IN DIESEM KAPITEL

Grundlagen der Programmierung

.NET als Softwareentwicklungsplattform

Einordnung der Programmiersprache C#

Kapitel 1
Grundlagen und Einführung

Sie haben sich dieses Buch zum Erlernen der Programmiersprache C# gekauft. In diesem Buch werden keinerlei Programmierkenntnisse vorausgesetzt, daher möchte ich Ihnen in diesem ersten Kapitel grundlegende Konzepte der Programmierung nahebringen. Einige dieser Begriffe und Konzepte sind für das spätere Verständnis der Programmiersprache C# sicherlich hilfreich. Beim Erlernen des Programmierens – ganz gleich in welcher Sprache – ist es wichtig, ein paar grundlegende Begriffe und Zusammenhänge verstanden zu haben, um tiefer in die Materie einzusteigen. Sollten Sie schon Programmiererfahrung haben, können Sie dieses Einstiegskapitel auch überspringen. Dieses Einstiegskapitel ist aber bewusst kurz gehalten, damit wir schnell mit dem eigentlichen Thema des Buches starten können, nämlich der Programmiersprache C#.

Grundlagen der Programmierung

Ganz ohne Theorie geht nichts! In diesem leider etwas »trockenen« Abschnitt lernen Sie ein paar Grundlagen zur Programmierung. Bevor Sie mit den ersten Programmzeilen beginnen, sollten Sie ein gewisses Grundverständnis dafür haben, was Programmierung überhaupt ist. Dabei möchte ich auf die folgenden Fragestellungen eingehen:

- ✔ Warum programmieren wir eigentlich?
- ✔ Welche Rolle übernehmen die Programmiersprachen hierbei?
- ✔ Was ist .NET?
- ✔ Wo lässt sich die Programmiersprache C# einordnen?

Warum programmieren wir eigentlich?

In erster Linie geht es bei der Programmierung darum, einen Computer dazu zu bringen, bestimmte Aufgaben zu übernehmen. Denken Sie dabei einfach mal an Ihren Alltag und überlegen Sie, wo Sie überall einen Computer oder eine Software einsetzen, um eine bestimmte Aufgabe zu erledigen. Das kann beispielsweise ein E-Mail-Programm, eine Bildbearbeitungssoftware oder die Smartphone-App sein, mit der Sie Textnachrichten versenden. All diese Programme hat irgendjemand programmiert, um dem Benutzer bei einer bestimmten Aufgabenstellung zu helfen.

Sie als angehender Entwickler haben bei der Programmierung nun die Aufgabe, dem Computer *Anweisungen* (auch Befehle genannt) zu geben. Sie sagen dem Computer also, was er genau tun soll, beispielsweise eine E-Mail versenden oder eine Webseite aufrufen. Diese Befehle werden dann vom Computer Schritt für Schritt ausgeführt. Das können Sie mit einem Kochrezept vergleichen. In einem Rezept wird vorgegeben, welche Schritte in welcher Reihenfolge ausgeführt werden müssen, damit am Ende beispielsweise ein Kuchen daraus wird. Im Grunde geben Sie dem Computer die Schritte vor, die er dann ausführen soll, um eine bestimmte Aufgabe zu erfüllen.

Die Arbeitsschritte zur Lösung einer bestimmten Problemstellung (also das Rezept) werden in der Informatik als *Algorithmus* bezeichnet. Bei den Programmen (auch Software oder Anwendung genannt), die Sie zukünftig programmieren werden, handelt es sich daher um eine Folge von Algorithmen. Die Algorithmen geben die Anweisungen vor, die vom Computer ausgeführt werden sollen.

Die zur Lösung einer Problemstellung definierten Arbeitsschritte nennt man in der Softwareentwicklung *Algorithmus*. Vergleichen lässt sich das in etwa mit Kochrezepten, Betriebsanleitungen oder Schritt-Für-Schritt-Anleitungen. Ein Algorithmus ist, so gesehen, eine Anleitung für den Computer, wie er ein bestimmtes Problem oder eine vorgegebene Aufgabenstellung zu lösen hat.

Ein *Programm* setzt sich aus vielen verschiedenen Algorithmen zusammen, um eine bestimmte Aufgabe zu lösen.

Als Programmieranfänger tut man sich oft sehr schwer, die Arbeitsschritte eines Algorithmus zu definieren. Beim Kochen eines Gerichts wird sehr häufig variiert oder improvisiert. Ein Computer hingegen ist sehr penibel, was die Ausführung der ihm vorgegebenen Schritte angeht. Diese werden genauso ausgeführt, wie sie im Algorithmus vorgegeben wurden.

Gerade am Anfang stellen Sie sich vielleicht die Frage: Wie lernt man Programmieren? Nach all den Jahren kann ich Ihnen sagen: Man lernt es nur, wenn man selbst programmiert. Getreu dem Motto: Learning by doing! Sie sollten

immer wieder selbst Programme erstellen oder bereits bestehende Programme nochmals überdenken und gegebenenfalls optimieren. Sie sollten sich auch nicht scheuen, mal den einen oder anderen Blick in ein Open-Source-Projekt zu werfen, auch hier kann man sehr viel lernen (dazu später mehr). Gerade am Anfang wird nicht alles direkt auf Anhieb funktionieren, aber das geht jedem Entwickler so. Das gehört zum Lernprozess.

Ein *Open-Source-Projekt* ist ein Programm, dessen Quelltext veröffentlicht wurde und jedem frei zugänglich ist. Es kann von jedem eingesehen und unter bestimmten Bedingungen auch verändert werden.

Im Verlauf des Buchs werde ich Ihnen einige interessante Open-Source-Projekte vorstellen und Ihnen auch zeigen, wie Sie solche Projekte in Ihren eigenen Programmen einsetzen können.

Programmiersprachen

Sie haben bestimmt schon einmal den folgenden Satz gehört: Computer arbeiten mit Nullen und Einsen. Hintergrund dafür ist die technische Ebene eines Computers in Form des Prozessors, der eben nur zwei Zustände versteht: null für »Strom aus« und eins für »Strom an«. Sie kennen eine solche Verarbeitung etwa aus dem Morsealphabet. Mit einem solchen Morse-*Code* ist es möglich, Buchstaben und Zahlen zu übermitteln, dabei wird ein Signal unterschiedlich lange ein- und dann wieder ausgeschaltet. Der Morsecode besteht aus drei Signalen oder Symbolen: kurzes Signal, langes Signal und Pause. Ein Computer hingegen kennt, wie bereits erwähnt, nur zwei Zustände. Diesen Code, in dem Informationen durch zwei unterschiedliche Zustände dargestellt werden, nennt man auch *Binärcode*. Binärcode ist von einem Computer direkt ausführbar und sorgt dafür, dass er auch die Dinge tut, die Sie ihm in Form eines Algorithmus vorgeben.

Als *Hardware* bezeichnet man alles rund um einen Computer, das man anfassen kann, also seine physischen Komponenten wie beispielsweise die Festplatte oder der Prozessor eines Computers.

Sie können sich jetzt sicher vorstellen, dass es für einen Entwickler außerordentlich kompliziert wäre, die Anweisungen an den Computer (also den Algorithmus) in Nullen und Einsen zu formulieren. Stellen Sie sich einmal vor, Sie hätten den folgenden Programmcode

```
001001100100011010111101
```

Das kann ein normaler Menschen weder lesen noch lässt sich erkennen, was diese Folge von Nullen und Einsen bewirken soll. Hier kommen dann die Programmiersprachen ins Spiel. Programmiersprachen helfen, die Interaktion mit dem Computer zu vereinfachen. Sie »verschonen« einen Entwickler also damit, beim Programmieren mit Nullen und Einsen hantieren zu müssen.

