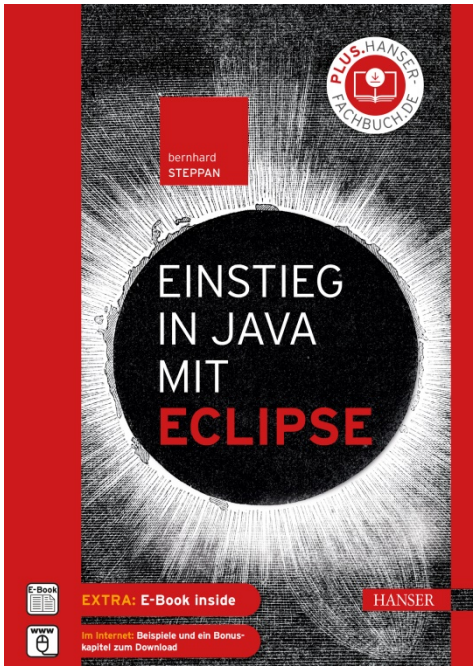
HANSER



Leseprobe

zu

„Einstieg in Java mit Eclipse“

von Bernhard Steppan

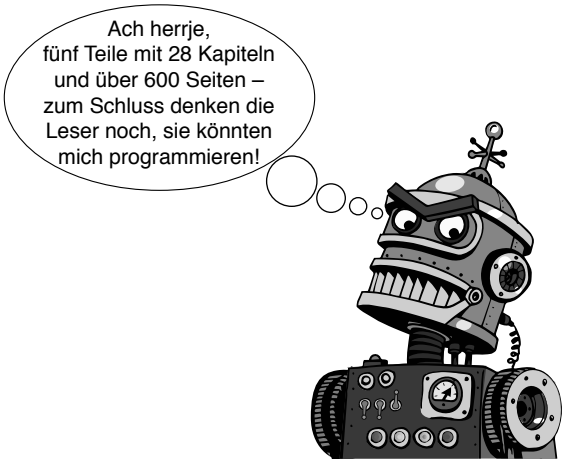
Print-ISBN: 978-3-446-45910-6
E-Book-ISBN: 978-3-446-45976-2
E-Pub-ISBN: 978-3-446-46326-4

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-45910-6
sowie im Buchhandel

Inhalt

Vorwort

Java ist zurzeit unumstritten die bedeutendste Programmiersprache. Daher möchten viele Java lernen. Der Einstieg ist leider nicht einfach, denn für das Programmieren mit Java sind mindestens zwei Dinge erforderlich: das Beherrschen der Programmiersprache *und* das Beherrschen einer Entwicklungsumgebung. Aus diesem Grund ist dieses Buch entstanden. Es zeigt anhand vieler Beispiele, wie die Sprache aufgebaut ist. Zusätzlich vermittelt das Buch am Beispiel der Eclipse-Entwicklungsumgebung, wie Sie mit diesem Werkzeug Java-Programme entwickeln.



Robert aus der Maschinenwelt begleitet Sie durch das Buch.

Der erste Teil »Grundlagen« vermittelt das Java- und Eclipse-Basiswissen. Dieser Teil legt die Programmiergrundlagen, gibt Ihnen einen Überblick über die Java-Technologie und zeigt Ihnen, was das Besondere an objektorientierter Programmierung ist. Ein Kapitel über die Eclipse-Entwicklungsumgebung rundet diesen Teil ab.

Im zweiten Teil »Sprache Java« dreht sich alles um die Feinheiten der Sprache Java. Hier entstehen die ersten kleinen Java-Anwendungen. Dieser Teil bietet eine Mischung aus Wissensteil und praktischen Übungen. An jedem Kapitelende finden Sie Aufgaben, die Sie selbstständig durchführen können. Mit den Lösungen zu den Aufgaben am Schluss dieses Buchs überprüfen Sie den Lernerfolg.

Die Technologie Java bildet den Schwerpunkt des dritten Teils »Plattform Java«. Dieser Teil stellt Ihnen zusätzlich vor, welche Gesetzmäßigkeiten Sie beim Programmieren beachten müssen, was Klassenbibliotheken sind und welche Vorteile sie haben. Zusätzlich erfahren Sie, wie Sie Programme testen, was Algorithmen sind und wie Sie sie programmieren.

Ein größeres Java-Projekt steht im Mittelpunkt des vierten Teils. Hier lernen Sie alle Elemente der vorigen Teile an einer Anwendung mit grafischer Oberfläche kennen. Das Projekt zeigt, wie man mit der Eclipse-Entwicklungsumgebung Stück für Stück eine größere Anwendung entwickelt.

Der fünfte Teil »Anhang« beschließt dieses Buch mit Lösungen zu den Aufgaben, mit Grundlagen der Informationsverarbeitung und einem Kapitel zu den häufigsten Fehlern, die bei der Arbeit mit Eclipse entstehen können.

Rahmenhandlung

Als Rahmenhandlung habe ich dem Buch den (fiktiven) Programmierkurs »Java mit Eclipse« von Professor Roth mit vier Studentinnen und Studenten zugrunde gelegt. Den Programmierkurs begleitet der Roboter namens Robert und – neben vielen anderen – vor allem diese fünf Figuren durch das gesamte Buch:

Der Programmierkurs mit Lukas, Anna, Professor Roth, Julia und Florian

Für wen ist das Buch?

Dieses Buch richtet sich an aktive Leser. Sie wollten nicht fertige Lösungen, sondern möchten selbst programmieren. Ohne dass Sie selbst aktiv programmieren und am Ball bleiben, bis Ihr selbstgeschriebenes Programm läuft, lernen Sie kein Java. Das Buch enthält verlockend viele fertig programmierte Beispiele, die Sie auf Knopfdruck ausführen können. Greifen Sie nur zu den Musterlösungen, wenn Sie nicht weiterkommen. Versuchen Sie zuerst, die Programme selbst einzugeben und aus den Fehlern zu lernen. Nur durch aktives Programmieren beherrschen Sie Java und die Eclipse-Entwicklungsumgebung.

Bonusmaterial

Das Buch enthält eine Fülle von Beispielen, die einfach in die Eclipse-Umgebung als Lösungen importiert werden können. Sie können Sie einfach von *https://plus.hanser-fachbuch.de* herunterladen (siehe auch Abschnitt 4.2.4, dort finden Sie im Abschnitt »Download der Beispielprogramme« nähere Informationen). Unter diesen Downloads finden Sie zudem ein Bonuskapitel, das aus Platzgründen nicht abgedruckt ist. Es erklärt die Programmierung von sogenannten Terminalprogrammen.

Schriftkonventionen

Verschiedene Textteile sind zur besseren Lesbarkeit wie folgt hervorhoben:

Textteil	Bedeutung
Datentypen im Fließtext	*Person*
Datentypen in Überschriften	»Person«
Schlüsselwörter im Fließtext	*implements*
Schlüsselwörter in Überschriften	»implements«
Variablen im Fließtext	*roland*
Variablen in Überschriften	»roland«
Fenster (grafische Oberfläche)	Eclipse IDE Launcher
GUI-Element (grafische Oberfläche)	Finish
Menü (grafische Oberfläche)	File
Menübefehl (grafische Oberfläche)	Menü → File → New → Java Project
Dateien	*Beispielprogramme.zip*
Verzeichnispfade	*C:/Programme/Eclipse*

Listing (Quellcode von Beispielprogrammen)	`1 package programmierkurs;`
	`2 public class Roboter {`
	`3 (...)`
	`4 }`

Programmausgabe	`Ergebnis = true`
URL	*http://eclipse.org*
`(...)`	Aus Platzgründen fehlt ein Teil des Quellcodes

Herzlichen Dank!

Hiermit möchte ich mich bei allen bedanken, die mich beim Schreiben dieses Buchs unterstützt haben: dem Carl Hanser Verlag und meiner Lektorin Brigitte Bauer-Schiewek für das Vertrauen in meine Arbeit und die große Geduld. Der Herstellung des Hanser Verlags möchte ich für die professionelle LaTeX-Vorlage danken und Herrn Korell für seine stets kompetente Unterstützung bei Fragen zum Satz dieses Buchs mit der LaTeX-Vorlage.

Meine Frau Christiane hat mich wie immer sehr bei diesem Projekt unterstützt. Herzlichen Dank für Deine Hilfe! Danken möchte ich zudem Herrn Dubau, der mein Buchmanuskript von Rechtschreibfehlern befreit hat. Herzlichen Dank auch an Valery Kachaev, von dem die Vorlagen für die Roboter-Cartoons stammen.

Kontakt

Trotz größter Sorgfalt lässt sich nicht immer verhindern, dass der eine oder andere Fehler in einem Buch übersehen wird. Wenn Sie Fehler finden, Verbesserungsvorschläge oder Fragen haben, senden Sie mir einfach eine E-Mail an *java-eclipse@steppan.net*. Ich werde Ihre Fragen möglichst schnell beantworten und versuchen, Ihre Verbesserungsvorschläge in kommenden Auflagen zu berücksichtigen. Die jeweils aktuellsten Ergänzungen und weitere Informationen finden Sie unter *http://www.steppan.net*. Nun wünsche ich viel Spaß beim Lesen und Entwickeln Ihrer Java-Programme mit Eclipse!

Bernhard Steppan Wiesbaden im August 2020

3 Objektorientierte Programmierung

3.1 Einleitung

Java ist eine objektorientierte Programmiersprache. Daher ist es wichtig, genau zu verstehen, wie objektorientierte Programmierung funktioniert. In diesem Kapitel dreht sich alles um Objekte und Klassen, um Attribute und Methoden sowie darum, wie sich Objekte vor Übergriffen anderer feindlicher Objekte schützen lassen. Sie erfahren dabei auch, warum Objektorientierung entstand und was so besonders daran ist.

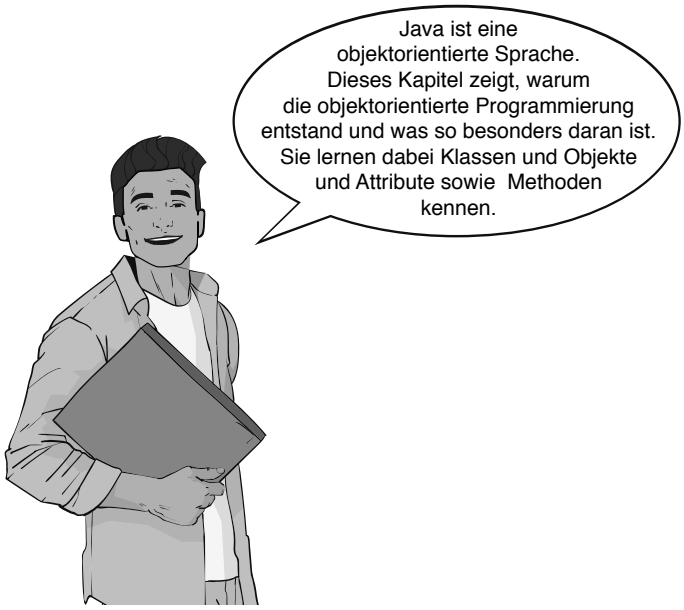


Abbildung 3.1 Florian erklärt Ihnen, was das Besondere an objektorientierter Programmierung ist.

