Jürgen Plate

Algorithmen und Datenstrukturen

Supplement zum Buch:
Jürgen Plate
Der Perl-Programmierer
Erschienen im Hanser-Verlag

Inhaltsverzeichnis

1

Informationsverarbeitung

Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vaccuum tubes and perhaps weigh 1.5 tons.

Popular Mechanics, März 1949

1.1 Einführung

Neben der Energie und der Materie ist Information eine weitere Basisgröße von universeller Bedeutung. Ihre Eigenständigkeit wurde erst spät erkannt, da ihre Weitergabe immer an energetische oder materielle Träger, genannt „Nachricht", gebunden ist. An Informationen gelangt man also über Nachrichten. Die Definition von Information in der Informatik deckt sich dabei nicht ganz mit dem umgangssprachlichen Gebrauch.

- Information ist die Kenntnis von Irgendetwas.
- Nachrichten dienen zur Darstellung von Information.

Anders formuliert: Die abstrakte Information wird durch die konkrete Nachricht mitgeteilt.

Nachrichten und Informationen sind nicht identisch, insbesondere kann die gleiche Nachricht (mit gleicher Information) auf verschiedene Empfänger unterschiedliche Wirkung haben. Es gibt aber auch Nachrichten, die subjektiv keine Information enthalten. Dazu ein Beispiel:

Bei welcher Nachricht ist die Information größer?

- Am 1. Juli war die Temperatur höher als 25 Grad.
- Am 1. Juli betrug die Temperatur 29,2 Grad.

Bei der ersten Nachricht gibt es nur zwei Möglichkeiten (kleiner oder größer 25 Grad), bei der zweiten sind theoretisch beliebig viele Möglichkeiten gegeben – je nach Genauigkeit der Messeinrichtung. Also ist dort die Information größer.

Die Fachgebiete, die sich mit der Verarbeitung von Information befassen, Informatik und Softwaretechnik, sind recht junge Ingenieurswissenschaften. Die meisten Informatik-Fachbereiche entstanden erst nach 1980, und der Weg vom Probieren und Basteln hin zu einer systematischen Form der Softwareerstellung ist längst noch nicht abgeschlossen. Beim Softwareentwickler ist oftmals noch die Versuchung riesengroß, einfach drauflos zu programmieren.

Die zu lösende Problemstellung, sprich die Kundenanforderungen, werden oftmals ungenügend und unvollständig erfaßt. Anstelle einer wohlüberlegten technischen Lösung in Form eines zeitgemäßen Softwareentwurfs stürzt sich der Programmierer auf das, was eigentlich erst sehr viel später kommt: das Schreiben von Programmcode. Es wird wild drauflos programmiert und manchmal das Programm, sobald es zum ersten Mal funktioniert, auf die Seite gelegt und in „Rohfassung“ dem Kunden ausgeliefert. Das Erwachen folgt in der Regel schnell: Der Kunde fühlt sich mißverstanden und mißbraucht. Er bekommt eine Software, die oftmals nicht das tut, was er wirklich braucht, oder zu ineffizient ist, und er fühlt sich als Versuchskaninchen, weil die Qualität nicht stimmt und er unfreiwillig zum Fehlerfinder wird. Das Geld ist ausgegeben, und Änderungen bzw. Erweiterungen an der Software sind in diesen Fällen am besten nur durch Wegwerfen und Neuerstellung machbar. Ein Zustand, der auf Dauer nicht tragbar (und auch nicht finanzierbar) ist.

In den vergangenen Jahren wurden beachtliche Fortschritte in der Methodik der Softwareerstellung gemacht. Es entstand die Disziplin des Softwareengineerings (Softwaretechnik), die auch von wissenschaftlicher Seite erforscht und behandelt wird. Prägend in den letzten zwei Dekaden im Bereich Programmiertechniken waren Ideen, wie die der strukturierten Programmierung, die sich später zur modularen Programmierung wandelte, und seit Anfang der neunziger Jahre die der objektorientierten Programmierung. Aber heute kann festgestellt werden: Keine brachte letztendlich die ultimative Lösung. Wenn man heute Programme analysiert, manchmal sogar aus führenden Firmen, so sind sie oftmals kaum besser als das, was man bereits früher als Bastelei abqualifiziert hat.

Woran liegt das? Es gibt keine einfache Antwort auf diese Frage. Einerseits sind z. B. strukturierte Programmiersprachen heute wenig gefragt, außer vielleicht in Ausbildungsstätten. Die Hardware hat in ihrer Leistung und Speicherkapazität in den letzten 10 Jahren derart enorm zugenommen, dass dieser Rückschritt in der Software scheinbar kompensiert wird. Die Rechner wachsen, haben also für komplexere und dickere Software Platz. Es ist nicht zu kritisieren, dass die Software umfangreich ist, sondern dass sie unübersichtlich und somit fehlerbehaftet ist. Man weiß, wie es besser zu machen ist, aber man tut es nicht. „Es lohnt sich nicht, saubere Software zu schreiben“, so bekommt man zu hören, „es erfordert zuviel Denkarbeit von hochbezahltem Personal“.

Ein Softwareentwickler ist ein „Handwerker“, kein Künstler; und es gilt nach wie vor: „Übung macht den Meister“. Sinn dieses Buchs ist es daher unter anderem, Ihnen die für eine erfolgreiche Softwareentwicklung notwendige Denke und das grundlegende Programmierhandwerk unter Verwendung der Programmiersprache Perl nahezubringen.

Dazu gehört auch,

- dass Sie nachdenken, bevor Sie in die Tasten hacken.
- dass Sie sich einen halbwegs klaren Stil aneignen.
- dass Sie lernen, Fallstricke und Unsauberkeiten der Sprache richtig zu behandeln.

1.1.1 Informationsverarbeitung

Nachrichten übermitteln uns Informationen, Daten enthalten sie. Diese können wir aufnehmen oder auch (als unwichtig) ignorieren. Oft sind die Datenmengen so immens, dass

wir sie nur aufnehmen können, wenn sie entsprechend aufgearbeitet, z. B. reduziert sind. Daten sollen bei Bedarf abrufbar gespeichert sein und an beliebige Stellen transportiert werden können. Wir bedienen uns für diese Aufgaben oft der Hilfe von Maschinen und Computern. Diese müssen in der Lage sein, unsere Wünsche der Datenaufbereitung zu erfüllen. Das bedeutet aber, dass wir die Fähigkeit haben müssen, ihnen unsere Wünsche (Anweisungen) auch mitzuteilen. Es muss eine Art Sprache zwischen Mensch und Maschine bestehen, die beide verstehen. Solche Anweisungen können nicht nur zur Verarbeitung von Daten dienen, sondern ebenfalls für zahlreiche Tätigkeiten auch im Sinne von Automaten und Robotern.

Zu Beginn wollen wir uns vor allem mit einem praktischen Aspekt der Informationsverarbeitung auseinandersetzen. Sie lernen kennen, wie man Abläufe darstellt und wozu eine Programmiersprache wie Perl dient. Hierzu gehört auch das Erfassen und Darstellen von Daten und zugehöriger Datenstrukturen und programmtechnischer Ablaufstrukturen.

Für diejenigen, die den Computer bisher nur als Arbeitsmittel verwendet haben, ohne sich dabei Gedanken über seine Arbeitsweise zu machen, möchte ich an dieser Stelle einen kleinen Ausflug in die Hardware-Grundlagen unternehmen. Wenn Sie nämlich wissen, wie so eine Maschine funktioniert, wird Ihnen auch verständlich, warum Programme und Programmiersprachen so sind, wie sie sind.

1.1.2 Der Computer und seine Fähigkeiten

Es mag Sie erstaunen, dass in den folgenden Kapiteln kaum je von Computern gesprochen wird, obwohl es doch um das Programmieren von Computern geht. In der Regel braucht ein Programmierer heute kaum noch etwas darüber zu wissen, was im Detail im Computer vor sich geht. Eine modellhafte Vorstellung genügt in den meisten Fällen. Auch der normale Autofahrer braucht keine technischen Einzelheiten des Motors zu kennen. Doch sollte man einige grundlegende Dinge verstanden haben, damit man sein neues Auto nicht in der ersten Kurve verschrottet. Ähnlich ist es, wenn man mit dem Computer sicher umgehen will.

In diesem Abschnitt werden einige Anwendungsfälle aufgezählt, die Arbeitsweise von Mensch und Computer verglichen und die Frage gestellt, wie der Computer zu seinen Fähigkeiten kommt.

Nehmen Sie als Beispiel moderne Kaufhauskassen. Sie schreiben jeden Betrag, mit Erklärung versehen, auf den Kassenbon. Das kann natürlich jede alte Registrierkasse auch. Was ist also Besonderes an den Kaufhauskassen? Diese Kassen sind mit einem zentralen Computer vernetzt und beziehen von dort beispielsweise die Artikelbezeichnung und den Preis, wenn der Strichcode auf dem Warenetikett gescannt wird. Der Zentralrechner kann seinerseits ermitteln, wie viel von jedem Artikel im Lauf des Tages verkauft wurde. Der Rechner kann am Abend nicht nur die Kassenabrechnung erledigen, sondern auch ausdrucken, welche Regale im Verkaufsraum aus dem Lager nachgefüllt werden müssen. Gleichzeitig können Bestellisten für die Lieferanten erstellt werden, ebenso Absatzstatistiken und vieles mehr. So gab es einen schrittweisen Wandel von der einfachen Kasse hin zu einen Warenwirtschaftssystem.

Am nächsten Beispiel, der Lohnabrechnung, werden Sie sehen, worin der Unterschied liegt zwischen dem, was ein Mensch tut, und dem, was der Computer macht. Schauen Sie sich zuerst die Arbeit eines Lohnbuchhalters an. Er stellt zuerst anhand der Stundenabrechnungen und Stempelkarten fest, wie viele Stunden der Arbeitnehmer gearbeitet hat. Die Stunden werden addiert, um die Gesamtarbeitszeit zu erhalten. Diese Zeit wird mit dem Stundenlohn multipliziert, und dann werden Steuern und Sozialabgaben abgezogen. Dabei muß der Buchhalter einige Fakten aus der Personalkartei heraussuchen - Kinderzahl, Familienstand, Steuerklasse, Versicherungsnummer etc. Ist die Berechnung beendet, werden

Lohnstreifen und Überweisungsformulare ausgefüllt und an die richtigen Stellen weitergeleitet.

Der Buchhalter erhält also gewisse Informationen, nämlich Arbeitszeit, Steuerklasse, Stundenlohn usw. Er verarbeitet dann diese Informationen nach einem bestimmten Schema und gibt schließlich neue Informationen, z. B. den Betrag des Nettolohns an andere weiter. Der Informationsfluss lässt sich also durch die Abfolge **Eingabe — Verarbeitung — Ausgabe** skizzieren.

Genauso arbeitet auch der Computer: er erhält aus einer Eingabestation die Daten über die Anwesenheitszeiten des Arbeitnehmers. Bereits das Addieren der einzelnen Zeiten wird „vom Programm“ übernommen, das dazu ebenfalls eingegeben werden mußte. Die eigentliche Verarbeitung erfolgt in der Zentraleinheit des Computers. Sie besteht im Wesentlichen aus drei Teilen: dem Arbeitsspeicher, der Recheneinheit und der Steuereinheit. Heutzutage sind all diese Komponenten in wenigen integrierten Schaltkreisen der Hauptplatine Ihres PCs enthalten. Für die Speicherung umfangreicher Datenmengen stehen externe Magnetplattenspeicher zur Verfügung, wobei die Verwaltung der Daten oft durch eine Datenbanksoftware unterstützt wird.

1.2 Computersysteme

Computer sind somit Systeme, die aus Hardware und Software bestehen. Jede der beiden Komponenten ist ohne die andere nichts wert. Die von Ihnen geschriebenen Programme nehmen, damit sie ablaufen, sowohl die Hardware als auch die Software des Computers in Anspruch. Die Hardware-Komponenten umfassen alle „anfaßbaren“ Komponenten. Dazu zählt der Computer selbst und alle Geräte, die daran angeschlossen sind: Bildschirm, Drucker, Barcodescanner, DVD-Laufwerk usw. Die Software ist „nicht anfaßbar“. Sie benötigt zum Transport, zur Speicherung und zum Funktionieren immer irgendwelche Hardware. Software läßt sich grob in das sogenannte Betriebssystem und die Anwendersoftware unterteilen.

In der zuerst von John von Neumann 1947 angegebenen Modellvorstellung eines Rechners besteht dieser aus vier Funktionseinheiten (Bild 1.1):

- die Recheneinheit,
- die Steuereinheit,
- dem Speicher (Memory) und
- die Ein- und Ausgabegeräte (E/A-Geräte, engl. *Input/Output Units* oder I/O).

Recheneinheit und Steuereinheit werden oft als Zentraleinheit oder Central Processing Unit (CPU) zusammengefasst. Die CPU ist das „Herz“ eines Computers. Sie bearbeitet Programme, führt Berechnungen aus und steuert die Vorgänge in den anderen Teilen. Der Speicher nimmt alle Daten auf, die der Computer braucht. Das reicht von Daten, die der nächste Programmschritt benötigt, bis hin zu ganz selten gebrauchten Daten. Schließlich stellen E/A-Geräte die Verbindung her zwischen dem Computer und seinem Anwender.

1.2.1 Zentraleinheit

Die CPU besteht aus mehreren Teilen, die eng zusammenarbeiten: Recheneinheit und Steuereinheit. Sie enthält mehrere Register für die Aufnahme der Daten, die bei der Ausführung eines Programms verfügbar sein müssen. So nimmt z. B. das Befehlsregister den jeweils auszuführenden Befehl auf. Ein Register kann man sich als schnellen (Zwischen-)Speicher für ein einzelnes Datenwort vorstellen. Nur der anstehende Befehl steht im Befehlsregister,

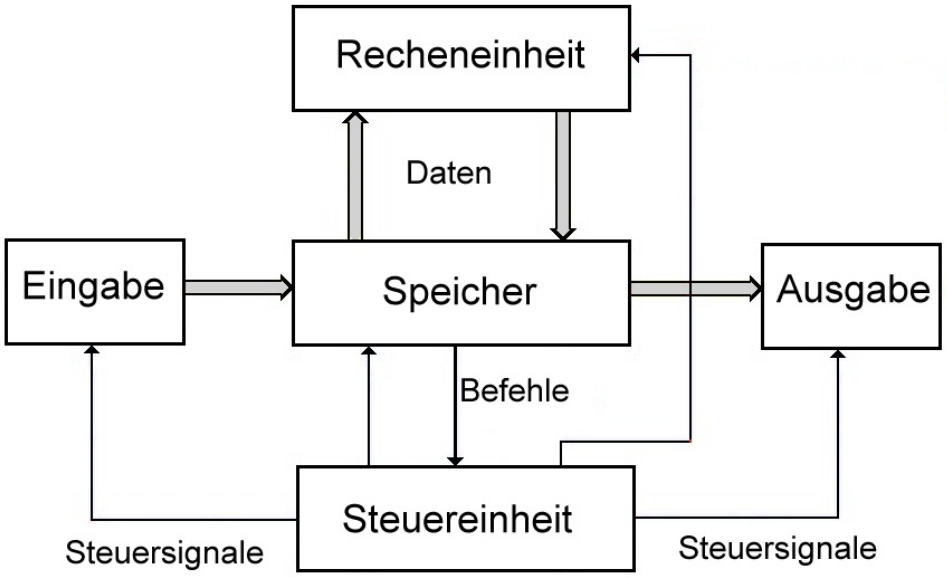

Bild 1.1: Aufbau eines Von-Neumann-Rechners

daher gibt es - von der Ausführung her gesehen - keinen Unterschied zwischen langen und kurzen Programmen oder zwischen schwierigen und einfachen. Die Befehle enthalten die Informationen über die Speicherplätze der benötigten Daten; bei der Ausführung des Befehls werden die Daten bereitgestellt und nach der Bearbeitung wieder abgelegt.

- Die **ALU** (Arithmetic and Logic Unit) der **Recheneinheit** führt alle Berechnungen aus und trifft logische Entscheidungen. Ihre Hauptaufgabe ist es, Vergleiche zwischen Werten vorzunehmen und damit Bedingungen zu überprüfen. Die Rechenfähigkeit beschränkt sich auf die Grundrechenarten, doch lassen sich daraus alle anderen arithmetischen Operationen aufbauen. Die komplizierten Berechnungen, die der Computer in kurzer Zeit ausführen kann, setzen sich aus einer Vielzahl kleinster Schritte in der ALU zusammen. Die Wortbreite der ALU, also die Anzahl der parallel mit einer Rechenoperation verarbeitbaren Bits, ist auch ein Maß für die Leistungsfähigkeit der CPU – heutzutage haben wir es mit 32-Bit- oder 64-Bit-CPUs zu tun. Moderne CPUs besitzen neben einer ALU für ganze Zahlen auch eine Fließkomma-Unit (Gleitkomma-Unit), die speziell für die Berechnung reeller Zahlen ausgelegt ist. Dabei müssen wir uns jedoch immer vor Augen halten, dass es sich nicht wirklich um reelle Zahlen handelt, da die Anzahl der verarbeitbaren Stellen begrenzt ist. So treten bei jeder Operation Rundungsfehler auf, die durch geeignete Programmierverfahren so klein wie möglich gehalten werden müssen.
- Die **Steuereinheit verarbeitet die elementaren Befehle des Computers (die sogenannten Maschinenbefehle). Sie liest Befehl für Befehl aus dem Arbeitsspeicher, decodiert ihn und führt den Befehl dann aus. Die Steuereinheit und die Recheneinheit arbeiten Hand in Hand, die Steuereinheit liefert die Signale zur Steuerung der gewünschten ALU-Funktion. Die ALU liefert umgekehrt Informationen an die Steuereinheit, die dieser logische Entscheidungen erlauben (z. B. ob das Ergebnis der letzten Rechenoperation 0, <0 oder >0 ist). Auch stellt die Steuereinheit die Verbindung zwischen allen Komponenten des Computers her, sie ist die Schaltzentrale, die alle Datenströme lenkt.**
- **Der Speicher nimmt alle Daten auf, dazu gehören die Programme ebenso wie die von ihnen zu verarbeitenden Werte. Bei der Verarbeitung werden Zwischenergebnisse im**

Speicher abgelegt und schließlich die Endergebnisse. Man unterscheidet beim Speicher zwischen dem Hauptspeicher (oder Arbeitsspeicher) und dem externen Speicher. Der Hauptspeicher ist fest im Computer eingebaut, und man kann ihn bis zu einer hardwaremäßig festgelegten Grenze erweitern. Im Hauptspeicher befindet sich das Programm, das Sie schreiben oder verändern wollen oder das der Computer bearbeiten soll. Auch die Daten, die beim Ablauf des Programms benötigt werden, werden dort abgelegt. Auf den Hauptspeicher kann die CPU direkt zugreifen, sie kann von dort neue Programmbefehle holen und kann Daten laden oder abspeichern.

- **Im externen Speicher werden die Programme und Werte aufbewahrt, die man zur Zeit nicht benötigt. Als Datenträger werden Halbleiterspeicher (USB-Stick, SD-Karte etc.), Festplatten und optische Speichermedien (CD-ROM, DVD) und manchmal noch Disketten (Floppy Disks) verwendet. Für das Abspeichern von Programmen vom Hauptspeicher auf die Festplatte oder Diskette gibt es besondere Kommando des Betriebssystems, ebenso für das Laden von einem Massenspeicher in den Hauptspeicher.**
- **Die Eingabe- und Ausgabegeräte stellen die Verbindung her zwischen einem Computer und seinem Benutzer, aber auch zwischen einem Computer und anderen Geräten oder Computern. Ohne Geräte zur Ein- und Ausgabe könnte man keine Programme eingeben und keine Daten an ein laufendes Programm liefern, man erhielte keine errechneten Ergebnisse. Ein Ausgabegerät wie der Drucker läßt sich einsetzen, um die Ergebnisse der Programmbearbeitung zu erhalten oder um die Werte auszugeben, die in einem Speicher abgelegt sind. Ein Eingabegerät wird komplementär eingesetzt, mit ihm versorgt man die CPU oder den Hauptspeicher durch die CPU mit neuen Daten.**
- **Zu den E/A-Geräten gehören auch die Netzwerkverbindungen, mit denen sich mehrere Computer zu einem Verbund zusammenschließen lassen. Sie erlauben einen sehr schnellen Datenaustausch zwischen verschiedenen Computersystemen. Ein Vorteil solcher Vernetzung liegt darin, dass mehrere Systeme gemeinsam auf periphere Geräte wie Drucker oder Festplatte zugreifen können. Damit erweitern sich die Fähigkeiten der einzelnen Maschinen ohne entsprechende zusätzliche Kosten.**

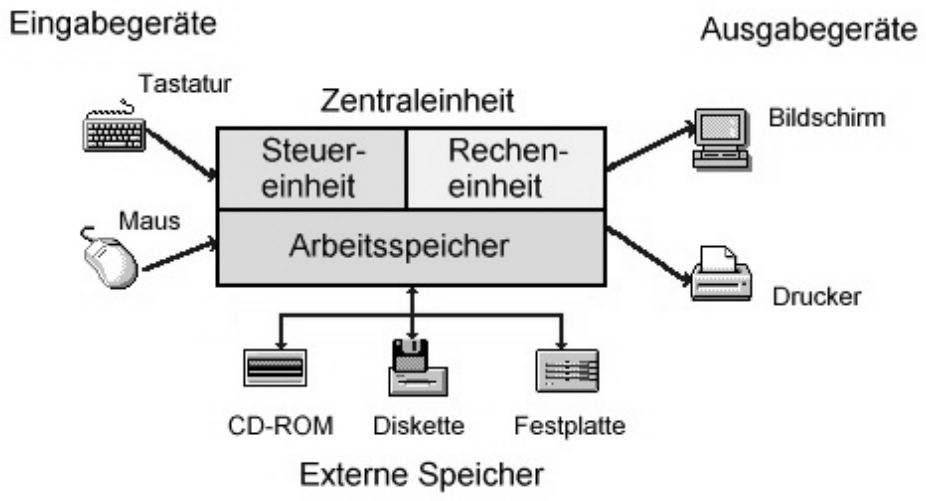

Bild 1.2: Schematischer Aufbau eines Computers

All diese Komponenten ergeben dann den kompletten Computer, wie er schematisch in Bild 1.2 dargestellt ist. Aber nicht immer ist ein Computer als solcher zu erkennen. In vielen Geräten des täglichen Lebens verbergen sich Computer, die das Gerät steuern und so

die frühere fest verdrahtete Steuerung ersetzen. Solche „Embedded Systems“ (eingebettete Systeme) finden Sie im Videorecorder genauso wie im Mobiltelefon oder zu Dutzenden im Auto.

1.2.2 Sichtweisen eines Rechners

Der Benutzer eines Rechners sieht von der Maschine je nach seinem Status einen mehr oder weniger eingeschränkten Teil.

- Der Anwender sieht den Rechner als das ihm zugängliche Anwendersystem, das genau auf seine Bedürfnisse zugeschnitten ist und mit dem er in einer für ihn bequemen Sprache Daten manipulieren kann.
- Der Anwendungsprogrammierer sieht den Rechner als das Betriebssystem, das ihm Compiler für verschiedene höhere Programmiersprachen zur Verfügung stellt und für ihn Dienstleistungen wie Editor und Dateiverwaltung bereithält. Wie das Betriebssystem realisiert ist, braucht auf dieser Stufe den Anwendungsprogrammierer nicht zu interessieren. Für ihn ist „der Rechner“ das Betriebssystem mit den Compilern.
- Der Systemprogrammierer sieht vom Rechner schon sehr viel mehr. Er kann alle Möglichkeiten der Hardware ausnützen und Programme als Befehlsfolgen für das Leitwerk schreiben. Er schreibt diese Befehle aber i.A. in einer für ihn lesbaren Form (Assemblersprache, siehe später), die noch durch ein spezielles Programm, den Assembler, in eine Folge von 0 und 1 gewandelt werden muß, die allein das Leitwerk des Rechners interpretieren kann. Für ihn ist Rechner und Assembler fast synonym.
- Der Hardwareingenieur beim Rechnerhersteller sieht die Hardware des Rechners in allen Einzelheiten und kennt den genauen Aufbau der Befehle, die sich z. T. aus Folgen einfachster Steuerkommandos (Mikroprogramm) zusammensetzen. Hier ist der interne Aufbau des Rechners von Interesse, und die technische Realisierung von Recheneinheit, Speicher und Steuereinheit tritt in den Vordergrund.

1.2.3 Embedded Systems

„Embedded Systems“ ist der englische Fachbegriff für eingebettete (Computer-)Systeme, die in der Regel unsichtbar ihren Dienst in einer Vielzahl von Anwendungsbereichen und Geräten versehen, z. B. in Flugzeugen, Autos, Kühlschränken, Fernsehern, Werkzeugmaschinen, Kaffeeautomaten, DVD-Playern oder anderen Geräten. In der Computerbranche werden solche Systeme auch gerne „Appliances“ genannt[1].

Wenn man den Begriff aufschlüsselt ergibt sich:

- Ein „System“ ist in diesem Kontext immer eine informationsverarbeitende Maschine.
- „Eingebettet“ bedeutet: für einen spezifischen Zweck in einer technischen Umgebung entworfen, eingebaut und betrieben.

Ein eingebettetes System interagiert mit seiner meist elektromechanischen Umwelt. Dabei ist kein menschlicher Benutzer vonnöten; er muss auch keinerlei Kenntnisse über die technischen Innereien des eingebetteten Systems haben, um das Gesamtsystem bedienen zu können.

„Embedded Systems“ vereinigen daher durch ihre oftmals sehr hardwarenahe Konstruktion die große Flexibilität von Software mit der Leistungsfähigkeit der Hardware. Die

[1] „appliance: A device or instrument designed to perform a specific function, especially an electrical device, such as a toaster.“

Softwareentwicklung für diese Systeme unterscheidet sich oft grundsätzlich von jener für Desktop- oder PC-Systeme: Oftmals werden Betriebssysteme eingesetzt, die zwar nicht über eine grafische Benutzeroberfläche verfügen, dafür jedoch Echtzeitanforderungen genügen.

Bevorzugte Programmiersprachen sind daher z. B. Assembler oder C. Bekannte Embedded-Betriebssysteme sind z.B. QNX, VxWorks, Windows CE, DOS, LynxOS, Nucleus, zunehmend auch spezielle Linux-Derivate, wenn die Echtzeitbedingungen keine dominante Rolle spielen. Einige Beispiele für Embedded Systems sind:

- Die Elektronik in einem Kaffeevollautomaten. Mit ihr wird beispielsweise die Kaffeemenge und -stärke sowie die Wassermenge reguliert. Die Kaffemaschine an sich ist kein datenverarbeitendes System.
- Auch die Elektronik in einem Videorecorder ist ein eingebettetes System, auch wenn der Videorecorder selbst ein (eingeschränkt) datenverarbeitendes System ist.
- Der Zündcomputer im Auto ist ebenfalls ein im Stillen wirkendes System, wobei in einem Fahrzeug der Oberklasse inzwischen einige Dutzend Computer werkeln.
- Die Steuerungen von NC-Werkzeugmaschinen (NC = Numeric Control) oder anderen industriellen Systemen sind fast immer Embedded Systems.
- Fast alle Peripheriegeräte in der Datenverarbeitung (Drucker, Scanner, Barcode-Leser, Router usw.) sind mit eigener Intelligenz zur Verarbeitung der Daten ausgerüstet.

Man setzt deshalb für die Definition des eingebetteten Systems voraus, dass

- die genaue Aufgabe des Systems vor der Entwicklung feststeht,
- immer nur ein einziges Programm abläuft und
- Hardware und Software eine funktionale Einheit bilden.

