# Einführung in die Informatik 2 für Ingenieure (MSE)

## Teil 1

Objektorientierte Modellierung (in UML)
und Programmierung (in Java)

## Teil 2

Datenbanksysteme eine Einführung
Alfons Kemper und Andre Eickler
Oldenburg Verlag, 10. Auflage, 2015

Prof. Alfons Kemper, Ph. D.
Wolf Rödiger, M.Sc.
Christoph Anneser, M.Sc.
Aaron Tacke
Institut für Informatik – Lehrstuhl III (I3)
Technische Universität München
Boltzmannstr. 3
D-85748 Garching bei München

25. Juli 2022

# Inhaltsverzeichnis

# 1. Objekte und Klassen

In diesem Kapitel werden wir die grundlegendsten Konzepte der objektorientierten Modellierung und Programmierung besprechen. Eines der wesentlichen Themen dieser Einführung ist die klare Differenzierung zwischen *Werten* und *Objekten*, sowie zwischen *primitiven Typen* und *Objekttypen*.

## 1.1 Unterscheidung zwischen Werten und Objekten

Wir unterscheiden zwischen *Objekten* und *Werten*: Erstere sind Elemente von *Objekttypen* (engl. *class*), letztere sind Elemente von *primitiven Typen*. *Objekte* repräsentieren komplexe Gegenstände (Entitäten) und sind selbst wieder aus Werten und/oder anderen Objekten „zusammengebaut“. Die grundlegendste Unterscheidung zwischen Objekten und Werten bezieht sich auf deren Zustand (engl. *state*): Objekte haben einen internen Zustand, der sich (möglicherweise) ändern kann. Demgegenüber sind Werte, wie beispielsweise der Zahlenwert 10, nicht änderbar. Werte kann man somit als „Gott-gegebene“, ewig invariante Gegenstände ansehen.

Andererseits werden Objekte von den Benutzern/Programmierern erzeugt. Sie werden von vorher definierten Objekttypen (Klassen) generiert – durch die sogenannte Instanziierung. Der Objekttyp spezifiziert die *strukturellen* und *verhaltensmäßigen* Charakteristika seiner Instanzen. Die individuellen Objekte können dann durch explizite Instanziierung des Objekttyps gebildet werden. Dazu dient in Java der *new*-Operator.

Das nachfolgende Diagramm fasst nochmals die in diesem Buch verwendete Terminologie zusammen:

| **Typen** | | **Instanzen** |
|---|---|---|
| primitive Typen | $\longrightarrow$ | Werte |
| Objekttypen (Klassen) | $\longrightarrow$ | Objekte |

Wenn wir nicht zwischen primitiven Typen und Objekttypen unterscheiden wollen, sprechen wir einfach von Typen. Wir sagen beispielsweise, dass eine Variable auf einen bestimmten Typ eingeschränkt ist – sei es ein primitiver Typ oder ein Objekttyp.

In Java gibt es eine ganze Reihe von vordefinierten primitiven Typen, unter anderem:

| **Sorte** | **Wertebereich** |
|---|---|
| **boolean** | {*true, false*} |
| **byte** | sehr kleine Integer-Zahlen |
| **short** | kleine Integer-Zahlen |
| **int** | Integer-Zahlen |
| **long** | große Integer-Zahlen |
| **float** | Fließkomma-Zahlen |
| **double** | Fließkomma-Zahlen doppelter Präzision |
| **char** | Character |

```
[public] [abstract] class A
   [extends B] [implements Schnittstellen] {
      Instanz-Variable;
      ...
      Instanz-Variable;

      Konstruktor;
      ...
      Konstruktor;

      Operation/Methode;
      ...
      Operation/Methode;
}
```

Abbildung 1.1: Syntax der Java-Klassendefinition

Diese vordefinierten Typen können in Anwendungsprogrammen ohne weitere Definition verwendet werden. Genau genommen kann man in Java gar keine primitiven Typen selber definieren; man ist also auf die vordefinierten primitiven Typen beschränkt.

## 1.2 Objekttypen

Wie bereits erwähnt werden ähnliche Objekte durch einen gemeinsamen Objekttyp (eine Klasse) modelliert. In dieser Hinsicht wird also eine Klasse verwendet, um ähnliche Objekte zu gruppieren und deren strukturelle und verhaltensmäßige Charakteristika zu modellieren. Basierend auf den vordefinierten primitiven Typen kann man beliebig komplex strukturierte Objekttypen definieren.

Die syntaktische Struktur einer Java-Klassendefinition ist in Abbildung 1.1 gezeigt. Die von eckigen Klammern $[\ldots]$ umschlossenen Teile sind optional und können somit auch fehlen. Ohne jetzt schon alle Details der Klassendefinition erklären zu wollen (das wird in nachfolgenden Abschnitten getan), wollen wir doch die wesentlichen Bestandteile kurz ansprechen.

Der Name der Klasse (hier: *A*) muss eindeutig sein, d.h., es darf keine weitere Klasse mit demselben Namen (innerhalb desselben Packages) geben. Ein Objekttyp hat genau einen Obertyp (hier: *B*). Wenn dieser in der *extends*-Klausel nicht angegeben ist, wird standardmäßig ein vordefinierter Objekttyp namens *Object*[1] angenommen. Die Klasse *B* wird in diesem Fall *Obertyp* (engl. *super type*) bezeichnet; konsequenterweise wird die Klasse *A* als Untertyp (engl. *sub type*) bezeichnet. Der Untertyp erbt alle Eigenschaften – also die strukturelle und die verhaltensmäßige Beschreibung – des Obertyps. In dieser Hinsicht unterstützt Java das Konzept der *einfachen Vererbung*, da eine Klasse von genau einer anderen Klasse erbt. Manchmal findet man auch die Terminologie, dass der Untertyp vom Obertyp abgeleitet wird – oder dass der Untertyp den Obertyp erweitert (wie in der Java-Syntax mit dem Schlüsselwort *extends* angedeutet). Ein Objekttyp kann zusätzlich mehrere Schnittstellen (engl. *interfaces*) implementieren. Unter einer Schnittstelle versteht man eine Menge von Operations-Signaturen, die dann in den implementierenden

[1] Präzise heißt diese Klasse *java.lang.object*.

Klassen realisiert werden müssen.

Nach Konvention wird die strukturelle Repräsentation der Objekte als erstes modelliert. Die Struktur der Objekte wird durch eine Menge von Instanzvariablen (oder Attributen) beschrieben. Die aktuellen Werte dieser Variablen repräsentieren den Zustand eines Objekts, der sich natürlich durch Anwendung der Operationen (oder durch explizite Zuweisung) verändern kann.

Als nächstes werden die Konstruktoren aufgeführt. Hierbei handelt es sich um spezielle Initialisierungs-Operationen, die bei der Instanziierung eines Objekts ausgeführt werden. Die Konstruktoren haben immer denselben Namen wie die Klasse, hier also *A()*. Wenn es mehr als einen Konstruktor gibt, unterscheiden sie sich in der Anzahl oder den Typen der Parameter.

Den Abschluss der Klassendefinition bilden die Operationen (manchmal auch Methoden genannt). Die Operationen dienen dem Benutzer – oft auch als *Klienten* bezeichnet – des Objekttyps, Instanzen desselben zu bearbeiten bzw. deren Zustand abzufragen. Somit repräsentieren die Methoden/Operationen die Schnittstelle des Objekttyps nach außen. Auf diese Art wird das Konzept der Verkapselung in Java realisiert; d.h., Objekte werden über die den Objekttypen zugeordneten Operationen bearbeitet, ohne dass man als Klient Zugriff auf die interne Repräsentation haben muss.

Die Definition einer Operation besteht aus der abstrakten Signatur und, falls dies notwendig ist, der Implementierung. Die Signatur legt dabei die Aufrufstruktur der Operation – den Namen, die Parameter-Typen sowie den Rückgabe-Typ – fest. Im Implementierungsteil wird der Java-Programmcode der Operation angegeben. Eine saubere, gut durchdachte, objektorientierte Modellierung zeichnet sich in der Regel dadurch aus, dass die einzelnen Operationen sehr einfach und übersichtlich implementiert werden können.

Aus dieser Übersicht der Klassendefinition wird deutlich, dass ein Objekttyp aus zwei sich gegenseitig ergänzenden Dimensionen besteht:

1. *Strukturelle Repräsentation*
   Die strukturelle Repräsentation bestimmt die interne Struktur der Objekte des jeweiligen Objekttyps. Die strukturelle Repräsentation dient dazu, den Zustand eines Objekts zu modellieren. Dieser Zustand kann sich natürlich durch Anwendung entsprechender Operationen im Verlauf der Lebensdauer eines Objekts ändern. Die strukturelle Repräsentation ergibt sich aus der Menge der Instanzvariablen, die im Objekttyp definiert wurden. Zusätzlich erbt ein Objekttyp noch die strukturelle Repräsentation seines Obertyps – also des Objekttyps, der in der *extends*-Klausel angegeben wurde.

2. *Verhaltens-Spezifikation*
   Das „Verhalten“ eines Objekts ergibt sich aus den Operationen, die auf Objekten des jeweiligen Objekttyps ausgeführt werden können. Präziser gesagt sind dies alle öffentlichen (*public*) Operationen, die im Objekttyp definiert wurden. Hierzu zählen auch die Operationen der Schnittstelle(n), die ein Objekttyp implementiert – angegeben in der *implements*-Klausel. Weiterhin kommen die geerbten Operationen des Obertyps hinzu. Operationen bestehen aus der Signatur, die festlegt, wie die Operation aufgerufen werden kann und welchen Ergebnistyp sie zurückgibt, sowie einer Implementierung. Geerbte Operationen müssen nicht (notwendigerweise) nochmals implementiert werden, da auch die Implementierung vom Obertyp geerbt wird.

Zunächst wollen wir uns auf die strukturelle Dimension der Objekttypen konzentrieren – auch wenn die beiden Dimensionen (Struktur und Verhalten) letztendlich eng miteinander zusammenhängen. Wir werden deshalb auch hier schon elementare Operation zum Lesen und Schreiben von Instanzvariablen in diesem Kontext behandeln.

## 1.3 Einfach Strukturierte Objekt-Typen: Datensätze/Records/Tupel

Datensatzorientierte Objekttypen sind einfach strukturiert und bestehen aus einer Anzahl von Instanzvariablen (Attributen). Die Klassendefinition sieht dann folgendermaßen aus:

**class** *TypName* {
    $Typ_1$ $Attr_1$ ;
    ...;
    $Typ_n$ $Attr_n$ ;
  // **Operationen** folgen hier
    ...
} // **end class** *TypName*;

Die Attributnamen $Attr_i$ für $(1 \leq i \leq n)$ müssen paarweise unterschiedlich sein; die Typnamen $Typ_i$ können mehrfach vorkommen. Der Typ $Typ_i$ muss entweder ein vordefinierter primitiver Typ (Sorte) oder ein Objekttyp (Klasse) sein. Man sagt dann, dass das Attribut $Attr_i$ auf den Typ $Typ_i$ eingeschränkt ist – oder salopp, dass $Attr_i$ vom Typ $Typ_i$ ist. Der Einfachheit halber erlaubt Java auch, dass mehrere Attribute vom gleichen Typ hintereinander aufgeführt werden; also beispielsweise `int i,j;`.

Es gibt keinerlei Restriktionen hinsichtlich komplexer rekursiver Objektstrukturen. Es ist sogar möglich, dass ein Attribut auf die eigene Klasse eingeschränkt wird. Ein anschauliches Beispiel dafür ist der Objekttyp *Person* mit einem Attribut *ehePartner*, das selbst auf den Objekttyp *Person* eingeschränkt ist:

```
class Person {
    public String name;
    public int alter;
    public Person ehePartner;
}
```

Dieser Objekttyp definiert also Personen-Objekte, die mittels der folgenden drei Attribute strukturell beschrieben sind: *name* ist auf den Objekttyp *String*, bei dem es sich um einen vordefinierten Java-Objekttyp handelt, eingeschränkt; *alter* ist vom Typ *int* und *ehePartner* ist auf den Objekttyp *Person* eingeschränkt.

Die Objekttyp-Designer können entscheiden, ob die Attribute den Klienten des Typs sichtbar gemacht werden oder nicht. Allgemein sichtbare Attribute werden als *public* definiert; anderenfalls werden sie als *private* oder *protected* markiert. Die Unterscheidung zwischen *private* und *protected* bezieht sich darauf, ob das betreffende Attribut im *Untertyp* verborgen oder sichtbar ist – darauf gehen wir aber in Kapitel 3 nochmals ein.

Wir wollen nun mit der Definition von Objekttypen für die geometrische Modellierung beginnen. Als Grundlage definieren wir den Objekttyp *Vertex*. Objekte dieses Typs werden strukturell durch die $x$-, $y$- und $z$-Koordinate beschrieben. Konsequenterweise hat der Objekttyp die drei Attribute $x$, $y$ und $z$, die alle auf den Typ *double* eingeschränkt sind.

```
class Vertex {
    public double x;
    public double y;
    public double z;
}
```

Um mit diesem Beispiel fortzufahren, wird als nächstes der sehr einfach strukturierte Objekttyp *Material* definiert. *Material*-Objekte werden durch die zwei Instanzvariablen *name* vom Typ *String* sowie *spezGewicht* vom Typ *double* repräsentiert. Die erstgenannte Variable gibt den üblichen Namen eines Materials, also etwa *Eisen* oder *Kupfer* an. Die Variable *spezGewicht* gibt für das jeweilige Material das spezifische Gewicht an. Die Objekttyp-Definition sieht dann wie folgt aus:

```
public class Material {
    public String name;
    public double spezGewicht;
}
```

Man beachte, dass wir bewusst die Attribute sichtbar gemacht haben. D.h. bei diesen Objekten gibt es keinerlei Verkapselung (engl. *information hiding*), da die Klienten direkt auf die Attribute zugreifen können. Wir werden den Sinn (bzw. den Nutzen) der Verkapselung später detaillierter diskutieren.

Wir sind jetzt soweit, dass wir einen etwas komplexer strukturierten Objekttyp definieren können: den *Quader*. Die Repräsentation eines Quaders ergibt sich aus den acht Eckpunkten. Dies ist in Abbildung 1.2 gezeigt. Es ist wichtig, dass die topologische Anordnung der acht Eckpunkte gemäß der Abbildung beibehalten wird, damit keine Inkonsistenzen auftreten. Diese Anordnung wird nämlich später bei der Realisierung der Operationen, wie beispielsweise *hoehe* und *breite* vorausgesetzt. Die Objekttyp-Definition des Quaders sieht wie folgt aus:

```
class Quader {
    public Vertex v1, v2, v3, v4, v5, v6, v7, v8;
    public Material mat;
    public double wert;
}
```

Ein *Quader*-Objekt besteht also aus den Variablen *v1*, *v2*, *v3*, *v4*, *v5*, *v6*, *v7* und *v8*, die alle auf den Objekttyp *Vertex* eingeschränkt sind. Diese acht Vertex-Objekte repräsentieren die acht Eckpunkte des Quaders im drei-dimensionalen Raum. Weiterhin gibt es die Instanzvariable *mat*, die auf den Objekttyp *Material* eingeschränkt ist, sowie eine *double*-Instanzvariable *wert*, die den (monetären) Wert des jeweiligen Quaders repräsentiert.

## 1.4 UML-Notation

UML (Unified Modeling Language) ist im Software Engineering „die“ standardisierte Modellierungssprache. Es gibt dafür mittlerweile auch vielfältige Werkzeuge, die insbesondere für die graphische Darstellung des objektorientierten Entwurfs genutzt werden. Einige dieser Computer-Werkzeuge gehen sogar so weit, dass sie aus dem graphischen Entwurf automatisch das Gerippe der Klassendefinitionen in verschiedenen Programmiersprachen wie Java, C++, Eiffel und C# generieren können.

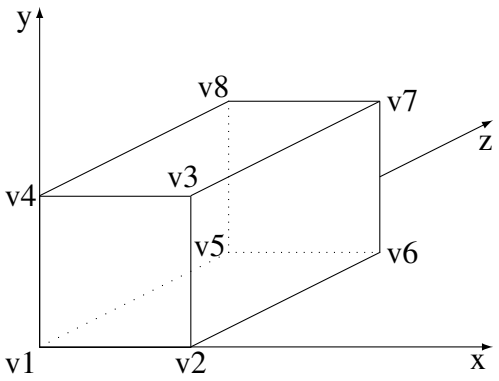



Abbildung 1.2: Die topologische Repräsentation eines *Quader*-Objekts

Die strukturelle UML-Repräsentation der drei Objekttypen *Vertex*, *Material* und *Quader* ist in Abbildung 1.3 gezeigt. In UML werden Klassen durch dreigeteilte Rechtecke beschrieben: Zunächst wird der Name der Klasse angegeben, dann folgen die Attribute und zuletzt die Operationen, die wir aber erst im nächsten Kapitel einführen werden.

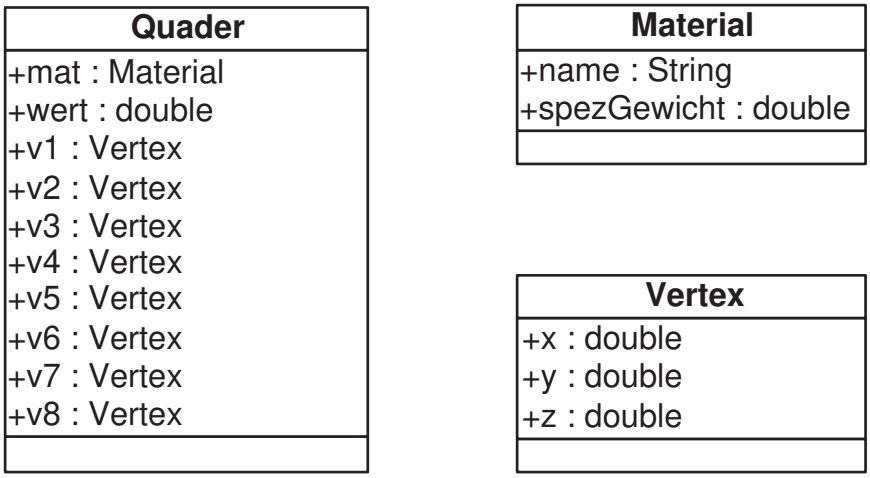



Abbildung 1.3: Die UML-Beschreibung der Klassen Quader, Vertex und Material

## 1.5 Instanziierung von Objekten

Objekte werden durch *Instanziierung* eines Objekttyps „geboren“. Deshalb werden die Objekte oft auch *Instanzen* des Objekttyps (der Klasse) genannt. In diesem Sinn kann man einen Objekttyp auch als Schablone auffassen, mit der man beliebig viele gleichartige Objekte kreieren kann. Ein weiteres anschauliches Bild ist das der *Plätzchenform* (= Objekttyp), mit der man beliebig viele gleich geformte *Plätzchen* (= Instanzen) ausstanzen kann.

Für die Instanziierung gibt es in Java den eingebauten Operator *new*, der auf einen Objekttyp angewendet wird, um ein neues Objekt diesen Typs zu generieren. In seiner

einfachsten Form[2] wird dadurch das Skelett eines Objekts, wie es in der strukturellen Repräsentation festgelegt wurde, gebildet. Die Instanzvariablen werden dabei wie folgt initialisiert:

- *int, short, byte, long*-Attribute werden auf den Wert 0 initialisiert.
- *double, float*-Attribute werden initial auf den Wert 0.0 gesetzt.
- *char*-Instanzvariablen werden auf den Wert „$\backslash u0000$“ initialisiert.
- *boolean*-Attribute werden auf *false* gesetzt.
- Attribute, die auf einen Objekttyp eingeschränkt sind, werden auf *null* gesetzt. Dies ist ein spezieller Wert, der angibt, dass (noch) keine Referenz auf ein Objekt existiert.
- *String*-Attribute werden initial auch auf *null* gesetzt, da Strings in Java als Objekttyp definiert sind.

Schon an dieser Stelle sollte darauf hingewiesen werden, dass Java bei lokalen Variablen (anders als bei Instanzvariablen) keine implizite Initialisierung durchführt. Diese muss von den Programmieren explizit erfolgen, beispielsweise durch:

```
int someInt = 0;
```

Im folgenden Programmfragment erzeugen wir eine neue Instanz der Klasse *Quader* mit dem *new* Operator:

```
Quader meinQuader;
meinQuader = new Quader();
```

Im obigen Programmfragment wurde zunächst die lokale Variable *meinQuader* vom Typ *Quader* eingeführt. In Java müssen Variablen generell vorab deklariert werden, d.h. ihr Typ muss festgelegt werden. Einer Variablen kann man dann nur Werte bzw. Objekte diesen Typs zuordnen (später werden wir sehen, dass auch Objekte eines Untertyps zugeordnet werden können).

Im obigen Programmfragment wurde ein neues *Quader*-Objekt durch *new Quader()* instanziiert und der Variablen *meinQuader* zugeordnet.

Der initiale Zustand des neu generierten Quaders ist in Abbildung 1.4 dargestellt. Es ist offensichtlich, dass der vordefinierte *new*-Operator – in Verbindung mit den Default-Konstruktoren – nur die leere „Hülle“ eines Objekts gemäß der strukturellen Repräsentation des Objekttyps generiert. Weiterhin erkennen wir, dass dem Objekt ein eindeutiger *Objektidentifikator* (OID) zugeordnet ist, den wir in unserer Darstellung als $id_1$ bezeichnet haben. Wir werden auf diesen sehr wichtigen Aspekt der Objektidentifikation später noch detaillierter eingehen – siehe Abschnitt 1.6.

Wir wollen jetzt unser kleines Beispiel dahingehend vervollständigen, dass wir den Quader $id_1$ auf den die Variable *meinQuader* verweist, zum Einheitswürfel initialisieren.

[2] Also ohne zusätzliche implizite Konstruktor-Anwendung zur Initialisierung, die wir erst später einführen werden.

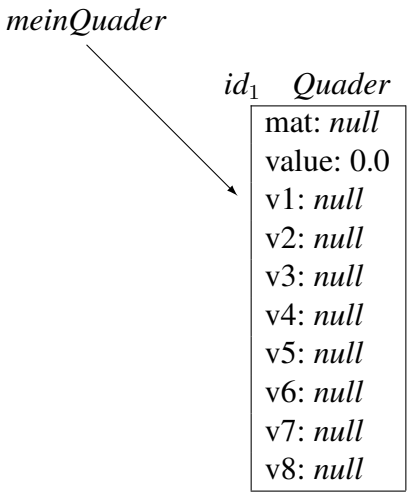



Abbildung 1.4: Die strukturelle Modellierung eines Quader-Objekts

Dazu werden den acht Instanzvariablen *v1*, ..., *v8* neu instanziierte *Vertex*-Objekte zugeordnet. Diese werden passend zu Abbildung 1.2 initialisiert.

Weiterhin erstellen wir eine neue *Material*-Instanz, deren Attribute *name* und *spezGewicht* wir auf *Eisen* und *0.89* setzen. Dieses *Material*-Objekt wird (durch eine Zuweisung an das entsprechende *mat*-Attribut) mit dem zuvor instanziierten *Quader* assoziiert. Selbstverständlich kann dieses *Material*-Objekt auch von anderen Objekten, insbesondere von anderen *Quader*n aus referenziert werden. Dies führt dann zum Konzept der gemeinsamen Unterobjekte (engl. *shared subobjects*), das wir später noch behandeln werden.

Das nachfolgende Programmfragment wird all diese Aufgaben „verrichten“:

```
meinQuader.v1 = new Vertex();          //instanziiere einen neuen Vertex
meinQuader.v1.x = 0.0;
meinQuader.v1.y = 0.0;
meinQuader.v1.z = 0.0;
meinQuader.v2 = new Vertex();
...     // instanziiere und initialisere die anderen Vertex-Objekte
meinQuader.v8 = new Vertex();
meinQuader.v8.x = 0.0;
meinQuader.v8.y = 1.0;
meinQuader.v8.z = 1.0;
meinQuader.wert = 39.99;
meinQuader.mat = new Material();
meinQuader.mat.name = "Eisen";
meinQuader.mat.spezGewicht = 0.89;
```

Der Zustand der Objekte – wir werden dies als *Objektbank* bezeichnen – ist in Abbildung 1.5 illustriert. Die Objektbank enthält 10 Objekte:

- Eine *Quader*-Instanz mit dem abstrakten Objektidentifikator $id_1$
- Acht *Vertex*-Instanzen mit den OIDs $id_{11}$, ..., $id_{18}$
- Eine *Material*-Instanz mit dem Objektidentifikator $id_{77}$

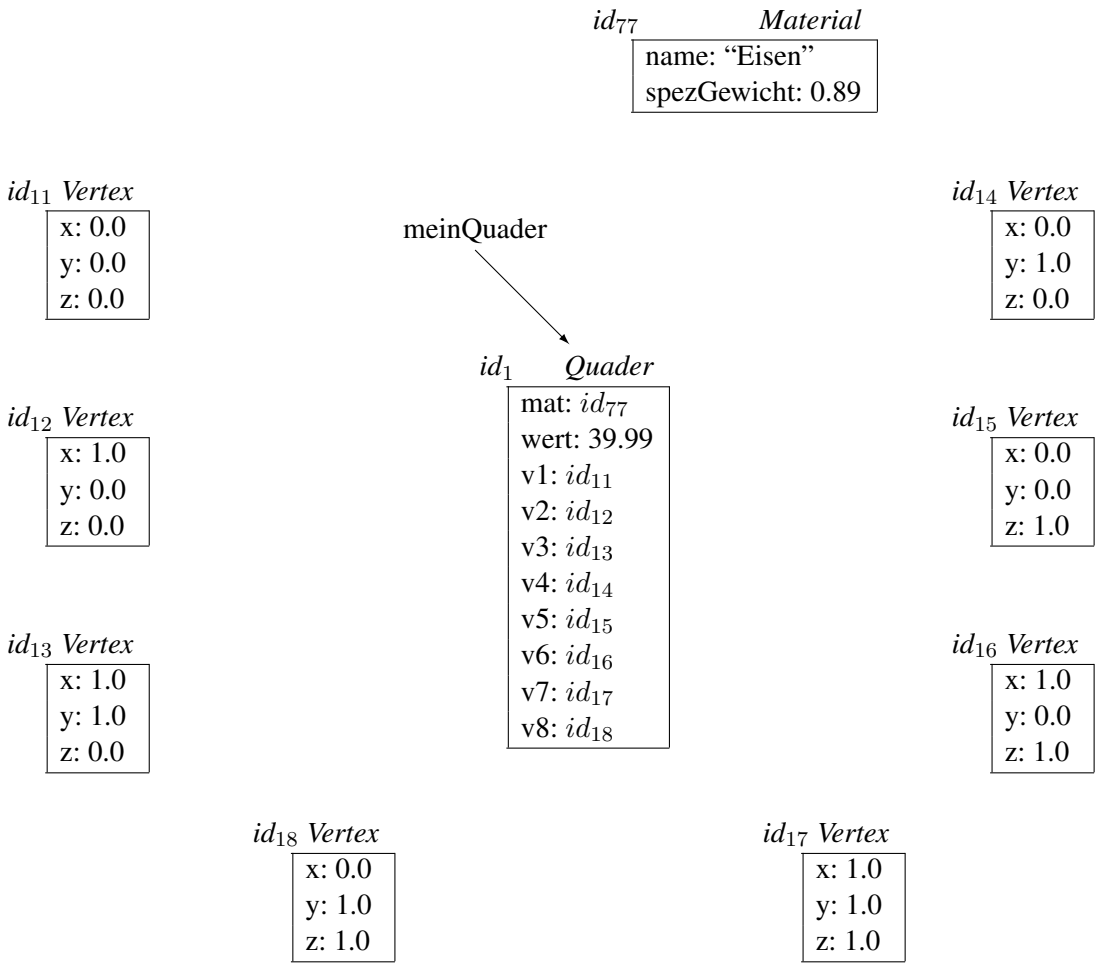



Abbildung 1.5: Der Quader und die zugeordneten *Vertex*- und *Material*-Objekte

Der Zustand der acht *Vertex*-Instanzen, d.h. deren $x$, $y$ und $z$-Attributwerte, ist in Abbildung 1.5 leicht erkennbar. Dass *Quader*-Objekts ist mit seinen neun Attribute, die selbst wieder auf andere Objekttypen eingeschränkt sind, komplexer strukturiert. Das Attribut *mat* verweist auf ein *Material*-Objekt. In Java-Terminologie sagt man, dass das *Material*-Objekt über die eindeutige OID $id_{77}$ von dieser Instanzvariablen *mat* referenziert wird. Diese Referenz wird dadurch realisiert, dass die Objektidentität des referenzierten Objekts (hier: $id_{77}$) in der entsprechenden Instanzvariablen (hier: *mat*) des referenzierenden Objekts (hier: $id_1$) gespeichert wird. Weiterhin betrachten wir die acht Instanzvariablen *v1*, ..., *v8*, die auf *Vertex*-Objekte eingeschränkt sind. Im derzeitigen Zustand der Objektbank verweisen sie auf die *Vertex*-Instanzen $id_{11}$, ..., $id_{18}$.

## 1.6 Objektidentifikation

In der vorangegangenen Diskussion haben wir das Thema der Objektidentifikation schon gestreift. Da es eines der zentralsten Konzepte der objektorientierten Datenmodellierung darstellt, werden wir dieses Thema hier detaillierter behandeln.

### 1.6.1 Motivation für die Objektidentität

Die Identität ist die Eigenschaft eines Objekts, die es von *allen* anderen Objekten unterscheidet. Leider wird das Konzept der Objektidentität oft mit anderen Eigenschaften eines Objekts vermischt. In einigen Anwendungsbereichen – insbesondere in relationalen Datenbanken – wird der Zustand (also die Werte bestimmter Attribute) zur Identifizierung von Objekten verwendet. Diesen Ansatz nennt man demzufolge *inhaltsbasierte* Identität. Diese Form der Objektidentifikation findet man auch „im realen Leben“ sehr häufig, wenn man beispielsweise an die Identifizierung von Personen anhand ihres Namens und Geburtsdatums denkt. Eine andere Form der Objektidentität, die insbesondere in Programmiersprachen wie Java und C++ eingesetzt wird, ist die *speicherortsbasierte* Objektidentität. Hierbei werden die Objekte über ihre physische Adresse im Speicher identifiziert. Beide Ansätze haben gewisse Probleme:

**Inhaltsbasierte Identität** Das erste Problem mit dieser Art der Objektidentität liegt darin begründet, dass man nur eine Möglichkeit hat, die Gleichheit zweier Objekte auszudrücken. $A == B$ bedeutet hierbei immer, dass die beiden Objekte $A$ und $B$ gleich sind hinsichtlich Zustand und Identität. Es gibt keine Möglichkeit, dass zwei Objekte den gleichen Zustand, aber unterschiedliche Identitäten haben.

Ein zweites Problem tritt auf, wenn der Zustand der identifizierenden Attribute sich doch mal ändern sollte. Jede/r Leser/in, die schon einmal den Namen (z.B. wegen einer Eheschließung) geändert hat, wird die lange anhaltenden Probleme leidvoll in Erinnerung haben. Diese Probleme rühren daher, dass in vielen realen Anwendungen der Name sowie das Geburtsdatum als identifizierende Attribute verwendet werden. Wenn sich nun doch einmal eines dieser Attribute ändert, so führt dies unweigerlich zu Adressierungsproblemen. In Computeranwendungen mit inhaltsbasierter Identität kann dies zu Verletzungen der referentiellen Integrität (also so genannten *dangling references*) führen, da es zu der OID kein entsprechendes Objekt mehr gibt. Diese inkonsistenten Referenzen müssen demnach auch alle geändert werden, bevor die Objektbank wieder in einem konsistenten Zustand ist.

**Speicherort-basierte Identität** Bei dieser Art der Objektidentität werden Objekte gemäß ihres Speicherorts identifiziert und adressiert. Diese Art der OID wird in den objektorientierten Programmiersprachen eingesetzt, indem die Anfangsadresse eines Objekts im Speicher (genauer im virtuellen Adressraum) verwendet wird. Man nennt diese Art der OIDs deshalb oft auch *physische Objektidentifikatoren*.

Ein möglicher Nachteil dieser Art der Objektidentifizierung besteht darin, dass der Speicherort eines Objekts nach dessen Löschung wiederbenutzt wird. Deshalb kann ein neu erzeugtes Objekt dieselbe OID haben wie ein zuvor gelöschtes Objekt. In manchen Programmiersprachen, wie C++, kann dies zu Problemen führen, da es möglicherweise noch Referenzen auf das alte Objekt gegeben hat. Diese Referenzen verweisen nun auf das neue Objekt, das möglicherweise einen anderen Typ hat.

In Java kann diese Art der Inkonsistenz nicht auftreten, da Java keine explizite Löschung von Objekten erlaubt. In Java werden Objekte erst dann vom so genannten *garbage collector* aus dem Speicher entfernt, wenn sie garantiert nicht mehr zugreifbar sind – so dass es also keine Referenzen mehr auf diese Objekte geben kann.

Deshalb sind die Java-Referenzen immer „sicher“ – auch wenn sie systemintern als physische OIDs realisiert sind.

Ein Nachteil der Speicherort-basierten OIDs besteht aber darin, dass Objekte nur mit großem Aufwand verschoben werden können, da alle auf sie verweisenden Referenzen entsprechend geändert werden müssen. Dies ist insbesondere in verteilten Datenbanken ein Problem, wenn Objekte von einem Knoten des Systems zu einem anderen migriert werden sollen.

Diese Diskussion verdeutlicht, dass die Objektidentität möglichst invariant über die gesamte Lebenszeit eines Objekts sein sollte. Eine Änderung der Objektidentität hat schwerwiegende Konsequenzen: Entweder führt sie zu Inkonsistenzen wegen falscher Referenzen oder sie macht es notwendig, dass die Objektbank nach Referenzen zu diesem nun anders identifizierten Objekt durchsucht werden muss. Im Idealfall sollte man also so genannte *logische Objektidentitäten* verwenden, die unabhängig vom Zustand (Inhalt) und vom Speicherort des Objekts sind. In unseren Diskussionen und Schaubildern gehen wir von derartigen logischen OIDs aus, die während der gesamten Lebenszeit des Objekts invariant bleiben.

### 1.6.2 Abstrakte Objektrepräsentation

Jedes Objekt $o$ kann man demnach als Tripel der folgenden Form auffassen:

$$o = (id_{\#}, \mathit{Typ}, \mathit{Rep})$$

Die drei Teile haben die folgende Bedeutung:

- $id_{\#}$ stellt den Objektidentifikator des Objekts $o$ dar.
- *Typ* spezifiziert den Objekttyp, von dem das Objekt instanziiert wurde.
- *Rep* entspricht dem internen Zustand (der derzeitigen strukturellen Repräsentation) des Objekts $o$.

Im Falle eines Datensatz-strukturierten Objekts $o$, stellt *Rep* die derzeitigen Werte der Attribute/Instanzvariablen dar. Falls $o$ ein Objekt eines Array-Typs ist, entspricht *Rep* den derzeitigen Elementen des Arrays – in der gegebenen Reihenfolge. Wir verweisen auf Kapitel 1.10 für eine detailliertere Beschreibung von Arrays.

Als Beispiel wollen wir uns jetzt die Objektbank aus Abbildung 1.5 anschauen: Es gibt ein Objekt mit dem OID $id_1$ vom Typ *Quader*. Weiterhin zeigt die Abbildung acht *Vertex*-Objekte mit den OIDs $id_{11}, \ldots, id_{18}$ sowie ein Objekt mit dem OID $id_{77}$ vom Typ *Material*. Der interne Zustand *Rep* des Objekts $id_1$ entspricht den aktuellen Werten der 10 Attribute *mat*, *wert* und *v1*, ..., *v8*.

In unseren Schaubildern werden wir Objektidentitäten immer abstrakt als $id_1$, $id_2$, etc. bezeichnen, um von der konkreten Realisierung zu abstrahieren. Wir gehen davon aus, dass jedem Objekt bei seiner Instanziierung ein eindeutiger, invarianter Objektidentifikator gegeben wird. Wir werden weiterhin davon ausgehen, dass die Objekte niemals gelöscht werden, so lange es noch Referenzen gibt, über die das Objekt erreichbar ist.

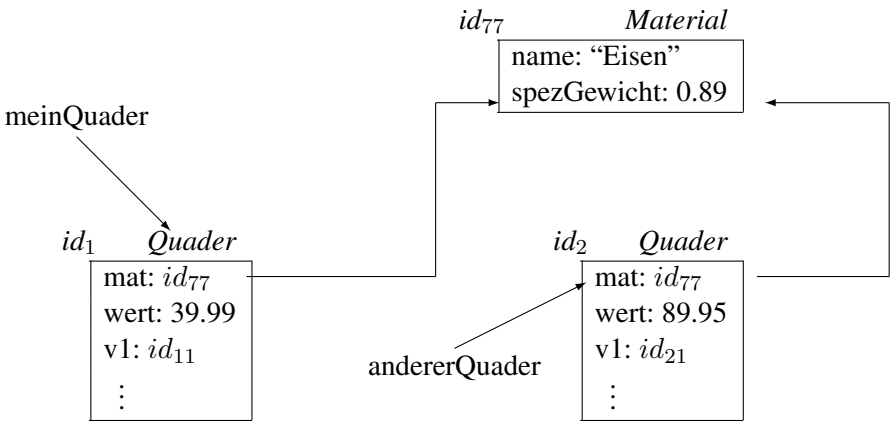



Abbildung 1.6: Illustration eines gemeinsamen Unterobjekts

## 1.7 Gemeinsame Unterobjekte

Die sogenannten *gemeinsamen Unterobjekte* (engl. *shared subobjects*) stellen eines der schwierigeren Konzepte bei der objektorientierten Datenmodellierung dar. In objektorientierten Datenmodellen bzw. Programmiersprachen wie Java gibt es keinerlei Restriktionen hinsichtlich der Anzahl der Referenzen, die auf ein Objekt verweisen können. Daher kann man beliebig komplexe Objektstrukturen definieren, bei denen Objekte auch mehrfach referenziert werden können.

Unter einem gemeinsamen Unterobjekt versteht man ein Objekt, das von mindestens zwei Objekten referenziert wird. Wir wollen dies an Hand eines einfachen Beispiels demonstrieren. Dazu betrachten wir das nachfolgende Programmfragment, bei dem wir mit dem in Abbildung 1.5 gezeigten Objektbank-Zustand starten:

```
Quader andererQuader;
   ...
andererQuader = new Quader();
andererQuader.mat = meinQuader.mat;
```

Die resultierende Objektbank-Struktur ist in Abbildung 1.6. gezeigt. Die *Material*-Instanz mit der OID $id_{77}$ ist jetzt ein gemeinsames Unterobjekt der beiden *Quader*-Objekte, da dieses *Material* jetzt zweimal referenziert wird: einmal über das *mat*-Attribut der *Quader*-Instanz $id_1$ und ein zweites Mal über das *mat*-Attribut der *Quader*-Instanz $id_2$. Auf diese Art kann ein und dasselbe Objekt beliebig oft referenziert werden. Wir werden später sehen, dass mittels dieser Mehrfach-Referenzierung ein Objekt auch in mehreren Kollektionen (Arrays, Mengen, Warteschlangen, oder dergleichen) enthalten sein kann.

Die Komplexität im Umgang mit gemeinsamen Unterobjekten resultiert daraus, dass diese Objekte dann auch über unterschiedliche Referenzen adressiert und manipuliert werden können. Objektmanipulationen, die über irgendeinen der unterschiedlichen Zugriffswege durchgeführt wurden, sind natürlich auch über alle anderen Zugriffswege sichtbar. Dies wird von ungeübten Designern objektorientierter Softwaresysteme oft nicht bedacht und führt deshalb häufig zu Fehlern.

In unserem Beispiel ist die folgende Modifikation der *Material*-Instanz $id_{77}$ auch über die Referenzkette *andererQuader.mat* sichtbar:

```
meinQuader.mat.name = "Kupfer";
meinQuader.mat.spezGewicht = 0.90;
```

Konkret bedeutet dies, dass nun auch die Referenzkette `andererQuader.mat.name` auf den neuen Namen und `andererQuader.mat.spezGewicht` auf das neue spezifische Gewicht verweist. Insbesondere sind die folgenden Prädikate wahr:

```
andererQuader.mat.name.equals("Kupfer");    // Objekt-Vergleich
andererQuader.mat.spezGewicht == 0.90;      // Wert-Vergleich
```

Dieses Beispiel sollte – auch wenn es sehr einfach gehalten ist – deutlich machen, dass die so genannten Seiteneffekte bei der Manipulation von gemeinsamen Unterobjekten manchmal schwer zu überblicken sind. Deshalb muss man bei der Definition komplexer Objektstrukturen mit gemeinsamen Unterobjekten extrem vorsichtig zu Werke gehen.

## 1.8 Beziehungen zwischen Objekten im UML-Modell

In der UML-Modellierung unterscheidet man verschiedene Arten von Unterobjekt-Beziehungen zwischen Objekten:

- Die nicht-exklusive Zuordnung eines Unterobjekts zu dem übergeordneten Objekt. Dies nennt man *Aggregation*. Hierbei kann dasselbe Unterobjekt also mit mehreren übergeordneten Objekten in Beziehung stehen. Dies führt dann zu *gemeinsamen Unterobjekten*, wie wir es am *Material*/*Quader*-Beispiel gesehen haben.

- Die exklusive und existenzabhängige Zuordnung eines Objekts zu einem übergeordneten Objekt. Diese Teil/Ganzes-Beziehung ist eine spezielle Form der Aggregation und wird als *Komposition* bezeichnet.

Diese beiden unterschiedlichen Teil/Ganzes-Beziehungen werden in UML-Notation durch entsprechende Rauten gekennzeichnet, wie dies in Abbildung 1.7 gezeigt ist. Die gefüllte Raute markiert eine exklusive Teil/Ganzes-Zuordnung (Komposition), die nicht-gefüllte Raute die Teil/Ganzes-Beziehung mit gemeinsam genutzten Teil-Objekten (Aggregation). In Abschnitt 1.14.4 gehen wir auf den Unterschied zwischen Aggregation und Komposition noch genauer ein.

## 1.9 Referenzierung und Dereferenzierung

In den vorangehenden Beispielprogrammen haben wir bereits gesehen, dass die Zuweisung eines Objekts an eine (Instanz-)Variable dazu führt, dass die OID des zugewiesenen Objekts dort gespeichert wird. In diesem Sinn realisieren die Objektmodelle die sogenannte *Referenz-Semantik*, wenn man mit Objekten hantiert. Bei Werten – also bei Elementen von primitiven Typen – wird demgegenüber die *Kopier-Semantik* angewendet. D.h. bei der Zuweisung eines Werts an eine Variable (sei es eine lokale Variable oder eine Instanzvariable) wird der Wert kopiert.

Variablen, die auf einen Objekttyp eingeschränkt sind, werden implizit (also automatisch) dereferenziert, wenn man darauf zugreift. Bei der Zuweisung eines Objekts an eine solche Variable erfolgt eine implizite Referenzierung. Wir wollen diese implizite Referenzierung und Dereferenzierung anhand eines Programmfragments beleuchten:

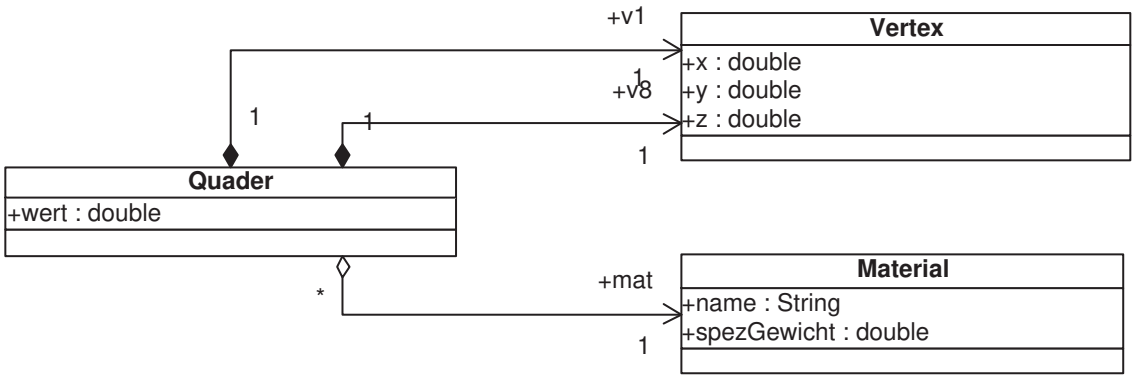



Abbildung 1.7: Die UML-Modellierung des Quaders mit den Unterobjekten Vertex und Material

```
   Quader meinQuader = new Quader();
   Material einMaterial;
   double w;
   ...
(1)einMaterial = new Material();     // einMaterial speichert die OID id88
(2)einMaterial.name = "Carbon";
(3)einMaterial.spezGewicht = 0.75;
(4)meinQuader.mat = einMaterial;     // meinQuader hat die OID id1
(5)w = meinQuader.mat.spezGewicht;
```

In der Zuweisung (1) wird zunächst ein neues *Material*-Objekt instanziiert. Dieses neu instanziierte *Material*-Objekt habe die abstrakte OID $id_{88}$. Diese OID wird in der Variablen *einMaterial* gespeichert. In den beiden nachfolgenden Zuweisungen (2) und (3) wird dieses *Material*-Objekt initialisiert. Dem Attribut *name* wird der String „Carbon“ und dem Attribut *spezGewicht* wird der Wert 0.75 zugewiesen. Die Zuweisung (4) generiert eine Referenz auf das *Material*-Objekt und speichert diese im Attribut *mat* des *Quader*-Objekts. Das Attribut *mat* des *Quader*-Objekts $id_1$ verweist nun auf das *Material*-Objekt $id_{88}$. Abbildung 1.8 zeigt den daraus resultierenden Zustand der Objektbank.

In der Zuweisung (5) sehen wir ein Beispiel für die automatische (implizite) Dereferenzierung: Ausgehend von der lokalen Variablen *meinQuader* wird zunächst auf das referenzierte *Quader*-Objekt $id_1$ zugegriffen, von dort geht es weiter über die Referenzkette *meinQuader.mat.spezGewicht* zu dem *Material*-Objekt mit der OID $id_{88}$, wo letztendlich der Wert des Attributs *spezGewicht* gelesen und der lokalen Variablen *w* zugewiesen wird.

Eine weitere „verdeckte“ implizite Dereferenzierung geschieht bei der Instanziierung der *Material*-Instanz. Im Ausdruck (1) wurde zunächst mittels *new Material()* ein neues Objekt vom Typ *Material* kreiert. Der *new*-Operator liefert eine Referenz zu diesem Objekt zurück, die der lokalen Variablen *einMaterial* zugewiesen wird.

## 1.10 Kollektionen mit Arrays

Zusätzlich zu den Datensatz-strukturierten Objekttypen lassen sich in Java auch einfache Kollektionstypen mithilfe des vorgegebenen Array-Konstruktors definieren. Es gibt weitergehende, funktional mächtigere Kollektionstypen, wie beispielsweise ArrayList, Stack,

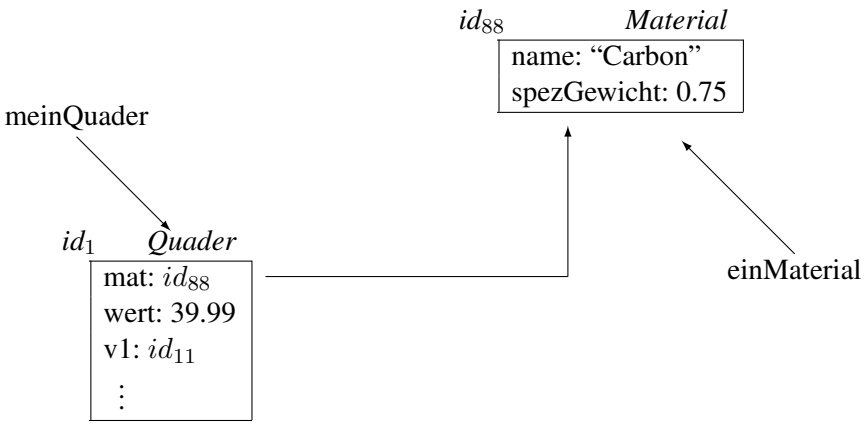



Abbildung 1.8: Objektbank zur Illustration der Referenzierung und Dereferenzierung

PriorityQueue, HashSet, TreeMap, und viele andere. Diese stehen über das Java Collections Framework zur Verfügung. Wir werden uns zunächst mit dem elementarsten Kollektionstypen basierend auf den Arrays beschäftigen, bevor wir in späteren Abschnitten die fortschrittlicheren Kollektionstypen besprechen werden.

Arrays sind eingebaute Java-Konstrukte und können wie folgt verwendet werden:

```
ElementTyp[] arrayName = new ElementTyp[25];
ElementTyp einElem;
arrayName[0] = einElem;
```

Der *ElementTyp* kann entweder eine Sorte (also ein primitiver Typ) oder ein Objekttyp (also eine Klasse) sein. Array-Objekttypen sind für alle eingebauten und benutzerdefinierten Objekttypen und Sorten vordefiniert und brauchen nur noch verwendet zu werden. Die Array-Typen sind in ihrer Funktionalität allerdings (notgedrungen) stark eingeschränkt, da man großen Wert auf Performanz gelegt hat. Ein Array-Objekt hat immer eine feste Länge – in unserem Beispiel hat es die Länge 25. Die einmal festgelegte Länge lässt sich nachträglich nicht mehr ändern. Die Indexpositionen des Arrays sind nummeriert und starten bei 0. Man kann auf ein Element an einer bestimmten Position zugreifen oder ein Element kann an einer bestimmten Position eingefügt werden. Weiterhin kann man ein Array-Objekt nach seiner Länge „fragen“ – dazu dient der Zugriff auf `arrayName.length`, der in unserem Beispiel den Wert 25 liefert.

Wir wollen jetzt anhand eines Beispiels den Umgang mit Array-Objekten demonstrieren, und zwar zunächst an einem *int[]*-Array.

```
int[] kempersTelefonNummern = {6082080,0,28833};
   // äquivalent zu:
   // kempersTelefonNummern = new int[3];
   // kempersTelefonNummern[0] = 6082080;
   // kempersTelefonNummern[1] = 0;
   // kempersTelefonNummern[2] = 28833;

int privNummer = kempersTelefonNummern[0];  // privNummer == 6082080
```

In diesem Programmfragment haben wir zunächst die lokale Variable *kempersTelefonNummern* deklariert und ihr ein implizit generiertes Array-Objekt zugewiesen. Die-

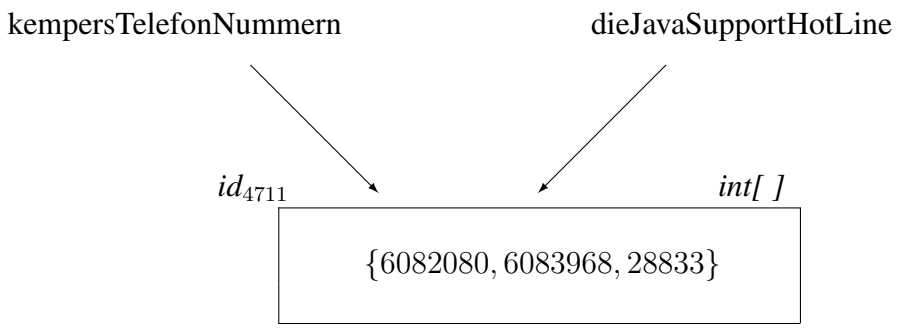



Abbildung 1.9: Objektbank-Zustand mit einem gemeinsamen Array-Objekt

ses Array hat die Länge 3 und wurde mit den drei *int*-Werten initialisiert. Beim Einfügen von Werten in ein Array kommt – genau wie bei der Zuweisung von Werten – die Kopier-Semantik zum Tragen. Wenn wir also im obigen Beispiel der lokalen Variablen *privNummer* einen anderen Wert zuweisen, hat dies keinerlei Auswirkung auf das Array *kempersTelefonNummern*.

Es ist durchaus möglich, dass zwei Variablen auf dasselbe Array-Objekt verweisen. Dann wird dieses Array-Objekt ein *shared subobject*. Dies ist im nachfolgenden Programmfragment gezeigt:

```
int[] dieJavaSupportHotLine;
dieJavaSupportHotLine = kempersTelefonNummern;
```

Dann werden die Modifikationen an dem Objekt mit der OID $id_{4711}$ über beide Zugriffswege gleichermaßen sichtbar. Das folgende Beispiel möge dies verdeutlichen:

```
dieJavaSupportHotLine[1] = 6083968;
```

Der Zustand der resultierenden Objektbank ist in Abbildung 1.9 gezeigt. Das zweite Element des Arrays (also das Element an Position 1) hat jetzt den Wert 6083968.

Array-strukturierte Objekte sind sozusagen „first-class citizens“ in Java. Sie werden ganz analog zu Objekten behandelt, die Datensatz-strukturiert sind (also aus einer Anzahl von Instanzvariablen bestehen). Insbesondere besitzen sie

- eine *Objektidentität*, in unserem Fall $id_{4711}$
- einen Typ, in unserem Fall *int[]*

Array-Typen können also insbesondere auch als Typeinschränkung für Attribute (Instanzvariablen) genommen werden.

Wie bereits erwähnt ist die Funktionalität der Array-Typen in Java recht bescheiden. Arrays werden bei der Instanziierung auf eine bestimmte Länge initialisiert. Die Initialisierung erfolgt beispielsweise wie folgt: `new int[3]`. Diese Länge kann sich dann nicht mehr ändern (Allerdings kann man natürlich der entsprechenden Variablen irgendwann ein anderes Array-Objekt anderer Länge zuweisen). Die Länge kann man abfragen, was insbesondere bei der Programmierung von Iterationen durch alle Array-Elemente von Bedeutung ist. Die Länge unseres Array-Objekts lässt sich durch folgenden Ausdruck ermitteln: `kempersTelefonNummern.length` ergibt 3. Auf die Elemente

des Arrays kann man gemäß ihrer Position zugreifen. Beispielsweise greift man mittels `kempersTelefonNummern[2]` auf das dritte Element zu – die Nummerierung beginnt bei 0. Die Zuweisung erfolgt auch gemäß der Indexposition:

```
kempersTelefonNummern[2] = 28833;
```

Im obigen Beispiel waren die Elemente des Arrays Werte, genauer gesagt *int*-Werte. In einem Array kann man aber auch beliebige Objekttypen als Elementtypen verwenden. Wir wollen dies am Beispiel eines Array-Typs für Quader-Elemente demonstrieren. Dieser Array-Typ hat folgenden Namen: *Quader[]*. Wir können jetzt zwei lokale Variablen diesen Typs deklarieren: *goldBarren* und *bauKloetze*, beide vom Typ *Quader[]*. Nachdem man diesen beiden Variablen neu instanziierte *Quader[]*-Objekte zugewiesen hat, kann man sie befüllen, indem man den entsprechenden Array-Positionen *Quader* zuweist:

```
Quader[] bauKloetze = new Quader[3];
Quader[] goldBarren = new Quader[1];

Quader meinQuader = new Quader();
bauKloetze[0] = meinQuader;
goldBarren[0] = ...;
```

Dieses Programmfragment könnte zu der in Abbildung 1.10 gezeigten Objektbank-Struktur führen. Wir merken an, dass die Arrays jetzt Referenzen auf (komplexe) Objekt-Elemente enthalten. Somit handelt es sich hierbei wieder um die *Referenz-Semantik*. Damit ist es ohne weiteres möglich, dass ein und dasselbe Objekt in mehreren Arrays enthalten ist. In der Tat haben wir das sogar in der Abbildung dokumentiert: Das Objekt mit der OID $id_3$ ist sowohl in dem Array-Objekt $id_{59}$, das über die Variable *bauKloetze* zugreifbar ist, als auch in dem Array $id_{60}$, das über die Variable *goldBarren* zu erreichen ist, enthalten. Es spielt dabei natürlich keine Rolle, ob dieses Objekt jeweils an derselben oder einer anderen Indexposition angesiedelt ist.

Basierend auf dem Array-Typ *Vertex[]* könnten wir nun auch einen abgeänderten *Quader2*-Typ definieren, der eine Instanzvariable vom Typ *Vertex[]* verwendet, um die acht Eckpunkte zu referenzieren. Die topologische Struktur des modellierten Quaders muss natürlich wie zuvor den Konventionen aus Abbildung 1.2 gehorchen. Die Definition der Klasse *Quader2* mit einer Instanzvariablen *eckPunkte* vom Typ *Vertex[]* geht dann folgendermaßen:

```
class Quader2 {
    public Vertex[] eckPunkte;
    public Material mat;
    public double wert;
}
```

Wir bezeichnen in diesem Zusammenhang die Klasse *Quader2* auch als Klient des (von Java automatisch generierten) Objekttyps *Vertex[]*. Indirekt ist *Quader2* natürlich auch ein Klient der Klasse *Vertex*, da die Klassendefinition darauf aufbaut.

Eine mögliche Instanz dieses Objekttyps *Quader2* ist in Abbildung 1.11 gezeigt. Wir nehmen an, dass die Attribute *mat*, *wert* und *eckPunkte* entsprechend der Abbildung initialisiert wurden. Die Instanzvariable *eckPunkte* muss dementsprechend wie folgt initialisiert worden sein:

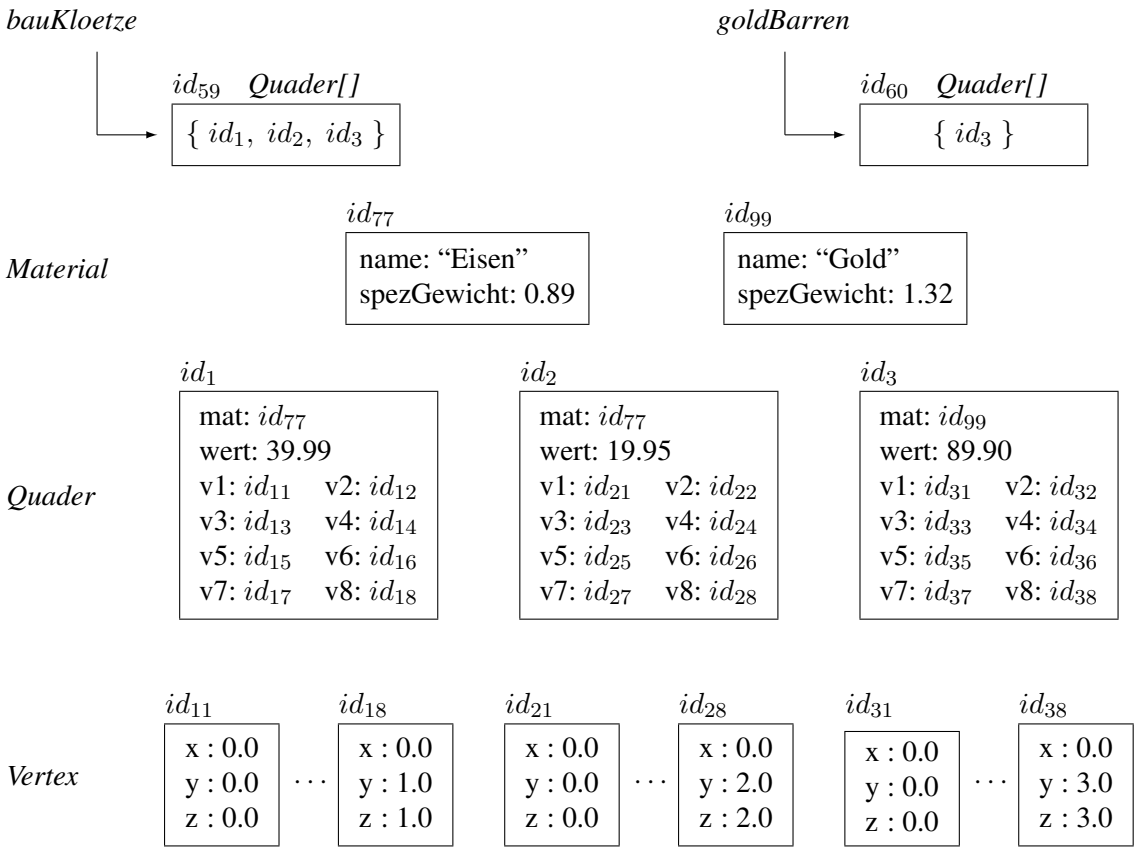



Abbildung 1.10: Objektbank mit zwei *Quader[]*-Objekten

```
einQuader.eckPunkte = new Vertex[8];
```

Wir nehmen an, dass die lokale Variable *einQuader* das Quader-Objekt $id_{299}$ referenziert. Die Objektidentitäten $id_{111}, \ldots, id_{118}$ referenzieren entsprechende *Vertex*-Instanzen, die in dieser Abbildung nicht gezeigt sind.

## 1.11 Typisierung

In diesem Abschnitt wollen wir Aspekte der Typisierung diskutieren, die relevant sind für eine typ-sichere Programmiersprache. Man unterschiedet zwischen *typ-sicheren* und *nicht typ-sicheren* Programmiersprachen. Die typ-sicheren Sprachen haben den Vorteil, dass alle auf Typ-Inkonsistenzen basierenden Fehler schon während der Übersetzungszeit ermittelt werden. Das heißt, dass der Compiler alle Programme zurückweist, die möglicherweise zur Laufzeit einen Fehler wegen einer Typ-Inkonsistenz verursachen könnten. Ein ganz einfaches Beispiel einer derartigen Typ-Inkonsistenz wäre der Zugriff auf ein nicht vorhandenes Attribut eines Objekts, also beispielsweise:

```
meinQuader.ehePartner
```

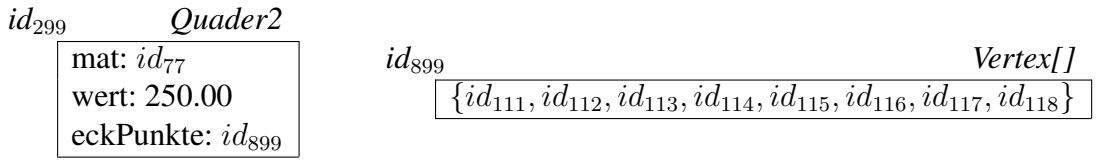



Abbildung 1.11: Repräsentation einer *Quader2*-Instanz

Das Attribut *ehePartner* ist bei einem Objekt vom Typ *Quader* nicht definiert. In einem typ-sicheren Modell wird dieser Fehler zum Zeitpunkt der Übersetzung aufgedeckt.

Jetzt kommt unweigerlich die Frage auf, warum es dann überhaupt nicht typ-sichere Objektmodelle gibt. Der Grund liegt darin, dass diese Modelle eine höhere Ausdrucksfähigkeit und mehr Flexibilität bei der Datenmodellierung als die typ-sicheren Modelle zulassen. Um Typfehler auszuschließen, müssen typ-sichere Sprachen alle Programmausdrücke dahingehend einschränken, dass sie unter allen möglichen Zuständen der Objektbank typ-konsistent sind. Nicht typ-sicheren Modelle hingegen übertragen diese Verantwortung an den Programmierer. Unglücklicherweise sind Typ-Konflikte in nicht-typisierten Modellen sehr schwer vorauszusehen, da deren Auftreten vom jeweilig aktuellen Zustand der Objektbank abhängt. Deshalb mag ein Programm lange Zeit fehlerfrei gelaufen sein und kann trotzdem irgendwann – bei einem ungünstigen Objektbank-Zustand – auf Grund eines Typ-Fehlers abstürzen.

