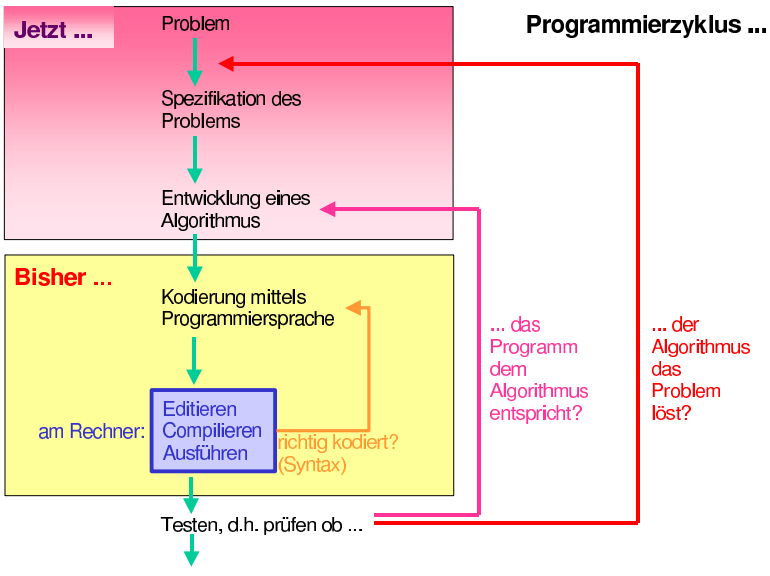
## Teil IV: Vom Problem zum Programm

1. Einordnung
2. Entwurf und Spezifikation
3. Modularisierung und Algorithmenentwicklung

# Einordnung

## Vorbemerkungen



**Ausgangspunkt ...**

**Programm ... ? – ein Problem ?**

**Feststellung :**

Es gibt keinen Algorithmus
für den Entwurf von Algorithmen !

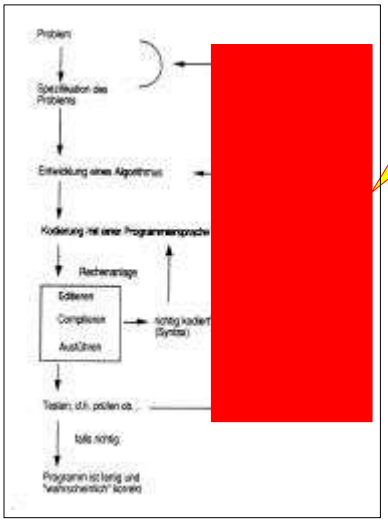
**Was ist Problemlösen ?**

Kreativer Prozess, der ein umfangreiches Verständnis der gestellten Aufgabe und viel Intuition, Erfahrung und Praxis erfordert !

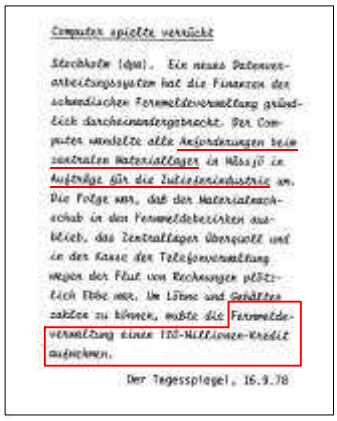
„Wäre das Programmieren ein strikt deterministischer Prozeß, der nach festen Regeln abläuft, so wäre es bereits seit langem automatisiert worden."

(aus N. Wirth. Systematisches Programmieren – Eine Einführung, 5. Aufl. B.G. Teubner, Stuttgart, 1985, pp. 120ff)

Im Falle fehlerhafter Spezifikationen drohen mitunter hohe Kosten ...




... Fehler können sehr teuer werden !



Computer spielte verrückt

Stockholm (dpa). Ein neues Datenverarbeitungssystem hat die Finanzen der schwedischen Fernmeldeverwaltung gründlich durcheinandergebracht. Der Computer wandelte alle Anforderungen beim zentralen Materiallager in Mässjö in Aufträge für die Zulieferindustrie um. Die Folge war, daß der Materialnachschub in den Fernmeldebezirken ausblieb, das Zentrallager überquoll und in der Kasse der Telefonverwaltung wegen der Flut von Rechnungen plötzlich Ebbe war. Um Löhne und Gehälter zahlen zu können, mußte die Fernmeldeverwaltung einen 110-Millionen-Kredit aufnehmen.

Der Tagesspiegel, 16.9.78

(aus R. Kimm, W. Koch, W. Simonsmeier, F. Tontsch. Einführung in Software Engineering. Walter de Gruyter, Berlin, 1979)

## Software Engineering

→ Softwaretechnik, Softwaretechnologie

**Inhalt und Zielsetzung**

Anwendung von Prinzipien, Methoden und Techniken auf die Planung, den Entwurf, die Implementierung und die Wartung von Programmen und Programmsystemen

→ vorwiegend ingenieurmässig ablaufender Prozess !

