Kapitel 2

Programmierung als Kern der Softwareentwicklung

Die Programmierung ist der Kern eines jeden Entwicklungsprozesses. In diesem Teilschritt wird das eigentliche Programm erstellt. Als Entwickler stehen uns dazu unterschiedlichste Vorgehensweisen zur Verfügung. Ebenso gibt es für unterschiedliche Problemstellungen und Lösungsansätze verschiedene Programmiersprachen.

In diesem Handbuch der Softwareentwicklung erhalten Sie einen umfassenden Überblick über alle wichtigen Teilbereiche dieser Aufgabe. Sie werden noch erfahren, dass die Entwicklung von Software ein sehr vielfältiger Prozess von der Planung bis zur Auslieferung an den Kunden ist. Das Kernelement bleibt jedoch die eigentliche Programmierung, d. h. die Erstellung des lauffähigen Computerprogramms. In den Anfängen der Softwareentwicklung wurden die einzelnen Schritte auch nicht streng voneinander getrennt. Vielmehr hatte der Entwickler eine Idee oder einen Auftrag, hat sich an den Rechner gesetzt und daraus ein Computerprogramm »gestrickt«. Diese Vorgehensweise ist für heutige komplexe Probleme natürlich nicht mehr geeignet.

In diesem Kapitel wollen wir klären, was man eigentlich unter Programmierung versteht. Ebenso benötigen wir einen Überblick darüber, welche grundsätzlichen Vorgehensweisen uns dafür zur Verfügung stehen. Welche Programmiersprachen gibt es überhaupt? Und kann man sie systematisieren? Trotz aller Unterschiede in den Sprachen gibt es so etwas wie einen Grundstock an essenziellen Merkmalen, der in nahezu allen Programmiersprachen vorhanden ist. Als angehender professioneller Softwareentwickler müssen Sie diese einfach draufhaben. Wenn wir Sie um 2:30 Uhr wecken und nach den Unterschieden zwischen der while- und der do-Schleife fragen, dann muss das wie aus der Pistole geschossen kommen. Lassen Sie uns direkt beginnen! Was genau versteht man eigentlich unter der Programmerstellung?

2.1 Die Programmierung

Was ist eigentlich ein Computerprogramm? Um sich mit dieser Frage genauer auseinanderzusetzen, betrachten Sie am besten Abbildung 2.1.

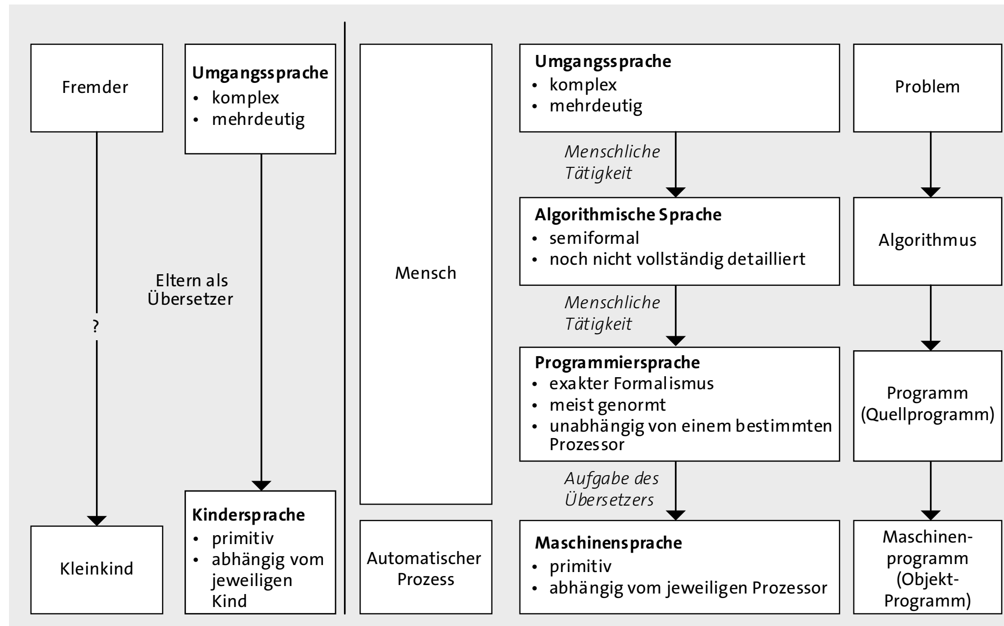

Abbildung 2.1 Die Mensch-Maschine-Kommunikation erfolgt über eine Programmiersprache. (Quelle: Balzert, 2005)

Das Ziel eines jeden Softwareentwicklungsprozesses ist die Bereitstellung von Programmen. Dabei spielt es vom Grundsatz zunächst keine Rolle, auf welchem System ein solches Computerprogramm laufen soll. Neben den klassischen Programmen für den Desktop spielen heute Anwendungen für die sogenannten mobilen Endgeräte wie Smartphones und Tablets eine zunehmende Rolle. Zur Entwicklung von Computerprogrammen bedient man sich einer Programmiersprache. Ausgangspunkt ist stets ein Problem der realen Welt. Beispielsweise sollen Berechnungen durchgeführt werden, oder die Daten der Kunden sind systematisch zu verwalten. Auch Computerspiele fallen in diese Kategorie.

Als Erstes muss das Problem analysiert werden. Mithilfe eines oder mehrerer Algorithmen wird es dann gelöst. Unter einem Algorithmus versteht man eine detaillierte und explizite Vorschrift zur schrittweisen Lösung des Problems. Es kann sich dabei um einfachste Rechen- oder Zählvorgänge handeln oder um komplexe Abläufe, wie zum Beispiel die Suche nach dem kürzesten Weg auf einer Karte. In den Algorithmen wiederum werden Daten verarbeitet. Die Themen Algorithmen und Daten bzw. Datenstrukturen sind essenziell für die Entwicklung von Computerprogrammen. Daher kümmern wir uns in Kapitel 3, »Algorithmen und Datenstrukturen«, ausschließlich darum.

Kommen wir jedoch nochmals auf Abbildung 2.1 zurück. Ein *Computerprogramm* dient also stets dazu, ein bestimmtes Problem zu lösen. Diese Lösung fällt jedoch nicht vom Himmel, sondern muss systematisch von Ihnen als Softwareentwickler erarbeitet werden. Damit der Computer das Problem bearbeiten kann, muss es mithilfe einer Programmiersprache in eine Form übertragen werden, die der Rechner versteht und abarbeiten kann.

Programmiersprachen sind mehr oder weniger an den Bedürfnissen der Problemstellung ausgerichtet und erlauben damit eine problemorientierte Formulierung der Lösung. Der Computer selbst versteht lediglich die Maschinensprache seines Prozessors, d. h. eine Folge von Nullen und Einsen. Die Schritte »Analyse des Problems«, »Formulierung in einer uns vertrauten Umgangssprache« und »Entwicklung eines Lösungsalgorithmus« sowie die letztendliche Codierung in einer konkreten Programmiersprache sind zum heutigen Stand der Technik vollständig nur durch Menschen zu leisten. Am Ende dieses Vorgangs liegt das Computerprogramm in Form eines Quelltexts vor. Dieser Quelltext wird dann in einem letzten Schritt durch einen *Compiler* bzw. *Interpreter* in die Maschinensprache transformiert. Die Maschinensprache wird von dem im Computer eingebauten Prozessor verstanden.

Letztendlich kann der gesamte Vorgang des Programmierens auch in Analogie zum Kommunikationsprozess zwischen Menschen aufgefasst werden. Auch dabei geht darum, dass eine erste Person einen Sachverhalt in Form einer Botschaft ausdrückt und versendet und eine zweite Person diese Botschaft entschlüsselt und entsprechende Handlungen daraus ableitet. Betrachten Sie dazu die Kommunikation eines Erwachsenen mit einem Kleinkind. Das Kleinkind hat nur einen begrenzten Wortschatz, den es versteht. Als Eltern kennt man die Syntax dieser hochindividuellen Sprache genau, und man kann sich mit seinem Kind verständigen. Als fremde Person ist es nicht möglich bzw. deutlich schwerer, da man den konkreten Wortschatz des Kindes nicht kennt. Ein Beispiel gefällig: Klein Herbert »brüllt« seit gefühlten fünf Minuten auf Opa ein, dass dieser ihm bitte den »Nüsselauto« überlassen soll. Opa weiß sich nicht zu helfen, auch die herbeieilende Oma Ute kann nicht für Abhilfe sorgen. Erst als der Vater von Herbert kommt, übersetzt er für die Großeltern: »Nüsselauto« heißt in korrekter deutscher Notation »Autoschlüssel«. Der Vater fungierte hier gewissermaßen als Interpreter zwischen dem Quelltext von Klein Herbert und der deutschen Sprache.

Der Übersetzer (Compiler) bestimmt also, für welche Zielsysteme, d. h. für welche Hard- und Software, das Programm erstellt wird. Er ist dabei unter anderem auf Betriebssystem und Prozessor abgestimmt (Abbildung 2.2).

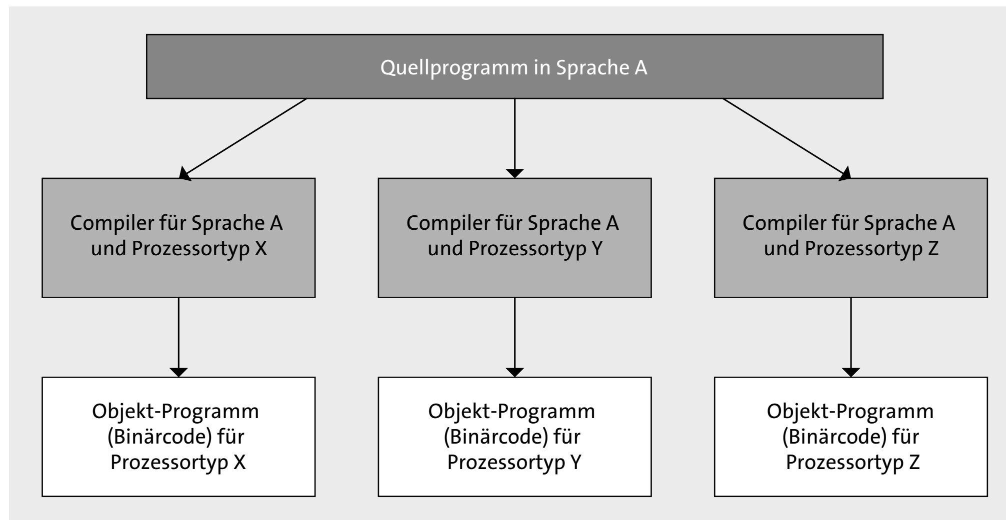

Abbildung 2.2 Der Übersetzungsvorgang erfordert einen individuellen Compiler für jeden Prozessortyp. (Quelle: Balzert, 2005)

Für eine Programmiersprache A müssen daher unterschiedliche Compiler erstellt werden, um die Programme für unterschiedliche Prozessortypen zu generieren.

Und was ist nun der Unterschied zwischen einem Compiler und einem Interpreter? Ein *Compiler* übersetzt den Quellcode bereits zur Entwicklungszeit in die jeweilige Maschinensprache. Zur Laufzeit wird das Programm dann lediglich ausgeführt. Ein *Interpreter* dagegen übersetzt den Quellcode erst zur Laufzeit in die Maschinensprache des jeweiligen Prozessors. Programme von kompilierten Programmiersprachen sind damit in der Regel schneller in der Ausführung. Andererseits gelingt es mit einer interpretierenden Sprache leichter, das Programm zwischen verschiedenen Systemen zu transferieren. Moderne Programmiersprachen mischen die beiden Konzepte. Dabei wird der ursprüngliche Quelltext während der Entwicklungszeit in eine Art Zwischensprache (*Intermediate Language*) übersetzt. Zur Laufzeit wird dann dieser Zwischencode in das endgültige Programm auf der jeweiligen Plattform transferiert.

Nach diesen Ausführungen, was man unter Computerprogrammen versteht und wie diese entstehen, geht es jetzt bereits einen Schritt weiter in Richtung eigentliche Programmentwicklung. Im gesamten Handbuch erlernen Sie vielfältige Methoden und Vorgehensweisen, wie heute auf professionelle Art und Weise Software entwickelt wird. Gleich zu Beginn sollte man sich einen Überblick darüber verschaffen, welche grundsätzliche Herangehensweisen überhaupt existieren, um ein Problem aus der realen Welt in eine Ausdrucksweise zu überführen, die man mithilfe des Computers lösen kann. Es geht um die sogenannten *Paradigmen* der Softwareentwicklung.

2.2 Paradigmen der Softwareentwicklung

Die Anfänge der Programmentwicklung waren durch ein intuitives »Drauflos-Programmieren« gekennzeichnet. Auf diese Weise lassen sich natürlich heute in keiner Weise mehr die Anforderungen an die Erstellung von hochkomplexer und möglichst fehlerfreier Software erfüllen. Die Art und Weise der Softwareentwicklung hat sich im Laufe der Zeit zu einem ingenieurmäßigen Vorgehen entwickelt. Vorgehensmodelle beschreiben, in welcher zeitlichen Reihenfolge die einzelnen Schritte des Entwicklungsprozesses durchlaufen werden. Methoden des Projektmanagements sorgen auf höherer Ebene dafür, dass der gesamte Prozess weitgehend planbar ist. Beide Themen behandeln wir übrigens ausführlich in Kapitel 4, »Softwareprojekte professionell planen«. Eine sehr wichtige Frage bei der Programmierung ist die Art und Weise, wie eine Aufgabe aus der realen Welt in Software abgebildet und damit bearbeitet wird. Dabei kann ein *Programm* als ein Modell der realen Welt aufgefasst werden. Je nach Problem existieren unterschiedliche Ansätze zum Aufbau dieses Modells. Diese Ansätze werden als *Programmierparadigmen* bezeichnet. Dabei wird zwischen den folgenden grundsätzlichen Paradigmen unterschieden:

- *Imperativ*: Das Programm besteht aus einer Folge von Anweisungen, die streng sequenziell abgearbeitet werden. Das Konzept des imperativen Programmierparadigmas beruht auf Funktionen und Prozeduren zur Abbildung der Funktionalität. Der bekannteste Vertreter ist die Programmiersprache C.
- *Objektbasiert*: Diese Programmiersprachen kennen Objekte, welche Daten und die zugehörigen Funktionen zu einer Einheit, d. h. zu einem Objekt zusammenfassen. Zum Beispiel hat das Objekt »Auto« wichtige Eigenschaften wie Größe und Anzahl der Türen. Die zugehörigen autotypischen Funktionen sind beispielsweise »beladen« oder »tanken«. Weitergehende Konzepte, wie beispielsweise *Vererbung* oder die Abbildung von Beziehungen zwischen Objekten, werden jedoch nicht angeboten. Objektbasierte Sprachen können damit als Vorstufe der objektorientierten Programmierung aufgefasst werden. Ein Beispiel ist die Scriptsprache Microsoft PowerShell.
- *Objektorientiert*: Programmiersprachen dieser Gattung erweitern das objektbasierte Programmierparadigma um typische Konzepte der Objektorientierung wie zum Beispiel die Vererbung. Fast alle modernen Programmiersprachen unterstützen diese Form der Entwicklung. Es ist heutzutage die am meisten angewendete Vorgehensweise. Als angehender professioneller Softwareentwickler müssen Sie die dahinterstehenden Konzepte vollständig verstanden haben. Im nächsten Abschnitt wird es um nichts anderes gehen.
- *Funktional*: Ein funktionales Programm besteht nur aus einer Reihe von Funktionsaufrufen. Nahezu alle Elemente können dabei als Funktionen aufgefasst werden. Einsatzgebiete sind Anwendungen der künstlichen Intelligenz, Compilerbau

und Computeralgebra-Systeme. Beispiele funktionaler Programmiersprachen sind *Lisp*, *Haskell*, *F#* und *Scala*.

- *Logisch*: Hier steht im Mittelpunkt der Aufbau einer Datenbasis, die aus Fakten und Regeln besteht. Fakten sind dabei wahre Aussagen im Sinne der Mathematik. Im Fokus steht die Problemformulierung, nicht der Lösungsalgorithmus. Die Sprache *Prolog* basiert auf dem logischen Paradigma.
- *Deklarativ*: Es ist der Überbegriff für das funktionale und das logische Programmierparadigma.

Es ist wichtig, dass Sie sich bereits jetzt folgende Aussage merken: *Computerprogramme können grundsätzlich nach unterschiedlichen Paradigmen aufgebaut werden*. Ein »falsch« oder »richtig« gibt es dabei nicht. Vielmehr kann man lediglich eine mehr oder weniger gute Eignung des einen oder des anderen Ansatzes feststellen. Moderne Programmiersprachen unterstützen meist auch nicht nur ein Paradigma, sondern mehrere Ansätze können miteinander kombiniert werden. Dazu ein Beispiel: Die Sprache C# ist vom Wesen her eine objektorientierte Programmiersprache. Dennoch ist bekannt, dass sich bestimmte Teile einer Programmieraufgabe, beispielweise mathematisch orientierte Probleme, eleganter unter Anwendung funktionaler Aspekte lösen lassen. Dem Programmierer stehen daher innerhalb der Sprache C# auch Elemente der funktionalen Programmentwicklung zur Verfügung. Ob er diese auch tatsächlich nutzt, bleibt seine individuelle Entscheidung, denn grundsätzlich würde man auch zur Lösung gelangen, wenn man ausschließlich die objektorientierte Programmierung anwendet. Entscheidet man sich hingegen für eine Implementierung in F# (einer funktionalen Programmiersprache), so kann man bei Bedarf auch auf objektorientierte Konzepte zurückgreifen. Diese sind auch in F# vorhanden, auch wenn das nicht der Schwerpunkt dieser Sprache ist.

Insgesamt kann man aus heutiger Perspektive sagen, dass sich die objektorientierte Programmierung seit Langem etabliert hat und heute als State of the Art in vielen Bereichen der Softwareentwicklung gilt. Im kommenden Abschnitt stellen wir die wichtigsten Konzepte der objektorientierten Programmierung vor.

2.3 Objektorientierte Programmentwicklung

In diesem Abschnitt erfahren Sie alles Wichtige zum Konzept der objektorientierten Programmierung. Es ist unseres Erachtens eines der essenziellen Grundpfeiler der modernen Programmentwicklung. Nahezu alle modernen Programmiersprachen unterstützen dieses Paradigma in einer Form. Für einen sanften Einstieg stellen wir Ihnen Grundideen und -konzepte zunächst weitgehend sprachneutral vor. In den späteren Ausführungen zu einzelnen Programmiersprachen werden diese konkreter.

2.3.1 Objektorientierung im Überblick

Die *objektorientierte Programmierung* basiert darauf, Elemente der Software, in Anlehnung an die Gegebenheiten der realen Welt, als Objekte aufzufassen. Solche Objekte verfügen über Eigenschaften, die sie genauer charakterisieren. Ebenso können Objekte bestimmte Tätigkeiten ausführen. Objekte können miteinander kommunizieren und in einer bestimmten Beziehung zueinander stehen. Beispielsweise weisen ähnliche Objekte einen gleichen Bauplan auf. Machen wir das an einem Beispiel etwas konkreter: Ein Automobil ist je nach Sichtweise ein mehr oder weniger komplexes Objekt, das sich durch *Attribute* (*Eigenschaften*) beschreiben lässt, zum Beispiel

- Hersteller,
- Typ,
- Farbe und
- Zulassungsjahr.

Es gibt weitere Attribute, die uns aber bei der Abbildung als Modell für eine Software nicht interessieren. Wollen wir beispielsweise eine Verwaltung für den Fuhrpark eines Unternehmens programmieren, so müssen Sie die dafür relevanten Eigenschaften der Fahrzeuge erfassen. Dies könnten die eben genannten Attribute sein. Nicht von Interesse für die Software ist dagegen, ob das Auto eine schöne Fußmatte hat und ob im Handschuhfach eine Beleuchtung eingebaut ist.

Die ausgewählten Eigenschaften repräsentieren also die Daten des Objekts. Ebenso verfügt ein Auto stets über bestimmte Fähigkeiten, d. h., es kann bestimmte Funktionen bzw. Aufgaben ausführen. Diese Fähigkeiten werden mit Blick auf die objektorientierte Programmierung als *Methoden* bezeichnet. Mit Bezug auf die zu entwickelnde Software könnten folgende Methoden von Interesse sein

- `BehördlichZulassen`, d. h. Anmeldung bei der zuständigen Behörde, und
- `FahrtenbuchAnalysieren`.

Natürlich verfügt das reale Objekt über viel mehr Funktionalität. Ein Automobil kann beispielsweise beschleunigen und bremsen. In der Software bilden wir jedoch nur die interessierenden Methoden ab. Wir erinnern uns: Software ist ein stark vereinfachtes Modell der Wirklichkeit. Fassen wir hier bereits die ersten Erkenntnisse zusammen: Unter *Objekten* versteht man die Zusammenfassung von Attributen und Methoden.

Der nächste Schritt besteht im Übergang von Objekten zu Klassen. Hier gilt: Ein konkretes Objekt gehört zu einer Klasse oder andersherum betrachtet, eine *Klasse* ist der Bauplan für die zugehörigen Objekte. Mit Blick auf unser Beispiel haben wir eine allgemeine Klasse `Auto` vorliegen. Nun, jeder weiß, was er sich unter einem Auto vorstellen muss. Andererseits heißt es auch, ein allgemeines Auto gibt es nicht! Konkretisiert man ein Auto anhand seiner spezifischen Attribute, so gelangt man zum Objektbegriff. Konkret: Das Fahrzeug mit dem Kennzeichen EF-LN 88 ist ein konkretes Objekt der Klasse `Auto`. Dabei weist es die folgenden Werte der oben genannten Attribute auf:

```
Hersteller = "Toyota";
Typ = "Avensis";
Farbe = "grau";
Zulassungsjahr = 2012;
```

Ein anderes Objekt der Klasse Auto hat andere Werte bezüglich dieser Attribute, beispielsweise:

```
Hersteller = "VW";
Typ = "Golf";
Farbe = "rot";
Zulassungsjahr = 2014;
```

Beide Objekte verfügen also über die gleichen Attribute, aber gegebenenfalls über andere Werte dieser Attribute. Sowohl der Toyota als auch der VW kennen die beiden oben genannten Methoden `BehördlichZulassen` und `FahrtenbuchAnalysieren`.

Methoden geben nicht nur an, was die konkreten Objekte einer Klasse können bzw. was man mit ihnen machen kann, sondern Methoden werden auch dazu verwendet, die Werte bestimmter Attribute zu verändern. Auch hier wieder ein einfaches Beispiel: Eine Methode `Lackieren` der Klasse `Auto` könnte also eine Veränderung der Farbe bewirken. Als *Argument* müsste die Methode die neue Farbe übermitteln. Aus unserem grauen Toyota könnte also mithilfe der Methode `Lackieren` ein farbenfrohes blaues Automobil werden. Ebenso dienen Methoden dazu, dass man die Werte von Attributen abfragen kann. Wir haben all das in Abbildung 2.3 für Sie zusammengefasst.

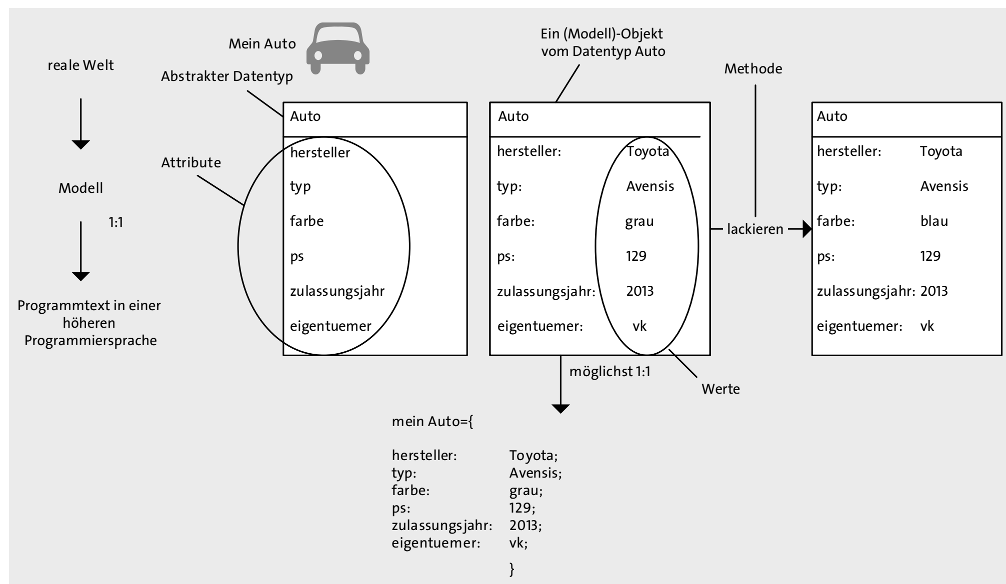

Abbildung 2.3 Klassen sind Baupläne für Objekte als Modelle der realen Welt.

