Herzlich willkommen zum Informatik-Studium an der RWTH Aachen!

Vorlesung Programmierung WS 98/99

Prof. Horst Lichter

Was ist "Informatik I"

- **In der Informatik können wir zwei komplementäre Aspekte unterscheiden:**
  - Beschäftigung mit dem Computer als einer Maschine, die aus ***Hardware-Teilen*** zusammengesetzt ist,
  - Beschäftigung mit dem Computer als einer Maschine, auf der ***Programme*** (Software) ablaufen.
- **Informatik-I beschäftigt sich mit dem *Softwareaspekt*:**
  - Was ist ein Programm?
  - Wie konstruiert (entwickelt) man ein Programm?
  - Wie benutzt man ein Programm?
  - Was kann man theoretisch über Programme sagen?
- **Informatik-I besteht aus den Vorlesungen**
  - ***Programmierung*** (Wintersemester)
  - ***Datenstrukturen*** und Algorithmen (Sommersemester)

Was ist Ziel der LV "Programmierung"

- **Wir werden uns mit der Entwicklung von Programmen beschäftigen:**
  - Was ist ein ***Programm***?
  - Was ist eine ***Programmiersprache***?
  - Mit welchen ***Mitteln*** beschreibt oder konstruiert man ein Programm?
  - Wie können wir es schrittweise aus Komponenten entwickeln?
  - Wie bringen wir es auf den ***Rechner***?
  - Welche unterschiedlichen ***Programmierarten*** kennen wir?
- **Sie sollen grundlegenden *Programmiertechniken* erlernen**
  - Software-Ingenieur ⟺ Künstler
- **Dazu bedienen wir uns der konkreten Programmiersprache *Modula-3***

Welche Ziele haben wir sonst noch

- **Verstehen, was Informatik ist**
  - Was kann man darunter verstehen
  - Was sind die ***Hauptaufgaben*** der Informatik
- **Fähigkeit erwerben, in Gruppen zu arbeiten**
  - ***Teamfähigkeit*** und ***Kommunikation*** sind wichtig
  - An der Hochschule und besonders später im Arbeitsleben
- **Wir wollen Ihnen helfen, sich an die Arbeitsweise an der Hochschule zu gewöhnen**
  - Lehrveranstaltungen sind ***Angebote***
  - Sie entscheiden, was Sie von diesen Angeboten wahrnehmen wollen
  - "Lernen, ***selbständig*** zu Lernen"

Wie wollen wir das erreichen?

- **"Programmierung" ist eine dreiteilige Lehrveranstaltung**
  - Vorlesung
  - Globalübung
  - Gruppenübung
- **Weshalb Vorlesung?**
  - kompakte Vermittlung von Wissen und Erfahrung
  - Möglichkeit zur Abstimmung und Rückkopplung
- **Weshalb Übung?**
  - das Gelernte überprüfen und vertiefen
  - in der Gruppe arbeiten
  - etwas und sich selbst darstellen lernen
  - Handwerkszeug handhaben lernen
  - Hilfe zur Selbsthilfe

Nutzen Sie diese Angebote!
Es ist ihre Lehrveranstaltung

Was läuft wo?

- **In der Vorlesung:**
  - Einführung in die Programmierung
  - Begriffe, Konzepte und Beispiele
  - Ergänzende und vertiefende Themen
- **In der Globalübung:**
  - Besprechung der Lösungen zu den Übungsaufgaben
  - Vorstellen von Themen, die für alle relevant sind (z.B. Arbeiten mit einer Programmierumgebung)
- **In der Gruppenübung**
  - Fragen der Vorlesungsthemen
  - Diskussion von speziellen Aspekten der Übungsaufgaben
  - Präsentation von Lösungen

Spezialist und Novize - 1

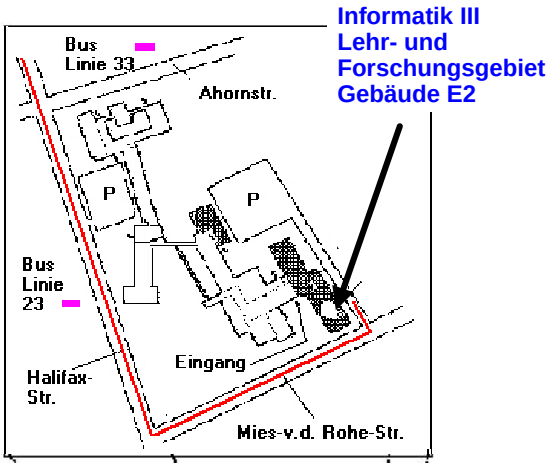
- **Viele haben bereits Erfahrungen ("Spezialisten")**
  - im Umgang mit Rechnern
  - im Programmieren mit einer Programmiersprache
  - aus eigenem Interesse am Thema oder durch Kurse am Gymnasium
- **Andere haben keine oder nur wenig Vorkenntnisse ("Novizen")**
  - alles ist neu und ungewohnt
  - Angst, schlechter als die "Spezialisten" zu sein

Diese Angst ist unbegründet !

Wir erarbeiten systematisch das Gebiet der Programmierung!

Wer macht was?

- **Vorlesung**
  - Horst Lichter
  - Raum: E2 - 6210
  - Sprechstunde:
    - Do, 12:00 - 13:00
- **Globalübung**
  - Moritz Schnizler
  - Raum: E2 - 6207
  - Sprechstunde:
    - Di, 12:00 - 13:00
- **Gruppenübung**
  - viele studentische Hilfskräfte



Termine, Termine !

- **Vorlesung**
  - Montag: 13:30 - 15:00 Ro
  - Donnerstag: 10:50 - 12:20 Ro
- **Globalübung**
  - Dienstag: 17:30 - 19:00 AH IV , Beginn: 27. Oktober
- **Gruppenübungen**
  - Donnerstag: 14:00 - 15:30 (1) Beginn: 29. Oktober
    15:45 - 17:15 (1)
    16:15 - 17:45 (2)
    17:15 - 18:45 (2)
    17:30 - 19:00 (1)
  - Freitag: 08:15 - 09:45 (4) Beginn: 30. Oktober
    10:00 - 11:30 (2)
    14:00 - 15:30 (2)

Wie zu was anmelden?

- **Anmeldung zu den Gruppenübungen**
- **Listen für die einzelnen Übungsgruppen hängen aus:**
  - Gebäude E2, 2. Etage, linker Flur
  - ab heute
  - bis ***Freitag 16.10.98, 17:00***
- **Bitte die Listen gleichmäßig ausfüllen !**
  - Bitte nur den vorgesehenen Platz verwenden!
- **Übungen werden in Kleingruppen bearbeitet und abgegeben:**
  - mindestens zwei, maximal drei Personen pro Kleingruppe
  - "Ein-Personengruppen" werden nicht akzeptiert!

Literatur zur Vorlesung

- **Diese Vorlesung stützt sich in großen Teilen auf die folgenden Materialien:**
  - Hans-Jürgen Appelrath, Jochen Ludewig: "***Skriptum Informatik - eine konventionelle Einführung***", B.G. Teubner Stuttgart,1995 .
  - László Böszörményi, Carsten Weich: "***Programmieren mit Modula-3 - Eine Einführung in stilvolle Programmierung*** ", Springer Verlag,1995.
  - Robert W. Sebesta: "***Concepts of Programming Languages***". Benjamin Cummings, 2. Auflage, 1993.
- **Wenn andere Quellen verwendet werden, wird dieses entsprechend angemerkt.**
- **Arbeiten an der Hochschule**
  - selbständig Inhalte bestimmen (besonders im Hauptstudium)
  - selbständig Inhalte erarbeiten, Umgang mit ***Literatur***

Unterlagen zur Vorlesung

- **Stehen im "world wide web" zur Verfügung**
  - ***http:/www-lufgi3.informatik.rwth-aachen.de/Programmierung***
    - Neuigkeiten,
    - Folien der Vorlesungen,
    - Übungsblätter,
    - Lösungen
- **Folien der Vorlesungen**
  - werden - nach Wunsch - in zwei Teilen vervielfältigt und abgegeben
    - Ende Dezember
    - Ende Wintersemester

Was erwarten wir von Ihnen?

- **Dies ist *Ihre* Lehrveranstaltung!**
- **Mitarbeit**
  - die LV ist keine ***Beschäftigungstherapie***
  - und kein Kabelprogramm
  - ohne Arbeit keine ***bleibenden Ergebnisse***
  - ohne ***aktive Mitarbeit*** keine neue Sichtweise
- **Gruppenarbeit**
  - ohne Gruppenarbeit keine ***Qualifikation*** zur Softwareentwicklung
  - ohne Gruppenarbeit wird das Studium ***zäh bis erfolglos***
- **Rückkopplung:**
  - über Inhalte
  - über Formen
  - über Probleme

Prüfung !

- **Die Diplom-Prüfungsordnung (DPO) regelt, welche Prüfungen Sie ablegen müssen.**
- **Vordiplomsprüfung**
  - Fachprüfung "Informatik I"
    - LV "Programmierung"
    - LV "Datenstrukturen"
- **Zulassung:**
  - Leistungsnachweis "Programmierung"
  - Übungsschein "Programmierung"

Diesen Übungsschein sollten Sie in dieser Veranstaltung erwerben!

Prüfungsmodalitäten ??

- **Wer erhält einen Übungsschein?**
  - Alle, die die am Semesterende angebotene "***Übungsscheinklausur***" bestanden haben.
- **Scheinklausur**
  - Dauer: 2 Stunden
  - Termin: 10. Februar 1999
  - Zeit: 15.45 -17.15
  - Ort: Roter Hörsaal, AH IV

- **Wer darf an der Scheinklausur teilnehmen?**
  - Alle, die ***50*** % der gestellten ***Übungsaufgaben*** gelöst haben!
- **Wieviel Übungsaufgaben gibt es?**
  - 13 ***Übungsblätter*** (mit unterschiedlicher Anzahl Aufgaben)
  - pro Übungsblatt werden ***20 Punkte*** vergeben!

Übungsbetrieb

- **Woche n**
  - **Montag**: ***Ausgabe*** des Übungsblatts ***n***
Abgabe der Lösungen zu Übungsblatt ***n-1***
  - **Mittwoch**: Globalübung
Vorstellung der Lösungen zum Übungsblatt ***n-1***
  - **Donnerstag**: Diskussion in den Gruppenübungen
  - **Freitag**: Diskussion in den Gruppenübungen
- **Ausgabe der Übungsblätter**
  - Montag, nach der Vorlesung, Gebäude E2, 2. Etage, linker Flur
- **Abgabe der Übungen**
  - Montag bis ***spätestens 13:00***
  - Gebäude E2, 2. Etage, linker Flur
- **Erstes Übungsblatt: *Montag 19. Oktober***

Informationen zum Rechnerbetrieb - 1

- **Sie benötigen Zugang zu den Rechnern, um**
  - auf die "online" zur Verfügung gestellten Informationen zugreifen zu können
  - um Programmieraufgaben lösen zu können
- **Rechner werden im sogenannten "Rechnerpool Informatik" zur Verfügung gestellt**
  - 19 Workstations unter Windows NT
  - 33 SUN-Workstations unter Solaris (Unix)
  - 13 HP-Workstations unter HP-UX (Unix)
- **Öffnungszeiten**
  - Mo - Do : 9:00 - 19:00, Fr 9:00 -17:00
  - In der ersten Gruppenübungsstunde werden Anträge für Benutzerkennungen ausgegeben!

Informationen zum Rechnerbetrieb - 1

- Folgende Zeiten sind für Hörer der Veranstaltung "Programmierung" reserviert:

- Montag: 15.00 - 19.00
- Dienstag: 09.00 - 12.00
- Mittwoch: 09.00 - 12.00
- Freitag: 09.00 - 12.00
  15.00 - 17.00

Geschichte der Informatik Grundlagen

- Was ist Informatik
- Geschichte der Informatik
- Algorithmus
- Software, Programm, Programmieren
- Der von-Neumann Rechner

Was ist Informatik?

- **Der Begriff "Informatik"**
  - ein ***Kunstwort*** , das zu Beginn der 60er Jahre zur Bezeichnung einer sich neu entwickelnden Disziplin geschaffen wurde. Es setzt sich aus Bestandteilen der beiden Worte ***Information*** und ***Mathematik*** zusammen. Darin kommt zum Ausdruck, daß Informatik die Wissenschaft von der Informationsverarbeitung ist, und eine große Nähe zur Mathematik hat.

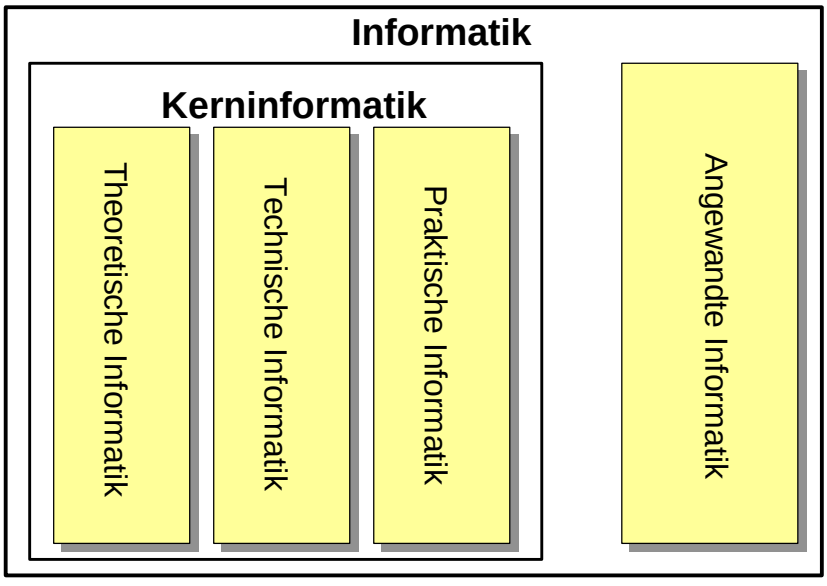
- **Informatik ist die Wissenschaft**
  - von der ***systematischen*** Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von ***Computern*** (vgl. DUDEN Informatik, 1993)
- **Informatik versteht sich als Wissenschaft**
  - der Analyse, Konzeption und Realisierung von Systemen, die aus miteinander und mit ihrer Umwelt kommunizierenden Akteuren bestehen (vgl. Studienführer Informatik, RWTH Aachen, 1998)

Hauptaufgaben der Informatik

- **Hauptaufgabe der Informatik ist die Entwicklung**
  - ***formaler, maschinell ausführbarer Verfahren*** zur Lösung von Informationsverarbeitungsproblemen, die häufig als Teilprobleme komplexer Kommunikations- oder Organisationsprobleme auftreten.

- **Die Forderung der Durchführbarkeit mittels einer Maschine (im allgemeinen eines Digitalrechners) bedingt,**
  - daß die zu verarbeitenden Informationen als ***maschinell verarbeitbare Daten*** dargestellt werden, und
  - daß die Lösungsverfahren bis ***ins kleinste Detail*** formal beschrieben werden.

Gebiete der Informatik -1



Gebiete der Informatik - 2

- **Theoretische Informatik**
  - ***Formale Modelle*** zur Beschreibung und Untersuchung von Algorithmen und Computern
  - Teilgebiete: Formale Sprachen, Automatentheorie, Komplexitätstheorie etc.
- **Technische Informatik**
  - Funktioneller ***Aufbau von Computern***, Entwurf und Entwicklung von Rechnern, Geräten und Schaltungen
  - Teilgebiete: Rechnerarchitektur, VLSI-Entwurf etc.
- **Praktische Informatik**
  - Prinzipien und Techniken der ***Programmierung*** im ***Kleinen*** und im ***Großen***
  - Teilgebiete: Software Engineering, Informationssysteme, Compilerbau, Künstliche Intelligenz, Betriebssysteme etc.

Gebiete der Informatik - 3

- **Angewandte Informatik**
  - ***Anwendung*** der Methoden der ***Kerninformatik*** in anderen Wissenschaften
    - Entwicklung ***spezieller*** Verfahren und Darstellungstechniken
  - Bindestrich-Informatik:
    - Wirtschafts-Informatik, Medizin-Informatik, Bio-Informatik, Rechts-Informatik
  - Grenzen zwischen ***Praktischer Informatik*** und ***Angewandter Informatik*** sind zum Teil fließend.
- **Das, was man unter Informatik versteht, kann man in solchen Definitionen jedoch nicht *endgültig* fassen. Schließlich ändert sich die Beschreibung der Definitionen einer Wissenschaft, wie in anderen Fällen auch, über die Zeit.**

Geschichte der Informatik - 1

- **Altertum–Mittelalter:**
  - Verwendung des ***Abakus*** (Brett mit verschiebbaren Kugeln) als Hilfsmittel für die vier Grundrechenarten.
- **9. JH.:**
  - Der arabische Mathematiker und Astronom ***Ibn Musa Al-Chwarismi*** schreibt das Lehrbuch “Kitab al jabr w' almuqabala” (“Regeln der Wiedereinsetzung und Reduktion”). Das Wort “***Algorithmus***” geht auf seinen Namen zurück.
- **1547: Adam Riese (1492–1559)**
  - veröffentlicht ein Rechenbuch, in dem er die ***Rechengesetze*** des aus Indien stammenden ***Dezimalsystems*** (5. Jh.n. Chr.) beschreibt. Im 17. Jahrhundert setzt sich das Dezimalsystem in Europa durch.
- **Wilhelm Schickard (1592–1635)**
  - konstruiert für seinen Freund Kepler (1571–1630) eine ***Maschine***, die addieren, subtrahieren, multiplizieren und dividieren kann. Sie bleibt unbeachtet.
- **1641: Blaise Pascal (1623–1662)**
  - konstruiert eine Maschine, mit der man ***sechsstellige Zahlen*** addieren kann.

Geschichte der Informatik - 2

- **1674: Gottfried Wilhelm Leibniz (1646–1716)**
  - konstruiert eine ***Rechenmaschine*** mit Staffelwalzen für die vier ***Grundrechenarten***. In diesem Zusammenhang befaßt er sich auch mit der binären Darstellung von Zahlen.
- **1774: Philipp Matthäus Hahn (1739–1790)**
  - entwickelte eine mechanische Rechenmaschine, die ***erstmals zuverlässig*** arbeitet.
- **Ab 1818:**
  - Rechenmaschinen nach dem Vorbild der Leibniz'schen Maschine werden serienmäßig hergestellt und dabei ständig weiterentwickelt.
- **1838: Charles Babbage (1792–1871)**
  - plant eine Maschine, die “***Analytical Engine***”, bei der die Reihenfolge der einzelnen Rechenoperationen durch nacheinander eingegebene ***Lochkarten*** gesteuert wird.
- **1886: Hermann Hollerith (1860–1929)**
  - entwickelt in den USA elektrisch arbeitende ***Zählmaschinen für Lochkarten***, mit denen die statistischen Auswertungen der Volkszählungen vorgenommen werden.

Geschichte der Informatik - 3

- **1934: Konrad Zuse (1910–1995)**
  - beginnt mit der Planung einer ***programmgesteuerten Rechenmaschine***. Sie verwendet das binäre Zahlensystem.
- **1937: Die mechanische Anlage *Z 1* von Zuse ist fertig.**
- **1941: Die elektromechanische Anlage Z 3 von Zuse ist fertig.**
  - Dies ist der ***erste funktionsfähige programmgesteuerte Rechenautomat***. Das Programm wurde mit ***Lochstreifen*** eingegeben. Die Anlage verfügt über 2000 Relais und eine Speicherkapazität von 64 Worten a 22 Bit. Multiplikationszeit: etwa 3 s.
- **1944: Howard H. Aiken (1900–1973)**
  - erstellt in Zusammenarbeit mit der Harvard-University und der Firma IBM die teilweise programmgesteuerte Rechenanlage MARK I. Additionszeit 1/3 s, Multiplikationszeit: 6 s.
- **1946: J. P. Eckert und J. W. Mauchly**
  - stellen die ENIAC (Electronic Numerical Integrator and Automatic Calculator) fertig. Dies ist der erste ***voll elektronische Rechner*** (18.000 Elektronenröhren). Multiplikationszeit: 3 ms.

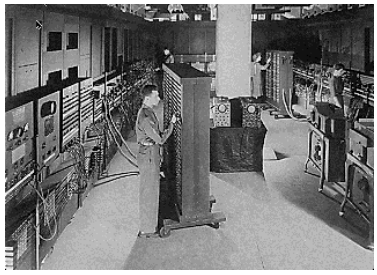
Geschichte der Informatik - 4

- **1946–1952:**
  - Auf der Grundlage der Ideen **John v. Neumanns** (1903–1957) (Einzelprozessor, Programm und Daten im gleichen Speicher; Von-Neumann-Rechner) und seiner Kollegen am Institute of Advanced Study at Princeton (H.H.Goldstine, A.W.Burks) werden weitere Computer in Universitätslabors entwickelt (“Pionierzeit”).
- **1949: M.V.Wihls (University of Manchester)**
  - stellt mit der EDSAC (Electronic Delay Storage Automatic Calculator) den **ersten universellen Digitalrechner** (gespeichertes Programm) fertig.
- **Ab 1950:**
  - **Industrielle** Rechnerentwicklung und -produktion.

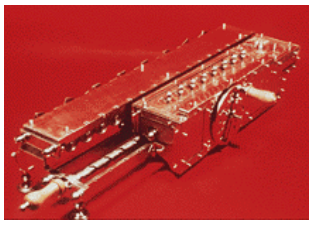
Historische Rechner - 1

Schickard

ENIAC



Leibniz



MARK1

Algorithmus

- **Ein Algorithmus ist ein Verfahren, welches**
  - in einem ***endlichen*** Text niedergelegt werden muß
  - ***effektiv*** ausführbar ist,
  - Elementaroperationen enthält, die durch die jeweilige Situation ***eindeutig*** bestimmt sind
  - ***Ein- und Ausgabe*** ermöglicht
  - durch eine mechanisch oder elektronisch arbeitende *Maschine* ***ausgeführt*** werden kann
- **Anzahl und Ausführungszeit der Elementaroperationen sind beschränkt**
- **Ein Algorithmus wird entsprechend seiner Vorschrift schrittweise ausgeführt**
  - die ausführende Instanz muß die Vorschrift ***interpretieren*** und ***korrekt*** ausführen
  - ein Algorithmus ***terminiert***, wenn er nach endlich vielen Schritten abbricht

Euklidischer Algorithmus - 1

Euklidischer Algorithmus

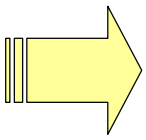
- Problem:
  - Man bestimme zu je zwei natürlichen Zahlen n und m den größten gemeinsamen Teiler ggt(n,m)

Algorithmus

	WENN n < m ist,	DANN vertausche man n und m
❷	WENN m = 0 ist,	DANN ist n der ggt(n,m) und man beende den Algorithmus
❸	WENN m≠ 0 ist,	DANN bilde man den Rest r, der bei der Division von n durch m bleibt, dann ersetze man n durch m und m durch r und beginne von vorn.

Euklidischer Algorithmus - 2

n = 6
m = 4



Erster Durchlauf

- da 6 < 10, so setzt man n = 10 und m = 6
- da 6 ≠ 0 ist, gehen zu Schritt 3
- r = 4. Man erhält also n = 6 und m = 4

Zweiter Durchlauf

- Da 6 ≥ 4 ist, gehe zu Schritt 2
- Da 4 ≠ 0 ist, gehe zu Schritt 3
- r = 2. Man erhält n = 4 und m = 2

Dritter Durchlauf

- Da 4 ≥ 2 ist, gehe zu Schritt 2
- Da 2 ≠ 0 ist, gehe zu Schritt 3
- r = 0. Man erhält n = 2 und m = 0

Abbruch

- Da 2 ≥ 0 ist, gehe zu Schritt 2
- Da m = 0 ist, ist ggt(6,4) = 2

Eigenschaften von Algorithmen - 1

- **Abstraktion**
  - ein Algorithmus löst i. a. eine ***Klasse von Problemen*** (z.B. Suchen eines Musters in einer Zeichenkette).
- **Finitheit**
  - statisch finit: ein Algorithmus besitzt eine ***endliche Länge***
  - dynamisch finit: während der Abarbeitung darf nur ***endlich viel Speicherplatz*** belegt werden
- **Terminierung**
  - terminierend: nach ***endlich vielen Schritten*** liegt ein Resultat vor
  - sonst ***nicht-terminierend*** (z.B. Steuerungsalgorithmen)
- **Determinismus**
  - deterministisch: zu jedem Zeitpunkt besteht ***höchstens eine*** Möglichkeit der Fortsetzung
  - nicht-deterministisch: an mindestens einer Stelle gibt es eine ***Wahlmöglichkeit*** für die Fortsetzung

Eigenschaften von Algorithmen - 2

- **Determiniertheit**
  - determiniert: bei ***gleichen Eingaben*** und Startbedingungen wird das ***gleiche Ergebnis*** erzielt
  - nicht-determiniert: es werden ***unterschiedliche Ergebnisse*** erzielt (Anwendung von heuristischen Methoden)
- **Bemerkung**
  - ein terminierender, deterministischer Algorithmus ist immer determiniert
  - ein terminierender, nicht-deterministischer Algorithmus kann determiniert oder nicht-determiniert sein.

Terminierung - Nichtterminierung

- **Sei**
  - power : $N^+ \times N \rightarrow N$

$$\text{power}(a, b) = \begin{cases} 1 & \text{falls } b = 0 \\ a * \text{power}(a, b-1) & \text{sonst} \end{cases}$$

Dieser Algorithmus ***terminiert*** im Definitionsbereich.

  - Wir weiten den Definitionsbereich aus:

$$\text{power2} : Z \times Z \rightarrow Z$$

$$\text{power2}(a, b) = \begin{cases} 0 & \text{falls } a = 0 \text{ und } b > 0 \\ 1 & \text{falls } a \neq 0 \text{ und } b = 0 \\ a * \text{power2}(a, b-1) & \text{sonst} \end{cases}$$

Dieser Algorithmus ***terminiert nicht*** für power2 (0,0) und power2 (a,b) mit $b < 0$ und $a \neq 0$)

Anmerkung zur Nichtterminierung

- **Definition:**

$$\text{power2} : Z \times Z \rightarrow Z$$

$$\text{power2}(a, b) = \begin{cases} 0 & \text{falls } a = 0 \text{ und } b > 0 \\ 1 & \text{falls } a \neq 0 \text{ und } b = 0 \\ a * \text{power2}(a, b-1) & \text{sonst} \end{cases}$$

Während wir die Nichtterminierung bei der ***manuellen Auswertung*** leicht feststellen können, fällt dies auf dem Rechner sehr viel schwerer. Solange der Rechner "***still vor sich hin***" rechnet, können wir den Unterschied zwischen einem ***nicht-terminierenden*** und einem ***sehr langwierigen Algorithmus*** von außen nicht feststellen.

Fragen im Kontext von Algorithmen

- **Wie kann man aus einer Lösungsidee einen Algorithmus konstruieren?**
  - "schrittweise Programmentwicklung"
- **Wie kann man Algorithmen darstellen?**
  - "Flußdiagramme", Programme
- **Wie beweist man, daß ein Algorithmus tatsächlich das tut, was er tun soll?**
  - Verifikation
- **Wie "gut" ist ein Algorithmus?**
  - Speicherverbrauch, benötigte Zeit
  - Aufwandsabschätzungen

Typische Problemklassen

- **Sortieralgorithmen**
  - Ordnen von Elementen
- **Suchalgorithmen**
  - Auffinden von Elementen
- **Algorithmen zur Verarbeitung von Zeichenfolgen**
  - Mustererkennung, Verschlüsselung, Komprimierung
- **Geometrische Algorithmen**
  - Schnittmenge von geometrischen Objekten
- **Algorithmen für Graphen**
  - Suchen im Graph, kürzester Weg
- **Mathematische Algorithmen**
  - Rechnen mit Polynomen und Matrizen

Software, Programm

- **Neben der Entwicklung der Hardware (Rechner)**
  - wurden seit ca. 1940 ***Programmiersprachen*** entwickelt, um Algorithmen zu formulieren, damit sie von einem Rechner ausgeführt werden können.
- **Rechner "realisieren" Algorithmen,**
  - durch ***schrittweise Abarbeitung*** von ***Programmen***, die in einer Programmiersprache geschrieben sind.
- **Um einen Rechner zu programmieren,**
  - muß die ***Syntax*** und ***Semantik*** der verwendeten Programmiersprache bekannt sein
- **In diesem Zusammenhang spricht man häufig auch von Software**
  - "***Software***" und "***Programm***" sind aber nicht dasselbe

Definition: Software

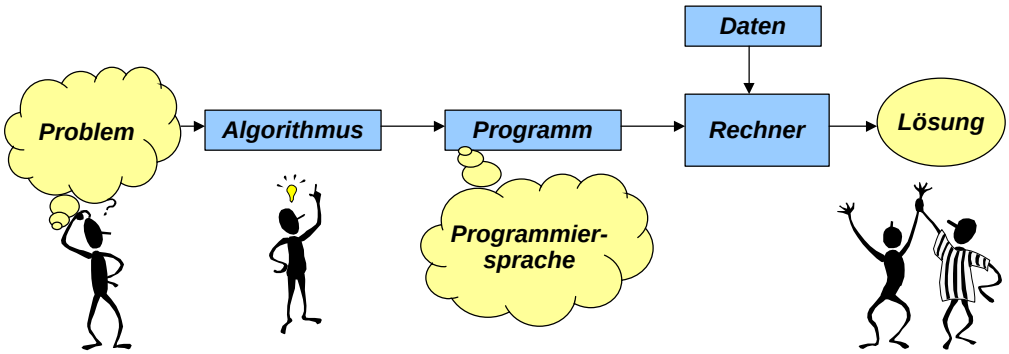
- **1. Definition: Software**
  - Informatik-Duden: Gesamtheit ***aller Programme***, die auf einer Rechenanlage eingesetzt werden können
    - ***Systemsoftware***: Programme die für den korrekten Ablauf einer Rechenanlage notwendig sind
    - ***Anwendungssoftware***: dient zur Lösung von Benutzerproblemen
- **2. Definition: Software**
  - IEEE Standrad Glossary of Software Engineering Terminoloy
    - "Computer ***programs***, ***procedures***, and possibly associated ***documentation*** and ***data*** pertaining to the operation of a computer system."

Definition: Programm

- **1. Definition: Programm**
  - Formulierung eines Algorithmus und der dazugehörigen Datenbereiche in einer Programmiersprache
    - sind ***exakt*** (formal) definiert
    - nehmen Bezug auf eine bestimmte Darstellung der ***Daten***
    - sind auf einer Rechenanlage ***ausführbar***
- **2. Definition: Software**
  - IEEE Standrad Glossary of Software Engineering Terminoloy
    - "A combination of ***computer instructions*** and ***data definitions*** that enable computer hardware to ***perform*** computational or control functions".

Programmieren

- **Unter dem Begriff Programmieren versteht man**
  - das **Lösen von Problemen** unter Zuhilfenahme eines **Rechners**

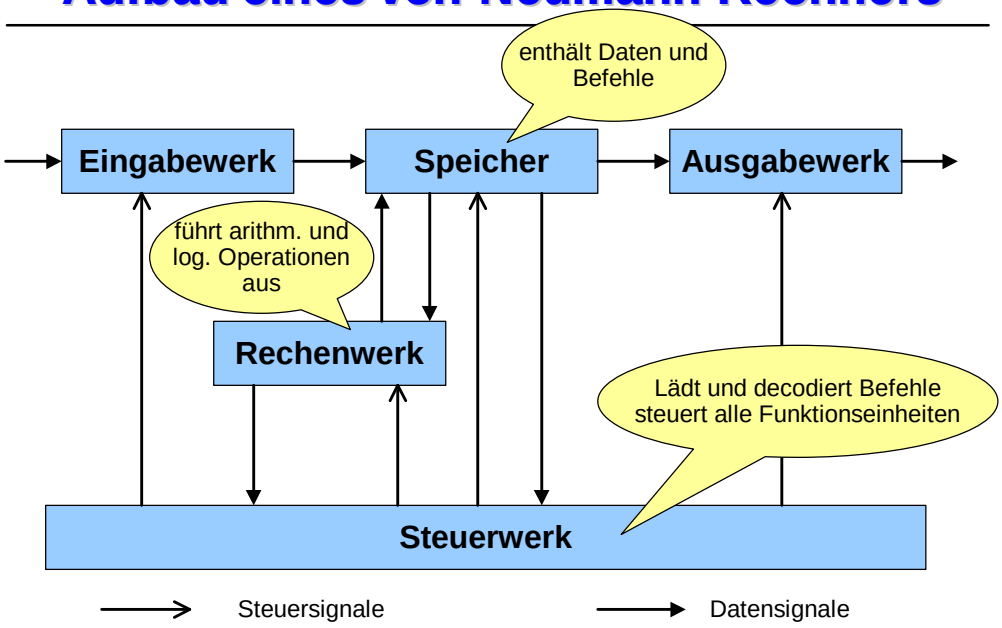


- **Programmieren [1] Software-Entwicklung**
- **Programmieren ist *ein Teil* der Software-Entwicklung**

Von-Neumann-Rechnerarchitektur

- **Definiert die wesentlichen Elemente eines Universalrechners**
  - Rechner soll nicht für eine ***bestimmte*** Problemklasse konstruiert sein
  - Zur Lösung des Problems muß ein ***Programm*** eingegeben und in den ***Speicher*** abgelegt werden
- **1946 von John von Neumann als Konzept für die EDVAC vorgeschlagen**
- **fast alle heutigen Rechner basieren darauf und sind Weiterentwicklungen davon**
  - Nicht-von-Neumann-Rechner sind Gegenstand der Forschung
- **diese Architektur prägt viele Programmiersprachen (imperative Programmiersprachen)**

Aufbau eines von-Neumann-Rechners



Aufbau des Steuerwerks

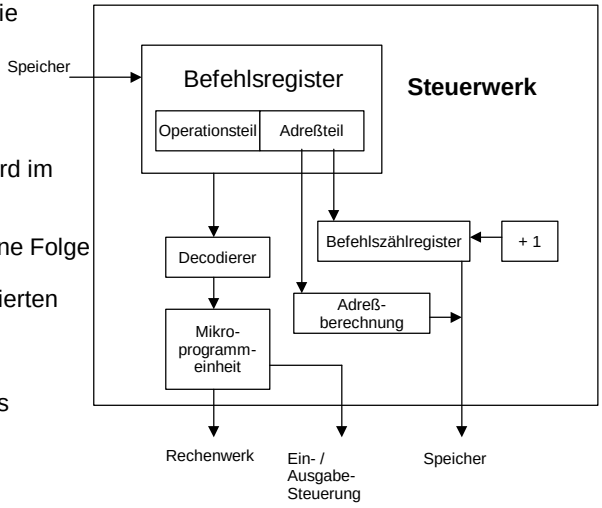
Lädt, decodiert und interpretiert die Befehle

Befehlsregister enthält den aktuellen Befehl

Der Operationsteil des Befehls wird im Decodierer entschlüsselt

Mikroprogrammeinheit erzeugt eine Folge von Signalen zur Ausführung des Befehls (abhängig von der decodierten Information)

Das Befehlzählregister speichert die Adresse des nächsten Befehls



Aufbau des Rechenwerks

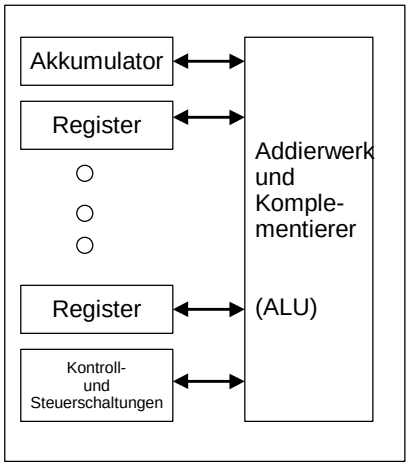
Führt arithmetische und logische Operationen durch (ALU)

erhält vom Steuerwerk die benötigten Operanden

wesentliche Einheiten sind Addierer und Komplementierer (damit können die Grundrechenarten durchgeführt werden)

implementiert Algorithmen für Multiplikation und Division

zusammen mit dem Steuerwerk nennt man es auch CPU



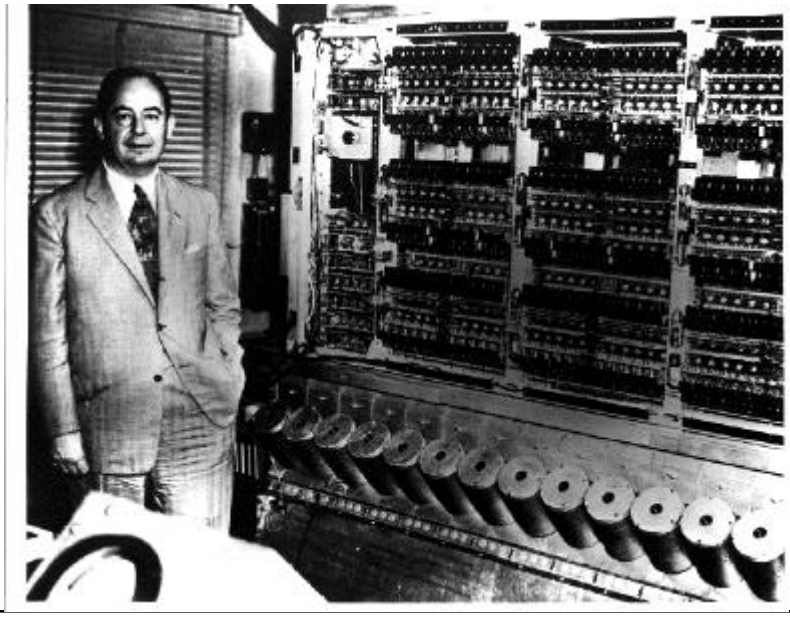
Von-Neumann-Prinzipien - 1

- Der Rechner besteht aus *Steuerwerk, Rechenwerk, Speicher, Eingabewerk* und Ausgabewerk.
- Die Struktur des von-Neumann-Rechners ist *unabhängig* von den zu bearbeitenden Problemen. Zur Lösung eines Problems muß von außen das *Programm* eingegeben und im Speicher abgelegt werden.
- Programme, Daten, Zwischen- und Endergebnisse werden in *demselben Speicher* abgelegt.
- Der Speicher ist in *gleichgroße Zellen* unterteilt, die fortlaufend durchnumeriert sind. Über die Nummer (*Adresse*) einer Speicherzelle kann deren Inhalt abgerufen oder verändert werden.
- Aufeinanderfolgende Befehle eines Programms werden in *aufeinanderfolgenden* Speicherzellen abgelegt. Das Ansprechen des nächsten Befehls geschieht vom Steuerwerk aus durch Erhöhen der Befehlsadresse um Eins.
- Durch *Sprungbefehle* kann von der Bearbeitung der Befehle in der gespeicherten Reihenfolge abgewichen werden.

Von-Neumann-Prinzipien - 2

- Es gibt zumindest
  - *arithmetische* Befehle wie Addieren, Multiplizieren usw.;
  - *logische* Befehle wie Vergleiche, logisches nicht, und, oder usw.;
  - *Transportbefehle*, z.B. vom Speicher zum Rechenwerk und für die Ein-/Ausgabe;
  - bedingte *Sprünge*.
  - Weitere Befehle wie Schieben, Unterbrechen, Warten usw. kommen hinzu.
- Alle Daten (Befehle, Adressen usw.) werden *binär codiert*. Geeignete Schaltwerke im Steuerwerk und an anderen Stellen sorgen für die richtige Entschlüsselung (*Decodierung*).

Historische Rechner - J. v. Neumann



Was haben wir gelernt!

- **Einordnung der Informatik als Wissenschaft**
  - Aufgabe der Informatik
  - Einteilung der Informatik
- **Wo kommt die Informatik her**
  - Entwicklung der Rechenmaschinen
- **Was versteht man unter einem Algorithmus**
  - Eigenschaften von Algorithmen
- **Was versteht man unter den zentralen Begriffen**
  - Software
  - Programm
  - Programmieren
- **Wir wissen, wie ein von-Neumann-Rechner aufgebaut ist**

Programmiersprachen Grundlagen

- **Was ist eine Programmiersprache**
- **Syntax und Semantik von Sprachen**
- **Grammatik einer Sprache**
- **EBNF und Syntaxdiagramme**
- **Entwicklung der Programmiersprachen**
- **Warum arbeiten wir mit Modula-3**

Programmiersprachen - Definition

- **Programmiersprachen sind *künstliche Sprachen* (keine natürlichen Sprachen), deren Syntax und Semantik definiert ist.**
- **Syntax:**
  - Die ***Syntax*** einer Sprache S ist die Definition aller in S ***zulässigen Aussagen***, die in einer Sprache formuliert werden können (Wörter).
- **Semantik:**
  - Die ***Semantik*** einer Sprache S ist die Definition der den zulässigen Aussagen zugeordneten ***Bedeutungen***.
  - Syntaktische ***falsche*** Aussagen haben ***keine*** Semantik
  - Aber auch syntaktisch korrekte Aussagen haben nicht immer eine Semantik
    (z.B. ein Programm, in dem durch 0 dividiert wird)

Beispiel: Syntax, Semantik

- **Natürliche Zahlen**
  - (ohne die Null) dargestellt im Dezimalsystem in arabischen Ziffern bilden eine einfache künstliche Sprache
  - ***Syntax***:
    - jede Zahl ist eine Sequenz von Ziffern (0,1, .. , 9), wobei die erste Ziffer nicht 0 ist
  - ***Semantik***:
    - der Wert einer Zahl ist definiert als der Wert ihrer letzten Ziffer, vermehrt um den zehnfachen Wert der links davon stehenden Zahl, falls diese vorhanden ist (***rekursive*** Definition)
- **Beispiel**
  - syntaktisch ***korrekt*** ist: 367 Semantik: 7+10* (36)
7+10*(6+10*(3))
7+10*(6+10*3)
  - syntaktisch ***falsch*** ist: 007 keine Semantik

Alphabet

- **Alphabet**
  - Ein Alphabet ist eine ***nichtleere endliche*** Menge von ***unterscheidbaren*** Zeichen ("Buchstaben")

    A = {a_1, a_2, a_3, …} mit einer ***Ordnungsrelation*** ≤ ($a_1 \le a_2 \le a_3$ ….)
  - **Beispiel**:
    - das lateinische Alphabet (a, b, c,.. z)
    - der ASCII-Code (bestehend aus 128 Zeichen)
    - das deutsche ABC ist kein Alphabet im definierten Sinn (Umlaute!)
  - **Wort über einem Alphabet**
    - ***endliche Folge*** von Buchstaben, die auch ***leer*** sein kann (ε leere Wort)
    - A* bezeichnet die ***Menge aller Wörter*** über dem Alphabet A (inkl. dem leeren Wort)

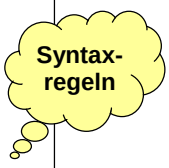
Formale Sprache

- **Definition:**
  - Sei A ein Alphabet. Eine (formale) Sprache (über A) ist ***jede beliebige Teilmenge von A****.
- **Beispiele:**
  - A = { 0,1}, A* = {ε , 0, 1, 01, 10, 10, 000, 100 , … }
  - L = {0, 1, 10, 11, 100, 101, .. } ∈ A*, die Menge der Binärdarstellungen natürlicher Zahlen (mit Null, ohne führende Nullen)
  - A = {(,), +, -, *, /, a}, A* = {ε, (), (+-a), (a*a), … }
  - die Sprache der korrekt geklammerten Ausdrücke EXPR ∈ A* : EXPR = { (((a))), (a+ b), (a -a)*a+ a /(a+ a) -a, .. }
- **Da solche Sprachen i.d.R. unendlich viele Wörter enthalten, benötigt man eine endliche Beschreibungsvorschrift**
  - ***Grammatik***, die die Sprache erzeugt
  - ***Automat***, der die Sprache erkennt

Grammatik - informal - 1

- **Definiert *Regeln*, die festlegen, welche Wörter über einem *Alphabet* zur Sprache gehören und welche nicht.**
- **Beispiel: Grammatik für "Hund-Katze-Sätze"**

1	<Satz>	->	<Subjekt> <Prädikat> <Objekt>
2	<Subjekt>	->	<Artikel> <Attribut> <Substantiv>
3	<Artikel>	->	ε
4	<Artikel>	->	der
5	<Artikel>	->	die
6	<Artikel>	->	das
7	<Attribut>	->	ε
8	<Attribut>	->	<Adjektiv>
9	<Attribut>	->	<Adjektiv> <Attribut>
10	<Adjektiv>	->	kleine
11	<Adjektiv>	->	bissige
12	<Adjektiv>	->	große
13	<Substantiv>	->	Hund
14	<Substantiv>	->	Katze
15	<Prädikat>	->	jagt
16	<Objekt>	->	<Artikel> <Attribut> <Substantiv>



Grammatik - informal - 2

- **Grammatik für "Hund-Katze-Sätze"**
  - durch diese Grammatik können z.B. die folgenden Sätze gebildet (abgeleitet) werden
    - "der kleine bissige Hund jagt die große Katze"
    - "die kleine Katze jagt der bissige Hund"
    - "das große Katze jagt der kleine große bissige kleine …. Katze"
  - folgende "Sätze" werden nicht durch diese Grammatik gebildet
    - "die Katze der Hund"
    - "Katze und Hund"
    - "der Hund jagt die Katze die jagt Hund"

Grammatik - Definition - 1

- **Definition:**
  - Eine Grammatik G für eine Sprache L ist definiert durch
  - ein ***Viertupel (N, T, P, S)***
- **N: Menge der *Nichtterminalsymbole***
  - sind Zeichen oder Zeichenfolgen für syntaktische ***Abstraktionen***
  - Beispiel: <Satz>, <Artikel>
  - kommen nicht in den Wörtern der Sprache vor
  - werden durch Anwendung der ***Produktionsregeln*** solange ersetzt, bis nur noch Terminalsymbole übrig sind
- **T: Menge der *Terminalsymbole***
  - sind Zeichen oder Zeichenfolgen des Alphabets, aus denen die Wörter der Sprache bestehen
  - Beispiel: Hund, der, bissige

Grammatik - Definition - 2

- **P: Menge von *Produktionsregeln***
  - definieren, wie aus bekannten Konstrukten ***neue Konstrukte*** geschaffen werden
  - Die Anwendung einer Regel bedeutet, daß in dem bereits erzeugten Konstrukt der Teil, der der linken Seite der Regel entspricht, durch die rechte Seite ***ersetzt*** wird
  - **Beispiel**:
    - der kleine bissige Hund <Prädikat> <Objekt>
    - der kleine bissige Hund *jagt* <Objekt>

    
  - Jedes durch Anwendung der Regeln erzeugbare Wort, ***das nur aus Terminalsymbolen besteht***, gehört zu der von der Grammatik erzeugten Sprache L(G)
- **S: das *Startsymbol***
  - ist ein spezielles Nichtterminalsymbol, aus dem ***alle Wörter*** der Sprache mit Hilfe der Grammatik erzeugt werden
  - **Beispiel**: <Satz>

Grammatik - Definition - 3

- **Es gilt:**
  - $N \cap T = \varnothing$
  - $V = N \cup T$ (Gesamtalphabet, Vokabular)
  - sei $p \in P$: $(\alpha \rightarrow \beta)$, $\alpha \in V^* N V^*$, $\beta \in V^*$

Terminale und Nichterminale sind verschieden

Auf der linken Seite einer Produktion steht wenigstens ein Nichtterminal

- **Ableitung**
  - Ableitungsprozeß ist eine Relation "$\Rightarrow$" auf V^*
  - Für $u,v, \beta \in V^*$ und $\alpha \in V^* N V^*$ gilt
    - $u\alpha v \Rightarrow u\beta v$ genau dann, wenn $(\alpha \rightarrow \beta) \in P$
- **Die von einer Grammatik *erzeugte Sprache* ist definiert als:**
  - $L(G) = \{ w \mid w \in T^*, S \overset{*}{\Rightarrow} w\}$

w ist herleitbar aus dem Startsymbol

zwei Grammatiken heißen ***äquivalent***, wenn sie dieselbe Sprache erzeugen

Typen von Produktionen

- **Je nach Gestalt der in P *zugelassenen Produktionen* definiert Chomsky 4 Typen von Grammatiken**

Produktion	Typ	Eigenschaften	CH-Typ
(α -> β)	allgemein	$\alpha, \beta \in V^*$ beliebig	Typ-0
(α -> ε)	ε -Produktion	$\alpha \in V^*$, $r = \varepsilon$	
(α -> β)	beschränkt	$\alpha, \beta \in V^*$, $1 <= \lvert\alpha\rvert <= \lvert\beta\rvert$	Typ-1
(uAv -> uβv)	kontextsensitiv	$A \in N$, $u, v, \beta \in V^*$, $\beta \neq \varepsilon$	Typ-1
(A -> β)	kontextfrei	$A \in N$, $\beta \in V^*$	Typ-2
(A -> Bx)	linkslinear	$A, B \in N$, $x \in T$	Typ-3
(A -> xB)	rechtslinear		Typ-3
(A -> x)	terminierend	$A \in N$, $x \in T$	

Chomsky-Grammatiken

- **Die Chomsky-Grammatiken bilden eine Hierarchie**
  - d.h. die Menge der von Typ-n-Grammatiken erzeugten Sprachen umfaßt die Menge der Sprachen, die von Typ n+1 Grammatiken erzeugt werden
- **Typ-0-Grammatik**
  - Gestalt der Produktionen ist nicht eingeschränkt
  - Alle Sprachen, die überhaupt mit endlichen Regelsystemen erzeugt werden können
- **Typ-1 oder kontextsensitive Grammatik**
  - Produktionen sind beschränkt oder kontextsensitiv
- **Typ-2 oder *kontextfreie* Grammatik**
  - Produktionen sind kontextfrei (d.h. die linke Seite einer Produktion ist immer ein Nichtterminal)
- **Typ-3 oder reguläre Grammatik**
  - Produktionen sind terminierend, links- , rechtslinear

Kontextfreie Grammatik

- **Wichtigste Klasse zur formalen Beschreibung der Syntax von Programmiersprachen.**
- **Es ist möglich, Automaten zu bauen, die Wörter einer kontextfreien Sprache erkennen**
  - ***Wortproblem***
    - Kann für eine kf. Grammatik G und ein Wort w ∈ T* festgestellt werden, ob w von G erzeugt wird oder nicht.
  - ***Analyseproblem***
    - Gibt es einen Algorithmus, der zu einer kf. Grammatik G und einem Wort w ∈ T* die syntaktische Struktur von w bestimmt, oder aber feststellt, daß w nicht in L(G) liegt
    - ***Parser*** (Zerteilungsalgorithmus)
- **Formalismen zur Darstellung kontextfreier Grammatiken**
  - ***Syntaxdiagramme***
  - ***Extended Backus-Naur-Form***

Grammatik - Beispiel

- **kf. Grammatik, die korrekt geklammerte arithmetische Ausdrücke (bzgl. der Operationen *, +) erzeugt**
- G1 = ({E,T,F}, {(,), a, +, *}, P, E) mit
  - P = { E -> T,
    ❷ E -> E + T,
    ❸ T -> F,
    ❹ T -> T * F,
    ❺ F -> a,
    F -> (E) }
- **a * (a + a) Î L(G1)**
  - denn a*(a+a) läßt sich aus E folgendermaßen ableiten
  - dabei wurde immer das am weitesten links stehende Nichtterminal ersetzt (***Linksableitung***)

E	-> T	
	-> T * F	❹
	-> F * F	❸
	-> a * F	❺
	-> a * (E)	
	-> a * (E + T)	❷
	-> a * (T + T)	
	-> a * (F + T)	❸
	-> a * (a + T)	❺
	-> a * (a + F)	❸
	-> **a * (a + a)**	❺

Diskussion Beispiel - 1

- **Betrachtet man die Erzeugung von arithmetischen Ausdrücken genauer:**
  - Erzeugungsprozeß ist rückwärts betrachtet ein Prozeß der ***sukzessiven Zusammenfassung*** von Teilausdrücken
  - anschließend wird der gesamte Ausdruck auf das Startsymbol ***zurückgeführt***
- **Dabei wird folgendes beachtet**
  - Vorrang der Klammerstruktur
  - Vorrang der Multiplikation von der Addition
  - Zusammenfassen von links nach rechts bei gleichrangigen Operatoren
- **Beispiel zeigt**
  - daß die Syntax bereits auf grundlegende Eigenschaften der Semantik ***abgestimmt*** sein kann!

Diskussion Beispiel - 2

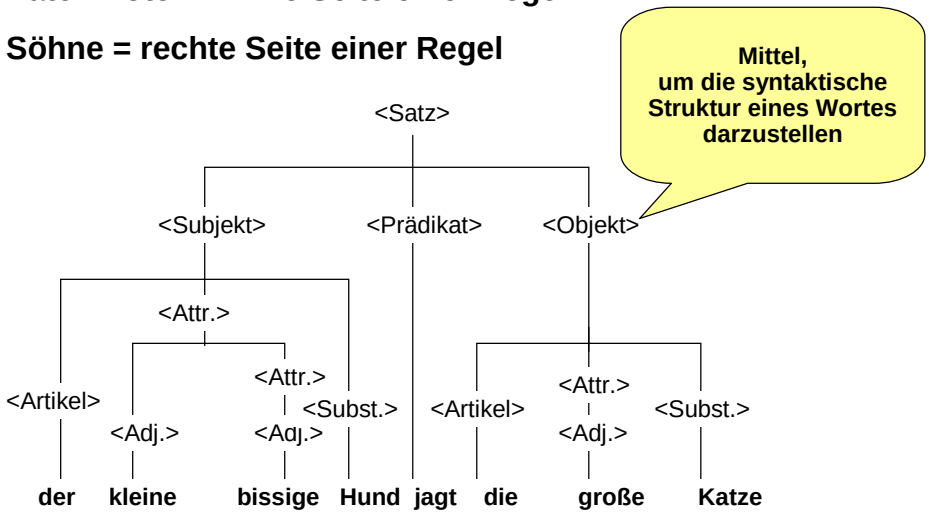
- **Folgende kf. Grammatik erzeugt dieselbe Sprache wie G1**
- G2 = ({E}, {(,), a, +, *}, P, E) mit
  - P = { ❶ E -> E + E,
    ❷ E -> E * E,
    ❸ E -> (E),
    ❹ E -> a }
- **Bemerkung**
  - bei G2 fehlt die bei G1 festgestellte Abstimmung der Syntax, d.h. des Erzeugungsprozesses auf die Regeln der Auswertung arithmetischer Ausdrücke
  - G1 kann als Modell für die Definition von Ausdrücken in höheren Programmiersprachen angesehen werden.

Grammatik - Beispiel

- **Sei G = (N, T, P, S) mit**
  - N = {A, B}
  - T = {a, b, c, d}
  - P = { (A -> aBbc),
    (B -> aBb),
    (aBb -> d) }
  - S = A
  - G ist keine kontextfreie Grammatik, da die dritte Produktionsregel auf der linken Seite mehr als nur das Nichtterminalsymbol enthält.
- **Ersetzt man in G P durch P', dann ist G' kontextfrei und L(G) = L(G')**
  - P' = { (A -> Bc),
    (B -> aBb),
    (B -> d) }

Ableitungsbaum -Strukturbaum

- **Vaterknoten = linke Seite einer Regel**
- **Söhne = rechte Seite einer Regel**



Syntaxdarstellung - EBNF - 1

- **EBNF**
  - **E**xtended **B**ackus-**N**aur-**F**orm
  - ***Meta-Sprache*** zur Beschreibung der Syntax formaler Sprachen
  - erstmals benutzt zur Definition der Sprache ***Algol-68***

  - ***Metasymbole*** von EBNF sind
    - = „definiert als“
    - (...|...) genau eine Alternative aus der Klammer muß stehen
    - [...] Inhalt der Klammer kann stehen oder nicht
    - { ... } Inhalt der Klammer kann n-fach stehen, n >= 0
    - . Ende der Produktion
    - Terminalsymbole werden in " " eingeschlossen

Syntaxdarstellung - EBNF - 2

- **Unsere einfache Grammatik für "Hund-Katze-Sätze" sieht in EBNF folgendermaßen aus:**

Satz	=	Subjekt Prädikat Objekt.
Subjekt	=	Artikel Attribut Substantiv.
Artikel	=	[("**der**" | "**die**" | "**das**")].
Attribut	=	[(Adjektiv | Adjektiv Attribut)].
Adjektiv	=	("**kleine**" | "**bissige**" | "**große**").
Substantiv	=	("**Hund**" | "**Katze**").
Prädikat	=	"**jagt**".
Objekt	=	Artikel Attribut Substantiv.

Syntaxdarstellung - EBNF - 2

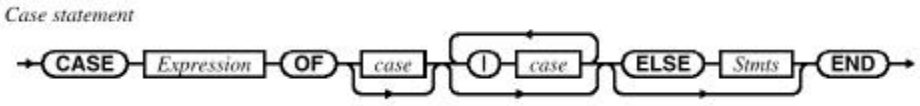
CaseStatement = „**CASE**“ Expression „**OF**“
[Case] { „**|**“ Case }
[„**ELSE**“ Stmts] „**END**“ .

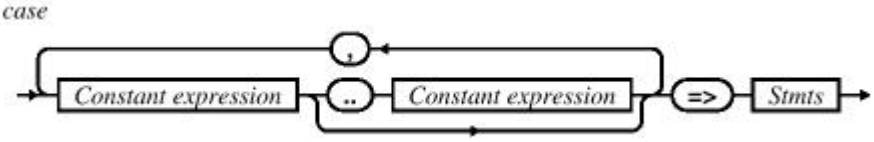
```
CASE  operator OF
  '+'  => resultat := a + b;
 |'-'  => resultat := a - b;
 |'*'  => resultat := a * b;
 |'/'  => resultat := a / b;
END;
```

```
CASE  operator OF
  '+'  => resultat := a + b;
 |'-'  => resultat := a - b;
 |'*'  => resultat := a * b;
 ELSE     resultat := a / b;
END;
```


Syntaxdiagramme - Beispiel

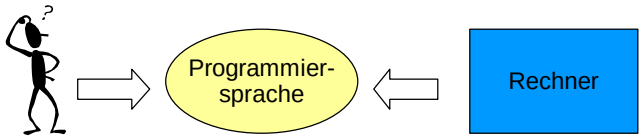
- **Syntaxdiagramme**
  - beschreiben Produktionen ***grafisch***
  - Nichterminalsymbole sind Rechtecke
  - Terminalsymbole sind Langrunde





Programmiersprachen

- **Die Programmiersprache bildet die *Schnittstelle* zwischen Mensch und Rechner**



Beide haben unterschiedliche Anforderungen

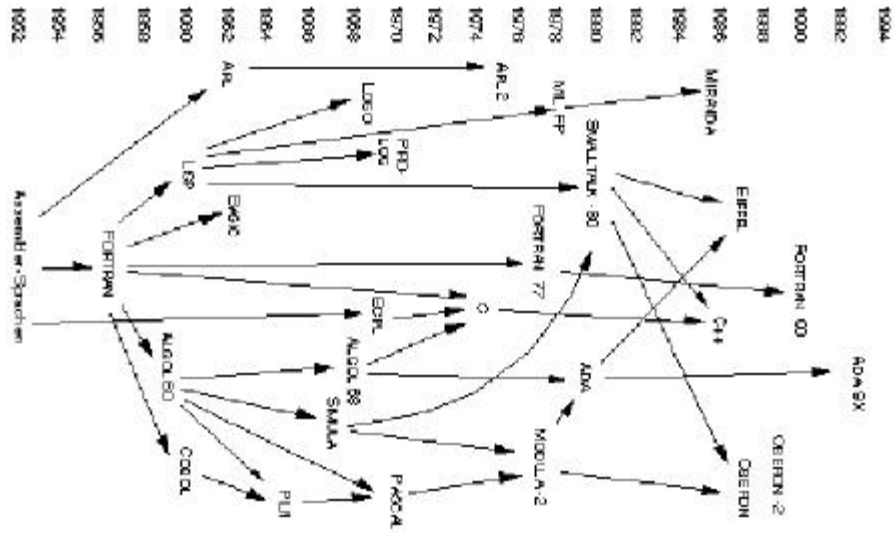
- **Mensch**
  - Erlernbarkeit
  - Lesbarkeit
  - Ausdrucksstärke

Problem-orientierte Sprachen

- **Rechner**
  - einfaches Übersetzen in Maschinensprache
  - effizienter Code soll generiert werden können

Maschinen-sprachen

Entwicklung der Programmiersprachen



Die Programmiersprache Modula-3

- **Warum verwenden wir Modula-3?**
- **Modula-3**
  - erlaubt Programmierkonzepte ***elegant*** zu formulieren
  - zeigt die zentralen Konzepte der ***imperativen*** und ***objektorientierten*** Programmierung
  - ist ***leicht*** erlernbar
  - ist im Sinne der ***software-technischen Qualität*** von Programmen entwickelt worden
  - das fördert einen "***guten***" Programmierstil
  - die erlernten Konzepte werden Sie später in anderen Sprachen in anderer Form wiederfinden
- **Wichtig ist (im Sinne der VL)**
  - nicht die Programmiersprache
  - sondern die ***Programmierkonzepte***

Exkurs

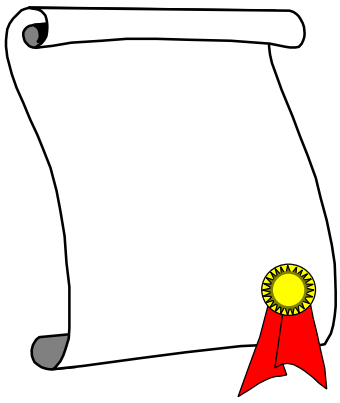
- **Programmiersprachen sind der *Werkstoff* des Informatikers**
- **Das, was wir erzeugen (Programme)**
  - ist ***immateriell***
  - wir bauen es aus dem Werkstoff "Programmiersprache"
- **Ein Informatiker**
  - sollte die ***Qualität*** und ***Eignung*** verschiedener Werkstoffe (Programmiersprachen) kennen
  - Welche Sprache ist wofür geeignet?
- **Analogie!**
  - Ein Bauingenieur verwendet andere Werkstoffe, wenn ein
    - Hochhaus (Stahl, Beton etc.)
    - oder ein Einfamilienhaus (Ziegelsteine, Holz, Beton, etc.)
  - gebaut werden soll

Was haben wir gelernt!

