# 6 Aspekte der Programmierung im Großen

Wie durch die Überschrift unseres zweiten Kapitels und in dessen Einleitung bereits angedeutet, können die bisher in diesem Buch angeschnittenen Problemkreise und behandelten Methoden als einer "Programmierung im Kleinen" zugehörig angesehen werden. Sie betreffen die Überlegungen, welche ein Programmierer bei der Konstruktion kleiner Programme oder elementarer Programmteile (wie Prozeduren und Funktionsprozeduren) anstellen sollte, um jenen Qualitätsansprüchen gerecht zu werden, die wir im zweiten Kapitel formuliert haben.

Demgegenüber werden wir uns nun mit Fragestellungen beschäftigen, die der "Programmierung im Großen" zuzuordnen sind, einem Thema, welches bisher allenfalls nur leise anklang: Etwa in der Einleitung des Kapitels 3, wo "Schrittweise Verfeinerung" als eine Ausprägung der allgemeineren und auch "im Grossen" häufig favorisierten "Top-Down"-Entwurfstechniken bezeichnet wurde. Oder im letzten Abschnitt des fünften Kapitels, in welchem wir gesehen haben, daß bei der Auflösung von Strukturkonflikten ein "System" von im Prinzip eigenständigen, aber dennoch miteinander kooperierenden Prozessen beziehungsweise Programmen entstehen kann.

Damit ist uns das Stichwort für das vorliegende Kapitel gegeben: "Programmierung im Großen" wird es mit der Erstellung von Software*systemen* zu tun haben. Bevor wir einige der dabei zu bewältigenden Probleme benennen und einige der zur Lösung dieser Probleme anwendbaren Methoden einer eingehenderen Erörterung unterziehen können, ist es freilich notwendig, daß wir uns eine ungefähre Vorstellung vom Gegenstandsbereich "Softwaresystem" verschaffen.

In der Tat ist dieser Gegenstandsbereich so vielfältig "wie das Leben selbst". Überall dort, wo Rechner für mehr als nur triviale Anwendungen eingesetzt werden sollen, muß Software geschrieben werden, deren Funktionsumfang so groß ist, daß sie sich kaum als ein monolithisches Programm realisieren läßt. Die Beispiele hierfür sind Legion, und an dieser Stelle seien daher nur einige kurz erwähnt. Als *Betriebssysteme* etwa bezeichnen wir die Software, welche die bequeme und effiziente Nutzung der (Hardware-)Ressourcen eines Rechners ermöglicht. *Datenbank-Management-Systeme* bilden häufig die Grundlage für spezielle *Informationssysteme*, indem sie allgemeine Dienstleistungen zur Definition und Manipulation anwendungsspezifischer Daten zur Verfügung stellen. Die Kontrolle und Steuerung technischer Abläufe, sei es in einem Kraftwerk, an Bord eines modernen Verkehrsflugzeugs oder in einer Fabrikhalle wird mit Hilfe von Software vorgenommen, die unter dem Oberbegriff *Prozeßleitsysteme* einzuordnen ist. Da bei derartigen Anwendungen die in "realer" Zeit ablaufenden rechnerexternen Prozesse das Geschehen diktieren, spricht man hier auch von *Real-*

*zeitsystemen* (bzw. *Echtzeitsystemen*). Auf vielen Sekretariatsschreibtischen und an anderen Büroarbeitsplätzen findet man inzwischen leistungsfähige Kleinrechner, die die alltägliche Arbeit zum Beispiel durch komfortable *Textverarbeitungssysteme* erleichtern, und die miteinander über ein Lokales Netz verbunden sind, welches von einem *Datenkommunikationssystem* regiert wird. Und die Software-Produktion selbst wird zunehmend von sogenannten *Softwareentwicklungssystemen* unterstützt, welche im Idealfalle all jene "Werkzeuge" enthalten, mit denen das Produkt Software hergestellt und während seiner gesamten Lebensdauer verwaltet und bearbeitet werden kann.

Natürlich können wir in einem Buch, das allgemeine Methoden der Softwareentwicklung zum Thema hat, nicht auf die jeweiligen Besonderheiten einzelner Softwaresysteme eingehen. Vielmehr müssen wir uns fragen, welches die allen solchen Systemen gemeinsamen Eigenschaften sind, und was, unabhängig von dem gegebenen Anwendungsbereich, bei ihrer Konstruktion zu beachten ist.

Wir wollen die Erfahrung vermitteln, daß eine "Programmierung im Großen" mit den in den Kapiteln 3-5 dargelegten Konzepten und Methoden allein nicht zu leisten ist. In dem Gemeinplatz *"Das Ganze ist mehr als die Summe seiner Teile"* interpretieren wir das *"mehr"* als die *Beziehungen*, die zwischen den Teilen bestehen, und als die *Organisation* der Gesamtheit der Teile zu einer wirkungsvollen Einheit. Damit freilich können wir jene Analogien aus dem Bereich der industriellen Fertigung von Produkten weiterführen, die wir beispielsweise zur Erläuterung des "Datenstrukturierten Programm-Entwurfs" herangezogen haben. Wir können, und dieser Vergleich wird sich wie der sprichwörtliche "Rote Faden" durch dieses Kapitel ziehen, ein Softwaresystem als Unternehmen auffassen, welches vielfältige Produkte und/oder Dienstleistungen anbietet, an deren Herstellung im allgemeinen viele Mitarbeiter und Manager beteiligt sind, von denen sich womöglich einige ihrerseits zu mehr oder weniger selbständigen Subunternehmen zusammengeschlossen haben.

## 6.1 Softwaresysteme

Einige allgemeine Aussagen, die man über große Softwaresysteme machen kann, sind die folgenden:

- Sie sind das Ergebnis oft mehrjähriger Arbeit eines ganzen Teams von Entwicklern. Der Arbeitsaufwand, der in ihnen steckt, erreicht nicht selten die Grössenordnung von mehreren hundert *Personenjahren*. (Ein Personenjahr ist, vereinfachend gesagt, die Arbeitsleistung eines Menschen während eines Jahres.) Ihre Planung und Realisierung kann daher nur arbeitsteilig erfolgen.
- Sie sind niemals wirklich "fertig". Vielmehr unterliegen sie "im Laufe ihres Lebens" ständig irgendwelchen Modifikationen. Man sagt, sie müssen *gewartet* werden.

- Die Dokumentation, welche bei ihrer Entwicklung angefertigt wird, ist in der Regel so umfangreich, daß sie von einer einzelnen Person im Detail nicht zu überblicken ist.
- Es werden meist hohe Qualitätsanforderungen - etwa in Hinblick auf *Fehler-Toleranz* und *Zuverlässigkeit* - an sie gestellt. Dies gilt in besonderem Maße zum Beispiel für Betriebssysteme und Prozeßleitsysteme, deren Fehlverhalten durchaus katastrophale Folgen haben kann.

Aus diesen Eigenschaften ergibt sich, daß wir von einer Methodik der "Programmierung im Großen" weit mehr verlangen müssen als Anleitungen zur Auffindung isolierter Algorithmen oder zur "stilvollen" Gestaltung eines Programmtextes. Es kommt vielmehr offenbar darauf an, daß uns eine solche Methodik in die Lage versetzt,
- große Softwaresysteme so zu strukturieren, daß
- Projekte, die ihre Herstellung zum Ziel haben, effizient durchführbar sind,
- und daß ihre Qualität jederzeit überprüft und gesichert werden kann.

Nun ist wohl die allgemeinste Aussage, welche sich über ein beliebiges Stück Software wie auch über andere technische Produkte treffen läßt, die, *daß mit ihnen irgendwelche definierten Zwecke erfüllt werden sollen.* Und die Tatsache, daß wir hier Software und andere technische Produkte in einem Atemzug nennen, legt es nahe, die Gültigkeit einer weiteren, ebenfalls sehr allgemeinen Aussage zu vermuten: *Softwaresysteme bestehen - ebenso wie die meisten technischen Produkte - aus Teilen, die in zweckdienlicher Weise zusammenwirken.*

Mit diesen beiden Feststellungen werden zwei für die "Programmierung im Grossen" grundlegende Fragen aufgeworfen:

- *Was* soll ein Softwaresystem leisten? Und:
- *Wie* soll ein Softwaresystem aufgebaut sein, damit es die gewünschte Leistung erbringt?

Eine Antwort auf die WAS-Frage wird üblicherweise als *Spezifikation* oder *Definition* bezeichnet, während die Antwort auf die WIE-Frage durch *Entwurf* und *Implementierung* erbracht wird. Es mag auf den ersten Blick erstaunlich erscheinen, daß wir Spezifikation (als Aktivität) der "Programmierung im Großen" zurechnen. Auf den zweiten Blick, und unter Berücksichtigung der oben an eine entsprechende Methodik erhobenen Forderungen, wird jedoch klar, daß es erstens unmöglich ist, die Struktur eines Softwaresystems vollkommen unabhängig von seinem Zweck zu bestimmen, und daß zweitens die Sicherung eines wesentlichen Qualitätsmerkmals, der Anforderungserfüllung nämlich, auf eine sehr genaue Formulierung dieser Anforderungen angewiesen ist. (Dazu mehr im folgenden Abschnitt.)

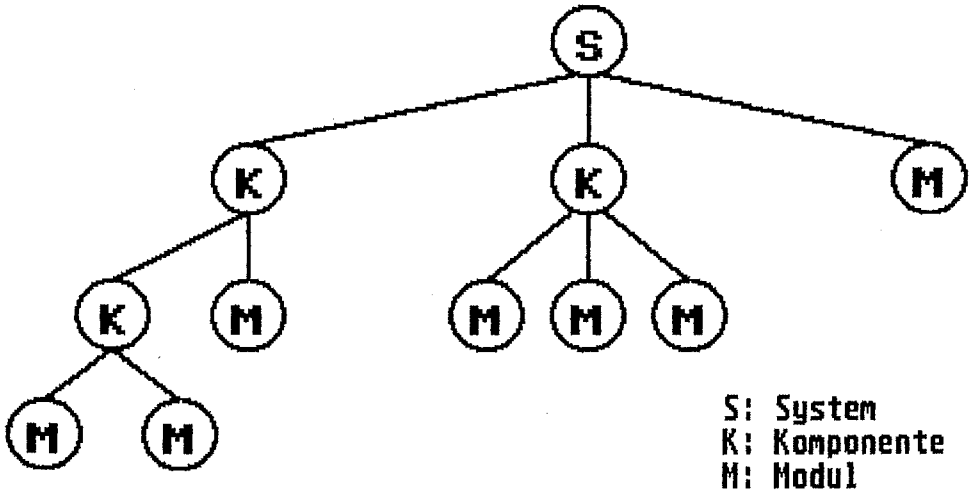
Die WAS-Frage kann natürlich niemals losgelöst von der speziellen Anwendungsumgebung beantwortet werden, in der die zu erstellende Software etwas

leisten soll. Vor einer solchen Antwort muß der Anwendungskontext also zunächst einer gründlichen *Analyse* unterzogen werden, mit dem Ziel, ein auf die wesentlichen Bestandteile und Zusammenhänge reduziertes *Modell* der (z.B.) betrieblichen oder maschinellen Umgebung zu erhalten, in der die geplante Software wirken oder - wie man auch sagt - eingebettet werden soll. Modellbildung ist daher eine wichtige Voraussetzung für die exakte Spezifikation eines Softwaresystems. In Abschnitt 6.3 werden wir auf dieses Thema detaillierter eingehen.

Entwurf und Implementierung müssen mit Rücksicht auf die im konkreten Fall vorhandene Basis-Software und -Hardware ausgeführt werden. Diese Basis kann "hoch" oder "niedrig" sein. Eine hohe Basis ist zum Beispiel ein Datenbank-Management-System, dessen Dienste man über eine komfortable Datenmanipulationssprache in Anspruch nehmen kann (s.o.). Eine sehr niedrige Basis ist die "nackte" Hardware. Mit einer solch niedrigen Basis hat man es bei Softwaresystemen, die für sogenannte "kommerzielle Anwendungen" (z.B. Lohnabrechnung, Lagerverwaltung, Bestellwesen, etc.) hergestellt werden, nur sehr selten zu tun. Vielmehr kann hier der Software-Entwickler bei Entwurf und Implementierung in der Regel eine *Basis-Maschine* voraussetzen, welche in einfacher Weise anzufordernde Dienstleistungen anbietet. Diese Basis-Maschine stellt sich ihm häufig als höhere Programmiersprache mit vielen vorgefertigten Prozeduren dar, oder etwa als ein Satz von "Makros", die von einem Assembler verstanden werden. Für Hardware-nahe Softwaresysteme ist die Situation völlig anders. Hier besteht eine wichtige Aufgabe des Entwurfs gerade darin, das Niveau der Basis-Maschine sukzessive anzuheben.

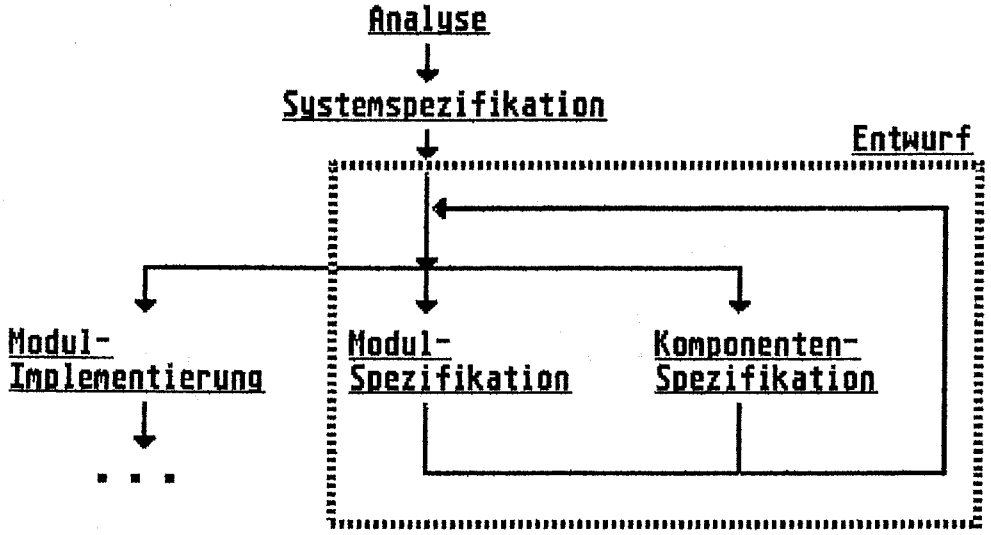
Dies bedeutet, daß der Entwickler nun seinerseits *Dienstleistungspakete* schnüren muß, auf denen er dann jeweils weiterbauen kann. Oftmals wird er dies nicht unmittelbar tun, sondern - ausgehend von der Systemspezifikation - zunächst *Subsysteme* oder - wie wir sagen werden - *Systemkomponenten* als Bestandteile einer *Systemzerlegung* definieren. (Nach der hier gewählten Sprechweise könnte man die Dienstleistungspakete auch als *Software-Bausteine* charakterisieren. Wir werden hierfür - und vorläufig noch ganz informell - den Terminus *Modul* verwenden. Dieser Begriff ist der Elektrotechnik entlehnt, wo er aus einzelnen Elementen zusammengesetzte Baugruppen bezeichnet.) In jedem Falle, also entweder bezogen auf einzelne Module oder auf Systemkomponenten, hat der mit der Ausführung eines Entwurfs betraute Entwickler die Aufgabe, das "nach außen sichtbare" Verhalten eines Teils des ganzen Systems zu spezifizieren.

In allgemeinster Form kann die Struktur eines Softwaresystems - nach dem eben Gesagten - "in erster Näherung" als Baum dargestellt werden, dessen Wurzel das System als Ganzes repräsentiert, während die internen Knoten die Komponenten und die Endknoten die Module abbilden:




In diesem Bild der *statischen* Struktur eines Softwaresystems wird lediglich die "Zerlegungsbeziehung" sichtbar: Ein Systemteil wird in zwei oder mehrere kleinere Teile zerlegt. Die Module sind die Endprodukte dieses Zerlegungsprozesses. Wir werden später sehen, daß es noch andere interessante Beziehungen zwischen Systemteilen (und insbesondere zwischen Modulen) gibt, die nur mit Hilfe allgemeinerer Graphen darstellbar sind.

Den Zusammenhang zwischen Analyse, Systemspezifikation und den aufeinanderfolgenden Entwurfsschritten zeigt die nachstehende Abbildung:




Der Leser möge sich an dieser Stelle an die Ausführungen in Abschnitt 3.1 erinnern, in dem wir ein ähnliches Bild zur Erläuterung der "Schrittweisen Verfeinerung" als einer Technik der "Programmierung im Kleinen" gezeichnet haben.

Dort kam es darauf an, zur Realisierung einer gewünschten Funktion die Existenz geeigneter abstrakter Teilfunktionen zunächst zu postulieren und dann deren Realisierung in Angriff zu nehmen. Die obige Abbildung veranschaulicht die Übertragung dieser Vorgehensweise auf die "Programmierung im Großen". Anstelle abstrakter Teilfunktionen stehen hier Spezifikationen von Komponenten und Modulen, also von Systemteilen, von denen jeder für bestimmte Aufgaben "verantwortlich" ist, beziehungsweise im allgemeinen mehrere (in einem noch zu präzisierenden Sinne) miteinander verwandte Dienstleistungen erbringt. Gemäß unserer Interpretation von Spezifikation ist das Ergebnis dieser Tätigkeit ebenfalls ein Abstraktum, nämlich eine von ihrer konkreten Implementierung unabhängige Beschreibung dieser Dienstleistungen.

Was aber meinen wir mit *Implementierung*, einem Begriff, den wir bereits mehrmals ohne weitere Erklärung verwendet haben? An der entsprechenden Stelle der entsprechenden Abbildung in Abschnitt 3.1 stand "Konstruktion aus elementaren Teilen". Von dieser, für unsere zukünftigen Zwecke etwas zu engen Auslegung werden wir im folgenden allerdings abrücken. Zu einem besseren Verständnis verhilft uns ein Blick in den "Großen Duden", ein sechsbändiges Wörterbuch der deutschen Sprache ([DGD]). Dort finden wir für das Wort "Implement" unter anderem die Bedeutung "Erfüllung (z.B. eines Vertrages)". In der Tat: Mit der Implementierung "erfüllen" wir das mit der Spezifikation gegebene Versprechen, bestimmte Dienstleistungen zu erbringen. Wir tun dies, indem wir entweder auf die (elementaren) Dienste der zugrundeliegenden Basis-Maschine zurückgreifen oder von bereits implementierten "höheren" Dienstleistungen Gebrauch machen. Auf diese Weise errichten wir - bildlich gesprochen - ein Gebilde aus aufeinander "geschichteten" Modulen. Während also die dominierende Entwurfsaktivität ganz allgemein als *Zerlegung* bezeichnet werden kann, bietet sich *Aufbau* als generelle Charakterisierung des Implementierens an. (Dabei sei der Leser freilich gewarnt, daß eine solche, die beiden Begriffe scharf trennende Charakterisierung "cum grano salis" zu genießen ist: Gerade unter dem Paradigma der "Objektorientierung", das wir im Abschnitt 6.4.4 behandeln werden, sind "Entwurf" und "Implementierung" sehr eng miteinander verflochten.)

Nach welchen Prinzipien und Gesichtspunkten man geeignete Systemzerlegungen (und System*schichtungen*!) finden kann, wie Module klassifiziert werden können und an welche Grenzen eine der "Schrittweisen Verfeinerung" nachempfundene "Top-Down"- Entwurfstechnik bei der Anwendung auf Softwaresysteme stößt, werden wir in Abschnitt 6.4 eingehend erörtern.

Zunächst aber werden wir in Abschnitt 6.2 einige Qualitätsmerkmale diskutieren, die Softwaresystemen insgesamt anhaften sollten. Welcher Wert dabei im einzelnen auf die jeweiligen Qualitäten zu legen ist, hängt natürlich von dem speziellen Anwendungsgebiet ab. Wie im Falle "greifbarer" Produkte wird auch die Qualität von Software nicht zuletzt durch die Organisation des Herstellungs-

prozesses bestimmt und davon, in welchem Maße bei der Herstellung geeignete Verfahren und Werkzeuge zum Einsatz kommen. Durch die damit zusammenhängenden Fragen werden wir zum Begriff *Software Engineering* geführt, der sich beinahe als ein Synonym für *Programmierung im Großen* erweisen wird.

## 6.2 Qualität im Großen und Software Engineering

In Kapitel 2 haben wir für die "Programmierung im Kleinen" aus der allgemeinen Forderung nach der *Brauchbarkeit* eines "elementaren Systemteiles" vier Qualitätsmerkmale hergeleitet: *Korrektheit*, *Integrierbarkeit*, *Wartbarkeit* und *Effizienz*. Auch für große Softwaresysteme bietet sich der Begriff *Brauchbarkeit* als Ausgangspunkt einer Erörterung spezieller wünschenswerter Eigenschaften solcher Systeme an. Die Fragen, die wir zu stellen haben, sind:

(1) Hat die Forderung nach der Brauchbarkeit eines Softwaresystems die gleichen Aspekte wie im Falle der "Programmierung im Kleinen", oder müssen die in Kapitel 2 besprochenen Qualitätsmerkmale in geeigneter Weise uminterpretiert und eventuell sogar umbenannt werden?

(2) Muß aus der Forderung nach Brauchbarkeit im Zusammenhang mit (großen) Softwaresystemen die Forderung nach weiteren Eigenschaften gefolgert werden, welchen wir bisher noch gar keine Beachtung geschenkt haben?
Und schließlich:

(3) Ist "Brauchbarkeit" als Oberbegriff überhaupt hinreichend, um alle notwendigen und wünschbaren Qualitäten großer Softwaresysteme zu subsumieren?

Zur Beantwortung der ersten Frage wenden wir uns der Reihe nach den vier oben erinnerten Merkmale zu:

### 6.2.1 Korrektheit und die Folgen

Wenn wir, wie im Abschnitt 2.1.1 herausgearbeitet, unter der Korrektheit eines Software-Produkts dessen Eigenschaft verstehen, genau die durch eine Anforderungs-Spezifikation verlangten Aufgaben zu erledigen, so müssen wir der Sicherung dieses Qualitätsmerkmals wohl auch in Bezug auf Softwaresysteme die oberste Priorität einräumen. Leider ist dieses Thema für große Softwaresysteme noch heikler, als wir dies bei der "Programmierung im Kleinen" verdeutlichen konnten. Die Erfahrung zeigt, daß umfangreiche Software-Produkte in der Praxis kaum so umfassend, detailliert und eindeutig spezifiziert werden, wie es nötig wäre, um "Anforderungserfüllung" zu einem nachprüfbaren Kriterium zu erheben. Natürlich darf uns dieses Defizit nicht davon abhalten, nach immer exakter formalisierten und damit besser (vielleicht sogar automatisch) überprüfbaren Formulierungen der Anforderungen an ein Softwaresystem zu streben, im Gegenteil, es muß uns dazu anspornen. Einen für eine spezielle Klasse von Systembausteinen praktikablen Ansatz zur formalen Spezifikation, auf dessen Grundlage

die Korrektheit einer Implementierung mit mathematischer Strenge beweisbar ist, werden wir immerhin in Abschnitt 6.4.3 dieses Kapitels vorstellen. Und um die formale Beschreibung des nach außen sichtbaren Verhaltens eines Softwaresystems als Ganzes wird es in Abschnitt 6.3 gehen.

Einer Tatsache sollten wir uns jedoch immer bewußt sein: Selbst unter der Voraussetzung einer vollständigen und den beabsichtigten Funktionen und Leistungen sowie den gegebenen Randbedingungen adäquaten formalen Spezifikation ist nicht zu erwarten, daß ein großes Softwaresystem (mit den in Abschnitt 6.1 beschriebenen Charakteristika) völlig fehlerfrei ist. Zur Rechtfertigung einer solchen Behauptung bedient man sich gern eines Wahrscheinlichkeitsarguments. Angenommen die einzelnen Module (Bausteine) eines Systems werden von unterschiedlich qualifizierten (motivierten, ausgeschlafenen, etc.) Mitgliedern eines Teams unabhängig voneinander entwickelt. Dann ist die Wahrscheinlichkeit $p$, daß ein Modul die an ihn gestellten (und aus den für das System als Ganzes geltenden Anforderungen abgeleiteten) Anforderungen erfüllt und also fehlerfrei ist, sicherlich kleiner als 1. Sagen wir es sei $p = 0{,}99$ und es gebe insgesamt 100 Module im System. Die Wahrscheinlichkeit, daß irgendwo im System ein Fehler steckt, ist also $1 - p^{100} \approx 0{,}634$. Und diese Überlegung berücksichtigt noch nicht einmal diejenigen Fehler, welche sich bei der notwendigen Verbindung der einzelnen Module einstellen können.

Diese kritische Einschätzung von *Korrektheit* (im oben definierten Sinne) als Qualitätsmerkmal großer Softwaresysteme legt es nahe, es durch andere, sicherlich verwandte, aber jedenfalls realistischere Bewertungskriterien zu ersetzen. Wenn es schon unbillig scheint, völlige Fehlerfreiheit zu verlangen, so sollten wir - einmal angenommen wir wären in der Lage desjenigen, der eine ihm gelieferte Software anwenden möchte - doch darauf dringen, daß

(i) allenthalben vorhandene Fehler dann, wenn sie manifest werden, keine unkontrollierbaren Auswirkungen hervorrufen.

Und vom Standpunkt des Softwareherstellers aus gesehen ist es zweifellos von großem Nutzen, wenn

(ii) die einzelnen Komponenten und Module eines Systems unabhängig voneinander auf die Erfüllung der an sie gestellten Anforderungen getestet und

(iii) bekannt gewordene Fehler mit möglichst geringem Aufwand beseitigt werden können.

Die durch (i) umschriebene Eigenschaft wollen wir *Integrität* nennen. Die durch (ii) und (iii) zum Ausdruck gebrachten Eigenschaften sind offenbar eng verwandt: Wenn die Module eines Systems unabhängig voneinander getestet werden können, so wird man sich auch bei der Fehlerkorrektur auf jeweils einen Modul beschränken, es sei denn, es handelt sich um Fehler beim Zusammenspiel der Module. Die Ursache eines Fehlers der letztgenannten Art kann nur im *Ent-*

*wurf* des Softwaresystems zu suchen sein, also bei derjenigen Aktivität, durch welche die Systemzerlegung und das Zusammenspiel der Module festgelegt wurden. Ungeachtet dessen, ob wir es mit *Implementierungs-* oder mit *Entwurfs-*fehlern zu tun haben, wollen wir (ii) und (iii) unter dem Begriff der *Prüfbarkeit* zusammenfassen. Zur Erfüllung von (iii) im Falle eines Entwurfsfehlers - soviel dürfen wir schon getrost behaupten - ist es mit Sicherheit sehr hilfreich, wenn die Ergebnisse der Entwurfsaktivitäten als gesondertes Dokument vorliegen.

Große Beispiele aus dem Bereich der "Programmierung im Großen", welche die Eigenschaften *Prüfbarkeit* und *Integrität* illustrieren könnten, würden natürlich den uns gesetzten Rahmen sprengen. Nichtsdestotrotz glauben wir, daß es auch in kleinerem Maßstab möglich ist, eine konkrete Vorstellung davon zu geben, worum es hier im Prinzip geht. Hinsichtlich der *Prüfbarkeit* erinnern wir uns zunächst der im vierten Kapitel (Abschnitt 4.1.3) vorgeschlagenen Technik, die innerhalb eines Programmstücks oder einer Prozedur als Kommentare notierten Prädikate in tatsächlich ausführbaren Code umzusetzen. Dies betrifft Vorbedingungen, Invarianten und Nachbedingungen gleichermaßen. Die Nichterfüllung einer solchen Bedingung ist gleichbedeutend mit dem Vorhandensein eines Fehlers: Wenn die Vorbedingung einer Prozedur nicht erfüllt wird, dann kann dies etwa daran liegen, daß die beim Aufruf der Prozedur übergebenen Parameterwerte nicht korrekt sind. In diesem Falle ist davon auszugehen, daß ein Fehler im aufrufenden Programmteil steckt. Die gerufene Prozedur sollte sich gegen solche Fehler "verteidigen" dürfen. Betrachten wir zum Beispiel nochmals die Prozedur *Division* im Abschnitt 4.2.4, deren Bedingungskommentare nicht in zusätzlichen Code umgesetzt sind:

```
PROCEDURE Division((*IN*) x,y:INTEGER;
                   (*OUT*) VAR q,r:INTEGER);
BEGIN
   (* x>=0 AND y>0 *)                    |<-- Vorbedingung
   r:=x; q:=0;
   (* r>=0 AND x=q*y+r *)                |<-- Invariante
   WHILE r>=y DO
      r:=r-y; q:=q+1
      (* r>=0 AND x=q*y+r *)             |<-- Invariante
   END
   (* r>=0 AND y>r AND x=q*y+r *)        |<-- Nachbedingung
END Division;
```

Dann wird der Aufruf "Division(zaehler, nenner, quotient, rest)" mit (vom Aufrufer irrtümlich so gesetzten) "zaehler = 6" und "nenner = -3" offenbar in einem Desaster enden. (Der Leser möge dies begründen.) Wie gesagt: Die "Schuld" daran trägt der Aufrufer.

Nehmen wir nun andererseits die in Abschnitt 4.1.3 abgebildete Version der Divisionsprozedur. Sie unterscheidet sich von der obigen Version nur durch die Abbruchbedingung der WHILE-Schleife. In 4.1.3 lautet die entsprechende Zeile "WHILE r > y DO", und wir haben bereits früher festgestellt, daß bei dieser Modifikation ein Aufruf "Division(zaehler, nenner, quotient, rest)" mit "zaehler = 6" und "nenner = 3" dazu führt, daß die Nachbedingung nicht erfüllt wird. In diesem Falle ist offenbar die gerufene Prozedur "Division" (mit der inkorrekten Abbruchbedingung) die "Schuldige". Sie liefert ein mit der durch die Nachbedingung gegebenen Spezifikation nicht übereinstimmendes (also ein falsches!) Ergebnis ab! Es wäre daher ohne Zweifel wünschenswert, wenn ein Aufrufer die Möglichkeit hätte, sich vor Resultaten in acht zu nehmen, die den weiteren Verlauf seiner Arbeit stören könnten.

Wir halten fest: Die *Integrität* eines Softwaresystems hängt ganz wesentlich davon ab, in welchem Umfang sich "benutzende" (im Beispiel: rufende) und "benutzte" (gerufene) Programmteile (Prozeduren) vor den möglichen Fehlern des jeweils anderen schützen.

Die Divisionsprozedur (hier mit der inkorrekten Abbruchbedingung) zum Beispiel könnten wir - mit Hilfe der sie kommentierenden Prädikate - so "aufbohren", daß sie dem Aufrufer eine Mitteilung machen kann
- sowohl wenn dieser ihr fehlerhafte Parameterwerte übergeben hat, als auch
- wenn sie "sich verrechnet" hat:

```
PROCEDURE Division((*IN*) x,y:INTEGER;
                          (*OUT*) VAR q,r:INTEGER;
                          (*OUT*) VAR fehler: CARDINAL);
BEGIN
  fehler := 0;
  IF NOT ((x>=0) AND (y>0)) THEN
    (* Vorbedingung nicht erfüllt *)
    fehler := 1;
    RETURN
  END (* IF *);
  r:=x;
  q:=0;
  IF NOT ((r>=0) AND (x=q*y+r)) THEN
    (* Invariante nicht erfüllt *)
    fehler := 2;
    RETURN
  END (* IF *);
  WHILE r>y DO
    r:=r-y;
    q:=q+1
```

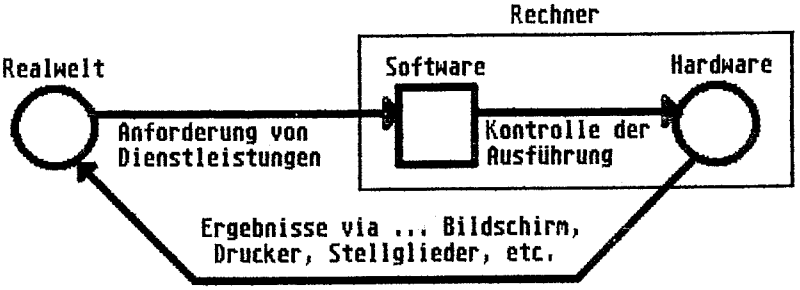
```
        IF NOT ((r>=0) AND (x=q*y+r)) THEN
          (* Invariante nicht erfüllt *)
          fehler := 2;
          RETURN
        END (* IF *)
      END (* WHILE *);
      IF NOT ((r>=0) AND (y>r) AND (x=q*y+r)) THEN
        (* Nachbedingung nicht erfüllt *)
        fehler := 3;
        RETURN
      END (* IF *);
    END Division;
```

Man sieht: *Integrität* hat ihren Preis, und zwar sowohl bezüglich der Programmgröße als auch der Ausführungsgeschwindigkeit! Es liegt nun in der Verantwortung des diese Prozedur benutzenden Programmteils, nach jedem Aufruf zu kontrollieren, ob über den VAR-Parameter "fehler" ein von 0 verschiedener Wert übergeben wird oder nicht. "fehler = 0" signalisiert, daß kein Fehler festgestellt wurde; "fehler = 1" besagt, daß der Aufrufer die Prozedur mit - gemäß Vorbedingung - unzulässigen Parameterwerten versorgt hat, und "fehler = 2" oder "fehler = 3" weisen auf einen Fehler innerhalb der gerufenen Prozedur hin. (Natürlich muß die Interpretation dieser Fehlercodes in der Benutzungsdokumentation der Prozedur enthalten sein! Vgl. Abschnitt 2.1.2.)

Man macht sich leicht klar, daß der soeben skizzierte Schutz einzelner Programmteile voreinander auch die *Prüfbarkeit* eines Programmsystems positiv beeinflußt, indem er die exakte Lokalisation von Fehlern ermöglicht. Einen weiteren Aspekt von *Integrität* werden wir später kennenlernen. Dieser betrifft die Vermeidung eines freien und unkontrollierten Zugriffs auf Ressourcen (Speicherbereiche, Variable) von verschiedenen Programmteilen aus. Daß dies ebenfalls zur besseren *Prüfbarkeit* beitragen kann, können wir aber schon jetzt verstehen: Denn wenn der Zugriff auf eine Ressource nur von einem gut überschaubaren Programmteil aus möglich ist und von diesem organisiert wird, dann werden irgendwelche Fehler, die bei diesem Zugriff gemacht werden, nur in jenem "zuständigen" Programmteil zu suchen sein.

*Integrität* und *Prüfbarkeit* sind gewissermaßen *innere* Eigenschaften eines Softwaresystems. Sie beziehen sich auf *interne* Fehler. Auf der anderen Seite wäre ein Softwaresystem aber ziemlich nutzlos, wenn es nicht auch allerlei *externen* Einflüssen ausgesetzt wäre. Zum Beispiel einem menschlichen Benutzer, der über eine Tastatur Eingaben macht und sich dabei nicht selten vertippt, oder einer Telekommunikationsleitung, die gelegentlich zusammenbricht, oder einem Drucker, der manchmal nicht angeschaltet ist und dem, wenn er angeschaltet ist, hin und wieder das Papier ausgeht, oder ..., oder ....

Wir dürfen nicht vergessen, daß Software nur eine Vermittlerrolle hat: Sie vermittelt zwischen einem Teil der realen Welt (einem *Realwelt-Ausschnitt*) und der Hardware des Rechnersystems. Sie soll die Hardware dazu veranlassen, diesem Realwelt-Ausschnitt irgendwelche nützlichen Dienste zu erbringen:




Wenn wir nun Rechner (bestehend aus Software und Hardware) und Realwelt als Teile eines Systems auffassen, so wird klar, daß sich der Begriff *Integrität* ohne weiteres auch auf die externen Beziehungen des Rechners übertragen läßt. Der Einfachheit halber, und um über eine praktische Situation reden zu können, nehmen wir an, daß die Realwelt aus einem Menschen besteht, der über Tastatureingaben, Bildschirm und Drucker mit dem Rechner kommuniziert. Wie es sich (für das Verhältnis zwischen Mensch und technischem Gerät) geziemt, sehen wir den Menschen als den *benutzenden* und den Rechner als den *benutzten* Teil des oben abgebildeten Systems. Mutatis mutandis können wir dann sagen, daß sich Mensch und Maschine vor den Fehlern des jeweils anderen "Partners" schützen müssen, um *integer* zu bleiben. Und zwar muß die Maschine als der *benutzte* Teil dem Menschen (also dem *benutzenden* Teil) unter Angabe von Gründen den Dienst verweigern dürfen, falls dessen Dienstanforderung nicht den für die Kommunikation zwischen Mensch und Maschine getroffenen Vereinbarungen entspricht. Außerdem muß die Maschine den Menschen davon in Kenntnis setzen, wenn sie wegen interner Probleme eine angeforderte Dienstleistung nicht erbringen kann. Diese internen Probleme können - wie oben am Beispiel der Divisionsprozedur demonstriert - durch Softwarefehler bedingt sein; sie können aber auch - und dieser Fall ist mindestens ebenso prekär - durch funktionsuntüchtige Hardware verursacht werden.

Natürlich ist es auf der Seite der Maschine im wesentlichen eine Aufgabe der Software, den Menschen über jeweils aufgetretene Fehler oder abnormale Zustände im Bereich der Hardware zu informieren. Ein Softwaresystem, welches dies leistet, ohne dabei - nach Möglichkeit - seinen Dienst komplett zu versagen, wird als *robust* bezeichnet. Allgemein wollen wir *Robustheit* als jene Qualität definieren, die einem Softwaresystem zukommt, wenn es - als Ganzes - sowohl gegenüber der Hardware des Rechners als auch gegenüber der von ihm bedienten Realwelt *Integrität* besitzt. Es versteht sich aber von selbst, daß sich dieses Qualitätsziel nur dann voll erreichen läßt, wenn die Hardware so konstruiert ist, daß ihre Fehlerzustände erkennbar sind.

Zur beispielhaften Verdeutlichung des Begriffs *Robustheit* nehmen wir an, daß der Mensch ein Textverarbeitungssystem benutzt, welches auf einem Mikrorechner läuft, der mit einem Diskettenlaufwerk als Externspeichermedium und mit einem Drucker ausgerüstet ist. Das Textverarbeitungssystem möge neben vielen anderen die folgenden Dienstleistungen anbieten:

Markieren: Damit kann durch das Setzen einer Anfangs- und einer Endmarke ein Textabschnitt abgegrenzt werden.

Cut-und-Paste: Damit kann ein markierter Textabschnitt "herausgeschnitten" und an anderer Stelle eingefügt werden.

Drucken: Ausdrucken des gerade bearbeiteten Textes auf dem angeschlossenen Drucker.

Speichern: Abspeichern des gerade bearbeiteten Textes auf der angeschlossenen Diskettenstation.

Laut Benutzungshandbuch darf die Cut-und-Paste-Operation nicht ohne vorherige Anwendung der Operation Markieren ausgeführt werden. Vom menschlichen Benutzer ist jedoch nicht zu erwarten, daß er sich immer an diese Regel hält. Wenn er es nicht tut und eine Cut-und-Paste-Operation verlangt, ohne einen Textabschnitt markiert zu haben, dann darf das nicht zu einem "Zusammenbruch" des Systems führen. Vielmehr muß das System seinen Benutzer auf die Nichtausführbarkeit der Cut-und-Paste-Operation aufmerksam machen und im übrigen in einem "konsistenten Zustand" bleiben beziehungsweise dorthin zurückkehren. Dies wäre im vorliegenden Fall etwa der Zustand, in dem sich das System vor der Anforderung der "illegalen" Operation befand.

Entsprechend wäre zu verlangen, daß ein Drucken-Kommando im Falle, daß der Drucker gar nicht angeschaltet ist oder kein Papier enthält, das System nicht zum Stillstand bringt. Die Software sollte die von der Drucker-Hardware ausgehenden Betriebssignale (beziehungsweise deren Fehlen) so interpretieren, daß sie dem Benutzer eine, die jeweilige Situation beschreibende Meldung macht und ihm die Möglichkeit geben kann, nach einem geeigneten Eingriff (z.B. Anschalten oder Papier einlegen) einen neuerlichen Versuch zu unternehmen. Im Prinzip die gleichen Bemerkungen gelten für jedes mit dem eigentlichen Rechner (der Zentraleinheit) verbundene Peripheriegerät. (Der Leser möge dies am Beispiel des Speichern-Kommandos nachvollziehen.)

*Robustheit* und *Integrität* sind - wie wir gesehen haben - zwei Seiten einer Medaille, welche *Fehler-Toleranz* heißt. (Gelegentlich nennt man diese Medaille auch *Zuverlässigkeit.*) Für große Softwaresysteme scheint uns diese Eigenschaft in der Tat wichtiger zu sein, als völlige Fehlerfreiheit.

### 6.2.2 Integrierbarkeit und die Folgen

Auch den Begriff der *Integrierbarkeit*, welcher den zweiten im Rahmen einer "Programmierung im Kleinen" für wichtig erachteten Aspekt von *Brauchbarkeit* bezeichnet, werden wir im folgenden einer differenzierteren Analyse unterziehen müssen. In Abschnitt 2.1.2 begnügten wir uns damit, *Integrierbarkeit* mit der Leichtigkeit des Einbaus einzelner Prozeduren in einem größeren Programm gleichzusetzen. Als entscheidende Voraussetzung hierfür haben wir eine saubere Dokumentation des Zwecks und der Parameter einer Prozedur herausgestellt. Nun, bezogen auf (große) Softwaresysteme, erhält dieser Begriff eine neue, größere Dimension.

Zunächst einmal haben wir es in diesem Rahmen an Stelle von Prozeduren mit Modulen und Komponenten zu tun (s.o.). Welche weiteren Aspekte und welches Gewicht, so ist zu fragen, hat *Integrierbarkeit* für diese Systembausteine? Eine Antwort ergibt sich aus einer einfachen Überlegung: Der mit der Entwicklung großer Systeme verbundene Arbeitsaufwand ist um so geringer, je mehr bereits vorhandene, im Laufe der Entwicklung früherer Systeme entstandene Bausteine, sei es direkt oder mit kleinen Modifikationen, übernommen oder - um unseren Begriff zu gebrauchen - *integriert* werden können. Die *Integrierbarkeit* der Bausteine eines Softwaresystems in andere Systeme hat also einen nicht zu unterschätzenden *ökonomischen* Effekt: Er macht die Systementwicklung billiger! Den so interpretierten *Integrierbarkeits*-Begriff präzisieren wir als *Wiederverwendbarkeit.*

Daß es überhaupt Sinn macht, *Wiederverwendbarkeit* als Qualitätseigenschaft großer Softwaresysteme (beziehungsweise der Module und Komponenten solcher Systeme) so stark zu betonen, zeigt die Beobachtung, daß in der Praxis der Softwareentwicklung "das Rad" gar nicht so selten "neu erfunden wird". Immer wieder wird Programmcode geschrieben, um beispielsweise einen Datenbestand zu durchzusuchen oder um - etwa bei der direkten Implementierung paralleler Prozesse - Prozesse zu synchronisieren, um Daten über Kanäle zu schicken, und so weiter. (Die Folge dieser Beispiele ließe sich noch lange fortsetzen.)

Zweifellos ist dies eine Vergeudung von Arbeitszeit, wenn man bedenkt, daß jeweils zumindest ähnlicher Code schon früher produziert wurde, vielleicht für ein anderes Projekt, vielleicht von einem (früheren) Kollegen. Und es brächte unter Umständen großen Gewinn, wenn man sich dieses Codes für das gerade anstehende Problem bedienen könnte. Günstige Voraussetzungen hierfür werden wir im weiteren Verlauf dieses Kapitels noch ausführlicher diskutieren. (Eine wichtige Maßnahme bleibt natürlich die schon in Abschnitt 2.1.2 erwähnte "saubere" Dokumentation.)

*Wiederverwendbarkeit* ist im wesentlichen eine Eigenschaft von Teilen. Wie aber steht es mit dem Softwaresystem als Ganzes? Hat *Integrierbarkeit* auch hierfür eine Bedeutung? Die Antwort ist "Ja", und zwar in zweierlei Hinsicht.

Erstens kann die *Brauchbarkeit* bestimmter Softwareprodukte dann außerordentlich verbessert werden, wenn sie mit anderen Softwareprodukten "zusammenarbeiten" können, um so noch größere und mächtigere Systeme zu bilden. Diese Art von *Integrierbarkeit* (eines Systems mit anderen Systemen) wird allgemein als *Kompatibilität* bezeichnet. Zum Beispiel wird die Leistungsfähigkeit eines Textverarbeitungssystems ganz sicher beträchtlich erhöht, wenn es auf von einem Datenbanksystem verwaltete Adressen und sonstige Informationen (Rechnungsnummern, Rechnungsbeträge, etc.) zurückgreifen kann. Es kann dann automatisch oder halb-automatisch Rechnungen oder Mahnungen an Kunden verschicken. Ferner wird es der Autor eines technischen Berichts sehr schätzen, wenn sein Textverarbeitungssystem Graphiken oder sonstige Abbildungen, die mit einem entsprechenden Softwareprodukt erzeugt wurden, in den eigentlichen Text einbinden kann. Die im Sinne dieser Beispiele verstandene *Kompatibilität* (d.h. die *Verträglichkeit*) verschiedener Systeme miteinander wird häufig durch die Vereinbarung von *Standards* hinsichtlich der Darstellung der von den jeweiligen Systemen produzierten und nach außen abgelieferten Daten (der "Datenformate") erreicht.

Zweitens ist zu berücksichtigen, daß Software an Hardware gebunden ist. Und wenn es sich speziell um "Anwendungs-Software" handelt, so kann man zur Hardware getrost das dazugehörige Betriebssystem hinzurechnen. Die Möglichkeit, eine gegebene Software mit einer anderen Hardware zu *integrieren*, sie auf eine andere Hardware (oder allgemeiner: in eine andere "Betriebsumgebung") zu *übertragen*, als die ursprünglich zugrundeliegende, nennen wir *Portabilität*.

Selbstverständlich kann sich diese Möglichkeit nicht auf den für den jeweiligen Hardware-Prozessor spezifischen Programmcode beziehen. Von einem in der Maschinensprache des Prozessors XY geschriebenen Programm ist nicht zu erwarten, daß es auf einem Prozessor ganz anderen Typs ebenfalls läuft. Es fragt sich aber, ob die Verwendung einer höheren Programmiersprache (wie z.B. MODULA-2) *Portabilität* nicht schon quasi "automatisch" garantiert, wenn nur jeweils geeignete Übersetzungsprogramme (Compiler) vorhanden sind. Leider ist dies nicht notwendigerweise der Fall. Um dies einzusehen, betrachten wir zwei verschiedene Hardware-Prozessoren, XY und AB, sowie die zugehörigen MODULA-2 - Compiler MOCO-XY und MOCO-AB, welche die Sprache MODULA-2 auf den Maschinen XY beziehungsweise AB - wie man sagt - "implementieren". Im allgemeinen gilt nun:

$$\text{XY-MODULA-2} \neq \text{AB-MODULA-2}.$$

Das Ungleichheitszeichen bedeutet: Es gibt mindestens ein MODULA-2-Programm,

> welches von MOCO-XY oder MOCO-AB als syntaktisch korrekt akzeptiert wird und auf der Maschine XY bzw. AB auch gemäß seiner Spezifikation läuft, das aber auf der jeweils anderen Maschine

* entweder von dem dort laufenden Compiler als syntaktisch nicht korrekt zurückgewiesen wird,
* oder zwar akzeptiert wird, aber bei der Ausführung ein ganz anderes Verhalten zeigt.

Die Gründe für diese Diskrepanz liegen auf der Hand:

(i) Der Compiler MOCO-XY erlaubt die Verwendung von Sprachelementen (z.B. die Bezeichner von Modulen oder Prozeduren, die in irgendwelchen Bibliotheken gesammelt sind), welche MOCO-AB nicht kennt.

(ii) Die Compiler nutzen bei der Umsetzung gleicher Sprachelemente jeweils spezifische Eigenschaften ihrer Maschinen so aus, daß ein unterschiedlicher Effekt entsteht.

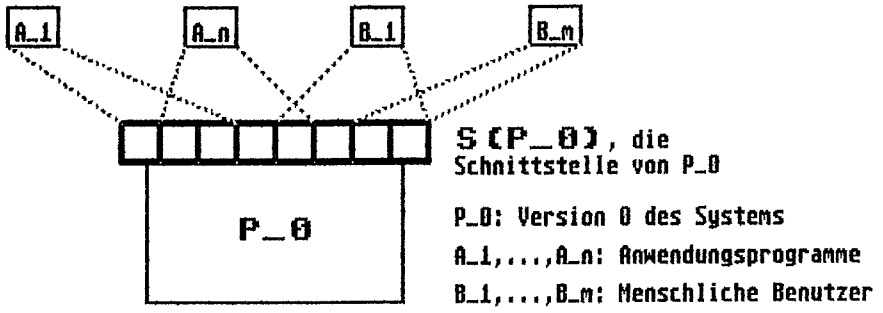
(Es ist klar, daß MODULA-2 bei dieser Argumentation durch jede beliebige andere höhere Programmiersprache ersetzbar ist.) Damit ist ein Problem angedeutet, das auch durch eine *Standardisierung* höherer Programmiersprachen nicht vollständig gelöst werden kann. Es würde in diesem Buch freilich viel zu weit führen, wenn wir diese Problematik vertieft behandelten. Bemerkt sei nur, daß neben einer allgemein verbindlichen Festlegung der Syntax einer Programmiersprache auch deren Semantik eindeutig definiert werden müßte, und zwar unabhängig von jeder zugrundeliegenden speziellen Hardware. Davon, daß dies keine leichte Aufgabe ist, haben wir in Kapitel 4 einen kleinen Eindruck bekommen.

Unterschiede im Sprachumfang und in der konkreten Übersetzung einzelner Sprachelemente bei verschiedenen Compilern sind nicht die einzigen Schwierigkeiten, die manchem *Portierungs*projekt entgegenstehen. Gerade wenn es sich um reine Anwendungssoftware handelt, also um - sagen wir - eine Kollektion aufeinander bezogener Programme, welche sämtlich auf den Dienstleistungen der mit der Hardware verbundenen Betriebssoftware aufbauen, gerade dann kann es um die *Portabilität* besonders schlecht bestellt sein. Das liegt daran, daß zwei verschiedene Betriebssysteme im allgemeinen nicht exakt den gleichen Satz von Operationen anbieten (nicht übereinstimmende Syntax!) und daß wechselseitig sich entsprechende Operationen beider Systeme dennoch nicht exakt in der gleichen Weise ausgeführt werden (nicht übereinstimmende Semantik!). Besonders lästig sind in diesem Zusammenhang nicht selten die Operationen zur Ein- und Ausgabe von Daten oder Signalen über periphere Geräte (Drucker, Lesegeräte, externe Speicher, etc.). Viele Anwendungsprogramme machen ferner ausgiebigen Gebrauch von dem im weiteren Sinne auch zur Betriebssoftware gehörigen Dateiverwaltungssystem eines Rechners. Dieser Komponente liegt jeweils eine - von Betriebssystem zu Betriebssystem - unterschiedliche Organisation der externen Speicher zugrunde. Auch deshalb können bei *Portierungen* aufwendige Umstrukturierungsmaßnahmen notwendig werden.

Aber selbst wenn eine Anwendungssoftware (wie oben charakterisiert) nur von einer früheren Version eines Betriebssystems auf eine durch irgendwelche Modifikationen entstandene neue Version des gleichen Betriebssystems übertragen werden soll, kann es zu Schwierigkeiten kommen. Die Erklärung dieser Tatsache gibt uns eine Gelegenheit, sowohl eine weitere Bedeutung des Begriffs *Kompatibilität* zu erkennen, als auch vom Begriff der *Schnittstelle*, den wir schon im Vorspann des Abschnitts 2.1 kurz erwähnten, eine plastischere Vorstellung zu gewinnen.

Wir behaupten nämlich: Nur wenn die neue Version des Betriebssystems mit der alten Version *kompatibel* ist, wird man "über" dem früheren System laufende Anwendungssoftware ohne Änderungen übernehmen können. Offenbar verwenden wir das Wort "kompatibel" dabei in einem ganz anderen Sinne als oben. Wir reden hier davon, daß ein Softwaresystem mit einer früheren Version seiner selbst *kompatibel* ist. Was kann das heißen? Man erinnere sich: Ein Softwaresystem soll die Hardware zu nützlichen Diensten antreiben, die entweder menschlichen Benutzern unmittelbar, anderen Programmen (welche Teile eines anderen Systems sein können) oder innerhalb einer technischen Installation zu leisten sind. Etwas weniger exakt formulierend können wir diese Aussage darauf verkürzen, daß "die Software eine Reihe von Dienstleistungen erbringt". (So haben wir oben schon davon gesprochen, daß Anwendungssoftware die Dienstleistungen des Betriebssystems und des Dateiverwaltungssystems eines Rechners benutzt.)

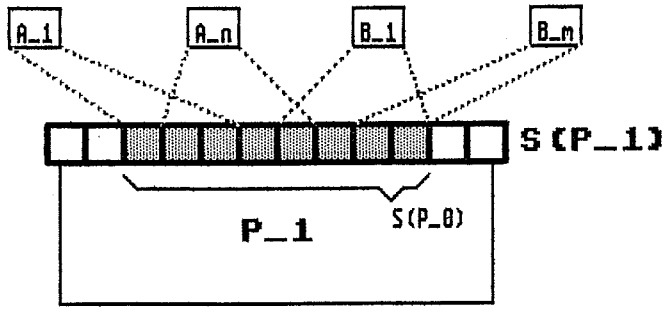
Die Gesamtheit dieser Dienstleistungen, die Art und Weise wie sie angeboten werden, sowie die Modalitäten ihrer Nutzung bilden zusammen die (*Benutzungs-*)*Schnittstelle* des Softwaresystems. Die folgende Abbildung zeigt die Version P_0 eines solchen Systems und seiner Schnittstelle:




Gründe, die es angeraten sein lassen, dieses Softwaresystem zu einer Version P_1 weiterzuentwickeln, können unter anderen sein:

- Steigerung der Effizienz,
- Implementierung zusätzlicher Dienstleistungen,
- Änderungen der Hardware-Konfiguration.

Die Benutzungsschnittstelle von P_1 sei S(P_1). Wir definieren: P_1 heißt *kompatibel* (genauer: *aufwärts-kompatibel*) mit P_0, falls S(P_1) ⊃ S(P_0).




Das heißt:

(i) Alle Anwendungsprogramme, welche P_0 benutzten, können ohne jede Änderung auch P_1 benutzen, und

(ii) alle menschlichen Benutzer von P_0 können auch P_1 benutzen, ohne irgendetwas hinzulernen zu müssen.

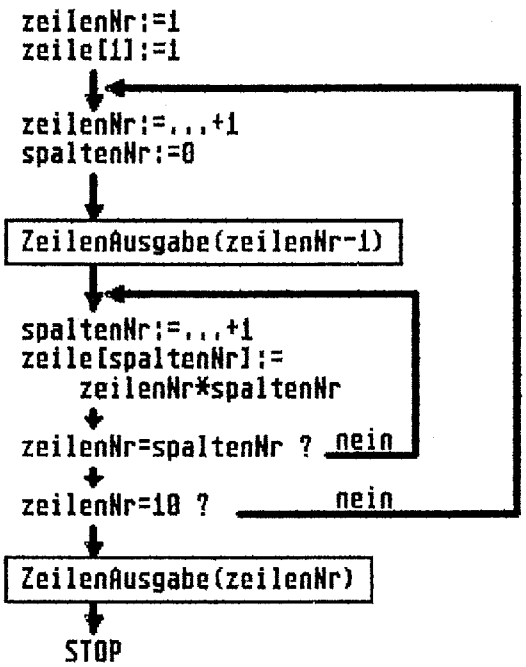
Es liegt auf der Hand, daß auch die Merkmale *Kompatibilität* (mit beiden Bedeutungen) und *Portabilität* von großem ökonomischen Nutzen sind, und zwar sowohl für den Hersteller als auch für den Anwender von Softwaresystemen. Je besser zum Beispiel die *Portabilität* eines Softwareproduktes ist, umso weniger Arbeitsaufwand ist nötig, um dieses Produkt für weitere, zur Zeit seiner Entwicklung vielleicht noch gar nicht verfügbare Rechner anbieten zu können. Der Anwender auf der anderen Seite wird sich freuen, wenn er ein einmal angeschafftes Softwaresystem zukünftig mit anderen Systemen in einfacher Weise kombinieren kann; und es wird ein geldwerter Vorteil für ihn sein, wenn frühere Anwendungsprogramme über der neuen Version eines Betriebssystems oder eines Datenbank-Management-Systems ohne Änderungen verwendbar bleiben.

### 6.2.3 Wartbarkeit und die Folgen

Die Vorstellungen, welche sich mit dem dritten der in Kapitel 2 angesprochenen Qualitätsmerkmale, der *Wartbarkeit,* im Kontext großer Softwaresysteme verbinden, haben durchaus starke Bezüge zur bisher in diesem Abschnitt geführten Diskussion. In 2.1.3 haben wir *Wartbarkeit* in der Praxis gleichgesetzt mit "guter Qualität der programm-internen Dokumentation", einer Dokumentation, aus der Aufbau und Funktion eines Programms klar hervorgehen sollen. Daran ist gewiß festzuhalten. Inzwischen haben wir allerdings eine ganze Reihe von Gründen näher kennengelernt, derentwegen Softwaresysteme "gewartet" werden müssen. Beispielsweise zur Beseitigung bekanntgewordener Fehler. Mit diesem Wartungsgrund haben wir uns oben bereits auseinandergesetzt und seinetwegen von einem Softwaresystem und seinen Teilen die Eigenschaft der *Prüfbarkeit* verlangt. Und ein Weg hierzu führt, wie wir sahen, in der Tat über die in Kapitel 4

studierte Form der internen Dokumentation. Ein zweite Gruppe von Gründen ist uns beim Übergang von einer Version eines Softwaresystems zur nächsten begegnet. Diese Gründe weisen jeweils auf besondere Aspekte von *Wartbarkeit* hin, welche ihrerseits Spezialisierungen des Qualitätsmerkmals *Adaptierbarkeit* darstellen. Dies ist keineswegs erstaunlich, ergibt sich doch die Notwendigkeit neuer Versionen im allgemeinen aus dem allfälligen Bedürfnis, ein Softwaresystem an veränderte externe Anforderungen *anzupassen. Adaptierbarkeit* ist also eine Art "Stetigkeitseigenschaft": Kleine Änderungen der externen Anforderungen (der Spezifikation!) sollten auch nur kleine Änderungen der bereits existierenden Software bedingen. (Damit schließen wir aus, daß Unmögliches verlangt wird: Es macht zum Beispiel sicherlich keinen Sinn, ein Textverarbeitungssystem zu einem Datenbanksystem weiterzuentwickeln. *Adaptierbarkeit* hat also ihre Grenzen. Das Problem, welches sich im Zusammenhang mit unserer Charakterisierung als "Stetigkeitseigenschaft" stellt, betrifft die Bestimmung eines "Masses", das es uns gestattet, von kleinen oder großen Änderungen zu sprechen. Mit Problemen dieser Art ist die sogenannte "Software-Metrik" befaßt, vgl. z.B. [HAS].) Zur Illustration dieses sehr wichtigen Gesichtspunkts bedienen wir uns des für den aktuellen Zweck abgewandelten Beispiels aus Kapitel 5, der "Multiplikationstabelle" (vgl. auch [JA1]):

An der diesem Beispiel zugrundeliegenden Aufgabenstellung wird sich zunächst nichts ändern: "Es ist ein Programm zu schreiben, welches die Ausgabe einer Multiplikationstabelle für das kleine Einmaleins bewirkt. Die Tabelle soll in unterer Dreiecksform auf dem Bildschirm erscheinen." Nehmen wir an, dem Lehrling im Software-Haus "ADAPTOR&Co." wird die Ausführung dieses Auftrages anvertraut. Wie es der "Stift" in der Schule gelernt hat, nimmt er ein Stück Papier, und nach einigem Nachdenken skizziert er, schon sehr routiniert auf Kästchen und Rauten weitgehend verzichtend, das nebenstehende "Flußdiagramm" als Programmentwurf.




Man beachte: Nachdem er von seinem Chef auf einen Kurs über "Strukturierte Programmierung" geschickt wurde, hat sich unser Lehrling redlich bemüht, nur als "ordentliche" Schleifen programmierbare Umlenkungen des Kontrollflusses vorzusehen; auch hat er sich daran erinnert, daß es sehr nützlich ist, wenn man gewisse allgemeine Aufgaben

- wie hier die "ZeilenAusgabe" - durch gesonderte Unterprogramme erledigen läßt.

Darüber, welche Gedanken ihm bei der Anfertigung der obigen Skizze sonst noch gekommen sein mögen, wollen wir uns hier nicht auslassen. Immerhin, er weiß, wie man ein Flußdiagramm in ein MODULA-2 - Programm umsetzt, und begibt sich sofort an die Kodierungsarbeit. Dabei überlegt er sich, daß es - zur besseren Übersicht - womöglich sinnvoll ist, die jeweils in den beiden Schleifen durchlaufenen Anweisungen ebenfalls zu Unterprogrammen zusammenzufassen. Dies ist sein Resultat:

```
MODULE EinMalEinsA;
FROM InOut IMPORT WriteCard, WriteLn;
VAR  zeilenNr, spaltenNr: CARDINAL;
       zeile: ARRAY [1..10] OF CARDINAL;
PROCEDURE ZeilenAusgabe(letztNr: CARDINAL);
VAR  i:CARDINAL;
BEGIN
  FOR i:=1 TO letztNr DO WriteCard(zeile[i],4) END;
  WriteLn
END ZeilenAusgabe;

PROCEDURE MacheEintrag;
BEGIN
  spaltenNr:=spaltenNr+1;
  zeile[spaltenNr]:=zeilenNr*spaltenNr
END MacheEintrag;

PROCEDURE MacheZeile;
BEGIN
  zeilenNr:=zeilenNr+1;
  spaltenNr:=0;
  ZeilenAusgabe(zeilenNr-1);
  REPEAT
    MacheEintrag
  UNTIL spaltenNr=zeilenNr
END MacheZeile;

BEGIN (* EinMalEinsA *)
  zeilenNr:=1;
  zeile[1]:=1;
  REPEAT
    MacheZeile
  UNTIL zeilenNr=10;
  ZeilenAusgabe(zeilenNr)
END EinMalEinsA.
```

Bei seinem Chef, der nur oberflächlich hinschaut (und dabei die zahlreichen Verstöße gegen unsere - in Kapitel 2 aufgestellten - Regeln guten Stils, geschweige denn die noch gravierenderen Mängel gar nicht wahrnimmt), findet dieses Programm zunächst Anklang, und der Auftraggeber ist für's erste auch zufrieden, da er die Multiplikationstabelle auf dem Bildschirm seines Rechners wie gewünscht zu sehen bekommt.

Nach einiger Zeit freilich gefällt unserem Kunden die Form, in der ihm die Tabelle präsentiert wird, nicht mehr. Anstatt in unterer möchte er sie nun in oberer Dreiecksform haben. Er wendet sich also an ADAPTOR&Co. und bittet um eine entsprechende Änderung, in der Erwartung, daß diese ja keinen großen Aufwand machen wird. Außerdem weist er den Chef des Software-Hauses darauf hin, daß auch in Zukunft mit Modifikations- und Erweiterungswünschen seinerseits zu rechnen sei. Zum Beispiel, so sagt er, könnte es sich als günstig erweisen, die Tabelle in Rechteckform zu bringen oder ihren Anwendungsbereich auf das große Einmaleins auszudehnen. Man solle sich schon jetzt darauf einstellen.

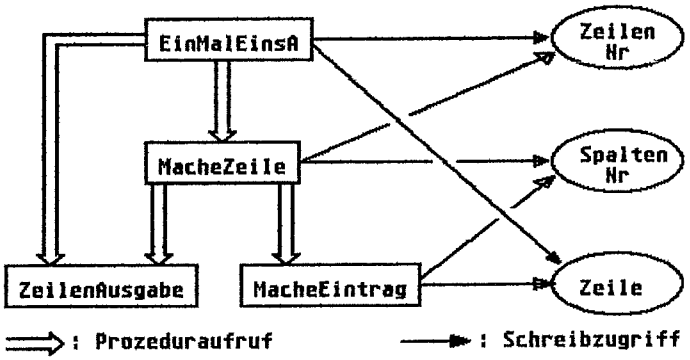
Natürlich wird wieder der Lehrling mit dem Änderungsauftrag betraut. Der Stift grübelt lange und kommt endlich zu dem Schluß, daß ein völlig neues Programm entworfen und geschrieben werden muß. Er teilt die Entscheidung seinem Chef mit. Dieser wird darob sehr mißtrauisch und läßt sich das ursprüngliche Programm zeigen. Abgesehen von schon erwähnten Stil-Mängeln (die zu lokalisieren auch der Leser aufgefordert ist) fallen ihm nunmehr - bei genauerer Betrachtung - noch weitere Merkwürdigkeiten und Ungereimtheiten auf. Unter anderem sind dies:

- Die Sonderbehandlung des ersten Tabellenelementes und die Tatsache, daß
- die Prozedur "ZeilenAusgabe" mit jeweils verschiedenen aktuellen Parameterausdrücken aufgerufen wird.

Sein besonderes Mißfallen aber erregt das Durcheinander beim Zugriff auf die global vereinbarten Variablen "zeilenNr", "spaltenNr" und "zeile":

- Veränderungen des Inhalts dieser Variablen werden aus jeweils mehreren Teilen des Programms vorgenommen.

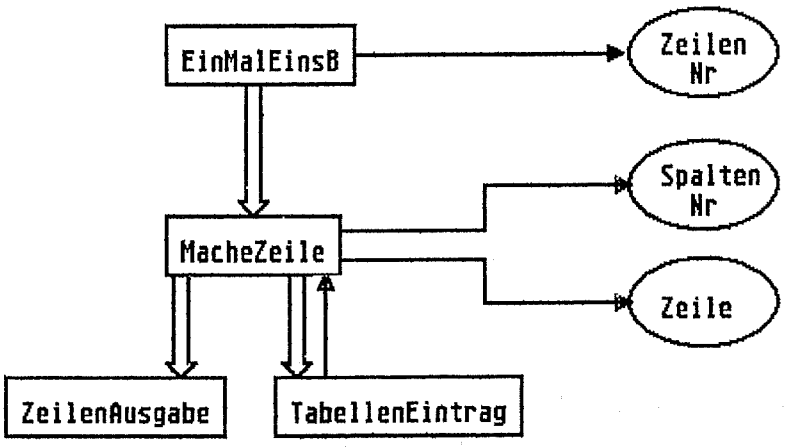
Mit einer einfachen Skizze (deren graphische Elemente selbsterklärend sind) macht er seinem Lehrling den Sachverhalt deutlich:




Und er empfiehlt ihm, beim nächsten Entwurf die folgenden Ratschläge zu beherzigen:

- Organisiere dein Programm wie einen arbeitsteiligen, hierarchisch gegliederten Produktionsbetrieb,
- dessen Abteilungen den Teilen des herzustellenden Produktes entsprechen:
  + Das Produkt Tabelle wird beim "Chef" (dem "Hauptprogramm") bestellt.
  + Dieser gibt Zeile für Zeile bei einem "Zeilenmacher" (der Prozedur "MacheZeile") in Auftrag.
  + Für die Produktion einer Zeile beansprucht der "Zeilenmacher" zunächst je Spalte die Dienste
    * eines "Rechenknechtes" (der Prozedur "TabellenEintrag") zur Berechnung eines Tabellenelementes, und danach die Dienste
    * eines "Zeilenausgebers" (der Prozedur "ZeilenAusgabe") zur Darstellung einer Zeile auf dem Ausgabemedium (zur Verpackung des Produktes gewissermaßen).

Die Aufruf- und Zugriffsstruktur des neuen Programms "EinMalEinsB" sollte demgemäß die folgende Gestalt haben:




(Dabei bedeutet der einfache Pfeil von "TabellenEintrag" zu "MacheZeile", daß "TabellenEintrag" Daten an "MacheZeile" liefert.)

Unter Beachtung einiger Stil-Regeln produziert unser ehrgeiziger und lernfähiger Auszubildender daraufhin sein zweites MODULA-2 - Programm:

```
MODULE EinMalEinsB;
FROM InOut IMPORT WriteCard, WriteString, WriteLn;
CONST TabgrC = 10; Blank4C = "    ";
TYPE  ZeilenTyp = ARRAY [1 . . TabgrC] OF CARDINAL;
VAR   zeilenNr, spaltenNr: CARDINAL;
      zeile: ZeilenTyp
PROCEDURE ZeilenAusgabe(erstNr, letztNr: CARDINAL);
VAR   i: CARDINAL;
```

```
BEGIN
  FOR i:=1 TO erstNr-1 DO WriteString(Blank4C) END;
  FOR i:=erstNr TO letztNr DO WriteCard(zeile[i],4) END;
  FOR i:=letztNr+1 TO TabgrC DO WriteString(Blank4C) END;
  WriteLn
END ZeilenAusgabe;

PROCEDURE TabellenEintrag: CARDINAL;
BEGIN
  RETURN zeilenNr*spaltenNr
END MacheEintrag;

PROCEDURE MacheZeile;
BEGIN
  FOR spaltenNr:=1 TO zeilenNr DO
    zeile[spaltenNr]:=TabellenEintrag
  END (*FOR*);
  ZeilenAusgabe(1,zeilenNr)
END MacheZeile;

BEGIN (* EinMalEinsB *)
  FOR zeilenNr:=1 TO TabgrC DO MacheZeile END
END EinMalEinsB.
```

Und präsentiert es voll Stolz seinem Chef, nicht ohne darauf hinzuweisen, daß

- alle Zeilen nun einheitlich behandelt werden,
- Schreibzugriffe auf die Variablen streng lokalisiert sind,
- die Prozedur "ZeilenAusgabe" verallgemeinert wurde und daß
- die Ausgabe einer Zeile dort veranlaßt wird, wo die Zeile hergestellt wird.

Er macht darauf aufmerksam, daß die vom Kunden vorgenommene Änderung der Spezifikation sich durch eine sehr geringe Modifikation einer einzigen Prozedur erfüllen läßt. In der Prozedur "MacheZeile" nämlich sind lediglich

- andere Grenzen der Laufanweisung und
- andere Parameter für die Zeilenausgabe

einzutragen:

```
PROCEDURE MacheZeile;
BEGIN
  FOR spaltenNr:=zeilenNr TO TabgrC DO MacheEintrag END;
  ZeilenAusgabe(zeilenNr, TabgrC)
END MacheZeile;
```

Ein Lob für diese Arbeit bleibt nicht aus. Doch dann bemerkt der Chef, daß seine Ratschläge offenbar doch noch nicht gründlich genug durchdacht und daher

auch nicht voll angewandt wurden. Wenn das Programm wie ein Produktionsbetrieb organisiert wird, dann sollte man auch bedenken, daß

- jede Abteilung des Betriebs (hier: jede Prozedur) ihre eigenen Ressourcen (hier: Speicher bzw. Variable) hat, für deren Verwendung sie allein verantwortlich ist (hier: auf die sie allein Schreibzugriff hat oder für die sie den Schreibzugriff delegieren kann), und daß daher
- keine Abteilung wissen sollte, wie die jeweils anderen Abteilungen ihre Aufgaben erledigen. Im Beispiel bedeutet dies insbesondere:
  + Der "Chef" (das Hauptprogramm) teilt dem "Zeilenmacher" (der Prozedur "MacheZeile") lediglich die Nummer der anzufertigenden Zeile mit.
  + Der "Zeilenmacher" gibt dem "Rechenknecht" (der Prozedur "Tabellen-Eintrag") lediglich die Informationen, welche nötig sind, um das jeweils gewünschte Element zu produzieren.
  + Wie das neu einzutragende Element errechnet wird, ist dem "Zeilenmacher" völlig gleichgültig.
  + Es ist ihm auch gleichgültig, wie der "Zeilenausgeber" (die Prozedur "ZeilenAusgabe") seinen Job erledigt. Er gibt diesem lediglich die Zeile und legt das auszugebende Zeilenstück fest.

Nach dieser Ermahnung erinnert sich unser Programmier-Lehrling daran, daß

- Parameter unter anderem dazu verwendet werden können, um "Datenobjekte" und allerlei Informationen zum Zwecke weiterer Bearbeitung an eine Prozedur zu übergeben, und daß
- die von einer Prozedur A verantworteten Speicherbereiche (Variablen) am besten innerhalb dieser Prozedur deklariert werden. Damit wird erstens verhindert, daß eine nicht von A umschlossene (d.h. nicht im Scope von A liegende) Prozedur auf diese Speicherbereiche schreibend zugreift, wenn es ihr nicht explizit per VAR-Parameterübergabe gestattet wird. Zweitens kann A auf diese Weise auch kontrollieren, wer in den ihr zugeordneten Speicherbereichen lesen kann.

Sein drittes MODULA-2 - Programm macht daher schon einen recht professionellen Eindruck:

```
MODULE EinMalEinsC;
FROM InOut IMPORT WriteCard, WriteString, WriteLn;
CONST TabgrC = 10; ZeilLgC = 10;
TYPE  ZeilenTyp = ARRAY [1 . . ZeilLgC] OF CARDINAL;
VAR  zeilenNr: CARDINAL;
PROCEDURE ZeilenAusgabe(erstNr, letztNr: CARDINAL;
                        zeile: ZeilenTyp);
CONST Blank4C = "    ";
```

```
VAR  i:CARDINAL;
BEGIN
   FOR i:=1 TO erstNr-1 DO WriteString(Blank4C) END;
   FOR i:=erstNr TO letztNr DO WriteCard(zeile[i],4) END;
   FOR i:=letztNr+1 TO ZeilLgC DO WriteString(Blank4C) END;
   WriteLn
END ZeilenAusgabe;

PROCEDURE TabellenEintrag(zNr, sNr:CARDINAL):CARDINAL;
BEGIN
   RETURN zNr*sNr
END MacheEintrag;

PROCEDURE MacheZeile(zNr);
VAR spaltenNr: CARDINAL;
      zeile: ZeilenTyp;
BEGIN
   FOR spaltenNr:=1 TO zNr DO
      zeile[spaltenNr]:=TabellenEintrag(zNr,spaltenNr)
   END (*FOR*);
   ZeilenAusgabe(1,zNr,zeile)
END MacheZeile;

BEGIN (* EinMalEinsC*)
   FOR zeilenNr:=1 TO TabgrC DO MacheZeile(zeilenNr) END
END EinMalEinsC.
```

Der Chef zeigt sich befriedigt darüber, daß - basierend auf dieser Version - nicht nur die aktuelle Spezifikationsänderung, sondern auch die vom Kunden für die Zukunft avisierten Wünsche durch relative kleine Modifikationen des Programmtextes erfüllt werden können. Mit anderen Worten: Das Programm "EinMalEinsC" besitzt die Eigenschaft, relativ leicht *adaptierbar* zu sein.

(Selbstverständlich ist der Leser auch hier aufgefordert, die notwendigen Anpassungen selbst vorzunehmen. Zur Übung möge er sich andererseits fragen, wie gut das obige Programm in Hinblick auf das Qualitätsmerkmal *Integrität* ist! Sollte er zu dem Schluß kommen, daß es in dieser Hinsicht nicht den Erwartungen entspricht, so möge er eine Version "...D" produzieren, die seinen gewachsenen Ansprüchen gerecht wird.)

Wir haben dieses Beispiel deshalb so gründlich (und in krassem Gegensatz zu der in Kapitel 5 vorgeführten Vorgehensweise) behandelt, weil es - gewissermaßen "in der Nußschale" - einen wichtigen Leitgedanken für den Entwurf von Softwaresystemen demonstriert: Die Schaffung von "Organisationseinheiten" mit genau festgelegten Rechten und Pflichten, mit dem diesen Rechten und Pflichten entsprechenden exklusiven Zugang zu Informationen und mit der ausschließlichen Verantwortung für die Verarbeitung oder Darstellung dieser Informationen.

Es mag dem Leser dämmern, daß die Beachtung dieses und weiterer Prinzipien nicht nur auf das Qualitätsmerkmal *Adaptierbarkeit* günstige Auswirkungen hat, sondern auch einige andere der bereits diskutierten Eigenschaften (welche?) verstärkt. Da wir darauf in diesem Kapitel noch ausführlicher eingehen werden, verzichten wir an dieser Stelle auf weitere Erklärungen.

Zum vierten Aspekt von *Brauchbarkeit*, zur *Effizienz*, können wir vom Standpunkt des Entwicklers großer Softwaresysteme nicht mehr beitragen, als wir dies in Kapitel 2 von der Warte des Programmieres "im Kleinen" bereits getan haben. Nach wie vor gilt, daß *Effizienz* eine Sache der geschickten Auswahl von Datenstrukturen und Algorithmen ist. Dieses Faktum erhält noch etwas mehr Gewicht dadurch, daß große Softwaresysteme es nicht selten auch mit großen Datenbeständen zu tun haben, die im allgemeinen auf externen Datenträgern (Platten, Bänder) untergebracht sind. Die Organisation dieser Daten und die Planung des Zugriffs auf sie sind daher außerordentlich wichtige Aufgaben, wenn strenge Anforderungen beispielsweise hinsichtlich der Dauer der Bearbeitung einer Anfrage an ein Datenbanksystem gestellt werden. Es würde jedoch den diesem Buch gesetzten Rahmen sprengen, wenn wir auf diese Problematik im einzelnen eingingen.

Wir können somit die zu Beginn dieses Abschnitts gestellte Frage (1) nach den vier Aspekten der *Brauchbarkeit* als erledigt betrachten.

### 6.2.4 Benutzungsfreundlichkeit

Frage (2) impliziert die Vermutung, daß es mindestens noch einen weiteren, bisher nicht beachteten, aber für viele große Softwaresysteme dennoch wichtigen Gesichtspunkt gibt. Daß dies in der Tat der Fall ist, haben wir schon in der Vorrede zu Kapitel 2 angedeutet. Es handelt sich um das Qualitätsmerkmal

*BENUTZUNGSFREUNDLICHKEIT*.

Dieser Begriff bezieht sich im wesentlichen auf die sogenannte *Benutzungsoberfläche* eines Softwaresystems. Ganz allgemein verstehen wir darunter die Art und Weise, wie ein menschlicher Benutzer die Dienstleistungen des Systems erhält. Dazu gehören die Möglichkeiten der Dienstleistungsanforderung ebenso wie die Präsentation von Ergebnissen der Dienstausführung. Je nachdem, wieviel Mühe ein Mensch bei dieser "Kommunikation mit der Maschine" aufwenden muß, spricht man von mehr oder weniger *Benutzungsfreundlichkeit.* Auch hier (wie schon bei einigen der oben diskutierten Qualitätsmerkmale) besteht die Krux darin, ein Maß für diese Mühe zu finden.

*Benutzungsfreundlichkeit* ist ein besonders wichtiges Kriterium für die Beurteilung von Software-Produkten, welche von vielen Menschen als Werkzeuge für die tägliche Arbeit eingesetzt werden. Dazu fallen uns natürlich zunächst die "klassischen" Standardanwendungen von Mikrorechnern (den Persönlichen Com-

putern, PC) ein, wie Textverarbeitung (und ihre hochkarätige Variante "Desktop Publishing"), Tabellenkalkulation (englisch: "Spreadsheet") und Datenverwaltung. Aber auch in anderen Bereichen kann die Brauchbarkeit eines Softwaresystems ganz entscheidend von seiner *Benutzungsoberfläche* abhängen. Man denke etwa an die computergestützte Überwachung der technischen Abläufe in einem Atomkraftwerk oder daran, daß in modernen Verkehrsflugzeugen der Flugingenieur dadurch "eingespart" wird, daß den Piloten alle relevanten Informationen über den Zustand der Bordsysteme auf einem Bildschirm dargestellt werden.

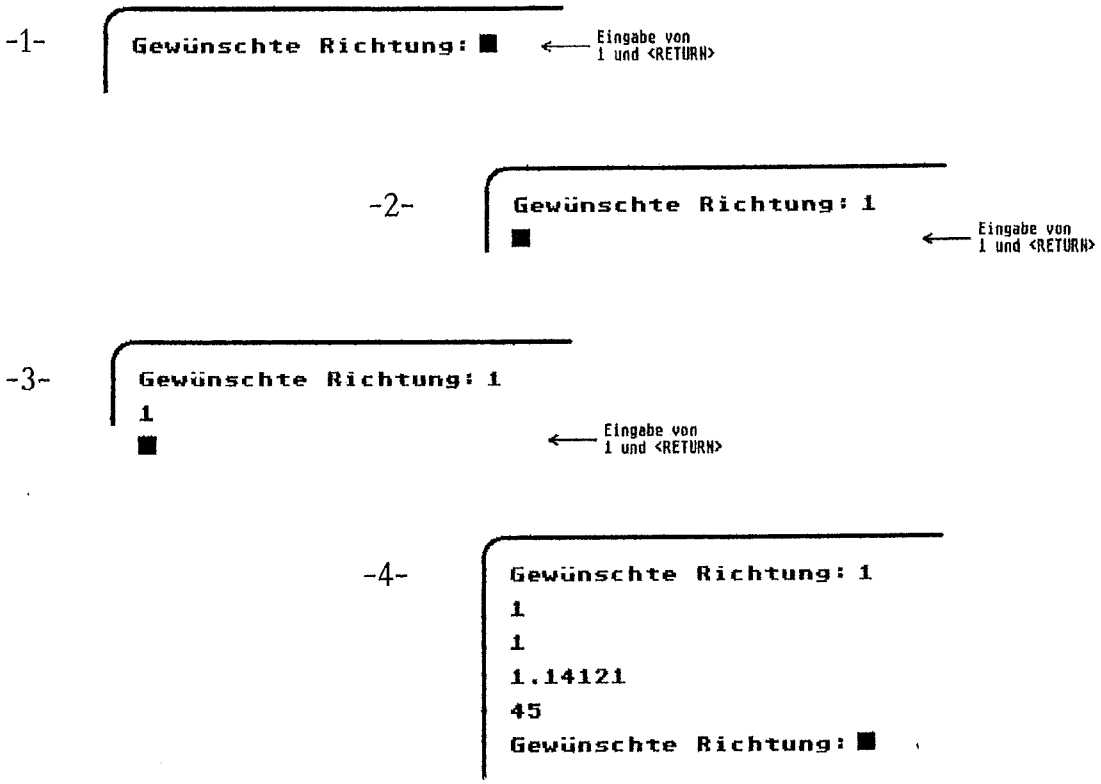
Diese (und viele ähnliche) Fälle verallgemeinernd ist festzuhalten, daß *Benutzungsfreundlichkeit* insbesondere für solche Softwaresysteme gefragt ist, die mit ihren menschlichen Benutzern *interagieren*, mit ihnen einen *Dialog* führen. Bei der Ausführung der Dienste derartiger (Dialog-) Systeme sind regelmäßige Eingriffe des Menschen notwendig, und es kommt darauf an, daß Mißverständnisse seitens des Menschen so weitgehend wie möglich vermieden werden und daß dessen Reaktionen unter keinen Umständen zu unerwünschten oder gar fatalen Reaktionen des Rechners führen.

Diese letzte Bemerkung ist im übrigen gleichbedeutend mit der Feststellung, daß ein an seiner *Benutzungsoberfläche* nicht *robustes* Softwareprodukt ganz sicher auch nicht *benutzungsfreundlich* ist! Diese Behauptung wird gewiß jeder bestätigen, der es (vielleicht in den Anfangszeiten des "Personal Computing") einmal erlebt hat, daß sich eine mehrseitige Schreibarbeit in Nichts auflöste, weil sein (zu früh auf den Markt gebrachtes) Textbearbeitungssystem die Eingabe eines bestimmten Kommandos gerade nicht für akzeptabel befand. (Der Leser erinnere sich an die Diskussion des Qualitätsmerkmals *Robustheit* weiter oben.)

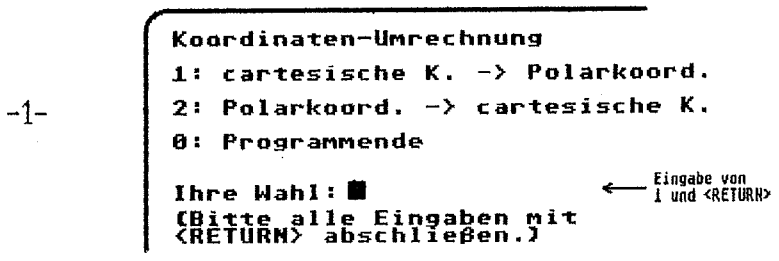
Die "humanen" Aspekte der *Benutzungsfreundlichkeit* von Softwaresystemen sind immerhin so interessant, daß deren Erforschung und ihre Umsetzung in die Praxis zu einer im Rahmen von Informatik und Computertechnik eigenständigen Disziplin, der *Software-Ergonomie*, herangewachsen sind (vgl. etwa [SCH]). Es ist das erklärte Ziel der Vertreter dieser Fachrichtung, Softwaresysteme handhabbarer zu machen, sie so zu gestalten, daß es keines langwierigen Studiums bedarf, um sich ihrer bedienen zu können, daß ihre Benutzung Vergnügen bereitet und daß sie nicht über Gebühr ermüdend ist. Es geht also darum (ähnlich wie im Falle greifbarer technischer Geräte), die Software-Produkte den menschlichen Fähigkeiten und Bedürfnissen anzupassen und ihnen - in diesem Sinne - möglichst einfache *Benutzungsoberflächen* zu verschaffen.

An einem (gezwungenermaßen wiederum sehr kleinen) Beispiel wollen wir uns mit einigen dieser "humanen" Aspekte vertraut machen. Wir betrachten ein Programm, welches die cartesischen Koordinaten eines Punktes in der Ebene umrechnet in seine Polarkoordinaten und umgekehrt. In einer ersten Version führt dieses Programm - unmittelbar nachdem es gestartet wird - den folgenden (hier als Sequenz von Bildschirmausschnitten wiedergegebenen) Dialog: (Das schwar-

ze Rechteck ist die Schreibmarke, der "Cursor", und der Benutzer ist offenbar aufgefordert, Eingaben über die Tastatur zu machen. Rechts ist jeweils die in der dargestellten Situation getätigte Eingabe vermerkt.)




