# Programmieren Sie

## Strukturiert programmieren in Basic? vielen Beispielen, wie. Alle Beispiele damit Sie sehen, wie es in einer

Kann man auch in Basic strukturiert programmieren?
Natürlich kann man. Die oft gehörte Behauptung, man könne strukturiert nur in einer »strukturierten« Computersprache programmieren, ist Unsinn. Sie beruht auf einem schlichten Denkfehler, nämlich dem, daß Programmieren identisch sei mit dem Schreiben von Programmbefehlen.

Tatsächlich läuft (vernünftiges) Programmieren in zwei voneinander unabhängigen Schritten ab:
Schritt 1: Entwerfen eines Programms
Schritt 2: Codieren des Programmentwurfs

Ein Programm entwerfen heißt einen Plan für das Programm machen. Dieser Plan ist an dem Problem orientiert, das man mit dem Computer lösen will, nicht an der Programmiersprache, mit Hilfe derer man dem Computer seine Absichten mitzuteilen gedenkt. Das ist wie wenn Sie ein Bild aufhängen. Das planen Sie auch nicht, indem Sie über den Hammer nachdenken. Vielmehr geht es um das Bild und die Wand und andere Bilder und Möbelstücke, die an dieser Wand stehen. Zu diesem Zeitpunkt, da Sie sich überlegen, an welcher Stelle genau das Bild einmal hängen soll, ist der Hammer noch völlig ohne Belang. Der Hammer wird erst dann akut, wenn Sie daran gehen, Ihren Plan zu realisieren, das Bild tatsächlich aufhängen.

Der Hammer ist das Werkzeug, das Sie benutzen, um Ihren Plan in die Tat umsetzen, mehr nicht. Eine Computersprache ist das Werkzeug, das Sie brauchen, um einen Programmentwurf in ein Programm umzusetzen, mehr Beachtung verdient sie nicht. Das heißt sie kommt erst dann ins Spiel, wenn der Programmentwurf fertig ist und codiert werden soll. Und an dieser Stelle erst kann die Frage gestellt werden, was der Unterschied zwischen einer unstrukturierten Sprache wie Basic und einer strukturierten Sprache wie Pascal oder Elan oder Comal ist.

Der Unterschied ist der: Eine »strukturierte« Sprache stellt Befehle zur Verfügung, die es leichter machen, einen gut strukturierten Programmentwurf in ein Computerprogramm umzusetzen. Eine »unstrukturierte« Sprache wie Basic hat keine oder nur ein paar einzelne vorgefertigte Befehlsstrukturen dieser Art. Das heißt aber nicht, daß deshalb die Umsetzung eines strukturierten Programmentwurfs nicht möglich wäre; das bedeutet nur, daß wir uns in Basic entsprechende Befehlsstrukturen selber machen müssen. Mit einem Wort: Basic ist unbequemer, aber das ist auch schon alles.

Ein Beispiel, damit Sie sehen, was ich meine. Eine Schleife soll so lange durchlaufen werden, bis die Taste »E« gedrückt wird; dann soll die Schleife verlassen werden.

In einer strukturierten Sprache wie Comal läßt sich dies so programmieren (Listing 1).

```
10   REPEAT
20   ...
30   ...
30   ...
40   taste$=KEY$
50   UNTIL taste$ = "e"
```

**Listing 1. Eine Schleife in Comal**

In Basic haben wir keine REPEAT-UNTIL Struktur, müssen also selber eine erfinden (Listing 2).

```
10   rem repeat
20   ...
30   ...
40   ...
50   get taste$
60   if not (taste$="e") then 20
70   rem until
```

**Listing 2. REPEAT...UNTIL in Basic**

Wenn wir eine solche Befehlsstruktur einmal erfunden haben, dann steht sie dem Basic-Codierer genau so zur Verfügung wie eine vorfabrizierte dem, der in Comal oder Pascal codiert. Und das heißt, das Codieren eines strukturierten Programmentwurfs wird in Basic (fast) genau so bequem wie in Comal oder Pascal. In dieser Aufsatzserie wollen wir solche Befehlsstrukturen für Basic entwickeln.

Denken Sie, wenn auch vielleicht ungern, kurz einmal an den Deutschunterricht in Ihrer Schulzeit zurück. Da mußten Sie Aufsätze schreiben. Sie lernten, daß ein Aufsatz aus Einleitung, Hauptteil und Schluß besteht; daß jeder Hauptgedanke seinen eigenen Absatz wert ist; daß Gedanken in einer logischen Abfolge dargestellt werden sollten und außerdem in einer einfachen und klaren Sprache.

Was Sie da lernten, waren nichts als ein paar technische Tricks, die, wenn Sie sie benutzten, das Aufsatzschreiben leichter machten. Mit Hilfe dieser Tricks kamen Sie schneller, effektiver ans Ziel. Mit weniger Frust und mehr Spaß. Und hatten als Ergebnis einen Aufsatz, den der Leser verstand. Der Zweck des Aufsatzes, sich mitzuteilen, war erreicht.

## Was heißt »strukturiert«, und warum überhaupt?

Programmieren ist dem Aufsatzschreiben recht ähnlich. Auch Programmieren soll schnell, mühelos, effektiv vonstatten gehen, und sein Ergebnis, das Programm, soll verständlich sein.

Wobei Sie sich zwei Adressaten gleichermaßen mitteilen wollen: Einmal natürlich dem Computer, der aufgrund des Programms haargenau das tun soll, was Sie sich ausgedacht haben; zum andern wollen Sie sich dem menschlichen Leser mitteilen. Sie meinen, wer könnte schon Ihr Programm lesen wollen? Vor allem Sie selbst!

Nichts ist so gut, als daß es nicht verbessert werden könnte, um mit dem Dichter zu sprechen. Wenn dieser Spruch überhaupt auf etwas zutrifft, dann auf Computerprogramme. Wer programmiert, weiß es. Wenn Sie aber ein Computerprogramm verbessern wollen, dann müssen Sie es so geschrieben haben, daß Sie's auch in einem halben Jahr noch verstehen.

Schreiben Sie also Ihre Programme mit Einleitung, Hauptteil und Schluß; machen Sie Absätze und ordnen Sie diese logisch an; benutzen Sie eine einfache, klare Ausdrucksweise. Mit andern Worten, programmieren Sie strukturiert.

