Ludwig-Maximilians-Universität München, Department „Institut für Informatik“
Lehr- und Forschungseinheit für Datenbanksysteme
Prof. Dr. Hans-Peter Kriegel

Folien zur Vorlesung

Informatik II

im Sommersemester 2002

Prof. Dr. Hans-Peter Kriegel

Organisatorisches

- Vorlesung: Di. 9 – 11, Hauptgebäude 201
  Do. 9 – 11, Hauptgebäude 101
  Pause: ja, Beginn?
- Folien: http://www.dbs.informatik.uni-muenchen.de/Lehre/InfoII

Die Vorlesung basiert auf dem Buch:

Hanspeter Mössenböck: Sprechen Sie Java? dpunkt.verlag, 2001.
ISBN 3-89864-117-1

- Übungen: http://www.dbs.informatik.uni-muenchen.de/Lehre/InfoII

Literaturhinweise

- Hanspeter Mössenböck: *Sprechen Sie Java?* dpunkt.verlag, 2001. ISBN 3-89864-117-1
- Klaus Echtle und Michael Goedicke: *Lehrbuch der Programmierung mit Java*. dpunkt.verlag 2000. ISBN 3-932588-22-3
- Judith Bishop: *Java™ lernen*. 2. Auflage, Addison-Wesley, 2001. ISBN 3-8273-1794-0
- Ken Arnold, James Gosling, David Holmes: *The Java™ Programming Language*. 3. Auflage, Addison-Wesley, 2000. ISBN 0201704331
- Mary Campione et al.: The Java™ Tutorial: *A Short Course on the Basics*, 3. Auflage, http://java.sun.com/docs/books/tutorial/
- Hermann Engesser, Volker Claus, Andreas Schwill: *Duden Informatik. Ein Fachlexikon für Studium und Praxis*. 3. Auflage, Bibliographisches Institut Mannheim, 2001. ISBN 3411052333

Kapitel 1: Einführung

- Zentraler Begriff „Algorithmus“
- Eigenschaften von Algorithmen
- Programme und Prozesse
- Notation von Algorithmen

Zentraler Begriff *Algorithmus*

Ein Algorithmus ist ein Verfahren mit einer *präzisen* (d.h. in einer genau festgelegten Sprache formulierten), endlichen Beschreibung unter Verwendung *effektiver* (d.h. tatsächlich ausführbarer) *elementarer* Verarbeitungs-schritte.
Zu jedem Zeitpunkt der Abarbeitung des Algorithmus benötigt der Algorithmus nur endlich viele Ressourcen.

Beispiel für Algorithmus

- Aufgabe: Berechne die Summe der Zahlen von 1 bis *n*.
- Algorithmus *SummeBis*(*n*):

  setze *summe* := 0
  setze *zähler* := 1
  solange *zähler* ≤ *n*:
      setze *summe* := *summe* + *zähler*
      erhöhe *zähler* um 1
  gib aus: „Die Summe ist: “ und *summe*

Bestandteile von Algorithmen

- Ein Algorithmus verarbeitet *Daten* mit Hilfe von *Anweisungen*.
  - „Programm = Daten + Befehle“
- Daten
  - *Werte*: Zahlen (im Bsp. 0, 1), Strings (“Die Summe ist: “), Zeichen (‘a‘), …
  - *Variablen*: Benannte Behälter für Werte (*summe*, *zähler*, *n*)
- Anweisungen
  - Zuweisung: **setze** *n* := 1
  - bedingte Anweisung: **falls** ⟨Bedingung⟩**:** ⟨Anweisung⟩
  - Folgen von Anweisungen: ⟨Anweisung 1⟩**;** …**;** ⟨Anweisung *k*⟩
  - Schleifen: **solange** ⟨Bedingung⟩**:** ⟨Anweisung⟩

Basiseigenschaften von Algorithmen

- Allgemeinheit
  - Lösung einer Klasse von Problemen, nicht eines Einzelproblems.
- Operationalität
  - Einzelschritte sind wohldefiniert und können auf entsprechend geeigneten Rechenanlagen ausgeführt werden.
- Endliche Beschreibung
  - Die Notation des Algorithmus hat eine endliche Länge.
- Funktionalität
  - Ein Algorithmus reagiert auf Eingaben und produziert Ausgaben.