Nur der Autor dieses Programmes selbst oder jemand, der den Programmtext sehr genau studiert hat, wird in der Lage sein, sowohl die durch die Schreibmarke repräsentierten Anfragen des Programms mit gültigen Eingaben zu beantworten, als auch den daraufhin produzierten Output zu interpretieren. Für den nicht Eingeweihten ist das Programm offenbar völlig wertlos. Schauen wir uns daher den Dialog an, der von einer nächsten, mit dem Ziel größerer *Benutzungsfreundlichkeit* verfaßten Version des Programms geführt wird:



-2-

```
Umrechnung
cartesische K. -> Polarkoord.

Ihre Eingaben bitte:
(Abbruch mit <ESC>)

x: ■                          <-- Eingabe von 1 und <RETURN>
```

-3-

```
Umrechnung
cartesische K. -> Polarkoord.

Ihre Eingaben bitte:
(Abbruch mit <ESC>)

x: 1
y: ■                          <-- Eingabe von 1 und <RETURN>
```

-4-

```
Umrechnung
cartesische K. -> Polarkoord.

(x,y) = (1,1)
Entsprechende Polarkoordinaten:

r = 1,14121    phi = 45 (Grad)

1: Weitere Umrechnung
2: Zurück zum Menü
0: Programmende
Ihre Wahl: ■
```

Die Unterschiede im nach außen sichtbaren Verhalten der beiden Programmversionen sind augenfällig genug. Wir bemerken, daß sich die zweite Version im Vergleich mit der ersten durch die folgenden Eigenschaften auszeichnet:

- Dem Benutzer wird klar gesagt, welche Leistungen das Programm anbietet und wie er diese Leistungen verlangen kann. Eine solche Aufzählung der Leistungen eines Programms wird üblicherweise als "Menü" bezeichnet.
- Der Benutzer wird nicht im Unklaren darüber gelassen, "wo er sich befindet": Er wird durch einen schriftlichen Hinweis darauf aufmerksam gemacht, welche Arbeit das Programm jeweils ausführt.
- In jeder Dialogsituation wird der Benutzer davon in Kenntnis gesetzt, welche Eingaben welchen Effekt haben.
- Es besteht jederzeit die Möglichkeit, die gerade laufende Aktivität abzubrechen. (Hier entweder die Auswahl aus einem Leistungsangebot oder die Ausführung einer Leistung selbst.)

Man darf vermuten, daß dieses Programm noch einige andere "freundliche" Charakterzüge hat, welche durch die obigen Bilder nicht illustriert werden konnten.

Zum Beispiel sollte es die in einer bestimmten Situation nicht erlaubten Tastatureingaben entweder einfach ignorieren (etwa bei der Menüauswahl) oder aber einen Fehlerhinweis geben sowie die Gelegenheit, den Fehler zu korrigieren (bei der Eingabe der Koordinatenwerte). (Vgl. unsere Ausführungen zur *Robustheit* !)

In [NIV] werden einige allgemeine Regeln für die Gestaltung der dialoghaften Interaktion mit Softwaresystemen formuliert, die wir hier - mit geringen Modifikationen - auszugsweise zitieren wollen:

> "Ein interaktives Softwaresystem soll sein Leistungsangebot strukturiert präsentieren."
>
> "Anfang und Ende sollen eindeutig markiert werden."
>
> "Der Benutzer soll aus dem Leistungsangebot beliebig auswählen können."
>
> "Die Ausführung von Leistungen und Teilleistungen soll sowohl wiederholbar sein als auch ausgelassen werden können."
>
> "Der Benutzer soll jederzeit auf legale Weise abbrechen können."
>
> "Der Benutzer bestimmt, wann eine weitere Leistung ausgeführt werden soll."

Diese Regeln können als Korollare zu der folgenden allgemeinen Forderung an interaktive technische Systeme aufgefaßt werden:

> "Nicht die Maschine (das Programm) diktiert das Geschehen, sondern - in jedem Fall und zu jeder Zeit - einzig und allein der die Maschine benutzende Mensch."

Für Dialogsysteme ist dieser Anspruch nebst seinen speziellen Korollaren nur zu verwirklichen, wenn der Entwickler solcher Systeme ihren Benutzern jederzeit eine Antwort auf die folgenden, ebenfalls in [NIV] gestellten Fragen, gestattet:

"Wo bin ich?"

"Was kann ich hier tun?"

"Wie kam ich hierhin?"

"Wo kann ich hin und wie komme ich dorthin?"

Heutzutage stehen für die Realisierung interaktiver Softwaresysteme professionelle und sogar standardisierte *Benutzungsoberflächen* in Form von jeweils zusammengehörigen Prozeduren und Funktionsprozeduren zur Verfügung, welche auf dem Bildschirm zum Beispiel das Modell einer Schreibtischoberfläche ("Desktop", also eine *Oberfläche* im wahren Sinne des Wortes!) erzeugen, auf der Ein- und Ausgabebereiche als "Schriftstücke" über- oder untereinandergeschoben werden können. Und auf die Gewohnheiten des Schreibtischarbeiters

wird dadurch Rücksicht genommen, daß ihm vertraute Gegenstände, wie Ablage, Papierkorb oder Radiergummi, durch entsprechende Bilder (engl. "Icons") anschaulich symbolisiert werden. Für die Manipulation der Objekte auf seinem "Schreibtisch" hat er ein per Hand nachführbares Zeigeinstrument (meist eine "Maus"), mit dem er Objekte auswählt, sie ergreift und zum Beispiel auf das Papierkorbsymbol schiebt (womit sie dann von der Oberfläche verschwinden). Grundvoraussetzung für die Bereitstellung solch komfortabler graphischer *Benutzungsoberflächen* ist wiederum der Fortschritt im Bereich der Hardware, und hier insbesondere bei der Bildschirm-Technologie.

Wie schon in anderen Fällen, müssen wir - um die diesem Buch gesetzten Grenzen nicht zu überschreiten, die Diskussion des Themas *Benutzungsfreundlichkeit* an dieser Stelle abbrechen. (Der Leser mag es bei der praktischen Arbeit mit der auf seinem eigenen Rechner laufenden Standard-Software weiterverfolgen!) Festzuhalten ist aber eine für die Strukturierung großer Softwaresysteme besonders wichtige Einsicht: Im "Produktionsbetrieb" Softwaresystem enthält die Gesamtheit der Prozeduren und Funktionsprozeduren zur Gestaltung der *Benutzungsoberfläche* gewissermaßen die für die "Auftragsannahme" und für die "Verpackung" der Produkte zuständigen Organisationseinheiten.

### 6.2.5 Software Engineering

Wir sind noch eine Antwort schuldig auf die dritte der Fragen, welche zu Beginn dieses Abschnitts formuliert wurden. Zur Erinnerung: Gibt es Qualitätsmerkmale (in Bezug auf große Softwaresysteme), die sich nicht aus der allgemeinen Forderung nach *Brauchbarkeit* herleiten lassen? Unsere Antwort hierauf ist "JA". Wenn wir von Brauchbarkeit reden, so meinen wir doch das fertige, im Einsatz befindliche Produkt. Und üblicherweise interessiert sich der Käufer eines Produktes nur für diesen Aspekt. Der Ingenieur hingegen muß auch den Prozeß der Produktentstehung selbst berücksichtigen. So können wir uns im Extremfall durchaus brauchbare Produkte vorstellen, die sich gar nicht herstellen lassen. Es gibt Beispiele (etwa aus dem militärischen Bereich) von großen Softwaresystemen, die man sich wohl ausdachte, deren Realisierung dann aber aufgegeben wurde. Man hatte eingesehen, daß sie zu komplex würden, als daß ihr Bau noch beherrschbar sein könnte und guten Gewissens verantwortbar.

Dies führt uns direkt zu jenem Begriff, auf den Frage (3) hinweist: *Machbarkeit.* Um hieraus Qualitätseigenschaften eines Softwareproduktes herleiten zu können, muß man sich jedoch zusätzlich vor Augen halten, daß ein Softwareprodukt mehr ist als der letztlich entstehende Programmcode. Tatsächlich ist es ein ganzes Bündel verschiedenartiger Dokumente. Insbesondere zählen darunter jene, welche die Spezifikation und den Entwurf beschreiben. Wir haben in Abschnitt 6.1 darauf hingewiesen, daß die Herstellung großer Softwaresysteme in der Regel einen Aufwand in der Größenordnung von vielen Personenjahren erfordert

und daß im Team gearbeitet werden muß, wenn gesetzte Lieferfristen eingehalten werden sollen. Ein Problem des Managements besteht also unter anderem darin, den Mitgliedern eines Teams Teilaufgaben zuzuweisen und deren Ausführung zu koordinieren. Spezifikation und Entwurf müssen die Voraussetzungen zur Lösung dieses Problems schaffen. Eine zweite Erinnerung an unsere Vorrede zu Abschnitt 6.1 legt das entscheidende Stichwort hierfür nahe: *Modularität* Da es im folgenden noch hinreichend Gelegenheit geben wird, die Facetten dieses Qualitätsmerkmals auszuleuchten, verzichten wir an dieser Stelle auf eine weitere Diskussion. Nur soviel: Es wird sich zeigen, daß die meisten der oben aus der Forderung nach *Brauchbarkeit* abgeleiteten äußeren Eigenschaften eines Softwaresystems durch die innere Qualität der *Modularität* bedingt sind.

Der Begriff *Machbarkeit* führt uns nun seinerseits zu einem Thema, welches zunächst ganz allgemein mit der Planung und Abwicklung technischer Großprojekte zu tun hat: Bei ihrer Planung betrifft es das schwierige Problem der Schätzung von Dauer und Kosten solcher Projekte, und bei der Projektabwicklung die nicht minder schwierige Einhaltung des jeweils gegebenen Zeit- und Kostenrahmens. Beides, also Zeit- und Kostengesichtspunkte, in den Griff zu bekommen, ist im Falle software-technischer Großprojekte das Hauptanliegen des *Software-Engineering*, einer Disziplin, auf die sich seit etwa dem Ende der sechziger Jahre die Aufmerksamkeit zahlreicher Informatiker konzentriert.

Damals waren die ersten Rechner der sogenannten "3. Generation" schon einige Zeit auf dem Markt. Deren Prozessorleistung und Speicherkapazitäten waren bereits so beachtlich, daß es äußerst unwirtschaftlich schien, sie mit den bis dahin üblichen einfachen Betriebssystemen für die Verarbeitung von Lochkartenstapeln auszurüsten. Die damals neuen Betriebssysteme mußten vielmehr in der Lage sein, die gleichzeitige bzw. quasi-gleichzeitige Abarbeitung mehrerer Programme zu ermöglichen und die Kommunikation mit vielen externen Geräten zu unterstützen. Diese Fähigkeiten waren zwingend notwendig, um die durch zahlreiche neuartige Anwendungen bedingten Anforderungen zu erfüllen.

Nun hatte man zwar einige Erfahrung bei der Planung und dem Bau von Rechner-Hardware, die Planung und Realisierung jedoch von komplexen Software-Systemen, wie sie für den effizienten Betrieb und die anspruchsvolle Anwendung der Rechner der neuen Generation erforderlich waren, wurden nur mangelhaft beherrscht. Tatsächlich wurde Hardware relativ immer billiger, während die Kosten für die Entwicklung von Software überproportional wuchsen. Außerdem zeigte sich, daß umfangreiche Software für Betriebssysteme und Anwendungen nie fehlerfrei war und oft auch ihren Zweck nicht hinreichend erfüllte. Dies führte zu weiteren, oft hohen Kosten für die Beseitigung von Fehlern, die Erfüllung der ursprünglichen Anforderungen oder die Anpassung an leicht geänderte Aufgabenstellungen. Am teuersten waren Fehler, welche in den Anfangsstadien eines Projekts begangen und erst spät entdeckt wurden (vgl. z.B. [BRO]).

Diese Situation wurde durch das Schlagwort von der *Software-Krise* charakterisiert, und *Software-Engineering* bezeichnete, zunächst noch vage, das Mittel zu ihrer Überwindung. Das Wort "Engineering" legt dabei eine Analogie nahe zwischen den Prinzipien und Methoden klassischer Ingenieurdisziplinen (wie Maschinenbau oder Hochbau) und denjenigen Prinzipien und Methoden, die der Planung und Realisierung von Software-Systemen zugrunde liegen sollten. Und so wie der "klassische" Ingenieur bei seiner Arbeit Werkzeuge benutzt, Reißbrett, Rechenschieber, Bauteilkataloge, und so weiter, so muß Werkzeuggebrauch auch ein charakteristisches Merkmal der ingenieurmäßigen Herstellung von Software sein. Natürlich bestehen die Werkzeuge des Software-Ingenieurs selbst ebenfalls aus Software, mit deren Hilfe die Dokumente, welche die Spezifikation, den Entwurf und die Implementierung eines Programm-Systems repräsentieren, generiert, manipuliert, analysiert und - nicht zuletzt - im Projektzusammenhang verwaltet werden können.

Wir können also *Software-Engineering* etwas präziser definieren als die Anwendung des methodischen Wissens, welches erforderlich ist, um

- große Software-Projekte gut (d.h kostengünstig und termingerecht) zu organisieren,
- die Qualität großer Software-Systeme zu sichern und
- die Entwicklung großer Software-Systeme durch geeignete Werkzeuge zu unterstützen.

Mit anderen Worten: *Software-Engineering* ist nichts anderes als die Praxis der *Programmierung im Großen*, der wir schon zu Beginn von Abschnitt 6.1 einige Erläuterungen gewidmet haben.

Es sollte inzwischen klar geworden sein, daß sich ein wesentlicher Teil dieses methodischen Wissens auf die Strukturierung von Software-Systemen selbst beziehen muß. Einem Projekt als Ganzem (und selbstverständlich dem darin eingebetteten eigentlichen Herstellungsprozeß) muß ein *Produkt-Modell* zugrundeliegen, also eine Vorstellung davon, aus welcher Art von Teilen das Produkt bestehen soll, und von welcher Art die zwischen diesen Teilen existierenden Beziehungen sein können. Ein solches allgemeines Modell zeigt die erste Abbildung auf Seite 232. Aber auch zur Projekt-Organisation ist eine ebenso klare Vorstellung von der Art der Aktivitäten notwendig, die im Rahmen eines Projektes stattfinden, und von deren Aufeinanderfolge. Diese Vorstellung nennen wir *Projekt-Modell*. *Projekt-* und *Produkt-Modell* sind zentrale Begriffe des Software-Engineering (vgl. z.B. [DEH]).

Die zweite Abbildung auf Seite 232 stellt ein Projekt-Modell dar, welches dem Produkt-Modell entspricht. Es enthält die Aktivitäten *Analyse*, *Spezifikation* und *Implementierung*, Spezialisierungen der Aktivität *Spezifikation* auf das System als Ganzes sowie auf Komponenten und Module (die Teiletypen des Produkt-Modells), und es veranschaulicht außerdem den möglichen Ablauf dieser Aktivi-

täten (die wir im übrigen schon zu Beginn des Abschnitts 6.1 genauer beschrieben haben). Die Spezifikation von Teilen eines Systems (also der Komponenten und Module) erscheint hier unter dem Oberbegriff *Entwurf*.

Dieses Projekt-Modell gehört zur Klasse der sogenannten *Phasenmodelle*, denenzufolge, wie aus der Bezeichnung hervorgeht, ein Projekt in einzelne, im Idealfall sequentielle, Abschnitte (*Phasen*) gegliedert wird. Für diese Phasen haben verschiedene Autoren unterschiedliche Namen und durchaus auch unterschiedliche inhaltliche Bestimmungen eingeführt (vgl. z.B. [SOM]). Außer Phasen, die den in unserem Modell so genannten Abschnitten *Analyse*, *(System-) Spezifikation*, *Entwurf* und *Implementierung* entsprechen, werden gewöhnlich auch Aktivitäten wie *Test*, *Integration* und *Wartung* gesondert ausgewiesen. Wir beschränken uns auf unser einfaches Modell und werden uns im weiteren Verlauf dieses Abschnitts 6.1 mit einigen wesentlichen, den vier Phasen dieses Modells zuzuordnenden Fragestellungen beschäftigen. (*Test* und *Integration* werden wir als Teilaktivitäten von *Implementierung* auffassen.) En passant müssen wir zuvor freilich zugeben, daß jene "klassischen" Phasenmodelle, die insbesondere in der Anfangszeit des Software-Engineering geradezu eine Monopol-Stellung einnahmen, inzwischen einige Konkurrenz erhalten haben. Leider können wir in diesem engen Rahmen auch darauf nicht weiter eingehen und müssen auf entsprechende Literatur verweisen (z.B. [HAL]).

Zu den Problemkreisen des Software-Engineering, von denen wir einige in den Unterabschnitten 6.3 und 6.4 berühren werden, zählen unter anderem die folgenden:

Analyse und System-Spezifikation
- Prinzipien der Bildung von Modellen der Teile der "realen Welt", in denen ein Software-System wirken soll.
- Formalisierung solcher Modelle.
- Ableitung von Anforderungen an ein Software-System.
- Formale Darstellung der Anforderungen.
- Gewährleistung eines reibungslosen Übergangs hin zu Entwurf und Implementierung.

Entwurf und Implementierung
- Kriterien für die Zerlegung eines Systems in Komponenten und Moduln.
- (Formale) Beschreibung des Zusammenhangs (d.h. der *Schnittstellen*) zwischen Komponenten und Moduln.
- Benutzung der Basis-Maschine (Basis-Software, Konstrukte der Programmiersprache(n), etc.) (die in vielen Fällen schon durch die allgemeinen Anforderungen an ein System vorgeschrieben sein kann), bzw. auch die Bereitstellung oder Auswahl einer geeigneten Basis-Maschine.
- Testverfahren und Integrationsstrategien.

## 6.3 Systemanalyse und Systemspezifikation

Ohne eine Vorstellung von seiner Umwelt wird sich kein Mensch in ihr bewegen können, geschweige denn in der Lage sein, ihr gestalterisch zu begegnen. Entsprechend der Absicht unseres Verhaltens wird sich unsere Vorstellung auf bestimmte Bereiche und bestimmtes Geschehen konzentrieren. Wollen wir uns zum Beispiel schlicht in einem Straßennetz orientieren, so gilt unsere Aufmerksamkeit den am Wege aufgestellten Orts- und Richtungsschildern. Und wenn es uns in den Sinn kommt, einen Garten anzulegen, so werden wir uns bemühen, Kenntnisse zu gewinnen über Bodenbeschaffenheit, Klimaverhältnisse und die dazu passenden Pflanzen. Es leuchtet ein, daß sowohl die Qualität der mit unserem Verhalten erreichten Wirkung als auch die Mühe, welche wir aufbringen müssen, entscheidend abhängen von der Angemessenheit der diesem Verhalten zugrundeliegenden Vorstellung, von dem Bild, das wir uns über den jeweils interessierenden Weltausschnitt gemacht haben.

In den meisten Situationen unseres alltäglichen Lebens bilden sich die Vorstellungen, an denen wir unser Wandeln und Handeln orientieren, ganz unbewußt. Wir verfügen, wie wohl alle Lebewesen, über zahlreiche, diesen Situationen angepaßte Filter, die sowohl genetisch als auch sozial determiniert sind. Diese Filter sorgen dafür, daß sich unser Gehirn nicht mit den großen Informationsmengen abplagen muß, welche unweigerlich durch unsere Sinnesorgane auf es eindringen, die aber zur Ausführung einer jeweils geforderten Aktivität völlig überflüssig sind.

Der Ingenieur jedoch, der ein technisches Artefakt konstruieren will, welches in einem gegebenen Einsatzbereich einen bestimmten Zweck erfüllen soll, hat es nicht so einfach. Er muß diesen Einsatzbereich unter Umständen einer sehr bewußten und sorgfältigen Analyse unterziehen, bevor er mit der eigentlichen Konstruktion beginnen kann. Andernfalls läuft er Gefahr, ein schwer handhabbares Gerät, eine ungeeignete Maschine oder sogar einen Schaden für die Umwelt zu schaffen. Insbesondere kann die Festlegung der Funktionen und sonstigen Eigenschaften eines nützlichen technischen Produkts niemals unabhängig von seiner geplanten Verwendung erfolgen. Diese durchaus selbstverständlich anmutende Aussage haben wir im übrigen schon in Abschnitt 6.1 in anderer Weise formuliert. Natürlich gilt sie mutatis mutandis für den Software-Ingenieur, der die Aufgabe hat, einen Rechner oder mehrere miteinander verbundene Rechner so zu programmieren, daß dieser beziehungsweise diese sich in einer bestimmten Umgebung in einer gewünschten Weise verhalten, z.B. im richtigen Moment einen auf diese Umgebung wirkenden Output produzieren oder einen Input von dieser Umgebung entgegennehmen.

Der Software-Ingenieur muß die "Umwelt seines Programms" und das Programm selbst als Einheit verstehen. Dabei ist es durchaus sinnvoll, die Hardware, auf welcher das Programm läuft, als einen Teil dieser Umwelt zu betrachten. Es sei

hier auf den in Abschnitt 6.2.1 skizzenhaft dargestellten Zusammenhang zwischen Realwelt, Software und Hardware verwiesen. Da es nun allgemein unpraktikabel und in jedem speziellen Fall auch unerwünscht ist, daß sich die Software mit "der ganzen Welt" beschäftigt, muß der Software-Ingenieur diejenigen Informationen aus der realen Welt herausfiltern, welche relevant sind für die jeweils anstehende Aufgabe. Mit anderen Worten: Er muß die Welt, in der seine Software ein gewünschtes Verhalten realisieren soll, auf ein *abstraktes Modell* reduzieren und in dessen Rahmen dieses Verhalten spezifizieren.

Eine wesentliche Aufgabe der *Systemanalyse* besteht also darin, angemessene Modelle zu finden. Und wenn wir hier von *System* sprechen, so meinen wir in der Regel ein existierendes System: die Umgebung nämlich, innerhalb derer die entsprechend programmierten Rechner eingesetzt werden sollen. (Eine Bank, die Verwaltung eines Wirtschaftsunternehmens, die Herstellung von Kraftfahrzeugen, ein Flugzeug, die Elektrizitätsversorgung, eine Zahnarztpraxis, und so weiter und so fort.) Natürlich dürfen wir nicht verschweigen, daß Systemanalyse weitere wichtige Tätigkeiten beinhaltet, wie zum Beispiel Wirtschaftlichkeitsbetrachtungen und erste Untersuchungen zur Machbarkeit einer software-technischen Lösung. In dem Begriff *Systemspezifikation* hingegen bezeichnet *System* die zu implementierende software-technische Lösung selbst. Stark verkürzt ausgedrückt wird deren Spezifikation unter anderem darin bestehen, diejenigen Teile des Realwelt-Modells "herauszupräparieren", welche durch Software ersetzt oder "gemanagt" werden können oder sollen.

In diesem Abschnitt werden wir uns zunächst einen für unsere Zwecke ausreichenden Überblick über verschiedene Ansätze zur Bildung von Realwelt-Modellen verschaffen und die für uns "als Software-Ingenieure" interessanten Aspekte herausstellen. Im folgenden werden wir uns dann mit einigen, in Theorie und Praxis eingeführten Formalisierungen beschäftigen, müssen jedoch schon an dieser Stelle bemerken, daß wir damit ein Gebiet nur streifen, welches viel zu umfangreich ist, als daß ein einzelnes Buch hierfür als Reiseführer dienen könnte. Die Formalisierungen, die wir auswählen, beschreiben die von uns als bedeutsam erachteten Modell-Aspekte mit zunehmender "Spezifikationsnähe". Während *Petrinetze* (und andere, mit diesen verwandte netzartige Formalismen) sehr allgemeine, von Software und anderen Implementierungsmöglichkeiten durchaus unabhängige Abstraktionsmechanismen und Systemdarstellungen anbieten, werden die dem sogenannten "*Jackson System Development*" zugrundeliegenden Diagrammtechniken und methodischen Anleitungen schon eher der direkten Beschreibung von Software im Zusammenhang mit ihrer Umgebung (s.o.) gerecht. Den unmittelbarsten Übergang in den Entwurf und die Implementierung von Programmen bieten vielleicht ausgeprägt *hierarchische* Systemmodelle, deren Entstehungsprinzip dem der "Schrittweisen Verfeinerung" im Kleinen entspricht, und von denen es - wie im Falle der netz-orientierten Modelle - zahlreiche Varianten gibt. Eine der ältesten wird duch das Kürzel *SADT* be-

zeichnet ("Structured Analysis and Design Technique"). Diese und eine weitere werden wir ebenfalls in ihren Grundzügen skizzieren. (Wir nennen diese Modelle ausdrücklich *ausgeprägt hierarchisch*, weil auch zum Beispiel netz-orientierte Modelle die Möglichkeit der Verfeinerung oder Vergröberung von Strukturen bieten, ohne daß dies jedoch ein besonders charakteristisches Merkmal dieser Formalismen ist.)

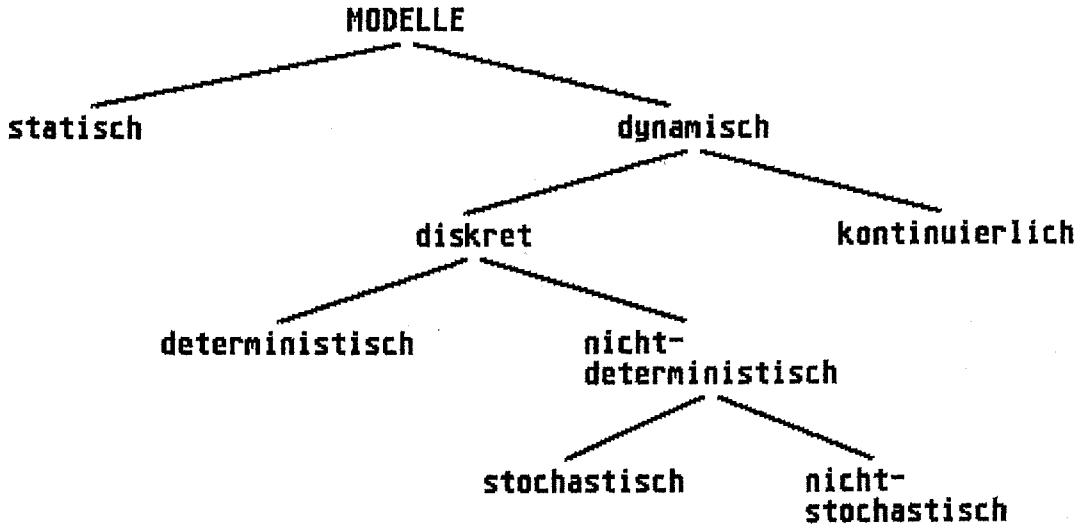
### 6.3.1 Realweltmodelle für die Software-Spezifikation

In verschiedenen Bereichen des Lebens und auf verschiedenen Gebieten von Wissenschaft und Technik hat der Begriff *Modell* jeweils verschiedene Ausprägungen. Aus unserer Kindheit erinnern wir vielleicht die Modell-Eisenbahn oder die Puppenstube. Beide waren in gewisser Weise Abbilder unserer Wirklichkeit. Die dort sich befindenden Objekte hatten manche Ähnlichkeit mit richtigen Lokomotiven oder echten Küchengerätschaften. Im Spiel konnten wir uns nachahmenderweise in der realen Welt der großen Leute wähnen. Später, in der Schule zum Beispiel, haben wir andere Modelle kennengelernt: Im Biologieunterricht etwa die anatomisch getreu geformte Menschengestalt, der man die Organe des Körpers einzeln entnehmen konnte; in der Chemiestunde die bunten, aus kleinen und großen Kugeln bestehenden Veranschaulichungen atomarer Verbindungen; und in der Physik schließlich die Reduktion eines massigen Gebildes - zum Beispiel eines Pendelkörpers - auf einen ausdehnungslosen Punkt oder die Darstellung eines Atoms als Kern, um den, Planeten einer Sonne gleich, auf Bahnen mit merkwürdigen Eigenschaften, die Elektronen kreisen.

All diesen Modellen ist gemein, daß sie die für ein gegebenes (Erkenntnis-, praktisches oder auch nur Spiel-) Interesse relevanten *Objekte*, *Beziehungen* und *Vorgänge* der realen Welt abbilden. Mit manchen dieser Modelle kann man sogar rechnen.

Tatsächlich ist es diese Charakterisierung, welche auch unser Verständnis des Modellbegriffs wiedergibt. *Objekte*, *Beziehungen* und *Vorgänge* sind die Namen jener Kategorien, nach denen sich die für das vorwiegend praktische Interesse des Software-Ingenieurs bedeutsamen Phänomene der realen Welt ordnen lassen. Praktisches Interesse legt zudem einen weiteren, in unserem Zusammenhang wichtigen Aspekt des Modellierens nahe: Wie ein Architekt vor der Ausführung des eigentlichen Baus ein maßstabsgetreues Modell des geplanten Gebäudes erstellt, um daran bestimmte Eigenschaften zu studieren (etwa um zu sehen, wie es in die Landschaft paßt), so wird sich aus vergleichbarem Grund auch der Software-Ingenieur vor der Implementierung ein Bild des von ihm erdachten Systems machen wollen. (Wir wollen nicht unterschlagen, daß es in anderen Lebens- und Wissenschaftsbereichen und inbesondere in der Mathematischen Logik, völlig andere Verwendungen des Begriffs *Modell* gibt.)

Wie nun die obige kurze Aufzählung von Beispielen für Modelle deutlich macht, wird es eine große Vielfalt unterschiedlicher Betrachtungsweisen hinsichtlich der genannten Kategorien geben, Betrachtungsweisen, von denen sicher nur einige für das praktische Anliegen der Konstruktion von Software-Systemen in Frage kommen. Um diese herauszuarbeiten, orientieren wir uns an der folgenden, möglichen (und sicher nicht vollständigen) Klassifikation von Modellen der realen Welt:




In *statischen* Modellen findet die Dimension "Zeit" keine Berücksichtigung. Es kommen, mit anderen Worten, nur die Kategorien *Objekt* und *Beziehung* ins Spiel. Der "Plastikmensch" und die Darstellung der Molekülstruktur sind solche statische Modelle. Systemanalyse und -spezifikation in unserem Sinne (s.o.) werden es mit statischen Modellen zum Beispiel dann zu haben, wenn es darum geht, die Daten zu sichten und zu ordnen, die für den Erfolg eines Unternehmens notwendig sind. Dann wird festzustellen sein, welche *Entitäten* (d.h. Arten von Gegenständen und Personen, also *Objektklassen*) in der realen Welt des Unternehmens vorkommen, welches die zwischen diesen Entitäten bestehenden Beziehungen sind und durch welche charakteristischen Attribute die Entitäten und Beziehungen beschrieben werden können.

*Dynamische* Modelle sind zu bilden, wenn die Dimension "Zeit" eine entscheidende Rolle spielt. Neben *Objekt* (bzw. *Entität*) und *Beziehung* ist *Vorgang* (oder, ein geläufiges, gleichbedeutendes Fremdwort benutzend, *Prozeß*) der für diese Klasse von Modellen wesentliche Grundbegriff. Objekte können sich in verschiedenen *Zuständen* befinden, und zwischen diesen Zuständen bestehen Beziehungen, die durch Worte wie *vorher* und *nachher* oder andere, zeitliche Befindlichkeiten bezeichnende Wendungen ausgedrückt werden. Zustände ändern sich aufgrund von *Ereignissen*, deren Eintreten ebenfalls an Zeitpunkten festzumachen ist. Das oben erwähnte Bohr'sche Atommodell gehört zur Klasse

der dynamischen Modelle. Und der die Konstruktion von Software vorbereitende Systemanalytiker wird sich kaum damit begnügen, *Entitäten* und ihre *Beziehungen* zu untersuchen. Vielmehr wird er sein Augenmerk darauf richten, wodurch und in welcher Weise sich diese im Laufe der Zeit ändern.

Im Gegensatz zu *kontinuierlichen* dynamischen Modellen werden bei *diskreter* dynamischer Modellierung sowohl der Raum der Zustände (von Objekten, Beziehungen und Vorgängen) als auch die Zeit selbst als *atomar* aufgefaßt. Dies bedeutet, daß Zustände und Zeitpunkte aus endlichen oder höchstens abzählbar unendlichen Wertevorräten entnommen sind. Das Modell des Pendels als Massenpunkt ist ein kontinuierliches dynamisches Modell, während das genannte Atommodell als diskret einzuordnen ist. Unser Systemanalytiker wird sich in der täglichen Praxis sicherlich weitaus mehr mit diskreten als mit kontinuierlichen Modellen beschäftigen. Dies gilt ganz besonders etwa dann, wenn er die Grundlagen eines betrieblichen Informationssystems erarbeitet. Dort spielt sich das relevante Geschehen im allgemeinen zu diskreten Zeitpunkten ab: die Zeit ist hier nichts als die abzählbare Folge der Stunden, Tage, Quartale oder Jahre, und auch die Zustände von Objekten (Mitarbeiter, Produkten, Lagerbestände, etc.) lassen sich als Vielfache von Einheiten oder durch Elemente aus endlichen Mengen angeben. Dennoch darf das Ausmaß der für die Produktion von Software notwendigen kontinuierlichen Modellierung nicht unterschätzt werden: Differentialgleichungen, die wohl am weitesten verbreiteten Formen kontinuierlicher dynamischer Modellierung, sind (um nur ein wichtiges Anwendungsgebiet zu nennen) gleichsam die Seele zahlreicher Softwaresysteme, welche technische Prozesse steuern oder simulieren.

Für den jeweiligen, unseren Betrachtungen zugrundeliegenden Teil der realen Welt kann es genügen, die dort zu beobachtenden Zustandsänderungen als *deterministisch* zu modellieren. Zwischen solchen Ereignissen besteht dann immer eine feste zeitliche Ordnung. In einen gegebenen Zustand versetzt wird sich ein gemäß dieser Vorstellung konstruiertes System nach ein und derselben äußeren Einwirkung immer in ein und demselben Folgezustand befinden. Seine zeitliche Entwicklung ist mit Sicherheit voraussagbar, wenn bekannt ist, welchen externen Einflüssen es ausgesetzt ist. (Schon die alten Griechen besaßen zum Beispiel Modelle der Bewegungen der Gestirne, die es ihnen gestatteten, das Eintreten von Sonnen- und Mondfinsternissen auf den Tag genau anzukündigen. Und im alltäglichen Geschäft kann der mit der Analyse der Lagerhaltung eines Unternehmens beauftragte Informatiker davon ausgehen, daß die Entnahme der Menge x eines Artikels A zu einer Reduktion um x des Bestandes von A im Lager führen wird.) Im allgemeinen sind in der Realität solch sichere Prognosen keineswegs immer möglich. Je nach Interesse müssen wir diese Allgemeinheit auch in unseren Modellvorstellungen zulassen: wir erhalten dann *nicht-deterministische* Modelle. Diese ihrerseits können noch weiter verfeinert werden, indem man den Zustandsübergängen eine Wahrscheinlichkeitsverteilung unterlegt, die empirisch

abgesichert sein mag. Man spricht insofern von *stochastischen* beziehungsweise *statistischen* Modellen.

Wir haben mit diesen Unterscheidungen nur den *diskreten* Ast unseres Modellbaumes verlängert, doch gelten sie entsprechend auch für die große Klasse der kontinuierlichen Modelle. Nota bene: Das Geschehen in der Erdatmosphäre zum Beispiel, gemeinhin als "Wetter" bezeichnet, wird üblicherweise mit Hilfe *dynamischer, kontinuierlicher, nicht-deterministischer statistischer* Modelle so beschrieben, daß ein geeignetes Rechnerprogramm aufgrund dieser Beschreibung mit Wahrscheinlichkeiten behaftete Aussagen über das Wetter von Morgen produzieren kann.

Obwohl sich also alle hier stichwortartig vorgestellten Varianten von Modellklassen der Entwicklung jeweils einschlägiger Software zuordnen lassen, versteht es sich von selbst, daß wir uns im Rahmen weiterer Ausführungen zu diesem Thema weise beschränken müssen, und dies sowohl hinsichtlich des Detaillierungsgrades als auch was die Auswahl der darzustellenden Modellklassen und ihrer Vertreter angeht. So verzichten wir einerseits auf die Darstellung statischer Datenmodelle, da diesen zum Beispiel in Lehrbüchern, welche speziell dem Entwurf Datenbank-basierter Informationssysteme gewidmet sind, genügend Raum gewährt wird (vgl. etwa [SCS], [BEY]). Auf der anderen Seite ignorieren wir die große Klasse der kontinuierlichen Modelle. Ein Exkurs in dieses Gebiet würde uns von unserem eigentlichen Ziel, Verständnis für die grundlegenden Probleme der Konstruktion großer Software-Systeme zu erlangen, doch allzuweit ablenken (und im übrigen mathematische "Landkarten" erfordern, die zu erläutern hier nicht der Platz wäre). Von unserem Modellbaum bleibt so "nur" der *dynamische* und *diskrete* Ast übrig, und auch von diesem werden wir noch den *stochastischen* Zweig abschneiden. Dennoch ist dieser Ast ohne Zweifel mächtig genug.

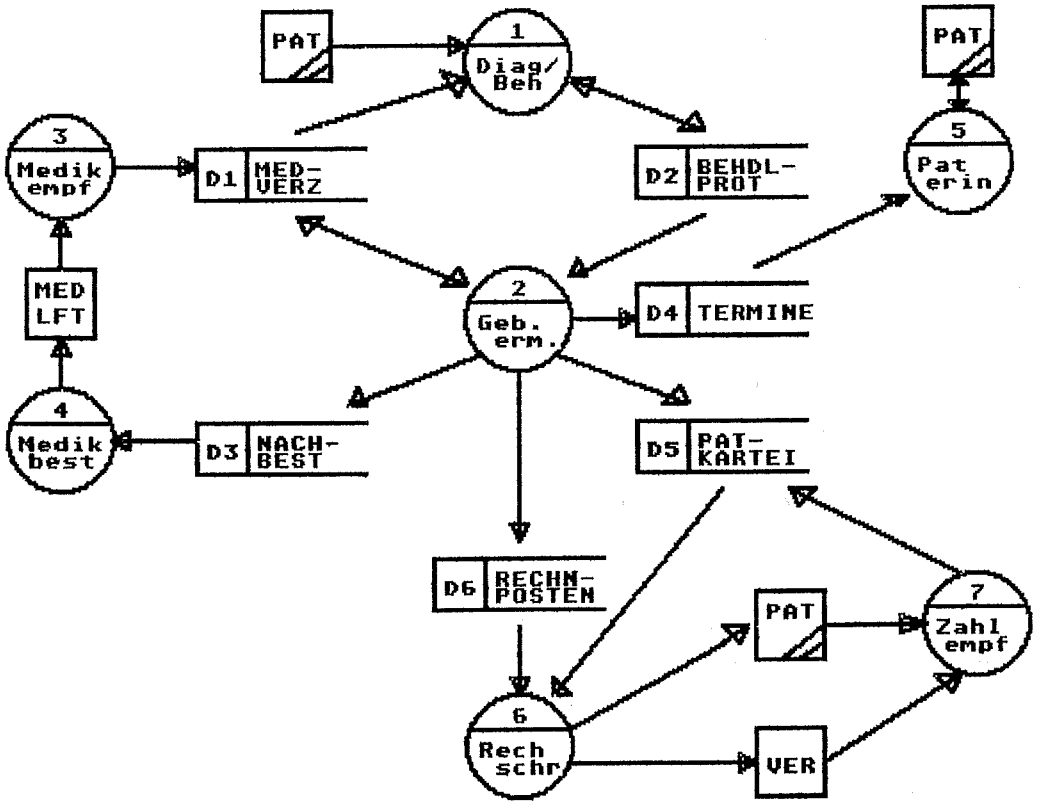
Wir wiederholen: Die für unsere Modelle relevanten Grundbegriffe sind *Objekte*, *Beziehungen* und *Prozesse*, mit letzteren als der für die Dynamik verantwortlichen Zutat, der folglich unser Augenmerk hauptsächlich gelten wird. Objekte können in verschiedener Weise mit Prozessen verbunden sein: Zum einen *aktiv*, als *Träger* oder, besser, als *ausführendes* Organ von Prozessen. (Der Prozessor eines Rechners, zum Beispiel, ist ausführendes Organ der Prozesse, die durch in Programmen niedergelegte Anweisungen gesteuert werden.) Zum zweiten, und das ist sicher weitaus häufiger der Fall, sind Objekte in *passiver* Weise Prozessen untergeordnet: Sie können durch Prozesse verändert werden und sie können von einem Prozeß zu einem anderen weitergereicht oder, um es etwas vornehmer auszudrücken, *kommuniziert* werden.

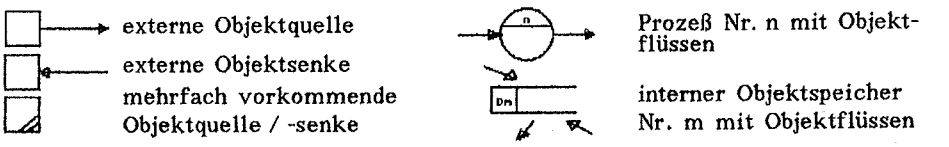
Wir wollen uns nun nicht in längere Erörterungen darüber einlassen, worin der Unterschied zwischen einem Prozeß und den ihn steuernden Anweisungen besteht (der Leser mag sich dazu unter Berücksichtigung des bisher Gesagten seine eigenen Gedanken machen) oder ob ein Objekt, nachdem es einem Pro-

zeß der Bearbeitung unterzogen wurde, noch immer das gleiche Objekt ist. Sicher haben solche Erörterungen einen Sinn, und es mag sein, daß wir zu gegebener Zeit wieder auf solche Fragen zurückkommen. Vorläufig wollen wir uns damit begnügen, eine ganz praktische Analogie zwischen unseren Grundbegriffen der Modellbildung einerseits und unserer Alltagswelt andererseits festzustellen, welche die in Kapitel 5 angedeuteten Vorstellungen fortsetzt. Danach sehen wir ein System im wesentlichen bestimmt durch eine (irgendwie strukturierte) Gesamtheit von Prozessen, die miteinander kommunizieren und (zur Erreichung bestimmter Ziele) kooperieren. Wir sehen wieder den Fertigungsbetrieb oder ein Dienstleistungsunternehmen mit Mitarbeitern unterschiedlicher Qualifikation (gewissermaßen den "Prozessoren"), die für die Ausführung der durch den Unternehmenszweck gegebenen einzelnen Prozesse zuständig sind. Bauteile und Dokumente sind im allgemeinen die Objekte, welche durch diese Prozesse bearbeitet und von einem Prozeß zum nächsten weitergereicht werden (wobei natürlich der physikalische Vorgang des Weiterreichens selbst von einem Prozessor/ Mitarbeiter besorgt wird). Und wie wir wissen, ist es nicht unüblich, daß sich ein Mitarbeiter um mehrere Prozesse kümmert, die durch ganz unterschiedliche Arbeitsanweisungen definiert sein können. (Der mit Rechner-Betriebssystemen hinreichend vertraute Leser wird hier - siehe oben - wiederum die Analogie zur Rechnerwelt erkennen.) Die Abbildung auf der folgenden Seite gibt eine noch relativ grobe Übersicht über Prozesse, die sich in einer Arztpraxis abspielen mögen, und über deren Zusammenhang. Darstellungen dieser Art gehen auf einen Vorschlag von Gane und Sarson ([GAS]) zurück; sie werden daher auch "Gane-Sarson Diagramme" genannt.

Sie machen die Kommunikationsbeziehungen zwischen den Prozessen eines Systems recht anschaulich sichtbar. Die graphischen Symbole, einschließlich der den "Objektfluß" beschreibenden Pfeile, sind durchaus selbsterklärend. Es gibt Prozesse, die mit der "Außenwelt" des Systems kommunizieren, indem sie Objekte entgegennehmen oder abgeben. Andere Prozesse erhalten Objekte lediglich aus systeminternen Objektspeichern und legen Objekte auch nur in solchen ab. (Im Falle von reinen Informationssystemen sind dies natürlich informationstragende Objekte, also, wenn man von unterschiedlichen Medien abstrahiert, irgendwelche Daten.) Derartige "Objekt-" oder "Daten-Fluß-Diagramme", ergänzt um weitere, die darin auftauchenden Elemente näher erläuternde Angaben, sind als relativ einfache und leicht verstehbare Systembeschreibungen in der Praxis recht beliebt. Es ist für den Systemanalytiker sehr wichtig, über solche Formen der Darstellung etwa von in einem Betrieb vorgefundenen Sachverhalten zu verfügen. Mit ihrer Hilfe kann er in Zusammenarbeit mit Gesprächspartnern, die in der Technik der Datenverarbeitung oftmals nicht versiert sind, ohne großen formalen Aufwand, aber dennoch hinreichend exakt seine Sicht der Dinge dokumentieren. (Zu bemerken ist noch, daß für die in Daten-Fluß-Diagrammen ver-

wendeten Symbole durchaus auch andere, von den hier gewählten abweichende, graphische Gestalten gebräuchlich sind.)




**Prozesse:**

1: Diagnose / Behandlung
2: Gebühren ermitteln
3: Medikamente empfangen
4: Medikamente bestellen
5: Patienten erinnern
6: Rechnungen schreiben
7: Zahlungen empfangen

**Objektspeicher:**

D1: Medikamentenverzeichnis
D2: Behandlungsprotokolle
D3: Nachbestell-Liste
D4: Besuchstermin-Kalender
D5: Patientenkartei
D6: Rechnungsposten

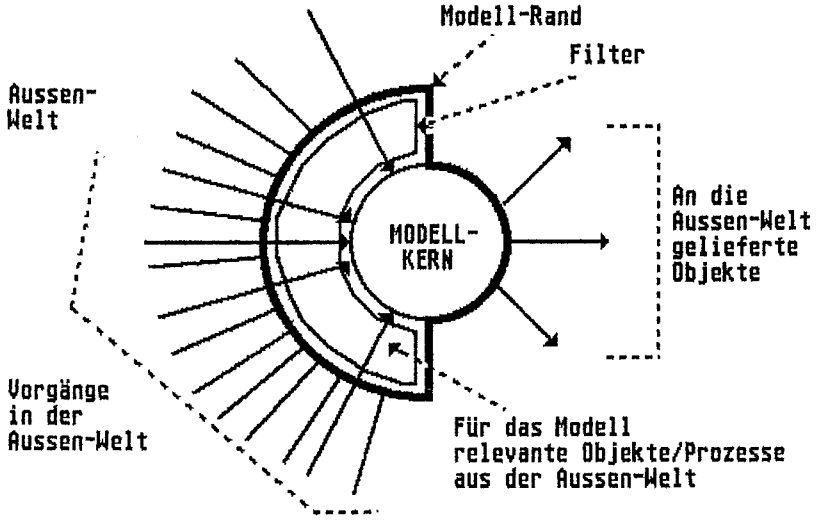
Daten-Fluß-Diagramme (mit entsprechenden Ergänzungen und Präzisierungen, auf die einzugehen hier nicht der Ort ist) sind sicherlich mögliche Vehikel auf dem Weg zum Ziel unserer Modellbildung, nämlich zum einen der

- Abgrenzung des Aufgabenbereichs durch Auffinden und Bezeichnen von relevanten
  * Objekten,
  * Beziehungen zwischen Objekten und von
  * Prozessen, an denen die Objekte aktiv oder passiv beteiligt sind,

und zum anderen der

- Darstellung der Strukturen von
  * Objekten,
  * Prozessen und der
  * Prozeß-Kommunikation.

Sie illustrieren ein allgemeines Konstruktionsprinzip, welches mehr oder weniger allen für die Entwicklung von Informationssystemen geeigneten Modellen zugrundeliegen sollte. Es verlangt die Unterscheidung von *Außenwelt*, *Modellrand*, *Filter* und *Modellkern*. Der *Modellrand* umschließt *Filter* und *Modellkern*. Der *Filter* sorgt dafür, daß nur die in Bezug auf das konkrete Modellierungsinteresse bedeutsamen Objekte und Prozesse der *Außenwelt* einen Einfluß auf das Geschehen im *Modellkern* haben. Andererseits muß das Modell erkennen lassen, was das System für die *Außenwelt* leistet, welche Objekte es produziert und in welcher Weise es auf externe Prozesse einwirkt. Dieser *Output* und der durch den *Filter* definierte *Input* bilden zusammen das *Interface* (die Schnittstelle!) des modellierten Systems. Seine Beschreibung kann unter Umständen recht komplex werden. Das folgende Bild zeigt die allgemeine Modellstruktur:




Zahlreiche natürliche und künstliche Systeme lassen sich anhand von Modellen, welche dieser Grundstruktur folgen, untersuchen und verstehen. Insbesondere

im Fall mancher künstlicher Systeme, zum Beispiel von Dienstleistungsunternehmen, ist dies nicht weiter verwunderlich, sind sie doch meist mehr oder weniger bewußt nach eben diesem Schema konzipiert. Der Schalter einer Bank etwa, mit den auf ihm bereitgestellten Formularen zur Anforderung von Überweisungen, Abhebungen, Einzahlungen und so weiter, kann als der Filter zum "Kern" der bankinternen Vorgänge begriffen werden.

Nun ist, wir erinnern uns, die Analyse eines Systems für den Software-Ingenieur (beziehungsweise für den mit ihm zusammenarbeitenden Systemanalytiker) keine *l'art pour l'art* Aktivität. Vielmehr erfolgt sie zum Zweck, eine Grundlage für die Spezifikation des Einsatzes eines oder mehrerer Rechner zu schaffen, welche alle oder einige der "Funktionen" des Systems übernehmen können. Das Ziel des Rechnereinsatzes ist also die *vollständige* oder *teilweise Automation* sowohl des Filters als auch des Modellkerns. Beispielsweise werden die Dienstleistung "Geldabheben" einer Bank und die damit verbundenen internen Buchungsvorgänge durch einen Geldausgabeautomaten vollständig, also ohne direkte Intervention eines Bankangestellten, automatisiert. Von teilweiser Automation würde man in diesem Fall sprechen, wenn dem Schalterbeamten ein rechnergestütztes Konto-Informations- und Transaktions-System zur Verfügung stünde.

Bei der Darstellung von Kommunikationsbeziehungen ist es oft von Interesse oder sogar von entscheidender Bedeutung, auch das *Medium*, mit dessen Hilfe die Kommunikation erfolgt, zu berücksichtigen. Wir unterscheiden grundsätzlich zwei Arten von Medien: *flüchtige* und *nicht-flüchtige*. Ein flüchtiges Medium zeichnet sich dadurch aus, daß es ein Objekt, welches ihm von dem mitteilenden Kommunikationspartner "anvertraut" wird, nicht für sich behält und (im Idealfall) in der ursprünglichen Form dem Empfänger zugänglich macht. Wenn es sich bei den Objekten um handfeste Gegenstände handelt, so ist dies eine Eigenschaft, die wir wohl von jedem für deren Transport verwendeten Medium erwarten. (Ein gutes Beispiel ist das Fließband.) Geht es hingegen um die Weiterleitung von (durch Signale und Daten repräsentierten) *Informationsobjekten*, so sind uns hierfür Medien durchaus vertraut, welche diese Eigenschaft nicht aufweisen. Wenn wir eine Zeitung lesen, so löscht dieser Vorgang die Druckerschwärze keineswegs, ihr Inhalt bleibt unverändert. Die Zeitung ist ein *nicht-flüchtiges* Medium oder, wie es sich auch anbietet zu sagen, ein *Informationsspeicher*. Demgegenüber wollen wir flüchtige Medien (wie das Fließband) als *Kanäle* bezeichnen. (Ein Vortragender, beispielsweise ein Dozent in einem Hörsaal, pflegt beide Arten von Medien simultan zu benutzen: Er spricht zu seinen Studenten und er schreibt an die Wandtafel.) Kanäle und Speicher begegnen uns in vielen Formen bei der Analyse realer Systeme, deren "Lebensprinzip" Kommunikation heißt. Entsprechend sind sie dann auch für die Spezifikation der Rechnerunterstützung solcher Systeme zu berücksichtigen. In den folgenden Abschnitten, welche jeweils verschiedenen (mehr oder weniger formalen) Methoden der (auf

die Spezifikation von Rechnerunterstützung gerichteten) Modellierung gewidmet sind, werden wir diesem Aspekt unsere besondere Aufmerksamkeit widmen.

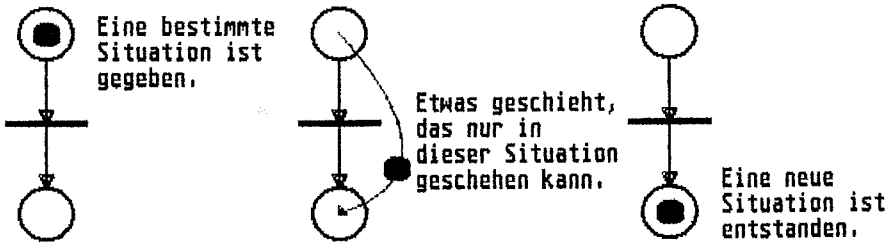
Bevor wir jedoch diesen Abschnitt beenden, wollen wir uns bewußt machen, in welchem Maße die "Realwelten", die wir mit Hilfe des künstlichen Hilfsmittels "Rechner" besser beherrschen wollen, ihrerseits höchst künstliche Gebilde sind. Es sind "soziotechnische Systeme" mit ihren von Menschen geschaffenen Verwaltungsabläufen, Gesetzen und Traditionen, oder von Ingenieuren konstruierte technische Anlagen, mit denen es der Analytiker zu tun bekommt. Sie sind oftmals selbst durch stringente Formalismen definiert, wie zum Beispiel durch Kunstsprachen, von denen jene, die zur Programmierung von Rechnern verwendet werden, dem Leser dieses Buches am geläufigsten sein sollten. Streng formalisierte Kunstsprachen finden Einsatz aber auch in ganz anderen Bereichen, etwa bei der Beschreibung von Dokumenten aller Art. Dies ist eine notwendige Voraussetzung dafür, daß Dokumente mittels Rechnern erstellt, bearbeitet und kommuniziert werden können.

Glücklich zu preisen ist vielleicht jener Programmierer, der die Freiheit hat, sich seine eigene Realwelt als ein Spiel mit von ihm allein bestimmten Regeln zu erträumen und ihr im Rechner Gestalt zu geben (vgl. aber Kapitel 1, Seite 2 !).

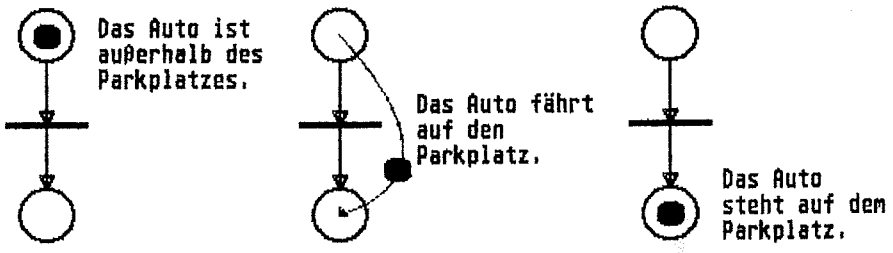
### 6.3.2 Petrinetze

Bei der Modellierung diskreter dynamischer Systeme kann man sich von der Vorstellung leiten lassen, daß sich der Zustand eines solchen Systems aufgrund von Ereignissen ergibt, für deren Eintreffen selbst wiederum der jeweils vorangegangene Zustand des Systems verantwortlich ist. Der unter der Bezeichnung *Petrinetze* in die Informatik eingegangene Formalismus ist ein zur Präzisierung dieser Vorstellung sehr gut geeignetes Instrument. Er beruht auf Ideen, die erstmals von C.A. Petri in dessen 1962 erschienenen Dissertation "Kommunikation mit Automaten" ([PET]) konkretisiert wurden. Die dem Formalismus eigene Darstellung der Begriffe *Zustand* und *Ereignis* erlaubt es, aus diesen beiden grundlegenden Elementen Systembeschreibungen anzufertigen, welche unter anderem Konzepte integrieren, die für das Studium der Kooperation und Kommunikation von Prozessen wesentlich sind. Diese Aussage durch einige Illustrationen zu belegen, muß, angesichts der überwältigenden Fülle des seit 1962 entstandenen Schrifttums zum Thema *Petrinetze*, der Zweck dieses Abschnitts bleiben. (Für das weiterführende und vertiefende Studium von *Petrinetzen* sei auf spezielle Lehrbücher verwiesen, z. B. [BAU].)

Die folgende Bilderserie möge zur Veranschaulichung der Grundidee beitragen:

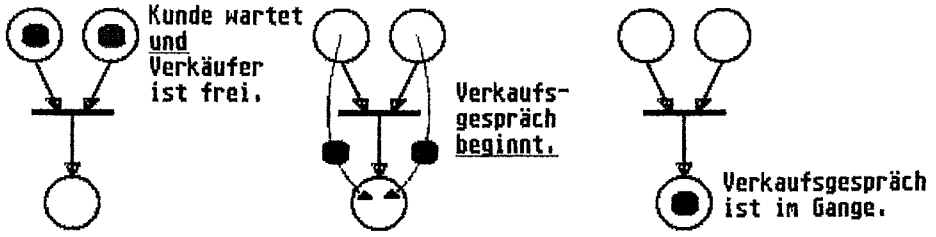



Eine konkrete Interpretation dieses Schemas aus "Zuständen" und Ereignissen, durchaus bereits ein vollständiges Petrinetz, könnte zum Beispiel folgendermassen lauten:

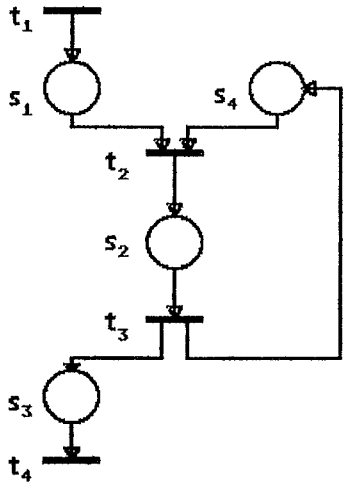



Den Begriff "Zustand" haben wir in Anführungszeichen gesetzt, denn noch wissen wir nicht, was eigentlich der Zustand eines Petrinetzes ist. Doch von Ereignissen zu sprechen, bereitet offenbar weniger Schwierigkeiten. Beschränken wir uns zunächst auf die gezeigten sehr einfachen Netze: Wir sehen, daß das mögliche Ereignis graphisch durch einen dicken Balken repräsentiert wird. Die Tatsache, daß die Bedingung für das Eintreten des Ereignisses erfüllt ist, wird durch die Marke (oft auch *Token* genannt) in jenem Kreis ausgedrückt, aus dem ein Pfeil zum Balken führt. Ist die Marke an jener *Vorgängerstelle* vorhanden, so kann das Ereignis *"Auto fährt auf den Parkplatz"* geschehen, muß aber nicht. Daß es geschieht, erkennt man daran, daß die Marke von der *Vorgängerstelle* abgezogen und eine Marke auf die *Nachfolgerstelle* gelegt wird. Das Ereignis wird in der Terminologie der Petrinetze auch als *Transition* bezeichnet und der Vorgang seines Eintretens wird mit den Worten *"die Transition schaltet"* umschrieben. Der *Zustand* des Netzes ist nun durch die jeweils bestehende Belegung seiner *Stellen* mit Marken definiert. Vergegenwärtigen wir uns hier vorläufig (d.h. noch ohne die genaue Kenntnis von Konstruktionsregeln und Beispielen) die Möglichkeit sehr viel komplexerer Netze, so wird einsichtig, daß deren "Verhalten" in einem bestimmten Zustand durch die in diesem Zustand erlaubten, das heißt schaltfähigen *Transitionen* gegeben ist.

Die folgende, an den obigen Beispielen orientierte Bilderserie zeigt, daß eine Transition auch mehrere Vorgängerstellen haben kann:




Das Verkaufsgespräch kann nur dann beginnen, wenn sowohl der Kunde wartet als auch der Verkäufer frei ist. Ist von diesen Bedingungen eine nicht erfüllt, so geschieht nichts. Wir lernen daraus: Eine Transition kann nur dann schalten, wenn alle ihre Vorgängerstellen mit mindestens einer Marke belegt sind. Das Schalten der Transition entzieht jeder Vorgängerstelle genau eine Marke. Dem nächsten Beispiel entnehmen wir, daß das Schalten einer Transition jeder ihrer Nachfolgerstellen genau eine Marke hinzufügt:




$t_1$ : Kunde betritt den Laden

$t_2$ : Kundenbedienung beginnt

$t_3$ : Kundenbedienung endet

$t_4$ : Kunde verläßt den Laden

---

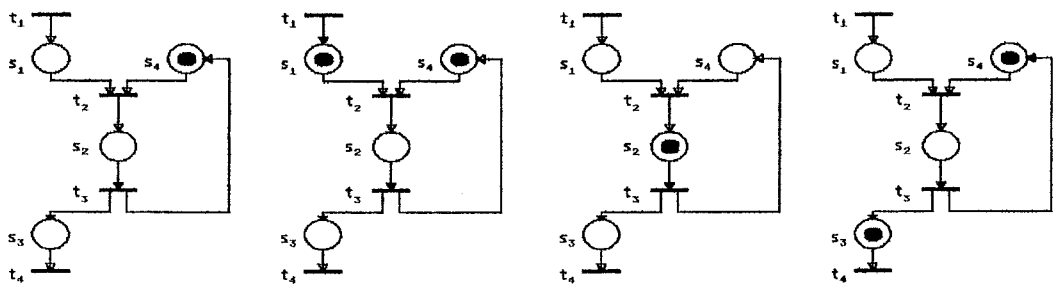
$s_1$ : Kunde wartet auf Bedienung

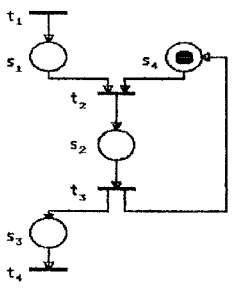
$s_2$ : Kunde wird bedient

$s_3$ : Kunde ist bedient

$s_4$ : Verkäufer ist frei

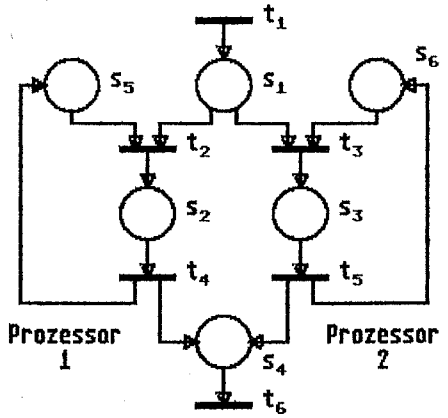
Interessant sind hier die Transitionen $t_1$ und $t_4$. Die Tatsache, daß $t_1$ keine Vorgängerstelle besitzt, bedeutet: $t_1$ ist immer schaltbereit, sie bildet gewissermaßen einen "Eingang" in jenes System, für das wir uns - mittels geeigneter Interpretation der Stellen und Transitionen - die Freiheit genommen haben, es als Laden mit einem Verkäufer zu sehen. $t_4$ andererseits ist ein "Ausgang": Falls diese Transition schaltet, so verschwindet eine Marke auf Nimmerwiedersehen. Eine mögliche Zustandsfolge ist:





Wie gesagt, dies ist nur eine von vielen möglichen Zustandsfolgen. Da wir im übrigen keinerlei restriktive Annahmen über die Anzahl der Marken auf einer Stelle getroffen haben, können sich auf einer Stelle durchaus mehrere Marken ansammeln. So ist nicht ausgeschlossen, daß $t_1$ eine zweites (drittes, viertes, ... ) Mal schaltet, bevor $t_2$ zum Zuge kommt. Ferner ist denkbar, daß sich Marken auf $s_1$ ansammeln, während eine Marke auf $s_2$ liegt, und so weiter.

In Verallgemeinerung unserer "Kunden - Verkäufer" Interpretation haben wir es hier mit einem sehr einfachen Prozeß-Modell zu tun: Ein Prozessor bearbeitet Aufträge, das ist alles. Nichts wird etwa darüber ausgesagt, in welcher Reihenfolge diese Aufträge ausgeführt werden oder welche Zeit einem einzelnen Auftrag gewidmet wird. Ohne die Berücksichtigung solcher (und anderer) Details wird ein mit zwei Prozessoren ausgestattetes System zur Auftragsbearbeitung durch das folgende Petrinetz modelliert:




$t_1$ : Auftrag trifft ein

$t_2$ : Bearbeitung durch Prozessor 1 beginnt

$t_3$ : Bearbeitung durch Prozessor 2 beginnt

$t_4$ : Bearbeitung durch Prozessor 1 endet

$t_5$ : Bearbeitung durch Prozessor 2 endet

$t_6$ : Auftragsbestätigung wird entnommen

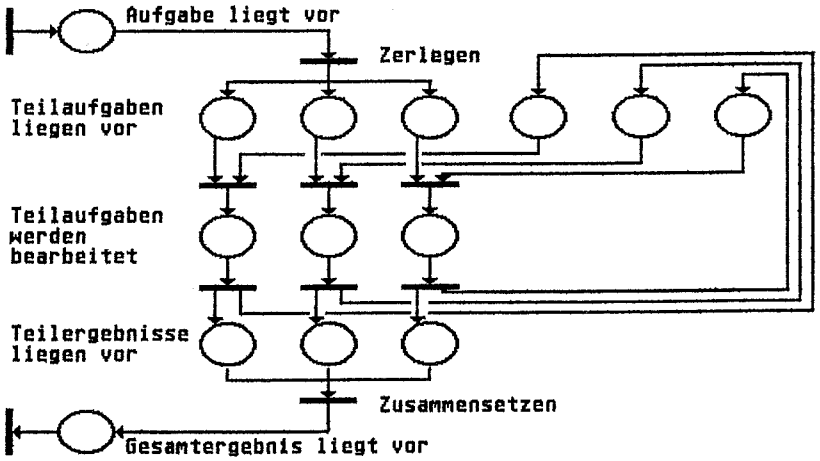
Immerhin zeigen diese Beispiele bereits einige wichtige *Aspekte der System-Modellierung* mit Hilfe von Petrinetzen. So fällt es nicht schwer,

- zyklische oder linear ablaufende Prozesse darzustellen, sowie
- Situationen, in denen sich zwei Ereignisse gegenseitig ausschließen (oder, wie man auch sagt, miteinander in Konflikt sind).

Letzeres findet sich im Modell des 2-Prozessor Systems: Falls $s_5$, $s_1$ und $s_6$ jeweils mit genau einer Marke belegt sind, so kann, wenn eine der beiden Transitionen $t_2$ und $t_3$ geschaltet hat, die jeweils andere Transition nicht mehr schalten. Dennoch zeigt das gleiche Netz auch, daß die durch die linken und rechten Netzteile repräsentierten Bearbeitungsvorgänge (*Prozesse*) unabhängig sind in dem Sinne, daß sie *parallel nebeneinander ablaufen* können: Falls Prozessor 1 arbeitet und Prozessor 2 frei ist, so kann dieser, wenn ein neuer Auftrag eintrifft, ohne Rücksicht auf seinen "Kollegen" Nr. 1 mit der Bearbeitung beginnen. (Der Leser mache sich dies durch geeignete Zustandsfolgen, entsprechend dem Mu-

ster des 1-Prozessor Systems, klar.) Gemäß Petrinetz Terminologie spricht man auch von *nebenläufigen* Prozessen.

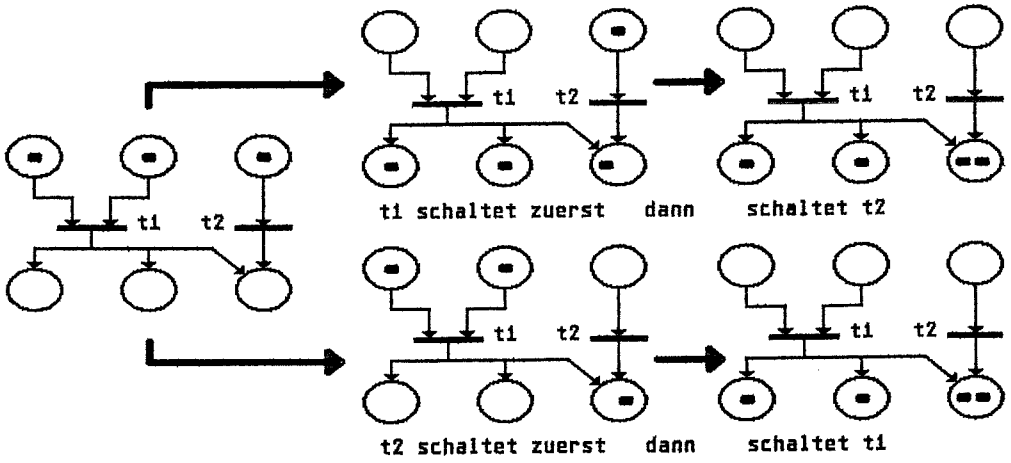
In unserem 2-Prozessor System müssen sich die Prozessoren bei der Bearbeitung eines Auftrags nicht "absprechen", sie brauchen sich nicht zu *synchronisieren.* In der *realen Welt* hingegen hat man es häufig mit zwar parallel ablaufenden, aber dennoch voneinander abhängigen Vorgängen zu tun. So kann das Ergebnis eines Vorgangs für die Fortsetzung eines anderen Vorgangs benötigt werden, oder die Erledigung einer Aufgabe erfolgt durch die parallele Erledigung von Teilaufgaben. Das folgende Netz zeigt eine entsprechende Modifikation des 1-Prozessor Systems: Hier wird eine Aufgabe in drei Teilaufgaben zerlegt, welche unabhängig voneinander bearbeitet werden. Über die Dauer der Teilbearbeitungen wird nichts ausgesagt. Doch es wird ganz klar, daß das Gesamtergebnis nicht hergestellt werden kann, bevor nicht sämtliche Teilresultate vorliegen.




Weitere Beispiele für die Modellierung des Zusammenspiels von Prozessen werden uns weiter unten begegnen.

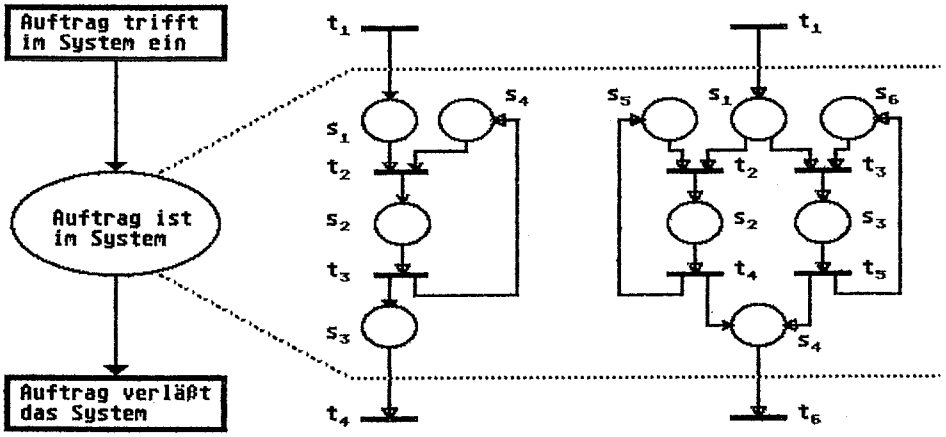
Von einigem Interesse bei der Modellierung von Phänomenen der realen Welt ist sicherlich die Frage, welche Rolle die mit dem Ablauf von Zeit verknüpften Gegebenheiten und Vorstellungen dabei spielen sollen. Petrinetze, soweit wir sie bisher kennengelernt haben, erlauben hierauf nur Antworten, welche ohne den Begriff einer *absoluten Zeit* auskommen müssen. Doch ist dies durchaus kein Mangel, ist doch Zeit (mitunter sogar in unserer eigenen sinnlichen Wahrnehmung) nicht anders als durch die *Aufeinanderfolge* (also die *relative* Lage) von Ereignissen gegeben. (Das Pendel, ein klassisches Instrument der Zeitmessung durch Zählung (!) von Ereignissen, führt uns dies ganz anschaulich vor Augen.) Die Ereignisse selbst und, mutatis mutandis, Transitionen in Petrinetzen haben keine *Dauer.* (Der Durchgang des Pendels durch die Null-Lage hat keine Dauer!) Die begriffliche Unterscheidung von Ereignis und Vorgang ist bei der Modellierung mit Petrinetzen daher sehr genau zu beachten: Ein Vorgang

wird erfahrbar als Folge von Ereignissen, welche in allgemeiner Form durch sprachliche Wendungen ausgedrückt werden wie "*Ein Vorgang beginnt* ", "*Ein Vorgang wird unterbrochen* ", "*Ein Vorgang endet* ", und so weiter. *Gleichzeitigkeit* von Ereignissen in einem System nebenläufiger Prozesse kann daher nur bedeuten, daß es zwischen diesen Ereignissen keine prädeterminierte vorher-nachher Ordnung gibt: Der Zustand, in welchem sich das System nach solchen Ereignissen befindet, ist unabhängig davon, in welcher Reihenfolge diese Ereignisse stattgefunden haben. Das folgende Beispiel möge dies illustrieren:

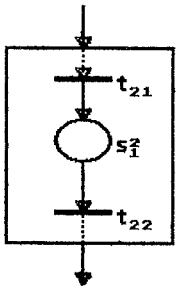



Vom links abgebildeten Anfangszustand aus ergibt jede Reihenfolge der Transitionen **t1** und **t2** den gleichen Endzustand. Gäbe es so etwas wie eine absolute Zeit, so könnten **t1** und **t2** zum exakt gleichen Zeitpunkt stattfinden, also im landläufigen Sinne "gleichzeitig" sein, ohne sich "gegenseitig ins Gehege zu kommen".
Natürlich darf es bei der Fülle der Literatur (s.o.) über Petrinetze nicht verwundern, daß es auch einige Ansätze gibt, Zeit und Zeitspannen (Dauer) als meßbare Größen bei der Modellierung einzubeziehen. Wir wollen und können dies im Rahmen eines knappen Überblicks jedoch nicht vertiefen.

Anders als die in Abschnitt 6.3.4 zu besprechenden graphischen Darstellungen sind Petrinetze vom Ansatz her nicht primär darauf ausgelegt, hierarchische Systembeschreibungen zu produzieren. Dennoch bieten sie die Möglichkeit, sowohl Situationen (repräsentiert durch Stellen) als auch Ereignisse (repräsentiert durch Transitionen) immer weiter zu verfeinern. Das 1-Prozessor System und das 2-Prozessor System (s.o.) können beide als Verfeinerungen des folgenden (ziemlich trivialen) Netzes aufgefaßt werden:




Sollte es im Modellierungsinteresse liegen, das Ereignis "Beginn der Auftragsbearbeitung", dem hier das Schalten der Transition $t_2$ (bzw. $t_3$) entspricht, im Detail zu studieren und in weitere, elementarere Ereignisse mit geeigneten "Zwischensituationen" aufzulösen, so steht auch dem nichts im Wege:



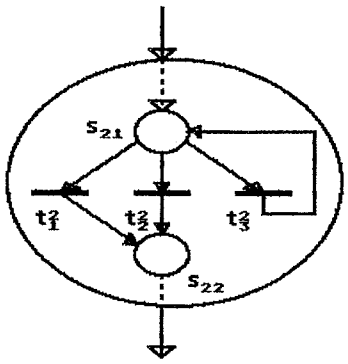

$t_{21}$: Beginn des Studiums der Auftragsdaten

$s_1^2$ : Studium der Auftragsdaten

$t_{22}$: Ende des Studiums der Auftragsdaten

En passant bemerken wir, daß es sich hier als nützlich erweisen könnte, zwei alternative Enden des Studiums der Auftragsdaten zuzulassen, eines, welches der Annahme und ein anderes, welches der Ablehnung des Auftrags gleichkommt. Der Leser ist aufgefordert, sich zu überlegen, in welcher Weise das 1-Prozessor System zu modifizieren ist, wenn auch diese Möglichkeit berücksichtigt werden soll.

Eine weitere Verfeinerung der obigen Netze wollen wir an der Stelle $s_2$ (bzw. $s_3$) vornehmen, die, wenn mit einer Marke belegt, der Situation "Auftrag wird bearbeitet" entspricht. Diese Situation wird, wie das folgende Bild zeigt, in einen etwas komplexeren Vorgang aufgelöst:




$s_{21}$: Bearbeitungsschritt ist im Gange

$s_{22}$: Bearbeitung ist beendet

---

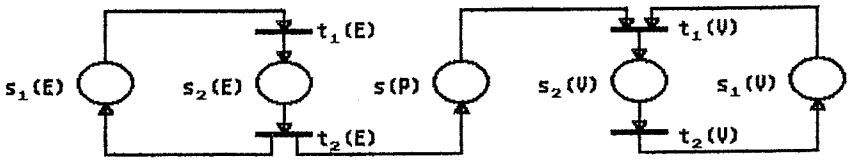
$t_1^2$ : Bearbeitung wird abgebrochen

$t_2^2$ : Letzter Bearbeitungsschritt wird beendet

$t_3^2$ : Nächster Bearbeitungsschritt wird begonnen

Nachdem wir nun - in groben Zügen - gesehen haben, wie Prozesse und einfache Systeme nebenläufiger Prozesse mit Hilfe von Petrinetzen dargestellt werden können, ist es an der Zeit zu erfahren, in welcher Weise Prozeßkommunikation und damit wesentlich komplexere Systeme nebenläufiger Prozesse im Rahmen dieses Formalismus zu behandeln sind. Wir beschränken uns dabei auf Fälle, die mit jenen zwei Grundformen der Kommunikation zu tun haben, welche gegen Ende des vorangegangenen Abschnitts bereits motiviert wurden: Kommunikation über sequentielle Medien (*Kanäle*) beziehungsweise Kommunikation mittels gemeinsam genutzter *Speicher*.

Der erste Fall betrifft die Beziehung zwischen zwei Prozessen, von denen sich der eine in der Rolle des *Erzeugers* (*Producer*) und der andere in der Rolle des *Verbrauchers* (*Consumer*) befindet. Der Verbraucher-Prozeß kann nur dann tätig werden, wenn ein noch unverbrauchtes Produkt des Erzeuger-Prozesses vorliegt. Dieser Zusammenhang wird durch das folgende Netz modelliert:




ERZEUGER

$s_1(E)$: Erzeuger ist untätig

$s_2(E)$: Erzeuger arbeitet

$t_1(E)$: Erzeugung beginnt

$t_2(E)$: Erzeugung endet

VERBRAUCHER

$s_1(V)$: Verbraucher ist untätig

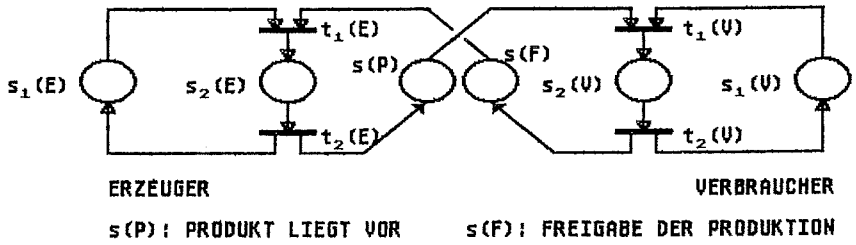
$s_2(V)$: Verbraucher arbeitet

$t_1(V)$: Verbrauch beginnt

$t_2(V)$: Verbrauch endet

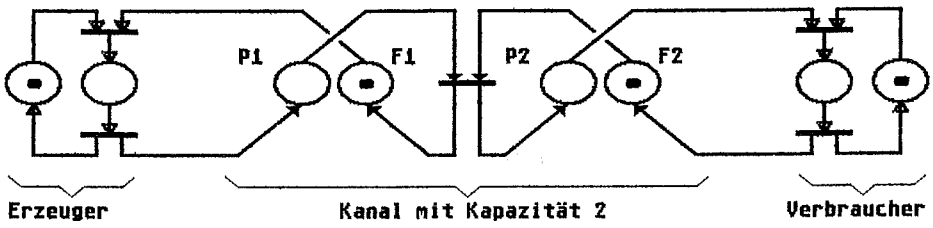
s(P): PRODUKT LIEGT VOR

Die Arbeit dieses Systems beginnt mit Marken auf $s_1(E)$ und $s_1(V)$. Das Modell impliziert offenbar keinerlei Restriktionen hinsichtlich der Geschwindigkeiten der beiden Prozesse. Beide können so schnell oder so langsam arbeiten wie sie wollen. Andererseits müssen Erzeuger und Verbraucher über ein Medium miteinander verbunden sein, welches den Transport der erzeugten Produkte gestattet, und es ist realistisch anzunehmen, daß dieses Medium nur ein beschränktes Fassungsvermögen hat. Es besteht die Möglichkeit, daß der Erzeuger schneller produziert als der Verbraucher konsumiert. Wenn nun jener Kanal vom Verbraucher kontrolliert wird, so benötigen wir einen Mechanismus, der es dem Verbraucher gestattet, dem Erzeuger zu signalisieren, daß der "Kanal gerade voll" und die Produktion daher zu stoppen ist. Für den Fall, daß die Kanalkapazität 1 beträgt, stellt das folgende Netz mit jeweils genau einer Marke auf $s_1(E)$, $s_1(V)$ und $s(F)$ einen solchen Mechanismus dar:



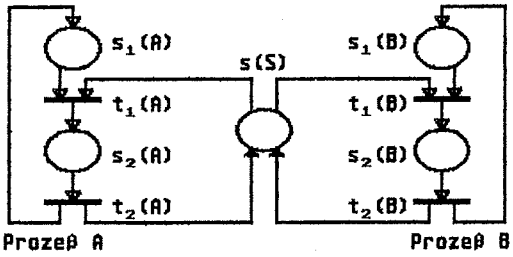

Der Erzeuger kann dann mit der Produktion beginnen, und erst wenn der Verbraucher ein Produkt konsumiert hat, wird wiederum eine Marke auf **s(F)** geschoben. Der Erzeuger wird damit gezwungen, mit der gleichen Geschwindigkeit zu produzieren, mit der der Verbraucher konsumiert. (Der Leser überlege sich, was es bedeutet, wenn **s(F)** zum Start des Systems mit mehr als einer Marke belegt ist.) Die Beziehung zwischen den beiden Prozessen ist damit durch ein sehr einfaches *Kommunikationsprotokoll* geregelt.

Wir gingen oben davon aus, daß der Verbraucher die Kontrolle über den Kanal ausübt. Genausogut freilich könnten wir annehmen, daß der Kanal von einem eigenen, ihm zugeordneten Prozeß kontrolliert wird, der keine andere als diese Aufgabe hat. Wir könnten wieder verlangen, daß der Kanal eine endliche Kapazität hat, und natürlich sollte dieser Prozeß auch dafür sorgen, daß die charakteristische Kanaleigenschaft erfüllt ist: daß die Produkte nur in genau der Reihenfolge entnommen werden, in der sie in den Kanal gesteckt wurden (Rohrpost!). Eine solche Konfiguration wird durch das folgende, mit einer geeigneten Anfangsmarkierung versehene Netz modelliert:




Es sei dem Leser empfohlen, die Dynamik dieses Netzes mit Hilfe von Münzen oder ähnlichen Gegenständen als Marken in spielerischer Manier nachzuvollziehen. Er wird feststellen, daß sich die Stellen **P1** und **P2** als "Fächer" zur Aufnahme von Produkten in der Reihenfolge ihrer Erzeugung deuten lassen, während Marken in **F1** und **F2** jeweils signalisieren, daß die korrespondierenden Fächer frei sind. Der Kanalprozeß verwaltet diese Fächer, indem er den Inhalt von Fach 1 nach Fach 2 schiebt und damit Fach 1 zur erneuten Belegung freigibt. Das Beispiel zeigt, daß sich Petrinetze nicht nur zur Modellierung eines Systems "auf hohem Niveau" eignen, sondern durchaus auch zur Darstellung sehr feiner Strukturen.

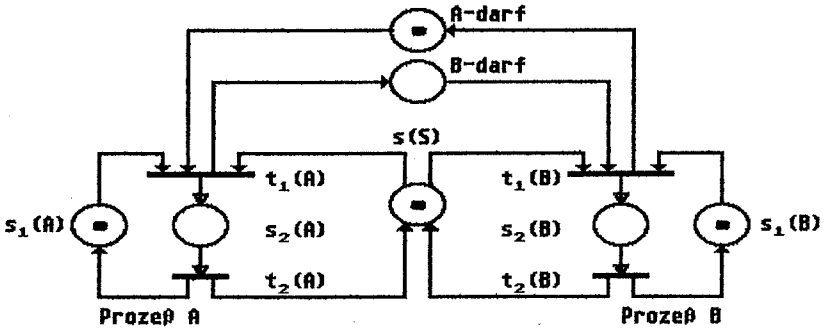
Unsere zweite Fallskizze hat zwei (oder mehr) Prozesse zum Gegenstand, die für ihren Ablauf auf ein *Betriebsmittel* (*Ressource*) angewiesen sind, welches nicht gleichzeitig von mehr als einem Prozeß benutzt werden darf. Ebenso wie das Erzeuger-Verbraucher Problem ist auch das Problem des wechselseitigen Ausschlusses beim Zugriff auf Ressourcen paradigmatisch insofern, als es hinter zahlreichen, in der Praxis von Systemanalyse und Systementwurf auftretenden Fragestellungen steckt. Wir präsentieren zunächst eine Lösung, in der sich zwei Prozesse mittels einer *Synchronisationsvariablen* verständigen, auf welche beide sowohl lesend als auch schreibend zugreifen dürfen (Kommunikation via Speicher!):




```
s1(X): Prozeß X wartet auf die Verfügbarkeit des Betriebsmittels
s2(X): Prozeß X benutzt das Betriebsmittel
-------------------------------------------------------------
t1(X): Prozeß X beginnt die Benutzung des Betriebsmittels
t2(X): Prozeß X beendet die Benutzung des Betriebsmittels
-------------------------------------------------------------
s(S):  BETRIEBSMITTEL IST VERFÜGBAR
```

Wir behaupten: Beginnt das System mit Marken auf den Stellen $s_1(A)$, $s_1(B)$ und $s(S)$, so werden $s_2(A)$ und $s_2(B)$ niemals gleichzeitig belegt sein. Und dies ist genau das, was wir wollen: daß nämlich die beiden Prozesse niemals simultan auf das Betriebsmittel zugreifen. (Der Leser möge sich vorerst damit begnügen, diese Behauptung mit Hilfe des Markenspiels zu verifizieren.)