Welche technischen Tricks Ihnen dabei helfen können, darum soll es in dieser Serie gehen.

Ein Computerprogramm besteht aus Bausteinen.

Es gibt Bausteine verschiedenster Komplexität. Einfache Bausteine sind zum Beispiel Befehlswörter wie »Print« oder »Input«, oder Daten wie »3.14« oder »pi«. Schon komplexer sind Befehlssätze wie »Print pi« oder »Input "Bitte geben Sie Ihren Namen ein"; name$«. Noch komplexer sind Befehlsstrukturen, wie sie oben illustriert wurden (Listings 1 und 2); sie sind vergleichbar den Absätzen in einem menschensprachlichen Text.

In dieser Serie geht es um solche komplexen Bausteine. Und zwar werden wir uns mit drei Bausteintypen beschäftigen:
1. den Steuerbausteinen — mit ihrer Hilfe wird der Programmfluß gelenkt
2. den Unterprogrammbausteinen — sie dienen dazu, zusammengehörige Befehle zusammenzufassen
3. den Modulbausteinen — aus diesen besteht schließlich das Programm

In diesem ersten Teil werden die Steuerbausteine behandelt; im nächsten geht es um Unterprogramme; zum Schluß wollen wir ein Programm bauen und dabei die Modulbausteine in den Mittelpunkt stellen.

Zunächst aber lassen Sie uns ein paar Grundregeln für die Codierung von Programm-Bausteinen zusammenstellen.

## Grundregeln für Programmbausteine

Die Bausteine, als denen Programmentwürfe zusammengesetzt sind, sind Gefäße für gedanklich zusammenhängende Einheiten. Im Programm werden sie deshalb als zusammenhängende Zeilenblocks realisiert (Bild 1).

# strukturiert! (Teil 1)

## Kein Problem. Wir zeigen Ihnen mit werden auch in Comal geschrieben, »strukturierten« Sprache aussieht.

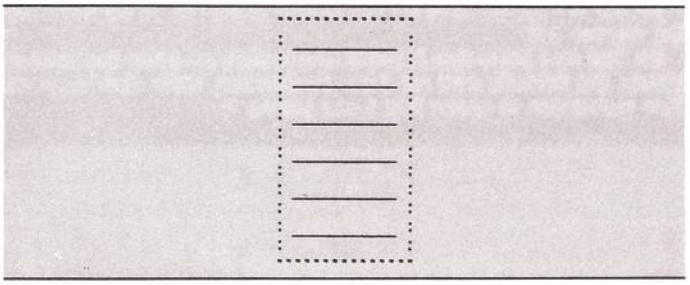

**Bild 1. Ein Zeilenblock besteht aus einer oder mehreren Programmzeilen**

Diese Zeilenblocks sind nach folgenden Regeln gebaut:
1. Ein Block besteht aus einer oder mehreren Programmzeilen. Dabei enthält jede Programmzeile nur einen Befehl.
2. Jeder Block hat nur einen Eingang und nur einen Ausgang. Der Eingang befindet sich in der ersten Zeile des Blocks, der Ausgang in der letzten Zeile.
3. Der GOTO-Befehl, soweit er überhaupt verwendet wird (in Schleifen- und in Verzweigungsblocks), darf nur zu Sprüngen innerhalb eines Blocks benutzt werden, und auch dabei gelten bestimmte Regeln.
4. Es kann manchmal nützlich und sinnvoll sein, einen Zeilenblock in eine Einzelzeile umzuwandeln (Bild 2).

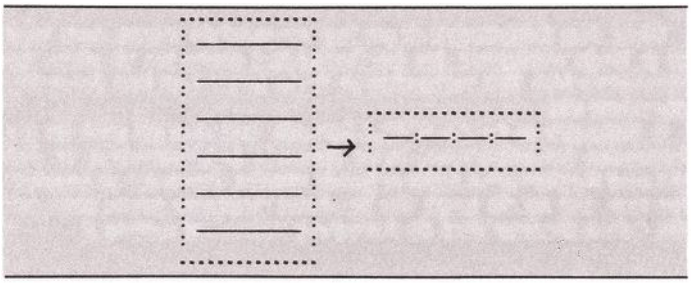

**Bild 2. Manchmal ist es vorteilhaft, aus mehreren Zeilen eine zu machen.**

Einzeilenblocks sind in vielen Fällen mit weniger Aufwand codierbar, übersichtlicher und leichter verständlich und meist schneller in der Programmausführung.

Unabdingbar ist bei einer derartigen Umwandlung allerdings, daß der gesamte Block, der ganze Baustein, in einer Zeile Platz findet, so daß die gedankliche Einheit des Bausteins nicht zerstört wird.

Wenn diese Regeln beachtet werden, dann werden dadurch automatisch Spaghetti-Programme vermieden, die, wie jeder weiß, der solche gemacht hat, das Leben des Programmierers sehr erschweren.

## Die Steuerbausteine

Wir werden die folgenden drei Typen von Steuerbausteinen besprechen:
1. die Abfolge oder Sequenz
2. die Schleife
3. die Verzweigung

Mehr als diese drei Typen braucht man nicht, mit diesen drei Typen läßt sich jedes Steuerungsproblem lösen.

Die Steuerbausteine unterscheiden sich in der Art und Weise, wie sie den Programmfluß lenken: In einer Sequenz arbeitet das Programm linear einen Befehl nach dem andern ab; die Schleife bewirkt, daß eine Befehlsreihe immer wieder von neuem durchlaufen wird; Verzweigungen machen es möglich, das Programm auf alternative Wege zu schicken, wobei die Wahl des Wegs von bestimmten Bedingungen abhängig sein kann.

Wir untersuchen diese drei Bausteintypen der Reihe nach. Dabei werden wir feststellen, daß es bei Schleifen und Verzweigungen Untertypen gibt, die jeweils für verschiedene Einsatzbedingungen geeignet sind. Diese sollen beschrieben und anhand kleiner Programmierprobleme illustriert werden.

Zuerst machen wir jeweils einen Programmentwurf, der die gedankliche Logik des Bausteins zeigt. Dann kommt der zweite Schritt, die Codierung. Wo Basic keine passende Befehlsstruktur zur Verfügung stellt, werden wir selber eine entwickeln.