3.2 Überblick

Wie kam das alles? Es begann alles Mitte der 60er-Jahre des 20. Jahrhunderts. Damals kam es zu einer Softwarekrise. Zu dieser Zeit stiegen die Anforderungen an Programme. Die Software wurde dadurch komplexer und fehlerhafter. Auf Kongressen diskutierten Experten die Ursachen der Krise und die Gründe für die gestiegene Fehlerrate.

Ein Teil der Softwareexperten kam zu dem Schluss, dass die Softwarekrise nicht mit den herkömmlichen Programmiersprachen zu bewältigen sei. Sie kritisierten an den herkömmlichen Programmiersprachen vor allem, dass sich die natürliche Welt bisher nur unzureichend abbilden lasse. Sie begannen deshalb, eine Generation von neuen Programmiersprachen zu entwickeln.

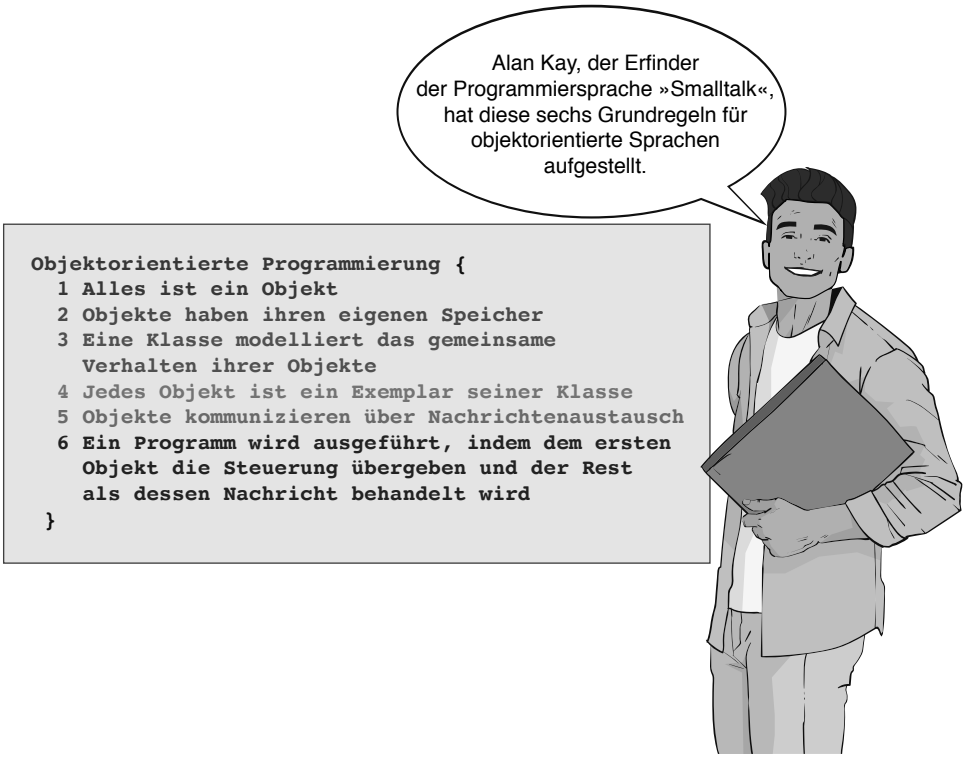


Abbildung 3.2 Merkmale objektorientierter Sprachen

Die Experten begannen damit, natürliche Begriffe aus der Formenlehre der klassischen griechischen Philosophie für die neuen Programmiersprachen zu verwenden. Sie wandelten diese Bezeichnungen für die Programmierung ab (Abbildung 3.2). Da sich alles um den Begriff des Objekts dreht, nannten sie die neue Generation von Sprachen »objektorientiert«.

3.3 Objekt

Objekte sind für ein Java-Programm das, was Zellen für einen Organismus sind: Aus diesen kleinsten Einheiten setzt sich ein Java-Programm zusammen. Wenn Sie eine Reihe von gleichartigen Objekten betrachten, fällt auf, dass ihr prinzipielles *Aussehen* gemeinsam ist. In der objektorientierten Programmierung bezeichnet man solche Objekte auch oftmals als *Exemplare* einer Klasse.

Als Beispiel soll wieder der Programmierkurs von Professor Roth dienen. An dem Programmierkurs nehmen Anna, Julia, Lukas und Florian teil. Greifen wir zunächst nur die Studentinnen Anna und Julia heraus. Beide Studentinnen sind Objekte mit vielen Gemeinsamkeiten. Gemeinsam ist ihnen zum Beispiel, dass sie Frauen sind, den gleichen Programmierkurs besuchen und an der gleichen Hochschule eingeschrieben sind.

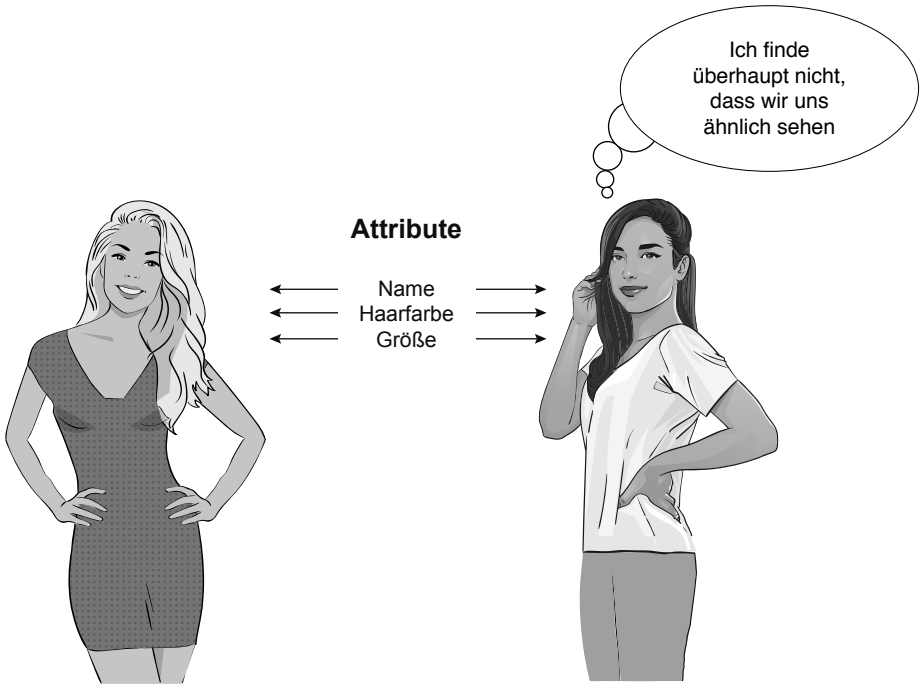


Abbildung 3.3 Die »Objekte« Anna und Julia sind sich ähnlich, haben aber unterschiedliche Attribute.

Aus Sicht der objektorientierten Programmierung bedeutet das: Diese beiden Objekte sind *ähnlich*. Die Unterschiede zwischen diesen Objekten ergeben sich aus dem unterschiedlichen Wert ihrer *Attribute*. Beiden Studentinnen haben zum Beispiel einen verschiedenen Namen und unterschiedliche Haarfarbe sowie Größe (Abbildung 3.3).

Gleichartige Objekte haben also nur ihre *prinzipielle* Gestalt und bestimmte Fähigkeiten gemeinsam. Alles andere ist *individuell*. Die gemeinsame Gestalt und die gemeinsamen Fähigkeiten von Objekten legt der Bauplan der Objekte fest. Diesen Bauplan nennt die objektorientierte Programmierung *Klasse*.

3.4 Klasse

Die *Klasse* ist es, die die prinzipielle Gestalt und die Fähigkeiten von Objekten wie den beiden Studentinnen prägt. Eine Klasse verhält sich zu einem Objekt wie der Bauplan eines Menschen zu einem realen Menschen. Die Klasse gibt verschiedenen Objekten der gleichen Art einen Obergriff. Man sagt auch, dass eine Klasse seine Objekte *klassifiziert* – daher auch der Name.

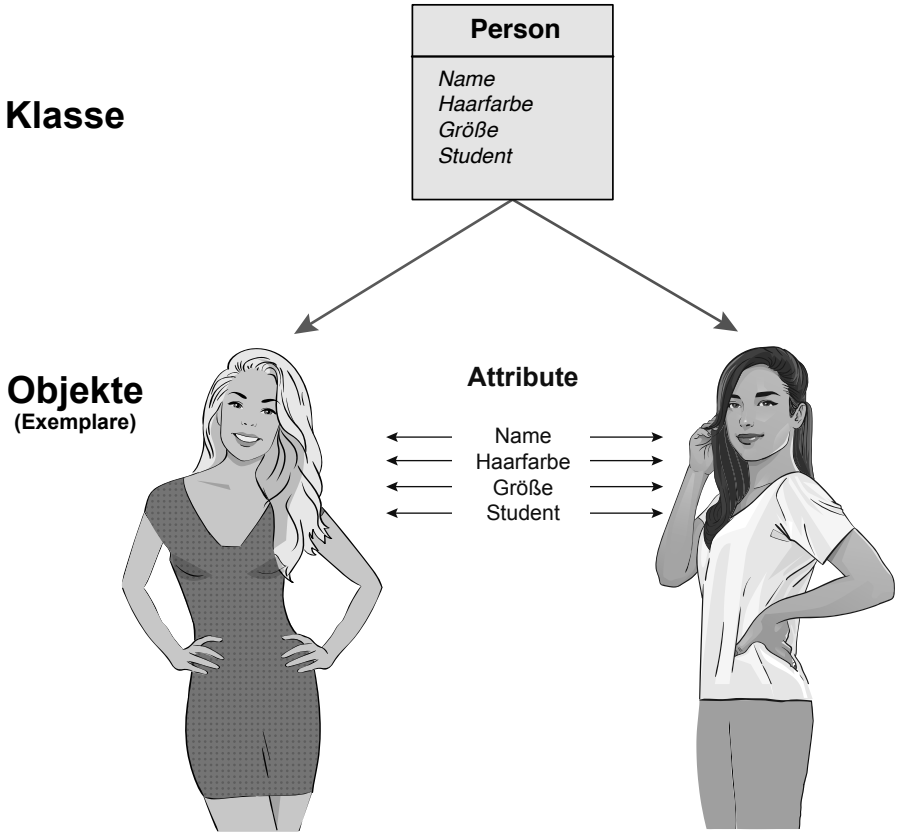


Abbildung 3.4 Die Klasse »Person« liefert den Bauplan für die »Objekte« Anna und Julia.

