Unterlagen zum Visual Basic for Applications-Kurs

Version 0.20 19.10.02

Professor Dr. Bernd Blümel

Fachhochschule Bochum

Universitätsstraße 150

44801 Bochum

Tel.: 0234-32-10614

email: Bernd.Bluemel@fh-bochum.de

www: http://www.fh-bochum.de/fb6/personen/bluemel/index.html

1 Vereinbarungen

Die in die Unterlagen eingefügten Programme sind in der Schriftart Courier Schriftgrößen 9. Alle Programme sind getestet und als Referenz in diesen Text kopiert. Reservierte Worte und Visual-Basic-Sprachkonstanten im laufenden Text sind ebenfalls mit Courier Schriftgröße 10 formattiert. Variablennamen im Fließtext sind durch Kursiv-Schrift hervorgehoben.

2 Einrleitende Beispiele

In diesem Kapitel stelle ich einige kleinere Programme vor. Dies soll Ihnen ein "Gefühl" für Programmierung geben. Sie sollen die Beispiele nicht vollständig verstehen, alle Programmkonstrukte, die in diesem Kapitel vorkommen, werden in eigenen Kapiteln eingehend behandelt. Darüber hinaus stelle ich einige Gründe dar, warum man sich überhaupt mit Programmierung beschäftigt.

Beispiel 2.1 Das Programm `“Hello World”`

```
Sub HelloWorld
’ Programm gibt Hello World in einem Fenster aus
’ Dateiname: helloWorld
       MsgBox("Hello World!“)
End Sub
```

Betrachten wir dieses kurze Programm. Die erste Zeile `Sub HelloWorld` leitet das Programm ein. `Sub` ist ein in Visual Basic for Applications (von jetzt an VBA abgekürzt) reserviertes Wort. Jedes VBA-Programm startet mit `Sub`[1]. Auf den Befehl `Sub` (reservierte Worte werden häufig auch als Befehle bezeichnet) folgt der Name des Programmes. Jedes VBA-Programm muß einen Namen haben. Der Name kann von uns frei gewählt werden. Er sollte jedoch in einem Zusammenhang mit dem Sinn des Programms stehen. Unser Programm heißt `HelloWorld`. Beachten Sie, daß `HelloWorld` kein Blanc enthält. Blancs sind innerhalb von Namen verboten.

Die nächste Zeile

```
’Programm gibt Hello World in einem Fenster aus
```

ist ein Kommentar. Kommentare werden beim Programmablauf ignoriert. Kommentare sind dazu da, unseren Programmcode (die Textform unserer Programme heißt Programmcode, Quellcode oder Programmquelle) verständlicher zu machen. Die obige Kommentarzeile erklärt den Sinn unseres Programmes.

```
’Dateiname: helloWorld
```

Diese Zeile ist wieder ein Kommentar und eigentlich selbsterklärend.

```
MsgBox("Hello World!“)
```

Dies ist der erste richtige Befehl des Programmes. (Was streng genommen gar nicht richtig ist, da `MsgBox` ein Funktions - oder Methodenaufruf ist. Doch was das ist erfahren Sie erst sehr viel später.) Dieser Befehl führt dazu, daß die VBA-Laufzeitumgebung ein Fenster aufblendet, das den Text Hello World (mit Blanc) enthält.

Der Befehl `End Sub` beendet das Programm.

Abbildung 2.1 zeigt die Bildschirmausgabe unseres ersten Programmes. Allerdings ist Beispiel 2.1 nicht das intelligenteste aller Programme.

1.Eigentlich bedeutet Sub Unterprogramm, kleines Programm.

Abbildung 2.1 Ausgabe von Beispiel 2.1

Wollen wir Hello World am Bildschirm ausgeben, können wir dies zum Beispiel ganz einfach dadurch erreichen, daß wir Hello World in ein Feld der Tabellenkalkulation eingeben, wie Abbildung 2.2 eindrucksvoll zeigt.

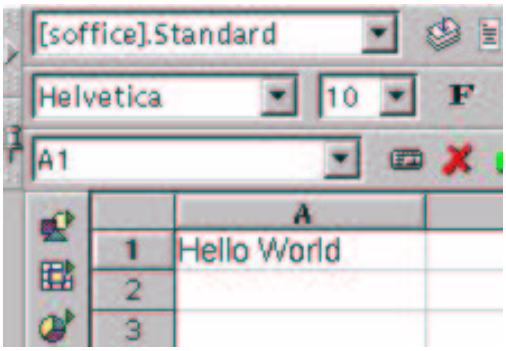

Abbildung 2.2 "Hello World" in einer Zelle der Tabellenkalkulation

Betrachten wir daher ein nächstes Beispiel:

Beispiel 2.2 Ein Additionsprogramm

```
Sub addition()

' Programm addiert die zwei Zahlen 9 und 10
' Dateiname: addition

       dim ersterSummand As Integer
       dim zweiterSummand As Integer
       dim summe As Integer

       ersterSummand = 9
       zweiterSummand = 10
       summe = ersterSummand + zweiterSummand

       MsgBox ("Summe: " & summe)
End Sub
```

Die ersten 3 Zeilen des Programms können wir bereits lesen und richtig interpretieren. Dieses Programm heißt *addition* und die Kommentarzeilen sagen uns, daß dieses Programm die Zahlen 9 und 10 addiert und unter dem Namen addition abgespeichert ist.

Danach jedoch sehen wir neue Dinge. Wir haben 3 relativ gleich aussehende Zeilen:

```
dim ersterSummand As Integer
dim zweiterSummand As Integer
dim summe As Integer
```

Diese Zeilen beginnen mit dem Schlüsselwort `dim`. `dim` leitet eine Variablendeklaration ein. Variablen werden in Kapitel 5 ausführlich erklärt. Grob gesagt sind Variablen die programminternen Namen für die Dinge, mit denen das Programm arbeiten soll. Mit den Variablen kann dann beschrieben werden, was mit den Dingen beim Programmlauf gemacht werden soll.

Hier werden also drei Variablen deklariert mit den Namen *ersterSummand*, *zweiterSummand* und *summe*. Hinter den Namen der Variablen sehen wir die reservierten Worte `As Integer`. Hierdurch wird der Typ der Variablen festgelegt. Durch den Typ wird festgelegt, welche Werte die Variablen annehmen dürfen. `As Integer`. bedeutet, daß auf diesen drei Variablen nur ganze Zahlen[2] abgespeichert werden dürfen.

Danach folgen die Zuweisungen

```
EsterSummand = 9
zweiterSummand = 10
```

Dadurch werden den Variablen *ersterSummand* und *zweiterSummand* Werte zugewiesen. *ersterSummand* erhält den Wert 9, *zweiterSummand* den Wert 10. Durch die Zeile

```
summe = ersterSummand + zweiterSummand
```

wird die Summe der Werte der Variablen *ersterSummand* und *zweiterSummand* der Variablen *summe* zugewiesen. Da *ersterSummand* den Wert 9 hatte und *zweiterSummand* den Wert 10, wird auf der Variablen *summe* nun der Wert 19 abgespeichert.

Die letzten beiden Zeilen kennen wir bereits aus Beispiel 2.1. Durch

```
MsgBox ("Summe: " & summe)
```

wird das Ergebnis in einem kleinen Fenster auf dem Bildschirm ausgegeben (vgl. Abbildung 2.3). Die Anführungszeichen um *"Summe: "* sorgen dafür das das, was zwischen den Anführungszeichen steht "as is " ausgegeben wird. Im Ergebnisfenster erscheint Summe: . Das & sagt VBA, daß sich weitere Ausgaben (zumindest eine) anschließen. Danach folgt *summe* ohne Anführungszeichen. Dadurch interpretiert VBA *summe* als Variablennamen. Variablen werden bei der Ausgabe (und eigentlich immer) durch ihre Werte ersetzt. Der Wert von *summe* ist (wie oben beschrieben) 19. Also erscheint 19 im Ausgabefenster. Die Zeile

```
End Sub
```

beendet das Programm.

2.Was ganze Zahlen sind haben Sie in Mathematik gelernt.

```
Sub addition()

' Programm addiert die zwei Zahlen 9 und 10
' Dateiname: addition

    dim ersterSummand As Integer
    dim zweiterSummar
    dim summe As Inte

    ersterSummand = 9
    zweiterSummand =
    summe = ersterSummand + zweiterSummand

    MsgBox ("Summe: " & summe)
End Sub
```

Abbildung 2.3 Ausgabe von Beispiel 2.2

Auch dieses Programm ist allerdings noch weit davon entfernt, in irgendeiner Weise sinnvoll zu sein, denn zwei Zahlen kann man mit einer Tabellenkalkulation doch etwas einfacher addieren (vgl. Abbildung 2.4).

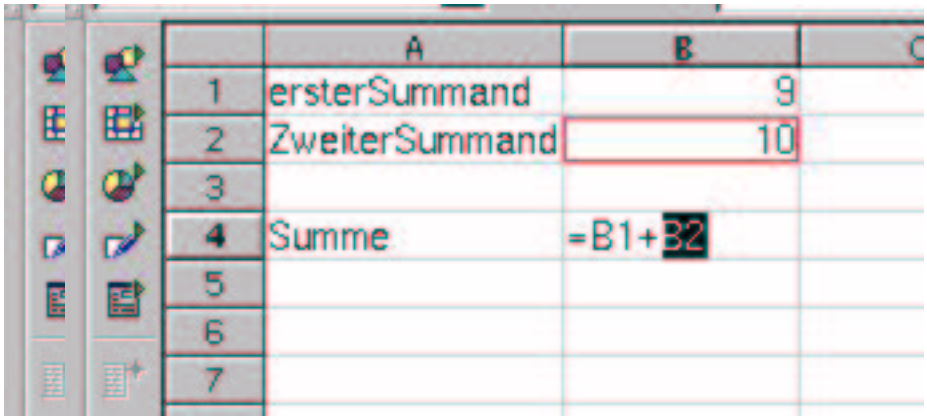

Abbildung 2.4 Addition in einer Tabellenkalkulation

Die Lösung in Abbildung 2.4 ist sogar noch viel besser, als unser Programm. Dort können zwei beliebige Zahlen addiert werden (einfach durch Eingabe in die Felder B1 und B2), unser Programm hingegen kann nur die Zahlen 9 und 10 addieren. Diese neue Funktionalität können wir aber ganz schnell in unser Programm aufnehmen:

Beispiel 2.3 Das Additionsprogramm (verbessert)

```
Sub addition2()
```

```
' Programm addiert die zwei einzugebende Zahlen
' Dateiname: addition

       dim ersterSummand As Integer
       dim zweiterSummand As Integer
       dim summe As Integer

       ersterSummand = InputBox("Bitte geben Sie den ersten Summanden ein")
       zweiterSummand =  InputBox("Bitte geben Sie den zweiten Summanden ein")

       summe = ersterSummand + zweiterSummand

       MsgBox ("Summe: " & summe)
End Sub
```

Dieses Programm unterscheidet sich von Beispiel 2.2 nur durch die zwei neuen Zeilen:

```
       ersterSummand = InputBox("Bitte geben Sie den ersten Summanden ein")
       zweiterSummand =  InputBox("Bitte geben Sie den zweiten Summanden ein")
```

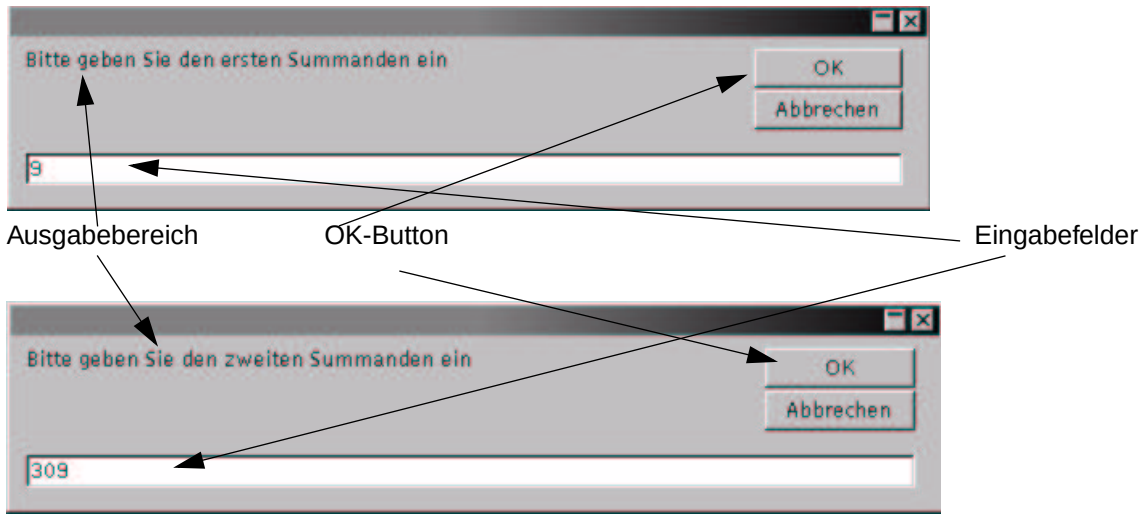

Abbildung 2.5 Die Input-Boxen aus Beispiel 2.3

`InputBox` ist wie `MsgBox` eine von VBA zur Verfügung gestellte Funktion. `InputBox` stellt ein Eingabefenster zur Verfügung. Alles was in den Klammern hinter `InputBox` folgt, wird im Eingabefenster dargestellt (vgl. Abbildung 2.5). Darüberhinaus stellt `InputBox` ein Eingabefeld zur Verfügung. Der Inhalt dieses Eingabefeldes wird, wenn der Benutzer den OK-Button klickt, der Variable *ersterSummand* zugewiesen. Die Zeile

```
       ersterSummand = InputBox("Bitte geben Sie den ersten Summanden ein")
```

ist übrigens eine Zuweisung (wird in Kapitel 6.3 behandelt). Nachdem beide Eingabefenster wie in Abbildung 2.5 ausgefüllt und mit klicken des OK-Buttons abgeschickt worden sind, ist der Wert der Variablen *ersterSummand* 9 und der Wert der Variablen *zweiterSummand* 309. Danach wird, wie in Beispiel 2.2, die Summe berechnet und vermittels eines Ausgabefensters ausgegeben (vgl. Abbildung 2.6).

```
Sub addition2()

' Programm addiert die zwei einzugebende Zahlen
' Dateiname: addition

    dim ersterSummand As Integer
    dim zweiterSummand As Integer
    dim summe As Integer

    ersterSummand = InputBox("Bitte geben Sie den erste
    zweiterSummand =  InputBox("Bitte geben Sie den zwe

    summe = ersterSummand + zweiterSummand

    MsgBox ("Summe: " & summe)
End Sub
```