In den nicht typ-sicheren Modellen verzichtet man auf die Typeinschränkung der einzelnen Objektbank-Komponenten, also den Instanzvariablen, Array-Elementen, Variablen, Parametern, etc. Dies ermöglicht natürlich eine sehr flexible Datenmodellierung, da man diesen Komponenten dann beliebige Objekte zuordnen kann. Zum Beispiel wäre folgendes Programmfragment in einer nicht typ-sicheren Programmiersprache möglich:

```
var meinQuader;        // nicht Java - hypothetische Sprache
...
if (b)
   meinQuader = new Vertex();
else
   meinQuader = new Quader();
meinQuader.value = 3.5;      // potenzieller Typkonflikt
...
```

Dieses Programm kann zu einem Typfehler führen, muss es aber nicht. In dem Fall, dass das Prädikat *b* der *if*-Anweisung *true* ist, wird der Variablen *meinQuader* eine *Vertex*-Instanz zugewiesen, anderenfalls eine *Quader*-Instanz. Falls ersteres der Fall ist, würde der nachfolgende Zugriff auf *meinQuader.value* zu einem Fehler führen, da *Vertex*-Objekte dieses Attribut nicht besitzen. Ob dieser Fehler aber wirklich auftritt, kann in der Regel nicht vom Compiler entschieden werden, da dies von der Programmausführung (und damit möglicherweise von externen Eingabewerten) abhängt.

Typ-sichere Modelle nennt man auch *statisch typ-sicher*, um zu betonen, dass die Typ-Sicherheit schon zur Compilezeit verifiziert wird. Im Gegensatz dazu gibt es die weniger rigiden *dynamisch typisierten* Objektmodelle, die diese Typsicherheit nicht zur Compilezeit garantieren können, da ihnen die Typeinschränkungen fehlen. Ein Vertreter dieser

Modelle ist die Programmiersprache Smalltalk; wohingegen Java zu den statisch typsicheren Objektmodellen zu zählen ist. Man bezeichnet diese typ-sicheren Modelle auch als *streng typisiert*, da die Programmierer für jeden Bezeichner eine Typ-Einschränkung angeben müssen. Daraus ergeben sich folgende Vorteile:

1. *Typ-Sicherheit*
   Der Java-Compiler verifiziert die Typkonsistenz. Dadurch wird die Gefahr von Laufzeitfehlern reduziert. Insbesondere wird das Auftreten von Laufzeitfehlern aufgrund von Typ-Inkonsistenzen – beispielsweise durch den Zugriff auf eine nicht vorhandene Instanzvariable eines Objekts – gänzlich eliminiert. Diese Art von Laufzeitfehlern ist eine sehr schwerwiegende Fehlerquelle in den nicht typ-sicheren *dynamisch typisierten* Programmiersprachen.

2. *Effizienz*
   Programmiersprachen und Objektmodelle ohne Typeinschränkungen sind inhärent ineffizienter als streng typisierte Programmiersprachen mit Typeinschränkungen. Das liegt darin begründet, dass der Compiler im Allgemeinen keine Typen für die Ausdrücke herleiten kann und deshalb viele Entscheidungen erst zur Laufzeit, wenn der Typ eines Ausdrucks feststellbar ist, gemacht werden können. Bei einer streng typisierten Programmiersprache kann der Compiler (meist) schon zur Übersetzungszeit den Typ der Ausdrücke ermitteln. Dadurch kann die Typüberprüfungen zur Laufzeit entfallen und es entstehen mehr Möglichkeiten für Optimierungen – beispielsweise durch das statische Binden von Operationen.

3. *Bessere Programmstruktur*
   Alle Daten-Komponenten – Attribute/Instanzvariablen, Array-Elemente, lokale Variablen, formale Parameter – sind auf einen bestimmten Typ eingeschränkt. Dadurch werden das Objektmodell und die zugehörigen Operationen oft besser strukturiert und lesbarer als bei weniger rigide typisierten Modellen. In dynamisch typisierten Objektmodellen, wie Smalltalk, fehlen die Typeinschränkungen der Datenkomponenten. Bestenfalls geben die Programmierer einen Hinweis auf deren Typ indem sinntragende Namen verwendet werden – wie beispielsweise *einQuader*, *andererVertex*, etc.

Bei streng typisierten Objektmodellen wie Java werden alle Datenkomponenten auf einen Typ eingeschränkt. Unter Datenkomponenten fallen:

- *Attribute (Instanzvariablen)*, die die interne Struktur der Objekte ausmachen
- *Variablen*, die Objektreferenzen oder Werte beinhalten
- *Parameter* der Operationen (siehe später)
- *Elemente* der Arraytypen

Wir wollen die Typisierungsregeln jetzt anhand einiger Beispiele erläutern, die auf den in Abbildung 1.12 gezeigten Objekttypen *Person* und *Stadt* basieren. In diesem UML-Beispielschema haben wir „normale“ Assoziationen, also solche, die keine Teil/Ganzes-Beziehung darstellen, zwischen den Klassen *Stadt* und *Person*.

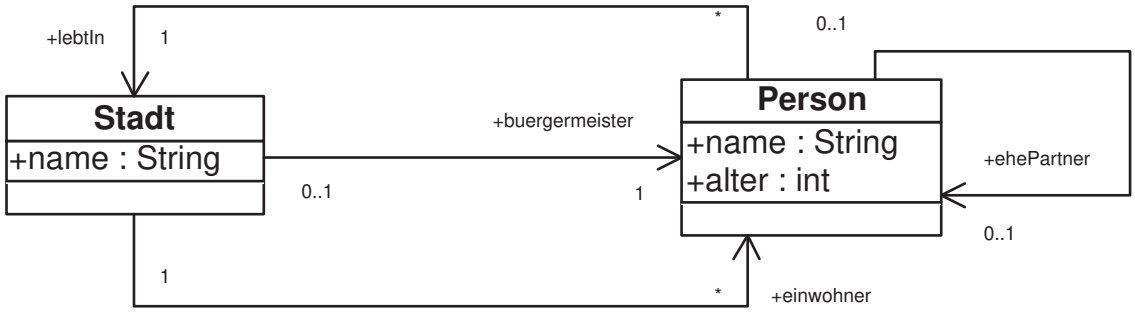



Abbildung 1.12: Die Assoziationen zwischen Person und Stadt

```
class Person {
    public String name;
    public int alter;
    public Person ehePartner;
    public Stadt lebtIn;
}

class Stadt {
    public String name;
    public Person buergermeister;
    public Person[] einwohner;
}
```

Abbildung 1.13: *Person* und *Stadt* Klassen-Definitionen

Basierend auf den Klassen-Definitionen *Stadt* und *Person* aus Abbildung 1.13 können wir nun eine Objektbank mit drei Personen und einer Stadt aufbauen. Dazu deklarieren wir zunächst vier lokalen Variablen:

```
Stadt cityOfLA;
Person mickey, mini, donald;
...     // Instanziierung und Initialisierung der Objekte
```

Mit diesen Variablen könnte man nun die in Abbildung 1.14 gezeigte Objektbank konstruieren. Die Variablen *mickey* und *donald*, die auf den Objekttyp *Person* eingeschränkt waren, referenzieren auch tatsächlich Objekte vom Typ *Person*. Auch verweisen die Attribute der Objekte (wie z.B. *buergermeister*) auf Typ-konforme Objekte. Es gibt allerdings eine Ausnahme: Das Attribut *ehePartner* des Objekts, auf das *donald* verweist, hat den Wert *null*. Dieser spezielle Wert gibt an, dass (derzeit) keine gültige Referenz auf ein entsprechendes Objekt (vom Typ *Person*) existiert. Dies wird durch die Typisierungsregeln nicht ausgeschlossen. Deshalb muss man bei der Programmierung explizit dafür sorgen, dass man nicht versucht, solche *null*-Referenzen zu navigieren. Dies würde nämlich zu einem Laufzeitfehler führen.

Weiterhin ist das Attribut *einwohner* des *Stadt*-Objekts, auf das *cityOfLA* verweist, typ-konsistent. Wie in der Klassendefinition gefordert, verweist dieses Attribut auf ein Array von *Person*en. Das Objekt $id_{115}$ vom Typ *Person[]* enthält drei Elemente, jeweils eine Referenz auf ein Objekt vom Typ *Person* – wie es gemäß der Typeinschränkung gefordert ist.

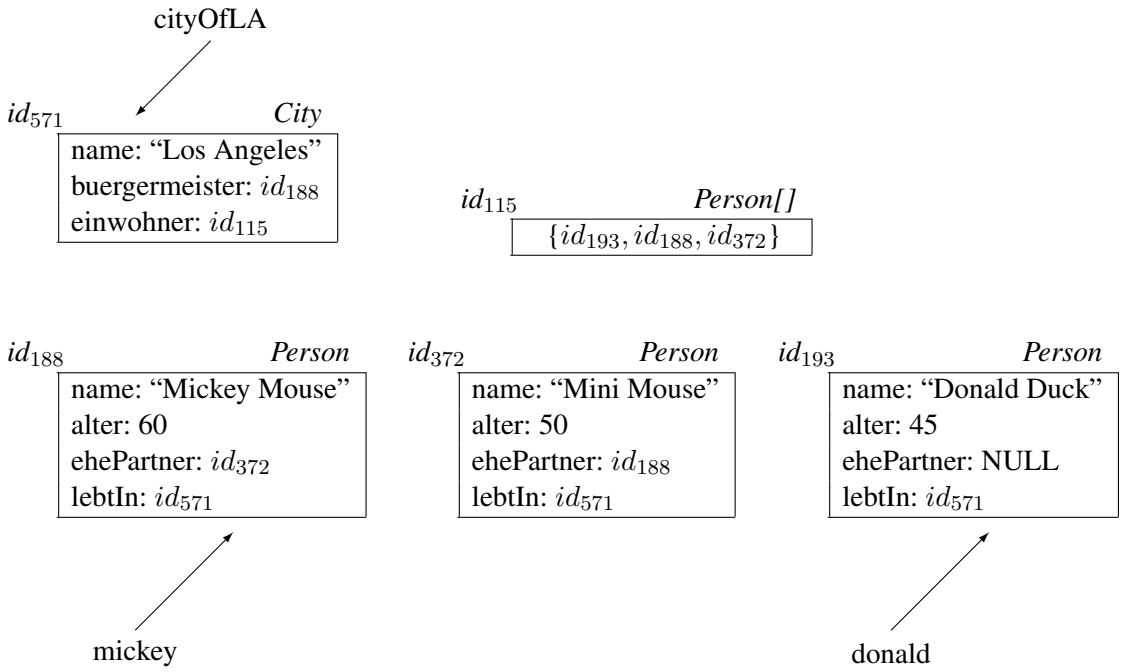
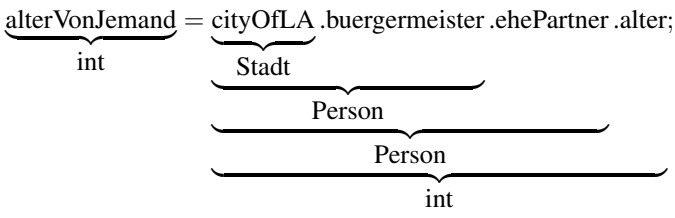



Abbildung 1.14: Objektbank mit drei *Person*en und einer *Stadt*

Der Compiler einer streng typisierten Sprache muss die Typ-Konsistenz statisch, also zur Übersetzungszeit, verifizieren. Dazu wird die Typ-Konsistenz aller Ausdrücke garantiert. Wir wollen die Vorgehensweise dieser Typüberprüfung hier nur ganz kurz anhand einiger Beispiele aus der Objektbank in Abbildung 1.14 illustrieren:

```
   int gesamtAlter, alterVonJemand;
   Person jemand;
   String name;
   ...
(1)alterVonJemand = cityOfLA.buergermeister.ehePartner.alter;
(2)for (int i = 0; i < cityOfLA.einwohner.length; i++) {
      jemand = cityOfLA.einwohner[i];
      gesamtAlter = gesamtAlter + jemand.alter;
   }
```

Im ersten Programmausdruck wird der Variablen *alterVonJemand* vom Typ *int* der Wert des Ausdrucks *cityOfLA.buergermeister.ehePartner.alter* zugewiesen. Um die Typ-Konsistenz zu garantieren, muss der Compiler verifizieren, dass der Typ der Variablen *alterVonJemand* mit dem Typ des Ausdrucks auf der rechten Seite der Zuweisung übereinstimmt. Diese Typüberprüfung wird auf der Basis der Typeinschränkungen in den Deklarationen durchgeführt. Für unser Beispiel kann man die Typ-Konsistenz des ersten Ausdrucks wie folgt verifizieren:



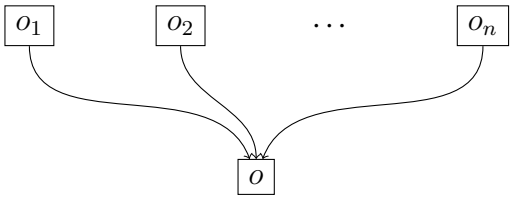
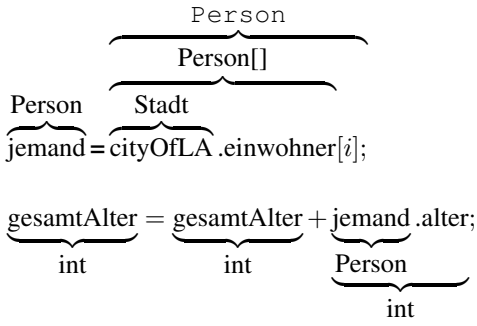



Abbildung 1.15: Gemeinsames Unterobjekt

Die Klammern geben dabei den Typ des jeweiligen Ausdrucks an.

Der zweite Ausdruck, also die beiden Zuweisungen innerhalb der *for*-Schleife, geht wie folgt vonstatten:




Zunächst wird die Typ-Konsistenz der Zuweisung zur *Person*-Variablen *jemand* verifiziert. Da das Array *cityOfLA.einwohner* Elemente vom Typ *Person* enthält, ist insbesondere auch das *i*-te Element vom Typ *Person*. Daraufhin kann man dann auch die Typ-Konsistenz der *int*-Zuweisung an die Variable *gesamtAlter* verifizieren.

## 1.12 Speicherbereinigung: Garbage Collection

Wenn man ein Objekt aus einem Array entfernt (beispielsweise indem man der Indexposition ein anderes Objekt zuweist) oder die Referenz eines Objekts von einer Instanzvariablen löscht (wiederum durch Zuweisung eines anderen Objekts) so führt dies nicht zur Löschung des betreffenden Objekts. Dies wäre ja auch gar nicht sinnvoll, da noch beliebig viele andere Referenzen auf das Objekt verweisen könnten. Man betrachte etwa die Objektbank in Abbildung 1.15.

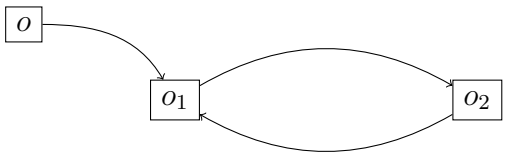
Das Objekt $o$ wird gemeinsam von den $n$ Objekten $o_1, o_2, \ldots, o_n$ benutzt/referenziert. Die Entfernung einer Referenz zwischen $o_1$ und $o$ darf natürlich nicht die anderen $n-1$ Referenzen beeinträchtigen. Deshalb muss die letztendliche Löschung eines Objekts sehr genau vom System kontrolliert werden. Dies ist die Aufgabe der *garbage collection* – im deutschen als Speicherbereinigung bezeichnet.

Nur solche Objekte, die gänzlich unerreichbar geworden sind, dürfen gelöscht werden. Ein Objekt heißt *unerreichbar*, falls es keine Referenzkette mehr gibt, über die das Objekt in einer Anwendung zugegriffen werden kann. Insbesondere darf das Objekt nicht mehr in einem erreichbaren Array enthalten sein oder von einer Instanzvariablen eines erreichbaren Objekts referenziert werden. Objekte, auf die es keine Referenzen mehr gibt,

sind sicherlich unerreichbar. Aber es kann auch Objekte geben, die noch referenziert werden, die aber dennoch unerreichbar sind. Dies passiert dann, wenn diese Referenzen von ihrerseits unerreichbaren Objekten ausgehen.

**Reference Counting** Die automatische Löschung von nicht mehr erreichbaren (also nicht mehr zugreifbaren) Objekten aus dem Speicher wird als *garbage collection* bezeichnet. Eine sehr einfache Realisierung der *garbage collection* könnte man mit Hilfe eines Referenzzählers durchführen, der jedem Objekt zugeordnet ist. Jedesmal wenn eine neue Referenz auf ein Objekt etabliert wird (durch Zuweisung an eine Variable, Einfügung in ein Array, etc.) wird dessen Referenzzähler erhöht. Sobald eine Referenz entfernt wird, wird der zugeordnete Referenzzähler um eins reduziert. Ein Referenzzähler mit dem Wert 0 zeigt ein nicht mehr erreichbares Objekt an, das folglich gänzlich gelöscht werden kann.

So elegant und einfach diese Methode auf den ersten Blick erscheint, so hat sie doch einen schwerwiegenden Nachteil, da nicht erreichbare Zyklen nicht erkannt werden können. Das ist in der nachfolgenden Graphik verdeutlicht:




Die Entfernung der Referenz von $o$ auf $o_1$ reduziert den Referenzzähler von $o_1$ auf 1 – und nicht auf 0, da es ja noch die Referenz von $o_2$ auf $o1$ gibt. Trotzdem sind die beiden Objekte $o_1$ und $o_2$ als Ganzes nicht mehr erreichbar, nachdem die Referenz von $o$ auf $o_1$ entfernt wurde. Um aber derartige unerreichbaren Zyklen finden zu können, benötigt man ausgefeiltere Techniken als die einfache Referenzzählermethode.

**Mark and Sweep** In Java-Implementierungen wird hierzu meist ein sogenannter *mark and sweep*-Algorithmus verwendet. Dabei werden in der ersten Phase, der Markierungsphase, alle erreichbaren Objekte markiert. Anschließend werden in der *sweep*-Phase die nicht erreichbaren Objekte (diejenigen, die in der ersten Phase nicht markiert wurden) gelöscht.

Natürlich verlangsamt der im Hintergrund laufende *garbage collection*-Algorithmus das System und kann bei zeitkritischen Anwendungen durchaus zu Leistungsengpässen führen. Aus diesem Grund wird bei einigen objektorientierten Programmiersprachen, wie z.B. C++, auf die automatische Speicherbereinigung verzichtet. Hier ist es die Aufgabe der Programmierer, die nicht mehr benötigten Objekte explizit zu löschen.

Das folgende Programmfragment veranlasst in einer hypothetischen Programmiersprache die Löschung des *Quader*-Objekts, das von der Variablen *andererQuader* referenziert wird:

```
delete andererQuader;
```

Diese Operation löscht aber nur das äußere *Quader*-Objekt; die referenzierten *Vertex*- und *Material*-Instanzen bleiben davon unberührt. Falls diese auch gelöscht werden sollen, muss dies explizit gemacht werden:

```
delete andererQuader.v1;
...
delete andererQuader.v8;
```

In objektorientierten Programmiersprachen mit expliziter Objektlöschung, wie C++, kann man eine Implementierung für den so genannten *Destruktor* angeben, um beispielsweise die Löschung der Unterobjekte automatisch anzustoßen. Wenn dann ein *Quader*-Objekt gelöscht wird, wird der *Destruktor* aufgerufen und löscht bei entsprechender Implementierung automatisch alle von diesem Quader referenzierten *Vertex*-Objekte.

Die explizite (Benutzer-initiierte) Löschung von Objekten ist aber potenziell gefährlich, da man dadurch so genannte *dangling references* verursachen kann. Wir wollen dies an Abbildung 1.15 verdeutlichen. Wenn man das Objekt $o$ löscht, so sind alle Referenzen von $o_i$ auf $o$ (für $1 \leq i \leq n$) inkonsistent (engl. *dangling*), da das Objekt $o$, auf das verwiesen wird, in der Objektbank nicht mehr existiert.

Dieses Beispiel illustriert, dass das explizite Löschen von Objekten potenziell gefährlich sein kann. In Programmiersprachen mit expliziter Löschung ist es die Aufgabe der Programmierer, dafür zu sorgen, dass nur solche Objekte gelöscht werden, auf die „ganz sicher“ nicht mehr zugegriffen wird. Andererseits darf man aber auch nicht zu großzügig vorgehen, indem man nicht mehr zugreifbare Objekte „leben lässt“, da dies zu so genannten Speicherlecks (engl. *storage leaks*) führt, so dass das System dann irgendwann aufgrund von Speichermangel abstürzt. Warum gibt es dann (noch) Programmiersprachen mit expliziter Objektlöschung? Der Grund liegt in der besseren Leistungsfähigkeit dieser Systeme. Der automatisch im Hintergrund laufende Garbage Collector verursacht natürlich einen gewissen Mehraufwand, der die Ausführung des Systems verlangsamen kann.

## 1.13 Klassen-Attribute

Die Attribute, die wir bislang für die strukturelle Repräsentation von Objekten verwendet hatten, zeichneten sich dadurch aus, dass jedes Objekt seinen eigenen individuellen Satz dieser Attribute bei der Instanziierung bekommen hat. Manchmal gibt es aber auch die Notwendigkeit, ein gemeinsames Attribut für alle Objekte eines Typs anzulegen. Beispielsweise könnte man in derartigen Attributen die Anzahl von Ecken sowie die Anzahl von Kanten speichern, die *Quader*-Objekte besitzen. Da diese Anzahl für alle Objekte der Klasse *Quader* gleich ist, bietet sich die Modellierung als sogenannte Klassen-Attribute an. In Java-Terminologie nennt man diese *statische Variablen*, um zu betonen, dass sie nicht dynamisch jedem Objekt individuell zugeordnet werden, sondern nur einmal pro Klasse existieren. Weiterhin wollen wir uns in einem Klassen-Attribut merken, wie viele *Quader*-Objekte existieren – dieser Wert muss natürlich entsprechend fortgeschrieben werden, wenn neue *Quader* instanziiert werden.

```
class Quader {
    public static final int anzahlKanten = 12;
    public static final int anzahlEcken = 8;
    public static int anzahlQuader = 0;
    public Vertex v1, v2, v3, v4, v5, v6, v7, v8;
    public Material mat;
    public double wert;
}
```

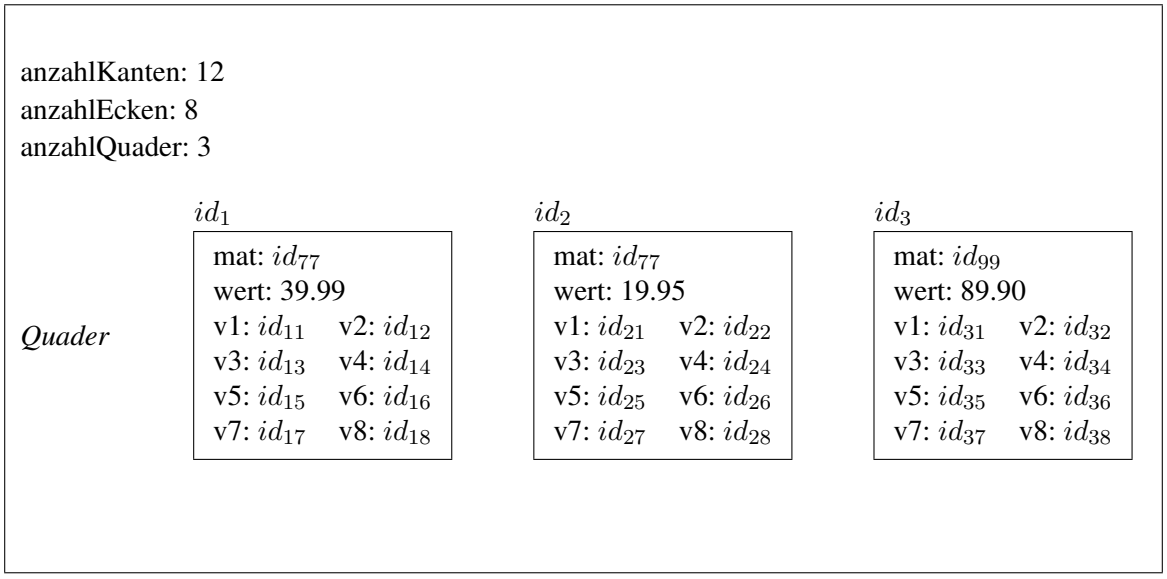



Abbildung 1.16: Illustration der Klassen-Attribute des Quaders

Wir haben in unserem Beispiel auch gleich sichergestellt, dass diese beiden Klassen-Attribute *anzahlEcken* und *anzahlKanten* nicht veränderbar sind. Das heißt, dass der einmal gesetzte Wert von 12 für *anzahlKanten* bzw. 8 für *anzahlEcken* nicht mehr geändert werden kann. Dies geschieht durch das Schlüsselwort *final*. Weiterhin wurde der Wert des Klassen-Attributs *anzQuader* auf 0 initialisiert.

Der Zugriff (bzw. die Modifikation, falls zulässig) eines Klassen-Attributs erfolgt mit der üblichen Punkt-Notation; außer dass dem Punkt jetzt statt einem Objekt der Klassenname vorangestellt ist:

```
System.out.println(Quader.anzahlKanten);
Quader.anzahlQuader = Quader.anzahlQuader + 1;
```

In Abbildung 1.16 wird die Unterscheidung zwischen Klassen- und Instanzvariablen nochmals deutlich illustriert.

## 1.14 Modellierung mit UML und Umsetzung in Java

Im Software-Engineering hat sich die objektorientierte Modellierung durchgesetzt, die mittlerweile auch durch viele verfügbare Modellierungswerkzeuge unterstützt wird. Über lange Zeit gab es konkurrierende Modelle für den objektorientierten Softwareentwurf. Glücklicherweise haben sich einige Forscher und Entwickler zusammengetan und einen gemeinsamen Standard in Form der Unified Modeling Language (UML) entworfen.

In diesem Modell gibt es sehr viele Untermodelle für den Entwurf von Softwaresystemen auf den verschiedensten Abstraktionsebenen. Unter anderen gibt es Teilmodelle für die statische Struktur, beispielsweise einen Diagrammtyp für die Klassenstruktur von Softwaresystemen. Das Zusammenspiel von Objekten bzw. deren Operationen in komplexeren Anwendungen lässt sich mit Hilfe von Sequenzdiagrammen beschreiben. Auf einer noch höheren Ebene kann man Anwendungsfälle (Use cases) des zu realisierenden

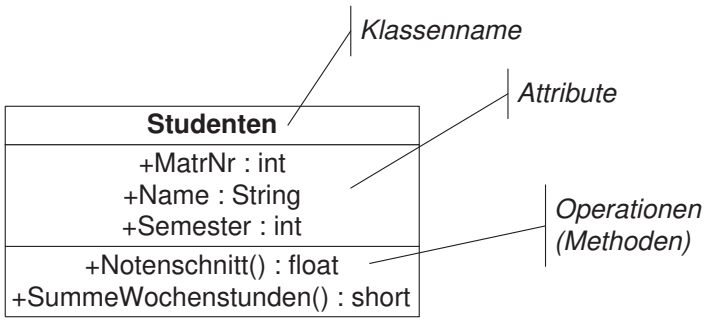



Abbildung 1.17: Eine Beispielklasse *Studenten*

Systems graphisch festlegen. Aktivitäts- und Zustandsdiagramme werden für die Spezifikation von Zustandsübergangen als Folge von ausgeführten Aktivitäten (Operationen, Benutzerinteraktionen) verwendet. Weiterhin gibt es graphische Notationen für die Zerlegung des Systems in Teilsysteme (Komponenten, Packages).

Für den konzeptuellen Entwurf ist zunächst die strukturelle Modellierung des Systems bestehend aus Klassen und Assoziationen zwischen den Klassen am wichtigsten. Wie in Java beschreiben UML-Klassen eine Menge von gleichartigen Objekten. Zusammenhänge zwischen Objekten werden als Assoziationen zwischen den Klassen beschrieben.

### 1.14.1 UML-Klassen

Eine Klasse beschreibt nicht nur die strukturelle Repräsentation (Attribute) der Objekte sondern auch deren Verhalten in der Form von zugeordneten Operationen. Wir wollen uns gleich die in Abbildung 1.17 gezeigte Beispielklasse *Studenten* anschauen: Demnach wird der Zustand von Objekten der Klasse *Studenten* durch die Attribute *MatrNr*, *Name* und *Semester* repräsentiert. Weiterhin sind der Klasse *Studenten* exemplarisch die beiden Operationen *Notenschnitt()* und *SummeWochenstunden()* zugeordnet. Erstere ermittelt aus den abgelegten Prüfungen (siehe unten) die Durchschnittsnote und letztere errechnet aus den gehörten Vorlesungen die Anzahl der Semesterwochenstunden.

In UML unterscheidet man zwischen öffentlich sichtbaren (mit + gekennzeichneten), privaten (mit − gekennzeichneten) und in Unterklassen sichtbaren (mit # gekennzeichneten) Attributen bzw. Operationen. Das Java-Pendant dazu sind die Schlüsselworte *public*, *private* und *protected*.

### 1.14.2 Assoziationen zwischen Klassen

Beziehungen zwischen den Objekten werden als Assoziationen zwischen Klassen in UML modelliert. Als Beispiel zeigen wir in Abbildung 1.18 die binären Assoziationen *hören* und *voraussetzen*. Zusätzlich zum Assoziationsnamen haben wir auch die Rollen angegeben: *Studenten* haben in der Assoziation *hören* die Rolle der *Hörer*. Wichtiger ist die Rollenspezifikation bei der rekursiven Assoziation *voraussetzen*. Hierbei fungiert eine Vorlesung als *Nachfolger*, die andere ist dann implizit (ohne dass wir dies nochmals explizit angegeben haben) die Vorgänger-Lehrveranstaltung.

Man kann Assoziationen in UML eine Richtung zuordnen. Dadurch wird angegeben, in welcher Richtung man auf die assoziierten Objekte zugreifen kann. In unserem

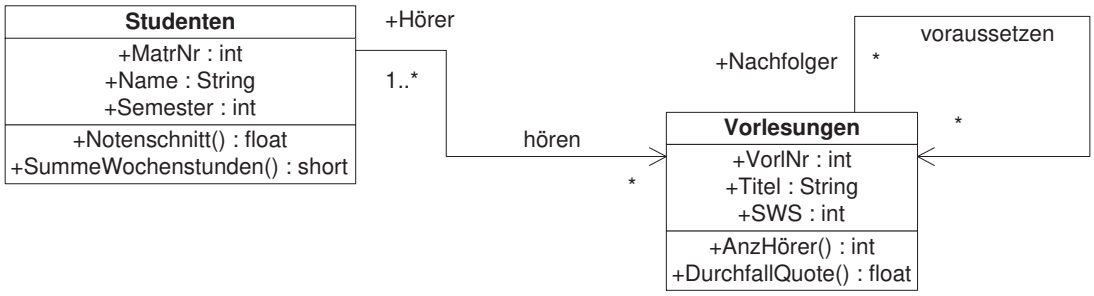



Abbildung 1.18: Assoziationen zwischen den Klassen *Studenten* und *Vorlesungen*

Beispiel wurde angegeben, dass man von einem Student aus die gehörten Vorlesungen ermitteln kann; umgekehrt kann man von einer Vorlesung aus die hörenden Studenten nicht (so leicht) ermitteln. Bei der Assoziation *voraussetzen* ist es analog: Von einer Vorlesung kann man zu den Vorgänger-Vorlesungen traversieren. Bei der objektorientierten Programmierung ist diese Traversierungsrichtung relevant, da man dort die Assoziationen als Referenzen auf die assoziierten Objekte modelliert. Diese Referenzen würden dann gemäß der Traversierungsrichtung in dem Objekt abgespeichert, von dem die gerichtete Assoziation ausgeht. Wenn eine Assoziation in beiden Richtungen traversierbar ist (keine Pfeile bzw. Pfeile an beiden Enden), dann werden die Referenzen entsprechend in beiden Objekten gespeichert. Wir werden dies später noch genauer beleuchten.

In UML besteht die Möglichkeit Beziehungen hinsichtlich der Anzahl der teilnehmenden Objekte zu beschreiben. In UML-Terminologie bezeichnet man dies als *Multiplizität* der Assoziation. In unserem Beispiel *hören* wurde diese als $1..*$ auf der Seite der *Studenten* und als $*$ auf der Seite der *Vorlesungen* angegeben. Die Intervallangabe $1..*$ bedeutet, dass eine Vorlesung mindestens $1$ Hörer hat und maximal beliebig viele. Studenten hören beliebig viele Vorlesungen, was mit dem $*$ auf der Seite der Vorlesungen angegeben ist.

Betrachten wir das abstrakte Beispiel in Abbildung 1.19: Wenn man in UML an einer Seite der binären Assoziation die Multiplizitätsangabe $i..j$ macht, so bedeutet dies, dass jedes Objekt der Klasse auf der anderen Seite mit mindestens $i$ und höchstens $j$ Objekten der Klasse auf dieser Seite in Beziehung stehen muss. Bezogen auf unser Beispiel bedeutet dies, dass jedes Objekt der Klasse $E_1$ mit mindestens $i$ und mit maximal $j$ Objekten der Klasse $E_2$ in Beziehung stehen muss/darf. Analog muss jedes Objekt der Klasse $E_2$ mit mindestens $k$ Objekten der Klasse $E_1$ in Beziehung stehen und es darf maximal mit $l$ Objekten der Klasse $E_1$ in dieser Beziehung stehen. Wenn die minimale und die maximale Anzahl übereinstimmt, also $m..m$ gilt, so vereinfacht man dies in UML zu einem einzigen Wert $m$. Dies gilt auch für $*..*$ was immer als $*$ angegeben wird.

### 1.14.3 Funktionalitäten

Eine Beziehung/Assoziation *R* zwischen den Klassen $E_1$ und $E_2$ kann als Relation im mathematischen Sinn angesehen werden. Demnach stellt die Ausprägung der Beziehung *R* eine Teilmenge des kartesischen Produkts der an der Beziehung beteiligten Klassen dar:

$$R \subseteq E_1 \times E_2$$

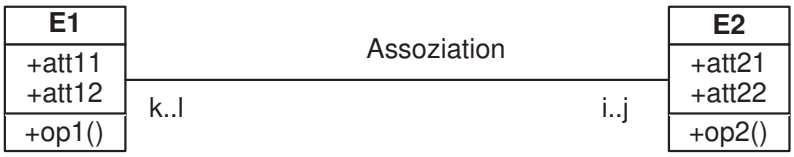



Abbildung 1.19: Die Multiplizität einer abstrakten Assoziation

Ein Element $(e_1, e_2) \in R$ nennt man eine Beziehungsinstanz des Beziehungstyps, wobei $e_i \in E_i$ für alle $1 \leq i \leq 2$ gelten muss. Eine solche Beziehungsinstanz ist also ein Tupel aus dem kartesischen Produkt $E_1 \times E_2$.

Man kann jetzt auch den Begriff der Rolle etwas formaler fassen. Dazu veranschaulichen wir uns nochmals die Beziehung *voraussetzen* aus unserem Beispielschema (siehe Abbildung 1.18). Gemäß dem oben skizzierten Formalismus gilt:

$$\text{voraussetzen} \subseteq \text{Vorlesungen} \times \text{Vorlesungen}$$

Um eine Instanz $(v_1, v_2) \in$ *voraussetzen* genauer zu charakterisieren, wird die jeweilige Rolle angegeben: ($\textit{Vorgänger} : v_1$, $\textit{Nachfolger} : v_2$). Dadurch ist klar ersichtlich, dass die Vorlesung $v_1$ die Voraussetzung für die Vorlesung $v_2$ darstellt.

**Beziehungstypen** Man kann Beziehungen hinsichtlich ihrer *Funktionalität* charakterisieren. Eine binäre Beziehung *R* zwischen den Objekttypen $E_1$ und $E_2$ heißt:

- *Eins-zu-eins* bzw. *1:1-Beziehung*, falls jedem Objekt $e_1$ aus $E_1$ höchstens ein Objekt $e_2$ aus $E_2$ zugeordnet ist und umgekehrt jedem Objekt $e_2$ aus $E_2$ ebenfalls maximal ein Objekt $e_1$ aus $E_1$. Man beachte, dass es auch Objekte aus $E_1$ (bzw. $E_2$) geben kann, denen kein „Partner“ aus $E_2$ (bzw. $E_1$) zugeordnet ist.

  In der UML-Notation aus Abbildung 1.19 ist eine Eins-zu-eins-Assoziation dadurch gekennzeichnet, dass $l = j = 1$ gilt.

  Ein Beispiel einer „realen“ 1:1-Beziehung ist *verheiratet* zwischen den Objekttypen *Männer* und *Frauen* – zumindest nach europäischem Recht.

- *Eins-zu-viele* bzw. *1:N-Beziehung*, falls jedem Objekt $e_1$ aus $E_1$ beliebig viele (also mehrere oder auch gar keine) Objekte aus $E_2$ zugeordnet sein können, aber jedes Objekt $e_2$ aus der Menge $E_2$ mit maximal einem Objekt aus $E_1$ in Beziehung steht.

  In der UML-Notation aus Abbildung 1.19 ist eine Eins-zu-viele-Assoziation dadurch gekennzeichnet, dass $l = 1$ und $j > 1$ gilt. Insbesondere kann $j = *$ gelten.

  Ein anschauliches Beispiel für eine $1{:}N$-Beziehung ist *angestelltBei* zwischen *Personen* und *Firmen*, wenn wir davon ausgehen, dass eine Firma i.a. mehrere Personen beschäftigt, aber eine Person nur bei einer (oder gar keiner) Firma angestellt ist.

- *N:1-Beziehung*, falls analoges zu obigem gilt.

- $N{:}M$*-Beziehung*, wenn keinerlei Restriktionen gelten müssen, d.h. jedes Objekt aus $E_1$ mit mehreren Objekten aus $E_2$ in Beziehung stehen kann und umgekehrt jedes Objekt aus $E_2$ mit mehreren Objekten aus $E_1$ assoziiert werden darf.

In der UML-Notation aus Abbildung 1.19 ist eine Viele-zu-viele-Assoziation dadurch gekennzeichnet, dass $l > 1$ und $j > 1$ gilt.

Man beachte, dass die Funktionalitäten Integritätsbedingungen darstellen, die in der zu modellierenden Welt immer gelten müssen. D.h. diese Bedingungen sollen nicht nur in der derzeit existierenden Objektbank (rein zufällig) gelten, sondern sie sollen Gesetzmäßigkeiten darstellen, deren Einhaltung erzwungen wird.

Die binären 1:1-, 1:$N$- und $N$:1-Beziehungen kann man auch als *partielle Funktionen* ansehen. Bei einer 1:1-Beziehung *R* zwischen $E_1$ und $E_2$ kann die Funktion sowohl als $R : E_1 \to E_2$ wie auch als $R^{-1} : E_2 \to E_1$ gesehen werden.

Bezogen auf unser Beispiel einer 1:1-Beziehung haben wir also:

$$\begin{aligned} \text{Ehemann} : &\quad \text{Frauen} \to \text{Männer} \\ \text{Ehefrau} : &\quad \text{Männer} \to \text{Frauen} \end{aligned}$$

Bei einer 1:$N$-Beziehung ist die „Richtung“ der Funktion zwingend. Die Beziehung *angestelltBei* ist z.B. eine partielle Funktion von *Personen* nach *Firmen*, also:

$$\text{angestelltBei} : \text{Personen} \to \text{Firmen}$$

Die Funktion geht also von dem „$N$“-Objekttyp zum „1“-Objekttyp. Wir werden gleich – im Zusammenhang mit der Umsetzung von UML-Schemata in Java-Klassen – nochmals auf diesen wichtigen Punkt zurückkommen. Analoges gilt natürlich wieder für $N$:1-Beziehungen, wobei wiederum die „Richtung“ der Funktion zu beachten ist.

Die den 1:1- bzw. 1:$N$-Beziehungen zugeordneten Funktionen sind partiell, weil es Entities aus dem Definitionsbereich geben kann, die gar keine Beziehung eingehen. Für diese Entities ist die Funktion somit nicht definiert.

In Abbildung 1.20 sind die oben verbal beschriebenen Funktionalitäten graphisch veranschaulicht. Die Ovale repräsentieren die Objekttypen: das linke den Objekttyp $E_1$ und das rechte den Objekttyp $E_2$. Die kleinen Quadrate innerhalb der Ovale stellen die Objekte des jeweiligen Typs dar und die die Objekte verbindenden Linien repräsentieren jeweils eine Instanz der Assoziation.

### 1.14.4 Aggregation

Als besondere Form der Beziehung zwischen Klassen gibt es die Aggregation. Eine *Aggregation* spezifiziert eine Teil/Ganzes-Beziehung zwischen zwei Klassen. Darüber hinaus gibt es mit der *Komposition* noch eine spezielle Form der Aggregation für Teil/Ganzes-Beziehungen, die existenzabhängigen Teil-Objekte exklusiv zu *einem* übergeordneten Objekt zuordnen. In Abbildung 1.21 wird die Zuordnung der *Prüfungen* zu der einen *Studentin* bzw. dem einen *Studenten* – dem Prüfling – gezeigt. Diese exklusive Zuordnung existenzabhängiger Unterobjekte wird in UML mit der ausgefüllten Raute auf der Seite der übergeordneten Klasse angegeben. Wegen der Existenzabhängigkeit und der Exklusivität muss auf der Seite der übergeordneten Objektklasse immer die Multiplizität 1 (was ja dem Intervall $1..1$ entspricht) angegeben sein, da jedes untergeordnete Objekt wegen der Existenzabhängigkeit mindestens und wegen der Exklusivität maximal einem übergeordneten Objekt zugeordnet ist.

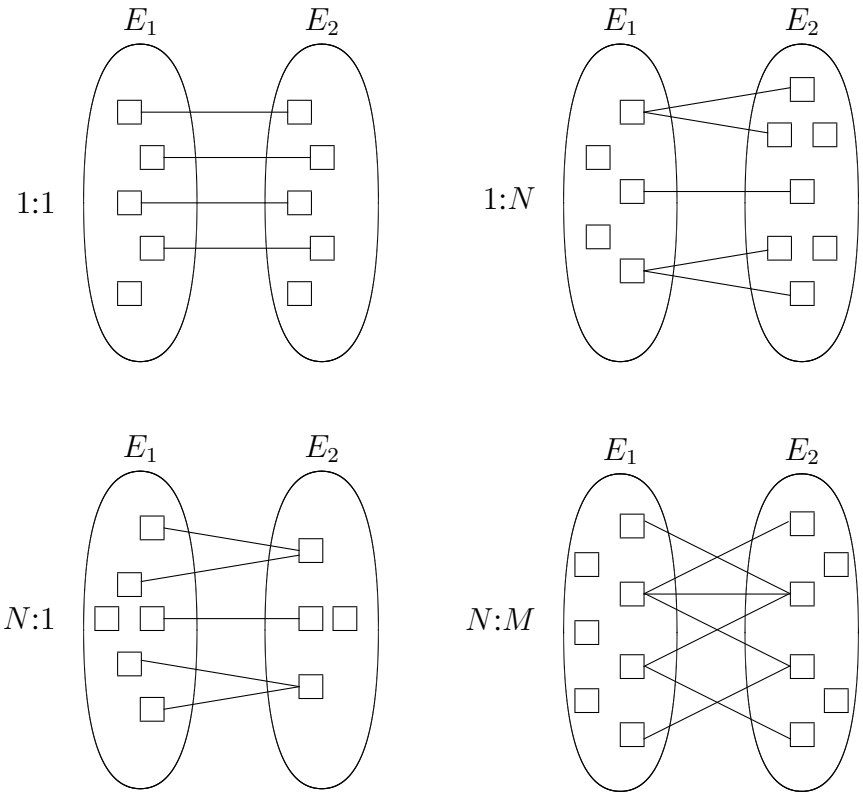



Abbildung 1.20: Graphische Veranschaulichung der Funktionalitäten einer binären Beziehung $R$ zwischen $E_1$ und $E_2$.

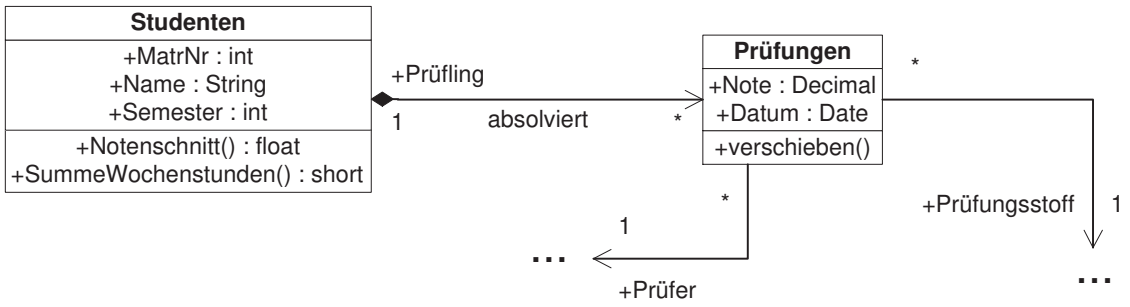



Abbildung 1.21: Die Aggregation zwischen *Studenten* und *Prüfungen*

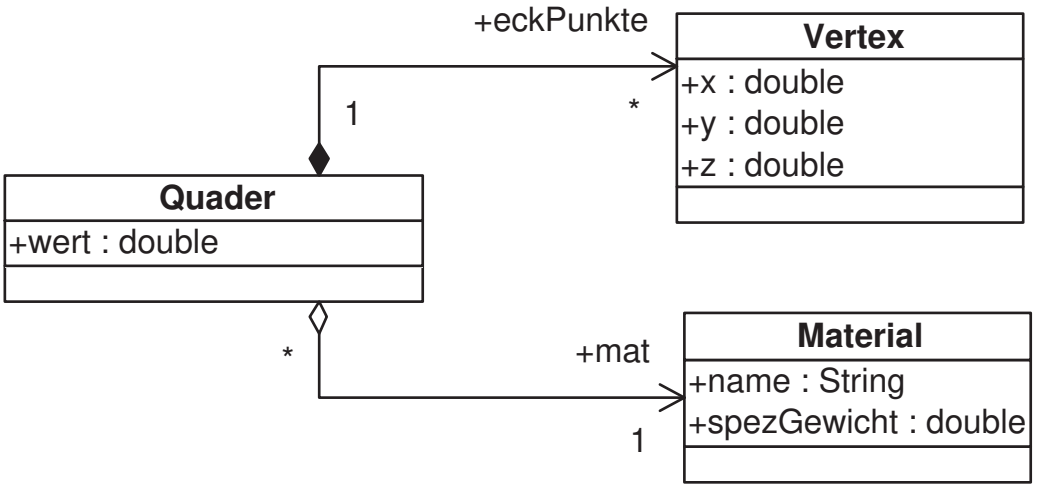
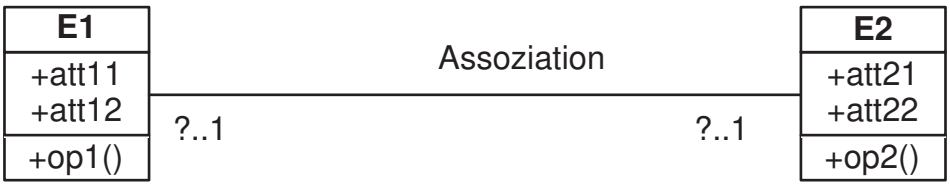



Abbildung 1.22: Die Assoziationen zwischen Quader und Vertex bzw. Material

Die beiden unterschiedlichen Arten der Aggregation – exklusive und nicht-exklusive Zuordnung finden sich auch in dem UML-Schema aus Abbildung 1.22 zur Modellierung von *Quadern*. Die Eckpunkte sind exklusiv zugeordnet, wohingegen das Material nicht-exklusiv zugeordnet ist. Zwei Quader können zwar aus demselben Material bestehen können, aber (zumindest in unserer Modellierung) keine Eckpunkte teilen.

### 1.14.5 Umsetzung von UML-Assoziationen in Java

**Eins-zu-eins-Assoziationen**




```
class E1 {
    public ... att11;
    public ... att12;
    public E2 zugeordnetesE2;
    public ... op1() {}
}

class E2 {
    public ... att21;
    public ... att22;
    public E1 zugeordnetesE1;
    public ... op2() {}
}
```

**Eins-zu-viele-Assoziationen**


```
class E1 {
    public ... att11;
    public ... att12;
    public E2[] zugeordneteE2;
    public ... op1() {}
}

class E2 {
    public ... att21;
    public ... att22;
    public E1 zugeordnetesE1;
    public ... op2() {}
}
```

Betrachten wir hierzu ein konkretes Beispiel: Unsere Quader-Modellierung aus Abbildung 1.22. Hier haben wir zwei Eins-zu-viele-Assoziationen – in entgegengesetzter Richtung. Die Assoziation *mat* ordnet jedem *Quader* ein *Material*, wobei dasselbe *Material* durchaus mehreren *Quader*n zugeordnet werden kann. Die Assoziation *eckPunkte* ordnet jedem *Quader* viele *Vertex*-Objekte zu.

Wenn wir die beiden Assoziationen nur in einer Richtung, nämlich jeweils von *Quader* ausgehend, modellieren wollen, so geschieht dies mit folgender Klassendefinition:

```
class Quader2 {
    public Vertex[] eckPunkte;
    public Material mat;
    public double wert;
}
```

**Viele-zu-viele-Assoziationen**


```
class E1 {
    public ... att11;
    public ... att12;
    public E2[] zugeordneteE2;
    public ... op1() {}
}
```

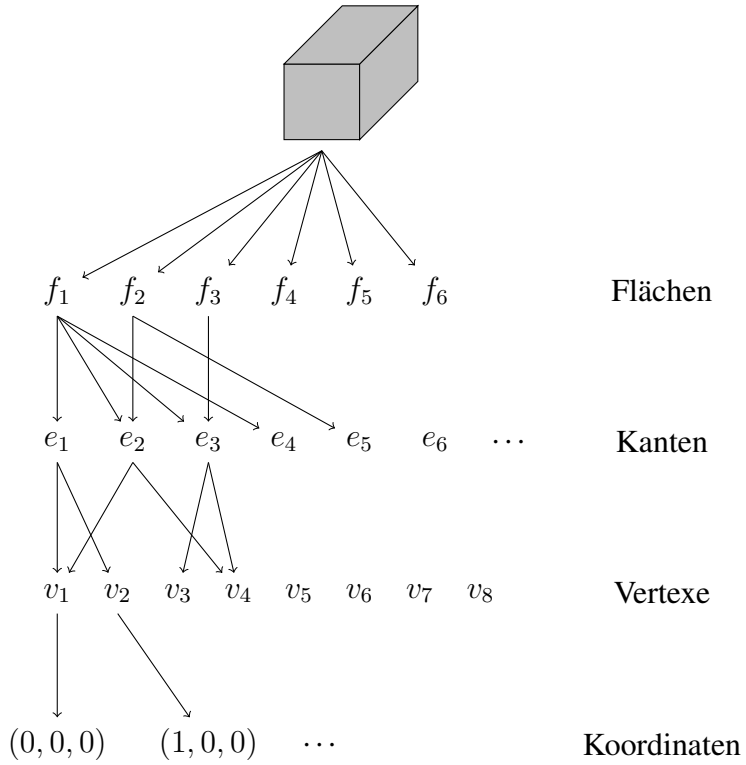



Abbildung 1.23: Begrenzungsflächen-Darstellung eines Quaders

```
class E2 {
    public ... att21;
    public ... att22;
    public E1[] zugeordneteE1;
    public ... op2() {}
}
```

### 1.14.6 Anwendungsbeispiel: Polyeder in UML

Im Text haben wir eine sehr einfache Repräsentation für *Quader*-Objekte verwendet. In CAD-Anwendungen wird hingegen häufig die so genannte Begrenzungsflächen-Darstellung (engl. *boundary representation*) verwendet. Bei dieser Darstellung wird ein geometrisches Objekt über sein Flächen (engl. *faces*) beschrieben, Jede Fläche wird durch ihre begrenzenden Kanten (engl. *edges*) und jede Kante durch ihre Endpunkte modelliert. Dies ist in Abbildung 1.23 graphisch dargestellt.

Aus Datenmodellierungssicht besteht die Begrenzungsflächen-Darstellung aus mehreren Abstraktionsebenen:

- Die *topologische* Repräsentation modelliert die Beziehung zwischen den Flächen, Kanten und Eckpunkten
- Die *metrische* Dimension repräsentiert die Koordinaten der Eckpunkte

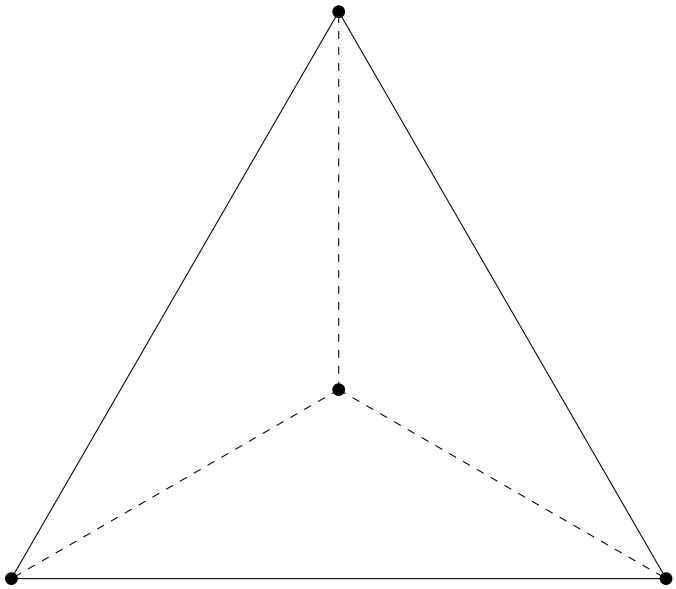


Abbildung 1.24: Graphische Darstellung eines Tetraeders

Die Höhe dieses Baums ist konstant 3. Ein komplexeres geometrisches Objekt als ein Quader führt lediglich zu einer stärkeren Verzweigung – mehr Flächen, mehr Kanten und mehr Eckpunkte. Hinsichtlich der Anzahl der Flächen, Kanten und Eckpunkte ist der Tetraeder das einfachste Polyeder-Objekt. Eine graphische Skizze des Tetraeders ist in Abbildung 1.24 gezeigt. Das korrespondierende UML-Schema ist in Abbildung 1.25 dargestellt. Die Kardinalitäten der Assoziationen ergeben sich aus den Minimalwerten eines Polyeders, nämlich dem Tetraeder.

In dieser Abbildung ist erkennbar, dass ein Polyeder minimal vier umhüllende Flächen besitzt – die maximale Anzahl ist beliebig und wird somit durch $*$ angegeben. Eine Fläche wird – im Falle eines Dreiecks – durch ein Minimum von drei Kanten begrenzt; wiederum ist die maximale Anzahl von begrenzenden Kanten beliebig. Jede Kante begrenzt bei einem Polyeder genau zwei Flächen. Eine Kante wird durch genau zwei Punkte beschrieben, die über die Beziehung *StartEnde* der Kante zugeordnet werden. Bei einem Polyeder gehört jeder Begrenzungspunkt – und nur solche sind in der Objektmenge *Punkte* enthalten – zu mindestens drei Kanten (siehe Tetraeder) und maximal zu beliebig vielen.

Mit *Aggregationen* modelliert man in UML *Teil/Ganzes*-Beziehungen. Die allgemeine Form der Aggregation schränkt nicht ein, zu wie vielen Objekten ein Unterobjekt zugeordnet sein kann oder ob die Existenz des Unterobjekts von einem anderen Objekt abhängt. Für diesen Zweck gibt es die *Komposition* als spezielle Form der Aggregation. Die Komposition ordnet die Teile dem Ganzen exklusiv und existenzabhängig zu. Am besten macht man sich diesen Unterschied an einem Beispiel klar: In Abbildung 1.25 ist die Begrenzungsflächendarstellung für Polyeder als UML-Klassendiagramm gezeigt. Die Aggregation *Hülle* ist eine exklusive Zuordnung von Flächen zu Polyedern und damit auch eine Komposition. Andererseits kann die Assoziation von *Kanten* zu *Flächen* nicht exklusiv sein, da sich immer 2 Flächen dieselbe Kante teilen – daher ist dies nur eine Aggregation und keine Komposition. In der anderen Richtung der Assoziation gilt, dass eine Fläche mindestens 3 Kanten hat. Analog zur Aggregation *Begrenzung* gilt für die Aggregationsassoziation *StartEnde*, dass die Zuordnung von *Punkten* zu *Kanten* nicht

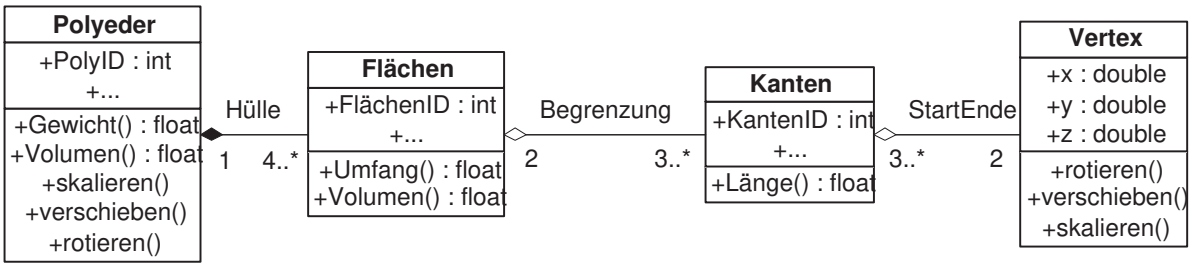



Abbildung 1.25: Die Begrenzungsflächendarstellung von Polyedern

exklusiv ist, da sich mindestens 3 Kanten eines Polyeders einen Punkt teilen. Jede Kante wird von genau 2 Punkten begrenzt.

In diesem Beispiel haben wir den Objektklassen auch jeweils einige Operationen zugeordnet. Es ist zu erkennen, dass der Objektklasse *Polyeder* die Operationen *skalieren()*, *rotieren()* und *verschieben()* zugeordnet sind, die es auch bei der Objektklasse *Vertex* gibt. Hierbei handelt es sich natürlich nicht um Vererbung sondern um die Gleichbenennung von semantisch ähnlichen Operationen. In der Tat würde man die Operationen des übergeordneten Objekttyps *Polyeder* auf der Basis der gleichbenannten Operationen des untergeordneten Objekttyps *Punkte* realisieren. Beispielsweise verschiebt man einen Polyeder, indem man jeden Begrenzungspunkt verschiebt. Man bezeichnet dies – bezogen auf die Implementierung der *verschieben()*-Operation des *Polyeders* – auch als *Delegation*.

### 1.14.7 Anwendungsbeispiel: Ein UML-Modell der Universität

In Abbildung 1.26 ist das konzeptuelle Schema einer Universitätsanwendung als UML-Klassendiagramm gezeigt. Das Diagramm verwendet alle Merkmale von Klassendiagramms, die wir bis jetzt kennen gelernt haben: Klassen mit Attributen und Operationen; eine Aggregation in Form einer Komposition; Beziehungen zwischen den Klassen mit Funktionalitäten, Rollen und definierter Navigierbarkeit. In Kapitel 9 erweitern wir das Modell um weitere Konzepte und erläutern Schritt für Schritt wie es in Java-Code umgesetzt werden kann.

Das Modell der Universität beschreibt *Studenten*, die *Vorlesungen* hören und *Prüfungen* ablegen. Vorlesungen werden von *Professoren* gehalten und haben andere Vorlesungen als Vorraussetzungen. Eine Student wird in einer Prüfung von einem Professor über eine Vorlesung geprüft. Ein Professor hat mehrere *Assistenten*, die für ihn arbeiten. Die nächsten Abschnitte beschreiben Klassen und Beziehungen der Modellierung im Detail.

Ein *Student* wird durch eine eindeutige Matrikelnummer, einen Namen und das aktuelle Semester repräsentiert. Zwei Operationen dienen dazu den Notenschnitt und die Anzahl der Wochenstunden eines Studenten abzufragen. Eine Komposition ordnet die Prüfungen eines Studenten diesem exklusiv zu. Von einem Studenten aus kann man auf die von ihm gehörten Vorlesungen über die gerichtete Beziehung *hören* zugreifen.

Auch eine *Vorlesung* ist eindeutig identifizierbar, in diesem Fall über die Vorlesungsnummer. Zwei andere Attribute speichern ihren Titel und die Semesterwochenstunden. Eine Methode erlaubt es, die Anzahl der Hörer abzufragen und eine andere, die Durchfallquote zu bestimmen. Die selbstgerichtete Beziehung *voraussetzen* modelliert die Tat-

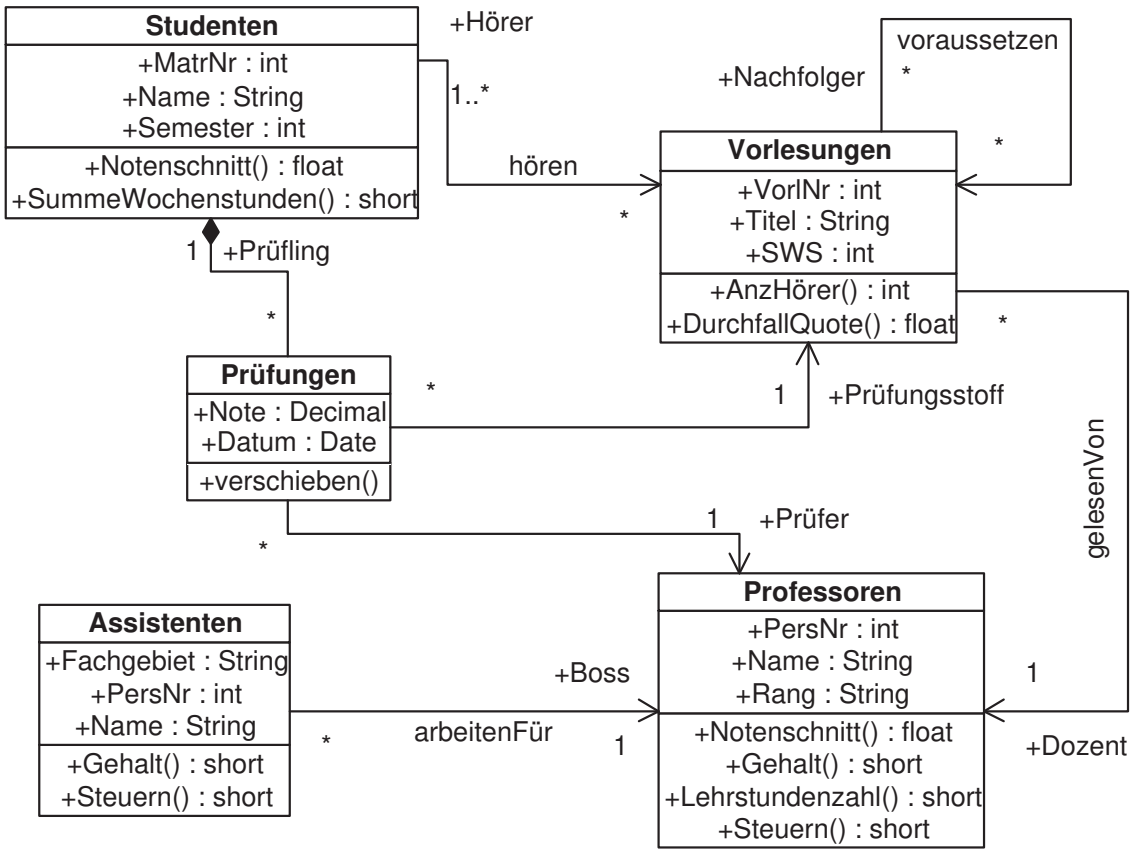



Abbildung 1.26: Die konzeptuelle Modellierung der Universität in UML

sache, dass Vorlesungen auf anderen Vorlesungen aufbauen. Der Dozent ist von einer Vorlesung aus über die Beziehung *gelesenVon* erreichbar.

Eine *Prüfung* ist definiert durch eine Note und ein Datum. Zusätzlich bestehen gerichtete Beziehungen zu einer Vorlesung, die mit einer Rolle als Prüfungsstoff bezeichnet ist und zu einem Professor als Prüfer. Jede Prüfung gehört exklusiv zur einem Studenten, dem Prüfling. Die Operation *verschieben* erlaubt es, den Prüfungstermin zu verlegen.