- Wie wissen was Programmiersprachen sind
- Wir kennen den Begriff "Formale Sprache"
- Wir haben gesehen, daß eine Grammatik eine Sprache erzeugt
- Wir kennen EBNF und Syntaxdiagramme
- Wir sehen Programmiersprachen als Werkstoff des Informatikers

Glossar- 2

- Programmiersprache
- Syntax
- Semantik
- Alphabet
- Formale Sprache
- Grammatik
  - Chomsky-Grammatik
  - Kontextfreie Grammatik
- EBNF
- Syntaxdiagramm



Einführung in Modula-3
Funktionale Programme

- **Was ist Modula-3**
- **Aufbau von Modula-3-Programmen**
- **Funktionale Programme in Modula-3**
- **Funktionen, Parameter**
- **Einfache Datentypen**
- **Rekursion**

Modula-3

As Sam Harbison writes in his book Modula-3,

Modula-3 is a member of the ***Pascal*** family of languages. Designed in the late 1980s at Digital Equipment Corporation and Olivetti, Modula-3 corrects many of the ***deficiencies*** of Pascal and Modula-2 for ***practical software engineering***. In particular, Modula-3 keeps the ***simplicity*** of type safety of the earlier languages, while providing ***new facilities*** for exception handling, concurrency, object-oriented programming, and automatic garbage collection. Modula-3 is both a practical implementation language for large software projects and an ***excellent teaching language***.

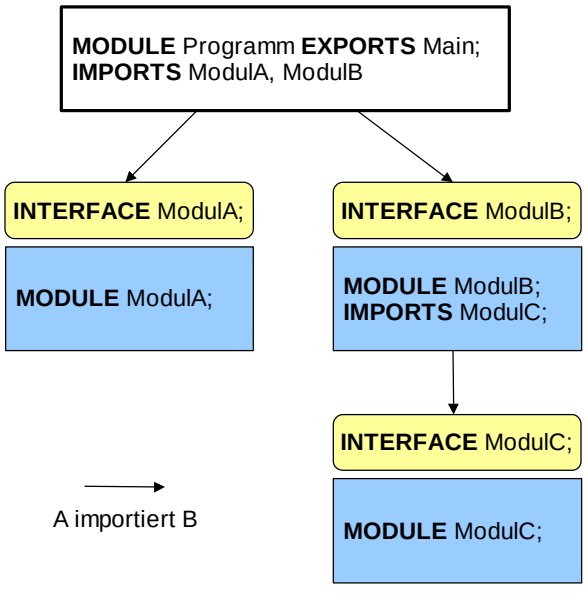
- **Modula = Modular Language**
  - Modul ist ein wichtiges Sprach- und Programmierkonzept
- **Modula-3 ist**
  - eine ***imperative*** (prozedurale) Programmiersprache
  - eine ***strukturierte*** Programmiersprache

Aufbau von Modula-3 Programmen

- **Modula-3 Programm**
  - besteht wenigstens aus einem ***Modul***, dem ***Hauptmodul***
- **Was ist ein Modul?**
  - ein Modul ist ein Programmteil, das ***sinnvoll*** zusammengehörende Elemente enthält
  - ein Modul besteht (bis auf das Hauptmodul) aus
    - ***Schnittstelle*** (interface)
      - definiert, was ein Modul exportiert
    - ***Implementierung***
      - enthält die Implementierung der exportierten Elemente
      - versteckt die Implementierung
- **Hauptmodul**
  - exportiert die vordefinierte ***leere*** Schnittstelle `Main`

Modul-Hierarchie

- ein Modul kann Elemente anderer Module ***benutzen***
  - ein Modul ***importiert*** dazu andere Module
  - von einem importierten Modul ist nur die ***Schnittstelle*** sichtbar

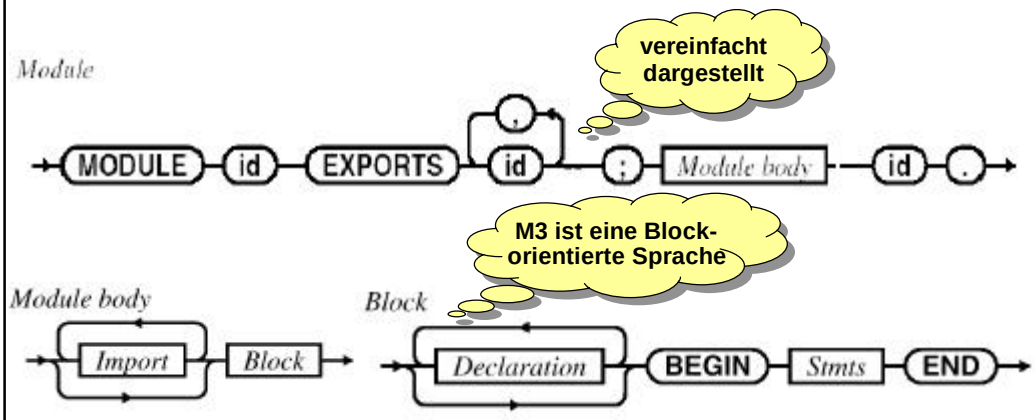


Vorteile modularer Programme

- **Vorteile modularer Programme**
  - Module können von ***unterschiedlichen*** Personen entwickelt und gepflegt werden
    - Software-Entwicklung ist Team-Arbeit
  - Module können einzeln ***getestet*** werden
    - Test großer Programme ist extrem aufwendig
  - Module können geordnet zum Gesamtsystem ***integriert*** werden
  - eine Implementierung eines Moduls kann leichter durch eine neue Implementierung ***ersetzt*** werden
    - z.B. durch eine effizientere Implementierung
  - Module können in verschiedenen Programmen ***wiederverwendet*** werden (Modul-Bibliothek)
    - dies senkt die Kosten für die Entwicklung

Aufbau eines Moduls

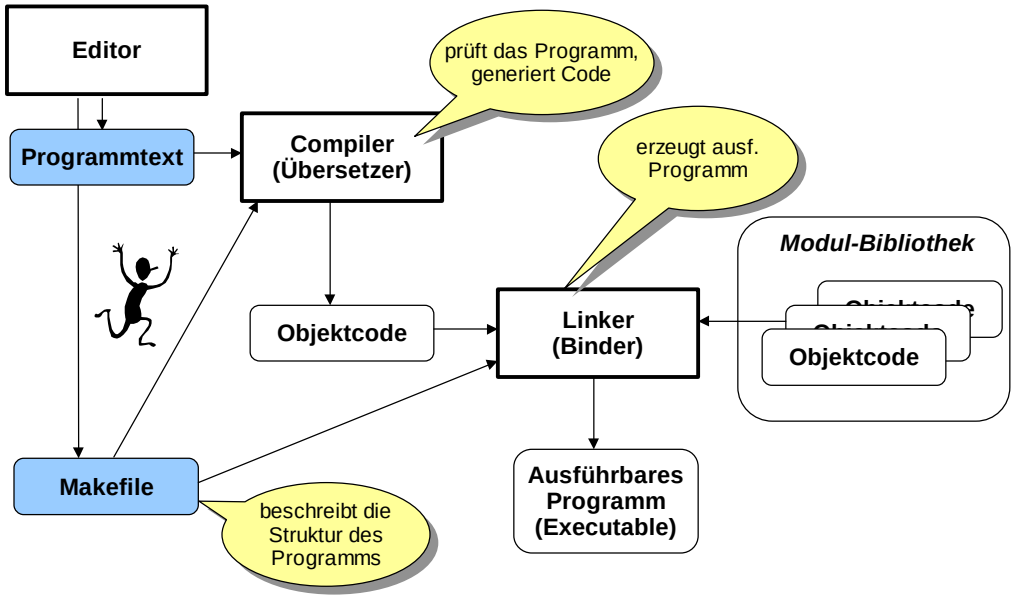
- **Ein Modul**
  - ***exportiert*** Elemente und ***importiert*** andere Module
  - enthält einen ***Block***
- **Ein Block**
  - enthält ***Deklarationen*** und ***Anweisungen***



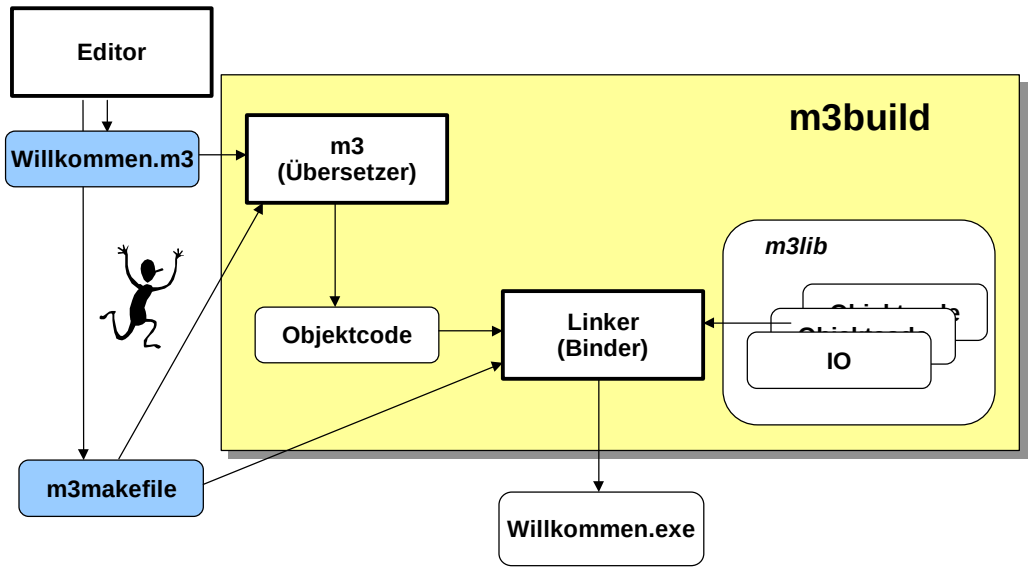
Das erste M3-Programm

```
MODULE Willkommen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss
   Autor           : Horst Lichter, RWTH Aachen
   Umgebung        : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt        : 16.08.98
   Letzte Aenderung: 20.08.98
*)

IMPORT SIO;
BEGIN
  SIO.PutText("Willkommen zum Studium in Aachen.");
END Willkommen.
```


Vom Programmtext zum ausf. Programm



Edit-Compile-Link-Run bei Modula-3



Funktionale Programmierung

- **Konzept**
  - Formulierung von ***Funktionsdefinitionen***
  - Ausführung eines funktionalen Programms besteht in der ***Berechnung*** eines ***Ausdrucks*** mit Hilfe dieser Funktionen
  - Berechnung liefert ein ***Ergebnis*** zurück
- **Was benötigt man dazu**
  - Daten / ***Datentypen***
  - ***elementare*** Funktionen
  - Möglichkeit, Funktionen zu ***definieren***
  - Ausdrucksmittel zur ***Vernetzung*** von Funktionen
- **Anmerkung:**
  - es gibt rein funktionale Programmiersprachen MIRANDA
  - viele sog. Hochsprachen erlauben eine gewisse Art der funktionalen Programmierung (Modula-3)

Funktionen in Modula-3

- **Eine Funktion**
  - hat einen ***Namen***
  - hat keinen oder mehrere ***Eingabeparameter*** (Argumentbereich)
  - hat einen ***Ergebnistyp*** (Ergebnisbereich)
  - hat eine ***Berechnungsvorschrift***
  - ist frei von ***Seiteneffekten***

Signatur

```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;
```

Berechnungsvorschrift **Name** **formale Eingabeparameter** **Ergebnistyp**

Aufruf einer Funktion

```
MODULE QuadratM EXPORTS Main;
(* Dieses Programm berechnet das Quadrat einer Zahl
   Autor           : Horst Lichter
   Umgebung        : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt        : 20.08.98    Letzte Aenderung: 20.08.98
*)

IMPORT SIO;

PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

BEGIN
  SIO.PutReal (Quadrat (SIO.GetReal()));
END QuadratM.
```

Deklaration und ***Definition*** der Funktion

Aufruf der Funktion mit einem ***aktuellen*** Parameter;

Das ***Ergebnis*** der Funktion wird an die Prozedur PutReal ***übergeben***

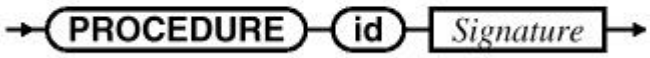
Formale & aktuelle Parameter

- **Parameter**
  - erlauben es, Funktionen mit ***Eingabewerten*** zu versorgen
  - haben eine ***Typ***
  - dadurch werden Funktionen ***flexible einsetzbar***
- **Formale Parameter**
  - werden in der ***Definition*** einer Funktion angegeben
  - dienen als ***Stellvertreter*** im ***Rumpf*** der Funktion für die zur Laufzeit des Programms übergebenen aktuellen Parameter
- **Aktuelle Parameter**
  - beim ***Aufruf*** einer Funktion müssen ihre formalen Parameter gemäß ihrer Definition an aktuelle Parameter ***gebunden*** werden
  - diese werden dann im Rumpf ***verwendet***.

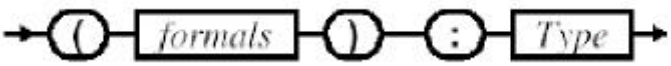
Syntax von M3-Funktionen - 1

Declaration

Procedure heading

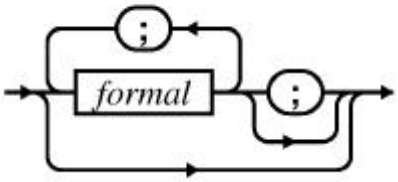


Signature

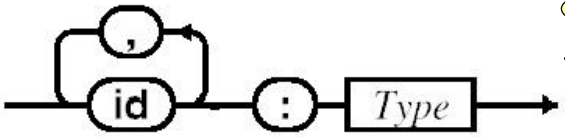


Syntax von M3-Funktionen - 2

formals



formal



Vereinfachte Darstellung !

Formale & aktuelle Parameter

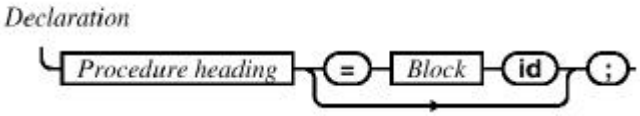
```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

BEGIN
  SIO.PutReal (Quadrat (2.5));
END QuadratM.
```

Binden der aktuellen Werte (Parameter) an die formalen Parameter (Stellvertreter)

Deklarationen (vorläufig)

- **Wir wissen bereits:**
  - ***Blöcke*** können ***Deklarationen*** enthalten
- **und bestehen aus *Anweisungen* (Statements)**

- **Deklarationen:**
  - Idee: Namen (***Bezeichner***) werden vereinbart, damit diese später benutzt werden können
  - Die in einem Block deklarierten Bezeichner sind nur innerhalb des Blockes ***gültig***



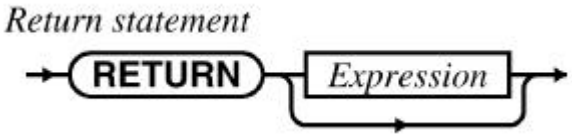
Anweisungen (vorläufig)

- **Anweisungen:**
  - sind in Blöcken enthalten
  - atomare Bausteine für Modula-3 Programme

  - Die Folge der Anweisungen eines Blocks wird bei Ausführung in der ***Reihenfolge der Aufschreibung*** abgearbeitet
- **Beispiele für Anweisungen:**
  - RETURN-Anweisung
  - CALL-Anweisung (Funktionsaufruf)
  - Anweisungen werden durch ";" von einander getrennt

Ausdrücke

- **Ausdrücke**
  - bezeichnen Elemente, die ***ausgewertet*** werden können
  - liefern einen Wert (Ergebnis)
- **Beispiele**
  - arithmetische Ausdrücke
    - x * x
  - logische Ausdrücke
    - x AND y
- **Viele Anweisungen erlauben, daß Ausdrücke verwendet werden können**



Datentypen

- **Typbegriff**
  - im Zusammenhang mit Programmiersprachen hat der Begriff ***Typ*** oder auch ***Datentyp*** eine zentrale Bedeutung
- **Definition (Informatik-Duden)**
  - Unter einem ***Datentyp*** versteht man die ***Zusammenfassung*** von ***Wertebereichen*** und ***Operationen*** zu einer ***Einheit***
- **Im Zusammenhang mit Funktionen bedeutet das:**
  - ***Argumentbereich*** und ***Ergebnisbereich*** bilden die Wertebereiche, die zu den Typen der Ein- und Ausgabewerte gehören. Dazu kommen noch die ***Operationen***, die darauf zulässig sind.
- **Man unterscheidet grob:**
  - ***einfache*** Datentypen
  - ***zusammengesetzte*** Datentypen

Einfache Datentypen in Modula-3

- **In Modula-3**
  - Modula-3 besitzt, wie viele andere Sprachen, eine Reihe ***einfacher*** Datentypen, die auch als ***elementare*** Datentypen bezeichnet werden.
- **Gruppierung der einfachen Datentypen in Modula-3**
  - **Ganze Zahlen**
    - darunter fallen die Typen INTEGER und CARDINAL
  - **Zeichen**
    - Werte eines bestimmten Zeichenvorrates; z.B. definiert der ASCII-Zeichenvorrat 128 Zeichen.
  - **Texte**
    - sind eine Folge von Zeichen
  - **Wahrheitswerte**
    - Werte sind {wahr, falsch} bzw. {TRUE, FALSE}
  - **Gleitkommazahlen**
    - reelle Zahlen, REAL, LONGREAL, EXTENDET

Ganze Zahlen

- **Typen: INTEGER und CARDINAL**
  - **Integer-Zahlen** sind ***ganzzahlige*** Werte innerhalb der Unter- und Obergrenze des jeweiligen Rechners
  - **Cardinal-Zahlen** sind sind ***nicht-negative*** ganzzahlige Werte, d.h. zwischen 0 und der Obergrenze des jeweiligen Rechners
- **Wertebereich**
  - auf einen 32-Bit-Rechner:
    - INTEGER [-2147483648 .. 2147483647] oder [-2^{31} .. $2^{31}-1$]
    - 1 Bit für das Vorzeichen, 31Bit für die Zahlendarstellung
  - Zahlen sind geordnet (***Ordinaltyp***)
- **Operationen**
  - arithmetische Operationen (+, -, *, DIV, MOD)
  - Vergleichsoperationen (=, #, <, >, <=, >=)
  - vordefinierte Funktionen (FIRST(type), LAST(type), INC(z), DEC(z), ABS(z))

Beispiel - Ganze Zahlen

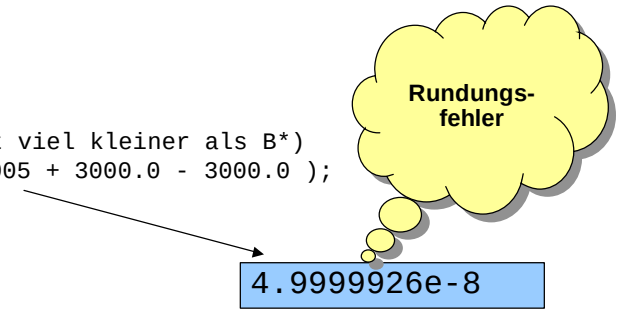
```
MODULE Zahlen EXPORTS Main;
(* Dieses Programm zeigt dem Umgang mit ganzen Zahlen *)

IMPORT SIO;

PROCEDURE Modulo (x,y: INTEGER): INTEGER =
(* MOD ist def. : x MOD y = x - y*(x DIV y) *)
BEGIN
  RETURN ( x - y*(x DIV y) );
END Modulo;

BEGIN
  (* Ausgabe der Unter- und Obergrenze von INTEGER *)
  SIO.PutInt (FIRST(INTEGER)); SIO.Nl();
  SIO.PutInt (LAST(INTEGER));  SIO.Nl();

  SIO.PutInt (20 DIV 6); SIO.Nl();     (* = 3 *)
  SIO.PutInt (20 MOD 6); SIO.Nl();     (* = 2 *)
  SIO.PutInt (Modulo(20,6));           (* = 2 *)
END Zahlen.
```


Gleitkommazahlen

- **Typen: REAL, LONGREAL, EXTENDED**
  - repräsentieren die rationalen und reellen Zahlen
- **Wertebereich**
  - ist ***beschränkt*** (im Unterschied zur Mathematik)
  - Genauigkeit der Darstellung ist ***beschränkt***
    - Bspl: wir haben 4 Stellen zur Verfügung
      größte Zahl: 9999
      kleinste Zahl: 0.001
    - Probleme: die Zahlen 0.0005 und 0.000089 sind nicht zu unterscheiden (es wird gerundet)
  - Rechnen mit Gleitkommazahlen ist immer **fehlerbehaftet**!
    - "Numerik" liefert Techniken und Algorithmen, um genau zu rechnen!

Beispiel für Gleitkommazahlen

```
MODULE Gleitkomma EXPORTS Main;
(* Dieses Programm zeigt Rundungsprobleme bei Gleitkommazahlen
   Autor    : Horst Lichter
   Umgebung : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt : 25.08.98   Letzte Aenderung: 26.08.98
*)

IMPORT SIO;


BEGIN
  (* A + B - B mit A ist viel kleiner als B*)
  SIO.PutReal ( 0.00000005 + 3000.0 - 3000.0 );

END Gleitkomma.
```



Zeichen

- **Typ CHAR**
  - CHAR (character) bezeichnet eine ***endliche***, ***geordnete Menge*** von Zeichen
  - CHAR ist ein ***Ordinaltyp***
- **Wertebereich**
  - viele Rechner benutzen den ASCII-Zeichen
  - Zeichenliterale: 'A' 'z' '1'
  - Spezialzeichen:

\n	Zeilenvorschub	\t	Tabulator	\\	Backslash
\'	Apostroph	\f	Seitenumbruch		
\r	Wagenrücklauf	\"	Anführungszeichen		

- **Operationen**
  - Vergleichsoperationen (=, #, <, >, <=, >=)
  - Vordefinierte Funktionen FIRST(CHAR), LAST(CHAR), INC(z), DEC(z), ORD(z), VAL(i)

Beispiel - Zeichen

```
MODULE KleinGross EXPORTS Main;
(* Dieses Programm wandelt einen Klein- in einen Grossbuchstaben um
   Autor    : Horst Lichter
   Umgebung : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt : 25.08.98   Letzte Aenderung: 26.08.98
*)

IMPORT SIO;

PROCEDURE Offset (): INTEGER =
BEGIN
  RETURN (ORD('A') - ORD('a'));
END Offset;


BEGIN
  (* Kleinbuchstaben einlesen und umwandeln *)
  SIO.PutChar ( VAL(ORD(SIO.GetChar()) + Offset(), CHAR) );

END KleinGross.
```


Texte

- **Typ: TEXT**
  - repräsentiert eine ***beliebig lange Folge*** von Zeichen (kann auch leer sein)
  - in vielen Sprachen nicht explizit vorhanden
- **Wertebereich**
  - Textliterale werden in " " notiert
  - z.B. "Das ist ein Text mit Zeilenvorschub\n"
"\"Dieser Text beginnt und endet mit einem Hochkomma\""
- **Operationen**
  - Konkatenation : &
    - Bspl: "Heute " & "ist " & "Freitag." ⇨ "Heute ist Freitag."
  - Schnittstelle des Moduls "Text"
    - Equal, Length, Empty, FindChar ….

Wahrheitswerte

- **Typ: BOOLEAN**
  - repräsentiert die beiden vordefinierten Wahrheitswerte
- **Wertebereich**
  - wahr; TRUE
  - falsch, FALSE
- **Operationen**
  - Komplement (NOT), Oder (OR), Und (AND)

p	q	NOT q	p OR q	p AND q
T	T	F	T	T
T	F	T	T	F
F	T	F	T	F
F	F	T	F	F

Beispiel für Wahrheitswerte

```
MODULE BooleanM EXPORTS Main;
(* Dieses Programm berechnet die Wahrheitstabelle NOT, OR, AND *)
IMPORT SIO, Fmt;

PROCEDURE NotOrAnd ( a, b : BOOLEAN) : TEXT =
BEGIN
  RETURN (Fmt.Bool(a) & " " & Fmt.Bool(b) & " : " &
          Fmt.Bool( NOT(b))       & " " &
          Fmt.Bool( a OR b)       & " " &
          Fmt.Bool(a AND b) );
END NotOrAnd;

BEGIN
  (* Ausgabe der Wertetabelle *)
  SIO.PutLine(NotOrAnd(FALSE, FALSE));
  SIO.PutText(NotOrAnd(TRUE, FALSE));
  SIO.PutLine(NotOrAnd(FALSE, TRUE));
  SIO.PutLine(NotOrAnd(TRUE, TRUE));

END BooleanM.
```

produzierte Ausgabe

```
FALSE FALSE : TRUE  FALSE FALSE
TRUE  FALSE : TRUE  TRUE  FALSE
FALSE TRUE  : FALSE TRUE  FALSE
TRUE  TRUE  : FALSE TRUE  TRUE
```

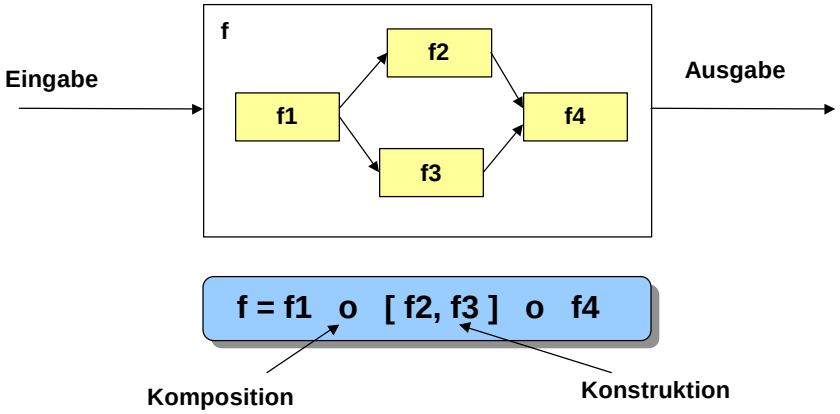
Vernetzung von Funktionen - 1

- **Um komplexe Ausdrücke zu berechnen,**
  - werden Funktionen vernetzt.
- **Funktionalformen (oder Funktionale)**
  - beschreiben "Vernetzungsmuster"
- **Beispiele für Funktionalformen**
  - **Komposition**
    - f o g : x = g : (f : x)
  - **Konstruktion**
    - [f, g, h, …] : x = (f : x, g : x, h : x, …)
  - **Bedingung**
    - if t then f else g : x =

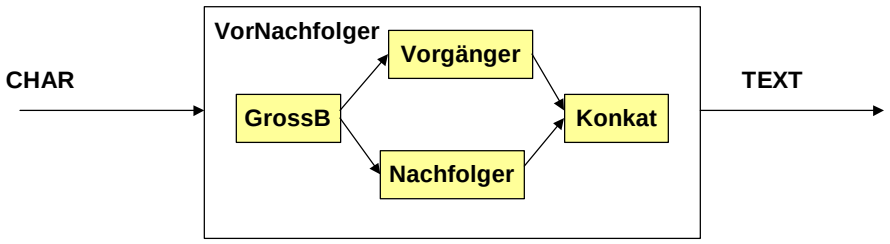
$$\begin{cases} f : x & \text{, falls } t : x = \text{true} \\ g : x & \text{, falls } t : x = \text{false} \\ ? & \text{, sonst} \end{cases}$$

Vernetzung von Funktionen - 2

- **Schematisches Beispiel:**



Vernetzung von Funktionen - 3



VorNachfolger = GrossB o [Vorgänger, Nachfolger] o Konkat

Beispiel:

Eingabe: 'h'

Ausgabe "GI"

Beispiel - Vernetzung von Funktionen

```
MODULE VorNachfolger EXPORTS Main;
IMPORT SIO;

PROCEDURE GrossB(c : CHAR) : CHAR =
...
PROCEDURE Vorgaenger (c : CHAR) : CHAR =
...
PROCEDURE Nachfolger (c : CHAR) : CHAR =
...
PROCEDURE AddOrd (c, d : CHAR) : CARDINAL =
. . .
PROCEDURE Konkat (c, d : CHAR) : TEXT =
...

BEGIN
  (* VorNachfolger = GrossB o   [ Vorgänger, Nachfolger ]   o   Konkat *)
  SIO.PutText ( Konkat ( Vorgaenger( GrossB( SIO.GetChar() )) ,
                         Nachfolger( GrossB( SIO.GetChar() ))
                       )
              );

END VorNachfolger.
```

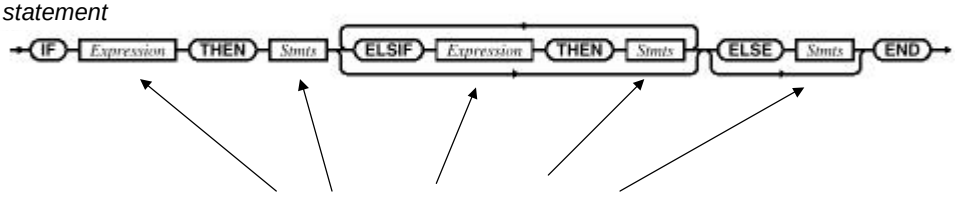
Da **keine Variablen** (Zwischenspeicher) verwendet werden, muß der Wert **zweimal** eingelesen werden !

Funktion GrossB muß **zweimal** ausgeführt werden, da Modula-3 die **Konstruktion** von Funktionen nicht unterstützt.

Bedingung in funktionalen Programmen

- **Idee:**
  - In Abhängigkeit von ***Bedingungen*** soll eine ***alternative*** Programmkomponente ausgeführt werden
  - Bedingungen müssen einen Wert vom Typ ***BOOLEAN*** liefern
  - Modula-3 bietet dazu u.a. die ***IF-Anweisung*** an

If statement



Bei funktionalen Programmen stehen hier ***ausschließlich*** Funktionsaufrufe!

Bedingung - IF-THEN-ELSE

- **Typische Formen von Bedingungen:**

```
IF bf THEN
  f1;
ELSE
  f2;
END;
```

Alternative "Entweder-Oder"

```
IF bf THEN
  f1;
END;
```

bedingte Anweisung

```
IF    bf1 THEN
  f1;
ELSIF bf2 THEN
  f2;
ELSIF bf3 THEN
  f3;
…
ELSE
  fn;
END;
```

Die ***Bedingungen***(bfi) werden der Reihe nach ausgewertet, bis eine ***WAHR*** ist. Dann wird die entsprechende Programmkomponente ausgeführt.

Ist ***keine*** Bedingung wahr, dann wird der ***ELSE-Zweig*** ausgeführt

Beispiel - IF-THEN-ELSE - 1

- **Aufgabe:**
  - Eingabe ist eine ganze Zahl
  - Stelle fest, ob diese 1-, 2-, 3-, mindestens 4-stellig oder negativ ist!

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF IstEinstellig(i)     THEN RETURN ("einstellig") END;
  IF IstZweistellig(i)    THEN RETURN ("zweistellig") END;
  IF IstDreistellig(i)    THEN RETURN ("dreistellig") END;

  IF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE RETURN ("negativ")
  END;
END ErmittleStelligkeit;

BEGIN
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```


Beispiel - IF-THEN-ELSE - 2

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF    IstEinstellig(i)     THEN RETURN ("einstellig")
  ELSIF IstZweistellig(i)    THEN RETURN ("zweistellig")
  ELSIF IstDreistellig(i)    THEN RETURN ("dreistellig")
  ELSIF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE                            RETURN ("negativ")
  END;
END ErmittleStelligkeit;



BEGIN
  (* Aufruf der Hauptfunktion *)
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```

ELSIF-Konstruktionen machen Bedinungen semantisch klarer

Beispiel - IF-THEN-ELSE - 3

```
MODULE Stelligkeit EXPORTS Main;
(* Dieses Programm berechnet die Stelligkeit von Zahlen *)
IMPORT IO;

PROCEDURE IstEinstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=0) AND (i<10) );
END IstEinstellig;

PROCEDURE IstZweistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=10) AND (i<100) );
END IstZweistellig;

PROCEDURE IstDreistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=100) AND (i<1000) );
END IstDreistellig;

PROCEDURE IstMinVierstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=1000) );
END IstMinVierstellig;
```

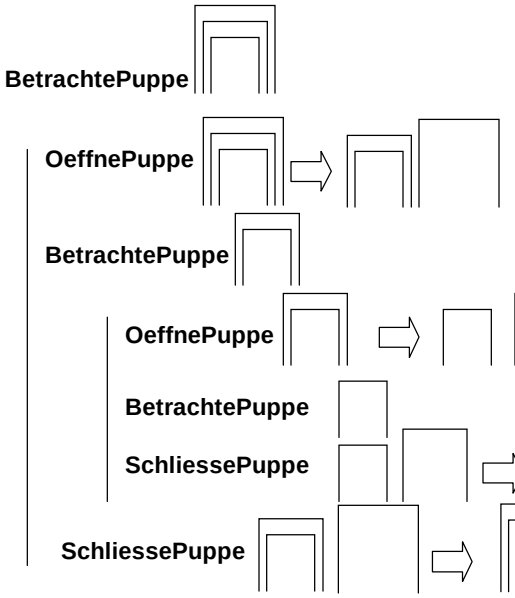

Rekursion

- **Idee:**
  - allgemein bezeichnet man mit ***Rekursion*** die Definition eines Problems, einer Funktion oder ganz allgemein eines Verfahrens "***durch sich selbst***"
- **Rekursive Funktion**
  - darunter verstehen wir Funktionen, die sich ***selbst wieder aufrufen***
- **Beispiel: Matrioschka-Puppen**

```
PROCEDURE BetrachtePuppe (puppe);
(* Pseudocode zum Betrachten einer Matrioschka *
BEGIN (* BetrachtePuppe *)
  IF NOT Massiv (puppe) THEN
    OeffnePuppe (puppe); (* gibt InhaltDerPuppe frei *)
    BetrachtePuppe (inhaltDerPuppe); (* Rekursion *)
    SchliessePuppe (puppe);
  END (* IF *);
END BetrachtePuppe;
```

Rekursions-abbruch

Rekursion - Beispiel



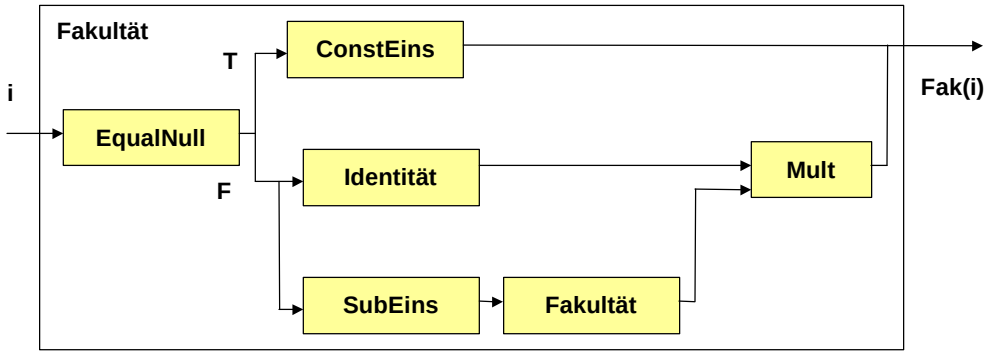
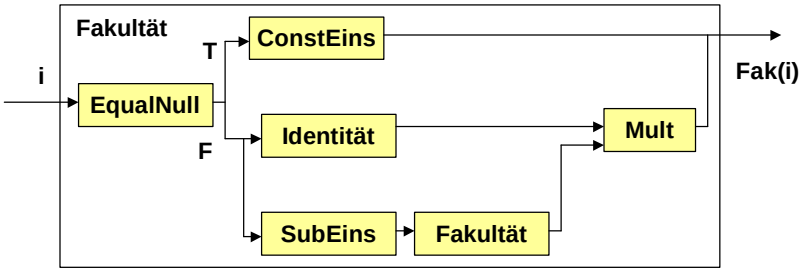

```
PROCEDURE BetrachtePuppe (puppe);
BEGIN
  IF NOT Massiv (puppe) THEN
   BetrachtePuppe(OeffnePuppe (puppe));
   SchliessePuppe (puppe);
  END (* IF *);
END BetrachtePuppe;
```

Jeder rekursive Aufruf erzeugt eine neue **Inkarnation** der Funktion.

Es wird solange rekursiv aufgerufen, bis der **Rekursions-abbruch** erreicht wird.

Anschließend werden die erzeugten Inkarnationen in **umgekehrter** Reihenfolge abgearbeitet

Anmerkungen zur Rekursion

- **Ziel:**
  - es darf keine ***unkontrollierte*** (unendliche) Rekursion entstehen
  - dies führt immer zu einem ***Laufzeitfehler***
- **Konsequenz**
  - Jeder rekursive Funktionsaufruf gehört in eine ***bedingte Anweisung***
    - Rekursionsabbruch: in definierten Fällen wird der rekursive Aufruf nicht ausgeführt
  - Muster:
    - ```
      IF … THEN
        rekursiver Aufruf
      ELSE
        Rekusionsabbruch
      ```
- **Bemerkung**
  - Rekursion führt zu ***prägnaten, knappen*** und ***eleganten*** Algorithmen
      ```

Rekursion - Beispiel Fakultät - 1

- **Fakultät is definiert:**
  - fak : n 1 , falls n = 0
    n * fak(n-1) n ∈ N
- **Funktionnetz für Fakultät**



Rekursion - Beispiel Fakultät - 2




```
PROCEDURE Fakultaet (i : INTEGER) : INTEGER =
BEGIN
  IF EqualNull(i)
  THEN RETURN(ConstEins(i))
  ELSE RETURN (Mult(Identitaet(i), Fakultaet(SubEins(i))));
  END;
END Fakultaet ;
...
  SIO.PutInt(Fakultaet(SIO.GetInt()));
```

Rekursion - Beispiel Fakultät - 3

```
PROCEDURE EqualNull ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( i = 0 );
END EqualNull;

PROCEDURE ConstEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( 1 );
END ConstEins;

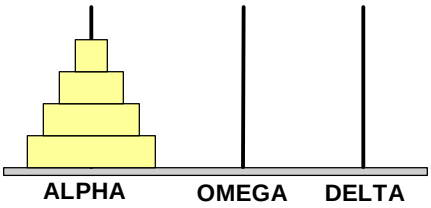
PROCEDURE Mult ( i, j : INTEGER) : INTEGER =
BEGIN
  RETURN ( i * j );
END Mult;

PROCEDURE Identitaet ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i );
END Identitaet;

PROCEDURE SubEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i - 1 );
END SubEins ;
```

Diese Funktionen sind nur im Sinne der **idealen funktionalen** Programmierung notwendig.

Sie können in der Definition der Funktion "Fakultaet" durch entsprechende **Ausdrücke** ersetzt werden!

Rekursion - Beispiel Fakultät - 4

```
MODULE FakultaetM1 EXPORTS Main;
(* Dieses Programm berechnet die Fakultaetsfunktion *)

IMPORT SIO;

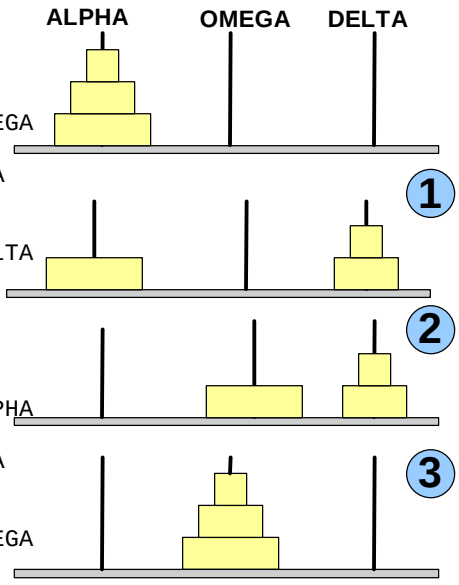
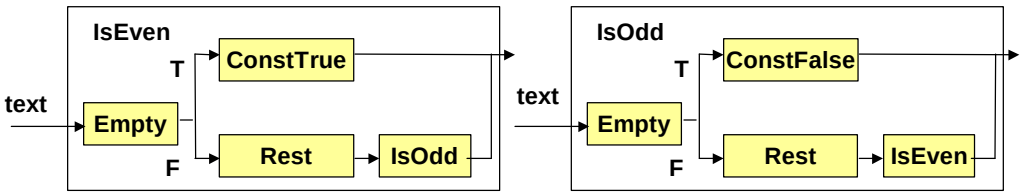

PROCEDURE Fakultaet ( i : INTEGER) : INTEGER =
BEGIN
  IF i = 0
  THEN RETURN 1
  ELSE RETURN (i * Fakultaet(i-1));
  END;
END Fakultaet ;


BEGIN
  SIO.PutInt(Fakultaet(SIO.GetInt()));
END FakultaetM1.
```

Rekursion - Türme von Hanoi - 1

- **Aufgabe:**
  - bewege die Scheiben des Turms von ALPHA nach OMEGA
  - es darf immer ***nur eine Scheibe*** bewegt werden
  - niemals darf eine Scheibe auf eine kleinere bewegt werden



- **Lösungsstrategie**
  - allgemeine Lösung für einen Turm der Höhe h von ALPHA nach OMEGA
    - h = 0 gar nichts machen
    - h > 0 1. Turm der Höhe h-1 von ALPHA nach DELTA über OMEGA
      2. Scheibe von ALPHA nach OMEGA legen
      3. Turm der Höhe h-1 von DELTA nach OMEGA über ALPHA

Rekursion - Türme von Hanoi - 2

```
MODULE Hanoi EXPORTS Main;
(* Ausgabe der Zugfolge fuer Tuerme von Hanoi *)

IMPORT SIO;

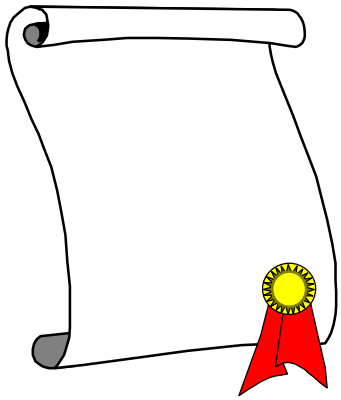
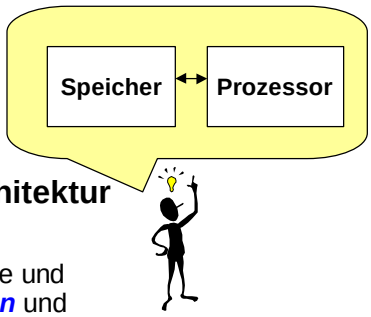
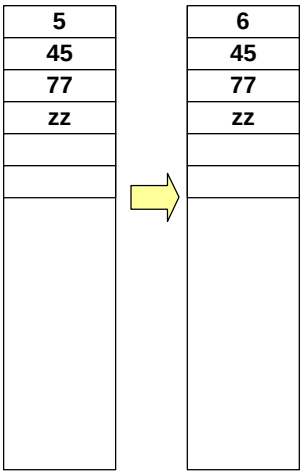
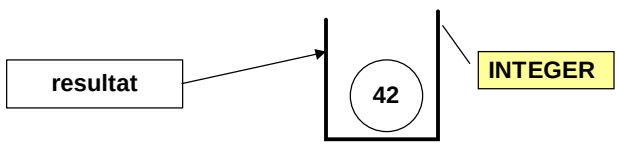
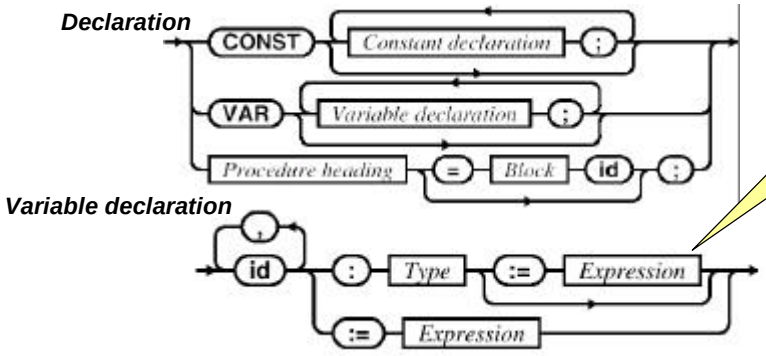
PROCEDURE DruckeZug (hoehe: CARDINAL; von, nach : TEXT)  =
BEGIN
  SIO.PutText ("Scheibe "); SIO.PutInt (hoehe);
  SIO.PutText (" von " & von & " nach " & nach); SIO.Nl();
END DruckeZug ;

PROCEDURE BewegeTurm ( hoehe : CARDINAL; von, nach, ueber: TEXT)  =
BEGIN
  IF hoehe > 0 THEN
    BewegeTurm (hoehe-1, von, ueber, nach);
    DruckeZug  (hoehe, von, nach);
    BewegeTurm (hoehe-1, ueber, nach, von);
  END;
END BewegeTurm;

BEGIN
  BewegeTurm(SIO.GetInt(), "ALPHA", "OMEGA", "DELTA" );
END Hanoi.
```

Rekursion - Türme von Hanoi - 3

```
3 ALPHA OMEGA DELTA
  2 ALPHA DELTA OMEGA
    1 ALPHA OMEGA DELTA
      0 ALPHA DELTA OMEGA
    ❷ Scheibe 1 von ALPHA nach OMEGA
    ❸ 0 DELTA OMEGA ALPHA
  ❷ Scheibe 2 von ALPHA nach DELTA
  ❸ 1 OMEGA DELTA ALPHA
      0 OMEGA ALPHA DELTA
    ❷ Scheibe 1 von OMEGA nach DELTA
    ❸ 0 ALPHA DELTA OMEGA
❷ Scheibe 3 von ALPHA nach OMEGA
❸ 2 DELTA OMEGA ALPHA
    1 DELTA ALPHA OMEGA
      0 DELTA OMEGA ALPHA
    ❷ Scheibe 1 von DELTA nach ALPHA
    ❸ 0 OMEGA ALPHA DELTA
  ❷ Scheibe 2 von DELTA nach OMEGA
  ❸ 1 ALPHA OMEGA DELTA
      0 ALPHA DELTA OMEGA
    ❷ Scheibe 1 von ALPHA nach OMEGA
    ❸ 0 DELTA OMEGA ALPHA
```



Fibonacci-Funktion (rekursiv)

- **Wachstum einer Kaninchen-Population**
  - Wieviele Kaninchen-Pärchen gibt es nach n Jahren
    - Jahr 1 : 1 Pärchen
    - Jedes Pärchen hat ab dem zweiten Jahr je ein Pärchen Nachwuchs
  - Jahr 1 2 3 4 5 6 7 8 9
  - Anzahl 1 1 2 3 5 8 13 21 34

```
PROCEDURE Fibonacci (arg: INTEGER): INTEGER =
BEGIN
  IF arg <= 2 THEN
    RETURN 1
  ELSE
    RETURN (Fibonacci (arg -1) + Fibonacci (arg - 2))
  END;
END Fibonacci ;
```

Indirekte Rekursion

- **Definition:**
  - ***Indirekte*** Rekursion kann in einem System von Funktionen definiert werden, die Seite an Seite vereinbart werden und sich ***gegenseitig stützen***.
- **Beispiel:**
  - Die beiden Funktionen (IsEven, IsOdd) sind ***indirekt*** rekursiv
  - können festzustellen, ob eine Zeichenfolge eine gerade oder ungerade Anzahl von Zeichen enthält



Beispiel - Indirekte Rekursion

```
MODULE EvenOdd EXPORTS Main;
(* Beispiel fuer die indirekte Rekursion *)
IMPORT SIO, Text;

PROCEDURE IsEven (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN TRUE;
  ELSE RETURN (IsOdd (Text.Sub(str,1)));
  END;
END IsEven ;

PROCEDURE IsOdd (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN FALSE;
  ELSE RETURN (IsEven (Text.Sub(str, 1)));
  END;
END IsOdd ;

BEGIN
  SIO.PutBool ( IsEven (SIO.GetWord()));     SIO.Nl();
  SIO.PutBool ( IsOdd  (SIO.GetWord()));
END EvenOdd.
```

Funktional, Rekursion und Iteration

- **Funktionale Algorithmen**
  - kennen keine ***Variablen*** zur Zwischenspeicherung von Ergebnissen
- **Nichtfunktionale Algorithmen**
  - speichern Zwischenergebnisse in ***Variablen*** ab
- **Rekursive Algorithmen**
  - lösen ein Problem, in dem sie sich ***selbst wieder aufrufen*** (direkt oder indirekt)
- **Iterative Algorithmen**
  - besitzen Abschnitte, die bei der Ausführung ***mehrmals*** durchlaufen werden

Was haben wir gelernt!

- **Aufbau von Modula-3 Programmen**
- **Funktionale Programmierung**
  - Funktion
  - Parameter
  - Vernetzung von Funktionen
- **Anweisungen und Ausdrücke**
- **Datentyp**
  - elementare Datentypen in Modula-3
- **Konzept der Rekursion**
  - rekursive Funktionen

Glossar- 3

- **Modul**
- **Funktionale Programmierung**
- **Funktion**
- **Funktionalform**
- **Parameter**
  - formal , aktuell
- **Rekursion**
  - direkt, indirekt
- **Datentyp**
  - einfacher elementarer Typ
  - Ordinaltyp



Imperative Programmierung

- Modell der imperativen Programmierung
- Variable und Wertzuweisung
- Symbolische Konstanten
- Mechanismen zur Übergabe von Parametern
- Gültigkeitsbereich
- Lebensdauer

Modell der imperativen Programmierung

- **Synomym:**
  - Befehls-orientierte Programmierung



- **Geprägt durch die von-Neumann-Architektur**
  - Die CPU führt ***Maschinenbefehle*** aus
  - Deshalb müssen über den sog. Bus Befehle und Daten vom Speicher in die CPU ***übertragen*** und die Ergebnisse ***rückübertragen*** werden.
- **Mit imperativen Programmiersprachen setzen wir Entwürfe um:**
  - ***Aktionen*** fassen Folgen von Maschinenbefehlen zusammen,
  - ***Variablen*** abstrahieren vom physischen Speicherplatz.

Semantik eines imperativen Progamms

- **Wesentliches Merkmal eines Programms**
  - ***Zustand*** der Daten im Speicher +
  - Stand des Befehlszählers
- **Semantik eines Befehls**
  - Übergang von ZUSTAND1 -> ZUSTAND2
- **Programmierer beschreibt einen Prozeß von Zustandsübergängen**
  - durch Anweisungen
  - durch Art des Kontrollflusses
- ***Variable*** **und** ***Wertzuweisung*** **modellieren Zustand und Zustands-übergang**



Imperatives Programmieren

- **Aufgaben des Programmierers**
  - Planung des ***Speicherbelegung***
    - Welche Daten braucht mein Programm?
  - Planung der ***Operationen***
    - Aus welchen Funktionen und Prozeduren soll das Programm bestehen?
    - Wie sollen diese die Werte der Daten verändern?
  - Planung des ***Kontrollflusses***
    - In welcher Reihenfolge sollen die Operationen abgearbeitet werden?
  - Planung des ***Datenflusses***
    - Welche Daten müssen von welchen Operationen an andere übergeben werden?

Objekte und Aktionen

- **Wir unterscheiden folgende Objektarten:**
  - ***Konstanten***: Objekte, deren Wert während der Ausführung des Algorithmus unverändert bleibt.
  - ***Variablen***: Objekte, deren Wert sich während der Ausführung des Algorithmus verändern kann.
  - Für beide Objektarten gilt, daß ihre Werte einen ***Typ*** haben. Darunter fassen wir zunächst die elementaren Datentypen
- **Wir unterscheiden folgende Arten von Aktionen:**
  - Veränderung des Werts einer Variablen durch ***Zuweisung***.
  - Festlegung der nächsten Aktion durch den sog. ***Kontrollfluß*** (Ablaufsteuerung).

Variable

- **Variable**
  - **logischer** Speicherplatz mit seinem Wert
  - besitzt einen **Namen**, unter dem man die Variable ansprechen kann
  - bei Sprachen mit Typsystem muß jede Variable einem **Datentyp** zugeordnet sein
  - Datentyp legt fest, welche **Werte** eine Variable annehmen kann und welche **Operationen** darauf ausgeführt werden können
  - Variablen werden im Deklarationsteil von Programmeinheiten (Blöcke, Module) vereinbart (deklariert)
  - Variable kann als Behälter betrachtet werden
    - hat einen Wert und einen Typ



Deklarationen (erweitert)



Initialisierung der Variablen

Syntaktisch korrekt

Laufzeitfehler?

```
VAR r : INTEGER := 4;
    x : INTEGER := (4 * 8 -2);

    b := FALSE;
    c := TRUE;
    d := (b = c);

    x : REAL;
```

```
VAR x  : INTEGER := 4;
    xx : INTEGER := (2 * x);
    y  : INTEGER := (4 DIV y);
```


Wertzuweisung

- **Wertzuweisung**
  - dient dazu, den Wert einer Variablen zu verändern
  - Syntax (vereinfacht): **Variable := Ausdruck**
  - Der Ausdruck wird zuerst ausgewertet, das Ergebnis wird anschließend der Variable zugewiesen
    - In diesem Zusammenhang sprechen wir oft von der rechten und der linken Seite einer Zuweisung:
    - (right-hand side - RHS, left-hand side - LHS).
  - Als Ausdruck auf der rechten Seite verwenden wir meist arithmetische und boolesche Ausdrücke, Vergleiche und Zeichen bzw. Zeichenketten.
  - ***Typkompatibilität***:
    - Der Typ des Bezeichners muß zum Typ des Ausdrucks passen, d.h. zunächst, die Typen müssen gleich sein.

Beispiele: Wertzuweisung

- **Nach dieser Deklaration**
  - `VAR x, y, z: CARDINAL;`
- **wären einige (korrekte und inkorrekte) Wertzuweisungen für x:**
  -
```
x :=  0;
x := MAX (CARDINAL);
x :=  y;   (* sicher richtig *)
```
  -
```
x := -1;
x := MAX (CARDINAL)+1;
x := -y-1; (* sicher falsch  *)
```
  -
```
x := y+z; x := z-y;(* richtig oder falsch,  *)
                        (* je nach Wert von y, z *)
```


Beispiel: Wertzuweisung

```
MODULE Vertausche1 EXPORTS Main;
(* Vertauscht zwei eingebenene Werte *)

IMPORT SIO;

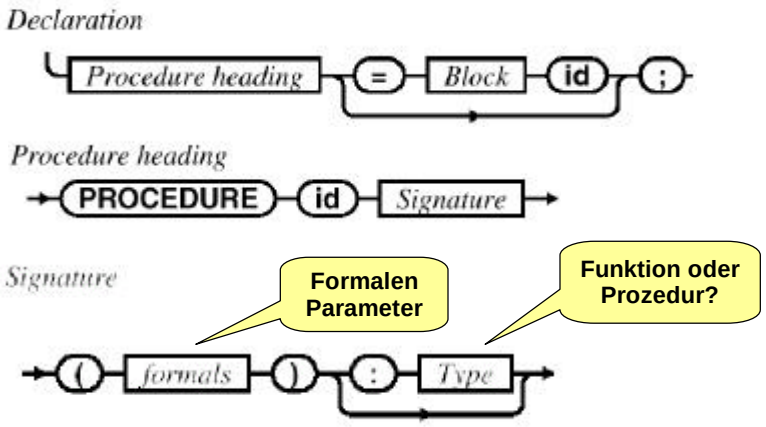
VAR x, y, hilfe : INTEGER;

BEGIN
  x := SIO.GetInt();
  y := SIO.GetInt();

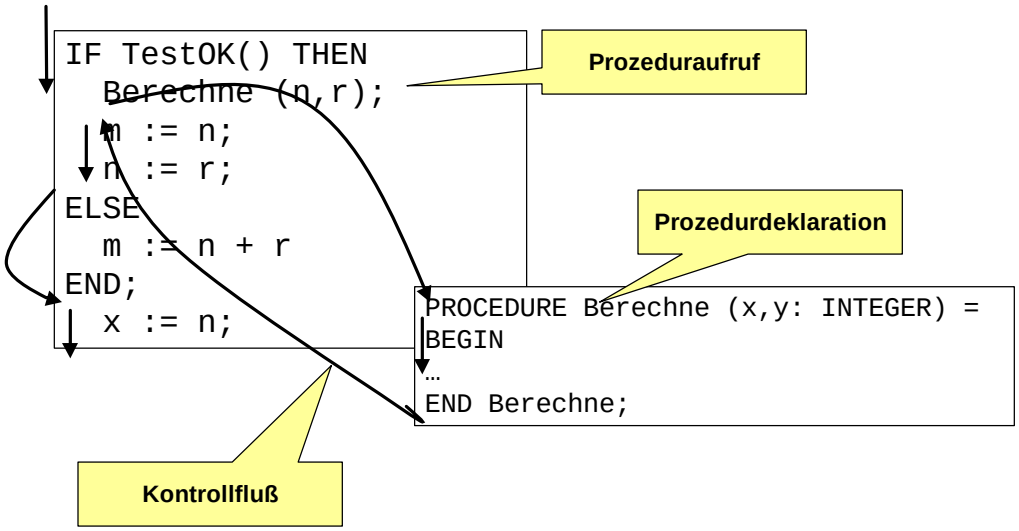
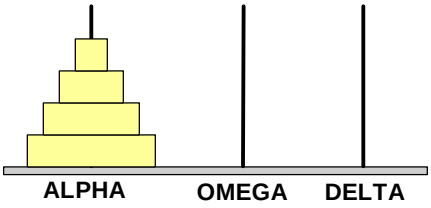
  hilfe := x;
  x     := y;
  y     := hilfe;

  SIO.PutText("x = "); SIO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); SIO.PutInt(y);
END Vertausche1.
```

Deklaration der Variablen

Initialisierung der Variablen

Diskussion der RETURN-Anweisung

```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
  VAR min: INTEGER;
BEGIN
  IF m <= n THEN
    min := m;
  ELSE
    min := n
  END;
  RETURN min;
END Minimum ;
```

```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
BEGIN
  IF m <= n THEN
    RETURN m
  ELSE
    RETURN n
  END;
END Minimum ;
```

- **Anmerkung:**
  - mehrere RETURN-Anweisungen machen eine Funktion leicht unübersichtlich
- **Empfehlung**
  - Code-Effizienz gegen Lesbarkeit abwägen!