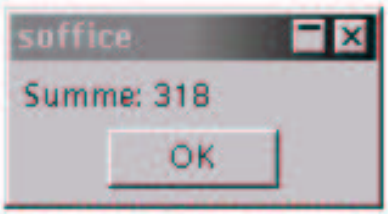

Abbildung 2.6 Ausgabe von Beispiel 2.3

Nun kann unser Beispiel das, was jede Tabellenkalkulation sowieso schon kann. Unser neues Beispiel scheint also wieder nicht so besonders sinnvoll zu sein. Doch mit etwas gutem Willen könnte Beispiel 2.3 tatsächlich sinnvoll sein[3]: Stellen wir uns ein imaginäres Unternehmen vor, in dem einige Mitarbeiter oft Zahlen addieren müssen. Allerdings konnten wir nur Mitarbeiter gewinnen, die nicht in der Lage sind, mit einer Tabellenkalkulation umzugehen. Sie können das Problem, 2 Zahlen mit einer Tabellenkalkulation zu addieren, nicht selber lösen.

Wir stellen also unseren Benutzern mit Beispiel 2.3 eine Problemlösung zur Verfügung.

Nun könnte man argumentieren, daß wir auch dafür kein Programm zu schreiben bräuchten. Wir könnten einfach die Datei aus Abbildung 2.4 abspeichern und unseren Nutzern geben. Aber Benutzer, die nicht in der Lage sind, mit einer Tabellenkalkulation zu addieren, sind im Regelfall sehr wohl in der Lage, den Inhalt einer Zelle unbeabsichtigt zu löschen. Und wird der Inhalt von Zelle B4 in Abbildung 2.4 gelöscht, funktioniert das Addieren nicht mehr.

Doch kommen wir nun zum nächsten Beispiel:

Beispiel 2.4 Ein weiteres Additionsprogramm

```
Sub addition3()
' Programm addiert alle natuerlichen Zahlen, bis zu einer einzugebenden Zahl
' Dateiname: addition

      dim summe As Integer
      dim bisZu As Integer
      dim i As Integer
```

3.wenn wir einfach mal außer Acht lassen, daß es so etwas wie Taschenrechner gibt.

```
        MsgBox ("Addition aller natuerlichen Zahlen von 1 bis zu der Zahl, die Sie eingeben!")
        bisZu = InputBox("Geben Sie nun die Zahl ein, bis zu der addiert werden soll!")

        summe = 0

        for i = 1 to bisZu
                summe = summe + i
        next i

        MsgBox ("Summe: " & summe)
End Sub
```

Das hier dargestellte Programm ist etwas komplexer, als die bisherigen Beispiele. Das in Beispiel 2.4 dargestellte Programm summiert alle natürlichen Zahlen[4] bis zu einer einzugebenden Zahl auf. Gehen wir das Programm im Einzelnen durch:

Die ersten Zeilen sind Ihnen schon vertraut (zumindestens haben Sie ähnliches jetzt bereits einige Male gesehen). Das Programm erhält den Namen *addition3*. In Kommentaren wird erklärt, was das Programm macht und in welcher Datei der Quellcode abgespeichert ist. Dann werden drei Variablen vom Typ `Integer` deklariert. Daraufhin wird dem Benutzer in einem Ausgabefenster (`MsgBox`) erläutert, was das Programm kann, und dann wird mit Hilfe einer `InputBox` die Zahl eingelesen bis zu der addiert werden soll. Diese Zahl wird der Variablen *bisZu* zugewiesen.

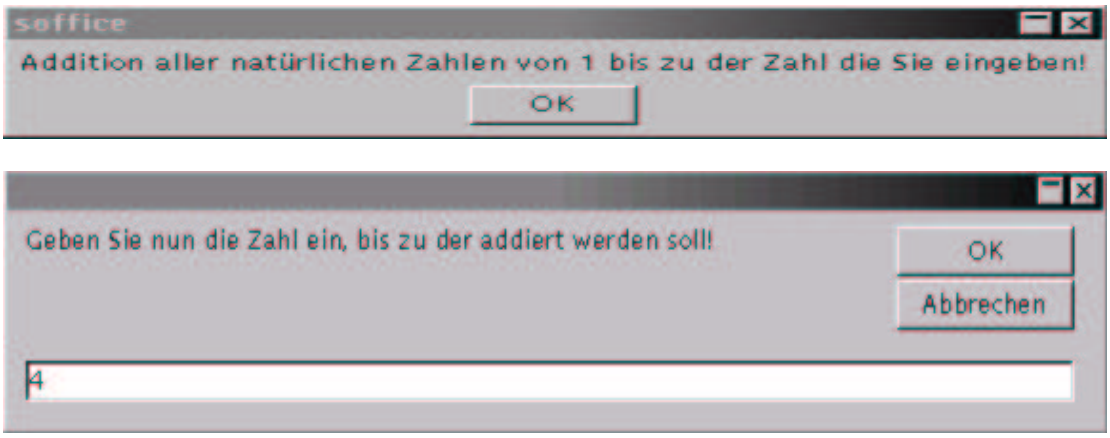

Abbildung 2.7 Erste Message-Box und Input-Box aus Beispiel 2.4

Dann jedoch kommt Neues. Zunächst wir die Variable *summe* mit 0 initialisiert.

```
        summe = 0
```

Dann folgt eine Schleife. Schleifen werden in Kapitel 9 ausführlich erklärt. Schleifen sind Teile unseres Programmcodes, die mehrfach durchlaufen (dies bedeutet durchgeführt) werden können. Die Schleife beginnt mit dem Befehl:

```
        for i = 1 to bisZu
```

4.Was natürliche Zahlen sind haben Sie in Mathematik gelernt.

Trifft VBA auf diese Zeile, wird die Variable *i* mit dem Wert 1 initialisiert. Danach wird überprüft, ob der Wert der Variablen *i* kleiner gleich dem Wert der Variablen *bisZu* ist. Wenn der Benutzer wie in Abbildung 2.7 vier eingegeben hat, ist dies der Fall. Daher werden nun die Anweisungen der Schleife abgearbeitet. Dies sind alle Anweisungen, die sich zwischen

```
for i = 1 to bisZu
```

und

```
next i
```

befinden. In unserem Beispiel ist dies nur die Anweisung:

```
summe = summe + i
```

Da die Variable *summe* mit dem Wert 0 und *i* mit dem Wert 1 initialisiert wurde, ergibt `summe + i` (0 + 1) den Wert 1. Dieser neue Wert wird der Variablen *summe* zugewiesen. *summe* hat also jetzt den Wert 1. Durch die Anweisung

```
next i
```

wird der Wert der Variablen *i* um 1 erhöht. Da *i* vorher den Wert 1 hatte und 1 plus 1 2 ergibt, hat `i` danach den Wert 2. Des Weiteren veranlaßt

```
next i
```

VBA zu der Anweisung

```
for i = 1 to bisZu
```

zurückzugehen. Da diese Anweisung nun zum zweiten Mal durchgeführt wird, wird die Initialisierung von *i* **nicht** mehr durchgeführt. Dies passiert nur bei der ersten Durchführung der `for`-Anweisung. *i* behält also den Wert 2. `for` überprüft nur , ob der Wert von *i* immer noch kleiner oder gleich dem Wert von *bisZu* ist. Da *i* 2 ist und *bisZu* 4, ist dies der Fall. Die Anweisung in der Schleife wird durchgeführt. Die Anweisung der Schleife war:

```
summe = summe + i
```

Da *summe* nach dem letzten Schleifendurchlauf den Wert 1 zugewiesen erhielt und *i* zur Zeit den Wert 2 hat ergibt `summe + i` (1 + 2) nun 3. Dieser neue Wert wird der Variablen *summe* zugewiesen. Durch die Zeile:

```
next i
```

wird *i* um 1 erhöht (`i` ist jetzt 3) und VBA kehrt zur `for`-Anweisung zurück. Hier wird erneut überprüft, ob der Wert von *i* immer noch kleiner oder gleich dem Wert von *bisZu* ist. Dies ist der Fall (*i* ist 3, *bisZu* immer noch 4), also wird wieder die Anweisung in der Schleife durchgeführt. *summe* war nach dem letzten Schleifendurchlauf 3, *i* ist zur Zeit auch 3, also ergibt `summe + i` (3 + 3) 6. Dieser neue Wert wird der Variablen *summe* zugewiesen.

```
next i
```

erhöht `i` wieder um 1, (`i` ist jetzt 4) und VBA kehrt zur `for`-Anweisung zurück. Hier wird erneut überprüft, ob ob der Wert von *i* immer noch kleiner oder gleich dem Wert

von *bisZu* ist. Dies ist der Fall (*i* ist 4, *bisZu* immer noch 4), also wird wieder die Anweisung in der Schleife durchgeführt. *summe* war nach dem letzten Schleifendurchlauf 6, *i* ist zur Zeit 4, also ergibt `summe + i` (6 + 4) 10. Dieser neue Wert wird der Variablen *summe* zugewiesen.

```
Next i
```

erhöht *i* wieder um 1, (`i` ist jetzt 5) und VBA kehrt zur `for`-Anweisung zurück. Hier wird erneut überprüft, ob ob der Wert von *i* immer noch kleiner oder gleich dem Wert von *bisZu* ist. Dies ist diesmal nicht der Fall (*i* ist 5, *bisZu* immer noch 4). Dies veranlaßt VBA nun, die Schleife zu beenden. Eine Schleife zu beenden bedeutet, mit der ersten Anweisung hinter der Schleife fortzufahren. Dies ist:

```
MsgBox ("Summe: " & summe)
```

Unser berechnetes Ergebnis wird jetzt ausgegeben. *summe* hatte zum Schluß den Wert 10, also erscheint dieser auch im Ausgabefenster (vgl. Abbildung 2.8).

```
Sub addition3()
' Programm addiert alle natuerlichen Zahlen, bis zu einer ein
' Dateiname: addition

    dim summe As Integer
    dim bisZu As Integer
    dim i As Integer

    MsgBox ("Addition aller natuerlichen Zahlen von 1 bis zu
    bisZu = InputBox("Geben Sie nun die Zahl ein, bis zu der

    summe = 0

    for i = 1 to bisZu
        summe = summe + i
    next i

    MsgBox ("Summe: " & summe)
End Sub
```

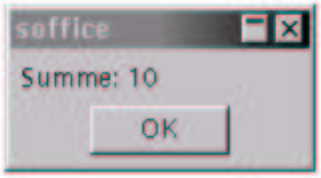

Abbildung 2.8 Ausgabe von Beispiel 2.4

In diesem Beispiel haben wir einige neue Dinge kennengelernt:

- Die Werte von Variablen können sich ändern. Tabelle 2.1 zeigt noch einmal die Änderungen der Werte der Variablen *summe* und *i* während der Schleifendurchläufe.
- Programmteile können mehrfach durchlaufen werden.
- Variablen können auf der rechten und auf der linken Seite einer Zuweisung stehen.

Tabelle 2.1 Werte der Variablen i und summe während des Programmlaufs (Wertetabelle)

Schleifendurchlauf	Wert der Variablen `i`	Wert der Variablen `summe`
1	1	1
2	2	3
3	3	6
4	4	10

- Es wird zuerst die rechte Seite einer Zuweisung ausgewertet. Das Ergebnis dieser Auswertung wird der Variablen auf der linken Seite zugewiesen.
- Variablen, die zuerst auf der rechten Seite einer Zuweisung vorkommen müssen initialisiert werden. Gäbe es nicht die Zeile:

```
summe = 0
```

  könnte VBA nicht wissen, welchen Wert *summe* beim ersten Durchlaufen der Anweisung

```
summe = summe + i
```

  besitzt.
- Schleifenvariablen (`i` ist die Schleifenvariable) müssen initialisiert werden.

Beispiel 2.4 kann nicht so ohne weiteres direkt in einer Tabellenkalkulation realisiert werden. Zwar ist es durchaus möglich, mit einer Tabellenkalkulation die Summe der ersten n natürlichen Zahlen zu berechnen, wie auch Abbildung 2.9 für n = 4 zeigt.