Ein *Professor* ist modelliert durch eine Personalnummer, einen Namen und einen Rang. Operationen erlauben es, den Notenschnitt der Prüfungen eines Professors, sein Gehalt, die Lehrstundenzahl und die Höhe seiner Steuerzahlungen zu erfragen.

Ein *Assistent* hat ebenfalls eine Personalnummer und einen Namen zusätzlich aber noch ein Fachgebiet in dem er forscht. Gehalt und Steuern lassen sich über zwei Methoden bestimmen. Ein Assistent hat einen Professor als Boss für den er arbeitet.

### 1.14.8 UML jenseits von Klassendiagrammen

Zum Abschluss der UML-Diskussion wollen wir noch kurz auf die weitergehenden Möglichkeiten von UML zur Modellierung komplexer Anwendungsszenarien und Objektinteraktionen eingehen. Die bisher durchgeführte strukturelle Modellierung von Objektklassen dient als Grundlage für die Implementierung von komplexeren Funktionsabläufen, bei der u.U. viele Objekte unterschiedlichen Typs miteinander kooperieren.

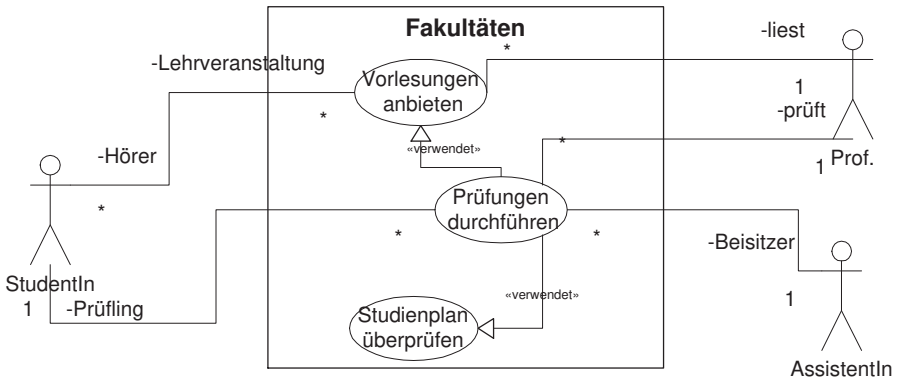



Abbildung 1.27: Identifizierung von Anwendungsfällen und Akteuren

**Anwendungsfalldiagramme** UML bietet auch für die dem Software-Entwurf vorgelagerte Phase der Anforderungsanalyse eine graphische Modellierungsmethode zur Definition von *Anwendungsfällen* (engl. *use cases*). Darin werden die wesentlichen Komponenten des zu erstellenden Informationssystems sehr abstrakt identifiziert. Anwendungsfälle sollen typische Anwendungssituationen dokumentieren, die nach der Anforderungsanalyse die weitere Softwareentwicklung begleiten. Die graphische Darstellung der Anwendungsfälle erleichtert die Kommunikation mit den Kunden und den späteren Anwendern des Systems und fördert das Verständnis der Entwickler für die spezielle Anwendung. Erste Schritte lassen sich besser mit konkreten Beispielen beschreiben als mit abstrakteren allgemeinen Fällen. Während der Modellierung und Softwareentwicklung dienen sie als Testfälle, um die Tauglichkeit des Systems zu überprüfen.

Das Augenmerk bei der Modellierung von Anwendungsfällen liegt auf der Identifikation der Akteure, die in diesen Anwendungsfällen miteinander bzw. mit dem System interagieren. Wir haben dies exemplarisch und auf einer sehr hohen Abstraktionsstufe für unser Universitäts-Informationssystem in Abbildung 1.27 dargestellt. Hier sind die bei der Abhaltung von Vorlesungen und Prüfungen interagierenden Akteure, nämlich die *Studenten*, *Professoren* und *Assistenten*, hervorgehoben.

Natürlich muss die gezeigte graphische Darstellung der Anwendungsfälle durch entsprechenden erläuternden Text begleitet werden, damit die intendierte Funktionalität des Softwaresystems hinreichend genau beschrieben ist.

**Interaktionsdiagramme** Die Anwendungsfall-Modellierung geschieht auf einer sehr hohen Ebene, die die geplante Nutzung und damit eine Außensicht auf das Informationssystems zum Ziel hat. Nachdem in der darauffolgenden Entwurfsphase die Objekttypen und deren Beziehungen festgelegt wurden, können die informell spezifizierten Anwendungsfälle detaillierter modelliert werden. Dazu gibt es in UML die *Interaktionsdiagramme*. Diese dienen dazu das Zusammenwirken von Objekten zu verdeutlichen. Einen Teilausschnitt des Anwendungsfalls aus Abbildung 1.27, nämlich die Durchführung von Prüfungen, ist in Abbildung 1.28 dargestellt. Die am Ablauf beteiligten Objekte werden horizontal nebeneinander dargestellt. Ein Pfeil zwischen zwei Objekten bezeichnet einen Kommunikationsvorgang – im Normalfall ist dies ein Methodenaufruf. Die Operationen sind nach ihrer Reihenfolge von oben nach unten geordnet.

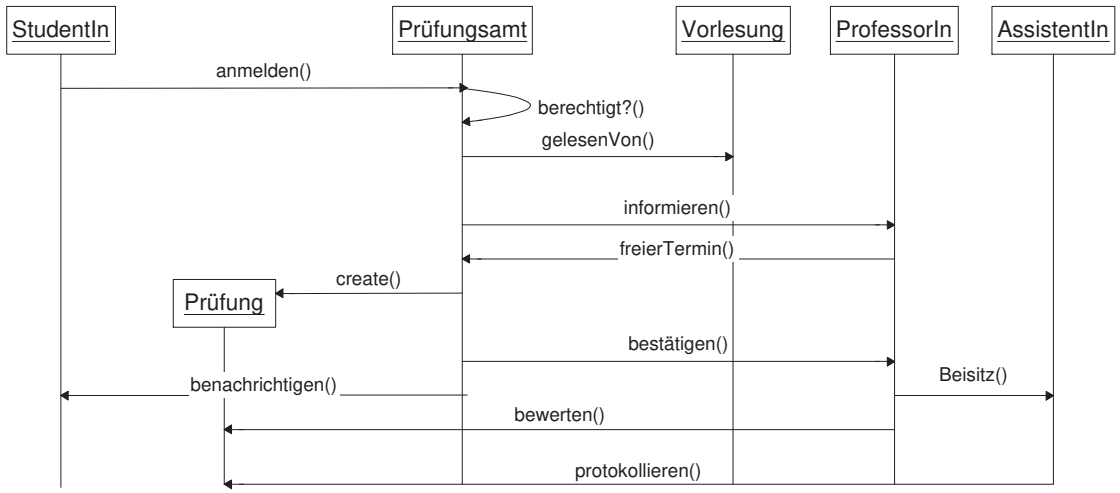



Abbildung 1.28: Das UML-Interaktionsdiagramm zur Anmeldung und Durchführung einer Prüfung

Das Interaktionsdiagramm in Abbildung 1.28 beginnt mit der Anmeldung eines/r Student/in, der sich prüfen lassen möchte. Vom Prüfungsamt wird die zu prüfende Vorlesung „befragt“, um die/den lesenden Professor/in für die Prüfung zu benachrichtigen. Der Professor meldet einen freien Termin, der dann bestätigt und dem Prüfling mitgeteilt wird. Das neue Objekt vom Typ *Prüfung* wird vom Prüfungsamt erstellt. Anschließend wählt der Professor einen Assistenten für den Beisitz aus, der später die Protokollierung übernimmt. Der Professor bewertet die Prüfung.

## 1.15 Übungen

**1.1** Basierend auf den Klassen *Person* und *Stadt* und der Objektbank aus Abbildung 1.14 ist die Typ-Konsistenz folgender Ausdrücke zu überprüfen:

```
donald.ehePartner = cityOfLA.buergermeister;
cityOfLA.buergermeister.ehePartner = cityOfLA.buergermeister;
cityOfLA.name = cityOfLA.buergermeister.ehePartner.lebtIn;
cityOfLA.name = cityOfLA.buergermeister.ehePartner.name;
```

Beachten Sie aber, dass die Aufgabe darin besteht, nur die Typ-Konsistenz zu überprüfen. Es ist dabei irrelevant, ob die Zuweisungen sinnvoll sind oder nicht.

**1.2** Betrachten Sie nochmals die Objektbank aus Abbildung 1.14. Verifizieren Sie folgenden Ausdruck:

```
donald.ehePartner.alter;
```

Ist dieser Ausdruck typ-konsistent? Was passiert zur Ausführungszeit?

**1.3** Wir haben den Unterschied zwischen Kopier-Semantik (der Werte) und Referenz-Semantik (der Objekte) ausführlich diskutiert. In unserer Objektbank aus Abbildung 1.14 ist die *Stadt*-Instanz namens „Los Angeles“ ein gemeinsames Unterobjekt aller drei *Person*en-Objekte, da alle diese Stadt über das Attribut *lebtIn* referenzieren. Diskutieren Sie den Effekt folgender Zuweisungen:

```
...
donald.lebtIn.buergermeister = donald;
System.out.println(donald.lebtIn.buergermeister.name);
...
mickey.lebtIn.buergermeister = mickey;
System.out.println(donald.lebtIn.buergermeister.name);
```

Warum unterscheidet sich obiges Programmfragment deutlich von den nachfolgenden Ausdrücken:

```
donald.alter = mickey.alter;
System.out.println(donald.alter);
...
mickey.alter = 70;
System.out.println(donald.alter);
```

**1.4** Angenommen, wir wollten sicherstellen dass *mickey* und *donald* immer den gleichen Wert für ihr *alter* haben. Wie kann man das erreichen?

Hinweis: Man definiere sich eine neue Klasse *AlterTyp* und verwende ein gemeinsames Unterobjekt.

**1.5** Definieren Sie die Java-Klassen für die Begrenzungsflächendarstellung von Polyedern (siehe Abbildung 1.23) und instanziieren Sie als Beispiel einen Einheits-Quader.

**1.6** Basierend auf der Objektbank aus Übung 1.5 sollen die folgenden Konzepte diskutiert werden:

- Gemeinsame Unterobjekte (shared subobjects)
- implizite Referenzierung
- implizite Dereferenzierung
- Referenz- versus Kopier-Semantik

**1.7** Entwickeln Sie ein sinnvolles Beispiel für ein Array-Objekt als gemeinsames Unterobjekt. Nehmen Sie beispielsweise den Objekttyp *Person* und modellieren Sie das Attribut *kinder*.

**1.8** Modellieren Sie eine Aggregationshierarchie in Java. Als Beispiel verwenden Sie das Objekt *Fahrrad*, das aus den in Abbildung 1.29 gezeigten Teilobjekten besteht.

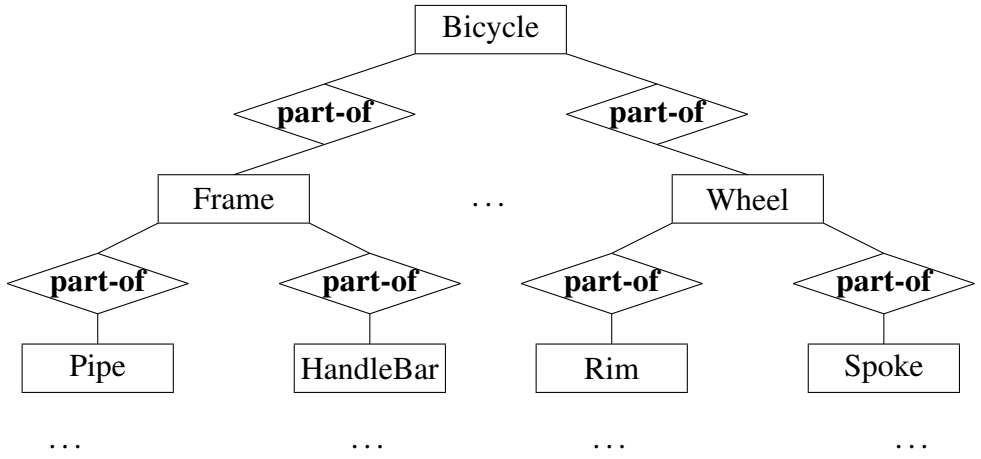



Abbildung 1.29: Aggregations-/Kompositions-Hierarchie eines Fahrrads

## 1.16 Bibliographie

Der Urvater aller objektorientierten Programmiersprachen ist die Sprache Simula-67, die von Dahl, Myrhaug und Nygaard (1970) entworfen wurde. Simula hatte schon viele Ähnlichkeiten zu Java. Auf Simula aufbauend wurde die dynamisch typisierte Sprache Smalltalk-80 von Goldberg und Robson (1983) entworfen. Smalltalk hat eine ähnliche Referenz-Semantik wie Java, außer dass in Smalltalk auch Werte (außer kleinen Integern) als identifizierbare und referenzierbare Objekte behandelt werden. Hinsichtlich der Semantik (nicht der Syntax) ist Eiffel – von Meyer (1992) entworfen – der Sprache Java sehr ähnlich. Auch das objektorientierte Datenmodell GOM, das von Kemper und Moerkotte (1994) als Datenbankmodell entworfen wurde, ist der Sprache Java sehr verwandt. Syntaktisch lehnt sich Java an die Programmiersprachen C und C++ an. C wurde von Kernighan und Ritchie (1988) entworfen. C++ ist eine objektorientierte Erweiterung von C und wurde von Stroustrup (1997) entwickelt. C++ ist in der industriellen Praxis sehr weit verbreitet, da sehr viel Wert auf Effizienz gelegt wurde. Die neueste objektorientierte Programmiersprache ist C# von Microsoft.

Für eine sehr gute Referenz die systematische Entwicklung objektorientierter Programme verweisen wir auf das Buch von Liskov und Guttag (2000). Arnold und Gosling (1998) geben eine gute Einführung in die Programmiersprachenkonstrukte von Java – die objektorientierte Entwurfsmethodik wird aber weniger stark betont. Eine sehr umfassende Abhandlung (fast) aller Java-Konstrukte und vordefinierter Programmbibliohteken findet sich in dem umfangreichen Buch von Deitel und Deitel (1999).

Die objektorientierte Entwurfsmethodik wird in den folgenden Büchern systematisch eingeführt: Booch (1991), Budd (1991), Coad und Yourdan (1991a), Coad und Yourdan (1991b). Diese unterschiedlichen Notationen wurden von Grady Booch (1998) zur Unified Modeling Language, die in diesem Buch verwendet wird, vereinheitlicht. Einen deutschsprachigen Zugang zur objektorientierten Modellierung mit UML findet man in dem Buch von Oestereich (1999). Das Buch von Brügge und Dutoit (2004) ist eine sehr gute und ausführliche Referenz für die objektorientierte Modellierung mit UML sowie für systematische Methoden des Software Engineering.

Die neueren Fortschritte des objektorientierten Software-Engineering erläutern Man-

drioli und Meyer (1992). Gamma et al. (1995) haben ein Buch über objektorientierte Software-Muster (engl. *patterns*) veröffentlicht.

# 2. Verhalten von Objekten

Wir haben bereits angemerkt, dass Objekttypen neben der strukturellen Repräsentation ihrer Instanzen auch deren Verhalten modellieren. Die Menge der Operationen, die auf den Instanzen der Klasse angewendet werden kann, stellt die *Schnittstelle* dar, mit der der Zustand der Objekte abgefragt bzw. manipuliert werden kann. Die Nutzung dieser im Objekttyp definierten Operationen realisiert das Geheimnisprinzip (engl. *information hiding*), wonach die Objekte als abstrakte Datentypen genutzt werden können, ohne dass man detaillierte Kenntnis über deren interne Repräsentation (bzw. Implementierung der Operationen) benötigt. Alles was man wissen muss, ist die Schnittstellen-Spezifikation, die aus den *Signaturen* der Objekte besteht.

## 2.1 Klassifikation der Operationen

Damit ein Objekttyp sinnvoll genutzt werden kann, müssen Operationen festgelegt werden, mit denen man Objekte initialisieren, ihren Zustand abzufragen und ihren Zustand modifizieren kann. Im Design des Objekttyps müssen die Programmierer den richtigen Kompromiss zwischen einer zu umfangreichen Schnittstelle, die unübersichtlich und überladen ist, und einer zu kargen Schnittstelle, bei der sinnvolle Operationen fehlen bzw. nur durch Kombinationen anderer Operationen erreicht werden können, finden.

Gemäß ihrer Semantik unterscheiden wir drei Klassen von Operationen:

1. *Konstruktoren*
   Diese Art von Operation wird genutzt um Instanzen des Objekttyps zu erzeugen. Ein Beispiel ist die Instanziierung eines *Studenten*-Objekts: *Student xeno = new Student(24002, "Xenokrates", 18)*. Dieser Aufruf erzeugt einen neuen Studenten mit Namen Xenokrates, der die Matrikelnummer 24002 hat und in 18. Semester studiert.

2. *Beobachter-Funktionen (engl. accessor)*
   Beobachter sind Funktionen, die Informationen über den derzeitigen Zustand des Objekts zurückliefern – ohne diesen Zustand zu verändern. Man sagt daher auch, dass sie frei von *Seiteneffekten* sind.

   Ein Beispiel dafür ist die Operation *notenschnitt()*, die für ein *Studenten*-Objekt dessen Notenschnitt liefert. Der Aufruf läuft wie folgt: *xeno.notenschnitt()*. Das Objekt *xeno* auf dem die Methode aufgerufen wird hat eine besondere Rolle: Man bezeichnet es als *Empfänger-Objekt* (engl. *receiver*) des Methodenaufrufes. In einigen objektorientierten Programmiersprachen bezeichnet man den Operationsaufruf deshalb auch als *message passing*, weil man dem Empfänger die Nachricht zukommen lässt, eine bestimmte Operation auszuführen.

3. *Mutatoren (engl. mutator)*
   Mutatoren sind Operationen, die den internen Zustand des Objekts verändern, auf

dem sie aufgerufenen werden. Beispielsweise verändert die Operation *xeno.belegeVorlesung(grundzuege)* den Studenten *xeno* indem die Vorlesung *grundzuege* zur Menge der von *xeno* gehörten Vorlesungen hinzugefügt wird.

Abhängig davon, ob man die Objekte einer Klasse verändern kann (durch Mutatoren oder explizite Zuweisung an öffentliche Instanzvariablen), bezeichnen wir eine Klasse als *mutierbar* (engl. *mutable*) oder *nicht-mutierbar* (engl. *immutable*).

Bei der Klassifizierung einer Klasse muss man vorsichtig sein: Sobald Instanzvariablen öffentlich sichtbar sind (Schlüsselwort *public*) kann man diesen neue Werte oder Referenzen zuweisen, was implizit einer Mutation des Objekts entspricht. Somit dürfen *nicht-mutierbare* Klassen nur *private* Instanzvariablen – also solche, die nicht von außen erreichbar sind – oder konstante Instanzvariablen haben, die zwar lesbar aber nicht änderbar sind.

Wir hatten bereits darauf hingewiesen, dass die Java-Strings ein Beispiel für einen *nicht-mutierbaren* Objekttyp darstellen. Die Leser mögen sich dies anhand der Java-Dokumentation verdeutlichen.

## 2.2 Operationen

Neben der strukturellen Repräsentation der Objekte legt die Klassendefinition auch deren Schnittstelle fest. Wir wollen dies am Beispiel unserer geometrischen Objekttypen *Vertex* und *Quader* demonstrieren. In UML werden die Operationen nur als Signaturen angegeben, wie dies in Abbildung 2.1 dargestellt ist.

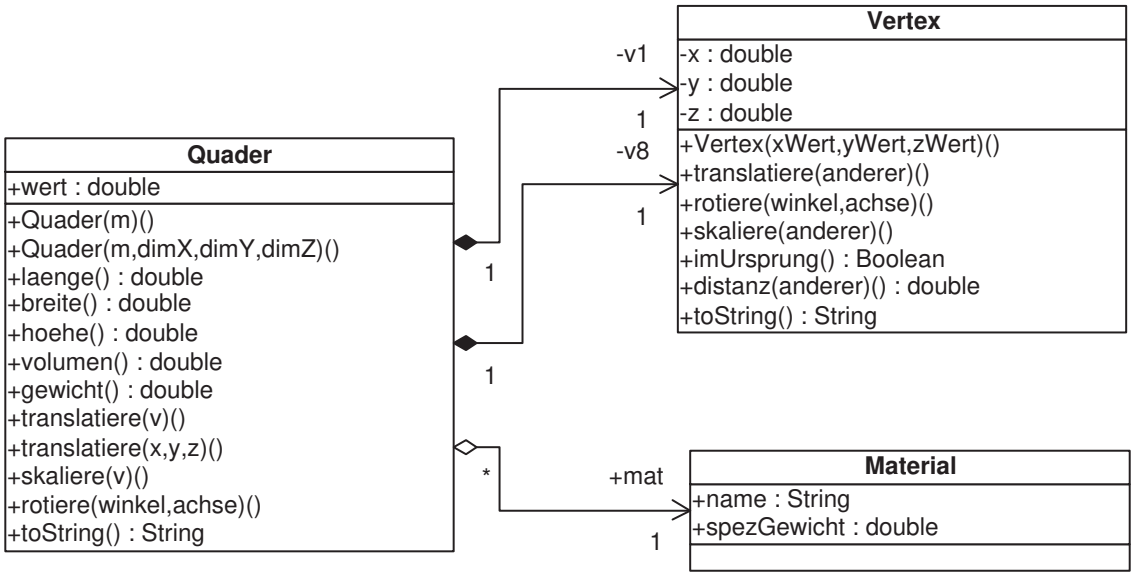



Abbildung 2.1: Die Modellierung des Verhaltens von Quader- und Vertex-Objekten

Für die sinnvolle Nutzung eines *Quader*-Objekts benötigt man beispielsweise die Beobachter-Funktionen *volumen* und *gewicht*. Die geometrischen Transformatoren sind Mutatoren. Dazu gehören *translatieren* und *rotieren*, um die Lage im Raum zu ändern, sowie *skalieren* um die Größe zu variieren. Bevor wir aber das Verhalten der Klasse *Quader*

```
1  public class Vertex {
2      double x;
3      double y;
4      double z;
5
6      public Vertex(double xWert, double yWert, double zWert) {
7          this.x = xWert; this.y = yWert; this.z = zWert;
8      }
9
10     public void translatiere(Vertex anderer) {  // Mutator
11         this.x = this.x + anderer.x;   // kürzer: x += anderer.x;
12         this.y = this.y + anderer.y;   //         y += anderer.y;
13         this.z = this.z + anderer.z;   //         z += anderer.z;
14     }
15
16     public void skaliere(Vertex anderer) {
17         // ...
18     }
19
20     public void rotiere(double winkel, char achse) {
21         // ...
22     }
23
24     public boolean imUrsprung() {  // liegt der Wert sehr nahe bei (0,0,0) ?
25         double epsilon = 0.000000001;  // Präzisionsgrenze
26         return ((this.x < epsilon) && (this.x > -epsilon) &&
27                 (this.y < epsilon) && (this.y > -epsilon) &&
28                 (this.z < epsilon) && (this.z > -epsilon));
29     }
30
31     public double distanz(Vertex anderer) { // Distanz zwischen this und anderer
32         double dx, dy, dz;
33         dx = this.x - anderer.x;
34         dy = this.y - anderer.y;
35         dz = this.z - anderer.z;
36         return Math.sqrt(dx * dx + dy * dy + dz * dz);
37     }    // sqrt ist in Math definiert
38
39     public String toString() {
40         return ("x:␣" + this.x + ",␣y:␣" + this.y + ",␣z:␣" + this.z);
41     }
42
43 } // public class Vertex
```

Abbildung 2.2: Definition der Klasse *Vertex* mit Operationen

implementieren, wollen wir die Klasse *Vertex*, auf der die Definition des Quaders aufbaut, vervollständigen. Die Klassendefinition des *Vertex* inklusive den wichtigsten Operationen ist in Abbildung 2.2 dargestellt.

Es ist offensichtlich, dass die Operationsdefinition aus zwei Teilen besteht:

1. Die Operationsdeklaration legt die Signatur, also den Namen, die formalen Parameter und den Rückgabetyp der Operation fest.

2. Die Operationsimplementierung enthält den Java-Code, der die Operation realisiert.

In unserem Beispiel haben wir drei Mutatoren, nämlich die geometrischen Transformatoren *translatiere*, *skaliere* sowie *rotiere* definiert. Weiterhin haben wir einige Beobachter-Funktionen, wie *imUrsprung* und *distanz* definiert. Erstere überprüft, ob der Vertex (gemäß dem derzeitigen Zustand der Koordinaten) sehr nahe beim Ursprung liegt. Die Operation *distanz* ermittelt die Distanz des Empfänger-Vertex, auf dem die Operation aufgerufen wird, zu einem anderen *Vertex*, der als Argument übergeben wird.

Die Definition einer Operation – bestehend aus Signatur und Codierung – sieht syntaktisch wie folgt aus:

```
[public|private|protected] <ErgebnisTyp> <Name>(<ParameterTypListe>) {
   // Implementierung
}
```

Die einzelnen Bestandteile haben die folgende Bedeutung:

- Der *<Name>* identifiziert die Operation innerhalb der Klasse. Man beachte, dass jede Klasse ihren eigenen Namensraum hat, so dass die gleichnamigen Operationen von *Vertex* bzw. *Quader* sich nicht gegenseitig ins Gehege kommen. Sie sind zwar logisch verwandt, haben aber aus der Sicht des Java-Compilers nichts miteinander zu tun. Anders verhält es sich da bei gleichnamigen Operationen innerhalb derselben Klasse. Auch hierfür haben wir ein Beispiel: Den Konstruktor *Vertex* gibt es zweimal – einmal mit drei Parametern und einmal ohne Parameter. Dies bezeichnen wir als Überladung (engl. *overloading*) der Operation. Es handelt sich hierbei um zwei unterschiedliche Operationen, die vom Compiler anhand der Parameter beim Aufruf auseinander gehalten werden können. Mehr dazu erklären wir in dem Abschnitt 2.6.

- Der *<ErgebnisTyp>* legt den Typ des Rückgabe-Objekts bzw. des Rückgabewerts fest. Falls es keine Rückgabe gibt, wird **void** angegeben.

- Die *<ParameterTypListe>* gibt die Anzahl und die Typen der Parameter an.

- Die Implementierung erfolgt zum Schluss – innerhalb der geschweiften Klammern. Man beachte, dass ein guter objektorientierter Entwurf im Allgemeinen dafür sorgt, dass die einzelnen Operationen sehr einfach und knapp realisiert werden können.

- Die so genannten *access modifier* **[public|private|protected]** legen die Sichtbarkeit einer Operation fest. Eine öffentliche Operation ist von überall aufrufbar und wird mit dem Schlüsselwort **public** gekennzeichnet. Die privaten Operationen sind nur innerhalb der Klassendefinition aufrufbar – es handelt sich hierbei um Hilfsroutinen zur Implementierung der öffentlichen Operationen. Die mit **protected** angegebenen Operationen können auch in einer Unterklasse verwendet werden. Diese Sichtbarkeitsregeln werden in Abschnitt Abschnitt 2.4 nochmals diskutiert, da sie essentiell für das Konzept des *Information Hidings* ist.

Bei der Implementierung werden die Parameter wie lokale Variablen behandelt. Für Operationen ohne zusätzliche Parameter wird eine leere *<ParameterListe>* – allerdings mit Klammern () – übergeben, so dass sich in Java der Aufruf einer Operation vom Zugriff auf eine Instanzvariable immer syntaktisch unterscheidet.

### 2.2.1 Aufruf einer Operation

Der Aufruf einer Operation geschieht immer durch das so genannte *Message-Passing*, indem man ein *Empfängerobjekt* beauftragt, die Operation auszuführen. Das Empfängerobjekt wird durch einen Ausdruck vor dem Punkt (engl. *dot*) des Operationsaufrufs bestimmt. Wir wollen dies anhand unseres *Vertex*-Beispiels demonstrieren:

```
Vertex meinVertex = new Vertex(1.0,0.0,0.0);;
Vertex translationsVertex = new Vertex(0.0,2.0,2.0);
...
meinVertex.translatiere(translationsVertex);
```

Das Objekt, das von der Variablen *meinVertex* referenziert wird, ist hierbei das Empfängerobjekt. Der *Vertex*, der von *translationsVertex* referenziert wird, dient als zusätzlicher Parameter. In diesem Beispiel sind beide Argumente vom gleichen Typ.

Im Implementierungscode kann man über die implizit deklarierte Variable *this* auf das Empfängerobjekt zugreifen. In unserem Aufruf von *translatiere* verweist *this* also auf das Empfängerobjekt, also den *Vertex*, auf den die lokale Variable *meinVertex* verweist. *this* ist ein impliziter Parameter, dessen Typ dem Objekttyp entspricht, in dem die Operation definiert wurde.

Die *dot*-Notation kann den Empfänger über beliebig lange Referenzketten (Pfadausdrücke) bestimmen. Dies ist in folgendem Beispiel gezeigt:

```
meinQuader.v1.translatiere(translationsVertex);
```

Hier ist das Empfängerobjekt der *Vertex*, auf den die Instanzvariable *v1* des von der Variablen *meinQuader* referenzierten *Quader*-Objekts verweist.

Im obigen Beispiel waren nur Instanzvariablen in dem Pfadausdruck. Das ist aber nicht zwingend; es können auch selbst wieder Operationsaufrufe darin vorkommen. Wir wollen dies an der Beispielklasse *Person* demonstrieren:

```
public class Person {
    public int alter;
    public Person ehePartner;
    public Person mutter;
    // ...
    public Person schwiegerMutter() {
       return ehePartner.mutter;
    }
    // ...
}
...
Person mickey;
...
mickey.schwiegerMutter();
mickey.schwiegerMutter().ehePartner;
```

Im ersten Beispiel wird die Operation *schwiegerMutter* direkt bei *mickey* aufgerufen. Beim zweiten Beispiel wird die Operation *schwiegerMutter* aufgerufen und anschließend der Ehepartner abgefragt – wenn alles konsistent modelliert ist, sollte das natürlich der Schwiegervater von *mickey* sein.

Man sollte allerdings bei tatsächlichem Code nicht vergessen, auf Nullwerte zu überprüfen. Im zweiten Beispiel würde ein Laufzeitfehler auftreten, falls das Attribut *ehePartner* bei *mickey* oder die *mutter* dieses *ehePartners* auf *null* gesetzt ist.

## 2.3 Divide and Conquer

Die objektorientierte Modellierung zeichnet sich durch die *divide-and-conquer*-Vorgehensweise aus. Man zerteilt Aufgaben in kleinere Aufgaben, die von einfacheren Basis-Klassen erledigt werden, um darauf aufbauend funktional komplexere Klassen definieren zu können, die die Basisklassen verwenden. Dies ist auch bei unserem Beispiel erkennbar. Nachdem wir *Vertex* und *Material* definiert haben, können wir jetzt die aus geometrischer Sicht komplexere Klasse *Quader* implementieren – siehe Abbildung 2.3.

Wir haben hier die drei Mutatoren *translatiere*, *rotiere* sowie *skaliere* definiert. Weiterhin gibt es Beobachter-Funktionen wie *hoehe*, *laenge*, *breite*, *volumen* und *gewicht*. Die Bedeutung dieser Operationen sollte klar sein. Interessanter ist die Art und Weise, wie diese Operationen implementiert wurden. Sie machen sich die Funktionalität der Klasse *Vertex* zunutze. Die Klasse *Quader* ist dadurch ein Klient der Klasse *Vertex*. In allen Implementierungen wird die eigentliche Arbeit an die über die Instanzvariablen *v1*, ..., *v8* referenzierten *Vertex*-Objekte *delegiert*. Man betrachte beispielsweise die Implementierung des Beobachters *laenge*, bei der die Distanz zwischen den beiden Eckpunkten *v1* und *v2* zurückgegeben wird. Diese wird einfach durch den Aufruf `v1.distanz(v2);` ermittelt. Genauso werden die beiden Beobachter *breite* und *hoehe* realisiert. Auch die Mutatoren stützen sich voll und ganz auf die Funktionalität der *Vertex*-Klasse. Beispielsweise wird der Quader translatiert indem jeder Eckpunkt verschoben wird.

## 2.4 Information Hiding

Objektorientierte Sprachen erlauben es dem Programmierer, den internen Zustand der Objekte zu verstecken (engl. *information hiding*). Als Klient kann man dann nur mit Hilfe der in der Klasse vordefinierten *public* Operationen den Zustand erfragen bzw. den Zustand ändern. Dies bezeichnet man auch als *Verkapselung* (engl. *encapsulation*).

Wir wollen die Vorteile der Verkapselung an einem Beispiel-Programmfragment erläutern. Gemäß der bisherigen Klassendefinition des *Quader*s ist es immer noch möglich, die wohl-definierten geometrischen Transformationen *translatiere*, *skaliere* und *rotiere* zu umgehen, indem man einzelne Eckpunkte modifiziert. Das könnte wie folgt passieren:

```
   Quader meinKomischerQuader;
   Material gold = new Material();   // neues Material
   gold.name = "Gold";
   ...
   meinKomischerQuader = new Quader(gold); // goldener Quader
   ...
(1)meinKomischerQuader.v1.x = 0.5;
(2)meinKomischerQuader.v1.y = 0.5;
(3)meinKomischerQuader.v1.z = 0.5;
```

Dieses Programmfragment transformiert den durch den Konstruktor wohl-geformten *Quader* in einen semantisch inkonsistenten Zustand, der in Abbildung 2.4 gezeigt ist.

Auf der linken Seite der Abbildung ist der semantisch konsistente Quader direkt nach der Instanziierung gezeigt; rechts der semantisch inkonsistente Quader, der nach der Ausführung der drei Zuweisungen (1), (2) und (3) resultiert.

```
1  public class Quader {
2      Vertex v1, v2, v3, v4, v5, v6, v7, v8;
3      Material mat;
4      double wert;
5
6      public Quader(Material m) {  // Konstruktor: Einheitswürfel wird kreiert
7          this.v1 = new Vertex(0.0,0.0,0.0);  this.v2 = new Vertex(1.0,0.0,0.0);
8          this.v3 = new Vertex(1.0,1.0,0.0);  this.v4 = new Vertex(0.0,1.0,0.0);
9          this.v5 = new Vertex(0.0,0.0,1.0);  this.v6 = new Vertex(1.0,0.0,1.0);
10         this.v7 = new Vertex(1.0,1.0,1.0);  this.v8 = new Vertex(0.0,1.0,1.0);
11         this.mat = m;
12         this.wert = 39.99;  // fiktiver Wert
13     }
14     public double laenge() {                // Observer-Funktion
15         return this.v1.distanz(this.v5);
16     }
17     public double breite() {
18         return this.v1.distanz(this.v2);
19     }
20     public double hoehe() {
21         return this.v1.distanz(this.v4);
22     }
23     public double volumen() {
24         return this.laenge() * this.hoehe() * this.breite();
25     }
26     public double gewicht() {
27         return this.volumen() * this.mat.spezGewicht;
28     }
29     public void translatiere(Vertex v) {    // Mutator
30         this.v1.translatiere(v);    this.v2.translatiere(v);
31         this.v3.translatiere(v);    this.v4.translatiere(v);
32         this.v5.translatiere(v);    this.v6.translatiere(v);
33         this.v7.translatiere(v);    this.v8.translatiere(v);
34     }
35     public void translatiere(double xWert, double yWert, double zWert) {
36         this.translatiere(new Vertex(xWert,yWert,zWert));
37                     // delegieren an obige translatiere Op.
38     }
39     public void skaliere(Vertex v) {
40         // ...
41     }
42     public void rotiere(double winkel, char achse) {
43         // ...
44     }
45     public String toString() { // observer
46        return("Quader␣der␣Dimension␣" + this.laenge() + "␣X␣" +
47                                         this.breite() + "␣X␣" + this.hoehe());
48     }
49
50 } // public class Quader
```

Abbildung 2.3: Definition der Klasse *Quader* mit Operationen

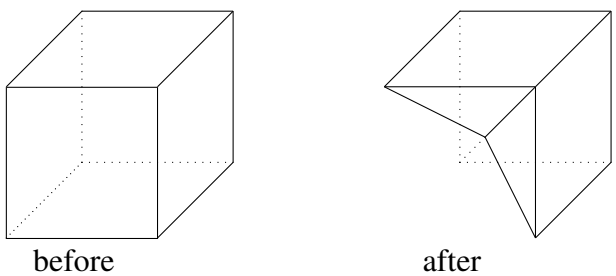



Abbildung 2.4: Semantisch inkonsistente Transformation eines *Quader*-Objekts

Die semantische Inkonsistenz resultiert daraus, dass ein einzelner Eckpunkt des Quaders transformiert wurde – ohne dass die anderen sieben Eckpunkte in analoger Weise modifiziert wurden. Um die semantische Konsistenz zu garantieren, müssen wir sicherstellen, dass immer alle Eckpunkte in konsistenter Weise transformiert werden. Dies leisten ja gerade die Mutatoren *translatiere*, *rotiere* und *skaliere* – solange sie korrekt implementiert wurden, was wir hier voraussetzen.

Um die Verkapselung gewährleisten zu können, kann man in Java die Sichtbarkeit von Attributen und Operationen über die so genannten *access modifier* steuern.

- **public:** Öffentliche Komponenten sind für alle Klienten zugreifbar und werden als *public* angegeben.
- **private:** Die verborgenen Komponenten werden als *private* spezifiziert und sind nur für den Code innerhalb der Klasse sichtbar.
- **protected:** Die mit *protected* angegebenen Komponenten sind wie bei *private* nicht allgemein sichtbar. Im Unterschied zu *private* können Unterklassen aber auf die als *protected* gekennzeichneten Komponenten ihrer Oberklassen zugreifen.
- **package:** Der Default – wenn also nichts angegeben ist – ist der access modifier *package*. Code innerhalb desselben Packages hat Zugriff auf die Komponenten.

In unserem Quader-Beispiel wurde die Inkonsistenz durch den Zugriff auf die Eckpunkte verursacht. Um dies auszuschließen sollte man die Attribute *v1*, ..., *v8* als *private* deklarieren. Allerdings sollte man sich dann noch lange nicht in Sicherheit wähnen, dass die als *private* vermeintlich geschützten Komponenten dann nicht nach außen sichtbar werden können. Dazu muss zusätzlich sichergestellt werden, dass keine Operation die entsprechenden Referenzen nach außen gibt. Weiterhin muss sichergestellt werden, dass keine anderen Referenzen (z.B. im Zuge der Initialisierung) mehr existieren, über die die privaten Datenobjekte erreicht werden können.

Wir wollen jetzt die Sichtbarkeit der Komponenten der Klasse *Quader* zusammenfassend darstellen:

```
public class Quader {
    private Vertex v1, v2, v3, v4, v5, v6, v7, v8;
    public Material mat;
    public double wert;

    public Quader(Material m) {  // Einheitsquader kreieren
        // ...
    }
    public double laenge() {
        // ...
    }
    public double breite() {
        // ...
    }
    public double hoehe() {
        // ...
    }
```

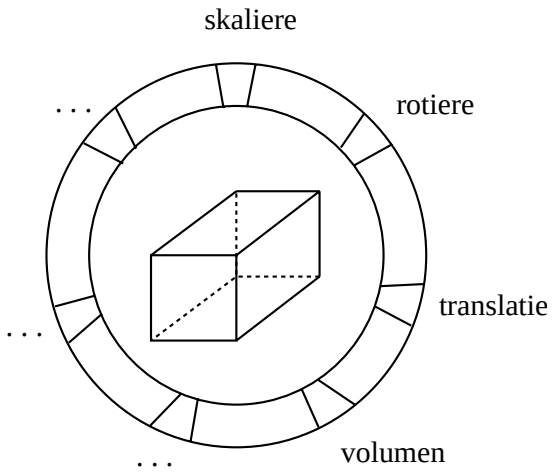



Abbildung 2.5: Schematische Darstellung der *Quader*-Verkapselung

```
    public double volumen() {
        // ...
    }
    public double gewicht() {
        // ...
    }
    public void translatiere(Vertex v) {
        // ...
    }
    public void translatiere(double xWert, double yWert, double zWert){
        // ...
    }
    public void skaliere(Vertex v) {
        // ...
    }
    public void rotiere(double winkel, char achse) {
        // ...
    }
    public String toString() {
        // ...
    }
} // public class Quader
```

Als Klienten des Typs kann man jetzt nur die öffentlichen Operationen wie *laenge*, ..., *rotiere* und *toString* aufrufen. Somit ist das Programmfragment, das die semantische Inkonsistenz des Quaders verursachte, jetzt nicht mehr möglich, da der Zugriff auf das Attribut *v1* jetzt verweigert wird. Der Zugriff innerhalb der Klasse *Quader* ist auch weiterhin möglich, so dass beispielsweise der Programmcode der Operation *translatiere* aus Abbildung 2.3, der auf die Eckpunkte *v1*, ..., *v8* zugreift, nach wie vor gültig ist. Die hiermit erzielte Verkapselung der *Quader*-Objekte ist in Abbildung 2.5 graphisch dargestellt.

In Java kann man leider nicht direkt zwischen dem lesenden und dem schreibenden Zugriff auf eine Instanzvariable differenzieren. Wenn man die Instanzvariable als *public* deklariert, ist sie damit sowohl lesbar als auch schreibbar. Wenn man nur den lesenden

Zugriff erlauben wollte, geht dies nur über den Umweg einer entsprechenden Leseoperation:

```
class Quader {
    private double wert;
    ...
    public double wert() {
       return this.wert;
    }
}
```

Hier haben wir die Instanzvariable *wert* als *private* verborgen und das Lesen über die Operation gleichen Namens erlaubt.

Syntaktisch unterscheidet sich der Zugriff auf eine Instanzvariable aber von der Invokation einer Operation. Der Operationsaufruf hat immer – auch bei leerer Parameterliste – die nachfolgenden Argumentklammern (...):

```
double wieViel;
Quader meinQuader;
...
wieViel = meinQuader.wert();
```

## 2.5 Initialisierung

In Abschnitt 1.5 haben wir gezeigt, wie man Objekte mit dem *new*-Operator instanziiert. Die Instanzvariablen sind zunächst auf *null*-Referenzen bzw. bestimmte Werte vorinitialisiert. Anschließend haben wir die Attribute initialisiert – mühevoll durch explizite Zuweisungen an die Instanzvariablen. Dies hat verschieden Nachteile:

1. *Verletzung der Verkapselung*
   Um beispielsweise die Eckpunkte instanziieren und dann den Attributen *v1*, ..., *v8* zuweisen zu können, benötigt man den Zugriff auf diese Attribute. Genau dies verstößt aber gegen das Prinzip der Verkapselung, da man sich die entsprechenden *Vertex*-Objekte ja beispielsweise in entsprechenden lokalen Variablen merken kann und später darauf zugreifen oder sie sogar manipulieren könnte. Dies könnte dann wiederum zu einem inkonsistenten *Quader* führen.

2. *Umständliche Vorgehensweise*
   Wenn man sehr häufig neue Objekte instanziieren und initialisieren muss, ist die bisher gezeigte Art der Initialisierung viel zu mühselig. Dies wird auch in dem Programmfragment aus Abschnitt 1.5 deutlich, da wir jeden Eckpunkt explizit instanziieren und dessen Koordinaten jeweils mühsam setzen mussten.

Eine wesentlich elegantere Vorgehensweise besteht darin, einen *Konstruktor* für die Klasse zu definieren. Ein Konstruktor wird genau so wie eine normale Operation definiert – außer dass sein Name dem Namen der Klasse entsprechen muss und dass Konstruktoren keinen Rückgabetyp (auch nicht *void*) haben. In unserem Beispiel heißt der Konstruktor also genau wie die Klasse *Quader*.

In unserem Beispiel wollen wir einen Quader also zunächst zum Einheitswürfel vorinitialisieren. In nachfolgenden geometrischen Transformationen kann man dann mittels den Operationen *skaliere*, *rotiere* und *translatiere* den Quader in jede gewünschte Größe, Lage und Orientierung im drei-dimensionalen Raum transformieren. Weiterhin wollen wir das *mat* Attribut zu einem dem Konstruktor übergebenen *Material* initialisieren. Das Attribut *wert* wird auf einen fiktiven Wert von 39.99 gesetzt. Der Konstruktor kann dann wie folgt definiert werden:

```
public class Quader {
    private Vertex v1, v2, v3, v4, v5, v6, v7, v8;
    public Material mat;
    public double wert;

    public Quader(Material m) {  // Konstruktor: Einheitswürfel
        this.v1 = new Vertex(0.0,0.0,0.0);
        this.v2 = new Vertex(1.0,0.0,0.0);
        this.v3 = new Vertex(1.0,1.0,0.0);
        this.v4 = new Vertex(0.0,1.0,0.0);
        this.v5 = new Vertex(0.0,0.0,1.0);
        this.v6 = new Vertex(1.0,0.0,1.0);
        this.v7 = new Vertex(1.0,1.0,1.0);
        this.v8 = new Vertex(0.0,1.0,1.0);
        this.mat = m;
        this.wert = 39.99;  // fiktiver Wert
    }
    // ...
} // public class Quader
```

Wenn der Klient einen benutzer-definierten Konstruktor verwendet, indem beim Aufruf des *new*-Operators die entsprechenden Parameter übergeben werden, wird dieser automatisch direkt nach der Instanziierung auf dem neuen Objekt aufgerufen. Es passieren also beim Aufruf des *new*-Operators zwei Vorgänge direkt nacheinander statt:

1. Instanziierung des neuen Objekts als „leere Hülle“
2. Aufruf des Konstruktors auf diesem neuen Objekt

Man beachte, dass der Compiler nur dann einen Argument-losen Default-Konstruktor anlegt, wenn kein anderer definiert wird. Um trotzdem noch einen Argument-losen Konstruktor zur Verfügung zu haben, muss man diesen explizit wieder definieren. Wir wollen dies am Beispiel der *Vertex*-Klasse zeigen. In Abbildung 2.2 war dieser Klasse ein drei-Argument-Konstruktor zugeordnet worden, der die drei Koordinaten des Vertex' entsprechend setzt. Wenn wir zusätzlich einen Argument-losen Konstruktor haben möchten, der alle Koordinaten auf 0.0 initialisiert, so muss man diesen explizit definieren:

```
class Vertex {
    // ...
    public Vertex() {
       this.x = 0.0;  // Initialisierung könnte auch an den
       this.y = 0.0;  // anderen Konstruktor delegiert werden:
```

```
        this.z = 0.0;  // this(0.0,0.0,0.0);
    }
    // ...
}
```

Wenn ein Konstruktor Argumente benötigt, müssen diese beim Aufruf des *new*-Operators übergeben werden. Für unser Quader-Beispiel geht das wie folgt:

```
Quader meinQuader;
Material gold;
gold = new Material();
   ...
meinQuader = new Quader(gold);
   ...
```

Hier wird also ein Einheitswürfel kreiert, dessen *mat*-Attribut auf das neu instanziierte und über *gold* referenzierte *Material*-Objekt verweist.

## 2.6 Overloading

Wir wollen uns als Motivation für das Überladen (engl. *overloading*) von Operationen nochmals den Konstruktor der Klasse *Quader* anschauen – siehe Abschnitt 2.5. Man kann sich leicht vorstellen, dass es sinnvoll ist unterschiedliche Konstruktoren zur Verfügung zu haben. Man denke etwa an einen zusätzlichen Konstruktor, der nicht den Einheitswürfel erstellt, sondern einen Quader mit vorgegebenen $x$-, $y$- und $z$-Dimensionen. Man kann durch *overloading* eine weitere Operation (in diesem Fall einen weiteren Konstruktor) gleichen Namens definieren, so dass man dann wahlweise die „am besten passende“ Operation verwenden kann. Man sollte natürlich nur semantisch eng verwandte Operationen überladen. Das Überladen führt also dazu, dass man in demselben Namensraum mehrere Operationen gleichen Namens hat. (Vorerst gehen wir davon aus, dass jede Klasse ihren eigenen Namensraum definiert, was später in Kapitel 3 durch Vererbung noch revidiert wird). Allerdings müssen sich diese gleich benannten Operationen in mindestens einem der nachfolgenden Kriterien unterscheiden, damit der Compiler entscheiden kann, welche Operation denn nun tatsächlich gemeint ist:

- Anzahl der Parameter
- Typen der Parameter

In unserem Beispiel wollen wir in der Klasse *Quader* also einen weiteren Konstruktor hinzufügen. Das geht wie folgt:

```
public class Quader {
    private Vertex v1, v2, v3, v4, v5, v6, v7, v8;
    public Material mat;
    public double wert;

    public Quader(Material m) {  //  Konstruktor: Einheitswürfel
        ... // Implementierung wie vorher
```

```
    }

    public Quader(Material m, double dimX, double dimY, double dimZ) {
        this.v1 = new Vertex(0.0,0.0,0.0);
        this.v2 = new Vertex(dimX,0.0,0.0);
        this.v3 = new Vertex(dimX,dimY,0.0);
        this.v4 = new Vertex(0.0,dimY,0.0);
        this.v5 = new Vertex(0.0,0.0,dimZ);
        this.v6 = new Vertex(dimX,0.0,dimZ);
        this.v7 = new Vertex(dimX,dimY,dimZ);
        this.v8 = new Vertex(0.0,dimY,dimZ);
        this.mat = m;
        this.wert = 39.99;  // fiktiver Wert
    }
    // ...
} // end class Quader
```

Die Nutzung wird dann in folgendem Programmfragment illustriert:

```
   Quader meinQuader;
   Quader deinQuader;
   Material gold;
      ...
(1)meinQuader = new Quader(gold);                 // goldener Einheitswürfel
(2)deinQuader = new Quader(gold, 2.0, 3.0, 5.0); // Goldbarren Quader
   ...                                            // der Dimension 2 × 3 × 5
```

In Ausdruck (1) wird der erste Konstruktor ausgeführt, was sehr einfach anhand der Anzahl der Parameter zu erkennen ist. Im zweiten Ausdruck wird der zweite Konstruktor ausgeführt, dem vier Parameter zu übergeben sind (ein *Material* und drei *double*).

Das Überladen ist aber nicht nur für Konstruktoren sinnvoll – hier ist es allerdings besonders wichtig, da Java verlangt, dass alle Konstruktoren denselben Namen (nämlich den der Klasse) haben müssen. Ein weiteres sinnvolles Beispiel für das Überladen einer Operation ist der Mutator *translatiere*. Dieser Operation kann man entweder einen *Vertex* als Argument übergeben oder in der überladenen Variante drei *double*-Werte. Dies kann dann wie folgt realisiert werden:

```
public class Quader {
    // ...
    public void translatiere(Vertex v) {
        this.v1.translatiere(v);    this.v2.translatiere(v);
        this.v3.translatiere(v);    this.v4.translatiere(v);
        this.v5.translatiere(v);    this.v6.translatiere(v);
        this.v7.translatiere(v);    this.v8.translatiere(v);
    }
    public void translatiere(double x, double y, double z) {
        // delegiere die Arbeit an die andere translatiere-Operation
        this.translatiere(new Vertex(x,y,z));
    // ...
} // public class Quader
```

In diesem Fall wurde in der zweiten Variante der translatiere-Operation die schon vorher implementierte *translatiere*-Operation mit dem *Vertex*-Argument ausgenutzt. Dazu wurden die drei *double*-Parameter in einen *Vertex* „gesteckt“ und an die entsprechende *translatiere*-Operation übergeben. Dies geschah durch den Aufruf *this.translatiere(new Vertex(...));*

Das folgende Programmfragment demonstriert die unterschiedliche Nutzung dieser beiden überladenen *translatiere*-Operationen:

```
Quader meinQuader = new Quader(...);;
Vertex translatiereVertex = new Vertex(0.0,3.0,2.0);
   ...
meinQuader.translatiere(translatiereVertex);
meinQuader.translatiere(30.0, 2.0, 1.75);
   ...
```

Wiederum ist es anhand der Anzahl der Parameter offensichtlich, welche der beiden *translatiere*-Varianten gemeint ist.

Man sollte noch betonen, dass das Überladen wirklich nur für semantisch eng verwandte Varianten einer Operation verwendet werden sollte.

## 2.7 Statische Operationen

Manchmal erscheint es etwas seltsam wenn man Operationen einem Empfänger-Objekttyp zuordnen muss, so dass man die Operation auf einem Empfängerobjekt aufzurufen hat. Ein Beispiel dafür haben wir sogar schon gesehen: Die Operation *distanz* berechnet den Abstand zweier *Vertex*-Objekte. Diese Operation war in Abbildung 2.2 so realisiert worden, dass eines der beiden *Vertex*-Objekte für die Berechnung zuständig ist (nämlich das Empfängerobjekt):

```
Vertex p1, p2;
double d;
   ...
d = p1.distanz(p2);
   ...
```

Für manche Programmierer mag der folgende „symmetrische“ Aufruf natürlicher sein:

```
   ...
d = distanz(p1, p2);
   ...
```

Java bietet das Konzept der statischen Methoden an um dies umzusetzen. Statische Operationen benötigen kein Empfängerobjekt sondern lassen sich auf der Klasse, in der sie definiert sind, aufrufen. Da es keinen Empfänger für den Aufruf gibt, darf man in statischen Operationen natürlich nicht auf Instanzvariablen zugreifen sondern nur auf Klassenattribute (statische Variablen).

Für unser Beispiel könnten wir also unsere Klasse *Vertex* um die statische Methode *distanz* erweitern.

```
public class Vertex {
    ...
    public static double distanz(Vertex erster, Vertex zweiter) {
         return erster.distance(zweiter);
    }
}
```

Um dann die *distanz* zwischen zwei Punkten zu berechnen, ruft man die statische Methode wie folgt auf:

```
Vertex a = new Vertex(0.0,0.0,0.0);
Vertex b = new Vertex(3.0,0.0,0.0);
System.out.println(Vertex.distanz(a,b));
```

Der Aufruf einer statischen Operation geschieht also wie der Zugriff auf statische Variablen indem man den Klassennamen als Präfix (hier *Vertex*) verwendet (siehe auch Abschnitt 1.13). Wir haben übrigens schon in der Implementierung der Operation *distanz* der Klasse *Vertex* eine statische Operation verwendet. Die vor-definierte Java-Klasse *Math* enthält allerlei nützliche Konstanten (wie *Math.PI*) und mathematische Routinen als statische Operationen. Unter anderem auch die Operation *sqrt* zur Berechnung der Quadratwurzel, die wir für die Berechnung des Abstandes zweier Punkte benötigt haben.

Eine weitere sehr häufig vorkommende Nutzung einer statischen Funktion besteht in der Definition der Routine *main* in einer Klassendefinition. Jede Klasse kann diese Routine enthalten, damit man die Klasse ausführen kann. Wir wollen diese statische Operation am Beispiel unserer *Quader*-Klasse vorführen:

```
public class Quader {
   ... // wie zuvor
   public String toString() { // observer
       return("Quader der Dimension " + this.laenge() + " X " +
              this.breite() + " X " + this.hoehe());
    }

    public static void main(String args[]) {
       Material eisen = new Material("Eisen", 0.89);
       Quader meinQuader = new Quader(eisen);
       Quader andererQuader = new Quader(eisen);
       andererQuader.translatiere(new Vertex(3.5,2.5,4.5));
       andererQuader.translatiere(2.0,3.0,4.0);
       System.out.println(meinQuader);   // impliziter Aufruf von toString
       System.out.println("meinQuader hat das Gewicht: "
                          + meinQuader.gewicht());
       System.out.println(andererQuader);
       System.out.println("andererQuader hat das Gewicht: "
                          + andererQuader.gewicht());
       Material carbon = new Material("Carbon", 0.75);
       meinQuader.mat = carbon;
       System.out.println("meinQuader hat jetzt das Gewicht: "
                          + meinQuader.gewicht());
    }
```

```
} // public class Quader
```

Eine statische Operation kann – wie bereits erwähnt – auf keine Instanzvariablen via *this* zugreifen. Intuitiv kann man sich das so vorstellen: Nicht-statische Operationen (in Java Terminologie auch Methoden genannt) werden innerhalb einer Instanz der Klasse ausgeführt und haben Zugriff auf die Komponenten (Instanzvariablen, Operationen, etc.) dieser Instanz – und auch die Klassen-Attribute und statischen Operationen. Im Gegensatz dazu werden statische Operationen außerhalb der Instanzen ausgeführt und haben deshalb nur Zugriff auf die statischen Variablen (Klassen-Attribute) und andere statische Operationen. Allerdings kann man im Code der statischen Operation durchaus Instanzen derselben oder anderer Klassen erzeugen, wir wir dies in der *main*-Operation der Klasse *Quader* auch demonstriert haben. Die statische Operation ist dann in der Rolle des Klienten der Instanzen dieser Klasse.

Die Operation *main* wird implizit aufgerufen, wenn man die Java Virtual Machine anweist eine Klasse auszuführen:

```
javac Quader.java
java Quader
```

Die Ausgabe ist dann wie folgt:

```
Quader der Dimension 1.0 X 1.0 X 1.0
meinQuader hat das Gewicht: 0.89
Quader der Dimension 1.0 X 1.0 X 1.0
andererQuader hat das Gewicht: 0.89
meinQuader hat jetzt das Gewicht: 0.75
```

## 2.8 Parameter-Übergabe

Bislang haben wir uns auf ein intuitives Verständnis der Parameterübergabe beim Operationsaufruf verlassen. Wir wollen dieses wichtige Thema hier detaillierter diskutieren.

Parameter werden benutzt, um Information aus der Umgebung des Operationsaufrufs in die Operation zu übergeben. Diese Information besteht entweder aus Werten oder aus Objekten. Beide werden zur Laufzeit durch die Auswertung der jeweiligen Ausdrücke bestimmt, die beim Aufruf der Operation angegeben wurden. Die Ausdrücke, die die Parameter bestimmen, müssen natürlich typ-konsistent sein mit der Typeinschränkung des formalen Parameters – also dem in der Signatur der Operation deklarierten Parameter. Das Ergebnis der Auswertung des Ausdrucks wird dem formalen Parameter der Operation zugewiesen. Im Falle der „normalen“ nicht-statischen Operationen haben wir zusätzlich zur Parameterliste auch noch den einen impliziten Parameter, der dem Empfängerobjekt des Operationsaufrufs entspricht. Dieser Parameter hat immer denselben Namen: *this*. Aus der Sicht der Parameterübergabe wird dieser Empfänger jedoch genauso behandelt wie jeder andere Parameter auch.

Wir wollen die Parameterübergabe am Beispiel der *distanz*-Operation, die der Klasse *Vertex* zugeordnet ist, demonstrieren:[1]

[1] Wir setzen voraus, dass der Code in einem Kontext ausgeführt wird, in dem die Instanzvariablen *v1* und *v7* sichtbar sind. Beispielsweise in der *main*-Methode der Klasse *Quader*.

```
meinQuader.v1.distanz(deinQuader.v7);
```

Der Empfänger des Aufrufs wird durch Auswerten des Ausdrucks *meinQuader.v1* bestimmt.

Nun muss man zwei verschiedene Arten der Parameterübergabe unterscheiden: *call by value* und *call by reference*. *Call by value* kommt zum Einsatz, wenn primitive Typen als Argument verwendet werden. In diesem Fall werden die Argumente kopiert. Bei Objekttypen bezeichnet man die Semantik der Parameterübergabe als *call by reference*, da in diesem Fall nur eine Referenz auf das Objekt übergeben wird und keine Kopie des Objekts. Ändert nun die Operation Eigenschaften beim übergebenen Objekt wirkt sich dies auch außerhalb der Operation auf das Objekt aus – es wurde ja gerade keine automatische Kopie erstellt. Möchte man ein übergebenes Objekt nur in der Operation verändern, muss man es selbst kopieren.

## 2.9 Ausnahmebehandlung

Java besitzt eine gute Ausnahmebehandlung (engl. *exception handling*), die es den Programmierern erlaubt außergewöhnliche Zustände während der Programmausführung abzufangen und entsprechenden Ausnahmebehandlungscode auszuführen, um die Ausnahme zu umgehen oder zu beheben. Natürlich kann man Ausnahmesituationen auch ganz einfach durch entsprechende *if*-Ausdrücke abfangen. Dies führt jedoch in vielen Fällen zu unübersichtlichem Code. Die Exception-Handling-Mechanismen von Java erlauben es, die Ausnahmebehandlung modular und verständlich zu gestalten. Man sollte aber davor warnen, die Ausnahmebehandlung für „Programmiertricks“ zu missbrauchen. Sie sollte nur für außergewöhnliche und seltene Fehlerbehebung verwendet werden.

Wir wollen die Ausnahmebehandlung anhand eines Beispiels vorstellen: Die Klasse *Quader* enthält den Konstruktor zur Initialisierung des Quaders zu einer vorgegebenen Länge, Breite und Höhe. Wenn eine dieser Dimensionen 0 (oder zumindest sehr nahe an 0) ist, so handelt es sich sicherlich um einen Fehler. Deshalb wollen wir in diesem Fall eine Ausnahme (Exception) signalisieren – in Java-Terminologie „throw an exception“. *Exception* ist eine vordefinierte Java-Klasse, die eine reichhaltige Funktionalität realisiert. Wir werden jedoch nur die eine Möglichkeit nutzen, der Exception eine bestimmte Nachricht (einen String) zu übergeben, die diese Ausnahme textuell erklärt. Mit der Methode *getMessage()* kann der Fehlerbehandlungscode diese Nachricht erlangen.

```
public class Quader {
    private Vertex v1, v2, v3, v4, v5, v6, v7, v8;
    public Material mat;
    public double wert;

    public Quader(Material m, double dimX, double dimY, double dimZ)
        throws Exception // wenn es kein richtiger Quader ist
    {
        double epsilon = 0.000000001;  // Präzisionalimit
        if ((dimX < epsilon) && (dimX > -epsilon) || // prüfe ob eine
            (dimY < epsilon) && (dimY > -epsilon) || // Dimension zu
            (dimZ < epsilon) && (dimZ > -epsilon))   // nahe zur 0 ist
```

```
                throw new Exception("Illegale Quader-Dimension");
        this.v1 = new Vertex(0.0,0.0,0.0);
        this.v2 = new Vertex(dimX,0.0,0.0);
        this.v3 = new Vertex(dimX,dimY,0.0);
        this.v4 = new Vertex(0.0,dimY,0.0);
        this.v5 = new Vertex(0.0,0.0,dimZ);
        this.v6 = new Vertex(dimX,0.0,dimZ);
        this.v7 = new Vertex(dimX,dimY,dimZ);
        this.v8 = new Vertex(0.0,dimY,dimZ);
        this.mat = m;
        this.value = 39.99;  // fiktiver Wert
    }
    // ...
} // public class Cuboid
```

Wenn jetzt ein neuer *Quader* instanziiert wird, kann der Programmcode diese Art von Exception „fangen“ (engl. *catch*). Dies geschieht mit dem try/catch-Ausdruck wie folgt:

```
try { ... } catch (...) { ... }
```

Wir könnten beispielsweise folgenden Code in der *main*-Routine der Klasse *TesteAusnahme* ausführen:

```
public class TesteAusnahme {
    public static void main(String[] args) {
        Material eisen = new Material("Eisen", 0.89);
        Quader meinQuader;
        try {
             meinQuader = new Quader(eisen,0.0,3.0,9.0);
        }
        catch (Exception e) {
             System.out.println(e.getMessage());
        }
    }
}
```

Anstatt eine generische *Exception* zu werfen, hätte man sich hier auch entschließen können, eine spezifischere Ausnahme, nämlich die *IllegalArgumentException* zu werfen.

