Grundlagen der Informatik

3., aktualisierte Auflage

Helmut Herold
Bruno Lurz
Jürgen Wohlrab
Matthias Hopf

Pearson

EXTRAS ONLINE

Grundlagen der Informatik

3., aktualisierte Auflage

Helmut Herold
Bruno Lurz
Jürgen Wohlrab
Matthias Hopf

Pearson

direkten Zugriff auf alle Kassen und Schlüssel der Schule ermöglichen, da dies zwangsläufig im Chaos enden würde.

Schritt 3: Wiederverwendung bereits vorhandener Software

Wie bereits angesprochen, werden in der objektorientierten Programmierung Objekte aufgrund vorliegender Klassendefinitionen gebildet. Somit besteht die Möglichkeit, Klassendefinitionen für oft benötigte Sachverhalte in so genannten „Klassenbibliotheken" zu hinterlegen. Durch einfaches Übernehmen dieser Definitionen (und eventuellem Hinzufügen neuer Teile mit Hilfe des Vererbungsmechanismus) wird die Softwareentwicklung stark vereinfacht. Man spricht auch von Softwareentwicklung durch Reproduktion (Nachbildung) von Objektbeschreibungen (Klassendefinitionen).

► Übung: Welche der folgenden Aussagen sind richtig?

1. Ein Objekt ist ein Exemplar einer Klasse.
2. Eine Klasse ist ein Objekt.
3. Bei der Vererbung erbt die übergeordnete Klasse alle Eigenschaften der Unterklasse.
4. Ein Objekt ist eine Klasse.
5. Eine Unterklasse erbt nur bestimmte, frei wählbare Eigenschaften von ihrer Oberklasse.
6. Ein Objekt ist eine Instanz einer Klasse.
7. Eine Unterklasse ist eine Klasse, die durch Vererbung aus einer anderen Klasse entsteht.
8. Das Neue an der Objektorientierung ist die klare Trennung zwischen Daten und Funktionen, so dass beliebige Funktionen jederzeit auf globale Daten zugreifen können.
9. Eine Unterklasse erbt alle Eigenschaften von ihrer Oberklasse.
10. Eine Klasse ist ein Exemplar eines Objekts.
11. Die Objektorientierung ermöglicht die Wiederverwendbarkeit von Software.

► Übung: Klassendiagramme

In welchem Zusammenhang stehen jeweils die folgenden Begriffe (Klasse, Objekt, Unterklasse)? Zeichnen Sie hierzu korrespondierende Modelle in UML-Notation!

Beispiel: Objektorientierte Sprache, PASCAL, Gesprochene Sprache, Sprache, Java, Englisch, Latein, Computersprache, Altgriechisch, C, Prozedurale Sprache, C++, Ausgestorbene Sprache, Deutsch

Abbildung 7.60 zeigt die Lösung zu dieser Aufgabenstellung.