Bei genauerem Hinsehen fällt uns freilich auf, daß die vorgeschlagene Lösung etwas "chaotisch" ist. Die Transitionen $t_1(A)$ und $t_1(B)$ sind nämlich offenbar im Konflikt miteinander, und es gibt keinerlei Regelung dafür, welcher Prozeß denn nun an das Betriebsmittel heran darf, wenn beide es benutzen wollen. Diesen Mißstand wollen wir mit der folgenden Modifikation unseres Netzes wenigstens insoweit beheben, als sich die Prozesse alternierenden Zugang zu der Ressource gestatten:



Mit einer Marke auf der Stelle **A-darf** (**B-darf**) wird dem Prozeß A (B) bedeutet, daß er nun an der Reihe ist. A (B) wird dann seinerseits das Benutzungsrecht an B (A) abgeben. Beide Prozesse sind in diesem Modell völlig gleichberechtigt. Natürlich handelt es sich auch hier um die Formalisierung eines Kommunikationsprotokolls. Aber gewiß nicht um der Weisheit letzter Schluß, denn es könnte ja zum Beispiel sein, daß einer der beiden Prozesse, aus welchen Gründen auch immer, das Betriebsmittel sehr viel häufiger benötigt als der andere. Der Leser mag sich daher an der Konstruktion "gerechterer" Zuteilungs-Schemata (bei Wahrung des wechselseitigen Ausschlusses bei der Benutzung, versteht sich!) versuchen und so ein Gefühl für die Delikatheit des Problems entwickeln. Auch die Weiterentwicklung der Lösungen für dieses Problem zeigt im übrigen die Stärke des Petrinetz-Formalismus bei der immer feineren Zeichnung von Details.

Eine weitere - und wesentliche - Stärke ist gewiß auch in der Tatsache zu sehen, daß Petrinetze sich in vielfältiger Weise mit mathematischen Hilfsmitteln, sowohl klassischen als auch speziell entwickelten, analysieren lassen. Viele Fragestellungen, die sich ganz zwangsläufig bei beziehungsweise nach der Konstruktion von Petrinetz-Modellen für natürliche oder künstliche Systeme ergeben, lassen sich in einfacher Weise in mathematische Form kleiden. Auf den vorangegangenen Seiten sind uns schon mehrere Beispiele für solche Fragestellungen begegnet. Die folgende (kleine) Sammlung enthält außerdem weitere:

- Kann ein bestimmter Netz-Zustand, also eine bestimmte Markierung der Stellen, überhaupt erreicht werden?
- Kann eine Stelle **s1** eine Marke tragen, falls eine andere Stelle **s2** bereits markiert ist?
- Ist die Anzahl der Marken auf einer Stelle beschränkt?
- Bleibt die Anzahl der Marken im Netz oder in einem bestimmten Bereich des Netzes konstant?
- Kann, bei einer gegebenen Anfangsmarkierung, eine bestimmte Transition **t** jemals schalten?
- Kann, von einer gegebenen Anfangsmarkierung aus, ein Zustand erreicht werden, in dem keine Transition mehr schaltfähig ist?
- Und viele, viele mehr.

Voraussetzung dafür, diese (und, wie gesagt, viele ähnliche) Fragen in mathematische Form zu übersetzen, ist die mathematische Definition von Petrinetzen und der ihnen eigenen Begriffe selbst. Mit einem kleinen Vorgeschmack hierauf wollen wir diesen Abschnitt abschließen. (Es bleibe der Phantasie des Lesers überlassen, die zahlreichen Möglichkeiten sowohl für Verallgemeinerungen als auch für Spezialisierungen des Formalismus zu entdecken. Und für den Fall, daß

ihm seine Phantasie nicht ausreichend scheint, sei er an die Literaturhinweise zu Beginn dieses Abschnitts erinnert.)

**Definition:** Ein *Petrinetz* ist gegeben durch ein 4-Tupel $\mathbf{P} = (S, T, pre, post)$ mit:

- $S$: Menge der *Stellen*,
- $T$: Menge der *Transitionen*,
- $pre, post : T \to 2^S$ Abbildungen, welche jeder Transition $t$ die Menge $pre(t)$ der *Vorgängerstellen* und die Menge $post(t)$ der *Nachfolgerstellen* zuordnen.

($2^S$ bezeichnet die Potenzmenge von $S$. In der Terminologie der Graphentheorie ist ein Petrinetz ein *bipartiter* Graph mit den beiden Knotenklassen *Stellen* und *Transitionen*.)

**Definition:** Sei $\mathbf{P} = (S, T, pre, post)$ ein Petrinetz. Eine Abbildung $\mu : S \to \mathbb{N} \cup \{0\}$ heißt *Markierung* beziehungsweise *Zustand* von $\mathbf{P}$.

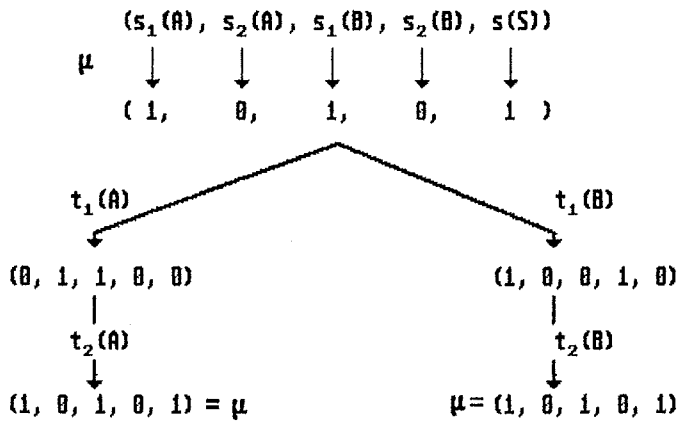
Damit sind wir in der Lage, die *Schaltregel* exakt anzugeben:

**Schaltregel**: Sei $\mathbf{P} = (S, T, pre, post)$ ein Petrinetz, sei $t \in T$ eine Transition und $\mu$ eine Markierung von $\mathbf{P}$. $t$ kann *schalten* bei $\mu$, wenn für jedes $s \in pre(t)$ gilt: $\mu(s) > 0$. Schaltung von $t$ überführt $\mu$ in eine Markierung $\mu'$ des Netzes $\mathbf{P}$ wie folgt:

$$\mu'(s) = \mu(s) - 1 \text{ für alle } s \in pre(t)$$
$$\mu'(s) = \mu(s) + 1 \text{ für alle } s \in post(t).$$

(Wir schreiben auch: $\mu \xrightarrow{t} \mu'$.)

**Definition:** Sei $\mathbf{P} = (S, T, pre, post)$ ein Petrinetz und $\mu$ eine Markierung von $\mathbf{P}$. Eine Markierung $\mu'$ von $\mathbf{P}$ heißt *erreichbar* von $\mu$ aus, falls es eine geeignete Folge $\mu = \mu_0 \xrightarrow{t_1} \mu_1 \to \ldots \xrightarrow{t_n} \mu_n = \mu'$ von Schaltungen gibt, welche $\mu$ schrittweise in $\mu'$ überführt. Wir notieren die von $\mu$ aus erreichbaren Markierungen (bzw. Zustände) als $\mathbf{R}(\mu) = \{\mu' \mid \mu' \text{ ist erreichbar von } \mu\}$.




Mit diesen Definitionen ist es nun leicht, die über das System der beiden, sich eine Ressource teilenden Prozesse A und B aufgestellte Behauptung (s.o.) zu beweisen. Mit der Aufzählung $(s_1(A), s_2(A), s_1(B), s_2(B), s(S))$ der Stellen des Netzes lautet die Anfangsmarkierung: $(1, 0, 1, 0, 1)$. Die Menge der möglichen Folgezustände ergibt sich nach Anwendung der Schaltregel. Sie kann als Baum (*Erreichbarkeitsbaum*) dargestellt werden, dessen Knoten die Markierungen und dessen Kanten

die jeweils bei einer Markierung schaltfähigen Transitionen repräsentieren.

Also:

$\mathbf{R}(\mu) = \{\mu' \mid \mu' \text{ ist erreichbar von } \mu\} = \{ (1, 0, 1, 0, 1), (0, 1, 1, 0, 0), (1, 0, 0, 1, 0)\}$

Ein Zustand der Form (-, 1, -, 1, -) ist darin nicht enthalten.

Der Leser versuche sich in der gleichen Weise an den ebenfalls oben betrachteten Erzeuger-Kanal-Verbraucher Systemen.

### 6.3.3 "Jackson System Development"

In seinem Buch *"System Development"* ([JA2]) hat Michael Jackson einen Ansatz zur Realisierung von rechnergestützten Systemen entwickelt, welcher die in Abschnitt 6.3.1 vorgestellte allgemeine Modell-Struktur in sehr direkter Weise konkretisiert. In der Tat: die Grundidee des *Jackson System Development* (im folgenden kurz JSD genannt) ist uns bereits bekannt:

Ein rechnergestütztes System, welches

- Informationen über Realwelt-Prozesse liefern
  und / oder
- für Realwelt-Prozesse Dienste leisten soll,

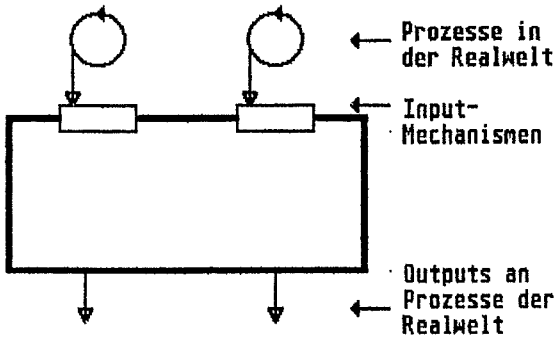
muß in geeigneter Weise

- auf einem Modell dieser Realwelt-Prozesse beruhen
  und
- mit den Realwelt-Prozessen in Verbindung stehen.

(Mit dem Begriff *rechnergestütztes System* meinen wir ein sowohl aus Software als auch aus Hardware bestehendes technisches System.) Im vorliegenden Abschnitt werden wir die dem JSD eigene Art der Erfüllung dieser Forderung anhand von zwei Beispielen skizzieren. Zwei Einschränkungen sind dabei zu beachten: Erstens halten wir uns nicht sklavisch an die von Jackson in [JA2] vorgegebene Terminologie. Zweitens verzichten wir darauf, die von Jackson vorgeschlagene Gliederung des Prozesses der Software-Entwicklung in allen Einzelheiten zu übernehmen. Unser Hauptanliegen ist vielmehr einerseits, die speziellen, am Anfang des Entwicklungsprozesses zu beachtenden Modellierungsaspekte herauszuarbeiten und andererseits die Benutzung der von Jackson zur Modelldokumentation eingeführten graphischen Darstellungselemente zu demonstrieren. Doch auch dabei werden wir teilweise eigene Wege gehen. So ersetzen wir - wie bereits in Kapitel 5 - die von Jackson erstmals in [JA1] und dann auch im JSD verwendeten Formalismen (baumartige Diagramme sowie einen speziellen Pseudocode) zur Beschreibung von Daten- und Ablaufstrukturen durch Klammerdiagramme, die wir der Einheitlichkeit und ihrer größeren Ausdruckskraft wegen bevorzugen. Zu bemerken ist schließlich, daß das JSD keineswegs als Weiterentwicklung der von Jackson in [JA1] propagierten Me-

thode des *datenstrukturierten Programmentwurfs* anzusehen ist. Zwar spielt im JSD (das in seiner Gesamtheit auch Anleitungen zur Systemimplementierung enthält) die Struktur von Datenflüssen eine große Rolle, doch zielt die Methode - im Gegensatz zum datenstrukturierten Entwurf - in erster Linie auf die Bewältigung der Probleme des Programmierens im Großen, also auf Systemspezifikation und -entwurf.

Den zur Erfüllung der oben formulierten Forderung an rechnergestützte Systeme notwendigen Rahmen wollen wir durch das folgende Bild darstellen:




"Rahmen" ist hier ganz wörtlich zu nehmen: Es ist die fett gezeichnete Umrandung, innerhalb derer sich das Systemgeschehen abzuspielen hat und welche somit das Areal für Systemspezifikation und -entwurf abgrenzt. Wenn es also heißt, daß das System auf einem geeigneten Modell der Realwelt beruhen soll, so muß sich innerhalb dieser fetten Umrandung etwas befinden, was dieses Modell repräsentiert. Dieses "etwas" sind nun genau die Abstraktionen derjenigen Realwelt-Prozesse, um die es dem Systementwickler bei seiner Arbeit geht. Es sind die infolge hinreichender Analyse der Realwelt-Prozesse gefundenen *Modellprozesse.* Die wichtigsten Schritte einer solchen Analyse sind:

- Bestimmung der relevanten Realwelt-Objekte
- Bestimmung der relevanten Ereignisse (bzw. Aktivitäten), die mit diesen Objekten zu tun haben;
- Bestimmung der Daten, die diesen Ereignissen (bzw. Aktivitäten) zugeordnet sind;
- Bestimmung der sequentiellen Beziehungen zwischen Ereignissen (bzw. Aktivitäten).

Die Nachbildung der so analysierten Realwelt-Prozesse im Modell wird auch als *Simulation* bezeichnet.

Auf der unteren Seite des schwarzen Kastens müssen den nach außen gerichteten Pfeilen Prozesse gegenüberstehen, welche die der Realwelt zu leistenden Dienste erbringen, Prozesse also, durch die letztlich die *Funktionalität* des Systems definiert wird. Mit diesen *funktionalen Prozessen* werden

- Informationen über die Modellprozesse (die im System simulierte Realwelt!) beschafft,
- Informationen an in der Realwelt ablaufende Prozesse geliefert und damit eventuell
- Realwelt-Prozesse gesteuert.

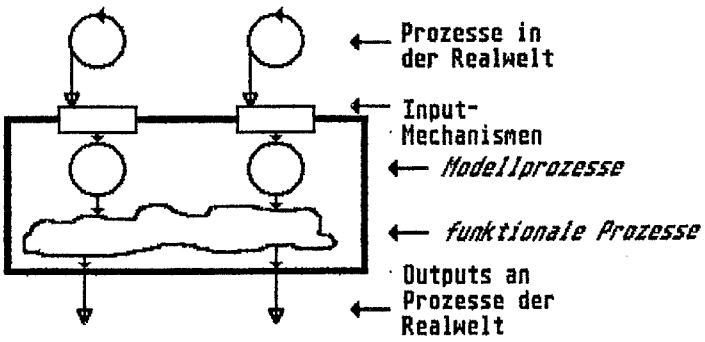
Zusammengefaßt verläuft die Spezifikation eines rechnergestützten Systems gemäß JSD also in drei Hauptschritten:

(1) Analyse der relevanten Realwelt-Prozesse und ihre Simulation durch *Modellprozesse*. (Diese bilden den "Kern" der Spezifikation und später des funktionierenden Systems.)

(2) Definition der den Output erzeugenden *funktionalen Prozesse*.

(3) Festlegen der *Verbindungen* zwischen Realwelt-Prozessen, Modellprozessen und funktionalen Prozessen.

Nach diesen Schritten umschließt der Rahmen ein *Netzwerk* von miteinander *kommunizierenden Prozessen*:




Der erste Realwelt-Ausschnitt, auf den wir die skizzierte Vorgehensweise anwenden wollen, ist eine Leihbücherei, welche als Verein organisiert ist (vgl. das Bibliothekssystem in [CAM]). Nur wer Mitglied des Vereins ist, darf Bücher ausleihen oder zur Ausleihe vormerken (reservieren) lassen. Welches sind die relevanten Objekte dieses Realwelt-Ausschnitts? Wie können wir sie von den nicht-relevanten Objekten unterscheiden? Eine allgemeine Antwort hierauf wird uns durch den bei der Analyse zu unternehmenden zweiten Schritt nahegelegt: Relevant sind diejenigen Objekte, welche passiv oder aktiv an Ereignissen beteiligt sind. Doch was ist ein Ereignis? Diese Frage wird beim JSD ähnlich beantwortet wie im Rahmen der Petrinetze: Ereignisse sind als zeitlos denkbare Übergänge von einem (Objekt-)Zustand in einen anderen zu verstehen. Sie haben keine Dauer, sondern sind Zeitpunkten zugeordnet, welche aufeinanderfolgen. Und so wie im Petrinetz eine Transition ohne Vorgängerstelle ein "Objekt" (eine Marke) erzeugen kann, so werden auch im JSD Akte der Erzeugung von Objekten als Ereignisse angesehen.

Doch nun zum Beispiel: Die unter dieser Perspektive relevanten Objekte der Leihbücherei sind:

BUCH, MITGLIED, RESERVIERUNG.

Daß BUCH und MITGLIED relevante Objekte unseres Realwelt-Ausschnitts sind, erscheint natürlich und unbestreitbar. Anders sieht es auf den ersten Blick für das "Objekt" RESERVIERUNG aus. Vor dessen Begründung jedoch wollen wir zunächst die mit den Objekten BUCH und MITGLIED verbundenen Ereignisse aufzählen. Wir geben dabei gleichzeitig die den Ereignissen zugeordneten Daten an. Sämtliche Ereignisse können im übrigen als Folgen irgendwelcher *Aktionen* interpretiert werden. (Dabei ist nicht von Interesse, wer der Agent dieser Aktionen ist. Da - wie gesagt - ein Objekt sowohl passiv als auch aktiv an Ereignissen beteiligt sein kann, kommt es natürlich selbst auch als Agent in Frage.) Zur Hervorhebung dieser Tatsache werden wir die Ereignisse durch Verben im Infinitiv bezeichnen.

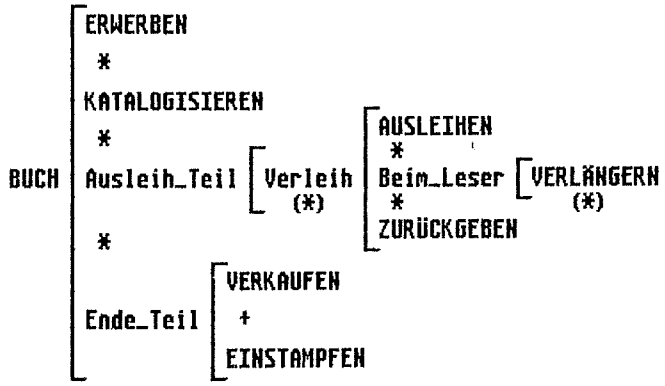
Die Ereignisse/Aktionen im "Leben eines BUCHes", für die wir uns interessieren, und die jeweils zugehörigen Daten sind:

| | |
|---|---|
| **ERWERBEN** | **(datum, titel, autor, ISBN, . . . )** |
| **KATALOGISIEREN** | **(buch-id-nr, datum, klassifikation, . . . )** |
| **AUSLEIHEN** | **(datum, ausleiher, . . . )** |
| **VERLÄNGERN** | **(datum, . . . )** |
| **ZURÜCKGEBEN** | **(datum, . . . )** |
| **VERKAUFEN** | **(datum, käufer, preis, . . . )** |
| **EINSTAMPFEN** | **(datum, . . . )** |

Das BUCH ist im übrigen ein Beispiel für ein passives Objekt, während das Objekt MITGLIED die für es relevanten Aktionen selbst auslöst. Die folgenden (und zum besseren Verständnis mit kurzer Erklärung kommentierten) Aktionen sind für uns von Interesse:

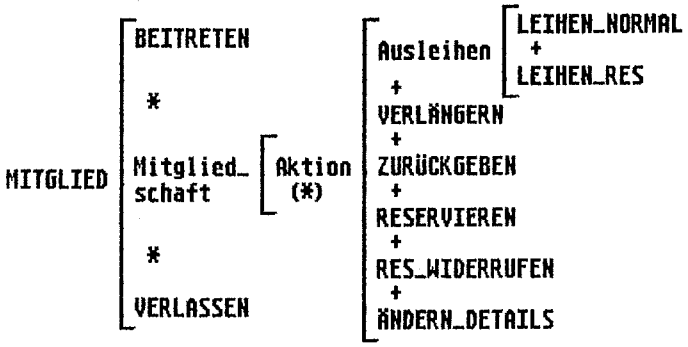
| | |
|---|---|
| **BEITRETEN** | **(mitglieds-nr, name, anschrift, datum, ... )** |
| **ÄNDERN-DETAILS** | **Ändern der Daten des Mitglieds ( ... )** |
| **LEIHEN-NORMAL** | **Ausleihaktion ohne vorherige Reservierung (buch-id-nr, datum, ... )** |
| **RESERVIEREN** | **Das Mitglied läßt sich für ein Buch vormerken (res-id, buch-id-nr, datum, ... )** |
| **RES-WIDERRUFEN** | **Das Mitglied läßt eine Vormerkung löschen (res-id, datum, ... )** |
| **LEIHEN-RES** | **Ausleihaktion mit vorheriger Reservierung (res-id, datum, ... )** |
| **VERLASSEN** | **Ende der Mitgliedschaft (datum, ... )** |

Die sequentiellen Beziehungen zwischen den Ereignissen im "Bibliotheksleben" eines Buches werden durch das folgende Klammerdiagramm veranschaulicht:




Wir erinnern uns: "*" bedeutet "danach folgt" und "+" heißt "oder" (vgl. die Abschnitte 3.3.2 und 5.1.1). Das "Bibliotheksleben" eines Buches beginnt also mit seinem ERWERB, danach folgt seine KATALOGISIERUNG. Ist diese erledigt, kann das Buch ausgeliehen werden. Es beginnt also eine Folge von "Verleih-Episoden", von denen jede die Struktur "AUSLEIHEN, Beim Leser, RÜCKGABE" hat. Während das Buch beim Leser ist, kann die Leihfrist mehrmals verlängert werden. Ebenso wie die Folge von Verleih-Episoden leer sein kann, ist es natürlich auch möglich, daß ein Leser das Buch ohne Verlängerung der Leihfrist zurückgibt. Dem "Ausleih-Teil" des Buchlebens folgt das Ende des Buches, welches entweder durch VERKAUFEN oder durch EINSTAMPFEN gegeben ist.

Die Bibliotheks-"Lebensgeschichte" eines Mitglieds wird durch das folgende Diagramm erzählt:




Offenbar gibt es unter den für Bücher und Mitglieder relevanten Aktionen solche, die nur für die Vertreter dieser jeweiligen *Objektklassen* von Bedeutung sind (ERWERBEN, KATALOGISIEREN ... von Büchern, BEITRETEN, VERLASSEN ... von Mitgliedern). Andererseits gibt es Aktionen, welche das "Leben" eines Buches mit dem eines Mitglieds verknüpfen (AUSLEIHEN, ZURÜCKGEBEN, VERLÄNGERN). Gewissermaßen "in der Luft" hängen die Akti-

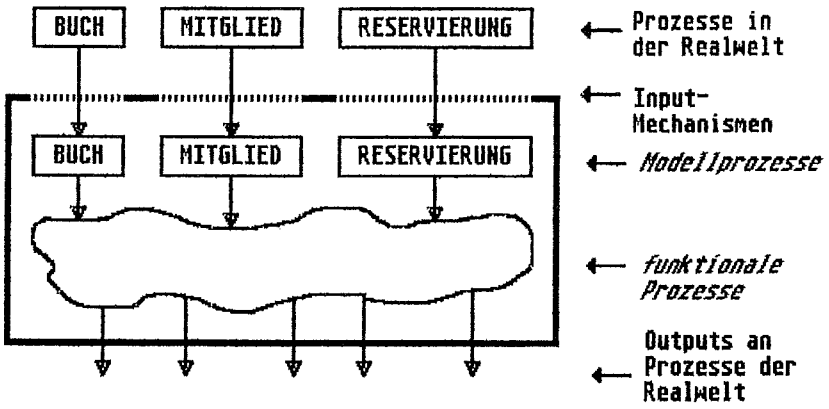
onen RESERVIEREN, LEIHEN-RES und RES-WIDERRUFEN. Zwar haben auch diese Aktionen mit Büchern zu tun, doch für das "Leben" eines Buches spielt es offenbar keine Rolle, ob es mit oder ohne vorangegangener Reservierung ausgeliehen wird. Aus der "Sicht eines Buches" ist es völlig gleichgültig, ob ein Mitglied sich seiner per LEIHEN-NORMAL oder per LEIHEN-RES bemächtigt. Wir wollen daher den Lebenslauf eines Buches nicht mit solchen für ihn unwichtigen Dingen belasten. Im Rahmen einer mitgliederfreundlichen Verwaltung der Bibliothek allerdings ist der Unterschied zwischen normaler Ausleihe und Ausleihe nach vorheriger Reservierung durchaus von einiger Bedeutung. Der Bibliotheksverwalter wird ihn kennen müssen, um eventuell die Vormerkung auf ein Buch, welche durch die RESERVIEREN-Aktion eines Mitglieds zustandekam, wieder zu löschen. Dies sollte genügen, um die Notwendigkeit des dritten *Objekttyps* in unserem Bibliothekssystem zu rechtfertigen: der Vormerkung oder, um die weiter oben bereits eingeführte Bezeichnung zu verwenden, der RESERVIERUNG. Das Leben einer RESERVIERUNG ist sehr einfach:

```
                ┌ RESERVIEREN
RESERVIERUNG    │  *   ┌ LEIHEN_RES
                │ Ende │ +
                └      └ RES_WIDERRUFEN
```

Außer durch die Aktion LEIHEN-RES kann eine Reservierung natürlich auch dadurch beendet werden, daß sie rückgängig gemacht (widerrufen) wird.

Wir haben damit die drei wesentlichen, in unserer Bibliothekswelt ablaufenden Prozesse beschrieben. Genauer gesagt: Wir haben *Prozeßtypen* beschrieben, zum Beispiel den Prozeßtyp BUCH. Das "Leben" jedes einzelnen Buches (von seinem Erwerb durch die Bibliothek an) ist ein Prozeß dieses Typs. Er kann lange dauern. Alles was wir von einem Prozeß wahrnehmen, sind die Aktionen, welche den *Prozeßzustand* ändern, und - eventuell mit Hilfe geeigneter Instrumente - der Prozeßzustand selbst. Die Aktionen können zeitlich weit auseinander liegen, und im Zeitraum zwischen zwei aufeinanderfolgenden Aktionen ist der Prozeß gewissermaßen unterbrochen (*suspendiert*). Was aber ist der *Zustand* eines Prozesses? Nun, bei den obigen Auflistungen von Aktionen/Ereignissen haben wir implizit zum Ausdruck gebracht, daß zu jedem Prozeßtyp auch eine Kollektion von Datentypen gehört, von denen manche die für einen Prozeß des gegebenen Typs konstanten und andere die für einen Prozeß des gegebenen Typs variablen Daten definieren. Die zur Aufnahme der variablen Daten eines Prozesses bestimmten (in irgendeiner Form vorhandenen) Speicher nennen wir die *Variablen* des Prozesses. Der Zustand eines Prozesses ist gegeben durch den aktuellen Inhalt seiner Variablen und durch die Menge der als unmittelbare Folgeaktionen in Frage kommenden Prozeß-Aktionen.

Im *Rahmen* (s.o.!) des rechnergestützten Bibliotheks-System müssen den drei Realwelt-Prozeßtypen BUCH, MITGLIED und RESERVIERUNG jeweils Modell-prozeßtypen (*Simulationen*) entsprechen:




Gegenüber den weiter oben gezeichneten "Rahmenbildern" haben wir uns hier eine "syntaktische" Modifikationen erlaubt, die wir auch in Zukunft beibehalten werden: die Repräsentation von Prozeßtypen und Prozessen durch Rechtecke. Die Input-Mechanismen, welche Realwelt und Modell verbinden, sind noch nicht näher spezifiziert. Es handelt sich hier, wie übrigens dann auch innerhalb des Rahmens, um Konstrukte, die der Kommunikation zwischen Prozessen dienen. In guter Übereinstimmung mit unseren Bemerkungen gegen Ende des Abschnitts 6.3.1 sieht das JSD zwei Arten der Prozeß-Kommunikation vor: Prozesse können miteinander kommunizieren, indem

- entweder ein Prozeß (P1) dem anderen (P2) über einen Kanal (K) Mitteilungen sendet,


- oder indem ein Prozeß (P1) dem anderen (P2) "gestattet", den Inhalt aller oder einiger der Zustandsspeicher (*Variablen* V) von P1 zu lesen.

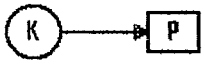

Bevor wir das Beispiel "Bibliothekssystem" fortsetzen, wollen wir uns mit der Bedeutung der soeben eingeführten graphischen Notation wiewohl informell, so doch in etwas größerem Detail vertraut machen. Die Unterscheidung zwischen einem Exemplar (eines Prozeß-, Kanal-, Variablen-Typs) und seinem Typ wird sich dabei aus dem konkreten Zusammenhang ergeben. Die Kommunikationsbeziehungen zwischen einem oder mehreren Exemplaren eines Typs und einem

oder mehreren Exemplaren eines anderen Typs können im übrigen als "1-zu-1", "1-zu-Viele", "Viele-zu-1" oder "Viele-zu-Viele" charakterisiert werden.

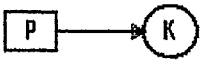

Prozeß(-Typ).


Kanal(-Typ): ein nach dem Rohrpost-Prinzip (First-In-First-Out, FIFO) funktionierender Mechanismus unbeschränkter Kapazität für den Transport von Daten/Signalen.


Variablen(-Typ): Zustandsspeicher eines Prozesses bzw. Prozeßtyps; ein alltägliches Analogon ist die Anzeigetafel.

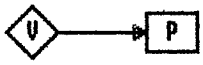



Der Prozeß P entnimmt Input aus dem Kanal K. Falls K leer ist, so kann P nicht weiterlaufen und wird unterbrochen (*suspendiert*, s.o.).

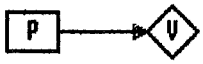



Der Prozeß P liefert Output an den Kanal K.


Der Prozeß P1 sendet Nachrichten (Daten bzw. Signale) an den Prozeß P2. Alle Nachrichten erreichen P2. ("1-zu-1")




Der Prozeß P hat lesenden Zugriff auf die Variable V.

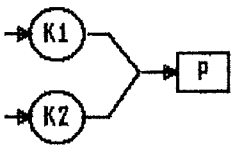



Der Prozeß P hat schreibenden Zugriff auf die Variable V.


Der Prozeß P1 zeigt dem Prozeß P2 in V seinen Zustand ganz oder teilweise an. Will P2 alle von P1 angezeigten Zustände erfahren, so muß er schnell genug arbeiten. ("1-zu-1")


P2 ist ein Prozeß-Typ. Jedem Exemplar von P2 ist ein Kanal (vom Typ K) zugeordnet. P1 kann aus diesen Kanälen denjenigen selektieren, über den er senden will. ("1-zu-Viele")


P2 ist ein Prozeß-Typ. Jedes Exemplar hat Lesezugriff auf die Variable V. ("1-zu-Viele")




Der Prozeß P entnimmt Inputs von K1 und K2 entsprechend der Reihenfolge ihrer Ankunft. Man sagt: P *mischt* die Kanäle K1 und K2. Sind beide Kanäle leer, so kann P nicht weiterlaufen und wird unterbrochen (s.o.!).


P1 ist ein Prozeß-Typ. Jedem Exemplar von P1 ist ein Kanal (vom Typ K) zugeordnet.
P2 mischt die Kanäle vom Typ K. ("Viele-zu-1")


P1 ist ein Prozeß-Typ. Jedem Exemplar von P1 ist eine Variable (vom Typ V) zugeordnet.
P2 hat auf diese selektiven Lesezugriff.
("Viele-zu-1")


P1 und P2 sind Prozeß-Typen. Jedes Exemplar von P1 kann über Kanäle (vom Typ K) Nachrichten an mehrere Exemplare von P2 schicken.
Jedes Exemplar von P2 kann Nachrichten von mehreren Exemplaren von P1 empfangen - und es mischt diese. ("Viele-zu-Viele")


P1 und P2 sind Prozeß-Typen. Jedes Exemplar von P2 hat Lesezugriff auf die Variablen (vom Typ V) der Exemplare von P1. ("Viele-zu-Viele")

Versetzen wir uns nun in die Lage des Verwalters der Bibliothek. Er benötigt, letztlich zum Nutzen der Mitglieder, eine große Zahl von Informationen. Einige seiner Anforderungen an das rechnergestützte System könnten sein:

(a) Jederzeit feststellen zu können, ob ein Buch gerade ausgeliehen ist und wenn ja, an wen;

(b) am Ende eines Monats und auch bei Bedarf eine Liste der überfälligen, d.h. ausgeliehenen und nicht vor Ablauf der vereinbarten Frist zurückgebrachten Bücher erhalten zu können;

(c) daß einem Mitglied, welches ein Buch zurückbringt, eine Rückgabe-Quittung ausgestellt wird;

(d) daß in regelmäßigen Zeitabständen eine Liste der jeweils im vergangenen Zeitraum erworbenen Bücher erstellt wird, welche die Preise je Buch, je Zeitraum und je Jahr enthält.

Wir nehmen an, daß der den realen BUCH-Prozeß im Rahmen unseres Systems simulierende Modellprozeß BUCH seine Inputs über einen Kanal erhält. Wann immer etwas mit dem realen Buch geschieht, wird sich dies für das simulierte Buch dadurch nachvollziehen, daß die "realen Daten" mit der der "realen Aktion" entsprechenden Transaktion dem rechnergestützten System in irgendeiner Form (hier sicherlich via Tastatur und Bildschirm) bekanntgegeben werden. Geschieht mit dem Buch nichts, so ist auch der korrespondierende Modellprozeß suspendiert, sein Zustand jedoch ist einsehbar. Und auf die Zustände der BUCH Modellprozesse beziehen sich offenbar die obigen Anforderungen (a) bis (d).

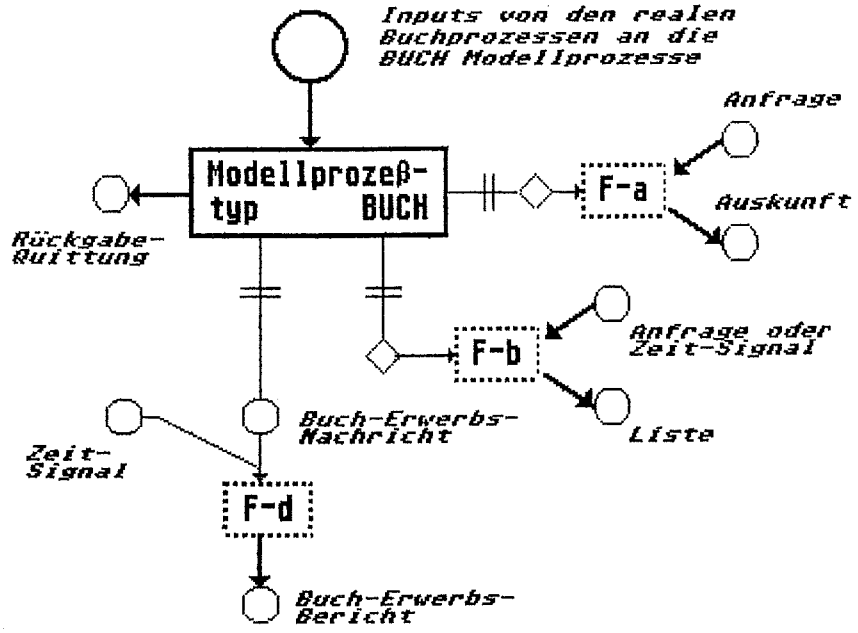
Ohne die Struktur und die Arbeitsweise der Modellprozesse im Detail zu kennen (wir wissen nur, daß sie den Realwelt-Prozessen in geeigneter Weise nachgebildet sind), können wir leicht einsehen, daß die genannten Anforderungen entweder durch *funktionale Prozesse* zu erfüllen sind, welche irgendwie mit den BUCH Modellprozessen kommunizieren, oder durch (eventuell zusätzliche) Outputs der Modellprozesse selbst. So wird es beispielsweise ein leichtes sein, die Rückgabequittung als Output der Modellprozesse vom Typ BUCH selbst zu erzeugen. Denn dieser Prozeß muß ja, wie oben bemerkt, von der realen Rückgabe Kenntnis erhalten. In Reaktion auf diesen Input kann er dann die Quittung ohne weiteres als Output produzieren.

Nicht ganz so unmittelbar lassen sich die übrigen Anforderungen erfüllen. (a) zum Beispiel könnte im Prinzip auf die gleiche Weise wie (c) erledigt werden, wenn der Verwalter auf die Möglichkeit verzichtete, die gewünschte Information nur auf ausdrückliche Anforderung zu erhalten. So muß hier ein funktionaler Prozeß konstruiert werden, der einerseits die Zustände der BUCH Modellprozesse inspiziert und andererseits den Anstoß dazu durch ein Signal erhält, welches ihm über einen Kanal zugeleitet wird. Wir nennen diesen funktionalen Prozeß F-a.

Auch die periodische (oder sonst bei Bedarf verlangte) Produktion einer Liste der überfälligen Bücher kann nach entsprechender Inspektion der Zustände der BUCH Modellprozesse erfolgen. Auch in diesem Fall ist ein zusätzlicher, über einen Kanal zugeführter Input notwendig, welcher entweder den Ablauf eines Monats oder das Vorliegen einer Anfrage signalisiert. Der die Anforderung (b) erfüllende funktionale Prozeß sei F-b.

Ein funktionaler Prozeß F-d besorgt die regelmäßige Auflistung der Neuerwerbungen. Durch die Frage, wie er mit den BUCH Modellprozessen zu verbinden sei, wird klar, daß bereits in diesem frühen Stadium der Systemspezifikation Effizienzüberlegungen durchaus eine Rolle spielen. Da der Zustand eines BUCH Modellprozesses das Datum des Beginns dieses Prozesses (und andere allgemeine Informationen) "enthält", wäre es prinzipiell möglich, F-d ebenfalls mittels Zustandsinspektion seine Arbeit verrichten zu lassen. Doch würde dies bedeuten, daß wann immer eine Liste zu erstellen ist, F-d die Zustände sämtlicher BUCH Prozesse anzuschauen hätte. Um dies zu vermeiden, legen wir Kanäle zwischen die BUCH Prozesse und F-d, über welche bei Beginn eines BUCH Prozesses (also zum Zeitpunkt des Erwerbs eines Buches) dieser dem Prozeß F-d eine "Buch-Erwerbs-Nachricht" zukommen läßt.

Das folgende Bild, ein *Prozeß-Kommunikations-Diagramm*, zeigt die bisher geschilderten Zusammenhänge:




Man beachte die weiter oben zu den einzelnen graphischen Elementen gegebenen Erläuterungen! Danach bedeutet zum Beispiel die *Zustandsspeicher*-Verbindung zwischen F-a und den Modellprozessen, daß F-a entsprechend der an ihn gerichteten Anfrage den Zustandsspeicher des "richtigen" BUCHprozesses selektiert. Für F-a kommt sinnvollerweise keine andere Verbindung zu den Modellprozessen in Betracht. Dagegen kann man sich für F-b - aus eben den Gründen, die wir für F-d geltend gemacht haben - sehr wohl eine Kanalverbindung zu den BUCHprozessen vorstellen. (Der Leser möge sich überlegen, welche Arten von Nachrichten über diesen Kanal transportiert werden sollten.)

Die bisherigen Anforderungen an das Bibliothekssystem konnten ohne jegliche Bezugnahme auf die MITGLIED- und RESERVIERUNGsprozesse erfüllt werden. Wir schließen dieses Beispiel mit der Darstellung eines *Netzwerks* von Prozessen ab, in welchem alle drei Modellprozesse integriert sind. Dieses Netzwerk ist ein System, welches zusätzlich zu F-a und F-d die folgenden funktionalen Prozesse enthält:

RP (RESERVIERUNG PRÜFEN)

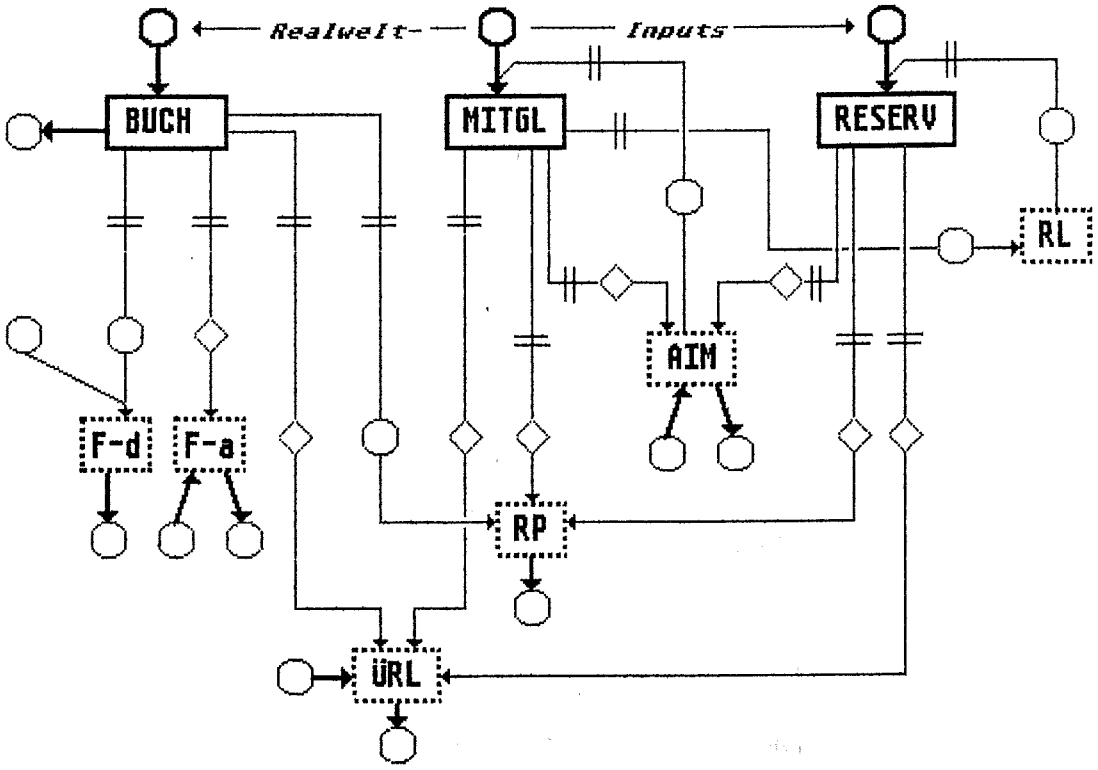
Bei Rückgabe eines Buches wird nachgeprüft, ob der Titel reserviert wurde. Falls ja, so werden die Namen und Adressen der wartenden Mitglieder ausgegeben.

AIM (AUSSCHEIDEN INAKTIVER MITGLIEDER)

Mitglieder die seit mehr als fünf Jahren nicht mehr aktiv waren, werden ausgeschieden und entsprechend benachrichtigt.

RL (RESERVIERUNG LÖSCHEN)

Wenn ein Mitglied ausscheidet, so werden eventuell noch vorhandene Reservierungen dieses Mitgliedes gelöscht.




Den oben eingeführten funktionalen Prozeß F-b haben wir hier modifiziert zu:

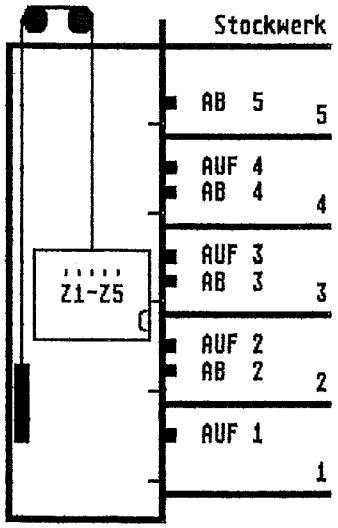
ÜRL (LISTEN DER ÜBERFÄLLIGEN UND RESERVIERTEN BÜCHER)

Am Ende eines Monats und auch bei Bedarf wird eine Liste der Bücher ausgegeben, deren Rückgabe überfällig ist und auf deren Titel Reservierungen eingetragen sind.

Beachtenswert ist die Tatsache, daß funktionale Prozesse (wie AIM und RL) ihrerseits den Modellprozessen Inputs liefern können und unter Umständen auch müssen. Der Prozeß RL zum Beispiel erzeugt für selektierte RESERVIERUNGsprozesse Nachrichten vom Typ RES-WIDERRUFEN, und beendet damit bestehende Vormerkungen.

Bei der Besprechung des Bibliothekssystems haben wir, um unsere Ausführungen nicht zu überfrachten, ganz bewußt davon abgesehen, den Übergang von den Realweltprozessen zu den Modellprozessen und die Modellprozesse selbst in allen Einzelheiten zu schildern. Mit dem zweiten "Projekt" dieses Abschnitts, welches einem ganz anderen Bereich der Anwendung von Rechnern entstammt (der Kontrolle technischer Anlagen), wollen wir die Zusammenhänge zwischen Realwelt und Modell im Rahmen des JSD etwas genauer studieren und formalisieren. Es handelt sich um die Spezifikation von Software zur Steuerung der

Bewegungen eines Fahrstuhls (vgl. [JA2]), dessen wesentliche Elemente aus der folgenden Skizze ersichtlich sind (die Anzahl der Stockwerke ist dabei natürlich willkürlich gewählt):




Neben der Fahrstuhltür eines Stockwerks (ausgenommen das unterste und das oberste) gibt es Knöpfe **AUF m** und **AB m** (**m** sei die Nummer des Stockwerks), mit denen signalisiert werden kann, ob man von der gegebenen Etage aus nach oben oder nach unten fahren möchte. Im untersten Stockwerk (**m = 1**) gibt es nur einen Knopf **AUF 1** und im obersten (hier **m = 5**) nur einen Knopf **AB m**. In der Kabine selbst befinden sich die (hier) fünf Zielknöpfe **Z1 - Z5**, mit denen die gewünschte Zieletage ausgewählt wird. In den Knöpfen befinden sich kleine Lampen, die nach Knopfdruck aufleuchten. Das Erreichen eines Stockwerks wird mit Hilfe geeigneter elektromechanischer Vorrichtungen an der Fahrstuhlkabine und an der Schachtwand festgestellt. Falls für ein Stockwerk ein Haltewunsch vorliegt (sei es von innerhalb oder außerhalb der Kabine), so wird der den Fahrstuhl bewegende Motor angehalten, Lampen werden ausgeschaltet und die Tür wird geöffnet. Doch damit sind wir schon weiter, als wir uns auf die Technik des Aufzugs einlassen wollen. Wir beschränken uns im wesentlichen auf die Steuerung des Antriebsaggregats und - weniger ausführlich - auf das Ein- und Ausschalten der Knopflampen. Entsprechend der für das JSD charakteristischen Vorgehensweise müssen wir uns zunächst nach den für das technische System "Fahrstuhl" relevanten Realwelt-Objekten umschauen, die mit diesen Objekten verknüpften Prozesse analysieren und diesen Prozessen dann innerhalb des "Rahmens der Rechnerunterstützung" (s.o.) geeignete Modellprozesse gegenüberstellen.

Wie finden wir diese Objekte? Die Antwort darauf ist nicht schwer: Es ist unser Hauptinteresse, den Motor zu steuern, ihm also "Kommandos" geben zu können, welche ihn anhalten, starten und eventuell seine Laufrichtung (Polarität) ändern lassen. Die Frage nach den Objekten der realen Welt, auf deren Modellierung die Rechnerunterstützung aufzubauen ist, muß also genauer lauten: Welche Informationen benötigen wir, um einen solchen Output zu erzeugen? Und dann: Wo werden diese Informationen ihrerseits erzeugt?

Um im Falle der Fahrstuhlanlage Kommandos der gewünschten Art produzieren zu können, müssen wir wissen, wo sich die Kabine jeweils befindet und welche Haltewünsche jeweils bestehen. Das "Wo" der Kabine erfahren wir gewissermaßen von ihr selbst - über den Mechanismus, der sie das Erreichen und Verlassen eines Stockwerks feststellen läßt. Und die Haltewünsche entstehen durch

die Betätigung von Ruf- und Zielknöpfen. Die beiden für unser Projekt relevanten Objekte (und daran anknüpfend Realwelt-Prozesse) sind also

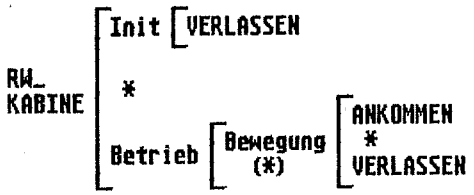
KABINE und KNOPF.

Dabei bedeutet KABINE tatsächlich ein einzelnes Objekt, während KNOPF (wie die Objektnamen im Beispiel des Bibliothekssystems) einen Objekttyp mit mehreren identifizierbaren Ausprägungen bezeichnet. Welche dies sind, ist aus der obigen Skizze der Anlage ersichtlich. Wir wollen die Identifikation der einzelnen Ausprägung hier als Parameter notieren:

$$\text{KNOPF} = \{\, \text{KNOPF(K-Id)} \mid \text{K-Id} \in \{\text{AUF-1, AUF-2, . . . , AUF-(n-1)}\} \cup \{\text{AB-2, AB-3, . . . , AB-n}\} \cup \{\text{Z-1, Z-2, . . . , Z-n}\} \,\} \quad (n = 5)$$

(Hätten wir die Aufgabe, nicht nur eine, sondern mindestens zwei Fahrstuhlkabinen zu steuern, so wäre im übrigen auch KABINE als Objekttypname zu interpretieren.)

Wir wissen schon: Die einzige Information, die uns die Kabine über ihren Ort liefert, ergibt sich aus den ihr zugeordneten Ereignissen ANKOMMEN und VERLASSEN, jeweils bezogen auf eine Etage. Das "Leben der Kabine", ihr Prozeß, ist eine Abfolge dieser beiden Ereignisse. Es beginnt damit, daß sie irgendwann einmal ein Stockwerk verläßt, um dann das Ereignispaar (ANKOMMEN, VERLASSEN) beliebig oft zu wiederholen. Das nachstehende Klammerdiagramm beschreibt dieses "Leben". Zur besseren Unterscheidung vom zugehörigen Modellprozeß, nennen wir den Prozeß der realen Kabine RW-KABINE.




Der Modellprozeß innerhalb unseres Softwaresystems muß, zur Simulation des Realwelt-Prozesses RW-KABINE und um die für die Produktion des gewünschten Outputs notwendigen Informationen über den Ort der Kabine bereitstellen zu können, von den Ereignissen im "Leben der Kabine" erfahren. Dafür gibt es, wie wir bereits früher gesehen haben, prinzipiell zwei Möglichkeiten, die Kanalverbindung zwischen RW-KABINE und S-KABINE (so nennen wir den die realen Kabinenbewegungen nachvollziehenden Modellprozeß) oder die Verbindung mittels Zustandsspeicher:


**KOK: Kabinen-Ortssignal-Kanal**


**KOA: Kabinen-Orts-Anzeige**

Welche der beiden Varianten wir wählen, hängt weitgehend von der Technik der fraglichen Anlage ab. Es ist denkbar, daß wir von den Sensoren jedesmal dann ein Signal erhalten, wenn die Kabine eine Etage erreicht oder verläßt, und daß dieses Signal über KOK dem Rechner zugeführt wird. Für den simulierenden Prozeß S-KABINE ist der Empfang dieses Signals gleichbedeutend mit den Ereignissen ANKOMMEN und VERLASSEN, und er kann diese Ereignisse - in welcher Form auch immer - an andere Prozesse im System weitermelden. Die Version 1 von S-KABINE lautet also:

```
S_KABINE
/*V. 1*/
   Init
      IN.KOK:RW_FS_Signal
      *
      /*Aktionen nach VERLASSEN einer Etage...*/
   *
   Betrieb
      Bewegung
         (*)
            IN.KOK:RW_FS_Signal
            *
            /*Aktionen nach ANKOMMEN ...*/
            *
            IN.KOK:RW_FS_Signal
            *
            /*Aktionen nach VERLASSEN ...*/
```

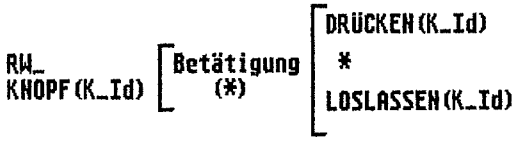
Die Operation "Kanalzugriff" des simulierenden Prozesses notieren wir hier - wie schon in Kapitel 5 - mit dem Schlüsselwort "IN", qualifiziert durch den Namen des Kanals. Die Operation "IN" blockiert (bzw. suspendiert, s.o.!) den Prozeß, falls kein Signal vorliegt.

Version 2 geht davon aus, daß die technische Verbindung zwischen Rechner und Fahrstuhlanlage ein Register enthält, in dem zu jeder Zeit der momentane Zustand der Kabine ablesbar ist. Wir nehmen an, der Zustand "Kabine ist auf oder passiert Etage" werde durch den Registerinhalt 1 dargestellt, während der Registerinhalt 0 andeutet, daß sich die Kabine zur Zeit nicht auf einer Etage befindet. Das Register nennen wir KOA. Eine Notation für die Operation des Lesens von Speicherinhalten haben wir ebenfalls im Rahmen der Beispiele des Kapitels 5 eingeführt.

```
S_KABINE
/*V. 2*/
   Init
      KOA:Kabinenort(=KO)
      *
      AufEtage
         ___
         (*WHILE KO=1)
            KOA:...
      *
      /*Aktionen nach VERLASSEN einer Etage ...*/
   *
   Betrieb
      Bewegung
         (*)
            Nirgends
               ___
               (*WHILE KO=0)
                  KOA:...
            *
            /*Aktionen nach ANKOMMEN ...*/
            *
            AufEtage
               ___
               (*WHILE KO=1)
                  KOA:...
            *
            /*Aktionen nach VERLASSEN ...*/
```

Anders als Version 1 von S-KABINE muß dieser Prozeß ständig aktiv sein, um sich "auf dem laufenden zu halten". Und er muß dazu hinreichend schnell arbeiten (was aber kein Problem sein dürfte)! Für die Spezifikation der funktionalen Prozesse ist es - wie uns noch klar werden wird - unerheblich, mit welcher Version wir es hier zu tun haben. Wir nehmen an, daß wir uns aufgrund der gegebenen Technik für Version 2 entscheiden müssen.

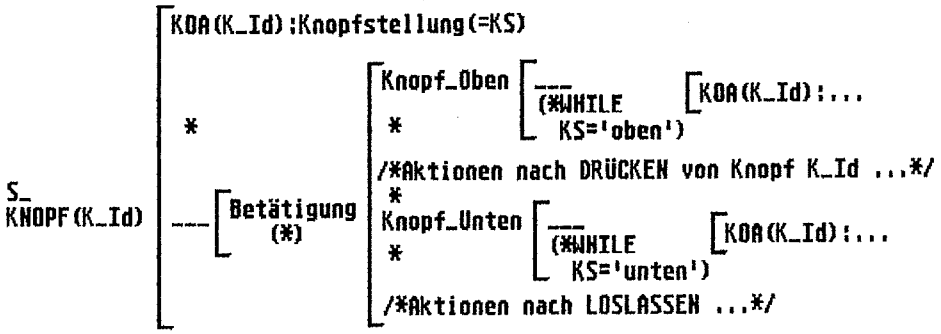
Wenden wir uns nun der zweiten Informationsquelle unserer Aufzugsanlage zu, den Knöpfen. Das "Leben" dieser Realwelt-Objekte ist ebenso ereignisarm wie das der Kabine. Knöpfe werden gedrückt, das ist alles. Nur: bei genauerem Hinsehen müssen wir zwei Ereignisse unterscheiden, DRÜCKEN und LOSLASSEN, denn ohne das letztere wäre das erste gar nicht erkennbar. Der Prozeßtyp RW-KNOPF hat also die folgende einfache Struktur (hier mit Angabe des formalen Parameters K-Id zur Identifizierung einzelner Ausprägungen):




Auch hier seien die technischen Gegebenheiten derart, daß nur eine Verbindung per Zustandsspeicher zwischen den Realwelt-Prozessen RW-KNOPF und den entsprechenden simulierenden Prozessen S-KNOPF in Frage kommt:

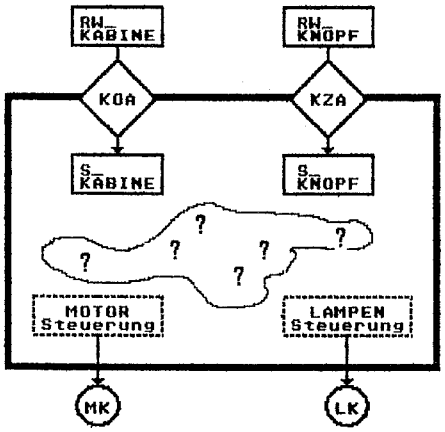

**KZA: Knopf-Zustands-Anzeige**

Mit der Verwendung dieses Darstellungselementes haben wir - wie der Leser nach nochmaliger Durchsicht unserer Liste graphischer Notationen leicht feststellt - ein wenig gemogelt. Der Liste zufolge gilt das hier für die Verbindung zweier Prozeßtypen benutzte Zeichen lediglich für einzelne Prozesse. Wir ergänzen daher die in der Liste angegebene Semantik, indem wir bestimmen, daß die (oben noch illegale) Notation "P1 ◇ P2" auch dann anwendbar sein soll, wenn P1 und P2 Prozeßtypen mit eineindeutig korrespondierenden Ausprägungen bezeichnen.



Die Modellprozesse, welche das "Leben" der Realwelt-Knöpfe nachvollziehen, nehmen den im obigen Diagramm beschriebenen Verlauf.

Man beachte hier (wie in Version 2 der S-KABINE) die Operation des "Vorweglesens" (vgl. Kapitel 5), um eine erste Bestimmung der Knopfstellung (bzw. des Kabinenortes) vorzunehmen. (Danach hätten wir eigentlich etwas genauer, unter Verzicht auf die Annahme, daß ein Knopfleben stets oben beginnt, eine Unterscheidung der weiteren Aktionen von S-KNOPF entsprechend dem Resultat des Vorweglesens treffen müssen. Mutatis mutandis gilt dies für Version 2 von S-KABINE. Der Leser möge es zur Übung selbst tun.)




Als wesentliche Erkenntnis nach den bisherigen Ausführungen zum Fahrstuhlprojekt bemerken wir, daß die den Realwelt-Prozessen nachempfundenen Modellprozesse eines rechnergestützten Systems im Prinzip nichts anderes tun, als ihr jeweiliges Gegenüber zu *überwachen* und sich in dessen Zustand (soweit er für das Gesamtsystem von Interesse ist) zu versetzen (vgl. den oberen Teil der nebenstehenden Abbildung).

Auch bei der weiteren Entwicklung unseres Softwaresystems lassen wir uns vom gewünschten Output leiten: Der Motor soll gesteuert werden, und wir wollen die Lampen in den Knöpfen ein- und ausschalten. Dazu bedarf es verschiedener Kommandos, für deren Anwendung wir das Vorhandensein geeigneter Vorrichtungen voraussetzen. Im unteren Teil der Abbildung werden sie als Kanal MK (für die Motorkommandos) und Kanaltyp LK (für die Lampenkommandos) dargestellt. Wir postulieren einen funktionalen Prozeß MOTOR-Steuerung sowie einen funktionalen Prozeßtyp LAMPEN-Steuerung respektive) mit den entsprechenden Ausprägungen, die ihre wesentlichen Outputs an den jeweiligen Kanal abgeben. Die Motorkommandos z.B. sind:

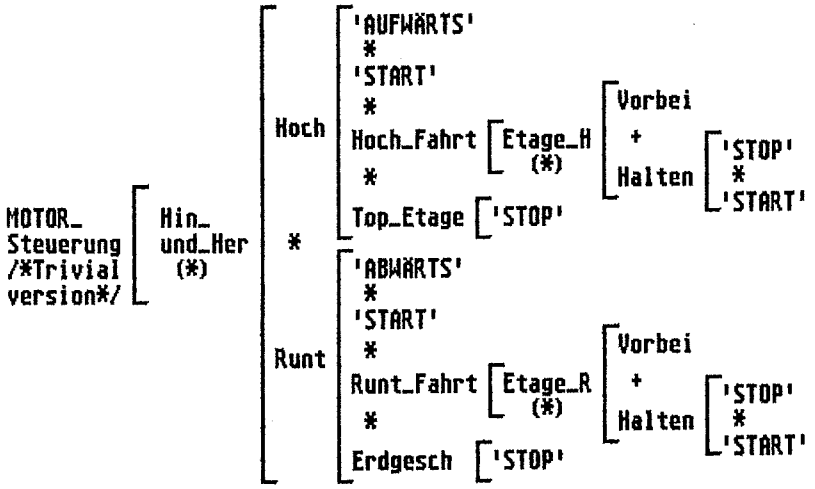
**AUFWÄRTS** - Polung zur Aufwärtsfahrt
**ABWÄRTS** - Polung zur Abwärtsfahrt
**START** - Motor starten
**STOP** - Motor stoppen

Wie - so lauten die nächsten Fragen - finden wir die funktionalen Prozesse und wie hängen sie mit den Modellprozessen und untereinander zusammen?

Zum "Wie" des Auffindens der funktionalen Prozesse gibt es zuerst eine schlechte Nachricht: Es ist dies eine Aufgabe, für die es leider keine rezeptartige Methode gibt, sondern die das ganze Geschick und die Erfindungsgabe des Ingenieurs erfordert. Die Motorsteuerung muß einer Vorstellung davon entsprechen, wie sich die Fahrstuhlkabine bewegen soll. Doch nun die guten Nachrichten: Erstens kann man solche Vorstellungen mit den uns zur Verfügung stehenden

Mitteln zur Darstellung sequentieller Abläufe recht gut formal ausdrücken, ohne dabei gleich an ein Rechnerprogramm zu denken, und zweitens ist es - wenn diese Spezifikation erst einmal geschrieben ist - nicht mehr schwer, auf solcher Basis zu einer Software-orientierten Beschreibung zu gelangen.

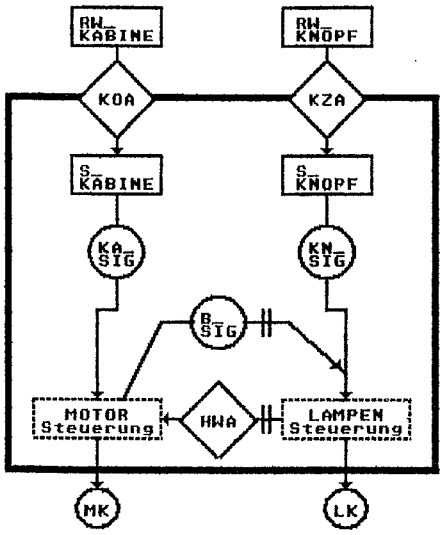
Für das Folgende machen wir uns eine sehr einfache Vorstellung von den Kabinenbewegungen: Die MOTOR-Steuerung soll dafür sorgen, daß die Kabine immer hin und her fährt, von ganz unten nach ganz oben und zurück und so weiter, und daß sie unten hält und oben sowie zwischendurch, wenn dies verlangt wird. Dieses simple Verhalten der Kabine wird offenbar durch die im nachstehenden Klammerdiagramm spezifizierte Folge von Motorkommandos bewirkt:




In dieser Version (wie in vermutlich jeder anderen auch) ist bei der Annäherung der Kabine an eine Etage zu entscheiden, ob die Kabine dort vorbeifahren oder anhalten soll. Die für diese Entscheidung notwendige Information muß von einem Prozeß kommen, der den jeweiligen Zustand der Knöpfe kennt, oder vielmehr, da sich die Knopfmanipulationen DRÜCKEN und LOSLASSEN auf die Lampen auswirken, den jeweiligen Zustand der Lampen. Wir postulieren daher eine Zustandsspeicher-Verbindung zwischen der MOTOR-Steuerung und der LAMPEN-Steuerung, wobei wir letztere in ebenso vielen Exemplaren vorfinden, als Lampen zu schalten sind. Die MOTOR-Steuerung kann also, wenn bei Erreichen eines Stockwerks die Entscheidung "Halten oder Vorbeifahren" zu treffen ist, gezielt die Zustände der für dieses Stockwerk in Frage kommenden Lampen ablesen. Sie interpretiert diese Zustände als Vorliegen oder Nicht-Vorliegen eines Haltewunsches. Wir nennen daher den entsprechenden Zustandsspeicher HWA ("HalteWunschAnzeige").

Andererseits muß unsere LAMPEN-Steuerung (die ja nichts anderes zu tun hat, als die Knopflampen ein- und auszuschalten) es erfahren, wenn die Kabine eine Etage erreicht, um dann eventuell leuchtende Lampen auszuschalten. Dazu legen wir - als naheliegende konstruktive Maßnahme - zwischen die MOTOR-Steue-

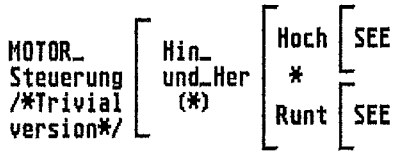
rung und die einzelnen Ausprägungen der LAMPEN-Steuerung je einen Kanal vom Typ B-SIG ("BesuchsSignal"). (Man beachte: Die "Herren" solcher Maßnahmen sind allein wir! Wir haben jede Freiheit, unser Softwaresystem so zu spezifizieren, wie wir es für richtig halten, solange wir dabei die von außen an uns gestellten Anforderungen zuverlässig erfüllen, in diesem Falle also den gewünschten Output an den Motor zu erzeugen.)




Um die beiden funktionalen (Steuerungs-) Prozesse auch mit den notwendigen Nachrichten über die Kabinenbewegungen und das Knöpfedrücken zu versorgen, verbinden wir den Prozeß S-KABINE mit der MOTOR-Steuerung und die S-KNOPF Prozesse mit den LAMPEN-Steuerungen, und wir wählen auch dazu jeweils Kanäle. Auch dies liegt nahe, denn die MOTOR-Steuerung zum Beispiel sollte ja nur dann aktiv werden, wenn Bedarf dafür gegeben ist, also wenn von S-KABINE die Ankunft der realen Kabine in einem Stockwerk registriert wird. In diesem Fall wird S-KABINE über den Kanal KA-SIG eine Botschaft an die MOTOR-Steuerung senden, die diese "aufweckt" und zu Aktionen veranlaßt. Die Struktur dieser Botschaft ist so trivial wie ein Weckruf, "Hallo" oder "Da" oder ähnlich, mehr Information wird nicht benötigt. Es ist einfach "etwas" im Kanal oder nicht.

Entsprechendes gilt für die Beziehung zwischen den S-KNOPF Prozessen und den LAMPEN-Steuerungen.

Aus der oben angegebenen Spezifikation des Verhaltens der Kabine läßt sich nun, wie angekündigt, sehr einfach eine Beschreibung des Prozesses "MOTOR-Steuerung" herleiten. Wir haben weiter nichts zu tun, als die Kommandos an den Motor (den "primären" Output) in Output-Operationen einzubetten, "sekundären" Output an die LAMPEN-Steuerungen einzufügen, sowie durch geeignete Inputs den Fortgang des Prozesses zu sichern und ihn jeweils die gewünschten Zweige (z.B. "Halten oder Vorbeifahren") einschlagen zu lassen:



```
Hoch
  OUT.MK:'AUFWÄRTS'
   *
  ---  (EtNr=1)
       OUT.B_SIG.AUF(EtNr):'DA'
       *OUT.B_SIG.Z(EtNr):'DA'
   *
  OUT.MK:'START'
   *
  Hoch_Fahrt  Etage_H          SEE
              (*WHILE EtNr<N)
   *
  Top_Etage   IN.KA_SIG:'DA'
              *OUT.MK:'STOP'
              *OUT.B_Sig.Z(EtNr):'DA'
```

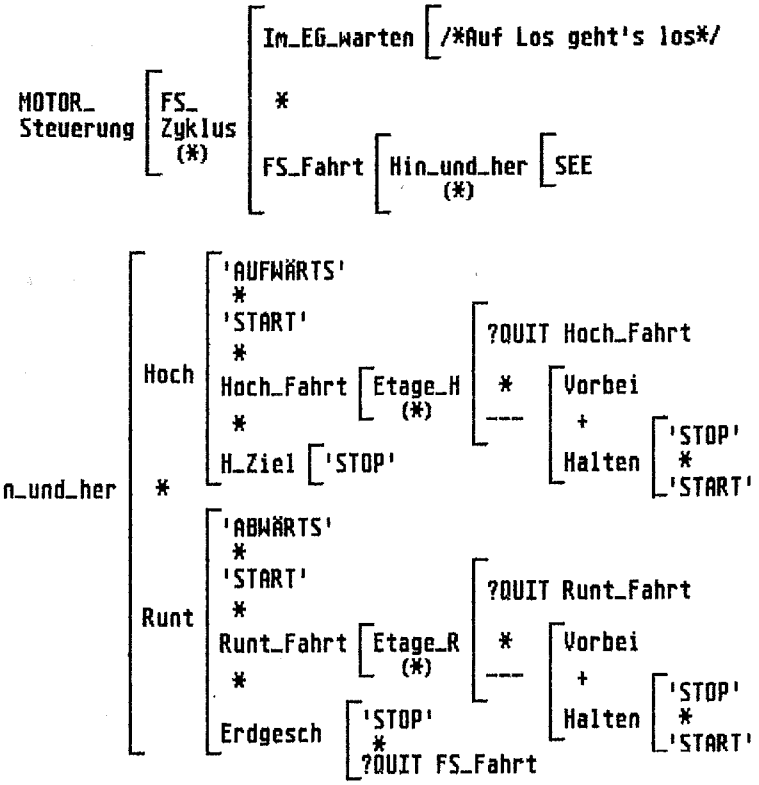
```
Runter
  OUT.MK:'ABWÄRTS'
   *
  OUT.B_SIG.AB(EtNr):'DA'
   *
  OUT.MK:'START'
   *
  Runt_Fahrt  Etage_R          SEE
              (*WHILE EtNr>1)
   *
  Erd_Gesch   IN.KA_SIG:'DA'
              *OUT.MK:'STOP'
```

```
Etage_H
  (EtNr=EtNr + 1)
  IN.KA_SIG:'DA'
   *
  HWA.AUF(EtNr):Haltewunsch(=HW_AUF)
   *
  HWA.Z(EtNr):Haltewunsch(=HW_Z)
   *   ?HW_AUF='JA'          OUT.MK:'STOP'
       OR HW_Z='JA';         *OUT.B_SIG.AUF(EtNr):'DA'
       Halten                *OUT.B_SIG.Z(EtNr):'DA'
  ---  +                     *OUT.MK:'START'

       ?ELSE;
       Vorbei  NONE
```

Die Beschreibung von ETAGE-R ist analog und sei dem Leser überlassen.

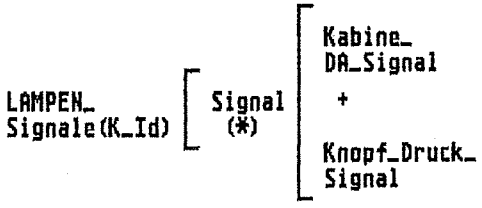
En passant bemerken wir, daß an diesem Beispiel der Unterschied zwischen *Anforderungsspezifikation* (engl. *requirement specification*) und *Systemspezifikation* in besonders prägnanter Weise zum Vorschein kommt: Als Bestandteil der *Anforderungsspezifikation* muß die Beschreibung des gewünschten Kabinenverhaltens gelten, ausgedrückt durch die Struktur des an den Motor gerichteten "Kommando-Stroms". Die obige, aus dieser Struktur hergeleitete Beschreibung des Prozesses "MOTOR-Steuerung" dagegen ist ein Bestandteil der *Systemspezifikation*, ebenso wie die Dokumentation der Kommunikationsbeziehungen zwischen den Prozessen innerhalb des "Rahmens" des Systems.

Der Übung halber sei dem Leser vorgeschlagen, die folgende Spezifikation des Kabinenverhaltens (eine *Anforderung* also!) zu verstehen:




Er möge versuchen, das in dieser Struktur des Kommando-Stroms zum Ausdruck gebrachte Verhalten verbal zu beschreiben und dann auch aus dieser Spezifikation eine (etwas realitätsgerechtere) Version eines Prozesses "MOTOR-Steuerung" zu erzeugen.

Ferner bleibt die Aufgabe, die LAMPEN-Steuerung auszuarbeiten. Hier ist zu beachten, daß für jede der Lampen die Inputs aus zwei Kanälen zu mischen sind: Aus dem Kanal KN-SIG der Knopfsignale und aus B-SIG, dem Kanal, über den die MOTOR-Steuerung den Besuch einer Etage signalisiert. Der Strom der von einer (einzelnen) LAMPEN-Steuerung (identifiziert durch K-Id, s.o.) zu verarbeitenden Signale hat also die folgende Gestalt:




Es ist wichtig, daß die LAMPEN-Steuerung die Herkunft der bei ihr eintreffenden Signale unterscheiden kann. Denn je nachdem, ob es durch einen Knopfdruck oder durch die Bewegung der Kabine hervorgerufen wurde, wird die Lampe ein- oder auszuschalten sein.

Auch die ausführliche Ausarbeitung der LAMPEN-Steuerung bleibe dem Leser überlassen.

Apropos "Realitätsgerechtigkeit": Der eher überblickshafte Charakter der Darstellungen dieses Kapitels ließ es leider nicht zu, daß wir bei der Behandlung speziell des Beispiels "Fahrstuhl" einen ernsthaften Gedanken den Problemen widmeten, die sich ergeben, wenn einmal etwas nicht "so läuft wie es sollte": Wenn die Tür klemmt oder der Motor den Dienst versagt oder dergleichen *Ausnahmesituationen* mehr eintreten. Es versteht sich von selbst, daß diesen Problemen bereits bei der Spezifikation des Systems besondere Aufmerksamkeit zu widmen ist. Schließlich wird es zu den "von Außen"an die Systementwickler gestellten Anforderungen gehören, auch in derartigen Ausnahmesituationen "gutmütige" Reaktionen des Systems zu garantieren. Immerhin: Ohne es anhand unserer Beispiele schlagkräftig demonstriert zu haben, behaupten wir dennoch, daß das JSD durch seine gründliche Analysetechnik und die Möglichkeit, bestimmte Anforderungen formal, aber dennoch relativ anschaulich zu spezifizieren, es schon in den Anfangsphasen eines Projekts gestattet, auch der Behandlung dieser Aspekte gebührenden Raum zu geben.

Es ist schließlich festzuhalten, daß das JSD - soweit wir es hier vorgestellt haben - sehr abstrakte Systemspezifikationen (in Begriffen wie Prozeß, Prozeßtyp, Prozeßkommunikation, etc.) liefert. Wie zu Beginn dieses Abschnitts bereits kurz erwähnt, enthält es allerdings auch Anleitungen zur Implementierung von Systemspezifikationen, also Vorschläge zur Abbildung der Spezifikation auf "real ablaufende" Software. Wir werden einige dieser Vorschläge in den späteren Abschnitten über "Entwurf" und "Implementierung" (wenn auch nicht notwendigerweise expressis verbis) wiederfinden.

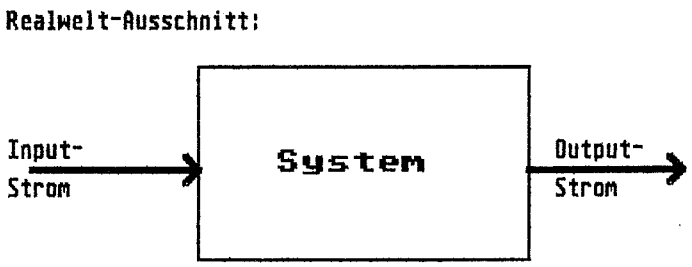
### 6.3.4 Hierarchische Systemmodelle

Wir erinnern an dieser Stelle zunächst an Abschnitt 6.1 dieses Kapitels, in dem wir behaupteten, daß sich die Struktur eines Softwaresystems "in erster Näherung" als Baum darstellen lasse, mit dem Gesamtsystem als Wurzelknoten und den vernünftigerweise nicht mehr weiter zerlegbaren Systemteilen, denen wir - ganz informell - die Bezeichnung "Module" gaben, als Blätter. Wir haben dieses Bild mit einer möglichen und plausiblen Vorgehensweise des Software-Entwicklers beim Entwurf seines Produkts begründet: "Top-Down", von oben nach unten, von der Gesamtheit zu den Einzelheiten. Und wir haben an die Verwandtschaft mit der "Schrittweisen Verfeinerung" als Verfahren zur Konstruktion "kleiner" Programme oder Programmteile erinnert.

Genau diese Vorstellungen liegen den im vorliegenden Abschnitt zu besprechenden *hierarchischen Systemmodellen* zugrunde. Wohlgemerkt: wir sind uns bewußt, daß wir hier von Modellen reden, deren Hauptzweck es ist, die wesentlichen Eigenschaften der Umgebung eines zukünftigen Rechnereinsatzes

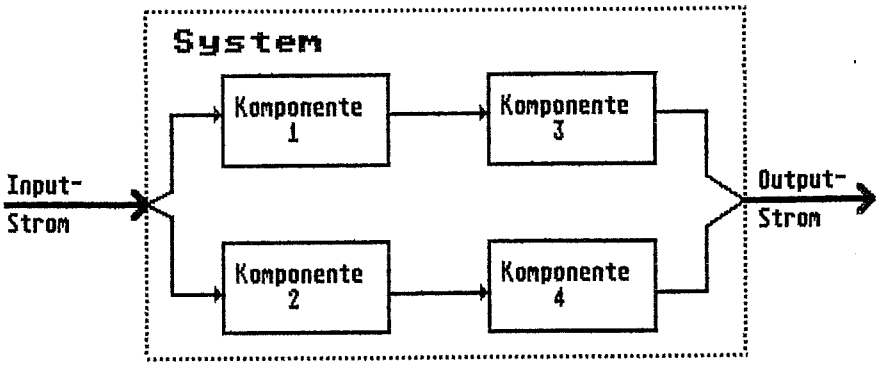
einzufangen und mit dem Ziel der Präzisierung von Anforderungen an ein Softwaresystem zu beschreiben. Die Spezifikation des Softwaresystems soll sich dann - hoffentlich nahtlos (wie etwa durchaus im Falle des JSD) - auf das im Modell formulierte Analyseergebnis aufsetzen lassen.

Der einem hierarchischen Modell folgende Analytiker eines Realweltsystems (der entweder mit dem Entwickler des dann für diese Realwelt zu bauenden Softwaresystems identisch ist oder aber mit diesem eng zusammenarbeitet) wird sich zunächst eine Abgrenzung des fraglichen Realweltausschnitts markieren, den "Modellrand" (vgl. 6.3.1), indem er die Daten- bzw. Objektströme identifiziert, welche in diesen bzw. aus diesem Ausschnitt fließen. Er wird seinem System einen Namen geben und eine Beschreibung der wesentlichen Leistungen ("Funktionen") beifügen. Die graphische Dokumentation dieses ersten Ergebnisses seiner Bemühungen unterscheidet sich nicht von dem Bild, das wir in Abschnitt 3.1. gezeichnet haben:




Man mag ferner bemerken, daß dieses Bild auch nicht so sehr verschieden ist von den "Rahmen", die wir im vorigen Abschnitt um unsere Systeme gelegt haben. Die Inputströme kamen dort von oben, während die Outputströme nach unten abgingen; eine fürwahr recht unbedeutende Variation. Doch halt! Das JSD gab uns eine ziemlich strikte Anleitung zum Auffinden dieser Ströme, und diese führte sehr schnell zu einer differenzierten Betrachtung der "Schnittstelle" des Systems (also dessen, was sich an seinem Rand abspielt), ohne daß wir uns schon Gedanken über das Systeminnere gemacht hätten.

Eine solche Anleitung werden wir bei der Beschäftigung mit hierarchischen Systemmodellen vermissen. Vielmehr sind wir bei der weiteren Differenzierung in viel stärkerem Maße auf unsere Intuition und unser "Fingerspitzengefühl" angewiesen. Die Zergliederung der Input- und Outputströme wird sich parallel mit der "funktionalen" Zerlegung des Systems vollziehen müssen. Eine erste Verfeinerung des "Systems als Ganzes" ergibt sich etwa durch die Identifikation von Gruppen "irgendwie" zusammengehöriger Funktionen. Diesen Funktionsgruppen werden dann jeweils korrespondierende Systemkomponenten zugedacht. Auf dieser Stufe wird sich ein Inputstrom aufspalten in Teilströme, die einzelnen dieser Komponenten zufließen. Entsprechendes gilt für den Output. Das folgende Bild zeigt das Ergebnis einer solchen Zerlegung:




Jede Komponente und die ihr zugeordneten Ströme werden analog behandelt, bis eine Stufe erreicht ist, auf der eine weitere Zerlegung nicht sinnvoll erscheint. Auf dieser letzten Stufe repräsentieren die Kästchen elementare Systemfunktionen mit den sie betreffenden Inputs und Outputs, in etwa entsprechend (jedoch nicht notwendigerweise identisch mit) den Prozessen, die wir beim JSD finden. Natürlich zwingt uns nichts, "je Kasten" (System, Komponente, Elementarfunktion) genau einen Inputstrom und genau einen Outputstrom zu sehen. Es dürfen, sollte dies opportun scheinen, durchaus mehrere sein, von denen sich dann - bei der weiteren Systemverfeinerung - jeder in zwei oder mehr Teilströme aufspalten kann.

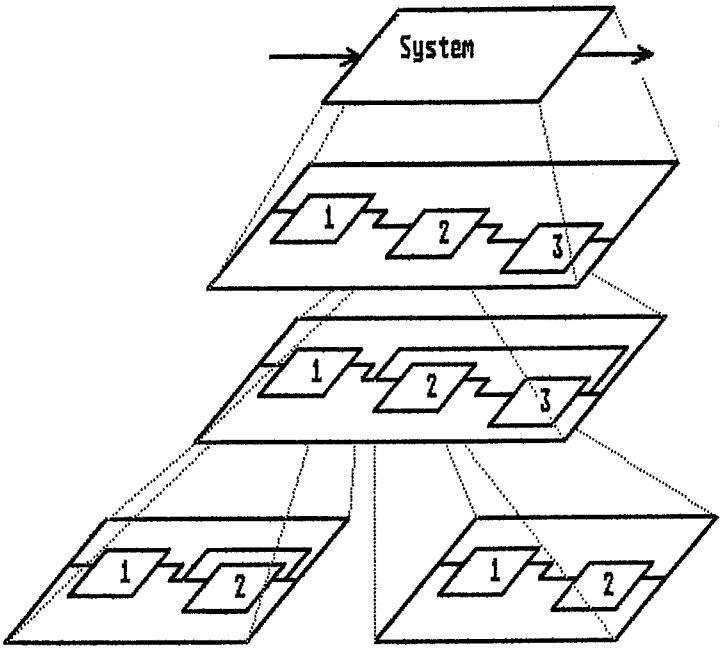



**Diagramm D-0 (Einleitung, Informationen über das System als Ganzes)**

**Diagramm D0 (noch Einleitung, Zerlegung in Komponenten K1, K2, K3)**

**Diagramm D2 (2. Kapitel, Zerlegung von K2 in K21, K22 und K23)**

**Diagramme D21 und D22 (Abschnitte 2.1 und 2.2, Zerlegungen in K211, K212 und K221, K222)**

Die Gliederung der Dokumentation eines hierarchischen Systemmodells ist naheliegend. Die Folge ihrer Kapitel und Abschnitte sollte sich am ehesten aus einer "depth first" Enumeration des "Systembaumes" ergeben: So würde sich

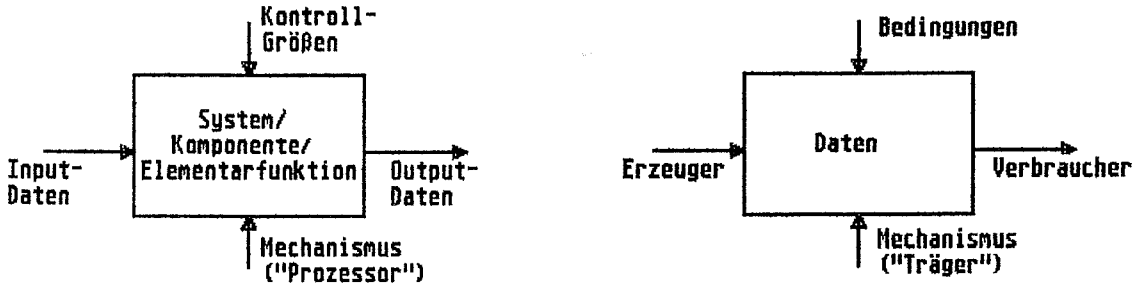
die Einleitung mit dem "System als Ganzem" beschäftigen, und jeder einzelnen Komponente der ersten Zerlegungsstufe wäre ein Kapitel zugeordnet, dessen Abschnitte den Komponenten der zweiten Zerlegungsstufe entsprechen, und so weiter, und so fort. Beispielsweise geht aus dem obenstehenden Bild die Struktur einer System-Dokumentation, ihres zweiten Kapitels und zweier Abschnitte desselben hervor.

Tatsächlich bildet die Gesamtheit solcher Diagramme, versehen mit allerhand, dem Projekt- und Produktmanagement dienenden Zusatzinformationen, den Kern der System-Dokumentation, welche durch die unter dem Akronym "SADT" (= Structured Analysis and Design Technique, vgl. [ROS]) bekanntgewordene Methode geliefert wird. Wir verwenden hier den Begriff "Methode", obwohl - wie oben gesagt - die (methodischen) Anleitungen zur Anfertigung hierarchischer Systemmodelle im allgemeinen nur recht schwach ausgeprägt sind. Die Verwendung dieses Begriffes läßt sich jedoch sicherlich umso eher rechtfertigen, je präziser die mit den Darstellungselementen und Konzepten eines gegebenen Modellierungsvorschlags verbundene Semantik ist.

Für SADT, das in den siebziger Jahren "erfunden" wurde und sich auch heute noch relativ großer Beliebtheit im Bereich der Entwicklung von Software für "kommerzielle" Anwendungen erfreut, trifft (ebenso wie für den zweiten, in diesem Abschnitt zu besprechenden hierarchischen Modellierungsansatz) diese Voraussetzung durchaus in gewissem Maße zu.

SADT wird nicht nur für die hierarchische Systemmodellierung (System - Komponente - Elementarfunktion) eingesetzt, sondern auch zur Datenmodellierung. Für beides ist ein Rechteck mit hinein- bzw. herausragenden Pfeilen die diagrammatische Grundform. Im Unterschied zur oben skizzierten "Grundversion" der hierarchischen Modellierung werden allerdings alle vier Seiten des Rechtecks als "Bedeutungsträger" genutzt.

