12. Konzepte der Computerprogrammierung

- Algorithmen und Beschreibungsformen
- Strukturierung von Programmabläufen
- Typen
- Abstraktionsmechanismen
- Anforderungen an eine maschinennahe Programmiersprache

12.1 Algorithmen und Ablaufstrukturierung

12.1.1 Intuitive Einführung

- Algorithmen sind detaillierte Vorschriften zur schrittweisen Lösung von Problemen.

- Algorithmen sind allgemeine Lösungsverfahren und nicht an eine bestimmte Ausführungsform gebunden (z.B. eine bestimmte Programmiersprache).

- Algorithmen bestehen aus
  - einer Abfolge von Arbeitsschritten,
  - Ein- und Ausgaben von Daten,
  - bedingten Verzweigungen,
  - Schleifen,
  - Unteralgorithmen.

- Jeder Schritt eines Algorithmus basiert auf elementaren Handlungen.
  Elementare Handlungen zeichnen sich dadurch aus, das sie wohldefiniert (zweifelsfrei) sind und keiner weiteren Erläuterung bedürfen.

Beispiel: Textuelle Beschreibung eines Algorithmus

Beispiel: Finden des "größten gemeinsamen Teilers" (GGT) zweier Zahlen

Gegeben seien zwei Zahlen x und y.

Wenn x größer y ist, **dann** subtrahiere von x den Wert y,
wenn nicht, dann subtrahiere von y den Wert x.

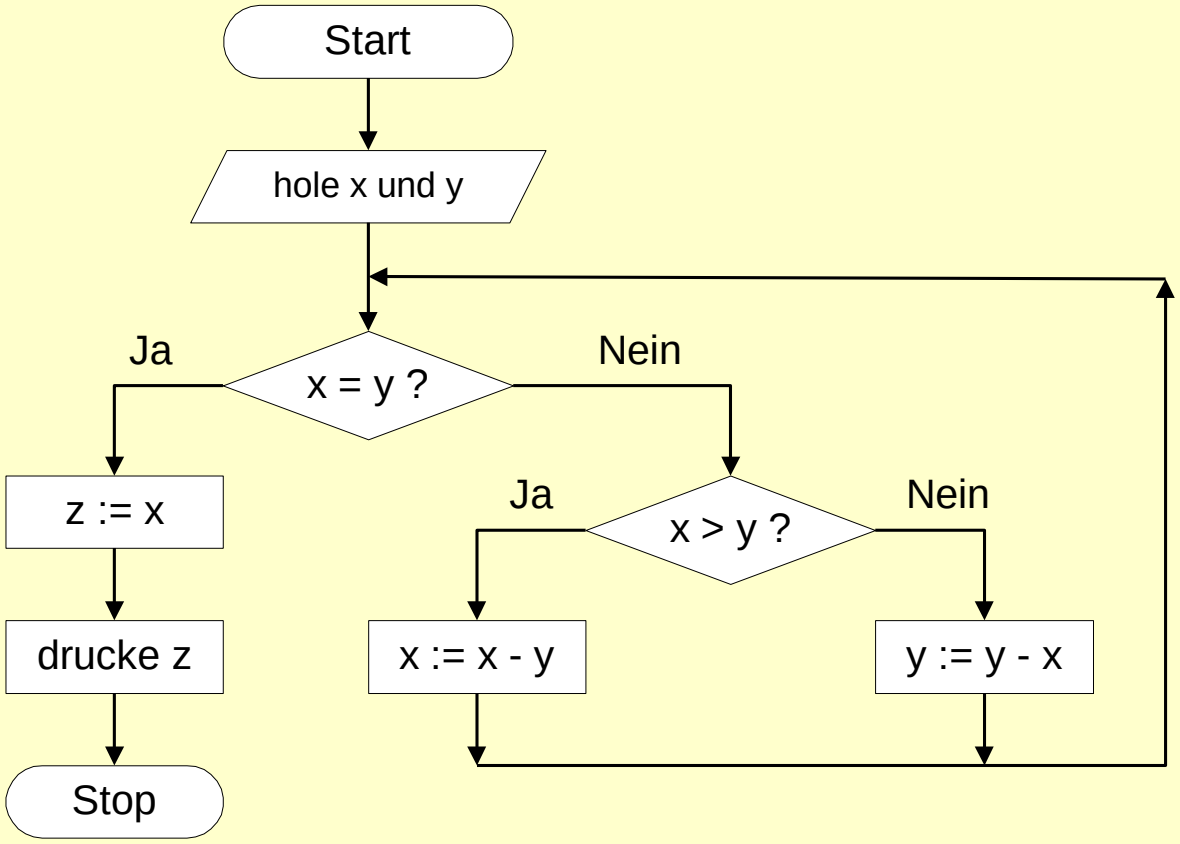
Wenn x und y nicht gleich sind, **dann wiederhole** den letzen Schritt, **ansonsten** drucke das Ergebnis aus.

z.B.: x = 27, y = 15

```
27 - 15 = 12    // x:=x-y (=12)
15 - 12 = 3     // y:=y-x (= 3)
12 -  3 = 9     // x:=x-y (= 9)
 9 -  3 = 6     // x:=x-y (= 6)
 6 -  3 = 3     =>   GGT = 3
```

12.1.2 Darstellungsform: Flussdiagramm

Beispiel: Finden des “größten gemeinsamen Teilers” (GGT) zweier Zahlen



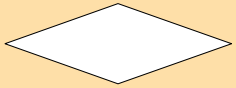
- älteste aller Darstellungsmethoden (für 1. Sprach-Generation: Assembler)
- basiert auf nur 5 Symbolen
- **Assembler-geeignet (Darstellung unstrukturierter Algorithmen**)
- genormt (DIN66001 u. ISO-Norm 5807)

Symbole des Flussdiagramms

Programmstart oder -ende

Aktion

Ein- und Ausgabe



Bedingung

Fortsetzung an anderer Stelle

Flussdiagramm – Vor- und Nachteile

Vorteile:

- sehr intuitiv und einfach erlernbar
- genormte Symbole
- kann leicht ergänzt werden

Nachteile:

- verleitet zu unstrukturiertem Denken (“Spaghetti-*Programmierung*”) → s. Beispiel