Symbolische Konstanten

- **Verwendung von Zahlen-Konstanten ("Literale") in Ausdrücken führt zu Problemen bei der Wartung!**
- **Konstante**
  - Bezeichner mit einem ***festen*** Wert
  - hat einen Datentyp
  - muß deklariert werden
  - überall, wo der Konstantenbezeichner auftritt, wird der Konstantenwert eingesetzt
  - Nach der Deklaration kann ihr ***kein Wert zugewiesen*** werden

- **Beispiel:**

```
CONST PI = 3.141;
VAR umfang, radius : REAL;

…

umfang := 2 * PI * radius
```

```
CONST
  Arbeitstage   = 5;
  Arbeitszeit   = 8;
  Wochenstunden = Arbeitstage *
                  Arbeitszeit;
```

Der Prozedurbegriff

- **Prozedur ist ein zentraler Begriff der prozeduralen Programmierung.**
- **Fachlich ist eine Prozedur**
  - die programmiersprachliche ***Realisierung*** eines Algorithmus.
- **Softwaretechnisch**
  - kann eine Prozedur zunächst als ***benannte Anweisungsfolge*** verstanden werden.
- **Die Grundidee ist,**
  - den Namen der Prozedur "***stellvertretend***" für diese Anweisungsfolge zu verwenden.

Algorithmische Abstraktion

- **Die Prozedur ist eine wesentliche Umsetzung des Konzepts der *algorithmische Abstraktion* (auch *Prozeßabstraktion* genannt):**
  - Statt einer expliziten Anweisungsfolge (der genauen Verarbeitungsvorschrift) wird ein davon ***abstrahierender*** Name verwendet.
- **Abstraktion wird hier sowohl als *Vorgang* als auch als *Ergebnis des Vorgangs* verstanden:**
  - Im Vorgang der algorithmischen Abstraktion sehen wir von der konkreten Anweisungsfolge ab und bringen diese auf "***einen Begriff***".
  - Das Ergebnis ist eine Entwurfs- und ***Programmeinheit*** – die Prozedur.

Beispiel: Abstraktionsprozeß

```
Programm Telefonieren
  Hörer_abheben;
  Telefonnummer_wählen;
  Gespräch_führen;
  Hörer_auflegen;
ENDE Telefonieren.
```

Algorithmische Abstraktion durch **Prozeduren**: Statt einer Anweisungsfolge wird ein **Name** verwendet.

```
Prozedur Telefonnummer_wählen
 IF Telefonnummer_gespeichert
THEN
   Kurzwahltaste_drücken
 ELSE
   Telefonnummer_eintippen
 END
ENDE Telefonnummer_wählen.
```


Kennzeichen von Prozeduren

- **Um den Prozedurbegriff verstehen zu können, benötigen wir drei Begriffe:**
- **Parametrisierung:**
  - Der Mechanismus zum ***Datenaustausch*** zwischen Prozedur und Umgebung.
- **Sichtbarkeit:**
  - Der Programmbereich, in dem ***Namen bekannt*** sind.
- **Lebensdauer:**
  - Der ***Zeitraum***, in dem die Werte von Programmobjekten zugegriffen werden können.

Prozeduren - 1

- **Wurden bisher zur Ausgabe verwendet**
  - SIO.PutText ("Hallo")
- **Sind Funktionen "ähnlich"**
  - werden mit PROCEDURE eingeleitet, können auch rekursiv sein
- **Unterschied zu Funktionen**
  - Prozeduren liefern ***kein*** Ergebnis im Sinne eines Funktionsergebnisses!
  - Konsequenz:
    - Prozeduren besitzen keinen ***Ergebnistyp***
    - Aufruf einer Prozedur ist kein Ausdruck, sondern eine Anweisung
- **Zweck einer Prozedur**
  - ***Zusammenfassen*** einer "Funktionalität" (im Sinne der Lokalität)
  - ***Verändern*** der ihr übergebenen Parameter
  - Durchführung einer ***Nebenwirkung*** (z.B. Ausgabe einer Meldung)

Prozeduren -2

- **Syntax:**



- **Beispiele:**

```
PROCEDURE PutChar(ch: CHAR; wr: Writer := NIL) = ...

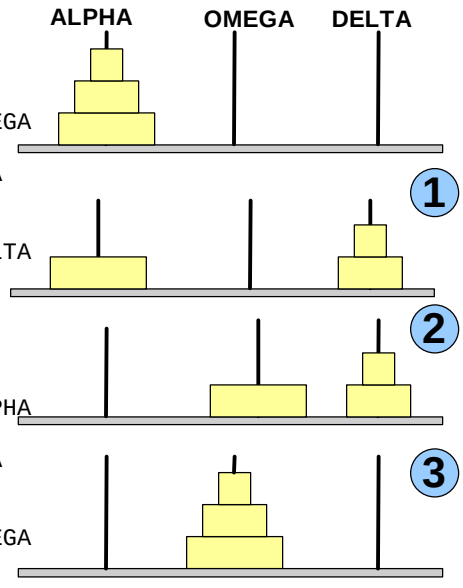
PROCEDURE Minimum(n,m : INTEGER): INTEGER = ...
```

Prozeduraufruf - 1

- **Beim Aufruf einer Prozedur werden die aktuellen Parameter an die formalen übergeben.**
- **Zur Übersetzungszeit wird überprüft:**
  - Der Name im Aufruf muß ***gleich*** dem Prozedurnamen sein.
  - Die ***Anzahl*** der aktuellen Parameter muß gleich der Anzahl der formalen sein.
  - Die Bindung der jeweiligen Parameter wird entsprechend ihrer ***Position*** im Aufruf und in der Prozedurdeklaration vorgenommen.
  - Die aktuellen Parameter müssen ***typkompatibel*** zu den formalen Parametern sein (d.h. meist typgleich).

```
PROCEDURE Minimum ( m, n : INTEGER) : INTEGER = …


res := Minimum (x, y); (* korrekter Aufruf)
res := Mini (x, y);
res := Minimum (x, y, z);
```


Prozeduraufruf - 2

- **Der Prozeduraufruf ist die explizite Anweisung,**
  - daß die Prozedur ***ausgeführt*** werden soll.
- **Eine Prozedur ist *aktiv*,**
  - nachdem sie gerufen wurde und in der Abarbeitung ihrer Anweisungen noch kein vordefiniertes Ende erreicht hat.
- **Für den Prozeduraufruf in imperativen Sprachen ist charakteristisch:**
  - Beim Aufruf wechselt die ***Kontrolle*** (d.h. die Abarbeitung von Anweisungen) vom Rufer zur Prozedur.
  - Dabei werden die aktuellen Parameter an die formalen ***gebunden***.
  - Prozeduren können ***geschachtelt*** aufgerufen werden. Dabei wird der Rufer unterbrochen (suspended), so daß die Kontrolle immer nur bei einer Prozedur ist.
  - Nach der Abarbeitung der Prozedur kehrt die Kontrolle zum Rufer zurück; die Abarbeitung wird mit der ***Anweisung nach dem Aufruf*** fortgesetzt.

Beispiel: Prozedurmechanismus



Rekursion - Türme von Hanoi - 1

- **Aufgabe:**
  - bewege die Scheiben des Turms von ALPHA nach OMEGA
  - es darf immer ***nur eine Scheibe*** bewegt werden
  - niemals darf eine Scheibe auf eine kleinere bewegt werden



- **Lösungsstrategie**
  - allgemeine Lösung für einen Turm der Höhe h von ALPHA nach OMEGA
    - h = 0 gar nichts machen
    - h > 0 1. Turm der Höhe h-1 von ALPHA nach DELTA über OMEGA
      2. Scheibe von ALPHA nach OMEGA legen
      3. Turm der Höhe h-1 von DELTA nach OMEGA über ALPHA

Rekursion - Türme von Hanoi - 2

```
MODULE Hanoi EXPORTS Main;
(* Ausgabe der Zugfolge fuer Tuerme von Hanoi *)

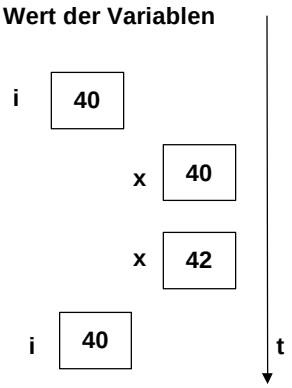
IMPORT SIO;

PROCEDURE DruckeZug (hoehe: CARDINAL; von, nach : TEXT)  =
BEGIN
  SIO.PutText ("Scheibe "); SIO.PutInt (hoehe);
  SIO.PutText (" von " & von & " nach " & nach); SIO.Nl();
END DruckeZug ;

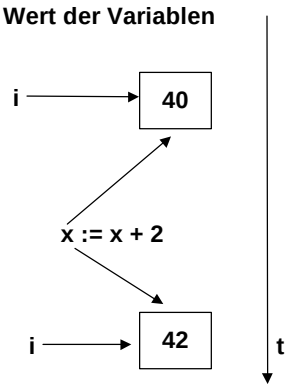
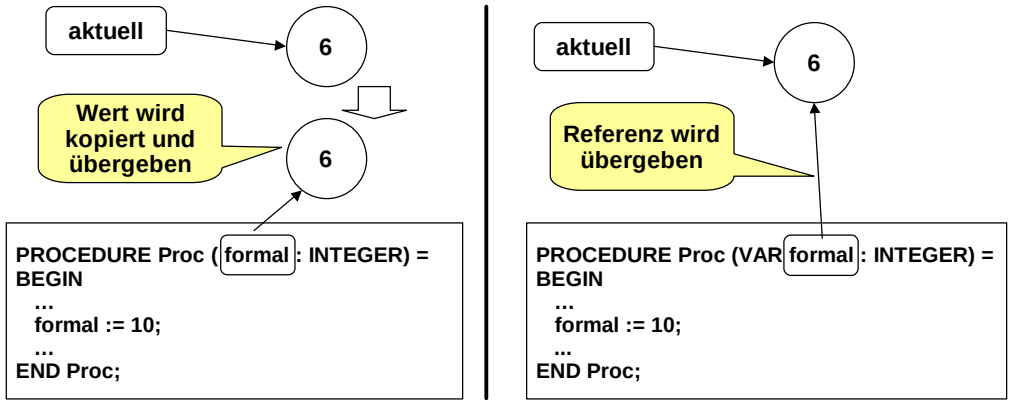
PROCEDURE BewegeTurm ( hoehe : CARDINAL; von, nach, ueber: TEXT)  =
BEGIN
  IF hoehe > 0 THEN
    BewegeTurm (hoehe-1, von, ueber, nach);
    DruckeZug  (hoehe, von, nach);
    BewegeTurm (hoehe-1, ueber, nach, von);
  END;
END BewegeTurm;

BEGIN
  BewegeTurm(SIO.GetInt(), "ALPHA", "OMEGA", "DELTA" );
END Hanoi.
```


Rekursion - Türme von Hanoi - 3

```
3 ALPHA OMEGA DELTA
  2 ALPHA DELTA OMEGA
    1 ALPHA OMEGA DELTA
      0 ALPHA DELTA OMEGA
    ❷ Scheibe 1 von ALPHA nach OMEGA
    ❸ 0 DELTA OMEGA ALPHA
  ❷ Scheibe 2 von ALPHA nach DELTA
  ❸ 1 OMEGA DELTA ALPHA
      0 OMEGA ALPHA DELTA
    ❷ Scheibe 1 von OMEGA nach DELTA
    ❸ 0 ALPHA DELTA OMEGA
❷ Scheibe 3 von ALPHA nach OMEGA
❸ 2 DELTA OMEGA ALPHA
    1 DELTA ALPHA OMEGA
      0 DELTA OMEGA ALPHA
    ❷ Scheibe 1 von DELTA nach ALPHA
    ❸ 0 OMEGA ALPHA DELTA
  ❷ Scheibe 2 von DELTA nach OMEGA
  ❸ 1 ALPHA OMEGA DELTA
      0 ALPHA DELTA OMEGA
    ❷ Scheibe 1 von ALPHA nach OMEGA
    ❸ 0 DELTA OMEGA ALPHA
```



Parameterübergabearten - 1

- **Bisher**
  - Funktionen besitzen ausnahmslos ***Eingabeparameter***
  - Wert dieser Parameter kann nicht ***geändert*** werden
- **Allgemein gibt es folgende Parameterarten für Prozeduren**
  - ***Eingabeparameter***
    - vor dem Aufruf wird der aktuelle Parameter ausgewertet und dem formalen Parameter zugewiesen (***call-by-value***)
  - ***Ausgabeparameter***
    - dienen dazu, Ergebnisse einer Prozedur an den Aufrufer zurückzugeben
    - Wert ist zum Zeitpunkt des Aufrufs undefiniert (***call-by-reference***)
  - ***Ein- / Ausgabeparameter***
    - vereinen Eigenschaften beider Arten
- **Modula-3 nutzt dazu zwei Parameterübergabearten**

Wertparameter - Call by Value

- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Wertparameter***
    - realisieren Eingangsparameter
    - Der aktuelle Parameter muß ein ***Ausdruck*** sein (Spezialfall: Variable, d.h. Bezeichner für ein Objekt).
    - Beim Aufruf der Prozedur wird ein dem Typ des formalen Parameters entsprechendes ***lokales Objekt*** angelegt. Ist der aktuelle Parameter eine Variable, so entsteht dabei eine ***Kopie*** des Parameter-Objekts.
    - In jedem Falle wird der Wert des aktuellen Parameters ***berechnet*** und dem formalen Parameter (-Objekt) zugewiesen.
    - Veränderungen des formalen Parameters in der Prozedur haben nur ***lokale Auswirkung***. Der aktuelle Parameter bleibt unverändert.
    - Schlüsselwort ***VALUE*** zeigt einen Wertparameter an
    - steht kein Schlüsselwort, ist es ***per default*** ein Wertparameter

Beispiel: Call by Value

```
PROCEDURE Proc1 (VALUE x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

IF i = 40 THEN
  SIO.PutLine ("Nichts passiert")
ELSE
  SIO.PutLine ("kein Call by Value")
END;
```



Variablenparameter - Call by Reference

- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Variablenparameter*** (oder Referenzparamter)
    - realisieren Ausgangs und Ein- / Ausgangsparameter
    - Der aktuelle Parameter muß ein ***Bezeichner für ein Objekt*** sein.
    - Beim Aufruf wird der formale Parameter durch einen ***Verweis*** auf den aktuellen Parameter ersetzt.
      D.h. der formale Parameter wird als lokaler Bezeichner für das aktuelle Parameterobjekt ***substituiert***.
    - Jede Änderung des formalen Parameters ist ***direkt*** im aktuellen Parameter wirksam.
    - Veränderungen des formalen Parameters in der Prozedur haben auf den aktuellen Parameter Auswirkung, d.h. Objekte im Namensraum des Rufers können ***verändert*** werden. Auf diese Weise können von einer Prozedur ***Ergebnisse*** zurückgegeben werden.
    - Schlüsselwort ***VAR*** zeigt einen Variablenparameter an

Beispiel: Call by Reference

```
PROCEDURE Proc1 (VAR x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

IF i = 40 THEN
  SIO.PutLine ("Call by Value")
ELSE
  SIO.PutLine ("Call by Reference")
END;
```



call-by-value <-> call-by-reference

```
VAR aktuell : INTEGER
…
aktuell := 6;
Proc (aktuell);
```



Beispiel: Wert- Variablenparameter

```
MODULE Parameter EXPORTS Main;
IMPORT IO, SIO;

VAR aktuell: INTEGER;

PROCEDURE Proc1 (VALUE formal : INTEGER) =
BEGIN
  formal := 10;
END Proc1;

PROCEDURE Proc2 (VAR formal : INTEGER) =
BEGIN
  formal := 10;
END Proc2;

BEGIN
  aktuell := 6;         Proc1 (aktuell);
  SIO.PutText("aktuell (Proc1) = "); IO.PutInt(aktuell); SIO.Nl();
  aktuell := 6;         Proc2 (aktuell);
  SIO.PutText("aktuell (Proc2) = "); IO.PutInt(aktuell);
END Parameter.
```

Wertparameter

Variablen-parameter

aktuell (Proc1) = 6
aktuell (Proc2) = 10

Ausgabe des Programms

Beispiel: Variablenparameter

```
MODULE Vertausche2 EXPORTS Main;
IMPORT IO, SIO;

PROCEDURE Vertausche (VAR w1, w2 : INTEGER) =
VAR hilfe : INTEGER;
BEGIN
  hilfe := w1;
  w1    := w2;
  w2    := hilfe;
END Vertausche;

VAR x, y : INTEGER;

BEGIN
  x := IO.GetInt();
  y := IO.GetInt();
  Vertausche (x, y);
  SIO.PutText("x = "); IO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); IO.PutInt(y);
END Vertausche2.
```

Vertauscht die Werte der beiden Parameter

Beispiel: Ausgabeparameter

- **Ausgabeparameter**
  - dient dazu, einen Wert an den Aufrufer zurückzugeben

```
MODULE QuadratM1 EXPORTS Main;

IMPORT SIO;
CONST eingabe = 2.5;
VAR  ergebnis : REAL;

PROCEDURE Quadrat ( VALUE x : REAL; VAR wert : REAL )=
BEGIN
  wert := ( x * x );
END Quadrat;

BEGIN
  Quadrat (eingabe , ergebnis);
  SIO.PutReal (ergebnis);
END QuadratM1.
```

Ausgabeparameter oder Ein- Ausgabeparameter ?

Wert ist beim Aufruf undefiniert

Datenaustausch: Beispiel - 1

In einer Reihe von Meßwerten soll der **laufende Mittelwert** berechnet werden. Benötigte Objekte und Aktionen:

```
Wert, Mittelwert, Summe : REAL;
Anzahl : INTEGER;
(* BerechneMWert
      Addiere neuen Wert und Summe,
      Erhöhe Anzahl um 1,
      Mittelwert := Summe / Anzahl *)
```

Austausch über Parameter:

```
PROCEDURE BerechneMWert (     wert     : REAL;
                          VAR anzahl    : INTEGER;
                          VAR summe, mw : REAL) =

BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  mw := summe / anzahl;
END BerechneMWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;
BerechneMWert (wert,anzahl,summe,mw);
BerechneMWert (2*wert,anzahl,summe,mw);
```

Datenaustausch: Beispiel - 2

Datenaustausch über Funktionsergebnis:

```
PROCEDURE MWert (    wert  : REAL;
                 VAR anzahl: INTEGER;
                 VAR summe : REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl:= anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;

SIO.PutReal (MWert(wert,anzahl,summe));
...
SIO.PutReal (MWert(2*wert,anzahl,summe));
```


Datenaustausch: Beispiel - 3

Datenaustausch über Funktionsergebnis:

```
VAR summe: REAL; anzahl: INTEGER;

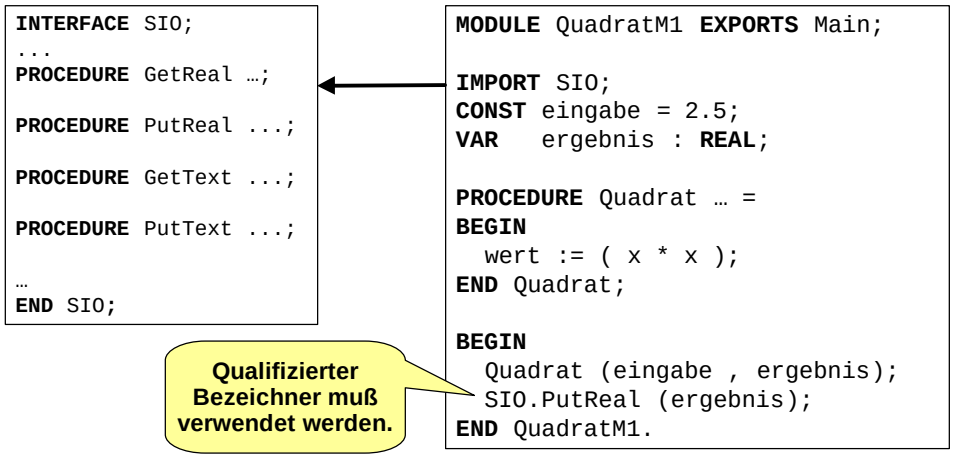
PROCEDURE MWert (wert:REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0;

SIO.PutReal (MWert(4.0);
...
SIO.PutReal (MWert(10.0));
```

Diskussion der Beispiele - 1

- **Beispiel 1:**
  - Die Verwendung von VAR-Parametern in einer Prozedur zur Rückgabe von Ergebnissen ist in der imperativen Programmierung üblich.
  - Sie führt oft zu ***Verständnisproblemen***, da sowohl in der Deklaration als auch in der Verwendung klar sein muß, was das eigentliche Ergebnisobjekt (hier: der Mittelwert) ist.
- **Beispiel 2:**
  - Die Verwendung einer Funktion zur Berechnung genau eines Wertes ist dann sauber, wenn alle anderen Parameter als ***Wertparameter*** verwendet werden.
  - Die Modellierung einer Zustandsveränderung (hier: Summe und Anzahl) außerhalb der Funktion mit Hilfe von VAR-Parametern ist ein ***Seiteneffekt***, der die ***Lokalität*** der Funktion zerstört.

Diskussion der Beispiele - 2

- **Beispiel 3:**
  - Die Verwendung von ***globalen*** Variablen, die in einer Prozedur oder Funktion als Seiteneffekt verändert werden, ist die ***schlechteste*** Lösung.
  - Zustandsveränderungen über globale Variablen, die in der Signatur der Prozedur nicht aufgeführt sind, sind unverständlich und extrem ***fehleranfällig***.
- **Funktionsprozeduren**
  - Eine Funktion kann in imperativen Sprachen nur dann sauber modelliert werden, wenn
    - alle Parameter als ***Wertparameter*** übergeben werden,
    - in der Funktion ***keine globalen Variablen*** verwendet werden.
  - In Funktionen sollte auch keine globalen Variablen ***lesend verwendet*** werden, da dies Funktionen an ihren Verwendungskontext ankoppelt.
  - Merke: Funktionen sollten "***in sich***" verständlich sein.

Regeln für die Parameterverwendung

- **Wertparameter sind Variablenparameter vorzuziehen**
  - Änderung an Parametern in Prozeduren ist häufig eine ***Fehlerquelle***, die schwer zu finden ist.
- **Variablenparameter sollten nur verwendet werden, wenn**
  - Prozedur-Ergebnisse ***übergeben*** werden sollen (Ausgabeparameter)
  - Kopieren des Parameters ***nicht möglich*** ist (bei gewissen Datenstrukturen)
  - Kopieren zu ***ineffizient*** ist (bei sehr großen Datenstrukturen)
- **Funktionen haben nur Wertparameter**
  - liefern ihr Ergebnis durch ihren ***Namen*** zurück
  - geht in Modula-3 nur, wenn
    - genau ein Wert zurückgegeben werden soll
  - Rückgabe durch Namen und Variablenparameter muß ***vermieden*** werden

Zusammenfassung: Prozedurkonzept

- **In imperativen Sprachen wird oft die Prozedur in den Vordergrund gestellt.**
- **Die Funktion ist gelegentlich (z.B. Modula-3) nur eingeschränkt verwendbar, kann dafür aber *Seiteneffekte* erzeugen.**
- **Nur durch eine saubere Definition der *Signaturen* von Routinen kann ein verständlicher und weiterverwendbarer Entwurf erreicht werden, d.h. vor allem, jeder Datenaustausch mit der Umgebung sollte *explizit* sein.**
- **Die Modellierung von Zuständen und ihrer Veränderung ist nur in Verbindung mit einem entsprechenden *Modulkonzept* softwaretechnisch sauber zu lösen.**
  - Das lernen wir dann später!

Gültigkeitsbereich und Lebensdauer

- **Gültigkeitsbereich (scope) eines Bezeichners**
  - der ***statische Teil*** des Programms, in dem der Bezeichner mit exakt ***gleicher Bedeutung*** verwendet werden darf
  - wird auch ***Sichtbarkeitsbereich*** oder ***Namensraum*** genannt
  - Bezeichner ist in seinem Sichtbarkeitbereich gültig
  - der Gültigkeitsbereich wird durch den Compiler überwacht
- **Lebensdauer eines Objekts (Variable, Prozedur)**
  - bezieht sich auf den zur ***Programmlaufzeit*** belegten Speicherplatz
  - macht nur Sinn für Objekte, die Speicher belegen
    - Konstanten
    - Typen
    - belegen keinen Speicher
- **Es ist wichtig, beide Begriffe klar zu unterscheiden!**

Gültigkeitsbereich - 1

- **Regeln für die Gültigkeit von Bezeichnern;**
  - Alle in einer Prozedur oder einem Modul deklarierten Bezeichner sind in der gesamten Prozedur / im gesamten Modul gültig.
  - Das gilt auch für den textuell vor der Deklaration eines Bezeichers liegenden Bereich
  - Davon ausgenommen sind
    - Prozedur- und Modulkopf

```
PROCEDURE Proc ( in : REAL )=

  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Hier können die Bezeichner X und C verwendet werden.

```
PROCEDURE Proc ( in : REAL )=

  CONST CC : C * C
  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Korrekte Vorwärts-referenz

Gültigkeitsbereich - 2

- **Durch IMPORT kann der Gültigkeitsbereich von Bezeichnern auf das *importierende Modul* erweitert werden.**



Prozeduren als Namensraum - 1

- **Namensraum**
  - Eine Prozedur bildet gegenüber der (textuellen) Umgebung ihrer Deklaration einen eigenen ***Namensraum***, d.h. sie kann lokale Objekte benennen und verwalten.
  - Die Namen der formalen Parameter sowie die im Deklarationsteil des Prozedurrumpfs deklarierten Bezeichner sind nur ***im Prozedurrumpf gültig***, d.h. bekannt.
  - In einer Prozedur können Bezeichner der Umgebung ***neu lokal*** deklariert werden; diese Bezeichner ***verdecken*** die global deklarierten Bezeichner, die zugehörigen Objekte sind in der Prozedur ***nicht sichtbar***.
- **Lebensdauer**
  - Die Lebensdauer von prozedur-lokalen Objekten entspricht dem Zeitraum, in dem der Aufruf der Prozedur abgearbeitet wird. Sie werden zu Beginn der Prozedurausführung angelegt.
  - Werte gehen ***verloren***, wenn die Ausführung der Prozedur beendet ist.

Prozeduren als Namensraum - 2

```
PROCEDURE Proc ()=

  VAR x : INTEGER;

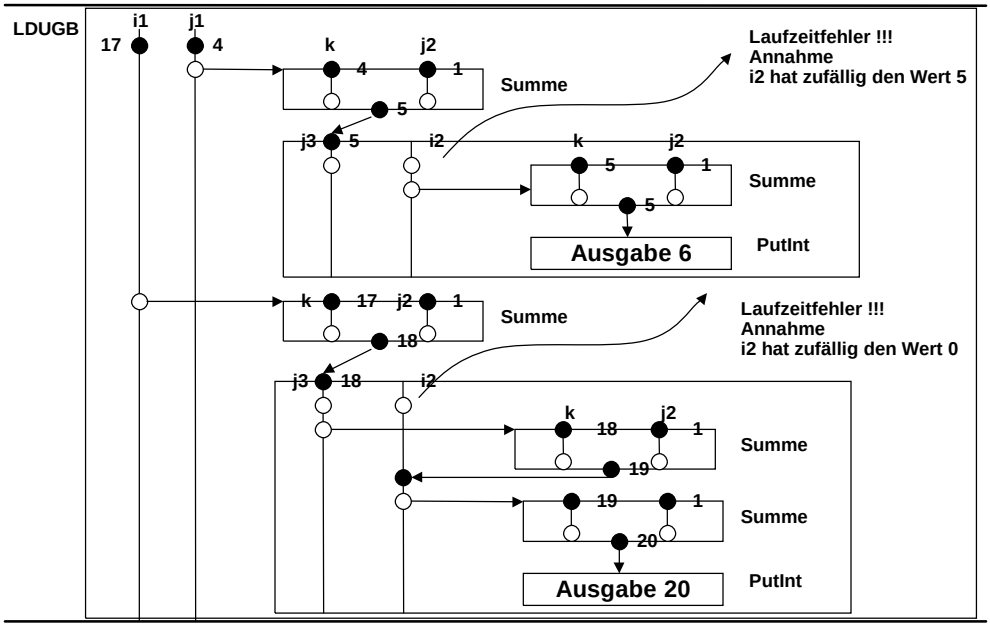
  PROCEDURE Proc1 (x :Integer) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Proc;
```

G2

G3

G1

- **Module und Prozeduren definieren eigene Namensräume**
  - Prozeduren können verschachtelt werden
  - Hierarchie von Gültigkeits-bereichen (Namensräumen)
- **Überlappung von Gültigkeitsbereichen**
  - Liegt innerhalb des Gültigkeitsbereiches G1 von X mit der Bedeutung B1 ein weiterer Gültigkeitsbereich G2 von X mit der Bedeutung B2, so handelt es sich um zwei ***verschiedene*** Objekte
  - innerhalb von G2 gilt nur B2, alle anderen Bedeutungen sind ***unsichtbar***.

Lebensdauer - 1

- **Durch Ausführung einer Prozedur P entsteht eine *Inkarnation*.**
- **Zur Inkarnation gehören *zur Laufzeit*:**
  - ein ***Ausführungspunkt*** (also ein Zeiger auf den gerade auszuführenden oder ausgeführten Befehl)
  - ***Speicherplätze*** für alle Bezeichner von Variablen und Wertparameter
  - ***Bezüge*** auf die konkreten Variablenparameter.
- **Informationen existieren bis zum Ende der Ausführung von P.**
- **Beispiel**
  - `PROCEDURE Test (ch: CHAR; VAR x:INTEGER)=`
    `VAR y: REAL`
- **`Test ('a', z)` führt zu einer Inkarnation mit**
  - Speicherplatz für ch und y
  - unter dem lokalen Bezeichner x einen Bezug (einer Referenz) auf z
  - nach Abschluß werden diese Speicherplätze wieder freigegeben

Lebensdauer - 2

- **Bemerkungen:**
  - **Sei M ein Modul,**
    - dann wird ***Speicherplatz*** für die Variablen permanent für die gesamte Laufzeit des Programms reserviert.
    - Eine eigentliche Inkarnation wird nur von dem Rumpf des Moduls gebildet; dieser hat weder Variablen noch Parameter.
  - Zu irgendeinem Zeitpunkt existieren i.a. neben der Inkarnation des ablaufenden Moduls ***Inkarnationen verschiedener*** Prozeduren bei **rekursiven** Aufrufen auch mehrere der gleichen Prozedur.
  - Nur in der ***jüngsten*** aller existierenden Inkarnationen wandert der Ausführungspunkt weiter; diese wird als erste beendet.
  - Die Lebensdauer einer Variablen ist ***identisch*** mit der Existenz der zugehörigen Prozedur-Inkarnation.
  - Zu Beginn der Lebensdauer ist der Wert einer Variablen ***undefiniert***, darf also nicht verwendet werden.

Beispiel: Lebensdauer - 1

```
MODULE LDUGB EXPORTS Main;
IMPORT SIO;
VAR i  , j  : INTEGER;

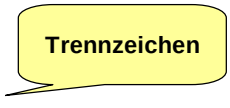
PROCEDURE Summe (k  : INTEGER) : INTEGER =
VAR j  : INTEGER;
BEGIN
  IF i  <10 THEN j  := 10 ELSE j  :=1 END;
  RETURN j  + k  ;
END Summe;

PROCEDURE Drucken (j  : INTEGER) =
VAR i  : INTEGER;
BEGIN
  IF i   #   j   THEN i  := Summe(j  ) END;
  SIO.PutInt (Summe(i  )); SIO.Nl();
END Drucken;

BEGIN
  i  := 17; j  := 4;
  Drucken (Summe(j  ));
  Drucken (Summe(i  );
END LDUGB.
```

Beispiel: Lebensdauer - 2



Terminologie: Gültigkeit & Lebensdauer

- **"Objekt" bezeichnet alles,**
  - was durch einen Bezeichner eingeführt wird (Modul, Prozedur, Konstante, Typ, Variable, Parameter),
  - keine Objekte sind demnach Operatoren oder Wortsymbole (z.B. VAR, END)
- **Ein Objekt heißt lokal in Block B,**
  - wenn es im Block B deklariert ist.
- **Ein Objekt heißt global,**
  - wenn es auf Modulebene deklariert ist.
- **Ein Objekt heißt global relativ zu B,**
  - wenn es in B gültig ist, aber nicht lokal in B ist

Beispiel: lokal, global, global relativ

```
MODULE Gueltigkeit EXPORTS Main;
  IMPORT SIO;
  VAR x : INTEGER;

  PROCEDURE Proc1 (x: INTEGER) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

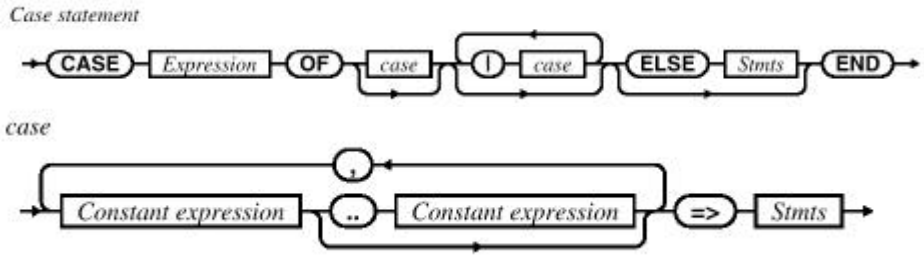
  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Gueltigkeit.
```

x ist global sichtbar im Modul

x ist lokal in der Prozedur Proc1

x ist global relativ in der Prozedur Proc2

y ist lokal in der Prozedur Proc2

Lokalität

- **Ziel**
  - Programme sollten mit ***möglichst wenig*** Aufwand korrigier- und modifizierbar sein.
- **Strategie**
  - ***hohe Lokalität*** durch enge Gültigkeitsbereiche.
  - Größtmögliche Lokalität ist daher ***vorrangiges Ziel*** einer guten Programmierung!
- **Ein Programm sollte dafür folgende Merkmale aufweisen:**
  - Die auftretenden Programmeinheiten (Prozeduren, Funktionen, Hauptprogramm) sind ***überschaubar***.
  - Die Objekte sind so ***lokal*** wie möglich definiert, jeder Bezeichner hat nur eine ***einzige***, bestimmte Bedeutung.
  - Die ***Kommunikation*** zwischen Programmeinheiten erfolgt vorzugsweise über eine möglichst kleine Anzahl von Parametern, nicht über globale Variablen.

Vorteile lokaler Variablen

- **Lokale Variablen haben softwaretechnisch einige Vorteile.**
  - Deklaration und Verwendung stehen in einem ***textlichen*** Zusammenhang. Das erhöht die Lesbarkeit.
  - Die ***unfreiwillige*** Verwendung von globalen Variablen wird vermieden, d.h. globale Variablen müssen nicht vollständig bekannt sein.
  - Der Speicherverbrauch durch Prozeduren wird ***minimiert***, da Speicher für lokale Variablen nur während ihrer Aktivierung vorgehalten werden muß.
  - Lokalität:
    - Variablen sollen ***nur in dem Kontext*** deklariert werden, wo sie bekannt sein müssen. Eine Verteilung erschwert die Änderbarkeit.
  - Kapselung:
    - Außerhalb eines Kontextes (Modul, Prozedur) sollen nur relevante Objekte sichtbar sein. Implementationsdetails werden im Inneren ***verborgen*** und sind nicht zugreifbar.

Was haben wir gelernt!

- **Modell der imperativen Programmierung**
  - Zustand, Zustandsübergang
- **Konzept der Variablen**
  - Wertzuweisung
- **Parameterübergabemechanismen**
  - Wertparameter
  - Variablenparameter
  - Wie geht man damit um
- **Gültigkeit von Bezeichnern**
- **Lebensdauer von Objekten**
- **Konzept der Lokalität**

Kontroll-strukturen

- **Auswahlanweisungen**
  - IF
  - CASE
- **Wiederholungsanweisungen**
  - WHILE, FOR
  - REPEAT-UNTIL, LOOP-EXIT

Kontrollfluß (Ablaufsteuerung)

- **Ablaufsteuerung in der von Neumann-Maschine:**
  - Befehle stehen ***hintereinander*** im Speicher, werden vom Steuerwerk in den zentralen Prozessor geholt, dort decodiert und verarbeitet.
  - Durch ***Sprungbefehle*** kann von der Reihenfolge der gespeicherten Befehle abgewichen werden.
- **Abstraktion:**
  - Die Ausführungsreihenfolge der Aktionen eines Algorithmus entspricht zunächst der textuellen Anordnung (***Sequenz***). Davon kann aber abgewichen werden. Dazu gibt es eigene Aktionen.
  - Ablaufsteuerung:
    - Fallunterscheidung,
    - Wiederholungsaktionen.
- **Ablaufsteuerung in imperativen Programmiersprachen:**
    - Anweisungsfolgen,
    - IF - THEN - ELSE- und CASE-Anweisungen,
    - Schleifenanweisungen.

Kontrollstrukturen

- **Berechnungen in imperativen Programmen erfolgen durch die *Auswertung von Ausdrücken* und die *Zuweisung* von Werten zu Variablen.**
- **Zur Erhöhung der Flexibilität von Programmen sind zwei weitere Sprachmechanismen notwendig:**
  - die Steuerung des Kontrollflusses zur ***Auswahl*** zwischen unterschiedlichen Anweisungen,
  - die ***wiederholte*** Ausführung einer Folge von Anweisungen.
- **Sprachmechanismen, die dies leisten,**
  - heißen ***Kontrollanweisungen***.
- **Kontrollanweisungen**
  - zusammen mit den Anweisungsfolgen, die von ihnen kontrolliert werden, heißen ***Kontrollstrukturen***.

Diskussion: Kontrollstrukturen

- **Von Mitte der 60er bis Mitte der 70er wurde in der Softwaretechnik (Informatik) viel über *Kontrollstrukturen* diskutiert.**
- **Ein Ergebnis war:**
  - Obwohl eine einzige ***Anweisung*** (das GOTO oder die unbedingte Verzweigung) ausreicht, sollte genau diese Anweisung durch eine ***kleine Zahl*** von Kontrollanweisungen ersetzt werden.
- **Bohm und Jacopini haben 1966 nachgewiesen, daß alle Algorithmen, die in Flußdiagrammen ausgedrückt werden können, durch zwei Kontrollanweisungen implementierbar sind:**
  - ***Auswahl*** zwischen alternativen Kontrollflüssen,
  - logisch kontrollierte ***Wiederholung***.
- **Kontrollstrukturen sollen einen Einstieg und einen Ausstieg (single entry, single exit) besitzen.**

Diskussion: Goto

- **Obwohl die *unbedingte Verzweigung* (Goto) ausreicht,**
  - alle anderen Kontrollstrukturen nachzubilden, führt ihre uneingeschränkte Verwendung zu unlesbaren und unzuverlässigen Programmen.
- **Hauptgrund:**
  - Durch Goto kann im Ablauf ***jede beliebige Reihenfolge*** von Anweisungen unabhängig von ihrer textlichen Anordnung erreicht werden.
  - In seinem berühmten Artikel ("Goto statement considered harmful", CACM, 1968, Vol.11, No.3, pp.147-149) schreibt E.W. Dijkstra: "The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."
- **Dies hat die Goto-Debatte entzündet,**
  - die zwar zur softwaretechnischen Ablehnung des ***uneingeschränkten Goto*** geführt hat,
  - aber nur ***wenige*** Programmiersprachen haben völlig auf dieses Konstrukt verzichtet (Modula-2, Modula-3, Java).

Konzept: Sequenz

- **Sequenz von Aktionen:**
  - Eine Aktion wird ***nach der anderen*** abgearbeitet.
  - Dazu muß nur klar sein, wie zwei Aktionsbeschreibungen voneinander ***getrennt*** sind.
  - Ein Aktion ist auch "tue nichts".
- **Beispiel:**
  - Algorithmus Telefonieren:
    - `hebe den Hörer ab;`
    - `wähle die Telefonnummer;`
    - `führe das Gespräch;`
    - `lege den Hörer auf.`



Konzept: Auswahlanweisung

- **Auswahlanweisungen kommen vor als:**
  - Einwegauswahl (one-way selection)
  - Zweiwegauswahl (two-way-selection)
  - Mehrfachselektion (multiple selection)

```
IF b THEN
  f2;
ELSE
  f3;
END;
```

Zweiwegauswahl

```
IF b THEN
  f2;
END;
```

Einwegauswahl

- **Anwendbar**
  - Typ des Ausdrucks, der die Auswahl bestimmt, ist BOOLEAN

Mehrfachselektion

- **Die *Mehrfachauswahl* ermöglicht die Auswahl aus einer *beliebigen* Anzahl von Alternativen.**
- **Designentscheidungen sind:**
  - Welche Form und Typ von Ausdruck ***kontrolliert*** die Mehrfachselektion?
  - Welche Form von Anweisung kann ***ausgewählt*** werden?
  - Wie ist der ***Kontrollfluß*** innerhalb der Mehrfachselektion?
  - Wie werden Selektionswerte behandelt, für die es ***keine*** passende Anweisungsalternative gibt?
- **Programmiersprachen realisieren die Mehrfachselektion durch die**
  - CASE-Anweisung

CASE-Anweisung - 1

- **Hängen die Fälle einer Fallunterscheidung nur von unterschiedlichen Werten *eines Ausdrucks* ab,**
  - können wir die in modernen imperativen Sprachen vorhandene *CASE-Anweisung* verwenden.
- **Der ELSE-Teil ist im Beispiel leer, um einen *Fehlerfall* zu vermeiden.**
- **Dies ist eine häufige aber *schlechte* Programmiertechnik.**
- **Besser:**
  - möglichen Fehlerfall explizit behandeln.

```
CASE zweierpotenz OF
  0 => ergebnis := 1;
| 1 => ergebnis := 2;
| 2 => ergebnis := 4;
| 3 => ergebnis := 8;
| 4 => ergebnis := 16;
ELSE
END;
```


CASE-Anweisung - 2

```
MODULE CaseDemo EXPORTS Main;
IMPORT SIO;
VAR zweierpotenz, ergebnis: INTEGER;

BEGIN
  zweierpotenz := SIO.GetInt();
  ergebnis := 0;

  CASE zweierpotenz OF
    0 => ergebnis := 1;
  | 1 => ergebnis := 2;
  | 2 => ergebnis := 4;
  | 3 => ergebnis := 8;
  | 4 => ergebnis := 16;
  ELSE
    SIO.PutText ("Wert nicht definiert!");
  END;

  SIO.Nl(); SIO.PutInt(ergebnis);
END CaseDemo.
```

Bemerkung:

Werden nicht alle Werte als Alternativen aufgeführt und fehlt des ELSE-Zweig, dann kann das zu *Laufzeitfehlern* führen!!

Fehlerbehandlung im ELSE-Zweig ???

CASE-Anweisung - 3

```
MODULE CaseDemo EXPORTS Main;
IMPORT SIO;
VAR zweierpotenz, ergebnis: INTEGER;

BEGIN
  zweierpotenz := SIO.GetInt();
  IF (zweierpotenz >= 0 AND zweierpotenz <= 4) THEN
    CASE zweierpotenz OF
      0 => ergebnis := 1;
    | 1 => ergebnis := 2;
    | 2 => ergebnis := 4;
    | 3 => ergebnis := 8;
    | 4 => ergebnis := 16;
    END;
    SIO.PutInt(ergebnis);
  ELSE
    SIO.PutText ("Wert nicht definiert!");
  END;
END CaseDemo.
```

Fehlersituation wird explizit vor Ausführung der CASE-Anweisung behandelt!

Syntax CASE-Anweisung



- **Zusatzbedingungen für die Case-Anweisung:**
  - Ergebnis von *Expression* und Ergebnis der *Constant expressions* müssen ***denselben*** Typ haben.
  - Zugelassen sind nur ***Ordinaltypen*** (z.B. BOOLEAN oder CHAR)
  - Die Werte dürfen jeweils ***nur einmal*** auftreten.

Konzept: Wiederholungsanweisung

- **Wiederholungsanweisungen werden benötigt,**
  - um ***iterative Algorithmen*** zu formulieren.
- **Die historisch ersten Sprachkonstrukte zur Wiederholung waren**
  - für die ***Array-Bearbeitung*** gedacht,
  - da anfangs vorrangig numerische Probleme gelöst werden mußten.
- **Designentscheidungen sind:**
  - Wie wird die Wiederholung kontrolliert,
    - durch einen ***logischen Ausdruck***,
    - durch ***Abzählen***?
  - Wo steht der Kontrollmechanismus im Programmtext,
    - am ***Anfang***,
    - am ***Ende***,
    - ***benutzerdefiniert***?

FOR-Anweisung

- **Die FOR-Anweisung ist eine spezielle Form der WHILE-Anweisung**
  - Anzahl der Wiederholungen ist ***im voraus*** bekannt
  - Wiederholungen werden durch Abzählen kontrolliert
- **Syntax:**

- **Anmerkungen:**
  - Die Steuerung und der Abbruch geschieht mit Hilfe der sog. ***Laufvariablen***, deren Schrittweite und Laufrichtung ggf. verändert werden kann.
  - In Modula-3: Die Laufvariable muß ***nicht deklariert*** werden.
  - Bei jedem Durchlauf wird die Laufvariable "***automatisch***" erhöht oder erniedrigt.
  - FOR-Schleifen werden oft bei der Bearbeitung von ***Arrays*** verwendet.

Beispiel: FOR-Anweisung

- **Ein Beispiel für die FOR-Schleife:**
  - ```
    (* Berechne Summe I = 1 bis 100 ueber I *)
    r := 0;
    FOR i := 1 TO 100 DO
      r := r + i
    END;
    ```
- **FOR-Schleife mit Schrittweite:**
  - ```
    (* Berechne Summe I = 1 bis 100 ueber I *)
    r := 0;
    FOR i := 100 TO 1 BY -1 DO
      r := r + i
    END;
    ```

Innerhalb der FOR-Schleife muß die Laufvariable wie eine ***Konstante*** behandelt werden, darf also nicht auf der linken Seite einer Wertzuweisung oder als Referenzparameter stehen und damit ***manipulierbar*** sein.

Schleife mit vorheriger Prüfung

- **WHILE-Anweisung**
  - "***ablehnende Schleife***", da Prüfung vor Schleifendurchlauf gemacht wird
- **Syntax:**
  ```
  WHILE E DO
    S;
  END;
  ```

- **Bemerkung**
  - Ergebnis der "Expression" muß vom ***Typ BOOLEAN*** sein
  - Der Programmierer muß dafür sorgen, daß das Ergebnis von "Expression" ***irgendwann FALSE*** ist; ansonsten Endlosschleife; Programm terminiert nicht
- **Semantik (rekursiv definiert):**
  - ```
    IF E THEN
      S;
      WHILE E DO S; END;
    END;
    ```
    ```

Beispiel: WHILE-Anweisung

- **Aufgabe:**
  - Man berechne den Quotienten und den Rest der ganzzahligen Division zweier positiver ganzer Zahlen a und b.

```
MODULE GanzzahlDivision EXPORTS Main;
IMPORT SIO;
VAR a, b, c: INTEGER;

BEGIN
  a := SIO.GetInt();
  b := SIO.GetInt();

  c := 0;  (*enthalte den Quotienten *)
  WHILE a >= b DO
    a := a - b;
    c := c + 1;
  END;

  SIO.PutText(" Quo :");  SIO.PutInt(c);
  SIO.PutText(" Rest :"); SIO.PutInt(a);

END GanzzahlDivision.
```

Wiederhol-bedingung

Schleife mit nachfolgender Prüfung

- **REPEAT-UNTIL Anweisung**
  - "***annehmende***" Schleife, da eine erste Ausführung stattfindet, ohne daß die Bedingung (in diesem Fall eine ***Abbruch-Bedingung***) geprüft wird
- **Syntax:**

- **Bemerkung**
  - REPEAT-UNTIL erfordert besondere Vorsicht
  - REPEAT-UNTIL ist immer dann sinnvoll, wenn die Bedingung erst durch die ***Anweisungen der Schleife*** entsteht
  - z.B. Eingabe mit Fehlerbehandlung

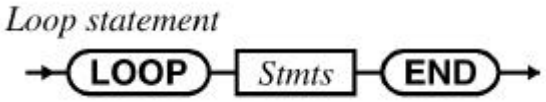
Beispiel: REPEAT-UNTIL

```
MODULE Einleseschleife EXPORTS Main;
IMPORT SIO;
VAR a: INTEGER;

BEGIN

  REPEAT
    SIO.PutText("Geben Sie eine Zahl zwischen 1 und 5 ein! ");
    SIO.Nl();
    a := SIO.GetInt();
  UNTIL (a >=1 AND a <=5);

  SIO.PutText("Ihre Eingabe war korrekt! ");

END Einleseschleife.
```

Abbruch-bedingung

Schleife ohne Prüfung

- **LOOP-Anweisung**
  - die Schleife wird solange durchlaufen, bis eine darin enthaltene ***EXIT-Anweisung*** ausgeführt wird
- **Syntax:**



- **Bemerkungen:**
  - in einer LOOP-Anweisung können ***mehrere*** EXIT-Anweisungen stehen
  - single-entry - single-exit wird dadurch verletzt
  - ist keine EXIT-Anweisung enthalten, so realisiert die LOOP-Anweisung eine ***nicht terminierende Schleife***

Beispiel: LOOP-Anweisung - 1

Algorithmus GGT

Falls n < m ist, so vertausche man n und m

❷ Falls m = 0 ist, dann ist n der ggt(n,m) und man beende den Algorithmus

❸ Falls m≠ 0 ist, so bilde man den Rest r, der bei der Division von n durch m bleibt, dann ersetze man n durch m und m durch r und beginne von vorn.

```
MODULE GGT EXPORTS Main;
VAR n, m, hilf, r : INTEGER;
BEGIN
  n := SIO.GetInt();
  m := SIO.GetInt();
  LOOP
    IF n < m THEN  (* 1 *)
      hilf := m;
      m := n;
      n := m;
    END;
    IF m = 0 THEN  (* 2 *)
      EXIT;
    ELSE           (* 3 *)
      r := n MOD m;
      n := m;
      m := r;
    END;
  END (* LOOP *);
  SIO.PutText("GGT = ");SIO.PutInt(n);
END GGT.
```


Beispiel: LOOP-Anweisung - 2

Bemerkungen

- LOOP-Anweisungen terminieren nicht!
- Die Ausführung eines Exits terminiert die Loop-Anweisung. Die Kontrolle geht zur Anweisung ***nach dem END***.
- Bei geschachtelten Loop-Strukturen bewirkt die Exit-Anweisung nur das Verlassen der ***zugehörigen*** Loop-Struktur.

Empfehlung

- Aus Gründen der besseren Lesbarkeit sollte man überall dort WHILE- und REPEAT-Schleifen zu verwenden, wo nur eine Abfrage am ***Anfang*** oder ***Ende*** der Schleife erforderlich ist!

```
MODULE GGT1 EXPORTS Main;
IMPORT SIO;
VAR n, m, r : INTEGER;

BEGIN
  n := SIO.GetInt();
  m := SIO.GetInt();
  r := n MOD m;
  LOOP
    IF r = 0 THEN
      EXIT;
    ELSE
      n := m;
      m := r;
      r := n MOD m;
    END;
  END;
  SIO.PutText ("GGT = ");
  SIO.PutInt(m);
END GGT1.
```

LOOP versus WHILE

```
n := SIO.GetInt();
m := SIO.GetInt();

r := n MOD m;
LOOP
  IF r = 0 THEN
    EXIT;
  ELSE
    n := m;
    m := r;
    r := n MOD m;
  END;
END;

SIO.PutText ("GGT = ");
SIO.PutInt(m);
```

```
n := SIO.GetInt();
m := SIO.GetInt();

r := n MOD m;
WHILE r # 0 DO
  n := m;
  m := r;
  r := n MOD m;
END;

SIO.PutText ("GGT = ");
SIO.PutInt(m);
```

- **Hier bietet sich die WHILE-Anweisung an,**
  - da am Beginn der Schleife die Bedingung geprüft werden soll.

Wahl des Schleifenkonstrukts

- **REPEAT...UNTIL**
  - ist mit Vorsicht zu verwenden, denn die Anweisung in der Schleife wird ***mindestens einmal*** durchlaufen.
  - Sie ist nur sinnvoll, wenn die Bedingung ***erst in der Schleife entsteht***
  - kann durch Schleife ohne Prüfung realisiert werden (sollte man aber nicht!)

```
REPEAT
  SIO.PutText("Zahl zwischen 1 und 5! ");
  SIO.Nl();
  a := SIO.GetInt();
UNTIL (a >=1 AND a <=5);

LOOP
 SIO.PutText("Zahl zwischen 1 und 5! ");
 SIO.Nl();
 a := SIO.GetInt();
 IF (a >=1 AND a <=5) THEN EXIT END;
END;
```

Was haben wir gelernt!

- **Auswahlanweisungen**
  - Einwegselektion IF-THEN
  - Zweiwegselektion IF-THEN-ELSE
  - Mehrwegselektion CASE
- **Wiederholungsanweisungen**
  - WHILE
  - FOR
  - REPEAT-UNTIL
  - LOOP-EXIT

- **Einsatz der Wiederholungsanweisungen**
  - Die Wahl der geeigneten Ausdrucksmittel hängt vom jeweiligen Konstruktionsproblem ab.
  - Implementationsgesichtspunkte sind ggf. zu berücksichtigen.
  - Softwaretechnische Überlegungen wie Verständlichkeit und Sicherheit spielen bei der Verwendung eine zentrale Rolle.

Datentypen I

- **Klassifikation von Typen**
- **Einfache benutzerdefinierte Datentypen**
  - Aufzählungstyp
  - Unterbereichstyp
- **Zusammengesetzte benutzerdefinierte Datentypen**
  - ARRAY-Type
  - RECORD-Type
  - SET-Type

Datentypen - Grundidee

- **Software dient zur Verarbeitung von *Anwendungsdaten*.**
  - Frage: Wie gut passen die verfügbaren Datentypen der verwendeten Programmiersprache zu den zu modellierenden Größen des ***Anwendungsbereichs***.
- **Daraus resultiert die Anforderung**
  - Programmiersprachen müssen einen ***sinnvollen*** Satz an Datentypen anbieten.
- **Ziel:**
  - Auf der Basis vorgegebener Datentypen sollen anwendungsbezogene Datenstrukturen konstruiert werden können
- **Zwei Lösungsansätze:**
  - Eine große Vielfalt von ***vordeklarierten*** Datentypen (wie in PL/I) soll möglichst viele Anwendungsfälle abdecken.
  - Ein kleiner Satz von elementaren Typen und flexiblen ***Konstruktions-mechanismen*** (wie in Algol 68) soll die anwendungs-bezogene Definition neuer Datentypen erlauben.

Bedeutung von Typen in imp. Sprachen

- **In imperativen Programmiersprachen hat ein Typ folgende Bedeutung:**
  - Festlegung des gültigen ***Wertebereichs*** für Variablen,
  - damit verbunden die ***Kardinalität*** (Anzahl der verschiedenen Werte),
  - Festlegen der ***Operationen***, die auf dem Wertebereich zulässig sind.
  - Statische ***Prüfung*** der Zulässigkeit von Operationen:
    - Zuweisung,
    - Parameterübergabe,
- **Festlegung von Maschineneigenschaften**
  - z.B. Reservierung von Speicherbedarf

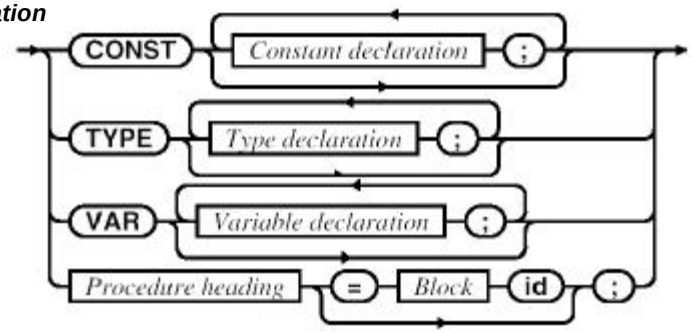
Erinnerung: Deklaration

- **In imperativen Programmiersprachen ist die explizite Deklaration von Bezeichnern notwendig.**
- **Grundidee der Deklaration:**
  - Eine Deklaration verbindet einen ***Bezeichner*** an Eigenschaften eines Programmobjekts, z.B. Typ und Sichtbarkeit.
  - Während sich z.B. bei Variablen der Wert des Objektes verändern kann, bleiben seine deklarierten Eigenschaften für die Lebensdauer erhalten.
- **Typdeklaration:**
  - Um das vorhandene ***Typkonzept*** erweitern zu können, sind in vielen imperativen Sprachen Typen selbst ***Programmobjekte***, die deklariert werden müssen.
  - Um Typen deklarieren zu können, benötigt man ***Typkonstruktoren***

Deklaration (erweitert)

- **Die Syntax für Deklarationen wird erweitert, damit *benutzerdefinierte* Typen deklariert werden können.**

Declaration



Einteilung von Typen - 1

- **In imperativen Programmiersprachen unterscheidet man einfache und zusammengesetzte Datentypen:**
  - ***Einfache Datentypen*** erlauben ***keinen*** Zugriff auf ihre innere Struktur. Ihre Werte können unmittelbar notiert werden. Die in einer Programmiersprache vorgegebenen einfachen Datentypen heißen elementar.
  - **Zusammengesetzte Datentypen** sind aus anderen Datentypen aufgebaut. Auf ihre einzelnen Elemente kann zugegriffen werden. Letztlich sind sie auf einfache Datentypen zurückführbar.
- **Vorgegebene und benutzerdefinierte Datentypen:**
  - **Vorgegebene Datentypen** haben einen vordeklarierten Namen und können unmittelbar zur Deklaration von Variablen verwendet werden.
  - **Benutzerdefinierte** Datentypen haben einen selbst definierten Namen und müssen deklariert werden. Sie werden mit Hilfe bereits deklarierter vorgegebener oder benutzerdefinierter Datentypen (ihren Basistypen) gebildet.

Einteilung von Typen - 1

- **Statische Datentypen**
  - Größe der Typobjekte ist von vornherein ***bekannt***
  - ***Statische*** Typkonstruktoren
    - ARRAY
    - RECORD
    - SET
- **Dynamische Datentypen**
  - Größe ist während der Laufzeit ***veränderbar***
  - ***Dynamischer*** Typkonstruktor
    - Zeiger
    - Pointer

Typen in Modula-3

- **Modula-3 kennt *vordefinierte einfache* Typen :**
  - SHORTINT, INTEGER, LONGINT, REAL, LONGREAL, BOOLEAN, SET, CHAR.
- **Modula-3 unterstützt wie viele imperative Sprachen *benutzerdefinierte* Datentypen.**
- **Häufig verwenden wir in der Definition die *Typkonstruktoren* für die zusammengesetzten Datentypen**
  - Array, Record, Set.
- **Eine wesentliche Erweiterung bringt der *Zeigertyp* (Pointer).**
  - Er ermöglicht den Aufbau ***dynamischer*** Datenstrukturen.

Benutzerdefinierte einfache Typen

- **Modula-3 erlaubt,**
  - benutzerdefinierte einfache Typen unter Verwendung einen vorgegebenen elementaren Typs zu deklarieren.
  - `TYPE Zeit = REAL;`
    `Alter = INTEGER;`
- **Unterbereichstyp (subrange type)**

Basistyp muß ein Ordinaltyp sein

  - benutzerdefinierte einfache Typen können als ***Einschränkung*** des Wertebereichs eines elementaren Typs deklariert werden
  - `TYPE Index = [1..10];`
    `Alter = [1 .. 120];`
- **Aufzählungstyp (emumeration type)**

Ordinaltyp

  - benutzerdefinierte einfache Typen können durch ***Aufzählung*** der zulässigen Werte deklariert werden
  - `TYPE Ampelfarbe = {rot, gelb, gruen};`
    `Parteien = {SPD, CDU, FDP, PDS, Gruene}`

Operationen auf Aufzählungstypen

- **Aufzählungstypen**
  - werden häufig systemintern auf natürliche Zahlen abgebildet
  - (z.B.: rot ->1, blau -> 2, gruen -> 3).
  - Dadurch sind die Werte von Aufzählungstypen dann ***vergleichbar*** – was aber ebenfalls selten Sinn macht (rot < gruen).
- **Bezeichner der Werte von Aufzählungstypen können bei *mehreren* Typen auftreten.**
  - `TYPE Ampelfarbe = {rot, gelb, gruen};`
  - `TYPE Parteifarbe = {rot, gelb, gruen, schwarz};`

```
VAR a : Ampelfarbe;
    p : Parteifarbe;
a := Ampelfarbe.gruen;
p := Parteifarbe.gruen
```

  - Gilt nicht für viele andere imperative Sprachen!!

Operationen auf Unterbereichstypen

- **Regel:**
  - Für einen Unterbreichstyp sind alle die Operationen definiert, die auch für seinen ***Basistyp*** definiert sind.
- **Es ist häufig unsinnig,**
  - ***arithmethische*** Operationen unmittelbar auf Unterbereichstypen anzuwenden, auch wenn dies die Sprache zuläßt (z.B. Addition zweier Jahreszahlen).
  -
```
TYPE AeraKohl = [1982 .. 1998];
VAR wahljahr1, wahljahr2, jahr : AeraKohl;

wahljahr1 := 1982; wahljahr2 := 1986;
jahr := wahljahr1 + wahljahr2;
```
    Laufzeitfehler
  - Vorsicht bei arithmetischen Operationen auf Unterbereichstypen, bei denen der Wertebereich überschritten wird – dies führt meist zu ***Laufzeitfehlern***.

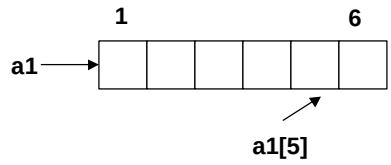
Merkmale benutzerdefinierter Typen

- **Benutzerdefinierte Typen**
  - erlauben die Vergabe ***anwendungsbezogener*** Namen,
    - z.B. `Zeit` statt REAL,
  - sind ein wesentlicher ***Abstraktionsmechanismus***, da sie die programmiersprachliche Realisierung von Datenstrukturen ***verbergen*** können,
    - später werden wir das Konzept der Abstrakten Datentypen kennenlernen
  - sind "***Baumuster***" für die Erzeugung anwendungsbezogener Datenstrukturen,
    - z.B. Struktur
```
Zugverbindung =  Abfahrt: Zeit;
                 Ankunft: Zeit;
```
  - liefern die "***Sprachelemente***" für die anwendungsbezogene Modellierung von Softwaresystemen.

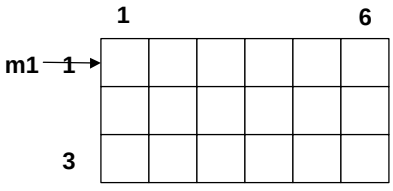
Array-Typen

- **Definition:**
  - Nach Informatik-Duden: Feld (engl. array): Aneinanderreihung von ***gleichartigen Elementen***, wobei auf die Komponenten mit Hilfe eines ***Index*** zugegriffen wird.
- **Eigenschaften von Arrays (Felder) in Modula-3:**
  - Die Anzahl der Elemente ist ***fest*** und heißt ***Länge*** des Array.
  - Der ***Name*** einer Array-Variablen bezeichnet das gesamte Array.
  - Ein einzelnes Array-Element wird durch einen ***Index*** bzw. mehrere Indizes (im Fall mehrdimensionaler Arrays) identifiziert.
  - Zur Indizierung kann jeder ***Ordinaltyp*** verwendet werden.
    - INTEGER, CARDINAL, CHAR, BOOLEAN, Aufzählungs- und Unterbereichstypen
  - Dem Array als Datenstruktur entspricht die ***FOR-Anweisung*** als Kontrollstruktur.
  - Bei mehrfach indizierten Arrays gibt es entsprechend ***geschachtelte*** Schleifen.

Beispiel: Array




```
TYPE Index = [1 .. 6];
     Vector = ARRAY Index OF INTEGER;

VAR a1 : Vector
```




```
TYPE Spalte  = [1 .. 6];
     Reihe   = [1 .. 3];

TYPE
  Matrix = ARRAY Spalte, Reihe
           OF INTEGER;

VAR m1 : Matrix;
```

mehrdimensionales Array

Arrays und FOR-Anweisung

- **Beispiel:**
  - Initialisieren eines zweidimensionalen Arrays

```
TYPE Spalte  = [1 .. 6];
     Reihe = [1 .. 3];

TYPE Matrix = ARRAY Spalte, Reihe OF INTEGER;

PROCEDURE Initialisieren (VAR m : Matrix) =
BEGIN
  FOR i := FIRST(Spalte) TO LAST(Spalte) DO
    FOR j := FIRST(Reihe) TO LAST(Reihe) DO
      m [i,j] := 0;
    END;
  END;
END Initialisieren;
```


Arrays als zusammengesetzte Typen

- **Betrachten wir Arrays als zusammengesetzte Typen, dann stellen wir fest:**
  - Der ***Typkonstruktor***, der in Deklarationen benutzt wird, ist in Modula-3 (vereinfacht):

    ```
    <ArrayTyp> = ARRAY Index OF Basistyp;
    ```

  - Als ***Selektor*** für ein einzelnes Element wird die Indexangabe verwendet:

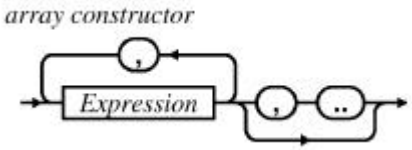
    ```
    val := a[3] (* Wert des 3. Elements von a *)
    ```

  - Üblicherweise wird die Indexangabe auch für die ***selektive Zuweisung*** verwendet:

    ```
    a[4]:= 42 (* 4. Elements von a wird 42 *)
    ```

ARRAY-Konstruktor

- **Mithilfe eines sog. *ARRAY-Konstruktors* können ARRAY-Objekte erzeugt und initialisiert werden:**
  - VAR x := Array_Type { e1, .., en}
    e1 bis en sind Ausdrücke; ihr Wert wird den Feldelementen initial zugewiesen




```
TYPE Index = [1 .. 6];
     Vector = ARRAY Index OF INTEGER;
VAR a1 := Vector {0, 0, 0, 0, 0, 0};
    a2 := Vector {0, ..}
```

```
TYPE Spalte = [1 .. 6];
     Reihe = [1 .. 3];

TYPE Matrix = ARRAY Spalte , Reihe OF INTEGER;
VAR m1 := Matrix {ARRAY Reihe OF INTEGER {0, ..}, ..};
```


Operationen auf ARRAYs - 1

- **Zuweisung**
  - zwei ARRAY-Objekte sind ***zuweisungskompatibel***, wenn sie
    - den gleichen ***Basistyp*** und die
    - gleiche ***Gestalt*** haben (gleiche Anzahl Elemente in jeder Dimension)
- **Vergleich**
  - zuweisungskompatible Arrays können auf ***Gleichheit*** und ***Ungleichheit*** geprüft werden.

