**Zeitschrift:** Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association suisse des électriciens, de l'Association des entreprises électriques suisses

**Herausgeber:** Schweizerischer Elektrotechnischer Verein ; Verband Schweizerischer Elektrizitätsunternehmen

**Band:** 78 (1987)

**Heft:** 1

**Artikel:** Modulare Programmierung mit Modula-2

**Autor:** Gutknecht, J.

**DOI:** https://doi.org/10.5169/seals-903794

# Modulare Programmierung mit Modula-2

J. Gutknecht

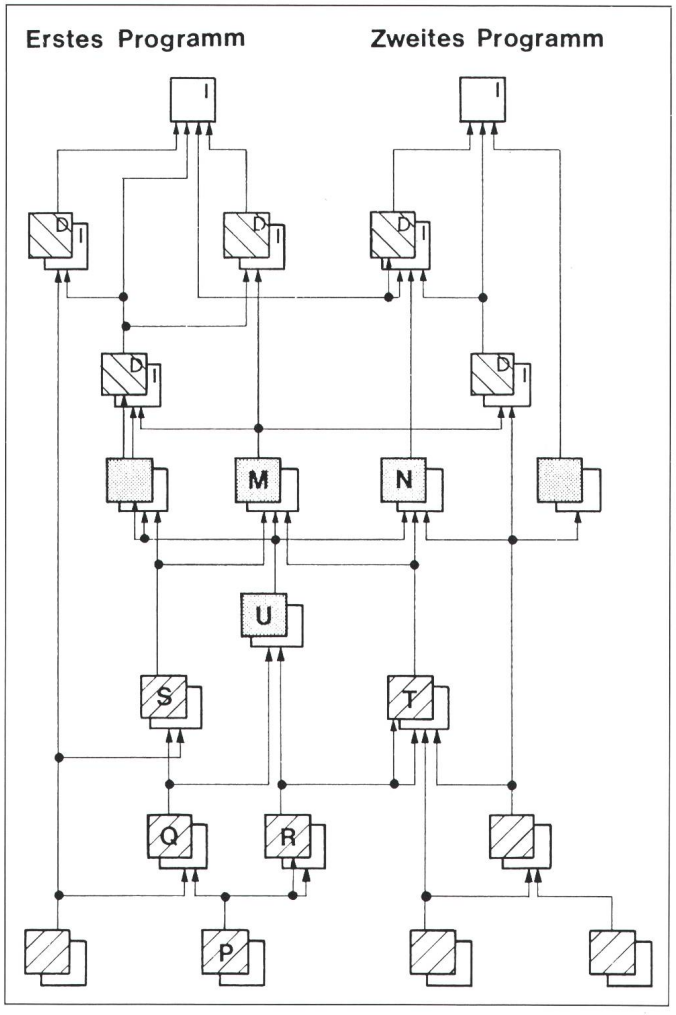
***Modula-2 führt den Modulbegriff ins Software-Engineering ein. Damit lassen sich komplexe Programme dank klarer Schnittstellendefinition konsequent strukturieren. Der Beitrag beschreibt die Konzepte von Modula-2, ohne vom Leser explizite Kenntnisse dieser Sprache zu verlangen. Modula-2 wurde wie übrigens auch Pascal von Niklaus Wirth an der ETH Zürich entwickelt.***

***Modula-2 introduit la notion de module dans le domaine du logiciel et permet de structurer systématiquement des programmes complexes grâce à une définition claire des interfaces. L'article décrit la conception de Modula-2 sans exiger des connaissances particulières de ce langage. Comme Pascal, Modula-2 a été conçu par Niklaus Wirth, de l'EPF de Zurich.***

**Adresse des Autors**
Prof. Dr. *Jürg Gutknecht*, Institut für Informatik, ETH-Zentrum, 8092 Zürich.

## 1. Das Software-Modul

Die tragende Idee der Modularisierung ist die Gliederung von Konstruktionen in autonome funktionale Einheiten, sogenannte *Module* (Tab. I). Im Gegensatz zu zahlreichen technischen Produkten, wie z. B. Bauwerke, TV- und Hi-Fi-Anlagen, Fotoapparate, Computer und elektronische Schaltungen, entbehrten Softwaresysteme bis vor kurzem einer expliziten modularen Struktur. Das Konzept des *Software-Moduls* geht auf *David Parnas* zurück [1]. *Parnas* erkannte, dass zur Meisterung der Komplexität grosser Programmsysteme neue Methoden des Software-Engineerings erforderlich sind. Wenig überraschend schlugen sich diese Ideen alsbald in einer qualitativen Weiterentwicklung der Programmiersprachen nieder. Tatsächlich ist in der Zwischenzeit eine Generation neuer, *modularer* Sprachen entstanden. Wir erwähnen in chronologischer Reihenfolge Mesa (Xerox PARC), Modula-2 [2], Chill und Ada.




**Figur 1**
**Schema eines modularen Softwaresystems**

- □ Hauptmodule
- ▨ Anwendermodule
- ▩ Bibliotheksmodule
- ▢ Betriebssystemmodule
- D Definitionsteil
- I Implementationsteil
- → Pfeilrichtung bezeichnet Importrichtung. Bsp. Modul M importiert, d.h. benützt die Module S, U, T. Man beachte, dass Definitions- und Implementationsteile Objekte importieren können.

## Modula-2-Begriffe

**Compiler**
Übersetzer. Übersetzt Programme, die in der Quellensprache (z.B. Modula-2) geschrieben sind, in die Maschinensprache des betreffenden Computers.

**Importieren**
In Modula-2 lassen sich Objekte aus fremden Modulen, genauer aus deren Definitionsteil, in eigenen Modulen verwenden. Im Gebrauch besteht zwischen diesen importierten und den eigenen Objekten kein Unterschied.

**Konstrukt**
Syntaktische Einheit einer Programmiersprache, z.B. eine Deklaration oder eine Anweisung.

**Modul**
In sich abgeschlossener Programmteil. Enthält Datenstrukturen und Operationen (Prozeduren) auf diesen Datenstrukturen. In Modula-2 zerfällt jedes Modul in einen Definitions- und einen Implementationsteil.