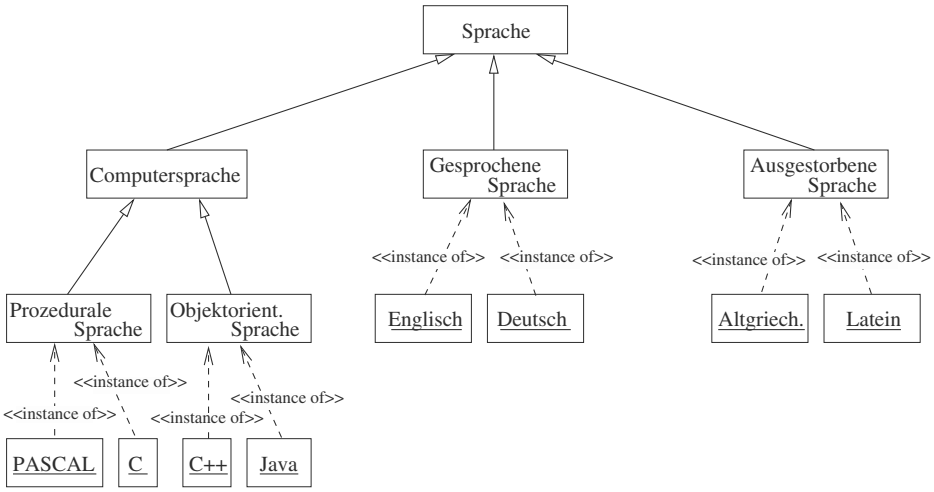

Abbildung 7.60: Objektorientiertes Modell zur Aufgabenstellung

Nachfolgend nun einige Übungen für Sie. Zeichnen Sie dazu das jeweils korrespondierende Modell (in UML-Notation) zu den nachfolgenden Begriffen!

Tiere mit und ohne Beine: Bello, Beinlos, Katze, „Ringel, meine Schlange“, Zweibeiner, Hund, Fisch, Mensch, Lebewesen, Vierbeiner, „Mein Haustier“, Schlange, Hai, Vogel, Rex, Mausi, „Mein Kater“.

Die C-Datentypen: Datentyp, Ganzzahl, Gleitpunktzahl, short, int, alphanumerisch, numerisch, char, float, char zeichen, double, int zaehler, float pi

Unterschiedliche Menschen: Adenauer, Seefahrer, Bundeskanzler, „Hans Meier“, „H. Hesse“, Pirat, Mensch, Nobelpreisträger, Politiker, Physik, Nixon, „A. Einstein“, Literatur, Lincoln, Sindbad, amerik. Präsident

7.6.3 Klassen und Objekte

In Kapitel 7.5.19 auf Seite 229 wurden Strukturen in C/C++ vorgestellt. Nachfolgend wird anhand dieser Strukturen das mangelnde Schutzkonzept in prozeduralen Sprachen wie C/C++ verdeutlicht. Definiert man in C/C++ Strukturvariablen wie in Listing 7.7, so kann auf diese Variablen und deren Komponenten jede Funktion zugreifen.

Listing 7.7: Programm cstruct.c: Unkontrollierter Zugriff auf Strukturen in C

```
#include <stdio.h>
#include <string.h>
 struct konto {
    int      nr;
    double stand;
    int      pin;
};
```

```
void legeKontoAn(int n, int p, struct konto *k) {
    k->nr   = n; /* Zugriff auf die Komponente über den Strukturzeiger k */
    k->stand = 0;
    k->pin = p;
}
void zahleEin(struct konto *k, double betrag) { k->stand += betrag; }
void hebeAb(struct konto *k, double betrag)  {  k->stand -= betrag; }

void ueberweise(struct konto *von, struct konto *nach, double betrag) {
    hebeAb(von, betrag);
    zahleEin(nach, betrag);
}
int  main(void) {
  struct konto  eins,  zwei;
    legeKontoAn(11111, 4711, &eins);
    zahleEin(&eins, 1000);
    printf("Konto %d: %.2lf\n", eins.nr, eins.stand);
    legeKontoAn(22222, 9999, &zwei);
    zahleEin(&zwei, 500);
    printf("Konto %d: %.2lf\n", zwei.nr, zwei.stand);
    printf("------------------------------ 300: eins --> zwei\n");
    ueberweise(&eins, &zwei, 300);
    printf("Konto %d: %.2lf\n", eins.nr, eins.stand);
    printf("Konto %d: %.2lf\n", zwei.nr, zwei.stand);
    printf("------------------------------ 150: von eins weg\n");
    hebeAb(&eins, 150); /* unkontrollierter Zugriff möglich */
    printf("Konto %d: %.2lf\n", eins.nr, eins.stand);
    printf("------------------------------ 200: zwei --> eins\n");
    ueberweise(&zwei, &eins, 200); /* unkontrollierter Zugriff möglich */
    printf("Konto %d: %.2lf\n", eins.nr, eins.stand);
    printf("Konto %d: %.2lf\n", zwei.nr, zwei.stand);
    /* ....  keinerlei  Schutz; sogar Zugriff auf Geheimnummer möglich */
    printf("------------------------------ Die GEHEIMNUMMERN\n");
    printf("Pin von Konto %d: %d\n", eins.nr, eins.pin);
    printf("Pin von Konto %d: %d\n", zwei.nr, zwei.pin);
    return 0;
}
```

