Programmierung Grundlagen

– mit Beispielen in Python

Ralph Steyer

1. Ausgabe, Oktober 2018

ISBN: 978-3-86249-833-8

PGPY

HERDT

1 Informationen zu diesem Buch 4

1.1 Voraussetzungen 4
1.2 Aufbau und Konventionen 5
1.3 HERDT BuchPlus – unser Konzept 6

2 Grundlagen zu Computing, Programmen und Programmiersprachen 7

2.1 Soft- und Hardware 7
2.2 Grundlagen zu Programmen 7
2.3 Computing und Computational Thinking 8
2.4 Qualitätskriterien und Dokumentation 11
2.5 Warum programmieren? 15
2.6 Klassifizierung von Programmiersprachen 15
2.7 Die Klassifizierung nach Generationen 16
2.8 Prozedurale und funktionale Programmiersprachen 18
2.9 Objektorientierte Programmiersprachen 20
2.10 Hybride Programmiersprachen und Skriptsprachen 22
2.11 Logische Programmiersprachen 24
2.12 Erziehungsorientierte Programmiersprachen und Minisprachen 24
2.13 Webprogrammierung und mobile Apps 25
2.14 Übungen 26

3 Programmlogik und Darstellungsmittel für Programmabläufe 28

3.1 Abstraktion der Wirklichkeit 28
3.2 Programmlogik und Programmablauf 28
3.3 Programmabläufe visualisieren 29
3.4 Programmablaufplan 30
3.5 Datenflussdiagramm 31
3.6 Struktogramme 32
3.7 Pseudocode 33
3.8 Entscheidungstabellen 34
3.9 Übung 35

4 Werkzeuge der Softwareentwicklung 36

4.1 Programme erstellen 36
4.2 Konzepte zur Übersetzung 37
4.3 Entwicklungsumgebungen 39
4.4 Standardbibliotheken und Wiederverwendung 40
4.5 Grundaufbau eines Programms am Beispiel Python 40
4.6 Übungen 45

5 Zahlensysteme und Zeichencodes 46

5.1 Zahlensysteme unterscheiden 46
5.2 Programme basieren auf Daten 48
5.3 Digitales Rechnen 50
5.4 Zeichencodes 52
5.5 Übung 54

6 Grundlegende Sprachelemente 55

6.1 Syntax und Semantik 55
6.2 Grundlegende Elemente einer Sprache 57
6.3 Standarddatentypen (elementare Datentypen) 59
6.4 Literale für primitive Datentypen 62
6.5 Variablen und Konstanten 62
6.6 Operatoren 68
6.7 Ausdrücke und Operatorenrangfolgen 72
6.8 Übungen 74

7 Kontrollstrukturen 75

7.1 Anweisungen und Folgen 75
7.2 Bedingungen und Kontrollstrukturen 77
7.3 Grundlagen zu Verzweigungen 78
7.4 Bedingte Anweisung 78
7.5 Verzweigung 79
7.6 Geschachtelte Verzweigung 80
7.7 Mehrfache Verzweigung (Fallauswahl) 81
7.8 Schleifen 83
7.9 Zählergesteuerte Schleife (Iteration) 84
7.10 Kopfgesteuerte bedingte Schleife 86
7.11 Fußgesteuerte bedingte Schleife 87
7.12 Schnellübersicht 89
7.13 Sprunganweisungen 91
7.14 Endlosschleifen 92
7.15 Übungen 92

8 Elementare Datenstrukturen 95

8.1 Warum werden Datenstrukturen benötigt? 95
8.2 Arrays 96

© HERDT-Verlag

8.3 Eindimensionale Arrays 97
8.4 Records 98
8.5 Zeichenketten 98
8.6 Tupel und Listen 99
8.7 Dictionaries 103
8.8 Mengen 104
8.9 Besondere Datenstrukturen anhand von Stapel (Stack) und Schlangen (Queue) 105
8.10 Übungen 108

9 Methoden, Prozeduren und Funktionen 110

9.1 Unterprogramme 110
9.2 Parameterübergabe 113
9.3 Parameterübergabe als Wert 114
9.4 Parameterübergabe über Referenzen 114
9.5 Rückgabewerte von Funktionen oder Methoden 115
9.6 Innere Funktionen – Closures 116
9.7 Standardbibliotheken und Built-in-Funktionalitäten 116
9.8 Übungen 121

10 Algorithmen 122

10.1 Eigenschaften eines Algorithmus 122
10.2 Iterativer Algorithmus 122
10.3 Rekursiver Algorithmus 124
10.4 Iterativ oder rekursiv? 125
10.5 Generischer Algorithmus 126
10.6 Übung 127

11 Reaktion auf Ereignisse 128

11.1 Grundlagen zu Ereignissen und Eventhandlern 128
11.2 Timer und Scheduler 129
11.3 Ereignisbehandlung bei Programmen mit grafischen Oberflächen 131
11.4 Verschiedene Techniken der Ereignisbehandlung 134
11.5 Schnellübersicht 136
11.6 Übung 136

12 Grundlagen der Softwareentwicklung 137

12.1 Software entwickeln 137
12.2 Methoden 139
12.3 Der Software-Lebenszyklus 140
12.4 Vorgehensmodelle im Überblick 144
12.5 Computergestützte Softwareentwicklung (CASE) 148
12.6 Fehler finden und identifizieren 149
12.7 Schnellübersicht 159
12.8 Übung 159

