkten. In vielen Fällen wesentlich wichtiger ist aber die Effizienz in der Programmerstellung und Wartung. Die *Portabilität* des Programms, die von den ProgrammiererInnen, aber auch von der Programmiersprache, deren Standards und deren Unterstützung durch die Softwareindustrie abhängt, ist in diesem Zusammenhang ein bedeutender Faktor.

1.3 Rezept für gute Programme

Der Titel dieses Abschnitts ist ironisch zu verstehen. Niemand kann ein allgemeingültiges Rezept dafür angeben, wie man gute Programme

schreibt. Dafür ist die Softwareentwicklung in ihrer Gesamtheit viel zu komplex. Nach wie vor ist die Programmierung eine Kunst – vor allem die Kunst, trotz unvollständigem Wissen über künftige Anforderungen, trotz vieler widersprüchlicher Zielsetzungen und oft unter großem Zeitdruck Lösungen zu entwickeln, die über einen längeren Zeitraum brauchbar sind. Das ist keine leichte Aufgabe. Ein einfaches Rezept, das immer zu guten Ergebnissen führt sofern man alle vorgeschriebenen Schritte korrekt durchführt, wird es vermutlich nie geben.

Trotzdem hat sich in den vergangenen Jahrzehnten auch in der Programmierung ein umfangreicher Erfahrungsschatz darüber entwickelt, mit welchen Problemen man in Zukunft rechnen muss, wenn man eine Aufgabenstellung auf eine bestimmte Art und Weise löst. Gute ProgrammiererInnen werden diese Erfahrungen gezielt einsetzen. Eine Garantie für den Erfolg eines Softwareprojekts gibt es natürlich trotzdem nicht. Aber die Wahrscheinlichkeit, dass EntwicklerInnen die Komplexität des Projekts meistern können, steigt. Damit können auch komplexere Aufgabenstellungen mit vertretbaren Erfolgsaussichten in Angriff genommen werden.

Gerade in der objektorientierten Programmierung ist es wichtig, dass EntwicklerInnen Erfahrungen gezielt einsetzen. Objektorientierte Sprachen bieten viele unterschiedliche Möglichkeiten zur Lösung von Aufgaben. Jede Lösungsmöglichkeit hat andere charakteristische Merkmale. Erfahrene EntwicklerInnen werden jene Möglichkeit wählen, deren Merkmale in späterer Folge am ehesten hilfreich sind. Weniger erfahrene EntwicklerInnen verwenden einfach nur die Lösungsmöglichkeit, die sie zuerst entdecken. Damit verzichten sie auf einen wichtigen Vorteil der objektorientierten Programmierung gegenüber einigen anderen Paradigmen. Generell kann man sagen, dass die objektorientierte Programmierung durch erfahrene EntwicklerInnen derzeit wahrscheinlich das erfolgversprechendste Paradigma der Programmierung überhaupt darstellt, andererseits aber GelegenheitsprogrammiererInnen und noch unerfahrene SoftwareentwicklerInnen oft überfordert.

1.3.1 Zusammenhalt und Kopplung

Ein gutes Programm erfüllt die Kriterien, die wir in Unterabschnitt 1.2.1 beschrieben haben. Leider sind einige wichtige Kriterien in der Entwurfsphase und während der Implementierung noch nicht bewertbar. Sie stellen sich erst später heraus. SoftwareentwicklerInnen müssen aber in jeder Phase wissen, wie sie vorgehen müssen, um möglichst hochwertige Soft-

ware zu produzieren. Vor allem eine gute Faktorisierung des Programms ist ein entscheidendes Kriterium. Daher gibt es einige Faustregeln, die EntwicklerInnen dabei unterstützen. Wir wollen hier zwei wichtige, eng miteinander verknüpfte Faustregeln betrachten, die in vielen Fällen den richtigen Weg zu einer guten Faktorisierung weisen. Zuvor müssen wir einige Begriffe einführen:

Verantwortlichkeiten (responsibilities): Wir können die Verantwortlichkeiten einer Klasse durch drei w-Ausdrücke beschreiben:

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

Das Ich steht dabei jeweils für die Klasse. Die Klasse muss die Verantwortung für diese Verantwortlichkeiten übernehmen. Wenn etwas geändert werden soll, das in den Verantwortlichkeiten einer Klasse liegt, dann sind dafür die EntwicklerInnen der Klasse zuständig.

Klassen-Zusammenhalt (class coherence): Der Zusammenhalt einer Klasse ist der Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse. Dieser Grad der Beziehungen ist zwar nicht einfach meßbar, oft aber intuitiv einfach fassbar. Der Zusammenhalt ist offensichtlich hoch, wenn alle Variablen und Methoden der Klasse eng zusammenarbeiten und durch den Namen der Klasse gut beschrieben sind. Das heißt, einer Klasse mit hohem Zusammenhalt würde etwas Wichtiges fehlen, wenn man beliebige Variablen oder Methoden entfernt; außerdem würde der Zusammenhalt niedriger, wenn man die Klasse sinnändernd umbenennt.

Objekt-Kopplung (object coupling): Unter der Objekt-Kopplung versteht man die Abhängigkeit der Objekte voneinander. Die Objekt-Kopplung ist stark, wenn

- die Anzahl der nach außen sichtbaren Methoden und Variablen hoch ist,
- im laufenden System Nachrichten (beziehungsweise Methodenaufrufe und Variablenzugriffe) zwischen unterschiedlichen Objekten häufig auftreten
- und die Anzahl der Parameter dieser Methoden hoch ist.

Das sind die Faustregeln:

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, dass bei Programmänderungen auch die Zerlegung in Klassen und Objekte geändert werden muss (*Refaktorisierung*, refactoring), kleiner.

Faustregel: Die Objekt-Kopplung soll schwach sein.

Schwache Objekt-Kopplung deutet auf eine gute Kapselung hin. Dadurch beeinflussen Programmänderungen wahrscheinlich weniger Objekte unnötig. Beeinflussungen durch unvermeidbare Abhängigkeiten zwischen Objekten sind natürlich unumgänglich.

Klassen-Zusammenhalt und Objekt-Kopplung stehen in einer engen Beziehung zueinander. Wenn der Klassen-Zusammenhalt hoch ist, dann ist oft auch 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 oft relativ leicht, bereits in einer frühen Phase der Softwareentwicklung zu erkennen, auf welche Art und Weise ein hoher Klassen-Zusammenhalt und eine schwache Objekt-Kopplung erreichbar sein wird. Die Simulation der realen Welt hilft dabei vor allem zu Beginn der Softwareentwicklung.

Wenn EntwicklerInnen sich zwischen mehreren Alternativen zu entscheiden haben, können Klassen-Zusammenhalt und Objekt-Kopplung der einzelnen Alternativen einen wichtigen Beitrag zur Entscheidungsfindung liefern. Der erwartete Klassen-Zusammenhalt sowie die erwartete Objekt-Kopplung jeder Alternative lässt sich im direkten Vergleich meist relativ leicht prognostizieren. Klassen-Zusammenhalt und Objekt-Kopplung sind Faktoren in der Bewertung von Alternativen. In manchen Fällen können jedoch andere Faktoren ausschlaggebend sein.

Auch noch so erfahrene EntwicklerInnen werden es normalerweise nicht schaffen, auf Anhieb einen optimalen Entwurf für ein Programm zu liefern, in dem die Zerlegung in Objekte später nicht mehr geändert werden braucht. Normalerweise muss die Zerlegung einige Male geändert werden; man spricht von Refaktorisierung. Eine Refaktorisierung ändert die Struktur eines Programms, lässt aber dessen Funktionalität unverändert. Es wird dabei also nichts dazugefügt oder weggelassen, und es werden auch keine inhaltlichen Änderungen vorgenommen. Solche Refaktorisierungen

sind vor allem in einer frühen Projektphase ohne größere Probleme und Kosten möglich. Glücklicherweise ist es oft so, dass einige wenige gezielt durchgeführte Refaktorisierungen sehr rasch zu einer stabilen Zerlegung der davon betroffenen Programmteile in Objekte führen und später diese stabilen Teile des Programms kaum noch refaktorisiert werden brauchen. Es geht also gar nicht darum, von Anfang an einen optimalen Entwurf zu haben, sondern eher darum, ständig alle nötigen Refaktorisierungen durchzuführen bevor sich Probleme, die durch die Refaktorisierungen beseitigt werden, über das ganze Programm ausbreiten. Natürlich dürfen Refaktorisierungen auch nicht so häufig durchgeführt werden, dass bei der inhaltlichen Programmentwicklung kein Fortschritt mehr erkennbar ist.

1.3.2 Wiederverwendung

Ein wichtiger Begriff im Zusammenhang mit effizienter Softwareentwicklung ist die *Wiederverwendung* (reuse). Es ist sinnvoll, bewährte Software so oft wie möglich wiederzuverwenden. Das spart Entwicklungsaufwand. Wir müssen aber zwischen zahlreichen Arten der Wiederverwendung unterscheiden. Hier sind einige Arten von Software, die wiederverwendet werden können:

Programme: Die meisten Programme werden im Hinblick darauf entwickelt, dass sie häufig (wieder)verwendet werden. Dadurch zahlt es sich erst aus, einen großen Aufwand zu betreiben, um die Programme handlich und effizient zu machen. Es gibt aber auch Programme, die nur für die einmalige Verwendung bestimmt sind.

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 Erfahrungen werden oft zwischen sehr unterschiedlichen Projekten ausgetauscht.

Code: Wenn man von Wiederverwendung spricht, meint man oft automatisch die Wiederverwendung von Programmcode. Viele Konzepte von Programmiersprachen, wie zum Beispiel enthaltender Polymorphismus, Vererbung und Generizität, wurden insbesondere im Hinblick auf die Wiederverwendung von Code entwickelt. Man kann mehrere

Arten der Codewiederverwendung mit verschiedenen Wiederverwendungshäufigkeiten unterscheiden:

Globale Bibliotheken: Einige Klassen in allgemein verwendbaren Klassenbibliotheken – zum Beispiel als Standardbibliotheken zusammen mit Programmierwerkzeugen oder separat erhältlich – werden sehr häufig (wieder)verwendet. Allerdings kommen nur wenige, relativ einfache Klassen für die Aufnahme in solche Bibliotheken in Frage. Die meisten etwas komplexeren Klassen sind nur in bestimmten Bereichen sinnvoll einsetzbar und daher für die Allgemeinheit nicht brauchbar.

Fachspezifische Bibliotheken: Komplexere Klassen und *Komponenten* – größere Einheiten, die aus mehreren Klassen zusammengesetzt sind – lassen sich in fach- oder auch firmenspezifischen Bibliotheken unterbringen. Ein Beispiel dafür sind Bibliotheken für grafische Benutzeroberflächen. Auch mit solchen Bibliotheken lässt sich manchmal ein hoher Grad an Wiederverwendung erreichen, aber wieder sind die am häufigsten wiederverwendeten Klassen und Komponenten eher einfacherer Natur.

Projektinterne Wiederverwendung: Zu einem hohen Grad spezialisierte Klassen und Komponenten lassen sich oft nur innerhalb eines Projekts, zum Beispiel in unterschiedlichen Versionen eines Programms, wiederverwenden. Obwohl der damit erzielbare Grad der Wiederverwendung nicht sehr hoch ist, ist diese Art der Wiederverwendung bedeutend: Wegen der höheren Komplexität der wiederverwendeten Software erspart bereits eine einzige Wiederverwendung viel Arbeit.

Programminterne Wiederverwendung: Ein und derselbe Programmcode kann in einem Programm sehr oft wiederholt ausgeführt werden, auch zu unterschiedlichen Zwecken. Durch die Verwendung eines Programmteils für mehrere Aufgaben wird das Programm einfacher, kleiner und leichter wartbar.

Gute SoftwareentwicklerInnen werden nicht nur darauf schauen, dass sie so viel Software wie möglich wiederverwenden, sondern auch darauf, dass neu entwickelte Software einfach wiederverwendbar wird. Die Erfahrung zeigt, dass durch objektorientierte Programmierung tatsächlich Codewiederverwendung erzielbar ist. Kosteneinsparungen ergeben sich dadurch aber normalerweise nur wenn

- SoftwareentwicklerInnen ausreichend erfahren sind, um die Möglichkeiten der objektorientierten Programmierung optimal zu nutzen
- und Zeit in die Wiederverwendbarkeit investiert wird.

Weniger erfahrene EntwicklerInnen investieren oft zu wenig oder zu viel oder an falscher Stelle in die Wiederverwendbarkeit von Klassen und Komponenten. Solche Fehlentscheidungen können sich später rächen und durch lange Entwicklungszeiten sogar zum Scheitern eines Projekts führen. Im Zweifelsfall soll man anfangs eher weniger in die Wiederverwendbarkeit investieren, diese Investitionen zum Beispiel durch Refaktorisierung aber nachholen, sobald sich ein Bedarf dafür ergibt.

1.3.3 Entwurfsmuster

Erfahrung ist eine wertvolle Resource zur effizienten Erstellung und Wartung von Software. Am effizientesten ist es, gewonnene Erfahrung in Programmcode auszudrücken und diesen Code direkt wiederzuverwenden. Aber in vielen Fällen funktioniert Codewiederverwendung nicht. In diesen Fällen muss man zwar den Code neu schreiben, kann dabei aber auf Erfahrung zurück greifen.

In erster Linie betrifft die Wiederverwendung von Erfahrung natürlich die persönlichen Erfahrungen der SoftwareentwicklerInnen. Aber auch kollektive Erfahrung ist von großer Bedeutung. Gerade für den Austausch kollektiver Erfahrung können Hilfsmittel nützlich sein.

In den letzten Jahren sind sogenannte *Entwurfsmuster* (design patterns) als ein solches Hilfsmittel populär geworden. Entwurfsmuster geben im Softwareentwurf immer wieder auftauchenden Problemstellungen und deren Lösungen Namen, damit die EntwicklerInnen einfacher darüber sprechen können. Außerdem beschreiben Entwurfsmuster, welche Eigenschaften man sich von den Lösungen erwarten kann. EntwicklerInnen, die einen ganzen Katalog möglicher Lösungen für ihre Aufgaben entweder in schriftlicher Form oder nur abstrakt vor Augen haben, können gezielt jene Lösungen auswählen, deren Eigenschaften den erwünschten Eigenschaften der zu entwickelnden Software am ehesten entsprechen. Kaum eine Lösung wird nur gute Eigenschaften haben. Häufig wählt man daher jene Lösung, deren Nachteile man am ehesten für akzeptabel hält.

Jedes Entwurfsmuster besteht im Wesentlichen aus vier Elementen:

Name: Der Name ist wichtig, damit man in einem einzigen Wort ein Problem und dessen Lösung sowie Konsequenzen daraus ausdrücken

kann. Damit kann man den Softwareentwurf auf eine höhere Ebene verlagern; man braucht nicht mehr jedes Detail einzeln ansprechen. Der Name ist auch nützlich, wenn man über ein Entwurfsmuster diskutiert. Es ist gar nicht leicht, solche Namen für Entwurfsmuster zu finden, die jeder mit dem Entwurfsmuster identifiziert. Wir verwenden hier Singleton als Beispiel für ein einfaches Entwurfsmuster.

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. Für Singleton lautet die Beschreibung folgendermaßen: „Ein Singleton definiert eine Klassen-Operation – eine statische Methode in Java –, die Zugriff auf die einzige Instanz der Klasse gewährt; ein Singleton kann für die Erzeugung seiner einzigen Instanz verantwortlich sein.“

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 Singleton-Beispiel enthält die Beschreibung Erklärungen dafür, wie man in bestimmten Programmiersprachen sicherstellt, dass nur eine Instanz der Klasse erzeugt wird und wie man trotzdem erreicht, dass Vererbung möglich ist.

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 Eigenschaft von Singleton ist, dass man die Entscheidung dafür, dass es nur eine Instanz einer Klasse gibt, nachträglich leicht ändern kann. Oft ist diese leichte Änderbarkeit von Vorteil. Wenn aber zum Beispiel aus Sicherheitsgründen diese Entscheidung auf keinen Fall nachträglich geändert werden soll, ist Singleton kein geeignetes Entwurfsmuster.

Entwurfsmuster scheinen oft die Lösung vieler Probleme zu sein, da man nur mehr aus einem Katalog von Mustern 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 ist sehr komplex und undurchsichtig. Damit ist die Programmerstellung langwierig und die Wartung schwierig, obwohl die über den Einsatz der Entwurfsmuster erzielten Eigenschaften anderes versprechen. SoftwareentwicklerInnen 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.

1.4 Paradigmen der Programmierung

Unter einem *Paradigma* der Programmierung versteht man im Wesentlichen einen Stil, in dem Programme geschrieben werden. Die meisten Programmiersprachen unterstützen einen bestimmten Stil besonders gut und weisen dafür charakteristische Merkmale auf. Am effektivsten wird man die Programmiersprache nutzen, wenn man Programme unter diesem Paradigma schreibt, also den durch die Programmiersprache zumindest zum Teil vorgegebenen Stil einhält. Dieses Paradigma soll natürlich mit der verwendeten Softwareentwicklungsmethode kompatibel sein.

Eine der wichtigsten Unterteilungen zwischen Programmiersprachen anhand ihrer vorherrschenden Paradigmen ist die zwischen *imperativen* und *deklarativen* Programmiersprachen.

1.4.1 Imperative Programmierung

Die *Rechnerarchitektur* hinter der imperativen Programmierung beruht auf einem hardwarenahen Berechnungsmodell wie beispielsweise der von Neumann-Architektur: Eine CPU (central processing unit) ist über einen Bus mit einem Speichermodul verbunden. Die CPU führt zyklisch folgende Schritte aus: Ein Maschinenbefehl wird aus dem Speicher geladen und ausgeführt, und anschließend werden die Ergebnisse in den Speicher geschrieben. Praktisch alle derzeit verwendeten Computer beruhen auf der von Neumann-Architektur oder einer ähnlichen Architektur.

Imperative Programmiersprachen werden dadurch charakterisiert, dass Programme aus *Anweisungen* – das sind Befehle – aufgebaut sind. Diese werden in einer festgelegten Reihenfolge ausgeführt, in parallelen imperativen Sprachen teilweise auch gleichzeitig beziehungsweise überlappend. Grundlegende Sprachelemente sind Variablen, Konstanten, Rou-

tinen und meist auch Typen. Der wichtigste Befehl in diesen Sprachen ist die *destruktive Zuweisung*: Eine Variable bekommt einen neuen Wert, unabhängig vom Wert, den sie vorher hatte. Die Menge der Werte in allen Variablen im Programm sowie ein Zeiger auf den nächsten auszuführenden Befehl beschreiben den Programmzustand, der sich mit der Ausführung jeder Anweisung ändert.

Im Laufe der Zeit entwickelte sich eine ganze Reihe von Paradigmen von Programmiersprachen aufbauend auf der imperativen Programmierung. Unterschiede zwischen diesen Paradigmen beziehen sich hauptsächlich auf die Strukturierung von Programmen. Die wichtigsten imperativen Paradigmen sind die prozedurale und objektorientierte Programmierung:

Prozedurale Programmierung: Das ist der konventionelle Programmierstil. Der wichtigste Abstraktionsmechanismus in prozeduralen Sprachen wie z. B. Algol, Fortran, Cobol, C, Pascal, Modula-2 und Ada 83 ist die Prozedur. Programme werden entsprechend den vorgegebenen Algorithmen in sich gegenseitig aufrufende, den Programmzustand verändernde Prozeduren zerlegt. Programmzustände werden im Wesentlichen als global angesehen; das heißt, Daten können an beliebigen Stellen im Programm verändert werden. Saubere prozedurale Programme werden mittels *strukturierter Programmierung* geschrieben.

Objektorientierte Programmierung: Die objektorientierte Programmierung ist eine Weiterentwicklung der strukturierten prozeduralen Programmierung, die den Begriff des Objekts in den Mittelpunkt stellt. Wie bei prozeduralen Sprachen erfolgen Zustandsänderungen von Objekten durch destruktive Zuweisungen. Der wesentliche Unterschied zur prozeduralen Programmierung ist der, dass zusammengehörende Operationen und Daten zu Objekten zusammengefasst werden. In vielen Fällen ist es möglich, die Programmausführung anhand der Zustandsänderungen in den einzelnen Objekten zu beschreiben, ohne globale Änderungen der Programmzustände betrachten zu müssen. Das ist vor allem bei der Wartung vorteilhaft. Man unterscheidet zwischen aktiven und passiven Objekten. Aktive Objekte sind nebenläufige (concurrent) Prozesse beziehungsweise threads, die sich tatsächlich durch den Austausch von Nachrichten in einem Netzwerk miteinander verständigen. Die meisten Programmiersprachen beruhen auf passiven Objekten, bei denen das Senden einer Nachricht in etwa einem Prozeduraufruf entspricht. Als weitere Abstrak-

tionsmechanismen neben Objekten bieten objektorientierte Sprachen (wie Java, C++, Eiffel, Smalltalk, Modula-3 und Ada 95) Klassen und Vererbung oder vergleichbare Sprachkonstrukte.

In Abschnitt 1.1 haben wir uns auf die Programmierung mit passiven Objekten konzentriert, da aktive Objekte in der Praxis nur selten eingesetzt werden und SoftwareentwicklerInnen, die den Umgang mit passiven Objekten beherrschen, vermutlich kaum Probleme mit aktiven Objekten haben. Die grundlegenden Konzepte sind überall dieselben.