Das Programm 7.7 liefert die folgende Ausgabe:

```
Konto 11111: 1000.00
Konto 22222: 500.00
------------------------- 300: eins --> zwei
Konto 11111: 700.00
Konto 22222: 800.00
------------------------- 150: von eins weg
Konto 11111: 550.00
------------------------- 200: zwei --> eins
Konto 11111: 750.00
Konto 22222: 600.00
------------------------- Die GEHEIMNUMMERN
Pin von Konto 11111: 4711
Pin von Konto 22222: 9999
```

In Programm 7.7 ist erkennbar, dass jede beliebige Funktion die Inhalte der beiden Strukturvariablen `eins` und `zwei` lesen und überschreiben kann, wie es in Abbildung 7.61 gezeigt ist. Dass ein solcher unkontrollierter Zugriff auf Konten in der realen Welt im Chaos enden würde, versteht sich wohl von selbst. Auch in der Softwareentwicklung sind unkontrollierte Zugriffe nicht ganz ungefährlich, da bei fehlerhaften Kontoständen nicht mehr nachvollziehbar ist, welche Funktion hierfür verantwortlich war.

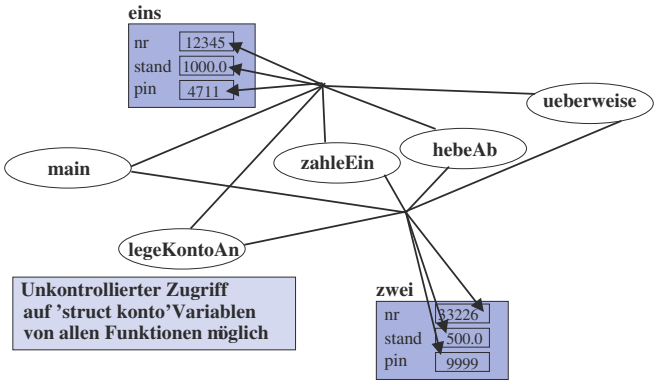

Abbildung 7.61: Unkontrollierter Zugriff in C auf Strukturen durch beliebige Funktionen

Um solche unkontrollierten Zugriffe zu vermeiden, bieten die objektorientierten Programmiersprachen Java und C++ zusätzlich zu Strukturen *Klassen* an, über die die prozedurale Programmiersprache C nicht verfügt. Entscheidend bei Klassen ist Folgendes:

Daten und Methoden (Memberfunktionen) in Klassen
: In Java lassen sich Funktionen (so genannte *Methoden*) einer Klasse zuordnen. Eine *Methode* ist ebenso wie die Daten ein Element (Bestandteil) einer Klasse. So ist sofort zu erkennen, welche Funktionen (Methoden) welche Daten bearbeiten.

Schutzmechanismen
: Um einen uneingeschränkten Zugriff auf Daten und/oder Methoden in einer Klasse zu vermeiden, führte man in Java die gleichen Schlüsselwörter wie in C++ ein, mit denen ein unterschiedlicher Zugriffsschutz für die einzelnen Elemente von Klassen festgelegt werden kann:

`public` – Zugriff von überall möglich
: Auf Daten und Methoden, die diesem Schutztyp zugeordnet sind, kann von überall aus zugegriffen werden, also auch von Funktionen, die keine Methoden dieser Klasse sind.

`private` – Zugriff nur von innerhalb der Klasse möglich
: Auf die diesem Schutztyp zugeordneten Daten und/oder Methoden kann nur von den Methoden zugegriffen werden, die innerhalb der Klasse definiert sind. Nach außen sind diese Daten und/oder Methoden unsichtbar.