```
TYPE Index = [1 .. 6];
     Vector = ARRAY Index OF INTEGER;
CONST A = ARRAY [11 .. 16] OF INTEGER {1,2,3,4,5,6};
VAR v : Vector;
…
  v := A;

  IF v = A THEN
```

Beispiel: Array - 1

```
MODULE StundenPlan EXPORTS Main;
IMPORT SIO;

TYPE
    Tage      = {Montag, Dienstag, Mittwoch, Donnerstag, Freitag};
    Stunden   = [7..20];
    Vormittag = [8..12];
    Fächer    = {Keine, Englisch, Software_1, Mathematik};
    Plan      = ARRAY Tage, Stunden OF Fächer;

CONST
   TagNamen  = ARRAY Tage OF TEXT   {"Montag", "Dienstag","Mittwoch",
                                     "Donnerstag", "Freitag"};
   FachNamen = ARRAY Fächer OF TEXT {"Keine","Englisch","Software_1","Mathematik"};

VAR stundenPlan : Plan;          (*Speichert den StundenPlan*)
```

aus Weich, Seite 156.

Beispiel: Array - 2

```
BEGIN
  FOR tag:= FIRST(Tage) TO LAST(Tage) DO
    FOR stunde:= FIRST(Stunden) TO LAST(Stunden)  Do
      stundenPlan[tag, stunde]:= Fächer.Keine;    (*Initialisierung auf Keine*)
    END; (*FOR stunde*)
  END; (*FOR tag*)

  FOR stunde:= 8 TO 18 DO    (*Fast den ganzen Montag Englisch*)
    stundenPlan[Tage.Montag, stunde]:= Fächer.Englisch;
  END; (*FOR stunde*)

  FOR tag:= Tage.Dienstag TO Tage.Freitag DO
    stundenPlan[tag, 10]:= Fächer.Software_1;
  END; (*FOR tag*)

  stundenPlan[Tage.Dienstag, 8]:= Fächer.Mathematik;
  stundenPlan[Tage.Freitag, 9]:= Fächer.Mathematik;

  FOR tag:= FIRST(Tage) TO LAST(TAGE) DO
    SIO.PutText(TagNamen[tag]& "\ n");
    FOR stunde:= FIRST(Vormittag) TO LAST(Vormittag) DO
      SIO.PutInt(stunde);
      SIO.PutText(":" & FachNamen[stundenPlan[tag, stunde]]);
    END; (*FOR stunde*)
    SIO.NL();
  END; (*FOR tag*)
END StundenPlan.
```

RECORD-Typen

- **Definitionen:**
  - Nach Informatik-Duden:
    - Record *(Verbund, Datensatz)*: ***Zusammenfassung*** von mehreren Datentypen zu einem Datentyp. Der neue Wertebereich ist das ***kartesische Produkt*** der Wertebereiche der einzelnen Datentypen, wobei die Anordnung keine Rolle spielt.
  - Nach Sebesta:
    - A record is a ***possibly heterogeneous*** aggregate of data elements in which the individual elements are identified by **names**.
  - Nach Ludewig:
    - Records (Verbunde) sind ***heterogene*** kartesische Produkte und dienen zur Darstellung ***inhomogener***, aber ***zusammengehöriger*** Informationen. Typische Beispiele sind
      - Personendaten (Name, Adresse, Jahrgang, Geschlecht)
      - Meßwerte (Zeit, Gerät, Wert)
      - Strings (tatsächliche Länge, Inhalt)

RECORD in Modula-3

- **Record in Modula-3:**
  - Datentyp, der eine Sammlung von Elementen auch ***verschiedenen*** Typs (Elementtyp) repräsentiert.
  - Der ***Name*** einer Record-Variablen bezeichnet den gesamten Record.
  - Ein einzelnes Record-Element heißt auch ***Komponente*** oder ***Feld*** (record field) und wird durch einen ***Namen*** (field identifier) bezeichnet.
  - Von außen wird ein Feld über seinen Bezeichner mit der sog. ***Punktnotation*** (dot notation) angesprochen:
  - `<RecordName>"."<FeldName>`

Anrede	Vorname	Name	PersNr
Herr	Franz	Mustermann	4711
Dr.	Josef	Wanninger	4712
Frau	Susanne	Mitternacht	4713

Ein Record

Beispiel: RECORD - 1

```
MODULE RecordDemo EXPORTS Main;

TYPE PersNr = [4700 .. 9999];
TYPE Anrede = {Frau, Herr, Dr};
TYPE Name  = TEXT;

TYPE Person = RECORD
                anrede   : Anrede;
                vorname  : Name;
                nachname : Name;
                persnr   : PersNr;
              END;

VAR person1 : Person;
BEGIN
  person1.anrede   := Anrede.Dr ;
  person1.vorname  := "Josef";
  person1.nachname := "Wanninger";
  person1.persnr   := 4712;
END RecordDemo.
```

Faßt unterschiedliche Typen zu einer gemeinsamen Struktur zusammen.

Beispiel: RECORD - 2

```
TYPE PersNr = [4700 .. 9999];
TYPE Anrede = {Frau, Herr, Dr};

TYPE Name   = RECORD
                vorname  : TEXT;
                nachname : TEXT;
              END;
TYPE Person = RECORD
                anrede : Anrede;
                name   : Name;
                persnr : Pers;
              END;
VAR person1 : Person;

person1.anrede        := Anrede.Dr ;
person1.name.vorname  := "Josef";
person1.name.nachname := "Wanninger";
person1.persnr        := 4712;
```

- **Bemerkung:**
  - Ziel ist, Typen so zu konstruieren, daß sie möglichst sinnvoll ***aufeinander*** aufbauen.
  - Vorteile:
    - erleichterte Modifikation
    - Wiederverwendbarkeit
    - bessere Lesbarkeit
    - Begriffe der Anwendung können verwendet werden

Records als zusammengesetzte Typen

- **Betrachten wir Records als zusammengesetzte Typen, dann stellen wir fest:**
  - Der ***Typkonstruktor***, der in Deklarationen benutzt wird, ist in Modula-3 (vereinfacht):

    RECORD <Feldname> : <Basistyp> {;<Feldname> : <Basistyp>} END

  - Der ***Selektor*** für ein Feld eines Records, der bei der Verwendung benutzt wird, ist in Modula-3:

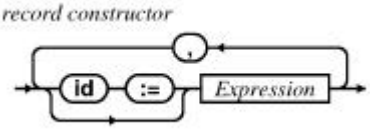
    <RecordBezeichner> . <FeldName>

  - Eine rekursive Typdeklaration eines Records ist ***nicht möglich***:

```
Liste = RECORD Listenkopf : CHAR;
               Listenrest : Liste;
        END (* geht nicht *)
```


RECORD-Konstruktor

- **Mit dem *RECORD-Konstruktor* werden initialisierte RECORD-Objekte erzeugt:**
  - VAR x := Record_Type { Bindings }
    Bindings: analog zur Bindung der aktuellen Parameter an formale Parameter beim Prozeduraufruf




```
TYPE Name   = RECORD
                 vorname  : TEXT;
                 nachname : TEXT;
              END;
VAR n1 := Name { vorname := "Josef",  nachname := "Maier"};

TYPE Person = RECORD
                anrede : Anrede;
                name   : Name;
                persnr : PersNr;
              END;

VAR p1 := Person {anrede := Anrede.Herr,
                  name   := Name { vorname := "Kai",  nachname := "Blau"},
                  persnr := 4700 };
```

Angabe der Werte per Name

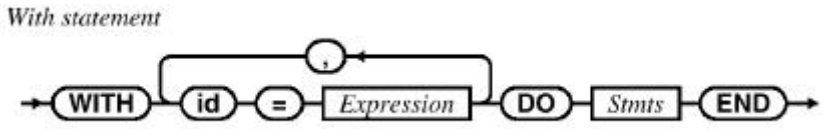
Operationen auf RECORDs - 1

- **Zuweisung**
  - zwei RECORD-Objekte sind ***zuweisungskompatibel***, wenn
    - alle Felder den gleichen ***Namen*** und den gleichen Typ haben
    - alle Felder in der gleichen ***Reihenfolge*** deklariert sind
- **Vergleich**
  - zuweisungskompatible Arrays können auf ***Gleichheit*** und ***Ungleichheit*** geprüft werden.

```
TYPE Name1  = RECORD
                vorname  : TEXT;
                nachname : TEXT;
              END;
TYPE Name2  = RECORD
                nachname : TEXT;
                vorname  : TEXT;
              END;
VAR n1 : Name1; n2 : Name2;
n1 := n2  nicht zuweisungskompatibel
```


Die WITH-Anweisung

- **WITH-Anweisung**
  - dient dazu, komplexe Selektoren, die mehrmals verwendet werden müssen, mit einem ***ALIAS-Namen*** zu versehen.
  - Code wird ***kompakter***, lesbarer
- **Syntax:**



- **Semantik:**
  - der im Binding eingeführte Bezeichner ist im Block bis zum END gültig
  - der eingeführte Bezeichner wird als "***Abkürzung***" verwendet

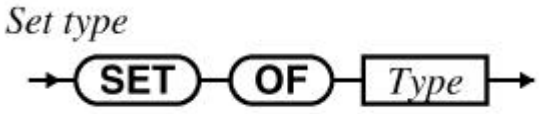
Beispiel WITH-Anweisung

```
TYPE Name   = RECORD                    TYPE Person = RECORD
                vorname  : TEXT;                anrede : Anrede;
                nachname : TEXT;                name   : Name;
              END;                              persnr : PersNr;
                                              END;

  VAR bundeskanzler : Person;

  bundeskanzler.anrede        := Anrede.Herr;
  bundeskanzler.name.vorname  := "Gerhard";
  bundeskanzler.name.nachname := "Schroeder";
  bundeskanzler.persnr        := 4700;
```

```
WITH bk = bundeskanzler DO
  bk.anrede := Anrede.Herr;
  WITH bkn = bk.name DO
    bkn.vorname := "Gerhard";
    bkn.nachname := "Schroeder";
  END;
  bk.persnr := 4700;
END
```


Mengen

- **Modula-3 bietet einen eigenen vordefinierten Mengentyp**
- **Syntax: (Typkonstruktor)**



- **Bemerkungen:**
  - Mengen sind ***ungeordnete*** Sammlungen von Elementen
  - der Elementtyp muß ein ***Ordinaltyp*** sein!
  - Elemente einer Menge können **nicht** indiziert werden
  - Wertebereich eines Mengentyps ist die ***Potenzmenge***
    - Menge aller Mengen über dem Elementtyp
    - Bspl: [rot, gruen] {} {rot} {gruen} {rot, gruen}
  - Aus Effizienzgründen sollen Mengen nur über Elementmengen mit ***kleiner Kardinalität*** gebildet werden.

Beispiel : Mengendeklaration

```
TYPE Lottozahl = [1 .. 49];

TYPE Ziehung = SET OF Lottozahl;

CONST Leer := Ziehung {};

VAR   z1   := Ziehung {1 .. 7};
      z2   := Ziehung {4,7,34,20,44,23};
```

Mengenkonstruktor

- **Operationen:**
  - Zuweisung
    - zuweisungskompatibel: ***Elementtypen*** sind gleich
  - Vereinigung, Differenz, Durchschnitt, Symmetrische Differenz
  - Gleichheit, Ungleichheit, Teilmenge, … , Enthalten

Beispiel: Buchstaben zählen - 1

```
MODULE BuchstabenOrg EXPORTS Main;
IMPORT SIO, Text;

TYPE Buchstabe = ['A' .. 'Z'];
     BuchstabenMenge = SET OF Buchstabe;

CONST Alle = BuchstabenMenge {'A' .. 'Z'};
VAR einmal, mehrmals, nie := BuchstabenMenge {};
    eingabe : TEXT;
    z : CHAR;
BEGIN
  eingabe := SIO.GetLine();

  FOR i := 0 TO Text.Length(eingabe) - 1 DO
    z := Text.GetChar(eingabe, i);
    IF z IN Alle THEN
      IF z IN einmal THEN
        mehrmals := mehrmals + BuchstabenMenge{z};
      ELSE
        einmal := einmal + BuchstabenMenge{z};
      END;
    END;
  END;
  nie := Alle - einmal;
  einmal := einmal - mehrmals;
```

Set-Konstruktor

Mengen-vereinigung

Mengen-differenz

Beispiel: Buchstaben zählen - 2

```
  SIO.PutLine("NIE:");
  FOR z := 'A' TO 'Z' DO
    IF z IN nie THEN
      SIO.PutChar(z)
    END;
  END;
  SIO.Nl();

  SIO.PutLine("EINMAL:");
  FOR z := 'A' TO 'Z' DO
    IF z IN einmal THEN
      SIO.PutChar(z)
    END;
  END;
  SIO.Nl();

  SIO.PutLine("MEHRMALS:");
  FOR z := 'A' TO 'Z' DO
    IF z IN mehrmals THEN
      SIO.PutChar(z)
    END;
  END;
  SIO.Nl();

END BuchstabenOrg.
```


Verbesserung 1 des Beispiels

```
PROCEDURE GebeMengeAus (m : BuchstabenMenge) =
BEGIN
  FOR z := FIRST(Buchstabe) TO LAST(Buchstabe) DO
    IF z IN m THEN
      SIO.PutChar(z)
    END;
  END;
END Ausgabe;

BEGIN (* Buchstaben1 *)
  eingabe := SIO.GetLine();
  FOR i := 0 TO Text.Length(eingabe) - 1 DO
    z := Text.GetChar(eingabe, i);
    IF z IN Alle THEN
      IF z IN einmal THEN
        mehrmals := mehrmals + BuchstabenMenge{z};
      ELSE
        einmal := einmal + BuchstabenMenge{z};
      END;
    END;
  END;
  nie := Alle - einmal;
  einmal := einmal - mehrmals;
  SIO.PutLine("NIE:");      GebeMengeAus(nie); SIO.Nl();
  SIO.PutLine("EINMAL:");   GebeMengeAus(einmal); SIO.Nl();
  SIO.PutLine("MEHRMALS:"); GebeMengeAus(mehrmals); SIO.Nl();
END Buchstaben1.
```

Verwenden einer Prozedur für die Ausgabe von Mengen

Verbesserung 2 des Beispiels - 1

```
TYPE Vorkommen = RECORD
                   einmal, mehrmals, nie := BuchstabenMenge {};
                 END;

PROCEDURE Vorkommenzaehlen (t : TEXT) : Vorkommen =
  CONST Alle     := BuchstabenMenge {'A' .. 'Z'};
  VAR   resultat : Vorkommen; z : CHAR; vorgekommen : BuchstabenMenge;
BEGIN
  WITH r = resultat DO
    FOR i := 0 TO Text.Length(t) - 1 DO
      z := Text.GetChar (t, i);
      IF z IN Alle THEN
        IF z IN vorgekommen THEN
          r.mehrmals := r.mehrmals + BuchstabenMenge{z};
        ELSE
          vorgekommen := vorgekommen + BuchstabenMenge{z};
        END;
      END;
    END;
    r.nie := Alle - vorgekommen ;
    r.einmal := vorgekommen - r.mehrmals;
  END;
  RETURN resultat;
END Vorkommenzaehlen;
```

Wäre ein Array eine Alternative?

Bezeichner werden nur für einen Zweck eingesetzt!

Verbesserung 2 des Beispiels - 2

```
MODULE Buchstaben1 EXPORTS Main;
…
VAR vork : Vorkommen;

PROCEDURE Vorkommenzaehlen (t : TEXT) : Vorkommen =
...


BEGIN

  vork := Vorkommenzaehlen(SIO.GetLine());

  SIO.PutLine("NIE:");       GebeMengeAus(vork.nie); SIO.Nl();
  SIO.PutLine("EINMAL:");    GebeMengeAus(vork.einmal); SIO.Nl();
  SIO.PutLine("MEHRMALS:"); GebeMengeAus(vork.mehrmals); SIO.Nl();

END Buchstaben1.
```

Verwenden die Ausgabeoperation für Mengen

Verbesserung 2 des Beispiels - 3

```
PROCEDURE Ausgabe (v: Vorkommen) =
CONST NIE      = "     NIE : ";
      EINMAL   = "  EINMAL : ";
      MEHRMALS = "MEHRMALS : ";
BEGIN
  SIO.PutText(NIE);      GebeMengeAus(v.nie); SIO.Nl();
  SIO.PutText(EINMAL);   GebeMengeAus(v.einmal); SIO.Nl();
  SIO.PutText(MEHRMALS); GebeMengeAus(v.mehrmals); SIO.Nl();
END Ausgabe;

PROCEDURE Vorkommenzaehlen (t : TEXT) : Vorkommen =
...

BEGIN (*Buchstaben2 *)

  Ausgabe(Vorkommenzaehlen(SIO.GetLine()));

END Buchstaben2.
```


Verbesserung 2 des Beispiels - 4

```
PROCEDURE LiesEingabeBis (stop : CHAR) : TEXT =
VAR eingabe : TEXT := "";
    zeichen : CHAR;
BEGIN
  zeichen := SIO.GetChar();
  WHILE zeichen # stop DO
    eingabe := eingabe & Text.FromChar(zeichen);
    zeichen := SIO.GetChar();
  END;
  RETURN eingabe;
END LiesEingabeBis;

…

BEGIN (*Buchstaben3*)

  Ausgabe(Vorkommenzaehlen(LiesEingabeBis(':')));

END Buchstaben3.
```

Arrays, Records, Mengen

- **Vergleich der Typkonstruktoren für**
  - statische, zusammengesetzte Typen

Aspekt	Array	Record	Menge
Größe	fest (!)	fest	fest
Element-typen	homogen	heterogen	homogen, Ordinaltyp
Zugriff	dynamisch indiziert	statisch	kein direkter Zugriff
Ordnung	statisch festgelegt Index ist geordnet	statisch festgelegt	ungeordnet

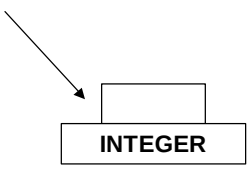
Datentypen II

- **Dynamische Datentypen**
  - Zeigertypen
  - dynamische Datenstrukturen
- **Anwendungsbeispiele**
  - lineare Liste
  - sortierte Liste
- **Prozedurtyp**

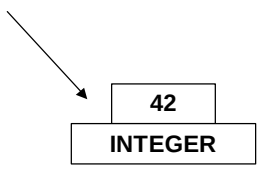
Bezeichner und Objekte

- **Wir haben als elementares imperatives Konzept die *Variable* kennengelernt.**
- **Kennzeichen sind:**
  - Ein Name dient als ***Bezeichner***, der mit einem ***getypten Wert*** verbunden werden kann.
- **Bei den bisher betrachteten Datentypen wurde der Zusammenhang von Bezeichner und Wert implizit in der Sprache durch den *Zuweisungsmechanismus* hergestellt.**

```
VAR Antwort: INTEGER;
```

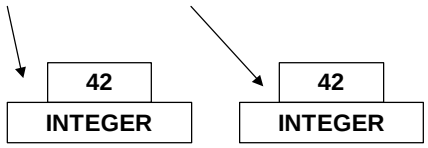



```
Antwort := 42;
```



Wertsemantik

- **Die bisher betrachteten Variablen und die Zuweisung basieren auf der *Wertsemantik*:**
  - Eine Variable hat einen ***definierten*** oder ***undefinierten*** Wert.
  - Bei der ***Zuweisung*** wird der Wert des Ausdrucks der rechten Seite an die Variable der linken Seite zugewiesen.
  - Eine ***Identität*** von Objekten wird nicht hergestellt.
  -
```
VAR Antwort1, Antwort2: INTEGER;
…
Antwort1 := 42; (* 1 *)
Antwort2 := Antwort1; (* 2 *)
Antwort1 := 24; (* 3 *)

Antwort1 := Antwort2;
```

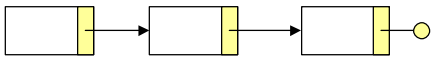


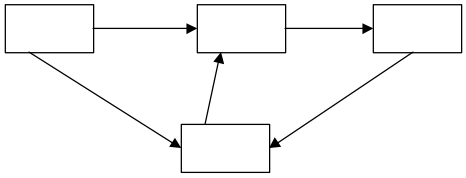
Statische Datentypen

- **Kennzeichnend für imperative Sprachen ist, daß *Wertsemantik* und *statische Datentypen* zusammenfallen:**
  - Die Zuordnung von Bezeichner und Wert geschieht über die Zuweisung.
  - Die Variablen eines statischen Typs ***behalten ihre Struktur*** während ihrer Lebensdauer bei.
  - Der ***Speicherbedarf*** einer statisch getypten Variablen wird bei der Übersetzung anhand der Deklaration ***festgelegt*** und bei der ersten Verwendung implizit zugewiesen.
- **Speicherbereiche für statisch getypte Variablen wird im sog. *Kellerspeicher* reserviert**
  - zusammenhängender Bereich pro Objekt
  - Bereich wird ***angelegt***, beim Eintritt in entsprechende Prozedur
  - Bereich wird ***freigegeben***, beim Verlassen der Prozedur

Dynamische Datentypen

- **Probleme mit statischen Datentypen**
  - Es werden Datenstrukturen benötigt, deren ***Größen starken Schwankungen*** unterworfen sind.
  - Es sollen ***komplizierte*** Datenstrukturen gebildet werden
    - Listen
    - Bäume
    - Graphen



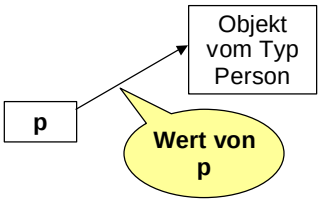


- **Lösung**
  - Dynamische Datentypen
  - oder Dynamische Variable
  - oder Zeigertypen

Zeigertyp - Referenztyp

- **Zeigertyp**
  - um einen Zeigertyp zu deklarieren, bietet Modula-3 den ***Typkonstruktor***
  - **`<ZeigerTyp> = REF <ReferenzierterTyp>`** an
  - damit kann aus ***jedem Typ*** ein Zeigertyp abgeleitet werden

```
TYPE Person = RECORD
                anrede : Anrede;
                name   : Name;
                persnr : PersNr;
              END;

TYPE PersonRef = REF Person;

VAR p : PersonRef;
```



  - p kann als Werte ***Zeiger auf Objekte*** vom Typ `Person` annehmen
  - zeigt p auf kein Objekt, dann wird dies durch den Wert ***`NIL`*** angezeigt
  - `NIL` ist Objekt jedes Zeigertyps

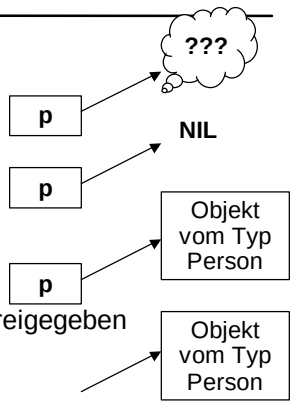
Eigenschaften dynamischer Datentypen

- **Eigenschaften von Objekten dynamischer Datentypen**
  - ***Lebensdauer*** ist nicht an die Ausführung einer Prozedur oder Moduls gebunden
  - sie werden zur Laufzeit ***explizit (vom Programmierer) erzeugt*** und eventuell auch wieder beseitigt
  - sie werden in einem speziell dafür vorgesehenen Speicherbereich angelegt (***Halde oder Heap***)
  - können in prinzipiell ***beliebiger*** Menge geschaffen werden
  - haben im Gegensatz zu den bisherigen Objekten ***keinen festen Bezeichner***
  - sie werden stattdessen über einen ***Zeiger (Pointer)*** identifiziert
  - Zeiger können im Kellerspeicher oder auf der Halde liegen
  - die referenzierten Objekte liegen ***immer*** auf der Halde

Erzeugen von Zeigerobjekten

- Beispiel:

```
PROCEDURE PROC …
  TYPE PersonRef = REF Person;
  VAR p : PersonRef;
BEGIN
  p := NIL;
  p := NEW (PersonRef);
END
```



- beim Betreten der Prozedur wird Speicher für p zur Verfügung gestellt und beim Verlassen wieder freigegeben
- durch den Aufruf von `NEW(PersonRef)`
  - wird auf der Halde ***Speicher*** für ein Objekt von Typ Person angelegt
  - die ***Adresse*** dieses Speicherplatzes wird zurückgeliefert und der Variablen p zugewiesen
  - Das Objekt selbst erhält keinen eigenen Bezeichner – man spricht von einer ***dynamischen*** oder auch von einer ***anonymen*** Variablen.
- dieser Speicher wird nicht freigegeben, wenn die Proz. verlassen wird

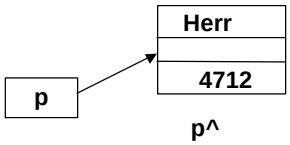
Dereferenzierung - 1

- **Eine gesetzte Referenzvariable verweist auf ein Objekt.**
- **Um das Objekt selbst zu erhalten, müssen wir dem Verweis "nachgehen".**
  - Diese Operation, bei der auf das referenzierte Objekt einer Referenzvariablen zugegriffen wird, heißt ***dereferenzieren***.
- **Dereferenzieren ist, neben dem Erzeugen der referenzierten Objekte, die**
  - zweite charakteristische Operation von Referenztypen.
  - Nur eine ***gesetzte Referenzvariable*** kann dereferenziert werden.
  - Der Versuch einer Dereferenzierung der Referenz NIL führt in den meisten Programmiersprachen zum ***Programmabbruch***.

Dereferenzierung

- **Dereferenzierung**
  - Zugriff auf ***Werte*** von Zeigerobjekten

```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

p^.persnr := 4732;

p := NEW (PersonRef);
p^.persnr := 4712;
p^.anrede := AnredeTyp.Herr;

nr := p^.persnr;
```



Laufzeitfehler: Zeigervariable nicht gesetzt

Dereferenzierungsoperator

  - In Modula-3 kann der ^-Operator entfallen,
    - wenn ein weiterer Operator folgt (z.B."[ ]", ".")
  - Das trägt ***nicht zur Klarheit*** bei !!!
  - Darum
    - Verwenden Sie ***immer*** den ^-Operator !!!

Operationen auf Referenztypen

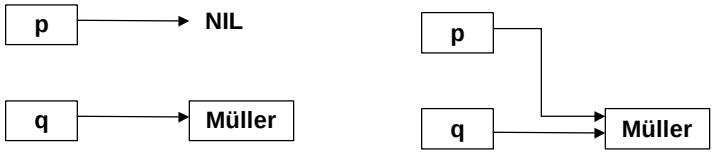
- **Zulässige Werte von Referenztypen**
  - sind Referenzen oder der Wert "***keine Referenz***" (NIL).
- **Gesetzte Referenzen haben keine externe Repräsentation.**
  - Entsprechend können sie ***nicht*** ausgegeben oder z.B. in ***Rechenoperationen*** verwendet werden.
- **Die einzigen zulässigen Operationen auf Referenzen sind:**
  - Test auf ***Gleichheit*** oder ***Ungleichheit***,
  - ***Zuweisung*** auf Variablen von kompatiblem Typ.
- **Beispiel in Modula-3:**
  - Der Effekt der Anweisung `VAR p1, p2 : PersonRef;` ist streng zu unterscheiden von der Wirkung der Anweisung

```
IF p1 = NIL THEN
   p1 := NEW (PersonRef)
END;
p2 := p1;
```

ist eine ***Referenzzuweisung***

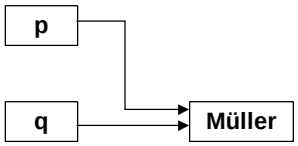
Das Alias-Problem

- **Die Zuweisung bei Referenztypen basiert auf der sog. *Referenzsemantik*:**
  - Der ***Verweis*** auf ein Objekt wird zugewiesen; nicht etwa der Wert des referenzierten Objekts.
  - Dies ist vielfach erwünscht, schafft aber auch folgendes Problem
- **Mehrere Referenzvariablen können auf dasselbe Objekt verweisen.**
  - Damit ist lokal oft nicht ***entscheidbar***, ob sich Veränderungen am Zustand eines referenzierten Objekts ergeben haben oder nicht.
  - Dies ist das ***Alias-Problem***, das bei allen Referenztypen auftritt.

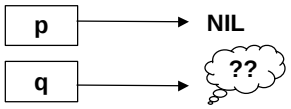


Freigeben von Zeigerobjekten

- **In der Regel müssen Zeigerobjekte vom Programmierer kontrolliert werden:**
  - Er muß sie explizit ***anlegen*** und ***freigeben***
- **Beim Freigeben können folgende Fehlersituationen auftreten**
  - Nicht mehr benötigte Zeigerobjekte werden ***nicht frei*** gegeben
    - Es wird zu viel Speicher belegt
  - Es werden Zeigerobjekte freigegeben, die noch ***benötigt*** werden
    - Problem der "***dangling pointers***"



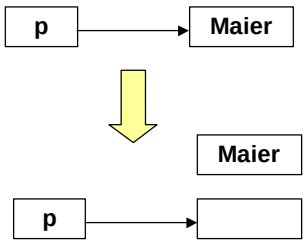

```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;
p := q;
DISPOSE(p);
q^.anrede := ...
```



Probleme mit Referenzvariablen

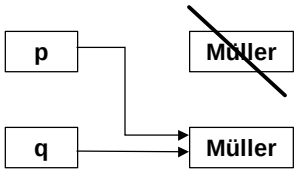
- **"Lost Object":**
  - Es kann dynamisch angelegte Objekte geben, auf die keine ***Referenzvariable*** mehr verweist.
  - Dieses Objekt kann nicht mehr erreicht werden und wird als ***Garbage*** bezeichnet.
- **Häufige Fehlersituation:**

```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

p := NEW (PersonRef);
p^.name := "Maier";
...
p := NEW (PersonRef);
```



Automatische Speicherbereinigung

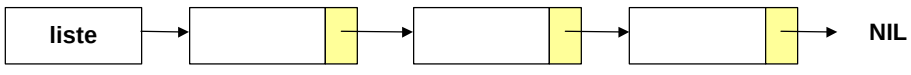
- **Einige Laufzeitumgebungen von Sprachen können Zeigerobjekte kontrollieren**
  - sie geben Zeigerobjekte, die ***nicht mehr erreicht*** werden können, automatisch frei
  - "***Garbage Collector***" (Müllsammler)



- **Diskussion Garbage Collector**
  - ***Vorteile***
    - Fehlerklasse "dangling pointer" ausgeschaltet
  - ***Nachteile***
    - Müllsammeln kostet Zeit

Dynamische Datenstrukturen

- **Ziel:**
  - Definition dynamischer Datenstrukturen, die ***einfach*** vom Programmierer erstellt, verwaltet und kontrolliert werden können
- **Referenzvariablen**
  - können auf zusammengesetzte Datenstrukturen verweisen, die selbst wieder ***Referenzvariablen*** enthalten. Wenn diese Strukturen rekursiv sind, sprechen wir von ***Verweisketten***, die den Aufbau dynamischer Datenstrukturen ermöglichen.
- **Beispiel: Einfach verkettete lineare Liste**
  - Elemente der Liste sind ***Zeigerobjekte***
  - Jedes Zeigerobjekt ist so konstruiert, das es ***selbst*** wieder auf ein Zeigerobjekt ***verweisen*** kann
  - zusätzlich enthält es die ***eigentlichen*** Informationen



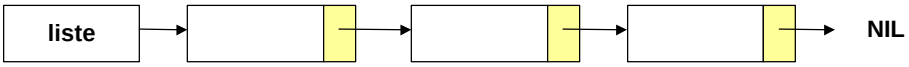
Lineare Liste

```
TYPE Jahr = [1890 .. 1998];

TYPE Person = RECORD
  name, vorname : TEXT;
  geburtsjahr : Jahr;
END;

TYPE ListenElement = REF
  RECORD
    person : Person;
    nachfolger : ListenElement;
  END;

VAR liste : ListenElement;
```

Typ, für die in der Liste verwaltete Information

Rekursives Verwenden des Typbezeichners

"Anker", um auf die Liste zugreifen zu können



Suchen in linearen Listen - 1

```
PROCEDURE Suche (liste : ListenElement;
                 was : Person): ListenElement =
VAR position : ListenElement;
    gefunden : BOOLEAN := FALSE;

BEGIN
  position := liste;
  WHILE (position # NIL AND NOT gefunden) DO
    IF was = position^.person THEN
      gefunden := TRUE;
    ELSE
      position := position^.nachfolger;
    END;
  END;
  RETURN position;
END Suche;
```

Sequentielles Suchen

Ergebnis : Zeiger auf das gefundene Element sonst NIL

Suchen in linearen Listen - 2

- **Listen sind rekursive Datenstrukturen**
  - ***Rekursive Datenstrukturen*** können *elegant* mit ***rekursiven Prozeduren*** bearbeitet werden!

```
PROCEDURE SucheRek (liste : ListenElement;
                    was : Person): ListenElement =
BEGIN
  IF liste # NIL THEN
    IF was = liste^.person THEN
        RETURN liste;
    ELSE
      RETURN SucheRek (liste^.nachfolger, was);
    END;
  ELSE
    RETURN NIL;
  END;
END SucheRek;
```

rekursiver Aufruf

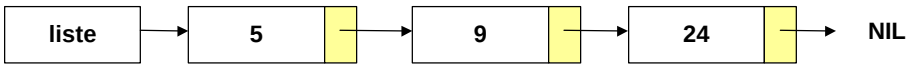
Sortierte lineare Liste

```
TYPE ListenElement = REF
  RECORD
    wert : INTEGER;
    nachfolger : ListenElement;
  END;

VAR liste : ListenElement;
```

Liste soll immer im Zustand "aufsteigend" sortiert sein

Für alle Elemente x der Liste gilt:

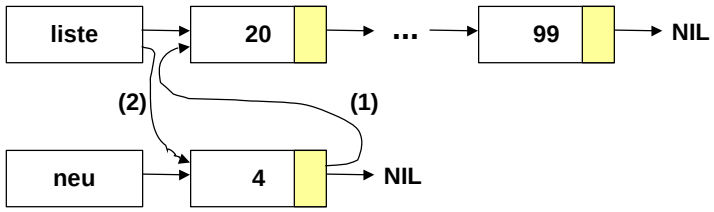
`x^.wert <= x^.nachfolger^.wert`



Einfügen in eine sortierte Liste - 1

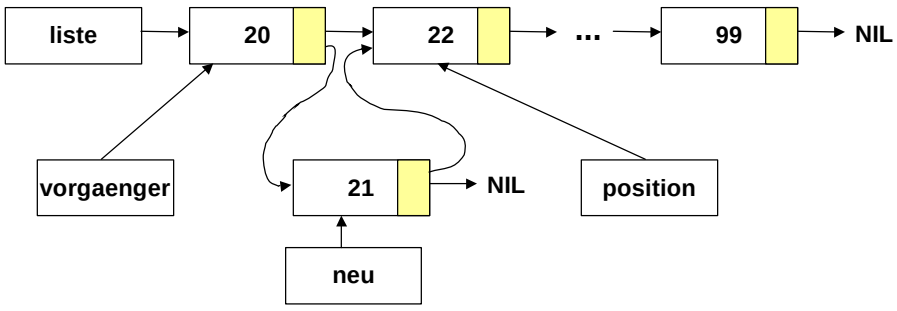
- **Fall 1: Die Liste ist leer.**

- **Fall 2: Element muß vorne eingefügt werden**



Einfügen in eine sortierte Liste - 2

- **Fall 3: Element muß sonst irgendwo eingefügt werden.**
  - Es muß nach der ***Einfügestelle*** gesucht werden
  - Damit man auf die Elemente vor und hinter der Einfügestelle zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das Element hinten angefügt werden muß !!

Einfügen: Lösung A -1

```
PROCEDURE Einfuegen (VAR liste : ListenElement; w : INTEGER) =
VAR neu, einfuegestelle : ListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN       (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);      (* neu vorne einhaengen *)
  ELSE
    einfuegestelle := SucheEinfuegestelle(liste, w);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```


Einfügen: Lösung A -2

```
PROCEDURE ErzeugeNeuesListenelement(w : INTEGER): ListenElement =
VAR elem : ListenElement;
BEGIN
  elem := NEW(ListenElement);
  elem^.wert := w;
  elem^.nachfolger := NIL;
  RETURN elem
END ErzeugeNeuesListenelement;
```

```
PROCEDURE EinfuegenVorne(VAR liste : ListenElement;
                         VAR neu   : ListenElement)=
BEGIN
  neu^.nachfolger := liste;
  liste := neu;
END EinfuegenVorne;
```

Einfügen: Lösung A -3

```
PROCEDURE SucheEinfuegestelle(liste : ListenElement;
                                  w : INTEGER): ListenElement =
VAR stelle, pos: ListenElement;
BEGIN
  pos := liste;
  stelle := liste;
  WHILE (pos # NIL) AND (pos^.wert <= w) DO
    stelle := pos;
    pos := pos^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```

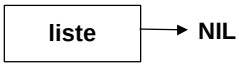
```
PROCEDURE FuegeAnEinfuegestelleEin(VAR neu   : ListenElement;
                                   VAR stelle: ListenElement)=
BEGIN
  neu^.nachfolger := stelle^.nachfolger;
  stelle^.nachfolger := neu;
END FuegeAnEinfuegestelleEin;
```


Einfügen: Lösung B

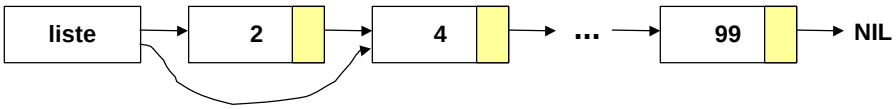
```
PROCEDURE Einfuegen (VAR liste : ListenElement; w : INTEGER) =
VAR neu, position, vorgaenger : ListenElement;
BEGIN
  neu := NEW(ListenElement);
  neu^.wert := w;
  neu^.nachfolger := NIL;

  IF liste = NIL THEN             (* liste ist leer *)
    liste := neu;
  ELSIF (w <= liste^.wert) THEN   (* w ist kleinest Element*)
    neu^.nachfolger := liste;     (* w vorne einhaengen *)
    liste := neu;
  ELSE                            (* Einfuegestelle suchen *)
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert <= w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                          (* vorgaenger = Einfuegestelle *)
    neu^.nachfolger := position;  (* w einhaengen *)
    vorgaenger^.nachfolger := neu;
  END;
END Einfuegen;
```

Löschen in einer sortierten Liste - 1

- **Fall 1: Die Liste ist leer.**

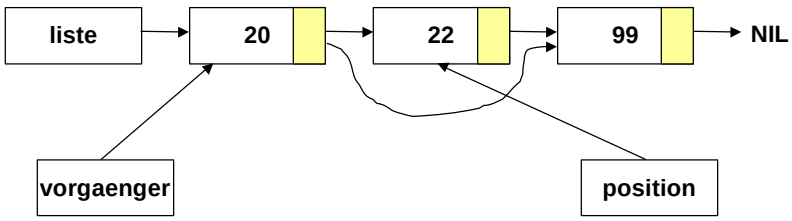


- **Fall 2: Das erste Element muß gelöscht werden**



Löschen in einer sortierten Liste - 2

- **Fall 3: Element muß sonst irgendwo gelöscht werden.**
  - Damit man auf die Elemente vor und hinter dem zu löschenden Element zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das letzte Element gelöscht werden muß !!

Löschen: Lösung A

```
PROCEDURE Loeschen (VAR liste    : ListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : ListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert # w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                          (* position = NIL v position^.wert = w *)
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        liste := position^.nachfolger;
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```


Löschen: Lösung B -1

```
PROCEDURE Loeschen (VAR liste    : ListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : ListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position);
    gefunden := (position # NIL);  (* position= NIL v position^.wert = w *)
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```

Löschen: Lösung B -2

```
PROCEDURE Suche (liste : ListenElement;
                 w : INTEGER;
                 VAR vorgaenger, stelle : ListenElement) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE LoescheVorne(VAR liste : ListenElement) =
BEGIN
  IF liste # NIL THEN
    liste := liste^.nachfolger;
  END;
END LoescheVorne;
```


Beobachtung !

- **Einfügen und Loeschen benutzen ähnliche Such-Prozeduren**

```
PROCEDURE Suche (liste : ListenElement; w : INTEGER;
                 VAR vorgaenger, stelle : ListenElement) =
BEGIN
  stelle := liste; vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE SucheEinfuegestelle(liste : ListenElement;
                              w : INTEGER): ListenElement =
VAR stelle, pos: ListenElement;
BEGIN
  pos := liste; stelle := liste;
  WHILE (pos # NIL) AND (pos^.wert <= w) DO
    stelle := pos;
    pos := pos^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```

Prozedurtyp

- **Frage:**
  - Gibt es eine Möglichkeit, Prozeduren so zu ***parametrisieren***, daß als Parameter ein Algorithmus (Prozedur) übergeben werden kann?
- **Prozedurtyp**
  - Ein Prozedurtyp definiert eine ***Signatur***
  - Die Werte eines Prozedurtyps sind ***Prozeduren***, die der vorgegebenen Signatur entsprechen.
  - Entsprechend können Variablen als ***Prozedurvariablen*** deklariert werden.
  - Auf Prozedurvariablen können ***passende*** Prozeduren zugewiesen werden.
  - Prozedurvariablen können als ***Parameter*** übergeben werden.
  - Gesetzte Prozedurvariablen können in Anweisungen mit ***aktuellen Parametern*** gerufen werden.

Beispiel: ProzedurTyp - 1

```
TYPE  Bereich      = [-10 .. 10]; Index   = [1 .. 10];
      Zahlenmenge  = SET OF Bereich;
      Zahlenfeld   = ARRAY Index OF Bereich;
      TestProzedur = PROCEDURE (a : INTEGER) : BOOLEAN;

PROCEDURE IstPositiv (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i > 0);
END IstPositiv;

PROCEDURE IstNegativ (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i < 0);
END IstNegativ;

PROCEDURE Filtern (        feld : FeldTyp;
                   VAR resultat : Zahlenmenge;  p : TestProzedur) =
BEGIN
  FOR i := FIRST(Index) TO LAST(Index) DO
    IF p(feld[i]) THEN
      resultat := resultat + Zahlenmenge{feld[i]};
    END;
  END;
END Filtern;
```

Gegeben ist ein Feld mit Zahlen. Bestimme die darin enthaltenen positiven und negativen Zahlen

Signatur-konforme Prozeduren

Prozedur-parameter

Beispiel: ProzedurTyp - 2

```
VAR proc : TestProzedur;

    werte := Zahlenfeld{3, -5, 9, 8, -6, 9, -6, -1, -1, 5};
    positiv, negativ := Zahlenmenge{};

BEGIN
  proc := IstPositiv;
  Filtern (werte, positiv, proc);
  proc := IstNegativ;
  Filtern (werte, negativ, proc);

  SIO.PutLine ("Positive Zahlen:");
  FOR e := FIRST(Bereich) TO LAST(Bereich) DO
    IF e IN positiv THEN SIO.PutInt(e); SIO.PutText(", "); END;
  END;
  SIO.Nl();
```

Prozedurvariable

Zuweisung einer Prozedur an die Prozedurvariable

Zuweisung an Prozedurvariable

- **Zuweisung**
  - `TYPE PType = PROCEDURE …`
    `VAR proc : Ptype`

    `proc := PT; (* Ausdruck der vom Typ PT ist *)`
  - Diese Zuweisung ist korrekt, wenn
    - PT = ***NIL*** ist (NIL ist mit jedem Wert eines Prozedurtyps kompatibel)
    - ***Anzahl*** der Parameter und deren ***Typen*** sind gleich für PT und PType
    - Beide haben den gleichen ***Ergebnistyp*** oder keinen

```
TYPE TestProzedur = PROCEDURE (a : INTEGER)  : BOOLEAN;
     PT1          = PROCEDURE (in : INTEGER) : BOOLEAN;

VAR test : TestProzedur;
      p1 : PT1;
…
p1 := test;
test := p1;
```

Typen sind signatur-konform

Verwendung Prozedurtyp - 1

```
TYPE Vergleichsoperation = PROCEDURE (a, b : INTEGER) : BOOLEAN;

PROCEDURE Suche (liste : ListenElement;
                     w : INTEGER;
                 VAR vorgaenger, stelle : ListenElement;
                    op : Vergleichsoperation) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND  NOT op(stelle^.wert, w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

signaturkonform

```
PROCEDURE IstGleich
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a = b;
END IstGleich;
```

```
PROCEDURE Groesser
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a > b;
END Groesser;
```


Verwendung Prozedurtyp - 2

```
PROCEDURE Loeschen (VAR liste : ListenElement; w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : ListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position, IstGleich);
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```

Verwendung Prozedurtyp - 3

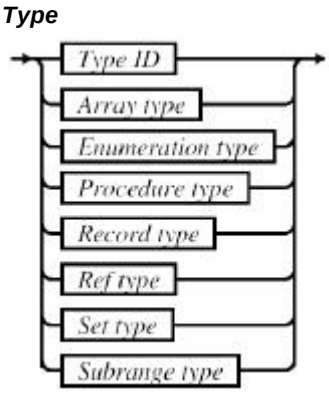
```
PROCEDURE Einfuegen (VAR liste : ListenElement; w : INTEGER) =
VAR neu, einfuegestelle, groessererWert : ListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                  (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN         (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);        (* neu vorne einhaengen *)
  ELSE
    Suche(liste, w, einfuegestelle, groesserWert , Groesser);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```


Diskussion: Dynamische Datentypen

- **Je deutlicher die Realisierung dynamischer Datenstrukturen durch Zeiger in der Sprache sichtbar ist,**
  - desto leichter fällt die software-technisch ***unsaubere*** Verwendung.
- **Die explizite Speicherverwaltung führt zu einigen Problemen.**
  - Moderne imperative Sprachen sollten daher einen ***Garbage Collector*** besitzen.
- **Erst in objektorientierten Sprachen,**
  - die vorrangig mit ***Referenzsemantik*** arbeiten,
  - können grundlegende Probleme der Referenzierung gelöst werden.

Zusammenfassung: Datentypen

- **Datentypen**
  - definieren ***Werte*** und zulässige ***Operationen***
  - helfen, ***Abstraktionen*** zu formulieren
  - repräsentieren das ***Vokabular*** eines Systems
    - Person, Liste etc.
  - dienen der ***Sicherheit*** der Programme
    - jedes Objekt hat einen Typ
    - Typ-Prüfung bei Parameterübergabe und Zuweisung
  - müssen wohlüberlegt ***entworfen*** werden
    - manifestieren sich im Programm
    - sind dann nur ***schwer veränderbar***



Das Modulkonzept

- **Modulkonzept**
  - Export-Schnittstelle
  - Import-Schnittstelle
- **Implementierung einer Schnittstelle**
- **Austausch einer Implementierung**
- **Vorteile modularer Architekturen**

Modulkonzept

- **Die imperative Programmierung ist eng mit dem *Modulkonzept* verknüpft.**
- **Entstanden aus der Notwendigkeit,**
  - große Programmtexte in für den ***Übersetzer faßliche Einheiten*** zu zerlegen,
  - Modulkonzept ist zum zentralen ***Organisationskonzept*** für Entwürfe und Programmtexte geworden.
- **Module werden hier als**
  - *Konstruktionsmerkmal* einer imperativen Sprache eingeführt.
- **Die Diskussion,**
  - wie das Modulkonzept genutzt werden sollte, folgt in einem eigenen Teil.

Definition: Modul

- **Nach Informatik-Duden:**
  - Ein Modul ist die ***Zusammenfassung von Konstanten, Datentypen, Variablen und Prozeduren*** zu einer Einheit. Soll ein Modul von einem anderen benutzt werden, so muß man angeben, welche Teile dieses Moduls von ***außen sichtbar*** sein sollen und welche nicht. Grundsätzlich bleibt aber die Implementierung eines Moduls, also die konkrete Realisierung der Datentypen und Prozedurrümpfe, vor allen anderen Modulen ***verborgen***.
- **Nach Goos:**
  - Unter einem Modul verstehen wir eine ***Sammlung von Objekten und Algorithmen*** mit der Eigenschaft, daß ihre Kommunikation mit der Außenwelt nur über eine klar ***definierte Schnittstelle*** erfolgt. Das ***Zusammensetzen*** mehrerer Module zu einer Gesamtlösung darf keine Kenntnis ihres ***inneren Aufbaus*** voraussetzen, und die Korrektheit eines Moduls muß ohne Kenntnis seiner Einbettung in die Gesamtlösung nachprüfbar sein.

Erinnerung: Lebensdauer

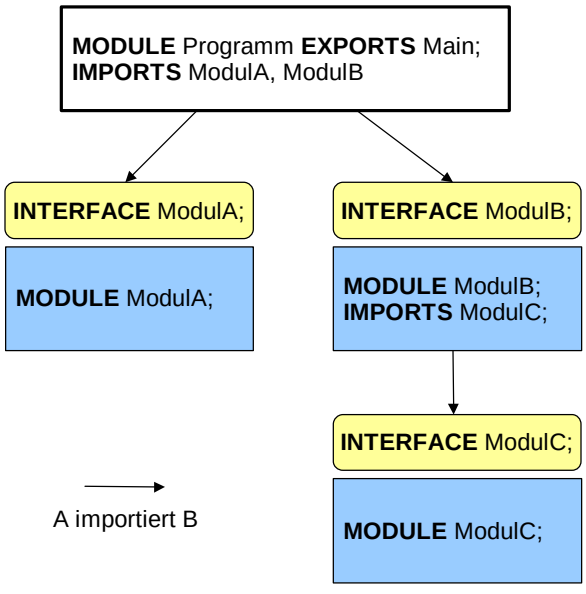
- **Prozedur:**
  - Alle Objekte im Namensraum einer Prozedur existieren nur solange die Prozedur aktiv ist.
  - Bei jedem neuen Aufruf einer Prozedur werden u.a. die lokalen Variablen neu angelegt.
- **Modul:**
  - Module sind ***statisch***, d.h. ihr Namensraum existiert solange das Programm oder die Anwendung ***insgesamt*** aktiv ist.
  - Variablen, die im Deklarationsteil eines Moduls eingeführt werden, haben die gleiche Lebensdauer wie das Modul; sie heißen ***global***.
- **In Modula-3**
  - können Module ***nicht*** ineinander geschachtelt werden!
  - Bei Prozeduren ist dies möglich!

Aufbau von Modula-3 Programmen

- **Modula-3 Programm**
  - besteht wenigstens aus einem ***Modul***, dem ***Hauptmodul***
- **Modul-Aufbau**
  - ein Modul besteht (bis auf das Hauptmodul) aus
    - ***Schnittstelle*** (interface)
      - definiert, was ein Modul exportiert
      - Exportschnittstelle
    - ***Implementierung***
      - enthält die Implementierung der exportierten Elemente
      - versteckt die Implementierung
- **Bisher**
  - bestanden unsere Programme lediglich aus einem Modul, dem Hauptmodul

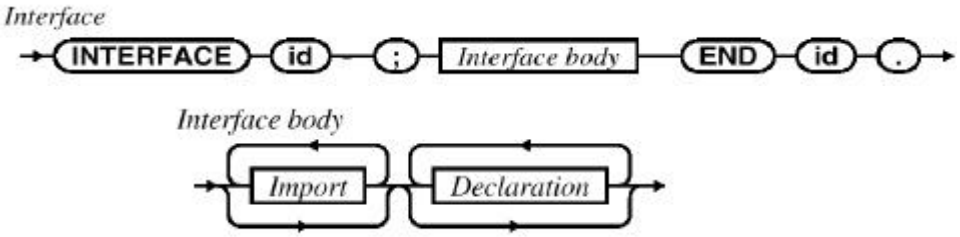
Modul-Hierarchie

- ein Modul kann Elemente anderer Module ***benutzen***
  - ein Modul ***importiert*** dazu andere Module
  - von einem importierten Modul ist nur die ***Schnittstelle*** sichtbar



Die Export-Schnittstelle

- **Wie sieht eine Export-Schnittstelle eines Moduls aus?**



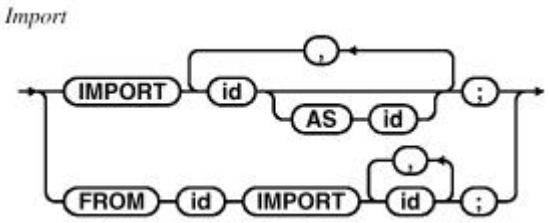
- **Alle in der Schnittstelle deklarierten Objekte werden vom Modul *exportiert***
  - können von importierenden Modulen ***verwendet*** werden.
- **Hinweis**
  - alles was in der Schnittstelle "***versprochen***" wird, muß auch von der Implementierung realisiert werden

Beispiel: Export-Schnittstelle

```
INTERFACE SIO;

IMPORT Fmt, Rd, Wr, Word;

...

