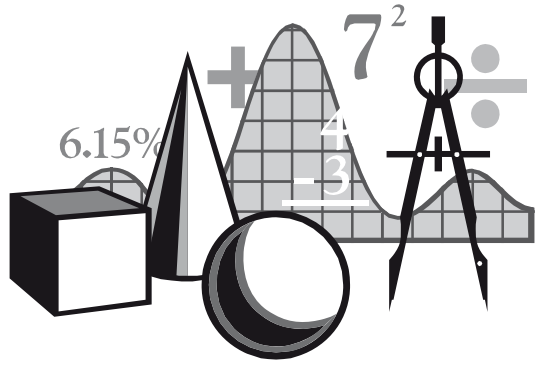
Kapitel 3

Entwurf von Programmen



3 Entwurf von Programmen

Bevor man mit einer Programmiersprache umzugehen lernt, muss man wissen, was ein Programm prinzipiell ist und wie man Programme entwirft. Damit wird sich unter anderem dieses Kapitel befassen. Leser, die bereits entwerfen können, sollten prüfen, ob sie tatsächlich die hier präsentierten Grundlagen (noch) beherrschen, und sollten gegebenenfalls dieses Kapitel „überfliegen“.

Kapitel 3.1 stellt Programme und ihre Abläufe vor. Kapitel 3.2 erläutert die Veranschaulichung des Ablaufs von Programmen mittels Nassi-Shneiderman-Diagrammen, welche auch unter der Bezeichnung Struktogramme bekannt sind. Kapitel 3.3 diskutiert die Möglichkeiten eines Pseudocodes.

3.1 Vom Problem zum Programm

Der Begriff **Programm** ist eng mit dem Begriff **Algorithmus** verbunden. Algorithmen werden in Programmen umgesetzt.

Algorithmen sind Vorschriften für die Lösung eines Problems, welche die **Handlungen und ihre Abfolge** – also die Handlungsweise – beschreiben.

Im Alltag begegnet man Algorithmen in Form von Bastelanleitungen, Kochrezepten und Gebrauchsanweisungen.

Abstrakt kann man sagen, dass die folgenden Bestandteile und Eigenschaften zu einem Algorithmus gehören:

- eine **Menge von Objekten**, die durch den Algorithmus bearbeitet werden,
- eine **Menge von Operationen**, die auf den Objekten ausgeführt werden,
- ein definierter **Anfangszustand**, in dem sich die Objekte zu Beginn befinden,
- und ein gewünschter **Endzustand**, in dem sich die Objekte nach der Lösung des Problems befinden sollen.

Dies sei am Beispiel Kochrezept erläutert:

Objekte:	Zutaten, Geschirr, Herd, ...
Operationen:	waschen, anbraten, schälen, passieren, ...
Anfangszustand:	Zutaten im „Rohzustand“, Teller leer, Herd kalt, ...
Endzustand:	fantastische Mahlzeit auf dem Teller

Was dann noch neben der Anleitung oder dem Rezept zur Lösung eines Problems gebraucht wird, ist jemand, der es macht – im angeführten Beispiel etwa ein Koch. Mit anderen Worten, man benötigt zur Lösung eines Problems einen Algorithmus – also eine Rechenvorschrift – und einen Prozessor.

Während aber bei einem Kochrezept viele Dinge gar nicht explizit gesagt werden müssen, sondern dem Koch aufgrund seiner Erfahrung implizit klar sind – z. B. dass

J. Goll, M. Dausmann, *C als erste Programmiersprache,*
DOI 10.1007/978-3-8348-2271-0_3, © Springer Fachmedien Wiesbaden 2014

er den Kuchen aus dem Backofen holen muss, bevor er schwarz wird –, **muss einem Prozessor alles** explizit und eindeutig durch ein Programm, das aus Anweisungen einer Programmiersprache besteht, **gesagt werden.**

Ein Algorithmus in einer imperativen Programmiersprache sagt einem Prozessor präzise, was dieser tun soll. Ein Programm besteht aus **Anweisungen**, die von einem Prozessor ausgeführt werden können.

Das folgende Bild symbolisiert die Abarbeitung von Anweisungen eines Programms durch den Prozessor:

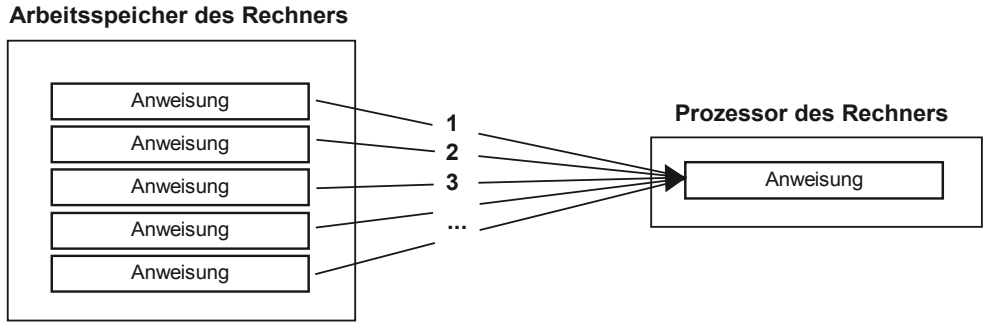


Bild 3-1 Der Prozessor bearbeitet eine Anweisung des Programms nach der anderen

Bild 3-1 zeigt Anweisungen, die im Arbeitsspeicher des Rechners abgelegt sind und nacheinander durch den Prozessor des Rechners abgearbeitet werden.

3.1.1 Der euklidische Algorithmus als Beispiel für Algorithmen

Als Beispiel wird der Algorithmus betrachtet, der von Euklid ca. 300 v. Chr. zur Bestimmung des **größten gemeinsamen Teilers** (**ggT**) zweier natürlicher Zahlen aufgestellt wurde. Der größte gemeinsame Teiler wird zum Kürzen von Brüchen benötigt:

$$\frac{x_{ungekürzt}}{y_{ungekürzt}} = \frac{x_{ungekürzt}\,/ggT(x_{ungekürzt}, y_{ungekürzt})}{y_{ungekürzt}\,/ggT(x_{ungekürzt}, y_{ungekürzt})} = \frac{x_{gekürzt}}{y_{gekürzt}}$$

Hierbei ist $ggT(x_{ungekürzt}, y_{ungekürzt})$ der größte gemeinsame Teiler der beiden Zahlen $x_{ungekürzt}$ und $y_{ungekürzt}$.

Beispiel: $\frac{24}{9} = \frac{24/ggT(24,9)}{9/ggT(24,9)} = \frac{24/3}{9/3} = \frac{8}{3}$

Der euklidische Algorithmus lautet:

Zur Bestimmung des größten gemeinsamen Teilers zwischen zwei natürlichen Zahlen x und y führe die folgenden Schritte aus:

Solange x ungleich y ist, wiederhole:
    Wenn x größer als y ist, dann:
        ziehe y von x ab und weise das Ergebnis x zu.
    Andernfalls:
        ziehe x von y ab und weise das Ergebnis y zu.
Wenn x gleich y ist, dann:
    x (bzw. y) ist der gesuchte größte gemeinsame Teiler.