Bitte beachten Sie: In welcher Form man die Werte von Attributen ändern kann, ist von Programmiersprache zu Programmiersprache etwas unterschiedlich. Historisch und in Anlehnung an die Grundidee der objektorientierten Programmierung kann man von außerhalb eines Objekts nicht auf die Attribute des Objekts direkt zugreifen, sie sind geheim. Dazu verwendet man Methoden. Wie man diese Methoden benennt, spielt grundsätzlich keine Rolle. Zur Vereinfachung dieser Notwendigkeit verwenden viele Sprachen spezielle Methoden, d. h. für das Lesen eine `get`- und für das Schreiben eine `set`-Methode. Beispielsweise könnte man mit der Methode `setFarbe` die Farbe des bestreffenden Autoobjekts seinen Wünschen anpassen. Wollte man die aktuelle Farbe wissen, müsste man den Wert über die Methode `getFarbe` auslesen. Ebenso könnte man – wie im obigen Beispiel angeführt – eine Methode `Lackieren` erstellen, die auch die Farbe anpasst und vielleicht darüber hinaus noch etwas anderes macht, zum Beispiel eine Schutzversiegelung auf den Lack aufbringt.

Da ständig auf die Attribute der Objekte lesend und schreiben zugegriffen werden muss, haben einige Programmiersprachen, zum Beispiel C#, noch mehr Vereinfachungen vorgesehen. Hier definiert man sogenannte `Properties`. Dabei kann man von außerhalb der Klasse so tun, als ob man direkt das betreffende Attribut liest oder schreibt. Im Hintergrund werden dazu jedoch weiterhin `get`- und `set`-Methoden aufgerufen. Um den Entwicklern das Leben jedoch zu erleichtern, werden diese `set`- und `get`-Methoden vom Compiler automatisch bei der Erstellung des Programms generiert. Wir kommen auf dieses Thema in Abschnitt 2.5.4, »Objektorientierung«, zurück.

Untereinander kommunizieren die Objekte miteinander über *Nachrichten*. Ein Objekt *A* kann auf diese Weise bei einem Objekt *B* eine Funktion auslösen, indem es eine Methode aufruft oder den Wert eines Attributs erfragt.

Wie gesagt, der konkrete Umgang mit der Objektorientierung ist von Programmiersprache zu Programmiersprache unterschiedlich. Spätestens wenn Sie ein konkretes Programm erstellen, werden Sie sich damit auseinandersetzen müssen. Das Grundverständnis von Klassen und ihren Objekten bleibt jedoch stets identisch.

Objekte sind also konkrete Ausprägungen einer Klasse. Dabei können von einer Klasse in der Regel beliebig viele Objekte, auch als *Instanzen* bezeichnet, erstellt werden. Die einzelnen Objekte können sich also in den konkreten Werten der Attribute unterscheiden. Wir haben es am obigen Beispiel der Klasse Auto deutlich gemacht, indem wir die Objekte Toyota Avensis und VW Golf erstellt haben. So weit, so gut!

Wir möchten an dieser Stelle noch auf die Möglichkeit der Definition von Ereignissen von Objekten eingehen. Streng betrachtet gehört dieses Konzept nicht zu den Grundpfeilern der objektorientierten Programmierung, sondern zum Konzept der *ereignisorientierten Entwicklung* von Programmen. Es ist jedoch mit dem Konzept der Klassen und Methoden eng verbunden, sodass wir es jetzt an dieser Stelle erklären möchten. Ein *Ereignis* wird von einem Objekt in einer bestimmten Situation ausge-

löst. Beispielsweise könnte ein Objekt der Klasse Auto ein Ereignis auslösen, wenn in den nächsten vier Wochen der TÜV abläuft. Die Software könnte in diesem Fall mit einem Hinweis den Bearbeiter auf diesen Umstand aufmerksam machen. Grundsätzlich stellen Ereignisse die Grundlagen für die Interaktionsfähigkeit der Anwendung dar. Ereignisse werden zum Beispiel auch dann ausgelöst, wenn der Nutzer auf ein Element der Benutzeroberfläche, zum Beispiel auf einen Button, klickt. Das zugehörige Ereignis heißt in diesem Fall zum Beispiel `ButtonClick`-Ereignis. Andere typische Ereignisse, die für die Reaktion der Anwendung auf Aktionen des Nutzers darstellen, sind beispielsweise Wischgesten bei einer Anwendung für Smartphone oder Tablet bzw. Mausbewegungen bei einem Zeichenprogramm. Um gegenseitig auf Ereignisse zu reagieren, registrieren sich die Objekte untereinander. Beispielsweise kann ein Objekt *A* sich beim Objekt *B* für ein bestimmtes Ereignis registrieren. Löst Objekt *A* dieses Ereignis aus, so bekommt Objekt *B* das mit und kann seinerseits ein bestimmtes Verhalten auslösen.

Das Ganze erscheint Ihnen etwas unverständlich. Damit das nicht so bleibt, betrachten wir dazu Abbildung 2.4.

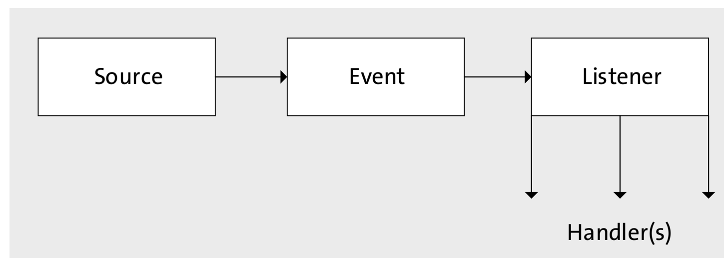

Abbildung 2.4 Das Prinzip von Ereignissen bei der objektorientierten Programmierung

Auf der linken Seite haben wir die Quelle (engl. *Source*), in Form eines konkreten Objekts einer Klasse. Diese Quelle feuert in einer bestimmten Situation ein Ereignis (engl. *Event*) ab. Beispielsweise könnte die Quelle der besagte Button sein, der auf das Klicken durch den Nutzer reagiert. Andere Objekte – auch als *Listener* bezeichnet – »lauschen« darauf, dass das betreffende Ereignis ausgelöst wird. Ist dies der Fall, dann reagieren sie in vorgesehener Art und Weise. Dabei können mehrere Listener parallel das Auslösen eines bestimmten Ereignisses überwachen. Ein Objekt der Klasse `Berechnungen` könnte eine neue Berechnung starten, sobald der Nutzer auf den Button geklickt hat. Gleichzeitig könnte ein Objekt der Benutzeroberfläche darauf reagieren und dem Nutzer mitteilen, dass nunmehr der Rechenvorgang gestartet wurde.

Und wie lange existiert ein einmal erstelltes Objekt? Nun, die Antwort ist recht einfach. Nachdem ein Objekt einer Klasse durch einen *Konstruktor* erstellt wurde, wird es im Speicher vorgehalten. Es sollte mindestens so lange existieren, wie es noch benötigt wird. Das ist der Fall, wenn noch auf das Objekt zugegriffen wird. Da einzelne Objekte einen beachtlichen Speicherbedarf haben können, sollten nicht mehr benötigte Objekte wieder gelöscht werden. Je nach Programmiersprache ist man entweder als Entwickler selbst für das Löschen nicht mehr benötigter Objekte zuständig, oder das System erledigt dies automatisch. Ersteres ist aufwendig und fehleranfällig. Die zweite Variante ist komfortabel und befreit den Entwickler vor lästiger Routinearbeit. Das System der automatischen Speicherfreigabe für nicht mehr benötigte Objekte wird als *Garbage-Collection* bezeichnet. Wir haben es in Abschnitt 3.4.2, »Systematik von Datenobjekten und Datentypen«, nochmals aufgegriffen.

Mit der Kenntnis von Attributen, Methoden und Ereignissen haben Sie die Grundzüge der objektorientierten Programmentwicklung kennengelernt. Weitere wichtige essenzielle Konzepte sehen wir uns jetzt genauer an.

2.3.2 Objektorientierte Konzepte im Detail

Die Grundpfeiler der objektorientierten Programmierung sind sinnbildhaft in Abbildung 2.5 zu sehen. Lassen Sie uns gemeinsam diese Eckpfeiler besprechen, um zu erkennen, welche Grundgedanken die Erfinder der objektorientierten Denkweise damit verfolgt haben. Bitte beachten Sie: Nicht jedes Prinzip wird in gleicher Art und Weise von jeder Programmiersprache umgesetzt. Hier gibt es Abweichungen und unterschiedliche Herangehensweisen.

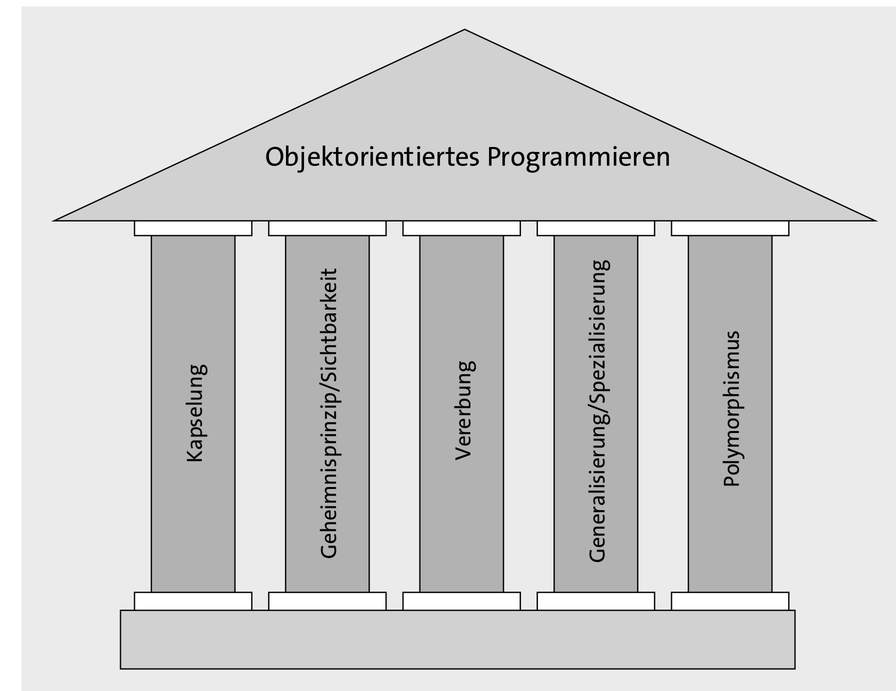

Abbildung 2.5 Grundpfeiler der objektorientierten Programmierung

Kapselung

Das Prinzip der *Kapselung* besagt, dass Attribute und Methoden in einer Einheit zusammengefasst sind. Diese Einheit ist die Klasse bzw. als konkrete Ausprägung ein Objekt. Dabei ist es nicht erlaubt, dass andere Objekte von außen auf die Attribute des betreffenden Objekts zugreifen. Dies geschieht stets über die zugehörigen Methoden (Abbildung 2.6).

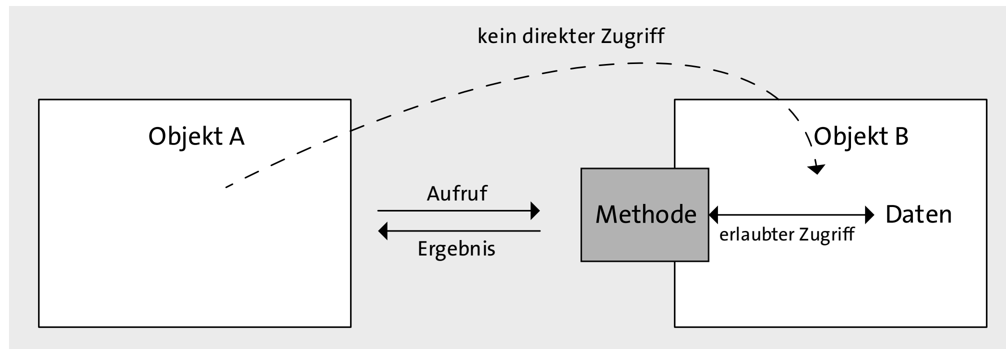

Abbildung 2.6 Das Prinzip der Kapselung verhindert den direkten Zugriff auf die Daten eines Objekts.

Wie bereits erläutert, wird der Datenzugriff bei einigen Programmiersprachen auch über spezielle Felder geregelt, die jedoch wiederum intern auf `get`- und `set`-Methoden zurückgreifen. Dieser streng geregelte Schutz bezüglich des Zugriffs auf die Daten verhindert, dass die Objekte in unberechtigter Weise von außen modifiziert werden.

Geheimnisprinzip/Sichtbarkeit

Das klingt spannend, als hätte es etwas mit dem amerikanischen Secret Service zu tun. Naja, vielleicht nicht ganz so brisant, aber ähnlich :-). Das *Geheimnisprinzip* besagt, dass ein Objekt nach außen nur die Informationen und Methoden bereitstellt, die zur Anwendung notwendig sind. Ein Beispiel: Alle Objekte der Klasse `Auto` verfügen über die Methode `FahrtenbuchAnalysieren`. Für einen Benutzer ist es nicht von Interesse, wie dies innerhalb der Klasse umgesetzt wird. Es interessiert nur, dass der Aufruf der Methode zum gewünschten Ergebnis führt. Die Funktionsweise der Klasse wirkt nach außen wie eine *Black Box* (Abbildung 2.7)

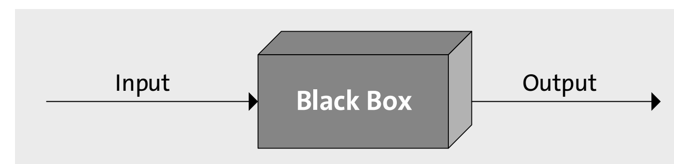

Abbildung 2.7 Das Geheimnisprinzip der objektorientierten Programmierung

Das Prinzip der Black Box hat hier mehrere Bedeutungen. Möchte man die Funktionalität einer Klasse bzw. seiner Objekte nutzen, muss man in keiner Weise dabei wissen, wie die Verarbeitung intern funktioniert. Dieses Prinzip ist essenziell für die Wiederverwendung von Programmcode. Wir werden darauf später noch zurückkommen. Jetzt nur so viel: Als Entwickler ist es üblich, für viele Standardaufgaben die Funktionalität einer allgemein zu verwendenden Klasse zu nutzen, beispielsweise einen speziellen Such-Algorithmus, der als Methode einer Klasse implementiert sein kann. Diese Algorithmen können sehr kompliziert sein, wie wir im kommenden Kapitel sehen werden. Die wirkliche Funktionsweise müssen wir jedoch nicht kennen. Es kommt lediglich darauf an, welche Inputdaten wir an den Algorithmus übergeben und welches Ergebnis wir zurückbekommen. Auch wenn zu einem späteren Zeitpunkt die Klasse aktualisiert wird, weil beispielsweise der verwendete Such-Algorithmus gegen ein leistungsfähigeres Exemplar ausgetauscht wird, ist das Geheimnisprinzip wichtig. Die Klasse kann problemlos in der ursprünglichen Form weiterverwendet werden. Dabei muss man sicherstellen, dass die Schnittstellen gegenüber der vorherigen Version unverändert bleiben. Die technische Umsetzung innerhalb der Klasse kann dabei angepasst werden.

Klassen sowie deren Attribute und Methoden können unterschiedliche Sichtbarkeiten aufweisen. Was heißt das? Die *Sichtbarkeit* bestimmt, welche Elemente eines Objekts für andere Objekte sichtbar sind. Die beiden Extreme lauten

- `private`: Das Element ist nur innerhalb der Klasse sichtbar.
- `public`: Das Element ist für alle anderen Objekte sichtbar.

Beispielsweise werden Attribute, die nur für eine Berechnung innerhalb einer Methode benutzt werden, als `private` definiert. Soll auf die Daten eines Objekts auch von außen zugegriffen werden, so ist das Element als `public` zu kennzeichnen. Für die Klasse `PKW` könnte es beispielsweise sinnvoll sein, die *Property* `Leistung` als `public` zu definieren, denn so können andere Objekte diesen Wert erfragen. Je nach Programmiersprache kann man zwischen den beiden Extremen `public` und `private` weitere Abstufungen in den Sichtbarkeiten definieren. Beispielsweise könnte das Schlüsselwort `protected` darauf hinweisen, dass ein Zugriff von außen auf das Klassenelement möglich, aber nur innerhalb des Moduls erlaubt ist.

Bitte beachten Sie: Als Attribute bezeichnen wir Variablen, die grundsätzlich nur innerhalb der Klasse verwendet werden können. Sie sind daher als `private` deklariert. Will man auf die Werte dieser Attribute von außerhalb des Objekts zugreifen, muss man das über Methoden erledigen. Moderne Programmiersprachen, wie C#, kennen auch das Konzept der Properties. Über eine Property kann von außen auf das zugeordnete Attribut zugegriffen werden. Die Property ist damit als `public` deklariert. Wir werden später in Abschnitt 2.5.4, »Objektorientierung«, nochmals darauf zurückkommen.

Vererbung

Und nun noch ein bisschen Biologie. Mithilfe der *Vererbung* können Attribute und Methoden einer übergeordneten Klasse auf nachgelagerte Klassen vererbt werden. Das bedeutet, wir können Hierarchien von Klassen erstellen, die untereinander in Beziehung stehen, genauer: die Vater-Kind- bzw. Mutter-Kind-Beziehungen einnehmen. Ein Beispiel zeigt Abbildung 2.8.

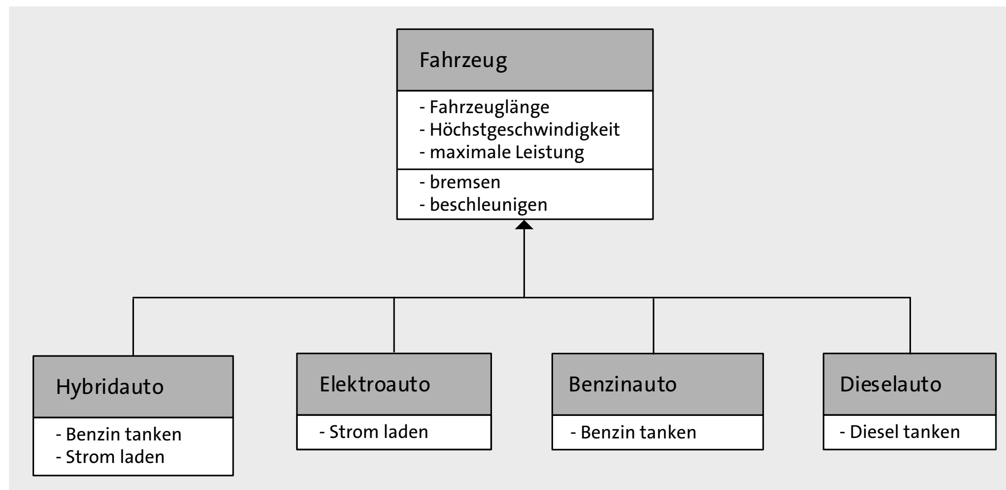

Abbildung 2.8 Das Prinzip der Vererbung als wichtiges Merkmal der objektorientierten Programmierung

Hierbei handelt es sich übrigens um ein sogenanntes *Klassendiagramm*. Zu dieser Form der grafischen Darstellung der Beziehungen von Klassen kommen wir später nochmals zurück. Jetzt studieren Sie bitte zunächst die Darstellung sorgfältig! Die Klasse `Fahrzeug` enthält die Attribute `Fahrzeuglänge`, `Höchstgeschwindigkeit` und `maximaleLeistung` und ebenso die Methoden `bremsen` und `beschleunigen`. Die Klasse `Fahrzeug` ist eine sogenannte *Oberklasse*. Von dieser Klasse werden andere Klassen abgeleitet. In unserem Fall sind dies die Klassen `Hybridauto`, `Elektroauto`, `Benzinauto` und `Dieselauto`. Alle abgeleiteten Klassen verfügen automatisch über dieselben Attribute und Methoden wie die Oberklasse. Dieses Prinzip nennt man *Vererbung*. In den abgeleiteten Klassen muss man dann nur die zusätzlichen Attribute und Methoden hinzufügen. Beispielsweise werden der Klasse `Hybridauto` die Methoden `BenzinTanken` und `StromLaden` hinzugefügt. Wie gesagt, auch diese Klasse kennt die Methoden `bremsen` und `beschleunigen`.

Generalisierung/Spezialisierung

Die Zuordnung von Attributen und Methoden zu Ober- und Unterklassen folgt dabei den Prinzipien von *Generalisierung* und *Spezialisierung* (Abbildung 2.9).

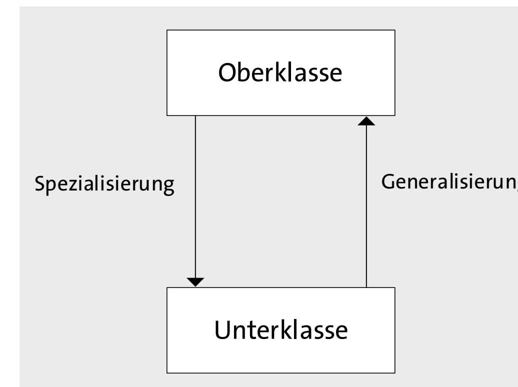

Abbildung 2.9 Das Prinzip von Spezialisierung und Generalisierung ist das Wesensmerkmal der Vererbung.

Innerhalb einer Klassenhierarchie sollten Attribute und Methoden so weit wie möglich oben angeordnet werden: Wenn alle Unterklassen über ein bestimmtes Merkmal verfügen, dann sollte dieses in die Oberklasse angeordnet werden, und die Unterklassen übernehmen dieses Merkmal automatisch per Vererbung. Auf diese Weise braucht man bestimmte Sachverhalte nur einmal codieren, d. h., man vermeidet Wiederholungen und Fehleranfälligkeiten. Bitte beachten Sie ebenfalls, dass eine Klassenhierarchie über viele Ebenen gebildet werden kann, d. h., Attribute und Methoden können sich in diesem Fall über mehrere Ebenen vererben.

An dieser Stelle möchten wir noch einen anderen Begriff bei der Bildung einer Klassenstruktur definieren. Es geht um sogenannte *abstrakte Klassen*. Was ist das? Das sind Klassen, die als Oberklassen fungieren, jedoch können von solchen Klassen selbst keine Objekte erstellt werden. In unserem Beispiel könnte die Klasse `Fahrzeug` als abstrakte Klasse ausgeführt werden. Ein Objekt der allgemeinen Klasse `Fahrzeug` ergibt in der Regel keinen Sinn. Ein Fahrzeug ist stets ein konkretes Fahrzeug des Typs Benzinauto, Dieselauto, Hybridauto oder Elektroauto. Die Klasse `Fahrzeug` sammelt daher alle Attribute und Methoden, die für alle Fahrzeugtypen gelten, selbst kann sie aber nicht als Bauplan für ein konkretes Objekt stehen.

Ebenso muss man bei der Erstellung einer eigenen Klassenbibliothek beachten, dass zwischen Ober- und Unterklassen auch wirklich ein inhaltlicher Zusammenhang besteht. Man sollte also nur da Objekte in eine Hierarchie bringen, wo es auch Sinn macht. Obwohl die Objekte `Hund` und `Tisch` beide über Beine verfügen, sollte man sie nicht als Unterklassen einer gemeinsamen Oberklasse mit dem Attribut `AnzahlBeine` bilden. Das ergibt inhaltlich wenig Sinn und verwirrt bei der Programmentwicklung. Denken Sie daran, dass die Klassenhierarchie ein modellhaftes Abbild eines Realitätsausschnitts sein soll. Hier kommt ja auch niemand auf die Idee, einen Hund und einen Tisch einer gemeinsamen Kategorie zuzuordnen.

Das Prinzip der Vererbung und der Bildung von Klassenhierarchien halten wir für so essenziell für das Verständnis der objektorientierten Programmierung, dass wir uns hier ein zweites Beispiel ansehen (Abbildung 2.10).