## 2.10 Übungen

**2.1** Augment the type definition of *Vertex* by the following operations:

- *distanceToOrigin*, a function that computes the distance of the receiver *Vertex* object to the origin of the coordinate system
- *rotate*, the mutator that rotates the receiver *Vertex* about the specified axis by an angle that is passed as a parameter
- *onStraightLine*, a Boolean function that checks whether the receiver *Vertex* lies on a straight line determined by two parameter objects of type *Vertex*

- *onPlane*, a Boolean function that checks whether the receiver *Vertex* lies on the plane formed by the parameter *Vertex* instances

You should specify the three declarations as well as the implementations of these operations.

**2.2** Define the object type *Tetrahedron*. Make sure that you provide a satisfactorily complete operational interface. In particular, you should include the analogous operations that are (verbally) specified in Exercise 2.6.

**2.3** In Exercise 1.5 the structural representation of a boundary representation schema was developed in Java. Augment the skeleton schema by operations for geometric transformations.

**2.4** Devise the kernel types of the constructive solid geometry representation of solid objects. Sketch the combinational operations union, difference, intersection, etc.

**2.5** In computer graphics books, the concept of *homogeneous coordinates* is introduced. Homogeneous coordinates require that a *Vertex* is represented by four, instead of three, coordinates. Define the type *Vertex*, which implements the homogeneous coordinates. Implement the three geometric transformations—i.e., translatiere, rotate, and scale—as matrix multiplications. Of course, you have to define the type *Matrix* first. Hint: Use lists of lists to implement the type *Matrix*.

**2.6** Augment the type definition of *Cuboid*. In particular, add the following operations:

- *surface*, yielding the surface value of the *Cuboid*
- *scale*, which scales the size of the *Cuboid*—be careful that the *scale* operation is properly implemented for *Cuboids* that have an arbitrary orientation in the coordinate system
- *center*, which determines the *Vertex* representing the center of the *Cuboid*
- *diagonal*, which computes the length of the diagonal of the *Cuboid*
- *minDistance*, a function that determines the minimum distance of a parameter *Vertex* from the receiver *Cuboid*—e.g., for assembly planning in order to avoid collisions
- *minDistance*, the overloaded function that determines the minimum distance of any boundary points of two *Cuboid* objects—one being passed as a parameter, the other one being the receiver object. Additionally, provide an alternative static operation for *minDistance* in, say, class `Measurer`. You may reuse the type-associated operation *minDistance* for implementing the free operation *minDistance*.

**2.7** The *call by value* or, in the case of object parameters, *call by reference* parameter passing sometimes leads to surprising effects. Consider the following operation *wrongSwap*:

```
class Swapper {
    static void wrongSwap(Cuboid v1, Cuboid v2) {
```

```
        Cuboid tmp;
        tmp = v1;
        v1 = v2;
        v2 = tmp;
    }
}
```

What happens when *wrongSwap* is invoked in the following program fragment *(smallCube* and *largeCube* being variables referring to *Cuboid* objects):

```
...
Cuboid smallCube, largeCube;
...
if (smallCube.volume() > largeCube.volume())
    Swapper.wrongSwap(smallCube,largeCube);
...
```

Illustrate your example on a sample object base. In particular, consider the two different cases:

1. *smallCube* refers to a Cuboid smaller than the one referred to by *largeCube*.
2. *smallCube* refers to a Cuboid larger than the one referred to by *largeCube*.

**2.8** Add exception handling code to the class definitions of *Vertex* and *Cuboid*.

## 2.11 Annotated Bibliography

The language features of Java discussed in this chapter rely on the object-oriented programming languages Simula 67 [Dahl und Nygaard (1966)] and its descendants, e.g., Eiffel [Meyer (1988)], Smalltalk-80 [Goldberg und Robson (1983)], C++ [Stroustrup (1990)]. Most of the relevant literature concerning object-oriented programming languages has already been cited at the end of the preceding chapter.

Liskov and Guttag Liskov und Guttag (1986) provide a very good treatment of program design using object-oriented features, such as information hiding, bottom-up design. Their work is—at least partially—founded on the programming language Clu [Liskov et al. (1981)]. Meyer [Meyer (1987)] discusses the advantages of the object-oriented languages with respect to program reusability. Another contribution on this topic was published by Micallef [Micallef (1988)].

Stefik and Bobrow [Stefik und Bobrow (1986)] survey object-oriented languages and modeling in artificial intelligence applications.

Liskov and Snyder [Liskov und Snyder (1979)] discuss exception handling in the context of the language Clu.

Overloading of operations has been introduced in a variety of programming languages, in particular Ada [Booch (1983), Institut (1983)]. Ada also provides means for exception handling.

Derret, Kent, and Lyngbaek [Derret, Kent und Lyngbaek (1985)] discuss the behavioral aspects of the object model Iris—which is heavily influenced by the functional data model DAPLEX [Shipman (1981)].

# 3. Vererbung

Eine Klasse definiert Objekte mit ähnlichen Charakteristika. Ähnliche Charakteristika bedeutet hierbei, dass alle Objekte des gleichen Typs dieselbe Menge von Attributen zur Repräsentation ihres Zustands und dieselbe Menge an Operationen zur Beschreibung ihres Verhaltens haben. Wenn Objekte sich auch nur geringfügig hinsichtlich ihrer Zustands- oder Verhaltensbeschreibung unterscheiden, muss man sie durch gesonderte Klassen modellieren. Bislang sind wir aber nicht in der Lage, den Zusammenhang dieser nur geringfügig unterschiedlichen Klassen auszudrücken. Dies wird durch die Vererbung erreicht, die es verschiedene Unterklassen erlaubt von einer gemeinsamen Oberklasse Attribute und Operationen erben.

## 3.1 Motivation

Alle Objekte einer Klasse haben dieselben Attribute und Operationen – der Zustand der Objekte, d.h., die Belegung der Attribute kann sich natürlich zwischen Instanzen unterscheiden. Die bislang eingeführten Typkonzepte erlauben es nicht, Variationen einer Klasse zu modellieren, ohne die gemeinsamen Komponenten in allen Variationen redundant zu definieren. Wir wollen dies am Beispiel der *Person* mit den Attribute *name*, *alter* und *ehePartner* sowie der Operation *heiraten* demonstrieren:

```
public class Person {
    public String name;
    public int alter;
    public Person ehePartner;
    // ...
    public Person(String n, int a) {
        this.name = n; this.alter = a;
    }
    public void heiraten(Person partner) {
        this.ehePartner = partner;
    }
    // ...
} // public class Person
```

Beachten Sie, dass wir bewusst eine Hälfte der Implementierung von *heiraten*, nämlich das Setzen des *ehePartner*-Attributs bei der geheirateten Person, ausgelassen haben – mehr dazu in Übung 3.1.

Unter der Annahme, dass einige, aber nicht alle Personen auch Angestellte sind, wird ein zusätzlicher Objekttyp *Angestellter* definiert. Diese Klasse hat alle für „normale“ Personen schon relevanten Komponenten, wie *name*, *alter*, *ehePartner*, und zusätzlich die nur für Angestellte relevanten Komponenten, wie *steuerNr*, *gehalt*, *istPensioniert()*, etc.

```
public class Angestellter {
    public String name;
    public int alter;
    public Person ehePartner;
    public int steuerNr;
    public double gehalt;
    public Angestellter boss;

    public Angestellter(String n, int a, int s, double g) {
        this.name = n;
        this.alter = a;
        this.steuerNr = s;
        this.gehalt = g;
    }

    public void heiraten(Person partner) {
        this.ehePartner = partner;
    }

    public boolean istPensioniert() {
        return (this.alter > 64);
    }
} // public class Angestellter
```

Zwei schwerwiegende Problem gibt es mit diesen beiden Klassendefinitionen:

1. *Mangelnde Wiederverwendbarkeit*
   Die Klasse *Person* enthält viele Komponenten, die in der Klasse *Angestellter* in gleicher Form nochmals repliziert wurden. Es wäre vorteilhafter gewesen, die Definition der Klasse *Angestellter* auf der Definition der Klasse *Person* aufzubauen.

2. *Mangelnde Flexibilität*
   Diese Problem der mangelnden Flexibilität ist noch schwerwiegender. Es wird dadurch verursacht, dass die beiden Klassen völlig isoliert voneinander definiert sind, und die Instanzen der beiden Klassen sich nicht gegenseitig vertreten können. Wir wollen dies an dem Attribut *ehePartner* der Klasse *Person* illustrieren. Dieses Attribut ist genau wie das Argument *partner* der Operation *heiraten* auf Objekte vom Typ *Person* eingeschränkt. Dies bedeutet, dass niemand eine/n Angestellte/n heiraten kann. Das folgende Programmfragment illustriert dieses Problem:

   ```
   Angestellter miniMouse;
   Person mickeyMouse;
   ...
   miniMouse.heiraten(mickeyMouse);    // okay, mickeyMouse ist eine Person
   mickeyMouse.heiraten(miniMouse);    // illegal, miniMouse ist keine Person
   ```

   Das Problem resultiert aus der fehlenden Subtypisierung der beiden Klassen *Person* und *Angestellter*. Dem Typsystem ist nicht bekannt, dass Angestellte auch Personen sind – wie es in der Realität der Fall ist.

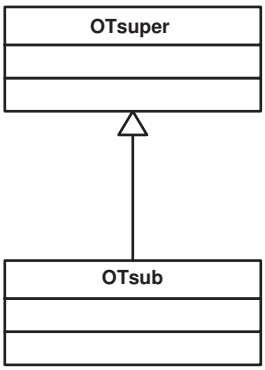



Abbildung 3.1: Graphische Repräsentation der *is-a*-Beziehung in UML

Im Folgenden werden wir Vererbung und Subtypisierung vorstellen. Diese beiden Konzepte lösen beide oben angesprochenen Probleme: Durch die Vererbung wird die Wiederverwendbarkeit von Klassendefinitionen ermöglicht und durch die Subtypisierung wird ermöglicht, dass speziellere Objekte (Angestellte) anstelle von generischeren Objekten (Personen) verwendet werden dürfen.

## 3.2 Allgemeine Idee: Vererbung und Subtypisierung

Die allgemeine Idee hinter der Subtypisierung und der Vererbung ist die Modellierung von *is-a*-Beziehungen zwischen Objekttypen. In der UML-Notation werden *is-a*-Beziehungen zwischen Klassen durch einen Pfeil von der Unterklasse zur Oberklasse dargestellt – siehe Abbildung 3.1. In diesem Schema ist $OT_{sub}$ der Untertyp (engl. *sub type*) des Obertyps (engl. *super type*) $OT_{super}$.

Wenn zwei Objekttypen $OT_{sub}$ und $OT_{super}$ in einer *is-a*-Beziehung zueinander stehen, werden alle Objekte des Untertyps $OT_{sub}$ auch als Objekte des Obertyps $OT_{super}$ angesehen. Somit impliziert die *is-a*-Beziehung eine *Untermengen*-Beziehung zwischen der Menge der Objekte des Untertyps und der Menge der Objekte des Obertyps. Mit der Subtypisierung gehen zwei weitere Aspekte einher: die *Vererbung* aller Komponenten des Obertyps an den Untertyp und die *Substituierbarkeit* der Objekte des Obertyps durch Objekte des Untertyps. Diese beiden Aspekte werden aber später noch sehr detailliert erklärt.

In Java repräsentiert man die *is-a*-Beziehung durch die Erweiterung (engl. *extension*) der Oberklasse in der Unterklasse. Für unser abstraktes Beispiel sieht das wie folgt aus:

```
class OTsuper {
    ...
}


class OTsub extends OTsuper {
    ...
}
```

In der graphischen UML-Notation zeigt der Pfeil der *is-a* Beziehung vom Untertyp zum Obertyp. Dadurch wird die Beziehung „$OT_{sub}$ **is-a** $OT_{super}$“ betont. Diese Sichtweise weist deutlich darauf hin, dass jedes Objekt vom Typ $OT_{sub}$ auch ein Objekt des Typs $OT_{super}$ ist. In dieser Hinsicht wird durch die Ober/Untertyp-Beziehung das Konzept der Generalisierung/Spezialisierung realisiert. Die Objekte des Obertyps sind generischer als die spezielleren Objekte des Untertyps.

In fast allen Objektmodellen – insbesondere auch in Java – wird die Subtypisierung mit der Vererbung gekoppelt. Der Untertyp erbt alle Komponenten des Obertyps – daraus resultiert auch die Java-Notation *extends*. Instanzen des Untertyps besitzen also alle Komponenten des Obertyps sowie die zusätzlichen, spezielleren Komponenten des Untertyps. Da die Vererbung über beliebig viele Hierarchiestufen gehen kann, erben Untertypen natürlich auch alle Komponenten der direkten und indirekten Obertypen ihres Obertyps. Somit ist der Untertyp – etwas gegensätzlich zum Namen *Unter*typ – eine Erweiterung des Obertyps, da er *alle* Komponenten des Obertyps (und möglicherweise noch weitere) besitzt.

Die Vererbung in Kombination mit der Subtypisierung lösen die oben geschilderten Probleme des *Person/Angestellter*-Schemas. Die **is-a**-Beziehung zwischen *Angestellter* und *Person* ist graphisch in Abbildung 3.2 dargestellt.

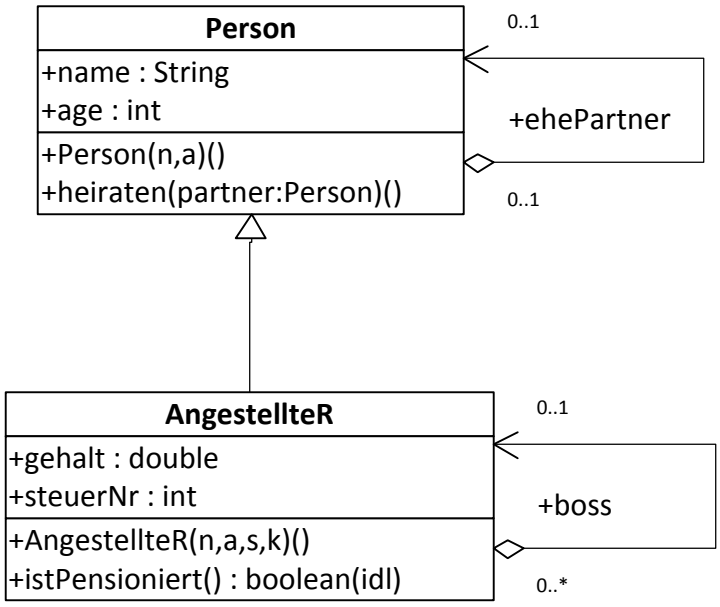



Abbildung 3.2: Die Generalisierung/Spezialisierung

Im Kontext des familiären Lebens, also als „normale“ Personen, besitzen Objekte wie *miniMouse* vom Typ *Angestellter* die Attribute *name*, *alter*, und *ehePartner*. Weiterhin kann ein/e Angestellter in diesem Kontext die Operation *heiraten* ausführen. Wenn dasselbe Objekt *miniMouse* im Kontext des Steuerzahlers, also als *Angestellter*, behandelt wird, sind die zusätzlichen Attribute *steuerNr*, *gehalt* und *boss* von Interesse. In diesem Kontext ist dann auch die Operation *istPensioniert* sinnvoll.

Die Java-Definition der Klasse *Angestellter* als Untertyp/Erweiterung der Klasse *Person* ist nachfolgend gezeigt:

```
class Angestellter extends Person {
    public int steuerNr;
    public double gehalt;
    public Angestellter boss;

    Angestellter(String n, int a, int s, double knete) {
        super(n, a);
        this.steuerNr = s;
        this.gehalt = knete;
    }

    public boolean istPensioniert() {
        return (this.alter > 64);
    }

} // class Angestellter
```

Wir können jetzt zwei Variablen *miniMouse* vom Typ *Angestellter* und *mickeyMouse* vom Typ *Person* deklarieren:

```
Angestellter miniMouse;
Person mickeyMouse;
```

Wir können jetzt neue Objekte der jeweiligen Klassen instanziieren und den Variablen zuweisen. Wir gehen davon aus, dass es in der Klasse *Person* einen geeigneten Konstruktor gibt, dem der *name* und das *alter* übergeben wird:

```
miniMouse = new Angestellter("Mini Mouse", 60, 007, 90000.0);
mickeyMouse = new Person("Mickey Mouse", 50);
```

Bislang haben wir uns bei der Diskussion der Vererbung hauptsächlich auf die Attribute konzentriert; aber die Operationen eines Obertyps werden genauso an den Untertyp vererbt. Deshalb haben Objekte vom Typ *Angestellter* neben der Operation *istPensioniert* beispielsweise auch die Operation *heiraten*, die in der Klasse *Person* definiert wurde.

## 3.3 Substituierbarkeit

Über die Wiederverwendbarkeit von Komponenten der Oberklasse hinaus, bietet die Definition einer Unterklasse auch die zusätzliche Flexibilität, die wir vorher – beispielsweise bei der Diskussion der Operation *heiraten* – vermisst hatten. Wir hatten analysiert, dass die Einschränkung von Parametern und Attributen auf einen *festen* Objekttyp oft zu rigide ist. Dies hatte ja gerade verhindert, dass Personen Angestellte heiraten können, da *ehePartner* (bzw. das Argument *partner* der Operation *heiraten*) auf Objekte vom Typ *Person* eingeschränkt waren. Durch die Subtypisierung werden diese Typeinschränkungen generell flexibler, da ein Objekt des Untertyps überall dort verwendet werden kann, wo ein Objekt des Obertyps gefordert ist. Dieses Prinzip bezeichnet man als *Substituierbarkeit* der Untertyp-Instanzen für Obertyp-Instanzen. Bezogen auf unser Beispiel bedeutet dies, dass man überall dort wo eine *Person* erwartet wird, auch einen *Angestellten* verwenden darf – insbesondere also auch bei der Heirat. Wir wollen dies an unserem Beispiel erläutern:

```
Angestellter miniMouse;
Person mickeyMouse;
...
miniMouse.heiraten(mickeyMouse);    // okay, mickeyMouse ist eine Person
mickeyMouse.heiraten(miniMouse);    // jetzt okay, miniMouse ist
                                    als Angestellter auch eine Person
```

Der erste Aufruf bleibt natürlich gültig. Wir können aber jetzt nicht mehr garantieren, dass die Variable *mickeyMouse* auf eine „normale“ *Person* verweist – es könnte wegen der Substituierbarkeit jetzt auch ein/e *Angestellter* sein. Der Operationsaufruf bleibt aber auf jeden Fall gültig, da das Substituierbarkeitsprinzip auch auf die Empfängerobjekte eines Operationsaufrufes zutrifft. Der zweite Aufruf von *heiraten* ist jetzt gültig, da sie als Argument eine *Person* verlangt. *miniMouse* ist als *Angestellter* aber für eine *Person* substituierbar und kann deshalb als *partner* verwendet werden.

Wir wollen die Typkonsistenz der Substituierbarkeit zunächst auf intuitive Weise diskutieren. Durch die Vererbung entlang der *is-a* Beziehung vom Ober- zum Untertyp erbt der Untertyp *alle* Komponenten – also Struktur und Verhalten – des Obertyps. Deshalb sind alle Operationsaufrufe, die auf einem Obertyp-Objekt erlaubt sind, auch auf einem Untertyp-Objekt möglich. Salopp ausgedrückt kann man sagen, dass Untertyp-Objekte mindestens so viel können wie Obertyp-Objekte – meistens sogar mehr, da ja noch zusätzliche Attribute und Operationen im Untertyp definiert sein können. Bezogen auf unser Beispiel gilt folgendes:

- Objekte vom Typ *Angestellter* enthalten *alle* Attribute der Klasse *Person*, also, *name*, *alter* und *ehePartner*. Zusätzlich haben *Angestellter*-Objekte noch die Attribute *steuerNr*, *gehalt* und *boss*.

- Jede *Angestellter*-Instanz versteht *alle* Operationen, die eine *Person* bearbeiten kann, also z.B. *heiraten*. Zusätzlich haben Objekte vom Typ *Angestellter* noch speziellere Untertyp-spezifische Operationen, nämlich *istPensioniert* in unserem Beispiel.

Schematisch ist dies in Abbildung 3.3 dargestellt.

Auf der linken Seite sind die Komponenten aufgezeigt, die ein Objekt $o_{Person}$ vom Typ *Person* besitzt. Rechts sind die Komponenten gezeigt, die ein Objekt $o_{Angestellter}$ vom Typ *Angestellter* vorzuweisen hat. Wir sehen, dass dies eine echte Obermenge der Komponenten von $o_{Person}$ ist. Somit kann ein Klient nicht erkennen, ob es sich um eine „normale“ Person oder um eine/n Angestellte/n handelt, solange man nur auf die Komponenten einer Person zugreift. Deshalb ist die Typkonsistenz immer gewährleistet, wenn wir Objekte vom Typ *Angestellter* dort substituieren, wo Objekte vom Typ *Person* gefordert sind. Dies war insbesondere bei der Heirat der Fall – dort kann sowohl der Empfänger der Operation *heiraten* als auch das Argument (also der *partner*) ein Objekt vom Typ *Angestellter* sein, obwohl in der Signatur beide Argumente auf den Typ *Person* eingeschränkt sind.

## 3.4 Terminologie

In diesem Abschnitt wollen wir die Terminologie, die im Zusammenhang mit der Vererbung und Subtypisierung gebräuchlich ist, zusammenfassen. Wir wollen dies anhand der

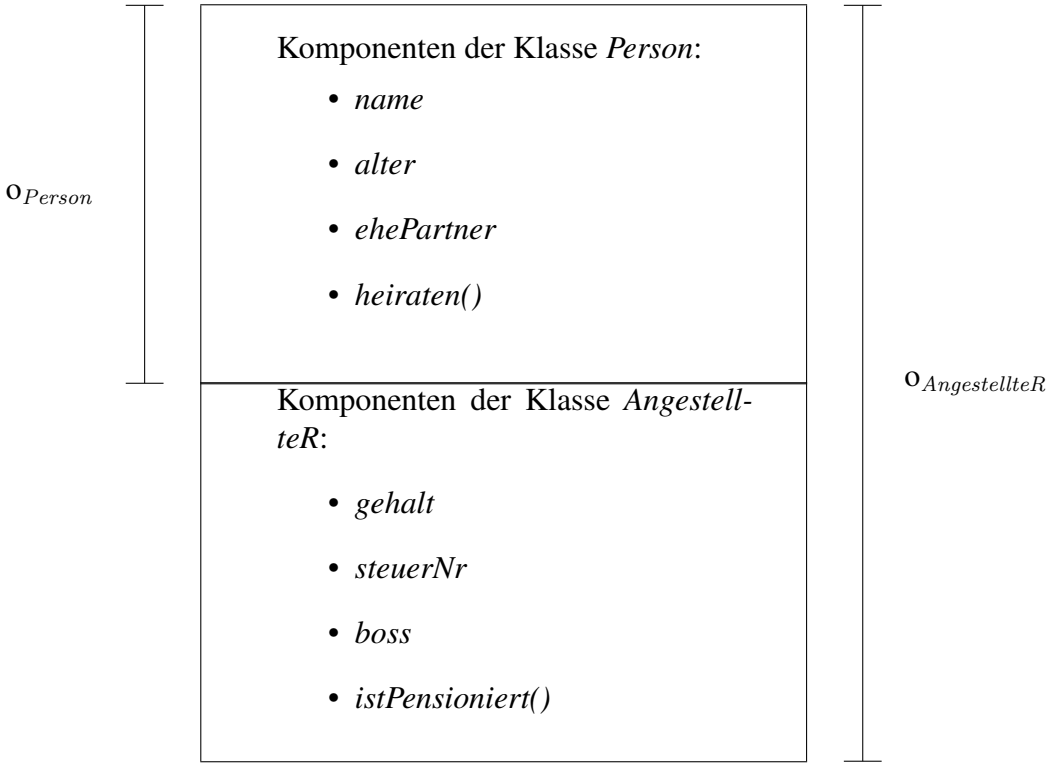



Abbildung 3.3: Schematische Darstellung der Vererbung

abstrakten Typhierarchie in Abbildung 3.4 tun. Es gibt drei Objekttypen, nämlich *Typ1* mit dem Attribut *A*, *Typ2* mit dem zusätzlichen Attribut *B* und *Typ3* mit einem weiteren Attribut *C*.

In Java wird diese Typhierarchie wie folgt definiert:

```
class Typ1 {
    public ... A;
}
```

```
class Typ2
  extends Typ1 {
    public ... B;
}
```

```
class Typ3
  extends Typ2 {
    public ... C;
}
```

Wir bezeichnen *Typ1* als direkten Obertyp (engl. *super type*) von *Typ2*. *Typ1* ist auch ein Obertyp von *Typ3* – allerdings kein direkter sondern ein indirekter Obertyp. In der entgegengesetzten Richtung bezeichnen wir *Typ2* als Untertyp von *Typ1*. Wiederum ist auch *Typ3* ein (indirekter) Untertyp von *Typ1*.

Auf der rechten Seite der Abbildung 3.4 sind auch drei Instanzen dieser drei Objekttypen gezeigt. Wie schon erläutert erben Untertypen alle Komponenten ihrer Obertypen – der direkten sowie der indirekten Obertypen. Also besitzen *Typ2*-Instanzen neben dem in *Typ2* definierten Attribut *B* auch das vom Obertyp *Typ1* geerbte Attribut *A*. Analog besitzen *Typ3*-Instanzen alle drei Attribute: das im *Typ3* definierte Attribut *C*, das vom direkten Obertyp geerbte Attribut *B* und das vom (indirekten) Obertyp *Typ1* geerbte Attribut *A*.

Es ist wichtig zu erkennen, dass Untertypen die durch die *access modifier* einmal festgelegte Sichtbarkeit (public, protected, private, package) nicht einschränken dürfen. Zum Beispiel kann unsere *Typ3*-Instanz auf Lese- und Schreibzugriffe auf die Attribute *A, B* und *C* aller möglichen Klienten reagieren, da alle Attribute als *public* definiert wurden. Es würde das Prinzip der Substituierbarkeit verletzen, wenn ein Untertyp den Zugriff auf


Abbildung 3.4: Schematische Veranschaulichung der Typhierarchie

eine der geerbten Komponenten den Klienten versperren würde, wenn diese Komponente im Obertyp für dieselben Klienten noch sichtbar war. Warum?

Die Menge aller Instanzen eines Objekttyps wird oft auch als *Extension* des Typs bezeichnet. Wir wollen die Extension von *Typ1* als $ext(Typ1)$ notieren. Die Subtypisierung verlangt, dass alle Instanzen eines Untertyps auch als Instanzen des Obertyps aufgefasst werden können. Dies wird durch die Mengeninklusion der Untertyp-Extension in die Obertyp-Extension erzielt. Für unsere Beispiel-Typhierarchie ist dies in Abbildung 3.5 dargestellt. Die Kreise repräsentieren die Instanzen des jeweiligen Objekttyps. Die Extension des *Typ3* ist eine echte Teilmenge der Extension des *Typ2*. In Java wird die Extension eines Objekttyps (einer Klasse) nicht explizit gespeichert. Allerdings kann man jedes Java-Objekt befragen, ob es zur Extension eines bestimmten Typs gehört – dies geschieht durch den Operator **instanceof**. Zum Beispiel ergibt die Befragung `(miniMouse instanceof Angestellter)` den Wahrheitswert `true`. Das gilt auch für den Ausdruck `(miniMouse instanceof Person)` – wegen der eben diskutierten Mengeninklusion der Untertyp-Extension in der Obertyp-Extension.

In Java darf in der *extends*-Klausel der Klassen-Definition nur eine Oberklasse angegeben werden, mit Java nur die einfache Vererbung möglich ist.

## 3.5 Object: Die gemeinsame Wurzelklasse

Es gibt Konzepte, die allen Objekten (allen Instanzen aller Klassen) gemein sind. Einige dieser Eigenschaften wurden schon in vorangegangenen Abschnitten diskutiert:

- *Objektidentität*
  Jedes Objekt besitzt einen eindeutigen Objektidentifikator, der sich während der Lebenszeit des Objekts nicht ändert.

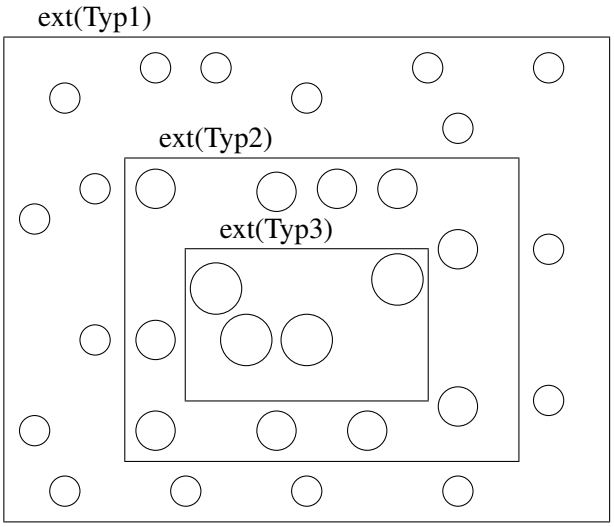



Abbildung 3.5: Visualisierung der Subtypisierung/Mengeninklusion

- *Test auf Identität*
  Man kann Ausdrücke, Variablen oder Attribute auf Identität testen. Dieser Test ergibt *true*, falls die beiden Ausdrücke dasselbe Objekt referenzieren. Das heißt, dieser Test ergibt *true* wenn die Identität der referenzierten Objekte gleich ist. Ein solcher Test ist beispielsweise

  ```
  mickeyMouse == micheyMouse.ehePartner.ehePartner
  ```

  und dieser sollte in einem vernünftigen Zustand der Objektbank *true* ergeben.

Eine weitere für alle Objekte vordefinierte Operation ist die *toString()*-Methode. Diese liefert einen String zurück, der das betreffende Objekt beschreibt. Standardmäßig wird die Identität und der Typ des Empfängerobjekts zurückgegeben – oft wird diese Operation aber in den Klassendefinitionen verfeinert – (siehe Abschnitt 3.6) um eine Klassenspezifische Beschreibung zurück zu liefern.

Es gibt eine Vielzahl weiterer, für die Benutzer oft nicht sichtbarer, Operationen, die allen Objekten zugeordnet sind. Eine Möglichkeit, all diese Operationen zur Verfügung zu stellen, besteht darin, sie jedem Benutzer-definierten Objekttyp implizit hinzuzufügen. In einem Objektmodell mit Vererbung gibt es jedoch eine elegantere Methode: Ein gemeinsamer Wurzeltyp aller Typhierarchien kann diese Funktionalität realisieren und sie somit an alle anderen Objekttypen vererben.

In Java heisst dieser Wurzeltyp *Object*. Dieser Objekttyp wird implizit als Obertyp einer Klasse angenommen, falls gar keine *extends*-Klausel angegeben wird. Somit sind die beiden nachfolgenden Klassendefinitionen äquivalent:

```
class OT {                      class OT extends Object {
   public ...;                     public ...;
}                               }
```

Die Funktionalität von *Object* wird somit entweder explizit oder implizit an den Objekttyp *OT* vererbt. Durch die direkte oder indirekte Vererbung wird die Funktionalität von *Object* an *alle* Java-Klassen vererbt. Die abstrakte Typhierarchie in Abbildung 3.6 verdeutlicht dies.

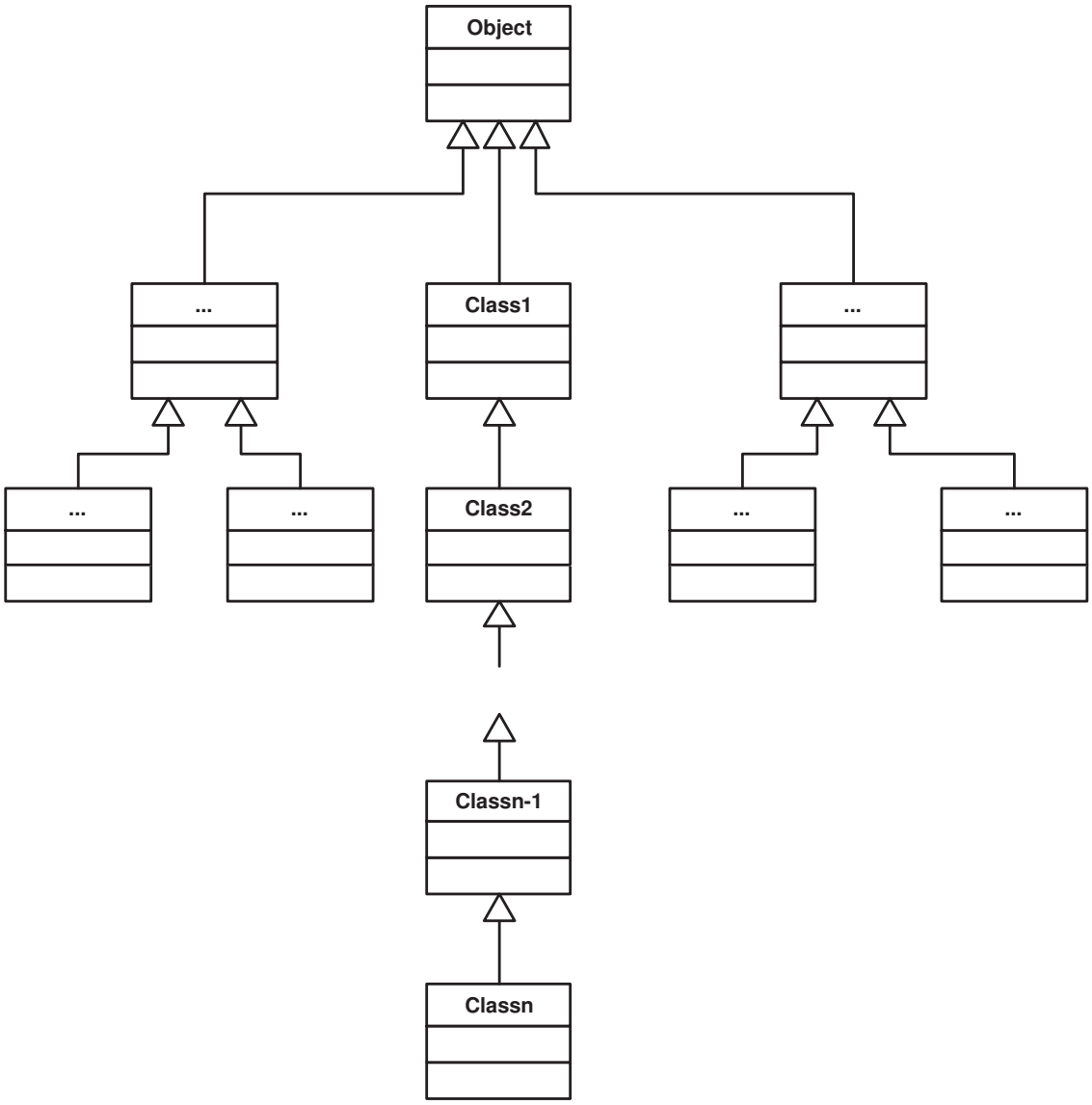



Abbildung 3.6: Die abstrakte Klassenhierarchie

## 3.6 Ein umfassendes Beispiel für Vererbung

Wir wollen die Vorteile der Vererbung anhand eines Beispiels aus der Computergeometrie illustrieren. Wir betrachten dazu die Typhierarchie aus Abbildung 3.7.

Der Obertyp *Zylinder* hat die folgende strukturelle Repräsentation:

- *center1* und *center2*, beide auf den Typ *Vertex* eingeschränkt
- *radius* vom Typ *float*

Instanzen vom Typ *Rohr*, also des direkten Untertyps von *Zylinder*, haben die zusätzlichen Attribute *innererRadius* vom Typ *float*. Der *innererRadius* gibt den halben Durchmesser des Hohlraums des Rohrs an, wohingegen der geerbte *radius* die Hälfte des Außendurchmessers darstellt.

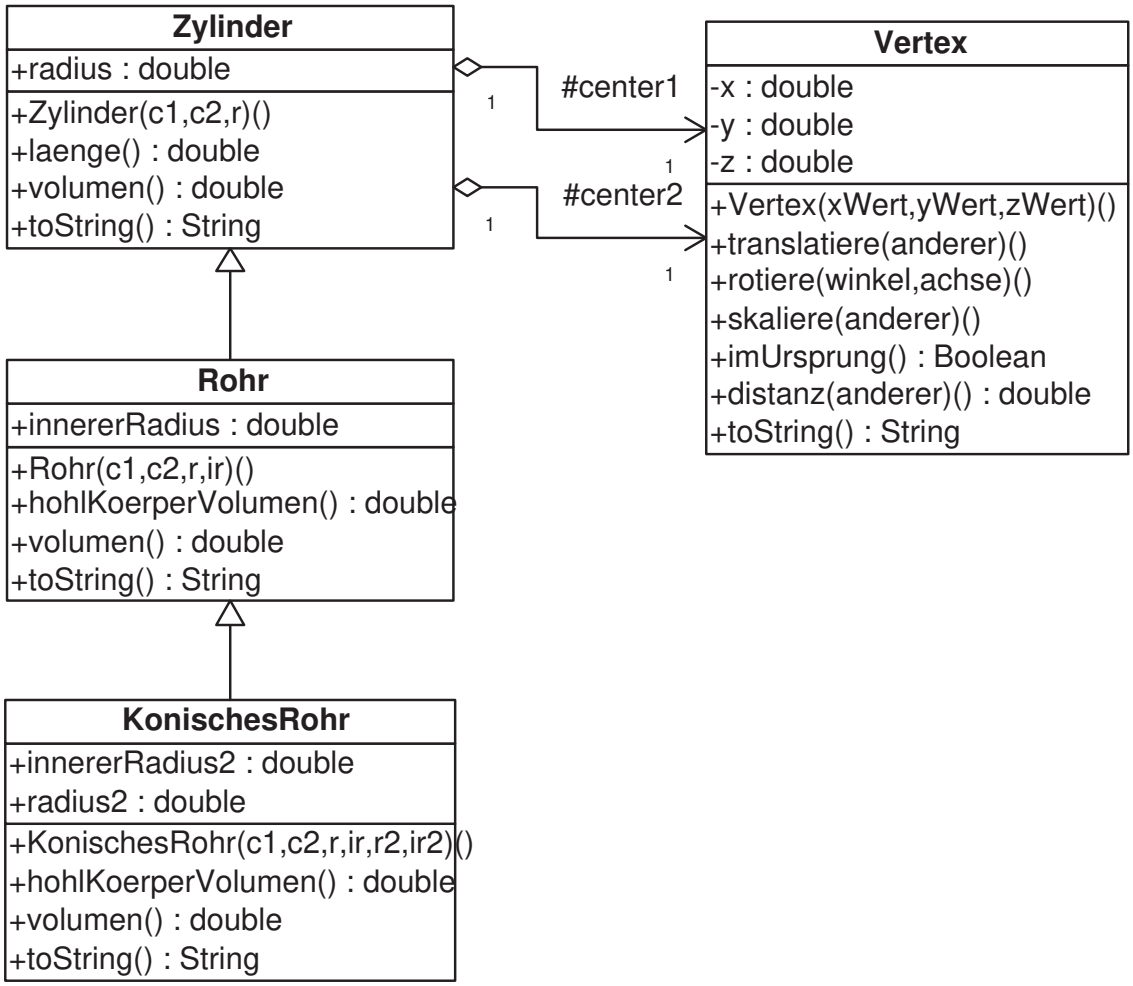



Abbildung 3.7: Die Generalisierung/Spezialisierung der Klassen Zylinder/Rohr/KonischesRohr

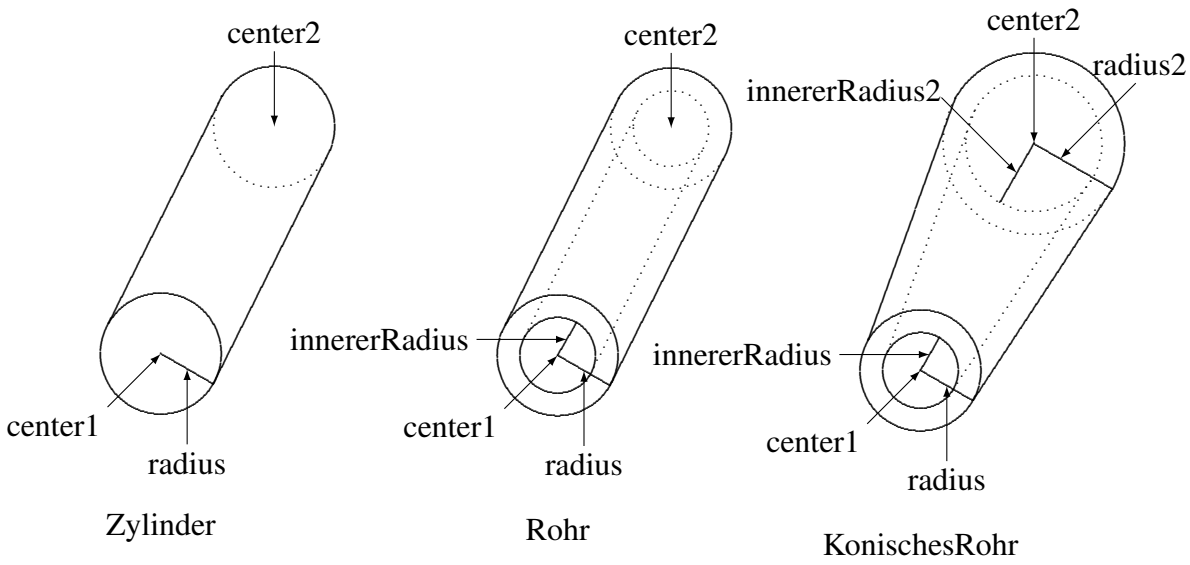



Abbildung 3.8: Strukturelle Repräsentation der *Zylinder*, *Rohr* and *KonischesRohr*-Objekte

Der dritte Objekttyp heißt *KonischesRohr* und modelliert ein sich gleichmäßig über die gesamte Länge verjüngendes Rohr. Für die Modellierung der Objekte vom Typ *KonischesRohr* speichern wir die *radius* und *innererRadius*-Werte auf der einen und die *radius2* und *innererRadius2*-Werte auf der entgegengesetzten Seite.

In Abbildung 3.8 wird die Bedeutung dieser Attribute anhand von Schaubildern klar.

Wir wollen jetzt diese Objekttypen als Java-Klassen definieren. Zunächst nehmen wir an, dass *Zylinder* keinen Obertyp – außer der impliziten Wurzel namens *Object* – besitzt. Später werden wir diese Entscheidung noch revidieren, wenn wir abstrakte Klassen einführen – siehe Kapitel 4. Die Klassendefinition für Zylinder ist in Abbildung 3.9 gezeigt.

Die einzigen Operationen – außer *toString* – sind die Beobachter *laenge* und *volumen*. In der Implementierung von *laenge* haben wir uns wieder auf die schon vorhandene *distanz*-Operation der Klasse *Vertex* abgestützt – genauer gesagt, wir haben die Arbeit an den *center1* delegiert, da die Länge des Zylinders der Distanz zwischen *center1* und *center2* entspricht. In dieser Hinsicht ist der Zylinder ein Klient der Klasse *Vertex*, da sowohl einige Attribute auf diesen Objekttyp eingeschränkt sind und auch einige Operationen von *Vertex* ausgenutzt werden. Das Volumen eines Zylinders ist nach der allgemein bekannten Formel zu berechnen – wir stützen uns wieder auf die mathematischen Routinen und Konstanten (hier $\pi$) der Klasse *Math* ab.

Als nächstes definieren wir die Klasse *Rohr*, die die Klasse *Zylinder* erweitert. *Rohr* ist also ein direkter Untertyp von Zylinder. Der Java-Code findet sich in Abbildung 3.10.

Der Hauptunterschied in der strukturellen Repräsentation von *Rohr*-Objekten besteht in dem zusätzlichen Attribut *innererRadius*. Die anderen drei Attribute werden ja von *Zylinder* geerbt. Weiterhin haben wir eine zusätzliche Operation *hohlKoerperVolumen* hinzugefügt.

Ein weiterer schwerwiegender Unterschied im Verhalten zwischen *Rohr* und *Zylinder* besteht in der unterschiedlichen Berechnungsformel von *volumen*. Das Volumen eines Zylinders ist nach folgender Formel zu berechnen:

```
1  public class Zylinder extends Object {
2      public double radius;
3      protected Vertex center1;
4      protected Vertex center2;
5
6      public Zylinder(Vertex c1, Vertex c2, double r) {
7          this.center1 = c1;
8          this.center2 = c2;
9          this.radius = r;
10     }
11
12     public double laenge() {
13         return this.center1.distanz(this.center2);
14     }
15
16     public double volumen() {
17         return this.radius * this.radius * Math.PI * this.laenge();
18     }
19
20     public String toString() {
21         return "Radius:␣" + this.radius + "␣Center1:" + this.center1 +
22             "␣Center2:␣" + this.center2 + "␣Länge:␣" + this.laenge() +
23             "␣Volumen:␣" + this.volumen();
24     }
25
26     public static void main(String args[]) {
27         Zylinder c = new Zylinder(new Vertex(1,2,3), new Vertex(2,3,4), 5.5);
28         System.out.println(c);
29     }
30 }
```

Abbildung 3.9: Java-Definition der Klasse Zylinder

```
1  public class Rohr extends Zylinder {
2      public double innererRadius;
3
4      public Rohr(Vertex c1, Vertex c2, double r, double ir) {
5          super(c1, c2, r);
6          this.innererRadius = ir;
7      }
8
9      public double hohlKoerperVolumen() {
10         return this.innererRadius * this.innererRadius * Math.PI *
11                this.laenge();
12     }
13
14     public double volumen() {
15         return super.volumen() - this.hohlKoerperVolumen();
16     }
17
18     public String toString() {
19         return super.toString() + "␣Hohlkörper-Volumen:␣" +
20                this.hohlKoerperVolumen();
21     }
22
23     public static void main(String args[]) {
24         Rohr p = new Rohr(new Vertex(1,2,3), new Vertex(2,3,4), 7, 6);
25         System.out.println(p);
26     }
27 }
```

Abbildung 3.10: Die Java-Definition der Klasse Rohr

$$radius ** 2.0 * \pi * laenge()$$

Demgegenüber muss das Volumen eines *Rohr*-Objekts wie folgt berechnet werden:

$$(radius ** 2.0 * \pi * laenge()) - (innererRadius ** 2.0 * \pi * laenge())$$

Deshalb würde die geerbte *volumen*-Operation ein falsches Ergebnis für ein Rohr liefern.

Um dieses Problem zu lösen, gibt es in objektorientierten Modellen das Konzept der *Operationen-Verfeinerung* (engl. *refinement*). Darunter versteht man die Möglichkeit, dass man die geerbte Operation neu implementieren darf, um ihr Verhalten den Erfordernissen des Untertyps anzupassen. In Java wird die Verfeinerung ganz einfach dadurch erreicht, dass eine geerbte Operation im Untertyp noch einmal definiert wird. Dies ist in Abbildung 3.10 für die *volumen*-Operation geschehen. Die Signatur einer verfeinerten Operation muss – bis auf zulässige Parameter-Namensänderungen – mit der Signatur der geerbten Operation identisch sein. Man kann also weder Typänderungen an den Parametern noch am Rückgabewert vornehmen. In dieser Hinsicht muss man in Java sehr gewissenhaft vorgehen, damit man die beiden Konzepte der Verfeinerung und des Überladens von Operationen nicht durcheinander bringt.

In der Verfeinerung wird also nur eine Implementierung der Operation geändert – nicht die Signatur. Man kann bei der Re-Implementierung oft auf die Funktionalität der geerbten Operation zurückgreifen. Dies ist auch in unserem Beispiel geschehen, da wir die geerbte Version mittels **super**.*volume()* aufgerufen haben. Von dem (falschen) Ergebnis von **super**.*volume()* wird zur Korrektur der Wert **this**.*hohlKoerperVolumen()* subtrahiert, um das korrekte Volumen eines Rohrs zu bestimmen.

Zu guter Letzt wird der dritte Objekttyp, *KonischesRohr*, in Java realisiert. Die Klassendefinition findet sich in Abbildung 3.11.

In dieser Klasse wurden gleich zwei geerbte Operationen verfeinert:

- *hohlKoerperVolumen* musste verfeinert werden, um der gleichmäßigen Verjüngung des Hohlkörpers von *innererRadius* zu *innererRadius2* gerecht zu werden.
- *volumen* musste angepasst werden, um der gleichmäßigen Verjüngung von *radius* zu *radius2* Rechnung zu tragen.

## 3.7 Dynamisches Binden verfeinerter Operationen

Es ist nicht ausreichend, die geerbten Operationen zu verfeinern, um sie den speziellen Konstellationen des Untertyps anzupassen. Es muss auch gewährleistet werden, dass die speziellste Variante einer verfeinerten Operation ausgeführt wird. Dies ist die Aufgabe des Laufzeitsystems, da der Compiler nicht mehr immer in der Lage ist, anhand des Programmcodes statisch zu entscheiden, welche Variante einer einmal oder mehrfach verfeinerten Operation ausgeführt werden muss.

Bezogen auf unser Beispiel muss bei einer Invokation der *volumen*-Operation auf einem Objekt vom Typ *KonischesRohr* die Version ausgeführt werden, die in *KonischesRohr* implementiert wurde. Wenn der Empfänger des Aufrufs aber „nur“ ein Rohr ist, muss die in der Klasse *Rohr* verfeinerte Variante ausgeführt – wir sagen gebunden – werden. Diese Anforderung erscheint trivial und sehr einfach zu realisieren, indem man

```
1  public class KonischesRohr extends Rohr {
2      public double innererRadius2;
3      public double radius2;
4
5      public KonischesRohr(Vertex c1, Vertex c2, double r,
6                           double ir, double r2, double ir2) {
7          super(c1, c2, r, ir);
8          radius2 = r2;
9          innererRadius2 = ir2;
10     }
11
12     public double hohlKoerperVolumen() {
13         return ((Math.PI * this.laenge() / 3) *
14                 (Math.pow(this.innererRadius, 2) +
15                  this.innererRadius * this.innererRadius2 +
16                  Math.pow(this.innererRadius2, 2)));
17     }
18
19     public double volumen() {  // Math.pow(x,n) berechnet x hoch n
20         return (((Math.PI * this.laenge() / 3) *
21                   (Math.pow(this.radius, 2) + this.radius * this.radius2 +
22                    Math.pow(this.radius2, 2))) - this.hohlKoerperVolumen());
23     }
24
25     public String toString() {
26         return super.toString() + "␣Hohlkörper-Volumen:␣" +
27                this.hohlKoerperVolumen();
28     }
29
30     public static void main(String args[]) {
31         KonischesRohr p = new KonischesRohr(new Vertex(1,2,3),
32                                             new Vertex(2,3,4), 7, 6, 5, 4);
33         System.out.println(p);
34     }
35 }
```

Abbildung 3.11: Die Java-Definition der Klasse KonischesRohr

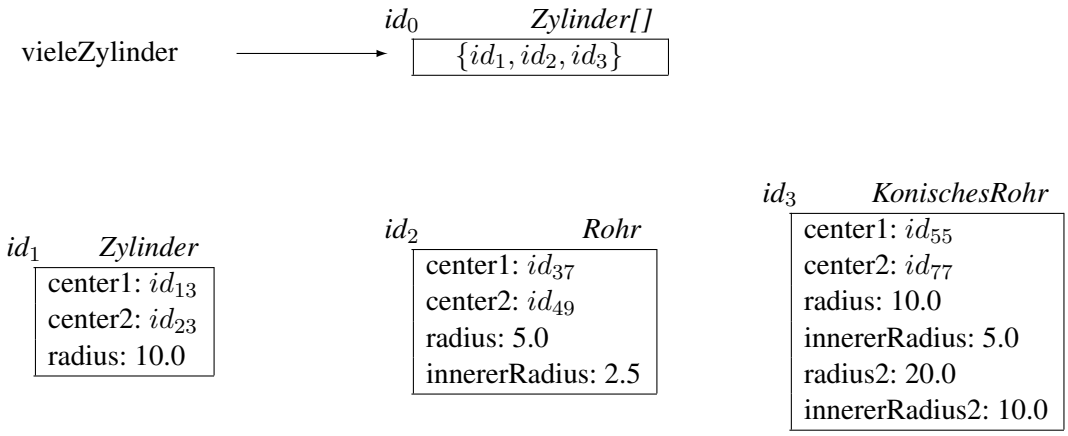



Abbildung 3.12: Objektbank mit einem *Zylinder[]*-Objekt

einfach anhand der Typeinschränkung den Empfängertypen bestimmt und demgemäß die Operation bindet, die für diesen Empfängertyp am besten geeignet, also am speziellsten ist. So einfach geht es aber wegen der Substituierbarkeit nicht. Man kann oft statisch nicht mehr den „wahren“ Typ des Empfängers bestimmen.

Ein Beispiel soll dies belegen. In Abbildung 3.12 ist eine Objektbank mit einem Zylinder-Array – also einem *Zylinder[]*-Objekt mit der OID $id_0$ – gezeigt.

Dieses Array enthält (Referenzen auf) drei Objekte:

- Das Objekt mit der OID $id_1$ ist vom Typ *Zylinder*
- Das Objekt mit der OID $id_2$ ist vom Typ *Rohr*
- Das Objekt mit der OID $id_3$ ist vom Typ *KonischesRohr*

Obwohl die Elemente einer *Zylinder[]*-Instanz alle vom Typ *Zylinder* sein müssen, ist der gezeigte Zustand der Objektbank natürlich typkonsistent. Wegen der Substituierbarkeit darf ein Zylinder-Array durchaus neben den eigentlichen *Zylinder*-Objekten auch Objekte vom Typ *Rohr* oder *KonischesRohr* enthalten. Die Substituierbarkeit besagt ja gerade, dass Untertyp-Objekte – also *Rohr*- und *KonischesRohr*-Instanzen – überall dort substituiert werden dürfen, wo Obertyp-Instanzen – also *Zylinder* – erwartet werden. Insbesondere gilt dies dann auch bei den Elementen eines Zylinder-Arrays.

Wir wollen uns jetzt das folgende Programmfragment anschauen, das das Gesamtvolumen der Objekte in dem Zylinder-Array *vieleZylinder* berechnen soll:

```
1  class DynBindingTest {
2      public static void main(String args[]) {
3          Zylinder cyl = new Zylinder(new Vertex(1,2,3),
4                                      new Vertex(2,3,4), 10.0);
5          Rohr p = new Rohr(new Vertex(1,2,3),
6                            new Vertex(2,3,4), 5.0, 2.5);
7          KonischesRohr cp = new KonischesRohr(new Vertex(1,2,3),
8                                               new Vertex(2,3,4),
9                                               10.0, 5.0, 20.0, 10.0);
10         Zylinder[] vieleZylinder = {cyl,p,cp};
11         double gesamtVolumen = 0.0;
12         Zylinder c;
13         for (int j = 0; j < vieleZylinder.length; j++) {
```

```
14              c = vieleZylinder[j];
15              gesamtVolumen = gesamtVolumen + c.volumen();
16          }
17          System.out.println("Gesamtvolumen:␣" + gesamtVolumen);
18      }
19  }
```

In der **for**-Schleife verweist die Variable *c* nacheinander auf die drei Elemente des *Zylinder[]*-Objekts, auf das *vieleZylinder* verweist. Nehmen wir an, dass die Elemente in der in der Abbildung 3.12 gezeigten Reihenfolge besucht werden:

1. In der ersten Iteration verweist *c* auf das Objekt mit der OID $id_1$, das eine *direkte Zylinder*-Instanz ist
2. In der zweiten Iteration verweist *c* auf die direkte *Rohr*-Instanz mit der OID $id_2$
3. In der abschließenden dritten Iteration verweist *c* auf $id_3$, also eine *KonischesRohr*-Instanz.

In allen drei Iterationen der **for**-Schleife muss eine anderen Variante der Operation *volumen* gebunden werden, da jeweils spezielle Formeln für die Volumen-Berechnung gelten:

1. $id_1.volume = \pi * r ** 2 * l$
2. $id_2.volume = \pi * r ** 2 * l - \pi * i ** 2 * l$
3. $id_3.volume = (\pi * l/3 * (r ** 2 + r * r_2 + r_2 ** 2)) - (\pi * l/3 * (i ** 2 + i * i_2 + i_2 ** 2))$

In diesen Formeln haben wir folgende Abkürzungen verwendet: $r$ für **this**.*radius*, $l$ für **this**.*laenge()*, $i$ für **this**.*innererRadius*, $r_2$ für **this**.*radius2* und $i_2$ für **this**.*innererRadius2*. Bei genauer Betrachtung kann man leicht sehen, dass diese Formeln den drei unterschiedlichen Implementierungen der *volumen*-Operation in den Klassen *Zylinder*, *Rohr* und *KonischesRohr* entsprechen.

Das dynamische oder späte Binden der Operation erzielt in unserem Beispiel folgenden Effekt:

1. Wenn $c$ auf Objekt $id_1$ verweist, also auf eine *Zylinder*-Instanz, wird die *volumen*-Implementierung aus der Klasse *Zylinder* ausgeführt.
2. Wenn $c$ auf eine *Rohr*-Instanz, wie beispielsweise $id_2$, verweist, wird die Implementierung von *volumen* aus der Klasse *Rohr* gebunden.
3. Wenn $c$ schließlich auf eine Instanz vom Typ *KonischesRohr* verweist, muss die Implementierung aus der Klasse *KonischesRohr* gebunden werden.

Zufälligerweise besitzen alle Klassen unserer Typhierarchie ihre eigene Implementierung von *volumen*. Dies ist jedoch im allgemeinen nicht der Fall. Auch verfeinerte Operationen werden natürlich weitervererbt. Es wird natürlich die verfeinerte Variante weiter nach unten in der Typhierarchie vererbt.

Wie funktioniert das dynamische Binden? Das Java-Laufzeitsystem, startet beim direkten Typ des Empfängerobjekts, also in unserem Beispiel beim direkten Typ des Objekts, auf das *c* verweist. Beginnend von diesem Typ aus wird Typ-Hierarchie in Richtung

der Wurzel-Klasse *Object* durchlaufen bis man eine Klasse mit einer Implementierung der betreffenden Operation findet. Diese Implementierung ist bezogen auf den direkten Typ des Empfängers die speziellste Variante der Operation, die man ausführen kann.

Es wird jetzt klar, warum man dieses Konzept *dynamisches* oder *spätes* Binden nennt. Das Binden der Operation muss zur Laufzeit erfolgen, da man zur Übersetzungszeit des Programms nicht immer wissen kann, welches der direkte Typ des Empfängerobjekts ist. In unserem Beispiel hängt dies u.a. auch von der Position der Objekte im Array ab.

Intuitiv kann man sich das dynamische Binden so vorstellen, dass der Empfänger des Operationsaufrufs befragt wird, von welcher Klasse er instanziiert wurde – diese Klasse haben wir vorher als direkten Typ des Empfängers bezeichnet. Dann beginnt man, wie gesagt, bei dieser Klasse und sucht nach einer Implementierung der aufgerufenen Operation. Wenn sie nicht schon in dieser Klasse vorhanden ist, geht man zur direkten Oberklasse und sucht dort. Wenn sie wiederum auch dort nicht vorhanden ist, geht man zur nächsthöheren Oberklasse. Wegen der Typsicherheit von Java wird man auf dem Weg zur gemeinsamen Wurzel *Object* (oder spätestens in der gemeinsamen Wurzel *object*) der Typhierarchie eine Implementierung der invokierten Operation finden.

Weiterhin ist zu bemerken, dass der Weg in Richtung zur Wurzel der Typhierarchie eindeutig ist. Dies ist eine Konsequenz aus der einfachen/singulären Vererbung, auf die sich Java beschränkt. Siehe hierzu auch die Typhierarchie in Abbildung 3.6 und betrachte den Weg vom Objekttyp *Class* in Richtung zur Wurzel *Object*.

## 3.8 Typsicherheit bei der Subtypisierung

Beim Design von Java wurden zwei Ziele verfolgt, die in gewisser Weise gegensätzlich sind:

- *Flexibilität des Modells*
  Die Nutzung der Vererbung in Kombination mit der Subtypisierung bzw. der Substituierbarkeit gewährleisten ein hohes Maß an Flexibilität des Objektmodells. Wir haben dies an unseren Beispiel-Klassen *Person* und *Angestellter* und der Beispieloperation *heiraten* illustriert. Weiterhin wurde an der Beispiel-Typhierarchie *Zylinder*, *Rohr*, und *KonischesRohr* die Ausdrucksmächtigkeit der Operationenverfeinerung zur Spezialisierung des Verhaltens der Unterobjekte und der einhergehenden dynamischen Bindung dieser Operationen demonstriert.

- *Typsicherheit*
  Die Typsicherheit verlangt, dass potenzielle Typfehler schon bei der Übersetzung entdeckt werden. Allerdings sollte man hier schon zur Vorsicht mahnen, da Java nicht alle „Schlupflöcher“ schließt: Wenn Programmierer darauf aus sind, Typfehler zur Laufzeit herbeizuführen, so ist dies in Java immer noch möglich.[1]

Diese beiden Ziele sind natürlich nicht isoliert voneinander zu sehen, sondern in gewisser Weise sogar gegensätzlich: Durch die strenge Typisierung wird die Flexibilität eingeschränkt und durch die Flexibilität wird die Typsicherheit eingeschränkt.[2]

[1] Gute Programmierer zeichnen sich dadurch aus, dass sie zwar wissen, wie es ginge, es aber nicht tun.

[2] Für die letztere Aussage gibt es in Java ein gutes Beispiel: Java behandelt beispielsweise ein *Rohr[]*-Objekt als substituierbar für ein *Zylinder[]*-Objekt, obwohl man dadurch Laufzeit-Typfehler herbeiführen kann. Warum?

### 3.8.1 Typisierungsregeln im Zusammenhang mit der Subtypisierung

Die Substituierbarkeit von Untertyp-Instanzen anstelle von Obertyp-Instanzen erhöht die Flexibilität des Objektmodells ungemein. Um aber die Typsicherheit aufrecht zu erhalten, muss man sehr rigide Regeln einhalten, damit man die Typkonsistenz schon während der Übersetzungszeit garantieren kann. Nur so kann man während der Ausführung auftretende Laufzeitfehler aufgrund inkonsistent typisierter Ausdrücke ausschließen. Die Grundprinzipien der Typüberprüfung sind die folgenden:

1. Stelle sicher, dass die statischen Typeinschränkungen eingehalten werden. Wenn die statischen Typeinschränkungen ein Objekt vom Typ $T$ verlangen, muss sichergestellt werden, dass bei jeder möglichen Programmausführung auch wirklich nur Objekte vom Typ $T$ oder einem Untertyp $T'$ von $T$ dort „auftauchen“ können.

2. Stelle sicher, dass alle notwendigen Komponenten (Operationen und Attribute) eines Typs auch wirklich vorhanden sind.

Das heißt, dass der *Type-Checker*, der ein Teil des Compilers ist, statisch die Typsicherheit auf der Basis der im Programm deklarierten Typeinschränkungen der Datenkomponenten (also den Attributen, lokalen Variablen, Parametern, Array-Elementen, etc.) verifizieren muss.

Wir wollen die Konsequenzen anhand eines Programmfragments demonstrieren:

```
   Person einePerson;
   Angestellter einAngestellter;
   ...
(1)einePerson = einAngestellter;
(2)...
(3)einAngestellter = einePerson;        // nicht erlaubt!
```

Die Zuweisung (1) ist sicherlich typ-konsistent. Die Variable *einePerson* verlangt – gemäß der Typeinschränkung der Variablendeklaration – ein Objekt vom Typ *Person* oder einem Untertyp davon. Deshalb kann man getrost ein *Angestellter*-Objekt der Variablen *einePerson* zuweisen. Die Zuweisung (3) ist jedoch nicht typsicher, da die Variable *einePerson* zwar wegen der Zuweisung in (1) zwar auf ein Objekt vom Typ *Angestellter* verweisen kann – dies aber nicht muss. Der Compiler kann im allgemeinen nicht den Programmfluss nachverfolgen, um zu verifizieren, dass die in (1) erfolgte Zuweisung an *einePerson* immer noch aktuell ist. Wer weiß schon was zum Beispiel in (2) passiert ist.

