de Gruyter Lehrbuch
Schnupp/Floyd · Software

Peter Schnupp · Christiane Floyd

Software

Programmentwicklung und Projektorganisation

2., durchgesehene Auflage

Walter de Gruyter · Berlin · New York 1979

Dr. rer. nat. *Peter Schnupp,*
Mitinhaber der Firma SOFTLAB (Softwarelabor für Systementwicklung und EDV-Anwendung) in München, Lehrbeauftragter für Systemprogrammierung an der Hochschule für Sozial- und Wirtschaftswissenschften in Linz

Dr. phil. *Christiane Floyd,*
Prof. für Softwaretechnik an der Technischen Universität Berlin

Das Buch enthält 74 Abbildungen und 6 Tabellen.

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Schnupp, Peter

Software: Programmentwicklung u. Projektorganisation/
Peter Schnupp; Christiane Floyd. – 2., durchges. Aufl. – Berlin,
New York: de Gruyter, 1978.

(de-Gruyter-Lehrbuch)
ISBN 3-11-007865-1

NE: Floyd, Christiane.

Satz: IBM-Composer, Walter de Gruyter, Berlin. Druck: Karl Gerike, Berlin.
Bindearbeiten: Lüderitz & Bauer Buchgewerbe GmbH, Berlin.

Vorwort

Dieses Buch entstand aus einer Vorlesung über den Entwurf von EDV-Systemen, sowie aus einer Seminarreihe über *Moderne Softwaretechnologie.* Dies war auch der Arbeitstitel während der Manuskripterstellung: es soll in die Methoden einführen, welche heute von Entwicklungszentren, wie EDV-Herstellern und Softwarehäusern, zur rationellen Herstellung großer Softwaresysteme eingesetzt werden. Ebenso wie die Computer müssen auch die auf ihnen ablaufenden Programme hochwertige Industrieprodukte sein. Diese aber können nur von Technikern und Ingenieuren entwickelt werden, geschulten Fachleuten, welche die Fortschritte der wissenschaftlichen Forschung in die tägliche Praxis zu übertragen verstehen.

Die Grundlage hierfür bietet die Informatik. Die aus ihren Ergebnissen abgeleiteten Regeln zur systematischen Programmentwicklung werden oft als *Strukturierte Programmierung* propagiert. Auch dies wäre ein möglicher Titel für dieses Buch gewesen.

Das Ziel der vorliegenden Darstellung ist eine Einführung in die „höhere Programmierung". Dies einerseits als Grundlage einer entsprechenden Vorlesung für Hochschul- und Fachschul-Studenten der Informatik und verwandter Fächer, andererseits aber auch für den in der praktischen Systementwicklung erfahrenen Programmierer, Systemplaner oder Projektleiter, welcher sich im Selbststudium über die neueren Entwicklungen in seinem Fachgebiet unterrichten will.

Das jedem EDV-Autor geläufige Problem, sein Buch an *eine* Programmiersprache wie COBOL, PL/1, FORTRAN, ALGOL oder gar einen existierenden oder frei erfundenen Assembler binden zu müssen, wurde dadurch umgangen, daß neue Konzepte jeweils zugleich mit ihrer Repräsentation in „einer" Programmiersprache eingeführt werden. Es wird eine Sprache verwendet, die für jedes logische Strukturelement eines Programms genau eine, zumindest für ALGOL- oder PL/1-Programmierer leicht verständliche Darstellung bietet.

> Daß diese Sprache PASCAL heißt, wird dem Leser zwar verraten, braucht ihn aber nur dann zu interessieren, wenn er sich mit ihr näher beschäftigen will.

Wie die einzelnen Strukturelemente in anderen Sprachen wiederzugeben sind, wird einem nicht ganz unerfahrenen Praktiker meist unmittelbar einsichtig sein. Für die häufigsten Programmiersprachen COBOL, FORTRAN und PL/1 werden im Text und in den Anhängen Hinweise gegeben.

Wenn dieses Buch wirklich sein Ziel erreicht, Methoden und Techniken der modernen Softwareplanung und Programmierung darzustellen, so ist dies vor allem auch ein Verdienst unserer Kollegen in SOFTLAB, welche die aus ihrer täglichen Entwicklungspraxis gewonnenen Erfahrungen und Kritiken beisteuerten, sowie von Frl. *Helga Wittmann*, die wegen der dadurch bedingten laufenden Änderungen und Verbesserungen das Manuskript mit unendlicher Geduld und Sorgfalt sicher zweimal geschrieben hat.

Christiane Floyd
Peter Schnupp

Inhalt

„Aufregung ist kein Programm."

Masaryk

„. . . als Projektleiter bereite ich die Probleme auf, bis ich sie den Programmierern überlassen kann . . ."

Aus einer Bewerbung

Einführung

Gute Programme sind Mangelware. Es gibt sie, aber sie fallen nicht auf. Den Alltag des EDV-Benutzers, sei er nun Rechenzentrumsleiter, Teilnehmer an einem Timesharingdienst, oder auch nur ein in die zivilisierte Gesellschaft eingebetteter Bürger, der täglich computergeschriebene Rechnungen, Mitteilungen, Mahnungen, Bescheide in seinem Briefkasten findet – diesen Alltag bestimmen viel stärker schlechte Programme und ihre Folgen. Zuweilen sind diese Folgen erheiternd, wie die mehrmalige Anmahnung eines offenstehenden Betrages von 0.00 DM. Meist jedoch sind sie lästig und teuer, wie die bis zu 25% der verbrauchten Rechenzeit ausmachenden Wiederholungsläufe falscher oder falsch bedienter Programme in kommerziellen Rechenzentren. Vielleicht sind sie gar katastrophal, wie der finanzielle Zusammenbruch einer amerikanischen Eisenbahngesellschaft auf Grund eines Fehlers in dem zur Überwachung des Waggonbestandes eingesetzten Datenbanksystem.

Die Ausbildung eines Programmierers darf sich deshalb nicht auf das Erlernen einer oder mehrerer Programmiersprachen als Rohmaterial zur Montage von Programmen beschränken. Ihr Ziel muß die Mitteilung und Einübung von Techniken zur Produktion *guter* Programme sein. Diese Verfahren werden oft unter dem Schlagwort *moderne Softwaretechnologie* zusammengefaßt.

