Programmierung: Konzepte und Begriffe*

Hans Jürgen Ohlbach

11. April 2018

Keywords: Programmierparadigmen: imperativ, funktional, objektorientiert, aspektorientiert, nebenläufig; Constraint Handling, Compiler und Interpreter, Virtuelle Maschinen, Typsysteme, Ausnahmebehandlung, Design Patterns, Testen, Debuggen, Profilen, Disassemblieren, Verifizieren, Kommentieren, Programmiersprachen

Empfohlene Vorkenntnisse: Grundkenntnisse zur Prozessorarchitektur, Maschinensprache, einfache Datentypen sind hilfreich

Inhaltsverzeichnis

*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linie für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Einleitung

Viele Einführungskurse zur Programmierung basieren auf einer konkreten Programmiersprache. Sie fangen mit ganz einfachen Sprachelementen an, und hangeln sich dann weiter zu komplexeren Sprachelementen. Am Ende können die Anfänger dann ganze einfache kleine Programme schreiben, gerade ausreichend, um die Klausuraufgaben zu lösen. Den ganzen Rest zur Programmierung, und der ist riesig, müssen sie sich selbst ganz alleine, oder zusammen mit anderen Anfängern in Praktika, aneignen.

In diesem Miniskript wird daher ein Überblick über „den ganzen Rest“ gegeben. Es ist nur ein Überblick. Er soll dazu dienen, Techniken kennen zu lernen, Dinge einzuordnen, Zusammenhänge zu verstehen, und vielleicht dazu anregen, sich mit einzelnen Aspekten genauer zu beschäftigen.

2 Die Abstraktionshierarchie und ihre Hilfsmittel

Zunächst einmal schaffen wir uns einen Überblick über die verschiedenen Ebenen, die in eine komplexe Anwendung involviert sind, von der Prozessorebene, bis zum Endprodukt, dem Anwendungsprogramm.

2.1 Maschinensprache

Die Prozessoren in einem Computer verstehen zunächst nur die Maschinensprache. Jeder Maschinenbefehl ist eine Bitfolge, bestehend aus der Nummer des Befehls (dem sog. Op-Code), und den Argumenten. Um z.B. zwei Zahlen zu addieren muss der Maschinenbefehl die Nummer des Addierbefehls enthalten, sowie die Positionen der Zahlen, die zu addieren sind, und die Position, wo das Ergebnis hin soll. In manchen Maschinen sind diese Positionen standardisiert, z.B. auf einem Stack. Dann braucht man nur die Nummer des Addierbefehls.

2.2 Assemblersprache

Niemand schreibt jedoch Programme direkt in Maschinensprache. Will man wirklich auf dieser Ebene programmieren, dann macht man es in *Assemblersprache*. Diese ist im wesentlichen eine menschenlesbare Form der Maschinensprache. Anstelle der Nummer des Addierbefehls, z.B. benutzt man das Wort ADD. Auch die Positionen der Argumente muss man nicht als binäre Adressen angeben, sondern man vergibt Namen (*Variablennamen oder Registernamen*), und diese werden den tatsächlichen Positionen zugeordnet.

Die Übersetzung von der Assemblersprache in Maschinensprache übernimmt ein Programm, der *Assembler*. Der Assembler kann auch noch weitere Funktionen übernehmen, z.B. die automatisierte Verwaltung von Sprungadressen, den Aufruf von Unterprogrammen usw.

2.3 Hochsprachen

Auch in Assemblersprache programmiert kaum jemand. Stattdessen benutzt man eine *Hochsprache*, eine Programmiersprache für menschliche Programmierer. Eine Wikipediaseite listet davon über 400 Sprachen auf, viele veraltet, aber auch noch viele in Benutzung. Immer wieder kommen auch noch neue Programmiersprachen hinzu.

2.4 Compiler und Interpreter

Es gibt mehrere Möglichkeiten, ein Programm in einer Hochsprache zum Laufen zu bekommen:

Compilation (Übersetzung): ein *Compiler* übersetzt das Programm in Maschinensprache, den sog. *Binärcode*. Oft wird das nicht direkt gemacht, sondern in einer oder mehreren Zwischenstufen. Typisch ist zunächst eine Übersetzung in die relativ maschinennahe Sprache C, von dort in einen Assembler, und dann in Maschinensprache. Der Vorteil von einer Übersetzung in C ist, dass C noch weitgehend unabhängig von der Prozessorarchitektur ist. Erst die Assemblersprachen sind prozessorspezifisch. Für die Entwicklung einer neuen Programmiersprache genügt es daher, eine Übersetzung in C zu entwickeln. Damit hat man dann automatisch Übersetzungen in alle Prozessorarchitekturen, für die des Compiler für C gibt.

Interpretation: Ein sog. *Interpreter* führt das Programm Zeile für Zeile direkt aus, so ähnlich wie ein Laie ein Kochrezept Schritt für Schritt abarbeitet. Man braucht dann keinen, manchmal zeitaufwendigen, Compilationsschritt, bevor man das Programm testen kann, sondern kann sofort mit dem Testen anfangen. Manche Interpretierer erlauben sogar, das Programm anzuhalten, dessen momentanen Status abzufragen, das Programm sogar zu ändern, und mit dem veränderten Programm weiterzurechnen.

Virtuelle Maschine: Dies ist eine Art Simulator für eine Prozessorarchitektur, mit eigener Maschinensprache. Das Programm wird in diese Maschinensprache übersetzt, und der Simulator führt es dann aus. Ein bekanntes Beispiel dafür ist die *Java Virtual Maschine (JVM)*. Der Java Compiler übersetzt in den sog. *Java Byte Code*, der dann von der JVM ausgeführt wird. Der Vorteil von dieser Architektur ist, dass man zwar die Virtuelle Maschine einmal für jedes Betriebssystem und jede Prozessorarchitektur entwickeln muss, dann aber die eigentliche Programmiersprache weiter entwickeln kann, indem man nur den Compiler in die virtuelle Maschinensprache ändert, aber nicht die Virtuelle Maschine selbst.

Mehr noch, man kann unterschiedliche Sprachen entwickeln, die in dieselbe virtuelle Maschine compilieren. Damit können sich Programmteile aus verschiedenen Sprachen sogar gegenseitig aufrufen. Beispiele sind Java, Ceylon, Clojure, Erjang, Free Pascal, Groovy, JRuby, Jython, Scala und Kotlin, die alle in die JVM übersetzen.

Die Java Virtual Maschine ist eine rein abstrakte Maschine, für die es keinen Prozessor gibt. Es gibt aber auch virtuelle Maschinen für existierende Prozessoren, sog. *Emulatoren*. Diese simulieren häufig alte Hardware, z.B. Spielekonsolen, die es nicht mehr auf dem Markt gibt, für die es aber noch die Programme gibt.

Die Zwischenschicht der Virtuellen Maschine verlangsamt die Programme leider etwas. Als Gegenmaßnahme hat man sog. *Just-In-Time Compiler* (JIT) entwickelt, die in einem laufenden

Programm beobachten, welche Teile häufig benutzt werden, und diese dann direkt in Maschinensprache übersetzen.

2.5 Integrated Development Environment (IDE)

In alten Zeiten hat man Programme in einem Texteditor geschrieben, in einen File gespeichert, und dann compiliert und ausgeführt. Damit ist die Entwicklung umfangreicher Programme sehr aufwendig, unübersichtlich und zeitaufwendig.

Heutzutage verwendet man spezielle Programme, die eine Vielzahl von Hilfsmitteln zur Verfügung stellen, um schnell und so fehlerfrei wie möglich komplexe und umfangreiche Programme zu entwickeln.

Dazu gehören insbesondere:

- ein syntaxgesteuerter Editor, der sofort Tippfehler bemerkt,
- eine *Autocompletion*, welches angefangene Wörter vervollständigt, sobald das eindeutig ist,
- ein Hilfesystem, welches während des Eintippens Informationen darüber gibt, was an dieser Stelle stehen könnte,
- eine Verwaltung von Programmteilen und Bibliotheken,
- ein integrierter Compiler, um mit einem Knopfdruck das Programm compilieren zu können,
- ein *Debugger*, um in ein laufendes Programm „hineinschauen“ zu können, um Fehler zu finden,
- Unterstützung für Build-Automatisierungssysteme wie Maven oder Gradle welche Programme, die aus vielen Programmteilen, Bibliotheken und Daten bestehen, automatisch zu einem lauffähigen Programm zusammenbinden,
- ein *Profiler*, um ein laufendes Programm vermessen zu können. Es zählt die Aufrufe von Unterprogrammen, und die Zeiten, die in den einzelnen Teilen des Programms verbracht werden,
- einen Anschluss zu einem Versionsverwaltungssystem wie SVN oder GIT,
- Unterstützung für das Testen von Programmen,
- ein Hilfesystem für die Programmiersprache selbst, so dass man kaum noch Manuals lesen muss.

Weit verbreitete IDEs, die insbesondere für Java entwickelt wurden, sind *Eclipse*, *Netbeans* und *IntelliJ*. Es gibt aber noch viele weitere IDEs. Wikipedia listet knapp 90 davon.

2.6 Frameworks

Komplexe Anwendungen erfordern sehr oft die Lösung derselben Probleme, z.B. Benutzerverwaltung, Login- und Passwortmechanismus, Datenbankanbindungen, Graphische Benutzeroberflächen,

Webseitenprogrammierung, Warenkörbe bei Internetshops, Bezahlmechanismen, Content Management usw.

Damit nicht jeder Anwender all diese Dinge neu programmieren muss, gibt es sog. Frameworks. Das sind im Wesentlichen Bibliotheken, die eine Unmenge von Funktionalitäten zur Verfügung stellen, die dann von den unterschiedlichsten Anwendungen genutzt werden können. Wikipedia listet derzeit fast 40 Frameworks.

Ganz grob kann man nun folgende Ebenen unterscheiden:

Anwendungsprogramme
Frameworks / Bibliotheken / Plugins
Integrated Development Environment (IDE)
Compiler / Interpreter
Assembler / Virtuelle Maschine
Maschinensprache /Prozessor

3 Strukturierung durch Module, Bibliotheken und Plugins

Komplexe Programme können Millionen von Programmzeilen umfassen. Um dabei noch einen Überblick zu behalten, müssen sie unterteilt und strukturiert werden. Jede Programmiersprache kennt zunächst kleinteilige Unterteilungen, wie z.B. Blöcke, Funktionen, Prozeduren, Methoden, Subroutinen, Klassen.

3.1 Module

Größere Einheiten bilden *Packages* oder *Module*. Diese Einheiten stellen meist eine Namenskonvention zur Verfügung, mit der man Namen in verschiedenen Einheiten auseinanderhalten kann. Typisch ist ein sog. *Namespace* Konzept. Hat man z.B. einen Modul/Package `Datenbank` und einen Modul/Package `Gui`, dann kann man in beiden eine Funktion wie z.B. `clear` haben. Innerhalb des eigenen Moduls/Packages kann man dann einfach den Namen `clear` benutzen. Von außen lassen sie sich unterscheiden durch `Datenbank:clear` und `Gui:clear`.

Module/Packages lassen sich meist auch schachteln. Z.B. innerhalb des Moduls/Packages `Gui` könnte es z.B. Untermodule/packages geben: `Gui:Window` und `Gui:Buttons`. Solche Schachtelungen kann es in beliebiger Tiefe geben.

3.2 Bibliotheken

Eine Menge von Modulen, die sich als nützlich für mehr als eine Anwendung erweisen, werden oft in eine *Bibliothek* zusammengefasst. Meist wird sie compiliert und nur der Binärcode, wenn möglich zusammen mit der Dokumentation, ausgeliefert. Die Bibliothek kann dann in einer anderen Anwendung

mitbenutzt werden, indem ihre Funktionen direkt im Programm aufgerufen werden. Das Programm selbst wird dann zusammen mit den verwendeten Bibliotheken vom sog. *Linker* zu einem lauffähigen Programm zusammengebunden.