**Modul-Definition**
Spezifikation der Schnittstelle und der Funktion des Moduls. Die Modul-Definition enthält die Deklaration öffentlicher (d.h. ausserhalb des Moduls gültigen) Konstanten, Typen und, in seltenen Fällen, öffentlicher Variablen. Ferner sind die Namen und Parameterlisten der vom entsprechenden Modul zur Verfügung gestellten (exportierten) Prozeduren wesentlicher Bestandteil der Modul-Definition.

**Modul-Implementation**
Die Implementation enthält private, d.h. rein modulinterne Daten und die eigentlichen Programm-Routinen, die zu den in der Definition spezifizierten Prozeduren gehören.

**Monitor**
Aktuell im Zusammenhang mit Systemen, in welchen (quasi-)gleichzeitig mehrere Prozesse ablaufen. Monitoren sind Module, welche kritische Programmabschnitte enthalten, d.h. Abschnitte, die auf gemeinsamen Datenstrukturen operieren. Monitoren garantieren gegenseitigen Ausschluss, d.h. sie gewährleisten, dass zu keiner Zeit mehr als ein Prozess einen kritischen Programmabschnitt durchläuft. Modula-2-Module werden durch die Angabe einer Priorität als Monitoren gekennzeichnet.

**Objekt**
Element eines Modula-2-Programms. Typische Objekte sind Konstanten, Variablen und Prozeduren. Datentypen werden ebenfalls als Objekte, gewissermassen als Objekte «höherer Stufe», betrachtet.

**Prozedur**
Programm-Routine, d.h. Unterprogramm. Eine Parameterliste spezifiziert die dem Unterprogramm beim Aufruf zu übergebenden Parameter. Prozeduren können in Modula-2 als Funktionen auftreten indem sie ein explizites Resultat zurückliefern. Ist z.B. sin eine Funktionsprozedur, die ein REAL-Resultat zurückliefert, so bewirkt die Anweisung y:=sin(x) die Zuweisung des Resultates an die REAL-Variable y. Der Resultattyp einer Funktion darf nicht strukturiert sein.

**Prozess**
Ablauf einer logisch zusammengehörigen Folge von Aktionen.

**Typ**
In Modula-2 ist jede Konstante und jede Variable einem Typ zugeordnet, der den möglichen Wertebereich bzw. die Struktur der Konstanten oder Variablen angibt. Standard-Typen sind INTEGER (Ganzzahl), REAL (reelle Zahl), BOOLEAN (Boolesche Grösse), CHAR (alphanumerisches Zeichen), BITSET (Zahlenmenge) usw. Ferner sind eigene Typen möglich, z.B. Aufzählungstypen (Color = (red, yellow, green)) oder strukturierte Typen, nämlich Folgen (ARRAY) und Datensätze (RECORD). In den Programmfragmenten dieses Textes treten z.B. die eigenen Typen File, Viewer und Event auf.

**Zeiger**
Zeiger sind Hilfsmittel zur Konstruktion von dynamischen Datenstrukturen wie Listen und Bäume. In Modula-2 sind Zeiger verbunden mit einem Basistyp. Jeder Zeiger zeigt auf ein Objekt des entsprechenden Basistyps. In einem Binärbaum mit Knoten des Typs T enthält beispielsweise jeder Knoten zwei Zeiger zu den beiden «Söhnen» dieses Knotens. Zeiger werden in Form von Speicheradressen realisiert.

**Zielmaschine**
Computer, in deren Maschinensprache ein Compiler übersetzt.

**Tabelle I**

Die zweite dieser Sprachen, Modula-2, ist Gegenstand des vorliegenden Aufsatzes. Sie wurde von *Niklaus Wirth* definiert und trat um 1980 die Nachfolge von Pascal an.

Als bemerkenswertes Faktum ist zu erwähnen, dass Modula-2 aus Modula-1, einer Studie zur Programmierung gleichzeitig ablaufender Prozesse, hervorgegangen ist. In Modula-1 spielen die Module die Rolle sogenannter *Monitoren*, das sind kritische Programmabschnitte, die nicht gleichzeitig von verschiedenen Prozessen durchlaufen werden dürfen. In Modula-2 sind Monitoren durch Hinzufügen einer *Priorität* zum Modulnamen gekennzeichnet. Die wichtigere und allgemeinere Bedeutung des Modulkonzeptes trat erst später in den Vordergrund.

Wesentlich zum Verständnis modularer Systeme ist die Erkenntnis, dass die einzelnen Module zwar autonom, nicht aber unabhängig voneinander sind. Vielmehr sind sie in einem Netz von Abhängigkeiten verwoben. Der Konstrukteur eines Moduls wird seine Arbeit auf vorhandene, möglicherweise anderswo entwickelte Module abstützen. Entscheidend dabei ist, dass er zwar deren *Schnittstellen*, d.h. funktionale *Definitionen* (s. Begriffe), genau kennen muss, nicht aber die Methoden der *Implementation*, zumindest nicht in allen Details.

Verweilen wir einen Moment bei diesem letzten Punkt. Er hat die weitreichende Konsequenz, dass keine direkten Abhängigkeiten zwischen den Implementationen der verschiedenen Module bestehen. Nach der Festlegung der funktionalen Definitionen (Schnittstellen) lassen sich in einem solchen System sämtliche Implementationen (eigentliche Programmroutinen) unabhängig voneinander ausarbeiten. Ohne das konstruktive Gebäude als Ganzes zu erschüttern, kann jede Implementation jederzeit durch eine neue ersetzt werden.

Betrachten wir nun das in Figur 1 abgebildete Softwaresystem. Ausser den beiden obersten Modulen besteht jedes Modul aus einem *Definitionsteil* und einem *Implementationsteil*. Auf den beiden obersten Modulen wird nicht weiter aufgebaut. Sie können als *Hauptmodule* interpretiert werden, die den dynamischen Ablauf steuern, sobald ihnen die Kontrolle übergeben wird. Ein Hauptmodul, zusammen mit allen direkt und indirekt *importierten*, d.h. verwendeten Modulen wird als *Programm* bezeichnet.