> Die moderne Softwaretechnologie bereitete übrigens einem ganzen Berufsbild ein jähes Ende: dem Programmierer als „Kodierer", der unverstandene Flußdiagramme weitgehend mechanisch in unverständliche Programme umsetzte. Deshalb und im Sinne von Dijkstra [DIJK72] soll in diesem Buch unter *Programmierer* jeder verstanden werden, der an der Konzeption, Planung und Realisierung eines Programms aktiven Anteil nimmt – auch der Systemplaner und Projektleiter, der *Chefprogrammierer* in der modernen Projektorganisation (vgl. Abschn. 2.6.2).

Was ist überhaupt ein „gutes" Programm? Wie erkennt man seine Qualität beim Studium des Programmtextes? Vor wenigen Jahren noch galten als einzige Kriterien für die Güte eines Programmes seine Laufzeit- und Speichereffektivität.

Deshalb dokumentierte ein Durchschnittsprogrammierer – von wenigen, schon damals skeptischen Ästheten abgesehen – sein Können durch ungewöhnliche Sprachkonstruktionen und Tricks, welche er in sein Programm hineinpackte. Wer etwa dynamisch zur Laufzeit Sprungbefehle generierte und wieder löschte, zeigte damit, daß er kein Anfänger mehr war, und als Meister suchte man sich dadurch auszuweisen, daß man die Bitmuster des Programmcodes gleichzeitig als Konstanten verwendete oder durch nicht allgemein bekannte Nebeneffekte eines Befehls einen anderen sparte.

Für den Benutzer jedoch, der ein Programm an seinen Folgen und nicht an seinem Code mißt, machen Tricks es nicht besser, sondern allenfalls schlechter: Tricks haben nun einmal die Eigenheit, zuweilen nicht zu funktionieren – im Gegensatz zum Zirkus leidet unter einem derartigen Versagen in der EDV nicht der Artist, sondern der unschuldige Konsument.

Deshalb konnte es nicht ausbleiben, daß sich mit dem wachsenden Unmut der EDV-Benutzer über die „Softwarekrise", die offenbar immer teurere Produktion und Wartung immer schlechterer Programme, ein benutzergerechterer Qualitätsmaßstab für Softwareprodukte durchsetzte. Heute gilt ein Programm als „gut", wenn es

- *benutzerfreundlich* ist, d. h. nach einfachen Bedienungsmaßnahmen das tut, was der Benutzer möchte und erwartet,
- *fehlerfrei* ist, d. h. seine Aufgabe nicht nur im Normalfall, sondern immer, auch bei ungewöhnlichen Datenkombinationen oder Bedienungsmaßnahmen, erfüllt,
- *wartbar* ist, d. h. auch von einem anderen Programmierer als seinem ursprünglichen Autor in endlicher Zeit verstanden und abgeändert werden kann (und dabei natürlich seine Fehlerfreiheit nicht verliert).

> Der Anteil der Wartung an den gesamten Softwarekosten wird meist unterschätzt. Eine britische Untersuchung in über 900 EDV-Installationen ergab einen durchschnittlichen Wartungsaufwand von 40% der gesamten Softwarekosten [ANON73a]. Mit zunehmender Komplexität der Programme verschiebt sich dieses Verhältnis noch drastisch: die US-Air Force ermittelte Entwicklungskosten von 75 $ pro Befehl, während bei der Wartung zuweilen 4000 $ pro Befehl erreicht wurden [TRAI73].

Die Planungs- und Programmierverfahren der Softwaretechnologie lassen die oben genannten benutzerorientierten Qualitätsforderungen bereits in den Programmentwurf eingehen. Sie erlauben es, die Güte des entstandenen Softwareprodukts schon an Hand des Programmtextes zu beurteilen und damit seine Benutzerfreundlichkeit, Fehlerfreiheit und Wartbarkeit nicht erst aus der Einsatz-

erfahrung zu verifizieren. Sie erzwingen zugleich den Verzicht auf alle liebgewordenen Kodiertricks.

Vielleicht fiel dem Leser auf, daß wir die gern zitierte Softwarekrise als eine *zunehmende* Verteuerung und Qualitätsverschlechterung der Programmproduktion charakterisierten. Woher kommt die darin implizierte negative zeitliche Entwicklungstendenz?

Sie geht parallel mit der wachsenden Größe der durchschnittlichen Programme. Das Betriebssystem einer Anlage der zweiten Generation hatte einige Tausend Anweisungen, eines der dritten Generation das zehn- bis hundertfache. Ähnlich ist das Verhältnis zwischen einem einfachen Assembler und einem Compiler für eine höhere, problemorientierte Sprache, oder das zwischen einem Personalstammband-Update und einem integrierten Personal-Informationssystem. Dieses Wachstum im Umfang der zu entwickelnden Programme führt nicht nur zu einer in der Regel überproportionalen Erhöhung des Entwicklungsaufwands und der Fehlerwahrscheinlichkeit. Es bedeutet auch, daß nicht mehr ein einziger Programmierer, sondern ein Team eingesetzt werden muß, um das gewünschte Produkt in vertretbarer Zeit zu erstellen.

Ein Team aber muß untereinander kommunizieren, es muß sich auf einheitliche Regeln, Verfahren und Schnittstellen festlegen, und es muß geleitet werden. Dies bringt neue Fehlerquellen und neuen, zusätzlichen Personalaufwand. Damit werden Projektorganisations- und -managementverfahren einschließlich der Spezifikations- und Dokumentationsmethodik zu wesentlichen Teilen der Softwareentwicklungstechnik.

Diese Verfahren sind aber noch schwerer lehrbar als Planungs- und Programmiertechniken. Saubere, fachlich einwandfreie Planung und Programmierung lassen sich an kleinen Übungsprogrammen vorführen und nachvollziehen – dies versuchen wir im ersten Teil des Buches. Die Schwierigkeiten des Projektmanagements und die zu ihrer Überwindung notwendigen Überlegungen und Verfahren können hingegen nur bei der tatsächlichen Realisierung eines größeren Projektes im Rahmen eines nicht nur aus *einem* Bearbeiter bestehenden Teams erlebt und erprobt werden. Das aber überschreitet meist die Möglichkeiten eines Programmierkurses und wohl immer die des Selbststudiums.