Anhang A: PAP, Struktogramm und Pseudocode 160

A.1 Beispiel Zinsberechnung 160
A.2 Beispiel Geldautomat 161

Anhang B: Installationen und Quellangaben 162

B.1 Python laden und installieren 162
B.2 Quellangaben im Internet 164

Stichwortverzeichnis 166

1 Informationen zu diesem Buch

In diesem Kapitel erfahren Sie

- an wen sich dieses Buch richtet
- welche Vorkenntnisse Sie mitbringen sollten
- welche Hard- und Software Sie für die Arbeit mit diesem Buch benötigen
- wie dieses Buch aufgebaut ist
- welche Konventionen verwendet werden

1.1 Voraussetzungen

Zielgruppe

- Programmiereinsteiger
- Auszubildende in IT-Berufen
- Schüler mit IT-Schwerpunkt
- Teilnehmer ECDL Module und Zertifikate

Empfohlene Vorkenntnisse

Folgende Kenntnisse werden für eine erfolgreiche Benutzung dieses Buchs vorausgesetzt:

- Grundkenntnisse im Umgang mit Windows, Linux oder MacOS
- Grundkenntnisse in der Bedienung von Anwendungsprogrammen
- Grundkenntnisse im Umgang mit dem Internet

Hinweise zur Software

- Bei den Beispielen im Buch wird überwiegend die Programmiersprache **Python** in der Version 3.6.x verwendet, wenn konkrete Programmierung durchgeführt wird. Python ist eine universelle, höhere Programmiersprache, die üblicherweise interpretiert wird. Python ist den meisten gängigen Programmiersprachen verwandt, wurde aber mit dem Ziel größter Einfachheit und Übersichtlichkeit entworfen. Hinweise zur Installation und Konfiguration von Python finden Sie im Anhang.
- Bei Sprachelementen, die Python nicht bereitstellt, werden diese bei Bedarf gelegentlich mithilfe von **Java** erläutert, ohne dass sich um eine konkrete Ausführung des Codes gekümmert wird. Dies ist eine an C/C++ angelehnte streng objektorientierte Sprache, die ursprünglich für Geräte der Konsumelektronik entwickelt wurde. Gegenüber C/C++ wurde auf verschiedene (fehlerträchtige) Konstrukte verzichtet. Deshalb ist Java eine relativ schlanke und übersichtliche Sprache. Mit dem Java-Compiler erzeugt man maschinenunabhängigen Code, sogenannten Bytecode, der in einer virtuellen Maschine ausgeführt wird.
- Als Beispiel für eine komfortable integrierte Entwicklungsumgebung wird **IDLE** verwendet. Diese gehört zum Python-System und wird meist mit Python automatisch installiert. Die Verwendung vereinfacht die Arbeit mit Python erheblich, weshalb im Buch auch nur auf deren Einsatz gesetzt wird. Hinweise zur Installation und Konfiguration von Python finden Sie im Anhang. Als reine Eingabemöglichkeit des Quelltextes kann man im Grunde aber jeden Editor verwenden, der in einem typischen Betriebssystem vorhanden ist. Dieser genügt auch für die Erstellung von Java-Quellcode.

© HERDT-Verlag

- Bei einigen Ausblicken werden bei Bedarf verschiedene weitere Programme und Tools verwendet oder erwähnt.
- Das verwendete Betriebssystem im Buch ist Windows 10, aber Linux und MacOS X werden ebenso berücksichtigt.

1.2 Aufbau und Konventionen

Aufbau und inhaltliche Konventionen des Buchs

- Am Anfang jedes Kapitels finden Sie die Lernziele und am Ende einiger Kapitel eine Schnellübersicht mit den wichtigsten Funktionen im Überblick.
- Die meisten Kapitel enthalten Übungen, mit deren Hilfe Sie die erlernten Kapitelinhalte einüben können.

Hervorhebungen im Text

Im Text erkennen Sie bestimmte Programmelemente an der Formatierung. So werden z. B. Bezeichnungen für Programmelemente wie Register immer *kursiv* geschrieben und wichtige Begriffe **fett** hervorgehoben.

Kursivschrift	kennzeichnet alle von Programmen vorgegebenen Bezeichnungen für Schaltflächen, Dialogfenster, Symbolleisten, Menüs bzw. Menüpunkte (z. B. *Datei - Schließen*) sowie alle vom Anwender zugewiesenen Namen wie Dateinamen, Ordnernamen, eigene Symbolleisten, Hyperlinks und Pfadnamen.
`Courier New`	kennzeichnet Programmtext, Programmnamen, Funktionsnamen, Variablennamen, Datentypen, Operatoren etc.
`Courier New kursiv`	kennzeichnet Zeichenfolgen, die vom Anwendungsprogramm ausgegeben oder in das Programm eingegeben werden.
[ ]	Bei Darstellungen der Syntax einer Programmiersprache kennzeichnen eckige Klammern optionale Angaben.
|	Bei Darstellungen der Syntax einer Programmiersprache werden alternative Elemente durch einen senkrechten Strich voneinander getrennt.

Was bedeuten die Symbole im Buch?