In vielen Programmiersprachen hat man eine ganz klare Trennung zwischen der Programmiersprache selbst, und den Funktionen, die eine Bibliothek zur Verfügung stellt. Es gibt jedoch Programmiersprachen, in denen die Unterscheidung zwischen der „nackten" Programmiersprache und der Verwendung von Bibliotheken verschwimmt. In C++ z.B. kann man *Operatoren überladen*. Es gibt dort z.B. den „Operator" +, mit dem zwei Zahlen addiert werden. Wenn in `x = y + z`, die Variablen `y` und `z` an Integerzahlen gebunden sind, dann werden diese als Integer addiert und das Ergebnis der Variablen `z` zugewiesen. Man könnte aber auch eine Matrix-Bibliothek hinzuladen, in welcher der Operator + auch für die Addition von zwei Matrizen definiert wird. Jetzt dürfen in `x = y + z` die Variablen auch Matrizen sein, und es wird die Matrixaddition ausgeführt.

3.3 Plugins

Dies sind Bibliotheken, die von Programmen zur *Laufzeit* hinzugebunden werden können. Im Gegensatz zu normalen Bibliotheken, bei denen der Programmierer bei der Programmentwicklung schon weiß, welche Bibliotheksfunktionen es gibt, und wie sie aufgerufen werden müssen, ist das dem Programmierer bei Plugins nicht zugänglich. Daher muss die Programmiersprache, der Compiler und das Laufzeitsystem auf die Verwendung von Plugins vorbereitet sein. Dazu vereinbart man i.A. *Schnittstellen*, die standardisierte Aufrufmechanismen vorschreiben, so dass der Programmierer *anonyme* Aufrufe programmieren kann, die dann von allen Plugins, die sich an den Standard halten, ausgeführt werden. Im Extremfall besteht das eigentliche Programm nur aus einer Art Rahmen. Die tatsächliche Funktionalität wird durch das Hinzuladen von Plugins erreicht. Dadurch erreicht man enorme Flexibilität, um ein Programm wechselnden Bedürfnissen des Benutzers anzupassen.

3.4 Schnittstellen/Interfaces

Die Aufteilung von großen Programmen in kleinere Komponenten, sei es Module, Bibliotheken, Plugins oder auch noch kleinteiliger dient neben der Verbesserung der Übersichtlichkeit ganz besonders auch der Möglichkeit, einzelne Komponenten auszutauschen, z.B. gegen neuere und bessere Versionen. Damit das Programm, welches eine solche Komponente benötigt, bei deren Austausch nicht auch noch umprogrammiert werden muss, muss man standardisierte Schnittstellen/Interfaces vereinbaren. Darin wird geregelt, welche Funktionen wie aufzurufen sind. Solange sich die ausgetauschte Komponente an diese Konvention hält, hat der Austausch keine Auswirkung auf den Rest des Programms.

Viele Programmiersprachen enthalten die Mittel, um solche Schnittstellen formal zu definieren, so dass auch ein Compiler überprüfen kann, ob die Programme sich daran halten.

4 Programmierparadigmen

Programme in Maschinensprache werden nacheinander Befehl für Befehl abgearbeitet. Nur ab und zu gibt es Sprünge an andere Stellen im Programm. Bei ganz einfachen Algorithmen ist das eine intuitive und übersichtliche Vorgehensweise. Sobald die Programme aber komplizierter werden, wird das sehr schnell unübersichtlich. Für Hochsprachen hat man daher andere Vorgehensweisen entwickelt, und diese in verschiedenen Sprachen umgesetzt. Moderne Sprachen sind allerdings nicht mehr für genau ein Programmierparadigma vorgesehen. Stattdessen wird es dem Programmierer überlassen, mit einem konkreten Programmierparadigma zu arbeiten, oder mehrere Paradigmen zu mischen.

4.1 Imperatives Programmieren

Dieses Paradigma ist der Arbeitsweise der Maschinensprache am nächsten: ein Programm besteht aus einer Folge von Befehlen, die nacheinander abgearbeitet werden. Ab und zu gibt es Sprünge an andere Stellen im Programm. Die Sprünge werden entweder explizit mit einer `goto`-Anweisung programmiert, wie z.B. in der Sprache FORTRAN, oder implizit durch `if`- und `while`-Anweisungen, wie in der Programmiersprache C. Explizite Sprungbefehle sind allerdings heute sehr verpönt, da sie sehr schnell zu sehr unübersichtlichen Programmen führen. In vielen Programmiersprachen sind sie daher gar nicht mehr vorgesehen.

Die wichtigsten Anweisungstypen in imperativen Sprachen sind die Zuweisung, die Verzweigung mit `if`, und das Schleifenkonstrukt mit `while`.

4.1.1 Zuweisung

Um die Zuweisung zu verstehen, muss man zunächst das Konzept der *Variable* verstehen: eine Variable steht für einen bestimmten Abschnitt im Hauptspeicher. D.h. man kann eine Variable als symbolischen Namen für einen Speicherplatz verstehen.

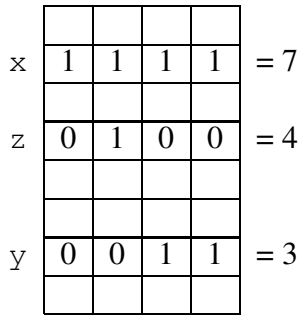
Die einfachste Art der Zuweisung sieht z.B. so aus: `x = 3`.
Sie bewirkt, dass in die Speicherzelle mit dem Namen `x` die Zahl 3 gespeichert wird.

Eine etwas komplexere Zuweisung ist: `x = y + z`.
Sie bewirkt, dass der Inhalt der Speicherzelle mit dem Namen y addiert wird zu dem Inhalt der Speicherzelle mit dem Namen z, und das Ergebnis in die Speicherzelle mit dem Namen x geschrieben wird.

Nach Abarbeitung von

```
y = 3;
z = 4;
x = y + z;
```

hätte die Speicherzelle mit dem Namen `x` den Wert 7.
Um das konkreter zu illustrieren, betrachten wir einen Hauptspeicher, wo die Speicherzellen 4 Bit breit sind. Die Zellen für `x, y` und `z` können an beliebiger Stelle im Speicher liegen.
Es könnte z.B. so aussehen,



wobei die leer gelassenen Zellen mit anderen Bits gefüllt wären. Bei modernen Computern sind die Speicherzellen allerdings länger als 4 Bits breit. Typischerweise sind sie 32 oder 64 Bits breit. Wo genau die Zellen für `x, y` und `z` liegen wird vom Compiler zusammen mit dem Betriebssystem festgelegt. Das ist i.A. bei jedem Start des Programms an anderen Stellen. Zum Glück muss sich der Programmierer nicht um diese Details kümmern.

Eine Anweisung wie `x = x + 1`, welche mathematisch einfach ein Widerspruch wäre, ist, als Zuweisung gelesen, ganz einfach zu verstehen: Erhöhe den Wert an der Speicherstelle mit dem Namen `x` um 1.

Gültigkeitsbereich von Variablen (Skoping)
Nachdem wir das Konzept von Variablen oben eingeführt haben, bleibt noch ein wichtiger Aspekt zu erwähnen: an welchen Stellen nach dem ersten Auftreten einer Variable, d.h. nach der *Variablendeklaration*, darf man die Variable überhaupt benutzen?

In vielen Programmiersprachen kann man Programmstücke in *Blöcke* aufteilen, die meist durch Klammern gekennzeichnet wurden. Die Blöcke können auch ineinander geschachtelt werden. Dann könnte man z.B. schreiben:

```
{
 x = 3;
  {y = x * x;}
}
```

Jetzt gilt das `x` innerhalb des äußeren Blocks sowie den Blöcken, die in diesem Block enthalten sind. Das `y` gilt nur innerhalb des inneren Blocks.

Eine Anweisung wie

```
{
 x = 3;
  {y = x * x;}
 z = y * y;
}
```

wäre dann fehlerhaft, weil das `y` für das `z` nicht mehr verfügbar ist. Den Gültigkeitsbereich einer Variable nennt man den *Skopus* der Variable.

Die meisten Programmiersprachen haben *statisches Scoping*, d.h. der Gültigkeitsbereich einer Variable orientiert sich am Programmtext, insbesondere an der Verschachtelung von Blöcken.

Es gibt aber auch Programmiersprachen mit *dynamischem Scoping*. Das bedeutet: wenn nach einer Variablendeklaration in einem Block ein Unterprogramm im selben Block aufgerufen wird, dann ist die Variable auch in dem Unterprogramm noch verfügbar. Die Nutzung von dynamischem Scoping kann allerdings zu sehr unübersichtlichen Programmen führen, wo Variablen benutzt werden, denen man gar nicht ansieht, wo sie herkommen. Daher wird heute eher davon abgeraten.

4.1.2 Verzweigung mit der `if`-Anweisung

Nur ganz einfache Algorithmen kommen mit Abfolgen von Zuweisungen aus. In viele Fällen muss man Alternativen vorsehen, deren Auswahl von den Daten abhängen. Will man z.B. `y = |x|` berechnen, d.h. `y` soll der Absolutwert von `x` sein, könnte man das so schreiben:

$$\texttt{if}\ \underbrace{\texttt{(x < 0)}}_{\text{Bedingung}}\ \texttt{then}\ \underbrace{\texttt{\{y = -x\}}}_{\text{then-Teil}}\ \texttt{else}\ \underbrace{\texttt{\{y = x\}}}_{\text{else-Teil}}$$

Die `if`-Anweisung wird folgendermaßen ausgewertet:
- Die Bedingung wird ausgewertet und muss einen Wahrheitswert ergeben: `wahr` oder `falsch`.
- Falls die Bedingung wahr wird, wird der then-Teil ausgeführt, und der else-Teil ignoriert.
- Falls die Bedingung falsch wird, wird der else-Teil ausgeführt, und der then-Teil ignoriert.

Manche Programmiersprachen erlauben auch folgende abkürzende Schreibweise:

```
y = (x < 0) ? -x : x
```

Die Wirkung ist die gleiche wie bei der `if`-Version oben.

4.1.3 Schleifen mit der `while`-Anweisung

Eine der großen Stärken von Computern ist die Möglichkeit, dieselbe Folge von Aktionen *beliebig oft* und sehr schnell ausführen zu können. Schleifen, die oft wiederholt werden sollen, programmiert man mit der `while`-Anweisung. Ein Beispiel wäre die Fakultätsberechnung: $n! = 1 \cdot \ldots \cdot n$.

Das könnte man so programmieren:

```
fakultaet = 1;
i = 1;
while(i <= n) {
    fakultaet = fakultaet * i;
    i = i+1;}
```

Abstrakt sieht die while-Schleife so aus:

```
while(Bedingung){Anweisungsfolge}
```

Ausgewertet wird sie, indem die Anweisungsfolge so oft ausgeführt wird, bis die Bedingung falsch geworden ist. Im Fakultätsbeispiel wird die Bedinung falsch sobald `i > n` geworden ist. Dann bricht

die Schleife ab.

Manche Programmiersprachen bieten noch andere Schleifenkonstrukte an. Z.B. könnte man die Fakultätsberechnung auch so programmieren:

```
fakultaet = 1;
for(i = 1;i <= n; i = i+1) {
    fakultaet = fakultaet * i;}
```

Hier wird ein Schleifenzähler, das `i` explizit eingeführt, mit 1 initialisiert, und mit `i = i+1` hochgezählt, solange bis die Bedingung `i <= n` falsch wird.

Alle Schleifenkonstrukte in den verschiedenen Programmiersprachen könnten im Prinzip durch eine `while`-Schleife ersetzt werden, sind aber oft einfacher zu benutzen als das `while`-Konstrukt.