Wir bemerken, dass verschiedene Programme nicht notwendigerweise

modulfremd sind. Beispielsweise werden die Module M und N von beiden in Figur 1 dargestellten Programmen importiert. Module, die so ausgelegt sind, dass sie in mehreren Programmen verwendet werden können, heissen *Bibliotheksmodule*. Bestimmte, in der Hierarchie weit unten (in Fig. 1 ebenfalls unten) angesiedelte Bibliotheksmodule stellen die Schnittstellen zu den Ressourcen des Computers wie Prozessor, Speicher, Disk, Netzwerk, Bildschirm, Eingabegeräte usw. dar. In ihrer Gesamtheit bilden sie das *Betriebssystem*.

Da sich alle Module in einheitlicher Form präsentieren, wirkt ihre Einteilung in verschiedene Klassen künstlich. Tatsächlich tritt in modularen Softwaresystemen die traditionelle Strukturierung in die *horizontalen* Schichten Betriebssystem, Bibliothek und Anwendung zugunsten einer thematischen Gliederung in *vertikale* Programme in den Hintergrund.

```
DEFINITION MODULE InOut;                          KOMMENTAR
   FROM FileSystem IMPORT File;
   VAR Done: BOOLEAN;                             Resultat der letzten Operation
      in, out: File;                              Ein- und Ausgabefiles
   PROCEDURE OpenInput (defext: ARRAY OF CHAR);Eröffne Eingabefile mit Namens-Suffix defext
   PROCEDURE OpenOutput (defext: ARRAY OF CHAR);Eröffne Ausgabefile mit Suffix defext
   PROCEDURE CloseInput;                          Schliesse Eingabefile (Rückkehr zur Tastatur)
   PROCEDURE CloseOutput;                         Schliesse Ausgabefile (Rückkehr zu Bildschirm)
   PROCEDURE Read (VAR ch: CHAR);                 Lies nächstes Zeichen vom Eingabefile
   PROCEDURE ReadString (VAR s: ARRAY OF CHAR);Lies Zeichenkette vom Eingabefile
   PROCEDURE Write (ch: CHAR);                    Schreibe nächstes Zeichen auf Ausgabefile
   PROCEDURE WriteLn;                             Schliesse Zeile ab
   PROCEDURE WriteString (s: ARRAY OF CHAR);Schreibe Zeichenkette auf Ausgabefile
END InOut;

DEFINITION MODULE SYSTEM;                         Für Motorola 68000
   TYPE ADDRESS = POINTER TO BYTE;                Adresse = Zeiger zu irgend einem Byte
      BYTE;                                       Kleinste adressierbare Einheit
   PROCEDURE ADR (VAR x: AnyType): ADDRESS;Speicheradresse der Variablen x
   PROCEDURE TSIZE (AnyType): INTEGER;            Grösse in Bytes von AnyType
   PROCEDURE VAL (NewType; x: AnyType): NewType;Uminterpretation der Grösse x als NewType
END SYSTEM;
```

**Tabelle II. Definitionsteile der Module InOut und System**

## 2. Das Modula-Konzept

### 2.1 Grundlagen

Eine wichtige Folge der im letzten Abschnitt besprochenen Neuorientierung der Struktur modularer Systeme ist die Forderung nach einer grossen Einsatzbandbreite der Programmiersprache. Neben dem Angebot an Konstrukten zur Formulierung abstrakter Abläufe und Datenstrukturen ist die Möglichkeit der «maschinennahen» Programmierung von zentraler Bedeutung. Der prinzipielle Aufbau von Modula-2, fortan kurz *Modula* genannt, zeigt eine interessante Lösung, die sich das Modulkonzept selbst zunutze macht. Die eigentliche Sprache besteht aus einem *minimalen* Satz allgemeiner, maschinenunabhängiger Konstrukte. Alle systemabhängigen Objekte werden über Modulschnittstellen (Definitionen) zur Verfügung gestellt, so beispielsweise Objekte zur Behandlung von Ein- und Ausgabe sowie von Files. Das Standardmodul *SYSTEM* nimmt eine Sonderstellung ein. Es stellt gewissermassen die Verbindung der Sprache mit dem Computer her und ermöglicht dadurch eine maschinennahe Programmierung. Beispielsweise lässt sich mit Hilfe dieses Moduls die Abbildung abstrakter Datenstrukturen in den Speicher explizit programmieren.

Zur Illustration flechten wir an dieser Stelle Auszüge aus den Definitionsteilen der Module *InOut* und *SYSTEM*[1] ein (Tab. II). Sie lassen erkennen, dass Operationen in Form von *Prozedurköpfen* definiert werden. Die IMPORT-Anweisung spezifiziert die aus fremden Modulen übernommenen Objekte, die übrigen Deklarationen des Definitionsteils legen die Objekte fest, die von anderen Modulen verwendet (importiert) werden können. In Modula ist jedem Objekt ein bestimmter, wohldefinierter *Typ* zugeordnet. Beispielsweise besitzt *Done* im Modul *InOut* den Standardtyp BOOLEAN, und *in* den von *FileSystem* importierten Typ *File*. Operationen sind nur dann legal, wenn die Typen der beteiligten Objekte verträglich sind[2]. Der Typ ARRAY OF BYTE gehört in die Kategorie der sogenannten *dynamischen Arraytypen*. Dynamische Arrays sind eines von drei wichtigen Modula-Konzepten, die eng mit dem Modulprinzip verknüpft sind. Bei den beiden anderen handelt es sich um *abstrakte Objekttypen* und *Prozedurvariablen*.

[1] Modula unterscheidet zwischen Gross- und Kleinbuchstaben.

[2] Die erwähnte Möglichkeit der maschinennahen Programmierung bei der Verwendung des Moduls *SYSTEM* beruht zu einem guten Teil auf den grosszügigen Verträglichkeitsregeln der Typen BYTE und ADDRESS. ADDRESS ist mit jedem Zeigertyp und mit ganzen Zahlen verträglich, BYTE mit jedem Typ von Bytegrösse, und ARRAY OF BYTE mit jedem Typ überhaupt. Es wird betont, dass, im Gegensatz zu Pointern, eine Variable vom Typ ADDRESS nicht notwendigerweise zur Basis eines Datensatzes zeigen muss. Adressen können beliebige Speicherpositionen ansprechen. Adressarithmetik wird durch die Verträglichkeit von Adressen mit ganzen Zahlen ermöglicht.