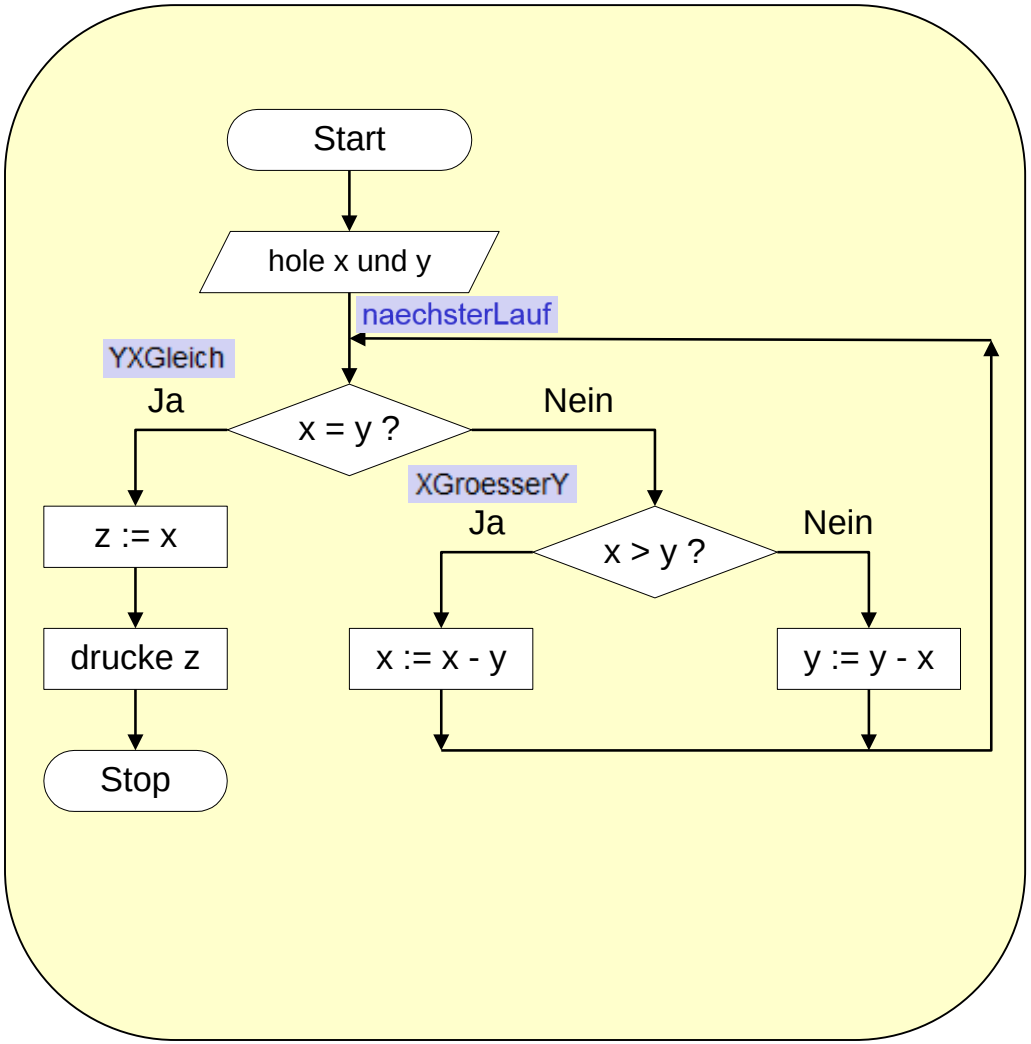
Bewertung:

- Die Urform des Flußdiagramms ist “*veraltet*” aber
- eine stark erweiterte Variante (*Aktivitätsdiagramm*) ist Teil der UML 2.

Anmerkung:

- Das Flußdiagramm ist eine häufig verwendete Beschreibungsform für Assemblerprogramme (z.B. Dokumentation von Altcode).
  Neue Assemblerprojekte sollten immer strukturiert geplant werden (s.u.).

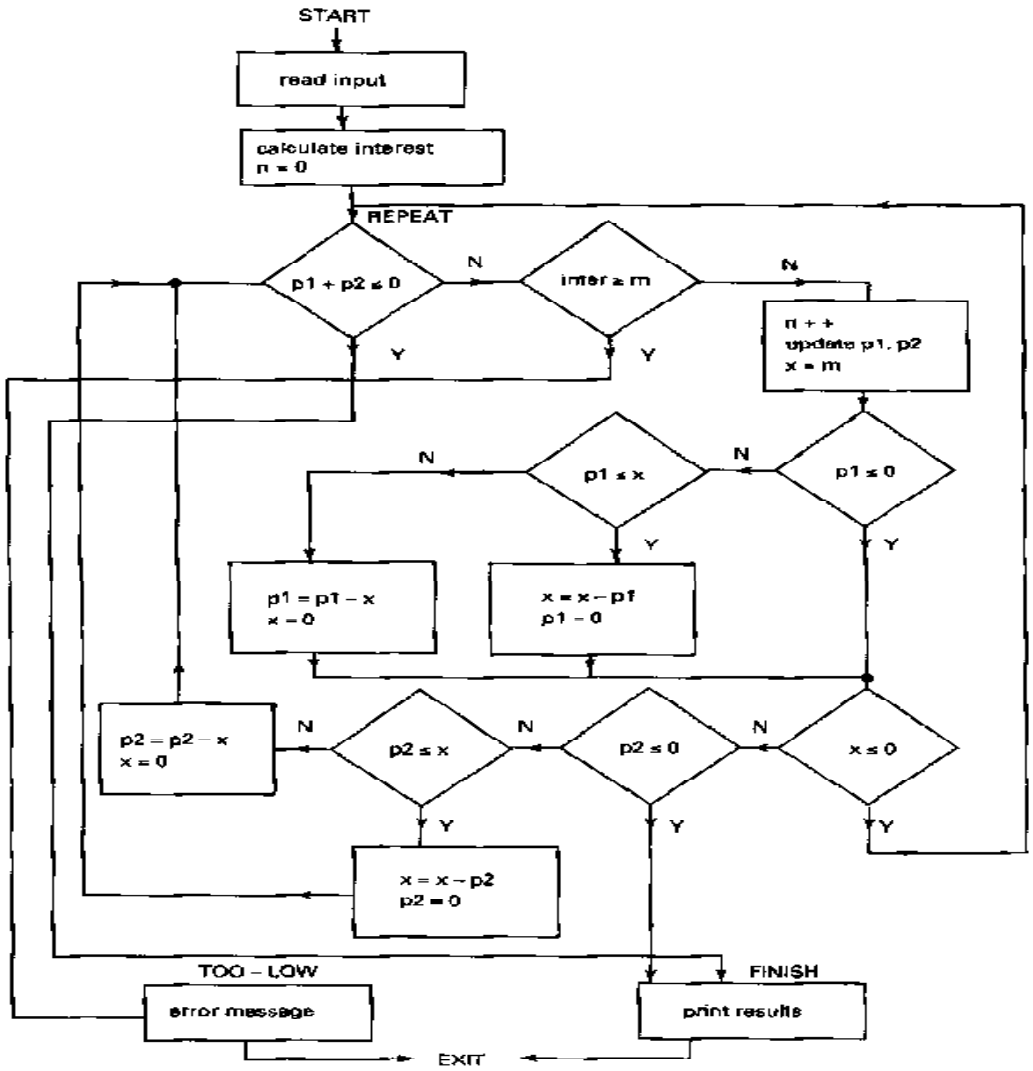
Beispiel: Flussdiagramm und Realisierung mit unstrukturierter Sprache



Beispiel einer unstrukturierten Sprache

```
PROGRAM GGT(x,y)
LABEL  naechsterLauf
IF x=y THEN GOTO  YXGleich
IF x>y THEN GOTO  XGroesserY
y=y-x
GOTO  naechsterLauf
LABEL  XGroesserY
x=x-y
GOTO  naechsterLauf
LABEL  XYGleich
z=x
PRINT(z)
END PROGRAM
```

Beispiel: Spaghetti-Programm (GoTo-Programmierung)



Selbst einfache Zusammenhänge werden sehr schnell unübersichtlich !