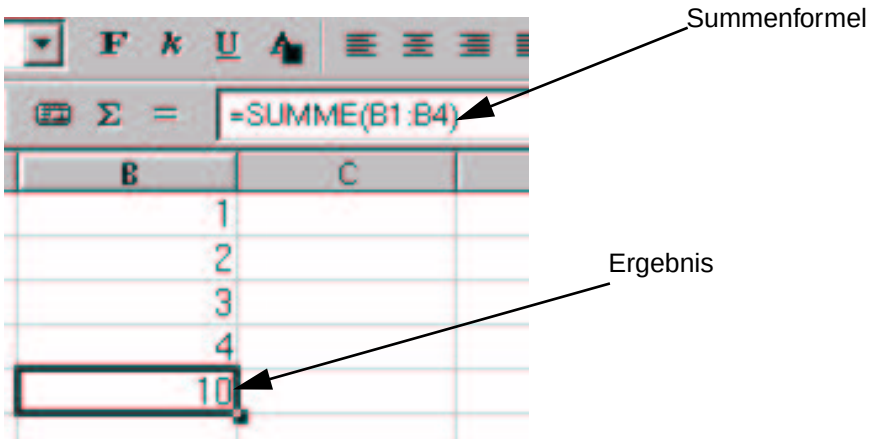

Abbildung 2.9 Aufsummierung der Zahlen 1 bis 4 in einer Tabellenkalkulation

Ändert sich aber die Zahl, bis zu der summiert werden soll, muß die Lösung geändert werden. Wenn beispielsweise die ersten 10 Zahlen aufsummiert werden sollen, müssen 6 weitere Zeilen eingefügt und mit den korrekten Werten belegt werden, ehe die Tabellenkalkulation dann das Ergebnis berechnen kann. Gehört es also zu meinen Tä-

tigkeiten[5] öfter mal Zahlen aufzusummieren, ist es einfacher und viel schneller unser Programm zu benutzen, als jedesmal die Tabellenkalkulation anzupassen. Wir haben durch unsere Programmierung diese Tätigkeit automatisiert. Dies ist natürlich ein praxisfernes Beispiel, aber betrachten wir nur mal die folgenden 2 Beispiele:

- **Klausurauswertung:** Ich muß in regelmäßigen Abständen Klausuren stellen, korrigieren und bewerten. Dies geschieht dadurch, daß ich für die einzelnen gelösten oder auch nicht gelösten Aufgaben Punkte vergebe, die Punkte für alle Aufgaben addiere und dann abhängig von der Punktzahl eine Note vergebe. Dies geschieht nach festgelegten Regeln. So ist z.B. eine Minimalpunktzahl von 50 % der erreichbaren Punkte für eine 4 erforderlich. Zum Schluß erzeuge ich eine Ausgabe, wie viele Einsen, Zweien usw. es gegeben hat und wieviel Prozent aller Teilnehmer das waren. Ich berechne die Durchschnittsnote und erzeuge Grafiken mit all diesen Informationen. Das kann man sicher alles von Hand machen. Ich habe aber ein Programm, wo ich nur die Punktzahl der Teilnehmer sowie die maximal erreichbare Punktzahl eingeben muß. Dann klicke ich auf "Bewerten" und mein Programm macht mir alles fertig.

- **Auswertung von Kundendaten:** Stellen Sie sich vor, Sie müssen Kundendaten nach bestimmten Kriterien auswerten, wie z.B. Gesamtumsätze pro Region und/oder pro Vertriebsbeauftragtem, neue vom Umsatz abhängige Rabattstufen für Kunden ermitteln, Provisionen für Vertriebsbeauftragte errechnen und einige dieser Dinge dann grafisch darstellen. Kundendaten werden in Unternehmen in den meisten Fällen in einer zentralen Datenbank vorgehalten. Auf diese Datenbank kann man mittels einer standardisierten Schnittstelle (ODBC= Open DataBase Connectivity) von Excel oder Access und auch von VBA aus zugreifen und sämtliche Auswertungen in VBA programmieren.

Übrigens, wer nicht so gut in Programmierung, aber dafür gut in Mathematik ist[6], hätte Beispiel 2.4 sehr viel einfacher lösen können. Es gilt nämlich:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Oder umgangssprachlich ausgedrückt: Die Summe der ersten n natürlichen Zahlen ist n mal (n+1) und das Ergebnis davon durch 2. Wir hätten Beispiel 2.4 also auch folgendermaßen codieren können:

Beispiel 2.5 Das Summierungsprogramm ohne Schleife

```
Sub addition4
' Programm addiert alle natürlichen Zahlen, bis zu einer einzugebenden Zahl
' Dateiname: addition4

      dim summe As Integer
      dim bisZu As Integer
      dim i As Integer
```

5.Das ist glücklicherweise nicht der Fall.

6.was zugegebenermaßen nicht so besonders häufig vorkommt

```
        MsgBox ("Addition aller natürlichen Zahlen von 1 bis zu der Zahl die Sie eingeben!")
        bisZu = InputBox("Geben Sie nun die Zahl ein, bis zu der addiert werden soll!")

        summe = (bisZu * (bisZu + 1)) / 2

        MsgBox ("Summe: " & summe)
End Sub
```

Das ist natürlich sehr viel einfacher und läuft auch schneller. Die Zeile:

```
        summe = (bisZu * (bisZu + 1)) / 2
```

zeigt uns, daß VBA nicht nur addieren, sondern auch multiplizieren und dividieren kann. Darüber hinaus kann man Klammern setzen.

Diese Aussage kann man übrigens auch mathematisch beweisen. Das hat zwar jetzt rein gar nichts mit Informatik zu tun, ist aber trotzdem ganz schön. In unserem Fall kann man vollständige Induktion als Beweismethode benutzen. Und das funktioniert so:

Wir zeigen zunächst, daß die Formel für eine bestimmte Zahl gilt, z.B. für 2. Das ist einfach:

$$\sum_{i=1}^{2} i = (1+2) = 3$$

$$\frac{2(2+1)}{2} = \frac{2 \times 3}{2} = \frac{6}{2} = 3$$

Das nennt man die Induktionsverankerung.

Als nächstes zeigen wir: Wenn die Formel für die Zahl n gilt, dann gilt sie auch für n + 1. Damit haben wir die Formel dann bewiesen. Denn: Daß die Formel für 2 gilt, haben wir ja bewiesen. Da sie für 2 gilt, gilt sie dann auch für 3. (n=2 -> n+1 = 3). Wenn sie aber für 3 gilt, dann auch für 4 usw.

Wir müssen also nur noch zeigen: Vorausgesetzt unsere Formel gilt für n dann auch für n + 1. Das ist etwas schwieriger:

$$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + n + 1$$

Da ja:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

ist gilt

$$\sum_{i=1}^{n} i + n + 1 = \frac{n(n+1)}{2} + n + 1 = \frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{n^2 + n + 2n + 2}{2} = \frac{n^2 + 3n + 2}{2}$$

andererseits:

$$\frac{(n+2)\times(n+1)}{2} = \frac{n^2 + 2n + n + 2}{2} = \frac{n^2 + 3n + 2}{2}$$

also gilt:

$$\sum_{i=1}^{n+1} i = \frac{(n+2)\times(n+1)}{2}$$

und damit ist diese Gleichung bewiesen.

Noch einige Bemerkungen:

- In allen Beispielen ist der Code eingerückt. Alle Codezeilen, die zu einem Programm gehören, sind um eine Tabulatorposition eingerückt. Die Programmzeilen in Beispiel 2.4, die zu der Schleife gehören, sind eine weitere Tabulatorposition eingerückt. Dies macht Programme leichter lesbar. Fehler werden schneller gefunden. Sie sollten sich diesen Programmierstil auch angewöhnen.
- Variablen haben sprechende Namen. Soll heißen: Der Name einer Variablen läßt Rückschlüsse auf ihren Inhalt zu. Unsere Variablen heißen *ersterSummand*, *zweiterSummand* und *summe* und nicht etwa *a*, *b*, *c*, *d*, was ja kürzer wäre. Programme mit "guten" Variablennamen sind aber weitaus leichter lesbar und viel verständlicher. Einzige Ausnahme ist die Schleifenvariable *i* in Beispiel 2.4. Schleifenvariablen nennen wir immer *i*, *j* oder *k*. Das macht jeder so und daher weiß man, wenn eine Variable diesen Namens in einem Programm vorkommt, ist es eine Schleifenvariable und die Verständlichkeit bleibt gewahrt.

3 Algorithmus

Bevor Sie mit der eigentlichen Codierung beginnen, müssen Sie wissen, **was** Sie programmieren wollen. Und nicht nur das, Sie müssen auch wissen, **wie** Sie es programmieren wollen. Beispiel 2.4 ist dafür ein gutes Beispiel. Dort ist die Aufgabenstellung, die ersten n natürlichen Zahlen aufzusummieren. Unsere Lösung sieht in einfachen Worten dargestellt, folgendermaßen aus:

- lies die Zahl n, bis zu der summiert werden soll, ein (Abspeichern auf Variable bisZu)
- initialisiere die Variable *summe* mit 0
- initialisiere die Variable *i* mit 1
- so lange die Variable *i* nicht größer als n ist
    - o addiere den Wert von *i* zu *summe* und weise das Ergebnis *summe* wieder zu
    - o erhöhe *i* um 1
- gib die errechnete *summe* aus.

Was wir jetzt formuliert haben, ist eine Beschreibung der Problemlösung. So etwas nennen wir ab jetzt einen Algorithmus.

Algorithmen können unterschiedlich komplex sein. In Beispiel 2.1 ist das Ziel, Hello World in einem Bildschirmfenster auszugeben. Der Algorithmus hierzu ist:

Gib Hello World in einem Bildschirmfenster aus!

Dies ist ein sehr einfacher Algorithmus. Er entspricht der Aufgabenstellung.

Konventionen:

Das Ziel (oder die Aufgabenstellung) heißt von nun an Problem.

Die Beschreibung des Ziels heißt demzufolge Problembeschreibung.

Ein Algorithmus ist eine Vorschrift für die Lösung eines Problems. Oder ausführlicher formuliert: Ein Algorithmus ist eine detaillierte Vorschrift für die schrittweise Lösung eines Problems.

Beispiel 3.1 Anwendung der obigen Konventionen auf Beispiel 2.3

Problembeschreibung: 2 Zahlen sollen addiert werden.

Algorithmus: Lies beide Zahlen ein

Bilde die Summe

Gib die Summe am Bildschirm aus

Die Programmierung in VBA haben wir bereits gesehen.

Dieses Beispiel zeigt:

1) Der Algorithmus zeigt uns die Problemlösung in kleinen Schritten an.

2) Der Algorithmus kommt **vor** der Programmierung. Programmieren kann ich erst, wenn ich einen Algorithmus gefunden habe!!

3) Der Algorithmus und damit auch die Findung des Algorithmus ist unabhängig von der eingesetzten Programmiersprache.

Daher ergibt sich folgende Reihenfolge bei der Programmierung:

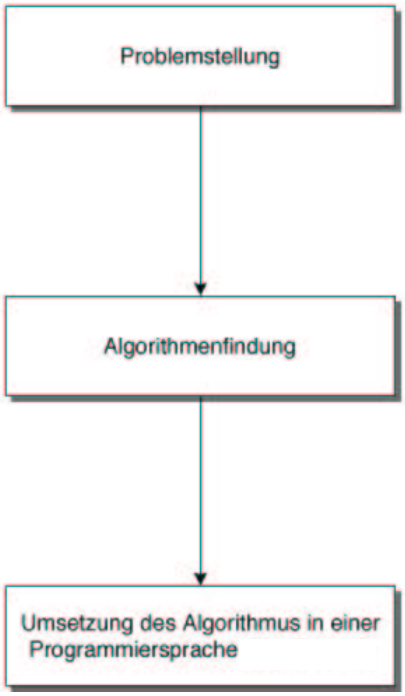

Abbildung 3.1 Programmierreihenfolge

Nachdem wir den Algorithmus entwickelt haben, muß er dokumentiert werden. Dazu gibt es viele Möglichkeiten. Die am weitesteten verbreiteten sind:

Pseudocode.

Nassi-Shneidermann-Diagramm (NSD).

Pseudocode oder strukturierte Sprache haben wir im vergangenen Semester im Rahmen der strukturierten Analyse bereits behandelt.

Elemente der Strukturierten Sprache (des Pseudocodes) sind:

- einfache Sätze.
- Verben in Imperativform.
- reservierte Wörter (if, while, else, etc).

Durch Pseudocode wird der Algorithmus dann durch eine Abfolge einfacher Sätze, die durch reservierte Worte strukturiert werden, abgebildet. Dies klingt furchtbar kompliziert, ist es aber eigentlich nicht. Wir werden dies im weiteren Text an Beispielen veranschaulichen.

Pseudocode 3.1 Pseudocode zu Beispiel 2.3:

```
Lies beide Zahlen ein
Bilde die Summe
Gib die Summe am Bildschirm aus
```

Nassi-Shneidermann-Diagramme (manchmal auch als Struktogramme bezeichnet) beschreiben den Algorithmus, wie der Pseudocode auch, mit einfachen Worten, unterstützen die Darstellung aber auch durch rudimentäre Grafiken.

Aufeinanderfolgende Anweisungen, wie der Pseudoode in Pseudocode 3.1, werden durch Rechtecke dargestellt (vgl. Abbildung 3.2 .)