Hilfreiche Zusatzinformation

Praxistipp

Warnhinweis

1.3 HERDT BuchPlus – unser Konzept

Problemlos einsteigen – Effizient lernen – Zielgerichtet nachschlagen

(weitere Infos unter *www.herdt.com/BuchPlus*)

Nutzen Sie dabei unsere maßgeschneiderten, im Internet frei verfügbaren Medien:

So können Sie schnell auf die BuchPlus-Medien zugreifen:

▶ Rufen Sie im Browser die Internetadresse *www.herdt.com* auf.

Katalog Shop DE AT CH
HERDT
Alles
Alles
Titel
Kategorien
Autor
Codes
Home / HERDT-Themen
HERDT-Themen
1 Wählen Sie *Codes*.
Codes
2 Geben Sie den folgenden Matchcode ein: *PGPY*.

2 Grundlagen zu Computing, Programmen und Programmiersprachen

In diesem Kapitel erfahren Sie

- was es mit den Begriffen „Computing“ und „Computational Thinking“auf sich hat
- was Programme sind
- wie sich die Programmiersprachen geschichtlich entwickelt haben
- nach welchen Kriterien Programmiersprachen eingeteilt werden

2.1 Soft- und Hardware

Was ist Software?

Der Begriff **Software** wird meist umfassender als der Begriff **Programm** verstanden. Als Software werden alle immateriellen Teile eines Computers verstanden. Das umfasst Programme, aber auch die zugehörigen Daten. Im täglichen Sprachgebrauch werden die Begriffe Software und Programm jedoch oft synonym verwendet.

Was ist Hardware?

Der Begriff **Hardware** wird meist für alle materiellen (physischen) Teile eines Computers verwendet. Also grob gesagt all das, was man anfassen kann. Zur Hardware eines Computers gehören beispielsweise diese Komponenten:

- Die Grundbestandteile der Rechnerarchitektur: Hauptplatine (auch Motherboard oder Mainboard genannt) mit dem Chip (CPU – Central Processing Unit) bzw. Prozessor und Arbeitsspeicher (RAM – Random-Access Memory)
- Massenspeicher und Speichermedien
- Erweiterungskarten (Grafikkarte, Soundkarte, Netzwerkkarte …)
- Ergänzende Komponenten wie Netzteil, Gehäuse, Lüfter ...
- Peripheriegeräte in Form von
  - Ausgabegeräten (Drucker, Bildschirm, Beamer, Lautsprecher …)
  - Eingabe- (Tastatur, Maus, Joystick …) und Einlesegeräten (Mikrofone, Kartenlesegeräte, Scanner …)

2.2 Grundlagen zu Programmen

Ein Programm oder Skript ist ganz allgemein ein Lösungsweg zur Bearbeitung einer Aufgabe durch einen Computer oder ein verwandtes System. Im Folgenden wird **Programm** als Synonym für Programm und Skript verwendet, wenn nicht ausdrücklich von einem Skript die Rede ist. Ein Programm wird auch **Applikation (engl. application)** oder **Anwendung** genannt. Die Kurzform der englischen Version von Applikation – **App** – ist vor allem bei der Programmierung für mobile Geräte gebräuchlich, setzt sich aber auch in anderen Bereichen mehr und mehr durch.

Vom Algorithmus über den Quellcode zum Programm

Ein **Programm** bzw. Skript ist also eine Beschreibung der Lösung einer vorgegebenen Aufgabe, muss aber in einer spezifischen Programmiersprache umgesetzt werden.

Die Lösung kann (und wird in der Regel) aus einzelnen Bearbeitungsvorschriften bestehen. Eine solche Bearbeitungsvorschrift wird **Algorithmus** genannt.

Ein Algorithmus muss bei jeder möglichen Eingabe von Daten die Verarbeitung nach **endlich vielen Schritten** beenden und einen **eindeutigen Ablauf** mit einem **reproduzierbaren Ergebnis** besitzen.

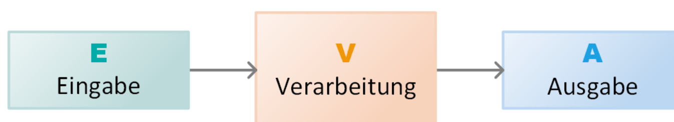
Ein Programm besteht aus Algorithmen, deren Arbeitsschritte in einer Programmiersprache (z. B. in Python, Java, C#, oder JavaScript formuliert sind. Dies ist der sogenannte **Quellcode** oder **Quelltext** (engl. **source code** oder kurz **source**). Gelegentlich findet man dafür auch den Begriff **Programmcode**.

Programme verarbeiten Daten

Ein Programm kann Ihnen beispielsweise Steuern berechnen. Sie teilen dem Programm die Daten mit, die für die Berechnung benötigt werden. Das Programm rechnet nach seinem Algorithmus mit Ihren Daten und liefert am Ende ein Ergebnis.

Programme verarbeiten **Daten**, z. B. Benutzereingaben, und liefern Daten zurück. Diesen Datenfluss durch ein Programm nennt man **EVA-Prinzip** (**E**ingabe - **V**erarbeitung - **A**usgabe).