Andererseits – ein erfolgreiches Arbeiten im Team setzt ein Verständnis seiner Organisation und seines Managements voraus. Und das nicht nur bei der Projektleitung, sondern auch bei den übrigen Mitarbeitern, soll es nicht dauernd zu unbeabsichtigten Mißverständnissen oder gar zur bewußten Sabotage der „Bürokratisierung" und des „Berichtswesens" kommen.

Deshalb gehen die modernen Projektleitungs- und -dokumentationsverfahren von den allgemeinen Projektplanungs- und Programmiermethoden aus. Als Erweiterung und Ergänzung dieser Techniken können sie im Rahmen eines Kurses über

Softwareentwicklungsmethoden zwar nicht erlebt, aber doch wenigstens plausibel gemacht und verstanden werden.

Um zumindest die Grundideen und die Logik der Softwareentwicklung im Team und des Chefprogrammierer-Projektmanagements zu vermitteln, schien es den Verfassern am besten, den zu behandelnden Stoff in zwei Teile aufzugliedern. Sie werden hier, vielleicht etwas willkürlich, als

- Programmentwicklung und
- Systementwicklung

bezeichnet.

> Diese Unterscheidung entspricht der von *Parnas* [PARN00] eingeführten Unterscheidung zwischen
>
> - *Solo-Programmierung*, der Programmierung eines bestimmten Problems ausschließlich durch den Benutzer selbst, und
> - *Kooperativer Programmierung*, der Programmierung eines fremden Problems durch ein Programmiererteam.
>
> Für Parnas ist der Übergang von der Solo-Programmierung zur kooperativen Programmierung zugleich der Übergang von der „naiven" Vorgehensweise zum *Software Engineering*, der ingenieurmäßigen Entwicklung von Programmen in einer industriellen Umgebung.

Unter *Programmentwicklung* sollen hier diejenigen Verfahren und Techniken zusammengefaßt werden, welche für jede, auch die kleinste, Programmieraufgabe angewandt werden sollten. Im Mittelpunkt stehen hier die *strukturierte Programmierung*, die *schrittweise Verfeinerung* und der Begriff der *virtuellen Maschine*.

Die *Systementwicklung* stellt den Programmierer dann in die Umgebung eines Teams, von Auftraggebern und Benutzern, und in den stärker formalisierten Ablauf eines Produktionsprozesses, wie er in der industriellen Softwareentwicklung bei Hardware-Herstellern und Softwarehäusern üblich ist. War der Programmierer eines wissenschaftlich-technischen Anwendungsprogramms häufig noch sein eigener Auftraggeber und gleichzeitig auch der einzige Benutzer seines Programms, so ist dies jetzt nicht mehr der Fall. Durch die Erfordernisse der *Projektleitung* und *-überwachung* erhalten die Methoden zur Spezifikation, (Grob-) Planung, (Programm-) Abnahme und Dokumentation eine Bedeutung, die sie bei der ad hoc-Entwicklung eines einfachen Anwendungsprogrammes noch nicht hatten.

Sie ändern aber – und das ist das wesentliche Anliegen dieses Buchs – nichts an den im ersten Teil dargestellten Programmentwicklungsverfahren. Die Maßstäbe für „gute" und „schlechte" Programmierung bleiben die gleichen, ob es sich nun um ein kleines Anwendungsprogramm oder aber um einen Baustein eines großen Systems handelt.

1. Programmentwicklung

1.1 Die Aufgabe eines Programms

1.1.1 Maschinen und ihre Zustände

Eine Rechenanlage ist eine *Maschine.* Phantasievolle Bezeichnungen wie „Elektronengehirn" versuchen zuweilen darüber hinwegzutäuschen. Beschäftigt man sich jedoch nicht nur literarisch mit der Datenverarbeitung, so sollte man dies nie vergessen.

Als Maschine wird eine Rechenanlage durch einige Eigenschaften qualifiziert, welche sie mit allen anderen Maschinen des täglichen Lebens gemeinsam hat:

- sie nimmt vom Benutzer über irgendwelche Einrichtungen, wie Hebel, Knöpfe, Tastaturen, *Eingaben* entgegen,
- sie erfüllt entsprechend der jeweiligen Eingabe (solange sie funktioniert) eine *Aufgabe* und liefert ein *Ergebnis*,
- die Bearbeitung der Aufgabe erfolgt (zumindest makroskopisch betrachtet) deterministisch,
- auf Grund dieser deterministischen Bearbeitung der Aufgabe hat die Maschine zu jedem Zeitpunkt einen bestimmten *Zustand.*

> Mikroskopisch, bei genügend detaillierter Betrachtungsweise und einem hinreichend fein unterteilten Zeitmaßstab, braucht die Maschine nicht deterministisch zu arbeiten.
>
> In welcher Reihenfolge eine Kaffeemühle die einzelnen Kaffeebohnen zerkleinert, ist sicher zufallsbedingt. Auf der den Benutzer ausschließlich interessierenden Betrachtungsebene mit den drei wesentlichen Zuständen
>
> - abgeschaltet,
> - mahlend mit Kaffeebohnen im Vorratsbehälter,
> - laufend mit leerem Vorratsbehälter
>
> funktioniert sie jedoch deterministisch.
>
> Daß und unter welchen Umständen ein intern nicht deterministisch ablaufender Mechanismus trotzdem von außen gesehen deterministische Ergebnisse liefern kann, ist nicht Gegenstand dieses Buches. Modelle hierfür liefert die Theorie der Petri-Netze [DENN72, MISU73].

Die Maschine *interpretiert* die Eingabe. Ihr Folgezustand nach der Eingabe ist im allgemeinen abhängig von ihrem Zustand im Moment der Eingabe.

Ob und in welcher Richtung der Motor eines Aufzugs anläuft, hängt davon ab, wo sich der Fahrkorb beim Drücken eines Rufknopfes befindet.

Je mehr mögliche Zustände eine Maschine hat, umso komplexer ist ihre Funktionsweise und umso schwieriger ist ihre Konstruktion. Deshalb versucht man, den Gesamtoperationsumfang der Maschine aus einfacheren Elementaroperationen zusammenzusetzen.