Dabei lassen wir uns anregen. Man soll ja, heißt es, das Rad nicht immer wieder neu erfinden. Die Anregung holen wir vor allem aus Comal. Diese Programmiersprache enthält eine Reihe sehr gut durchdachter Befehlsstrukturen, die in der Regel so aufgebaut sind, daß die Bausteine direkt, ohne geistige Umwege, das heißt der menschlichen Logik folgend, umgesetzt werden können. Diese Befehlsstrukturen können als Vorbild für unsere eigenen Basic-Befehlsstrukturen dienen.

Wir wollen deshalb unsere Programmentwürfe jeweils zuerst in Comal codieren und dann versuchen, ähnliches in Basic zu realisieren.

In einigen Fällen werden zusätzlich noch alternative Codierungsmöglichkeiten aufgezeigt, weil sie vielleicht schneller sind, oder übersichtlicher, oder bequemer zu codieren.

Was Comal angeht, so codieren wir meist in der (fast) kostenlosen Diskettenversion 0.14. Wo einmal die Steckmodulversion 2.01 benutzt wird, wird ausdrücklich darauf hingewiesen. (Bezugsinformation für Comal am Schluß dieser Folge.)

## Die Abfolge oder Sequenz

Der einfachste Steuerbaustein ist die Abfolge oder Sequenz von Befehlen, die so, wie sie dastehen, hintereinander abgearbeitet werden. Eine spezielle Befehlsstruktur gibt es dafür weder in Basic noch in Comal. (Bild 3: Dollarrechner).


**Bild 3. Die Sequenz**

Codiert in Comal als Sequenzblock sieht dieser Baustein so aus (Listing 3).

```
0010    sequenz
0020    kurs:=3.21
0030    Input "Bitte Dollarbetrag eingeben:":dollarbetrag
0040    mark:=dollarbetrag*kurs
0050    Print mark
0060    ende sequenz
```

**Listing 3. Die Sequenz in Comal**

Die Basic-Version dieses Bausteins ist fast identisch; wobei zu beachten ist, daß Basic bei Variablennamen nur die beiden ersten Zeichen berücksichtigt, es würde also genügen, KU, DO und MA zu schreiben. Aber das macht die Basic-Codierung natürlich schwerer verständlich (Listing 4).

```
10   rem sequenz
20   :kurs=3.21
30   :input "Bitte Dollarbetrag eingeben"; dollarbetrag
40   :mark=dollarbetrag*kurs
50   :print mark
60   rem ende sequenz
```

**Listing 4. Die Sequenz in Basic**

Hinweis: Das Einrücken von Zeilen kann in Commodore-Basic auf zweierlei Weise erreicht werden. Einmal durch Verwendung von Doppelpunkten wie in Listing 4. Zum andern so: Nach der Zeilennummer geben Sie irgendein Grafikzeichen ein (zum Beispiel SHIFT N), dann so viele Leerzeichen, wie Sie brauchen, dann den Befehl, der in der Zeile stehen soll. Das Grafikzeichen verschwindet, wenn Sie die Zeile listen. Aber Vorsicht: Wenn Sie später in der Zeile eine Änderung vornehmen, geht das (unsichtbare) Grafikzeichen verloren, und der Zeilentext rückt wieder nach links. Sie müssen dann die Zeile neu »formatieren«.

## Die Schleifen

Der Bausteintyp Schleife wird dann verwendet, wenn eine Reihe von Befehlen mehrmals hintereinander wiederholt werden soll. Dabei sind mehrere Arten von Schleifen zu unterscheiden:

1. die Zählschleife
2. die WHILE-Schleife
3. die UNTIL-Schleife
4. die LOOP-Schleife
5. die Endlosschleife

Die Zählschleife benutzen wir immer dann, wenn wir von vornherein wissen oder ausrechnen können, wie oft die Schleife durchlaufen werden soll (siehe Bild 4).

Dieser Baustein soll die einzelnen Buchstaben eines Texts senkrecht untereinander schreiben. Die Anzahl der Schleifendurchgänge hängt von der Länge des Texts ab und läßt sich also ausrechnen. Der Text wird dem Baustein mitgegeben.

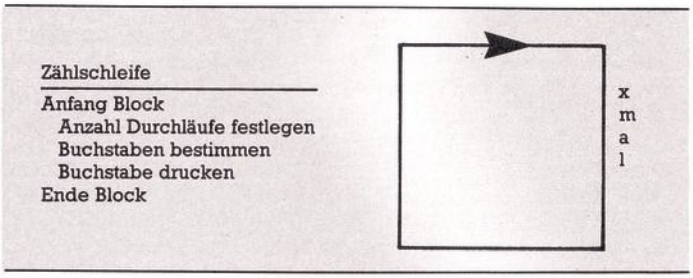


**Bild 4. Die Zählschleife**

Für Zählschleifen stellen sowohl Comal wie Basic eine (fast identische) Befehlsstruktur zur Verfügung (Listing 5).

```
0010   FOR position:=1 TO LEN(text$) DO
0020     buchstabe$:=text$(position)
0030     PRINT buchstabe$
0040   ENDFOR position

10     for ps=1 to len(text$,ps,1)
20       buchstabe$=mid$(text$,ps,1)
30       print buchstabe$
40     next ps
```

**Listing 5. Die Zählschleife in Comal (oben) und Basic (unten)**

Die Befehlsfolge innerhalb des Schleifenrahmens können wir zusammenfassen zu einem einzigen Befehl (siehe Listing 6), was uns die Möglichkeit gibt, den gesamten Block in einer einzigen Zeile unterzubringen (wodurch die Schleife schneller durchlaufen wird).

Oft weiß man nicht von vornherein und kann es auch nicht ausrechnen, wie oft eine Befehlsfolge durchlaufen werden muß, sondern die Anzahl der Durchläufe hängt von unvorhersehbaren Situationsbedingungen ab, zum Beispiel davon, ob eine Variable schon einen bestimmten Wert überschritten hat. Diese Situationsbedingung muß regelmäßig überprüft werden. Je nachdem, wann die Überprüfung

```
Der Einzeilenblock in Comal
   0010 FOR position:=1 TO LEN(text$) DO PRINT text$(position)
Der Einzeilenblock in Basic
   10 for ps=1 to len(text$) :buchst$=mid$(text$,ps,1) :print buchst$
:next
```

**Listing 6. Der Einzeilenblock für Listing 5**

stattfindet, ob vor dem Schleifendurchlauf oder danach oder mittendrin oder überhaupt nicht, ergeben sich verschiedene weitere Schleifentypen.