3.4.1 Attribute

Unsere neue Klasse *Person* soll neben den Attributen *Name*, *Haarfarbe* und *Größe* noch zusätzlich das Attribut *Student* bekommen. Dieses Attribut legt fest, ob jemand Student ist oder nicht. Wenn aus dieser Klasse neue Personenobjekte entstehen, besitzen alle Exemplare einen individuellen *Namen*, eine individuelle *Haarfarbe*, eine individuelle *Größe* und einen Wert für die Markierung *Student* (Abbildung 3.4).

Konstanten

Beispielsweise soll *Anna* über folgende Attribute verfügen: Name = Anna, Haarfarbe = blond, Größe = 1,71 m. Ihre Freundin aus dem Programmierkurs heißt Julia, hat die Haarfarbe braun und ist 1,72 m groß. Obwohl beide Personen nach dem gleichen Bauplan (Klasse) erzeugt worden sind, sind zwei deutlich unterschiedliche Objekte entstanden: Beide haben unterschiedliche Namen, haben unterschiedliche Haarfarbe und sind beide unterschiedlich groß.

Zustände

Es ist Ihnen vielleicht aufgefallen, dass bei den bisherigen Attributen der beiden Personen einige mit festen Werten belegt waren, andere hingegen mit veränderlichen Werten. Die flexiblen Attribute beschreiben den *Zustand* des Objekts. Zum Beispiel beschreibt das Attribut *Student*, ob eine Person gerade an einer Hochschule eingeschrieben ist. Der Zustand eines Objekts kann sich im Laufe der Zeit ändern.

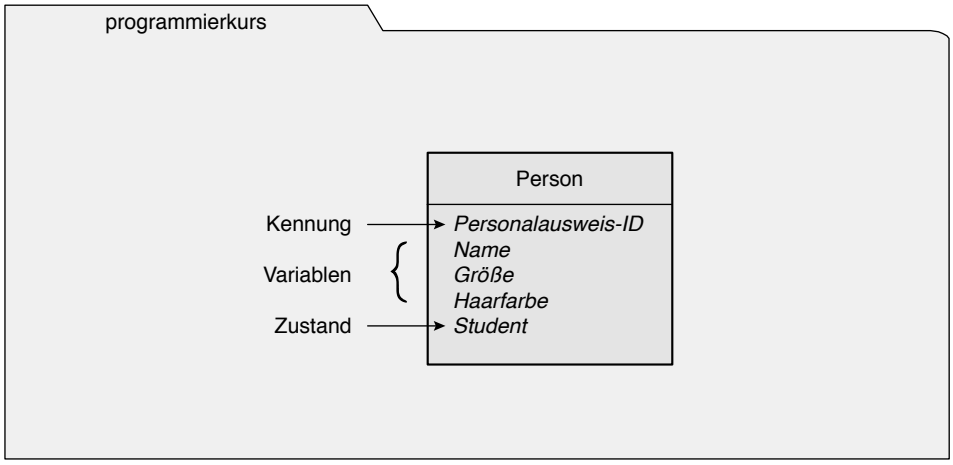


Abbildung 3.5 Variablen, Zustände und Kennungen sind Attribute einer Klasse.

Kennungen

Was würde passieren, wenn man die Objekte *Anna* und *Julia* so erzeugen würde, dass sie die gleiche *Größe*, die gleiche *Haarfarbe* und den gleichen Zustand *Student* besitzen? Wie könnte man sie dann unterscheiden? In diesem Fall haben beide Objekte zwar individuelle Werte für ihre Attribute bekommen, aber diese sind zufällig gleich. Damit gleichen sich auch die Objekte in einem Programm wie eineiige Zwillinge.

Um Objekte also besser zu unterscheiden, benötigt man so etwas wie einen genetischen Fingerabdruck. In der Programmierung vergibt der Entwickler eine sogenannte Kennung. Diese Kennung ist ein zusätzliches Attribut, bei dem darauf geachtet wird, dass es *eindeutig* ist. Erst die Kennung eines Objekts sorgt dafür, dass das Programm unterschiedliche Exemplare auch dann unterscheiden kann, wenn ihre Attribute zufällig die gleichen Werte besitzen.

3.4.2 Methoden

Angenommen, Sie wollen *Anna* mitteilen, dass sie auf eine Frage antworten soll. Im wirklichen Leben stellen Sie Anna einfach eine Frage oder schicken ihr eine Nachricht. In der objektorientierten Programmierung rufen Sie stattdessen eine Methode des Objekts *Anna* auf (Abbildung 3.6). Eine Methode ist der objektorientierte Begriff für eine Funktion. Der Begriff wurde eingeführt, um auszudrücken, dass eine objektorientierte Funktion viel leistungsfähiger ist als eine Funktion einer klassischen Programmiersprache.

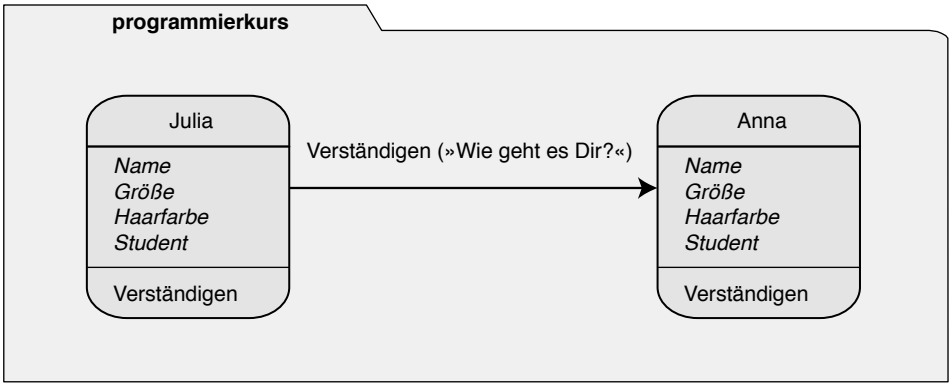


Abbildung 3.6 Objekte verständigen sich durch den Aufruf von Methoden.

Aber egal, wie der Begriff genannt wird, eines ist gleich: Verhaltensweisen wie *Verständigen* bestimmen die Fähigkeit eines Objekts, zu kommunizieren und Aufgaben zu erledigen. Objekte lassen sich also über Methoden steuern. Es existiert nicht nur eine Art von Methoden, sondern es gibt folgende fünf Grundtypen: *Konstruktoren* (»Erbauer«), *Destruktoren* (»Zerstörer«), *Änderungsmethoden* (»Setter«), *Abfragemethoden* (»Getter«) und *Operationen* (»Funktionen«).

Konstruktoren

Wie im natürlichen Leben haben auch Objekte eines Computerprogramms einen Lebenszyklus. Sie werden geboren, also erzeugt, und irgendwann sterben sie, das heißt, sie werden gelöscht und ihr Speicher wird freigegeben. Die wichtigste Methode ist die, die ein Objekt erzeugt. Sie werden demzufolge auch »Konstruktoren« genannt. Sie konstruieren, das heißt erschaffen ein Objekt.

Destruktoren

Methoden, die ein Objekt zerstören, nennen sich in der objektorientierten Programmierung »Destruktoren«. In manchen Programmiersprachen können Sie diese Destruktoren direkt aufrufen und damit unmittelbar ein Objekt zerstören. In Java funktioniert dies nicht. Hier wird ein Objekt automatisch zerstört, wenn es nicht mehr benötigt wird. Dazu später mehr.

Änderungsmethode

Methoden, die den Wert eines Attributs verändern, nennen sich »Änderungsmethoden« oder englisch »Setter«. Sie verändern den Zustand des Objekts. Mit einer solchen Methode lässt sich die Geschwindigkeit ändern, mit der *Anna* sich fortbewegt (Abbildung 3.7). Die entsprechende Methode nennt sich *Laufen* und verfügt über einen sogenannten Parameter, der den neuen Zustand, die *Geschwindigkeit*, vorgibt.

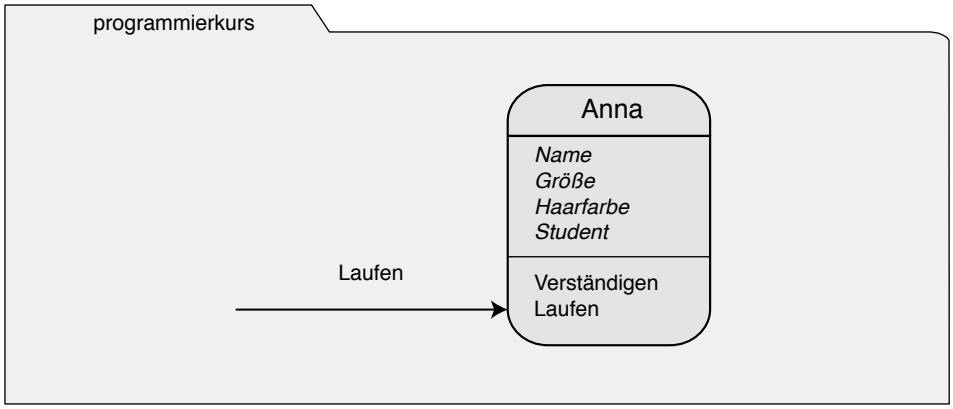


Abbildung 3.7 Die Änderungsmethode »Laufen« ändert den Zustand von »Anna«.

Abfragemethode

Abfragemethoden oder englisch »Getter« sind Methoden, die nur ein bestimmtes Attribut abfragen. Sie ändern nichts am Zustand des Objekts. Eine solche Methode wäre zum Beispiel die Abfrage, ob Anna an einer Hochschule eingeschrieben ist (Abbildung 3.8). Diese Methode besitzt einen sogenannten Rückgabewert: den Status des Studenten, der mit *true* oder *false* beantwortet werden kann.

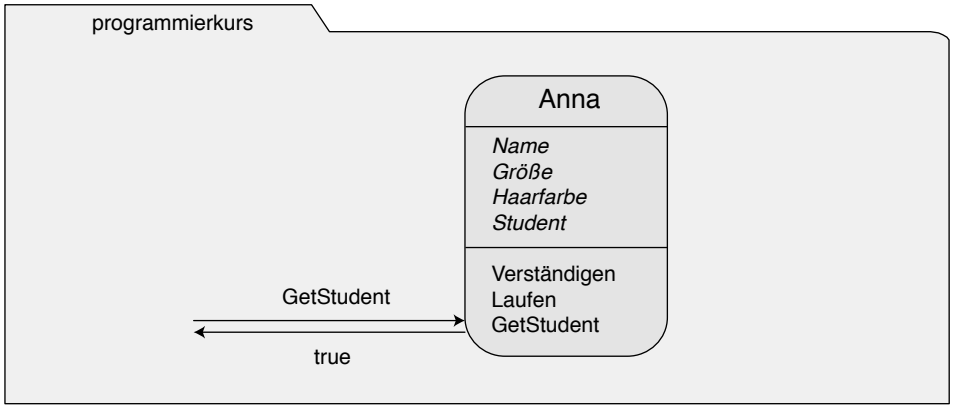


Abbildung 3.8 Die Abfragemethode erfragt den Studentenstatus von Anna.