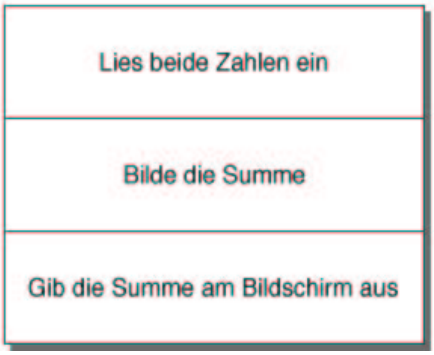

Abbildung 3.2 Nassi-Shneidermann-Diagramm zu Beispiel 2

Für jede Problemstellung muß vor der Realisierung in VBA der von Ihnen gefundene Algorithmus in Pseudocode oder in einem Nassi-Shneidermann-Diagramm dargestellt werden.

Durch Pseudocode oder NSD beschriebene Algorithmen lassen sich dann "leicht" in eine Programmiersprache überführen.

Dazu benötigt man Kenntnisse über Programmiersprachen zugrundeliegende Konzepte. Diese Konzepte werden Sie nun am Beispiel der Sprache VBA erlernen.

Merke: Die Hauptaufgabe ist immer, den Algorithmus zu finden und zu dokumentieren!!!!

4 Programmstart und Programmende

Jedes VBA-Programm beginnt mit dem reservierten Wort *sub*. Darauf folgt der Name des Programmes gefolgt von runden Klammern. Der Name ist vom Programmierer frei wählbar (mit gewissen Ausnahmen, er muß mit einem Buchstaben beginnen und darf außer dem Unterstrich (_) keine Sonderzeichen (Ausnahme später) enthalten. Er darf kein von VBA reserviertes Wort sein. Er sollte allerdings in einem Zusammenhang mit dem Sinn des Programms stehen.

Beispiel:

```
sub addition()
```

Deutsche Sonderzeichen sind erlaubt. Es ist jedoch keine besonders gute Idee, deutsche Sondereichen in einem Programmnamen zu verwenden. VBA ist mit die einzige Programmiersprache, die die Verwendung landesspezifischer Sonderzeichen zuläßt. Hat man sich einmal angewöhnt, deutsche Sonderzeichen zu verwenden und muß dann irgendeine andere Programmiersprache benutzen, sind Fehler vorprogrammiert. Außerdem erschwert die Nutzung deutscher Sonderzeichen die Portierung der Programme in eine andere Sprache.

Jedes VBA-Programm endet mit dem reservierten Wort `End Sub`.

5 Variablen und Datentypen

Alle Programme manipulieren Daten, um gewünschte Ergebnisse zu erzielen.

Die Daten werden oftmals erst während des Programmlaufs eingelesen (z. B. die beiden Zahlen, die addiert werden sollen, in Beispiel 2.3).

Das Programm muß einen Namen haben, für die Daten, mit denen es später arbeiten soll, damit die Dinge (später nennen wir Dinge Operationen), die mit den Daten gemacht werden sollen, beschrieben werden können.

Beispiel:

```
summe = ersterSummand + zweiterSummand
```

Hierbei ist

ersterSummand:	Der Name für die erste zu addierende Zahl (das erste einzulesende Datum).
zweiterSummand:	der Name für die zweite zu addierende Zahl (das zweite einzulesende Datum)
summe::	der Name für das Resultat (das auszugebende Datum).

Die Anweisung

```
summe = ersterSummand + zweiterSummand
```

ist nur möglich, weil die Daten, mit denen das Programm arbeiten soll, Namen bekommen haben.

Variablen sind die programminternen Namen für die Daten, mit denen das Programm arbeiten soll. Mit den Variablen kann dann beschrieben werden, was mit den Daten beim Programmlauf getan werden soll.

Beim Programmstart (Programm wird in den Hauptspeicher kopiert) muß das Programm im RAM Platz reservieren für die Daten, die später eingelesen werden. Daher muß es wissen, mit wieviel Daten es zu tun bekommt und von welcher Art die Daten sind (ein Buchstabe nimmt im RAM weniger Platz weg, als eine reelle Zahl (vgl. Abbildung 5.1 und Abbildung 5.2).

Das Programm `addition` muß dem Betriebssystem also mitteilen, wieviel Platz für die Daten, mit denen es arbeiten will, (Variablen im Programm) im RAM reserviert werden muß.

Variablen sind also Platzhalter mit Namen, denen ein bestimmter Datentyp zugeordnet werden kann. Sie können dann nur Daten dieses Typs aufnehmen.

VBA kennt darüberhinaus einen allgemeinen Datentyp. Variablen diesen Typs können beliebige Werte annehmen. VBA erkennt bei der Zuweisung (was das ist -> später) den Typ der Variablen. Dies wird durch erhöhten Speicherverbrauch erkauft, denn da auf einer solchen Variablen alle Datentypen gespeichert werden können, muß VBA

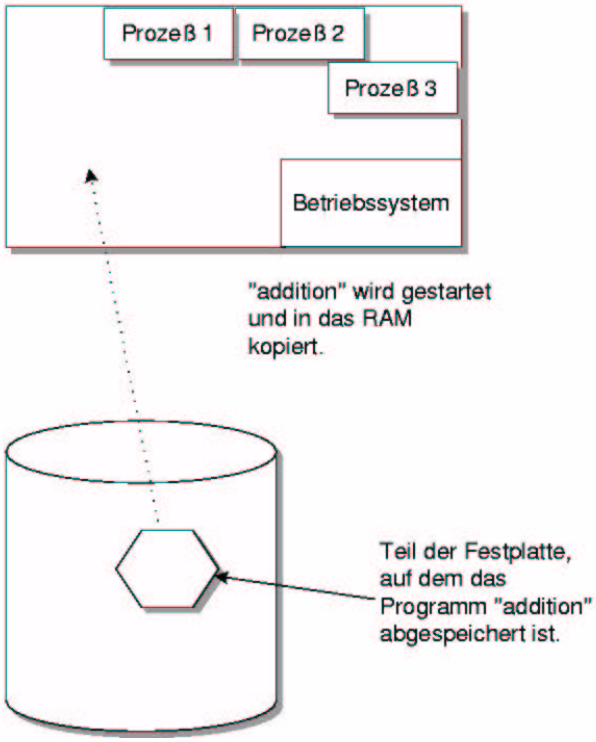

Abbildung 5.1 Start des Programmes addition

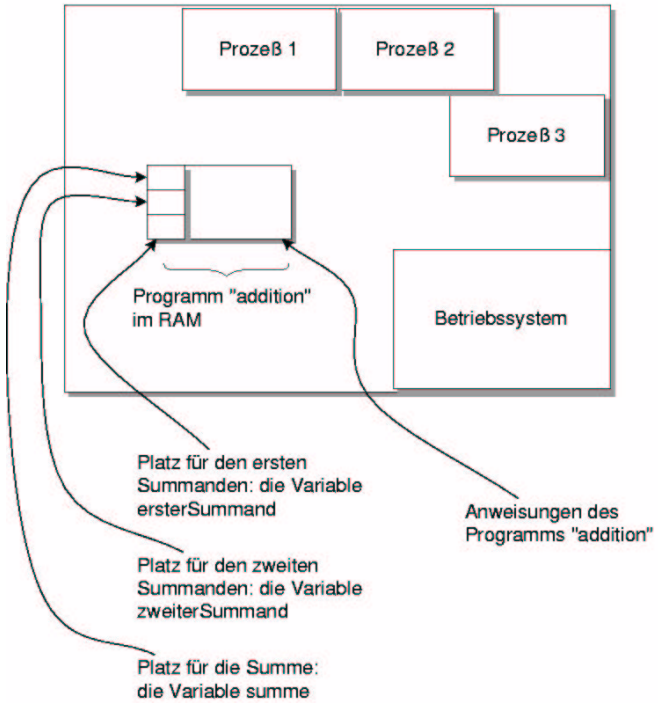

Abbildung 5.2 Das Programm addition im RAM

um auf der sicheren Seite zu sein, maximalen Speicherplatz für Variablen diesen Typs reservieren.

5.1 Datentypen in VBA

VBA unterscheidet im wesentlichen zwischen:

- Ganzzahlvariablen: Auf Variablen diesen Typs können ganze Zahlen z. B. -1, 1000, 3, 7, 12, -1000) gespeichert werden.
- reellen oder Fließkommazahlen: z.B. -1.2, -100.001, 3.1, 7.3, 1000,02)

  Merke: . (Punkt) statt , (Komma) in VBA als Dezimaltrenner
- Wahrheitswerten: (true oder false)
- Strings: Zeichenketten z.B. "Bernd Blümel", "Christiane Schäfer"

Tabelle 5.1 gibt eine Übersicht über die in VBA möglichen Datentypen.

Tabelle 5.1 Datentypen in VBA

Name	Kürzel	Beschreibung	Größe
Integer	%	für ganze Zahlen zwischen -32768 und 32767	2 Byte
Long	&	für ganze Zahlen zwischen -2147483648 und 2147483647	4 Byte
Byte	-	für ganze Zahlen zwischen 0 und 255	1 Byte
Single	!	für Fließkommazahlen mit 8 Stellen Genauigkeit	4 Byte
Double	#	für Fließkommazahlen mit 16 Stellen Genauigkeit	8 Byte
Boolean	-	für Wahrheitswerte	4 Byte
String	$	für Zeichenketten	10 Byte plus 2 Byte pro Zeichen
Date	-	für Datum und Uhrzeit	8 Byte
Currency	@	Festkommazahlen mit 15 Stellen vor und 4 Stellen nach dem Komma	8 Byte
Object	-	Speichert Verweis auf ein Objekt, wird viel später, wenn überhaupt behandelt	4 Byte
Variant		Dies ist der Defaultdatentyp. Nimmt je nach Bedarf einen der obigen Datentypen an, der Speicherbedarf beträgt mind. 16 Byte , bei Strings sogar 22 Byte plus 2 Byte für jedes Zeichen	mind. 16 Byte

Wie wir sehen, gibt es zwei Datentypen für Fließkommazahlen und drei Datentypen für ganze Zahlen.

Bei Fließkommazahlen liegt der Grund in der Möglichkeit von Rundungsfehlern. Bei der geringen Genauigkeit von Single-Variablen kann es bei hintereinandergeschalteten Rechenoperationen mit Daten signifikant unterschiedlicher Größe zu falschen Ergebnissen kommen. Die höhere Genauigkeit der Double-Variablen erkauft man sich durch höheren Hauptspeicherverbrauch.

Die Grund für die Existenz von drei Datentypen für ganze Zahlen liegt auf der Hand. Wenn ich weiß, wie groß die Werte der Variablen im Programmlauf werden können, kann ich durch Wahl des geeigneten Variablentyps Hauptspeicher sparen.

String-Variablen benötigt man, um (trivialerweise) mit Strings zu arbeiten. Was Variablen vom Typ Boolean sind und wozu man sowas braucht, wird später erklärt (fängt an in Kapitel 6.2.2 auf Seite 35).

Variablen (also die Namen/Platzhalter, mit denen das Programm später arbeiten soll) müssen im Programm vereinbart (ein weiteres Wort "deklariert") werden.

Die Anweisung, um Variablen zu deklarieren, ist `dim`. Um der Variablen einen Typ zuzuweisen, gibt es dann 2 Möglichkeiten:

- Der Typ der Variablen wird durch das Schlüsselwort `As` gefolgt vom Typ der Variablen, wie in der ersten Spalte von Tabelle 5.1 definiert, festgelegt. Schauen wir uns, um dies zu illustrieren, noch einmal auf den Code von Beispiel 2.4. an. Durch die Zeilen:

  ```
  dim ersterSummand As Integer
  dim zweiterSummand As Integer
  dim summe As Integer
  ```

  werden also 3 Variablen vom Typ Integer vereinbart. Auf jeder dieser Variablen kann eine ganze Zahl zwischen -32768 und 32767 gespeichert werden.

- Alternativ können Variablen von Datentypen, die über ein Kürzel verfügen (zweite Spalte von Tabelle 5.1), durch den Variablennamen direkt (und ohne Blanc) gefolgt vom Kürzel deklariert werden. Die Variablendeklaration von Beispiel 2.4 hätten wir also auch folgendermaßen schreiben können:

  ```
  dim ersterSummand%
  dim zweiterSummand%
  dim summe%
  ```

Beispiel 5.1 Variablendeklarationen

```
Sub variablen()
' Beispiele fuer Variablendeklarationen
' Dateiname variablen
       dim reelleZahl As Single
       dim reelleZahl2#
       dim reelleZahl3 As Double
       reelleZahl3 = 7
       reelleZahl2 = 4.7
       reelleZahl = reelleZahl2 + reelleZahl3
       MsgBox("reelleZahl: " & reelleZahl)
       dim ganzeZahl As Integer
```

```
        dim ganzeZahl2%
        dim ganzeZahl3 As Long
        ganzeZahl3=5
        ganzeZahl2=8
        ganzeZahl = ganzeZahl2 + ganzeZahl3
        MsgBox("ganzeZahl: " & ganzeZahl)
        dim chamaeleon
        chamaeleon = 6
        MsgBox("chamaeleon zum Ersten: " & chamaeleon)
        chamaeleon = "sieben"
        MsgBox("chamaeleon zum Zweiten: " & chamaeleon)
        ganzeZahl = reelleZahl2
        MsgBox("ganzeZahl zum Zweiten: " & ganzeZahl)
        ganzeZahl = "sieben"
        MsgBox("ganzeZahl zum Dritten: " & ganzeZahl)
        test=9
        MsgBox("test: "& test)
End Sub
```

In den ersten 3 Zeilen von Beispiel 5.1 werden Fließkommavariablen vereinbart. Zwei davon sind vom Typ Double (einmal durch die lange Schreibweise mit `As Double` und einmal durch das Anfügen des Typkürzels #). Dann werden sie addiert und die Summe wird ausgegeben. Alle Ausgaben zeigt Abbildung 5.3. Man sieht hier auch, daß die beiden unterschiedlichen Formate automatisch ineinander konvertiert werden (natürlich nur soweit es sinnvoll ist; bei der Konvertierung einer Single- in eine Double-Variable sind die letzten Stellen der Genauigkeit zufällig, im umgekehrten Fall werden sie abgeschnitten.).

