

KV Software Engineering
Prof. Dr. Martin Glinz

Kapitel 2

Systematisches Programmieren



Universität Zürich
Institut für Informatik

2.1 Einführung

2.2 Namengebung

2.3 Datendefinitionen

2.4 Ablaufkonstrukte

2.5 Unterprogramme

2.6 Optimierung

2.7 Dokumentation

Das Problem

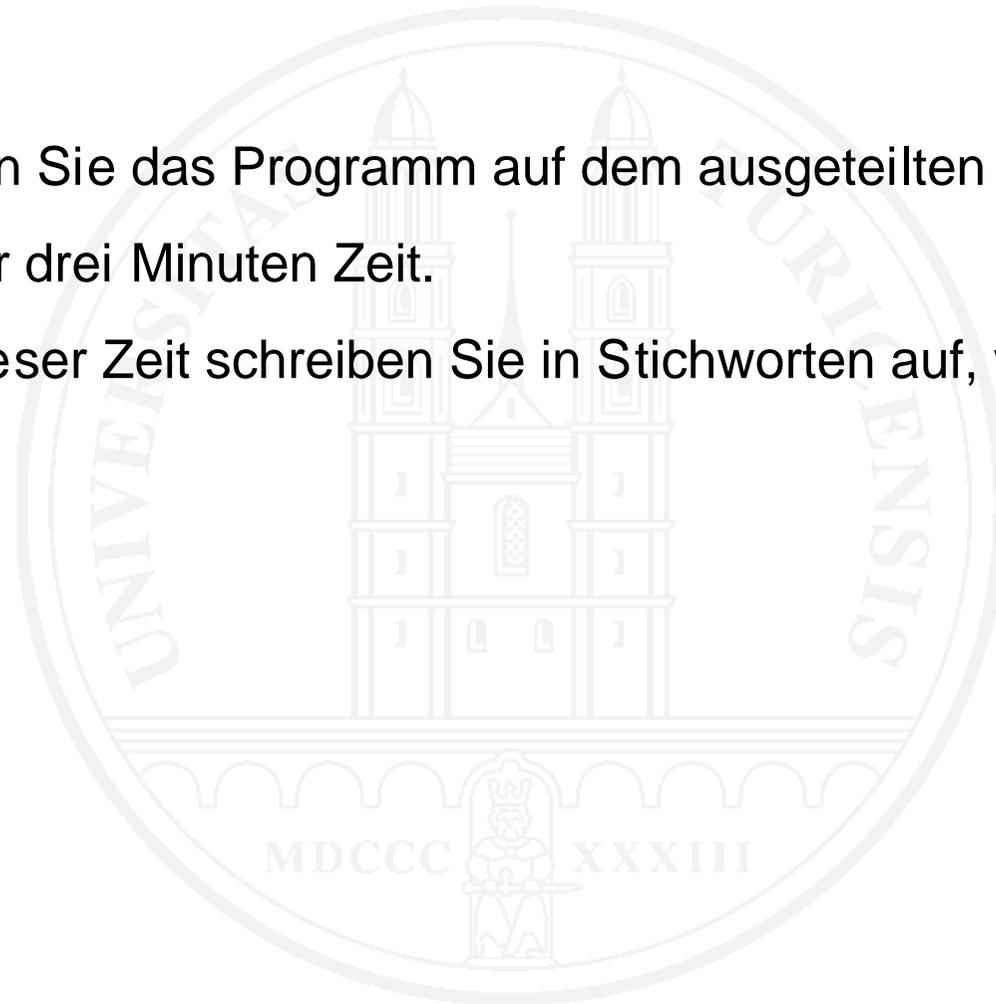
- Programme haben mehr **Leser** als **Schreiber**
 - ⇒ Leichte **Lesbarkeit** ist **wichtiger** als leichte **Schreibbarkeit**
- Programme werden (meistens) nicht von den Leuten gepflegt und weiterentwickelt, die sie geschrieben haben
 - ⇒ Programme müssen (durch Dritte!) **lesbar** und **verstehbar** sein
- Schlechte Qualität ist **teuer**
 - ⇒ Sorgfältiges Programmieren ist **billiger** als hacken
- ⇒ **Systematisches Programmieren**

Experiment: Verstehen eines Programms

Aufgabe: Lesen Sie das Programm auf dem ausgeteilten Blatt.

Sie haben dafür drei Minuten Zeit.

Nach Ablauf dieser Zeit schreiben Sie in Stichworten auf, was dieses Programm tut.



2.1 Einführung

2.2 Namengebung

2.3 Datendefinitionen

2.4 Ablaufkonstrukte

2.5 Unterprogramme

2.6 Optimierung

2.7 Dokumentation

Gute Namen sind wesentlich für die Lesbarkeit

Was ist hier falsch?

- `PS_YCRD = PS_XCRD + AD_BRK * AD_BCO;`
- `y = x + d *b;`
- `IF GAGA = BALA PERFORM GUGUS.`
- `void strc (char *s, char *t)`
- `double MaxSchrittweite, upperLimit, AVG_DLT;`

Wahl guter Namen

Warum ist das besser?

- Textmitte := Textlaenge DIV 2;
- 01 Neuer-Kunde.
 - 05 Name PIC X(20).
 - 05 Vornamen PIC X(32).
- private static final double TOLERANZ = 0.30;
- if (temp.wert < (referenzwert * (1.0 - TOLERANZ))) {...

Regeln für die Wahl von Namen

- **Variablennamen** bezeichnen ihren **Inhalt**: nextState, topWindow, brake_coefficient, ARTIKEL-NR
- **Prozedur-** bzw. **Methodennamen** bezeichnen ihre **Aufgabe**: PrintPage, CalculateDelay, Compare_with_limit
- **Symbolische Konstanten** bezeichnen ihren **Wert**: MaxOpenWindows, DEFAULT_SPEED
- **Grundtypen** bezeichnen einen **Gegenstand** oder einen **Begriff** und haben **einfache Namen**: File, Table, Speed
- **Abgeleitete Typen** und **Komponententypen** haben entsprechend **zusammengesetzte Namen**: SequentialFile, TableIndex, TableValue

Groß oder Klein? Zusammen oder getrennt?

- Unterschiedliche Konventionen möglich, zum Beispiel:
 - Jeder Namenbestandteil beginnt mit Großbuchstaben: DefaultValue
 - Typen beginnen immer groß, Variablen immer klein: File (Typ), logFile (Variable)
 - Namenbestandteile werden durch Trennzeichen getrennt:
 - KUNDEN-ADRESSE (nur in Cobol, sonst Subtraktion!)
 - default_init_value (Pascal- und C-Familie, Java)
- Sich an die Codierrichtlinien der Organisation, in der man arbeitet, halten
- Verwendete Konventionen konsequent durchhalten
- Keine Namen, die sich nur durch Groß-/Kleinschreibung unterscheiden
- Sprachen und Schreibstile nicht mischen

Länge von Namen

- Namen mit **kleinem Gültigkeitsbereich** können **kurz** sein
- Namen mit **großem Gültigkeitsbereich** müssen **selbsterklärend** sein
- **Kurznamen** (i, m, y, dx, Rs) demnach
 - nur für **Schleifenindizes** in kurzen Schleifen
 - oder in **einfachen mathematischen Formeln** in kurzen Prozeduren / Methoden
 - aber **niemals für Prozedur-/Methodennamen** oder für Typnamen
- **Abkürzungen vermeiden**: DistanceCounter ist besser als DST_CTR
- **Alles mit Maß**: CarControlMainBrakingSystemMaximumDistancePointerDefaultValue ist zu viel des Guten
- **Faustregel**: 8-20 Zeichen für Variablen, 15-30 Zeichen für Prozeduren/Methoden