Operationen

Methoden, die zum Beispiel nur eine Rechenoperation durchführen, werden in der objektorientierten Programmierung meistens als Operationen oder Funktionen bezeichnet. Sie dürfen trotzdem nicht mit den gleichnamigen Funktionen klassischer Programmiersprachen verwechselt werden, denn sie werden wie andere Methoden auch von Klasse zu Klasse weitervererbt (Abschnitt 3.6, »Vererbung«). Zudem lassen sie sich überladen und überschreiben (Abschnitt 3.13, »Polymorphie«).

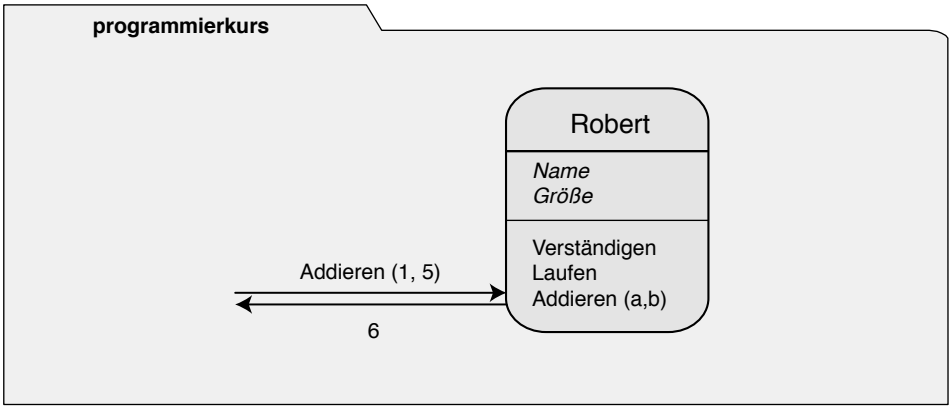


Abbildung 3.9 Die Operation »Addieren« liefert die Summe als Ergebnis zurück.

3.5 Abstraktion

Vielleicht werden Sie jetzt sagen: »Das ist doch alles Unsinn. Die Fähigkeiten und Attribute einer Person sind viel komplexer und können nicht auf Größe und Farbe sowie auf Verständigen reduziert werden.« Das ist in der natürlichen Welt richtig, aber in der künstlichen Welt der Softwareentwicklung in der Regel völlig falsch.

Richtig wäre es nur dann, wenn man die Natur in einem Programm vollständig abbilden müsste. Aber für so eine übertriebene Genauigkeit gibt es bei der Programmierung selten einen Grund. Die objektorientierte Programmierung erleichtert eine möglichst natürliche Abbildung der realen Welt und fördert damit gutes Softwaredesign.

Sie verführt damit auch zu übertriebenen Konstruktionen. Die Kunst besteht darin, dem entgegenzusteuern und die Wirklichkeit so genau wie nötig, aber so einfach wie möglich abzubilden. Wie Sie später bei größeren Beispielprogrammen sehen werden, bereitet gerade die Analyse der für das Programm wesentlichen und richtigen Bestandteile oftmals große Probleme.

Wenn man innerhalb eines Programms nur die für die Funktionalität wesentlichen Teile programmiert, dann hat das praktische Gründe: Das Programm lässt sich schneller entwickeln, es wird billiger und schlanker. Somit benötigt es weniger Speicherplatz, und es wird in der Regel schneller ablaufen als ein Programm, das mit unnötigen Informationen überfrachtet ist.

Um diese Kompaktheit zu erreichen, ist es notwendig, die meist extrem komplizierten natürlichen Objekte und deren Beziehungen so weit es geht zu abstrahieren, also zu vereinfachen. Der Fachbegriff für diese Technik nennt sich demzufolge auch *Abstraktion* (Abbildung 3.10).

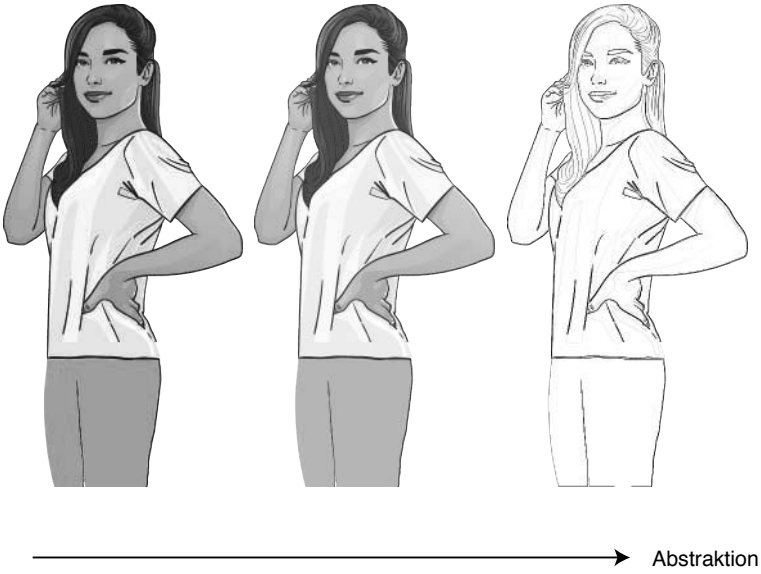


Abbildung 3.10 Durch Abstraktion erhält man das Wesentliche einer Klasse.

3.6 Vererbung

Nach der Einführung von Klassen, Objekten, Methoden und Attributen wird es Zeit, diese neuen Begriffe in Zusammenhang mit dem Begriff der *Vererbung* zu bringen. Vererbung gestattet es, das Verhalten zwischen Klassen und damit auch zwischen Objekten mithilfe eines Bauplans zu übertragen. Die Vererbung kopiert Attribute und Methoden der Basisklasse auf die abgeleitete Klasse.

Um das zu verdeutlichen, wieder ein Beispiel: Menschen und Roboter sind sich in mancher Hinsicht ähnlich, aber in vielfältiger Hinsicht doch extrem verschieden. Diese Unterschiede sind von anderer Güte als die Unterschiede zwischen zwei Menschen: Menschen und Roboter haben eine deutlich unterschiedliche Gestalt (Abbildung 3.11).

Dass Menschen im Sinne der Formenlehre eine andere Gestalt besitzen als Roboter, muss man angesichts der wuchtigen Erscheinung von Robert eigentlich nicht besonders betonen. Um die Unterschiede auf den Punkt zu bringen, hilft es, wenn Sie einfach einmal versuchen, die auf den vorhergehenden Seiten aufgestellten Attribute von Personen mit denen eines Roboters in Einklang zu bringen. Wie Sie sehen werden, funktioniert das nur für einen bestimmten Teil der Attribute.

Was bedeutet das für die Programmierung? Das bedeutet, dass Sie auf die gerade gezeigte Weise herausfinden können, ob Objekte zu einer gemeinsamen Klasse gehören. In allen

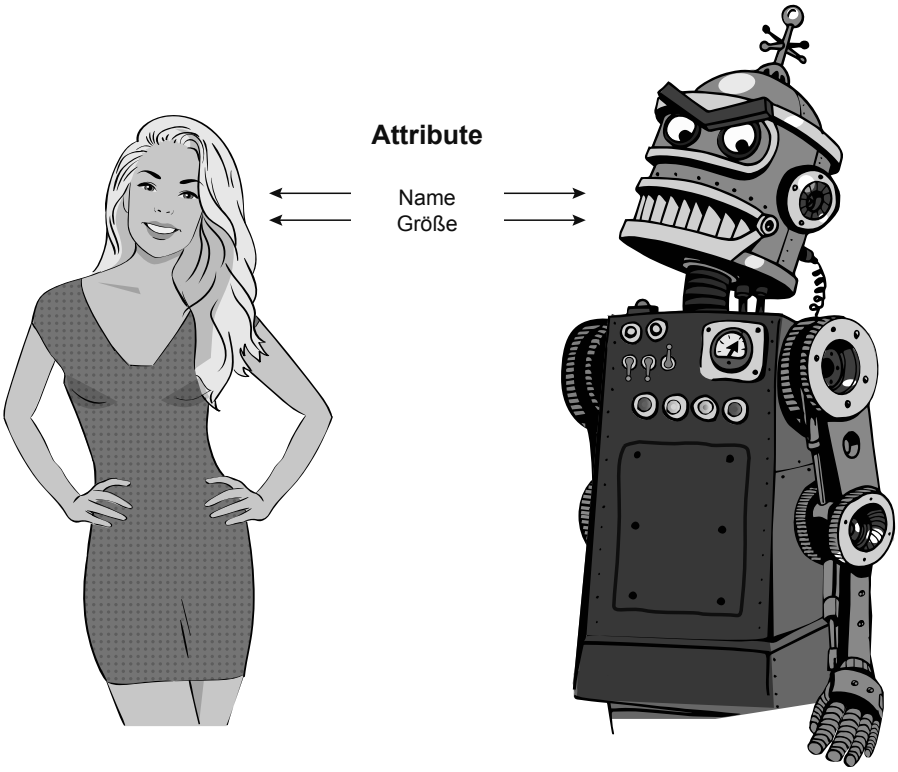


Abbildung 3.11 Objekte verschiedener Klassen unterscheiden sich in ihrer Form.

Fällen, in denen es auf die Gemeinsamkeiten ankommt, sollte man den Objekten eine gemeinsame Klasse zuordnen. In allen Fällen, in denen es auf die Unterschiede zwischen Objekten ankommt, weist man ihnen besser getrennte Klassen zu.

3.6.1 Basisklassen

Bis jetzt haben wir die Unterschiede zwischen Mensch und Roboter betont. Was passiert, wenn wir es einmal von der anderen Seite betrachten: Was haben ein Mensch wie Anna und eine humanoide Maschine wie Robert gemeinsam? Aus Sicht der Programmierung hat jeder Mensch wie Anna und jeder Roboter wie Robert eine bestimmte Größe. Sie können beide laufen und sich verständigen.

Die genannten Eigenschaften teilen sie mit einer Vielzahl von Lebewesen. Sind Roboter Lebewesen? Wohl eher nicht. Die meisten Menschen würden ihnen sicher die Fähigkeit zu leben absprechen. Man könnte sie aber als künstliche Wesen oder Maschinenwesen bezeichnen. Aus Sicht der objektorientierten Programmierung bedeutet das, dass wir Mensch und Roboter einer gemeinsamen Basisklasse *Wesen* zuordnen könnten. Abbildung 3.12 zeigt, wie eine solche Basisklasse aussehen würde.