### 2.2 Dynamische Arrays

In Bibliotheksprozeduren, die einen Array als Parameter aufweisen, ist die Spezifikation der Länge des Arrays oft unerwünscht. Der Kreis der potentiellen Kunden der Prozedur ist viel grösser, wenn beim Aufruf ein Array beliebiger Länge akzeptiert wird. Dies kann in Modula dadurch erreicht werden, dass der Indexbereich in der Prozedurdeklaration offengelassen wird.

*Beispiele:*

```
PROCEDURE WriteString
(s: ARRAY OF CHAR);
PROCEDURE InnerProduct
(a, b: ARRAY OF REAL): REAL;
PROCEDURE WriteBlock
(f: File; b: ARRAY OF BYTE)
```

*WriteString* schreibt eine Zeichenkette s beliebiger Länge auf den Bildschirm. *InnerProduct* ist eine *Funktion*. Sie akzeptiert Zahlenvektoren a und b beliebiger (gleicher) Länge und gibt deren Skalarprodukt als Resultat (vom Typ REAL) zurück. *WriteBlock* erweitert das File f um einen Block b beliebiger Grösse und Struktur.

### 2.3 Abstrakte Objekttypen

Die Modulidee ist dann in besonders reiner Form verwirklicht, wenn innerhalb des Moduls eine bestimmte Art von Objekten vollständig abgehandelt wird, d.h. wenn der Modul selbst sämtliche für diese Objektart benötigten Operationen zur Verfügung stellt. In diesem Fall ist die Struktur der Objekte für die Kunden des Mo-

duls belanglos. Es ist z. B. für den Benützer eines Moduls ComplexNumbers unwesentlich, ob komplexe Zahlen die Struktur
TYPE Complex = ARRAY [1...2] OF REAL
oder
TYPE Complex = RECORD re, im: REAL
END
aufweisen. Durch Weglassen jeglicher Strukturbeschreibung deklariert man den abstrakten TYPE Complex.

### 2.4 Prozedurvariablen

Moderne interaktive Systeme sind gelegentlich «ereignisgesteuert». Dies bedeutet, dass das Ablaufs-Kontrollprogramm die Ereignisse (Eingabe, Zeigen auf ein am Bildschirm dargestelltes Objekt, Alarm der internen Uhr usw.) feststellt und den interessierten Programmen zur Behandlung übergibt. Es müssen also Module von tieferer Hierarchiestufe übergeordnete Module anstossen können. Dies setzt aber voraus, dass in den ersteren Behandlungsprozeduren installiert werden können. In Modula lassen sich Prozeduren als Parameter übergeben und als Prozedurvariablen (im hierarchisch tieferen Modul) registrieren.

*Beispiel:*

```
TYPE Handler = PROCEDURE
(Viewer, Event);
PROCEDURE OpenViewer
(VAR v: Viewer; h: Handler)
```

Viewer und Event sind in diesem Beispiel Datentypen, die ein Bildschirmfenster bzw. ein Ereignis beschreiben. Beim Aufruf von OpenViewer wird eine Behandlungsprozedur h übergeben. Diese wird als Prozedurvariable des Fensterobjektes v registriert und aufgerufen, wenn auf das betreffende Fenster gezeigt wird.

### 2.5 Weitere Aspekte der Modularisierung

Damit kennen wir alle wichtigen modulorientierten Einrichtungen von Modula. Die Erfahrung hat gezeigt, dass ihre sinnvolle Verwendung, d. h. die gute Modularisierung, viel schwieriger ist, als man auf den ersten Blick vermutet. Der Modul-Designer sieht sich häufig mit kontroversen Zielsetzungen konfrontiert. Die folgenden Ausführungen mögen dies beleuchten.

Wir rufen zunächst in Erinnerung, dass kein Modul – was die Struktur angeht – von der Implementation eines anderen Moduls abhängig ist. Deshalb hat die Veränderung einer Implementation überhaupt keine Auswirkungen auf die Umgebung. Hingegen beeinflussen Modifikationen einer Modul-Definition prinzipiell alle direkt oder indirekt abhängigen Module. Beispielsweise invalidiert die Modifikation der Definition von P in Figur 1 die Module Q, R, T, U, M und N, da Q und R direkt und T, U, M und N indirekt von P abhängig sind. Der Modul S hingegen bleibt gültig, d. h. strukturell intakt, da nur der Implementationsteil betroffen ist, was definitionsgemäss nach aussen nicht sichtbar ist. Unerwünschte Kettenreaktionen bei der Invalidierung von Modulen entstehen stets dann, wenn abhängige Modul-Definitionen (wie R und T) ins Spiel kommen. Deshalb sollte auf grösstmögliche Unabhängigkeit der Modul-Definitionen geachtet werden. Anderseits ist natürlich der intensive Gebrauch der bestehenden Module ein Grundziel. Eine weitere Kontroverse betrifft die Art der in die Modul-Schnittstelle aufgenommenen Operationen. Je grösser die Komplexität, desto enger der Verwendungsbereich der Operation, je geringer die Komplexität, desto grösser der Verwendungsbereich, aber um so geringer der Nutzen. Gute Modularisierung bedeutet Suchen nach dem goldenen Mittelweg. Dieser ist im allgemeinen abhängig von der Art des Moduls. Der interessierte Leser möge sich überlegen, welche der genannten kontroversen Zielsetzungen bei Anwendermodulen und welche bei Bibliotheksmodulen im Vordergrund stehen.

## 3. Modula-Compiler als Beispiel der Modularisierung

Zur Illustration der Modularisierung soll ein besonders wichtiges und attraktives Programm, nämlich ein *Modula-Compiler*, dienen. Der eigentlichen Behandlung der Modularisierungsfrage schicken wir einige Bemerkungen über die Aufgabe und Funktionsweise des Compilers voraus, die von übergeordnetem Interesse sind.