Man kann theoretisch zeigen, dass mit der Zuweisung, der `if`-Anweisung und der `while`-Schleife alles, was überhaupt berechnet werden kann, auch programmiert werden kann. Das würden aber extrem lange und unleserliche Programme. Daher hat man in den verschiedenen Programmiersprachen noch viele weitere Anweisungstypen eingeführt.

4.1.4 Unterprogramme

Wir haben oben gesehen, wie man die Fakultätsfunktion berechnen kann. Wenn man allerdings an fünf verschiedenen Stellen die Fakultätsfunktion berechnen muss, dann möchte man natürlich nicht fünfmal das Schleifenkonstrukt hinschreiben. Man möchte es einmal als Funktion programmieren, und dann an den fünf verschiedenen Stellen aufrufen, z.B. so oder so ähnlich:

```
Fakultaet(n) {
   fakultaet = 1;
   i = 1;
   while(i <= n) {
      fakultaet = fakultaet * i;
      i = i+1;}
   return fakultaet;}
```

Jetzt könnte man an einer Stelle `Fakultaet(5)` und an einer anderen Stelle `Fakultaet(10)` aufrufen.

Selbst Maschinensprachen erlauben die Definition und den Aufruf von solchen Unterprogrammen. Daher geht das natürlich auch in jeder höheren Programmiersprache. Dort heißen sie dann Funktion, Unterprogramm, Subroutine, Methode usw. Es ist aber im Prinzip immer das gleiche: Codestücke werden so ausgezeichnet, dass man sie mit verschiedenen Argumenten beliebig oft von beliebigen Stellen des Programms aus aufrufen kann.

4.1.5 Innere Zustände von Unterprogrammen

Rein mathematische Funktionen, wie insbesondere die Fakultätsfunktion, liefern für das gleiche Argument auch immer wieder den gleichen Wert zurück. Es gibt jedoch auch Anwendungen, wo sich frühere Aufrufe des Unterprogramms auf spätere Aufrufe auswirken, und daher u.U. trotz gleicher Argumente später einen anderen Wert liefern sollen. Das typische Beispiel dafür ist ein Zufallsgenerator, der beliebig oft aufgerufen werden soll, und immer wieder eine andere Zahl liefern soll. So ein Unterprogramm muss sich intern Daten merken können, um beim nächsten Aufruf anders reagieren zu können als bei früheren Aufrufen.[1]

In imperativen Programmiersprachen ist es sehr leicht, sog. *globale Variablen* zu definieren, die während des ganzen Programmlaufs verfügbar sind. Damit lassen sich *innere Zustände* von Unterprogrammen für spätere Wiederverwendung speichern. In rein funktionalen Sprachen wie Haskell geht das nicht direkt. Stattdessen muss man den Zustand in ein Argument packen, welches, verändert, als Teil des Ergebnisses zurück geliefert wird, und dann beim nächsten Aufruf eine andere Reaktion erzeugen kann.

Imperatives Programmieren zusammen mit der Zerlegung eines Programmes in Unterprogramme/Prozeduren wird auch als *Prozedurales Programmieren* bezeichnet.

4.1.6 Übergabetechniken für Unterprogramme

Für die Übergabe von Parametern an Unterprogramme gibt es verschiedene Möglichkeiten. Der Unterschied zwischen den verschiedenen Techniken macht sich aber erst bemerkbar, wenn im Unterprogramm die Variable, die für die Übergabe der Parameter benutzt wird, verändert wird. Für die Fakultätsfunktion oben spielt das daher keine Rolle. Um die Unterschiede zu demonstrieren, ändern wir die Fakultätsfunktion etwas ab:

```
Fakultaet(n,fakultaet) {
   i = 1;
   while(i <= n) {
      fakultaet = fakultaet * i;
      i = i+1;};}
```

Man könnte diese Funktion jetzt z.B. so aufrufen:

```
fak = 1;
Fakultaet(6,fak)
```

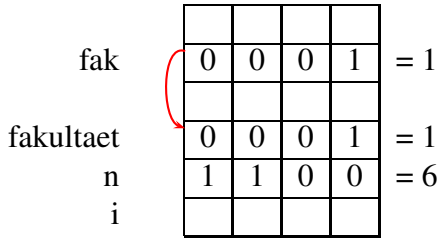
und hoffen, dass nach Beendigung der Funktion die Variable `fak` den Wert 120 hat. Das klappt leider nicht immer. Man muss nämlich unterscheiden:

Call-By-Value:
In diesem Fall wird an das Unterprogramm *eine Kopie* des Werts vom Hauptprogramm übergeben.

[1] Bessere Zufallsgeneratoren bekommen jedoch Informationen von außen, z.B. die aktuelle Prozessortemperatur, aus der sie jedes mal eine Zahl generieren können, ohne sich etwas merken zu müssen.

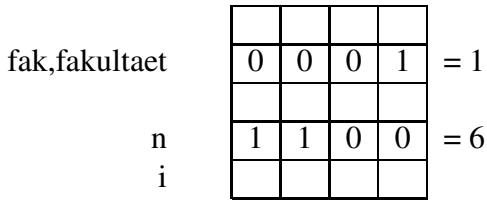
Im obigen Beispiel bedeutet das, dass den Variablen `fak` und `fakultaet` *zwei verschiedene* Speicherplätze zugeordnet werden. Beim Aufruf des Unterprogramms wird der Wert im Speicherplatz für `fak` in den Speicherplatz von `fakultaet` *kopiert*. Das Unterprogramm arbeitet dann nur mit dem Speicherplatz für `fakultaet`, und die 1 im Speicherplatz von `fak` bleibt unverändert.

Die Speicherbelegung bei 4-Bit Worten könnte, direkt nach dem Aufruf von `Fakultaet(6,fak)` so aussehen:



Call-By-Name:
In diesem Fall sorgt das Laufzeitsystem dafür, dass beim Aufruf des Unterprogramms derselbe Speicherplatz wie im Hauptprogramm genommen wird. D.h. im obigen Beispiel, dass für die Variable `fakultaet` genau der Speicherplatz für die Variable `fak` genommen wird. Jetzt hat nach Beendigung des Unterprogramms die Variable fak tatsächlich den Wert 120.

In diesem Fall könnte die Speicherbelegung bei 4-Bit Worten direkt nach dem Aufruf von `Fakultaet(6,fak)` so aussehen:



Call-By-Reference:
In machen Programmiersprachen, insbesondere in C und C++, ermöglicht man beide Übergabemechanismen, Call-By-Value und Call-By-Name. Um sie aber unterscheiden zu können, realisiert man Call-By-Name etwas anders, nämlich als *Call-By-Reference*. Dabei übergibt man nicht den Wert einer Variablen an das Unterprogramm, sondern die Adresse des Speicherplatzes. Im folgenden Unterprogramm bedeutet `*fakultaet`, dass die Adresse eines Speichplatzes übergeben wird. In `*fakultaet = *fakultaet * i` bedeutet der `*fakultaet`, dass jetzt der Speicherplatz selbst genommen werden soll.

```
Fakultaet(n,*fakultaet) {
    i = 1;
    while(i <= n) {
        *fakultaet = *fakultaet * i;
        i = i+1;};}
```

Im Aufruf muss man dann statt der Variablen `fak` selbst, ihre Adresse `&fak` übergeben.

```
fak = 1;
Fakultaet(6,&fak)
```

Die Speicherbelegung bei 4-Bit Worten könnte, direkt nach dem Aufruf von `Fakultaet(6,fak)` jetzt so aussehen, wobei der Speicherplatz von `*fakultaet` die *Adresse* des Speicherplatzes der `fak`-Variable beinhaltet:

	5					
fak	4	0	0	0	1	= 1
	3					
*fakultaet	2	0	1	0	0	= 4
n	1	1	1	0	0	= 6
i	0					

4.2 Funktionales Programmieren

Viele Programmieraufgaben laufen darauf hinaus, *Funktionen* zu implementieren, so wie man sie aus der Mathematik kennt: man ruft sie mit irgendwelchen Argumenten auf, und sie liefern ein Ergebnis zurück. Wenn sie mit denselben Argumenten noch einmal aufgerufen werden, dann liefern sie auch wieder dasselbe Ergebnis zurück.

Funktionale Programmiersprachen realisieren genau dieses Paradigma: anstelle von Sequenzen von Anweisungen schreibt man ineinander geschachtelte Funktionsaufrufe. Ein Beispiel ist wieder die Fakultätsfunktion, diesmal funktional rekursiv implementiert:

```
fakultaet(n) = (n == 1) ? 1 : fakultaet(n-1)*n
```

oder in extrem funktionaler Präfix-Schreibweise:

```
fakultaet(n) = if(==(n,1), 1,*(fakultaet(n-1),n))
```

Die Funktion == testet, ob die beiden Argument gleich sind.
Die `if`-Funktion ist die funktionale Version des `if-then-else` Konstrukts.

Die rekursive Implementierung nutzt den Zusammenhang: $n! = (n-1)! \cdot n$.

Solcherart Funktionen können natürlich keine internen Zustände haben. Will man daher z.B. einen Zufallsgenerator funktional implementieren, dann muss man den Zustand als extra Argument kodieren, welches, verändert, zusammen mit der Zufallszahl als Ergebniswert zurück geliefert wird, und daher bei einem späteren Aufruf eine andere Reaktion der Funktion erzeugen kann.

4.2.1 Variablen und Aufrufmechanismen

Variablen in imperativen Sprachen sind nur Namen für Speicherplätze. In funktionalen Sprachen können Variablen dagegen auch an Terme gebunden werden, die erst später ausgewertet werden. Um das zu illustrieren betrachten wir folgende Funktion:

```
quadrat(x) = x * x
```

Für die Auswertung von `quadrat(3)` wird `x` an die Zahl 3 gebunden, so dass aus dem Funktionsrumpf `3 * 3` wird. Das kann jetzt auf die erwartete Weise ausgewertet: mit Ergebnis 9.

Jetzt probieren wir `quadrat(quadrat(3))`

Um das auszuwerten gibt es verschiedene Möglichkeiten:

inside-out: Hierbei werden die inneren Terme zuerst ausgewertet:

`quadrat(quadrat(3))` → `quadrat(3*3)` → `quadrat(9)` → `9*9` → `81`

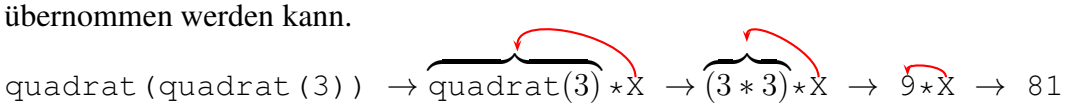
outside-in: Jetzt werden die Terme von außen nach innen ausgewertet:

`quadrat(quadrat(3))` → `quadrat(3)*quadrat(3)` →
`(3*3)*(3*3)` → `9*9` → `81`

Ein Nachteil ist offensichtlich: der Term (3*3) wird zweimal ausgewertet.

Eine Optimierung ist die

verzögerte Auswertung: Bei Mehrfachvorkommen von Variablen werden die hinteren Vorkommen durch Zeiger auf das erste Vorkommen ergänzt, so dass das Ergebnis von der ersten Auswertung übernommen werden kann.



Die funktionalen Auswertungsstrategien haben Optimierungsmöglichkeiten, die bei imperativen Programmiersprachen nicht möglich sind. Um das zu illustrieren, betrachten wir folgende rekursive Definition der Multiplikation (die man natürlich so nicht zu machen braucht):

$$X * 0 = 0$$
$$0 * X = 0$$
$$X * Y = X + X*(Y-1)$$

Jetzt könnte man z.B. auswerten: `0 * Fakultaet(1000)`. In jeder imperative Sprache würde auf jeden Fall völlig unnötigerweise `Fakultaet(1000)` ausgerechnet werden. In einer funktionalen Sprache dagegen erhält man mit der obigen Definition direkt: `0 * Fakultaet(1000)` → `0`, ohne dass `Fakultaet(1000)` überhaupt angefasst werden muss. Sogar wenn `Fakultaet(1000)` nicht einmal terminieren würde, wäre das Ergebnis 0.