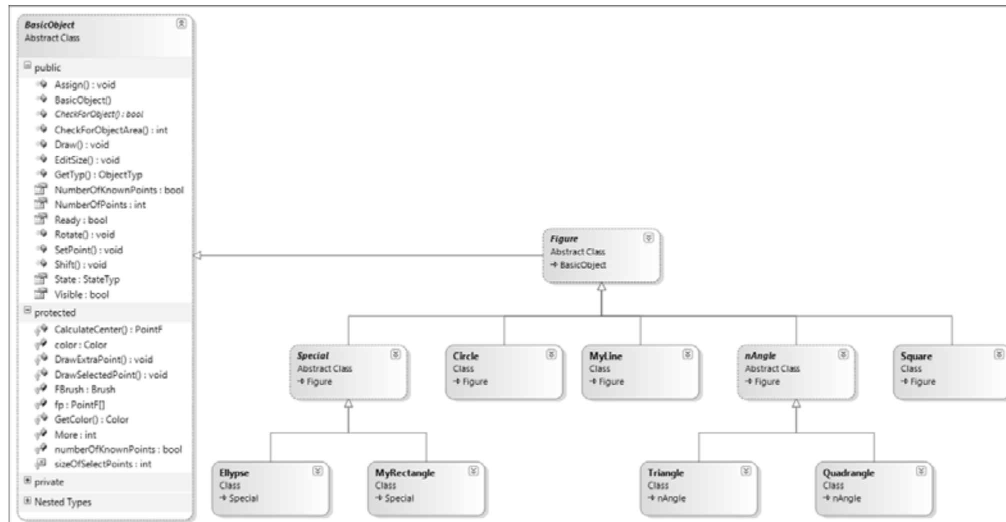

Abbildung 2.10 Klassenhierarchie für ein vektorbasiertes Zeichenprogramm

Das Beispiel stammt aus einem vektorbasierten Grafikprogramm. Hier können wir einige interessante Beobachtungen machen:

- *Zweck*: Die Klassenhierarchie ordnet typische Zeichenobjekte der Ebene in eine mehrstufige Vererbungsbeziehung zueinander. Ein Beispiel: Die Klassen `Triangle` (Dreieck) und `Quadrangle` (Viereck) sind spezielle Klassen der abstrakten Klasse `nAngle` (Vieleck). Man kann also kein Objekt der Klasse `nAngle` erzeugen, aber durchaus ein Objekt der Klasse `Triangle`. Für die anderen Klassen gelten ähnliche Zusammenhänge.
- *Ober-/Unterklassen*: die oberste Basisklasse für die Zeichenobjekte ist die Klasse `Figure`. Sie ist ebenfalls abstrakt und leitet von einer weiteren Klasse, `BasicObject`, ab. Die Klasse `BasicObject` enthält alle essenziellen Methoden zum Zeichnen. Sie könnte weitere Unterklassen außer `Figure` haben, die hier nicht dargestellt werden.

Bitte beachten Sie: Es gibt nicht die eine richtige Hierarchie zur Bildung von Ober- und Unterklassen. Wichtig ist es, stets den sachlichen Bezug zu wahren und zu überlegen, welche Methoden und Attribute auf welcher Ebene angeordnet werden sollen.

Polymorphie

Kommen wir zur letzten tragenden Säule der objektorientierten Entwicklung. Gemeint ist die sogenannte *Polymorphie*. Das Prinzip ist einfach und genial zugleich. Objekte verschiedener Klassen müssen gelegentlich gleiche Funktionen ausführen. Auch dazu haben wir ein kleines Beispiel für Sie vorbereitet (Abbildung 2.11).

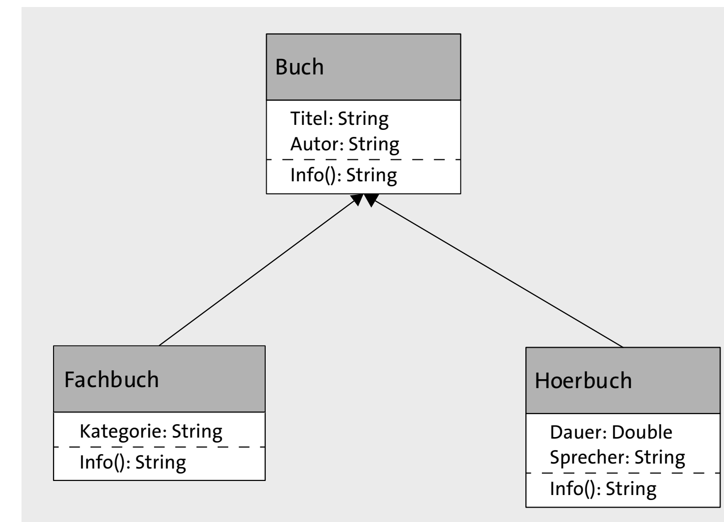

Abbildung 2.11 Polymorphie: In Unterklassen wird die Funktionalität einer Oberklasse spezifiziert.

Sehen wir uns das etwas genauer an. Wir haben eine Oberklasse `Buch` und zwei abgeleitete Klassen `Fachbuch` und `Hoerbuch`. Und was fällt Ihnen auf? Die Klasse `Roman` fehlt. Das war ein Spaß! Uns interessiert im Moment lediglich die Methode `info()`. Sie liefert eine Zeichenkette vom Datentyp `String`, die eine Information zum Buch ausgibt. In der Klasse `Buch` wird diese Methode nur als Platzhalter als sogenannte *abstrakte Methode* definiert, sie enthält keinen Code. Nur Objekte der abgeleiteten Klassen `Fachbuch` und `Hoerbuch` verfügen über die entsprechenden Daten, um Informationen auszugeben. Dazu wird jeweils eine Methode `info()` in den Klassen `Fachbuch` und `Hoerbuch` definiert. Diese Methoden enthalten dann auch den jeweils spezifischen Quellcode. Wie wirkt sich diese Art der Konstruktion in der Programmierpraxis aus? Nehmen wir an, wir definieren ein Objekt in der folgenden Form:

```
Buch einBuch = new Fachbuch();
```

Der Variablen `einBuch` wird ein Objekt von der Klasse `Fachbuch` zugeordnet. Später ist es durchaus erlaubt, das Objekt neu zuzuweisen, vielleicht dieses Mal als `Hoerbuch`. Wir würden dann schreiben:

```
buch = new Hoerbuch;
```

Egal zu welchem Zeitpunkt, wir können jederzeit die Methode `info()` aufrufen. Je nachdem, ob es sich um ein Objekt der Klasse `Fachbuch` oder der Klasse `Hoerbuch` handelt, wird die korrekte Methode aufgerufen.

Zusammengefasst: Das Prinzip der Polymorphie sorgt also dafür, dass der Compiler stets die richtigen Attribute und Methoden auswählt. Würden wir unsere kleine Klassenbibliothek tatsächlich später um eine Klasse `Roman` ergänzen, dann müsste auch diese eine Methode `info()` haben. Es würden sich nicht die Konventionen ändern, man braucht auch wieder nur die Methode `info()` aufrufen, und man bekommt dann die Informationen zum Roman angezeigt.

Mithilfe der Polymorphie ist es auch möglich, in einer Oberklasse eine allgemein verwendbare Methode zu definieren und auch diese mit entsprechenden Quellcode zu versehen. Abgeleitete Unterklassen haben dann grundsätzlich zwei Optionen:

1. Sie verwenden die Methode der Oberklasse.
2. Sie können eine eigene Methode mit gleichem Namen definieren und gewissermaßen die Methode der Oberklasse überschreiben.

Damit gilt folgendes Prinzip: Beim Aufruf einer Methode wird immer geprüft, ob es eine speziellere Methode der zugehörigen Klasse gibt. Ist das nicht der Fall, dann wird die Methode der Oberklasse verwendet. Es gilt also: Eine spezielle Implementierung (Attribut oder Methode) hat stets Vorrang vor einer allgemeineren (generischen) Implementierung.

Wenn Sie mit uns gemeinsam bis zu dieser Stelle der objektorientierten Programmierung vorgedrungen sind, dann kennen Sie wirklich schon die wichtigsten Konzepte und Möglichkeiten. Sie werden Ihnen in Fleisch und Blut übergehen, wenn Sie sie aktiv bei einem Projekt anwenden. Bitte beachten Sie: Die Konzepte der objektorientierten Programmierung sind nicht starr, sie unterliegen einer fortlaufenden Entwicklung. Auch unterstützen nicht alle Programmiersprachen sämtliche Möglichkeiten vollständig.

2.3.3 Objektorientierung und Wiederverwendung

Die objektorientierte Programmierung fördert insbesondere den Ansatz der Wiederverwendung. Quellcode zur Lösung allgemeiner Probleme kann und sollte für ähnliche Fragestellungen zur Verfügung gestellt werden. Dies geschieht in Form von Klassenbibliotheken, die als Referenzen in Projekte eingebunden werden. Bei der objektorientierten Programmierung führt es dann zu dem in Abbildung 2.12 beschriebenen grundsätzlichen Systemaufbau.

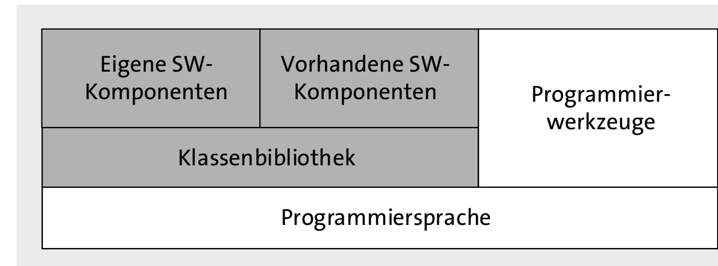

Abbildung 2.12 Komponenten der objektorientierten Programmierung

Ausgangsbasis ist die jeweilige Programmiersprache, die die Prinzipien der objektorientierten Programmierung möglichst umfassend unterstützt. Die Software als Modell wird dann in Form einer Klassenbibliothek umgesetzt. Dabei wäre es natürlich möglich, sämtliche Funktionalität vollständig neu zu implementieren. Das wäre jedoch nicht sinnvoll und nicht effektiv. Viele Aufgaben der Programmentwicklung sind wiederkehrend. Typische Beispiele sind:

- Benutzeroberflächen bestehen in fast allen Programmen aus ähnlichen Komponenten wie Buttons, Labels oder Textboxen.
- Funktionen wie Dateioperationen oder Suchfunktionen werden immer wieder benötigt.

Es ergibt also Sinn, den Quellcode in einer wiederverwendbaren Form zu schreiben und in allgemein zugänglichen Klassenbibliotheken abzulegen. Ein Programm nutzt dabei also eigene und vorhandene Softwarekomponenten.

Alle eigenen und fremden Softwarekomponenten werden dabei üblicherweise in einem Projekt zusammengefasst. Die Verwaltung der Bibliotheken und deren Abhängigkeiten untereinander muss man dabei üblicherweise nicht per Hand vornehmen. Dabei helfen komfortable Programmierwerkzeuge in Form von *integrierten Entwicklungsumgebungen* (*IDE*). Damit man die Bibliotheken nicht mühsam an unendlich vielen Orten zusammensuchen muss, werden diese in zentralen *Repositorien* bereitgestellt. Als Entwickler können Sie die gewünschten Bibliotheken suchen, in Ihr Projekt einbinden sowie gegenseitige Abhängigkeiten verwalten und auflösen. Bei Aktualisierungen werden Sie benachrichtigt. Lassen Sie uns das auch an einem Beispiel verdeutlichen. Sehen Sie sich dazu Abbildung 2.13 an.

Wir sehen die Entwicklungsumgebung Visual Studio. Am rechten Rand ist der Projektmappen-Explorer eingeblendet, der alle Bestandteile des aktuellen Projekts zeigt. Unterhalb von VERWEISE sehen Sie, welche externen Bibliotheken im aktuellen Projekt eingebunden sind. In diesem Fall sind es die beiden Bibliotheken `Microsoft.NetCore.UniversalWindowsPlatform` und `Microsoft.Xaml.Behavios.Uwp.Managed`. Deren Funktionen spielen jetzt nur eine untergeordnete Rolle. Es sind zwei grundsätzliche

Bibliotheken zum Erstellen von Apps für die Universal Windows Platform. Im linken Teilbereich des Fensters sehen Sie den NuGet-Paketmanager. Mit dessen Hilfe werden die Bibliotheken zentral verwaltet. Die Entwickler der Bibliotheken können ihn über einen zentralen Server anderen Entwicklern zur Verfügung stellen. Mittels des NuGet-Managers kann man auf öffentlich zugängliche Bibliotheken über das Internet zugreifen. Alternativ können Softwareentwicklungsunternehmen ihre eigenen Softwarebibliotheken aufbauen und auf einem eigenen Unternehmensserver verwalten.

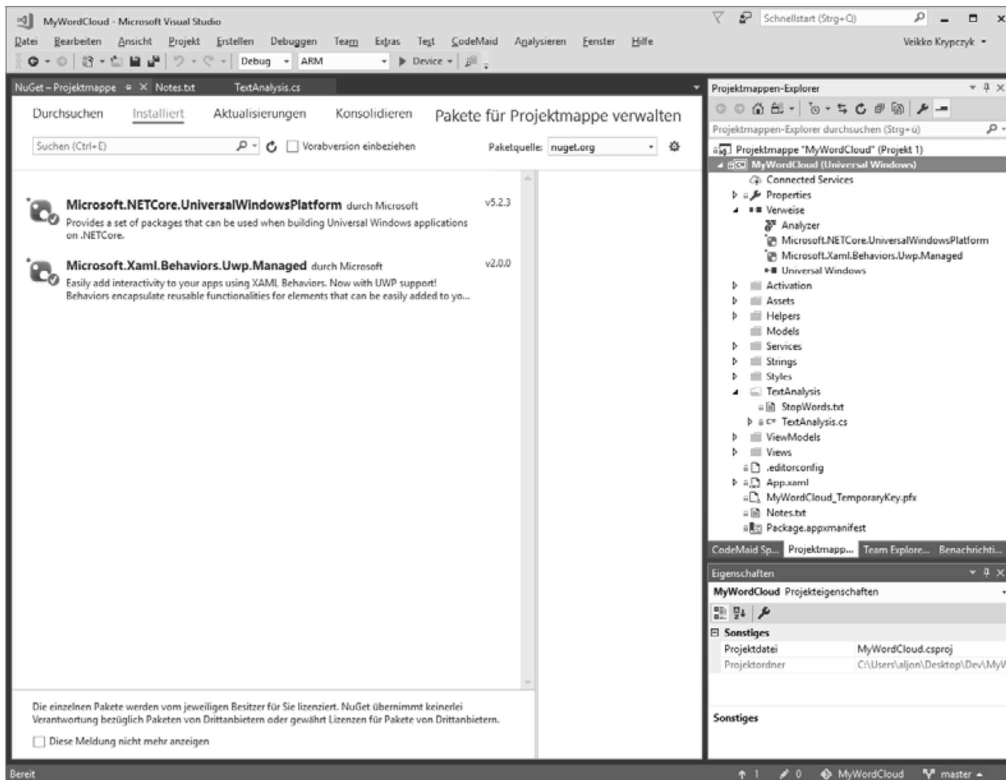

Abbildung 2.13 Die Verwaltung der Softwarekomponenten erfolgt komfortabel in der Entwicklungsumgebung, hier durch den Paketmanager NuGet.

Je nach Programmtyp und Vorgehensweise wird also bei der Erstellung eines neuen Projekts ein bestimmtes Set an vorhandene Softwarekomponenten zur Nutzung direkt angebunden. Weitere Bibliotheken werden dann während der Programmentwicklung durch den Softwareentwickler verwendet. Woher kommen diese Softwarekomponenten? Sie können vom Hersteller der Programmiersprache, von sogenannten Drittanbietern und aus eigenen früheren Projekten stammen. Üblicherweise ist es so, dass ein Projekt letztendlich auf einen Großteil von bereits vorhandenen Softwarekomponenten zugreifen kann. Nur die spezifische Funktionalität

eines Programms wird dann neu entwickelt. Tatsächlich ist es erst dadurch möglich, die immer weiter steigenden Anforderungen an neue Software in Sachen Umfang und Komplexität zu bewältigen.

Ein sehr gutes Beispiel ist das .Net-Framework, das eine sehr umfassende Klassenbibliothek mit mehreren Tausend Klassen für ein sehr breites Anwendungsspektrum bereitstellt. Für nahezu alle möglichen Anwendungsszenarien finden sich bereits vorgefertigte Implementierungen, auf die bei der Programmierung der eigenen Anwendung dann zurückgegriffen werden kann.

Ein weiteres Beispiel ist die Umsetzung von grafischen Benutzeroberflächen für Standardapplikationen. Diese bestehen aus typischen Elementen wie speziellen Buttons, Menü-Elementen, Textfeldern usw. Programmiert man in Java, so steht dem Entwickler auch hier eine umfassende Klassenbibliothek zur Verfügung. Selbstverständlich finden Sie online bei den Herstellern der Bibliotheken die entsprechenden Dokumentationen. Für einen ersten Überblick bzw. bei der späteren Verwendung ist es jedoch äußert hilfreich, sich die relevanten Zusammenhänge in einem vereinfachten Klassendiagramm anzusehen.

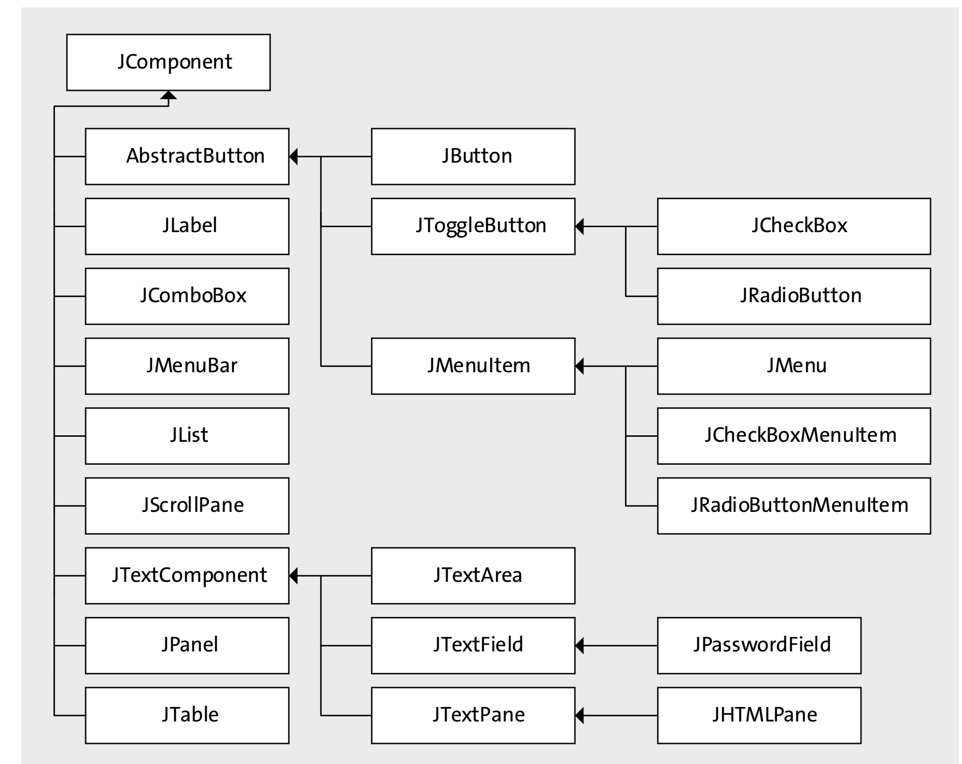

Abbildung 2.14 Auszug aus dem Klassendiagramm für die UI-Erstellung in Java (Quelle: Habelitz, 2016)

Abbildung 2.14 zeigt einen solchen Ausschnitt für die Java-Klassenbibliothek zum Erstellen von grafischen Benutzeroberflächen. Anhand eines solchen Klassendiagramms können Sie erkennen, wie die konkreten Klassen als Ober- und Unterklassen zusammenhängen.

2.3.4 Visualisierung: Objektorientierung und UML

Bei der Darstellung der Grundlagen zur objektorientierten Programmierung haben wir schon an der einen oder anderen Stelle auf grafische Visualisierungen zurückgegriffen. Wir haben die Zusammenhänge zwischen den Klassen in Form von Klassendiagrammen dargestellt. Die grafische Visualisierung hilft beim Entwurf von komplexen Softwaresystemen. Ebenso gelingt die Analyse bestehender Systeme mithilfe von Diagrammen deutlich einfacher. Mithilfe von Tools gelingt der Transfer zwischen Quellcode und grafischem Modell dabei weitgehend vollautomatisch. Während der Konzeption kann man zeichnerisch ein grafisches Modell der Klassenbibliothek erstellen und dieses danach automatisch in die jeweilige Programmiersprache transferieren lassen. Üblicherweise modelliert man auf grafischer Ebene nur den konzeptionellen Rahmen und vervollständigt dann die Klassenstruktur im Quellcode. Ähnlich hilfreich sind grafische Modelle, um sich einen Überblick über die Struktur eines Anwendungssystems zu informieren.

Dabei hat sich die Modellierungssprache *UML* etabliert. Die Abkürzung UML steht für *Unified Modeling Language*. Die UML bietet ein ganzes Spektrum an Diagrammtypen für die Visualisierung der unterschiedlichsten Sachverhalte und ist auf den objektorientierten Ansatz ausgerichtet; unter anderem gehört das Klassendiagramm dazu. Begibt man sich auf die Ebene der einzelnen Objekte, so ist das *Objektdiagramm* zur grafischen Visualisierung geeignet. Beide Diagrammtypen liefern gewissermaßen eine statische Sichtweise auf das Softwaresystem. Eine dynamische Sichtweise bekommt man über das *Sequenzdiagramm*. Nachfolgend stellen wir alle drei Diagrammtypen etwas näher vor. Sie werden sie mit Sicherheit gelegentlich bei der Programmierung selbst erstellen oder bei der Einarbeitung in fremden Quellcode inspizieren.

Klassendiagramm

Klassendiagramme stellen das zentrale Konzept der UML dar. Eingesetzt werden sie in allen Phasen des Softwareentwicklungsprozesses. Sie können unterschiedliche Detaillierungsgrade aufweisen und beschreiben grafisch die Beziehungen zwischen den Klassen einer Anwendung. Neben dem Namen der Klasse werden die Attribute und Methoden dargestellt. Ein Beispiel für ein Klassendiagramm über zwei Ebenen zeigt Abbildung 2.15.

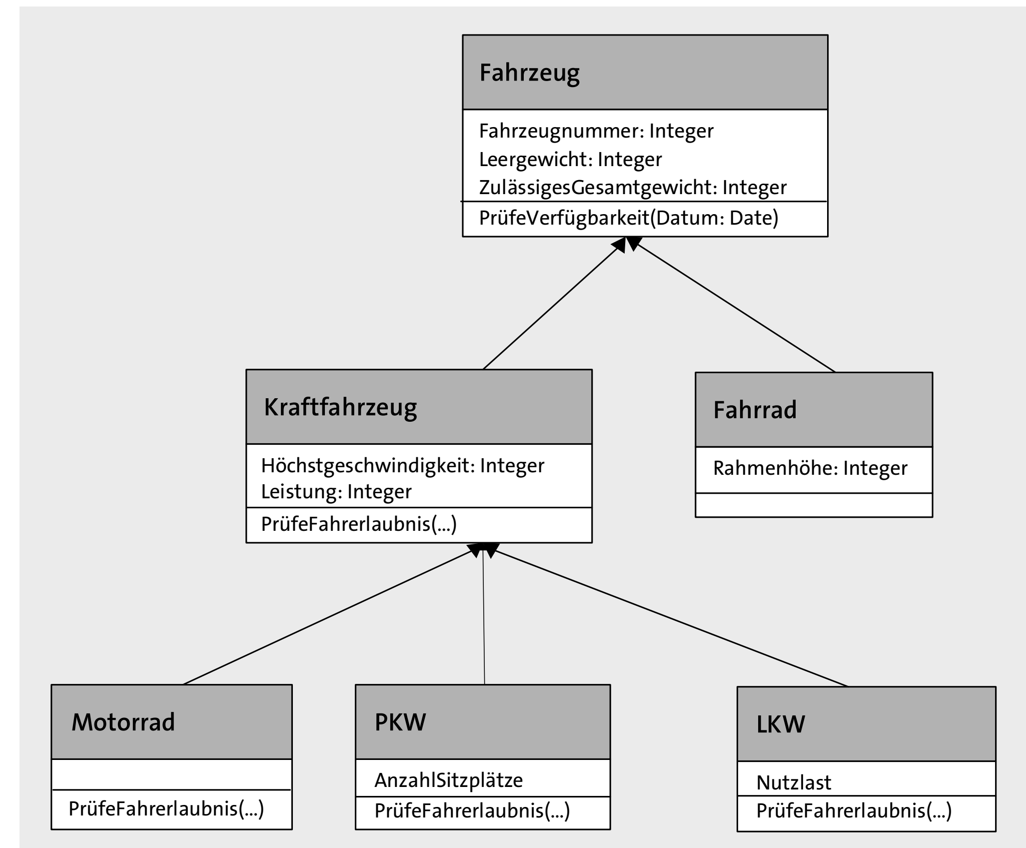

Abbildung 2.15 Beispiel für ein Klassendiagramm

Angegeben sind jeweils der Name der Klasse und darunter die Attribute. Es folgen die Methoden. Die Klammern bei den Methoden, zum Beispiel bei `PruefeFahrerlaubnis(...)`, deuten an, dass an die Methode Parameter zur Bearbeitung übergeben werden. Über Notationselemente sind Erweiterungen denkbar, zum Beispiel können bei den Attributen zusätzlich die Sichtbarkeit (`public`, `private`) und/oder der Datentyp angeben werden.