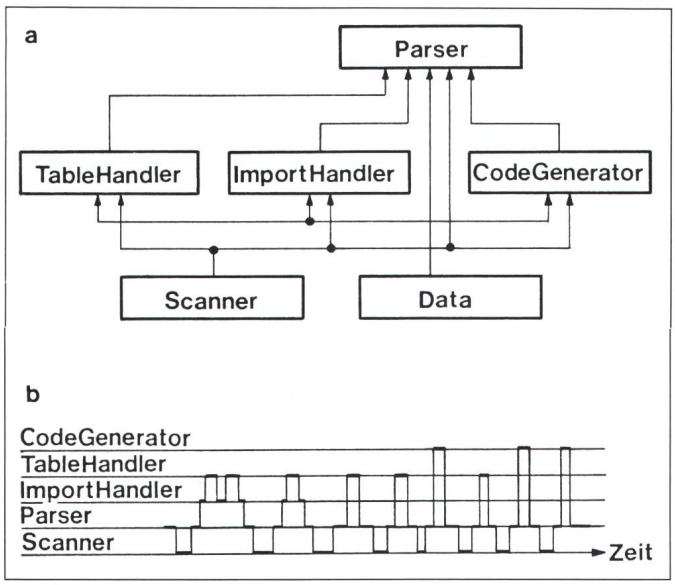
Ein Programmsystem stellt eine Spezifikation von Abläufen dar. Damit diese Abläufe vom Computer ausgeführt werden können, muss das Programmsystem in der Maschinensprache des Computers vorliegen. Die Hauptaufgabe des Compilers ist die Übersetzung von Programmtexten in Folgen von Maschinenbefehlen. Es wäre natürlich unpraktisch, wenn nach der geringsten Änderung das ganze System neu übersetzt werden müsste. Wir wissen beispielsweise bereits, dass sich eine Änderung innerhalb einer Implementation nicht auf die Umgebung auswirkt. Deshalb macht der Modula-Compiler Implementationsteile einer *separaten* Übersetzung zugänglich. Im Gegensatz zur viel einfacher zu realisierenden *unabhängigen* Übersetzung wird bei der *separaten* Übersetzung die Konsistenz mit dem zugehörigen Definitionsteil und mit den importierten Modulen überprüft.

Nach dieser kurzen Beschreibung der Aufgabe des Compilers wenden wir uns nun seiner Konstruktion zu. Da der Modula-Compiler selbst ein Programm ist, kommt sofort die Idee auf, ihn in seiner eigenen Sprache, also in Modula, zu formulieren. Diese Idee erhält noch mehr Gewicht durch den folgenden Sachverhalt: Nach der Fertigstellung eines Compilers wird häufig der Wunsch laut, ihn einer neuen Zielmaschine (Tab. I) anzupassen. Die Anpassung geht natürlich um so leichter von der Hand, je besser der zielmaschinenabhängige Teil isoliert ist. Im Idealfall besteht die Anpassung lediglich im Austausch eines Moduls.

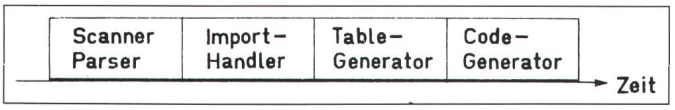
Die Figur 2a zeigt den modularen Aufbau des Modula-Compilers. Es handelt sich um einen sogenannten *Einphasencompiler*, der Programmtexte in einem einzigen Durchlauf in Maschinenbefehle übersetzt. Die Themen der Module entsprechen den Stationen, die bei der Verarbeitung der Texteinheiten zu Befehlsfolgen durchlaufen werden müssen.

Der *Scanner* übersetzt die Zeichenfolge in eine Folge von Modula-Sprachsymbolen, der *Parser* prüft die syntaktische Korrektheit der Symbolfolge, der *ImportHandler* liest die importierten Objekte ein und prüft ihre gegenseitige Konsistenz, der *TableHandler* nimmt die deklarierten Objekte mit ihren Attributen in die sogenannte Symboltabelle auf, und der *CodeGenerator* schliesslich erzeugt die Folge der Maschinenbefehle.

Einige ergänzende Bemerkungen sind angebracht. Sie mögen dem Leser einen Blick hinter die Kulissen dieses Programmaufbaues verschaffen. Zu-




**Figur 2**
**Einphasen-Modula-Compiler**
a Modularer Aufbau
b Zeitdiagramm




**Figur 3**
**Ablauf einer Mehrphasen-übersetzung**

## 4. Entwicklung eines Modula-Programms

Wir wenden uns einem letzten Problem zu: Anhand einer einfachen Textformatieraufgabe soll versucht werden, eine kurze, exemplarische Einführung in die Programmierung mit Modula zu geben. Die Problemstellung lautet: Es soll ein Programm erstellt werden, das einen Text (Tab. III) in mehrspaltige Blocksatzform giesst und seitenweise ausdruckt (Tab. IV). Der Ausgangstext liegt in Form einer Folge von Zeilen willkürlicher Länge vor. Eine Leerzeile signalisiere das Ende eines Paragraphen.

Die dynamische Struktur des Formatierers liegt auf der Hand: Fortgesetztes Lesen der Eingabe-Textzeilen und seitenweiser Aufbau des formatierten Textes im Speicher. Sobald eine ganze Seite aufgebaut ist, sollte diese ausgegeben werden, so dass im Speicher Platz für eine neue Seite geschaf-