12.1.3 Strukturierte Programmierung

Idee entstand Ende der 60'er Jahre als Folge der sog. "*1. Softwarekrise*"*:*
Problem: → *"unentwirrbare Programmabläufe" (Spaghetticode)*

Untersuchungen von *Nassi-Shneiderman* haben gezeigt:

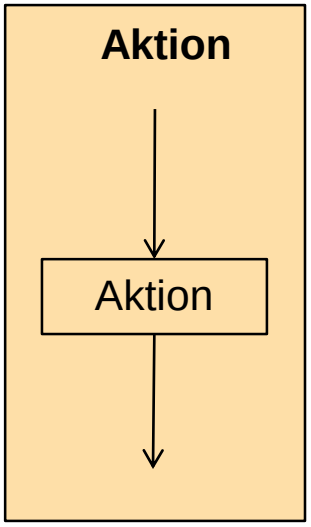
Anwendungen sind (goto-frei) durch nur 7 Strukturblockarten darstellbar !

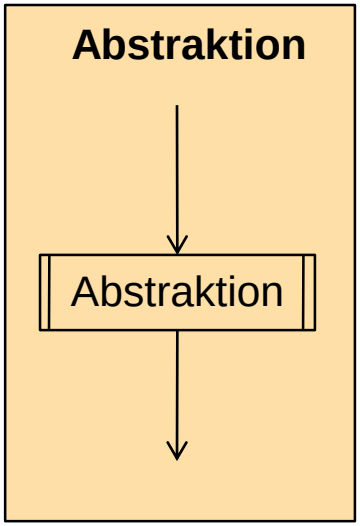
Strukturblöcke sind aus Kombinationen von Bedingungen und Aktionen darstellbar !

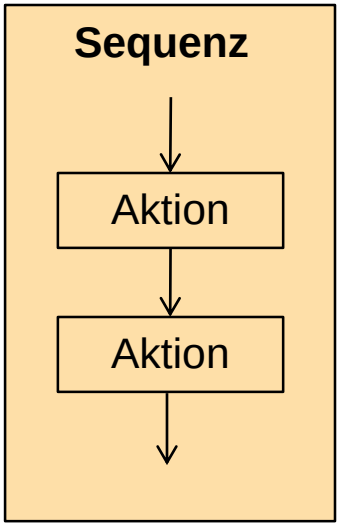
⇒ **Strukturierte Programmierung**

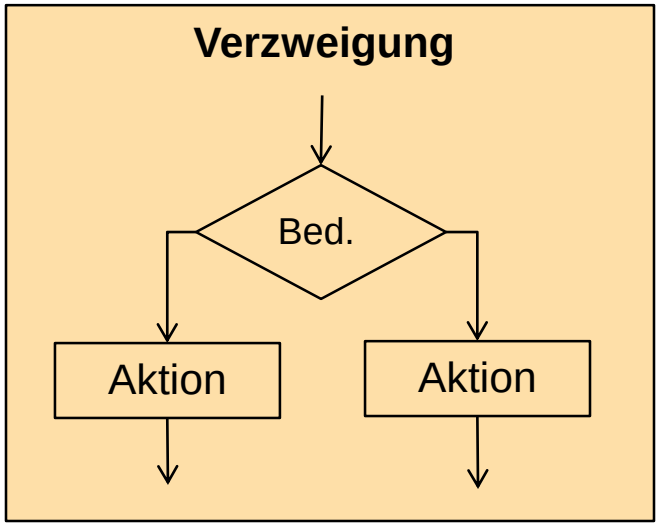
Fazit: Unstrukturierte Entwürfe können immer in strukturierte Entwürfe umgeformt werden !

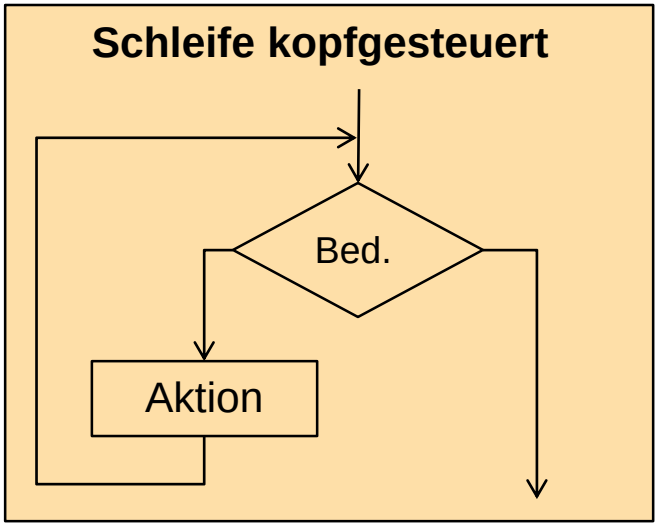
Strukturblockarten nach Nassi-Shneiderman