Listing 7.8: Java-Programm `Konto.java` mit Schutzmechanismen

```
import java.io.*;
public class Konto
{
    // .....   private Membervariablen (von ausserhalb nicht zugreifbar)
    private int        nr, pin;
    private double     stand;
    private Eingabe    ein;

    // .....   private Methoden (von ausserhalb nicht aufrufbar)
    private void kontoStand() { System.out.println("; Neuer Kontostand: " + stand); }
    private boolean liesGeheimNr() {
        if (ein.readInt("Geheimnummer von Kto " + nr + ": ") != pin) {
            System.out.println(" ..... Falsche Geheimzahl");
            return false;
        }
        return true;
    }
    // .....   public Methoden (von ausserhalb aufrufbar)
    public void initKonto(int n, int p) {
        nr    = n;
        stand = 0;
        pin   = p;
        ein   = new Eingabe();
    }
    public void zahleEin(double betrag) {
        stand += betrag;
        System.out.print("Kto " + nr + ":  +" + betrag);
        kontoStand();
    }
    public boolean hebeAb(double betrag) {
        if (liesGeheimNr()) {
            stand -= betrag;
            System.out.print("Kto " + nr + ":  -" + betrag);
            kontoStand();
            return true;
        }
        return false;
    }
    public void ueberweise(Konto nach, double betrag) {
        if (hebeAb(betrag))
            nach.zahleEin(betrag);
    }
    public void getKontostand() { System.out.println("Kontostand von "+nr+": "+stand); }
```

Listing 7.9 zeigt die `main()`-Methode, die zwei `Konto`-Objekte `eins` und `zwei` anlegt, und dann die von der Klasse `Konto` angebotenen `public`-Methoden aufruft.

Listing 7.9: Java-Programm `KontoMain.java`

```
public class KontoMain {
   public static void main (String args[]) {
      Konto eins = new Konto(),
            zwei = new Konto();
      eins.initKonto(11111, 4711);
      zwei.initKonto(22222, 9999);
      eins.zahleEin(700);
      zwei.zahleEin(500);
      eins.ueberweise(zwei, 300);
      zwei.hebeAb(150);
      eins.getKontostand();
   }
}
```

Möglicher Ablauf des Programms 7.9 (`KontoMain.java`):

```
Kto 11111: +700.0; Neuer Kontostand: 700.0
Kto 22222: +500.0; Neuer Kontostand: 500.0
Geheimnummer von Kto 11111: 4711
Kto 11111: −300.0; Neuer Kontostand: 400.0
Kto 22222: +300.0; Neuer Kontostand: 800.0
Geheimnummer von Kto 22222: 9999
Kto 22222: −150.0; Neuer Kontostand: 650.0
Kontostand von 11111: 400.0
```

Abbildung 7.62 verdeutlicht nochmals, dass

- von Funktionen außerhalb der Klasse `Konto` im Programm 7.8 nur auf die Methoden `initKonto()`, `zahleEin()` usw. zugegriffen werden kann,
- nicht jedoch auf die Membervariablen `nr`, `stand`, `pin`, `ein` und auch nicht auf die privaten Methoden `kontoStand()` und `liesGeheimNr()`. Innerhalb der Klasse kann jedoch auf diese Membervariablen ebenso zugegriffen werden wie auf die beiden privaten Methoden `kontoStand()` und `liesGeheimNr()`.

Der Vorteil hier ist, dass die Membervariablen nicht unkontrolliert von außerhalb der Klasse, sondern nur noch über die nach außen sichtbaren `public`-Methoden verändert werden können.

Abbildung 7.63 zeigt ein UML-Klassendiagramm zur Klasse `Konto`.

Objekte werden in UML-Diagrammen ähnlich wie Klassen dargestellt, nur dass der Objektname unterstrichen wird. Für die Attribute können beispielhaft Werte eingesetzt werden, wie dies in Abbildung 7.64 veranschaulicht ist. Auch bei Objekten ist nur der unterstrichene Objektname immer anzugeben, während die Attribute und ihre Werte wieder weggelassen werden können, wenn diese für den jeweiligen Anwendungsfall nicht von Interesse sind. Abbildung 7.64 zeigt mögliche Notationen für Objekte (am Beispiel der Klasse `Konto`).