- **Eingabe**: In das Programm werden Daten eingegeben, z. B. über eine Tastatur oder eine Datenbank.
- **Verarbeitung**: Das Programm verarbeitet diese Daten nach einem vorgegebenen Algorithmus.
- **Ausgabe**: Die Ergebnisse des Programms werden ausgegeben, z. B. auf einen Bildschirm, einen Drucker oder in eine Datenbank.



2.3 Computing und Computational Thinking

Da das ECDL-Modul explizit die Begriffe „Computing“ und „Computational Thinking“ verwendet, sollen diese Begriffe zuerst kompakt erläutert werden.

Computing und Automatisierung

In der IT-Szene wird der Begriff „Computing“ als solches zwar kaum verwendet, aber er ist schon auf einer sehr theoretischen und allgemeinen Ebene standardisiert. Allgemein bezeichnet man damit alle zielorientierten Tätigkeiten, die

- auf Computern beziehungsweise algorithmischen (berechenbaren, reproduzierbaren) Prozessen aufbauen,
- von ihnen profitieren oder
- solche hervorbringen.

© HERDT-Verlag

Computing ist wie gesagt ein sehr theoretischer und umfassender Begriff. Dieser weist eine Vielzahl unterschiedlicher und teils spezieller Ausprägungen auf. Etwa diese:

- Der Entwurf von Hardware und Software
- Die Entwicklung sowie die Herstellung von Hardware und Software
- Die Verarbeitung, Strukturierung sowie Verwaltung verschiedener Arten von Informationen
- Das wissenschaftliche Forschen an und mit Computern
- Das Sammeln von Informationen für bestimmte Zwecke
- Rechenarbeiten mit hohem Bedarf an Rechenleistung oder Speicherkapazität
- Die Vernetzung des Alltags mittels „intelligenter" (smarter) Gegenstände

Aber es gibt noch eine Vielzahl weiterer Ausprägungen, denen letztendlich nur gemein ist, dass sich die damit verbundenen Aufgaben (effizient) **automatisieren** lassen. Und diese Automatisierung basiert auf den gerade eingeführten Algorithmen und Programmen, was im Fokus des Buches stehen soll.

Computational Thinking

Das sogenannte „Computational Thinking" (computergestütztes Denken) bezeichnet eine der vielen Definitionen für einen computergestützten Denkprozess, die sich über die Jahre entwickelt haben. Bei der so bezeichneten Vorgehensweise ist es das Ziel, ein Problem zu formulieren und seine Lösung(en) so auszudrücken, dass der Prozess von einem Menschen oder einer Maschine bzw. einem Computer gleichermaßen effektiv ausgeführt werden kann.

Eine weitere Charakterisierung des „Computational Thinking" als theoretisches Konzept versteht dieses als einen iterativen Prozess, der auf drei abstrakten Stufen basiert.

- Eine Person kann eine Problemstellung identifizieren und abstrakt modellieren.
- Die Problemstellung kann danach in Teilprobleme oder -schritte zerlegen werden.
- Es lassen sich darauf aufbauend Lösungsstrategien entwerfen und auszuarbeiten und diese formalisiert so darzustellen, dass sie von einem Menschen oder auch einem Computer verstanden und ausgeführt werden können.

Um die Vorgehensweise zu verdeutlichen, kann man sich eine einfache mathematische Aufgabenstellung vorstellen.

Die Aufgabenstellung soll lauten, dass eine Person die Zahlen 1 und 2 addieren und das Ergebnis mit 3 multiplizieren soll. Auch ein Computer soll diesen Vorgang später durchführen können, und es soll ein Muster formuliert werden, mit dem dann auch andere Zahlen so verarbeitet werden können.

1. Problemstellung identifizieren und abstrakt modellieren
   Um die einfache Aufgabenstellung nicht unnötig zu verkomplizieren, sei direkt eine explizit abstrakt ausformulierte mathematische Formel notiert, die die einfache Berechnung mit den konkreten Zahlen löst:

   Nehme die Zahl 1 und addiere dazu die Zahl 2 und multipliziere dann mit der Zahl 3

   Beachten Sie, dass in dem Problem bzw. einer echten mathematischen Darstellung (1 + 2 * 3) die Punkt-Vor-Strichrechnung relevant ist – darauf wird gleich eingegangen.

2. In Teilprobleme oder -schritte zerlegen
   Die in Schritt 1 formulierte mathematische Formel ist das Resultat von 2 Teilaufgaben, die sich einzeln und nacheinander abarbeiten lassen. Man spricht hier von einer sogenannten **Sequenz** der Teilschritte.

   Wenn man diese Aufgabe also zerlegt, gibt es hier die folgenden zwei Teilschritte:

   Die Addition von den Zahlen 1 und 2 (1 + 2).
   Die Multiplikation des Ergebnisses der Addition mit dem Wert 3 (3 * 3).

Diese müssen nacheinander ausgeführt werden, da sich die Reihenfolge der Sequenz aus der Zerlegung der abstrakt formulierten mathematischen Formel in Teilschritte von links kommend ergeben hat.

Wir haben hier ein Beispiel für einen Algorithmus (die gesamte Berechnung) und auch die einzelnen Teilschritte sind (kleinere) Algorithmen, aus denen sich eben der gesamte Algorithmus zusammensetzt.