Objektdiagramm

Das Objektdiagramm ist eine Spezifizierung des Klassendiagramms. Es stellt die Beziehungen der tatsächlich erzeugten Objekte zu einem bestimmten Zeitpunkt zur Laufzeit dar. Objektdiagramme sind als Ergänzung zu Klassendiagrammen aufzufassen. Im Objektdiagramm werden der Name des Objekts und durch Doppelpunkt die zugehörige Klasse angegeben. Da die Abbildung zu einem bestimmten Zeitpunkt erfolgt, weisen die Attribute bestimmte Werte auf, die angegeben werden können. Brauchen Sie ein Beispiel? Hier kommt es (Abbildung 2.16).

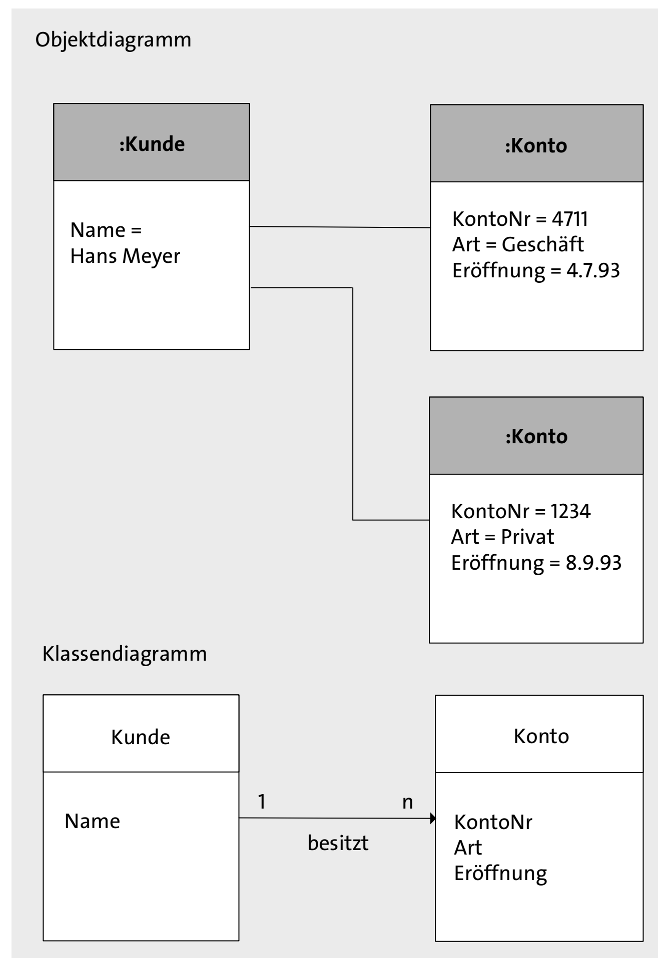

Abbildung 2.16 Objektdiagramm mit zugehörigem Klassendiagramm

Dieses Mal beginnen wir von unten, die Abbildung zu interpretieren. Wir haben ein Klassendiagramm, das aus den Klassen `Kunde` und `Konto` besteht. Zu beiden Klassen sind wichtige Attribute genannt. Methoden sind aus Gründen der Vereinfachung nicht aufgeführt. Bitte beachten Sie, dass das die übliche Vorgehensweise bei der Erstellung von Diagrammen ist. Man stellt nur diejenigen Sachverhalte grafisch dar, die im gerade betrachteten Kontext wirklich interessieren.

Zwischen beiden Klassen besteht eine 1:n-Beziehung, ein Kunde kann kein, ein oder beliebig viele Konten haben. Im oberen Teil ist ein Objektdiagramm abgebildet. Es bezieht sich genau auf dieses Klassendiagramm und stellt eine Momentaufnahme zu einem bestimmten Zeitpunkt dar. Wir erkennen: Der Kunde Meyer verfügt über zwei Konten. Objektdiagramme können also genutzt werden, um einen abstrakten Sachverhalt eines Klassendiagramms zu konkretisieren.

Sequenzdiagramm

Sequenzdiagramme dienen dazu, den Nachrichtenfluss zwischen den Objekten darzustellen. Insbesondere wird der zeitliche Ablauf der Nachrichten verdeutlicht. Die Notationselemente sind die Lebenslinie (gestrichelte Linie), die Nachricht und der Interaktionsrahmen. Eine Lebenslinie ist genau einem Objekt einer Klasse, dargestellt als Rechteck, zugeordnet. Die Lebenslinie symbolisiert die passive Lebenszeit des Objekts. Wird das Objekt verwendet, kommt es zur Aktivierung. Mittels eines Balkens wird dieser Zustand verdeutlicht. Objekte von Klassen werden mithilfe eines *Konstruktors* erzeugt. Die Kommunikation zwischen den Objekten wird mithilfe von Nachrichten dargestellt, das entspricht dem Aufruf einer Operation. Es ist es auch möglich, dass Objekte Nachrichten an sich selbst senden. Oft wird davon abgesehen, das Zerstören bzw. Löschen eines Objekts explizit abzubilden. Der Grund: Moderne Laufzeitumgebungen sorgen meist eigenständig dafür, dass nicht mehr benötigte Objekte aus dem Speicher gelöscht werden. Abbildung 2.17 zeigt ein Beispiel für ein Sequenzdiagramm.

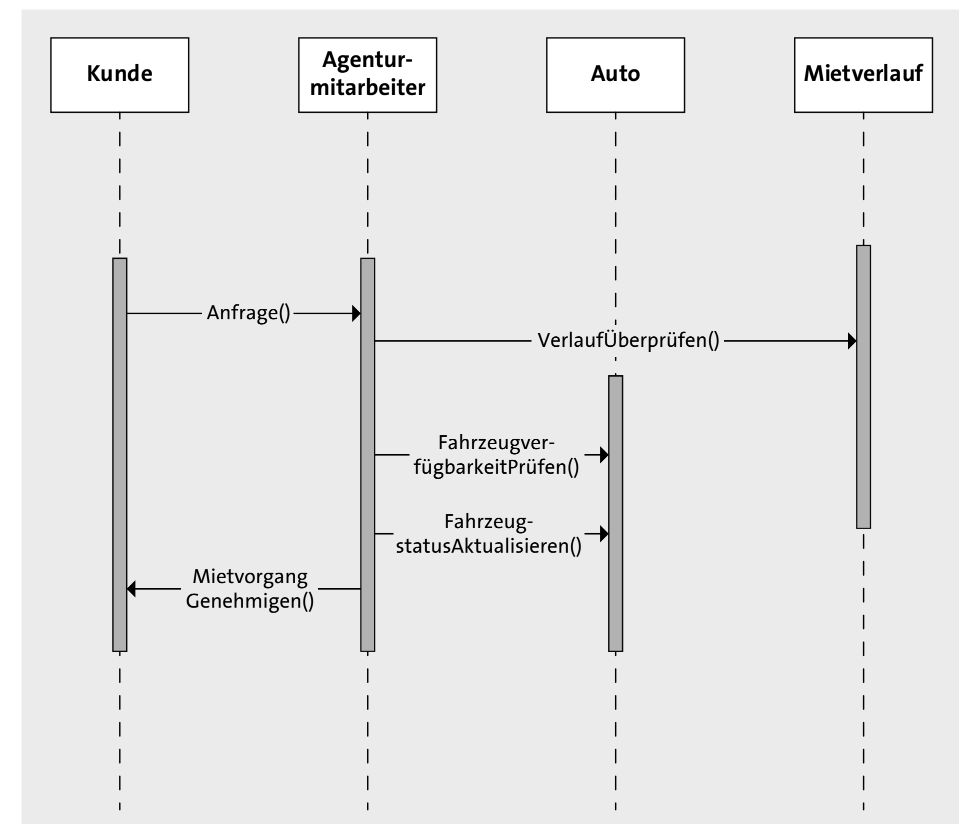

Abbildung 2.17 Beispiel für ein Sequenzdiagramm (Quelle: https://msdn.microsoft.com/de-de/library/bb979230.aspx)

Wie ist es zu lesen? Eigentlich ganz einfach: Wir haben vier unterschiedliche Klassen, nämlich `Kunde`, `Agenturmitarbeiter`, `Auto` und `Mietverlauf`. Attribute und Methoden der Klassen spielen in diesem Diagrammtyp keine Rolle. Ein Objekt vom Typ `Kunde` kann eine Anfrage an einen konkreten Mitarbeiter der Agentur senden (Objekt der Klasse `Agenturmitarbeiter`). Dieser sendet drei Botschaften. An ein Objekt der Klasse `Mietverlauf` wird die Botschaft übermittelt, den bisherigen Mietverlauf zu prüfen. Ebenso werden an ein Objekt der Klasse `Auto` zwei Botschaften gesendet. Es sollen Verfügbarkeit und Status des Fahrzeugs überprüft werden. Je nach Rückmeldungen der Objekte der Klassen `Auto` und `Mietverlauf` kann der Mitarbeiter die Anfrage des Kunden beantworten, d. h. im positiven Fall bestätigen.

Sequenzdiagramme dienen also dazu, den Ablauf zu visualisieren. Anders als klassische Flussdiagramme nehmen sie jedoch stets Bezug auf die Objekte. Man erkennt also die Interaktion der Objekte untereinander.

Auch hier soll ein zweites Beispiel für ein noch besseres Verständnis sorgen. Wir haben uns dazu den Zeichenvorgang für das bereits zuvor genannte vektorbasierte Zeichenprogramm herausgesucht. Das Ziel: Es soll eine Linie auf einer Zeichenfläche erstellt werden. Eine Linie besteht bekanntermaßen aus zwei Punkten, einem Start- und einem Endpunkt. Wie funktioniert ein solcher Zeichenvorgang? Folgende Schritte muss man als Anwender dazu üblicherweise abarbeiten:

1. Der Anwender selektiert das ausgewählte Zeichenwerkzeug aus einer Toolpalette.
2. Mit dem Mauszeiger wird der Startpunkt auf der Zeichenfläche angesteuert, und der Zeichenvorgang wird mit einem Klick auf die linke Maustaste gestartet.
3. Danach bewegt der Anwender den Mauszeiger zum Endpunkt. Es wird interaktiv die Linie neu gezeichnet.
4. Der Endpunkt wird erneut durch einen Mausklick bzw. durch Loslassen der linken Maustaste erfasst. Die Linie wird endgültig gezeichnet.

Diese Abläufe müssen programmiert werden. Essenziell sind dabei die Ergebnisse »Maustaste gedrückt« und »Mausbewegung«. Sehen wir uns den konkreten Ablauf nun mithilfe von Abbildung 2.18 an.

Beteiligt sind Objekte der Klassen `Grafik` und `Zeichenobjekt`. Ein Objekt der Klasse `Grafik` repräsentiert die Zeichenfläche. Ebenso erstellen wir eine `Linie` als ein konkretes Objekt der Klasse `Zeichenobjekt`. Mit dem Start des Zeichenvorgangs wird durch die Klasse `Grafik` in einem ersten Schritt ein Zeichenobjekt erzeugt. Nachdem der Benutzer die linke Maustaste gedrückt hat, wird die Mausposition als Startpunkt an das Objekt übergeben. Die Ermittlung der Mausposition, d. h. der *x*- und *y*-Koordinaten, erfolgt durch die Zeichenfläche, daher ist die Nachricht `ErfasseMausposition` auf sich selbst gerichtet. Dann findet der interaktive Zeichenvorgang, wie er eben beschrieben wurde, statt. Dazu wird immer wieder die aktuelle Mausposition ermittelt, das Objekt an der bisherigen Stelle wird gelöscht, die temporären Koordinaten des Endpunkts werden an das Linienobjekt übermittelt, und die Linie wird neu gezeichnet.

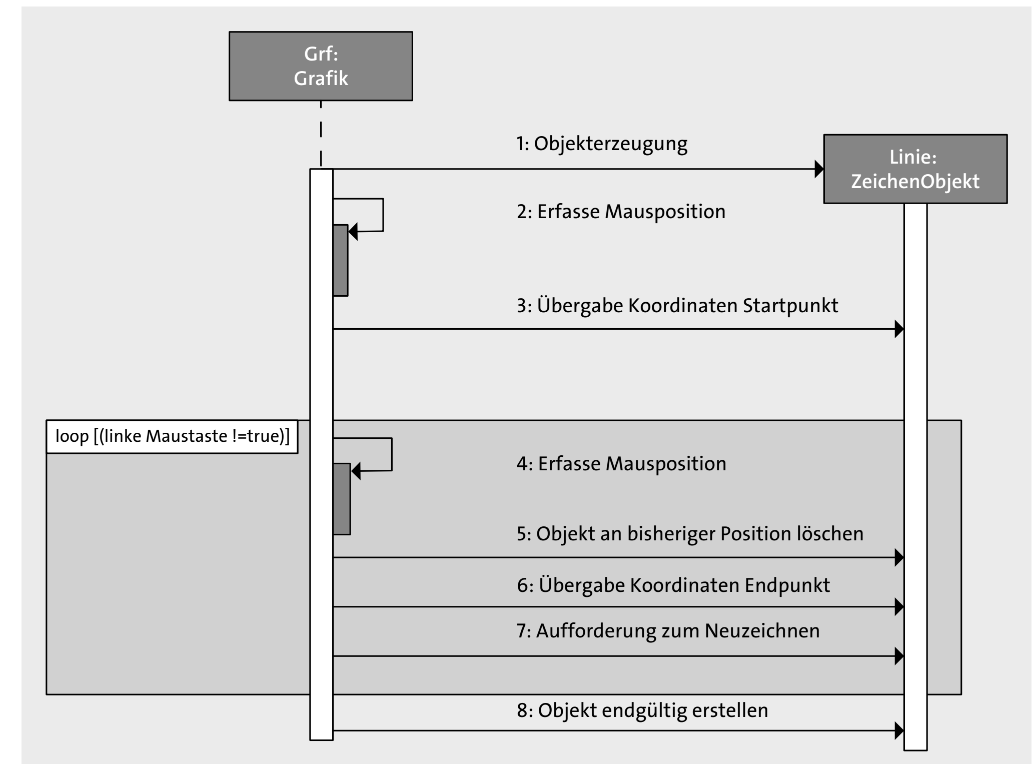

Abbildung 2.18 Sequenzdiagramm für den Zeichenvorgang einer Linie

Diese Schritte werden sehr schnell bei jeder Mausbewegung ausgeführt, bis der Nutzer den Zeichenvorgang durch einen weiteren Mausklick abschließt. Am Ende wird das Objekt abschließend erstellt. Im Sequenzdiagramm ist die Reihenfolge der Nachrichten numerisch angegeben. Diese Angabe ist optional. Der sich wiederholende Zeichenvorgang ist durch den Zusatz *loop* und durch den zusätzlichen Interaktionsrahmen gekennzeichnet.

Sie haben gesehen, wie man mithilfe von Diagrammen die wichtigsten Schritte beim objektorientierten Programmentwurf unterstützen kann. Ebenso werden Sie bemerkt haben, dass sich der Detaillierungsgrad von Diagramm zu Diagramm unterscheiden kann. Je nach Ziel bringt man in der Praxis mehr oder weniger Informationen in einem Diagramm unter. Nichtrelevante Informationen werden zugunsten einer schnellen Modellierung und einer besseren Übersichtlichkeit vernachlässigt. Ebenso werden Form und Layout der Diagramme von den Softwaretools unterschiedlich interpretiert. Die grundsätzliche Arbeitsweise bleibt jedoch identisch.

Nachdem wir Ihnen die wichtigen Paradigmen der Programmentwicklung und die objektorientierte Vorgehensweise vorgestellt haben, beschäftigen wir uns nun mit konkreten Programmiersprachen. Welche Sprachen gibt es? Kann man diese nach bestimmten Kriterien systematisieren, und welche Sprachen sind denn nun heute angesagt?

2.4 Programmiersprachen

In diesem Kapitel beschäftigen wir uns mit der wirklich großen Vielfalt der zur Auswahl stehenden Programmiersprachen. Dazu blicken wir zurück zu den Anfängen der Entwicklung und ordnen die Sprachen nach ihren bestimmenden Merkmalen.

2.4.1 Historische Entwicklung und sprachliche Vielfalt

In diesem Abschnitt geben wir Ihnen einen kompakten Überblick über wichtige Programmiersprachen. Dabei stellt sich gleich die Frage: Welche ist die beste Sprache? Welche soll ich als Erstes oder als Nächstes lernen? Hier müssen wir Sie enttäuschen. Es gibt keine ideale, für alle Zwecke geeignete Programmiersprache. Je nach Problemstellung eignet sich die eine oder andere Sprache besser, sie wird durch das gewählte Betriebssystem umfassender unterstützt oder dafür verfügbare Werkzeuge wie beispielsweise Entwicklungsumgebungen sind besonders leistungsfähig. Ebenso kann man die einzelnen Sprachen danach ordnen, welche grundsätzlichen Konzepte ihnen zugrunde liegen. Handelt es sich um eine prozedurale, um eine objektorientierte oder um eine funktionale Sprache? Moderne Programmiersprachen erlauben es auch, mit unterschiedlichen Ansätzen zu arbeiten. Ein Beispiel: Ursprünglich basierten die Sprachen C++, C# und Java nahezu ausschließlich auf dem objektorientieren Sprachparadigma. Mit der Zeit hat man jedoch festgestellt, dass sich viele Fragestellungen besser mit einem funktionalen Ansatz lösen lassen. Die Erfinder der jeweiligen Sprachen haben darauf reagiert und zunehmend funktionale Sprachfeatures eingebaut. Ein großer Vorteil: Während der Programmierung kann man flexibel entscheiden, in welchem Programmieransatz sich die Lösung am besten umsetzen lässt.

Auch die historische Entwicklung von Programmiersprachen spielt bis heute eine große Rolle. Kaum einer hat die Entwicklung des Computers so geprägt wie Konrad Zuse. Einen kompakten Überblick zur historischen Entwicklung haben wir für Sie im Kasten »Kleine Geschichtsstunde zur Programmierung« aufbereitet. Wir hoffen, dass er für Sie interessant ist, auch wenn Sie im Geschichtsunterricht in der Schule gelegentlich eingeschlafen sind.

Kleine Geschichtsstunde zur Programmierung

Erste Ideen, die mit dem heutigen Ansatz der Programmierung im weitestgehenden Sinne vergleichbar sind, stammen bereits aus dem 17. Jahrhundert. Als erste mechanische Rechenmaschine (1635) wird die sogenannte *Rechenuhr* des deutschen Mathematikers Wilhelm Schickard (1592–1635) bezeichnet. Den Aufzeichnungen zufolge sollte sie alle vier Grundrechenarten mit Zahlen bis zu einer Größe von einer Million beherrschen. Ob diese Maschine tatsächlich gebaut wurde, ist nicht klar. Möglich wäre es jedoch gemäß den Aufzeichnungen gewesen. Den gleichen Ansatz verfolgt die 1643 gebaute *Pascaline* des französischen Mathematikers Blaise Pascal (1623–1662). Aufgrund von Ungenauigkeiten in der Konstruktion produzierte die Maschine jedoch fehlerhafte Ergebnisse.

Zu erwähnen ist in diesem Zusammenhang auch die Einführung des *programmierbaren Webstuhls* von Joseph-Marie Jacquard. Der Webstuhl wurde durch Lochstreifen gesteuert, d. h., das zu webende Muster wurde damit festgelegt. Die ersten Ideen zur Programmierung von Rechnern gehen zurück auf Charles Babbage (1791–1871). Er entwarf zwei Rechenmaschinen, die *Difference Engine* (1823) und die *Analytical Engine* (1834), die jedoch niemals fertiggestellt wurden. Die Operation, die ausgeführt werden sollte, war die Addition von 27-stelligen Dezimalzahlen. Die Logik steckte ohne Programmiersprache in einfachen Maschinenkonstruktionen.

Als Pionier auf dem Gebiet der Programmierung gilt Konrad Zuse (1910–1995). Konrad Zuse wurde am 22. Juni 1910 in Berlin geboren. Er hatte die Vision, stupide Rechenarbeiten auf eine Maschine zu übertragen. Konrad Zuse wollte binär arbeitende Rechner bauen. Diese sollten mit sogenannten *bistabilen Bauelementen* arbeiten. Nach diesem Prinzip, der Aussagenlogik, sollten nicht nur Zahlen verarbeitet werden, sondern die gesamte Maschine sollte darauf beruhen. Die erste Maschine, *Z1*, konstruierte er in den Jahren 1936 bis 1938. Es war die erste programmierbare Rechenmaschine der Welt. Einen Nachbau kann man heute im Museum für Verkehr und Technik in Berlin besichtigen. Weiterentwicklungen waren die *Z2* (Konstruktion 1938 bis 1939) und die *Z3* (Konstruktion 1941). Die Z3 beinhaltete ca. 600 Relais im Rechenwerk und 1.400 Relais im Speicher. Die Z3 gilt heute als der erste funktionsfähige, frei programmierbare, auf dem binären Zahlensystem und der binären Schaltungstechnik basierende Rechner der Welt. Die *Z4*, begonnen im Jahr 1942, wurde 1949 nach dem Krieg restauriert und war ab 1950 fünf Jahre lang an der Eidgenössischen Technischen Hochschule in Zürich im Einsatz.

In den Jahren 1942 bis 1946 arbeitete Konrad Zuse an einer universellen algorithmischen Sprache, *Plankalkül*. Plankalkül enthielt unter anderem Zuweisungszeichen, Datenstrukturen, Datentypen wie Gleit- und Festkommazahlen, Unterprogrammtechnik, verschiedene Schleifentypen und darüber hinaus umfangreiche Abhandlungen zu Schachprogrammen. Es wurde erst 1971 veröffentlicht.

Zur Macht der Computer soll Konrad Zuse gesagt haben »Wenn die Computer zu mächtig werden, dann zieht doch einfach den Stecker aus der Steckdose«.

Zu viele Programmiersprachen gibt es, als dass wir sie hier alle erwähnen könnten. Einige Sprachen haben sich universell etabliert und weisen eine breite Palette an Konzepten auf, andere Sprachen haben auf neueste Entwicklungen reagiert und damit bewusst mit »Altlasten« gebrochen. Eine wie gesagt nicht vollständige Übersicht bekannter und zugleich historischer Programmiersprachen gibt Tabelle 2.1.

Name der Sprache	Sprachkategorie	Anwendungsgebiet/Bemerkung
Fortran 1954	prozedural	mathematisch-technische Probleme
Algol 60 1958	prozedural	mathematisch-wissenschaftliche Probleme
Cobol 1959	prozedural/ objektorientiert	kaufmännische Probleme; keine klare Definition, unsystematischer Aufbau, eine der am weitestverbreiteten Sprachen; 1997 erweitert zu OO-Cobol mit objektorientierter Konzeption
Lisp 1959	funktional	gut geeignet für sogenannte symbolische Berechnungen
Basic 1963	prozedural	universell; Sprachumfang nicht einheitlich festgelegt
PL/I 1964	prozedural	mathematisch-technische und kaufmännische Probleme; sehr umfangreich
Simula 67 1965	prozedural	mathematisch-wissenschaftliche Probleme und Simulationen
Pascal 1971	prozedural	mathematisch-technische und kaufmännische Probleme; als Lehrsprache bekannt
C 1974	prozedural	systemnahe Programmierung; sehr verbreitet
Modula-2 1976	prozedural/ objektorientiert	mathematisch-technische und kaufmännische Probleme; Weiterentwicklung zu Modula-3 (objektorientiert)
Prolog 1977	prädikativ	Anwendung mit symbolischen Formeln; Einsatz bei Expertensystemen

Tabelle 2.1 Wichtige historische und aktuelle Programmiersprachen im Überblick

Name der Sprache	Sprachkategorie	Anwendungsgebiet/Bemerkung
Ada 1979	prozedural/ objektorientiert	Echtzeitanwendung; Weiterentwicklung von Pascal; Ada-95 um Objektorientierung erweitert
SQL 1970	deklarativ	Datenbankanwendung; seit 1983 genormt
Smalltalk-80 1970	objektorientiert	universell; erste objektorientierte Sprache
C++ 1980	prozedural/ objektorientiert	universell; Obermenge von C
Eiffel 1986	objektorientiert	umfangreiche Softwaresysteme
Pearl 1987	prozedural/ modular/teilweise objektorientiert	plattformunabhängige, interpretierte Scriptsprache
Java 1990	objektorientiert	universell
Haskell 1990	funktional	Anwendungen in der Wissenschaft
Python 1991	multiparadigmatisch	universelle, üblicherweise interpretierte höhere Programmiersprache
R 1993	funktional/ dynamisch/ objektorientiert	freie Programmiersprache für statistische Berechnungen und Grafiken
JavaScript 1995	objektorientiert/ prozedural/ funktional	Scriptsprache für interaktive Web-Anwendungen
C# 2002	objektorientiert	Anwendung auf Microsoft-Plattformen

Tabelle 2.1 Wichtige historische und aktuelle Programmiersprachen im Überblick (Forts.)