PROCEDURE GetChar(rd: Reader := NIL): CHAR RAISES {Error};
(* Read next character from stream rd and return it *)

PROCEDURE PutChar(ch: CHAR; wr: Writer := NIL);
(* Write ch to outputstream wr. *)

PROCEDURE GetText(rd: Reader := NIL; len: CARDINAL): TEXT;
(* Read a sequence of len characters from rd and return them If there are not
enougth characters return what is there *)

PROCEDURE PutText(t: TEXT; wr: Writer := NIL);
(* Write character sequence t to outputstream wr. *)

PROCEDURE GetLine(rd: Reader := NIL): TEXT RAISES {Error};
(* Read a full line of text terminated by the nextRETURN from
   inputstream rd and return it (without RETURN!). *)
. . .
END SIO.
```

IMPORT-Schnittstelle - 1

- **Um Programmobjekte über Modulgrenzen zu verwenden,**
  - müssen sie ***gezielt angefordert*** werden.
  - Die IMPORT-Klausel dient dazu,
    - die Leistungen in einem anderen Modul ***sichtbar*** zu machen.
- **Mögliche IMPORT-Varianten:**
  - importieren ***aller*** Leistungen eines Moduls
  - importieren ***aller*** Leistungen eines Moduls unter einem ***Alias-Namen***
  - importieren nur der Leistungen eines Moduls, die ***tatsächlich*** vom importierenden Modul verwendet werden



IMPORT-Klausel - 2

- **Beispiele für Importe:**
  - wird nicht selektiv importiert, muß der Modulname als ***Qualifikator*** verwendet werden

```
IMPORT Text;                          (* import aller Deklarationen *)
IMPORT Rd AS Reader;                  (* import mit Alias-Namen *)
FROM SIO IMPORT                       (* selektives Importieren *)
  (*PROCS*) PutLine, PutText, GetChar;
…
VAR eingabestrom : Reader.T;
…
n := Text.Length(t1);
…
PutText("abcdefg");
```

Diskussion: Import-Varianten

- **Globales Importieren**
  - ↑ An ***jeder Stelle*** im Programmtext ist ersichtlich, wo das jeweilige Objekt deklariert ist.

    ```
    Text.Length(t1);   Length(m3);   List.Length(l1);
    ```
  - ↑ ***Identisch deklarierte*** Bezeichner verschiedener Module können verwendet werden.
  - ↓ Zum Teil erheblich ***längere Schreibweise.***
  - ↓ Erst mit einem Werkzeug kann einfach ermittelt werden, was ***tatsächlich*** alles von einem Modul verwendet wird.
- **Selektives Importieren**
  - ↑ ***kürzere*** Schreibweise
  - ↑ Es ist alles das, was verwendet wird, auch ***explizit angegeben-***
  - ↓ Es ist nicht direkt ersichtlich, wo ein Bezeichner deklariert ist.

Regeln

- **1. Schnittstelle und Implementierung eines Moduls sind in *unterschiedlichen* Dateien (Programmtextdatei) enthalten.**
  - Jede Programmtextdatei bildet eine ***Übersetzungseinheit*** und kann vom Übersetzer getrennt behandelt werden.
- **2. Importieren Sie immer alle Deklarationen eines Moduls**
  - Verwenden Sie den Qualifikator!
- **3. Zyklische Importe sind verboten!**
  - A importiert B, B importiert A
  - zyklischer Import ist ein Hinweis auf eine ***schlechte Modularisierung***

Implementierung einer Schnittstelle

- **Regel:**
  - Die Implementierung einer Schnittstelle realisiert ***alle*** in der Schnittstelle ***deklarierten*** Prozeduren (Funktionen).
- **EXPORTS-Klausel**
  - gibt an, welche Schnittstelle eine Implementierung realisiert

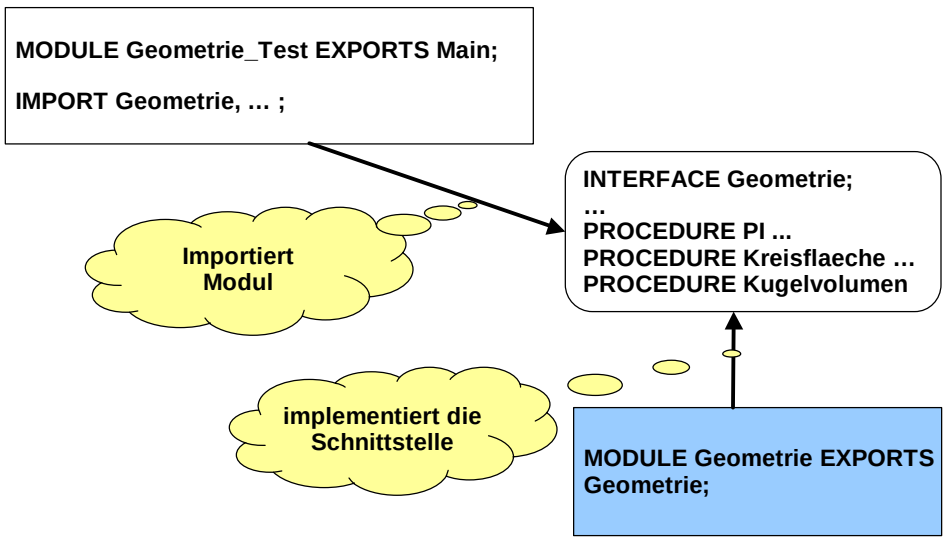
    ```
    MODUL Geometrie EXPORTS Geometrie;
    ```

  - Fehlt die EXPORTS-Klausel, dann realisiert die Implementierung eine Schnittstelle ***gleichen Namens.***

    ```
    MODUL Geometrie;
    ```

- **Modul-Initialisierung**
  - Im ***Block*** eines Moduls wird das Modul initialisiert.
  - Regel: Ein importiertes Modul wird vor dem importierenden Modul initialisiert.

Beispiel



Beispiel: Schnittstelle

```
INTERFACE Geometrie;

PROCEDURE PI () : REAL;
PROCEDURE Kreisflaeche(r :REAL): REAL;
PROCEDURE Kugelvolumen(r : REAL): REAL;

END Geometrie.
```

```
MODULE Geometrie_Test EXPORTS Main;
IMPORT Geometrie, SIO;

VAR radius : REAL;
BEGIN
  SIO.PutText ("Geben Sie bitte einen Radius ein: ");
  radius := SIO.GetReal();
  SIO.PutText ("Kreisflaeche: ");
  SIO.PutReal (Geometrie.Kreisflaeche(radius));
  SIO.Nl();
  SIO.PutText ("Kugelvolumen: ");
  SIO.PutReal (Geometrie.Kugelvolumen(radius));
END Geometrie_Test.
```


Beispiel: Implementierung

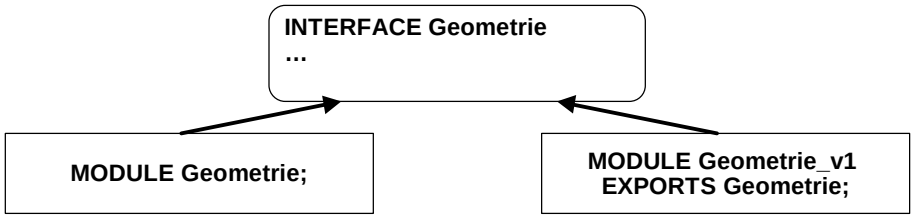
```
MODULE Geometrie EXPORTS Geometrie;

VAR pi : REAL;

PROCEDURE PI () : REAL =
BEGIN
  RETURN pi;
END PI;

PROCEDURE Kreisflaeche(radius :REAL): REAL =
BEGIN
  RETURN (pi * radius * radius);
END Kreisflaeche;

PROCEDURE Kugelvolumen(radius : REAL): REAL =
BEGIN
  RETURN ( (4.0/3.0) * pi * radius * radius * radius );
END Kugelvolumen;

BEGIN
  pi := 3.14147;
END Geometrie.
```

Interne Variable

Realisierung der Prozeduren / Funktionen der Schnittstelle

Initialisierung der Moduls

Austausch der Implementierung




```
% Makefile ...
…

% module(Geometrie)

interface(Geometrie)
implementation (Geometrie_v1)

implementation (Geometrie_Test)
program (Geometrie_Test)
```

```
MODULE Geometrie_v1 EXPORTS Geometrie;

PROCEDURE PI () : REAL =
BEGIN
  RETURN 3.14147;
END PI;

PROCEDURE Kreisflaeche(r :REAL): REAL = ...

PROCEDURE Kugelvolumen(r : REAL): REAL = ...
BEGIN
END Geometrie_v1.
```


Vorteile modularer Programme

- **Vorteile modularer Programme**
  - Module können von ***unterschiedlichen*** Personen entwickelt und gepflegt werden.
    - Software-Entwicklung ist Team-Arbeit!
  - Module können einzeln ***getestet*** werden.
    - Test großer Programme ist extrem aufwendig!
  - Module können geordnet zum Gesamtsystem ***integriert*** werden.
  - Eine Implementierung eines Moduls kann leicht durch eine neue Implementierung ***ersetzt*** werden.
    - z.B. durch eine effizientere Implementierung
  - Module können in verschiedenen Programmen ***wiederverwendet*** werden (Modul-Bibliothek).
    - Dies senkt die Kosten für die Entwicklung!

Diskussion: Modulkonzept

- **Module sind Sammlungen von Programmobjekten und Algorithmen:**
  - Sie sind keine ***direkt aufrufbaren*** Programmeinheiten (wie Prozeduren).
  - Sie sind eine Einheit für die ***Übersetzung***.
- **Als Sammlung sollen sie *keine* beliebige Anordnung sein**
  - Kriterien für die Zusammenstellung eines Moduls müssen geklärt werden.
  - Module verbergen ***Implementierungen*** d.h. ihren inneren Aufbau und zeigen nur ihre Schnittstelle:
  - Es gibt unterschiedliche ***starke Möglichkeiten*** des Verbergens.
  - Der Aufbau einer Schnittstelle unterliegt bestimmten Kriterien.
- **Module benutzen andere Module:**
  - Das Zusammenspiel verschiedener Module bezeichnet man auch als ***Architektur***.
  - Die ***Benutzt-Beziehung*** koppelt Module miteinander.

Information Hiding

- **Prozeß- und Datenabstraktion**
- **Information Hiding**
- **Objektmodule (Datenkapselung)**
- **Abstrakte Datentypen**

Programmieren im Großen

- **Programmieren im Kleinen**
  - befaßt sich mit der Konstruktion eines ***Programms***. Wir haben bisher die dazu notwendigen Konzepte kennengelernt:
    - Daten- und Kontrollstrukturen,
    - Typen,
    - Prozeduren und Funktionen.
- **Programmieren im Großen**
  - für die Entwicklung großer Softwaresysteme müssen weitere Konzepte hinzukommen. Das vorrangige Problem ist, die ***Komplexität*** großer Softwaresysteme zu beherrschen.
  - Die gewählte Lösung heißt ***Abstraktion***.
- **Für den modularen Softwareentwurf sind zwei Abstraktionskonzepte entscheidend:**
  - ***Prozeßabstraktion***,
  - ***Datenabstraktion***.

Prozeß- und Datenabstraktion

- **Prozeßabstraktion (auch algorith. Abstraktion):**
  - Funktionen und Prozeduren werden zu ***abstrakten*** Konzepten.
  - Bekannt sind nur die ***Eingabe***- und ***Ausgabegrößen***, aber nicht die Implementation.
  - z.B.: Für eine Liste ist wesentlich, daß wir ein Element anfügen oder entfernen können, aber nicht wie das geschieht.
- **Datenabstraktion:**
  - Schließt konzeptionell die Prozeßabstraktion ein.
  - Zusätzlich werden die ***Details*** der verwendeten Daten verborgen.
  - Beispiel:
    - Für eine Liste ist wichtig, daß sie ein Behälter für Elemente ist und daß bestimmte Zugriffsoperationen erlaubt sind,
    - aber nicht, wie die Elemente der Liste gespeichert und bearbeitet werden.
  - Eine weitergehende Form von Datenabstraktion abstrahiert auch von der ***Art der Elemente***, die in der Liste gespeichert werden.

Information Hiding

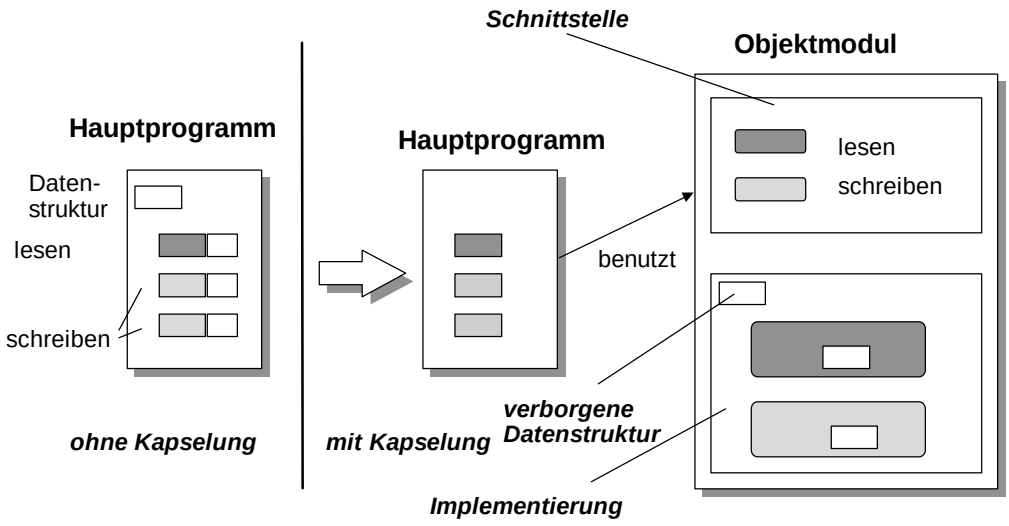
- **Prinzip:**
  - Es werden nur die Informationen zur Verfügung gestellt, die ***absolut notwendig*** sind!
  - Alle anderen, insbesondere die ***wichtigen Informationen*** werden ***versteckt***!
  - Der Zugriff auf diese Informationen geschieht über "***Vermittler***".
- **Beispiel: "Offene Bank"**
  - Funktioniert nicht, weil
    - die Buchhaltung nicht klappt (Änderungen werden nicht jedem bekannt sein)
    - doch gestohlen wird (was mißbraucht werden kann, wird mißbraucht)
- **Deshalb:**
  - Das wertvolle wird versteckt (Geld -Tresor)
  - Es werden Vermittler eingesetzt (Mitarbeiter, Automaten)

D. Parnas, 1972

Objektmodul - Datenkapsel

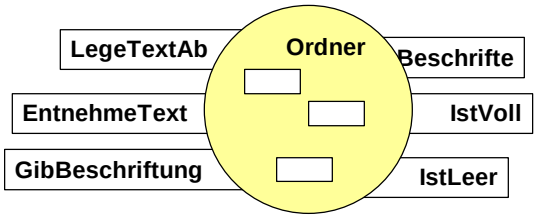
- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Datenstruktur von ihren sichtbaren Eigenschaften.
- **Merkmale:**
  - Eine Datenstruktur wird in einem Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen sichtbar***, die den allgemeinen Umgang mit der Datenstruktur beschreiben.
  - Die Datenstruktur selbst ist ***verborgen***.
- **Jedes Objektmodul**
  - beschreibt und realisiert nur eine ***einzige sog. abstrakte Datenstruktur***.
- **Kapselung von Daten**
  - ist ein zentraler Denkansatz und ein wesentliches ***Entwurfsprinzip***

Schema: Objektmodul



Beispiel: Das Objektmodul Ordner

- **"Ordner " als Konzept**
  - ***enthält*** Texte
  - Texte können ***abgelegt*** und ***entnommen*** werden
  - ein Ordner kann ***beschriftet*** werden
  - ein Ordner kann ***leer*** oder ***voll*** sein
- **Ein Objektmodul**
  - realisiert ein ***fachliches Konzept*** (z.B. das Konzept "Ordner")
  - als ***genau eine abstrakte Datenstruktur***



Realisierung Objektmodul

```
INTERFACE Ordner;

PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();

END Ordner.
```

Objektmodul Ordner ist ein Behälter für Daten des Typs TEXT

Abstrakte Beschreibung des fachlichen Konzepts Ordner

Das eine Objekt Ordner soll mehrfach pro Programmablauf verwendbar sein

Verwendung eines Objektmoduls

```
MODULE Ordner_Test EXPORTS Main;

IMPORT Ordner,  SIO;

BEGIN
  Ordner.Beschrifte ("Kleine Gedichte");
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
  Ordner.LegeTextAb ("Herr von Ribeck auf Ribeck ...");
  Ordner.LegeTextAb ("Von drauss vom Walde komm ich her ...");
  SIO.PutLine (Ordner.GibBeschriftung());
  SIO.PutLine ("--------------------------");
  SIO.PutLine (Ordner.EntnehmeText());
  SIO.PutLine (Ordner.EntnehmeText());
  SIO.PutLine (Ordner.EntnehmeText());
  Ordner.Initialisiere();
END Ordner_Test.
```

Diskussion: Wie darf ein Objekt-modul verwendet werden?

Verwendung eines Objektmoduls

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Feststellung:**
  - `LegeTextAb` und `Entnehme` sind ***nicht in jedem Zustand*** des Ordners sinnvoll:
    - ein voller Ordner kann keine weiteren Texte aufnehmen; ein leerer keine herausgeben.
  - Solche Operationen sind nur in bestimmten Situationen (abhängig von bestimmten Bedingungen) sinnvoll.
  - Um den sicheren Umgang mit einem solchen Objekt zu gewährleisten, werden an der Schnittstelle entsprechende ***Testfunktionen*** wie `IstLeer` oder `IstVoll` zur Verfügung.

Einsatz der Testfunktionen

Prüfen der Vorbedingung

Das Objektmodul **Ordner** wird vor der ersten Verwendung initialisiert, d.h. der Inhalt wird gelöscht

```
Ordner.Initialisiere;

IF Ordner.IstVoll() THEN
  SIO.PutLine ("Ordner ist bereits voll");
ELSE
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
END;

IF Ordner.IstLeer() THEN
  SIO.PutLine ("Ordner ist leer");
ELSE
  t := Ordner.EntnehmeText();
END;
```

Vor Entnahme wird der Zustand des Objektmoduls Ordner geprüft und entsprechend gehandelt.

Entwurfskonzept

- **Um ein fachliches Konzept als Objektmodul zu realisieren, stellen wir *drei Arten von Operationen* zur Verfügung.**
- **Prozeduren:**
  - ***verändern*** den ***Zustand*** des Objekts. Meist sind sie von Vorbedingungen abhängig, d.h. sie können nicht in jedem Zustand ausgeführt werden.
- **Funktionen:**
  - liefern Informationen, ***ohne*** den Objektzustand nach außen sichtbar ***zu verändern***.
  - Fachliche Funktionen:
    - liefern ***fachliche Informationen*** und sind vom Objektzustand abhängig.
  - Testfunktionen:
    - ***prüfen*** den Objektzustand und werden zum Prüfen der Vorbedingungen verwendet.

Entwurfskonzept Beispiel: Ordner

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Prozeduren:**
  - `Initialisiere`, `LegeTextAb`, `EntnehmeText`, `Beschrifte` sind verändernde Prozeduren.
- **Fachliche Funktionen:**
  - `GibBeschriftung` ist eine fachliche Funktion; sie verändert im Gegensatz zu `EntnehmeText` nicht den Ordnerzustand.
- **Textfunktionen:**
  - `IstLeer` und `IstVoll`

Ein Blick ins Innere - 1

```
MODULE Ordner EXPORTS Ordner;

CONST MaxTexte = 20;
TYPE Fassungsvermoegen = [1 .. MaxTexte];
TYPE Texte = ARRAY Fassungsvermoegen OF TEXT;
VAR ordnerInhalt : Texte;
    anzahlTexte : CARDINAL;
    beschriftung : TEXT := "";

PROCEDURE LegeTextAb (t : TEXT)=
BEGIN
  anzahlTexte := anzahlTexte + 1;
  ordnerInhalt[anzahlTexte] := t;
END LegeTextAb;

PROCEDURE EntnehmeText () : TEXT =
VAR t : TEXT;
BEGIN
  t := ordnerInhalt[anzahlTexte];
  ordnerInhalt[anzahlTexte] := "";
  anzahlTexte := anzahlTexte - 1;
  RETURN t;
END EntnehmeText;
```

Es wird ein Array verwendet

Es wird der zuletzt abgelegte Text entnommen

Ein Blick ins Innere - 2

```
PROCEDURE IstVoll () : BOOLEAN =
BEGIN
  RETURN (anzahlTexte = MaxTexte);
END IstVoll
...

PROCEDURE Beschrifte (t : TEXT)=
BEGIN
  beschriftung := t;
END Beschrifte;
...

PROCEDURE Initialisiere () =
BEGIN
  FOR i := FIRST(Fassungsvermoegen) TO LAST(Fassungsvermoegen) DO
    ordnerInhalt[i] := "";
  END;
  anzahlTexte := 0;
END Initialisiere;

BEGIN
  Initialisiere ();
END Ordner.
```

IstVoll verwendet die Variable anzahlTexte

Zusammenfassung Objektmodul

- **Eigenschaften eines Objektmoduls:**
  - Das Modul verwaltet seine Daten ***selbst***.
  - Die interne Repräsentation der Daten ist vollständig ***verborgen***.
  - Die interne Repräsentation ist ***austauschbar***.
  - Zur Laufzeit existiert immer ***nur ein Exemplar*** eines Objektmoduls, d.h. es können z.B. nicht mehrere Ordner erzeugt werden.
  - Das Objektmodul kann von ***mehreren*** anderen "Kunden" gleichzeitig verwendet werden.
  - Dadurch kann z.B. ein ***gemeinsamer*** Ordner verwaltet werden.

Module als Sprachelement

- **Ein Modul kann zwar**
  - ***definiert*** und zur Laufzeit von anderen Modulen ***importiert*** werden,
  - aber es ist kein ***primäres*** Sprachelement (first class), d.h.
  - ein Modul kann ***nicht*** wie ein Typ ***zur Deklaration*** von Bezeichnern verwendet werden.

  - `o1, o2 : Ordner (* geht nicht, da Ordner Modul ist *)`
- **Folge:**
  - es gibt ***nur ein Exemplar*** eines Objektmoduls.
- **Problem:**
  - Es werden oft mehrere Exemplare eines durch ein Objektmodul realisierten Objekts gebraucht.
- **Lösungsansatz:**
  - Wir formulieren ***einen Typ*** für die im Modul beschriebenen Objekte.

Konzept Abstrakter Datentyp

- **Kann als *Generalisierung* des Objektmoduls (Datenkapsel) betrachtet werden.**
- **Anstatt eines Objektes wird ein Typ (für diese Objekte) definiert.**
- **Betrachten wir den ADT als (formale) Spezifikation eines Typs,**
  - dann entwerfen wir ihn durch Angabe von
  - ***Typnamen***
  - ***Signaturen*** (Operationen)
    - zum Erzeugen von Objekten, zum Verändern etc.
  - ***Axiome***
    - formulieren den semantischen Zusammenhang der Operationen.
  - ***Vorbedingungen***
    - geben ab, in welchem Zustand welche Operationen gültig sind.

Beispiel 1: ADT Bool

```
TYPE BOOL

FUNCTIONS

  true:  -> BOOL
  false: -> BOOL
  not:   BOOL -> BOOL
  and:   BOOL x BOOL -> BOOL
  or:    BOOL x BOOL -> BOOL

AXIOMS

  not(true) = false
  not(false) = true

  For any x: BOOL

    not(not(x)) = x

    or(true, x) = true     and(false, x) = false
    or(x, true) = true     and(x, false) = false
    or(false, x) = x       and(true, x)  = x
    or(x, false) = x       and(x, true)  = x
```


Beispiel 2: ADT NAT

```
TYPE    NAT

FUNCTIONS

  zero: -> NAT
  succ: NAT -> NAT
  iszero: NAT -> BOOL
  pred: NAT -> NAT

  add, mult, sub : NAT x NAT -> NAT

AXIOMS                                  PRECONDITIONS

  For any x, y: NAT                     pred(x: NAT)
                                        requires iszero(x) = false

    pred(succ(x))    = x
    iszero(zero)     = true
    iszero(succ(x))  = false

    add(zero, x)     = x
    add(succ(x), y)  = succ(add(x, y))
    sub(x, zero)     = x
    sub(x, succ(y))  = pred(sub(x, y))
    mult(x, zero)    = zero
    mult(x, succ(y)) = add(mult(x, y), x)
```

Beispiel: ADT

TYPE

Stack [G]

FUNCTIONS

put: Stack[G] x G -> Stack[G]

remove: Stack[G] -> Stack[G]

item: Stack[G] -> G

empty: Stack[G] -> Boolean

new: Stack[G]

AXIOMS

For any x: G, s: Stack[G]

item (put(s,x) = x

remove (put (s,x)) = s

empty(new)

not empty(put (s,x))

PRECONDITIONS

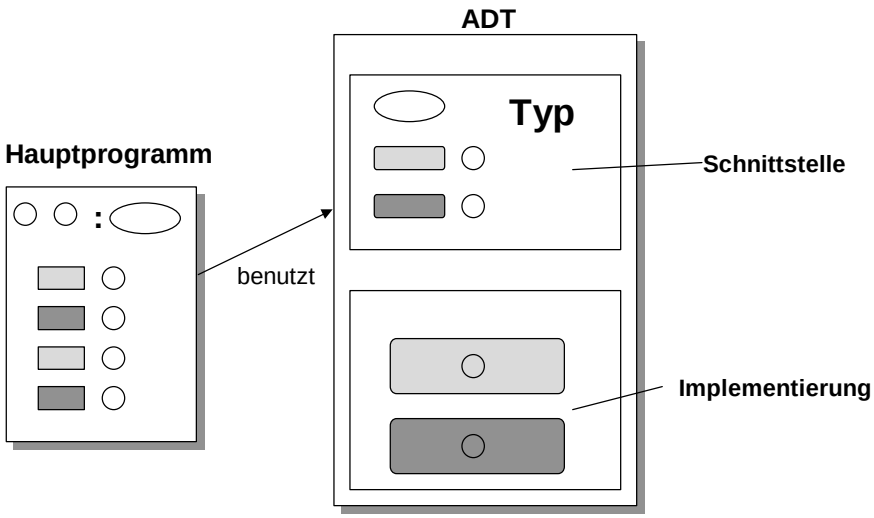
remove (s: Stack[G]) require not empty (s)

item (s: Stack[G]) require not empty (s)

Realisierung ADTen durch Module

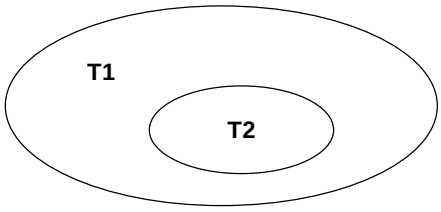
- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Menge gleichartiger Datenstrukturen von ihren allgemeinen Eigenschaften.
- **Merkmale:**
  - Die Beschreibung der gleichartigen Exemplare einer Datenstruktur wird in einem sog. ADT-Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen*** sichtbar, die den allgemeinen Umgang mit ***jedem Exemplar*** der Datenstruktur beschreiben.
  - Ein ***Bezeichner für den Typ*** der Datenstruktur wird exportiert.
  - Die Implementierung der Datenstruktur selbst ist ***verborgen***.
- **Jedes ADT-Modul beschreibt und realisiert eine Menge von Exemplaren der abstrakten Datenstruktur.**
  - Die Exemplare werden von ***ihren Kunden*** verwaltet. Ihre Bearbeitung geschieht nur mit Hilfe der exportierten Operationen des ADT-Moduls.

Realisierungsschema ADT



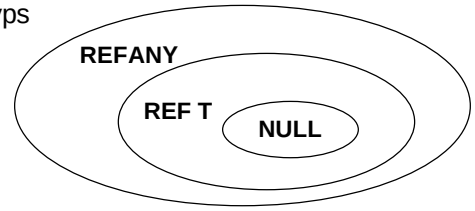
Exkurs: Subtypen in Modula-3

- **Subtypen in Modula-3**
  - Modula-3 kennt das Konzept der ***Subtyp-Bildung***
  - Subtypbeziehung wird durch "***<:***" angezeigt
- **Definition**
  - Seien T1 und T2 Typen und die Relation T2 <: T1 besteht, dann sind ***alle Werte*** von T2 auch ***Werte*** von T1
  - T1 nennt man ***Obertyp***; T2 nennt man ***Untertyp***
  - Es gilt: ein Typ kann beliebig viele Subtypen haben, ein Typ kann ***maximal*** einen Obertyp haben.



Subtypen von Referenztypen

- **Modula-3**
  - definiert zwei vorgegebene Referenztypen
  - ***REFANY*** und ***NULL*** mit folgender Subtypbeziehung
  - `NULL <: REF T <: REFANY`
- **Interpretation:**
  - jeder Referenztyp in ***Untertyp*** von REFANY
  - NULL ist ***Untertyp*** jedes Referenztyps
    - Der Referenztyp NULL kennt nur den Wert NIL
    - NIL ist Wert jedes Referenztyps



Realisierung eines ADT in Modula-3

- **Forderung:**
  - In der Schnittstelle darf die Typstruktur eines ADTs ***nicht sichtbar*** sein.
  - Lediglich der ***Name*** soll bekannt gemacht werden.
- **Realisierung im Modula-3**
  - In der Schnittstelle wird ein ***opakter Typ*** (verdeckter Typ) deklariert
  - Dies geschieht, indem der Typ als ***Untertyp*** zum vordefinierten Typ REFANY deklariert wird.
  - ***Obertyp*** eines opakten Typs muß ein ***Referenztyp*** sein.
  - In der Implementierung des ADTs wird der opake Typ ***offengelegt*** ("enthüllt")
  - Bspl:

```
TYPE Ordner <: REFANY;
```

Mithilfe des Subtyp-Konzepts und mit opaken Tyoen können die Forderungen umgesetzt werden

Schnittstelle des ADT-Moduls Ordner

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t :
PROCEDURE EntnehmeText (VAR o: Ordner; ) :  EXT ;
PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
PROCEDURE Beschrifte (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung ( o: Ordner; ) : TEXT;
PROCEDURE Anlegen ( ) : Ordner;

END OrdnerADT.
```

Nur der Name des ADT ist sichtbar!

Alle Operationen erhalten als ersten Parameter das jeweilige Ordnerobjekt

Erzeugt ein neues Ordnerobjekt und gibt es zurück

Beispiel: ADT Ordner

Angabe des Typnamens

Angabe der Operationen

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
PROCEDURE Beschrifte (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung ( o: Ordner; ) : TEXT;
PROCEDURE Anlegen ( ) : Ordner;

END OrdnerADT.
```

```
IMPORT OrdnerADT;
VAR
  ord1, ord2 : OrdnerADT.Ordner

BEGIN
  ord1 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ord1, "Kleine Gedichte");
  OrdnerADT.LegeTextAb (ord1, "Nicht immer sind bequeme ...");

  ord2 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ord2, "Musikstuecke");
  OrdnerADT.LegeTextAb (ord2, "Tief im Westen ...");
```

Implementierung des ADT Ordner

```
MODULE OrdnerADT EXPORTS OrdnerADT;

CONST MaxTexte = 20;
TYPE Fassungsvermoegen = [1 .. MaxTexte];
     Texte = ARRAY Fassungsvermoegen OF TEXT;

REVEAL Ordner = BRANDED REF RECORD
                  ordnerInhalt : Texte;
                  anzahlTexte  : Fassungsvermoegen;
                  beschriftung : TEXT := "";
                END;
```

- **REVEAL-Anweisung**
  - damit wird die ***interne Struktur*** des opakenTyps bekannt gegeben
  - Steht immer in der ***Implementierung*** eines ADTs (information hiding).
  - Der äußere Typ-Konstruktor ***muß*** ein mit einem ***Brandzeichen*** versehener Referenztyp sein
  - Dieses unterscheidet den Typ von anderen ***strukturell gleichen*** Typen.

Diskussion : ADT-Realisierung in Modula-3

- **Wir können feststellen**
  - Als Typ für einen ADT müssen wir einen opaken ***Referenztyp*** wählen.
  - ***Grund***: Nur so kann der Übersetzer für Objekte eines ADTs ausreichend Speicherplatz reservieren, ohne den inneren Aufbau des Typs zu kennen.
- **Konsequenz:**
  - Es können neben den Operationen der Schnittstelle des ADTs auch die Operationen
    - ***Zuweisung*** und
    - ***Vergleich*** auf Objekten des ADTs durchgeführt werden (Pointer-Zuweisung und Pointer-Vergleich).
  - Diese sollten jedoch ***nicht genutzt*** werden!
  - Das Brandzeichen verhindert, daß einem Objekt eines ADT zufälligerweise ein Wert eines ***strukturell gleichen Typs*** zugewiesen werden kann.

Zuweisung und Vergleich von ADT-Objeken

```
ordner1 := OrdnerADT.Anlegen();
OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
OrdnerADT.LegeTextAb (ordner1, "Nicht immer sind bequeme Stuehle ...");

ordner2 := OrdnerADT.Anlegen();
OrdnerADT.Beschrifte (ordner2, "Kleine Gedichte");
OrdnerADT.LegeTextAb (ordner2, "Nicht immer sind bequeme Stuehle ...");

IF ordner1 = ordner2 THEN
  SIO.PutLine ("1 ordner sind gleich");
ELSE
  ordner2 := ordner1;
  OrdnerADT.Beschrifte (ordner1, "Romane");
END;
IF ordner1 = ordner2 THEN
  SIO.PutLine ("nach Zuweisung sind die Ordner gleich");
END;

SIO.PutLine (OrdnerADT.GibBeschriftung(ordner1));
SIO.PutLine ("----------------------------");
SIO.PutLine (OrdnerADT.GibBeschriftung(ordner2));
```

Vergleich der Referenzen (Adressen

ordner2 zeigt auf die selbe Adresse wie ordner1

Ändert ordner1 und ordner2

Modul als Entwicklungseinheit

- **Ein Modul ist charakterisiert durch eine *Entwurfs-entscheidung*.**
  - (z.B. wir wollen Ordner verarbeiten können)
- **Jedes Modul basiert auf genau einer Entwurfsentscheidung.**
- **Module *verbergen* Implementierungsentscheidungen.**
  - Jedes Modul hat sein Geheimnis.
- **Es werden immer die Entscheidungen**
  - in einem Modul gekapselt, die vielleicht ***revidiert*** werden müssen als andere.
  - z.B. Benutzungsoberfläche
- **Der Änderungsaufwand muß möglichst immer auf ein Modul begrenzt werden.**

Zusammenfassung ADT

- **In imperativen Sprachen**
  - mit einem ***Modulkonzept***
  - realisieren wir abstrakte Datentypen mit einem ADT-Modul.

- **Ein ADT-Modul zeigt folgende Eigenschaften:**
  - Das Modul liefert ***Exemplare einer Datenstruktur***, die beim Kunden aufbewahrt werden.
  - Dadurch können ***mehrere Exemplare*** eines ADT-Moduls bearbeitet werden.
  - Die Datenstruktur ist nur als ***Verweis*** auf die interne Repräsentation bekannt.
  - Die Repräsentation selbst ist ***verborgen*** und ***austauschbar***.
  - Die Datenstrukturen des ADT-Moduls können (fast) nur über die ***exportierten Operationen*** des Moduls bearbeitet werden.

Was haben wir gelernt!

- **Modularisierung**
  - Mittel, um ***handhabbare*** Programmeinheiten zu konstruieren
  - Module bestehen aus ***Schnittstelle*** und ***Implementierung***

- **Information Hiding**
  - Ziel:
    - Implementierung wesentlicher Datenstrukturen ist ***nicht*** nach ***außen sichtbar.***
  - Objektmodul:
    - Es wird in einem Modul ***ein Objekt*** gemäß Information Hiding realisiert.
  - Abstrakter Datentyp:
    - Es wird ein ***geschützter Typ*** realisiert. Objekte des ADTs können nur mit den Operationen des ADTs manipuliert werden.

Beispiel einer Programmentwicklung

- Aufgabenstellung
- Erste Lösung
- Lösung mit ausgelagerten Prozeduren
- Lösung mit einem Objektmodul
- Lösung mit einem Abstrakten Datentyp
- Lösung mit zwei Abstrakten Datentypen

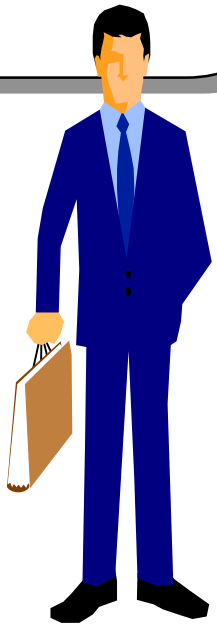
Aufgabenstellung

Herr X, ein engagierter Jungunternehmer, hat viele geschäftliche Verbindungen.

Da er sich die Adressen seiner Geschäftspartner nicht merken kann und die
Sammlung der Visitenkarten im Geldbeutel bereits sehr unübersichtlich
geworden ist, wünscht sich Herr X eine Möglichkeit, um die **Adressen** seiner
Geschäftspartner zu verwalten.

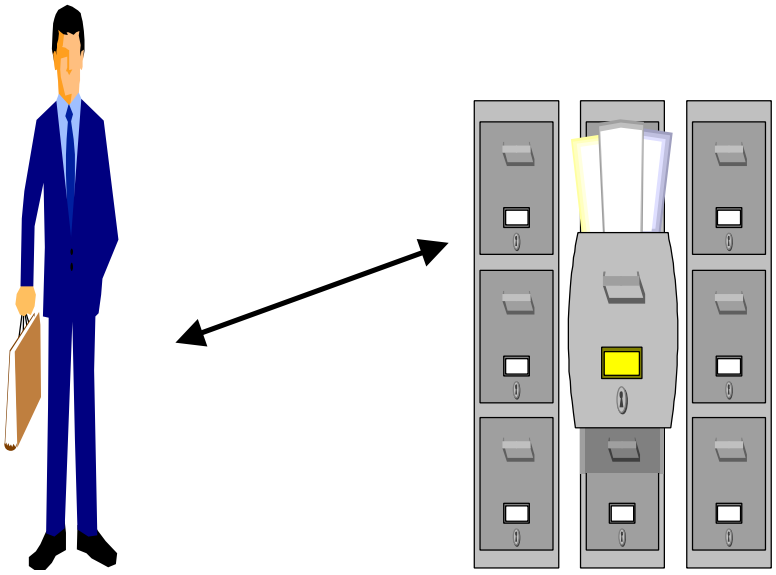
Er möchte folgende Angaben für jede Adresse vorfinden:

Name, **Vorname**, **PLZ**, **Ort** und **Telefon-Nummer**.



1. Entwicklungsschritt

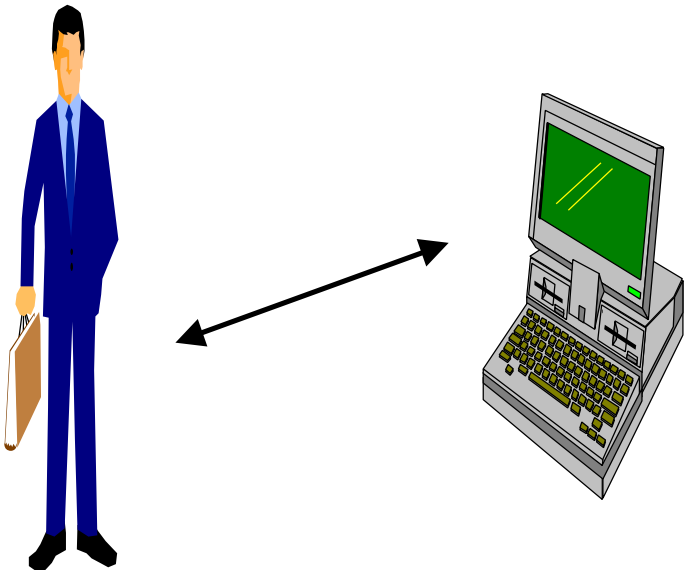
Herr X kauft sich ***Karteikarten*** und einen ***Karteikasten***.
Er verwaltet die Adressen in "traditioneller" Art und Weise.



- **Anmerkung**
  - Diese Lösung ist sicherlich nicht schlecht und praktikabel, aber für uns wenig lehrreich !

2. Entwicklungsschritt

Herr X kauft sich einen **Computer**.
Er trägt die **Adressen** mit einem Text-Editor in eine **Datei** ein.

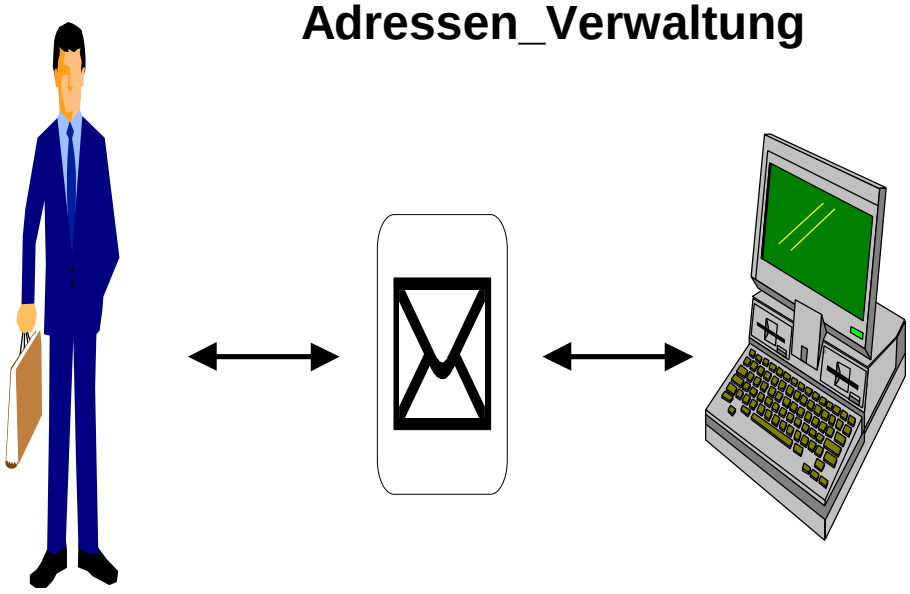


- **Anmerkung**
  - Suchen und Sortieren ist eher mühsam.
  - Daten können leicht ***unbeabsichtigt zerstört*** oder ***verfälscht*** werden (cut & paste)

3. Entwicklungsschritt

Herr X entwickelt ein **Programm**,
mit dem er einfach
die PLZ, den Ort und die Telefon-Nr
einer Adresse **ändern** kann.

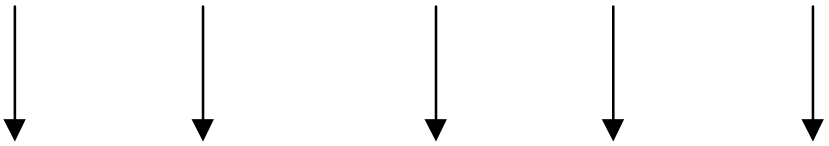
Adressen_Verwaltung



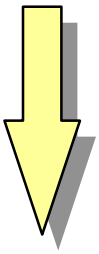
Erste Realisierungsidee

Eine Adresse

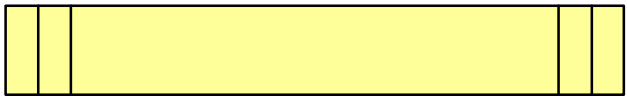
Kugler	Hans	8021	Zürich	01-4330570



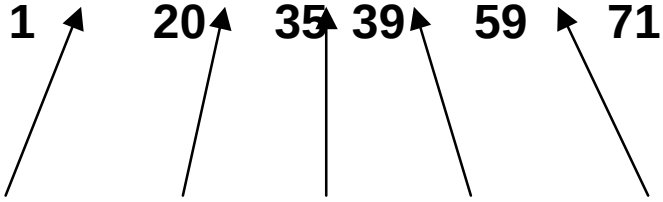
20 Zeichen 15 Zeichen 4 Zeichen 20 Zeichen 12 Zeichen



STRING



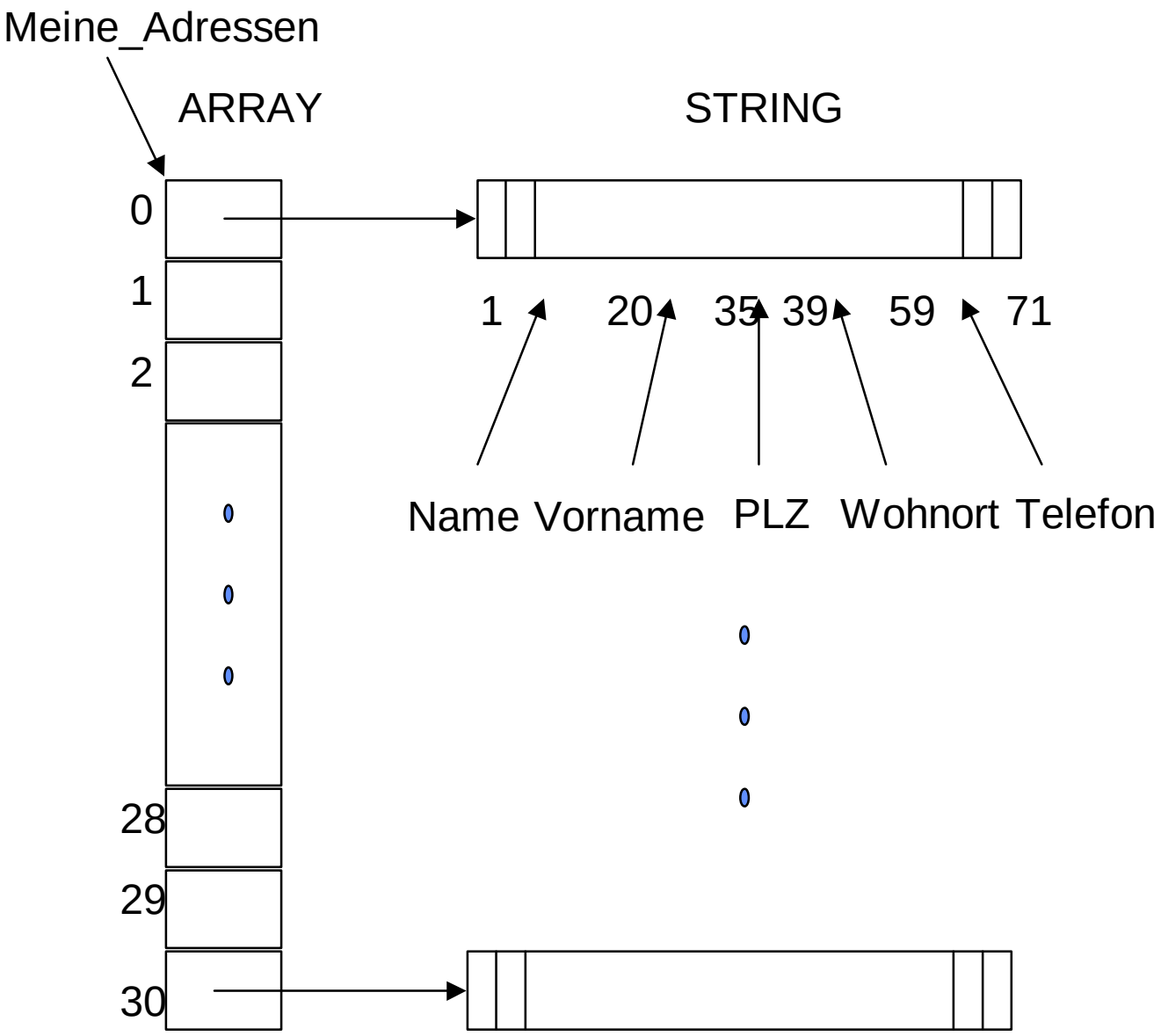
1 20 35 39 59 71



Name Vorname PLZ Wohnort Telefon

Entworfene Datenstruktur

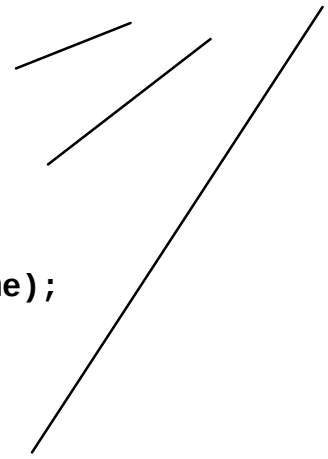
Variable



Erstes Hauptprogramm -1

```
MODULE Adressen_1 EXPORTS Main;

IMPORT SIO, IO, Wr, Rd, Text, Fmt;

CONST Geschaefts_Adr : TEXT = "gesch.adr";
      Max_Adr = 30;

TYPE Adresse = ARRAY [1 .. 71] OF CHAR;         (* 71 Zeichen*)
     Adressen_Liste = ARRAY [1..Max_Adr] OF Adresse;

VAR Meine_Adressen : Adressen_Liste;
    Name, Ort, Vorname,
    PLZ, Tel, Leer : TEXT;
    Gefunden : BOOLEAN := FALSE;
    Anz,I    : INTEGER;
    Adress_Datei_lesen     : Rd.T;
    Adress_Datei_schreiben : Wr.T;

(* Operationen zum Arbeiten mit Arrays *)
PROCEDURE Insert(VAR Adr : Adresse; start,stop : INTEGER; was: TEXT) =
PROCEDURE Get(Adr : Adresse; start,stop : INTEGER): TEXT =
PROCEDURE FillBlanks (t : TEXT) : TEXT =

BEGIN (*Adressen_1 *)
  Adress_Datei_lesen := IO.OpenRead(Geschaefts_Adr);
  Anz := 0;

  WHILE NOT SIO.EOF(Adress_Datei_lesen) DO
    Name := SIO.GetLine(Adress_Datei_lesen);
    Insert(Meine_Adressen[Anz+1], 1, 20, Name);

    Vorname := SIO.GetLine(Adress_Datei_lesen);
    Insert(Meine_Adressen[Anz+1], 21, 35, Vorname);

    PLZ := SIO.GetLine(Adress_Datei_lesen);
    Insert(Meine_Adressen[Anz+1], 36, 39, PLZ);

    Ort := SIO.GetLine(Adress_Datei_lesen);
    Insert(Meine_Adressen[Anz+1], 40, 59, Ort);

    Tel := SIO.GetLine(Adress_Datei_lesen);
    Insert(Meine_Adressen[Anz+1], 60, 71, Tel);

    Leer := SIO.GetLine(Adress_Datei_lesen);
    Anz := Anz + 1;
  END;

  Rd.Close (Adress_Datei_lesen);
```

direkter Eingriff



Erstes Hauptprogramm -2

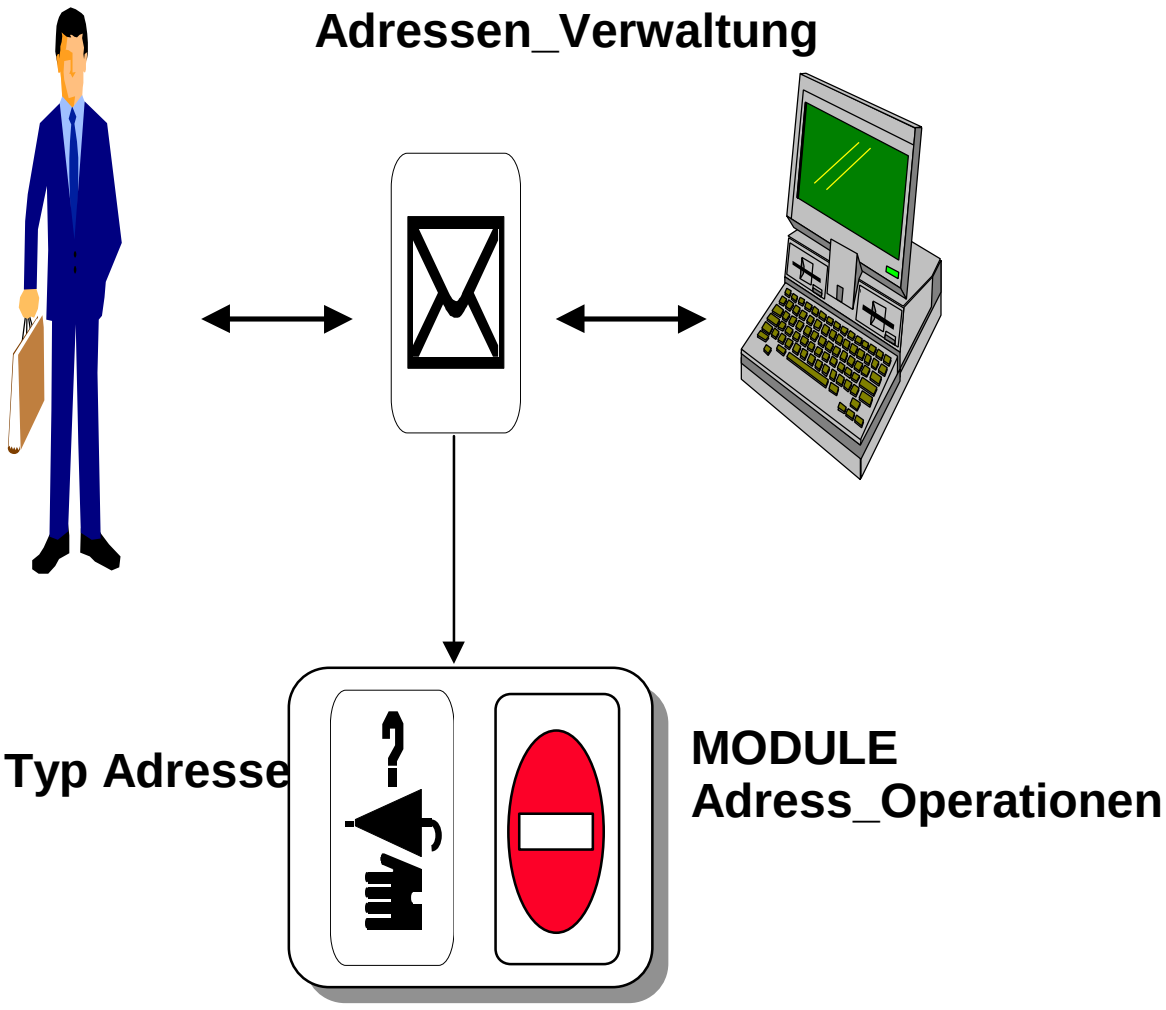
```
SIO.PutText ("Den zu suchenden Namen bitte > ");  (* Namen einlesen*)
Name := SIO.GetLine();
I := 1;
WHILE (NOT Gefunden) AND (I <= Anz) DO
  Gefunden := Text.Equal(Get(Meine_Adressen[I],1,20),
                         FillBlanks(Name));
  I := I + 1;
END;
(* ------------------------------------------------------------------*)
IF Gefunden THEN
  SIO.PutLine(Get(Meine_Adressen[I-1],1,20));     (* Daten anzeigen*)
  SIO.PutLine(Get(Meine_Adressen[I-1],21,35));
  SIO.PutLine(Get(Meine_Adressen[I-1],36,39));
  SIO.PutLine(Get(Meine_Adressen[I-1],40,59));
  SIO.PutLine(Get(Meine_Adressen[I-1],60,71));

  SIO.PutLine("\nDie neuen Angaben bitte!");
  SIO.PutText ("PLZ > ");                              (* PLZ *)
  PLZ:=SIO.GetLine ();
  Insert(Meine_Adressen[I-1],36,39,PLZ);

  SIO.PutText("Ort > ");                               (* Wohnort*)
  Ort:=SIO.GetLine ();
  Insert(Meine_Adressen[I-1],40,59,Ort);

  SIO.PutText("Telefon-Nr. > ");                       (* Telefon*)
  Tel:=SIO.GetLine ();
  Insert(Meine_Adressen[I-1],60,71,Tel);

  SIO.PutLine("\nGeaenderte Daten :");            (* Daten anzeigen*)
  SIO.PutLine(Get(Meine_Adressen[I-1],1,20));
  SIO.PutLine(Get(Meine_Adressen[I-1],21,35));
  SIO.PutLine(Get(Meine_Adressen[I-1],36,39));
  SIO.PutLine(Get(Meine_Adressen[I-1],40,59));
  SIO.PutLine(Get(Meine_Adressen[I-1],60,71));
ELSE
  SIO.PutLine("Spezifizierter Datensatz nicht vorhanden!");
END;
(* ------------------------------------------------------------------*)
Adress_Datei_schreiben := IO.OpenWrite(Geschaefts_Adr);
FOR J:=1 TO Anz DO
  SIO.PutLine(Get(Meine_Adressen[J], 1,20), Adress_Datei_schreiben);
  SIO.PutLine(Get(Meine_Adressen[J],21,35), Adress_Datei_schreiben);
  SIO.PutLine(Get(Meine_Adressen[J],36,39), Adress_Datei_schreiben);
  SIO.PutLine(Get(Meine_Adressen[J],40,59), Adress_Datei_schreiben);
  SIO.PutLine(Get(Meine_Adressen[J],60,71), Adress_Datei_schreiben);
  SIO.Nl(Adress_Datei_schreiben);
END;
Wr.Close (Adress_Datei_schreiben);
```

Monolithische Lösung

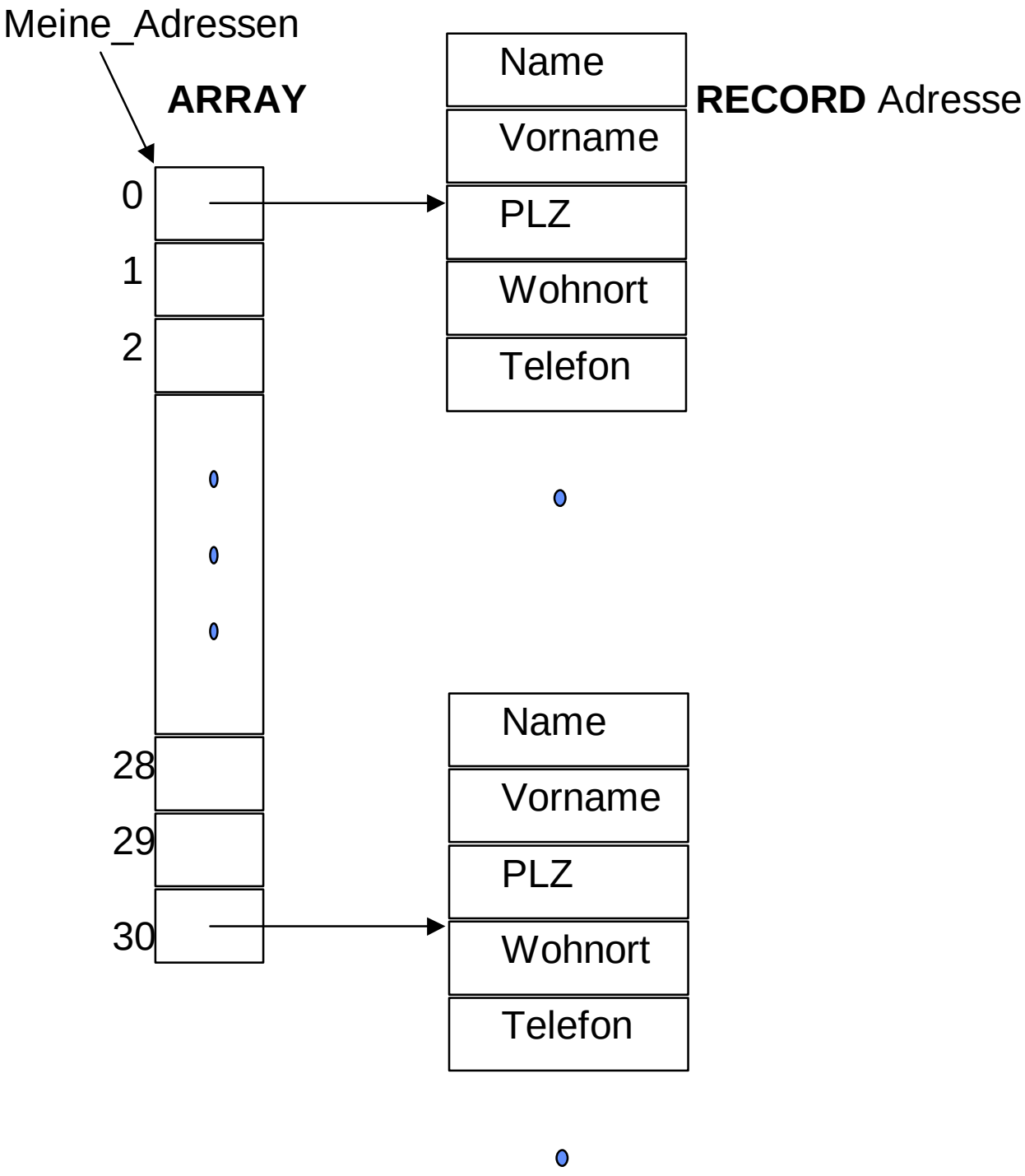
- **Folgendes ist zu kritisieren:**
  - Es wird ***direkt*** auf der gewählten Datenstruktur manipuliert.
  - Wenn der strukturelle Aufbau der Adressen ***verändert*** werden muß (z.B. Länge der PLZ = 5), müssen ***viele Stellen*** im Programm verändert werden.
  - Eine Änderung kann nicht an ***einer Stelle*** vorgenommen werden, sondern verteilt sich auf den gesamten Programmtext.
  - Es werden ***neue Fehler*** gemacht, wenn die notwendigen Änderungen durchgeführt werden.
  - Es werden Literale verwendet, Konstanten sind dafür geeigneter.
  - ...

4. Entwicklungsschritt

Herr X entscheidet sich, einen eigenen Typ **Adresse** zu entwerfen und diesen sowie die dazugehörenden **Manipulationsoperationen** in ein eigenes **Modul auszulagern** und dieses im Hauptprogramm zu **verwenden**.



Datenstruktur mit Record



Modul Adress_Operationen-1

```
INTERFACE Adress_Operationen;

IMPORT Rd,Wr;

TYPE Adresse = RECORD
       Name : TEXT;
       Vorname : TEXT;
       PLZ : TEXT;
       Ort : TEXT;
       Tel : TEXT;
     END;

CONST Max_Adr = 30;

TYPE Listen_Laenge = [1 .. Max_Adr];
     Adressen_Liste = ARRAY Listen_Laenge OF Adresse;

VAR  Meine_Adressen   : Adressen_Liste;  (* Datensaetze *)

(*-------------------------------------------------------------*)

PROCEDURE Setze_Name(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_Vorname(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_PLZ(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_Ort(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_Tel(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Hat_Namen(I: Listen_Laenge; Wert : TEXT): BOOLEAN;