Die wohl erste funktionale Sprache wurde schon 1959 von John McCarthy entwickelt: **Lisp**. Lisp war lange die Sprache der Wahl für Programme im Bereich Künstliche Intelligenz. Derzeit wird dagegen die Sprache **Haskell** als funktionale Sprache favorisiert.

Die meisten als funktional charakterisierte Programmiersprachen sind jedoch nicht rein funktional, sondern haben auch imperative Aspekte, insbesondere bei der Verwendung von Variablen als Namen für Speicherplätze.

4.3 Logisches Programmieren

Logische Programmiersprachen, allen voran die Sprache **Prolog**, haben eine ganz andere Herangehensweise als die anderen Programmiersprachen. Wie der Name schon sagt, basieren Sie auf Logik. Sie rechnen nicht, sondern sie ziehen Schlussfolgerungen.

Die Bestandteile eines logischen Programms sind *Fakten* und *Regeln*. Fakten könnten z.B. sein:

```
1 Vater(tom, karl).
2 Mutter(tom, maria).
3 Vater(karl, wilhelm).
4 Mutter(karl, eva).
```

mit der offensichtlichen Bedeutung. `tom, karl` usw. sind dabei *Konstantensymbole*[2].

Regeln könnten sein:

```
1 Elter(X,Y)     :- Vater(X,Y).
2 Elter(X,Y)     :- Mutter(X,Y).
3 Vorfahre(X,Y)  :- Elter(X,Y).
4 Vorfahre(X,Y)  :- Elter(X,Z), Vorfahre(Z,Y).
```

`X, Y` und `Z` sind *Variablen*. „`:-`“ ist als Rückwärtsimplikation zu lesen, also z.B.
wenn Y Vater von X ist, dann ist auch Y Elter von X.

Die letzte Regel ist zu lesen:
wenn Z Elter von X ist, **und** Y Vorfahre von Z, dann ist auch Y Vorfahre von X.
Das Komma steht also für das logische und.

Es gibt jetzt zwei verschiedene Auswertungsstrategien: vorwärts wie in der Sprache **Datalog** und rückwärts wie in der Sprache **Prolog**.

Vorwärtsauswertung im Beispiel: Hierbei werden die Regeln von rechts nach links so lange auf die Fakten angewendet, bis keine neuen Fakten mehr hergeleitet werden können. Im Beispiel erhält man mit den ersten beiden Regeln die neuen Fakten:

```
Elter(tom,karl). Elter(tom,maria).
Elter(karl,wilhelm). Elter(karl,eva).
```

Mit Regel 3 bekommt man dann:

```
Vorfahre(tom,karl). Vorfahre(tom,maria).
Vorfahre(karl,wilhelm). Vorfahre(karl,eva).
```

Und schließlich mit Regel 4 aus den abgeleiteten Fakten noch die Großeltern:

```
Vorfahre(tom,wilhelm). Vorfahre(tom,eva).
```

Rückwärtsauswertung am Beispiel: Während Vorwärtsauswertung einfach alles ableitbare auch ableitet, geht Rückwärtsauswertung von einer *Anfrage* aus, und versucht, rückwärts, aus den Regeln und Fakten Antworten auf die Anfrage zu finden. Zur besseren Übersicht ist hier nochmal das Programm aufgelistet.

```
1 Vater(tom, karl).
```

[2]Klein geschriebene Namen ist Konstanten, groß geschriebene Namen sind Variablen.

```
2 Mutter(tom,maria).
3 Vater(karl,wilhelm).
4 Mutter(karl,eva).
5
6 Elter(X,Y)     :- Vater(X,Y).
7 Elter(X,Y)     :- Mutter(X,Y).
8 Vorfahre(X,Y) :- Elter(X,Y).
9 Vorfahre(X,Y) :- Elter(X,Z), Vorfahre(Z,Y).
```

Beispiel: wir fragen `Vorfahre(tom,V)` (welche Vorfahren hat tom?). Die Sprache Prolog geht jetzt die Fakten und Regeln von oben nach unten durch und sucht nach passenden Fakten oder Regeln. Die erste Regel, die passt ist die Regel 8.

`Vorfahre(tom,V)` wird mit `Vorfahre(X,Y)` *unifiziert*, d.h. durch Einsetzungen der Variablen gleich gemacht: `X` $\rightarrow$ `tom`, `Y` $\rightarrow$ `V`. Um `Vorfahre(tom,V)` abzuleiten, muss man nach dieser Regel also `Elter(tom,V)` herleiten. Das geht mit den Regeln 6 und 7. Dabei ergibt sich die Notwendigkeit, `Vater(tom,V)` und/oder `Mutter(tom,V)` herzuleiten. Aus Fakt 1 ergibt sich `V = karl` und aus Fakt 2: `V = maria`.

Jetzt kennt das System schon zwei Eltern und Vorfahren von tom: karl und maria.

Es gibt aber noch die Regel 9, aus der sich noch weitere Vorfahren herleiten lassen. Dazu braucht man zunächst `Elter(tom,V)`. Das sind wiederum `Z = karl` und `Z = maria`, und von denen braucht man wiederum Vorfahren. Auf analoge Weise kann man `Vorfahre(karl,wilhelm)` und `Vorfahre(karl,eva)` ableiten.

Damit ergibt sich `Vorfahre(tom,wilhelm)` und `Vorfahre(tom,eva)`.

Die Abfolge kann folgendermaßen etwas übersichtlicher dargestellt werden:

$$
\begin{aligned}
\texttt{Vorfahre(tom,V)} &\xrightarrow{8} \texttt{Elter(tom,V)} \xrightarrow{6} \texttt{Vater(tom,V)} \xrightarrow{1} \texttt{V = karl} \\
&\qquad\qquad\qquad\quad \xrightarrow{7} \texttt{Mutter(tom,V)} \xrightarrow{2} \texttt{V = maria} \\
&\xrightarrow{9} \texttt{Elter(tom,Z), Vorfahre(Z,V)} \\
&\xrightarrow[\texttt{Elter(tom,karl)}]{} \texttt{Vorfahre(karl,V)} \\
&\xrightarrow{8} \texttt{Elter(karl,V)} \xrightarrow{6} \texttt{Vater(karl,V)} \xrightarrow{3} \texttt{V = wilhelm} \\
&\qquad\qquad\qquad\quad \xrightarrow{7} \texttt{Mutter(karl,V)} \xrightarrow{4} \texttt{V = eva}
\end{aligned}
$$

Die Alternativen werden aber nur bei Sackgassen oder mit expliziter Aufforderung verfolgt (Backtracking).

4.4 Objektorientiertes Programmieren

Die ersten Anwendungen von Computern waren fast nur numerische Programme. Als Datentypen genügten die Zahlentypen, Integer und Floatingpoint Zahlen. Sobald man aber nicht-numerische Probleme hat, werden die Datentypen komplexer. Um z.B. eine Personenliste zu verwalten, braucht man einen Typ Person, mit Attributen wie Name, Adresse, Geburtsdatum, Beruf usw. Eine Adresse besteht

aus Straßenname, Hausnummer, Postleitzahl, Ort. Ein Geburtsdatum besteht aus Tag, Monat, Jahr.

Die Personenliste kann nur vernünftig organisiert werden, wenn man die Attribute, die zu einer Person gehören, zusammenfassen und mit *einer* Adresse ansprechen kann. Genau das ist die Motivation für die Einführung von objektorientierten Sprachen.

Unter Benutzung der vordefinierten Typen `String` und `Integer` könnte man z.B. Person und Adresse und Geburtstag so definieren.

```
class Person {          class Adresse {          class Datum {
  String Name             String Strassenname      Integer Tag
  Adresse Adresse         Integer Hausnummer       Integer Monat
  Datum Geburtstag        Integer Postleitzahl     Integer Jahr}
  String Beruf}           String Ort}
```

Jetzt könnte man konkrete Objekte erzeugen:

```
tomsGeburtstag = new Datum(1,1,2000)
tomsAddresse   = new Adresse(Hautstrasse,14,1111,Tomdorf)
tom = new Person(Tom,tomsAdresse,tomsGeburtstag,Student)
```

Die Wirkung des `new`-Operators ist, zunächst mal geeignet viel Speicherplatz zu reservieren, und dann die Daten in den Speicherplatz zu schreiben.

Anschließend kann man auf die Daten wieder zugreifen, z.B. `tom.Geburtsdatum.Jahr` ergibt die Zahl 2000.

Vererbung

Ein weitere wichtige Komponente von objektorientierten Programmiersprachen ist die *Vererbung*. Nachdem man die Klasse `Person` definiert hat, möchte man z.B. die Klasse `Student` definieren. Jeder Student ist eine Person, hat aber noch weitere Attribute, z.B. Matrikelnummer. Das könnte z.B. so gehen:

```
class Student extends Person {
   Integer Matrikelnummer
   String Studienfach
   Integer Semester}
```

Jetzt hat ein `Student`-Objekt auch alle Attribute der `Klasse` Person.

`Person` ist die *Oberklasse* und `Student` ist die *Unterklasse*. In Programmiersprachen wie Java kann jede Klasse maximal eine Oberklasse haben (*Monovererbung*). In C++ z.B. kann jede Klasse mehrere Oberklassen haben (*Multivererbung*). Das klassische Beispiel für Multivererbung ist das Amphibienfahrzeug, welches Unterklasse von Schiff und Landfahrzeug ist.

Methoden
In jeder objektorientierten Programmiersprache kann man zu den Klassendefinitionen noch *Methodendefinition* hinzufügen. Methoden sind Funktionen, die für ein konkretes Objekt aufgerufen werden, und auf die Attribute dieses Objekts zugreifen können.

Im Personen-Beispiel könnte man z.B. eine Methode `Alter` definieren, die das Alter einer Person berechnet.

Sie könnte z.B. so aussehen:

```
class Person {
  String   Name
  Adresse  Adresse
  Datum    Geburtsdatum
  String   Beruf

  Integer  Alter(Datum heute) {return Geburtsdatum.dauer(heute)}
  }

class Datum {
  Integer Tag
  Integer Monat
  Integer Jahr}

  Integer dauer(Datum heute) {return heute.Jahr - Jahr}
```

Mit

```
jetzt = new Datum(27,3,2018}
tom.Alter(jetzt)
```

erhält man das Alter 18.

Methoden unterscheiden sich von den üblichen Funktionen in folgenden Aspekten:

- das Objekt, für das sie aufgerufen werden, ist nicht ein Argument der Methode, sondern wird üblicherweise mit `objekt.methode(...)` aufgerufen.
- Die Methode hat Zugriff auf alle Attribute des Objekts. Daher ist das Objekt ein ausgezeichnetes Argument der Methode, welches die besondere Aufrufsyntax rechtfertigt.

Objektorientierte Sprachen mit Vererbung, erlauben auch, in den sog. Unterklassen Methoden, die in der Oberklasse definiert wurden, in der Unterklasse neu zu definieren.

Als Beispiel: wir erweitern die Oberklasse `Person` mit der Unterklasse `Amtsträger`, die noch ein Attribut `Titel` hat. Angenommen, die Oberklasse hat eine Methode `druckeAdresse`, die Namen und Adresse druckt. Für die Unterklasse `Amtsträger` möchte man aber noch den Titel in dem Namen drucken. Jetzt braucht man eine veränderte Methode `druckeAdresse`, die den Titel noch mitdruckt. Diese kann man einfach in der Unterklasse definieren. Für ein konkretes Objekt, wie oben

`tom` z.B. wird dann beim Aufruf von `tom.druckeAdresse()` die Methode genommen, die zur Klasse von `tom` gehört. Wenn `tom` ein Amtsträger ist, wird die Methode der Unterklasse genommen, wenn er eine „einfache“ Person ist, die Methode der Oberklasse.