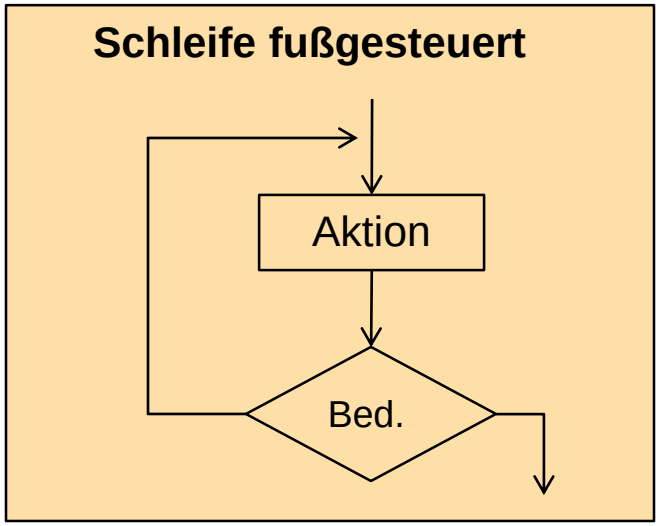










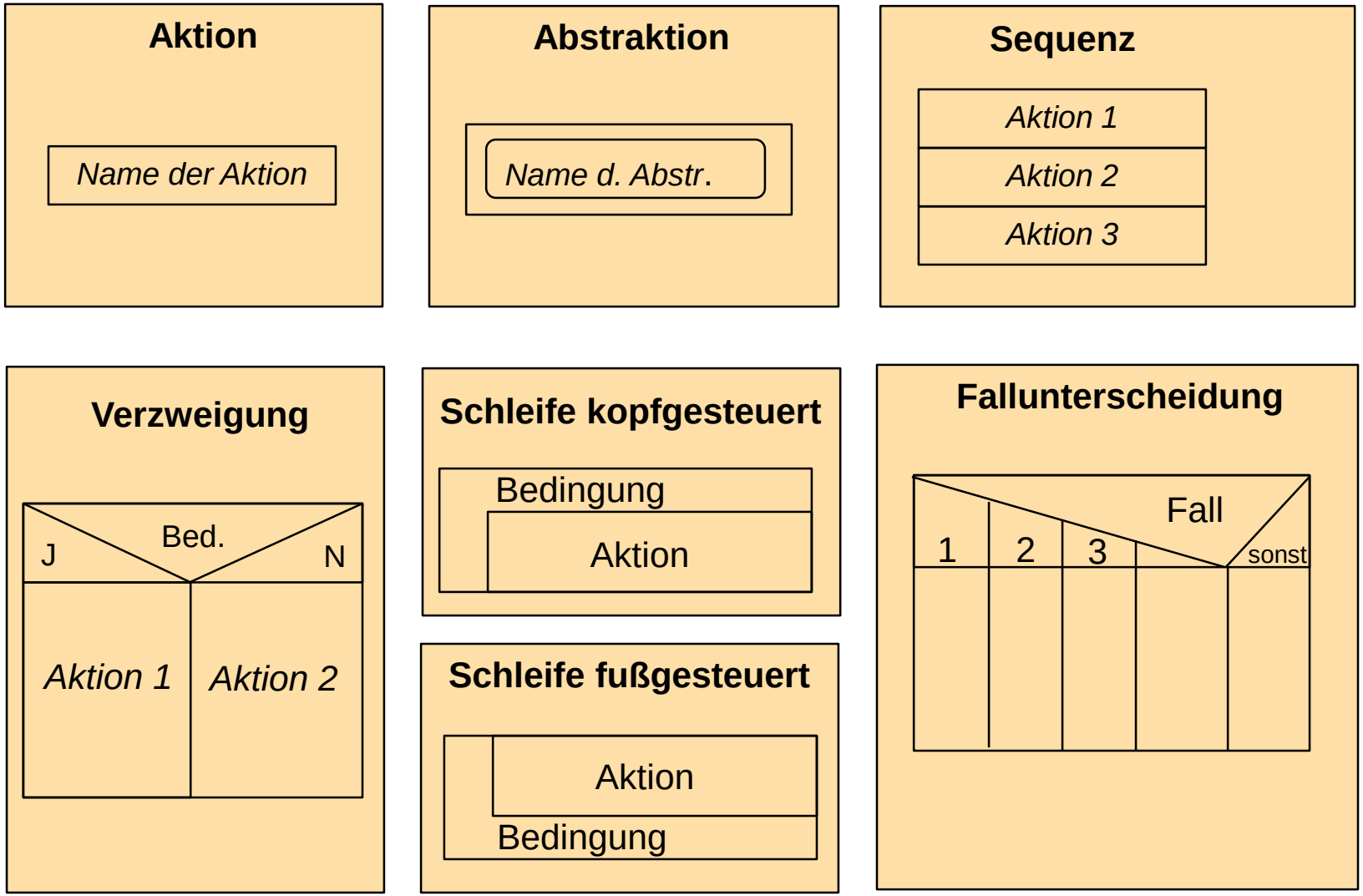


+

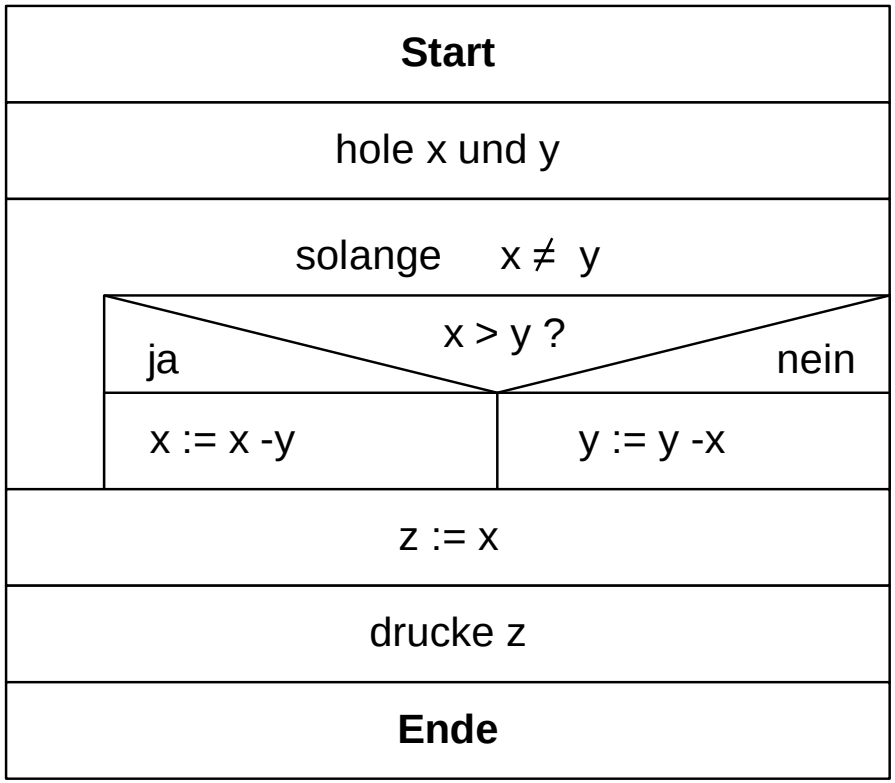
mehrfache Verzweigung

(= Fallunter-scheidung)

Struktogramm nach Nassi-Shneiderman



Beispiel: GGT-Algorithmus als Struktogramm



Strukturblöcke können nur

gestapelt oder
geschachtelt

werden !

Unstrukturierte Algorithmen-entwürfe müssen entsprechend umgebaut werden !

Struktogramm – Vor- und Nachteile

Vorteile:

- grundsätzlich strukturiert (goto-frei)
  → führt zu wesentlich übersichtlicheren und verständlichere Programmen
- Algorithmenentwurf hierarchisch gliederbar (Top-Down- bzw. Bottom-up-Entwurf)
- genormt (DIN 66261)