Die Anweisungen, die wir dem Computer in Form einer Programmiersprache geben, werden als Text formuliert und in ganz gewöhnlichen Textdateien gespeichert. Diese Dateien werden als Quelldateien (engl.: source files) bezeichnet und der Inhalt einer solchen Datei heißt Quelltext oder Quellcode (engl.: source code). Die Gesamtheit des Quelltexts wird unter dem Begriff Programm zusammengefasst (wobei ein Programm aus einer oder mehreren Quellcode-Dateien bestehen kann).

Der Quellcode wird nach genau festgelegten Regeln formuliert. Diese Regeln können je nach gewählter Programmiersprache variieren und sind durch die Grammatik der gewählten Programmiersprache festgelegt. Anders als natürliche Sprachen sind Programmiersprachen wenig flexibel. Die festgelegten Grammatikregeln müssen unbedingt eingehalten werden. Jedes Zeichen, wie beispielsweise ein Punkt oder Komma, hat seine Bedeutung. Selbst ein kleiner Fehler führt dazu, dass das komplette Programm nicht ausgeführt werden kann. In C# gibt es beispielsweise eine Regel, dass jede Anweisung mit einem Semikolon beendet werden muss. Das hört sich jetzt schlimmer an, als es in Wirklichkeit ist. Moderne Entwicklungsumgebungen erkennen Syntaxfehler schon während der Entwicklung und geben einem Entwickler entsprechende Hinweise, dass ein Fehler vorliegt. Manche Entwicklungsumgebungen sind sogar in der Lage, solche Fehler automatisch zu korrigieren.

In Kapitel 2 *Entwicklungswerkzeuge und Tools* werde ich Ihnen eine Entwicklungsumgebung vorstellen, die viele der beschriebenen Funktionen bereitstellt.

Eine Quellcode-Datei lässt sich leider nicht ohne Weiteres auf dem Rechner ausführen. Damit der Computer den Quelltext ausführen kann, den Sie geschrieben haben, muss der Quelltext in ein Format übersetzt werden, das der Computer auch versteht (also Nullen und Einsen). Dieses Format wird auch als *Maschinencode* oder *Maschinenbefehle* bezeichnet.

Alles was ein Entwickler in einer bestimmten Programmiersprache formuliert, wird als *Quellcode* bezeichnet. Bei *Maschinencode* handelt es sich um Code, der vom Prozessor eines Computers gelesen und ausgeführt werden kann.

Sie müssen sich natürlich nicht selbst um die Übersetzung von Quellcode in Maschinencode kümmern, denn dann bräuchten wir ja keine Programmiersprache. Dazu gibt es im Wesentlichen zwei verschiedene Wege:

- ✔ zum einen gibt es *Compiler* und
- ✔ zum anderen sogenannte *Interpreter*

Es gibt eine Vielzahl von Programmiersprachen. Manche Programmiersprachen verwenden einen Compiler und werden daher auch als *kompilierte* Programmiersprachen bezeichnet. Daneben gibt es sogenannte *interpretierte* Programmiersprachen, die einen Interpreter benötigen. Es gibt aber noch eine dritte Variante: Programmiersprachen, die beides benötigen, sogenannte *Zwischensprachen*. Bevor wir uns anschauen, was ein Compiler und was ein Interpreter ist und worin sich die genannten Sprachtypen unterscheiden, kann ich einen Punkt schon einmal vorwegnehmen (Achtung: Spoiler!): Bei der Programmiersprache C# handelt es sich um eine Zwischensprache, die sowohl Compiler als auch Interpreter benötigt.

Kompilierte Programmiersprachen

Bei einer kompilierten Programmiersprache wird der Quellcode mithilfe eines sogenannten Compilers in Maschinencode kompiliert. Bei diesem Vorgang spricht man auch oft von »übersetzen«. Wenn Sie das Betriebssystem Windows von Microsoft einsetzen und dort ein Programm starten (beispielsweise ein Programm für die Bildbearbeitung) führen Sie eine sogenannte *.exe-Datei aus. Eine *.exe-Datei enthält den ausführbaren Maschinencode und ist das Ergebnis eines Kompiliervorgangs. Die Programme, die durch einen Compiler erzeugt wurden, können dann direkt ohne weitere Hilfskomponenten oder sonstige Laufzeitumgebungen auf dem Betriebssystem, für das sie kompiliert wurden, ausgeführt werden. In Abbildung 1.1 ist der Vorgang des Kompilierens dargestellt.

Abbildung 1.1: Ein Compiler übersetzt Quelltext in ausführbaren Maschinencode (*.exe-Datei).

Ein *Compiler* ist ein Computerprogramm, das Quellcode einer bestimmten Programmiersprache in ausführbaren Maschinencode übersetzt, der dann von einem Computer mit einem bestimmten Betriebssystem gelesen und ausgeführt werden kann.

Als Beispiele für kompilierte Programmiersprachen können C oder auch C++ genannt werden. Damit ein in C++ geschriebenes Programm ausgeführt werden kann, muss es vorher über einen entsprechenden C++-Compiler in ausführbaren Maschinencode übersetzt werden. Jetzt ist es leider so, dass für die unterschiedlichen Betriebssysteme jeweils ein eigener C++-Compiler benötigt wird. Das bedeutet, dass für jedes Betriebssystem auch eine eigene Version des Programms kompiliert werden muss. Um das Programm also unter Windows laufen zu lassen, muss es zuvor unter Windows mit einem entsprechenden Compiler kompiliert werden. Dies gilt analog für Linux, macOS und jedes andere Betriebssystem. Das Kompilieren für die unterschiedlichen Betriebssysteme ist mit recht viel Aufwand verbunden und auch gleichzeitig einer der größten Nachteile von kompilierten Programmiersprachen.

Programme, die für ein bestimmtes Betriebssystem kompiliert wurden, nennt man auch *native Programme*.

Trotz des ganzen Aufwands haben kompilierte Programme einen entscheidenden Vorteil: Der erzeugte Maschinencode kann direkt, ohne weitere Hilfsprogramme und ohne Laufzeitumgebung, auf dem jeweiligen Betriebssystem ausgeführt werden. Auf dem Zielbetriebssystem muss keine weitere Software installiert werden, um das kompilierte Programm ausführen zu können. Das ist ein großer Unterschied zu interpretierten Programmiersprachen, die ich Ihnen jetzt kurz vorstelle.

Interpretierte Programmiersprachen

Bei interpretierten Programmiersprachen ist es nicht notwendig, den Quellcode für ein bestimmtes Betriebssystem zu kompilieren. Der Quellcode wird hier nicht mit einem Compiler übersetzt, sondern durch einen Interpreter (zur Laufzeit des Programms) eingelesen, analysiert und ausgeführt. Zwingende Voraussetzung ist hier, dass der Interpreter auf dem Zielrechner installiert sein muss, auf dem das Programm ausgeführt werden soll. Der Interpreter muss natürlich für das jeweilige Betriebssystem zur Verfügung stehen. Im Gegensatz zum Compiler, der nur auf dem Computer installiert sein muss, auf dem der Quellcode kompiliert wird (im Normfallfall ist das der Entwicklungsrechner). Abbildung 1.2 verdeutlicht diesen Vorgang.

Abbildung 1.2: Ein Interpreter wertet den Quelltext direkt aus und wandelt ihn in Maschinencode um.

Ein *Interpreter* ist ein Computerprogramm, das Quellcode im Gegensatz zu einem Compiler nicht in eine auf dem System direkt ausführbare Datei übersetzt, sondern den Quellcode einliest, analysiert und direkt zeilenweise ausführt. Die Übersetzung des Quellcodes erfolgt also erst zur Laufzeit des Programms. Der *Interpreter* ist dabei Bestandteil des Betriebssystems oder muss dort nachträglich installiert werden.

Interpretierte Programmiersprachen erfreuen sich großer Beliebtheit, da kein Compiler notwendig ist und der zusätzliche Kompiliervorgang entfällt. Einer der bekanntesten Vertreter von interpretierten Programmiersprachen ist JavaScript. Nachteil interpretierter Sprachen ist allerdings, dass diese Programme tendenziell weniger performant sind als kompilierte Programme, da die Umwandlung des Quellcodes zur Laufzeit des Programms stattfindet (diese Umwandlung zur Laufzeit entfällt bei kompilierten Programmen).