Weitere Eigenschaften von Algorithmen

• Terminierung	Alg. läuft für jede Eingabe nur endlich lange
• Vollständigkeit	Alg. liefert alle gewünschten Ergebnisse
• Korrektheit	Alg. liefert nur richtige Ergebnisse
• Determinismus	Ablauf ist für dieselbe Eingabe immer gleich
• Determiniertheit	Ergebnis ist festgelegt für jede Eingabe
• Effizienz	Alg. ist sparsam im Ressourcenverbrauch
• Robustheit	Alg. ist robust gegen Fehler aller Art
• Änderbarkeit	Alg. ist anpassbar an modifizierte Anforderungen

Algorithmen: Terminierung

- Ein *terminierender* Algorithmus läuft für jede (!) beliebige Eingabe jeweils in endlicher Zeit ab.
- Gegenbeispiel
  - fak(n): falls n = 0, liefere 1
    sonst liefere n * fak(n–1)

  - Ergebnis von fak(2)?
  - Ergebnis von fak(–3)?

Algorithmen: Vollständigkeit

- Ein *vollständiger* Algorithmus gibt die gewünschten Ergebnisse vollständig aus.
- Gegenbeispiel
  - Teilermenge (n):
    setze i := 1;
    solange $i < \sqrt{n}$:
        falls n/i ganzzahlig, gib i und n/i aus;
        erhöhe i um 1;

  - Ausgabe von Teilermenge(12)?
  - Ausgabe von Teilermenge(9)?

Algorithmen: Korrektheit

- Ein *korrekter* Algorithmus liefert nur richtige Ergebnisse.
- Gegenbeispiel
  - Teilermenge‘ (n):
    setze i := 1;
    solange $i \leq$ sqrt(n):
        gib i und n/i aus;
        erhöhe i um 1;

  - Ausgabe von Teilermenge‘(12)?
  - Ausgabe von Teilermenge‘(9)?

Algorithmen: Determinismus

- Ein *deterministischer* Algorithmus läuft für ein- und dieselbe Eingabe immer auf dieselbe Art und Weise ab.
- Ein *nichtdeterministischer* Algorithmus kann für ein- und dieselbe Eingabe unterschiedlich ablaufen.
- Beispiel:
  - Absolutbetrag (x):
    wähle Fall $\begin{cases} \text{falls } x \leq 0, \text{ liefere } -x \\ \text{falls } x \geq 0, \text{ liefere } x \end{cases}$
  - Ergebnis von Absolutbetrag(3)?
  - Ergebnis von Absolutbetrag(-5)?
  - Ergebnis von Absolutbetrag(0)?

Algorithmen: Determiniertheit

- Ein *determinierter* Algorithmus liefert für ein- und dieselbe Eingabe immer dasselbe Ergebnis.
- Gegenbeispiel
  - RotGrünBlau (n):
    wähle Fall: $\begin{cases} \text{falls n/2 ganzzahlig:} & \text{liefere „rot“} \\ \text{falls n/3 ganzzahlig:} & \text{liefere „grün“} \\ \text{sonst:} & \text{liefere „blau“} \end{cases}$
  - Ergebnis von RotGrünBlau(5)?
  - Ergebnis von RotGrünBlau(6)?
  - Wichtiges nicht-determiniertes Beispiel: Zufallszahlengenerator

Algorithmen: Effizienz

- Für eine gegebene Eingabe sollen die benötigten Ressourcen möglichst gering (oder sogar minimal) sein.
  - Betrachtung von *Speicherplatz* und *Rechenzeit* (Anzahl Einzelschritte).
  - Ggf. auch Analyse von *Plattenzugriffen* (I/Os) oder *Netzzugriffen*.
- Beispiel
  - Die iterative Berechnung der Summe von 1 bis n benötigt n Schritte.
  - Verwende Summenformel $n*(n+1)/2$ für einen effizienteren Algorithmus.
- Unterscheide Effizienz und Effektivität
  - Effektivität ist „Grad der Zielerreichung“ (Vollständigkeit, Korrektheit).
  - Effizienz ist „Wirtschaftlichkeit der Zielerreichung“.