4.5 Aspektorientiertes Programmieren

Für imperatives, funktionales, logisches und objektorientiertes Programmieren gibt es jeweils Programmiersprachen, die genau auf dieses Programmierparadigma abgestimmt sind, obwohl darin oft auch die anderen Programmierpardigmen möglich sind.

Aspektorientiertes Programmieren dagegen kann man nicht einer konkreten Programmiersprache zuordnen, sondern es ist eine Methodik, die für eine konkrete Sprache einen Zusatzmechanismus zur Verfügung stellt, den man für bestimmte Anwendungen nutzen kann.

Die Idee ist folgende:
Angenommen man hat ein Programm geschrieben, welches eine komplexe Berechnung macht und lange rechnet. Während der Entwicklung möchte man z.B. eine Monitormöglichkeit schaffen, die an ganz bestimmten Punkten im Programm Daten abgreift und auf einem Bildschirm anzeigt. Nachdem das Programm ausgetestet ist, ist die Monitormöglichkeit nicht mehr nötig.

Mit der „klassischen“ Programmiermethodik, müsste man das Programm abändern, um diese Monitormöglichkeit einzubauen. Anschließend müsste man die Änderungen wieder rückgängig machen. Mit aspektorientiertem Programmieren spezifiziert man die Änderungen völlig getrennt vom eigentlichen Programm. Ein spezieller Compiler baut dann *vollautomatisch* die Änderungen an den betroffenen Stellen in den compilierten Code des Programms ein. Ruft man dann den normalen Compiler wieder auf, enthält das compilierte Programm keine Änderungen mehr.

Für Java gibt es dafür das System AspectJ.

4.6 Nebenläufiges Programmieren

Das bezeichnet die Möglichkeit, Programme zu *parallelisieren*. Moderne Prozessoren können kaum noch verbessert werden, indem man ihre Taktfrequenz erhöht. Stattdessen baut man immer mehr Rechenkerne in einen Prozessor ein. Die Rechenkerne rechnen parallel. Wenn man also einen Algorithmus parallelisieren kann und auf z.B. 8 Rechenkerne verteilt, dann kann man den Algorithmus im Prinzip um den Faktor 8 beschleunigen. Da inzwischen jedes Notebook, und sogar die Handys mehr als einen Rechenkern haben, hat ein parallelisiertes schnelleres Programm einen Marktvorteil gegenüber einem seriellen Programm, welches dann nur einen Bruchteil der Rechenkapazität nutzt.

Um ein Programm zu parallelisieren gibt, es im Prinzip zwei Möglichkeiten:

- Man erzeugt mehrere separate Prozesse, die z.B. mit Pipes miteinander kommunizieren. Das geht praktisch mit allen Programmiersprachen.

- Die Programmiersprache stellt sog. *Threads* zur Verfügung. Ein Thread ist ein separater Durchlauf durch ein Programm. Mehrere Threads rechnen also gleichzeitig mit demselben Programm, aber an verschiedenen Stellen, und auch mit eigenen Daten.

Ein ganz wichtiges Einsatzgebiet für nebenläufiges Programmieren sind Serverprogramme. Ein Serverprogramm, z.B. ein Suchserver wie bei Google, *muss* nebenläufig sein. Wenn er das nicht wäre, müsste die Bearbeitung einer Anfrage warten, bis die vorherige Anfrage fertig wäre. Das würde zu unerträglichen Wartezeiten führen. Ein nebenläufiger Server startet stattdessen für jede Anfrage einen eigenen Thread oder Prozess. Die können dann die Anfragen parallel bearbeiten.

Moderne Supercomputer haben teilweise hunderttausende von Prozessoren. Diese können nur sinnvoll genutzt werden, wenn die Programme auf massiven Parallelismus ausgelegt sind. Das ist daher ein hochaktuelles Forschungsgebiet.

4.7 Constraintprogrammierung

Viele Probleme aus dem Alltag sind *Planungsprobleme*, deren Lösung aus Suchschritten und sog. Constraint-Propagation Schritten bestehen. Um das zu illustrieren, betrachten wir ein Problem, mit dessen Lösung jeder von uns schon leben musste, nämlich einen Stundenplan für eine Schule zu erstellen.

Der arme Lehrer, der das in den Ferien machen muss, hat neben der Anzahl Klassen, der Klassenräume, der Lehrer und dem Stundenraster folgende Bedingungen, sog. *harte Constraints*, die er beachten muss:

- kein Lehrer kann gleichzeitig mehr als eine Unterrichtstunde geben,
- keine Klasse kann gleichzeitig mehr als eine Unterrichtstunde haben,
- in keinem Unterrichtsraum kann gleichzeitig mehr als eine Unterrichtsstunde stattfinden.
- die Klasse 5 hat 3 Stunden Deutsch, 2 Stunden Englisch usw.

Dazu kommen noch sog. *weiche Constraints*

- die Lehrer Maier, Müller, ... haben ein Lehrdeputat von 20 Stunden ($\pm$ 2),
  usw. für die anderen Lehrer
- der Lehrer Maier gibt Deutsch und Englisch (und vielleicht noch Französisch),
  usw. für die anderen Lehrer
- Lehrer Müller will am Freitag ganz frei haben (wenn möglich)
- ...

Für die Stundenplanung könnte der Lehrer vielleicht so anfangen:

- Maier gibt Deutsch in der 5. Klasse.
  Daraus ergeben sich Konsequenzen:
  - alle anderen Lehrer geben nicht Deutsch in der 5. Klasse,
  - Das restliche Deputat von Maier erniedrigt sich um 3 Stunden,
- Die erste Stunde Montags in der 5. Klasse ist Deutsch in Raum 001.
  Daraus ergeben sich wiederum Konsequenzen:
  - Raum 001 ist in der ersten Stunde Montags für alle anderen Klassen gesperrt,
  - Für die 5. Klasse bleiben noch weitere 2 Stunden Deutsch einzuplanen,
  - Die erste Stunde Montags für die 5. Klasse ist blockiert für alle anderen Unterrichtsstunden.
- usw.

Die Planung besteht offensichtlich aus zwei Phasen:
- eine mehr oder weniger willkürliche Entscheidung muss getroffen werden,
  z.B. Maier gibt Deutsch in der 5. Klasse,
- dann müssen alle Konsequenzen dieser Entscheidung berechnet werden,
  damit die weiteren Entscheidungen nicht zu Konflikten führen.

Diese zwei Phasen wechseln sich ab, bis entweder eine Lösung gefunden wurde, oder ein Konflikt auftaucht, der nicht mehr zu reparieren ist. In diesem Fall geht man zurück zum letzten Punkt, wo eine willkürliche Entscheidung getroffen wurde, und probiert eine andere Alternative (*Backtracking*).

Diese Art von Problem sind komplex, kommen aber oft vor. Jedes Speditionsunternehmen muss seine Fahrten planen, jeder Handwerker und jede Firma muss ihre Aufträge planen, sogar jeder Briefträger muss seinen Weg planen. Luftfahrtunternehmen müssen ihre Flüge planen, Roboter müssen Aktionsfolgen planen usw.

Da die Probleme oft vorkommen, das Lösungsverfahren aber im Prinzip immer das gleiche ist: Suchschritte, jeweils gefolgt von Constraint Propagation, und evtl. Backtracking, hat man dafür eine Programmiertechnik entwickelt, mit der man das leichter implementieren kann: *Constraintprogramming*. Der Programmierer spezifiziert die Constraints, und der Rest läuft automatisch.

Realisiert wird das durch Bibliotheken für verschiedene Programmiersprachen, z.B. die clpfd-Bibliothek für Prolog.

5 Typsysteme

Prozessoren arbeiten nur mit Bitfolgen. Ob eine Bitfolge eine Zahl, ein Buchstabe, eine Adresse oder sonst etwas darstellt, ist den Prozessoren selbst völlig egal. Für einen Programmierer macht es aber natürlich einen großen Unterschied, ob der Wert einer Variablen `x` z.B. ein Integerwert, eine Floatingpoint Zahl, ein ASCII-Buchstabe, ein Boolescher Wert, eine Adresse oder noch etwas ganz anderes darstellt. Daher wurden schon sehr früh sog. *Typsysteme* eingeführt. Damit kann man Variablen und Funktionswerte so kennzeichnen, dass der Programmierer, oder zumindest der Compiler erkennen kann, von welchem Typ der Wert einer Variablen ist. Eine typische Variablendeklaration wäre z.B.
`int i = 0`
Damit wird festgelegt, dass die Werte der Variablen `i` nur Integerzahlen sein dürfen.

Typsysteme bieten mehrere nützliche Möglichkeiten:

- Sie machen die Programme übersichtlicher und helfen daher dem Programmierer.
- Sie erlauben schon relativ früh, entweder schon beim Editieren eines Programms, oder spätestens beim Compilieren, Typfehler zu erkennen. Ein Typfehler wäre z.B. einer Integervariablen einen Floatingpoint Wert zuzuweisen.
- Man kann damit *Typkonversionen* automatisieren. Z.B. die Anweisungsfolge:
  ```
  int i = 2;
  float j = i;
  ```
  erfordert, dass die Integerdarstellung des Werts der Variablen `i` in eine Floatingpoint Darstellung für die Variable `j` umgewandelt wird. Das ist eine relativ komplexe Operation, die aber automatisiert werden kann. Ohne die Typinformation ist nicht zu erkennen, dass da überhaupt eine Umwandlung nötig ist.

5.1 Statische Typisierung

Hierbei muss vom Programmierer jede Variable mit ihrem Typ gekennzeichnet werden. Auch für jede Funktion/Methode muss der Typ des Ergebniswertes angegeben werden.

```
int i = 2;
String j = i;
```

würde dann sofort als Typfehler erkannt werden. Schlaue Editoren erkennen das dann schon beim Eintippen.

Bei

```
int i = 2;
float j = i;
```

könnte der Compiler eine Typumwandlung einbauen, die bei

```
int i = 2;
int j = i;
```

nicht notwendig wäre.

Die Programmiersprache Java ist statisch typisiert, was von machen Programmierern als Einschränkung ihrer Freiheit empfunden wird. Es hilft aber, übersichtliche und nachvollziehbare Programme zu schreiben, und Fehler frühzeitig zu erkennen.

5.2 Dynamische Typisierung

In diesem System tragen die Daten selbst die Typinformation. Der Wert einer Integervariablen, z.B. bestünde aus dem eigentlichen Zahlenwert und einer Kennziffer für Integer. Der vom Compiler erzeugte Code muss daher *zur Laufzeit* die notwendigen Typüberprüfungen machen. Während der Programmierung kann man Typfehler kaum erkennen. Eine Anweisungsfolge

```
i = 2;
i = "Karl";
```

wo der Wert der Variablen erst eine Zahl, und dann ein String ist, ist dabei völlig legal, und möglicherweise sogar gewollt.

```
i = 2;
j = i + 2.5;
```

ist auch legal, da man zur Laufzeit erkennen kann, dass, hier ein Integerwert zu einem Floatingpoint Wert zu addieren ist, und daher der Integerwert vorher umgewandelt werden muss (und kann). Bei

```
i = 2;
j = "Karl";
k = i + j;
```

wird es schon unklarer. Einen Integerwert und einen String zu addieren geht eigentlich nicht. Das sollte eine Fehlermeldung produzieren. Manche Programmiersprachen machen es aber trotzdem, indem der Integerwert zuerst in die Dezimaldarstellung als String umgewandelt wird und die beiden Strings dann aneinander gehängt werden. Der Wert von `j` wäre dann "2Karl".

Eine der ersten funktionalen Programmiersprachen, LISP, hatte dynamische Typisierung.