Bei einer Waschmaschine wären derartige Elementaroperationen etwa

- Wassereinlauf,
- Wasser abpumpen,
- Wasser aufheizen,
- Waschmittel einspülen,
- Trommel bewegen,
- Schleudern.

Als Kombinationen dieser Elementaroperationen können dann viele verschiedene, komplexe Waschabläufe auf einfache Weise durch die Wahl unterschiedlicher Programme aktiviert werden.

Programme zur Steuerung der Aufgabenbearbeitung durch Maschinen sind also keine Besonderheit der elektronischen Datenverarbeitung.

Der Vergleich mit der Programmsteuerung einer konventionellen Maschine wurde hier nicht grundlos herangezogen. Er demonstriert nämlich den Unterschied zwischen drei Grundbegriffen, die auch in der EDV oft verwechselt werden:

- die *Aufgabe* ist definiert durch *Eingaben* (den *Input*) und soll als Ergebnis die *Ausgabe* (den *Output*) erzeugen,
- der Ablauf des Lösungsvorgangs der Aufgabe ist ein *Prozeß*, dessen Fortschritt durch den zu jedem Zeitpunkt erreichten *Zustand* (*State*) definiert wird,
- die Vorschrift für den Ablauf dieses Prozesses in Abhängigkeit von den jeweiligen Zuständen ist das *Programm*.

1.1.2 Benutzer- und Basismaschine

Die korrekte Herstellung von Programmen für Rechenmaschinen, durch den einzelnen Benutzer oder durch ein Team als „industrielle" Tätigkeit bei Hardware-Herstellern oder Softwarehäusern, ist der Gegenstand dieses Buchs.

Daß hierbei die exakte Definition der Aufgabe des Programms (Eingabe und Ausgabe) sowie die Darstellung und Abfrage der Zustandsvariablen während seines Ablaufs eine wichtige Rolle spielen werden, versteht sich nach dem vorher Gesagten von selbst.

Die *Programmierung* ist in der elektronischen Datenverarbeitung von zentraler Bedeutung, weil Rechenanlagen im Gegensatz zu allen üblichen Maschinen *frei programmierbar* sind. Bei anderen programmgesteuerten Maschinen werden die Programme dem Kunden vom Hersteller mitgeliefert. Ihr Aufbau und ihre physikalische Realisierung (als Schaltwalze, Lochstreifen, Verdrahtung . . .) sind ihm meist völlig gleichgültig und unbekannt. Dagegen ist eine EDV-Anlage, so wie sie vom Hardware-Hersteller zur Verfügung gestellt wird, zuerst einmal völlig „nutzlos". Sie ist eine aus Speichern, Verarbeitungs- und Ein-/Ausgabe-Einheiten bestehende *Basismaschine.* Erst das meist vom Benutzer selbst zu schreibende oder von ihm irgendwie sonst zu beschaffende Programm transformiert sie in die *Benutzermaschine*, die seine Aufgabe lösen kann (Abb. 1.1.2-1).

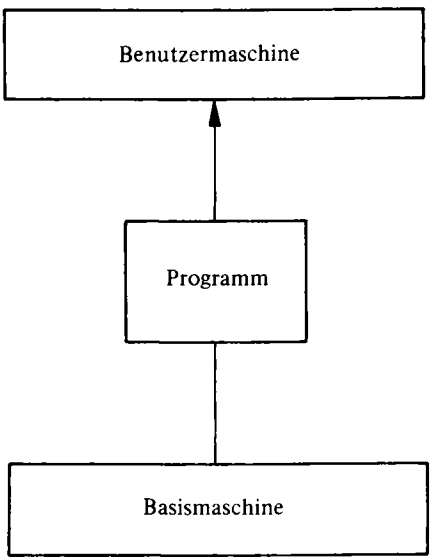

Abb. 1.1.2-1. Die Transformationsaufgabe eines Programms

Es ist verständlich, daß diese „nur" durch ein Programm realisierte Maschine oft als „weniger wirklich" als die Basismaschine empfunden wird. Es hat sich deshalb eingebürgert, die Hardware als *reale Maschine* zu bezeichnen – im Gegensatz zu einer *virtuellen* („scheinbaren") *Maschine*, die erst durch Software, durch ein oder mehrere Programme, auf der realen simuliert wird.

> Für den außenstehenden Benutzer ist die virtuelle Benutzermaschine jedoch ebenso „real" wie die Hardware – er kann nicht unterscheiden, welche Funktionen unmittelbar von der Hardware oder erst

mittelbar über ein Programm ausgeführt werden. Streng genommen ist sogar der Grundbefehlsvorrat einer modernen Rechenmaschine, etwa einer IBM /370, der einer „virtuellen" Maschine – er wird nämlich durch *Mikroprogramme* [HUSS70, SCHN72] aus einfacheren Operationen zusammengesetzt.

Ein Programm kann somit als Realisierung einer (virtuellen) Maschine mit Hilfe der Funktionen, des *Anweisungsvorrats*, einer anderen angesehen werden.

Der Vorteil der freien Programmierbarkeit ist, daß durch Auswechseln des Programms die *gleiche* Basismaschine in viele *unterschiedliche* Benutzermaschinen transformiert werden kann.

Die Hardware soll möglichst leistungsfähig und universell verwendbar sein. Deshalb ist ihr Anweisungsvorrat in der Regel komplex und nicht gut auf die jeweils zu lösende konkrete Anwendungsaufgabe abgestimmt. Um dem Programmierer die Überbrückung dieses großen Abstands zwischen der realen Hardware und der gewünschten Benutzermaschine zu erleichtern, werden von Hardware-Herstellern und Softwarehäusern Programmsysteme zur Vergüfung gestellt, die als *Basissoftware* dem Anwendungsprogrammierer bereits eine problemnähere und bequemer zu handhabende (virtuelle) Maschine simulieren.

Hierzu gibt es grundsätzlich zwei Methoden: *Betriebssysteme* [BRIN70, DIJK68b, HOAR72] und *Interpreter* für höhere Programmiersprachen realisieren diese „höheren" virtuellen Maschinen unmittelbar durch entsprechende Programme auf „tieferliegenden". *Compiler* [COCK70, GRIE71] hingegen übersetzen ein in einer höheren Programmiersprache (den Anweisungen einer virtuellen COBOL-, PL/1- oder FORTRAN-Maschine) geschriebenes Programm in eines für eine „tiefere" Basismaschine.