1.4.2 Deklarative Programmierung

Deklarative Programme beschreiben Beziehungen zwischen Ausdrücken in einem System. Es gibt keine zustandsändernden Anweisungen. Statt zeitlich aufeinander folgender Zustände gibt es ein sich nicht mit der Zeit änderndes Geflecht von Beziehungen zwischen Ausdrücken. In deklarativen Programmen kann man Zustände simulieren, indem man Zustände durch Ausdrücke darstellt und durch Vorher-nachher-Beziehungen verknüpft. Deklarative Sprachen entstanden aus mathematischen Modellen und stehen meist auf einem höheren Abstraktionsniveau als imperative Sprachen, die sich aus Maschinensprachen und hardwarenahen Berechnungsmodellen entwickelten. Grundlegende Sprachelemente sind Symbole, die sich manchmal in mehrere Gruppen wie Variablensymbole, Funktionssymbole und Prädikate einteilen lassen. Daher spricht man auch von *symbolischer Programmierung.*

Die wichtigsten Paradigmen in der deklarativen Programmierung sind die funktionale und logikorientierte Programmierung:

Funktionale Programmierung: Eines der für die Informatik bedeutendsten theoretischen Modelle ist der *Lambda-Kalkül*, der den mathematischen Begriff *Funktion* formal definiert. Programmiersprachen, die auf diesem Kalkül beruhen, heissen funktionale Sprachen. Beispiele sind Lisp, ML, Miranda und Haskell. Alle Ausdrücke in diesen Sprachen werden als Funktionen aufgefasst, und der wesentliche Berechnungsschritt besteht in der Anwendung einer Funktion auf einen Ausdruck. Der Lambda-Kalkül hat auch die historische Entwicklung der imperativen Sprachen beeinflußt. Manchmal werden funktionale Sprachen als saubere Varianten prozeduraler Sprachen angesehen, die ohne unsaubere destruktive Zuweisung auskommen. Durch das Fehlen der destruktiven Zuweisung und anderer Seiten-

effekte haben funktionale Programme eine wichtige Eigenschaft, die als *referentielle Transparenz* bezeichnet wird: Ein Ausdruck in einem Programm bedeutet immer dasselbe, egal wann und wie der Ausdruck ausgewertet wird. Im Gegensatz zu anderen Paradigmen brauchen ProgrammiererInnen nicht zwischen einem Objekt und der Kopie des Objekts unterscheiden; solche Unterschiede werden nirgends sichtbar.

Logikorientierte Programmierung: Sprachen für die logikorientierte Programmierung beruhen auf einer (mächtigen) Teilmenge der Prädikatenlogik erster Stufe. Die Menge aller wahren Aussagen in einem Modell wird mittels Fakten und Regeln beschrieben. Um einen Berechnungsvorgang zu starten, wird eine Anfrage gestellt. Das Ergebnis der Berechnung besagt, ob und unter welchen Bedingungen die in der Anfrage enthaltene Aussage wahr ist. Der wichtigste Vertreter dieser Sprachen, Prolog, hat eine prozedurale Interpretation; das heisst, Fakten und Regeln können als Prozeduren aufgefasst und wie in prozeduralen Sprachen ausgeführt werden. Spezielle Varianten der logikorientierten Programmierung spielen bei Datenbankabfragesprachen eine bedeutende Rolle.

1.4.3 Paradigmen für Modularisierungseinheiten

Programmierparadigmen beziehen sich nicht nur auf das zu Grunde liegende Rechenmodell – hardwarenahes Modell, Lambda-Kalkül oder Prädikatenlogik –, sondern auch auf die Art und Weise, wie größere Programme und Programmteile in kleinere Einheiten zerlegt werden, also auf die Faktorisierung. Folgende Paradigmen können im Prinzip sowohl mit imperativen als auch deklarativen Paradigmen kombiniert werden:

Programmierung mit Modulen: Dieses Paradigma legt großen Wert auf Modularisierungseinheiten, das sind Gruppierungen von Variablen, Prozeduren, Funktionen, Typen, Klassen, etc. Ein Programm besteht aus einer Menge solcher Module. In einem Modul kann man angeben, welche Dienste das Modul nach außen hin anbietet. Modula-2 und Ada sind für die Programmierung mit Modulen bekannt. Module sind unverzichtbar, wenn größere Programme in kleinere Einheiten zerlegt werden sollen. Anders als in der objektorientierten Programmierung gibt es bei der Programmierung mit Modulen keine Unterscheidung zwischen Klassen und Objekten.

Programmierung mit abstrakten Datentypen: Ein abstrakter Datentyp, kurz ADT, versteckt die interne Darstellung seiner Instanzen. Nach außen hin wird eine Instanz als abstraktes Objekt ohne innere Struktur, beispielsweise als Adresse repräsentiert. Auf diesen Objekten sind nur die vom ADT exportierten Operationen anwendbar. Manchmal wird ein ADT als Modul implementiert, das Instanzen als Zeiger auf Instanzen eines nicht exportierten Typs darstellt. Diese Zeiger und einige darauf anwendbare Routinen werden exportiert. In vielen objektorientierten Sprachen ist ein ADT ein Verbund (record), der neben Daten auch Routinen enthält. Einige Komponenten werden durch ein Typsystem vor Zugriffen von außen geschützt. Im Großen und Ganzen entsprechen Klassen abstrakten Datentypen. Die Programmierung mit abstrakten Datentypen entspricht der objektorientierten Programmierung, abgesehen davon, dass es keinen enthaltenden Polymorphisumus und keine Vererbung gibt.

Generische Programmierung: Dieses Paradigma unterstützt die Entwicklung *generischer* Abstraktionen als modulare Programmeinheiten auf einer sehr hohen Ebene. Generische Einheiten werden zur Compilations- oder Laufzeit zu konkreten Datenstrukturen, Klassen, Typen, Funktionen, Prozeduren, etc. instanziert, die im Programm benötigt werden. Die generische Programmierung wird vor allem mit der objektorientierten Programmierung und der funktionalen Programmierung kombiniert. Zum Beispiel ist `List<a>` eine generische Klasse mit dem generischen Typparameter `a`, für den beliebige Typen eingesetzt werden können. Die konkreten Klassen `List<int>`, `List<float>` und `List<Person>` werden durch Instanzierung der generischen Klasse erzeugt, wobei `int`, `float` und `Person` den Typparameter ersetzen. Wir haben die Klassen der Listen von ganzen Zahlen, Fließkommazahlen und Personen aus einer einzigen generischen Klasse erzeugt, die im Programm nur einmal definiert ist.

Manchmal wird auch die objektorientierte Programmierung zu den Paradigmen gezählt, die beliebig mit imperativen und deklarativen Paradigmen kombinierbar sind. Tatsächlich gibt es funktionale und logikorientierte Sprachen, die auch die objektorientierte Programmierung unterstützen. Derzeit ist die deklarative objektorientierte Programmierung mit vielen Problemen behaftet oder beruht auf einer in eine deklarative Programmiersprache eingebetteten imperativen Teilsprache. In der Praxis wird nur die imperative objektorientierte Programmierung verwendet.

Kapitel 2

Enthaltender Polymorphismus und Vererbung

Vererbung und enthaltender Polymorphismus sind auf Grund ihrer Definitionen zwei sehr unterschiedliche Konzepte, die aber häufig in einem einzigen Sprachkonstrukt zusammengefasst sind: Vererbung ist auf solche Weise eingeschränkt, dass sie auch die wichtigsten Anforderungen des enthaltenden Polymorphismus erfüllen kann.

In diesem Kapitel werden wir zunächst in Abschnitt 2.1 die Grundlagen des enthaltenden Polymorphismus untersuchen. In Abschnitt 2.2 gehen wir auf einige wichtige Aspekte des Objektverhaltens ein, die ProgrammiererInnen bei der Verwendung von enthaltendem Polymorphismus beachten müssen. Danach betrachten wir in Abschnitt 2.3 einige Aspekte der Vererbung, vor allem im Zusammenhang mit Codewiederverwendung. Schließlich behandeln wir in Abschnitt 2.4 Klassen, Vererbung und das Konzept der Interfaces in Java.

2.1 Das Ersetzbarkeitsprinzip

Die wichtigste Grundlage des enthaltenden Polymorphismus ist das Ersetzbarkeitsprinzip:

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

Über dieses Ersetzbarkeitsprinzip definieren wir enthaltenden Polymorphismus. Daher ist, per Definition, eine Instanz eines Untertyps überall

verwendbar, wo eine Instanz eines Obertyps erwartet wird. Insbesondere benötigt man das Ersetzbarkeitsprinzip für

- den Aufruf einer Routine 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

Wann ist das Ersetzbarkeitsprinzip erfüllt? Diese Frage wird in der Fachliteratur intensiv behandelt. Wir wollen die Frage hier nur so weit beantworten, als es in der Praxis relevant ist. Wir gehen davon aus, dass Typen Schnittstellen von Objekten sind, die in Klassen beziehungsweise Interfaces – vereinfachte Klassen ohne Implementierung, siehe Abschnitt 2.4 – spezifiziert wurden. Es gibt in Java auch Typen wie `int`, die keiner Klasse entsprechen. Aber für solche Typen gibt es in Java keine Untertypen; deshalb werden wir sie hier nicht näher betrachten.

Eine Voraussetzung für das Bestehen einer Untertypbeziehung in Java ist, dass auf den entsprechenden Klassen oder Interfaces eine Vererbungsbeziehung besteht. Die dem Untertyp entsprechende Klasse (oder das Interface) muss also durch `extends` oder `implements` von der dem Obertyp entsprechenden Klasse (oder dem Interface) direkt oder indirekt abgeleitet sein. Ähnliche Voraussetzungen gibt es auch in anderen Programmiersprachen wie C++ oder Eiffel. Solche Voraussetzungen sind praktisch sinnvoll, wie wir später sehen werden. Man kann Untertypbeziehungen aber auch ohne eine solche Voraussetzung definieren. Objective-C und Smalltalk sind Beispiele für Sprachen, in denen man Vererbung nicht als Voraussetzung für das Bestehen einer Untertypbeziehung ansieht.

Nun wollen wir einige generelle Bedingungen für das Bestehen einer Untertypbeziehung betrachten. Generell gilt, dass jeder Typ Untertyp von sich selbst ist. Nehmen wir an, U und T bezeichnen zwei Typen beziehungsweise Schnittstellen. Dann ist U ein Untertyp von T, wenn folgende Bedingungen erfüllt sind:

- Für jede Konstante (beziehungsweise Variable, auf die nach der Initialisierung nur lesend zugegriffen werden kann) 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.

Auf eine Konstante kann 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 in Wirklichkeit vom Typ U ist, wenn also eine Instanz von U verwendet wird, wo eine Instanz 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 gleich sind.

  Auf eine Variable kann lesend und schreibend zugegriffen werden. Ein lesender Zugriff entspricht der oben beschriebenen Situation bei Konstanten; der deklarierte Typ B der Variable in U muss also ein Untertyp des deklarierten Typs A der Variable in T sein. Wenn man eine Variable in einem Objekt vom Typ T schreibt, erwartet man sich, dass man jede Instanz vom Typ A der Variablen zuweisen darf. Diese Erwartung soll auch erfüllt sein, wenn das Objekt in Wirklichkeit 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. Da Untertypbeziehungen antisymmetrisch sind, müssen A und B gleich sein.

- Für jede Methode in T gibt es eine entsprechende Methode in U, wobei der deklarierte Ergebnistyp der Methode in U ein Untertyp des Ergebnistyps der Methode in T ist, die Anzahl der formalen Parameter der beiden Methoden gleich ist und der deklarierte Typ jeden formalen Parameters in U ein Obertyp des deklarierten Typs des entsprechenden formalen Parameters in T ist.

  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 tatsächlich 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 tatsächlich 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 nur für Sprachen wie Java, in denen Argumente nur vom Aufrufer an die aufgerufene Methode übergeben werden (Eingangsparameter). In Sprachen wie beispielsweise C++ und Ada können über Parameter auch Objekte von der aufgerufenen Methode an den Aufrufer zurück gegeben werden (Ausgangsparameter). Für die Typen solcher Parameter gelten dieselben Bedingungen wie für Ergebnistypen. In solchen 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 gleich sein.

Wenn man eine Schnittstelle aus einer anderen ableitet, also einen Untertyp bildet, kann man nicht nur neue Elemente hinzufügen, sondern auch deklarierte Typen der einzelnen Elemente ändern; die deklarierten Typen der Elemente können daher teilweise variieren. Je nach dem, wie diese Typen variieren können, spricht man von Kovarianz, Kontravarianz und Invarianz:

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 (auch 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 gleich dem 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.

Betrachten wir ein Beispiel in einer Java-ähnlichen Sprache:

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

Entsprechend den oben angeführten Bedingungen erfüllt `meth` in Klasse `B` alle Voraussetzungen, um an Stelle von `meth` in `A` verwendet zu werden: Der Ergebnistyp ist kovariant verändert, der Parametertyp kontravariant. In einer Java-ähnlichen Sprache kann `B` daher tatsächlich ein Untertyp von `A` sein, wobei `meth` in `B` überschrieben ist. Java ist aber stärker eingeschränkt als es durch diese Bedingungen notwendig ist. Daher überschreibt die Methode in `B` jene in `A` nicht, wie wir weiter unten sehen werden; stattdessen wird `meth` geerbt und überladen, so dass es in Instanzen von `B` beide Methoden nebeneinander gibt.

Obige Bedingungen für Untertypbeziehungen sind in gewisser Weise vollständig: Man kann keine weglassen oder aufweichen, ohne mit dem Ersetzbarkeitsprinzip in Konflikt zu kommen. Die meisten dieser Bedingungen stellen keine praktische Einschränkung dar. ProgrammiererInnen kommen kaum in Versuchung sie zu brechen. Nur eine Bedingung, nämlich die geforderte Kontravarianz von formalen Parametertypen, möchte man manchmal gerne umgehen. Sehen wir uns dazu ein Beispiel an:

```
class Point2D {
    public int x, y;
    public boolean equal (Point2D p) {
        return x == p.x && y == p.y;
    }
}
class Point3D extends Point2D {
    public int z;
    public boolean equal (Point3D p) {
        return x == p.x && y == p.y && z == p.z;
    }
}
```

In diesem Programmstück erfüllt `equal` nicht die Kriterien für Untertypbeziehungen, da der Parametertyp kovariant und nicht, wie gefordert,

kontravariant ist. In Java kann die Methode `equal` in `Point3D` jene in `Point2D` daher nicht überschreiben. Eine Methode, bei der ein formaler Parametertyp stets gleich der Klasse ist, in der die Methode definiert ist, heißt *binäre Methode*. Der Begriff *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 expliziten Parameters. Binäre Methoden sind über den einfachen enthaltenden Polymorphismus, wie wir ihn hier verwenden, prinzipiell nicht realisierbar. In Java wird in diesem Beispiel `equal` überladen, nicht überschrieben; die Klasse `Point3D` beschreibt zwei Methoden namens `equal`.

Untertypbeziehungen sind in Java wesentlich stärker eingeschränkt, als es durch obige Bedingungen notwendig wäre. In Java sind alle Typen invariant. Der Grund dafür liegt vermutlich darin, dass Konstanten nur als spezielle Variablen angesehen werden, die ohnehin invariant sein müssen, und darin, dass Methoden überladen sein können. Da überladene Methoden durch die Typen der formalen Parameter unterschieden werden, wäre es für ProgrammiererInnen schwierig, überladene Methoden von Methoden mit kontravariant veränderten Typen auseinanderzuhalten. Beispiele für Überladen, wo man Überschreiben erwarten könnte, haben wir bereits gesehen. Warum Ergebnistypen von Methoden invariant sein müssen, ist, abgesehen von der Einfachheit der Regel „Alles ist invariant“, nicht ganz nachvollziehbar. Man spricht bereits davon, dass künftige Java-Versionen in diesem Fall vielleicht kovariante Typen erlauben werden.

Vererbung in Java ist so eingeschränkt, dass diese Bedingungen für Untertypbeziehungen erfüllt sind. Die Bedingungen werden bei der Übersetzung eines Java-Programms überprüft.

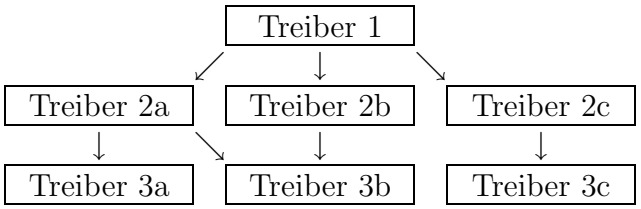
2.1.2 Untertypen und Codewiederverwendung

Die wichtigste Entscheidungsgrundlage für den Einsatz des enthaltenden Polymorphismus ist sicherlich die erzielbare Wiederverwendung. Der richtige Einsatz von enthaltendem Polymorphismus eröffnet durch das Ersetzbarkeitsprinzip einige Möglichkeiten, die auf den ersten Blick aber 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. Weiters schreiben wir

einige Anwendungen, die die Treiberklasse verwenden. Daneben werden vielleicht auch von anderen EntwicklerInnen, 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 dazufügen. Nach obiger Definition ist der Typ der neuen Klasse ein Untertyp des alten Typs. Neue Treiber – das sind Instanzen 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, die wir nicht einmal kennen. Das Beispiel können wir beliebig fortsetzen, indem wir immer wieder neue Varianten von Treibern schreiben und neue Anwendungen entwickeln oder bestehende Anwendungen anpassen, die 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:

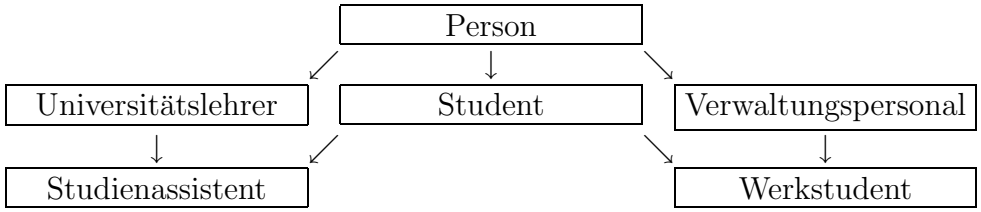


An diesem Bild fällt die Version 3b auf: Sie vereinigt zwei inkompatible Vorgängerversionen, nämlich 2a und 2b. Ein Untertyp kann mehrere Obertypen haben, die zueinander in keiner Untertypbeziehung stehen.

Die Wiederverwendung zwischen verschiedenen Versionen funktioniert nur dann gut, wenn die Schnittstellen zwischen den Versionen stabil bleiben. Das heißt, eine neue Version darf die Schnittstellen nicht beliebig ändern, sondern nur so, dass die in Unterabschnitt 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.

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 StudienassistentInnen sowohl UniversitätslehrerInnen als auch StudentInnen, und WerkstudentInnen an der Universität gehören zum Verwaltungspersonal und sind StudentInnen. 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 unterscheiden, sondern funktioniert für jede Instanz des Typs `Person`, auch wenn es tatsächlich eine Instanz des Typs `Studienassistent` ist. Diese Methode wird also für alle Arten von Personen (wieder)verwendet. Ebenso funktioniert eine Methode zum Ausstellen eines Zeugnisses für alle Instanzen von `Student`, auch wenn es StudienassistentInnen oder WerkstudentInnen sind.

Solche Klassenstrukturen können helfen, Auswirkungen nötiger Programmänderungen möglichst 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 Instanzen der Typen `Student`, `Studienassistent` oder `Werkstudent` zugreifen. Aber Programmteile, die auf Instanzen von `Person` zugreifen, sollten von der Änderung auch dann nicht betroffen sein, wenn die Instanzen tatsächlich vom Typ `Student` sind. Diese Programmteile verwenden keine geänderten Eigenschaften der Instanzen.

Falls bei der nötigen Programmänderung die Schnittstelle der Klasse unverändert bleibt, 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 möglichst stabil sind. Eine Programmänderung führt möglicherweise zu einer Reihe weiterer nötiger Änderungen, wenn dabei die Schnittstelle geändert wird. Die Anzahl wahrscheinlich nötiger Änderungen hängt dabei 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 in der Struktur. Eine Schlußfolgerung aus diesen Überlegungen ist, dass man, wenn möglich, nur von solchen Klassen erben soll, deren Schnittstellen bereits – oft nach mehreren Refaktorisierungsschritten – recht stabil sind.

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.

2.1.3 Dynamisches Binden

Bei Verwendung von enthaltendem Polymorphismus kann der dynamische Typ einer Variablen oder eines Parameters ein Untertyp des statischen beziehungsweise deklarierten Typs sein. Eine Variable vom Typ `Person` kann zum Beispiel eine Instanz von `Werkstudent` enthalten. Oft ist zur Übersetzungszeit des Programms 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 Variable steht, erst zur Laufzeit an die auszuführende Methode gebunden werden. In Java wird, unabhängig vom statischen Typ, immer die Methode ausgeführt, die in der Klasse des Objekts definiert ist. Die Schnittstelle dieser Klasse entspricht dem spezifischsten dynamischen Typ der Variablen.

Wir demonstrieren die Funktionsweise dynamischen Bindens an folgendem kleinen Beispiel:

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

Wenn wir diese Klassen compilieren und `DynamicBindingTest` ausführen, erhalten wir am Bildschirm folgende Ausgabe:

```
foo1A
foo2A
foo1B
foo2B
```

Die ersten Zeilen sind einfach erklärbar: Nach dem Programmaufruf wird die Methode `main` ausgeführt, die `test` mit einer neuen Instanz 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 statischen und dynamischen Typ. Es werden daher `foo1` und `foo2` in `A` ausgeführt.

