Zur Entwicklung eines guten Programmierstils

Jörg Fliege
Lehrstuhl für Wissenschaftliches Rechnen
Fachbereich Mathematik
Universität Dortmund
44221 Dortmund
Email fliege@math.uni-dortmund.de
Tel. 0231 / 755-5415
Fax 0231 / 755-5416

1. Dezember 2003

Zusammenfassung

Einige aus der Literatur bekannte stilistische Techniken zur Erstellung verständlichen, wartbaren Programmcodes werden vorgestellt und diskutiert.

1 Einleitung

Jedes Programm, das man für mehr als ein paar Stunden benötigt, muß dokumentiert werden. Das Dokumentieren des Codes darf nicht verschoben werden! Programme, die heute völlig klar und leicht lesbar erscheinen, können schon morgen unklar sein. Wenn man selbst bereits Schwierigkeiten hat, den eigenen Code zu lesen, wie lange brauchen dann andere Leute, um solchen Code zu verstehen?

Stil ist lediglich eine einfache Möglichkeit, um Programmcode leicht lesbar und leichtverständlich zu halten. Eine gute Organisation des Codes und sinnvolle Variablenbezeichnungen erhöhen die Lesbarkeit, großzügige Verwendung von Kommentaren erläutern die Logik des Programms (Was geschieht wo und warum?).

Eine der einfachsten Möglichkeiten, Stilfragen zu klären, besteht in der Betrachtung eines fragwürdigen Programms. Wir betrachten den folgenden *stilistisch schlechten* MATLAB-Code.

```
1   function m=s
2   m=0; seg=[6 2 5 5 4 5 6 3 7 6];
3   for d1=0:1 for d2=0:9 for d3=0:5 for d4=0:9
4   if ~((d1==0) & (d2==0)) & ~((d1==1) & (d2>2))
5   if (d1==0)
6   ts=seg(d2+1)+seg(d3+1)+seg(d4+1);td=d2+d3+d4;
7   if (ts==td) m=m+1; % disp([ num2str(d2) ':' num2str(d3) num2str(d4) ]);
8   end
9   else
10  ts=seg(d1+1)+seg(d2+1)+seg(d3+1)+seg(d4+1);td=d1+d2+d3+d4;
11  if (ts==td)
12  m=m+1; % disp([ num2str(d1) num2str(d2) ':' num2str(d3) num2str(d4) ]);
13  end; end; end; end; end; end; end
```

Was macht dieser Code? Wie lang braucht man, um jemanden zu erklären, wie er funktioniert? Wie lange braucht man, um den Code zu verstehen? Wenn dieser Code fehlerhaft ist, wie lange braucht man dann, um die Fehler zu korrigieren? Was geschieht, wenn man selbst diesen Code geschrieben hat und eine andere Person muß den Fehler finden? Was geschieht, wenn man selbst den Fehler finden muß, obwohl man den Code nicht selbst geschrieben hat? (Eine typische Situation beim Schreiben von Anwendungssoftware.)

Diese Gedanken führen zu folgendem **stilistischen Grundprinzip**.

> *Programmiere und dokumentiere so, wie Du es Dir wünscht, daß andere es täten.*

2 Einrücken

Das obige Programm ist nicht konsistent eingerückt, genauer: der Code ist überhaupt nicht eingerückt. Es gibt Dutzende verschiedener Einrückungs-Stile, daher sollen hier nur einige der wichtigsten Ideen genannt werden:

1. Innerhalb eines Programms soll der gleiche Stil zum Einrücken verwendet werden.
2. Code innerhalb eines Blocks (z. B. einer `for`-Schleife oder einer `while`-Anweisung soll eingerückt sein.
3. Ein Codeblock, der in einen anderen Codeblock verschachtelt ist, soll tiefer eingerückt sein als der umgebende Block.

4. Zu tief eingerückte „Treppenstrukturen“ sollten vermieden werden, sie behindern die Lesbarkeit des Codes. (Falls Code zu tief eingerückt wird, kann z. B. die Einrücktiefe verringert werden, etwa von acht Leerzeichen auf vier o. ä..)