Um einen Zusammenhang zwischen Objekten und Klassen in UML zu modellieren, wird von den Objekten ein gestrichelter Pfeil in Richtung der jeweiligen Klasse, zu der sie gehören, gezeichnet. Dieser Pfeil wird dann mit `<<instance of>>` beschriftet, wie dies in Abbildung 7.65 gezeigt ist.

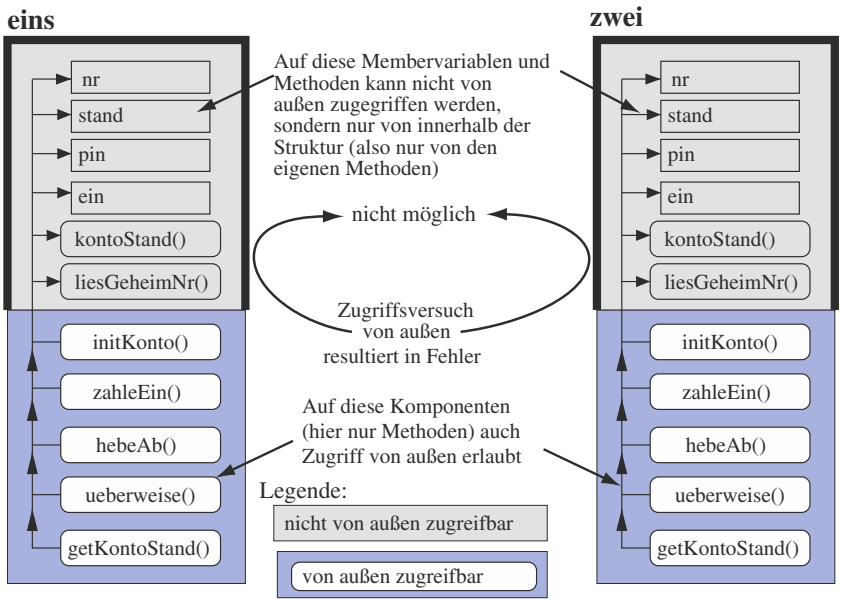

Abbildung 7.62: Zugriffsschutz für bestimmte Komponenten der Klasse `Konto`

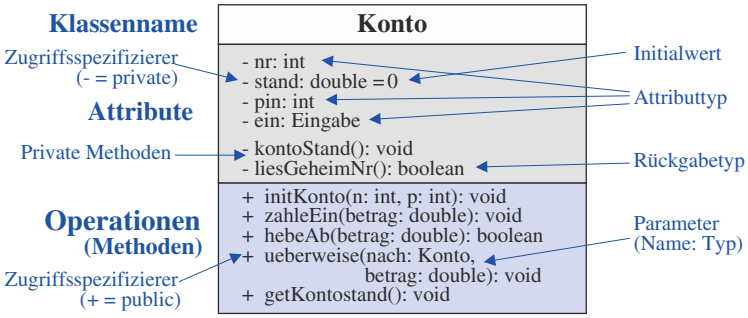

Abbildung 7.63: Klassen in UML (am Beispiel der Klasse `Konto`)

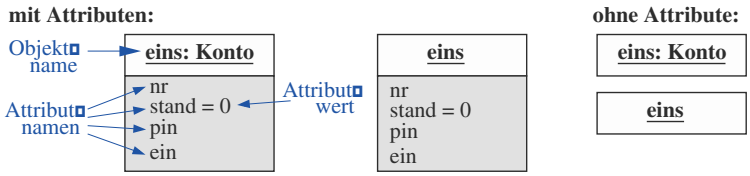

Abbildung 7.64: Notationsmöglichkeiten für Objekte in UML

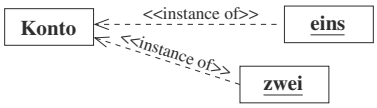

Abbildung 7.65: Klasse-Objekt-Beziehung in UML

Objekte als Instanz einer Klasse werden in Java mit dem Operator `new` angelegt. Üblicherweise, erst nach Anlegen eines Objekts einer Klasse, existieren die Daten und kann auf die Methoden des Objekts zugegriffen werden. Eine Ausnahme bilden nur die so genannten mit `static` definierten Klassenvariablen bzw. -methoden, die bereits auf Seite 192 erwähnt wurden.