Der zweite Aufruf von `test` übergibt eine Instanz von `B` als Argument. Dabei ist `A` der deklarierte Typ von `x`, aber der dynamische Typ ist `B`. Wegen dynamischem Binden werden diesmal `foo1` und `foo2` in `B` ausgeführt. Die dritte Zeile der Ausgabe enthält das Ergebnis des Aufrufs von `foo1` in einer Instanz 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, das ist eine Instanz von `B`, auf. 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` in `A` aufgerufen wird, da `foo2` ja nicht explizit in `B` steht, und daher `fooX` in `A` aufruft. Tatsächlich wird aber `fooX` in `B` aufgerufen, da `B` der spezifischste Typ der Umgebung ist.

Dynamisches Binden ist mit `switch`-Anweisungen und geschachtelten `if-then-else`-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 anredeArt, String name ) {
  switch(anredeArt) {
    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:

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

Durch dynamisches Binden wird automatisch die gewünschte Version von `gibAnredeAus()` aufgerufen. Statt einer `switch`-Anweisung wird in der objektorientierten Variante also dynamisches Binden verwendet. Ein Vorteil der objektorientierten Variante 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 Anredeart mit dem auszugebenden Namen verknüpft, wodurch man im Programm stets nur eine Instanz von `Adressat` anstatt einer ganzen Zahl und einem String verwalten muss. Ein anderer Vorteil der objektorientierten Variante ist auch sehr wichtig: Wenn sich herausstellt, dass neben „Frau“ und „Herr“ noch weitere Arten von Anreden, etwa „Repräsentanten der Firma“, benötigt werden, kann man diese leicht durch Hinzufügen einer weiteren Klasse einführen. Es sind keine weiteren Änderungen nötig. Insbesondere bleiben die Methodenaufrufe unverändert.

Auf den ersten Blick mag es scheinen, als ob die konventionelle Variante mit `switch`-Anweisung 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. Die objektorientierte Lösung hat dieses Problem nicht, da alles auf die Klasse `Adressat` und dessen Unterklassen konzentriert ist. Es bleibt auch dann alles konzentriert, wenn zu `gibAnredeAus()` weitere Methoden dazukommen.

2.2 Ersetzbarkeit und Objektverhalten

In Abschnitt 2.1 haben wir einige Bedingungen kennen gelernt, die erfüllt sein müssen, damit ein Typ Untertyp eines anderen Typs sein kann. Die Erfüllung dieser Bedingungen wird vom Compiler überprüft. Die Bedingungen sind aber nicht in jedem Fall ausreichend, um die uneingeschränkte Ersetzbarkeit einer Instanz eines Obertyps durch eine Instanz eines Untertyps zu garantieren. Dazu müssen weitere Bedingungen hinsichtlich des Objektverhaltens erfüllt sein, die von einem Compiler nicht über-

prüft werden können. SoftwareentwicklerInnen müssen ohne Compilerunterstützung sicherstellen, dass diese Bedingungen erfüllt sind.

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 dessen, was das Objekt tut, sein soll. Einerseits beschreibt die Schnittstelle eines Objekts das Objekt nur sehr unvollständig; eine genauere Beschreibung wäre wünschenswert. Andererseits enthält die Implementierung des Objekts, 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 Objektschnittstelle 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. Der Server muss diesen Vertrag ebenso einhalten wie jeder Client. Generell sieht der Vertrag folgendermaßen aus:

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

Im einzelnen regelt der Vertrag für jeden vom Server angebotenen Dienst, also für jede aufrufbare Methode, folgende Details:

Vorbedingungen (preconditions): Das sind Bedingungen, für deren Erfüllung vor Ausführung der Methode der Client verantwortlich ist. Vorbedingungen beschreiben hauptsächlich, welche Eigenschaften die Argumente, mit denen die Methode aufgerufen wird, erfüllen müssen. Zum Beispiel muss ein bestimmtes Argument ein Array von aufsteigend sortierten ganzen Zahlen im Wertebereich von 0 bis 99 sein. Vorbedingungen können aber auch den Zustand des Servers oder bereits vergangene Methodenaufrufe einbeziehen, soweit diese Information dem Client zugänglich ist. Zum Beispiel ist eine Methode nur aufrufbar, wenn eine Variable des Servers einen Wert größer 0 hat und unmittelbar zuvor eine bestimmte andere Methode des Servers ausgeführt wurde.

Nachbedingungen (postconditions): Für die Erfüllung dieser Bedingungen 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. Die Methode liefert einen Boolschen Wert zurück, der besagt, ob ein als Argument übergebenes Objekt vor dem Aufruf bereits in der Menge enthalten war; am Ende muss dieses Objekt auf jeden Fall in der Menge sein. Diese Beschreibung der Methode kann man als Nachbedingung auffassen.

Invarianten (invariants): Für die Erfüllung dieser Bedingungen sowohl vor als auch nach der Ausführung jeder Methode ist grundsätzlich der Server zuständig. Zum Beispiel muss das Guthaben auf einem Sparbuch immer durch eine positive Zahl dargestellt werden, egal welche Operationen auf dem Objekt, das das Sparbuch darstellt, durchgeführt werden. Die Gültigkeit einer Invariante kann auch von Bedingungen abhängen. Zum Beispiel bleibt ein Objekt immer in einer Menge enthalten, sobald es in sie eingefügt wurde. Jede Invariante impliziert eine entsprechende Nachbedingung auf jeder Methode des Servers beziehungsweise der Klasse.

Vorbedingungen, Nachbedingungen und Invarianten sind verschiedene Arten von *Zusicherungen*.

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 an den richtigen Stellen in den Programmcode schreiben kann. Diese Zusicherungen werden zur Laufzeit überprüft. Sprachen wie Java unterstützen überhaupt keine Zusicherungen – abgesehen von trivialen `assert`-Anweisungen in neueren Versionen, die sich aber kaum zur Beschreibung von Verträgen eignen. Sogar in Eiffel benötigt man nicht selten aufwendigere Zusicherungen, die in der Sprache selbst kaum ausdrückbar sind. In diesen Fällen kann und soll man Zusicherungen als Kommentare in den Programmcode schreiben und händisch überprüfen.

(für Interessierte)

Anmerkungen wie diese geben zusätzliche Informationen für interessierte Leser. Ihr Inhalt gehört nicht zum Prüfungsstoff.

Ein Beispiel in Eiffel soll veranschaulichen, wie Zusicherungen in Programmiersprachen integrierbar sind. Zu jeder Methode kann man vor der eigentlichen Implementierung (in einer `do`-Klausel) eine Vorbedingung (`require`-Klausel) und nach der Implementierung eine Nachbedingung (`ensure`-Klausel) angeben. Invarianten stehen am Ende der Klasse. In jeder Zusicherung steht eine Liste Boolscher Ausdrücke, die durch Strichpunkt getrennt sind. Der Strichpunkt steht für eine Konjunktion (Und-Verknüpfung). Die Zusicherungen 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. Zum Beispiel bezeichnet `old guthaben` den Wert der Variable `guthaben` zum Zeitpunkt des Methodenaufrufs.

```
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
```

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:

```
class Konto {
    public int guthaben;
    public int ueberziehungsrahmen;
    // einzahlen addiert summe zu guthaben; summe >= 0
    public void einzahlen (int summe) {
        guthaben = guthaben + summe;
    }
    // abheben zieht summe von guthaben ab;
    // summe >= 0; guthaben+ueberziehungsrahmen >= summe
    public void abheben (int summe) {
        guthaben = guthaben - summe;
    }
    // guthaben >= -ueberziehungsrahmen
}
```

Wir betrachten im Folgenden Zusicherungen, unabhängig davon, ob sie durch eigene Sprachkonstrukte oder in Kommentaren beschrieben sind, als zum Typ eines Objekts ghörend. Ein Typ besteht demnach aus

- dem Namen einer Klasse oder eines Interfaces,
- der Schnittstelle der Instanzen dieser Klasse
- und Zusicherungen auf den Methoden und Konstruktoren.

Der Name sollte eine kurze Beschreibung des Zwecks der Klasse geben. Die Schnittstelle enthält alle vom Compiler überprüfbaren Bestandteile des Vertrags zwischen Clients und Server. Zusicherungen enthalten schließlich alle Vertragsbestandteile, die nicht vom Compiler überprüft werden.

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 Unterabschnitt 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 geändert wird.

ProgrammiererInnen können die Genauigkeit der Zusicherungen selbst bestimmen. Dabei sind Auswirkungen der Zusicherungen zu beachten: Clients dürfen sich nur auf das verlassen, was in der Schnittstelle und in den Zusicherungen vom Server zugesagt wird, und der Server auf das, was von den Clients zugesagt wird. 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 Progammänderungen 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 Objektschnittstelle steht.

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 Objektschnittstelle 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 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.

2.2.2 Untertypen und Verhalten

Zusicherungen, die zu Typen gehören, müssen auch bei der Verwendung von enthaltendem Polymorphismus 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:

- 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.
- 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 die Nachbedingung in T automatisch erfüllt sein, wenn sie in U erfüllt ist. Wenn Nachbedingungen in U aus T übernommen werden, können sie mittels Und-Verknüpfungen stärker werden. Lautet die Nachbedingung in T zum Beispiel „`result` > 0“, kann sie in U auch „`result` > 0 und `result` > 2“, also „`result` > 2“ sein.
- 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 eine Instanz von U statt einer von T verwendet wird. Der Server kennt seinen eigenen spezifischsten Typ, weshalb das Ersetzbarkeitsprinzip aus der Sicht des Servers nicht erfüllt 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 zwischen Nachbedingungen und Invarianten ist notwendig, da Invarianten im Wesentlichen entsprechende Nachbedingungen auf allen Methoden des Typs implizieren.

Diese Erklärung geht davon aus, dass Instanzvariablen nicht durch andere Objekte verändert werden. Ist dies doch der Fall, so müssen Invarianten, die sich auf global änderbare Variablen beziehen, in U und T überein stimmen. 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 Instanzvariablen nie durch andere Objekte verändert werden.

Im Prinzip lassen sich obige Bedingungen auch formal überprüfen. In Programmiersprachen wie Eiffel, in denen Zusicherungen formal definiert sind, wird das tatsächlich gemacht. 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 die ProgrammiererInnen alle nötigen Überprüfungen per Hand durchführen. Im Einzelnen muss sichergestellt werden, dass

- obige Bedingungen für Untertypbeziehungen eingehalten werden,
- die Implementierungen der Server die Nachbedingungen und Invarianten erfüllen und nichts voraussetzen, was nicht durch Vorbedingungen oder Invarianten festgelegt ist
- und Clients die Vorbedingungen der Aufrufe erfüllen und nichts voraussetzen, was nicht in Nachbedingungen und Invarianten vom Server zugesichert wird.

Es kann sehr aufwendig sein, alle solchen Überprüfungen vorzunehmen. Einfacher geht es, wenn ProgrammiererInnen während der Codeerstellung und bei Programmänderungen stets an die einzuhaltenden Bedingungen denken, die Überprüfungen also nebenbei erfolgen. Wichtig ist dabei, darauf zu achten, dass die Zusicherungen unmissverständlich formuliert sind.

Betrachten wir ein Beispiel für einen Typ beziehungsweise eine Klasse mit Zusicherungen in Form von Kommentaren:

```
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“, also „genau dann wenn“) in eine Instanz 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 Instanzen 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 beschreiben, was sich ein Client vom Aufruf einer Methode erwartet, sind Nachbedingungen oft tatsächlich nur Beschreibungen von Methoden.

Folgende Klasse unterscheidet sich von `Set` nur durch eine zusätzliche Invariante:

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

Die Invariante besagt, dass eine Zahl, die einmal in der Menge war, stets in der Menge bleibt. Offensichtlich ist `SetWithoutDelete` ein Untertyp von

`Set`, da nur eine Invariante dazugefügt wurde, die Invarianten insgesamt also strenger wurden. Wie kann ein Client eine solche Invariante nutzen? Betrachten wir dazu eine kuze Codesequenz für einen Client:

```
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 eine Instanz von `SetWithoutDelete` anstatt einer 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` für Clients darf man sich nicht dazu verleiten lassen, generell starke Nachbedingungen und Invarianten zu verwenden. Solche umfangreichen Zusicherungen können die Wartung erschweren (siehe Unterabschnitt 2.2.1). Zum Beispiel können wir `Set` problemlos um eine Methode `delete` (zum Löschen einer Zahl aus der Menge) erweitern:

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

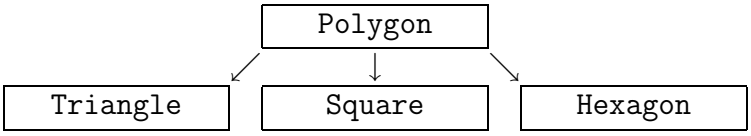
Aber `SetWithoutDelete` können wir nicht um eine solche Methode erweitern. Zwar wäre eine derart erweiterte Klasse mit obigen Bedingun-

gen für Zusicherungen bei Untertypbeziehungen vereinbar, aber die Nachbedingung von `delete` steht in Konflikt zur Invariante. Es wäre also unmöglich, `delete` so zu implementieren, dass sowohl die Nachbedingung als auch die Invariante erfüllt ist. 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 sie wirklich braucht. Andernfalls verbaut man sich Wiederverwendungsmöglichkeiten.

2.2.3 Abstrakte Klassen

Klassen, die wir bis jetzt betrachtet haben, dienen der Beschreibung der Struktur ihrer Instanzen, der Erzeugung und Initialisierung neuer Instanzen und der Festlegung des spezifischsten Typs der Instanzen. Im Zusammenhang mit enthaltendem Polymorphismus benötigt man aber oft nur eine der Aufgaben, nämlich die Festlegung des Typs. Das ist dann der Fall, wenn im Programm keine Instanzen der Klasse selbst erzeugt werden sollen, sondern nur Instanzen von Unterklassen. Dafür unterstützen viele objektorientierte Sprachen sogenannte *abstrakte Klassen*, von denen keine Instanzen erzeugt werden können. In Java gibt es daneben auch Interfaces, die einen ähnlichen Zweck erfüllen (siehe Abschnitt 2.4).

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 möglichen 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 Instanzen von `Triangle`, `Square` und `Hexagon` vorkommen. Es werden keine Instanzen der Klasse `Polygon` selbst benötigt, sondern nur Instanzen der Unterklassen. `Polygon` ist ein typischer Fall einer abstrakten Klasse.

In Java sieht die abstrakte Klasse etwa so aus:

```
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 angeben, da die Methoden sowieso nicht ausgeführt werden; es gibt ja keine Instanzen. Nicht-abstrakte Unterklassen – das sind *konkrete Klassen* – müssen Implementierungen für abstrakte Methoden bereitstellen, diese also überschreiben. Abstrakte Unterklassen brauchen abstrakte Methoden natürlich nicht überschreiben.

Die konkrete Klasse `Triangle` könnte so aussehen:

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

Ähnlich wie `Triangle` müssen auch die Klassen `Square` und `Hexagon` 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 vor allem von stabilen Klassen erben. Außerdem soll man möglichst stabile Typen für formale Parameter und Variablen verwenden. Da es oft leichter ist, abstrakte Klassen ohne Implementierungen stabil zu halten, ist man gut beraten, hauptsächlich solche Klassen für Parameter- und Variablentypen 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 entsprechend den Wünschen der ProgrammiererInnen durch Erweitern und Überschreiben abgeändert wird. Die resultierende Klasse ist die Unterklasse. Wenn man nur Vererbung betrachtet und Einschränkungen durch enthaltenden Polymorphismus ignoriert, haben ProgrammiererInnen 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 lernen und dann die Bedeutungen dieser Beziehungen für die Codewiederverwendung untersuchen.

2.3.1 Reale Welt versus Vererbung versus Ersetzbarkeit

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

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

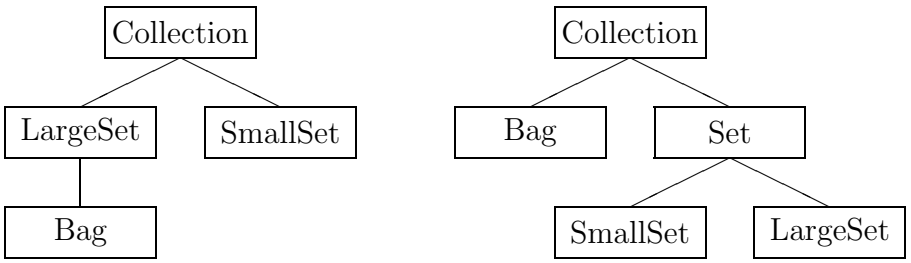
Vererbungsbeziehung: Das ist eine Beziehung zwischen Klassen, bei der eine Klasse durch Abänderung einer anderen Klasse entsteht. Es ist nicht nötig, aber wünschenswert, dass dabei Code aus der Oberklasse in der Unterklasse direkt wiederverwendet wird. Für eine reine Vererbungsbeziehung ist das Ersetzbarkeitsprinzip irrelevant.

Reale-Welt-Beziehung: In der Analysephase und zu Beginn der Entwurfsphase haben sich oft schon abstrakte Einheiten herauskristallisiert, die in späteren Phasen zu Klassen weiterentwickelt werden. Auch Beziehungen zwischen diesen Einheiten existieren bereits sehr früh. Sie spiegeln angenommene *ist-ein-Beziehungen* („is a“) in der realen Welt wider. Zum Beispiel haben wir die Beziehung „ein Studierender ist eine Person“, wobei „Studierender“ und „Person“ abstrakte Einheiten sind, die später voraussichtlich zu Klassen weiterentwickelt werden. Durch die Simulation der realen Welt sind solche Beziehungen bereits sehr früh intuitiv klar, obwohl die genauen Eigenschaften der Einheiten noch gar nicht feststehen. Normalerweise entwickeln sich diese Beziehungen während des Entwurfs zu Untertyp- und Vererbungsbeziehungen zwischen Klassen weiter. Es kann sich aber auch

herausstellen, dass Details der Klassen dem Ersetzbarkeitsprinzip widersprechen und Vererbung nicht sinnvoll einsetzbar ist. In solchen Fällen wird es zu Refaktorisierungen kommen, die in dieser frühen Entwicklungsphase noch recht 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 Untertyp- und Vererbungsbeziehungen. Genaugenommen setzen Untertypbeziehungen, zumindest in Java und ähnlichen objektorientierten Sprachen, Vererbungsbeziehungen voraus und sind derart eingeschränkt, dass die vom Compiler überprüfbaren Bedingungen für Untertypbeziehungen stets erfüllt sind. Das heißt, das wesentliche Unterscheidungskriterium ist das, ob die Zusicherungen zwischen Unter- und Oberklasse kompatibel sind. Diese Unterscheidung können nur die SoftwareentwicklerInnen treffen. In allen anderen Kriterien sind in Java reine Vererbungsbeziehungen von Untertypbeziehungen nicht unterscheidbar.

Dennoch kann man leicht unterscheiden, ob SoftwareentwicklerInnen nur reine Vererbungs- oder Untertypbeziehungen anstreben. Betrachten wir dazu ein Beispiel:



reine Vererbungsbeziehung Untertypbeziehung

Es ist das Ziel der reinen Vererbung, so viele Teile der Oberklasse wie möglich direkt in der Unterklasse wiederzuverwenden. 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 beziehungsweise 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 bzw. Löschen von Elementen einander ausschließende Bedeutungen haben, obwohl `Set` und `Bag` dieselbe Schnittstelle haben können.

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 der Sprache 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. Natürlich möchten wir einen möglichst hohen Grad an Codewiederverwendung erzielen. Ist es daher günstig, Untertypbeziehungen unberücksichtigt zu lassen? Diese Frage muss man ganz klar mit Nein beantworten. Durch die Nichtbeachtung des Ersetzbarkeitsprinzips – das heißt, Untertypbeziehungen sind nicht gegeben – ist es nicht mehr möglich, eine Instanz eines Untertyps zu verwenden, wo eine Instanz eines Obertyps erwartet wird. Wenn man trotzdem eine Instanz einer Unterklasse statt einer Oberklasse verwendet, kann ein Programmfehler auftreten. Verzichtet man auf die 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 durch die reine Vererbung bessere direkte Codewiederverwendung in kleinem Umfang, tauscht diese aber gegen viele Möglichkeiten für die indirekte Codewiederverwendung in großem Umfang durch die Ersetzbarkeit.

Der allgemeine Ratschlag ist daher ganz klar: Ein wichtiges Ziel ist die Entwicklung geeigneter Untertypbeziehungen. Vererbungsbeziehungen sind nur Mittel zum Zweck; das heißt, sie sollen sich den Untertypbeziehungen unterordnen. Im Allgemeinen soll es im Programm keine Vererbungsbeziehung geben, die nicht auch eine Untertypbeziehung ist, bei der also alle Zusicherungen kompatibel sind.

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 ProgrammiererInnen 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.

Es kommt auch 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:

```
class A {
    public void foo() { ... }
}
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` verwendbar. Diese Art des Zugriffs auf Oberklassen funktioniert allerdings nicht über mehrere Vererbungsebenen hinweg.