nächst erwähnen wir, dass der Parser gleichzeitig die Rolle des Dirigenten der Übersetzung einnimmt. Der Parser-Modul steuert als Hauptprogramm den dynamischen Ablauf (höchste Hierarchiestufe). Der bisher unerwähnte Modul *Data* zeigt eine interessante Spielart des Software-Moduls. Seine Aufgabe besteht nicht in der Ausführung irgendwelcher Operationen, sondern in der Präsentierung der gemeinsamen globalen Datentypen und Daten. Die Figur 2b zeigt das Zeitdiagramm einer Einphasenübersetzung und zum Vergleich die Figur 3 den Ablauf einer sogenannten Mehrphasenübersetzung. In der letzteren findet in jeder Phase eine bestimmte Transformation des Eingabestromes in einen Ausgabestrom statt. Der Eingabestrom der ersten Phase ist der Programmtext, der Ausgabestrom der letzten Phase ist die Folge der Maschinenbefehle. Die Themen der einzelnen Phasen entsprechen ziemlich genau den Themen der Module in Figur 2a. Im Gegensatz zur *statischen* Gliederung des Programms in Module handelt es sich bei der Mehrphasen-Compilation um eine *dynamische* Gliederung in Phasen. Zum Schluss dieses Abschnitts sei darauf hingewiesen, dass *CodeGenerator* der einzige wesentlich von der Zielmaschine abhängige Modul ist. Damit sind wir dem Ziel einer leichten Übertragbarkeit des Compilers auf neue Zielmaschinen ziemlich nahe gekommen.

```
Programmierung


Im vorangehenden Abschnitt haben wir gesehen,
dass ein Computer U durch Vorgabe eines Programmes P
auf eine bestimmte Anwendung zugeschnitten werden kann,
ja sogar zugeschnitten werden muss.
Im Laufe der Zeit stellte sich heraus, dass die Tätigkeit
des Programmierens bestimmten Gesetzen gehorcht und Gedankengänge erfordert,
die weitgehend unabhängig von der speziellen,
ins Auge gefassten Anwendung sind.



Als Folge davon sind im Laufe der Zeit Regeln, Methoden
und Techniken entstanden,
welche die Wissenschaft der Programmierung an sich begründeten.
Das Programmieren im allgemeinsten Sinne hat sich geradezu
zum Kern der Informationsverarbeitung oder Informatik entwickelt.
Was also ist Programmieren?
Wir haben ein Programm bereits als eine Folge von Befehlen erklärt,
die auf eine bestimmte Menge von Daten wirkt.



Natürlich verbindet sich mit dem Begriff Programm die Vorstellung
seiner Ausführung. Tatsächlich läuft Programmieren darauf hinaus,
einen dynamischen Prozess als statischen Text zu formulieren.
Die Dinge werden jedoch noch komplizierter.
Im allgemeinen erwartet ein Programm Eingabedaten oder Parameter,
von welchen seine Ausführung abhängt.
(Das Universalprogramm U beispielsweise erwartet ein Programm P als Parameter.)
Deshalb beschreibt ein Programmtext im allgemeinen nicht nur einen,
sondern eine ganze Klasse von Prozessen.
Ein Programm als korrekt zu bezeichnen bedeutet offensichtlich,
dass alle diese Prozesse korrekt ablaufen,
d.h. (in endlicher Zeit) die korrekten Resultate erzeugen.
Korrekte Programme zu schreiben ist mehr als nur ein edles Ziel,
falls diese Programme zur Steuerung von Flugzeugen
oder Atomkraftwerken vorgesehen sind.



In den meisten Fällen ist das "Auffächern" eines Programmes in alle Prozesse,
die es beschreibt, hoffnungslos kompliziert.
Eine vielversprechendere Methode zum Beweis der Korrektheit
sind Absicherungen im (statischen) Programmtext selbst.
Programmieren in diesem rigorosen Sinne ist
eine hochgradig mathematische Tätigkeit.
Interessanterweise sind die Rollen im Laufe der Zeit vertauscht worden:
die Mathematik ist zu einem Instrument der Informatik geworden.
```

**Tabelle III. Beispiel: Unformatierter Text**

Programmierung

Im vorangehenden Abschnitt haben wir gesehen, dass ein Computer U durch Vorgabe eines Programmes P auf eine bestimmte Anwendung zugeschnitten werden kann, ja sogar zugeschnitten werden muss. Im Laufe der Zeit stellte sich heraus, dass die Tätigkeit des Programmierens bestimmten Gesetzen gehorcht und Gedankengänge erfordert, die weitgehend unabhängig von der speziellen, ins Auge gefassten Anwendung sind.

Als Folge davon sind im Laufe der Zeit Regeln, Methoden und Techniken entstanden, welche die Wissenschaft der Programmierung an sich begründeten. Das Programmieren im allgemeinsten Sinne hat sich geradezu zum Kern der Informationsverarbeitung oder Informatik entwickelt. Was also ist Programmieren? Wir haben ein Programm bereits als eine Folge von Befehlen erklärt, die auf eine bestimmte Menge von Daten wirkt.

Natürlich verbindet sich mit dem Begriff Programm die Vorstellung seiner Ausführung. Tatsächlich läuft Programmieren darauf hinaus, einen dynamischen Prozess als statischen Text zu formulieren. Die Dinge werden jedoch noch komplizierter. Im allgemeinen erwartet ein Programm Eingabedaten oder Parameter, von welchen seine Ausführung abhängt. (Das Universalprogramm U beispielsweise erwartet ein Programm P als Parameter.) Deshalb beschreibt ein Programmtext im allgemeinen nicht nur einen, sondern eine ganze Klasse von Prozessen. Ein Programm als korrekt zu bezeichnen bedeutet offensichtlich, dass alle diese Prozesse korrekt ablaufen, d.h. (in endlicher Zeit) die korrekten Resultate erzeugen. Korrekte Programme zu schreiben ist mehr als nur ein edles Ziel, falls diese Programme zur Steuerung von Flugzeugen oder Atomkraftwerken vorgesehen sind.

In den meisten Fällen ist das "Auffächern" eines Programmes in alle Prozesse, die es beschreibt, hoffnungslos kompliziert. Eine vielversprechendere Methode zum Beweis der Korrektheit sind Absicherungen im (statischen) Programmtext selbst. Programmieren in diesem rigorosen Sinne ist eine hochgradig mathematische Tätigkeit. Interessanterweise sind die Rollen im Laufe der Zeit vertauscht worden: die Mathematik ist zu einem Instrument der Informatik geworden.

**Tabelle IV. Beispiel: Formatierter Text**

fen wird. Natürlich laufen die Aktivitäten «Lesen einer Zeile» und «Schreiben einer Seite» asynchron ab.