Bei der WHILE-Schleife wird zu Beginn überprüft, ob sie durchlaufen werden soll.

Ein Text wird bei jedem Schleifendurchgang um einen Buchstaben kürzer. Die Schleife wird nur dann durchlaufen, wenn überhaupt ein Text vorhanden ist (siehe Bild 5 und Listing 7).

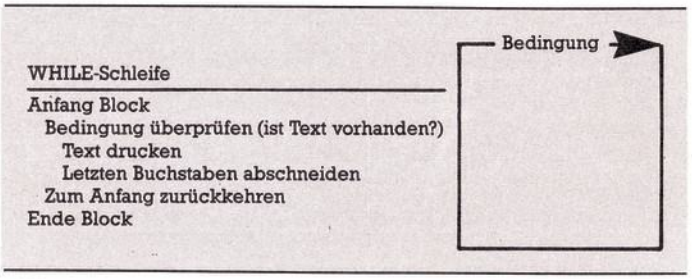


**Bild 5. Die WHILE-Schleife**

```
0010   WHILE text$ > "" DO
0020   PRINT text$
0030   text$:=text$(1:LEN(text$)-1)
0040   ENDWHILE
```

**Listing 7. Die WHILE-Schleife in Comal**

Wenn dem Schleifenblock ein Text mitgegeben wurde, dann wird er so lange wiederholt, bis die Textlänge 0 geworden ist; wenn von vornherein kein Text vorhanden ist, findet überhaupt kein Schleifendurchlauf statt.

In Basic gibt es für die WHILE-Schleife (und dies gilt auch für die folgenden Schleifentypen) keine spezielle Befehlsstruktur, so daß wir eine Blockstruktur dafür erfinden müssen. Diese soll so beschaffen sein, daß sie die Logik des Bausteins möglichst genau abbildet; die Comal-Struktur wird als Anregung mitbenutzt (Listing 8).

```
10   rem while
20     if not (text$ > "") then 60
30       print text$
40       text$=left$(text$,len(text$)-1)
50     goto 20
60   rem endwhile
```

**Listing 8. Die WHILE-Schleife in Basic**

Das Programm bleibt so lange in diesem Block, wie ein Text vorhanden ist. Wenn die Bedingung, daß ein Text vorhanden ist, nicht mehr gilt, wird der Block verlassen, und zwar durch den einzigen Ausgang, der vorgesehen ist, also durch die Zeile, die die Ende-Markierung enthält. Dies ist eine REM-Zeile. REM-Zeilen sind bei dieser Art von Codierung oft integraler Bestandteil des Programms und dürfen dann auf keinen Fall gelöscht werden!

Bitte beachten Sie besonders, daß bei der Basic-Version der WHILE-Schleife die Bedingung für den Schleifendurchlauf mit NOT negiert werden muß (dies werden wir noch öfter finden). Das kann

```
20   if text$ "" then print text$ :
       text$=left$(text$,len(text$)-1) : goto 20
```

**Listing 9. Die WHILE-Schleife in Basic als Einzeiler**

zu Denkschwierigkeiten führen und ist vielleicht dadurch fehleranfällig. Man kann diesem Problem jedoch entgehen, wenn man versucht, WHILE-Schleifen in einer einzigen Zeile zu codieren (Listing 9).

Diese Zeile wird nur ausgeführt, wenn die Bedingung zu Beginn zutrifft. Wenn zu viele Befehle auszuführen sind, so daß die Zeile nicht ausreicht, kann man die ganze Befehlsreihe als Unterprogramm auslagern. Die Codierung könnte dann so aussehen (Listing 9a):

```
  20    if text$ > ""then gosub 1000 : goto 20
  90    end
1000    rem proc:text kürzen
1010        print text$
1020        text$=left$(text$,len(text$)-1
1030    return
```

**Listing 9a. Die WHILE-Schleife mit Unterprogrammaufruf**

Übrigens kennt auch Comal einzeilige WHILE-Schleifen, sofern nur ein einziger Befehl auszuführen ist. Listing 9b entfernt überzählige Leerzeichen zu Beginn einer Zeichenkette.

```
0010    WHILE text$(1)=" " DO text$:=text$(2:LEN(text$))
```

**Listing 9b. Die WHILE-Schleife in Comal als Einzeiler**

Können Sie dasselbe in einer Basic-Zeile ausdrücken? Denken Sie daran, daß einzeilige WHILE-Schleifen in Basic mit IF beginnen und mit einem GOTO-Befehl auf dieselbe Zeile enden.

WHILE-Schleifen sind immer dann angebracht, wenn man sicherstellen will, daß die Schleife eventuell ganz übersprungen wird. Alle anderen Schleifentypen (mit Ausnahme der LOOP-Schleife, die einen speziellen Fall darstellt) werden mindestens einmal durchlaufen. Das gilt für die Zählschleife ebenso wie für die Endlosschleife und die UNTIL-Schleife, der wir uns nun als nächstes zuwenden.

Die UNTIL-Schleife überprüft die Situationsbedingungen erst am Ende eines Durchlaufs und entscheidet dann, ob ein weiterer notwendig ist (siehe Bild 6 und Listing 10).

DATA-Zeilen werden so lange gelesen, bis ein bestimmter Suchbegriff gefunden ist. Dann wird die Schleife verlassen.


**Bild 6. Die UNTIL-Schleife**

```
Comal:  0010    REPEAT
        0020      READ text$
        0030    UNTIL text$=gesucht$

Basic:    10  rem repeat
          20    read text$
          30    if not (text$=gesucht$) then 20
          40  rem until
```

**Listing 10. Die UNTIL-Schleife in Comal (oben) und Basic (unten)**

Die Bedingung wird auch hier wieder mit NOT negiert. Leichter zu codieren ist, ähnlich wie bei der WHILE-Schleife, ein Einzeilenblock (Listing 11).

```
20    read text$ : if not (text$=gesucht$) then 20
```

**Listing 11. Basic-Einzeiler für die UNTIL-Schleife**

Kennzeichnend für UNTIL-Schleifen dieser Art ist IF-NOT-THEN als letzte Befehlsgruppe und ein Sprung zum Beginn der Zeile.

