Programmieren lernen

Grundlagen für Studium und Beruf - praxisnah und sprachunabhängig

Kapitel 1

Einführung in die Welt der Computerprogramme

Bevor ich Sie mit den Grundlagen und Konzepten der Programmierung vertraut mache, erhalten Sie in diesem Kapitel eine allgemeine Einführung zum Thema Programmieren. Dazu gehört ein kurzer Abriss aus der Geschichte, hier erfahren Sie, wie es zur Erfindung von Computerprogrammen kam und wie sich Programme und Programmiersprachen über die Jahrzehnte entwickelt haben. Danach erkläre ich Ihnen kurz, wie Programmiersprachen und Programmcode funktioniert und anschließend bekommen Sie eine Übersicht über die Einsatzgebiete von Computerprogrammen sowie über die gängigen Programmiersprachen.

1.1 Eine kleine Reise durch die Geschichte der Programmierung

Auch wenn wir heute auf Computern programmieren, überlegt man kurz, wird klar, dass ein Computer nur funktioniert, weil darauf ein Programm läuft, das das Arbeiten mit dem Computer erst möglich macht. Programmiersprachen, Programme und das Schreiben eben jener wurde also vor dem modernen Computer erfunden. Die Programmierung von Maschinen ist älter, als viele denken.

Vor den Computern gab es Rechenmaschinen und vor den Rechenmaschinen gab es nur mechanische Maschinen. Diese Maschinen waren in der Lage, bestimmte Aufgaben oder Arbeitsschritte automatisch zu erledigen. Die Funktion basiert auf deren Bauweise bzw. der mechanischen Konstruktion. Die Rede ist von industriellen Maschinen wie etwa Stanzen und Pressen, aber auch komplexeren Konstruktionen, wie etwa mechanischen Webstühlen. Aber egal wie komplex die Arbeitsschritte auch waren, die Ausführung erfolgte ohne jegliche Logik. Zwar konnten Werkzeuge und Aufsätze getauscht werden, dafür war aber ein menschliches Eingreifen erforderlich. Um andere Arbeitsschritte auszuführen, benötigte man eine andere mechanische Konstruktion. So konnten die ersten mechanischen Webstühle immer nur ein bestimmtes Muster weben.

Zu Beginn des 19. Jahrhunderts erfand der Franzose Joseph-Marie Jacquard den programmierbaren Webstuhl. Und auch wenn dieser noch keine Rechenmaschine war, folgte dieser dem Grundprinzip der elektronischen Datenverarbeitung: Ein-

gabe – Verarbeitung – Ausgabe (EVA). Über Lochstreifen konnte dem Webstuhl das zu webende Muster übermittelt werden.

Die erste mechanische Rechenmaschine wurde 1837 von Charles Babbage erfunden. Als erste mathematisch-logische Programmierung gilt die Vorschrift zur Berechnung von Bernoulli-Zahlen[1] von Ada Lovelace, die als erste Programmiererin der Geschichte gilt.

Zu Beginn des 20. Jahrhunderts wurden weitere Rechenmaschinen entworfen und dafür wiederum Vorschriften zum Lösen mathematischer Probleme entwickelt. Als Meilensteine gelten die vom deutschen Bauingenieur Konrad Zuse erfundenen und gebauten Rechenmaschinen. In den Vierzigerjahren des 20. Jahrhunderts griff der österreichisch-ungarische Mathematiker John von Neumann einige Ideen Zuses auf und beschrieb mit der Von-Neumann-Architektur ein Referenzmodell für Computer. Diese bildet die Grundlage für die meisten der heute bekannten Computer.