Beispiel: Suche in einem Array

- Aufgabe
  Sei $A[1] \ldots A[n]$ eine Reihung („*Array*“) von n verschiedenen Zahlen.
  Suche die Position i einer gegebenen Zahl x, also das i mit $A[i] = x$.
- Algorithmus *SequentielleSuche* (A, x):
  Durchlaufe das Array A der Reihe nach für $i := 1, ..., n$:
  Falls $A[i] = x$, gib i aus und beende den Durchlauf.
  Falls x nicht gefunden, gib Fehlermeldung „nicht gefunden“ aus.
- Anzahl der Vergleiche (= Analyse der Laufzeit)
  - Erfolgreiche Suche: n Vergleiche maximal, $n/2$ im Durchschnitt
  - Erfolglose Suche: n Vergleiche

Beispiel: Binäre Suche

Falls das Array sortiert ist, kann man auch wie folgt suchen:

BinäreSuche (A, x)

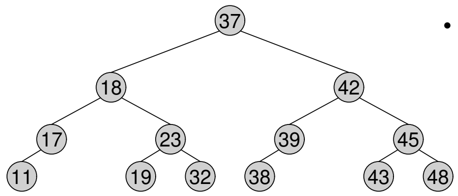
- Vergleiche den Eintrag $A[i_{mitte}]$ in der Mitte von A mit x:
  - Falls $x = A[i_{mitte}]$, gib i_{mitte} aus und beende die Suche.
  - Falls $x < A[i_{mitte}]$, suche in der linken Hälfte von A weiter.
  - Falls $x > A[i_{mitte}]$, suche in der rechten Hälfte von A weiter.
- In der jeweiligen Hälfte wird ebenfalls mit *BinäreSuche* gesucht.
- Falls die neue Hälfte des Arrays leer ist, gib die Fehlermeldung „nicht gefunden“ aus.

Beispiel für die binäre Suche

- Beispiel

11	17	18	19	23	32	37	38	39	42	43	45	48

- Entscheidungsbaum



- Analyse der Laufzeit
  - Entscheidungsbaum hat Höhe $h = \log_2(n+1)$.
  - Suche benötigt maximal h viele Vergleiche.
- Vergleich der Suchverfahren
  - Beispiel n = 1.000
    - Sequentiell: 1.000 Vergleiche
    - Binäre Suche: 10 Vergleiche
  - Beispiel n = 1.000.000
    - Sequ.: 1.000.000 Vergleiche
    - Binäre Suche: 20 Vergleiche

Algorithmen: Robustheit

- Der Ablauf soll auch für fehlerhafte Eingaben oder Daten wohldefiniert sein.
- Beispiele
  - Bedienfehler
    - Leere Eingaben in (grafische) Eingabefelder
    - Eingabe von Strings in Eingabefelder für Zahlen
  - Systemfehler
    - Zugriff auf Ressourcen (Dateien, Netz, etc.) nicht möglich

Algorithmen: Änderbarkeit

- Einfache Anpassung an veränderte Aufgabenstellungen.
- Beispiele
  - Erweiterung einer Adressenverwaltung für inländische Kunden auf Kunden im Ausland
  - Umstellung der PLZ von vier auf fünf Stellen
  - Einführung des EURO
  - „Jahr 2000“ Problem (vierstellige statt zweistellige Jahreszahlen)

Programme und Prozesse

- Programm
  - Beschreibung eines Ablaufes, der auf einer Rechenanlage durchgeführt werden kann (= Algorithmus, *statisch*).
  - Programme sind (meist) in Dateien gespeichert.
  - Programme entstehen durch Konstruktion.
- Prozess
  - Konkreter Ablauf eines Programms (*dynamisch*).
  - Prozesse benötigen Betriebsmittel (Rechenzeit, Speicher, Dateizugriffe).
  - Prozesse werden durch das Betriebssystem verwaltet.
  - Prozesse entstehen durch Aufrufe von Programmen.

Übersetzung von Programmen

- Problemstellung
  - Programme werden meist in höheren Programmiersprachen formuliert.
  - Die Hardware kann nur Programme in Maschinensprache ausführen.
  - Abbildung höherer Programmiersprachen in Maschinensprachen nötig.
- Zwei Vorgehensweisen
  - *Übersetzung:*
    Programme werden vor dem Ablauf übersetzt (compiliert).
  - *Interpretation:*
    Programme werden während des Ablaufs interpretiert.

Übersetzung und Interpretation

- Übersetzung
  - Das Programm wird nur einmal analysiert (*statisch*).
  - Das resultierende (Maschinen-)Programm wird gespeichert.
  - Die Ausführung ist in der Regel schneller als bei Interpretation.
- Interpretation
  - Das Programm wird bei jeder Ausführung analysiert (*dynamisch*).
  - Die Ausführung ist in der Regel langsamer als bei Übersetzung.
  - Die Zyklen „Programmänderung   Ausführung“ sind kürzer.