Unter dem Begriff Performance versteht man das Leistungsverhalten von Hard- und Software. In der Softwareentwicklung verwendet man den Begriff Performance oft im Sinne von »wie schnell« eine bestimmte Aufgabe ausgeführt werden kann.

Ein weiterer Nachteil interpretierter Sprachen ist, dass Syntaxfehler erst zur Laufzeit erkannt werden und somit beim Programmieren schnell übersehen werden können. Bei kompilierten Sprachen hingegen werden Syntaxfehler direkt beim Kompilieren erkannt, die erkannten Fehler müssen dann zwingend für einen erfolgreichen Kompilierungsvorgang behoben werden.

Zwischensprachen

Es gibt einige Programmiersprachen, die sowohl Compiler als auch Intertreter benötigen. Da man diese Programmiersprachen nicht eindeutig einer der beiden Kategorien zuordnen kann, werden diese Sprachen auch als Zwischensprachen bezeichnet. Wie ich bereits vorweggenommen habe, ist C# (wie auch Java) eine solche Zwischensprache. In Abbildung 1.3 sehen Sie schematisch, wie eine solche Zwischensprache ausgeführt wird.

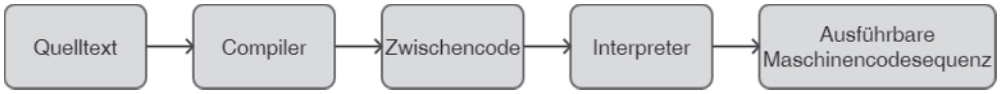

Abbildung 1.3: Es gibt auch Programmiersprachen, die eine Kombination aus Compiler und Interpreter verwenden.

In C# ist es so, dass der Quelltext durch einen Compiler in sogenannten *CIL*-Code (*Common Intermediate Language*) kompiliert wird. Der CIL-Code ist eine Art Zwischencode, der seinerseits einen Interpreter benötigt, um ausgeführt werden zu können. Dazu erfahren Sie gleich mehr! Der Vorteil: C#-Anwendungen müssen nicht (wie bei rein kompilierten Sprachen) auf dem gleichen Betriebssystem kompiliert werden, auf dem sie später ausgeführt werden, da der *CIL*-Code grundsätzlich unabhängig vom Betriebssystem und der verwendeten Programmiersprache ist. Das Einzige, was auf dem jeweiligen Zielbetriebssystem vorhanden sein muss, ist ein Interpreter für diesen CIL-Code, mit anderen Worten: eine Laufzeitumgebung. Für C# heißt diese Laufzeitumgebung .NET-Framework, sie ist Bestandteil der .NET-Plattform. Im nächsten Kapitel möchte ich Sie mit der .NET-Philosophie und den damit verbundenen Konzepten, Begriffen und Features vertraut machen.

.NET-Plattformarchitektur

Als Autor bin ich jetzt etwas in der Zwickmühle: Eigentlich finde ich Bücher langweilig, die sich zu Beginn lang und breit mit einer Technologie befassen und in denen ein Fachbegriff nach dem anderen eingeführt wird. Beim Schreiben musste ich jedoch feststellen, dass das leider unumgänglich ist. Um die Programmiersprache C# zu verstehen, müssen wir uns zunächst die Plattform .NET anschauen, damit Sie überhaupt mit C# programmieren können.

Daher lernen Sie in diesem Abschnitt einige elementare Grundlagen von .NET. Zwangsläufig werden hier ein paar Begriffe fallen, die Ihnen möglicherweise zu diesem Zeitpunkt nicht sehr viel sagen. Ich werde die theoretischen Grundlagen aber auf ein Minimum reduzieren und mich auf das beschränken, was für einen Einstieg in die Programmiersprache C# absolut notwendig ist. Es werden vermutlich einige Begriffe fallen, die Sie nicht sofort einordnen können. Das wird sich aber im weiteren Verlauf des Buches alles aufklären.

Ganz allgemein gesprochen ist .NET (gesprochen Dotnet) eine Plattform der Firma Microsoft für die Entwicklung von Software, die im Jahre 2002 offiziell vorgestellt wurde. Mit der Einführung der .NET-Technologie wurde auch eine Vielzahl neuer Begriffe eingeführt und es kommen ständig neue hinzu. Ich möchte Ihnen hier lediglich einen groben Überblick über die wichtigsten Begriffe und Konzepte geben, denn alles andere würde den Rahmen dieses Buchs sprengen. Zum Lernen der Programmiersprache C# müssen Sie auch nicht alle Begriffe und Konzepte kennen, es geht hier lediglich um ein Grundverständnis der .NET-Plattform.

.NET wurde als moderne und betriebssystemunabhängige Softwareentwicklungsplattform geschaffen, die es ermöglicht, Anwendungen für unterschiedliche Plattformen zu entwickeln. Dabei eignet sich die .NET-Plattform zum Erstellen von Applikationen für das Web, für Microsoft Windows, für Smartphones und für andere Betriebssysteme.

Bedingt durch die lange Historie von .NET gibt es allerdings einen gravierenden Nachteil: Im Laufe der Zeit ist eine schier unendliche Menge an Klassen entstanden, die fast unüberschaubar geworden ist. Microsoft hat dieses Problem erkannt und treibt die Umstrukturierung der .NET-Plattform voran. Darauf möchte ich aber nicht weiter eingehen. Wichtig für Sie: Gerade am Anfang wird es schwer sein, ich zu orientieren. Geben Sie nicht zu schnell auf, es lohnt sich. Es gibt eine sehr gute Dokumentation und wenn der Einstieg einmal geschafft ist, kommt der Rest von ganz allein. Selbst die erfahrensten Entwickler kennen wahrscheinlich nicht alle Klassen, die die .NET-Plattform zu bieten hat.

Doch was ist jetzt so toll an .NET und was bringt es dem Entwickler?

- ✔ **Objektorientierung** – In .NET arbeiten Sie durchweg mit Klassen und Objekten. Was das genau bedeutet, lernen Sie in einem späteren Kapitel. Sie haben jedenfalls eine moderne Grundlage für die Entwicklung Ihrer Anwendungen.
- ✔ **Plattformunabhängigkeit** – .NET-Anwendungen sind grundsätzlich plattformunabhängig. Für Sie hat das den folgenden Vorteil: Sie programmieren eine Anwendung und diese lässt sich dann auf Microsoft Windows, Linux oder macOS ausführen.
- ✔ **Sprachunabhängigkeit** – .NET ist grundsätzlich sprachunabhängig und unterstützt mehrere Programmiersprachen. Dies bedeutet, dass Sie als Entwickler in einer von vielen Sprachen für die .NET-Plattform entwickeln können.
- ✔ **Speicherverwaltung** – In einigen Programmiersprachen müssen Sie sich als Entwickler explizit um die Freigabe von nicht mehr benötigtem Speicher kümmern. Das ist im .NET-Umfeld nicht mehr so, dort gibt es den sogenannten *Gargabe Collector*. Der Garbage Collector erkennt automatisch nicht mehr benötigte Objekte und entfernt diese aus dem Speicher.

In Abbildung 1.4 ist das alles noch einmal zusammengefasst.