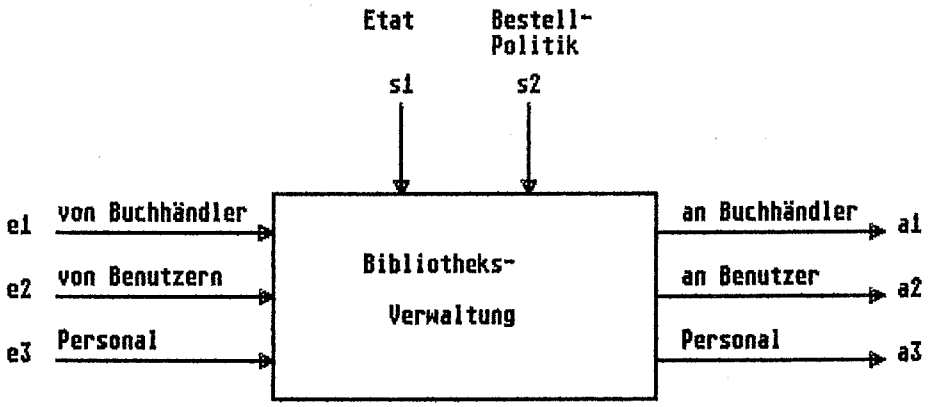
Für die beiden Spielarten, *SADT-Prozeßmodell* und *SADT-Datenmodell*, schaut dies wie folgt aus:




Die syntaktische Äquivalenz der Darstellung und die komplementären Semantiken der beiden Modelle gestatten es, hier von einer "Prozeß-Daten-*Dualität*" zu sprechen. Erzeuger und Verbraucher im rechten Diagramm sind Systeme, Komponenten oder Elementarfunktionen, während die Input- und Outputdaten

am linken Kasten eben Daten sind. Der am oberen Rand eines "Prozeß-Kastens" eintreffende Pfeil bezeichnet Einflüsse, welche die Verarbeitungen innerhalb des Kastens in irgendeiner Weise steuern. Und an dem Pfeil, der an den oberen Rand eines "Daten-Kastens" führt, werden Bedingungen notiert, unter denen die Daten vom Erzeuger geholt bzw. an den Verbraucher weitergegeben werden. Mit der Beschriftung der (unterstützenden!) Pfeile von unten schließlich werden die jeweiligen Mechanismen benannt, welche das prozeßhafte Geschehen einerseits und die Datenspeicherung andererseits besorgen. Jeder dieser vier Pfeile kann auch fehlen. Andererseits ist es zum Beispiel möglich, daß der Output eines Prozeß-Kastens die Funktion oder Funktionen eines anderen Kastens kontrolliert oder gar den Mechanismus für das Funktionieren eines anderen Kastens bereitstellt.

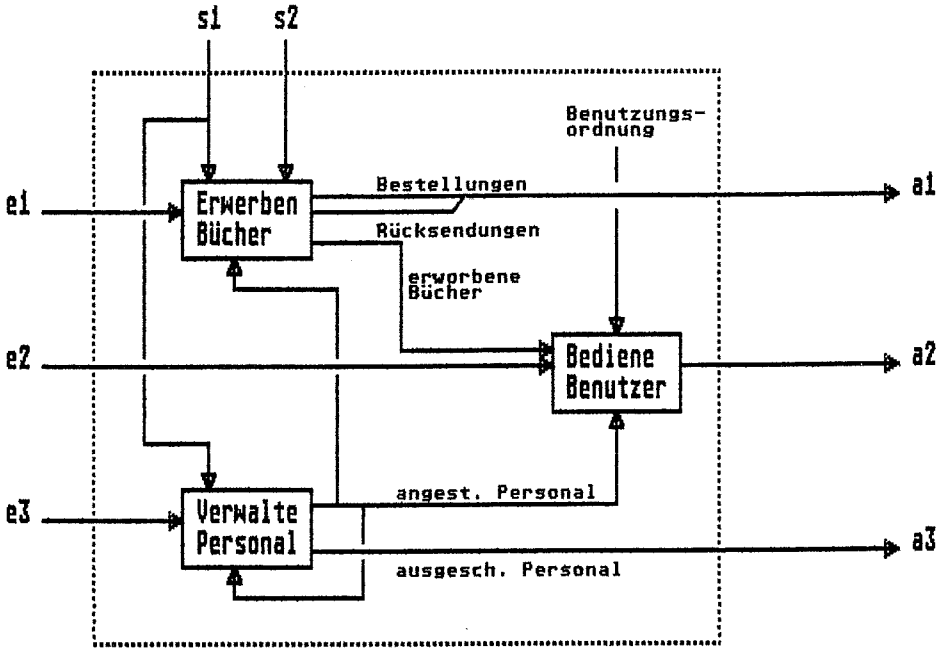
Zur Illustration der SADT Darstellungselemente zitieren wir ein Beispiel aus [KST], nämlich die ersten Diagramme, welche etwa im Zuge der Analyse einer Bibliotheksverwaltung entstehen mögen.




Das obige Diagramm D-0 zeigt die Bibliotheksverwaltung in ihrer Umgebung. Sie erhält Inputs von Buchhändlern bzw. Verlagen, von den Benutzern der Bibliothek, und schließlich stellt sie Personal ein. Personal kann die Verwaltung auch wieder verlassen. Outputs werden außerdem wiederum sowohl an Buchhändler/Verlage als auch an die Benutzer gegeben. Das Funktionieren der Bibliotheksverwaltung hängt zudem in hohem Maße ab von den verfügbaren Mitteln (Etat) und den allgemeinen Richtlinien für die Zusammensetzung des Buchbestandes (Bestellpolitik). Beides wird in der Regel von außen diktiert.

Eine SADT-"Faustregel" besagt, daß sich in einem Verfeinerungsschritt nicht weniger als drei und nicht mehr als sechs neue "Kästchen" ergeben sollten. Wir möchten als weitere Faustregel hinzufügen, daß die Funktion dieser Kästchen entweder mit den diversen Inputströmen oder den Outputströmen oder beidem zu tun haben sollte. (Nach Kombination beider "Regeln" erhält man dann die Empfehlung, je Kasten nicht mehr als drei Inputströme hinein und nicht mehr als drei Outputströme hinaus zu lassen.)

Wir erkennen folglich drei große Tätigkeitskomplexe innerhalb der Bibliotheksverwaltung. Der erste betrifft den Erwerb von Büchern über Buchhändler und Verlage; der zweite die Bedienung der Benutzer, und im dritten beschäftigt sich die Verwaltung mit ihrem eigenen Personal. Diese Bereiche sowie die Daten- und Objektflußbeziehungen zwischen ihnen zeigt Diagramm D0, die erste Verfeinerung der äußeren Systemsicht D-0:

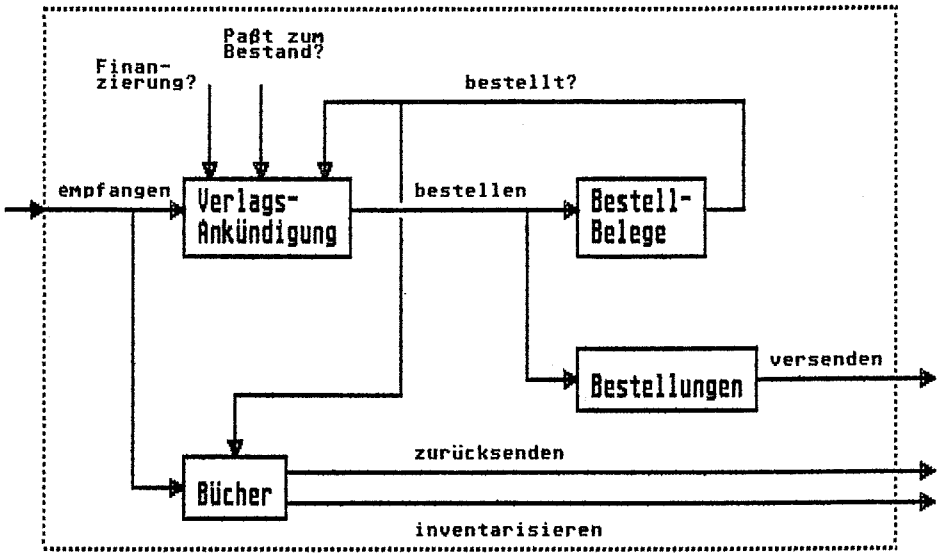



Diese erste Sicht "in das System" vermittelt eine gewisse Vorstellung vom Umgang mit Datenflüssen und "sonstigen Pfeilen" gemäß SADT. Einige Beobachtungen scheinen dabei erwähnenswert:

- Die Outputs "Bestellungen" und "Rücksendungen" der Systemkomponente "Erwerben Bücher" fließen zusammen in den Outputstrom des Gesamtsystems an die Buchhändler.
- Die *Kontrollgröße* "Etat" steuert sowohl den Bereich "Erwerbe Bücher" als auch die Aufgaben von "Verwalte Personal". Der entsprechende Pfeil verzweigt sich daher innerhalb von D0.
- Das "angestellte Personal", einer der Outputs von "Verwalte Personal", übernimmt die Rolle des "prozessierenden Mechanismus" für die Funktionen aller drei Bereiche, also auch für "Verwalte Personal" selbst.
- Mit der "Benutzungsordnung" hat "Bediene Benutzer" eine "Kontrollgröße", welche von außen nicht sichtbar ist.

Betrachten wir zum Beispiel die weitere Analyse der Komponente "Erwerben Bücher". Der Inputstrom e1 (von Buchhändlern/Verlagen) muß dabei offensichtlich differenzierender betrachtet werden. Einen Anhaltspunkt dafür liefert uns

das SADT-Datenmodell, in diesem Fall bezogen sowohl auf die Daten bzw. Objekte, welche mit e1 transportiert werden, als auch auf die Daten und Objekte, die durch "Erwerben Bücher" erzeugt werden:



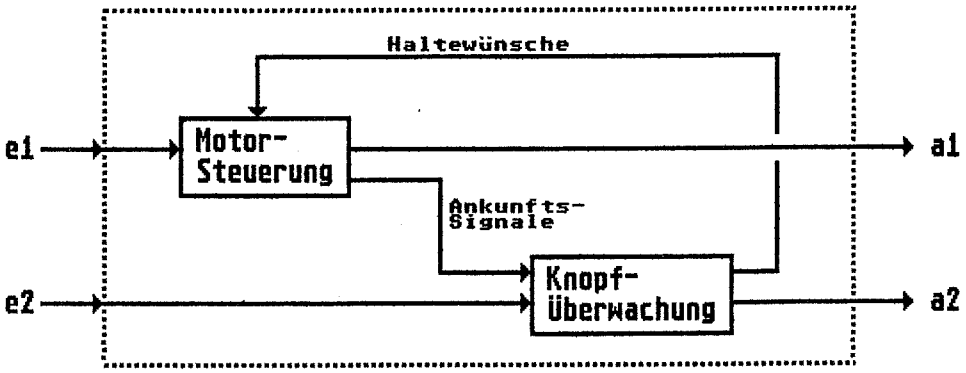

Im Inputstrom e1 gibt es Verlagsankündigungen und Bücher. Beide werden über den "Erzeugungsprozeß" "Empfangen" dem System zugeführt. Die an sie gebundenen "Verbraucherpfeile" machen deutlich, in welche Komponenten der Tätigkeitsbereich "Erwerben Bücher" zerfällt: "Bestellen", "(Bestellungen) Versenden", "(Bücher) Zurücksenden" und "(Bücher) Inventarisieren". Offensichtlich wird "Inventarisieren" dann diejenige Komponente von "Erwerben Bücher", von welcher der Output "erworbene Bücher" dem Tätigkeitsbereich "Bediene Benutzer" zufließt. Interessant ist in dem obigen Diagramm auch die Möglichkeit, ganz klar zu sagen, daß der Vorgang des Bestellens außer der Bestellung selbst gewissermaßen einen internen Vermerk erzeugt (den Bestellbeleg), der als Bedingung an die Bearbeitung von Verlagsankündigungen geknüpft wird. Daß die Bedingungspfeile "Finanzierung?" und "Paßt zum Bestand" unmittelbar von den Kontrollgrößen "Etat" und "Bestellpolitik" abgeleitet sind, erwähnen wir noch en passant. Der Leser möge sich nun selbst die (geringe) Mühe machen, die Komponente "Erwerben Bücher" entsprechend zu verfeinern.

Wir gewinnen anhand dieses Beispiels durchaus den Eindruck, daß SADT eine ausdrucksmächtige "Sprache" für die Beschreibung von Systemen unseres Interesses beinhaltet. Andererseits müssen wir zugeben (oder eingestehen?), daß es doch einige Ungewißheit bei der Interpretation von Darstellungselementen gibt. So ist es oft keineswegs klar, ob Daten oder Objekte als Inputs oder als "Kontrollgrößen" an einen "Prozeß-Kasten" herangeführt werden sollen. Ferner sehen wir die Grundvorstellung, die wir uns von Systemen gemacht haben (Netzwerke von miteinander kommunizierenden Prozessen), nur ungenügend unterstützt. Immerhin ist es möglich, Input-/Outputpfeil-Verbindungen im SADT-Diagramm

als Kommunikation via Kanäle und "Kontrollgrößen"-Verbindungen als Kommunikation via Variablenspeicher zu deuten. Ein Versuch, das Fahrstuhlsystem aus dem vorigen Abschnitt mit den Mitteln des SADT darzustellen, mag dies belegen:

SADT-Diagramm D-0:


SADT-Diagramm D0:

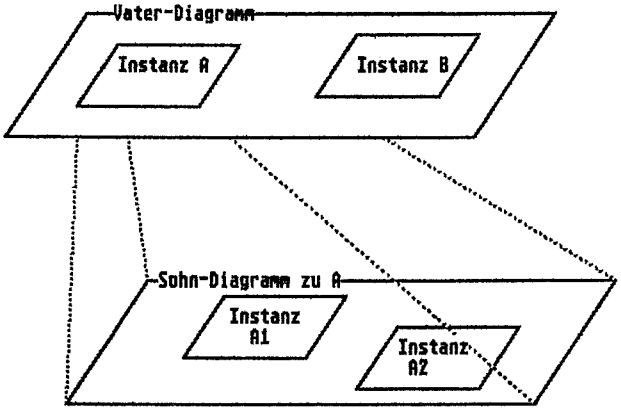



Dennoch: Der Leser mache sich klar, um wieviel aussagekräftiger die Unterlagen zur Spezifikation des Fahrstuhlsystems sind, welche wir mit Hilfe des JSD gewonnen haben. Ohne wesentliche Erweiterung der Ausdrucksmittel ist es mit SADT allein nicht möglich, z.B. Parallelität, Mehrfachbeziehungen und die Struktur von Datenströmen darzustellen.

An dieser Stelle bietet es sich an einzufügen, daß sämtliche der bisher in Abschnitt 6.3 behandelten graphischen Formalismen, von den Petrinetzen bis hin zu SADT, nach Rechnerunterstützung geradezu verlangen: Unterstützung bei der Anfertigung von Spezifikationsdokumenten, aber auch bei deren Verwaltung. Es ist hier nicht der Ort, sich im einzelnen über Systeme auszulassen, welche die für solche Aufgaben geeigneten Werkzeuge anbieten (vgl. dazu z.B. [BAL]). Solche Systeme sind zweifellos wichtige Bestandteile der Gesamtheit aller Arbeitsmittel, die einem Software-Ingenieur zur Verfügung stehen sollten. Im Englischen wird diese Gesamtheit auch als *Software Engineering Environment* bezeichnet und Rechnerunterstützung in demselben als *CASE = Computer Assisted Software Engineering.* CASE, nur soviel sei hier gesagt, gibt es in der Tat für Gane-Sarson-Diagramme und Petrinetze, für JSD System- und Datenstrukturdiagramme, für Klammerdiagramme und SADT und für viele andere graphische, halbgraphische oder rein textliche Systembeschreibungsverfahren. ([BAL] gibt einen recht guten Überblick über den aktuellen "CASE-Markt".)

Es überrascht daher nicht, daß auch für die zweite hierarchische Modellierungstechnik, die wir hier betrachten wollen, "ein CASE existierte". Bei diesem

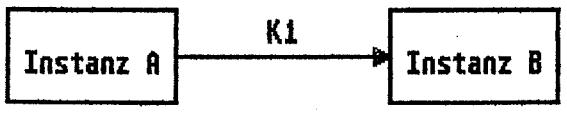
"CASE" handelte es sich um ein Produkt namens "INNOVATOR-RAD"*) (RAD = Requirements Analysis and Design), welches inzwischen jedoch nicht mehr vermarktet wird. Gleichwohl kann diese Technik - und das ist der Grund, weshalb wir ihr unsere Aufmerksamkeit widmen - sowohl unter syntaktischen als auch semantischen Gesichtspunkten als etwas raffiniertere, aber dennoch relativ leicht handhabbare Version des SADT aufgefaßt werden. Was im SADT als "System", "Komponente" und "elementare Systemfunktion" erscheint, wird nun unter dem Oberbegriff *Instanz* subsumiert, und verschiedene Arten der Kommunikation zwischen Instanzen werden unterschieden (vgl. [MID]). Über die Grundform der von RAD gelieferten Dokumentation gibt es nichts prinzipiell Neues zu sagen. Die spezielle, auf die hierarchischen Beziehungen zwischen Diagrammen bezogene Terminologie geht aus dem folgenden Bild hervor:



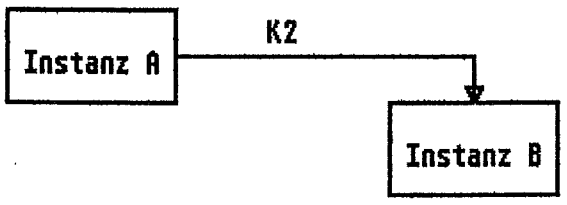

Der Vater hat zwei Söhne, die Instanzen A und B.

Sohn A hat seinerseits zwei Söhne, A1 und A2.

Auch in RAD-Diagrammen werden Instanzen repräsentierende Rechtecke - wie kaum anders zu erwarten - durch Pfeile miteinander verbunden. Die Semantik dieser Pfeile jedoch unterscheidet sich beträchtlich von der Bedeutung ihrer SADT-Pendants. Pfeile stellen in jedem Falle Kanäle dar, über die Daten ("Informationsobjekte") oder sonstige Objekte in der durch die Pfeilspitze bezeichneten Richtung transportiert werden. Mit dem Begriff "Kanal" verbindet sich dabei die uns bereits bekannte Vorstellung (s.o. Abschnitt 6.3.1). Je nachdem freilich, wo die Pfeilspitze endet, ergibt sich für den jeweiligen Kanal eine andere Betrachtungsweise:



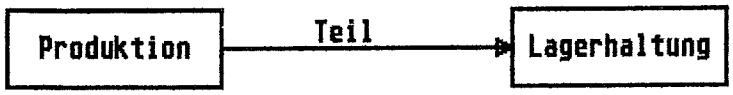

Instanz B stellt anderen Instanzen den Kanal K1 zur Verfügung.
B ***besitzt (ist owner von)*** K1.
Hier: A ***benutzt (ist user von)*** K1. A ***liefert*** (Informations-)Objekte an B.



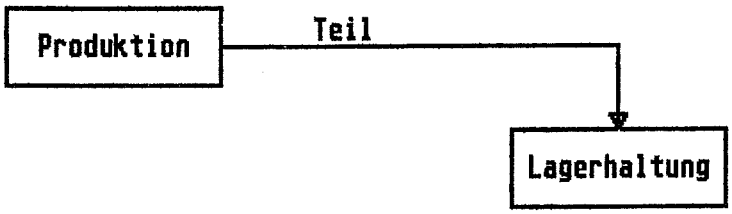

Instanz A stellt anderen Instanzen den Kanal K2 zur Verfügung.
A ***besitzt (ist owner von)*** K2.
Hier: B ***benutzt (ist user von)*** K2.
B ***holt*** (Informations-)Objekte von A.

*) INNOVATOR ist eingetragenes Warenzeichen der Firma MID GmbH, Nürnberg

Das folgende Beispiel zeigt, daß die Kommunikationsbeziehung zwischen zwei Instanzen unter diesen beiden komplementären Aspekten gesehen werden kann. Welchem Aspekt dabei der Vorzug zu geben ist, muß sich offenbar nach dem jeweiligen Modellierungsinteresse entscheiden.

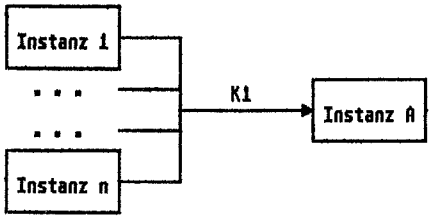



Die Produktion ***liefert*** Teile über den Kanal "Teil" and die Lagerhaltung. Die Produktion ist ***aktiv***, die Lagerhaltung verhält sich ***passiv***.

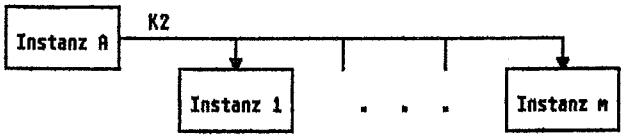



Die Lagerhaltung ***holt*** sich Teile über den Kanal "Teil" von der Produktion. Die Lagerhaltung ist ***aktiv***, die Produktion verhält sich ***passiv***.

An ein und demselben Kanal können auch mehrere Instanzen angeschlossen sein. Die dabei jedoch zu beachtende Grundregel ist, daß ein Kanal nur einen Besitzer haben darf. Außerdem gestattet RAD (naheliegenderweise) nicht, daß eine Instanz sich mit den gleichen Objekten beliefert, die sie selbst produziert, bzw., daß sie sich Objekte holt, die sie selbst herstellt. Daraus ergeben sich die folgenden weiteren möglichen Kommunikationsbeziehungen:

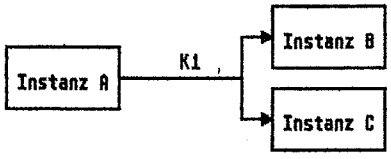


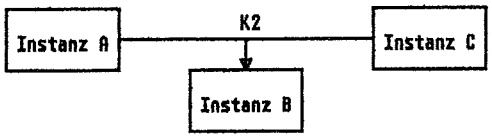

Instanz A (der Besitzer von K1) wird von den Instanzen 1 ... n über den gleichen Kanal K1 beliefert.



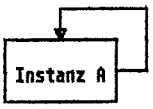

Die Instanzen 1 ... m holen sich (Informations-)Objekte aus dem gleichen, im Besitz von A befindlichen Kanal K2.

Dagegen sind die folgenden Verbindungen nicht erlaubt:



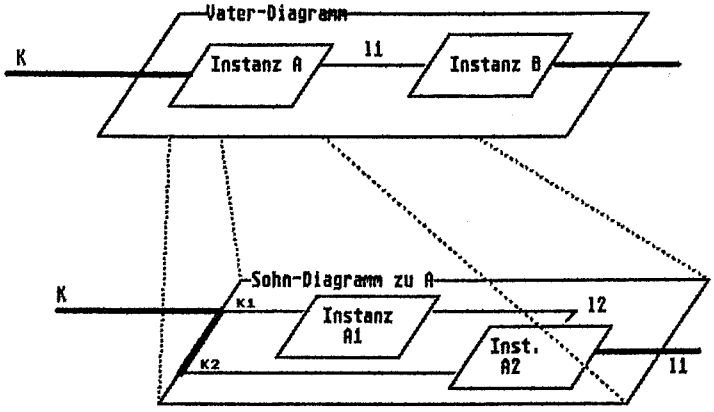



In diesen beiden Fällen hätte derselbe Kanal (K1 bzw. K2) zwei verschiedene Besitzer.

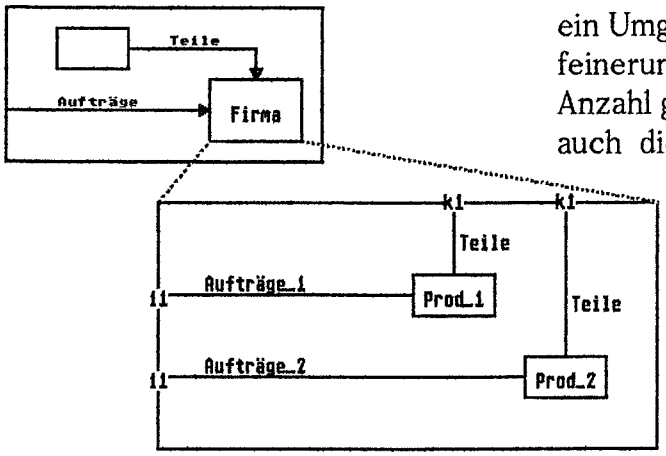




Hier liefert (holt) eine Instanz von ihr selbst produzierten Output an (von) sich selbst.

Auch die Verzweigung bzw. Aufspaltung von Objektströmen wird - entsprechend den Bemerkungen zu Beginn dieses Abschnitts auf den Begriff gebracht.




Der Rand eines eine Instanz repräsentierenden Rechtecks markiert die Grenze zwischen der Instanz selbst und ihrer Umgebung (*environment*). Ein am Rand einer Instanz auftreffender Kanal wird daher auch als *Umgebungs-Kanal (Environment-Channel)* (dieser Instanz) bezeichnet. Das Bild zeigt die an der Grenze zu einer Instanz stattfindende Aufspaltung eines solchen Umgebungs-Kanals (hier der Kanal K) in *verfeinerte Kanäle (refined channels)* (hier die Kanäle K1 und K2). Kanäle zwischen Instanzen innerhalb eines Diagramms (also nach dessen Verfeinerung) werden *lokale Kanäle (local channels)* genannt. Im Bild sind dies l1 und l2. Man beachte, daß der lokale Kanal l1 des Vater-Diagramms zu einem Umgebungs-Kanal im Diagramm des Sohnes A wird. Entsprechend würde bei einer weiteren Verfeinerung von Sohn B der Kanal l1 zu einem Umgebungs-Kanal bezüglich B. In beiden Fällen dürfte l1 natürlich ebenfalls einer Verfeinerung unterzogen werden. Andererseits kann ein Umgebungs-Kanal anläßlich der Verfeinerung eines Diagramms auch in eine Anzahl gleicher Exemplare (die also dann auch die gleichen Objekte transportieren) vervielfacht werden. Das nebenstehende Beispiel zeigt beide Möglichkeiten.




Hier zerfällt der Umgebungs-Kanal "Aufträge" der Instanz "Firma" in die verfeinerten Kanäle "Aufträge-1" und "Aufträge-2".

Dagegen wird der Umgebungs-Kanal "Teile" der Instanz "Firma" im Sohn-Diagramm dieser Instanz unzerlegt aber verdoppelt übernommen und beiden Produktions-Instanzen zugeführt.

Wir verzichten an dieser Stelle auf die Ausarbeitung eines ausführlicheren Beispiels zum RAD. Stattdessen möge der Leser selbst versuchen, die bisherigen Beispiele des Abschnitts 6.3 unter Verwendung dieser Modellierungstechnik nachzuvollziehen, um so einen Eindruck von ihren Stärken und Schwächen zu bekommen.

Am Ende dieses Abschnitts angelangt bleibt festzustellen und zu betonen, daß es für die Aufgaben der Systemanalyse und der Systemspezifikation gewiß kein universell brauchbares Verfahren geben kann. Ein Modellierungsverfahren ist für die Bearbeitung eines bestimmten realen Problems so gut, wie sich dieses Problem mit den Konzepten, die dem Verfahren zugrundeliegen, formulieren läßt. Es ist wie mit einem Modellbaukasten: Was man damit anfangen kann, hängt offensichtlich von seinem Inhalt ab. Enthält er keine Räder, so kann man keine Fahrzeuge modellieren. Bietet ein Verfahren zur Modellierung von Realwelt-Systemen keine Konstrukte zur Beschreibung parallel stattfindender Vorgänge, so müssen wichtige (und oftmals entscheidende) Aspekte eines Systems unberücksichtigt bleiben. Andererseits macht es, wenn bestimmte Aspekte keine Rolle spielen, wenig Sinn, ein Verfahren anzuwenden, das speziell auf die Erfassung dieser Aspekte ausgerichtet ist.

## 6.4 Systementwurf und Systemimplementierung

Ein Entwurf, so definiert es das "Große Wörterbuch der deutschen Sprache" ([DGD]), ist in erster Bedeutung eine "*Zeichnung, nach der man etwas ausführt, anfertigt: der Entwurf eines Hauses, ...* ". In zweiter Bedeutung, so finden wir, ist er die "*schriftliche Festlegung einer Sache in ihren wesentlichen Punkten: der Entwurf einer Verfassung, zu einem Roman, ...* ". Für unser Thema sind beide Erklärungen nützlich. Ersetzen wir nur *Haus* durch *Softwaresystem*, so fordert uns die erste Erklärung auf, ein gewünschtes System so detailliert durch eine Zeichnung zu beschreiben, daß es anschließend direkt angefertigt, "gebaut", werden kann. Sie legt eine Analogie nahe zwischen Software-Ingenieur und Architekt. Wie der Architekt wird jedoch auch der Software-Ingenieur nicht ohne *schriftliche Festlegungen* auskommen, um seinen Plan hinreichend deutlich zu formulieren, und folglich können wir auch in der zweiten Erklärung die "Sachen" *Verfassung* oder *Roman* durch *Softwaresystem* ersetzen. Mit ihr wird zudem der textliche Charakter von Software betont: schließlich wird diese ja "nur" geschrieben!

Insbesondere die erste Erklärung grenzt das Thema *Entwurf* recht scharf ab: gegen *Systemanalyse* und *Systemspezifikation* auf der einen Seite, und auf der anderen Seite gegen die nachfolgende Phase der *Implementierung*. (Vgl. auch die entsprechende Abbildung in Abschnitt 6.1!) Die Ergebnisse von Systemanalyse und -spezifikation sind eben nicht derart, daß ihnen die *Anfertigung* unmittelbar folgen könnte. Um bei der Analogie zum Architekten zu bleiben: Die Spezifikation des Hauses unternimmt dieser zusammen mit dem Bauherrn nach dessen Wünschen und aufgrund einer sorgfältigen Analyse der örtlichen Gegebenheiten. Die Größe und die relative Lage der Räume mögen dabei schon festgelegt werden, doch die den Bau ausführenden Handwerker benötigen wesentlich mehr Informationen. Und diese Informationen werden in detaillierteren Plänen, mit dem *Entwurf* eben, vom Architekten bereitgestellt.

Ähnlich ist es in unserem Falle: Systemanalyse und -spezifikation liefern eine den Anforderungen des Auftraggebers entsprechende Sicht des Systems, zum Beispiel als Netzwerk von Prozessen und Datenspeichern. Diese Darstellung ist weitgehend unabhängig von den Mitteln der späteren Realisierung. Der "Architekt" eines Softwaresystems muß daher seinen "Bauleuten" (den *Implementierern*) die Programme genau beschreiben, welche zur

- Ausführung und Steuerung der Prozesse, für
- die Kommunikation der Prozesse miteinander, für
- die Verwaltung der Daten und den Zugriff auf die Daten

erforderlich sind. Letzteres setzt ferner voraus, daß er entsprechende Hinweise zur Organisation der Daten gibt. Er muß also - und hierzu haben wir bereits in Abschnitt 6.1 einige Bemerkungen gemacht - die Teile (Bausteine!) *spezifi-*

*zieren*, aus denen das System zusammengesetzt werden soll, und die Art und Weise ihrer Verbindung. Er setzt das "WAS" der Systemspezifikation um in ein "WIE" der *Software-Architektur*. Obwohl er bei der Spezifikation der Teile dieser Architektur nach Möglichkeit vermeiden sollte, seinen Bauleuten zu einengende Vorschriften für die Realisierung zu geben, sich also hier seinerseits auf das jeweilige "WAS" beschränken sollte, kann er es - wie der reale Architekt - mitunter nicht vermeiden, schon im Entwurf "des Gebäudes" gewisse Gegebenheiten des "Geländes" in Betracht zu ziehen. Für den Software-Architekten und seine Implementierer ist dieses Gelände zum Beispiel u.a. gegeben durch

- Betriebssystem,
- Dateisystem,
- evtl. Datenbank-Management-System (DBMS),
- Software zur Gestaltung von Benutzungsoberflächen (vgl. 6.2.4)
- Programmiersprachen

und nicht selten durch die Hardware selbst. Es ist die in Abschnitt 6.1. erwähnte *Basis-Maschine*, mit deren Feinheiten letztlich natürlich die Implementierer zu tun (und manchmal zu kämpfen) haben. In welchem Maße jedoch schon die Architektur von ihr abhängen kann, wollen wir an einem einfachen Beispiel zeigen.

Wir nehmen an, daß laut Systemspezifikation die folgenden, durch Kanäle miteinander verbundenen Prozesse innerhalb eines Softwaresystems zu realisieren seien:


Die Architektur des dieser Prozeßkette entsprechenden Systems kann sich an der vorhandenen Hardware und ihrer Betriebssoftware orientieren, und sie muß dies unter Umständen sogar. Wir unterscheiden drei Voraussetzungen dieser Art:

(i) Mit der Hardware stehen uns mindestens drei gleichberechtigte und universelle (d.h. für alle möglichen Aufgaben geeignete) Prozessoren zur Verfügung (*Multiprozessor* Rechnerarchitektur), sowie ein Betriebssystem, das es gestattet, diese Prozessoren unabhängig voneinander zu beschäftigen und Daten zwischen ihnen auszutauschen.

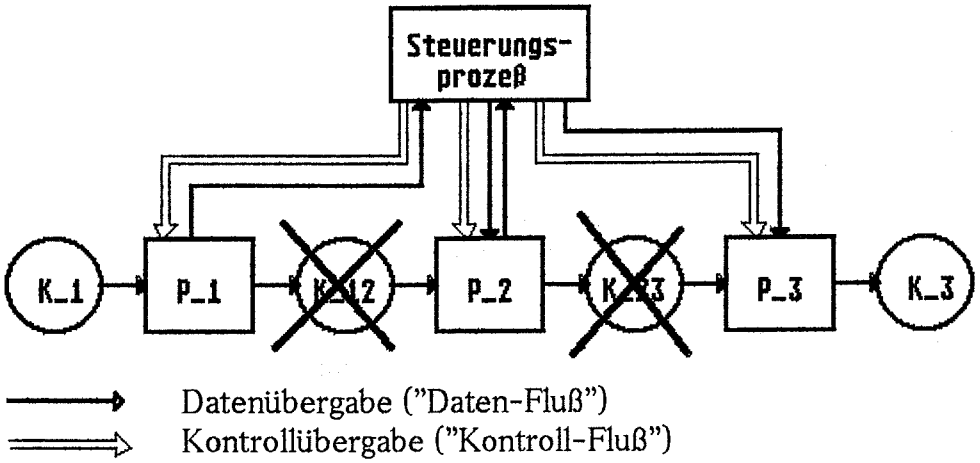
(ii) Die Hardware enthält nur einen universellen Prozessor (*Monoprozessor*), wird aber von einem Betriebssystem gesteuert, das den Prozessor jeweils und abwechselnd für kleine Zeiteinheiten (sog. *Zeitscheiben*) verschiedenen, voneinander unabhängigen Prozessen (Aufgaben, *Tasks*) zuteilt (*Time-Sharing* und *Multi-Tasking/Multi-Processing*). Es ermöglicht ferner den Austausch von Daten zwischen den Prozessen.

(iii) Wir haben eine Monoprozessor-Hardware, deren Betriebssystem keinerlei Unterstützung von *Multi-Processing* und entsprechender Prozeß-Kommunikation anbietet.

Die beiden ersten Voraussetzungen bieten für den Entwurf des Softwaresystems offenbar wenig Probleme. Im ersten Fall können wir die drei Prozesse eins-zu-eins drei realen Prozessoren zuordnen, und im zweiten Fall findet die eins-zu-eins-Zuordnung zwischen Prozessen und durch Zeitscheiben realisierten *virtuellen* Prozessoren statt. (Wie in diesen Fällen die Kommunikation von Prozeß zu Prozeß mit Hilfe geeigneter Sprachmittel beschrieben und *implementiert* werden kann, ist allerdings eines der Probleme, mit denen wir uns im letzten Unterabschnitt (6.4.5) dieses Abschnitts beschäftigen werden.)

Die dritte Basis-Maschine erfordert einige Entwurfs-Überlegung. Prozeß-Kommunikation und Prozeß-Steuerung müssen mit "architektonischen" Mitteln, d.h. durch geeignete Anordnung und Verbindung der die Prozesse ausführenden Teile des Systems, speziell gestaltet werden.

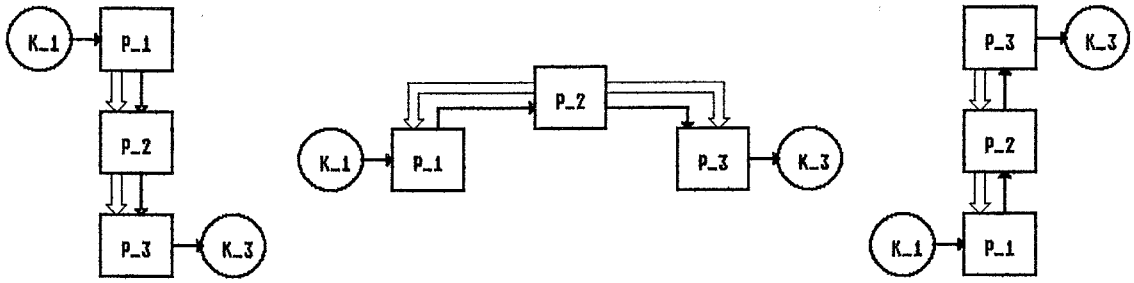
Eine erste Möglichkeit zeigt das folgende Diagramm:




Ähnliche Bilder sind uns bereits in Kapitel 3 begegnet (S. 73, 79), wo auch die beiden Pfeilarten erläutert wurden. Die Abfolge der Prozesse und der Fluß der Daten über die Kanäle werden von einem eigens konzipierten Systemteil, genannt "Steuerungsprozeß" (*"Scheduler"*), organisiert. Er übergibt - nachdem er selbst gestartet wurde - die Kontrolle über den Prozessor an den Systemteil P1, welcher Input vom Kanal K1 holt und seinen Output dem Steuerungsprozeß zur weiteren Verwendung "hochreicht". Gleichzeitig erhält der Steuerungsprozeß die Kontrolle über den Prozessor zurück. Das gleiche Spiel erfolgt mit P2 und P3 und wiederholt sich dann, wieder mit P1 beginnend.

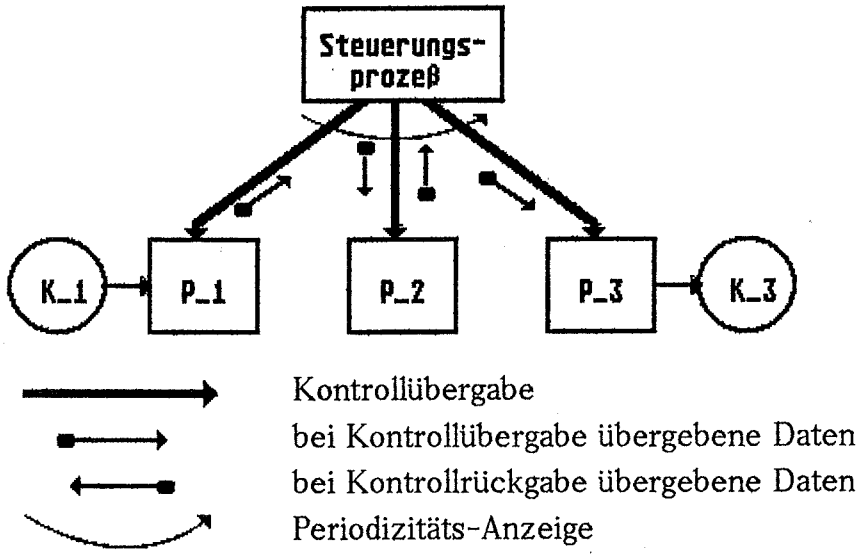
Auf die Einführung eines zusätzlichen Steuerungsprozesses kann man andererseits auch verzichten. Man kann, alternativ, jedem der drei Prozesse P1, P2 oder P3 "die Führung anvertrauen". Dies bedeutet, in den die jeweiligen Prozesse

ausführenden Systemteilen die Übergabe von Kontrolle und Daten an die jeweils anderen Systemteile in geeigneter Weise einzubauen. (Jackson spricht in diesem Zusammenhang von der *Invertierung* eines Prozesses gegen einen Kanal.) Die drei folgenden Diagramme zeigen diese Konfigurationen:




Somit haben wir vier mögliche Architekturen für ein System, welches die einfache Kette von drei über Kanäle miteinander kommunizierenden Prozessen realisiert. Natürlich müssen wir zugeben, daß dabei einiges von der in der ursprünglichen Systemspezifikation enthaltenen Semantik der sequentiellen Arbeitsweise unseres Prozessors geopfert worden ist. Beispielsweise haben wir einen Kanal als ein Transportmedium unbeschränkter Kapazität verstanden. Gemäß jedem der hier vorgeschlagenen Entwürfe laufen die Prozesse jedoch niemals gleichzeitig. Es wird "zwischen zwei - in der Kette - benachbarten" Prozessen nur maximal ein Datenobjekt stehen. Lediglich für den Input- und den Outputkanal können wir die Fiktion der Unbeschränktheit aufrechthalten.

Eine wichtige Information zum Ablauf der Prozesse ist in keinem der obigen Diagramme enthalten. Die Periodizität bestimmter Vorgänge oder Vorgangsketten kommt nicht zum Ausdruck. Mit einer auf Yourdon und Constantine ([YOC]) zurückgehenden graphischen Entwurfsnotation läßt sich dieser Mangel beheben und zum Beispiel unsere "Scheduler"-Lösung wie folgt skizzieren:



(Zur Übung möge der Leser die drei Invertierungslösungen mit Hilfe dieser Notation darstellen.)

Im weiteren Verlauf dieses Abschnitts (6.4.1) werden wir zunächst etwas ausführlicher über die allgemeinen Ziele des Systementwurfs sprechen, sowie ebenfalls allgemein über grundlegende, zur Erreichung dieser Ziele geeignete Techniken. Im zweiten Unterabschnitt (6.4.2) wird es dann speziell um die in 6.1 als dominierend charakterisierte Entwurfsaktivität gehen: das *Zerlegen* bzw. *Modularisieren.* "Was ist ein Modul" und "Welche Arten von Modulen gibt es" gehören zu den Fragen, die es zu beantworten gilt. Einer Formalisierung des Modulbegriffs, den *Abstrakten Datentypen,* ist Unterabschnitt 6.4.3 gewidmet. Sie gibt uns die Möglichkeit, die *Schnittstellen* und die *Semantik* von Modulen gewisser Arten hinreichend exakt zu definieren. Weiters wird sie uns ein Werkzeug liefern zur Führung von Korrektheitsbeweisen bei der *abstrakten Implementierung* solcher Module. Ferner werden wir die grundlegenden Konzepte einer in den letzten Jahren unter der Bezeichnung *Objektorientierung* sehr populär gewordenen Entwurfs- und Implementierungstechnik vorstellen. Der bisher in diesem Buch gelegentlich verwendete Begriff *Objekt* wird im Sinne dieser Technik präzisiert. Es wird sich zeigen, daß er eng mit den Begriffen *Modul* bzw. *Abstrakter Datentyp* verwandt, jedoch von noch größerer Tragweite ist. Schließlich werden wir uns mit verschiedenen Möglichkeiten der direkten Implementierung paralleler und kommunizierender Prozesse beschäftigen, mit Möglichkeiten, die ihrerseits erst unter dem Paradigma der *Abstrakten Datentypen* voll zu würdigen sind.

Eine gängige Klassifizierung von Software-Entwurfs-Aktivitäten unterscheidet zwischen

- Daten-Entwurf, dem
- Entwurf der Architektur eines Systems und
- prozeduralem Entwurf.

Es sei hier nochmals betont, daß wir uns in diesem Abschnitt im wesentlichen mit Aspekten des Entwurfs der Architektur, also des *modularen Aufbaus* von Softwaresystemen beschäftigen. Bezüglich des Daten-Entwurfs verweisen wir auf unsere Bemerkung in 6.3.1 über statische Modelle (S. 267). Und mit prozeduralem Entwurf (genauer, mit der Konstruktion von Programmteilen) haben wir uns in den Kapiteln 3-5 bereits ausführlich auseinandergesetzt.

### 6.4.1 Ziele und Techniken

Entwurf - unter der soeben eingeschränkten Betrachtungsweise - ist also die Herleitung einer Systemarchitektur aus der Systemspezifikation. Die Systemarchitektur soll zum einen die Gewähr dafür bieten, daß die das System implementierende Software so geschrieben werden kann, daß sie

(i) die in der Systemspezifikation festgelegten Anforderungen erfüllt (soweit sich dies formal oder informell nachprüfen läßt), und daß sie

(ii) (nach vorzugebenden Prioritäten) den in Abschnitt 6.2 diskutierten Qualitätskriterien genügt.

Im Falle großer Softwaresysteme, und diesen gilt ja unser besonderes Interesse, muß der Entwurf ferner dafür sorgen, daß

(iii) die Produktion der Software arbeitsteilig erfolgen kann und gut kontrollierbar ist.

Entsprechend der in Abschnitt 6.2.5 gegebenen Charakterisierung nimmt der Entwurf daher im Rahmen des Software-Engineering die zentrale Stellung ein.

Ziel (iii) macht es unter anderem offenbar notwendig, das gewünschte System nach Möglichkeit so in Teile ("Module", vgl. Abschnitt 6.1) zu zerlegen, daß diese weitgehend unabhängig voneinander beschrieben und verstanden werden können. Nur so ist zu erreichen, daß die Implementierungsarbeit in wohldefinierten Portionen verteilt und ohne übermässigen Zeitaufwand für Abstimmungen im Projektteam erledigt werden kann.

*Unabhängige Beschreibbarkeit*

und

*unabhängige Verstehbarkeit*

sind zwei der fünf Eigenschaften einer Kollektion von Modulen (des Ergebnisses der Zerlegung!), die B. Meyer in [MEY] als maßgeblich für die Beurteilung von Entwurfsmethoden herausstellt. *Unabhängige Verstehbarkeit* muß, obwohl oberflächlich gesehen synonym mit *unabhängiger Beschreibbarkeit*, doch eine zusätzliche Forderung sein, wenn Ziel (iii) erreicht werden soll. So mag es wohl sein, daß sich das "nach außen sichtbare" Verhalten eines Moduls unabhängig von allen anderen Modulen beschreiben läßt, daß aber die Implementierung dieses Moduls nicht ohne die Kenntnis der Implementierung anderer Module zu leisten ist. In einem solchen Fall bedarf es bei der Systemimplementierung einiger Mühe, um die jeweiligen Abhängigkeiten zu klären. Damit steigt der Aufwand für die Koordinierung des Projektablaufs. Andererseits kann die Wartbarkeit eines fertigen und laufenden Systems beträchtlich erschwert sein, wenn dessen Module nicht aus sich heraus verständlich sind. Hier sind die übrigen drei der von Meyer genannten Eigenschaften:

*Kombinierbarkeit,*

*Stetigkeit,*

*Abgeschirmtheit.*

*Kombinierbarkeit* bezeichnet die Eignung eines Moduls als *Baustein* zur Verwendung in anderen Zusammenhängen als dem, in dem er ursprünglich definiert wurde. Diese Eigenschaft ist in der Tat synonym mit der Software-Qualität

*Wiederverwendbarkeit* bezogen auf Module. Wir verweisen hierzu auf die entsprechenden Bemerkungen in Abschnitt 6.2.2.

Auch der Begriff *Stetigkeit* ist uns bereits früher, bei der Erörterung des Qualitätsmerkmals *Wartbarkeit* (in der speziellen Interpretation als *Adaptierbarkeit*, vgl. Abschnitt 6.2.3), begegnet und braucht daher an dieser Stelle nicht weiter vertieft zu werden.

Entsprechendes gilt im wesentlichen auch für die letzte der oben aufgezählten Eigenschaften, die in engem Bezug steht zu den in Abschnitt 6.2.1 vorgeschlagenen "Abschwächungen" des Korrektheitsbegriffs: *Prüfbarkeit*, *Integrität*, *Fehler-Toleranz*. *Abgeschirmtheit* bedeutet, daß die Auswirkungen von Fehlern, welche während der Benutzung von Software auftreten, auf einen oder wenige Module beschränkt sind. Wir meinen dabei - wie in 6.2.1 - zum Beispiel Fehler im Bereich der (zentralen oder peripheren) Hardware, fehlerhaften Input oder den Mangel an bei der Abarbeitung eines Programms benötigten Ressourcen.

Lassen sich, so lautet nun die Frage, diese noch recht allgemein formulierten Eigenschaften einer modularen Zerlegung (gewissermaßen aus den obigen Postulaten (i) - (iii) abgeleitete Ziele) als Folgen konkreter technischer Merkmale einer Systemarchitektur verstehen? Eine Antwort gibt B. Meyer ([MEY]) mit fünf Imperativen, die beim Entwurf einer Systemarchitektur zu beachten sind:

*(1) Textuelle Einheiten*:
Ein Modul soll in einer textuellen Einheit vollständig beschreibbar sein und nach Möglichkeit auch einer textuellen Einheit in der Implementierungssprache entsprechen.

Die stärkste Begründung für die Anwendung dieses Gebots ergibt sich aus den Forderungen nach *unabhängiger Beschreibbarkeit*, *Kombinierbarkeit* und *Abgeschirmtheit*. Tatsächlich ist dessen erster Teil identisch mit einer Forderung an die zur Beschreibung eines Moduls verwendete Sprache: Diese muß es erlauben, einen Modul als syntaktisch in sich abgeschlossenes Textstück darzustellen. Bei Verwendung einer Sprache, die dies nicht zuläßt, müßte die Beschreibung eines Moduls unter Umständen über mehrere Textstücke verteilt werden, und man liefe Gefahr, sie mit den Beschreibungen anderer Module zu vermischen. Wohlgemerkt: Wir meinen hier nicht notwendigerweise die Sprache, in der wir die einen Modul implementierenden Programme schreiben, sondern in erster Linie die zur Festlegung der Eigenschaften und Arbeitsweise eines Moduls benutzte Notation. Diese wird als *Entwurfssprache* (oder gelegentlich auch als *Modulbeschreibungssprache*) bezeichnet und kann von der Programmier- (bzw. *Implementierungs*-) Sprache durchaus verschieden sein.

Der zweite Teil des Gebots freilich betrifft die Implementierungssprache direkt, und es mag zunächst verwundern, daß wir sie an dieser Stelle überhaupt ins Spiel bringen, sollte doch die Architektur eines Gebäudes nicht unbedingt schon das Material erkennen lassen, aus dem die Wände bestehen! Andererseits:

Gleichgültig welches Baumaterial wir verwenden, die vom Architekten definierten Zellen sollten jede für sich eine Einheit bilden. Nur so können sie miteinander kombiniert werden, und nur so kann Fehlersuche (die ja letztlich im geschriebenen Programm stattfindet) auf einen wohlabgegrenzten Bezirk beschränkt bleiben. Und was die programmiersprachliche Form eines Moduls betrifft, so kann man noch weitergehen und verlangen, daß sie bezüglich eines gegebenen Compilers eine separat kompilierbare Einheit darstellt. Die Kompilate der Module sind dann die Software-Bausteine (die "Zellen"), aus denen das System zusammengesetzt wird. Sprachen, die diesen Forderungen genügen, sind zum Beispiel ADA und MODULA-2.

*(2) Wenige Schnittstellen*:
Ein Modul soll mit möglichst wenig anderen Modulen in Beziehung stehen.

(Was immer die Beziehungen zwischen Modulen sein mögen:) Unterhält ein Modul Beziehungen mit vielen anderen Modulen, so ist dies keinem der oben gesetzten Ziele förderlich. Insbesondere gilt: Je weniger Beziehungen zwischen den Modulen eines Systems bestehen, je schwächer also die gegenseitigen Abhängigkeiten sind, umso größer ist die Chance, einen Modul aus sich selbst zu verstehen (-> *unabhängige Verstehbarkeit*), und umso geringer werden die Auswirkungen lokaler Änderungen sein (-> *Stetigkeit*). Die Möglichkeit der Ausbreitung von Fehlern ist eingeschränkt (-> *Abgeschirmtheit*), und man hat bessere Aussichten, einen Modul in anderem Zusammenhang verwenden zu können (-> *Kombinierbarkeit*).

*(3) Schmale Schnittstellen*:
Miteinander in einer Kommunikationsbeziehung stehende Module sollen möglichst wenig Information austauschen.

Damit wird eine weitere Maßnahme zur Reduktion der zwischen Modulen existierenden Abhängigkeiten formuliert. Analog zu Gebot (2) gilt: Je geringer die Menge der Information ist, über die miteinander kommunizierende Module sich "verständigen" müssen, umso unabhängiger sind sie voneinander, desto schwächer sind sie aneinander gekoppelt. Starke Kopplung liegt zum Beispiel dann vor, wenn Module Datenspeicher gemeinsam benutzen. Dies ist nach Möglichkeit zu vermeiden.

*(4) Explizite Schnittstellen*:
Wenn zwei Module miteinander kommunizieren, dann soll dies aus den (entwurfs- bzw. implementierungssprachlichen) Modultexten klar ersichtlich sein.

Dieses Gebot bringt (wie schon im Falle der textuellen Einheiten) eine Forderung mindestens an die Modulbeschreibungssprache zum Ausdruck. Es verlangt die Existenz von Sprachkonstrukten zur Darstellung der Beziehungen zwischen Modulen. Und natürlich verlangt es die Verwendung dieser Konstrukte. Nun läßt sich der Zugriff aus getrennten (Entwurfs- oder Programm-) Texteinheiten auf

gemeinsam benutzte Datenspeicher in der Regel (d.h. ohne übermäßig gekünstelte Notationen zu erfinden) sprachlich nicht anders darstellen als der Zugriff auf (bezüglich der betroffenen Einheiten) private Datenspeicher. Mangels dieser klaren Unterscheidbarkeit empfiehlt also auch das Gebot der *expliziten Schnittstellen* - mit anderen Worten als sein Vorgänger - den Verzicht auf starke Kopplung mittels gemeinsam benutzter Datenspeicher. (Vgl. dazu auch die Beispiele in Abschnitt 2.2.2 zur Regel "*Seiteneffekte durch Funktionsprozeduren sind zu vermeiden.*")

*(5) Geheimhaltung ("Information hiding")*:
Über einen Modul soll nur soviel veröffentlicht werden, wie nötig ist, um ihn im Zusammenspiel mit anderen Modulen zu benutzen.

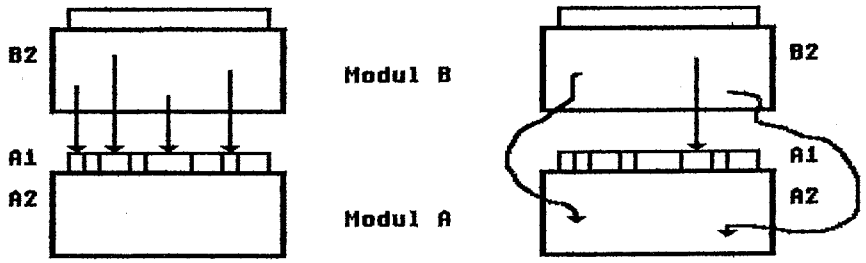
In der Tat: Dies klingt geheimnisvoll! Was soll es bedeuten? Ein der Welt der Geheimdienste entlehntes Gebot? Daß nur jeder über genau die Informationen verfügen darf, die er zur Ausübung seiner Tätigkeit braucht, und über keinen Deut mehr? Aber warum sollte ein Entwerfer nicht mit allem herausrücken, was er sich zu einem Modul ausgedacht hat? Warum sollte nicht der gesamte Modultext allen an einem Projekt Beteiligten bekannt sein? Nun, Gebot (5) ist sicherlich nicht so zu verstehen, daß große Teile der Projektdokumentation in einem Tresor mit gesonderten Schließfächern und Zugangsrechten für jeden Modul aufzubewahren wären. (Teamkollegen sollten doch voneinander lernen dürfen!)

Aber es sagt zum Beispiel wieder etwas aus über die Modulbeschreibungssprache (und - wie oben - schließlich über die Implementierungssprache): Die textuelle Einheit Modul A (um der Sache einen Namen zu geben) soll in mindestens zwei Abschnitte, A1, A2, ..., unterteilbar sein, von denen genau einer, sagen wir A1, all jene Informationen enthält, die bekannt sein müssen, um ganz A in - wie immer gewünschte - Beziehung zu anderen Modulen zu setzen. Für die möglichen Beziehungen eines Moduls zu anderen Modulen haben wir in den Geboten (2) - (4) den Ausdruck *Schnittstelle* gebraucht. Abschnitt A1 des Textes von Modul A muß also genau diese Schnittstelle beschreiben, und nicht mehr.

Wir können dies präzisieren: In unserer Formulierung des Gebots (5) haben wir die fundamentale Beziehung zwischen Modulen beim Namen genannt: *Benutzung*. Module *benutzen* sich gegenseitig, was sonst! Das heißt: Sie verlangen und erbringen Dienste. (Und wenn es nur der wäre, für eine Weile die Kontrolle des Prozessors zu übernehmen!) Entwerfer und Implementierer eines Moduls B müssen, wenn sie Modul A von B aus benutzen wollen, wissen, welche Dienste A unter welchen Bedingungen zu leisten imstande ist. Sie müssen nicht wissen, wie A diese Dienste ausführt. So weiß der Kunde einer Bank, der einen Geldbetrag von seinem Konto auf das Konto eines Geschäftspartners bei einer anderen Bank überweisen lassen möchte, im allgemeinen nicht, welche Vorgänge sich dabei innerhalb der jeweiligen Bank und zwischen den Banken abspielen. Er braucht es nicht zu wissen. Allein wichtig ist für ihn, daß der Betrag nach

angemessener Frist dem Empfängerkonto gutgeschrieben wird. Die gewünschte Dienstleistung unterliegt natürlich bestimmten Bedingungen, und sie hat einen wohldefinierten Effekt.

Zurück zu Modul A: Welche Dienste A anbietet und die Bedingungen ihrer Inanspruchnahme, das WAS des Moduls A also, dies ist der Inhalt des Abschnitts A1, des "öffentlichen" Teils von A. Die Ausführung dieser Dienste, das WIE, ist der Inhalt der Abschnitte A2, ..., des "privaten" Teils von A. (Selbstverständlich genügt es danach, immer von nur zwei Teilen eines Modultextes zu sprechen, dem öffentlichen und dem privaten.) Der private Teil kann nun tatsächlich "geheim" bleiben, gewissermaßen der Gegenstand einer Verschwörung zwischen dem Entwerfer und dem Implementierer des Moduls. Es wäre durchaus fatal und bezüglich unserer Ziele kontraproduktiv, wenn Entwurf und / oder Implementierung des Moduls B vom WIE des Moduls A abhinge. Änderungen im privaten Teil A2 würden dann mit großer Wahrscheinlichkeit auch Änderungen in B notwendig machen. Beispiel Bank: Das Verhalten des Bankkunden, der eine Überweisung in Auftrag gibt, braucht sich in der Regel nicht zu ändern, wenn sich die bankinternen Prozeduren ändern. Im Gegenteil: Die Banken, als gute Dienstleister, sind sicherlich darauf erpicht, ihre Kunden so wenig wie möglich von dem spüren zu lassen, was sich hinter den Kulissen abspielt, und Änderungen an den Schnittstellen - zum Beispiel die Einführung eines neuen Überweisungsformulars - nur höchst selten vorzunehmen. Und weiter: Wäre es dem Kunden erlaubt, zur Erledigung seiner Überweisung selbst direkt in das zwischenbankliche Geschehen einzugreifen, so könnte dies - durch böswillige Absicht oder fahrlässigerweise - durchaus nachteilige Auswirkungen sowohl auf die Banken als auch auf andere Kunden haben. Auf die Softwareentwicklung übertragen bedeutet dies: Wäre es dem Entwerfer / Implementierer erlaubt, bei der Gestaltung von Modul B sich der im privaten Teil von A *verborgenen Informationen* (über Daten und Prozeduren, zum Beispiel) zu bedienen, diese gar zu manipulieren, so könnten kaum kontrollierbare Auswirkungen auf das gesamte System die Folge sein. Wir sehen: Auch das Gebot der *Geheimhaltung* dient letztlich allen aus unseren Qualitätsansprüchen abgeleiteten Entwurfszielen.




"Legale" und "illegale"
Benutzung des Moduls A durch Modul B.

Das Prinzip *Information Hiding* wurde erstmals zu Beginn der siebziger Jahre von D.L. Parnas in mehreren (und seither vielzitierten) Artikeln ([PA1], [PA2]) formuliert. Die Einsicht in die überragende Bedeutung dieses Prinzips liegt zahlreichen, in der Folgezeit stattgefundenen Sprachentwicklungen zugrunde. Dabei meinen wir sowohl Entwurfs- als auch Programmiersprachen. Entwurfssprachen, soviel sollte nun klar sein, dienen in erster Linie der möglichst präzisen Beschreibung der Schnittstellen (also der "öffentlichen" Teile) von Modulen und von deren Verbindungen. Parnas selbst und viele andere Autoren haben hierzu Vorschläge unterbreitet. Die spektakuläreren (d.h. publikumswirksameren) Fortschritte freilich wurden auf dem Feld der Programmiersprachen gemacht. MODULA-2, die Sprache, auf die wir uns in diesem Buch stützen, ist ein Beispiel hierfür. (Im weiteren Verlauf dieses Kapitels werden wir sehen, wie sie im Sinne dieser Ausführungen anzuwenden ist.) Von den Sprachen, die wir hier im Auge haben, ist ferner ADA zu nennen ([LED]), deren Definition vom US-amerikanischen Verteidigungsdepartement in der zweiten Hälfte der siebziger Jahre in Auftrag gegeben wurde. Gemeinsam ist diesen Sprachen, daß sie die vom Geheimhaltungsprinzip nahegelegte Zweiteilung eines Modultextes in einen öffentlichen und einen privaten Teil fördern. Auch die *objektorientierten* Sprachen (vgl. Abschnitt 6.4.4) sind hinzuzurechnen, von denen die zur Zeit vielleicht populärste C++ (sprich: Zeh-plus-plus) genannt wird ([STR]). (Eine Weiterentwicklung der von Kernighan und Ritchie Anfang der siebziger Jahre im Rahmen der Implementierung des Betriebssystems UNIX geschaffenen Sprache C ([KER])). Bemerkenswerte Vertreter der Familie der objektorientierten Sprachen sind ferner (unter anderen!) das bei der Firma Xerox entstandene *Smalltalk* ([GOR]) sowie EIFFEL, eine Sprache, die von B. Meyer stammt, deren Gebrauch in seinem schon zitierten Buch ([MEY]) ausführlich beschrieben wird, und die wir in Abschnitt 6.4.4 vorstellen werden.

Doch besinnen wir uns: Wir fragen nach Zielen und Techniken des Systementwurfs. Die (allgemeinen) Ziele glauben wir hinreichend präzise gesetzt zu haben, und zum Thema Techniken gab es zumindest Andeutungen. Bevor wir uns im nächsten Abschnitt mit einigen, speziell zum besseren Verständnis von *Modularisierung* geeigneten Beispielen beschäftigen (,um so ein deutlicheres Gefühl dafür zu gewinnen, worauf es ankommt), sollten wir es uns durchaus noch hier angelegen sein lassen, zwei grundlegend verschiedene mögliche Vorgehensweisen beim Entwurf einer Systemarchitektur zu kommentieren. In der einschlägigen Literatur firmieren sie gewöhnlich unter ihren englischen Bezeichnungen *Top-Down* (von oben nach unten) und *Bottom-Up* (von unten nach oben). Von diesen ist uns der Begriff *Top-Down* schon mehrfach begegnet, zum erstenmal im Zusammenhang mit *Schrittweiser Verfeinerung*, einer Technik der "Programmierung im Kleinen" (oder, wie wir jetzt auch sagen können, des *prozeduralen Entwurfs*), und zum zweiten Mal bei der Darstellung hierarchischer Systemmodelle, als Charakterisierung einer Methode der Systemana-

lyse und -spezifikation. In beiden Fällen konnten wir einsehen, daß es sich um respektable Strukturierungsansätze handelt: *Schrittweise Verfeinerung* zur Herleitung der Struktur eines Programms, welches im wesentlichen eine (mehr oder weniger) genau festgelegte Funktion (bzw. Aufgabe) hat, und *hierarchische Modellierung* zur Ermittlung der "Anwendersicht" eines Systems, seiner Datenflüsse und Prozesse also. In beiden Fällen gibt es tatsächlich so etwas wie ein *Top*, eine oberste "grobe" Sicht der Dinge, die Spitze einer Pyramide sozusagen, von der aus man die "unteren Ebenen" (der Prozeduren und Statements bzw. der Daten und ihrer Verarbeitung) stufenweise erschließen kann. Welche Bedeutung, so ist nun aber zu fragen, hat der Begriff *Top-Down* für die Herleitung der Architektur eines Softwaresystems?

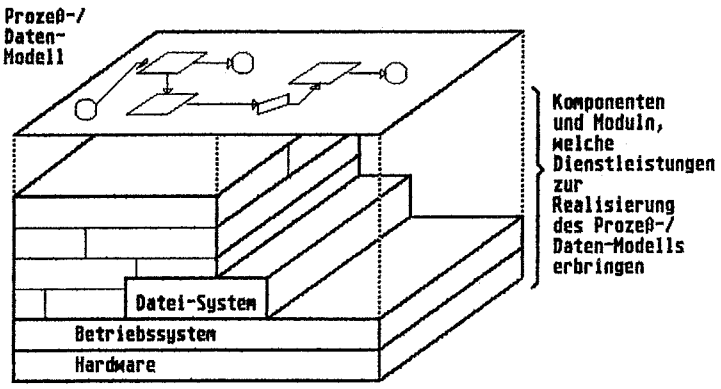
Unsere, manchem Fachgenossen vielleicht etwas zu kategorische Antwort lautet, daß sich der Begriff in bisheriger Interpretation im allgemeinen nicht zur Charakterisierung sinnvoller Vorgehensweisen beim Systementwurf eignet. Ein Ergebnis von Systemanalyse und -spezifikation besteht ja im allgemeinen gerade darin, daß sich die Leistungen eines Systems nicht als eine einzige ("Top"-) Funktion beschreiben lassen, sondern daß es eine Vielzahl von mehr oder weniger gleichrangigen Funktionen (Dienstleistungen) zu implementieren gilt. Für große Informations-, Transaktions- oder Prozeßsteuerungs-Systeme ist dies ziemlich offensichtlich. Aber selbst beim Entwurf eines Compilers, also eines Programms, das auf den ersten Blick nur die eine Funktion hat, in höherer Programmiersprache geschriebene Texte in Maschinensprache zu übersetzen, kann es sehr nützlich sein, einzelne Aufgaben, wie lexikalische Analyse, Parsing und Codegenerierung, getrennt zu betrachten, und den Programmaufbau nicht von vornherein von der zeitlichen Abfolge dieser Aktivitäten abhängig zu machen. So erhält man eine gute Chance, Teile des Compilers zukünftig vielleicht für ganz andere Zwecke (etwa einen syntaxgesteuerten Editor) (wieder) verwenden zu können.

Wir erwähnten Informations- und Transaktionssysteme. Natürlich wird man für diese in der Regel bei jeder praktischen Realisierung eine Benutzungsoberfläche vorfinden, also so etwas wie einen *Top*, und man kann sicher mit Berechtigung fragen, warum dieser nicht zum Ausgangspunkt des Entwurfs gemacht werden soll. Aus mehreren Gründen ist hier Skepsis angebracht: Erstens gehört die Benutzungsoberfläche eines Systems sicherlich zu den Teilen, für die künftige Anpassungen (an geänderte Kundenwünsche zum Beispiel) am leichtesten möglich sein sollten (dies ist kein Widerspruch zum obigen Bankbeispiel, im Gegenteil!); zweitens hat sie gewöhnlich nur vermittelnde Funktion, indem sie die Auswahl aus den Dienstleistungen des Systems ermöglicht, und damit streng genommen gar nicht zum System selbst gehört. (Vgl. aber dazu auch die Bemerkungen am Ende von Abschnitt 6.2.4!) Und drittens, gewissermaßen zur Verstärkung des zweiten Einwands, ist zu sagen, daß wohl die durch den Informationsfluß innerhalb des Systems gegebenen Verhältnisse an

der Benutzungsoberfläche in geeigneter Weise zum Vorschein kommen müssen, daß aber nicht umgekehrt die Benutzungsoberfläche diese Verhältnisse bestimmt. (Tatsächlich liegt es zum Beispiel im Interesse der Hersteller von Standardsoftware, Benutzungsschnittstellen so austauschbar wie möglich zu machen, um die Software, auf die es eigentlich ankommt, für möglichst viele Rechnertypen anbieten zu können.) Dies gesagt, dürfen wir selbstverständlich nicht leugnen, wie wichtig gute Benutzungsoberflächen sind, und daß sie mit großer Sorgfalt entworfen werden müssen. Davon haben wir aber bereits an anderer Stelle geschrieben (vgl. Abschnitt 6.2.4). (Manchmal scheint die Benutzungsoberfläche einem Kunden sogar wichtiger als das darunterliegende System, und die Anforderungsspezifikation beginnt dann mit der Anfertigung von "hohlen" Prototypen, also von Programmen, die einen Dialog mit dem (noch) nicht vorhandenen System simulieren. Oft ist der Grund für diese - *Rapid Prototyping* (vgl. z.B. [HAL]) genannte - Vorgehensweise die Schwierigkeit, sich ohne solche konkreten Hilfen ein Bild von den Leistungen eines Softwaresystems zu machen.)

Zu bemerken bleibt, daß sich die relative Popularität des "klassischen" *Top-Down*-Ansatzes für den Systementwurf - also der Zerlegung einer "Hauptfunktion" in "Teilfunktionen" und deren weiterer Verfeinerung - möglicherweise aus einer Verwechslung von Analyse mit Entwurf, von hierarchischem Systemmodell mit Softwarearchitektur ergibt.

Die umgekehrte Richtung, *Bottom-Up*, stand bisher noch nicht zur Debatte. Gemeint ist die Konstruktion eines Systems aus existierenden oder zu definierenden Bausteinen, "von unten nach oben". Nach dem, was wir in diesem Abschnitt über Ziele, Techniken und insbesondere über Module erfahren haben, drängt sich dieser Ansatz durchaus auf. Doch welche Bausteine sind zu wählen, welche wie zu definieren? Diese Fragen können offenbar nur mit Blick auf die gegebene Spezifikation des Systems als Ganzem beantwortet werden. Also doch *Top-Down*, nur mit anderem Verständnis? In der Tat, an einem Ansatz "ganz oben" kommt ein Software-Ingenieur sicher nicht vorbei. Er muß von dem abstrakten Modell des Systems ausgehen, wie es Analyse und Spezifikation ergeben haben, und diesem - wie es die nebenstehende Abbildung zeigt - "schichtenweise" Module unterlegen, von denen jeder ganz spezifische Dienstleistungen für die Realisierung des Modells erbringt. Die Module der obersten Schicht sind so zu de-



finieren, daß sie das System direkt implementieren, während die Module jeweils tieferer Schichten denen auf höherer Ebene zu Diensten sind. (Wir erinnern daran, daß wir diese Form der Hierarchisierung übrigens schon früher - in Abschnitt 6.2.3 - gewissermaßen "in der Nußschale" empfohlen haben: mit den Worten des Chefs des Softwarehauses ADAPTOR&Co, der seinem Lehrling den Rat gab, Programme wie arbeitsteilige, hierarchisch gegliederte Produktionsbetriebe zu organisieren.) In dieser Sicht sind auch Datei- und Betriebssystem (und letztlich die Hardware selbst) des Rechners nichts anderes als Module (bzw. Komponenten), und es bleibt zu hoffen, daß zumindest einige der über dieser Basissoftware liegenden Bausteine aus früheren Projekten oder aus "Standardbaukästen" (*Bibliotheken*) (mit eventuell geringen Änderungen) übernommen werden können.

Wir wollen uns bemühen, diese - in gewissem Sinne eine Synthese von *Top-Down* und *Bottom-Up* darstellende - Vorgehensweise in den folgenden Abschnitten anhand von Beispielen noch plastischer werden zu lassen.

### 6.4.2 Modularisierung

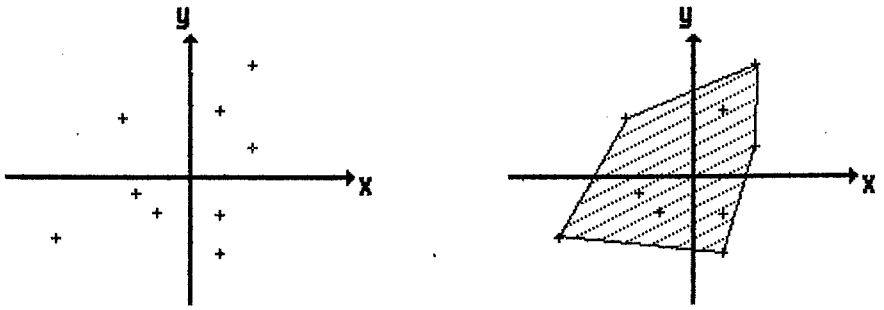
Die Kernfrage lautet: An welchen Elementen des Systemmodells soll sich die Definition von Modulen orientieren? Wie können wir erreichen, daß die Module unseres Systems die im vorangegangenen Abschnitt erläuterten Eigenschaften besitzen? Die Antwort hierauf können wir nur durch praktisches Tun finden, das sich - aus naheliegenden Gründen - in einem Buch wie diesem natürlich nicht auf lebensgroße Projekte erstrecken kann. Man kann dies zum Vorteil wenden und argumentieren, daß bei der Darstellung eines echten Projektes nur zu leicht der Blick auf das Wesentliche versperrt würde. Versuchen wir also, aus überschaubaren Beispielen unsere Lehren zu ziehen.

Als erstes wollen wir uns die Aufgabe stellen, ein Programm zu entwerfen, welches zu einer Punktmenge in der cartesischen Ebene die konvexe Hülle berechnet und auf einem geeigneten Ausgabegerät darstellt. In unserem Zusammenhang kommt es selbstverständlich nicht darauf an, das Berechnungsverfahren zu entwickeln. Dieses setzen wir als bekannt und gegeben voraus. Unser Ziel ist es, den Aufbau, die Architektur des Programms zu klären. Dennoch: Zur Verdeutlichung des Begriffs "konvexe Hülle einer Punktmenge in der Ebene" betrachte man die Abbildung auf der folgenden Seite.

Die Analyse des Umfelds der Anwendung des geplanten Programms hat eine Reihe von Anforderungen ergeben, von denen wir hier nur die wichtigsten nennen:

- Die Koordinaten (x und y) der Punkte müssen ein bestimmtes Format haben, dessen genaue Struktur für unsere Zwecke ebenfalls nicht interessiert.

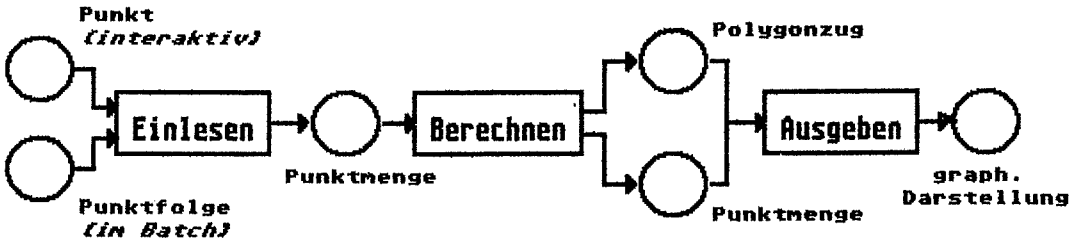
- Die Koordinaten der Punkte sollen dem Programm entweder *interaktiv* (d.h. im Dialog) oder als Inhalt einer Datei (im *Batch*, wie man auch sagt) übergeben werden können. Das Format dieser Datei ist zwar anfänglich fixiert, doch ist nicht damit zu rechnen, daß es für alle Zeiten festgeschrieben ist.
- Die Darstellung der konvexen Hülle soll, wie in der Skizze angedeutet, durch einen geschlossenen Polygonzug erfolgen, dessen Ecken eine Untermenge der jeweiligen Punktmenge bilden. Auch die innerhalb des Polygons liegenden Punkte der Menge sollen im Bild erscheinen.




**Eine Punktmenge ... ... und ihre konvexe Hülle**

(Weitere, für unsere Zwecke nicht so wesentliche Anforderungen könnten zum Beispiel Bezug nehmen auf die Art und Weise, wie Eingaben an der *Benutzungsschnittstelle* vorzunehmen sind, oder darauf, wie die Wahl des Ausgabegerätes zu treffen ist, usw.)

Die Spezifikation des "Systems", welches die gewünschten Leistungen erbringt, ist nicht schwer, sofern wir uns auf die graphische Beschreibung der Prozeß/Kanal-Struktur (des Datenflusses) beschränken (und dabei die in Abschnitt 6.3.3 eingeführten Notationselemente benutzen):



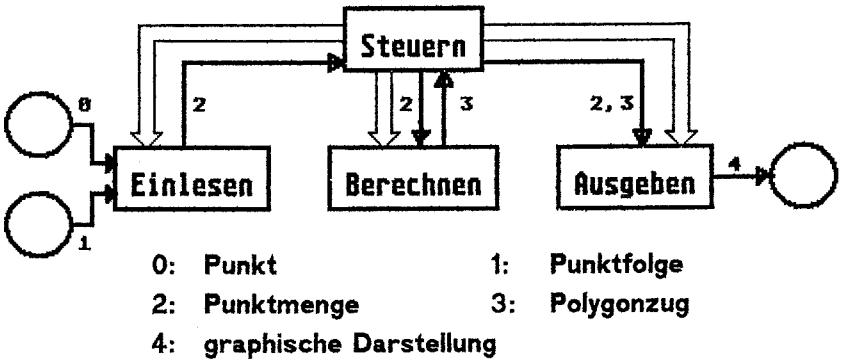

Der Prozeß *Einlesen* erhält seine Inputs entweder als einzelne Punkte über den *interaktiven* Kanal oder als Folge von Punkten, welche in irgendeiner Repräsentation in einer Datei gespeichert sind. Er produziert die Punktmenge, welche dem Prozeß *Berechnen* zugeführt wird. Dieser Prozeß stellt die gleiche Punktmenge sowie die konvexe Hülle in Form eines Polygonzuges dem Prozeß *Ausgeben* zur Verfügung. Letzterer erzeugt aus beidem ein Bild.

Für den Entwurf gehen wir davon aus, daß die Maschine, auf der das Programm zu implementieren sein wird, ein Mono-Prozessor ist, dessen Betriebssystem

uns keine Mechanismen für Multiprocessing anbietet (vgl. die Option (iii) in der Einleitung dieses Kapitels).

Aus unserer früheren Beschäftigung mit der Technik der *schrittweisen Verfeinerung* (Abschnitt 3.2) für die *Programmierung im Kleinen* übernehmen wir die Erkenntnis, daß immer zwei Arten von *Entwurfsentscheidungen* zu treffen sind: *Repräsentationsentscheidungen* und *Zerlegungsentscheidungen*. Die ersten beziehen sich auf die "rechnergerechte" Darstellung von Datenobjekten, während die zweiten Antworten auf die Frage nach der Systemstruktur liefern. Wie weit Repräsentationsentscheidungen beim Architekturentwurf tatsächlich gehen sollten, ist eine Frage, der wir uns weiter unten eingehender widmen werden.

Die erste Zerlegungsentscheidung machen wir uns leicht. Wir orientieren uns dabei an dem zu Beginn des Kapitels skizzierten "Standardaufbau", bei dem die Ausführung der produktiven Prozesse jeweils von Modulen getragen wird, die ihrerseits von einem "Steuermodul" zur Arbeit gerufen und mit Daten versorgt werden. (In traditioneller Terminologie könnte man diesen als das "Hauptprogramm" bezeichnen.) In der folgenden Entwurfszeichnung haben die Pfeile die schon früher erklärten Bedeutungen. Die "Marken" an den Datenfluß-Pfeilen verweisen auf die jeweils übergebenen Daten.




Noch besserer Einsicht ermangelnd und noch ohne Repräsentationsentscheidungen zu treffen, brechen wir diese, strikt am Datenfluß orientierte Zerlegung hier zunächst ab und formulieren eine möglichst knappe Beschreibung der Schnittstellen (also der Außenansichten) der vier Module.

MODUL Steuern

| | |
|---|---|
| INPUTS | Punktmenge (von Einlesen),<br>Polygonzug (von Berechnen) |
| OUTPUTS | Punktmenge (an Berechnen, an Ausgeben),<br>Polygonzug (an Ausgeben) |
| OPERATIONEN | Steuerung der Prozeßkette<br>"Einlesen - Berechnen - Ausgeben", |

| | |
|---|---|
| MODUL Einlesen | |
| INPUTS | Punkt (interaktiv), Punktfolge (von Datei) |
| OUTPUTS | Punktmenge (an Steuern) |
| OPERATIONEN | Produktion einer Punktmenge. |
| MODUL Berechnen | |
| INPUTS | Punktmenge (von Steuern) |
| OUTPUTS | Polygonzug (an Steuern) |
| OPERATIONEN | Berechnung des Polygonzugs, der die konvexe Hülle der Punktmenge begrenzt. |
| MODUL Ausgeben | |
| INPUTS | Punktmenge (von Steuern), Polygonzug (von Steuern) |
| OUTPUTS | graphische Darstellung |
| OPERATIONEN | graphische Darstellung der Punktmenge und des Polygonzugs, der ihre konvexe Hülle begrenzt. |

Inwieweit, so ist zu fragen, hält diese Modularisierung jene fünf Gebote ein, die wir im vorangegangenen Abschnitt aufgestellt haben? Das Gebot der *textuellen Einheit* ist zweifellos erfüllt. Augenscheinlich gibt es *wenige Schnittstellen*, denn direkte Kommunikation besteht nur zwischen dem "Dirigenten" Steuern einerseits und jedem der drei "Produzenten" andererseits. Gibt es andererseits auch indirekte Kommunikation zwischen diesen letzteren? Verborgene Schnittstellen sozusagen? Wie *schmal* sind diese, und wie *explizit* ? Offenbar wird dies von der noch nicht festgelegten Repräsentationen der Daten abhängen.

Nehmen wir nun an, um die Erfüllung des Gebots der *Geheimhaltung* zu prüfen, daß die Module von vier verschiedenen Mitgliedern eines Teams implementiert werden sollen. Beim bisherigen Stand des Entwurfs verfügen diese offenbar nicht über alle für ihre Arbeit benötigten Informationen. Sie haben zwei Möglichkeiten: Entweder sie bitten den Entwerfer, ihnen weitere und genaue Vorgaben hinsichtlich der für Punktmengen und Polygonzüge zu verwendenden Datenstrukturen zu machen, oder sie setzen sich selbst zusammen und verständigen sich über solche Strukturen. In jedem Fall müssen diese für alle Bearbeiter verbindlich sein, denn jeder Modul hat es mit mindestens einem der genannten Datentypen zu tun. Mag also die eigentliche Ausführung der von den vier Modulen geforderten Operationen das *Geheimnis* des jeweiligen Teammitglieds sein, die Repräsentation der Daten ist es nicht. Sie muß bei dieser Version unseres Entwurfs öffentlich sein und als Teil aller Modulschnittstellen gelten.

Aufgrund unseres Bemühens um "methodische Reinheit" zögern wir übrigens, ausgerechnet dem Entwerfer die Aufgabe aufzubürden, die Repräsentation der Daten so detailliert zu definieren, daß den Implementierern in dieser Hinsicht

nichts mehr zu tun bleibt. Denn so wie wir im Zusammenhang mit Algorithmen von Implementierung sprechen, wenn es um deren konkrete Formulierung in einer Programmiersprache geht, so können wir auch das Geschäft der rechnergerechten Repräsentation von Daten als deren Implementierung bezeichnen. Im vorliegenden Fall argumentieren wir daher zugunsten des Programmiererteams, dessen Professionalität Respekt verdient und nicht durch kleinliche Vorschriften abzuwerten ist. Die Crux dieser noblen Haltung besteht nun freilich darin, daß vier Mitarbeiter ihre wertvolle Zeit mit Abstimmung und Koordination verbringen müssen. Die für das arbeitsteilige *Teamwork* notwendige *unabhängige Beschreibbarkeit* und die *unabhängige Verstehbarkeit* sind dahin. Doch noch schlimmer: Es wird sich zeigen, daß unsere bisher definierten Module auch den übrigen wünschenswerten Eigenschaften *Kombinierbarkeit, Stetigkeit* und *Abgeschirmtheit* überhaupt nicht gerecht werden.

Um die Sache konkret zu machen, nehmen wir an, daß man sich auf folgenden Vorschlag einigt, der - in MODULA-2 Notation - als *Definitionsmodul* verpackt wird:

```
DEFINITION MODULE Punktmenge;
CONST   MaxanzC = ...;
TYPE    PunktTyp    =       RECORD
                              x, y: REAL
                            END;
        PunktmengenTyp =    RECORD
                              maechtigkeit: INTEGER;
                              punkte:
                                 ARRAY [1..MaxanzC] OF PunktTyp
                            END;
        PolygonzugTyp  =    PunktmengenTyp;
END Punktmenge.
```

(Wir gehen hier, wie auch schon bisher, davon aus, daß der Leser mit den Möglichkeiten der Sprache MODULA-2 hinreichend vertraut ist - vgl. Vorwort.)

Und ebenfalls in Definitionsmodulen legen sie die Schnittstellen zwischen den jeweiligen Systemteilen fest:

```
DEFINITION MODULE Einlesen;
FROM Punktmenge IMPORT PunktmengenTyp;
PROCEDURE Einlesen(VAR pM: PunktmengenTyp; VAR err: BOOLEAN);
END Einlesen.
```

```
DEFINITION MODULE Berechnen;
FROM Punktmenge IMPORT PunktmengenTyp, PolygonzugTyp;
PROCEDURE Berechnen(VAR pM: PunktmengenTyp;
                    VAR pZ: PolygonzugTyp);
(* Obwohl "Berechnen" Punkte nur liest, muß pM, da von zusammen-
gesetztem Typ, als VAR-Parameter deklariert werden.*)
END Berechnen.

DEFINITION MODULE Ausgeben;
FROM Punktmenge IMPORT PunktmengenTyp, PolygonzugTyp;
PROCEDURE Ausgeben(VAR pM: PunktmengenTyp;
                   VAR pZ: PolygonzugTyp);
(* Kommentar wie oben*)
END Ausgeben.
```

Für "Steuern" wird es offenbar nur einen *Implementierungs-Modul* geben, welcher seinerseits die Prozeduren "Einlesen", "Berechnen" und "Ausgeben" importiert.

Jeder der vier Programmierer muß die getroffenen Typvereinbarungen strengstens beachten, das leuchtet sofort ein. Jeder muß sich an die gewählten Bezeichner halten. Alle Implementierungs-Module, die sie schreiben, müssen auf die verabredete Struktur und die Namen ihrer Komponenten Bezug nehmen.