**Vorgehensmodelle**

**Ziel**: Benennung und Ordnung von produktbezogenen Aktivitäten bei der Softwareentwicklung

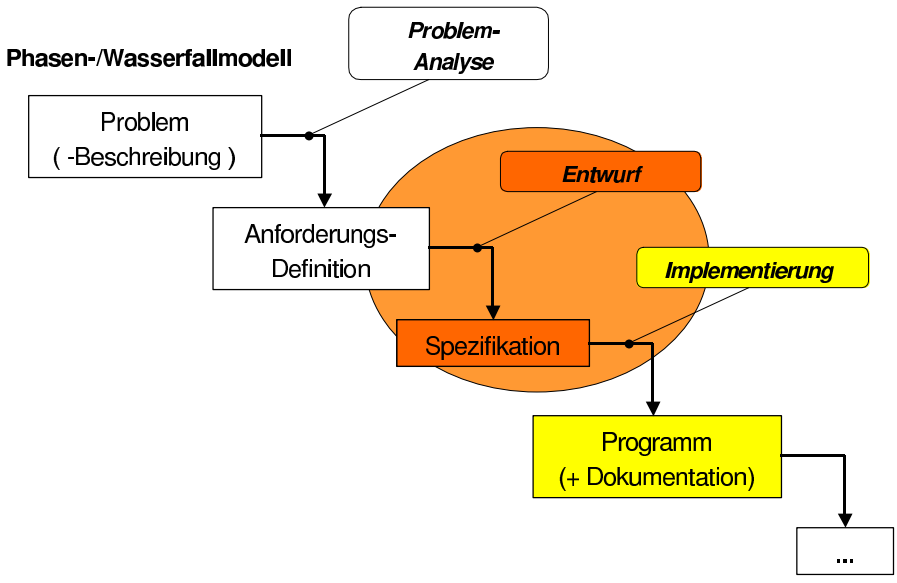
Verschiedene Modelle des „Software Life Cycle“

- **Lineares Phasenmodell / Wasserfallmodell**

  → grundlegendes Projektmodell, das die Softwareherstellung als Folge von Aktivitäten (Phasen) beschreibt !

- **Spiralmodell**
- **Zyklisches Modell**
- **Evolutionäre Systementwicklung und Prototyping**

**Phasen-/Wasserfallmodell**



# Entwurf und Spezifikation

## Entwurf

**Zielsetzung**

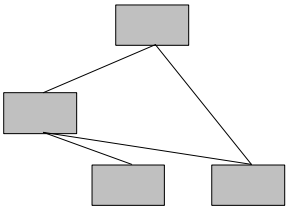
Ein **Entwurf** beinhaltet die
Entwicklung eines Modells des Gesamtsystems,
das die Anforderungen erfüllt – und damit das realisierte
Programm (bzw. die realisierten Programme) die
Problemlösung realisiert (realisieren) !

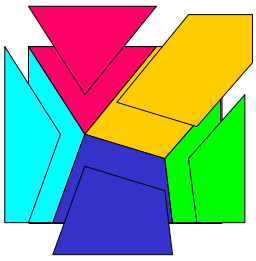
Also ...
eine eindeutige, vollständige und in der Komplexität bewältigbare Systemspezifikation

1. **Anforderungsdefinition**: Beschreibung der Aufgabe
2. **Spezifikation**: Lösung der Aufgabe
   (→ weitere Detaillierungen notwendig, ehe die Lösung ausführbar wird !)

**Elementare Vorgehensweise**

- **Zerlegung** eines komplexen Gesamtsystems
  - voneinander unabhängig realisierbare Bausteine (**Module**)
  - Beschreibung des Zusammenwirkens über **Schnittstellen**

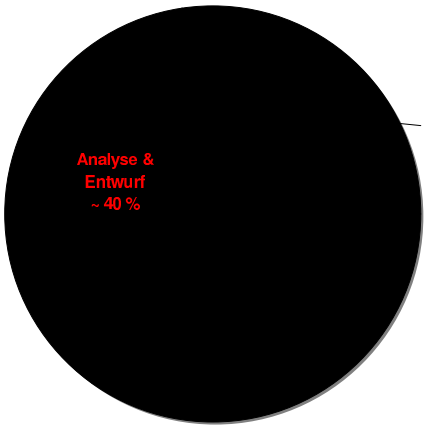




Zerlegungsprinzipien folgen verschiedenen Prinzipien, z.B. Unabhängigkeit der individuellen Entwicklung, Verständlichkeit, Änderbarkeit.

- Hierarchische **Modularisierung**
  - „Top-down" Methode (→ schrittweise Verfeinerung)
  - „Bottom-up" Methode

**Relativer Zeitbedarf der Phasen der Softwareentwicklung** (Abschätzung)




(vgl. R. Kimm, W. Koch, W. Simonsmeier, F. Tontsch.
Einführung in Software Engineering. Walter de Gruyter, Berlin, 1979)

**Entwurf – „Fallbeispiel“**

**Auftrag an einen Bautrupp** (aus Kimm et al., 1979):

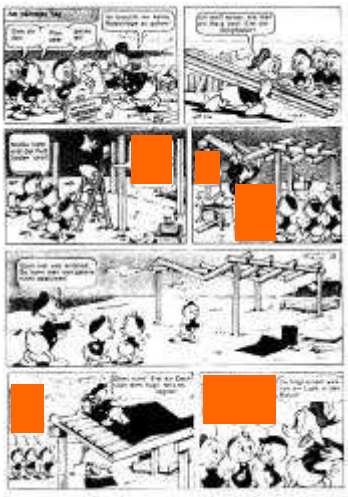
„Baue ein 10-geschossiges Hochhaus mit 10 3-Zimmer-Wohnungen (à 80m$^2$), 20 2-Zimmer-Wohnungen (à 55m$^2$) und 10 1-Zimmer-Wohnungen (à 35m$^2$). Jede Wohnung soll ausser den Wohnräumen ein Bad, eine Toilette und eine Küche besitzen. Darüberhinaus soll das Haus alle üblichen Service-Einrichtungen aufweisen.“

Die Anforderungs-Definition ist zu ungenau ...

**Einschätzung :**

Was würde man denken, wenn der Trupp aufgrund dieses Auftrags anrückt und an einer Ecke beginnt, eine Grube auszuheben, Zement hineinzuschütten und die ersten Steine daraufzusetzen ... ?

Etwa so wie hier ...



Micky Maus, Heft 32/71, © Walt Disney Productions

**Einordnung:**

⇒ **Metapher für die häufig anzutreffende Vorgehensweise beim Programmieren !**

**Prinzip** : keine Zeit verlieren und Programme so schnell wie möglich auf eine Maschine bringen ...

---

**Programmieren ≠ Hacken !!**

## Spezifikation

**Inhalt**

Eine **Spezifikation** ist eine **vollständige**, **detaillierte** und **eindeutige** Darstellung

der Eigenschaften, der Verhaltensweisen, des Zusammenwirkens und des Aufbaus

von Programmen und Systemen

unter
Abstrahierung von Details und konkreter Implementierung

**Ziel** : Entwicklung eines klaren Verständnisses des Problems, seiner Eigenschaften und seiner Lösungen

Die **Spezifikation** ist ...