Generell wird eine Zuweisung als typkonsistent verifiziert, wenn der für die linke Seite der Zuweisung bestimmte Typ mit dem Typ auf der rechten Seite übereinstimmt – oder ein Obertyp davon ist.

Für die Zuweisung in Ausdruck (3) bestimmt der Compiler folgende Typen:

$$\underbrace{\text{einAngestellter}}_{Angestellter} = \underbrace{\text{einePerson}}_{Person};$$

Da der Typ auf der linken Seite des „=“-Zeichens kein Obertyp des für die rechte Seite bestimmten Typs ist, wird der Compiler die Zuweisung als nicht typsicher zurückweisen.

Der zweite Teil der Typsicherheits-Überprüfung verlangt, dass die Verfügbarkeit der zugegriffenen Komponenten (Attribute und Operationen) eines Objekttyps auch gewährleistet ist. Betrachten wir folgendes Programmfragment:

```
(1)einePerson.name;         // okay
(2)einePerson.gehalt;       // nicht erlaubt!
(3)einAngestellter.gehalt; // okay
```

Für eine *Person* gibt es kein Attribut *gehalt*; deshalb wird Ausdruck (2) vom Compiler abgewiesen. Natürlich kann die Variable *einePerson* immer noch auf ein *Angestellter*-Objekt verweisen – nur kann sich der Compiler darauf nicht verlassen. Im schlimmsten Fall verweist die Variable gemäß der Typeinschränkung auf eine *Person*-Instanz und die besitzt gemäß der Klassendefinition kein Attribut *gehalt*. Andererseits ist der Ausdruck (3) typsicher, da *einAngestellter* im schlimmsten Fall auf eine *Angestellter*-Instanz verweisen kann – und niemals auf eine Person. Deshalb ist die Existenz des Attributs *gehalt* hier garantiert.

### 3.8.2 Beispiele für die Typisierungsregeln

Wir wollen die Typisierungsregeln und die Vorgehensweise bei der Überprüfung der Typkonsistenz anhand einer Beispiel-Objektbank illustrieren. Man beachte, dass hier Mickey ein *Angestellter* ist und Mini eine *Person* – im Unterschied zu den bisherigen Beispielen.

```
   Person miniMouse;
   Angestellter mickeyMouse;
   Angestellter chef;
   int i;
   ...
(1)mickeyMouse.ehePartner = miniMouse;            // okay
(2)miniMouse.ehePartner = mickeyMouse;            // okay
(3)mickeyMouse.boss = chef;                       // okay
(4)miniMouse.ehePartner.boss = chef;              // nicht erlaubt!
(5)i = mickeyMouse.boss.steuerNr;                 // okay
(6)i = miniMouse.ehePartner.boss.steuerNr;        // nicht erlaubt!
(7)i = miniMouse.ehePartner.ehePartner.alter;     // okay
(8)i = mickeyMouse.ehePartner.boss.steuerNr;      // nicht erlaubt!
(9)mickeyMouse.boss.ehePartner.heiraten(chef);    // okay
```

In Abbildung 3.13 ist ein möglicher Zustand dieser Objektbank gezeigt, die in dem ersten Teil des obigen Programmfragments so angelegt worden sein könnte. Wir wollen jetzt Ausdruck (4) genauer untersuchen:

$$\underbrace{\underbrace{\underbrace{\text{miniMouse}}_{Person}.\text{ehePartner}}_{Person}.\text{boss}}_{potential\ ERROR} = \underbrace{\text{chef}}_{Angestellter} \;;$$

Die Unterklammerung dieser Zuweisung zeigen den jeweiligen Typ der Ausdrücke an, wie der Compiler sie aus den Typeinschränkungen ermitteln kann. Man beachte, dass

der Compiler (bzw. der *Type-Checker* des Compilers) sich nicht auf den sich dynamisch ändernden Zustand der Objektbank aus Abbildung 3.13 verlassen kann – alle Typüberprüfungen müssen auf der Basis der statischen Typeinschränkungen durchgeführt werden. Deshalb bestimmt der Compiler den Typ der Variablen *miniMouse.ehePartner* als *Person*. Dass der Ehepartner von *miniMouse* in unserer Objektbank ein *Angestellter* ist, ist für den Compiler nicht ersichtlich und möglicherweise auch nur vorübergehend der Fall. Legal ist der Verweis auf einen Angestellten natürlich allemal wegen der Substituierbarkeit.

Auf jeden Fall muss der Ausdruck *miniMouse.ehePartner.boss* als potenziell typinkonsistent abgelehnt werden, da Personen das Attribut *boss* nicht besitzen. Mit Blick auf unsere Objektbank erscheint dies zwar etwas sonderbar, da der semantisch äquivalente Ausdruck (3) akzeptiert wird. Aber, nochmals sollte betont werden, dass (3) nur in unserer derzeitigen Objektbank-Konstellation äquivalent ist – dies kann sich jederzeit ändern.

Wir wollen uns jetzt noch mit dem Ausdruck (7) etwas genauer befassen. Seine Typkonsistenz lässt sich wie folgt verifizieren:

$$\underbrace{\mathrm{i}}_{int} = \underbrace{\underbrace{\underbrace{\underbrace{\mathrm{miniMouse}}_{Person}.\mathrm{ehePartner}}_{Person}.\mathrm{ehePartner}}_{Person}.\mathrm{alter}}_{int};$$

Da der für die linke Seite der Zuweisung hergeleitete Typ mit dem für die rechte Seite hergeleiteten Typ übereinstimmt, wird diese Zuweisung als typsicher akzeptiert.

Wie schon mehrfach betont, ist bei Objekttypen eigentlich keine Typgleichheit bei der Zuweisung notwendig: der Typ der linken Seite muss gleich dem Typ der rechten Seite sein oder ein Obertyp des Typs der rechten Seite darstellen. Dies gilt wegen der Substituierbarkeit.

## 3.9 Übungen

**3.1** In Section 3.1, only one-half of the *heiraten* operation was implemented. The ehePartner attribute of **this** was set to the *partner*, but the *partner*'s ehePartner attribute was not set to **this**. Give the full implementation taking care of both attributes. Provide this implementation for both types, *Person* and *Angestellter*. Which of the definitions is legal, which is not? Why?

**3.2** Illustrate the problems that may occur if the **public** specifications were not inherited by the subtype.

Hint: In this case, the outer signature of the subtype is not a superset of the outer signature of the super type.

**3.3** Inheritance and subtyping are commonly associated with one another—as in the object model Java. However, this mash of two (slightly) different concepts may lead to problems. As an example consider modeling of *Cube*s and *Cuboid*s. Assume that a *Cube* is modeled by one attribute *laenge* and a *Cuboid* by three attributes *laenge*, *width*, and *height*. It is common usage to represent the type *Cuboid* as a

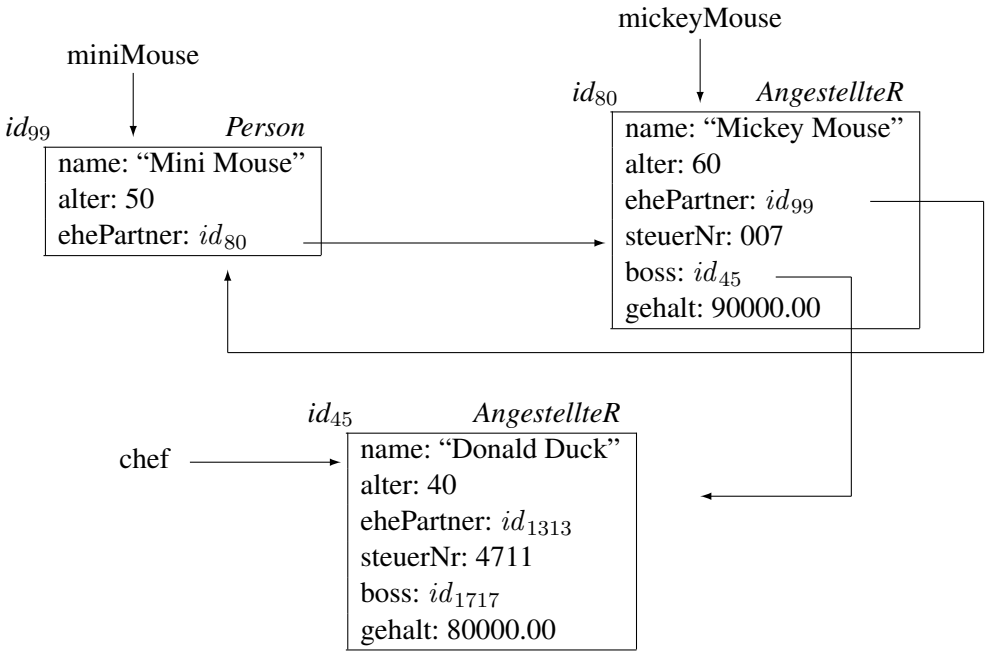



Abbildung 3.13: Beispiel-Objektbank mit *Person* und *Angestellter*-Objekten

subtype of *Cube*, which inherits the one attribute *laenge* and augments the structural representation by two further attributes *width* and *height*.

- Illustrate the problems arising from this type structure. You should illustrate the discussion on a sample program fragment.

  Hint: The problems occur because the "real" **is-a** relationship between *Cube*s and *Cuboids* was inverted in the type definition in order to exploit inheritance.

- Consider the alternative modeling that a *Cube* is a subtype of *Cuboid*. What problems occur now?

  Hint: The type *Cube* now has more attributes than necessary and meaningful.

Reconsider the last point in more detail. Assume that *local* attributes are available and cannot be made public. These are *not* inherited. Further, it is allowed to implement two value-receiving and value-returning operations, which cover attribute accesses. These can be used used to "refine" attributes. How do these assumptions help? Aside: Why can local attributes not be public?

**3.4** Extend the *Person*/*Angestellter* type hierarchy by additional types *Manager*, *CEO* (chief executive officer), *Student*, etc.

Augment the behavioral specifications of the types—in particular make use of operation inheritance and refinement.

**3.5** Consider the following two operations in which the second is an attempted—though invalid—refinement of the first:

```
class Rohr extends Zylinder {
    // ...
    public void connect(Rohr nextOne) {
        // ...
    }
}

class KonischesRohr extends Rohr {
    // ...
    public void connect(KonischesRohr nextOne) {
        // ...
    }
}
```

Why is this refinement invalid? Illustrate the problems that can arise due to the refined signature ("hole" in the applicability of the *connect* operation).

**3.6** In the text, we denied the possibility to refine type constraints of inherited attributes. Illustrate that a specialization (i.e., the new type constraint is a subtype of the original one) as well as a generalization (i.e., the new type constraint being a super type of the original one) cause typing problems.

**3.7** For attributes that are only read by clients—i.e., those for which only the read operation is made **public**—a specialization of the type constraint is legitimate. Verify this claim.

**3.8** The typing rules can all be unified to valid operation refinement rules. For this purpose, an attribute is modeled as the pair of read/write-operations and array-structured objects are modeled via their operations provided for clients. Illustrate this idea.

## 3.10 Annotated Bibliography

The super-/subtype concept has its origin in the forerunner of all object-oriented systems: Simula-67 [Dahl, Myrhaug und Nygaard (1970), Nygaard und Dahl (1981)]. Simula was the first language to support subtype substitutability; though the terminology of object-oriented programming had not been established at the time Simula was conceived.

In parallel to the object-oriented languages, inheritance concepts were developed in knowledge representation languages, e.g., in KRL by Bobrow and Winograd [Bobrow und Winograd (1977), Bobrow und Winograd (1979)].

The inheritance of Simula was restricted to single inheritance. Only later developments, especially in Lisp-based languages, such as Loops [Bobrow et al. (1990)], Flavors [Moon (1986)], CLOS [Keene (1989)], extended the single inheritance mechanism to multiple inheritance.

The typing rules, presented in this chapter, that reconcile strong typing with polymorphism by constraining a object base component to refer to descendants of its specified type is adopted from Simula. Similar rules are incorporated in Eiffel [Meyer (1988)], a new object-oriented programming language that is similar to Simula (except that it provides

multiple inheritance). In contrast, Smalltalk [Goldberg und Robson (1983)] is dynamically typed; a component of the system can refer to any object instance. The actual type of an object is determined at run time only. Two C-based object-oriented languages with inheritance are C++ [Stroustrup (1990)] and Objective C [Cox (1986)].

Alan Snyder [Snyder (1986)] discusses the implications of inheritance on the encapsulation and information hiding of objects. Stein, Lieberman, and Ungar [Stein, Lieberman und Ungar (1989)] review the different aspects of behavior sharing among objects—inheritance is merely one approach to achieve this goal.

Bruce and Wegner try to formalize inheritance concepts in [Bruce und Wegner (1990)]. Another formalization, called *F-Logic*, of object-oriented features—including subtyping—was developed by Kifer and Lausen [Kifer und Lausen (1989)]. In [Kemper und Moerkotte (1990)] Kemper and Moerkotte discuss shortcomings of the inheritance concept, which are mainly due to exceptions that apply to subtypes.

There is some prior work on inheritance in the object base context. The functional model DAPLEX [Shipman (1981)] incorporates inheritance as one of the earliest proposals. More recent object base projects that are based on the functional approach of data representation adopted the DAPLEX inheritance concept. Among these are Iris [Derret et al. (1986)] and Probe [Dayal et al. (1987)].

The database system Gemstone [Butterworth, Otis und Stein (1991)] is founded on the Smalltalk object model and therefore provides only single inheritance; just like the object manager Vbase [Ontologic (1987)] (for which—however—multiple inheritance is announced in future releases).

Type safety in the presence of subtyping is discussed by several authors. Among the first is Cardelli [Cardelli (1988)] and Cardelli and Wegner [Cardelli und Wegner (1985)]. Danforth and Tomlinson [Danforth und Tomlinson (1988)] follow up this work and provide a thorough overview of typing issues. The typing rules that are employed in the Java object model are formalized by the authors of this book in [Kemper und Moerkotte (1989)] and [Kemper und Moerkotte (1992)]. A more thorough treatment can be found in a German monograph by Kemper [Kemper (1992)]. Type safety on the basis of type inference was studied in the context of the Machiavelli project [Ohori (1988), Ohori, Buneman und Breazu-Tannen (1989), Breazu-Tannen, Buneman und Ohori (1989)]. The seminal work on type inference was carried out by Milner [Milner (1978)]. The *anchor* concept to increase the flexibility of otherwise purely static type constraints was borrowed from the Eiffel language [Meyer (1988)].

# 4. Abstrakte Klassen und Type Casting

Objekttyp-Hierarchien ermöglichen die Repräsentation von Generalisierung und Spezialisierung. Dabei werden die gemeinsamen Komponenten der Untertypen herausfaktorisiert und im Obertyp (dem Generalisierungstyp) definiert. Alle Untertypen erben dann diese Komponenten und können weitere Komponenten definieren. Durch das Konzept der Operationen-Verfeinerung in Kombination mit dem dynamischen Binden lässt sich das Verhalten geerbter Operationen noch an die besonderen Erfordernisse des Untertyps (also des spezialisierten Objekttyps) anpassen.

Trotz dieser flexiblen und ausdrucksstarken Modellierungsmöglichkeiten, bleiben zwei Probleme bestehen:

1. Die Operation, die man aus den Untertypen herausfaktorisieren möchte, lässt sich in dem gemeinsamen Obertyp noch gar nicht realisieren, weil nicht genug Information vorhanden ist.

2. Die Operation, die man herausfaktorisieren möchte, hat unterschiedliche Parameter und lässt sich deshalb gar nicht als *eine* Operation des Obertyps spezifizieren.

In diesem Kapitel werden wir zwei neue Konzepte beschreiben, um diese beiden Probleme zu lösen. Die *abstrakten Klassen* werden benutzt, um auch Operationen in einem Obertyp deklarieren zu können, die noch gar nicht implementierbar sind. Wenn man eine Routine schreiben will, deren Verhalten von dem direkten Typ eines (oder mehrerer) Argument-Objekte abhängt, kann man dies durch Typabfragen (instanceof) und das Type-Casting der Objekte schaffen. Auf diese Art und Weise kann man dann auch Operationen dynamisch anpassen.

## 4.1 Abstrakte Klassen

Nicht alle Objekttypen sind so vollständig, dass sie auch tatsächlich instanziiert werden können. Diese Objekttypen liefern nur das abstrakte Rahmenwerk für die gemeinsame Funktionalität der instanziierbaren Untertypen. Deshalb bezeichnet man diese Objekttypen auch als *abstrakte Objekttypen*. Ein Beispiel dafür hatten wir eigentlich schon kennengelernt: der Objekttyp *Object* ist ein solcher nicht-instanziierbarer, abstrakter Objekttyp.

Wir wollen uns hier aber einem weiteren Beispiel zuwenden. Der Typ *GeoPrimitive* soll die Wurzel einer Typhierarchie für geometrische Objekte bilden, die in Abbildung 4.1 gezeigt ist. Diese Klasse enthält die Attribute *geoID*, *farbe* und *mat* (mit der offensichtlichen Bedeutung).

```
public class GeoPrimitive {
    public int geoID;
    public Material mat;
    public String farbe;
```

```
    public double spezGewicht() {
        return mat.spezGewicht;
    }

} // class GeoPrimitive
```

Dieser Objekttyp kann keine Annahmen über die geometrischen Eigenschaften der Instanzen machen, da diese sehr unterschiedlich sein können (Quader, Zylinder, Pyramiden, etc.). Deshalb sind auch keine Operationen, die diese Information benötigen, wie beispielsweise *volumen* oder *gewicht* realisierbar in *GeoPrimitive*. Diese können erst in den jeweiligen Untertypen, wenn die entsprechenden Attribute für die geometrische Repräsentation vorhanden sind, realisiert werden. Dies ist sehr ungünstig, wenn wir beispielsweise das nachfolgende Programmfragment betrachten:

```
GeoPrimitive[] basisTeile;
```

In diesem Array *basisTeile* wollen wir Bauteile unterschiedlicher geometrischer Form speichern, um daraus vielleicht ein komplexeres Aggregat zusammenbauen zu können. Auf dieser Basis können wir aber leider das nachfolgende Programm, das das *gesamtGewicht* und *gesamtVolumen* aller Bauteile ermitteln soll, nicht realisieren, da Java dies wegen der strengen Typisierungsregeln als potenziell typ-unsicher ablehnt:

```
GeoPrimitive b;
double gesamtGewicht = 0.0;
double gesamtVolumen = 0.0;
for (int i = 0; i < basisTeile.length; i++) {
   b = basisTeile[i];
   gesamtGewicht = gesamtGewicht + b.gewicht();
   gesamtVolumen = gesamtVolumen + b.volumen();
}
```

Dieses Programm wird abgelehnt, da der Compiler für die Variable *b* den Typ *GeoPrimitive* ermittelt. Dieser Typ hat aber keine Operation namens *gewicht* oder *volumen*. Dies ist umso unerfreulicher, als das Array vermutlich nur Untertyp-Instanzen enthält, die alle diese Operationen „beherrschen“. Ist dies der Preis, den wir für die Typsicherheit (der Programmiersprache Java) bezahlen müssen? Glücklicherweise ist die Antwort *nein*.

Es gibt mehrere nahe liegende, aber nicht-realisierbare Herangehensweisen, das Problem zu beheben:

1. Andere Typisierung der Variablen *basisTeile*:
   Man könnte *Zylinder[]* als Typ verwenden. Dann sind die Operationen *volumen* und *gewicht* garantiert anwendbar. Aber dann kann man in dem Array *basisTeile* keine Polyeder mehr speichern, sondern nur noch Zylinder und deren Untertyp-Instanzen. Dies ist aber nicht tolerierbar, da wir ja alle Arten von geometrischen Objekten dort einfügen wollen.

2. Definition der Operation *volumen* schon in *GeoPrimitive*:
   Das scheitert leider daran, dass wir keine Möglichkeit haben, die Operation dort zu realisieren.

Die Lösung des Problems besteht darin, dem Objekttypen *GeoPrimitive* die Operationen *volumen*, *gewicht* und *toString* teilweise ohne Realisierung wie folgt hinzuzufügen:

```
1  public abstract class GeoPrimitive {
2      public int geoID;
3      public Material mat;
4      public String farbe;
5
6      public double spezGewicht() {
7          return mat.spezGewicht;
8      }
9
10     public abstract String toString();
11
12     public abstract double volumen();
13
14     public double gewicht() {
15         return this.volumen() * this.spezGewicht();
16     }
17
18 } // end abstract class GeoPrimitive
```

Diese Definition macht aus *GeoPrimitive* einen *abstrakten Objekttyp*, von dem keine Instanzen instanziiert werden können. Die Klasse *GeoPrimitive* enthält die beiden abstrakten Operationen *volumen* und *toString* sowie die beiden „normalen“ Operationen *spezGewicht* und *gewicht*. Die beiden abstrakten Operationen haben keine Implementierung – sie dürfen in Java auch gar keine haben. Sie müssen in den instanziierbaren Subtypen der Klasse *GeoPrimitive* als Verfeinerung implementiert werden. Die Verfeinerung einer geerbten abstrakten Methode erfolgt so wie die Verfeinerung einer normalen Methode (siehe Abschnitt 3.6).

Wie bereits gesagt kann man eine abstrakte Klasse nicht instanziieren. Deshalb gibt es keine Objekte, deren direkter Typ *GeoPrimitive* ist. Die dürfte es ja auch gar nicht geben, wenn man nachfolgendes Programmfragment betrachtet:

```
einGeoObjekt.volumen()
```

Wenn *einGeoObjekt* auf eine Instanz der Klasse *GeoPrimitive* verweisen würde, dann hätten wir ein Problem: Die aufgerufene Operation *volumen* wäre nicht ausführbar. Wenn aber die Variable *einGeoObjekt* auf eine Instanz eines instanziierbaren Untertyps von *GeoPrimitive* verweist, wird mittels dynamischen Bindens eine Verfeinerung der abstrakten Operation *volumen* ausgeführt.

Jede Unterklasse von *GeoPrimitive* muss entweder alle geerbten abstrakten Operationen implementieren oder sie ist selbst als nicht-instanziierbare, abstrakte Klasse zu definieren. Wir sollten noch darauf hinweisen, dass in der Implementierung der Operation *gewicht* eine abstrakte Operation, hier *volumen*, aufgerufen wird. Das ist völlig legitim, auch wenn *volumen* in diesem Objekttyp noch gar keine Implementierung hat. Durch die dynamische Bindung wird erreicht, dass zur Ausführungszeit von *gewicht* die Verfeinerung der abstrakten Operation *volumen* ausgeführt wird.

Wir wollen als erstes den Subtyp *Zylinder* betrachten. In diesem Subtyp werden die beiden geerbten abstrakten Operationen *volumen* und *toString* implementiert. Man beachte, dass wir *gewicht* nicht realisieren müssen, da die geerbte Operation aus *GeoPrimitive* völlig ausreicht.

```
class Zylinder extends GeoPrimitive {
    public double radius;
    protected Vertex center1;
```

```
    protected Vertex center2;

    public Zylinder(Vertex c1, Vertex c2, double r) {
        center1 = c1;
        center2 = c2;
        radius = r;
    }

    public double laenge() {
        return center1.distanz(center2);
    }

    public double volumen() {
        return this.radius * this.radius * Math.PI * this.laenge();
    }

    public String toString() {
        return "Zylinder mit dem  Radius: " + this.radius +
               " Center1:" + this.center1 + " Center2: " +
               this.center2 + " Länge: " + this.laenge() +
               " Volumen: " + this.volumen();
    }
}
```

Aufbauend auf dieser Typhierarchie wird unser vormals ungültiges Programmfragment jetzt typkonsistent:

```
GeoPrimitive[] basisTeile = new GeoPrimitive[...];
...
GeoPrimitive b;
double gesamtGewicht = 0.0;
double gesamtVolumen = 0.0;
for (int i = 0; i < basisTeile.length; i++) {
   b = basisTeile[i];
   gesamtGewicht = gesamtGewicht + b.gewicht();
   gesamtVolumen = gesamtVolumen + b.volumen();
}
```

Die Variable *b* ist jetzt zwar immer noch auf den Typ *GeoPrimitive* eingeschränkt. Der Compiler ist sich aber sicher, dass jedes Objekt in dem Array *basisTeile* von einem Subtypen von *GeoPrimitive* instanziiert wurde, der alle abstrakten Operationen implementiert. Bei dem Aufruf von *b.volumen()* wird dann diese Verfeinerung dynamisch gebunden. Das gleiche passiert übrigens bei der Ausführung von *b.gewicht()*: In der Implementierung von *gewicht* wird ja auch *volumen* aufgerufen, so dass diese Operation dann auch dynamisch gebunden werden muss.

Es ist durchaus möglich, dass der Subtyp eines abstrakten Objekttyps selbst auch wieder abstrakt ist. Dafür gibt es zwei Möglichkeiten:

1. Nicht alle geerbten abstrakten Operationen werden verfeinert.

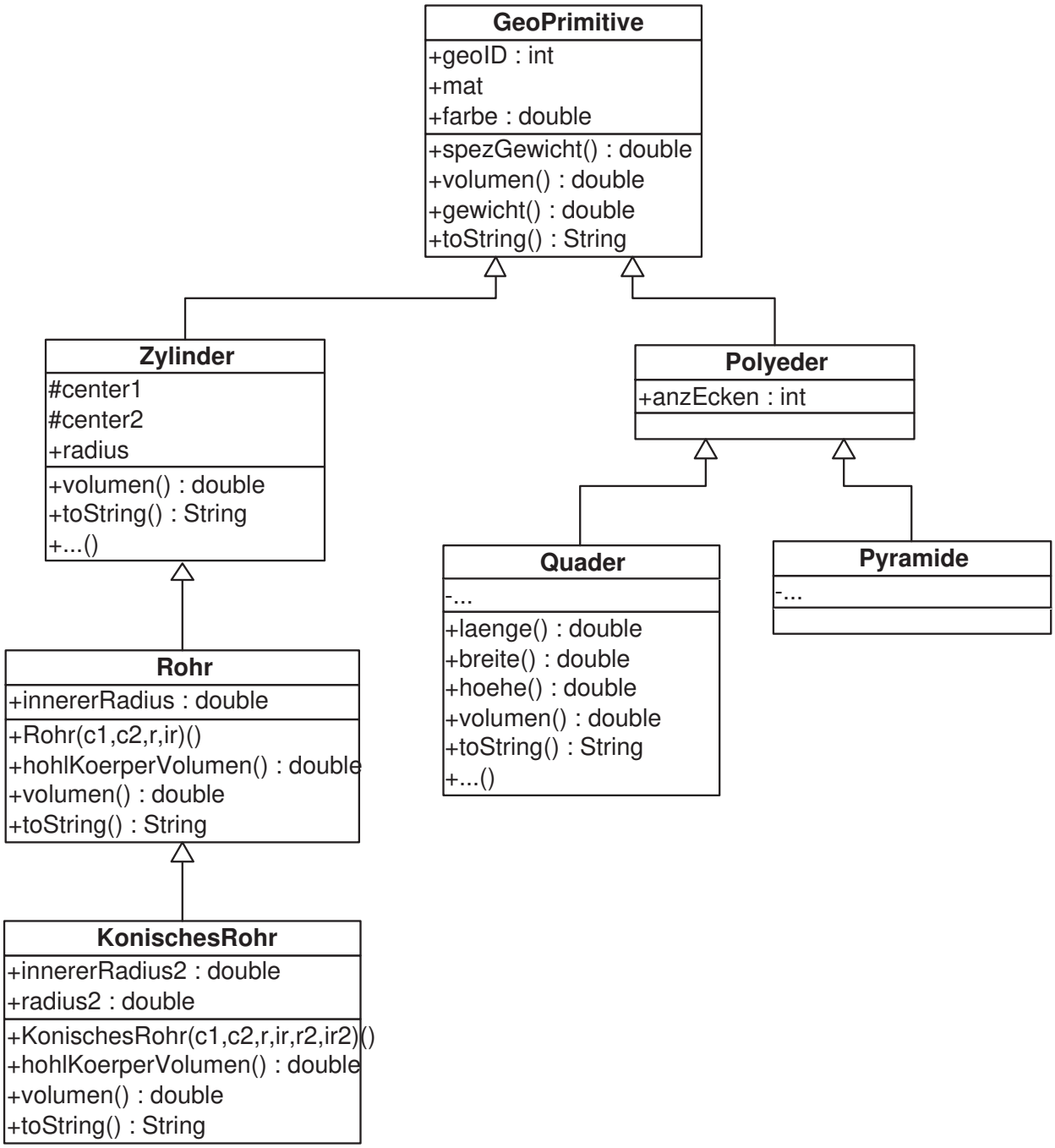



Abbildung 4.1: Geometrische Typenhierarchie mit abstrakten Klassen

2. Der Subtyp führt selbst neue abstrakte Operationen ein.

Gemäß 2. ist es sogar möglich, dass ein nicht-abstrakter Objekttyp selbst abstrakte Subtypen hat.

Der Objekttyp *Polyeder* wird jetzt definiert als abstrakter Subtyp von *GeoPrimitive*:

```
abstract class Polyeder extends GeoPrimitive {
    public int anzahlEcken;
    // ...
} // end abstract class Polyeder
```

*Polyeder* verfeinert also keine der beiden geerbten abstrakten Operationen. Dies „überlässt“ er seinen Untertypen *Quader* und *Pyramide*, wovon wir nur den *Quader* hier zeigen:

```
public class Quader extends Polyeder{
    private Vertex v1, v2, v3, v4, v5, v6, v7, v8;

    public Quader(Material m) {  // Konstruktor: Einheitswürfel
        this.v1 = new Vertex(0.0,0.0,0.0);
        this.v2 = new Vertex(1.0,0.0,0.0);
        this.v3 = new Vertex(1.0,1.0,0.0);
        this.v4 = new Vertex(0.0,1.0,0.0);
        this.v5 = new Vertex(0.0,0.0,1.0);
        this.v6 = new Vertex(1.0,0.0,1.0);
        this.v7 = new Vertex(1.0,1.0,1.0);
        this.v8 = new Vertex(0.0,1.0,1.0);
        this.mat = m; this.anzahlEcken = 8;
    }

    public double laenge() {
        return this.v1.distanz(this.v5);
    }
    public double breite() {
        return this.v1.distanz(this.v2);
    }
    public double hoehe() {
        return this.v1.distanz(this.v4);
    }
    public double volumen() {
        return this.laenge() * this.hoehe() * this.breite();
    }
    public double gewicht() {
        return this.volumen() * this.mat.spezGewicht;
    }

    public String toString() {
       return("Quader der Dimension " + this.laenge() + " X "
              + this.breite() + " X " + this.hoehe());
    }
```

```
    // ...

} // public class Quader
```

## 4.2 Schnittstellen

Zusätzlich zu abstrakten Klassen gibt es auch das Konzept der Schnittstelle (Interface in der Java-Terminologie). Eine abstrakte Klasse kann eine partielle Implementierung des Objekttyps enthalten. Diese partielle Implementierung besteht aus Attributen und einigen implementierten Operationen. Dies ist in einer Schnittstelle nicht möglich; sie darf nur Operationen-Signaturen und Konstanten enthalten.

In einer anderer Hinsicht sind Schnittstellen mächtiger als abstrakte Klassen: Eine Klasse kann nicht von mehreren abstrakten Klassen erben, da es in Java keine Mehrfachvererbung gibt. Im Gegensatz dazu kann eine Klasse aber sehr wohl mehrere Schnittstellen implementieren.

Als Beispiel wollen wir eine Schnittstelle *QuaderStack* spezifizieren und zwei mögliche Implementierungen dieser Schnittstelle skizzieren. Graphisch sind die Schnittstellen- und Klassendefinitionen in Abbildung 4.2 gezeigt.

```
interface QuaderStack {
    public boolean isEmpty();
    public void push(Quader c);
    public Quader pop();
}

class QuaderStackMitArray implements QuaderStack {
    private Quader[] storage;
    private int cardinality;
    ...
}

class QuaderStackMitVerketteterListe implements QuaderStack {
    private QuaderKnoten top;  // class QuaderKnoten muss
    private int cardinality;   // als "Container" für einen Quader
        ...                    // und einem Attr. QuaderKnoten definiert sein
}
```

Interface-Operationen und Konstanten sind defaultmäßig alle *public*. Deshalb hätte man den *access modifier* **public** auch weglassen können.

In einem Programm kann man dann die beiden unterschiedlichen Implementierungsvarianten der Schnittstelle verwenden. Diese unterscheiden sich hinsichtlich ihres Verhaltens gegenüber einem Klienten nicht – sie sind aber hinsichtlich der Leistungsfähigkeit sicherlich unterschiedlich:

```
...
QuaderStack turmLinks, turmMitte, turmRechts;
turmLinks = new QuaderStackMitArray(...);
turmMitte = new QuaderStackMitVerketteterListe(...);
```

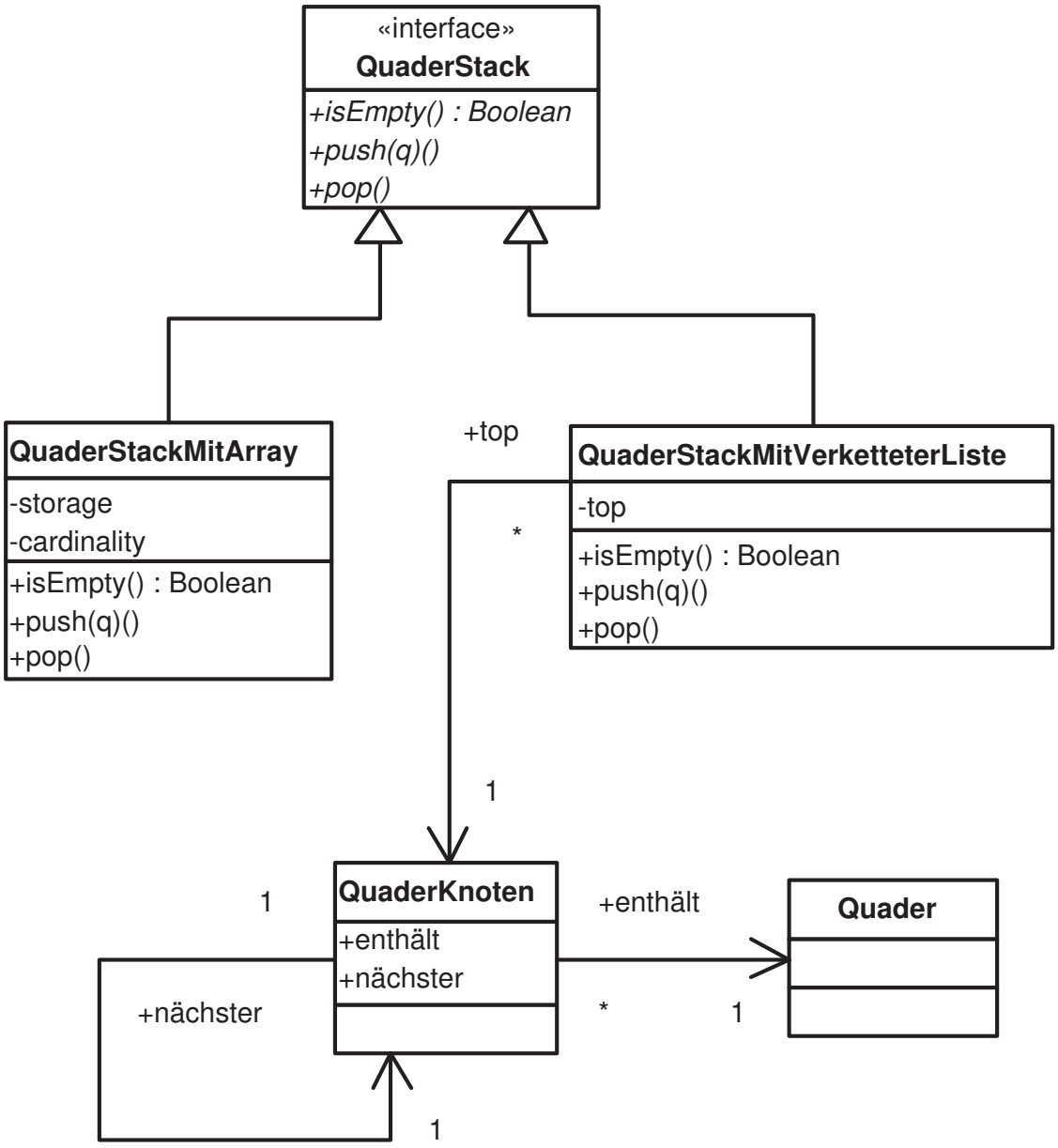



Abbildung 4.2: Schnittstelle QuaderStack und zwei Implementierungen

```
turmRechts = new QuaderStackMitArray(...);
...
turmLinks.push(new Quader(...));
turmMitte.push(turmLinks.pop());
...
```

Wenn man gegen Schnittstellen programmiert, d.h. nur die Operationen verwendet, die in der Schnittstelle definiert sind, kann man sehr einfach die konkrete Klasse, die man verwendet, gegen eine andere austauschen. Dies nennt man auch *lose Kopplung* und ist oft erstrebenswert.

## 4.3 Typ-Anfragen und Type-Casting

Mit abstrakten Klassen kann man solche Operationen, die in *allen* Untertypen definiert werden, herausfaktorisieren. Durch das dynamische Binden wird dann die spezielle Implementierung, die im jeweiligen Untertyp definiert wurde, während der Laufzeit gebunden. Dadurch erhöhen die abstrakten Klassen die Flexibilität des Objektmodells. Allerdings müssen die Implementierungen in den Untertypen die Verfeinerungsbedingungen beachten; d.h. die Signatur muss mit der Signatur der geerbten abstrakten Operation eins-zu-eins übereinstimmen. Die Verfeinerungen müssen also hinsichtlich Anzahl und Typen der Parameter übereinstimmen. Manchmal hat man aber logisch verwandte Operationen, die sehr unterschiedliche Parameter benötigen. Dies lässt sich dann nicht durch Verfeinerungen realisieren. Wenn es *gar nicht anders geht* um eine natürliche Modellierung der „realen Welt“ zu erzielen, kann man in Ausnahmefällen auch auf Typ-Abfragen und Type-Casting zurückgreifen, um das Verhalten einer Operation an die unterschiedlichen Subtypen, denen das Argument der Operation angehören kann, anzupassen. Wir wollen dies anhand eines Beispiels erläutern. Es soll – vielleicht zu Weihnachten – eine besonders engagierte Person, die über die Variable *preistraeger* referenziert wird, ausgezeichnet werden. Dies ist in dem nachfolgenden Programmfragment realisiert. Wir gehen dabei von der in Abbildung 4.3 gezeigten Typhierarchie aus.

Wir haben bewusst nur die Attribute aufgeführt, die für das Programm *Auszeichnung* relevant sind:

- *Student*en haben das Attribut *notenSchnitt*
- Ein/e *Angestellter* hat das Attribut *gehalt* vom Typ *double*
- *Manager* haben das Attribut *dienstWagen*, das ein Auto, das sie fahren dürfen, referenziert.

Wie sieht nun die Typ-spezifische Auszeichnung aus, die je nach dem direkten Typ der von *preistraeger* referenzierten Person sehr unterschiedlich ist:

- Ein *Student* bekommt einen „upgrade“ seines *notenSchnitt*s.
- Ein *Angestellter* bekommt eine *gehalt*s-Erhöhung.
- Ein *Manager* wird dadurch ausgezeichnet, dass er den Jaguar fahren darf.

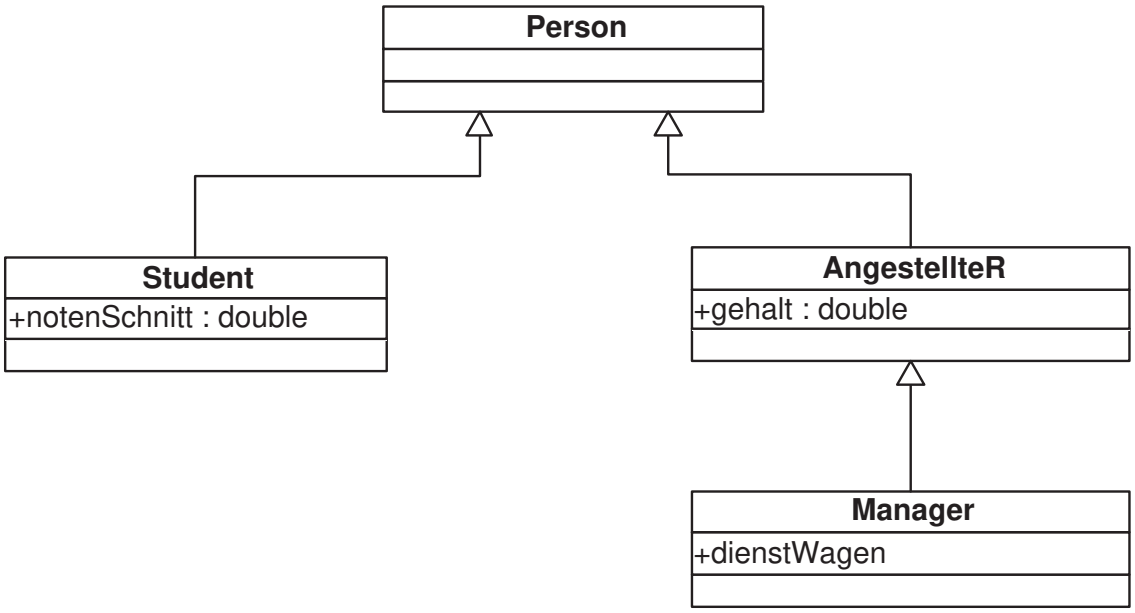



Abbildung 4.3: Typ-Hierarchie zur Erläuterung von Typ-Abfragen und Type-Casting

- Eine „normale“ *Person* bekommt lediglich ein Dankesschreiben.

Diese Auszeichnung kann wie folgt realisiert werden:

```
Object preistraeger;    // sollte eine Person sein ... but who knows
Car jaguar = new Car("Jaguar",340,250);  // 340 PS, 250 km/h
double notenSchnittBonus = 0.9;   // default notenSchnitt-bonus
double gehaltBonus = 1.1;  // default gehalt-bonus
   // ...
        if (preistraeger instanceof Manager) {
            Manager m = (Manager)preistraeger;
            m.dienstWagen = jaguar;
            System.out.println("new car");
        }
        else if (preistraeger instanceof Angestellter) {
            Angestellter e = (Angestellter)preistraeger;
            e.gehalt = e.gehalt * gehaltBonus;
            System.out.println("Gehalt erhöht.");
        }
        else if (preistraeger instanceof Student) {
            Student s = (Student)preistraeger;
            s.notenSchnitt = s.notenSchnitt * notenSchnittBonus;
            System.out.println("NOTENSCHNITT upgrade.");
        }
        else if (preistraeger instanceof  Person)
            System.out.println("Großartig -- Danke!");
        else
            System.out.println("Deinen Typ kenne ich nicht!");
    }
```

Der Ausdruck `(preistraeger instanceof Manager)` ist erfüllt, falls das Objekt *preistraeger* eine direkte oder indirekte Instanz der Klasse *Manager* ist. Das heißt, wenn `(preistraeger instanceof Manager)` gilt, dann auch `(preistraeger instanceof Angestellter)`. Dies gilt wegen der Mengen-Inklusion der Extension des Subtypen *Manager* in der Extension des Obertypen *Angestellter*. Deshalb ist es essentiell, dass wir mit der Typabfrage beim speziellsten Objekttyp anfangen und uns „hocharbeiten“ zum generischeren Objekttyp. Warum? (Sonst würden die Manager niemals den Jaguar bekommen . . . )

Nachdem wir den *preistraeger* durch die Typabfrage als Instanz eines speziellen Typs identifiziert haben, müssen wir noch einen *Type-Cast* durchführen, damit wir auch tatsächlich auf die Attribute dieses speziellen Typs zugreifen können. Andernfalls würde der Compiler einen potenziellen Typfehler ermitteln. Durch den *Type-Cast* teilen wir dem Compiler sozusagen mit: “Wir wissen was wir tun – sei unbesorgt”. Syntaktisch erfolgt der *Type-Cast* durch `(Manager)preistraeger`. In Java können solche *Type-Casts* auch „misslingen“ und zu einem Laufzeitfehler führen, wenn wir versuchen ein Objekt in einen Objekttyp zu „casten“, der weder mit dem direkten Typ des Objekts übereinstimmt noch einem Obertyp des direkten Typs des Objekts entspricht. Deshalb sollte man aus Sicherheitsgründen

- *Type-Casts* sehr selten einsetzen,
- es nur dann tun, wenn man genau weiß, dass das referenzierte Objekt mit dem Typ kompatibel ist und
- sicherheitshalber vorher noch mal fragen – mit *instanceof*.

Nachdem der *preistraeger* in *Manager* „ge-cast-ed“ wurde, kann man auf das Attribut *dienstWagen* zugreifen.

Wie gesagt, sollte man *Type-Casts* nur sehr selten verwenden, da sie die Typisierungsregeln von Java außer Kraft setzen.

## 4.4 Übungen

**4.1** The *abstract types* are only meaningful in strongly typed object models in which data components are constrained to a particular type. Why is the abstract type concept not needed in non-type safe models, i.e., those models where all data components can refer to any object of any possible type? Illustrate your discussion on an example application.

**4.2** Sketch the type definitions for the following types:

- *RealEstate* being an abstract type with the abstract operation *estimatedValue*.
- *Factory* being a subtype of *RealEstate* with attributes *profit*, *numberOfEmployees*, and *sales*. The abstract operation *estimatedValue* should be refined and coded as a function of *profit*, #*employees*, and *sales*.
- *Housing* being an abstract subtype of *RealEstate* with an additional attribute *yearBuilt*.

- *Condominium* is a subtype of *Housing*. The operation *estimatedValue* is coded as a function of *yearBuilt* and *size*.
- *House* is also a subtype of *Housing* and contains also a refinement of *estimatedValue*—now as a function of *size* and *yearBuilt* and *yardSize*.

**4.3** Illustrate the use of **type casting** on extracting particular typed objects from a heterogeneous array. Consider, for example, a given array *myGeoObjects* of type *GeoPrimitive[]* (see Figure 4.1). This array is suitable to store any kind of geometric objects. In an application however, we may have to manipulate the objects depending on their more specific type. Therefore, we want to separate the elements into *myZylinders*, *myPipes*, *myConicalPipes*, and *myPolyeders*. Note that the array referenced by *myZylinders* should contain only direct instances of *Zylinder* and, e.g., no *Pipes*. Why is it advantageous to constrain *myZylinders* to the type *Zylinder[]* as opposed to *GeoPrimitive[]*?

**4.4** In Exercise 4.1 we stated that the abstract type concept is not needed in non-type safe models that impose no type constraints on data components. This is not true for the **instanceof** operator for querying objects about their type. On the opposite, the **instanceof** operator would be even more important in non-type safe models. Discuss and illustrate this statement on some example application. What about type casting—is it needed in non-type safe models?

**4.5** In the text we indicated that the bonus program fragment does not violate type safety. Elaborate this topic:

- Why is it required that at least one branch applies for each possible type encountered?
- Illustrate your discussion on an example type hierarchy of your choice (but it should not be the one used in the text).

**4.6** Provide for each type in the hierarchy of Figure 4.3 a *laudat* operation without any parameter. These operations should behave like the specialized code to honor a *preistraeger* given in the text. Do not use type casting! Give a second implementation of these operations in which the *notenSchnittBonus* and the *gehaltBonus* are given as parameters instead of being hard coded into the operation.

## 4.5 Annotated Bibliography

Virtual types were already incorporated in the language Simula 67 [Dahl und Nygaard (1966), Nygaard und Dahl (1981), Dahl, Myrhaug und Nygaard (1970)]. The virtual type concept is also used in Eiffel [Meyer (1988)]—there it is called *deferred class*, though. Also, C++ [Stroustrup (1990)] provides some form of virtual type. The *instanceof* construct can be traced back to Smalltalk-80 [Goldberg und Robson (1983)], where the programmer can inquire the type of an object and then, depending on the outcome of this message, channel the control flow appropriately. In some object models, e.g., in GOM [Kemper und Moerkotte (1994)] a more convenient and type-safe construct called a **type case** was introduced. This construct is like a switch-statement where the branching depends on the outcome of an **instanceof** type query.

# 5. Aufzählungstypen

Ein Datentyp für die Repräsentation anwendungsspezifischer Mengen ist der Aufzählungstyp (engl. *enumerated type; enum*). Dieser Datentyp ist in Java erst seit der Version 1.5 verfügbar. Andere Programmiersprachen, wie C, C++ und C#, bieten Aufzählungstypen schon seit langem an. Vor der Verfügbarkeit von Aufzählungstypen in Java musste man sich mit Konstanten behelfen, beispielsweise, um sich eine Aufzählung aller Monate zu definieren:

```
1    public static final int MONTH_JAN =  1;
2    public static final int MONTH_FEB =  2;
3    ...
4    public static final int MONTH_NOV = 11;
5    public static final int MONTH_DEC = 12;
```

Diese Notlösung birgt aber einige Schwächen:

- Sie ist nicht typsicher, da es sich bei den Monaten im Grunde nur um *int*-Werte handelt. Dadurch kann man beispielsweise zwei Monate addieren, was keinen Sinn ergibt. Wird ein Monat als Parameter erwartet, kann man einen beliebigen *int*-Wert übergeben. Bei einer nachträglichen Änderung von Konstantenwerten ergibt sich dann ein unvorhersehbares Verhalten. Ein weiterer Nachteil der Lösung mit Konstanten fällt auf, wenn man Methoden mit mehreren Parametern, wie beispielsweise Tag und Monat, betrachtet. Vertauscht man aus Versehen die Reihenfolge der Parameter beim Aufruf, hat der Compiler keine Möglichkeit, dies festzustellen, da es sich lediglich um zwei *int*-Parameter handelt. Derartige Fehler sind oftmals schwer zu finden.

- Die gezeigte Lösung definiert Konstanten, die vom Java-Compiler aus Optimierungsgründen in die Klassen, die diese Konstanten verwenden, hineinkompiliert werden. Ändert man die Konstanten, beispielsweise durch Einfügen einer neuen Konstante zwischen zwei existierenden, muss man alle Klassen, die diese Konstanten verwenden, neu übersetzen. Ohne Neuübersetzung erhält man ein undefiniertes Ergebnis.

- Gibt man die Konstanten beispielsweise zu Testzwecken aus, werden nur die *int*-Werte ausgegeben, nicht die Namen der Konstanten. Durch diesen *int*-Wert erhält man allerdings weder Aussagen über den Typ noch über die Semantik des Werts.

Mit einem Aufzählungstyp kann man die Konstanten ersetzen durch

```
1    public enum Month { JAN, FEB, MAR, APR, MAY, JUN,
2        JUL, AUG, SEP, OCT, NOV, DEC };
```

Mit dieser Anweisung definieren wir den Datentyp *Month*, der den angegebenen Wertebereich hat. Auf die einzelnen Werte kann man folgendermaßen zugreifen:

```
1    System.out.println(Month.JAN);
```

Das Beispiel zeigt auch, dass die Ausgabe (in diesem Fall `JAN`) von Aufzählungstypen informativer ist, als die Ausgabe von *int*-Konstanten.

Die Aufzählungstypen in Java sind mächtiger als bisher gezeigt. So lassen sich Aufzählungstypen weitere Informationen und sogar Methoden zuweisen[1]. Standardmäßig bieten Aufzählungstypen eine statische *values()*-Methode. Diese Methode gibt ein Array zurück, das alle Werte des Aufzählungstypen in der Reihenfolge der Definition enthält. Diese erweiterten Möglichkeiten der Aufzählungstypen wollen wir kurz in einem Beispielprogramm vorstellen:

```
1  public class Enums {
2    public enum Month {
3      JAN  (31,  -2.2), FEB  (28,  -0.8), MAR  (31,   3.1),
4      APR  (30,   9.0), MAY  (31,  12.7), JUN  (30,  15.9),
5      JUL  (31,  20.1), AUG  (31,  17.1), SEP  (30,  15.4),
6      OCT  (31,   7.8), NOV  (30,   3.1), DEC  (31,  -0.8);
7
8      private final int     days;
9      private final double  avgTemperature;
10
11     Month(int days, double avgTemperature) {
12       this.days = days;
13       this.avgTemperature = avgTemperature;
14     }
15
16     public int    days()           { return days;  }
17     public double avgTemperature() { return avgTemperature; }
18
19     public double fractionOfYear() {
20       return days / 365.0;
21     }
22   };
23
24   public static void prettyPrint(Month m) {
25     System.out.println("Monat:␣" + m + ",␣Anzahl␣Tage:␣" + m.days() +
26                        ",␣Anteil␣am␣Jahr:␣" +  m.fractionOfYear());
27   }
28
29   public static void main(String[] args) {
30     prettyPrint(Month.JAN);
31
32     for (Month m : EnumSet.range(Month.FEB, Month.MAY))
33       System.out.println(m);
34   }
35 }
```

In dem Beispiel wird in den Zeilen 2–6 der Aufzählungstyp *Month* definiert. Jedem Monat werden dabei zwei Werte zugewiesen: die Anzahl der Tage des Monats und die Durchschnittstemperatur. Der Rest der Definition des Aufzählungstyps bis Zeile 22 entspricht dem einer Klasse. Die Informationen, die den Monaten zugewiesen sind, werden dem in den Zeilen 11–14 angegebenen Konstruktor übergeben. Dieser speichert die Werte in den Instanzvariablen *days* und *avgTemperature*. Die beiden Methoden *days()* und *avgTemperature()* erlauben den lesenden Zugriff auf die abgespeicherten Werte. Die Methode *fractionOfYear()* berechnet den Anteil eines Monats an einem Jahr. Die Methode *prettyPrint(m)* zeigt die Verwendung des Typs *Month* als Parametertyp und gibt alle Informationen über einen Monat übersichtlich aus. In der *main()*-Methode demonstrieren wir die Verwendung des Aufzählungstyps. Java bietet die zwei Klassen *java.util.EnumSet* und *java.enum.EnumMap* zur komfortablen Arbeit mit Aufzählungstypen. Der Aufruf der Methode *EnumSet.range(Month.FEB, Month.MAY)* liefert beispielsweise eine Aufzählung der Monate von Februar bis einschließlich Mai.

Aufzählungstypen können auch in *switch*-Anweisungen verwendet werden, wie folgendes Beispiel zeigt:

[1] Intern verwaltet Java Aufzählungstypen als Klassen.

```
1   Month m = Month.JAN;
2
3   switch (m) {
4     case SEP:
5     case OCT:
6     case NOV:
7     case DEC:
8     case JAN:
9     case FEB:
10    case MAR:
11    case APR:
12      System.out.println("kalt");
13      break;
14    case MAY:
15    case JUN:
16    case JUL:
17    case AUG:
18      System.out.println("warm");
19      break;
20  }
```

Wie wir gesehen haben, sind Aufzählungstypen der Verwendung von Konstanten vorzuziehen. Aufzählungstypen sollten immer dann verwendet werden, wenn eine feste Anzahl von Konstanten benötigt wird und diese Konstanten bereits alle zur Übersetzungszeit bekannt sind. Dabei ist es allerdings nicht nötig, dass die Menge der Werte eines Aufzählungstyps für immer und ewig konstant ist. Beispielsweise können für neue Versionen eines Programms neue Werte hinzugefügt werden.

## 5.1 Übungen

**5.1** Implementieren Sie eine Klasse, die eine Karte eines Kartenspiels repräsentiert. Alle möglichen Kartenfarben und -werte sollen jeweils durch einen Aufzählungstyp repräsentiert werden. Schreiben Sie eine Methode, die einen vollständigen Satz Spielkarten generiert und in einem Array zurückgibt. Schreiben Sie eine weitere Methode, die das Austeilen von Karten simuliert. Dabei sollen jeweils fünf Karten an 4 Spieler ausgeteilt werden. Geben Sie auf dem Bildschirm die Karten der jeweiligen Spieler aus.

**5.2** Definieren Sie einen Aufzählungstyp, der alle Planeten inklusive Masse und Durchmesser enthält. Implementieren Sie eine Methode für den Aufzählungstyp, die die durchschnittliche Dichte des Planeten berechnet und ausgibt.

# 6. Generische Typen

Mit Java 1.5 wurden *generische Typen* (engl. *generics*) in Java eingeführt. Mit Hilfe von Generics kann man durch Typparameter von konkreten Typen abstrahieren. Häufig genutzt werden Generics bei der Implementierung von Kollektionen, also Sammlungen von Objekten. Wir werden einen einfachen *Stack* als Beispiel für einen Kollektionstyp verwenden. Einen Stack kann man sich als Stapel Teller vorstellen: Man kann einen neuen Teller immer nur oben auf den Stapel legen und auch immer nur den obersten Teller vom Stapel herunternehmen. Üblicherweise spricht man bei den Objekten, die eine Kollektion verwaltet, von *Elementen*.

## 6.1 Motivation

Eine einfache Implementierung eines Stacks für *Quader*-Objekte ist in Abbildung 6.1 links gezeigt. Mit dem Konstruktor in den Zeilen 5–7 kann man einen Stack mit einer festen Größe instanziieren. Mit *push(e)* legen wir ein Element auf dem Stack ab, während wir mit *pop()* das oberste Element vom Stack herunternehmen. Die Methode *isEmpty()* beantwortet, ob der Stack leer ist, oder nicht. Wenn wir einen *ZylinderStack* benötigen, müssen wir den Code der *QuaderStack*-Klasse replizieren und die entsprechenden Vorkommen (Attribut-, Rückgabe- und Parametertypen) der Klasse *Quader* durch die Klasse *Zylinder* ersetzen. Die dabei entstehende, zur Ursprungsklasse beinahe identische Klasse, ist in Abbildung 6.1 rechts gezeigt. Dieser *ZylinderStack* kann natürlich nicht nur *Zylinder*-Objekte aufnehmen, sondern wegen der Substituierbarkeit auch *Rohr*- und *KonischesRohr*-Objekte. Ein *Quader*-Objekt kann allerdings nicht in diesen Stack gelegt werden.

Wir sehen an dem kleinen Beispiel sehr deutlich ein Problem: Wir müssen den Code von *QuaderStack* replizieren und manuell ändern, um einen *ZylinderStack* zu erhalten.

```
public class QuaderStack {
  private Quader[] elements;
  private int cardinality = 0;

  public QuaderStack(int capacity) {
    elements = new Quader[capacity];
  }
  public void push(Quader e) {
    elements[cardinality++] = e;
  }
  public Quader pop() {
    return elements[--cardinality];
  }
  public boolean isEmpty() {
    return (cardinality == 0);
  }
}
```

```
1  public class ZylinderStack {
2    private Zylinder[] elements;
3    private int cardinality = 0;
4
5    public ZylinderStack(int capacity) {
6      elements = new Zylinder[capacity];
7    }
8    public void push(Zylinder e) {
9      elements[cardinality++] = e;
10   }
11   public Zylinder pop() {
12     return elements[--cardinality];
13   }
14   public boolean isEmpty() {
15     return (cardinality == 0);
16   }
17 }
```

Abbildung 6.1: Die Klassen *QuaderStack* und *ZylinderStack*

Dies erfordert – zumindest bei komplexeren Kollektionstypen – einen nicht unerheblichen Arbeitsaufwand und bietet auch zahlreiche Fehlerquellen. Weitere, nicht minder schwerwiegende Probleme liegen in der Pflege der entstandenen Klassen: Beheben wir einen Fehler in *QuaderStack* oder fügen eine weitere Methode hinzu, müssen wir das manuell in alle anderen Stack-Implementierungen übertragen. Eine Möglichkeit, diese unerwünschte Code-Verdoppelung zu verhindern, liegt darin, einen generischen Stack zu implementieren. Mit den Mitteln, die wir bisher kennengelernt haben, heißt das, dass wir einen *Object*-Stack implementieren. Weil jede Klasse von *Object* abgeleitet ist, können wir durch die Substituierbarkeit Objekte beliebigen Typs in einem derartigen Stack verwalten. Unsere Implementierung sieht damit folgendermaßen aus:

```
1  public class Stack {
2    private Object[] elements;
3    private int cardinality = 0;
4
5    public Stack(int capacity) {
6      elements = new Object[capacity];
7    }
8    public void push(Object e) {
9      elements[cardinality++] = e;
10   }
11   public Object pop() {
12     return elements[--cardinality];
13   }
14   public boolean isEmpty() {
15     return (cardinality == 0);
16   }
17 }
```

Diese Möglichkeit wurde auch in den Kollektionstypen von Java vor der Version 1.5 verwendet. Auf diese Weise wird zwar die Verdoppelung des Codes vermieden, allerdings geben wir damit auch teilweise die Typsicherheit auf, die uns der Java-Compiler bieten kann, da wir Casts verwenden müssen. Dadurch ist die Verwendung der Klasse *Stack* unbequemer, als die der spezialisierten Stacks, und außerdem unsicherer. Wir wollen das an folgendem Beispiel verdeutlichen:

```
1  // Quader q anlegen
2  // ...
3  QuaderStack myQuaderStack = new QuaderStack(5);
4  myQuaderStack.push(q);
5  Quader q2 = myQuaderStack.pop();
6
7  Stack myQuaderStack2 = new Stack(5);
8  myQuaderStack2.push("Dies␣führt␣später␣zu␣einem␣Fehler!");
9  myQuaderStack2.push(q);
10 Quader q3 = (Quader)myQuaderStack2.pop();
11 Quader q4 = (Quader)myQuaderStack2.pop();
```

Wir sehen, dass bei der Verwendung der *Stack*-Klasse Casts nötig sind, um das Ergebnis der *pop()*-Methode einer *Quader*-Variable zuzuweisen. Die Casts auf *Quader* in den Zeilen 10 und 11 drücken das Wissen (oder besser gesagt die Hoffnung) aus, dass in *myQuaderStack2* nur *Quader*-Objekte abgelegt wurden, die Casts also gültig sind. Allerdings weiß der Compiler nicht, dass wir den Stack *myQuaderStack2* nur für *Quader*-Objekte verwenden wollten, deshalb kompiliert er den gezeigten Programmabschnitt anstandslos. Während der Ausführung des Programms tritt allerdings eine *ClassCastException* auf, da in Zeile 11 ein *String* zurückgeliefert wird und nicht, wie vom Programmierer erwartet, ein *Quader*-Objekt. Verhängnisvoll daran ist, dass der eigentliche Programmierfehler darin besteht, dass ein String in den Stack *myQuaderStack2* eingefügt worden ist, obwohl der Stack für die Verwaltung von *Quader*-Objekten gedacht war. Eine Excepti-

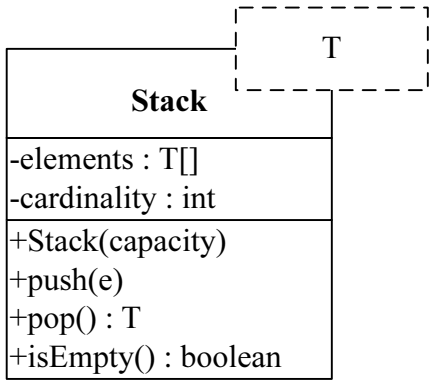
on erhalten wir erst, wenn wir dieses Objekt wieder vom Stack nehmen wollen. Derartige Fehler sind in realen Programmen oftmals sehr schwer zu lokalisieren, da möglicherweise an vielen verschiedenen Stellen im Code Objekte in eine Kollektion eingefügt werden.

Wir benötigen uns also eine – über Subtypisierung und Vererbung hinausgehende – Möglichkeit, Kollektionen zu implementieren. Die Verwendung der Kollektionen soll so bequem und sicher sein, wie die von spezialisierten Kollektionen, die nur Objekte von bestimmten Typen speichern können. Allerdings sollen die Kollektionen so flexibel sein, dass sie beliebige Objekte verwalten können. Genau diese Möglichkeit bieten uns generische Typen, die es uns ermöglichen, statt konkreter Typen Typvariablen zu verwenden.