Gültigkeitsbereich von Namen

- Jeder Name hat in seinem Gültigkeitsbereich **nur eine Bedeutung**
 - Beispiel: Eine Prozedur berechnet eine Iterationsformel mit einer gegebenen Schrittweite Δx ; als Resultate werden das Ergebnis und die Abweichung von einem Referenzwert zurückgegeben
Es wäre falsch, eine Variable mit dem Namen Delta
 - während der Berechnung für die Schrittweite
 - danach bei der Ausgabe für die Abweichung zu verwenden
- **Vorsicht bei der Überlagerung** von Gültigkeitsbereichen: führt leicht zu Fehlern, wenn die Überlagerung beim Programmieren oder Lesen übersehen wird

Mini-Übung 2.1

Gegeben ist folgendes Programmfragment in C++:

```
const int maxBufferSize = 1024;
char zeichen = 'A';
...
// String im Puffer auf maximal 512 Byte kappen
for (int i=0; i<maxBufferSize; i++) {
    const char zeichen = '\0'; //Nullbyte
    if (buffer[i] == zeichen) break;
}
if (i>=512) buffer[i] = zeichen; //Abschneiden
...
```

- Nehmen Sie Stellung zu den gewählten Namen
- Funktioniert dieses Programmfragment so wie es soll?

Merkmale Namengebung

- Die Wahl der Namen ist wesentlich für das Verständnis eines Programms
- Namen nach einheitlichem Stil und einheitlichen Regeln wählen
- Kurznamen nur einfache Variablen mit kleinem Gültigkeitsbereich
- Namen von Prozeduren / Methoden und Typen selbsterklärend

2.1 Einführung

2.2 Namengebung

2.3 Datendefinitionen

2.4 Ablaufkonstrukte

2.5 Unterprogramme

2.6 Optimierung

2.7 Dokumentation

Variablen

- Variablen sind **Behälter für Daten**
- Variablen
 - können eine **Struktur** haben
 - geeignete Struktur wählen
 - haben einen **Gültigkeits-/Sichtbarkeitsbereich**
 - so klein wie möglich
 - so groß wie nötig
 - nur eine Bedeutung im ganzen Gültigkeitsbereich
 - müssen **initialisiert** sein, bevor sie das erste Mal gelesen werden
 - Unterschied von statischer und dynamischer Initialisierung beachten

Definition von Daten und Datenstrukturen

Wo stecken die Probleme in den folgenden Codefragmenten?

```
public double pi = 3.141593;
```

```
01  Strasse          PIC X(20).  
01  Hausnummer     PIC 9(4) COMP.
```

```
public class Betriebsart {  
    public static boolean online = false;  
  
    // Führt das System aus dem offline Mode hoch in den online Mode  
    public static boolean Startup() {  
        if ( CheckSensors() & InitializeDatabase() && OpenLog () )  
            return (online = true); else return false;  
    }  
    ...  
}
```

Definition von Daten und Datenstrukturen – 2

Welches Problem haben wir hier?

* Diese Variablen werden nacheinander gebraucht; spart 20 Byte

01 Lese-Zaehler PIC 9(10).

01 Hilfs-Feld REDEFINES Lese-Zaehler PIC X(10).

01 Schreib-Zaehler REDEFINES Lese-Zaehler PIC 9(10).

*

Datenstrukturen

Variablen können eine **innere Struktur** haben

- Diese **Struktur** in den Datendefinitionen **abbilden**
 - Cobol: gestufte Felder
 - Pascal, Ada: Records
 - C, C++: Structs, Klassen
 - Java: Klassen
- **Strukturen konsistent manipulieren**
 - Verwendung von Typen, Typprüfung bei/vor Übersetzung
 - Datenmanipulation nur über Methoden

Globale vs. lokale Variablen

- **Lokale Variablen** sind nur im definierenden Programm/Unterprogramm und allen darin eingeschachtelten Unterprogrammen sichtbar
- **Sichtbar** heißt **lesbar** und **schreibbar**(!)
- **Globale Variablen** sind überall sichtbar bzw. können durch Import sichtbar gemacht werden
- Globale Variablen
 - + **einfachste** Form der Kommunikation zwischen verschiedenen Programmteilen
 - koppeln die Programmteile stark
 - führt zu unerwünschten **Nebenwirkungen**
 - **verschlechtert** die **Wartbarkeit** dramatisch

Datentypen: Äpfel nicht mit Birnen vergleichen

- **Einfache Typen:** int, float, char, boolean, PIC X, PIC S9999V99
- **Satztypen:** In Cobol direkt:
 - 01 Neuer-Kunde.
 - 05 Name PIC X(20).
 - 05 Vornamen PIC X(32).

In Java und C++ mit Klassen zu programmieren, in C++ auch mit struct möglich:

```
class Datum {  
    int Jahr;  
    int Monat;  
    int Tag;  
}  
  
struct Student {  
    char Name[30];  
    int Matrikelnummer;  
}
```

- **Felder (arrays):**
 - int matrix [16] [12]
 - 05 KONTO OCCURS 8 TIMES PIC X(10).

Datentypen – 2

- **Aufzählungstypen:**

```
enum {online, offline}; // C++
```

```
class OperatingMode { // Java
    private int mode = 0;
    public boolean isOffline() {return mode == 0;}
    public boolean isOnline() {return mode == 1;}
    public void setOffline() {mode = 0;}
    public void setOnline() {mode = 1;}
}
```

```
OperatingMode monitor = new OperatingMode(); ...
... if ( monitor.isOnline() ) ...
```

```
05      OPERATING-MODE PIC 9. >* Cobol
      88      OFFLINE      VALUE 0.
      88      ONLINE      VALUE 1.
```

Benannte, benutzerdefinierte Typen

- C++: mit **typedef**, aber leider **ohne Prüfung**

```
typedef int Tag; typedef int Preis;  
Tag heute; Preis tagesPreis;
```

```
if (heute == tagesPreis) ... // falsch, wird aber vom Compiler akzeptiert
```

- Java: Mit **Klassen** programmieren

- Cobol: **Nicht möglich**. Strukturgleichheit kann durch **COPY** sichergestellt werden

Konstanten

- The maintainer's nightmare: Literale in Programmen

Warum sind diese Konstruktionen ein Wartungs Alptraum?

```
PERFORM VARYING W-I FROM 1 BY 1 UNTIL W-I = 7  
  COMPUTE W-SUMME = W-SUMME + W-WERT (W-I)  
END-PERFORM.
```

```
if (i >= 512) buffer[i] = zeichen;
```

```
String home = "http://www.ifi.unizh.ch/req";  
...  
if (done) SetPage(home);
```

Konstanten – 2

Warum sind diese Konstruktionen besser?

78 Wochentage VALUE 7. >* Cobol

const int bufferSize = 512; // Size of input buffer C++

public static final String HOME = "http://www.ifi.unizh.ch/req"; // Java

Hinweis: Konstanten, deren Voreinstellung **änderbar** sein soll, müssen als **Daten** abgelegt und **eingelese**n werden. In Java und C++ werden solche Werte durch Kapselung in Klassen gegen unbeabsichtigte Veränderung gesichert.