„ ... eines der zentralen Probleme,

wenn nicht **das** zentrale Problem überhaupt

in der industriellen Softwareherstellung ... “

(R. Kimm, W. Koch, W. Simonsmeier, F. Tontsch.
Einführung in Software Engineering. Walter de Gruyter, Berlin, 1979)

**Grundsätzliche Qualitätsmerkmale**

**Vollständigkeit und Widerspruchsfreiheit :**

- alle geforderten Funktionen des Programms und die relevanten Rahmenbedingungen für die Problemlösung müssen beschrieben werden
- die Komponenten müssen exakt aufeinander abgestimmt sein

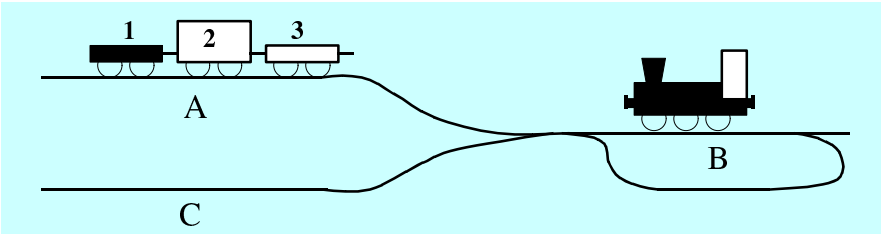
**Detailliertheit, Minimalität und Verständlichkeit :**

- Angabe aller zur Problemlösung notwendigen Entwurfsentscheidungen – aber keine weiteren ...
- Problem: Überspezifikation, d.h. z.B. die Aufnahme von Implementierungsentscheidungen (unabhängig vom Problem !)
  ➔ Einschränkung der Menge der Lösungsmöglichkeiten
- Nachvollziehbarkeit der Problemlösung für Programmierer

**Eindeutigkeit :**

- Angabe klarer Kriterien, wann eine vorgeschlagene Lösung akzeptabel ist

**Beispiel 1 – Rangierproblem**

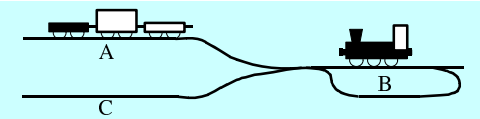



**Spezifikation :**

Eine Lokomotive soll die im Gleisabschnitt A befindlichen Wagen 1, 2, 3 in der Reihenfolge 3, 2, 1 auf Gleisstück C abstellen

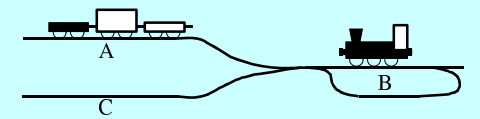
**Beobachtung :**

Die Problembeschreibung genügt noch **nicht** den Anforderungen an eine **Spezifikation**: Sie ist unvollständig, nicht genügend detailliert und nicht eindeutig !




**Der Weg zu einer korrekten Spezifikation ...**

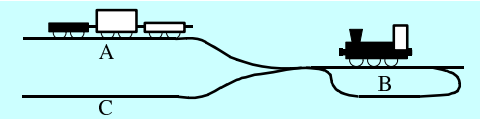
- **Vollständigkeit und Widerspruchsfreiheit :**
  - Wieviele Wagen kann die Lok auf einmal ziehen ?
  - Wieviele Wagen passen auf Gleisstück B ?




**Der Weg zu einer korrekten Spezifikation ...**

- **Vollständigkeit und Widerspruchsfreiheit :**
  - Wieviele Wagen kann die Lok auf einmal ziehen ?
  - Wieviele Wagen passen auf Gleisstück B ?

- **Detailliertheit, Minimalität und Nachvollziehbarkeit :**
  - Welche Aktionen / Operationen kann die Lok ausführen ?
    (z.B. fahren, koppeln, entkoppeln, Weichen stellen)




**Der Weg zu einer korrekten Spezifikation ...**

- **Vollständigkeit und Widerspruchsfreiheit :**
  - Wieviele Wagen kann die Lok auf einmal ziehen ?
  - Wieviele Wagen passen auf Gleisstück B ?
- **Detailliertheit, Minimalität und Nachvollziehbarkeit :**
  - Welche Aktionen / Operationen kann die Lok ausführen ?
    (z.B. fahren, koppeln, entkoppeln, Weichen stellen)
- **Eindeutigkeit :**
  - Ist es erlaubt, dass die Lok am Ende zwischen den Wagen steht ?
  - Ist es erlaubt, dass die Lok am Ende hinter den Wagen steht ?

**Beispiel 2 – Grösster gemeinsamer Teiler 2er Zahlen (ggT)**

**Spezifikation :**

> Für beliebige Zahlen A und B berechne den grössten gemeinsamen Teiler ggT(A,B), d.h. die grösste Zahl, die sowohl A als auch B teilt

**Vollständigkeit und Widerspruchsfreiheit :**

Welche Zahlen sind für A und B zugelassen ?

(→ ganze, rationale Zahlen, sind auch negative Zahlen und 0 zugelassen ?)

**Detailliertheit, Minimalität und Verständlichkeit :**

Welche Operationen sind erlaubt ?

(→ +, - oder auch DIV, MOD ?) → siehe Euklid'scher Algorithmus

**Eindeutigkeit :**

- Wie sollen fehlerhafte Eingaben für A oder B behandelt werden ?
- Soll das Ergebnis ausgedruckt werden ?