Dann sehen wir dasselbe mit Ganzzahlvariablen. Zwei Integer-Variablen werden deklariert (einmal durch die lange Schreibweise mit `As Integer` und einmal durch das Anfügen des Typkürzels %) und eine Variable vom Typ `Long`. Danach werden sie addiert und die Summe wird ausgegeben. Man sieht hier auch, daß die beiden unterschiedlichen Formate automatisch ineinander konvertiert werden (natürlich nur soweit die Größe der `Long`-Variable so etwas zuläßt).

Als nächstes definieren wir eine Variable mit Namen *chamaeleon* ohne jeden Typ. So eine Variable ist automisch vom Typ `Variant` und kann jeden beliebigen Inhalt aufnehmen. Zuerst weisen wir der Variablen den Wert 0 zu. *chamaeleon* ist dann eine Ganzzahl-Variable. Durch die Zuweisung einer Zeichenkette (Zeichenketten werden in VBA in Anführungszeichen eingeschlossen) ändern wir den Typ von Ganzzahl nach String, der String wird ausgegeben.

Dann weisen wir einer ganzen Zahl eine reelle Zahl zu.

```
        ganzeZahl = reelleZahl2
```

Dies funktioniert ohne Fehlermeldung. Der Wert von *reelleZahl2* war 4.7. Bei der von VBA jetzt automatisch durchgeführten Konvertierung von Fließkommazahl nach Ganzzahl wird gerundet. *ganzeZahl* hat also jetzt den Wert 5, wie auch Abbildung 5.3 zeigt.

Danach wird einer Ganzzahlvariablen ein String zugewiesen:

```
        ganzeZahl = "sieben"
```

Abbildung 5.3 Die Bildschirmausgaben von Beispiel 5.1

Auch dies funktioniert (ärgerlicherweise) ohne Fehlermeldung. Denn es gibt keine Möglichkeit, einen String, der nicht aus Ziffern besteht, in eine Zahl umzuwandeln. Die Ausgabe zeigt dies auch deutlich: *ganzeZahl* hat nun den Wert 0 und der hat ganz sicher nichts mit dem String "sieben" zu tun.

Zum Schluß weisen wir einer Variablen einen Wert zu, ohne sie vorher deklariert zu haben.

```
test=9
```

`dim test` kommt in unserem Programm nicht vor. Trotzdem funktioniert auch dies, VBA erzeugt einfach automatisch eine Variable vom Typ Variant.

Nun stellt sich die Frage: Warum überhaupt Variablen deklarieren und so viel tippen, wenn VBA eigentlich automatisch (vielleicht) das richtige macht, wenn wir Variablen einfach so in Betrieb nehmen, ohne uns um Deklarationen zu kümmern?

Betrachen wir dazu zwei Beispiele:

Beispiel 5.2 Merkwürdige Addition

```
Sub addition5()
' Dies Programm addiert
' zwei Zahlen fehlerhaft
' Dateiname addition5

       ersteEingabe=InputBox("GebenSie die erste Zahl ein!")
       zweiteEingabe=InputBox("GebenSie die zweite Zahl ein!")

       ergebnis = ersteEingabe + zweiteEingabe
```

```
        MsgBox ("Die Summe ist: " & ergebnis)
end Sub
```

Dieses Programm unterscheidet sich nicht groß von Beispiel 2.3. Es fehlen eigentlich nur die Variablendeklarationen. Wir erwarten, daß die zwei eingegebenen Zahlen addiert und die Summe ausgegeben wird. Schauen wir uns nun an, was wirklich passiert. Abbildung 5.4 zeigt Eingaben mit zugehöriger Ausgabe.

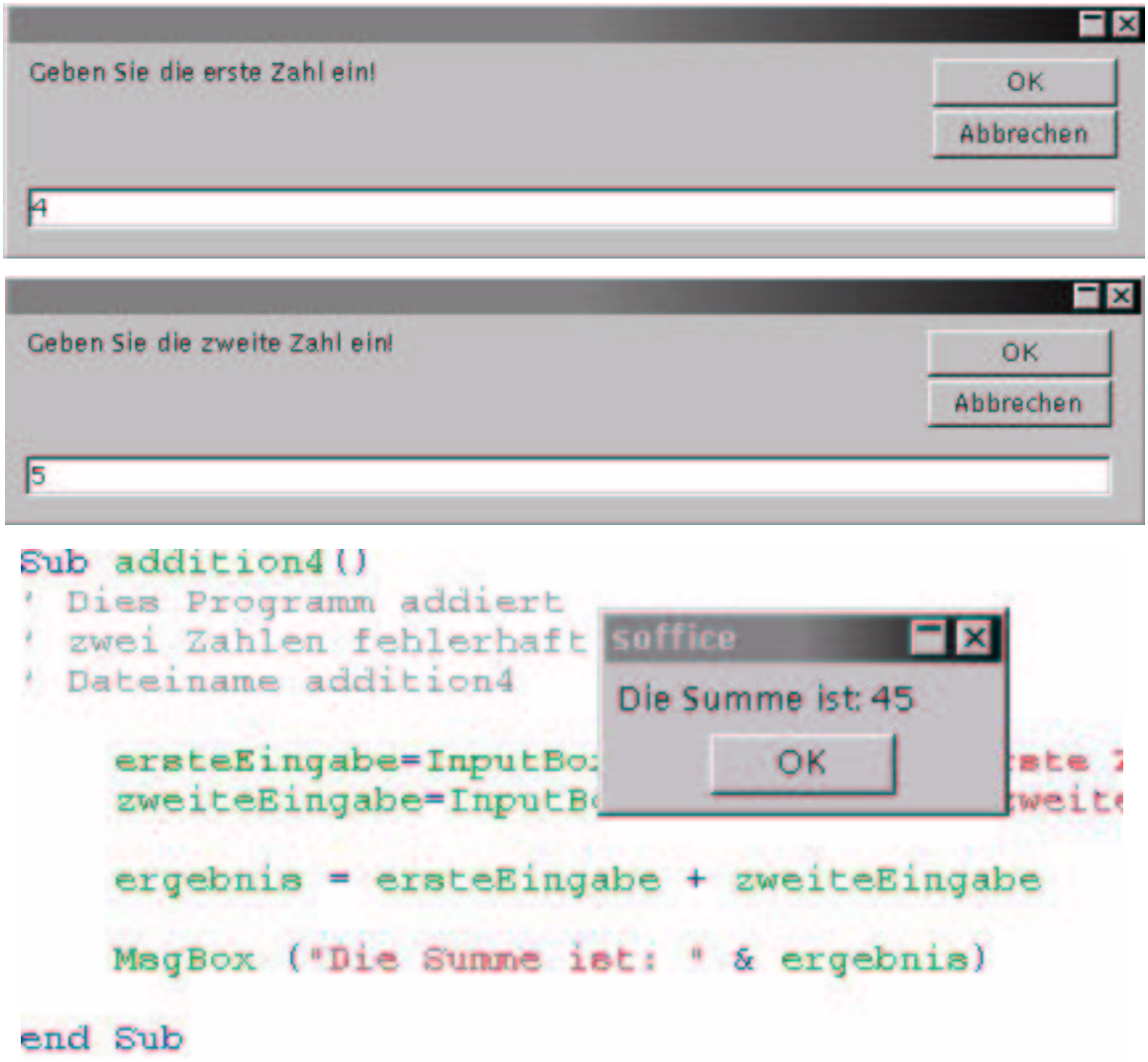

Abbildung 5.4 Die Bildschirmeingaben und die Ausgabe von Beispiel 5.2

Eingegeben wurden 4 und 5. Rein intuitiv würden wir als Summe 9 erwarten. Ist nicht. Die Summe ist 45. Wie kommt das nun zustande? Dadurch, daß wir die Variablendeklarationen weggelassen haben, weiß VBA nicht, welchen Typ die Eingaben haben. Ich habe nun 4 und 5 eingegeben. Das können Zahlen sein, oder auch die Strings "4" und "5". InputBox interpretiert alle Eingaben grundsätzlich als Strings. Da VBA nichts über die Typen der Variablen *ersteEingabe* und *zweiteEingabe* weiß, wird auch nichts konvertiert. Hinzu kommt , daß das + Zeichen auch für Strings eine besondere Bedeu-

tung hat (vgl Kapitel 6.2.4). Strings werden nämlich einfach zu einem großen String zusammengefaßt.

Ist nämlich:

```
string1="ab"
string2="cd"
```

dann hat

```
stringGesamt = string1 + string2
```

den Wert "abcd".

Damit wird aber auch unser Beispiel klar: *ersteEingabe* hatte den Wert "4", *zweiteEingabe* den Wert "5", also hat

```
summe = ersteEingabe + zweiteEingabe
```

den Wert "45". Dies ist aber ganz sicher nicht das, was wir wollten. Aber selbst, wenn wir Variablen deklarieren, können wir unerwünschte Ergebnisse erhalten. Betrachten wir folgendes Programm:

Beispiel 5.3 Schreibfehler führt zu falschem Ergebnis

```
Sub additionUndMultiplikation()
' Dies Programm addiert und multipliziert
' zwei Zahlen fehlerhaft
' Dateiname additionUndMultiplikation

       Dim ersteEingabe As  Integer
       Dim zweiteEingabe As Integer
       Dim ergebnis As Integer

       ersteEingabe=InputBox("Geben Sie die erste Zahl ein!")
       zweiteEingabe=InputBox("Geben Sie die zweite Zahl ein!")

       ergebnis = ersteEingabe + zweiteEingabe

       MsgBox ("Die Summe ist: " & ergebnis)

       ergbnis = ersteEingabe * zweiteEingabe

       MsgBox ("Das Produkt ist: " & ergebnis)


end Sub
```

Abbildung 5.5 zeigt meine Eingaben und die Ausgaben des kurzen Programes.

Das Ergebnis in der ersten `MsgBox` ist schon mal gut. Dadurch daß ich die beiden Variablen *ersteEingabe* und *zweiteEingabe* als Integervariablen deklariert habe, wandelt `InputBox` die eingelesenen Strings in Integers um und weist die Integer-Zahlen auf *ersteEingabe* bzw. *zweiteEingabe* zu. Damit funktioniert die Addition und das erste Ergebnis ist richtig. Das Ergebnis in der zweiten `MsgBox` dann weniger.

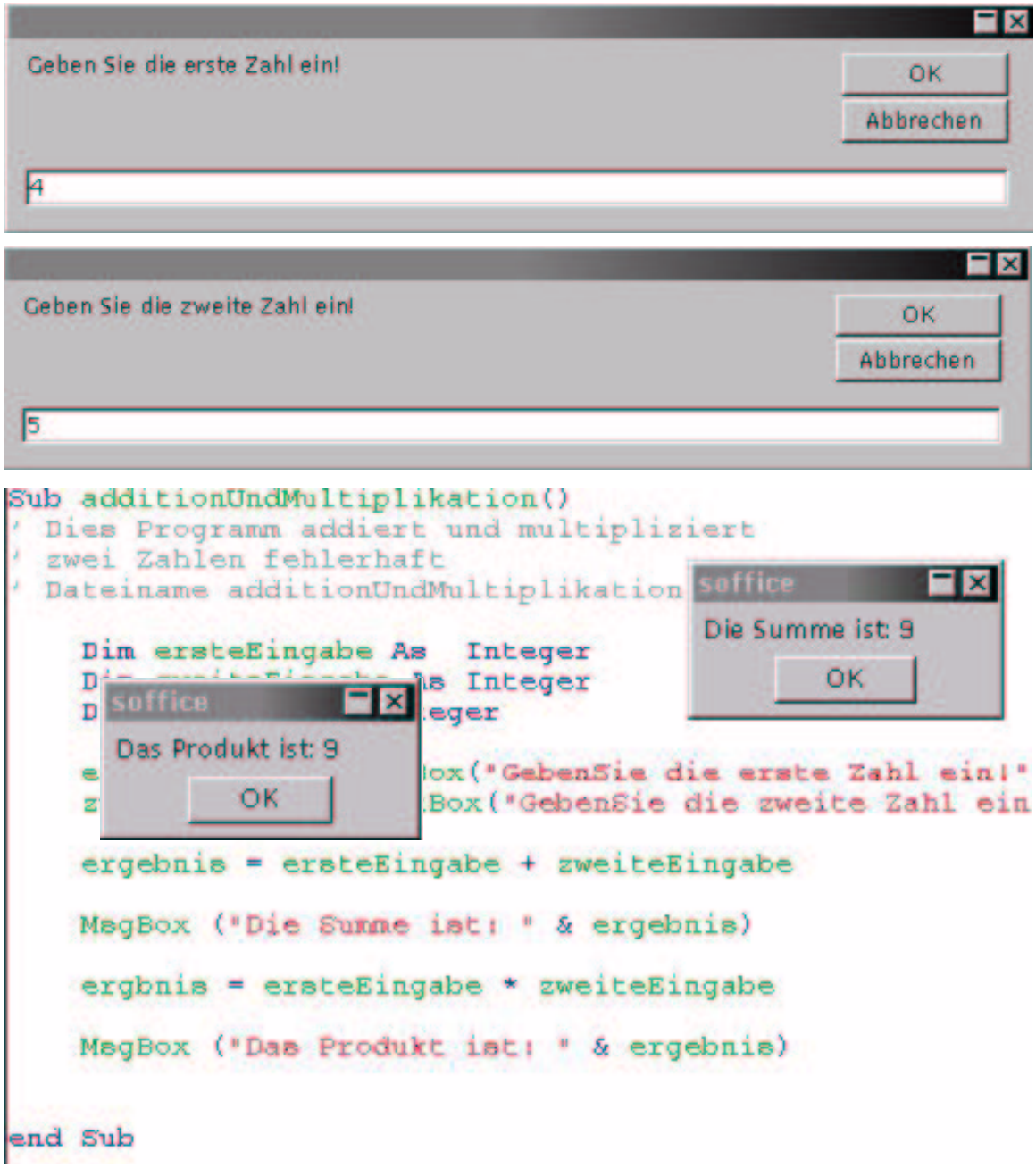

Abbildung 5.5 Die Bildschirmeingaben und die Ausgabe von Beispiel 5.3

Auch das Produkt ist 9. Nun ist aber 4*5 definitiv 20 und nicht 9. Des Rätsels Lösung ist die Zeile:

```
ergbnis = ersteEingabe * zweiteEingabe
```