3. In der dritten Phase von „Computational Thinking" als theoretisches Konzept sollen Lösungsstrategien entworfen und ausgearbeitet sowie formalisiert so dargestellt werden, dass sie von einem Menschen als auch einem Computer verstanden und ausgeführt werden können.

   In diesem Beispiel macht die Notwendigkeit der Punkt-Vor-Strichrechnung in der echten mathematischen Formel deutlich, dass diese Darstellung der Aufgabe in Punkt 1 bereits für Menschen missverständlich ist oder nur unter Zuhilfenahme von „Metainformationen" korrekt gelöst werden kann (in der bisherigen mathematischen Darstellung müsste zuerst die Multiplikation ausgeführt werden). Auch Computer werden normalerweise eine sogenannte Priorität von Operatoren (Vorrang, in welcher Reihenfolge Dinge zu tun sind) beachten und dabei zuerst die Multiplikation und dann erst die Addition durchführen, wenn man den Vorgang als 1 + 2 * 3 formuliert (also erst 2 * 3 und dann 1 + 6).

   Wenn man hingegen die nachfolgende Darstellung wählt (unter der Voraussetzung, dass Klammern gültig sind und den Vorrang festlegen), ist das Problem eindeutig für Mensch oder Maschine korrekt zu lösen:

   (1 + 2) * 3

   Damit gibt man die Sequenz, in der die Teilalgorithmen auszuführen sind, sauber formalisiert, reproduzierbar und auf andere Situationen übertragbar an.

   Oder aber man schreibt explizit die einzelnen Schritte nacheinander bzw. untereinander hin.

Mit der Vorgehensweise hat man in jedem Fall ein Muster in dem gesamten Problem als auch in den Teilproblemen identifiziert und kann Standardlösungen für ähnliche Fälle anbieten und verwenden.

Die Modellierungs- und Problemlösungsprozesse sind dabei von einer Programmiersprache unabhängig.

Zum Abschluss soll nicht verschwiegen werden, dass die Definition des „Computational Thinking" von vielen Fachleuten stark kritisiert wird.

- ✔ Das Konzept des Computational Thinking wird oft als zu vage kritisiert, da selten klar gemacht wird, wie es sich von anderen Formen des Denkens unterscheidet.
- ✔ Einige Computerwissenschaftler machen sich Sorgen über die Förderung von Computational Thinking als Ersatz für eine breitere Informatikausbildung, da Computational Thinking nur einen sehr kleinen Teil des Feldes darstellt.
- ✔ Andere Forscher befürchten, dass der Schwerpunkt Computational Thinking Computerwissenschaftler dazu ermutigen könnte, bei der Lösung von Problemen viel zu eng vorzugehen und vor allen Dingen die sozialen, ethischen und ökologischen Auswirkungen der von ihnen geschaffenen Technologie zu ignorieren.

© HERDT-Verlag

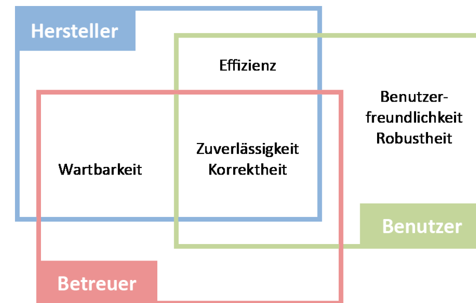
2.4 Qualitätskriterien und Dokumentation

An Software werden hohe Qualitätsansprüche gestellt. Die Software muss dabei den Anforderungen zweier Interessengruppen entsprechen – denen der Hersteller und denen der Kunden. Zur Erfüllung von Qualitätsanforderungen werden **Qualitätskriterien** festgelegt.

Der Kunde achtet bei der Software z. B. auf Benutzerfreundlichkeit und auf den Kundendienst einer Softwarefirma. Für ihn ist es meist wichtig, dass die Software zum bestellten Termin und zum vereinbarten Preis ausgeliefert wird. Außerdem spielt der Einarbeitungsaufwand eine große Rolle.

Dem Hersteller dagegen ist es wichtig, bei der Entwicklung und der nachfolgenden Wartung rentabel zu arbeiten. Sobald das Produkt ausgeliefert ist, geht die Verantwortung vom Hersteller auf den Betreuer der Software, meistens den Hersteller selbst, über. Dieser übernimmt die Wartung der Software, z. B. in Form des Kundendienstes.

Die nebenstehende Abbildung zeigt die Verbindung von Qualitätskriterien.



Korrektheit, Robustheit, Zuverlässigkeit

Die **Korrektheit** stellt bei allen Programmen ein Kriterium dar. Ein Programm ist dann korrekt, wenn es ein vorgegebenes Problem fehlerfrei löst. Die Beurteilung der Korrektheit basiert auf Anforderungen an das Programm, der Produktdefinition (oder dem Pflichtenheft). Die Problematik innerhalb dieses Kriteriums besteht darin, dass Sie nicht entscheiden können, ob ein Programm 100-prozentig korrekt ist. Der vollständige Test aller möglichen Programmzustände ist schon für kleine Programme nur mit erheblichem Aufwand durchführbar. Beim Programmieren sehr umfangreicher Programme scheitert jedoch dieses Verfahren.