Da es für einen Programmierer zur Lösung seiner Programmieraufgabe grundsätzlich belanglos sein sollte, wie seine Basismaschine realisiert ist, kann man sich für die Diskussion einer Programmieraufgabe auf den Standpunkt stellen, daß der Funktionsvorrat der jeweiligen Basismaschine genau durch die verwendete Programmiersprache gegeben ist.

Für die Beispiele in diesem Buch wurde als Basismaschine, soweit nichts anderes gesagt wird, eine PASCAL Maschine gewählt. Diese interpretiere die Programmiersprache PASCAL [JENS74]. PASCAL-Programme werden im allgemeinen auch von Programmierern, die ALGOL oder PL/1 beherrschen, ohne große Mühe verstanden[1].

[1] Die nicht unmittelbar verständlichen Sprachkonstruktionen von PASCAL werden wir jeweils in Anmerkungen informell erklären. Für die formale Definition dieser Sprache sei auf die Literatur verwiesen [WIRT70].

Die Aufgabe jedes Programms ist es also, eine Basismaschine in eine Benutzermaschine zu transformieren. Damit ist zwangsläufig die genaue Definition dieser beiden Maschinen zugleich die Definition der zu lösenden Programmieraufgabe. In der industriellen Softwareproduktion ist dies der wichtigste Teil der *Spezifikation* eines Programmprodukts. Dieses Dokument ist die verbindliche Grundlage für Auftragsvergabe, Planung und Implementierung und wird in Abschnitt 2.2 näher besprochen. Aber auch ohne daß von einem Auftraggeber oder einer Finanzierungsstelle eine formale Spezifikation gefordert wird, sollte keine Programmentwicklung begonnen werden, solange nicht schriftlich zumindest die folgenden Angaben über die Aufgabenstellung niedergelegt sind:

Was ist die *Benutzermaschine*?

- Welche Eingabedaten verarbeitet sie, und woher stammen diese (Lochkarten, Tastatureingaben, zu verarbeitende Magnetbänder . . .)?
- Welche Ausgaben erzeugt sie (Druckerprotokolle, Sichtgeräteausgaben, Zeichnungen, neue oder geänderte Dateien . . .)?
- Welche Funktionen bietet die Benutzermaschine, und wie werden sie angefordert (Steuerkarten, Kommandos von Terminals oder vom Bedienungsblattschreiber . . .)?
- Wie soll sich die Benutzermaschine bei Fehlbedienung, Ausnahmebedingungen (z. B. Fehlen bestimmter Eingabedaten) oder falschen Eingaben verhalten?

Was ist die *Basismaschine*?

- Welche Programmiersprache wird benutzt?
- Welche Externspeicher stehen zur Verfügung?
- Welche Ein-/Ausgabe-Peripherie wird verwendet?

Je nachdem, ob die Basismaschine die „nackte" Hardware ist oder ob sie selbst bereits durch ein Betriebssystem und eine höhere Programmiersprache in eine virtuelle Maschine transformiert wurde, werden die Externspeicher und die Ein-/Ausgabe-Peripherie entweder physikalische Geräte oder symbolisch benannte Dateien und Ein-/Ausgabe-Datenströme sein. In beiden Fällen ist jedoch eine genaue Festlegung ihrer Eigenschaften notwendig. Zugriffsmöglichkeiten (Lesen, Schreiben, Ändern), zugelassene Verarbeitungsreihenfolgen (direkt, sequentiell), Steueroperationen (Zeilenvorschub, Rücksetzen, Suchvorgänge), vorgesehene Satzformate, Zeichenvorräte sowie die zu erwartenden und vom Programm zu bearbeitenden Ausnahme- und Fehlerbedingungen sind Vorgaben, die bereits die Programmplanung wesentlich bestimmen.

1.2 Die Aufgabenlösung durch „schrittweise Verfeinerung"

1.2.1 Beispiel: Häufigkeitszählung von Worten

Eine übliche Formulierung einer Programmieraufgabe wäre etwa die folgende:

1.2.1/1

> „Es soll ein PASCAL-Programm geschrieben werden, welches die Häufigkeit der einzelnen Worte in einem eingegebenen Text zählt und ausgibt."

Das reicht als Grundlage für die tatsächliche Programmierung noch nicht aus.

> Immerhin – es ist eine bessere Basis für eine Programmentwicklung als das, was häufig in umfangreichen Dokumenten festgelegt wird. Viele Vorgaben enthalten nämlich *zu viele* Angaben darüber, wie etwas getan werden soll (Algorithmen, Flußdiagramme, Datenformate, interne Tabellen, Bitmuster . . .) und *zu wenig* Information, was eigentlich die Aufgabe ist. Da ein Programm aber immer die Vorschrift für die Lösung einer Aufgabe durch die Rechenmaschine ist, sollte sich die Problemstellung auf das „was" beschränken: die Formulierung der jeweiligen Aufgabe mit ihrer Ein- und Ausgabe. Das „wie", die programmtechnische Realisierung, ist Sache des Programmierers. Man sollte ihm die Arbeit nicht dadurch erschweren, daß man ihn zwingt, aus einer vielleicht falschen oder unzweckmäßigen Beschreibung des „wie" erst auf das „was" zurückzuschließen.

Die obige Problemstellung definiert zwar eindeutig und klar die zu lösende Aufgabe – sie ist das, was man in der industriellen Softwareentwicklung als *Spezifikationsrahmen* bezeichnet. Sie ist aber noch zu allgemein: zu einer programmierungsreifen Vorgabe (einer *Spezifikation*) wird sie entsprechend Abschnitt 1.1.2 erst durch die genaue Festlegung der Benutzer- und Basismaschine mit ihren Ein- und Ausgaben, den Bedienungsmöglichkeiten und den Fehlerreaktionen.

> Wie weit würden Sie die folgende Spezifikation als vollständig und genau empfinden?

Was ist die *Benutzermaschine*?