Merkmale Datendefinitionen

- Jede Variable mit passendem Namen und nur für einen Zweck
- Gültigkeitsbereiche so klein wie möglich
- Geeignete Datenstrukturen wählen
- Konsistenz und Verarbeitungssicherheit durch Verwendung von Typen (und Typprüfung) gewinnen
- Literale als symbolische Konstanten definieren

2.1 Einführung

2.2 Namengebung

2.3 Datendefinitionen

2.4 Ablaufkonstrukte

2.5 Unterprogramme

2.6 Optimierung

2.7 Dokumentation

Ein schlecht lesbares Stück Code

```
...
IF (Monat = 1) AND (Tag = 1) PERFORM Init10.
  PERFORM A-Umsatz.
N10.
  PERFORM A-Prognose.
  IF (Tag = 1) AND (Monat > 1) GO TO Init 20.
  IF Tag > 1 PERFORM M-Umsatz.
N20.
  PERFORM M-Prognose.
  GO TO 99.
Init10 SECTION.
  PERFORM Init-A-Umsatz.
Init20.
  PERFORM Init-M-Umsatz.
  IF Monat > 1 GO TO N20.
GO TO N10.
```

- 1 -

Weshalb ist dieser Code schlecht lesbar?

```
***** Unterprogramme *****
A-Umsatz SECTION.
...
*****
99.
  STOP RUN.
```

- 2 -

2.4.1 GOTO und geschlossene Ablaufkonstrukte

- **Problem:** Der **dynamische Ablauf** des Programms ist aus der **statischen Programmstruktur** nur mit Mühe rekonstruierbar
- Bei gut strukturierten Programmen **stimmen** statische Struktur und dynamischer Ablauf weitgehend **überein**
- **“Go To Statement Considered Harmful”**
“... we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.” (Dijkstra 1968)
- GOTO **bricht** bei unbedachter Verwendung die **Übereinstimmung** zwischen statischer und dynamischer Struktur
- ⇒ GOTO (und Anverwandte, z. B. **break** in C/C++ und Java) nur unter Erhaltung geschlossener Ablaufkonstrukte verwenden

Ein strukturiertes Programm

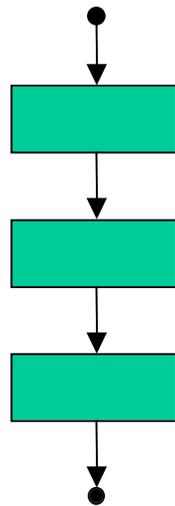
Das gleiche Programmfragment restrukturiert:

```
... - 1 -  
Umsatz SECTION.  
  IF Tag > 1      >* kein Monatsanfang  
    PERFORM M-Umsatz  
    PERFORM A-Umsatz  
  ELSE            >* Monatsanfang  
    PERFORM Init-M-Umsatz  
    IF Januar  
      PERFORM Init-A-Umsatz  
    ELSE  
      PERFORM A-Umsatz  
    END-IF  
  END-IF.  
END-IF.
```

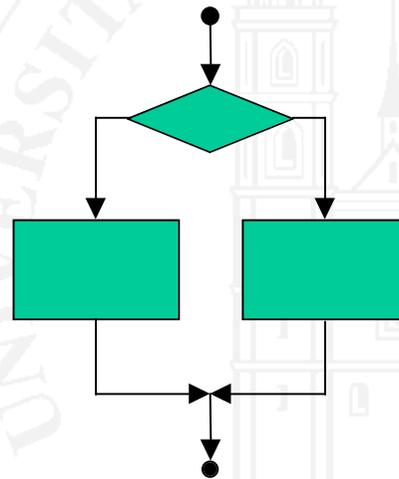
```
- 2 -  
Prognose SECTION.  
  PERFORM A-Prognose.  
  PERFORM M-Prognose.  
  STOP RUN.  
***** Unterprogramme *****  
A-Umsatz SECTION.  
...  
  EXIT.  
*****
```

Geschlossene Ablaufkonstrukte

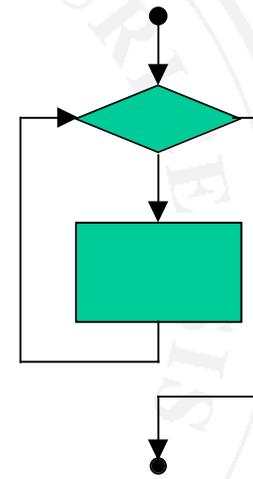
Ein **geschlossenes Ablaufkonstrukt** ist ein Element des Programmablaufs mit **genau einem Eintritts-** und **einem Austrittspunkt**



Sequenz



Alternative

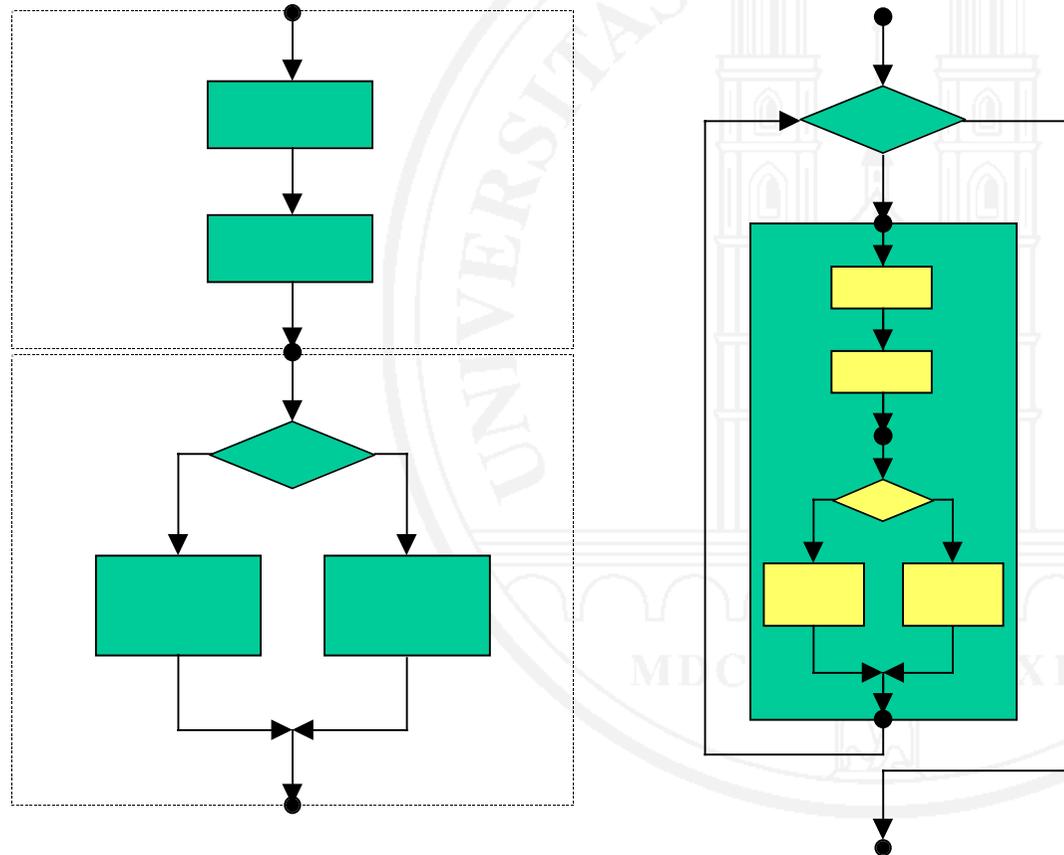


Iteration

Jedes sequentielle Programm ist aus den Grundelementen Sequenz, Alternative und Iteration konstruierbar (Böhm und Jacopini, 1966)

Geschlossene Ablaufkonstrukte – 2

Geschlossene Ablaufkonstrukte können beliebig **aneinandergereiht** und **verschachtelt** werden