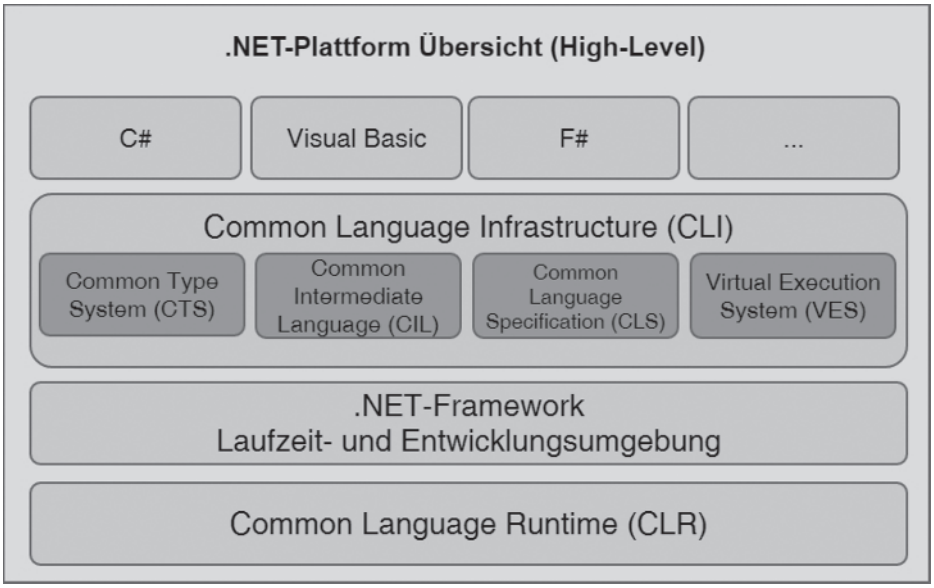

Abbildung 1.4: Übersicht .NET-Plattform (High-Level).

Sollten Sie sich weiter in das Thema .NET-Plattform einarbeiten wollen, legen ich Ihnen das Portal von Microsoft ans Herz:

`https://www.microsoft.com/net/`

Dort erhalten Sie weiterführende Informationen zu verschiedenen .NET-Themen und den unterschiedlichen Möglichkeiten, die die .NET-Plattform bietet.

In den nächsten Abschnitten lernen Sie die einzelnen Bestandteile noch etwas näher kennen und insbesondere, wie eine .NET-Sprache grundsätzlich funktioniert.

Wie funktioniert eine .NET-Sprache?

Die .NET-Plattform unterstützt unterschiedliche Programmiersprachen, beispielsweise C# und Visual Basic. Egal in welcher .NET-Sprache Sie ein Programm schreiben, der Quellcode wird immer in ein und dieselbe Zwischensprache übersetzt, die sprachunabhängig ist. Sie können sich bestimmt vorstellen, dass es bei der Übersetzung in diese Zwischensprache einiges zu beachten gibt. Dieses Regelwerk ist in der *Common Language Specification* (CLS) festgehalten. Der Compiler einer .NET-Sprache erzeugt also keinen prozessorspezifischen Maschinencode, sondern einen von der Plattform und der verwendeten Hardware

unabhängigen Zwischencode. Dieser Zwischencode wird auch *CIL-Code* (*Common Intermediate Language*) genannt. Die *Common Intermediate Language* (*CIL*) ist in frühen Entwicklungsphasen von .NET als Microsoft Intermediate Language (MSIL) bezeichnet worden, diese Bezeichnung wurde aber mit fortschreitender Standardisierung verworfen.

Wird ein .NET-Programm ausgeführt, wird der CIL-Code in den prozessorspezifischen ausführbaren Maschinencode umgewandelt. Die Umwandlung übernimmt der sogenannte *Just-In-Time-Compiler* (JIT-Compiler), der den CIL-Code analysiert und schlussendlich ausführt. Je nach verwendetem Betriebssystem und eingesetztem Prozessor kommen verschiedene Optimierungen zum Einsatz. Da der Just-In-Time-Compiler sehr schnell arbeitet, ist der Performanceverlust im Vergleich zu kompilierten Sprachen kaum der Rede wert. Abbildung 1.5 zeigt den Kompiliervorgang eines .NET-Programms.

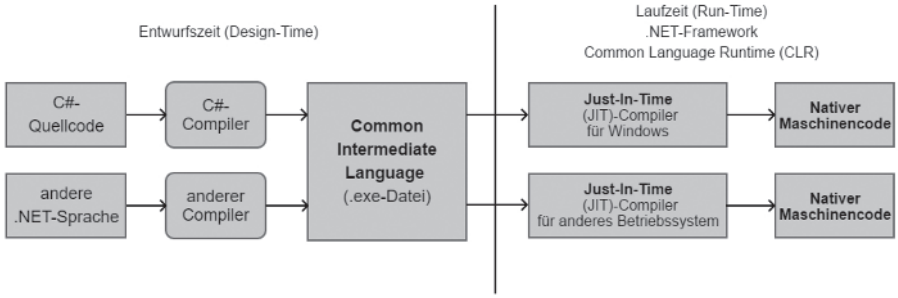

Abbildung 1.5: Kompilierung eines .NET-Programms.

Der Quellcode wird immer in dieselbe Common Intermediate Language (CIL) übersetzt. Dabei ist es egal, welche .NET-Programmiersprache Sie verwenden, beispielsweise C# oder VB.NET. Der CIL-Code, der entsteht, ist grundsätzlich plattformunabhängig und kann daher auf jedem Betriebssystem ausgeführt werden, unter welchem ein .NET-Framework installiert werden kann. Damit spart man sich die Übersetzung des Quellcodes für unterschiedliche Betriebssysteme, wie man das bei kompilierten Sprachen tun muss.

Nach der ganzen grauen Theorie schauen wir uns jetzt endlich ein erstes C#-Beispielprogramm an, das in Listing 1.1 zu sehen ist.

```
1  using System;                // Importieren von Namespaces
2
3  class Test                   // Klassendeklaration
4  {
5    static void Main()         // Methodendeklaration
6    {
```

```
 7      int x = 12 * 30;        // Anweisung/Befehl 1
 8      Console.WriteLine(x);   // Anweisung/Befehl 2
 9      Console.ReadKey();      // Anweisung/Befehl 3
10    }                         // Ende der Methode
11  }                           // Ende der Klasse
```

Listing 1.1: Beispielprogramm in C# .

Auch ohne große Vorkenntnisse in einer Programmiersprache kann man schon ungefähr erahnen, was das oben gezeigte Programm tun wird, oder? Schauen wir uns das Programm im Detail an: In Zeile 1 wird ein sogenannter Namensraum (engl. namespace) eingebunden, was das genau bedeutet, werde ich ihnen später noch genauer erklären. Für die eigentliche Funktionalität des gezeigten Programms ist das in Moment noch nicht von Interesse. Bei C# handelt es sich, wie schon erwähnt, um eine objektorientierte Sprache und aus diesem Grund wird in Zeile 3 eine Klasse definiert. Die Funktionen einer Klasse werden in sogenannten Methoden bereitgestellt, wie in Zeile 5 zu sehen. Das sind die Minimalanforderungen an ein C#-Programm. Die erste Anweisung (Zeile 7) multipliziert die Zahlen 12 und 30 und speichert das Ergebnis in einer sogenannten Variable namens `x`. Der Typ der Variable ist `integer`, was das genau bedeutet, lernen Sie im weiteren Verlauf des Buchs. Die zweite Anweisung (Zeile 8) gibt das Ergebnis dann auf dem Bildschirm aus und die Anweisung `Console.ReadKey()` in Zeile 9 sorgt dafür, dass gewartet wird, bis der Benutzer eine beliebige Taste drückt. Dadurch wird das Programm nicht automatisch beendet. Ansonsten würde das Fenster nur kurz aufflackern und man hätte keine Möglichkeit, die Bildschirmausgabe zu betrachten. Eigentlich gar nicht so schwer, oder?

In Listing 1.1 können Sie auch schon ein paar Syntaxregeln der Programmiersprache C# erkennen: Beispielsweise muss jede öffnende Klammer mit einer schließenden Klammer geschlossen und jede Anweisung mit einem Semikolon beendet werden. Wenn Sie das Programm kompilieren erhalten Sie als Output eine *.exe-Datei. Diese Datei enthält den entsprechenden CIL-Code, der vom Just-In-Time-Compiler ausgeführt werden kann. Der CIL-Code für das Programm aus Listing 1.1 ist in Abbildung 1.6 dargestellt.