Man erkennt in diesem Beispiel für einen **Algorithmus** das Folgende:

- Es gibt eine **Menge von Objekten**, mit denen etwas passiert: x und y. Diese Objekte x und y haben am Anfang beliebig vorgegebene Werte, am Schluss enthalten sie den größten gemeinsamen Teiler.
- Es gibt **gewisse Grundoperationen**, die nicht weiter erläutert werden und implizit klar sind: vergleichen, abziehen und zuweisen.
- Es handelt sich um **eine sequenzielle Folge von Anweisungen** (**Operationen**), d. h. die Anweisungen werden der Reihe nach hintereinander ausgeführt.
- Es gibt aber auch bestimmte Konstrukte, welche die einfache sequenzielle Folge (Hintereinanderausführung) gezielt verändern: eine Auswahl zwischen Alternativen (**Selektion**) und eine Wiederholung von Anweisungen (**Iteration**). Diese Konstrukte werden **Kontrollstrukturen** genannt.

Es gibt auch Algorithmen zur Beschreibung von **parallelen Aktivitäten**, die zum gleichen Zeitpunkt nebeneinander ausgeführt werden. Diese Algorithmen werden u. a. bei Betriebssystemen oder in der Prozessdatenverarbeitung benötigt. Im Folgenden werden bewusst nur **sequenzielle Abläufe** behandelt, bei denen zum selben Zeitpunkt nur eine einzige Operation durchgeführt wird.[24]

3.1.2 Beschreibung sequenzieller Abläufe

Die Abarbeitungsreihenfolge von Anweisungen wird auch als **Kontrollfluss** bezeichnet.

Den Prozessor stört es überhaupt nicht, wenn eine Anweisung einen Sprungbefehl zu einer anderen Anweisung enthält. Solche Sprungbefehle werden in manchen Programmiersprachen beispielsweise mit dem Befehl `GOTO` und Marken wie z. B. `100` realisiert:

```
      IF(a > b) GOTO 100
      Anweisungen2
      GOTO 300
100   Anweisungen1
300   Anweisungen3
```

In Worten lauten diese Anweisungen an den Prozessor: „Vergleiche die Werte von `a` und `b`. Wenn[25] `a` größer als `b` ist, springe an die Stelle mit der Marke `100`. Führe an

[24] Threads nach C11 werden in Kapitel 23 behandelt.

[25] „Wenn“ wird ausgedrückt durch das Schlüsselwort `IF` der hier verwendeten Programmiersprache FORTRAN.

der Stelle mit der Marke `100` die Anweisungen `Anweisungen1` aus. Fahre dann mit den Anweisungen `Anweisungen3` fort. Ist aber die Bedingung `a > b` nicht erfüllt, so arbeite die Anweisungen `Anweisungen2` ab. Springe dann zu der Marke `300` und führe die Anweisungen `Anweisungen3` aus."

Will jedoch ein Programmierer ein solches Programm lesen, so verliert er durch die Sprünge sehr leicht den Zusammenhang und damit das Verständnis.

Für den menschlichen Leser ist es am einfachsten, wenn ein Programm einen einfachen und überschaubaren Kontrollfluss hat.

Während typische Programme der sechziger Jahre noch zahlreiche Sprünge enthielten, bemühen sich die Programmierer seit Dijkstras grundlegendem Artikel „Go To Statement Considered Harmful" [Dij68], möglichst einen Kontrollfluss ohne Sprünge zu entwerfen. Beispielsweise kann der oben mit `GOTO` beschriebene Ablauf in C auch folgendermaßen realisiert werden:

```
if (a > b)
   {Anweisungen1}
else
   {Anweisungen2}
Anweisungen3
```

Hierbei ist wieder `if (a > b)` die Abfrage, ob `a` größer als `b` ist. Ist dies der Fall, so werden die Anweisungen `Anweisungen1` ausgeführt. Ist die Bedingung `a > b` nicht wahr, d. h. nicht erfüllt, so werden die Anweisungen `Anweisungen2` des `else`-Zweiges durchgeführt. Damit ist die Fallunterscheidung zu Ende. Unabhängig davon, welcher der Zweige der Fallunterscheidung abgearbeitet wurde, werden nun die Anweisungen `Anweisungen3` ausgeführt. Dieser Ablauf ist in Bild 3-2 grafisch veranschaulicht:

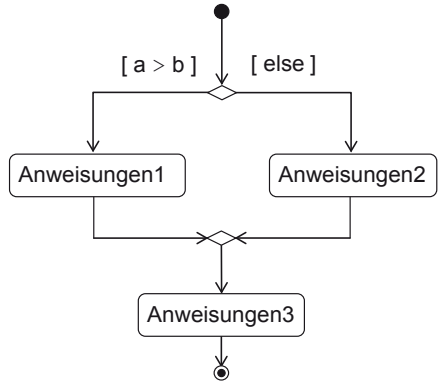


Bild 3-2 Grafische Darstellung der Verzweigung

Im Gegensatz zu der grafischen Darstellung müssen im obigen Programmtext Anfang und Ende einer Anweisungsfolge speziell gekennzeichnet werden. Dazu dienen die geschweiften Klammern, durch welche eine Anweisungsfolge zu einem sogenannten **Block** (engl. **compound statement**) wird. In C kann ein Block überall da stehen, wo auch eine einzelne Anweisung möglich ist.

Unter einer **Kontrollstruktur** versteht man eine Anweisung, welche die Abarbeitungsreihenfolge von Anweisungen beeinflusst.

Zu den **Kontrollstrukturen** gehört die **Fallunterscheidung** (**Selektion**), bei der in Abhängigkeit davon, ob eine Bedingung erfüllt ist oder nicht, entweder die eine oder die andere Anweisung bzw. der eine oder der andere Block abgearbeitet wird. Bei der Kontrollstruktur der **Wiederholung** (**Iteration**) wird eine Anweisung (ein Block) mehrfach aufgerufen. Zu den Kontrollstrukturen zählt auch die sogenannte **Sequenz**. Eine Sequenz ist eine Anweisungsfolge in Form eines Blockes. Wie schon bekannt, ist ein Block von Anweisungen von der Sprachsyntax her als eine einzige Anweisung zu werten.

Betrachtet man nur **sequenzielle Abläufe**, so gibt es **Kontrollstrukturen** für

- die **Selektion**,
- die **Iteration**
- und die **Sequenz**.

Diese Kontrollstrukturen für die sequenziellen Abläufe werden später noch ausführlich erläutert.

Im Beispiel des euklidischen Algorithmus in Kapitel 3.1.1 stellt

> Solange `x` ungleich `y` ist, wiederhole:
> . . . .

eine **Iteration** und

> Wenn `x` größer als `y` ist, dann:
> . . . .
> Andernfalls:
> . . . .

eine **Fallunterscheidung** (**Selektion**) dar.

3.1.3 Programmierung ohne Sprünge

Die Ideen von Dijkstra und anderen fanden ihren Niederschlag in den Regeln für die **Strukturierte Programmierung**. Die Strukturierte Programmierung ist ein programmiersprachenunabhängiges Konzept. Es umfasst die Zerlegung eines Programms in Teilprogramme (Haupt-[26] und Unterprogramme) sowie die folgenden Regeln für den Kontrollfluss:

- Danach gilt, dass in einer **Sequenz** eine Anweisung nach der anderen – d. h. in einer linearen Reihenfolge – abgearbeitet wird.