Nach dem Zweiten Weltkrieg entstanden, in den USA, die ersten höheren Programmiersprachen FORTRAN, Lisp und COBOL. Die an der Entwicklung von COBOL beteiligte Grace Hopper entwickelte auch den ersten Compiler (Abschnitt 1.2.1 »Kompilierte Programmiersprachen«). In den Sechzigern und Siebzigern des vergangenen Jahrhunderts wurde eine Vielzahl weiterer Programmiersprachen entwickelt. Grundlage war der technische Fortschritt in der Entwicklung von Computer-Hardware. Viele dieser Sprachen sind längst vergessen, andere werden noch heute verwendet oder entwickelten sich zu einigen der aktuellen Sprachen weiter. Nennenswerteste Vertreter sind die Sprachen BASIC, welche durch die ersten leistbaren Heimcomputer der Siebziger populär wurde, sowie C, welche 1972 für das neue Betriebssystem Unix entwickelt wurde.

Ende der Siebzigerjahre stellte das US-Verteidigungsministerium erschrocken fest, dass über 450 teils nicht standardisierte Programmiersprachen in deren Projekten genutzt wurden. Der Versuch, eine Sprache zu finden, die alle Anforderungen des US-Militärs erfüllte, scheiterte. Das Militär entwickelte daraufhin eine eigene Sprache, Ada, benannt nach Ada Lovelace.

1983 stellte Bjarne Stroustrup C++ vor, eine Erweiterung von C, die jene neuen Konzepte enthielt, die seit der Erfindung von C Einzug in die Computerprogrammierung gehalten hatten. Das Wichtigste hierbei ist die objektorientierte Programmierung, bei der die Architektur eines Programms sich an jenen Objekten der wirklichen Welt anlehnt, die die Aufgabenstellung des Programms betreffen (siehe Kapitel 4 »Objektorientierte Programmierung«).

1 Als Bernoulli-Zahlen wird eine Reihe von rationalen Zahlen bezeichnet, die in verschiedenen mathematischen Bereichen Anwendung finden.

Die schnelle Ausbreitung des Internets stellte eine ganze eigene Herausforderung dar, denn plötzlich mussten Inhalte auf eine Vielzahl verschiedener Endgeräte angepasst werden. Der Erfolg des Internets begründet sich unter anderem auf HTML – keine Programmiersprache, sondern eine Auszeichnungssprache –, welche die Gestaltung von Webinhalten übernahm, und PHP, der Programmiersprache für Serverlogik und dynamische/interaktive Inhalte.

1995 folgte die plattformunabhängige, objektorientierte Programmiersprache Java und 2001 die von Microsoft beauftragte Sprache C#.

Die nachfolgenden Entwicklungen brachten Sprachen hervor, die entweder Schwachstellen in den bereits verbreiteten Sprachen beheben sollten oder die für spezielle Einsatzgebiete gedacht waren (mehr dazu siehe Kapitel 7 »Welche Programmiersprache ist die richtige für mich?«).

1.2 Wie funktioniert das Programmieren von Computern?

Was ist Programmieren eigentlich? In der deutschen Sprache spricht man vom Programmieren, wenn man sein Haushaltsgerät oder ein anderes elektronisches Gerät konfiguriert. Wer alt genug ist, hat vielleicht schon mal einen Videorekorder »programmiert«. Unter Software- oder Computerprogrammierung versteht man aber das Schreiben von Befehlsabläufen, die einen Computer anweisen, bestimmte Aufgaben durchzuführen.

Wichtig zu erwähnen ist, dass der Computer bzw. genauer gesagt dessen Prozessor die Programmiersprache, in der die Befehlsabläufe geschrieben sind, gar nicht versteht. Sie dient dazu, die Befehlsabläufe, sogenannte Algorithmen, dem menschlichen Verständnis möglichst nahekommend zu formulieren. Die Befehle, die mit einer bestimmten Programmiersprache geschrieben werden, müssen erst in einen Maschinencode übersetzt werden, um vom Computer ausgeführt werden zu können.

Hinweis

Die folgende Erklärung zur Programmierung von Computern, die Sie hier finden, soll ein grundlegendes Verständnis für die Thematik schaffen. Der tatsächliche Vorgang ist sehr technisch und komplex und eine Erläuterung aller Details passt nicht in den Rahmen dieses Buchs. Sollte Sie das Thema interessieren, finden Sie dazu online umfangreiche Erläuterungen. Auch in der Fachliteratur gibt es ein umfangreiches Angebot an Werken zu diesem Thema, etwa Code von Charles Petzold, ebenfalls erhältlich beim mitp-Verlag.