Kennzeichnende Merkmale von eingebetteten Systemen sind:

- Sie sind fester Bestandteil eines technischen Systems.
- Sie sind zweckbestimmt (im Gegensatz zum Universalrechner).
- Sie interagieren mit der Umgebung durch Sensoren und Aktoren.
- Sie reagieren meist in Realzeit.

Sekundäre Merkmale von embedded Systemen sind:

- Sie werden oft für Regelungs- und Steuerungsaufgaben vorgesehen.
- Sie sind häufig Massenware, Konsumgut, billig.
- Sie sind vielfach schlecht bzw. nicht wartbar und nicht erweiterbar.
- Sie sind manchmal auch sicherheitskritisch.

Eingebettete Systeme beanspruchen einen Marktanteil bei der Prozessorproduktion von ca. 98 %. Die restlichen 2 % dienen dem Aufbau von interaktiven Systemen wie z. B. Laptops, Desktop-Computern und Servern. Nahezu die Hälfte der gesamten Mikrocontroller-Jahresproduktion sind immer noch 8-Bit-Prozessoren. Bereits heute gibt es mehr eingebettete Systeme als Menschen auf der Welt. Deren Elektronik dient oft als Wegwerfartikel (z. B. RFID-Tags, Grusspostkarten).

Man setzt deshalb für die Definition des eingebetteten Systems voraus, dass

- die genaue Aufgabe des Systems vor der Entwicklung feststeht,
- immer nur ein einziges Programm abläuft und
- Hardware und Software eine funktionale Einheit bilden.

Anders als stationäre Systeme haben eingebettete Systeme besondere Anforderungen hinsichtlich Robustheit, Energieverbrauch und Speicherbedarf. Im Gegensatz zu einem Arbeitsplatzrechner wird ein eingebettetes System im Allgemeinen nicht „heruntergefahren", bevor man es ausschaltet. Es muss im Gegenteil damit zurechtkommen, dass ihm der Strom jederzeit und ohne Vorwarnung abgestellt werden kann. Bewegliche Teile wie Lüfter oder Festplatte sind ebenfalls meist nicht erwünscht.

1.2.4 Software

Damit kennen Sie nun in etwa den Hardwareaufbau eines Computers. Es stellt sich jetzt die Frage nach seinen Einsatzgebieten und nach der Herkunft seiner Fähigkeiten. Seine Universalität erklärt sich aus den sehr elementaren Maschinenbefehlen, der Rechnerarchitektur und der Tatsache, dass der Computer, bevor er überhaupt etwas tun kann, erst in einem Programm (Software) erklärt bekommen muß, was er zu tun hat. Es gibt eine riesige Vielfalt an Programmen, geschrieben für die unterschiedlichsten Aufgaben, die aber auf ein und derselben unveränderten Hardware lauffähig sind.

Die Software eines Computersystems kann grob in zwei Teile gegliedert werden: die Anwendersoftware und das Betriebssystem. Ein Anwenderprogramm ist für die Lösung einer ganz bestimmten Aufgabe geschrieben worden, der Anwender setzt es dann (und nur dann) ein, wenn er eine solche Aufgabe lösen will.

Wenn man allgemein von Software spricht, meint man meist Anwenderprogramme, also Programme, die eine spezielle Aufgabe lösen. Einige typische Anwenderprogramme sind

- Programme zur Textverarbeitung,
- Spielprogramme,
- Programme für Tabellenkalkulation,
- Lernprogramme und
- Programme für die Verwaltung von Dateien,
- Programme für die Verwaltung großer Datenmengen (Datenbanken),
- Programme zum Steuern von Maschinen,
- Programme zum Regeln industrieller Prozesse,
- Programmiersprachen-Compiler und vieles mehr.

Wenn Sie in eine Computerzeitschrift schauen, finden Sie viele weitere Beispiele für Anwenderprogramme. Und auch die Programme dieses Skriptums sind zu den Anwenderprogrammen zu zählen, jedes wird für die Lösung eines bestimmten Problems geschrieben.

Dagegen ist das Betriebssystem eine anwendungsneutrale Software. Was die Infrastruktur einer Stadt mit Straßen, Buslinien, Trambahnen, Telefonnetzen und Fernverkehrsanbindungen für den Einzelnen und die Geschäftswelt ist, stellt das Betriebssystem als komfortable standardisierte Umgebung für die Anwenderprogramme dar. Möchte man in der Stadt von einem Ort zum anderen Waren transportieren, so benutzt man mit dem Auto die bestehenden Straßen und baut keine neuen. Ähnlich ist es mit den Anwendungsprogrammen. Sie sind für eine spezielle Aufgabe geschaffen worden. Der Ersteller möchte

sich aber nicht unbedingt mit der Infrastruktur, der Hardware, auseinandersetzten. Das Betriebssystem wird ständig benötigt, um die Vorgänge in der Hardware zu steuern und um die Wechselwirkung zwischen dem Computer und seinen Anwendungsprogrammen zu koordinieren.

Die „nackte“ Hardware eines Computers hat nicht die Fähigkeiten, ein Anwenderprogramm zu bearbeiten. Auch wenn die CPU die Vorgänge in der Hardware auf einer niedrigen Ebene koordinieren kann, muss eine Verbindung zwischen dem Anwenderprogramm und der jeweiligen Hardware hergestellt werden. Die Bearbeitung des Programms erfordert den Einsatz verschiedener Eingabe- und Ausgabegeräte. Es ist zunächst extern gespeichert und muß in den Hauptspeicher geladen werden. Diese und viele weitere Abläufe werden vom Betriebssystem gesteuert. Es ist ein sehr umfangreiches Programm, das die Vorgänge in der Hardware steuert und koordiniert. Der Benutzer braucht sich nicht selbst darum zu kümmern, wie die einzelnen Schritte bei der Bearbeitung seines Programms intern ablaufen.

Wenn ein Computer nur ein einziges Programm zu bearbeiten hätte wie z. B. der Mikroprozessor in einer Waschmaschine, dann bräuchte er kein Betriebssystem. Die Aufgabe eines Betriebssystems ist es, eine Umgebung zu schaffen, in welcher verschiedene Anwenderprogramme ablaufen können. Es bietet verschiedene Möglichkeiten an, den Computer einzusetzen. Da das Betriebssystem unerläßlich ist, wird es vom Hersteller mitgeliefert und läßt sich kaum verändern.

Von den vielen Aufgaben des Betriebssystems sollten Sie einige kennen. Zum Betriebssystem gehört beispielsweise Software, das den Zugriff auf externe Speicher steuert und die gespeicherten Dateien verwaltet. Bei größeren Systemen und Netzwerken kontrolliert das Betriebssystem z. B. den Zugang zum Computer mit der Eingabe eines Paßwortes und organisiert die gleichzeitige Arbeit mehrerer Benutzer. Das Betriebssystem ist verantwortlich für das Speichern und Laden von Programmen und für die Zuteilung der benötigten Hilfsmittel. Aus der Sicht des Programmierers ist besonders wichtig, dass vom Betriebssystem her die Programmierumgebung geschaffen wird. Es liefert die Umgebung, in der Sie Programme schreiben können, und es steuert den Ablauf Ihrer Programme. Sie brauchen sich nicht um die Einzelheiten zu kümmern, das Betriebssystem nimmt Ihnen (fast) alles ab. Die Entwicklung immer leistungsfähigerer Betriebssysteme hat es ermöglicht, die Programmierarbeit immer komfortabler zu machen.

2

Softwareerstellung

The sooner you start to code,
the longer the program will take.

Roy Carlson

Kennen Sie die Geschichte des Herrn X., der sich in eine astronomische Vorlesung verirrt hatte? Obwohl die dargestellten Gedankengänge fremd und neu für ihn waren, meinte er doch verstehen zu können, wie die Astronomen ihre Teleskope einsetzten, um die Entfernung der Gestirne von der Erde zu ermitteln. Es erschien ihm auch verständlich, daß sie die relative Positionen der Sterne und ihre Bewegungen voraussagen konnten. Was er aber gar nicht begreifen konnte: Wie zum Teufel konnten die Astronomen die Namen der Sterne und der Sternbilder herausbekommen?

Manche Leute haben das gleiche Problem bei den Programmiersprachen: Sie halten eine Sprache für ein kompliziertes mathematisches Begriffssystem, das von den ersten Informatikern mit viel Glück entdeckt worden ist. Es handelt sich aber nicht um einen Code, den es zu knacken galt, sondern um eine künstliche Sprache, die von Menschen aus ersten bescheidenen Anfängen heraus immer weiter entwickelt und verbessert worden ist.

2.1 Was sind Algorithmen?

Sie haben mit Sicherheit im Augenblick, in dem Sie diese Zeilen lesen, eine ganze Reihe von Problemen. Diese können Sie in verschiedene Kategorien einteilen, beispielsweise

- berufliche und private Probleme
- kurzfristige und langfristige Probleme
- bedeutende und banale Probleme usw.

Sie können sie aber auch unter einem Aspekt sehen, der uns ganz besonders interessiert, nämlich mit Computern lösbare bzw. nicht lösbare Probleme. Hier interessieren wir uns für die erste der beiden Gruppen. Eine klare Trennungslinie zwischen beiden läßt sich im Allgemeinen nicht ziehen, denn Probleme, die gestern noch als unlösbar galten, werden morgen von Computern vielleicht gelöst. Es gilt aber ein Grundsatz:

Jedes Problem, dessen Lösung durch einen Algorithmus beschrieben werden kann, ist im Prinzip durch einen Computer lösbar.

Aus dieser Aussage können Sie zwei Schlüsse ziehen:

1. Ein Computer ist ein Werkzeug, welches Ihnen bei der Lösung gewisser Probleme hilft.
2. Ein Algorithmus ist eine Art Anleitung oder Vorschrift, wie man zu einem Problem eine Lösung findet.

Das Wort „Algorithmus“ wird im allgemeinen Sprachgebrauch kaum verwendet und ist Ihnen daher vermutlich auch nicht geläufig. Es geht zurück auf den arabischen Autor Al-Khowarizmi (ca. 825 n. Chr.), der ein Lehrbuch über Mathematik geschrieben hat. In der Informatik versteht man darunter eine Lösungsvorschrift. Wir wollen den Begriff „Algorithmus“ folgendermaßen definieren:

Ein Algorithmus ist eine eindeutige Beschreibung eines endlichen Verfahrens zur Lösung einer bestimmten Klasse von Problemen.

Zuviel auf einmal? Ich werde im nächsten Abschnitt auf die einzelnen Eigenschaften eines Algorithmus noch genauer eingehen. Zunächst soll der Begriff des Algorithmus jedoch mit „Leben“ gefüllt werden, damit Sie ganz konkrete Vorstellungen damit verbinden können.

2.1.1 Alltagsalgorithmen

Die Alltagswelt, die Sie umgibt, ist voller Algorithmen - Sie haben es bisher nur nicht gewußt. Wenn Sie beispielsweise eine Coladose öffnen, führen Sie mit dem Objekt „Coladose“ eine Folge koordinierter Bewegungsabläufe aus, die schließlich die gewünschte Lösung produzieren. Die Lösungsvorschrift hierzu haben Sie sich durch Lernen und Üben erworben und in Ihrem Gehirn abgespeichert. Sie tragen also den Algorithmus „Coladose öffnen“ zusammen mit unzähligen anderen Algorithmen in Ihrem Kopf herum. Eine ganze Reihe von Alltagsalgorithmen ist schriftlich formuliert, z. B. Bedienungsanleitungen oder Kochrezepte, aber auch Wegbeschreibungen in einem Wanderführer oder Gesetze. Andere Algorithmen sind in Form eines Bildes formuliert, z. B. die Anleitung zum Falten eines Papierfliegers, wie in Bild 2.1:

Die Algorithmen haben alle einige Gemeinsamkeiten. Sie bestehen aus kurzen und knappen Formulierungen, welche dem „Ausführenden“ gewisse Anweisungen geben, die zur Lösung seines Problems führen. Je komplizierter ein Problem ist, umso mehr Anweisungen wird man in der Regel benötigen. Da als Adressat dieser Anweisungen ein mit Vernunft begabter Mensch vorausgesetzt wird, ist es hinreichend, wenn man die Anweisungen in einem kurzen deutschen Satz formuliert. Verständnisschwierigkeiten bei deren Interpretation treten im Allgemeinen nicht auf. Der Mensch verfügt auch über ein gewisses Repertoire an Hilfsmaßnahmen, wenn unvorhergesehene Störungen auftreten. Dazu ein Beispiel:

Jeder von uns kann in einer Telefonzelle erfolgreich telefonieren (solange es noch Telefonzellen gibt). Nehmen wir an, wir hätten einen Gast von einem fernen Planeten, der perfekt Deutsch spricht und liest, der aber noch niemals eine Telefonzelle benutzt hat (vermutlich, weil seinesgleichen per Telepathie kommuniziert). Er soll nun von uns auf einem Blatt Papier eine exakte Handlungsanweisung für die Benutzung dieser Telefonzelle erhalten, damit er uns nach dem Shopping anrufen kann. Es wird sich also um ein Ortsgespräch handeln, und wir unterstellen, daß unser Gast einige Münzen in der Tasche hat und unsere Telefonnummer kennt. Außerdem setzen wir einige Randbedingungen als bekannt voraus, obwohl gerade diese in der Praxis oft zu großen Realisierungsproblemen führen, wie z. B. die Frage: „Was ist der Hörer und wie herum wird er gehalten?“.

Von Personen, die sich bisher mit der algorithmischen Lösung solcher Alltagsprobleme nicht beschäftigt haben, wird meist folgende Handlungsanweisung genannt:

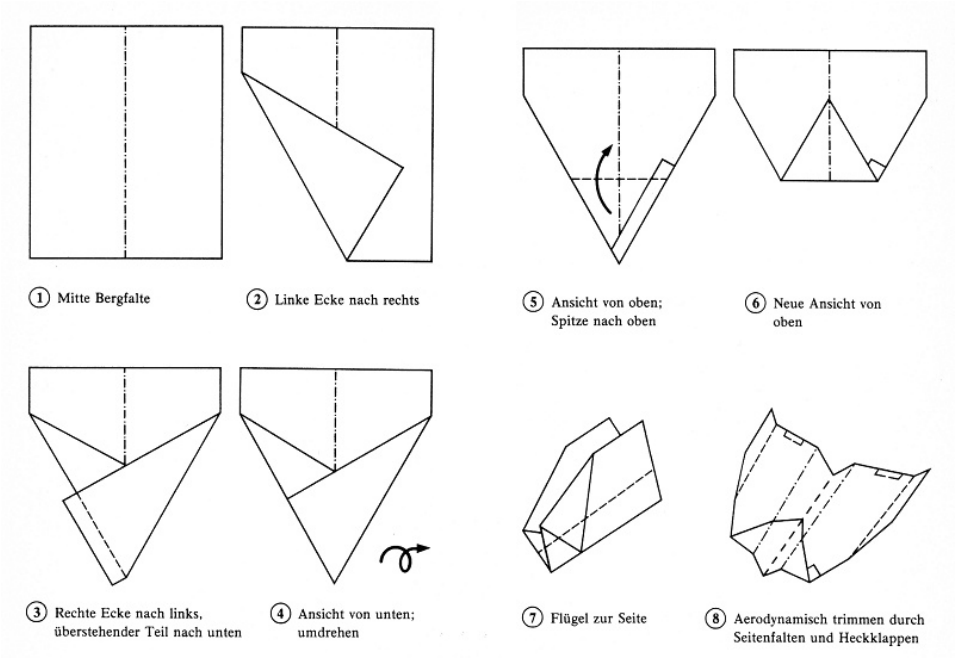

Bild 2.1: Falten Sie doch mal einen Papierflieger nach diesem Algorithmus

1. Nimm Hörer ab.
2. Wirf 20 Cent ein
3. Wähle ...

Bereits zwischen der ersten und zweiten Anweisung wurde übersehen, daß niemand von uns eine Münze einwirft, wenn er in dem zum Ohr geführten Hörer keinen Dauerton hört. Und schließlich würden wir nicht wählen, wenn eine der eingeworfenen Münzen durchgefallen wäre. Auch der jetzt naheliegende Schritt, daß man diese Münze dem Rückgabefach entnimmt und es mit ihr nochmals probiert, muß nach wenigen gleichartigen Versuchen ersetzt werden durch das Austauschen der offensichtlich ungeeigneten Münze. Denn ein derart „falsch“ programmierter Computer würde die gleiche durchgefallene Münze immer wieder erneut einwerfen, da er anders als der Mensch diese Handlungsweise niemals als unsinnig erkennen würde.

Tatsächlich lautet die korrekte Handlungsanweisung, nach der wir alle in der Praxis vorgehen:

1. Nimm Hörer ab.
2. Falls Dauerton vorhanden, dann wirf 20 Cent ein, sonst hänge Hörer ein und verlasse Telefonzelle.
3. Falls eine Münze durchfällt, dann prüfe, ob diese Münze schon einmal durchgefallen ist. Falls ja, tausche sie aus, sonst werfe sie erneut ein.
4. Wähle ...

Die Lesbarkeit dieser Handlungsanweisung läßt sich durch eine geeignete grafische Strukturierung, etwa durch Spiegelstriche oder -Punkte, verbessern:

1. Nimm Hörer ab.

2. Falls Dauerton vorhanden,
   - dann wirf zwanzig Cent ein,
   - sonst hänge Hörer ein und verlasse Telefonzelle.
3. Falls eine Münze durchfällt,
   - falls die Münze schon einmal durchgefallen ist,
     - dann wechsle sie aus,
     - sonst werfe sie erneut ein,
   - sonst wähle;
4. ...

Es ist offensichtlich, daß bei alltäglichen Handlungen, also auch bei berufsbezogenen Abläufen, ein großer Unterschied zwischen „tun können“ und „exakt beschreiben können“ besteht. Soll der Algorithmus jedoch von einer Maschine (einem Computer) ausgeführt werden, dann müssen diese Anweisungen viel präziser formuliert und alle möglichen Sonderfälle berücksichtigt werden. Auch wenn man selbst nicht programmiert, so ist diese Art zu denken eine wichtige Grundlage für eine sinnvolle Anwendung von Computern. Angeblich können nur Juristen und Programmierer einen Sachverhalt *präzise* beschreiben.

Eine weitere Gemeinsamkeit der aufgeführten Alltagsalgorithmen besteht darin, daß sie Anweisungen zur Manipulation von Objekten geben. Der Algorithmus oben beschreibt beispielsweise, wie die Objekte Hörer, Münze, Tastenfeld usw. zu manipulieren sind. Besonders deutlich tritt diese Eigenschaft bei Kochrezepten zum Vorschein. Als Beispiel könnte man die Zubereitung einer Gulaschsuppe nehmen.

2.1.2 Eigenschaften von Algorithmen

In der Definition des Begriffs „Algorithmus“ im vorausgegangenen Abschnitt kommen einige Adjektive vor, welche die Eigenschaften von Algorithmen beschreiben. Wir wollen Sie der Reihe nach betrachten.

- **Eindeutige Beschreibung**
  Bei der Verwendung einer natürlichen Sprache wie Deutsch, Englisch, Französisch usw. läßt es sich nicht vermeiden, daß gewisse Wörter und Sätze eine Mehrdeutigkeit beinhalten, d. h. dass man sie verschieden interpretieren kann. Denken Sie z. B. an folgende Wörter:
  - Stollen: Weihnachtsgebäck oder Gang im Bergwerk oder Teil eines Fußballschuhes.
  - Bund: Zusammenschluß oder Teil einer Hose.
  - Mutter: Weibliche Person mit Kind oder Teil einer Schraube.

  In einem Algorithmus dürfen solche mehrdeutigen Formulierungen nicht vorkommen. Der „Ausführende“ – ganz gleich ob Mensch oder Maschine – darf nie im Zweifel darüber sein, wie eine bestimmte Anweisung zu interpretieren ist.
- **Endliches Verfahren**
  Ein Algorithmus muß ein endliches Verfahren beschreiben (das Gegenteil wäre ein unendliches). Man meint damit, dass das durch den Algorithmus beschriebene Verfahren nach einer endlichen Zahl von Schritten eine Lösung produziert. Die genaue Anzahl hängt natürlich davon ab, wer den Algorithmus ausführt. Man sagt auch: „Das Verfahren muß terminieren“, also zu einem Abschluß kommen. Mit der Endlichkeit hängt ein weiterer Begriff zusammen: die Effizienz. Man sucht zur Lösung eines Problems einen

möglichst effizienten Algorithmus (eigentlich müßte man genauer sagen: einen Algorithmus, der ein möglichst effizientes Verfahren beschreibt).

Auf Computer bezogen bedeutet das zweierlei:

1. Die Ausführungszeit soll möglichst kurz sein.
2. Der Speicherplatzbedarf soll möglichst gering sein.

- **Bestimmte Problemklasse**
  Ein Algorithmus soll ein Lösungsverfahren für eine ganze Klasse von Problemen beschreiben und nicht nur für ein isoliertes Einzelproblem. Man sagt auch: „Das Verfahren soll allgemeingültig sein." Machen Sie sich diese Eigenschaft an einem ganz einfachen Beispiel klar. Nehmen Sie einmal an, Sie sollen die Zahlen 7, 12 und 16 addieren. Sie könnten das Lösungsverfahren folgendermaßen beschreiben:

  1. Addiere 7 und 12, nenne das Ergebnis SUMME1.
  2. Addiere 16 zu SUMME1, nenne das Ergebnis SUMME2.
  3. Schreibe als Lösung SUMME2 auf.

  Diese Lösungsvorschrift ist nach unserer Definition kein Algorithmus, da sie lediglich für ein ganz spezielles Teilproblem die Lösung liefert. Soll man beispielsweise die Zahlen 7, 12 und 17 addieren, dann kann man diesen Algorithmus nicht mehr verwenden. Ein erster Schritt zur Verallgemeinerung besteht darin, statt der Konstanten 7, 12 und 16 drei Variablen einzuführen, z. B. ZAHL1, ZAHL2 und ZAHL3. Das dadurch beschriebene Verfahren gilt dann für die Addition dreier beliebiger Zahlen.

  Das Problem, die Summe aus drei Zahlen zu berechnen, gehört aber zu einer allgemeineren Problemklasse, nämlich die Summe aus N Zahlen zu berechnen. Dabei ist N eine natürliche Zahl größer als 1. Unser spezielles Problem, die Zahlen 7, 12 und 16 zu addieren, ist also nur ein Spezialfall dieses allgemeinen Problems für N = 3. Die Lösungsvorschrift für dieses Problem ist dann ein Algorithmus im Sinn unserer Definition. Durch die Forderung nach Allgemeinheit erreicht man also, dass man wenige mächtige Algorithmen erhält und sich nicht in einer Vielzahl von Lösungsvorschriften für Sonderfälle verzettelt.

Weitere Punkte, die auch oft als Eigenschaften von Algorithmen gefordert werden, sich aber aus den obigen Eigenschaften ergeben, sind:

1. **Vollständigkeit der Beschreibung**
   Es muß eine komplette Anweisungsfolge vorliegen. Eine Bezugnahme auf unbekannte Information darf nicht enthalten sein.

2. **Wiederholbarkeit des Algorithmus**
   Im Sinne eines physikalischen Experiments muß man die Ausführungen eines Algorithmus beliebig oft wiederholen können und jede Wiederholung muß bei gleichen Eingabedaten das gleiche Resultat liefern (Reproduzierbarkeit). Ist ein Algorithmus endlich und definit, so ergibt sich diese Forderung automatisch.

3. **Korrektheit des Algorithmus**
   Diese Forderung ist zwar selbstverständlich, aber in der Praxis ist die Korrektheit nur sehr schwer nachzuweisen. Man bedient sich daher Tests, bei denen für die vorgegebenen Eingabedaten das Ergebnis bereits bekannt ist, und vergleicht dann die erzeugten Ausgabedaten mit den Ergebnissen. (Ein solcher Test ist insofern problematisch, da alle möglichen Fälle abgedeckt werden müssen. Im Extremfall muß dieser Test sogar für jede mögliche Eingabe durchgeführt werden.) Der Begriff der Korrektheit nimmt zwar in der Literatur einen sehr großen Raum ein, da es nicht trivial ist, die Korrektheit nachzuweisen. Er ist jedoch für die Definition eines Algorithmusbegriffs nicht erforderlich.

Neben diesen Eigenschaften gibt es noch weitere Eigenschaften von Algorithmen, die sich auf die Art und Weise der Ausführung des Algorithmus beziehen. Hier sind zu nennen:

- Die *Effizienz* der Beschreibung und der Ausführung (umständlich oder einfach).
- Die Art der Ausführung einzelner Anweisungen (sequenziell oder parallel).
- Die *Komplexität* bei der Ausführung. (Sie ist ein Maß für den Aufwand bei der Durchführung eines Algorithmus).

Zusammenfassen lassen sich alle diese Eigenschaften in einer kurzen, aber präzisen Definition des Begriffs **Algorithmus**.

Eine Bearbeitungsvorschrift heißt **Algorithmus**, *wenn sie folgende Eigenschaften erfüllt:*

1. *Die Vorschrift ist mit endlichen Mitteln beschreibbar.*
2. *Sie liefert auf eine eindeutig festgelegte Weise zu einer vorgegebenen Eingabe in endlich vielen Schritten genau eine Ausgabe.*

Da man nicht alle Bearbeitungsvorschriften durch eindeutige Folgen von Anweisungen als Algorithmus darstellen kann, wurde im Laufe der Zeit der Begriff ausgedehnt. Man unterscheidet heute zwischen **deterministischen** und **nicht-deterministischen** Algorithmen. Enthält ein Algorithmus elementare Anweisungen, deren Ergebnis durch einen Zufallsmechanismus beeinflusst wird, so heißt dieser Algorithmus **nicht-deterministisch**. Liefert er bei der gleichen Eingabe immer die gleiche Ausgabe, so heißt er **deterministisch**.

Spielen wir nun noch ein Beispiel für eine Verfahrensbeschreibung durch. Objekte des Verfahrens sind kleine Striche (einige Millimeter lang). Beschreibung des Verfahrens „S“:

(S1) Wenn keine Striche mehr in der Eingabe vorhanden sind, so höre auf.

(S2) Nimm einen Strich aus der Eingabe weg. Wenn bereits vier Striche ungebündelt in der Ausgabe liegen, lege einen Querstrich über sie; andernfalls füge einen Strich in der Ausgabe hinzu. Dann gehe nach (S1).

Bei dieser Art der Darstellung eines Verfahrens gilt die Regel: Wenn man alle Handlungen eines Schritts durchgeführt hat und nicht ausdrücklich durch „höre auf“ oder „gehe nach“ etwas anderes gesagt bekommt, so geht man zum nächsten Schritt.

Die Eingabe des Verfahrens „S“ besteht aus einer Anzahl von Strichen, die irgendwie angeordnet sind, z. B. wie in Bild 2.2.

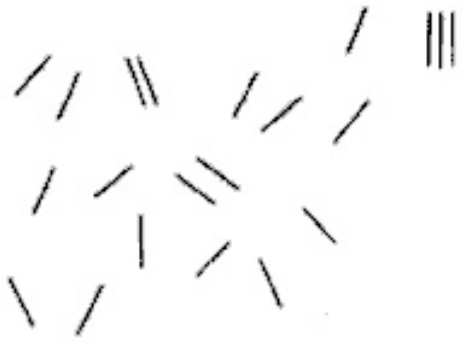

Bild 2.2: Eingabe des Verfahrens „S“

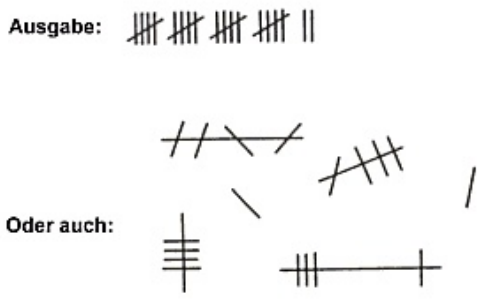

Bild 2.3: Mögliche Ausgaben des Verfahrens „*S*“

Führen Sie jetzt das Verfahren „*S*“ durch. In (S2) ist eine Bedingung; diese ist zu prüfen und, wenn sie erfüllt ist, die dahinterstehende Anweisung ausführen. Als Ausgabe erhält man, was in Bild 2.3 dargestellt ist.