Man beachte, daß gute Editoren diese Ideen automatisch berücksichtigen! Werden diese Regeln auf den obigen Code angewandt, ergibt sich etwa folgendes Bild.

```
1   function m = s
2
3   m = 0;
4   seg = [ 6 2 5 5 4 5 6 3 7 6 ];
5
6   for d1=0:1
7       for d2=0:9
8           for d3=0:5
9               for d4=0:9
10                  if ~((d1==0) & (d2==0)) & ~((d1==1) & (d2>2))
11                      if (d1 == 0)
12                          ts = seg(d2+1) + seg(d3+1) + seg(d4+1);
13                          td   = d2 + d3 + d4;
14                          if (ts == td)
15                              m = m + 1;
16                              % disp([ num2str(d2) ':' num2str(d3) num2str(d4) ]);
17                          end
18                      else
19                          ts = seg(d1+1) + seg(d2+1) + seg(d3+1) + seg(d4+1);
20                          td   = d1 + d2 + d3 + d4;
21                          if (ts == td)
22                              m = m + 1;
23                              % disp([ num2str(d1) num2str(d2) ':' ...
24                              %      num2str(d3) num2str(d4) ]);
25                          end
26                      end
27                  end
28              end
29          end
30      end
31  end
```

Hier wurden außerdem Kommandos, die auf einer Zeile auftraten, auf mehrere Zeilen verteilt. Des weiteren wurden um der besseren Lesbarkeit willen Leerzeichen zwischen Operatoren (+) und Zuweisungen (=) eingefügt.

3 Einteilen

Unglücklicherweise kann man jetzt kaum noch den Code innerhalb eines Editors auf einer Bildschirmseite betrachten, ohne den Text von links nach rechts zu scrollen. Eine mögliche Abhilfe wäre eine Verringerung der Einrücktiefe. Eine andere Möglichkeit besteht im Auslagern bestimmter Codeblöcke in Unterfunktionen. Diese Möglichkeit sollte immer angewandt werden, wenn erkennbar ist, daß ein bestimmter Teil des Codes noch anderweitig verwertbar sein könnte oder die Geschehnisse im entsprechenden Codeblock nichts mit dem umgebenden Code zu tun haben. Im obigen Beispiel ist außerdem der Quelltext der Funktion beinahe so lang wie eine Bildschirmseite. Dies wird häufig als klares Zeichen gedeutet, den Code in kleinere Abschnitte sinnvoll zu unterteilen. Das Auftreten von Unterprogrammen oder -funktionen kann bei weniger intelligenten Compilern oder Interpretern die Laufzeit des Gesamtprogramms minimal erhöhen. Auf diesen Einwand gegen Codeaufteilung aus Effizienzgründen soll hier nicht näher eingegangen werden.

```
1   function m = s
2
3   m = 0;
4
5   for d1=0:1
6       for d2=0:9
7           for d3=0:5
8               for d4=0:9
9                   m = m + count(d1, d2, d3, d4);
10              end
11          end
12      end
13  end
14
15  % disp([ 'The number of times displayed is ' num2str(matches) '.' ]);
16
17  function m = count(d1, d2, d3, d4)
18
19  seg = [ 6 2 5 5 4 5 6 3 7 6 ];
20  m = 0;
21
22  if ~((d1==0) & (d2==0)) & ~((d1==1) & (d2>2))
23      if (d1 == 0)
24          ts = seg(d2+1) + seg(d3+1) + seg(d4+1);
25          td   = d2 + d3 + d4;
26          if (ts == td)
27              m = m + 1;
```

```
28              % disp([ num2str(d2) ':' num2str(d3) num2str(d4) ]);
29          end
30      else
31          ts = seg(d1+1) + seg(d2+1) + seg(d3+1) + seg(d4+1);
32          td   = d1 + d2 + d3 + d4;
33          if (ts == td)
34              m = m + 1;
35              % disp([ num2str(d1) num2str(d2) ':' num2str(d3) num2str(d4) ]);
36          end
37      end
38  end
```

Die Unterfunktion `count` strukturiert den Code jetzt noch besser. Die Variable `seg` ist jetzt eine lokale Variable innerhalb der Funktion `count`, da sie keinen weiteren Einfluß auf das Hauptprogramm hat.

Außerdem sieht man leicht, daß die Kommandos `m = m + 1` jeweils durch `m = 1` ersetzt werden können. Dies soll im folgenden geschehen.