Es gibt noch eine weitere Möglichkeit, UNTIL-Schleifen in Basic zu codieren, allerdings in etwas unüblicher Weise (dafür ist die Schleife jedoch schneller). Wir benutzen dafür den Zählschleifenbefehl FOR-NEXT (Listing 12).

```
10    rem repeat
20      for i=0 to 1
30        read text$
40        i=abs(text$=gesucht$)
50      next i
60    rem until
```

**Listing 12. Die UNTIL-Schleife in Basic mit FOR...NEXT**

Das funktioniert so: Die Schleife wird auf jeden Fall einmal durchlaufen. Dann wird in Zeile 40 überprüft: (TEXT$=GESUCHT$). Dieser Boolesche Ausdruck ergibt den Wert 0, wenn er falsch ist, das heißt wenn der gelesene Text noch nicht mit dem gesuchten Text übereinstimmt. In diesem Fall erhält I in Zeile 40 den Wert 0, und wenn das Program zu NEXT kommt, erfährt er es, daß I den Endwert nocht nicht erreicht hat und deshalb ein weiterer Durchlauf erforderlich ist.

Dies geht so lange, bis der richtige String gefunden ist. Dann hat (TEXT$=GESUCHT$) den Wert —1; durch die Funktion ABS wird das Minuszeichen entfernt, und I wird zu 1. Wenn aber I=1 ist, dann ist die Aufgabe der Schleife erfüllt und der ganze Block wird verlassen.

Übrigens entsteht ein »OUT OF DATA«-Fehler, wenn die gesuchte Zeichenkette gar nicht unter den Daten vorhanden ist. Deshalb muß normalerweise noch eine andere Bedingung geprüft werden, zum Beispiel, ob (TEXT$="ENDE") ist. Dies ist ohne Schwierigkeiten codierbar, es muß nur die Zeile 40 in folgender Weise geändert werden (Listing 12a):

```
40    i=abs(text$=gesucht$ or text$="ende"
```

**Listing 12a. Sicherheitsmaßnahme gegen OUT OF DATA**

In derselben Weise sind die Bedingungsausdrücke in den anderen Beispielen zu ändern.

Auch diese Codierungsart kann im übrigen in einer Zeile geschehen (Listing 13).

```
20    for i=0 to 1 : read text$ : i=abs(text$=gesucht$ or text$="ende":
      next
```

**Listing 13. Listing 12 und 12a als Einzeiler**

UNTIL-Schleifen diesen Typs erkennt man daran, daß sie von 0 bis 1 zählen und im vorletzten Befehl den Wert der Zählvariablen (hier I) berechnen. Da sie auf diese Weise genügend deutlich markiert sind, können wir uns die beiden REM-Zeilen in Listing 12 sparen.

Die am häufigsten benötigte Schleifenart ist die UNTIL-Schleife, danach kommt die WHILE-Schleife, am seltensten besteht ein Bedürfnis für die LOOP-Schleife, der Schleife mit Abbruch. Bei der LOOP-Schleife wird die Situation während des Schleifendurchlaufs, also innerhalb der Schleife, überprüft; wenn die Prüfbedingung zutrifft, wird der gerade stattfindende Durchlauf abgebrochen, wie das nächste Beispiel illustriert (Bild 7).

Die Schleife wird so lange durchlaufen, bis eine befriedigende Antwort eingeht; wenn dies der Fall ist, wird abgebrochen.


**Bild 7. Überprüfen einer Eingabe mit der LOOP-Schleife**

```
0010    LOOP
0020      INPUT ''Bitte geben Sie JA oder NEIN ein'': antwort$
0030      EXIT WHEN antwort$="JA"OR antwort$="NEIN"
0040      PRINT "Bitte nur JA oder NEIN eingeben!"
0050    ENDLOOP
```

**Listing 14. Bild 7, die LOOP-Schleife, mit Comal realisiert**

Die LOOP-Schleife, die übrigens nur in der Comal-Version 2.01 auf diese Weise codierbar ist (Listing 14), überprüft, wie gesagt, die Situation inmitten des Schleifendurchlaufs; das heißt dieser Schleifentyp ist nur dann gerechtfertigt, wenn sowohl vor der Überprüfung als auch danach Anweisungen ausgeführt werden können.

```
10    rem loop
20      input "Bitte geben Sie JA oder NEIN ein'; antw$
30      if antw$="JA" or antw$="NEIN" then 60
40      print 'Bitte nur JA oder NEIN eingeben!"
50      goto 20
60    rem endloop
```

**Listing 15. Die LOOP-Schleife in Basic**

Das Beispiel illustriert übrigens ein typisches Einsatzgebiet für die LOOP-Schleife: Mit ihrer Hilfe kann man den Notstand proben, einfacher ausgedrückt: voraussehbare Fehler abfangen. Die LOOP-Schleife wird auf jeden Fall »halb«, das heißt bis zur Abbruchstelle, durchlaufen. Fortsetzung und Neuanfang der Schleife geschehen nur dann, wenn der vorausgeahnte Notstand eintrifft (Listing 15 = Basic-Version).

Gelegentlich benötigt man auch einen letzten Schleifentyp, die Endlosschleife, zum Beispiel bei einem Demonstrationsprogramm, das pausenlos immer wieder von neuem ablaufen soll (Bild 8 und Listing 16).

Dafür gibt es keine spezielle Befehlsstruktur, was aber auch nicht erforderlich ist, da sich Endlosschleifen ohne Mühe codieren lassen.