Der CIL-Code ist nicht mehr so einfach zu lesen wie der zugehörige C#-Code des Beispielprogramms. Zunächst sehen Sie verschiedene Informationen, die das .NET-Framework für die Ausführung des Programms benötigt. Es sind aber auch die Funktionen wie `System.Console::WriteLine` und `System.Console::ReadKey` zu sehen. Der CIL-Code wird bei der Ausführung der *.exe-Datei vom JIT-Compiler in ausführbaren Maschinencode übersetzt. Der C#-Code orientiert sich eher an der natürlichen Sprache und lässt sich intuitiv lesen, wohingegen der CIL-Code für einen Menschen nicht wirklich lesbar ist. Man könnte also sagen, dass Programmiersprachen als eine Art Vermittler zwischen der natürlichen Sprache und dem für Menschen nur sehr schwer zu verstehenden Maschinencode agieren.

```
.method private hidebysig static
    void Main (
        string[] args
    ) cil managed
{
    // Method begins at RVA 0x2050
    // Code size 21 (0x15)
    .maxstack 1
    .entrypoint
    .locals init (
        [0] int32
    )

    IL_0000: nop
    IL_0001: ldc.i4 360
    IL_0006: stloc.0
    IL_0007: ldloc.0
    IL_0008: call void [mscorlib]System.Console::WriteLine(int32)
    IL_000d: nop
    IL_000e: call valuetype [mscorlib]System.ConsoleKeyInfo [mscorlib]System.Console::ReadKey()
    IL_0013: pop
    IL_0014: ret
} // end of method Program::Main
```

Abbildung 1.6: CIL-Code des ersten C#-Programms.

Das .NET-Framework

Ein Framework ist vergleichbar mit einem Baukasten. Dieser Baukasten enthält verschiedene Bausteine und Werkzeuge, um Anwendungen zu entwickeln, zu kompilieren und auszuführen. Sie wissen ja bereits, dass es sich bei C# um eine sogenannte Zwischensprache handelt und zur Ausführung einer Zwischensprache wird auf dem Zielrechner eine Laufzeitumgebung benötigt.

In Falle von .NET heißt diese Laufzeitumgebung *.NET-Framework*. Das .NET-Framework enthält alles, was Sie für die Entwicklung und Ausführung von .NET-Programmen benötigen. Dabei ist es möglich, sowohl klassische Desktop- als auch Web-Anwendungen zu erstellen. Für die Ausführung der Anwendungen ist beispielsweise der schon angesprochene Just-In-Time-Compiler fester Bestandteil des .NET-Frameworks. Daher muss das .NET-Framework, zur Ausführung eines .NET-Programms, auf dem jeweiligen Zielrechner installiert sein. Abbildung 1.7 zeigt einige wichtige Bestandteile des .NET-Frameworks.

Das .NET-Framework beinhaltet alles, was Sie für Entwicklung und Ausführung einer C#-Applikation benötigen. Neben verschiedenen Basisklassen (beispielsweise zum Lesen und Schreiben von Dateien oder zur Verarbeitung von XML-Dokumenten) enthält das .NET-Framework verschiedene Technologien für die Entwicklung von Benutzeroberflächen. Dabei können sowohl klassische Desktop-Anwendungen mit *Windows Forms* oder *WPF* als auch Web-Anwendungen mit *ASP.NET* entwickelt werden. Auf die unterschiedlichen Technologien zur Entwicklung von Benutzeroberflächen werde ich in einem späteren Kapitel noch genauer eingehen. Im nächsten Kapitel zeige ich Ihnen, wie eine .NET-Sprache grundsätzlich funktioniert, das heißt, wie ein Programm kompiliert und ausgeführt werden kann.

Die Laufzeitschicht, in der die .NET-Anwendungen ausgeführt werden, wird als *Common Language Runtime* (CLR) bezeichnet. Diese Laufzeitumgebung ist eine zwingende

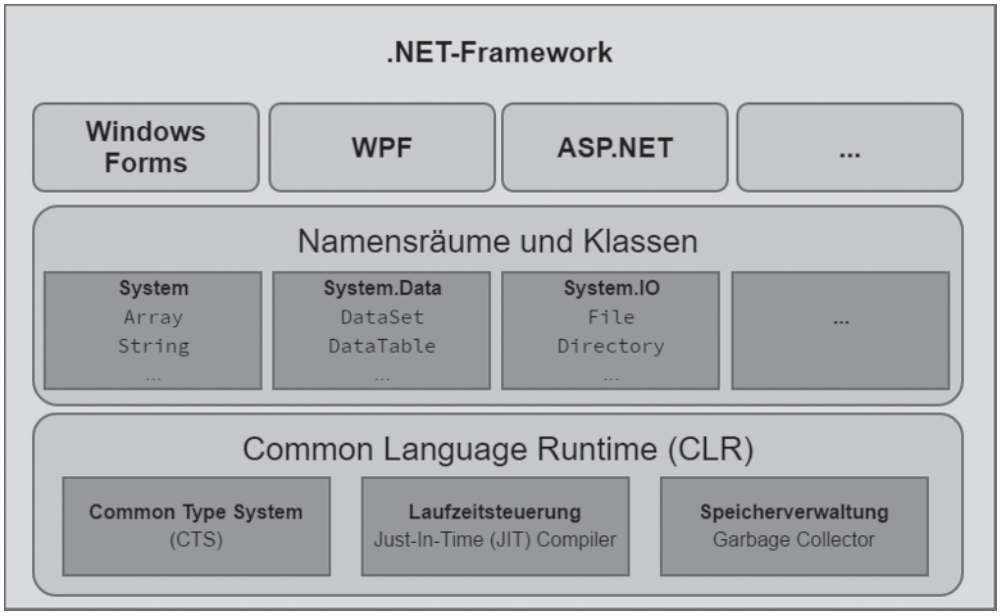

Abbildung 1.7: Wichtige Bestandteile des .NET-Frameworks.

Voraussetzung auf jedem Rechner, auf dem eine .NET-Anwendung ausgeführt werden soll. Die Laufzeitumgebung ist wiederum Bestandteil des .NET-Frameworks.

Verwalteter und nicht verwalteter Code

Bei der Arbeit mit dem .NET-Framework wird Ihnen häufig der Begriff »verwalteter Code« (engl. managed code) begegnen. Sehr einfach ausgedrückt, ist verwalteter Code genau das, was der Name besagt: Code, dessen Ausführung von einer Runtime verwaltet wird, also der Common Language Runtime. Die CLR übernimmt die Aufgabe, den verwalteten Code in Maschinencode zu kompilieren und dann auszuführen. Darüber hinaus bietet die Common Language Runtime mehrere wichtige Dienste wie die automatische Arbeitsspeicherverwaltung, Sicherheitsgrenzen, Typsicherheit usw.

Demgegenüber steht »nicht verwalteter Code«, dazu zählen beispielsweise C-/C++-Programme. Im nicht verwalteten Bereich ist der Programmierer für nahezu alles selbst zuständig. Das tatsächliche Programm ist im Wesentlichen eine Binärdatei, die vom Betriebssystem in den Arbeitsspeicher geladen und gestartet wird. Alles Übrige, von der Arbeitsspeicherverwaltung bis hin zu Sicherheitsaspekten, liegt in der Verantwortung des Programmierers.

Für die Entwicklung einer .NET-Anwendung ist die Minimalanforderung eine installierte .NET-Framework-Version auf dem Entwicklungsrechner. In neueren Versionen des Betriebssystem Microsoft Windows (ab Windows 7) ist das .NET-Framework bereits standardmäßig enthalten. Welche .NET-Framework-Version mit welchem Betriebssystem ausgeliefert wird, zeige ich Ihnen in Kapitel 2 *Entwicklungswerkzeuge und Tools*. Dort erfahren Sie auch mehr über Entwicklungswerkzeuge, die für die Entwicklung einer .NET-Anwendung notwendig sind. Kommen wir zunächst (endlich!) zum eigentlichen Kern dieses Buchs, nämlich der Programmiersprache C#.