Name der Sprache	Sprachkategorie	Anwendungsgebiet/Bemerkung
Go 2009	multiparadig-matisch	entwickelt von Google; kompilierbare Programmiersprache, die Nebenläufigkeit unterstützt und über eine automatische Speicherbereinigung verfügt
Rust 2010	multiparadig-matisch	als sichere, nebenläufige und praxisnahe Sprache von Mozilla Research entwickelt
Swift 2014	multiparadig-matisch	Programmiersprache von Apple für iOS, macOS, tvOS, watchOS und Linux

Tabelle 2.1 Wichtige historische und aktuelle Programmiersprachen im Überblick (Forts.)

2.4.2 Die Systematik der Programmiersprachen

Programmiersprachen lassen nicht nur nach ihrem Verwendungszweck (kaufmännisch, technisch, mathematisch) und ihrer Sprachkategorie (objektorientiert, funktional) einteilen, sondern auch nach ihrem Grad der Abstraktion. Nach diesem Kriterium unterteilt man sie in Sprachen der ersten bis vierten Generation. Sie denken völlig in die richtige Richtung, wenn Sie hier auch eine zeitliche Entwicklung vermuten. Heute kommen primär Programmiersprachen der dritten und vierten Generation zum Einsatz, das heißt aber nicht, dass man eine Sprache einer früheren Generation nicht mehr verwendet. Sehen wir uns diese Einteilung genauer an. Sie gibt auch gleichzeitig einen anderen Blickwinkel auf die Entstehung und Weiterentwicklung der Sprachen.

Sprachen der ersten Generation: Maschinensprachen

Computer waren zu Beginn ausschließlich in *Maschinensprache*, d. h. als Folge von Nullen und Einsen zu programmieren. Befehle und zu bearbeitende Daten wurden dabei in Kombination an den Prozessor gesandt. Typische Operationen sind das Verschieben von Daten aus dem Speicher in bestimmte Register des Prozessors und einfachste arithmetische Operationen, zum Beispiel Addition und Subtraktion, mit den Inhalten der Register. Wie muss man sich einen solchen Programmcode in Maschinensprache vorstellen? Dazu ein Beispiel (Listing 2.1):

```
1: 10110000 01100011
2: (hex) B0 63
3: Operand 0x63 wird in AL-Register geladen
```

Listing 2.1 Beispiel für Maschinensprache

Sie sehen: Auf diese Weise einen Computer zu programmieren, ist sehr aufwendig. Zeile 1 und 2 sind identisch, lediglich einmal in dezimaler und einmal in hexadezimaler Schreibweise. In Zeile 3 steht die Erläuterung, d. h., der betreffende Operand wird in ein bestimmtes Register des Prozessors verschoben. Für komplexere Sachverhalte ist die direkte Programmentwicklung in Maschinensprache nicht zu machen. Man kann sagen, dass die Programmierung in Maschinensprache aus heutiger Sicht fast keine Bedeutung mehr hat. Dennoch ist es für das Gesamtverständnis zum Entstehen eines Computerprogramms sehr wichtig, zu wissen, wie Maschinensprache funktioniert. Letztendlich wird jedes Computerprogramm durch den Compiler bzw. Interpreter in Maschinensprache übersetzt, jedoch eben nicht von Hand. Ein weiteres entscheidendes Merkmal ist, dass sich die konkrete Ausprägung der Maschinensprache von Prozessortyp zu Prozessortyp unterscheidet. Für jeden Prozessortyp muss also ein eigenes Programm in Maschinensprache erstellt werden, damit es die gleichen Aufgaben ausführt. Kommen wir zu den Sprachen der zweiten Generation.

Sprachen der zweiten Generation: Assemblersprachen

Als Programmiersprachen der zweiten Generation werden sogenannte *Assemblersprachen* bezeichnet. Diese sind ebenfalls auf die spezielle Hardware des Computers, d. h. auf den Befehlssatz des Prozessors ausgerichtet. Es erfolgt jedoch eine Ersetzung der hexadezimalen Codes der Maschinensprache durch im Klartext formulierte Anweisungen. Diese Anweisungen beschreiben Operationen mit den Registern des Prozessors. Für einen menschlichen Bearbeiter sind also Programme in Assemblersprachen bei entsprechendem Wissen um die speziellen Erfordernisse des Prozessors verständlich. Assemblerprogramme werden durch einen Übersetzer, der sehr oft als Assembler bezeichnet wird, in die vom Prozessor ausführbare Maschinensprache transformiert. Assemblerprogrammierung ist sehr hardwarenah. Wo wird das heute noch eingesetzt? Überall dort, wo man die direkte Kontrolle über den Programmfluss und die zugrundeliegende Hardware haben muss, beispielsweise bei der Entwicklung von *Hardwaretreibern* oder bei der Programmierung von Software von *Mikrocontrollern*.

Bei der Programmierung von Treibern muss man direkt auf die Ein- und Ausgänge zugreifen. Ebenso muss diese Art von Software meist ohne zeitliche Verzögerung ablaufen (*zeitkritisch*), d. h., es muss die direkteste Programmierung der elektronischen Bauteile gewählt werden. Mikrocontroller sind gewissermaßen »Mini-PCs«, die in unzähligen Geräten unseres alltäglichen Lebens verbaut sind, angefangen von der Kaffeemaschine bis zur Elektronik im Automobil. Die Ressourcen eines einfachen Mikrocontrollers, d. h. Speicher und Rechenleistung, reichen oft nicht aus, um in einer Programmiersprache mit einem höheren Abstraktionsniveau zu entwickeln.

So, wir haben Ihnen darüber so viel erzählt, und jetzt wollen wir Ihnen auch noch zeigen, wie ein solches Assemblerprogramm aussieht. Hier kommt ein Beispiel (Listing 2.2):

```
START
   Btfsc      GPIO, 3     ; Abfrage des Eingangs GPIO3, ob Taste gedrückt
   Goto       START       ; Taste nicht gedrückt, springe zur Marke >START<
   Movfw      einer       ; Laden der Variablen >einer< in das W-Register
   Call       TabellePIN  ; Sprung zum Label >TabellePIN<

   bsf STATUS, RPO        ; Auswahl Bank 1
   movwf      TRISIO      ; Setzen der aktuellen Pin-Konfiguration
   bcf STATUS, RPO        ; Auswahl Bank 0
   movfw      einer       ; Laden der Variablen >einer< in das W-Register
   call       TablleLED   ; Sprung zum Label >TabelleLED<, Laden der Variablen
WARTE
   Btfss      GPIO, 3     ; Abfragen des Eingangs GPIO3, ob Taste gedrückt
   Goto       WARTE       ; Taste gedrückt, springe zur Marke >WARTE<

   Decfsz     einer, 1    ; Tastendruck zählen bis 0 erreicht
   Goto       START       ; Zurück zu >START<, sofern nicht 0
                          ; bei 0 geht es mit der Folgeanweisung weiter
   Movlw      D'8         ; Laden der Konstanten <acht>
   Movewf     einer       ; Erneutes Setzen von >einer< auf >acht<
   Goto       START       ; und von vorn
```

Listing 2.2 Ein Beispiel für einen Programmcode in Assemblersprache

Nun, was sehen wir? Auf jeden Fall wirkt es gleich etwas verständlicher als die reine Maschinensprache. Die Befehle bestehen aus kurzen Buchstabenkombinationen. Das sind beispielsweise `goto` und `call`. Der Befehl `goto` führt einen Sprung aus, das Programm wird an einer anderen Stelle fortgesetzt, und der Befehl `call` ruft ein Unterprogramm auf. Mit etwas Übung kann man sich die Abkürzungen für die Befehle merken. Dabei darf man nicht vergessen, dass es für jeden Prozessor einen eigenen Befehlssatz gibt. Hinter den Befehlen stehen die Argumente, die verarbeitet werden. Das können Zahlenwerte oder Variablen sein. In Assemblerprogrammen kann man also schon ein wenig abstrakt arbeiten, d. h., man kann zum Beispiel Variablen und Sprungmarken definieren. Das erleichtert die Arbeit gegenüber der direkten Codierung in Maschinensprache wesentlich. Ganz wichtig sind die Kommentare in einem Assemblerprogramm. Im Beispiel werden sie durch Semikolon (;) nach jeder Programmanweisung notiert. Auf diese Weise möchte man sicherstellen, dass man das Programm auch noch zu einem späteren Zeitpunkt versteht oder dass die Arbeitsweise von einem anderen Programmierer nachvollzogen werden kann.

Sprachen der dritten Generation: Höhere Programmiersprachen

Programmiersprachen der sogenannten dritten Generation umfassen unterschiedliche Ansätze. Das Ziel dieser Entwicklungen war es, eine neue Ebene der Abstraktion

zu erreichen und weitestgehend unabhängig von der verwendeten Hardware zu programmieren. Diese Art von Programmiersprachen werden daher auch als *höhere Programmiersprachen* bezeichnet. Die ersten Sprachen dieser Art wurden zwischen 1950 und 1960 entwickelt. Alle existierenden Sprachen vorzustellen, würde unter keinen Umständen gelingen und wäre auch wenig sinnvoll. Ebenfalls sind die Verflechtungen und Ableitungen untereinander sehr unübersichtlich und vielfältig. Es werden daher einige Meilensteine auf dem Weg der Entwicklung der Programmiersprachen vorgestellt. Die höheren Programmiersprachen lassen sich den zugrundeliegenden Programmierparadigmen zuordnen. Sehen wir uns die Meilensteine der Entwicklung an: Eine der ersten Sprachen (1954) war *Fortran* (*Formula Translator*). 1959 wurde die Sprache *Lisp* (*List Processor*) entwickelt, und ebenfalls 1959 wurde die Sprache *Cobol* (*Common Business Oriented Language*) für primäre betriebswirtschaftliche Anwendungen entwickelt. Die Sprache *Algol 60* (1960) basierte auf dem Vorläufer *Algol 58* (1958) und hat die Entwicklung und Struktur der Programmiersprachen wesentlich mitbestimmt.

Die Sprache Algol war maßgebend für die Entwicklung weiterer Sprachen. Dazu zählen unter anderem *Pascal*, *Simula* und *C*. Mit der Einführung der Sprache *BASIC* (*Beginner's All-purpose Symbolic Instruction Code*) sollte der Einstieg in die Programmierung für Anfänger erleichtert werden. BASIC, in den Anfängen wegen der oft fehleranfälligen Struktur des erstellten Quellcodes kritisiert, hat dennoch eine beachtliche Erfolgsgeschichte. BASIC-Dialekte existieren für nahezu alle Plattformen. Bekannte Meilensteine waren QuickBASIC und die Entwicklung von Visual Basic. Heute ist Visual Basic .NET eine der wichtigen Programmiersprachen bei der Entwicklung von Anwendungen für das .Net-Framework. Die grundlegenden Sprachmerkmale sind mit denen der ersten Version nach wie vor vergleichbar, was einen Einstieg erleichtert. Hinzugekommen sind professionelle Konzepte wie Objektorientierung und weitere moderne Ausdrucksformen. Dennoch nimmt ihre Bedeutung laut Tiobe-Index (*https://www.tiobe.com/tiobe-index/*) ab.

Die Programmierung wird bis heute durch das objektorientierte Paradigma geprägt. Kernelement ist die Vereinigung von Daten und der zugehörigen Methoden innerhalb eines Objekts. Einen ersten Ansatz in diese Richtung wurde durch die Sprache *Simula 67* gegangen. Simula war eine Spezialsprache zur Simulation von Vorgängen in der realen Welt. Die Sprache *Smalltalk* gilt bis heute als die erste objektorientierte Sprache, da ihr Konzept auf der reinen Objektorientierung beruht. Die Terminologie der heutigen objektorientierten Sprachen beruht zu einem Großteil auf den Begrifflichkeiten in Smalltalk.

Um 1970 wurde durch Niklaus Wirth die Programmiersprache *Pascal* entwickelt. Pascal erlangte sehr schnell Bedeutung in Lehre und Forschung. Der hauptsächliche Grund dafür ist die klare Strukturierung der Sprache und der mit ihr erzeugten Programme. Die Weiterentwicklung von Pascal – primär um objektorientierte Sprachkon-

zepte, aber auch um Anpassungen an neue Entwicklungsmöglichkeiten – führte zu *Object Pascal* und *Delphi*. Delphi steht heute dabei für mehr als eine Programmiersprache. Es handelt sich um einen umfassenden Entwicklungsansatz für die effiziente Erstellung von Anwendungen für unterschiedliche Betriebssysteme. Die Kernelemente der zugrundeliegenden Programmiersprache beruhen jedoch auf Pascal. Auf Pascal beruht ebenfalls die Weiterentwicklung zur Programmiersprache *Modula-2*. Diese verfügt über ein strenges Modul- und Schnittstellenkonzept. Modula-2 wurde zu *Oberon* und später zu *Oberon-2*, einer echten objektorientierten Sprache, ausgebaut.

Die Sprache *C* hat bis heute große Bedeutung. Sie ist weiterhin die häufigste Programmiersprache im Bereich der hardwarenahen Entwicklung wie zum Beispiel Treiber, Betriebssysteme und Software für eingebettete Systeme. *C++* und *Objective-C* sind zwei objektorientierte Erweiterungen von C. C++ war sehr lange die dominierende Sprache bei der Entwicklung von professionellen Anwendungen. Eine neuere Sprachentwicklung ist *Java*. Java verzichtet gegenüber C++ auf das fehleranfällige Zeigerkonzept. Die Plattformunabhängigkeit ermöglicht es, erstellte Anwendungen ohne größere Änderungen in verschiedenen Systemumgebungen zu starten. Voraussetzung dafür ist, dass eine *Laufzeitumgebung* auf dem Zielsystem vorhanden ist. Java wird auf Rechnern unterschiedlichster Kategorien und Leistungsklassen angewendet.

Gehen wir noch einen Schritt in Richtung Zukunft: Weitere, neuere Sprachentwicklungen sind *C#* und *F#*, die für die Softwareentwicklung mithilfe des .Net-Frameworks zur Verfügung stehen. C# ist dabei aus Java hervorgegangen, enthält aber auch Elemente aus C. Die Sprache ist inzwischen ausgereift und stellt den Quasi-Standard für die Entwicklung von Anwendungen mit .Net dar. F# ist eine noch relativ junge Programmiersprache, die das funktionale Paradigma in den Fokus rückt. Eine weitere wichtige Programmiersprache – primär für die Entwicklung von dynamischen Web-Anwendungen – ist die Sprache *PHP*, die auch kontinuierlich weiterentwickelt wird.

Die Entwicklung der Programmiersprachen hat auch in der jüngeren Zeit nicht aufgehört. Neue Sprachen sind beispielsweise *Swift* von Apple (2014), *Go* von Google (2009) und *Rust* von Mozilla (2010).

Wir könnten diese Aufzählung noch sehr lange fortsetzen. Die Verflechtungen der Sprachen untereinander sind intensiv. Dies ist die Folge daraus, dass neue Programmiersprachen aus dem Umstand heraus entwickelt werden, um Probleme auf eine andere Art und Weise zu lösen. Der steigenden Komplexität versucht man entgegenzuwirken, indem man den Grad der Abstraktion erhöht. Die Formulierung einer Lösung erfolgt heute in den meisten Sprachen weitaus mehr problemorientiert als in den Anfängen der Programmentwicklung.

Vielleicht fragen Sie sich, mit welcher Programmiersprache Sie sich beschäftigen sollen? Diese Frage ist gar nicht so einfach zur beantworten. Aufgrund wiederkehrender Konzepte ist es fast ein wenig egal! Suchen Sie sich eine Sprache aus, am besten eine

Sprache, mit der Sie ein konkretes Projekt angehen können, und steigen Sie ein. Bleiben Sie nicht an der Oberfläche, gehen Sie bis zum Grund! Es ist sinnvoller, *eine* Sprache *richtig* zu können, als viele Sprachen nur ein wenig. Im Falle eines notwendigen Wechsels: Fällt das Umlernen dann nicht wirklich schwer? Sollen wir Ihnen etwas empfehlen? Ja, das machen wir! Lernen Sie eine Sprache, die möglichst viele Konzepte des modernen Programmierstils enthält und einen breiten Einsatzbereich umfasst. Dazu gehören beispielsweise Java, C#, JavaScript und Swift.

Java erlaubt einen breiten, plattformübergreifenden Einsatz. Es können Anwendungen aller Art mit Java erstellt werden, und es gibt eine Vielzahl von Entwicklungsumgebungen, die teilweise kostenfrei zur Verfügung stehen. Die Sprache selbst bietet nahezu alle modernen Konzepte der Programmierung. Ihr Fokus liegt auf dem objektorientierten Ansatz. C# wurde von Microsoft entwickelt und wird fortlaufend weiter dem technischen Fortschritt angepasst. Sie ist die erste Wahl für Anwendungen, die auf der Basis .Net-Framework basieren. Zunächst lediglich auf das Ökosystem von Microsoft Windows beschränkt, erweitert sich der Anwendungsbereich stetig auf neue Plattformen und Anwendungstypen. C# ist ebenfalls auf die objektorientierte Programmierung fokussiert; zunehmend finden jedoch auch funktionale Aspekte Eingang in diese Programmiersprache. Vom Namen her klingt JavaScript ähnlich wie Java, sie haben jedoch keine großen Gemeinsamkeiten. JavaScript war anfänglich lediglich eine Skriptsprache. Erweiterungen, Bibliotheken und Frameworks haben jedoch dazu geführt, dass bei der modernen Webprogrammierung an JavaScript kein Weg mehr vorbeigeht. Wenn Sie sich heute für moderne dynamische Web-Applikationen jeder Größenordnung interessieren, dann führt an dem Tripel aus Seitenbeschreibungssprache *HTML*, Auszeichnungssprache *CSS* und JavaScript für die Implementierung der Logik kein Weg mehr vorbei. Bleiben wir beim Web: Unzählige *Content Management Systeme* und dynamische Web-Applikationen sind in PHP geschrieben. Auch diese Sprache hat sich stets weiterentwickelt und bietet einfache Möglichkeiten, mit Datenbanken umzugehen.

Nach dieser kurzen Charakterisierung der Programmiersprachen der dritten Generation sehen wir uns nun noch Sprachen an, die als Programmiersprachen der vierten oder höheren Generation bezeichnet werden.

Sprachen der vierten und höheren Generation

Programmiersprachen der vierten und höheren Generation unterliegen keinem speziellen Paradigma. Es handelt sich um Sprachen mit einem höheren Abstraktionsniveau, die auf spezielle Anwendungsbereiche ausgerichtet sind. In diese Kategorie fallen zum einem Skriptsprachen wie zum Beispiel *SQL* (Datenbankabfragesprache) oder *TeX* (Seitenlayoutsprache). Zum anderen gehören deskriptive Sprachen dazu. Diese Sprachen dienen der Beschreibung von Inhalten, weisen jedoch keine Pro-

grammlogik auf. Beispiele sind *XML* oder *HTML*. Mit beiden Sprachen werden Sie als Softwareentwickler noch sehr viel zu tun haben. HTML ist die Beschreibungssprache für alle Internetseiten. Jede Webseite, die im Browser angezeigt wird, verarbeitet HTML. XML hat sich als universelles Format zur Beschreibung von Datenstrukturen jeder Form und Größe etabliert. Sowohl HTML- als auch XML-Dokumente können dabei manuell oder automatisch erstellt werden.

Nach diesem Abriss zur Einteilung der Programmiersprachen sehen wir uns jetzt bestimmte Sprachmerkmale an, die in nahezu jeder Programmiersprache vorhanden sind.

2.5 Essenzielle Sprachmerkmale

Trotz der großen Vielfalt und Unterschiedlichkeit der Programmiersprachen gibt es bestimmte essenzielle Sprachmerkmale, die in allen Sprachen in ähnlicher Form vorhanden sind. Hat man einmal die Funktionsweise und die häufigsten Anwendungsfelder dieser wesentlichen Sprachmerkmale verstanden, so fällt es deutlich leichter, sich mit der spezifischen Syntax der einen oder anderen Programmiersprache auseinanderzusetzen und diese zu erlernen. Ein Beispiel: Alle modernen Programmiersprachen verfügen über die Syntax, bestimmte Schleifenkonstruktionen abzubilden. Natürlich ist die konkrete Ausdrucksweise in jeder Sprache etwas anders, aber die Konzepte und Verwendung bleiben identisch.

In diesem Abschnitt möchten wir Ihnen wichtige Sprachmerkmale vorstellen. Damit dies nicht ganz abstrakt bleibt, zeigen wir die konkrete Umsetzung anhand von Quellcodeausschnitten in den Sprachen Java bzw. C#. Beides sind universell einsetzbare Programmiersprachen mit einem hohen Verbreitungsgrad, und sie liegen schon in einer fortgeschrittenen Entwicklungsstufe vor. Ebenso werden beide Sprachen weiterhin aktiv weiterentwickelt.

Als erstes Sprachmerkmal beschäftigen wir uns mit Kommentaren, also damit, wie man einen Quelltext bzw. einen Ausschnitt eines solchen mit Anmerkungen versieht.

2.5.1 Kommentare

Kommentare sind Notizen, die direkt im Quelltext eingefügt und durch den Compiler beim Übersetzungsvorgang ignoriert werden. Sie haben mit der eigentlichen Programmerstellung nichts zu tun, sie dienen später dazu, den Programmcode besser zu verstehen. Zwar sollte der Quelltext grundsätzlich alleine gut verständlich sein, dennoch ist es häufig angezeigt, Erläuterungen zu ergänzen. Dies geschieht in Form von Kommentaren direkt im Quelltext.

Auch wenn Programmierer sich in fremden Quellcode einarbeiten müssen, sind sie über Kommentare dankbar. Mit Kommentaren sollte man nicht sparsam umgehen, auch wenn es beim Codieren etwas mehr Aufwand verursacht; diese Zeit spart man später um ein Vielfaches ein, wenn es gilt, den Quelltext erneut zu analysieren.

Wie sehen Kommentare in der Sprache Java aus? Innerhalb des Quelltexts wird ein Kommentar wie folgt geschrieben:

```
a = b + c; // die Zahlen b und c werden addiert und a zugewiesen
```

Der schräge Doppelstrich (//) beschränkt einen Kommentar auf eine Zeile. Soll ein Kommentar über mehrere Zeilen gehen, ist folgende Syntax zu wählen (Listing 2.3):

```
/* Beginn des Kommentars
   Zeile 2 des Kommentars
   Zeile 3 des Kommentars
*/
```

Listing 2.3 Mehrzeiliger Kommentar in Java

Mehrzeilige Kommentare dienen dazu, einen Sachverhalt umfangreicher zu erläutern. Man findet diese Form von Kommentaren üblicherweise am Beginn einer Quelltextdatei.

Und wie kommentiert man C#? Beide Sprachen machen keinen Unterschied! Wie das folgende Beispiel zeigt, funktioniert das Kommentieren hier analog zu Java. Man unterscheidet ebenso ein- und zweizeilige Kommentare. Und wieder ein Beispiel für einen einzeiligen Kommentar:

```
private float nettoBetrag; // Betrag nach der Kalkulation
```

Und das Beispiel für den zweizeiligen Kommentar (Listing 2.4):

```
/* Der folgende Quelltext dient
der Berechnung der Zwischensumme.
*/
```

Listing 2.4 Mehrzeiliger Kommentar in C#

Nicht für alle Kommentare sind wir als Entwickler selbst verantwortlich. Manche erscheinen dankenswerterweise wie von selbst in unseren Quelltextdateien. Bei der Programmierung verwenden wir in der Regel eine integrierte Entwicklungsumgebung, die uns bei den vielfältigen Aufgaben des Schreibens und der Strukturierung des Quelltexts hilft. Erstellt man eine neue Datei, so wird diese aus einer Vorlage generiert und mit Kommentaren angereichert, sodass man sich in dieser schneller zurechtfindet. Abbildung 2.19 zeigt ein Beispiel.