Dynamische Typisierung gibt dem Programmierer mehr Flexibilität in der Nutzung von Variablen, hat aber den Nachteil, dass man Typfehler, die man sonst schon während der Programmierung erkennen könnte, erst zur Laufzeit erkennt. Je komplexer ein Programm wird, um so nachteiliger wirkt sich das aus.

5.3 Duck-Typing

Dies ist ein Typsystem für objektorientierte Programmiersprachen, wo statische und dynamische Typisierung gemischt ist. Dabei richtet sich der Typ eines Objekts nach den Methoden, die für das Objekt verfügbar sind.

Zur Illustration betrachten wir zwei Klassen `Person` und `Auto`.
Angenommen, beide Klassen haben eine Methode `getHöhe()`, die die Größe liefert. Für ein Objekt `x` rufen wir jetzt `x.getHöhe()` auf.
Jetzt ist es egal, ob der Wert von `x` eine Person oder ein Auto ist. In beiden Fällen gibt es die Methode `getHöhe()`, also kann man die auch aufrufen. Es gibt damit keinen Typfehler.

Der Name `Duck-Typing` kommt aus einem Gedicht von James Whitcomb Riley:

When I see a bird that walks like a duck and swims like a duck and quacks like a duck,
I call that bird a duck.

Die Programmiersprache Groovy, eine Erweiterung von Java, sowie auch Skriptsprachen wie JavaScript, Python und Ruby benutzen Duck-Typing. Allerdings hat sich auch hier herausgestellt, dass bei komplexen Programmen immer mehr eigentlich vermeidbare Typfehler entstehen, die oft auch schwierig zu finden sind.

5.4 Typinferenz

Mit statischer Typisierung kann man am schnellsten Typfehler finden. Programmiersprachen mit statischer Typisierung helfen daher ganz besonders bei der Entwicklung korrekter Programme. Allerdings empfinden viele Programmierer den Zwang zur Angabe der Typen als nervig, insbesondere weil es oft ganz klar ist, welchen Typ eine Variable haben muss.

Die nächste Stufe bei der Typisierung von Programmiersprachen ist daher die automatische Berechnung der Typen, da wo es eindeutig ist.

Um das zu illustrieren betrachten wir mal eine funktional- rekursive Definition der Fakultätsfunktion:

```
Fakultaet(n) = if(n == 1) then 1
               else n * Fakultaet(n-1);
```

Unter Ausnutzung folgender Fakten kann man alle Typen automatisch herausfinden:

- Die Zahl 1 ist vom Typ Integer.
- Die Operatoren == (Gleichheit), - (Subtraktion) und * (Multiplikation) arbeiten nur mit Argumenten gleichen Typs.

Wegen 1:Integer und `n == 1`, sowie auch `n-1` muss auch `n` vom Typ Integer sein. Wenn n vom Typ Integer ist, dann muss wegen `n * Fakultaet(n-1)` auch der Ergebniswert von `Fakultaet` vom Typ Integer sein.

Durch eine geeignete Analyse kann man also in vielen Fällen (nicht immer) Typen automatisch herausfinden, und damit den Programmierer entlasten. Funktionale Sprachen wie Haskell haben diese Art von Typinferenz eingebaut.

5.5 Typen und Klassen

Bei objektorientierten Programmiersprachen mit statischer Typisierung können sich die Begriffe Typ und Klasse leicht vermischen. Um das Problem zu illustrieren, betrachten wir die Klasse `Person`, und davon eine Unterklasse `Student`. Jetzt könne man programmieren:

```
Person tom = new Student(...)
```

Die Variable `tom` ist vom *Typ* `Person`. Ihr Wert aber ist ein Objekt der *Klasse* `Student`. Hier sieht man den Unterschied zwischen Typ und Klasse.

Würde man jetzt z.B. `tom.getMatrikelnummer()` aufrufen, wobei `getMatrikelnummer` in der Klasse `Student` definiert ist, dann würde der Compiler eine Fehlermeldung generieren. Da `tom` den Typ `Person` hat, und die Klasse `Person` keine Methode `getMatrikelnummer` hat, ist das bei statischer Typisierung nicht erlaubt. Bei Duck-Typing wäre das dagegen kein Typfehler, und würde auch richtig funktionieren.

6 Ausnahmebehandlung (Exceptions)

Selbst ein kleines Programm, das auf Anhieb unter allen Umständen und für alle möglichen Eingaben immer korrekt funktioniert, ist ein kleines Wunder. Daher muss sich der Programmierer mit der Fehlerbehandlung auseinandersetzen. Man unterschiedet dabei drei Arten von Fehlern:

- Datenfehler; Bsp. der Benutzer gibt für ein Datum den Monat 13 ein,
- Programmierfehler, z.B. Division durch 0,
- Systemfehler, z.B. nicht genügend Speicherplatz.

Gegen Datenfehler sollte man so früh wie möglich ankämpfen, z.B. indem man mit Eingabemasken arbeitet, wo der Benutzer nur korrekte Eingaben machen kann.

Programmierfehler sollte man, zumindest in der Testphase, nicht abfangen. Wenn sie auftreten, muss man sie sofort im Programm beheben.

Systemfehler sind komplizierter zu beheben. Vielleicht findet man einen besseren Algorithmus, der weniger Speicherplatz braucht, oder man kann das Problem auf mehrere Rechner verteilen, oder man muss neue Hardware kaufen.

Datenfehler sind jedoch die am häufigsten auftretenden Fehler, und nicht immer kann man sie an der Quelle abfangen. In vielen Programmiersprachen hat man daher dafür spezielle Mechanismen entwickelt, mit denen man die Fehler abfangen und kontrolliert behandeln kann. Der am meisten verwendete Mechanismus arbeitet mit sog. *Exceptions*. Die Annahme dabei ist, dass der Fehler meist nicht genau an der Stelle behandelt werden kann, wo er auftritt.

Als einfaches Beispiel: ein Programm liest ein Geburtsdatum ein und schreibt es in eine Datenbank. Die Datenbank überprüft, ob das Datum plausibel ist, und stellt fest, dass die Person über 200 Jahre alt sein müsste, was eigentlich nicht sein kann. Da die Datenbank selbst nicht weiß, wo die Daten her kommen, kann sie auch nicht direkt reagieren. Stattdessen muss sie dem aufrufenden Programm melden, dass etwas faul ist, und das Programm muss reagieren.

Mit dem Exception-Mechanismus geht das dann folgendermaßen: Die Datenbank erzeugt ein sog. Exception-Objekt, in das die Fehlermeldung verpackt ist. Mit einem Befehl `throw` „wirft“ sie die Exception. Das bewirkt, dass das Datenbankprogramm sofort verlassen wird und die Kontrolle an das aufrufende Programm zurück gegeben wird. Das aufrufende Programm muss mit einem sog. `try-catch`-Ausdruck darauf vorbereitet sein, die Exception aufzufangen und darauf zu reagieren. Wenn keine Exception geworfen wird, wird der catch-Ausdruck ignoriert. Wenn eine Exception geworfen wird, dann wird der catch-Ausdruck ausgeführt.

In dem Datenbankbeispiel könnte das schematisch vielleicht so aussehen.

```
Datenbank:
schreibeGeburtsdatum(Geburtsdatum) {
    ...
    if(dauer(heute,Geburtsdatum) > 200 Jahre) {
```

```
        throw(new Exception("Alter mehr als 200 Jahre")}
    ... }

Anwendungsprogramm:
    ...
    while(true) {
          Geburstdatum = leseDatum()
          try{Datenbank.schreibeGeburtsdatum(Geburtsdatum);}
          catch(Exception ex) {
              print(ex.message);
              print("neues Geburtsdatum?");
              continue;}
           break;}
    ...
```

Im Anwendungsprogramm würde bei einer Exception die Nachricht ausgedruckt werden, dann die Aufforderung nach einem neuen Geburtsdatum gedruckt werden, und dann mit `continue` die `while`-Schleife wieder von vorne laufen. Erst wenn die Eingabe korrekt ist und keine Exception gefangen wird, wird die `while`-Schleife mit `break` beendet.

7 Design Patterns (Entwurfsmuster)

Es gibt inzwischen bestimmt Abermillionen verschiedene Programm, die im Umlauf sind. Es gibt aber garantiert nicht Abermillionen verschiedene Probleme, die auf unterschiedliche Weise programmiert werden müssen. D.h. in ganz vielen Programmen werden ganz ähnliche Probleme gelöst. Zur Lösung ähnlicher Programmierprobleme kann man folgende Überlegungen anstellen:

- Wie kann man die Probleme klassifizieren und benennen?
- Wie kann man die Problemklassen auf einheitliche Weise lösen?
- Kann man in den Programmiersprachen Unterstützung geben für diese einheitlichen Lösungen?
- Hilft es Programmierer A, wenn ihm Programmierer B sagt, ich habe Problem X mit der Standardmethode Y gelöst?

Es gibt natürlich Probleme, die so standardisiert sind, dass man sie in Bibliotheken packt, und einfach die Bibliotheksfunktion aufruft. Z.B. die Berechnung des Sinus eines Winkels implementiert man einmal, packt sie in eine Bibliothek, und verteilt die Bibliothek.

Andere Probleme sind aber nicht so einfach als Bibliotheksfunktionen zu lösen, haben aber doch eine so eindeutige Struktur, dass man dafür ein standardisiertes Lösungsverfahren angeben kann. Genau das sind die *Design Patterns*.

7.1 Das Observer-Pattern

Wir illustrieren das Prinzip am *Observer-Pattern* (Beobachter-Muster).

Ein konkreter Fall:
Softwarefirma A entwickelt eine graphische Benutzeroberfläche (GUI) mit Fenstern, Buttons (Knöpfe) usw., und verkauft sie an die Firmen B und C, die sie für irgendwelche Anwendungsprogramme brauchen. Wenn ein Button gedrückt wird, soll natürlich etwas passieren. Aber was passieren soll, bestimmen die Firmen B und C. Allerdings möchte Firma A keine Spezialversion ihrer GUI für die Firmen B und C herstellen.

Das Observer-Pattern empfiehlt nun folgendes: In der Button-Klasse von Firma A wird eine Liste von *Observern* vorgesehen. Observer sind Objekte, die eine bestimmte Methode, sagen wir `button-pressed` haben. Außerdem hat die Button-Klasse von Firma A Methoden `add-Observer` und `remove-Observer`, mit denen man einen Observer hinzufügen und wegnehmen kann.

Wenn nun ein Button gedrückt wird, läuft eine Schleife über die Liste von Observern und ruft die `button-pressed` Methode auf.

Jetzt können die Firmen B und C jeweils ihre eigenen `button-pressed`-Methoden definieren, und mit `add-Observer` dem Button hinzufügen. Wenn der Button gedrückt wird, passiert das, was die Firmen B und C wollen, ohne dass die Firma A noch irgendwie involviert wird.

Der abstrakte Fall:
Ein Teilprogramm X erzeugt Ereignisse (z.B. Button gedrückt). Die Teilprogramme Y_1 bis Y_n wollen das mitbekommen und darauf reagieren.

Lösung: Teilprogramm X sieht eine Liste von Observern vor, zusammen mit Methoden, um Observer hinzuzufügen und wegzunehmen. Die Teilprogramme Y_1 bis Y_n erzeugen jeweils ihre eigenen Observer und fügen sie der Liste hinzu. Wann immer in Teilprogramm X das Ereignis eintritt, läuft eine Schleife über die Liste und aktiviert die Observer in der Liste indem sie eine vorher vereinbarte Methode aufruft.

Das Observer-Pattern, wie auch all die anderen Design-Patterns sind nichts, was man als Bibliotheksfunktionen implementieren kann. Wenn man sich aber daran hält, und Programmierer A dem Programmierer B sagt, er hat das Observer-Pattern implementiert, dann weiß Programmierer B, was das bedeutet, und wie es funktioniert.