Exkurs: GOTO-freies Programmieren

- GOTO-freie Programme sind **nicht immer gut strukturierte** Programme:
 - Entscheidend ist, dass die zu Grunde liegende **Logik klar** in Code **umgesetzt** wird
 - Manchmal kann ein GOTO ein Programm sogar **vereinfachen**
- Beispiel: Vier unterschiedliche, funktional äquivalente Programme

Variante 1: Schlecht strukturiert, aber GOTO-frei

```
int ComputeAResult (int x, int y, int n)
{
    boolean a = false, b = false; int z = 0;
    for (int i = 1; !a & !b & i <= n; i++)
    {
        a = GetCondition (x, i);
        if (!a)      b = GetCondition (y, i);
        else if (!b)  z = 1;
    }
    if (a)      z = Compute (x);
    else if (b)  z = Compute (y);
    else      z = Compute (x*y);
    return z;
}
```

Variante 2: Besser strukturiert, aber kompliziert und ineffizient

```
int ComputeAResult (int x, int y, int n)
{
    boolean a = false, b = false; int z = 0;
    for (int i = 1; !a & !b & i <= n; i++)
    {
        a = GetCondition (x, i);
        if (a) z = Compute (x);
        else
        {
            b = GetCondition (y, i);
            if (b) z = Compute (y);
            else z = Compute (x*y);
        }
    }
    return z;
}
```

Wann und wo ineffizient?

In jedem Fall funktional
äquivalent mit Variante 1?

Variante 3: Effizienter, aber immer noch kompliziert

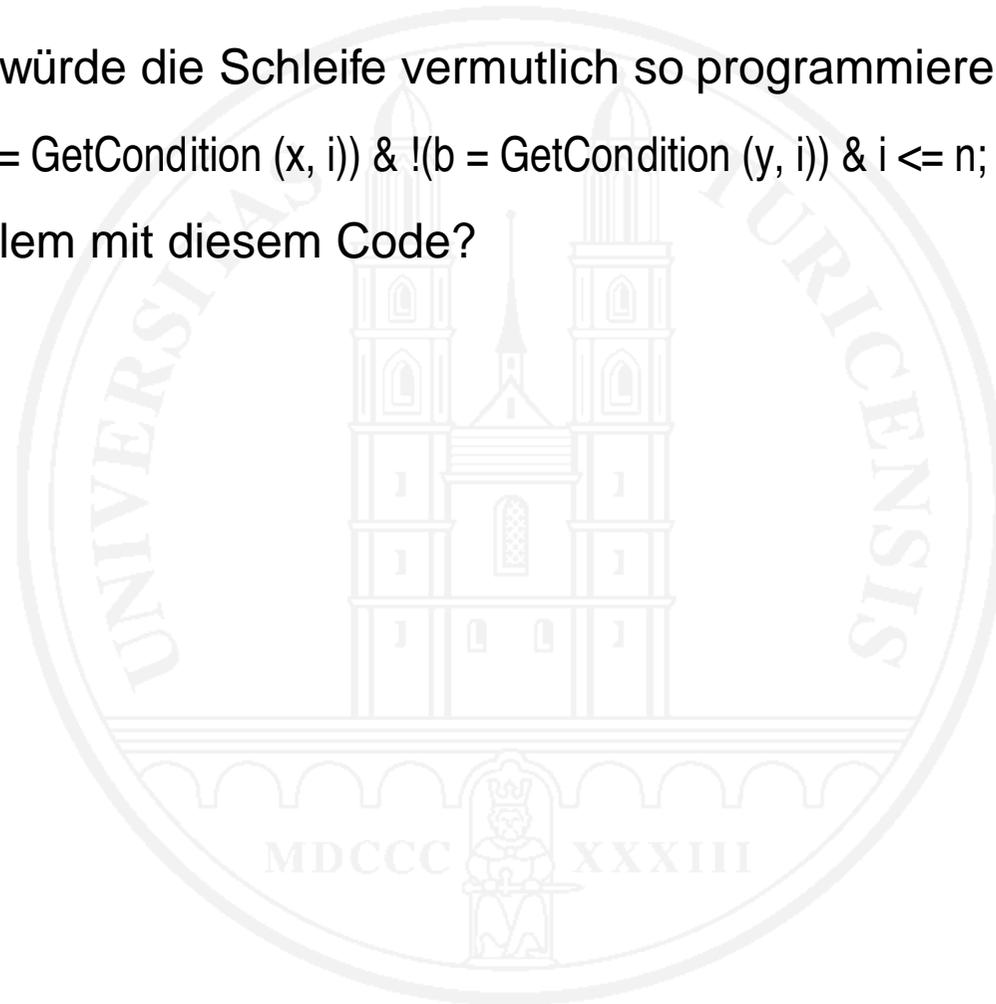
```
int ComputeAResult (int x, int y, int n)
{
    boolean a = false, b = false; int z = 0;
    for (int i = 1; !a & !b & i <= n; i++)
    {
        a = GetCondition (x, i);
        b = GetCondition (y, i);
    }
    if (a)    z = Compute (x);
    else if (b) z = Compute (y);
    else     z = Compute (x*y);
    return z;
}
```

Anmerkung zu Variante 3

Ein C-Liebhaber würde die Schleife vermutlich so programmieren:

```
for (int i = 1; !(a = GetCondition (x, i)) & !(b = GetCondition (y, i)) & i <= n; i++);
```

Was ist das Problem mit diesem Code?



Variante 4: Besser als alle anderen Varianten

```
int ComputeAResult (int x, int y, int n)
{
    for (int i = 1; i <= n; i++)
    {
        if (GetCondition (x, i))    return Compute (x);
        else if (GetCondition (y, i)) return Compute (y);
    }
    return Compute (x*y);
}
```

- **return** ist ein GOTO zum Ende der Methode
- Klarste und einfachste Umsetzung des Algorithmus

2.4.2 Fallunterscheidung

- Verschachtelte IF-Anweisungen sorgfältig aufbauen
- In Verschachtelungen
 - immer **Blockklammern/END IF** verwenden
 - IF ohne ELSE-Zweig vermeiden
 - ELSE IF ist besser als THEN IF
- **Große Fallunterscheidungen** mit **switch** (Java, C++) / **CASE** (Pascal, Modula-2) bzw. **EVALUATE** (Cobol) aufbauen

Beispiel: Textcharakteristika bestimmen (Java)

```
String TextCharacteristics (String text, int length, int firstPos, int lastPos)
// PRE
// text   Zeichenkette mit n Zeichen
// length Länge der Zeichenkette, d.h. n
// firstPos   Index des ersten nicht leeren Zeichens in text
// lastPos    Index des letzten nicht leeren Zeichens in text
// firstPos und lastPos haben den Wert -1, falls die Zeichenkette leer ist (n=0) oder nur
// aus Leerzeichen besteht

// POST
// Funktionswert ist ...
// "empty"   wenn length = 0
// "oneBlank"  wenn length = 1 und nur Leerzeichen
// "twoBlanks" wenn length = 2 und nur Leerzeichen
// "noText"   wenn length > 2 und nur Leerzeichen
// "leadingBlanksOnly" wenn firstPos > 0 und lastPos = length-1
// "trailingBlanksOnly" wenn firstPos = 0 und lastPos < length-1
// "leadingAndTrailingBlanks" wenn firstPos > 0 und lastPos < length-1
// "textOnly" wenn firstPos = 0 und lastPos = length-1
```

Schlechte Struktur

```
{
if (firstPos == -1)
{
if (length > 0)
{
if (length == 1) return "oneBlank";
if (length == 2) return "twoBlanks";
else return "noText";
}
else return "empty";
}
else if (firstPos > 0)
{
if (lastPos == length-1) return "leadingBlanksOnly";
else return "leadingAndTrailingBlanks";
}
else if (lastPos < length-1) return "trailingBlanksOnly";
else return "textOnly";
}
```