Übersetzungs- und Interpretationshierarchie

- Die Abbildung auf maschinennähere Sprachen kann sich über mehrere Stufen erstrecken.
- Beispiel Java:

Programm in Java-Quellcode (nicht unmittelbar ausführbar)

↓ Übersetzung

Programm in Java-Bytecode (läuft auf *Java Virtual Machine*)

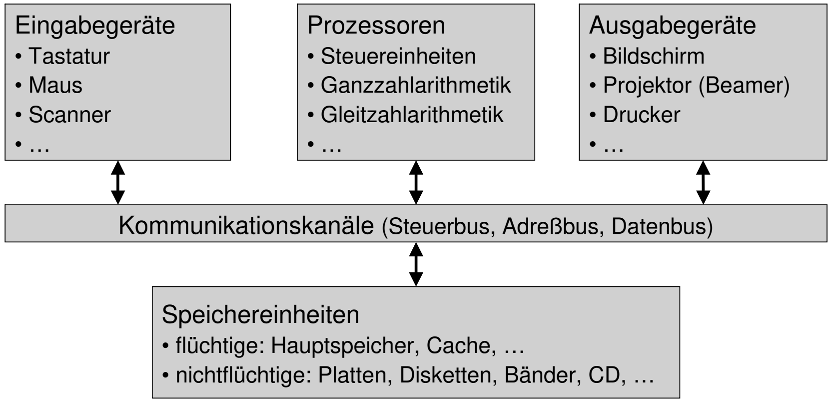
↓ Interpretation

Programm in Maschinensprache (läuft auf konkreter Hardware)

Hardware und Software

- Hardware
  - „Harte Ware“: Geräte
  - Prozessor, Speicher (flüchtig/nicht-flüchtig), Ein-/Ausgabegeräte
  - Modellbeispiel: „von Neumann-Rechner“
- Software
  - „Weiche Ware“: Programme
  - Systemprogramme zur Steuerung der Hardware
  - Anwendungsprogramme zur Bearbeitung von Nutzeraufgaben
- Firmware
  - In Hardware gegossene Software (Gerätesteuerungen, Mikroprogramme)

Rechnermodell von John von Neumann



Notationen für Algorithmen

- Informell
  - Natürliche Sprache
  - Pseudocode (halbformell)
- Graphische Darstellungen
  - Programmablaufplan (Flussdiagramm, DIN 66001)
  - Struktogramm (Nassi-Shneiderman-Diagramm, DIN 66261)
- Formale Sprachen (Programmiersprachen)
  - Programm als formaler Text

Beispielalgorithmus *SummeBis*(*n*)

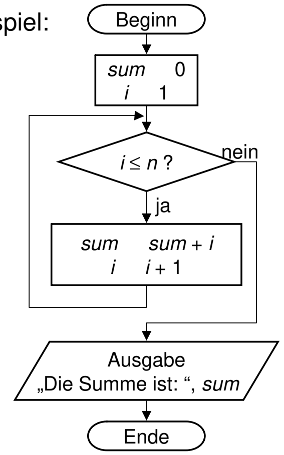
- Aufgabe: Berechne die Summe der Zahlen von 1 bis *n*.
- Natürliche Sprache

  Initialisiere eine Variable *summe* mit 0. Durchlaufe die Zahlen von 1 bis *n* mit einer Variable *zähler* und addiere *zähler* jeweils zu *summe*. Gib nach dem Durchlauf den Text “Die Summe ist: “ und den Wert von *summe* aus.
- Pseudocode

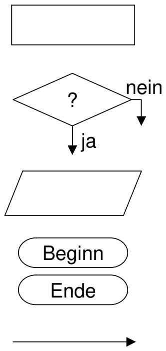
  setze *summe* := 0;
  setze *zähler* := 1;
  solange *zähler* ≤ *n*:
          setze *summe* := *summe* + *zähler*;
          erhöhe *zähler* um 1;
  gib aus: “Die Summe ist: “ und *summe*;

Programmablaufplan für *SummeBis*(*n*)

Beispiel:



Symbole:



Aktion

Verzweigung

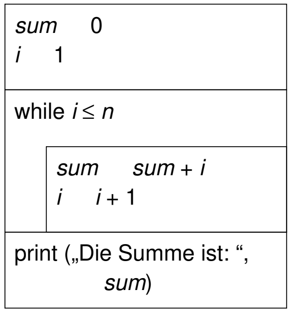
Ein-/Ausgabe

Anfangs- und Endpunkt

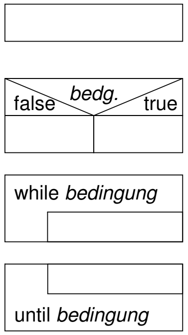
Ablauflinien

Struktogramm für *SummeBis*(*n*)