[26] Mit dem Hauptprogramm wird das Programm gestartet. Das Hauptprogramm kann Unterprogramme aufrufen. Unterprogramme können ebenso Unterprogramme aufrufen.

- Man geht über einen einzigen Eingang (engl. **single entry**), nämlich von der davor stehenden Anweisung in eine Anweisung hinein und geht über einen einzigen Ausgang (engl. **single exit**) aus der Anweisung heraus und kommt direkt zur nächsten Anweisung.

Dies soll durch das folgende Bild symbolisiert werden:

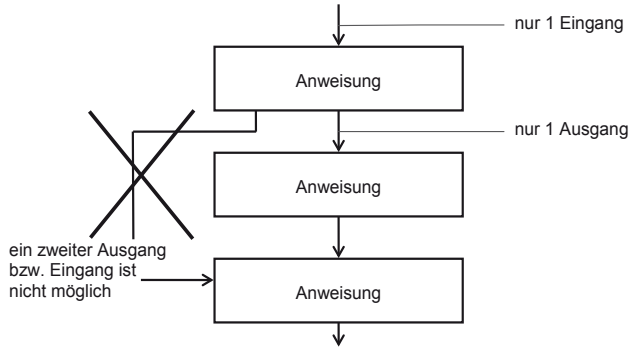


Bild 3-3 Single entry und single exit bei der Sequenz

Haben Kontrollstrukturen für die **Selektion** und **Iteration** die gleichen Eigenschaften wie einzelne Anweisungen, nämlich single entry und single exit, so erhält man für alle Anweisungen einen linearen und damit überschaubaren Programmablauf. Programme, die nur Kontrollstrukturen mit dieser Eigenschaft aufweisen, gehorchen den Regeln der **Strukturierten Programmierung** und können mit Hilfe von **Nassi-Shneiderman-Diagrammen** visualisiert werden (siehe Kapitel 3.2).

Bei der **Strukturierten Programmierung** wird ein Programm in Teilprogramme zerlegt und jede Anweisung hat einen einzigen Eingang und Ausgang.

Mit der Anweisung `GOTO MARKE`, d. h. einer Sprunganweisung, wäre es möglich, die Ausführung eines Programms an einer ganz anderen Stelle, nämlich an der Stelle, an der `MARKE` steht, fortzusetzen. Dies ist aber in der Strukturierten Programmierung nicht zulässig.

3.1.4 Variablen und Zuweisungen

Die durch den Algorithmus von Euklid behandelten Objekte sind natürliche Zahlen. Diese Zahlen sollen jedoch nicht von vornherein festgelegt werden, sondern der Algorithmus soll für die Bestimmung des größten gemeinsamen Teilers beliebiger natürlicher Zahlen verwendbar sein.

Anstelle fest vorgegebener Zahlen werden im Programm Bezeichner verwendet, die einem Speicherplatz im Arbeitsspeicher oder einem Register zugeordnet sind und als **variable Größen** oder kurz **Variablen** bezeichnet werden. Den Variablen werden im Verlauf des Algorithmus konkrete Werte zugewiesen.

Diese Wertzuweisung an Variablen ist eine der grundlegenden Operationen, die ein Prozessor ausführen können muss. Auf Variablen wird noch ausführlicher in Kapitel 7.3 eingegangen.

Der im obigen Beispiel beschriebene Algorithmus kann auch von einem menschlichen „Prozessor“ ausgeführt werden – andere Möglichkeiten hatten die Griechen in der damaligen Zeit nicht. Als Hilfsmittel braucht man dazu Papier und Bleistift, um die Zustände der Variablen – im obigen Beispiel die Zustände der Variablen `x` und `y` – zwischen den Verarbeitungsschritten festzuhalten. Man erhält dann eine Tabelle, die auch **Trace-Tabelle**[27] genannt wird und für die Zahlen `x` gleich `24` und `y` gleich `9` das folgende Aussehen hat:

Verarbeitungsschritt	Werte von x	Werte von y
Initialisierung `x = 24, y = 9`	24	9
`x = x - y`	15	9
`x = x - y`	6	9
`y = y - x`	6	3
`x = x - y`	3	3
Ergebnis: ggT = 3		

Tabelle 3-1 Trace der Variableninhalte für die Initialwerte `x` ist gleich 24 und `y` ist gleich `9`

Diese Tabelle zeigt sehr deutlich die **Funktion der Variablen** auf:

Variablen speichern Werte. Sie repräsentieren über den Verlauf des Algorithmus hinweg ganz unterschiedliche Werte.

Zu Beginn eines Algorithmus werden den Variablen definierte Anfangs- oder Startwerte zugewiesen. Diesen Vorgang bezeichnet man als **Initialisierung** der Variablen.

Die **Werteänderung** von Variablen erfolgt durch sogenannte **Zuweisungen**.

Als **Zuweisungssymbol** wird hier das Gleichheitszeichen der natürlichen Sprache (=) benutzt, wie es in C üblich ist.

[27] Mit der Trace-Tabelle verfolgt man die Zustände (Werte) der Variablen.

Für eine andere Ausgangssituation sieht die Trace-Tabelle beispielsweise so aus:

Verarbeitungsschritt	Werte von x	Werte von y
Initialisierung		
`x = 5, y = 3`	5	3
`x = x - y`	2	3
`y = y - x`	2	1
`x = x - y`	1	1
Ergebnis: ggT = 1		

Tabelle 3-2 Trace der Variableninhalte für die Initialwerte `x` ist gleich `5` und `y` ist gleich `3`

Die Schreibweise `x = x - y` ist zunächst etwas verwirrend. Diese Schreibweise ist nicht als mathematische Gleichung zu sehen, sondern meint etwas ganz anderes: Auf der rechten Seite des Gleichheitszeichens steht ein arithmetischer Ausdruck, dessen Wert zuerst berechnet werden soll. Dieser so berechnete Wert wird dann in einem zweiten Schritt der Variablen zugewiesen, deren Name auf der linken Seite steht. Im Beispiel also:

- Nimm den aktuellen Wert von x. Nimm den aktuellen Wert von y. Ziehe den Wert von y vom Wert von x ab.
- Der neue Wert von `x` ist die soeben ermittelte Differenz von `x` und `y`.

Eine Zuweisung verändert den Wert der Variablen, also den Zustand der Variablen, auf der linken Seite.

Bei einer **Zuweisung** wird zuerst der Ausdruck rechts vom Zuweisungsoperator berechnet und der Wert dieses Ausdrucks dem Speicherplatz auf der linken Seite des Zuweisungsoperators zugewiesen.

Die Beispiele in diesem Kapitel zeigen, wie ein Algorithmus sequenzielle Abläufe und Transformationen seiner Variablen beschreibt. Solche Transformationen ändern die Werte der Variablen ab und ändern also den Zustand der Variablen. Daher werden solche Transformationen auch als **Zustandstransformationen** bezeichnet. Wird derselbe Algorithmus zweimal durchlaufen, wobei die Variablen am Anfang unterschiedliche Werte haben, dann erhält man in aller Regel auch unterschiedliche Abläufe. Sie folgen aber ein und demselben Verhaltensmuster, das durch den Algorithmus beschrieben ist.