Gute Struktur

```
{
if (length == 0)
    {return "empty";}
else if (length == 1 & firstPos == -1)
    {return "oneBlank";}
else if (length == 2 & firstPos == -1)
    {return "twoBlanks";}
else if (length > 2 & firstPos == -1)
    {return "noText";}
else if (firstPos > 0 & lastPos == length-1)
    {return "leadingBlanksOnly";}
else if (firstPos == 0 & lastPos < length-1)
    {return "trailingBlanksOnly";}
else if (firstPos > 0 & lastPos < length-1)
    {return "leadingAndTrailingBlanks";}
else
    {return "textOnly";}
}
```

2.4.3 Iteration

- Das Prinzip der Iteration:
 - Wiederholte Ausführung einer Gruppe von Anweisungen in einer **Schleife**
 - **Vorwärtsberechnung**: Resultat wird typisch inkrementell aufgebaut
 - Schleife muss **explizit gesteuert** werden: **Initialisierung**, **Schleifenbedingung**, **Fortschaltung** (letzteres nur bei Zählschleifen)
- Alle Elemente der Schleifensteuerung sind anfällig auf **Fehler**
- **Systematische Konstruktion** von Schleifen:
 - Konstruktion des **Schleifenkörpers** so, dass bei Verlassen der Schleife das erwartete Resultat vorliegt
 - Davon ausgehend **Initialisierung**, **Schleifenbedingung** und ggf. **Fortschaltung** bestimmen
 - Prüfen, ob die Schleife immer **terminiert**

Grundmuster für Schleifen

Muster 1: Datensätze verarbeiten

```
lies erstes datum;  
while (nicht fertig) {  
    verarbeite datum;  
    lies nächstes datum;  
}
```

Muster 2: Datenwerte akkumulieren

```
initialisiere zähler;  
while (nicht fertig) {  
    akkumuliere;  
    erhöhe zähler;  
} // meistens als for-Schleife programmiert
```

Abweisende vs. annehmende Schleifen

- **Abweisende Schleifen** prüfen die Schleifenbedingung **vor** dem Durchlauf durch den Schleifenkörper
 - Bei a priori nicht erfüllter Schleifenbedingung wird die Schleife **nicht** durchlaufen
 - Java, C++: **while, for** Cobol: PERFORM UNTIL, PERFORM VARYING
 - **Annehmende Schleifen** prüfen die Schleifenbedingung erst **nach** dem Durchlauf durch den Schleifenkörper
 - Unabhängig von der Schleifenbedingung wird der Schleifenkörper **mindestens einmal durchlaufen**
 - Java, C++: **do - while** Cobol: PERFORM WITH TEST AFTER UNTIL
 - Falsch, wenn die Schleifenbedingung a priori nicht erfüllt ist
 - **Häufige Fehlerquelle**
- ⇒ Abweisende Schleifen sind **sicherer** als annehmende

Mini-Übung 2.2: Wo ist der Fehler in dieser Prozedur?

```
PROCEDURE LokalisiereLetztes (text: ARRAY OF CHAR): INTEGER;  
(* Liefert die Position des letzten nicht leeren Zeichens in der Zeichenkette text oder -1,  
wenn text nur aus Leerzeichen besteht oder gar keine Zeichen enthält  
*)  
CONST leer = " ";  
VAR letztePos: INTEGER;  
BEGIN  
    letztePos := Length (text) - 1;  
    REPEAT  
        IF text[letztePos] = leer  
            THEN letztePos := letztePos - 1;  
        END (* IF *);  
    UNTIL (letztePos < 0) OR (text [letztePos] <> leer);  
    RETURN letztePos;  
END LokalisiereLetztes;
```

Sprache: Modula-2

Heuristiken für die Konstruktion guter Schleifen

- Bei Java/C++ immer **Blockklammern** verwenden

«Dieses Programm funktioniert einfach nicht, obwohl der Algorithmus absolut richtig programmiert ist!»

```
preisTotal = 0;
for (i=0; i<= max-1; i++);
    preisTotal = preisTotal + bestellung[i].einzelPreis;
```

- Immer die Fälle **kein Durchlauf**, **ein Durchlauf**, **maximal mögliche Zahl** von Durchläufen auf Korrektheit überprüfen
- Annehmende Schleifen nur verwenden wenn in jedem Fall mindestens ein Durchlauf erforderlich ist
- **Nebenwirkungen** vermeiden (siehe 2.4.5)

Formale Konstruktion korrekter Schleifen

- Konstruktion des Schleifenkörpers
- Bestimmung einer geeigneten Schleifeninvariante
Schleifeninvariante: Ein Prädikat, das nach jeder Prüfung der Schleifenbedingung wahr ist
- Ableitung von Initialisierung, Schleifenbedingung und Fortschaltung aus der Schleifeninvariante
- Prüfen, ob die Schleife terminiert
- Schwierigkeit: Finden einer geeigneten Invariante
- Typisches Vorgehen:
 - Ausgehen vom erwarteten Resultat der Schleife
 - Finden eines Ausdrucks, welcher inkrementell zu diesem Resultat führt

Konstruktionsbeispiel – 1: Problem und Invariante

Zu berechnen sei die **Summe eines n-elementigen Koeffizientenvektors**

Konstruktionsidee für die Schleife:

```
double [ ] vektor; double vektorsumme;  
vektorsumme = ?;  
for (int i = ?; ? ; ?)  
    vektorsumme = vektorsumme + vektor[i];
```

Schleifeninvariante: $\text{vektorsumme} = \sum_{k=0}^{i-1} \text{vektor}[k]$

Konstruktionsbeispiel – 2: Fortschaltung

Konstruktion der **Fortschaltung**, so dass die Invariante wahr ist :

Fortschaltung: $i++$

Beweis: Zu Beginn des m -ten Schleifendurchlaufs habe i den Wert p und es gelte

$$\text{vektorsumme} = \sum_{k=0}^{i-1} \text{vektor}[k].$$

Berechnet wird im m -ten Schleifendurchlauf mit $i = p$:

$$\begin{aligned} \text{vektorsumme} &= \\ \text{vektorsumme} + \text{vektor}[p] &= \sum_{k=0}^{p-1} \text{vektor}[k] + \text{vektor}[p] = \sum_{k=0}^p \text{vektor}[k] \end{aligned}$$

Damit bei der nächsten Prüfung der Abbruchbedingung die Invariante noch gilt, muss i dann den Wert $p+1$ haben, d.h. i muss im m -ten Durchlauf um 1 erhöht werden.