Gewünscht war natürlich das obere Beispiel. Also hat unsere Verfahrensbeschreibung noch etliche Schwachstellen. Sie müßte präzisiert werden, etwa durch: „Füge einen Strich von 8 mm Höhe parallel zu den anderen Strichen und parallel zur langen Kante des Papiers im Abstand von 0,5 mm von dem vorhergehenden auf derselben Grundlinie wie die anderen an“. Man könnte der Beschreibung auch noch einen vernünftigen Namen geben: „Verfahren zur Bündelung von Strichen in Fünfergruppen“.

Damit ein Verfahren einem Computer mittels einer Programmiersprache erklärt werden kann, muß es den in Bild 2.4 gezeigten Bedingungen genügen. Der Begriff Algorithmus ist ein zentraler Begriff der Programmierung. Er baut immer auf einer Menge von Grundoperationen auf.

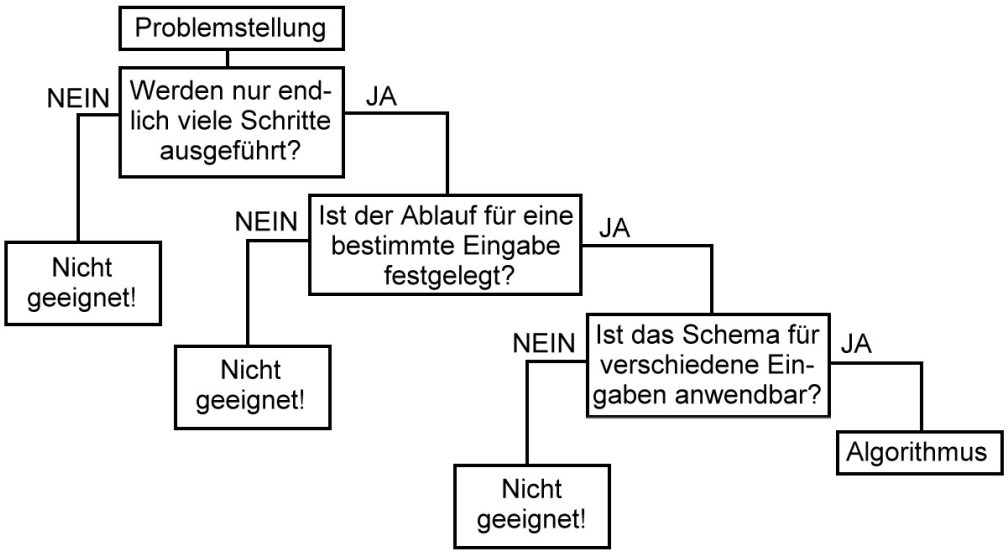

Bild 2.4: Bedingungen für einen Algorithmus

2.2 Was ist Programmieren?

Das Verfahren „S“ ist in deutscher Sprache geschrieben, die für ein und dieselbe Handlung oft mehrere Ausdrücke erlaubt. So läßt sich „wegnehmen“ auch durch „entfernen“ oder „streichen“ ersetzen. Auch die Sätze lassen sich auf verschiedene Art konstruieren: „Wenn keine Striche mehr in der Eingabe sind, höre auf“, „sind keine Striche mehr in der Eingabe, dann Ende“, „Schluß, falls keine Striche mehr in der Eingabe sind“, „Solange noch Striche vorhanden sind gehe zu S2, sonst höre auf“.

2.2.1 Programme

Da dem Computer jeder zu vollziehende Schritt genau zu erklären ist, müßte ihm auch erklärt werden, dass die obigen Konstruktionen und Worte die gleiche Bedeutung besitzen. Man muss dann aber auch klären, wann die Begriffe nicht das Gleiche bedeuten. Die Schwierigkeiten, die dabei entstehen, sind der Grund, weshalb man dem Computer nur eine ganz bestimmte Sorte von Sprachen, sogenannte normierte Sprachen, anbieten darf. Die Programmiersprachen sind normierte Sprachen, die der Beschreibung von Verarbeitungsvorschriften, Datenstrukturen sowie Ein- und Ausgabe dienen.

Programmieren bedeutet also, einen Computer dazu zu bringen, Algorithmen auszuführen. Dafür muss man den Algorithmus so formulieren, dass der Computer ihn versteht – in einer Programmiersprache. Mit dem, was Sie bisher wissen, können Sie definieren, was man unter einem Programm versteht:

Ein Programm ist die Formulierung eines Algorithmus in einer Programmiersprache.

Programme sind also Algorithmen, die in einer besonderen Sprache formuliert sind. Den Begriff „Programmieren“ kann man somit wie folgt definieren:

Unter Programmieren versteht man das Aufschreiben eines Algorithmus in einer Programmiersprache.

Wenn Sie eine höhere Programmiersprache beherrschen wollen, müssen Sie zuerst einmal ihr Vokabular und ihre Grammatik lernen. Zum Verständnis der Programmiersprache müssen Sie zusätzlich auch deren Konzepte erlernen, um die Sie sich bei natürlichen Sprachen nicht kümmern müssen. Bei imperativen Programmiersprachen wie Perl sind dies z. B. Variablen, Anweisungen, Prozeduren. Alleine durch das Auswendiglernen von Vokabeln und Grammatikregeln schaffen Sie es jedoch nicht, mit der Programmiersprache umgehen zu können. Sie müssen die Sprache auch konsequent einsetzen und durch ständiges Üben Erfahrungen sammeln.

Das Erlernen einer Programmiersprache ist nicht schwierig, da das Vokabular und die Grammatik nicht besonders umfangreich sind. Was sehr viel schwieriger ist, ist das Programmieren lernen, d. h. das Erlernen des Programmentwicklungsprozesses:

- Wie komme ich von einem gegebenen Problem hin zu einem Programm, das das Problem korrekt und vollständig löst?
- Wie finde ich eine Lösungsidee bzw. einen Algorithmus, der das Problem löst?
- Wie setze ich den Algorithmus in ein Programm um?

Während das Erlernen einer Programmiersprache ein eher mechanischer Prozess ist, bei dem die Verwendung des Compilers und das Hilfesystem helfen können, ist die Programmentwicklung ein kreativer Prozess, der Intelligenz voraussetzt. An dieser Stelle sind Sie gefragt! Programmieren lernen bedeutet in noch stärkerem Maße als das Erlernen einer Programmiersprache: Üben und Erfahrungen sammeln.

- Schauen Sie sich die Programme anderer Programmierer an und überlegen Sie: Wieso hat der das Problem so gelöst?
- Denken Sie sich selbst Probleme aus und versuchen Sie, hierfür Programme zu entwickeln.
- Fangen Sie mit einfachen Aufgaben an und steigern Sie nach und nach den Schwierigkeitsgrad.
- Ganz wichtig ist: Versuchen Sie, Programmierpartner zu gewinnen, mit denen Sie Probleme und Lösungsansätze diskutieren können.

Der Ausdruck „Programmieren“ bezeichnet also im engen Sinn lediglich das Erstellen eines Programms. Oft bezeichnet man jedoch mit „Programmieren“ alle Tätigkeiten eines Programmierers, die von der Problemstellung zur fertigen Lösung führen. Das Problemlösen mithilfe eines Computers umfaßt folgende Stufen:

1. Analyse der Problemstellung
2. Entwurf des Algorithmus
3. Erstellen des Programms
4. Prüfen auf Korrektheit
5. Dokumentation des Programms
6. Anwendung des Programms

Ein Programm ist also eine eindeutige, logische Folge von (genormten) bekannten Bearbeitungsschritten endlicher Länge. Beim Computer richten sich diese Befehle an das Steuerwerk. Die Zahl der verschiedenen Befehle ist bei den Computern aus verständlichen Gründen beschränkt.

Als Beispiel soll ein Algorithmus aus der Mathematik betrachtet werden: die näherungsweise Berechnung des Kreisumfangs.

Algorithmus „Kreisumfang“:
(K1) Nimm den Radius aus der Eingabe.
(K2) Multipliziere den Radius mit 2,0 und nenne das Ergebnis „Durchmesser“.
(K3) Multipliziere den Durchmesser mit 3,1415926 und bringe das Ergebnis in die Ausgabe.

Dieser Algorithmus liefert beispielsweise zur Eingabe 7,0 die Ausgabe 43,982296. Wenn der Computer das Multiplizieren nicht beherrscht, dann müßte man ihm noch einen Algorithmus, also ein Verfahren für dessen Durchführung geben.

Algorithmus „Multiplikation“:
(M1) Schreibe die beiden Zahlen aus der Eingabe nebeneinander.
(M2) Nimm die erste Ziffer der zweiten Zahl.
(M3) Schreibe die erste Zahl so oft untereinander, wie die Ziffer der zweiten Zahl angibt, und zwar so, dass die letzte Ziffer unter der betrachteten Ziffer der zweiten Zahl steht. Für den Fall, dass besagte Ziffer 0 ist, schreibe 0.
(M4) Falls noch Ziffern in der zweiten Zahl vorhanden sind, nimm die nächste Ziffer und gehe nach (M3).
(M5) Addiere alle mit (M3) erzeugten Zahlen.
(M6) Zähle bei dem Ergebnis so viele Stellen von rechts ab, wie beide Eingabewerte zusammen an Stellen hinter dem Komma besitzen. Setze hier das Komma im Ergebnis.
(M7) Bringe dies Endergebnis in die Ausgabe.

Beispiel für die Eingaben 3,14 und 14,0:

```
 3,14 x 14,0
-----------
    314
     314
     314
     314
     314
        0
----------
   43,960
```

Dieser Algorithmus für die Multiplikation kann dann überall dort verwendet werden, wo „multipliziere“ steht. Aus dieser Tatsache lassen sich zwei Erkenntnisse gewinnen:

- Wenn eine komplizierte Tätigkeit öfters benötigt wird, kann man für diese Tätigkeit einen eigenen Algorithmus definieren. Dieser wird dann Unteralgorithmus genannt.
- Man kann eine Programmiersprache auf eine bestimmte, begrenzte Menge von Operationen und Denkstrukturen beschränken.

Versuchen Sie doch einmal, den Algorithmus „Kreisumfang“ so umzuformen, dass er überall dort, wo multipliziert wird, der Algorithmus „Multiplikation“ als „Unteralgorithmus“ aufgerufen wird. Auf dieser Idee beruht die Theorie der Softwareentwicklung. Komplizierte und komplexe Strukturen und Tätigkeiten werden durch bekannte Strukturen und einfachere Unteralgorithmen realisiert. Sie werden auch sehen, dass auch Programme auf entsprechend konstruierte „Unterprogramme“ zurückgreifen können. Und es sei hier schon angemerkt, dass diese Möglichkeit die wichtigste und mächtigste Eigenschaft der Programmiersprachen ist. Die oben erwähnten komplizierten Strukturen sind genau jene, die der menschlichen Denkweise entsprechen. Zum Beispiel die Struktur:

„Wenn die Bedingung erfüllt ist, dann tue dies; sonst tue jenes.“

Dass dabei eine ganze Reihe von Handlungen nacheinander erforderlich sind, wird dem Menschen gar nicht bewußt. Zuerst muß die Bedingung ausgewertet werden. Dazu müssen alle Informationen beschafft werden, die zu dieser Auswertung nötig sind. Danach müssen sie verarbeitet werden, bis festgestellt ist, ob die Bedingung erfüllt ist. Erst jetzt können die entsprechenden Tätigkeiten ausgeführt werden, was unter Umständen wieder recht kompliziert werden kann.

2.2.2 Programmiersprachen

Die höheren Programmiersprachen bieten also schon als Grundoperationen recht komplizierte Dinge an. Ebenso werden komplizierte Denkstrukturen angeboten, die, wie gesagt, dem menschlichen Denken sehr verwandt sind.

Sie sind normierte Sprachen, die der Beschreibung von Algorithmen dienen. Genaueres über solche „algorithmische Sprachen“ sollen Sie im folgenden Abschnitt erfahren. Durch die Anpassung an die menschliche Denkweise wird bei den Programmiersprachen das Schreiben von Algorithmen erleichtert; andererseits läßt sich aus einem Programm eine Beschreibung in natürlicher Sprache zurückgewinnen.

Die fest vorgeschriebenen Teile der Programmiersprachen orientieren sich meist am Englischen. So wird aus der Struktur **„Wenn ... dann ... sonst ...“** in der Programmiersprache Perl **„if (...)** { ... } **else** { ... }**“**.

Allerdings wurden bei der Konstruktion der höheren Programmiersprachen die Ausdrucksmöglichkeiten so ausgewählt, dass Doppeldeutigkeiten unmöglich sind. Dazu wird die Form der Ausdrucksmöglichkeiten vorgeschrieben und ihre Bedeutung eindeutig und genau erklärt.

Der entscheidende Punkt bei den höheren Programmiersprachen ist aber folgender. Dadurch, dass die Form der Programme, die in höheren Programmiersprachen geschrieben sind, bestimmten, festen Regeln genügt, können diese Regeln automatisch, also auch von einem Computer (mit einem entsprechenden Programm) analysiert werden. Da zu jeder erkannten Form die zugehörige Bedeutung genau festgelegt ist, kann das Programm automatisch (d. h. wieder vom Computer) auf eine andere Form mit der gleichen Bedeutung, jedoch mit einfacheren Grundoperationen und Grundstrukturen umgewandelt werden.

Die Umwandlung kann so erfolgen, dass sich die Form völlig, die Bedeutung aber überhaupt nicht ändert. Wenn nun diese neue Form in einer vom Computer ausführbaren Sprache (Maschinensprache) besteht, so wurde eine höhere Programmiersprache (die der Mensch versteht) in die Maschinensprache (die der Rechner versteht) übersetzt.

2.2.3 Maschinensprache

Wenn man die Befehle durchnumeriert, erhält man eine Maschinensprache; das ist der elementarste Programmiercode. In diesem Code kann die Nummer eines Befehls als sein Name verwendet werden. Ein Maschinensprachprogramm ist nichts weiter als eine lange Folge von solchen Codewörtern.

In einer Maschinensprache zu programmieren, ist nicht schwer, aber unglaublich mühsam. Zum Glück hatte einer der frühen Programmierer eine brillante Idee, wie man sich die Arbeit erleichtern kann. Wenn man ein Maschinenprogramm schreibt, das kurze Buchstabenfolgen erkennen und in zugehörige Maschinenbefehle übersetzen kann, dann braucht der Programmierer nicht die Codierung der Befehle in der Maschinensprache zu lernen. Stattdessen kann er jeden Maschinenbefehl als Klartext-Abkürzung (engl. mnemonic) hinschreiben. Die entsprechenden Übersetzungsprogramme, die man „Assembler“ nennt, wurden bald für alle Computer entwickelt.

Das Programmieren in einer Assemblersprache ist etwas weniger mühsam. Ein Programm ist eine Folge von Kommandos, die jeweils aus zwei bis vier Buchstaben bestehen und denen Adressen von Speicherplätzen angefügt werden. Zum Beispiel bedeutet das Kommando **ADDA $2000** „Addiere den Inhalt des Speicherplatzes mit der Adresse $2000 zum Inhalt des Akkumulator-Registers“. Das folgende Beispiel zeigt den Assembler-Code der CPU 68HC11. Es handelt sich um ein Unterprogramm, in dem das Akkumulatorregister um den Inhalt der Speicherzelle $2000 (mit dem symbolischen Namen „tflag“) erhöht und das Ergebnis auf einem Ausgangsport ausgegeben wird. In der linken Spalte ist der erzeugte Binärcode zu sehen.

```
bb2000    count   adda   tflag
b71004            staa   portb
39                rts
```

Noch bis zum Ende der fünfziger Jahre bestand das Programmieren tatsächlich aus der minutiösen Übersetzung von Befehlen in binär, oktal oder sedezimal dargestellten Zahlen und deren Aneinanderreihung zu einem sinnvollen Programm. Man bezeichnet diese Tätigkeit als **Codieren**. Die Unzulänglichkeiten dieses Verfahrens traten aber mehr und mehr hervor, als die Computer schneller und größer wurden:

- Der Codierer war gezwungen, sein Programm auf die Eigenheiten des spezifischen Computermodells auszurichten. Er benötigte dazu genaueste Kenntnisse aller Details dieser Maschine und deren Befehlssatz. Der Austausch von Programmen zwischen verschiedenen Maschinen wurde dadurch unmöglich, und Kenntnisse der Codiermethoden eines Computers waren oft wertlos für die Codierung eines anderen. Jeder erstellte eigene Programme und war gezwungen, bei der Neuanschaffung eines Computers diese aufzugeben und mit der Codierarbeit von vorne zu beginnen. Es wurde klar, dass

das gezielte Ausrichten von Algorithmen auf die merkwürdigsten Eigenheiten eines bestimmten Computers eine schlechte Verwendung des menschlichen Intellektes war.

- Der Programmierer wurde durch seine enge Bindung an eine Rechenanlage und die extrem begrenzten Hardwaremöglichkeiten – langsam, wenig Arbeitsspeicher, extrem teuer (mehrere MegaDMs/Rechner) – nicht nur befähigt, sondern sogar ermuntert, alle möglichen Tricks zu erfinden, um aus den Eigenheiten des Computers ein Maximum herauszuwirtschaften. Zu dieser Zeit galten die Programmierer noch als Beigabe des Hardwareherstellers, denn im Vergleich zu der Hardware, kosteten die Arbeitskräfte kaum was. Als die verzwickte Programmierung in Mode war, verwendeten Programmierer nicht nur viel Zeit zur Erstellung **optimaler** Programme, auch deren Tests stellte sich als äußerst schwierig dar. Es war für einen Mitarbeiter beinahe unmöglich, die Funktionsweise eines fremden Programms herauszubekommen (und oft war es sogar schwierig, das eigene zu überblicken). Heute ist das Teuerste die Erstellung von Anwendersoftware. Deshalb vermeiden heutige Programmierer die Anwendung von Tricks um jeden Preis.
- Der sogenannte Maschinencode enthielt sehr wenig Redundanz, mit der ein Fehler hätte entdeckt werden können. Bereits kleine Schreibfehler verändern einen Maschinencode in einen anderen gültigen Maschinencode, der aber völlig andere Bedeutung und somit Auswirkungen auf die Funktion des Programms hat. Solche Fehler sind nur sehr schwer zu entdecken, obwohl sie bei der Ausführung des Programms verheerende Folgen haben konnten.
- Die Darstellung eines Vorgangs als unstrukturierte, lineare, monotone Sequenz von Befehlen ist eine für den Menschen ungeeignete Form, komplizierte Prozesse zu beschreiben und zu überblicken. Die Präsenz von hierarchischen Beschreibungsstrukturen, die eine Sicht auf die Funktionen in unterschiedlicher Detaillierungsstufe ermöglichen, ist das hauptsächlichste Hilfsmittel, um den Überblick zu wahren und Programme systematisch zu erstellen.

Diese Unzulänglichkeiten führten unter anderem zur Entwicklung sogenannter höherer Programmiersprachen, die zum einen nach den Gewohnheiten und Fähigkeiten des Menschen im Ausdruck seiner Gedanken ausgerichtet sind und sich zum anderen in ihrer Art und im Umfang am Anwendungsgebiet orientieren (und nicht an der zugrunde liegenden Hardware).

2.2.4 Interpreter, Compiler

Mit den Kommandowörtern der Assemblersprache war ein erster Schritt getan, die Programme für den Menschen verständlich zu schreiben. Man wählte Bezeichnungen, die wie ADD auf die Operation hinwiesen. Doch warum sollte man sich auf Wörter mit drei Buchstaben beschränken? Es wäre noch leichter, Programme zu schreiben, wenn man näher an der Umgangssprache formulieren könnte. Nun wiederholte sich der gleiche Schritt, der von Maschinensprachen zu Assemblersprachen geführt hatte. Man entwickelte (kompliziertere) Übersetzungsprogramme, sie heißen Interpreter und Compiler. Diese neuen Programme übersetzten zunehmend komplexe Buchstabenfolgen in eine Form, die der Computer verstehen kann (Bild 2.5). Damit konnten die Programme in einer Sprache geschrieben werden, die aus Wörtern der Umgangssprache bestand.

Meist tun diese Programme noch mehr, als nur zu übersetzen: Sie melden dem Programmierer auch Stellen in seinem Programm, die nicht den Konventionen der Sprache genügen – Programmierfehler also. Ein Maschinenprogramm läßt sich nur noch unter den größten Schwierigkeiten verstehen. Ein Beispiel soll das verdeutlichen:

1. In natürlicher Sprache: Schreibe das Wort „PASCAL“.

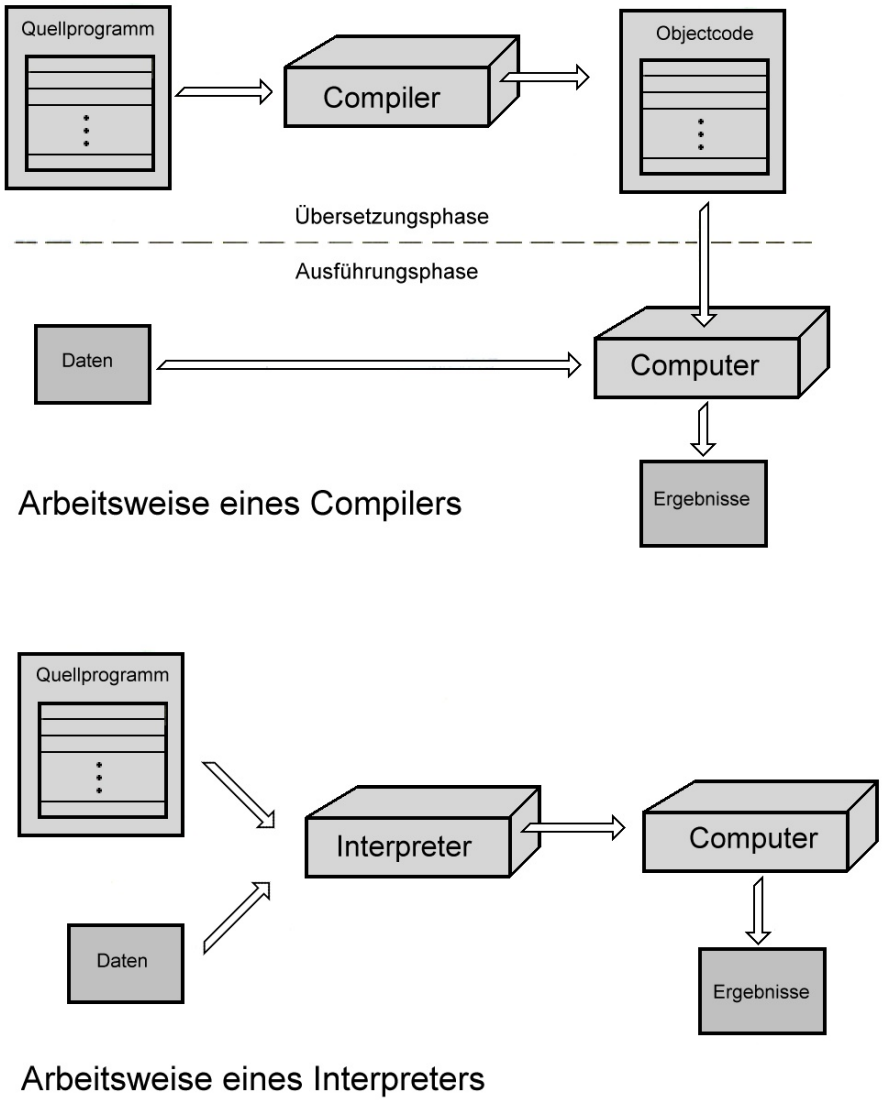

Bild 2.5: Arbeitsweise von Compiler und Interpreter

2. In der höheren Sprache Perl: print „PASCAL“;

3. In der Maschinensprache des – inzwischen historischen – programmierbaren Taschenrechners TI-59:
   69 00 03 03 01 03 03 06 01 05 01 03 69 01
   02 07 00 00 00 00 00 00 00 00 69 02 69 05

Wie Sie sehen, bieten die höheren Programmiersprachen ganz entscheidende Vorteile. Es gibt also:

- **Assemblersprachen (kurz: Assembler, maschinenorientiert)**
  Hier wird jeder Maschinenbefehl durch eine mnemotechnische Abkürzung bezeichnet. Speicheradressen können mit symbolischen Namen versehen werden. Die Zuordnung von Assemblerbefehl zu Maschinenbefehl ist 1:1, d. h. für jeden Maschinenbefehl wird

ein Assemblerbefehl benötigt. Die Assemblersprache ist daher extrem stark an den jeweiligen Prozessor gebunden. Ein Transport des Programms auf einen anderen,nicht kompatiblen Prozessor ist nicht möglich.

- **Höhere Programmiersprachen (problemorientiert)**
  Hier wird die Programmierung in einer eigens entwickelten problemorientierten Sprache vorgenommen (algorithmische Sprache). Die Zuordnung der Befehle ist nicht mehr 1:1, ein Befehl in einer höheren Sprache hat in der Regel eine ganze Folge von Maschinenbefehlen als Ergebnis. Der Vorteil einer problemorientierten Sprache liegt auch darin, dass sie maschinenunabhängig ist. Ein Transfer der Programme auf andere Prozessoren ist mit wenig Aufwand möglich.

Die Programmiersprachen sind in erster Linie dazu da, die Lösung einer Aufgabe in computergerechter Form zu formulieren. Die Lösung einer hier betrachteten Aufgabe soll also durch einen Algorithmus dargestellt in einer Programmiersprache beschrieben sein. Wie muß eine solche Programmiersprache (algorithmische Sprache) nun aufgebaut sein, damit sie geeignet ist, beliebige Algorithmen darzustellen?

2.3 Algorithmische Sprachen

In diesem Abschnitt wollen wir gemeinsam untersuchen, welche Eigenschaften die Sprachen besitzen müssen, die zur Beschreibung von Algorithmen verwendet werden. Im letzten Abschnitt haben wir die wesentlichen Merkmale der algorithmischen Sprachen bereits kennengelernt. Bei ihnen handelt es sich um normierte Sprachen, die der Beschreibung von Algorithmen dienen. In diesem Abschnitt wird versucht, zwei Kernfragen zu beantworten:

1. Wie kann man eine Sprache normieren?
2. Was ist zur Beschreibung von Algorithmen erforderlich?

In jeder Sprache, sei es die deutsche, englische, italienische, griechische oder irgend eine andere Sprache, wird ein Text als Folge von Zeichen niedergeschrieben. Die Menge dieser Zeichen ist bei den verschiedenen Sprachen nicht immer die gleiche, es gibt auch nicht immer gleich viel Zeichen. Aber für jede Sprache existiert eine endliche Menge von Zeichen, die zur Niederschrift eines Textes verwendet werden dürfen.

Wenn man diese Menge von Zeichen in einer festgelegten Reihenfolge ordnen kann, spricht man von einem Alphabet. Es ist aber noch nicht möglich, einen Text zu schreiben, wenn man nur das Alphabet kennt. Wenn Sie zum Beispiel unser Alphabet betrachten, dann ergibt „qwesdfcvxwcbvhgnrzrrdegfbcvdgftejsmbljouri“ noch keinen Text.