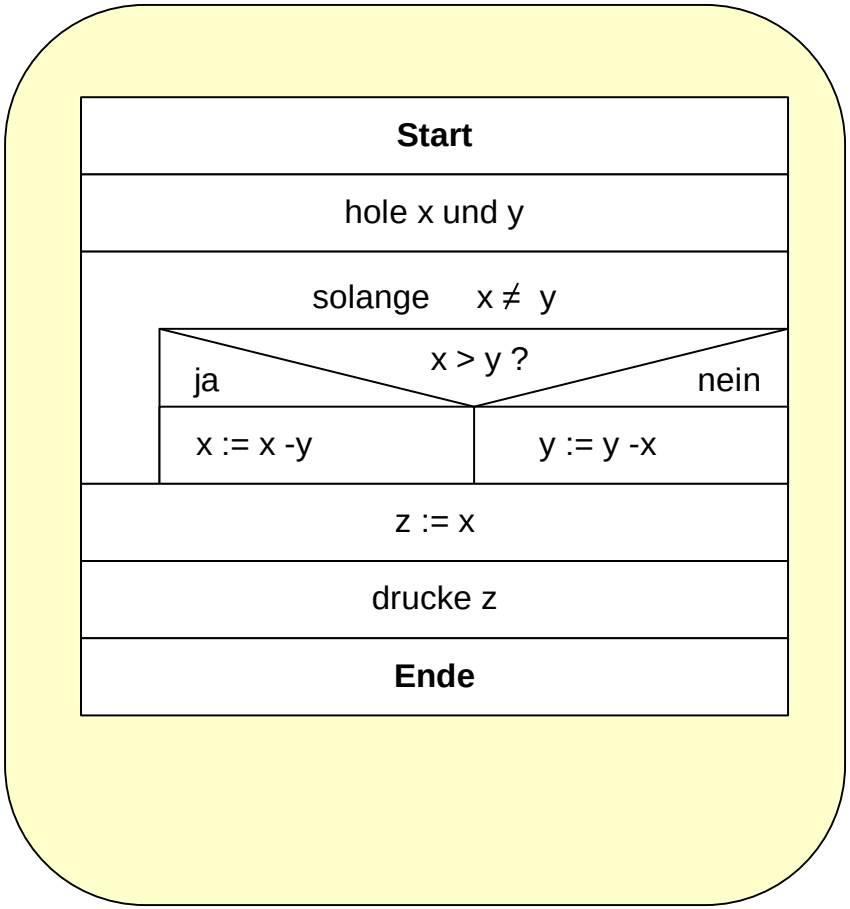
Nachteile:

- mit Papier und Bleistift ist der Entwurf ziemlich umständlich
- nur mit geeigneten Werkzeugen einsetzbar

Bewertung:

- Erfüllt mit geeigneten Werkzeugen durchaus seinen Zweck,
- ist "***veraltet***" und wird kaum noch verwendet,
- Grund: Der im Folgenden beschriebene *Pseudocode* (s.u.) besitzt die gleichen Vorteile wie Struktogramme und vermeidet dessen Nachteile

Beispiel: Struktogramm und Realisierung mit strukturierter Sprache



Beispiel einer strukturierten Sprache

```
PROGRAM GGT(x,y)
WHILE  ( x ungleich y )
        IF (x > y)
                x=x-y
        ELSE
                y=y-x
        ENDIF
ENDWHILE
z=x
PRINT(z)
END PROGRAM
```

Vergleich: Unstrukt. Programmiersprache vs. strukt. Programmiersprache

```
PROGRAM GGT(x,y)
LABEL naechsterLauf
IF x=y THEN GOTO YXGleich
IF x>y THEN GOTO XGroesserY
y=y-x
GOTO  naechsterLauf
LABEL  XGroesserY
x=x-y
GOTO  naechsterLauf
LABEL  XYGleich
z=x
PRINT(z)
END PROGRAM
```

```
PROGRAM GGT(x,y)
WHILE  ( x ungleich y )
        IF (x > y)
                x=x-y
        ELSE
                y=y-x
        ENDIF
ENDWHILE
z=x
PRINT(z)
END PROGRAM
```

12.1.4 Darstellungsform: Pseudocode

Die Algorithmusbeschreibung ist Programmiersprachen-ähnlich.

Die Strukturierungselemente sind angelehnt an die des Struktogramms.
Es gibt einige praktische Erweiterungen.

Strukturierungselemente des Pseudocode *(hier: LaTeX-Style: algoritmicx)*

a) Zuweisung

$$x \leftarrow x + 1$$

b) Kopfgesteuerte Schleife

```
1: while <text> do
2:     <body>
3: end while
```

Beispiel

```
1: sum ← 0
2: i ← 1
3: while i ≤ n do
4:     sum ← sum + i
5:     i ← i + 1
6: end while
```

→ while-Bedingung ist eine Laufbedingung (in der Schleife bleiben, **solange** …. gilt)

c) Kopfgesteuerte Zählschleife (Erweiterung)

```
1: for <text> do
2:     <body>
3: end for
```

Beispiel

```
1: sum ← 0
2: for i ← 1, n do
3:     sum ← sum + i
4: end for
```

Fussgesteuerte Schleife

```
1: repeat
2:     <body>
3: until <text>
```

Beispiel

```
1: sum ← 0
2: i ← 1
3: repeat
4:     sum ← sum + i
5:     i ← i + 1
6: until i > n
```

→ until-Bedingung ist eine Abbruchbedingung (bleibe in der Schleife **bis** …… gilt)

Iteration über eine Elementemenge (Erweiterung)

```
1: for all <text> do
2:     <body>
3: end for
```

Alternative

```
1: if <text> then
2:     <body>
   [
3: else if <text> then
4:     <body>
   ...
   ]
   [
5: else
6:     <body>
   ]
7: end if
```

Beispiel

```
1: if quality ≥ 9 then
2:     a ← perfect
3: else if quality ≥ 7 then
4:     a ← good
5: else if quality ≥ 5 then
6:     a ← medium
7: else if quality ≥ 3 then
8:     a ← bad
9: else
10:    a ← unusable
11: end if
```

Abstraktion durch Prozeduren

```
1: procedure <NAME>(<params>)
2:     <body>
3: end procedure
```

Beispiel

```
1: procedure GGT(x, y)
2:     while x ≠ y do
3:         if x > y then
4:             x ← x − y
5:         else
6:             y ← y − x
7:         end if
8:     end while
9:     return x
10: end procedure
```

Pseudocode – Vor- und Nachteile

Vorteile:

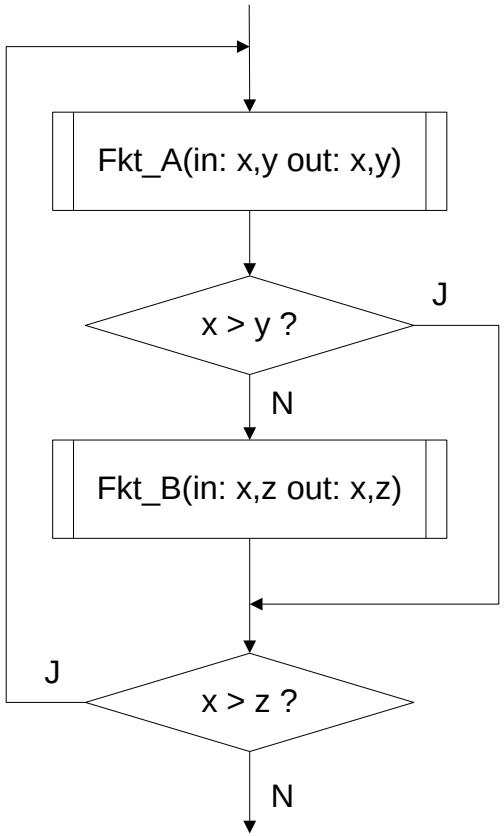
- grundsätzlich strukturiert (goto-frei)
  → führt zu übersichtlicheren und verständlichere Programmen
- Algorithmenentwurf hierarchisch gliederbar (Top-Down- bzw. Bottom-up-Entwurf)
- sprachunabhängig aber trotzdem implementierungsnah
- leicht als Kommentartext in Programme integrierbar
- leicht zu erstellen und zu ändern

Nachteile:

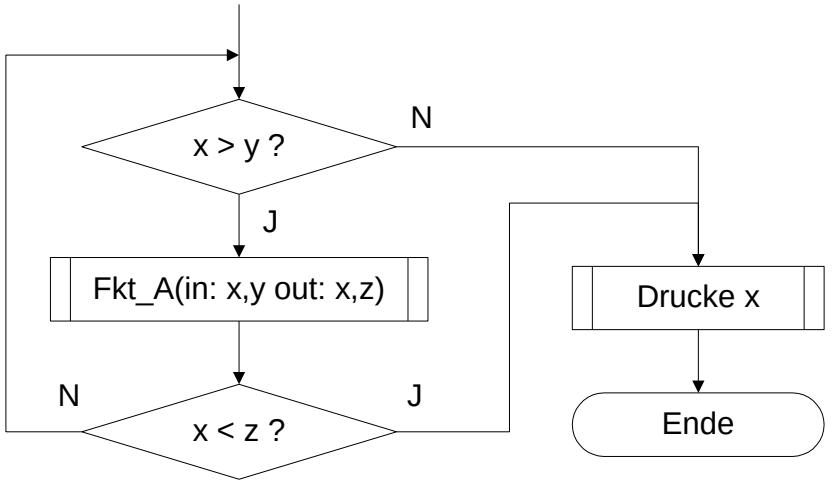
- Gefahr der zu unpräzisen Algorithmusbeschreibung

ÜBUNG: Umwandeln Flussdiagramm -> Pseudocode

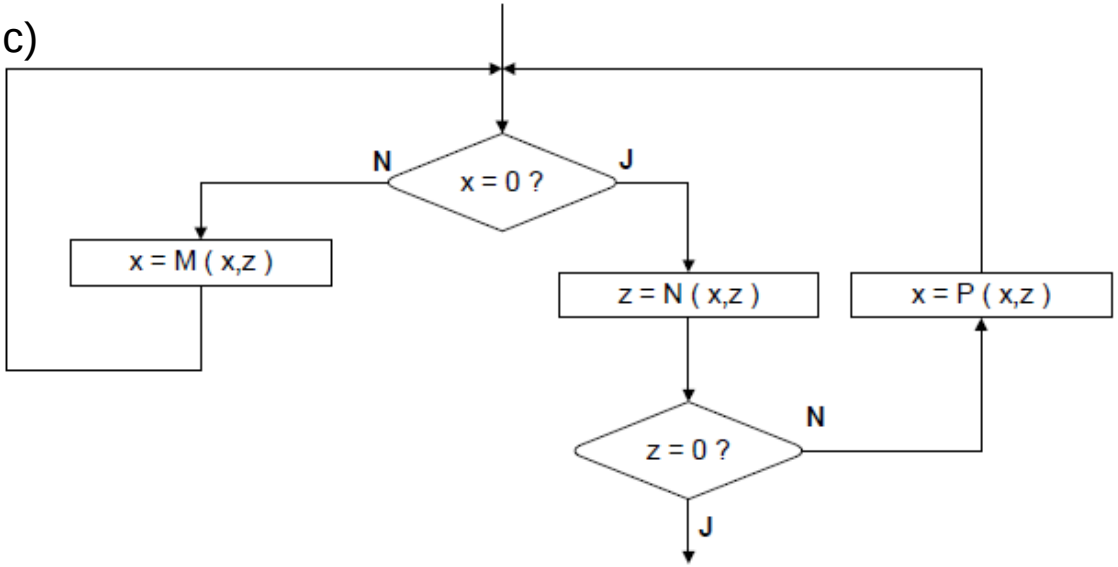
a)



b)



c)



12.1.5 Konsequenzen für Programmiersprachen

- Die meisten **Assemblersprachen** stellen für die Ablaufgestaltung lediglich bedingte Sprünge zur Verfügung (*Branch if zero, Branch if not carry, u.ä.*).
  Strukturierung wird daher nicht erzwungen sondern kann **nur durch Programmier-disziplin** (per Konvention) erreicht werden.

  Fazit: Assembler besitzt i.a. keine Sprachkonstrukte zur Ablaufstrukturierung

- Alle **höheren** prozeduralen und viele objektorientiere **Programmiersprachen** stellen Sprachkonstrukte für die Ablaufstrukturierung im Sinne der Nassi-Shneiderman-Strukturblockarten zur Verfügung, z.B. :

  - **if** (*Bedingung*) **then** {*Programmsequenz*} **else** {*Alternativsequenz*} **endif**
  - **while** (*Bedingung*) **do** {*Programmsequenz*} **endwhile**
  - **repeat** {*Programmsequenz*} **until** (*Bedingung*) **endrepeat**

12.2 Typen

- **Typen** klassifizieren Daten aufgrund ihrer Eigenschaften und ihrer Verwendungs-möglichkeiten.

- **Typen** sollen die unbeabsichtigte oder falsche Interpretation oder Verwendung von Daten verhindern.

- Eine **Typüberprüfung** stellt die Kompatibilität zwischen Operator und seinen Operanden sicher.

- Programmiersprachen in denen es Typen gibt, heißen „***typisiert***“.

- Eine Programmiersprache heißt „***statisch typisiert***“, wenn die Typüberprüfung zur Compilezeit überprüft wird (z.B. C).

- Eine Programmiersprache heißt „***stark typisiert***“, wenn der Compiler die Typ-konsistenz garantiert (*strong typing*).

  Beispiel : Assembler → nicht typisiert

12.3 Abstraktionsmechanismen

12.3.1 Prozedurale Programmiersprachen → ablauforientiert

Abstraktion hilft Komplexität zu beherrschen.

Ein wichtiges Abstraktionsmittel sind *Prozeduren*. Eine Prozedur ist eine Softwareeinheit mit einer Ein- und Ausgabeschnittstelle sowie einem beschreibbaren Verhalten. Das Wesen der Prozedur ist, das das der Anwender nur das Schnittstellenverhalten kennen muß. Die Implementierungsdetails sind für die Verwendung unwesentlich.

Programmiersprachen, die hauptsächlich durch die Untergliederung in verschiedene Prozeduren strukturiert sind, heißen *prozedurale Programmiersprachen*.

In prozeduralen Programmiersprachen ist die Verarbeitung strukturiert, aber nicht zwangsläufig die Daten !