Konstruktionsbeispiel – 3: Initialisierung

Konstruktion der **Initialisierung**, so dass die Invariante beim ersten Test der Schleifenbedingung wahr ist :

Initialisierung: $i = 0$, $\text{vektorsumme} = 0$

Beweis: Nach der ersten Prüfung der Schleifenbedingung muss gelten

$$\text{vektorsumme} = \sum_{k=0}^{i-1} \text{vektor}[k] = 0.$$

Dies ist bei der üblichen Definition des Summensymbols der Fall:

$$\sum_{k=n}^m s_k = 0 \text{ für alle } m < n$$

Hinweis: Überlegen Sie sich eine andere Initialisierung, welche die Schleifeninvariante ebenfalls erfüllt

Konstruktionsbeispiel – 4: Schleifenbedingung

Konstruktion der **Schleifenbedingung**, so dass

- die **Invariante** beim Verlassen der Schleife **wahr** ist
- das **Ergebnis** der Schleife **aus** der **Invariante abgeleitet** werden kann

Schleifenbedingung: $i < n$

Beweis: Beim Verlassen der Schleife ist $i = n$, also

$$\text{vektorsumme} = \sum_{k=0}^{n-1} \text{vektor}[k]$$

- ⇒ Die Invariante ist wahr
- ⇒ Die Invariante ist gleich dem gewünschten Resultat, wobei auch die Gültigkeit des Resultats bewiesen ist

Konstruktionsbeispiel – 5: Terminierung

Zeigen, dass die Schleife immer **terminiert**

Beweis:

i wird bei jedem Schleifendurchlauf inkrementiert, n bleibt konstant

- ⇒ Die Bedingung $i < n$ ist nach endlich vielen Schritten falsch
- ⇒ Die Schleife terminiert

Verifikation von Schleifen

- **Schleifeninvarianten** können auch verwendet werden, um die **Korrektheit** einer bereits programmierten Schleife zu **verifizieren**:
Sei S eine Schleife und seien
 - N ein Prädikat, welches das erwartete Ergebnis von S beschreibt
 - V ein Prädikat, das die notwendigen Voraussetzungen für S beschreibt
 - b die Schleifenbedingung S
 - Inv eine Schleifeninvariante von S
- Durch **Beweis** von
 - (i) $V \Rightarrow Inv$
 - (ii) $(Inv \wedge \neg b) \Rightarrow N$
 - (iii) S terminiertwird die **Korrektheit** von S **verifiziert**

Mini-Übung 2.3: Schleifenverifikation

Verifizieren oder falsifizieren Sie die Korrektheit der folgenden Schleife:

```
double [] vektor; double vektorprodukt;  
//ASSERT  $n > 0 \wedge$  vektor hat  $n$  Komponenten  
    vektorprodukt = vektor [0];  
    for (int i = 0; i < n ; i++)  
        vektorprodukt = vektorprodukt * vektor[i];  
//ENSURE vektorprodukt = Produkt der Komponenten von vektor
```

2.4.4 Rekursion

Das Prinzip der Rekursion

- Mehrfache Ausführung von Anweisungen durch **wiederholten Selbstaufruf**
- **Rückwärtsberechnung**: Resultat wird durch rekursiven Abstieg gewonnen
- Steuerung ist **implizit**: Nur Reduktion und Verankerung müssen sichergestellt sein
- Im Vergleich dazu **Iteration**:
 - **Vorwärtsberechnung**
 - **Explizite** Steuerung erforderlich

Beispiel zur Rekursion – 1

Zu berechnen sei die Summe eines n-elementigen Koeffizientenvektors

Rekursionsformel:

$$\sum_{i=1}^n x_i = \begin{cases} x_n + \sum_{i=1}^{n-1} x_i & \text{falls } n > 0 \\ 0 & \text{sonst} \end{cases}$$

Beispiel zur Rekursion – 2

Mit einer einfachen Substitution (das n-te Element in vektor hat den Index n-1) kann diese Formel direkt programmiert werden:

```
double Vektorsumme (double [ ] vektor, int n);  
// Berechnet die Summe der Komponenten des Vektors vektor mit n Komponenten  
int indexDesNten; //Variable programmtechnisch überflüssig; nur zur Verdeutlichung  
{  
    indexDesNten = n - 1;  
    if (n <= 0) return 0;  
    else return vektor[indexDesNten] + Vektorsumme (vektor, n-1);  
}
```

Vor- und Nachteile der Rekursion

- + Rekursive Lösungen sind einfacher und kürzer als iterative
- + Bei gegebenen Rekursionsformeln ist die Korrektheit viel einfacher zu zeigen als bei Schleifen: Zu verifizieren sind
 - die eigentliche Rekursionsformel (reduziert sie korrekt?)
 - die Rekursionsverankerung (ist der Startwert korrekt?)
- Mehrfachrekursion kann zu Laufzeit- und Speicherproblemen führen
Beispiel: Fibonacci-Zahlen $\text{fib}(1) = \text{fib}(2) = 1, \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
 - Ab $n > 3$ mehr als $\text{fib}(n)$ rekursive Aufrufe nötig
 - Zum Vergleich: iterativ mit einer Schleife der Länge n lösbar
- Rekursion ist gedanklich schwieriger nachzuvollziehen als Iteration

Rekursion vs. Iteration

- Jede Iteration kann in eine Rekursion transformiert werden und umgekehrt
- Die Programmkonstrukte dem Problem anpassen:
 - Iterativ formulierte Probleme mit Iteration
 - Rekursiv formulierte Probleme mit Rekursion lösen
- Keine Angst vor Rekursion!

2.4.5 Nebenwirkungen

Was macht dieses Programmfragment in C++?
Warum ist das schlechter Code?

```
char *t = "Programm mit Nebenwirkungen:";
{ ...
while (*t++);
t--;
...
}
```

Wie geht es besser?

Merkmale Ablaufkonstrukte

- Programmablauf gut strukturieren
- Geschlossene Ablaufkonstrukte verwenden
- Jedes sequentielle Programm ist mit geschlossenen Ablaufkonstrukten (Sequenz, Alternative, Iteration) konstruierbar
- Fallunterscheidungen und Schleifen systematisch konstruieren
- Konstruktion/Verifikation von Schleifen ist möglich
- Passend zum Problem Rekursion oder Iteration einsetzen
- Nebenwirkungsfreie Programmkonstrukte wählen
- Größere Programme in Prozeduren/Methoden und Klassen bzw. Module gliedern (vgl. Abschnitt 2.5)

2.1 Einführung

2.2 Namengebung

2.3 Datendefinitionen

2.4 Ablaufkonstrukte

2.5 Unterprogramme

2.6 Optimierung

2.7 Dokumentation

Was ist ein Unterprogramm

Unterprogramm – Benanntes, **abgegrenztes Programmstück**, das unter seinem Namen **aufzurufbar** ist

- Beim Aufruf eines Unterprogramms **verzweigt** die Steuerung zum Anfang des Unterprogramms und **kehrt** nach Ausführung des Unterprogramms an die Aufrufstelle **zurück**
- Wozu Unterprogramme?
 - Programme **besser strukturieren** ⇔ Lesbarkeit
 - Programmstücke **gemeinsam nutzen** ⇔ Effizienz
- Es gibt verschieden mächtige Formen von Unterprogrammen

Formen von Unterprogrammen

- **Benannter Block** (Unterprogramm in Assembler, PERFORM-Block in Cobol)
 - Syntaktisch **separiert**
 - **Benannt** und **aufrufbar**
 - Kein separater Namensraum und keine lokalen Daten
 - Keine Parameterersetzung
- **Prozedur** (procedure, function in Pascal, C, etc. Programmverbindung in Cobol)
 - **Separater Namensraum**, **lokale** Daten, **Parameterersetzung**
 - Parameterübergabe mit “call by value” oder “call by reference”
 - Datenaustausch auch über globale Daten möglich
 - Statische Bindung an den aufgerufenen Stellen durch Übersetzer

Formen von Unterprogrammen – 2

○ Methode

- Wie Prozedur, aber mit **dynamischer Bindung** an Aufrufer zur Laufzeit
 - **Polymorphie möglich**, d.h. verschiedene Methoden gleichen Namens innerhalb einer Vererbungshierarchie. Die Auswahl der passenden Methode erfolgt zur Laufzeit
-

○ Makro (zur Abgrenzung, Makros sind **keine Unterprogramme**)