3.2 Entwurf mit Nassi-Shneiderman-Diagrammen

Zur Visualisierung des Kontrollflusses von Programmen – das heißt, zur grafischen Veranschaulichung ihres Ablaufs – wurden 1973 von Nassi und Shneiderman [Nas73] grafische Strukturen, die sogenannten **Struktogramme** (siehe DIN 66261 [DIN85]), vorgeschlagen. Diese Struktogramme werden nach ihren Urhebern oftmals auch als **Nassi-Shneiderman-Diagramme** bezeichnet.

Nassi-Shneiderman-Diagramme enthalten kein `GOTO`, sondern nur die Sprachmittel der **Strukturierten Programmierung**, d. h. die **Bildung von Teilprogrammen** und die **Kontrollstrukturen Sequenz, Iteration** und **Selektion**.

Entwirft man Programme mit Nassi-Shneiderman-Diagrammen, so genügt man automatisch den Regeln der Strukturierten Programmierung. Nassi und Shneiderman schlugen ihre Struktogramme als Ersatz für die bis dahin üblichen **Flussdiagramme** (siehe DIN 66001 [DIN83]) vor. Traditionelle Flussdiagramme erlauben einen Kontrollfluss mit beliebigen Sprüngen in einem Programm.

Spezifiziert und programmiert man strukturiert, so geht der Kontrollfluss eines solchen Programmes einfach von oben nach unten – eine Anweisung folgt der nächsten. Wilde Sprünge, welche die Übersicht erschweren, sind nicht zugelassen.

Im Folgenden werden die Diagramme für die Sequenz, Selektion und Iteration in abstrakter Form, d. h. ohne Notation in einer speziellen Programmiersprache, vorgestellt. Die Kontrollstrukturen für die Selektion und Iteration können – wie von Nassi und Shneiderman vorgeschlagen – in grafischer Form oder alternativ auch mit Hilfe eines Pseudocodes dargestellt werden. Was ein Pseudocode ist, wird in Kapitel 3.3 erläutert.

Das wichtigste Merkmal der Struktogramme ist, dass jeder **Verarbeitungsschritt** durch ein **rechteckiges Sinnbild** dargestellt wird. Ein Verarbeitungsschritt kann dabei eine Anweisung oder eine Gruppe von zusammengehörigen Anweisungen sein.

Das folgende Symbol zeigt das Sinnbild für einen Verarbeitungsschritt nach Nassi-Shneiderman:

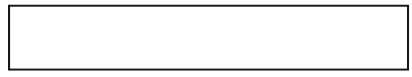


Bild 3-4 Sinnbild für einen Verarbeitungsschritt

Die obere Linie eines Rechtecks bedeutet den Beginn eines Verarbeitungsschrittes, die untere Linie bedeutet das Ende dieses Verarbeitungsschrittes.

Jedes Sinnbild erhält eine Innenbeschriftung, die den Verarbeitungsschritt näher beschreibt.

3.2.1 Diagramme für die Sequenz

Bei der **Sequenz** folgen Verarbeitungsschritte sequentiell hintereinander.

Dies wird durch ein Nassi-Shneiderman-Diagramm für die Verarbeitungsschritte V1 und V2 wie folgt dargestellt:

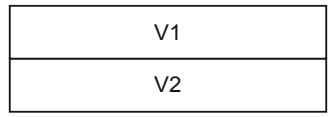


Bild 3-5 Nassi-Shneiderman-Diagramm für die Sequenz

Eine Kontrollstruktur für die Sequenz ist der **Block**.

Ein **Block** stellt eine Folge logisch zusammenhängender Verarbeitungsschritte dar. Er kann einer **Methode** oder **Funktion**[28] in einer Programmiersprache entsprechen, kann aber auch nur einfach **mehrere Verarbeitungsschritte** unter einem einzigen Namen zusammenfassen. Das folgende Bild visualisiert den Block:

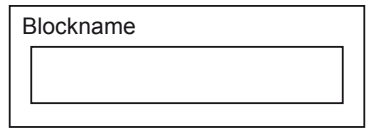


Bild 3-6 Sinnbild für einen Block

Wie in Bild 3-6 zu sehen ist, wird der Name des Blockes im Diagramm den Verarbeitungsschritten vorangestellt.

Das Diagramm Bild 3-7 stellt das „Hello World“-Programm aus Kapitel 1.1 in grafischer Form dar:

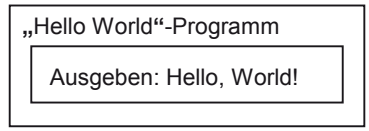


Bild 3-7 Einfaches Beispiel eines Struktogramms

Aus der Darstellung ist zu entnehmen, dass die Details einer Programmiersprache auf dieser Abstraktionsebene keine Rolle spielen.

3.2.2 Diagramme für die Selektion

Bei den Kontrollstrukturen für die Selektion kann man zwischen der **einfachen Alternative**, der **bedingten Verarbeitung** und der **mehrfachen Alternative** unterscheiden.

[28] Anweisungsfolgen, die unter einem Namen aufgerufen werden können, heißen in der objektorientierten Programmierung „Methoden“, in der klassischen Programmierung „Funktionen“ – wie z. B. in C – oder auch „Prozeduren“.

Die **einfache Alternative** stellt eine Verzweigung im Programmablauf dar. Das entsprechende Struktogramm ist in Bild 3-8 zu sehen:

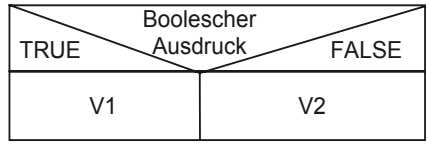


Bild 3-8 Struktogramm für die einfache Alternative

Ein **boolescher Ausdruck** wie z. B. `a > b` kann die Wahrheitswerte `TRUE`[29] bzw. `FALSE` annehmen. Ein solcher boolescher Ausdruck wird auch als **Bedingung** bezeichnet.

Bei der **einfachen Alternative** wird überprüft, ob ein **boolescher Ausdruck**[30] wahr ist oder nicht. Je nach dem Wert der Bedingung wird etwas anderes getan.

Ist der Ausdruck wahr, so wird der Zweig für `TRUE` ausgewählt und der Verarbeitungsschritt `V1` ausgeführt. Ist der Ausdruck nicht wahr, so wird der `FALSE`-Zweig ausgewählt und der Verarbeitungsschritt `V2` durchgeführt. Jeder dieser Zweige kann einen Verarbeitungsschritt bzw. einen Block von Verarbeitungsschritten enthalten.

Hier der Pseudocode für eine solche Verzweigung:

```
if (a > b) V1
else V2
```

Bei der **bedingten Verarbeitung** erfolgt ein Verarbeitungsschritt, wenn die Bedingung erfüllt ist.

Bei der **bedingten Verarbeitung** (siehe Bild 3-9) wird der `TRUE`-Zweig ausgewählt, wenn der Ausdruck wahr ist. Ansonsten wird direkt zu dem nächsten Verarbeitungsschritt übergegangen. Das folgende Bild visualisiert die bedingte Verarbeitung:

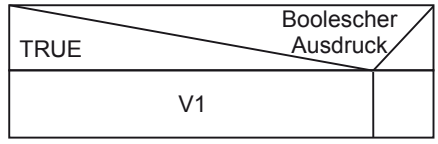


Bild 3-9 Struktogramm für die bedingte Verarbeitung

Der Pseudocode für die bedingte Verarbeitung entspricht dem der einfachen Alternative, allerdings fehlt der sogenannte `else`-Teil:

```
if (a > b) V1
```

[29] `TRUE` und `FALSE` sind Pseudocode-Symbole.