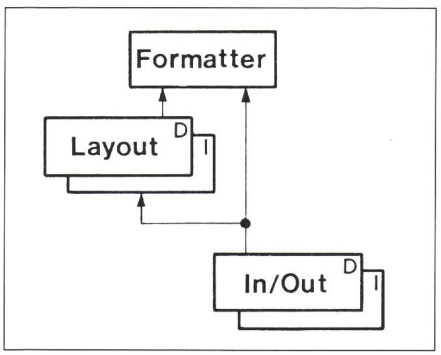
Als funktionale Einheit bietet sich das Auslegen und Drucken einer Seite an. Wir ordnen ihr den Modul *Layouter* (Fig. 4) zu. Wie sieht die Schnittstelle dieses Moduls aus? Im wesentlichen besteht sie aus einer einzigen Operation, genannt *Layout*, welche eine geeignete Texteinheit übernimmt und auslegt. Als Texteinheit bietet sich natürlich der *Paragraph* an. Damit ist die ursprüngliche Aufgabe auf das paragraphenweise Lesen des Eingabetextes und Übergeben an den Modul *Layouter* reduziert. Diese reduzierte Aufgabe überlassen wir dem Hauptmodul *Formatter*. Damit besteht unser System bis jetzt aus zwei Modulen, von denen das eine, *Formatter*, das anderen, *Layout*, importiert. Ferner verwenden wir das Bibliotheksmodul InOut, dem wir bereits früher begegnet sind. Es wird von *Formatter* für die Eingabe und von *Layout* für die Ausgabe importiert.

Bevor wir mit der eigentlichen Programmierung beginnen, legen wir die Datenstrukturen der beiden Module fest. Das Hauptmodul muss sicher einen Zeichenpuffer zur Aufnahme eines Paragraphen enthalten. Wir nennen ihn *buf*. Sein Füllzustand wird durch den Positionszeiger *lim* markiert. Im Layoutmodul wird ein zweidimensionaler Array *page* zur Darstellung der formatierten Seite verwendet. Die Koordinaten *row* und *col* legen den Formatierzustand der Seite fest, d.h. sie geben die aktuelle Position des gerade behandelten Zeichens an.

Wir beginnen die Programmierung mit der Definition des Moduls *Layouter*. Bei genauer Überlegung stellen wir fest, dass die Schnittstelle neben der Prozedur *Layout* eine Prozedur *OutPage* zur Ausgabe der gerade formatierten Seite umfassen sollte. Normalerweise wird *OutPage* vom Layouter selbst aufgerufen, sobald eine ganze Seite ausgelegt ist. Beim Erreichen des Endes des Eingabe-Textes jedoch muss der Impuls zur Ausgabe der letzten, nur teilweise gefüllten Seite vom Hauptprogramm gegeben werden.

```
DEFINITION MODULE Layouter;
  PROCEDURE Layout
  (VAR buf: ARRAY OF CHAR;
  lim: INTEGER);
  PROCEDURE OutPage;
END Layouter
```

Man beachte die Verwendung eines dynamischen Arrays als Parameter.




**Figur 4. Beispiel Textformatierer**

Der Präsentation der Implementation stellen wir zum besseren Verständnis, aber auch zur allgemeinen Information, einige Punkte voran, in denen Modula syntaktisch oder semantisch von Pascal abweicht.

1. Die Verwendung von Konstanten-*Ausdrücken* ist erlaubt.
2. LOOP END ist ein Konstrukt zur Programmierung von Schleifen mit Ausgängen (EXIT) in beliebiger Anzahl und an beliebigen Stellen.
3. Die IF- und die WHILE-Anweisungen sind mit einem expliziten END-Symbol abgeschlossen. Die Klammerung BEGIN END von Anweisungsfolgen erübrigt sich deshalb.
4. Die IF-Anweisung enthält in ihrer allgemeinsten Form eine beliebige Anzahl ELSIF-Teile.

   *Beispiel:*

   ```
   IF sym = comma THEN GetSym
   ELSIF sym = ident THEN err(11)
   ELSIF sym = var THEN err(11); GetSym
   ELSE EXIT
   END
   ```

5. INC und DEC sind Standardfunktionen zur schnellen Inkrementierung und Dekrementierung.
6. Das Zeichen «#» bedeutet «nicht gleich».
7. Boolesche Ausdrücke werden *bedingt* ausgewertet. Ihre exakte Definition ist:

   ```
   a OR b = IF a THEN TRUE ELSE b END
   a & b = IF a THEN b ELSE FALSE END
   ```

   *Beispiel:*

   Das folgende Programm sucht das Element x im Array a[0], ..., a[N-1]:

   ```
   i: = 0;
   WHILE (i # N) & (a[i] # x) DO INC(i) END
   ```

8. Die Anweisungsfolge zwischen BEGIN und END eines Implementationsmoduls wird nach dem Laden des Moduls automatisch abgearbeitet. Sie dient der Initialisierung der globalen Modulvariablen.

Bei der Implementation des Textformatierprogrammes empfiehlt es sich, mit dem Hauptmodul «Formatter» zu beginnen, da es die gesamte Ablaufstruktur widerspiegelt. In Tabelle V ist das vollständige Programm dargestellt und kommentiert. Zum besseren Verständnis seien noch folgende Erklärungen zum Modul InOut beigefügt:

Die in *InOut* enthaltenen Prozeduren *OpenInput* und *OpenOutput* bieten die interessante Möglichkeit, die Quelle der Eingabe und das Ziel der Ausga-

be zur Laufzeit des Programmes interaktiv anzugeben. Alle *Read*-Operationen beziehen sich auf die angegebene Quelle und alle *Write*-Operationen auf das angegebene Ziel. *Done* ist eine Boolesche Variable, die den Erfolg der letzten Eingabe- oder Ausgabeoperation anzeigt.