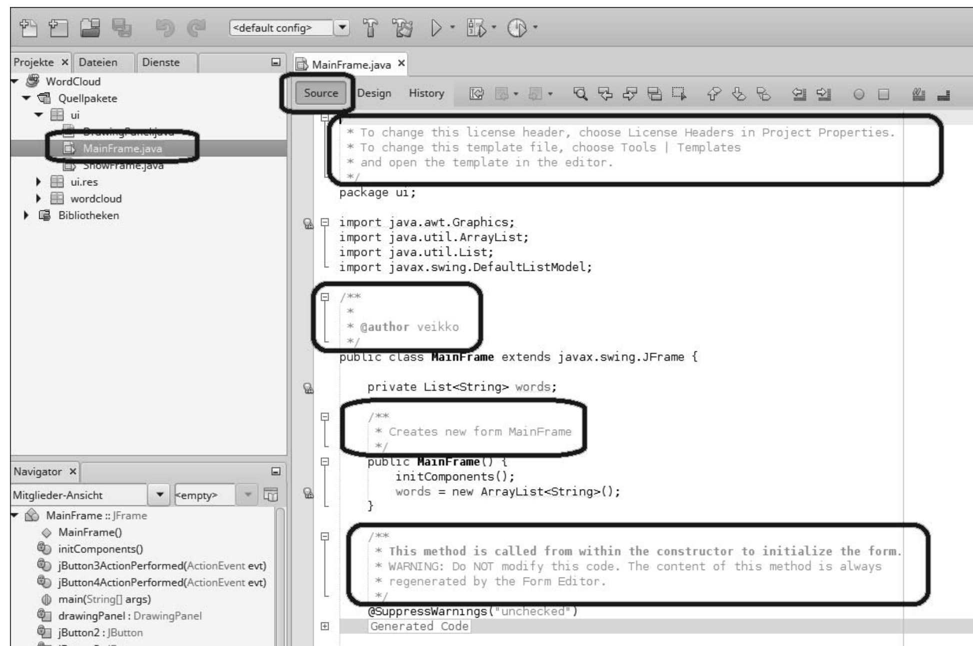

Abbildung 2.19 Automatisch generierte Kommentare durch die Entwicklungsumgebung, hier durch IDE NetBeans

Ebenso dienen Kommentare dazu, bei der weiteren Entwicklung dem Programmierer live während des Schreibens des Programmcodes entsprechende Hilfe anzubieten. Dieses Feature der Entwicklungsumgebungen nennt man *IntelliSense*. Wir zeigen Ihnen die Arbeitsweise am Beispiel von Visual Studio. Zunächst betrachten wir den Quellcode in Listing 2.5:

```
/// <summary>
/// Objekte dieser Klasse sind eine Person.
/// </summary>
public class Person
    {
        /// <summary>
        /// Der Nachname.
        /// </summary>
        public string Name { get; set; }
        /// <summary>
        /// Der Vorname.
        /// </summary>
        public string FirstName { get; set; }
    }
```

Listing 2.5 Klasse in C# mit Kommentar für IntelliSense

Wir sehen eine Klasse `Person`, mit zwei Attributen `Name` und `FirstName`. Sowohl die Klasse als auch die Attribute werden mit speziellen Kommentaren versehen. Gegenüber den üblichen Kommentaren in C# werden diese durch `///` gekennzeichnet und in den Tags `<summary>` und `</summary>` eingeschlossen. Diese Form der Kommentare zeigt dem Compiler, dass er diese innerhalb der Entwicklungsumgebung direkt bei der Erfassung von Quellcode verwenden kann.

Verwendet man jetzt die Klasse `Person` bzw. deren Methoden, so erhält der Entwickler direkt eine Hilfe. Wie man in Abbildung 2.20 sehen kann, erhält man bei Eingabe des Klassennamens direkt die verfügbaren Attribute und Methoden angezeigt. Für das Attribut `Name` wird so gleich der eben definierte Hinweistext eingeblendet. Dieses Feature hilft also direkt dem Entwickler bei der Erfassung des Quellcodes. Damit man es effektiv einsetzen kann, sollte der Entwickler die eigenen Klassen auch entsprechend den Vorgaben der Entwicklungsumgebung mit Kommentaren versehen.

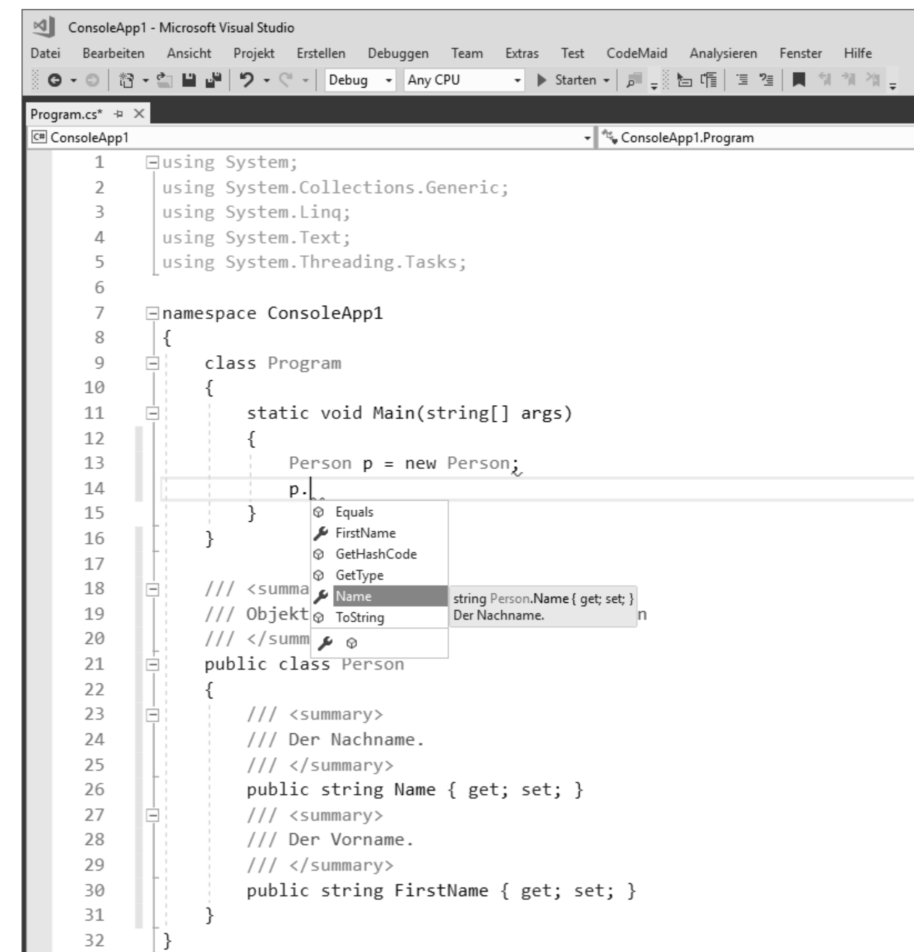

Abbildung 2.20 IntelliSense verwendet direkt die Kommentare der Klasse.

Andere Programmiersprachen und Entwicklungsumgebungen verwenden ähnliche Systeme. Machen Sie sich mit dem jeweiligen System vertraut, nur dann kann Ihnen die Entwicklungsumgebung die bestmögliche Unterstützung bieten. Im Ergebnis heißt das: Kommentieren Sie Ihren Quellcode, wo immer sie es für notwendig erachten. Ein Zuviel gibt es hier eigentlich nicht.

Vielleicht stellen Sie sich an dieser Stelle die Frage, in welcher Sprache Sie die Quelltextkommentare schreiben sollten. Hier möchten wir Ihnen eine klare Empfehlung geben: Notieren Sie alle Kommentare und Hinweise in der Ihrem Team am meisten vertrauten Sprache. Wenn Sie in multinationalen Teams arbeiten oder beispielsweise Auszüge aus Ihrem Quellcode als Bibliotheken anderen zur Verfügung stellen möchten, dann empfiehlt sich ein Kommentieren in englischer Sprache. Im Übrigen gibt es Tools, die eine teilweise automatische Kommentierung der Klassen und ihrer Methoden erlauben. Dazu wird aus dem Namen der Attribute und Methoden ein kurzer Standardtext generiert, der in vielen Fällen als Kommentar brauchbar ist. Voraussetzung für eine solche automatische Kommentarfunktion ist, dass man sich an bestimmte Konventionen bei der Namensvergabe für Methoden und Attribute hält.

2.5.2 Operatoren und Vergleiche

Jetzt wird es ein kleines bisschen mathematisch. Sie wissen ja: Ein Computer ist eigentlich ein Rechner, da intern alle Befehle auf grundlegende Rechenoperationen zurückgeführt werden. Wir kommen zu Operatoren wie zum Beispiel der Verknüpfung von zwei Zahlen oder Variablen mithilfe der Grundrechenarten. Definieren wir also: *Operatoren* verknüpfen Variablen miteinander und führen Berechnungen aus. Sehen wir uns die wichtigsten Ausdrucksformen in Java und in C# an.

Java kennt mathematische Operatoren für die Grundrechenarten sowie die sogenannte Modulo-Operation. Dabei wird der Rest einer Division von zwei ganzen Zahlen ermittelt. Tabelle 2.2 stellt diese Grundrechenarten jeweils inklusive eines Beispiels vor.

Operator	Bedeutung	Beispiel
`+`	Addition	`var = 3 + 4; // var = 7;`
`-`	Subtraktion	`var = 3 - 4; // var = -1;`
`*`	Multiplikation	`var = 3 * 4; // var = 12;`

`/`	Division	`int var = 7 / 4; // var = 1;`
		`double var = 7.0 / 4.0; // var = 1.75;`

`%`	Modulo (Rest einer Division)	`var = 7 % 4; // var = 3;`

Tabelle 2.2 Die Syntax der Grundrechenarten in Java

In Java gibt es auch die üblichen Vergleichsoperationen, beispielsweise zwischen zwei Zahlenwerten. Diese sind in Tabelle 2.3 aufgeführt.

Operator	Vergleich	Beispiel
`==`	Gleichheit	`if (3 == 1) // false`
`!=`	Ungleichheit	`if (3 != 1) // true`
`<`	Kleiner als	`if (3 < 1) // false`
`<=`	Kleiner gleich	`if (2 <= 1) // false`
`>`	Größer als	`if (2 > 1) // true`
`>=`	Größer gleich	`if (1 >= 1) // true`

Tabelle 2.3 Vergleichsoperatoren in der Sprache Java

Die Beispiele dürften selbsterklärend sein. Zu beachten ist dabei der Operator für die Identität (Gleichheit). Hier kommt das doppelte Gleichheitszeichen (`==`) zum Einsatz, da das einfache Gleichheitszeichen (=) als Zuweisungsoperator dient. Schreiben Sie also `if (a = 3)`, so wird immer der Wert `true` zurückgegeben, denn die Zuweisung der Zahl 3 zur Variablen *a* schlägt nicht fehl. Wollen Sie dagegen prüfen, ob die Variable *a* den Wert 3 hat, müssen Sie `if (a == 3)` schreiben.

In einem zweiten Schritt sehen wir uns die Möglichkeiten an, die uns C# bietet. Erfreulich hier: Es gibt keine Unterschiede zur Programmiersprache Java. Die Syntax ist also identisch, sodass wir diese hier nicht nochmals aufführen müssen. Wichtig und interessant ist noch die Frage nach der Priorität. Sieht ein Ausdruck mehrere Operatoren vor, so stellt sich die Frage nach der Reihenfolge der Bearbeitung. Hier gelten die bekannten Zusammenhänge aus der Mathematik, also zum Beispiel ein Vorgang der Multiplikation und Division (Punktrechnung) vor der Addition und Subtraktion (Strichrechnung). Mittels Klammern kann die Reihenfolge der Bearbeitung den eigenen Anforderungen angepasst werden. Insgesamt ergibt sich folgende Priorität der Operatoren:

- Klammern: `()`
- logisches Not: `!`
- Multiplikation: `*`
- Division: `/`
- Modulo: `%`
- Addition: `+`
- Subtraktion: `-`

Es folgen die Vergleichsoperatoren:

- kleiner als: <
- kleiner gleich: <=
- größer als: >
- größer gleich: >=
- gleich: ==
- ungleich: !=

Bei einer Reihung von Operationen gleicher Priorität gilt das bekannte Prinzip von links nach rechts.

2.5.3 Kontrollstrukturen

Vom Rechnen allein wird der Programmierer nicht glücklich. Um den Programmfluss zu kontrollieren, benötigt man sogenannte *Kontrollstrukturen*. Damit kann man die Auswahl aus einer Anzahl von Möglichkeiten nach einem bestimmten Kriterium vornehmen (Auswahlstrukturen) oder Wiederholungen (Schleifen) veranlassen. Wir werden jetzt beide Varianten näher beschreiben und uns dabei wieder auf Beispiele aus den Sprachen Java und C# stützen. Los geht es mit den Auswahlstrukturen.

Auswahlstrukturen

Der Name ist hier Programm, d. h., es geht um die Auswahl zwischen verschiedenen Alternativen. Beginnen wir mit einem einfachen Beispiel.

Blicken wir als Erstes in den oberen Teil der Abbildung 2.21. Als Darstellung haben wir übrigens ein sogenanntes Aktivitätsdiagramm gewählt, das auch ein Diagrammtyp der UML ist. Das spielt jedoch jetzt keine weitere Rolle. Wir sind sicher, die Syntax ist selbsterklärend. Die Auswahl zwischen zwei Alternativen ist also stets eine Entscheidung. Konkret wird es im unteren Teil von Abbildung 2.21. Nach dem Putzen des Fensters betrachten wir nochmals unser Ergebnis. Die Frage lautet: Ist das Fenster sauber, oder haben wir noch Schlieren? Ist es nicht verschmiert, dann sind wir fertig, können uns einen Kaffee holen und weiter programmieren. Der Vorgang bzw. Prozess ist abgeschlossen und endet. Gibt es noch Mängel, dann müssen wir den Putzlappen schwingen und es blitzeblank wienern. Damit dürfte das Prinzip einer Auswahlentscheidung vollständig klar sein. Kommen wir zur Umsetzung in Java und C#. Zuständig ist die `if-else`-Anweisung in beiden Sprachen. Die Syntax unterscheidet sich auch dieses Mal nicht. Andere Sprachen verwenden eine ähnliche Schreibweise. Wichtig ist das Prinzip!

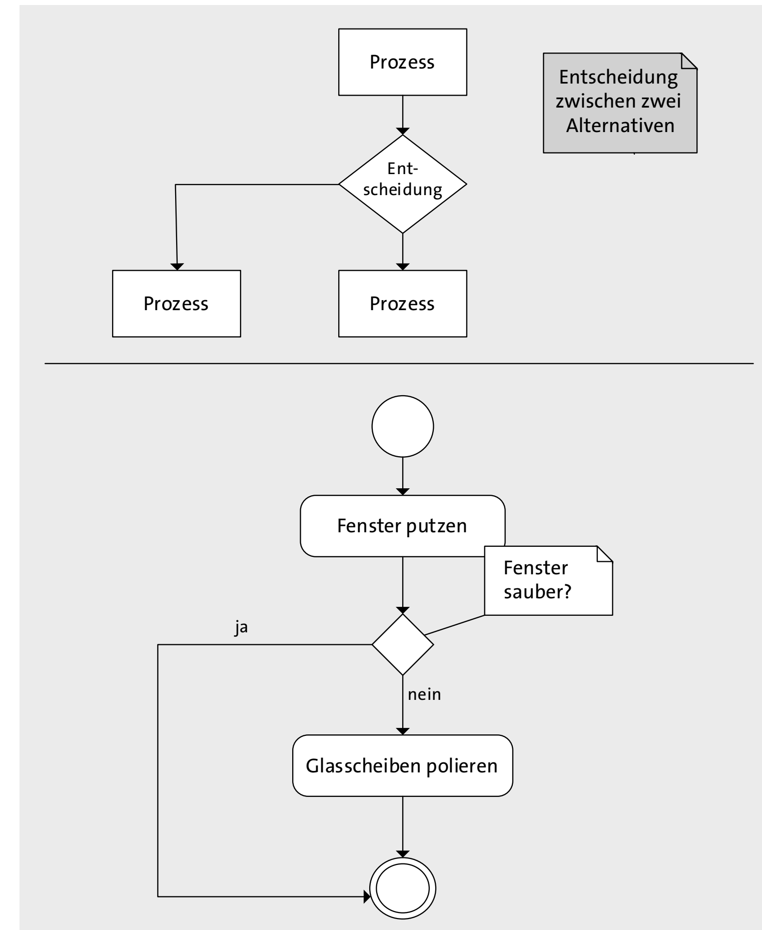

Abbildung 2.21 Das Prinzip einer Auswahlstruktur (Aktivitätsdiagramm)

Die einfache `if`-Schreibweise ermöglicht die alternative Ausführung von Anweisungen. Deren allgemeine Syntax lautet (Listing 2.6):

```
if (Bedingung)
....Anweisungsblock 1
[else
....Anweisungsblock 2]
```

Listing 2.6 Allgemeine Syntax der if-else-Anweisung in C# und Java

Sofern die Bedingung hinter dem Schlüsselwort `if` erfüllt ist, kommt es zu einer Ausführung von Anweisungsblock 1. Der `else`-Zweig ist optional und ist daher in eckigen Klammern dargestellt. Ist er vorhanden, werden die zugehörigen Anweisungen ausgeführt, die in diesem Block integriert sind. Mehrere Anweisungen sind in die üblichen

geschweiften Klammern { und } zu einem Block einzubinden. Ist die Entscheidung aus mehr als aus zwei Alternativen zu treffen, so ist es möglich, if-else-Konstrukte zu schachteln. Auch dazu ein Beispiel (Listing 2.7):

```
if ((Tag == 24) && (Monat == 12))
{
....Ergebnis = "Es ist Weihnachten."
}
elseif ((Tag == 6) && (Monat == 12))
{
........Ergebnis = "Es ist Nikolaus."
}
else
{
      Ergebnis = "Es ist ein anderer schöner Tag."
}
```

Listing 2.7 Beispiel für eine if-else-Konstruktion in C# und Java

Für die zu prüfende Bedingung gilt lediglich, dass diese vom Typ boolean (true oder false) sein muss. Typischerweise kommen die Relationszeichen »==«, »!=«, »<« und »>« zum Einsatz. Zwei Beispiele:

```
if (a == 2) und
if (a != 2)
```

Im ersten Fall ist die Bedingung wahr, wenn die Variable genau den Wert 2 aufweist, im zweiten Fall fällt die Prüfung immer dann positiv aus, wenn *a* ungleich 2 ist.

Für weniger komplexe Situationen können if-else-Verzweigungen mithilfe des Bedingungsoperators ?: ausgedrückt werden. Dieser erwartet als einziger Operator *drei* Operanden: Eine Bedingung und zwei Ausdrücke, von denen je nach Ergebnis der Bedingung einer ausgeführt wird. Dieser Operator wird in der Praxis nach unseren Erfahrungen weniger verwendet, obwohl er bei einfachen Prüfungen eine kompaktere Schreibweise erlaubt. Programmierer lieben eigentlich eine solche Ausdrucksweise, denn damit spart man wertvolle Zeichen Quelltext :-). Kommen wir zur allgemeinen Syntax:

```
Bedingung ? Ausdruck1: Ausdruck2;
```

Ein Beispiel ist hoffentlich wieder etwas aussagekräftiger:

```
label1.Text = checkbox1.Checked ? "Rot" : "Grün";
```

Diese Befehlszeile liest man wie folgt: Ist die Checkbox ausgewählt, dann ist der Rückgabewert »Rot«, ist das nicht der Fall, dann lautet der Rückgabewert »Grün«. Wir können diesen Umstand auch mithilfe der if-else-Anweisung darstellen. Sie lautet in diesem Fall:


```
if (checkbox.Checked == true) label1.Text = "Rot"
   else label1.Text = "Grün";
```

Wie gesagt, welche Darstellung Sie wählen, das ist Geschmackssache. Bei einer Vielzahl von Verzweigungen ist die Konstruktion mit if-else und einer mehrfachen Verschachtelung wenig übersichtlich. Die meisten modernen Programmiersprachen, so auch C# und Java, kennen darüber hinaus ein weiteres Konstrukt für die Fallunterscheidung. Gemeint ist die sogenannte *switch-Verzweigung*. Mit deren Hilfe kann ein Ausdruck, zum Beispiel eine Variable, mit mehreren möglichen Werten verglichen werden. Die zu unterscheidenden Werte sind in sogenannte case-Blöcke geordnet. Trifft kein case-Block zu, werden die Anweisungen nach dem Schlüsselwort default – hier in der Bedeutung »für alle anderen Fälle« – ausgeführt. Auch hier, wer hätte das gedacht, sind C# und Java bezüglich der Schreibweise identisch. Die allgemeine Syntax einer switch-Verzweigung lautet (Listing 2.8):

```
switch (Prüfausdruck)
{
[case Wert1:]
........// Anweisungen für Fall 1
........break;

[case Wert2:]
........// Anweisungen für Fall 2
........break;

[case Wert3:]
........// Anweisungen für Fall 3
........break;


   …
   …
   …
[default:]
       // Anweisungen für diejenigen Fälle,
       // in denen keine der vorhergehenden Konstellationen
       // erfüllt ist.
       break;
}
```

Listing 2.8 Syntax der switch-Verzweigung in C# und Java

Machen wir es wieder mithilfe eines Beispiels etwas plastischer (Listing 2.9).

```
switch (int Schulnote)
{
   case 1:
       Ergebnis = "Eine sehr gute Leistung"
       break;
    case 2:
       Ergebnis = "Eine gute Leistung"
       break;
    case 3:
       Ergebnis = "Noch akzeptabel, aber bitte noch mehr üben"
       break;
    case 4:
       Ergebnis = "Gerade noch bestanden"
       break;
   case 5:
       Ergebnis = "Viel schlechter geht es nicht"
       break;
   case 6:
       Ergebnis = "Es kann nicht schlechter werden"
       break;
   default:
       Ergebnis = "Diese Note gibt es nicht"
       break;
}
```

Listing 2.9 Ein Beispiel für eine switch-Verzweigung in C#

Ich denke, dieses Beispiel müssen wir nicht großartig erläutern. Die `break`-Anweisung in jedem `case`-Block sorgt dafür, dass die Überprüfung endet, wenn der betreffende Fall zutrifft. Die Schulnote ist ein ganzzahliger numerischer Wert. Je nachdem, ob wir eine Eins, eine Zwei usw. bekommen haben, wird eine verbale Einschätzung in Form einer Zeichenkette zurückgegeben. Positiv ist ebenfalls die Möglichkeit, mehrere Fallkonstellationen zusammenzufassen: Sollen beispielsweise bei den Werten `a == 3` und `a == 5` die gleichen Anweisungen ausgeführt werden, so können zwei `case`-Schlüsselwörter untereinanderstehen (Listing 2.10):

```
switch (a)
{
   case 3:
   case 5:
       {Anweisungen...}
}
```

Listing 2.10 Zusammenfassen von mehreren Fällen zu einer Prüfung (C#)

Für den Prüfausdruck ist eine Besonderheit zu beachten: Im Gegensatz zur `if-else`-Konstruktion darf es sich lediglich um einen ordinalen Wert, eine ganze Zahl, einen Aufzählungstyp oder eine Zeichenkette (`string`) handeln. Nicht zulässig sind beispielsweise Gleitkommazahlen oder Dezimalzahlen. Diese Bedingung schränkt die Verwendung der switch-Verzweigung in der Praxis etwas ein. Dennoch sollten Sie diese, wenn möglich, gegenüber dem `if-else`-Konstrukt bevorzugt verwenden. Jeden `case`-Block einer `switch`-Verzweigung müssen Sie mit dem Befehl `break` abschließen. Diese Anweisung soll signalisieren, dass die Prüfung abgeschlossen ist, wenn eine Fallkonstellation zutrifft und ausgeführt wurde.

Was ist mit goto?

In den Anfängen der Programmierung, insbesondere in den ersten Dialekten der Sprache BASIC, wurde immer wieder der unmittelbare und direkte Sprungbefehl `goto` verwendet. Die `goto`-Anweisung gehört auch heute noch zum Sprachumfang vieler Programmiersprachen. Den Einsatz dieses Befehls müssen Sie jedoch vermeiden! Er führt zu schwer lesbarem und schlecht wartbarem Code, sogenanntem »Spaghetti-Code«. Es gilt: Alle auftretenden Programmiersituationen lassen sich im Normalfall durch andere Sprachkonstruktionen ersetzen!

Kommen wir jetzt zum Aufbau von Schleifen, die ebenfalls ein sprachliches Mittel sind, um den Programmfluss zu kontrollieren.

Schleifen

Bei der Anwendung von *Schleifen* in einem Programm geht es um die Wiederholung von bestimmten Vorgängen. Diese Vorgänge können identisch sein oder in Bestandteilen sich von Wiederholung zu Wiederholung nach einem bestimmten Muster ändern. Diese Variation wird durch den Schleifendurchlauf selbst bestimmt und beispielsweise durch die Schleifenvariablen gesteuert. In der Theorie existieren verschiedene Schleifentypen, die auch Eingang in die Programmiersprachen gefunden haben. Wir wollen sie Ihnen jetzt vorstellen. Bezüglich der Systematik können zwei Arten von Schleifen unterschieden werden: Die kopfgesteuerten und die fußgesteuerten Schleifen.