Mathematisch gesehen können Sie meist entscheiden, ob ein Programm (die verwendeten Algorithmen etc.) 100-prozentig korrekt ist. Da die Ausführung von Software aber auch noch von anderen Bedingungen abhängt (Hardware, Betriebssystem), kann nicht davon ausgegangen werden, dass sie in Bezug auf diese Bedingungen korrekt ist (Anpassungen sind notwendig). Korrektheit ist somit eher ein theoretisches Maß.

Eine Software ist **robust**, wenn sie beispielsweise bei falschen Eingaben sinnvoll reagiert und bei Eingabefehlern nicht abstürzt. Dies kann von einer einfachen Fehlermeldung bis zu einer automatischen Fehlerkorrektur reichen.

Ein robustes Programm stürzt nicht ab, sondern reagiert auf mögliche Fehler durch eine geeignete Fehlerbehandlung.

Die **Zuverlässigkeit** ist das wichtigste Kriterium von Software bezüglich ihrer Fehlerhaftigkeit. Es kann davon ausgegangen werden, dass jede Software Fehler enthält und diese auch nicht auszuschließen sind. Um zu entscheiden, wie zuverlässig ein Programm ist, müssten die folgenden vier Fragen beantwortet werden:

- Wie oft tritt ein und derselbe Fehler auf?
- Wie viele unterschiedliche Fehler gibt es?
- Sind es Fehler, die bei der Programmierung hätten entdeckt werden müssen? Oder sind es selten auftretende Fehler, die nur schwer zu lokalisieren sind?
- Führt der Fehler zum Systemabsturz, Verlust von Daten oder zu einem falschen Ergebnis?

Software ist zuverlässig, wenn selten Fehler auftreten und diese nur geringe Auswirkungen haben. Zuverlässige Programme sind auch stets robust.

© HERDT-Verlag

Benutzerfreundlichkeit

Ist eine Software sowohl von erfahrenen als auch unerfahrenen Benutzern einfach zu bedienen, ist sie **benutzerfreundlich**. Wesentliche Aspekte dabei sind die Entwicklung von Benutzeroberflächen nach **software-ergonomischen Kriterien**, wie beispielsweise der übersichtliche Aufbau und der Inhalt der Bildschirmfenster. Hinweise zur Software-Ergonomie finden Sie in zahlreichen Publikationen. Die ständig verfügbare Hilfefunktion und die Möglichkeit, den Umfang von Funktionen an den Kenntnisstand des Nutzers anzupassen, sind weitere wichtige Gesichtspunkte.

Effizienz

Bei der **Effizienz** (Wirksamkeit und Wirtschaftlichkeit) wird zwischen der Laufzeit- und der Speicherplatzeffizienz unterschieden. Die Grundlage für effiziente Programme sind leistungsfähige Algorithmen und deren Verwendung in der jeweiligen Anwendung. Konzentrieren Sie sich während der Programmierung jedoch zu sehr auf diesen Gesichtspunkt, kann dies zulasten der Übersichtlichkeit gehen. Deshalb sollten Sie bereits eine Vorauswahl für effiziente Algorithmen treffen und diese so implementieren, dass sie einfach durch effizientere austauschbar sind.

Bei der Softwareentwicklung orientieren Sie sich zunächst meist an den anderen Qualitätskriterien. Erst wenn bei Tests der Software eine mangelnde Effizienz festzustellen ist, wird versucht, durch gezieltes Austauschen der Algorithmen eine Verbesserung zu erreichen.

Wartbarkeit

Immens wichtig für die Hersteller von Software ist die **Wartbarkeit**. Wartbarkeit umfasst den Aufwand bei der Fehlersuche, der Fehlerkorrektur und bei funktionalen Erweiterungen, der sich in den anfallenden Kosten widerspiegelt. Für den Benutzer spielt dieses Kriterium in Bezug auf die Softwarequalität jedoch keine direkte Rolle.

Dokumentation, Programmbeschreibung und Programmspezifikation

Softwareprogramme sollten gut dokumentiert werden. Dabei gibt es verschiedene Faktoren, die zu erfassen sind:

- Benutzerschnittstelle zum Anwender
- Systemschnittstelle zu anderen Produkten
- Kommentierter Quelltext für den Hersteller
- **Dokumentation** für die Anwender und den Hersteller

„Guter“ Programmierstil

Es ist eine Selbstverständlichkeit, dass ein Programm fehlerfrei sein muss und das macht, was es soll. Aber das ist nur eine Facette eines „guten“ Programms. Die Art und Weise, wie der Quellcode geschrieben ist, ist fast genauso wichtig. Eine Sprache wie Python zwingt Programmierer bereits an einigen Stellen zu einem „guten“ Programmierstil, weil sie

- streng Einrückungen einfordert,
- Groß- und Kleinschreibung unterscheidet,
- mehrere redundante oder überflüssige Syntaxelemente weglässt etc.

Doch auch in anderen Programmiersprachen kann man nach einem „guten“ Programmierstil vorgehen. Es gibt ein paar allgemeine Regeln, die man beim Schreiben von Quellcode einhalten sollte und damit auch einer modernen **Softwaredokumentation aus dem Quellcode heraus** Rechnung tragen:

- Quellcode soll selbsterklärend sein.
- Die Bezeichner von Variablen und Funktionen sollen für Menschen intuitiv verständlich sein.