- Benanntes, abgegrenztes Programmstück, das unter seinem Namen referenzierbar ist
- Bei Referenzierung wird der Makrokörper vom Übersetzer an der Referenzstelle in den Code inkopiert
- Parameterersetzung ist möglich
- Vor allem in maschinennahen Programmiersprachen nützlich

Bildung von Unterprogrammen

- Unterprogramme werden gebildet
 - Zur **Strukturierung** langer Programme in inhaltlich zusammenhängende Einheiten
 - Zur **gemeinsamen Nutzung** von Code, der an verschiedenen Stellen eines Programms benötigt wird
- Gute Unterprogramme haben
 - **hohe Kohäsion**
 - Starker innerer Zusammenhang
 - Optimal: Unterprogramm berechnet eine Funktion
 - **geringe Kopplung** (mit anderen Programmen/Unterprogrammen)
 - So wenig gemeinsame Daten wie möglich
 - Keine direkte Beeinflussung des Kontrollflusses in einem Unterprogramm durch ein anderes Programm/Unterprogramm

Mini-Übung 2.4

Beurteilen Sie Kohäsion und Kopplung:

- a) Das Unterprogramm berechnet das Alter eines Mitarbeiters aus seinem Geburtsdatum. Zu diesem Zweck wird dem Unterprogramm der Mitarbeiter-Stammdatensatz (mit insgesamt rund 50 Feldern) zur Verfügung gestellt.
- b) Das Unterprogramm druckt wahlweise die Wochenumsatzstatistik, die Monatsumsatzstatistik oder die Jahresumsatzstatistik. Die Auswahl wird über ein Flag gesteuert. Die Daten befinden sich in Dateien; der jeweilige Dateiname wird als Parameter übergeben.
- c) Das Programm saldiert den Monatsumsatz, die Überzeitguthaben der Mitarbeitenden und die Zahl der beratenen Kunden.

Bildung von Unterprogrammen – 2

- Jedes Unterprogramm bildet eine **Kapsel**, welche die **lokalen Daten** und **Algorithmen** nach außen **unsichtbar** macht, soweit dies möglich ist
- In Situationen wo
 - ein Unterprogramm lokale Daten mit anderen Unterprogrammen teilen muss,
 - die Daten aber vor den Verwendern des Unterprogramms verborgen werden können,
 - ⇒ die **Daten zusammen** mit allen darauf arbeitenden **Unterprogrammen** in einer Klasse **kapseln**

Parameter

Parameter dienen der **Kommunikation** zwischen dem **Aufrufer** und dem **Unterprogramm**

- **Formale Parameter** im Unterprogramm: Liste von Platzhaltern
- **Aktuelle Parameter** beim Aufruf: Aktuelle Werte / Variablen, die ans Unterprogramm übergeben werden
- **Zuordnung** aktuelle Parameter → formale Parameter nach **Reihenfolge** in der Liste
- **Anzahl** und **Datentypen** der aktuellen Parameter sollten mit den formalen Parametern **übereinstimmen**

Parameter – 2

- **Parameterarten**
 - **Eingabeparameter** (im Unterprogramm nur gelesen)
 - **Ausgabeparameter** (im Unterprogramm nur geschrieben)
 - **Ein/Ausgabeparameter** (gelesen und geschrieben)
- **Arten der Parameterübergabe**
 - **“by value”**: Parameterwert wird **kopiert**, **Original** im Aufrufer **bleibt unverändert**
 - **“by reference”**: **Adresse** wird übergeben, Schreiben des Parameters im Unterprogramm **verändert** das **Original** im Aufrufer
- **Hinweis**: Wird in objektorientierten Sprachen ein Objekt als Parameter übergeben, so erfolgt die Übergabe **“by value”**, d.h. eine Kopie der Objektreferenz wird übergeben. Faktisch bedeutet dies jedoch auch Schreibzugriff(!) auf das übergebene Objekt

Anweisungs-Unterprogramme und Funktionen

- Unterprogrammaufrufe können **Anweisungen** sein
 - Beispiele: CALL Kalkuliere-Angebot.
PrintImage (coordinates, source);
- Unterprogramme können **Funktionen** sein
 - Geben einen **Wert** zurück
 - Aufruf ist **Bestandteil eines Ausdrucks** im aufrufenden Programm
 - Alle Parameter sind **Eingabeparameter**
 - Beispiele: result = cMin * WeightedAverage (timeSeries, first, last);
if (signal.isRed()) {engine.Stop(); }
- **Sonderfall:** Unterprogramm ist **inhaltlich** eine **Anweisung**, aber **syntaktisch** eine **Funktion**, die als Funktionswert einen Status zurückgibt
 - Wie ein Anweisungsunterprogramm behandeln
 - Beispiel: done = OpenFile ("coefficients.dat");

Globale Variablen

- **Globale Variablen** können für die **Kommunikation** zwischen **Aufrufer** und **Unterprogramm** herangezogen werden
- Variablen werden so deklariert, dass sie sowohl für den Aufrufer als auch im Unterprogramm **sichtbar** sind
 - **public** in C++/Java
 - **GLOBAL** in Cobol
- **Vorteil**
 - Parameterübergabe entfällt → **Effizienz**
- **Nachteile**
 - Namensersetzung nicht möglich → **weniger flexibel**
 - Bewirkt in der Regel **starke Kopplung**

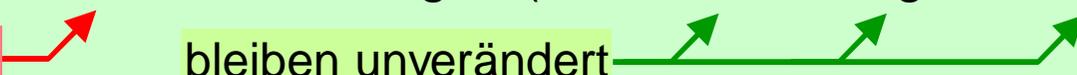
Kopplung und Nebenwirkungen minimieren

- So wenig Daten wie möglich übergeben
- Nur die benötigten Felder statt ganzer Strukturen übergeben
- Lokale Daten im Unterprogramm kapseln ⇨ von außen nicht sichtbar
- In objektorientierten Sprachen nur das Zielobjekt einer Methode verändern, als Parameter übergebene Objekte unverändert lassen

Beispiel: artikelStamm.Hinzufuegen (neuerArtikel, lager, kategorie);

wird verändert

bleiben unverändert



Kopplung und Nebenwirkungen minimieren –2

- Funktionen, welche vom Charakter her Prozeduren sind (d.h. als Funktionswert nur einen Status liefern) nicht in Ausdrücken aufrufen

Beispiel: Schlecht: `if (OpenFile("param.dat")) ReadSettings (); else ... ;`

Besser: `done = OpenFile("param.dat");
if (done) { ReadSettings (); }
else ...;`

Mini-Übung 2.5

Beurteilen Sie die Codequalität in nachfolgendem Programmfragment

- Informationsaustausch zwischen Startup() und seinem Aufrufer
- Implementierung von Startup()

```
public class Betriebsart {  
    public static boolean online = false;  
  
    // Führt das System aus dem offline Mode hoch in den online Mode  
    public static boolean Startup() {  
        if ( CheckSensors() & InitializeDatabase() && OpenLog () )  
            return (online = true); else return false;  
    } ...  
}  
...  
done = Betriebsart.Startup();  
if (Betriebsart.online) { ...
```

Länge von Unterprogrammen

- **Nicht zu lange** Unterprogramme schreiben; die Übersicht leidet
- **Faustregel:** nicht mehr als 200 Codezeilen (ohne Leerzeilen und Kommentare)
- Methoden in objektorientierten Programmen sind tendenziell **kürzer**
- Aber keine sturen Regeln, das Hauptkriterium ist **funktionale Kohäsion**
- Bei Bedarf Unterprogramme **schachteln**
- Sind Unterprogramme mit 1-3 Codezeilen sinnvoll?
 - Verbessern die Lesbarkeit, wenn oft benötigt
 - Beispiel:

```
public static final double MM_PER_INCH = 25.4;  
// Konvertiert Punkte bei gegebener Auflösung in Millimeter  
static double PointsToMM (int points, int resolution) {  
    return ((double)points/((double)resolution*MM_PER_INCH);  
}
```

Merkmale Unterprogramme

- Unterprogramme sind ein zentrales Mittel zum Aufbau lesbarer und wartbarer Programme
- Gute Unterprogramme haben hohe Kohäsion und geringe Kopplung
- Daten und Algorithmen eines Unterprogramms so weit wie möglich kapseln
- Daten primär über Parameter austauschen, nur in zwingenden Fällen über globale Variablen

2.1 Einführung

2.2 Namengebung

2.3 Datendefinitionen

2.4 Ablaufkonstrukte

2.5 Unterprogramme

2.6 Optimierung

2.7 Dokumentation

Drei Grundregeln für die Optimierung

Regeln von Jackson:

- 1. Tu es nicht!
- 2. Wenn du es dennoch tust oder tun musst, dann tu es später!