In komplizierten Situationen ist geschickte Faktorisierung notwendig:

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

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

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

Nun braucht man nur mehr einen Teil der ursprünglichen Methode `foo` überschreiben. Eine andere Möglichkeit besteht darin, die Unterschiede durch einen zusätzlichen Parameter einer Hilfsfunktion zu beschreiben und die eigentlichen Funktionen nur zum Setzen der Parameter zu verwenden:

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

```
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` 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 die 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 eine Instanz einer Unterklasse verwendet wird, wo eine Instanz 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. Da es in Java aber keine solchen Möglichkeiten gibt, wollen wir nicht näher darauf eingehen.

2.4 Klassen und Vererbung in Java

In den vorhergehenden Abschnitten haben wir einige wichtige Konzepte in objektorientierten Sprachen betrachtet. In diesem Abschnitt geben wir einen Überblick über entsprechende Konzepte in Java und deren Umsetzung in die konkrete Programmiersprache.

2.4.1 Klassen in Java

Den Aufbau einer Klasse in Java, eingeleitet durch das Schlüsselwort `class`, haben wir bereits in einigen Beispielen gesehen:

```
class Klassenname { ... }
```

Namen von Klassen werden per Konvention mit großen Anfangsbuchstaben geschrieben, Namen von Konstanten oft nur mit Großbuchstaben und alle anderen Namen mit kleinen Anfangsbuchstaben. In Java wird streng zwischen Groß- und Kleinschreibung unterschieden. Die Namen `A` und `a`

sind daher verschieden. Der Inhalt der Klasse steht innerhalb geschwungener Klammern.

Eine Klasse kann mehrere explizit definierte Konstruktoren enthalten:

```
class Circle {
    int r;
    Circle(int r) { this.r = r; }       // 1
    Circle(Circle c) { this.r = c.r; }  // 2
    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 einer neuen Instanz werden dem Konstruktor Argumente übergeben. Anhand der Anzahl und der 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 das Schlüsselwort `this` wie den Namen einer Variable verwendet. Tatsächlich bezeichnet `this` immer die aktuelle Instanz der Klasse. In Konstruktoren ist das die Instanz, die gerade erzeugt wurde. Im ersten Konstruktor benötigen wir `this`, um die Variable `r` in der neuen Instanz, das ist `this.r`, vom formalen Parameter `r` des Konstruktors zu unterscheiden. Wie in diesem Beispiel können formale Parameter (oder lokale Variablen) Variablen in der aktuellen Instanz der Klasse verdecken, die denselben Namen haben. Über `this` kann man dennoch auf die Instanzvariablen 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 der aktuellen Instanz der Klasse als Argument. Zum Beispiel liefert `new Circle(this)` innerhalb der Klasse `Circle` eine Kopie der aktuellen Instanz.

Falls in einer Klasse kein Konstruktor explizit definiert ist, enthält die Klasse automatisch einen Defaultkonstruktor:

```
public Klassenname() { super(); }
```

Wie wir noch sehen werden, ruft `super()` den Konstruktor der Oberklasse auf. Ist keine Oberklasse explizit angegeben, wird implizit `Object` als vordefinierte Oberklasse verwendet.

Instanzvariablen, das sind Variablen, die zu den Instanzen einer Klasse gehören, werden in der Klasse – so wie `r` in `Circle` – einfach durch Hinschreiben des Typs, gefolgt vom Namen der Variablen und einem Strichpunkt, deklariert. Man kann nach dem Typ auch mehrere, durch Komma getrennte Namen von Variablen, die alle denselben Typ haben, hinschreiben und/oder jede dieser Variablen initialisieren. Eine Variablendeklaration sieht beispielsweise so aus:

```
int x = 2, y = 1, z;
```

Diese Zeile deklariert drei Instanzvariablen und initialisiert `x` mit 2 und `y` mit 1. Wenn keine Initialisierung angegeben ist, wird eine Defaultinitialisierung vorgenommen. Für Variablen, die Zahlen enthalten, erfolgt die Defaultinitialisierung mit 0, für Variablen, die Objekte enthalten können, mit `null`. Jede Instanz der Klasse enthält eigene Kopien dieser Variablen.

Eine spezielle Syntax gibt es in Java für Variablen, die Arrays enthalten. Beispielsweise sind die beiden folgenden Zeilen äquivalent:

```
int x[] = new int[32], y[], z;
int[] x = new int[32], y = null; int z = 0;
```

Arrays werden durch eckige Klammern gekennzeichnet. Da Arrays, abgesehen von der speziellen Syntax, gewöhnliche Objekte sind, werden Variablen, die Arrays enthalten, per Default mit `null` initialisiert.

Manchmal benötigt man Variablen, die nicht zu einer bestimmten Instanz einer Klasse gehören, sondern zur Klasse selbst. Solche *Klassenvariablen* kann man in Java einfach durch Voranstellen des Schlüsselwortes `static` deklarieren. Hier ist ein Beispiel für eine Klassenvariable:

```
static int maxRadius = 1023;
```

Solche Variablen stehen nicht in den Instanzen der Klasse, sondern in der Klasse selbst. Falls diese Deklaration in `Circle` steht, kann man über den Namen der Klasse darauf zugreifen – z. B. `Circle.maxRadius`. Auf Instanzvariablen kann man hingegen nur über eine Instanz zugreifen, wie z. B. in `c.r`, wobei `c` eine Variable vom Typ `Circle` ist. Ein Zugriff auf `Circle.r` ist nicht erlaubt.

Konstanten stellen einen Spezialfall von Klassenvariablen dar. Sie werden durch `static final` gekennzeichnet:

```
static final int MAX_SIZE = 1024;
```

Der Wert einer solchen Variable kann nach der Initialisierung nicht mehr geändert werden.

Eine Methode, die durch `static` gekennzeichnet wird, gehört ebenfalls zur Klasse und nicht zu einer Instanz der Klasse. Ein Beispiel dafür ist die Methode `main`:

```
static void main (String[] args) { ... }
```

Solche *statischen Methoden* werden über den Namen einer Klasse – z. B. `Circle.main(args)` – aufgerufen, nicht über den Namen der Instanz einer Klasse. Daher ist während der Ausführung der Methode keine aktuelle Instanz der Klasse bekannt und man darf nicht auf Instanzvariablen zugreifen. Auch `this` ist in statischen Methoden nicht verwendbar.

Konstruktoren machen es den ProgrammiererInnen leicht, komplexere Initialisierungen von Instanzvariablen vorzunehmen. *Static initializers* bieten eine derartige Möglichkeit auch für Klassenvariablen:

```
static { ... }
```

Ein *static initializer* besteht also nur aus dem Schlüsselwort `static` und einer beliebigen Sequenz von Anweisungen in geschwungenen Klammern. Diese Codesequenz wird vor der ersten Verwendung der Klasse ausgeführt.

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 zurück liefern. Wir werden nicht näher auf Destruktoren eingehen, da sie aufgrund einiger Eigenschaften von Java kaum sinnvoll einsetzbar sind: Java verwendet garbage collection; Objekte werden automatisch finalisiert und entfernt (recycled) sobald sie nicht mehr zugreifbar sind. Leider ist kaum abschätzbar, wann die Finalisierungen erfolgen. Unter Umständen bleiben nicht mehr zugreifbare Objekte bestehen, solange das Programm läuft.

Wie wir in Abschnitt 2.2 gesehen haben, können Klassen und Methoden in Java auch abstrakt sein. Eine Klasse ist abstrakt, wenn sie zumindest eine abstrakte Methode enthält. Solche Klassen, wie auch abstrakte Methoden, für die keine Implementierungen angegeben sind, müssen mit dem Schlüsselwort `abstract` gekennzeichnet sein.

In neueren Java-Versionen gibt es *geschachtelte Klassen* (nested classes), die innerhalb anderer Klassen definiert sind. Geschachtelte Klassen können überall definiert sein, wo Variablen deklariert werden dürfen. Innerhalb geschachtelter Klassen kann man auch private Variablen und Me-

thoden aus der Umgebung verwenden. Es gibt zwei Arten geschachtelter Klassen:

Statische geschachtelte Klassen: Diese werden mit dem Schlüsselwort `static` versehen. Hier ist ein Beispiel:

```
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. Instanzvariablen und Instanzmethoden der umschließenden Klasse sind nicht zugreifbar. Instanzen der geschachtelten Klasse im Beispiel können durch `new EnclosingClass.StaticNestedClass()` erzeugt werden.

Innere Klassen („inner classes“): Jede innere Klasse gehört zu einer Instanz der umschließenden Klasse. Das ist ein Beispiel dafür:

```
class EnclosingClass {
    ...
    class InnerClass { ... }
    ...
}
```

Instanzvariablen und Instanzmethoden aus `EnclosingClass` können in `InnerClass` uneingeschränkt verwendet werden. Innere Klassen dürfen jedoch keine statischen Methoden und keine statischen geschachtelten Klassen enthalten. Eine Instanz von `InnerClass` wird beispielsweise durch `new a.InnerClass()` erzeugt, wobei `a` eine Variable vom Typ `EnclosingClass` ist.

Abgesehen davon entsprechen geschachtelte Klassen den nicht geschachtelten Klassen. Sie können abstrakt sein und von anderen Klassen erben.

2.4.2 Vererbung in Java

Java unterstützt Einfachvererbung. Jede Klasse, außer der vordefinierten Klasse `Objekt`, hat genau einen direkten Vorgänger in der Vererbungshier-

archie. Ist kein direkter Vorgänger explizit angegeben, wird dafür `Object` verwendet. Den direkten Vorgänger kann man nach `extends` angeben:

```
class Unterklasse extends Oberklasse { ... }
```

In den geschwungenen Klammern stehen die Unterschiede zwischen Unterklasse und Oberklasse. Die Unterklasse kann die Oberklasse um neue Methoden und Variablen erweitern und Methoden aus der Oberklasse durch neue Methoden überschreiben, wie wir in Abschnitt 2.1 erfahren haben. Nicht überschriebene Methoden werden geerbt.

Beim Erzeugen einer neuen Instanz einer Unterklasse wird nicht nur ein Konstruktor der Unterklasse 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 irgend ein Konstruktor nicht mehr mit `this` beginnt. Sonst hätten wir eine Endlosschleife.

Es kann vorkommen, dass eine Variable in der Unterklasse denselben Namen hat wie eine Variable der Oberklasse. Die Variable der Unterklasse *verdeckt* in diesem Fall 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 deren 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 bezeichnet `super.v` die in der Oberklasse deklarierte Variable. Namen, die bereits weiter oben in der Klassenhierarchie verdeckt wurden, kann man aber nur durch eine explizite Typumwandlung ansprechen. Zum Beispiel ist `((Oberklasse)this).v` die Instanzvariable, die in `Oberklasse` mit dem Namen `v` angesprochen wird. Eine verdeckte Klassenvariable kann man leicht über den Klassennamen ansprechen, beispielsweise durch `Oberklasse.v`.

Überschriebene Instanzmethoden aus der Oberklasse, von der direkt abgeleitet wird, kann man ebenfalls über `super` ansprechen, wie wir in Abschnitt 2.3 gesehen haben. Mittels Typumwandlung kann man überschriebene Instanzmethoden aus Oberklassen aber niemals ansprechen:

Eine Typumwandlung ändert nur den deklarierten Typ. Aufgrund von dynamischem Binden hat der deklarierte Typ keinen Einfluss auf die Methode, die ausgeführt wird. Statische Methoden aus Oberklassen kann man einfach 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 gleichzeitig: Die deklarierten Typen der Argumente entscheiden, welche 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 einer Methode in einer Klassenbibliothek aus Sicherheitsgründen, z. B. zur Vermeidung der Umgehung einer Passwortabfrage, verbieten will.

Man kann auch ganze Klassen mit dem Schlüsselwort `final` versehen. Solche Klassen können keine Unterklassen haben. Dadurch ist es natürlich auch nicht möglich, die Methoden der Klasse zu überschreiben. Solche Klassen verwendet man in manchen objektorientierten Programmierstilen um klarzustellen, dass das Verhalten der Instanzen durch die Implementierung festgelegt ist. Clients können sich auf alle Implementierungsdetails verlassen, ohne auf mögliche Ersetzungen Rücksicht zu nehmen. Änderungen solcher Klassen sind aber mit einem vergleichsweise hohen Aufwand verbunden, da alle Clients überprüft und gegebenenfalls geändert werden müssen. Abstrakte Klassen dürfen natürlich nicht `final` sein.

2.4.3 Zugriffskontrolle in Java

In Java kann man die Sichtbarkeit und Zugreifbarkeit einzelner Methoden, Variablen, Klassen, etc. über einige vorgegebene Sichtbarkeitsstufen regeln. Dazu dienen die Schlüsselwörter `public`, `protected` und `private`. Um deren Bedeutung beschreiben zu können, müssen wir zuerst das Konzept der Pakete (oder Module) in Java betrachten, die in engem Zusammenhang mit Dateien und Ordnern stehen.

Jede compilierte Java-Klasse wird in einer eigenen Datei gespeichert. Der Name der Datei entspricht dem Namen der Klasse mit der Endung `.class`. Der Ordner, der diese Datei enthält, entspricht dem Paket, zu der die Klasse gehört. Der Name des Ordners ist der Paketname. Während der Softwareentwicklung steht der Quellcode einer Klasse meist im selben Ordner 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 müssen aber alle außer einer, nämlich der, deren Name gleich dem Dateinamen ist, als `private` definiert sein. Bei der Übersetzung wird in jedem Fall eine eigene Datei für jede Klasse erzeugt.

Generell können Namen im Quellcode den Namen des Paketes enthalten. 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. Wenn die Quelldatei oder compilierte Datei in einem anderen Ordner 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. 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 Regelung der Sichtbarkeit. Bei diesen Möglichkeiten sind Einheiten 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 die Default-Sichtbarkeit. Sie sind überall 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`	Default	`private`
sichtbar im selben Paket	ja	ja	ja	nein
sichtbar in anderem Paket	ja	nein	nein	nein
vererbbar im selben Paket	ja	ja	ja	nein
vererbbar 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-ProgrammiererInnen ist es gar nicht leicht, stets die richtigen Sichtbarkeitseigenschaften zu wählen. Hier sind einige Ratschläge, die diese Wahl erleichtern sollen:

- Alle Methoden, Konstanten und in seltenen Fällen auch Variablen, die man bei der Verwendung der Klasse oder von Instanzen der Klasse benötigt, sollen `public` sein.

- Man verwendet `private` am besten für alle Methoden und Variablen in einer Klasse, die nur innerhalb der Klasse verwendet werden sollen. Das betrifft meist Methoden, deren Funktionalität außerhalb der Klasse nicht verständlich ist, sowie Variablen, die diese Methoden benötigen.

- Wenn Variablen und Methoden für die Verwendung einer Klasse und ihrer Instanzen nicht nötig sind, diese Methoden und Variablen aber bei späteren Erweiterungen der Klasse hilfreich sein könnten, verwendet man am besten `protected`.

- In einem Paket sollen alle Klassen stehen, die eng zusammenarbeiten. In der Regel sind das nur wenige Klassen. Methoden und Variablen, die nur innerhalb eines Paketes gebraucht werden, sollen außerhalb des Paketes auch nicht sichtbar sein. Man verwendet dafür am besten die Default-Sichtbarkeit. Wenn aber außerhalb des Pakets Unterklassen von Klassen im Paket sinnvoll sind, sollen Methoden und Variablen, die in einer Unterklasse hilfreich sein könnten, mit `protected` definiert werden.

Es ist oft schwierig, geeignete Zusicherungen für Zugriffe auf Variablen anzugeben. Das ist ein wichtiger Grund für die Empfehlung, Variablen generell nicht `public` zu machen. Statt einer solchen Variablen kann man in der nach außen sichtbaren Schnittstelle eines Objekts immer auch eine Methode zum Abfragen des aktuellen Wertes (get method) und eine zum Setzen des Wertes (set method) schreiben. Obwohl solche Methoden oft weniger problematisch sind als Variablen, ist es noch besser, wenn solche Methoden 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.

2.4.4 Schnittstellen in Java

Java unterstützt nur Einfachvererbung auf Klassen. Das ist eine grobe Einschränkung, die sich in einigen Fällen sehr hinderlich auf die Festlegung von Untertypverhältnissen auswirken kann. Um diese Einschränkungen großteils zu beseitigen, gibt es in Java sogenannte Interfaces. Das sind im Wesentlichen eingeschränkte abstrakte Klassen, in denen alle Methoden abstrakt sind. Auf Interfaces wird Mehrfachvererbung unterstützt.

Interfaces unterscheiden sich von abstrakten Klassen wie folgt:

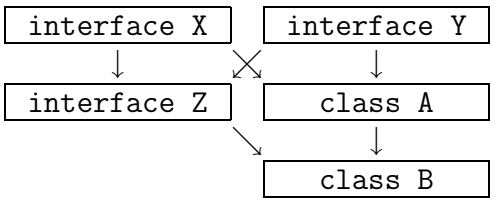
- Interfaces beginnen mit dem Schlüsselwort `interface` an Stelle von `abstract class`.
- Alle Methoden sind abstrakt, enthalten also keine Implementierungen. Da ohnehin klar ist, dass diese Methoden abstrakt sind, kann man das Schlüsselwort `abstract` weglassen.
- Variablen dürfen nicht deklariert werden, wohl aber Konstanten, also mit `static final` gekennzeichnete Variablen, die mit einem unveränderlichen Wert initialisiert sind.
- Alle Methoden sind `public` Instanzmethoden. Die Schlüsselwörter `static`, `final`, `protected` und `private` dürfen in Methodendeklarationen nicht vorkommen.
- Da alle Einheiten immer `public` sind, kann man das Schlüsselwort `public` im Interface weglassen. Es bleibt trotzdem alles `public`.
- Nach dem Schlüsselwort `extends` können mehrere, durch Komma getrennte Namen von Interfaces stehen. Das bedeutet, dass Interfaces von mehreren anderen Interfaces erben können (Mehrfachvererbung), aber nicht von Klassen.

Klassen können von Interfaces erben, wobei auch Mehrfachvererbung unterstützt wird, wie folgendes Beispiel zeigt.

```
interface X {
    static final double PI = 3.14159;
    double fooX();
}
interface Y {
    double fooY();
}
```

```
interface Z extends X, Y {
    double fooZ();
}
class A implements X, Y {
    double factor = 2.0;
    public double foo() { return PI; }
    public double fooX() { return factor * PI; }
    public double fooY() { return factor * fooX(); }
}
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 dem Schlüsselwort `implements` um anzudeuten, dass die in den Interfaces deklarierten Methoden in der Klasse zu implementieren sind. In einer Klassendefinition kann nach `extends` nur eine Klasse stehen. Dies kann man an der Klasse `B` sehen, die von `A` und `Z` erbt. Folgende Abbildung zeigt die Vererbungsbeziehungen zwischen diesen Klassen und Interfaces:



Interfaces sind, wie Klassen und abstrakte Klassen, als Typen verwendbar. In dieser Hinsicht unterscheiden sie sich nicht von abstrakten und konkreten Klassen. Wie für abstrakte Klassen ohne Implementierungen gilt die Faustregel, dass Interfaces stabiler sind als Klassen mit Implementierungen. Aufgrund der 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.

Interfaces dienen fast ausschließlich der Festlegung von Untertypbeziehungen, die das Ersetzbarkeitsprinzip erfüllen. Reine Vererbung ist mit Interfaces kaum realisierbar. Wie bei Klassen gilt auch bei Interfa-

ces, dass die entsprechenden Typen gemachte Zusicherungen einschließen. Man sollte also stets Zusicherungen hinschreiben und die Kompatibilität der Zusicherungen händisch überprüfen. Obiges Beispiel ist eigentlich unvollständig, da Zusicherungen fehlen.

Kapitel 3

Generizität und Ad-hoc-Polymorphismus

In Kapitel 2 haben wir uns mit enthaltendem Polymorphismus beschäftigt. In Kapitel 3 werden wir alle weiteren Arten von Polymorphismus, die wir in objektorientierten Sprachen finden, betrachten. Die Abschnitte 3.1 und 3.2 sind der Generizität und ihrer praktischen Verwendung gewidmet. Einige objektorientierte Sprachen, wie beispielsweise Java, unterstützen Generizität jedoch nicht. Daher 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 eine Möglichkeit aufzeigen, mehrfaches dynamisches Binden in Programmiersprachen zu verwenden, die nur einfaches dynamisches Binden bereitstellen. Obwohl Ausnahmebehandlungen nicht zu den klassischen Ausformungen des Polymorphismus zählen, werden wir uns damit in Abschnitt 3.5 beschäftigen.

3.1 Generizität

Generische Klassen, Typen und Routinen können Typparameter enthalten, für die Typen eingesetzt werden. Damit ist Generizität eine weitere Form des universellen Polymorphismus, der die Wiederverwendung verbessern kann. Da Java derzeit keine Generizität unterstützt, sind viele Beispiele in diesem Abschnitt in GJ (Generic Java), einer Erweiterung von Java um verschiedene Programmiersprachkonzepte geschrieben.

3.1.1 Wozu Generizität?

Bei der Programmierung mit Generizität werden an Stelle expliziter Typen im Programm Typparameter verwendet. Typparameter sind Namen, die später durch Typen ersetzt werden. Anhand eines Beispiels wollen wir zeigen, dass eine Verwendung von Typparametern anstelle von Typen und die spätere Ersetzung der Typparameter durch Typen sinnvoll sein kann:

Beispiel. Programmcode für Listen soll entwickelt werden. Alle Elemente in einer Liste sollen vom selben Typ, sagen wir `String` sein. Es ist einfach, entsprechenden Programmcode zu schreiben. Bald stellt sich jedoch heraus, dass wir auch eine Liste mit Elementen vom Typ `Integer` sowie eine mit Instanzen von `Student` brauchen. Da der existierende Programmcode 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`. Der Name `Element` ist dabei kein tatsächlich existierender Typ, sondern einfach nur ein Typparameter. Den Code für Listen mit Instanzen von `String`, `Integer` und `Student` kann man daraus erzeugen, indem man alle Vorkommen von `Element` im Programmcode durch diese Typnamen ersetzt.

Warum soll man den Code für Listen mit einem Typparameter `Element` schreiben? Diesen Effekt kann man anscheinend auch erzielen, wenn man alle Vorkommen von `String` im Code der Listen von Zeichenketten durch `Integer` beziehungsweise `Student` ersetzt. Leider gibt es dabei aber ein Problem: Der Name `String` kann auch für ganz andere Zwecke als für Elementtypen 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 in keiner anderen Bedeutung vorkommt.