## 6.2 Generische Typen in Java

Generische Typen (oder auch parametrisierte Typen) ermöglichen es uns, generische Klassen (oder auch Schnittstellen) zu implementieren, ohne die Typen von allen Komponenten (Attribute, Parameter und Rückgabewerte) festlegen zu müssen. Für die Komponenten, deren Typ wir nicht fest angeben wollen, verwenden wir einfach Typparameter. In Java werden dazu normalerweise einzelne Großbuchstaben verwendet. Im Falle des Stack-Beispiels können wir also statt *Quader* oder *Zylinder* einen Typparameter, z.B. *T*, nutzen, um den Typ der Elemente, die in der Kollektion gespeichert werden sollen, generisch anzugeben. Während der Instanziierung einer generischen Klasse muss dann angegeben werden, für welchen konkreten Typ ein Typparameter stehen soll. Betrachten wir zuerst die generische Implementierung des Stacks mit Typparameter *T* und das zugehörige UML-Diagramm:

```
1  public class Stack<T> {
2     private T[] elements;
3     private int cardinality = 0;
4
5     public Stack(int capacity) {
6        elements = (T[])new Object[capacity];
7     }
8     public void push(T e) {
9        elements[cardinality++] = e;
10    }
11    public T pop() {
12       return elements[--cardinality];
13    }
14    public boolean isEmpty() {
15       return (cardinality == 0);
16    }
17 }
```




Eine generische Klasse sieht in UML also fast aus, wie eine normale Klasse. Der einzige Unterschied besteht darin, dass oben rechts an der Klasse ein gestricheltes Rechteck angebracht ist, in dem die Typparameter der Klasse stehen. Wenden wir uns nun dem Programm zu: In Zeile 1 geben wir nach dem Klassennamen in spitzen Klammern an, welche Typparameter die Klasse hat. Im gezeigten Beispiel existiert nur der Parameter *T*, es können aber auch mehrere, durch Komma getrennte Typparameter angegeben werden. Lesen kann man den Ausdruck „*Stack<T>*“ als „Stack vom Typ *T*“. Wie bereits beschrieben, verwenden wir den Typparameter an allen Stellen, an denen wir vorher einen konkreten Typen für die Elemente des Stacks angegeben haben. Eine Besonderheit findet sich in Zeile 6: Hier legen wir ein *Object*-Array an und casten es hinterher in ein *T*-Array. Diese Konstruktion ist in Java derzeit leider unvermeidbar und hängt damit zusammen, wie der

Java-Compiler Generics umsetzt. Wir werden später genauer darauf eingehen, an dieser Stelle ist nur wichtig, dass ein *T*-Array angelegt wird. Verwendet wird eine generische Klasse wie der Stack folgendermaßen:

```
1 // Quader q anlegen
2 // ...
3 Stack<Quader> myQuaderStack = new Stack<Quader>(5);
4 myQuaderStack.push(q);
5 // myQuaderStack.push("Dies würde der Compiler beim Übersetzen monieren!");
6 Quader q2 = myQuaderStack.pop();
```

In Zeile 3 legen wir einen *Stack* an, der *Quader*-Objekte (und natürlich alle dafür substituierbaren Objekttypen) verwalten kann. Durch die Verwendung des Typparameters ist es nun möglich, das Wissen des Programmierers, dass nur *Quader*-Objekte auf *myQuaderStack* abgelegt werden dürfen, dem Compiler verfügbar zu machen. Dadurch kann der Compiler auf die Einhaltung dieser Bedingung während des Übersetzens von Programmen achten und damit Typsicherheit garantieren.

Die Verwendung dieser Klasse ist dadurch genauso typsicher und bequem, wie die der auf einen Typen spezialisierten Stacks *QuaderStack* und *ZylinderStack*, aber dabei sehr viel flexibler, da bei der Instanziierung eines Stacks angegeben werden kann, welche Objekttypen darin abgelegt werden können. Außerdem müssen wir den Code für einen Stack nicht für jeden neuen Elementtyp replizieren und anpassen, was der Wiederverwendbarkeit des Codes sehr zuträglich ist. Ablegen (Zeile 4) von Elementen auf dem Stack und Herausholen von Elementen (Zeile 6) aus dem Stack können ohne lästige Casts vorgenommen werden. Das Einfügen eines *String*-Objekts, das bei der Implementierung eines generischen Stacks durch Zurückgreifen auf *Object* noch zu schwer zu findenden Fehlern führen konnte, kann bei der Verwendung von Typparametern vom Compiler verhindert werden, da die Methode *push(T e)* einen Parameter vom Typ *T*, in diesem Falle also vom Typ *Quader* erwartet.

## 6.3 Java Collections Framework

Seit JDK 1.2 gibt es das generische Java Collections Framework. Dieses bietet neben einem *Stack*, der ganz ähnlich zu unserem Beispiel ist, noch viele weitere generische Kollektionstypen, die beliebige Objekte enthalten können. Ein kleiner (und vereinfachter) Ausschnitt aus der Klassen- und Schnittstellenhierarchie ist in Abbildung 6.2 gezeigt.

Nahezu alle Kollektionstypen basieren auf dem gemeinsamen Interface *Collection*. Diese Schnittstelle definiert die Methoden *add()* und *remove()* um Objekte zu einer Kollektion hinzuzufügen und wieder daraus zu entfernen. Weiterhin spezifiziert sie die Methode *toArray()*, die zu einer Kollektion ein Array erzeugt. Zuletzt definiert die Schnittstelle die Methode *contains()*, die feststellt, ob ein Element in einer Kollektion enthalten ist.

### 6.3.1 Mengen, Listen und Warteschlangen

Die drei Interfaces *Set* für Mengen, *List* für Listen und *Queue* für Warteschlangen erweitern *Collection* um die für diese Datenstrukturen charakteristischen Methoden. Listen sind immer geordnet und können Duplikate enthalten. Mengen können keine Duplikate enthalten und erlauben direkten Zugriff auf ihre Elemente. Eine Warteschlange realisiert eine FIFO-Datenstruktur bei der Elemente hinten eingefügt und vorne entfernt werden.

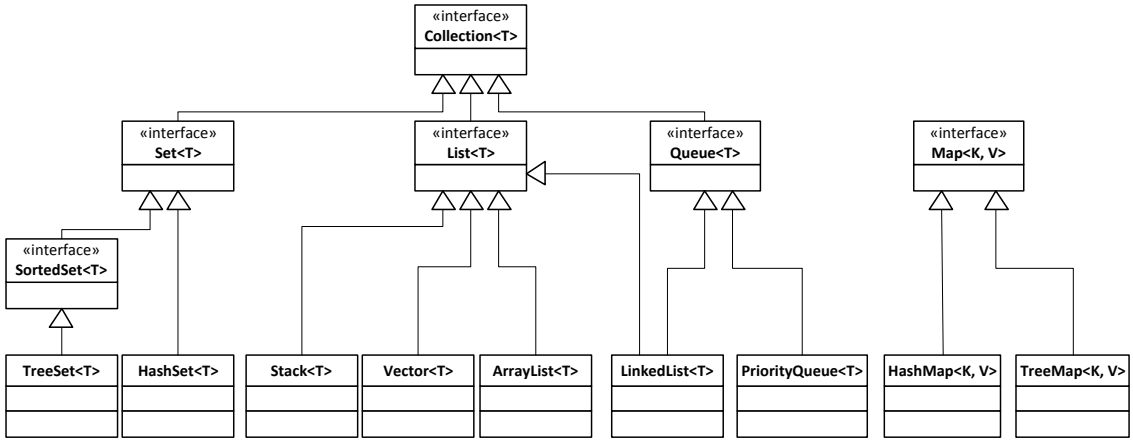



Abbildung 6.2: Überblick über das Java Collections Framework

Das Framework enthält für diese drei Grundtypen verschiedene Implementierungen, die für unterschiedliche Einsatzzwecke geeignet sind. Interessant ist die *LinkedList*, die sowohl das Interface *List* als auch *Queue* implementiert. Sie ist ein Beispiel dafür, dass in Java die Realisierung von mehreren Interfaces im Gegensatz zur Mehrfachvererbung möglich ist.

### 6.3.2 Abbildungen

Das Java Collections Framework enthält weiterhin die weitgehend unabhängige Schnittstelle *Map* (dt. Abbildung). Dies dient als Grundgerüst für Kollektionen, die Schlüssel auf Werte *abbilden*. Kollektionen, die die Schnittstelle *Map* implementieren, legen unter eindeutigen Schlüssel Werte ab, die wiederum durch Punktzugriffe mithilfe des zugehörigen Schlüssels abgefragt werden können. Werte können sie dabei im Gegensatz zu Schlüsseln auch mehrfach enthalten. Die Schnittstelle *Map* wird z.B. von der Hashtabelle *HashMap* realisiert, die einen sehr effizienten Zugriff auf ihre Elemente bietet. Bei der *TreeMap* ist der direkte Zugriff teurer, dafür ermöglicht sie aber auch Bereichsanfragen.

Wir möchten nun diese beiden Abbildungstypen des Java Collection Frameworks am Beispiel eines Telefonbuches genauer vorstellen. Die Telefonbuch-Klasse soll es dem Anwender ermöglichen zu einem Namen die zugehörige Nummer nachzuschlagen. Da wir hier unter einem Schlüssel (dem Namen) einen Wert ablegen wollen (die Nummer), bietet sich an, dass wir eine *Map* verwenden. Hier haben wir jetzt die Wahl zwischen einer *HashMap* und einer *TreeMap*. Die *HashMap* erlaubt das Nachschlagen in konstanter Zeit, d.h. es dauert immer gleich lang einen Eintrag nachzuschlagen – unabhängig davon wie viele Einträge die *HashMap* beinhaltet. Im Gegensatz dazu benötigt die *TreeMap* zum Nachschlagen lineare Zeit, d.h. die Zeitdauer, die für einen einzelnen Nachschlagevorgang benötigt wird, ist proportional zur Anzahl der in der *TreeMap* enthaltenen Elemente. Doch die *TreeMap* bietet auch einen Vorteil gegenüber der *HashMap*: Mit ihr ist es möglich Bereichsanfragen effizient durchzuführen: Für das Telefonbuch bedeutet dies, dass wir z.B. schnell nach allen Nummern suchen können, die für Personen eingetragen sind, deren Namen mit „M“ anfängt. Aus diesem Grund haben wir uns in der nachfolgenden

Implementierung dafür entschieden, eine *TreeMap* zu verwenden.

Unsere Implementierung des Telefonbuchs berücksichtigt außerdem die beliebte Option der Inverssuche, d.h. die Suche nach der Person, die für eine Telefonnummer eingetragen ist. In diesem Fall haben wir uns für eine *HashMap* entschieden, da diese Einfügen und Nachschlagen in konstanter Zeit erlaubt und wir keine Bereichssuchen auf der Telefonnummer durchführen wollen.

Doch nun zur Implementierung selbst: Am Anfang steht der übliche Import der benötigten Pakete. Das Java Collections Framework befindet sich dabei im *java.util* Namensraum. Wir importieren daraus die Schnittstelle *Map*, deren zwei Implementierungen *HashMap* und *TreeMap* sowie die Schnittstelle *Set*, die wir für die Bereichsanfragen benötigen. Anschließend folgt die Definition der Klasse selbst. Zuerst deklarieren wir zwei Datenfelder: `nameToNumber` vom Typ `TreeMap<String, Integer>` für die Abbildung des Namens (vom Typ `String`) auf die Telefonnummer (vom Typ `Integer`) und `numberToName` vom Typ `HashMap<Integer,String>` für die umgekehrte Abbildung von der Telefonnummer auf den Namen.

```
1  import java.util.Map;
2  import java.util.HashMap;
3  import java.util.TreeMap;
4  import java.util.Set;
5
6  // A class representing a phone book
7  class PhoneBook {
8          // Map for the standard lookup
9          TreeMap<String, Integer> nameToNumber;
10         // Map for the reverse lookup
11         HashMap<Integer, String> numberToName;
12
13         // Constructor
14         public PhoneBook() {
15                 nameToNumber = new TreeMap<String, Integer>();
16                 numberToName = new HashMap<Integer, String>();
17         }
18
19         // Add an entry to the phone book
20         void addEntry(String name, Integer phoneNumber) {
21                 nameToNumber.put(name, phoneNumber);    // O(log(n))
22                 numberToName.put(phoneNumber, name);    // O(1)
23         }
24
25         // Standard lookup: get the phone number for a name
26         Integer lookup(String name) {
27                 return nameToNumber.get(name);          // O(log(n))
28         }
29
30         // Reverse lookup: get the name for a phone number
31         String reverseLookup(Integer phoneNumber) {
32                 return numberToName.get(phoneNumber);   // O(1)
33         }
34
35         // Get all entries of the phone book whose names lie in the given range
36         Set<Map.Entry<String, Integer>> rangeLookup(String from, String to) {
37                 return nameToNumber.subMap(from, to).entrySet();
38         }
39
40         // Executable main method
41         public static void main(String[] args) {
42                 // Create the phone book
43                 PhoneBook phoneBook = new PhoneBook();
44                 phoneBook.addEntry("Maus, Micky", 4711);
45                 phoneBook.addEntry("Duck, Donald", 1234);
46                 phoneBook.addEntry("Maus, Minni", 1704);
47                 phoneBook.addEntry("Kolumbus, Christoph", 1492);
48                 // Lookup
```

| | **TreeMap** | **HashMap** |
|---|---|---|
| Nachschlagen | $\mathcal{O}(\log n)$ | $\mathcal{O}(1)$ |
| Einfügen | $\mathcal{O}(\log n)$ | $\mathcal{O}(1)$ |
| Entfernen | $\mathcal{O}(\log n)$ | $\mathcal{O}(1)$ |

Tabelle 6.1: Laufzeitvergleich der *TreeMap* und *HashMap* für wichtige Operationen

```
49                  println("Donald's␣number:␣" + phoneBook.lookup("Duck,␣Donald"));
50                  println("1492␣belongs␣to␣" + phoneBook.reverseLookup(1492));
51                  for (Map.Entry<String, Integer> entry
52                       : phoneBook.rangeLookup("Maier", "Meier")) {
53                          println(entry.getKey() + ":␣" + entry.getValue());
54                  }
55          }
56  }
```

Nun fehlen noch die Methoden der Klasse: Im Konstruktor der Telefonbuch-Klasse rufen wir lediglich die Konstruktoren zur Initialisierung der beiden Datenfelder auf. Interessanter ist die Methode `addEntry`, die dem Telefonbuch einen neuen Eintrag hinzufügt. Hier müssen wir beachten, dass wir den Eintrag in beide Abbildungen einfügen, damit dieser sowohl bei der Vorwärts- als auch der Rückwärtssuche gefunden werden kann und unser Telefonbuch somit konsistent bleibt. Das Einfügen in die *TreeMap* benötigt logarithmische Zeit in Bezug zur Anzahl der schon vorhandenen Elemente. Dies wird in der sogenannten Landau-Notation kurz mit $\mathcal{O}(\log n)$ bezeichnet, wobei $n$ für die Anzahl der Elemente in der Abbildung steht und das kaligraphische $\mathcal{O}$ dafür, dass ein Einfügevorgang nicht unbedingt exakt $\log n$ Operationen benötigt, aber zumindest ungefähr (für die Experten: asymptotisch gesehen). Das Einfügen in die *HashMap* benötigt wie schon angesprochen immer gleich lang, d.h. konstante Zeit (kurz $\mathcal{O}(1)$). Die nächsten beiden Operationen des Telefonbuchs dienen dem Nachschlagen von Nummern bzw. Namen. Die Methode `lookup` nimmt einen Namen als Argument und liefert dafür die Telefonnummer zurück. Intern verwendet sie dafür unsere *TreeMap*. Das Nachschlagen benötigt hierbei wiederum logarithmische Zeit. `reverseLookup` bietet die umgekehrte Abfrage und ist ähnlich implementiert. Der einzige Unterschied liegt darin, dass wir hier die *HashMap* verwenden – wodurch das Nachschlagen auch nur konstante Zeit braucht.

Neben den einfachen Punktabfragen wollten wir ja auch eine Bereichssuche unterstützen. Dazu dient die Methode `rangeLookup`, die einen Start- und Endnamen entgegennimmt und damit die Bereichsanfrage auf der *TreeMap* durchführt. Die von ihr aufgerufene Methode `subMap` ist für eine *TreeMap* definiert; bei einer *HashMap* ist sie hingegen nicht verfügbar. Eine *HashMap* könnte die Bereichsanfrage auch gar nicht ausführen, sie unterstützt nur Punktanfragen. Zuletzt verbleibt nur noch die `main` Methode, in der wir unsere Telefonbuchklasse exemplarisch verwenden. Neben dem Einfügen von Einträgen und dem Nachschlagen nach Namen und Nummern, sieht man hier auch wie das Ergebniss einer Bereichsanfrage verwendet werden kann. Dabei iterieren wir mit einer *for each loop* über das Ergebnis der Anfrage vom Typ `Set` und geben für jeden der gefundenen Einträge vom Typ `Map.Entry<String, Integer>` den Namen mit `entry.getKey()` sowie die Telefonnummer mit `entry.getValue()` aus.

In den beiden nachfolgenden Kapiteln 7 und 8 werden wir hinter die Kulissen schauen und herausfinden, welche Datenstrukturen und Algorithmen sich hinter der *TreeMap* und

*HashMap* verbergen. Dabei wird auch deutlich werden, wie es zu den Laufzeiten aus Tabelle 6.1 kommt.

### 6.3.3 Anwendungsbeispiel

Die Schnittstellen und Klassen des Java Collections Frameworks sind im Package *java.util* enthalten und können wie in dem folgenden Beispiel verwendet werden:

```
1  import java.util.Set;
2  import java.util.HashSet;
3
4  public class Student {
5    public Set<Vorlesung> vorlesungen;
6
7    public Student(int matrNr, String name, int semester) {
8      vorlesungen = new HashSet<Vorlesung>();
9    }
10
11   public void belegeVorlesung(Vorlesung vorlesung) {
12     if (!vorlesungen.contains(vorlesung)) {
13       vorlesungen.add(vorlesung);
14       vorlesung.erhoeheAnzahlHoerer();
15     }
16   }
```

Im unserem Beispiel wollen wir für einen Studenten die Menge an Vorlesungen speichern, die er hört. Dabei haben wir uns für ein *Set* entschieden, da wir davon ausgehen, dass ein Student eine Vorlesung nicht mehrfach hören kann. Wir verwenden konkret ein *HashSet*, da wir keine Bereichsanfragen, sondern nur Punktanfragen an die Menge stellen wollen (ansonsten wäre ein *TreeSet* eventuell besser geeignet gewesen). Welche Kollektionstypen sich für den jeweiligen Einsatzbereich am besten eignen, kann man der Java Dokumentation entnehmen.

Zuerst importieren wir die benötigte Schnittstelle und Klasse in Zeile 1 und 2, damit Java die Typen auflösen kann. Anschließend deklarieren wir das Attribut *vorlesungen* in Zeile 5 als *Set* und weisen ihm im Konstruktor in Zeile 8 ein neu erstelltes Objekt der konkreten Implementierung *HashSet* zu. Die Methode *belegeVorlesung()* fügt eine Vorlesung zur Menge hinzu (Zeile 13), wenn der Student sie noch nicht hört (Zeile 12) und erhöht in diesem Fall auch die Anzahl der Hörer einer Vorlesung (Zeile 14).

### 6.3.4 Iteratoren

Die Schnittstelle *Collection* erweitert *Iterable*. Daher müssen alle abgeleiteten Kollektionstypen auch die Methode *iterator()* realisieren. Diese Methode erstellt für eine Kollektion einen Iterator mit dem man alle darin enthaltenen Elemente aufzählen kann. Dies können wir in unserem Beispiel dazu verwenden, um für einen Studenten die Semesterstunden seiner gehörten Vorlesungen zusammen zu zählen:

```
1  public short summeWochenstunden() {
2    short summeSWS = 0;
3    Iterator<Vorlesung> it = vorlesungen.iterator();
4    while (it.hasNext()) {
5      summeSWS += it.next().sws;
6    }
7    return summeSWS;
8  }
```

Zunächst erstellt man sich für eine Kollektion einen Iterator durch den Aufruf der Methode *iterator()*. Anschließend kann man mit der Methode *hasNext()* testen, ob die

Kollektion weitere noch nicht besuchte Elemente enthält und mit *next()* das nächste Element abfragen. Achtung: Verändert man während der Iteration eine Kollektion – indem man Elemente einfügt oder löscht – kann dies zu sehr subtilen Fehlern führen!

## 6.4 Subtypisierung von Generics und Platzhalter

Die Subtypisierung von generischen Typen erfordert eine genauere Betrachtung, die ein – auf den ersten Blick – überraschendes Ergebnis liefern wird. Betrachten wir die Zeilen 1 und 2 des folgenden Programmfragments:

```
1 Stack<Quader> myQuaderStack = new Stack<Quader>(10);
2 Stack<Object> myObjectStack = myQuaderStack;
3
4 myObjectStack.push(new Object());
5 Quader q = myQuaderStack.pop(); // Führt zu ClassCastException!
```

Sind die beiden Zeilen 1 und 2 korrekt? Aufgrund der Überlegung, dass ein Stack von Quadern insbesondere natürlich ein Stack von Objekten ist, werden sich viele Leser dazu verleiten lassen, für die Korrektheit der beiden Zeilen zu plädieren. Die misstrauischen Naturen werden allerdings sagen: „Wenn schon so gefragt wird, wahrscheinlich nicht.“ Und damit haben sie auch Recht. Die Zeilen 4 und 5 zeigen, warum ein Stack von Quadern nicht einem Stack von beliebigen Objekten zuweisbar ist: In Zeile 4 wird auf den *Object*-Stack vollkommen legal ein neues *Object* abgelegt. Da *myObjectStack* und *myQuaderStack* dasselbe Objekt referenzieren, wurde damit allerdings auch ein *Object* auf den *Quader*-Stack abgelegt. Dies führt auch in Zeile 5, in der wir versuchen, das oberste Element von *myQuaderStack* herunterzunehmen, zu einem Laufzeitfehler, weil wir statt des erwarteten Quaders ein *Object* erhalten.

Da wir bereits darauf hingewiesen haben, dass der Compiler bei der Verwendung von Generics die Typsicherheit garantiert, ist es keine Überraschung, dass der Compiler obiges Programmfragment nicht klaglos übersetzt. Vielmehr gibt er während der Übersetzung eine Fehlermeldung (*incompatible types*) bezüglich der Zuweisung in Zeile 2 aus und verhindert den von uns provozierten Laufzeitfehler in Zeile 5.

Entgegen unserer Intuition müssen wir also lernen und verinnerlichen, dass für einen generischen Typ *G* gilt: auch wenn *B* ein Subtyp von *A* ist, ist *G*<*B*> kein Subtyp von *G*<*A*>. Aufgrund des kleinen Beispiels oben sollte es zumindest leichter fallen, diese Tatsache zu akzeptieren.

Wir wollen nun ein Problem mit Generics beleuchten, das aus der eben gelernten Tatsache erwächst. Stellen wir uns vor, wir wollen eine Hilfsmethode schreiben, der wir einen Stack übergeben und die alle Elemente des Stacks der Reihe nach auf dem Bildschirm ausgibt. Ohne Generics ist dies eine einfache Fingerübung für uns:

```
1   public static void printStack(Stack s) {
2     while (!s.isEmpty())
3       System.out.println(s.pop());
4   }
```

Um eine Lösung mit Generics zu finden, sind wir vielleicht versucht, folgendes Programmfragment zu schreiben:

```
1   public static void printStack(Stack<Object> s) {
2     while (!s.isEmpty())
3       System.out.println(s.pop());
4   }
```

Das Problem mit dieser Methode ist, dass wir sie nicht für beliebige Stacks verwenden können, da *Stack<Object>* eben kein Supertyp aller Arten von Stacks ist! Die eben vorgestellte Methode kann also nur Stacks von *Objects* ausgeben. Wie kann man dann eine Methode implementieren, die alle Arten von Stacks ausgeben kann? Wie gibt man den Supertyp aller Stacks an? Java bietet dafür Platzhaltertypen (engl. *wildcard types*) an: *Stack<?>* bezeichnet einen Stack von unbekanntem Typ und dient als Supertyp für Stacks mit beliebigen Typen. Damit können wir folgende korrekte Implementierung erstellen:

```
1    public static void printStack(Stack<?> s) {
2      while (!s.isEmpty())
3        System.out.println(s.pop());
4    }
```

Der Typ der Elemente des Stacks ist zwar unbekannt, die Elemente können aber immer sicher als vom Typ *Object* behandelt werden. Die *pop()*-Methode in Zeile 3 liefert also ein *Object* zurück. Java ruft auf diesem *Object* für uns automatisch die Methode *toString()* auf, die jedes Objekt implementiert, um eine Stringrepräsentation von sich selbst auszugeben. Ein *String*-Objekt gibt sich bei einem Aufruf von *toString()* einfach selbst zurück. Nun können wir die Methode beispielsweise mit einem *String*-Stack aufrufen:

```
1    Stack<String> s = new Stack<String>(10);
2    s.push("gehts!");
3    s.push("So");
4    printStack(s);
```

Das Auslesen von Elementen aus einer Kollektion mit unbekanntem Typ ist also sicher, wie wir gesehen haben, das Hinzufügen von Elementen allerdings nicht, da der Elementtyp unbekannt ist. Das folgende Programmfragment ist also nicht zulässig:

```
1    Stack<?> s = new Stack<String>(10);
2    s.push(new Object()); // Fehler beim Übersetzen!
```

Die Methode *push(e)* in Zeile 2 erwartet ein Objekt des Typs *T* oder eines Subtyps von *T*. Da *T* in diesem Fall *?*, also unbekannt ist, kann keine Methode von *Stack* aufgerufen werden, die einen Parameter vom Typ *T* erwartet und damit auch nicht *push(e)*.[1] Wird ein Objekt also durch eine Referenz *ref* vom Typ eines Platzhaltertypen, beispielsweise *Stack<?>*, referenziert, ist die Menge der aufrufbaren Methoden auf dem durch *ref* referenzierten Objekt eingeschränkt: alle Methoden des Objekts, die mindestens einen Parameter vom Typ *T* erwarten, sind nicht aufrufbar. Alle Methoden, die keine Parameter vom Typ *T* erwarten, bleiben verwendbar, auch die Methoden, die einen Rückgabewert vom Typ *T* liefern.

Java bietet nicht nur den eben vorgestellten (uneingeschränkten) Platzhaltertyp *?* an, sondern auch eingeschränkte Platzhalter (engl. *bounded wildcards*). Wir wollen an einem kleinen Beispiel die Verwendung dieser Art von Platzhaltern verdeutlichen. Wir erinnern uns an die Klassenhierarchie *Zylinder*, *Rohr* und *KonischesRohr*. Nehmen wir an, wir wollen Objekte dieser Typen in einer Liste verwalten. Außerdem wollen wir eine Methode implementieren, der eine Liste von derartigen Objekten übergeben werden kann und die das Volumen aller Objekte aufsummiert. Java bietet bereits einen generischen Typen für Listen an: *List*. In einer *List<Zylinder>* können also alle drei Typen wegen der Substituierbarkeit verwaltet werden. Die gewünschte Methode sieht also beispielsweise so aus:

```
1    public static double sumVolume(List<Zylinder> list) {
```

[1] Die einzige Ausnahme von dieser Regel ist *null*, da eine *null*-Referenz zu jedem Typ kompatibel ist.

```
2       double sum = 0;
3       for (Zylinder z : list) {
4         sum += z.volumen();
5       }
6       return sum;
7     }
```

In den Zeilen 3–5 verarbeiten wir der Reihe nach alle Elemente, die in der Liste gespeichert sind. Auf die verwendete Syntax kommt es hier nicht an, wir werden später darauf zurückkommen. Um was es uns eigentlich geht, ist die Methodendeklaration von *sumVolume(list)*. Auf den ersten Blick sieht unsere Lösung korrekt aus: wir können eine *List<Zylinder>* übergeben und eine solche Liste kann *Zylinder*-, *Rohr*- und *Konisches-Rohr*-Objekte verwalten. So weit, so gut. Wenn wir jetzt aber eine *List<Rohr>* vorliegen haben, können wir die Methode nicht aufrufen. Der Methodenrumpf könnte eine derartige Liste problemlos bearbeiten, aber Java wird einen Typfehler melden. Für solche Fälle unterstützt Java *bounded wildcards*, die wir bei unserer Methodendeklaration einsetzen können:

```
1     public double sumVolume(List<? extends Zylinder> list) {
2       ...
3     }
```

Durch diesen eingeschränkten Platzhaltertyp haben wir ausgedrückt, dass die Methode einen Parameter vom Typ „Liste einer beliebigen Art von Zylindern“ erwartet. Nun können wir der Methode Listen vom Typ *List<Zylinder>*, *List<Rohr>* oder allgemein *List<S>*, wobei *S* für eine beliebige Subklasse von *Zylinder* steht, übergeben und haben damit tatsächlich die Methode vorliegen, die wir uns gewünscht haben. Auch bei der Verwendung von eingeschränkten Platzhaltern gibt es Einschränkungen bezüglich der Methoden, die noch aufgerufen werden können. Innerhalb von *sumVolume* ist nur bekannt, dass es sich bei *list* um eine Liste von einem unbekannten Subtyp von *Zylinder* handelt. Damit dürfen wir wieder keine Methoden auf *list* aufrufen, die einen Parameter mit diesem generischen Typ erwarten, können also der Liste beispielsweise keine neuen Elemente hinzufügen. Der generische Typ als Rückgabetyp ist dagegen wieder unproblematisch, da der Compiler weiß, dass es sich zumindest immer um einen *Zylinder* oder einen Subtyp davon handelt.

Java unterstützt auch eingeschränkte Platzhalter der Form *<? super Zylinder>*. Dieses Beispiel steht für einen unbekannten Supertyp von *Zylinder* und bildet das Gegenstück zu *<? extends Zylinder>*. Da dieser Platzhalter recht selten verwendet wird, wollen wir nicht näher darauf eingehen und lediglich auf Übungsaufgabe 6.5 verweisen.

## 6.5 Wrapper für Sorten

Eine Einschränkung von generischen Typen besteht darin, dass nur Klassen für Typparameter eingesetzt werden können, keine Sorten (also z.B. *int* oder *float*). Aber auch bei der Verwendung von *Object*-Variablen für die Implementierung von Kollektionen bestand dieses Problem. Java bietet deshalb von jeher so genannte Wrapperklassen für alle Sorten an, um diese in Objekten abspeichern zu können, z.B. *Integer* für *int* und *Float* für *float*. Eingepackt in diese Wrapper kann man beispielsweise auch *int*-Werte in einem Stack verwalten:

```
1  Stack<Integer> myIntegerStack = new Stack<Integer>(10);
2  int fortyseveneleven = 4711;
```

```
3 myIntegerStack.push(new Integer(fortyseveneleven)); // Einpacken für Weihnachten
4 Integer surprise = myIntegerStack.pop();            // Immer noch eingepackt
5 int unwrapped = surprise.intValue();                // Auspacken :)
```

Das manuelle „Verpacken“ der *int*-Werte in *Integer*-Objekte und das anschließende „Auspacken“ führen dazu, dass der Code aufgebläht und sehr unübersichtlich wird. Deshalb wurde in Java 1.5 ein Automatismus für diese Arbeit integriert, der *Autoboxing/-unboxing* genannt wird. Damit kümmert sich der Compiler automatisch darum, im Bedarfsfall Sorten in die entsprechenden Wrapper-Objekte zu verpacken bzw. die Sorten aus den Wrappern auszupacken. Dadurch hat der Programmierer weniger Arbeit und der Code ist übersichtlicher:

```
1 Stack<Integer> myIntegerStack = new Stack<Integer>(10);
2 int fortyseveneleven = 4711;
3 myIntegerStack.push(fortyseveneleven); // Automatisches Verpacken
4 int myInt = myIntegerStack.pop();      // Automatisches Auspacken
```

Durch Autoboxing/-unboxing kann der Programmierer den Unterschied zwischen Sorten und Klassen weitestgehend ignorieren. Allerdings wollen wir an dieser Stelle zwei Hinweise geben:

1. Ein *Integer* mit dem Wert *null* führt beim Autounboxing zu einer *NullPointerException*.

2. Das Autoboxing/-unboxing bringt Laufzeiteinbußen mit sich, da immer neue Objekte angelegt werden, um Werte darin abzuspeichern, bzw. Methoden aufgerufen werden, um Werte aus Objekten auszulesen. Auch der Speicherverbrauch eines Programms kann dadurch steigen. Auch wenn dem Programmierer die Verwendung von Sorten also stark vereinfacht wird und damit sogar die Trennung zwischen Sorten und Klassen etwas verwischt wird, sollte einem guten Programmierer dies stets bewusst sein!

## 6.6 Generische Methoden

Java unterstützt nicht nur generische Typen, sondern auch *generische Methoden*. Als kleines Beispiel wollen wir eine Methode implementieren, die ein Array erhält und alle im Array enthaltenen Elemente auf einem ebenfalls übergebenen Stack ablegt. Ein erster Versuch könnte so aussehen:

```
1   public static void array2Stack(Object[] array, Stack<?> stack) {
2     for (int i = 0; i < array.length; i++)
3       stack.push(array[i]); // Fehler während der Übersetzung!
4   }
```

Da wir uns an die Subtypisierung von generischen Typen erinnert haben, haben wir gleich vermieden, einen *Stack<Object>* zu verwenden, was die Einsetzbarkeit unserer Methode über die Maßen eingeschränkt hätte. In den Zeilen 2–3 werden die Elemente des Arrays nacheinander ausgelesen und auf dem Stack abgelegt. Dummerweise moniert der Compiler Zeile 3, weil wir versuchen, in einen Stack von unbekanntem Typ Elemente einzufügen. Genau für solche Fälle gibt es generische Methoden. Genau wie bei generischen Typen haben generische Methoden einen oder mehrere Typparameter. In Java sieht das folgendermaßen aus:

```
1    public static <T> void array2Stack(T[] array, Stack<T> stack) {
2      for (int i = 0; i < array.length; i++)
3        stack.push(array[i]);
4    }
```

Die Methode *array2Stack(array, stack)* erwartet also ein Array vom Typ *T* und einen Stack vom Typ *Stack<T>* als Parameter. In der Methode werden dann alle Elemente (die natürlich Typ *T* haben) aus dem Array ausgelesen und auf dem Stack abgelegt. Das Schöne ist, dass wir beim Aufruf einer generischen Methode keinen Typparameter angeben müssen, weil sich den der Java-Compiler selbst herleitet:

```
1    String[] stringarray = {"0", "1", "2", "3"};
2    Stack<String> stringstack = new Stack<String>(4);
3    Stack<Object> objectstack = new Stack<Object>(4);
4
5    array2Stack(stringarray, stringstack);
6    array2Stack(stringarray, objectstack);
```

In Zeile 1 legen wir ein Stringarray an, das in Stacks kopiert werden soll. In den Zeilen 2 und 3 legen wir einen Stack vom Typ *String* und einen Stack vom Typ *Object* an. Dann rufen wir in Zeile 5 die Methode *array2Stack* auf. Da es sich um ein *String*-Array und um einen *String*-Stack handelt, leitet der Compiler für den Typparameter *T* den Typ *String* her. In Zeile 6 kopieren wir das *String*-Array in einen *Object*-Stack. Der Compiler wird für *T* also *Object* herleiten. Der Aufruf klappt, weil Java es zulässt, dass ein *String[]* als *Object[]* behandelt wird.

## 6.7 Implementierung von Java Generics: Type Erasure

Bei der Arbeit mit generischen Typen und Methoden werden wir immer wieder auf – auf den ersten Blick seltsam erscheinende – Einschränkungen und Probleme treffen. In diesem Abschnitt wollen wir kurz erläutern, wie Generics vom Java-Compiler umgesetzt werden und welche Beschränkungen daraus entstehen.

Ein generischer Typ wird vom Java-Compiler einmal in Bytecode übersetzt und als Datei abgespeichert – wie jede andere Klasse auch. Es wird also nicht für jede Belegung des Typparameters eine eigene Klasse erzeugt. Java nutzt dazu eine Technik namens *type erasure*, um einen generischen Typ auf einen nicht-generischen Typ abzubilden. Im Prinzip kann man sich den Vorgang so vorstellen, dass der Java-Compiler aus einem generischen Typ einen nicht-generischen Typ generiert, indem er Typparameter entfernt bzw. durch einen konkreten Typ ersetzt und Casts einfügt. Wir wollen uns dies am Beispiel des generischen Stacks einmal ansehen:

```
public class Stack<T> {
   private T[] elements;
   private int cardinality = 0;

   public Stack(int capacity) {
      elements = (T[])new Object[capacity];
   }
   public void push(T e) {
      elements[cardinality++] = e;
   }
   public T pop() {
      return elements[--cardinality];
   }
   public boolean isEmpty() {
      return (cardinality == 0);
   }
}
```

```
1  public class Stack {
2    private Object[] elements;
3    private int cardinality = 0;
4
5    public Stack(int capacity) {
6      elements = (Object[])
7        new Object[capacity]; }
8    public void push(Object e) {
9      elements[cardinality++] = e;
10   }
11   public Object pop() {
12     return elements[--cardinality];
13   }
14   public boolean isEmpty() {
15     return (cardinality == 0);
16   }
17 }
```

Der generische Stack auf der linken Seite wird also in die nicht-generische Form auf der rechten Seite überführt. Diese nicht-generische Form entspricht weitestgehend dem Stack, den wir am Anfang des Kapitels implementiert haben, als wir generische Typen noch nicht eingeführt hatten. Auch der Code zur Verwendung eines generischen Stacks wird entsprechend umgeformt:

```
Stack<String> stack = new Stack<String>(4);
stack.push("Test");
String s = stack.pop();
```

```
1  Stack stack = new Stack(4);
2  stack.push("Test");
3  String s = (String)stack.pop();
```

Java nutzt das Wissen über generische Typen also nur zur Übersetzungszeit zur Typprüfung. Zur Laufzeit sind die Informationen über generische Typen nicht mehr verfügbar. Dies führt zu folgenden Besonderheiten bei der Verwendung von generischen Typen:

- Generische Typen können auch immer ohne Typparameter verwendet werden. In Java heißen diese Typen *raw types*. Wir können also beispielsweise einen *Stack* anlegen, nicht nur einen *Stack<String>*. Dadurch ist auch die Kompatibilität und Zusammenarbeit von Code mit generischen Typen und älterem Code, in dem noch kein Gebrauch von generischen Typen gemacht wurde, sichergestellt.

- Arrays dürfen als Elementtyp keinen generischen Typ und keinen Typparameter haben. Aus diesem Grund mussten wir bei der Implementierung des Stacks auch ein *Object[]* anlegen und nach *T[]* casten. Aber auch ein Array mit Elementen beispielsweise vom Typ *Stack<String>* ist nicht erlaubt. Die einzige Ausnahme ist die Wildcard-Instanziierung des generischen Typs, also beispielsweise ein Array mit Elementen vom Typ *Stack<?>*.

- Generische Exceptions sind nicht erlaubt.

- Casts auf generische Typen, wie z.B. *(T[])* in der Implementierung unseres Stacks, liefern immer eine *unchecked cast* Warnung des Compilers, der uns damit mitteilen will, dass Java diesen Cast zur Laufzeit nicht überprüfen wird. Immer, wenn der Compiler mindestens eine *unchecked cast* oder eine andere *unchecked* Warnung ausgibt, kann er die Typsicherheit eines Programms nicht mehr garantieren und Typfehler zur Laufzeit können auftreten. Es liegt in der Verantwortung des Programmierers, sämtliche von Java gemeldeten Warnungen zu überprüfen. In unserer

Stack-Implementierung ist beispielsweise der Cast nach *T[]* harmlos (aber unverzichtbar), wie man in der nicht-generischen Version leicht erkennen kann.

- Zur Laufzeit ist es nicht möglich, festzustellen, von welchem Typ ein Objekt, das eine Instanz eines generischen Typs ist, genau ist. Man kann zwar mit

  ```
  list instanceof Stack
  ```

  feststellen, ob das Objekt, auf das die Variable *list* verweist, vom Typ *Stack* ist, der Test

  ```
  list instanceof Stack<String>
  ```

  ist allerdings verboten.

Zuletzt noch ein kleiner Appell: Trotz der Einschränkungen von Generics sollten sie eingesetzt werden, wo immer es geht, da die bessere Lesbarkeit des Codes und die erhöhte Typsicherheit den höheren Entwicklungsaufwand mehr als rechtfertigen.

## 6.8 Übungen

**6.1** Wir beziehen uns hier auf die generische Methode aus Abschnitt 6.6:

```
1  public static <T> void array2Stack(T[] array, Stack<T> stack) {
2    ...
3  }
```

Welche der nachfolgenden Aufrufe von *array2Stack* wäre gültig und welchen Typ würde der Compiler jeweils für *T* herleiten?

```
1  // Anlegen der Arrays;
2  Object[]       objectarray   = { new Object(), new Object() };
3  String[]       stringarray   = { "0", "1", "2", "3" };
4  GeoPrimitive[] geoprimarray  = { ... };
5  Zylinder[]     zylinderarray = { ... };
6  Polyeder[]     polyederarray = { ... };
7
8  // Stacks anlegen
9  Stack<Object>       objectstack  = new Stack<Object>(100);
10 Stack<String>       stringstack  = new Stack<String>(100);
11 Stack<GeoPrimitive> geoprimstack = new Stack<GeoPrimitive>(100);
12
13 // Aufrufe von array2String
14 array2Stack(objectarray,   objectstack);
15 array2Stack(objectarray,   stringstack);
16 array2Stack(objectarray,   geoprimstack);
17 array2Stack(stringarray,   objectstack);
18 array2Stack(stringarray,   stringstack);
19 array2Stack(stringarray,   geoprimstack);
20 array2Stack(zylinderarray, geoprimstack);
21 array2Stack(polyederarray, geoprimstack);
22 array2Stack(geoprimarray,  objectstack);
23 array2Stack(geoprimarray,  stringstack);
24 array2Stack(geoprimarray,  geoprimstack);
```

**6.2** Zur Begründung, warum Arrays keinen generischen Elementtyp haben dürfen, dient das folgendes Codefragment (das natürlich so nicht übersetzt werden kann!):

```
1 List<String>[] lsa = new List<String>[10];
2 Object o = lsa;
3 Object[] oa = (Object[]) o;
4 List<Integer> li = new ArrayList<Integer>();
```

```
5 li.add(new Integer(3));
6 oa[1] = li;
7 String s = lsa[1].get(0);
```

Erklären Sie, welcher Fehler auftreten würde, wenn der Compiler das generische Array *lsa* in Zeile 1 zulassen würde.

**6.3** Implementieren Sie die Methode *pushAll* der generischen Klasse *Stack<T>*, die eine Liste als Parameter erhält und alle Elemente der Liste nacheinander auf den Stack legt. Können Sie mit Ihrer Implementierung eine *List<String>* in einen *Stack<Object>* einfügen? Sollte das möglich sein?

**6.4** Implementieren Sie eine möglichst flexible generische Methode, die zwei Stacks als Parameter erhält und alle Elemente des einen Stacks ausliest und auf dem anderen Stack ablegt. Welche Beziehung muss zwischen dem Typen des Ursprungsstacks und dem des Zielstacks bestehen?

**6.5** Implementieren Sie die Methode aus Aufgabe 6.4 mit folgender Änderung: Das letzte Element, das von dem einen auf den anderen Stack gelegt wird, soll als Ergebnis der Methode zurückgegeben werden.
Hinweis: Denken Sie an eingeschränkte Platzhalter der Form *<? super T>* und achten Sie auf Ihren Rückgabetyp.

## 6.9 Annotated Bibliography

Type parameters in the context of database languages are surveyed by Cardelli Cardelli (1988). Polymorphic operations are analyzed by Cardelli and Wegner Cardelli und Wegner (1985). They introduced the term *bounded polymorphism*. Also, Danforth and and Tomlinson Danforth und Tomlinson (1988) survey polymorphic operation concepts of various languages. The programming language ML—described by Milner Milner (1978)—incorporates very general polymorphic operations. In ML the type consistency is verified by static type derivation, rather than type constraints specified by the user. A similar polymorphic operation capability forms the basis of the data model Machiavelli, designed by Ohori, Buneman, and Breazu-Tannen Ohori, Buneman und Breazu-Tannen (1989).

B. Meyer Meyer (1986) was one of the first authors to point out that inheritance and genericity are two dual concepts—none of which can be satisfactorily modeled by the other. Liskov and Guttag Liskov und Guttag (1986) wrote an excellent text on data modeling using generic type facilities. Their book is based on the type concepts of the Clu language Liskov et al. (1981). The programming language Ada Institut (1983) provides a limited generic type concept.

# 7. Suchbäume

## 7.1 Binäre Suchbäume

- linker Teilbaum enthält kleinere Elemente
- rechter Teilbaum enthält größere Elmente
- Inorder-Durchlauf = Sortierung
- Degenerierung zur linearen Liste, wenn die Elemente sortiert (oder umgekehrt sortiert) eingegeben werden
- Degenerierter Suchbaum hat O(N)-Komplexität für
  - Suchen
  - Einfügen
  - Löschen

## 7.2 Beispiel

Abbildung 7.1 zeigt beispielhaft, wie ein binärer Suchbäume durch das Einfügen von Elementen erzeugt wird. Als erstes fügen wir in einen leeren Baum das Element 42 ein, welches dadurch zur Wurzel wird. Als nächstes fügen wir die 37 hinzu. Da die 37 kleiner ist als die Wurzel 42 wird sie zu derem linken Kind. Die als nächstes eingefügte 47 landet entsprechend rechts von der 42. Jetzt soll die 7 eingefügt werden; da die 42 aber schon ein linkes Kind hat, wird die 7 nun auch mit diesem verglichen. Die 7 ist kleiner als die 37 und wird dementsprechend links eingefügt. Wir wiederholen das Einfügen mit der 40, 67 und 17.

## 7.3 AVL-Bäume: Balancierte binäre Suchbäume

Die Höhe eines Baums mit der Wurzel $V$ ist definiert als $h(v) = 1 + \max(h(T_l), h(T_r))$, wobei $T_l$ und $T_r$ das linke bzw. das rechte Kind (bzw. die Teilbäume) von $v$ sind. Die Höhe eines leeren Teilbaums ist 0.

Ein binärer Suchbaum erfüllt die AVL-Eigenschaften, wenn für jeden Knten *v* des Baums gilt:

$$|h_l - h_r| \leq 1$$

Der AVL-Baum ist benannt nach Adel'son-Vel'skii und Landis.

Den Wert $h_l - h_r$ nennt man den Balance-Faktor des Knotens. Gültige Werte für diesen Balance-Faktor sind $-1, 0, 1$. Wenn beim Einfügen oder Löschen eine "Unbalanciertheit" auftritt, muß diese durch entsprechende Transformationen revidiert werden.

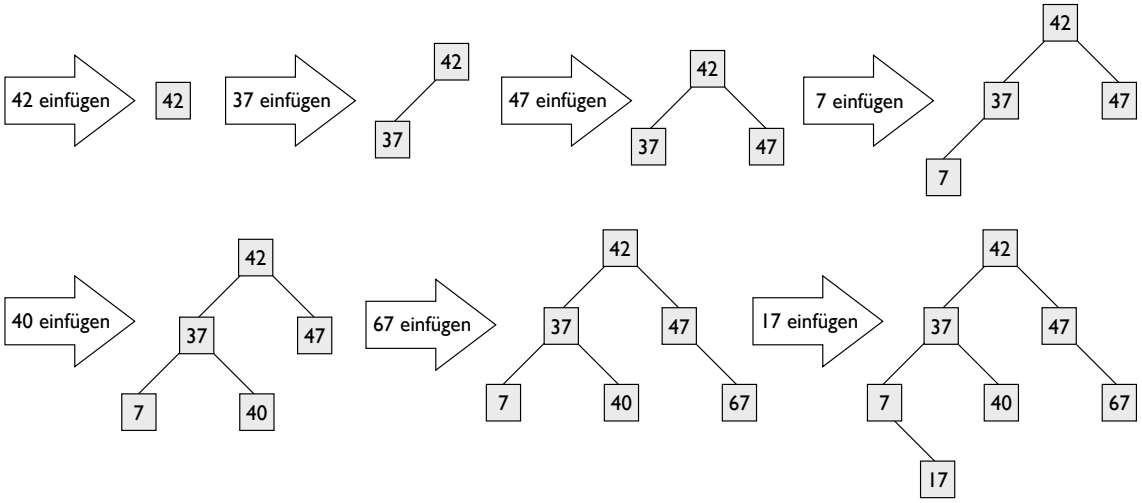



Abbildung 7.1: Beispiel für einen binären Suchbaum

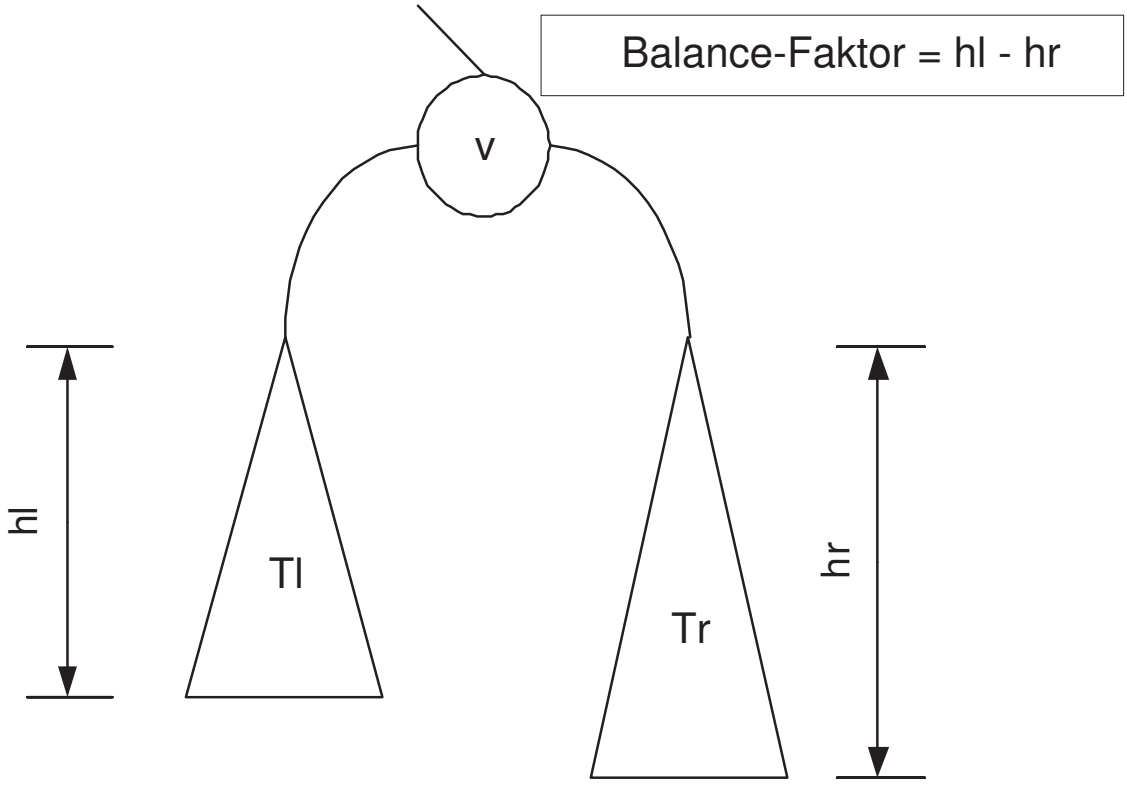



Abbildung 7.2: Der Balance-Faktor des Knotens $v$

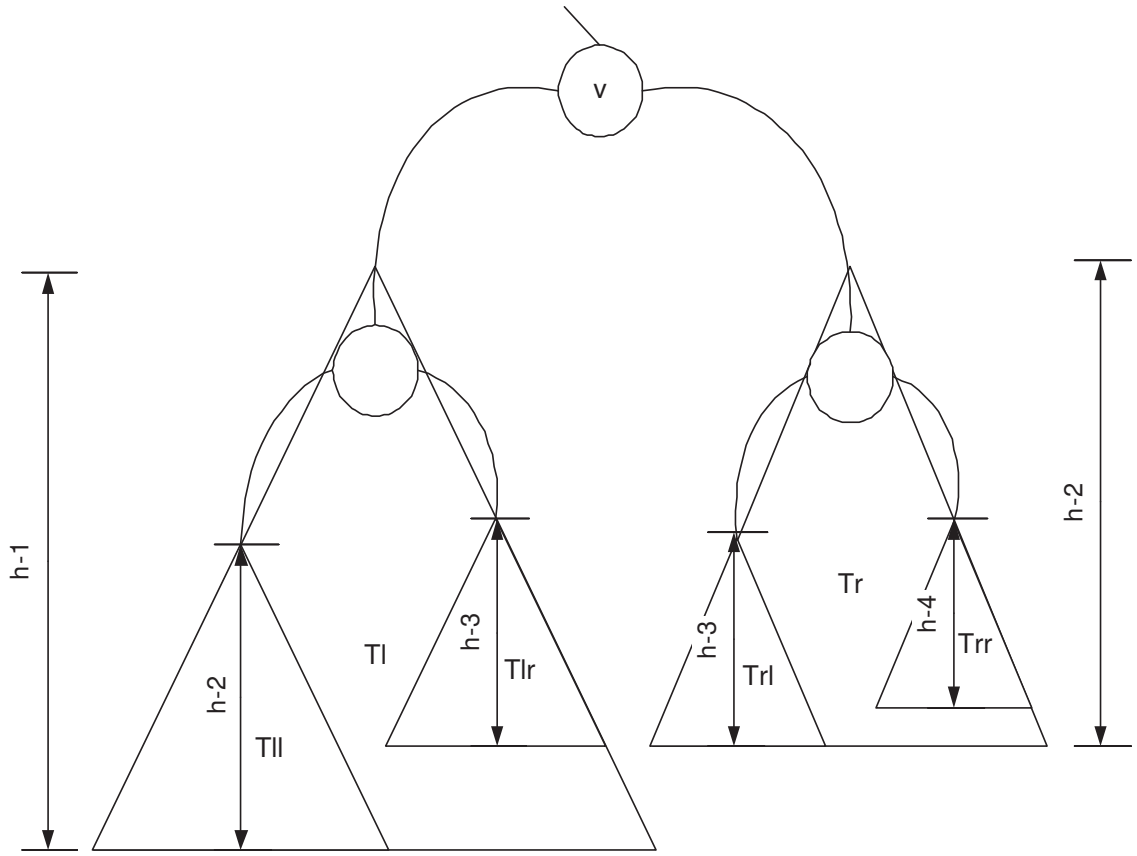



Abbildung 7.3: Die Balancierung von AVL Bäumen

### 7.3.1 Höhenvergleich eines AVL-Baums mit den Extremfällen: vollständiger Suchbaum und degenerierter Suchbaum

Wir wollen nachweisen, dass ein AVL-Baum mit *N* Einträgen maximal die Höhe $2log_2N$ hat. Zu diesem Zweck überlegen wir uns, wieviele Einträge ein gültiger AVL-Baum der Höhe *h* mindestens haben muss, damit jeder Knoten einen gültigen Balance-Faktor hat. Wenn wir mit $n(h)$ die minimale Anzahl der Knoten eines AVL-Baums der Höhe $h$ bezeichnen, so gilt für die "nidrigen" Bäum folgendes:

$$
\begin{aligned}
n(1) &= 1 && (7.1)\\
n(2) &= 2 && (7.2)\\
n(3) &= 4 && (7.3)\\
n(4) &= 7 && (7.4)\\
n(5) &= 12 && (7.5)\\
&\ldots && (7.6)\\
n(h) &= 1 + n(h-1) + n(h-2) && (7.7)
\end{aligned}
$$

Da $n(h-1)$ offensichtlich größer ist als $n(h-2)$ folgt folgende Ungleichung:

$$
\begin{aligned}
n(h) &= 1 + n(h-1) + n(h-2) && (7.8)\\
n(h) &\geq 2 * n(h-2) && (7.9)\\
n(h) &\geq 2 * 2 * n(h-4) && (7.10)\\
n(h) &\geq 2 * 2 * 2 * n(h-6) && (7.11)\\
&\ldots && (7.12)\\
n(h) &\geq 2^i * n(h-2i) \quad \text{für alle } i \text{ mit } n-2i \geq 1 && (7.13)\\
& && (7.14)
\end{aligned}
$$

Jetzt setzen wir für $i$ den Wert $\lceil h/2 \rceil - 1$ ein und erhalten:

$$
\begin{aligned}
n(h) &\geq 2^{\lceil h/2 \rceil - 1} * n(h - 2(\lceil h/2 \rceil - 1)) && (7.15)\\
n(h) &\geq 2^{\lceil h/2 \rceil - 1} * n(1) && (7.16)\\
n(h) &\geq 2^{\lceil h/2 \rceil - 1} && (7.17)\\
& && (7.18)
\end{aligned}
$$

Hieraus folgt für die Höhe $h$ eines AVL-Baums:

$$log_2 n(h) > h/2 - 1$$

$$h < 2log_2 n(h) + 2$$

Also hat ein AVL-Baum mit $n$ Einträgen höchstens die Höhe $2log_2 n + 2$. Das heißt, ein AVL-Baum ist auf keinen Fall mehr als "gut" doppelt so hoch wie ein vollständiger binärer Suchbaum, der einen optimalen Suchbaum darstellt. Dieser Vergleich ist in Abbildung 7.4 skizziert. Man beachte, dass ein normaler (unbalancierter) binärer Suchbaum zur linearen Liste degenerieren kann und somit im schlechtesten Fall die Höhe $n$ hat.

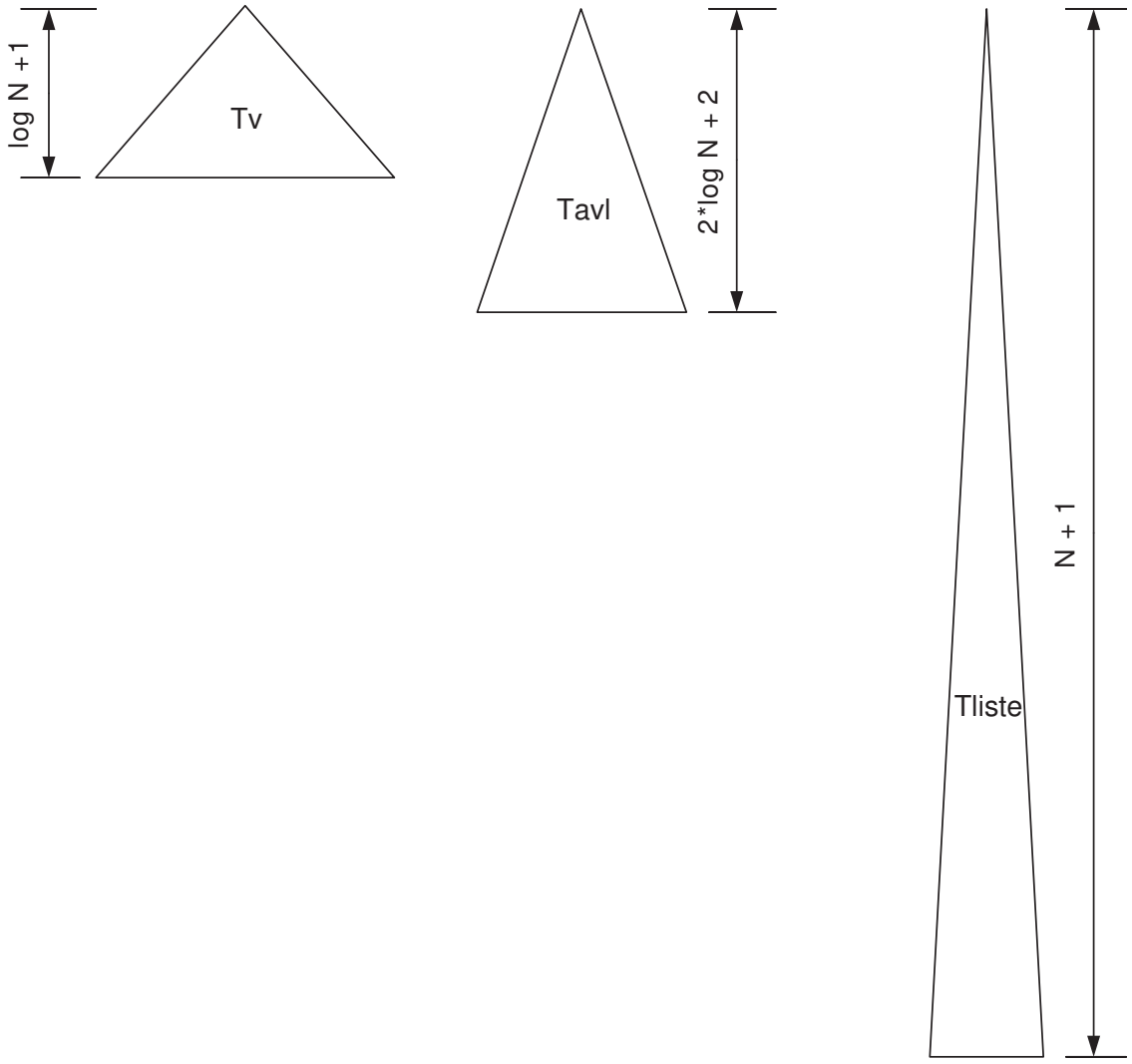



Abbildung 7.4: Höhenvergleich: vollständiger Suchbaum, AVL-Baum, degenerierter Suchbaum

### 7.3.2 Die Transformationen zur Höhenbalancierung

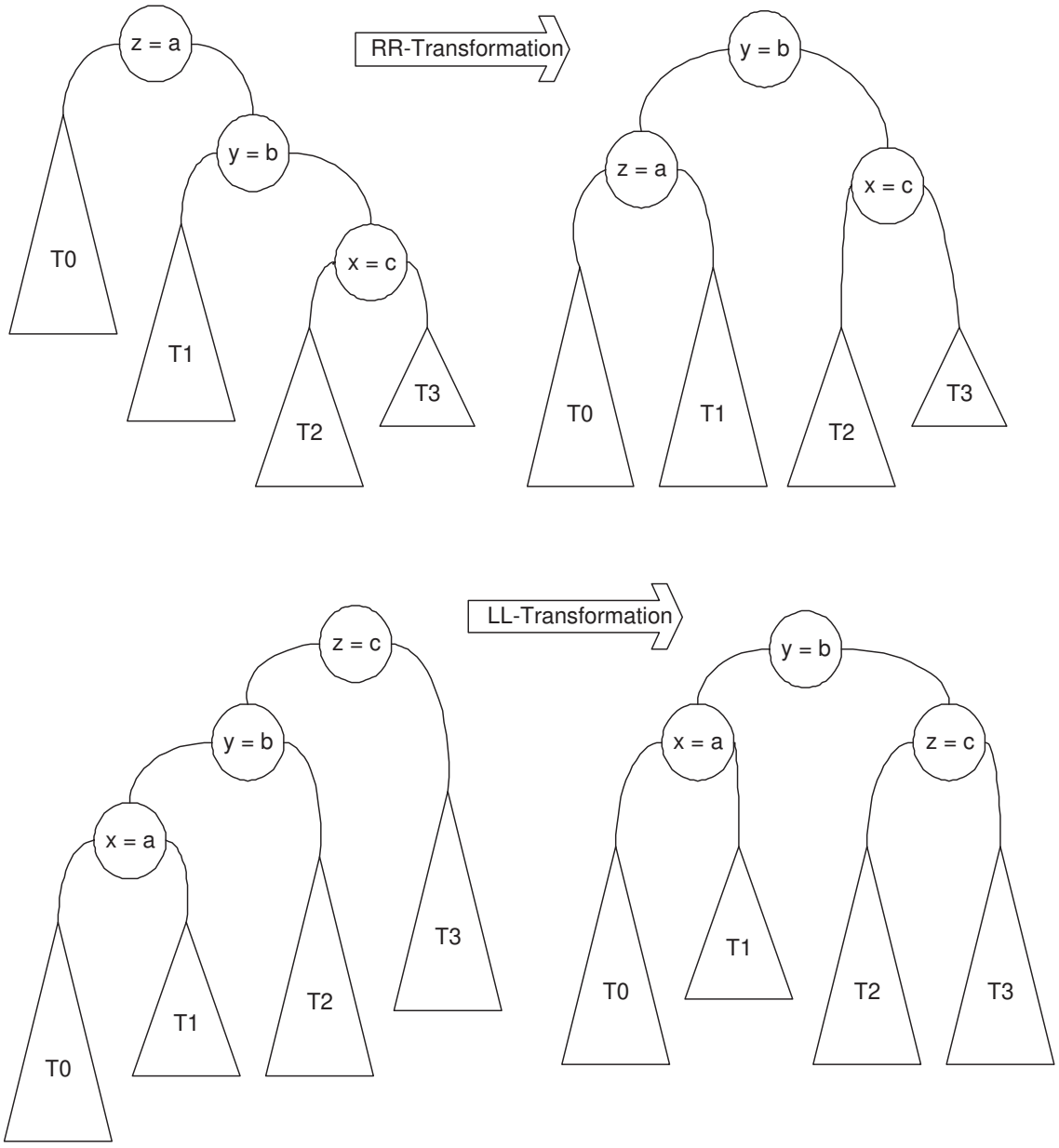



Abbildung 7.5: Die einfachen Rotationen LL und RR

### 7.3.3 Der sukzessive Aufbau eines AVL-Baums

## 7.4 B-Bäume

Normale Binärbäume wurden als Suchstruktur für den Hauptspeicher konzipiert. Sie eignen sich nicht als Speicherstruktur für den Hintergrundspeicher, da sie sich nicht effektiv auf Seiten abbilden lassen. Man verwendet daher für die Hintergrundspeicherung Mehrwegbäume, deren Knotengrößen auf die Seitenkapazitäten abgestimmt werden. Ein Knoten des Baums entspricht einer Seite des Hintergrundspeichers.