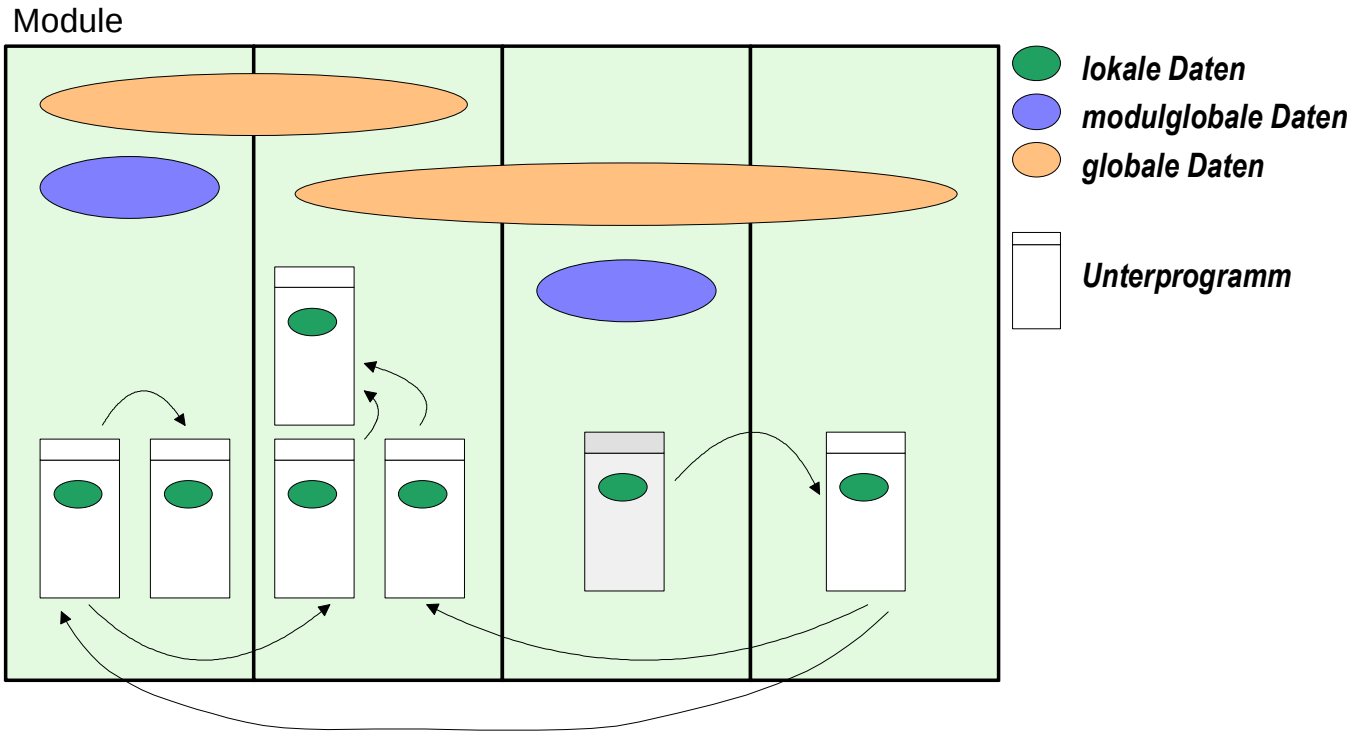
In statisch typisierten Sprachen wird die Typkorrektheit der Ein- und Ausgabewerte der Prozeduren zur Complilezeit überprüft.

Bsp.: Assemblersprachen stellen i.allg. Mechanismen für die Bildung von Prozeduren zur Verfügung (bl, bx).
Eine Typüberprüfung der Ein- und Ausgabewerte findet aber nicht statt.

Grenzen prozeduraler Programmiersprachen

In prozeduralen Programmen gibt es keine feste Kopplung zwischen den Prozeduren und Daten.

Änderungen/Erweiterungen an den Datenstrukturen oder Prozeduren führen daher häufig zu schwer überblickbaren Änderungen am gesamten Softwaresystem.

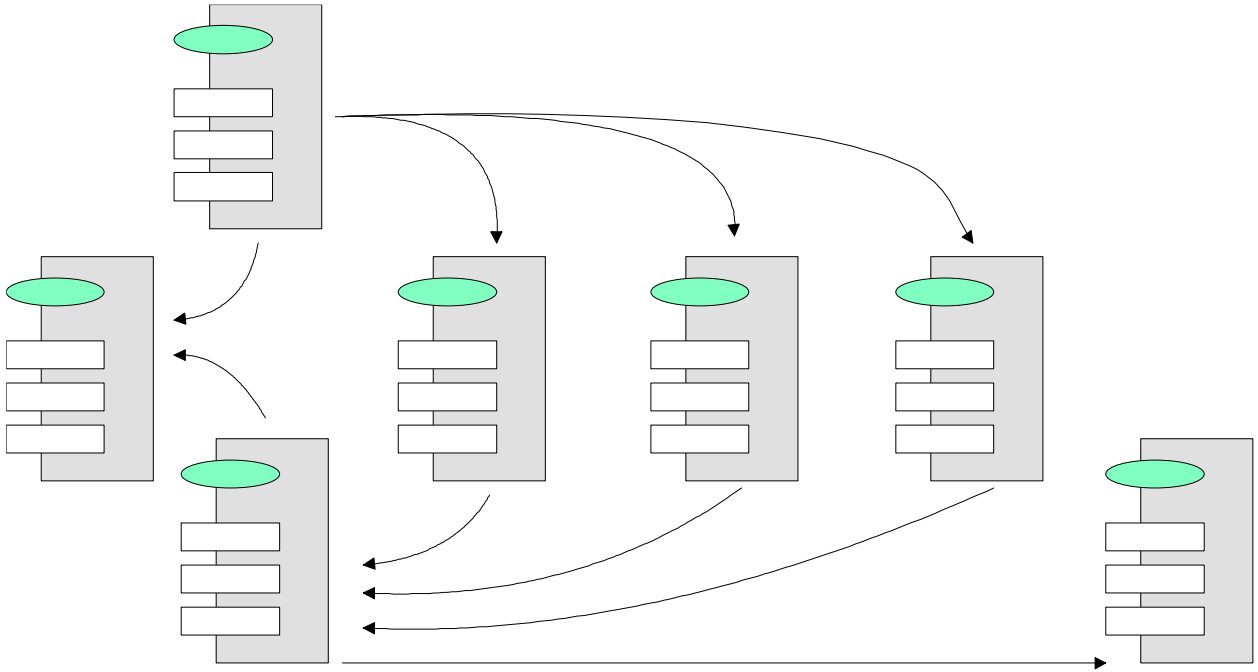


In Zeiten zunehmender Softwarekomplexität (Anfang der 90er Jahre) war dieser Sachverhalt eine zunehmendes Implementierungshindernis.

→„*Wiederverwendungskrise*“ der Softwareentwicklung

12.3.2 Objektorientierte Programmiersprachen → interagierende Objekte

Objektorientierte Programmiersprachen vermeiden das geschilderte Problem u.a. durch feste Kopplung der Daten und der mit ihnen arbeitenden Prozeduren.



Ziele objektorientierter Programmiersprachen sind also eine verbesserte Änderbarkeit, Wartbarkeit und Wiederverwendbarkeit von Softwareteilen.

12.4 Mindestanforderungen an höhere, maschinennahe Sprachen

effizient

Gutes Laufzeitverhalten, wenig Overhead.

typisiert

Operationen und Unterprogrammaufrufe werden zur Compilezeit daraufhin überprüft, ob die Parameter den vereinbarten Typ haben.

strukturiert

Es stehen Sprachkonstrukte im Sinne der Nassi-Shneiderman-Strukturblockarten zur Verfügung.

prozedural

Das Programm kann in benennbare Funktionsblöcke zerlegt werden. Von einer einmal geschriebenen Funktion muss nur noch das Schnittstellenverhalten bekannt sein.

modular

Der Quellcode kann in sinnvolle Einheiten unterteilt werden, z.B. mit dem Ziel der Wiederverwertbarkeit von Codeteilen. → Bibliotheken
Beispiele: Mathematische Funktionen, Grafikfunktionen, …..