4 Benennen

Eine starkes Hindernis beim Lesen des Codes sind jetzt noch die kryptischen Variablennamen. (Die folgenden Bemerkungen beziehen sich auch auf Namen von Funktionen und Unterfunktionen.) Jedes Objekt und jede Variable in einem Programm sollte einen Namen tragen, der dem Leser Informationen über Sinn und (Einsatz-)Zweck mitteilt. Ausnahmen gelten höchstens für generische Schleifenindizes für `for`-Schleifen, die ohne weiteres `i`, `j` und `k` heißen dürfen. In diesem Zusammenhang kann dann der obere Laufindex `n` bzw. `m` heißen. Für alle anderen Variablen sollten die folgenden Prinzipien gelten:

1. Namen sollten einen Sinn tragen. Wenn eine Variable die Wochenarbeitszeit eines Angestellten in Stunden trägt, so ist `hours` ein sinnvoller Name. Dagegen ist `hours_worked_per_week` zwar noch deutlicher, doch machen derartig lange Variablennamen den Code wieder schlechter lesbar. (Schon die Addition von 1 zu einer Variablen mit so langem Namen nimmt eine ganze Zeile in Anspruch!) Ein vernünftiger Kompromiß wäre etwa `hours_per_week`.

2. Innerhalb eines Variablennamens erhöhen Unterstriche die Lesbarkeit: man vergleiche `hoursperweek` mit `hours_per_week`.

3. Zu jeder Variable muß zu Beginn des Codes ein Kommentar existieren, der ihre Bedeutung und ihre Verwendung erläutert. Dies hat nichts mit der Pflicht zu tun,

gute Namen zu wählen: beim Lesen von Code springt man nicht gern zwischen Kommentarblock und Verwendung der Variablen hin und her.

4. Abkürzungen können Code lesbarer machen: `hrs_per_week` ist eine sinnvolle Abkürzung (Die sich, wie häufig, durch Weglassen der Vokale ergibt.). Andererseits darf man Abkürzungen nicht übertreiben: `hrs_p_wk` ist zu unklar, um verwendet zu werden. Nur Standardabkürzungen sollten verwendet werden!

5. Variablen sollten nicht innerhalb einer Routine für andere Zwecke wiederverwendet werden. Intelligente Compiler und Interpreter benutzen nicht mehr benötigten Speicherplatz sowieso wieder, daher kann mit solchen Techniken kein Speicherplatz eingespart werden. Stattdessen wird der Code schlechter lesbar, die Fehlersuche und eine eventuelle Portierung wird in erheblichem Maße erschwert.

6. Bezeichnungsregeln für Variablen müssen ohne Ausnahmen gelten. Haben etwa alle Zeiger außer einem am Ende ihrer Bezeichnung ein `_p`, wird man sich fragen, was an dieser Ausnahme so besonders ist.

Bei Programmiersprachen mit engen Beschränkungen für die Länge einer Variablenbezeichung (z. B. FORTRAN77 mit 6 Zeichen) ist der in Punkt 3 angesprochene Kommentar natürlich umso wichtiger! In einem solchen Fall sollte man sich an stringente Regeln für jeden einzelnen Buchstaben in der Variablenbezeichnung halten. Ist etwa der erste Buchstabe ein `Z`, so ist die Variable eine komplexe Zahl, ist dagegen der erste Buchstabe ein `P`, wird die Variable als Zeiger verwendet usw. Diese Regeln müssen natürlich ebenfalls durch einen Kommentar erläutert werden.