# Modularisierung und Algorithmenentwicklung

## Entwurfsentscheidungen und Modularisierung

**Prinzip**

Zerlegung eines grossen Problem in Teilprobleme

Forderung:

- klare Abgrenzung der Teilprobleme
- getrennte Bearbeitung muss möglich sein
- weitgehende Unabhängigkeit der Funktionalität
  → Teamarbeit möglich ?!
- Lösungen mit autonomer Funktion
  → keine Seiteneffekte

Bsp.: Telegrammabrechnung

Aufgabe:

- Einlesen eines Telegramms, Bestimmung von (ID-Code, Wortzahl, Preis) und Eintragung in Abrechnungstabelle
- Berechnung der Grössen ‚Gesamtwortzahl' und ‚Gesamtpreis'
- Ausgabe der aktualisierten Tabelle

**Eingabe-Format** :

Form (als regulärer Ausdruck ...)

( zzzzzz (b)+ ((c)*(b))* <STOP> (b)+ <STOP> (b)+ )+

mit z ∈ {0, ..., 9}, b : Leerzeichen, c : Druckzeichen,
(.)+ : Metazeichen zur 1- oder mehrmaligen Wiederholung,
(.)* : Metazeichen zur 0- oder mehrmaligen Wiederholung

Beispiel :

635444 WEIHNACHTEN IST NAHE STOP STOP 634560 IM
WINTERSEMESTER GIBT ES EINE ERHOLUNGSPAUSE
IN DER MITTE DER VORLESUNGSZEIT STOP UND BALD
IST ES SOWEIT STOP STOP

**Analyse:**

a) Telegramm :
- 6stelliger ID-Code
- Menge von Wörtern (getrennt durch Leerzeichen)
- STOP sind Begrenzungszeichen (zählen nicht mit ...)

b) Preis :
- pro begonnenes Zeichen eines Wortes : 0,60 DM
- Mindestpreis : 4,20 DM

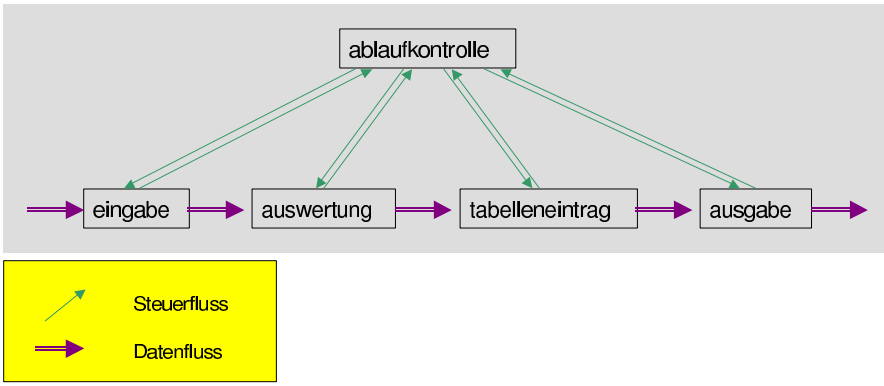
Wort ~ eine in Leerzeichen eingeschlossene Zeichenkette unter 12 Zeichen (längere Zeichenketten zählen pro begonnene 12 Zeichen ein Wort mehr !)

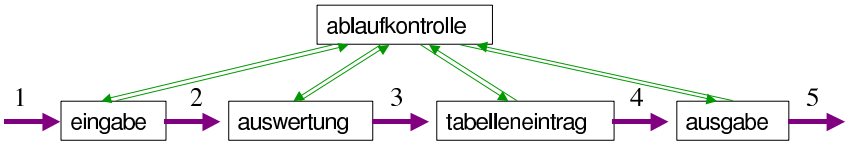
c) Erstellung einer Abrechnungstabelle;

Inhalt :
- für jedes Telegramm : ID-Code, Wortzahl, Preis
- insgesamt : Gesamtwortzahl, Gesamtpreis

**1. Lösungsansatz :** → Zerlegung nach zeitlicher Verarbeitungsfolge

Ablauf :






Wechsel der Ausführung (Kontrolle)

Datenfluss

1 : Telegramm (externe Darstellung)
2 : Telegramm (interne Repräsentation)
3 : Abrechnungstriple
4 : Tabelle (interne Darstellung)
5 : Tabelle (externe Darstellung)

1. MODUL eingabe
   Einlesen eines Telegramms
2. MODUL auswertung
   - Wortzahl z feststellen
   - Preis p berechnen
   - Bereitstellung desTripels (ID-Code, z, p)
3. MODUL tabelleneintrag
   - Eintragung desTripels in geordneter Tabelle
   - Aktualisierung von Gesamtzahl und–preis
4. MODUL ausgabe
   Ausgabe der Tabelle
5. MODUL ablaufkontrolle
   - Aufruf der einzelnen Module in richtiger Reihenfolge
   - Fehlerbehandlung

**Probleme** mit diesem Entwurf und seiner Modularisierung :

**Kriterien:**

1. **Unabhängigkeit der Entwicklung**

   Die Entwickler der Module müssen zuvor die verwendeten Datenformate für die Telegramme definieren und anschliessend präzise anwenden

2. **Verständlichkeit des Gesamtsystems und der Module**

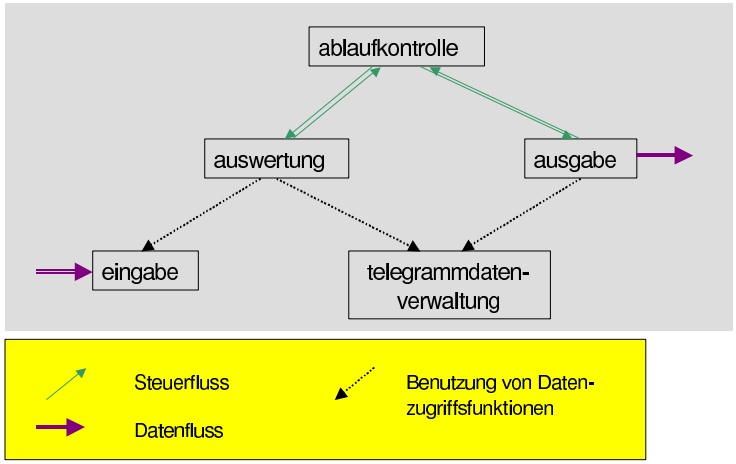
   Zum Verständnis der Funktionalitäten müssen die Datenformate bekannt sein und auch die Funktionsabläufe in den anderen Moduln

