„Qualitätsmerkmale“ von Programmen

Ziel der Softwareproduktion: gute Programme mit geringem Aufwand

Was ist ein „gutes Programm“? — Einfacher zu beantworten ist, welche Mängel in den meisten Programmen auftreten:

- Autorenabhängigkeit (jeder hat seinen persönlichen Stil), deshalb wartungsunfreundlich
- Bandwurmtechnologie (oft mit „Darmverschlingung“), Spezialanfertigung für eine einzige Aufgabe
- kaum flexibel, nicht nachnutzbar
- unzureichend dokumentiert, Details sind nicht zu erkennen, ohne das Programm als Ganzes zu verstehen

Merkmale für die Qualität von Programmen
(die sich leider teilweise widersprechen)

(1) Richtigkeit

Der spezifizierte Funktionsumfang soll vollständig und korrekt realisiert werden (nicht mehr und nicht weniger), z.B. soll eine Funktion `sqrt(x)` für $x > 0$ den Wert $\sqrt{x}$ zurückgeben (nicht etwa den Wert der Variablen `x` verändern).

(2) Zuverlässigkeit

Aus korrekten Eingangsdaten sollen korrekte Ausgangsdaten produziert werden, wenn man eine korrekte Bedienung und die korrekte Funktion der Maschine (incl. Betriebssystem) voraussetzt.

(3) Robustheit

Das Programm soll auf Fehler des Nutzers (Eingabe, Bedienung) oder der Maschine (E/A-Fehler, ...) „definiert“ reagieren:

- Fehlersituation eindeutig erkennen
- Fehler beheben, falls möglich (und sinnvoll)
- eindeutige Mitteilung (Art und Ort des Fehlers), nicht etwa:
  ```
  print *,'ERROR'
  stop
  ```
- ein Fehlerindikator (Integer-Variable) als Parameter sollte die Fehlersituation ans rufende Programm zurückmelden und diesem die weitere Behandlung überlassen

(4) Korrigier- und Änderbarkeit

Fehler werden oft erst nach längerer Testphase entdeckt, deshalb sollte man auch dann noch in der Lage sein, diese zu beheben.

- Kommentare verwenden (von Anfang an!)
- Namen für Variable und Unterprogramme „sinnvoll“ wählen, z.B. `IER, EPS, INPUT`, aber nicht notwendig ist ein Name wie `Index_der_Zeile`, wo ein `i` auch reicht.

- modularer Programmaufbau (Baukastenprinzip): kurze, übersichtliche Unterprogramme (leichter zu verifizieren), keine „Spaghetti-Programme“ (zieht man links, bewegt es sich rechts),

(5) Portabilität

Quelltext soll ohne (wesentliche) Änderungen auf andere Rechner oder Betriebssysteme übertragbar sein

- (wenige) definierte Schnittstellen zur „Umwelt“
- systemabhängige Konstanten nur an einer (zentralen) Stelle im Programm festlegen.
- Programmiertricks vermeiden, bzw. bei bewusster Ausnutzung einer hardwarenahen Eigenschaft durch Kommentar darauf hinweisen.

  Kritisch sind z.B. `EQUIVALENCE`-Anweisungen oder gerätespezifische Programmierung wie Escape-Steuerfolgen für spezielle Bildschirmausgaben.

(6) Effektivität

Effektive Nutzung aller Ressourcen, das sind insbesondere

- Speicherplatz, sowohl intern (RAM) als auch extern (Dateisystem)
- Rechenzeit: CPU-Zeit und Gesamtlaufzeit

Schwerpunkt der Optimierung sind die am häufigsten genutzten Programmteile: elementare Unterprogramme, die innersten bei geschachtelten Schleifen.

Verwendung speziell für die Hardware optimierter (Standard-) Routinen wie BLAS (Basic Linear Algebra Software) für Vektor- und Matrix-Operationen

Weitere Tipps zur Konzeption von Programmen:

- Ein Algorithmus wird immer als Unterprogramm (nicht als Hauptprogramm) realisiert; mit der Parameterliste wird die felxible Nutzung des Programms ermöglicht.
- Man prüfe die Aufgabenstellung eines Unterprogramms auf Verallgemeinerungswürdigkeit (kann durch Modifikation eines Parameters eine andere ähnliche Aufgabe gelöst werden?)
- Unterprogramme sollten frei von `READ`- und `WRITE`-Anweisungen sein (außer spezielle Ein- oder Ausgabe-Unterprogramme)
- Protokollierung des Rechenablaufs (`write ...`) mit steuerbarem Umfang realisieren

  ```
  IF(IPROT .GT. 2) call outvec(N,X,'Vektor x')
  ```

- Die Optimierung durch den Compiler (Flags `-O2` u.ä.) liefert meist sehr gute Ergebnisse (kann aber auch eine zusätzliche Fehlerquelle sein!)

  Warum wird optimiert? – Rechenzeit- und/oder Speicherplatzminimierung

  Was wird optimiert? – arithmetische Ausdrücke, `DO`-Schleifen

  Wie wird optimiert?

  – Das Programm wird als Ganzes übersetzt, nicht jede Anweisung einzeln

- Einsparung von Speicherzugriffen durch effektive Verwendung von CPU-Registern als Zwischenspeicher
- Indexrechnung in Schleifen wird ersetzt durch entsprechende Rechnung mit den Speicheradressen (statt den Index um 1 zu erhöhen wird die Adresse um 4 bzw. 8 erhöht)
- Ausklammern konstanter (Teil-)Ausdrücke, auch über mehrere Anweisungen hinweg und aus Schleifen heraus.

  Das kann (bei fehlerhaften Programmen) zu unterschiedlichem Laufzeit-Fehlerverhalten mit oder ohne Optimierung führen. Ein Beispiel (das man so nie programmieren würde):

Quelltext	nach der Optimierung
`  DO 1 I=1,5`	`_H1_=A*B`
`  X(I)=A*B`	`_H2_=K/N`
`  IF(N .NE. 0) J(I)=K/N`	`DO 1 I=1,5`
`1 CONTINUE`	`  X(I)=_H1_`
	`  IF(N .NE. 0) J(I)=_H2_`
	`1 CONTINUE`

  Falls nun `A*B` einen Überlauf erzeugt und `N=0`, dann erhält man in der nichtoptimierten Version 5 Überläufe. im optimierten Programm nur einen Überlauf, aber auch eine Division durch Null. Das Ergebnis ist aber am Ende gleich.

Was kann der Programmierer selbst zur Optimierung beitragen

- `COMMON`-Blöcke und `EQUIVALENCE`-Anweisungen behindern die Optimierung
- ein klar strukturiertes Programm erleichtert die Optimierung (natürlich ist damit die inhaltliche Struktur gemeint; die „optische“ Strukturierung des Quelltextes ist wichtig für obigen Punkt (4))
- komplizierte Indexausdrücke vermeiden; Standardfunktionen verwenden, wenn es sich anbietet (sie sind immer die bessere Alternative) wie z.B.

```
  DO 1 I=1,40
1 F(2*I-1) = F(2*I-1)+A(J)**0.5
```

  besser so:

```
  A1=SQRT(A(J))
  DO 1 I=1,79,2
  F(I)=F(I)+A1
1 CONTINUE
```

- arithmetische Ausdrücke:

  keine gemischten Datentypen in einem Ausdruck (Konvertierung kann Zeit und Genauigkeit kosten)

  Potenzrechnung: wenn möglich, `A**4` statt `A**4.0` verwenden;
  ersteres wird als `(A**2)**2` durch 2 Multiplikationen berechnet, letzteres mit Logarithmen (ungenauer).