Wird die Bedingung, ob die Schleife ein weiteres Mal durchlaufen wird, *vor* dem Schleifendurchlauf geprüft, so spricht man von einer *abweisenden Aussagenschleife* bzw. einer *kopfgesteuerten Schleife*. Ist die Bedingung schon bei der ersten Prüfung nicht erfüllt, so wird die Schleife niemals durchlaufen. Sehen Sie sich dazu bitte Abbildung 2.22 an, die ein Struktogramm für diesen Schleifentyp darstellt.

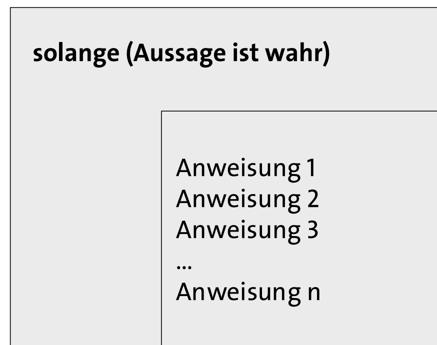

Abbildung 2.22 Das Prinzip der kopfgesteuerten Schleife

Dagegen werden *nichtabweisende Schleifen* mindestens einmal durchlaufen. Sie werden auch als *fußgesteuerte Schleifen* bezeichnet. Das Abbruch- bzw. Terminationskriterium wird am Ende des Schleifendurchlaufs überprüft (Abbildung 2.23).

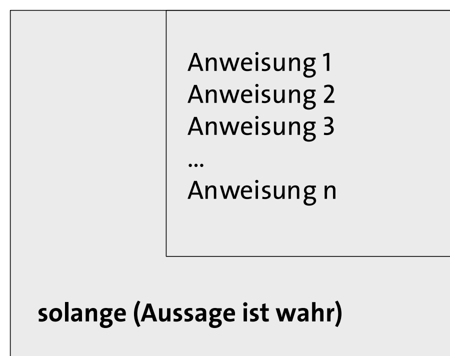

Abbildung 2.23 Das Prinzip der fußgesteuerten Schleife

Welchen Schleifentyp sollen Sie denn nun einsetzen? Hier gibt es keine generelle Antwort! Die Entscheidung ist zum einen von der Aufgabe abhängig und zum anderen vom persönlichen Stil des Entwicklers. Mit anderen Worten: Eine Programmieraufgabe kann mit verschiedenen Schleifentypen umgesetzt werden. Die kommenden Beispiele verdeutlichen das. Es gilt eine möglichst verständliche und übersichtliche Vorgehensweise zu finden. Die Verständlichkeit des Quelltexts steht dabei im Vordergrund.

while-Schleife

Kommen wir wieder zur Umsetzung in den Programmiersprachen. Die kopfgesteuerte Schleife, auch als while-Schleife bezeichnet, hat in C# und Java folgende allgemeine Syntax (Listing 2.11):


```
while (Bedingung == true)
{
    // Schleifenkörper
}
```

Listing 2.11 Syntax der kopfgesteuerten Schleife in C# und Java

Auch dazu sehen Sie sich bitte folgendes Beispiel an (Listing 2.12):

```
string[] Laender = { "Deutschland", "Italien", "Spanien", "Portugal" };
int i = 0;
while (i < 4)
{
   textBox.Text = Laender[i];
   i++;
}
```

Listing 2.12 Beispiel für eine while-Schleife in C#

Was passiert hier? Wir definieren ein *Array*, das aus vier Ländernamen besteht. Hintergrund: Ein Array ist ein spezieller Datentyp, um gleichartige Daten zu speichern. Auf die Elemente kann mittels Index zugegriffen werden, dabei hat das erste Element den Index 0. Der Wert von `Laender[1]` lautet also «Italien«. Arrays stellen wir in Kapitel 3, »Algorithmen und Datenstrukturen«, genauer vor. Jetzt kommen wir zurück zur Schleife. Ebenso definieren wir eine Variable *i*, die wir zu Beginn mit dem Wert 0 initialisieren. Die Schleife wird mit der Prüfung `while (i < 4)` eingeleitet. Hier wird geprüft, ob die Variable einen Wert von kleiner als 4 hat. Ist dies der Fall, wird die Schleife (erneut) durchlaufen. Innerhalb der Schleife passiert dann Folgendes: Einer Textbox wird der aktuelle Werte des Arrays `Laender` zugewiesen. Ebenso wird in jedem Schleifendurchlauf der Wert der Variablen *i* um eins erhöht (`i++`). Danach wird erneut geprüft, ob *i* noch kleiner als 4 ist. Ist diese Bedingung nicht mehr erfüllt, so wird die Schleife nicht mehr durchlaufen, und es geht mit dem Code nach der Schleife weiter. Damit die Schleife also nicht endlos durchlaufen wird, d. h. das Computerprogramm faktisch abgestürzt ist, muss man sicherstellen, dass innerhalb der Schleife die Abbruchbedingung irgendwann erfüllt wird. In diesem Fall muss sich also die Variable *i* anpassen. Zum Thema Endlosschleifen kommen wir am Ende des Abschnitts nochmals zurück.

for-Schleife

Eine besondere Form der kopfgesteuerten Schleife ist die *for-Schleife*. Auch diese Schleifenform gibt es in nahezu allen Programmiersprachen. Es handelt sich um eine sogenannte *Zählschleife*. Bezüglich der Syntax ist es die einfachste Schleifenform. Die Anweisungen innerhalb des Schleifenkörpers werden in einer bestimmten Anzahl

durchgeführt. Bereits am Schleifenkopf wird definiert, wie oft die Schleife durchlaufen wird. Die Schleifenvariable kann innerhalb des Schleifenkopfs deklariert werden. Nach jedem Schleifendurchlauf findet im Regelfall eine Erhöhung der Schleifenvariablen um den Wert 1 statt. Innerhalb des Schleifenkörpers können Operationen einen Rückgriff auf diese Schleifenvariable durchführen. So können Berechnungen mithilfe der Schleifenvariablen ausgeführt oder Werte eines Arrays ausgelesen werden. Zur Verdeutlichung folgt ein einfaches Beispiel in der Sprache C# (Listing 2.13):

```
string[] Laender = { "Deutschland", "Italien", "Spanien", "Portugal" };
for (int i = 0; i < 4; i++)
{
....textBox.Text = Laender[i];
}
```

Listing 2.13 Ein Beispiel für eine for-Schleife in C#

Dieser Quellcode in Listing 2.13 führt zu exakt dem gleichen Ergebnis wie die while-Schleife aus Listing 2.12. Sehen wir uns die Sache noch etwas genauer an: Die Schleifenvariable lautet *i* und wird vom Typ `int` definiert. Sie wird mittels der Anweisung `i++` bei jedem Durchlauf erhöht (inkrementiert). Der Vorgang wird solange wiederholt, bis die Bedingung `i < 4` nicht mehr erfüllt ist. Innerhalb der Schleife wird das Element mit dem Index *i* des Arrays `Laender` ausgegeben. Die Schleifenvariable sollte vom Typ eine ganze Zahl sein, um Fehler durch Rundungen zu vermeiden. Im Beispiel wurde die Schleife vorwärts durchlaufen, beim ersten Durchlauf hatte die Variable den Wert 0, dann 1, dann 2 usw.

Natürlich ist es auch möglich, die Schleifenvariable bei jedem Durchlauf zu vermindern und damit die Elemente in umgekehrter Reihenfolge anzusprechen (Listing 2.14):

```
string[] Laender = { "Deutschland", "Italien", "Spanien", "Portugal" };
for (int i = 4; i >= 1; i--)
{
    textBox.Text = Laender[i];
}
```

Listing 2.14 for-Schleife mit umgekehrten Durchlauf in C#

Je nach Situation in der Praxis kann man die Schleife also vor- oder rückwärts durchlaufen. Sehr häufig verwendet man den Variablennamen *i* als Schleifenvariable. Schachtelt man for-Schleifen ineinander, so nimmt man als Zählvariablen dann die Buchstaben *i*, *j* usw. Natürlich kann man als Schleifenvariablen auch andere Bezeichnungen verwenden, die aussagekräftiger sind und damit die Lesbarkeit des Quellcodes erhöhen.

Greift man innerhalb einer for-Schleife über einen Index auf eine Liste, wie im Beispiel auf ein Array, zu, so besteht die Gefahr, dass ein Zugriff auf die Datenstruktur außerhalb des zulässigen Indexbereichs durchgeführt wird. Das wäre dann der Fall, wenn man beispielsweise definieren würde, dass die Schleife 7-mal durchlaufen werden soll, jedoch im Array nur 4 Elemente gespeichert sind. In diesem Fall würde es zu einem Fehler führen, den man erst zur Laufzeit des Programms feststellen könnte. Was kann man dagegen machen? Man kann es vermeiden, indem der Grenzwert aus der Datenstruktur selbst bestimmt wird. Werden die Elemente eines Arrays durchlaufen, so könnte man die Anzahl der Elemente des Arrays ermitteln. Hierzu ein kleiner Exkurs: Mit der Funktion `Length` kann man in C# die Anzahl der Elemente des Arrays bestimmen. Konkret also mit `Laender.Length`. Damit können wir den Quellcode etwas verbessern (Listing 2.15).

```
string[] Laender = { "Deutschland", "Italien", "Spanien", "Portugal" };
for (int i = 0; i < Laender.Length; i++)
{
....textBox.Text = Laender[i];
}
```

Listing 2.15 Die Anzahl der Schleifendurchläufe wird im Schleifenkopf automatisch bestimmt (C#).

foreach-Schleife

Moderne Varianten der Programmiersprachen bieten oft eine weitere Sonderform der kopfgesteuerten Schleife, genauer die *foreach-Schleife*. Mit der foreach-Schleife kann nacheinander jedes Element einer Liste durchlaufen werden. Natürlich ist es auch möglich, dies mit anderen Schleifenkonstruktionen nachzubilden, zum Beispiel mithilfe einer for-Schleife. Mithilfe der foreach-Schleife können jedoch einige Konstellationen einfacher und damit fehlerfreier abgebildet werden. Ein besonderes Merkmal dieser Schleifenform besteht darin, dass keine separate Schleifenvariable für die Steuerung und die Prüfung der Bedingung benötigt wird. Sämtliche Informationen werden automatisch aus der zu durchlaufenden Liste entnommen. Wendet man die foreach-Schleife auf das Beispiel an, so ergibt sich folgender Quellcode (Listing 2.16):

```
string[] Laender = { "Deutschland", "Italien", "Spanien", "Portugal" };
foreach (string Land in Laender)
{
      textBox.Text = Laender[i];
}
```

Listing 2.16 Die foreach-Schleife in C#

Mit jedem Schleifendurchlauf wird unmittelbar der laufende Wert aus dem Array `Laender` ausgelesen und verarbeitet. Möglich ist jedoch nur ein lesender Zugriff, da das Feld schreibgeschützt ist. Auch ist die foreach-Schleife in der Verarbeitungsgeschwindigkeit langsamer als die for-Schleife. Diesen Umstand müssen Sie bei zeitkritischen Anwendungen bedenken. Insgesamt können wir festhalten: Die foreach-Schleife ist eine moderne Form, um alle Elemente einer Auflistung nacheinander anzusprechen. Sie kann jederzeit durch eine andere Schleifenform ersetzt werden, ihre Vorteile sind jedoch die intuitive Ausdrucksweise und die gute Lesbarkeit.

do-Schleife

Kommen wir jetzt noch zur Ausdrucksweise der fußgesteuerten Schleife in C# bzw. Java. Sie wird als *do-Schleife* bezeichnet. Wie gesagt wird die Abbruchbedingung erst am Ende der Schleife geprüft. Wenn Sie diese Schleifenform wählen, dann dürfen Sie nicht vergessen, dass sie mindestens einmal durchlaufen wird. Zur Verdeutlichung schreiben wir unser Beispiel erneut um (Listing 2.17):

```
string[] Laender = { "Deutschland", "Italien", "Spanien", "Portugal" };
int i = 0;
do
{
       textBox.Text = Laender[i];
       i++;
} while (i < 4);
```

Listing 2.17 Die do-Schleife in C#

Auch dieser Quellcode führt zu der bereits bekannten Ausgabe der einzelnen Länder.

Endlosschleifen vermeiden

Eine Schleife wiederholt einen Vorgang. Das geschieht so lange, bis eine Bedingung für ein erneutes Durchlaufen der Schleife nicht mehr erfüllt ist. Werden für die Prüfung der Abbruchbedingung numerische Werte herangezogen, ist darauf zu achten, dass dieser bestimmte Wert auch wirklich jeweils erreicht wird. Rechenergebnisse können durch den Computer durch die Kalkulation mithilfe von Näherungswerten verfälscht werden. Ein Beispiel: Die Abbruchbedingung lautet (`a==3`). Wenn *a* das Ergebnis einer mathematischen Operation ist und aufgrund von Rundungen vielleicht 2,99 lautet, so wird diese Bedingung nie erfüllt und in der Folge die Schleife endlos durchlaufen. Abhilfe kann man zum Beispiel erreichen, wenn man die Prüfung nicht gegen den exakten Wert durchführt, sondern eine bestimmte Toleranzgrenze, d. h. eine *Fehlerschranke* zulässt.

2.5.4 Objektorientierung

Im Abschnitt 2.3, »Objektorientierte Programmentwicklung«, haben wir die Prinzipien der objektorientierten Programmierung relativ ausführlich vorstellt. Nun wollen wir uns ansehen, wie man die Klassen in Quellcode umsetzt. Zunächst zur allgemeinen Syntax. In C# wird eine Klasse mit dem Schlüsselwort `class` deklariert. Also zum Beispiel (Listing 2.18):

```
class Auto
{
       // Konstruktoren
       // Attribute
       // Methoden
       // Ereignisse
}
```

Listing 2.18 Allgemeine Syntax zur Definition einer Klasse in C#

Bei der Klassendefinition muss man angeben, welche Sichtbarkeit diese aufweist. C# kennt die in Tabelle 2.4 genannten Sichtbarkeiten.

Modifizierer	Sichtbarkeit
public	Unbeschränkt, von innen und auch von außen können Objekte dieser Klasse erstellt werden.
internal	Nur innerhalb des aktuellen Projekts sichtbar. Außerhalb des Projekts ist kein Objekt dieser Klasse erstellbar. Gilt als Standard, falls keine andere Angabe erfolgt.
private	nur innerhalb von Unterklassen sichtbar

Tabelle 2.4 Sichtbarkeiten auf Klassenebene in C#

Die Schlüsselworte für die Sichtbarkeit (`public`, `internal`, `private`) bezeichnet man auch als *Modifizierer*, denn es wird damit bestimmt, welche Objekte auf das betreffende Element zugreifen dürfen. `public` verwenden Sie dann, wenn Sie beispielsweise eine Klasse definieren, auf die auch von außerhalb des Projekts zugegriffen werden kann. Dies kann beispielsweise der Fall sein, wenn Sie mehrere Projekte zu einer Projektmappe zusammenfassen und eine *Serviceklasse* erstellen, die Dienstleistungen für alle Projekte erbringt.

Auch wenn Sie eine allgemein zu verwendende Klassenbibliothek erstellen, müssen Sie für die betreffenden Klassen `public` angeben, damit Ihre Klasse von anderen Entwicklern in deren Projekten benutzt werden kann. Der Modifizierer `internal` ist der Standard in C#. Hier gleich ein Hinweis: Wenn Sie sich mit der Sichtbarkeit in ande-

ren Programmiersprachen beschäftigen, dann prüfen Sie, was dort als Standard definiert ist!

`internal` kommt also für alle Klassen zum Einsatz, auf die innerhalb eines Projekts zugegriffen werden kann. `private` spielt auf Klassenebene eine eher untergeordnete Rolle. Meist definiert man eine Klasse, um von außen Objekte dieser Klasse zu erzeugen. Das ist nicht möglich, wenn Sie die Klasse mit `private` im Zugriff einschränken. Es wird dann verwendet, wenn man Klassen ineinander schachtelt. Dieser Programmierstil ist jedoch in C# eher unüblich. Daher gilt: Eine Klasse ist in der Regel `internal` (keine Angabe notwendig), oder sie wird explizit mit `public` für von außen sichtbar gemacht.

Wenn wir gleich zu den Methoden und Attributen kommen, werden Sie sehen: Da haben wir noch ein etwas granulares System der Sichtbarkeiten. Vielleicht stellen Sie sich an dieser Stelle die Frage, warum man nicht stets `public` als Sichtbarkeit angibt und damit alle Probleme des Datenzugriffs von vornherein ausschließt. Dies würde den Prinzipien des objektorientierten Ansatzes völlig entgegenstehen. Das *Geheimnisprinzip* (siehe Abschnitt 2.3.2, »Objektorientierte Konzepte im Detail«) besagt, dass man ein Klasse von außen nur wie eine Black Box benutzt; wie diese intern funktioniert, spielt keine Rolle. Damit verbunden ist das Prinzip der *Kapselung* (siehe ebenfalls Abschnitt 2.3.2, »Objektorientierte Konzepte im Detail«), das bestimmt, dass man von außen nur über definierte Attribute und Methoden auf die Daten eines Objekts zugreifen darf. Nur wenn man das streng beachtet, kann man sicherstellen, dass keine Daten von außen auf eine nicht dafür vorgesehene Art und Weise modifiziert werden. Würde man den Zugriff auf interne Daten der Klasse zulassen, wären Fehler möglich, die dann schwer ermittelbar wären.

Nach diesem notwendigen Exkurs kehren wir zurück zu den Elementen einer Klasse in C#, also zu den Konstruktoren, Attributen, Methoden und Ereignissen. Für diese Elemente einer Klasse gelten die Angaben gemäß Tabelle 2.5 bezüglich der Sichtbarkeit.

Modifizierer	Sichtbarkeit
public	unbeschränkter Zugriff
internal	innerhalb der Klasse und der daraus abgeleiteten Klassen
internal protected	innerhalb des aktuellen Projekts oder der abgeleiteten Klassen
private	nur innerhalb der Klasse

Tabelle 2.5 Mögliche Sichtbarkeiten von Attributen, Methoden und Ereignissen

Wir gehen jetzt auf die einzelnen Elemente, d. h. auf Attribute, Konstruktoren, Methoden und Ereignisse, noch genauer ein.

Attribute

Variablen, die nur innerhalb der Klasse verwendet werden, bezeichnet man als *Attribute*. Sie sollten sie lediglich mit der Sichtbarkeit `private` deklarieren, denn der externe Zugriff auf die internen Daten eines Objekts soll nur über Properties bzw. Methoden stattfinden. Attribute werden also in der folgenden Form definiert:

```
private Datentyp Attributname;
```

Ein Beispiel:

```
private double preis;
```

Konstruktoren

Diese werden dazu verwendet, ein Objekt der betreffenden Klasse zu erstellen. C# generiert intern einen Standardkonstruktor, der nicht eigenständig implementiert werden muss. Dieser kann von außen mit `new` aufgerufen werden. Ein Objekt einer bestimmten Klasse erzeugt man wie folgt:

```
Klassenname objektname = new Klassenname();
```

Der Standardkonstruktor trägt also den Klassennamen und wird ohne weitere Parameter aufgerufen. Konstruktoren mit einem eigenen Satz von Parametern kann man alternativ definieren.

Properties

Properties werden in C# dazu genutzt, den Zugriff auf die Attribute einer Klasse von außen zu regeln. Der Zugriff kann lesend oder schreibend sein bzw. beide Varianten umfassen. Nehmen wir an, wir wollen eine Property `Fahrzeugnummer` definieren, so können wir dies wie folgt machen (Listing 2.19):

```
private int fahrzeugNummer;
public int FahrzeugNummer
{
   get
   {
          return fahrzeugNummer;
   }
   set
   {
          fahrzeugNummer = value;
   }
}
```

Listing 2.19 Definition einer Property in C#

Auch dazu braucht es ein paar Erläuterungen: Zunächst definieren wir ein Attribut `fahrzeugNummer`. Das geschieht über die folgende Syntax: `private int fahrzeugNummer`.

Es handelt sich um den Datentyp `int`, d. h. um ganze Zahlen. Durch den Zusatz `private` ist der Zugriff jedoch nur innerhalb der Klasse möglich. Um auch von außen die Fahrzeugnummer lesen und schreiben zu können, definieren wir die Property. Diese hat die Bezeichnung `FahrzeugNummer`, also identisch mit dem Attribut. Das ist das übliche Vorgehen, dass Elemente, die nur lokal zugreifbar sind, mit Kleinbuchstaben beginnen und dass von außen verfügbare Properties den gleichen Namen tragen, jedoch mit einem Großbuchstaben beginnen. Halten Sie sich an diese Konvention, damit Ihr Quelltext auch für andere Programmierer gut lesbar ist. Das Attribut `fahrzeugNummer` benötigt dann einen *Getter* und einen *Setter*, um den lesenden und schreiben Zugriff zu ermöglichen. Die komplette Implementierung für die Eigenschaft sieht dann wie in Listing 2.19 dargestellt aus.

Für eine einzelne Property ist es ein großer Schreibaufwand. Sie werden sich also zu Recht fragen, ob das Ganze auch etwas kürzer geht bzw. ob man den Aufwand bei der Implementierung verringern kann. Beides ist möglich, wie wir Ihnen jetzt kurz zeigen werden. Den Quellcode von Listing 2.19 kann man zu einer einzigen Zeile verkürzen. Er lautet dann wie folgt:

```
public int FahrzeugNummer{get; set;}
```

Was ist anders? Was fällt Ihnen auf? Zunächst fehlt die komplette Definition des Attributs `fahrzeugNummer`. Stattdessen wird gleich die Property mit den zugehörigen `set`- und `get`-Methoden definiert. Die Funktionsweise ändert sich nicht, intern erstellt der Compiler immer das Attribut. Es wird jedoch vor dem Programmierer verdeckt.

Sollte man nun nicht immer diese verkürzte Schreibweise verwenden? Das kommt drauf an. Es gibt Fälle, da benötigt man innerhalb der Klasse das Attribut für andere Zwecke, d. h., man muss diese explizit definieren. Ebenso kann man bei der Langschreibweise in den `set`- oder `get`-Methoden noch weiteren Code unterbringen, in etwa in der folgenden Form (Listing 2.20):

```
private int fahrzeugNummer;
public int FahrzeugNummer
{
   get
   {
          return fahrzeugNummer;
   }
   set
   {
          fahrzeugNummer = value;
           // hier kann weiterer Code stehen
   }
}
```

Listing 2.20 Definition einer Property mit zusätzlichem Code in der set-Methode in C#

Innerhalb der `set`-Methode ist es nur durch einen Kommentar angedeutet. Ein Beispiel: Wenn die Fahrzeugnummer eines Objekts von außen geändert wird, dann wird die `set`-Methode der Property aufgerufen. Diese könnte dann wiederum eine Aktualisierung anderer Objekte veranlassen.

Auch die Entwicklungsumgebung Visual Studio unterstützt den Programmierer umfassend und versucht, ihm diesen lästigen Schreibaufwand abzunehmen. Nehmen wir an, wir haben ein Attribut `fahrzeugNummer` vom Typ `int` innerhalb einer Klasse, also:

```
class Fahrzeug
{
        private int fahrzeugNummer;
}
```

Listing 2.21 Ausgangssituation: Eine Klasse mit einem Attribut in C#

In Visual Studio können wir nun mit der rechten Maustaste auf das Attribut `fahrzeugNummer` klicken und den Eintrag SCHNELLAKTIONEN UND REFACTORING... wählen. Daraufhin wird uns die Option angeboten, zum ausgewählten Attribut eine Property automatisch generieren zu lassen (Abbildung 2.24).