– Ihre Eingabe besteht aus einem fortlaufenden *Text*. Dieser besteht aus *Worten*. Worte werden durch jeweils einen oder mehrere *Zwischenräume* und/oder einen *Punkt* (Satzende) getrennt. Der Text ist auf 80 Zeichen langen *Zeilen* (Lochkarten) gespeichert. Das Zeilenende ist immer auch Wortende.

- Als Ausgabe soll eine Aufstellung aller im Text vorkommender Worte mit der Zahl der Vorkommen jedes Wortes im eingelesenen Text ausgedruckt werden.
- Nach Laden und Starten des Programmes (der Benutzermaschine) soll ein Stapel von Textzeilen (Lochkarten) vom Lochkartenleser bis zum Ende eingelesen werden. Nach Ausdrucken der Ausgabeliste ist das Programm zu Ende (die Benutzermaschine schaltet ab).
- Sonderfälle:
  Enthält der Eingabekartenstapel keine oder ausschließlich leere Karten, so ist auch die Ausgabeliste leer.
  Die Zahl der möglichen verschiedenen Worte im Text und die maximale Wortlänge sind durch zwei einstellbare Parameter *listlength* und *wordlength* beschränkt. Sobald *listlength* verschiedene Worte erkannt wurden, werden neue Worte nicht mehr bearbeitet. Worte mit mehr als *wordlength* Zeichen werden auf diese Länge gekürzt.

Was ist die *Basismaschine*?

- Die Basismaschine sei eine PASCAL-Maschine.
- Die Karteneingabe sei eine sequentielle Datei mit dem Namen *input*.
- Die Druckausgabe sei eine sequentielle Datei mit dem Name *output*.

Für eine „gute“ Spezifikation fehlen auch in dieser Aufgabenstellung noch eine Reihe notwendiger Angaben:

Die Fehlerbehandlung durch die Benutzermaschine ist außerordentlich primitiv. So ist etwa jedes auf einer Lochkarte codierbare Zeichen (selbst wenn es vom Drucker nicht abdruckbar ist!) zulässiger Bestandteil eines Wortes, bis auf die (einzigen) Trennzeichen *Zwischenraum* und *Punkt*.

„Alkoholgehalt=85%; Vorsicht:Gift!“ wäre ein einziges Wort.

Ferner ist keine Sortierung der Ausgabeliste verlangt. Für die praktische Anwendung wäre sicher eine Sortierung nach dem Alphabet oder nach der ermittelten Worthäufigkeit zweckmäßig.

Suchen Sie andere Angaben in der Definition von Benutzer- und Basismaschine, die in der Grobformulierung der Aufgabe noch nicht vorhanden sind.

Fällt Ihnen ein noch nicht definierter Fehler- oder Ausnahmefall auf? (Hinweis: Was geschieht mit den Punkten am Ende des Satzes „Langsam verschwand der Kahn im Mondlicht . . .“?)

1.2.2 Vorgehen bei der Problemlösung

Die Programmieraufgabe besteht darin, die in Abschnitt 1.2.1 definierte Benutzer- auf die Basismaschine abzubilden, d. h. ein PASCAL-Programm zu schreiben, das den auf Lochkarten eingegebenen Text zu einer Liste der in ihm enthaltenen Worte mit ihren Häufigkeiten aufbereitet.

Dazu müssen aus den Basisanweisungen der PASCAL-Sprache höhere Programmeinheiten zusammengesetzt werden: komplexe Datenstrukturen, wie die aufzubereitende Liste mit Worten und Wortzählern, oder Anweisungsfolgen, wie etwa eine Wiederholungsschleife zur Ausgabe der (intern) erstellten Wortliste auf den Drucker nach Einlesen der letzten Eingabezeile.

Die Planung, der Entwurf des Programms, und damit die eigentliche intellektuelle Aufgabe des Programmierers besteht in der Zerlegung der Gesamtaufgabe in derartige kleinere Unteraufgaben. Diese sollen als eigenständige logische Einheiten den Abstand zwischen dem Anweisungsvorrat der Benutzermaschine (hier nur die einzige Funktion „lese Eingabekartenstapel und drucke Liste") und der Basismaschine (hier der PASCAL-Maschine) überbrücken,

> Dieser Abstand ist umso größer, je unterschiedlicher die beiden Maschinen sind: eine Programmierung in Assembler erfordert deshalb mehr Anweisungen, mehr Zwischenstufen und damit auch mehr Planung als die Benutzung einer „bequemeren" höheren Sprache.

Bei der Zerlegung in Unterkomponenten sind grundsätzlich zwei Vorgehensweisen möglich. Man kann entweder aus den Elementaranweisungen der Basismaschine höhere Programmeinheiten wie Prozeduren (Subroutinen) und Datenstrukturen entwickeln, aus diesen wieder höhere, und diesen Prozess solange fortsetzen, bis man (hoffentlich) bei der zu realisierenden Benutzermaschine „ankommt", bis sich also aus den höchsten so entwickelten Komponenten das gewünschte Programm einfach zusammensetzen läßt.

Man kann aber auch versuchen, das zu lösende Problem in „tiefere" logische Einheiten aufzuteilen, diese wieder in tiefere, bis derartige Einheiten sich leicht in wenigen Anweisungen der Basismaschine formulieren lassen. Das sicher unpräzise Wort „tief" soll hier „näher an der Basismaschine" bedeuten – ein Unterproblem, welches mit weniger Anweisungen dieser Basismaschine formulierbar ist als das Ausgangsproblem.

> Die dabei implizierte Vorstellung der „oben" liegenden Benutzermaschine und der „darunter" liegenden Struktur von Programmkomponenten bis zur ganz „tiefen" Basismaschine ließ für die erste Methode den Ausdruck *Bottom Up*, für die zweite die Bezeichnung *Top Down* entstehen.

Die erste Vorgehensweise, der schrittweise Aufbau immer höherer Programmeinheiten von „unten" nach „oben", ist sicher die näherliegende, naturgemäßere, damit aber auch die naivere. Es ist die Methode, nach der ein Kind seinen Baukasten benutzt, der Primitive seine Hütte baut und der Radioamateur sein erstes Funkgerät zusammensetzt. Es ist deshalb auch das Verfahren, nach dem wohl jeder Anfangsprogrammierer, noch unsicher in der von ihm benutzten Programmiersprache sein erstes Programm „zusammenbastelt".