(Anmerkungen zu Java)

Die Klasse `Object` in Java enthält die Methode `public String toString()`, die eine Zeichenkette zur Beschreibung der Instanz von `Object` zurück gibt. Jede Klasse erweitert `Object` und erbt oder überschreibt diese Methode.

Natürlich kann man sich Schreibaufwand ersparen, wenn man eine Kopie eines Programmstücks anfertigt und darin alle Vorkommen eines Typ-

parameters 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, warum viele moderne (nicht nur objektorientierte) Programmiersprachen Generizität unterstützen: ProgrammiererInnen schreiben ein Programmstück nur einmal und kennzeichnen 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. Änderungen sind nach dem nächsten Compiliervorgang überall sichtbar, wo das Programmstück verwendet wird.

In vielen Fällen braucht der Compiler gar keine Kopien der Programmstücke erzeugen, sondern kann ein und denselben übersetzten Code für unterschiedliche Zwecke – beispielsweise Listen von Instanzen unterschiedlicher Typen – verwenden. Generizität erspart damit nicht nur Schreibarbeit, sondern kann das übersetzte Programm auch kürzer und effizienter machen. Generizität bedeutet nur einen kleinen zusätzlichen Aufwand für den Compiler, aber in der Regel keinen nennenswerten Zusatzaufwand zur Laufzeit. Generizität ist damit ein rein statischer Mechanismus.

3.1.2 Einfache Generizität in GJ

Die Programmiersprache GJ entspricht im Wesentlichen Java, unterstützt aber auch eine einfache Form der Generizität. Generische Klassen und Interfaces haben ein oder mehrere Typparameter, die in spitze Klammern geschrieben und durch Beistriche voneinander getrennt sind. Innerhalb der Klassen und Interfaces sind diese Typparameter beinahe wie Klassennamen verwendbar. Das erste GJ-Beispiel verwendet zwei generische Interfaces mit je einem Typparameter `A`:

```
interface Collection<A> {
    void add (A elem);
    Iterator<A> iterator();
}
interface Iterator<A> {
    A next();
    boolean hasNext();
}
```

Mit diesen Definitionen bezeichnet beispielsweise `Collection<String>` ein Interface, das durch Ersetzung aller Vorkommen des Typparameters `A` im Rumpf von `Collection<A>` generiert wird. Daher enthält das Interface `Collection<String>` die Methoden `void add (String elem)` und `Iterator<String> iterator()`, wobei `Iterator<String>` die Methoden `String next()` und `boolean hasNext()` enthält. Der Typparameter kann durch den Namen jeder beliebigen Klasse und jedes Interfaces ersetzt werden, aber nicht durch Typen wie `int`, `char` oder `boolean`, deren Instanzen keine Referenzobjekte sind.

Die generische Klasse `List<A>` implementiert `Collection<A>`:

```
class List<A> implements Collection<A> {
    protected class Node {
        A elem;
        Node next = null;
        Node (A elem) { this.elem = elem; }
    }
    protected head = null, tail = null;
    protected class ListIter implements Iterator<A> {
        protected Node p = head;
        public boolean hasNext() { return p != null; }
        public A next() {
            if (p == null)
                return null;
            A elem = p.elem;
            p = p.next;
            return elem;
        }
    }
    public void add (A x) {
        if (head == null)
            tail = head = new Node(x);
        else
            tail = tail.next = new Node(x);
    }
    public Iterator<A> iterator() {
        return new ListIter();
    }
}
```

Diese Klasse enthält die beiden inneren Klassen `Node` und `ListIter`, deren Instanzen 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 werden nur Default-Konstruktoren verwendet. Explizite Konstruktoren haben in GJ dieselbe Syntax wie in Java, beispielsweise `List(){...}`; es werden also keine Typparameter für `List` explizit angeschrieben.

Folgendes Programmstück zeigt den Umgang mit generischen Klassen:

```
class ListTest {
    public static void main (String[] args) {
        List<Integer> xs = new List<Integer>();
        xs.add (new Integer(0));
        Integer x = xs.iterator().next();
        List<String> ys = new List<String>();
        ys.add ("zerro");
        String y = ys.iterator().next();
        List<List<Integer>> zs =
            new List<List<Integer>>();
        zs.add(xs);
        // zs.add(ys); ! Compiler meldet Fehler !
        List<Integer> z = zs.iterator().next();
    }
}
```

An `ListTest` fällt auf, dass statt einfacher Instanzen von `int` Instanzen der Standardklasse `Integer` verwendet werden müssen, da gewöhnliche Zahlen keine Referenzobjekte sind. In Java gibt es zu jedem einfachen Typ wie `int`, `char` oder `boolean` genau deswegen einen entsprechenden Referenztyp wie `Integer`, `Character` oder `Boolean` weil in einigen Sprachkonstrukten nur Referenztypen erlaubt sind. Referenztypen bieten im Großen und Ganzen dieselbe Funktionalität wie die einfachen Typen, da Instanzen stets vom einen Typ zum anderen konvertiert werden können. Ein Nachteil der Referenzobjekte ist der etwas kompliziertere und ineffizientere Umgang mit ihnen.

Das Beispiel zeigt, dass Listen auch andere Listen enthalten können. Allerdings muß jedes Listenelement den durch den Typparameter festgelegten Typ haben. Der GJ-Compiler ist klug genug, um eine Instanz von `List<Integer>` von einer Instanz von `List<String>` zu unterscheiden. Diese beiden Listentypen sind nicht miteinander kompatibel.

Generizität bietet statische Typsicherheit. Beispielsweise garantiert bereits der Compiler, dass in eine Instanz von `List<String>` nur Zeichenketten eingefügt werden können. Der Versuch, eine Instanz eines inkompatiblen Typs einzufügen, wird erkannt und als Fehler gemeldet. Geübte Java-ProgrammiererInnen, die den Umgang mit Collections, Listen und ähnliche Datenstrukturen ohne Unterstützung durch Generizität gewohnt sind, kennen die Probleme mangelnder statischer Typsicherheit, 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:

```
interface Comparator<A> {
    int compare (A x, A y); // result <  0 if x <  y
                            // result == 0 if x == y
                            // result >  0 if x >  y
}
class Collections {
    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` im Interface `Comparator<A>` vergleicht zwei Objekte des selben Typs und retourniert das Vergleichsergebnis als ganze Zahl. Unterschiedliche Comparatoren, also voneinander verschiedene Objekte mit einem solchen Interface, werden unterschiedliche Vergleiche durchführen. Die statische Methode `max` in `Collections` wendet Comparatoren wiederholt auf Elemente in einer Instanz von `Collection<A>` an, um das maximale 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 getrennte Typparameter stehen.

Generische Methoden haben den Vorteil, dass man die für Typparameter zu verwendenden Typen nicht explizit angeben muss, wie folgendes Programmstück zeigt:

```
List<Integer> xs = ...;
List<String>  ys = ...;
Comparator<Integer> cx = ...;
Comparator<String>  cy = ...;
int rx = Collections.max (xs, cx);
int ry = Collections.max (ys, cy);
// int rz = Collections.max (xs, cy); ! Fehler !
```

Der Compiler erkennt 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. Den Vorgang zur Berechnung der Typen nennt man *Typinferenz*.

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

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

Ein Comparator für Zeichenketten wird zwar etwas komplizierter, aber nach demselben Schema aufgebaut sein.

3.1.3 F-gebundene Generizität in GJ

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 Instanzen dieser Typen bestimmte Methoden oder Variablen haben. Es kann ja jeder beliebige Referenztyp den Typparameter ersetzen.

Über manche Typparameter benötigt man mehr Information, um auf Instanzen der entsprechenden Typen zugreifen zu können. Gebundene Typparameter liefern diese zusätzliche Information: In GJ kann man für jeden Typparameter eine Klasse oder ein Interface als Schranke angeben. Nur Untertypen dieser Schranke dürfen den Typparameter ersetzen. Damit ist statisch bekannt, dass in jeder Instanz des Typs, für den der Typparameter steht, die in der Schranke festgelegten Methoden und Variablen verwendbar sind. Man kann Instanzen des Typparameters daher wie Instanzen der Schranke verwenden, wie folgendes Beispiel zeigt:

```
interface Hashable {
    int hashCode();
}
class Hashtable<Key implements Hashable, Value> {
    public void put (Key k, Value v) {
        int index = k.hashCode();
        ...
    }
    ...
}
```

Die generische Klasse `Hashtable` hat zwei Typparameter. Auf `Key` ist `Hashable` als Schranke definiert; auf `Value` gibt es keine Schranke. Wir verwenden die Schranke auf `Key`, damit wir im Rumpf von `put` die Methode `hashCode` in der Instanz `k` des Typs, der `Key` ersetzt, aufrufen können. Das als Schranke verwendete Interface beschreibt diese Methode.

Wenn die Schranke ein Interface ist, schreibt man sie nach dem Typparameter und dem Schlüsselwort `implements` hin, wie im Beispiel. Bei einer Klasse verwendet man als Schlüsselwort `extends`. Pro Typparameter ist nur eine Klasse oder ein Interface als Schranke erlaubt. Ist ein Typparameter ungebunden, das heißt, es ist keine Schranke angegeben, wird implizit `Object` als Schranke angenommen, da jede Klasse von `Object` abgeleitet ist. Die in `Object` definierten Methoden sind daher immer verwendbar.

(Anmerkungen zu Java)

In Java beschreibt bereits die Klasse `Object` die Methode `hashCode` wie in obigem Beispiel. Daher ist es gar nicht nötig, `Hashable` als Schranke für `Key` anzugeben; `hashCode` ist in jedem Fall aufrufbar. Generell beschreibt `Object` relativ viele komplexe Methoden – ein häufiger Kritikpunkt an Java, da diese Methoden auch vorhanden sein müssen, wenn sie gar nicht gebraucht werden. Andere objektorientierte Sprachen unterstützen wesentlich schlankere Klassen. Eine Ursache für den Umfang von `Object`

liegt am Fehlen von Mehrfachvererbung: Die häufigsten Nachteile der ausschließlichen Einfachvererbung werden umgangen, indem alle oft benötigten Methoden in der Wurzel der Typhierarchie definiert sind.

Das nächste Beispiel zeigt, dass Typparameter auch rekursiv verwendet werden dürfen. Obiges Beispiel mit Comparatoren wird abgewandelt:

```
interface Comparable<A> {
    int compareTo (A that); // res. <  0 if this <  that
                            // res. == 0 if this == that
                            // res. >  0 if this >  that
}
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;
    }
}
class Collections2 {
    public static <A implements 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` 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. Auch in der Schranke des Typparameters `A` von `max` in `Collections2` kommt eine ähnliche Rekursion vor. Diese Form der Generizität mit rekursiven Typparametern wird nach dem

formalen Modell, in dem solche Konzepte untersucht wurden, *F-gebundene Generizität* genannt.

Die Klasse `Integer` in obigem Beispiel entspricht, abgesehen von der Verwendung von Generizität, einer vereinfachten Form der in Java standardmäßig vorhandenen Klasse gleichen Namens. GJ hat alle Standardklassen und -interfaces um Generizität erweitert und beinahe nahtlos integriert, ohne inhaltliche Änderungen an den Javaklassen und -interfaces vornehmen zu müssen.

Generizität in GJ unterstützt keine impliziten Untertypbeziehungen. Zum Beispiel besteht zwischen `List<X>` und `List<Y>` keine Untertypbeziehung wenn `X` und `Y` verschieden sind, auch dann nicht, wenn `Y` von `X` abgeleitet ist. 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` ungleich `List<X>` ist.

In Java wie auch in GJ können bei Verwendung von Arrays leicht Typfehler zur Laufzeit auftreten, die bei Verwendung von Generizität in GJ wegen der Nichtunterstützung impliziter Untertypbeziehungen vom Compiler abgefangen werden, wie folgendes Beispiel zeigt:

```
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();
    }
}
class Loophole {
    public static String loophole (Integer y) {
        String[] xs = new String[10];
        Object[] ys = xs;  // no compile-time error
        ys[0] = y;
        return xs[0];      // throws exception
    }
}
```

Die Klasse `Loophole` wird vom Compiler unbeanstandet übersetzt, da in Java für jede Untertypbeziehung auf Typen automatisch eine entsprechende Untertypbeziehung auf Arrays von Elementen solcher Typen angenommen wird, obwohl das Ersetzbarkeitsprinzip verletzt sein kann. Im Beispiel nimmt der Compiler an, dass `String[]` ein Untertyp von `Object[]` ist, da `String` ein Untertyp von `Object` ist. Tatsächlich ist diese Annahme falsch, wie das Beispiel verdeutlicht.

GJ wurde aus Java entwickelt, ohne die JVM (java virtual machine) zu ändern und kommt mit minimalen Änderungen von Java aus. Aufgrund von Kompatibilitätsbedingungen mussten in GJ Kompromisse gemacht werden, die die Verwendbarkeit von Generizität einschränken. Generell, also in anderen Sprachen als Java und GJ, treten keine solchen Unterschiede in der Typsicherheit von Arrays und generischen Collections auf. Im Gegenteil: Der einfachere und sicherere Umgang mit Arrays dient manchmal als Begründung für die Einführung von Generizität in eine Programmiersprache. Als weitere Einschränkung in GJ können Typparameter nicht zur Erzeugung neuer Objekte verwendet werden. Daher ist `new A()` illegal, wenn `A` ein Typparameter ist. Eigentlich gibt es auch keinen Bedarf für derartige Verwendungen von Typparametern. In der Praxis iteressanter ist der Ausdruck `new A[n]`, der ein neues Array für `n` Instanzen 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. GJ erlaubt die Verwendung aber trotzdem und erzeugt wie Java eine Exception zur Laufzeit, falls ein Typfehler auftritt. Weitere Einschränkungen der Generizität im Zusammenhang mit Java gibt es bei expliziten Typkonvertierungen, wie wir später sehen werden.

Die Kompatibilität von GJ mit Java hat noch einen kleinen Nachteil: Programme führen zur Laufzeit Typüberprüfungen durch, obwohl der Compiler bereits zur Übersetzungszeit zugesichert hat, dass solche Fehler gar nicht auftreten können. Daher ist mit der Verwendung von Generizität ein kleiner Verlust an Laufzeiteffizienz verbunden. In anderen Programmiersprachen hat Generizität keinen Einfluß auf die Laufzeit.

3.2 Verwendung von Generizität im Allgemeinen

Wir wollen nun betrachten, wie man Generizität in der Praxis einsetzt. Unterabschnitt 3.2.1 gibt einige allgemeine Ratschläge, in welchen Fällen

sich die Verwendung auszahlt. In Unterabschnitt 3.2.2 werden wir uns kurz mit möglichen Übersetzungen generischer Klassen beschäftigen und einige Alternativen zu GJ in anderen Sprachen vorstellen, um ein etwas umfassenderes Bild davon zu bekommen, was Generizität leisten kann.

3.2.1 Richtlinien für die Verwendung von Generizität

Wann und wie soll man Generizität einsetzen? Generell ist der Einsatz 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 (oder Faustregeln) anführen:

Gleich strukturierte Klassen oder Routinen. Man soll Generizität immer verwenden, wenn es mehrere gleich strukturierte Klassen (oder Typen) beziehungsweise Routinen gibt, oder voraussehbar ist, dass es solche geben wird. Typische Beispiele dafür sind Containerklassen wie Listen, Stacks, Hashtabellen, Mengen, etc. und Routinen, 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 Instanzen anderer Typen sinnvoll ist. 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.

Es zahlt sich aus, Generizität bereits beim geringsten Verdacht, dass eine Containerklasse auch für andere Elementtypen sinnvoll sein könnte, zu verwenden: Elementtypen sind in der Objektschnittstelle sichtbar, und Änderungen der Schnittstelle verursachen einen erheblichen Wartungsaufwand. Man will daher nach Möglichkeit vermeiden, dass diese Typen nachträglich geändert werden müssen. Dies kann man erreichen, indem man statt konkreter Elementtypen in der Schnittstelle nur Typparameter verwendet. Eine nachträgliche Änderung der Elementtypen in einem Container ist damit ohne großen Wartungsaufwand möglich. Andererseits verursacht die Verwendung von Generizität bei der ursprünglichen Erstellung der Containerklasse nur einen unbedeutenden, vernachlässigbaren Mehraufwand. Die Laufzeiteffizienz wird durch die Verwendung von Generizität kaum oder überhaupt nicht beeinträchtigt. Es zahlt sich daher aus, Generizität bereits frühzeitig zu verwenden.

Obwohl man auf einen vernünftigen Einsatz von Generizität achtet, passiert es leicht, dass man die Sinnhaftigkeit von Typparametern an bestimmten Stellen im Programm erst spät erkennt. In diesen Fällen soll man das Programm so schnell wie möglich refaktorisieren, also die Klasse oder Routine mit Typparametern versehen. Ein Hinauszögern der Refaktorisierung führt leicht zu unnötigem Programmcode.

Üblicher Programmcode enthält nur relativ wenige generische Containerklassen. Der Grund dafür liegt einfach darin, dass die meisten Programmiersprachen mit umfangreichen Bibliotheken ausgestattet sind, welche die am häufigsten verwendeten, immer wieder gleich strukturierten Klassen und Routinen bereits enthalten. Man braucht diese Klassen und Routinen also nur zu verwenden, statt sie neu schreiben zu müssen. Sehr viele Klassen und Routinen in Bibliotheken sollten generisch sein. Für Java trifft das leider nicht zu, da die Sprache keine Generizität unterstützt. Aber GJ bietet die Java-Bibliotheken praktisch unverändert mit durchwegs generischen Schnittstellen an.

Abfangen erwarteter Änderungen. Generizität ermöglicht es, Programmteile unverändert zu lassen, obwohl sich Typen ändern. Insbesondere betrifft das Typen von formalen Parametern. Generizität ist hervorragend dafür geeignet, erwartete Änderungen der Typen von formalen Parametern bereits im Voraus zu berücksichtigen. Man soll daher gleich von Anfang an Typparameter verwenden, wenn man sich erwartet, dass sich Typen formaler Parameter irgendwann ändern. Das gilt auch dann, wenn es sich nicht um Elementtypen in Containerklassen handelt.

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 vermutlich 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. Bei einer Erweiterung der Geschäftstätigkeit der Bank erleichtert dies die Programmänderung wesentlich. Vor einiger Zeit wäre dieses Beispiel für eine Änderung der Währung von Schilling auf Euro noch überzeugender gewesen. Ebenso ist es vermutlich sinnvoll, den Inhaber des Kontos nicht auf eine Person festzulegen, da ja auch Firmen Konten haben und einzelne Konten mehreren oder vielleicht auch keinem Inhaber mehr zuordenbar sein können. Auch in diesem Fall ist ein Typparameter hilfreich. Anders als

bei vielen Typparametern in Containerklassen werden wir hier aber gebundene Typparameter verwenden. Von einer Währung oder einem Kontoinhaber erwarten wir ja, dass sie bestimmte Eigenschaften erfüllen. Wir erwarten also, dass alle Währungen oder Kontoinhaber, die zusammen mit dem Konto verwendet werden, Untertypen von bestimmten Klassen oder Interfaces, sagen wir `Currency` und `Owner`, sind.

Untertyprelationen und Generizität sind manchmal eng miteinander verknüpft, wie das Beispiel zeigt. 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, um Änderungen von Typen formaler Parameter abzufangen, wenn das Ersetzbarkeitsprinzip nicht erfüllt ist, während Untertypbeziehungen den Ersatz einer Instanz eines Obertyps durch eine Instanz eines Untertyps auch unabhängig von Typen formaler Parameter ermöglichen.

Verwendbarkeit. Generizität und Untertypbeziehungen sind Konzepte in Programmiersprachen, die oft gegeneinander austauschbar sind. Das heißt, man kann ein und dieselbe Aufgabe mit Generizität oder über Untertypbeziehungen lösen, ohne dass eine Lösung der anderen hinsichtlich Wartbarkeit überlegen wäre. Ist es daher vielleicht möglich, dass ein Konzept das andere komplett ersetzen kann? Das ist nicht möglich, wie man an folgenden zwei Beispielen sieht:

Generizität ist sehr gut dafür geeignet, wie in obigen Beispielen eine Listenklasse zu schreiben, wobei eine Instanz nur Elemente eines Typs enthält und eine andere Instanz 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 Listenklasse 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 beispielsweise nur durch Untertypbeziehungen, nicht machbar wäre.

Mit Generizität ohne Untertypbeziehungen, also auch ohne gebundene Generizität, 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 Untertypbeziehungen, 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 und/oder Untertypbeziehungen zur Erreichung des Ziels unbedingt verwenden muss.

Laufzeiteffizienz. Wie oben argumentiert hat die Verwendung von Generizität keine oder zumindest kaum negative Auswirkungen auf die Laufzeiteffizienz. Andererseits ist die Verwendung von dynamischem Binden in Zusammenhang mit Untertypbeziehungen immer etwas weniger effizient als statisches Binden. Aufgrund dieser Überlegungen kommen ProgrammiererInnen 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 durch die Strukturänderung eine switch-Anweisung zusätzlich ausgeführt werden muss, ist die Effizienz ziemlich sicher schlechter geworden. Deswegen soll man Effizienzüberlegungen in der Entscheidung, ob man Generizität oder Untertypbeziehungen einsetzt, beiseite lassen. Solche Optimierungen auf der untersten Ebene sind wirklich nur etwas für Experten, die Details ihrer Compiler und ihrer Hardware sehr gut kennen, und auch dann sind die Optimierungen meist nicht portabel. Viel wichtiger ist es, auf die Einfachheit und Verständlichkeit des Programms zu achten.