7.6.4 Konstruktoren

Wird ein Objekt zu einer Klasse angelegt, so sind die Daten dieses Objekts zunächst undefiniert. Da die Daten meist privat sind, ist es auch nicht möglich, ihnen explizit (von außen) einen Wert zuzuweisen. Deswegen stellt man manchmal eigene Initialisierungsfunktionen zur Verfügung, die der Nutzer einer Klasse dann explizit nach dem Anlegen eines Objekts aufrufen muss, wie z. B. die Methode `initKonto()` im Programm 7.8 Solche explizit aufzurufenden Initialisierungsmethoden sind zum einen sehr umständlich und bergen zusätzlich die Gefahr, dass nach dem Anlegen eines Objekts der notwendige Aufruf der entsprechenden Initialisierungsmethode vergessen wird. Besser wäre es, wenn man ein Objekt bereits bei seiner Deklaration initialisieren könnte, wie man es z. B. für die Standarddatentypen kennt:

```
int    sum = 0;
double wert = 7.24;
```

Java bietet über so genannte *Konstruktoren* diese Möglichkeit der Initialisierung auch für Objekte an. Um die Datenelemente eines Objekts bereits bei seiner Deklaration initialisieren zu können, muss man eine spezielle Methode mit dem gleichen Namen wie die Klasse, einen so genannten *Konstruktor*, angeben und darin die gewünschten Initialisierungen durchführen. Programm 7.10 ist eine Version mit Konstruktoren zu unserem Konto-Beispiel.

Listing 7.10: Java-Programm Konto2.java mit Konstruktor

```
import java.io.*;

public class Konto2
{
    // .....   private Membervariablen (von ausserhalb nicht zugreifbar)
    //  identisch zu Programm 7.8
    // .....   private Methoden (von ausserhalb nicht aufrufbar)
    //  identisch zu Programm 7.8
```

```
    // .....   public Methoden (von ausserhalb aufrufbar)
    public Konto2(int n, int p) { // statt: initKonto(int n, int p)
        ein    = new Eingabe();
        stand = 0;
        nr     = n;
        pin    = p;
    }
    // .....   restliche  Methoden identisch zu Programm 7.8
}
```

Programm 7.11 zeigt das Anlegen der beiden `Konto`-Objekte `eins` und `zwei`, wobei diese nun mittels der Konstruktor-Aufrufe bereits beim Anlegen geeignet initialisiert werden.

Listing 7.11: Java-Programm KontoMain2.java mit Konstruktor-Aufrufen

```
public class KontoMain2
{
    public static  void main (String args[])
    {
        Konto2 eins = new Konto2(11111, 4711),
                zwei = new Konto2(22222, 9999);

        eins.zahleEin(700);
        zwei.zahleEin(500);
        eins.ueberweise(zwei, 300);
        zwei.hebeAb(150);
        eins.getKontostand();
    }
}
```

7.6.5 Vererbung und Polymorphismus

Nehmen wir nochmals die Einteilung der Lebewesen, wie sie in Abbildung 7.52 auf Seite 240 gezeigt ist. In einer solchen Hierarchie gilt dann Folgendes:

- *Alle Lebewesen einer Klasse verfügen über identische Eigenschaften.*
- *Klassen in einer tieferen Hierarchiestufe sind eine Spezialisierung der direkt übergeordneten Klasse.*
- *Die von einer Klasse abgeleiteten Unterklassen verfügen immer über alle Eigenschaften der Oberklasse.* Eigenschaften werden also automatisch an Unterklassen *weitervererbt*.
- *Die von einer Klasse abgeleiteten Unterklassen können neben den geerbten Eigenschaften weitere eigene Eigenschaften hinzufügen.* So erbt z. B. die Klasse `Hund` in Abbildung 7.52 auf Seite 240 alle Eigenschaften von der Klasse `Landtier`, erweitert diese aber um eigene speziellere Eigenschaften, wie z. B. „*kann bellen*“.