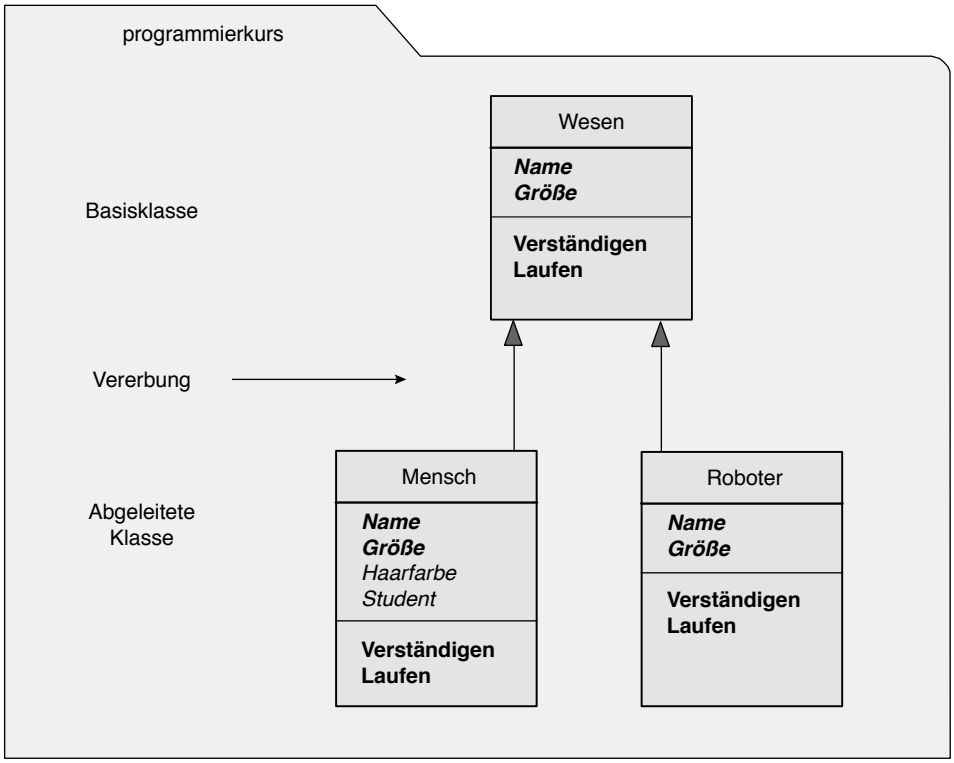


Abbildung 3.12 Die Basisklasse überträgt Basiseigenschaften und -verhalten.

Beachten Sie besonders die fett hervorgehobenen Wörter: Damit man nicht für jede der Klassen *Mensch* und *Roboter* die Attribute *Name* und *Größe* sowie das Verhalten *Verständigen* und *Laufen* neu programmieren muss, bietet es sich an, dieses Verhalten in eine Basisklasse zu verlagern. Die fett gedruckten Attribute und Methoden sind aus Sicht der objektorientierten Programmierung die *Gemeinsamkeiten* beider Klassen. Die gemeinsamen Attribute der Basisklasse erleichtern nicht nur die Programmierung. Sie vereinheitlichen auch alle abgeleiteten Klassen.

3.6.2 Abgeleitete Klassen

Angenommen, Sie möchten eine neue Klasse namens *Tier* auf Basis der Klasse *Wesen* erzeugen. In der objektorientierten Programmierung würde man sagen, man leitet von der Klasse *Wesen* eine neue Klasse namens *Tier* ab. Wie in der Ahnenforschung bei Stammbäumen sagt man auch, die Klasse *Tier* stammt von der Klasse *Wesen* ab. Die neue Klasse *Tier* würde, wie schon zuvor die Klassen *Mensch* und *Roboter*, die Attribute *Name, Größe* sowie das Verhalten mit *Verständigen* und *Laufen* von der Basisklasse *Wesen* übernehmen. Attribute und Verhalten vererben sich also (Abbildung 3.13).

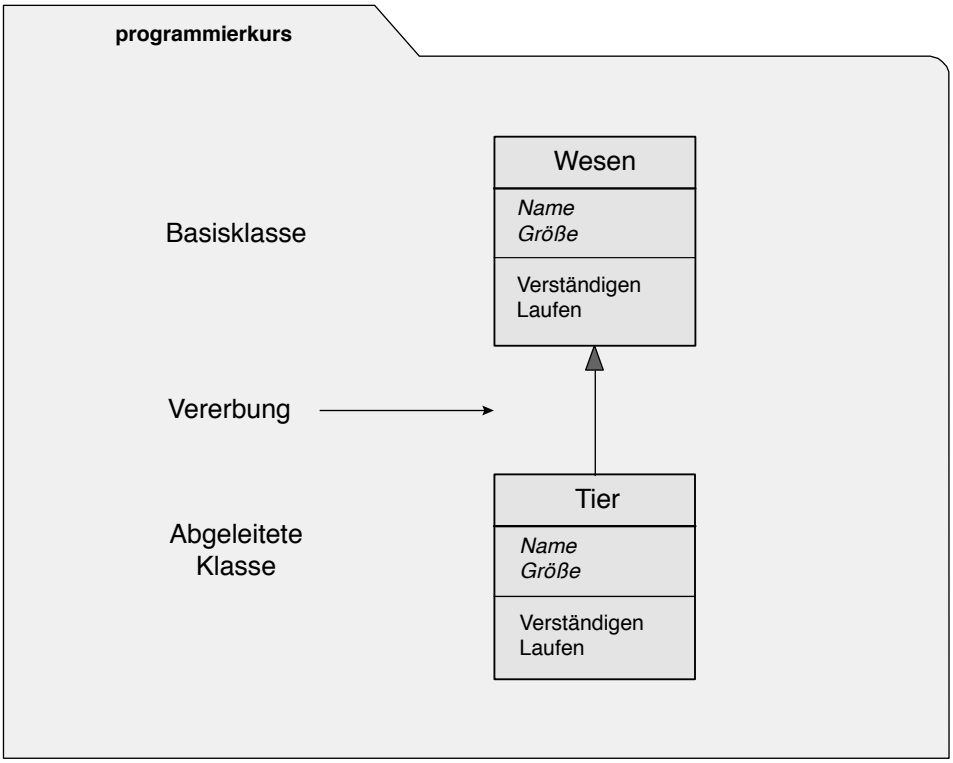


Abbildung 3.13 Die neue Klasse »Tier« ist eine von »Wesen« abgeleitete Klasse.

3.6.3 Mehrfachvererbung

In der Natur ist sie üblich, in der Programmiersprachen Java jedoch aus gutem Grund nicht erlaubt: die Mehrfachvererbung. Sie wäre dann praktisch, wenn Sie zwei Klassen verschmelzen wollten, zum Beispiel die Klasse *Mensch* mit der Klasse *Pferd*. Die neue Kreuzung *Kentaur* würde Attribute und Verhalten beider Basisklassen erben (Abbildung 3.14). Aber welche Attribute und welches Verhalten? Sollen sich Kentauren verständigen und laufen wie Menschen oder wie Pferde? Können Kentauren Studenten sein?

So schön das Beispiel in der Mythologie ist, bei derartigen Szenarien kommt die Softwareentwicklung an die Grenze des technisch Sinnvollen. Es ist eben nicht sinnvoll, Erbinformationen nach dem Zufallsprinzip zu übertragen, um die Natur zu imitieren. Der Anwender wünscht sich im Regelfall Programme, die über definierte Eigenschaften verfügen und deren Verhalten vorhersehbar ist. Aus den genannten Gründen haben sich die Entwickler der Programmiersprache Java bewusst gegen die konventionelle Mehrfachvererbung entschieden.

Wie Sie trotzdem mehrere Basisklassen ohne Nebenwirkungen miteinander verbinden können, stellt Ihnen Kapitel 9, »Klassen und Objekte«, Abschnitt 9.5, »Interfaces«, vor.

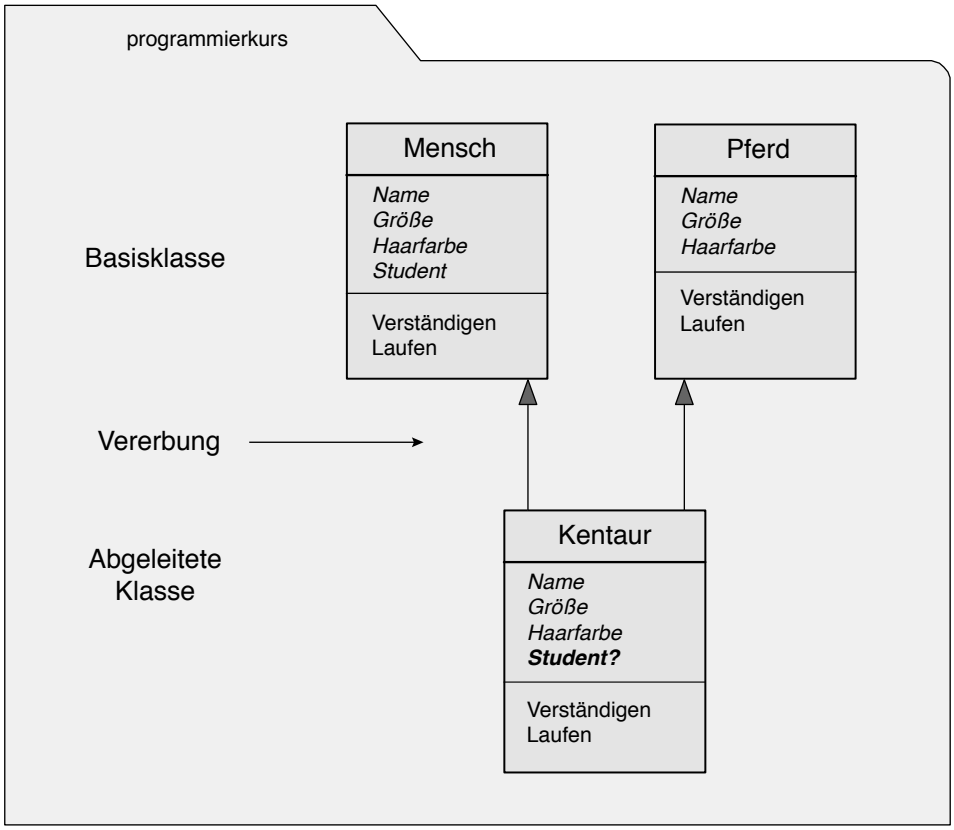


Abbildung 3.14 Mehrfachvererbung am Beispiel einer Kreuzung

3.7 Sichtbarkeit

Eines der wichtigsten Merkmale objektorientierter Sprachen ist der Schutz von Objekten und ihren Attributen vor unerwünschtem Zugriff. Vielleicht erinnern Sie sich noch an den Anfang dieses Kapitels. Die objektorientierte Programmierung wurde erfunden, um die Softwarekrise zu überwinden, die durch fehlerhafte Software ausgelöst wurde. Die Software sollte durch die neue Programmierung robuster werden.