In dieser Zeile ist mir ein Schreibfehler unterlaufen (*ergbnis* statt *ergebnis*). Und VBA erzeugt einfach eine weitere Variable namens *ergbnis* und weist das Ergebnis der Multiplikation dieser Variablen zu. Ausgegeben wird aber wieder die Variable *ergebnis*:

```
MsgBox ("Das Produkt ist: " & ergebnis)
```

Dadurch, daß Variablen nicht deklariert werden müssen, können Schreibfehler also recht drastische Auswirkungen haben[7].

Das perfide an diesen Fehlern ist, daß der Computer bzw. die VBA-Umgebung diese Fehler nicht finden kann. Die Programme scheinen zu laufen, nur die Ergebnisse sind falsch. Besser sind immer Fehler[8], die von VBA selbst erkannt werden können, wie in Abbildung 5.6. Dort fehlen die schließenden Anführungszeichen. VBA erkennt den Fehler, positioniert einen roten Pfeil an die fehlerhafte Stelle und teilt uns in einem Fehlerfenster mit, welche Art Fehler aufgetreten ist.

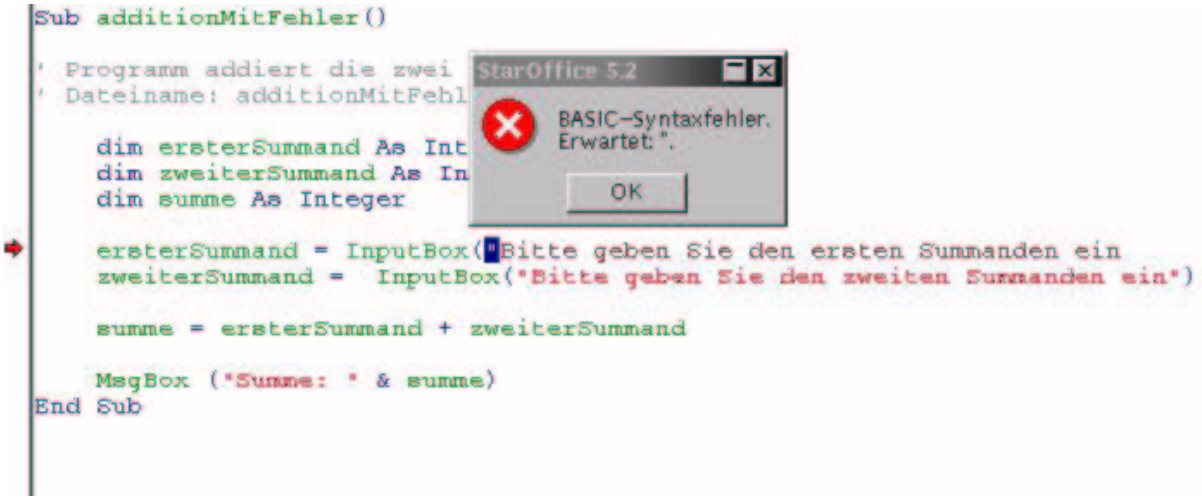

Abbildung 5.6 Von der VBA-Laufzeitumgebung erkannter Fehler

Glücklicherweise können wir aber in VBA die Variablenbehandlung ändern. Fügen wir in unsere Programme als **erste** Anweisung noch vor `Sub Programmname` die Schlüsselworte `Option Explicit` ein, wird Variablendeklaration zwingend vorgeschrieben. Nicht deklarierte Variablen werden mit einer Fehlermeldung zurückgewiesen. Beispiel 5.4 zeigt das verbesserte Programm und Abbildung 5.7 die ausgegebene Fehlermeldung.

Beispiel 5.4 Schreibfehler wird von der VBA-Laufzeitumgebung erkannt

```
Option Explicit
Sub additionUndMultiplikation()
' Dies Programm addiert und multipliziert
' zwei Zahlen fehlerhaft
' Dateiname optionExplicit

      Dim ersteEingabe As  Integer
      Dim zweiteEingabe As Integer
      Dim ergebnis As Integer

      ersteEingabe=InputBox("Geben Sie die erste Zahl ein!")
      zweiteEingabe=InputBox("Geben Sie die zweite Zahl ein!")
```

7.FORTRAN ist eine ältere Programmiersprache, in der Variablen ebenfalls nicht deklariert werden müssen. Der NASA hat ein Schreibfehler in einem FORTRAN-Programm (Programm zur Satellitensteuerung) in den 70er-Jahren 3 Millionen US Dollar (und einen Satelliten) gekostet.

8.Soweit Fehler überhaupt gut sein können..

```
        ergebnis = ersteEingabe + zweiteEingabe

        MsgBox ("Die Summe ist: " & ergebnis)

        ergbnis = ersteEingabe * zweiteEingabe

        MsgBox ("Das Produkt ist: " & ergebnis)

end Sub
```

Unser Programm wird gar nicht ausgeführt Ein Fehlerfenster teilt mit, daß die Variable ergbnis nicht deklariert ist. Die Zeile, in der der Fehler auftritt, ist blau unterlegt.

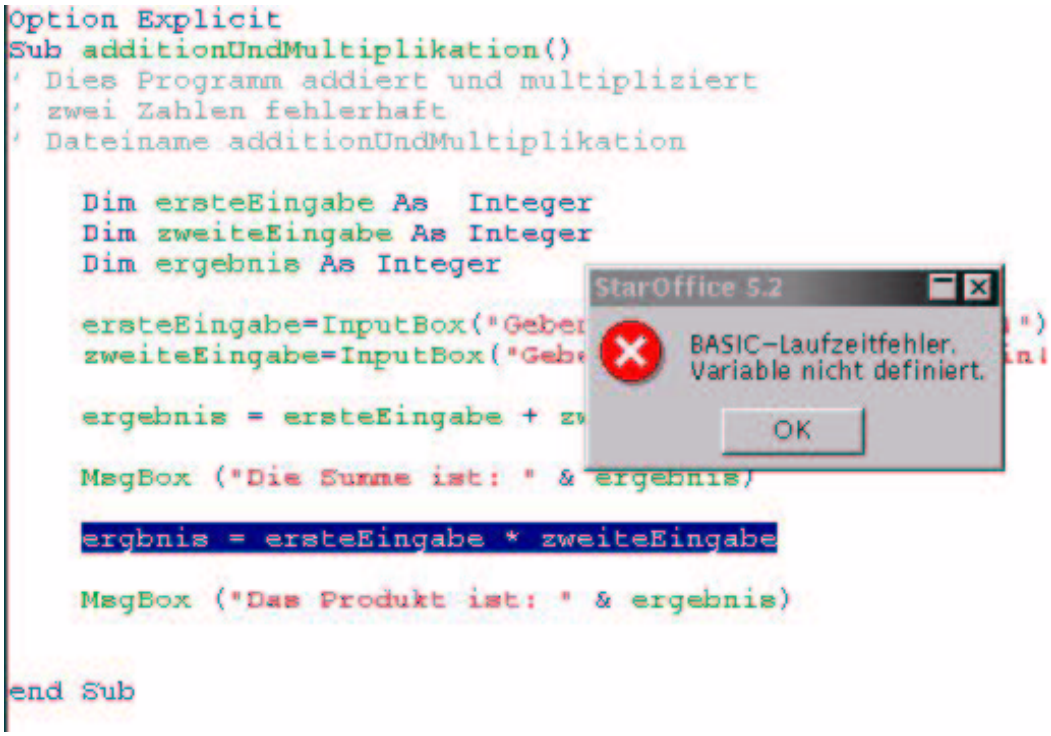

Abbildung 5.7 Nicht deklarierte Variable wird angemeckert

5.2 Regeln zu Variablennamen

Merke: Diese Regeln gelten auch für alle anderen Namen, die Sie selbst vergeben können.

- Variablennamen beginnen mit einem Buchstaben.
- Variablennamen können nach dem ersten Buchstaben Ziffern und Buchstaben in beliebiger Reihenfolge enthalten.
- Variablennamen können beliebig lang sein (naja in Wirklichkeit 255 Zeichen).
- Groß- und Kleinschreibung spielt keine Rolle (*summe* = *Summe* = *suMME* = *SUMME*).

- Variablennamen dürfen außer dem _ keine Sonderzeichen enthalten. Wie in Kapitel 4 weise ich darauf hin, daß deutsche Umlaute erlaubt sind. Das vergessen Sie aber sofort wieder.
- Der Programmname darf kein Variablenname sein.
- In VBA reservierte Worte dürfen keine Variablennamen sein.
- Variablennamen sollten einen Bezug zu den Daten haben, die sie später aufnehmen sollen.

6 Erste einfache Programmstrukturen

6.1 Ausdrücke

Ein Ausdruck ist eine Regel zur Berechnung eines Wertes Das klingt kompliziert, kennen wir aber schon aus unseren vielen Beispielen. Die Summenbildung, die wir ja aus schon so vielen Beispielen kennen, ist ein typischer Ausdruck.

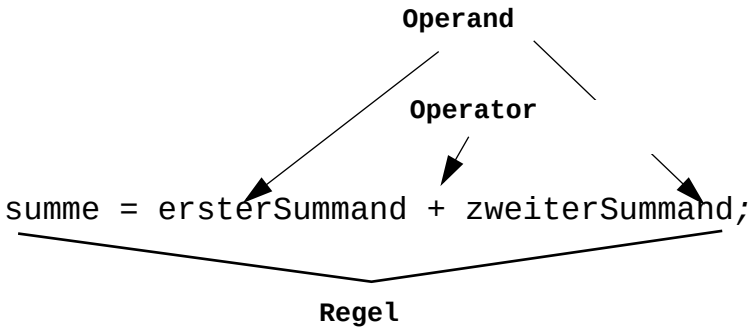

Abbildung 6.1 Operand, Operator und Regel

Ein Ausdruck besteht aus mehreren (oder auch nur einem) Operanden und mehreren (oder auch nur einem) Operator.

Ein Operand kann sein:

- eine Variable (richtiger: ihr Inhalt)
- eine Konstante

```
bruttobetrag = 1.15 * nettobetrag;
```

- ein Funktionsaufruf (-> später)

6.2 Operatoren

Tabelle 6.1 zeigt eine Übersicht der in VBA vorhandenen Operatoren. In den folgenden Abschnitten dieses Kapitels werde ich die Operatoren aus Tabelle 6.1 kurz besprechen.

Tabelle 6.1 Operatoren in VBA (Auswahl)

Rang	Operatoren	
arithmetische Operatoren	-	negatives Vorzeichen
	+, -, *, /	Grundrechenarten
	^	Potenz
	\	Integerdivision
	Mod	Modulo-Operator
Zeichenketten-Operatoren	+	verbindet Zeichenketten
	&	zahlen werden vor der Verbindung in Zeichenketten gewandelt
Vergleichsoperatoren	=, , ,	gleich
	< >	ungleich
	<, <=	kleiner, kleiner gleich
	>, > =	größer, größer gleich
Logische Operatoren	And	logisches und
	Or	logisches Oder
	Xor	exklusives oder (entweder a oder b, aber nicht beide)
	Not	logische Negation
Zuweisungsoperator	=	Zuweisung (vgl. Kapitel 6.4)

6.2.1 Arithmetische Operatoren

Die Wirkungsweise arithmetischer Operatoren werde ich anhand Beispiel 6.1 und Abbildung 6.2 darstellen.