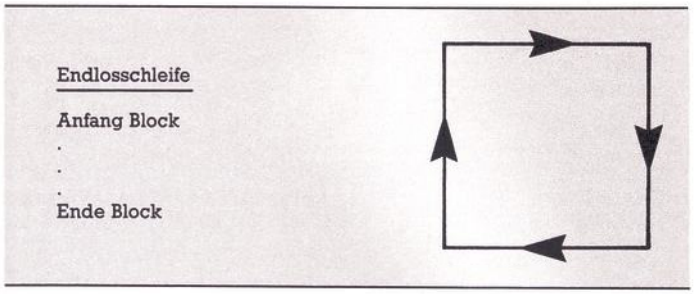


**Bild 8. Die Endlosschleife**

```
0010    REPEAT                    0010    LOOP
0020    ...                       0020    ...
0030    ...                       0030    ...
0040    ...                       0040    ...
0050    UNTIL TRUE=FALSE          0050    ENDLOOP

          10    rem endlos
          20      ...
          30      ...
          40      ...
          50      goto 20
          60    rem ende endlos
```

**Listing 16. Die Endlosschleife in Comal 0.14 (links oben), Comal 2.01 (rechts oben) und in Basic (unten)**

## Die Verzweigungen

Befehlssequenzen weisen den Computer an, eine Reihe von Befehlen hintereinander abzuarbeiten; Schleifen veranlassen ihn, eine Reihe von Befehlen unter bestimmten Umständen mehrmals zu durchlaufen; der dritte Bausteintyp, die Verzweigung, ermöglicht es, den Computer mal zur einen, mal zu einer anderen Befehlsgruppe zu schicken. Wohin er geht, hängt von der jeweils herrschenden Situation ab.

Wir können drei Untertypen der Verzweigung unterscheiden:

1. den Abstecher
2. die Gabelung
3. die Mehrfachverzweigung (oder Kreuzung)

Die einzelnen Verzweigungstypen unterscheiden sich in der Art und/oder Anzahl der alternativen Wege, zwischen denen zu entscheiden ist.

Einen »Abstecher« machen, heißt im Leben, den eigentlichen Weg, auf dem man sich befindet, kurz zu verlassen, etwas zu erledigen, um dann wieder auf den Hauptweg zurückzukehren. Dazu muß natürlich ein spezieller Grund vorhanden sein. Beispielsweise, man ist auf dem Nachhauseweg, verspürt plötzlich Appetit auf Kaffee, tritt ins Kaffegeschäft, das am Weg liegt, trinkt die Tasse Kaffee und setzt anschließend den Heimweg fort. Es kann aber auch sein, daß man keinen Kaffeedurst hat, dann beachtet man das Kaffeegeschäft überhaupt nicht und bleibt unbeirrt auf dem Heimweg.

Dasselbe gilt für Computerprogramme. Unter einer bestimmten Bedingung wird das Hauptprogramm kurz unterbrochen, eine Befehlsreihe wird ausgeführt, dann läuft das Programm auf dem Hauptweg weiter.

Ein Name wird in Klein/Großschreibung eingegeben. Wenn der erste Buchstabe klein geschrieben wurde, wird er in einen Großbuchstaben umgewandelt; wenn er schon groß ist, findet kein Abstecher statt (Bild 9 und Listing 17).

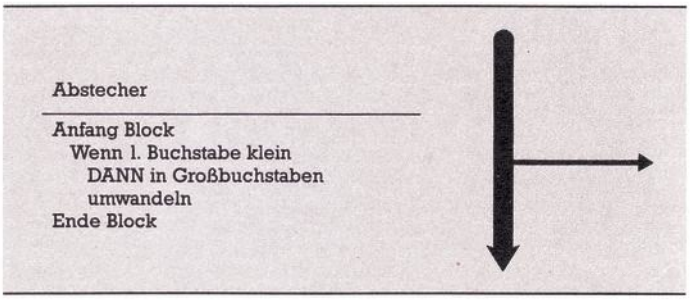


**Bild 9. Großschreibung**

```
0010    IF ORD(name$) < 91 THEN
0020      ALT:=ord(name$)
0030      neu:=alt+128
0040      name$:=chr$(neu)+name$(2:LEN(name$))
0050    ENDIF

10      rem if
20        if not(asc(name$) < 91) then 60
30          alt=asc(name$)
40          neu=alt+128
50          name$=chr$(neu)+mid$(name$,2)
60      rem endif
```

**Listing 17. Die Verzweigung als Abstecher in Comal und Basic**

Wie früher schon, müssen auch hier die Bedingungen negieren, damit der natürliche Gedankenfluß erhalten bleibt.

Oft wird beim Abstecher allerdings nur ein einziger Befehl ausgeführt; dann können wir den ganzen Block in einer Zeile unterbringen, was die Verständlichkeit auch in Basic erhöht, schon deshalb, weil wie auch woanders, die Bedingung in ihrer »Grundform« verwendet werden kann. Dies sei am obigen Beispiel gezeigt, indem wir die Befehlsreihe zu einem einzigen Befehl zusammenfassen (Listing 17a).

```
0010    IF ORD(name$) < 91 THEN
          name$:=CHR$(ORD(name$)+128)+name$(2:LEN(name$))

10    if asc(name$) < 91 then name$=chr$(asc(name$)+128) +
      mid$(name$,2)
```

**Listing 17a. Listing 17 als Einzeiler (oben: Comal, unten: Basic)**

In Basic kann dem THEN übrigens nicht nur ein Befehl folgen, sondern mehrere — so weit halt in der Zeile Platz bleibt. Wie immer, kann man aber auch eine größere Anzahl von Befehlen unterbrin-

gen, indem man sie als Unterprogramm auslagert (siehe Listing 9).

Eine »Gabelung« in einem Programm ist wie eine Weggabelung: Es gibt zwei Möglichkeiten weiterzugehen, und es muß entschieden werden, welchen der beiden Wege man nimmt (Bild 10).

```
Gabelung
-------------------------
Anfang Block
  WENN das Jahr Schaltjahr
    DANN hat Feb. 29 Tage
  ANDERNFALLS hat er 28 Tage
Ende Block
```

**Bild 10. Die Verzweigung als Gabelung**

```
0010   IF jahr MOD 4=0 THEN
0020     tage:=29
0030   ELSE
0040     tage:=28
0050   ENDIF
```

```
10   rem if else
20     if not(jahr/4=int(jahr/4)) then 50
30       tage=29
40     goto 60
50       tage=28
60   rem endif
```

**Listing 18. Die »Gabelung« in Comal und in Basic**

(Der Operator MOD errechnet den Divisionsrest; Beispiel: 18 MOD 4 ergibt den Rest 2.)

Die von Comal zur Verfügung gestellte Befehlsgruppe IF=THEN-ELSE erlaubt die Codierung analog zur menschlichen Logik. In Basic müssen wir, um diese Logik zu erhalten, die Bedingung wiederum mit NOT negieren (Listing 18).