## 5. Schlusswort

Modula ist eine moderne Programmiersprache zur Konstruktion von grossen modularen Softwaresystemen. Als Strukturelement ist das Modul in seiner Relevanz durchaus mit der Prozedur und dem damit verbundenen Lokalitätsprinzip vergleichbar. Die separate Übersetzung ist gleichzeitig Garantie für effizienten Unterhalt und dauernde Konsistenz des Gesamtsystems. Obwohl einfach in seiner Konzeption, hat sich das Modul als ein überraschend schwierig zu beherrschendes Hilfsmittel herausgestellt. Es tritt in vielfältiger Gestalt auf, z. B. als Package von Prozeduren, als Verwaltungsinstanz von Ressourcen, als Schnittstelle zur Hardware, als Black Box zur Verarbeitung von Objekten, als Monitor und als Datenbasis. Das modulare Denken eröffnet neue Horizonte und führt zu weitreichenden und oft unerwarteten Konsequenzen, sogar für Betriebssystem-Designer und Computer-Architekten. Beispielsweise hat die Modulorganisation bereits ihren Niederschlag in neueren Mikroprozessor-Architekturen gefunden, so etwa in der Prozessor-Familie 32000 von National Semiconductor.

**Tabelle V**
**Textformatierprogramm**

```
MODULE Formatter;
   FROM Layouter IMPORT Layout, OutPage;          Importiere Prozeduren Layout und OutPage
   FROM InOut IMPORT OpenInput, OpenOutput,       Importiere Prozeduren zur Ein- und Ausgabe
                     CloseInput, CloseOutput,
                     Read, Done;
   CONST EOL = 36C;                               36C ist ASCII-Code des Zeilenabschlusses
   VAR ch: CHAR; lim: INTEGER;                    Hilfsvariablen
      buf: ARRAY [0..1000] OF CHAR;               Puffer für den nächsten Textparagraphen
BEGIN
   OpenInput("TXT"); OpenOutput("DOK");           Eröffne Ein- und Ausgabefiles
   lim := 0; Read(ch);                            Initialisierung
   WHILE Done DO                                  Solange ein Zeichen gelesen wurde
      WHILE ch # EOL DO                           Fülle nächste Zeile in den Puffer
         INC(lim); buf[lim] := ch; Read(ch)
      END;
      INC(lim); buf[lim] := " "; Read(ch);
      IF ch = EOL THEN                            Falls zusätzliche Leerzeile
         Layout(buf, lim); lim := 0; Read(ch)     Aktiviere Paragraph-Layouter
      END
   END;
   OutPage;                                       Forciere Ausgabe der letzten Seite
   CloseOutput; CloseInput                        Schliesse Ein- und Ausgabefiles
END Formatter.


DEFINITION MODULE Layouter;

   PROCEDURE Layout (VAR buf: ARRAY OF CHAR; lim: INTEGER);
   PROCEDURE OutPage;

END Layouter.


IMPLEMENTATION MODULE Layouter;
   FROM InOut IMPORT Write;
   CONST
      EOL = 36C;                                  36C ist ASCII-Code des Zeilenabschlusses
      EOP = 14C;                                  14C ist ASCII-Code des Seitenabschlusses
      len = 30; wid = 33;                         Layout-Konstanten
      maxRow = 70; maxCol = 3*len;
      bndCol = maxCol - len;
   VAR row, col: INTEGER;                         Momentane Koordinaten
      page: ARRAY [0..maxRow-1], [0..maxCol-1] OF CHAR;Seitenpuffer

   PROCEDURE OutPage;                             Prozedur zur Ausgabe der Seite
      VAR c, r, p: INTEGER;
   BEGIN r := 0;
      WHILE r # maxRow DO c := 0;                 Schleife zur Ausgabe der Zeilen
         WHILE c # col DO p := 0;                 Schleife zur Ausgabe der Spalten
            WHILE p # len DO
               Write(page[r, c + p]); INC(p)
            END;
            WHILE p # wid DO Write(" "); INC(p) END;
            c := c + len                          Nächste Spalte
         END;
         IF r < row THEN p := 0;                  Ausgabe der letzten Spalte
            WHILE p # len DO
               Write(page[r, c + p]); INC(p)
            END
         END;
         Write(EOL);
         INC(r)                                   Nächste Zeile
      END;
      Write(EOP)                                  Seitenende
   END OutPage;

   PROCEDURE Layout (VAR buf: ARRAY OF CHAR; lim: INTEGER);Formatier-Prozedur
      VAR beg, cur, end, pos, wds, spc, rem: INTEGER;
   BEGIN
      beg := 0; cur := 0;
      REPEAT INC(cur) UNTIL buf[cur] = " ";       Bestimme erstes Wort
      REPEAT wds := 0; INC(beg);                  Schleife zur Behandlung der Zeilen
         LOOP                                     Schleife zur Bestimmung der nächsten Zeile
            INC(wds); end := cur;                 Erhöhe Anzahl Wörter dieser Zeile
            IF end = lim THEN spc := 0; rem := 0; EXIT END;Paragraphenende erreicht
            REPEAT INC(cur) UNTIL buf[cur] = " "; Bestimme nächstes Wort
            IF cur - beg > len THEN               Falls Zeilenlänge überschritten
               IF wds > 1 THEN
                  spc := (len + beg - end) DIV (wds - 1);Berechne zusätzlichen Wortzwischenraum
                  rem := (len + beg - end) MOD (wds - 1)Berechne restliche Leerstellen
               END;
               EXIT                               Ende Bestimmung der Zeile
            END
         END;
         pos := 0;
         WHILE pos < len DO                       Initialisiere Zeile
            page[row, col + pos] := " "; INC(pos)
         END;
         pos := 0;
         WHILE beg # end DO                       Layout der Zeile im Seitenpuffer
            page[row, col + pos] := buf[beg];
            IF buf[beg] = " " THEN pos := pos + spc;  Wortzwischenraum
               IF rem > 0 THEN INC(pos); DEC(rem) ENDZusätzliches Leerzeichen
            END;
            INC(pos); INC(beg)
         END;
         INC(row);                                Nächste Zeile
         IF row = maxRow THEN                     Spaltenlänge erreicht
            IF col = bndCol THEN OutPage; col := 0  Seitenausgabe, nächste Seite
               ELSE col := col + len              Nächste Spalte
            END;
            row := 0
         END
      UNTIL end = lim
   END Layout;

BEGIN col := 0; row := 0
END Layouter.
```

**Literatur**


[1] *D. L. Parnas:* On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(1972)12, p. 1053...1058.

[2] *N. Wirth:* Programming in Modula-2 Third edition. - Texts and monographs in computer science - Berlin/Heidelberg, Springer-Verlag, 1985.