Man benötigt also Elemente der Sprache, gebildet aus den Zeichen des Alphabets. Gebilde, die durch Aneinanderreihen von Zeichen eines Alphabets nach irgendwelchen Regeln entstehen, heißen Worte. Worte sind also Zeichenfolgen. Ein Wort unterliegt gewissen Bildungsregeln. „3az45“ ist kein Wort der deutschen Sprache, da es gegen die Regeln der deutschen Wortbildung verstößt. Das einzelne Wort lebt wiederum von der Kombination mit anderen Worten. Eine Kombination von Worten heißt Satz.

Zeichen + Regeln → Wort
Wort + Regeln → Satz

Aus einer festgelegten Menge von Zeichen werden nach bestimmten Regeln Worte gebildet, die selbst wieder nach festen Regeln zu Sätzen vereinigt werden. Ein Text besteht dann aus einer Folge von Sätzen. Dieser formale Aufbau einer Sprache heißt Grammatik oder **Syntax** der Sprache. Die Syntax beschreibt im Wesentlichen die Regeln, die Sie beachten müssen, damit die Form des Textes richtig ist.

Die Syntax ist auch bei Worten wie BARTSCHE, DRAUDELN, GRUBBEND oder WRDLBRMPFT und Sätzen wie „DER COMPUTER BETRITT DEN RASEN“ völlig in Ordnung. Aber Worte und Sätze, die von der Syntax her richtig (also syntaktisch korrekt) sind, ergeben keinen Sinn, keinen Inhalt, sie bedeuten nichts. Der Sinngehalt eines Satzes oder Wortes muß also ebenfalls festgelegt werden. Das bedeutet, wenn Sie ein Wort schreiben, müssen Sie die Bedeutung des Wortes kennen. Wer MAUS schreibt und dabei an einen großen, grauen Dickhäuter denkt, verstößt gegen jene Regeln der Sprache, die für den Sinngehalt wesentlich sind, die Regeln der **Semantik**. Mit der Erkenntnis, dass die Syntax einer Sprache ohne eine ausreichende Definition der Semantik nicht für die Normierung der Sprache ausreicht, ist die erste der beiden anfangs gestellten Fragen beantwortet. Es stellt sich jetzt die Frage, was in einer normierten Sprache alles vorhanden sein muß, um Algorithmen zu beschreiben.

Also benötigt eine algorithmische Sprache Möglichkeiten zur Beschreibung der Eingabe- und Ausgabedaten und zur Beschreibung der Verarbeitungstätigkeit. Die Beschreibung der Daten muß die Art der Daten, (z. B. Zahlen, Striche, Texte, usw.) sowie die Anzahl der Daten enthalten. Die Beschreibung der Tätigkeiten soll der menschlichen Denkweise möglichst gut angepaßt sein, muß aber trotzdem normiert sein. Dazu ein Beispiel:

1. Wenn es regnet, nehme ich den Schirm mit, sonst nicht.
2. Regnet es, so nehme ich den Schirm mit, sonst halt nicht.
3. Ich nehme den Schirm nur dann mit, wenn es regnet.
4. Nur falls es regnet, nehme ich den Schirm mit.
5. Ich nehme, sollte es regnen, den Schirm mit, sonst nicht.
6. Dann und nur dann, wenn es regnet, nehme ich den Schirm mit.

Eine Normierung tut not, denn alle diese Sätze haben dieselbe Bedeutung. Also entwerfen wir eine Syntaxregel:

```
WENN  <Bedingung>
  DANN <Erfuellt-Aktion>
  SONST <Nicht-erfuellt-Aktion>
ENDEWENN
```

Dann lautet das Beispiel:

```
WENN es regnet DANN  ich nehme Schirm mit
               SONST ich lasse Schirm zu Hause
ENDEWENN
```

Es gibt nur eine Möglichkeit, das Ganze noch anders darzustellen, und zwar:

```
WENN es regnet nicht DANN  ich lasse Schirm zu Hause
                    SONST ich nehme Schirm mit
ENDEWENN
```

Aufgrund der Normierung der algorithmischen Sprache bleibt von den vielen Ausdrucksvariationen der natürlichen Sprache nur noch eine einzige Version übrig. Abkürzungen und Umstellungen sind nicht mehr zulässig. Die Sätze müssen immer in der vorgeschriebenen Reihenfolge aufgeschrieben und die Wörter immer ausgeschrieben werden. Wie in der natürlichen Sprache gibt es auch in einer algorithmischen Sprache zahlreiche festgelegte Worte, die sich nicht auf Gegenstände oder Tätigkeiten beziehen, sondern der Verbindung

der Worte zu Sätzen dienen. Im Deutschen sind dies zum Beispiel die Präpositionen oder die Konjunktionen; im letzten Beispiel waren es die Worte WENN, DANN und SONST. Diese Worte dienten der Verbindung der Sätze: „es regnet“, „ich nehme Schirm mit“, „ich lasse Schirm zu Hause“. Die drei **Schlüsselworte** verbinden die genannten Sätze in festgelegter Weise; man nennt sie daher auch **Wortsymbole** oder **reservierte Worte**.

Die im Beispiel oben gewählte Syntax wurde so gewählt, dass aufgrund der Kenntnis der deutschen Sprache der Sinn des „Gebildes“ ungefähr klar wird. Bedingung steht für einen Satz, der nach anderen Regeln gebildet werden muß, wie die Sätze hinter den Worten DANN und SONST und der, wie es von einer Bedingung erwartet wird, entweder erfüllt oder nicht erfüllt sein muß. Ist die Bedingung erfüllt, wird die Tätigkeit: <Erfüllt-Aktion> ausgeführt, im anderen Fall die mit <Nicht-Erfüllt-Aktion> bezeichnete Aktion. Somit hätten wir die Syntax und die Semantik unseres Beispiels definiert.

Natürlich muß auch die Reihenfolge der Bearbeitung festgelegt werden: Prüfe zuerst die Bedingung und führe dann die entsprechende Tätigkeit aus. Die Wortsymbole sind also festgelegte Worte mit fest definierter Bedeutung. Sie werden in den algorithmischen Sprachen meist durch Symbolzeichen wie „+“ für die Addition oder „/“ für die Division ergänzt. Auch die Bedeutung dieser Zeichen ist festgelegt und nicht zu ändern.

Die Menge aller Wort- und Zeichensymbole bildet zusammen mit der Beschreibung von Syntax und Semantik die Definition einer algorithmischen Sprache. Die Symbole dürfen auch in keiner anderen Bedeutung verwendet werden als in der festgelegten. Die Zahl der Symbole sollte so klein wie möglich, aber trotzdem so universell wie möglich sein. Die Vorteile von wenigen, der menschlichen Denkweise angepaßten Symbolen und Zeichen liegt auf der Hand:

- Wenig Symbole erfordern geringen Lernaufwand sowohl für Syntaxregeln als auch für die Semantik.
- Eine geringe Anzahl von Symbolen kann auch durch ein relativ kurzes und effektives Übersetzungsprogramm erkannt und ausgewertet werden.
- Die Universalität der Symbole soll verhindern, dass ein Programmierer „um die Ecke denkt“.

Mit den Symbolen alleine kommt die algorithmische Sprache aber immer noch nicht aus. Sie möchten ja den Sätzen einen Inhalt und den Dingen, die sie behandeln, und den Tätigkeiten einen Namen geben. Erinnern Sie sich noch an das Verfahren „S“? „S“ war dabei ein völlig willkürlich gewählter Name für das Verfahren. Anschließend wurde für das gleiche Verfahren ein anderer, ebenfalls willkürlicher, aber einleuchtenderer Name gewählt: „Verfahren zur Bündelung von Strichen in Fünfergruppen“. So wie dort die Namen der Verfahren frei gewählt wurden, können auch die Namen für Dinge, Daten, Personen, Verfahren und Tätigkeiten in einer algorithmischen Sprache frei gewählt werden. Damit Sie sich bei den frei gewählten Bezeichnungen später auch auskennen, sei Ihnen hier erklärt, was mit einer solchen Bezeichnung eigentlich benannt wird.

Der Gegenstand der Namensgebung wird zuerst durch seinen Namen und seine Art bekanntgemacht (z. B. Zahl, Tätigkeit, Person, Menge, Buchstabe, Operation, Unteralgorithmus usw.). Ferner wird er durch die Tätigkeiten, die auf ihn angewendet werden charakterisiert. Steht zum Beispiel irgendwo „X/12“ und jemand fragt: „Was ist X?“, können Sie nur antworten: „Es handelt sich um irgendetwas, das durch 12 geteilt wird“. So eine Antwort ist aber recht unbefriedigend. Als Lehre sollten Sie daraus ziehen, alle Dinge sinnvoll bezeichnen und zusätzliche Erklärungen und Kommentare für den Leser hinzufügen. Die Anweisung „$Y = X/12$“ im Beispiel ist also sicher schlechter Stil. Hingegen weiss man bei „$EIERKARTONZAHL = EIERZAHL/12$“ sofort, worum es geht.

Das Beispiel enthält die frei gewählten Bezeichnungen: X, Y, EIERKARTONZAHL, EIERZAHL. Es fehlt nur noch die Einordnung der Zahl 12. Es handelt sich hierbei um eine allgemein übliche und allseits bekannte Bezeichnung. Sie wird daher auch in algorithmischen Sprachen als allgemein üblich betrachtet und Standardbezeichnung genannt. Die Ausdrucksmöglichkeiten einer algorithmischen Sprache werden also durch Wortsymbole, Symbolzeichen, Standardbezeichnungen und frei wählbare Bezeichnungen im Rahmen einer (computerübersetzbaren) Syntax festgelegt. Die Semantik der Symbole und Standardbezeichnungen ist vordefiniert und wird vom Compiler schrittweise in eine Maschinensprache übersetzt, während die Semantik der frei wählbaren Bezeichnungen von der durch die Syntax festgelegten Deklarationen (Bekanntmachungen) festgelegt wird.

Ideal ist es, wenn eine solche, syntaktisch festgelegte Form mit ihrer Semantik dem Menschen, der die Verarbeitungsvorschrift verfaßt, leicht verständlich und zugänglich ist. Die erzeugten Verarbeitungsvorschriften müssen lesbar und der menschlichen Denk- und Ausdrucksweise angepaßt, aber trotzdem normiert und computerübersetzbar sein. Seit Mitte der sechziger Jahre werden Programmiersprachen (= algorithmische Sprachen) entwickelt.

Diese höheren Programmiersprachen wie BASIC, FORTRAN, Pascal, C Ruby, Python oder Perl wurden entwickelt, um Probleme leichter lösen zu können. Dagegen waren die Maschinensprachen und auch die Assemblersprachen eher dafür bestimmt, die internen Abläufe im Computer zu steuern. Wenn man eine höhere Programmiersprache verwendet, braucht man sich nicht mehr darum zu kümmern, wie man die Befehle erhält und auf welche Speicherplätze man zugreifen kann. Einige Programmiersprachen haben sich an den Bedürfnissen bestimmter Anwendungsbereiche orientiert. Ebenso wie verschiedene Typen von Taschenrechnern etwa für statistische, für kommerzielle oder für wissenschaftliche Berechnungen entwickelt wurden, gibt es anwendungsspezifische Programmiersprachen. Man kann die meisten Programme in jeder Programmiersprache schreiben.

2.4 Algorithmen und das Lösen von Aufgaben

Bevor man ein Programm schreiben kann, muß man einen Algorithmus für die Lösung der (hier vorgegebenen!) Aufgabe entwerfen. Ein Algorithmus faßt alle Schritte zusammen, die man auf dem Wege zur Lösung gehen muß. Er ist im Allgemeinen so detailliert dargestellt, dass er die Grundlage für ein Programm bilden kann, aber noch nicht in einer Programmiersprache formuliert wird. Ein guter Programmierer wird einen Algorithmus ohne Schwierigkeiten aus der Umgangssprache in jede Programmiersprache übertragen können. Für den Anfänger stellt dagegen die „Unschärfe“ der Umgangssprache häufig ein Problem dar.

2.4.1 Zwei Beispiele

Um das oben gesagte zu verdeutlichen, möchte ich zwei Beispiele anführen:

Beispiel: Vertauschen von Zahlen

Beim Algorithmus für die Division durch einen Bruch muß der Kehrwert gebildet werden, d. h. der Zähler ist mit dem Nenner zu vertauschen. Man ist versucht, das auf folgende Weise zu machen:

- Nimm den Zähler und den Nenner auf.
- Gib dem Zähler den Wert des Nenners.
- Gib dem Nenner den Wert des Zählers.

Bemerken Sie den Fehler, der auftreten würde, wenn der Computer diese Anweisungsfolge pedantisch ausführt (und das tut er tatsächlich!)? Beide Variablen, die für den Zähler und die für den Nenner, hätten zum Schluß den gleichen Wert: den des ursprünglichen Nenners. Das liegt daran, dass man den Wert des ursprünglichen Zählers nicht zwischengespeichert hat. Ein Algorithmus für den Computer muß das ausdrücklich vorsehen:

- Nimm den Zähler und den Nenner auf.
- Speichere den Zähler.
- Gib dem Zähler den Wert des Nenners.
- Gib dem Nenner den gespeicherten Wert.

Beispiel: Fallentscheidung

Gegeben sei eine Variable X. Wenn X den Wert 2 hat, soll X den Wert 1 erhalten und umgekehrt. Ein an sich klarer und einfacher Algorithmus. Doch bei der Umsetzung gibt es ein paar Fallen.

- *Lösung 1:*
  Prüfe, ob X den Wert 2 besitzt.
  Falls ja, setze X auf 1.
  Setze X auf 2.

- *Lösung 2:*
  Prüfe, ob X den Wert 2 besitzt.
  Falls ja, setze X auf 1.
  Sonst setze X auf 2.

- *Lösung 3:*
  Setze X auf das Ergebnis der Rechnung 3 - X.

Welche Lösung ist richtig? Erstaunlicherweise keine! Bei der ersten Lösung hat X nach Ablauf immer den Wert 2. Die beiden anderen Lösungen scheinen beide richtig, wobei die dritte zwar elegant erscheint, aber für den Leser des Algorithmus schwerer verständlich ist. Sie sind aber deshalb falsch, weil nicht überprüft wird, ob X einen der beiden zulässigen Werte (1,2) besitzt. Richtig wäre also:

1. Prüfe, ob X den Wert 1 oder den Wert 2 besitzt.
2. Falls nein, melde den Fehler und beende die Arbeit.
3. Prüfe, ob X den Wert 2 besitzt.
4. Falls ja, setze X auf 1.
5. Sonst setze X auf 2.

Diese Beispiele sollen Sie nicht in Panik versetzen. Sie brauchen nicht an Ihren Fähigkeiten, Algorithmen für den Computer zu entwerfen, zu verzweifeln. Diese Fähigkeiten sind keineswegs unveränderlich, sie lassen sich sehr wohl entwickeln und trainieren.

2.4.2 Denken und Arbeitsweisen schulen

Das Denkvermögen des Menschen läßt sich ebenso trainieren wie seine körperliche Leistungsfähigkeit. In den Kapiteln dieses Skriptums werden Sie eine Vielfalt von Ansätzen zur Lösung von Problemen kennenlernen. Auch Vorgehensweisen wie die schrittweise Verfeinerung eines Problems und das Top-Down-Verfahren beim Erstellen von Programmen werden an mehreren Beispielen ausführlich dargestellt. Es wird untersucht, wie sich die Eleganz eines Algorithmus auf das Programm und seine Effizienz auswirkt.

Wenn man herausgefunden hat, wie ein Problem zu lösen ist, und einen Algorithmus auf dem Papier (oder im Kopf) hat, dann ist es an der Zeit, ein Programm zu entwerfen. Dazu muß gesagt werden: In den letzten Jahren hat es gewaltige Veränderungen beim Erstellen von Software gegeben.

Im Gegensatz zu den Anfangsjahren ist jetzt die Software und nicht mehr die Hardware der Faktor, der die größten Kosten verursacht. Untersuchungen haben gezeigt, dass beträchtlich mehr Zeit in die Überarbeitung und die Anpassung vorhandener Programme investiert wird als in die Entwicklung neuer Software. Das Interesse konzentriert sich daher auf Verfahren, wie man Programme schreiben kann, die nicht nur korrekt arbeiten, sondern auch von anderen verstanden werden können. Der neue Bereich der Softwareerstellung, den man Softwareengineering (Softwaretechnik) nennt, befindet sich in einer stürmischen Entwicklung.

Es darf die systematische Arbeitsweise (Arbeitsdisziplin) nicht vergessen werden: Lernen und Verbessern erfordert einen Rückkopplungsprozeß. Allgemein: Dazu definiert sich der Ingenieur ein quantitativ erfaßbares Ziel und versucht, es zu erreichen. Dabei ist es wichtig, ein objektives Maß (Metrik) dafür zu haben, wie nahe er dem Ziel ist. Aus der Abweichung ergibt sich Steuerungsbedarf im Bestreben, dem Ziel näher zu kommen. Dieser interative Prozeß der Anäherung an das gewünschte Ziel kann nur verbessert (Optimierung) und an Andere weitergegeben (Erfahrungsaustausch) werden, wenn er dokumentiert und objektiv nachvollziehbar (entspricht in der Mathematik: „nachrechenbar“) ist. Für die Softwareentwicklung bedeutet dies: Alle Arbeitsschritte, Metriken und Denkprozesse müssen nachvollziehbar (schriftlich) dokumentiert sein, inklusive der getroffenen technischen Entscheidungen. Nur so kann man selbst und alle zusammen an Können gewinnen.

3

Vom Problem zur Lösung

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs and the universe trying to produce bigger and better idiots. So far, the universe is winning.

Richard Cook

Heute legt man großen Wert auf einen transparenten Programmierstil. In früheren Zeiten wurden Programme nur danach beurteilt, ob sie liefen und ihre Aufgabe lösten. Wie sie entworfen und geschrieben waren, spielte eine untergeordnete Rolle. Ein gutgeschriebenes transparentes Programm „überlistet“ nicht einfach den Computer (und ggf. etwaige Leser des Programms). Es ist so geschrieben, dass irgendjemand auch noch nach Jahren das Programm lesen, verstehen, ausbessern und erweitern kann. Es enthält die einzelnen Schritte, die der Computer erfordert, ohne einen menschlichen Leser zu verwirren oder ganz „abzuhängen“. Außerdem sollte es so wohlbedacht gestaltet sein, dass es in einer realen Umgebung mit (hoffentlich) allen denkbaren technischen Pannen und menschlichem Versagen „vernünftig“ weiterarbeitet. Dazu gehört auch, dass das Fehlverhalten erkannt und so reagiert wird, dass möglichst keine Gefahr für den Menschen und die Umwelt entsteht und etwaiger Sachschaden minimiert wird. Entscheidend ist hier das Qualitätsbewusstsein des Entwicklers und der daraus resultierende konsequente, aber der Sache angemessene Einsatz der bekannten Hilfsmittel der Softwaretechnik.

Auch der Programmierstil läßt sich durch Training verbessern. Früher meinte man, zum Programmieren gehöre ein angebotenes Talent: der eine besitze es, der andere nicht. Doch das Programmieren läßt sich wie das Denken schulen und verbessern. Viele Probleme lassen sich auf unterschiedlichen Wegen lösen. Im Interesse derjenigen, die Ihre Programme lesen (und verstehen!) müssen, sollten Sie ein Gespür dafür entwickeln, welcher Lösungsweg gut ist und welcher nicht. Die Problemlösung läuft also meist nach dem Schema von Bild 3.1 ab.

Das Wichtigste ist beinahe vergessen worden: die Kundenanforderungen! Es ist erstaunlich, wie viel Software am eigentlichen Nutzer vorbei entwickelt wird, nur weil der Entwickler denkt, er wisse, was der Kunde braucht, und sich nicht erkundigt, was der Kunde wirklich will. Oftmals verliert während des Entwurfs der Entwickler den Kunden aus den Augen und baut technikverliebt nicht gewollte Komplexität ins Programm ein, die meistens viel Zeit beim Testen kostet. Eine erste Faustformel könnte lauten:

KISS (*K*eep *i*t *s*imple and *s*mart!) und nicht quick and dirty!

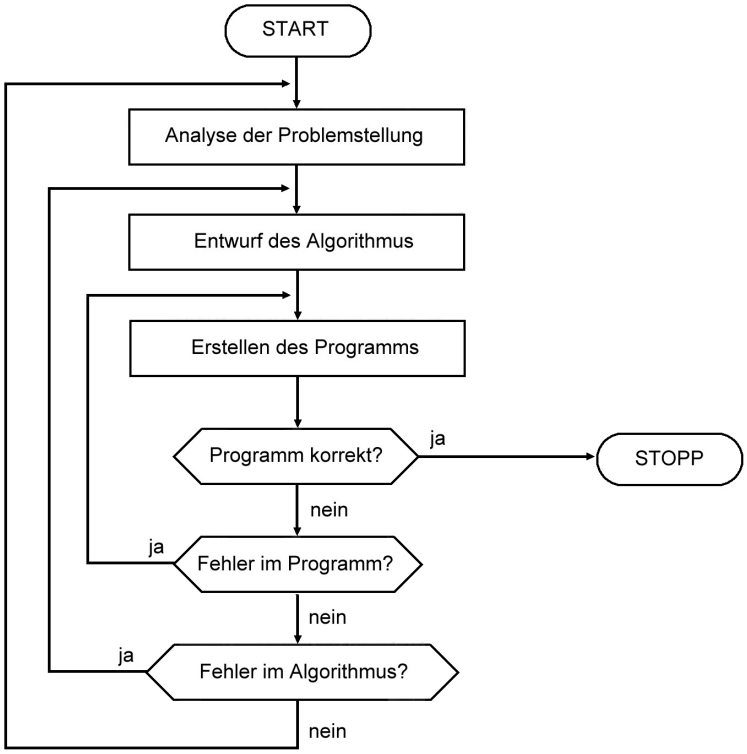

Bild 3.1: Problemlösungszyklus bei Programmen

3.1 Kundenanforderung und Problemanalyse

Bevor der Softwareerstellungsprozess beginnt, ist das Wichtigste, das Ziel festzulegen: Wofür und warum muss Software erstellt werden und was muß sie können, um diese Aufgabe zu erledigen?

Es gibt mindestens zwei Beteiligte, wobei der eine mehrere Rollen verkörpert: den Kunden und den Entwickler. Der erste Schritt ist das schriftliche Festhalten der Kundenanforderungen. Der Entwickler schlüpft hier in die Rolle des Systemanalytikers. Er versucht, den Kunden und dessen Anwendungswelt zu verstehen und aus einer unstrukturierten Aufgabenbeschreibung (*Lastenheft*) die für die Softwareerstellung nötigen Zusammenhänge zu erfassen und daraus eine realisierbare, in sich konsistente - für die Realisierung verbindliche - Aufgabenbeschreibung (*Pflichtenheft)* zu erstellen. Er muss vor allem das Wesentliche vom Unwesentlichen unterscheiden, abstrahieren und auf den Kunden eingehen können, um das zu erfassen, *was* zu tun ist. Im nächsten Schritt ist der Entwickler in der Rolle des Systemdesigners. Ähnlich wie ein Architekten konstruiert er, geleitet von seinen Lösungsideen und der Machbarkeit, die Softwarestruktur (*Softwaredesign*). Er legt fest, *wie* die Aufgabe in Software umgesetzt wird. Parallel dazu wechselt er in die Rolle des Testplaners (Qualitätssicherung), überlegt sich die notwendige Vorgehensweisen und erstellt die *Testspezifikation.* Bislang ist nur Papier entstanden und ungefähr die Hälfte der verfügbaren Zeit vergangen. Im nächsten Schritt übernimmt er die Rolle des Programmierers (Programmcode und Dokumentation) und hoffentlich ein anderer die Rolle des Testers (*Testprotokolle*). Irgendwann ist die Software fertig getestet und wird vom Kunden abgenommen (*Abnahme*). Hier testet der Kunde die Software gegen die im Pflichtenheft versprochenen Funktionen auf Fehlerfreiheit, oftmals im Beisein des Entwicklers. Es wird ein Protokoll erstellt, und bei Fehlerfreiheit der Software muß der Kunde den vereinbarten Preis bezahlen.

Selbstverständlich ist der Entwicklungsablauf nicht so zeitlich und aufgaben-sequentiell wie hier geschildert. Je nach Größe des Projektes kommen während der Entwicklungsphase immer noch Änderungswünsche der unterschiedlichsten Art vom Kunden und auch vom Entwickler. Der Kunde entwickelt sich mit seinen Aufgaben weiter, und der Entwickler versteht seine Aufgabe immer besser, je länger er daran arbeitet. Damit das alles nicht aus dem Ruder läuft, gibt es noch die Rolle des Projekt- oder Entwicklungsleiters. Dessen Verantwortung und Aufgabe ist es, mit den Hilfsmitteln des Projektmanagements die Softwareentwicklung so zu steuern, dass die Software in der geplanten Zeit, im geplanten Umfang und dem festen Budget fertig wird; das ist keine leichte Aufgabe.

3.1.1 Problemanalyse

In seinen „Regeln zur Leitung des Geistes“ stellt der französische Philosoph und Mathematiker Descartes folgende Maximen auf:

1. Übereilung und Vorurteil sind sorgfältig zu meiden.
2. Jede der zu untersuchenden Schwierigkeiten ist in so viele Teile zu zerlegen, wie es möglich und zur besseren Lösbarkeit wünschenswert ist.
3. Mit den einfachsten und faßlichsten Objekten ist zu beginnen und von da aus schrittweise zur Erkenntnis der kompliziertesten fortzuschreiten.
4. Hinreichend vollständige Aufzählungen und allgemeine Übersichten sind anzufertigen, um sicherzugehen, dass nichts ausgelassen wurde.

Dies sind allgemeine Regeln, die jedem Problemlöser in den Sinn kommen (*Heuristik*, griech.: Kunst des Findens bzw. Erfindens). Für den Fall, dass das zu entwerfende Verfahren einer Maschine zur Ausführung überantwortet werden soll, nehmen sie eine spezielle Gestalt an, die Thema der folgenden Kapitel ist.

Die wichtigste Phase beim Erstellen eines Algorithmus ist die Analyse der Problemstellung und das Nachdenken über die Lösung. Dieser Teil, die **Problemanalyse** und die nachfolgende **Skizze des Lösungswegs**, ist wesentlich wichtiger und zeitaufwendiger als die eigentliche Codierung in irgendeiner Programmiersprache. Nicht zu Unrecht arbeiteten in den Anfängen der EDV die Systemanalytiker und Programmierer relativ getrennt voneinander. Damals gab es aber auch keinerlei wissenschaftliche Aufarbeitung des Problems, also das, was heute als „Softwaretechnik“ bezeichnet wird. Programmieren fand eher intuitiv und mit Versuch und Irrtum statt – so wie heute auch oft noch von Anfängern programmiert wird. Welche Phasen beim Entwickeln eines Algorithmus durchlaufen werden, soll an zwei Beispielen gezeigt werden:

Beispiel 1: Berechnung des Jahrestags

Zu einem vorgegebenen Datum (Tag, Monat, Jahr) soll ausgerechnet werden, um den wievielten Tag im Jahr es sich handelt.

Der erste Schritt ist natürlich immer das **Nachdenken** über die Problemlösung. Diese Phase sollte nicht zu kurz sein, denn oft findet sich eine optimale Lösung erst nach dem zweiten oder dritten Ansatz. Und kürzere Programme sind nicht nur schneller, sondern haben (rein statistisch betrachtet) auch weniger Fehler. Dann sollte man den **Lösungsweg skizzieren** und dabei auch überlegen, ob auch alle Randbedingungen und Ausnahmesituationen berücksichtigt wurden.