© HERDT-Verlag

- Wo sich bereits die formale Programmiersprache selbst ausreichend erklärt und die Struktur durch geeignetes Einrücken bereits hinreichend deutlich wird, darf **keine** zusätzliche und unabhängige Beschreibung angefertigt werden.
- Eine Dokumentation soll so weit wie möglich in den Quellcode eingearbeitet sein. Das kann durch Kommentare und Kommentarzeilen erreicht werden, die in unmittelbarer Nähe der Anweisungen im Quellcode stehen und bei deren Veränderung sofort aktualisiert werden können. Viele Programmiersprachen stellen auch spezielle Dokumentationskommentare zur Verfügung, die automatisch von Dokumentationstools genutzt werden können.
- Das Einhalten von Konventionen ist extrem wichtig. Das betrifft sowohl die Namensgebung an sich, die Formatierung des Codes, aber auch Groß- und Kleinschreibungen, auch wenn diese von einer Sprache nicht gefordert werden.
- Man sollte nach dem **DRY**-Prinzip arbeiten (Don't repeat yourself, englisch für „Wiederhole dich nicht"; auch bekannt als once and only once – „einmal und nur einmal"). Redundanz ist dabei zu vermeiden oder zumindest zu reduzieren. Das erhöht die Wartbarkeit, denn eventuelle Änderungen brauchen nur an einer Stelle vorgenommen zu werden, und man vermeidet Tipparbeit. Zudem ist so ein Code klarer und weniger fehleranfällig.
- Konventionen und klare, übersichtliche Formatierungen sollten damit einhergehen, dass Codestrukturen selbst immer möglichst einfach und kurz bleiben. Es gibt eine Regel, die als **KISS**-Prinzip bekannt ist. Das steht für „Keep it simple, stupid" (englisch für „Halte es einfach und dumm"). Das ist kein Widerspruch zu einer intelligenten Programmierung – im Gegenteil. Denn „intelligent" bedeutet in der Programmierung einfach und langweilig. Und damit ist die Erstellung des Codes erst einmal wenig fehlerträchtig, aber der Code ist auch leichter wartbar, was mittel- bis langfristig genauso wichtig ist. Einfach bedeutet, dass Sie nicht unnötig komplexe oder lange Strukturen schreiben sollten. Lieber schreiben Sie zwei kurze Funktionen als eine lange Funktion. Wie gesagt, langweilige Programmierung ist fast immer gut. Doch was soll das bedeuten? Als recht leicht merkbare Regel können Sie sich merken, dass Sie immer so programmieren sollten, dass sich jemand Fremdes über Ihren Code nicht wundern würde. Erwartbare Strukturen sind gut, unerwartete Strukturen schlecht. Wenn Sie etwa an einer Stelle eine `while`-Schleife verwenden und in einer vergleichbaren folgenden Situation eine `for`-Schleife, dann ist das unerwartet und schlecht, wenn es keinen unvermeidbaren Grund gibt. Ein erfahrener Programmierer, der den Code sieht, würde sich fragen, warum einmal die eine Schleife und das andere Mal die andere Schleife verwendet wurde. Und dies vollkommen überflüssig, wenn es eben keinen zwingenden Grund gibt. Oder wenn Sie einmal eine Konstante vollkommen groß schreiben und das andere Mal nur den ersten Buchstaben groß schreiben, ist das unerwartet und damit schlecht (was wieder auf die Konventionen zurückkommt). Python erzwingt oder unterstützt zumindest durch seine reduzierten Syntaxstrukturen ohnehin an vielen Stellen die Einhaltung des KISS-Prinzips.

Beschreibung und Dokumentation

Mit einer **Programmbeschreibung** wird bei einer Software beschrieben, was das Programm **macht**. Mit der **Programmspezifikation** wird vorgegeben, was es **machen soll**. Beide Elemente bilden den Kern der **Software-dokumentation**. Diese erklärt eine Software aus unterschiedlicher Sicht. Es gibt in der Regel spezielle Dokumentationen für die Entwickler oder die Anwender, wobei verschiedene Dokumentationsteile nur den Entwicklern zugänglich sind, während andere für die Anwender verfügbar sein müssen.

Man beschreibt darin beispielsweise,

- wie ein Programm funktioniert (Programmiererdokumentation),
- wie ein Programm über die Zeit gepflegt wird (Entwicklungsdokumentation),
- was ein Programm für Ausgaben erzeugt und welche Eingaben notwendig sind (Datendokumentation),
- wie ein Programm getestet wurde (Testdokumentation),
- wie ein Programm zu benutzen ist (Benutzerdokumentation),
- was zum Betrieb eines Programms erforderlich ist (Installationsdokumentation) oder
- auf welchen Grundlagen die Software entwickelt wurde (Methodendokumentation).

Einige Dokumentationsteile erfüllen neben technischen auch juristische Zwecke und können Vertragsbestandteile im Rahmen eines **Pflichtenhefts** sein. Diese werden auch und im Fall von Gewährleistungsansprüchen berücksichtigt.

Bei der Dokumentation für eine Software werden zwei Arten unterschieden:

- Systemdokumentation
- Benutzerdokumentation

Systemdokumentation

Sie beschreibt die Eigenschaften der Software vom Beginn der Entwicklung bis zur letzten Version. Die Systemdokumentation enthält die Definition des Produkts, die Spezifikation der Software, die exakten Beschreibungen der Strukturen. Diese Dokumentation ist sehr detailliert beschrieben und bildet die Grundlage für die Wartung der entsprechenden Software.