1.2.1 Kompilierte Programmiersprachen

Diese Art von Programmiersprachen verwenden einen sogenannten *Compiler* (deutsch: Übersetzer). Dabei handelt es sich um ein Computerprogramm, das den Quellcode, der in einer bestimmten Programmiersprache geschrieben wurde, in vom Prozessor verständlichen Maschinencode übersetzt (kompiliert).

Der Maschinencode enthält Befehle, die vom jeweiligen Prozessor, der den Code ausführt, verstanden werden. Unterschiedliche Hardware kann dabei unterschiedliche Befehlssätze aufweisen. Beim Kompilieren wird der Code auch optimiert, das heißt, er wird gekürzt, anders angeordnet und durchläuft weitere Änderungen, die das Programm schneller machen und weniger Speicher verbrauchen lassen.

```
1  int main() {
2      int a = 4;
3      int b = 1;
4      int c = a + b;
5      return c;
6  }
```

Listing 1.1: Simpler Programmcode, der in C, Java, C# und anderen Sprachen geschrieben worden sein könnte

```
1   55
2   48 89 E5
3   C7 45 FC 04
4   C7 45 F8 01
5   8B 45 F8
6   8B 55 FC
7   01 D0
8   89 45 F4
9   8B 45 F4
10  5D
11  C3
```

Listing 1.2: So könnte der Maschinencode für das Programm aus Listing 1.1 aussehen.

Ein simples Programm wie das aus Listing 1.1 ist selbst für Menschen ohne Programmiererfahrung halbwegs verständlich, die Variablen a und b werden addiert und das Ergebnis in die Variable c gespeichert, welche an die aufrufende Stelle zurückgegeben wird. Maschinencode (Listing 1.2) hingegen ist für einen Menschen nicht mehr lesbar. Natürlich wäre es möglich, Maschinencode zu lernen, das Unterfangen wäre aber eher mühsam. Dazu kommt noch die Tatsache, dass der Maschinencode je nach Plattform ganz unterschiedlich ausfallen kann.

Sie sehen also, es handelt sich um ein eher sinnloses Unterfangen. Dank des Compilers können Programmierer den Code mithilfe von Programmiersprachen in einem relativ einfach lesbaren Format schreiben und diesen dann für verschiedene Plattformen übersetzen lassen, ohne den Code für jede Plattform einzeln anpassen zu müssen. Bekannteste Vertreter kompilierter Programmiersprachen sind C bzw. C++.

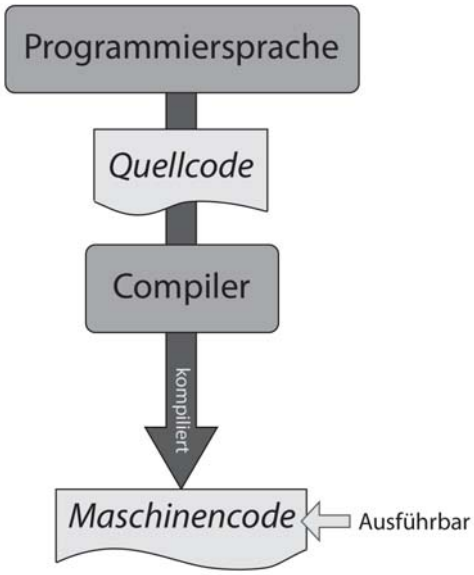

Abb. 1.1: Schematischer Ablauf bei kompilierten Programmiersprachen

Bis zur Entwicklung des ersten Compilers durch Grace Hopper Mitte der Fünfzigerjahre war das direkte Schreiben von Maschinencode allerdings die einzige Möglichkeit, Programme zu entwickeln. Damals gab es aber auch weniger verschieden Plattformen und Funktionalitäten der damaligen Computer – und somit war die Anzahl an Befehlen geringer als heute.