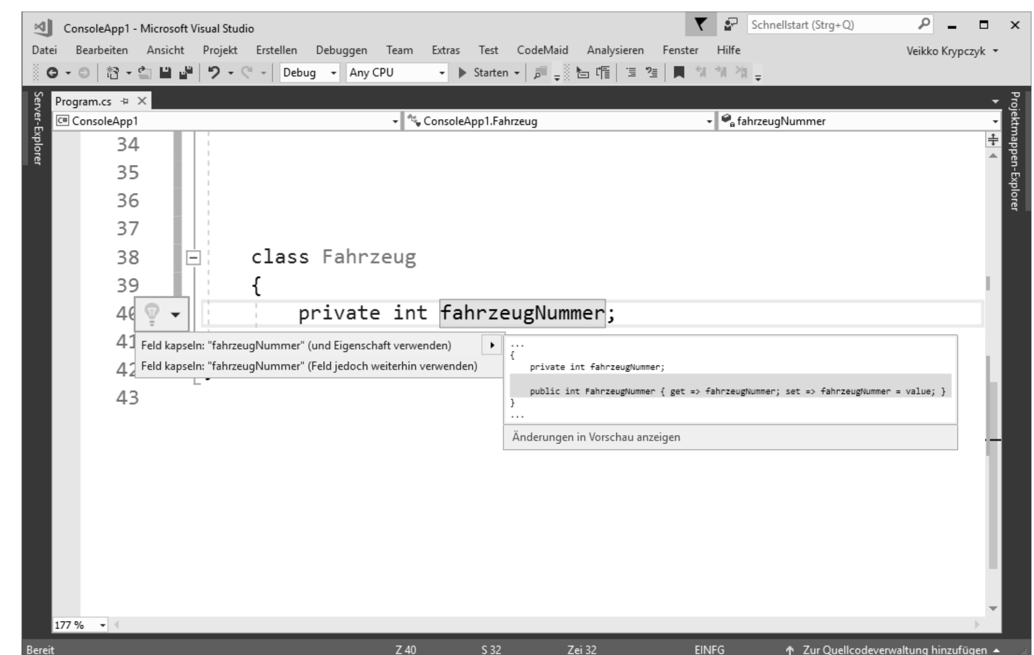

Abbildung 2.24 Automatisches Generieren einer Property in der Entwicklungsumgebung Visual Studio

Die beiden angebotenen Optionen unterscheiden sich danach, ob weiterhin das Attribut oder die neu erstellte Property benutzt werden soll. Da wir das Attribut bisher nicht benutzt haben, macht es für uns hier keinen Unterschied. Lassen wir Visual Studio für uns arbeiten, dann haben wir einen Augenblick später die automatisch generierte Property vorliegen, wie Abbildung 2.25 beweist.

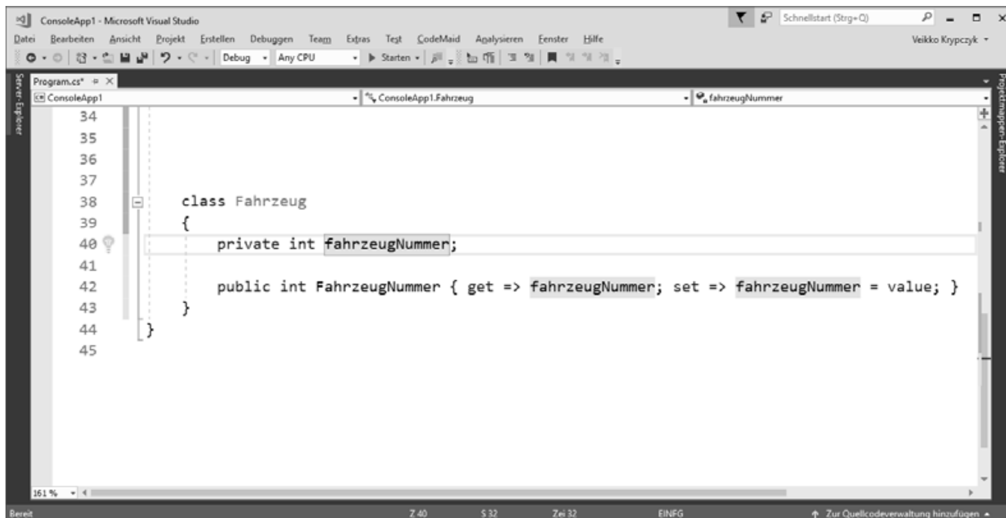

Abbildung 2.25 Das Ergebnis: Eine Property in Kurzschreibweise, automatisch erstellt durch Visual Studio

Halt! Die sieht doch ganz anders aus. Wir sehen hier den folgenden Quellcode (Listing 2.22):

```
class Fahrzeug
{
   private int fahrzeugNummer;
   public int FahrzeugNummer { get => fahrzeugNummer; set =>
             fahrzeugNummer = value; }
}
```

Listing 2.22 Eine automatisch generierte Property aus einem Attribut (Schreibweise in C# ab Version 7)

Sie haben vollkommen richtig beobachtet: Diese Schreibweise unterscheidet sich von der in Listing 2.20 dargestellten Form. Ab C# Version 7 hat sich Microsoft dazu entschlossen, bei automatisch durch Visual Studio generierten Properties auf die sogenannte Lambda-Ausdrucksweise zurückzugreifen. Das hier präsentierte Ergebnis bekommen Sie, wenn Sie C# in Version 7 bzw. Visual Studio 2017 verwenden. Setzen Sie dagegen auf die Vorversion von C# und auf Visual Studio 2015, wird der in Listing 2.20 dargestellte Quellcode generiert. Beide Varianten erzielen also die gleiche Wirkung. Der Trend geht jedoch zunehmend zu einer kompakteren Darstellungsweise, die dennoch ausdrucksstark und gut zu lesen sein soll.

Das automatische Generieren von Quellcode ist natürlich kein exklusives Feature von Visual Studio. Andere integrierte Entwicklungsumgebungen leisten für andere Programmiersprachen Ähnliches. Setter- und Getter-Methoden kann man von vielen Entwicklungsumgebungen automatisch generieren lassen. Für Sie heißt das: Machen Sie sich frühzeitig schlau, und probieren Sie aus, welche tollen Features die von Ihnen gewählte Entwicklungsumgebung bereithält, um Sie von Routinearbeiten zu entlasten!

Kommen wir jetzt zu den Methoden einer Klasse.

Methoden

Methoden haben in ihrer Grundform die Syntax, die am folgenden Beispiel deutlich wird (Listing 2.23):

```
public bool PruefeVerfuegbarkeit()
{
       // Implementierung des Prüfvorgangs.
}
```

Listing 2.23 Implementierung einer Methode in C#

Nach einem Schlüsselwort, das die Sichtbarkeit angibt (hier `public`), folgen der Rückgabetyp (hier `bool` für einen Wahrheitswert) und dann der Name der Methode. Dieser wird in der Regel in der Form Verb mit angehängten Substantiv geschrieben. Also `PruefeVerfuegbarkeit`, `LeseText`, `SchreibeDaten` usw. Auch hier gilt: Halten Sie sich an diese Konvention. Wie bei den Kommentaren empfehlen wir, auch an dieser Stelle auf englische Bezeichner zu wechseln, wenn Sie in mehrsprachigen Teams arbeiten oder den Quellcode für andere Entwickler verfügbar machen wollen. Nach dem Namen der Methode folgen in Klammern optionale Parameter, die durch die Methode verarbeitet werden. In unserem einfachen Beispiel gibt es keine Parameter.

Innerhalb einer Klasse kann man mehrere Methoden mit gleichem Namen, aber unterschiedlichen Parametern definieren. Die folgende Variante ist also möglich und üblich (Listing 2.24):

```
public bool PruefeVerfuegbarkeit()
{
       // Implementierung des Prüfvorgangs ohne Parameter (wie auch immer…)
```

```
}
public bool PruefeVerfuegbarkeit(Date Datum)
{
       // Implementierung des Prüfvorgangs mit Datum
}
```

Listing 2.24 Zwei Methoden innerhalb einer Klasse mit gleichem Namen aber unterschiedlichen Parametern

Beide Methoden lauten gleich. Die zweite Variante unterscheidet sich darin, dass ihr ein Parameter vom Typ `Date`, das Datum, übergeben wird. Beim Aufruf der Methode hat man dann die Auswahl, welche Variante man benutzen möchte.

Ereignisse

Ereignisse dienen bekanntermaßen dazu, dass sich Objekte untereinander benachrichtigen. Wir sehen uns die Funktionsweise beispielhaft in C# an. Da es bei Ereignissen um die Interaktion zwischen zwei Klassen geht, brauchen wir auch zwei davon. Wir haben sie hochkreativ `KlasseA` und `KlasseB` bezeichnet. Los geht es mit dem Quelltext:

```
class KlasseA
{
        // Einen Ereignistyp definieren
        internal delegate void EinWichtigerEreignisTyp();
        // Das Ereignis definieren
        internal event EinWichtigerEreignisTyp DasEreignis;

        private void EineMethode()
        {
            // Das Ereignis auslösen
            DasEreignis();
        }

}

class KlasseB
{
        KlasseB()
        {
            // ein Objekt der Klasse A erzeugen
            KlasseA klasseA = new KlasseA();
            // das Ereignis aus Klasse A abonnieren
            klasseA.DasEreignis += KlasseA_DasEreignis;
        }

        void KlasseA_DasEreignis()
        {
            // Hier wird angegeben, was in Klasse B passiert,
            // wenn das Ereignis in Klasse A eintritt.
        }
}
```

Listing 2.25 Definition eines Ereignisses in C#

Zuerst wird über die beiden Quellcodezeilen

```
internal delegate void EinWichtigerEreignisTyp();
internal event EinWichtigerEreignisTyp DasEreignis;
```

ein neuer Ereignistyp definiert und dann ein konkretes Ereignis davon erstellt. Das Ereignis kann später intern von der Klasse ausgelöst werden.

Das Ereignis eines Objekts der `KlasseA` kann dann von einem anderen Objekt einer anderen Klasse abonniert werden, hier von der `KlasseB`.

```
klasseA.DasEreignis += KlasseA_DasEreignis;
```

Wenn das Ereignis in `KlasseA` ausgelöst wird, bekommt `KlasseB` dies mit und kann seinerseits darauf reagieren.

Beispiel

Nun fügen wir die Erkenntnisse zu einem Beispiel zusammen. Dazu bemühen wir nochmals das oben genannte Klassendiagramm aus Abbildung 2.15 zu den Fahrzeugen. Damit Sie nicht ständig blättern müssen und mit Blick auf die Leser des E-Books, haben wir das Diagramm hier nochmals dargestellt (Abbildung 2.26).

Ebenso können wir an diesem Beispiel ein paar Merkmale der objektorientierten Programmierung darstellen, die in den letzten Abschnitten nicht erläutert wurden. Dies betrifft unter anderem die Vererbung und abstrakte Klassen. Als Programmiersprache dient erneut C#.

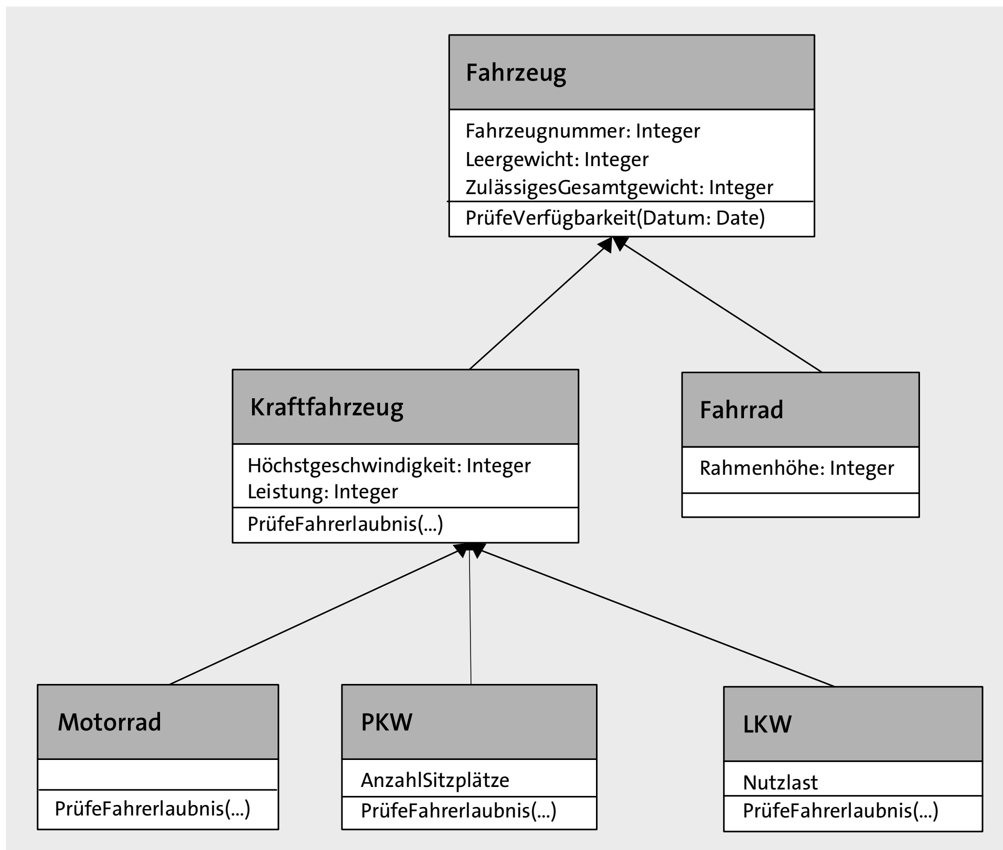

Abbildung 2.26 Das Klassendiagramm für unser objektorientiertes Beispiel

Natürlich brauchen Sie in der Praxis nicht immer zuerst ein Klassendiagramm, um mit der Umsetzung in der Programmiersprache Ihrer Wahl zu beginnen. Um Ihnen jedoch die Zusammenhänge besser verdeutlichen zu können, ist es an dieser Stelle allerdings sinnvoll. Es sind die Klassen und deren Vererbungsbeziehungen zu definieren. Die Sichtbarkeiten der gesamten Klasse und ihrer *Member* (Attribute, Methoden) sind festzulegen. Der Quellcode sieht wie folgt aus (Listing 2.26):

```
public abstract class Fahrzeug
    {
        // Attribute
        private int fahrzeugNummer;
        private double leergewicht;
        private double zulaessigesGesamtgewicht;
        // Properties
        public int FahrzeugNummer
        {
```



```
            get => fahrzeugNummer;
            set => fahrzeugNummer = value;
        }
        public double Leergewicht
        {
            get => leergewicht;
            set => leergewicht = value;
        }
        /// <summary>
        /// Liest oder schreibt das zulässige Gesamtgewicht
        /// </summary>
        public double ZulaessigesGesamtgewicht
        {
            get => zulaessigesGesamtgewicht;
            set => zulaessigesGesamtgewicht = value;
        }
        // Methoden
        public bool PruefeVerfuegbarkeit()
        {
            // Hier erfolgt die konkrete Implementierung.
            // Hilfsweise wird true zurückgegeben.
            return true;
        }
    }

    public abstract class Kraftfahrzeug : Fahrzeug
    {
        private double hoechstgeschwindigkeit;
        private double leistung;
        public double Hoechstgeschwindigkeit
        {
            get => hoechstgeschwindigkeit;
            set => hoechstgeschwindigkeit = value;
        }
        public double Leistung
        {
            get => leistung;
            set => leistung = value;
        }
        // abstrakte Methode, Platzhalter für abgeleitete Klassen
        public abstract bool PruefeFahrerlaubnis();
    }
    public class Motrorrad : Kraftfahrzeug
```

```
{
    public override bool PruefeFahrerlaubnis()
    {
        // die Implementierung, hilfsweise wird true zurückgegeben
        return true;
    }
}
public class PKW : Kraftfahrzeug
{
    private int anzahlSitzplaetze;
    public int AnzahlSitzplaetze
    {
        get => anzahlSitzplaetze;
        set => anzahlSitzplaetze = value;
    }

    public override bool PruefeFahrerlaubnis()
    {
        // die Implementierung, hilfsweise wird true zurückgegeben
        return true;
    }
}

public class LKW : Kraftfahrzeug
{
    private int nutzlast;
    public int Nutzlast
    {
        get => nutzlast;
        set => nutzlast = value;
    }
    public override bool PruefeFahrerlaubnis()
    {
        // die Implementierung, hilfsweise wird true zurückgegeben
        return true;
    }
}

public class Fahrrad : Fahrzeug
{
    private double rahmenhoehe;
    public double Rahmenhoehe
    {
```



```
            get => rahmenhoehe;
            set => rahmenhoehe = value;
        }
}
```

Listing 2.26 Quellcode in C# zum Klassendiagramm gemäß Abbildung 2.26

Wir denken, ein paar Bemerkungen sind angebracht. Also los! Schwenken Sie den Kopf zwischen Klassendiagramm (siehe Abbildung 2.26), Listing 2.26 und diesen Erläuterungen (für Zerrungen im Halswirbelbereich können wir leider keine Haftung übernehmen)! E-Book-Anwender schieben die Seiten mit den Fingern hin und her...

Mit `public abstract class Fahrzeug` wird die Definition der abstrakten Klasse `Fahrzeug` eingeleitet. Zunächst erfolgt die Definition der Attribute, zum Beispiel `private int fahrzeugNummer`. Damit man von außerhalb der Klasse auf die Daten der Objekte zugreifen kann, müssen namensgleiche Properties definiert werden. Zum Attribut `fahrzeugNummer` sieht das wie folgt aus (Listing 2.27):

```
public int FahrzeugNummer
{
    get => fahrzeugNummer;
    set => fahrzeugNummer = value;
}
```

Listing 2.27 Kurzschreibweise zur Definition Properties in C#

Methoden werden am Ende der Klasse definiert. Für die Klasse `Fahrzeug` ist es (Listing 2.28):

```
public bool PruefeVerfuegbarkeit()
{
      // Hier erfolgt die konkrete Implementierung.
      // Hilfsweise wird true zurückgegeben.
      return true;
}
```

Listing 2.28 Methodendefinition in C#

Es fehlt natürlich die Implementierung, diese spielt aber jetzt keine Rolle. Von der Klasse `Fahrzeug` leitet unter anderem die Klasse `Kraftfahrzeug` ab. Sie ist ebenfalls abstrakt, da es ein Kraftfahrzeug in allgemeiner Form auch nicht gibt. Sie dient wiederum als Oberklasse der Klassen `Motorrad`, `PKW` und `LKW`.

Die Vererbung von Klassen drückt man bei der Klassendefinition durch einen Doppelpunkt (:) und der Angabe der Oberklasse (*Basisklasse*) aus. Also konkret:

```
public abstract class Kraftfahrzeug : Fahrzeug
```

In der Klasse Fahrzeug werden weitere Attribute und die zughörigen Properties definiert. Interessant und neu ist eine sogenannte *abstrakte Methodendefinition* am Ende dieser Klasse. Sie lautet:

```
public abstract bool PruefeFahrerlaubnis();
```

Bei einer abstrakten Methodendefinition wird nur der Kopf der Methode, also die Zugriffsberechtigung, der Name, die Parameter und der Rückgabetyp notiert. Es wird ausdrücklich kein Code zur Methode hinterlegt. Damit wird festgelegt, dass abgeleitete Klassen diese Methode unbedingt implementieren müssen. Machen Sie dies nicht, meldet der Compiler einen Fehler.

Genau diese Vorgabe erfüllt die Klasse Motorrad, die von der Klasse Kraftfahrzeug ableitet:

```
public class Motrorrad : Kraftfahrzeug
```

In dieser Klasse wird die Methode neu definiert. Da sie gewissermaßen die Methode der Oberklasse überschreibt, wird bei der Methodendefinition der Zusatz override angegeben. Ebenso wird die Methode nun mit Leben gefüllt, d. h., neben dem identischen (!) Methodenkopf gibt es nun auch zwingend einen Methodenrumpf. Das ist notwendig, denn die Methode ist nun nicht mehr abstrakt. Im Beispiel wird der Inhalt der Methode nur mithilfe von Kommentaren angedeutet (Listing 2.29):

```
public override bool PruefeFahrerlaubnis()
{
        // Hier erfolgt die konkrete Implementierung.
        // Hilfsweise wird true zurückgegeben.
         return true;
}
```

Listing 2.29 Überschreiben einer Methode in C#

Die anderen Klassen weisen keine Besonderheiten bezüglich des objektorientierten Paradigmas auf. Sie leiten von der jeweiligen Oberklasse ab, erweitern diese um eigene Attribute und Methoden bzw. überschreiben letztere.

Anhand dieses Beispiels haben wir Ihnen weitere wichtige Konzepte der Programmierung, insbesondere des objektorientierten Vorgehens vorgestellt. Es waren aber wirklich nur die Basics. Heutige Programmiersprachen bieten noch mehr, um Softwaremodelle auf einem hohen Abstraktionsniveau zu bauen. Wir haben uns lediglich meist auf die Sprache C# beschränkt, andere Sprachen bieten, wie gesagt, eine ähnliche Palette an Features. Die Stärken, Schwächen und die dahinterliegende Philosophie sind jedoch stets etwas unterschiedlich.

Ebenso gilt: Diese Ausführungen können natürlich nicht das systematische Studium der Dokumentation einer Programmiersprache ersetzen, aber das grundsätzliche Vorgehen sollte deutlich geworden sein – und darum ging uns hier!

2.6 Fazit und Ausblick

In diesem Kapitel haben wir Ihnen die Grundlagen der Programmentwicklung, und zwar deren Kern, die Programmierung, vorgestellt. Sie wissen nun, was man unter einem Computerprogramm versteht, und haben einen groben Überblick über die Entwicklung der Programmiersprachen und über deren systematische Einordnung. Ebenso haben wir Ihnen die objektorientierte Programmierung anhand der wesentlichen Eckpfeiler dieses Vorgehens erläutert. Sie haben wichtige Sprachmerkmale am Beispiel der Programmiersprache C# (und gelegentlich auch der Programmiersprache Java) kennengelernt. Ebenso hoffen wir, dass Sie das abschließende Beispiel der objektorientierten Programmierung intensiv durchgearbeitet und am Rechner nachvollzogen haben.

Mit diesem Wissen sind Sie bestens gerüstet, um weiter in die Geheimnisse der professionellen Programmierung einzusteigen. Mit dem folgenden Kapitel bekommen Sie weiteres wichtiges Rüstzeug an die Hand. Wir behandeln die Themen Algorithmen und Datenstrukturen, die essenziell für das Erstellen eines jeden Computerprogramms sind. Auch diese Themen werden wir – wie übrigens die meisten Inhalte im Buch – weitgehend sprachneutral vorstellen. Die Kenntnis der grundsätzlichen Vorgehensweise und der notwendige Überblick sind uns an dieser Stelle wichtiger als die später zweifelsfrei notwendige Detailkenntnis der Syntax einer Programmiersprache. Sie haben Recht, dass Sie sich irgendwann für das Erlernen einer Sprache entscheiden müssen. Dies wird Ihnen jedoch dann leichter gelingen, wenn Sie nach der Lektüre der Kapitel 2, »Programmierung als Kern der Softwareentwicklung«, und Kapitel 3, »Algorithmen und Datenstrukturen«, dieses Fachbuchs den notwendigen Überblick bekommen haben.

Um sich mit der Syntax und Vorgehensweise der von Ihnen ausgewählten Sprache zu beschäftigen und diese zu erlernen, gibt es unterschiedliche Ansätze. Neben unzähligen Informationen im Internet erlauben wir uns hier etwas Werbung für den Verlag. Dieser hat zu nahezu allen aktuellen Sprachen Fachbücher für den Einstieg im Programm. Auf jeden Fall empfehlen wir Ihnen ein *Learning by Doing*: Suchen Sie sich rechtzeitig ein konkretes Projekt, um Vorgehensweisen und Konzepte auszuprobieren. Dieses sollte nicht allzu kompliziert sein, aber auch genug Ansätze zur Erprobung bieten. An Ideen wird es Ihnen nicht mangeln, denn wir Softwareentwickler sind kreativ und haben in der Regel mehr auf der ToDo- bzw. Ideenliste, als wir jeweils in einem langen Entwicklerleben umsetzen könnten. Jetzt bleiben Sie aber erst ein-

mal bei diesem Buch und folgen uns auf die äußert interessante Reise durch die Welt der Algorithmen.

2.7 Literatur und Links

Balzert, Helmut: Lehrbuch Grundlagen der Informatik, Spektrum Akademischer Verlag, 2005.

Doberenz, Walter; Gewinnus, Thomas; Saumweber, Walter: Visual C# 2017, Carl Hanser Verlag, 2017.

Habelitz, Hans-Peter: Programmieren lernen mit Java, Rheinwerk Verlag, 2016.

Henning, Peter A. (Herausgeber); Vogelsang, Holger (Herausgeber): Taschenbuch Programmiersprachen, Carl Hanser Verlag, 2007.

Kecher, Christoph; Salvanos, Alexander: UML 2.5: Das umfassende Handbuch, Rheinwerk Verlag, 2015.

Ratz, Dietmar; Scheffler, Jens; Seese, Detlef; Wiesenberger, Jan: Grundkurs Programmieren in Java, Carl Hanser Verlag, 2014.

Theis, Thomas: Einstieg in C# mit Visual Studio 2017, Rheinwerk Verlag, 2017.

Zuse, Horst: Geschichte der Programmiersprachen, Technische Universität Berlin, Fachbereich Informatik, verfügbar online unter: *http://www.horst-zuse.homepage.t-online.de/HNF-PNN.pdf*.

http://www.horst-zuse.homepage.t-online.de/kz-bio.html