Ergänzende Regel (Glinz):

- 3. Tu es vorher! (Erst denken, dann codieren)

Vorgehen

- Optimieren ist aufwendig, darum
 - Sich das Optimieren des Codes ersparen durch Wahl guter Datenstrukturen und effizienter Algorithmen (Regel 3)
 - Code nie auf Verdacht optimieren (Regel 1)
- Sondern (Regel 2)
 - Zuerst messen
 - Die Flaschenhälse erkennen
 - Falls nötig, durch gezielte lokale Optimierung die Flaschenhälse beseitigen

2.1 Einführung

2.2 Namengebung

2.3 Datendefinitionen

2.4 Ablaufkonstrukte

2.5 Unterprogramme

2.6 Optimierung

2.7 Dokumentation

Dokumentation: Was und warum

- Qualität und Umfang der Dokumentation haben entscheidenden Einfluss auf die Lesbarkeit und Wartbarkeit eines Programms
- ⇒ Programmdokumentation ist **nicht einfach „Kommentar“**
- Jedes Stück Code ist dokumentiert
 - **Verwaltungsdokumentation** (Autor, Datum,...)
 - **Schnittstellendokumentation** (Voraussetzungen, Ergebniszusicherungen, ...)
 - **Deklarationsdokumentation** (Bedeutung von Konstanten und Variablen)
 - **Ablaufdokumentation** (Verdeutlichen des Algorithmus)
 - **Strukturdokumentation** (Statischer Aufbau des Programms; typisch durch Einrücken und Zwischentitel)

Richtig dokumentieren: Was ist hier falsch?

- `if (x > 0) y = sin(x)/x; // negative Werte dürfen nicht bearbeitet werden`
- `u = 2*PI*r; // u = 2pr`
`i++; // i inkrementieren`
- `WHILE (*t++); /* Ende des Strings suchen. t zeigt danach auf das Byte nach dem String-Terminator */`
- `i := 0; j := 0; c := 0;`
`REPEAT IF x[i] = " " THEN INC (c); ELSE y[j] := x[i]; INC (j); END;`
`INC (i); UNTIL i = n;`
- Auftragsnummern-Vergabe SECTION.
 - * Bildet Auftragsnummer aus Jahr und laufender Nummer *
 - * Achtung: funktioniert nur bei weniger als 1000 Aufträgen pro Jahr! *

COMPUTE Auftrags-Nr = Jahr * 1000 + Lfd-Nummer.

Regeln für das Dokumentieren

- Dokumentation und Code müssen **konsistent** sein
- Kein **Nachbeten** des Codes
- **Schlechten Code** und **Tricks** nicht dokumentieren, sondern **neu schreiben**
- Programmstruktur durch **Einrücken** dokumentieren
- **Geeignete Namen** wählen
- **Codierrichtlinien** beachten
- Falscher Code wird durch ausführliche Dokumentation **nicht richtig**
- Schlechter Code wird durch Dokumentation **nicht besser**
- Nicht **überdokumentieren**

Mini-Übung 2.6

Wo ist das Problem bei diesem Codestück aus einem Cobol-Lehrbuch?

...

B400.

```
DISPLAY (24, 1) 'WEITERE BERECHNUNGEN (J/N) : '.
```

```
ACCEPT (24, 40) S-WEITER WITH AUTO-SKIP.
```

```
IF S-WEITER = 'J'           * > Nur Großbuchstaben  
    GO TO B100              * > werden berücksichtigt
```

```
END-IF.
```

```
IF S-WEITER = 'N'
```

```
    GO TO B900
```

```
END-IF.
```

**** falsche Eingabe:

```
GO TO B400.
```

B900.

```
STOP RUN.
```

Was ist gute Dokumentation?

- Gute Dokumentation beschreibt, was **nicht** im Code steht:
 - **Intention** des **Programms**
 - **Intention** für die Verwendung bestimmter **Daten**
 - Getroffene **Annahmen**
 - **Semantik (Bedeutung)** von Schnittstellen
- Gute Dokumentation **gliedert** und **erläutert** den Aufbau eines Programms, wo dies der Code nicht ausreichend tut
 - **Untertitel** für Abschnitte und Blöcke
 - **Hinweise** auf verwendete Algorithmen
 - **Erläuterung** schwierig zu verstehender Konstrukte (die nicht einfacher programmierbar sind)
 - **Hinweise**, dass ein Codestück aus **Optimierungsgründen** gerade so programmiert worden ist

Modifikation dokumentierter Programme

- Sich weder auf Dokumentation noch auf den Code **blind verlassen** – beide können falsch sein
- Bei Änderungen im Code Dokumentation immer **konsistent mitändern**
- Änderungen
 - entweder im angetroffenen Codier- und Dokumentationsstil vornehmen
 - oder den Codier- und Dokumentationsstil einer ganzen Komponente vollständig an einen neuen Stil anpassen

Merkmale Dokumentation

- Undokumentierter Code ist weder lesbar noch wartbar
- Dokumentation liefert zusätzliche Informationen, sie betet den Code nicht nach
- Dokumentation und Code müssen widerspruchsfrei sein
- Schlechten und/oder falschen Code nicht dokumentieren, sondern neu schreiben
- Nicht überdokumentieren

Literatur

Berner, S., S. Joos, M. Glinz (1997). Entwicklungsrichtlinien für die Programmiersprache Java. *Informatik/Informatique* **4**, 3 (Jun 1997). 8-11.

Berner, S., M. Glinz, S. Joos, J. Ryser, S. Schett (2001). *Java Entwicklungsrichtlinien*. Version 2.0.1 (März 2001). Institut für Informatik, Universität Zürich.

Böhm, C. G. Jacopini (1966). Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules. *Communications of the ACM* **9**, 5 (May 1966). 366-371.

Dijkstra, E.W. (1968). Go To Statement Considered Harmful. *Communications of the ACM* **11**, 3 (March 1968). 147-148.

Keller, D. (1990). A Guide to Natural Naming. *SIGPLAN Notices* **25**, 5 (May 1990). 95-102.

Kernighan, B.W., P.J. Plauger (1978). *The Elements of Programming Style*. New York: McGraw-Hill

McConnell, S. (1993). *Code Complete: A Practical Handbook of Software Construction*. Redmond: Microsoft Press.

Mössenböck, H. (2001). *Sprechen Sie Java? : Eine Einführung in das systematische Programmieren*. Heidelberg: dpunkt-Verlag.

Vermeulen, A., S.W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur, P. Thompson (2000). *The Elements of Java Style*. Cambridge: Cambridge University Press.