1.2.2 Interpretierte Programmiersprachen

Technisch etwas anders sieht es bei den sogenannten *interpretierten Sprachen* aus. Diese werden nicht direkt in Maschinencode übersetzt, ehe das Programm ausgeführt werden kann. Der Programmcode bleibt grundsätzlich, wie er ist. Erst beim Ausführen des Programms wird er an eine spezielle Software – den sogenannten *Interpreter* – übergeben. Dieser interpretiert den geschriebenen Code in Echtzeit und kennt alle Anweisungen der Programmiersprache, die er dann als Maschinencode ausführt. Bekannte Vertreter von interpretierten Sprachen sind Python und JavaScript.

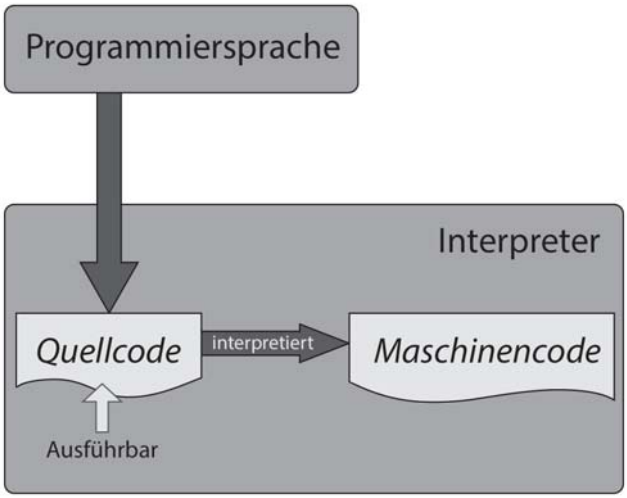

Abb. 1.2: Schematischer Ablauf bei interpretierten Programmiersprachen

1.2.3 Kompilieren vs. Interpretieren

Der Unterschied zwischen den beiden Vorgängen kann etwas verwirrend sein. Am besten hilft hier ein analoges Beispiel. Denken Sie an eine Gebrauchsanleitung oder eine Aufbauanleitung und nehmen Sie an, diese ist in einer Sprache geschrieben, die Sie nicht verstehen. Sie haben zwei Möglichkeiten, die Anweisungen der Anleitung umzusetzen:

- Option A: Es gibt bereits eine Übersetzung der Anleitung. Diese wurde im Vorfeld von einer Person (Compiler) angefertigt, die sowohl die Sprache der Originalanleitung als auch Ihre Sprache beherrscht.
- Option B: Sie kennen jemanden (Interpreter), der sowohl Ihre Sprache spricht als auch die Sprache der Anleitung versteht, und bitten denjenigen, die Anleitung direkt für Sie zu übersetzen.

Vereinfacht ausgedrückt, machen beide Methoden dieselbe Arbeit, aber zu einem anderen Zeitpunkt (vor dem Ausführen des Programms vs. während dem Ausführen) und an einem anderen Ort (Rechner des Entwicklers vs. Rechner des Anwenders).

Natürlich haben beide Methoden gewisse Vor- und Nachteile. Diese sind für Anfänger eventuell noch schwieriger zu verstehen als die unterschiedlichen Funktionsweisen der beiden Methoden. Aber gerade als Anfänger sollten Sie sich auch nicht allzu viele Gedanken darüber machen. Ihr Fokus sollte auf dem Lernen der Grundlagen liegen, dem Üben und dem Sammeln von Erfahrung durch das Schreiben von Code.

Dennoch möchte ich Ihnen im Folgenden eine vereinfachte Übersicht über Vor- und Nachteile der beiden Methoden bieten.

An dieser Stelle sollte noch erwähnt werden, dass, auch wenn von interpretierten bzw. kompilierten Programmiersprachen die Rede ist, diese Übersetzungsmethoden keine Eigenschaft der Sprache selbst sind, sondern davon abhängen, wie die Sprache implementiert, also technisch umgesetzt wird. Es gibt Programmiersprachen, die sowohl kompiliert als auch interpretiert werden können. Python ist hier ein gutes Beispiel. Primär als Skriptsprache eingesetzt, wird Python interpretiert. Durch die wachsende Beliebtheit und die breit gefächerten Einsatzmöglichkeiten gibt es für diverse Plattformen auch Python-Compiler.