Die Programmiersprache C#

Die Programmiersprache C# wurde 2002 offiziell mit dem ersten Release der .NET-Plattform vorgestellt und ist exklusiv für diese Plattform entwickelt worden. Aus diesem Grund ist C# für die Entwicklung von .NET-Anwendungen optimiert und daher auch am häufigsten anzutreffen. C# ist eine typische C-Sprache, die zwar ursprünglich sehr stark an Java angelehnt war, sich inzwischen aber enorm weiterentwickelt hat und einige exklusive Sprachfeatures bietet.

Die Programmiersprache C# wird ständig weiterentwickelt und trotz der langen Historie kommen immer wieder neue Funktionen und Syntaxkonstrukte hinzu. Mit den neuen Versionen des .NET-Frameworks ist es möglich, plattformunabhängig mit C# zu entwickeln. Sie können sogar die ursprünglich für Microsoft Windows entwickelten UI-Technologien, wie WPF, auf anderen Plattformen wie macOS oder Linux einsetzen. Somit ist C# bestimmt keine schlechte Wahl, um mit dem Programmieren zu beginnen. Es gibt eine Vielzahl von Programmiersprachen und ein Großteil dieser Programmiersprachen unterstützt die objektorientierte Programmierung. Was das genau bedeutet, schauen wir uns im nächsten Abschnitt an.

Das Prinzip der objektorientierten Programmierung

Wenn Sie mit C# programmieren, arbeiten Sie ständig mit *Klassen* und *Objekten*. Doch was genau bedeutet das? In jeder objektorientierten Programmiersprache sind Klassen und Objekte der Dreh- und Angelpunkt eines Programms. In den nächsten Abschnitten werden Sie lernen, was man genau unter den beiden Schlüsselbegriffen Klasse und Objekt versteht. Dabei werden ich Ihnen auch gleich die Grundkonzepte der objektorientierten Programmierung näher bringen.

Objekte und ihre Klassen

Stellen Sie sich vor, Sie sitzen in Ihrem Auto oder einem öffentlichen Verkehrsmittel und sind auf dem Weg zur Arbeit. Im Straßenverkehr begegnen Ihnen viele Fahrzeuge (beispielsweise PKWs und LKWs). All diese Fahrzeuge haben bestimmte Eigenschaften, beispielsweise

eingelegter Gang oder die aktuell gefahrene Geschwindigkeit und verfügen über Funktionen wie Starten, Beschleunigen oder Bremsen.

Das Konzept der objektorientierten Programmierung besteht darin, Objekte aus der realen Welt mit ihren Eigenschaften und Verhaltensweisen möglichst originalgetreu als Softwareobjekte abzubilden.

Die objektorientierte Programmierung soll den Entwickler bei der folgenden Herausforderung unterstützen: Mit Klassen werden Objekte aus der realen Welt abgebildet und daher soll der Programmablauf auch so gestaltet werden, wie er in der realen Welt stattfinden würde. Welchen Vorteil bietet das jetzt? Nun das erleichtert Ihnen den Dialog mit den Endanwendern. Es ist nämlich viel leichter, mit einem Anwender über Objekte und Abläufe aus der realen Welt zu sprechen, als über abstrakte Softwareobjekte, mit denen ein Endanwender, in den meisten Fällen, wenig anfangen kann. In der objektorientierten Terminologie würde man in unserem Beispiel etwa von den Klassen `PKW` und `LKW` sprechen. Die Klasse `PKW` stellt *Eigenschaften*, zum Beispiel Baujahr, und *Methoden*, wie beispielsweise Beschleunigen, zur Verfügung.

Mit einer Klasse werden Objekte aus der realen Welt abgebildet. Eine Klasse beschreibt *Attribute* (Eigenschaften) und *Methoden* (Verhaltensweisen) dieser Objekte.

Der konkrete PKW, in dem Sie beispielsweise zur Arbeit fahren, wird als *Objekt* der Klasse `PKW` bezeichnet.

Ein *Objekt* (auch *Instanz* genannt) repräsentiert in der objektorientierten Programmierung ein Exemplar einer bestimmten *Klasse*.

Um die Zusammenhänge zu verdeutlichen, setzt man sogenannte Klassendiagramme ein. Abbildung 1.8 zeigt ein Klassendiagramm für die Klasse *PKW*.

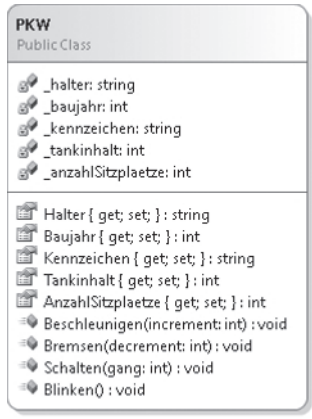

Abbildung 1.8: Klassendiagramm für einen PKW.

Innerhalb von Klassen werden *Eigenschaften*, *Methoden* und *Ereignisse* definiert. Zur Laufzeit eines Programms arbeiten Sie mit konkreten *Objekten* der jeweiligen *Klasse*.

Die Erzeugung eines konkreten Objekts einer bestimmten Klasse wird auch als *Instanziierung* bezeichnet. Für die Instanziierung wird ein sogenannter *Konstruktor* ausgeführt. Was ein Konstruktor ist, werden Sie bald lernen!

Das hat den großen Vorteil, dass alle Eigenschaften und Methoden, die zu einem bestimmten Objekt gehören, zusammengefasst (gekapselt) sind. Der Verwender des Objekts greift dann über eine definierte Schnittstelle (öffentliche Attribute und Methoden) auf das jeweilige Objekt zu. Die genauen Implementierungsdetails sind dem Verwender nicht bekannt und müssen ihn auch gar nicht belasten. Beispielsweise hat ein Verwender der Klasse `PKW` keine Kenntnis darüber, wie die Methode `Beschleunigen()` genau definiert ist, er ruft einfach die Methode auf und das Objekt der Klasse `PKW` beschleunigt. Abbildung 1.9 zeigt beispielhaft, wie die Objekte zur Laufzeit aussehen könnten.

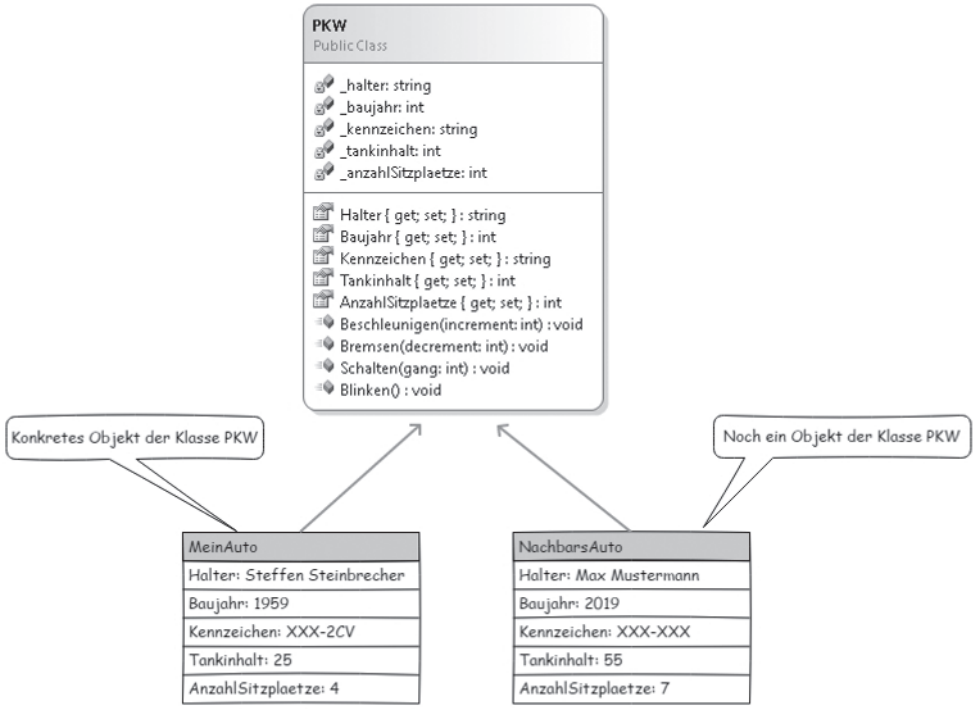

Abbildung 1.9: Konkrete Objekte einer Klasse `PKW` zur Laufzeit eines Programms.