[30] Ein **Ausdruck** ist eine Verknüpfung von Operanden durch Operatoren und runden Klammern (siehe Kapitel 9).

Bei der **mehrfachen Alternative** wird je nach dem Wert eines arithmetischen Ausdrucks jeweils der entsprechende Zweig ausgeführt.

Bei der **mehrfachen Alternative** (siehe Bild 3-10) wird geprüft, ob ein **arithmetischer Ausdruck**[31] einen von n vorgegebenen Werten c_1 ... c_n annimmt. Ist dies der Fall, so wird der entsprechende Zweig ausgeführt, ansonsten wird direkt zu dem nächsten Verarbeitungsschritt übergegangen. Das folgende Bild visualisiert die mehrfache Alternative:

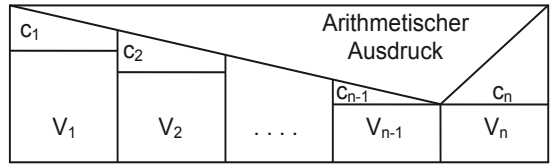


Bild 3-10 Struktogramm für die mehrfache Alternative

Der entsprechende Pseudocode ist relativ komplex. Daher wird auf eine Darstellung hier verzichtet.

3.2.3 Diagramme für die Iteration

Bei der Iteration kann man drei Fälle von Kontrollstrukturen unterscheiden:

- Wiederholung mit vorheriger Prüfung,
- Wiederholung mit nachfolgender Prüfung
- und Wiederholung ohne Prüfung.

Bei der **Wiederholung mit vorheriger Prüfung** (**abweisende Schleife**[32]) wird zuerst eine Bedingung geprüft. Solange diese Bedingung erfüllt ist, wird der Verarbeitungsschritt wiederholt.

Ist diese Bedingung bereits zu Anfang nicht erfüllt, wird der Verarbeitungsschritt `V` nicht ausgeführt – die Ausführung der Schleife wird abgewiesen.

Das Struktogramm einer abweisenden Schleife ist in Bild 3-11 dargestellt:

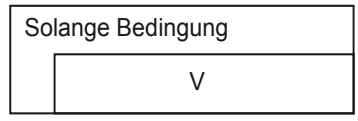


Bild 3-11 Struktogramm der Wiederholung mit vorausgehender Bedingungsprüfung

[31] Bei einem arithmetischen Ausdruck werden arithmetische Operatoren auf die Operanden angewandt, wie z. B. der binäre Minusoperator im Ausdruck `6 - 2` auf die Operanden `6` und `2`.

[32] Synonym für „abweisende Schleife“ wird auch der Ausdruck „kopfgesteuerte Schleife“ verwendet.

In einem Pseudocode kann man eine abweisende Schleife folgendermaßen darstellen:

```
WHILE (Bedingung) DO V
```

Hat zu Beginn der Schleife die Bedingung `Bedingung` den Wert `TRUE`, dann müssen die Verarbeitungsschritte in der Schleife dafür sorgen, dass der Wert der Bedingung irgendwann `FALSE` wird, sonst entsteht eine Endlosschleife[33]. Die `FOR`-Schleife (siehe auch Kapitel 10.3.2) ist ebenfalls eine abweisende Schleife. Sie stellt eine spezielle Ausprägung der `WHILE`-Schleife dar. `FOR`-Schleifen bieten eine syntaktische Beschreibung des Startzustands und der Iterationsschritte (z. B. Hoch- oder Herunterzählen einer Laufvariablen, welche die einzelnen Iterationsschritte durchzählt).

Bei der **Wiederholung mit nachfolgender Prüfung** (**annehmende Schleife**[34]) erfolgt die Prüfung der Bedingung erst am Ende.

Das zugehörige Struktogramm ist in Bild 3-12 dargestellt:

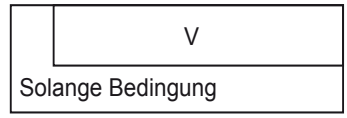


Bild 3-12 Struktogramm der Wiederholung mit nachfolgender Bedingungsprüfung

Die annehmende Schleife kann man in einem Pseudocode folgendermaßen darstellen:

```
DO V WHILE (Bedingung)
```

Die annehmende Schleife wird mindestens einmal durchgeführt. Erst dann wird die Bedingung bewertet. Die `DO-WHILE`-Schleife wird typischerweise dann benutzt, wenn der Wert der Bedingung erst in der Schleife entsteht, beispielsweise wie in der folgenden Anwendung „Lies Zahlen ein, solange keine `0` eingegeben wird“. Hier muss zuerst eine Zahl eingelesen werden. Erst dann kann geprüft werden, ob sie `0` ist oder nicht.

Prüfungen müssen nicht immer notwendigerweise zu Beginn oder am Ende stattfinden. Eine Bedingung muss manchmal auch in der Mitte der Verarbeitungsschritte einer Schleife geprüft werden. Zu diesem Zweck gibt es die **Wiederholung ohne Prüfung**.

Bei einer **Wiederholung ohne Prüfung** ist die Prüfung in den Verarbeitungsschritten „versteckt“.

[33] Eine Endlosschleife ist eine Schleife, deren Ausführung nie abbricht.
[34] Synonym für „annehmende Schleife“ wird auch der Ausdruck „fußgesteuerte Schleife“ verwendet.

Das Struktogramm für eine Wiederholung ohne Prüfung ist in Bild 3-13 dargestellt:

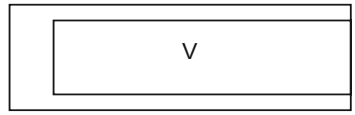


Bild 3-13 Struktogramm der Wiederholung ohne Bedingungsprüfung

In einem Pseudocode kann die Schleife ohne Bedingungsprüfung folgendermaßen angegeben werden:

```
LOOP V
```
[35]

Die Schleife ohne Bedingungsprüfung wird verlassen, wenn in einem der Verarbeitungsschritte `V` eine `BREAK`-Anweisung ausgeführt wird.

Eine `BREAK`-Anweisung ist eine spezielle Sprunganweisung und sollte nur eingesetzt werden, damit bei einer Schleife ohne Wiederholungsprüfung keine Endlosschleife entsteht. Die Regel, dass eine Kontrollstruktur nur einen einzigen Eingang und einen einzigen Ausgang hat, wird dadurch nicht verletzt, sondern der zunächst fehlende Ausgang wird erst durch die `BREAK`-Anweisung zur Verfügung gestellt.