Just-in-time-Compiler

Für immer mehr Sprachen stehen Just-in-time-Compiler zur Verfügung. Diese speziellen Compiler übersetzten Code bzw. die Änderungen am Code in Echtzeit. Sie vereinen den flexiblen Entwicklungszyklus von interpretierten Sprachen mit der schnellen Ausführung von kompilierten Sprachen.

Ausgeliefert wird immer noch kompilierter Code. Durch die Echtzeitübersetzung muss der Code aber nicht nach jeder Änderung neu kompiliert werden, um ihn zu testen. JIT-Compiler gehen so weit, dass das Programm auch nicht mehr beendet werden muss, um Änderungen am Code vorzunehmen. Das ist besonders bei sehr komplexen Programmen hilfreich, wo es mitunter einige Anwendungsschritte und Zeit benötigt, um die getätigte Änderung sichtbar zu machen. Auch bei Fehlern, die nur schwierig zu reproduzieren sind, helfen JIT-Compiler enorm.

Vorteile kompilierter Programmiersprachen

Kompilierte Programme sind in der Regel schneller, da das Übersetzen in Maschinencode schon vor dem Ausführen passiert. Der Anwender erhält das bereits übersetzte Programm und benötigt dafür auch keine zusätzlichen Ressourcen.

Nachteile kompilierter Programmiersprachen

Das Kompilieren erfordert einen zusätzlichen Zeit- und Ressourcenaufwand zwischen dem Schreiben und Ausführen des Programms. Dieser Aufwand kann bei großen Programmen stark anwachsen und verlangsamt den Entwicklungszyklus. Code muss immer wieder getestet werden, aber bevor das geschehen kann, muss der Code kompiliert werden. Dasselbe gilt für die Fehlersuche. Um den Code zu verändern, muss das Programm gestoppt und nach der Änderung wieder kompiliert werden.

Bei großen Softwareprojekten arbeiten mehrere Entwickler am Code. Dieser wird dann nur noch zu Testzwecken am jeweiligen Rechner des Entwicklers kompiliert. Das Kompilieren des finalen Programms, ehe dieses produktiv genutzt werden kann, erfolgt auf eigenen Servern. Je mehr Code kompiliert werden muss, umso mehr Ressourcen muss für diese Server bereitgestellt werden.

Ein weiterer Nachteil ist, dass ein Compiler immer plattformabhängig ist. Soll ein Programm auf verschiedenen Plattformen laufen, muss der Code für jede Plattform separat kompiliert werden.

Vorteile interpretierter Programmiersprachen

Interpretierte Programmiersprachen sind in der Regel etwas flexibler. Da der Interpreter das Übersetzen auf der Zielplattform übernimmt, ist der Code selbst immer plattformunabhängig und kann von jeder Plattform genutzt werden. Interpreter erlauben bzw. vereinfachen gewisse Programmierparadigmen und -funktionen (z.B.: Reflection und dynamic typing), deren Erklärung aber über den Rahmen dieses Buchs hinausgeht.

Nachteile interpretierter Programmiersprachen

Primärer Nachteil ist die langsamere Ausführungszeit von Programmen, die erst interpretiert werden müssen. Außerdem muss der Anwender einen Interpreter für die verwendete Sprache installiert haben.

Ein weiterer Nachteil ist, dass gewisse Fehler, die Kompilierfehler (compiler error), wie der Name schon verrät, nicht vorab gefunden werden. Normalerweise werden sie zum Zeitpunkt des Kompilierens erkannt und führen zu einem Abbruch des Kompilierens. Der Interpreter aber arbeitet eine Codezeile nach der anderen ab und vergisst, was in den Zeilen zuvor passiert ist. Fehler treten dann zum Zeitpunkt des Ausführens auf und lassen das Programm abstürzen oder führen zu ungewolltem Verhalten bzw. falschen Ergebnissen. Wird Code nicht gewissenhaft getestet, bevor er ausgeliefert wird, fallen Fehler erst beim Anwender auf und das Programm ist nicht ausführbar.