Oft hilft es, den Algorithmus zuerst anhand eines **Beispiels** durchzuspielen. Dabei werden einem oft Zusammenhänge und Fallstricke klar. Versuchen wir das gleich mit der Problemstellung von oben:

```
Datum = 7.7.99
Jahrestag (JT) = 31 + 28 + 31 + 30 + 31 + 30 + 7 = 188
```

Das Verfahren addiert also bis zum Juni die Monatslängen und zählt dann noch die 7 Tage des Juli hinzu. Spätestens hier sollte einem einfallen, dass der Februar auch 29 Tage haben kann. Der Programmierer macht sich also eine Notiz:

Nachsehen, wann ein Jahr ein Schaltjahr ist.

Danach sollte eine **problemnahe Darstellung des Algorithmus** folgen. Diese sieht beispielsweise so aus:

1. Eingabe von Tag, Monat, Jahr.
2. Ermittle die Monatslänge des Februar (Schaltjahr?).
3. Addiere die Monatslängen der Vormonate (also bis Monat-1) und den Tageswert des eingegebenen Monats.
4. Gib das Ergebnis aus.

Dabei wird davon ausgegangen, dass die Eingabe korrekt ist (also nicht der 35.13.9999 eingegeben wird. Eine Optimierungsmöglichkeit des Algorithmus ist gegeben, wenn man für Werte von Monaten, die kleiner als 2 sind, die Schaltjahresberechnung wegläßt.

Für die Ermittlung des Schaltjahres brauchen wir noch einen Unteralgorithmus. Selbst bei diesem recht einfachen Beispiel zeigt sich also schon ein Grundsatz der strukturierten Programmierung, die Zerlegung einer Aufgabe in kleinere und damit überschaubare Teilaufgaben. Nach dem Gregorianischen Kalender handelt es sich um ein Schaltjahr,

- wenn die Jahreszahl ohne Rest durch 4 teilbar ist.
- nicht aber, wenn die Jahreszahl ohne Rest durch 100 teilbar ist.
- jedoch schon, wenn die Jahreszahl ohne Rest durch 400 teilbar ist.

Das Jahr 2000 ist also ein Schaltjahr (was viele Programme nicht richtig machen). Der Algorithmus lautet also:

```
Eingabe Jahr.
Falls (Jahr {\bf mod} 4) {\bf !=} 0: Ausgabe \qq{NEIN} und ENDE.
Falls (Jahr {\bf mod} 100) {\bf !=} 0: Ausgabe \qq{JA} und ENDE.
Falls (Jahr {\bf mod} 400) {\bf =} 0: Ausgabe \qq{JA}
sonst: Ausgabe \qq{NEIN}.
```

Anmerkung: **mod** ist der Modulo-Operator, also der Rest der ganzzahligen Division. **!=** bedeutet „ungleich“.

Nun steht der Jahreszahlberechnung eigentlich nichts mehr im Weg. Die einzige Schwierigkeit liegt noch darin, dass die Monatslängen unterschiedlich sind:

Jan.	Feb.	März	April	Mai	Juni	Juli	Aug.	Sept.	Okt.	Nov.
31	28/29	31	30	31	30	31	31	30	31	30

Nun könnte man einen Algorithmus niederschreiben, der elf WENN-DANN-Abfragen enthält, welche die Tagessumme der Vormonate berechnen:

```
Eingabe Tag, Monat, Jahr.
Wenn Monat = 1 dann Jahrestag = Tag und ENDE.
Wenn Monat = 2 dann Jahrestag = 31 + Tag und ENDE.
Wenn Monat = 3 dann Jahrestag = 31 + 28 + Tag.
      Wenn Schaltjahr, erhöhe Jahrestag um 1.
      ENDE.
Wenn Monat = 4 dann Jahrestag = 31 + 28 + 31 + Tag.
      Wenn Schaltjahr, erhöhe Jahrestag um 1.
      ENDE.
Wenn Monat = 5 dann Jahrestag = 31 + 28 + 31 + 30 + Tag.
      Wenn Schaltjahr, erhöhe Jahrestag um 1.
      ENDE.
...
```

Damit ist die Aufgabe sicher zufriedenstellend gelöst. Es stellt sich die Frage, ob es nicht einfacher, kürzer und *eleganter* geht. Denn es ist nicht immer der erste Lösungsweg, der einem einfällt, auch der beste und effizienteste Weg. Setzt man versuchsweise alle Monate mit 31 Tagen an, werden ab März zu viele Tage genommen, und zwar:

März	April	Mai	Juni	Juli	Aug.	Sept.	Okt.	Nov.	Dez.
3	3	4	4	5	5	5	6	6	7

Nach einigem Nachdenken und/oder Probieren kommt man vielleicht auf eine höchst interessante Korrekturformel:

$$Y = 0,4 * Monat + 2,3 \tag{3.1}$$

Diese Formel liefert die Werte:

März	April	Mai	Juni	Juli	Aug.	Sept.	Okt.	Nov.	Dez.
3,3	3,9	4,3	4,7	5,1	5,5	5,9	6,3	6,7	7,1

Der ganzzahlige Teil der Ergebnisses stimmt offensichtlich mit den Fehlerwerten der vorhergehenden Tabelle überein. Mit dieser Formel kann man den Jahrestags-Algorithmus folgendermaßen definieren (**INT** nimmt den Ganzzahlanteil einer Zahl):

```
Eingabe Tag, Monat, Jahr.
Jahrestag = Tag + 31 * (Monat - 1).
Falls Monat $>$2 ist:
      Vermindere Jahrestag um INT(0,4 * Monat + 2,3) und
      falls Schaltjahr = JA ist, erhöhe Jahrestag um 1.
Ausgabe Jahrestag.
```

Wie man sieht, ist dieser Algorithmus wesentlich kürzer als der erste Ansatz – sowohl von der Laufzeit, als auch von Speicherbedarf her. Nachteilig gegenüber dem ersten Ansatz ist nur, dass dieser Algorithmus dem Leser nicht sofort einleuchtet und daher sorgfältig dokumentiert werden muß.

Beispiel 2: Primzahlberechnung

Gesucht wird ein Algorithmus, der feststellt ob eine gegebene Zahl eine Primzahl ist. Eine positive Zahl X ist bekanntlich dann eine Primzahl, wenn sie nur durch sich selbst und durch 1 teilbar ist. Ein erste Lösungsansatz ist:

1. Setze den Teiler T auf 2.
2. Ist X ohne Rest durch T teilbar, gib „nicht prim“ zurück und beende.

3. Erhöhe T um 1.
4. Ist T $<$ X, fahre fort bei Schritt 2.
5. Gib „prim“ zurück und beende.

Dieser Algorithmus ist (hoffentlich) **effektiv**, d. h. er leistet das Gewünschte. Ist er aber auch **effizient**, d. h. leistet er das Gewünschte bestmöglich? Die Antwort ist natürlich ein dickes „NEIN“. Es werden nämlich unnötig viele Zahlen getestet. Bereits durch kleine Modifikationen kann man den Aufwand für den Primzahlentest deutlich verringern:

- Ersetze den Test T $<$ X durch T * T $<$ X, denn der kleinste Teiler einer Nicht-Primzahl muß sicher kleiner als deren Quadratwurzel sein.
- Statt jede Zahl zu testen, testet man den Teiler 2, dann den Teiler 3 und erhöht danach den Teiler um 2, denn 2 ist die einzige gerade Primzahl.

Der geänderte Algorithmus lautet dann wie folgt:

1. Ist X ohne Rest durch 2 teilbar, gib „nicht prim“ zurück und beende.
2. Setze den Teiler T auf 3.
3. Ist X ohne Rest durch T teilbar, gib „nicht prim“ zurück und beende.
4. Erhöhe T um 2.
5. Ist T * T $<$ X, fahre fort bei Schritt 3.
6. gib „prim“ zurück und beende.

3.1.2 Lösungsstrategien

Programmieren heißt, den zeichnerisch oder verbal dargestellten Algorithmus in eine Programmiersprache umzusetzen und zu testen. Dabei werden die Schritte Codierung, Eingabe, Übersetzung und Testen zumeist wiederholt durchlaufen. Der Übersetzungslauf als gesonderter Schritt ist bei Sprachen mit Compiler, nicht aber bei solchen mit Interpreter erforderlich. Das Testen erfolgt nicht nur als Computertest, sondern auch als Schreibtischtest. Der Compiler hilft uns zwar, syntaktische Fehler zu finden; logische Fehler im Programm lassen sich aber nur durch Nachdenken und Schreibtischtests herausfinden.

Dokumentation: Ein verantwortungsbewußter Programmierer sollte seine Tätigkeit als Erfüllung eines **Kontraktes** auffassen: Zu einem vertraglich fixierten Problem wird ein Programm geliefert, welches das Problem mehr oder weniger effizient löst. Voraussetzung der genauen Erfüllung eines solchen Kontraktes ist eine **Spezifikation** des Problems mit der Eigenschaft, dass sowohl Auftraggeber als auch Auftragnehmer wissen, was sie unterschreiben. Das Programm als Vertragsgegenstand muß in einer Weise geliefert werden, dass es durch den Auftraggeber angewandt (benutzt) und gewartet werden kann; unentbehrliches Hilfsmittel hierfür ist die Dokumentation. Man unterscheidet zwei Arten von Dokumenten:

- **Gebrauchsanweisung (Benutzerhandbuch):** Dem Programmbenutzer, der mit der Programmentwicklung nichts zu schaffen hatte, braucht über die Art der Problemlösung, den Algorithmusentwurf und andere Interna nichts mitgeteilt zu werden. Folgendes sollte die Gebrauchsanweisung jedoch enthalten:
  - Zweck und Anwendungsbereich des Programms

  - Beschreibung der Steueranweisungen zum Starten und Anhalten des Programms
  - Beschreibung der Dialogführung mit Hinweis auf vorprogrammierte Hilfen
  - Form und Bedeutung der Eingabedaten
  - Form und Bedeutung der Ausgabedaten
  - Erläuterung der einzuhaltenden Wertebereiche
  - Erläuterung der Fehlermeldungen
  - Dialogbeispiele

  Zusätzlich zum Benutzerhandbuch sollte eine Kurzanleitung vorliegen, die nur die wichtigsten, für den Umgang mit dem Programm notwendigen Schritte und Anweisungen für den Interessenten bereithält.

- **Programmbeschreibung:** Für spätere Änderungen und Ergänzungen (die sogenannte Wartung) des Programms wird zusätzliche Dokumentation benötigt:
  - Name des Programmautors, Datum der Erstellung und Titel
  - Beschreibung des Problems
  - Modellbildung und Spezifikation
  - Entwicklung des Lösungsalgorithmus
  - Formulierung der Algorithmen in normierter Entwurfssprache
  - Programmquelle
  - Programmlauf
  - Diskussion
  - Überlegungen zur Korrektheit und zur Effizienz
  - Benutzte Literatur

Problemanalyse und Entwicklung des Algorithmus: Zentraler Teil der Programmentwicklung ist der Programmentwurf und nicht – wie es manchem Anfänger scheinen mag – die Programmierung bzw. Codierung in einer Programmiersprache.

Software-Engineering: Angesichts der steigenden Softwarekosten geht man immer mehr dazu über, die Programmentwicklung und dabei besonders den Programmentwurf industriell und ingenieurmäßig vorzunehmen: Software-Engineering lautet die darauf verweisende Begriffsbildung. Auf einige der im Rahmen des Software-Engineering eingesetzten Programmiertechniken sowie Entwurfsprinzipien gehe ich nun ein.

Entwurfstechniken und -prinzipien

Entwurfstechniken werden durch Begriffe wie Modularisierung, Normierung, Jackson-Methode, Top-Down-Entwurf, Bottom-Up-Entwurf, Unterprogrammtechnik, strukturierter Entwurf und Menütechnik geprägt. Im Folgenden werden diese Begriffe erläutert:

- Die **Modularisierung** von Software berücksichtigt, dass ein in kleine Teile bzw. Module gegliedertes Problem bzw. Programm einfacher zu bearbeiten ist. „Klein“ heißt, dass ein Modul nur eine maximale Anzahl von Anweisungen umfassen darf. Ein Modul ist ein Programmteil mit einem Eingang und einem Ausgang und kann selbstständig übersetzt und ausgeführt werden. Module verkehren nur über Schnittstellen miteinander, über die Werte (Parameter genannt) vom rufenden an das aufgerufene Modul übergeben werden; ein Modul darf als Black Box nichts vom Innenleben eines anderen Moduls wissen.

- Die **Normierung** von Programmabläufen als Vereinheitlichung durch eine standardisierte Ablaufsteuerung wird bei der Entwicklung komplexer kommerzieller Softwarepakete vorgenommen, an der zumeist mehrere Mitarbeiter beteiligt sind. Oftmals hat jedes Softwarehaus seine eigenen Normen.
- Die **Jackson-Methode** geht bei der Programmentwicklung von der exakten Analyse der Datenstrukturen aus, um dann die entsprechenden Programm- bzw. Ablaufstrukturen zu entwerfen. Denn in der kommerziellen DV sind die Daten zumeist bis in die Details vorgegeben, während die Abläufe den Daten gemäß formuliert werden müssen. Anders ausgedrückt: Die Datenstruktur prägt die Programmstruktur.
- Dem **Top-Down-Entwurf** als Von-oben-nach-unten-Entwurf entspricht die Technik der schrittweisen Verfeinerung: Vom Gesamtproblem ausgehend bildet man Teilprobleme, um diese dann schrittweise weiter zu unterteilen und zu verfeinern bis hin zum lauffähigen Programm. Der Top-Down-Entwurf führt immer zu einem hierarchisch gegliederten Programmaufbau. Es besteht jedoch die Gefahr, dass erst in einer relativ tiefen Ebene bemerkt wird, dass die Realisierung unmöglich ist. Der Entwurf muß dann völlig neu begonnen werden.
- Der **Bottom-Up-Entwurf** als Gegenstück zum Top-Down-Entwurf geht als Von-unten-nach-oben-Entwurf von den oft verwendeten Teilproblemen der untersten Ebene aus, um sukzessive solche Teilprobleme zu integrieren. Hier besteht die Gefahr, dass zum Schluß die Lösung nicht zum Problem passt. Beide Entwurfsprinzipien werden in der Praxis zumeist kombiniert angewendet. Man versucht beim Top-Down-Entwurf, schon möglichst bald die Probleme bei der endgültigen Realisierung zu erkennen und diese dann Bottom-Up zu lösen.
- Die **Unterprogrammtechnik** wird in folgenden Fällen genutzt:

  1. Ein Ablauf wird mehrfach benötigt.
  2. Das Programm wird in seiner Struktur zu unübersichtlich.
  3. Mehrere Personen kooperieren und liefern Unterprogramme ab.
  4. Menügesteuerter Dialog (Menütechnik).

  Der Begriff des Unterprogramms (Funktion, Prozedur) entspricht dabei in etwa dem des Moduls. Die bekannteste Schnittstelle ist der Unterprogrammaufruf mit Parameterübergabe.
- Der **strukturierte Entwurf** beinhaltet, dass ein Programm unabhängig von seiner Größe nur aus den vier (später erklärten) grundlegenden Programmstrukturen aufgebaut sein darf:

  – Folgestrukturen
  – Auswahlstrukturen
  – Wiederholungsstrukturen
  – Unterprogrammstrukturen

  Dabei soll auf unbedingtes Verzweigen aus einer Struktur heraus verzichtet werden. Jede Programmstruktur bildet einen Strukturblock. Blöcke sind entweder

  – hintereinander angeordnet oder
  – vollständig eingeschachtelt.

  Die teilweise Einschachtelung (Überlappung) ist nicht zulässig. Sogenannte blockorientierte Sprachen wie C, Pascal, Modula-2 oder Perl unterstützen das Prinzip des strukturierten Entwurfs weit mehr als die unstrukturierten Sprachen wie Assembler oder das Ur-BASIC.

Die oben nur stichwortartig dargestellten Prinzipien dürfen nicht getrennt betrachtet werden; unter dem Informatik-Sammelbegriff „strukturierte Programmierung“ faßt man sie zu einem heute allgemein anerkannten Vorgehen zusammen. Die tragenden Prinzipien sind:

1. Top-Down-Entwurf mit der schrittweisen Verfeinerung.
2. Strukturierter Entwurf mit der Blockbildung.

3.2 Entwicklung und Darstellung des Algorithmus

Die Anweisungen (Befehle) eines Programms bestehen aus wenigen, festen Strukturen. Im Folgenden werden diese Strukturen besprochen und ihre Darstellung als Programmablaufplan (PAP) und Struktogramm gezeigt. Die Struktogramme verzichten auf Ablauflinien und vermeiden so die Gefahr der Unübersichtlichkeit. Zudem ist es mit dieser Darstellung schwerer, unbedingte Sprünge aus Anweisungsstrukturen heraus darzustellen, die vielfach zu Fehlern oder zu *trickreicher* Programmierung führen („Spaghetti-Code“).

Die Strukturierung von Programmen in Anweisungsblöcke mit fest definierten Eintritts- und Austrittspunkten bietet zudem die Möglichkeit, Programme zu verifizieren, d. h. Programme mit garantierter Zuverlässigkeit zu erstellen, indem für jeden Anweisungsblock eine wohldefinierte Beziehung zwischen den Werten einer Variablen vor und nach Durchlaufen einer Anweisungsstruktur festgelegt werden kann (das dies bei großen Programmsystemen aus Zeitgründen nicht gemacht wird, ändert nichts am Faktum).

3.2.1 Programmstrukturen

Es gibt vier grundlegende Programmstrukturen: Sequenz (Folge), Auswahl, Wiederholung und Unterprogramm. Grundlegend sind sie in zweifacher Hinsicht:

- Zum einen gelangt man beim Zerlegen noch so umfangreicher Programmabläufe immer auf diese vier Programmstrukturen als Grundmuster.
- Zum anderen kann jeder zur Problemlösung erforderliche Programmablauf durch geeignetes Anordnen dieser vier Programmstrukturen konstruiert werden.

Grundsätzlich lassen sich Programmstrukturen beliebig schachteln, d. h. innerhalb einer Programmstruktur kann wieder eine andere Programmstruktur ausgeführt werden und so fort. Betrachtet man die Strukturen etwas genauer, ergeben sich nur wenige Möglichkeiten:

Sequenz (Folgestruktur)

Linearer Ablauf: Jedes Programm besteht aus einer Aneinanderreihung von Anweisungen an den Computer. Die Sequenz ist eine Folge von Anweisungen, die in der Reihenfolge ihrer Niederschrift ausgeführt werden. Man spricht deshalb auch vom linearen Ablauf bzw. unverzweigten Ablauf, vom Geradeaus-Ablauf oder von einer Sequenz. In verschiedenen Programmiersprachen wird die Klammerung einer solchen Anweisungsfolge unterstützt (Pascal: BEGIN ... END, Perl, C: { ... }).

Auswahl (Bedingte Verzweigung, Mehrfachauswahl)

Sie ist gekennzeichnet durch einen nicht linearen Ablauf mit einer Vorwärtsverzweigung. Der Ablauf gelangt an einen Entscheidungspunkt, an dem, abhängig von einer Bedingung, unterschiedliche Verarbeitungswege eingeschlagen werden. Das Entscheidungssymbol gibt eine Bedingung an (i. a. ein bedingter Ausdruck), deren Ergebnis ein Wahrheitswert ist (WAHR oder FALSCH).

- **Einseitige Auswahl:** Diese Alternativstruktur führt nur auf einem der beiden Verzweigungspfaden eine Anweisung(sfolge) aus und endet in der Zusammenführung beider Pfade.

```
{\bf WENN} Bedingung {\bf DANN} Anweisungsfolge;
```

- **Zweiseitige Auswahl:** Bei dieser Alternativstruktur führt jeder Verzweigungspfad auf jeweils eine eigene Anweisungsfolge. Sie endet auch wieder in einer Zusammenführung der Pfade.

```
{\bf WENN} Bedingung {\bf DANN} Anweisungsfolge 1\\
{\bf SONST} Anweisungsfolge 2;
```

- **Mehrfachauswahl:** Bei dieser Struktur gibt es mehr als zwei Auswahlpfade, die aus einer Verzeigung ihren Ausgang nehmen und in einer Zusammenführung enden. Hier erfolgt die Abfrage nicht nach einer Bedingung, sondern der bedingte Ausdruck liefert einen Wert. Für jeden möglichen Ergebniswert ist ein Zweig vorgesehen. Existiert nicht für jeden möglichen Ergebniswert der Bedingung ein Pfad, ist ein zusätzlicher Pfad für alle nicht behandelten Fälle vorzusehen (SONST-Zweig).

```
{\bf WENN} bedingter Ausdruck \\
{\bf Wert 1:} Anweisung 1; \\
{\bf Wert 2:} Anweisung 2; \\
\dots ; \\
{\bf Wert n:} Anweisung n; \\
{\bf SONST} Anweisung s; \\
```

(Wenn der Ausdruck den Wert „Wert i“ besitzt, wird die Anweisungsfolge „Anweisung i“ ausgeführt).

Wiederholung

Wiederholungsstrukturen ergeben sich, wenn eine Anweisungsfolge zur Lösung einer Aufgabe mehrfach durchlaufen werden soll (z. B. das Bearbeiten aller Komponenten eines Vektors oder Berechnung des Monatslohns aller Mitarbeiter). Es liegt ein nichtlinearer Verlauf mit Rückwärtsverzweigung vor. Die Programmierung einer Wiederholungsstruktur führt zu einer sogenannten Programmschleife. Wichtig ist die Terminierung der Schleife, d. h. mindestens eine Anweisung muß dafür sorgen, dass nach einer endlichen Zahl von Durchläufen die Bedingung für die Wiederholung nicht mehr erfüllt ist.

- **Abweisende Wiederholung (Eingangsbedingung):** In diesem Fall steht die Bedingung zu Beginn der Schleife (also vor der Anweisungsfolge). Ist die Bedingung schon beim Eintritt in die Anweisungsstruktur nicht erfüllt, wird die Anweisungsfolge überhaupt nicht ausgeführt.

  SOLANGE Bedingung **FÜHRE AUS** Anweisungsfolge;

- **Nichtabweisende Wiederholung (Ausgangsbedingung):** In diesem Fall steht die Bedingung am Ende der Schleife (also nach der Anweisungsfolge). Die Anweisungsfolge wird auf jeden Fall mindestens einmal ausgeführt.

  FÜHRE AUS Anweisungsfolge **BIS** Bedingung;

- **Zählschleife:** Diese Anweisungsstruktur stellt eine Sonderform der abweisenden Wiederholung dar. Eine Zählvariable wird vor dem Durchlaufen der Schleife mit einem

Anfangswert besetzt. Darauf folgt eine abweisende Wiederholung, wobei in der Abbruchbedingung die Zählvariable auf das Erreichen eines Endwerts geprüft wird. Die Anweisungsfolge wird um eine Anweisung zum Inkrementieren bzw. Dekrementieren der Zählvariable erweitert. Die Zählvariable muß skalar sein (z. B. ganzzahlig).

VON Zählvariable = Startwert **BIS** Zählvariable = Endwert **FÜHRE AUS** Inkrement/Dekrement Zählvariable, Anweisungsfolge;

Unterablaufstrukturen (Unterprogramme)

Es wurde bereits die Aufteilung von Programmen in einzelne Module angesprochen. Diese Aufteilung auch in einem Programm zu vollziehen, erscheint daher wünschenswert. In allen höheren Programmiersprachen und auch im Befehlsumfang nahezu aller Prozessoren ist diese Möglichkeit in Form von Unterprogrammen realisiert. Für die Verwendung von Unterprogrammen (UP) sprechen weitere Gründe:

- Übersichtlichkeit
  Durch die Verwendung von UP steigen Lesbarkeit und Verständlichkeit des Programms, da sich UP als direkte Abbildung eines Funktionsblocks unter einem aussagekräftigen Namen ins Programm einfügen lassen. Zudem erlauben sie den vom Rest des Programms unabhängigen Test von einzelnen Funktionsblöcken und machen so auch große Programme überschaubar und (relativ) fehlerfrei.

- Wirtschaftlichkeit
  Häufig werden die gleichen Anweisungsfolgen für unterschiedliche Daten und an verschiedenen Stellen im Programm benötigt. Durch ein Unterprogramm kann eine Anweisungsfolge definiert werden, die dann mehrfach und mit unterschiedlichen Daten „aufgerufen“ werden kann → Code wird kürzer → weniger Speicherbedarf.

- Änderungsfreundlichkeit
  Änderungen (z. B. aufgrund neuer Hardware) und Optimierungen betreffen immer nur einige wenige Unterprogramme. Da die Verbindung zum Rest des Programms über eine fest definierte Schnittstelle erfolgt, wirken sich Änderungen in einem Unterprogramme normalerweise nicht auf den übrigen Programmcode aus. Voraussetzung dafür sind möglichst unabhängige Unterprogramme.

- Lokalität
  Die lokale Begrenztheit von Variablen kann durch die Vereinbarung einzelner Variablen im UP hervorgehoben und unterstützt werden. Viele Hochsprachen erlauben es, solche Variablen in ihrer Gültigkeit auf das Unterprogramm zu beschränken (siehe später). Wird der gleiche Variablenname in verschiedenen Unterprogrammen verwendet, ergeben sich keine Konflikte der lokalen Variablen untereinander.

- Allgemeingültigkeit
  Durch die Erarbeitung möglichst allgemein gültiger Unterprogramme kann eine Unterprogrammbibliothek erstellt werden, deren Inhalt immer wieder bei Programmierproblemen herangezogen werden kann. Die Zeit zum Erstellen neuer Programme wird verkürzt. Sind die Bibliotheksroutinen sorgfältig getestet, sinkt auch die Fehlerrate bei neu erstellten Programmen. Vielfach lassen sich Bibliotheken zur Lösung spezieller Probleme auch käuflich erwerben.

- Programmentwicklung im Team
  Mehrere Mitarbeiter entwickeln jeder für sich Unterprogramme mit fest definierten Schnittstellen für die Ein- und Ausgabe von Daten. Die Unterprogramme werden später zum Gesamtprogramm zusammengeführt.

Ein Unterprogramm muss vor seiner Verwendung vereinbart werden. Die Anweisungsfolge des Unterprogramms wird unter einem möglichst aussagekräftigen Namen zusammengefasst. Der Aufruf besteht dann nur noch in der Nennung des Unterprogrammnamens. Dem Unterprogramm können beim Aufruf aktuelle Werte zur Verarbeitung übergeben werden: „Argumente“ oder „Parameter“.

Einige Programmiersprachen unterscheiden bei Unterprogrammen Funktionen und Prozeduren (z. B. Pascal). „Funktionen“ sind Unterprogramme, die einen Wert zurückliefern und daher auf der rechten Seite einer Wertzuweisung auftreten dürfen (z. B. Y = SIN(X), SIN ist eine Funktion mit einem Real-Argument, die einen Real-Wert zurückliefert). „Prozeduren“ sind Unterprogramme, die keine Werte zurückliefern. Bei anderen Sprachen (etwa Perl oder C) gibt es ausschließlich Funktionen. Zusätzlich gibt es einen Datentyp VOID, den leeren Datentyp, mit dem Funktionen gekennzeichnet werden, die keinen Wert zurückliefern. Für die technische Realisierung von Unterprogrammen auf Maschinenebene bieten sich zwei Möglichkeiten an:

- Offene Unterprogramme
  Bei dieser Form wird der Code des UP an der jeweiligen Aufrufstelle einkopiert. Das UP bildet also nur auf der Ebene der Programmiersprache eine abgeschlossene Einheit und ist im fertigen Programmcode unter Umständen mehrfach enthalten. Diese Form wird auch als „Makro-Technik“ bezeichnet (naives Einkopieren).