Nun wäre das womöglich nicht weiter tragisch, wenn man davon ausgehen könnte, daß die gewählte Repräsentation ein für allemal feststeht. Doch leider ist damit im allgemeinen nicht zu rechnen. Man nehme zum Beispiel an, daß sich nach der auf der Basis obiger Punktmengen-Darstellung erfolgten Implementierung herausstellt, daß der Wert der Konstanten "MaxanzC" häufig geändert werden muß, da es das Programm mit immer mächtigeren Punktmengen zu tun bekommt. Abgesehen davon, daß jede Änderung der Konstanten eine Neukompilierung bedingt, ist es im laufenden Betrieb sicher nicht angenehm, immer wieder in Ausnahmesituationen zu geraten, wenn die durch "MaxanzC" gegebene Mächtigkeitsschranke überschritten wird. (Der Modul "Steuern" wird auf die in "err" übergebene Fehlermeldung des Moduls "Einlesen" entsprechend zu reagieren haben!) Unser Programmiererteam entschließt sich daher nach einiger Zeit, eine hinsichtlich dieser Unzulänglichkeiten bessere Repräsentation von Punktmengen einzuführen, eine Repräsentation als lineare "verzeigerte" Liste zum Beispiel, die so lange werden darf, wie beim Lauf des Programms Speicher zur Verfügung steht:

```
DEFINITION MODULE Punktmenge; (* Version 2 *)
TYPE  PunktTyp =        RECORD
                          x,y: REAL
                        END;
```

```
        PMRefTyp =          POINTER TO PunktmengenTyp;
        PunktmengenTyp =    RECORD
                              punkt: PunktTyp;
                              nxtpkt: PMRefTyp
                            END;
        PolygonzugTyp =     PunktmengenTyp;
    END Punktmenge.
```

Auch ohne die Programme, welche auf der alten Repräsentation von Punktmengen beruhen, explizit angegeben zu haben, leuchtet es unmittelbar ein, daß die Entscheidung unseres Teams für eine andere rechnerinterne Punktmengen-Darstellung eine schwerwiegende Konsequenz hat: Sämtliche Module müssen praktisch neu geschrieben werden. Und wieder eine Weile, nachdem man sich dieser Mühe unterzogen hat, mag einem der Programmierer die Idee kommen, daß sich der (für eventuelle spätere Erweiterungen vielleicht notwendige) Zugriff auf einzelne Punkte der Menge wesentlich effizienter gestalten ließe, wenn nur die Punkte der Menge schon in bestimmter Weise sortiert wären. Man entschließt sich also ein zweites Mal zu einer neuen Datenstruktur für Punktmengen, etwa zu einem binärem Baum, in dem die Punkte gemäß ihres Abstandes zum Ursprung des Koordinatensystems sortiert sind; für Polygonzüge dagegen soll die alte lineare Liste beibehalten werden:

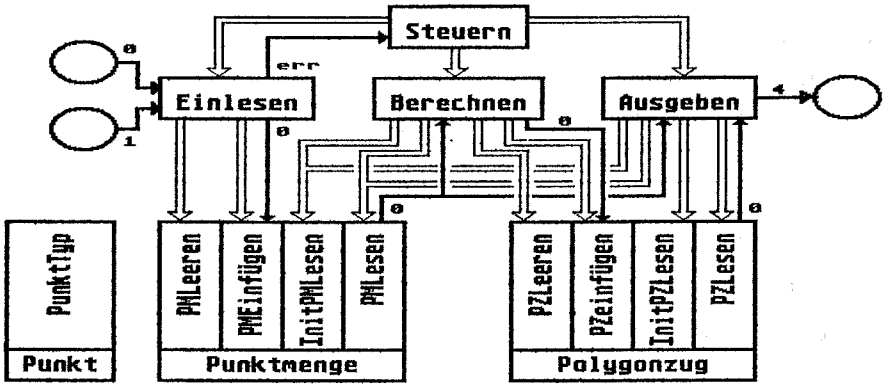
```
    DEFINITION MODULE Punktmenge; (*Version 3 *)
    TYPE  ...
        PunktmengenTyp =    RECORD
                              punkt: PunktTyp;
                              naeher: PMRefTyp;
                              weiter: PMRefTyp
                            END;
        PZRefTyp =          POINTER TO PolygonzugTyp;
        PolygonzugTyp =     RECORD
                              punkt: PunktTyp;
                              nxtpkt: PZRefTyp
                            END;
    END Punktmenge.
```

Und wieder müssen alle Module nicht weniger tiefgreifend geändert werden als beim ersten Mal. Fehler, die der Programmierer eines der Module macht, können ohne weiteres unangenehme Auswirkungen auf die Funktionen der jeweils anderen Module haben. (Wer einmal durch falsch initialisierte oder falsch "umgehängte" Pointer verursachte "Bugs" gesucht hat, weiß davon sicher ein Lied zu singen!)

Es dürfte inzwischen klar geworden sein, daß von den drei Eigenschaften *Kombinierbarkeit*, *Stetigkeit* und *Abgeschirmtheit* nicht eine einzige durch den bisherigen Systementwurf gefördert wird. Die vier Module sind nicht anders kombinierbar als miteinander, kleine Änderungen der Datenstrukturen erzwingen die Änderung sämtlicher Programme, und Fehler bei der Implementierung eines der Module haben Auswirkungen auf alle übrigen.

Bevor sie in eine dritte oder gar vierte Runde dieses Spiels einsteigen, werden sich unsere Freunde daher gewiss überlegen, was sie besser machen können, um dem Übel der wiederholten Neu-Programmierung sämtlicher Module des Systems ein für allemal abzuhelfen. Sie werden bemerken, daß jeder von ihnen (oder, besser gesagt, jeder der von ihnen verfaßten Module) auf den durch die jeweils gegebenen Typ-Deklarationen strukturierten Punktmengen-Speicher in ganz spezifischer Weise zugreift: Der "Einleser" fügt Punkte in die Menge ein (ist also eigentlich ein "Einschreiber"!), und sowohl der "Berechner" als auch der "Ausgeber" entnehmen dem Speicher Punkte zur allfälligen Bearbeitung. Natürlich muß der "Einleser" dafür sorgen, daß der Speicher zu Beginn seiner Arbeit in einem ordentlichen Zustand ist (er sollte leer sein!), und "Berechner" und "Ausgeber" müssen die Möglichkeit haben, vor der Entnahme von Punkten den Speicher in einen Zustand zu versetzen, welcher garantiert, daß die erste Entnahmeaktion auch wirklich den ersten Punkt liefert. Analoge Überlegungen gelten für den Speicher, der die den Polygonzug bestimmenden Punkte enthalten wird. Das ist aber auch alles.

Die Lösung des Problems liegt also auf der Hand: Sowohl die textuelle Beschreibung als auch die Ausführung der genannten Operationen muß von den Modulen, die diese Operationen benutzen, sauber getrennt werden. Die Module "Steuern", "Einlesen", "Berechnen" und "Ausgeben" müssen von den speziellen, für Punktmengen und Polygonzüge jeweils gewählten Repräsentationen unabhängig sein.




(0, 1 und 4 haben die gleiche Bedeutung wie in der Abbildung des ersten Entwurfs; "err" enthält eine Meldung über den (Miß-)Erfolg des Einlesevorgangs.)

Sie dürfen auf diese Repräsentationen nur über geeignete, den jeweiligen Operationen entsprechende Prozeduren zugreifen. Damit ist der Systementwerfer wieder gefordert! Außer den bereits bekannten Modulen muß er zwei weitere definieren, einen für Punktmengen und einen für Polygonzüge, in denen die mit diesen "Objekten" erlaubten Operationen *Repräsentations-unabhängig* zusammengefaßt sind. Die jeweiligen Operanden sind Punkte, und nur deren Darstellung wird - ebenfalls in Form eines Moduls - explizit bekanntgegeben. (Es geht auch ohne *explizite* Angabe, wie wir später noch sehen werden.) Das obige Diagramm zeigt den neuen Entwurf.

Und dies ist die textuelle Version des Entwurfs in Form von (kommentierten) MODULA-2 Definitionsmodulen:

```
DEFINITION MODULE Steuern;
(* Dieser Modul exportiert nichts. Er sorgt für den Ablauf der
spezifizierten Prozesse in der Reihenfolge "Einlesen", "Berechnen",
"Ausgeben". Ein Definitionsteil ist hier nur der Vollständigkeit halber
aufgeführt. *)
END Steuern.

DEFINITION MODULE Einlesen;
PROCEDURE Einlesen(VAR err: BOOLEAN);
(* Die Prozedur stellt eine Punktmenge bereit. Die Punkte dieser Menge
werden entweder interaktiv erfaßt oder von einer Datei gelesen. Die
Entscheidung hierüber wird nach einem entsprechenden Dialog mit dem
(menschlichen) Benutzer des Programms gefällt. Soll von einer Datei
gelesen werden, so wird der Name dieser Datei erfragt. Der Parameter
"err" erhält den Wert TRUE, falls der Einlesevorgang korrekt beendet
wurde; andernfalls enthält "err" den Wert FALSE. *)
END Einlesen.

DEFINITION MODULE Berechnen;
PROCEDURE Berechnen;
(* Die Prozedur berechnet für eine gegebene Punktmenge in der
kartesischen Ebene die konvexe Hülle. Ist die Menge leer, so geschieht
nichts. *)
END Berechnen.

DEFINITION MODULE Ausgeben;
PROCEDURE Ausgeben;
(* Die Prozedur stellt eine Punktmenge in der kartesischen Ebene
sowie deren konvexe Hülle auf einem im Dialog mit dem (menschlichen)
Benutzer spezifizierten Ausgabemedium dar. Ist die Menge leer, so
wird eine entsprechende Meldung ausgegeben. *)
END Ausgeben.
```

```
DEFINITION MODULE Punkt;
(* Dieser Modul macht die Struktur der Repräsentation eines Punktes
in der kartesischen Ebene projektweit bekannt. Sein Implementations-
teil ist leer. *)
TYPE PunktTyp = RECORD
                  x,y: REAL
                END;
END Punkt.

DEFINITION MODULE Punktmenge;
(* Dieser Modul "verkapselt" eine Punktmenge. Er verbirgt die Repräsen-
tation der Punktmenge vor seiner "Außenwelt" und gestattet den Zugriff
auf sie nur über die im folgenden aufgelisteten Operationen. *)
FROM Punkt IMPORT PunktTyp;

PROCEDURE PMLeeren;
(* Die Prozedur erzeugt eine leere Punktmenge. *)

PROCEDURE PMEinfuegen(VAR punkt: PunktTyp);
(* Die Prozedur fügt den in "punkt" übergebenen Punkt in die Punktmenge
ein. *)

PROCEDURE InitPMLesen;
(* Die Prozedur eröffnet den Vorgang des Lesens der Punkte einer
gegebenen Punktmenge (die auch leer sein kann). *)

PROCEDURE  PMLesen(VAR punkt: PunktTyp; VAR eopm: BOOLEAN);
(* Die Prozedur stellt den jeweils nächsten Punkt einer gegebenen
Punktmenge in "punkt" zur Verfügung. Ist kein weiterer Punkt vorhanden,
so erhält "eopm" den Wert TRUE; sonst enthält "eopm" den Wert FALSE.*)

END Punktmenge.

DEFINITION MODULE Polygonzug;
(* Die Prozeduren diese Moduls entsprechen denen des Moduls
"Punktmenge". Auf eine detaillierte Beschreibung wird verzichtet. *)
FROM Punkt IMPORT PunktTyp;

PROCEDURE PZLeeren;

PROCEDURE PZEinfuegen(VAR punkt: PunktTyp);

PROCEDURE InitPZLesen;

PROCEDURE PZLesen(VAR punkt: PunktTyp; eopz: BOOLEAN);

END Polygonzug.
```

Was haben wir gewonnen? Erinnern wir uns wieder an die "fünf Gebote" des guten Modularisierens! Wir müssen zugeben, daß es bei unserem ersten Systementwurf zwischen den Modulen "Einlesen", "Berechnen" und "Ausgeben"

in der Tat eine dicke und implizite Schnittstelle gab. Sie war durch die Punktmenge und den Polygonzug gegeben. Diese Schnittstelle ist in unserer zweiten Modularisierung zwar (notwendigerweise) immer noch vorhanden, aber sie ist *schmal* und *explizit* geworden. Während sie sich im ersten Entwurf als vollständige Beschreibung der jeweils gewählten Datenstrukturen präsentierte (und in den Parameterlisten der Prozeduren "Einlesen", "Berechnen" und "Ausgeben" manifest wurde), erscheint sie im zweiten Entwurf nur noch mittels der Operationen, welche die drei "Prozeß-Module" benötigen, um die geforderten Aktionen und Berechnungen durchzuführen. Die Module "Punktmenge" und "Polygonzug" sind direkte Ergebnisse der Anwendung des *Geheimnisprinzips*. Sie machen die Prozeß-Module von den jeweiligen Datenstrukturen unabhängig. Dies bedeutet: Solange sich an der Semantik der in den Definitionsmodulen "Punktmenge" und "Polygonzug" aufgelisteten Prozeduren nichts ändert, können die für die Punktmenge und den Polygonzug gewählten Datenstrukturen beliebig geändert werden, ohne daß dies irgendwelche Auswirkungen auf die Funktionen der "Prozeß-Module" hätte. Und insbesondere müßten diese ihrerseits nicht modifiziert werden. (Ein möglicher Effekt einer Änderung der Datenstrukturen könnte, wie wir anhand der Beispiele oben zu argumentieren versucht haben, natürlich ein Gewinn oder Verlust an Effizienz sein.) Es ist nun ganz unbestreitbar die Aufgabe des Entwerfers, die Gegenstände ("Objekte"!) des Interesses anzugeben sowie die mit bzw. an diesen Gegenständen auszuführenden Operationen und deren Semantik zu definieren, und es bleibt dem Implementierer überlassen, die jeweils geignete rechnerinterne Repräsentation dieser Gegenstände zu bestimmen. (Damit haben wir im übrigen eine Antwort auf die Frage, wie weit der Systemarchitekt bei Repräsentationsentscheidungen gehen sollte: nicht sehr weit!) Und en passant bemerken wir, daß sich MODULA-2, insofern damit Definitionsmodule formuliert werden, sogar als Entwurfssprache eignet.

Betrachten wir den zweiten Entwurf nun andererseits im Lichte der zu Beginn dieses Abschnitts aufgeworfenen "Kernfrage": An welchen Elementen der Systemspezifikation haben wir diesen zweiten Entwurf orientiert? Natürlich auch an den Prozessen und ihrer sequentiellen Abfolge. Doch ist dies nicht charakteristisch für den zweiten Entwurf, denn genau diese Orientierung liegt auch unserer ersten Version der Systemarchitektur zugrunde. Charakteristisch für die zweite Version ist vielmehr die "Einkapselung" der von den Prozessen manipulierten Datenobjekte in Module, und damit - wie am Ende von Abschnitt 6.4.1 versprochen - die Schaffung einer unter den "Prozeß-Modulen" liegenden "Schicht", welche den Prozessen bestimmte Dienstleistungen erbringt. Eine solche Dienstleistung besteht im vorliegenden Fall aus der Ausführung eines Auftrags zur Manipulation einer Punktmenge bzw. eines Polygonzuges. Übrigens wollen wir die betreffenden Module fortan als "*Daten-Module*" typisieren.

Wie der Leser mit Sicherheit bemerkt hat, stützen wir uns in diesem Buch gelegentlich gern auf intuitive Analogien zu anderen Bereichen des (mehr oder

weniger) alltäglichen Lebens, wenn es darum geht, bestimmte, für das Programmieren uns wichtig scheinende Ideen plausibel zu machen. An dieser Stelle bietet es sich an, die Module "Punktmenge" und "Polygonzug" des zweiten Entwurfs mit Werkstätten zu vergleichen, die auf die Herstellung und Bearbeitung irgendwelcher Produkte spezialisiert sind. Die "Prozeß-Module" sind die Kunden dieser Werkstätten. Ihr Verhalten ist mit dem des (normalen) Besitzers eines Autos vergleichbar, der sein Gefährt dem Kfz-Mechaniker übergibt, da ihm selbst das Wissen über die unter der Motorhaube verborgenen Innereien fehlt. Es besteht eine ganz klare Trennung von Verantwortlichkeiten. Der Besitzer des Autos muß (nach Recht und Gesetz) alles tun, damit sein Wagen verkehrstüchtig bleibt; er delegiert die dazu notwendigen Arbeiten an Motor, Bremse und Lenkung an seine Werkstatt. Diese ist für die sachgerechte Ausführung des Auftrags zuständig und kann im Fall undokumentierter mangelhafter Erledigung haftbar gemacht werden.

Ähnlich wie das Verhältnis zwischen dem Autofahrer und seiner Werkstatt implizit oder explizit durch einen Vertrag geregelt ist, muß auch der Entwerfer eines (Daten-)Moduls seine "Allgemeinen und speziellen Geschäftsbedingungen" bekanntgeben. Er tut dies mit dem Text der Definition seines Moduls. Hier muß ganz klar festgelegt sein, was der "Kunde" erwarten kann und was nicht. Natürlich kann er dem Kunden nicht vorschreiben, wie dieser sich nach Erhalt der gewünschten Dienstleistung verhalten soll. So ist insbesondere der Anforderer eines Dienstes für seine Reaktion auf einen ihm signalisierten Fehler bei der Dienstausführung ganz allein verantwortlich. (Auch der Autofahrer muß selbst entscheiden, ob er die Weiterfahrt wagt, nachdem ihm sein Mechaniker mitgeteilt hat, daß sich die Bremse beim besten Willen nicht mehr reparieren läßt.)

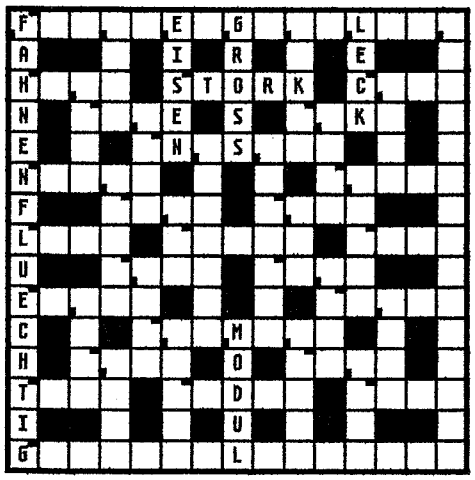
Im folgenden Abschnitt werden wir eine Möglichkeit der exakten Formulierung von (Daten-)Modul-"Nutzungsverträgen" kennenlernen. Zuvor jedoch sollten wir noch ein wenig bei der "Feinstruktur" eines für den Umgang mit Daten von bestimmtem Typ konzipierten Moduls verweilen. Auch wird die Frage nach verschiedenen Modul-Arten und dem, was nützlicherweise in einem Modul verborgen werden sollte, unsere Aufmerksamkeit verlangen.

Bei der Definition der Daten-Module "Punktmenge" und "Polygonzug" haben wir uns sehr eng an den unmittelbaren Bedürfnissen der Prozeß-Module orientiert. Wir haben nur diejenigen Operationen "verpackt", die wirklich benötigt wurden. Natürlich hätten wir etwas großzügiger sein können, etwa in Hinblick auf zukünftig für Punktmengen zu erledigende Aufgaben, oder auch um den Implementierern der Prozeß-Module mehr Komfort zu bieten. Für den Programmierer von "Berechnen" wäre es zum Beispiel von Vorteil, wenn er mit einer als "PROCEDURE PMleer(): BOOLEAN" gegebenen Operation schon vor Beginn der eigentlichen Arbeit prüfen könnte, ob die Punktmenge leer ist oder nicht. Sein Programmtext würde damit sicher an Verständlichkeit gewinnen. Weiter kann man

sich vorstellen, daß es irgendwann einmal nützlich ist zu wissen, wieviele Punkte die Menge überhaupt enthält ("PROCEDURE PMCard():CARDINAL"). Vielleicht benötigt man später auch die Möglichkeit, den Punkt mit dem größten (oder kleinsten) Abstand zum Koordinatenursprung herauszufinden ("PROCEDURE PMMaxPunkt(VAR punkt: PunktTyp; VAR err: BOOLEAN)"): Es wäre dem "Kunden" eines Punktmengen-Modul wohl kaum zuzumuten, diese Operation gewissermaßen bei sich durchzuführen. Und schließlich sollte unser Punktmengen-Handwerker auch in der Lage sein, einen ihm angezeigten Punkt aus der Menge zu entfernen ("PROCEDURE PMLoeschePunkt(VAR punkt: PunktTyp; VAR err: BOOLEAN)").

Wir sehen: So eine auf die Arbeit mit bestimmten Datenobjekten spezialisierte Werkstatt kann uns Dienstleistungen verschiedenster Arten anbieten. Und es liegt nun auf der Hand, diese Dienstleistungsarten etwas detaillierter zu kategorisieren. Zuvor jedoch wollen wir uns noch etwas mehr Anschauungsmaterial verschaffen. Es stammt aus einem kleinen Programmsystem (mit dem Acronym K.U.S.S. = Kreuzworträtsel-UnterStützungs-System), welches im Rahmen eines Praktikum-Projektes entstand.

K.U.S.S. soll beim Erstellen und Lösen jener Kreuzworträtsel behilflich sein, die allwöchentlich im Freitagsmagazin einer deutschen Tageszeitung (früherer Werbeslogan: "Dahinter steckt immer ein kluger Kopf.") publik gemacht werden. (Für ein Beispiel siehe unten.) Es würde zu weit führen, hier eine komplette Spezifikation als Ausgangspunkt des Entwurfs darzulegen. Es möge genügen, auf einige der Dienstleistungen des Systems als ganzem hinzuweisen.




Bei der Erstellung eines Rätsels zum Beispiel müssen Felder, in die keine Buchstaben geschrieben werden sollen, blockiert werden. (Dies kann, wie im Bild, durch Schwärzung geschehen. Bei den zur Debatte stehenden Rätseln erfolgt diese Blockierung übrigens immer in schöner Symmetrie!) In bestimmten Feldern müssen Marken angebracht werden, welche dann anzeigen, ob ein Wort in senkrechter und/oder waagrechter Richtung einzutragen ist (siehe die Marken in der rechten oberen (für "Waagrecht") bzw. linken unteren (für "Senkrecht") Ecke eines Feldes). (Die Entscheidung für diese Art der Identifizierung von Wortanfängen im Rätsel wurde durch ein technisches Merkmal der Benutzungsschnittstelle beeinflußt, nämlich durch das Vorhandensein einer "Maus" als "Zeigeinstrument".) Man überlegt sich leicht, daß die Markierung eines "Blockmusters" im allgemeinen einem einfachen Schema folgt, sodaß eine

Automatisierung dieser Operation wünschenswert ist. (Korrekturmöglichkeiten muß es freilich geben.) Auch sollten, zur eventuellen Produktion einer gedruckten Version des Rätsels, die markierten Felder numeriert werden können. Und schließlich müssen die Rätselfragen formuliert und den markierten Feldern zugeordnet werden.

Dem über der Lösung des Rätsels grübelnden Zeitgenossen soll der Rechner auf Anforderung mindestens die zu einer Marke gehörige Frage anzeigen und den Eintrag des gesuchten Wortes ermöglichen. (Zum Beispiel kann die Anforderung durch Führen des Mauszeigers in die Nähe der Marke und anschließendes Betätigen einer Maustaste erfolgen.) Doch wir leben im Zeitalter des leichten Zugriffs auf Informationen aller Art. Warum sollten wir dem Rater daher nicht auch ein elektronisches Rätselwörterbuch gönnen, von dem er sich bei Bedarf Lösungsvorschläge machen lassen kann, die als Eintrag zu übernehmen ihm freigestellt sein muß. Man kann sich mannigfache Hilfen dieser Art vorstellen, und wir wollen keine von vornherein ausschließen.

Wie auch immer: in jedem System, welches die skizzierte Unterstützung in der einen oder anderen Form bietet, muß es ein Objekt geben, in dem sich vertikal und horizontal angeordnete Felder unterscheiden lassen; Felder, denen man Buchstaben, Marken und (Frage-)Texte zuordnen und die man gegen das Beschreiben auch schützen kann. Es ist - wie die Punktmenge in unserem ersten Beispiel - ein Objekt, welches durch Operationen definiert ist, also durch das, was man mit ihm tun oder über es erfahren kann. Wie dieses Objekt letztlich aussieht oder rechnerintern dargestellt wird, ist für die Bedeutung der beschriebenen Manipulationen unerheblich. Und wie im Falle der Punktmenge fassen wir diese Operationen in einem MODULA-2 Definitionsmodul zusammen (auch hier freilich ohne den sowieso unerfüllbaren Anspruch auf Vollständigkeit zu erheben).

```
DEFINITION MODULE Kreuzwort;
(* Definition des Objektes "Kreuzworträtsel" durch die für dieses Objekt
   spezifischen Operationen. Einzelne Felder des Kwr werden durch Zeilen-
   und Spaltennummer identifiziert: (z,sp) bezeichnet das Feld in Zeile z und
   Spalte sp. *)

PROCEDURE OeffneKWR;
(* Erzeugt den Rahmen für die Erstellung eines Kreuzworträtsels und trifft
   alle zur visuellen Darstellung des Kwr notwendigen vorbereitenden Maß-
   nahmen. Diese Operation ist einmalig vor Ausführung der übrigen
   Operationen dieses Moduls auszuführen. *)

PROCEDURE SetzeKWRzurück;
(* Macht alle bisher mit dem Kreuzworträtsel vorgenommenen Operationen
   - auch visuell - rückgängig. *)
```

```
PROCEDURE IstBlockiert(z,sp:INTEGER):BOOLEAN;
(* Liefert TRUE, falls das Feld (z,sp) blockiert ist; FALSE sonst. *)
PROCEDURE IstMarkiert(z,sp:INTEGER):BOOLEAN;
(* Liefert TRUE, falls das Feld (z,sp) markiert ist; FALSE sonst. *)
PROCEDURE HatWaagMarke(z,sp:INTEGER):BOOLEAN;
(* Liefert TRUE, falls das Feld (z,sp) mit einer "Waagrecht"-Marke
versehen ist; FALSE sonst. *)
PROCEDURE HatSenkMarke(z,sp:INTEGER):BOOLEAN;
(* Liefert TRUE, falls das Feld (z,sp) mit einer "Senkrecht"-Marke
versehen ist; FALSE sonst. *)
PROCEDURE Nummer(z,sp:INTEGER):CARDINAL;
(* Falls das Feld (z,sp) markiert ist und "IstAktuellNumerierung" (s.u.) den
Wert TRUE ergibt, so liefert "Nummer" die laufende Nummer des Feldes
(s.u. "AutoNumeriere"); andernfalls liefert "Nummer" den Wert 0. *)
PROCEDURE Inhalt(z,sp:INTEGER):CHAR;
(* Falls das Feld (z,sp) nicht blockiert ist, so liefert "Inhalt" den in dieses
Feld eingetragenen Buchstaben oder - falls kein Buchstabe eingetragen
ist - das Leerzeichen; ist das Feld blockiert oder wird durch (z,sp) kein
Feld des Kwr bezeichnet, so liefert "Inhalt" das Zeichen mit der Ordinal-
zahl 255 (377C). *)
PROCEDURE IstAktuellNumerierung():BOOLEAN;
(* Liefert TRUE, falls seit der letzten "AutoNumeriere"-Operation - s.u. -
keine die Numerierung verändernde Operation vorgenommen wurde;
FALSE sonst. *)
PROCEDURE SetzeBlock(z,sp:INTEGER);
(* Falls (z,sp) ein gültiges Feld des Kwr bezeichnet, das nicht markiert ist,
wird dieses Feld gegen Beschreiben blockiert. Die Blockierung wird in
geeigneter Weise sichtbar gemacht. Falls (z,sp) kein gültiges Feld des Kwr
bezeichnet, oder ein markiertes Feld, so geschieht nichts. Dies ist eine die
Numerierung verändernde Operation. *)
PROCEDURE SetzeWaagMarke(z,sp:INTEGER);
(* Falls (z,sp) ein gültiges Feld des Kwr bezeichnet, das nicht blockiert ist,
wird dieses Feld mit einer "Waagrecht"-Marke versehen. Die Markierung
wird in geeigneter Weise sichtbar gemacht. Falls (z,sp) kein gültiges Feld
des Kwr bezeichnet, oder ein blockiertes Feld, so geschieht nichts. Dies
ist eine die Numerierung verändernde Operation. *)
PROCEDURE SetzeSenkMarke(z,sp:INTEGER);
(* Wie "SetzeWaagMarke" mit "Senkrecht" anstatt "Waagrecht". *)
```

```
PROCEDURE UmschalteBlock(z,sp:INTEGER);
(* Falls (z,sp) ein gültiges Feld des Kwr bezeichnet, das nicht markiert ist,
so wird das Feld blockiert, falls es nicht blockiert ist, und ent-blockiert,
falls es blockiert ist. Falls (z,sp) kein gültiges Feld des Kwr bezeichnet,
oder ein markiertes Feld, so geschieht nichts. Dies ist eine die Numerie-
rung verändernde Operation. *)

PROCEDURE UmschalteSymmBloecke(z,sp:INTEGER);
(* Anwendung der "UmschalteBlock"-Operation auf das Feld (z,sp) sowie
auf die zu (z,sp) bezüglich der horizontalen und der vertikalen Achse sowie
des Zentrums des Kwr symmetrisch gelegenen Felder. Dies ist eine die
Numerierung verändernde Operation. *)

PROCEDURE UmschalteWaagMarke(z,sp:INTEGER);
(* Falls (z,sp) ein gültiges Feld des Kwr bezeichnet, das nicht blockiert ist,
so wird das Feld "Waagrecht" markiert, falls es eine solche Marke nicht
hat; die "Waagrecht"-Marke wird entfernt, falls sie vorhanden ist.
Falls (z,sp) kein gültiges Feld des Kwr bezeichnet, oder ein blockiertes
Feld, so geschieht nichts. Dies ist eine die Numerierung verändernde
Operation. *)

PROCEDURE UmschalteSenkMarke(z,sp:INTEGER);
(* Wie "UmschalteWaagMarke" mit "Senkrecht" anstatt "Waagrecht". *)

PROCEDURE AutoMarkiere;
(* Setzt Marken - "Waagrecht" und/oder "Senkrecht" - in geeignete Felder
des Kwr. Auf die Beschreibung der Eignungskriterien wird an dieser Stelle
verzichtet. *)

PROCEDURE AutoNumeriere;
(* Numeriert die markierten Felder im zeilenweisen Duchlauf. *)

PROCEDURE SchreibeWaagText(z,sp:INTEGER;
                           VAR text:ARRAY OF CHAR);
(* Falls (z,sp) ein gültiges und "Waagrecht" markiertes Feld bezeichnet, so
wird ihm der Text in "text" als "Waagrecht"-Text zugeordnet; andernfalls
geschieht nichts. *)

PROCEDURE SchreibeSenkText(z,sp:INTEGER;
                           VAR text:ARRAY OF CHAR);
(* Wie "SchreibeWaagText" mit "Senkrecht" anstatt "Waagrecht". *)

PROCEDURE HoleWaagText(z,sp:INTEGER; VAR text:ARRAY OF CHAR;
                       VAR err: BOOLEAN);
(* Falls (z,sp) ein gültiges Feld mit "Waagrecht" Marke ist, wird der (z,sp)
zugeordnete "Waagrecht"-Text in "text" übergeben und "err" erhält den
Wert FALSE; andernfalls erhält "err" den Wert TRUE. *)
```

```
PROCEDURE HoleSenkText(z,sp:INTEGER; VAR text:ARRAY OF CHAR;
                                    VAR err: BOOLEAN);
(* Wie "HoleWaagText" mit "Senkrecht" anstatt "Waagrecht". *)
PROCEDURE LoescheWaagText(z,sp:INTEGER);
(* Falls (z,sp) ein gültiges Feld mit zugeordnetem "Waagrecht"-Text
bezeichnet, so wird diese Zuordnung aufgehoben; andernfalls geschieht
nichts. *)
PROCEDURE LoescheSenkText(z,sp:INTEGER);
(* Wie "LoescheWaagText" mit "Senkrecht" anstatt "Waagrecht". *)
PROCEDURE SetzeBuchst(z,sp:INTEGER; c:CHAR);
(* Falls (z,sp) ein gültiges und nicht blockiertes Feld bezeichnet, wird
diesem Feld der Buchstabe "c" eingeschrieben aber nicht sichtbar gemacht;
andernfalls geschieht nichts. *)
PROCEDURE SetzeUndZeigeBuchst(z,sp:INTEGER; c:CHAR);
(* Wie "SetzeBuchst"; der Buchstabe wird jedoch auch sichtbar gemacht.*)
PROCEDURE ZeigeBuchst(z,sp:INTEGER);
(* Falls (z,sp) ein gültiges und nicht blockiertes Feld mit Beschriftung
bezeichnet, wird der in dieses Feld geschriebene Buchstabe sichtbar
gemacht; andernfalls geschieht nichts. *)
PROCEDURE ZeigeAlleBuchst;
(* Die Beschriftung aller gültigen und nicht blockierten Felder des Kwr
wird sichtbar gemacht. *)
PROCEDURE HoleWaagRes(z,sp:INTEGER; VAR res:ARRAY OF CHAR
                                    VAR err:BOOLEAN);
(* Falls (z,sp) ein gültiges Feld mit "Waagrecht" Marke ist, wird das zu
(z,sp) gehörige "Waagrecht" Lösungswort - eventuell mit Leerzeichen - in
"res" übergeben, und "err" erhält den Wert FALSE; andernfalls erhält "err"
den Wert TRUE. *)
PROCEDURE HoleSenkRes(z,sp:INTEGER; VAR res:ARRAY OF CHAR
                                    VAR err:BOOLEAN);
(* Wie "HoleWaagRes" mit "Senkrecht" anstatt "Waagrecht". *)
PROCEDURE ZeigeWaagRes(z,sp:INTEGER; VAR err:BOOLEAN);
(* Falls (z,sp) ein gültiges und "Waagrecht" markiertes Feld bezeichnet, so
wird das zu (z,sp) gehörige Lösungswort - eventuell mit Leerzeichen - im
Kwr sichtbar gemacht, und "err" erhält den Wert FALSE; andernfalls erhält
"err" den Wert TRUE. *)
PROCEDURE ZeigeSenkRes(z,sp:INTEGER; VAR err:BOOLEAN);
(* Wie "ZeigeWaagRes" mit "Senkrecht" anstatt "Waagrecht". *)
END Kreuzwort.
```

Immerhin hat sich eine recht stattliche Liste von Dienstleistungen an dem Objekt "Kreuzworträtsel" ergeben, von denen ganz offensichtlich nicht alle den gleichen Charakter haben. Einige dieser Dienste sind reine Informationsdienste, während die Ausführung anderer den Zustand des Objekts verändert (z.B. "SetzeBlock", "SchreibeWaagText"). Unter den Informationsdiensten lassen sich solche ausmachen, die Informationen über das Objekt liefern (z.B. "IstBlockiert", "IstAktuellNumerierung"), sowie solche, die Information aus dem Objekt herausziehen ("extrahieren") (z.B. "Inhalt", "HoleSenkText"). Und unter den letzteren wiederum gibt es einige, welche die visuelle Darstellung von Information bewirken (z.B. "SetzeUndZeigeBuchst", "ZeigeWaagRes"). Unter den zustandsverändernden Diensten gibt es insbesondere einen ("OeffneKWR"), der das Objekt in einen definierten Anfangszustand versetzt. Er nimmt eine Sonderstellung insofern ein, als er ausgeführt werden muß, bevor irgendein anderer der Dienste in Anspruch genommen werden kann.

Diesen Beobachtungen gemäß wollen wir die Operationen, welche den von einem *Daten-Modul* angebotenen Diensten entsprechen, in vier Kategorien einteilen:

*Initialisierung* - Versetzen eines Datenobjekts in einen "Anfangszustand",
*Transformation* - Veränderung des Zustands eines Datenobjekts,
*Information* - Liefern von Angaben über den Zustand eines Datenobjekts,
*Extraktion* - verfügbar machen von Komponenten eines Datenobjekts.

Für jede der diesen Kategorien zugehörigen Operationen kann (muß aber natürlich nicht) ein an der "Mensch-Maschine"-Schnittstelle des Systems wahrnehmbarer Effekt definiert sein.

Die Operationen der Module "Kreuzwort" und "Punktmenge" (und dieser mit den oben vorgeschlagenen Erweiterungen) sind den genannten Kategorien wie folgt zuzuordnen:

*Initialisierung:*
Kreuzwort: OeffneKWR
Punktmenge: PMleeren, InitPMLesen

*Transformation:*
Kreuzwort: SetzeKWRzurueck,
SetzeBlock, SetzeWaagMarke, SetzeSenkMarke,
UmschalteBlock, UmschalteSymmBloecke,
UmschalteWaagMarke, UmschalteSenkMarke,
AutoMarkiere, AutoNumeriere, SchreibeWaagText,
SchreibeSenkText, LoescheWaagText, LoescheSenkText,
SetzeBuchst, SetzeUndZeigeBuchst
Punktmenge: PMEinfuegen, PMLoeschen

*Information:*

Kreuzwort: IstBlockiert, IstMarkiert, HatWaagMarke, HatSenkMarke, IstAktuellNumerierung
Punktmenge: PMleer, PMCard

*Extraktion:*

Kreuzwort: Nummer, Inhalt, HoleWaagText, HoleSenkText, ZeigeBuchst, ZeigeAlleBuchst, HoleWaagRes, HoleSenkRes, ZeigeWaagRes, ZeigeSenkRes
Punktmenge: PMLesen, PMMaxPunkt

Warum, so mag sich der Leser fragen, haben wir im Modul "Kreuzwort" für offensichtlich ganz ähnliche Operationen verschiedene Prozeduren vorgesehen? Konkret: Die Operationen des Extrahierens der einem Feld zugeordneten Waagrecht- und Senkrecht-Erklärungstexte zum Beispiel werden durch die beiden Prozeduren "HoleWaagText" und "HoleSenkText" realisiert. Warum nicht durch eine einzige Prozedur "HoleText" mit einem zusätzlichen Parameter zur Unterscheidung von Senkrecht und Waagrecht? Selbstverständlich hätten wir dies tun können, und sogar durchaus unter Ausnutzung der der Sprache MODULA-2 eigenen Selbstdokumentations-Fähigkeiten (vgl. 2.2.1): Wir hätten unserem Definitionsmodul nur einen Enumerationstyp "RichtungsTyp = (waagrecht, senkrecht)" hinzuzufügen brauchen und "HoleText" mit einem zusätzlichen Parameter von diesem Typ versehen müssen. Immerhin hätten wir auf diese Weise die Gesamtzahl der im Definitionsmodul aufgelisteten Prozeduren um acht reduzieren können.

Wir müssen einräumen, daß sich Fragen dieser Art nicht aufgrund eindeutiger oder gar formaler Regeln beantworten lassen. Die Antwort ist letztlich eine Sache des Fingerspitzengefühls oder gar des persönlichen Geschmacks. Wenn wir das im vorangegangen Absatz vorgebrachte Argument auf die Spitze trieben, so würden wir bei der MODULA-2 Beschreibung unseres Moduls eigentlich mit einer einzigen Prozedur auskommen, deren Parameterliste dann nicht nur all jene Deklarationen enthalten müßte, welche für die an dem Objekt auszuführenden Operationen notwendig sind, sondern noch einen weiteren Parameter (etwa eine Nummer) zur Identifikation der gewünschten Operation. Doch soweit wird der Entwerfer eines Moduls gewiß nicht gehen wollen. Er wird versuchen, eine Mitte zu finden zwischen Kompaktheit und Verständlichkeit der Beschreibung. (So haben wir in unserem Modul "Kreuzwort" versucht, soviel Bedeutung in die Prozedurnamen zu legen wie möglich, um so auf die Definition eines weiteren Typs verzichten zu können.)

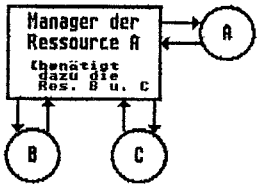
Sorge um die Implementierung braucht den Entwerfer bei diesem Balanceakt eigentlich nicht anzufechten: Dem Implementierer ist es schließlich freigestellt, ob er bei der Programmierung verschiedener, aber ähnliche Operationen realisierender Prozeduren, "streckenweise" den gleichen Code benutzt oder nicht.

Zur Illustration dieser Behauptung möge der folgende Ausschnitt aus einer möglichen Version des Implementierungsmoduls "Kreuzwort" dienen:

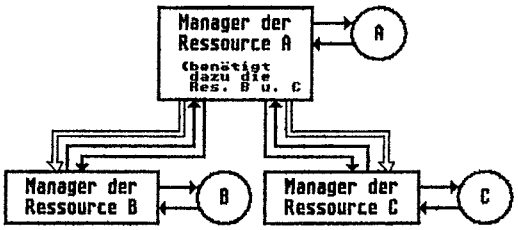
```
IMPLEMENTATION MODULE Kreuzwort;
FROM ... IMPORT ... ;
FROM ... IMPORT ... ;
...
TYPE KreuzwortRätselType = ... ;
VAR kwr: KreuzwortRätselType;

PROCEDURE KoordOK(z,sp:INTEGER):BOOLEAN;
BEGIN
  (* ... *)
END KoordOK;

PROCEDURE HoleText(z,sp:INTEGER; VAR text:ARRAY OF CHAR);
BEGIN
  (* ... *)
END HoleText;
...
...
PROCEDURE HoleWaagText(z,sp:INTEGER; VAR text:ARRAY OF CHAR;
                                     VAR err:BOOLEAN);
BEGIN
  err:= NOT(KoordOK(z,sp)) OR NOT(HatWaagMark(z,sp));
  IF NOT err THEN
    (* ... spezifisch für Waagrecht *)
    HoleText(z,sp,text);   (* gemeinsam für Waagrecht und Senkrecht *)
  END (*IF*)
END HoleWaagText;

PROCEDURE HoleSenkText(z,sp:INTEGER; VAR text:ARRAY OF CHAR;
                                     VAR err:BOOLEAN);
BEGIN
  err:= NOT(KoordOK(z,sp)) OR NOT(HatSenkMark(z,sp));
  IF NOT err THEN
    (* ... spezifisch für Senkrecht *)
    HoleText(z,sp,text);   (* gemeinsam für Waagrecht und Senkrecht *)
  END (*IF*)
END HoleSenkText;
...
END Kreuzwort.
```

Beiden Modulen, "Kreuzwort" und "Punktmenge", ist gemein, daß sie jeweils genau ein Datenobjekt verwalten. Natürlich ist ein solches Datenobjekt in einem Speicherbereich untergebracht (dessen Größe sich im Laufe der Arbeit mit dem Objekt unter Umständen ändert, weil das Objekt wachsen oder schrumpfen kann). Insofern stellt es eine *Ressource* dar, welche durch die Einkapselung im Modul vor unbefugtem Zugriff geschützt wird, da die Struktur dieses Speicherbereichs nur den Prozeduren des Moduls selbst bekannt ist.

Diese Beobachtung läßt uns eine weitere, allgemeinere Aufgabe von Modulen verstehen, eine Aufgabe, die mit der vor dem Ende von Abschnitt 6.4.1 eingefügten Abbildung schon angedeutet wurde: Eben die des Ressourcen-"Managers". So wird sich in einem Multi-Processing Betriebsystem (vgl. die Einleitung zu 6.4) eines Rechners ein Modul finden, der sich um die Zuteilung von Prozessorzeit an rechenbereite Prozesse kümmert, also diesbezügliche Operationen anbietet, und einen, von dem durch diese Prozesse der jeweils benötigte Speicher angefordert werden kann. Auf höherer Ebene wird es einen Datei-Manager geben, welcher die externen Speicher verwaltet, deren Struktur er kennt (weil er - bzw. sein Implementierer - diese ja selbst definiert hat) und in ihren Einzelheiten vor den die Dateien benutzenden Prozessen verbirgt. Letztere sehen Dateien als Objekte, mit denen Operationen wie "Kreieren", "Öffnen", "Lesen" (einer Anzahl von Bytes), "Schreiben" (einer Anzahl von Bytes), "Zurücksetzen", "Umbenennen", "Löschen", und so weiter, möglich sind; mit den zur Realisierung dieser Operationen erforderlichen Datenstrukturen haben sie dagegen nichts zu tun. Selbstverständlich benötigen die Ressourcen-Manager zur Bewältigung ihrer Aufgaben in der Regel selbst wieder irgendwelche Ressourcen. Sollten sie deren Verwaltung und Manipulation selbst übernehmen oder dies ihrerseits speziell dafür "ausgebildeten" Modulen überlassen? Im allgemeinen (d. h., falls nicht dringende Effizienzgründe dagegen sprechen) wird die Antwort wohl zugunsten einer weiteren "Schichtung" ausfallen:




Nicht so . . .



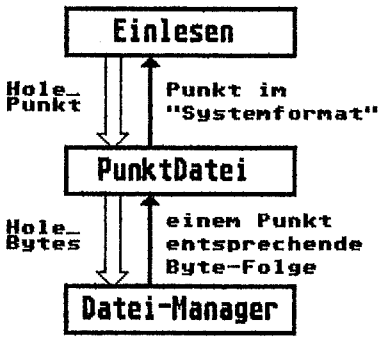

. . . sondern so

Der Grund für diese Antwort ist klar: Wir brauchen ja nur wieder einen Extremfall zu betrachten, den nämlich, daß auch bei der Verwaltung von B und C weitere Ressourcen in's Spiel kommen, und so fort. Der Manager der Ressource A würde "verrückt" (oder, in Software-Terminologie: unwartbar), wenn er sich um alles höchstpersönlich kümmern müßte! (Der Manager der Ressource "Kreuzworträtsel" wird sich übrigens des Managers einer Ressource "Bildschirm" be-

dienen, um all die visuellen Effekte zu erzielen, die bei manchen seiner Operationen zu bewirken sind. Eine der Import-Deklarationen im oben skizzierten Implementationsmodul muß sich auf einen entsprechenden Modul beziehen.)

Betrachten wir nun noch einmal, um unser Verständnis der Schichtung von Software-Systemen zu vertiefen, den oben erwähnten Datei-Manager. Es wäre sicherlich zu viel verlangt, wollten wir ihm auch die Kenntnis des Inhalts der von ihm verwalteten Dateien aufbürden. Für die Interpretation der etwa nach einer Leseoperation vom Datei-Manager an den anfordernden Prozeß übergebenen Bytefolge muß dieser Prozeß vielmehr selbst verantwortlich sein. Er kann natürlich auch diese Aufgabe an einen eigens dafür konzipierten Modul delegieren und sollte es sogar, wie wir uns am Beispiel unseres Systems "Konvexe Hülle" klarmachen wollen.

Man erinnere sich: Es hieß, daß das Format der vom "Einleser" aus einer Datei zu lesenden Punkt-Daten zwar anfänglich fixiert ist, daß aber nicht damit zu rechnen ist, daß dieses Format für alle Zeiten unverändert bleibt. Würde es geändert, so müßte auch der Implementationsmodul des "Einlesers" entsprechend modifiziert werden. Da nun die wesentliche Aufgabe dieses Prozeß-Moduls darin besteht, für den Aufbau einer Punktmenge zu sorgen (insofern bezieht er sich auf eine ganz konkrete Entwurfsentscheidung!), und nicht darin, mit möglicherweise wechselnden Datenformaten zu jonglieren, empfiehlt sich nun in der Tat die Einschaltung eines Moduls "PunktDatei", dessen wesentliche Dienstleistung darin besteht, auf Anforderung den jeweils nächsten Punkt in der durch den Modul "Punkt" systemweit vereinbarten Darstellung anzuliefern. Der Modul "PunktDatei" weiß, wieviele Bytes in einer Datei einem Punkt entsprechen (soviele wird er sich dann jeweils vom Datei-Manager geben lassen!), und was mit diesen Bytes zu tun ist, um einen Punkt im "Systemformat" herzustellen. Offenbar ist dies eine Operation, welche sich keiner der vier oben für Datenobjekte unterschiedenen Kategorien zuordnen läßt. Hier wird gewissermaßen etwas *produziert* (ein der Initialisierung von Datenobjekten durchaus ähnlicher Vorgang), doch geschieht dies nicht aus dem Nichts: Tatsächlich wird eine Darstellung eines Objekts in eine andere Darstellung umgewandelt, *konvertiert*. Wir haben es hier also mit einer *Konversions*-Operation zu tun, mit einer Operation, die für Module der Art "PunktDatei", welche spezielle Formate vor den übrigen Teilen eines Systems verbergen, typisch ist.




Mutatis mutandis lassen sich übrigens auch für die Ausgabeseite des Systems "Konvexe Hülle" Überlegungen anstellen, die auf die Einfügung eines die Charakteristika von Ausgabegeräten verbergenden Moduls gerichtet sind. Solche Module werden auch als *(Geräte-) Treiber* bezeichnet.

Bevor wir uns einer wesentlichen Verallgemeinerung der bisher betrachteten, auf den Umgang mit einem Datenobjekt (bzw. Ressource) beschränkten Module zuwenden, erscheint es angebracht, ein mögliches, in der einschlägigen Literatur tradiertes Mißverständnis aufzuklären: In seinem schon oben erwähnten Aufsatz ([PA2]) nennt Parnas als wichtigstes Kriterium für die Zerlegung eines Systems in Module die Tatsache, daß jeder Modul eine *Entwurfsentscheidung* verbergen müsse. Im Lichte unserer bisherigen Diskussion ist es freilich notwendig, diese Aussage etwas zu differenzieren. Wir haben erkannt: Ein Modul besteht immer aus zwei Teilen, seiner *Definition* und seiner *Implementierung*. (Programmiersprachen wie MODULA-2 haben diese Zweiteilung nicht selbst gesetzt, sondern nur aufgegriffen und praktisch nutzbar gemacht.) Die Definition eines Moduls verbirgt nichts, im Gegenteil, sie macht alles, was der Modul kann, öffentlich. Die Definitionsteile der Module eines Systems bilden in ihrer Gesamtheit den Entwurf der Systemarchitektur. Als formale Dokumente fassen sie zusammen, was mit den für das System als relevant erkannten Objekten zu tun ist. In den Implementierungsteilen der Module dagegen wird das Wie, werden Repräsentationsentscheidungen sowie Entscheidungen hinsichtlich des prozeduralen Entwurfs (der "Programmierung im Kleinen") tatsächlich verborgen, vor der Außenwelt "geheimgehalten". Nur in dieser Bedeutung ist daher der Begriff "Entwurf" in der Parnas'schen Formulierung zu verstehen.

Doch nun zu der angekündigten Verallgemeinerung. Sie wird durch die Werkstattanalogie nahegelegt. Eine Autowerkstatt zum Beispiel ist im allgemeinen nicht für einen einzigen Wagen da. Vielmehr sind die Werkzeuge sowie das Wissen und die Fähigkeiten der Mechaniker auf alle Exemplare (mindestens) einer Automarke anwendbar. Warum also sollten wir, so ist zu fragen, dem Modul, der die für das System "Konvexe Hülle" grundlegende Punktmenge bedient, nicht auch erlauben, irgendwelche Punktmengen zu manipulieren, zu inspizieren, usw.? Natürlich müssen wir, wenn wir dies gestatten, bei jeder Anforderung einer Dienstleistung diejenige Punktmenge identifizieren, mit der die Operation ausgeführt werden soll. Dann ist weiter die Frage: Woher bekommen wir die Punktmenge? Darauf ist die Antwort nicht schwer: Wir bekommen sie vom Modul selbst! (So wie wir annehmen, daß wir vom Besitzer der Autowerkstatt Autos auch kaufen können.) Dazu muß es eine spezielle Operation geben. Allerdings: So wie der Autokäufer einen Parkplatz (oder eine Garage) in der Nähe seines Hauses haben sollte, so muß auch der Kunde unseres zu erweiternden Moduls "Punktmenge" die Identifikationen der für ihn vom Modul verwalteten Objekte irgendwo unterbringen können. Die Definition des Moduls "Punktmenge" muß daher die Deklaration des Typs solcher Identifikationen (z.B. 'TYPE PunktmengenIdTyp;') enthalten. (Der Name eines solchen Typs entspricht, um im Bild unserer Analogie zu bleiben, dem Namen der Automarke!) Mit einem den Modul benutzenden Programm wird man sich dann per Deklaration von Variablen dieses Typs (z.B. 'VAR pm1, pm2: PunktmengenIdTyp;') so viele (Speicher für die)

Identifikationen von (verschiedenen) Punktmengen einrichten können, wie man davon benötigt. (Diese Identifikationen entsprechen, wenn man so will, z.B. den Fahrgestellnummern der gekauften Autos.) Entscheidend jedoch ist bei alledem, daß die Deklaration des Typs der Identifikationen selbst keinerlei Auskunft über die Struktur einer Punktmenge gibt, daß diese Typdeklaration also lediglich einen Typnamen (den Markennamen!) veröffentlicht. Die Struktur der Punktmenge muß weiterhin im Implementierungsteil des Moduls verborgen bleiben. Anschaulich gesprochen, wird mit einer derartigen Moduldefinition die folgende Aussage formuliert: "Ich verkaufe Objekte des Typs XYZ. Ich sage weder, wie diese Objekte aussehen noch wie sie aufgebaut sind, doch im Auftrag meiner Kunden kann ich damit folgendes tun: Operation-1, Operation-2, ... !"

Hier ist der Definitionsteil des nach diesen Überlegungen verallgemeinerten Moduls "Punktmenge" (inklusive der weiteren als nützlich erkannten Operationen):

```
DEFINITION MODULE Punktmenge; (*verallgemeinerte Version*)
(* Dieser Modul stellt seinen Kunden Punktmengen zur Verfügung und
verwaltet diese im Auftrag seiner Kunden. Er verbirgt die Repräsen-
tation von Objekten vom PunktmengenTyp vor den Kunden und gestattet
den Zugriff auf die Objekte nur über die im folgenden aufgelisteten
Operationen. Auf Objekte vom PunktmengenTyp muß die Operation
"PMLeeren" angewandt werden, bevor irgendeine andere der Operationen
des Moduls angewandt werden darf. *)

FROM Punkt IMPORT PunktTyp;

TYPE PunktmengenIdTyp;   (* Dies ist der besagte "Markenname"! *)

PROCEDURE PMLeeren(VAR pm: PunktmengenIdTyp);
(* Die Prozedur leert die durch "pm" identifizierte Punktmenge. *)

PROCEDURE PMLeer(pm: PunktmengenIdTyp): BOOLEAN;
(* Die Prozedur liefert TRUE, falls die durch "pm" identifizierte
Punktmenge leer ist, und FALSE sonst. *)

PROCEDURE PMCard(pm: PunktmengenIdTyp): CARDINAL;
(* Die Prozedur liefert die Mächtigkeit der durch "pm" identifizierten
Punktmenge. *)

PROCEDURE PMEinfuegen(VAR pm: PunktmengenIdTyp;
                      VAR punkt: PunktTyp);
(* Die Prozedur fügt den in "punkt" übergebenen Punkt in die durch "pm"
identifizierte Punktmenge ein. *)

PROCEDURE InitPMLesen(VAR pm: PunktmengenIdTyp);
(* Die Prozedur eröffnet den Vorgang des Lesens der Punkte der durch
"pm" identifizierten Punktmenge (die auch leer sein kann). *)
```

```
PROCEDURE PMLesen(VAR pm: PunktmengenIdTyp;
                  VAR punkt: PunktTyp;
                  VAR eopm: BOOLEAN);
(* Die Prozedur stellt den jeweils nächsten Punkt der durch "pm"
identifizierten Punktmenge in "punkt" zur Verfügung. Ist kein weiterer
Punkt vorhanden, so erhält "eopm" den Wert TRUE; sonst enthält "eopm"
den Wert FALSE.*)
PROCEDURE PMMaxPunkt(pm: PunktmengenIdTyp; VAR punkt: PunktTyp;
                     VAR err: BOOLEAN);
(* Falls die durch "pm" identifizierte Punktmenge nicht leer ist, liefert die
die Prozedur in "punkt" den absolut am weitesten vom
Koordinatenursprung entfernten Punkt der Punktmenge und in "err" den
Wert FALSE; andernfalls erhält "err" den Wert TRUE, und der Inhalt von
"punkt" ist undefiniert. *)
PROCEDURE PMLoeschePunkt(VAR pm: PunktmengenIdTyp;
                         VAR punkt: PunktTyp;
                         VAR err: BOOLEAN);
(* Falls die durch "pm" identifizierte Punktmenge den Punkt "punkt"
enthält, so wird dieser aus der Punktmenge entfernt, und "err" erhält den
Wert FALSE; andernfalls erhält "err" den Wert TRUE. *)
END Punktmenge.
```

Eine mögliche *Konkretisierung* des PunktmengenTyps kann dann wie folgt ausschauen:

```
IMPLEMENTATION MODULE Punktmenge; (* verallgemeinerte Version *)
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
...
TYPE PunktmengenIdTyp = POINTER TO PunktmengenVerwTyp;
     PunktmengenVerwTyp = RECORD
                            aktpkt: PMRefTyp;
                            punktMenge: PMRefTyp
                          END;
     PMRefTyp = POINTER TO PunktmengenTyp;
     PunktmengenTyp = RECORD
                        punkt: PunktTyp;
                        nxtpkt: PMRefTyp
                      END;
...
(* Implementation der Prozeduren zur Verwaltung von Punktmengen *)
END Punktmenge.
```

Daß innerhalb der Punktmengen-"Werkstatt" außer der Deklaration des "PunktmengenTyps" und des "PMRefTyps" (die uns beide schon aus der vorangegangenen Diskussion vertraut sind) auch die eines "PunktmengenVerw(altungs)Typs" benötigt wird, ist der Tatsache zuzuschreiben, daß den "Arbeitern" in dieser "Werkstatt" je bearbeitetem Objekt gewisse Informationen über dieses Objekt bekannt sein müssen. (Die Automechaniker etwa werden wissen wollen, welche Arbeiten bei der letzten Wartung erledigt wurden, und daher intern je Wagen eine eigene Akte anlegen!) Ein Objekt und die über das Objekt notwendige Information sind in unserem Beispiel in einem RECORD mit den durch die Deklaration des "PunktmengenVerwTyps" gegebenen Komponenten zusammengefaßt. Im vorliegenden Fall betrifft jene "Metainformation" ("aktpkt") die Lage des Punktes (innerhalb der die Menge repräsentierenden linearen Liste), der nach der nächsten Ausführung der "PMLesen"-Operation abzuliefern ist. Es ist nun dieses, aus einer Punktmenge und der zugehörigen Verwaltungsinformation bestehende Objekt, auf das sich der "Kunde" bezieht. Bemerkenswert (zum wiederholten Male) ist, daß er sich dabei der Existenz solcher Zusatzinformationen überhaupt nicht gewahr sein muß.

In MODULA-2 ist übrigens ein in einem Definitionsmodul deklarierter Datentyp, mit dessen Ausprägungen - wie im Beispiel - irgendwelche Objekte identifiziert werden (hier der "PunktmengenIdTyp"), im Implementationsmodul immer als POINTER-Typ zu realisieren. (Die Identifikation zeigt auf ein Objekt!) Natürlich ist der Leser eingeladen, die Prozeduren des Moduls "Punktmenge" auf der Grundlage der durch die obigen Typdeklarationen gegebenen Repräsentation auszuformulieren. Tut er dies, so wird er erkennen, warum wir bereits vorsorglich die IMPORT-Klausel "FROM Storage IMPORT ... " notiert haben.)

Das Stichwort für die Charakterisierung von Modulen der soeben betrachteten Art ist bereits beinahe gefallen: Wir sprachen von der *Konkretisierung* des PunktmengenTyps im Implementationsmodul! Mit anderen Worten: Soweit es den Definitionsmodul betraf, hatten wir es offenbar nicht mit einem konkreten, sondern mit einem *Abstrakten Datentyp* zu tun! In der Tat: Wir hätten dem Benutzer des Moduls "Punktmenge" durchaus die Illusion geben können, daß er nicht eigentlich nur Identifikationen von Punktmengen sieht, sondern die Punktmengen selbst. Es macht für ihn (fast) keinen Unterschied, ist er doch nur an den jeweils möglichen Operationen interessiert. Anstatt "TYPE PunktmengenIdTyp;" hätten wir im Definitionsmodul also auch einfach "TYPE Punktmenge;" schreiben können (mit den entsprechenden nachfolgenden Notations-Änderungen in den Parameterlisten und im Implementationsteil, versteht sich).

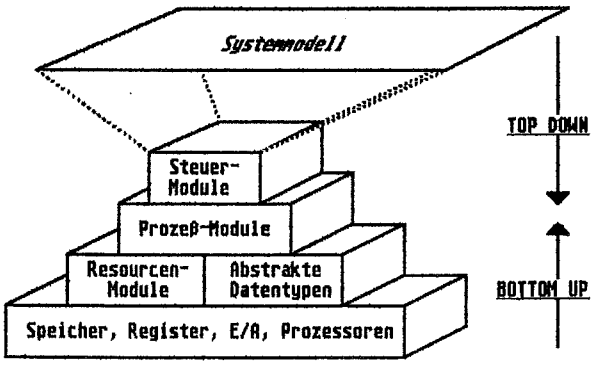
Der (nun so genannte) *Abstrakte Datentyp - Modul* "Punktmenge" kann in (fast) der gleichen Weise benutzt werden wie die der Programmiersprache inhärenten skalaren Typen CARDINAL, INTEGER oder REAL. Auch diese sind letztlich durch die mit den entsprechenden Objekten ausführbaren (arithmetischen und

sonstigen) Operationen definiert. Wir sagen "fast", denn es genügt nicht, einfach die Variablendeklaration "VAR pm:Punktmenge;" zu notieren, um mit "pm" auch schon arbeiten zu können; "pm" muß zuerst *initialisiert* werden, in unserem Beispiel mit der Operation "PMLeeren". Im Gegensatz dazu besorgt dies das Kompiliersystem automatisch bei Deklarationen von Variablen für Objekte der oben genannten (und weiteren) skalaren Typen.

Wie auch immer: Die Möglichkeit, in einer Programmiersprache Abstrakte Datentyp-Module definieren zu können, kann als Erweiterbarkeit der Sprache verstanden werden. Eine derart erweiterbare Sprache bildet das Material, aus dem wir die Bausteine für das "Haus" Software-System backen. Ein kleiner Baustein für das System "Konvexe Hülle" wäre zum Beispiel der folgende abstrakte Datentyp "Punkt" (für den wir gleich ein paar weitere nützliche Operationen definiert haben):

```
DEFINITION MODULE Punkt;
TYPE PunktTyp;
PROCEDURE InitPunkt(x,y:REAL):PunktTyp;
(* Liefert einen Punkt mit den kartesischen Koordinaten x und y *)
PROCEDURE Xkoord(p:PunktTyp):REAL;
(* Liefert den Wert der x-Koordinate des Punktes p *)
PROCEDURE Ykoord(p:PunktTyp):REAL;
(* Liefert den Wert der y-Koordinate des Punktes p *)
PROCEDURE DistUrsprung(p:PunktTyp):REAL;
(* Liefert die Entfernung des Punktes p vom Ursprung des
Koordinatensystems *)
PROCEDURE Abstand(p1,p2:PunktTyp):REAL;
(* Liefert den Abstand der Punkte p1 und p2 voneinander *)
END Punkt.
```

Fassen wir zusammen: Die Bewohner des "Hauses" Software-System und ihre Bedürfnisse sind durch das *Systemmodell* beschrieben. Das "Fundament" des Hauses ist andererseits durch die *Hardware* gegeben. Das Betriebssystem, ein eventuell vorhandenes Datenbank-Management-System sowie die Programmiersprache(n) stellen *Ressourcen-Module* und *Abstrakte Datentypen* zur Verfügung, oder ermöglichen deren Herstellung. Ihre Dienste bilden die Grundlage für das durch *Steuer-Module*



kontrollierte Wirken der anwendungsspezifischen *Prozeß-Module*, denen sie *Bottom-Up* entgegengebaut werden, und welche ihrerseits *Top-Down* aus dem Systemmodell abzuleiten sind. (Sie entsprechen - um die Analogie zu wahren - den Bewohnern des Hauses.) Natürlich werden die Dienste der Ressourcen-Module und der Abstrakten Datentyp (ADT) Module nicht frei erfunden; ihre Notwendigkeit muß sich jeweils aus den Bedürfnissen der Prozeß-Module ergeben, und ein guter Teil der Entwurfsarbeit besteht gerade darin, diese Bedürfnisse zu erkennen und - soweit sie durch die vorhandene Basissoftware nicht unmittelbar zu befriedigen sind - in *Modul-Spezifikationen* (die wir bisher als kommentierte Definitionsmodule formuliert haben) umzusetzen. Doch schadet es nicht, wie an den Beispielen dieses Abschnitts gezeigt wurde, bei dieser Spezifikation Weitsicht walten zu lassen und mehr als die unbedingt notwendigen Dienstleistungen zu definieren.

Wir werden übrigens später (im letzten Abschnitt dieses Kapitels) sehen, daß sich die Dienste der zwischen Hardware und Anwendung gelegenen Schicht ganz direkt auch auf die Unterstützung der Kommunikation zwischen den Anwendungsprozessen beziehen können. Indirekt gilt dies schon für unseren Beispielmodul "Punktmenge", der ja gewissermaßen die Funktion eines Kanals zwischen den Prozessen "Einlesen", "Berechnen" und "Ausgeben" übernahm. Der speziellen Voraussetzungen über das Betriebssystem unseres Rechners wegen benötigten wir zur Implementierung des Systemmodells "Konvexe Hülle" den Steuermodul zur Vermittlung der Kommunikation zwischen den drei Anwendungsprozessen. Unter anderen Voraussetzungen kann - und dies wird im letzten Abschnitt klar werden - ein solcher Steuermodul entfallen und der Austausch von Datenobjekten zwischen den Prozesse durch "betriebssystemnahe" Module geregelt werden.

Über diese Bemerkungen hinaus sollte klar sein, daß das Bild, welches wir oben von unserem "System-Haus" gezeichnet haben, nur ein sehr grobes ist. Auf viele mögliche Verfeinerungen einzugehen ist jedoch nicht möglich ohne eine gründlichere Betrachtung spezieller Software-Systeme. Darauf müssen wir leider verzichten.

### 6.4.3 Abstrakte Datentypen

Die im vorigen Abschnitt besprochenen kommentierten Definitionsmodule dienen zwei Zwecken: Erstens legen sie die Namen und (die Typen und Reihenfolge der) Parameter der Prozeduren fest, die den einzelnen Leistungen eines Moduls entsprechen. Insbesondere wird damit dem Benutzer eines Moduls der Aufbau des programmiersprachlichen Ausdrucks mitgeteilt, welchen er notieren muß, um den Modul zur Arbeit zu bewegen. Dieser Zweck betrifft also die *Syntax*, die Formulierung eines Auftrags an einen Modul. Verzichtet man auf jeglichen Kommentar zu Prozedurnamen und Parametern, sowie zu den eventu-

ellen Beziehungen der Prozeduren (Operationen) untereinander, so ist dies in der Tat der einzige Zweck, den ein MODULA-2 Definitionsmodul erfüllen kann. Zweitens sollte der Benutzer eines Moduls aus dem Text der Definition jedoch auch erfahren, was die Prozeduren leisten, welchen Operationen sie entsprechen, wie diese Operationen zusammenhängen, und welchen Regeln und/oder Einschränkungen ihre Anwendung unterliegt. Mit anderen Worten: der Definitionstext muß Auskunft geben über die Bedeutung, die *Semantik* der von einem Modul erbrachten Dienste.

Einer so wichtigen Anforderung nur mit dem informellen, durch keinerlei Regeln gebundenen Sprachmittel des "Kommentars" genügen zu können, gibt durchaus Anlaß zu Unbehagen. Denn schließlich bildet die Formulierung der Semantik eines Moduls jenen "Nutzungsvertrag", in dem die Rechte und Pflichten beider Parteien, Diensterbringer und Dienstnachfrager, niedergelegt sind (vgl. die entsprechenden Bemerkungen im vorigen Abschnitt). Und ein Rechtsstreit auf der Grundlage unklarer Gesetze oder nicht eindeutiger Verträge ist gewiß kein Vergnügen!

Man stelle sich vor: Einem Benutzer des ADT-Moduls "Punkt" (siehe Ende des vorigen Abschnitts / ADT = Abstrakter Daten Typ ) fällt plötzlich auf, daß die von den Prozeduren "Xkoord", "Ykoord" und "DistUrsprung" nach Anwendung auf einen Punkt p gelieferten Werte keineswegs der Beziehung

DistUrsprung(p) = Quadratwurzel(Xkoord(p)^2 + Ykoord(p)^2)

genügen, so wie dies - gemäß analytischer Geometrie - eigentlich zu erwarten wäre. Offenbar hat der Implementierer des ADT-Moduls "Punkt" entweder einen Fehler gemacht, oder er hat die Spezifikation des Moduls nicht in dem gleichen Sinne verstanden wie der Entwerfer des Moduls.

Aber andererseits: Warum hat der Entwerfer des Moduls in seiner Spezifikation nicht klar und unmißverständlich gesagt, was er will? So daß die Spezifikation selbst die Kriterien für die Korrektheit der Implementierung liefert?

Anstatt quasi tautologisch zu sagen "Xkoord liefert den Wert der x-Koordinate des Punktes p", hätte er formal deutlich machen können, was er unter der x-Koordinaten eines Punktes in der kartesischen Ebene versteht. Er hätte nur den entsprechenden und offenbar von ihm gewünschten Zusammenhang zwischen den Operationen "InitPunkt" und "Xkoord" darzustellen brauchen:

Xkoord (InitPunkt (x,y)) = x.

Dies ist eine von jeder Implementierung des ADT-Moduls "Punkt" (welcher Mittel sie sich auch immer bedienen mag) zu erfüllende Bedingung, und eine ohne weiteres nachprüfbare dazu. Das gilt natürlich auch für die bereits genannte Beziehung zwischen "DistUrsprung", "Xkoord" und "Ykoord", sowie für die entsprechenden, die Bedeutung der beiden übrigen Operationen definierenden Gleichungen.

Es scheint also sehr wohl möglich zu sein, die Semantik eines ADT-Moduls soweit zu formalisieren, daß die Korrektheit (oder Fehlerhaftigkeit) einer Implementierung auf der Grundlage einer solchen Formalisierung nachgewiesen werden kann. Genauer: Die Formalisierung der Semantik im Rahmen der Spezifikation eines ADT-Moduls sollte es einerseits erlauben, mit Hilfe von Tests das Vorhandensein von Fehlern in einer fehlerhaften Implementierung festzustellen, und anderseits sollte sie dazu geeignet sein, die Korrektheit einer korrekten Implementierung zu verifizieren. Es ist das relativ bescheidene Hauptanliegen dieses Abschnitts, die Einsicht in die Möglichkeit solcher Korrektheitsbeweise zu vermitteln. Der Leser möge sich hier an Kapitel 4 "Programmieren durch Beweisen" erinnern, in dem wir zu Beginn ausdrücklich darauf hinwiesen, daß Programm-Verifikation dort nicht stattfinden werde. Das durch Begriffe wie "schwächste Vorbedingung", "Nachbedingung" und "Schrankenfunktion" gekennzeichnete Instrumentarium setzten wir vielmehr zur Programm-Konstruktion ein. Jetzt freilich sind wir in einer ganz anderen Situation, haben es nicht mit "Bedingungen über Speicherzustände" (für deren Erfüllbarkeit die Programme in Kapitel 4 die "Beweise" waren) zu tun, sondern mit Bedingungen über das Zusammenspiel abstrakter Operationen, und wir werden sehen, daß es hier darum gehen wird, die Erfülltheit dieser Bedingungen aufgrund der speziellen Eigenschaften einer Implementierung zu beweisen. (Wir wollen damit natürlich nicht sagen, daß die Konstruktion der Implementierung eines ADT-Moduls nicht bei jenen Bedingungen ansetzen sollte, im Gegenteil. Aber auch das werden wir noch genauer sehen.)

Hinsichtlich Form und Inhalt orientieren wir uns in diesem Abschnitt an [GHM]. Es versteht sich allerdings fast von selbst, daß wir im Rahmen dieses, der Praxis des Programmierens verpflichteten Buches gewisse Probleme in mathematische Grundlagenbereiche der Informatik, wie Logik und Berechenbarkeitstheorie, auch dann nicht verfolgen werden, wenn sie eine derartige Vertiefung eigentlich verdienten.

Zur Verwirklichung des Vorhabens, formale Beweise der Korrektheit von Implementierungen eines ADT-Moduls zu führen, bedarf es nun offensichtlich einer geeigneten Notation für Syntax und Semantik eines solchen Moduls. Streng genommen müßten wir auch für diese Spezifikationssprache (wie für jede auf praktische Anwendungen hin konzipierte formale Sprache) Syntax und Semantik exakt definieren. Wir verzichten auf diese (sicherlich nicht triviale) Übung und vertrauen darauf, daß der Leser das nötige Verständnis aus dem Umgang mit den Beispielen gewinnt. Nur soviel sei bemerkt (vgl. [GHM]):

Mit unserer Notation müssen darstellbar sein:
- die sequentielle Verknüpfung von Operationen (Hintereinanderausführung),
- die Gleichheitsrelation (z.B. mit "="),
- zwei unterschiedliche Konstanten (TRUE und FALSE genannt),
- unbeschränkt viele freie Variable (mit geeigneten Namen).

Ferner verlangen wir:

- Die booleschen Operatoren AND, OR, NOT, => (Implikation) und <=> (Äquivalenz) stehen zur Verfügung, ebenso ganze (und - falls nötig - auch reelle) Zahlen sowie die zugehörigen arithmetischen Operatoren.
- Es gibt eine Operation "IF-THEN-ELSE (b,q,r)", deren erstes Argument ein boolescher Ausdruck ist, und deren weitere Argumente beliebigen Typs sein können. Für diese Operation gilt:
  IF-THEN-ELSE (TRUE,q,r) = q und IF-THEN-ELSE (FALSE,q,r) = r.
  Anstatt "IF-THEN-ELSE (b,q,r)" schreiben wir auch "IF b THEN q ELSE r".

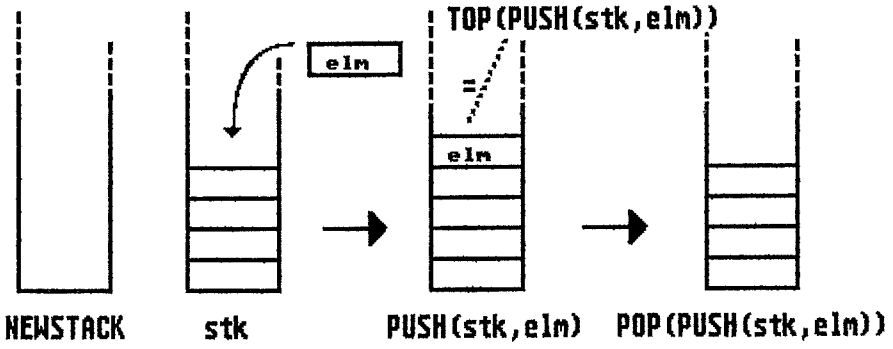
Die Schlüsselworte unserer Spezifikationssprache werden wir durch Unterstreichen hervorheben.

Als erste Beispiele formulieren wir die Spezifikationen zweier wohlbekannter Datentypen: "Stack" und "Array". Da unsere Notation an keine Programmiersprache gebunden ist (also insbesondere nicht an MODULA-2), brauchen wir nicht den Typ der Elemente zu präzisieren, die wir in "Stack" beziehungsweise "Array" unterbringen wollen. Es kann sich um irgendeinen Typ handeln. In der ADT Spezifikation bringen wir dies dadurch zum Ausdruck, daß dem Typnamen "Stack" ein Parameter vom Typ "Type" beigefügt wird. ("Type" ist also gewissermaßen der Typ der Typen, eine - wie man wissen sollte - etwas gewagte Konstruktion, doch für unsere Zwecke gerade noch vertretbar.) Man beachte ferner den Gebrauch von Groß- und Kleinschreibung, bei dem wir uns ebenfalls nicht an unsere bisherigen programmierstilistischen Konventionen halten müssen.

```
type    Stack [elementtype: Type];
syntax
        NEWSTACK -> Stack;
        PUSH (Stack, elementtype) -> Stack;
        POP (Stack) -> Stack;
        TOP (Stack) -> elementtype ∪ { UNDEF };
        ISNEW (Stack) -> Boolean;
        REPLACE (Stack, elementtype) -> Stack;
semantics
  declare  stk:Stack; elm:elementtype;
        POP (NEWSTACK) = NEWSTACK;
        POP(PUSH(stk,elm)) = stk;
        TOP(NEWSTACK) = UNDEF;
        TOP(PUSH(stk,elm)) = elm;
        ISNEW(NEWSTACK) = TRUE;
        ISNEW(PUSH(stk,elm)) = FALSE;
        REPLACE(stk,elm) = PUSH(POP(stk), elm).
```

Einige Bemerkungen zum Verständnis der formalen und inhaltlichen Elemente dieser Spezifikation scheinen nun durchaus angebracht:

- Die Spezifikation besteht aus drei Abschnitten, dem Typ-Abschnitt, dem Syntax-Abschnitt und dem Semantik-Abschnitt.
- Der Typ-Abschnitt gibt den Namen sowie den oder die Parameter des Typs der fraglichen Objekte bekannt.
- Es ist hilfreich, sich einen solchen Typ als Menge von Objekten vorzustellen und die mit diesen Objekten ausgeführten Operationen als Abbildungen zwischen Mengen. Die Abbildungen können gar kein Argument, ein Argument oder mehrere Argumente haben. (Komponenten des Wertebereichs können mit Komponenten des Argumentbereichs identisch sein. In diesem Fall hat man eine Operation, bei deren Anwendung ein Objekt gegebenen Typs in ein anderes Objekt des gleichen Typs umgewandelt wird.)
- Diese Vorstellung liegt dem Syntax-Abschnitt zugrunde. Der Operation NEWSTACK zum Beispiel entspricht eine Abbildung ohne Argumente. Eine solche Abbildung wird üblicherweise als Verweis auf ein spezielles Element der Menge interpretiert. Die Operation NEWSTACK liefert dieses Element. Bei der Operation PUSH ist der Wertebereich mit einem der Argumentbereiche identisch. PUSH erzeugt also aus einem Element der "Menge" "Stack" und einem Element der "Menge" "elementtype" ein Element von "Stack".
- Die im Semantik-Abschnitt geforderten Resultate der Verknüpfung der Operationen POP, TOP und ISNEW mit der Operation NEWSTACK beschreiben die wesentlichen Eigenschaften des von NEWSTACK gelieferten Objektes.
- Das Stichwort ist gefallen: Im Semantik-Abschnitt werden *Forderungen* aufgestellt, Forderungen hinsichtlich des Effekts und des Zusamenwirkens der im Syntax-Abschnitt aufgelisteten Operationen. Diese Forderungen erhalten die Form von Gleichungen über freie Variable für Objekte der jeweils von den Operationen behandelten Typen. Die Variablen werden in einer "declare-Klausel" benannt. Die im Semantik-Abschnitt aufgelisteten Gleichungen werden auch als *Postulate* oder *Axiome* bezeichnet.
- Der Semantik-Abschnitt formalisiert also die intuitive Vorstellung des Entwerfers von dem, was man mit den fraglichen Objekten tun kann, zum Beispiel hier mit Objekten vom Typ "Stack":



Von einem Array verlangen wir (neben der nun schon quasi Standardforderung, mittels einer Operation NEWARRAY ein solches überhaupt anlegen zu dürfen), daß in ihm Objekte eines gegeben Typs (des "rangetype") untergebracht werden können (mittels einer Operation ASSIGN), und daß man auf diese Objekte zugreifen kann (mittels einer Operation ACCESS). Die "Behälter" für Objekte vom "rangetype" werden ihrerseits durch Objekte eines "domaintype" identifiziert (bzw. *indexiert*). (Jeder im Gebrauch einer höheren Programmiersprache wie z.B. MODULA-2 Geübte weiß, daß als "domaintype" keineswegs nur Subtypen etwa der ganzen oder natürlichen Zahlen in Frage kommen!)

Wir verlangen ferner, daß der Gebrauch dieser Operationen den folgenden Regeln unterworfen ist:

- In einem durch NEWARRAY angelegten Array gibt es keine Objekte.
- Wird einem durch "dval" indexierten Behälter des Arrays ein Objekt "rval" zugewiesen, so sollte der Zugriff auf diesen Behälter auch genau dieses Objekt wieder zutage fördern.

Diese intuitive Vorstellung von einem Array und die damit verbundenen Forderungen werden durch den folgenden ADT formalisiert:

```
type Array [domaintype: Type, rangetype: Type ];
syntax
      NEWARRAY –> Array;
      ASSIGN (Array, domaintype, rangetype) –> Array;
      ACCESS (Array, domaintype) –> rangetype ∪ {UNDEF};
semantics
  declare  arr: Array; dval, dval1: domaintype; rval: rangetype;
      ACCESS (NEWARRAY, dval) = UNDEF;
      ACCESS (ASSIGN (arr, dval, rval), dval1) = IF dval=dval1 THEN
                                                    rval
                                                 ELSE
                                                    ACCESS (arr, dval1).
```

Die beiden Beispiele verdeutlichen bereits ein allgemeines, bei der Definition eines ADT zu beachtendes Prinzip. Wir erinnern uns: Die von einem Datenmodul angebotenen Dienste lassen sich den vier Kategorien *Initialisierung*, *Transformation*, *Information* und *Extraktion* zuordnen (vgl. Abschnitt 6.4.2). Mutatis mutandis gilt diese Zuordnung auch für die Operationen eines ADT. So sind NEWSTACK und NEWARRAY die Initialisierungsoperationen der ADTn "Stack" beziehungsweise "Array". PUSH, POP und REPLACE sind transformierende Operationen des ADT "Stack", und das gleiche gilt für "ASSIGN" bezüglich "Array". Eine eingehendere Betrachtung der genannten Stack-Operationen lehrt nun jedoch, daß eine feinere Unterscheidung der Transformationen möglich ist. Die Operation PUSH nämlich nimmt insofern eine Sonderstellung ein, als sie für den Aufbau, das "Wachstum" oder - wie wir sagen werden - die *Konstruktion*

von Objekten des Typs "Stack" sorgt. Im Falle des Array übernimmt ASSIGN diese Rolle. Im Semantik-Abschnitt einer ADT-Definition werden - und dies ist das Prinzip, auf das wir hinaus wollen - die Wirkung der Initialisierungs- und Konstruktions-Operationen auf die übrigen Operationen beschrieben. Was, so lautet die Frage für unser erstes Beispiel, soll die Operation POP (bzw. TOP, ISNEW) liefern, wenn wir zuvor die Operation NEWSTACK ausgeführt haben? Und was liefert sie nach Anwendung der Konstruktions-Operation PUSH? Entsprechend postulieren wir bei "Array" das Resultat der Anwendung von ACCESS nach vorheriger Ausführung der Operationen NEWARRAY beziehungsweise ASSIGN.

Eine gewisse Sonderstellung nimmt die Operation REPLACE des Stack-Beispiels ein. Wie sich später herausstellen wird, haben wir sie aus purer Bequemlichkeit eingeführt. Aus der Beschreibung ihrer Semantik geht nämlich hervor, daß sie nichts anderes bewirkt als die Hintereinanderausführung der (elementareren) Operationen POP und PUSH. Unserer intuitiven Vorstellung gemäß ersetzt sie was immer sich in einem Stack "zuoberst" befindet durch ein anderes Objekt vom "elementtype". Der Semantik-Abschnitt eines ADT kann also auch derartige "Abkürzungsdefinitionen" enthalten.

Die hier vorgestellte Formalisierung von ADT-Modulen erlaubt es nun, den Begriff der *Implementierung* wesentlich schärfer zu fassen, als dies bisher möglich war. Die in Abschnitt 6.1 gegebene Erklärung läßt sich kurz wie folgt zusammenfassen:

> "Implementierung ist die Realisierung eines gewünschten und entsprechend spezifizierten Verhaltens mit Hilfe gegebener oder neu zu schaffender Mittel."

Diese "Definition" können wir ohne weiteres in die Terminologie der Abstrakten Datentypen übertragen:

> "Die Implementierung eines Abstrakten Datentyps A ist die Realisierung von dessen Objekten und Operationen mit Hilfe der Objekte und Operationen eines oder mehrerer ADTn B, C, ... , die entweder bereits gegeben sind oder in geeigneter Weise neu definiert werden müssen."

Konkret bedeutet dies: Wir benötigen

(i) eine Vorschrift, nach der die Objekte $\Omega_A$ von A durch Objekte aus $\Omega_B$, $\Omega_C$, ... dargestellt werden, und

(ii) auf diesen Darstellungen beruhende Formulierungen der Operationen von A durch Operationen von B, C, ... .

Die Implementierung eines ADT wird also aus zwei Teilen bestehen, von denen der erste, (i) entsprechende, lediglich erklärt, daß eine Folge von Objekten aus

$\Omega_B, \Omega_C, \ldots$ ein Objekt aus $\Omega_A$ repräsentiert. Formal kann dies durch eine Abbildung REP: $\Omega_B \times \Omega_C \ldots \rightarrow \Omega_A$ ausgedrückt werden. "REP" ist dabei ein frei wählbarer Name für die Repräsentation.