PROCEDURE Zeige_Adresse(I: Listen_Laenge);

PROCEDURE Lese_Adresse(I: Listen_Laenge; File : Rd.T);

PROCEDURE Schreibe_Adresse(I: Listen_Laenge; File : Wr.T);

END Adress_Operationen.
```

Adressen-Typ

Zentrale Variable

Deklaration der Operationen

Zweites Hauptprogramm

```
MODULE Adressen_Verwaltung EXPORTS Main;
IMPORT Adress_Operationen AS AO;

BEGIN  (* -- des Hauptprogramms*)
  Adress_Datei_lesen := IO.OpenRead(Gesch_Adr);
  I := 0;
  WHILE NOT SIO.EOF(Adress_Datei_lesen) DO
    I := I + 1;
    AO.Lese_Adresse (I, Adress_Datei_lesen);
  END;
  Anz := I;
  Rd.Close(Adress_Datei_lesen);
(*-------------------------------------------------------------*)
  SIO.PutText ("Den zu suchenden Namen bitte > ");
  Name:=SIO.GetLine ();
  I := 1;
  WHILE (NOT Gefunden) AND (I <= Anz) DO   (* -- Daten suchen*)
    Gefunden := AO.Hat_Namen(I, Name);
    I := I + 1;
  END;
(*-------------------------------------------------------------*)
  IF Gefunden THEN
    AO.Zeige_Adresse(I-1);
    SIO.PutLine("Die neuen Angaben bitte!");
    SIO.PutText("PLZ > ");          (* PLZ *)
    PLZ := SIO.GetLine ();
    AO.Setze_PLZ(I-1, PLZ);

    SIO.PutText("Ort > ");          (* Wohnort*)
    Ort := SIO.GetLine ();
    AO.Setze_Ort(I-1, Ort);

    SIO.PutText("Telefon-Nr. > "); (* Telefon*)
    Tel := SIO.GetLine ();
    AO.Setze_Tel(I-1, Tel);

    AO.Zeige_Adresse(I-1);          (* Daten anzeigen*)
  ELSE
    SIO.PutLine("Spezifizierte Adresse nicht vorhanden!");
  END;
(*-------------------------------------------------------------*)
  Adress_Datei_schreiben := IO.OpenWrite (Gesch_Adr);
  FOR J := 1 TO Anz DO
    AO.Schreibe_Adresse(J, Adress_Datei_schreiben);
  END;
  Wr.Close (Adress_Datei_schreiben);

END Adressen_Verwaltung.
```

Import der Operationen

Aufruf der Operationen

Lösung mit Prozeduren

- **Es ist möglich, die definierten Operationen zu umgehen**
  - Die Operationen müssen im Hauptprogramm nicht verwendet werden; sie können verwendet werden!
- **Die direkte Manipulation der Variable `Meine_Adressen` ist im Hauptprogramm immer noch möglich!**

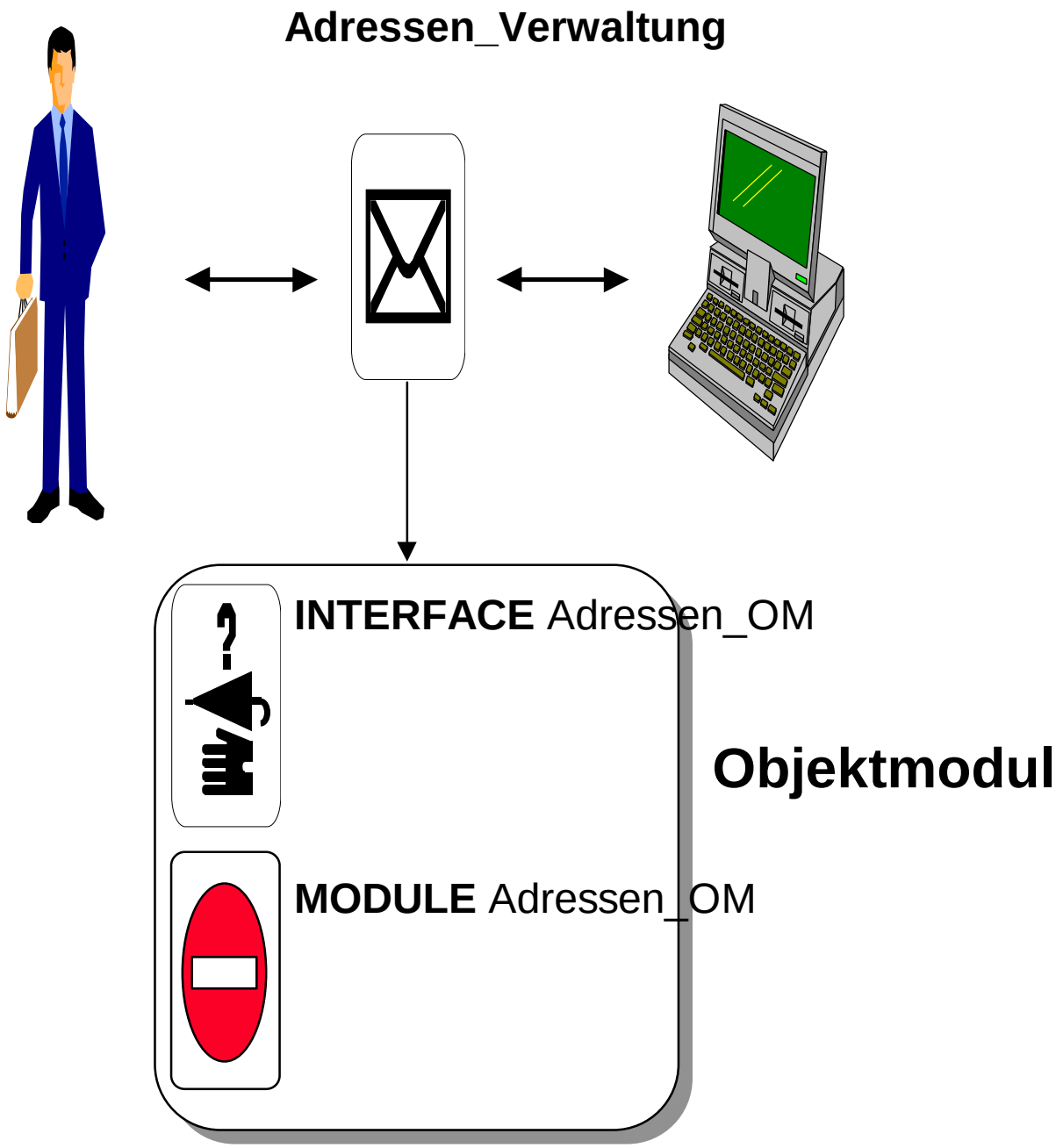
```
  SIO.PutLine("Die neuen Angaben
bitte!");
  SIO.PutText("PLZ > ");          (*
PLZ *)
  PLZ := SIO.GetLine ();
  AO.Setze_PLZ(I-1, PLZ);

  Meine_Adressen[I-1].PLZ :=
'abcd';
```

Aufbau der Datenstruktur ist bekannt. Sie kann direkt manipuliert werden!

5. Entwicklungsschritt

- **Herr X hat einen Software-Engineering Kurs gehört.**
  - Dort hat er gelernt, daß das ***Geheimnisprinzip*** angewendet werden muß, wenn Informationen geschützt werden sollen
  - Er entscheidet sich, die Variable `Meine_Adressen` durch ein ***Objektmodul*** (Datenkapsel) zu schützen.
- **Gleichzeitig soll sein Programm um folgende Funktionalität erweitert werden:**
  - es soll ein Datensatz ***eingegeben*** werden können
  - es soll nach einem Datensatz ***gesucht*** werden können (Name)
  - es soll ein Datensatz ***modifiziert*** werden können
  - es soll ein Datensatz ***angezeigt*** werden können
  - es soll auf den Anfang der Sammlung ***positioniert*** werden können
  - es soll ***geblättert*** werden können

Architektur mit Objektmodul



Schnittstelle des Objektmoduls

```
INTERFACE Adress_OM;
(* Dieses Modul realisiert ein Objektmodul, mit dem maximal
30 Adressen verwaltet werden koennen. Bevor eine Operation
des Objektmoduls ausgefuehrt werden kann, muss die Prozedur
Lese_Adressen ausgefuehrt worden sein.*)

PROCEDURE Neue_Adresse();
(* Fordert den Benutzer auf, eine neue Adresse einzugeben*)

PROCEDURE Modifiziere_Adresse();
(*Fordert den Benutzer auf, Angaben fuer die aktuelle
  Adresse einzugeben.*)

PROCEDURE Suche_Adresse();
(*Fordert den Benutzer auf, einen Namen einzugeben. Ist
eine Adresse mit diesem Namen vorhanden, wird diese
angezeigt. Diese Adresse wird die aktuell selektierte
Adresse.*)

PROCEDURE Lese_Adressen();
(*Initialisiert das Modul und liest alle Adressen von Datei
ein. Die erste Adresse ist die aktuell selektierte Adr.*)

PROCEDURE Schreibe_Adressen();
(*Schreibt die Adressen zurueck auf Datei.*)

PROCEDURE Zeige_Adresse();
(*Zeigt die aktuelle Adresse auf dem Bildschirm.*)

PROCEDURE Zuruecksetzen();
(*Es wird auf den Anfang der Adressensammlung positioniert.
Die erste Adresse wird die aktuelle Adresse.*)

PROCEDURE Blaettern();
(*Zeigt ausgehend von der aktuellen Adresse die naechste
Adresse am Bildschirm.*)

END Adress_OM.
```

Hauptprogramm 3

```
MODULE Adress_Verwaltung EXPORTS Main;
IMPORT Adress_OM AS Adr;
(*-------------------------------------------------*)
PROCEDURE Zeige_Adress_Menue(): CHAR =
BEGIN
  SIO.PutLine("--------------------");
  SIO.PutLine("Zuruecksetzen (r)");
  SIO.PutLine("Blaettern     (b)");
  SIO.PutLine("Suchen        (s)");
  SIO.PutLine("Aendern       (a)");
  SIO.PutLine("Eintragen     (e)");
  SIO.PutLine("Zeigen        (z)");
  SIO.PutLine("BEENDEN       (sonst)");
  SIO.PutLine("--------------------");
  SIO.PutText("AUSWAHL > ");
  RETURN Text.GetChar(SIO.GetLine(), 0);
END Zeige_Adress_Menue;

PROCEDURE Menu() =
VAR Wahl : CHAR := 'r';
    Kommandos := SET OF CHAR {'r','b','s','a','z','e'};
BEGIN
  REPEAT
    Wahl := Zeige_Adress_Menue();
    IF Wahl IN Kommandos THEN
      CASE Wahl OF
         'r'    => Adr.Zuruecksetzen();       |
         'b'    => Adr.Blaettern();           |
         's'    => Adr.Suche_Adresse();       |
         'a'    => Adr.Modifiziere_Adresse(); |
         'z'    => Adr.Zeige_Adresse();       |
         'e'    => Adr.Neue_Adresse();
      END;
    ELSE
      SIO.PutLine ("ENDE DES PROGRAMMS");
    END;
  UNTIL NOT Wahl IN Kommandos;
END Menu;
(*-------------------------------------------------------*)
BEGIN (* des Hauptprogramms*)
  Adr.Lese_Adressen();
  Menu();
  Adr.Schreibe_Adressen();
END Adress_Verwaltung.
```

Impl. desObjektmoduls - 1

```
MODULE Adress_OM;

IMPORT IO, Rd, Wr,Text;

TYPE Adresse = RECORD
       Name    : TEXT;
       Vorname : TEXT;
       PLZ     : TEXT;
       Ort     : TEXT;
       Tel     : TEXT;
     END;

CONST Adr_Datei_Name = "gesch.adr";
      Max = 30;

TYPE
  Listen_Laenge = [1..Max];
  Adressen_Liste = ARRAY Listen_Laenge OF Adresse;

(*-----------------------------------------------*)
(* gekapselte Daten*)

VAR Meine_Adressen : Adressen_Liste;
    Letzte_Adresse : Listen_Laenge := 1;
    Akt_Adresse    : Listen_Laenge := 1;
```

Impl. desObjektmoduls - 2

```
PROCEDURE Lese_Adressen() =
VAR I : Listen_Laenge;
    Zeile : TEXT;
    Adr_Datei_lesen : Rd.T;
BEGIN
  Adr_Datei_lesen := IO.OpenRead (Adr_Datei_Name);
  I := 1;
  WHILE NOT IO.EOF (Adr_Datei_lesen) DO
    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Name := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Vorname :=  Zeile;
    ...
    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Ort := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Tel := Zeile;

    Zeile:=IO.GetLine(Adr_Datei_lesen);   (* Leerzeile*)
    INC(I);
  END;
  Letzte_Adresse := I - 1; Akt_Adresse := 1;
  Rd.Close (Adr_Datei_lesen);
END Lese_Adressen;

PROCEDURE Zuruecksetzen() =
BEGIN
  Akt_Adresse := 1; Zeige_Adresse();
END Zuruecksetzen;

PROCEDURE Blaettern() =
BEGIN
  IF Akt_Adresse < Letzte_Adresse THEN
    INC(Akt_Adresse);
    Zeige_Adresse();
  ELSE
    IO.Put("Kein Datensatz mehr vorhanden!\n");
  END;
END Blaettern;
```

Impl.Objektmoduls - 3

```
PROCEDURE Gebe_Adresse_Aus (Adr: Adresse;
                            Seperator :TEXT;
                            Wo : Wr.T := NIL) =
BEGIN
IO.Put(Adr.Name    & Seperator &
       Adr.Vorname & Seperator &
       Adr.PLZ     & Seperator &
       Adr.Ort     & Seperator &
       Adr.Tel     & Seperator , Wo );
END Gebe_Adresse_Aus;

PROCEDURE Zeige_Adresse() =
BEGIN
  IO.Put("\n");
  Gebe_Adresse_Aus(Meine_Adressen[Akt_Adresse], " ");
  IO.Put("\n");
END Zeige_Adresse;

PROCEDURE Schreibe_Adressen() =
VAR Adr_Datei_schreiben: Wr.T;
BEGIN
  Adr_Datei_schreiben := IO.OpenWrite (Adr_Datei_Name);
  FOR I:= 1 TO Letzte_Adresse DO
    Gebe_Adresse_Aus (Meine_Adressen[I],
                      "\n",
                      Adr_Datei_schreiben);
    IO.Put("\n",Adr_Datei_schreiben);
  END;
  Wr.Close (Adr_Datei_schreiben);
END Schreibe_Adressen;
```

6. Entwicklungsschritt

Da Herr X mit der Verwaltung der Adressen seiner **Geschäftspartner** sehr zufrieden ist, möchte er die Adressen seiner **privaten Bekannten** ebenfalls mit seinem Programm verwalten.

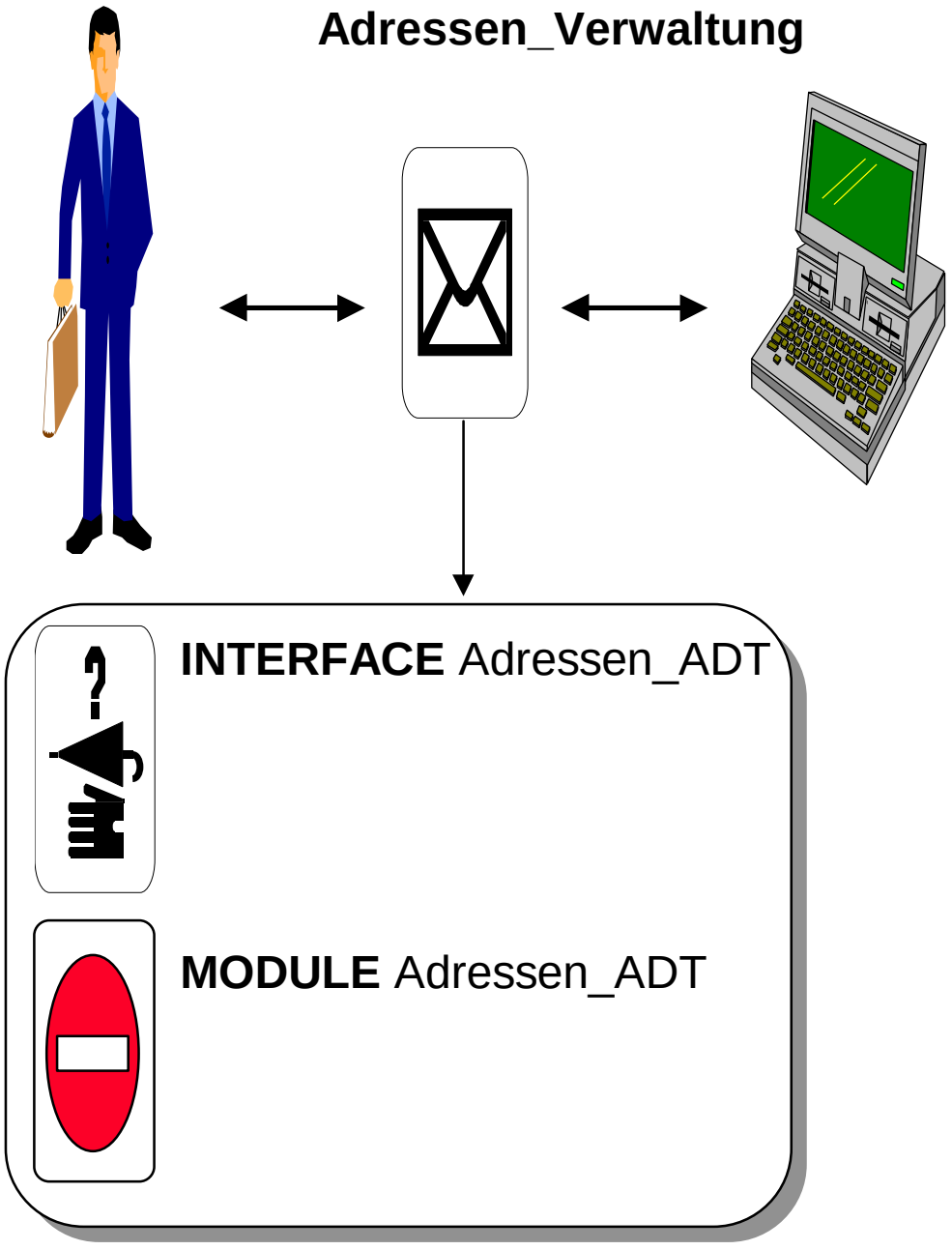
Wie kann er dies einfach erreichen?

- Er kann den Programmcode ***duplizieren*** und eine neue Datei für die Adressen seiner privaten Bekannten verwenden.
- Er kann sein Programm mit der Adressen-Datei ***parametrisieren***.
- Er kann seine Datenkapsel zu einem ***Abstrakten Datentyp*** erweitern.

Vom Objektmodul zum ADT

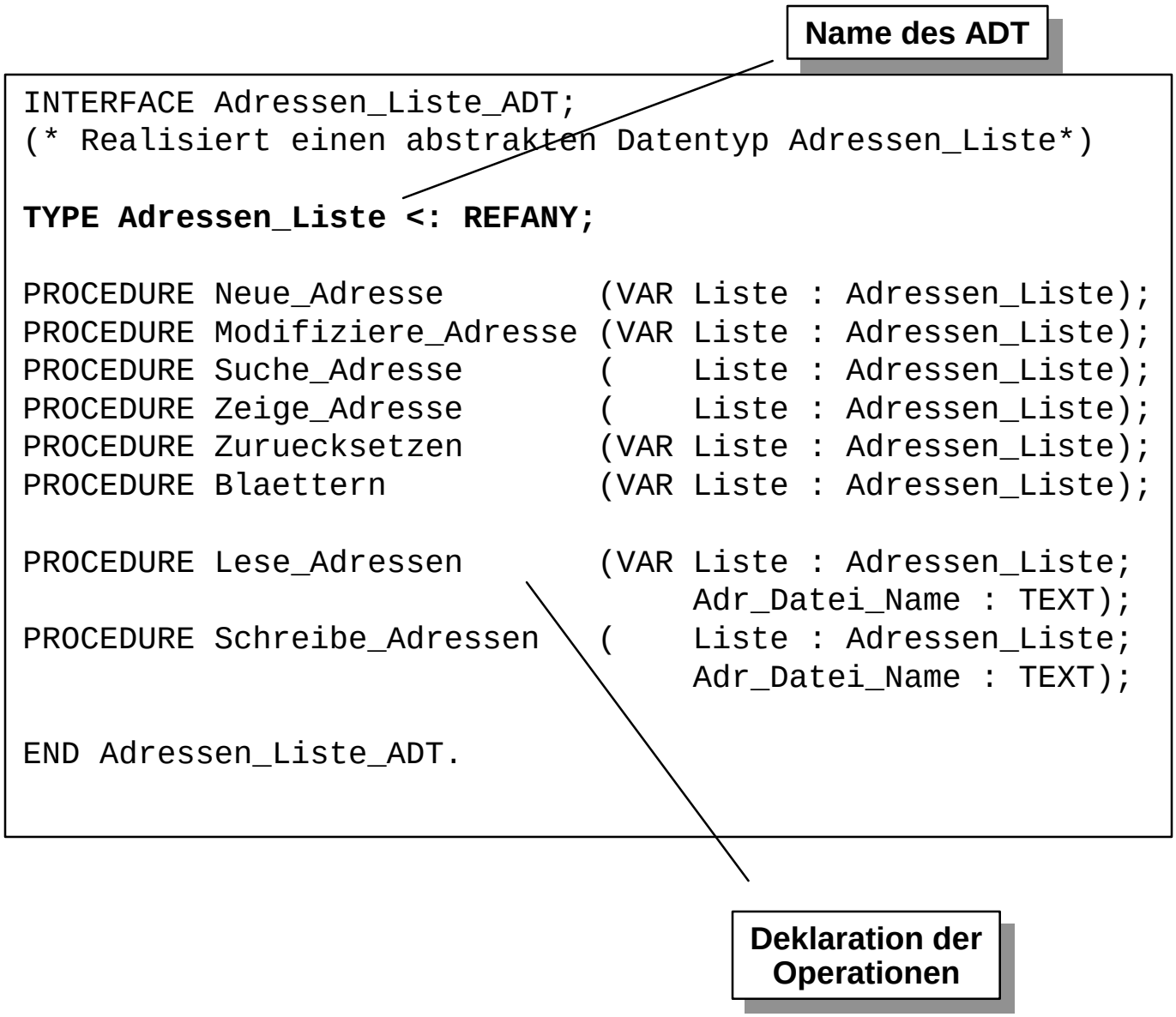
- **Gekapselte Daten werden zu einem Typ zusammengefaßt**
  - typischerweise ein Record
- **Die Operationen des Objektmoduls werden um einen zusätzlichen Parameter erweitert.**
  - Dieser Parameter ist eine ADT-Exemplar und steht per Konvention an erster Stelle.
- **In den Operations-Rümpfen muß der neue Parameter entsprechend verwendet werden.**
- **Exemplare des ADT´s werden im Hauptprogramm erzeugt und die Operationsaufrufe entsprechend angepaßt.**

Architektur mit einem ADT



ADT
Adressen_Liste

Der ADT Adressen_Liste-1



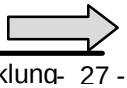
Viertes Hauptprogramm-1

```
MODULE Adress_Verwaltung EXPORTS Main;
(*Testprogramm fuer den Abstrakten Datentypen.*)

IMPORT SIO, Text;
IMPORT Adressen_Liste_ADT AS AL;

CONST Private_Adr    = "priv.adr";
      Geschaefts_Adr = "gesch.adr";

VAR P_Adressen : AL.Adressen_Liste;
    G_Adressen : AL.Adressen_Liste;

…
PROCEDURE Interaktion()=
…

(*-----------------------------------------------------*)
BEGIN (* des Hauptprogramms*)

  AL.Lese_Adressen(P_Adressen, Private_Adr);
  AL.Lese_Adressen(G_Adressen, Geschaefts_Adr);

  Interaktion();

  AL.Schreibe_Adressen(P_Adressen, Private_Adr);
  AL.Schreibe_Adressen(G_Adressen, Geschaefts_Adr);

END Adress_Verwaltung.
```

Import des ADTs

Deklaration von Variablen des ADTs

Verwenden der Operationen des ADTs



Viertes Hauptprogramm-2

```
PROCEDURE Zeige_Auswahl_Menue (): CHAR =
BEGIN
  SIO.PutLine("Private Adressen    (a)");
  SIO.PutLine("Geschaefts Adressen (b)");
  SIO.PutLine("BEENDEN             (sonst)");
  SIO.PutText("AUSWAHL > ");
  RETURN Text.GetChar(SIO.GetLine(), 0);
END Zeige_Auswahl_Menue;

(*-------------------------  --------------------------*)

PROCEDURE Interaktion()=
  VAR Auswahl : CHAR := 'r';
BEGIN
  REPEAT
    Auswahl:= Zeige_Auswahl_Menue();
    CASE Auswahl OF
      'a' => Menue(P_Adressen, "PRIVATE-ADRESSEN");    |
      'b' => Menue(G_Adressen, "GESCHAEFTS-ADRESSEN");
    ELSE
      SIO.PutLine ("ENDE DES PROGRAMMS!");
    END;
  UNTIL (Auswahl # 'a') AND (Auswahl # 'b');
END Interaktion;
```

Viertes Hauptprogramm-3

```
PROCEDURE Zeige_Adress_Menue(): CHAR =
BEGIN
  SIO.PutLine("--------------------");
  SIO.PutLine("Zuruecksetzen (r)");
  SIO.PutLine("Blaettern     (b)");
  SIO.PutLine("Suchen        (s)");
  SIO.PutLine("Aendern       (a)");
  SIO.PutLine("Eintragen     (e)");
  SIO.PutLine("Zeigen        (z)");
  SIO.PutLine("BEENDEN       (sonst)");
  SIO.PutLine("--------------------");
  SIO.PutText("AUSWAHL > ");
  RETURN Text.GetChar(SIO.GetLine(), 0);
END Zeige_Adress_Menue;

(*-------------------------------------------------------*)
PROCEDURE Menue(VAR Adress_Liste : AL.Adressen_Liste;
                    Ueberschrift : TEXT) =
VAR Wahl : CHAR;
    Kommandos := SET OF CHAR {'r','b','s','a','z','e'};
BEGIN
  REPEAT
    SIO.PutLine("--------------------");
    SIO.PutLine(Ueberschrift);

    Wahl := Zeige_Adress_Menue();

    IF Wahl IN Kommandos THEN
      CASE Wahl OF
       'r'    => AL.Zuruecksetzen      (Adress_Liste); |
       'b'    => AL.Blaettern          (Adress_Liste); |
       's'    => AL.Suche_Adresse      (Adress_Liste); |
       'a'    => AL.Modifiziere_Adresse(Adress_Liste); |
       'z'    => AL.Zeige_Adresse      (Adress_Liste); |
       'e'    => AL.Neue_Adresse       (Adress_Liste);
      END;
    ELSE
      SIO.PutLine ("<---");
    END;
  UNTIL NOT Wahl IN Kommandos;
END Menue;
```

Verwenden der Operationen des ADTs

Blick ins Innere des ADTs - 2

```
MODULE Adressen_Liste_ADT;

IMPORT IO, Rd, Wr, Text;

TYPE Adresse = RECORD
       Name : TEXT;
       Vorname : TEXT;
       PLZ : TEXT;
       Ort : TEXT;
       Tel : TEXT;
     END;

CONST Max = 30;

TYPE
  Listen_Laenge = [1..Max];

  AdressenSpeicher = RECORD
    Letzte_Adresse : Listen_Laenge := 1;
    Akt_Adresse    : Listen_Laenge := 1;
    Adressen       : ARRAY Listen_Laenge OF Adresse;
  END;

  REVEAL
    Adressen_Liste = BRANDED REF AdressenSpeicher;
```

Definition der Struktur des ADTs

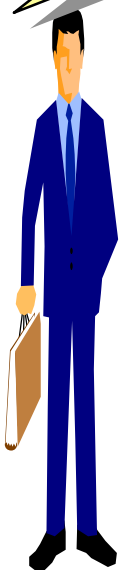
Blick ins Innere des ADTs - 2

```
PROCEDURE Lese_Adressen(VAR Liste: Adressen_Liste;
                        Adr_Datei_Name : TEXT) =
VAR Zeile : TEXT;
    Adr_Datei_lesen : Rd.T;
BEGIN
  Liste:=NEW(Adressen_Liste);
  Adr_Datei_lesen := IO.OpenRead (Adr_Datei_Name);
  Liste.Letzte_Adresse := 1;
  WHILE NOT IO.EOF (Adr_Datei_lesen) DO
    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Name := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Vorname := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].PLZ := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Ort := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Tel := Zeile;

    Zeile:=IO.GetLine(Adr_Datei_lesen);    (* Leerzeile *)
    INC(Liste.Letzte_Adresse);
  END;
  Liste.Akt_Adresse := 1;
  DEC(Liste.Letzte_Adresse);
  Rd.Close (Adr_Datei_lesen);
END Lese_Adressen;
```

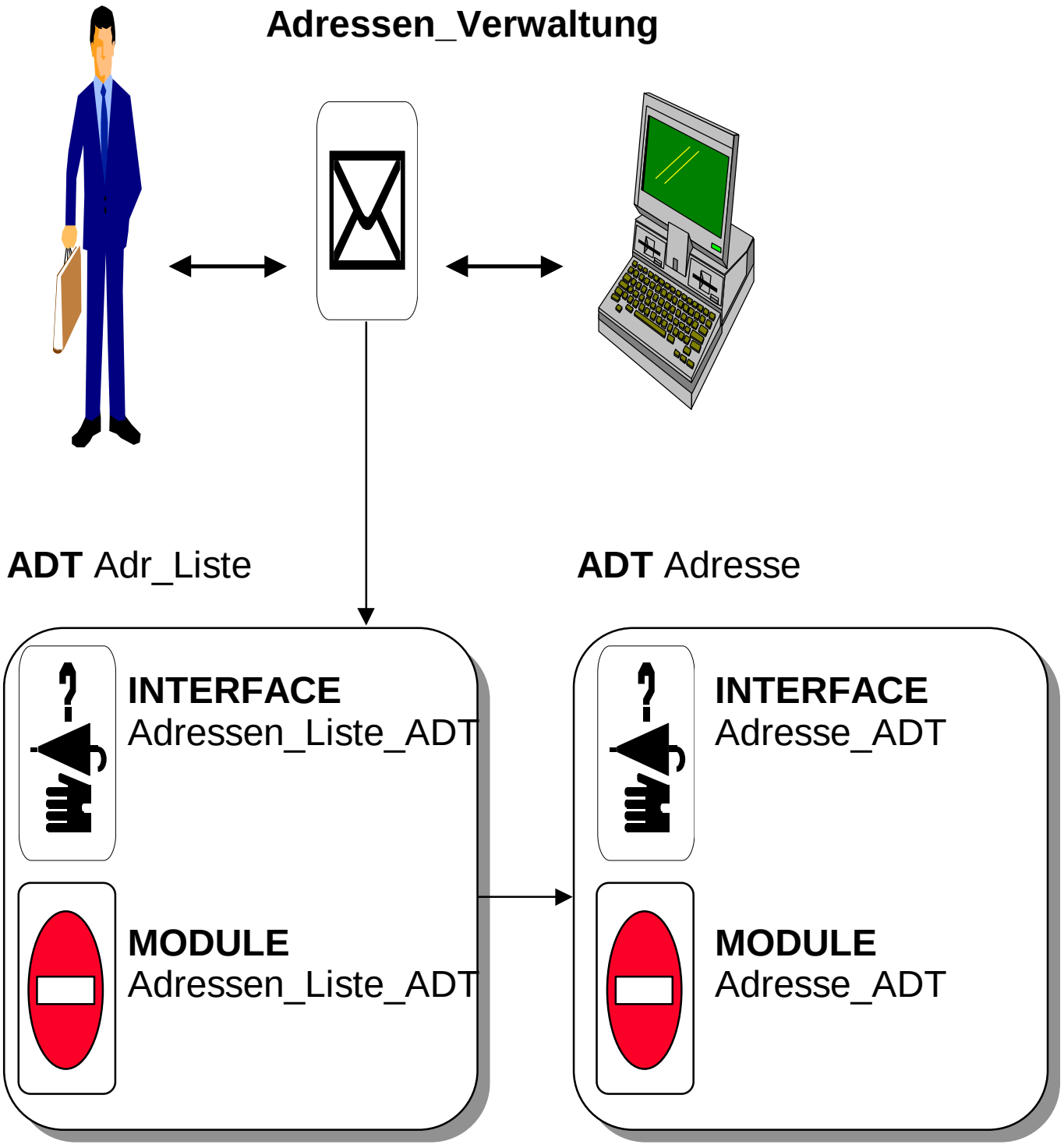
7. Entwicklungsschritt

Fasziniert vom Konzept der ADTen
beschließt Herr X
den Typ ADRESSE aus dem Paket
Adressen_ADT zu **extrahieren**
und einen **eigenen ADT ADRESSE** zu bilden

Damit wird das Programm besser erweiterbar und wartbar!



Architektur mit zwei ADTen

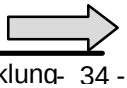


Der ADT Adresse-1

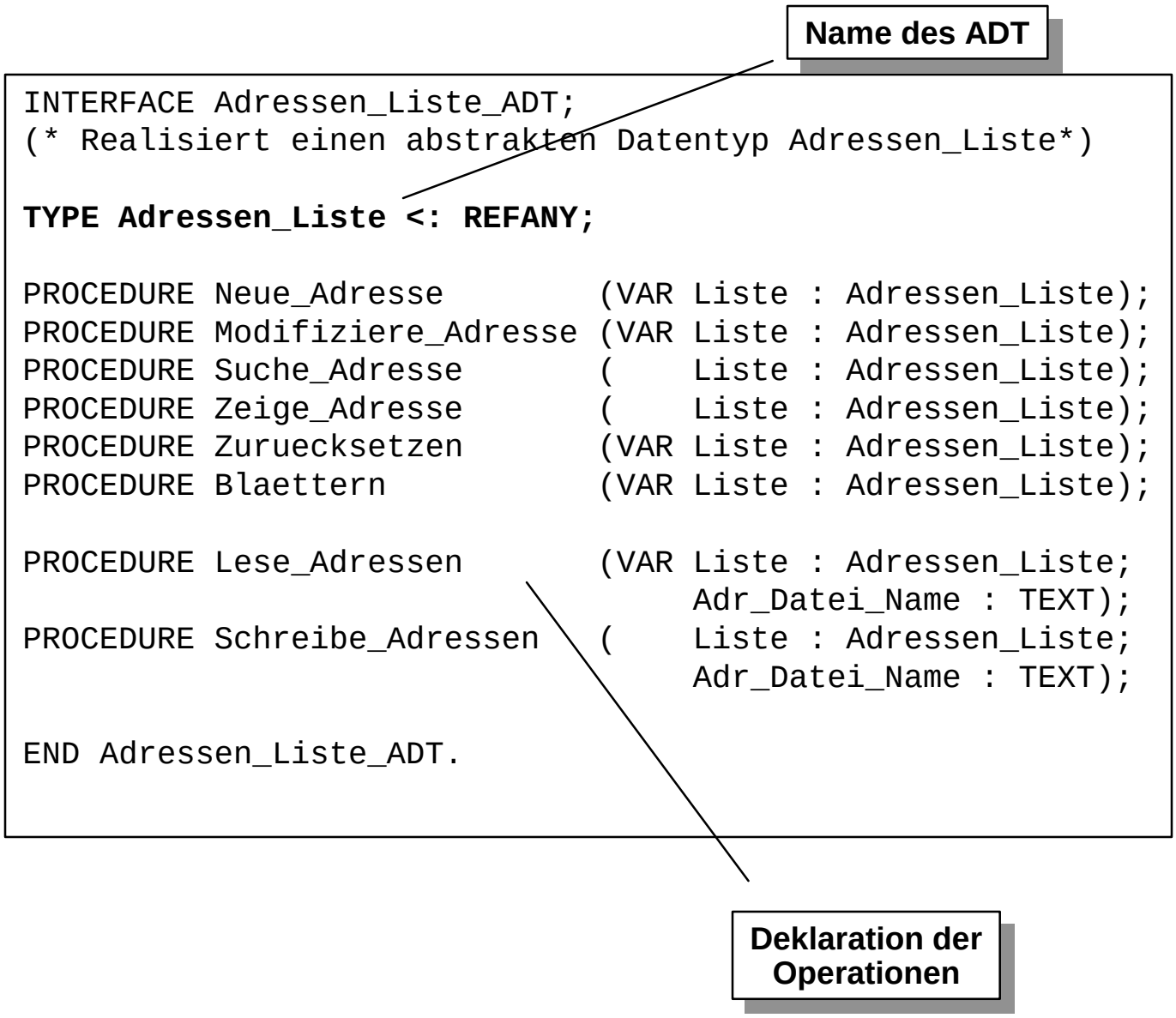
```
INTERFACE Adresse_ADT;
(* realisiert den ADT Adresse mit folgenden Operationen*)

IMPORT Rd, Wr;

TYPE Adresse <: REFANY;

PROCEDURE Lese_Eine_Adresse(VAR Datei: Rd.T) : Adresse;
(* liest eine Adresse von der angegebenen Datei und
   liefert diese zurueck *)

PROCEDURE Schreibe_Eine_Adresse(Adr: Adresse; VAR Datei: Wr.T);
(* schreibt die Adresse Adr auf Datei *)

PROCEDURE Zeige_Eine_Adresse(Adr: Adresse);
(*  zeigt die Adresse Adr am Bildschirm an *)

PROCEDURE Lese_Eine_Neue_Adresse():Adresse;
(* Fordert den Benutzer auf, Angaben fuer ein neue Adresse
   einzugeben und liefert die eingegebene Adresse zurueck. *)

PROCEDURE Modifiziere_Eine_Adresse(VAR Adr: Adresse);
(* Fordert den Benutzer auf, neue Angaben fuer die Adresse Adr
   einzugeben.*)

PROCEDURE Liefere_Suchattribut():TEXT;
(* Liefert das Suchattribute fuer Adressen.*)

PROCEDURE Hat_Namen (Adr: Adresse;  Name: TEXT):BOOLEAN;
(* Testet, ob die Adresse Adr den angegebene Namen hat.*)

END Adresse_ADT.
```

Name des ADT

Deklaration der Operationen



Der ADT Adressen_Liste -1



Der ADT Adressen_Liste -2

```
MODULE Adressen_Liste_ADT;

IMPORT IO, Rd, Wr;
IMPORT Adresse_ADT AS Adr;

CONST Max = 30;

TYPE Listen_Laenge = [1..Max];
     AdressenSpeicher = RECORD
       Letzte_Adresse : Listen_Laenge := 1;
       Akt_Adresse    : Listen_Laenge := 1;
       Adressen       : ARRAY Listen_Laenge OF Adr.Adresse;
     END;

     REVEAL Adressen_Liste = BRANDED REF AdressenSpeicher;

(*-----------------------------------------------------------*)

PROCEDURE Zeige_Adresse(Liste: Adressen_Liste) =
BEGIN
  IO.Put("\n");
  Adr.Zeige_Eine_Adresse(Liste.Adressen[Liste.Akt_Adresse]);
  IO.Put("\n");
END Zeige_Adresse;

PROCEDURE Zuruecksetzen(VAR Liste: Adressen_Liste) =
BEGIN
  Liste.Akt_Adresse := 1;
  Adr.Zeige_Eine_Adresse(Liste.Adressen[Liste.Akt_Adresse]);
END Zuruecksetzen;

PROCEDURE Blaettern(VAR Liste: Adressen_Liste) =
BEGIN
  IF Liste.Akt_Adresse < Liste.Letzte_Adresse THEN
    INC(Liste.Akt_Adresse);
    Adr.Zeige_Eine_Adresse(Liste.Adressen[Liste.Akt_Adresse]);
  ELSE
    IO.Put("Kein Datensatz mehr vorhanden!\n");
  END;
END Blaettern;
```

Import ADT Adresse

Der ADT Adressen_Liste -3

```
PROCEDURE Suche_Adresse(Liste: Adressen_Liste) =
VAR Suchbegriff : TEXT;
       Gefunden : BOOLEAN := FALSE;
BEGIN
  IO.Put("Es kann nur nach dem Attribut: " &
          Adr.Liefere_Suchattribut()      &
          " gesucht werden!\n");
  IO.Put ("Attributwert > ");
  Suchbegriff := IO.GetLine ();
  Liste.Akt_Adresse := 1;
  WHILE (NOT Gefunden) AND
        (Liste.Akt_Adresse <= Liste.Letzte_Adresse) DO
    IF Adr.Hat_Namen(Liste.Adressen[Liste.Akt_Adresse],
                     Suchbegriff) THEN
      Gefunden := TRUE;
      Zeige_Adresse(Liste);
    ELSE
      INC(Liste.Akt_Adresse);
    END;
  END;
  IF NOT Gefunden THEN
    IO.Put("Der spezifizierte Datensatz ist nicht vorhanden!\n");
  END;
END Suche_Adresse;

(*-------------------------------------------------------------*)

PROCEDURE Lese_Adressen(VAR Liste: Adressen_Liste;
                            Adr_Datei_Name : TEXT) =
VAR Adr_Datei_lesen : Rd.T;
BEGIN
  Liste:= NEW(Adressen_Liste);
  Adr_Datei_lesen := IO.OpenRead (Adr_Datei_Name);
  Liste.Letzte_Adresse := 1;
  WHILE NOT IO.EOF (Adr_Datei_lesen) DO
    Liste.Adressen[Liste.Letzte_Adresse]:=
                    Adr.Lese_Eine_Adresse(Adr_Datei_lesen);
    INC(Liste.Letzte_Adresse);
  END;
  DEC(Liste.Letzte_Adresse);
  Liste.Akt_Adresse := 1;
  Rd.Close (Adr_Datei_lesen);
END Lese_Adressen;
```

Der ADT Adressen_Liste -4

```
PROCEDURE Schreibe_Adressen( Liste: Adressen_Liste;
                             Adr_Datei_Name: TEXT) =
VAR Adr_Datei_schreiben:Wr.T;
BEGIN
  Adr_Datei_schreiben := IO.OpenWrite (Adr_Datei_Name);
  FOR I:= 1 TO Liste.Letzte_Adresse DO
    Adr.Schreibe_Eine_Adresse(Liste.Adressen[I,
                              Adr_Datei_schreiben);
  END;
  Wr.Close (Adr_Datei_schreiben);
END Schreibe_Adressen;
(*-------------------------------------------------------------*)

PROCEDURE Neue_Adresse(VAR Liste: Adressen_Liste) =
BEGIN
  IF Liste.Letzte_Adresse = Max THEN
    IO.Put("Es ist leider keine Platz mehr!\n");
  ELSE
    INC(Liste.Letzte_Adresse);
    Liste.Akt_Adresse := Liste.Letzte_Adresse;
    Liste.Adressen[Liste.Akt_Adresse] :=
                                  Adr.Lese_Eine_Neue_Adresse();
    IO.Put("Eingegebene Adresse :\n");
    Zeige_Adresse(Liste);
  END;
END Neue_Adresse;

(*-------------------------------------------------------------*)

PROCEDURE Modifiziere_Adresse(VAR Liste:Adressen_Liste) =
BEGIN
  IO.Put("Adresse :");
  Zeige_Adresse(Liste);
  Adr.Modifiziere_Eine_Adresse
                           (Liste.Adressen[Liste.Akt_Adresse]);
  IO.Put("\nGeaenderte Adresse :");
  Zeige_Adresse(Liste);
END Modifiziere_Adresse;

BEGIN
END Adressen_Liste_ADT.
```

Vertragsmodell

- **Idee des Vertragsmodells**
- **Zusicherungen**
- **Realisierung von Zusicherungen mit Pragmas**
- **Ausnahmebehandlung**
- **Zusicherungen mittels Ausnahmebehandlung**

Erinnerung

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Feststellung:**
  - `LegeTextAb` und `Entnehme` sind ***nicht in jedem Zustand*** des Ordners sinnvoll:
    - ein voller Ordner kann keine weiteren Texte aufnehmen; ein leerer keine herausgeben.
  - Solche Operationen sind nur in ***bestimmten Situationen*** (abhängig von bestimmten Bedingungen) sinnvoll.
  - Um den sicheren Umgang mit einem solchen Objekt zu gewährleisten, stellen wir an der Schnittstelle entsprechende ***Testfunktionen*** wie `IstLeer` oder `IstVoll` zur Verfügung.

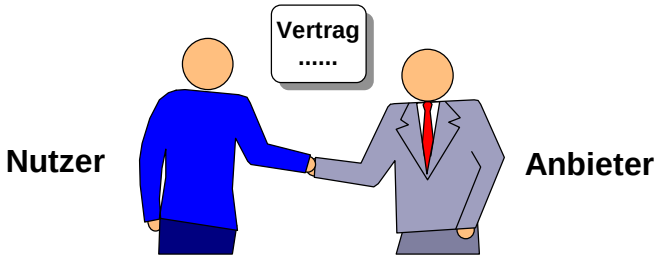
Einsatz der Testfunktionen

```
Ordner.Initialisiere;

IF Ordner.IstVoll() THEN
  SIO.PutLine ("Ordner ist bereits voll");
ELSE
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
END;

IF Ordner.IstLeer() THEN
  SIO.PutLine ("Ordner ist leer");
ELSE
  t := Ordner.EntnehmeText();
END;
```

Prüfen, ob die Operation angewedet werden darf.

- **Frage:**
  - Wer soll ***sicherstellen***, daß eine Operation immer richtig angewendet wird?
  - ***Nutzer*** oder ***Anbieter***?

Idee des Vertragsmodells



- **Vertrag**
  - zwischen Nutzer und Anbieter einer Operation regelt, wer der beiden Partner welche ***Verpflichtungen*** einhalten muß (und welchen ***Nutzen*** er dadurch hat)
- **Operation**
  - arbeitet korrekt, wenn sie vertragsgemäß ***benutzt*** bzw. ***realisiert*** wird.
- **Frage**
  - Wie kann ein solcher Vertrag ***programmtechnisch*** realisiert werden?

Zusicherungen - 1

- **Zusicherungen**
  - sind eine Technik, um eine bestimmte Art von Verträgen zwischen Anbieter und Nutzer zu formulieren.
- **Zusicherungen werden formuliert als**
  - ***Vorbedingungen*** für Operationen
  - ***Nachbedingungen*** von Operationen
  - ***Invarianten*** von abstrakten Datentypen
- **Zusicherungen**
  - erhöhen die ***Benutzbarkeit***, indem sie diese formaler definieren
  - verbessern die ***Testbarkeit***
  - verbessern die ***Fehlersuche*** (debugging)
  - verlangen vom Entwickler ein ***abstraktes*** Denken

Zusicherungen - 2

- **Vorbedingung**
  - beschreibt eine Bedingung, die der ***Nutzer*** (Aufrufer) einer Operation ***einhalten*** muß, damit die Operation korrekt arbeitet
- **Nachbedingung**
  - beschreibt einen Zustand, der nach dem erfolgreichen Ausführen der Operation vorhanden ist
  - diese garantiert der ***Anbieter***

Die Technik der Zusicherungen sollte bei der Entwicklung von Programmkomponenten (Modulen) eingesetzt werden, die wiederver-wendet werden sollen!

Zusicherungen - erstes Beispiel

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
  requires NOT IstVoll(o)
  ensures  NOT IstLeer(o)

PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
  requires NOT IstLeer(o)
  ensures  NOT IstVoll(o)

PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
. . .

END OrdnerADT.
```

Vorbedingung

Nachbedingung

Verpflichtung <-> Nutzen

- **Vorbedingungen sind**
  - Verpflichtungen für den Benutzer
  - Nutzen für den Anbieter
- **Nachbedingungen sind**
  - Nutzen für den Benutzer
  - Verpflichtungen für den Anbieter

Operation `EntnehmeText`

	Verpflichtungen	Nutzen
Nutzer	Der Ordner darf nicht leer sein.	Das zuletzt eingegebene Text wird entnommen und zurückgegeben.
Anbieter	Verändere den Ordner so, daß der zuletzt eingegebene Text entnommen wird.	Ist sicher, daß es noch einen Text im Ordner gibt

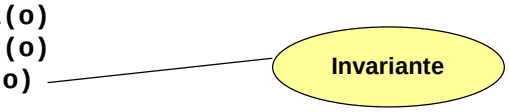
Invariante

- **Invariante beschreibt Bedingungen,**
  - die erfüllt sind, wenn Objekte eines ADTs ***erzeugt*** werden
  - die während der ***gesamten*** Lebenszeit der Objekte gelten
  - d.h. von allen Operationen ***nicht verletzt*** werden
- **Beispiel: ADT Ordner**
  - zu jedem Zeitpunkt im "Leben" eines Ordnerobjekts gilt:
  - **`anzahlTexte >= 0 AND anzahlTexte <= MaxTexte`**
  - ADT Ordner wird um eine interne Funktion erweitert, die die Invariante prüft.

```
PROCEDURE Invariante (o : Ordner): BOOLEAN =
BEGIN
  RETURN (o^.anzahlTexte >= 0 AND o^.anzahlTexte <= MaxTexte);
END Invariante;
```


ADT Ordner mit Invariante

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
  requires NOT IstVoll(o)
  ensures  NOT IstLeer(o)
  ensures  Invariante(o)

PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
  requires NOT IstLeer(o)
  ensures  NOT IstVoll(o)
  ensures  Invariante(o)

PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
. . .
END OrdnerADT.
```



Realisierung von Zusicherungen in M3

- **Pragmas in Modula-3**
  - Pragmas sind Anweisungen an den ***Übersetzer***
  - Sie ändern die ***Semantik*** des Programmes nicht
  - Die Implementierung eines Pragmas ist ***übersetzerspezifisch***
  - Pragmas können ***irgendwo*** im Programmtext auftreten
  - Pragmas müssen einer vordefinierte Syntax entsprechen
    - *<* Pragmaname Parameter *>*
- **Das Pragma ASSERT**
  - <* ASSERT AUSDRUCK *>
  - Der AUSDRUCK muß einen Wert vom Typ BOOLEAN liefern
  - Der Ausdruck wird zur ***Laufzeit*** ausgewertet
  - Ist das Ergebnis des Ausdrucks FALSE
    - wird ein ***Laufzeitfehler*** generiert
    - das Programm bricht ab!

Zusicherungen mit Pragma ASSERT

```
PROCEDURE EntnehmeText (VAR o: Ordner): TEXT =
VAR t : TEXT;
BEGIN
  <* ASSERT NOT IstLeer(o) *>

  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;

 <* ASSERT NOT IstVoll(o)  *>
 <* ASSERT Invariante(o)   *>

  RETURN t;
END EntnehmeText;
```

```
VAR ordner1, ordner2 : OrdnerADT.Ordner; t : TEXT;

BEGIN
  ordner1 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
  OrdnerADT.LegeTextAb (ordner1, "Nicht immer...");
  t := OrdnerADT.EntnehmeText(ordner1);
  t := OrdnerADT.EntnehmeText(ordner1);
```

Laufzeitfehler

Diskussion: Einsatz von ASSERT

- **ASSERT**
  - bietet eine einfache Möglichkeit, Zusicherungen zu implementieren.
- **Nachteile**
  - "brutale" Implementierung
  - es wird keine Information über die Verletzung des Vertrages geliefert
  - es gibt keine Möglichkeit, auf die Verletzung des Vertrages zu reagieren
- **Bessere Lösung**
  - Entwerfen eines Moduls zur Prüfung von Zusicherungen

```
PROCEDURE Require (expr : BOOLEAN);
PROCEDURE Ensure  (expr : BOOLEAN);
```

- **Frage**
  - Was soll geschehen, wenn eine Zusicherung verletzt wird?
  - Verletzung: Ausnahmesituation!

Exkurs: Ausnahmebehandlung

- **Beispiel für Ausnahmesituationen**
  - Während des Schreibens einer Datei auf Diskette wird die Diskette entfernt
  - Kein Plattenplatz mehr verfügbar
  - Berechnung eines Addition ist größer als LAST(INTEGER)
  - Falsche Daten werden von einer Datei eingelesen.
- **Definition: Ausnahme**
  - IEEE Glossary: "An event that causes suspension of normal program execution. Types include addressing exception, data exception, operation exception, overflow exception, protection exception, underflow exception."
  - Ausnahmen sind Programmzustände, die nicht im normalen Programmablauf vorgesehen sind.

Merkmale von Ausnahmen

- **Merkmale**
  - Ausnahmen entstehen zur ***Laufzeit***
  - Einige Programmiersprachen (Java, Ada, Modula-3) erlauben, benutzerdefinierte ***Ausnahmen*** zu deklarieren und ***Ausnahmebehandlung*** durchzuführen
  - Beispiel: vorgegebene Ausnahme
    - *SIO.Error*
      Wird von den IO-Modulen generiert, wenn Datei-Operationen nicht intendiert durchgeführt werden können.
- **Deklaration benutzerdefinierter Ausnahmen**
  - EXCEPTION <name>;
  - Wird eine Ausnahme von einer Schnittstelle exportiert, können auch die Klienten diese Ausnahme generieren.
- **Generieren einer Ausnahme**
  - RAISE-Anweisung

Beispiel: Arbeiten mit Ausnahmen

```
MODULE Ausnahme EXPORTS Main;
IMPORT SIO;
VAR eingabe : INTEGER;
BEGIN
  LOOP
    TRY
      SIO.PutLine ("Geben Sie bitte eine ganze Zahl ein:");
      eingabe := SIO.GetInt();
      EXIT;
    EXCEPT
      SIO.Error => SIO.PutLine ("*** Eingabeformat falsch");
                   line := SIO.GetLine();
    END;
  END;
  . . .
END Ausnahme.
```

Schützt Anweisungen, bzgl- einer Ausnahmebehandlung

Ausnahmebehandlung "exception handler"

Weiterleiten von Ausnahmen

- **Idee:**
  - Wenn in einer Prozedur eine Ausnahme nicht behandelt werden soll, dann kann diese Prozedur die Ausnahme an die ***sie rufende Prozedur*** weiterleiten.
  - So können Ausnahmen über ***mehrere Stufen*** weitergeleitet und an der entsprechenden Stelle behandelt werden.
- **Weiterleitung von Ausnahmen**
  - muß bei der Prozedur-Deklaration angegeben werden
  - PROCEDURE <name> signature RAISES {ex1, .. exN}
- **Beispiele:**
  - viele Operationen des Moduls SIO leiten die Ausnahme Error weiter
  - `PROCEDURE GetChar(rd: Reader := NIL): CHAR RAISES {Error};`

Kontrollfluß bei Ausnahmen

- Eine Ausnahme tritt in TRY-EXCEPT-Anweisung auf und die Ausnahme wird dort behandelt, dann
  - werden die in dem EXCEPT-ELSE-Teil für die Ausnahme stehenden Anweisungen durchgeführt,
  - Das Programm wird anschließend nach dem END der TRY-EXCEPT-Anweisung ***fortgeführt***.
- Eine Ausnahme tritt in einem ***ungeschützten*** Bereich einer Prozedur auf und die Ausnahme ist Element der RAISES-Liste der Prozedur, dann,
  - wird die Prozedur abgebrochen und die Ausnahme an die die Prozedur rufende Prozedur ***weitergeleitet***
- Kann eine Ausnahme weder behandelt noch weitergeleitet werden, dann,
  - wird das Programm mit einem Laufzeitfehler ***abgebrochen***

Schnittstelle des Moduls Assertion

```
INTERFACE Assertion;

EXCEPTION Violated;
          Terminate;

PROCEDURE Require (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated};

PROCEDURE Ensure  (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated};

PROCEDURE EnableAssertions();
PROCEDURE DisableAssertions();

END Assertion.
```

Ausnahme `Violated` wird generiert, wenn eine Zusicherung verletzt wird

Prüfung der Zusicherungen kann unterdrückt werden

Implementierung des Moduls Assertion - 1

```
MODULE Assertion;
IMPORT SIO;

VAR enabled : BOOLEAN;

PROCEDURE Require (expr : BOOLEAN; procName: TEXT)
                  RAISES {Violated} =
BEGIN
  IF enabled AND NOT expr THEN
    SIO.PutLine("*** Precondition violated: " & procName);
    RAISE Violated;
  END;
END Require;

PROCEDURE Ensure  (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated} =
BEGIN
  IF enabled AND NOT expr THEN
    SIO.PutLine("*** Postcondition violated: " & procName);
    RAISE Violated;
  END;
END Ensure;
```

Geschützte Variable

Implementierung des Moduls Assertion - 2

```
PROCEDURE EnableAssertions()=
BEGIN
  enabled := TRUE;
END EnableAssertions;

PROCEDURE DisableAssertions()=
BEGIN
  enabled := FALSE;
END DisableAssertions;


BEGIN
  EnableAssertions();
END Assertion.
```


Ordner-Operationen mit Assertions - 1

```
INTERFACE OrdnerADT;

IMPORT Assertion AS As;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb     (VAR o: Ordner; t : TEXT)
                          RAISES {As.Violated};
PROCEDURE EntnehmeText   (VAR o: Ordner): TEXT
                          RAISES {As.Violated};
PROCEDURE IstVoll        (o: Ordner): BOOLEAN;
PROCEDURE IstLeer        (o: Ordner): BOOLEAN;
PROCEDURE Beschrifte     (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung(o: Ordner): TEXT;
PROCEDURE Anlegen        ( ) : Ordner RAISES {As. Violated};

END OrdnerADT.
```

Ordner-Operationen mit Assertions - 2

```
PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT)
                     RAISES {As. Violated} =
BEGIN
  As.Require (NOT IstVoll(o), "OrdnerADT.LegeTextAb");
  o^.anzahlTexte := o^.anzahlTexte + 1;
  o^.ordnerInhalt[o^.anzahlTexte] := t;
  As.Ensure (NOT IstLeer(o), "OrdnerADT.LegeTextAb");
  As.Ensure (Invariante(o), "OrdnerADT.LegeTextAb");
END LegeTextAb;

PROCEDURE EntnehmeText (VAR o: Ordner) : TEXT
                       RAISES {As. Violated} =
VAR t : TEXT;
BEGIN
  As.Require (NOT IstLeer(o), "OrdnerADT.EntnehmeText");
  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;
  As.Ensure (NOT IstVoll(o), "OrdnerADT.EntnehmeText");
  As.Ensure (Invariante(o), "OrdnerADT.EntnehmeText");
  RETURN t;
END EntnehmeText;
```


Umgang mit den Ausnahmen - 1

- **Empfehlung**
  - der aufrufende Block klammert alle Anweisungen in einer TRY-EXCEPT-Anweisung
  - Im EXCEPT-Teil werden noch mögliche Abschluß-Operationen durchgeführt (z.B. Schließen von Dateien) und eine entsprechende Meldung ausgegeben und die Ausnahme `Termine` generiert.