Sinnvolle Variablennamen ergeben sich leider erst dann, wenn man denn Sinn des Programms versteht. Unterstellt man ein solches Verständnis beim Beispielprogramm, ergibt sich etwa der folgende Code.

```
 1  function matches = sumdigits
 2
 3  matches = 0;
 4
 5  for hour1=0:1
 6      for hour2=0:9
 7          for tens=0:5
 8              for oness=0:9
 9                  matches = matches + count_segments(hour1, hour2, tens, oness);
10              end
11          end
12      end
13  end
14
```

```
15  % disp([ 'The number of times displayed is ' num2str(matches) '.' ]);
16
17  function m = count_segments(hour1, hour2, tens, oness)
18
19  segments = [ 6 2 5 5 4 5 6 3 7 6 ];
20  m = 0;
21
22  if ~((hour1==0) & (hour2==0)) & ~((hour1==1) & (hour2>2))
23      if (hour1 == 0)
24          total_segments =   segments(hour2+1) + segments(tens+1) ...
25                           + segments(oness+1);
26          total_digits = hour2 + tens + oness;
27          if (total_segments == total_digits)
28              m = 1;
29              % disp([ num2str(hour2) ':' num2str(tens) num2str(oness) ]);
30          end
31      else
32          total_segments =   segments(hour1+1) + segments(hour2+1) ...
33                           + segments(tens+1) + segments(oness+1);
34          total_digits = hour1 + hour2 + tens + oness;
35          if (total_segments == total_digits)
36              m = 1;
37              % disp([ num2str(hour1) num2str(hour2) ':' ...
38              %      num2str(tens) num2str(oness) ]);
39          end
40      end
41  end
```

Die Namen sind immer noch relativ unverständlich, aber schon jetzt muß Code teilweise auf mehrere Zeilen umgebrochen werden. (Zeile 24–25, 32–33 und 37–38.) Dies ist der Preis für längere Variablennamen. Noch immer fehlt eine Dokumentation der Variablen. Dies soll als nächstes eingefügt werden, zusammen mit kurzen Kommentarblöcken über den Sinn verschiedener `if`-Abfragen.

```
1  function matches = sumdigits
2
3  % Variables: matches: output variable
4  %            hour1:   the first (leftmost) digit in the hour two-digit pair
5  %            hour2:   the second (rightmost) digit in the hour two-digit pair
6  %            tens:    the ten's digit in the minute two-digit pair
7  %            oness:   the one's digit in the minute two-digit pair,
8  %                     spelled in this way because "one" is a Matlab function.
```

```
 9
10
11  matches = 0; % count the number of times the sums match
12
13  for hour1=0:1
14      for hour2=0:9
15          for tens=0:5
16              for oness=0:9
17                  matches = matches + count_segments(hour1, hour2, tens, oness);
18              end % for oness
19          end % for tens
20      end % for hour2
21  end % for hour1
22
23  % disp([ 'The number of times displayed is ' num2str(matches) '.' ]);
24
25  function m = count_segments(hour1, hour2, tens, oness)
26
27  % Variables: segments:       array with the number of segments needed
28  %                            to display each digit.
29  %            total segments: number of segments needed to display each digit.
30  %            total_digits:   sum of digits in the time value.
31
32  segments = [ 6 2 5 5 4 5 6 3 7 6 ];
33  m = 0;
34
35  if ~((hour1==0) & (hour2==0)) & ~((hour1==1) & (hour2>2))
36      % We do not want to consider times that start with '00' (like 00:35) or
37      % anything with hours above 12 (like 16:14).
38      if (hour1 == 0)
39          % If the time is between 12:59 and 10:00, the leftmost digit is 0.
40          % A clock does not display this zero, so we do not count its segments.
41          total_segments = segments(hour2+1) + segments(tens+1) ...
42                           + segments(oness+1);
43          total_digits   = hour2 + tens + oness;
44          if (total_segments == total_digits)
45              m = 1;
46              % disp([ num2str(hour2) ':' num2str(tens) num2str(oness) ]);
47          end
48      else % inner if
49          % Now we do count the leftmost hour digit.
50          total_segments = segments(hour1+1) + segments(hour2+1) ...
51                           + segments(tens+1) + segments(oness+1);
```

```
52          total_digits   = hour1 + hour2 + tens + oness;
53          if (total_segments == total_digits)
54              m = 1;
55              % disp([ num2str(hour1) num2str(hour2) ':' ...
56              %        num2str(tens) num2str(oness) ]);
57          end
58      end % inner if
59  end % outer if
```

Alle Kommentare und alle Variablennamen sind in Englisch. Das ist selbstverständlich. Dank elektronischer Kommunikation kann beliebiger Code praktisch überall auf der Erde (und darüber hinaus) verwendet werden und sollte auch überall verstanden werden können. Internationale Programmierteams sind der Normalfall, Englisch ist für Kommentare de facto der Standard.