Die Design-Patterns sind nicht unbedingt spezifisch für bestimmte Programmiersprachen. Es gibt aber Sprachen, die ein Pattern besser unterstützen als andere. Das Observer-Pattern, z.B., ist sehr leicht in objektorientierten Sprachen zu implementieren. Ohne das Klassenprinzip dieser Sprachen wird es dagegen schwieriger zu nutzen.

Es werden laufend neue Design-Pattern definiert. Inzwischen gibt es über 70 davon.

8 Dokumentieren

Die Erfahrung zeigt: ein Programm ohne Dokumentation wird selbst dem Programmierer schon nach relativ kurzer Zeit mysteriös, wenn nicht gar unverständlich.

Daher gilt: **fast so wichtig wie das Programm selbst ist seine Dokumentation.**

Deshalb wird das Dokumentieren von der Programmiersprache selbst, aber noch mehr von speziellen Dokumentationssystemen unterstützt. Die Programmiersprache bietet Möglichkeiten, Kommentare eben als Kommentare auszuzeichnen, die das Programm nicht beeinflussen.

Für die Programmiersprache Java gibt es das Programm **Javadoc**. Es analysiert ein ganzes Projekt, bestehend aus Java Packages mit ihren Klassen, extrahiert daraus die Klassenstruktur und die Dokumentationen, und bereitet sie als HTML-Seiten auf.

Für die Sprachen C++, C, Objective-C, Java, Python, Fortran und IDL gibt es das Programm **Doxygen**, welches ebenfalls die Strukturen und Dokumentationen zu HTML-Seiten aufbereitet.

9 Debuggen und Profilen

Auch mit größter Sorgfalt implementierte und getestete Programme können noch Fehler haben. Tritt ein Fehler auf, muss man ihn finden und beheben, d.h. man muss das Programm *debuggen*.

Will man das Programm schneller machen, muss man Stellen finden, wo sich Optimierungen lohnen, Es nutzt nämlich gar nichts, den Teil des Programms zu optimieren, in dem das Programm sowieso nur einen Bruchteil seiner Zeit verbringt. *Profiler* sind Programme, mit denen man die Aufrufhäufigkeiten und Zeiten in einem laufenden Programm analysieren kann. Mit den Ergebnissen kann man dann entscheiden, wo sich Optimierungen lohnen.

9.1 Debuggen

Moderne Entwicklungsumgebungen sind so komfortabel, und die Rechner sind so schnell, dass man eine ganz einfache Art des Debuggens machen kann:

An den Stellen, wo man einen Verdacht hat, dass etwas schief läuft, schreibt man im Quellcode ein Druckkommando hin, um die Werte von Variablen ansehen zu können. Mit einem einzigen Knopfdruck wird dann das Programm neu compiliert und laufen gelassen.

Etwas komfortabler sind *Haltepunkte* (*Breakpoints*). Das sind Stellen, wo das Programm angehalten wird, so dass man dessen momentanen Zustand, insbesondere die Werte der gerade sichtbaren Variablen inspizieren kann. Anschließend kann man das Programm an der Stelle wieder weiter laufen lassen. Solche Breakpoints können entweder von der Entwicklungsumgebung, oder von speziellen Debugger-Programmen gesetzt werden. Bei interpretierten Sprachen ist es u.U. sogar möglich, das

Programm an einem Breakpoint zu verändern, und dann mit dem veränderten Programm weiterzurechen.

Viele Fehler kann man mit diesen einfachen Methoden finden. Wie die Erfahrung zeigt, gibt es aber auch in höchst professionellen Programmen noch Fehler, die erst nach Jahren entdeckt werden. Das Debuggen von komplexen Programmen ist eine echte Kunst, die sehr viel Wissen und Erfahrung erfordert.

Disassembler
Wenn man mit der Fehlersuche gar nicht mehr weiterkommt, kann als letztes Mittel ein *Disassembler* helfen. Das ist ein Programm, welches Maschinencode zurück transformieren kann in eine menschenlesbare Form, und das ist meist der Assemblercode. An dem disassemblierten Code können erfahrene Programmierer erkennen, wie das Maschinenprogramm tatsächlich funktioniert. Wenn es nicht so funktioniert, wie der Programmierer es eigentlich erwartet, kann das ein Hinweis auf den Fehler sein.

Auch wenn das Programm fehlerfrei arbeitet, aber vielleicht zu langsam ist, kann man an dem disassemblierten Code u.U. Redundanzen erkennen, die man wegoptimieren kann.

9.2 Profilen

Mancher Anfänger unter den Programmierern hat sich schon gewundert, dass er eine tolle Optimierung in seinem Programm gemacht hat, aber am Ende wurde das Programm nur minimal schneller. Wenn man den Teil des Programms 100 mal schneller macht, der aber bisher nur 1 % der Zeit benötigt hat, dann hat man das Programm insgesamt von 100 Zeiteinheiten auf 99,01 Zeiteinheiten beschleunigt. Das ist frustrierend.

Profis unter den Programmieren untersuchen dagegen zuerst, in welchen Teilen des Programms wieviel Zeit zugebracht wird, und welche Funktionen wie oft aufgerufen werden. Dann kann man versuchen, die zeitaufwendigsten Teile zu optimieren. Ein Hilfsmittel dazu ist ein *Profiler*. Er modifiziert ein Programm so, dass bei allen Funktionsaufrufen Zeit- und Häufigkeitsdaten gesammelt werden. Wenn das modifizierte Programm neu compiliert ist und läuft, dann werden die Daten gesammelt und in einen File geschrieben. Mit einem anderen Programm kann der File dann eingelesen, die Daten aufbereitet und visualisiert werden.

Ein sehr guter Debugger und Profiler für Linux Programme ist valgrind. Valgrind kann sogar mit compilierten Programmen, ohne Quellcode arbeiten. Es gibt aber noch eine Reihe anderer Profiler und Debugger. Gute Entwicklungsumgebungen bieten diese Möglichkeiten ebenfalls an.

10 Testen

Jeder Programmierer, auch der blutigste Anfänger wird ein fertig implementiertes Programm oder Programmteil als erstes mal testen. D.h. er lässt es mit Testdaten laufen. Das Testen von Programmen ist aber ein so wichtiger Teil der Softwareentwicklung, dass es auch dafür detaillierte Empfehlungen

oder sogar Vorschriften gibt. In machen Softwarefirmen sind Personen sogar nur zum Testen von Programmen angestellt.

Man unterscheidet beim Testen mehrere Stufen.

10.1 Komponententest (Modultest, Unittest)

Als erstes testet i.A. der Programmierer selbst die einzelnen Komponenten, Funktionen, Methoden, Module eines Programms, indem er sie mit Testdaten laufen lässt und die Ergebnisse mit den erwarteten Werten vergleicht.

Ganz wichtig ist dabei, die Testfälle auszuprogrammieren und für spätere Verwendung abzuspeichern. Die Entwicklungsumgebungen unterstützen einen dabei, Testfälle zu programmieren und automatisch ablaufen zu lassen. Ändert man das Programm, kann man die Tests mit einem Knopfdruck wieder automatisch durchlaufen lassen.

Es gibt sogar eine Programmiermethodik, *Testgetriebenes Programmieren*. Dabei implementiert man zuerst die Tests, und entwickelt dann das Programm. Bei jeder Verfeinerung des Programms kann man dann sehen, inwieweit die Tests schon, bzw. nach Änderungen noch, korrekte Ergebnisse liefern.

10.2 Integrationstest

Hierbei testet man die Zusammenarbeit voneinander abhängiger Komponenten. Der Testschwerpunkt liegt auf den Schnittstellen der beteiligten Komponenten und soll korrekte Ergebnisse über komplette Abläufe hinweg nachweisen.

10.3 Systemtest

In dieser Stufe wird das gesamte System gegen die gesamten Anforderungen (funktionale und nichtfunktionale Anforderungen) getestet. Dazu braucht man eine Testumgebung und Testdaten, die schon möglichst nahe an der Produktivumgebung des Kunden sind, wo aber Fehler noch keinen Schaden anrichten können.

10.4 Abnahmetest (Verfahrenstest, Akzeptanztest oder auch User Acceptance Test (UAT))

Jetzt wird das Programm durch den Kunden in seiner Umgebung mit seinen Daten getestet. Erst wenn der Test erfolgreich war, muss der Kunde bezahlen.

10.5 Black-Box- und White-Box-Tests

Bei einem Black-Box-Test wird nur das Verhalten der zu testenden Einheit nach außen getestet. Dazu benötigt man nur die Spezifikation der Schnittstellen, und testet, ob zu den Eingabedaten auch die richtigen Ergebnisse berechnet werden.

Ein White-Box-Test kann nur vom Entwickler selbst durchgeführt werden, der die interne Struktur des zu testenden Teils genau kennt. Er kann dann die Testfälle so auswählen, dass kritische Teile des Programms genau getestet werden. Allerdings besteht die Gefahr, dass er die Testfälle nach seinem Programm, und nicht nach der Spezifikation auswählt, und daher Fälle übersieht.

Ein einfaches Beispiel: das Programm berechnet die Fakultätsfunktion, allerdings erst ab 1 und nicht ab 0. Der Programmierer, der das weiß, testet daher nur die Fälle ab 1. Eine andere Person würde bei einem Black-Box-Test auch auf die Idee kommen, die 0 zu testen, und findet dann den Fehler.

Eine Mischung zwischen Black-Box- und White-Box-Tests sind *Grey-Box-Tests*. Dabei orientiert man sich in erster Linie an der Spezifikation der Schnittstelle, kennt aber auch die interne Struktur des Programms, und kann daher Testfälle ausprobieren, die für die interne Struktur kritisch sind.

10.6 Auswahl der Testfälle

Fast alle Programme sollten, zumindest theoretisch, für unendlich viele Daten korrekt funktionieren. Man kann aber nicht unendlich viele Fälle testen. Daher muss eine Auswahl getroffen werden. Bei der Auswahl der Testfälle sollte man zwei Aspekte berücksichtigen:

- Die Auswahl sollte alle Fälle abdecken, die das Programm unterschiedlich behandeln muss (das geht eigentlich nur bei White-Box- und Grey-Box-Tests).
- Die Auswahl richtet sich nach den Datentypen für die Eingaben. Insbesondere sollten Extremwerte bei den Eingaben getestet werden. Akzeptiert das Programm z.B. einen String als Eingabe, testet man auch dem leeren String.

11 Verifizieren

Leider gibt es eine alte Weisheit der Softwareentwicklung, die zu allen Zeiten gilt:

Tests können nur die Anwesenheit von Fehlern beweisen, aber nie deren Abwesenheit!

Die *Abwesenheit* von Fehlern kann man nur durch einen mathematischen Beweis nachweisen. Es gibt dabei drei Aspekte zu beweisen:

Terminierung: Terminiert das Programm für alle zulässigen Eingaben, d.h. insbesondere dass keine Endlosschleifen und keine unendliche Rekursion auftritt.
Das ist natürlich nur relevant für Programme, die auch terminieren sollen.
Betriebssysteme oder Serverprogramme sollten natürlich möglichst nie terminieren.

Korrektheit: Sind die Ergebnisse für alle zulässigen Eingaben richtig?
Hierfür benötigt man eine formale Spezifikation, d.h. eine mathematische Beschreibung dessen, was das Programm leisten soll.

Vollständigkeit: Deckt das Programm auch alle geforderten Eingaben ab?

Ein **Terminierungsbeweis** wird notwendig, wenn das Programm Schleifen hat, oder rekursive Aufrufe. Eine typische Beweistechnik dabei ist, eine sog. *Abstiegsfunktion* zu finden, und nachzuweisen, dass sie bei jedem Schleifendurchgang bzw. bei jedem rekursiven Aufruf näher an den Wert kommt, bei dem die Schleife oder die Rekursion abbricht.

Bei unserem Lieblingsbeispiel, der Fakultätsfunktion,

```
Fakultaet(n) = if(n == 0) then 1 else n*Fakultät(n-1)
```

ist die Abstiegsfunktion einfach das Argument `n` selbst. Es wird bei jedem rekursiven Aufruf um 1 kleiner, und endet bei der 0.