Aus dem Grund besitzt jedes Objekt eine Art von Kapsel, die die Daten und Methoden des Objekts schützt. Die Kapsel versteckt die Teile des Objekts, die von außen nicht oder nur durch bestimmte andere Objekte erreichbar sein sollen. Die Stellen, an denen die Kapsel durchlässig ist, nennen sich *Schnittstellen*. Die Idee des Ganzen ist, nur die Informationen zu einem Objekt durchzulassen, die es unbedingt erreichen müssen. Daher lässt sich jedes Objekt wie eine Zelle gestalten (Abbildung 3.15).

Die wichtigste Schnittstelle einer Klasse ist sein Konstruktor. Nehmen wir als Beispiel die Klasse *Roboter*. Über den Konstruktor dieser Klasse lässt sich das Objekt *Robert* erzeugen. Ein anderes Beispiel für eine solche Schnittstelle ist die Methode *Verständigen* der Klasse

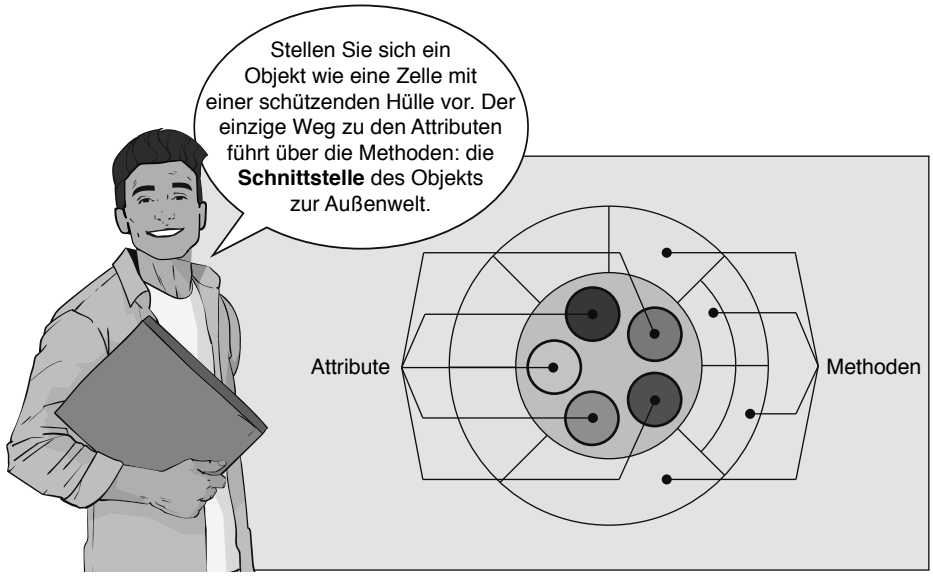


Abbildung 3.15 Wie eine Zelle schützt das Objekt seine Attribute vor unerwünschten Zugriffen.

Roboter. Sie ist öffentlich zugänglich. Im Gegensatz dazu ist seine interne Methode *Addieren* nicht öffentlich zugänglich. Anna kommuniziert mit *Robert* über diese Schnittstelle und teilt darüber *Robert* mit, was er berechnen soll (Abbildung 3.16).

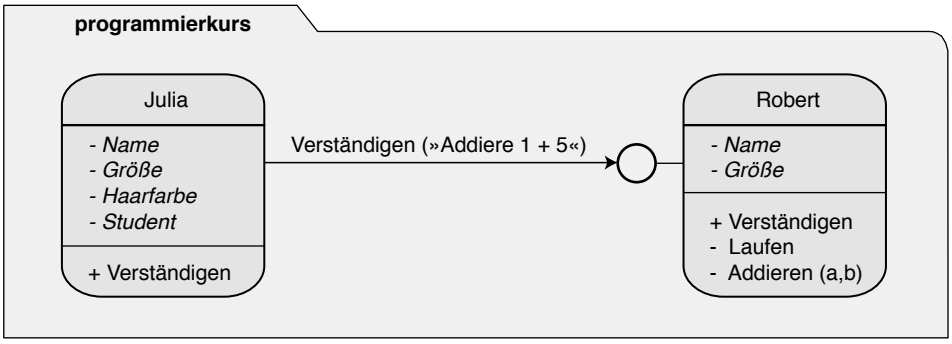


Abbildung 3.16 Objekte kommunizieren nur über ihre Schnittstellen.

Sämtliche Teile, auf die man von außen zugreifen darf, sind mit einem Pluszeichen markiert. Alle Teile, die von außen nicht manipuliert werden können, sind mit einem Minuszeichen versehen. Das Objekt *Anna* darf aber hierbei nicht sämtliche Daten von *Robert* über diese Schnittstelle verändern. Zum Beispiel soll es Anna keinesfalls erlaubt sein, den Namen des Roboters zu ändern. Gäbe es eine öffentlich zugängliche Methode wie zum Beispiel *Umbenennen*, so könnte sie *Robert* damit verändern. So ist das aber nicht gestattet.

3.8 Beziehungen

Klassen und deren Objekte unterhalten in einem Programm die unterschiedlichsten Beziehungen untereinander. In den vorangegangenen Abschnitten haben Sie bereits mehrere Formen der Beziehungen kennengelernt. Grundsätzlich gibt es Beziehungen ohne Vererbung und Beziehungen mit Vererbung.

3.8.1 Beziehungen ohne Vererbung

Die objektorientierte Programmierung nimmt es mit Beziehungen sehr genau. Sie kennt gleich drei verschiedene Arten von Beziehungen ohne Vererbung (Abbildung 3.17).

Abbildung 3.17 Abseits der Vererbungsbeziehung gibt es drei Beziehungstypen.

3.8.1.1 Assoziation

Assoziation ist die einfachste Form einer Beziehung zwischen Klassen und Objekten. Die Abhängigkeiten sind bei dieser Beziehungsart im Vergleich zur Vererbung gering. Man sagt auch, die Objekte sind lose gekoppelt.

Eine Assoziation besteht zum Beispiel, wenn ein Objekt namens *Anna* einem Objekt namens *Robert* die Botschaft *Addieren* sendet (Abbildung 3.18). Die beiden Objekte *Anna* und *Robert* existieren getrennt und erben nichts voneinander.

Abbildung 3.18 Eine einfache Assoziation zwischen Mensch und Roboter

3.8.1.2 Aggregation

Eine Steigerung der Assoziation ist die Aggregation. Eine solche Beziehung besteht dann, wenn sich ein Objekt aus anderen Objekten zusammensetzt. Zum Beispiel soll ein Roboter aus einer nicht näher bestimmten Anzahl von Schrauben bestehen (Abbildung 3.19). Das bedeutet zum Beispiel, dass ein Roboter eine »Besteht-aus-Beziehung« zur Schraube unterhält.

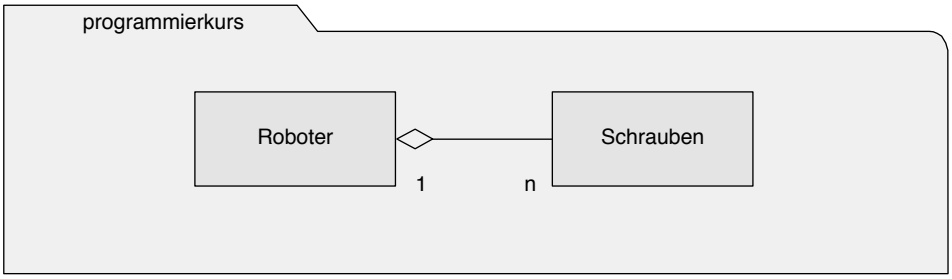


Abbildung 3.19 Aggregation zwischen Roboter und Schraube

Diese Beziehung ist aber von einer völlig anderen Qualität als im vorangegangenen Beispiel zwischen einem Menschen und einem Roboter. Während Mensch und Roboter allein und unabhängig voneinander existieren können, setzt sich der Roboter (unter anderem) aus Schrauben zusammen. Wichtig ist hierbei wieder, dass beide Objekte nichts voneinander erben und jedes Schrauben-Objekt auch allein lebensfähig ist. Dieses Beispiel unterscheidet sich von der strengeren Komposition.

3.8.1.3 Komposition

Die stärkste Form der Beziehungen, die nicht auf Vererbung beruhen, stellt die *Komposition* dar. Wie bei der Aggregation liegt wieder eine »Besteht-aus-Beziehung« vor, sie ist aber im Gegensatz zur Aggregation abermals verschärft. Die Abhängigkeiten sind nochmals stärker.

Ein Beispiel für eine Komposition ist das Verhältnis zwischen einem Mensch und seinen zwei Beinen. Hier besteht glücklicherweise eine sehr enge Beziehung, denn ein Bein ist – im Gegensatz zur Schraube – als selbstständiges Objekt vollkommen sinnlos. Bei der

Erzeugung eines Menschen-Objekts bekommt dieses automatisch zwei individuelle Beine, die im Zusammenhang mit anderen Objekten nicht verwendet werden können.

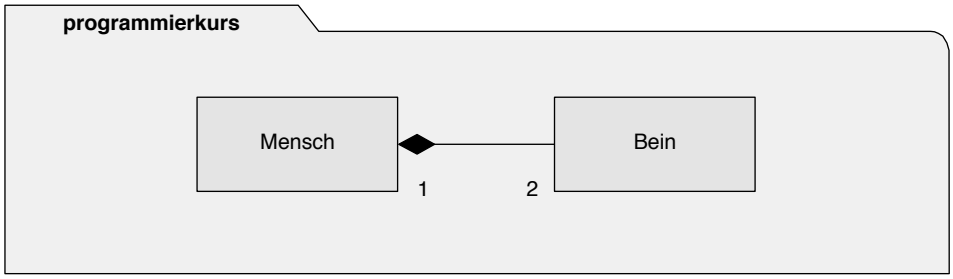


Abbildung 3.20 Ein Mensch und seine zwei Beine als Komposition

Menschenbeine sind also ohne ein geeignetes Objekt der Klasse *Mensch* nicht lebensfähig. Wenn ein Menschen-Objekt stirbt, so sterben auch seine Menschenbeine.

3.8.2 Vererbungsbeziehungen

Vererbungsbeziehungen nennen sich auch Generalisierung (Verallgemeinerung) oder Spezialisierung (Verfeinerung). Dies sind nicht etwa Unterarten der Vererbung, sondern alternative Begriffe für Vererbungsbeziehungen. Welchen der zwei alternativen Begriffe man verwenden möchte, hängt vom Blickwinkel ab, aus dem man die Vererbungsbeziehung betrachtet.