Moderne Programmierumgebungen (IDEs, siehe Abschnitt 5.2 »Die Entwicklungsumgebung«) bieten diverse Funktionen, um das Schreiben von Code zu erleichtern. Diese können für interpretierte Sprachen eingeschränkt oder gar nicht zur Verfügung stehen.

1.2.4 Bytecode und Laufzeitumgebungen

Da Computerprogramme lange nicht mehr nur mathematische Probleme lösen, sondern komplexe Aufgaben übernehmen sollen, wie Dateien zu erstellen und zu manipulieren, oder über das Netzwerk zu kommunizieren, reicht ein Compiler heute nicht mehr aus. Als Programmierer wollen Sie aber den Maschinencode für

diese Aufgaben (meist) nicht selbst schreiben. Höhere Programmiersprachen bieten eine Vielzahl und Funktionen und Schnittstellen zu allen möglichen Hardwarekomponenten eines Computers, wie etwa grafische Ausgabe, Tonwiedergabe, Arbeits- und Festplattenspeicher oder eben Netzwerkschnittstellen. Damit Ihr Programm diese Funktionen nutzen kann, muss der Compiler entweder all diese Funktionen zusammen mit Ihrem Quellcode in das Programm integrieren oder der erzeuge Code wird in einer sogenannten *Laufzeitumgebung* (Runtime Environment) ausgeführt.

Hierbei wird der Quellcode nicht direkt in Maschinencode kompiliert, sondern in sogenannten *Bytecode*. Der Bytecode liegt in binärer Form (Folgen von Nullen und Einsen) vor, er ist also vom Menschen nicht mehr lesbar. Er ist immer noch plattformunabhängig, hat aber bereits einige Optimierungen durchlaufen. Der Bytecode wird von der Laufzeitumgebung entweder zu Maschinencode kompiliert oder direkt interpretiert.

Je nach Betriebssystem sind Laufzeitumgebungen für einige Sprachen bereits integriert. Andere Laufzeitumgebungen müssen Sie selbst nachinstallieren. Das trifft im Übrigen auch dann zu, wenn Sie ein fertiges Programm einfach nur ausführen wollen und nicht selbst programmieren. Die bekanntesten Laufzeitumgebungen sind das Java Runtime Environment (JRE) oder das .Net-Framework für C# und weitere verwandte Sprachen.

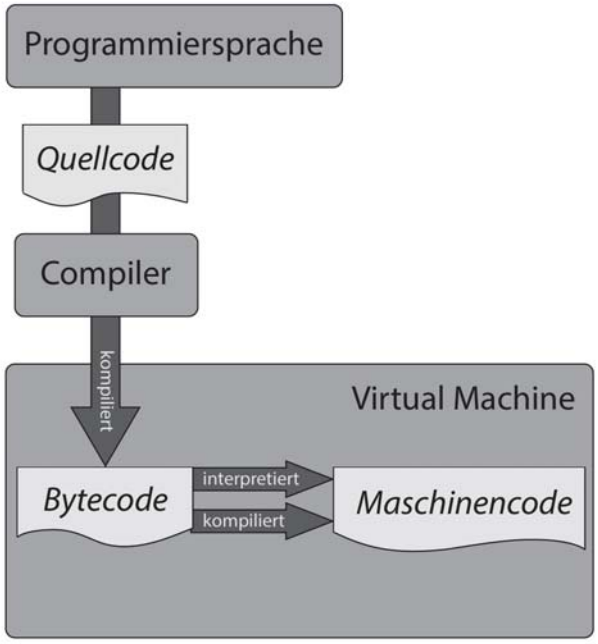

Abb. 1.3: Schematischer Ablauf bei Programmiersprachen mit Laufzeitumgebung