Gemäß Definition des ADTs A sind dessen Objekte völlig "amorph"; das einzige, was im Definitionstext über sie bekanntgegeben wird, ist, was man mit ihnen tun kann (die "A-Operationen"), und die Gesetzmäßigkeiten, denen die "A-Operationen" unterliegen sollen. Durch den Repräsentationsteil der Implementierung erhalten die Elemente von $\Omega_A$ eine Struktur als Tupel $(\omega_B, \omega_C, \ldots) \in \Omega_B \times \Omega_C \ldots$. Die Frage ist nun: Welche "B-Operationen", "C-Operationen" usw. sind auf die Elemente eines solchen Tupels anzuwenden, um insgesamt die in der Definition von A geforderten Effekte zu erzielen? Die Antwort hierauf muß der zweite Teil des Implementierungs-Textes geben. Sie besteht aus Gleichungen, welche im einfachsten Falle von der Gestalt

$$A\_OP\_i\,(REP(\omega_B, \omega_C, \ldots)) = REP(B\_OP\_j(\omega_B), C\_OP\_k(\omega_C), \ldots)$$

sind. Dabei stehen A_OP_i, B_OP_j und C_OP_k jeweils für eine der A-(B-, C-) Operationen. Im allgemeinen können die rechten Seiten dieser Gleichungen Ausdrücke sein, deren Syntax beispielsweise in derjenigen der oben skizzierten ADT-Spezifikationssprache enthalten ist. (Man beachte insbesondere, daß es diese Sprache erlaubt, Ausdrücke über boolesche Konstanten und Variablen zu bilden!) Wir bezeichnen solche Ausdrücke als *Programme*, da mit ihnen gesagt wird, wie eine gegebene A-Operation durch geeignete Operationen auf den Objekten der Repräsentation zu realisieren ist. Wir illustrieren diese *abstrakte Implementierung* abstrakter Datentypen durch ein erstes einfaches Beispiel: die Implementierung des ADTs "Stack" durch die ADTn "Array" und "Integer". (Letzteren stellt uns die Spezifikationssprache implizit zur Verfügung, s.o.! Was die Parameter des ADTs "Array" betrifft, so sind wir frei zu wählen, was immer uns opportun erscheint. Der aufmerksame Leser wird feststellen, daß wir in diesem Beispiel nichts anderes tun, als die oben graphisch skizzierte Vorstellung, die wir von einem Stack haben, zu formalisieren. Von *abstrakter* Implementierung sprechen wir, weil die dabei verwendeten Mittel ihrerseits abstrakte Datentypen sind.)

```
representation STACK(Array[Integer,elementtype], Integer) ->
                                                    Stack[elementtype];
programs
  declare arr: Array; t: Integer; elm: elementtype;
    NEWSTACK = STACK(NEWARRAY, 0);
    PUSH(STACK(arr,t), elm) = STACK(ASSIGN(arr,t+1,elm), t+1);
    POP(STACK(arr,t)) = IF (t=0) THEN
                          STACK(arr,0)
                        ELSE
                          STACK(arr,t-1);
```

```
TOP(STACK(arr,t)) = ACCESS(arr,t);
ISNEW(STACK(arr,t)) = (t=0);
REPLACE(STACK(arr,t), elm) = IF (t=0) THEN
                               STACK(ASSIGN(arr,1,elm),1)
                             ELSE
                               STACK(ASSIGN(arr,t,elm),t).
```

Den oben vorgebrachten allgemeinen Bemerkungen über den program-Teil einer Implementierung zufolge sollen die jeweils rechten Seiten dieser Gleichungen, interpretiert als Operationen auf Stack-Repräsentationen, eben jenen Gesetzmäßigkeiten unterliegen, die wir mit der Definition des ADTs "Stack" gefordert haben. Ist dies der Fall, so bezeichnen wir die Implementierung als *korrekt*, und es stellt sich nun die Frage, ob wir diese Eigenschaft formal *beweisen* können.

Zum Beispiel haben wir für irgendein Objekt "stk" vom Typ "Stack" und ein "elm" vom Typ "elementtype" verlangt, daß gilt: TOP(PUSH(stk, elm)) = elm. Gilt dies auch, so lautet nun unser Problem, wenn wir "stk" durch irgendeine Repräsentation STACK(arr,t) ersetzen, und TOP und PUSH durch ihre "Programme" auf den entsprechenden rechten Seiten des obigen program-Teils? Versuchen wir es! Sei STACK(arr,t) = stk $\in \Omega_{\text{Stack}}$; dann ist

```
TOP(PUSH(stk,elm)) = TOP(PUSH(STACK(arr,t), elm))
                     (* Ersetze PUSH durch das PUSH-program: *)
                   = TOP(STACK(ASSIGN(arr,t+1,elm), t+1))
                     (* Ersetze TOP durch das TOP-program: *)
                   = ACCESS(ASSIGN(arr,t+1,elm), t+1)
                     (* Wende ACCESS-Axiom des ADT Array an: *)
                   = elm
```

Damit ist in der Tat gezeigt, daß das vierte, im semantics-Teil der Definition des ADTs "Stack" aufgelistete *Postulat* (bzw. *Axiom* oder, um es im Rahmen unseres juristischen Bildes zu formulieren, die vierte "Stack-Vertragsklausel") durch die Programme unserer Implementierung erfüllt wird. Die in den Programmen von TOP und PUSH festgelegte Anwendung der Array- und Integer-Operationen führt, bei entsprechender Verknüpfung, zu eben dem Ergebnis, welches für die Hintereinanderausführung der Stack-Operationen TOP und PUSH gefordert wurde. Der Leser möge sich am Beweis der Erfülltheit der übrigen, an Stack-Operationen gestellten Forderungen versuchen.

Wir wollen die hier "in der Nußschale" vorgeführte Technik der abstrakten Implementierung noch an einem zweiten, etwas umfangreicheren und ebenfalls aus [GHM] referierten Beispiel demonstrieren. Es hat mit der Aufgabe zu tun, für eine *blockstrukturierte* Programmiersprache einen Compiler zu bauen. Um es zu verstehen, müssen wir uns zunächst einige, diesen Gegenstandsbereich betreffende Tatsachen in Erinnerung rufen.

In einer *blockstrukturierten* Programmiersprache geschriebene Programme bestehen aus ineinandergeschachtelten Textblöcken. Beispiele hierfür sind die Prozeduren der in diesem Buch verwendeten Sprache MODULA-2. Innerhalb eines solchen Textblocks können Variablen deklariert werden, die nur für den die Deklaration umfassenden Textblock Gültigkeit besitzen. Man sagt, der *Scope* (oder *Gültigkeitsbereich*) einer Variablen ist der Block des Programmtextes, in dem sie deklariert wurde. Eine Variablendeklaration legt den Variablen-Namen (einen *Identifier*) und den Typ der Variablen fest. Angenommen, ein Block B1.1 ist in einem (äußeren) Block B1 enthalten. In B1 werde eine Variable vom Typ INTEGER und in B1.1 eine Variable vom Typ REAL benötigt. Obwohl verschieden kann ihnen der Programmierer den gleichen Namen (sagen wir "y") geben. Der so entstehende Namenskonflikt wird gelöst, indem die Deklaration der Variablen y im äußeren Blocks durch die Deklaration der Variablen gleichen Namens im inneren Block (B1.1) aufgehoben wird. Anders gesagt: In B1.1 ist das in B1 deklarierte y nicht mehr bekannt. Jede Referenz in B1.1 auf eine Variable namens "y" meint das in B1.1 deklarierte "y". Das folgende einfache Beispiel (in dem BLOCKANF und BLOCKEND die den Anfang und das Ende eines Blockes markierenden Schlüsselworte einer hypothetischen Programmiersprache sind) erläutert diesen Sachverhalt:

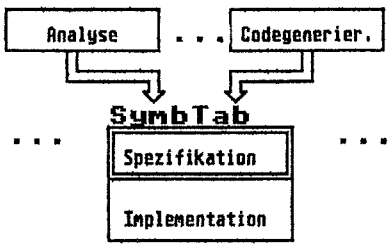
```
BLOCKANF          (* Block B1 *)
   VAR x, y: INTEGER;
   .
   .   (* Hier bezeichnet "y" eine Variable V1 vom Typ INTEGER *)
   .
   BLOCKANF       (* Block B1.1 *)
     VAR y: REAL;
     .
     .  (* Hier bezeichnet "y" eine Variable V2 vom Typ REAL *)
     .
   BLOCKEND
   .
   .   (* Hier bezeichnet "y" wieder die Variable V1 vom Typ INTEGER *)
   .
BLOCKEND
```

Wir haben hier, um die tatsächliche Verschiedenheit der beiden, mit dem gleichen Namen "y" identifizierten Variablen hervorzuheben, im Kommentar die zusätzlichen Bezeichnungen "V1" und "V2" eingeführt. Im Gegensatz zu "y" übrigens, ist die als INTEGER in B1 deklarierte Variable mit dem Namen "x" auch in B1.1 unter diesem Namen zugreifbar.

Ein Compiler hat nun die Aufgabe, nach sorgfältiger Analyse des Programmtextes den Speicher für die im Programm deklarierten Variablen zu strukturieren und den auf diese Variablen Bezug nehmenden Code zu generieren. Dazu bedient er sich einer *Symboltabelle* genannten Datenstruktur, in der er die im Verlauf der Analyse gefundenen Variablen-Bezeichner (*Identifier*) und deren

*Attribute* (also unter anderem die zugehörigen Typen) registriert. Aus dieser Symboltabelle muß zum Beispiel hervorgehen, ob ein gegebener Identifier in einem gegebenen Block (bzw. Scope) bereits deklariert wurde. Insofern muß die Symboltabelle die Blockstruktur des zugehörigen Programms exakt wiedergeben.

Nach allem was wir bisher gelernt haben, sind wir als Compilerbauer gehalten, die Symboltabelle in einem speziell für sie zuständigen Modul (nennen wir ihn "SymbTab") zu verpacken, der den Analyse- und Übersetzungsprozessen eine Reihe von Dienstleistungen zur Manipulation von Objekten des Typs "Symboltabelle" anbietet. Welche Dienstleistungen dies sein müssen, ergibt sich aus den oben kurz skizzierten Aufgaben, die der Compiler mit Hilfe der Symboltabelle zu erledigen hat. Wir beschreiben diese Dienste im folgenden als Operationen:




| | |
|---|---|
| INIT | Anlegen einer Symboltabelle für den äußersten Block. (Zu Beginn der Analyse des Programmtextes.) |
| ENTERBLOCK | Erweiterung der Symboltabelle bei Eintritt in einen inneren Block.(Bei Erkennen einer BLOCKANF-Marke.) |
| ADDID | Einfügen eines Variablen-Bezeichners und der zugehörigen Attribute. (Nach Erkennen einer Variablen-Deklaration in einem Block.) |
| LEAVEBLOCK | Reduktion der Symboltabelle beim Austritt aus einem inneren Block. (Bei Erkennen einer BLOCKEND-Marke.) |
| ISINBLOCK | Feststellen, ob ein Identifier im aktuellen Block bereits verwendet wurde. |
| RETRIEVE | Extraktion der einem gegebenen Identifier zugeordneten Attribute. |

Es ist klar, daß das Zusammenspiel dieser Operationen gewissen Gesetzen gehorchen muß. Was liegt daher näher, als unseren Modul "SymbTab" als ADT-Modul mit dem abstrakten Datentyp "Symboltabelle" zu spezifizieren? Zuvor bemerken wir, daß neben der Initialisierungs-Operation INIT die Operationen ENTERBLOCK und ADDID eine konstruktive (d.h. aufbauende) Wirkung haben.

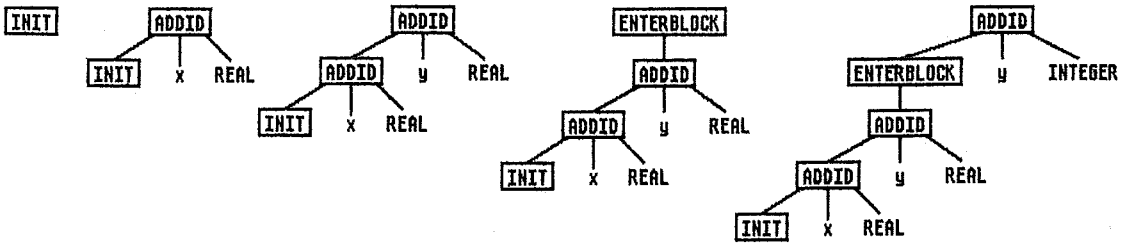
```
type Symboltabelle;
syntax
        INIT -> Symboltabelle;
        ENTERBLOCK (Symboltabelle) -> Symboltabelle;
        ADDID (Symboltabelle, Identifier, Attributliste) -> Symboltabelle;
        LEAVEBLOCK (Symboltabelle) -> Symboltabelle;
```

```
        ISINBLOCK (Symboltabelle, Identifier) -> Boolean;
        RETRIEVE (Symboltabelle, Identifier) -> Attributliste ∪ {UNDEF};
    semantics
      declare  syta: Symboltabelle; id,id1: Identifier; atl: Attributliste;
        LEAVEBLOCK (INIT) = INIT;
        LEAVEBLOCK (ENTERBLOCK (syta)) = syta;
        LEAVEBLOCK (ADDID (syta, id, atl)) = LEAVEBLOCK (syta);
        ISINBLOCK (INIT, id) = FALSE;
        ISINBLOCK (ENTERBLOCK (syta), id) = FALSE;
        ISINBLOCK (ADDID (syta, id, atl), id1) = IF (id1=id) THEN
                                                   TRUE
                                                 ELSE
                                                   ISINBLOCK (syta, id1);
        RETRIEVE (INIT, id) = UNDEF;
        RETRIEVE (ENTERBLOCK (syta), id) = RETRIEVE (syta, id);
        RETRIEVE (ADDID (syta, id, atl), id1) = IF (id1=id) THEN
                                                  atl
                                                ELSE
                                                  RETRIEVE (syta, id1).
```

Der Leser mache sich klar, daß die hier geforderten Zusammenhänge zwischen den konstruktiven und den nicht-konstruktiven Operationen des ADT "Symboltabelle" unserem Verständnis der Bearbeitung eines blockstrukturierten Programmtextes sehr genau entsprechen. Zum Beispiel besagt das dritte LEAVEBLOCK-Axiom, daß wir bei Verlassen eines Blockes alle in jenem Block deklarierten Variablen-Identifier "vergessen". Die RETRIEVE-Axiome zusammen besagen: Wenn wir einen Identifier nicht innerhalb eines bestimmten Blocks finden, so müssen wir ihn im jeweils äußeren Block suchen, und wenn es ihn auch im äußersten Block nicht gibt, so ist er niemals deklariert worden.

Diese Argumentation läßt sich noch besser verstehen, wenn wir die Axiome eines ADTs als Vorschriften zur Manipulation der durch die konstruktiven Operationen aufgebauten Objekte interpretieren. Tatsächlich liefern diese Operationen die Bausteine für eine *Standard-Repräsentation* der Objekte eines ADTs als Bäume. In unserem Beispiel sind dies:

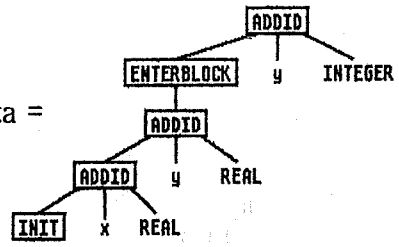

Verwendet der ADT-Modul "SymbTab" diese Repräsentation, so werden die Symboltabellen, welche er im Auftrag eines Compilers erzeugt, der den weiter oben angegebenen blockstrukturierten Programmtext bis hin zur Registrierung der Deklaration im Block B1.1 bearbeitet, durch die folgenden Bäume dargestellt:




Bezüglich dieser Repräsentation sind die im semantics-Teil der ADT-Definition aufgelisteten Axiome nichts anderes als Programme (im Sinne der oben diskutierten abstrakten Implementierung) zur Ausführung der nicht-konstruktiven Operationen des ADTs. Mit anderen Worten: Die ADT-Axiome geben an, wie Standard-Repräsentationen zu manipulieren sind. Wir zeigen dies am Beispiel der RETRIEVE-Axiome:

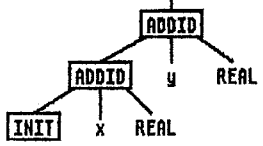
Angenommen, der Compiler möchte innerhalb des Blockes B1.1 wissen, welche Attribute der Variablen mit dem Identifier "x" zugeordnet sind. Die Standard-Repräsentation der in diesem Block aktuellen Symboltabelle ist:

syta =



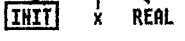

Die Ausführung der RETRIEVE-Operation ergibt:

RETRIEVE(syta, x) = RETRIEVE( ENTERBLOCK , x) (* gemäß drittem RETRIEVE-Axiom *)




= RETRIEVE( ADDID , x) (* gemäß zweitem RETRIEVE-Axiom *)


= RETRIEVE( ADDID , x) = REAL (* beides gemäß drittem RETRIEVE-Axiom *)




(Zur Übung versuche sich der Leser in analoger Weise an einer Standard-Repräsentation der Stack-Objekte und an der entsprechenden Ausführung der Stack-Axiome als Programme.)

Über die Standard-Repräsentation der Objekte eines ADTs als Bäume (mit den konstruktiven Operationen als Knoten) gewinnt man also sozusagen eine *Standard-Implementierung* des ADTs. Es sollte jedoch klar sein, daß diese nur eine andere Interpretation der Spezifikation des ADTs ist, und daß es daher keinen Sinn macht, von ihrer Korrektheit (relativ zu irgendeiner Spezifikation) zu sprechen. Sie ist per definitionem korrekt. Wenn man so will, kann man sie als *Referenz-Implementierung* bezeichnen, an der jede andere Implementierung (per Korrektheitsbeweis!) "zu messen" ist.

Dennoch: Angesichts der Möglichkeit einer *Standard-Implementierung* eines ADTs wird man sich natürlich fragen, warum wir überhaupt nach anderen Implementierungen (s.o.) suchen und uns damit der Mühe eines Korrektheitsbeweises unterziehen. Die Antwort ist nicht schwer: Standard-Implementierungen verlangen die Manipulation von Bäumen, über deren Struktur sie selbst keine Kontrolle haben, und die daher sehr aufwendig sein kann. Es sind also im wesentlichen Effizienzgründe, die uns diese konzeptuell einfache Lösung im allgemeinen verwerfen lassen. Um eine effizientere (abstrakte) Implementierung des ADTs "SymbTab" wollen wir uns im folgenden bemühen.

Die dieser Implementierung zugrundeliegende Idee ist freilich auch einfach. Sie beruht auf zwei Beobachtungen:

(i) Eine Symboltabelle enthält Abbildungen, welche Objekten vom Typ "Identifier" solche vom Typ "Attributliste" zuordnen.

(ii) Die im Verlauf der Bearbeitung eines Programmtextes jeweils aktuelle Symboltabelle enthält genau jene Abbildungen, welche dem aktuellen Block und den diesen umfassenden Blöcken entsprechen. Und beim Verlassen des aktuellen Blocks wird die ihm entsprechende Abbildung aus der Symboltabelle entfernt. Wird ein neuer Block betreten, so wird eine ihm entsprechende Abbildung zur Symboltabelle hinzugefügt.

Die in (ii) beschriebene "Dynamik" einer Symboltabelle kommt uns bekannt vor. In der Tat: es ist das gleiche Verhalten, welches ein Stack (von Objekten beliebigen Typs) aufweist, und das wir oben, unserem intuitiven Verständnis folgend, durch den ADT "Stack" formalisiert haben. Die (lineare) Wanderung durch den blockstrukturierten Programmtext läßt sich ohne weiteres eins-zu-eins auf eine Abfolge von Stack-Operationen (PUSH bei BLOCKANF, POP bei BLOCKEND) übertragen. Und was da "gePUSHt" und "gePOPt" werden sollte, ist nach (i) auch klar: Es sind Abbildungen, Objekte von einem Typ freilich, den wir bisher noch nicht spezifiziert haben.

Wir begnügen uns mit Minimalforderungen. Von einer "Abbildung" ("Mapping") verlangen wir, daß sie eine eindeutige Zuordnung herstellt zwischen den Elementen einer Urbildmenge ("domaintype") und den Elementen einer Bildmenge ("rangetype"), und daß man sie (wie zuvor bei "Stack" und "Array") zunächst einmal überhaupt anlegen (initialisieren) kann (Operation NEWMAP). Zweitens

sollte man sie verändern können, indem einem Element der Urbildmenge ein (anderes) Element der Bildmenge zugeordnet oder indem eine bestehende Zuordnung aufgehoben wird (Operation DEFMAP). Drittens wollen wir von jedem Element der Urbildmenge wissen, ob diesem ein Element der Bildmenge zugeordnet ist (Operation ISDEFINED), und viertens - falls ja - wollen wir das zugeordnete Element auch haben (Operation EVMAP). Danach können wir den ADT "Mapping" folgendermaßen definieren:

```
type    Mapping[domaintype:Type, rangetype:Type];
syntax
        NEWMAP -> Mapping;
        DEFMAP(Mapping, domaintype, rangetype) -> Mapping;
        ISDEFINED (Mapping, domaintype) -> Boolean;
        EVMAP(Mapping, domaintype) -> rangetype ∪ {UNDEF};
semantics
  declare  map:Mapping; dv, dv1: domaintype; rv: rangetype;
        ISDEFINED (NEWMAP, dv) = FALSE;
        ISDEFINED (DEFMAP(map, dv, rv), dv1) = IF (dv1 = dv) THEN
                                                  TRUE
                                               ELSE
                                                  ISDEFINED(map, dv1);
        EVMAP(NEWMAP, dv) = UNDEF;
        EVMAP(DEFMAP(map, dv, rv), dv1) = IF (dv1 = dv) THEN
                                             rv
                                          ELSE
                                             EVMAP(map, dv1).
```

Damit haben wir alles, was wir für die abstrakte Implementierung des ADTs "Symboltabelle" als Stack von Abbildungen (von einem domaintype "Identifier" in einen rangetype "Attributliste") benötigen:

```
representation
        SYMT(Stack [Mapping[Identifier, Attributliste]]) -> Symboltabelle;
programs
  declare  stk: Stack; id: Identifier; atl: Attributliste;
        INIT = SYMT(PUSH(NEWSTACK, NEWMAP));
        ENTERBLOCK(SYMT(stk)) = SYMT(PUSH(stk, NEWMAP));
        ADDID(SYMT(stk), id, atl) =
                    SYMT(REPLACE(stk, DEFMAP(TOP(stk), id, atl);
        LEAVEBLOCK(SYMT(stk)) = IF ISNEW(POP(stk)) THEN
                                   SYMT(REPLACE(stk, NEWMAP))
                                ELSE
                                   SYMT(POP(stk));
        ISINBLOCK(SYMT(stk), id) = ISDEFINED(TOP(stk), id);
```

```
RETRIEVE (SYMT (stk), id) =
                              IF ISNEW (stk) THEN
                                UNDEF
                              ELSE IF ISDEFINED (TOP (stk), id) THEN
                                EVMAP (TOP (stk), id)
                              ELSE
                                RETRIEVE (SYMT (POP (stk)), id).
```

Als einzigen Kommentar zu dieser Implementierung schließen wir hier die Bemerkung an, daß sich im Rahmen des Beispiels "Symboltabelle" nun offenbar die Weisheit unserer früheren Entscheidung herausstellt, die Stack-Operation REPLACE (als Abkürzung für die Hintereinanderausführung von POP und PUSH) zu definieren. Im übrigen wollen wir sogleich an die Arbeit gehen und die Erfülltheit von zweien der Axiome des ADTs "Symboltabelle" beweisen, des ersten LEAVEBLOCK-Axioms nämlich und des dritten RETRIEVE-Axioms (die Begründungen für die jeweils vorgenommenen Umformungen vermerken wir als Kommentare):

```
LEAVEBLOCK (INIT)
    = LEAVEBLOCK (SYMT (PUSH (NEWSTACK, NEWMAP)))
      (* INIT-Programm der Implementierung von "Symboltabelle" *)
    = IF ISNEW (POP (PUSH (NEWSTACK, NEWMAP))) THEN
        SYMT (REPLACE (PUSH (NEWSTACK, NEWMAP), NEWMAP))
      ELSE ...
      (* LEAVEBLOCK-Programm *)
    = IF ISNEW (NEWSTACK) THEN
        SYMT (REPLACE (PUSH (NEWSTACK, NEWMAP), NEWMAP))
      ELSE ...
      (* POP-Axiom des ADTs "Stack" *)
    = IF TRUE THEN
        SYMT (REPLACE (PUSH (NEWSTACK, NEWMAP), NEWMAP))
      ELSE ...
      (* ISNEW-Axiom des ADTs "Stack" *)
    = SYMT (REPLACE (PUSH (NEWSTACK, NEWMAP), NEWMAP))
      (* IF ... THEN ... ELSE ... - Eigenschaft *)
    = SYMT (PUSH (POP (PUSH (NEWSTACK, NEWMAP)), NEWMAP))
      (* REPLACE-Axiom des ADTs "Stack" *)
    = SYMT (PUSH (NEWSTACK, NEWMAP))
      (* POP-Axiom des ADTs "Stack" *)
    = INIT
      (* INIT-Programm der Implementierung von "Symboltabelle" *)
```

Wie zu beweisen war! Der Nachweis des dritten RETRIEVE-Axioms wird - mit solch minutiöser Angabe jedes Schritts - noch aufwendiger, und der Leser wird es uns nachsehen, wenn wir ihm die Aufgabe überlassen, die Erfülltheit der übrigen sieben Axiome des ADTs "Symboltabelle" zu überprüfen.

In der Tat: Bevor wir uns mit dem bewußten RETRIEVE-Axiom befassen, müssen wir zunächst jene Bemerkungen, die wir weiter oben über die hier benutzte Spezifikationssprache gemacht haben, etwas genauer beim Wort nehmen. Diese Bemerkungen implizierten insbesondere das Vorhandensein eines (prädefinierten) ADTs "Boolean". Die "IF...THEN...ELSE...-Eigenschaft", von der wir im Beweis von LEAVEBLOCK(INIT) = INIT Gebrauch gemacht haben, ist nichts anderes als ein Axiom dieses ADTs, dessen Spezifikation wie folgt notiert werden kann:

```
type     Boolean;
syntax
         TRUE --> Boolean;
         FALSE --> Boolean;
         IF-THEN-ELSE (Boolean, Boolean,Boolean) --> Boolean;
semantics
  declare  p, q, r: Boolean;
         IF-THEN-ELSE(TRUE, q, r) = q;
         IF-THEN-ELSE(FALSE, q, r) = r.
```

Mit den bekannten Axiomen der Boole'schen Algebra im Hinterkopf hätten wir dem syntaktischen Teil dieser Spezifikation natürlich noch zum Beispiel die Operationen

```
AND(Boolean, Boolean) --> Boolean;
OR(Boolean, Boolean) --> Boolean;
NOT(Boolean) --> Boolean;
IMPL(Boolean, Boolean) --> Boolean;
EQUIV(Boolean, Boolean) --> Boolean
```

hinzufügen können, mit einer durch IF-THEN-ELSE definierten Semantik:

```
AND(p, q) = IF-THEN-ELSE(p, q, FALSE);
OR(p,q) = IF-THEN-ELSE(p, TRUE, q);
NOT(p) = IF-THEN-ELSE(p, FALSE, TRUE);
IMPL(p,q) = IF-THEN-ELSE(p, q, TRUE);
EQUIV(p,q) = IF-THEN-ELSE(p, q, NOT(q)).
```

(Das ist nichts anderes als die bei der Definition der Semantik von REPLACE angewandte Methode! Zur Übung zeige man, daß man mit diesen Definitionen den bekannten Axiomen der Boole'schen Algebra genügt.)

Zum Beweis des dritten RETRIEVE-Axioms (inzwischen sollte sich auch der mathematisch weniger versierte Leser daran gewöhnt haben, daß man Axiome

manchmal beweisen muß!) werden wir nun einige Eigenschaften der IF-THEN-ELSE Operation benötigen, die sich ohne weiteres aus den Axiomen des "Boolean"-ADTs ergeben. Wir notieren sie wieder in der vertrauteren "IF...THEN... ELSE..."-Schreibweise. Seien p, q und r vom Typ Boolean:

(i) (IF p THEN q ELSE q) = q

(ii) (IF p THEN TRUE ELSE FALSE) = p

(iii) (IF (IF p THEN q ELSE r) THEN a ELSE b) =
(IF p THEN (IF q THEN a ELSE b) ELSE (IF r THEN a ELSE b))

(iv) (IF p THEN q[p] ELSE r) = (IF p THEN q[p –> TRUE] ELSE r)
(IF p THEN q ELSE r[p]) = (IF p THEN q ELSE r[p –> FALSE])

q[p] bezeichnet hier einen Ausdruck, in dem p Teilausdruck ist. "p –> TRUE" bezeichnet die Substitution von p durch die Konstante TRUE. Ist T irgendein ADT, so läßt sich die Liste der T-Operationen übrigens immer um eine Operation "IF-THEN-ELSE (Boolean, T, T) –> T" erweitern, welche die den beiden obigen Boolean-Axiomen entsprechende Semantik erhält. Die Eigenschaften (i) und (iv) gelten dann auch für die so verallgemeinerte IF-THEN-ELSE-Operation.

Für beliebige Repräsentanten SYMT (stk) eines "Symboltabelle"-Objektes lautet das dritte RETRIEVE Axiom:

```
RETRIEVE (ADDID (SYMT (stk), id, atl), id1) = IF (id = id1) THEN atl
                                              ELSE
                                                RETRIEVE (SYMT (stk), id1).
```

Dies ist nun unsere Behauptung. Ihr Beweis ist geleistet, wenn wir zeigen können, daß sich beide Seiten dieser Gleichung mit Hilfe der Programme der Implementierung und mit den Axiomen der ADTn "Stack", "Mapping" und "Boolean" (und eventuell aus ihnen abgeleiteten Eigenschaften) auf den gleichen Ausdruck reduzieren lassen. Zu diesem Zweck attackieren wir zunächst die linke Seite (LS), und wie üblich notieren wir die Begründung jeder Umformung als Kommentar:

```
LS  = RETRIEVE (ADDID (SYMT (stk), id, atl), id1)
    = RETRIEVE (SYMT (stk1), id1)
      (* ADDID-Programm; "stk1" ist hierin als Abkürzung von
      REPLACE (stk, DEFMAP (TOP (stk), id, atl)) gesetzt *)
    = IF ISNEW (stk1) THEN
        UNDEF
      ELSE IF ISDEFINED (TOP (stk1), id1) THEN
        EVMAP (TOP (stk1), id1)
      ELSE
        RETRIEVE (SYMT (POP (stk1)), id1)
      (* RETRIEVE-Programm *)
```

```
= IF ISDEFINED (DEFMAP (TOP (stk), id, atl), id1) THEN
    EVMAP (DEFMAP (TOP (stk), id, atl), id1)
  ELSE
    RETRIEVE (SYMT (POP (stk)), id1)
  (* wegen TOP (stk1)  =  TOP (REPLACE (stk, DEFMAP (...)))
                       =  TOP (PUSH (POP (stk), DEFMAP (...)))
                       =  DEFMAP (TOP (stk), id, atl),
  wegen POP (stk1) = POP (PUSH (POP (stk), DEFMAP (...))) = POP (stk)
  und wegen ISNEW (stk1) = FALSE *)

= IF    (IF (id = id1) THEN
           TRUE
         ELSE
           ISDEFINED (TOP (stk), id1))  THEN
    EVMAP (DEFMAP (TOP (stk), id, atl), id1)
  ELSE
    RETRIEVE (SYMT (POP (stk), id1)
  (* ISDEFINED-Axiom *)

= IF (id = id1) THEN
    IF TRUE THEN
      EVMAP (DEFMAP (TOP (stk), id, atl), id1)
    ELSE
      RETRIEVE (SYMT (POP (stk), id1)
  ELSE
    IF ISDEFINED (TOP (stk), id1) THEN
      EVMAP (DEFMAP (TOP (stk), id, atl), id1)
    ELSE
      RETRIEVE (SYMT (POP (stk)), id1)
  (* IF...THEN...ELSE...-Eigenschaft (iii) *)

= IF (id = id1) THEN
    EVMAP (DEFMAP (TOP (stk), id, atl), id1)
  ELSE IF ISDEFINED (TOP (stk), id1) THEN
    EVMAP (DEFMAP (TOP (stk), id, atl), id1)
  ELSE
    RETRIEVE (SYMT (POP (stk)), id1)
  (* erstes IF...THEN...ELSE...-Axiom *)

= IF (id = id1) THEN
    IF (id =id1) THEN
      atl
    ELSE
      EVMAP (TOP (stk), id1)
```

```
        ELSE IF ISDEFINED (TOP (stk), id1) THEN
           IF (id =id1) THEN
              atl
           ELSE
              EVMAP (TOP (stk), id1)
        ELSE
           RETRIEVE (SYMT (POP (stk)), id1)
        (* EVMAP-Axiom *)

    = IF (id = id1) THEN
           IF TRUE THEN
              atl
           ELSE
              EVMAP (TOP (stk), id1)
        ELSE IF ISDEFINED (TOP (stk), id1) THEN
           IF FALSE THEN
              atl
           ELSE
              EVMAP (TOP (stk), id1)
        ELSE
           RETRIEVE (SYMT (POP (stk)), id1)
        (* IF...THEN...ELSE...-Eigenschaft (iv) für den Typ "Attributliste" *)

    = IF (id = id1) THEN
           atl
        ELSE IF ISDEFINED (TOP (stk), id1) THEN
           EVMAP (TOP (stk), id1)
        ELSE
           RETRIEVE (SYMT (POP (stk)), id1)
        (* IF...THEN...ELSE...-Axiome *)
```

Damit ist schließlich die linke Seite der behaupteten Gleichung auf einen einigermaßen übersichtlichen Ausdruck reduziert, in welchem zwar immer noch die Anwendung des RETRIEVE-Programms notiert ist, doch nun mit einer durch einen "kleineren" Stack (nämlich POP(stk)) repräsentierten (und entsprechend "kleineren") Symboltabelle als Argument. Wir könnten so fortfahren und den rekursiven "Aufruf" des RETRIEVE-Programms bis zum "Boden" des Stacks auflösen. Doch wollen wir zunächst sehen, was uns die Umformung der rechten Seite der behaupteten Gleichung mit Hilfe des RETRIEVE-Programms bringt:

```
RS  = IF (id =id1) THEN
         atl
      ELSE
         RETRIEVE (SYMT (stk), id1)
```

```
= IF (id = id1) THEN
    atl
  ELSE IF ISNEW(stk) THEN
    UNDEF
  ELSE IF ISDEFINED(TOP(stk), id1) THEN
    EVMAP(TOP(stk), id1)
  ELSE
    RETRIEVE(SYMT(POP(stk)), id1)
```

Wäre nicht der durch die Möglichkeit "ISNEW(stk) = TRUE" gegebene Zweig dieses IF...THEN...ELSE (IF...) - Ausdrucks, so hätten wir unser Ziel erreicht. Wenn wir nun argumentieren könnten, daß dieser Fall gar nicht eintreten kann, daß also immer gilt "ISNEW(stk) = FALSE", so würde uns die Anwendung des zweiten IF...THEN...ELSE-Axioms in der Tat den gleichen Ausdruck liefern, den wir durch die Umformungen der linken Seite erhalten haben.

Wir besinnen uns an dieser Stelle darauf, daß wir es hier ja nicht mit irgendwelchen Stacks zu tun haben, sondern immer mit solchen, welche Symboltabellen repräsentieren. Diese Stacks werden durch einige der Programme der Implementierung des ADTs "Symboltabelle" erzeugt, nämlich durch INIT, ENTERBLOCK, ADDID und LEAVEBLOCK. Das gesuchte Argument ist daher nichts anderes als eine Induktion über die Erzeugung von Symboltabellen-Repräsentationen, mit der "Induktionsverankerung" durch die Operation INIT:

(1) Sei stk derart, daß INIT = SYMT(stk);
dann gilt: stk = PUSH(NEWSTACK, NEWMAP)
und folglich (nach zweitem ISNEW-Axiom): ISNEW(stk) = FALSE.

Zur Vervollständigung der Induktion müssen wir zeigen, daß die übrigen Operationen, angewandt auf durch Stacks repräsentierte Symboltabellen, diese Eigenschaft nicht zerstören. Für die folgenden Beweisschritte nehmen wir generell an, daß "stk" und "stk1" Stacks seien, welche Symboltabellen "syta" und "syta1" repräsentieren, für die also gilt: syta = SYMT(stk) und syta1 = SYMT(stk1).

(2) Sei stk1 zusätzlich zur generellen Voraussetzung derart, daß
ENTERBLOCK(SYMT(stk)) = SYMT(stk1);
dann gilt: stk1 = PUSH(stk, NEWMAP)
und nach zweitem ISNEW-Axiom: ISNEW(stk1) = FALSE.

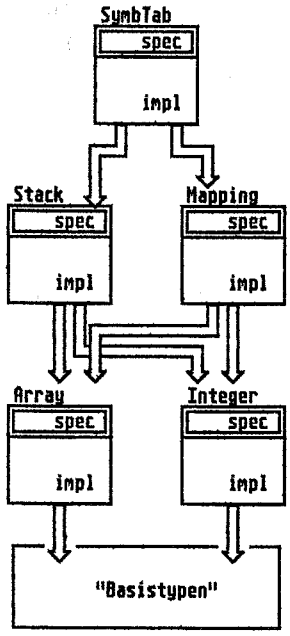
(3) Sei stk1 zusätzlich zur generellen Voraussetzung derart, daß
ADDID(SYMT(stk), id, atl) = SYMT(stk1);
dann gilt: stk1 = REPLACE(stk, DEFMAP(...))
= PUSH(POP(stk), DEFMAP(...))
und nach zweitem ISNEW-Axiom: ISNEW(stk1) = FALSE.

(4) Sei stk1 zusätzlich zur generellen Voraussetzung derart, daß
LEAVEBLOCK(SYMT(stk)) = SYMT(stk1);

dann gilt: stk1 = REPLACE(stk, NEWMAP), falls ISNEW(POP(stk))
und stk1 = POP(stk);
im ersten Fall ist wieder nach zweitem ISNEW-Axiom
ISNEW(stk1) = FALSE, und im zweiten Fall gilt dies trivialerweise.

Man beachte, daß wir für (2), (3) und (4) "ISNEW(stk) = FALSE" nicht voraussetzen mußten.

Wie gesagt, der Leser ist eingeladen, nun auch die Gültigkeit der übrigen sieben Axiome des ADTs "Symboltabelle" aufgrund unserer Implementierung mittels "Stack" und "Mapping" zu verifizieren. Ferner möge er sich überlegen, wie der ADT "Mapping" (abstrakt) implementiert werden könnte. Eine naheliegende Möglichkeit ist sicher durch den sehr ähnlich ausschauenden Datentyp "Array" gegeben. In [GHM] wird zum Beispiel eine Implementierung vorgeschlagen, welche "Mapping" mit Hilfe von "Array" und "Integer" als sogenannte Hash-Tabelle (vgl. etwa [OTW]) realisiert. Zusammen mit der schon bekannten Implementierung des ADTs "Stack" ergibt sich so eine Hierarchie von Implementierungsebenen ("Schichten") mit dem ADT "Symboltabelle" an der Spitze. Jede Implementierung innerhalb dieser Hierarchie macht nur von den Spezifikationen jeweils tieferliegender ADTn Gebrauch, so wie wir dies im vorangegangenen Abschnitt über "Modularisierung" gefordert haben. Auf der untersten Ebene liegen schließlich Typen wie "Register", "Speicherzellen" und "Adressen" (in der Abbildung unter dem Sammelbegriff "Basistypen" vereint), welche durch die Hardware des Rechners repräsentiert werden.




Wenn wir nochmals die in diesem Abschnitt geführten Beweise Revue passieren lassen, so fällt auf, daß sie in jedem Fall auf ziemlich mechanischen Manipulationen von Ausdrücken einer formalen Sprache beruhten. Die Vermutung liegt daher nahe, daß sich diese Beweise bis zu einem gewissen Grade durchaus automatisieren lassen. Für jeden Beweisschritt haben wir die Auswahl aus einem begrenzten Vorrat möglicher Ersetzungen ("*Substitutionen*"), und es ist nicht schwer, sich ein (Software-)Werkzeug vorzustellen, welches einem Implementierer, dem ja letztendlich die Last des Beweises der Korrektheit seiner Arbeit obliegt, diesen Vorrat anbietet. Man kann sich ferner vorstellen, daß ein derartiges Werkzeug die Ersetzungen selbständig vornimmt, um so eine Behauptung - sollte sie denn wahr sein - auf den Wahrheitswert TRUE zu *reduzieren*. Dabei wird es - ganz so wie die im dritten Kapitel besprochenen Programme zur Lösung des Problems der acht Damen und zur Suche eines Auswegs aus einem Labyrinth - nach der Methode des "Versuchs und Irrtums" (*Backtracking*) vor-

gehen. In der Tat sind solche Werkzeuge von mehr oder minder großer Mächtigkeit für Formalismen der hier betrachteten Art entwickelt worden. Leider ist in diesem Buch nicht der Platz, um auf diese Arbeiten im Einzelnen einzugehen (vgl. jedoch [ILL]). Wir müssen auch, wie weiter oben schon angedeutet, darauf verzichten, das Bewußtsein dafür zu untermauern, daß dem "automatischen Beweisen" Grenzen gesetzt sind; Grenzen, die beim Umgang mit hinreichend ausdrucksfähigen formalen Kalkülen prinzipiell nicht überschritten werden können (vgl. hierzu z.B. [HER] und [SMU]).

Diese Grenzen sind im übrigen auch für Überlegungen relevant, die sich u.a auf die Frage nach der *Konsistenz* (Widerspruchsfreiheit) der Definition eines ADTs beziehen, also letztlich auf die Frage, ob die durch den ADT beschriebenen Objekte überhaupt existieren können. Wir stehen hier vor dem gleichen Problem, das ein Mathematiker hat, der für eine axiomatisch definierte algebraische Struktur nachweisen soll, daß in ihr nicht sowohl eine Aussage als auch ihre Negation ableitbar sind.

### 6.4.4 Objekte, Klassen und Vererbung

Wir haben - im wesentlichen ohne weitere begriffliche Erläuterungen der Mühe wert zu halten - das Wort *Objekt* in diesem Buch in verschiedenen Zusammenhängen (zuletzt im vorangegangenen Abschnitt) mit einer gewissen Freizügigkeit und Nonchalance zur Sprache gebracht. Andererseits ist zu konstatieren, daß dieses Wort in allerlei Kombinationen (wie *Objektorientierte Analyse - OOA, Objektorientierter Entwurf - OOD ("D"* für *Design), Objektorientierte Programmierung - OOP, Objektorientierte Programmiersprachen - OOPL*) seit Beginn der achtziger Jahre in mehr oder weniger werbewirksamer Weise und mit zunehmender Lautstärke auf dem Markt der Software-Ideen, -Produkte und -Meinungen ausgerufen wird. (Es ist dabei interessant zu beobachten, mit welcher Macht ein in Mode gekommenes Wort manchen Trittbrettfahrer anzieht, der alten oder gar nicht so recht passen wollenden Hüten das plötzlich so erfolgreiche Etikett anheftet.) So drängt es sich auf zu fragen, ob hinter dem Wort- und Werbeschleier tatsächlich stringente Konzepte stecken, und wenn ja, ob sich unser recht vager und salopper bisheriger Sprachgebrauch zumindest partiell durch sie rechtfertigen läßt.

Wir wollen den Leser über den wesentlichen Inhalt unserer Antwort nicht lange im Unklaren lassen: Ja, es stecken klare Konzepte dahinter (wenn auch unter den Protagonisten nicht immer Einigkeit hinsichtlich ihrer Bedeutung und ihres Gewichts bestehen mag!), und unsere bisherige Verwendung des Wortes "Objekt" (insbesondere bezogen auf Modularisierung und Abstrakte Datentypen) hat durchaus etwas mit ihnen zu tun. Unserer Erörterung von Aspekten der "Programmierung im Großen" würden daher wesentliche Elemente fehlen, wenn wir die *Objektorientierung* nicht vom Geruch des Schlagwortes befreiten und sie

nicht in den Kreis der für die Softwareentwicklung wichtigsten Methoden und Techniken aufnähmen. Sie ist nämlich aufs engste mit jenen Einsichten verbunden, welche während der siebziger Jahre in Prinzipien der Modularisierung und in die Theorie der Abstrakten Datentypen gewonnen wurden. *Objektorientierte Programmierung* bezeichnet nichts anderes als einen (gelegentlich mit vielleicht etwas zu viel Optimismus vorgetragenen) Ansatz zur Strukturierung und Implementierung von Softwaresystemen, der diese Einsichten für die Praxis des Software-Engineering fruchtbar machen soll, und das heißt insbesondere für die Erreichung der in Abschnitt 6.2 diskutierten Qualitäten von Software. Von besonderem Interesse sind dabei die Aspekte *Wiederverwendbarkeit* (vgl. Abschnitt 6.2.2) und *Adaptierbarkeit* (vgl. Abschnitt 6.2.3).

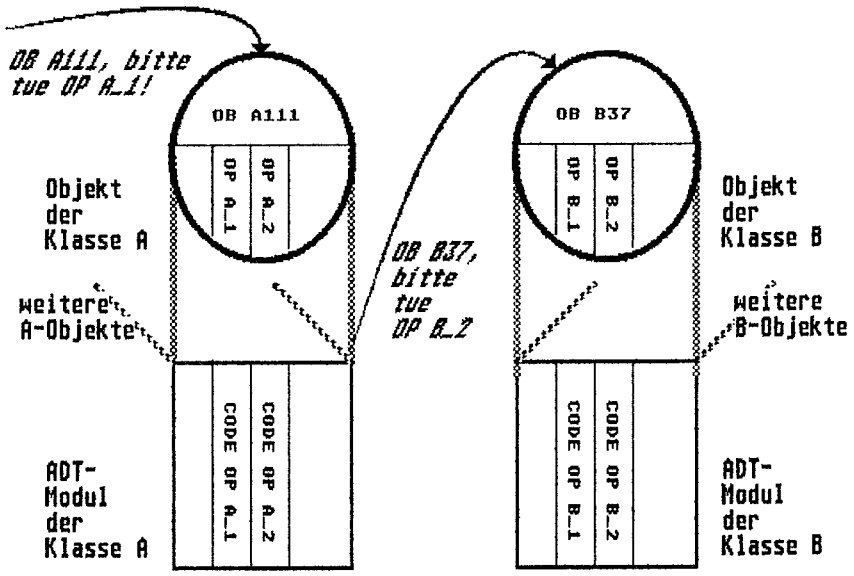
Die Grundlage aller objektorientierten Ansätze ist - wie sollte es anders sein - das Verständnis des Geschehens in Realwelt- und Softwaresystemen als *Interaktion von Objekten.* (Vgl. dazu auch die Diskussion in Abschnitt 6.3.1. Dort sprachen wir freilich von Prozessen und Objekten. Weiter unten werden wir sehen, daß man mit dem Objektbegriff durchaus auskommen kann.) Über Objekte können wir allerdings beim besten Willen zunächst nicht mehr sagen, als daß sie durch irgendwelche *Eigenschaften* definiert sind. Damit meinen wir Eigenschaften im weitesten Sinne. Wenn wir zum Beispiel von einem bestimmten Menschen sprechen, so können wir uns dabei sowohl auf seine Haarfarbe, Körpergröße usw. beziehen, als auch auf seine Talente und Fähigkeiten, also nicht nur auf das, was ihn sichtbar auszeichnet, sondern auch auf das, was er weiß oder tun kann. Wir könnten dieser Unterscheidung durch Begriffspaare wie *passiv / aktiv* oder *statisch / dynamisch* Ausdruck geben. Doch wird dies eigentlich unnötig, wenn wir uns auf die (in den vorangegangenen Abschnitten vorgenommene) Klassifizierung der von einem Datenmodul angebotenen Operationen besinnen. Die für Benutzer relevanten Eigenschaften der von einem ADT-Modul verwalteten Objekte sind mit den Eigenschaften (den Wirkungen und den Bedingungen der Anwendung) jener Operationen identisch, welche mit diesen Objekten ausführbar sind. Informationen über andere Eigenschaften gibt es nicht! Man betrachte etwa den am Schluß von Abschnitt 6.4.2 definierten ADT-Modul "Punkt": Als Eigenschaften eines Punktes in der Ebene mögen dessen kartesische Koordinaten bezüglich eines gegebenen Achsenkreuzes gelten, aber auch seine "Fähigkeit", seinen Abstand von irgendeinem anderen Punkt mitzuteilen, oder seine Polarkoordinaten bezüglich des gegebenen Achsenkreuzes. Und diese Eigenschaften entsprechen umkehrbar eindeutig den für Objekte vom "Punkt-Typ" definierten Operationen! (Weitere Eigenschaften wären hinzuzufügen: etwa die Fähigkeit eines Punktes, sich per Streckung oder per Translation in eine andere Lage zu versetzen, usw.) Wie leicht (z.B. durch einen einzigen Zugriff auf ein in einer Record-Komponente gespeichertes Attribut) oder schwierig (durch langwierige Berechnungen) diese Operationen auszuführen sind, spielt dabei natürlich keine Rolle.

Diese Bemerkungen legen es nahe, *Objekte* generell als *Ausprägungen* (engl.: *instances*) Abstrakter Datentypen zu verstehen, und die Eigenschaften (engl.: *features*) eines Objektes als die mit ihm ausführbaren und in der Beschreibung seines Abstrakten Datentyps festgelegten Operationen. Etwas bildhafter gesprochen sind Objekte Individuen vergleichbar, die irgendwelchen *Klassen* zugehören und die "wissen", was sie - entsprechend ihrer Klassenzugehörigkeit - tun können oder was man mit ihnen tun kann. Objekte können ferner (wie menschliche Individuen!) einen inneren *Zustand* haben, und auch von diesem Zustand kann abhängen, was das Objekt jeweils kann oder was man mit ihm tun kann. Im laufenden Software-System entstehen und vergehen Objekte je nach Bedarf, und sie interagieren, indem sie sich untereinander zum Gebrauch ihrer Fähigkeiten aufrufen. Aus dieser Sicht ergibt sich nun in der Tat ein ziemlich radikaler Zugang zu Entwurf und Implementierung von Software-Systemen, nämlich allein durch die Beschreibung der Klassen (d. h. der Abstrakten Datentypen) jener Objekte (Individuen!), die in einem System zur Laufzeit agieren sollen. Nimmt man eine leichte Modifikation der in Abschnitt 6.3.1 vorgetragenen Darstellung eines Systems als Verbund kommunizierender Prozesse in Kauf, so bietet dieser Zugang den vielleicht direktesten Weg von der Systemspezifikation zur Implementierung. Die Modifikation besteht in einem Verzicht auf den Begriff "Prozeß" und dessen Ersetzung durch "Objekt". Diese Ersetzung ist möglich, weil wir Objekten erlauben, einen inneren Zustand zu besitzen, der sich - und das ist ja gerade das Charakteristikum von Prozessen - mit der Zeit ändern kann. Objekte mit inneren Zuständen können daher ohne weiteres auch für die Modellierung von Vorgängen herangezogen werden. Die "Buch"-Prozesse in Abschnitt 6.3.3 zum Beispiel lassen sich in diesem geänderten Sinne verstehen: Es sind Objekte (individuelle Bücher), mit den folgenden Eigenschaften:

- Man kann in jedem Fall ihren Autor, ihren Titel, ihr Erscheinungsjahr, ihren Erscheinungsort und ihre ISBN-Nummer feststellen;
- Sie haben einen Zustand, und je nach Zustand kann man sie bestellen, erwerben, katalogisieren, ausleihen, verlängern, zurückgeben, ausrangieren;
- man kann ihren Zustand feststellen, d.h. man kann sich darüber informieren, ob ein bestimmtes Buch bestellt wurde, ob es bezahlt ist, katalogisiert, ausgeliehen, usw.;
- in Abhängigkeit vom Zustand kann man weitere Informationen erhalten, etwa über die Klassifikation eines Buches und seinen Standort, an welchen Bibliotheksbenutzer es ausgeliehen wurde, usw.

Wir halten fest: Objektorientierter Entwurf ist darauf aus, die *Klassen* der für ein System als relevant erkannten *Objekte* zu definieren. Die Definition einer Klasse beschreibt die Eigenschaften (im oben erklärten Sinne) ihrer Objekte. Klassen können als Abstrakte Datentypen aufgefaßt werden. Für die Ausführung der klassenspezifischen Objekt-Operationen sind die den jeweiligen Klassen

entsprechenden ADT-Module zuständig. Wir können dabei Objekte grundsätzlich in einer aktiven (prozeßhaften) Rolle sehen und daher die Formulierung "sind zuständig" so auslegen, daß der für eine Operation benötigte Code im ADT-Modul vorhanden ist, und daß seine Ausführung immer von einem ganz bestimmten Objekt "verlangt" wird. Es macht also gar keinen Unterschied, wenn wir sagen, daß die in einem laufenden System tätigen Akteure die Objekte selbst sind, die - mittels der ADT-Module ihrer Klassen - Operationen ausführen. Und bei der Ausführung einer Operation können jeweils andere Objekte ihrerseits zur Ausführung von Operationen veranlaßt werden:




Ein zweiter Blick auf die ADT-Version des Moduls "Punktmenge" in Abschnitt 6.4.2 mag zum besseren Verständnis dieser Auffassung beitragen. Angenommen, ein den ADT-Modul "Punktmenge" benutzender ADT-Modul "Xyz" arbeitet mit der durch "VAR pm1: PunktmengenIdTyp" identifizierten Punktmenge und benötigt dabei den Punkt mit dem größten Abstand zum Koordinatenursprung. Dieser Punkt soll einem mit "VAR pMax: PunktTyp" deklarierten Identifier zugeordnet werden. ("PunktTyp" ist im ADT-Modul "Punkt" deklariert.) Im Text des ADT-Moduls "Xyz" findet sich also an geeigneter Stelle der Prozeduraufruf

"PMMaxPunkt (pm1, pMax, pm1Leer)".

(Natürlich ist "pm1Leer" - vom Typ BOOLEAN - ebenfalls im Text von "Xyz" deklariert.) Objektorientiert sehen wir diese Situation nun wie folgt:

- Ein Objekt (nennen wir es "Xyz17") der Klasse "Xyz" (d.h. eine Ausprägung des Abstrakten Datentyps "Xyz") führt eine Operation aus, bei der es ihrerseits ein Objekt (genannt "pm1") der Klasse "Punktmenge" manipuliert. (Die Eigenschaften der Objekte dieser Klasse sind im DEFINITION MODULE "Punktmenge" beschrieben.)

- Im Laufe dieser Manipulation möchte "Xyz17" von "pm1" wissen, welcher ihrer Punkte den größten Abstand zum Usprung des Koordinatensystems hat.
- Es schickt an das Objekt "pm1" die folgende Nachricht
  "Bitte führe die Operation PMMaxPunkt aus, ordne - falls du nicht leer bist - deinen Punkt mit maximalem Abstand dem Identifier pMax zu und setze pm1Leer auf FALSE; falls du leer bist, so setze pm1Leer auf TRUE!"
  (Dies ist die objektorientierte Interpretation des obigen Prozeduraufrufs. Man kann auch sagen, daß "Xyz17" eine Eigenschaft von "pm1" *benutzt*, nämlich seine Fähigkeit, die betreffende Operation auszuführen.)
- Ist "pm1" nicht leer und hat es den Punkt mit maximalem Abstand nicht schon als Teil seines Zustandes (!) "im Gedächtnis", so wird es nun, um der empfangenen Bitte nachzukommen, jeden seiner Punkte (das sind Objekte der Klasse "Punkt"!) nach seinem Abstand zum Koordinatenursprung befragen müssen.
- Jedes Objekt "p" (der Klasse "Punkt") von "pm1" wird also die Nachricht erhalten:
  "Bitte führe die Operation DistUrsprung aus!"
  "pm1" wird die jeweiligen Resultate dann zur Erfüllung der von "Xyz17" geäußerten Bitte verarbeiten.

So weit, so gut. Wir sehen, daß man ADT-Modulen, die in MODULA-2 implementiert sind, einen objektorientierten Anstrich zumindest in jenen Farben verpassen kann, die uns bis jetzt zur Verfügung stehen. Diese "Farben" heißen u. a.: *Objekt*, *Klasse*, *Eigenschaften*, gegenseitige *Benutzung* von Objekten. Und der Anstrich besteht im wesentlich in der Verwendung eines diese Begriffe aufnehmenden Sprachgebrauchs.

Die Frage ist nun freilich, inwieweit die bloße Verwendung dieser Begriffe und der ihnen zugrundeliegenden Anschauungs- und Denkweisen für die Entwicklung von Software-Systemen von größerem Nutzen sein kann als die bisher vorgetragenen Ideen zur Herstellung qualitativ hochwertiger Software. Schließlich haben wir weiter oben implizit behauptet, daß die objektorientierte Vorgehensweise insbesondere die Qualitäten *Wiederverwendbarkeit* und *Adaptierbarkeit* fördert. Wir erinnern uns: Unter *Wiederverwendbarkeit* verstehen wir die Integrierbarkeit existierender Software-Bausteine in Systemen, die verschieden sind von jenen, für die diese Bausteine ursprünglich verfaßt wurden. Und Software heißt gut *adaptierbar*, wenn kleine Änderungen der (externen) Spezifikation auch nur einen geringen Aufwand bei der Anpassung der Programme zur Folge haben. Weder für die eine noch für die andere Qualität scheint das bis jetzt vorgestellte objektorientierte Instrumentarium etwas besonderes zu leisten. Es ermöglicht nur, so jedenfalls sieht es vorläufig aus, einen anderen, vielleicht einfacheren und rationelleren Weg bei der Auffindung von Systemarchitekturen.

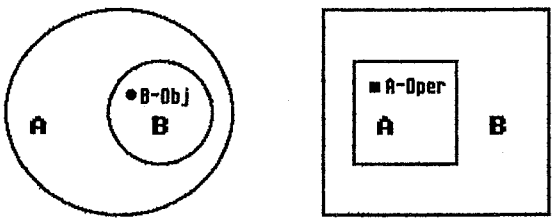
Die gegenüber herkömmlichen Vorgehensweisen entscheidenden Vorteile des objektorientierten Ansatzes werden durch das diesem Ansatz eigentümliche

Konzept der *Vererbung* (engl.: *inheritance*) ins Spiel gebracht:

> Existierende Klassen können die Eigenschaften ihrer Objekte auf die Objekte anderer, neu zu definierender Klassen *vererben.*

Die Mächtigkeit dieses Konzepts und seiner Weiterungen sowie deren Auswirkungen auf die Qualitäten *Wiederverwendbarkeit* und *Adaptierbarkeit* werden wir uns im verbleibenden Teil dieses Abschnitts bewußt machen. Um es für die Software-Entwicklung voll auszunutzen, bedarf es freilich geeigneter sprachlicher Ausdrucksmittel.

Immerhin, der Grundgedanke ist überaus einfach und naheliegend: Angenommen, Klasse A existiert, und Klasse B ist neu zu definieren. Wir wollen, daß die Objekte der Klasse B alle Eigenschaften haben, die auch den Objekten der Klasse A zukommen. Das heißt, alle Objekte der Klasse B sollen auch Objekte der Klasse A sein, nur eben etwas speziellere. Nach unserer Gleichsetzung von "Klasse" und "Abstrakter Datentyp", und nach unserer im vorangegangenen Abschnitt als hilfreich apostrophierten Vorstellung eines ADTs als "Menge von Objekten" ist Klasse B (die Menge aller B-Objekte) nichts anderes als eine Teilklasse von Klasse A (der Menge aller A-Objekte). (Aus dieser Sicht ist also die Klasse B "kleiner" als die Klasse A.)




Die Klassen A und B

als Mengen ... ... und als Module

Da nun Klassen in eineindeutiger Weise Modulen entsprechen, bedeutet die Vererbbarkeit klassenspezifischer Eigenschaften u. a., daß bei der Konstruktion neuer Module die Operationen bestehender Module übernommen werden dürfen. Der der Klasse B entsprechende Modul enthält also insbesondere alle Operationen des zur Klasse A gehörenden Moduls. Was ein A-Objekt kann, das kann auch ein B-Objekt, aber nicht umgekehrt. (Aus dieser Sicht erscheint die Klasse B sozusagen "größer" als die Klasse A.) Durch den Vererbungsbegriff wird eine Beziehung zwischen Modulen etabliert, die von der in den Abschnitten 6.4.2 und 6.4.3 ausschließlich betrachteten Beziehung zwischen dem Forderer und dem Erbringer von Dienstleistungen (der "A benutzt B" - Beziehung) grundverschieden ist.

Weder die älteren noch die meisten der moderneren Sprachen zur imperativen Programmierung (FORTRAN, COBOL, ALGOL, PASCAL, ADA, usw., und insbesondere auch MODULA-2) verfügen über die Mittel, mit denen diese, durch die Vererbung klassenspezifischer Eigenschaften gestiftete Beziehung zwischen Modulen für die Implementierung von Software-Systemen nutzbar gemacht werden kann. Der objektorientierte Ansatz, begründet auf Paradigmen, welche sich auf den Software-Entwurf beziehen, kommt erst durch das Vehikel einer ob-

jektorientierten Programmiersprache zu voller Geltung, also einer Sprache, die neben den "üblichen" Konstrukten zur Unterstützung von funktionaler Abstraktion (Prozeduren und Funktionsprozeduren) und Datenabstraktion mindestens auch die Darstellung von Vererbungsbeziehungen und die Verwendung damit zusammenhängender programmiertechnischer Mechanismen erlaubt. Von diesen Sprachen - auf einige haben wir schon an anderer Stelle (Abschnitt 6.4.1) hingewiesen - wählen wir B. Meyers EIFFEL, um die neben *Objekt* und *Klasse* wesentlichen Konzepte der objektorientierten Software-Entwicklung zu exemplifizieren. Diese Sprache zeichnet sich - jedenfalls im Urteil des Autors dieses Buches - durch besondere Klarheit, Einfachheit und Eleganz aus. (Leider sehen wir uns jedoch auch hier gezwungen darauf hinzuweisen, daß es uns - aus naheliegenden Gründen - im vorgegebenen Rahmen nicht möglich ist, diese Behauptung durch eine auch nur annähernd vollständige Darstellung von EIFFEL zu belegen. Doch mag sich der Leser durch die wenigen Aspekte, die wir in diesem Abschnitt aufgreifen, zur Lektüre der bereits zitierten Meyer'schen Monographie ([MEY]) angeregt fühlen.)

Das einzige in EIFFEL erlaubte Mittel zur Herstellung der Bausteine von Software-Systemen ist das (hier ohne die später noch zu ergänzenden Elemente dargestellte) Konstrukt:

```
class ‹Name der Klasse› export
   -- Liste der benutzbaren Eigenschaften
   -- der Objekte von ‹Name der Klasse›
features
   -- Beschreibung der Eigenschaften
   -- der Objekte von ‹Name der Klasse›
   -- (einschließlich der von Objekten
   -- anderer Klassen nicht benutzbaren)
end
```

("--" markiert einen Kommentar, Schlüsselworte werden unterstrichen und kursiv gedruckt.)

Eine *class* ist gleichzeitig Modul und Abstrakter Datentyp. Der Hauptteil des Textes einer *class* beschreibt die Eigenschaften (*features*) der Objekte von ‹Name der Klasse›. Diese *features* sind - aus dem Verständnis von *class* als Modul - nichts anderes als die Operationen (oder, mit anderen Worten, die Dienstleistungen), welche von den zu ‹Name der Klasse› gehörenden Objekte ausgeführt werden können. (Nach den Bemerkungen weiter oben werden wir dieser Sprechweise hier den Vorzug geben gegenüber der eher "konventionellen" Formulierung: "... welche mit den ... Objekten ausgeführt werden können".) *class* ist darüberhinaus auch das einzige Mittel zur Definition neuer Typen überhaupt. (Neben den vier *einfachen Typen* BOOLEAN, CHARACTER, INTEGER und REAL sowie einigen prädefinierten "Klassentypen" wie z.B. STRING. Ein

gesondertes Konstrukt *type*, wie in MODULA-2, PASCAL und ähnlichen Sprachen, gibt es nicht. Auch für die traditionellen Konstruktoren von Datenstrukturen wie *array* oder *record* gibt es in EIFFEL keine unmittelbaren Entsprechungen. Vielmehr werden diese, wie sich an unseren Beispielen noch zeigen wird, durch das *class* Konstrukt realisiert.)

Nicht alle *features* einer *class* müssen von Objekten anderer Klassen benutzbar sein. Diejenigen *features*, die anderen Klassen für die Definition der Eigenschaften ihrer Objekte zur Verfügung gestellt werden, sind in der sogenannten *export*-Liste aufgezählt.

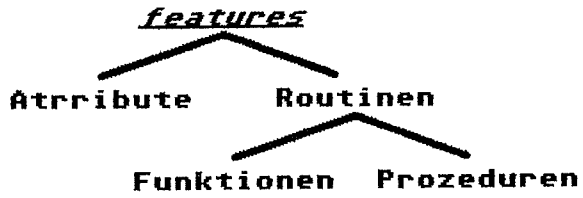
Die folgende *class* entspricht dem am Ende von Abschnitt 6.4.2 angegebenen (und um die Operation "verschiebe" erweiterten) ADT-Modul "Punkt":

```
class PUNKT export
   xkoord, ykoord, abstand, dist_ursprung, verschiebe
feature
   xkoord, ykoord: REAL;
   abstand (anderer_punkt: PUNKT): REAL is
      do
         Result := sqrt((xkoord - anderer_punkt.xkoord)^2 +
                        (ykoord - anderer_punkt.ykoord)^2)
      end;
   dist_ursprung: REAL is
      do
         Result := sqrt(xkoord^2 + ykoord^2)
      end;
   verschiebe (a, b: REAL) is
      -- verschiebe horizontal um a, vertikal um b Einheiten
      do
         xkoord := xkoord + a;
         ykoord := ykoord + b
      end;
end
```

Bevor wir uns eingehender mit der "objektorientierten Vererbungslehre" befassen können, ist es nötig, einige der für EIFFEL charakteristischen Notationen gut zu verstehen (und dies insbesondere im Lichte der Ausführungen von Abschnitt 6.4.1). Wir schließen daher an den Anschrieb der *class* "PUNKT" die folgenden Bemerkungen an:

(i) Die im *feature*-Abschnitt beschriebenen Eigenschaften der PUNKT-Objekte sind entweder durch Speicherinhalte (hier die Werte der *Attribute* "xkoord" und "ykoord", welche auch den *Zustand* eines PUNKT-Objekts bestimmen) oder durch *Berechnungsroutinen* gegeben. Diese Berechnungsroutinen sind entweder *funktional* oder *prozedural*. Die funktionalen Routinen (hier

"abstand" und "dist_ursprung") liefern - ohne den Zustand eines Objekts zu verändern (d.h. ohne Seiteneffekte!) - jeweils einen durch den Standardnamen "Result" bezeichneten Wert, während eine prozedurale Routine (hier "verschiebe") im allgemeinen den Zustand eines Objekts verändert.




(ii) Zur Illustration der *Benutzung* der im *export*-Abschnitt einer *class* aufgelisteten Eigenschaften durch andere Klassen möge das folgende Beispiel dienen:

```
class STRECKE export
   anfang, ende, laenge, neigung, verschiebe
feature
   anfang, ende: PUNKT;
   laenge: REAL is
      do
         Result := anfang.abstand (ende)
      end;
   neigung: REAL is
      do
         Result := (ende.ykoord - anfang.ykoord) /
                   (ende.xkoord - anfang.xkoord)
      end;
   verschiebe (a, b: REAL) is
      do
         anfang.verschiebe (a,b);
         ende.verschiebe (a,b)
      end;
end
```

Die Attribute "anfang" und "ende" bezeichnen jeweils ein Objekt der *class* PUNKT. Für die Berechnung der Eigenschaft "laenge" eines Objekts der *class* STRECKE wird hier die Eigenschaft "abstand" (von einem anderen Punkt) des PUNKT-Objekts "anfang" benutzt, und das Verschieben eines STRECKE-Objekts (horizontal um a, vertikal um b Einheiten) wird auf das Verschieben von Anfangs- und Endpunkt zurückgeführt. Die Notation

‹objekt_name›.‹feature_name› [‹parameter›]

entspricht der Aufforderung an ein Objekt, sich der durch ‹feature_name› bezeichneten - und eventuell durch Parameter präzisierten - Eigenschaft gemäß zu verhalten. (In manchen objektorientierten Sprachen, wie z.B. Smalltalk, ist

in diesem Zusammenhang sogar von "messages" - Botschaften - an Objekte die Rede.) Es ist bemerkenswert, daß alle Arten von *features*, Attribute und Routinen, bei ihrer Benutzung syntaktisch gleich behandelt werden. Der Grund hierfür wird uns bald einleuchten. Die Notation zur Anforderung von Diensten (also *features*) einer Klasse für eines ihrer Objekte macht es im übrigen möglich, ein und denselben *feature*-Namen (hier "verschiebe") in verschiedenen Klassen zu verwenden. Da "anfang" ein PUNKT-Objekt bezeichnet, sagt

"anfang.verschiebe(a,b)"

ganz deutlich, daß hier nur die "verschiebe"-feature der *class* PUNKT gemeint sein kann. Dies ist die EIFFEL eigene Form des sogenannten *overloading*. (Ähnliches erlaubt zum Beispiel die Programmiersprache ADA, in der man verschiedenen Prozeduren den gleichen Namen geben kann, sofern sich deren Parameterlisten typweise unterscheiden.)

(iii) Dem aufmerksamen Leser wird nicht entgangen sein, daß unsere EIFFEL Version des ADT-Moduls PUNKT kein der Operation "InitPunkt(...)" entsprechendes *feature* enthält. Dies ist nicht von ungefähr. Denn ohne daß es ausdrücklich in ihrem Text erwähnt werden muß, sieht jede *class* ("standardmäßig") für ihre Objekte ein *feature* namens *Create* vor. Tatsächlich bewirkt die Aufforderung

pkt1.Create

bezogen auf einen mit

pkt1: PUNKT

als (Attribut-)*feature* vom (Klassen-)Typ PUNKT deklarierten Identifier die Erzeugung eines Objekts der *class* PUNKT und dessen "Bindung" an den Identifier "pkt1". Die Attribut-*features* eines Objekts erhalten bei dessen Erzeugung - ebenfalls "standardmäßig" - bestimmte Werte, und zwar Attribute vom Typ

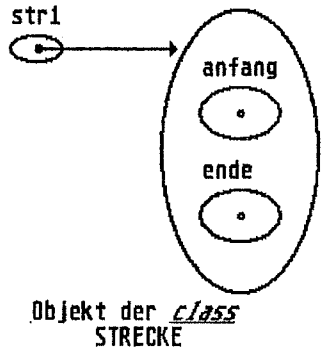
| | |
|---|---|
| BOOLEAN | false |
| INTEGER | 0 |
| REAL | 0.0 |
| CHARACTER | (ASCII) nul; |

Attribute (wie z.B. "anfang" in *class* STRECKE), deren Typ durch eine Klasse gegeben ist (also Attribute vom "Klassentyp") werden dagegen standardmäßig als "leere" Referenz initialisiert.

Es ist klar, daß von Attributen (vom Klassentyp), die leere Referenzen enthalten und somit (noch) kein Objekt bezeichnen, auch nicht die durch den jeweiligen Klassentyp definierten Operationen verlangt werden können. Angenommen, wir wenden die Standardoperation *Create* auf ein durch "str1: STRECKE" deklariertes Attribut an. Wir würden damit (s.o.)

- erstens ein Objekt der *class* STRECKE erzeugen
- und zweitens in "str1" eine Referenz auf dieses Objekt.

Das durch "str1" referenzierte Objekt enthält nun freilich seinerseits, der Initialisierungsregel gemäß, in seinen Attributen "anfang" und "ende" leere Referenzen (vgl. Abbildung). Von diesem Objekt zum Beispiel die Operation "laenge" zu verlangen, ist daher sinnlos (und führt bei einem Programmlauf zu einem Fehler). Wir müßten zuvor die Create-Operation auf "str1.anfang" und "str1.ende" anwenden. Da nun erstens diese Vorgehensweise schon bei nur mäßig komplexen Objektstrukturen recht umständlich sein kann, und weil zweitens die Standardinitialisierungen nicht immer die zweckmäßigsten scheinen, erlaubt EIFFEL die klassenspezifische *Redefinition* der Create-Operation.




Für die Klassen PUNKT und STRECKE bietet es sich an, "Create" etwa wie folgt zu redefinieren:

```
class PUNKT export
   xkoord, ykoord,
   abstand, dist_ursprung,
   verschiebe
feature
   Create (x, y: REAL) is
      do
         xkoord := x;  ykoord := y;
      end;
   xkoord, ykoord: REAL;
   -- alle anderen features wie oben
   . . .
end
```

```
class  STRECKE export
   anfang, ende, laenge,
   neigung, verschiebe
feature
   Create (xa, ya, xe, ye: REAL) is
      do
         anfang.Create (xa, ya);
         ende.Create (xe, ye)
      end;
   anfang, ende: PUNKT;
   -- alle anderen features wie oben
   . . .
end
```

(Man beachte, daß die Create-Operation - auch wenn sie redefiniert wurde - nicht in die *export*-Liste aufgenommen wird!)

Die standardmäßige Existenz einer Create-Operation und ihr Gebrauch in EIFFEL-Programmen sind in guter Übereinstimmung mit der weiter oben vorgebrachten Interpretation von Objekten als Prozesse: Ein EIFFEL-Programm läuft ab als Interaktion zwischen Objekten. Damit aber Objekte untereinander agieren können, müssen sie zunächst einmal explizit erzeugt worden sein. (Mit einer anderen, ebenfalls standardmäßig gegebenen Operation mit dem Namen *Forget* kann man übrigens die Referenz eines Objektes (A) auf ein Objekt (B) auf "leer" zurücksetzen; damit wird B von A aus unzugänglich. In der von Meyer beschriebenen EIFFEL-Laufzeitumgebung ist es möglich, den Speicherplatz, der von gänzlich unzugänglich gewordenen Objekten besetzt wird, automatisch oder in (vom Programmierer) kontrollierter Weise zurückzugewinnen. Diese Technik ist unter der Bezeichnung *garbage collection* (Müll-Sammlung) bekannt.)

(iv) Dem aufmerksamen Leser wird ferner nicht entgangen sein, daß der Text einer EIFFEL-*class* offenbar nicht in einen öffentlichen (die Definition der Benutzungsschnittstelle enthaltenden) und einen privaten (die Implementierung detaillierenden) Teil gegliedert wird. Auf den ersten Blick scheint dies im Widerspruch zu dem in Abschnitt 6.4.1 geforderten Geheimnisprinzip zu stehen. Ein zweiter Blick (auf 6.4.1) jedoch lehrt, daß dieses Prinzip keineswegs kategorisch die syntaktische Trennung von Schnittstellendefinition und Implementierung verlangt. Es kommt vielmehr ausschließlich darauf an, daß der Benutzer eines Moduls auf dessen Interna nur über wohldefinierte - und damit natürlich gesondert mitteilbare - Operationen Zugang erhält.

Letzteres ist für *class*-Beschreibungen in EIFFEL aber ohne weiteres gewährleistet: Erstens sind die Objekte einer *class* nur mittels ihrer - in der *export*-Liste genannten - *features* benutzbar, und zweitens lassen sich alle für die Benutzung der Objekte einer *class* notwendigen Informationen aus der Beschreibung herausfiltern und (einem Programmierer) zur Verfügung stellen.

Zum Beispiel lauten die Schnittstellenbeschreibungen der Klassen PUNKT und STRECKE wie folgt :

```
class interface PUNKT
exported features
   xkoord, ykoord, abstand,
   dist_ursprung, verschiebe
feature specification
   Create (x,y: REAL)
   xkoord: REAL
   ykoord: REAL
   abstand (anderer_punkt: PUNKT): REAL
   dist_ursprung: REAL
   verschiebe (a, b: REAL)
   -- verschiebe horizontal um a,
   -- vertikal um b Einheiten
end interface -- class PUNKT
```

```
class interface STRECKE
exported features
   anfang, ende, laenge,
   neigung, verschiebe
feature specification
   Create (xa, ya, xe, ye: REAL)
   anfang: PUNKT
   ende: PUNKT
   laenge: REAL
   neigung: REAL
   verschiebe (a, b: REAL)
   -- verschiebe horizontal um a,
   -- vertikal um b Einheiten
end interface -- class STRECKE
```

Sie werden aus den Texten der Klassen durch Entfernung sämtlicher Klauseln ("*is* ...", usw.) gewonnen, welche auf die Implementierung Bezug nehmen. Dies ist eine sehr einfache Textmanipulation. Zur Verdeutlichung und zur Abgrenzung gegen die vom Compiler übersetzbaren *class*-Texte wurden die Benennungen der einzelnen Abschnitte leicht modifiziert (*class interface*, usw.).

Natürlich kann man solche Schnittstellenbeschreibungen auch als Vorgaben für die Modulimplementierung im Rahmen einer Systementwicklung einsetzen. Über die Realisierung der einzelnen *features* kann dann der EIFFEL-Programmierer ebenso entscheiden wie dies der MODULA-2-Programmierer bei der Gestaltung eines IMPLEMENTATION MODULE tun kann. Insbesondere hat er für die Rea-

lisierung von parameterlosen *features*, die einen Wert liefern (in den Beispielen oben sind das "xkoord", "ykoord", "dist_ursprung", "anfang", "ende", "laenge" und "neigung"), oft die Wahl zwischen Attributen und Funktionen, also zwischen der Speicherung des Wertes oder seiner Berechnung. So könnte ein Programmierer, der den Auftrag erhält, aufgrund der obigen Schnittstellenbeschreibung die *class* PUNKT zu implementieren, auch mit der folgenden Lösung aufwarten:

```
class PUNKT export
    xkoord, ykoord, abstand, dist_ursprung, verschiebe
feature
    Create (x, y: REAL) is
        do
            xkoord := x; ykoord := y;
            dist_ursprung := sqrt (x^2 + y^2)
        end;
    xkoord: REAL;
    ykoord: REAL;
    abstand (anderer_punkt: PUNKT): REAL is
        do
            Result := sqrt((xkoord - anderer_punkt.xkoord)^2 +
                           (ykoord - anderer_punkt.ykoord)^2)
        end;
    dist_ursprung: REAL;
    verschiebe (a, b: REAL) is
        -- verschiebe horizontal um a, vertikal um b Einheiten,
        -- und aktualisiere die Distanz zum Ursprung
        do
            xkoord := xkoord + a; ykoord := ykoord + b;
            dist_ursprung := sqrt(xkoord^2 + ykoord^2)
        end;
end
```

Der Wert des *feature* "dist_ursprung" wird bei dieser Implementierung nicht wie zuvor durch eine funktionale Routine berechnet, sondern ist als Inhalt eines "Attribut-Speichers" verfügbar. Für einen Benutzer der *class* PUNKT ist dieser Unterschied allerdings unsichtbar. Er erhält den Wert von "dist_ursprung" für ein durch "pkt1" referenziertes Objekt vom Klassentyp PUNKT nach wie vor über die Notation "pkt1.dist_ursprung".

Es gibt gute Gründe sowohl für als auch gegen das Angebot einer Programmiersprache, syntaktisch separate und für sich übersetzbare Einheiten zur Definition der Schnittstelle eines Moduls zu formulieren. Wir werden uns hier an einer Diskussion dieser Gründe nicht beteiligen. Nur soviel: Für den Entwerfer der Sprache EIFFEL überwog offenbar das Argument, so wenig Redundanz wie

möglich im Text eines Programmsystems zuzulassen. Jegliche für die Benutzung eines Moduls nötige Dokumentation soll aus dem vollständigen Modultext herausgezogen (*abstrahiert* !) werden können. Andererseits werden wir weiter unten, bei der Vorstellung von Sprachkonstrukten zur Anwendung des Vererbungskonzepts, sehen, daß eine (kompilierbare) EIFFEL-*class* auch ohne Implementierungsdetails geschrieben werden kann und somit eine dem Definitionsmodul in Sprachen wie MODULA-2 vergleichbare (wenn auch viel mächtigere) Rolle zu spielen in der Lage ist.

Apropos Benutzungsdokumentation: Dafür gibt es in EIFFEL noch mehr "Lekkerbissen", Prädikate (etwa im Sinne unseres Kapitels 4) nämlich, die in den Text von Klassen integrierbar sind. Damit können unter anderem sowohl die Vorbedingungen für die Anwendung eines *feature* als auch die durch ein *feature* zu garantierenden Nachbedingungen ausgedrückt werden. Der EIFFEL-Compiler bietet die Option, Code zur Überprüfung dieser Bedingungen zu generieren, der beim Ablauf eines Programmes - im allgemeinen - ein ähnliches Verhalten erzeugt wie die gegen Ende von Abschnitt 4.1.3 vorgeschlagene MODULA-2-Prozedur "Assertion": Er veranlaßt den Abbruch des Programms unter gleichzeitiger Ausgabe einer Fehlermeldung.

Mit Vor- und Nachbedingungen, wir erinnern uns, läßt sich die Semantik einer Berechnungsroutine beschreiben. In Kapitel 4, wir wiederholen es, haben wir sie zur Grundlage der Programmentwicklung gemacht. Vorbedingungen legen fest, was erfüllt sein muß, damit eine Routine ein definiertes Ergebnis liefern kann, und Nachbedingungen beschreiben eben dieses Ergebnis. Im Lichte der Abschnitte 6.4.2 und 6.4.3 können wir diese Bedingungen daher auch als Teile jener "Nutzungsverträge" ansehen, die zwischen dem Erbringer und dem Nutzer von Dienstleistungen geschlossen werden. Und bis zu einem gewissen Grade können wir - wie sich sogleich zeigen wird - mit ihnen sogar die in Abschnitt 6.4.3 durchgenommene axiomatische Definition der Semantik eines ADTs nachvollziehen.

Eine vollständige Darstellung der in EIFFEL verfügbaren Mittel zur Formulierung von Prädikaten sprengt zwar den Rahmen dieses summarischen Überblicks, doch lohnt es sich, sie wegen ihrer Einsatzmöglichkeiten sowohl bei der Dokumentation, als auch für den Test von Programmsystemen, zumindest ansatzweise zu skizzieren. Wir betrachten dazu das (vorläufige) Gerüst eines als *class* implementierten ADT "STACK_OF_INT" (mit dem - gegenüber dem Beispiel in Abschnitt 6.4.3 - zusätzlichen *feature* "anz_elemente"):

```
class STACK_OF_INT export
    anz_elemente, push, pop, top,
    isnew, replace
feature
    anz_elemente: INTEGER;
```

```
    push (x: INTEGER) is
      do  ... end ;
    pop is
      do  ... end;
    top: INTEGER is
      do  ... end;
    isnew: BOOLEAN is
      do  ... end;
    replace (x: INTEGER) is
      do  ... end;
  end  -- class STACK_OF_INT
```

Die in 6.4.3 angegebene Operation NEWSTACK wird hier durch das standardmäßige (und daher nicht explizit notierte) Create-*feature* realisiert. "Standard-Create" weist, wie wir gelernt haben, dem Attribut "anz_elemente" den Wert 0 zu. Das Axiom (vgl. 6.4.3) "ISNEW(NEWSTACK) =TRUE" läßt sich daher als Nachbedingung für das *feature* "isnew" auffassen:

"*feature* 'isnew' liefert den Wert von (anz_elemente = 0)".

Dagegen wird "ISNEW(PUSH(stk, elm)) = FALSE" zu einer Nachbedingung für das *feature* "push":

"Nach Benutzung von *feature* 'push' liefert 'isnew' den Wert *false*".

Das Axiom "TOP(NEWSTACK) = UNDEF" übersetzen wir in eine Vorbedingung für das *feature* "top":

"*feature* 'top' darf nur verlangt werden, wenn 'isnew' den Wert *false* hat".

"TOP(PUSH(stk, elm)) = elm" führt zu einer weiteren Nachbedingung für das *feature* "push":

"Nach 'push(x)' liefert 'top' den Wert x".

Etwas weniger offensichtlich ist die Umsetzung der beiden POP-Axiome. Deren zweites ("POP(PUSH(stk,elm)) = stk") gibt Anlaß zu Nachbedingungen sowohl für das *feature* "push" als auch für "pop":

"Nach 'push' ist 'anz_elemente' um 1 größer"

und

"Nach 'pop' ist 'anz_elemente' um 1 kleiner".

Wegen "POP(NEWSTACK) = NEWSTACK" muß die letztere Nachbedingung noch um den Fall ergänzt werden, daß vor Benutzung von "pop" "isnew" den Wert *true* hatte, und "anz_elemente" sich durch "pop" nicht ändert.

Für "replace" schließlich ergibt sich die Nachbedingung aus jenen für "push" und "pop":

"Nach 'replace(x)' liefert 'top' den Wert x
und 'anz_elemente' ändert sich nicht,

falls vor 'replace' 'isnew' den Wert *false* hatte;
andernfalls ist 'anz_elemente' um 1 größer".

In EIFFEL werden Vorbedingungen in einer *require*-Klausel und Nachbedingungen in einer *ensure*-Klausel verpackt. Damit kann man (wenn man will!) im Text einer *class* eine (formale aber nicht notwendigerweise vollständige!) Spezifikation der *features* ihrer Objekte unterbringen. Für die STACK-*class* sieht das folgendermaßen aus:

```
class STACK_OF_INT export
   anz_elemente, push, pop, top, isnew, replace
feature
   anz_elemente: INTEGER;
   push (x: INTEGER) is
      do  ...
      ensure
         not isnew; top = x;
         anz_elemente = old anz_elemente + 1
      end ;
   pop is
      do  ...
      ensure
         (not old isnew and (anz_elemente = old anz_elemente - 1))
         or (old isnew and Nochange)
      end;
   top: INTEGER is
      require
         not isnew
      do  ...
      end;
   isnew: BOOLEAN is
      do  ...
      ensure
         Result = (anz_elemente = 0)
      end;
   replace (x: INTEGER) is
      do  ...
      ensure
         top = x and
         ((not old isnew and Nochange)
         or (old isnew and (anz_elemente = old anz_elemente + 1)))
      end;
end  -- class STACK_OF_INT
```

(Der Operator *old* liefert den Wert seines Operanden vor Benutzung des jeweiligen *features* ; "Nochange" ist der Standardname für einen booleschen Ausdruck, der genau dann den Wert *true* hat, wenn die Benutzung des jeweiligen *features* die Werte der Attribute der *class* nicht geändert hat. Die mit den Operatoren *and*, *or*, etc. gebildeten Ausdrücke sind mit den üblichen Präzedenzregeln zu interpretieren.)