- Geschlossene Unterprogramme
  Bei dieser Form existieren speziellen Befehle für den Ansprung eines Unterprogramms und den Rücksprung zur Aufrufstelle. Das Unterprogramm ist als Code nur einmal vorhanden. Bei jedem Aufruf wird die aktuelle Position in der Anweisungsfolge zwischengespeichert und das Unterprogramm ausgeführt. Danach wird die Programmausführung an der Aufrufstelle fortgesetzt. Wir werden uns nur mit dieser Unterprogrammtyp befassen.

Anmerkungen:

- Es gibt verschiedene Bezeichnungen für den Unterprogrammsprung: Jump-to-Subroutine (Maschinensprache), Function-Call, Funktionsaufruf, Procedure-Call, ...

- Der Rücksprung muss nur in Maschinensprache explizit programmiert werden, z. B. als Return-from-Subroutine-Befehl (RTS); in höheren Programmiersprachen übersetzt der Compiler das Ende einer Funktion in den entsprechenden Maschinenbefehl.

- Von beliebigen Stellen des Hauptprogramms aus kann

  dasselbe Unterprogramm aufgerufen werden. Es wird jedoch dafür gesorgt, dass der entsprechende Rücksprung jeweils zu der Anweisung erfolgt, die im Hauptprogramm auf den zuletzt ausgeführten Unterprogrammsprung folgt.

3.2.2 Darstellung von Algorithmen

Algorithmen lassen sich auf unterschiedliche Weise darstellen:

- **Verbale Beschreibung:** Der Algorithmus wird in natürlicher Sprache beschrieben. Man kann dies in einer Aufzählung der Befehle machen oder zur Vereinfachung und um die Beschreibung eindeutig zu gestalten, die sprachliche Beschreibung formalisieren; beispielsweise:

  WENN x größer 0 **DANN** berechne Y **SONST** Fehlermeldung

  Also genau das, was bisher hier verwendet wurde.

- **Programmablaufplan (PAP):** Dies ist eine zeichnerische Darstellung mit genormten Symbolen (DIN 66001). Die Verarbeitungssymbole werden von geometrischen Formen umschlossen, deren Bedeutung genormt ist. Ihre Verknüpfung erfolgt mit einer Verbindungslinie, deren Durchlaufrichtung mit Pfeilen markiert ist. Der PAP wird oft auch „Flussdiagramm“ genannt. Der Vorrat an Symbolen ist relativ groß, denn es sind auch Symbole für z. B. Plattenspeicher und Ähnliches dabei. Bild 3.2 zeigt nur die wichtigsten von ihnen.

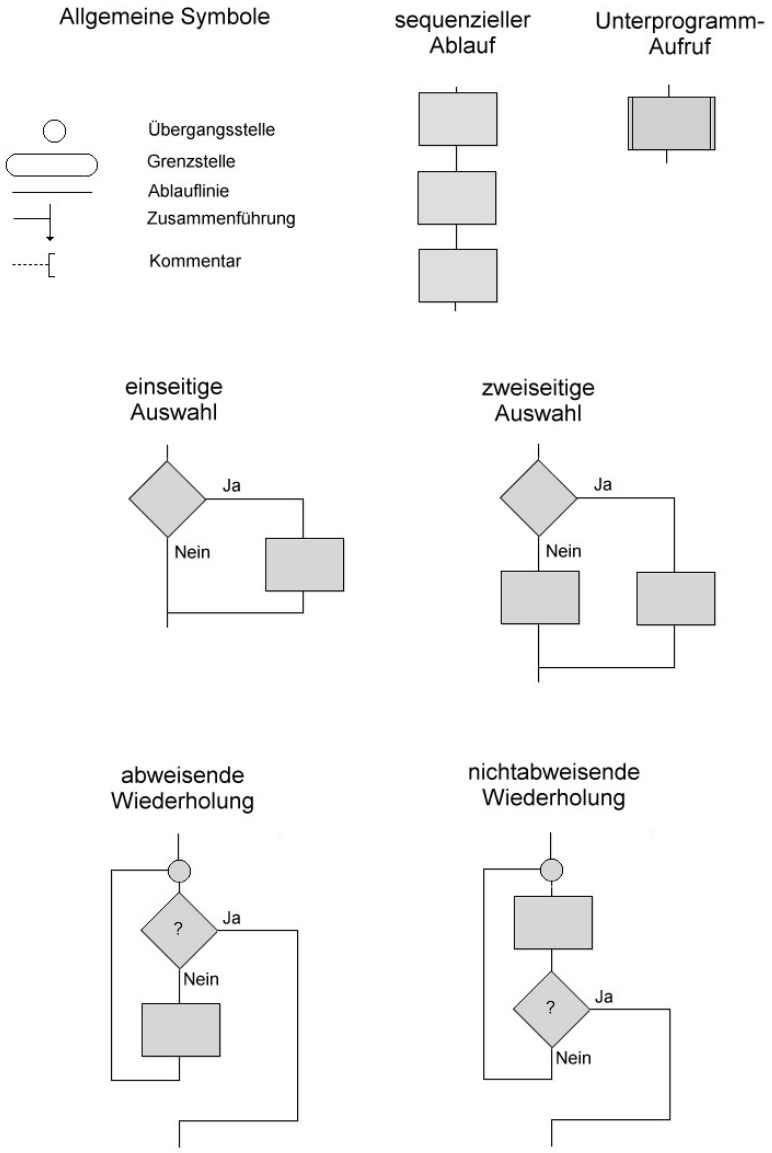

Bild 3.2: Einige Elemente eines PAP

- **Struktogramm:** Auch dies ist eine zeichnerische Darstellung (Nassi-Shneiderman), die jedoch auf Ablauflinien und Übergangsstellen verzichtet (bessere Übersichtlichkeit). Die Beschreibung des Algorithmus besteht aus geschachtelten Strukturelementen, was dafür sorgt, dass keine wilden Verzweigungen im Programmablauf vorgenommen werden können. Bild 3.3 zeigt die Elemente eines Struktogramms.

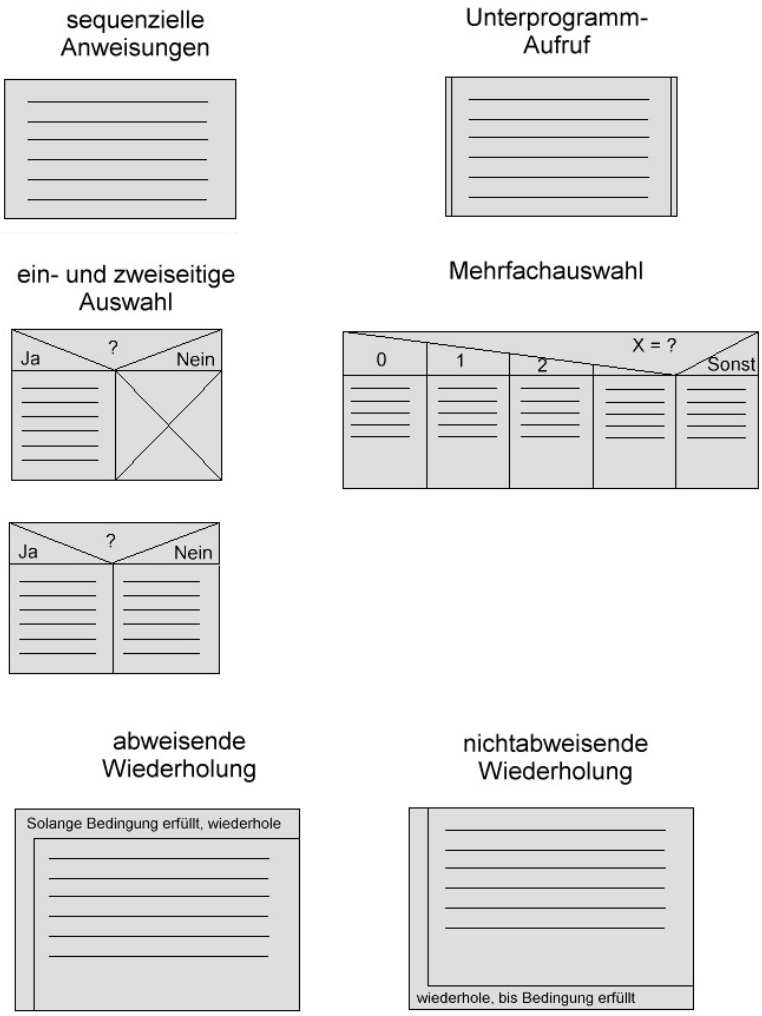

Bild 3.3: Elemente von Struktogrammen

3.3 Programmentwicklung

Dieser Abschnitt will Ihnen die Techniken der Programmentwicklung ein wenig näher bringen, denn die Programme, die in diesem Skript dargestellt sind, haben wegen ihrer Kürze wirklich nur Beispielcharakter. Bei der Leistungsfähigkeit der heutigen Großrechner können Programmsysteme mit 100 000 oder mehr Zeilen entstehen. Es zeigte sich in den sechziger Jahren, dass die Programme ab einer bestimmten Größe stets Fehler enthielten. Die Fehler ließen sich auch fast nie vollständig beseitigen, weil die Korrekturen neue Fehler zur Folge hatten.

Aus diesem Grund entwickelte sich eine neue Disziplin der Informatik, das „Software Engineering“. Im Rahmen dieser Disziplin werden Modelle und Methoden entworfen, mit denen auch große und größte Programmsysteme so geschrieben werden können, dass sie fehlerfrei arbeiten. Die wichtigsten Kriterien, denen solche Programme genügen, sind:

- **Zuverlässigkeit und Korrektheit**
  Für jede korrekte Eingabe wird das nach dem gewählten Algorithmus richtige Ergebnis abgeliefert. Falsche Eingaben werden, soweit möglich, vom Programm erkannt, abgefangen und gemeldet. Ausfälle der Hardware können abgefangen oder in ihren Aus-

wirkungen stark gemildert werden (Datensicherung, Wiederaufsetzen von Aufträgen). Durch die Zerlegung des Problems in kleine Häppchen ist schon ein hohes Maß an Korrektheit gesichert, da die Teilprobleme übersichtlich darzustellen sind. Da für jeden Programmteil Eingabegrößen, Ausgabegrößen und der Lösungsweg bereits feststeht, da schon im Entwurf festgelegt worden, wird die „trickreiche“ Programmierung weitgehend verhindert.

Durch strukturierte Programmierung allein kann man noch keine Zuverlässigkeit erreichen. Es ist schließlich auch möglich, die dümmsten Programme sauber zu strukturieren. Aber durch die Strukturierung ist es sehr einfach, Prüf- und Testalgorithmen einzufügen. Weiterhin kann die Datensicherung und der Neustart von Programmen durch definierte Übergänge zwischen den Programmteilen (Modulschnittstellen) erleichtert werden.

- **Benutzerfreundlichkeit**
  Für den gegebenen Benutzerkreis stellt das Programm keine zu hohen Anforderungen von der Ein- und Ausgabe her, die Fragen sind dem Benutzer verständlich; unvollständige Eingaben resultieren in Nachforderungen; Fehlermeldungen sind im Klartext und geben Hinweise auf die Behebung des Fehlers (also nicht „ERROR EXIT 05 AT 54767“, sondern „FEHLER IN FUNKTION NULLSTELLE WEGEN DIVISION DURCH 0“).

- **Flexibilität**
  Wenn sich ein neuer Benutzerwunsch ergibt, so bedeutet das im Allgemeinen eine Änderung im Programm. Bei einem gut strukturierten Programm beschränkt sich die Änderungsarbeit meist auf nur wenige Teilprobleme (Anpassbarkeit, Adaptibilität). Bei nicht strukturierten Programmen traten bei den Änderungen oft Fehler an ganz anderen Stellen im Programm auf, weil dort beispielsweise Variablen für einen völlig anderen Zweck verwendet wurden. Andererseits soll es möglich sein, ohne große Schwierigkeiten die Hardware zu wechseln (Übertragbarkeit, Portabilität).

- **Lesbarkeit**
  Wenn Sie sich daran gewöhnt haben, die Programme „von unten nach oben“, das heißt, erst den Anweisungsteil des Hauptprogramms und dann die Unterprogramme zu lesen, erhalten Sie für jedes Teilproblem eine kurze Beschreibung, die die Lösung des Problems in übersichtlicher Form enthält. Für Details wird auf Unterprogramme verwiesen, die die Teilprobleme in analoger Weise behandeln. Für die Lesbarkeit von Programmen ist also entscheidend, dass

  1. aussagekräftige Namen verwendet werden,
  2. relativ kurze Unterprogramme und Programmteile geschrieben werden,
  3. viele Kommentare im Text stehen und
  4. jeder Programmteil sauber strukturiert ist (z. B. Einrückungen).

  Die Punkte 2 und 4 sind Ergebnisse der strukturierten Programmierung; Punkte 1 und 3 stellen ein ungeschriebenes Gesetz dar. Lesbarkeit ist Voraussetzung für *Wartbarkeit*!

- **Effizienz**
  Die Effizienz kann unter der Strukturierung insofern leiden, dass C-Programme redundanter sind als z. B. in Assembler geschriebene und strukturierte Programme redundanter als nicht strukturierte. Gegenüber den Vorteilen, die sich bei Änderungen ergeben und in Bezug auf die Sicherheit des Ablaufs bei der Ausführung der Programme spielen die Effizienzverluste kaum noch eine Rolle. (Es wird hier von den Problemen bei der Echtzeitverarbeitung von Daten abgesehen, wo die Effizienz eine dominierende Rolle spielt.)

Das Programm kann an der benötigten Laufzeit und an den Speicheranforderungen gemessen werden. Auf dieses Kriterium darf aber erst dann geachtet werden, wenn eine fertige Version des Programms vorliegt, die allen anderen Anforderungen genügt. Erst dann können zusätzliche Effizienzverbesserungen, ausführlich erläutert, toleriert werden.

Es ist klar, dass solche Anforderungen besondere Methoden der Programmentwicklung notwendig machen. Die Programmentwicklung beginnt nicht mit der Programmierung, sondern schon lange vorher, die Programmierung ist eine der letzten Stufen der Programmentwicklung. Zuerst muß das programmtechnisch zu lösende Problem genau analysiert werden. Diese Problemanalyse ist auch wieder in einzelne Phasen unterteilt, die im Folgenden kurz umrissen wird. Zuerst wird die gegebene Situation genau untersucht, wobei die Wünsche für die EDV und die Einsatzmöglichkeiten der EDV festgehalten werden (Ist-Analyse). Ausgehend von der Beschreibung der Probleme wird dann eine Beschreibung der Lösungen für den Anwender und eine Aufgabendefinition für den Hersteller der Software erstellt (Sollkonzept). Nun wird festgestellt, ob es möglich ist, eine derartige Aufgabe überhaupt zu lösen (technische Durchführbarkeit) und ob die gefundenen Lösungen wirtschaftlich sind (ökonomische Durchführbarkeit). Die optimale Lösungsmethode wird vervollständigt (Projektplanung).

Der Aufwand der Vorarbeiten zu einem Programm ist natürlich von der Größe des Projekts abhängig, das Planungsprinzip ist aber auch auf kleine Programme anwendbar. Auf jeden Fall sollten die Aufgabenstellung, die Lösungsmethoden, die Benutzerwünsche, die Qualifikation der Benutzer und die im Rahmen der Problemanalyse zutage getretenen Argumente schriftlich fixiert werden.

In diesem Kapitel bleiben Sie von theoretischen Betrachtungen nicht ganz verschont, aber gerade die in diesem Kapitel unternommenen Betrachtungen sind sehr wichtig für die praktische Programmierarbeit.

Es wurden schon kurz Fehler erwähnt, die sich bei der Erstellung großer Programme einschleichen. Damit Ihnen später nicht das Gleiche passiert, wird Ihnen im folgenden Abschnitt gezeigt, wie das Problem und nachher das Programm strukturiert und modularisiert, das heißt in kleinere Teile zerlegt wird. Die Strukturierung wird von Perl durch das Sprachkonzept unterstützt und gefördert, was aber nicht bedeutet, dass man in Perl keine unverständlichen und unlesbaren Programme schreiben kann.

Im Rahmen der Projektplanung wird die *Aufgabe* des Programms beschrieben, über die Entwicklung des Programms wollen wir hier sprechen. Ausgehend von der Aufgabenstellung entsteht durch schrittweise Verfeinerung des Problems schließlich ein fertiges Programm. Die Vorgehensweise ist immer dieselbe, egal wie groß das Programm wird:

Das Problem wird in einzelne Teilprobleme zerlegt. Dies können Funktionen sein oder aber auch sogenannte Objekte (Gegenstände). Die Kunst besteht darin, die Zerlegung derart intelligent zu gestalten, dass die Teilprobleme möglichst nicht voneinander abhängen (Prinzip der starken lokalen Bindung) und das Teilproblem in sich alleine – ohne Zuhilfenahme der Lösung anderer Teilprobleme der gleichen Zerlegungsstufe – zu lösen ist (Prinzip der möglichst losen Koppelung). Diese so gefundenen Teilprobleme werden weiter zerlegt (verfeinert), bis die einzelnen Verfeinerungsstufen so wenig umfangreich sind, dass sie problemlos programmiert werden können (Top-Down-Design). Diese Methode sichert sauber strukturierte und verifizierbare Programme oder Objekte, wenn Sie die folgenden Grundregeln beachten.

- Jeder Zerlegungsschritt sollte auf einer, höchstens zwei DIN-A4-Seiten beschrieben werden. Die Beschreibung besteht aus:

  1. den erforderlichen Voraussetzungen
  2. den nachzuprüfenden Anforderungen

3. der Zerlegung in Teilprobleme
4. einer algorithmischen Beschreibung des Zusammenhangs der Teilprobleme

- Alle Teilprobleme sollen etwa den gleichen Umfang besitzen.
- Eine klare Gliederung muß den Überblick garantieren. In Bild 3.4 ist die Zerlegung in Teilprobleme skizziert. Wie Sie sehen, zeichnet sich eine Struktur von einzelnen Verfeinerungsstufen oder Schichten ab.

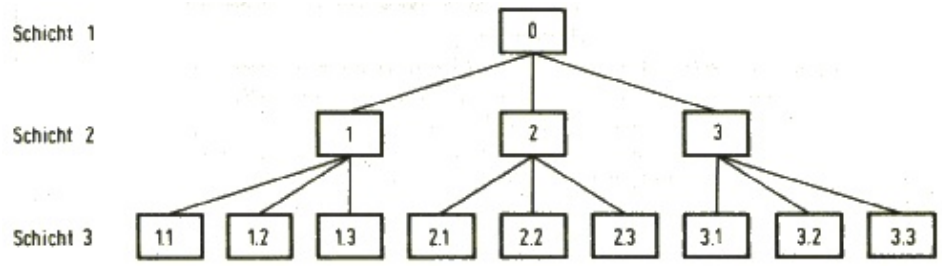

Bild 3.4: Die Teilprobleme des Programms lassen sich in Ebenen gliedern

Unter einer solchen Schicht können Sie die Beschreibung des Problems in einem bestimmten Abstraktionsgrad verstehen. Sie können sich auch vorstellen, dass auf jeder Schicht ganz bestimmte, abstrakte Grundoperationen existieren. Wenn Sie sich zum Beispiel an den Kreisalgorithmus vom Kapitelanfang erinnern, so existiert in einer Schicht die Operation der Multiplikation, in der nächst niedrigeren gibt es die Multiplikation nicht mehr, sondern nur noch die Addition.

Die Grundoperationen werden von Schicht zu Schicht nach unten einfacher. Sie bewegen sich so von sehr abstrakten und komplexen Konstruktionen zu konkreten Operationen. In der Informatik wird eine Schicht als abstrakte Maschine aufgefasst, also als ein gedachter Computer, der alle Operationen der entsprechenden Detaillierungsstufe beherrscht. Aus diesem Gedankenkonzept folgt sofort, dass auch die Verfeinerung schichtweise geschehen soll. Eine Schicht sollte daher erst vollständig aufgebaut sein, bevor die nächste Verfeinerung in Angriff genommen wird. Diese Vorgehensweise hat einen sehr wesentlichen Vorzug, denn alle Teilprobleme haben etwa den gleichen Abstraktionsgrad und ungefähr den gleichen Umfang (Bild 3.5). Es kann also nicht vorkommen, dass Sie sich einerseits um die Optimierung eines Algorithmus bemühen, während Sie in einem anderen Teilproblem noch nach dem Lösungsweg suchen. Oder Sie stellen fest, dass ein Teilproblem der Stufe zwei unlösbar ist, während der Rest bereits völlig fertig ist.

Jedes Teilproblem hat einen gewissen Abstraktionsgrad, der für alle Probleme etwa gleich ist. Wenn Sie an die Teilprobleme gewisse Anforderungen stellen, wird die Korrektheit des Entwurfs überprüfbar; man sagt dann, das Programm sei dann verifizierbar. Die Verifizierbarkeit eines Programms ist wohl die wichtigste Folge der strukturierten Programmierung. Die Beschreibung eines Teilproblems besteht aus

- den benötigten Voraussetzungen
- den abzuliefemden Ergebnissen
- dem Algorithmus zur Lösung des Problems auf der Basis von Grundoperationen und Operationen, die Probleme der nächstniedrigeren Schicht darstellen.

Aus dieser Beschreibung lassen sich dann die Voraussetzungen und Anforderungen für die Probleme der nächsten Schicht definieren. Ein derart beschriebenes Teilproblem wird Programmmodul, kurz Modul genannt. Ein Modul hat also Eingänge (die Voraussetzungen),

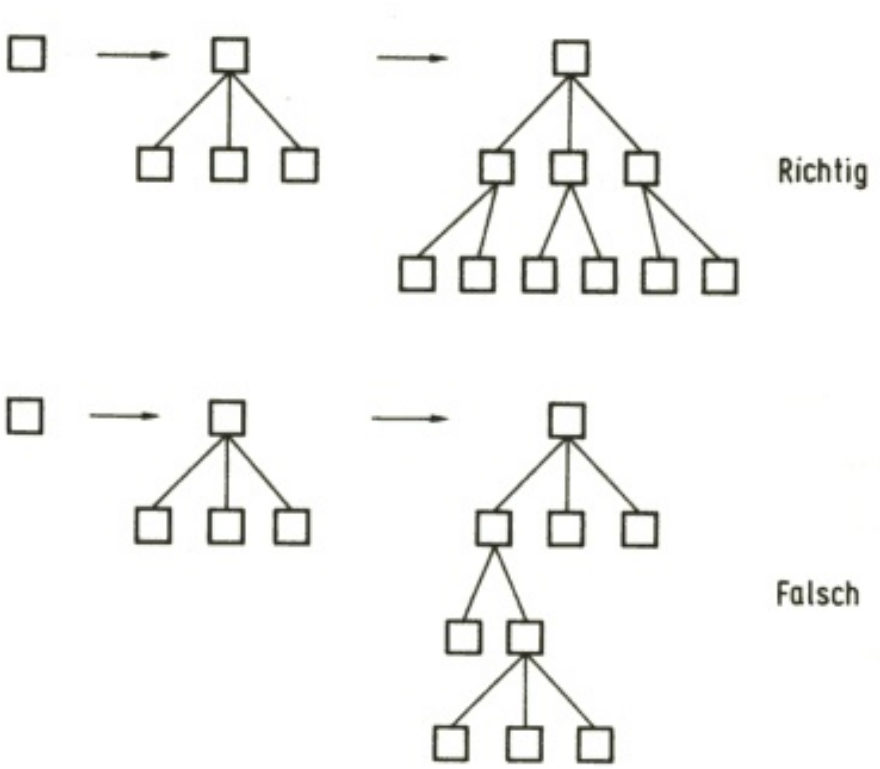

Bild 3.5: Der Abstraktionsgrad der Ebenen sollte bei allen Teilproblemen ungefähr gleich groß sein

Ausgänge (die Ergebnisse) und eine genau festgelegte Funktion mit definierten Operationen.

Zum Schluss dieses Abschnitts will ich noch einige Bemerkungen zu den „Programming Standards“ machen. Das sind Vorschriften und Regeln, an die sich alle an einem Projekt beteiligten Programmierer zu halten haben. Diese Regeln können Länge und Komplexität von Unterprogrammen betreffen, sie können Bezeichnungskonventionen oder Schreibweisen betreffen oder auch Regeln für die Erstellung von Kommentaren sein.

Ein Beispiel für einen einfachen „Programming Standard“ wäre: *Ein Unterprogramm soll höchstens ein bis zwei DIN-A4-Seiten umfassen. Namen beginnen mit „K-“, falls es sich um Konstanten handelt. Zu jedem Unterprogramm werden im Kommentar am Anfang des Unterprogramms der Verfasser, das Herstellungsdatum, die Änderungsdaten und eine genaue Beschreibung der Parameter festgehalten. Jeder Sprung ist ausführlich zu kommentieren.*

Solche Regeln dienen der übersichtlichen Gestaltung von Programmen und damit auch wieder der Lesbarkeit. In unseren Beispielen folgen wir auch einem gewissen Standard (wenn auch nicht sklavisch) durch Einrücken der einzelnen Strukturelemente. Diese optische Wiederholung der Programmstruktur bringt eine erhebliche Verbesserung der Übersichtlichkeit und so bessere Lesbarkeit. Grundgedanke bei der Erstellung der Einrückungsregeln war, dass Anweisungen, die von anderen Anweisungen kontrolliert werden, um eine feste Anzahl von Spalten eingerückt werden.

3.4 Qualitätssicherung bei Software

Bisher wurde vom Algorithmus, von Datenstrukturen und Programmen gesprochen. Sowohl beim Entwickeln des Algorithmus als auch beim Programmieren schleichen sich Fehler ein. Der Softwarebenutzer erwartet aber ein fehlerfreies Produkt. Streckenweise ist dies sogar überlebenswichtig. Beispielsweise dürfen die Avioniksoftware oder auch die Software zur Steuerung von Geräten in der Intensivmedizin keine Fehler aufweisen, deren Auswirkungen in irgendeiner Weise ein Menschenleben gefährden können. Ziel einer jeden Softwareentwicklung muss es demnach sein, ein „möglichst" fehlerfreies Produkt zu erstellen. Fehlerfreiheit ist aber nur ein Qualitätsaspekt. Vor allem müssen auch die dem Nutzer zugesicherten Produkteigenschaften vorhanden sein usw.. Allgemein lässt sich eine sogenannte Softwarequalität definieren:

Softwarequalität ist die Gesamtheit der Merkmale und Merkmalswerte eines Softwareproduktes, die sich auf dessen Eignung beziehen, festgelegte und vorausgesetzte Erfordernisse zu erfüllen (nach DIN ISO 9126).

Wichtig ist dass die Erfüllung der obigen Erfordernisse quantitativ beschreibbar vorgegeben und somit objektiv überpüfbar ist. Gebräuchlich sind hier sechs Qualitätsmerkmale, die in der Softwarebeurteilung zum Tragen kommen:

1. Funktionalität
2. Zuverlässigkeit
3. Benutzbarkeit
4. Effizienz
5. Änderbarkeit
6. Übertragbarkeit

Für jedes dieser Merkmale existieren unterschiedliche Maßstäbe und Verfahren, um eine quantitative Erfüllung zu überprüfen. Jede Abweichung vom gewünschten Ergebnis ist ein *Fehler*. Ideal wäre es, Fehler zu vermeiden, bevor sie entstehen. Prävention ist eine Strategie, die sich auszahlt: Fehler entstehen an den verschiedensten Stellen bei der Softwareentwicklung. Es beginnt bei der Anforderungsdefinition, die schon fehlerhaft sein kann, über Fehler bei Spezifikation und Entwurf bis hin zu Fehlern im Programm, das sich dann nicht wie gewünscht verhält. Am wichtigsten ist es also, mit dem Kunden, dem Softwarenutzer, zu reden und die Anforderungen möglichst genau und widerspruchsfrei zu erfassen. Aber wie findet man nun die Fehler, die man vermeiden möchte?