Dass das nicht immer so einfach ist, zeigt folgende Funktion:

$$\text{Collatz}(n) = \begin{cases} 1 & \text{falls } n = 1 \text{ ist} \\ \text{Collatz}(n/2) & \text{falls } n \text{ gerade ist} \\ \text{Collatz}(3n+1) & \text{sonst} \end{cases}$$

Bei allen bisherigen Tests hat sie mit dem Wert 1 terminiert. Es konnte aber noch niemand eine Abstiegsfunktion finden, mit der der Terminierungsbeweis funktioniert.

Um Korrektheit und Vollständigkeit zu beweisen, muss man das Programm zusammen mit seiner Spezifikation in ein mathematisches Problem transformieren, für das man dann mathematische Beweise führen kann. Das ist wirklich komplex, und Gegenstand aktueller Forschung.

12 Programmiersprachen beurteilen

Erfahrene Programmierer, die sich an eine für sie neue Programmiersprache heranwagen, informieren sich zunächst über eine ganze Reihe von Aspekten zur Sprache selbst:

- Was ist das vorherrschende Programmierparadigma:
  imperativ, funktional, logisch, objektorientiert?
- Werden die Programme interpretiert oder compiliert, oder beides?
- Arbeiten die Programme mit einer virtuellen Maschine oder direkt auf Prozessorebene?
  Eine virtuelle Maschine hat den Vorteil, dass die compilierten Programme unabhängig von der Prozessorarchitektur sind.
- Basiert sie auf 32-Bit oder 64-Bit Architektur, oder beidem?
  (Java basiert grundsätzlich auf 32-Bit Architektur)
- Welche Basisdatentypen werden unterstützt: Integer, Boolean usw.

- Sind auch komplexerer Datentypen in die Sprache eingebaut, insbesondere z.B. Listen?
- Gibt es auch in die Sprache eingebaute Algorithmen?
  In der Sprache Perl z.B. sind Reguläre Ausdrücke Teil der Sprache. Daher eignet sie sich besonders für die Textverarbeitung.
  In der Sprache Prolog ist Suche mit Backtracking, sowie Constraint Handling eingebaut.
- Gibt es ein Typsystem, und wie sieht es aus?
- Wie wird die Fehlerbehandlung programmiert?
- Wie sind die Ein-Ausgabeschnittstellen?
- Wie werden Programme kommentiert?
- Wird Parallelisierung unterstützt, z.B. durch Threads?
- Sind Funktionen auch sog. *first-class-objects*, d.h. Objekte, die man an Variablen binden kann, bzw. gibt es Alternativen dazu?
  Das braucht man, wenn man Algorithmen schreiben will, die mit beliebigen Funktionen umgehen sollen. Ein Beispiel wäre ein numerisches Integrationsverfahren, dass mit allen integrierbaren Funktionen arbeiten können sollte.
- Wie sieht die konkrete Syntax aus?

Neben der eigentlichen Programmiersprache gibt es noch eine Reihe weitere Aspekte, die für oder gegen eine konkrete Programmiersprache sprechen:

- In welchen Betriebssystemen ist die Sprache verfügbar?
- Welche Entwicklungsumgebungen unterstützen die Sprache?
  Programmieranfänger würden wahrscheinlich zunächst mit Eclipse arbeiten wollen.
- Gibt es Debugger, Profiler, Disassembler, Kommentar-Aufbereitungsprogramme
  und evtl. weitere Unterstützungsprogramme für die Sprache?
- Hat die Sprache einen *interaktiven Modus*, d.h. die Möglichkeit, Code direkt am Terminal einzutippen und laufen zu lassen. Zum Ausprobieren, für Entwicklung und Tests ist das oft ein nützliches Hilfsmittel.
- Welche Bibliotheken für die Sprache gibt es?
  Sind sie Open-Source oder kommerziell?
  Für viele Sprachen gibt es eine Unmenge an Bibliotheken. Ein guter Programmierer erfindet nicht das Rad neu, sondern benutzt Bibliotheken, wo immer es geht.
- Gibt es fertige Frameworks, insbesondere für Web-Anwendungen in der Sprache?
  Komplexe Anwendungen werden kaum noch von Grund auf programmiert, sondern benutzen Frameworks.
- Wer pflegt die Sprache und entwickelt sie weiter?
  Insbesondere: gibt es eine Community, die sich um die Sprache kümmert?
  Ganz neue Sprachen werden oft von Universitäten entwickelt, wo nicht klar ist, wie lange die Sprache das Ende der Promotion des Entwicklers überlebt.

Auch für einen Anfänger kann es nützlich sein, sich zunächst wenigstens grob mit diesen Themen zu befassen, bevor er sein erstes Programm schreibt.

13 Programmiersprachen

Es gibt Hunderte von Programmiersprachen. Viele davon sind schon veraltet. Andere sind nur für Spezialanwendungen. Eine ganze Reihe von Sprachen sind jedoch heutzutage weit verbreitet, so dass es sich vielleicht lohnt sich mit ihnen auseinanderzusetzen.

Eine sicherlich völlig unvollständige Auswahl davon ist:

C und C++
C wurde von Dennis Ritchie in den frühen 1970er Jahren entwickelt. Man kann C als die nächste Stufe über den Assemblersprachen sehen. C ist rein imperativ, und noch sehr nahe an der Prozessorarchitektur. Programme, bei denen es auf jeden einzelnen Schritt des Prozessors ankommt, kann man statt in Assemblersprache auch in C schreiben. Daher wurde C besonders benutzt, um Betriebssystemkerne zu implementieren. Die erste Anwendung war in der Tat die Implementierung des Unix Betriebssystems.

Da C noch sehr „low-level“ ist, wurde es ab 1979 von Bjarne Stroustrup zu C++ erweitert. C++ ist inzwischen eine mächtige objektorientierte Sprache, die aber noch alle low-level Möglichkeiten von C hat. Wenn man einerseits abstrakt und komfortabel programmieren will, und trotzdem so effizienten Code wie möglich braucht, dann ist C++ eine gute Wahl. C++ Programme compilieren in Maschinencode, und laufen daher nur auf der Architektur, für die der Code erzeugt wurde. Ohne Neucompilation sind die Programme daher nicht auf andere Architekturen portabel.

Java
Die Sprache wurde in den 1990ern von Sun Microsystems entwickelt und dann von Oracle übernommen. Es ist eine streng typisierte objektorientierte Sprache, deren Programme auf einer virtuellen Maschine laufen, der *Java Virtual Machine* (JVM). Der Slogan dafür heißt: *Write once, run anywhere*. Das stimmt zumindest da wo die gleiche JVM läuft.

Die JVM ist so populär geworden, dass auch eine ganze Reihe anderer Sprachen in sie übersetzen: Ceylon, Clojure, Erjang, Free Pascal, Groovy, JRuby, Jython, Scala und Kotlin. Diese können daher beliebig mit Java gemischt werden.

Für Java gibt es ausgereifte Entwicklungsumgebungen, eine Unmenge an Bibliotheken und etliche Framweworks.

Java bietet für Anfänger den Vorteil, dass sie, und das seit fast 20 Jahren, im ersten Semester Java lernen können, und dann nach dem Studium einen Job finden können, wo sie Java programmieren müssen. Kaum eine andere Sprache hat das über diese lange Zeit geschafft.

C# (Visual C#)
wurde von Microsoft entwickelt. Sie ist eine statisch typisierte objektorientierte Sprache, die Konzepte von Java, C, C++, Haskell sowie Delphi übernommen hat. Die meisten für das Windows Betriebssystem entwickelten Programme werden in C# geschrieben. Inzwischen ist es aber auch möglich,

C#-Programme für macOS, iOS, Android und Linux zu schreiben. C# compiliert, ähnlich wie Java in eine Zwischensprache, die dann von einer virtuellen Maschine verarbeitet wird. Daher ist sie ebenfalls weitgehend plattformunabhängig.

Python
Python ist eine interpretierte Sprache und unterstützt mehrere Programmierparadigmen, insbesondere objektorientierte, aspektorientierte und die funktionale Programmierung. Sie bietet eine dynamische Typisierung. Die Sprache hat ein offenes, gemeinschaftsbasiertes Entwicklungsmodell, das durch die gemeinnützige Python Software Foundation, gestützt wird.

Python hat eine knappe gut lesbare übersichtliche Syntax, die gerade für Anfänger gut zu lernen ist. Die Sprache wird aber auch für professionelle Software immer beliebter.

Als interpretierte Sprache laufen ihre Programme überall, wo der Interpretierer verfügbar ist.

Haskell
Haskell ist die modernste rein funktionale Sprache. Es gibt nur Funktionen ohne Nebeneffekte. Auch die Werte von Variablen kann man nicht verändern. Allerdings kann man mit einem speziellen Typ, sog. *Monaden* zustandsbehaftete Rechnungen, wie z.B. ein Zufallsgenerator, rein funktional implementieren. Haskell ist streng typisiert, und hat auch Funktionstypen. Funktionen sind daher auch First-Class-Objects, die wiederum Argumente anderer Funktionen sein können.

Prolog
Dies ist die am besten entwickelte rein logische Programmiersprache. Sie ist inzwischen über Jahrzehnte gewachsen und enorm optimiert worden. Insbesondere hat sie ein integriertes Constraint Handling. Prolog eignet sich daher besonders für die Lösung von Suchproblemen mit Constraints.

PHP
PHP ist keine Programmiersprache, wie die anderen Sprachen, sondern ein sog. *Hypertext Preprocessor*. PHP-Code wird in HTML-Code eingebaut, der dann von einem Web-Server in reines HTML umgewandelt wird.

Ein Beispiel wäre:

```
<html>
    <head>
        <title>Hallo-Welt-Beispiel</title>
    </head>
    <body>
        <?php echo 'Hallo Welt!'; ?>
    </body>
</html>
```

PHP bietet insbesondere eine gute Datenbankanbindung, sowie Interaktionsmöglichkeiten mit Web-Clients.

JavaScript
JavaScript hat, anders als der Name vermuten lässt, nichts mit Java zu tun. Es ist eine sog. Scriptsprache, die in einem Web-Browser ausgeführt wird. Typische Anwendungen sind Berechnungen, die ein Web-Browser lokal durchführen kann, ohne den Web-Server zu bemühen. Dazu hat JavaScript Zugriff auf das sog. *Document Object Model* (DOM). Das ist die Datenstruktur, mit der HTML-Seiten im Browser repräsentiert werden. Mit JavaScript kann man diese Datenstruktur, und damit die Webseite direkt im Browser manipulieren.

Viele Anwendungen sind jedoch nicht mehr mit reinem JavaScript programmiert, sondern man benutzt das JavaScript-Webframework *AngularJS*.

Fortran
Fortran wurde ursprünglich 1953 von John W. Backus bei IBM in erster Linie für numerische Berechnungen entwickelt. Sie ist eine der ganz wenigen Programmiersprachen, die all die Jahrzehnte überlebt hat, und immer weiter verbessert wurde. Die letzte Version ist Fortran 2008.

Fortran unterstützt insbesondere Vektoren und Matrizen, und lässt sich in effizienten Code für Vektorprozessoren und Multiprozessor-Rechner compilieren. Komplexe numerische Berechnungen auf Supercomputern werden daher häufig in Fortran programmiert.

COBOL
COBOL steht für „Common Business Oriented Language“ und ist einer der leider noch überlebenden Saurier aus den 1950er Jahren. Niemand würde heute noch neue Programme in COBOL schreiben. Es gibt aber immer noch alte COBOL-Programme, die noch in Benutzung sind, und noch gewartet werden müssen. Die wenigen Experten, die noch in COBOL programmieren können, sind daher sehr gefragt.

Stichwortverzeichnis