3. **Änderbarkeit**

   Mögliche Änderungen der Entwurfsentscheidungen:
   - Format des Identifizierungscodes
   - Implementierung der Telegrammdaten-Tabelle (Baum, verkettete lineare Liste, Tripel von Feldern)

**2. Lösungsansatz :** → Zerlegung nach dem Prinzip des „Verbergens“ von Information („information hiding“)

Ablauf :




1. MODUL eingabe
   Funktionen, die den Code und die Wörter des Telegramms lesen können
2. MODUL telegrammdaten-verwaltung
   Tabelle der Abrechnungsdaten und Gesamtdaten
   Funktionen zum
   - Einfügen von Abrechnungsdaten in die Tabelle,
   - das geordnete Lesen der Abrechnungsdaten,
   - die Aktualisierung und das Lesen der Gesamtdaten
3. MODUL auswertung
   Berechnung der Abrechnungsdaten eines Telegramms
   (unter Verwendung der von den Moduln „eingabe“ und „telegrammdaten-verwaltung“ zur Verfügung gestellten Funktionen)
4. MODUL ausgabe
   Erstellung der formatierten Tabelle der Abrechnungsdaten
   (unter Verwendung der Lesefunktionen des Moduls „telegrammdaten-verwaltung“)
5. MODUL ablaufkontrolle
   - Aufruf der einzelnen Module in richtiger Reihenfolge
   - Fehlerbehandlung

**Probleme gelöst ... ?**

**Kriterien:**

1. **Unabhängigkeit der Entwicklung**
   - → benötigt Spezifikationen der Zugriffsoperationen
2. **Verständlichkeit des Gesamtsystems und der Module**
   - → kein Modul benötigt Kenntnisse über Interna der anderen Module
3. **Änderbarkeit**
   - → Einzelne Zugriffsfunktionen innerhalb eines betroffenen Moduls müssen ggf. angepasst werden, jedoch ohne andere Implementierungsdetails zu beeinflussen

⇒ Datenstrukturen werden mit ihren Zugriffsoperationen in Moduln isoliert und verpackt

→ **Abstrakte Datentypen** (→ **Allgemeine Informatik II**)

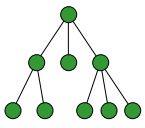
## Funktionelle Dekomposition – Schrittweise Verfeinerung

**Prinzip**

Funktionelle Dekomposition → „Top-down" Methode mittels schrittweiser Verfeinerung

Zerlegung so lange bis die Lösung genügend detailliert ist (~ häufig korrespondierend mit einer Notation in einer Programmiersprache oder einer nachempfundenen Pseudo-Codierung)

**Konzept :**



- . startet (auf der obersten Ebene) mit den Hauptkomponenten des Programms
- . Verwendung von im wesentlichen unstrukturierten Komponenten
- . diese Komponenten werden anschliessend auf der nächsten Ebene verfeinert
- . die noch nicht genügend detaillierten Elemente werden auf der wiederum nächsten Ebene weiter verfeinert ...

**Prinzip :** Die wichtigsten funktionellen Einheiten eines Programms werden zuerst betrachtet, die Betrachtung der Details wird auf einen späteren Zeitpunkt aufgeschoben

## Beispiel – Notizbuch bereinigen

### Motivation

- Schrittweise Verfeinerung
- Prozeduren (→ Hilfsmittel der starken Abstraktion ...)

### Problembeschreibung:

- In einem Notizbuch haben sich im Laufe der Zeit die Namen von einigen hundert Menschen (Bekannte, Freunde, Verwandte, Kollegen, ...) angesammelt. Am Anfang haben wir uns – vielleicht – bemüht, die Namen alphabetisch aufzuschreiben, aber nach einiger Zeit haben wir sie dann an recht willkürlichen Stellen eingetragen. Es ist dann auch vorgekommen, dass ein Name nicht schnell genug zu finden war und daher nochmals aufgenommen worden ist.
- Jetzt haben wir uns ein neues Adressbuch gekauft und wir wollen unser Notizbuch bereinigen, mit
  - . der Anlage einer alphabetischen Liste der Namen
  - . Jeder Name (Eintrag) darf nur einmal vorkommen

**Methode :**

„Top-down"-Entwurf mittels schrittweiser Verfeinerung

### Ebene 1 :

**Vorgehensweise :** . Suche nach dem alphabetisch ersten Namen in der Liste
. Name dort auslöschen;
. Name hinten an die neue Liste anhängen.

```
Notizbuch-bereinigen:
   WHILE alte-liste-ist-noch-nicht-leer DO
      trage-namen-in-neue-liste-ein
   END.
```

### Ebene 2 :

Verfeinerung des Teilalgorithmus

```
hole-alphabetisch-ersten-namen-aus-der-alten-liste
```

**Notwendige Schritte :**

. Die gesamte alte Liste muss durchsucht werden;
dabei muss für jeden Namen festgestellt werden, ob er alphabetisch vor allen anderen schon betrachteten Namen kommt
(→ Variable : ‚**`erster-name`**')

. Der gefundene Name muss aus der alten Liste gelöscht werden

**Notwendige Schritte :**

. Die gesamte alte Liste muss durchsucht werden;
dabei muss für jeden Namen festgestellt werden, ob er alphabetisch vor allen anderen schon betrachteten Namen kommt
(→ Variable : ‚**`erster-name`**')