Was ist in Abbildung 1.9 zu sehen? Dort sehen Sie zwei Objekte:

- ✔ *MeinAuto* – Objekt der Klasse `PKW`
- ✔ *NachbarsAuto* – noch ein Objekt der Klasse `PKW`

Mit den gezeigten Objekten können Sie während des Laufzeit eines Programms arbeiten. Sie könnten die Eigenschaften der Objekte abfragen, um beispielsweise das Baujahr des Objekts *MeinAuto* zu erfahren oder die `Beschleunigen()`-Methode aufrufen. Und das ohne dabei genaue Kenntnis darüber zu haben, wie die Implementierungsdetails aussehen. Das ist aber noch nicht alles, was die Objektorientierung zu bieten hat. Eine Klasse ist nicht nur einfach ein Bauplan für Objekte, es kommen noch die folgenden grundlegenden Konzepte der Objektorientierung zum Tragen:

- ✔ Abstraktion
- ✔ Kapselung von Daten und Methoden
- ✔ Vererbung
- ✔ Polymorphie

Das hört sich für einen Programmieranfänger wahrscheinlich noch sehr abstrakt an. Was man im Einzelnen unter diesen Konzepten versteht, erkläre ich Ihnen in den nächsten Abschnitten.

Abstraktion

Ziel der Abstraktion ist, die Realwelt möglichst originalgetreu in Klassen abzubilden. Das können die unterschiedlichsten Anforderungen sein, beispielsweise ein Programm zur Verwaltung eines Online-Shops. Dort wird es Klassen für Bestellungen, Rechnungen, Lieferscheine und viele weitere geben. Im Endeffekt sollen sich die erstellten Softwareobjekte dann wie die »echten« Gegenparts aus der Realwelt verhalten. Das kann je nach Anforderung eine sehr komplexe Aufgabe sein.

Das haben Sie auch schon am Beispiel der Fahrzeuge gesehen: Wir haben eine Klasse `PKW`, die Eigenschaften und Methoden bereitstellt, die ein PKW auch in der realen Welt hat. Die Herausforderung für den Programmierer besteht darin, die Funktionen, wie beispielsweise `Beschleunigen()`, in die gewünschte Programmiersprache zu übertragen. Die Abstraktion hilft auch in der Kommunikation mit dem Endanwender: Hier wird dann nicht über informatisch-technische Details gesprochen, sondern über Objekte, die in der realen Welt existieren. Das macht es Ihnen als Programmierer auch einfacher, eine komplexe Anforderung in einzelne Teilbereich aufzusplitten.

Stellen Sie sich einen Online-Shop vor: Dort wird es Klassen für Produkte, Kunden, Rechnungen und Lieferscheine geben, die zusammen ein komplexes System darstellen. Durch die Aufteilung in einzelne Klassen reduziert sich die Komplexität und jede Klasse übernimmt einen Teilbereich des Gesamtsystems. Das hat die folgenden Vorteile: Die einzelnen Klassen können von mehreren Entwicklern programmiert werden und am Ende wird alles zu einem Gesamtsystem verbunden. Sollte sich einmal ein Fehler einschleichen, was in der Softwareentwicklung immer wieder vorkommt, lässt sich das Problem sehr viel schneller eingrenzen: Nehmen wir einmal an, dass innerhalb des Online-Shops die Rechnungssumme falsch berechnet wird. Durch die Aufteilung in einzelnen Klassen wird der Fehler dann vermutlich in der Klasse `Rechnung` zu finden sein. Das macht die Sache wesentlich übersichtlicher und die Wartbarkeit der Software wird auch erheblich besser.

Kapselung

Die Eigenschaften eines Objekts werden im Idealfall nur durch die Methoden des Objekts geändert und nicht direkt durch den Verwender eines Objekts. Um die Geschwindigkeit eines Autos zu ändern, müssen Sie in der realen Welt beschleunigen oder bremsen. Übertragen auf das Softwareobjekt Auto heißt das, die entsprechenden Methoden `Beschleunigen()` oder `Bremsen()` aufzurufen. Es soll also nicht so einfach möglich sein bei einem Objekt, das ein Auto darstellt, die Geschwindigkeit auf einen anderen Wert zu setzen, ohne die zugelassenen Methoden zu benutzen. Stellen Sie sich einmal vor, in der realen Welt wäre es möglich, die Geschwindigkeit eines Autos von außen zu beeinflussen, da wären Unfälle im wahrsten Sinne des Wortes vorprogrammiert. Durch die Verwendung der Methoden `Beschleunigen()` und `Bremsen()` wird gewährleistet, dass sich das Objekt immer in einem konsistenten Zustand befindet.

Es bietet noch weitere Vorteile, Funktionen in einer Klasse zu kapseln: Details der Implementierung werden innerhalb der Klasse »versteckt«, der Verwender kennt die Implementierungsdetails nicht. Möchte ein Verwender Daten eines Objekts lesen oder verändern, muss er sich über eine klar definierte Schnittstelle an das Objekt wenden und eine Änderung der Daten anfordern. Dies sorgt dafür, dass das Objekt nur gemäß seiner Spezifikation verwendet wird. Übertragen auf unser Beispiel mit dem PKW bedeutet das Folgendes: Ein PKW hat Eigenschaften, die nicht änderbar sein sollen, dazu zählen beispielsweise das Baujahr oder die Fahrgestellnummer. Diese Eigenschaften sind fest vorgegeben und dürfen deshalb nicht modifiziert werden. Der Besitzer des PKWs hingegen kann sich ändern, etwa wenn der PKW verkauft wird. Diese Änderung wird durch die Zulassungsstelle, also von »außen«, vorgenommen. Der Kilometerstand des PKWs ist ebenfalls änderbar, aber mit der Besonderheit, dass der Kilometerstand nicht von »außen« änderbar sein soll. Der Kilometerstand des PKWs soll nur durch das Objekt selbst verändert werden können, und zwar dann und nur dann, wenn der PKW gefahren wird. Wäre der Kilometerstand von außen zu ändern, könnte jeder Verwender den Kilometerstand eines Objekts beliebig manipulieren. In der realen Welt wäre das ein Betrugsdelikt und daher wollen wir das in unserer Klasse auch nicht erlauben. Als Kapselung bezeichnet man also den kontrollierten Zugriff auf Eigenschaften und Methoden von Klassen. Damit sind Objekte in der Lage, sich selbst um die Konsistenz ihrer Daten zu kümmern.

Vererbung

Genau wie in der realen Welt können in der objektorientierten Programmierung Klassen untereinander eine Beziehung eingehen. Dieses Konzept wird auch *Vererbung* genannt und spielt eine große Rolle in der Softwareentwicklung. Die Vererbung wird beispielsweise dazu eingesetzt, ein Programm zu strukturieren. Sinnvoll eingesetzt kann Ihnen die Vererbung einiges an Arbeit abnehmen. Was ist damit gemeint? Nun mit der Vererbung ist es möglich, aus einer bereits existierenden Klasse eine neue Klasse zu erzeugen. Die vererbende Klasse wird Basis- oder Elternklasse genannt, die erbende Klasse Unter- oder Kindklasse. Wenn Klassen voneinander erben, spricht man auch oft von *Ableitung* oder *Spezialisierung*. Abbildung 1.10 verdeutlicht das.