Und wieder gibt es die Möglichkeit, diese Negierung zu vermeiden, indem wir versuchen, den Gabelungsblock in einer einzigen Zeile unterzubringen (Listing 19).

```
20 tage=28 :    if jahr/4=int(jahr/4) then tage=29
```

**Listing 19. Listing 18 als Einzeiler**

Kommen wir zum letzten Verzweigungstyp, der Mehrfachverzweigung. Sie funktionert wie eine Wegekreuzung, an der wir die Wahl zwischen drei oder mehr Alternativen haben (Bild 11 und Listing 20).

Für jeden Monat soll die entsprechende Anzahl von Tagen ermittelt werden können.

```
Mehrfachverzweigung
-------------------------
Anfang Block
  WENN der 1., 3., 5., 7., 8., 10., 12. Monat
    DANN 31 Tage
  WENN HINGEGEN der 4., 6., 9., 11. Monat
    DANN 30 Tage
  WENN HINGEGEN der 2. Mt.
    DANN 28 Tage
  ANDERNFALLS Fehlermeldung
  und 0 Tage
Ende Block
```

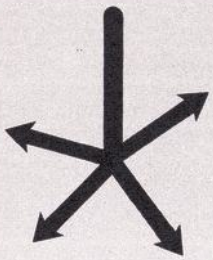

**Bild 11. Die Mehrfachverzweigung**

```
0010   CASE monat OF
0020   WHEN 1,3,5,7,8,10,12
0030     tage:=31
0040   WHEN 4,6,9,11
0050     tage:=30
0060   WHEN 2
0070     tage:=28
0080   OTHERWISE
0090     PRINT "Einen";monat.". Monat gibt's nicht."
0100     tage:=0
0110   ENDCASE
```

**Listing 20. Die Mehrfachverzweigung in Comal**

Die Case-Struktur funktioniert so: Es wird zuerst der Wert der Variablen MONAT überprüft. Angenommen, es handelt sich um den Juni, also den 6. Monat, dann geht das Programm in den Zweig, der in seiner Werteliste den Wert 6 enthält, also zur Zeile 40, und arbeitet die dazugehörigen Befehle ab; so wird TAGE zu 30. Danach wird der Block verlassen. Wenn der Wert von MONAT in keiner der Wertelisten enthalten ist, das heißt wenn MONAT kleiner als 1 oder größer als 12 ist, dann verzweigt das Programm auf OTHERWISE.

In Basic kann dieser (spezielle) Fall so codiert werden (Listing 21):

```
10    rem case
20    on mt goto 30,70,30,50,30,50,30,30,50,30,50,30 : goto 90
30      tage=31
40    goto 110
50      tage=30
60    goto 110
70      tage=28
80    goto 110
90      print"Einen"mt". Monat gibt's nicht."
100     tage=0
110   rem endcase
```

**Listing 21. Die Mehrfachverzweigung in Basic**

Die ON-Struktur in Basic (Zeile 20) überprüft, so wie dies auch bei CASE der Fall war, den Wert der Variablen MT. Wenn dieser Wert 6 ist, wählt das Programm die sechste Zeilennummer der Liste aus, also 50, und springt zu dieser Zeile. TAGE erhält dort den Wert 30, und das Programm verläßt den Block über die letzte Zeile. Wenn MT einen Wert hat, zu dem es keine Zeilennummer in der Liste gibt (0 oder größer als 12), springt das Programm zum nächsten Befehl, also zu GOTO 90. Eine negative Zahl führt zu einer Fehlermeldung.

Die Variable MT kann bei diesem Beispiel die Werte 1-12 annehmen, das heißt die Werte einer Zahlenfolge. Dies ist auch der einzige Fall, wo diese Struktur in Basic verwendet werden kann.

Die CASE-Struktur in Comal kennt diese Einschränkung nicht. Zum einen kann die Variable zu jedem Typ gehören, kann also auch eine Stringvariable sein. Zum anderen brauchen die Werte, welche die Variable annimmt, nicht Elemente einer Folge sein. Schließlich sind bei Zahlenvariablen auch negative Werte legal. Die Case-Struktur ist also in weit mehr Situationen einsetzbar als die mit Hilfe von ON imitierte Struktur in Basic.

Wenn die Bedingungsverhältnisse nicht einer Zahlenfolge entsprechen oder überhaupt komplexer sind, müssen wir eine andere Struktur benutzen, und zwar eine, die von der Comal-Struktur IF-ELIF-ELSE abgeleitet ist. (ELIF ist zusammengezogen aus ELSE IF).

Aufgabe: Ein Text soll verändert werden. Der Mehrfachverzweigungsblock bearbeitet jeweils ein Zeichen (Bild 12 und Listing 22/23).

```
Mehrfachverzweigung
-------------------------
Anfang Block
  WENN Vokal
    DANN ersetzen durch '*'
  WENN HINGEGEN Konsonant
    DANN in Großbuchstaben umwandeln
  WENN HINGEGEN Zahl
    DANN weglassen
  ANDERNFALLS ohne Veränderung übernehmen
Ende Block
```

**Bild 12. Textumwandlung**

Auch hier (Listing 23) müssen wir wieder die Bedingungen mit NOT negieren, damit die normale Logik im Programmfluß erhalten bleibt.

Damit sind wir mit der Beschreibung der Steuerbausteine am Ende.

Allerdings nicht zu Ende sind wir mit dem Thema des strukturierten Programmierens. Wer sich mit Steuerbausteinen der beschriebenen Art begnügt und ansonsten wie üblich vor sich hin programmiert, der programmiert noch lange nicht strukturiert. Damit ist noch kein logischer und übersichtlicher Aufbau des Programms gewährleistet. Sicher, ohne Steuerbausteine ist strukturiertes Programmieren nicht möglich, aber die Benutzung von Steuerbausteinen ist nur der erste Schritt. Zu diesen müssen noch zwei weitere Bausteintypen kommen: Unterprogramm- und Modulbausteine. Fortsetzung auf S. 154

Fortsetzung von Seite 125

```
0010    IF zeichen$ IN "aeiou" THEN
0020      neu$:="*"
0030    ELIF ORD(zeichen$) > 64 AND ORD(zeichen$) < 91 THEN
0040    neu$:=CHR$(ORD(zeichen$)+128)
0050    ELIF ORD(zeichen$) > 47 AND ORD(zeichen$) < 58 THEN
0060      neu$:=""
0070    ELSE
0080      neu$:zeichen$
0090    ENDIF
```