3.8.2.1 Generalisierung

Wenn Sie die Basisklasse aus dem Blickwinkel der abgeleiteten Klasse betrachten wollen, ist Generalisierung der passende Begriff dazu. Zum Beispiel ist die Klasse *Wesen* eine Generalisierung der Klassen *Mensch* und *Roboter*. Mit anderen Worten: Die Klasse *Wesen* ist der allgemeine Begriff (= Generalisierung) für die Klassen *Mensch* und *Roboter*.

3.8.2.2 Spezialisierung

Wenn Sie die abgeleitete Klasse aus dem Blickwinkel der Basisklasse betrachten wollen, ist Spezialisierung der passende Begriff dazu. Zum Beispiel sind die Klassen *Mensch* und *Roboter* eine Spezialisierung der Klasse *Wesen*. Mit anderen Worten: Die Klassen *Mensch* und *Roboter* stellen eine Verfeinerung der Klasse *Wesen* dar.

3.8.2.3 Vererbung kann problematisch sein

Vererbungsbeziehungen stellen eine sehr starke Kopplung zwischen Klassen und damit auch zwischen Objekten her. Eine solch starke Kopplung hat nicht nur Vorteile, sondern auch gravierende Nachteile, wie das folgende Beispiel zeigt:

Eine Klasse namens *Fisch* soll aus der Klasse *Tier* erzeugt werden, die wiederum von *Wesen* abstammt (Abbildung 3.21). Die neue Klasse erbt die Attribute *Name* und *Größe* sowie die

Methoden *Verständigen* und *Laufen*. Moment mal: *Verständigen* und *Laufen*? Hier kommt man ins Grübeln. Können sich Fische verständigen? Vielleicht. Aber laufen, bis auf wenige Ausnahmen, können Fische sicher nicht.

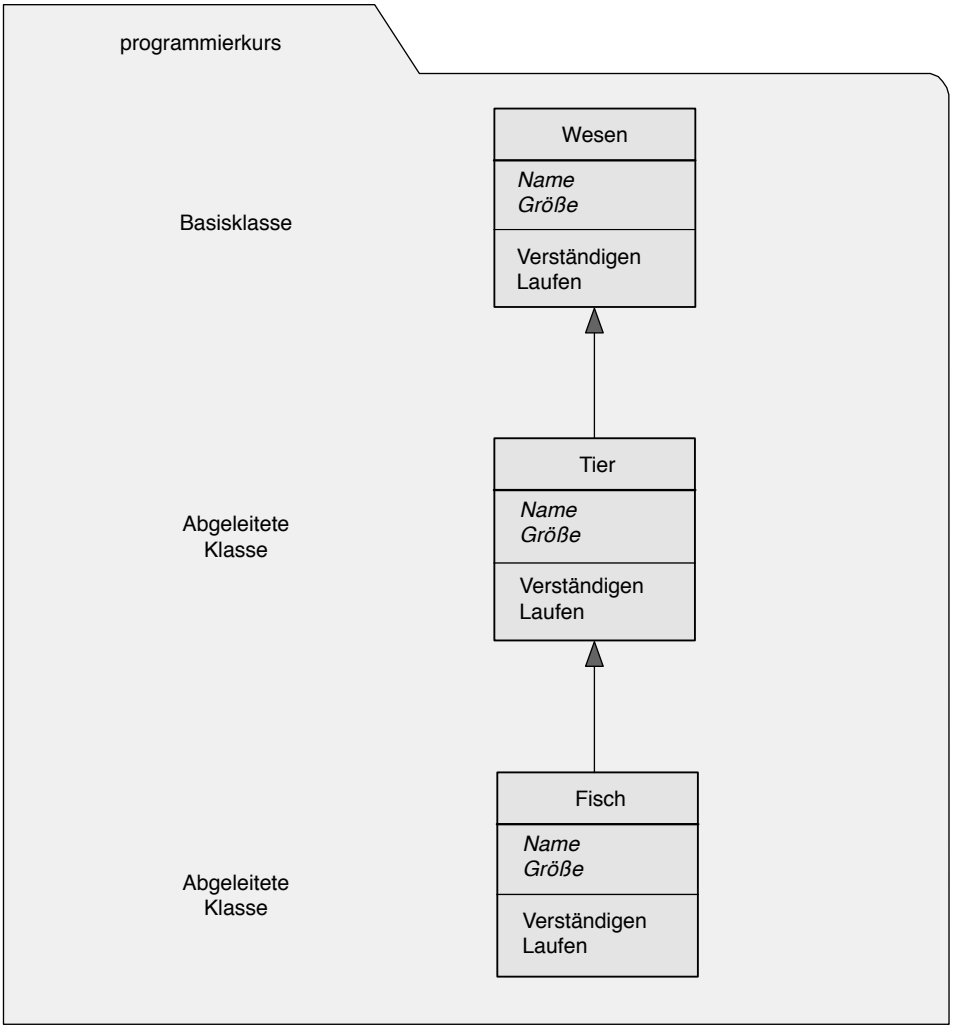


Abbildung 3.21 Durch Vererbung vererben sich auch Designfehler.

Hier ist genau das passiert, was tagtäglich zu den Problemen der objektorientierten Programmierung gehört: Die Funktionalität der Basisklasse ist nicht ausreichend analysiert worden. Vereinfacht gesagt: Hier liegt ein Designfehler vor, den man dadurch beheben muss, dass man zumindest die Methode *Laufen* durch die Methode *Fortbewegen* ersetzt. Aber das hätte für die beiden Klassen *Mensch* und *Roboter* einige Konsequenzen.

3.9 Designfehler

Sie können sich vielleicht vorstellen, dass es sehr unangenehm ist, wenn die Basisklasse aufgrund eines Designfehlers geändert werden muss. Durch die starke Kopplung zwischen Basisklasse und abgeleiteter Klasse pflanzen sich etwaige Änderungen lawinenartig in alle Programmteile fort, in denen Objekte des Typs *Mensch* und *Roboter* mit der Methode *Laufen* verwendet wurden. An allen Stellen des Programms, wo die Methode *Laufen* der Klasse *Wesen* verwendet wurde, muss sie durch die Methode *Fortbewegen* ersetzt werden (Abbildung 3.22)

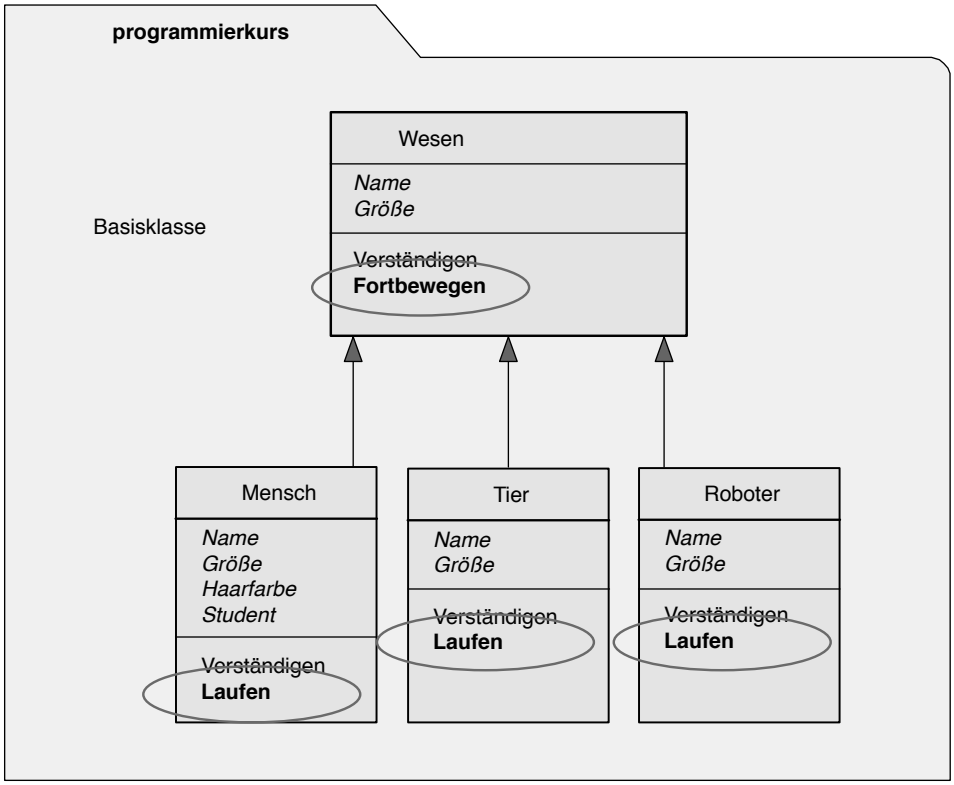


Abbildung 3.22 Durch Redesign lassen sich Fehler bei der Vererbung beheben.

Im Fall von Designfehlern stellt sich die Technik der Vererbung als großer Nachteil heraus. Vererbung hat neben diesem Manko auch den Nachteil, dass sich nicht nur Designfehler, sondern alle anderen vorzüglich gestalteten, aber unerwünschten Teile der Basisklasse in die abgeleiteten Klassen in Form von Ballast übertragen: Die Nachkommen solcher übergewichtiger Klassen werden immer fetter und fetter. Daher sollten Sie Vererbung stets kritisch betrachten, sparsam einsetzen und wirklich nur dort verwenden, wo sie sinnvoll ist.

3.10 Umstrukturierung

Aber zurück zu den Designfehlern. Wie geht man mit Fehlern dieser Art um? Sie sind trotz der Vererbung heute kein so großes Problem mehr wie noch vor ein paar Jahren. Mit einem Werkzeug wie Eclipse ist es relativ leicht, die notwendige Umstrukturierung (Refactoring) vorzunehmen. Allerdings sollten Sie Software möglichst nur während der Analyse- und Designphase der Software umstrukturieren. Als Regel gilt: Je später Änderungen vorgenommen werden, desto höher ist der damit verbundene Aufwand. Kapitel 19, »Entwicklungsprozesse«, beleuchtet das Thema nochmals ausführlicher.

3.11 Modellierung

Um solche Designfehler und damit kostspielige Umstrukturierungen zu vermeiden, ist es bei größeren Projekten sinnvoll, ein Modell der Software zu entwerfen. Genauso wie man im Automobilbau vor jedem neu zu konstruierenden Automobil ein Modell entwickelt, ist es auch in der Softwareentwicklung sinnvoll, ein Modell zu konstruieren, bevor man mit der eigentlichen Umsetzung des Projekts beginnt. Ein Modell, das eine getreue Nachbildung eines kompletten Ausschnitts der Software darstellt, nennt sich Prototyp (Muster, Vorläufer).