13. Programmiersprache C

13.1 Historie

- C wurde 1972 von Dennis Ritchie bei den AT&T Bell Lab als Systemprogrammier-sprache zur Implementierung von UNIX für die PDP-11 entwickelt.

- Ziel von C war die Entwicklung einer Hochsprache für lesbare und portable Systemprogramme, aber einfach genug, um auf die zugrunde liegende Maschine abgebildet zu werden.

- In 1973/74 wurde C von Brian Kernighan verbessert. Daraufhin wurden viele Unix-Implementierungen von Assembler nach C umgeschrieben.

- Um die Vielzahl der entwickelten Compiler auf einen definierten Sprachumfang festzulegen, wurde C 1983 durch die amerikanische Normbehörde ANSI normiert. Diese Sprachnorm wird als ANSI-C bezeichnet. Die aktuelle Norm stammt aus dem Jahr 2011 (ANSI = **A**merican **N**ational **S**tandardisation **I**nstitute).

- Erweiterung für objektorientierte Programmierung (Objekte, Klassen, Vererbung): Sprache „C++“ (oft nicht unterschieden/gemeinsam betrachtet: „C/C++“)

13.2 Einführende Anmerkungen

13.2.1 Stärken und Schwächen

Stärken:

- standardisiert (ANSI)
- sehr effizient (hardwarenah implementiert)
- universell verwendbar
- sehr weit verbreitet, speziell in technischen Anwendungen
- Typüberprüfung (weitgehend strong typing)

Schwächen:

- Code kann beliebig unlesbar geschrieben werden, da Fragen des Programmierstils nur in (freiwilligen) Konventionen festgelegt sind
  --> Erfahrung und persönlicher Stil haben entscheidenden Einfluss auf die Softwarequalität !
- teilweise etwas kryptische Notation

13.2.2 Die größte Gefahr: Stilloser Code

```
/* ASCII to Morsecode                                          */
/* Obfuscated C Code Contest :   „Best“ small Program */

#include<stdio.h>
#include<string.h>

main()
{
        char*O,l[999]="'`acgo\177~|xp .-\0R^8)NJ6%K4O+A2M(*0ID57$3G1FBL";
        while(O=fgets(l+45,954,stdin)){
                *l=O[strlen(O)[O-1]=0,strspn(O,l+11)];
                while(*O)switch((*l&&isalnum(*O))-!*l){
                case-1:{char*I=(O+=strspn(O,l+12)+1)-2,O=34;
                        while(*I&3&&(O=(O-16<<1)+*I---'-')<80);
                        putchar(O&93?*I&8||!(I=memchr(l,O,44))?'?':I-l+47:32);
                        break;
                case 1:     ;}*l=(*O&31)[l-15+(*O>61)*32];
                        while(putchar(45+*l%2),(*l=*l+32>>1)>35);
                case 0:     putchar((++O,32));}
        putchar(10);}
}
```

13.2.3 Eigenschaften

C ist .....

klein

- 32 Schlüsselwörter und
- 40 Operatoren

modular

- alle Erweiterungen stecken in Funktionsbibliotheken
- unterstützt das Modulkonzept

maschinennah

- geht mit den gleichen Objekten um wie die Hardware:
  Zeichen,
  Zahlen,
  Adressen,
  Speicherblöcke.

13.3 Ein erster Blick auf (einfache) C-Programme

13.3.1 Beispiel eines einfachen C-Programms

```
#include <stdio.h>

int main( )
{
    printf("Hello World\n");
    return 0;
}
```

Eine **main-Funktion** wird immer benötigt, damit der Compiler den Beginn des Hauptprogramms erkennt. Das Programm steht zwischen { ... }.

Der Rückgabewert der Funktion ist "**int**„ (ganze Zahl). In diesem Programm ist der Rückgabewert der **main()** - Funktion 0:

return 0;

Dies bedeutet, dass das Programm fehlerfrei beendet wurde.

13.3.2 Komponenten eines C-Programms

Präprozessor

`#include <stdio.h>` ist kein direkter Bestandteil der Sprache C, sondern ein Befehl des sogenannten **Präprozessors**. Er führt vor dem eigentlichen Übersetzungsvorgang Textersetzungen durch und erlaubt die Steuerung des Übersetzungsvorganges. Präprozessorbefehle erkennt man am # in der ersten Spalte.

`#include` kopiert vor der Übersetzung den Text der Datei `<stdio.h>` vor den Programmtext.

Ergebnisausgabe

Mit `printf("Hello World\n")` wird der in " " stehende Text ausgegeben. `"\n"` ist eine sog. Escape-Sequenz und bedeutet „Zeilenumbruch".

Zeilenende ";"

Alle Kommandos werden mit einem ";" abgeschlossen.

Kommentare /* ..... */

Kommentare sind in `/* Kommentar ... */` eingeschlossen.

13.3.3 Einfache Datentypen und formatierte Terminalausgabe

```
#include <stdio.h>
int main ()  /* Beginn des Hauptprogramms */
{
      int       i = 10;
      double    db = 1.23;
      printf("Zahl=%d",i);
      printf("\n");
      printf("%10.2lf",db);
      return 0;
}
```

Variablendeklaration

Alle verwendeten Variablen müssen deklariert werden, d.h. der Datentyp und Name der Variablen werden bekannt gemacht.

`double` deklariert z.B. eine reelle Zahl. Das Dezimaltrennzeichen ist der ".".

Formatierte Ergebnisausgabe

Ergebnisse können durch Formatbeschreiber (z.B. %d, %2d, %8.3lf) im Formatbeschreibungs-String formatiert ausgegeben werden.

%2d bedeutet, eine Integerzahl wird in einem Feld von 2 Zeichen ausgegeben.
%8.3lf bedeutet, eine double-Zahl wird in einem Feld von 8 Zeichen (Vor- und Nachkommazahlen und das Komma selbst), mit 3 Nachkommastellen ausgegeben.

13.3.4 Tastatureingabe

```
int main ()
{
      int i;
      printf("Bitte geben Sie eine Zahl ein : ");
      scanf("%d",&i);       /*Wartet auf Eingabe*/
      printf("Die Zahl die Sie eingegeben haben war %d\n",i);
      return 0;
}
```

Mit scanf() können Texte von der Tastatur eingelesen werden.
Der Formatbeschreiber (hier "%d") gibt den Typ der eingelesenen Werte an.

Die Variable muß den Adressoperator & vorangestellt haben.

13.4 Sprachelemente von C

13.4.1 Übersicht

C besitzt 6 **Wortklassen**:

- Bezeichner
- reservierte Worte
- Konstanten
  - Ganzzahlkonstanten
  - Gleitkommakonstanten
  - Zeichenkonstanten
- Strings
- Operatoren
- Trenner (Leerstellen, Zeilentrenner, Kommentare)