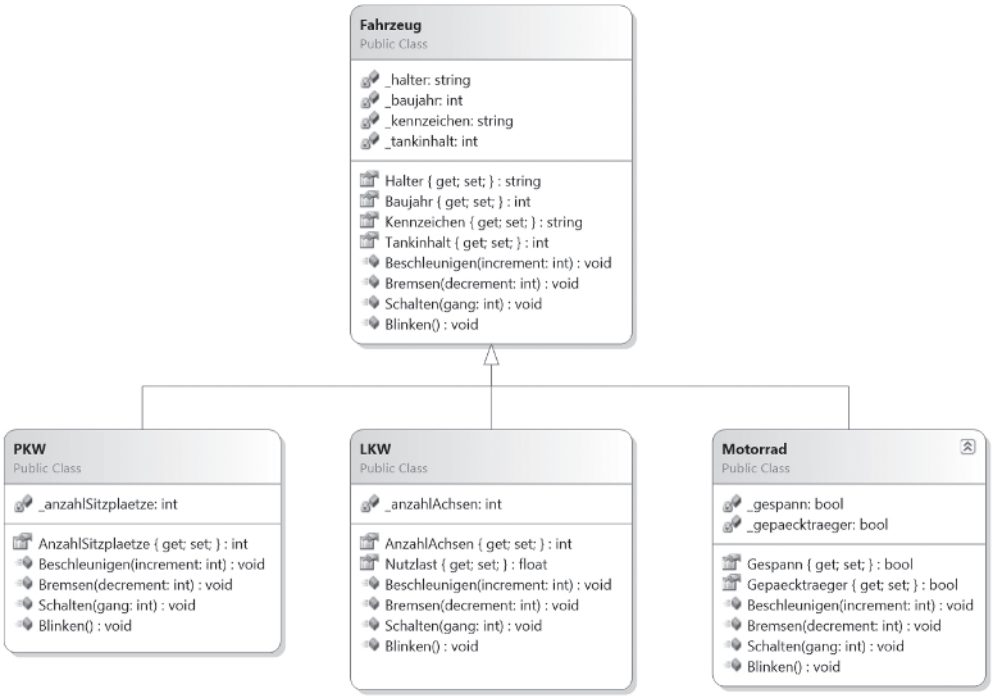

Abbildung 1.10: Klassendiagramm Fahrzeug und davon abgeleitete Klassen.

In diesem Klassendiagramm gibt es die allgemeine Basisklasse `Fahrzeug` und die abgeleiteten Klassen (Spezialisierungen) `PKW`, `LKW` und `Motorrad`. Dabei erben die abgeleiteten Klassen automatisch alle Eigenschaften und Methoden der Basisklasse. Ein `LKW` hat also alle Eigenschaften der Basisklasse `Fahrzeug`, und man kann alles mit ihm machen, was mit einem Fahrzeug so möglich ist, also etwa Aktionen wie Beschleunigen, Bremsen oder Schalten. Zusätzlich kann eine abgeleitete Klasse weitere Eigenschaften und Methoden haben, beispielsweise kann bei einem LKW die Anzahl der Achsen interessant sein oder bei einem Motorrad die Angabe, ob ein Gepäckträger vorhanden ist oder nicht. Die abgeleiteten Klassen können immer weiter spezialisiert werden, indem Sie beispielsweise `Sattelschlepper` oder `Kipper` von `LKW` ableiten.

Polymorphie

Polymorphismus ist ein weiterer wichtiger Aspekt der objektorientierten Programmierung. Es handelt sich dabei um ein griechisches Wort, das »Vielgestaltigkeit« bedeutet. Im Grunde ist die Vererbung eine schöne Sache, doch denken Sie einmal einen Schritt weiter: Ist die Implementierung der Methode `Beschleunigen()` in der Basisklasse `Fahrzeug` anforderungsgerecht? Wenn Sie sich im Straßenverkehr bewegen, habe Sie bestimmt schon festgestellt, dass ein PKW schneller beschleunigt als ein LKW. Ein Motorrad wiederum schafft den Sprint von 0 auf 100 noch deutlich schneller als der PKW.

Dadurch stehen wir vor der Frage: Wie kann eine Methode in der Basisklasse implementiert werden, wenn sich das reale Verhalten in den Methoden der abgeleiteten Klassen `PKW`, `LKW` und `Motorrad` spürbar unterscheidet? Wir könnten die Methode jetzt einfach aus der Basisklasse nehmen, was aber nicht zur Problemlösung beiträgt. Denn bezogen auf die Methode `Beschleunigen()` soll es ja weiterhin möglich sein, dass ein Fahrzeug beschleunigen und jede abgeleitete Klasse die Methode auch verwenden kann.

Um das zu realisieren, besteht die Möglichkeit, das Verhalten in den abgeleiteten Klassen zu überschreiben. Das bedeutet, dass die abgeleitete Klasse eine eigene Implementierung bereitstellt, um das geänderte Verhalten zu realisieren. Unter *Polymorphie* versteht man in der Objektorientierung das Folgende: Unterschiedliche Klassen können die gleiche Schnittstelle nach außen haben, sich aber trotzdem unterschiedlich verhalten. Für unser Beispiel würde das folgendes bedeuten: Wird auf einem Objekt der Klasse `Motorrad` die `Beschleunigen()`-Methode aufgerufen, wird sich diese anders verhalten als beispielsweise die `Beschleunigen()`-Methode der Klasse `LKW`.

Gerade als Programmieranfänger ist man versucht, die neu erlernten Konzepte so viel wie möglich einzusetzen, aber gerade beim Thema Vererbung kann sich das nachteilig auswirken. Die häufige Verwendung von Vererbungen ist keinesfalls ein Zeichen eines guten objektorientierten Entwurfs oder guter Programmierung. *Vererbung* birgt immer das Risiko, dass Klassen zu eng aneinander gekoppelt werden. Es droht auch die Gefahr, dass Klassenhierarchien entstehen, die nachträglich nicht mit vertretbarem Aufwand geändert werden können. Solche Konstrukte sind nur sehr schwer zu überblicken und ich habe schon Programme gesehen, die durch zur viel Vererbung praktisch nicht mehr zu bearbeiten waren. In solchen Fällen ist es oftmals besser, nochmal von vorne zu beginnen, als am zu komplex konstruierten Programm herumzufrickeln.

Um zu komplexe Klassenhierarchien zu vermeiden, gibt es eine vielversprechende Alternative in Form von sogenannten *Interfaces* (deutsch *Schnittstellen*). Auf die Programmierung mit Interfaces werde ich im Verlauf des Buchs noch ausführlich eingehen.

In Kapitel 6 *Objektorientierte Programmierung mit C#* lernen Sie, wie in C# objektorientiert programmiert wird.

Fassen wir noch einmal zusammen was Sie in diesem Abschnitt gelernt haben: Neben ein paar theoretischen Grundlagen zur Programmierung haben Sie ein paar wichtige Dinge über die .NET-Plattform und deren Konzepte erfahren. Damit haben Sie den Theorieteil auch schon geschafft und den Grundstein gelegt, um in die Programmierung mit C# einzusteigen.

Zusammenfassung

- ✔ .NET ist eine Softwareentwicklungsplattform für die Entwicklung von Desktop- und Web-Anwendungen in einer der verschiedenen .NET-Programmiersprachen.
- ✔ Ein Compiler ist ein Programm zur Übersetzung einer Programmiersprache in ausführbaren Maschinencode.
- ✔ Das .NET-Framework ist eine Laufzeitumgebung für .NET-Sprachen und muss auf dem Computer installiert sein, um ein .NET-Programm auszuführen zu können.
- ✔ C# ist eine Programmiersprache, die zu den sogenannten Zwischensprachen gehört (für Zwischensprachen werden ein Compiler und ein Interpreter benötigt).
- ✔ CIL steht für Common Intermediate Language und ist eine Zwischensprache, die in nativen Maschinencode übersetzt wird (sogenannte JIT-Kompilierung).
- ✔ Unter Objektorientierung versteht man ein Programmierparadigma, das auf den Konzepten Abstraktion, Kapselung, Vererbung und Polymorphie aufbaut.