. Der gefundene Name muss aus der alten Liste gelöscht werden

**Strategien :** 1. Alte Liste wird nochmals durchlaufen, bis ein Name angetroffen wird, der mit ‚**`erster-name`**' übereinstimmt; dieser wird dann gelöscht

2. Beim Suchen von ‚**`erster-name`**' wird sich immer die Position des Antreffens gemerkt (Index !), so dass dieser Eintrag zum Schluss ohne erneutes Suchen gelöscht werden kann

3. Beim Suchen von ‚**`erster-name`**' diesen sofort löschen, wenn er angetroffen wird

**Notwendige Schritte :**

. Die gesamte alte Liste muss durchsucht werden;
dabei muss für jeden Namen festgestellt werden, ob er alphabetisch vor allen anderen schon betrachteten Namen kommt
(→ Variable : ‚**`erster-name`**')

. Der gefundene Name muss aus der alten Liste gelöscht werden

**Strategien :** 1. Alte Liste wird nochmals durchlaufen, bis ein Name angetroffen wird, der mit ‚**`erster-name`**' übereinstimmt; dieser wird dann gelöscht

2. Beim Suchen von ‚**`erster-name`**' wird sich immer die Position des Antreffens gemerkt (Index !), so dass dieser Eintrag zum Schluss ohne erneutes Suchen gelöscht werden kann

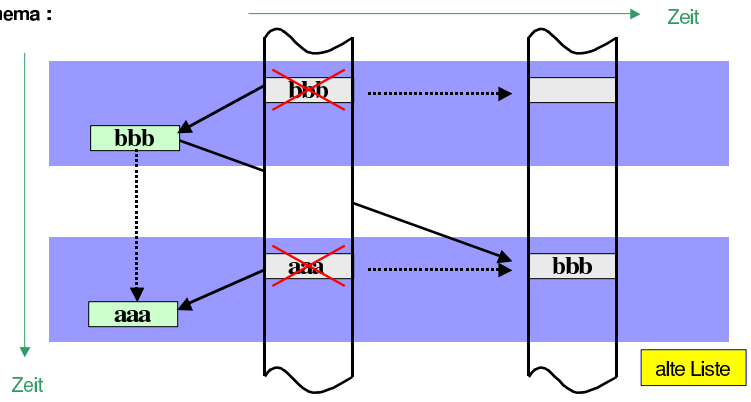
3. Beim Suchen von ‚**`erster-name`**' diesen sofort löschen, wenn er angetroffen wird

**Problem :** Während des Durchlaufs wechselt der Wert von ‚**`erster-name`**', d.h. erst am Ende des Durchlaufs steht fest, welcher der aktuelle alphabetisch erste Name ist (→ die zu einem Zeitpunkt aufgenommenen und dann wieder verworfenen Kandidaten dürfen nicht gelöscht werden !)

**Lösungsvorschlag :**

In solchen Fällen wird der (neue) erste Name ausgelöscht (aber gemerkt) und an diese Stelle in der alten Liste der vorher entfernte Name geschrieben !
(→ die alte Liste gerät damit nur noch mehr durcheinander, macht aber nichts ...)

**Ablauf-Schema :**



Verfeinerung :

```
Hole-alphabetisch-ersten-namen-aus-der-alten-liste :
   erster-name :=  erster-name-in-der-alten-liste;
   loesche-erster-name-aus-der-alten-liste;
   WHILE die-alte-liste-ist-noch-nicht-leer DO
      dieser-name :=  folgender-name-aus-der-alten-liste;
      IF
         THEN  loesche-dieser-name-aus-der-alten-liste;
               schreibe-an-diese-stelle-erster-name;
               erster-name :=  dieser-name
      ENDIF (* -- bis jetzt erster name ist aus der alten Liste geloescht *)
   END.(* -- endgueltig erster name ist aus der alten Liste geloescht *)
```

**Ebene 3 :**

Verfeinerung des Teilalgorithmus (aus Ebene 2)

```
dieser-name-kommt-alphabetisch-vor-erster-name
```

**Betrachtung :**

Ein Name setzt sich stetsaus einem (oder mehreren) Vornamen und dem Nachnamen zusammen. Bei gleichen Nachnamen müssen die Vornamen auf ihre alphabetische Reihenfolge untersucht werden.

```
Dieser-name-kommt-alphabetisch-vor-erster-name :
   IF nachname-von-dieser-name <> nachname-von-erster-name
      THEN  kommt-nachname-von-dieser-name-vor-nachname-
            von-erster-name?
      ELSE  kommt-vorname-von-dieser-name-vor-vorname-von-
            erster-name?
   END.
```

Hier müsste natürlich noch spezifiziert werden, wie die Aktion aussehen soll, wenn der Test positiv ausfällt !

Die Operation (~ Algorithmus) des Namensvergleichs (hier: Feststellung der Ordnung) muss also möglicherweise auf unterschiedlichen Objekten (Nachname, Vorname) durchgeführt werden. Die weitere Verfeinerung ergibt einen eigenen Algorithmus :

```
Kommt-wort1-vor-wort2? :
   Eingabe :  wort1, wort2
   Ausgabe :  {TRUE, FALSE}

   IF wort1-vor-wort2
      THEN  TRUE
      ELSE  FALSE
   ENDIF.
```

**Hinweis :**

Die Einführung dieses neuen Typs von (Unter-) Programm ist eine wichtige Neuerung im Vergleich zu den bisherigen Verfeinerungen: Es werden Anweisungen für verschiedene Parameter wiederverwendet !

```
Notizbuch-bereinigen:

   WHILE alte-liste-ist-noch-nicht-leer DO
      hole-alphabetisch-ersten-namen-aus-der-
            alten-liste;
      [redacted]
   END.
```

**Ebene 2 (cont'd) :**

Verfeinerung des Teilalgorithmus (aus Ebene 1)

**Trage-namen-in-neue-liste-ein**

Die neue Liste ist alphabetisch geordnet. Ein Name kann in der alten Liste **mehrfach** vorkommen, soll aber nur einmal in die neue Liste eingetragen werden.

```
Notizbuch-bereinigen:

   WHILE alte-liste-ist-noch-nicht-leer DO
      hole-alphabetisch-ersten-namen-aus-der-
            alten-liste:
      [redacted]
   END.
```

**Ebene 2 (cont'd) :**

Verfeinerung des Teilalgorithmus (aus Ebene 1)

**`Trage-namen-in-neue-liste-ein`**

Die neue Liste ist alphabetisch geordnet. Ein Name kann in der alten Liste **mehrfach** vorkommen, soll aber nur einmal in die neue Liste eingetragen werden.