```
PROCEDURE ArbeiteMitOrdner() RAISES {As.Terminate} =
VAR ordner1 : OrdnerADT.Ordner;
BEGIN
  TRY
    ordner1 := OrdnerADT.Anlegen();
    OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
    SIO.PutLine (OrdnerADT.EntnehmeText(ordner1));
    ...
  EXCEPT
    As.Violated =>
      SIO.PutLine ("*** Called from: ArbeiteMitOrdner");
      RAISE As.Terminate;
  END;
END ArbeiteMitOrdner.
```

Umgang mit den Ausnahmen - 2

```
MODULE Ordner_Test EXPORTS Main;

IMPORT Assertion AS As;
IMPORT OrdnerADT,  SIO;

PROCEDURE ArbeiteMitOrdner()RAISES {As.Terminate}=
BEGIN
  ...
END ArbeiteMitOrdner;

BEGIN
  As.EnableAssertions();
  TRY
    ArbeiteMitOrdner();
  EXCEPT
    As.Terminate =>
      SIO.PutLine ("*** Program terminated ");
  END
END Ordner_Test.
```

```
*** Precondition violated: OrdnerADT.EntnehmeText
*** Called from: ArbeiteMitOrdner
*** Program terminated
```


Arbeiten mit Zusicherungen - 1

- **Im Rumpf einer Operation darf die Vorbedingung nicht geprüft werden**
  - widerspricht dem herkömmlichen ***defensiven*** Programmieren
- **Vorteil**
  - schon mittelgroße Systeme enthalten ca. 10 -20% Code, um solche Eingangsprüfungen durchzuführen
  - dabei entstehen komplexe Prüfungen
  - Prüfroutinen sind häufig Ursachen für Fehler

```
PROCEDURE Wurzel (x: REAL): REAL is
  require X >= 0
BEGIN
END Wurzel ;

PROCEDURE Wurzel (x: REAL): REAL is
  require X >= 0
BEGIN
  if x < 0 then
    “handle error”
  else
    RETURN (x * x);
  end
END Wurzel ;
```

Arbeiten mit Zusicherungen - 1

- **Zusicherungen sollen nicht verwendet werden, um Spezialfälle zu behandeln**
  - Zusicherungen sind Aussagen über die ***korrekte*** Nutzung.
  - Sollen Spezialfälle behandelt werden, dann werden herkömmliche ***Kontrollanweisungen*** verwendet (IF oder CASE).
- **Beispiel**
  - Wenn `Wurzel` den Fall x < 0 als Spezialfall behandeln soll, dann ist das zu programmieren.
  - Wenn x >= 0 die Vorbedingung ist, dann ist ein Aufruf

    ```
    Wurzel (-1);
    ```

    nicht erlaubt, also eine ***nicht vertragskonforme Benutzung***!
- **Wird eine Zusicherung verletzt (zur Laufzeit)**
  - Vorbedingung: Nutzer hat den Vertrag verletzt
  - Nachbedingung: Anbieter hat den Vertrag verletzt

Was haben wir gelernt!

- **Vertragsmodell**
  - Regelt die korrekte Benutzung von Operationen.
  - Mit Hilfe von Zusicherungen kann ein Vertrag formuliert werden

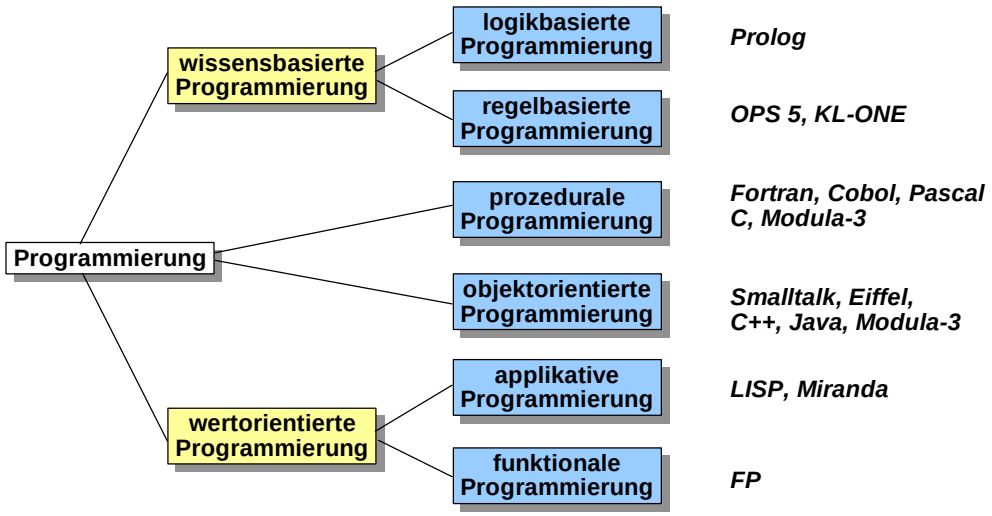
- **Zusicherungen**
  - Zusicherungen definieren Anforderungen an den Nutzer und an den Hersteller einer Programmkomponente.
  - Werden Zusicherungen verletzt, kann entschieden werden, wer den Vertrag gebrochen hat.
  - Zusicherungen können realisiert werden mit
    - Pragmas
    - mit Hilfe des Konzepts der Behandlung von Ausnahmen
- **Vertragsmodell sollte immer im Zusammenhang mit Objektmodulen und ADT-Modulen implementiert werden!**

Objektorientierte Programmierung I

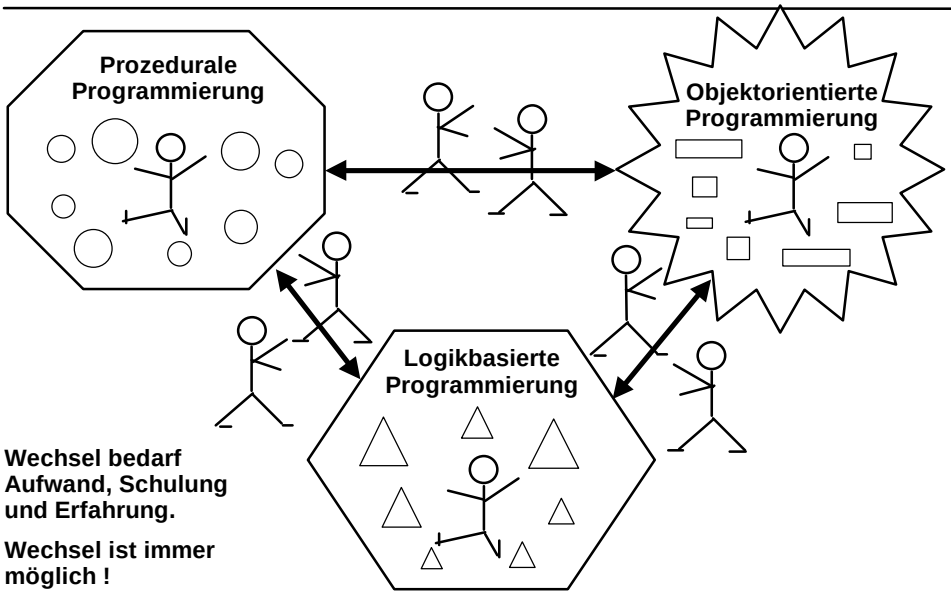
- Verständnis der objektorientierten Programmierung
- Objekte
- Klassen
  - konkrete und abstrakte Klassen
- Vererbung

Programmier-Paradigmen

- Es gibt unterschiedliche Arten der Programmierung:



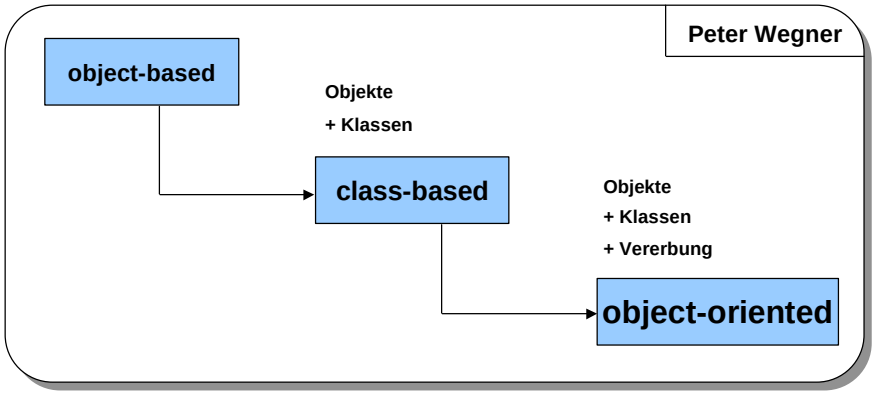
Wechsel des Programmier-Paradigmas



Wechsel bedarf Aufwand, Schulung und Erfahrung.

Wechsel ist immer möglich !

Merkmale objektorientierter Systeme/Sprachen



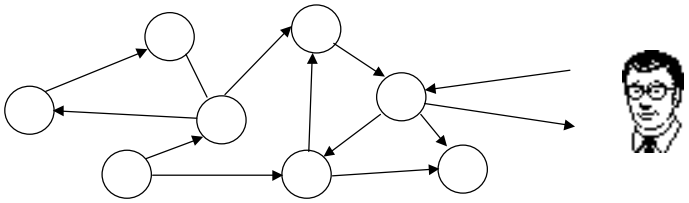
Ada Modula-2	CLU (Ada) (Modula-2)	Simula Smalltalk-80 Eiffel C++ Java Modula-3

Klassifikation

- **Objektbasiert**
  - Objektbasierte Programmiersprachen bieten die Möglichkeit, **Objekte** im Sinne einer **Datenkapsel** resp. eines **Objektmoduls** zu realsieren.
  - Jedes Objekt wir **einzeln** beschrieben und benutzt
- **Klassenbasiert**
  - Klassenbasierte Programmiersprachen bieten die Möglichkeit, Objekte in Forn von **Objekttypen (Klassen)** zu beschreiben.
  - Von Objekttypen können beliebig viele **Exemplare** (Objekte des Typs) erzeugt werden.
- **Objektorientiert**
  - Objektorientierte Programmiersprachen erlauben, Objekttypen (Klassen) mithilfe der **Vererbungs-Beziehung** zu strukturieren.
  - Dadurch lassen sich **Objekttyp-Hierarchien** im Sinne der Spezialisierung resp. Generalisierung modellieren.

Was ist ein objektorientiertes System

- **Ein Software-System besteht aus Objekten**
- **Jede Systemaktivität ist die Aktivität eines Objektes**
- **Objekte**
  - haben eine Lebensdauer (werden erzeugt und vernichtet)
  - sind identifizierbar
  - sind aktiv
  - verändern sich während ihrer Lebensdauer
  - senden und empfangen Nachrichten



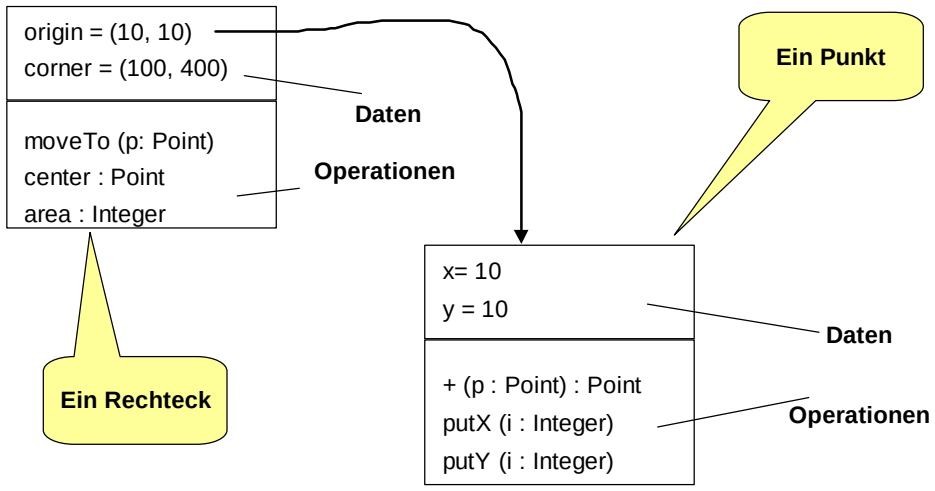
Objekte - die Dynamik des Systems

- **Ein Objekt ist eine Datenkapsel, die aus zwei Teilen besteht**
  - Der Wert der Daten repräsentiert den **Zustand** des Objekts
  - Daten können nur mithilfe von **Operationen** verändert werden (Kapselung)

Daten
Operationen

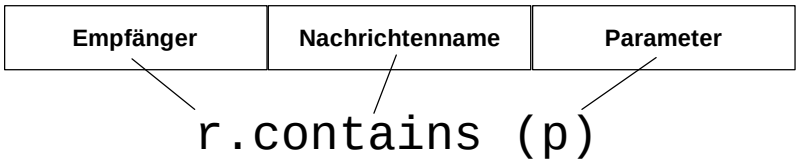
- **Die Aktivität der Objekte ist die Ausführung ihrer Operationen**
  - Eine Operation wird ausgeführt, wenn ein Objekt eine entsprechende **Nachricht** erhält.
  - Der Objektzustand kann sich **verändern**.
  - Mögliche Operationen sind durch den **Objekttyp** bestimmt.
- **Ein Objekt ist ein Exemplar genau einer Klasse.**
- **Objekte existieren nur zur Laufzeit**
  - können aber auch persistent gespeichert werden

Beispiele für Objekte

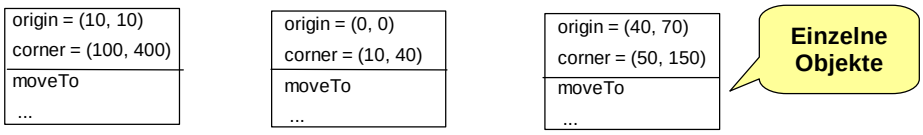


Nachrichten (Botschaften)

- **Objekte kommunizieren miteinander,**
  - dadurch daß sie **Nachrichten** versenden und Nachrichten empfangen.
- **Objekte reagieren auf Nachrichten,**
  - indem sie eine **Operation** (oder **Methode**) ausführen.
- **Der Empfänger einer Nachricht (immer ein Objekt) ist verantwortlich für**
  - **Entschlüsselung** der Nachricht (verstehe ich die Nachricht?)
  - **Wirkung** (was tue ich?)



Klassen - die Statik des Systems



- **Gleichartige Objekte werden zusammengefaßt**
  - und an einer Stelle beschrieben (Objekttyp oder Klasse)
- **Eine Klasse definiert für ihre Objekte**
  - die **Speicherstruktur**
    - Daten, Attribute, Exemplarvariablen
  - die **Operationen** (Methoden, Routinen),
    - von denen ein Teil exportiert wird (exported, public <-> private)
    - andere werden nur intern benötigt
- **Von einer Klasse können beliebig viele Objekte erzeugt werden**

Beispiel : Klasse Point

```
class Point

feature
  x, y : Integer;

feature
  +: (p : Point) is
      result : Point;
    do
      result.x(x + p.x);
      result.y(x + p.y);
    end;
  x: (i : Integer) is
    do
      x := i;
    end;
  y: (i : Integer) is
    do
      y := i;
    end;
  ...
end -- class Rectangle
```

Exemplarvariablen

exportierte Operationen

Beispiel : Klasse Rechteck

```
class Rectangle

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean is
    do ...
    end;

feature {NONE}
  extent : Point
    do
      -- liefert einen Punkt, der die Höhe und
      -- Weite des Rechtecks repräsentiert
    end;
end -- class Rectangle
```

Exemplarvariablen

exportierte Operationen

private Operationen

Klassen - Objekt

- **Eine Klasse entspricht einem ADT**
- **Ein Objekt entspricht einer Datenkaspel (Objektmodul)**
- **Von einer Klasse sind außerhalb sichtbar:**
  - der Klassenname
  - die exportierten Operationen
- **Jedes Objekt ist ein Exemplar**
  - einer Klasse des Programms
- **Objekte einer Klasse kennen dieselben Operationen**
- **Objekte einer Klasse unterscheiden sich in den Werten ihrer Daten**

Objekte und Klassen

- Nach außen sichtbar ist

```
class Rectangle
interface
  Create;
  origin : Point;
  corner : Point;
  moveTo (p : Point);
  contains (p : Point)
   Boolean:
end -- class Rectangle
```

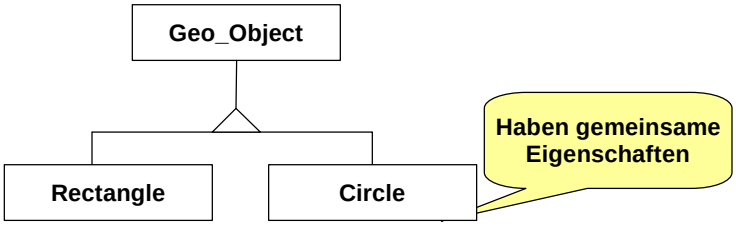
Zusammenhang Objekt <-> Klasse

```
      r : Rectangle;
      p : Point

      p.Create;
      r.Create;

      r.contains (p);
```

```
class Rectangle

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean is
    do
    end;
feature {NONE}

  extent : Point
    do
      -- liefert ...
    end;
end -- class Rectangle
```

Vererbung

- **Gemeinsame Eigenschaften verschiedener Klassen**
  - werden in einer **eigenen Klasse** zusammengefaßt und definiert,
  - und anschließend an diese vererbt.
- **Beispiel**



- **Regel:**
  - Eine Klasse A erbt von einer Klasse B genau dann, wenn A eine **Spezialisierung** (Unterbegriff) von B ist.

Beispiel: Vererbung - 1

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;
  ...
end -- class Rectangle
```

```
class Circle

inherit
  Geo_Object
    define moveTo, contains

feature
  center : Point;
  radius : Integer:
  ...
end -- class Circle
```

Abstrakte Klasse
Spezifikationsklasse

Einfachvererbung

Beispiel: Vererbung - 2

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;
contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;
  ...
end -- class Rectangle
```

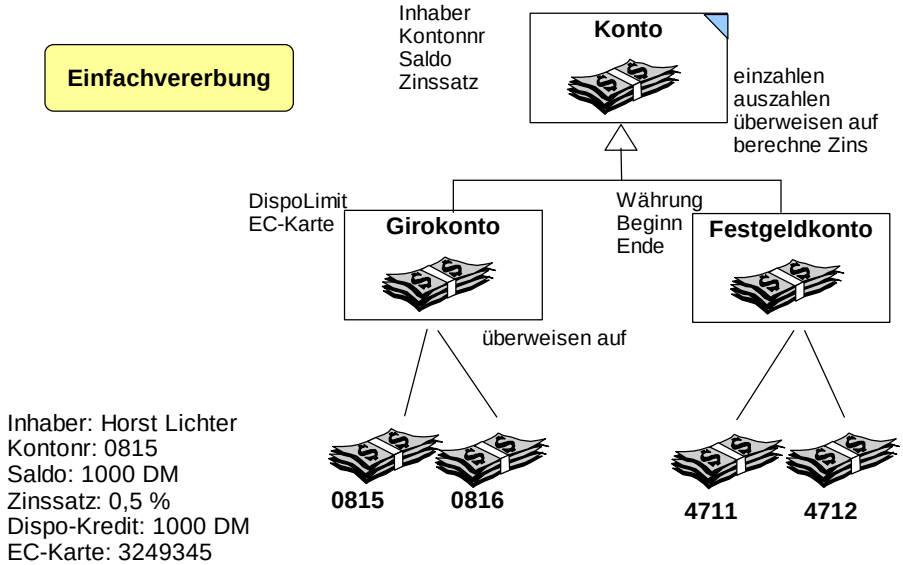
```
class DisplayableObject

feature
  psDescription : String is
    deferred
    end;
end -- class DisplayableObject
```

```
class DisplayableRectangle

inherit
  Rectangle

inherit
  DispayableObject
    define psDescription

feature
  ...
end -- class DisplayableRectangle
```

Mehrfachvererbung

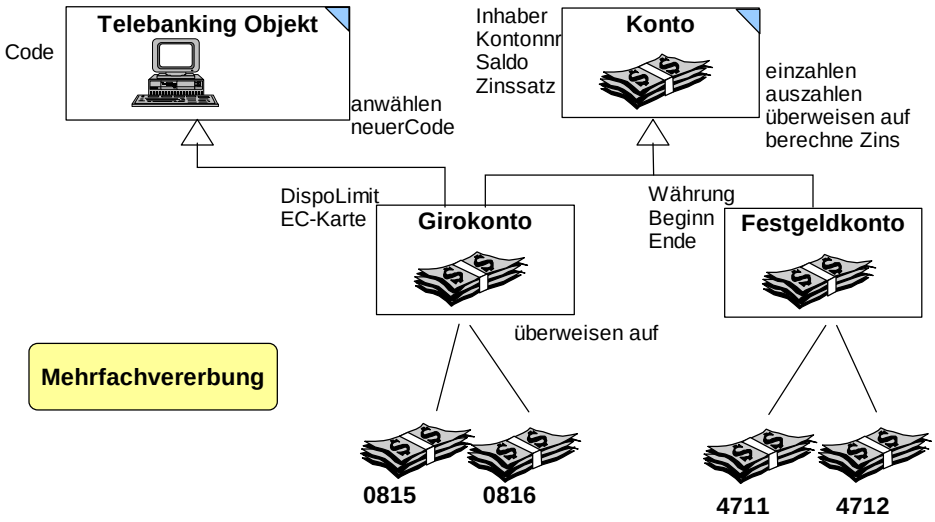
Beispiel: Vererbung - 3

Einfachvererbung



Inhaber: Horst Lichter
Kontonr: 0815
Saldo: 1000 DM
Zinssatz: 0,5 %
Dispo-Kredit: 1000 DM
EC-Karte: 3249345

Beispiel: Vererbung - 4

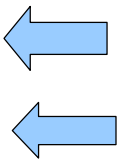


Abstrakte Klassen

- **Eine Klasse ist ein**
  - möglicherweise **nicht vollständig implementierter** ADT.
- **In konkreten Klassen (implementiert, effektiv)**
  - sind alle Operationen **ausführbar.**
- **In abstrakten Klassen (deferred, virtual)**
  - sind einige Operationen ***noch nicht implementiert*** , sondern nur spezifiziert (deklariert).
- **Von konkreten Klassen**
  - können **beliebig viele** Objekte (Exemplare) erzeugt werden.
  - new- bzw. create- Nachrichten (Instanziierung)
- **Von abstrakten Klassen**
  - können **keine** Objekte erzeugt werden!

Beschreibungsschema von Klassen

- **Eine Klasse ist definiert durch**
- **ihren Namen**
- **ihre direkten Oberklassen**
  - die erbt_von-Beziehung muß zyklenfrei sein
- **eine Speicherstrukturbeschreibung**
  - erweitert die geerbten Beschreibungen
- **eine Menge von Operationsbeschreibungen**
  - erweitert die geerbten Beschreibungen



Oberklasse <-> Unterklasse

Geerbte Eigenschaften können auf drei Arten in einer Unterklasse modifiziert werden

Erweitern	etwas Neues hinzufügen
Redefinieren	sich ähnlich verhalten
Definieren	etwas Versprochenes realisieren

Erweitern

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo,
   contains

feature
  origin, corner :
   Point;

feature

  height : Integer is
    do ...
    end;
  width : Integer is
    do ...
    end;
  ...
end -- class Rectangle
```

Spezifische Eigenschaften werden hinzugefügt

Definieren

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean
   is
    do
      ...
    end;
  ...
end -- class Rectangle
```

Definieren versprochener, aber noch nicht implementierter Eigenschaften

Redefinieren

```
class DisplayableObject
  feature
  psDescription: String is
    deferred
    end;
end -- class DisplayableObject
```

```
class Rectangle
...
feature
  initialize is do
      origin.Create; corner.Create;
      origin.initialize;
      corner.initialize;
    end
  ...
end -- class Rectangle
```

```
class DisplayableRectangle
inherit
  Rectangle
    rename initialize as initRect
    redefine initialize
...
feature
  color : Color;

feature
  initialize is do
    current.initRect;
    color.create; color.black;
  end;
...
end -- class DisplayableRectangle
```

Die ererbte Implementierung paßt im Kontext der Unterklasse nicht mehr.

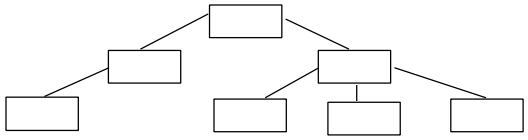
Sie wird redefiniert!

Varianten der Vererbung

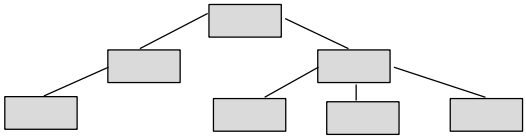
Modifikationsmöglichkeiten \ Anzahl der Oberklassen	eine oder keine	beliebig viele
nur Erweitern und Definieren	***strikte Einfachvererbung***	***strikte Mehrfachvererbung***
Erweitern Redefinieren Definieren	***nicht-strikte Einfachvererbung***	***nicht-strikte Mehrfachvererbung***

Merkmale objektorientierter Architekturen - 1

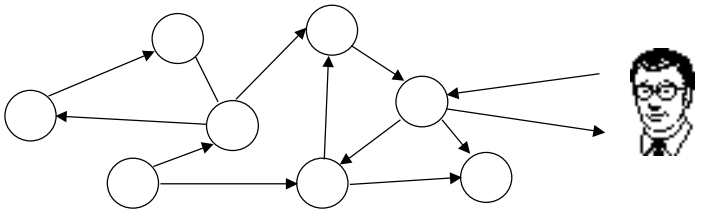
Entwurf



Programm



Ablauf



Merkmale objektorientierter Architekturen - 2

Entwurf

Klassen
- zur Spezifikation
- zur Implementierung

Beziehungen zwischen Klassen
- benutzt (use)
- erbt_von (inherit)

Programm (Statik)

Klassen
- (partiell) implementierte abstrakte Datentypen

Ablauf (Dynamik)

Objekte
- Exemplare der Klassen des Programms

Beziehungen zwischen Objekten
- Versenden und empfangen Nachrichten

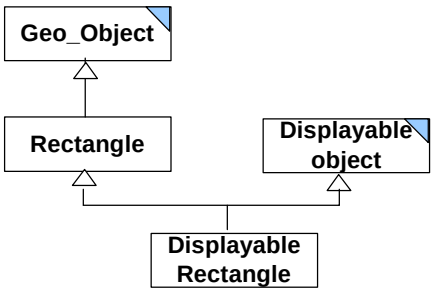
Polymorphismus

- **Allgemein:**
  - "Polymorphismus ist die Fähigkeit von Etwas, von verschiedener
  - Gestalt zu sein"
- **In Programmiersprachen:**
  - Etwas: Variablen, Funktionen, Prozeduren
  - Gestalt: Typ
- **Eine polymorphe "Entität" kann in verschiedenen Umgebungen verwendet werden,**
  - die unterschiedliche Typen verlangen.
- **Beispiele**

```
write (100, 3)
write ('a')
```

Beispiel: Polymorphismus - 1

- **Die Vererbung ist ein Mechanismus, um Polymorphismus in Programmiersprachen zu realisieren.**



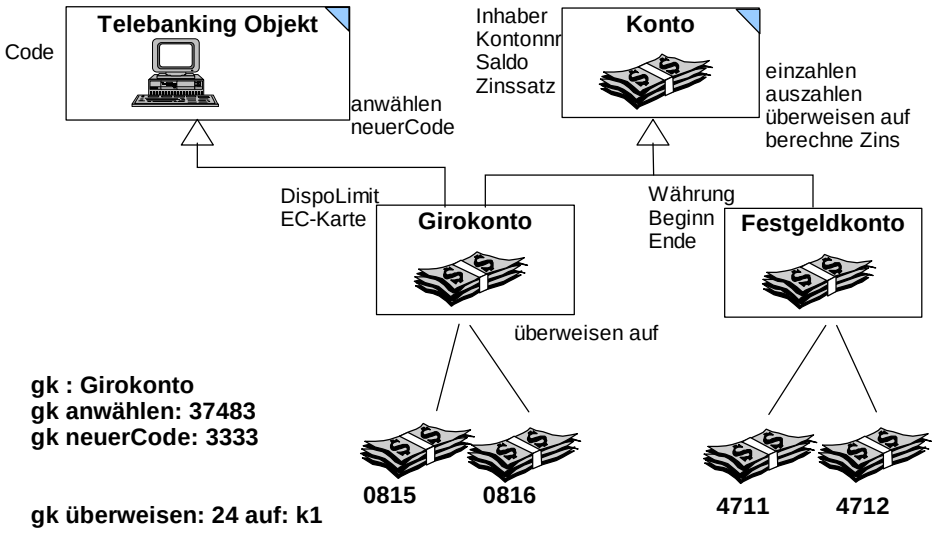
ein DisplayableRectangle **verhält sich wie** ein Rechteck und wie ein DisplayableObject

```
dr : DisplayableRectangle;
dr.Create;

dr.contains (p);
x := dr.center;
dr.moveTo (p);

s := dr.psDescription
```

Rectangle

DisplayableObject

Beispiel: Polymorphismus - 1



Das dynamische Binden

```
g : Geo_Object;
r : Rectangle;
c : Circle;

r.Create; c.Create

g := r;
g.contains (p);

g := c;
g.contains (p);
```

```
class Geo_Object

feature
  contains (p : Point) : Boolean is
    deferred
    end;
end -- class Geo_Object
```

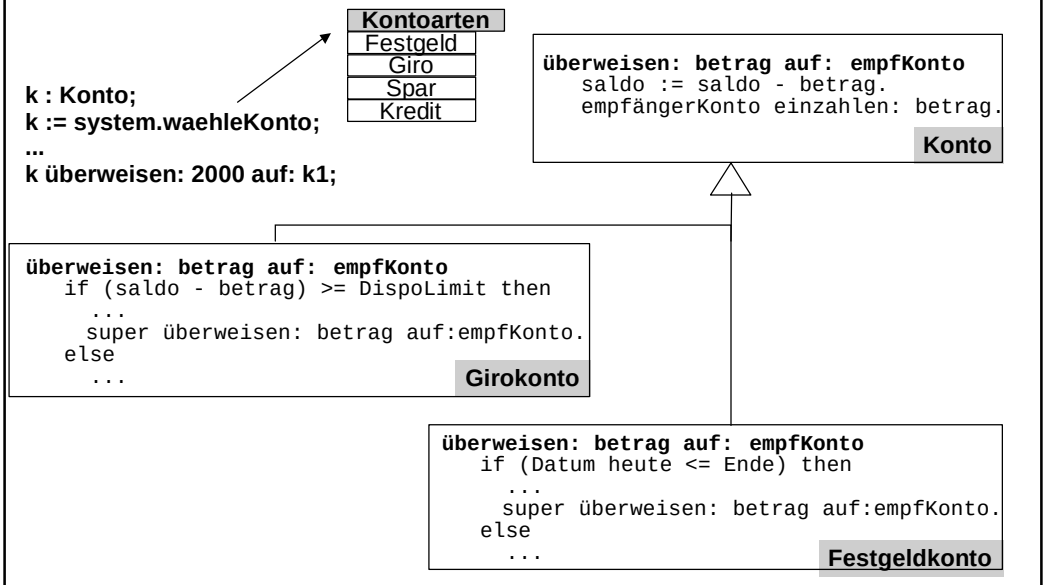
```
class Rectangle

feature
   contains (p : Point) :Boolean is
     ...
end -- class Rectangle
```

```
class Circle

feature
  contains (p : Point) :Boolean is  ...
end -- class Circle
```

Dynamisches Binden heißt:

die **richtige** Implementierung zur **Laufzeit** finden

Beispiel: Dynamisches Binden



Was haben wir gelernt!

- **Klassifikation**
  - objektbasiert, klassenbasiert
  - objektorientiert
- **Klassen sind (partiell) implementierte ADTen**
  - abstrakte Klassen
  - konkrete Klassen
- **Vererbung**
  - dient dazu, Spezialisierungsbeziehung zwischen Begriffen programmiertechnisch zu realisieren.
  - Unterschied Einfach- Mehrfachvererbung
- **Polymorphismus**
  - kann mit Hilfe der Vererbung realisiert werden
  - führt zum dynamischen Binden von Implementierungen

Objektorientierte Programmierung in Modula-3

- Realisierung von Klassen in Modula-3
- Objekttypen
- Konzept der Untertypen
- Realisierung von Vererbung mittels Untertypen
- Kritik

Wiederholung ADT

- **ADTs in Modula-3**
  - ein ADT ist immer ein **Referenztyp**
  - in der Schnittstelle wird ein **opaker Typ als Untertyp** des vordefinierten Referenztyps REFANY deklariert
- **Ein ADT entspricht dem Konzept einer Klasse.**
- **Eine Exemplar eines ADTs entspricht einem Objekt**
- **Mit Hilfe der ADTs**
  - kann eine **klassenbasierte** Programmierung in Modula-3 umgesetzt werden.
- **Zur Objektorientierung fehlen**
  - Sprachkonstrukte, um die **Vererbung** zu modellieren.
- **Deshalb**
  - stellt Modula-3 das Konzept der **Objekttypen** bereit.

Objekttypen in Modula-3

- **Mit Hilfe der Objekttypen können in Modula-3 Klassen beschrieben werden.**
- **Ein Objekttyp gibt an**
  - Bezeichner der Klasse (des Objekttyps)
  - Bezeichner der Oberklasse
    - Es kann maximal eine Oberklasse geben
  - Bezeichner der Exemplarvariablen
  - Signaturen der Methoden
  - Bezeichner der Methoden, die in der Klasse redefiniert werden
    - in Modula-3 spricht man von überschreiben (overrides)

Einfachvererbung

- **Ein Objekttyp (Klasse) wird**
  - in einer Modulschnittstelle deklariert
  - in Implementierung enthält den strukturellen Aufbau des Objekttyps (Klasse) und die Realisierung der Methoden.

Beispiel für einen Objekttyp - 1

```
TYPE Point = ROOT OBJECT
                x : INTEGER; y : INTEGER;
             METHODS
                getX(): INTEGER          := PointGetX;
                setX (value : INTEGER)   := PointSetX;
                getY (): INTEGER         := PointGetY;
                setY (value : INTEGER)   := PointSetY;
                add (p: Point) : Point   := PointAdd;
                . . .
             END;
```

- **Die Klasse**
  - hat die vordefinierte Klasse **ROOT** als Oberklasse
  - deklariert zwei Exemplarvariablen x und y
- **In der METHODS-Klausel**
  - kann die **Bindung** zwischen Methode und der Prozedur, die sie realisiert, hergestellt werden!
- **Objekttypen sind spezielle Referenztypen**
  - Objekte werden mit der NEW-Operation erzeugt.

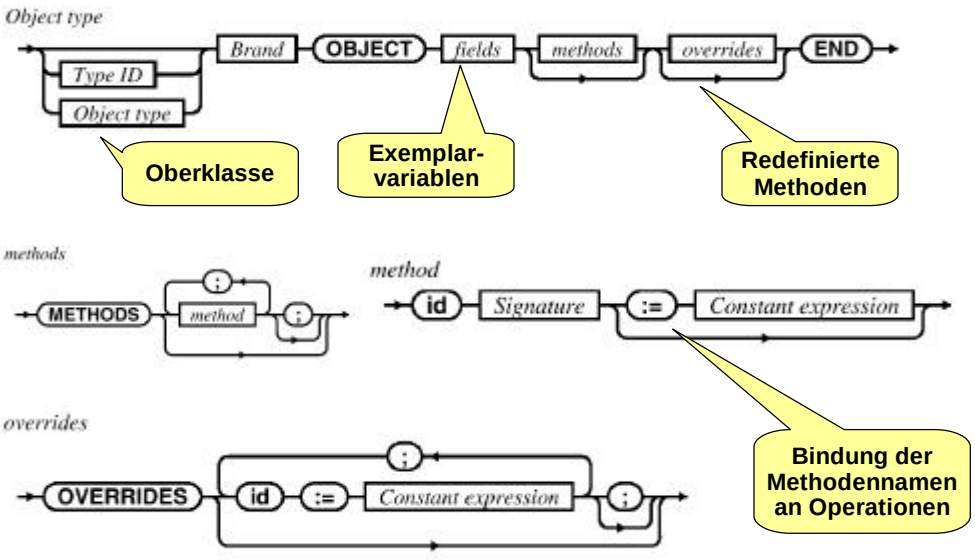
Beispiel für einen Objekttyp - 2

```
MODULE Point_Test EXPORTS Main;
TYPE Point = OBJECT
               x : INTEGER;
               y : INTEGER;
             METHODS
               ...
             END;
PROCEDURE PointGetX(self : Point): INTEGER =
BEGIN
  RETURN self.x;
END PointGetX;

PROCEDURE PointSetX(self: Point; value : INTEGER) =
BEGIN
  self.x := value;
END PointSetX;

PROCEDURE PointAdd (self: Point; p: Point) : Point =
VAR newP : Point;
BEGIN
  newP := NEW(Point);
  newP.setX(self.x + p.getX());
  newP.setY(self.y + p.getY());
  RETURN newP;
END PointAdd;
...
```

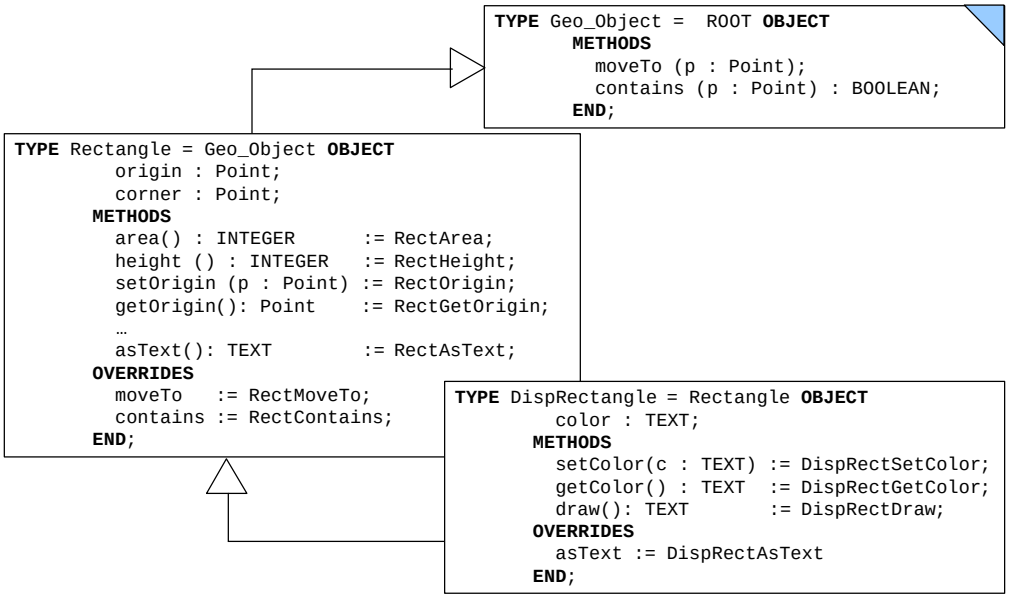
Die Prozeduren, die Methoden realisieren, erwarten als ersten Parameter ein Objekt der Klasse.

```
VAR p1, p2, p3 : Point;
BEGIN
  p1 := NEW(Point);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
END Point_Test.
```


Syntax der Objekttyp-Deklaration - 1



Beispiel: Vererbung

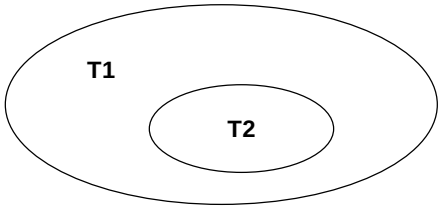


Geschützte Objekttypen

- **Um geschützte Objekttypen zu deklarieren**
  - verwenden wir die M3-Sprachkonstrukte, die wir zur Realisierung ADTen benutzt haben!
    - **Trennung** von Schnittstelle und Implementierung
      - Pro Objekttp (Klasse) ein Modul
    - **Opake** Typen
    - Konzept der **Untertypen**
    - Konvention der **T-Notation** für Objekttypen
- **Der Name der Klasse wird als opaker Typ deklariert**
  - Deklaration in der Schnittstelle.
  - Ein opaker Typ muß immer ein Untertyp eines Referenz- oder Objekttyps sein.
  - Der Obertyp (ein Objektyp) deklariert die **öffentliche** Schnittstelle
    - D.h. nur die Signaturen der Methoden werden angegeben

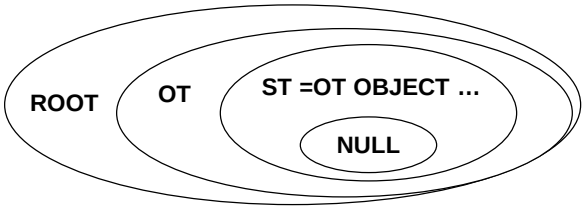
Untertypen in Modula-3

- **Subtypen in Modula-3**
  - Modula-3 kennt das Konzept der Konstruktion von ***Untertypen***
  - Subtypbeziehung wird durch "***<:***" angezeigt
- **Definition**
  - Seien T1 und T2 Typen und die Relation T2 <: T1 besteht, dann sind ***alle Werte*** von T2 auch ***Werte*** von T1
  - T1 nennt man ***Obertyp***; T2 nennt man ***Untertyp***
  - Es gilt: ein Typ kann beliebig viele Untertypen haben, ein Typ kann ***maximal*** einen Obertyp haben.



Untertypen von Referenz- und Objekttypen

- **Modula-3 definiert zwei vorgegebene Referenztypen**
  - ***REFANY*** und ***NULL*** mit folgender Subtypbeziehung
  - `NULL <: REF T <: REFANY`
- **Modula-3 definiert einen vorgegebenen Objekttyp**
  - ROOT mit folgender Subtypbeziehung
  - `NULL <: ST = OT OBJECT … END <: OT <: ROOT <: REFANY`
- **Interpretation:**
  - jeder Objekttyp ist ***Untertyp*** von ROOT (und damit ein Referenztyp)



Beispiel: Geschützter Objekttyp

```
INTERFACE Point;

TYPE Point <: PointPublic;

     PointPublic = ROOT OBJECT
                METHODS
                  getX(): INTEGER;
                  setX (value : INTEGER);
                  getY (): INTEGER;
                  setY (value : INTEGER);
                  add (p: Point) : Point;
                  minus (p : Point) : Point;
                  asText(): TEXT;
                END;
END Point.
```

Point ist Untertyp des "öffentlichen Teils von Point"

Dieser Typ wird exportiert und kann benutzt werden!!!!!!

In der Implementierung müssen die Exemplar-variablen und die Operationen, die die Methoden realisieren angegeben werden!

Point <: PointPublic <: ROOT

Beispiel: Geschützter Objekttyp

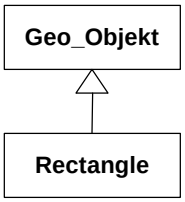
```
MODULE Point;

REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX;
        getX := PointGetX;
        setY := PointSetY;
        getY := PointGetY;
        add := PointAdd;
        minus := PointMinus;
        asText := PointAsText;
      END;

PROCEDURE PointGetX(self : Point): INTEGER =
BEGIN
  RETURN self.x;
END PointGetX;
```

- **Point wird als Objekttyp deklariert**
  - PublicPoint ist der **Obertyp**
  - Point <: PublicPoint <: ROOT
- **Point erweitert den Obertyp**
  - um die fehlenden **Instanzvariablen.**
- **Point bindet**
  - an die Methoden entsprechende **Implementierungen.**

Diskussion dieser Realisierung

- **Untertyp-Konzept wird verwendet, um**
  - **Spezialisierungsbeziehung** zwischen Klassen auszudrücken

```
TYPE Rectangle = Geo_Object OBJECT
        origin : Point;
        corner : Point;
      METHODS
        area() : INTEGER      := RectArea;
        …
        asText(): TEXT        := RectAsText;
      OVERRIDES
        moveTo   := RectMoveTo;
        contains := RectContains;
      END;
```



  - geschützte Objekttypen (Klassen) zu **implementieren**

```
REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX; ...
      END;
```

???

T-Konvention

T bezeichnet immer den im Modul realisierten Objekttyp (Klasse)

```
INTERFACE Point;

TYPE T <: PointPublic;

     PointPublic = ROOT OBJECT
                METHODS
                  setX(): INTEGER;
                  ...
                  add (p: T) : T;
                  minus (p : T) : T;
                  asText(): TEXT;
                END;
END Point.
```

```
MODULE Point_Test  EXPORTS Main;

IMPORT Point,  SIO;

VAR p1, p2, p3 : Point.T;
BEGIN
  p1 := NEW(Point.T);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point.T);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
  SIO.PutLine(p3.asText());
END Point_Test.
```

Implementierung der Klasse Point

```
PROCEDURE AsText (self : T) : TEXT =
BEGIN
  RETURN (Fmt.Int(self.x)  &  "&"  &  Fmt.Int(self.y));
END AsText;

PROCEDURE GetX(self : T): INTEGER =
BEGIN
  RETURN self.x;
END GetX;

PROCEDURE GetY(self : T): INTEGER =
BEGIN
  RETURN self.y;
END GetY;

...

PROCEDURE Add (self: T; p: T) : T =
VAR newP : T;
BEGIN
  newP := NEW(T);
  newP.setX(self.getX() + p.getX());
  newP.setY(self.getY() + p.getY());
  RETURN newP;
END Add;
```


Beispiel: Klassenhierachie Geo_Objects

```
INTERFACE Geo_Object;
IMPORT Point;

TYPE T =  ROOT OBJECT
  METHODS
    moveTo (p : Point.T);
    contains (p : Point.T) : BOOLEAN;
  END;
END Geo_Object.
```

Abstrakte Klasse

```
INTERFACE Rectangle;
IMPORT Geo_Object, Point;

TYPE T <: Public;
     Public = Geo_Object.T OBJECT
              METHODS
                area() : INTEGER;
                height () : INTEGER;
                setOrigin (p : Point.T);
                getOrigin(): Point.T;
                setCorner(p : Point.T);
                getCorner(): Point.T;
                asText () : TEXT;
              END;
END Rectangle.
```

```
INTERFACE DisplayableRectangle;
IMPORT Rectangle;

TYPE T <: Public;
     Public = Rectangle.T OBJECT
              METHODS
                setColor(c : TEXT);
                getColor() : TEXT;
                draw(): TEXT;
              END;
END DisplayableRectangle.
```

Implementierung von Rectangle

```
MODULE Rectangle EXPORTS Rectangle;
IMPORT Point;

REVEAL
  T = Public BRANDED OBJECT
        origin : Point.T;
        corner : Point.T;
      OVERRIDES
        setOrigin := SetOrigin;
        getOrigin := GetOrigin;
        setCorner := SetCorner;
        getCorner := GetCorner;
        area := Area;
        height := Height;

        moveTo   := MoveTo;
        contains := Contains;
        asText   := AsText;
      END;

PROCEDURE AsText(self : T) : TEXT =
BEGIN
  RETURN ("Rect origin: " & self.origin.asText() &
          " corner: " & self.corner.asText());
END AsText;
```

Notwendig, um geschützte Implementierung zu erhalten

Echte Redefinitionen

```
PROCEDURE MoveTo (self : T; p : Point.T)=
VAR newPoint := NEW(Point.T);
BEGIN
  newPoint := self.getCorner();
  self.setCorner(newPoint.add(p));

  newPoint := self.getOrigin();
  self.setOrigin(newPoint.add(p));
END MoveTo;


PROCEDURE Contains (self : T; p : Point.T):
                    BOOLEAN =
BEGIN
  ...
END Contains;
```


Implementierung von DispRectangle

```
MODULE DisplayableRectangle EXPORTS DisplayableRectangle;

IMPORT Rectangle;
TYPE SuperClass = Rectangle.T;

REVEAL
  T = Public BRANDED OBJECT
        color : TEXT;
      OVERRIDES
        setColor := SetColor;
        getColor := GetColor;
        draw     := Draw;

        asText := AsText;
END;
```

Echte Redefinition

- **Direkter Aufruf einer Methode der Oberklasse**
  - Sollte im Normalfall **nicht** gemacht werden.
  - Sinnvoll jedoch, wenn **rekursiv redefiniert** wird, d.h. das die Leistung der redefinierten Methode bei der neuen Implementierung verwendet werden soll.

```
PROCEDURE AsText(self : T) : TEXT =
VAR t : TEXT;
BEGIN
  t := SuperClass.asText(self);
  RETURN (t & " color: " & self.color );
END AsText;
```

Verwenden der gerade redefinierten der Oberklasse

Umgang mit Objekten

```
IMPORT Point,  SIO, Rectangle, DisplayableRectangle;
VAR
   p1, p2, p3 := NEW(PointO.T);
   r1         := NEW(DisplayableRectangle.T);

BEGIN
  p1.setX(10);
  p1.setY(10);
  SIO.PutLine(p1.asText());

  p2.setX(100);
  p2.setY(100);
  SIO.PutLine(p2.asText());

  r1.setOrigin(p1);
  r1.setCorner(p2);
  r1.setColor("black");
  SIO.PutLine(r1.asText());

  p3.setX(50);
  p3.setY(50);
  SIO.PutLine(p3.asText());