5 Dokumentieren

Mit Dokumentieren ist die interne Dokumentation des Programmcodes gemeint, nicht evtl. mitgelieferte Handbücher o. ä. Interne Dokumentation erläutert weniger die Handhabe und Verwendung eines Programms (auch wenn dies sicher kein Nachteil ist), sondern die Arbeitsweise und interne Logik des verwendeten Algorithmus. Idealerweise kann man von der internen Dokumentation auf die Handhabe und Verwendung des Programms schließen. Natürlich muß *nicht* jede Zeile Code dokumentiert werden. Triviale Operationen wie `m = 1` müssen *nicht* durch einen Kommentar `% Set m equal to one.` erläutert werden.

Das Beispielprogramm besitzt noch keine Dokumentation. Offensichtlich ist Dokumentation wertvoll. (Oder ist Sinn, Zweck und Arbeitsweise des Programms schon klar?)

Einige Prinzipien für sinnvolle Kommentare:

1. Jede Funktion benötigt einen Kommentar mit den folgenden Angaben:
   (a) Name der Funktion,
   (b) Einsatzzweck,
   (c) eine Erläuterung der Ein- und Ausgabeparameter,
   (d) Autor der Routine (mit Emailadresse und evtl. weiteren Angaben zur Erreichbarkeit),
   (e) der Name der Person, die die letzte Änderung durchgeführt hat (mit Emailadresse und evtl. weiteren Angaben zur Erreichbarkeit),

(f) das Datum (und evtl. die Uhrzeit) der letzten Änderung,

(g) die aktuelle Versionsnummer,

(h) einer Liste von Änderungen (sofern erfolgt),

(i) einer Liste bekannter Fehler,

(j) eine Liste verwendeter Routinen,

(k) evtl. Literaturangaben zu verwendeten Algorithmen.

2. Komplexe Codeabschnitte sollten mit einem Block von Kommentarzeilen beginnen, in dem erläutert wird, was geschieht.

3. Kommentare sollten prinzipiell eher erläutern, *was* geschieht. *Wie* etwas geschieht, beschreibt der Code selbst meist besser als jeder Kommentar.