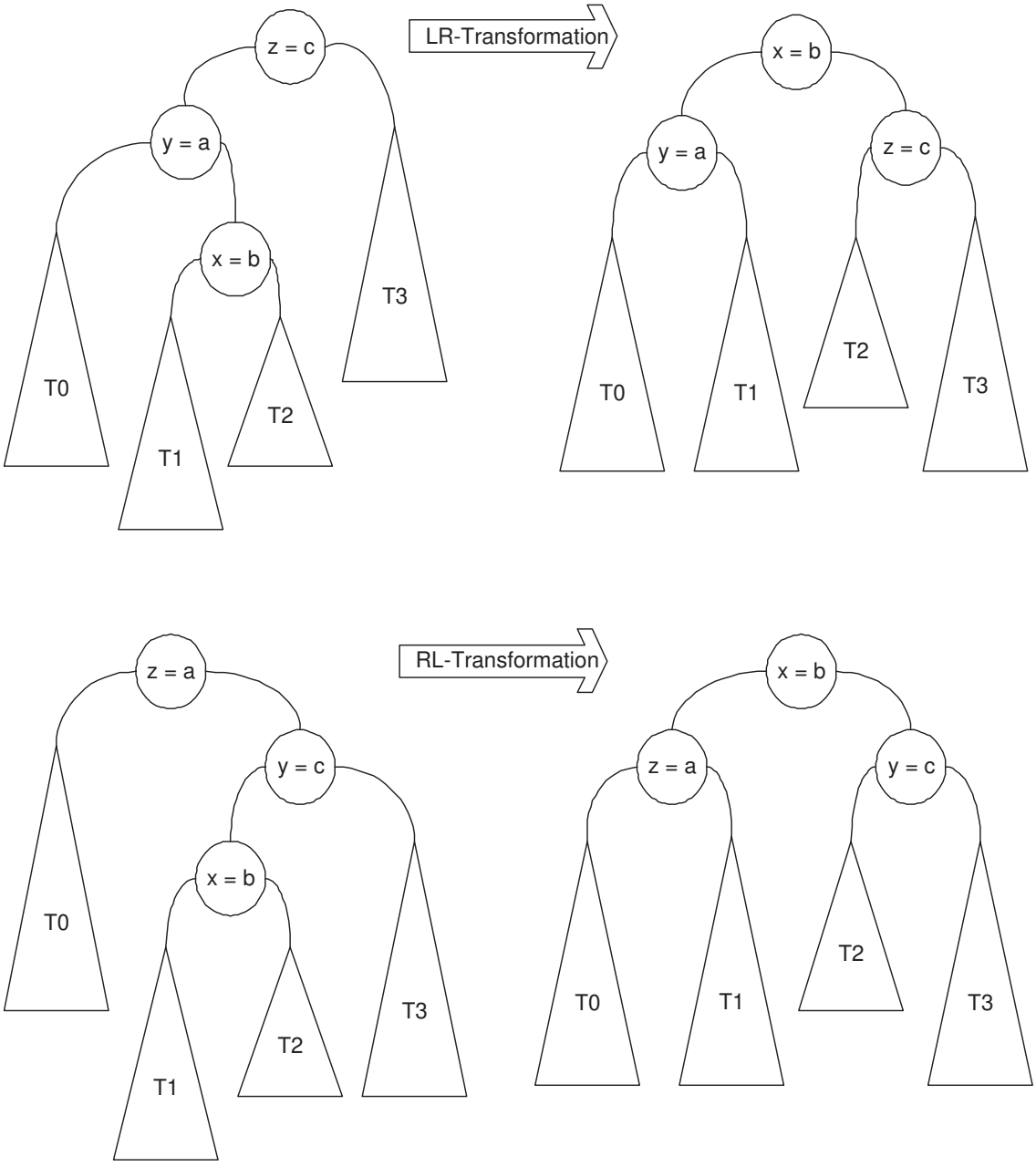



Abbildung 7.6: Die schwierigeren (doppelten) Rotationen LR und RL

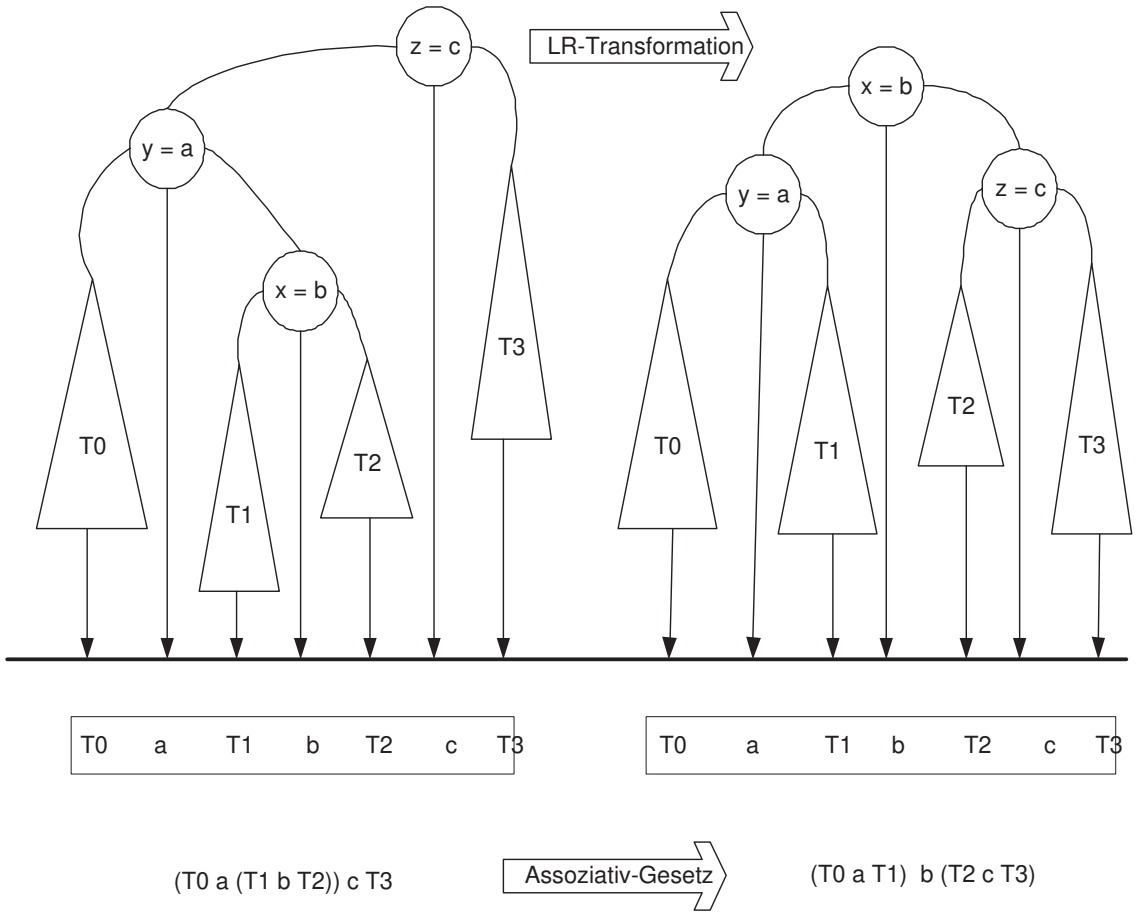



Abbildung 7.7: Die Invarianz des Inorder-Durchlaufs und die Transformation als Anwendung des Assoziativgestzes

**Einfügen: 18, 28, 40, 20, 4, 2, 9, 6, 12, 7, 22, 32, 47, 30, 35,**

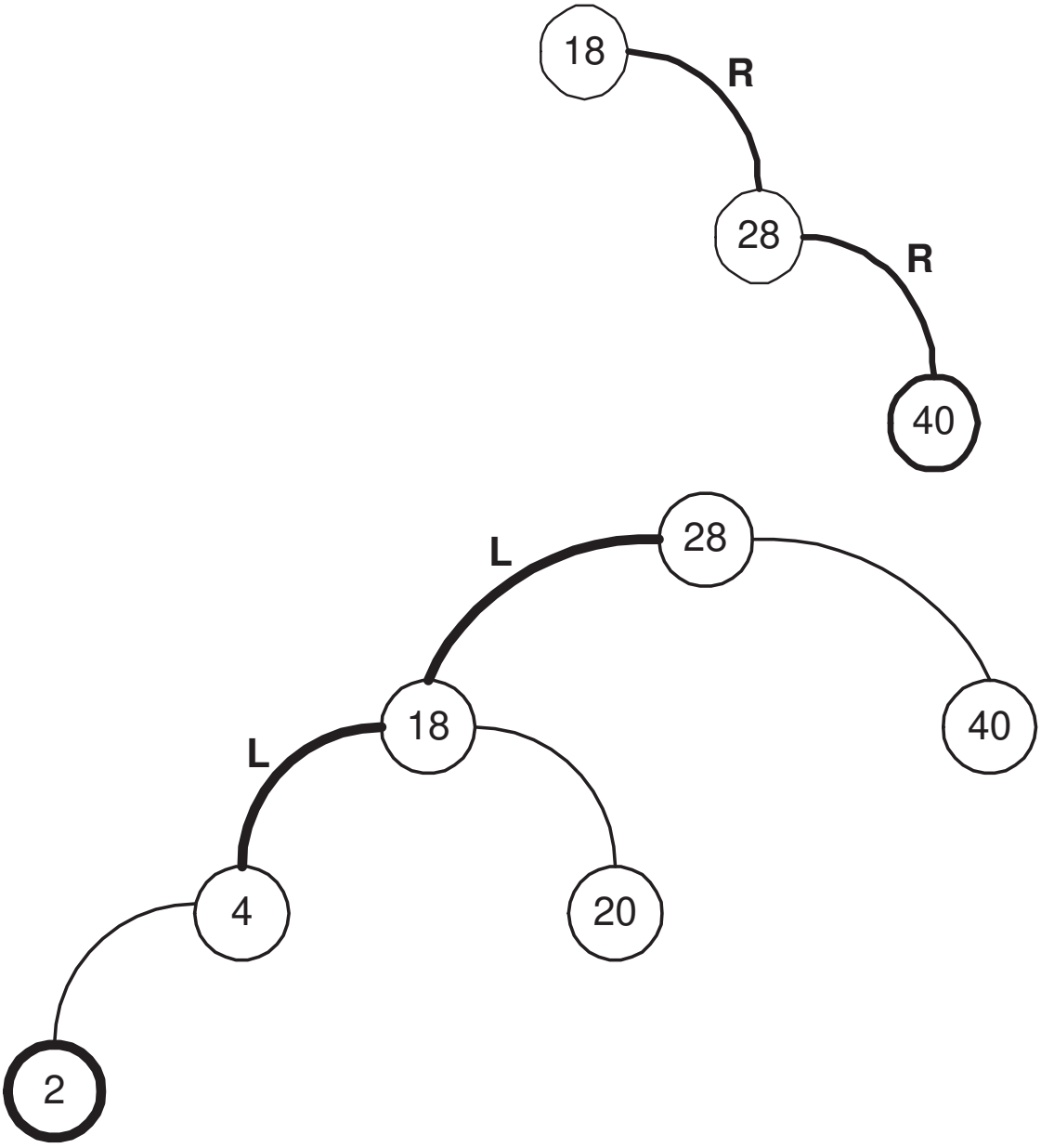



Abbildung 7.8: Einfügen in den AVL-Baum

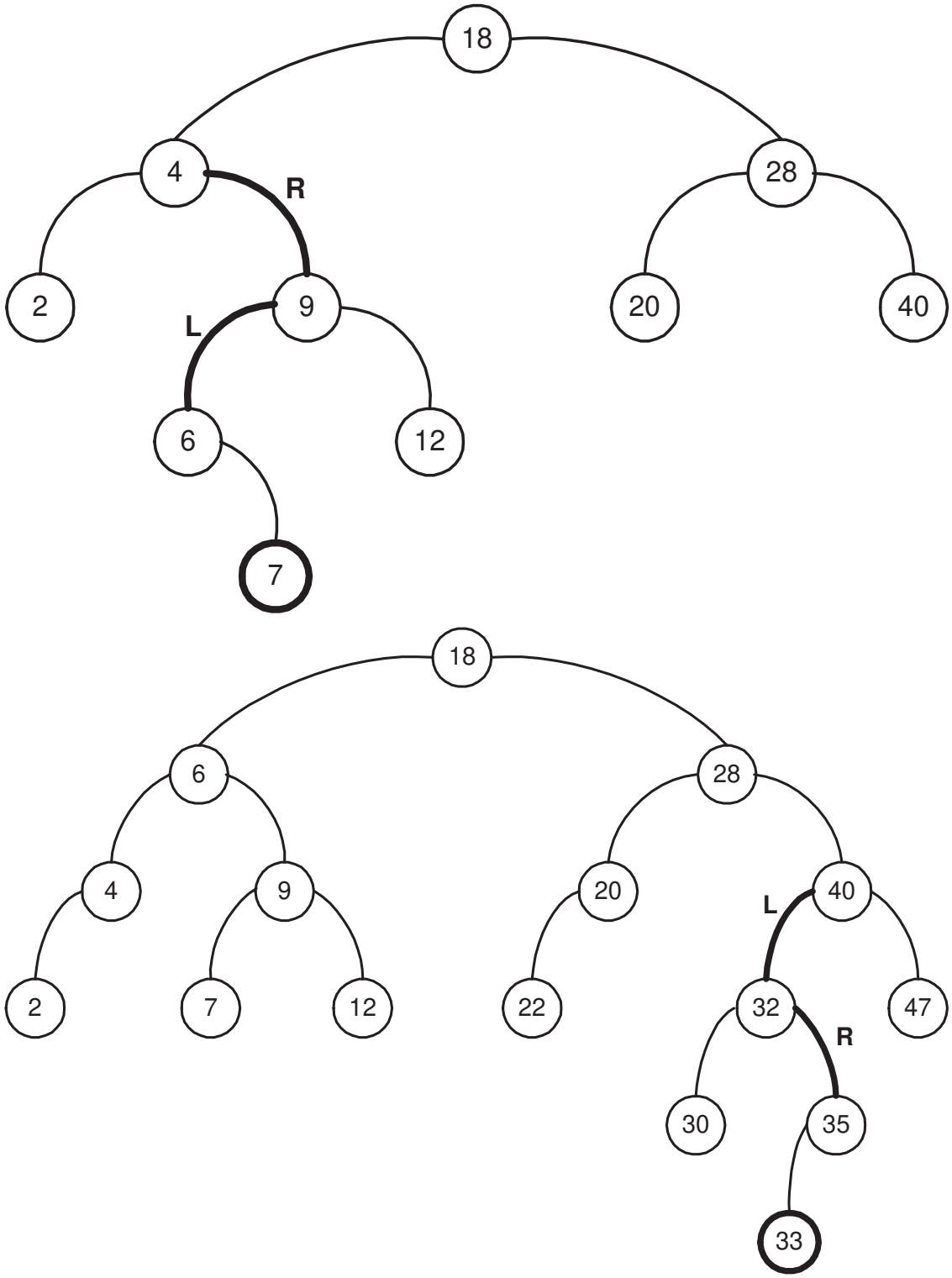



Abbildung 7.9: Einfügen in den AVL-Baum

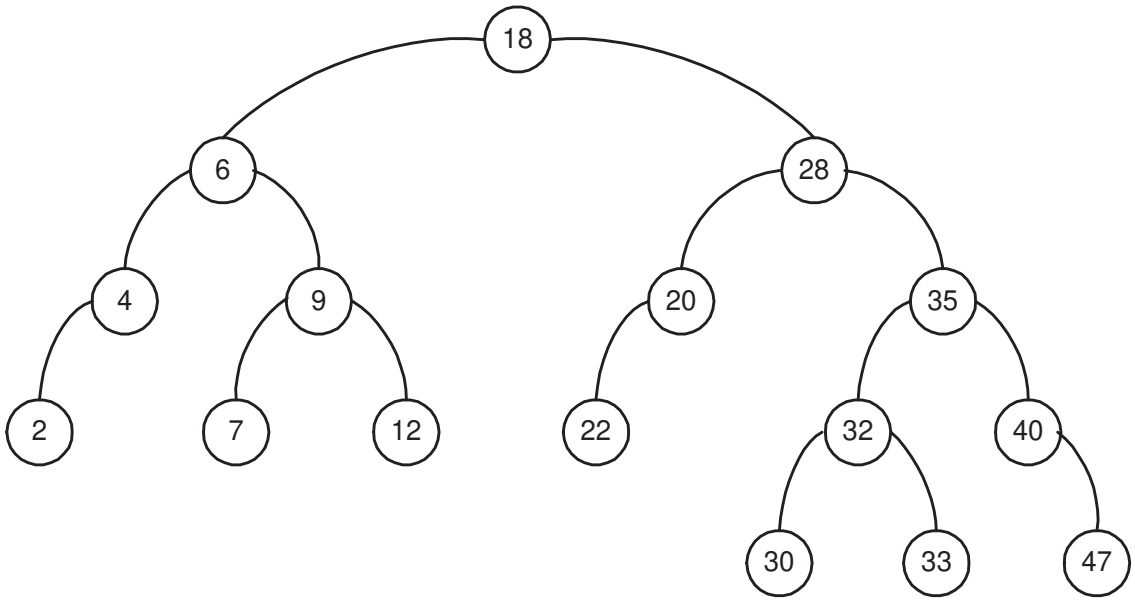



Abbildung 7.10: Einfügen in den AVL-Baum

B-Bäume und deren Varianten bieten sowohl für die Auslastung als auch für die Anzahl der Seitenzugriffe bei einer Suche feste Grenzen. Ein Seitenwechsel ist nur notwendig, wenn eine Kante verfolgt wird. Die maximale Anzahl der Seitenzugriffe während eines Suchvorgangs wird also durch die Höhe des Baums begrenzt. Bild 7.11 zeigt eine schematische Darstellung eines B-Baums. Aufgrund der Balancierung ist jeder Weg von der Wurzel zu einem Blatt im Baum gleich lang.

In dieser Darstellung nehmen wir vereinfachend an, daß eine Seite, entsprechend einem Knoten des Baums, maximal vier Einträge aufnehmen kann. In der Praxis ist das Fassungsvermögen von Seiten um Größenordnungen höher. Ein Eintrag besteht aus dem Schlüssel $S_i$ und dem Datensatz $D_i$, der diesen Schlüssel enthält. Bei einem Sekundärindex werden nicht die Datensätze, sondern die TIDs der Datensätze (also Verweise) eingetragen. Zu jedem Eintrag $S_i$ gibt es einen Verweis $V_{i-1}$ auf Knoten, die kleinere Schlüsselwerte enthalten, und einen Verweis $V_i$ entsprechend auf Knoten mit größeren Schlüsselwerten. Der in Abbildung 7.11 vergrößert dargestellte Knoten enthält zwei Einträge, die verbleibenden zwei Einträge sind frei.

Ein B-Baum mit Grad $k$ ist also durch die folgenden Eigenschaften charakterisiert:

1. Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge.

2. Jeder Knoten außer der Wurzel hat mindestens $k$ und höchstens $2k$ Einträge. Die Wurzel hat zwischen einem und $2k$ Einträgen. Die Einträge werden in allen Knoten sortiert gehalten.

3. Alle Knoten mit $n$ Einträgen, außer den Blättern, haben $n + 1$ Kinder.

4. Seien $S_1, \ldots, S_n$ die Schlüssel eines Knotens mit $n + 1$ Kindern. $V_0, V_1, \ldots, V_n$ seien die Verweise auf diese Kinder. Dann gilt:

   (a) $V_0$ weist auf den Teilbaum mit Schlüsseln kleiner als $S_1$.

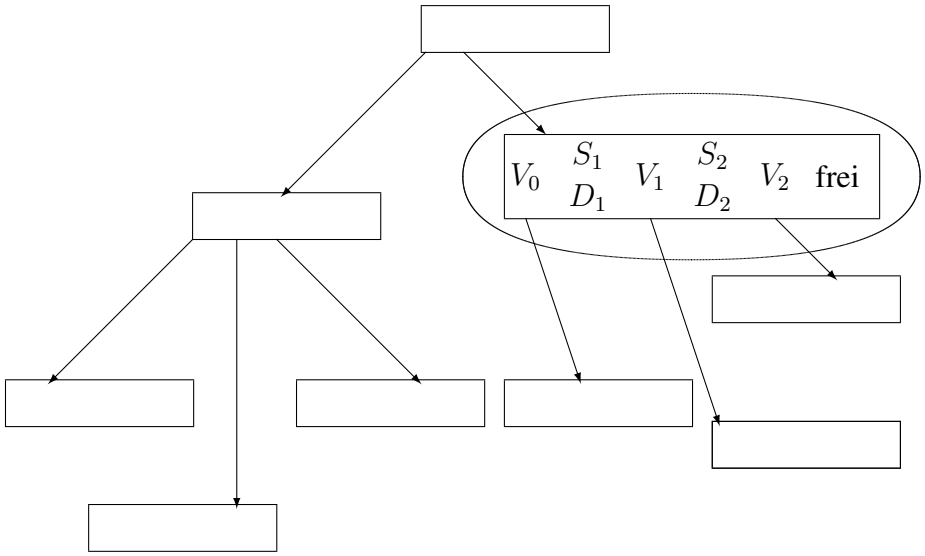



Abbildung 7.11: Aufbau eines B-Baums

(b) $V_i$ $(i = 1, \ldots, n-1)$ weist auf den Teilbaum, dessen Schlüssel zwischen $S_i$ und $S_{i+1}$ liegen.

(c) $V_n$ weist auf den Teilbaum mit Schlüsseln größer als $S_n$.

(d) In den Blattknoten sind die Zeiger nicht definiert.

In der obigen Definition nehmen wir zur Vereinfachung die Eindeutigkeit des Schlüssels an (siehe dazu auch Aufgabe **??**).

Um die geforderte Eigenschaft nach einer Mindestbelegung von $k$ Einträgen pro Knoten einhalten zu können, müssen unter Umständen beim Löschen unterbelegte Knoten zusammengelegt werden. Ebenso muß, falls bei der maximalen Belegung von $2k$ Einträgen noch ein weiterer eingefügt werden soll, ein Knoten eventuell aufgeteilt werden. In manchen Fällen ist auch ein Ausgleich mit benachbarten Knoten möglich.

**Einfügen von Schlüsseln.** Das wollen wir anhand des vereinfachten Beispiels in Bild 7.12 demonstrieren. In dem dort abgebildeten B-Baum vom Grad 2 soll die Zahl 17 eingefügt werden - also der Datensatz mit dem Schlüssel 17, der hier allerdings nicht näher gezeigt wird. Es wird zunächst durch Absteigen im Baum die Einfügestelle gesucht, in diesem Fall zwischen der Zahl 16 und 18. In dem zugehörigen Knoten ist allerdings nicht mehr genügend Platz vorhanden; er muß aufgeteilt werden. Dazu wird der mittlere Eintrag, die Zahl 16, hochgeschoben in den Elternknoten. Die Zahlen, die vorher links und rechts von der 16 standen, bilden danach je einen separaten Knoten, wie in Bild 7.13 dargestellt. Diese beiden neuen Knoten erfüllen die geforderte Minimalbelegung. Unter Umständen kann sich ein Aufteilvorgang bis zur Wurzel fortsetzen. In dem Fall, daß auch die Wurzel vollständig belegt ist, muß ein neuer Wurzelknoten angelegt werden, und die ursprünglichen Einträge der Wurzel werden auf die zwei neuen Kinder aufgeteilt. Der Baum wächst so um eine Stufe in die Höhe. Abbildung 7.14 beschreibt den Einfügenvorgang noch einmal in algorithmischer Form.

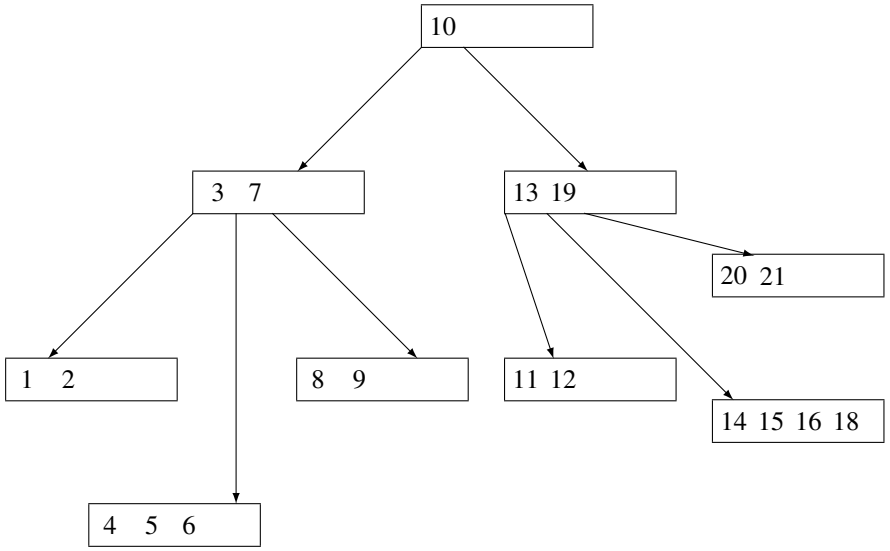



Abbildung 7.12: Ein Beispielbaum ($k = 2$)

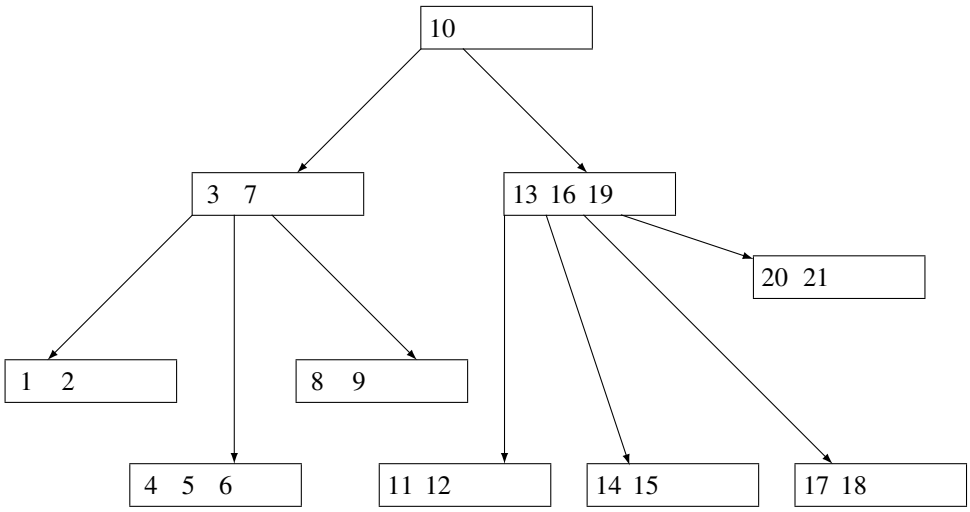



Abbildung 7.13: Einfügen einer 17

1. Führe eine Suche nach dem Schlüssel durch; diese endet (scheitert) an der Einfügestelle.
2. Füge den Schlüssel dort ein.
3. Ist der Knoten überfüllt, teile ihn
   - Erzeuge einen neuen Knoten und belege ihn mit den Einträgen des überfüllten Knotens, deren Schlüssel größer ist als der des mittleren Eintrags.
   - Füge den mittleren Eintrag im Vaterknoten des überfüllten Knotens ein.
   - Verbinde den Verweis rechts des neuen Eintrags im Vaterknoten mit dem neuen Knoten.
4. Ist der Vaterknoten jetzt überfüllt?
   - Handelt es sich um die Wurzel, so lege eine neue Wurzel an.
   - Wiederhole Schritt 3 mit dem Vaterknoten.

Abbildung 7.14: Algorithmus zum Einfügen in einen B-Baum

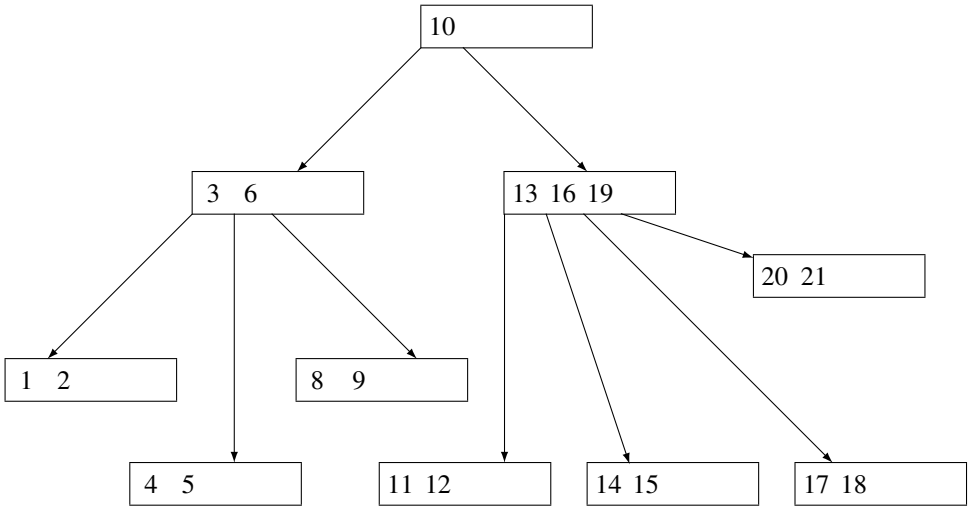



Abbildung 7.15: Löschen der 7

**Löschen eines Schlüssels.** Die Vorgehensweise beim Löschen hängt davon ab, ob ein Eintrag aus einem Blattknoten oder aus einem inneren Knoten entfernt werden soll. In einem Blattknoten kann ein Eintrag einfach gelöscht werden. In einem inneren Knoten muß die Verbindung zu den Kindern des Knotens bestehen bleiben, daher wird der nächstgrößere (oder nächstkleinere) Schlüssel gesucht und an die Stelle des alten Schlüssels plaziert. In beiden Fällen kann ein Blattknoten unterbelegt zurückbleiben – im zweiten Fall ist es der ursprüngliche Aufenthaltsort des nächstgrößeren (-kleineren) Schlüssels. Damit der Baum nicht gegen die Bedingung 2 der Definition verstößt, wird der Knoten mit einem seiner Nachbarn entweder ausgeglichen oder verschmolzen. Ein Ausgleich bewirkt die gleichmäßige Verteilung der Inhalte der beiden Knoten. Ein Verschmelzen ist nur möglich, wenn beide Knoten minimal belegt sind. Dann tritt an deren Stelle ein Knoten, der zusätzlich zu deren Inhalt noch den zugehörigen Schlüssel aus dem Vaterknoten enthält. Das kann wiederum zur Unterbelegung des Vaterknotens führen und den Verschmelzungs- bzw. Ausgleichsvorgang nach oben fortsetzen.

Bild 7.15 zeigt den B-Baum, nachdem die 7 gelöscht wurde. Als Ausgleich sollte die 8 an die Stelle der 7 geschoben werden, das hätte zu einer Unterbelegung geführt. Ein Aus-

gleich mit dem Nachbarknoten führt dazu, daß die $6$ den Platz im Vaterknoten einnimmt. Ein weiterer Löschversuch in diesem Teil des Baums, z.B. der $5$, zieht kompliziertere Aktionen nach sich: Jetzt ist eine Verschmelzung zweier Blattknoten notwendig. Zusätzlich ergibt sich daraus eine Unterbelegung des Vaterknotens, der durch einen Ausgleich mit der rechten Baumhälfte beseitigt werden müßte. Erfahrungen mit realen Datenbanken zeigen jedoch, daß Löschoperationen im Verhältnis zu Einfügeoperationen selten sind. Daher wird in Implementierungen von B-Bäumen häufig sogar auf Verschmelzungen ganz verzichtet – wodurch natürlich die in Bedingung 2 geforderte Minimalbelegung zumindest zeitweise verletzt werden kann.

Es soll noch einmal betont werden, daß die vorgeführten Größenordnungen nicht realistisch sind. Reale B-Bäume haben Verzweigungsgrade in der Größenordnung von 100 – abhängig natürlich von der Größe der Datensätze und dem Fassungsvermögen der Seiten. Deshalb reichen z.B. etwa vier Seitenzugriffe (entsprechend der Höhe des B-Baums), um einen Datensatz unter $10^7$ Einträgen zu finden.

## 7.5 B$^+$-Bäume

Dadurch, daß jeder Knoten eine Seite des Hintergrundspeichers belegt, hängt die Höhe eines B-Baums direkt mit der Anzahl der Seitenzugriffe zum Auffinden eines Datums zusammen. Bei B-Bäumen ist also ein hoher Verzweigungsgrad wünschenswert, denn je weiter ein Baum verzweigt ist, desto flacher ist er. Der Verzweigungsgrad bei B-Bäumen hängt von der Satzgröße ab, wenn die Datensätze innerhalb der Knoten gespeichert werden. Bei B$^+$-Bäumen[1] wird die Höhe dadurch reduziert, daß Daten nur noch in den Blättern gespeichert werden. Daher spricht man auch von einem *hohlen* Baum. Die inneren Knoten enthalten lediglich Referenzschlüssel $R_i$ als Wegweiser („road map“). Eine Suche nach einem Datensatz $D_i$ muß deshalb immer komplett bis zu den Blättern durchgeführt werden. Der schematische Aufbau eines B$^+$-Baums ist in Bild 7.16 dargestellt.

Um zusätzlich eine effiziente sequentielle Verarbeitung der Datensätze zu ermöglichen, sind die Blattknoten jeweils mit einem Zeiger auf den vorhergehenden ($P$) und nachfolgenden Blattknoten ($N$) in der gewünschten Suchreihenfolge verbunden.

Ein B$^+$-Baum vom Typ $(k, k^*)$ hat also folgende Eigenschaften:

1. Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge.

2. Jeder Knoten – außer Wurzeln und Blättern – hat mindestens $k$ und höchstens $2k$ Einträge. Blätter haben mindestens $k^*$ und höchstens $2k^*$ Einträge. Die Wurzel hat entweder maximal $2k$ Einträge, oder sie ist ein Blatt mit maximal $2k^*$ Einträgen.

3. Jeder Knoten mit $n$ Einträgen, außer den Blättern, hat $n + 1$ Kinder.

4. Seien $R_1, \ldots, R_n$ die Referenzschlüssel eines inneren Knotens (d.h. auch der Wurzel) mit $n + 1$ Kindern. Seien $V_0, V_1, \ldots, V_n$ die Verweise auf diese Kinder.

[1] Die Terminologie ist hier nicht ganz klar, vielfach wird auch der Name B*-Baum benutzt. Der von Knuth (1973) ursprünglich definierte B*-Baum ist eine Variante des B-Baums, bei dem eine Mindestbelegung der Knoten von 2/3 durch Umverteilungen garantiert wird. Der in diesem Abschnitt vorgestellte Baum wurde von Knuth nicht mit Namen versehen. Wir folgen einem Vorschlag von Comer (1979) und nennen ihn B$^+$-Baum.

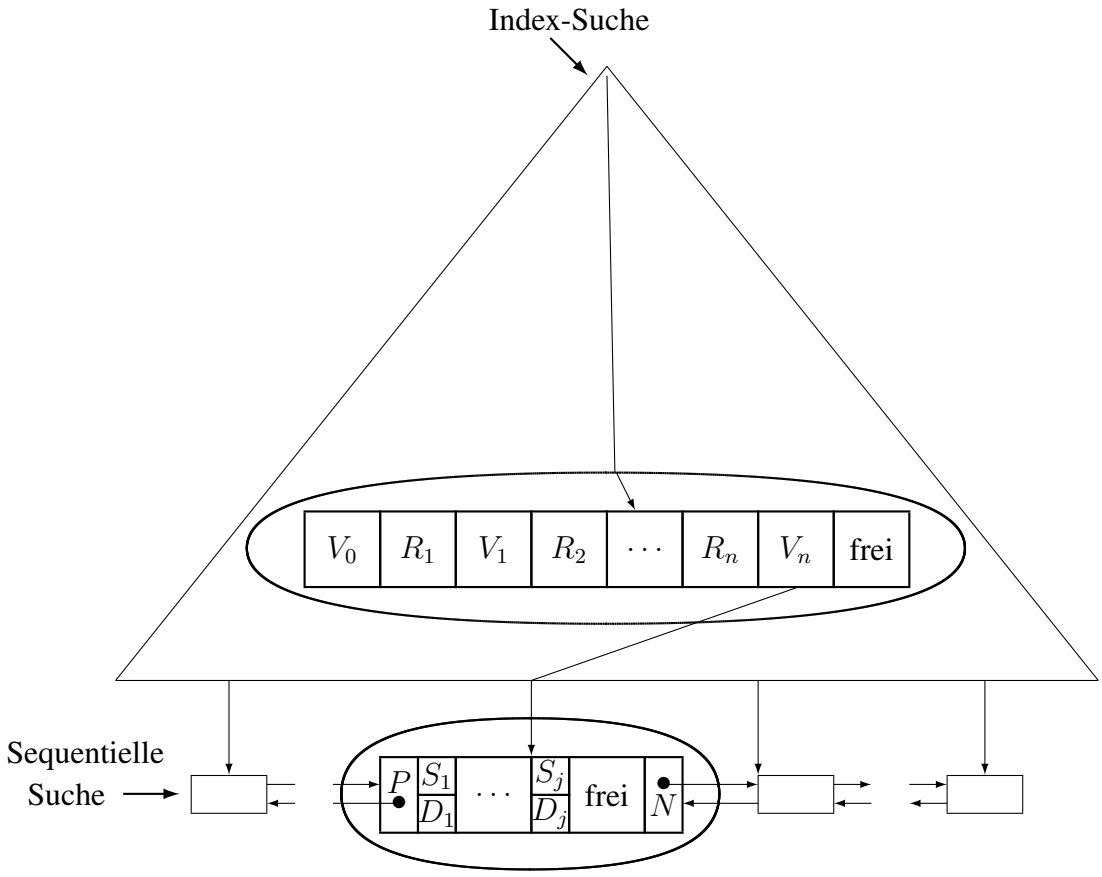



Abbildung 7.16: Schematischer Aufbau eines B$^+$-Baums

(a) $V_0$ verweist auf den Teilbaum mit Schlüsseln kleiner oder gleich $R_1$.

(b) $V_i$ $(i = 1, \ldots, n-1)$ verweist auf den Teilbaum, dessen Schlüssel zwischen $R_i$ und $R_{i+1}$ liegen (einschließlich $R_{i+1}$).

(c) $V_n$ verweist auf den Teilbaum mit Schlüsseln größer als $R_n$.

Ein zusätzlicher Vorteil des B$^+$-Baums ist die effizientere Wartung durch die Verwendung von Referenzschlüsseln. Referenzschlüssel müssen nicht unbedingt einem realen Schlüssel entsprechen. Daher brauchen Referenzschlüssel nur gelöscht zu werden, falls Blattknoten zusammengelegt werden und eventuell bei den sich daraus ergebenden weiteren Verschmelzungen. Beim Aufteilen von Knoten wird der mittlere Referenzschlüssel nicht in den Vaterknoten verschoben, sondern wandert z.B. in die linke Hälfte. Im Vaterknoten wird eine Kopie eingetragen.

## 7.6 Präfix-B$^+$-Bäume

Eine zusätzliche Verbesserungsmöglichkeit bei B$^+$-Bäumen ist der Einsatz von Schlüsselpräfixen anstelle von kompletten Schlüsseln. Werden z.B. längere Zeichenketten als Schlüssel verwendet, wird der Verzweigungsgrad des B$^+$-Baums klein. Da B$^+$-Bäume nur Referenzschlüssel enthalten, braucht nur irgendein Schlüssel gefunden zu werden, der die

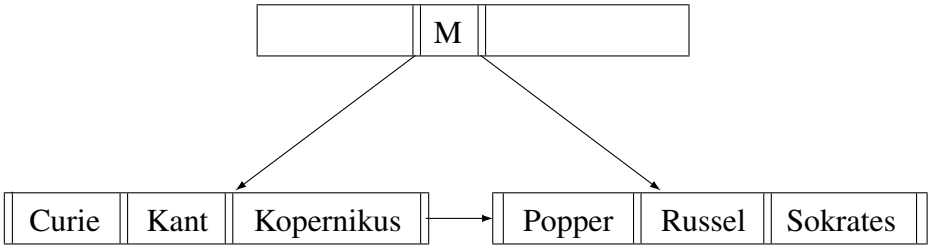



Abbildung 7.17: Schematische Darstellung zum Präfix-B$^+$-Baum

Teilbäume zur linken und zur rechten trennt. Die Situation wird schematisch in Bild 7.17 verdeutlicht.

Normalerweise wäre „Kopernikus“ der eingetragene Referenzschlüssel, anhand dessen die Verzweigungsentscheidung getroffen wird. Platzsparender ist es jedoch, irgendeinen anderen kürzestmöglichen Schlüssel $R$ einzutragen, der die Eigenschaft

$$\text{Kopernikus} \leq R < \text{Popper}$$

erfüllt, z.B. ein „M“. Bei dicht beieinanderliegenden Schlüsseln kann das Verfahren jedoch versagen, z.B. wenn ein $R$ gesucht wird mit

$$\text{Systemprogramm} \leq R < \text{Systemprogrammierer}$$

# 8. Hashing

Das ultimative Ziel aller Bemühungen um ein gutes physisches Design ist es, wirklich nur diejenigen Seiten vom Hintergrundspeicher zu lesen, die absolut benötigt werden. Hash-Verfahren ermöglichen es, ein bestimmtes Datum im Durchschnitt mit einem bis zwei Seitenzugriffen zu finden. Bäume benötigen Seitenzugriffe in der Ordnung von $\log_k(n)$, wobei $k$ der durchschnittliche Verzweigungsgrad und $n$ die Anzahl der eingetragenen Datensätze ist.[1]

Beim Hashing wird mit Hilfe einer sogenannten *Hashfunktion* der Schlüssel auf einen Behälter (engl. *bucket*) abgebildet, der das dem Schlüssel zugehörige Datum aufnehmen soll. Im allgemeinen ist nicht für den gesamten Wertebereich des Schlüssels Platz im Speicher vorhanden. Es kann daher vorkommen, daß mehrere Datensätze an die gleiche Stelle gespeichert werden sollen. In diesem Fall wird entweder eine hier nicht weiter ausgeführte Kollisionsbehandlung eingeschaltet oder das sogenannte *offene Hashing* verwendet, das weiter unten erläutert wird.

Formaler ausgedrückt ist also eine Hashfunktion (oder auch *Schlüsseltransformation*) $h$ eine Abbildung

$$h : S \to B,$$

wobei $S$ eine beliebig große Schlüsselmenge und $B$ eine Nummerierung der $n$ Behälter, also ein Intervall $[0..n)$ ist. Normalerweise ist die Anzahl der möglichen Elemente in der Schlüsselmenge sehr viel größer als die Anzahl der zur Verfügung stehenden Behälter ($|S| \gg |B|$), daher kann $h$ i.a. nicht injektiv sein. Es sollte aber die Elemente von $S$ gleichmäßig auf $B$ verteilen, da eine Kollisionsbehandlung bzw. der Überlauf eines Behälters zusätzlichen Aufwand verursacht. Gilt für zwei Schlüssel $S_1$ und $S_2$, daß $h(S_1) = h(S_2)$ ist, nennt man $S_1$ und $S_2$ *synonym*.

Nehmen wir an, die Daten der Studenten werden häufig anhand ihrer Matrikelnummer gesucht. Deshalb sollen sie in eine Hash-Tabelle eingetragen werden, für die drei Seiten reserviert sind, die jeweils zwei Einträge aufnehmen können. Häufig wird als Hashfunktion der Schlüsselwert modulo der Tabellengröße verwendet. Für einen aus drei Seiten bestehenden Speicherbereich könnte also die folgende Hashfunktion verwendet werden:

$$h(x) = x \bmod 3$$

Dieses *Divisionsrestverfahren* ist die gebräuchlichste Art einer Hashfunktion. Es hat sich gezeigt, daß man am günstigsten eine Primzahl für die Berechnung des Divisionsrestes wählt, um eine gute Streuung zu gewährleisten.

Bild 8.1 zeigt die Hash-Tabelle nachdem Xenokrates ($h(24002) = 2$), Jonas ($h(25403) = 2$) und Schopenhauer ($h(27550) = 1$) eingetragen wurden. Die durchgezogenen Linien deuten Seitengrenzen an.

Versucht man, in diese Tabelle noch Fichte ($h(26120) = 2$) einzutragen, tritt ein Überlauf auf, da Seite 2 bereits durch Xenokrates und Jonas belegt ist. Beim offenen Hashing wird nun in der Seite ein Zeiger auf einen weiteren Behälter gespeichert. Dieser weitere

[1] Diese Zahlen werden meist durch Pufferungseffekte relativiert.

| | |
|---|---|
| 0 | |
| | |
| 1 | (27550, 'Schopenhauer', 6) |
| | |
| 2 | (24002, 'Xenokrates', 18) |
| | (25403, 'Jonas', 12) |

Abbildung 8.1: Eine aus drei Seiten bestehende Hash-Tabelle

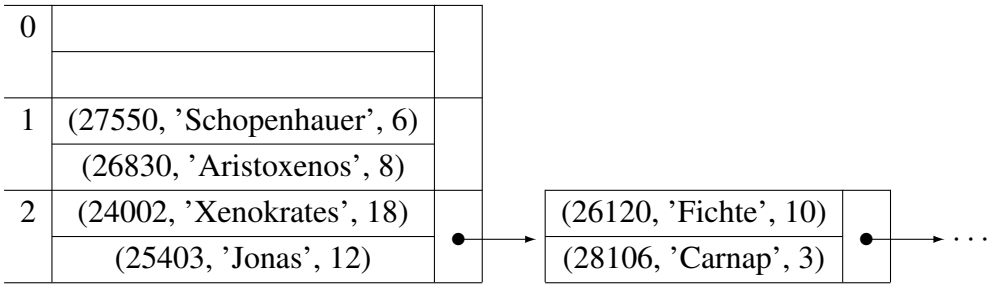



Abbildung 8.2: Kollisionsbehandlung durch Überlaufbehälter

Behälter ist ein Überlaufbereich fester Größe (in unserem Fall eine Seite), der die zusätzlichen Kandidaten für den zugehörigen Speicherplatz enthält. Ein Überlaufbehälter kann wiederum überlaufen und einen Verweis auf weitere Behälter enthalten (siehe Bild 8.2). Man sieht schon, daß unsere Hashfunktion $h(x)$ nicht gut gewählt wurde. Es gibt zu viele Matrikelnummern, die auf den Platz $2$ abgebildet werden.

### 8.0.1 Hashfunktionen/Kollisionen

Wenn zwei Einträge auf denselben Bucket abgebildet werden, spricht man von einer Kollision. Ziel einer guten Hashfunktion $h$ sollte eine gleichmäßige Streuung der Schlüssel auf die Buckets sein.

Als Anzahl der Buckets sollte man eine Primzahl $N$ wählen, damit die *modulo*-Funktion die Schlüssel geleichmäßig streut.

$$h(K) = K \bmod N$$

Wenn $N$ keine Primzahl ist, werden Cluster nicht gut gestreut. Als Beispiel betrachten wir eine gerade Zahl $N$. Dann werden gerade Schlüssel immer auf gerade Speicheradressen abgebildet. Nehmen wir $N = 12$:

- 38 mod 12 = 2
- 40 mod 12 = 4
- 42 mod 12 = 6
- 44 mod 12 = 8

Im Vergleich dazu betrachten wir jetzt $N = 13$:

- 38 mod 13 = 12
- 40 mod 13 = 1
- 42 mod 13 = 3
- 44 mod 13 = 5

Wie wahrscheinlich sind Kollisionen? Selbst bei sehr guter Wahl der Hashfunktion sind Kollisionen in der Praxis nicht nur nicht auszuschließen, sondern sie sind so gut wie sicher. Wir wollen uns dies an einem kleinen Beispiel veranschaulichen. Wir gehen wieder von 13 Buckets aus, auf die wir 8 Einträge platzieren wollen.

Die Wahrscheinlickeit, dass es keine Kollision gibt errechent sich wie folgt:

$$W(kK) = 1 * 12/13 * 11/13 * 10/13 * 9/13 * 8/13 * 7/13 * 6/13 \approx 0,06$$

Daraus folgt für die Wahrscheinlichkeit, dass es mindestens eine Kollision gibt:

$$W(meK) = 1 - W(kK) \approx 0,94$$

Also gibt es mit einer Wahrscheinlichkeit von 94% mindestens eine Kollision – bei bestmöglicher Hashfunktion, die die Schlüssel absolut gleichmäßig streut.

Diese Beobachtung ist auch als so genanntes Geburtstags-Paradox in der Literatur bekannt. Die Wahrscheinlichkeit, dass es in einer Schulklasse von 30 Schülern mindestens zwei Schüler gibt, die am selben Tag Geburtstag feiern, ist relativ hoch. Wie hoch?

## 8.1 Erweiterbares Hashing

Das bislang vorgestellte Hash-Verfahren ist für eine reale Datenbasis zu statisch. Da eine einmal angelegte Hash-Tabelle nicht effizient vergrößert werden kann, gibt es nur zwei wenig wünschenswerte Alternativen, wenn viele Einfügeoperationen erwartet werden: Entweder es wird von vornherein sehr viel Platz für die Tabelle reserviert, so daß der Platz zunächst verschwendet ist, oder es entstehen im Laufe der Zeit immer längere Überlaufketten. Diese Überlaufketten können nur durch Änderung der Hashfunktion und aufwendige Reorganisation der Tabelle beseitigt werden.

Eine Verbesserung bietet das *erweiterbare Hashing*. Dazu wird die Hashfunktion $h$ so modifiziert, daß sie nicht mehr unbedingt auf einen Index eines tatsächlich vorhandenen Behälters abbildet, sondern auf einen wesentlich größeren Bereich. Das Ergebnis einer Berechnung von $h(x)$ wird binär dargestellt und nur ein Präfix dieser binären Darstellung berücksichtigt, der dann auf den tatsächlich verwendeten Behälter verweist.

Bild 8.7 zeigt eine schematische Darstellung des erweiterbaren Hashings. Die binäre Darstellung des Ergebnisses der Hashfunktion wird in zwei Teile aufgeteilt: $h(x) = dp$. $d$ gibt die Position des Behälters im *Verzeichnis* an. Das Verzeichnis (engl. *directory*) faßt in der gezeigten Konstellation $2^2$ Einträge, also werden zwei Bits für $d$ gebraucht. Die Größe von $d$ wird die *globale Tiefe* $t$ genannt. $p$ ist der zur Zeit nicht benutzte Teil des Schlüssels.

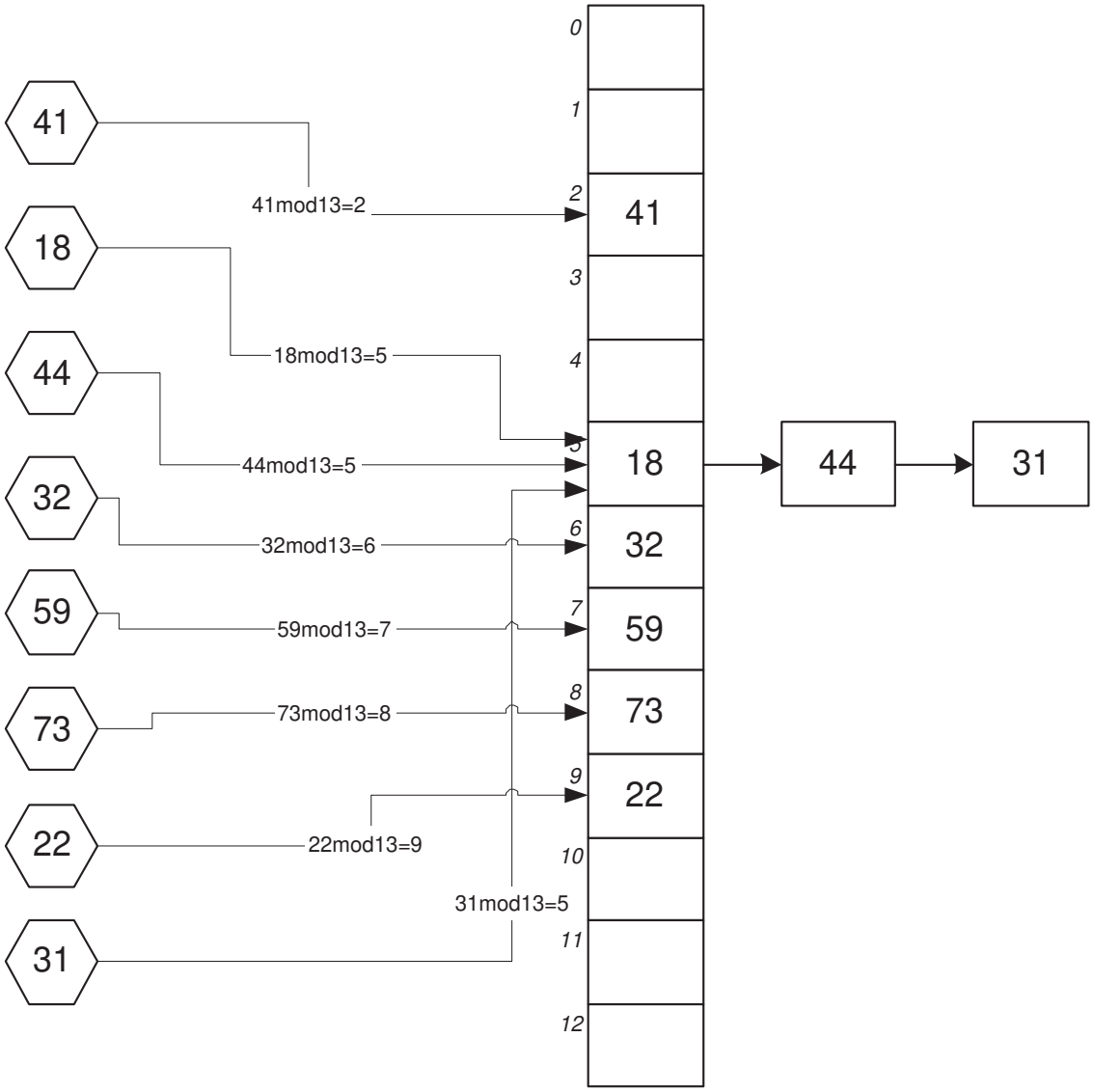



Abbildung 8.3: Hash-Tabelle mit Verkettung als Kollisionsbehandlung

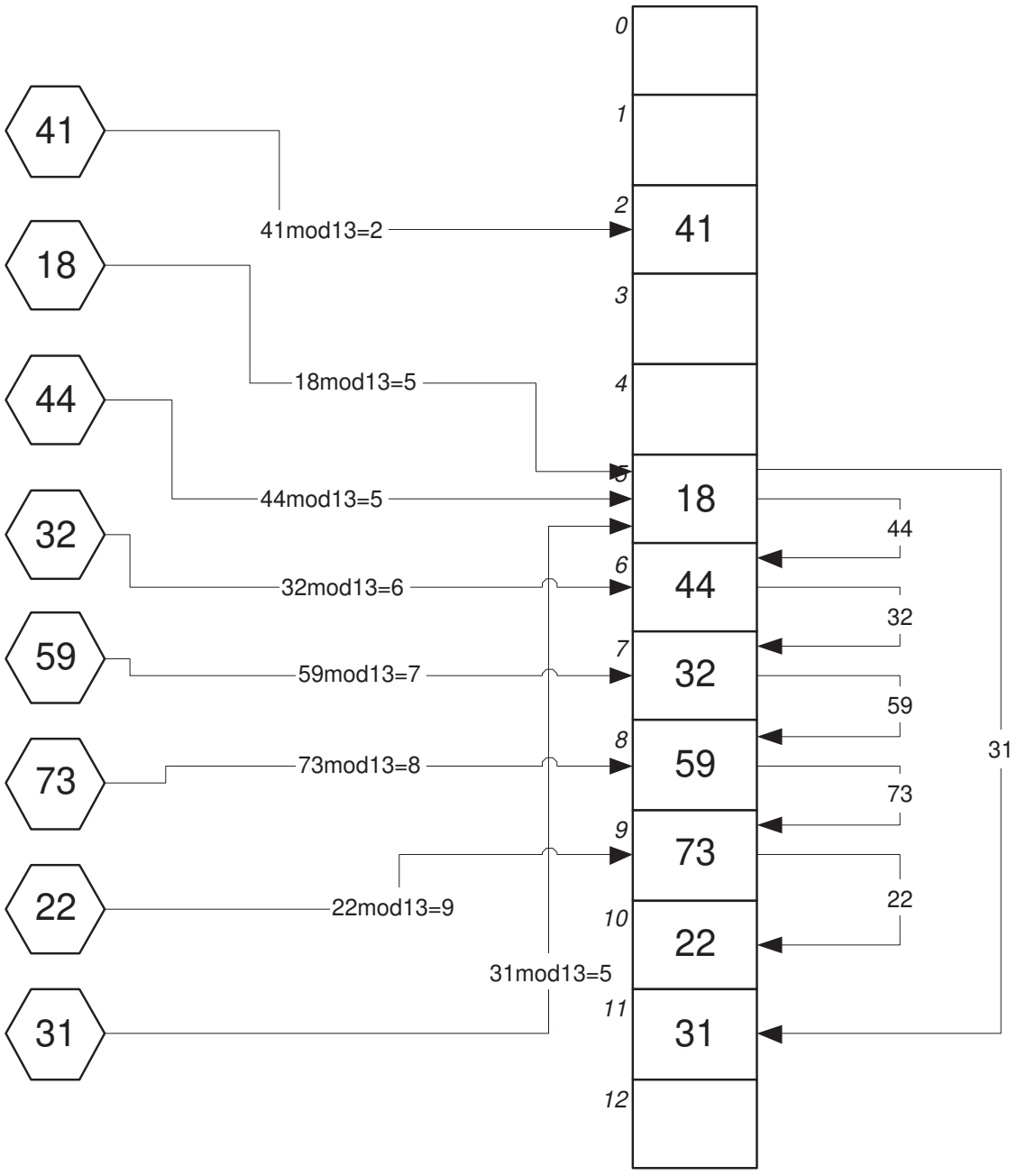



Abbildung 8.4: Hash-Tabelle mit *Linear Probing* als Kollisionsbehandlung

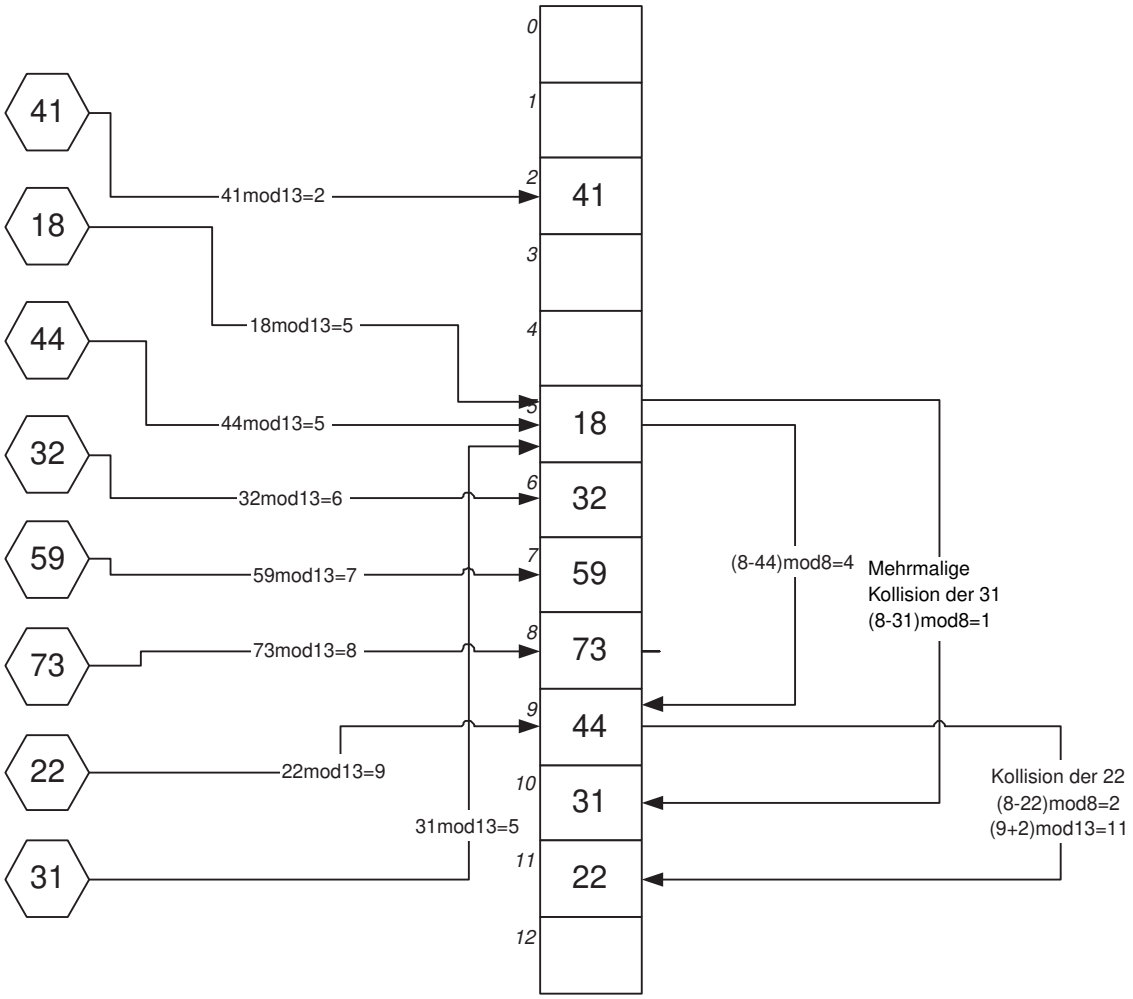



Abbildung 8.5: Hash-Tabelle mit *Double Hashing* als Kollisionsbehandlung: $h_1(K) = K \bmod 13$ und $h_2(K) = (8 - K)\ mod\ 8$ und der Funktion $h_j(K) = (h_1(K) + j * h_2(K))\ mod\ 13$, wobei $j$ inkrementiert wird, bis ein freier Slot gefunden wird.