Natürlichkeit. Häufig hört man auf die Frage, ob man in einer bestimmten Situation Generizit oder Subtyping einsetzen soll, die Antwort, dass der natürlichere Mechanismus der am besten geeignete sei. Für erfahrene EntwicklerInnen ist diese Antwort durchaus zutreffend: Mit einem gewissen Erfahrungsschatz kommt es ihnen ganz selbstverständlich

vor, den richtigen Mechanismus zu wählen, ohne die Entscheidung wirklich begründen zu können. Hinter der Natürlichkeit eines bestimmten Lösungsweges verbirgt sich oft ein großer Erfahrungsschatz. Leider sehen AnfängerInnen kaum, was natürlicher ist. Daher ist der Ratschlag an AnfängerInnen, den natürlicheren Mechanismus zu wählen, mit Vorsicht zu genießen. 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 Routinen in ausführbaren Code gibt es zwei Möglichkeiten, die homogene und die heterogene Übersetzung. In GJ wird eine *homogene* Übersetzung verwendet. Dabei wird jede generische Klasse, genauso wie jede nichtgenerische Klasse auch, in genau eine Klasse mit JVM-Code übersetzt. Jeder gebundene Typparameter wird im übersetzten Code einfach durch die Schranke des Typparameters ersetzt, jeder ungebundene Typparameter durch `Object`. Wenn eine Methode eine Instanz eines Typparameters zurück gibt, wird der Typ der Instanz nach dem Methodenaufruf 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 aber vom Compiler garantiert.

Bei der *heterogenen* Übersetzung wird für jede Verwendung einer generischen Klasse oder Routine mit anderen Typparametern eigener übersetzter Code erzeugt. Die heterogene Übersetzung entspricht also eher der Verwendung von „copy and paste“, wie in Unterabschnitt 3.1.1 argumentiert, wobei in jeder Kopie alle Vorkommen von Typparametern durch die entsprechenden Typen ersetzt sind. Dem Nachteil einer größeren Anzahl übersetzter Klassen und Routinen stehen einige Vorteile gegenüber: Da für alle Typen eigener Code erzeugt wird, sind einfache Typen wie `int`, `char` oder `boolean` ganz problemlos, ohne Einbußen an Laufzeiteffizienz als

Ersatz für Typparameter geeignet. Zur Laufzeit brauchen keine Typumwandlungen und damit zusammenhängende Überprüfungen durchgeführt werden. Außerdem sind auf jede übersetzte Klasse eigene Optimierungen anwendbar, die von den Typen abhängen. Daher haben Programme bei heterogener Übersetzung oft bessere Laufzeiteffizienz. Heterogene Übersetzung wird beispielsweise für *templates* in C++ verwendet.

Einige Programmiersprachen legen nicht fest, ob homogene oder heterogene Übersetzung verwendet wird. In diesen Sprachen wählt der Compiler eine heterogene Übersetzung, wenn dies für einfache Typen notwendig ist oder zu besserer Laufzeiteffizienz führt. Manchmal, beispielsweise in eingebetteten Systemen, ist Speichereffizienz wichtiger als Laufzeiteffizienz, also eine homogene Übersetzung besser geeignet als eine heterogene.

Große Unterschiede zwischen Programmiersprachen gibt es im Zusammenhang mit gebundener Generizität. Sprachen wie GJ und Eiffel verlangen, dass ProgrammiererInnen eine Schranke vorgeben und nur Untertypen der Schranke den Typparameter ersetzen können. ProgrammiererInnen 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.

Durch die heterogene Übersetzung von templates brauchen ProgrammiererInnen in C++ keine Schranken angeben, um Eigenschaften der Typen, die Typparameter ersetzen, verwenden zu können. Es wird einfach für jede übersetzte Klasse getrennt überprüft, ob die Typen alle vorausgesetzten Eigenschaften erfüllen. In dieser Hinsicht ist Generizität mit heterogener Übersetzung flexibler als Generizität mit homogener Übersetzung. Unterschiedliche Typen, die einen Typparameter ersetzen, brauchen keinen gemeinsamen Obertyp haben. Allerdings vermindert die heterogene Übersetzung oft die Qualität von Fehlermeldungen, da sie sich auch auf generierten Programmcode beziehen können, den ProgrammiererInnen normalerweise gar nicht zu sehen bekommen.

(für Interessierte)

Folgendes Beispiel zeigt einen Ausschnitt aus einer generische 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;
    ...
};
```

```
Pair<int>    anIntPair (2, 3);
Pair<Person> aPersonPair (new Person ("Susi"),
                          new Person ("Strolchi"));
Pair<char *> aStringPair ("Susi", "Strolchi");
```

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. Die erste Verwendung der Klasse ersetzt `T` durch `int` und erzeugt eine neue Variable `anIntPair`, die mit einer neuen Instanz von `Pair` initialisiert ist. Der Konstruktor wird implizit 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 Verwendungen korrekt sind, hängt davon ab, ob ProgrammiererInnen für die Typen `Person` und `char*` (Zeiger auf Zeichen, als String verwendbar) den Operator `+` implementiert hat. (In C++ können Operatoren überladen werden.) Falls dem so ist, werden im Rumpf von `sum` die entsprechenden Implementierungen von `+` aufgerufen. Andernfalls 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 GJ erkennt der Compiler anhand der Typen der aktuellen Parameter, welcher Typ für den Typparameter zu verwenden ist. In diesem Beispiel wird vorausgesetzt, dass der Operator `>` auf allen Typen, die für `T` verwendet werden, definiert ist.

Eine noch flexiblere Variante für den Umgang mit gebundenen Typparametern, die keine heterogene Übersetzung voraussetzt, wurde in Ada gewählt. Als Schranke geben ProgrammiererInnen keinen Typ an, sondern Eigenschaften, welche die Typen, die Typparameter ersetzen, erfüllen müssen. Beispielsweise wird explizit angegeben, dass ein Typ eine Routine mit bestimmten Parametern unterstützt. Auf den ersten Blick ist diese Variante genauso flexibel wie templates in C++. Jedoch kann man in Ada

bei jeder Verwendung einer generischen Einheit getrennt angeben, welche Routine eines Typs für eine bestimmte Eigenschaft verwendet werden soll. Routinen werden wie Typen als generische Parameter behandelt.

(für Interessierte)

Eine generische Funktion in Ada 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 der Klausel „`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 auf ganzen Zahlen verwendet, so dass von `IntMin` das kleinere Argument zurück gegeben wird. Anders als in C++ und GJ ergeben sich die für Typparameter zu verwendenden Typen nicht implizit aus der Verwendung, sondern müssen explizit angegeben werden. Dies entspricht der Philosophie von Ada, wonach alles, was die Bedeutung eines Programms beeinflussen kann, explizit im Programm stehen soll, um die Lesbarkeit zu erhöhen.

Ohne Sprachunterstützung ist eine Flexibilität wie in Ada, beispielsweise durch Verwendung von Comparatoren als zusätzliche Parameter (siehe Unterabschnitt 3.1.2) auch erzielbar, jedoch nur mit hohem Aufwand. Noch höher ist der notwendige zusätzliche Aufwand, wenn die bereits existierenden Klassenhierarchien die Angabe einer passenden Schranke nicht erlauben. In diesem Fall muss man Klassenhierarchien ändern, oder passende Klassenhierarchien nachprogrammieren.

3.3 Typabfragen und Typumwandlung

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 keine dynamische Typinformation. Im Gegensatz dazu wird in objektorientierten Programmiersprachen dynamische Typinformation für das dynamische Binden zur Ausführungszeit benötigt. Da dynamische Typinformation ohnehin verfügbar ist, erlauben viele objektorientierte Sprachen 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 Klasse ist, sowie zur Umwandlung des deklarierten Typs eines Objekts. Wir wollen den Umgang mit dynamischer Typinformation untersuchen. In Unterabschnitt 3.3.1 finden sich allgemeine Hinweise dazu. In den nächsten beiden Unterabschnitten werden spezifische Probleme durch Verwendung dynamischer Typinformation gelöst: Unterabschnitt 3.3.2 ist Problemen gewidmet, die durch die fehlende Generizität in Java entstehen. In Unterabschnitt 3.3.3 werden kovariante Probleme behandelt, die sich aus der Unmöglichkeit von kovarianten Parametertypen in Untertyprelationen ergeben.

3.3.1 Verwendung dynamischer Typinformation

Jede Instanz von `Object` (und daher überhaupt jedes Referenzobjekt) hat eine Methode `getClass`, welche die Klasse des Objekts als Ergebnis zurück gibt. Diese Methode bietet die direkteste Möglichkeit des Zugriffs auf den dynamischen Typ. Davon wird aber nur selten Gebrauch gemacht, da diese Typinformation an Programmstellen, an denen die Klasse eines Objekts nicht ohnehin bekannt ist, kaum benötigt wird.

Wesentlich 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 (entsprechend der Typhierarchie aus Unterabschnitt 2.1.2) `p` vom Typ `Student`, `Studienassistent` oder `Werkstudent` ist. Die Abfrage liefert `false`, wenn `p` gleich `null` oder von einem anderen dynamischen Typ ist. Solche dynamische Typabfragen können, wie alle anderen Vergleichsoperationen, an beliebigen Programmstellen stehen, an denen Boolsche Ausdrücke erlaubt sind.

Mittels Typabfragen lässt sich zwar der Typ eines Objektes zur Laufzeit bestimmen, aber Typabfragen reichen nicht aus, um Eigenschaften des dynamich 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. Folgendes Beispiel modifiziert ein Beispiel aus Unterabschnitt 2.1.1:

```
class Point3D extends Point2D {
    private int z;
    public boolean equal (Point2D p) {
        if (p instanceof Point3D)
            return super.equal(p)
                   && ((Point3D)p).z == z;
        return false;
    }
}
```

Die Methode `equal` liefert als Ergebnis `false`, wenn der dynamische Typ des Arguments kein Untertyp von `Point3D` ist. Sonst wird die Methode aus `Point2D` aufgerufen, und die zusätzlichen Instanzvariablen `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 Instanzen 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 Unterabschnitt 3.3.3.

Typumwandlungen sind auf Referenzobjekten nur durchführbar, wenn der Ausdruck, dessen deklarierter Typ in einen anderen Typ umgewan-

delt 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 sehr umständlich machbar wäre. Allerdings kann die 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 oft nur mehr wenig mit dem Typ zu tun hat, dessen Instanzen ProgrammiererInnen als Werte erwarten. Beispielsweise ist der deklarierte Typ `Objekt`, obwohl ProgrammiererInnen sich erwartet, dass nur Instanzen von `Integer` oder `Person` vorkommen; einen konkreteren gemeinsamen Obertyp dieser beiden Typen als `Objekt` gibt es im System ja nicht. Um auf Eigenschaften von `Integer` oder `Person` zuzugreifen, werden dynamische Typabfragen und Typumwandlungen eingesetzt. Wenn in Wirklichkeit statt einer Instanz von `Integer` oder `Person` eine Instanz 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 es zu keiner Ausnahmebehandlung kommt, sondern einfach nur die Ergebnisse falsch sind, oder – noch schlimmer – falsche Daten 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 sich ProgrammiererInnen vermutlich nach wie vor auf statische Typsicherheit in Java verlassen.

Neben der Fehleranfälligkeit ist die schlechte Wartbarkeit ein weiterer Grund, um Typabfragen (auch ohne Typumwandlungen) nur sehr sparsam zu nutzen. Insbesondere gilt dies für geschachtelte Typabfragen:

```
if (x instanceof T1)
    doSomethingOfTypeT1 ((T1)x);
else if (x instanceof T2)
    doSomethingOfTypeT2 ((T2)x);
...
else
    doSomethingOfAnyType (x);
```

Wir wissen bereits aus Abschnitt 2.1, dass `switch`-Anweisungen und (geschachtelte) `if-then-else`-Anweisungen durch dynamisches Binden ersetzt werden können. Das soll man auch tun, da dynamisches Binden in der Regel wesentlich wartungsfreundlicher ist. Das gilt auch für (geschachtelte) dynamische Typabfragen, welche die möglichen Typen im Programmcode fix verdrahten und daher bei Änderungen der Typhierarchie ebenfalls geändert werden müssen. Oft sind solche (geschachtelte) dynamische Typabfragen einfach durch

```
x.doSomething();
```

ersetzbar. Die Auswahl des auszuführenden Programmcodes erfolgt hier durch dynamisches Binden. Die Klasse des deklarierten Typs von `x` implementiert `doSomething` entsprechend `doSomethingOfAnyType`, und die Unterklassen `T1`, `T2` und so weiter entsprechend `doSomethingOfTypeT1`, `doSomethingOfTypeT2` und so weiter.

Manchmal ist es nicht einfach, dynamische Typabfragen durch dynamisches Binden zu ersetzen. Dies trifft vor allem in diesen Fällen zu:

- Der deklarierte Typ von `x` ist zu allgemein; die einzelnen Alternativen decken nicht alle Möglichkeiten ab. Das ist genau die oben erwähnte gefährliche Situation, in der die statische Typsicherheit von Java umgangen wird. In dieser Situation ist eine Refaktorisierung des Programms angebracht.

- Die Klassen, die dem deklarierten Typ von `x` und dessen Untertypen entsprechen, können nicht erweitert werden. Als (recht aufwendige) Lösung kann man parallel zur unveränderbaren Klassenhierarchie eine gleich strukturierte Hierarchie aufbauen, deren Klassen (Wrapper-Klassen) die zusätzlichen Methoden beschreiben.

- Manchmal ist die Verwendung dynamischen Bindens statt dynamischer Typabfragen schwierig, weil die einzelnen Alternativen auf private Variablen und Methoden zugreifen. Methoden anderer Klassen ist diese Information nicht zugänglich. Oft lässt sich die fehlende Information durch Übergabe geeigneter Argumente beim Aufruf der Methode 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.

Ein wichtiger Ratschlag an Java-ProgrammiererInnen ist, Typabfragen und Typumwandlungen nur sehr sparsam einzusetzen. 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 primitiven Typen wie `int`, `char` und `float` unterstützt. In diesen Fällen haben Typumwandlungen aber eine andere Bedeutung, da für Variablen primitiver Typen die dynamischen Typen immer gleich den statischen und deklarierten Typen sind. Bei primitiven Typen wird tatsächlich die Instanz des Typs umgewandelt, nicht der deklarierte Typ. Aus einer Fließkommazahl wird beispielsweise durch Abschneiden der Stellen nach dem Komma, also durch Runden gegen Null, eine ganze Zahl. Dabei kann natürlich Information verloren gehen, aber es kommt zu keiner Ausnahmebehandlung. Daher haben Typumwandlungen auf primitiven Typen eine ganz andere Qualität als auf Referenztypen und machen im Normalfall keinerlei Probleme. Typumwandlungen zwischen primitiven Typen und Referenztypen werden nicht unterstützt.

3.3.2 Typumwandlungen und Generizität

Die homogene Übersetzung einer generischen Klasse oder Routine in eine Klasse oder Routine ohne Generizität ist im Prinzip sehr einfach: Alle Ausdrücke in spitzen Klammern werden weggelassen. Jedes andere Vorkommen eines Typparameters wird durch `Object` oder, falls eine Schranke angegeben ist, durch die Schranke ersetzt. Für das Beispiel aus Unterabschnitt 3.1.2 wird folgender Code erzeugt:

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

```
class List implements Collection {
    protected class Node {
        Object elem;
        Node next = null;
        Node (Object elem) { this.elem = elem; }
    }
    protected head = null, tail = null;
    protected class ListIter implements Iterator {
        protected Node p = head;
        public boolean hasNext() { return p != null; }
        public Object next() {
            if (p == null)
                return null;
            Object elem = p.elem;
            p = p.next;
            return elem;
        }
    }
    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();
    }
}
```

Etwas komplizierter ist die Verwendung einer generischen Routine oder einer Methode in einer generischen Klasse, die eine Instanz eines Typparameters als Ergebnis zurück gibt. Das Ergebnis muss in eine Instanz des Typs, der den Typparameter ersetzt, umgewandelt werden:

```
List xs = new List();
xs.add (new Integer(0));
Integer x = (Integer)xs.iterator().next();
List ys = new List();
ys.add ("zerro");
String y = (String)ys.iterator().next();
```

Abgesehen von einigen unbedeutenden Details, auf die wir hier nicht näher eingehen, ist die Übersetzung so einfach, dass sie ProgrammiererInnen auch ohne Unterstützung durch den Compiler durchführen können. Das ist vermutlich ein Grund dafür, dass Java keine Generizität kennt. ProgrammiererInnen 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. Beispielsweise liefert der GJ-Compiler für

```
List<Integer> xs = new List<Integer>();
xs.add (new Integer(0));
String y = xs.iterator().next();
```

eine Fehlermeldung, nicht aber für folgenden Code:

```
List xs = new List();
xs.add (new Integer(0));
String y = (String)xs.iterator().next();
```

Der Vorteil von Generizität liegt also in erster Linie in der höheren Typsicherheit.

Praktisch alle 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 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.

Dieser Argumentation folgend ist es leicht, sich auch bei der Programmierung in einer Sprache ohne Generizität einen Programmierstiel anzugewöhnen, der nur „sichere“ Typumwandlungen einsetzt: Typumwandlungen sind sicher (lösen keine Ausnahmebehandlung aus), wenn

- in einen Obertyp des statischen Objekttyps umgewandelt wird,
- oder davor eine dynamische Typabfrage erfolgt, die sicher stellt, dass das Objekt einen entsprechenden dynamischen Typ hat,
- oder man das Programmstück so schreibt, als ob man Generizität verwendet, 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 (up-cast). Die beiden anderen Fälle beziehen sich auf weniger harmlose Typumwandlungen nach unter (down-casts). Der zweite Punkt impliziert, dass es einen sinnvollen Programmzweig geben muss, der im Falle des Scheiterns des Typvergleichs ausgeführt wird. Leider erweisen sich gerade falsche Typannahmen in alternativen Zweigen zu Typabfragen als häufige Fehlerquellen. Bei Zutreffen des dritten Punktes braucht es keine solche Alternative geben. Statt dessen 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 des letzten Punktes kann die Intuition manchmal sehr 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.

Generizität ist, mit Einschränkungen, auch in dynamischen Typabfragen und Typumwandlungen einsetzbar, wie das nächste Beispiel zeigt:

```
static <A> Collection<A> up (List<A> xs) {
    return (Collection<A>) xs;
}
static <A> List<A> down (Collection<A> xs) {
    if (xs instanceof List<A>)
        return (List<A>)xs;
    else { ... }
}
static List<String> bad (Object o) {
    if (o instanceof List<String>)    // error
        return (List<String>)o;       // error
    else { ... }
}
```

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 überprüfen, 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 fest legt, falls es sich tatsächlich um eine Liste handelt. Der GJ-Compiler ist intelligent genug, solche Situationen zu erkennen. Bei der Übersetzung nach Java 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.

GJ erlaubt die gemischte Verwendung von generischen Klassen und Klassen, die durch homogene Übersetzung daraus erzeugt wurden:

```
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());
    }
}
```

In diesem Beispiel sind `List<A>` und `Iterator<A>` generische Klassen, und `List` und `Iterator` die entsprechenden übersetzten Klassen. Wenn Ausdrücke in spitzen Klammern angegeben sind, erfolgt die statische Typüberprüfung für generische Klassen, wenn sie weggelassen sind, erfolgen keine Überprüfungen durch den Compiler.

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 jedoch gerade kovariante Eingangsparametertypen. Entspre-

chende 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();
    }
}
```

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 realisierbar. 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 (vielleicht unerwarteten) Reaktionen des Tieres 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 vom Aufrufer vermutlich nicht erwünscht sind.

Kann man das Programm so umschreiben, dass die unerwünschten Aktionen sicher verhindert werden? Für ein kovariantes Problem geht das nicht. Durch Umschreiben des Programms kann man zwar Typabfragen und Typumwandlungen vermeiden, aber die unerwünschten Aktionen bleiben dadurch 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 genau kennt.

(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. Das heißt, der Compiler garantiert, dass `friss` in einer Instanz von `Rind` nur mit einem Argument vom Typ `Gras` und in einer Instanz von `Tiger` nur mit einem Argument vom Typ `Fleisch` aufgerufen wird. Kann der Compiler jedoch die Art des Tieres oder die Art des Futters nicht statisch feststellen, meldet er einen Fehler. Tatsächlich ergibt sich dadurch derselbe Effekt, als ob in Java die Klasse `Tier` keine Methode `friss` enthalten würde. Von einer echten Lösung des Problems kann man daher nicht sprechen.

Einen besseren Ansatz scheinen *virtuelle Typen* zu bieten, die derzeit noch in keiner gängigen Programmiersprache verwendet werden. 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` ist mit einer Klasse der Funktionalität von `Fleisch` überschrieben. 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`. Durch die Verwendung virtueller Typen hat man auf den ersten Blick nichts gewonnen: 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 virtuelle Typen haben im Gegensatz zu anderen Ansätzen einen Vorteil: Wenn `Tier` (und daher auch `Rind` und `Tiger`) eine Methode hat, die eine Instanz vom Typ `Futter` als Ergebnis liefert, kann man das Ergebnis eines solchen Methodenaufrufs als Argument eines Aufrufs von `friss` in derselben Instanz verwenden. Dabei braucht man die Art des Tieres nicht zu kennen und ist trotzdem vor Typfehlern sicher. Die Praxisrelevanz dieses Vorteils ist derzeit mangels Erfahrungen kaum abschätzbar.