Wie gesagt: Man kann diese Klauseln einbauen, muß es aber nicht. Tut man es, so kann man den Compiler anweisen, Code zur Überprüfung der entsprechenden Bedingungen zu erzeugen. Der Nutzen, den man hiervon beim Test eines Programmsystems haben kann, ist offensichtlich. Andererseits: *require*- und *ensure*-Klauseln gehören zur Interface-Abstraktion und sind damit Teil der Benutzungs- bzw. Entwurfsdokumentation einer *class*. Sie vermitteln wichtige (wenn auch nicht notwendigerweise sämtliche) Informationen über die Semantik der Objekte einer *class*. Die aus der *class* STACK (soweit oben ausgearbeitet) abstrahierte Schnittstellenbeschreibung lautet zum Beispiel wie folgt:

```
class interface STACK_OF_INT
exported features
   anz_elemente, push, pop, top, isnew, replace
feature specification
   anz_elemente: INTEGER;
   push (x: INTEGER)
      ensure
         not isnew; top = x;
         anz_elemente = old anz_elemente + 1
   pop
      ensure
         (not old isnew and (anz_elemente = old anz_elemente - 1))
         or (old isnew and Nochange)
   top: INTEGER
      require
         not isnew
   isnew: BOOLEAN
      ensure
         Result = (anz_elemente = 0)
   replace (x: INTEGER)
      ensure
         top = x and
         ((not old isnew and Nochange)
         or (old isnew and (anz_elemente = old anz_elemente + 1)))
end interface  -- class STACK_OF_INT
```

(v) Mit einer letzten Bemerkung vor unserem Einstieg in die "objektorientierte Vererbungslehre" wollen wir darauf hinweisen, daß EIFFEL, ganz ähnlich wie der im vorangegangenen Abschnitt für die Definition und abstrakte Implementierung von ADTn verwendete Formalismus, die Möglichkeit bietet, Klassen zu *parametrisieren.* Die Motivation hierfür liegt auf der Hand: Wir haben es oftmals mit Datenstrukturen zu tun, deren elementare Teile alle von ein und demselben Typ, sagen wir T, sind. Beispiele sind STACK und ARRAY, aber auch alle Arten von Listenstrukturen. Die für die genannten Strukturen interessanten Operationen machen von den speziellen Eigenschaften der Objekte des Typs T freilich im allgemeinen keinerlei Gebrauch. So ist es für die Definition des *feature* "push" eines Stacks unerheblich, ob das Objekt, welches auf den Stapel gelegt werden soll, eine ganze Zahl ist, ein Stück Kuchen oder ein Array von Abbildungen. (Wichtig ist allein, daß das zuletzt auf den Stack gelegte Objekt dort auch an oberster Stelle zu finden ist.) In EIFFEL wird dieser Einsicht dadurch Rechnung getragen, daß man bei der Definition von Klassen diese mit formalen *Klassenparametern* (bzw. *Typparametern*) versehen darf. Anstatt also der Klassen "STACK_ OF_ INT", "STACK_OF_REAL", "STACK_OF_ POINT", etc. definiert man eine einzige *class* "STACK[T]", wobei T ein Klassenparameter (bzw. Typparameter) ist:

```
class STACK[T] export
    anz_elemente, push, pop, top,
    isnew, replace, isfull
feature
    anz_elemente: INTEGER;
    push(x: T) is
        do  ...  end ;
    pop is
        do  ...  end;
    top:  T is
        do  ...  end;
    isnew: BOOLEAN is
        do  ...  end;
    replace(x: T) is
        do  ...  end;
    isfull: BOOLEAN is
        do  ...  end;
end  -- class STACK[T]
```

(Wir haben hier, um das Beispiel etwas realistischer zu machen, das zusätzliche *feature* "isfull" eingeführt - schließlich sollte auch einem Stack nicht unbegrenzt viel Speicherplatz zur Verfügung stehen. Der Leser überlege sich, wie nach dieser Modifikation die *require* - und *ensure* - Klauseln - insbesondere für "pop" und "push" ausschauen sollten.)

Ein Benutzer dieser *class* muß bei der Deklaration eines Attributs vom Klassentyp STACK natürlich angeben, von welcher Klasse (bzw. welchem Typ) die Objekte sind, die er auf dem durch dieses Attribut referenzierten Stack unterbringen will. So bezeichnet ein als

intstk: STACK [INTEGER]

deklariertes Attribut "intstk" einen Stack für Objekte vom Typ INTEGER, während das durch

ptstk: STACK [POINT]

deklarierte Attribut "ptstk" einen Stack für Objekte der *class* POINT meint.

Nehmen wir weiter an, daß der gleiche Benutzer zwei Attribute wie folgt deklariert hat:

"int1: INTEGER" und "pt1: POINT".

Dann ist es offensichtlich zum Beispiel erlaubt, auf dem durch "intstk" referenzierten Stack Werte des Attributs "int1" und auf dem durch "ptstk" referenzierten Stack Werte des Attributs "pt1" abzulegen. Die folgenden *feature* - Anwendungen sind daher legitim:

"intstk.push(int1)" und "ptstk.push(pt1)".

Dagegen sind zum Beispiel die *feature* - Anwendungen

"intstk.replace(pt1)", "ptstk.push(int1)" und "pt1 := intstk.top"

verboten, da die aktuellen Typ-Parameter der STACK-Attribute nicht mit den Typen der Stack-Elemente übereinstimmen. (Ein Punkt darf nicht auf einen Stack für ganze Zahlen gelegt werden, usw.) Solche *feature* - Anwendungen werden vom EIFFEL-Compiler abgelehnt.

Klassen, welche mit Typ-/Klassenparametern definiert sind, werden auch als *generisch* bezeichnet, da sie gewissermaßen ein formales Muster für die Erzeugung aktueller Klassen liefern. Der zweite, im vorangegangenen Abschnitt diskutierte ADT, das ARRAY, würde als EIFFEL - *class* die folgende *generische* Gestalt annehmen:

```
class ARRAY[DT, RT]
   -- DT ist der Domain-Typ und
   -- RT ist der Range-Typ
export
   assign, access
feature
   Create(...) is
      do ... end;
   assign (dval: DT; rval: RT) is
      do ... end;
   access (dval: DT): RT is
      do ... end;
end -- class ARRAY[DT, RT]
```

Hier gibt es jedoch offenbar ein Problem: "Create" müßte, um ein Objekt dieser *class* zu erzeugen, einige Kenntnis über den Domain-Typ und über Eigenschaften von dessen Objekten haben. Beispielsweise müßte es wissen (etwa um genügend Speicherplatz zu reservieren), wieviele Elemente DT umfaßt, und ob auf die Objekte des ihm in DT übergebenen Typs ihrerseits die Create-Operation anwendbar ist. Andererseits haben wir ausdrücklich ausgeschlossen, daß bei der Realisierung der *features* einer generischen Klasse von irgendwelchen speziellen Eigenschaften der als aktuelle Parameter zulässigen Klassen/Typen Gebrauch gemacht wird. Es soll eben jede Klasse und jeder einfache Typ als aktueller Parameter auftreten können. Aus diesem Grunde kann für den Domain-Typ einer ARRAY-*class* kein formaler Parameter eingesetzt werden. Der Range-Typ dagegen, das ist klar, kann völlig offen bleiben. So begnügen wir uns mit einer wie folgt definierten generischen ARRAY-*class*:

```
class ARRAY[T]
   -- T ist der Typ der in einem
   -- ARRAY-Objekt gespeicherten Elemente
export
   low, high, size, assign, access
feature
   Create(l,h: INTEGER) is
      -- Erzeugt ein ARRAY-Objekt mit unterem Index l
      -- und oberem Index h; das Objekt ist leer, wenn l > h
      do ... end;
   low, high, size: INTEGER;
   assign (i: INTEGER; val: T) is
      do ... end;
   access (i: INTEGER): T is
      do ... end;
end -- class ARRAY[T]
```

Dies ist jedenfalls hinreichend, um die in Abschnitt 6.4.3 vorgeführte (abstrakte und generische) Implementierung des STACK-ADT durch den ARRAY-ADT im Formalismus der EIFFEL-Klassen nachzuvollziehen.

(Man beachte dabei, daß
- *features* als Attribute definiert werden können, deren Typ ein generischer Klassentyp ist, und daß
- nicht alle *features* der Implementierung exportiert werden.

Die *require* - und *ensure* - Klauseln haben wir der Kürze halber nicht mitübernommen. Der Leser möge dies nachholen.)

```
class STACK[T] export
   anz_elemente, push, pop, top,
   isnew, replace, isfull
```

```
feature
   stack_array: ARRAY[T];
   max_hoehe: INTEGER;
   anz_elemente: INTEGER;
   Create(n: INTEGER) is
      -- falls n>0, erzeuge ein STACK-Objekt für maximal
      -- n Objekte; sonst erzeuge ein STACK-Objekt für
      -- 0 Objekte
      do
         if n>0 then max_hoehe := n end;
         -- andernfalls wird "Create" "max_hoehe"
         -- automatisch mit 0 initialisieren
         stack_array.Create(1, max_hoehe)
      end;
   push (x: T) is
      do
         anz_elemente := anz_elemente + 1;
         stack_array.assign(anz_elemente, x)
      end ;
   pop is
      do
         if not isnew then
            anz_elemente := anz_elemente - 1;
         end
      end;
   top: T is
      do
         Result := stack_array.access(anz_elemente)
      end;
   isnew: BOOLEAN is
      do
         Result := (anz_elemente = 0)
      end;
   replace (x: T) is
      do
         pop; push(x)
      end;
   isfull: BOOLEAN is
      do
         Result := (anz_elemente = max_hoehe)
      end;
end -- class STACK[T]
```

Wir haben nun hinreichenden Einblick in die Werkstatt des EIFFEL-Programmierers bekommen, um auch sein wichtigstes und mächtigstes Instrument, das Konzept der *Vererbung*, kennen und schätzen zu lernen. Die naheliegendste Motivation für die Unterstützung dieses Konzeptes durch geeignete Wendungen einer Programmiersprache ergibt sich aus dem Wunsch, bei der Definition von Modulen (bzw. Klassen) nicht "bei Adam und Eva" beginnen zu müssen, sondern auf bereits geleisteter Arbeit, sprich auf vorhandenen Moduldefinitionen, aufbauen zu können. Aus gutem Grund haben wir "Definition" (von Modulen) hier unterstrichen, denn daß wir uns bei der Implementierung eines Moduls bereits existierender Module bedienen (müssen), ist inzwischen ja ein "alter Hut". Und dafür kommen wir mit der Realisierung des Modulkonzepts à la MODULA-2 (oder ADA oder ähnlicher Programmiersprachen) auch ohne weiteres aus.

Angenommen, unser (EIFFEL-)Programmierer hat die Aufgabe, eine *class* B zu beschreiben. Er stellt fest, daß die Objekte dieser *class* im Prinzip die gleichen Eigenschaften haben wie die Objekte einer schon vorher (von ihm selbst oder wem auch immer) beschriebenen *class* A. Er stellt jedoch auch fest, daß

- erstens die B-Objekte einige zusätzliche Eigenschaften haben, und daß
- sich zweitens einige der Eigenschaften, welche B-Objekte mit A-Objekten gemeinsam haben, für B-Objekte einfacher ausdrücken (berechnen!) lassen als für A-Objekte.

Um es konkret zu machen: Die Objekte der *class* A mögen die *features* "fA_1", ..., "fA_n" haben, und die Objekte der *class* B zusätzlich die *features* "fB_1", ..., "fB_m". Nicht alle *features* von A werden auch exportiert, und von B sollen irgendwelche *features* exportiert werden, die teilweise von A stammen und teilweise für B spezifisch sind. Für B-Objekte mögen sich die *features* "fA_1" und "fA_2" einfacher berechnen lassen als für A-Objekte. Damit kann *class* B wie folgt beschrieben werden:

```
class B export
   -- einige (alte) A- und einige (neue) B-features
   . . .
inherit
   A redefine fA_1, fA_2
feature
   Create(...) is
      do
         -- für B-Objekte spezifische Create-Operation
      end
   fA_1 ... is
      do ... end;
   fA_2 ... is
      do ... end;
```

```
    fB_1 ... is
      do ... end;
    . . .
    fB_m ... is
      do ... end;
  end -- class B
```

Die entscheidende Klausel dieser Definition heißt *inherit* mit der Subklausel *redefine*. Mit "*inherit* A" übernehmen ("erben") die B-Objekte alle *features* (Attribute und Routinen) der A-Objekte. In der *feature*-Klausel von B brauchen diese dann nicht mehr aufgeführt zu werden. B heißt auch *Erbe* bzw. *Nachkomme* (engl. *descendant*) von A, und umgekehrt heißt A *Elter* bzw. *Vorfahre* (engl. *ancestor*) von B.

Wie bei der biologischen Vererbung sind die Eigenschaften von Nachkommen nicht hundertprozentig identisch mit denen der Eltern-Klasse. Abgesehen davon, daß Nachkommen mehr Eigenschaften haben können als ihre Eltern, gilt in EIFFEL die Regel, daß bei der Übernahme der Eigenschaften der Eltern-Klasse die Semantik dieser Eigenschaften (ausgedrückt in *require*- und *ensure*-Klauseln) erhalten bleibt, während die Implementierung sich ändern kann. Ist dies gewünscht, so wird die betreffende Eltern-Eigenschaft in die *redefine*-Klausel aufgenommen und in der *feature*-Klausel dann tatsächlich neu geschrieben.

Es gilt ferner die Regel, daß für jede Nachkommen-Klasse ein spezielles "Create" geschrieben werden muß, welches die besonderen Eigenschaften der Objekte dieser Klassen berücksichtigt. "Create" wird also in keinem Falle vererbt.

Es ist an der Zeit, das bisher Gesagte durch Beispiele zu illustrieren. Wir betrachten die folgenden Objekte: Vierecke (ganz allgemein), gleichseitige Vierecke ("Rauten"), Rechtecke und Quadrate.

Vierecke (ganz allgemein) haben (pardon!) "vier Eckpunkte", einen "Umfang", eine "Fläche", man kann sie "verschieben" und einiges mehr mit ihnen tun. Rauten, Rechtecke und Quadrate haben - sie sind ja spezielle Vierecke - die gleichen Eigenschaften. Doch erstens lassen sich manche dieser Eigenschaften in diesen speziellen Fällen einfacher bestimmen als im allgemeinen Fall, und zweitens mag es in den speziellen Fällen einige Eigenschaften geben, die nur für diese interessant sind. Wir definieren zunächst die *class* "Vierecke":

```
class VIERECK export
    ecken, umfang, flaeche, verschiebe
feature
    Create(e1,e2,e3,e4: POINT) is
      do ... end;
    ecken: ARRAY[POINT]; -- zum Beispiel
    umfang: REAL is
      do ... end;
```

```
    flaeche: REAL is
      do ... end;
    verschiebe(a, b: REAL) is
      do ... end;
  invariant
    ecken.size = 4;
    ecken.access(ecken.low).xkoord <
                ecken.access(ecken.low+1).xkoord;
    ecken.access(ecken.low+1).ykoord <
                ecken.access(ecken.low+2).ykoord;
    ecken.access(ecken.low+2).xkoord >
                ecken.access(ecken.low+3).xkoord;
    ecken.access(ecken.low+3).ykoord >
                ecken.access(ecken.low).ykoord
  end -- class VIERECK
```

Dabei haben wir von einer weiteren (bisher unerwähnten) Möglichkeit Gebrauch gemacht, die Objekte einer *class* durch Prädikate zu beschreiben: In der *invariant*-Klausel wird festgelegt, was unter allen Umständen für alle Objekte gelten soll, was also insbesondere nicht durch die Anwendung irgendeiner der *features* verändert werden darf. Im vorliegenden Fall wird die Lage der Eckpunkte des Vierecks in der kartesischen Ebene beschrieben. (Im Array "ecken" werden die Punkte in der Reihenfolge gegen den Uhrzeiger von "links unten" bis "links oben" gespeichert.) Auch die Überprüfung von *Invarianten* kann übrigens durch die in Bemerkung (iv) erwähnte Compileroption aktiviert werden.

Der "umfang" eines Objekts der *class* VIERECK wäre etwa durch die folgende Routine zu berechnen:

```
umfang: REAL is
  local
    ind: INTEGER
  do
    from
      ind := ecken.low
    until
      ind = ecken.high
    loop
      Result := Result + ecken.access(ind).abstand(ecken.access(ind + 1));
      ind := ind + 1
    end;
    Result := Result + ecken.access(ind).abstand(ecken.access(ecken.low));
  end;
```

(Dem inzwischen geübten Leser dürfte es nicht mehr schwer fallen, die hier

neu in's Spiel gebrachte *local*-Klausel sowie das Schleifenkonstrukt einzuordnen und zu verstehen. "Result" wird automatisch mit 0 initialisiert. Zur weiteren Übung schreibe er eine Routine, die das *feature* "verschiebe" realisiert.)

Wie gesagt: Rauten haben die gleichen Eigenschaften, doch um eine von ihnen zu erzeugen, ist es vielleicht sinnvoller, anstelle der vier Eckpunkte die Endpunkte einer der Seiten anzugeben und den Winkel, den diese Seite mit einer vereinbarten Nachbarseite bildet. Dies wird in der speziell für Rauten zu schreibenden Create-*feature* auszudrücken sein. Die allgemeine Routine zur Berechnung des Umfangs eines Vierecks auf eine Raute anzuwenden, muß ebenfalls als wenig zweckmäßig angesehen werden. Und schließlich sollten für Rauten mindestens noch die zusätzlichen *features* "seite", "winkel1" und "winkel2" angeboten werden. Diese Überlegungen führen zu der folgenden *class* RAUTE:

```
class RAUTE export
   ecken, umfang, flaeche, verschiebe,
   seite, winkel1, winkel2
inherit
   VIERECK redefine umfang
feature
Create(p1, p2: POINT; wkl: REAL) is
   do
      winkel1 := wkl;
      -- Berechnung der Eckpunkte
   end;
umfang: REAL is
   do
      Result := 4 * seite
   end;
seite: REAL is
   do
      Result := ecken.access(ecken.low).abstand(ecken.access(ecken.low+1))
   end;
winkel1: REAL;
winkel2: REAL is
   do ... end;
invariant
   -- die Invariante der Klasse VIERECK gilt auch
   -- für alle ihre Nachkommen; außerdem gilt:
   ecken.access(ecken.low + 1).abstand(ecken.access(ecken.low + 2)) = seite;
   ecken.access(ecken.low + 2).abstand(ecken.access(ecken.low + 3)) = seite;
   ecken.access(ecken.low + 3).abstand(ecken.access(ecken.low)) = seite
end -- class RAUTE
```

Als einen weiteren Abkömmling der *class* VIERECK beschreiben wir die *class* RECHTECK, für die wir die zusätzlichen *features* "seite1" und "seite2" einführen. "Create" soll RECHTECK-Objekte aufgrund der Vorgabe der Eckpunkte der ersten Seite (s.o.) und der Länge der anliegenden Seiten erzeugen. Außerdem entscheiden wir uns hier für eine (im speziellen Fall von Rechtecken) bessere Version der Routinen, welche die *features* "umfang" und "flaeche" realisieren.

```
class RECHTECK export
   ecken, umfang, flaeche, verschiebe,
   seite1, seite2
inherit
   VIERECK redefine umfang, flaeche
feature
   Create(p1, p2: POINT; sl: REAL) is
      do
         seite1 := sl;
         seite2 := p1.abstand(p2);
         -- Berechnung der Eckpunkte
      end;
   umfang: REAL is
      do
         Result := 2 * (seite1 + seite2)
      end;
   flaeche: REAL is
      do
         Result := seite1 * seite2
      end;
   seite1: REAL;
   seite2: REAL;
invariant
   -- die Invariante der Klasse VIERECK gilt auch
   -- für alle ihre Nachkommen; außerdem gilt:
   -- gegenüberliegende Seiten sind "seite1" bzw. "seite2"
   -- lang, und alle Winkel sind rechte
   -- (vom Leser zu formalisieren).
end -- class RECHTECK
```

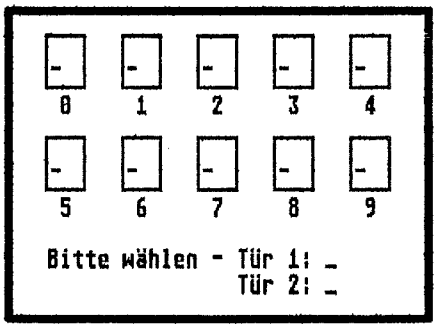
Der letzte oben genannte Vertreter im Kreis unserer Vierecks-Familie schließlich ist das Quadrat. Wir können es als Nachkomme entweder der Raute oder des Rechtecks definieren. Wir entscheiden uns für das Rechteck als *Elter*. Zur Erzeugung eines QUADRAT-Objekts genügt die Angabe der Endpunkte der "ersten" Seite (s.o.). Das Quadrat erbt alle *features* des Rechtecks, und es hat eigentlichen keinen Sinn, irgendeines davon zu redefinieren. Unter diesen *features* ist freilich auch "seite2", ein Attribut, das nun ziemlich überflüssig ist. Aber

wir brauchen es ja nicht zu exportieren. Und damit der Name "seite1" keine Verwirrung stiftet, taufen wir dieses *feature* einfach um! Dies führt zu der folgenden Klassen-Definition:

```
class QUADRAT export
   ecken, umfang, flaeche, verschiebe, seite
inherit
   RECHTECK rename seite1 as seite
feature
   Create(p1, p2: POINT) is
      do
         seite := p1.abstand(p2);
         seite2 := seite;
         -- Berechnung der Eckpunkte
      end;
invariant
   -- die Invariante der Klasse RECHTECK gilt und:
   seite = seite2
end -- class QUADRAT
```

An dieser Stelle wollen wir ein wenig innehalten und uns vorstellen, daß zwei Programmiererinnen, Evi und Monika, die Aufgabe gestellt ist, ein (zugegebenermaßen ziemlich simples) Glücksspiel zu programmieren, bei dem es darauf ankommt, Vierecke zu manipulieren und deren Umfang und Fläche zu berechnen. Die ins Spiel gebrachten Vierecke können von ganz allgemeiner Gestalt sein, oder aber spezielle Eigenschaften haben, die sie als Raute, Rechteck oder Quadrat charakterisieren.

Das Spiel hat folgenden Verlauf: Auf dem Bildschirm wird eine Anzahl (sagen wir 10) numerierter Türen dargestellt (siehe Abbildung). Hinter jeder dieser Türen hat "der Rechner" ein Viereck versteckt, und zwar ganz willkürlich ausgewählt eine Raute, ein Rechteck, ein Quadrat oder ein Viereck ohne besondere Eigenschaften. Der Spieler soll nun (mittels Eingabe von Nummern) zwei Vierecke auswählen. Er gewinnt, wenn das Viereck hinter der ersten gewählten Tür einen größeren Umfang und eine größere Fläche hat als das hinter der zweiten Tür. (Natürlich kann man dieses Spiel auch etwas intelligenter gestalten, doch darauf kommt es uns hier wirklich nicht an.)




Von den beiden Damen hat übrigens Monika ihre Ausbildung in MODULA-2 erhalten, während Evi eine EIFFEL-Spezialistin ist.

Bevor Evi die *class* EIN_SPIEL schreibt, wird sie den Katalog der bereits im EIFFEL Programmiersystem vorhandenen Klassen konsultieren und (unter manchem anderem) die *class* VIERECK finden. Innerhalb der *class* EIN_SPIEL wird sie ein *feature*, nennen wir es "vierecke", benötigen, welches vom Klassentyp ARRAY[VIERECK] ist. Bei der Erzeugung eines Objekts der Klasse EIN_SPIEL wird zunächst dieses Array zur Aufnahme von zehn Vierecken angelegt und dann mit zufällig als Raute, Rechteck, usw. kreierten Figuren "gefüllt". Es gibt ferner zwei (Attribut-)*features* "erste_wahl" und "zweite_wahl", beide vom Typ INTEGER, die durch Speicher für die vom Spieler gewählten Türnummern realisiert sind. Ein weiteres *feature*, "tuer_auswahl", sorgt dafür, daß der Spieler die Auswahl tatsächlich trifft, und macht die gewählten Nummern als *feature* "erste_wahl" bzw. "zweite_wahl" zugänglich. Und schließlich gibt es das *feature* "tuer_vergleich: BOOLEAN", welches genau dann *true* liefert, wenn sowohl der Umfang als auch die Fläche des hinter der ersten Tür verborgenen Vierecks größer sind als die entsprechenden Maße des Vierecks hinter der zweiten Tür. Wir geben hier zunächst die wesentlichen Textpassagen der *class* EIN_SPIEL wieder und werden uns erst danach von Evis Geschick als EIFFEL-Programmiererin überzeugen:

```
class EIN_SPIEL export
    tuer_auswahl, tuer_vergleich, ...
    -- beachte: "vierecke", "erste_wahl" und "zweite_wahl"
    -- werden nicht exportiert
feature
    vierecke: ARRAY[VIERECK];
    erste_wahl, zweite_wahl: INTEGER;
    Create is
      local
        ve: VIERECK;
        ra: RAUTE;
        re: RECHTECK;
        qu: QUADRAT;
        ind, z: INTEGER;
      do
        vierecke.Create(0,9);
        from
          ind := vierecke.low
        until
          ind > vierecke.high
        loop
          -- erzeuge Zufallszahl z aus {0, 1, 2, 3}
          if z = 0 then
            -- erzeuge zufällige Daten für allgemeines Viereck
```

```
                ve.Create(...);
                vierecke.assign(ind, ve);
                ve.Forget  -- zu "Forget" siehe oben; eventuell
                -- muß auf "ve" wieder Create angewendet werden!
              elsif  z = 1 then
                -- erzeuge zufällige Daten für Raute
                ra.Create(...);
                vierecke.assign(ind, ra);
                ra.Forget
              elsif  z = 2 then
                -- erzeuge zufällige Daten für Rechteck
                re.Create(...);
                vierecke.assign(ind, re);
                re.Forget
              else  -- z = 3
                -- erzeuge zufällige Daten für Quadrat
                qu.Create(...);
                vierecke.assign(ind, qu);
                qu.Forget
              end;
              ind := ind + 1
            end
          end;
        tuer_auswahl is
          do
            -- entsprechend der Beschreibung oben
          end;
        tuer_vergleich: BOOLEAN is
          do
            Result := (vierecke.access(erste_wahl).umfang
                    > vierecke.access(zweite_wahl).umfang)
                    and
                    (vierecke.access(erste_wahl).flaeche
                    > vierecke.access(zweite_wahl).flaeche)
          end;
        ...
      end  -- class EIN_SPIEL
```

Was hat Evi hier getan? Sie hat offenbar - bevor sie mit der Arbeit an EIN_SPIEL begann - die Klassen RAUTE, RECHTECK und QUADRAT als Nachkommen der *class* VIERECK definiert, und wir nehmen an, daß sie eben jene Definitionen getroffen hat, die wir oben notiert haben. Die *features* "flaeche" und "umfang" sind also jeweils entsprechend redefiniert. Wenn wir nun Evis *feature*

"Create" etwas genauer unter die Lupe nehmen, so machen wir eine Entdeckung, die uns als hartgesottene MODULA-2 (oder PASCAL- oder ADA-) Programmierer durchaus erstaunt: Da werden doch tatsächlich Objekte (nämlich Rauten, Rechtecke und Quadrate) einem Attribut ("vierecke") zugeordnet, dessen (Klassen-)Typ sich von den Typen dieser Objekte sehr wohl unterscheidet. Soviel Freiheit sind wir nicht gewohnt, denn bei der Arbeit mit Sprachen wie MODULA-2, PASCAL, usw. haben wir sehr genau darauf zu achten, daß in die einzelnen "Zellen" einer etwa als vom "ARRAY OF ViereckTyp" deklarierten Variablen auch wirklich nur die Inhalte von Variablen des Typs "ViereckTyp" übertragen werden dürfen. Ausnahmen bilden allenfalls bereits in den Sprachdefinitionen festgelegte sogenannte Kompatibilitätsregeln, denen zufolge zum Beispiel in einem MODULA-2 Programm der Inhalt einer als CARDINAL deklarierten Variablen einer Variablen vom Typ INTEGER zugewiesen werden darf.

In EIFFEL gelten weniger strenge Vorschriften was die "Typtreue" betrifft (oder, wenn man so will, großzügigere Kompatibilitätsregeln). Doch Regeln gibt es allemal: Schließlich bemerken wir, daß alle Objekte, die hier den "Zellen" des ARRAYS "vierecke" zugeordnet werden, zu Klassen gehören, welche von der *class* VIERECK abstammen.

Diese Beobachtung ist in der Tat der Schlüssel zum Verständnis der obigen Create-Operation. In größerer Allgemeinheit dargestellt haben wir es mit folgender Situation zu tun:

Gegeben sind eine *class* A und Nachkommen A_1, A_2, ..., A_n dieser *class*. Angenommen, in einer anderen *class*, nennen wir sie A_KUNDE, deklarieren wir die Attribute:

```
a_array: ARRAY[A];
a_att: A;
a_1_att: A_1;
...;
a_n_att: A_n;
```

Dann ist es gestattet, im Text von A_KUNDE die folgenden Zuweisungen zu notieren:

```
a_att := a_i_att;
```

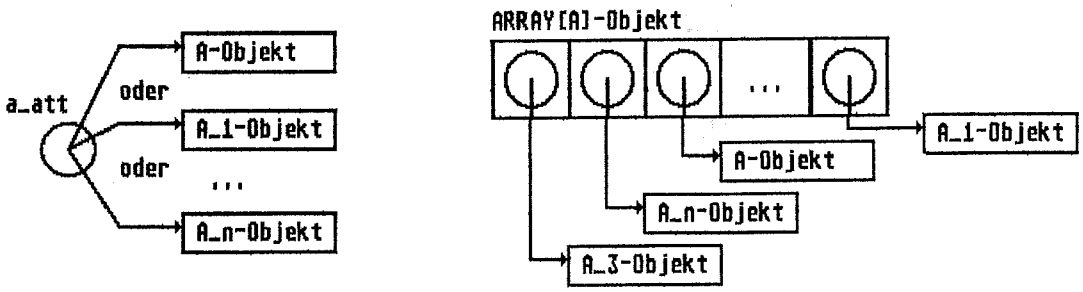
und

```
a_array.assign(ind, a_i_att);
```

wobei $i \in \{1, 2, ..., n\}$.

Mit anderen Worten: Das Attribut "a_att" kann nicht nur auf Objekte vom (Klassen-)Typ A, sondern auch auf Objekte der von A mit Hilfe des Mechanismus der Vererbung abgeleiteten Klassen A_1, ..., A_n verweisen. Und daher kann ein Objekt der *class* ARRAY[A] auch Verweise auf Objekte enthalten, die zu Klassen gehören, welche Erben der *class* A sind (vgl. die Abbildung auf der nächsten Seite). Diese Möglichkeit, mit ein und demselben Attribut Objekte

verschiedener Klassen zu referenzieren, wird als *Polymorphie* bezeichnet. In Sprachen wie MODULA-2 kann man ähnliches nur mit "Tricks" (z.B. durch Rekurs auf Sprachelemente, bei denen keine Typüberprüfung durch den Compiler stattfindet) erreichen, welche schwer durchschaubar sind und deshalb die Programmierung in hohem Maße fehleranfällig machen. In EIFFEL ist *Polymorphie* auf die Abkömmlinge einer *class* beschränkt und damit genau kontrollierbar.




Doch was, so müssen wir uns nun fragen, hat Evi davon, wenn sie sich der der Sprache EIFFEL eigenen *Polymorphie* von Attributen bedient? Um dies zu verstehen, schauen wir uns das *feature* "tuer_vergleich" etwas genauer an.

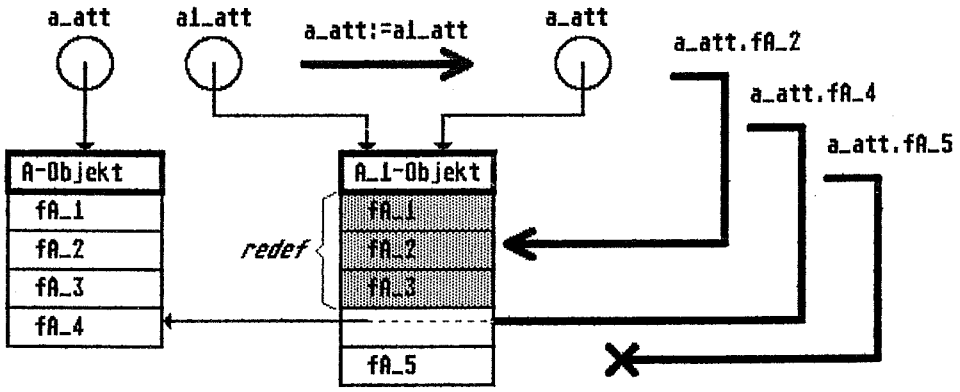
Zur Laufzeit des durch den Text der *class* EIN_SPIEL gesteuerten EIFFEL-Programms kann - wie wir ja soeben gelernt haben - "vierecke.access(erste_wahl)" ein Objekt referenzieren, welches zu einer der vier Klassen VIERECK, RAUTE, RECHTECK oder QUADRAT gehört (je nachdem, welcher Art das Viereck ist, das von "Create" hinter der gewählten Tür versteckt wurde). Auf das jeweils referenzierte Objekt wendet "tuer_vergleich" nun die Routinen "umfang" und "flaeche" an. Aber welche Versionen dieser Routinen sind gemeint? Wir blicken zurück: Für die speziellen Vierecke "Raute", "Rechteck" und "Quadrat" haben wir, die Möglichkeit der Redefinition von *features* ausnutzend, in den betreffenden *class*-Texten teilweise sehr viel einfachere Verfahren für die Berechnung dieser Größen notiert. Werden beim "tuer_vergleich" nun in jedem Fall die komplizierteren Routinen der *class* VIERECK angewandt oder im Einzelfall die weniger aufwendigen Methoden einer der anderen Klassen?

Die Antwort ergibt sich aus der folgenden Regel:

> Sei A eine *class* mit den *features* fA_1, ..., fA_n und A_1 ein Nachkomme von A. In A_1 mögen die *features* fA_1, ..., fA_m ($m < n$) redefiniert sein. Es seien ferner "a_att" und "a1_att" Attribute vom (Klassen-)Typ A beziehungsweise A_1. Dann bewirkt die aufgrund der *Polymorphie* von "a_att" mögliche Zuweisung "a_att := a1_att", daß die *feature*-Anwendungen "a_att.fA_1", ..., "a_att.fA_m" auf die in A_1 redefinierten *features* zugreifen.

(Dabei ist natürlich stillschweigend vorausgesetzt, daß "a1_att" tatsächlich auf ein Objekt der *class* A_1 verweist, daß also das *feature* "a1_att.Create" irgendwann vorher angewandt wurde (vgl. die nachstehende Abbildung). Aus ihr geht

auch hervor, daß "a_att.f_x" nicht erlaubt ist, wenn das *feature* "f_x" nicht schon für Objekte der *class* A definiert wurde.)




In den Kontext unseres Spiels übertragen bedeutet dies: Abhängig davon, ob hinter der vom Spieler gewählten Tür eine Raute, ein Rechteck, ein Quadrat oder ein sonstiges Viereck steckt, werden dasjenige "umfang"-*feature* und dasjenige "flaeche"-*feature* ausgeführt, welche für die Klasse des hinter der Tür verborgenen Objekts definiert wurden. Verweist "vierecke.access(erste_ wahl)" auf ein RECHTECK-Objekt, so werden "umfang" und "flaeche" nach den einfachen Formeln für Rechtecke berechnet; verweist es dagegen auf ein allgemeines Viereck, so werden die in der *class* VIERECK beschriebenen und für beliebige Vierecke gültigen *features* benutzt. Entscheidend ist: Die Programmiererin braucht sich bei der Beschreibung der von ihrem Programm zu kontrollierenden Aktionen nicht selbst um die Wahl des jeweils für eine spezielle Datenstruktur geeignetsten Verfahrens zu kümmern; sie hat dies implizit bereits getan, als sie Nachfolger-Klassen definierte, *features* redefinierte und sich der *Polymorphie* bediente.

Wir fassen zusammen: *Polymorphie* erlaubt es, an ein Attribut Objekte zu *binden*, deren Typ nicht notwendig mit dem (im Programmtext deklarierten) Typ des Attributs übereinstimmt, sondern zu diesem in der Nachkommens-Beziehung steht. Diese Bindung ist *dynamisch* in dem Sinne, daß zu verschiedenen Zeiten während des Programmlaufs (abhängig vom Zufall und/oder von Interaktionen des Rechners mit seiner Umgebung) Objekte verschiedenen Typs von ein und demselben Attribut referenziert werden können. *Dynamische Bindung* (engl. *dynamic binding*) sorgt (ohne daß eine weitere Intervention des Programmierers nötig ist) dafür, daß jeweils die *features* eines Objekts zur Anwendung kommen, die der speziellen Klasse dieses Objekts entsprechen.

Wir wollen Monika nicht vergessen, unsere zweite Programmiererin, die MODULA-2 Expertin. Wir sollten uns fragen, wie sie die Aufgabe bewältigen würde, welche Evi in EIFFEL so elegant erledigt hat. Wir nehmen an, daß Monika ihrer Kollegin über die Schulter geschaut hat und nun den Ehrgeiz hat, ihrer-

seits mit einer unter allen softwaretechnischen Wassern gewaschenen Lösung aufzuwarten. Wie Evi so wird auch Monika in einem Katalog von Moduldefinitionen nachschauen, um dort vielleicht einen passenden "Viereckmodul" zu finden. Tatsächlich entdeckt sie einen vor längerer Zeit von einem Kollegen verfaßten ADT-Modul, der alles enthält, was ihr Herz begehrt, jedoch ... - und darin unterscheidet sich ihre Situation zunächst nicht von der ihrer Kollegin - nur für "allgemeine Vierecke":

```
DEFINITION MODULE Viereck;
TYPE ViereckTyp;
TYPE VeParamTyp = ARRAY [0 ... 7] OF REAL;
PROCEDURE Create(VAR param: VeParamTyp; VAR ve: ViereckTyp);
PROCEDURE Ecken(ve: ViereckTyp; VAR ecken: VeParamTyp);
PROCEDURE Umfang(ve: ViereckTyp): REAL;
PROCEDURE Flaeche(ve: ViereckTyp): REAL;
PROCEDURE Verschiebe(ve: ViereckTyp; a,b: REAL)
END Viereck.
```

Sie hat natürlich sofort die Idee, diesen Modul zu nehmen und sowohl seine Definition als auch seine Implementierung so "aufzubohren", daß er in der dann von ihr neu erzeugten Version die geeignetsten Routinen für alle benötigten Spezialfälle (wie Raute, Rechteck und Quadrat) enthält. Leider erfährt sie, daß der Modul "Viereck" von vielen Anwendungsprogrammen ausgiebig benutzt wird, und daß es nicht angeht, seine Definition so mir nichts dir nichts zu ändern. Es wäre nicht abzusehen, welche Auswirkungen dies auf zukünftige Versionen der Anwendungsprogramme hätte. Sie wird aufgefordert, den Modul "Viereck" doch bitte nicht anzurühren. So entschließt sie sich, um wenigstens nicht ganz bei Null aufsetzen zu müssen (eingedenk dessen, was ihr zum Thema *Wiederverwendbarkeit* gelehrt wurde), eine Kopie der beiden Teile des Moduls "Viereck" herzustellen, diese dann umzubenennen und - wie schon gesagt - "aufzubohren". Hier ist das Ergebnis ihrer Arbeit:

```
DEFINITION MODULE ViereckNeu;
TYPE ViereckTyp;
TYPE ArtTyp = (allgemein, raute, rechteck, quadrat);
TYPE VeParamTyp = ARRAY [0 ... 7] OF REAL;
PROCEDURE Create(art: ArtTyp; VAR param: VeParamTyp;
                 VAR ve: ViereckTyp);
(* "param" ist entsprechend der gewünschten Viereckart zu belegen *)
PROCEDURE Ecken(ve: ViereckTyp; VAR ecken: VeParamTyp);
PROCEDURE Umfang(ve: ViereckTyp): REAL;
PROCEDURE Flaeche(ve: ViereckTyp): REAL;
PROCEDURE Verschiebe(ve: ViereckTyp; a,b: REAL)
END ViereckNeu.
```

Die für die verschiedenen Arten jeweils günstigen Routinen zur Erzeugung von Vierecken und zur Berechnung von Fläche und Umfang wird sie in der Implementierung der entsprechenden Prozeduren unterbringen. Wir geben im folgenden die wesentlichen Teile ihres Implementierungsmoduls wieder, ohne jedoch auf Einzelheiten der hier gewählten Repräsentation von Vierecken näher einzugehen.

```
IMPLEMENTATION MODULE ViereckNeu;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
TYPE ViereckTyp = POINTER TO ViereckRepTyp;
TYPE ViereckRepTyp = RECORD
                        veArt: ArtTyp;
                        veEcken: VeParamTyp
                     END;
PROCEDURE Create(art: ArtTyp; VAR param: VeParamTyp;
                 VAR ve: ViereckTyp);
(* lokale Deklarationen *)
BEGIN
   NEW(ve); ve^.veArt := art;
   CASE art OF
      allgemein: (* Berechnung von veEcken für den allgemeinen Fall *)|
      raute: (* Berechnung von veEcken für den Fall Raute *)|
      rechteck: (* Berechnung von veEcken für den Fall Rechteck *)|
      quadrat: (* Berechnung von veEcken für den Fall Quadrat *)
   END (*CASE*)
END Create;
PROCEDURE Ecken(ve: ViereckTyp; VAR ecken: VeParamTyp);
(* lokale Deklarationen *)
BEGIN
   (* ve^.veEcken wird in "ecken" übergeben *)
END Ecken;
PROCEDURE Umfang(ve: ViereckTyp): REAL;
(* lokale Deklarationen *)
BEGIN
   CASE ve^.veArt OF
      allgemein: (* Berechnung von Umfang für den allgemeinen Fall *)|
      raute: (* Berechnung von Umfang für den Fall Raute *)|
      rechteck: (* Berechnung von Umfang für den Fall Rechteck *)|
      quadrat: (* Berechnung von Umfang für den Fall Quadrat *)
   END (*CASE*);
   RETURN ...
END Umfang;
```

```
PROCEDURE Flaeche(ve: ViereckTyp): REAL;
(* lokale Deklarationen *)
BEGIN
   CASE ve^.veArt OF
      allgemein: (* Berechnung von Flaeche für den allgemeinen Fall *)|
      raute: (* Berechnung von Flaeche für den Fall Raute *)|
      rechteck: (* Berechnung von Flaeche für den Fall Rechteck *)|
      quadrat: (* Berechnung von Flaeche für den Fall Quadrat *)
   END (*CASE*);
   RETURN ...
END Flaeche;

PROCEDURE Verschiebe(ve: ViereckTyp; a,b: REAL)
(* lokale Deklarationen *)
BEGIN
   (* Verschiebe ve^.veEcken um a horizontal, b vertikal *)
END Verschiebe;

END ViereckNeu.
```

Wir beobachten an dieser Stelle, daß die Fallunterscheidungen in den obigen Prozeduren nun die ganze "Intelligenz" enthalten, die wir zuvor mit Hilfe der EIFFEL eigenen Vererbungsmechanismen erreicht haben.

Vom Modul "EinSpiel" soll uns hier nur der Implementationsteil interessieren:

```
IMPLEMENTATION MODULE EinSpiel;
FROM ViereckNeu IMPORT  ViereckTyp, ArtTyp, VeParamTyp,
                        Create, Ecken, Umfang, Fläche;
CONST VeLowC = 0; VeHighC = 9; ZzC = 3;
VAR vierEcke: ARRAY [LowC ... HighC] OF ViereckTyp;
    ersteWahl, zweiteWahl: [LowC ... HighC];

PROCEDURE VeCreate;
VAR ind: [VeLowC ... VeHighC]; z: [0 ... ZzC];
    art: ArtTyp; param: VeParamTyp;
BEGIN
   FOR  ind := VeLowC TO  VeHighC  DO
      (* erzeuge Zufallszahl z aus {0, ..., ZzC} *)
      CASE z OF
         0: art := allgemein;
            (* erzeuge zufälligen Inhalt von "param" für allg. Viereck *)|
         1: art := raute;
            (* erzeuge zufälligen Inhalt von "param" für Raute *)|
         2: art := rechteck;
            (* erzeuge zufälligen Inhalt von "param" für Rechteck *)|
```

```
            3: art := quadrat;
               (* erzeuge zufälligen Inhalt von "param" für Quadrat *)
         END (* CASE *);
         Create (art, param, vierEcke[ind])
      END (* FOR *)
   END VeCreate;

   PROCEDURE TuerAuswahl;
   (* lokale Deklarationen *)
   BEGIN
      (* entsprechend der Beschreibung des Spiels *)
   END TuerAuswahl;

   PROCEDURE TuerVergleich: BOOLEAN;
   BEGIN
      RETURN  (Umfang(vierEcke[ersteWahl])
               > Umfang(vierEcke[zweiteWahl]))
               AND
               (Flaeche(vierEcke[ersteWahl])
               > Flaeche(vierEcke[zweiteWahl]));
   END TuerVergleich;

   END EinSpiel.
```

Vergleichen wir nun die beiden Lösungen, die in MODULA-2 geschriebene mit der in EIFFEL verfaßten. Zunächst ist festzustellen, daß es dem (MODULA-) Modul "EinSpiel" als Kunde von "ViereckNeu" offenbar nicht schlechter geht als der (EIFFEL-)*class* EIN_SPIEL. Auch ihm wird versprochen, daß die jeweils angemessenen Routinen für die Erzeugung der jeweiligen Vierecke ablaufen, und daß die jeweils besten Verfahren zur Berechnung von Fläche und Umfang zur Anwendung gelangen. Er braucht sich um die Auswahl dieser Verfahren jedenfalls nicht selbst zu kümmern. Insofern sehen wir also "auf der Kundenseite" keinen Nachteil für die MODULA-2-Lösung: Der Kunde ("EinSpiel") ist unabhängig von der vom Diensterbringer ("ViereckNeu") gewählten Implementierung.

Etwas anders sieht es auf der Seite des Diensterbringers aus. Wir vergegenwärtigen uns noch einmal Monikas Schwierigkeiten: Zunächst einmal konnte sie den existierenden Modul "Viereck" nicht ohne weiteres wiederverwenden, da die Arbeit an ihm bereits abgeschlossen worden war. Sie mußte zunächst eine Kopie anfertigen und ihr einen neuen Namen geben. Doch die Art und Weise, wie sie diese Kopie dann "wiederverwendet" hat, verdient die Bezeichnung *Wiederverwendung* eigentlich nicht. Tatsächlich mußte sie, um ihre Teile hinzufügen zu können, die Details der ihr vorliegenden Implementierung - einschließlich der dafür gewählten Repräsentation von Vierecken - genau verstehen, und sie mußte die Repräsentation ändern. Sie hatte also viel und - wie man hinzufügen muß - fehleranfällige Arbeit!

Natürlich hätte sie von vornherein ganz anders vorgehen können. Zusätzlich zum bereits vorhandenen Modul "Viereck" hätte sie drei weitere Module "Raute", "Rechteck" und "Quadrat" mit den entsprechenden Dienstleistungen definieren, implementieren und dem "Kunden"-Modul "EinSpiel" anbieten können. Doch hätte sie damit nicht nur sich selbst etwa genauso viel Arbeit aufgeladen wie zuvor, auch die Implementierung des Kunden-Moduls hätte sie beträchtlich erschwert. Denn nun wäre der "Kunde" nicht nur dafür verantwortlich, genau darüber "Buch zu führen", welche Art von Viereck er hinter welcher Tür versteckt, er muß auch - über umfängliche Fallunterscheidungen - die Auswahl des für die jeweilige Viereckssart günstigsten Berechnungsverfahren selbst vornehmen, wie der folgende Programmtext zeigt:

```
IMPLEMENTATION MODULE EinSpiel;
IMPORT Viereck, Raute, Rechteck, Quadrat;
(* Alles wird importiert, was von diesen Modulen exportiert wird. *)
TYPE ArtTyp = (allgemein, raute, rechteck, quadrat);
     ViereckSpielTyp = RECORD
                         CASE veArt: ArtTyp OF
                           allgemein: ve: Viereck.ViereckTyp|
                           raute: ra: Raute.RauteTyp|
                           rechteck: re: Rechteck.RechteckTyp|
                           quadrat: qu: Quadrat.QuadratTyp
                         END
                       END (*RECORD*);
CONST VeLowC = 0; VeHighC = 9; ZzC = 3;
VAR vierEcke: ARRAY [LowC ... HighC] OF ViereckSpielTyp;
    ersteWahl, zweiteWahl: [LowC ... HighC];

PROCEDURE VeCreate;
VAR ind: [VeLowC ... VeHighC]; z: [0 ... ZzC];
    veParam: Viereck.ParamTyp; raParam: Raute.ParamTyp;
    reParam: Rechteck.ParamTyp; quParam: Quadrat.ParamTyp;
BEGIN
  FOR ind := VeLowC TO VeHighC DO
    (* erzeuge Zufallszahl z aus {0, ..., ZzC} *)
    CASE z OF
      0: (* erzeuge zufälligen Inhalt von "veParam" *)
         vierEcke[ind].veArt := allgemein;
         Viereck.Create(veParam, vierEcke[ind].ve)|
      1: (* erzeuge zufälligen Inhalt von "raParam" *)
         vierEcke[ind].veArt := raute;
         Raute.Create(raParam, vierEcke[ind].ra)|
      2: (* erzeuge zufälligen Inhalt von "reParam" *)
         vierEcke[ind].veArt := rechteck;
```

```
            Rechteck.Create(reParam, vierEcke[ind].ve)|
          3: (* erzeuge zufälligen Inhalt von "quParam" *)
            vierEcke[ind].veArt := quadrat;
            Quadrat.Create(quParam, vierEcke[ind].ve)|
        END (* CASE *)
      END (* FOR *)
    END VeCreate;

    PROCEDURE TuerAuswahl;
    (* lokale Deklarationen *)
    BEGIN
      (* entsprechend der Beschreibung des Spiels *)
    END TuerAuswahl;

    PROCEDURE TuerVergleich: BOOLEAN;
    VAR umfang1, umfang2, flaeche1, flaeche2: REAL;
    BEGIN
      CASE vierEcke[ersteWahl].veArt OF
        allgemein: umfang1 := Viereck.Umfang(vierEcke[ersteWahl].ve);
                   flaeche1 := Viereck.Flaeche(vierEcke[ersteWahl].ve)|
        raute:     umfang1 := Raute.Umfang(vierEcke[ersteWahl].ra);
                   flaeche1 := Raute.Flaeche(vierEcke[ersteWahl].ra)|
        rechteck:  umfang1 := Rechteck.Umfang(vierEcke[ersteWahl].re);
                   flaeche1 := Rechteck.Flaeche(vierEcke[ersteWahl].re)|
        quadrat:   umfang1 := Quadrat.Umfang(vierEcke[ersteWahl].qu);
                   flaeche1 := Quadrat.Flaeche(vierEcke[ersteWahl].qu)
      END (*CASE*);
      CASE vierEcke[zweiteWahl].veArt OF
        allgemein: umfang2 := Viereck.Umfang(vierEcke[zweiteWahl].ve);
                   flaeche2 := Viereck.Flaeche(vierEcke[zweiteWahl].ve)|
        raute:     umfang2 := Raute.Umfang(vierEcke[zweiteWahl].ra);
                   flaeche2 := Raute.Flaeche(vierEcke[zweiteWahl].ra)|
        rechteck:  umfang2 := Rechteck.Umfang(vierEcke[zweiteWahl].re);
                   flaeche2 := Rechteck.Flaeche(vierEcke[zweiteWahl].re)|
        quadrat:   umfang2 := Quadrat.Umfang(vierEcke[zweiteWahl].qu);
                   flaeche2 := Quadrat.Flaeche(vierEcke[zweiteWahl].qu)
      END (*CASE*);
      RETURN (umfang1 > umfang2) AND (flaeche1 > flaeche2);
    END TuerVergleich;

    END EinSpiel.
```

Nach dem Motto "Der Kunde ist König" ist dies ohne Zweifel eine nicht zu akzeptierende Lösung. Doch auch mit Blick auf die erste, von Monika tatsächlich realisierte Version halten wir fest: Vom Standpunkt der *Wiederverwendbarkeit*

ist dem durch den Vererbungsmechanismus ermöglichten EIFFEL-Programm uneingeschränkt der Vorzug zu geben.

Dies gilt übrigens auch, wenn wir die beiden (EIFFEL- und MODULA-) Programme hinsichtlich eines zweiten, für die Ökonomie des *Software Engineering* nicht weniger entscheidenden Qualitätsmerkmals, der *Adaptierbarkeit*, vergleichen. Allein das Konzept der Vererbung erlaubt, wie wir gesehen haben, die Anpassung existierender Software (in der Gestalt von *class*-Modulen) an neue und ursprünglich nicht eingeplante Aufgabenstellungen, ohne daß diese Software dabei selbst geändert werden müßte. Dies ist, wie sich der Leser bewußt machen möge, eine unmittelbare Konsequenz aus der Art und Weise wie die Nachkommen einer *class* erzeugt werden, nämlich durch den Bezug (in einer *inherit*-Klausel) auf die Elter-*class*, sowie durch die Erweiterung um neue und die Redefinition von alten *features*. Solche Anpassung geschieht natürlich nie zum Selbstzweck, sondern wird zum Beispiel dann notwendig, wenn im Rahmen einer bestimmten Anwendung neue Anforderungen gestellt werden (welche für die Programme, die diese Anwendung realisieren, entsprechende Änderungen erforderlich machen).

Konkret: Für unser Spiel möge verlangt werden, daß außer allgemeinen Vierecken, Rauten, Rechtecken und Quadraten nun auch Trapeze hinter den Türen versteckt werden sollen. Für die EIFFEL-Programmiererin läuft die Erfüllung dieses Wunsches auf die Erzeugung eines weiteren Nachkommens (*class* TRAPEZ) der *class* VIERECK und auf eine ziemlich triviale Modifikation der *class* EIN_SPIEL hinaus. Es versteht sich (fast) von selbst, daß auch diese Modifikation durch Vererbung erledigt wird: Sie schreibt eine *class* EIN_SPIEL_NEU, welche bis auf das Create-*feature* alles von EIN_SPIEL erbt. (Der Leser möge sich in Evis Lage versetzen!) Entscheidend ist: An bestehenden Texten wird nichts geändert (es werden nur Texte hinzugefügt)! Andere Anwendungen, die sich auf diese Texte stützen, werden nicht betroffen.

Ganz anders müßte Monika, die MODULA-Programmiererin, an diese einfache Aufgabe herangehen. Sie müßte die entsprechenden Erweiterungen direkt in ihren Modulen "ViereckNeu" und "EinSpiel" anbringen. Falls nun "ViereckNeu" (und vielleicht sogar "EinSpiel") inzwischen von anderen Programmen (auch anderer Verfasser) benutzt werden, so steht sie vor exakt dem gleichen Problem, mit dem sie schon bei der Lösung der ursprünglichen Aufgabe konfrontiert war: Sie muß, im schlechtesten Fall, beide Module neu schreiben, um die Arbeit ihrer Kollegen nicht zu stören. *Wiederverwendung* ist hier auf das bloße Kopieren und (mehr oder weniger das) Studieren von Programmteilen reduziert.

Vererbung muß nicht auf einen einzigen Vorfahr beschränkt sein. (Fast könnte man sagen: Wie im wirklichen Leben!) Eine *class* kann durchaus mehrere Eltern haben. Man nennt diese Möglichkeit *Mehrfachvererbung* (engl. *multiple inheritance*). Werden bei der Definition einer *class* C in ihrer *inherit*-Klausel zwei

Eltern, A und B, genannt, so übernimmt C alle *features* von A und B. Natürlich braucht sie nicht alle zu exportieren. Sie kann, wie von der "einfachen" Vererbung her bekannt, *features* redefinieren und *features* neu hinzufügen, und sowohl bei der einen als auch bei der anderen Modifikation kann sie die originalen und die redefinierten *features* von A und B selbstverständlich benutzen. Wir wollen die Vorteile der *Mehrfachvererbung* an einem Beispiel studieren, anhand dessen wir gleichzeitig den Nutzen eines weiteren Konzepts erkennen werden, welches die Definition kompilierbarer Klassen erlaubt, ohne daß in diesen jedoch schon sämtliche Details der Implementierung ausgearbeitet sein müßten. (Wir lösen damit ein unter Bemerkung (iv) - *interface* - Abstraktion - gegebenes Versprechen ein.)

Dieses Konzept heißt *deferred class* (*deferred* = *aufgeschoben, zurückgestellt*). Eine solche *class* enthält neben voll ausformulierten *features* auch solche, von denen nur der Name bekannt ist, deren Implementierung aber den Erben dieser *class* überlassen wird! (Man sagt: Die Implementierung wird - bis auf weiteres - *zurückgestellt*, *deferred*.) Als Beispiel sei hier die uns schon bekannte *class* STACK[T] (vgl. Seite 396) als *deferred class* notiert:

```
deferred class STACK[T] export
    anz_elemente, push, pop, top, isnew, replace, isfull
feature
    anz_elemente: INTEGER is
        deferred
        end;
    push (x: INTEGER) is
        require
            not isfull
        deferred
        ensure
            not isnew; top = x;
            anz_elemente = old anz_elemente + 1
        end;
    pop is
        deferred
        ensure
            (not old isnew and (anz_elemente = old anz_elemente - 1))
            or (old isnew and Nochange)
        end;
    top: T is
        require
            not isnew
        deferred
        end;
```

```
    isnew: BOOLEAN
      do
        Result := (anz_elemente = 0)
      ensure
        Result = (anz_elemente = 0)
      end;
    replace (x: T)
      do
        pop; push(x)
      ensure
        top = x and
        ((not old isnew and Nochange)
        or (old isnew and (anz_elemente = old anz_elemente + 1)))
      end;
    isfull: BOOLEAN is
      deferred
      end;
  invariant
    anz_elemente >= 0
  end -- class STACK[T]
```

Bis auf "isnew" und "replace" sind alle *features deferred*, das heißt der *do*-Teil dieser *features* ist durch das Schlüsselwort *deferred* ersetzt. Im übrigen enthält der Text dieser *class* sämtliche *require*-, *ensure*- und *invariant*-Prädikate, und damit alles was wir auch in einer *class interface* Abstraktion ausgedrückt hätten. (In der Tat: Wäre das Wörtchen *deferred* nicht und wären nicht einige *features* voll beschrieben, so würde sich diese *class* nicht wesentlich von ihrer *interface*-Abstraktion unterscheiden.)

Wie gesagt: *deferred* Klassen sind compilierbar, sie können also in der "Programmierumgebung" verfügbar gemacht werden, und man kann folglich von ihnen erben. Nur eins hat eine *deferred class* nicht: Objekte! Die Create-Operation existiert nicht für eine solche Klasse. Und das ist auch gut so, denn wozu sollte ein Objekt taugen, das *features* (also Eigenschaften) hat, von denen man nur weiß, daß sie existieren, sonst aber nichts? Ein Erbe kann *deferred features* implementieren (sie - wie man sagt - *effektiv* machen), muß es aber nicht. (Enthält auch er *deferred features*, so ist auch er eine *deferred class*. Natürlich sollten irgendwann alle *features effektiv* gemacht werden.) Und natürlich darf ein Nachkomme *features*, die er von einer *deferred class* erbt, neu definieren (unter Beachtung der gegebenen Vor- und Nachbedingungen, versteht sich). "Aufgeschobene" Klassen bilden somit Vorlagen für die "Implementierung durch Vererbung". Dabei können, wie wir sogleich am Beispiel von STACK und ARRAY sehen werden, mehrere Eltern bei der Erzeugung des "lebensfähigen" (d.h. *effektiven*) Nachkommen beteiligt sein.

Mit Hilfe der uns bereits bekannten *class* ARRAY[T] (vgl. Seite 398) implementieren wir den als *deferred class* definierten STACK[T] als "beschränkten Stack" (der gebotenen Kürze halber verzichten wir hier auf die nochmalige Wiedergabe der diversen Prädikate von STACK[T]). Der Leser vergleiche diese Implementierung mit unserer früheren, die auf der üblichen "Kunde-Verkäufer-Beziehung" zwischen Modulen beruhte.

```
class BOUNDED_STACK[T] export
   anz_elemente, push, pop, top, isnew, replace, isfull
inherit
   STACK[T] redefine replace;
   ARRAY[T] rename Create as array_Create
feature
   Create(n: INTEGER) is
      do
         array_Create(1, n)
      end;
   anz_elemente: INTEGER;
   push (x: INTEGER) is
      do
         anz_elemente := anz_elemente + 1;
         assign(anz_elemente, x)
      end;
   pop is
      do
         if anz_elemente > 0 then
            anz_elemente := anz_elemente - 1
         end
      end;
   top: T is
      do
         Result := access(anz_elemente)
      end;
   replace (x: T)
      do
         assign(anz_elemente, x)
      end;
   isfull: BOOLEAN is
      do
         Result := (anz_elemente = size)
      end;
end -- class BOUNDED_STACK[T]
```

Einige Bemerkungen hierzu:

1. Die bereits in der *deferred class* STACK[T] *effektive* Version des *features* "replace" wird redefiniert, da es sich in der ARRAY-Implementierung effizienter ausdrücken läßt (durch nur eine Operation anstatt durch die Hintereinanderschaltung von zwei Operationen).

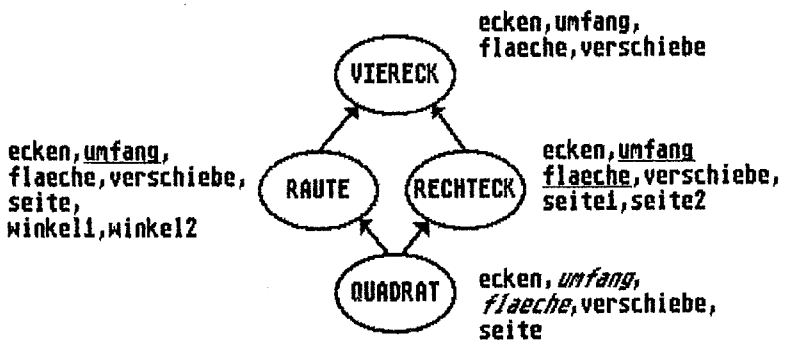
2. Da es, wie im vorliegenden Fall, nützlich sein kann, das Create-*feature* eines "Elternteils" z.B. im Rahmen der Ausarbeitung des Create-*features* des Nachkommen zu verwenden, ist es erlaubt, das "Create" eines Vorfahren innerhalb einer *rename* - Klausel geeignet umzubenennen.

3. Wird ein parameterloses und wertlieferndes *feature* - in unserem Beispiel "anz_elemente" - in der *deferred class* als *deferred* deklariert, so hat man bei der Definition einer Nachkommens-Klasse die Freiheit, dieses *feature* entweder als Attribut oder als Routine zu realisieren. Wir haben hier die erste Möglichkeit gewählt.

4. Natürlich taucht das *feature* "isnew" im Text von BOUNDED_STACK[T] nicht mehr explizit auf, da es bereits in STACK[T] *effektiv* definiert ist und im Nachfolger nicht redefiniert wird.

5. Der Vergleich mit unserer ersten Implementierung von STACK[T], bei der wir Stacks durch ein Attribut vom Klassen-Typ ARRAY[T] dargestellt haben, zeigt, daß die mittels der Vererbungsmechanismen entstandene Version nicht nur klarer, sondern auch notationell einfacher ist. In ihr stehen uns ja alle *features* von ARRAY[T] direkt, ohne den Umweg über ein speziell eingeführtes Attribut zur Verfügung, und sie können daher auch direkt benannt werden.

Wir kehren, bevor wir diesen Abschnitt beschließen, noch einmal zu unseren geometrischen Objekten zurück, und zwar zunächst zum QUADRAT. Als wir diese Klasse definierten, zögerten wir kurz ob der Wahl zwischen RAUTE und RECHTECK als Elter-*class*. Wir entschieden uns für RECHTECK, obwohl uns die Umfangsberechnung für Objekte von RAUTE vielleicht doch sympathischer gewesen wäre. Mit unserem neuen Wissen über *Mehrfachvererbung* sollte es möglich sein, RAUTE als Nachkomme von beiden Klassen zu erzeugen, um so etwa die Berechnung der Fläche von RECHTECK und die des Umfangs von RAUTE zu erben. Das einzige Problem dabei ist, daß beide Klassen ein und denselben Vorfahr (VIERECK) haben, und daß einige ihrer *features* redefiniert und gleichnamig sind (vgl. Abbildung).



Dieses Phänomen ist unter der Bezeichnung *wiederholte Vererbung* (engl. *repeated inheritance*) bekannt. Wenn QUADRAT sowohl von RAUTE als auch von RECHTECK erbt, so gibt es *features* (hier: "ecken" und "verschiebe") des gemeinsamen "Urahnen" VIERECK, die über beide Elternteile unverändert und ohne Umbenennung auf den Nachkommen übertragen werden. EIFFEL hat für diesen Fall eine einfache Konvention: Derartige (von beiden Eltern geerbte) *features* brauchen im Nachkommen nicht umbenannt zu werden, falls man sie nicht (aus welchen Gründen auch immer) als verschieden ansehen will. Anders verhält es sich mit *features*, die, obwohl in den Elter-Klassen gleichnamig, doch irgendwo im "Klassen-Stammbaum" umdefiniert wurden. In unserem Beispiel sind dies "umfang" und "flaeche", die - obwohl verschieden implementiert - in RAUTE und RECHTECK genauso heißen. Dies ist ein *Namenskonflikt*, der mit Hilfe der schon bekannten *rename* - Klausel aufgelöst wird. Da QUADRAT nun "umfang" von RAUTE und "flaeche" von Rechteck übernehmen möchte, müssen in der *inherit* - Klausel das RECHTECK-*feature* "umfang" und das RAUTE-*feature* "flaeche" umbenannt werden (siehe die Abbildung auf der vorigen Seite). Es ergibt sich die folgende Klassendefinition:

```
class QUADRAT export
    ecken, umfang, flaeche, verschiebe, seite
inherit
    RECHTECK
        rename umfang as re_umfang;
    RAUTE
        rename flaeche as ra_flaeche
        redefine seite;
feature
    Create(p1, p2: POINT) is
        do
            seite := p1.abstand(p2);
            seite1 := seite;  seite2 := seite;
            -- Berechnung der Eckpunkte
        end;
    seite: REAL;
invariant
    -- die Invarianten der Klassen RECHTECK
    -- und RAUTE gelten
end  -- class QUADRAT
```

Die in der *export* - Klausel genannten *features* "umfang" und "flaeche" beziehen sich jetzt offenbar eindeutig auf die entsprechenden *features* der Klassen RAUTE und RECHTECK. Wir haben uns hier außerdem noch die Freiheit genommen, das von RAUTE übernommene *feature* "seite" als Attribut zu redefinieren, da wir dessen Wert ja bereits durch die Create-Operation bereitstellen.

Auf einem Vehikel namens EIFFEL sind wir damit am Ende unserer Parforce-Tour durch das Reich der *objektorientierten Programmierung* angelangt. Wir wollen es an diesem Schluß nicht versäumen, den Erbauer des Vehikels selbst zu Wort kommen zu lassen, der mit einigem und - wie wir meinen - nicht unberechtigtem Stolz behauptet, daß man mit diesem Gefährt bis in den "siebenten Himmel" der objektorientierten Glückseligkeit vorstoßen kann. Er unterscheidet auf diesem Weg die folgenden Etappen (oder "Himmel niedrigerer Stufen", vgl. [MEY], Abschnitt 4.9 "*Seven steps towards object-based happiness*"):

I. Die modulare Struktur von Software-Systemen orientiert sich an den Daten, welche für die jeweils zu behandelnden Objekte charakteristisch sind (*objektbasierte Modularisierung*).

II. Die Objekte können als Implementationen Abstrakter Datentypen beschrieben werden (*Datenabstraktion*).

III. Die Implementierung der Programmiersprache erlaubt es, Objekte nach Bedarf zu erzeugen und nicht mehr benutzte Objekte automatisch "aus dem Verkehr" ziehen zu lassen (*automatische Speicher-Verwaltung, automatische garbage collection*).

IV. Die sprachliche Repräsentation von Modulen erfolgt durch Klassen; Klassen beschreiben Objekte eines Typs; jeder nicht einfache Typ ist eine Klasse, und jede Klasse ist ein Typ (*Modul = Typ = Klasse*).

V. Klassen können als Erweiterungen oder Einengungen anderer (bereits vorhandener) Klassen definiert werden (*Vererbung*).

VI. Innerhalb eines Programms kann man sich unter ein und demselben Namen auf Objekte verschiedener Klassen beziehen, und über diesen Namen angeforderte Operationen werden in der für das jeweils referenzierte Objekt gültigen Implementation ausgeführt (*Polymorphie und dynamische Bindung*).

VII. Eine Klasse kann sowohl als Erbe mehrerer Klassen als auch mehrfach als Erbe derselben Klasse definiert werden (*Mehrfachvererbung und wiederholte Vererbung*).

Mit einer Sprache wie MODULA-2 kann man diesen Weg nur bis zur zweiten Etappe zurückzulegen. Sprachen wie ADA, die auch solch spezielle Techniken wie *overloading* und *Generizität* zulassen, führen im Prinzip nicht weiter. Natürlich gibt es allenthalben Versuche, die durch die Schritte III bis VII markierten Abschnitte mit solchen "nur" *modularen* Sprachen zu erreichen (mitunter sogar unter Zuhilfenahme von Erweiterungen und anderer Modifikationen der Sprache selbst). Mangels genuiner Konstrukte wirken derartige Versuche jedoch etwas verkrampft, wenig zwingend und, sagen wir es ruhig, "an den Haaren herbeigezogen". Wir haben daher auf ihre Darstellung verzichtet und es vorgezogen, die Konzepte der *objektorientierten Programmierung* anhand einer Sprache zu vermitteln, in der diese direkt und ohne Umschweife ausgedrückt werden können.

### 6.4.5 Parallele Prozesse

Unabhängig davon, ob wir Anhänger einer objektorientierten "Weltanschauung" sind oder nicht, behält die Modellierung von Realwelt-Systemen und - darauf basierend - die Spezifikation von Software-Systemen als Netzwerke kommunizierender Prozesse ihre überragende Bedeutung. Schließlich lehrt uns der vorige Abschnitt, daß wir die in einem System interagierenden Objekte als Prozesse auffassen können (und umgekehrt die Prozesse als Objekte). Und unabhängig davon, ob wir Prozesse als Objekte betrachten oder nicht, haben wir uns bisher darauf beschränkt, sie in eine Umgebung einzubetten, die keinen *parallelen* (d.h. gleichzeitigen) Ablauf verschiedener und weitgehend unabhängiger Vorgänge erlaubt. Vielmehr mußte der Programmierer diese Vorgänge selbst in eine festgefügte sequentielle Ordnung bringen und sie damit für die Bearbeitung durch einen einzigen Prozessor zugänglich machen. (Wir nannten diese Umgebung "Monoprozessor ohne Multitasking".) So mühsam dies auch gelegentlich sein mochte, es ersparte ihm eine Reihe haariger Probleme, die es zu lösen gilt, wenn im Prinzip parallele Prozesse auch parallel ablaufen sollen. Diesen Problemen ist der vorliegende letzte Abschnitt dieses Buches gewidmet. In Auswahl und Darstellung orientiert er sich teilweise an einem (immer noch aktuellen und) ausgezeichneten Übersichtsartikel von G.R. Andrews und F.B. Schneider ([ANS]) aus dem Jahre 1983.

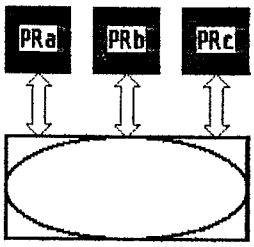
Wir rufen uns zunächst die zu Beginn von 6.4 abgebildete Prozeßkette und unsere dortige Bemerkung über verschiedene mögliche Basismaschinen für deren Implementierung ins Gedächtnis zurück:

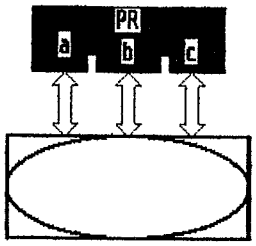

Unter bestimmten Voraussetzungen, so hieß es, könnten wir die Prozesse P_1, P_2 und P_3 in das Betriebssystem des Rechners einbetten und dann - unter der Aufsicht des Betriebssystems - gewissermaßen sich selbst überlassen. Einmal gestartet würden sie parallel ablaufen, und die zwischen ihnen erforderliche und in den entsprechenden Programmen zu beschreibende Kommunikation würde durch geeignete Mechanismen des zugrundeliegenden Betriebssystems geregelt werden.

Wir bezogen uns dabei sowohl auf den Aufbau der Rechner-Hardware, als auch auf die Art und Weise, wie Prozessoren ihre Arbeit zugeteilt werden kann. Wir trafen Unterscheidungen, die es nun etwas genauer darzustellen und zu analysieren gilt.

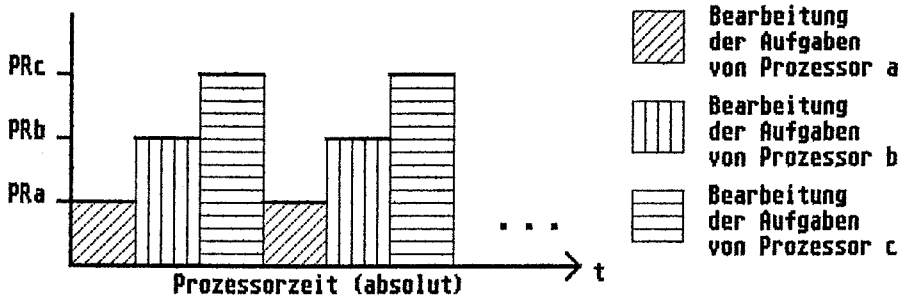
Wir betrachten als erstes eine Rechner-Architektur, die durch das Vorhandensein mehrerer gleichberechtigter und voneinander unabhängig programmierbarer Prozessoren charakterisiert ist, welche auf einen allen gleichermaßen zugäng-

lichen Speicher zugreifen, sich diesen also teilen (vgl. Abbildung). Durch Schreiben in und Lesen von diesem Speicher können sie sich gegenseitig, ihrer Programmierung gemäß, zum Beispiel über den Zustand der von ihnen bearbeiteten Prozesse informieren und/oder Daten untereinander austauschen (etwa im Sinne eines Kanals). Dies ist die einzige Verbindung zwischen den Prozessoren. Die Betriebsart eines solchen Rechners wird durch den Begriff *Multiprocessing* mit gemeinsamem Speicher (engl. *shared memory*) beschrieben. Wir gehen davon aus - und dies ist für alle folgenden Überlegungen von entscheidender Bedeutung -, daß die Prozessoren nicht "chaotisch" auf den allgemeinen Speicher zugreifen können. Vielmehr nehmen wir an, daß es einen (z.B.) auf der Ebene der Hardware implementierten Mechanismus gibt, der dafür sorgt, daß zu jedem Zeitpunkt nur genau ein Prozessor eine *elementare Speicheroperation* ausführen darf. Elementaroperationen sind das Schreiben in eine Speicherzelle und das Lesen aus einer Speicherzelle. Eine solche Elementaroperation ist durch den besagten Mechanismus *nicht unterbrechbar*. Man nennt sie daher auch *atomar*.



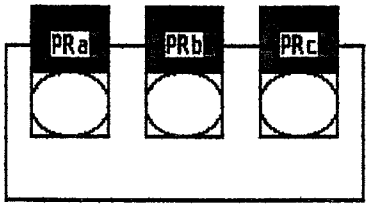




Multiprozessor-Systeme (wie PRa, PRb und PRc oben) kann man sehr leicht auch auf einem einzigen Prozessor PR (*Monoprozessor*) simulieren. Man muß diesen Prozessor nur abwechselnd die Arbeit von PRa, PRb und PRc tun lassen. Die Zeit von PR wird also auf die Aufgaben der zu simulierenden Prozessoren aufgeteilt. Man nennt diese Betriebsart *Multiprogramming* durch *time sharing*. Sie wird durch das folgende Diagramm illustriert:




In den jeweiligen Zeitintervallen verhält sich PR wie PRa, PRb und PRc, denen dann allerdings der Speicher jeweils ausschließlich zur Verfügung steht. PRx kann, wenn er "an der Reihe" ist, möglicherweise eine größere Anzahl elementarer Speicheroperationen in Folge ausführen. Darüber jedoch, ob beim Ablauf eines Zeitintervalls eine aus mehreren solcher atomarer Operationen bestehende nicht-elementare Operation unterbrochen wird oder nicht, können wir keinerlei Aussage machen. Auch hinsichtlich dieser Unsicherheit gibt es daher keinen wesentlichen Unterschied zum Multiprozessor-System.

In einer dritten Konfiguration haben wir wieder mehrere Prozessoren mit ihnen zugeordneten lokalen Speichern, die über ein Netzwerk (von Leitungen) miteinander verbunden sind. Einen gemeinsam genutzten Speicher gibt es hier nicht. Eine derartige Konfiguration wird als *verteilte* Rechner-Architektur bezeichnet, und der (englische) Terminus Technicus für den Betrieb einer solchen Hardware-Struktur lautet *distributed processing*. Wir werden hierauf am Ende dieses Abschnitts nur mit einigen allgemeinen Bemerkungen eingehen können.




In der Praxis findet man natürlich nicht nur diese Architekturen und Betriebsarten in Reinkultur, sondern auch zahlreiche Mischformen. Es ist hier nicht der Ort, darauf auch nur in Andeutungen einzugehen. Der kundige Leser wird wissen (und dem unkundigen sei es gesagt), daß er sich in Lehrbücher über *Rechner-Architektur* (z.B. [GIL]) und *Rechner-Betriebssysteme* (z.B. [TAN]) vertiefen muß, wenn er mehr und Genaueres erfahren möchte. (Die genannten Werke sind nur stellvertretend für eine außerordentlich umfangreiche Literatur über diese Gebiete genannt.)

Die beiden ersten Szenarien lassen sich noch weiter idealisieren. Für das Multiprozessor-System und entsprechend für seine Simulation durch den Monoprozessor dürfen wir zum Beispiel annehmen, daß die Anzahl der Prozessoren im Prinzip beliebig ist, und daß uns, wenn es denn geboten scheint, immer noch ein Prozessor für die Übernahme weiterer Arbeiten zur Verfügung steht. Daraus ergibt sich eine Gleichsetzung von Prozessor und Prozeß. (Aus objektorientierter Sicht würde dies bedeuten, daß jedem Objekt sein eigener Prozessor und Speicherbereich zugeordnet werden. Natürlich läßt sich auch das dritte Szenario in dieser Weise verallgemeinern, doch mit ihm wollen wir uns, wie gesagt, kaum beschäftigen.)

```
parprogram ‹programm_name›;
‹globale Deklarationen›

   process ‹prozeß_1›;
   ‹lokale Deklarationen›
   begin ... end;
   ... ...
   process ‹prozeß_n›;
   ‹lokale Deklarationen›
   begin ... end;

end.
```

Mit dieser Begründung erlauben wir uns, den Text, mit dem wir den Ablauf paralleler Prozesse auf Rechnersystemen mit gemeinsam genutztem Speicher beschreiben, als eine Liste von *Prozeßdeklarationen* zu notieren, und diese als *paralleles Programm* zu bezeichnen. Die für alle oder jeweils mehrere dieser Prozesse gemeinsamen Speicher werden als globale Variable dargestellt. Über den syntaktischen Formalismus der Notation paralleler Programme selbst brauchen wir keine weiteren Worte zu verlieren. Er ist (für den inzwischen geübten Leser) selbsterklärend. Zur Semantik zunächst nur soviel: Der Start eines parallelen Programms

bedeutet, daß alle in seinem Text deklarierten Prozesse (im Prinzip) gleichzeitig beginnen. (Wir sagen "im Prinzip", denn bei jeder der von uns betrachteten Architekturen wird natürlich einer der Prozesse zuerst "an die Reihe" kommen, nur können wir nicht wissen, welcher es ist. Diese Unsicherheit oder - anders gesagt - diese Notwendigkeit, alle möglichen Reihenfolgen in Betracht ziehen zu müssen, modellieren wir als "Gleichzeitigkeit".) Auf die Einführung sprachlicher Mittel, welche geeignet sind, die Erzeugung von Prozessen durch Prozesse zu beschreiben, müssen wir im Rahmen dieses kurzen Überblicks allerdings verzichten (vgl. aber [ANS]).

Um ein erstes Gefühl für die Probleme zu bekommen, mit denen wir es zu tun haben, wenn - wie beschrieben - die Ausführung der elementaren Speicheroperationen zweier oder mehrerer Prozesse keiner vorhersehbaren zeitlichen Ordnung unterworfen ist, betrachten wir das folgende *parprogram*:

```
parprogram Verwirrung;
var x: integer initial (0);
  process P1;
    begin
      x := x + 1
    end {P1};
  process P2;
    begin
      x := x + 2
    end {P2};
end {Verwirrung}.
```

(Mit dem Zusatz *initial* (...) in der Deklaration einer Variablen wird dieser ein Anfangswert zugewiesen.) Welchen Wert enthält die Variable x nach Beendigung des parallelen Programms "Verwirrung"? Die von P1 und P2 vorgenommenen Operationen sind nicht elementar. Wir nehmen aber an, daß eine Operation der Art "x := x + a" als Folge elementarer Operationen ausgeführt wird:

```
LADE ({Inhalt von} x {in den Akkumulator})
ADDIERE (a {zum Inhalt des Akkumulators})
SPEICHERE ({den Inhalt des Akkumulators nach} x)
```

(Als "Akkumulator" bezeichnen wir hier ein für Rechnungen aller Art zu verwendendes Register der gegebenen Prozessoren.) Die den Prozessen P1 und P2 entsprechenden Folgen lauten also:

```
LADE_1 (x);   ADDIERE_1 (1);   SPEICHERE_1 (x);
LADE_2 (x);   ADDIERE_2 (2);   SPEICHERE_2 (x);
```

(Um anzudeuten, unter welchen Prozessen diese Operationen jeweils stattfinden, haben wir sie durch die Suffixe "_1" bzw. "_2" gekennzeichnet.) Was geschieht nun in einem Multiprozessor-System, dessen Prozessoren alternierend auf den

Speicher (lesend oder schreibend) zugreifen dürfen? In welche zeitliche Ordnung werden die Operationsfolgen gebracht, wie werden sie miteinander verzahnt? Unter den gegebenen Voraussetzungen gibt es offenbar zwei Möglichkeiten. Wir notieren sie in Tabellenform und vermerken dabei jeweils die Inhalte der Variablen x und der Akkumulatoren (abgekürzt "akk1" und "akk2") nach Ausführung der elementaren Operation.

I.

| Systemzeit | Operation | x | akk1 | akk2 |
|---|---|---|---|---|
| 1 | LADE_1(x) | 0 | 0 | |
| 2 | LADE_2(x) | 0 | | 0 |
| 3 | ADDIERE_1(1) | 0 | 1 | |
| | ADDIERE_2(2) | 0 | | 2 |
| 4 | SPEICHERE_1(x) | 1 | | |
| 5 | SPEICHERE_2(x) | 2 | | |

II.

| Systemzeit | Operation | x | akk1 | akk2 |
|---|---|---|---|---|
| 1 | LADE_2(x) | 0 | 0 | |
| 2 | LADE_1(x) | 0 | | 0 |
| 3 | ADDIERE_1(1) | 0 | 1 | |
| | ADDIERE_2(2) | 0 | | 2 |
| 4 | SPEICHERE_2(x) | 2 | | |
| 5 | SPEICHERE_1(x) | 1 | | |

Es ist klar: Je nachdem, welcher Prozeß als letzter die Gelegenheit erhält, den Inhalt seines Akkumulators abzuspeichern, ist der Inhalt von x entweder 1 oder 2, und niemals 3, wie eigentlich zu erwarten wäre. Dies ist ein in der Tat verwirrendes Ergebnis. (Die beiden ADDIERE-Operationen haben wir übrigens einem einzigen "Systemzeitpunkt" zugeordnet, da sie, obschon elementar, nicht den gemeinsamen Speicher berühren; sie können im Multiprozessor-System echt parallel ablaufen.)

Noch verwirrender wird die Geschichte, wenn wir die parallele Abarbeitung von P1 und P2 auf einem nach dem oben skizzierten Verfahren des *time sharing* funktionierenden Monoprozessor im Multiprogramm-Betrieb simulieren. Wir dürfen hier keinerlei Kenntnis darüber voraussetzen, wann ein Prozeß (P1 oder P2) von der Uhr unterbrochen wird, um dem jeweils anderen Prozeß Rechenzeit zu gewähren. Dies bedeutet, daß wir für unsere Analyse alle möglichen Verzahnungen der jeweiligen Folgen von Elementaroperationen berücksichtigen müssen. Hinzu kommt, daß es in diesem Fall auch nur einen Akkumulator gibt und die auf den Akkumulator beschränkten Operationen (ADDIERE_1 und ADDIERE_2) nicht gleichzeitig ausgeführt werden können. Wir greifen drei Möglichkeiten heraus:

I.

| Systemzeit | Operation | x | akk |
|---|---|---|---|
| 1 | LADE_1(x) | 0 | 0 |
| 2 | LADE_2(x) | 0 | 0 |

| | Systemzeit | Operation | x | akk |
|---|---|---|---|---|
| | 3 | ADDIERE_1(1) | 0 | 1 |
| | 4 | ADDIERE_2(2) | 0 | 3 |
| | 5 | SPEICHERE_1(x) | 3 | |
| | 6 | SPEICHERE_2(x) | 3 | |

II.

| Systemzeit | Operation | x | akk |
|---|---|---|---|
| 1 | LADE_1(x) | 0 | 0 |
| 2 | ADDIERE_1(1) | 0 | 1 |
| 3 | LADE_2(x) | 0 | 0 |
| 4 | SPEICHERE_1(x) | 0 | |
| 5 | ADDIERE_2(2) | 0 | 2 |
| 6 | SPEICHERE_2(x) | 2 | |

III.

| Systemzeit | Operation | x | akk |
|---|---|---|---|
| 1 | LADE_2(x) | 0 | 0 |
| 2 | ADDIERE_2(2) | 0 | 2 |
| 3 | LADE_1(x) | 0 | 0 |
| 4 | SPEICHERE_2(x) | 0 | |
| 5 | ADDIERE_1(1) | 0 | 1 |
| 6 | SPEICHERE_1(x) | 1 | |

Je nach "Laune" des Rechners produziert unser *parprogram* in der Variablen x die Inhalte 1, 2 oder 3, eine untragbare Situation. (Der mit Theorie und Praxis von Betriebssystemen vertraute Leser wird uns an dieser Stelle nachsehen, daß wir zum Zweck einer eindringlicheren Darstellung des Problems auf das in Multiprogramming-Systemen übliche sogenannte *context-switching* verzichtet haben.) Aus dem bisherigen Studium beider Szenarien ziehen wir den Schluß, daß Prozesse, die mit gemeinsamen Variablen arbeiten, sich unter keinen Umständen "ins Gehege" kommen dürfen, wenn bestimmte, diese Variablen betreffende Berechnungen laufen. Die Phasen während eines Prozesses, in denen dies der Fall ist, heißen *kritische Abschnitte*. Sie entsprechen denjenigen Passagen des einem Prozeß zugrundeliegenden Programmtextes, welche nicht-elementare Manipulationen der gemeinsamen Variablen beschreiben. In unserem Beispiel sind dies offenbar die Anweisungen "x := x + 1" und "x := x + 2".

Die Frage lautet nun: Wie kann sich ein Prozeß, der "die Absicht hat", in einen kritischen Abschnitt einzutreten, vor unerwünschten Interventionen seiner Konkurrenten schützen? Mögliche Antworten gibt das alltägliche Leben: Wollen zum Beispiel zwei Personen sich ein und derselben Sache bedienen, so müssen sie sich darüber einigen, wer wann diese Sache benutzen darf. Sie müssen ihren Wunsch anmelden und dann eventuell so lange warten, bis der jeweils andere die Sache seinerseits freigibt. Das kann etwa durch geeignete Zeichengabe geschehen, also durch bestimmte Veränderungen in der von beiden Personen wahrnehmbaren Umwelt. Man nennt solche Zeichen bekanntlich auch *Signale*,

und die mit ihrer Hilfe getroffene Festlegung einer zeitlichen Ordnung von irgendwelchen Aktionen heißt *Synchronisation.* Für Personen, die an einer gemeinsamen Aufgabe arbeiten und dabei gelegentlich die gleichen Werkzeuge benötigen, ist die Synchronisation ihrer Tätigkeiten eine notwendige Voraussetzung für den Erfolg.