Solche Hierarchien lassen sich jedoch nicht nur für Lebewesen entwerfen, sondern ebenso für Daten und Konzepte, wie sie in der Softwareentwicklung vorkommen. Da eine solche Hierarchie dem Stammbaumprinzip entspricht, spricht man auch von *Vererbung* von Eigenschaften. Die Klasse, die Eigenschaften weitervererbt, wird *Ober-*

klasse (bzw. *Basisklasse* oder *Superklasse*) genannt, und Klassen, die etwas erben, heißen *Unterklassen* (bzw. *Subklassen*) oder *abgeleitete Klassen*.

Im Zusammenhang mit der Vererbungshierarchie verwendet man die zwei Begriffe:

- *Generalisierung*: Eine Oberklasse ist eine Generalisierung der Unterklassen,
- *Spezialisierung*: Eine Unterklasse ist eine Spezialisierung der Oberklasse.

Eine abgeleitete Klasse kann wieder als Basisklasse für weitere Unterklassen dienen.

Einfache Vererbung

Das UML-Klassendiagramm in Abbildung 7.66 zeigt eine Basisklasse `Mensch`, von der zwei Unterklassen `Student` und `Angestellter` abgeleitet sind, die beide sowohl die Attribute `name` und `Alter` als auch die Methode `getDaten()` von der Klasse `Mensch` erben. Ihrerseits erweitern sie aber diese geerbten Eigenschaften noch um `matrikelnr` und `getStudentDaten()` bzw. um `gehalt` und `getAngestDaten()`. In Abbildung 7.66 wurde neben den Zugriffsspezifizierern `+` und `-` für `public` und `private` noch ein weiterer Schutztyp mit dem Namen `protected` verwendet, der dort mit dem Zeichen # dargestellt ist:

`protected` ist die Zwischenstufe zwischen `private` und `public`

Beim Vererben von Datenelementen werden diese in jedem Objekt der erbenden Klasse jeweils in einer eigenen Kopie neu angelegt. Da allerdings `private` Elemente einer Klasse grundsätzlich nicht weitervererbt werden, müsste man ohne dieses Schlüsselwort `protected` alle Elemente einer Basisklasse, die weitervererbt werden sollen, als `public` einstufen. Damit könnten aber auch andere „fremde“ Funktionen auf diese Elemente zugreifen, womit das wichtigste Prinzip der Objektorientierung aufgehoben wäre. Um nun nur den erbenden Klassen und nicht dem ganzen Programm Zugriff auf bestimmte Elemente der Klasse zu erlauben, musste eine Zwischenstufe zwischen den beiden extremen Schutztypen `private` und `public` eingeführt werden, nämlich der Schutztyp `protected`. Bei Verwendung von `protected` für Klassenelemente erhält eine Klasse zwei Schnittstellen nach außen, und zwar

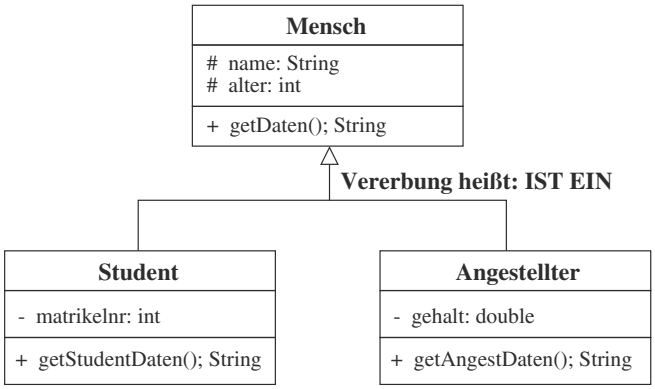

Abbildung 7.66: Vererben von Klasse `Mensch` an Klassen `Student` und `Angestellter`

Copyright