4. Kommentare sollten geschrieben werden, *während* der Code programmiert wird. **Danach ist es zu spät!**

Die Anwendung dieser Prinzipien führt z. B. zu folgendem Beispielprogramm.

```
1   function matches = sumdigits
2   %
3   % SUMDIGITS counts the matches in a common digital display.
4   %
5   % Input:  none
6   % Output: matches (integer), the number of matches found
7   %
8   %      A common digital display clock, such as an alarm clock or
9   %      a digital wristwatch, creates numbers by lighting segments in a
10  %      standard 7-segment display.  This program assumes that the digits
11  %      look like this:
12  %                 _        _    _         _    _    _    _    _
13  %               | |   |   _|   _|  |_|  |_   |_     |  |_|  |_|
14  %               |_|   |  |_    _|    |   _|  |_|    |  |_|   _|
15  %
16  %      Thus, 6 segments are needed to display the digit 0, for example.
17  %      The program creates all legal twelve-hour times (plus some illegal
18  %      ones that are logically eliminated from consideration) and for each
19  %      determines if the number of segments in the displayed time equals
20  %      the sum of the digits in the display.  For example, 7:21 requires
21  %      10 segments and the sum of the digits is also 10 (7+2+1).  All such
22  %      times can be output by the program (after uncommenting some
23  %      disp-statements below) and the total number of such times is determined.
24  %
```

```
25  %        Input:  No input (either from the keyboard or from a file) is
26  %              required by this program.
27  %
28  %       Output:  The times are displayed one per line to the standard output.
29  %              The number of times displayed is output at the end.
30  %
31  %    Algorithm:  The times are generated by a set of 4 nested FOR loops.
32  %              Times are composed of 4 digits, one loop per digit.  Legal
33  %              times can start with a 0 or a 1, but the 0 is never shown
34  %              on a clock.  Hours go from 01 through 12, so the second
35  %              digit of the hour value can range from 0 through 9, as can
36  %              the rightmost digit of the minute.  The left digit in the
37  %              minute value ranges from 0 through 5 only.  For each time
38  %              generated by the loops, the count_segments function
39  %              determines if the segment sum equals the digit sum; see
40  %              the internal documentation of count_segments for details
41  %              on its operation.
42  %
43
44  % To learn more about the code, uncomment the disp commands below.
45
46  % Author:  Joerg Fliege, porting the C version by McCann, 2000.
47  %          fliege@math.uni-dortmund.de
48  % Date:    November 14, 2003
49  % Version: 1.0
50  %
51  % Known bugs: none
52  % Functions used: count_segments (in this file)
53
54  % Variables: matches: output variable
55  %            hour1:   the first (leftmost) digit in the hour two-digit pair
56  %            hour2:   the second (rightmost) digit in the hour two-digit pair
57  %            tens:    the ten's digit in the minute two-digit pair
58  %            oness:   the one's digit in the minute two-digit pair,
59  %                     spelled in this way because "one" is a Matlab function.
60
61
62  matches = 0; % count the number of times the sums match
63
64  for hour1=0:1
65      for hour2=0:9
66          for tens=0:5
67              for oness=0:9
```

```
68                  matches = matches + count_segments(hour1, hour2, tens, oness);
69              end % for oness
70          end % for tens
71      end % for hour2
72  end % for hour1
73
74  % disp([ 'The number of times displayed is ' num2str(matches) '.' ]);
75
76  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
77
78  function m = count_segments(hour1, hour2, tens, oness)
79  %
80  %      count_segments computes the number of segments
81  %      a digital clock will need to display the time given by
82  %      the parameters. It then computes the sum of the digits
83  %      and compares the two totals.  If they match, the success
84  %      is recorded by incrementing the sum 'matches' and by
85  %      displaying the time.
86  %              The number of segments a digital clock uses to
87  %      display any of the ten numbers 0-9 is stored in the array
88  %      segments.  Here, the number of segments needed to
89  %      display a '0' is in position 1 of the array, and '1' is in
90  %      position 2, etc.
91  %
92  %  Input: hour1: In a two-digit hour value, this is the leftmost
93  %                digit.  Example:  In the time 12:34, hour1 would hold 1.
94  %         hour2: In a two-digit hour value, this is the rightmost
95  %                digit.  Example:  In the time 12:34, hour2 would hold 2.
96  %         tens:  In a two-digit minute value, this is the leftmost
97  %                digit.  Example:  In the time 12:34, tens would hold 3.
98  %         oness: In a two-digit minute value, this is the rightmost
99  %                digit.  Example:  In the time 12:34, ones would hold 4.
100 %                Spelled in this way because "one" is a Matlab function.
101 %
102 %  Output: m:    The sum of the times the number of segments equals the
103 %               sum of the digits.
104
105 % Variables: segments:       array with the number of segments needed
106 %                            to display each digit.
107 %            total segments: number of segments needed to display each digit.
108 %            total_digits:   sum of digits in the time value.
109
110 % Author:  Joerg Fliege, porting the C version by McCann, 2000.
```

```
111  %          fliege@math.uni-dortmund.de
112  % Date:    November 14, 2003
113  % Version: 1.0
114  %
115  % Known bugs: none
116  % Functions used: none
117
118
119  % Initialization:
120  segments = [ 6 2 5 5 4 5 6 3 7 6 ];
121  m = 0;
122
123  if ~((hour1==0) & (hour2==0)) & ~((hour1==1) & (hour2>2))
124      % We do not want to consider times that start with '00' (like 00:35) or
125      % anything with hours above 12 (like 16:14).
126      if (hour1 == 0)
127          % If the time is between 12:59 and 10:00, the leftmost digit is 0.
128          % A clock does not display this zero, so we do not count its segments.
129          total_segments = segments(hour2+1) + segments(tens+1) ..
130                           + segments(oness+1);
131          total_digits   = hour2 + tens + oness;
132          if (total_segments == total_digits)
133              m = 1;
134              % disp([ num2str(hour2) ':' num2str(tens) num2str(oness) ]);
135          end
136      else % inner if
137          % Now we do count the leftmost hour digit.
138          total_segments = segments(hour1+1) + segments(hour2+1) ..
139                           + segments(tens+1) + segments(oness+1);
140          total_digits   = hour1 + hour2 + tens + oness;
141          if (total_segments == total_digits)
142              m = 1;
143              % disp([ num2str(hour1) num2str(hour2) ':' ..
144              %        num2str(tens) num2str(oness) ]);
145          end
146      end % inner if
147  end % outer if
```

Kann man das Programm jetzt besser verstehen als zuvor?