Strategien : 1. Von mehrfach auftretenden Namen in der alten Liste (Variable ‚**`erster-name`**' enthält den aktuell ersten Namen) werden die Positionen gemerkt und bei endgültiger Feststellung des ersten Namen nach Durchlauf durch die Liste alle weiteren gleichen Einträge gelöscht

2. Der jetzige Algorithmus wird „stereotyp" wiederholt. Mehrfacheinträge ergeben dann in nacheinanderfolgenden Suchläufen gleichlautende erste Namen der alten Liste. Um Mehrfacheinträge in der neuen Liste zu verhindern, wird vor Übernahme der Name jeweils mit dem Eintrag am Ende der neuen Liste verglichen.

Wegen der eleganteren – allerdings u.U. Auch zeitaufwendigeren – algorithmischen Lösung wird Strategie 2 verfolgt !

**Achtung :** Beim ersten Eintrag in die neue Liste kann der notwendige Vergleich nicht durchgeführt werden, da diese noch leer ist !

Verfeinerungen :

```
Trage-namen-in-neue-liste-ein :

   IF neue-liste-noch-leer
      THEN [redacted]
      ELSE
   ENDIF.

Trage-ein :

   haenge-erster-name-an-neue-liste-an.

Pruefe-und-trage-ein :

   IF erster-name <> letzter-eintrag-in-neuer-liste
      THEN  haenge-erster-name-an-neue-liste-an
   ENDIF.
```

## Zusammenfassung des gesamten Algorithmus (Übersicht)

```
Name des Algorithmus :  Liste-bereinigen
   Eingabe :  alte Liste
   Ausgabe :  neue Liste

   WHILE alte-liste-ist-noch-nicht-leer DO
      hole-alphabetisch-ersten-namen-aus-der-alten-liste;
      trage-namen-in-neue-liste-ein
   END.

(* -- Verfeinerungen *)

Hole-alphabetisch-ersten-namen-aus-der-alten-liste :

   erster-name :=  erster-name-in-der-alten-liste;
   loesche-erster-name-aus-der-alten-liste;
   WHILE es-gibt-noch-weitere-namen-in-der-alten-liste DO
      dieser-name :=  folgender-name-aus-der-alten-liste;
      IF dieser-name-kommt-alphabetisch-vor-erster-name
         THEN  loesche-dieser-name-aus-der-alten-liste;
               schreibe-an-diese-stelle-erster-name;
               erster-name :=  dieser-name
      ENDIF
   END.
```

## Cont'd.

```
Dieser-name-kommt-alphabetisch-vor-erster-name :

   IF nachname-von-dieser-name <> nachname-von-erster-name
      THEN  kommt-wort1-vor-wort2?(nachname-von-dieser-name,
                                   nachname-von-erster-name)
      ELSE  kommt-wort1-vor-wort2?(vorname-von-dieser-name,
                                   vorname-von-erster-name)
   ENDIF.

Trage-namen-in-neue-liste-ein :

   IF neue-liste-noch-leer
      THEN  trage-ein
      ELSE  pruefe-und-trage-ein
   ENDIF.

Trage-ein :

   haenge-erster-name-an-neue-liste-an.

Pruefe-und-trage-ein :

   IF erster-name <> letzter-eintrag-in-neuer-liste
      THEN  haenge-erster-name-an-neue-liste-an.
```

**Cont'd.**

```
Name des Algorithmus :  Kommt-wort1-vor-wort2?
   Eingabe :  wort1, wort2
   Ausgabe :  {TRUE, FALSE}

   IF wort1-vor-wort2
      THEN  TRUE
      ELSE  FALSE
   ENDIF.
```

**Bemerkung**

Der vorgeschlagene Algorithmus lässt sich natürlich noch weiter verbessern. So könnte das Verfahren zur Austausch der Elemente in der alten Liste um eine Fallunterscheidung erweitert werden. Folgende Fälle werden dann behandelt :

- ‚**dieser-name**' kommt alphabetisch NACH ‚**erster-name**' :
  keine Aktion (wie bisher)
- ‚**dieser-name**' kommt alphabetisch VOR ‚**erster-name**' :
  ```
  loesche-name-aus-der-alten-liste;
  schreibe-an-diese-stelle-erster-name;
  erster-name :=  dieser-name
  ```
- ‚**dieser-name**' IST GLEICH ‚**erster-name**' :
  ```
  loesche-name-aus-der-alten-liste
  ```

`Hole-alphabetisch-ersten-namen-aus-der-alten-liste` würde jedoch mehr leisten, als ursprünglich mit der Namensgebung intendiert !