Kein Ingenieur jedoch plant so eine Maschine, kein Architekt ein Haus und kein Komponist eine Symphonie. Der Fachmann unterscheidet sich vom unerfahrenen und ungeschulten Amateur dadurch, daß er mit einem Grobentwurf des Gesamtvorhabens anfängt und diesen schrittweise verfeinert. Dementsprechend wollen auch wir hier den Weg von der Benutzer- zur Basismaschine von „oben" nach „unten" zurücklegen.

> Diese Methode der *schrittweisen Verfeinerung*, der *Top Down-Programmentwicklung*, wurde von erfahrenen Programmierern schon seit langem intuitiv, aber oft inkonsequent durchgeführt. Als zweckmäßige heuristische Vorgehensweise zum Entwurf „besserer" Programme wurde sie erst Ende der 60er Jahre vor allem durch Dijkstra [DIJK69, DAHL72] und Wirth [WIRT71] propagiert. Ihre Einführung als verbindliche Standardmethode bei der industriellen Softwareentwicklung begann sich schließlich Anfang der 70er Jahre durch die Erfolge in einigen Großprojekten [MILL71] durchzusetzen. Diese Erfolge beruhen vor allem darauf, daß sich auf dem Top Down-Verfahren eine wesentlich bessere Methodik der Projektleitung und -überwachung aufbauen läßt, die wir in Abschnitt 2 dieses Buches besprechen werden.

Grundsätzlich läßt sich die Methode der schrittweisen Verfeinerung in jeder Programmiersprache anwenden. „Moderne" Programmiersprachen legen dieses Verfahren jedoch näher als ältere, wie etwa COBOL oder FORTRAN.

Welche Eigenschaften sollte eine Programmiersprache haben, um eine bequeme Top Down-Entwicklung eines Programmes zu ermöglichen?

Da schrittweise Verfeinerung bedeutet, daß – in der Regel wiederholt – ein „höheres" logisches Konzept in eine Reihe von „niedrigeren" aufgelöst werden muß, sollte die dadurch definierte Hierarchie von Anweisungen auch im Programmtext sichtbar sein. Dafür ist die Prozedurerklärung in PASCAL, ebenso wie in ALGOL oder PL/1, ein geeignetes sprachliches Mittel. Sie ermöglicht es, mehrere Anweisungen unter einem Namen zu einer Einheit zusammenzufassen. Eine logisch übergeordnete Prozedur kann sie nur über diesen Namen ansprechen.

Innerhalb einer Prozedur können jeweils neue lokale Variable und Prozeduren deklariert werden. Sie gelten nur innerhalb dieser und der von ihr eingeschlosse-

nen Prozeduren, nicht aber außerhalb. Eine derartige Prozedur kann als weitgehend unabhängig von ihrer Umgebung betrachtet werden. Sie ist dann nicht darauf angewiesen, daß höhere Prozeduren ihr die benötigten Arbeitsdaten-Bereiche als globale Datenbereiche zur Verfügung stellen.

> Wie weit erfüllen die folgenden Sprachelemente älterer Sprachen die obigen Forderungen:
>
> – die Kapitel- (SECTION-) Einteilung in COBOL [IBM00d],
> – das Unterprogramm (SUBROUTINE) in FORTRAN [USAS66],
> – der Kontrollabschnitt (CSECT) in BAL (/360 oder 4004-Assembler) [STRU71]?

Nicht nur (ausführbare) Anweisungen der Programmiersprache sollen hierarchisch strukturierbar sein, sondern auch Datenbereiche. Aus den Basis-Datentypen, wie sie die meisten Programmiersprachen bereitstellen (z. B. *integer, real, Boolean*), sollen dem jeweiligen Problem oder Unterproblem angepaßte höhere Datenstrukturen (**records**) zusammengestellt werden können.

Und ähnlich wie eine Prozedur erklärt, mit einem Namen versehen und an der für ihre Ausführung vorgesehenen Stelle im Code durch Aufruf ihres Namens vertreten werden kann, so soll auch einer Datenstruktur ein Name gegeben werden können. Eine derartige *Typendeklaration* macht sie dann zu einem höheren *Datentyp*, auf den sich der Programmierer bei der Deklaration von Variablen wie auf einen Basisdatentyp über den vereinbarten Namen beziehen kann.

> Die aus COBOL [IBM00d] und PL/1 [IBM00e, WEIN70a] bekannte Datenstrukturierung über *Level*-Nummern ist eine derartige Strukturierungsmöglichkeit. Diese Sprachen erlauben jedoch nicht das Benennen derartiger Strukturen durch Typen-Namen.
>
> Wie weit simulieren das DEFINED-Attribut in PL/1 und die *Dummy-Section* (DSECT) in BAL [STRU71] derartige Typendeklarationen?

PASCAL ist eine der wenigen Sprachen, welche auch zur Definition von höheren Datentypen sprachliche Mittel bereitstellen. Die Programmierung der Aufgabe 1.2.1/1 in PASCAL soll dafür als einfaches Beispiel dienen. Erst dann soll darauf eingegangen werden, welche Strukturierungsmittel für Programme bevorzugt oder ausschließlich verwendet werden sollten und wie sie in den üblichen Programmiersprachen realisierbar sind.

1.2.3 Die Top Down-Entwicklung des Programms

Wir beginnen die Programmplanung am besten mit der Festlegung der Datenstrukturen der Benutzermaschine. Laut Abschnitt 1.2.1 sind dies die (Loch-

karten-)Zeile (*Card*) von 80 Zeichen, das Wort (*Word*) von maximal *wordlength* Zeilen und der Listeneintrag (*Listentry*), der eine zusammengesetzte Datenstruktur (**record**) aus je einem Wort und einem ganzzahligen Zähler (*counter*) für die zu ermittelnde Worthäufigkeit ist. Die *Typenerklärungen* für diese Datenstrukturen in PASCAL sind

1.2.3/1

```
type    Card = array [1 . . cardlength] of char;
        Word = array [1 . . wordlength] of char;
        Listentry = record
                        word : Word;
                        counter : integer
                    end;
```

char, ein Zeichen, und *integer*, eine ganze Zahl, sind hierbei Basis-Datentypen von PASCAL. Der obere Index der beiden **array**s wurde jeweils symbolisch durch *cardlength* bzw. *wordlength* benannt. Durch vorangestellte Konstantenerklärungen, z. B.

```
const cardlength = 80;
      wordlength = 20;
```

kann diesen symbolischen Bezeichnungen ein konkreter Wert zugewiesen werden.