Benutzerdokumentation

Hier findet der Anwender ein Handbuch zur Einführung in die Softwarenutzung sowie ein komplettes Nachschlagewerk zu allen angebotenen Funktionen. Sollte es notwendig sein, muss ein gesondertes Handbuch für den Systemadministrator, beispielsweise für die Systeminstallation, in der Benutzerdokumentation integriert sein. Die Nutzungsbedingungen sind ebenso einzufügen wie die Angaben zu den Systemvoraussetzungen, unter denen das Programm effizient arbeitet. Angaben zu den Autoren der Dokumentation bzw. dem Hersteller der Software sind ebenfalls Bestandteil einer guten Dokumentation.

Eine Dokumentation muss klar verständlich sein, der aktuellen Programmversion entsprechen, einen logischen Aufbau besitzen und sachlich präzise geschrieben sein. Ungebräuchliche Bezeichnungen sind in einer Dokumentation zu vermeiden bzw. sollten bei Gebrauch näher erläutert werden.

Zunehmende Qualitätsanforderungen

Software wird heute in vielen Bereichen eingesetzt, in denen ein Ausfall kritisch ist. Softwarefehler können verheerende Folgen haben. In sensiblen Bereichen steht die Forderung nach der Null-Fehler-Software. Trotz Anwendung wissenschaftlicher Ergebnisse bei der Qualitätssicherung können diese Anforderungen (noch) nicht erfüllt werden. Das ergibt sich auch aus der Tatsache, dass umfassende Tests neu entwickelter Software mitunter sehr schwierig und teuer sind.

Zwei Typen der Softwaredokumentation

Zusammenfassend kann die Softwaredokumentation wie folgt unterschieden werden:

- Die **Projektdokumente** beschreiben, was von den Personen zu tun ist, die an der Entwicklung beteiligt sind, und warum das zu tun ist. Das umfasst beispielsweise die Ziele und Anforderungen, die Methodik, den Zeitrahmen (Planungsdokumente) und die Festlegung, womit die Entwicklung zu erfolgen hat.
- Die **Systemdokumente** beschreiben, woraus das System besteht, was es tut (Funktionen), was es erzeugt (Ergebnisse), welche Daten es verarbeitet, wie es zu bedienen ist etc.

© HERDT-Verlag

2.5 Warum programmieren?

Bestehende Programme decken viele Einsatzgebiete ab, wie z. B. die Verwaltung von Geschäftsprozessen, die Textverarbeitung, die Tabellenkalkulation, Spiele usw. Warum sollten Sie noch selbst programmieren?

Standard-Software anpassen

Standardisierte Software, die für ein bestimmtes Einsatzgebiet konzipiert wurde, stößt an ihre Grenzen, wenn spezielle Anforderungen an sie gestellt werden bzw. wenn die Anwendungsgebiete erweitert werden.

Beispiel

Sie haben sich mit einem Programm eine Datenbank mit Ihren Kundendaten aufgebaut. Später müssen Sie auch mit anderen Programmen auf diese Datenbank zugreifen. In diesen Programmen ist keine Möglichkeit vorgesehen, auf die Daten Ihrer Kundendatenbank zuzugreifen. Sie müssten also die Daten wieder manuell eingeben. Oft ist es dann einfacher, wenn Sie ein Programm schreiben, das den Zugriff auf die Daten ermöglicht.

Individual-Software

Hierbei handelt es sich um Software, die eigens für einen bestimmten Anwendungsbereich oder für spezielle Abteilungen innerhalb einer Firma erstellt wird.

Für verschiedene Branchen und Einsatzgebiete gibt es spezielle Anforderungen, sodass nur eine genau auf die Bedürfnisse abgestimmte Software infrage kommt. Individual-Software kann, auf Kundenwunsch hin, erweitert und verändert werden.

2.6 Klassifizierung von Programmiersprachen

Im Gegensatz zu natürlichen Sprachen, z. B. Deutsch, Englisch, gehören **Programmiersprachen** zu den **formalen Sprachen** (künstlichen Sprachen).

Programmiersprachen lassen sich nach verschiedenen Kriterien einordnen. Nach ihrer **historischen Entwicklung** werden verschiedene Generationen unterschieden:

- Erste Generation: Maschinensprachen (Maschinencode)
- Zweite Generation: Assembler-Sprachen
- Dritte Generation: Prozedurale Sprachen
- Vierte Generation: 4GL (**G**eneration **L**anguage)
- Fünfte Generation: Künstliche Intelligenz

Programmiersprachen unterscheiden sich erheblich in der zugrunde liegenden **Programmiertechnik**, auch Programmierparadigma genannt. Nach Programmiertechniken und Konzepten können die Programmiersprachen wie folgt klassifiziert werden, wobei diese Einteilung weder strikt zu trennen noch zwingend ist:

- Prozedurale und funktionale Programmiersprachen
- Objektorientierte Programmiersprachen
- Hybride Programmiersprachen
- Skriptsprachen
- Logische Programmiersprachen
- Erziehungsorientierte Programmiersprachen und Minisprachen