Bild 3-14 zeigt das Sinnbild für eine solche Abbruchanweisung:

Bild 3-14 Abbruchanweisung

Im Falle der Programmiersprache C sind die Kontrollstrukturen der Wiederholung mit vorheriger Prüfung und mit nachfolgender Prüfung als Sprachkonstrukt vorhanden, d. h., es gibt in C Anweisungen für diese Schleifen. Eine Wiederholung ohne Prüfung kann in C auch formuliert werden, aber es gibt keine spezielle Anweisung dafür. Bild 3-15 stellt ein Beispiel für eine Schleife ohne Wiederholungsprüfung mit Abbruchanweisung dar:

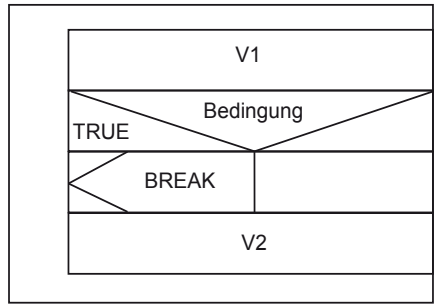


Bild 3-15 Struktogramm einer Schleife ohne Wiederholungsprüfung mit Abbruchbedingung

[35] Die Sprache C enthält kein spezielles Schlüsselwort für eine Endlosschleife. In dem hier vorgeschlagenen Pseudocode wird hierfür das Schlüsselwort `LOOP` verwendet. Endlosschleifen in C werden in Kapitel 10.3.2 vorgestellt.

Nach der Ausführung des Verarbeitungsschritts `V1` wird die Bedingung geprüft. Hat die Bedingung nicht den Wert `TRUE`, so wird der Verarbeitungsschritt `V2` abgearbeitet und dann die Schleife mit dem Verarbeitungsschritt `V1` beginnend wiederholt. Der Durchlauf der Schleife mit der Reihenfolge „Ausführung `V1`“, „Bedingungsprüfung“, „Ausführung `V2`“ wird solange wiederholt, bis die Bedingung den Wert `TRUE` hat. In diesem Fall wird die Schleife durch die Abbruchbedingung verlassen.

3.2.4 Vom Struktogramm zum Programm

Mit den Mitteln der Struktogramme kann nun der Algorithmus von Euklid, der in Kapitel 3.1.1 eingeführt wurde, in grafischer Form dargestellt werden:

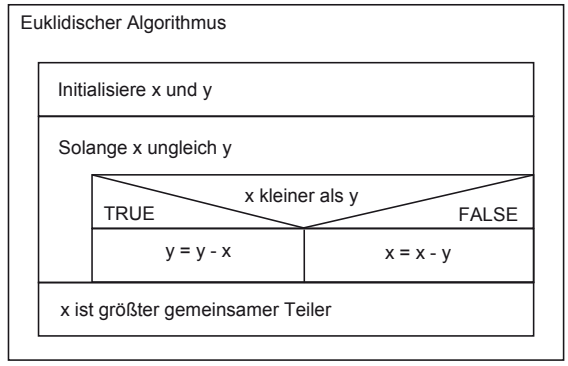


Bild 3-16 Struktogramm des euklidischen Algorithmus

In diesem Struktogramm ist ein Block mit dem Namen „Euklidischer Algorithmus“ zu sehen, der eine Folge von drei Anweisungen enthält: zuerst kommt die Initialisierung der Variablen, dann eine Wiederholung mit vorheriger Prüfung und am Ende die Feststellung des Ergebnisses. Die Wiederholungsanweisung enthält eine einfache Alternative mit der Bedingung `x kleiner als y`.

Der in Bild 3-16 dargestellte Algorithmus von Euklid kann natürlich auch mit Hilfe eines Pseudocodes formuliert werden:

```
Euklidischer Algorithmus
{
   Initialisiere x und y
   while (x ungleich y)
   {
      if (x kleiner als y)
         y = y - x
      else
         x = x - y
   }
   x ist der größte gemeinsame Teiler
}
```

Dieser Pseudocode und die grafische Darstellung als Nassi-Shneiderman-Diagramm in Bild 3-16 sind äquivalent.

3.2.4.1 Das Programm von Euklid in C mit fest vorgegebenen Anfangswerten

Der Schritt zu einem lauffähigen Programm ist nun überhaupt nicht mehr groß: Das, was soeben noch umgangssprachlich formuliert wurde, muss jetzt in C ausgedrückt werden. Die wesentlichen Punkte betreffen hier die Kommunikation des Programms mit seinem Benutzer: Welche Werte sollen `x` und `y` bekommen und in welcher Form soll das Ergebnis dargestellt werden? Da die Details der Ein- und Ausgabe von Daten in C-Programmen an dieser Stelle nicht behandelt werden sollen, arbeitet das folgende C-Programm für den euklidischen Algorithmus mit fest vorgegebenen Werten für `x` und `y`:

```
/* Datei: euklidconst.c */
#include <stdio.h>

int main (void)
{
   int x = 24;
   int y = 9;

   while (x != y)
   {
      if (x < y)
         y = y - x;
      else
         x = x - y;
   }
   printf ("Der groesste gemeinsame Teiler ist: %d\n", x);
   return 0;
}
```

Die Ausgabe des Programms ist:

```
Der groesste gemeinsame Teiler ist: 3
```

Viele Elemente dieses Programms sind bereits beispielsweise durch das „Hello World“-Programm aus Kapitel 1.1 oder durch den Pseudocode für den euklidischen Algorithmus bekannt. Es werden hier zwei Variablen `x` und `y` vereinbart, die ganze Zahlen speichern können und jeweils mit einem vorgegebenen Wert initialisiert werden:

```
int x = 24;
int y = 9;
```

Die Abfragen, ob Werte `ungleich` sind bzw. ob ein Wert `kleiner als` ein anderer ist, werden in C durch die Operatoren `!=` bzw. `<` ausgedrückt.

An dieser Stelle soll nicht weiter auf die Details der Ein- und Ausgabe eingegangen werden. Daher ist das vorgestellte Programm für den euklidischen Algorithmus zwar lauffähig, aber in der Benutzung sehr ineffizient. Denn wenn man den größten gemeinsamen Teiler von anderen Zahlen als den vorgegebenen bestimmen will, muss man das Programm an den entsprechenden Stellen ändern und neu übersetzen. In Kapitel 3.2.4.2 befindet sich eine elegantere Version des euklidischen

Algorithmus als C-Programm, bei dem die Inhalte der Variablen x und y im Dialog eingelesen werden.

3.2.4.2 Praktikables Euklidisches Programm mit Einlesen von x und y

Das C-Programm für den euklidischen Algorithmus aus Kapitel 3.2.4.1 soll nun so modifiziert werden, dass es für einen Benutzer besser zu bedienen ist. Dabei soll zum einen die Initialisierung von x und y durch das Einlesen von der Tastatur und nicht hart codiert im Programm erfolgen, zum anderen sollen der Startwert von x, der Startwert von y sowie der größte gemeinsame Teiler ausgegeben werden. Das Programm hat die Form:

```
/* Datei: euklid.c */
#include <stdio.h>

int main (void)
{
   int x;
   int y;

   printf ("\nGeben Sie bitte einen Wert fuer x ein: ");
   scanf ("%d", &x);
   printf ("Geben Sie bitte einen Wert fuer y ein: ");
   scanf ("%d", &y);
   printf ("Der ggT von %d und %d ist: ", x, y);
   while (x != y)
   {
      if (x < y)
         y = y - x;
      else
         x = x - y;
   }
   printf ("%d\n", x);
   return 0;
}
```

Man beachte hierbei die folgenden Punkte:

- Mit Hilfe der Bibliotheksfunktion `scanf()` kann man Werte einlesen. So kann man beispielsweise auf die Ausgabe:

  ```
  Geben Sie bitte einen Wert fuer x ein:
  ```

  antworten durch die Eingabe:

  ```
  72 <RETURN>
  ```

  Dabei bedeutet `<RETURN>` den Anschlag der Return-Taste (↵). Der eingegebene Wert wird dann von `scanf()` in der Variablen x abgelegt. Eine Besonderheit von `scanf()` ist, dass diese Funktion als Argument nicht die Variable selbst, in die ein Wert eingelesen werden soll, haben möchte, sondern die Adresse dieser Variablen. Im Vorgriff auf Kapitel 8 wird hier einfach angegeben, dass die Adresse einer Variablen x ermittelt wird, indem man vor diese Variable den Adressoperator & schreibt.