Dazu gibt es verschiedene Strategien. Zum einen kann man durch Vorgaben, wie z. B. Programmierrichtlinien, Checklisten und den Einsatz von entsprechenden Werkzeugen, Programmiersprachen und Methoden im Vorfeld den Rahmen für den einzelnen Softwareentwickler festlegen (konstruktives Qualitätsmanagement). Gewisse Tricks in der Programmierung werden untersagt, die Art und der Umfang der Kommentierung des Quellcodes vorgegeben und eine Programmiersprache gewählt, die passend zur zu lösenden Aufgabe ist und dem Ausbildungsstand der beteiligten Programmierer entsprechen.

Zum anderen prüft man mittels *Tests*, ob das Geschaffene mit dem übereinstimmt, was vorgegeben war (*analytisches Qualitätsmanagement)*. Was vorgegeben ist, richtet sich nach der Softwareentwicklungsphase, in der man sich gerade befindet. Wie getestet werden kann, richtet sich nach den Möglichkeiten und dem Aufwand, den man investieren möchte. Bleiben wir doch gleich beim Testen.

3.4.1 Testen von Software

In den vorangegangenen Abschnitten war vom Algorithmus die Rede, der die computerprogrammierbare Lösung einer Aufgabe darstellt. Aufgabe eines Tests wäre es zu zeigen, ob ein gefundener und fertig formulierter Algorithmus auch das leistet, was in der Aufgabenforderung formuliert wurde. Eine einfache Vorgehensweisen hierfür ist der Schreibtischtest.

Schreibtischtest

Dieser Test simuliert den Einsatz des Algorithmus „im Kopf". Anhand einer sinnvollen Auswahl von Eingabedaten wird händisch das richtige Funktionieren des Algorithmus Schritt für Schritt nachvollzogen. Richtig heißt hier, dass in jedem Schritt der Algorithmus das tun muss, was der Erfinder eigentlich gewollt hat, und natürlich auch die bekannte richtige Ausgabe liefert. Der Schreibtischtest ist heute obligatorisch für jeden Softwareentwickler. Jedes Stück neue Software wird ebenfalls in dieser Form getestet, denn jede Teilsoftware hat definierte Eingabedaten und muss in irgendeiner Form bekannte Ausgabedaten erzeugen. Viele Denkfehler werden dabei aufgedeckt und können aufwandsarm beseitigt werden. Aber: Es ist aber auf gar keinen Fall der einzige Test!

Stillschweigend wurde hier vorausgesetzt, dass sinnvolle Testfälle bekannt sind. Das heißt, für einen Satz Eingabedaten ist das Ergebnis, die Ausgabedaten, bekannt. Nur dann kann ein Test auf richtiges/falsches Funktionieren des Algorithmus und seiner Implementierung als Computerprogramm überhaupt durchgeführt werden. In der Regel können aufgrund der Komplexität nicht alle möglichen Variationen der Eingabedaten für Tests verwendet werden. Hat z. B. eine Programmfunktion zwei Eingangsvariablen, deren jeweiliger Wertebereich 2^{64} ist, so wären theoretisch 2^{128} unterschiedliche Möglichkeiten von Eingabedaten zu testen. Selbst wenn nur eine Picosekunde für jeden einzelnen Test veranschlagt wäre, so käme man auf knapp $4 * 10^{21}$ Tage Rechenzeit. Damit ist klar, dass nicht alle Fehler durch Tests aufgedeckt werden können. Deshalb ist es nicht verwunderlich, wenn Tests des Softwareentwicklers keine Fehler mehr zutage bringen, Tests des Softwareanwenders aber schon. Die sinnvolle Auswahl der Testfälle ist entscheidend. Dazu gibt es unterschiedliche Sichtweisen, was getestet werden soll. Dazu zwei Beispiele:

- **Black-Box-Test**
  Der Black-Box-Test ist ein sogenannter funktionaler Test. Er betrachtet das Softwaresystem (Algorithmus) als schwarzen Kasten, in den man nicht hineinschauen kann. Der Anwender einer Software ist ein klassischer Black-Box-Tester. Er ist nur an den versprochenen Funktionen interessiert, aber wie die Software intern funktioniert, ist ihm egal. Also werden im Black-Box-Test alle vereinbarten Funktionen durch systematische Benutzung mit sinnvollen Testfällen aus Anwendungssicht durchgespielt. Dies ist in der entsprechenden Spezifikation im Vorhinein niedergelegt. Der Black-Box-Test ist der klassische Abnahmetest. Der Kunde, der Software kauft, testet sie nach der Lieferung, oftmals im Beisein des Entwicklers, und wenn sie aus seiner Sicht funktioniert muß er sie bezahlen. Auch in der Entwickung selbst wird dieses Verfahren häufig eingesetzt. Jeder Algorithmus und jede Programmfunktion ist in der Regel schriftlich spezifiziert, d. h. es ist festgelegt, was sie unter welchen Umständen wie können muss. Somit testet der Entwickler sein Programm gegen die Spezifikation mit daraus abgeleiteten Testfällen.

  Problematisch beim Black-Box-Testverfahren ist die sogenannte *Testüberdeckung*. Die Interna des Programms werden nicht berücksichtigt. So kann es vorkommen, dass gewisse Programmteile, wie Schleifen oder Abfragen, bei solchen Tests nie oder immer mit *harmlosen* Daten durchlaufen werden. Sinnvolle Testfälle, die sich nicht aus der Verwendung, sondern einzig aus dem Aufbau und der Formulierung des Programms ergeben, werden beim Black-Box-Test nicht systematisch durchgeführt. Die Konsequenz: unentdeckte Fehler.

Um dies zu vermeiden, gibt es das Prinzip des wesentlich aufwendigeren White-Box-Tests.

- **White-Box-Test**
  Ausgehend vom Programmquelltext werden im White-Box-Test der Kontrollfluss des Programms untersucht und Testfälle daraus abgeleitet. Der Kontrollfluss ist der Weg durch das Programm, den dessen Ausführung durch den Rechner nimmt. Alle möglichen Wege mit allen möglichen Datenkombinationen zu testen, ist in der Regel zu aufwendig.

  Eine Variante des White-Box-Tests sieht vor, dass die Testfälle so generiert werden, dass zumindest alle Programmanweisungen mindestens einmal durchlaufen werden. Dieser wird *Anweisungsüberdeckungstest* (C0-Test) genannt. Durch Verzweigungen im Programm kann eine Anweisung auf mehreren Wegen erreicht werden. Der Anweisungsüberdeckungstest nimmt hier keine Rücksicht. Hauptsache, jede Anweisung wurde einmal durchlaufen.

  Beim sogenannten *Zweigüberdeckungstest* (C1-Test), werden Testfälle so konstruiert, dass alle Zweige durchlaufen werden. Dieser ist aufwendiger als der Anweisungsüberdeckungstest. Aber selbst dieser reicht nicht aus, um alle Fälle abzudecken. Schleifen und datenabhängige Verzweigungen führen zu unterschiedlichen Wegen durch das Programm, diese werden *Pfade* genannt. Die Kombination der beschrittenen Zweige hat unterschiedliche Auswirkungen und kann somit Fehlerquelle sein. Dieses berücksichtigen die sehr aufwendigen *Pfadüberdeckungstests.* Selbst hier ist noch nicht alles getan. Die datenabhängigen Verzweigungen haben meist komplexe Entscheidungsbedingungen, beispielsweise

```
if (A && B || C ) { ... } else { ... }
```

  Das Ergebnis der Bedingung `A && B || C` hängt von den Ergebnissen der Bewertung der Teilausdrücke `A`, `B`, `C` ab. Eigentlich müssten für alle möglichen Wertekombinationen dieser Teilausdrücke die Verzweigung getestet werden. Im Pfadüberdeckungstest wird hierauf keine Rücksicht genommen. Erst der sogenannte *Bedingungsüberdeckungstest* kümmert sich darum.

  Betrachtet man nun noch die möglichen Daten, die ein Programm erzeugt und die es eingespeist bekommt, so finden noch die sogenannten *datenflussorientierten Testverfahren* Anwendung. Statt nur den Kontrollfluss zu betrachten, wird hier der Datenfluss mit den möglichen Werten und Wertebereichen der einzelnen Variablen im Programm untersucht.

In der Praxis wird man sich für einen sinnvollen realisierbaren Mix aus den vorgestellten White-Box-Testverfahren und dem Black-Box-Testverfahren entscheiden. Bei der Auswahl der Testdaten kann man sich von folgenden Gesichtspunkten leiten lassen:

- Wählen Sie die Testdaten so, dass das Programm für alle Aufgaben, die in der Spezifikation gefordert werden, getestet wird,
- Wählen Sie die Testdaten möglichst so, dass jeder Zweig des Programms mindestens einmal durchlaufen wird,
- Wählen Sie spezielle Testdaten derart, dass alle Sonderfälle des Programms erfasst werden,
- Wählen Sie zufällige Testdaten, auch wenn sie nicht sinnvoll erscheinen, um Testlücken aufgrund scheinbarer Selbstverständlichkeiten zu vermeiden.

3.4.2 Verifikation von Software

Testen von Software führt normalerweise nicht zu fehlerfreier Software, da nicht alles getestet wird. Das Gebiet der Verifikation von Software nimmt für sich in Anspruch, den quasi allgemeinen mathematischen *Beweis* anzutreten, dass *ein* gegebenes Programm korrekt (fehlerfrei) ist und für *alle* zu bearbeitenden Eingabedaten fehlerfrei funktioniert. Damit braucht man nicht mehr zu testen. Die Aufwand liegt darin, zu definieren, was korrekt ist. Man formuliert dabei für jede Anweisung des Programms *Zusicherungen*, die vor Ausführung der Anweisung gelten, und solche, die nach der Ausführung gelten.

Der Aufwand, den Verifikationsweg zu beschreiten, ist enorm und wird nur in ausgesuchten Situationen beschritten. Es gibt abgewandelte Verfahren wie z. B. das *symbolische Testen*, die öfter in der Praxis eingesetzt werden. Das Finden von Fehlern mittels einer Menge von Testfällen ist das bisher am häufigsten eingesetzte Verfahren.

3.4.3 Testplanung und Testdokumentation

In der Praxis wird zwar getestet, aber oftmals ziel- und planlos. Sollte doch getestet worden sein, so meistens ohne Protokoll. Treten später Fehler auf, so kann der Entwickler nicht lernen, warum der Fehler bis zum Auftreten unentdeckt blieb. Deshalb ist es wichtig, einen Testplan zu erstellen, in dem – im einfachsten Fall – alle durchzuführenden Tests festgehalten werden. Jede Einzeltestbeschreibung nennt hier den Testgrund, eine Durchführungsanweisung, die Eingangsdatenbelegung und das zu erwartende Ergebnis im fehlerfreien Fall. Das zu führende Testprotokoll verzeichnet neben Verweis auf Testfall im Testplan, Zeitangaben sowie Angabe der Testperson das Ergebnis des Tests.

Damit ist es möglich, nachzuvollziehen, was getan und getestet wurde. Idealerweise setzt man heute Testwerkzeuge ein, die es gestatten, Tests aufzuzeichnen und beliebig oft automatisch zu wiederholen. Diese Feature ist dann besonders gefragt, wenn die Software erweitert wird und damit wieder vollständig getestet werden muss. Manuell wäre das bei großen Softwaresystemen sehr aufwendig.

Ist beim Testen ein Fehler erkannt worden, geht es an seine Lokalisierung und sodann an die Therapie. Typische Fehler sind:

- Schleifen, die entweder einmal zu oft oder einmal zu wenig durchlaufen werden,
- nicht initialisierte Variablen,
- Sprung in das Innere von Schleifen,
- Indexüberschreitung bei Tabellen,
- unerwartete Nebenwirkungen von Prozeduren und Funktionen,
- mehrfache Benutzung der gleichen Variablen,
- numerische Überraschungen (z. B. Rundungsfehler oder Auslöschung führender Ziffern bei der Subtraktion benachbarter Zahlen).

Die Überprüfung eines Programms auf Erfüllung seiner Spezifikation kann also nach zwei verschiedenen Methoden erfolgen:

1. Formale Methode (Korrektheitsbeweis). Mittels logischer Herleitungen wird die Einhaltung der Bedingungen an die Zwischen- und Endergebnisse des Programms nachgewiesen.
2. Empirische Methode (Testen). Mit bestimmten ausgesuchten Daten, für die das Ergebnis bekannt ist, wird das Programm ausprobiert. Dabei kann nur das Vorhandensein von Fehlern, nicht aber die Fehlerfreiheit nachgewiesen werden.

3.5 Datenstrukturen

In der kommerziellen Datenverarbeitung treten sehr oft Problemstellungen auf, bei denen es weniger auf die Algorithmen ankommt (weil diese beispielsweise sehr einfach sind), sondern auf die Umformung der gegebenen Daten. So gibt es zum Beispiel Listengeneratoren, also Programme, die nichts anderes tun, als bereits bekannte Daten in eine übersichtliche Form zu bringen (was ja u. a. ein Grundgedanke bei der Entwicklung von Perl war). Für die Aufgabenstellung ist der Algorithmus selbst von geringer Bedeutung. Hier kommt es in erster Linie auf die Ein- und Ausgabedaten sowie deren Umformung an. So hat sich neben der problemorientierten Programmentwicklung die datenorientierte Programmentwicklung herausgebildet. Hier wird zuerst die Datenstruktur für jede zu verarbeitende Datenmenge definiert. Danach stellt man fest, welche Arten der Zuordnung existieren. Dabei sollen

- Sequenzen (der Datentyp A besteht aus B, gefolgt von C, gefolgt von D),
- Wiederholungen (der Datentyp E besteht aus einem oder mehreren Datentypen F) und
- Auswahlen (der Datentyp G besteht entweder aus dem Typ H oder J)

unterschieden werden. Dann werden alle Einzeloperationen in ungeordneter Reihenfolge aufnotiert und anschließend jede Operation den Datenstruktur-Diagrammen und den daraus entwickelten Programmstrukturen zugeordnet. Dieses Entwurfsprinzip ist aber ausschließlich für **umformende Problemstellungen** geeignet.

Wichtig für die allgemeinen Probleme ist, dass die Eingabe- und Ausgabedaten eines Programms genau analysiert und strukturiert werden. Das führt meist auch zu kürzeren und übersichtlicheren Programmen. Bei der Datenanalyse können Sie folgendermaßen vorgehen:

- Welche Daten sollen ausgegeben werden?
- Welche Daten werden zur Berechnung benötigt? (Je weniger das sind, desto besser! Das Prinzip „Am besten nehme ich alle erreichbaren Daten und suche mir nachher die benötigten heraus“ führt nämlich in den meisten Fällen zu einem Datenmüll, der nicht mehr zu bewältigen ist.)
- Welche Werte können die Daten annehmen?
- Welche der so gefundenen Daten (die sogenannten Datenelemente) gehören zusammen?
- Welche Struktur haben zusammengehörende Daten:
  - Ist die Anzahl fest oder variabel?
  - Sind stets alle Daten relevant?
  - Bilden verschiedene Datenelemente einen neuen Datentyp?

Es hat sich auch beim Entwurf von Programmiersprachen schnell herausgestellt, dass die zu verarbeitenden Daten nach ihrer Art, ihrem Typ, unterschieden werden sollten. In strukturierten Sprachen ist es üblich, alle verwendeten Variablen am Anfang eines Programmes summarisch aufzuführen (Datendefinition bzw. -deklaration). Das trägt nicht nur zu Verständlichkeit und Lesbarkeit wesentlich bei, sondern kann manchmal sogar Programmierfehler schon bei der Übersetzung des Quellcodes ans Licht bringen. Insbesondere gehört zu einer guten Dokumentation die explizite Angabe des Wertebereiches einer jeden Variablen.[1] Die wichtigsten Gründe dafür sind die folgenden:

[1] Perl hat leider nur wenige Datentypen und kommt zu Not auch ohne Deklaration aus, was zum schlechten Ruf der Sprache beigetragen hat.

1. Die Kenntnis des Wertebereiches einer Variablen ist entscheidend für das Verständnis eines Algorithmus. Ohne explizite Angabe des Wertebereiches ist es meistens schwierig festzustellen, welche Art von Objekten eine Variable repräsentiert, und die Ermittlung möglicher Fehler wird erheblich erschwert.

2. Zweckmäßigkeit und Korrektheit eines Programms sind in den meisten Fällen abhängig von den Anfangswerten der Argumente, und sie sind nur für bestimmte Bereiche garantiert. Daher gehört es zur Dokumentation eines Algorithmus, dass diese Bereiche explizit aufgeführt sind.

3. Der Speicherbedarf für die Repräsentation einer Variablen im Speicher eines Computers hängt von deren Wertebereich ab. Damit ein Compiler die nötigen Speicherzuordnungen vornehmen kann, ist die Kenntnis der Wertebereiche unerlässlich.

4. Operatoren, die in Ausdrücken vorkommen, sind nur für gewisse Wertebereiche ihrer Argumente wohldefiniert. Ein Compiler kann aufgrund der angegebenen Wertebereiche prüfen, ob die vorkommenden Kombinationen von Operatoren und Operanden zulässig sind. Die Angabe der Wertebereiche der Variablen dient also als **Redundanz** zur Überprüfung des Programms während seiner Übersetzung.

5. Die Realisierung von Operatoren in Rechenanlagen hängt unter Umständen von den Wertebereichen der Operanden ab. Ihre Kenntnis ist daher zu einer zweckmäßigen und effizienten Realisierung oft unerlässlich. So werden z. B. in den meisten Rechenanlagen verschiedene interne Darstellungsarten für ganze und für reelle Zahlen verwendet, und die detaillierten Abläufe von arithmetischen Operationen unterscheiden sich stark für die beiden Darstellungsweisen.

6. Eine automatische Prüfung von Grenzverletzungen durch das Laufzeitsystem während das Programm abläuft ist möglich → sichere Programme (auf Kosten der Geschwindigkeit).

Bedenken Sie immer, dass ein Programm zwar mehrfach geändert oder erweitert, jedoch wesentlich häufiger gelesen wird. Dann sollte der Leser – schlimmstenfalls ist man das selbst – auch nach Jahren schnell verstehen, worum es geht. Dazu eine Anekdote: Ich hatte gegen Ende meines Studiums ein spezielles Programm für einen Uni-Lehrstuhl geschrieben, in dem es um die Protokollierung von Messergebnissen ging. Nach über zwei Jahrzehnten erreichte mich im Januar des Jahre 2000 eine E-Mail: ... das Programm gibt jetzt bei der Jahreszahl „100“ statt „00“ aus ... Zum Glück gab es noch den Quellcode, der sogar noch vom aktuellen C-Compiler akzeptiert wurde und nach einer halben Stunde war das Jahr-2000-Problem dank halbwegs brauchbarer Dokumentation gelöst.

Merke:

- Schlechte Dokumentation ist besser als gar keine.
- Gute Dokumentation ist besser als schlechte.
- Manche Programme laufen länger, als man denkt.
- Der Kunde hat immer Recht (bis Du Dein Geld hast).

4

Datenstrukturen

The real danger is not that computers will begin to think like men,
but that men will begin to think like computers.

Sydney J. Harris

In diesem Kapitel werden grundlegende Eigenschaften von Datentypen und -strukturen behandelt. Dabei wird nur allgemein auf Datentypen sowie deren Bedeutung und Eigenschaften eingegangen, wobei aber teilweise schon die Notation der Sprache Perl verwendet wird. In einem späteren Kapitel werden dann die Besonderheiten der Datentypen in Perl ausführlicher behandelt.

4.1 Datentypen

Der Wertebereich einer Variablen spielt eine derart wichtige Rolle in der Charakterisierung einer Variablen, dass er als deren **Typ** bezeichnet wird. Die Werte eines Bereichs werden als Konstanten dieses Typs bezeichnet. Es wird daher die bindende Konvention eingeführt, dass in jedem Programm alle Variablen vereinbart werden. Dies geschieht durch Angabe einer Liste der gewählten Variablenbezeichnungen und bei vielen Programmiersprachen durch deren Typen. Eine Variablenvereinbarung in der Programmiersprache C hat beispielsweise die Form *TYP variable*, wobei *variable* eine Variablenbezeichnung und *TYP* der Typ der Variablen ist. So bezeichnet „int j“ eine Ganzzahlvariable namens „j“ oder „double Summe“ eine doppelt genaue Gleitpunktzahl „Summe“. Bei Perl gibt es eine wesentlich schwächere Typisierung der Variablen. Eine Perl-Variable kann unter anderem Integerwerte, Gleitpunktzahlen, Buchstaben und Zeichenketten aufnehmen. Auch müssen Variablen nicht vorher vereinbart werden, sondern der notwendige Speicherplatz wird beim ersten Auftauchen der Variablen belegt. Trotzdem sollte man auch bei Perl die benötigten Variablen und Konstanten vorab definieren.

Die Konvention, alle verwendeten Bezeichnungen am Anfang eines Programms zu vereinbaren, hat zudem den großen Vorteil, dass ein Compiler bei jeder im Programm vorkommenden Bezeichnung prüfen kann, ob sie überhaupt vereinbart ist. Im negativen Fall, der beispielsweise durch Schreibfehler verursacht wird, kann der Compiler durch eine entsprechende Fehlermeldung auf den Lapsus aufmerksam machen. Die zusätzliche Redundanz wird also wiederum zur Erhöhung der Programmiersicherheit ausgenutzt.

Wie werden nun Datentypen in einem Programm eingeführt, und welche Wertbereiche können durch Computer zweckmäßig dargestellt werden? Vorab sei bemerkt, dass es üblich ist, Datentypen in verschiedene Arten einzuteilen. Das wesentliche Merkmal eines Typs ist die Struktur seiner Werte. Ist ein Wert unstrukturiert, also nicht in einzelne Komponenten zerlegbar, so wird er – und damit auch sein Typ – als **skalar** bezeichnet. Ist er dagegen in einzelne Komponenten zerlegbar, nennt man ihn **strukturiert**.

4.1.1 Konstante

Literale sind Datenwerte, die direkt in der Anweisungsfolge eines Programms eingetragen werden können (Standardbezeichnung, z. B. Zahlen). Als Konstante bezeichnet man üblicherweise Namen für Literale (frei wählbare Bezeichnung). Durch die Vereinbarung von Konstantennamen wird ein ganz bestimmter Datenwert mit einem Namen versehen und ist über diesen Namen jederzeit zugreifbar. Der Wert einer Konstanten kann normalerweise nicht geändert werden. Konstante dienen der Übersichtlichkeit und Lesbarkeit von Programmen, z. B. bei Perl durch:

```
use constant BUFFER_SIZE    => 2048;
use constant ONE_YEAR       => 365.2425 * 24 * 60 * 60;
use constant PI             => 3.141592653589793238462;
use constant DEBUGGING      => 0;
use constant EMAIL          => 'info@foo.de';
```

Bei der aktuellen Version von Perl hat dieses Konstrukt aber noch ein paar Haken und Ösen: So wird der Konstantenname nicht innerhalb von Zeichenketten aufgelöst, einige Namen sind nicht erlaubt usw. Deshalb treten bei Perl-Programmen anstelle der Konstanten oftmals vorbelegte Variablen – diese sind dann mit der entsprechenden Sorgfalt zu behandeln.

4.1.2 Variablen

Datenobjekte, die während der Ausführung des Algorithmus unterschiedliche Werte aus einer bestimmten Wertemenge (festgelegt im Datentyp) annehmen können, nennt man „Variable“. Sie sind an die Variablen, die man aus der Mathematik kennt, angelehnt. Jede Variable besitzt einen Namen und einen Wert. Sie stellt somit einen Behälter für ihren Wert dar. Der Compiler reserviert entsprechend dem Typ eine bestimmte Menge Speicherplatz für den Behälter. Die Variable kann als Benennung von einem oder mehreren Speicherworten aufgefasst werden. In Perl ist es sogar noch etwas komplizierter, denn die Datentypen der Variablen sind relativ verwaschen. Wenn man nur einmal skalare Variablen betrachtet, kann deren Inhalt sowohl eine Zeichenkette, eine Zahl, eine Referenz, ein Filehandle oder ein Wahrheitswert sein. Im Gegensatz zu anderen Sprachen wird der Datentyp nicht vorher festgelegt und kann sich jederzeit ändern. Ein skalarer Wert wird als WAHR interpretiert, wenn er kein Leerstring oder die Zahl 0 (oder deren Stringrepräsentation „0“) ist.

In vielen Programmiersprachen gibt es zwei verschiedene Arten von Variablenvereinbarungen, Definitionen und Deklarationen. Bei der **Definition** werden Variablen namentlich bekannt gemacht und auch Speicher für die Variable reserviert. Damit hat die Variable aber noch keinen definierten Wert. Bei etlichen Programmiersprachen wird, sofern nicht bei der Definition ein Wert zugewiesen wurde, einfach der (zufällige) Speicherinhalt der Variablen verwendet. Bei Perl erhält die Variable bis zur ersten Wertzuweisung den Wert „undef“.

Deklarationen legen nur die Art der Variablen bzw. die Schnittstelle der Funktionen fest, d. h. die Funktionsköpfe. Während Definitionen von Variablen und Funktionen dazu dienen, Datenobjekte bzw. Funktionen im Speicher anzulegen, machen Deklarationen Datenobjekte bzw. Funktionen bekannt, die in anderen Übersetzungseinheiten oder in derselben Übersetzungseinheit erst nach ihrer Verwendung definiert werden. Eine Deklaration

umfasst stets den Namen eines Objektes und seinen Typ. Damit weiß der Compiler, mit welchem Typ er einen Namen verbinden muss. Kurz ausgedrückt:

Definition und Deklaration: Definition = Deklaration + Reservierung des Speicherplatzes

Die **Zuweisung** eines Wertes an eine Variable ist eine fundamentale Operation in Computerprogrammen. Eine Variable zeigt eine Verhaltensweise, die einer Wandtafel ähnlich ist: Sie kann jederzeit gelesen (sie liefert den Wert), aber andererseits auch ausgewischt und überschrieben werden. In fast allen höheren Programmiersprachen müssen Variablen vor ihrer Verwendung deklariert werden, in Perl kann dies explizit (per Definition der Variablen) oder implizit (bei der ersten Verwendung der Variablen) geschehen. Die implizite Definition birgt natürlich die Gefahr, dass Tippfehler im Variablennamen zu einer impliziten Definition einer neuen Variablen und damit zu schwer auffindbaren Programmfehlern führen. Daher ist die explizite Definition (mit Vorbesetzung der Variablen) dringend zu empfehlen.

Die Zuweisung eines Wertes an eine Variable wird im Allgemeinen durch das Operatorzeichen = bezeichnet (zur Unterscheidung vom Gleichheitszeichen verwenden manche Programmiersprachen auch einen Linkspfeil oder „:=“). Bild 4.1 versinnbildlicht diesen Vorgang.