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, die die Methode enthält. Im Prinzip kann man binäre Methoden auf dieselbe Weise behandeln wie alle anderen kovarianten Probleme. Hier ist eine weitere Lösung für `equal` in `Point2D` und `Point3D`:

```
abstract class Point {
    public boolean equal (Point that) {
        if (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` keine Spezialisierung von `Point2D` ist. In `Point` ist `equal` definiert und wird in den Unterklassen nicht überschrieben. 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 Unterabschnitt 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 Unterabschnitt 3.3.1 ist falsch, da `equal` in `Point2D` mit einem Argument vom Typ `Point3D` als Ergebnis `true` liefern kann.

(für Interessierte)

Die Programmiersprache Ada 95 bietet zwar keine Lösung für den Umgang mit kovarianten Problemen im Allgemeinen, aber binäre Methoden werden unterstützt: 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 95 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 an Hand des deklarierten Typs von `y` auswählen, welche der überladenen Methoden auszuführen ist. Für das dynamische Binden ist `y` 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 es 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*.

Leider haben Java- und C++-ProgrammiererInnen immer wieder Probleme damit, klar zwischen Überladen und Multimethoden zu unterscheiden. Das kann zu Fehlern führen. In Unterabschnitt 3.4.1 werden wir die Unterschiede zwischen Überladen und Multimethoden klar machen. In Unterabschnitt 3.4.2 werden wir sehen, dass man Multimethoden auch in Sprachen wie Java recht einfach simulieren kann.

3.4.1 Unterschiede zwischen Überladen und Multimethoden

Folgendes Beispiel soll vor Augen führen, dass bei der Auswahl zwischen überladenen Methoden in Java nur der deklarierte Typ eines Arguments entscheidend ist, nicht der dynamische Typ. Wir verwenden das Beispiel zu kovarianten Problemen aus Unterabschnitt 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 eine Instanz von `Gras` enthält; der dynamische Typ von `gras` ist `Gras`, da `gras` direkt vor dem Methodenaufruf mit einer Instanz 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 wissen ProgrammiererInnen in solchen Fällen, dass `gras` eine Instanz von `Gras` enthält, und nehmen an, dass die Methode mit dem

Parameter vom Typ `Gras` gewählt wird. Diese Annahme ist aber falsch! Man muss stets auf den deklarierten Typ achten, auch wenn man den dynamischen Typ kennt.

Was wäre, wenn die erste Zeile des Beispiels

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

lauten würde? Wegen dynamischem Binden würde `friss` natürlich weiterhin in `Rind` ausgeführt werden. Aber zur Auswahl überladener Methoden kann der Compiler nur deklarierte Typen verwenden. Die überladenen Methoden werden daher 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 `Gras` ist. Wie das Beispiel zeigt, kann sich die Auswahl zwischen überladenen Methoden stark von der Intuition vieler ProgrammiererInnen unterscheiden. Daher ist hier 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. Generell ist es empfehlenswert, Überladen nur so zu verwenden. Für je zwei unterschiedliche überladene Methoden

- soll es zumindest eine Parameterposition geben, an der sich die Typen der Parameter unterscheiden und nicht in Untertyprelation zueinander stehen,
- 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 mehr so wichtig.

Das Problem mit der häufigen Verwechslung von dynamischen und deklarierten Typen könnte man auch nachhaltig lösen, indem man zur Methodenauswahl generell die dynamischen Typen aller Argumente verwendet. Statt überladener Methoden hätte man dann Multimethoden.

Unter der Annahme, dass Java Multimethoden unterstützt, könnte man die Klasse `Rind` im Beispiel aus Unterabschnitt 3.3.3 kürzer schreiben:

```
class Rind extends Tier {
    public void friss (Gras x) { ... }
    public void friss (Futter x) {
        erhoeheWahrscheinlichkeitFuerBSE();
    }
}
```

Die Typabfrage, 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 Typ des Arguments ungleich `Gras` ist. Java unterstützt aber keine Multimethoden.

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, was sich negativ auf die Effizienz auswirken kann. 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 von Multimethoden versteht man oft etwas anderes als die damit verbundene Laufzeiteffizienz: Für ProgrammiererInnen ist nicht gleich erkennbar, unter welchen Bedingungen welche Methode ausgeführt wird. Eine allgemeine 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 im Allgemeinen aber nicht hinreichend, wenn Multimethoden mehrere Parameter haben, wie folgendes Beispiel zeigt:

```
frissDoppelt (Futter x, Gras y) { ... }
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 mehrere 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. Keine dieser Möglichkeiten bietet klare Vorteile gegenüber der anderen. Daher scheint eine weitere Variante günstig zu sein: Der Compiler verlangt, dass es immer eine eindeutige speziellste Methode gibt. ProgrammiererInnen müssen eine weitere Methode

```
frissDoppelt (Gras x, Gras y) { ... }
```

hinzufügen, die das Auswahlproblem beseitigt. Dieses Beispiel soll klar machen, dass Multimethoden im Allgemeinen tatsächlich sowohl für den Compiler als auch für ProgrammiererInnen eine deutlich höhere Komplexität haben als überladene Methoden. In den üblichen Anwendungsbeispielen haben Multimethoden keine höhere Komplexität als überladene Methoden. Die Frage, ob ProgrammiererInnen eher Multimethoden oder eher Überladen haben wollen, bleibt offen.

In Java kann es leicht zu Fehlern kommen, wenn ProgrammiererInnen unbewusst Überladen statt Überschreiben verwenden, wenn sie also eine Methode überschreiben wollen, die überschreibende Methode sich aber in den Parametertypen von der zu überschreibenden Methode unterscheidet. Man muss stets darauf achten, dass so etwas nicht passiert. Beispielsweise werden in C++ die häufigsten derartigen Probleme vom Compiler erkannt, weil es Einschränkungen beim Überladen ererbter Methoden gibt. In Java können ererbter Methoden beliebig überladen werden, wodurch unabsichtliches Überladen oft nur schwer erkennbar ist.

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 Unterabschnitt 3.3.3 und eliminieren damit dynamische Typabfragen und Typumwandlungen:

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

Die Methoden `friss` in `Rind` und `Tiger` machen nicht sehr viel. Sie rufen nur Methoden in `Futter` auf, die die eigentlichen Aufgaben durchführen. 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 Instanzen von `Rind` und `Tiger`. Bei einem Aufruf von `tier.friss(futter)` wird zwei mal dynamisch gebunden. Das erste dynamische Binden unterscheidet zwischen Instanzen von `Rind` und `Tiger`. Diese Unterscheidung spiegelt sich im Aufruf von `vonRindGefressen` und `vonTigerGefressen` wider. Ein zweites dynamisches Binden unterscheidet zwischen Instanzen von `Gras` und `Fleisch`. In den Unterklassen von `Futter` sind insgesamt vier Methoden implementiert, die alle möglichen Kombinationen von Tierarten mit Futterarten darstellen.

Die Namen der Methoden `vonRindGefressen` und `vonTigerGefressen` sind beliebig wählbar. Wegen der Möglichkeit des Überladens hätten wir für diese Methoden auch denselben Namen wählen können, da sie sich durch die Typen der formalen Parameter unterscheiden.

Stellen wir uns vor, diese Lösung des Beispiels sei dadurch zu Stande gekommen, 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 in Multimethoden 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 automatisch (ohne große intellektuelle Anstrengungen) durchführen.

Diese Lösung kann auch dadurch erzeugt worden sein, dass in der ursprünglichen Lösung aus Unterabschnitt 3.3.3 `if-then-else`-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.

Mehrfaches dynamisches Binden wird in der Praxis häufig benötigt. Die Lösung wie in unserem Beispiel entspricht dem *Visitor Pattern*, einem klassischen Entwurfsmuster. Klassen wie `Futter` werden Visitorklassen genannt, und Klassen wie `Tier` heißen Elementklassen. Visitor- und Elementklassen sind oft gegeneinander austauschbar.

Da unser Beispiel sehr klein ist, zeigt es den größten Nachteil des Visitor Patterns nicht: Die Anzahl der benötigten Methoden wird schnell sehr groß. Nehmen wir an, wir hätten M unterschiedliche Tierarten und N unterschiedliche Futterarten. Dann werden zusätzlich zu den M Visitor-Methoden $M \cdot N$ Methoden für die eigentliche Arbeit benötigt. Noch

rascher steigt die Anzahl der Methoden mit der Anzahl der dynamischen Bindungen. Bei $n \geq 2$ dynamischen Bindungen mit der Anzahl N_i an Möglichkeiten für die $i$te Bindung ($i = 1 \ldots n$) werden $N_1 \cdot N_2 \cdots N_n$ inhaltliche Methoden und $N_1 + N_1 \cdot N_2 + \cdots + N_1 \cdot N_2 \cdots N_{n-1}$ Visitor-Methoden benötigt, insgesamt also sehr sehr viele. Für $n = 4$ und $N_1, \ldots, N_4 = 10$ kommen wir bereits auf 11.110 Methoden. Außer für sehr kleine n und kleine N_i ist diese Technik nicht sinnvoll einsetzbar. Durch Vererbung lässt sich die Zahl der nötigen Methoden nur unwesentlich reduzieren.

Lösungen mit Multimethoden oder dynamischen Typabfragen haben manchmal einen großen Vorteil gegenüber Lösungen mit dem Visitor Pattern: Die Anzahl der nötigen Methoden bleibt viel kleiner. 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 Ausnahmebehandlung

Ausnahmebehandlungen dienen vor allem dem Umgang mit unerwünschten Programmzuständen. Beispielsweise werden in Java Ausnahmebehandlungen ausgelöst, wenn das Objekt bei einer Typumwandlung keine Instanz des gegebenen Typs ist, oder eine Nachricht an `null` gesendet wird. In diesen Fällen kann der Programmablauf nicht normal fortgeführt werden, da grundlegende Annahmen verletzt sind. Ausnahmebehandlungen geben ProgrammiererInnen die Möglichkeit, das Programm auch in solchen Situationen noch weiter ablaufen zu lassen. In Unterabschnitt 3.5.1 gehen wir auf Ausnahmebehandlungen in Java ein, und in Unterabschnitt 3.5.2 geben wir einige Hinweise auf den sinnvollen Einsatz von Ausnahmebehandlungen.

3.5.1 Ausnahmebehandlung in Java

Ausnahmen sind in Java gewöhnliche Objekte, die über spezielle Mechanismen als Ausnahmen verwendet werden. Alle Instanzen von `Throwable` sind dafür verwendbar. Praktisch verwendet man nur Instanzen der Unterklassen von `Error` und `Exception`, zwei Unterklassen von `Throwable`.

Unterklassen von `Error` werden hauptsächlich für vordefinierte schwer wiegende 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 wiederum in zwei Bereiche gegliedert – Ausnahmen, die von ProgrammiererInnen selbst definiert wurden, und Ausnahmen, die Instanzen von `RuntimeException` sind, einer vordefinierten Unterklasse von `Exception`. Beispiele für Unterklassen letzterer Klasse sind `IndexOutOfBoundsException` (bei versuchten Zugriffen auf Arrays außerhalb der erlaubten Indexbereiche), `NullPointerException` (beim Versuch, eine Nachricht an `null` zu senden) und `ClassCastException` (bei einer versuchten Typumwandlung, wenn der dynamische Typ dem gewünschten Typ nicht entspricht). Oft ist es sinnvoll, Ausnahmen, die Instanzen von `Exception` sind, abzufangen und den Programmablauf an geeigneter Stelle fortzusetzen.

Vom Java-Laufzeitsystem werden nur Instanzen der vordefinierten Unterklassen von `Error` und `RuntimeException` als Ausnahmen ausgelöst. ProgrammiererInnen können explizit Ausnahmen auslösen, die Instanzen jeder beliebigen Unterklasse von `Throwable` sind. Dies geschieht mit Hilfe der `throw`-Anweisung, wie im folgenden Beispiel:

```
class Help extends Exception { ... }
...
if (helpNeeded())
    throw new Help();
```

Falls `helpNeeded` als Ergebnis `true` liefert, wird eine neue Instanz 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 Schlüsselwort `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 eine Instanz 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 jede Instanz von `Help` auch eine Instanz von `Exception` ist. Wenn es mehrere passende `catch`-Klauseln gibt, wird einfach die erste passende gewählt. Nach einer abgefangenen Ausnahme wird das Programm so fortgesetzt, als ob es gar 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. ProgrammiererInnen müssen `catch`-Klauseln so schreiben, dass sie mögliche Inkonsistenzen beseitigen.

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 zurück gegeben 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. Letzterer Fall führt zum Programmabbruch.

Methoden dürfen nicht Ausnahmen beliebiger Typen zurück geben, sondern nur Instanzen von `Error` und `RuntimeException` sowie Ausnahmen von Typen, die im Kopf der Methode ausdrücklich angegeben sind. Soll eine Methode `foo` beispielsweise auch Instanzen von `Help` als Ausnahmen zurück geben können, so muss dies durch eine `throws`-Klausel deklariert sein:

```
void foo() throws Help { ... }
```

Alle möglichen Ausnahmen, die im Rumpf der Methode ausgelöst, aber

nicht zurück gegeben 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 Ausnahmentypen sind für das Bestehen von Untertyprelationen relevant. Das Ersetzbarkeitsprinzip verlangt, dass die Ausführung einer Methode eines Untertyps nur solche Ausnahmen zurück liefern 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 { ... }
}
```

Zum Abschluss seien noch `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, Resourcen auch beim Auftreten von Ausnahmen freizugeben. Dabei ist aber Vorsicht geboten, da oft nicht klar ist, ob eine bestimmte Resource bereits vor dem Auftreten einer Ausnahme angefordert war.

3.5.2 Einsatz von Ausnahmebehandlungen

Ausnahmen werden in folgenden Fällen eingesetzt:

Unvorhergesehene Programmabbrüche: Wird eine Ausnahme nicht abgefangen, kommt es zu einem Programmabbruch. Die entsprechende Bildschirmausgabe 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 weitergeht. Leider können Ausnahmebehandlungen echte Programmfehler nicht beheben, sondern nur den Benutzer darüber informieren und dann das Programm abbrechen, oder weiterhin (eingeschränkte) Dienste anbieten.

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 von ProgrammiererInnen erwartet (im Gegensatz zum Auftreten von bestimmten Fehlern). Es ist daher relativ leicht, entsprechende Ausnahmebehandlungen durchzuführen, die eine sinnvolle Weiterführung des Programms erlauben.

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 zurück

gegeben, die er abfangen kann. Damit ist es möglich, dass die Methode an den Aufrufer in Ausnahmesituationen Objekte zurück gibt, die nicht den deklarierten Ergebnistyp der Methode haben.

Die beiden ersten Punke beziehen sich auf fehlerhafte Programmzustände, die durch Ausnahmen möglichst eingegrenzt werden. ProgrammiererInnen wollen solche Situationen vermeiden; es gelingt ihnen aber nicht immer. Die beiden letzten Punkte beziehen sich auf Situationen, in denen Ausnahmen und Ausnahmebehandlungen von ProgrammiererInnen 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.

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 gänzlich zu vermeiden.

Es gibt aber auch Fälle, in denen der Einsatz von Ausnahmen und deren Behandlungen die Programmlogik wesentlich vereinfachen kann, beispielsweise weil 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 Lesbarbeit am ehesten erhöhen können.

Wir wollen einige Beispiele betrachten, die Grenzfälle für den Einsatz von Ausnahmebehandlungen 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 eine große Anzahl von Iterationen kann die zweite Variante etwas 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 von Ausnahmen 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 (vermutlich ungewollt) schon. Auf solche nicht-lokalen Effekte wird oft vergessen.

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, kann es zu keinen nicht-lokalen Effekten kommen. Nach obrigen Kriterien steht einer derartigen Verwendung von Ausnahmebehandlungen nichts im Wege. Allerdings entspringen beide Varianten einem schlechten Programmierstil: Typabfragen sollen, soweit es möglich ist, vermieden werden. Wenn, wie in diesem Beispiel, nach vielen Untertypen eines gemeinsamen Obertyps unterschieden wird, ist es sinnvoll, dynamisches Binden statt Typabfragen einzusetzen.

Das folgende Beispiel zeigt einen Fall, in dem die Verwendung von Ausnahmen 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 der Methode mit `"Error"` verglichen werden muss, bevor es weiter verwendet werden kann. Wird ein Vergleich vergessen, pflanzt sich der Fehler in andere Programmzweige fort. Wird eine Ausnahme ausgelöst, gibt es dieses Problem nicht:

```
public static String addB (String x, String y)
        throws NoNumberString {
    if (onlyDigits(x) && onlyDigits(y)) {
        ...
    }
    else
        throw new NoNumberString();
}
```

Bei dieser Art des Umgangs mit Fehlern kann sich der Fehler nicht leicht fortpflanzen. Immer dann, wenn ein bestimmter Ergebniswert fehlerhafte Programmzustände anzeigt, ist es ratsam, statt diesem Wert eine Ausnahme zu verwenden. Diese Verwendung von Ausnahmen ist zwar nicht lokal, aber die Verwendung der speziellen Erbegniswerte erzeugt genauso nicht-lokale Abhängigkeiten im Programm.

Kapitel 4

Softwareentwurfsmuster

Nun beschäftigen wir uns mit dem bereits in Abschnitt 1.3 angeschnittenen Thema der Entwurfsmuster (design patterns), die der Wiederverwendung kollektiver Erfahrung dienen. Wir wollen exemplarisch einige häufig verwendete Entwurfsmuster betrachten. Da das Thema der Lehrveranstaltung die objektorientierte Programmierung ist, konzentrieren wir uns dabei auf Implementierungsaspekte und erwähnen andere in der Praxis wichtige Aspekte nur am Rande. Jedem, der sich für Entwurfsmuster in der Software interessiert, sei folgendes Buch empfohlen:

> 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, die ebenso empfehlenswert sind. Auch eine deutsche Übersetzung ist erschienen:

> E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, Bonn, 1996.

Wir betrachten im Skriptum und in der Lehrveranstaltung nur einen kleinen Teil der im Buch beschriebenen und in der Praxis häufig eingesetzten Entwurfsmuster. Wie im Buch gliedern wir die beschriebenen Entwurfsmuster in drei Bereiche: Muster zur Erzeugung neuer Objekte (creational patterns) werden in Abschnitt 4.1 behandelt, jene, die die Struktur der Software beeinflussen (structural patterns) in Abschnitt 4.2, und schließlich jene, die mit dem Verhalten von Objekten zu tun haben (behavioral patterns), in Abschnitt 4.3.

4.1 Erzeugende Entwurfsmuster

Unter den erzeugenden Entwurfsmustern betrachten wir drei recht einfache Beispiele – 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. Konkret wollen wir uns Möglichkeiten zur Erzeugung neuer Objekte vor Augen führen, die über die Verwendung des Operators `new` in Java hinausgehen.

4.1.1 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.

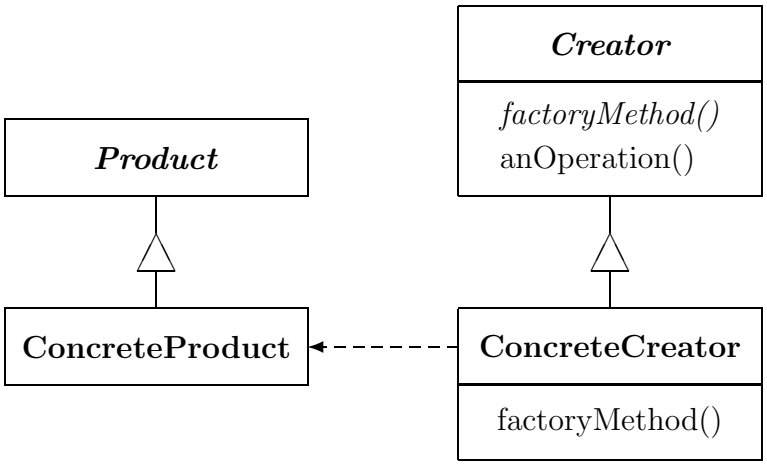
Als Beispiel für eine Anwendung der Factory Method kann man sich ein System zur Verwaltung von Dokumenten unterschiedlicher Arten (Texten, Grafiken, Videos, etc.) vorstellen. Dabei gibt es eine (abstrakte) Klasse mit der Aufgabe, neue Dokumente anzulegen. Nur in einer Unterklasse, der die Art des neuen Dokuments bekannt ist, kann die Erzeugung tatsächlich durchgeführt werden.

Generell ist das Entwurfsmuster anwendbar wenn

- eine Klasse neue Objekte erzeugen soll, deren Klasse aber nicht kennt;
- eine Klasse möchte, dass ihre Unterklassen die Objekte bestimmen, die die Klasse erzeugt;
- Klassen Verantwortlichkeiten an eine von mehreren Unterklassen delegieren, und man das Wissen, an welche Unterklasse delegiert wird, lokal halten möchte.

Die Struktur dieses Entwurfsmusters sieht wie in der folgenden Grafik aus. Wir werden in solchen Grafiken Klassen als Kästchen darstellen, die die Namen der Klassen in Fettschrift enthalten. Durch einen waagrechten Strich getrennt können auch Namen von Methoden (mit einer Parmeterliste) und Variablen (ohne Parameterliste) in den Klassen in normaler Schrift angegeben sein. Namen von abstrakten Klassen und Methoden sind kursiv dargestellt; konkrete Klassen und Methoden sind nicht kursiv. Unterklassen sind mit deren Oberklassen durch Striche und Dreiecke,

dessen 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 eine Instanz 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. Sie haben keine inhaltliche Bedeutung.



Die (oft abstrakte) Klasse „Product“ ist (wie „Dokument“ 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 Instanzen 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. Dies kann unter anderem bei kovari-

anten Problemen hilfreich sein. Beispielsweise erzeugt eine Methode `generiereFutter` in der Klasse `Tier` nicht direkt Futter einer bestimmten Art, sondern liefert in der Unterklasse `Rind` eine neue Instanz von `Gras` und in `Tiger` eine von `Fleisch` zurück.

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.

Folgendes Beispiel zeigt eine Anwendung von Factory Methods mit *lazy initialization*:

```
abstract class Creator {
    private Product product = null;
    protected abstract Product createProduct();
    public Product getProduct() {
        if (product == null)
            product = createProduct();
        return product;
    }
}
```

Eine neue Instanz des Produkts 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. Generizität könnte dafür einen Ausweg bieten:

```
class GenCreator<P extends Product> extends Creator {
    public Product createProduct() {
        return new P();
    }
}
```

Leider ist dieses Beispiel kein legaler GJ-Code: Nach `new` darf kein Typparameter stehen. Die Übersetzung von `StandardCreator` liefert daher eine Fehlermeldung. In anderen Sprachen wie beispielsweise C++, in denen für Generizität eine heterogene Übersetzung erfolgt, funktioniert dieser Ansatz aber problemlos.

4.1.2 Prototype

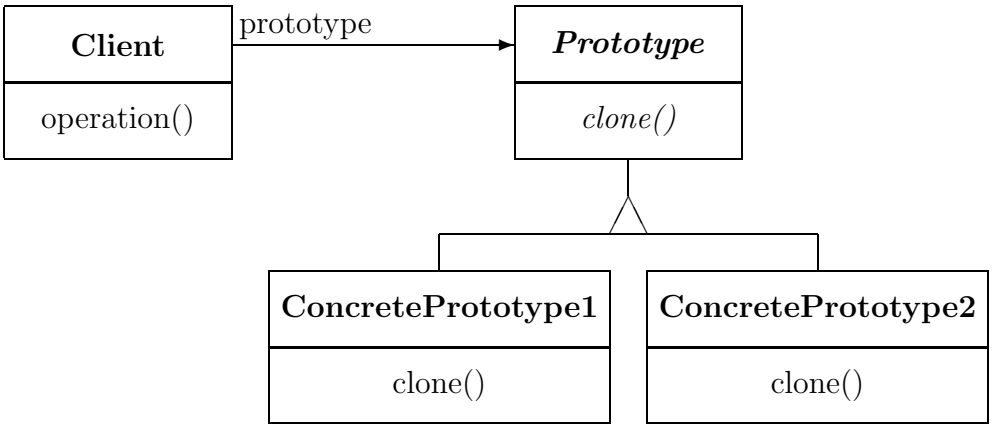
Das Entwurfsmuster *Prototype* dient dazu, die Art eines neu zu erzeugenden Objekts durch ein Prototyp-Objekt zu spezifizieren; neue Objekte werden durch Kopieren dieses Prototyps erzeugt.

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 sein. Das neue Polygon kann etwa durch Ändern seiner Größe oder Position einen vom kopierten Polygon verschiedenen Zustand erhalten.

Generell ist dieses Entwurfsmuster anwendbar, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengestzt und dargestellt werden, und wenn

- die Klassen, von denen Instanzen erzeugt werden sollen, erst zur Laufzeit bekannt sind (beispielsweise wegen dynamischem Laden), oder
- vermieden werden soll, eine Hierarchie von „Creator“-Klassen zu erzeugen, die einer parallelen Hierarchie von „Product“-Klassen entspricht (Factory Method), oder
- jede Instanz einer Klasse nur einen von wenigen unterschiedlichen Zuständen annehmen kann; es kann einfacher sein, für jeden möglichen Zustand einen Prototyp zu erzeugen und diese Prototypen zu kopieren, als Instanzen manuell durch `new` zu erzeugen und jedesmal den passenden Zustand anzugeben.

Die Struktur des Entwurfsmusters entspricht folgender Grafik. Dabei bedeutet ein durchgezogener Pfeil, dass jede Instanz der Klasse, von der der Pfeil ausgeht, auf eine Instanz der Klasse (oder des Typs), 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. 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 (clients) und reduzieren damit die Anzahl der Klassen, die Anwender kennen müssen. Die Anwender brauchen nicht geändert werden, wenn neue Produktklassen dazu kommen oder geändert werden.
- Prototypen können auch zur Laufzeit jederzeit dazu gegeben und weg genommen 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 statt durch die Definition neuer Klassen erzeugt werden, beispielsweise durch die Spezifikation von Werten in Objektvariablen. Die Erzeugung einer Kopie eines Objekts ähnelt der Erzeugung einer Klasseninstanz. Der Zustand eines Prototyps kann sich aber jederzeit ändern, während Klassen zur Laufzeit unveränderlich sind.
- Sie vermeiden übertrieben große Anzahlen an Unterklassen. Im Gegensatz zu Factory Methods ist es nicht nötig, parallele Klassenhierarchien zu erzeugen.

- Sie erlauben die dynamische Konfiguration von Programmen. In Sprachen 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 immer zu einem Programmabbruch aus Speichermangel führt. Wie solche zyklischen Referenzen aufgelöst werden sollen, hängt im Wesentlichen von der Anwendung ab.

ProgrammiererInnen können kaum den Überblick über ein System behalten, das viele Prototypen enthält. Das gilt vor allem dann, wenn Prototypen zur Laufzeit dazu oder weg kommen. Zur Lösung dieses Problems haben sich *Prototyp-Manager* bewährt. Das sind assoziative Datenstrukturen (kleine Datenbanken), in denen nach geeigneten Prototypen gesucht werden kann.

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. Daher ist es oft nötig, dass die Klassen spezielle Methoden zur Initialisierung beziehungsweise zum Ändern des Zustands bereit stellen.

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* fest eingebaut und bildet dort die einzige Möglichkeit zur Erzeugung neuer Instanzen; es gibt keine Klassen, sondern nur Objekte, die als Prototypen verwendbar sind.

4.1.3 Singleton

Das Entwurfsmuster *Singleton* sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globalen Zugriff auf diese Instanz. Wir haben Singleton als Beispiel zur Erklärung von Entwurfsmustern in Abschnitt 1.3 verwendet.

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 Instanzen der Klasse erzeugt werden. Es ist besser, die Klasse selbst für die Verwaltung ihrer einzigen Instanz verantwortlich zu machen. Das ist die Aufgabe eines Singleton.

Dieses Entwurfsmuster ist anwendbar wenn

- es genau eine Instanz einer Klasse geben soll; diese soll global zugreifbar sein;
- die Klasse durch Vererbung erweiterbar sein soll, und Anwender die erweiterte Klasse ohne Änderungen verwenden können sollen.

Aufgrund der Einfachheit dieses Entwurfsmusters verzichten wir auf eine grafische Darstellung. Ein Singleton besteht nur aus einer gleichnamigen Klasse mit einer statischen Methode „instance“, welche die einzige Instanz der Klasse zurück gibt.

Singletons haben unter Anderem folgende Eigenschaften:

- Sie erlauben den kontrollierten Zugriff zur einzigen Instanz.
- Sie vermeiden unnötige Namen im System, da auf globale Variablen verzichtet wird.
- Sie unterstützen Vererbung.
- Sie erlauben auch mehrere Instanzen. Man kann die Entscheidung zugunsten nur einer Instanz im System jederzeit ändern und auch die

Erzeugung mehrerer Instanzen ermöglichen. Die Klasse hat weiterhin vollständige Kontrolle darüber, wie viele Instanzen erzeugt werden.

- Sie sind flexibler als statische Methoden, da statische Methoden kaum Änderungen erlauben und dynamisches Binden nicht unterstützen.

Einfache Implementierungen dieses Entwurfsmusters sind recht einfach, wie folgendes Beispiel zeigt:

```
class Singleton {
    private static Singleton singleton = null;
    protected Singleton() {}
    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 ProgrammiererInnen eine Auswahl zwischen den Alternativen treffen können.

Überraschenderweise ist die Implementierung eines solchen Singletons gar nicht einfach. Die folgende Lösung ist noch am einfachsten:

```
class Singleton {
    private static Singleton singleton = null;
    public static int kind = 0;
    protected Singleton() { ... }
    public static Singleton instance() {
        if (singleton == null)
            switch (kind) {
            case 1:  singleton = new SingletonA(); break
            case 2:  singleton = new SingletonB(); break
            default: singleton = new Singleton();
            }
        return singleton;
    }
}
```

```
class SingletonA extends Singleton {
    SingletonA() { ... }
}
class SingletonB extends Singleton {
    SingletonB() { ... }
}
```

Durch Zuweisung eines Wertes an die Klassenvariable `kind` können ProgrammiererInnen vor dem ersten Aufruf von `instance` bestimmen, welche Alternative gewählt werden soll.

Um die feste Verdrahtung der Alternativen in `Singleton` zu vermeiden, kann man die Implementierung von `instance` in den Untertypen von `Singleton` durchführen:

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

ProgrammiererInnen können einfach zwischen den Alternativen wählen, indem Sie den ersten Aufruf von `singleton` im entsprechenden Untertyp von `Singleton` durchführen. Alle weiteren Aufrufe von `instance` geben stets das im ersten Aufruf erzeugte Objekt zurück. Allerdings ist nicht mehr die Klasse `Singleton` alleine für die Existenz nur einer Instanz 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.

4.2 Strukturelle Entwurfsmuster

Wir wollen zwei einfache Vertreter der strukturellen Entwurfsmuster betrachten, die man häufig braucht. Diese Muster haben eine ähnliche Struktur, aber unterschiedliche Verwendungen.

4.2.1 Decorator

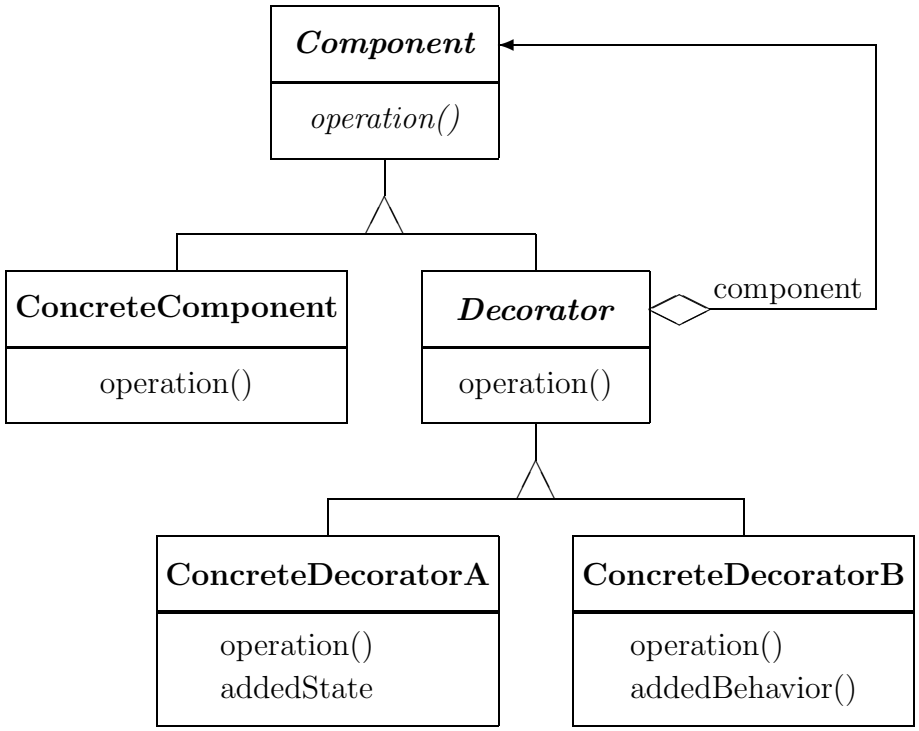
Das Entwurfsmuster *Decorator*, auch *Wrapper* genannt, gibt Objekten dynamisch zusätzliche Verantwortlichkeiten. Decorators stellen eine flexible Alternative zur Vererbung bereit.

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 dazu kommt und auch wieder weg genommen wird.

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.

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“ definiert eine Schnittstelle für Objekte, an die Verantwortlichkeiten dynamisch hinzugefügt werden können. Die Klasse „ConcreteComponent“ ist, wie beispielsweise die Klasse eines Fensters, eine konkrete Unterklasse davon. Die (abstrakte) Klasse „Decorator“ definiert eine Schnittstelle für Verantwortlichkeiten, die dynamisch zu Komponenten hinzugefügt werden können. Jede Instanz dieses Typs enthält eine Referenz namens „component“ auf eine Instanz 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 bereits in „Component“ definiert ist, aufgerufen, so wird dieser Aufruf einfach an das Objekt, das über „component“ referenziert ist, weitergegeben.

Decorators haben einige positive und negative Eigenschaften:

- Sie bieten mehr Flexibilität als statische Vererbung. Wie bei statischer Erweiterung einer Klasse durch Vererbung werden Verantwort-

lichkeiten 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 Eigenschaften (features) 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.

- Instanzen von „Decorator“ und die dazugehörenden Instanzen 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 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 unnötig umfangreich und ineffizient.

Dekoratoren eignen sich gut dazu, die Oberfläche beziehungsweise das Erscheinungsbild eines Objekts zu erweitern. Sie sind nicht gut dafür geeignet, inhaltliche Erweiterungen von Objekten vorzunehmen. 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.

4.2.2 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, das einen Text darstellt, der irgendwo im Internet verfügbar ist. Wenn das Textobjekt erzeugt wird, braucht nicht gleich auf das Eintreffen der Daten aus dem Netzwerk gewartet werden, sondern erst dann, wenn die Daten wirklich benötigt werden. Statt dem eigentlichen Textobjekt verwendet man in der Zwischenzeit einen Platzhalter, der erst bei Eintreffen der Daten durch das eigentliche Objekt ersetzt wird. Falls nie oder erst viel später auf die Daten zugegriffen wird, erspart man sich das Warten auf die Daten.

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.

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:

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 kompexere 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.

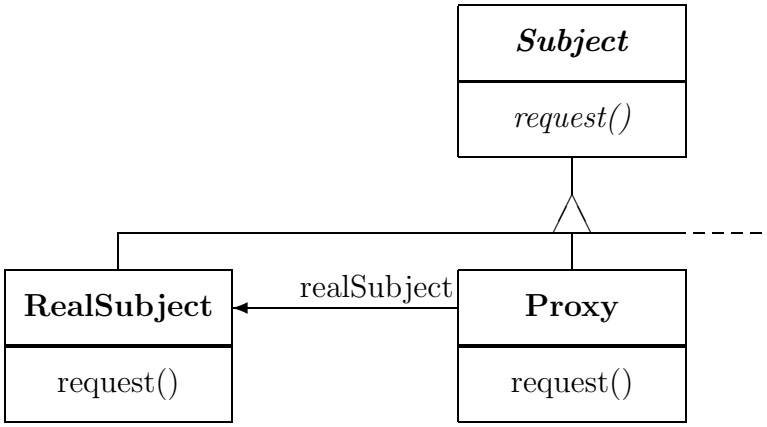
Protection Proxies kontrollieren Zugriffe auf Objekte. Solche Proxies sind sinnvoll, wenn Objekte je nach Zugreifer oder Situation unterschiedliche Zugriffsrechte haben sollen.

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;
- das Zusichern, dass während des Zugriffs auf das Objekt kein gleichzeitiger Zugriff durch einen anderen Thread erfolgt.

Wie das eingangs gebrachte Beispiel zeigt, gibt es zahlreiche weitere Einsatzmöglichkeiten.

Die Struktur dieses Entwurfsmusters ist recht einfach:



Die abstrakte Klasse oder das Interface „Subject“ definiert die gemeinsame Schnittstelle für Instanzen von „RealSubject“ und „Proxy“. Instanzen von „RealSubject“ und „Proxy“ können gleichermaßen verwendet werden, wo eine Instanz 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 Instanzen von „RealSubject“ (oder auch andere Instanzen 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 Instanzen 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 Instanzen 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.

4.3 Entwurfsmuster für Verhalten

Zwei Beispiele zu Entwurfsmustern für Verhalten, nämlich Iterator und Visitor, haben wir bereits in Kapitel 3 beschrieben. Hier wollen wir nur einige ergänzende Bemerkungen zu Iteratoren machen. Ein weiteres Entwurfsmuster, nämlich Template Method soll dazu anregen, beim Entwerfen und Programmieren von Software der eigenen Fantasie freien Lauf zu lassen und auch dort Möglichkeiten für die Wiederverwendung von Programmcode zu finden, wo es keine spezielle Unterstützung durch eine Programmiersprache gibt.

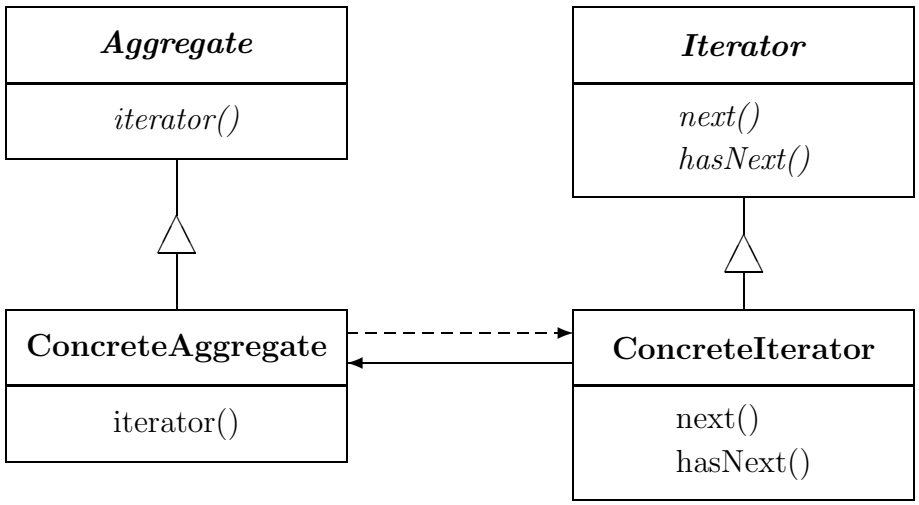
4.3.1 Iterator

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 offen zu legen.

Dieses Entwurfsmuster ist verwendbar um

- auf den Inhalt eines Aggregats zugreifen zu können, ohne die innere Darstellung offen legen zu müssen;
- mehrere 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.

Das Entwurfsmuster hat folgende Struktur:



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 eines Aggregats. Die abstrakte Klasse oder das Interface „Aggregate“ definiert eine Schnittstelle für die Erzeugung eines neuen Iterators. Die Klasse „ConcreteAggragate“ implementiert diese Schnittstelle. Ein Aufruf von „iterator“ erzeugt eine neue Instanz von „ConcreteIterator“, was durch den strichlierten Pfeil angedeutet ist. Um die aktuelle Position im Aggregat verwalten zu können, braucht jede Instanz von „ConcreteIterator“ eine Referenz auf die entsprechende Instanz von „ConcreteAggragate“, 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 Abarbeitungsreihenfolgen zu implementieren.

- Iteratoren vereinfachen die Schnittstelle von „Aggregate“, da Zugriffsmöglichkeiten, die über Iteratoren bereit gestellt werden, durch die Schnittstelle von „Aggregate“ nicht unterstützt werden müssen.
- Auf ein und demselben Aggregat können gleichzeitig mehrere Abarbeitungen stattfinden, da jeder Iterator selbst den aktuellen Abarbeitungszustand verwaltet.

Es gibt zahlreiche Möglichkeiten zur Implementierung von Iteratoren. Beispiele dafür haben wir bereits gesehen. 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.

- 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.
- 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 zu aufwendig. Ein *robuster Iterator* erreicht dasselbe Ziel ohne das ganze Aggregat zu kopieren. Es ist recht aufwendig, robuste Iteratoren zu schreiben. Einen allgemeinen Ansatz dazu gibt es nicht, da die Detailprobleme stark von der Art des Aggregats abhängen.
- Tatsächlich sinnvoll sind Iteratoren nur auf Aggregaten, die Elemente enthalten. Aus Gründen der Allgemeinheit ist es oft praktisch, Iteratoren auch auf leeren Aggregaten bereitzustellen. In einer Anwendung braucht man die Schleife dann nur so lange ausführen, so lange es Elemente gibt – bei leeren Aggregaten daher nie – ohne eine eigene Behandlung für den Spezialfall zu brauchen.

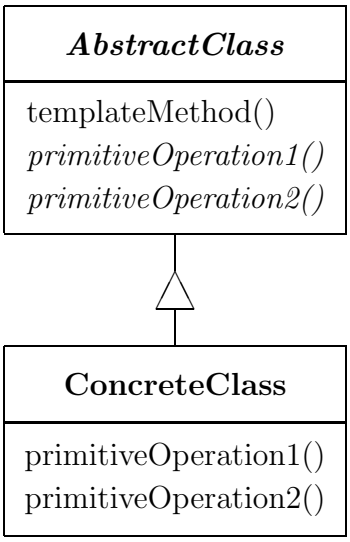
4.3.2 Template Method

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 ändern zu müssen.

Dieses Entwurfsmuster ist anwendbar

- um den unveränderlichen Teil eines Algorithmus 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 (siehe weiter unten).

Die Struktur dieses Entwurfsmusters ist recht einfach:



Die (meist abstrakte) Klasse „AbstactClass“ definiert (abstrakte) primitive Operationen, welche konkrete Unterklassen als Schritte in einem Algorithmus implementieren, 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. Sie sind vor allem in Klassenbibliotheken 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 umgekehrt.
- Sie rufen oft nur eine von mehreren Arten von Operationen auf:
  - konkrete Operationen (entweder in „ConcreteClass" oder in der Klasse, in der die Template Methods angewandt werden);
  - 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 die SchreiberInnen von Unterklassen wissen, welche Operationen dafür vorgesehen sind, überschrieben zu werden. Alle Operationen, bei denen es Sinn macht, dass sie in Unterklassen überschrieben werden, 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 als `final` deklariert 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, um so komplizierter wird die direkte Wiederverwendung von „AbstractClass".