- Die Funktion `scanf()` hat wie die Funktion `printf()` als erstes Argument einen **Formatstring** (**Steuerstring**). Da in eine `int`-Variable eingelesen wird, wird das Formatelement `%d` benutzt.
- Da der Wert der Variablen `x` bzw. `y` durch den Algorithmus verändert wird, ist es erforderlich, den eingegebenen Anfangswert von `x` und `y` frühzeitig auszugeben.
- Die Schnittstelle der Funktion `scanf()` ist ebenfalls wie die Schnittstelle der Funktion `printf()` in der Header-Datei `<stdio.h>` zu finden.

Ein möglicher Dialog mit diesem Programm ist:

```
Geben Sie bitte einen Wert fuer x ein: 72
Geben Sie bitte einen Wert fuer y ein: 45
Der ggT von 72 und 45 ist: 9
```

3.3 Pseudocode

Kapitel 3.3.1 behandelt natürliche und formale Sprachen. Kapitel 3.3.2 geht anschließend auf einen freien und formalen Pseudocode ein.

3.3.1 Natürliche und formale Sprachen

Generell kann man bei Sprachen zwischen **natürlichen Sprachen** wie der Umgangssprache oder den Fachsprachen einzelner Berufsgruppen und **formalen Sprachen** unterscheiden.

Formale Sprachen sind beispielsweise die Notenschrift in der Musik, die Formelschrift in der Mathematik oder Programmiersprachen beim Computer. Nur das, was durch eine formale Sprache – hier die Programmiersprache – festgelegt ist, ist für den Übersetzer verständlich.

3.3.2 Freier und formaler Pseudocode

Hat man als Programmierer ein neues Problem zu lösen, dann ist es empfehlenswert, nicht gleich auf der Ebene einer formalen Programmiersprache zu beginnen. Würde man das tun, dann müsste man sich sofort mit den vielen formalen Details der Programmiersprache befassen, statt erst nachzudenken, wie der Lösungsweg denn aussehen soll.

Wenn ein Programm entworfen, also quasi „erfunden“ werden soll, kann man grafische Mittel wie Struktogramme nutzen oder man greift auf eine „halbformale“ Sprache, einen sogenannten freien Pseudocode, zurück, der es erlaubt, zuerst die Struktur eines Programms festzulegen und die Details aufzuschieben.

Ein **Pseudocode** ist eine Sprache, die dazu dient, Programme zu entwerfen. Pseudocode kann von einem freien Pseudocode bis zu einem formalen Pseudocode reichen.

Bei einem **freien Pseudocode** formuliert man Schlüsselwörter für die Iteration, Selektion und Blockbegrenzer und fügt in diesen Kontrollfluss Verarbeitungsschritte ein, die in der Umgangssprache beschrieben werden.

Ein **formaler Pseudocode**, der alle Elemente enthält, die auch in einer Programmiersprache enthalten sind, ermöglicht eine automatische Codegenerierung für diese Zielsprache. Dennoch ist es das eigentliche Ziel eines Pseudocodes, eine Spezifikation zu unterstützen.

Freie Pseudocodes sind für eine grobe Spezifikation vollkommen ausreichend.

Das Beispiel in Kapitel 3.1.1 war bereits in einem freien Pseudocode formuliert. Es wurden deutsche Schlüsselwörter wie „solange“, „wenn“ oder „andernfalls“ benutzt. Meist lehnt man sich aber bei einem Pseudocode an eine bekannte Programmiersprache an, sodass dann englische Begriffe dominierend sind.

3.4 Zusammenfassung

Dieses Kapitel behandelt den Entwurf von Programmen.

Kapitel **3.1** stellt Algorithmen und ihre Abläufe vor. Algorithmen sind Vorschriften für die Lösung eines Problems, welche die Handlungen und ihre Abfolge – also die Handlungsweise – beschreiben. Ein Algorithmus in einer imperativen Programmiersprache sagt einem Prozessor präzise, was dieser tun soll. Ein Programm besteht aus Anweisungen, die von einem Prozessor ausgeführt werden können.

Die Abarbeitungsreihenfolge von Anweisungen wird auch als Kontrollfluss bezeichnet. Für den menschlichen Leser ist es am einfachsten, wenn ein Programm einen einfachen und überschaubaren Kontrollfluss hat. Unter einer Kontrollstruktur versteht man eine Anweisung, welche die Abarbeitungsreihenfolge von Anweisungen beeinflusst.

Betrachtet man nur sequenzielle Abläufe, so gibt es Kontrollstrukturen für

- die Selektion,
- die Iteration
- und die Sequenz.

Bei der Strukturierten Programmierung wird ein Programm in Teilprogramme zerlegt und jede Anweisung hat einen einzigen Eingang und Ausgang.

Anstelle fest vorgegebener Zahlen werden im Programm Bezeichner verwendet, die einem Speicherplatz im Arbeitsspeicher oder einem Register zugeordnet sind und als variable Größen oder kurz Variablen bezeichnet werden. Den Variablen werden im

Verlauf des Algorithmus konkrete Werte zugewiesen. Variablen speichern also Werte. Die Variablen repräsentieren über den Verlauf eines Algorithmus hinweg ganz unterschiedliche Werte. Zu Beginn des Algorithmus werden den Variablen definierte Anfangs- oder Startwerte zugewiesen. Diesen Vorgang bezeichnet man als Initialisierung der Variablen. Die Werteänderung von Variablen erfolgt dann durch sogenannte Zuweisungen. Der von einer Variablen gespeicherte Wert kann durch eine Zuweisung mit einem neuen Wert überschrieben werden. Bei einer Zuweisung wird zuerst der Ausdruck rechts vom Zuweisungsoperator berechnet und der Wert dieses Ausdrucks dem Speicherplatz auf der linken Seite des Zuweisungsoperators zugewiesen.

Kapitel **3.2** erläutert die Veranschaulichung des Ablaufs von Programmen mittels Nassi-Shneiderman-Diagrammen, welche auch unter der Bezeichnung Struktogramme bekannt sind. Nassi-Shneiderman-Diagramme enthalten kein GOTO, sondern nur die Sprachmittel der Strukturierten Programmierung, d. h. die Bildung von Teilprogrammen und die Kontrollstrukturen Sequenz, Iteration und Selektion. Spezifiziert und programmiert man strukturiert, so geht der Kontrollfluss eines solchen Programmes einfach von oben nach unten – eine Anweisung folgt der nächsten. Wilde Sprünge, welche die Übersicht erschweren, sind nicht zugelassen. Das wichtigste Merkmal der Struktogramme ist, dass jeder Verarbeitungsschritt durch ein rechteckiges Sinnbild dargestellt wird. Ein Verarbeitungsschritt kann dabei eine Anweisung oder eine Gruppe von zusammengehörigen Anweisungen sein. Die obere Linie eines Rechtecks bedeutet den Beginn eines Verarbeitungsschrittes, die untere Linie bedeutet das Ende dieses Verarbeitungsschrittes.