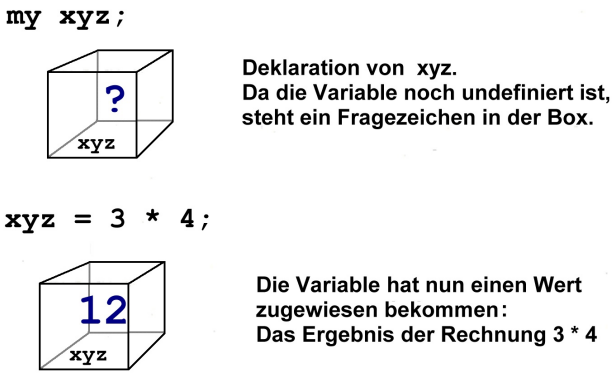

Bild 4.1: Variablendefinition und Wertzuweisung

Wichtig ist die Unterscheidung zwischen der Wertzuweisung im Programm und einer Gleichung im mathematischen Sinn. So würde die mathematische Gleichung

$$X = X + 1$$

mathematisch wenig Sinn machen (es gibt nämlich keine Lösung), in einer Programmiersprache bedeutet der Ausdruck jedoch „Addiere 1 zum Wert von X und speichere das Ergebnis wieder in X“ oder kürzer „Erhöhe X um 1“.

In Perl ist eine Wertzuweisung auch bei der Definition erlaubt. Es handelt sich um ein sehr sinnvolles Feature, denn eine Variable besitzt ja erst nach der Wertzuweisung einen definierten Wert. Nicht initialisierte Variablen (d. h. Variablen ohne Anfangswert) sind oft die Ursache von Programmfehlern. Dazu ein Beispiel (in Perl beginnen übrigens alle skalaren Variablen mit einem $-Zeichen):

```
my $X = 42;
my $name = "Zaphod Beeblebrox";
```

Merke:
Steht die Variable auf der rechten Seite einer Zuweisung, bezeichnet sie ihren Inhalt. Steht die Variable auf der linken Seite einer Zuweisung bezeichnet sie einen Behälter.

4.2 Ausdrücke

Ein Ausdruck (engl. expression) ist eine Formel (eine Rechenregel), die stets einen (Resultat-)Wert liefert. Der Ausdruck besteht aus Operanden (Konstanten, Variablen und Funktionen) und Operatoren. Operatoren werden üblicherweise eingeteilt in

- monadische Operatoren = einstellige Operatoren, z. B. Vorzeichen +/- vor Zahlen, NOT-Operator vor Wahrheitswerten
- dyadische Operatoren = zweistellige Operatoren, z. B. die Operatoren der Grundrechenarten (+ - * /)

Sofern Ausdrücke mit mehr als einem Operator vorkommen, ist die Reihenfolge ihrer Ausführung eindeutig festzulegen. Dies kann durch drei Möglichkeiten erfolgen:

- Reihenfolge der Aufschreibung von links nach rechts
- allgemeine Vorrangregeln (beim Rechnen „Punkt vor Strich“)
- Klammerung um die einzelnen Operanden

Die Hierarchie nimmt in der oben aufgeführten Liste von oben nach unten zu, d. h. Klammern binden stärker als Vorrangregeln und diese stärker als die Reihenfolge. Der Begriff „Ausdruck“ beschränkt sich dabei nicht auf arithmetische Ausdrücke (beispielsweise $a * y + b$), es gibt auch logische Ausdrücke oder solche, die Zeichen, Zeichenfolgen oder Speicheradressen zum Ergebnis haben.

4.2.1 Ausdrücke und Anweisungen

In Programmen sind Anweisungen und Ausdrücke nicht das Gleiche. Sie unterscheiden sich durch den Rückgabewert:

- Ausdrücke haben immer einen Rückgabewert. Sie können damit Teil eines größeren Ausdrucks sein. Was ist aber nun genau der Rückgabewert? Das soll anhand des Ausdrucks `13 + 16.5` erklärt werden. Durch die Anwendung des Additionsoperators **+** auf seine Operanden `13` und `16.5` ist der Rückgabewert des Ausdrucks (`29.5`) eindeutig festgelegt. Aus den Typen der Operanden ergibt sich auch der Typ des Rückgabewertes. Werden wie in diesem Beispiel unterschiedliche Datentypen in einem Ausdruck verwendet, führt der Compiler eine sogenannte implizite Typumwandlung nach vorgegebenen Regeln durch. Als Erstes prüft der Compiler die Typen der Operanden. Der eine Operand ist eine ganze Zahl vom Typ `int`, der andere eine Gleitpunktzahl vom Typ `float`. Damit ist eine Addition zunächst nicht möglich. Es muss zuerst vom Compiler eine für den Programmierer unsichtbare sogenannte implizite Typumwandlung der Zahl 13 in den Typ `float` (also zu 13.0) durchgeführt werden. Erst dann ist die Addition möglich. Der Rückgabewert der Addition ist die Zahl 29.5 vom Typ `float`. Jeder Rückgabewert hat also auch einen Typ.
- Anweisungen können keinen Rückgabewert haben. Sie können damit nicht Teil eines größeren Ausdrucks sein. In der Regel sind dies Programmbefehle. In Perl gibt es verschiedene Formen von Anweisungen:
  - Selektionsanweisungen,
  - Iterationsanweisungen,
  - Sprungananweisungen und

– Ausdrucksanweisungen.

Die ersten drei Anweisungen werden im nächsten Kapitel ausführlich behandelt, auf die letze Form wird hier eingegangen. In Perl kann man durch Anhängen eines Semikolons an einen Ausdruck erreichen, dass dieser zu einer Anweisung wird. Man spricht dann von einer sogenannten „Ausdrucksanweisung". In einer solchen Ausdrucksanweisung wird der Rückgabewert eines Ausdruckes nicht immer verwendet. Oft kommen auch nur Seiteneffekte zum Tragen. Dazu ein Beispiel:

```
my i = 0;            # Zuweisung eines Anfangswertes an die Variable

i++;                 # immer sinnvoll; i wird um 1 erhoeht

i + 1;               # zulaessig, aber nicht in allen Sprachen sinnvoll
                     # bei Perl beispielsweise Rückgabewert einer Funktion
```

4.2.2 L-Werte und R-Werte

Ausdrücke habe eine unterschiedliche Bedeutung, je nachdem, ob sie links oder rechts vom Zuweisungsoperator stehen. Wie schon erwähnt, steht beim Ausdruck `a = b` die rechte Seite für einen Wert, während der Ausdruck auf der linken Seite die Stelle angibt, an der der Wert zu speichern ist. Wenn wir das Beispiel noch etwas modifizieren, wird der Unterschied noch deutlicher:

```
a = a + 5*c
```

Beim Namen `a` ist rechts vom Zuweisungsoperator der Wert gemeint, der in der Speicherzelle `a` gespeichert ist, und links ist die Adresse der Speicherzelle `a` gemeint, in der der Wert des Gesamtausdrucks auf der rechten Seite gespeichert werden soll. Aus dieser Stellung links oder rechts des Zuweisungsoperators wurden auch die Begriffe **L-Wert** und **R-Wert** abgeleitet.

L-Wert:
Ein Ausdruck stellt einen **L-Wert** („lvalue" oder „left value") dar, wenn er sich auf ein Speicherobjekt bezieht. Ein solcher Ausdruck kann links und rechts des Zuweisungsoperators stehen.

Ein Ausdruck, der keinen L-Wert darstellt, stellt einen **R-Wert** („rvalue" oder „right value") dar. Er darf nur rechts des Zuweisungsoperators stehen. Einem R-Wert kann man also nichts zuweisen.

Es gilt demnach:

- Ein Ausdruck, der einen L-Wert darstellt, darf auch rechts vom Zuweisungsoperator stehen. In diesem Fall wird dessen Namen bzw. Adresse benötigt, um an der entsprechenden Speicherstelle den Wert der Variablen abzuholen. Dieser Wert wird dann zugewiesen.
- Links des Zuweisungsoperators muss immer ein L-Wert stehen, da man den Namen bzw. die Adresse einer Variablen braucht, um an der entsprechenden Speicherstelle den zugewiesenen Wert abzulegen.
- Weiter wird zwischen modifizierbarem und nicht modifizierbarem L-Wert unterschieden. Ein nicht modifizierbarer L-Wert ist z. B. eine Konstante oder der Name eines Arrays. Dem Namen entspricht zwar eine Adresse. Diese ist jedoch konstant und kann nicht modifiziert werden. Auf der linken Seite einer Zuweisung darf also nur ein modifizierbarer L-Wert stehen.

- Bestimmte Operatoren können nur auf L-Werte angewandt werden. So kann man den Inkrementoperator **++** oder den Adressoperator & nur auf L-Werte anwenden. `5++` ist falsch, `i++` ist korrekt (sofern i eine Variable ist).

4.2.3 Globale vs. lokale Variablen

Es gibt bei Perl und vielen anderen Programmiersprachen die Möglichkeit, innerhalb von Unterprogrammen **lokale** Konstante und Variablen zu definieren, die dann nur innerhalb des jeweiligen Unterprogramms existieren und nur dort verwendbar sind. Das hat u. a. den Vorteil, dass man in der Wahl der Namen für Konstante und Variablen relativ frei ist, denn Variablen gleichen Namens in verschiedenen Unterprogrammen stören sich nicht gegenseitig. Will man nun Werte in mehreren Unterprogrammen verwenden, böte sich an, die entsprechenden Variablen **global** zu definieren, um den Zugriff von allen Programmteilen aus zu erlauben. Das hat aber einige negative Konsequenzen, zum Beispiel:

- Da alle Unterprogramme auf die globalen Datenelemente lesend und schreibend zugreifen können, ist es oft nur schwer nachzuvollziehen, wann und von wem deren Inhalt verwendet oder verändert wurde. Deshalb sind Algorithmen mit globalen Datenelementen oft schwer zu verstehen und nur aufwendig zu testen.
- Wenn die globalen Datenelemente in ihrer Struktur verändert werden, (weil sich beispielsweise herausstellt, dass statt eines Arrays ein Hash besser geeignet wäre), sind Änderungen an all jenen Algorithmen erforderlich, die auf dieses globale Datenelement zugreifen.
- Hängen mehrere globale Datenelemente logisch zusammen (z. B. ein Feld A und die aktuelle Anzahl seiner Elemente), müssen diese Datenelemente konsistent gehalten werden (z. B. muss bei Eintrag eines neuen Werts in das Feld A auch n um 1 erhöht werden).

Aus diesem Grund wird empfohlen, auf die Verwendung globaler Datenelemente zu verzichten und stattdessen diese in „Datenkapseln“ (*data capsules*) zu verpacken, um sie so vor ihrer Umgebung (den Algorithmen) geheim zu halten. Dieses Konzept heißt daher „Geheimnisprinzip“ (*principle of information hiding*). Den Algorithmen ist der direkte Zugriff auf die gekapselten Daten verwehrt, auf sie kann nur mehr indirekt über die „Zugriffsoperationen“ zugegriffen werden. Nur diese kennen den genauen Aufbau der gekapselten Datenobjekte. Eine missbräuchliche Verwendung ist damit ausgeschlossen, und Änderungen ihrer Struktur schlagen sich nicht in den Algorithmen, sondern nur in den Zugriffsoperationen nieder. Letztendlich hat dies zu den objektorientierten Sprachkonzepten geführt, bei denen Datenelemente und Zugriffsalgorithmen zu einem Objekt gekapselt sind.

4.3 Skalare Standardtypen

Wie schon erwähnt, ist die Definition der Variablen und Konstanten u. a. wichtig für die Verständlichkeit des Programms. Insbesondere gehört zu einer guten Dokumentation die explizite Angabe des Wertebereichs. Nachfolgend werden vier häufig vorkommende Standardtypen vorgestellt. Trotzdem sind nicht alle Typen auch in jeder Programmiersprache verfügbar. So kennt Perl nur einen „Allerweltstyp“, C kennt neben ganzen Zahlen (int, long int) und Gleitpunktzahlen (float, double) auch einen Aufzählungstyp. Man kann die fehlenden Typen jedoch in fast allen höheren Programmiersprachen nachempfinden.

- **Boolean**
  Dieser Typ bezeichnet den Bereich der logischen Werte, bestehend aus zwei Elementen, WAHR (TRUE , 1, Wert ungleich 0) und FALSCH (FALSE, 0, Wert gleich 0). Auf

den Werten dieses Typs sind vier Operatoren definiert: OR (logisches Oder), AND (logisches Und), EXOR (logische Antivalenz), NOT (logisches Nicht). Die Gesetze für diese Operatoren entsprechen jenen der Schaltalgebra bzw. Aussagelogik. Außerdem sind Vergleichsoperatoren definiert (=, #, $<$, $>$, $\leq$, $\geq$), die beim Vergleich von skalaren Operatoren einen Wert vom Typ Boolean liefern. In Perl nicht vorhanden.

- **Integer**
  Dieser Typ bezeichnet den Bereich der ganzen Zahlen. Es sind die folgenden Operatoren definiert:
  - + Addition
  - - Subtraktion
  - * Multiplikation
  - / Ergebnis der ganzzahligen Division
  - % Rest der ganzzahligen Division (Modulo-Operator)

  Bei jedem Computer spezifiziert dieser Datentyp im streng mathematischen Sinn nur eine Untermenge der ganzen Zahlen, und zwar die Menge aller Zahlen, die betragsmäßig kleiner sind, als ein bestimmter Maximalwert. Dies bedeutet, dass die Axiome der Arithmetik nur innerhalb dieser Untermenge gelten. Normalerweise wird eine Länge gewählt, die einem Vielfachen der Wortlänge des Prozessors entspricht.

- **Character**
  Dieser Typ bezeichnet eine endliche, geordnete Menge von Zeichen (Characters). Dieser Typ stützt sich auf den im jeweiligen Computer verwendeten Code zur Darstellung von Zeichen (ASCII, ANSI etc.). Er ist also nur insofern Standard, als der zugrunde liegende Zeichensatz Standard ist. Mit diesem Typ werden auch zwei vom Zeichensatz abhängige Konversionsmöglichkeiten zwischen Character und Integer eingeführt. Um die verschiedenen nationalen Zeichensätze unter einen Hut zu bringen, wurde UNICODE entwickelt. Das hat zur Folge, dass Werte dieses Typs 8 oder 16 Bit lang sein können.

- **Real** bzw. **Float**
  Dieser Datentyp stellt nicht Variablen aus dem Wertebereich der reellen Zahlen, sondern Gleitpunktzahlen dar. Die Repräsentation dieser Werte hängt auch wieder von der Realisierung im jeweiligen Rechnersystem ab (Zahl der gültigen Stellen, Wertemaxima). Bei vielen Systemen wird dieser Datentyp auch durch Nachbildung per Software (Emulation) realisiert. Insbesondere sind die assoziativen und distributiven Gesetze nicht mehr gültig! Auf dem Datentyp Float/Real sind vier Operatoren definiert:
  - + Addition
  - - Subtraktion
  - * Multiplikation
  - / Division

  Eine explizite Konversion von Integer nach Float/Real existiert nicht. Diese Konversion wird meist implizit bei der Übersetzung des Ausdrucks vorgenommen (falls nicht, hilft manchmal eine Multiplikation des Integerwertes mit 1.0).

4.4 Strukturierte Datentypen

4.4.1 Reihung, Feld (Array), Liste

Variablen dieses Typs besitzen eine Menge von Komponenten gleichen Typs und haben folgende Eigenschaften:

- Jedes Element ist einzeln referenzierbar (benennbar). Zur Bezeichnung einer bestimmten Array-Komponente wird der Name des Arrays mit einen Zusatz, genannt Index, versehen, der die Komponente eindeutig bezeichnet. Als Indexwerte werden berechenbare Größen eines aufzählbaren, skalaren Typs verwendet (z. B. Integer, Character).
- Die Anzahl der Elemente wird bei der Vereinbarung festgelegt und bleibt danach bei vielen Programmiersprachen unveränderlich (Perl bildet eine Ausnahme). Bei der Vereinbarung eines Array-Typs wird sowohl der Komponententyp als auch der Indextyp spezifiziert. Es besteht eine eindeutige Zuordnung zwischen Indexwerten und Array-Komponenten. Der Indexwert wird in eckigen Klammern hinter den Namen des Arrays geschrieben (z. B. $xyz[6]$), wobei an der Indexposition meist ein beliebiger Ausdruck stehen kann (etwa $xyz[i * 10 + j]$). Der Index wird oft von 0 ab gezählt. Die Dimension gibt an, wie viele Komponenten das Feld haben soll. Die Indexmenge reicht also von 0 bis $(Dimension - 1)$.

Zwei Array-Variablen werden als gleich bezeichnet, wenn sie vom selben Typ sind (d. h. identische Vereinbarung besitzen) und alle Komponenten paarweise gleich sind (d. h. A = B, wenn A[i] = B[i] für alle i des Indexbereichs).

Als Komponententyp eines Array kann auch wieder ein Arraytyp verwendet werden → mehrdimensionale Arrays. Die Vereinbarung wird in den meisten Programmiersprachen zu einer verkürzten Vereinbarung zusammengezogen, z. B. `int A [100][50]`.

Bei Perl gibt es drei Besonderheiten beim Arraytyp: Zum einen sind die Arraygrenzen nicht festgelegt, ein Array kann also durch das Hinzufügen von Komponenten dynamisch wachsen. Zweitens kann der Typ der Komponenten beliebig wechseln (z. B. kann A[0] eine Zahl und A[1] eine Zeichenkette sein). Drittens kennt Perl noch den Typ des „assoziativen Arrays“ (auch „Hash“ genannt), bei dem der Indextyp beliebig sein kann (insbesondere auch ein String). Dieser Typ erlaubt eine sehr flexible Programmierung und erstaunliche Effekte bei der Bearbeitung von Datenlisten.

Daneben kennt Perl noch den Datentyp **Liste**, eine Reihung, die ohne Index auskommt. Um eine Verarbeitung von Listen mit Arrayoperationen zu erleichtern, haben Listen implizit einen Integer-Index, der bei 0 beginnt. Die Anwendung ist manchmal recht subtil, daher wird an verschiedenen Stellen im Buch darauf eingegangen. Von der Schreibweise her ist der Unterschied gut sichtbar:

```
xyz[i]       # i-tes Element eines Array namens 'xyz'

(xyz)[i]     # i-tes Element einer Liste namens 'xyz'
```

4.4.2 Verbund (Record, Structure)

Im Gegensatz zum Array besitzt ein Verbund Komponenten unterschiedlichen Typs und stellt damit die flexibelste Datenstruktur dar. Ein Verbund besteht aus einer festen Anzahl von Komponenten, wobei jede Komponente mit einem eigenen Namen bezeichnet wird. In der Programmiersprache C hat ein Verbund beispielsweise folgendes Aussehen:

```
struct Kundensatz
   {
   int Kundennummer;
   char Name[20];
   char Vorname [20];
   float Umsatz;
   } Kunde;
```

Als Verbundkomponenten können wieder beliebige Datentypen stehen, also auch wieder Array- oder Verbundtypen. Umgekehrt kann auch ein Array Verbunde als Komponenten besitzen, etwa ein Array mit Komponenten vom oben gezeigten Kundentyp. Auf einzelne Komponenten des Verbunds wird über die Kombination des Verbundnamens und der gewünschten Verbundkomponente zugegriffen. Als Trennzeichen zwischen den Namen wird oft ein Punkt verwendet. Perl kennt derzeit (leider) keinen Verbundtyp.

4.4.3 Hash-Tabelle

Die Hash-Tabelle bzw. Streuwerttabelle ist eine spezielle Indexstruktur, bei der die Speicherposition direkt berechnet werden kann. Beim Einsatz einer Hash-Tabelle zur Suche in Datenmengen spricht man auch vom Hash-Verfahren. Bei sehr großen Datenmengen kann eine verteilte Hashtabelle zum Einsatz kommen. In Perl werden Arrays, deren Index nicht aus Skalaren, sondern aus beliebigen Werten (meist Zeichenketten) besteht, als „Hash“ bzw. „assoziatives Array“ bezeichnet, da die interne Speicherung sich an Hash-Tabellen anlehnt. Die Hashes von Perl stellen ein mächtiges Werkzeug bei der Datenverwaltung dar.

4.4.4 Dateien

Alle bisher betrachteten Datentypen haben sich dadurch ausgezeichnet, dass der Wert einer Variablen durch ihren Namen referenziert und die Zahl der Komponenten bei der Vereinbarung festgelegt wurde. Es stellt sich die Frage, welcher Datentyp verwendet werden soll, wenn die Zahl der Komponenten nicht von Anfang an festliegt (und keine dynamischen Arrays existieren). Ein zweiter Aspekt ist die Ein- und Ausgabe. Die bisher behandelten Variablen liegen immer im Arbeitsspeicher. Es muss also eine Möglichkeit zur permanenten Speicherung von Daten außerhalb des Hauptspeichers geboten werden – und zur Ein-/Ausgabe (E/A) über beliebige Geräte.

Dateien sind ein allgemeines Konzept für die permanente Speicherung bzw. Ein-/Ausgabe. Dateien bestehen aus beliebig vielen Komponenten eines bestimmten Datentyps. Je nach Dateiart können die einzelnen Komponenten sequenziell oder wahlfrei gelesen werden. Am Ende einer Datei können weitere Komponenten hinzugefügt werden. Die Abbildung der abstrakten Dateistruktur auf reale Speichermedien oder E/A-Geräte (Platte, Band, Drucker etc.) erfolgt durch das Betriebssystem.

Eine Datei besitzt also lauter Komponenten gleichen Typs. Zusammen mit der Dateivariable wird implizit auch ein Pufferbereich im Arbeitsspeicher definiert, der mindestens eine Dateikomponente (→ aktueller Wert) aufnehmen kann. Auf diesen Datenwert wird dann über die Dateivariable (Filehandle) zugegriffen. Zusammen mit dem Typ Datei müssen auch einige Standardoperationen definiert sein, die den Zugriff auf die Datei erlauben (Bild 4.2):

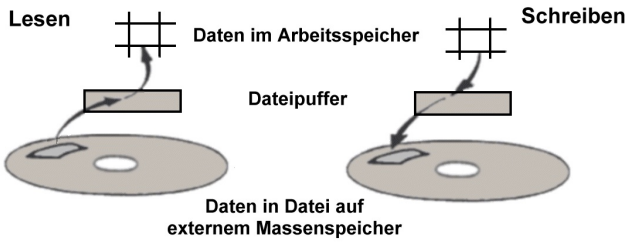

Bild 4.2: Datei-Ein-/Ausgabe

- **Open:** Öffnen einer Datei zur Inspektion des Inhalts (Lesen) oder zum Anfügen neuer Komponenten am Ende (Schreiben). Mit dieser Prozedur wird nicht nur der Zugriffsmodus festgelegt, sondern gegebenenfalls auch eine leere Datei generiert (beim Modus „Schreiben“ bei einer noch nicht vorhandenen Datei).
- **Close:** Beenden des Zugriffs auf eine Datei. Das Betriebssystem wird angewiesen, die Datei zu schließen. Damit verbunden sind Verwaltungsvorgänge des Betriebssystems, z. B. Wegschreiben des Inhalts interner Pufferbereiche.
- **Read, Write:** Lesen und Schreiben von Komponenten einer Datei. Die Arbeitsweise dieser Operationen hängt von der Art der Datei ab. Auf diese Arbeitsweise wird im folgenden Kapitel noch näher eingegangen.

Nach Art des Zugriffs auf die Komponenten wird unterschieden in sequenzielle Dateien und Dateien mit wahlfreiem Zugriff.

- Sequenzielle Dateien (*sequential files*)
  In den Anfängen der Computertechnik gab es nur sequenzielle Dateien, da das Lesen/Schreiben bei Magnetbändern und Lochstreifen nur streng sequenziell möglich war. Auch der Drucker wird über sequenzielle Dateivariablen angesprochen. Beim Öffnen einer Datei muss zwischen Lesen und Schreiben unterschieden werden:
  - **Sequenzielles Lesen:** Der Dateizeiger wird auf die erste Komponente positioniert. Nach dem Zugriff auf eine Komponente wird auf die nächstfolgende Komponente positioniert. Das Ende einer Datei wird durch eine spezielle Standardfunktion (EOF = End Of File) zurückgemeldet:
    * EOF = FALSE: Dateiende noch nicht erreicht
    * EOF = TRUE: Dateiende erreicht

    Von einer Datei darf also nur gelesen werden, wenn EOF = FALSE ist.
  - **Sequenzielles Schreiben:** Der Dateizeiger wird hinter der letzten Komponente positioniert; durch das Schreiben wird die Datei um eine Komponente erweitert. Danach wird wieder hinter der Datei positioniert.

  Um eine Datei mehrfach lesen zu können, gibt es bei vielen Systemen eine Standardoperation zum Zurücksetzen auf den Dateianfang (Rewind, Reset).
- Dateien mit wahlfreiem Zugriff (*random files*)
  Bei Magnetplatten und anderen Massenspeichern kann wahlfrei auf jeden Datenblock der Platte zugegriffen werden. Mit der Verbreitung solcher Speicher lag es nahe, den Datentyp Datei um diese Möglichkeit zu erweitern. Bei sequenziellen Dateien wird die Positionierung auf eine Komponente implizit vorgenommen und unterliegt nicht dem Einfluss des Programms. Bei Dateien mit wahlfreiem Zugriff werden die Lese- und Schreiboperationen um die Angabe eines Komponentenindex ergänzt. Der Komponentenindex wird dabei in der Regel von 1 ab aufwärts gezählt. Damit besitzt die Datei mit wahlfreiem Zugriff eine starke Ähnlichkeit mit dem Datentyp Array. Der Versuch, eine Komponente zu lesen, die *hinter* dem Dateiende liegt, führt zu einem Fehler. Beim Versuch, eine Komponente zu schreiben, deren Index I größer als die Anzahl N der aktuell vorhandenen Komponenten ist, können zwei Fälle unterschieden werden:
  - I = N + 1: Die Datei wird um eine Komponente erweitert (wie bei der sequenziellen Datei).
  - I $>$ N + 1: Bei einigen Systemen führt der Versuch zu einem Fehler. Bei anderen Implementierungen wird der Zwischenraum mit leeren Komponenten aufgefüllt (gefährlich, da die Datei plötzlich sehr groß werden kann).

Dateien, deren Komponenten Schriftzeichen sind (Typ Character), nehmen eine Schlüsselrolle ein, da die Eingabe- und Ausgabedaten der meisten Computerprogramme Textfiles sind (darunter fällt beispielsweise auch die Druckausgabe). Prinzipiell sind es natürlich auch ganz normale Dateien, die aus einer Folge von Bits oder Bytes bestehen, nur wird ihr Inhalt anders interpretiert. Ein Programm kann vielfach allgemein als eine Datentransformation von einer Textdatei in eine andere aufgefasst werden.

Nun sind Texte normalerweise in Zeilen unterteilt, und es stellt sich die Frage, wie diese Zeilenstruktur auszudrücken ist. In der Regel enthält der in einem System verwendete Zeichencode spezielle Steuerzeichen, von denen eines als Zeilenendezeichen verwendet werden kann. Bedauerlicherweise verhindert die Realisierung von Textdateien auf Betriebssystemebene eine einfache Realisierung der Textdatei als „file of character“.

- Bei einigen Betriebssystemen wird ein einziges Steuerzeichen als Zeilenende interpretiert (z. B. bei UNIX: Linefeed, beim Apple Mac: Carriage Return). Das verwendete Zeichen ist jedoch nicht einheitlich festgelegt.
- Bei anderen Systemen besteht das Zeilenende aus zwei Zeichen (in der Regel Carriage Return und Linefeed, etwa bei Windows).

Datenbanken kennen noch weitere Strukturierungen von Dateien, einerseits, um bestimmte Datensätze schnell auffinden zu können, andererseits, um über Indexdateien die Suche nach Inhalten zu ermöglichen.

Mit dem Dateityp „Datei“ bin ich auch am Ende dieses Kapitels angelangt. Im Folgenden werden die hier noch recht allgemein geschilderten Erkenntnisse auf die Programmiersprache Perl angewendet.