Beispiel 6.1 Arithmetische Operatoren

```
Option Explicit
Sub operatoren()
'Dies ist ein Testprogramm fuer
'Operatoren
'Dateiname: operator
       dim x%
       dim y%
       dim z%
       x = 3
       y = 7
       z = -x
       MsgBox("negatives Vorzeichen - 3 = " & z)
       z = y - x
```

```
        MsgBox("Subtraktion 7 - 3 = " & z)
        z = y + x
        MsgBox("Addition 7 + 3 = " & z)
        z = y * x
        MsgBox("Multiplikation 7 * 3 = " & z)
        z = y / x
        MsgBox("Division 7 / 3 = " & z)
        z = y ^ x
        MsgBox("Potenzieren 7 ^ 3 = " & z)
        z = y Mod x
        MsgBox("Modulo-Bildung 7 Mod 3 = " & z)
        z = y \ x
        MsgBox("Integer Division 7 durch 3 = " & z)
        Dim v#
        v = y / x
        MsgBox ("Division, Zuweisung auf einen Double" & v)
        v = y \ x
        MsgBox ("Integer Division, Zuweisung auf einen Double " & v)
        y = 8
        z = y / x
        MsgBox("Division 8 / 3 = " & z)
        z = y \ x
        MsgBox("Integer Division 8 durch 3 = " + z)
End Sub
```

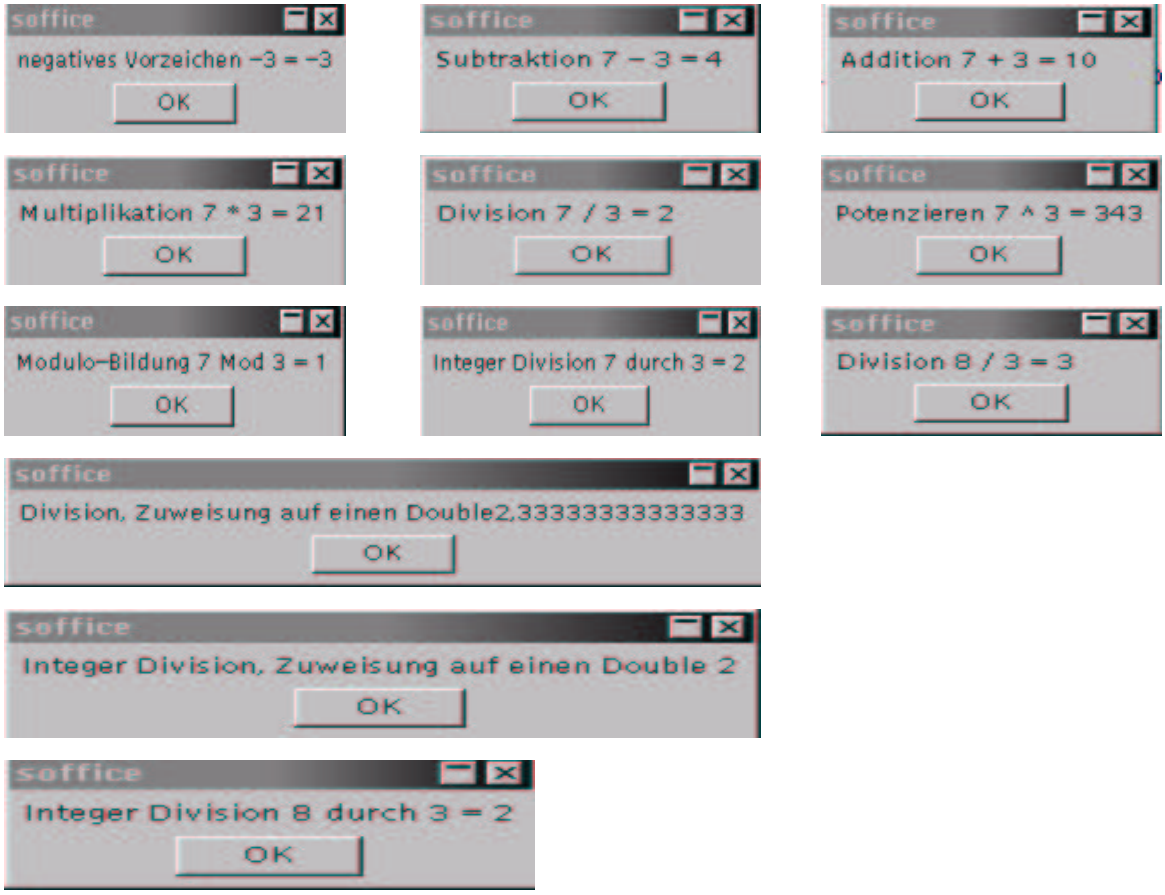

Abbildung 6.2 Die Bildschirmausgaben von Beispiel 6.1

Die Grundrechenarten, potenzieren und das Ändern des Vorzeichens funktionieren so, wie Sie das von der Schule her kennen. Die zunächst vielleicht verwirrenden Ergebnise der beiden Divisionen 7/3 = 2 und 8/3 = 3 ergeben sich, weil die Variable *z*, auf die das Ergebnis zugewiesen wird, vom Typ Integer ist. Integer-Variablen können nur ganze Zahlen aufnehmen. Daher wird das Ergebnis der Division (2,33 bzw. 2,67) gerundet. Daß VBA auch richtig teilen kann, sehen wir, wenn wir das Ergebnis der Division auf eine `double`-Variable zuweisen (`single` hätte es hier natürlich auch getan).

Der Backslash (\) steht für die Integerdivision. Der Modulo-Operator liefert als Ergebnis den Rest dieser Division (Beispiel 5 dividiert durch 3 ist 1 Rest 2, 1 ist das Ergebnis der Integerdivision, 2 das Ergebnis der Modulo-Bildung), Integerdivision und Modulo-Bildung werden in Kapitel 7 auf Seite 43 ausführlich besprochen.

6.2.2 Vergleichsoperatoren

Vergleichsoperatoren werden eigentlich immer in Verbindung mit Kontrollstrukturen und Schleifen genutzt. Beide Programmierkonstrukte wurden bisher noch nicht behandelt. Darum wirken die jetzt vorgestellten Beispiele etwas konstruiert[9]. Um die Ergebnisse der Vergleiche zu visualisieren, benutze ich Variablen vom Typ Boolean. Solche Variablen können nur zwei Werte annehmen: `true` oder `false`. Das Ergebnis eines Vergleichs ist aber auch immer wahr oder falsch. Doch nun zum Beispielprogramm und seinen Ausgaben:

Beispiel 6.2 Vergleichsoperatoren

```
Option Explicit
Sub vergleich()
'Dies ist ein Testprogramm fuer Vergleiche
'Dateiname: vergleich

      dim ergebnis As Boolean
      dim i%, j%, k%
      i = 6
      j = 8
      ergebnis = (i < j)
      MsgBox("Vergleich  6 < 8 " & ergebnis)
      ergebnis = (i > j)
      MsgBox("Vergleich  6 > 8 " & ergebnis)
      ergebnis = (i <= j)
      MsgBox("Vergleich  6 <= 8 " & ergebnis)
      ergebnis = (i >= j)
      MsgBox("Vergleich  6 >= 8 " & ergebnis)
      ergebnis = (i = j)
      MsgBox("Vergleich  6 = 8 " & ergebnis)
      ergebnis = (i <> j)
      MsgBox("Vergleich  6 ungleich 8 " & ergebnis)
End Sub
```

9.Dies ist das Henne-Ei-Problem. Kontrollstrukturen kann man nicht ohne Vergleiche erklären und Vergleiche schlecht ohne Kontrollstrukturen.

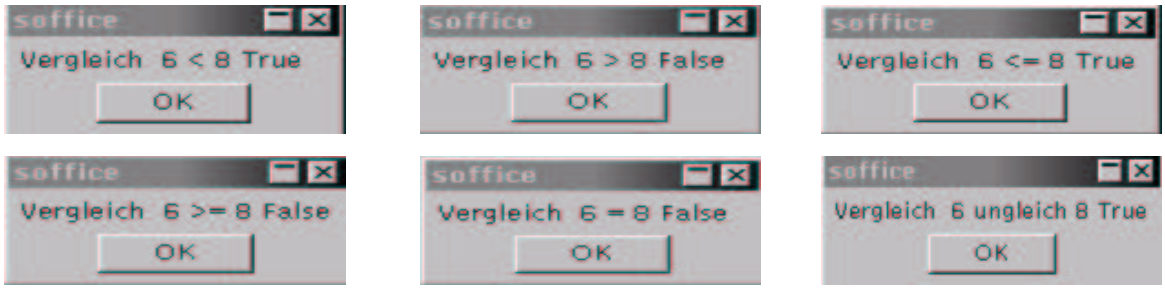

Abbildung 6.3 Die Bildschirmausgaben von Beispiel 6.2

Zeilen, wie

```
ergebnis = (i < j)
```

haben wir in dieser Form noch nicht besprochen. Dennoch ist auch dies nur ein einfacher Ausdruck. Links steht eine Variable vom Typ Boolean. Rechts sehen wir zwei weitere Variablen und zwischen ihnen den Kleiner-Operator.

Was nun passiert, ist recht einfach. VBA führt den Vergleich durch. Ist das Ergebnis wahr (*i* ist kleiner als *j*, wie in unserem Fall), wird der Variable vom Typ Boolean (kürzen wir ab jetzt mit bool'scher Variable ab) `true` zugewiesen, stimmt der Vergleich nicht `false`.

Ansonsten sehen wir, daß sich Vergleichsoperatoren so verhalten, wie wir das erwarten.

6.2.3 Logische Operatoren

Logische Operatoren werden im wesentlichen dazu benutzt, um zwei Vergleiche zu verknüpfen. Darum trifft das, was ich in der Einleitung zu Kapitel 6.2.2 gesagt habe, auch hier zu. Betrachten wir auch hier ein Beispiel:

Beispiel 6.3 Logische Operatoren

```
Option Explicit
Sub logischeOperatoren()
'Dies ist ein Testprogramm fuer logische
'Operatoren
'Dateiname: logOperatoren

       dim i%
       dim j%
       dim k%
       dim l%
       i = 1
       j = 1
       k = 3
       l = 4
       dim ergebnis As Boolean
       ergebnis = (i = j) And (k < l)
       Msgbox ("(1 = 1) And (3 < 4) " & ergebnis)
       ergebnis = (i < j) And (k < l)
       Msgbox ("(1 < 1) And (3 < 4) " & ergebnis)
       ergebnis = (i = j) Or (k < l)
```

```
        Msgbox ("(1 = 1) Or (3 < 4) " & ergebnis)
        ergebnis = (i > j) Or (k = l)
        Msgbox ("(1 > 1) Or (3 = 4) " & ergebnis)
        ergebnis = (i = j) Xor (k < l)
        Msgbox ("(1 = 1) Xor (3 < 4) " & ergebnis)
        ergebnis = Not (i=j)
        MsgBox("Not (1=1) " & ergebnis)
End Sub
```

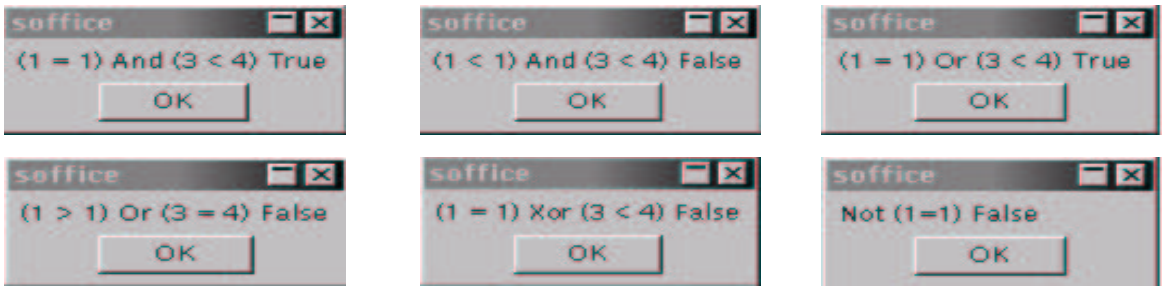

Abbildung 6.4 Die Bildschirmausgaben von Beispiel 6.3

Die Zeile

```
        ergebnis = (i = j) And (k < l)
```

ähnelt den Ausdrücken des vorherigen Kapitels stark. Der einzige Unterschied besteht darin, daß wir auf der rechten Seite des Ausdrucks zwei Vergleiche sehen. Beide Vergleiche sind durch einen logischen Operator (in diesem Beispiel And) verbunden. VBA wertet zunächst beide logischen Ausdrücke aus und danach die Verbindung. Das Ergebnis wird der bool'schen Variable *ergebnis* zugewiesen.

Die folgenden Tabelle zeigen die möglichen Ergebnisse der logischen Operatoren. Tabelle 6.2 ist folgendermaßen zu lesen: Wenn *a* den Wert `true` besitzt und *b* ebenfalls, dann ist *a* `And` *b* `true`. Wenn *a* `true` ist und *b* `false`, dann ist *a* `And` *b* `false`. Die folgenden Tabellen sind analog zu lesen.

.