Das Gleiche gilt, *mutatis mutandis*, auch für in Rechnern ablaufende Prozesse, welche auf denselben Speicher zugreifen. Die von unseren Prozessen P1 und P2 beanspruchte Sache ist die Variable x, und die von beiden wahrnehmbare Umgebung, in der Signale abgesetzt werden können, ist der restliche Speicher. Die Signalisier-Aktionen selbst müssen ununterbrechbar sein, also durch Elementaroperationen realisiert. In [ANS] wird eine von G.L. Peterson stammende Lösung des *Problems des wechselseitigen Ausschlusses* (engl.: *mutual exclusion*, vgl. auch Abschnitt 6.3.2) zweier Prozesse bei der Benutzung gemeinsamer Betriebsmittel (*Ressourcen*) zitiert; sie gilt unter den Voraussetzungen beider hier von uns betrachteter Szenarien (Multiprocessing und Multiprogramming):

```
parprogram Ordnung;
var x: integer initial (0);
    will1, will2: boolean initial (false, false);
    dran: integer initial (1) {oder 2}
  process P1;
    begin
      {vor Eintritt in den kritischen Abschnitt}
      {auszuführendes "Eintrittsprotokoll":}
      will1 := true; dran := 2;
      while will2 and dran = 2 do {warte} end;
      {kritischer Abschnitt von P1:}
      x := x + 1;
      {nach Austritt aus dem kritischen Abschnitt}
      {auszuführendes "Austrittsprotokoll":}
      will1 := false
    end {P1};
  process P2;
    begin
      {Eintrittsprotokoll:}
      will2 := true; dran := 1;
      while will1 and dran = 1 do {warte} end;
      {kritischer Abschnitt von P2:}
      x := x + 2;
      {Austrittsprotokoll:}
      will2 := false
      end {P2};
end {Ordnung}.
```

Wir behaupten: Während ein Prozeß in seinem kritischen Abschnitt ist, wird keine zum kritischen Abschnitt des anderen Prozesses gehörige Operation ausgeführt, und x hat nach Beendigung dieses parallelen Programms den Inhalt 3, unabhängig davon, welcher Prozeß zuerst seinen kritischen Abschnitt erreicht.

Um dies einzusehen, bemerken wir zunächst, daß die "Eintrittsprotokolle" mit zwei elementaren Schreiboperationen beginnen, von denen eine die gemeinsame Variable "dran" betrifft. Wieder kommt es offenbar darauf an, wie die Eintrittsprotokolle beider Prozesse miteinander verzahnt werden. Wir betrachten auch hier drei Möglichkeiten:

| I | | II | | III | |
|---|---|---|---|---|---|
| P1 | will1 := true | P1 | will1 := true | P1 | will1 := true |
| P1 | dran := 2 | P2 | will2:= true | P2 | will2:= true |
| P2 | will2 := true | P1 | dran := 2 | P2 | dran := 1 |
| P2 | dran := 1 | P2 | dran := 1 | P1 | dran := 2 |
| ... | ... | ... | ... | ... | ... |

Im weiteren Verlauf des Eintrittsprotokolls finden nur elementare Leseoperationen statt, deren Ergebnisse jeweils lokal (bezogen auf die beiden Prozesse) ausgewertet werden. Was bewirken nun diese Leseoperationen in den oben angegebenen Fällen?

Fälle I und II:

Solange P1 seine Signal-Variable "will1" nicht auf "false" setzt, ermittelt P2 als Ergebnis seiner Leseoperationen: (will1 *and* (dran=1)) = true; er darf den kritischen Abschnitt (in dem die Variable "x" manipuliert wird) also nicht betreten. Für P1 dagegen ist (will2 *and* (dran=2)) = false; er geht somit durch seinen kritischen Abschnitt und signalisiert erst danach "false" in "will1".

Fall III:

Hier ist die Situation genau umgekehrt, also zugunsten von P2. P1 muß solange warten, bis P2 im Austrittsprotokoll bestätigt, daß er sich nicht mehr in seinem kritischen Abschnitt aufhält.

Dafür, welcher Prozeß als erster "an x heran" darf, ist offenbar letztlich der Inhalt von "dran" verantwortlich. Und diesen Inhalt bestimmt der Prozeß, welcher bei der Abarbeitung des Eintrittsprotokolls als letzter die auf "dran" bezogene elementare Schreiboperation ausführt. Anschaulich gesprochen verhalten sich die beiden Prozesse außerordentlich höflich zueinander. Jeder bekundet (in "will1" und "will2") zwar seinen Willen, in den kritischen Abschnitt überzugehen, doch unmittelbar nach dieser Willensbekundung wird dem jeweils anderen Prozeß (in "dran") gewissermaßen der Vortritt gelassen. Der letzte, der seine "Verzichtserklärung" abgibt, hat dann tatsächlich das Nachsehen, vorausgesetzt, der andere hat bereits seinen Willen signalisiert. Der Leser möge sich übrigens davon überzeugen, daß es dank unserer Anfangsbelegungen der Variablen "will1",

"will2" und "dran" keineswegs ausgeschlossen ist, daß P1 und P2 in unserem Rechnersystem auch unverzahnt ablaufen können, daß also zuerst alle Operationen von P1 ausgeführt werden und dann die Operationen von P2 (oder umgekehrt).

Auf jeden Fall werden sich, nach diesen Überlegungen, die zu den kritischen Abschnitten gehörigen Operationen der Prozesse P1 und und P2 nicht überlappen. (Natürlich kann die Folge der Operationen des kritischen Abschnitts zum Beispiel von P1 durch Operationen von P2 unterbrochen werden, die nicht zum kritischen Abschnitt von P2 gehören. Doch das ist nicht von Schaden.) Und somit gilt "x = 3" nach Beendigung des parallelen Programms "Ordnung".

(Bevor wir fortfahren, möge sich der Leser bewußt machen, daß die hier vorgebrachte Argumentation im wesentlichen nichts anderes ist als eine Plausibilitätsbetrachtung, und daß es wesentlich "schwererer Geschütze" bedarf, um formale, hieb- und stichfeste Korrektheitsbeweise über parallele Programme zu führen. Diese sind jedoch außerhalb der Gegenstandsbereiche dieses Buches.)

Betriebssysteme, welche die gleichzeitige Ausführung mehrerer Prozesse (auf welcher Rechnerarchitektur auch immer) gestatten (*Multitasking*), bieten den Prozessen in der Regel die Möglichkeit, sich zeitweilig "zur Ruhe zu setzen" und etwa bis zum Vorliegen einer dem Betriebssystem signalisierten Bedingung keine Prozessorzeit in Anspruch zu nehmen. Die zitierte Lösung des Problems des gegenseitigen Ausschlusses zweier Prozesse vom Zugriff auf gemeinsame Ressourcen macht von solchen Möglichkeiten offensichtlich keinen Gebrauch. Mit ihr sind die Prozesse dem Betriebssystem gegenüber immer im "Bereit-Zustand", ständig den Prozessor beschäftigend, während sie ihre Synchronisationsvariablen inspizieren und darauf lauern, daß der andere seinen kritischen Abschnitt verläßt. Derartige Synchronisationsverfahren werden durch den Ausdruck "geschäftiges Warten" (engl.: *busy waiting*) charakterisiert. Dies mag im Rahmen weniger komfortabler Betriebssysteme gerechtfertigt sein. Doch kann man sich andere Gründe vorstellen, derentwegen eine solche Lösung eigentlich zu verwerfen ist: Nicht zuletzt ihre relative Kompliziertheit und die Tatsache, daß die konzeptuell wichtige Trennung der Ressourcenbenutzung von der Kontrolle der Ressourcenbenutzung durch keinerlei softwaretechnische Maßnahme (z.B. Abstraktion) unterstützt wird.

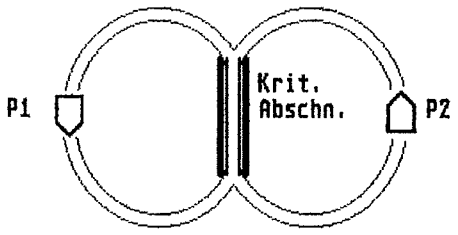
Immerhin zeigt die *busy-waiting*-Synchronisation die generelle Struktur von Lösungen des Problems des gegenseitigen Ausschlusses:

‹unkritischer Abschnitt›
‹Eintrittsprotokoll›
‹kritischer Abschnitt›
‹Austrittsprotokoll›
‹unkritischer Abschnitt›

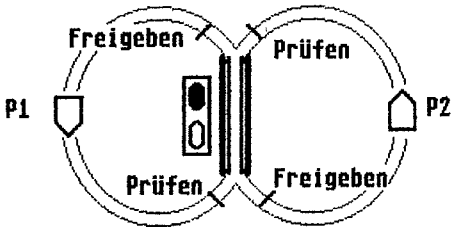
Danach sind vor Beginn des kritischen Abschnitts gewisse Operationen einzuflechten, einerseits um zu vermeiden, daß man dem konkurrierenden Prozeß in's Gehege kommt, und andererseits, um zu garantieren, daß man selbst nach Betreten des kritischen Abschnitts nicht gestört wird. Diese zu-

sätzlichen Operationen bilden das *Eintrittsprotokoll.* Das Verlassen des kritischen Abschnitts wird dann in einem *Austrittsprotokoll* bekanntgegeben. Natürlich enthält jeder Prozeß im allgemeinen auch Abschnitte, die nicht kritisch sind.

Ein zweites Beispiel für Prozesse mit kritischen Abschnitten zeigt das folgende Bild:




Hier fahren zwei Autos immer im Kreis herum. Die Crux ist nur, daß sie sich eine einspurige Brücke teilen müssen und ohne weitere Vorkehrungen Gefahr laufen, auf diesem Wegstück frontal aufeinanderzuprallen. Natürlich kennt jeder Autofahrer die Vorrichtung, die solche Gefahr bannt: die Verkehrsampel (in Spanien übrigens *semaforo* genannt). Sie muß in diesem Fall so gesteuert sein, daß sie P1 (P2) mindestens solange "rot" zeigt, als sich P2 (P1) auf der Brücke befindet. Das kann zum Beispiel dadurch geschehen, daß nach Verlassen der Brücke eine Schwelle überfahren wird, welche die Ampel umschaltet. Vor der Brücke muß jeder Autofahrer die Ampel prüfen und - wie üblich - eventuell warten, bis sie auf "grün" steht. Steht sie schon auf "grün", so muß ein Steuerungsmechanismus dafür sorgen, daß sie den Verkehr aus der jeweils anderen Richtung stoppt.




Diese einfache Ampel-Idee liegt dem von E. Dijkstra im Jahre 1968 ([DIJ]) vorgeschlagenen (abstrakten) Datentyp *Semaphor* zugrunde, der ein Attribut "s" vom Typ Integer sowie zwei Operationen umfaßt, die traditionell "P" und "V" genannt werden. Sei "sp" eine Variable vom Typ *Semaphor.* Dann sind dieses Attribut und diese Operationen durch die folgenden Eigenschaften definiert:

(i) s(sp) >= 0

(ii) Falls s(sp) > 0 vor der Ausführung von P(sp), so gilt nach der Ausführung von P(sp): s(sp) = *old* s(sp) - 1;
falls s(sp) = 0, so "blockiert" P(sp) den (die Semaphore) benutzenden Prozeß solange, bis s(sp) > 0, und nach der Ausführung von P(sp) ist s(sp) = *old* s(sp) - 1. ("Blockieren" heißt, den Prozeß - wie oben beschrieben - "zur Ruhe zu setzen"; *old* s(sp) bezeichnet hier den Wert, den s(sp) nach Beendigung der "Blockade" hat.)

(iii) Nach der Ausführung von V(sp) gilt s(sp) = *old* s(sp) + 1.

(iv) Die Operationen P und V sind ununterbrechbar.

Die meisten Multitasking-Betriebssysteme bieten diesen (oder einen ähnlichen) Datentyp an. Wir wollen sehen, wie wir uns seiner zur Lösung des Problems des wechselseitigen Ausschlusses bedienen können.

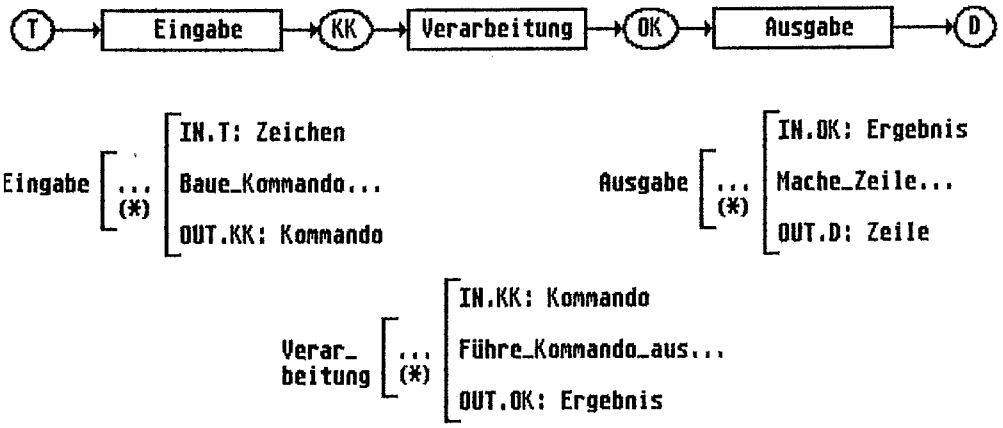
```
parprogram Ausschluss;
var mutex: Semaphor initial (1) {d.h. s(mutex) = 1};
  process P1;
  begin
    loop
      P(mutex); {Eintrittsprotokoll}
      {kritischer Abschnitt}
      V(mutex); {Austrittsprotokoll}
      {unkritischer Abschnitt}
    end
  end {P1};
  process P2;
  begin
    loop
      P(mutex); {Eintrittsprotokoll}
      {kritischer Abschnitt}
      V(mutex); {Austrittsprotokoll}
      {unkritischer Abschnitt}
    end
  end {P2};
end.
```

Dieses Programm ist eine ziemlich getreue Beschreibung des Verhaltens der beiden unermüdlichen Autofahrer. "mutex" ist die an der Brücke aufgestellte Ampel. Die Operation "P" entspricht der "Prüfung" und "V" der "Freigabe". s(mutex) hat den Wert 1, wenn die Brücke frei ist, und 0 sonst. Derjenige, der nach Beginn der Ausführung des Programms zuerst die Brücke (den kritischen Abschnitt) erreicht (sagen wir P1), findet - entsprechend der Initialisierung - s(mutex) = 1. Er fährt ohne anzuhalten weiter, vermindert aber mittels der Operation P(mutex) (z.B. beim Passieren der Schwelle) s(mutex) um 1. Kommt P2 an die Brücke, bevor P1 diese verlassen hat, sieht er s(mutex) = 0, und er wird gestoppt. Erst wenn P1 die Brücke hinter sich gelassen und s(mutex) mittels der Operation V(mutex) um 1 erhöht hat, darf P2 seine Fahrt fortsetzen.

Diese Steuerung trägt übrigens durchaus der Tatsache Rechnung, daß Prozesse (Autos) verschieden schnell voranschreiten (fahren) können: Auch wenn z.B. P2 in seinem unkritischen Abschnitt (z.B. wegen einer Reifenpanne oder weil er gerade auf einen Input von außen wartet) beliebig lange aufgehalten wird, kann P1 ohne Verzögerungen die Brücke benutzen.

Abgesehen davon, daß Semaphoren die guten Eigenschaften von Multitasking-Betriebssystemen ausnutzen, sind sie gegenüber dem Verfahren des *busy waiting* auch auf einer höheren Abstraktionsstufe: So sind sie im Programmtext (zumindest in diesem) ganz klar als Synchronisationsmechanismen zu erkennen.

Kehren wir nun zu der anfangs erinnerten Prozeßkette zurück und überlegen wir uns, wie diese im Rahmen eines Multitasking-Betriebssystems zu implementieren wäre. Wir werden drei Lösungen von steigendem Abstraktionsgrad skizzieren, mit sprachlichen Ausdrucksmitteln, die diesen Abstraktionsgraden jeweils angepaßt sind. Doch zunächst soll unsere Prozeßkette eine konkrete Aufgabe erhalten, eine Aufgabe, die aus den nachstehenden Illustrationen hinreichend klar wird:




Im Sinne der bisherigen Erörterungen sind der "Kommando-Kanal" KK und der "Output-Kanal" OK gemeinsam benutzte Ressourcen (*shared resources*). Auf KK wird von den Prozessen "Eingabe" und "Verarbeitung" zugegriffen, und auf OK von "Verarbeitung" und "Ausgabe". (Hinter T verbirgt sich die Tastatur und hinter D der Drucker. Dies sind Kanäle, auf die nur von je einem Prozeß zugegriffen wird, und die uns daher hier nicht interessieren.)

Sollen also "Eingabe", "Verarbeitung" und "Ausgabe" parallel ablaufen, so ist dafür zu sorgen, daß die Zugriffe auf diese Kanäle vom jeweiligen "Nachbarn" nicht gestört werden. Dies ist nichts anderes als die (z. B. unter Zuhilfenahme von Semaphoren) schon gelöste Aufgabe, den wechselseitigen Ausschluß von Prozessen beim Durchlaufen ihrer kritischen Abschnitte zu garantieren.

Es gibt aber noch ein Problem: Die Kapazitäten der Kanäle sind (realistischerweise) beschränkt. Wir nehmen an, daß sie als "Puffer" realisiert sind, welche nur eine bestimmte Anzahl von Plätzen (*slots*) für Objekte eines gegebenen Typs enthalten. Es hat keinen Sinn, wenn ein Prozeß etwas aus einem leeren Puffer holen möchte, und es darf auch nicht sein, daß ein Prozeß etwas in einem Puffer ablegt, in dem alle Plätze schon besetzt sind. Vielmehr muß zum Beispiel der Prozeß "Verarbeitung" unter Umständen darauf warten, daß der ihm vorgeschaltete Puffer KK ein Kommando enthält oder daß im nachgeschalteten Puffer OK ein Platz für Berechnungsergebnisse frei wird. Jede Realisierung der Operationen IN und OUT, bezogen auf die Kanäle (Puffer) KK bzw. OK, muß, in welcher Form auch immer, von diesen Bedingungen Notiz nehmen.

Das folgende *parprogram* zeigt, daß Semaphoren, deren Attribut s ja jeden positiven ganzzahligen Wert annehmen kann, außer zur Synchronisation vor kritischen Abschnitten auch dafür geeignet sind, Ereignisse zu signalisieren, die mit der Auslastung der Kapazität bestimmter Ressourcen zu tun haben.

```
parprogram EVA_0;
var   mutex_KK, mutex_OK : Semaphor initial (1,1);
      anz_KK, anz_OK: Semaphor initial (0,0);
      frei_KK, frei_OK: Semaphor initial (N,N);
      KK: PufferTyp (KommandoTyp);
      OK: PufferTyp (ErgebnisTyp);

  process Eingabe;
  var kom: KommandoTyp;
  begin
    loop
      HoleZeichen; {IN.T: Zeichen}
      BaueKommando;
      {OUT.KK: Kommando}
      P(frei_KK); {Warten auf Platz in KK}
      P(mutex_KK); {Reserviere KK für exklusiven Zugriff}
      KK.Ablegen (kom);
      V(mutex_KK); {KK freigeben}
      V(anz_KK); {Anzeige der Ablage eines Kommandos in KK}
    end;
  end {Eingabe};

  process Verarbeitung;
  var kom: KommandoTyp; erg: ErgebnisTyp;
  begin
    loop
      {IN.KK: Kommando}
      P(anz_KK); {Prüfen, ob Kommando in KK; notfalls warten}
      P(mutex_KK); {Reserviere KK für exklusiven Zugriff}
      KK.Hole(kom);
      V(mutex_KK); {KK freigeben}
      V(frei_KK); {Anzeige eines freien Platzes in KK}
      FuehreKommandoAus;
      {OUT.OK: Ergebnis}
      P(frei_OK); {Warten auf Platz in OK}
      P(mutex_OK); {Reserviere OK für exklusiven Zugriff}
      OK.Ablegen (erg);
      V(mutex_OK); {OK freigeben}
      V(anz_OK); {Anzeige der Ablage eines Kommandos in OK}
```

```
        end;
    end {Verarbeitung};
    process Ausgabe;
    var erg: ErgebnisTyp;
    begin
        loop
            {IN.OK: Ergebnis}
            P(anz_OK); {Prüfen, ob Ergebnis in OK; notfalls warten}
            P(mutex_OK); {Reserviere OK für exklusiven Zugriff}
            OK.Hole(erg);
            V(mutex_OK); {OK freigeben}
            V(frei_OK); {Anzeige eines freien Platzes in OK}
            MacheZeile;
            DruckeZeile; {OUT.D: Zeile}
        end;
    end {Ausgabe};
end {EVA_0};
```

EVA verwendet drei Paare von Semaphoren mit jeweils verschiedenen Initialisierungen des Attributs s:

- "mutex_KK" und "mutex_OK" mit anfänglich s(mutex_KK) = s(mutex_OK) = 1, zur Sicherung des wechselseitigen Ausschlusses beim Zugriff auf KK bzw. OK;
- "anz_KK" und "anz_OK" mit anfänglich s(anz_KK) = s(anz_OK) = 0, zur Kontrolle der Anzahl belegter Plätze in KK bzw. OK;
- "frei_KK" und "frei_OK" mit anfänglich s(frei_KK) = s(frei_OK) = N, zur Kontrolle der Anzahl der in KK bzw. OK freien Plätze; diese Anzahl beträgt maximal N.

Zum besseren Verständnis von EVA verfolgen wir kurz den Ablauf des Prozesses "Verarbeitung". Mit P(anz_KK) wird dieser Prozeß blockiert, falls s(anz_KK) = 0, und nur der Prozeß "Eingabe" kann ihn mit einem die OUT.KK-Operation abschließenden V(anz_KK) deblockieren. "Verarbeitung" "weiß" nun, daß in KK mindestens ein Kommando für ihn bereitliegt. Um es herauszuholen, muß er sich zunächst mit P(mutex_KK) den exklusiven Zugriff auf diese Ressource sichern, also notfalls darauf warten, daß "Eingabe" mit V(mutex_KK) den Puffer freigibt. Erst dann darf er die für die Entnahme des anliegenden Kommandos notwendigen Manipulationen der den Puffer implementierenden Variablen vornehmen. Er muß danach seinerseits KK mit V(mutex_KK) freigeben und dem (möglicherweise in P(frei_KK) schon wartenden) Prozeß "Eingabe" mit V(frei_KK) die Verfügbarkeit eines weiteren Platzes in KK anzeigen. Das Zusammenspiel von "Verarbeitung" und "Ausgabe" verläuft ganz analog. Man beachte, daß

hier - wie im *parprogram* "Ausschluß" - die Prozesse zwar auch mit unterschiedlichen Geschwindigkeiten fortschreiten können, doch nur solange der Füllstand der Puffer dies gestattet. Unter Umständen muß eben einer auf den anderen warten. Keine Kommunikation ohne Synchronisation!

Die eigentlichen Kanalzugriffe haben wir in dem obigen Programm übrigens in einem abstrakten (und parametrierten) PufferTyp verborgen. Wir hätten gewiss auch die Semaphor-Operationen dazupacken und so einen "unterhalb" der Prozeßkette liegenden Kanalmodul schaffen können. Einige unangenehme Probleme bei der Verwendung von Semaphoren als Synchronisationsmechanismen wären jedoch auch damit nicht aus der Welt geschafft. Dies sind z.B.:

- Einerseits sind Semaphoren zwar vielseitig einsetzbar, zur Bewachung kritischer Abschnitte ebenso wie zur Signalisierung von Ereignissen aller Art. Aber andererseits wird ihr jeweiliger Zweck aus der Notation selbst nicht ersichtlich. Mit Semaphor-Operationen durchsetzte Programme sind daher oft schwer verständlich.
- Die Programmierung mit Semaphoren verlangt eine strenge Disziplin. Es ist leicht, Fehler zu machen. Eine V-Operation zu vergessen oder bestimmte Semaphor-Operationen in einer anderen als der korrekten Reihenfolge anzuwenden, kann unerwünschte (und mitunter sehr unangenehme) Folgen haben.

Ein Beispiel für diese letzte Behauptung ergibt sich, wenn wir im Prozeß "Eingabe" des *parprogram* "EVA" P(mutex_KK) vor anstatt - wie es richtig ist - nach P(frei_KK) ausführen lassen. Angenommen "Eingabe" reserviert sich KK mit P(mutex_KK), wenn KK voll ist. Dann wird "Eingabe" mit P(frei_KK) solange blockiert, bis "Verarbeitung" mit V(frei_KK) einen freien Platz signalisieren kann. Falls nun "Verarbeitung" zum Zeitpunkt, da sich "Eingabe" den Puffer reserviert, gerade ein Kommando ausführt oder mit der Ausgabe eines Ergebnisses auf den Kanal OK beschäftigt ist, wird dieser Prozeß danach zwar ohne blockiert zu werden P(anz_KK) passieren, doch dann wird er mit P(mutex_KK) gestoppt, da KK ja bereits von "Eingabe" mit Beschlag belegt wurde. Das Resultat? Sehr einfach: Nichts bewegt sich mehr zwischen "Eingabe" und "Verarbeitung", und nach der eventuellen Leerung von OK durch "Ausgabe" kommt das System zum Stillstand. Man bezeichnet diese Situation als *Deadlock*, oder völlige Systemblockade.

Angesichts dieser relativen "Gefährlichkeit" des Programmierens mit Semaphoren ist es nur naheliegend, nach besseren - und das heißt hier zunächst vor allem: sichereren - Möglichkeiten der Formulierung von kritischen Abschnitten und der ihnen zugeordneten Eintrittsbedingungen zu suchen. Ein in diese Richtung zielendes und unter der Bezeichnung *bedingter kritischer Bereich* (engl.: *conditional critical region*) bekannt gewordenes Sprachkonstrukt wurde von Hoare ([HOA]) und Brinch Hansen ([BR1]) vorgeschlagen. Danach müssen Variable, auf die mehrere Prozesse Zugriff haben sollen, explizit als *Ressource*

(engl: *resource*) deklariert werden. Manipulationen dieser Variablen dürfen innerhalb des Programmtextes nur in Form spezieller, durch das Schlüsselwort *region* gekennzeichneter Anweisungen spezifiziert werden. Damit wird bestimmt, daß diese Manipulationen einen kritischen Abschnitt ausmachen, und es wird (durch die Implementierung von *region*) garantiert, daß sich zu jedem Zeitpunkt nur höchstens ein Prozeß in ihm befinden kann. In der *region*-Anweisung wird außerdem die Bedingung angegeben, unter der der Bereich überhaupt betreten werden darf. Das *parprogram* "EVA" nimmt unter Verwendung dieses Sprachkonstrukts die folgende Gestalt an:

```
parprogram EVA_1;
type  PufferTyp (T) = record
                        plaetze: array [0..N-1] of T;
                        in, out: 0 .. N-1 initial (0,0);
                        fuellstand: 0 .. N initial (0)
                      end;
var   KK: PufferTyp (KommandoTyp);
      OK: PufferTyp (ErgebnisTyp);
resource  ires: KK;
          ores: OK;

  process Eingabe;
  var kom: KommandoTyp;
  begin
    loop
      HoleZeichen; {IN.T: Zeichen}
      BaueKommando;
      {OUT.KK: Kommando}
      region ires when KK.fuellstand < N do
        KK.plaetze [KK.in] := kom;
        KK.fuellstand := KK.fuellstand + 1;
        KK.in := (KK.in + 1) mod N
      end
    end;
  end {Eingabe};

  process Verarbeitung;
  var kom: KommandoTyp; erg: ErgebnisTyp;
  begin
    loop
      {IN.KK: Kommando}
      region ires when KK.fuellstand > 0 do
        kom := KK.plaetze [KK.out];
        KK.fuellstand := KK.fuellstand - 1;
```

```
            KK.out := (KK.out + 1) mod N
          end;
          FuehreKommandoAus;
          {OUT.OK: Ergebnis}
          region ores when OK.fuellstand < N do
            OK.plaetze [OK.in] := erg;
            OK.fuellstand := OK.fuellstand + 1;
            OK.in := (OK.in + 1) mod N
          end
        end;
      end {Verarbeitung};
      process Ausgabe;
      var erg: ErgebnisTyp;
      begin
        loop
          {IN.OK: Ergebnis}
          region ores when OK.fuellstand > 0 do
            erg := OK.plaetze [OK.out];
            OK.fuellstand := OK.fuellstand - 1;
            OK.out := (OK.out + 1) mod N
          end;
          MacheZeile;
          DruckeZeile; {OUT.D: Zeile}
        end;
      end {Ausgabe};
    end {EVA_1};
```

Obwohl nun bedingte kritische Bereiche gegenüber dem fehleranfälligen Gebrauch von Semaphoren zweifellos einen bedeutenden Fortschritt darstellen, dürfen wir uns mit ihnen als inzwischen erfahrene Software-Techniker noch keineswegs zufrieden geben. Es läuft unseren Modularisierungsprinzipien eklatant zuwider, wenn wir den auf den "Ressourcen-Zugriff" bezogenen Programmtext von den Beschreibungen der übrigen Prozeßaktivitäten nicht strikt trennen können. Aus der Sicht der Prozesse besteht ja schließlich ein Interesse nur darin, die Operationen des Holens und Ablegens von Daten ausführen zu lassen. Wie und eventuell mit welchen Verzögerungen dies geschieht, muß ihnen gleichgültig sein.

Wir greifen daher auf die nach der Präsentation von Version "EVA_0" vorgebrachte und - nach allem, was wir über Modularisierung gelernt haben - sehr naheliegende Anregung zurück, alle mit den Pufferzugriffen verbundenen Operationen und Manipulationen in einem ADT-Modul zu unterzubringen. In der Definition eines solchen Moduls muß dann jedoch auch zum Ausdruck kommen,

daß sich zu jeder Zeit nur höchstens ein Prozeß einer der Modul-Operationen bedienen darf. In "EVA_1" haben wir die entsprechende Anforderung dadurch erfüllt, daß kritische Bereiche im Programmtext durch ein spezielles Schlüsselwort markiert wurden. Nun werden wir ein nämliches tun, einem ADT-Modul den "Titel" *Monitor* (d.h. "Wächter") verleihen und ihm damit bedeuten, daß er den wechselseitigen Ausschluß von Prozessen beim Gebrauch seiner Operationen gewährleisten soll. Die folgende Version des generischen PufferTyps ist damit (fast) selbsterklärend:

```
type PufferTyp(T) = monitor
  var   plaetze: array [0 .. N-1] of T;
        in, out: 0 .. N-1;
        fuellstand: 0 .. N;
        nicht_voll: condition fuellstand < N;
        nicht_leer: condition fuellstand > 0;
  procedure Ablegen (p:T);
  begin
    nicht_voll.wait;
    plaetze[in] := p;
    fuellstand := fuellstand + 1;
    in := (in + 1) mod N
  end;
  procedure Hole(var p:T);
  begin
    nicht_leer.wait
    p := plaetze[out];
    fuellstand := fuellstand - 1;
    out := (out +1) mod N
  end;
begin
  fuellstand := 0; in := 0; out := 0
end;
```

Wie gesagt: Die Deklaration des PufferTyps als *Monitor* impliziert, daß sich, wie man auch zu sagen pflegt, nicht zwei oder mehr Prozesse gleichzeitig in diesem Modul aufhalten können. Einer Erklärung bedürfen die als vom Typ *condition* deklarierten Variablen "nicht_voll" und "nicht_leer": Dieser Typ findet nur in Monitoren Verwendung; er enthält die einzige Operation *wait*. Nehmen wir an ein Prozeß ruft die Monitor-Prozedur "Ablegen" auf. In "nicht_voll.*wait* " wird die mit der Variablen "nicht_voll" assoziierte Bedingung "fuellstand < N" ausgewertet. Ergibt sich "true", so wird der Prozeß ohne Verzögerung weiterlaufen. Andernfalls gibt der Prozeß den Monitor frei und setzt sich bezüglich "nicht_voll" in Warteposition. Wann immer ein Prozeß den Monitor freigibt, findet eine

Überprüfung sämtlicher als *condition* deklarierter Variabler statt, "vor denen" andere Prozesse warten, und einem der wartenden Prozesse, deren *condition* sich als "true" herausstellt, wird der Zugriff auf den Monitor gestattet.

Mit dem als Monitor deklarierten PufferTyp nimmt "EVA" die gewünschte knappe und von anwendungsfremden Elementen freie Form an:

```
parprogram EVA_2;
var   KK: PufferTyp (KommandoTyp);
      OK: PufferTyp (ErgebnisTyp);
  process Eingabe;
  var kom: KommandoTyp;
  begin
    loop
      HoleZeichen; {IN.T: Zeichen}
      BaueKommando;
      {OUT.KK: Kommando}
      KK.Ablegen(kom)
    end;
  end {Eingabe};
  process Verarbeitung;
  var kom: KommandoTyp; erg: ErgebnisTyp;
  begin
    loop
      {IN.KK: Kommando}
      KK.Hole(kom);
      FuehreKommandoAus;
      {OUT.OK: Ergebnis}
      OK.Ablegen(erg)
    end;
  end {Verarbeitung};
  process Ausgabe;
  var erg: ErgebnisTyp;
  begin
    loop
      {IN.OK: Ergebnis}
      OK.Hole(erg);
      MacheZeile;
      DruckeZeile; {OUT.D: Zeile}
    end;
  end {Ausgabe};
end {EVA_2};
```

Die zur Darstellung der Beispiele dieses Abschnitts verwendete Notation hat - dem Leser wird es nicht entgangen sein - durchaus einiges mit real existierenden Programmiersprachen zu tun. Aber das ist auch schon alles. Ihr einziger Zweck war es nämlich, in hinreichend exakter Weise Parallelität formulierbar zu machen, und dies insbesondere über das Konstrukt *process*. Es drängt sich daher nun die Frage auf, welche Mittel in der Praxis verwendete Programmiersprachen hierfür in petto haben. Die erste Antwort ist sicherlich ernüchternd: Unter den dem Leser mit einiger Wahrscheinlichkeit bekannten Sprachen für imperative Programmierung gibt es keine, in der sich die in diesem Abschnitt behandelten Konzepte direkt umsetzen lassen. Weder "Klassiker" wie FORTRAN oder COBOL noch die "Schulsprachen" BASIC und PASCAL erlauben es, in einem geeigneten Rechnersystem parallel ablaufende Vorgänge zu beschreiben. Die zweite Antwort lautet daher: Entweder sind existierende Sprachen gezielt zu erweitern, oder es sind neue, speziell für die parallele Programmierung ausgelegte Sprachen zu erfinden. In der Tat sind beide Wege beschritten worden, und die dabei erreichten Resultate haben ganz unterschiedliche Bekanntheitsgrade und Benutzungshäufigkeiten erreicht. "Concurrent PASCAL" ([BR2]) und MODULA ([WI3]) (ein Vorläufer von MODULA-2) sind jeweils Beispiele für die eine und die andere Vorgehensweise, Beispiele, deren Wirkung freilich nie sehr weit über den akademischen Bereich hinausging. Eine Sprache, die von Anbeginn insbesondere auch für die Programmierung paralleler Prozesse entworfen wurde, ist das schon an früherer Stelle erwähnte ADA ([LED]). Dank ihrer besonderen Entstehung (im Auftrag des US-amerikanischen Verteidigungsministeriums, wie gesagt, welches damit einen eigenen Beitrag zur Überwindung seiner Softwarekrise leisten wollte) ist sie inzwischen zu relativ weiter industrieller Verbreitung gelangt.

Übrigens firmieren die genannten Sprachen auch unter der Bezeichnung *real-time* (oder *Echtzeit-*) Sprachen. Leider müssen wir es hierzu mit den gegebenen kargen Hinweisen bewenden lassen und, der Zielsetzung dieses Buches eingedenk, den interessierten Leser an dieser Stelle zum Selbststudium ermuntern. (Als weitere Quelle mag ihm dabei z.B. [BOV] dienen.)

Doch einige Bemerkungen, wenn auch nicht zu jenen Realzeit-Sprachen, sind wir noch schuldig. Denn bei der ersten Antwort oben hat vielleicht der eine oder andere Leser gestutzt und sich gefragt, warum wir denn MODULA-2, das wir schließlich als Vehikel für die Überbringung mancher Ideen zur Methodik des Programmierens eingesetzt haben, keiner Erwähnung wert halten, wenn es um die Programmierung paralleler Prozesse geht. Immerhin, so wird man sich seiner ersten Unterweisungen in dieser Sprache erinnern, gibt es in MODULA-2 doch eine NEWPROCESS genannte Prozedur, und so scheint es nur billig, daß wir zumindest andeutungsweise auf die Zusammenhänge eingehen, die zwischen der in diesem Abschnitt skizzierten Realisierung paralleler Prozesse und dem nominell verwandten Instrumentarium von MODULA-2 bestehen.

Als sogenannte "*low-level facilities*" enthält dieses Instrumentarium neben

NEWPROCESS (P: PROC; a: ADDRESS; n: INTEGER; VAR pr: ADDRESS)

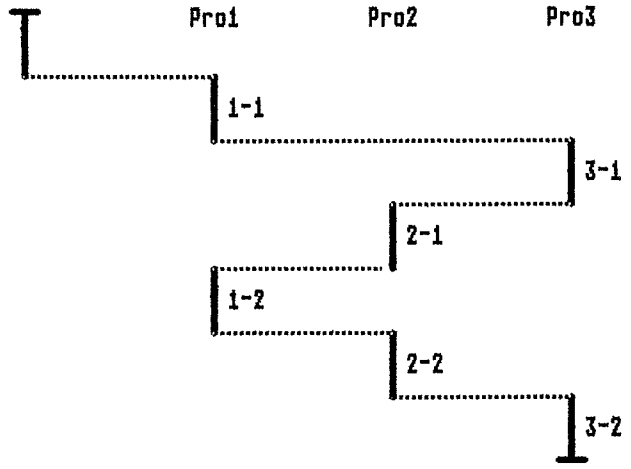
auch die Prozedur

TRANSFER (VAR pr1, pr2: ADDRESS).

Mit NEWPROCESS wird eine auf der obersten Hierarchiestufe eines Programms deklarierte parameterlose Prozedur P zur *Koroutine* umfirmiert. Ihr wird dabei ein bei a beginnender Adreßraum der Größe n zur lokalen Verfügung zugeordnet, sowie die Anfangsadresse in pr. Koroutinen sind die MODULA-2 Analoga unserer parallelen Prozesse. Sie können einander, unter Aufruf von TRANSFER, in der Kontrolle eines Prozessors beliebig abwechseln. Wir wollen uns dies an einem schematischen Beispiel etwas genauer anschauen:

```
MODULE Koroutinen;
FROM SYSTEM IMPORT ADDRESS, NEWPROCESS, TRANSFER;
FROM Storage IMPORT ALLOCATE;

VAR w1, w2, w3: ADDRESS; (*lokale Adreßräume von Pr1, Pr2, Pr3*)
    pr1, pr2, pr3: ADDRESS; (*Anfangsadressen von Pr1, Pr2, Pr3*)

PROCEDURE Pro1;
BEGIN
  (* Teilstück 1-1*); TRANSFER (pr1, pr3);
  (* Teilstück 1-2*); TRANSFER (pr1, pr2);
END Pr1;

PROCEDURE Pro2;
BEGIN
  (* Teilstück 2-1*); TRANSFER (pr2, pr1);
  (* Teilstück 2-2*); TRANSFER (pr2, pr3);
END Pr2;

PROCEDURE Pro3;
BEGIN
  (* Teilstück 3-1*); TRANSFER (pr3, pr2);
  (* Teilstück 3-2*)
END Pr3;

BEGIN (*Koroutinen*)
  (*Die - hier willkürlich gewählte - Größe der lokalen
  Adreßräume ist 500*)
  ALLOCATE (w1, 500);   NEWPROCESS (Pro1, w1, 500, pr1);
  ALLOCATE (w2, 500);   NEWPROCESS (Pro2, w2, 500, pr2);
  ALLOCATE (w3, 500);   NEWPROCESS (Pro3, w3, 500, pr3);
  TRANSFER (pr1, pr1)
END Koroutinen.
```

Die Prozedur TRANSFER(pr2,pr1) zum Beipiel läßt die Prozedur Pro1 von genau dem Punkt an weiterlaufen, an dem sie zuletzt ihrerseits einen TRANSFER-Aufruf getätigt hat. (Im Gegensatz zum klassischen Prozeduraufruf- oder *Subroutine-Call* - führt dies zu einer völlig symmetrischen Beziehung zwischen gleichberechtigten Programmteilen.) Der durch das obige Programm induzierte Kontrollfluß zwischen den Prozeduren Pro1, Pro2 und Pro3 wird durch das folgende Diagramm dargestellt:




Das ist den Verhältnissen in einem multiprogrammierten System offenbar gar nicht so unähnlich. Dort freilich geben die Prozesse nicht nur "freiwillig" den Prozessor ab, sondern sie werden dazu gelegentlich auch von der Uhr gezwungen (vgl. die Skizze auf Seite 425). Für die mit NEWPROCESS erzeugten Koroutinen sieht die Definition von MODULA-2 einen solchen Automatismus nicht vor. (Das wäre für eine Sprache mit dem Anspruch, universell einsetzbar zu sein, auch wenig sinnvoll.) Ein Programmierer, der irgendwelche Prozesse als MODULA-2 Koroutinen implementieren will, muß höchstpersönlich (durch Einschaltung entsprechender TRANSFER-Aufrufe) dafür sorgen, daß jeder Prozeß mit einem fairen Anteil an Rechenzeit bedacht wird. Durch die Tatsache jedoch, daß sie sich sozusagen selbst unterbrechen können, haben Koroutinen mit "echten" Prozessen die Fähigkeit gemein, beispielsweise auf bestimmte, durch andere Koroutinen herbeigeführte Ereignisse zu "warten". Denn das tun "echte" Prozesse in einem automatisch organisierten Multiprogrammbetrieb ja schließlich auch hin und wieder, wenn sie - wie im Beispiel des Systems "EVA" - kooperieren sollen.

Es liegt daher nahe, die TRANSFER-Operation speziell für solche Zwecke zu disziplinieren. Sie dürfte nicht direkt benutzbar sein, sondern wäre in einem Modul zu verbergen, welcher Koroutinen, die dann ganz als Prozesse im Sinne dieses Abschnitts aufzufassen wären, bestimmte Dienstleistungen anbietet. Dazu gehört etwa, einen Prozeß bezüglich des Eintreffens eines bestimmten Signals in Wartestellung zu versetzen oder ihm seinerseits die Möglichkeit zu geben, ein bestimmtes Signal abzusetzen, um damit einen anderen Prozeß aus dessen Wartestellung zu befreien. Dies sind im wesentlichen die Semaphor-Operationen P und V. In [WI1] hat Wirth selbst die Definition eines solchen Moduls angegeben, der die Illusion vermittelt (bzw. die Abstraktion!), mit Koroutinen wie mit "echten" parallelen Prozessen umzugehen.

```
DEFINITION MODULE Processes;
TYPE  Signal;
PROCEDURE StartProcess (P: PROC; n: INTEGER);
(*Startet die Ausführung der Prozedur P als parallelen Prozeß mit lokalem
Arbeitsbereich der Größe n*)
PROCEDURE SEND (VAR s: Signal);
(*Ein auf das Signal s wartender Prozeß wird wiederaufgenommen*)
PROCEDURE WAIT (VAR s: Signal);
(*Warten darauf, daß ein Prozeß das Signal s sendet*)
PROCEDURE Awaited (s: Signal): BOOLEAN;
(*Stellt fest, ob mindestens ein Prozeß auf das Signal s wartet*)
PROCEDURE Init (VAR s: Signal);
(*Initialisierung des Signals s*)
END Processes.
```

Das Bemerkenswerte an diesem Modul ist nun nicht eigentlich die Tatsache, daß seine Prozeduren - wie in [WI1] gezeigt - mit Hilfe von Koroutinen und der TRANSFER-Operation realisiert werden können, sondern vielmehr die Tatsache, daß sie auch direkt auf entsprechende Dienstleistungen eines Realzeit-Betriebssystems abbildbar sind, und in diesem Fall MODULA-2 zur "echten" Realzeit-Sprache machen.

Für eine ausführlichere Diskussion der Möglichkeiten, Abstraktionen im Zusammenhang mit der Implementierung paralleler Prozesse (wie z.B. Monitore) mit den Mitteln von MODULA-2 darzustellen, sei auf spezielle, dieser Sprache gewidmete Lehrbücher des Programmierens (z.B. [FOW]) verwiesen.

Zum Schluß dieses Abschnitts müssen wir ein anfängliches Versprechen einlösen und zumindest mit einigen wenigen Worten auf das Thema der "parallelen Prozesse in verteilten Systemen" (*distributed processing*) eingehen. In der Tat führt dieses Thema hin zu dem weiten Gebiet der *Rechnernetze* (*Computer Networks*) und seinen speziellen Problemen. Das wohl charakteristischste dieser Probleme ist durch den Begriff *Kommunikationsprotokoll* gegeben. "Kommunikation in einem System physikalisch entfernter und miteinander durch ein Leitungsnetz verbundener Prozesse" geschieht durch den Austausch von Nachrichten (*messages*) über eben jenes Netz. Wenn dieser Austausch reibungslos und störungsfrei über eine unter Umständen große Zahl von mehr oder weniger unzuverlässigen oder in ihren Kapazitäten beschränkten technischen Komponenten vonstatten gehen soll, so sind für ihn strenge und formale Regeln zu setzen.

Es sind diese Regeln, welche als *Kommunikationsprotokolle* bezeichnet werden. Ihre Definition und Implementierung sind exemplarisch für die Entwicklung und den Einsatz solider Methoden des Software-Engineering. Für den Software-

Konstrukteur stellen sie sicherlich eine der größten Herausforderungen dar an sein Qualitätsbewußtsein und an seine Fähigkeit, Komplexität überschaubar zu machen. Es ist vielleicht nicht übertrieben zu behaupten, daß dem Prinzip der modularen Schichtung nirgendwo so viel Beachtung geschenkt werden muß, wie bei der Herstellung von Software, welche Anwendungsprozesse in verteilten Systemen zu Kommunikation und Kooperation befähigt. Der Grund liegt in der Struktur der Protokolle selbst, die sich als Folge aufeinander aufbauender Ebenen präsentiert, auf deren jeder ganz präzise Verantwortlichkeiten und Funktionen zusammmengefaßt sind. Unter dem Namen "*Open Systems Interconnection Reference Model (OSI)*" ist diese Struktur sogar als internationaler Standard der ISO (International Standards Organisation) bekanntgeworden.

Der Leser bemerkt, daß auch hier eigentlich ein anderes Buch beginnen müßte. Aber auch dieses ist ein Buch, das in vielen Ausprägungen bereits existiert (z.B. [HAA], oder [SLK]).

## Literatur zu Kapitel 6


[ANS] Andrew, G. R. und F. B. Schneider: Concepts and notations for concurrent programming; ACM Computing Surveys, Vol. 15, No. 1, pp. 3-43; März 1983

[BAL] Balzert, H. (Hrsg.): CASE - Systeme und Werkzeuge; Bibliographisches Institut; Mannheim; 1990

[BAU] Baumgarten, Bernd: Petri-Netze, Grundlagen und Anwendungen; Bibliographisches Institut; Mannheim; 1990

[BEY] Benyon, D.: Information and Data Modelling; Blackwell Scientific Publications; Oxford; 1990

[BRO] Brooks, F. P.: The Mythical Man-Month; Addison-Wesley; Reading (Mass.); 1975

[BOV] Bolch, G. und M.-M. Vollath: Prozeßautomatisierung; Teubner Verlag; Stuttgart; 1991

[BR1] Brinch Hansen, P.: Structured multiprogramming; Communications of the ACM, Vol. 15, No. 7, pp. 574-578; Juli 1972

[BR2] Brinch Hansen, P.: The programming language Concurrent Pascal; IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 199-206; Juni 1975

[CAM] Cameron, J. R.: An overview of JSD; IEEE Transactions of Software Engineering, Vol. SE-12, No. 2, pp. 222-240; Feb. 1986

[DEH] Denert, E. und W. Hesse: Projektmodell und Projektbibliothek, Grundlagen zuverlässiger Software-Entwicklung und Dokumentation; Informatik Spektrum, Vol. 3, No. 4, pp. 215-228; Nov. 1980

[DGD] DUDEN, das große Wörterbuch der deutschen Sprache; Bibliographisches Institut; Mannheim; 1976

[DIJ] Dijkstra, E. W.: Cooperating sequential processes; in: F. Genuys (Ed.), Programming Languages; Academic Press; New York; 1968

[ENS] Engels, G. und W. Schäfer: Programmentwicklungsumgebungen, Konzepte und Realisierung; Teubner Verlag; Stuttgart; 1989

[FOW] Ford, G. A. und R. S. Wiener: MODULA-2, a Software Development Approach; John Wiley & Sons; New York; 1985

[GAS] Gane T. and C. Sarson: Structured System Analysis; McDonnell Douglas; 1982

[GHM] Guttag, J. V., Horowitz, E. und D. R. Musser: Abstract data types and software validation; Communications of the ACM, Vol. 21, No. 12, pp. 1048-1063; Dez. 1978

[GIL] Giloi W. K.: Rechnerarchitektur; Springer Verlag; Berlin, Heidelberg, New York; 1981

[GOR] Goldberg, A. und D. Robson: Smalltalk-80, the Language and its Implementation; Addison-Wesley; Reading (Mass.); 1983

[HAA] Halsall F.: Data Communications, Computer Networks and Open Systems; 3rd Ed.; Addison-Wesley; Wokingham (Engl.); 1992

[HAL] Hallmann, M.: Prototyping komplexer Softwaresysteme; Teubner Verlag; Stuttgart; 1990

[HAS] Halstead, M.: Elements of Software Science; Elsevier; New York; 1977

[HER] Hermes, H.: Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit - Einführung in die Theorie der Rekursiven Funktionen, 3. Aufl.; Springer Verlag; Berlin, Heidelberg, New York; 1978

[HOA] Hoare, C. A. R.: Towards a theory of parallel programming; in: C. A. R. Hoare und R. H. Perrott (Eds.), Operating Systems Techniques; Academic Press; New York; 1972

[HOS] Horowitz, E. und S. Sahni: Fundamentals of Computer Algorithms; Pitman Publishing Ltd.; London; 1979

[ILL] Igarashi, S., London, R. L. und D. C. Luckham: Automatic program verification I, a logical basis and its imlementation; Acta Informatica, Vol. 1, No. 4, pp. 145-182; 1975

[JA1] Jackson, M.: Principles of Program Design; Academic Press; London, New York; 1975

[JA2] Jackson, M.: System Development; Prentice-Hall; Englewood Cliffs; 1983

[KER] Kernighan, B.W. und D. M. Ritchie: The C Programming Language; Prentice-Hall; Englewood Cliffs; 1978

[KST] Kimm, R., Koch, W., Simonsmeier, W. und F. Tontsch: Einführung in Software Engineering; Walter de Gruyter; Berlin; 1979

[LED] Ledgard, H.: ADA, an Introduction; Springer Verlag; New York, Heidelberg, Berlin; 1980

[MEY] Meyer, B.: Object Oriented Software Construction; Prentice Hall; New York; 1988

[MID] MID GmbH, Nürnberg: INNOVATOR Produktdokumentation; 1989

[NIV] Nievergelt J. und A. Ventura: Die Gestaltung interaktiver Programme; Teubner Verlag; Stuttgart; 1983

[OTW] Ottmann, T. und P. Widmayer: Algorithmen und Datenstrukturen; Bibliographisches Institut; Mannheim; 1990

[PA1] Parnas D. L.: A technique for software module specification with examples; Communications of the ACM, Vol. 15, No. 5, pp. 330-336; Mai 1972

[PA2] Parnas D. L.: On the criteria to be used in decomposing systems into modules; Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058; Dez. 1972

[PET] Petri, Carl Adam: Kommunikation mit Automaten; Schriften des Instituts für instrumentelle Mathematik; Bonn; 1962

[ROS] Ross, D. T. und K. E. Schoman Jr.: Structured analysis for requirements definition; IEEE Transactions on Software Engineering, Vol. 3, No. 1, pp. 6-15; Jan. 1977

[SCH] Schmitt, A. A.: Dialogsysteme - kommunikative Schnittstellen, Software-Ergonomie und Systemgestaltung; Bibliographisches Institut; Mannheim; 1983

[SCS] Schlageter, G. und W. Stucky: Datenbanksysteme, Konzepte und Modelle; Teubner Verlag; Stuttgart; 1983

[SLK] Sloman M. und J. Kramer: Verteilte Systeme und Rechnernetze; Hanser Verlag; München; 1989

[SMU] Smullyan, R. M.: Forever Undecided - a Puzzle Guide to Goedel; Oxford University Press; Oxford; 1988

[SOM] Sommerville, J.: Software Engineering, 4th Ed.; Addison-Wesley; Wokingham (Engl.); 1992

[STR] Stroustrup, B.: Die C++ Programmiersprache; Addison-Wesley (Deutschland); Bonn, 1987

[TAN] Tanenbaum A. S.: Operating Systems - Design and Implementation; Prentice-Hall; Englewood Cliffs; 1987

[WI1] Wirth, N.: Programming in Modula-2; Springer Verlag; Berlin, Heidelberg, New York; 1988

[WI3] Wirth, N.: Modula, a language for modular multiprogramming; Software Practice and Experience, Vol. 7, pp. 3-35; 1977

[YOC] Yourdon E. and L. L. Constantine: Structured Design; Prentice-Hall; Englewood Cliffs; 1979

# Stichwortverzeichnis


Abbildung, 113
Abbruch,
  ...bedingung, 140, 150, 237
    -> Schleifenbedingung
  eines Programmlaufs, 128, 392
Abgeschirmtheit, 322f, 334, 337
Ablaufstruktur, 284
abortion,
  -> Abbruch eines Programmlaufs
Abstrakter Datentyp, 15, 321, 355, 357ff, 379, 381, 384
  Ausprägung von ...en, 381
  instance von ...en, 381
  Konsistenz der Definition von ...en, 379
  Standard-Implementierung, 370
  Standard-Repräsentation, 368ff
Abstrakte Funktion, 64
Abstraktion, 102, 432
  Daten-, 385
  funktionale, 385
  ...s-Mechanismen, 263
  Prinzip der, 66
Acht-Damen-Problem, 80, 378
  rekursive Lösung, 92
ADA, 8ff, 16, 324, 327, 384, 388, 400, 408, 423, 443
Adaptierbarkeit, 246, 253, 323, 380, 383f, 417
ADT,
  -> Abstrakter Datentyp
Agent, 287
Aktion, 287
Aktionsdiagramm, 98
ALGOL, 384
Algorithmus, 28, 50, 253, 334
All-Quantor, 117ff
Analyse, 260
  lexikalische, 328
  objektorientierte, 379
  -> System-Analyse
ancestor, 401
Anfangsmarkierung, 282
Anforderung, 322
  ...s-Definition, 24
  ...s-Erfüllung, 234
  ...s-Spezifikation, 67, 303, 329
  -> Spezifikation
Anwendung,
  ...en, kommerzielle, 231, 308
  ...s-Programm, 51, 243
  ...s-Software, 242ff
Anzeigetafel, 291
Array-ADT, 362ff
Arztpraxis, 268
Assembler, 7, 231
Assembler-Sprache, 7
Assemblierer, 7
Assertion, 120, 124, 392
Ästhetik, 34
Attribut, 265, 386
  -Speicher, 391
aufwärts-kompatibel, 245
Ausnahmesituation, 305, 335
Außenwelt, 270
Ausschluß,
  Problem des wechselseitigen ...es, 281f, 430, 433, 441
Austrittsprotokoll, 430, 432
Automation,
  teilweise, 271
  vollständige, 271
Axiom, 361, 365, 368, 393
  Erfülltheit von ...en, 372
backtracking, 95, 195, 378
Bank, 325f
Bankschalter, 271
BASIC, 9ff, 16, 443
Basismaschine, 12, 231, 233, 261, 318f, 424
Basis-Hardware, 231
Basis-Software, 51, 231, 261, 330
Basistypen, 378
Baustein, 322, 329f, 356, 385
  -> Software
B-Baum, 219

Benutzung, 325
...sfreundlichkeit, 18, 253ff
...soberfläche, 253f, 318, 328f
graphische, 258
...sschnittstelle, 244, 329, 331, 342
von Modulen, 325
Berechenbarkeitstheorie, 46, 359
Berechnungsobjekt, 204
Berechnungsroutine, 386
funktionale, 386
prozedurale, 386
Beschreibbarkeit, 322f, 334
Betriebsart, 425
Betriebsmittel, 281
Betriebssoftware, 243, 318
Betriebssystem, 51, 228, 230, 242ff, 259, 268, 318, 330, 356, 424, 432
Realzeit-, 446
Beweisen,
automatisches, 379
-> Programmieren
Bezeichner,
-> Variablenbezeichner
-> Wertbezeichner
Beziehung, 265ff
Kommunikations-, 268
zwischen Objekten, 270
sequentielle ...en, 285
Bibliothek,
-> Programm
Bibliothekssystem, 286ff
Bibliotheksverwaltung, 309
Black-Box-Modell, 64, 68, 70
Boolean-ADT, 373
Boole'sche Algebra, 373
border-clash, 227
bottom-up, 327, 329f, 357
Brauchbarkeit, 18, 234, 241f, 253, 258
bug, 336
busy waiting, 432, 434
C, 8ff, 16, 327
C++, 327
CASE (Computer Assisted Software Engineering), 312
CASE-Anweisung, 131, 146
class, 385ff
-interface, 390
deferred, 418ff
COBOL, 8ff, 16, 384, 443
Codegenerierung, 328
collating, 209
Compiler, 8, 51, 53, 59, 121, 123, 126, 161, 185, 200, 242f, 324, 328, 365ff, 390, 395, 409
für EIFFEL, 397
für MODULA-2, 50, 72, 75, 129
Compilierungseinheit, 30
Computer Networks, 446
Concurrent PASCAL, 443
condition, 441
conditional critical region, 438
context-switching, 429
Create-feature, 388
Datei, 350
Mischen von ...en, 209
sequentielle, 168, 216
-Manager, 350f
-Verwaltungssystem, 243f, 318, 330
Daten,
Organisation von, 317
Repräsentation von, 334
Verwaltung von, 317
Zugriff auf, 317
-fluß, 70, 285, 310, 319, 331f
-Diagramm, 268f
-format, 242, 351
-kommunikationssystem, 229
-komplex, 177, 179, 216
-modellierung, 308
-objekt, 168f, 185, 199, 251, 332
Analyse von ...en, 185
Produktion von ...en, 179ff
Verknüpfung von ...en, 169
-reihe, 173
-speicher,
gemeinsam benutzter, 324f
privater, 325
-strom, 168f, 172, 208, 216, 306
Input-, 199f, 211, 219
Output-, 199f, 217, 219
parallele ...e, 179

Daten,
-strom,
Struktur eines ...s, 312
-struktur, 50, 185, 253, 284, 333, 386, 396
effiziente, 28
-träger, externer, 253
-typ, 289, 333
-> Abstrakter Datentyp
-verwaltung, 254
Datenbank
-Management-System, 51, 228, 231, 245, 318, 356
-System, 242, 246, 253
Deadlock, 438
Deduktionssysteme, 12
Definition,
-> Spezifikation
depth-first, 95, 307
Detail-Entwurf, 102, 109
descendant, 401
desktop, 257
Desktop Publishing, 254
Dienst, 325f
Dienstleistung, 340ff, 357, 367, 384f, 392, 445
Dienstleistungsunternehmen, 268, 271
Differentialgleichung, 266
Direktzugriffs-Medium, 216
distributed processing, 426, 446
Dokumentation, 24, 30ff, 230, 241, 258, 392
Benutzungs-, 238, 392, 396
Entwurfs-, 96, 179, 395
externe, 25
graphische, 306
interne, 25, 245f
-> Selbstdokumentation
dynamic binding, 410
dynamische Bindung, 410
Echtzeit,
-System, 229
-> Programmiersprachen
-> Programmierung
Editor,
syntaxgesteuerter, 328
Effizienz, 18, 27ff, 50ff, 234, 253, 293, 370
Verbesserungsregeln, 52ff
arithmetische Ausdrücke, 52, 53ff
boolesche Ausdrücke, 52, 59
Indizierung, 52, 60
Iterations-Anweisungen, 52, 55
Prozeduraufrufe, 52, 61
Selektions-Anweisungen, 52, 58
EIFFEL, 327, 385ff
-Compiler, 397
-Laufzeitumgebung, 389
Einkapselung, 340, 350
EIN_SPIEL-class, 406
Eintrittsprotokoll, 430, 432
Elter, 401
Entität, 265f
Entscheidung
Repräsentations-, 65f, 69, 71, 83, 332, 340, 352
Zerlegungs-, 69, 332
Entwurf, 179, 261, 305
Daten-, 321
der Architektur, 321, 327, 352
eines Hauses, 317
objektorientierter, 379, 381
prozeduraler, 321, 327, 352
zu einem Roman, 317
...s-Entscheidung, 332, 351f
...s-Fehler, 236
...s-Notation, nach Yourdon und Constantine, 320
...s-Sprache, 323, 327, 340
-> Programm-Entwurf
-> Software-Entwurf
-> System-Entwurf
Erbe, 401
Ereignis, 265f, 272f, 276, 285f
Aufeinanderfolge von ...sen, 276
Erreichbarkeitsbaum, 283
Erzeuger-Verbraucher-Problem, 279ff
EVA-parprogram, 436
Existenz-Quantor, 116ff
Expertensystem, 81, 95
Extension,
-> Prädikate

Extraktion, 347f, 362
-> Operationen
Fahrstuhl, 296, 305
-System, 312
Fehler,
-Code, 238
interner, 238
-korrektur, 235
-meldung, 335, 392
-Toleranz, 230, 240, 323
Fertigung,
...sbetrieb, 268
...splanung, 167
Filter, 262, 270f
Fingerspitzengefühl, 144, 306, 348
First-In-First-Out (FIFO), 291
Fließband, 271
flow-chart, 96
Fluß-Diagramm, 96, 246
Folgedatum, 67ff
FOR-Anweisung, 157
Formatierung, 31
FORTRAN, 8ff, 16, 35, 46, 384, 443
fünf Gebote, 333, 339
Gane-Sarson-Diagramm, 268, 312
garbage collection, 389
Geheimhaltung, 325ff, 333
Geheimnisprinzip, 340, 390
Geldausgabeautomat, 271
Generizität, 423
Gerätetreiber, 351
Gleichheitsrelation, 359
Gleichungssystem,
lineares, 98, 101, 107
Gleichzeitigkeit, 277, 427
Glückseligkeit,
objektorientierte, 423
GOTO, 46
Grenz-Konflikt, 227
Großprojekt, technisches, 259
Gültigkeitsbereich,
-> Scope
Hash
-Tabelle, 378
-Verfahren, 219
Heapsort, 29
Icon, 258
IF-, IF-THEN-ELSE-Anweisung, 132, 146, 156
Implementierung, 179, 230f, 233, 235, 260, 264, 305, 317, 334, 348, 359, 363ff, 390
...s-Fehler, 236
...s-Sprache, 323, 325
abstrakte, 321, 364, 371, 396, 398
eines Abstrakten Datentyps, 363
generische, 398
paralleler Prozesse, 321
Referenz-, 370
-> Abstrakter Datentyp
-> Modul
-> System
-> Vererbung
IN:, 187, 202, 206, 435
qualifiziert, 190, 210, 217, 298
Indentierung, 32
Regeln, 32ff
Induktion, 377
Information, 347f, 362
-> Operationen
information hiding, 325ff
Informationsspeicher, 271
Informationssystem, 228, 268, 328
betriebliches, 266
Datenbank-basiertes, 267
inheritance, 384, 400ff
multiple, 417ff
repeated, 422
Initialisierung, 347, 362
-> Operationen
INNOVATOR-RAD, 313
Instanz, 313
Kommunikation zwischen ...en, 313
Integration, 261
Integrierbarkeit, 18, 24f, 234, 241ff
Integrität, 235ff, 252, 323
Interface, 270
-Abstraktion, 395, 418
-> Schnittstelle
International Standards Organisation, 447
Interpreter, 4

Invariante, 122, 402
-> Schleifen-Invariante
Invertierung,
eines Prozesses, 320
Iteration, 46, 63, 96f, 100, 105f, 107, 126, 135ff, 171
(*)-..., 172
(#)-..., 172f, 182
(&)-..., 178, 182
...s-Anweisung, 145, 168
Konstruktion, 148ff
...s-Objekt, 169, 172f, 178, 192
Jackson System Development, 15, 263, 284ff
JSD, 284ff, 306
Kalküler, formaler, 379
Kanal, 112, 210, 213, 215ff, 224f, 241, 271, 280, 290, 292, 313ff, 318, 320, 357, 425, 435
interner, 225
lokaler, 315
Mischen von ...en, 291f
owner eines ...s, 313
Umgebungs-, 315f
user eines ...s, 313
verfeinerter, 315
-modul, 438
-Typ, 290f
-Verbindung, 294, 297
-zugriff, 298, 435
Klammerdiagramm, 98, 104ff, 168, 186, 284, 288, 312
Klasse, 15, 379ff
...nparameter, 396
...ntyp, 388
aufgeschobene, 419
generische, 397f
parametrisierte, 396
Definition von ...n, 400
Kombinierbarkeit, 322f, 334, 337
Kommentar, 7, 11, 25, 30, 41, 70, 106, 120ff, 157, 188, 208, 358, 385
Kommunikation,
über Kanäle, 279, 312
über Speicher, 279, 281, 312
zwischen Instanzen, 313
Kommunikation,
...sprotokoll, 280, 282, 446
-> Prozeß
Kompatibilität, 242, 244f
...sregeln, 408
Komplexität,
von Programmen, strukturelle, 30, 42ff
...stheorie, 29
Komponente,
von Datenobjekten, 169
optionale, 176f, 192
Top-, 176
-> Objekt
-> System
Konflikt, 275, 281
Konkatenation, 130
Kontenbewegungsbericht, 170ff, 175, 200ff
Konto-Informations und -Transaktions-System, 271
Kontrollfluß, 46, 319
Kontrollgröße, 310f
Kontrollübergabe, 320
Konversion, 351
konvexe Hülle, 330ff, 351f
Kopplung, starke, 324f
Koroutine, 444ff
Korrektheit, 13, 18, 19ff, 111, 144, 148, 185, 204, 234ff, 321, 323, 359, 378
einer Implementierung, 359, 365, 370
paralleler Programme, 432
Korrespondenz,
-Analyse, 219
zwischen Input und Output, 200, 208, 211, 219, 225
Kreuzworträtsel, 342ff
kritischer Abschnitt, 429
kritischer Bereich,
bedingter, 438
Kunde, 341f
K.U.S.S., 342
Labyrinth,
Suche im, 88, 378
rekursive Lösung, 91

Layout,
  eines Programmtextes, 32ff
Lebensgeschichte, 288
Leihbücherei, 268ff
lineare Suche, 160, 163
LISP, 12, 16
Liste,
  ...nstrukturen, 396
  lineare, 335
Literal, 37
Logik, 359
  mathematische, 264
Lokalitätsprinzip, 66, 250
low-level facilities, 444
Machbarkeit, 258f
Makro, 231
Mapping-ADT, 371
Marke (im Petri-Netz), 273
Markierung, 283
  erreichbare, 283
Maschinensprache, -code, 4, 50, 59, 126, 242, 328
Maus, 258, 342
Medium, 271
  flüchtiges, 271
  nicht-flüchtiges, 271
Menge, 113
Mengen-Operator, 115
Mensch-Maschine
  -Dialog, 18
  -Kommunikation, 253
  -Schnittstelle, 347
  -> Benutzungsschnittstelle
Menü, 256
Mergesort, 29, 156f
merging, 209
Metainformation, 355
Modell, 231
  abstraktes, 263
  deterministisches, 266
  diskretes, 266
  dynamisches, 265
  für Teile der realen Welt, 261
  Klassifikation von ...en, 265
  kontinuierliches, 266
  netz-orientiertes, 263f
Modell,
  nicht-deterministisches, 266
  statisches, 321
  statistisches, 265
  stochastisches, 267
  -baukasten, 316
  -dokumentation, 284
  -Eisenbahn, 264
  -kern, 270f
  -Klassen, 267
  -Prozeß, 285f, 293
  -rand, 270, 306
  -Struktur, 284
Modellierung,
  hierarchische, 328
Modellierungsinteresse, 278, 314
Modul, 14, 231ff, 235f, 241, 260, 305, 321f, 324, 329f, 330ff, 357, 384, 445
  Abstrakter-Datentyp-, 355ff
  ADT-, 358, 382f, 440f
    Implementierung eines ...s, 358
    Semantik eines ...s, 359
    Spezifikation eines ...s, 359f
  Daten-, 340f, 347
  Definition eines ...s, 352, 400
  Definitions-, 334ff, 357
  Definitionsteil eines ...s, 352f
  Entwurf und Implementierung, 326
  Implementierung eines ...s, 352, 390, 400
  Implementierungs-, 335, 349
  Implementierungsteil eines ...s, 352f
  Prozeß-, 340f, 357
  Ressourcen-, 356f
  Schnittstellen von ...en, 327, 332f, 391
  Steuer-, 332, 356f
  -Arten, 341
  -beschreibungssprache, 323ff
  -Spezifikation, 357
  -Semantik, 321, 358f
  -Spezifikation, 358
  -Syntax, 357, 359
MODULA, 443
MODULA-2, 8ff, 16f, 24ff, 67f, 97, 106, 111f, 114, 123f, 126, 128f, 161,

MODULA-2, 166, 169, 181f, 184f, 188f, 194, 196, 206f, 214, 222, 242, 247, 249, 251, 324, 327, 334ff, 358, 362, 366, 383f, 386, 390, 392, 400, 405, 408ff, 423, 443ff
Modularisieren, 321
Modularisierung, 327, 330ff, 378f, 440
Modularität, 259
Monitor, 441, 446
Monoprozessor, 318, 424f
Müll-Sammlung, 389
Multiplikationstabelle, 165, 179ff, 246ff
Multi-Processing, 318f, 332, 350, 425, 430
Multi-Programming, 425, 428, 430, 445
Multi-Tasking, 318, 424, 432, 434f
multi-threading-clash, 227
mutual exclusion, 430
Nachbedingung, 121, 124, 148, 158, 236f, 359, 392f
Nachfolgestelle, 273, 283
Nachkomme, 401
Namensgebung, 34ff
Namenskonflikt, 422
Nassi-Shneiderman-Diagramm, 98, 99ff, 104, 109
Netzwerk,
  kommunizierender Prozesse, 286, 294, 424
  von Prozessen und Datenspeichern, 317
Nullobjekt-Typ, 176, 211f, 223
Objekt, 15, 169, 258, 264f, 268, 270, 321, 338, 340, 343, 355, 362, 379ff, 424, 426
  aktives, 267
  analysieren, 165
  Botschaften an ...e, 388
  Eigenschaften eines ...s, 380f, 385
  Erzeugung von ...en, 286
  features eines ...s, 381, 385, 401
  Interaktion von ...en, 380
  Konstruktion von ...en, 363
  Menge von ...en, 361, 384
  messages an ...e, 388
Objekt,
  passives, 267
  produzieren, 165
  Semantik von ...en, 395
  -fluß, 268
  -Fluß-Diagramm, 226, 268
  -Klasse, 265, 288
  -Komponente, 176
  -orientierung, 321, 379ff
    -> Analyse
    -> Entwurf
    -> Programmiersprache
    -> Programmierung
  -speicher, 268
  -struktur, 166, 168, 169ff
  -typ, 172, 177, 297
    parametrierter, 183
    rekursiver, 178
Open Systems Interconnection, 447
Operationen, 337ff, 343, 356, 358, 361, 363, 390
  atomare, 425
  Kategorien von, 347
  Klassifizierung von, 380
  Konstruktions-, 363
  -> Extraktion
  -> Information
  -> Initialisierung
  ->Transformation
Optimierung, 51
ordering-clash, 227
Organisationseinheit, 252, 258
OUT:, 181, 202, 205f, 435
  qualifiziert, 190, 217
overloading, 423
Paradigma, 12, 81, 95, 281, 321, 348
Parallelität, 312
  räumliche, 169
  -> Programmierung
  -> Prozeß
Parsing, 328
PASCAL, 8ff, 16, 40, 44, 46, 169, 182, 196, 384, 386, 408, 443
Periodizitäts-Anzeige, 320
Personenjahr, 229, 258
Petri-Netz, 263, 272ff, 283, 286, 312

Phase, 261
Phasenmodell, 261
Plateau-Problem, 161
Polymorphie, 409ff
Portabilität, 102, 242f, 245
POSIT-ADMIT-Konstruktion, 191, 194, 196, 199
Postulat, 361, 365
Prädikat, 112ff, 124, 392, 402
  als Kommentar, 120ff, 236f
  Extension eines ...s, 114, 117, 158
  Operationen mit ...en, 114
  ...en-Transformierer, 125ff
Pretty-Printer, 34
Produktionsbetrieb, 249, 251, 258, 330
Produktionsprozeß, 166f
Produkt-Modell, 260
Programm,
  abstraktes, 364, 369
  Haupt-, 332
  leeres, 127
  -Beschreibung, 181, 191, 209, 213
  -Bibliothek, 25, 35, 330
  -Entwurf, 186, 246, 263
    datenstrukturierter, 15, 165ff, 229, 285
      Schritte des ..., 208, 210, 217, 222
  -qualität, 13, 14, 18ff
  -struktur, 169ff
  -system, 14, 15, 238, 392
  -verifikation, 23, 111, 359
  -> Komplexität
  -> System
Programmieren,
  durch Beweisen, 111ff, 359
  Logik des ...s, 111
  Psychologie des ...s, 12, 63
  zielorientiertes, 111, 121, 144
Programmierstil, 17, 30ff, 248, 360
Programmiersprache,
  blockstrukturierte, 365ff
  Erweiterbarkeit einer, 356
  deskriptive, 5
  Echtzeit-, 443
  höhere, 5, 8ff, 51, 112, 243, 328,
Programmiersprache,
  höhere, 362
  imperative, 5, 8, 12, 112, 384, 443
  objektorientierte, 379, 385
  real-time-, 443
  Realzeit-, 446
  Semantik von, 243
  Syntax von, 243
Programmier-Tricks, 28, 45, 46, 51, 409
Programmierumgebung, 419
Programmierung,
  Echtzeit-, 168
  im Großen, 14, 15, 228ff, 260
  im Kleinen, 14, 15, 17, 66, 157, 228, 232, 234, 241, 253, 327, 332, 352
  imperative, 1ff
  objektorientierte, 15, 379ff, 423
  parallele, 443
  Realzeit-, 168
  strukturierte, 99, 246
  -> Programmieren
Projekt-Modell, 260f
Projekt-Organisation, 260
PROLOG, 12, 16, 95
Prototyp, 329
Prototyping, rapid, 329
Prozeß, 15, 167, 265, 267, 270, 307, 381
  Ausführung eines ...es, 317
  der Software-Entwicklung, 284
  funktionaler, 285f, 293, 299f
  Invertierung eines ...es, 320
  Kommunikation von ...en, 272, 290, 317
  kommunizierende ...e, 226f, 268, 320, 381
  Kooperation von ...en, 272
  kooperierende ...e, 226, 228, 268
  linear ablaufender, 275
  nebenläufige ...e, 185, 276
  parallele ...e, 185, 241, 275, 424ff
  suspendierter, 289
  technischer, 168
    Steuern, Simulieren, 266
  Variablen eines ...es, 289
  zyklischer, 275

Prozeß,
-Aktionen, 289
-Beschreibung, 183, 187, 191, 202, 204, 209, 219
-Daten-Dualität, 308
-deklaration, 426
-kette, 318
-Kommunikation, 270, 279, 319
...s-Diagramm, 293
-leitsystem, 228, 230
-Modell, 275
-Steuerung, 317, 319
-steuerungssystem, 328
-Synchronisation, 241, 276
-Typ, 184, 289ff
-und Prozessor, 426
-zustand, 289
Prozessor,
Gleitkomma-, 53
virtueller, 319
-> Monoprozessor
Prüfbarkeit, 236, 238, 245, 323
Pseudo-Code, 96, 284
psychologische Distanz,
zwischen Namen, 37
Puffer, 435
Punktmenge, 330ff
Puppenstube, 264
Qualität,
im Großen, 234ff
im Kleinen, 17ff
...sanforderung, 230
...sbewußtsein, 447
...skriterium, 322
...smerkmal, 233f, 235, 245, 252, 253, 258
Quadratwurzel, 159
Quicksort, 29
QUIT, 191, 192
RAD-Diagramm, 313
Rapid Prototyping, 329
reale Welt, 15
Modelle für Teile der, 261
Objekte, Beziehungen, Vorgänge, 264
Realwelt, 239, 263, 272, 306
-Ausschnitt, 239, 262, 286f, 306
Realwelt,
-Modell, 263, 264ff
-Objekt, 285, 296
-Prozeß, 285f, 293
Simulation von ...en, 297
-System, 306
Modellierung, 316
Realzeit,
-System, 228
-> Programmiersprache
-> Programmierung
Rechnerarchitektur,
Multiprozessor, 318, 424
verteilte, 426
Rechnernetz, 446
Rechnungs-Ausschrieb, 98, 103, 108, 167, 215
Redefinition, 389
Referenz,
leere, 388
regulärer Ausdruck, 225
Reihenfolge-Konflikt, 227
rekursive Funktionen, 12
rekursive Prozedur, 93
Repräsentationsunabhängigkeit, 338
Requirements Engineering, 24
-> Anforderung
Requirement Specification, 303
-> Spezifikation
Ressource, 238, 251, 281, 323, 350, 438
gemeinsam benutzte, 435
...n-Manager, 350
Robustheit, 239f, 254, 257
Rohrpost, 280, 291
SADT, 263, 308ff, 313
-Datenmodell, 308
-Diagramm, 311
-Faustregel, 309
-Prozeßmodell, 308
Schalt-Regel (im Petri-Netz), 283
Schaltjahr-Regel, 75
Scheduler, 319f
Schicht, 329f
-> System
Schichtung,
-> System

Schleifenbedingung, 148, 150f, 154, 158
Schleifen-Invariante, 139ff, 145, 148, 151, 154, 157f, 168, 236
  Auffindung von ...n, 157ff
    durch Verallgemeinerung, 158
Schnittstelle, 18, 67, 244, 261, 306, 321, 325f
  ...nbeschreibung, 390f, 395
  ...ndefinition, 390
  dicke ...n, 340
  explizite ...n, 324, 333, 340
  implizite ...n, 340
  schmale ...n, 324, 333, 340
  wenige ...n, 324, 333
  -> Benutzungsschnittstelle
  -> Modul
Schranken-Funktion, 139ff, 145, 148, 151, 154, 359
schrittweise Verfeinerung, 15, 63ff, 111f, 157, 168, 228, 232f, 263, 305, 327f, 332
  Diagramm-Techniken für, 96ff
Scope, 119, 251, 366f
Seiteneffekt, 31, 47ff, 129f, 195, 199, 325, 387
Selbstdokumentation, 27, 30ff, 348
Selektion, 46, 63, 96f, 99f, 105, 107, 126, 131ff
  ...s-Anweisung, 124, 145ff
  ...s-Objekt, 169, 171, 173f, 211f, 222
Semantik, 111, 124ff, 223, 308, 313, 320, 340
  der CASE-Anweisung, 132
  der IF-Anweisung, 133
  der Sequenzbildung, 131
  der WHILE-Anweisung, 136, 138
  der Zuweisung, 129
  eines ADTs, 392
  einer Berechnungsroutine, 392
  Formalisierung der, 359
  -> Modul
  -> Objekt
Semaphor, 433ff, 445
Sequentialität,
  räumliche, 169
  zeitliche, 169
Sequenz, 47, 96f, 99, 105f, 126, 129ff
shared memory, 425
shared resources, 435
Signal, 429
Simulation, 22,
  -> Realwelt
Smalltalk, 327, 387
Software
  -Architektur, 318
  -Baustein, 231, 324
  -Engineering, 24, 234, 258ff, 322, 380, 417, 446
    -Environment, 312
  -Entwicklungssystem, 229
  -Entwurf, 66, 169, 230f, 233, 252, 258, 384
  -Ergonomie, 254
  -Krise, 260, 443
  -Metrik, 42, 246
  -Produkt, 234, 258
  -Produktion, 322
  -Qualität, 17
  -System, 17, 169, 228, 229ff, 317f, 365, 381
    interaktives, 257
    Planung und Realisierung, 259f
    modularer Aufbau eines ...s, 321
    Struktur eines ...s, 231, 305
    Strukturierung von ...en, 258, 260, 380
  -Werkzeuge, 52, 97f, 179, 229, 234, 260, 312, 378
  -> Programmierung
  -> System
Sortieren, 28, 123, 151
  durch Mischen, 156, 163
Spaghetti-Programm, 46
Speicher, 112ff
  -Effizienz, 19, 27
  -operation, elementare, 425
  -zustand, 113, 120, 123f, 127, 129, 131f, 135
Spezifikation, 13, 21, 24, 185, 205, 230, 233, 246, 260
  durch Prädikate, 121
  eines ADT, 396

Spezifikation,
formale, 23, 234, 394
sequentieller Abläufe, 301
von Systemteilen, 260f
von Verhalten, 302
...ssprache, 359f, 364, 373
-> Modul
-> System
spreadsheet, 254
Stack
-ADT, 360ff
und Symboltabelle, 370
Standards, 242, 447
Stelle (im Petri-Netz), 273, 277, 283
Stetigkeit, 322f, 334, 337
Stetigkeitseigenschaft, 246
Steuerungsprozeß, 319
structure clash, 226
Structured Analysis and Design Technique, 264
-> SADT
Struktogramm, 99
Strukturbruch, 226
Strukturkonflikt, 222ff, 228
Symboltabelle, 366ff
Synchronisation, 430
...smechanismen, 434
-> Prozeß
-> Variable
Syntaxanalyse, 186
System, 263, 268, 317
Anwendersicht eines ...s, 328
Dialog-, 254, 257
diskretes, dynamisches, 272
Funktionalität eines ...s, 285
künstliches, 271, 282
Leistung eines ...s, 328
natürliches, 282
nebenläufiger Prozesse, 277, 279
rechnergestütztes, 284
soziotechnisches, 272
verteiltes, 446f
-Analyse, 231f, 261, 262ff, 317, 328
-architektur, 321, 323, 383
-Beschreibung, 272
hierarchische, 277
System,
-Darstellung, 263
-Entwurf, 15, 252, 260f, 285, 317ff, 381
-funktion, elementare, 307
-Implementierung, 15, 260f, 285, 317ff, 381, 384
-Komponente, 231, 233, 235, 241, 260, 306
-Modell, 356
hierarchisches, 263, 305ff, 327
Dokumentation, 307
-Schichtung, 233, 340, 350f, 378, 447
-Spezifikation, 232, 258, 260f, 262ff, 285, 303, 317f, 322, 328, 331, 381, 424
-Struktur, 332
-Zerlegung,
funktionale, 306
-Zusammenbruch, 240
-> Programm
-> Software
Tabellenkalkulation, 254
Task, 318
Teamwork, 334
Termersetzungssysteme, 12
Terminieren, 128, 138, 140
Test, 261, 359, 392, 395
Testen, 22, 235
textuelle Einheit, 323, 325, 333
textuelle Ersetzung, 116, 119, 130
Textverarbeitung, 254
...s-System, 229, 240, 242, 246
Time-Sharing, 318, 425, 428
Token (im Petri-Netz), 273
Top-Down, 63, 228, 233, 305, 327ff, 357
Transaktionssystem, 328
Transformation, 347, 362
Transition, 273, 276, 277, 283
Schalten einer, 273
Turing-Maschine, 12
Übersetzer, 4
-> Compiler
UNIX, 327
Variable, 8, 112ff, 292, 366
globale, 31, 47, 248

Variable,
  Synchronisations-, 281
  ...n-Bezeichner, 118, 366f
    Attribute eines ...s, 366f
    freie, 359
    gebundene, 118
  ...nname, 180
  ...en eines Prozesses, 289
  ...n-Typ, 290f
  -> Prozeß
Verantwortlichkeit, 341
Vererbung, 15, 379, 384, 400ff
  ...sbeziehungen, 385
  ...slehre, objektorientierte, 386
  Implementierung durch, 419
  Mehrfach-, 417ff, 421
  wiederholte, 422
Verfeinerung, 264, 277f
Verflechtungs-Konflikt, 227
Vergröberung, 264
Verifikation, 13
  -> Programmverifikation
Verstehbarkeit, 25, 322, 334
Vertrag, 233, 341, 358, 392
  ...sklausel, 365
VIERECK-class, 401ff
von Neumann-Architektur, 2, 12
von Neumann-Rechner, 5
Vorbedingung, 121, 124, 148, 236, 392, 394
  schwächste, 124ff, 359
Vorfahr, 401
Vorgang, 265, 276
  parallele ...e, 316
  -> Prozeß
Vorgängerstelle, 273, 283
Vorweg-Lesen, 188, 191, 205, 213, 300
Wahrheitswert, 113, 115
Warnier-Orr-Diagramm, 104
Wartbarkeit, 18, 25ff, 51, 234, 245ff, 322f
Wartung, 261
weakest precondition, 125
Werkstatt, 341f, 352, 355
Wertbezeichner, 118, 123, 180, 219
Wetter, 267
WHILE-Anweisung, 124, 135, 157
Wiederverwendbarkeit, 241, 323, 380, 383f, 411, 416
Wiederverwendung, 414, 417
Widerspruchsfreiheit, 379
Wirtschaftlichkeitsbetrachtung, 263
Zauberlehrling, 1, 2
Zeigeinstrument, 258, 342
Zeit, 265f
  Ablauf von, 276
  absolute, 276f
  im JSD, 286
  -scheibe, 318f
  -spanne, 277
Zeit-Effizienz, 19, 27, 50
Zerlegung,
  modulare, 323
Zusicherung
  -> Assertion
Zustand, 265,
  im Petri-Netz 272f, 283
  von Objekten, Vorgängen, 266, 381, 386
  ...sspeicher, 290f
    -Verbindung, 294, 297, 299
Zuverlässigkeit, 230, 240
Zuweisung, 124, 126, 129ff