Bei der Sequenz folgen Verarbeitungsschritte sequentiell hintereinander. Eine Kontrollstruktur für die Sequenz ist der Block.

Bei den Kontrollstrukturen für die Selektion kann man zwischen

- der einfachen Alternative,
- der bedingten Verarbeitung
- und der mehrfachen Alternative

unterscheiden.

Ein boolescher Ausdruck wie z. B. `a > b` kann die Wahrheitswerte „wahr“ bzw. „falsch“ annehmen. Ein solcher boolescher Ausdruck wird auch als Bedingung bezeichnet.

Bei der einfachen Alternative wird überprüft, ob ein boolescher Ausdruck wahr ist oder nicht. Je nach dem Wert der Bedingung wird etwas anderes getan. Bei der bedingten Verarbeitung erfolgt ein Verarbeitungsschritt, wenn die Bedingung erfüllt ist. Bei der mehrfachen Alternative wird je nach dem Wert eines arithmetischen Ausdrucks jeweils der entsprechende Zweig ausgeführt.

Bei der Iteration kann man drei Fälle von Kontrollstrukturen unterscheiden:

- Wiederholung mit vorheriger Prüfung,
- Wiederholung mit nachfolgender Prüfung
- und Wiederholung ohne Prüfung.

Bei der Wiederholung mit vorheriger Prüfung (abweisende oder kopfgesteuerte Schleife) wird zuerst eine Bedingung geprüft. Solange diese Bedingung erfüllt ist, wird der Verarbeitungsschritt wiederholt. Bei der Wiederholung mit nachfolgender Prüfung (annehmende oder fußgesteuerte Schleife) erfolgt die Prüfung der Bedingung erst am Ende. Bei einer Wiederholung ohne Prüfung ist die Prüfung in den Verarbeitungsschritten „versteckt“.

Eine `BREAK`-Anweisung ist eine spezielle Sprunganweisung und sollte nur eingesetzt werden, damit bei einer Schleife ohne Wiederholungsprüfung keine Endlosschleife entsteht. Die Regel, dass eine Kontrollstruktur nur einen einzigen Eingang und einen einzigen Ausgang hat, wird dadurch nicht verletzt, sondern der zunächst fehlende Ausgang wird erst durch die `BREAK`-Anweisung zur Verfügung gestellt.

Kapitel **3.3** diskutiert die Möglichkeiten des Pseudocodes. Ein Pseudocode ist eine Sprache, die dazu dient, Programme zu entwerfen. Pseudocode kann von einem freien Pseudocode bis zu einem formalen Pseudocode reichen. Bei einem freien Pseudocode formuliert man Schlüsselwörter für die Iteration, Selektion und Blockbegrenzer und fügt in diesen Kontrollfluss Verarbeitungsschritte ein, die in der Umgangssprache beschrieben werden. Ein formaler Pseudocode, der alle Elemente enthält, die auch in einer Programmiersprache enthalten sind, ermöglicht eine automatische Codegenerierung für diese Zielsprache. Dennoch ist es das eigentliche Ziel eines Pseudocodes, eine Spezifikation zu unterstützen. Freie Pseudocodes sind für eine grobe Spezifikation vollkommen ausreichend.

3.5 Übungsaufgaben

Aufgabe 3.1: Nassi-Shneiderman-Diagramm Quadratzahlen

Vervollständigen Sie die unten angegebenen Nassi-Shneiderman-Diagramme für ein Programm, welches in einer (**äußeren**) **Schleife** ganze Zahlen in eine Variable `n` einliest. Die Reaktion des Programms soll davon abhängen, ob der in die Variable eingelesene Wert positiv, negativ oder gleich null ist. Treffen Sie die folgende Fallunterscheidung:

- Ist die eingelesene Zahl `n` größer als null, so soll in einer inneren Schleife ausgegeben werden(die Titel müssen Sie nicht ausgeben):

```
Zahl          Quadratzahl
1               1
2               4
.               .
.               .
.               .
n               n*n
```

- Ist die eingelesene Zahl `n` kleiner als null, so soll ausgegeben werden:

```
Negative Zahl
```

- Ist die eingegebene ganze Zahl `n` gleich null, so soll das Programm (**die äußere Schleife**) abbrechen.

Lösen Sie die Aufgabe auf zwei Arten:

a) Mit einer Wiederholung ohne Prüfung.
b) Mit einer Wiederholung mit nachfolgender Prüfung (annehmende Schleife).

Nutzen Sie dazu folgende zwei Diagramme:

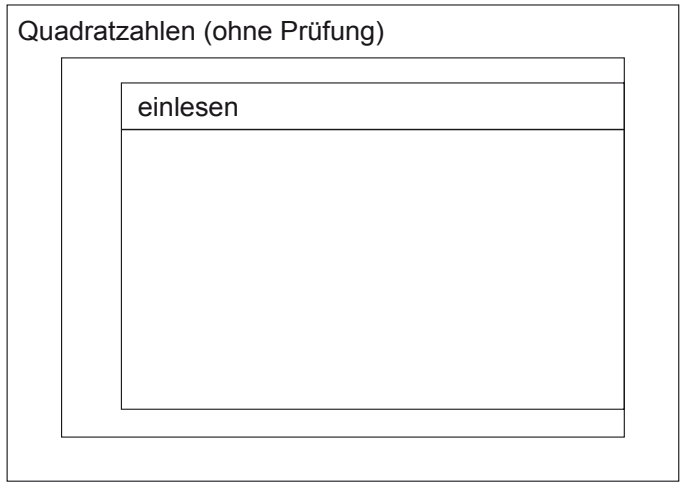


Bild 3-17 Nassi-Shneiderman-Diagramm für das Programm Quadratzahlen mit einer Wiederholung ohne Prüfung

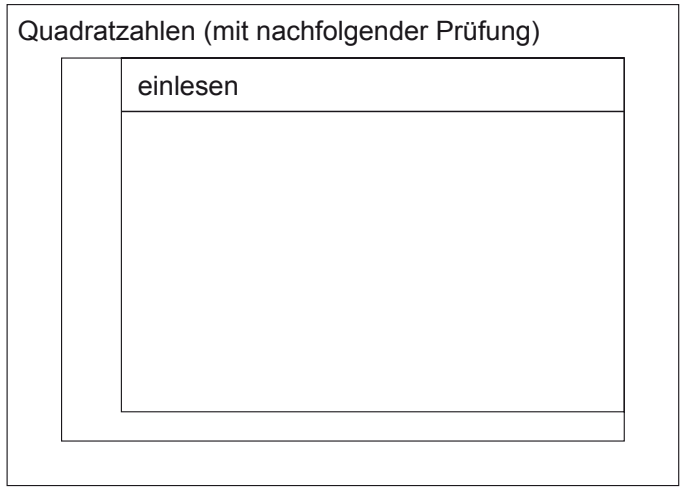


Bild 3-18 Nassi-Shneiderman-Diagramm für das Programm Quadratzahlen mit einer Wiederholung mit nachfolgender Prüfung