  r1.moveTo(p3);
  SIO.PutLine(r1.asText());
```

Deklarieren der Variablen und erzeugen er Objekte

Senden von Nachrichten an die erzeugen Objekte

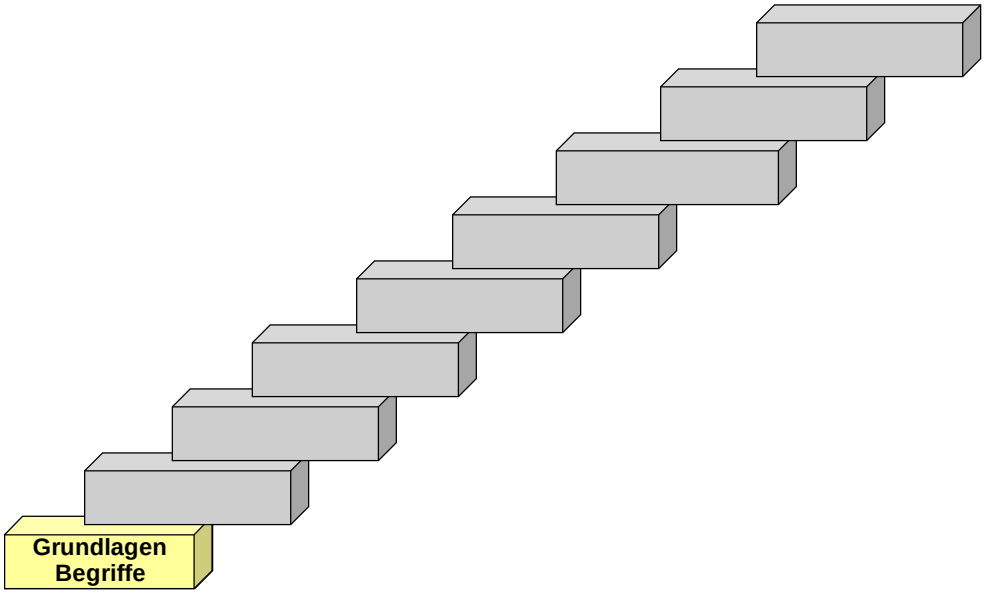
```
10&10
100&100
Rect origin: 10&10 corner: 100&100 color: black
50&50
Rect origin: 60&60 corner: 150&150 color: black
```


Diskussion: Objektorientierung in M3

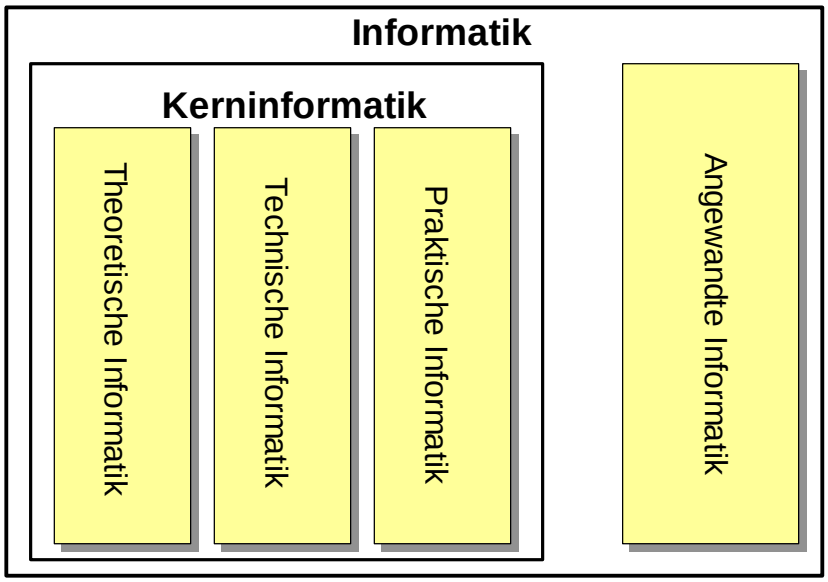
- **Feststellung:**
  - Es ist **möglich**!
- **Kritik**
  - Modula-3 ist eine **hybride** Sprache
  - Sie erlaubt **imperative** und **objektorientierte** Programmierung
  - Beides kann bunt durcheinander **gemischt** werden
    - trägt nicht zur guten Verständlichkeit und Lesbarkeit bei!
- **Die Implementierung der objektorientierten Konzepte**
  - basiert auf dem Untertyp-Konzept der Sprache
  - ist m. E. nicht **besonders gut gelungen**
- **Konsequenz**
  - Soll durchgängig objektorientiert entwickelt werden, dann sollte man eine **rein objektorientierte** Sprache verwenden, z.B. Java, Eiffel, Smalltalk!

Zusammenfassung

Aufbau der Vorlesung



Gebiete der Informatik -1

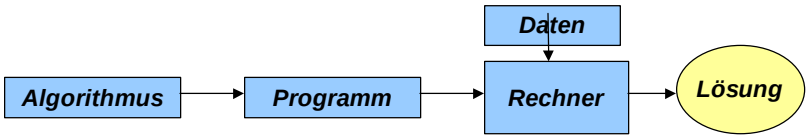


Algorithmus

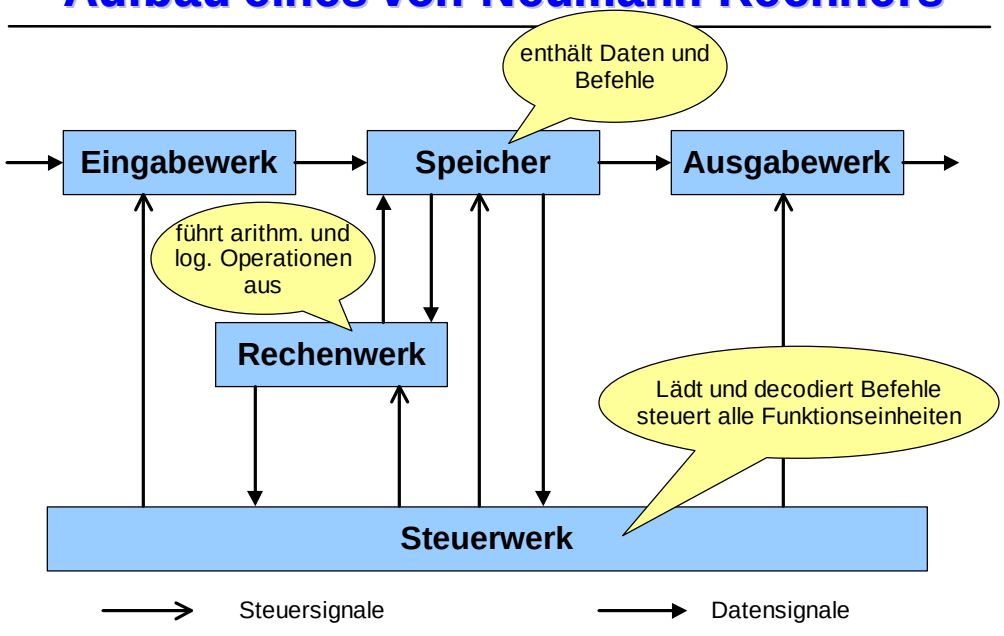
- **Ein Algorithmus ist ein Verfahren, welches**
  - in einem ***endlichen*** Text niedergelegt werden muß
  - ***effektiv*** ausführbar ist,
  - durch eine mechanisch oder elektronisch arbeitende *Maschine* ***ausgeführt*** werden kann
  - Anzahl und Ausführungszeit der Elementaroperationen sind beschränkt
  - Ein Algorithmus wird entsprechend seiner Vorschrift schrittweise ausgeführt
- **Eigenschaften**
  - Abstraktion
  - Finitheit
  - Terminierung
  - Determinismus
  - Determiniertheit

Definition: Software, Programm

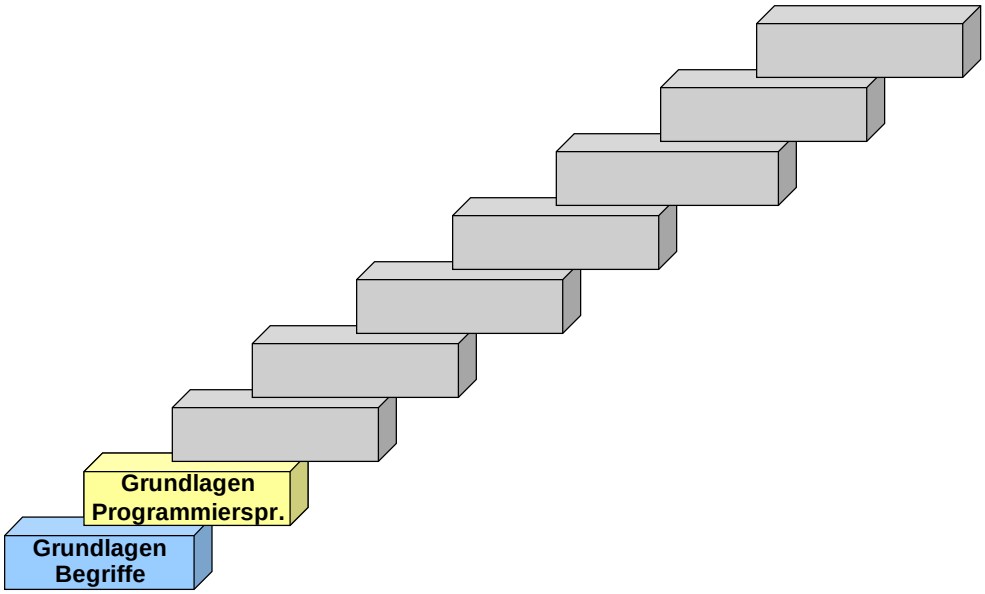
- **Definition: Software**
  - IEEE Standrad Glossary of Software Engineering Terminoloy
    - "Computer ***programs***, ***procedures***, and possibly associated ***documentation*** and ***data*** pertaining to the operation of a computer system."
- **Definition: Programm**
  - IEEE Standrad Glossary of Software Engineering Terminoloy
    - "A combination of ***computer instructions*** and ***data definitions*** that enable computer hardware to ***perform*** computational or control functions".
- **Unter dem Begriff Programmieren versteht man**
  - das **Lösen von Problemen** unter Zuhilfenahme eines **Rechners**



Aufbau eines von-Neumann-Rechners



Aufbau der Vorlesung



Alphabet, Wort, Formale Sprache

- **Alphabet**
  - Ein Alphabet ist eine ***nichtleere endliche*** Menge von ***unterscheidbaren*** Zeichen ("Buchstaben")
    A = {a_1, a_2, a_3, …} mit einer ***Ordnungsrelation*** ≤ ($a_1 \leq a_2 \leq a_3$ ….)
- **Wort über einem Alphabet**
  - ***endliche Folge*** von Buchstaben, die auch ***leer*** sein kann (ε leere Wort)
  - A* bezeichnet die ***Menge aller Wörter*** über dem Alphabet A (inkl. dem leeren Wort).
- **Formale Sprache**
  - Sei A ein Alphabet. Eine (formale) Sprache (über A) ist ***jede beliebige Teilmenge von A****.

Grammatik - Definition - 1

- **Definition:**
  - Eine Grammatik G für eine Sprache L ist definiert durch
  - ein ***Viertupel (N, T, P, S)***
  - N: Menge der ***Nichtterminalsymbole***
  - T: Menge der ***Terminalsymbole***
  - P: Menge von ***Produktionsregeln***
  - S: das ***Startsymbol***
- **Wort einer Sprache**
  - Jedes durch Anwendung der Produktionsregeln aus S erzeugbare Wort, ***das nur aus Terminalsymbolen besteht***, gehört zu der von der Grammatik erzeugten Sprache L(G)

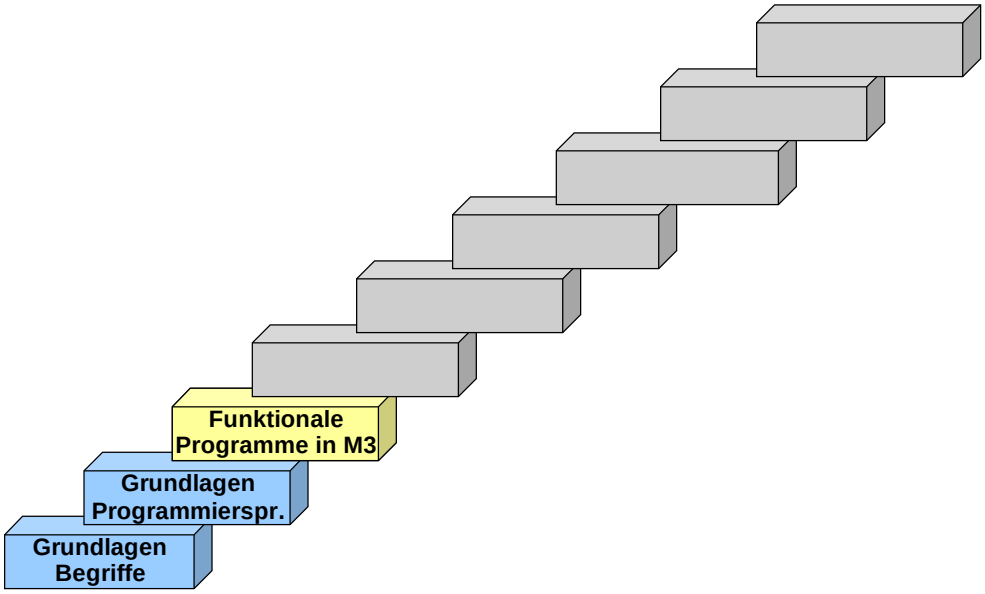
Chomsky-Grammatiken

- **Typ-0-Grammatik**
  - Gestalt der Produktionen ist nicht eingeschränkt
  - Alle Sprachen, die überhaupt mit endlichen Regelsystemen erzeugt werden können
- **Typ-1 oder kontextsensitive Grammatik**
  - Produktionen sind beschränkt oder kontextsensitiv
- **Typ-2 oder *kontextfreie* Grammatik**
  - Produktionen sind kontextfrei (d.h. die linke Seite einer Produktion ist immer ein Nichtterminal)
- **Typ-3 oder reguläre Grammatik**
  - Produktionen sind terminierend, links- , rechtslinear

EBNF u. Syntaxdiagramme

- **EBNF**
  - **E**xtended **B**ackus-**N**aur-**F**orm
  - ***Meta-Sprache*** zur Beschreibung der Syntax formaler Sprachen
  - ***Metasymbole*** von EBNF sind
    - = „definiert als“
    - (...|...) genau eine Alternative aus der Klammer muß stehen
    - [...] Inhalt der Klammer kann stehen oder nicht
    - { ... } Inhalt der Klammer kann n-fach stehen, n >= 0
    - . Ende der Produktion
    - Terminalsymbole werden in " " eingeschlossen
- **Syntaxdiagramme**
  - beschreiben Produktionen ***grafisch***
  - Nichterminalsymbole sind Rechtecke
  - Terminalsymbole sind Langrunde

Aufbau der Vorlesung



Funktionale Programmierung

- **Konzept**
  - Formulierung von ***Funktionsdefinitionen***
  - Ausführung eines funktionalen Programms besteht in der ***Berechnung*** eines ***Ausdrucks*** mit Hilfe dieser Funktionen
  - Berechnung liefert ein ***Ergebnis*** zurück
- **Was benötigt man dazu**
  - Daten / ***Datentypen*** und ***elementare*** Funktionen
  - Möglichkeit, Funktionen zu ***definieren***
  - Ausdrucksmittel zur ***Vernetzung*** von Funktionen
- **Eine Funktion**
  - hat einen ***Namen***
  - hat keinen oder mehrere ***Eingabeparameter*** (Argumentbereich)
  - hat einen ***Ergebnistyp*** (Ergebnisbereich)
  - hat eine ***Berechnungsvorschrift*** , ist frei von ***Seiteneffekten***

Formale & aktuelle Parameter

- **Parameter**
  - erlauben es, Funktionen mit ***Eingabewerten*** zu versorgen
  - haben eine ***Typ***
  - dadurch werden Funktionen ***flexible einsetzbar***
- **Formale Parameter**
  - werden in der ***Definition*** einer Funktion angegeben
  - dienen als ***Stellvertreter*** im ***Rumpf*** der Funktion für die zur Laufzeit des Programms übergebenen aktuellen Parameter
- **Aktuelle Parameter**
  - beim ***Aufruf*** einer Funktion müssen ihre formalen Parameter gemäß ihrer Definition an aktuelle Parameter ***gebunden*** werden
  - diese werden dann im Rumpf ***verwendet***.

Deklaration, Anweisung, Ausdruck

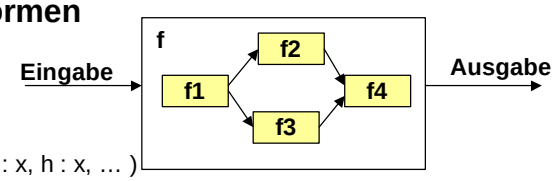
- **Deklarationen:**
  - Idee: Namen (***Bezeichner***) werden vereinbart, damit diese später benutzt werden können
  - Die in einem Block deklarierten Bezeichner sind nur innerhalb des Blockes ***gültig***.
- **Anweisungen:**
  - Sind in Blöcken enthalten.
  - Die Folge der Anweisungen eines Blocks wird bei Ausführung in der ***Reihenfolge der Aufschreibung*** abgearbeitet.
- **Ausdrücke**
  - werden ***ausgewertet***
  - liefern einen Wert (Ergebnis)
  - Viele Anweisungen erlauben, daß Ausdrücke verwendet werden können

Datentypen

- **Typbegriff**
  - im Zusammenhang mit Programmiersprachen hat der Begriff ***Typ*** oder auch ***Datentyp*** eine zentrale Bedeutung
- **Man unterscheidet grob:**
  - ***einfache*** Datentypen
  - ***zusammengesetzte*** Datentypen
- **Einfachen Datentypen in Modula-3**
  - **Ganze Zahlen**
  - **Zeichen**
  - **Texte**
  - **Wahrheitswerte**
  - **Gleitkommazahlen**

Vernetzung von Funktionen

- **Um komplexe Ausdrücke zu berechnen,**
  - werden Funktionen vernetzt.
- **Funktionalformen (oder Funktionale)**
  - beschreiben "Vernetzungsmuster"
- **Beispiele für Funktionalformen**
  - **Komposition**
    - f o g : x = g : (f : x)
  - **Konstruktion**
    - [f, g, h, …] : x = (f : x, g : x, h : x, …)
  - **Bedingung**
    - if t then f else g : x =



f : x , falls t : x = true
g : x , falls t : x = false
? , sonst

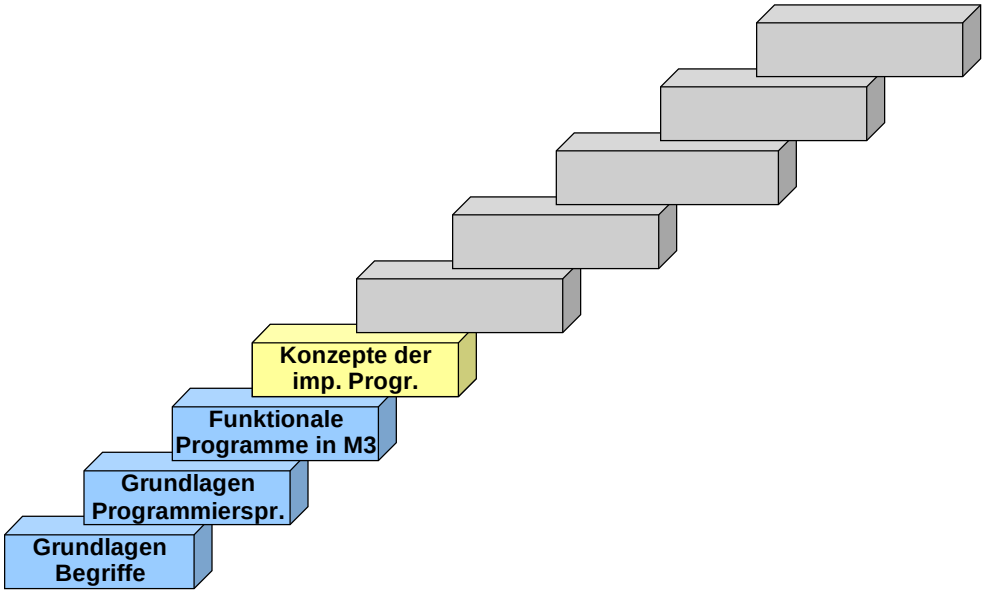
Rekursion

- **Idee:**
  - allgemein bezeichnet man mit ***Rekursion*** die Definition eines Problems, einer Funktion oder ganz allgemein eines Verfahrens "***durch sich selbst***"
- **Rekursive Funktion**
  - darunter verstehen wir Funktionen, die sich ***selbst wieder aufrufen***
- **Indirekte Rekursion:**
  - ***Indirekte*** Rekursion kann in einem System von Funktionen definiert werden, die Seite an Seite vereinbart werden und sich ***gegenseitig stützen***.
- **Anmerkung:**
  - Es darf keine ***unkontrollierte*** (unendliche) Rekursion entstehen
  - Jeder rekursive Funktionsaufruf gehört in eine ***bedingte Anweisung***
  - Rekursionsabbruch

Funktional, Rekursion und Iteration

- **Funktionale Algorithmen**
  - kennen keine ***Variablen*** zur Zwischenspeicherung von Ergebnissen
- **Nichtfunktionale Algorithmen**
  - speichern Zwischenergebnisse in ***Variablen*** ab
- **Rekursive Algorithmen**
  - lösen ein Problem, in dem sie sich ***selbst wieder aufrufen*** (direkt oder indirekt)
- **Iterative Algorithmen**
  - besitzen Abschnitte, die bei der Ausführung ***mehrmals*** durchlaufen werden

Aufbau der Vorlesung



Variable und Wertzuweisung

- **Variable**
  - **logischer** Speicherplatz mit seinem Wert
  - besitzt einen **Namen**, unter dem man die Variable ansprechen kann
  - Datentyp legt fest, welche **Werte** eine Variable annehmen kann und welche **Operationen** darauf ausgeführt werden können
  - Variable kann als Behälter betrachtet werden
    - hat einen Wert und einen Typ
- **Wertzuweisung**
  - dient dazu, den Wert einer Variablen zu verändern
  - Der Ausdruck wird zuerst ausgewertet, das Ergebnis wird anschließend der Variable zugewiesen
    - In diesem Zusammenhang sprechen wir oft von der rechten und der linken Seite einer Zuweisung:
    - (right-hand side - RHS, left-hand side - LHS).
  - LHS und RHS müssen Typkompatibel sein

Abstraktion durch Prozeduren

- **Fachlich ist eine Prozedur**
  - die programmiersprachliche ***Realisierung*** eines Algorithmus.
- **Softwaretechnisch**
  - kann eine Prozedur zunächst als ***benannte Anweisungsfolge*** verstanden werden.
- **Die Grundidee ist,**
  - den Namen der Prozedur "***stellvertretend***" für diese Anweisungsfolge zu verwenden.
- **Die Prozedur ist eine wesentliche Umsetzung des Konzepts der *algorithmische Abstraktion* (auch *Prozeßabstraktion* genannt):**
  - Statt einer expliziten Anweisungsfolge (der genauen Verarbeitungsvorschrift) wird ein davon ***abstrahierender*** Name verwendet.

Parameterübergabearten

- **Allgemein gibt es folgende Parameterarten für Prozeduren**
  - ***Eingabeparameter***
  - ***Ausgabeparameter***
  - ***Ein- / Ausgabeparameter***
- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Wertparameter***
  - realisieren Eingangsparameter
  - Veränderungen des formalen Parameters in der Prozedur haben nur ***lokale Auswirkung***. Der aktuelle Parameter bleibt unverändert.
- **Der formale Parameter beim *Call by Reference* ist ein**
  - ***Variablenparameter*** (oder Referenzparamter)
  - realisieren Ausgangs und Ein- / Ausgangsparameter
  - Jede Änderung des formalen Parameters ist ***direkt*** im aktuellen Parameter wirksam.

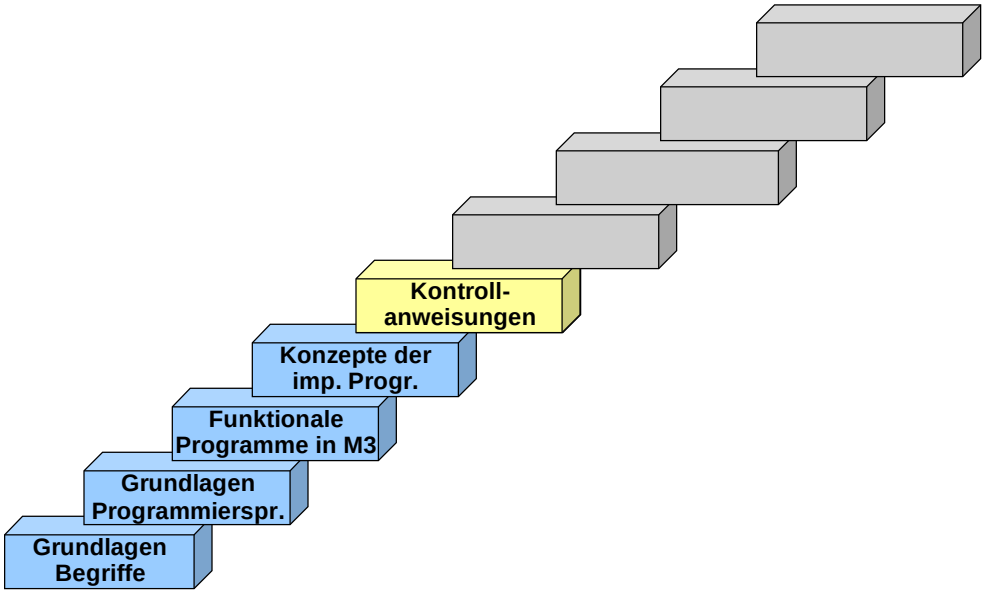
Gültigkeitsbereich und Lebensdauer

- **Gültigkeitsbereich (scope) eines Bezeichners**
  - der ***statische Teil*** des Programms, in dem der Bezeichner mit exakt ***gleicher Bedeutung*** verwendet werden darf
  - wird auch ***Sichtbarkeitsbereich*** oder ***Namensraum*** genannt
  - Bezeichner ist in seinem Sichtbarkeitbereich gültig
  - der Gültigkeitsbereich wird durch den Compiler überwacht
- **Lebensdauer eines Objekts (Variable, Prozedur)**
  - bezieht sich auf den zur ***Programmlaufzeit*** belegten Speicherplatz
  - macht nur Sinn für Objekte, die Speicher belegen
    - Konstanten
    - Typen
    - belegen keinen Speicher

Terminologie: Gültigkeit & Lebensdauer

- **"Objekt" bezeichnet alles,**
  - was durch einen Bezeichner eingeführt wird (Modul, Prozedur, Konstante, Typ, Variable, Parameter),
  - keine Objekte sind demnach Operatoren oder Wortsymbole (z.B. VAR, END)
- **Ein Objekt heißt lokal in Block B,**
  - wenn es im Block B deklariert ist.
- **Ein Objekt heißt global,**
  - wenn es auf Modulebene deklariert ist.
- **Ein Objekt heißt global relativ zu B,**
  - wenn es in B gültig ist, aber nicht lokal in B ist

Aufbau der Vorlesung



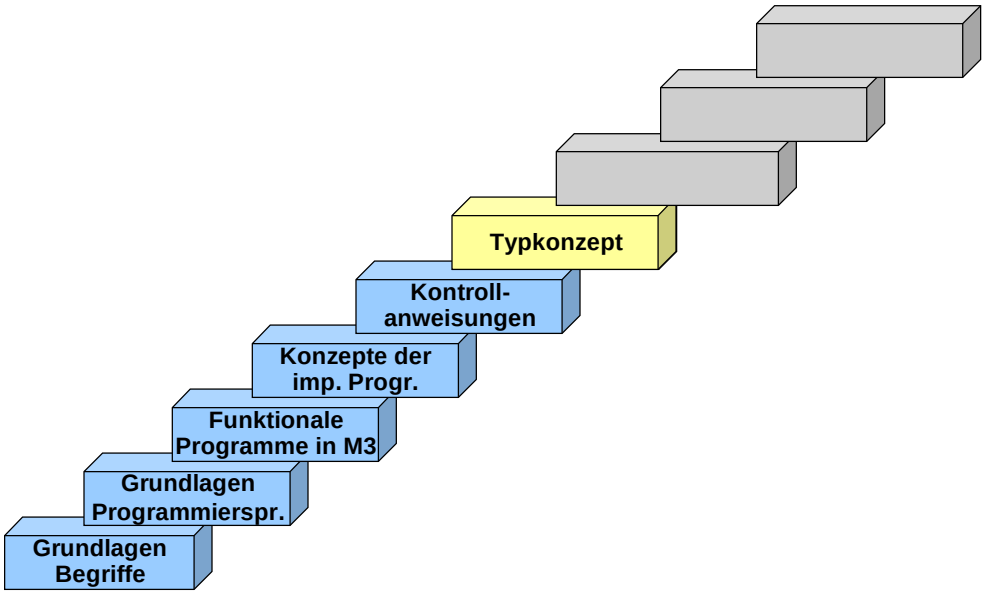
Begriffe

- **Abstraktion:**
  - Die Ausführungsreihenfolge der Aktionen eines Algorithmus entspricht zunächst der textuellen Anordnung (***Sequenz***). Davon kann aber abgewichen werden. Dazu gibt es eigene Aktionen.
  - Ablaufsteuerung:
    - Fallunterscheidung (Auswahl)
    - Wiederholungsaktionen (Iteration)
- **Sprachmechanismen, die dies leisten,**
  - heißen ***Kontrollanweisungen***.
- **Kontrollanweisungen**
  - zusammen mit den Anweisungsfolgen, die von ihnen kontrolliert werden, heißen ***Kontrollstrukturen***.

Arten von Kontrollanweisungen

- **Sequenz von Aktionen:**
  - Eine Aktion wird ***nach der anderen*** abgearbeitet.
- **Auswahlanweisungen kommen vor als:**
  - Einwegauswahl (one-way selection) IF-THEN
  - Zweiwegauswahl (two-way-selection) IF-THEN-ELSE
  - Mehrfachselektion (multiple selection) CASE
- **Wiederholungsanweisungen werden benötigt,**
  - um ***iterative Algorithmen*** zu formulieren.
  - Schleife mit vorheriger Prüfung WHILE und FOR
  - Schleige mit nach nachfolgender Prüfung REPEAT-UNTIL
  - Schleife ohne Prüfung LOOP (EXIT)

Aufbau der Vorlesung



Einteilung von Typen

- **Einfache und zusammengesetzte Datentypen:**
  - ***Einfache Datentypen*** erlauben ***keinen*** Zugriff auf ihre innere Struktur. Ihre Werte können unmittelbar notiert werden.
  - **Zusammengesetzte Datentypen** sind aus anderen Datentypen aufgebaut. Auf ihre einzelnen Elemente kann zugegriffen werden.
- **Vorgegebene und benutzerdefinierte Datentypen:**
  - **Vorgegebene Datentypen** haben einen vordeklarierten Namen und können unmittelbar zur Deklaration von Variablen verwendet werden.
  - **Benutzerdefinierte** Datentypen haben einen selbst definierten Namen und müssen deklariert werden.
- **Statische und dynamische Datentypen**
  - **Statisch**: Größe der Typobjekte ist von vornherein ***bekannt***
  - **Dynamisch**: Größe ist während der Laufzeit ***veränderbar***

Benutzerdefinierte einfache Typen

- **Modula-3 erlaubt,**
  - benutzerdefinierte einfache Typen unter Verwendung einen vorgegebenen elementaren Typs zu deklarieren.
  - `TYPE Zeit = REAL;`
    `Alter = INTEGER;`

Basistyp muß ein Ordinaltyp sein

- **Unterbereichstyp (subrange type)**
  - benutzerdefinierte einfache Typen können als ***Einschränkung*** des Wertebereichs eines elementaren Typs deklariert werden
  - `TYPE Index = [1..10];`
    `Alter = [1 .. 120];`

Ordinaltyp

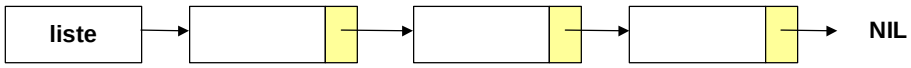
- **Aufzählungstyp (emumeration type)**
  - benutzerdefinierte einfache Typen können durch ***Aufzählung*** der zulässigen Werte deklariert werden
  - `TYPE Ampelfarbe = {rot, gelb, gruen};`
    `Parteien = {SPD, CDU, FDP, PDS, Gruene}`

Zusammengesetzte Typen

- **Array-Typ:**
  - Aneinanderreihung von ***gleichartigen Elementen***, wobei auf die Komponenten mit Hilfe eines ***Index*** zugegriffen wird.
  - Die Anzahl der Elemente ist ***fest*** und heißt ***Länge*** des Array.
  - Zur Indizierung kann jeder ***Ordinaltyp*** verwendet werden.
- **Record-Typ**
  - Records (Verbunde) sind ***heterogene*** kartesische Produkte und dienen zur Darstellung ***inhomogener***, aber ***zusammengehöriger*** Informationen.
- **Mengen-Typ**
  - Mengen sind ***ungeordnete*** Sammlungen von Elementen
  - der Elementtyp muß ein ***Ordinaltyp*** sein!
  - Elemente einer Menge können **nicht** indiziert werden

Eigenschaften dynamischer Datentypen

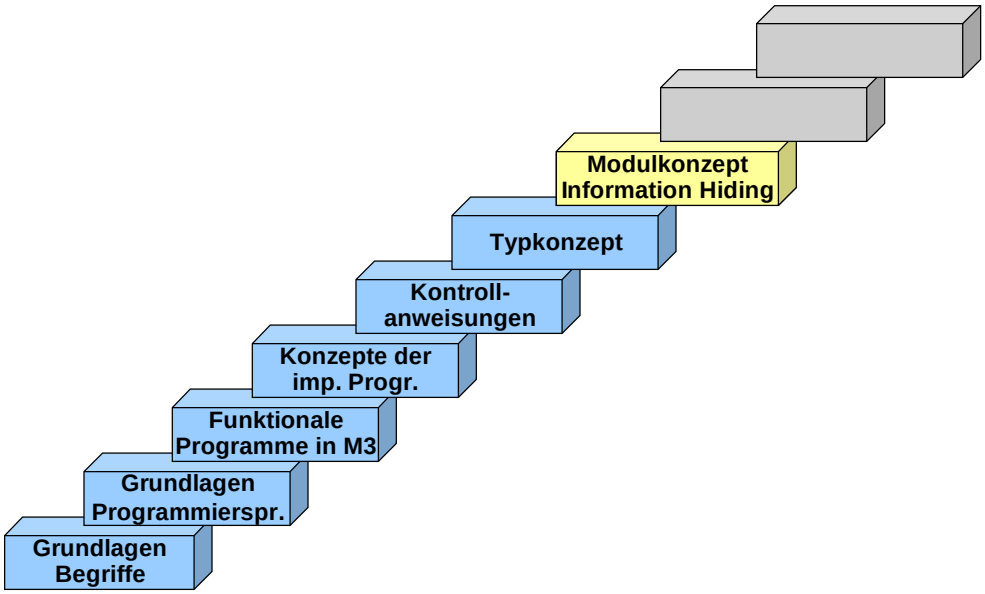
- **Eigenschaften von Objekten dynamischer Datentypen**
  - ***Lebensdauer*** ist nicht an Prozedur oder Moduls gebunden
  - werden ***explizit erzeugt*** und eventuell auch wieder beseitigt
  - sie werden in einem speziell dafür vorgesehenen Speicherbereich angelegt (***Halde oder Heap***)
  - können in prinzipiell ***beliebiger*** Menge geschaffen werden
  - haben im Gegensatz zu den bisherigen Objekten ***keinen festen Bezeichner***
  - sie werden stattdessen über einen ***Zeiger (Pointer)*** identifiziert
- **Dynamische Strukturen: Einfach verkettete lineare Liste**
  - Elemente der Liste sind ***Zeigerobjekte***
  - Jedes Zeigerobjekt ist so konstruiert, das es ***selbst*** wieder auf ein Zeigerobjekt ***verweisen*** kann



Prozedurtyp

- **Frage:**
  - Gibt es eine Möglichkeit, Prozeduren so zu ***parametrisieren***, daß als Parameter ein Algorithmus (Prozedur) übergeben werden kann?
- **Prozedurtyp**
  - Ein Prozedurtyp definiert eine ***Signatur.***
  - Die Werte eines Prozedurtyps sind ***Prozeduren***, die der vorgegebenen Signatur entsprechen.
  - Entsprechend können Variablen als ***Prozedurvariablen*** deklariert werden.
  - Auf Prozedurvariablen können ***passende*** Prozeduren zugewiesen werden.
  - Prozedurvariablen können als ***Parameter*** übergeben werden.
  - Gesetzte Prozedurvariablen können in Anweisungen mit ***aktuellen Parametern*** gerufen werden.

Aufbau der Vorlesung



Modulkonzept

- **Entstanden aus der Notwendigkeit,**
  - große Programmtexte in für den ***Übersetzer faßliche Einheiten*** zu zerlegen,
  - Modulkonzept ist zum zentralen ***Organisationskonzept*** für Entwürfe und Programmtexte geworden.
- **Nach Goos:**
  - Unter einem Modul verstehen wir eine ***Sammlung von Objekten und Algorithmen*** mit der Eigenschaft, daß ihre Kommunikation mit der Außenwelt nur über eine klar ***definierte Schnittstelle*** erfolgt. Das ***Zusammensetzen*** mehrerer Module zu einer Gesamtlösung darf keine Kenntnis ihres ***inneren Aufbaus*** voraussetzen, und die Korrektheit eines Moduls muß ohne Kenntnis seiner Einbettung in die Gesamtlösung nachprüfbar sein.

Information Hiding

- **Prinzip:**
  - Es werden nur die Informationen zur Verfügung gestellt, die ***absolut notwendig*** sind!
  - Alle anderen, insbesondere die ***wichtigen Informationen*** werden ***versteckt***!
  - Der Zugriff auf diese Informationen geschieht über "***Vermittler***".
- **Information Hiding wird realisiert durch**
  - Datenkaspelung in Objektmodulen
  - Abstrakte Datentypen

Objektmodul - Datenkapsel

- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Datenstruktur von ihren sichtbaren Eigenschaften.
- **Merkmale:**
  - Eine Datenstruktur wird in einem Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen sichtbar***, die den allgemeinen Umgang mit der Datenstruktur beschreiben.
  - Die Datenstruktur selbst ist ***verborgen***.
- **Jedes Objektmodul**
  - beschreibt und realisiert nur eine ***einzige sog. abstrakte Datenstruktur***.

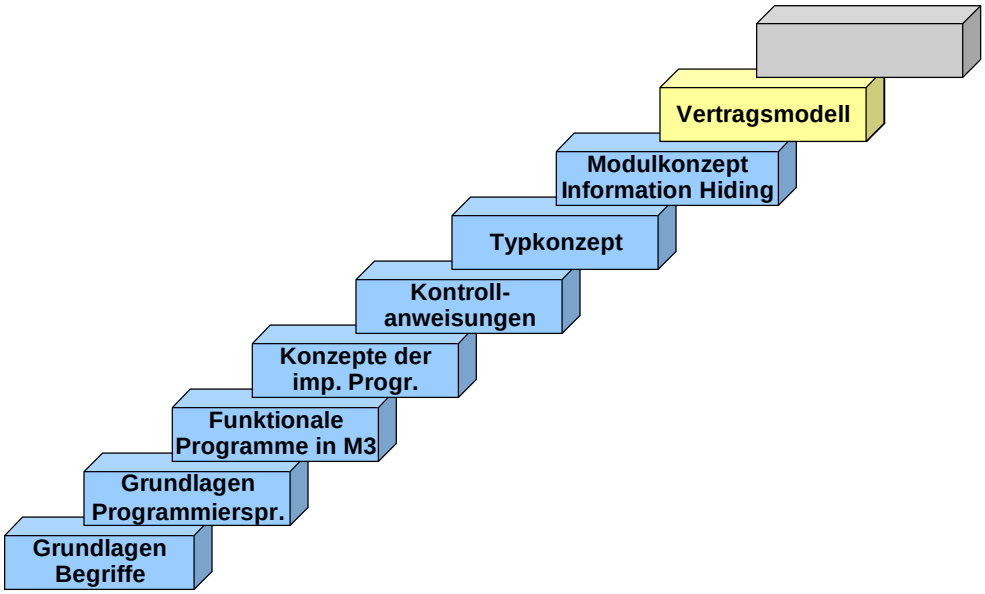
Entwurfskonzept

- **Um ein fachliches Konzept als Objektmodul zu realisieren, stellen wir *drei Arten von Operationen* zur Verfügung.**
- **Prozeduren:**
  - ***verändern*** den ***Zustand*** des Objekts. Meist sind sie von Vorbedingungen abhängig, d.h. sie können nicht in jedem Zustand ausgeführt werden.
- **Funktionen:**
  - liefern Informationen, ***ohne*** den Objektzustand nach außen sichtbar ***zu verändern***.
  - Fachliche Funktionen:
    - liefern ***fachliche Informationen*** und sind vom Objektzustand abhängig.
  - Testfunktionen:
    - ***prüfen*** den Objektzustand und werden zum Prüfen der Vorbedingungen verwendet.

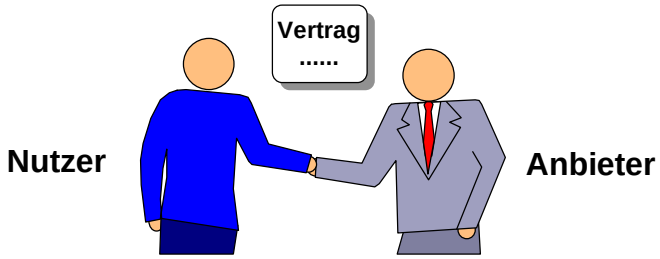
Konzept Abstrakter Datentyp

- **Kann als *Generalisierung* des Objektmoduls (Datenkapsel) betrachtet werden.**
  - Anstatt eines Objektes wird ein Typ (für diese Objekte) definiert.
- **Betrachten wir den ADT als (formale) Spezifikation eines Typs,**
  - dann entwerfen wir ihn durch Angabe von
  - ***Typnamen***
  - ***Signaturen*** (Operationen)
    - zum Erzeugen von Objekten, zum Verändern etc.
  - ***Axiome***
    - formulieren den semantischen Zusammenhang der Operationen.
  - ***Vorbedingungen***
    - geben an, in welchem Zustand welche Operationen gültig sind.

Aufbau der Vorlesung



Idee des Vertragsmodells



- **Vertrag**
  - zwischen Nutzer und Anbieter einer Operation regelt, wer der beiden Partner welche ***Verpflichtungen*** einhalten muß (und welchen ***Nutzen*** er dadurch hat)
- **Operation**
  - arbeitet korrekt, wenn sie vertragsgemäß ***benutzt*** bzw. ***realisiert*** wird.

Zusicherungen - 1

- **Zusicherungen**
  - sind eine Technik, um eine bestimmte Art von Verträgen zwischen Anbieter und Nutzer zu formulieren.
- **Zusicherungen werden formuliert als**
  - ***Vorbedingungen*** für Operationen
  - ***Nachbedingungen*** von Operationen
  - ***Invarianten*** von abstrakten Datentypen
- **Zusicherungen**
  - erhöhen die ***Benutzbarkeit***, indem sie diese formaler definieren
  - verbessern die ***Testbarkeit***
  - verbessern die ***Fehlersuche*** (debugging)
  - verlangen vom Entwickler ein ***abstraktes*** Denken

Realisierung von Zusicherungen in M3

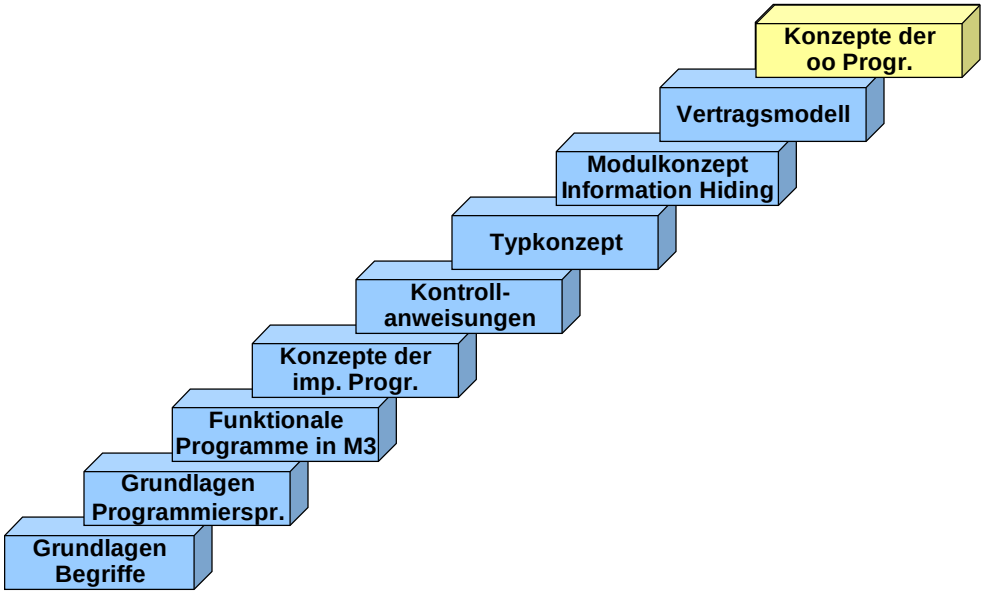
- **Mit Hilfe des Pragmas ASSERT**

```
PROCEDURE EntnehmeText (VAR o: Ordner): TEXT =
VAR t : TEXT;
BEGIN
 <* ASSERT NOT IstLeer(o) *>
 . . .
 <* ASSERT NOT IstVoll(o)  *>
 <* ASSERT Invariante(o)   *>
END EntnehmeText;
```

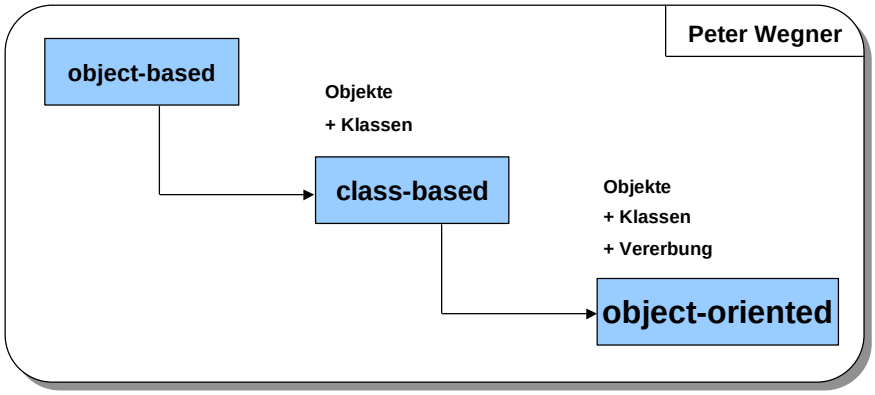
- **Mit Hilfe von Ausnahmen**

```
PROCEDURE EntnehmeText (VAR o: Ordner) : TEXT
                        RAISES {As. Violated} =
VAR t : TEXT;
BEGIN
  As.Require (NOT IstLeer(o), "OrdnerADT.EntnehmeText");
  . . .
  As.Ensure (NOT IstVoll(o), "OrdnerADT.EntnehmeText");
  As.Ensure (Invariante(o), "OrdnerADT.EntnehmeText");
END EntnehmeText;
```

Aufbau der Vorlesung



Merkmale objektorientierter Systeme/Sprachen



Ada
Modula-2

CLU
(Ada)
(Modula-2)

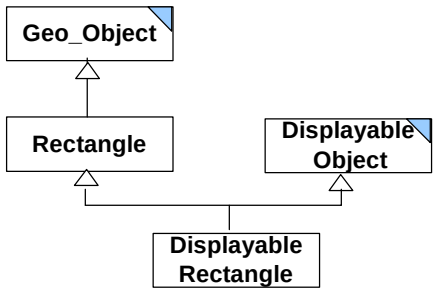
Simula
Smalltalk-80
Eiffel
C++
Java
Modula-3

Objekt, Nachricht, Klasse, Vererbung

- **Ein Objekt ist eine Datenkapsel, die aus zwei Teilen besteht**
  - Der Wert der Daten repräsentiert den **Zustand** des Objekts
  - Daten können nur mithilfe von **Operationen** verändert werden
- **Objekte kommunizieren miteinander,**
  - dadurch daß sie **Nachrichten** versenden und Nachrichten empfangen.
- **Gleichartige Objekte werden in einer Klasse beschrieben**
  - **Speicherstruktur**
  - **Operationen** (Methoden, Routinen)
  - konkrete Klassen, abstrakte Klassen
- **Gemeinsame Eigenschaften verschiedener Klassen**
  - werden in einer **eigenen Klasse** zusammengefaßt und definiert, und anschließend an diese vererbt.
  - Einfachvererbung - Mehrfachvererbung

Beispiel: Polymorphismus - 1

- **Die Vererbung ist ein Mechanismus, um Polymorphismus in Programmiersprachen zu realisieren.**



ein DisplayableRectangle **verhält sich wie** ein Rechteck und wie ein DisplayableObject

```
dr : DisplayableRectangle;
dr.Create;

dr.contains (p);
x := dr.center;
dr.moveTo (p);

s := dr.psDescription
```

Rectangle

DisplayableObject

Das dynamische Binden

```
g : Geo_Object;
r : Rectangle;
c : Circle;

r.Create; c.Create

g := r;
g.contains (p);

g := c;
g.contains (p);
```

```
class Geo_Object

feature
  contains (p : Point) : Boolean is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

feature
    contains (p : Point) :Boolean is
      ...
end -- class Rectangle
```

```
class Circle

feature
  contains (p : Point) :Boolean is  ...
end -- class Circle
```

Dynamisches Binden heißt:

die **richtige** Implementierung zur **Laufzeit** finden

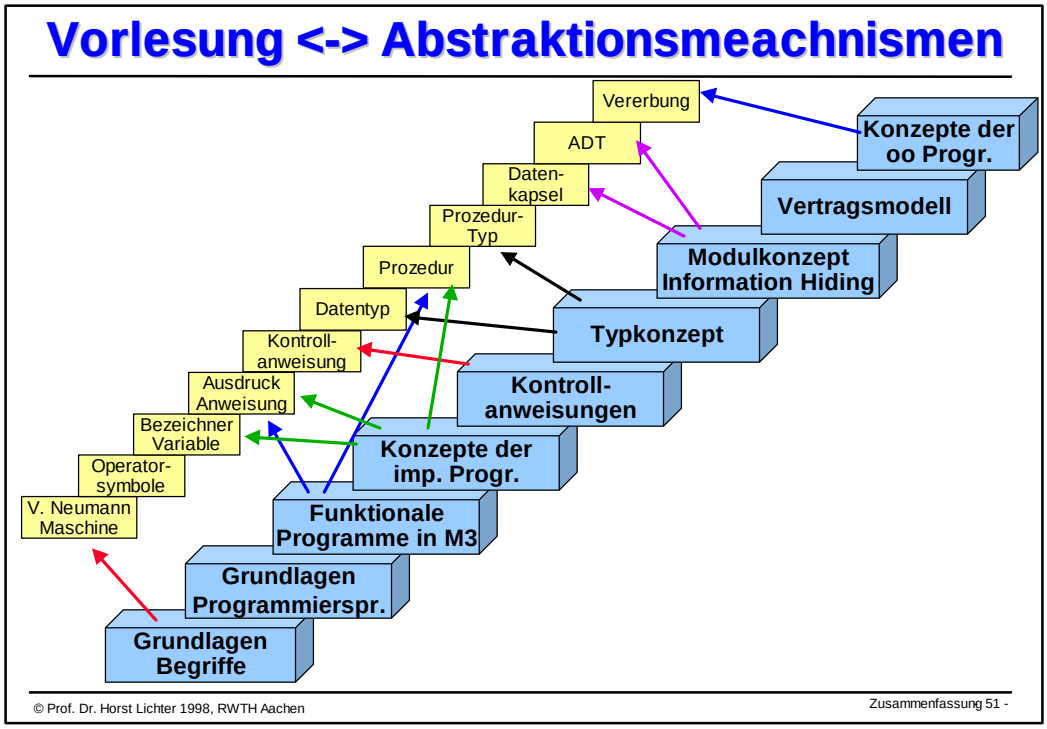
Klassen in Modula-3 - Objekttypen

```
INTERFACE Geo_Object;
IMPORT Point;

TYPE T =  ROOT OBJECT
  METHODS
    moveTo (p : Point.T);
    contains (p : Point.T) : BOOLEAN;
  END;
END Geo_Object.
```

Abstrakte Klasse

```
INTERFACE Rectangle;
IMPORT Geo_Object, Point;

TYPE T <: Public;
     Public = Geo_Object.T OBJECT
              METHODS
                area() : INTEGER;
                height () : INTEGER;
                setOrigin (p : Point.T);
                getOrigin(): Point.T;
                setCorner(p : Point.T);
                getCorner(): Point.T;
                asText () : TEXT;
              END;
END Rectangle.
```

```
INTERFACE DisplayableRectangle;
IMPORT Rectangle;

TYPE T <: Public;
     Public = Rectangle.T OBJECT
              METHODS
                setColor(c : TEXT);
                getColor() : TEXT;
                draw(): TEXT;
              END;
END DisplayableRectangle.
```

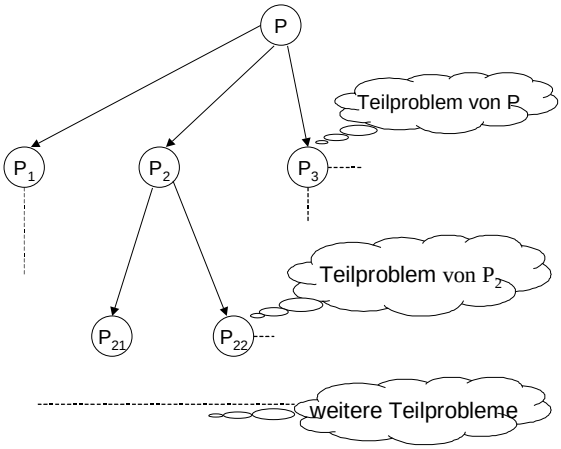

Vorlesung <-> Abstraktionsmeachnismen
Vererbung
ADT
Daten-kapsel
Prozedur-Typ
Prozedur
Datentyp
Kontroll-anweisung
Ausdruck Anweisung
Bezeichner Variable
Operator-symbole
V. Neumann Maschine
Konzepte der oo Progr.
Vertragsmodell
Modulkonzept Information Hiding
Typkonzept
Kontroll-anweisungen
Konzepte der imp. Progr.
Funktionale Programme in M3
Grundlagen Programmierspr.
Grundlagen Begriffe
© Prof. Dr. Horst Lichter 1998, RWTH Aachen
Zusammenfassung 51 -

Exkurs: ENTWURF VON ALGORITHMEN

Der Entwurf eines Algorthmus zur Lösung eines Problems ist in vielen Fällen nicht trivial. Eine systematische Vorgehensweise zahlt sich deshalb gerade in dieser Phase des Programmierprozesses besonders aus. Falls das zu lösende Problem P nicht gerade von ganz primitiver Art ist, so drängt sich die folgende Vorgehensweise geradezu auf:

> "Zerlege das Problem P in Teilprobleme P_1, P_2, P_3, ...,
> löse diese Teilprobleme unabhängig voneinander und
> baue die Teillösungen zu einer Gesamtlösung zusammen.
> Falls nötig, wende diese Strategie auf die Teilprobleme P_1,
> P_2, P_3 , ... sowie deren Zerlegungen an."

In graphischer Form ist diese Strategie in Figur 1 dargestellt.

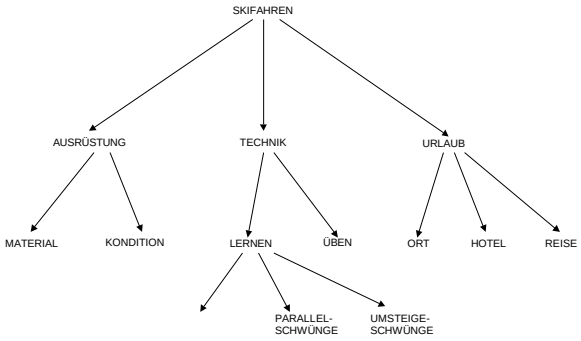


Figur 1: Zerlegung von Problemen

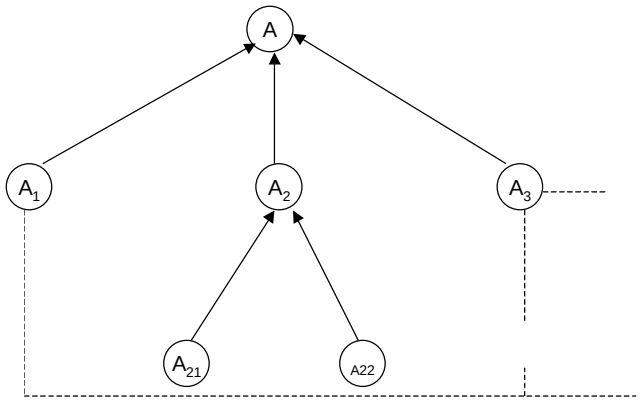
Für die oben beschriebene Vorgehensweise zur Lösung eines Problems werden in der deutsch- und englischsprachigen Literatur oft folgende Begriffe verwendet:

- schrittweise Verfeinerung (stepwise refinement)
- top-down Entwurf (top-down design, top-down-strategy)
- Zerlegungsstrategie (divide-and-conquer, decomposition)

Beispiel: Problem des Skifahrens



Nachdem zu jedem Teilproblem P_1, P_2, P_3, ..., P_{21}, P_{22}, ... eines gemäß Figur 1 zerlegten Problems P ein Lösungsalgorithmus A_1, A_2, A_3, ..., A_{21}, A_{22}, ... gefunden wurde, muß daraus der Lösungsalgorithmus A für das Gesamtproblem P zusammengesetzt werden. (siehe Figur 2)



Figur 2: Kombination von Algorithmen

Die Frage der Zusammensetzung von Algorithmen zu umfangreicheren Algorithmen soll zunächst recht allgemein betrachtet werden. Erstaunlicherweise reichen lediglich drei Konstruktionsschemata aus, um alle überhaupt algorithmisch lösbaren Probleme zu erfassen, nämlich

- die **Hintereinanderausführung** von Algorithmen ("sequentielle Ausführung")
- die **wahlweise Ausführung** von Algorithmen ("alternative Ausführung")
- die **wiederholte Ausführung** von Algorithmen ("iterative Ausführung").

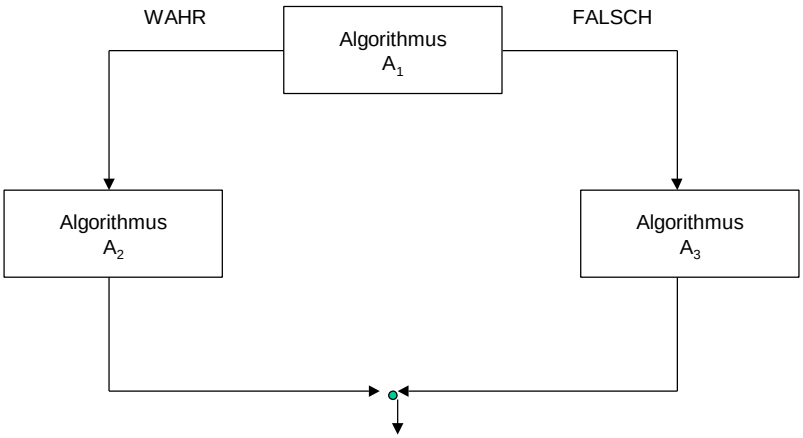
Sind A_1 und A_2 zwei Algorithmen, dann bedeutet die Hintereinanderausführung von A_1 und A_2, daß zunächst A_1 gestartet wird und im Anschluß an seine Beendigung Algorithmus A_2 durchgeführt wird. Figur 3 zeigt diese Anordnung von Algorithmen in graphischer Form.

Figur 3: Hintereinanderausführung von Algorithmen

In linearer Notation wird die sequentielle Anordnung zweier Algorithmen wie folgt bezeichnet:

A_1 ; A_2

Es seien A_1, A_2 und A_3 Algorithmen und A_1 außerdem von der Art, daß er bei seiner Durchführung als Resultat "WAHR oder "FALSCH" liefert. Bei der wahlweisen Ausführung dieser Algorithmen wird durch A_1 zunächst eines der angegebenen Resultate berechnet und in Abhängigkeit davon entweder A_2 oder A_3 gestartet. (Siehe Figur 4)

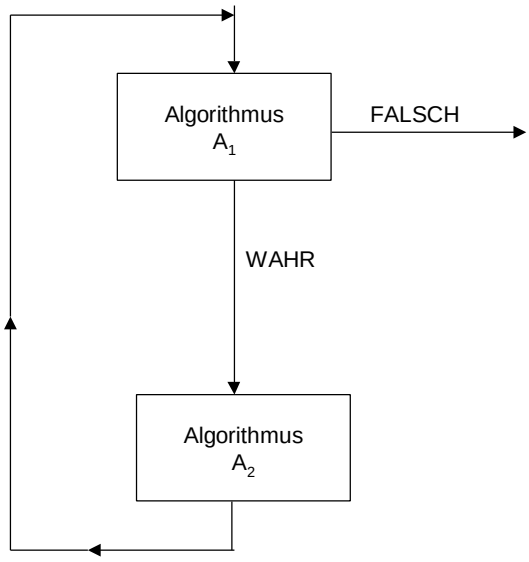


Figur 4 Wahlweise Ausführung von Algorithmen

In linearer Schreibweise wird dieses Prinzip durch

wenn A_1 dann A_2 sonst A_3

ausgedrückt.
Es seien A_1 und A_2 Algorithmen und A_1 liefere als Resultat stets "WAHR" oder "FALSCH". Bei der wiederholten Ausführung dieser beiden Algorithmen werden A_1 und A_2 solange abwechselnd wiederholt, bis A_1 das Resultat "FALSCH" errechnet. In Figur 5 ist diese Vorschrift in graphischer Form dargestellt. Man ersieht daraus, daß A_2 nie zur Ausführung gelangt, falls bereits die erste Aktivierung von A_1 das Resultat "FALSCH" liefert. Um sicherzustellen, daß der durch Iteration aus A_1 und A_2 zusammengesetzte Algorithmus auch irgendwann zum Stillstand kommt, muß darauf geachtet werden, daß A_1 irgendwann das Resultat FALSCH errechnet.



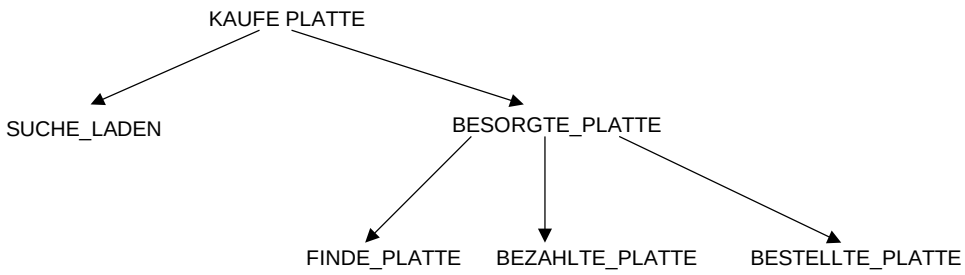
Figur 5 Wiederholte Ausführung von Algorithmen

Die lineare Form des iterativen Schemas ist

solange A_1 tue A_2

Es gibt einige Varianten der oben vorgestellten sequentiellen, alternativen und iterativen Schemata.. Alle diese Varianten können aber auf die drei vorangegangenen Konstruktionsschemata zurückgeführt werden.

Beispiel: Das Problem, eine Schallplatte zu kaufen, läßt sich etwa wie folgt in Teilprobleme aufspalten.



Aus Algorithmen zur Lösung dieser Teilprobleme läßt sich folgender Algorithmus zusammenbauen.

```
SUCHE_LADEN;
wenn PLATTE_GEFUNDEN dann BEZAHLTE_PLATTE
                      sonst BESTELLTE_PLATTE
```

Dabei wurde je einmal das sequentielle und das alternative Schema zum Zusammenbau von Algorithmen verwendet.

Beispiel: (Suchproblem): In einer Liste von 100 Namen soll nach einem vorgegebenen Namen gesucht werden.

LISTE	UDO	MAX	UTE	.....	PETER	...	KARL	EVA
	1	2	3		66		99	100

Die einzelnen Listenplätze sollen durch LISTE[1], LISTE[2], ..., LISTE[100] angesprochen werden. Der gesuchte Name sei mit GESUCHT bezeichnet. Die Einträge in LISTE seien in keiner bestimmten Weise angeordnet, d.h. irgendwie zufällig angeordnet, so daß als einzig vernünftige Vorgehensweise die folgende in Frage kommt ("lineare Suche", "sequentielle Suche"):

```
Vergleiche GESUCHT der Reihe nach mit den Einträgen
LISTE[1], LISTE[2], LISTE[3], ... . Höre auf, falls
GESUCHT mit einem Eintrag LISTE[I]für eine Zahl I
zwischen 1 und 100 übereinstimmt oder das Ende der
Liste erreicht wurde.
```

Mit Hilfe der oben eingeführten Schreibweise läßt sich dies folgendermaßen beschreiben.

```
weise I den Wert 1 zu;
solange I nicht größer ist als 100 tue
   wenn LISTE[1] mit GESUCHT übereinstimmt
      dann weise I den Wert 1000 zu
```

In diesem Beispiel kommt das sequentielle, alternative und iterative Schema zum Zusammenbau von Algorithmen je einmal vor.

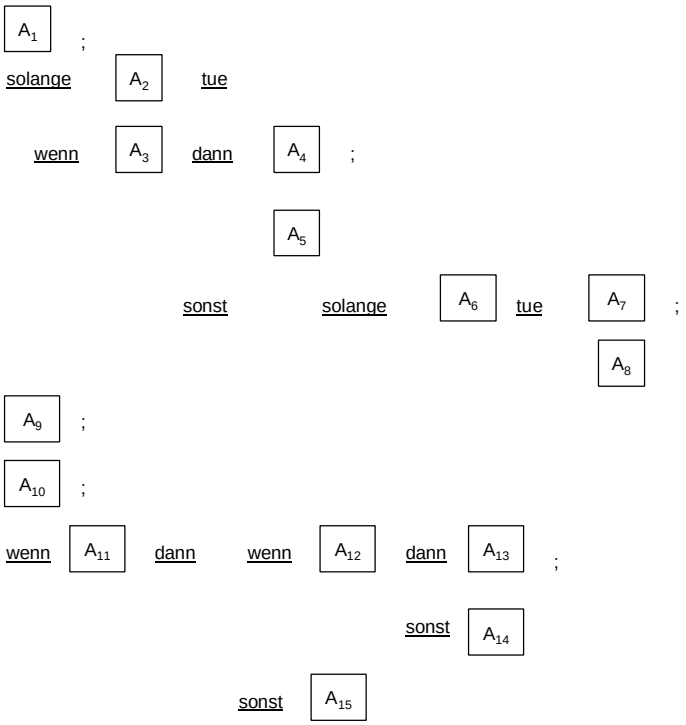
Das Aufbaumuster des Algorithmus aus dem vorangegangenen Beispiel sieht so aus:

A_1;
solange A_2 tue A_3

wobei A_3 einen Algorithmus der Form

wenn A_4 dann A_5
sonst A_6

darstellt. Mit anderen Worten: Die drei Aufbauschemata für Algorithmen dürfen ineinander verschachtelt werden. Dabei ist die zulässige Schachtelungstiefe nach oben unbeschränkt, so daß durchaus ein Muster wie das in Figur 6 dargestellte entstehen kann.



Figur 6: Mehrfach geschachtelter Aufbau von Algorithmen

Exkurs: Dateien

- **Dateisystem**
- **Operationen auf Dateien**
  - lesen
  - schreiben
- **Dateien in Modula-3**
  - wichtige Datei-Operationen

Dateien: Zweck

- **Verarbeiten von Daten**
  - Daten müssen in vielen Fällen dauerhaft (***persistent)*** gespeichert werden
- **Bisher:**
  - Daten wurden im ***Arbeitsspeicher*** zur Laufzeit des Programms erzeugt
  - Nachdem das Programm beendet ist, sind diese Daten ***verloren***
  - "***flüchtiger***" Speicher
- **Hintergrundspeicher**
  - persistenter Speicher
  - Diskette, CD, Festplatte etc.
- **Frage:**
  - Wie können wir Daten aus dem Arbeitsspeicher in den Hintergrundspeicher bringen und umgekehrt?

Dateisystem

- **Betriebssystem:**
  - stellt ***Dienstleistungen*** zur Verfügung, damit der Umgang mit dem Rechner einfach und auf ***hohem Abstraktionsniveau*** möglich ist
  - eine angebotene Dienstleistung des Betriebssystems ist das ***Dateisystem***
- **Dateisystem**
  - erlaubt, den Hintergrundspeicher in ***einzelnen Bereiche*** aufzuteilen
  - diese nennt man ***Dateien*** (file)
  - Dateien können in sogenannten ***Verzeichnissen*** gruppiert werden (directory)
  - jede Datei hat einen ***Namen***
  - aus der Sicht des Rechners ist eine Datei eine Folge von ***Informationseinheiten*** (bytes)
  - ein Verzeichnis verwaltet für alle seine Dateien den Namen und die Position, wo die Dateien auf dem Hintergrundspeicher physisch liegen

Operationen auf Dateien

- **Dateien können**
  - angelegt und gelöscht werden
  - gelesen und geschrieben werden
- **Dabei gibt es zwei Modi für das Lesen und Schreiben:**
  - ***Sequentiell***
    - Daten werden von Anfang bis zum Schluß der Datei ***nacheinander*** gelesen (geschrieben).
    - Es gibt keine Möglichkeit einen Ausschnitt der Datei zu ***überspringen***
  - ***Direkt***
    - Es kann eine ***Position*** angegeben werden, ab der gelesen (geschrieben) werden soll
    - Wechseln der Position ist ***beliebig*** möglich
- **Das Betriebssystem**
  - kennt für jede Datei einen Zähler, der die aktuelle Position kennzeichnet.

Arbeiten mit Dateien

- **Programmiersysteme**
  - bieten in der Regel Möglichkeiten und Mechanismen, um vom Programm aus ***Dateioperationen*** ausführen zu können
  - dabei werden i.d.R. nicht die Funktionen des Betriebssystems benutzt
  - Programmiersysteme stellen eine ***abstrakte Schnittstelle*** zum Dateisystem zur Verfügung
- **Wichtige Dateioperationen**
  - Datei ***öffnen***
    - öffnen zum Lesen
    - öffnen zum Schreiben (von vorne)
    - öffnen zum Schreiben (neue Zeichen werden hinten angehängt)
  - neue Datei ***erzeugen***
  - ***lesen*** und ***schreiben***
  - Abfragen des ***Dateiendes*** (end of file, EOF)
  - Datei ***schließen***

Dateien in Modula-3

- **Sprachumgebung von Modula-3**
  - stellt Module zur Verfügung, um Dateien manipulieren zu können
- **Ein- / Ausgabestrom**
  - Mit einem Ein- / Ausgabestrom kann auf eine Datei zugegriffen werden
  - In der Standardbibliothek sind diese definiert
    - `reader, writer`
  - Ein Ein- / Ausgabestrom kann bei seiner Initialisierung mit einer Datei verbunden werden

```
IMPORT IO, Rd, Wr;
VAR eingabestrom : Rd.T;
    ausgabestrom : Wr.T;
…
eingabestrom := IO.OpenRead ("eingabe.txt");
ausgabestrom := IO.OpenWrite("ausgabe.txt");
```

Beispiel: Kopieren einer Datei

```
IMPORT SIO,IO;
IMPORT Wr AS Writer, Rd AS Reader;

PROCEDURE KopiereDatei (original, kopie : TEXT)=
VAR original_datei : Reader.T;
    kopie_datei    : Writer.T;
    zeile : TEXT;
BEGIN
  original_datei := IO.OpenRead (orginal);
  kopie_datei := IO.OpenWrite(kopie);
  WHILE NOT IO.EOF(original_datei) DO
    zeile := SIO.GetLine(original_datei);
    SIO.PutLine(zeile, kopie_datei);
  END;
  Reader.Close(original_datei);
  Writer.Close(kopie_datei);
END KopiereDatei;
```


Wichtige Dateioperationen in M3

- **Modul: IO**
  - PROCEDURE **EOF** (rd: Rd.T := NIL): BOOLEAN;
    - *Return TRUE iff rd is at end-of-file.*
  - PROCEDURE **OpenRead** (f: TEXT): Rd.T;
    - *Open the file name f for reading and return a reader on its contents. If the file doesn't exist or is not readable, return NIL.*
  - PROCEDURE **OpenWrite** (f: TEXT): Wr.T;
    - *Open the file named f for writing and return a writer on its contents. If the file does not exist it will be created. If the process does not have the authority to modify or create the file, return NIL.*
- **Modul: Rd und Wr**
  - PROCEDURE **Close** (rd: T) RAISES {Failure, Alerted};
    - *Release any resources associated with rd and set closed(rd) := TRUE.*
  - PROCEDURE **Close** (wr: T) RAISES {Failure, Alerted};
    - *Flush wr, release any resources associated with wr, and set closed(wr) := TRUE.*