Beispiel:



Symbole:



Strukturblock

Verzweigung

abweisende Schleife („*solange noch*“)

Durchlauf-Schleife („*solange bis*“)

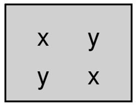
Programmiersprache *Java*

```
class SummeBis {
   public static void main (String[] arg) {
        int n = Integer.parseInt(arg[0]);

        int sum = 0;
        int i = 1;
        while (i <= n) {
          sum = sum + i;
          i = i + 1;
        }
        System.out.println („Die Summe ist: “ + sum);
   }
}
```

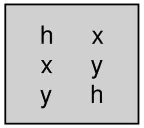

„Schreibtischtest“ für Algorithmen

- Veranschaulichung der Funktionsweise eines Algorithmus durch ein *Ablaufprotokoll* der Variableninhalte.
- Beispiel: Vertausche die Inhalte der Variablen x und y
- Versuch:



x	y
3	5
5	
	5

- Lösung: mit Hilfsvariable



x	y	h
3	5	
		3
5		
	3	

Anzahl der Dezimalziffern einer Zahl

Algorithmus *NumDigits*(*n*)

d 1
while *n* > 9   *n* *n* / 10   *d* *d* + 1
Ausgabe von *d*

„Schreibtischtest"

n	d
437	1
43	2
4	3

Korrektheit: Euklidischer Algorithmus

- Aufgabe: größter gemeinsamer Teiler zweier Zahlen
- Lösung von Euklid (um 300 v.Chr.)

- Euklid (in *a*, in *b*, out *ggt*):

rest *a* mod *b*
while *rest* ≠ 0   *a* *b*   *b* *rest*   *rest* *a* mod *b*
ggt *b*

Nachweis der Korrektheit

(ggt teilt *a*) ∧ (ggt teilt *b*)

ggt teilt (*a* – *b*)

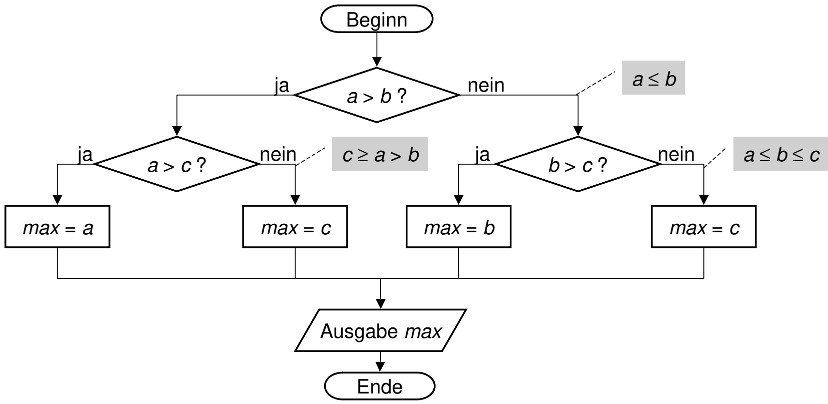
ggt teilt (*a* – *k***b*)

ggt teilt *rest*

GGT (*a*,*b*) = GGT (*b*,*rest*)

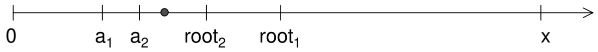
Verwendung von Zusicherungen

Beispiel: Maximum dreier Zahlen max(*a*,*b*,*c*)



Nicht-ganzzahlige Arithmetik

- Bsp: Berechnung der Quadratwurzel durch schrittweise Näherung



root *x* / 2 *a* *x* / *root*
while *a* ≠ *root*   //-- *a* * *root* = *x*   *root* (*root* + *a*) / 2   *a* *x* / *root*   //-- *a* * *root* = *x*
//-- *a* = *root* Ausgabe *root*

x	root	a
10	5	2
	3.5	2.85714
	3.17857	3.14607
	3.16232	3.16223
	3.16228	3.16228

Problem mit Terminierung:
a = *root* wird (fast) nie erreicht,
besser |*a* – *root*| > 1.0e-8 testen.