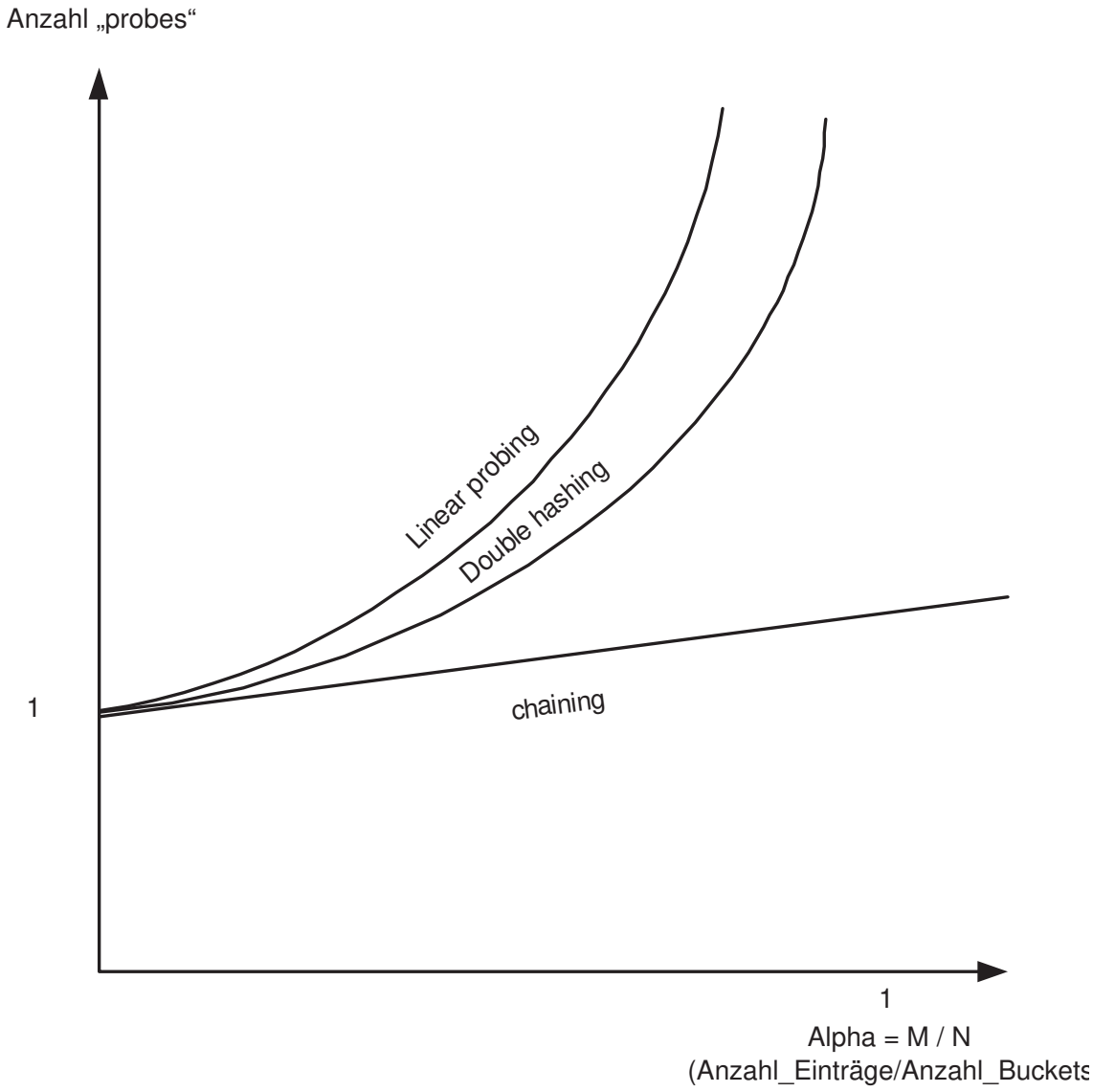



Abbildung 8.6: Leistungsfähigkeit der Hash-Verfahren

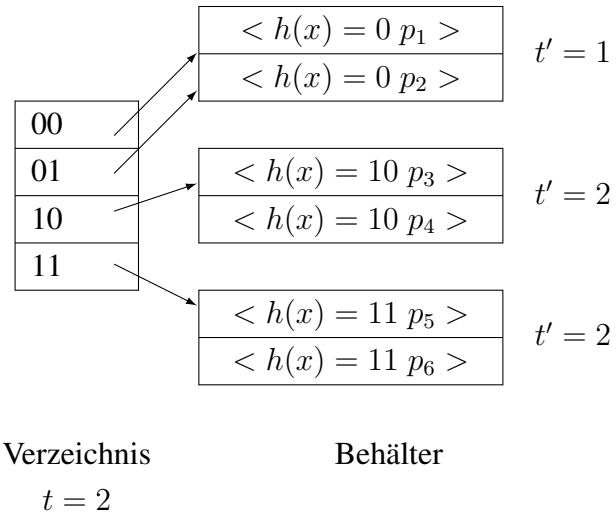



Abbildung 8.7: Schematische Darstellung des erweiterbaren Hashings

Müßte ein neuer Datensatz in einen bereits vollen Behälter eingetragen werden, würde er aufgeteilt werden. Die Aufteilung erfolgt anhand eines weiteren Bits des bisher unbenutzten Teils $p$. Ist die globale Tiefe nicht ausreichend, um den Verweis auf den neuen Behälter eintragen zu können, muß das Verzeichnis verdoppelt werden. Es wäre – insbesondere bei Anwendung der Hashfunktion auf einen Nichtschlüssel – denkbar, daß mehr Datensätze auf denselben (vollständigen) Hashwert abgebildet werden, als in einem Behälter Platz haben. In diesem Fall muß man das erweiterbare Hashing mit einer Überlauftechnik wie in Abbildung 8.2 kombinieren.

Die *lokale Tiefe* $t'$ eines Behälters gibt an, wieviele Bits des Schlüssels für diesen Behälter tatsächlich verwendet werden. Eine Verdoppelung des Verzeichnisses erfolgt also, wenn nach einer Aufteilung eines Behälters die lokale Tiefe größer als die globale Tiefe ist.

Bild 8.8 zeigt einen Hash-Index auf dem Attribut *PersNr* der Relation *Professoren*, in dem schon Sokrates, Russel und Kopernikus eingetragen sind. Als Hashfunktion wird die umgedrehte binäre Darstellung der Personalnummer verwendet. In realistischen Anwendungen sollte jedoch zur besseren Streuung noch z.B. das Divisionsrestverfahren vorgeschaltet werden. Zur Orientierung ist oberhalb des Indexes eine Tabelle mit den Hashwerten der Personalnummern angegeben. Nun soll Descartes eingefügt werden.

Descartes hat die Personalnummer 2129 und fällt in den bereits von Sokrates und Kopernikus vollständig belegten Behälter. Die globale Tiefe stimmt mit der lokalen Tiefe dieses Behälters überein, also muß das Verzeichnis verdoppelt werden (Bild 8.9). Durch die Vergrößerung des maßgebenden Teils des Hash-Wertes kann Descartes jetzt eingeordnet werden.

Ist durch die Hinzunahme eines neuen Bits zum relevanten Teil des Hash-Wertes immer noch keine Aufteilung des angestrebten Behälters möglich, muß das Verzeichnis nochmals verdoppelt werden.

Werden Daten gelöscht, ist es unter Umständen möglich, Behälter wieder zu verschmelzen oder gar das Verzeichnis zu halbieren. Eine Verschmelzung ist immer dann möglich, wenn sich der Inhalt zweier benachbarter Behälter zusammen in einem unter-

| | $h(x)$ | |
|---|---|---|
| $x$ | $d$ | $p$ |
| 2125 | 1 | 01100100001 |
| 2126 | 0 | 11100100001 |
| 2127 | 1 | 11100100001 |

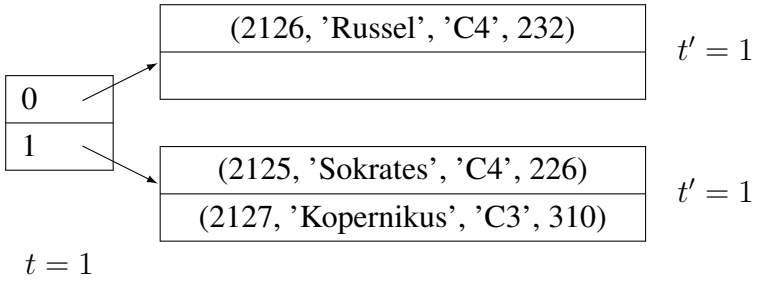



Abbildung 8.8: Ein Hash-Index

| | $h(x)$ | |
|---|---|---|
| $x$ | $d$ | $p$ |
| 2125 | 10 | 1100100001 |
| 2126 | 01 | 1100100001 |
| 2127 | 11 | 1100100001 |
| 2129 | 10 | 0010100001 |

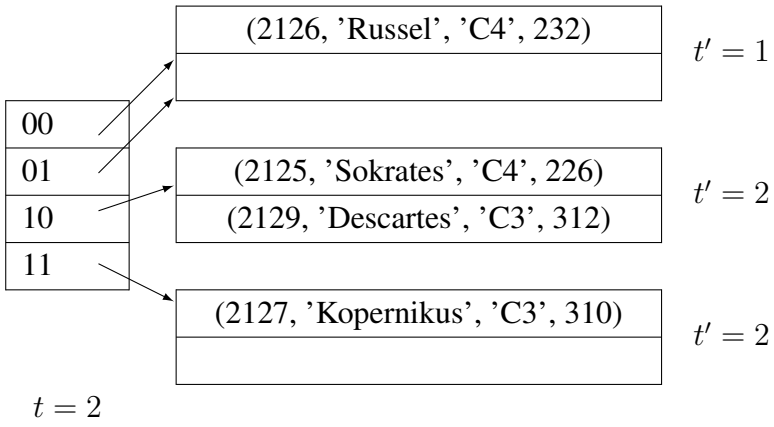



Abbildung 8.9: Einfügen von (2129, Descartes, C3, 312)

bringen läßt. „Benachbart“ sind Behälter, wenn sie die gleiche lokale Tiefe haben und der Wert der ersten $t' - 1$ Bits des Hash-Wertes (von links) übereinstimmt. In Bild 8.9 sind die unteren beiden Behälter benachbart. Sie haben beide die lokale Tiefe $t' = 2$, der $d$-Anteil des Hash-Wertes ist binär $10$ und $11$. Hätten sie insgesamt zwei Einträge, könnten sie wieder zusammengelegt werden. Ihre lokale Tiefe würde sich um eins erniedrigen. Der obere Behälter hat keinen Nachbarn.

Das Verzeichnis kann immer dann halbiert werden, wenn alle lokalen Tiefen echt kleiner sind als die globale Tiefe $t$. Durch das Halbieren erniedrigt sich die globale Tiefe um eins.

## 8.2 Erweiterbares Hashing

Für die Hashfunktion $h$ muss gelten, dass sie einen Bitstring erzeugt, also:

$$h : \text{Schlüsselmenge} \rightarrow \{0,1\}^*$$

Dieser Bitstring muss lang genug sein, um alle Elemente auf die Buckets abbilden zu können. Anfangs wird nur ein kurzer Präfix dieses Bitstrings verwendet, um das betreffende Bucket zu bestimmen. Wenn mehr und mehr Elemente in die Hashstruktur eingefügt werden, wird nach und nach dieser Präfix verlängert.

Der Einfachheit halber verwenden wir in unserem Beispiel als Hashfunktion die gespiegelte Binärdarstellung der Personalnummer. Damit erhalten wir folgende Hashfunktionen-Werte:

- h(004) = 001000000. . . (4=0..0100)
- h(006) = 011000000... (6=0..0110)
- h(007) = 111000000... (7 =0..0111)
- h(013) = 101100000... (13 =0..01101)
- h(018) = 0100100000... (18 =0..010010)
- h(032) = 000001000... (32 =0..0100000)
- h(048) = 000011000... (48 =0..0110000)

In Abbildung 8.10 ist die grundlegende Datenstruktur eines erweiterbaren Hashverfahrens gezeigt. Man unterscheidet die zwei Ebenen:

1. Das Directory, das man sich als binären Entscheidungsbaum vorstellen kann.
2. Die Buckets, in denen die eigentlichen Datensätze abgespeichert sind.

Der binäre Entscheidungsbaum (manchmal auch binärer Trie genannt) dient dazu, anhand des Hashcodes zu ermittlen, in welchem Bucket der betreffende Datensatz sich befindet (bzw. einzufügen ist). Bezogen auf unser Beispiel kann mna erkennen, dass alle Datensätze, deren Hashcode mit einer 1 beginnt, sich in dem Bucket “rechts oben” befinden. Die Datensätze, deren Hashcode mit 000 beginnt, befinden sich in dem Bucket “links unten”.

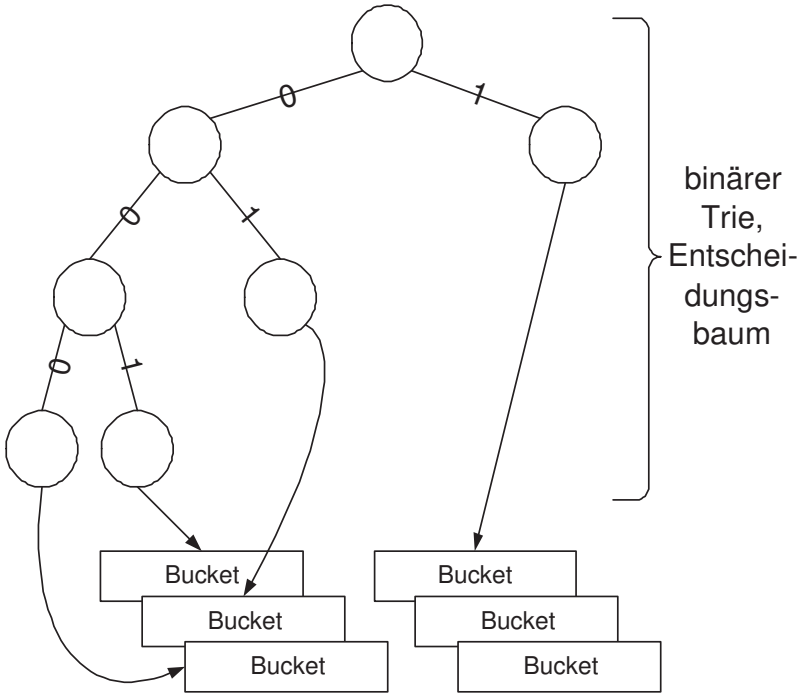



Abbildung 8.10: Illustration der Datenstrukturen des erweiterbaren Hashings: Das Directory und die Buckets

In der Praxis entsprechen diese Buckets natürlich Seiten auf dem Hintergrundspeicher, so dass die Verweise vom Directory auf ein Bucket durch Seitennummern realisiert sind. Verweise auf Buckets finden sich nur in den Blättern des Entscheidungsbaums.

Das Directory des erweitergbaren Hashingverfahrens wird in der Praxis sehr groß, so dass es auch auf den Hintergrundspeicher abgelegt werden muss. Leider eignet sich ein binärer Baum sehr schlecht dazu, ihn auf den Hintergrundspeicher abzubilden. Deshalb wird nicht der Baum abgebildet, sondern alle möglichen Pfade durch diesen binären Entscheidungsbaum werden separat abgebildet. Um dies auf einfache Art und Weise zu erreichen, wird der Entscheidungsbaum durch Zusatzknoten erweitert, so dass alle Pfade die gleiche Länge haben. Dies ist in Abbildung 8.11 gezeigt. Die schattierten Knoten wurden neu eingefügt. Wenn ein Knoten des Entscheidungsbaums erweitert werden muss, so verweisen alle Erweiterungsknoten, die Blättern des Entscheidungsbaums entsprechen, auf das Bucket, das schon von dem erweiterten Knoten aus referenziert wurde. In unserem Beispiel verweisen die vier rechten schattierten Blätter alle auf das Bucket “rechts unten”, auf das ja schon der Knoten, der dem Präfix “1” entsprach, verwiesen hat. Dies ist leicht nachvollziehbar, da diese vier Blätter allen möglichen dreistelligen Hash-Präfixen, die mit einer eins beginnen, entsprechen, also 100, 101, 110 und 111 (von links nach rechts). In der Abbildung 8.11 sind auch einige Datensätze in den Buckets symbolisiert – wir zeigen nur die Hash-Schlüsselwerte: die Datensätze 7 und 13 liegen in dem Bucket rechts oben, da deren Hashcode jeweils mit der 1 beginnt. Die Datensätze 32 und 48 liegen in dem Bucket links unten, da deren Hashcode jeweils mit dem Präfix 000 beginnt.

In Abbildung 8.13 ist dargestellt, wie der Entscheidungsbaum in ein Array abgebildet wird. Jeder Pfad in dem erweiterten Entscheidungsbaum entspricht einer Indexposition in dem Array. Dazu kann man einfach die dreistelligen Binärzahlen als Indexe des Ar-

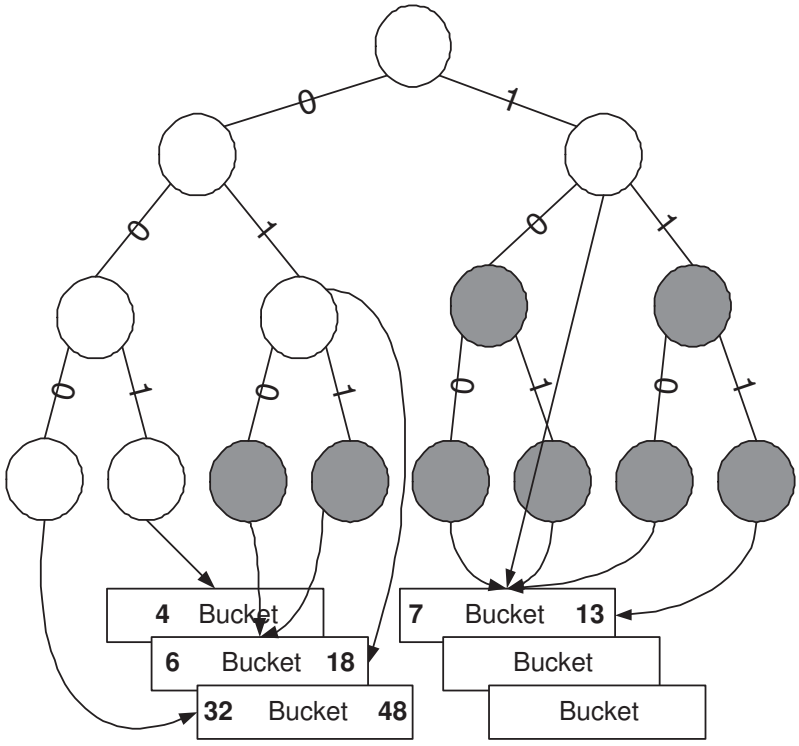



Abbildung 8.11: Die Erweiterung des Entscheidungsbaums

rays auffassen. Das Array selbst enthält dann die Verweise auf die Buckets – also die Seitennummern der Seiten, die den Buckets entsprechen.

Abbildung 8.14 zeigt jetzt das Directory für unser Beispiel als Array. Weiterhin wird die globale Tiefe illustriert, die der Länge des derzeit benutzten Hash-Präfixes entspricht. Sie entspricht somit auch der Länge der Pfade in dem erweiterten binären Entscheidungsbaum. In unserem Beispiel ist die globale Tiefe gleich drei. Die Buckets haben noch eine lokale Tiefe, die angibt wieviele Bitpositionen des Präfixes nötig waren, um ihre Datensätze abzubilden. Da das Bucket rechts oben alle Datensätze mit dem Präfix 1 enthält, ist dessen lokale Tiefe gleich eins. Das Bucket links oben enthält die Datensätze, deren Hashcode mit dem dreistelligen Präfix 001 beginnt. Deshalb ist dessen lokale Tiefe gleich drei – also genaus groß wie die globale Tiefe. Allgemein wird ein Bucket genau $2^{gT-lTE}$ mal referenziert, wobei *gt* und *lt* der globalen bzw. der lokalen Tiefe entsprechen.

In Abbildung 8.14 wird auch das Einfügen neuer Datensätze demonstriert. Wir fügen nacheinander die Datensätze 12 und 20 ein, die beide in das Bucket lnks oben fallen, da ihre Hashcodes jeweils mit 001 beginnen. Das Bucket links oben wird ja von dem Knoten im Entscheidungsbaum referenziert, der dem Hash-Präfix 001 entspricht – man überzeuge sich, dass man in dem Entscheidungsbaum durch die Antwortenfolge “0”, dann wieder “0” und dann die “1” zu diesem Blatt gelangt, in dem dann der Verweis auf das betreffende Bucket abgelegt ist. Das Einfügen der 12 ist noch problemlos möglich, da wir eine Bucketkapazaität von 2 Datensätzen voraussetzen und das Bucket erst einen Datensatz enthält, nämlich die 4. Aber beim Einfügen der 20 kommt es zu einem Überlauf des Buckets. Dieser Überlauf muss durch einen Ausgleich der Datensätze auf zwei Buchets gelöst werden. Dazu ist es also notwendig, eine weitere Bitposition des Hshcodes mit

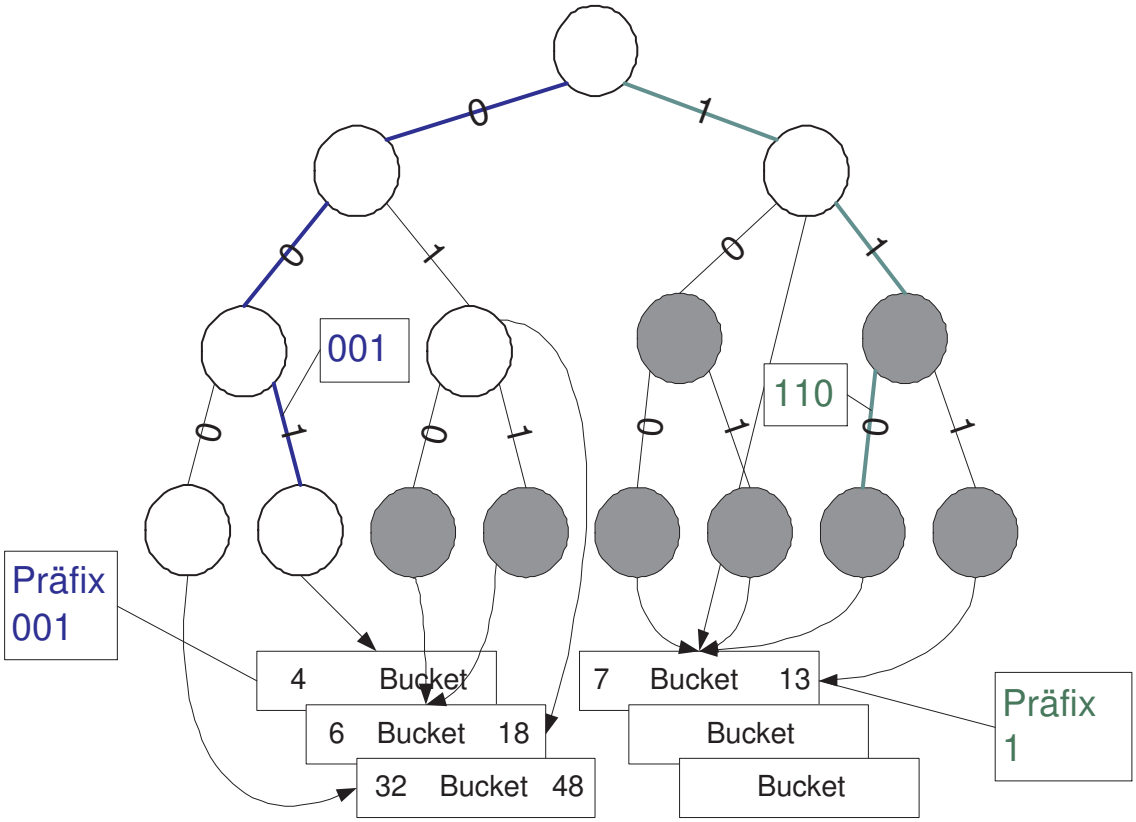



Abbildung 8.12: Illustration des erweiterbaren Hashings

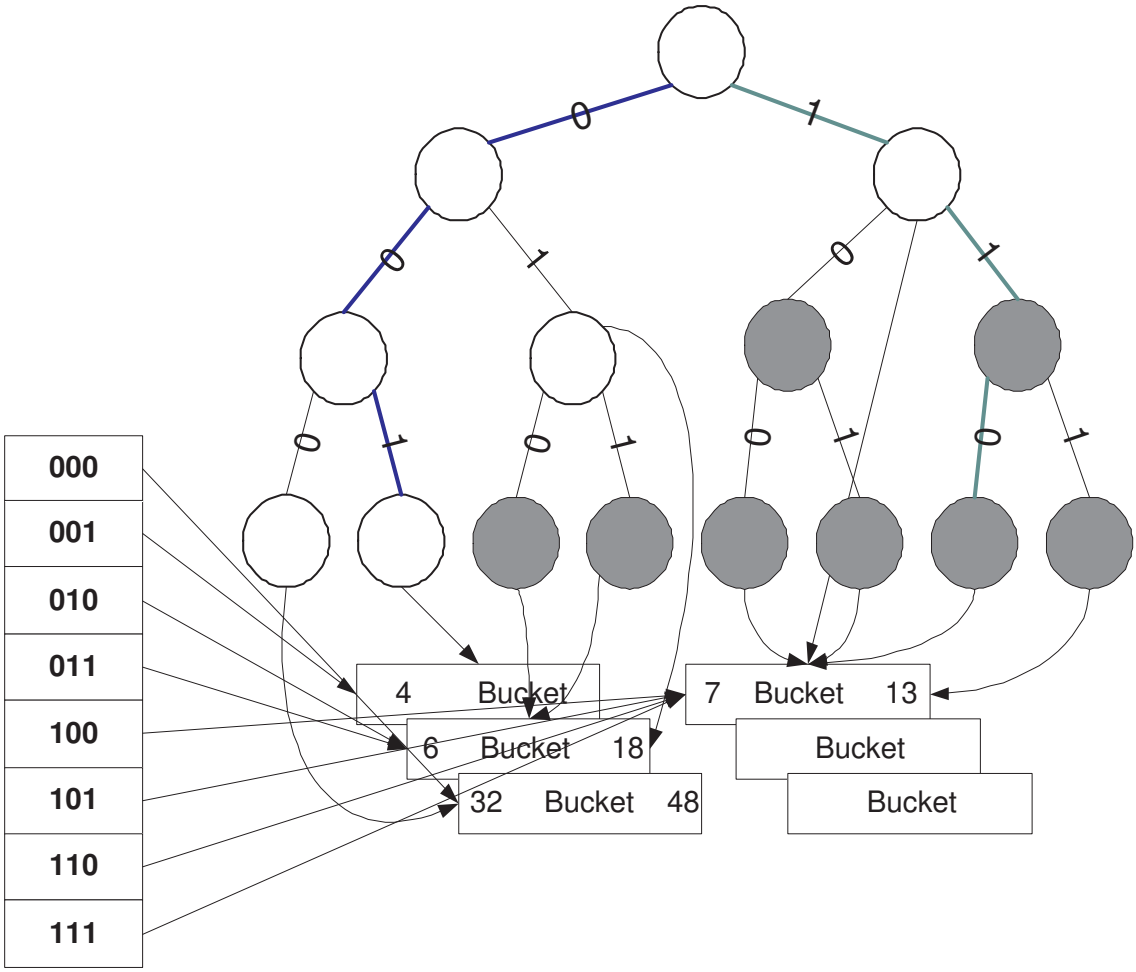



Abbildung 8.13: Illustration des erweiterbaren Hashings

heranzuziehen. Unsere drei Elemnte haben den folgenden Hashcode:

- h(4) = **0010**00..0
- h(12) = **0011**00..0
- h(20) = **0010**10..0

Wenn wir also vier Bitpositionen berücksichtigen, werden die beiden Einträge 4 und 20 in das Bucket mit dem Präfix **0010** und der Eintrag 12 in das Bucket mit dem Präfix **0011** abgebildet. Der hier auftretende Überlauf führt allerdings auch zu einer Verdoppelung des Directory-Arrays, da die lokale Tiefe des übergelaufenen Buchets der globalen Tiefe des Directorys entspricht. Das resultierende verdoppelte Directory sowie die Aufteilung der drei Elmente auf die beiden Buckets ist in Abbildung 8.15 gezeigt.

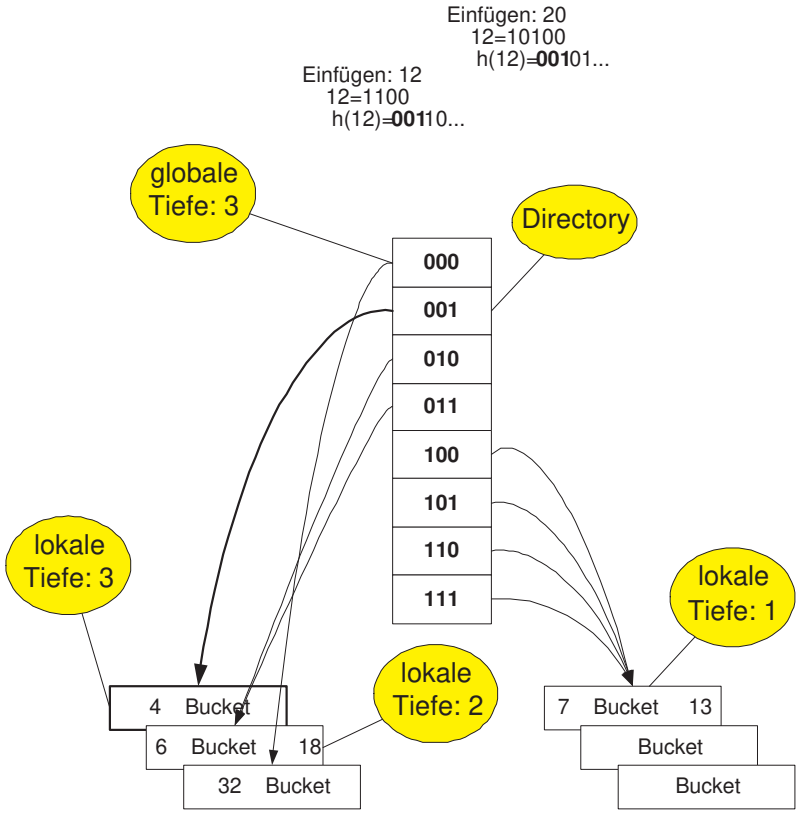



Abbildung 8.14: Illustration des erweiterbaren Hashings: Dirctory als Array und Überlauf des Buckets für den Präfix 001

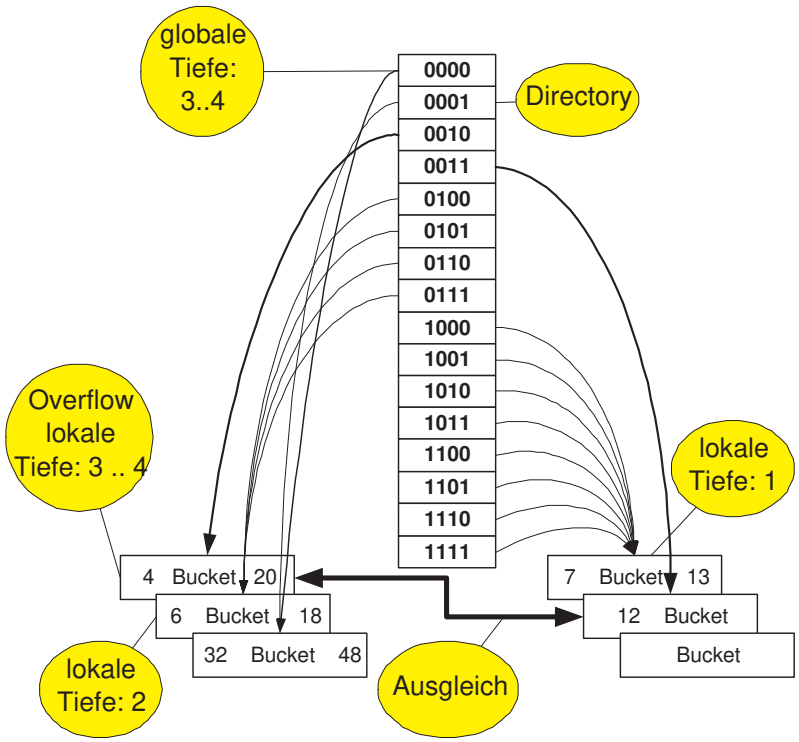



Abbildung 8.15: Erweiterbare Hash-Struktur nach der Verdoppelung des Directorys

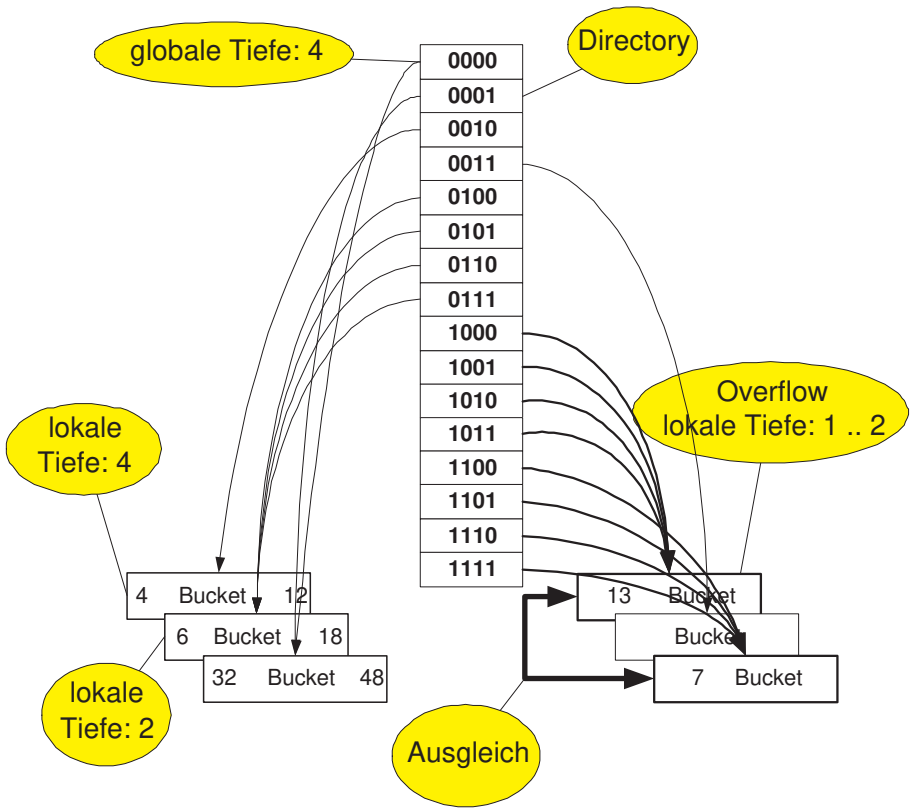



Abbildung 8.16: Überlauf und Ausgleich eines Buckets ohne Directory-Verdoppelung

In Abbildung 8.16 ist abschließend noch der Fall eines Bucket-Überlaufs gezeigt, der einfacher zu handhaben ist, da die lokale Tiefe des Buckets kleiner war als die globale Tiefe des Directorys. Der Überlauf des Buckets rechts oben kann in der Regel dadurch ausgeglichen werden, dass man eine zusätzliche Bitposition mit einbezieht. Dann werden die Elemente als auf die beiden Buckets für die Präfixe **10** und **11** aufgeteilt. Der Eintrag 13 verbleibt in dem ursprünglichen Bucket, das jetzt alle Datensätze mit dem Präfix 10 aufnimmt; der Eintrag 7 wandert in ein neues Bucket, das für den Präfix 11 "zuständig" ist. Die beiden Buckets haben dann jeweils die lokale Tiefe 2 – wohingegen die globale Tiefe des Directorys invariant bei 4 bleibt.

## 8.3 Übungen

**8.1** Betrachten Sie die Hash-Tabelle mit *Double Hashing* als Kollisionsbehandlung (Abbildung 8.5).

Fügen Sie in diese Hashtabelle die Zahl 2 hinzu. Geben Sie an, welche Hashfunktionen verwendet werden und welche Kollisionen auftreten.

# 9. Anwendungsbeispiel: Vom UML-Modell zum Java-Programm

In Abbildung 1.26 ist das konzeptuelle Schema einer Universitätsanwendung als UML-Klassendiagramm gezeigt. Das Diagramm verwendet fast alle Merkmale von Klassendiagrammen, die wir bis jetzt kennen gelernt haben: Klassen mit Attributen und Operationen; Vererbung; eine Aggregation in Form einer Komposition; Beziehungen zwischen den Klassen mit Funktionalitäten, Rollen und definierter Navigierbarkeit.

Das Diagramm wurde in Abschnitt 1.14.7 ausführlich beschrieben. Als nächstes werden wir dieses UML-Diagramm in ein Java Programm umsetzen. Die folgenden Abschnitte beschreiben, wie man die verschiedenen Konzepte in Java umsetzt und was man dabei beachten muss.

In Java ist es Konvention Klassen in der Einzahl zu benennen. Im UML Diagramm verwenden wir hingegen die Mehrzahl um auf die gesamte Extension hinzuweisen.

## 9.1 Angestellter.java

*Professoren* und *Assistenten* ähneln sich in einigen Aspekten: Beide Personengruppen erhalten ein Gehalt und müssen entsprechend auch Steuern zahlen. Sie haben einen Namen und werden von der Verwaltung über Personalnummern identifiziert. Sowohl *Professoren* als auch *Assistenten* sind an einer Universität angestellt und alle Angestellten an einer Universität teilen sich diese Merkmale. Dementsprechend sind diese Attribute und Methoden im Universitätsmodell in einer eigenen Klasse – den *Angestellten* – zusammengefasst. *Professoren* und *Assistenten* sind von dieser gemeinsamen Oberklasse abgeleitet.

Diese Modellierung fördert die Erweiterbarkeit, da leicht andere Gruppen von *Angestellten* hinzugefügt werden können. Auch müsste das System bei Inkrafttreten eines neuen Steuergesetzes nur an einer Stelle angepasst werden (erhöhte Wartbarkeit).

Die Implementierung der Java-Klasse für eine/n *Angestellte/n* enthält Felder für Personalnummer und Name sowie Methoden für die Berechnung eines Grundgehalts und der Steuern abhängig vom Gehalt.

```
1  public class Angestellter {
2    public int persNr;
3    public String name;
4
5    public Angestellter(int persNr, String name) {
6      this.persNr = persNr;
7      this.name = name;
8    }
9
10   public int gehalt() {
11     return 2000;
12   }
13
14   public int steuern() {
15     return gehalt()/2;
16   }
17 }
```

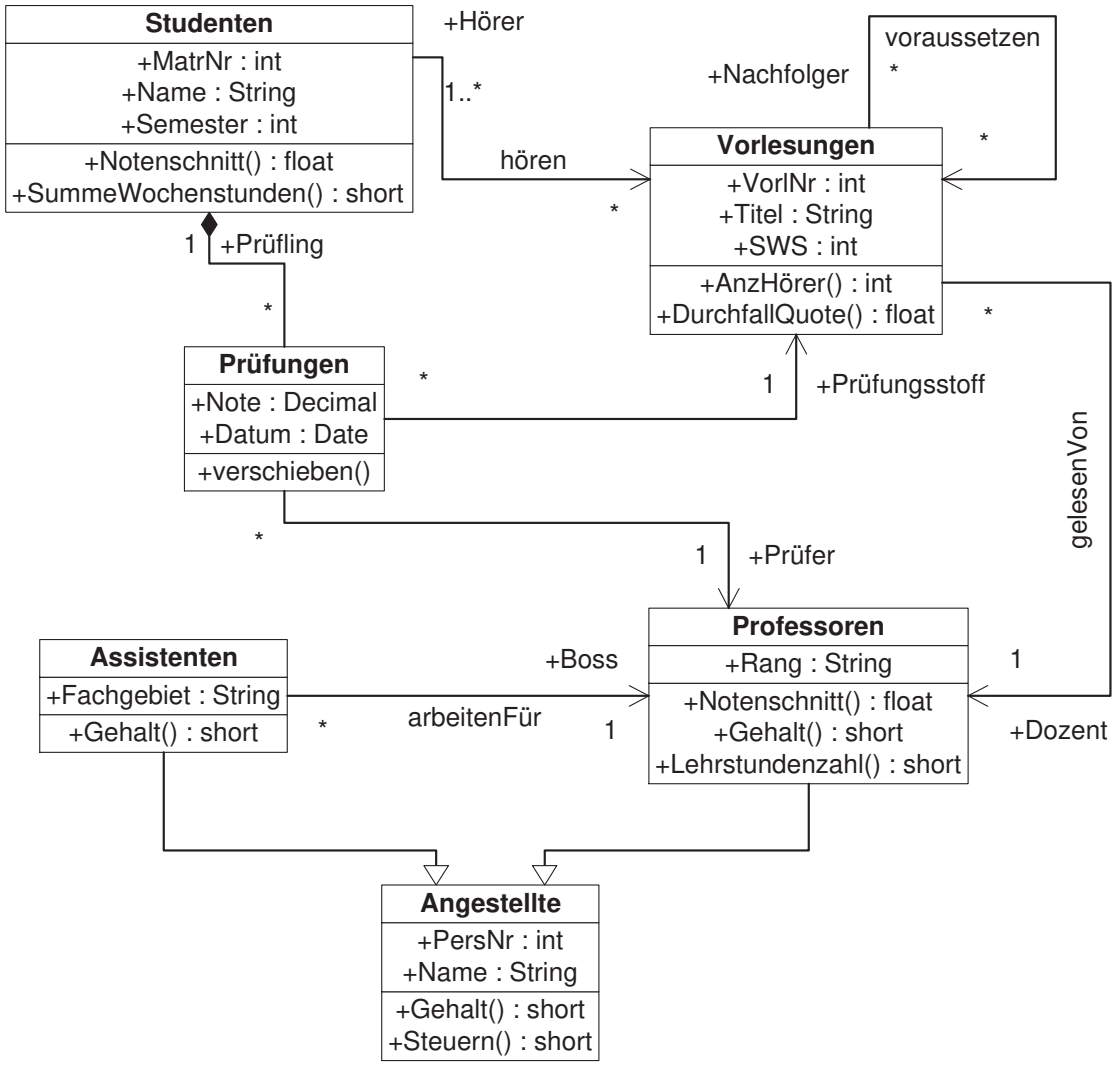



Abbildung 9.1: Die konzeptuelle Modellierung der Universität in UML

## 9.2 Assistent.java

Die Klasse *Assistent* erweitert die Klasse *Angestellter* mit einem Fachgebiet, in dem er/sie tätig ist, und einem *Professor* bzw. einer Professorin als Boss.

Im Diagramm ist die Assoziation *arbeitenFür* als gerichtete Beziehung ausgehend von der Klasse *Assistent* in Richtung der Klasse *Professor* modelliert. Entsprechend ist die Beziehung in unserem Java-Programm als Referenz `boss` in der Klasse *Assistent* realisiert, die auf das zugehörigen *Professor*-Objekt verweist, und nicht umgekehrt.

Der Konstruktor der Klasse *Assistent* ruft den Konstruktor der Oberklasse *Angestellter* mittels `super()` auf und setzt anschließend noch Fachgebiet und Boss. Die Methode `gehalt()` überschreibt die geerbte Implementierung der gleichnamigen Methode aus der Oberklasse.

```
1 public class Assistent extends Angestellter {
2   public String fachgebiet;
```

```
3     public Professor boss;
4
5     public Assistent(int persNr, String name, String fachgebiet, Professor boss) {
6       super(persNr, name);
7       this.fachgebiet = fachgebiet;
8       this.boss = boss;
9     }
10
11    public int gehalt() {
12      return 2500;
13    }
14  }
```

## 9.3 Professor.java

Objekte der Klasse *Professor* haben zusätzlich zu dem von *Angestellten* geerbten Eigenschaften noch einen Rang, der über die Gehaltsstufe entscheidet. Der Rang ist hier als Aufzählungstyp implementiert, da so alle zulässigen Werte des Attributs festgelegt sind – im Unterschied zu einer Modellierung als String-Wert. In Kapitel 5 haben wir Aufzählungstypen schon genauer erläutert.

Zusätzlich besitzt ein *Professor*-Objekt private Felder, die die Anzahl der Lehrstunden einerseits und die Anzahl und Summe der vergebenen Noten andererseits speichern. Anzahl und Summe der Noten werden für die Berechnung des Notenschnitts einer *Professorin* bzw. eines *Professors* in der Methode `notenschnitt()` verwendet und bei jeder neuen *Prüfung* durch die Operation `pruefen()` aktualisiert. Die Lehrstunden werden von der Operation `leseVorlesung()` für jede neu gelesene *Vorlesung* entsprechend erhöht und können mit der Methode `lehrstundenzahl()` abgefragt werden. Die Methode `gehalt()` bestimmt das Gehalt in einer Fallunterscheidung über den Rang.

```
1  public class Professor extends Angestellter {
2    public enum Rang {
3      C1, C2, C3, C4
4    }
5
6    public Rang rang;
7
8    private short lehrstunden;
9    private int notenAnzahl;
10   private int notenSumme;
11
12   public Professor(int persNr, String name, Rang rang) {
13     super(persNr, name);
14     this.rang = rang;
15   }
16
17   public void leseVorlesung(Vorlesung vorlesung) {
18     lehrstunden += vorlesung.sws;
19   }
20
21   public short lehrstundenzahl() {
22     return lehrstunden;
23   }
24
25   public void pruefen(Pruefung pruefung) {
26     notenAnzahl++;
27     notenSumme += pruefung.note;
28   }
29
30   public float notenschnitt() {
31     return (float)notenSumme/notenAnzahl;
32   }
```

```
33
34    public int gehalt() {
35      int gehalt;
36      switch (rang) {
37        case C1: gehalt = 3000;
38        break;
39        case C2: gehalt = 3200;
40        break;
41        case C3: gehalt = 3400;
42        break;
43        case C4: gehalt = 3600;
44        break;
45        default: gehalt = 3000;
46        break;
47      }
48      return gehalt;
49    }
50  }
```

## 9.4 Student.java

Ein Objekt der Klasse *Student* besitzt Attribute für die Matrikelnummer, den Namen und das aktuelle Semester.

Für die Umsetzung der Beziehung *hören* definiert ein die Klasse *Student* noch eine Menge (engl. *Set*) von Referenzen auf die Vorlesungen, die er gerade hört. Das Attribut `vorlesungen` hat den ungewohnten Typ `Set<Vorlesung>`. `Set` ist ein generischer Typ, den man mit verschiedenen Typen verwenden kann. In diesem Fall ist es ein `Set` von *Vorlesungen*. Generische Typen wurden in Kapitel 6 genauer erläutert. Die Umsetzung der Beziehung *hören* in der Klasse *Student* entspricht der im Diagramm definierten Navigierbarkeit der Assoziation ausgehend von *Student* nach *Vorlesung*. Die Komposition der *Prüfungen* wurde ebenfalls mit dem generischen Typ `Set` umgesetzt, diesmal wird `Set` aber mit dem Typ *Prüfung* verwendet.

Der Konstruktor setzt die Attribute und erzeugt zwei neue Mengen für die Assoziation *hören* und die Komposition von *Prüfungen*. Die Methode `belegeVorlesung()` fügt die *Vorlesung* zur Menge der von diesem *Student* bzw. dieser *Studentin* gehörten *Vorlesungen* hinzu und erhöht bei der *Vorlesung* die Anzahl der Hörer. Wenn ein *Student* bzw. eine *Studentin* geprüft wird, fügt die Methode `prüfen()` die *Prüfung* zur Menge der abgelegten *Prüfungen* hinzu.

Der Notenschnitt und die Summe der Semesterwochenstunden lassen sich über die Methoden `notenschnitt()` respektive `summeWochenstunden()` abfragen. Die Methode `notenschnitt()` iteriert dafür in einer so genannten *for-each Loop* über alle *Prüfungen*. Bei einer *for-each Loop* gibt man vor dem Doppelpunkt eine lokale Variable an, die nacheinander alle Elemente der Kollektion annimmt, die nach dem Doppelpunkt angegeben ist. `summeWochenstunden()` verwendet für die gleiche Aufgabe einen Iterator um in diesem Fall die Menge der *Vorlesungen* aufzuzählen. Iteratoren haben wir schon in Abschnitt 6.3.4 kennen gelernt.

```
1  import java.util.Set;
2  import java.util.HashSet;
3  import java.util.Iterator;
4
5  public class Student {
6    public int matrNr;
7    public String name;
8    public int semester;
```

```
9     public Set<Vorlesung> vorlesungen;
10    public Set<Pruefung> pruefungen;
11
12    public Student(int matrNr, String name, int semester) {
13      this.matrNr = matrNr;
14      this.name = name;
15      this.semester = semester;
16
17      vorlesungen = new HashSet<Vorlesung>();
18      pruefungen = new HashSet<Pruefung>();
19    }
20
21    public void belegeVorlesung(Vorlesung vorlesung) {
22      if (!vorlesungen.contains(vorlesung)) {
23        vorlesungen.add(vorlesung);
24        vorlesung.erhoeheAnzahlHoerer();
25      }
26    }
27
28    public void pruefen(Pruefung pruefung) {
29      pruefungen.add(pruefung);
30    }
31
32    public float notenschnitt() {
33      float durchschnittsNote = 0;
34      for (Pruefung p : pruefungen) {
35        durchschnittsNote += p.note/pruefungen.size();
36      }
37      return durchschnittsNote;
38    }
39
40    public short summeWochenstunden() {
41      short summeSWS = 0;
42      Iterator<Vorlesung> it = vorlesungen.iterator();
43      while (it.hasNext()) {
44        summeSWS += it.next().sws;
45      }
46      return summeSWS;
47    }
48  }
```

## 9.5 Pruefung.java

Ein *Prüfungs*-Objekt besteht aus einer Note und einen Prüfungstermin, für den wir den vordefinierten Java-Objekttyp `Calendar` verwenden. Zusätzlich verweisen drei Referenzen auf die geprüfte *Studentin* bzw. den geprüften *Studenten*, die geprüfte *Vorlesung* und die prüfende *Professorin* bzw. den prüfenden *Professor* – wiederum in Übereinstimmung mit den im Diagramm definierten gerichteten Beziehungen.

Der Konstruktor einer *Prüfung* setzt alle Instanzvariablen entsprechend den übergebenen Parametern und das boolesche Flag `bewertet` auf falsch.

Ein Prüfungs-Objekt bekommt mit der Methode `bewerten()` eine Note zugeordnet, falls sie noch nicht bewertet wurde. Dies wird anhand des Flags `bewertet` entschieden. Anschließend wird bei allen Beteiligten die Methode `pruefen()` aufgerufen, damit diese ihren Status entsprechend anpassen können. Beispielsweise ändert sich bei dem betroffenen Studenten-Objekt durch eine bewertete Prüfung der Notenschnitt.

Die Klasse Prüfung bietet zusätzlich die Methode `verschieben()` an, über die der Prüfungstermin verschoben werden kann, falls die Prüfung noch nicht stattgefunden hat.

```
1  import java.util.Calendar;
2
3  public class Pruefung {
```

```
4     public double note;
5     public Calendar datum;
6     public Student pruefling;
7     public Vorlesung pruefungsstoff;
8     public Professor pruefer;
9
10    private boolean bewertet;
11
12    public Pruefung(Student student, Vorlesung vorlesung, Professor professor,
13                    Calendar termin) {
14      this.pruefling = student;
15      this.pruefungsstoff = vorlesung;
16      this.pruefer = professor;
17      this.datum = termin;
18      bewertet = false;
19    }
20
21    public void bewerten(double note) {
22      if (!bewertet) {
23        bewertet = true;
24        this.note = note;
25        pruefling.pruefen(this);
26        pruefungsstoff.pruefen(this);
27        pruefer.pruefen(this);
28      }
29    }
30
31    public void verschieben(Calendar neuesDatum) {
32      if (datum.compareTo(Calendar.getInstance()) > 0) {
33        datum = neuesDatum;
34      }
35    }
36  }
```

## 9.6 Vorlesung.java

Die Klasse *Vorlesung* spezifiziert Felder für die Vorlesungsnummer, den Titel, die Semesterwochenstunden, die lesende *Professorin* bzw. den lesenden *Professor* und die Vorgänger-Vorlesungen. Zusätzlich hat sie private Felder für die Anzahl der Hörer, Prüfungen und durchgefallenen *Studenten*.

Der Konstruktor setzt die Attribute des neu erstellten Objekts entsprechend den übergebenen Parametern. Zusätzlich wird eine neue Menge für die Vorraussetzungen erstellt und beim Dozent die Methode `leseVorlesung()` aufgerufen.

Die Methode `pruefen()` erhöht die Anzahl der zu dieser *Vorlesung* gehörenden *Prüfungen* und die Anzahl der durchgefallenen *Studenten*, falls die Note schlechter als 4.0 ist. Die Operation `erhoeheAnzahlHoerer()` realisiert genau das, was ihr Name beschreibt. `anzahlHoerer()` liefert die Anzahl der *Studenten* zurück, die diese *Vorlesung* hören. `durchfallQuote()` bestimmt anhand der Anzahl der *Prüfungen* und der durchgefallenen *Studenten* die Durchfallquote der *Vorlesung*.

```
1  import java.util.Set;
2  import java.util.HashSet;
3
4  public class Vorlesung {
5    public int vorlNr;
6    public String titel;
7    public int sws;
8    public Professor dozent;
9    public Set<Vorlesung> voraussetzungen;
10
11   private int anzahlHoerer;
```

```
12    private int anzahlPruefungen;
13    private int anzahlDurchgefallen;
14
15    public Vorlesung(int vorlNr, String titel, int sws, Professor dozent) {
16      this.vorlNr = vorlNr;
17      this.titel = titel;
18      this.sws = sws;
19      this.dozent = dozent;
20
21      voraussetzungen = new HashSet<Vorlesung>();
22
23      dozent.leseVorlesung(this);
24    }
25
26    public void pruefen(Pruefung pruefung) {
27      if (pruefung.note > 4.0) {
28        anzahlDurchgefallen++;
29      }
30      anzahlPruefungen++;
31    }
32
33    public void erhoeheAnzahlHoerer() {
34      anzahlHoerer++;
35    }
36
37    public int anzahlHoerer() {
38      return anzahlHoerer;
39    }
40
41    public float durchfallQuote() {
42      return (float)anzahlDurchgefallen/anzahlPruefungen;
43    }
44  }
```

# Literaturverzeichnis


Arnold, K. und J. Gosling (1998). *The Java Programming Language*. Addison-Wesley, Reading, MA, USA.

Bobrow, D. G., K. Kahn, G. Kiczales, L. Masinter, M. J. Stefik und F. Zdybel (1990). *CommonLoops: Merging Lisp and Object-Oriented Programming*. In: Cardenas, A. und D. McLeod, Hrsg.: *Research Foundations in Object-Oriented and Semantic Database Systems*, S. 70–90, Englewood Cliffs, NJ, USA. Prentice Hall.

Bobrow, D. G. und T. Winograd (1977). *An Overview of KRL, a Knowledge Representation Language*. Cognitive Science, 1(1):3–46.

Bobrow, D. G. und T. Winograd (1979). *KRL, another Perspective*. Cognitive Science, 3(1).

Booch, G. (1983). *Software Engineering with Ada*. Benjamin/Cummings, Menlo Park, CA.

Booch, G. (1991). *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA, USA.

Breazu-Tannen, V., P. Buneman und A. Ohori (1989). *Can Object-Oriented Databases be Statically Typed?*. In: *Database Programming Languges Workshop*, S. 226–237, Portland, OR.

Bruce, K. B. und P. Wegner (1990). *An Algebraic Model of Subtype and Inheritance*. In: Bancilhon, F. und P. Buneman, Hrsg.: *Advances in Database Programming Languages*, S. 75–96. ACM Press, Addison Wesley.

Brügge, B. und A. H. Dutoit (2004). *Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java*. Pearson Studium, München.

Budd, T. (1991). *An Introduction to Object-Oriented Programming*. Addison-Wesley, Reading, MA, USA.

Butterworth, P., A. Otis und J. Stein (1991). *The GemStone Object Database System*. Communications of the ACM, 34(10):64–77.

Cardelli, L. (1988). *Types for Data-Oriented Languages*. In: *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, Bd. 303 d. Reihe *Lecture Notes in Computer Science (LNCS)*, S. 1–15, New York, Berlin, etc. Springer-Verlag.

Cardelli, L. und P. Wegner (1985). *On Understanding Types, Data Abstraction, and Polymorphism*. ACM Computing Surveys, 17(4):471–522.

Coad, P. und E. Yourdan (1991a). *Object-Oriented Analysis*. Prentice Hall, Englewood Cliffs, NJ, USA, 2. Auflage

Coad, P. und E. Yourdan (1991b). *Object-Oriented Design*. Prentice Hall, Englewood Cliffs, NJ, USA, 2. Auflage

Comer, D. (1979). *The ubiquitous B-tree*. ACM Computing Surveys, 11(2):121–137.

Cox, B. J. (1986). *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, MA, USA.

Dahl, O. und K. Nygaard (1966). *Simula, an Algol-Based Simulation Language*. Communications of the ACM, 9:671–678.

Dahl, O. J., B. Myrhaug und K. Nygaard (1970). *Simula 67: Common Base Language*. Publication NS 22, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway.

Danforth, S. und C. Tomlinson (1988). *Type Theories and Object Oriented Programming*. ACM Computing Surveys, 20(1):29–72.

Dayal, U., F. Manola, A. Buchmann, U. Chakravarthy, S. Heiler, J. Orenstein und A. Rosenthal (1987). *Simplifying Complex Objects: The PROBE Approach to Modeling and Querying Them*. In: Schek, H. J. und G. Schlageter, Hrsg.: *Informatik Fachberichte No. 136*, S. 17–38, Berlin. Springer-Verlag.

Deitel und Deitel (1999). *Java, How to Program*. Prentice Hall, Englewood Cliffs, NJ, USA, 3. Auflage

Derret, N., W. Kent und P. Lyngbaek (1985). *Some Aspects of Operations in an Object-Oriented Database*. IEEE Database Engineering, 8(4):66–74.

Derret, N. P., D. H. Fishman, W. Kent, P. Lyngbaek und T. A. Ryan (1986). *An Object-Oriented Approach to Data Management*. In: *Proc. COMPCON*, S. 330–335.

Gamma, E., R. Helm, R. Johnson und J. Vlissides (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA.

Goldberg, A. und D. Robson (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, USA.

Grady Booch, et al (1998). *Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, USA.

Institut, American National Standards (1983). *The Programming Language Ada Reference Manual*. Lecture Notes in Computer Science No. 155, Springer-Verlag.

Keene, S. E. (1989). *Object-oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA.

Kemper, A. (1992). *Zuverlässigkeit und Leistungsfähigkeit objektorientierter Datenbanken*, Bd. 298 d. Reihe *Informatik Fachberichte*. Springer-Verlag, New York, Berlin, etc.

Kemper, A. und G. Moerkotte (1989). *Typing in Object Bases*. In: *Proc. Advanced Database Symposium*, S. 19–31, Kyoto, Japan.

Kemper, A. und G. Moerkotte (1990). *Correcting Anomalies of Standard Inheritance—A Constraint Based Approach*. In: *Proc. of the Intl. Conf. on Database and Expert Systems Applications (DEXA)*, S. 49–55, Vienna, Austria. Springer-Verlag.

Kemper, A. und G. Moerkotte (1992). *A Framework and a Type Inference System for Strong Typing in (Persistent) Object Models*. In: *Proc. of the Intl. Conf. on Database and Expert Systems Applications (DEXA)*, S. 257–263.

Kemper, A. und G. Moerkotte (1994). *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice Hall, Englewood Cliffs, NJ, USA.

Kernighan, B. W. und D. M. Ritchie (1988). *The C Programming Language*. Prentice Hall, 2 Auflage

Kifer, M. und G. Lausen (1989). *F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme*. In: *Proc. of the ACM SIGMOD Conf. on Management of Data*, S. 134–146, Portland, OR, USA.

Knuth, D. (1973). *The Art of Computer Programming – Sorting and Searching*, Bd. 3. Addison-Wesley, Reading, MA, USA.

Liskov, B., R. Atkinson, T. Bloom, E. Moss, J. G. Schaffert, R. Scheifler und A. Snyder (1981). *CLU Reference Manual*, Bd. 114 d. Reihe *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, New York, Berlin, etc.

Liskov, B. und J. Guttag (1986). *Abstraction and Specification in Program Development*. The MIT Electrical Engineering and Computer Science Series. The MIT Press, Cambridge, MA, USA.

Liskov, B. und J. Guttag (2000). *Programm Development in Java*. Addison-Wesley, Reading, MA, USA.

Liskov, B. und A. Snyder (1979). *Exception Handling in CLU*. IEEE Trans. Software Eng., 5(6):546–558.

Mandrioli, D. und B. Meyer, Hrsg. (1992). *Advances in Object-Oriented Software Engineering*. Object-Oriented Series. Prentice Hall, Englewood Cliffs, NJ, USA.

Meyer, B. (1986). *Genericity versus Inheritance*. In: *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, S. 391–405. ACM SIGPLAN Notices, Vol. 21, No. 11.

Meyer, B. (1987). *Reusability: The Case for Object-Oriented Design*. IEEE Software, 4(2):43–53.

Meyer, B. (1988). *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, USA.

Meyer, B. (1992). *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, Englewood Cliffs, NJ, USA.

Micallef, J. (1988). *Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages*. Journal of Object-Oriented Programming Languages, 1(1):12–36.

Milner, R. (1978). *A Theory of Type Polymorphism*. Journal of Computer and System Sciences, 17:348–375.

Moon, D. A. (1986). *Object-Oriented Programming with Flavors*. In: *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, S. 1–8.

Nygaard, K. und O. J. Dahl (1981). *Simula 67*. In: Wexelblat, R. W., Hrsg.: *History of Programming Languages*.

Oestereich, B. (1999). *Objektorientierte Softwareentwicklung*. R. Oldenbourg.

Ohori, A. (1988). *Semantics of Types for Database Objects*. In: *Proc. of the Intl. Conf. on Database Theory (ICDT)*, S. 239–251, Bruges, Belgium.

Ohori, A., P. Buneman und V. Breazu-Tannen (1989). *Database Programming in Machiavelli: A Polymorphic Language with Static Type Inference*. In: *Proc. of the ACM SIGMOD Conf. on Management of Data*, Portland, OR.

Ontologic (1987). *Vbase Integrated Object System: System Documentation*. Ontologic Corp., The Structure of Possibility, Billerica, MA 01821, USA.

Shipman, D. (1981). *The Functional Data Model and the Data Language DAPLEX*. ACM Trans. on Database Systems, 6(1):140–173.

Snyder, A. (1986). *Encapsulation and Inheritance in Object-Oriented Programming Languages*. In: *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, S. 38–45.

Stefik, M. und D. G. Bobrow (1986). *Object-Oriented Programming: Themes and Variations*. The AI Magazine, 6(4):40–62.

Stein, L. A., H. Lieberman und D. Ungar (1989). *A Shared View of Nothing: TheTreaty of Orlando*. In: Kim, W. und F. H. Lochovsky, Hrsg.: *Object-Oriented Concepts, Databases, and Applications*, S. 31–48. Addison-Wesley.

Stroustrup, B. (1990). *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, 2. Auflage

Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, 3. Auflage