> Was ist der Vorteil einer derartigen symbolischen Benennung von Konstanten? Kennen Sie andere Programmiersprachen, die diese Möglichkeit bieten (Hinweis: EQU-Anweisung in BAL [STRU71])? Wie weit sind DATA in FORTRAN [USAS66], VALUE in COBOL [IBM00d] und INIT in PL/1 [WEIN70a] ein Ersatz hierfür?

In der **record**-Erklärung für die Datenstruktur *Listentry* wurde der vorher erklärte Typ *Word* verwendet.

> Eine Änderung des Datentyps *Word*, z. B. durch eine Änderung der maximalen Zeichenzahl *wordlength*, würde damit automatisch auch für *Listentry* wirksam.

Mit den so definierten Datentypen können nun diejenigen Variablen erklärt werden, die *globalen* Charakter für das Programm haben. Dies sind sicher die Ein- oder Ausgaben der Benutzermaschine, laut Abschnitt 1.2.1 also die sequentielle Eingabe-Datei *input* aus Zeilen (*Card*) und eine sequentielle Ausgabe-Datei *output* von *Listentry*-Daten für die Druckausgabe (*print*).

Ferner benötigen wir eine (intern aufzubauende) Wortliste *list*, die maximal *listlength* Worte mit ihren Häufigkeitszählern fassen kann und am Ende ausgedruckt werden soll, sowie einen Zähler *entrycounter*, der zu jedem Zeitpunkt angibt, wie viele *Listentry*-Einträge diese Liste bereits enthält.

Die Erklärung dieser globalen Variablen lautet in PASCAL

1.2.3/2[2]

```
var     input  [in]    : file of Card;
        output [print] : file of Listentry;
        list : array [1 .. listlength] of Listentry;
        entrycounter : integer;
```

Die mit dem Attribut *print* erklärte Datei *output* ist eine Druckausgabe-Datei von *Listentry*-Einträgen[3]. Deklaration 1.2.3/2 definiert *list* als eine Datei von *Listentry*-Daten. Gemäß 1.2.3/2 sind dies selbst keine Basis-Datentypen, sondern aus Feldern mit den Namen *word* und *counter* zusammengesetzte **records**[4].

Nachdem die Daten auf oberster Ebene definiert sind, ist die grobe Formulierung des Algorithmus nahezu trivial. Es muß

- zuerst sicher eine *Initialisierung* stattfinden,
- dann folgt eine *Inputverarbeitung*
- und schließlich ein *Listedrucken*.

1.2.2/3

```
begin
Initialisierung;
Inputverarbeitung;
Listedrucken
end.
```

[2] **file** erklärt eine sequentielle Datei von Daten des angegebenen Typs. [*in*] bedeutet, daß die Datei *input* eine Eingabedatei ist. Mit der PASCAL-Anweisung *get* (*input*) kann diese auf das nächste Eingabedatum vom Typ *Card* vorgesetzt werden: die jeweils aktuelle, gerade gelesene Kartenzeile wird in den auf sie bezugnehmenden Anweisungen mit *input*↑ bezeichnet.
Zu jeder Eingabe-Datei sieht PASCAL eine Boole'sche Variable *eof* vor, welche angibt, ob das Ende der betreffenden Eingabedatei erreicht wurde. Eine Boole'sche Variable kann die Werte *false* und *true* haben.

[3] Mit *put* (*output*) kann jeweils die aktuelle Zeile ausgeschrieben und die nächste initialisiert werden. Diese noch nicht gedruckte Zeile wird – entsprechend der Notation bei Eingabedateien – durch *output*↑ bezeichnet.

[4] In PASCAL wird ein Feld in einem **record** dadurch angegeben, daß der Feldname, durch einen Punkt abgetrennt, hinter den **record**-Namen geschrieben wird. Ist also etwa *z* als record *x* : *integer*; *y* : *Boolean* **end** deklariert, so bedeutet *z.x* das (integer-) Feld *x* im **record** *z*.

Initialisierung, *Inputverarbeitung* und *Listedrucken* sind hier Aufrufe von Prozeduren[5]. Die *schrittweise Verfeinerung* besteht darin, daß zuerst symbolische Namen für die (u. U. komplexen) Aktionen eingesetzt werden und diese dann in den folgenden Entwurfsschritten selbst als Prozeduren formuliert werden. Dies wird solange wiederholt, bis die „tiefsten" Prozeduren keine Prozeduraufrufe mehr, sondern nur noch Anweisungen der Basissprache enthalten.

> Wenn alle Prozeduren formuliert sind, kann entschieden werden, ob sie im endgültigen Programm auch als Prozeduren belassen werden sollen. Sie müssen dann jeweils zwischen den Variablendeklarationen (**var** . . .) und den ausführbaren Anweisungen (**begin** . . . **end**) des Programms als **procedure** . . . erklärt werden. Kommen ihre Aufrufe (wie *Initialisierung*, *Inputverarbeitung* und *Listedrucken*) im Programm hingegen nur einmal vor, so wird man in der Regel aus Effektivitätsgründen ihren Text bei der endgültigen Fertigstellung des Programms anstelle des Aufrufes als *inline*-Code einfügen.

1.2.3/4 faßt die bis jetzt in 1.2.3/1 bis 1.2.3/3 geleistete Programmentwicklung zusammen:

1.2.3/4

```
program Häufigkeitszählung;
{Konstanten-Erklärungen[6]}
type Card      = array  [1 . . cardlength] of char;
     Word      = array  [1 . . wordlength] of char;
     Listentry = record
                   word  : Word;
                   counter : integer
                 end;

var  input  [in]    : file of Card;
     output [print] : file of Listentry;
     list : array [1 . . listlength] of Listentry;
     entrycounter : integer;

{Prozedur-Erklärungen}

begin
Initialisierung;
Inputverarbeitung;
Listedrucken
end.
```

Fußnoten 5 und 6 folgen auf Seite 28.