Formale Sprachen

- Syntax
  - Regeln, nach denen Programme aufgeschrieben werden dürfen.
  - Beispiel: *Zuweisung ::= Variable „ “ Ausdruck*
    - A 3 + 5
    - B 2
    - 7 4
- Semantik
  - Bedeutung von korrekt gebildeten Programmen.
  - Im Beispiel: *Weise den Wert des Ausdrucks an die Variable zu*
    - Nach „A 3 + 5“ enthält A den Wert der Summe 3+5, also 8.

EBNF: Erweiterte Backus-Naur-Form

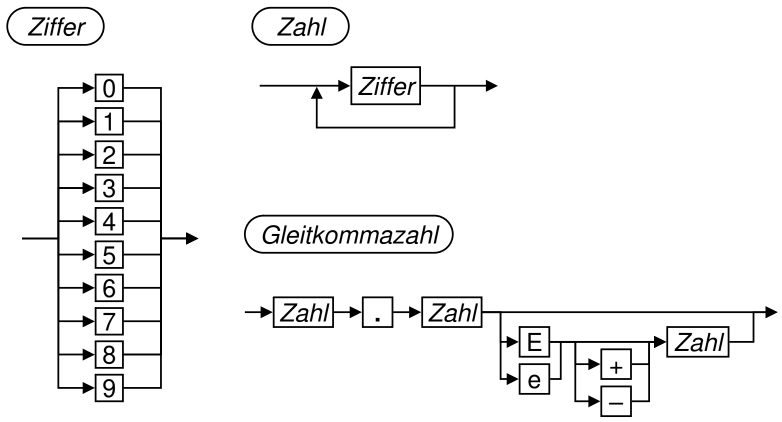
- EBNF: einfache textuelle Darstellung von Grammatiken
  - John Backus: Entwickler der Programmiersprache *Fortran*
  - Peter Naur: Herausgeber des Algol60-Reports
- Metazeichen für EBNF-Regeln

		Bsp.	steht für
=	trennt Regelseiten		
.	schließt Regel ab		
|	trennt Alternativen	x | y	x, y
( )	klammert Alternativen	(x | y) z	xz, yz
[ ]	wahlweises Vorkommen	[x] y	xy, y
{ }	0..*n*-maliges Vorkommen	{x} y	y, xy, xxy, …

Beispiel: Grammatiken für Zahlen

- Beispiel: Grammatik für ganze Zahlen (ohne Vorzeichen: 1, 4711, …)
  Ziffer = “0“ | “1“ | “2“ | “3“ | “4“ | “5“ | “6“ | “7“ | “8“ | “9“.
  Zahl = *Ziffer* { *Ziffer* }.
- Aufbau von Regeln für Grammatiken
  - *Terminalsymbole* werden nicht mehr weiter zerlegt (im Bsp. in “ “).
  - *Nichtterminalsymbole* werden ersetzt (im Bsp. kursiv).
  - Auf der linken Seite einer Regel steht immer ein Nichtterminalsymbol.
  - Rechte Seite kann Terminalsymbole und Nichtterminalsymbole enthalten.
- Beispiel: Gleitkommazahlen (ohne Vorzeichen: 1.0E-2, 3.14159E0, …)
  Gleitkommazahl = *Zahl* “.“ *Zahl* [(“E“ | “e“) [“+“ | “–“] *Zahl*].

Grafische Alternative: Syntaxdiagramme



Programmierung und Softwareentwicklung

- „Programmierung im Kleinen“
  - Konstruktion von Teilprogrammen
  - Anweisungen, Prozeduren, Objekte, Klassen
  - Zusammenfassung in Bibliotheken
- „Programmierung im Großen“
  - „Software Engineering“
  - Konstruktion großer Programmsysteme mit Hilfe von Bibliotheken
  - Vorgehensmodelle zur Zerlegung und Integration

Paradigmen der Programmierung

- Imperative Programmierung
  - Folgen von Anweisungen, Verzweigungen, Schleifen, Prozeduraufrufe
    - Beispiele: BASIC, Pascal, Modula2, …
- Objektorientierte Programmierung
  - Klassen beschreiben Struktur und Verhalten interagierender Objekte
    - Beispiele: Simula, Smalltalk, C++, Java, …
- Logik- oder deklarative Programmierung
  - Automatische Ableitung der Lösung aus der Beschreibung der Aufgabe
    - Beispiele: Prolog, …
- Funktionale oder applikative Programmierung
  - Rekursive Anwendung von Funktionen
    - Beispiele: LISP, SML, Scheme, …