**Listing 22. Textumwandlung in Comal**

```
10      rem if elif
20        if not (z$="a" or z$="e" or z$="i" or z$="o" or z$="u")
          then 50
30          neu$="*"
40          goto 120
50        if not (asc(z$) > 64 and asc(z$) < 91) then 80
60          neu$=chr$(asc(z$)+128)
70          goto 120
80        if not (asc(z$) > 47 and asc(z$) < 58) then 110
90          neu$=""
100         goto 120
110       neu$=z$
120     rem endif
```

**Listing 23. Textumwandlung in Basic**

Das wird häufig vergessen. So heißt es zum Beispiel in letzter Zeit des öfteren, das Basic 7.0 des neuen Commodore 128 mache nun endlich strukturiertes Programmieren möglich, auch in dieser Zeitschrift war solches zu lesen. Gewiß, der erste Schritt wird erleichtert, und das sei dankbar vermerkt. Aber wer ein wirklich »strukturiertes« Basic möchte, der muß sich entweder einen anderen Computer anschaffen, wie zum Beispiel das BBC-Micro, oder eine Basic-Erweiterung, die neben vorgefertigten Steuerbausteinen auch mindestens einige vorgefertigte Unterprogrammbausteine anbietet, wie zum Beispiel das auch in dieser Zeitschrift schon besprochene Macro-Basic.

Um Unterprogrammbausteine, also um Prozeduren und Funktionen, wird es im zweiten Teil dieser Serie gehen.

## Zusammenfassung

Fassen wir die Bausteine, die wir kennengelernt haben, noch einmal in einer Übersichtstabelle zusammen (Bild 13).

| | Bausteine | |
|---|---|---|
| **Abfolge oder Sequenz:** | **Schleife** | **Verzweigung** |
| | Zählschleife | Abstecher |
| | WHILE-Schleife | Gabelung |
| | UNTIL-Schleife | Mehrfachverzweigung |
| | LOOP-Schleife | |
| | Endlosschleife | |

Wir haben gesehen, daß diese Bausteine sowohl in einer »strukturierten« Computersprache wie Comal codiert werden können als auch in einer »unstrukturierten« wie Basic. Der Unterschied liegt nur darin, daß Comal für die meisten Bausteine entsprechende Befehlsblocks zur Verfügung stellt, so daß diese Denkschritt für Denkschritt umgesetzt werden können. In Basic hingegen sind wir gezwungen, uns entsprechende Befehlsstrukturen selber zu schaffen. Aber so schwer ist das, wie wir gesehen haben, gar nicht, und wenn wir sie erst einmal haben, dann ist die Bausteinlogik genau so leicht zu codieren wie in Comal.

Entsprechendes gilt für die anderen Bausteintypen, mit denen wir uns noch beschäftigen wollen. (Prof. Burkhard Leuschner/gk)

Hinweise:
Besonders anregend fand ich folgende neuere Literatur:
Kopp, Martin (1984), Neue Strukturen im alten Basic, INFO (Rundbrief des PTC, Offenburg) 6(1), 57-58
Lührmann, Arthur (1983), Slicing through spagetti code, The Computing Teacher 10(8), 9-15
Lührmann, Arthur (1984), Structured Programming in Basic, Creative Computing 10(5), 152-156; 10(6), 152-163; 10(7), 125-136; 10(9), 171-177
Metzler, Richard C.(1985), If rules then better structured Basic, The Computer Teacher 12(4), 12-14

Wenn Sie Comal unerwarteterweise noch nicht haben — die Diskettenversion 0.14 (C 64, CBM 40332/8032) ist gegen geringe Unkosten zu erhalten. Andere Versionen (auch für andere Computer, zum Beispiel IBM-PC) kosten Geld. Wenn Sie Näheres wissen wollen: Adressieren Sie einen Umschlag, schreiben Sie »Drucksache« drauf und frankieren Sie ihn mit 50 Pfennig. Schicken Sie diesen Umschlag an: »Professsor Burkhard Leuschner, Kennwort Comal, Pädagogische Hochschule, Oberbettringer Str. 200, 7070 Schwäbisch Gmünd«.

Sie erhalten dann die neuesten Bezugsinformationen. Bitte keine Mitteilungen beilegen, sie werden nicht gelesen! Und ein paar Tage Geduld haben!

# Softlearning — Lernen auf ganz neue Art?

## Softlearning erschließt eine neue Methode des Lernens, das »Superlearning«. Wie erfolgreich ist diese neue Art des Lernens?

Über diese neue Art des Lernens, das auf psychologischen und biochemischen Vorgängen beruht, wurde in der Fachwelt viel diskutiert. Das Buch »Superlearning« der beiden Autoren Ostrander und Schröder kann als Vorbild für das Computerprogramm von SM Software angesehen werden. Mit der neuen »Amadeus«-Reihe wurde das bisherige Softlearning um eine neue Variante bereichert.

Schon vor der Geburt im Mutterleib lernt der Mensch ununterbrochen. Dieses für das Überleben so wichtige Lernen geschieht freiwillig und automatisch. Wesentliche Faktoren sind neben dem Inhalt aber auch die Intensität und die Bereitschaft Information aufzunehmen.

Beispielsweise lernt man als Kind deshalb so schnell und leicht die erste Sprache, da man mit dieser seine Wünsche wesentlich besser artikulieren und in Kommunikation mit seinen Mitmenschen treten kann. In der Schule verliert man im allgemeinen diese positive Einstellung zum Lernen, man muß sich dazu zwingen und ungeliebten Stoff in sich hineinpauken. Das wird um so anstrengender und weniger effektiv, je weniger man den Sinn dieses Stoffes versteht. Von diesen Erkenntnissen ausgehend, versuchte man neue Formen des Lernens zu finden. Ergebnis ist das sogenannte »Superlearning«, das in einer Tiefenentspannungsphase Information direkt ins Unterbewußtsein bringt. Auf diese Art verabreichtes Wissen sitzt dann wesentlich fester im Gedächtnis und benötigt wesentlich weniger Zeit zur Vermittlung. Man versucht dabei Lernstoff an der Bewußtseinsschwelle vorbei direkt in das Lang-


**Multiple Choice bei SM-Softlearning**

Fortsetzung auf Seite 157