3.12 Persistenz

Ein Programm erzeugt Objekte, die an ihrem Lebensende wieder zerstört werden. Diese Objekte bezeichnet man als transient, also flüchtig. Manchmal ist aber ein »Leben nach dem Tod« auch für Objekte erstrebenswert. Sie sollen auch dann wieder zum Leben erweckt werden, wenn das Programm beendet ist und der Anwender des Programms nach Hause geht. Am nächsten Tag startet der Anwender das Programm erneut und möchte mit dem gleichen Objekt weiterarbeiten.

Solche »unsterblichen« Objekte bezeichnet man als persistent (dauerhaft). Das bedeutet nichts anderes, als dass sie in geeigneter Form gespeichert werden. Sie befinden sich dann in einer Art Tiefschlaf in einer Datei auf einer Festplatte oder im Verbund mit anderen Objekten in einer Datenbank.

3.13 Polymorphie

Der Name Polymorphie kommt aus dem Griechischen und bedeutet so viel wie Vielgestaltigkeit, Verschiedengestaltigkeit. Der Begriff klingt mehr nach Mineralienkunde als nach Informatik, und so wundert es Sie vielleicht auch nicht, dass der Chemiker Mitscherlich die Polymorphie bei Mineralien Anfang des 19. Jahrhunderts entdeckte. Er stellte fest, dass

manche Mineralien wie Kalziumcarbonat ($CaCO_3$) unterschiedliche Kristallformen annehmen können, ohne ihre chemische Zusammensetzung zu ändern. Das bedeutet, sie können je nach Druck und Temperatur eine verschiedene Gestalt annehmen.

Alles sehr schön bis jetzt, aber was hat das mit objektorientierter Programmierung zu tun? Das bedeutet auf keinen Fall, dass ein Objekt wie *Robert* so radikal seine Form verändern kann wie ein Mineral. Es bedeutet, dass *Robert* bei geschickter »Programmierung« situationsbedingt verschieden reagieren kann. Klingt wie Zauberei, ist es aber nicht.

3.13.1 Statische Polymorphie

Stellen Sie sich vor, das Objekt *Anna* teilt dem Objekt *Robert* mit, dass *Robert* eine Addition mit zwei ganzzahligen Werten durchführen soll. Was wird passieren? – Natürlich ist es für *Robert* kein Problem, diese Werte zu addieren und *Anna* das Ergebnis zu nennen. Was würde aber passieren, wenn *Anna* abermals *Robert* mitteilt, er solle addieren, und zwar mit Bruchzahlen? Entweder würde *Robert* die Aufgabe zerlegen oder er würde eine Addition direkt mit Bruchzahlen durchführen, weil er eine interne Methode dafür hätte (Abbildung 3.23).

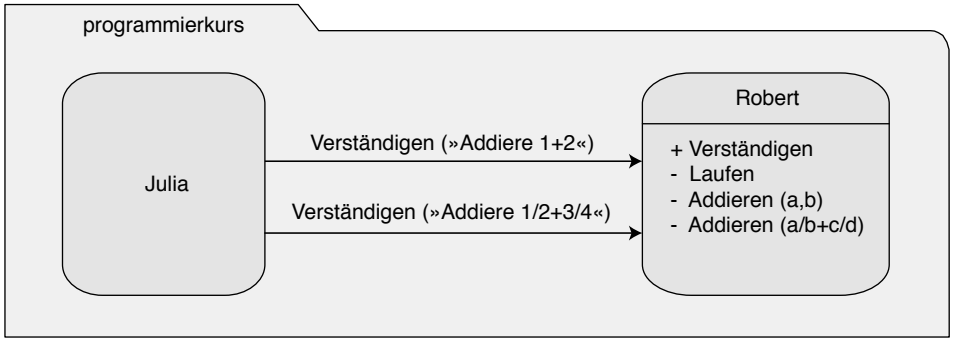


Abbildung 3.23 »Robert« verfügt über zwei verschieden gestaltete Methoden namens »Addieren«.

Damit *Robert* den verschiedenen Anweisungen von *Anna* Folge leisten kann, benötigt er Methoden »unterschiedlicher Gestalt«. Er benötigt eine Methode, die auf zwei Parameter Geschwindigkeit reagiert, und eine Methode, die auf drei Parameter reagiert. Obwohl die Methoden den gleichen Namen tragen, führen sie zu einer unterschiedlichen Verarbeitung durch das Objekt *Robert*. Der Fachausdruck für diese Technik heißt *Überladen*.

3.13.2 Dynamische Polymorphie

Anders als bei der Mehrfachvererbung sieht es aus, wenn man Eigenschaften der Basisklasse bei der Vererbung bewusst umgehen möchte. Dazu möchte ich nochmals auf das Beispiel der Basisklasse *Wesen* zurückgreifen. Angenommen, Sie möchten in der abgeleiteten Klasse *Roboter* bestimmen, auf welche Weise sich Roboter-Objekte verständigen. Dazu *überschreiben* Sie die Methode *Verständigen* und legen die Art und Weise des Verständigens in der Klasse *Roboter* für die abgeleiteten Objekte fest.

Das Überschreiben von Methoden ist ein sehr mächtiges Mittel der objektorientierten Programmierung. Es erlaubt Ihnen, unerwünschte Erbinformationen teilweise oder ganz zu unterdrücken und damit eventuelle Designfehler – in Grenzen – auszugleichen beziehungsweise Lücken in der Basisklasse zu füllen. Dabei ist die Technik extrem simpel. Es reicht aus, eine identische Methode in der abgeleiteten Klasse *Roboter* zu beschreiben, damit sich Objekte wie *Robert* »plötzlich« anders verhalten.

3.14 Designregeln

Objektorientierte Programme sind keine Garantie für sauber strukturierte und logisch aufgebaute Programme. Die objektorientierte Programmierung erleichtert gutes Softwaredesign, sie erzwingt es jedoch nicht. Daher sollten Sie einige Grundregeln beachten. Sie erleichtern es Ihnen, schlanke, schnelle und robuste Java-Programme zu entwickeln (Abbildung 3.24).

```
Designregeln {
• Vermeiden Sie Vererbung
• Reduzieren Sie die Anforderungen
  auf das Wesentliche
• Kapseln Sie alle internen
  Attribute und Methoden
• Zeichnen Sie ein fachliches Modell
  mit allen Abhängigkeiten
• Arbeiten Sie mit einem Prototyp }
```

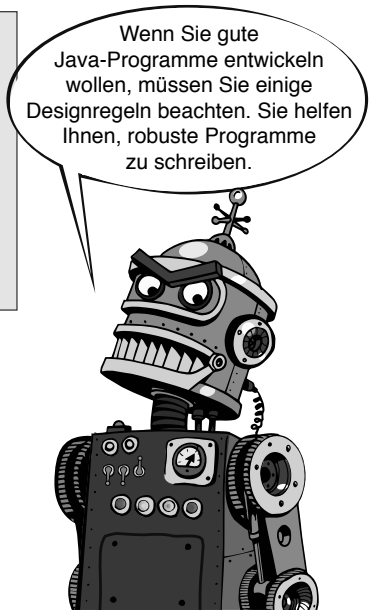


Abbildung 3.24 Robert empfiehlt einige Grundregeln für gutes Design.

3.15 Zusammenfassung

Die objektorientierte Programmierung war eine Antwort auf die Softwarekrise in der Mitte der 60er-Jahre des letzten Jahrhunderts. Durch Objektorientierung lässt sich die natürliche Welt leichter in Computerprogrammen umsetzen. Diese objektorientierten Computerprogramme bestehen aus einem oder mehreren Objekten.

Die Objekte haben ihren eigenen Speicher. Klassen modellieren das gemeinsame Verhalten ihrer Objekte. Jedes Objekt ist hierbei ein Exemplar seiner Klasse. Objekte verständigen sich über Nachrichtenaustausch. Ein Programm wird ausgeführt, indem dem ersten Objekt die Kontrolle übergeben und der Rest als dessen Nachricht behandelt wird.

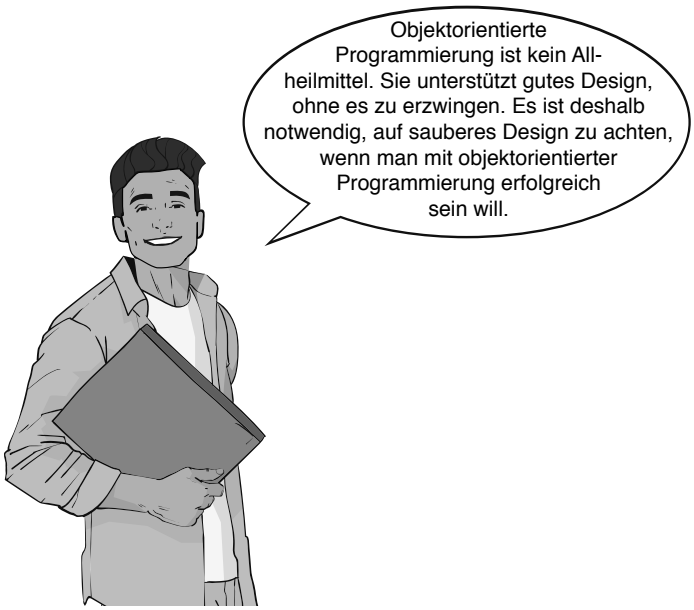


Abbildung 3.25 Florian fasst die Besonderheiten der objektorientierten Programmierung zusammen.

Ein Objekt lässt sich mit einem natürlichen Lebewesen vergleichen und verfügt über eine Gestalt und Fähigkeiten. Die Gestalt prägen Attribute, während die Fähigkeiten von Methoden bestimmt sind. Beide Bestandteile eines Objekts sind in der Klasse festgelegt, von der ein Objekt abstammt. Sie liefert den Bauplan für gleichartige Objekte.

3.16 Aufgaben

- Aufgabe 1: Konzipieren Sie eine Adressdatenbank der Studenten und Studentinnen sowie Mitarbeiter einer Hochschule. Die Datenbank soll dem Sekretariat helfen, Personen zu finden und sie anzuschreiben oder anzurufen. Orientieren Sie sich hierbei an Abbildung 3.26. Überlegen Sie sich passende Personenobjekte mit Attributen für Mitarbeiter sowie Studenten und zeichnen diese in ein Objektdiagramm.

Abbildung 3.26 Verschiedene Personenobjekte an einer Hochschule

- Aufgabe 2: Leiten Sie aus den Personenobjekten eine oder mehrere Klasse(n) mit passenden Attributen ab und zeichnen diese in ein Klassendiagramm.
- Aufgabe 3: Versuchen Sie, aus den gefundenen Klassen eine oder mehrere gemeinsame Basisklasse(n) mit gemeinsamen Attributen zu entwickeln. Ergänzen Sie das Klassendiagramm und begründen das Design.

Die Lösungen zu den Aufgaben finden Sie in Kapitel 25, »Lösungen«, ab Seite 586.

3.17 Literatur

Wikipedia: Alan Kay; *https://de.wikipedia.org/wiki/Alan_Kay*

Stichwortverzeichnis