## Elemente imperativer Programmiersprachen

**Allgemeine Ausdruckmittel**

1. **Primitiva (elementare Ausdrücke)**
   - Grundoperationen (Zuweisungen)
   - Elementare Datenobjekte (Variablen)
2. **Mittel zur Kombination**
   - Konstruktion zusammengesetzter Ausdrücke
   - Bedingte Ausführung und Wiederholungen
   - Bildung komplexer(er) Operationen durch Prozeduren / Funktionen (und deren Anwendung)
3. **Mittel zur Abstraktion**
   - Erzeugung abstrakter Objekte aus zusammengesetzten Ausdrücken
   - Benennung von Objekten (Daten, Prozeduren / Funktionen)

Der Prozedurbegriff steht im Zentrum der imperativen Programmierung

## Strukturierter Entwurf und Programmierung

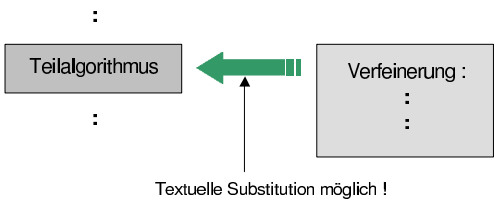
(→ „Top-down“ Verfeinerung und funktionelle Abstraktion)

**2 Kontruktionsmittel :**

- **schrittweise Verfeinerung**
- **Definition neuer (abstrakter) Algorithmen** – die an mehreren Stellen benutzt werden können

**1. Verfeinerung (~ schwache Abstraktion)**

Benennung eines Teilalgorithmus, der genau einmal ausgewendet wird




**2. Neuer (abstrakter) Algorithmus ... Prozedur (~ starke Abstraktion)**

→ Beispielsweise 2 ähnliche Teilagorithmen (~ 2 Verfeinerungen), die sich systematisch nur an wenigen Stellen unterscheiden (und zwar insbesondere durch die Objekte, auf denen der Algorithmus operieren soll)

⇒ **Abstraktionsschritt :**
Abstrahierung von der konkreten Anwendung → **Prozedur**
systematische **Parametrisierung** an unterschiedlichen Stellen, z.B.

| Konkrete Werte (Eingabeparameter) | „Platzhalter“ (formale Parameter) |
|---|---|
| `nachname-von-dieser-name` | `wort1` |
| `vorname-von-dieser-name` | |
| `nachname-von-erster-name` | `wort2` |
| `vorname-von-erster-name` | |

**Vereinbarung einer Prozedur**

- **Kopf :** . Name
  . Angabe- seiner formalen Parameter
  . - Ergebnistyp
- **Rumpf :** Programmtext (→ Definition der Prozedur)

**Vereinbarung einer Prozedur**

- **Kopf :** . Name
  . Angabe- seiner formalen Parameter
  . - Ergebnistyp
- **Rumpf :** Programmtext (→ Definition der Prozedur)

**Hinweis**: In einigen Programmiersprachen werden Prozeduren mit und ohne Resultat(styp) explizit unterschieden, z.B.

in Pascal : PROCEDURE(...)
FUNCTION(...): Resultats-Typ

**jedoch** in Modula-2 :
PROCEDURE(...)[: Resultats-Typ]

in C :
<typ> function(...) (Prozedur ohne Resultats-Typ : void !)

**Vereinbarung einer Prozedur**

- **Kopf :** . Name
  . Angabe- seiner formalen Parameter
  . - Ergebnistyp
- **Rumpf :** Programmtext (→ Definition der Prozedur)

**Hinweis**: In einigen Programmiersprachen werden Prozeduren mit und ohne Resultat(styp) explizit unterschieden, z.B.
in Pascal : PROCEDURE(...)
FUNCTION(...): Resultats-Typ
**jedoch** in Modula-2 :
PROCEDURE(...)[: Resultats-Typ]
in C :
<typ> function(...) (Prozedur ohne Resultats-Typ : void !)

**Aufruf :**

<name>(Liste der aktuellen Parameter)

**Strukturierungsmittel in Programmiersprachen**

Gängige Programmiersprachen bieten keine Unterscheidung zwischen schwacher und starker Abstraktion !

→ **Für die schwache Abstraktion werden ebenfalls Prozeduren verwendet !**

„You should not hesitate, however, from formulating an action as a procedure – even when called only once – if done so enhances the readability of a program."

(aus K. Jensen, N. Wirth. Pascal user manual and report, 3rd edition. Springer, Berlin, 1985, p.106)

zusätzlich in Pascal, Modula 2, etc. :

Möglichkeit der Definition lokaler Prozeduren (/ Funktionen) → Blockstruktur

- Der **Entwurf** entwickelt ein Modell des Gesamtsystems, wobei die Aufgabe beschrieben (**Anforderungsdefinition**) und die Lösung des Problems **spezifiziert** wird
- Die Strategie beim Softwareentwurf beruht auf der Zerlegung eines komplexen Systems in **Moduln** und der Beschreibung der **Schnittstellen**, ohne Vorwegnahme von Implementierungsdetails
- Die **Spezifikation** muss eine **vollständige** Problemlösung mit deren Merkmalen definieren, alle **notwendigen Operationen** festlegen und dabei **eindeutig** sein
- Die **Modularisierung** spiegelt die Entwurfseintscheidungen wider; Kriterien sind **Unabhängigkeit**, **Verständlichkeit** und **Änderbarkeit**
- Bei der **schrittweisen Verfeinerung** werden nacheinander elementare Operationen definiert und dann hierarchisch weiter detailliert; **Prozeduren** bilden elementare Strukturierungsmittel, die durch Angabe von **Parametern** für verschiedene Berechnungen verwendet werden können

**Literatur-Hinweise zum Inhalt:**

D. Bell, I. Morrey, J. Pugh. The essence of program design. London, Prentice-Hall, 1997.

E. Hering. Software Engineering, 3. Aufl. Braunschweig, Friedr. Vieweg & Sohn, 1992.

R. Kimm, W. Koch, W. Simonsmeier, F. Tonsch. Einführung in Software Engineering. Berlin, Walter de Gruyter, 1979.

G. Pomberger, G. Blaschek. Software Engineering – Prototyping und objektorientierte Software-Entwicklung. München, Carl Hanser Verlag, 1993.

Schöne
Weihnachten

und

Einen guten Rutsch
ins neue Jahr ... !

*Klaus Mumm, Heiko Neumann,*
*Friedhelm Schwenker, Stefan Geschwentner und die TutorInnen*