Jetzt bestehen knapp zwei Drittel des gesamten Codes aus Kommentaren. Derartige Größenverhältnisse sind völlig normal.

Bei dieser Kommentierung wie auch bei der eigentlichen Programmierung setzte der Autor natürlich verschiedene persönliche stylistische Vorlieben ein —wie es bei jeder handwerklichen oder kreativen Tätigkeit geschieht. Auch kann dieses Programm sicherlich verbessert werden, so wie jedes andere auch.

6 Weitere Regeln

- Ein langsames, korrektes Programm ist besser als ein schnelles, falsches Programm.
- Ein schnelles, korrektes Programm ist besser als ein langsames, korrektes Programm.
- Ein Programm muß erst laufen, bevor man es effizienter machen darf.
- Die Programmstruktur muß feststehen, bevor die erste Zeile programmiert wird.
- Etwas schnell aber schlecht zu erledigen, damit es fertig wird, *scheint* produktiv zu sein. Sollten Autos so gebaut werden? Oder Fahrstühle? Oder Flugzeuge?
- Klarheit ist professionell, Unklarheit ist unprofessionell.
- Klarheit korreliert mit Qualität.
- Man darf eine Stilregel auch brechen, wenn man einen klaren Vorteil sieht.
- Schreibe klare, einfache Anweisungen; versuche nicht, zu schlau zu sein.
- Sage, was Du willst.
- Verwende Bibliotheken, erfinde nicht das Rad neu.
- Vermeide temporäre Variablen.
- Laß den Computer die Drecksarbeit machen.
- Klammern setzen vermeidet Unklarheiten.
- Vermeide unnötige `if-else`-Konstrukte.
- Benutze die guten Dinge in MATLAB, vermeide die schlechten.
- `if-else` ist kein Ersatz für ein logisches Konstrukt.
- Wenn man den Code beim Vorlesen am Telephon nicht versteht, ist er schlecht geschrieben.
- Mit `if-else` kann man klarmachen, daß genau eine Aktion von zwei möglichen durchgeführt wird.

- Programme müssen von vorne nach hinten gelesen werden können.
- Schreibe erst in einer einfachen Pseudo-Sprache, übersetze dann in richtigen Code.
- Verwende eine Datenstruktur, die das Programm vereinfacht.
- Höre niemals auf, wenn die erste Version funktioniert.
- Modularisiere. Verwende Unterprogramme.
- Jede Funktion sollte eine Sache gut machen.
- Jede Funktion muß irgendetwas verstecken.
- Die Daten strukturieren das Programm. Nicht umgekehrt.
- Schlechter Code wird nicht verbessert. Schlechter Code wird weggeworfen und neu geschrieben.
- Ein großes Programm muß stückweise geschrieben und getestet werden.
- Rekursive Datenstrukturen brauchen rekursive Funktionen.
- Input muß auch auf Plausibilität getestet werden.
- Der Benutzer muß Input leicht vorbereiten und Output leicht weiterverarbeiten können.
- Input muß leicht auf Fehler getestet werden können.
- Höre nie auf zu testen, wenn Du einen Fehler gefunden hast.
- 10.0 mal 0.1 ist selten 1.0.
- Vergleiche niemals Gleitkommazahlen.
- Spezialfälle müssen wirklich etwas besonderes sein.
- Einfache Dinge sind häufig schneller.
- Spiel nicht herum, um einen Code schneller zu machen. Such einen besseren Algorithmus.
- Führe eine Zeitmessung durch, bevor Du versuchst, Code schneller zu machen.
- Kommentare und Code müssen übereinstimmen.
- Kommentiere nie schlechten Code. Schreibe ihn neu.
- Dokumentiere auch die Datenstruktur.

- Korrektheit korreliert mit Qualität.
- Sei paranoid. Dinge werden schiefgehen.
- Fang klein an. Mach die Software langsam komplizierter.
- Löse das Hauptproblem zuerst.
- Plane ein, Codeteile wegzuwerfen. Du wirst es sowieso tun müssen.

Literatur

[1] Brian W. Kernighan und O. J. Plauger: *The Elements of Programming Style*. McGraw-Hill 1974

[2] Lester McCann: Towards Developing a Good Programming Style. `http://basil.cs.uwp.edu/staff/mccann/style.html`