Tabelle 6.2 Der And Operator

a \ b	true	false
true	true	false
false	false	false

Tabelle 6.3 Der Or Operator

a \ b	true	false
true	true	true
false	true	false

Tabelle 6.4 Der Xor Operator

a \ b	true	false
true	false	true
false	true	false

Tabelle 6.5 Der Not Operator

a	Not a
true	false
false	true

6.2.4 Zeichenketten-Operatoren

Die beiden Zeichenketten-Operatoren dienen dazu, zwei Zeichenketten (Strings) zu verknüpfen. Sie unterscheiden sich in ihrer Wirkungsweise. Der +-Operator verknüpft zwei String-Variablen, beide Variablen müssen vom Typ String sein. Der &-Operator verknüpft ebenfalls zwei Strings, nur wandelt er Variablen, die nicht vom Typ String sind, vor der Verknüpfung in Strings um. Den &-Operator haben wir eigentlich in allen unseren Beispielen bisher genutzt. Beispiel 6.4 und Abbildung 6.5 zeigen weitere Beispiele für die Zeichenketten-Operatoren und die zugehörigen Bildschirmausgaben.

Beispiel 6.4 Die Zeichenketten-Operatoren

```
Option Explicit
Sub zeichenkettenOperatoren()
'Dies ist ein Testprogramm fuer
'Zeichenketten-Operatoren
'Dateiname: zeichenkettenOperator

       dim s1$
       dim s2$
       dim s3$
       dim i%
       s1="abcd"
       s2="efg"
       i = 9
       s3 = s1 + s2
       MsgBox("Strings verbinden: " + s3)
       s3 = s1 & i
       MsgBox("String mit Integer verbinden: " & s3)
End Sub
```

Abbildung 6.5 Die Bildschirmausgaben von Beispiel 6.4

6.2.5 Rangfolge von Operatoren

Operatoren besitzen eine Rangfolge. Sie werden zunächst dem Rang nach ausgeführt. Wir kennen dies ja auch aus der Mathematik (z.B. Punktrechnung geht vor Strichrechnung). So ist es auch hier. Multiplikationen werden vor Additionen durchgeführt.

Operatoren gleichen Ranges werden von links nach rechts abgearbeitet.

Die Rangfolge kann durch Klammerung geändert werden.

Beispiele

6 * 2 + 4 * 3 = 24

6 * (2 + 4) * 3= 108

5 + 3 * 7 = 26

(5 + 3) * 7 = 56

6.3 Anweisungen

Anweisungen entsprechen den Aktionen des Programms. Wir haben in den bisherigen Beispielen schon einige Anweisungen kennengelernt, so z.B.:

- *sub Programmname* sagt, daß die Verarbeitung des Programms beginnen soll.
- *end sub* Beendet das Programm.

6.4 Die Zuweisung (Ergibt-Anweisung)

Eine der wichtigsten Anweisungen ist die Zuweisung. Die Zuweisung (oder der Zuweisungsoperator) ist einfach das Gleicheitszeichen.

```
summe = ersterSummand + zweiterSummand
```

Folge: Der Ausdruck auf der rechten Seite wird ausgewertet (berechnet) und das Ergebnis wird der Variablen auf der linken Seite zugewiesen. Daher muß auf der rechten Seite ein Ausdruck und auf der linken Seite eine Variable stehen.

Merke: Der Ergibt-Operator (anderer Name für das Gleichheitszeichen) überschreibt. Das bedeutet, daß der vorherige Wert von `summe` vom neuen Wert (dem berechneten Wert) erstetzt wird.

Merke: Variablen können daher im Programmablauf verschiedene Werte haben.

Beispiel 6.5 Beispiele für Zuweisungen

```
Option Explicit

sub zuweisung()
'Beispielprogramm fuer zuweisungen
'Dateiname: zuweisung
```

```
dim x As Integer
dim y As Integer
dim z As Integer
x = 2                'x hat Wert 2, y Wert ?, z Wert ?)
MsgBox("Ausgabe 1: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) &  _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)

y = 3                'x hat Wert 2, y hat Wert 3, z Wert ?)
MsgBox("Ausgabe 2: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) & _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)

z = x + y     'x hat Wert 2, y hat Wert 3, z hat Wert 5)
MsgBox("Ausgabe 3: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) & _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)

x = 4                'x : 4, y : 3, z : 5)
MsgBox("Ausgabe 4: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) & _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)

x = x + 7     'x : 11, y : 3, z : 5)
MsgBox("Ausgabe 5: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) & _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)

y = x                'x : 11, y : 11, z : 5)
MsgBox("Ausgabe 6: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) & _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)

y = z                'x : 11, y : 5; z : 5)
MsgBox("Ausgabe 7: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) & _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)

z = 7         'x : 11, y : 5, z : 7)
MsgBox("Ausgabe 8: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) & _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)

x = z + 7 - 8'x : 6, y : 5, z : 7)
MsgBox("Ausgabe 9: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) & _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)

y = z                'x : 6, y : 7, z : 7)
MsgBox("Ausgabe 10: " & Chr$(13) & _
             "Wert von x: " & x & Chr$(13) & _
             "Wert von y: " & y & Chr$(13) & _
             "Wert von z: " & z)
```

```
end  sub
```

Dies Programm erzeugt folgende Ausgaben:

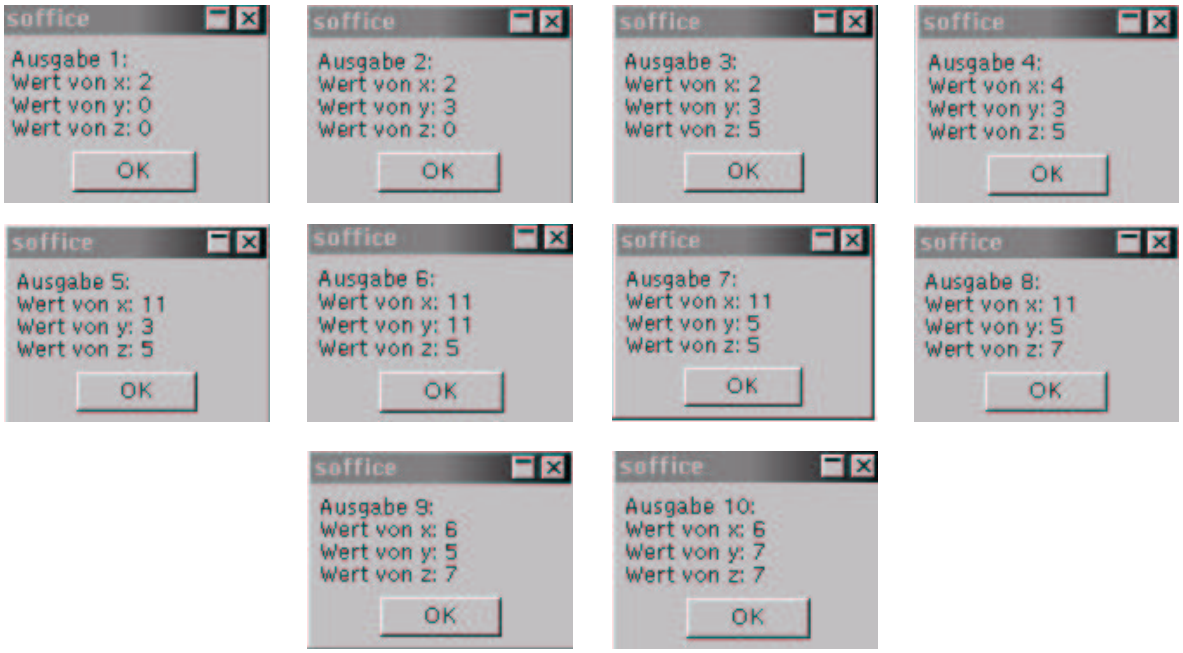

Abbildung 6.6 Die Ausgabe von Beispiel 6.5

Beispiel 6.5 überrascht uns mit kompliziert aussehenden `MsgBox`-Anweisungen:

```
MsgBox("Ausgabe 5: " & Chr$(13) & _
                    "Wert von x: " & x & Chr$(13) & _
                    "Wert von y: " & y & Chr$(13) & _
                    "Wert von z: " & z)
```

Aber auch diese wie schwarze Magie aussehenden Zeilen können wir recht einfach verstehen. Bislang (vor diesem Beispiel) erfolgten alle Ausgaben in den `MsgBox`-en in einer Zeile. Da wir hier aber die Werte von 3 Variablen ausgeben, ist es schöner, wenn die Ausgaben der einzelnen Variablen untereinander erfolgen. Wir müssen also der `MsgBox` irgendwie mitteilen, daß jetzt eine neue Zeile kommen soll. Dies macht man mit `Chr$(13)`[10].

Des weiteren ist der Text innerhalb der Klammern der `MsgBox` zu lang, um ihn in eine Zeile zu schreiben. Wir möchten das Kommando `MsgBox` also über mehrere Zeilen verteilen. Dazu dient der _ (Unterstrich). Der _ teilt VBA mit, daß das Kommando noch nicht zu Ende ist, sondern in der nächsten Zeile fortgesetzt wird. Ja und die `&` benötigen wir, um die einzelnen Ausgaben miteinander zu verketten.

Den übrigen Quellcode von Beispiel 6.5 will ich nicht erläutern, er besteht nur aus Zuweisungen und wie oben beschriebenen Ausgabefenstern. An den Ausgaben sehen Sie aber folgendes:

10.Chr$() ist in Wirklichkeit eine Funktion. Warum genau Chr$(13) zu einem Zeilenvorschub führt, werden Sie später erlernen. Was eine Funktion ist auch.

- VBA belegt nicht definierte Integer-Variablen mit dem Wert 0 vor. Nach der ersten Zuweisung hat nur die Variable x einen definierten Wert. Im Ausgabefenster Ausgabe 1 erscheinen aber als Werte für y und z jeweils 0.

- Die Zuweisung hat nichts mit dem mathematischen Gleichheitszeichen zu tun. Dies können wir schön an der Codezeile:

  ```
  x = x + 7
  ```

  sehen. Denn würden wir diese Zuweisung wie eine Gleichung behandeln, könnten wir x auf beiden Seiten subtrahieren:

  0 = x + 7 - x

  und daher:

  0 = 7,

  was ja so sicher nicht stimmt. Bei einer Zuweisung wird zunächst der Ausdruck auf der rechten Seite der Zuweisung ausgewertet (In unserem Beispiel wird 7 zum Wert der Variable x hinzuaddiert). Dann wird das Resultat der Berechnung der Variable x zugewiesen, x ändert dadurch seinen Wert.

- Wir sehen, wie sich die Werte der Variablen x, y und z im Programmverlauf beständig ändern.

6.5 Einfache Datenein- und -ausgabe

Programme manipulieren Daten (*ersterSummand*, *zweiterSummand* in Beispiel 2.3).

Daten müssen daher an das Programm übergeben werden.

Wir würden das Ergebnis der Manipulationen/Berechnungen gern sehen =>

Das Programm muß Daten an uns zurückgeben können.

VBA stellt dafür die Funktionen `InputBox` und `MsgBox` zur Verfügung. Die Nutzung dieser Funktionen haben wir schon an vielen Beispielen gesehen, so daß ich auf eine erneute Diskussion der beiden Funktionen verzichte.

7 Das Raketenbeispiel r

Ich beginne jetzt mit dem Beispiel, das sich durch alle Veranstaltungen hindurchziehen wird. In den Übungen werden Sie den folgenden Stoff immer anhand dieses, allerdings etwas abgewandelten Beispiels einüben. Um einen guten Start zu haben, stelle ich die Grundgedanken hier vor.

Exkurs: Integerdivision und Modulo-Bildung

Dem Algorithmus zur Problemlösung des folgenden Beispiels liegt Integerdivision und Modulo-Bildung zugrunde. Das hat an sich wenig mit Informatik zu tun. Es ist eher elementare Mathematik. Zum besseren Verständnis des folgenden werde ich die Grundgedanken kurz erläutern.

Wenn wir zwei Zahen durcheinander teilen, ist das Ergebnis entweder eine ganze Zahl (so z.B. wenn wir 6 / 3 rechnen, kommt 2 raus) oder ein Bruch. So ist 10/4 = 2,25.

Es gibt aber Situationen, in denen wir an den Nachkommastellen nicht interessiert sind (wie das folgende Beispiel zeigen wird) und eigentlich nur den ganzzahligen Anteil der Lösung nutzen wollen. Dies nennt man Integerdivision. VBA benutzt für die Integerdivision den Operator \. so ist 10 \ 4 = 2. Oder umgangssprachlich formuliert: Integerdivision beantwortet die Frage: Wie oft geht der Nenner in den Zähler. Und in obigem Beispiel: Wie oft geht die 4 in die 10. Übrigens ist auch 11 \ 4 = 2.

Wenn man eine Integerdivision durchführt, bleibt ein Rest (der Nachkommaanteil der "normalen" Division). Diesen erhält man, indem man das Ergebnis der Integerdivision mit dem Nenner multipliziert und das Ergebnis der Multiplikation vom Zähler abzieht. In unserem ersten Beispiel ist der Rest 0, denn 6 \3 = 2. Wir können das einfach ausrechnen, denn der Nenner ist hier 3, der Zähler 6 und das Ergebnis der Integerdivision ist 2. Mit obiger Formel erhalten wir also: Rest von 6\3 = 6 - 3 * 6\3 = 6 - 3*2 =6 - 6=0. Im zweiten Beispiel ist der Rest 2 (Rest von 10\4=10-4*10\4=10-4*2=10-8=2) und im dritten Beispiel 3 (Rest von 11\4=11-4*11\4=11-4*2=11-8=3). Dies nennt man in der Mathematik Modulo-Bildung und VBA verfügt dafür über den Operator mod. So ist also 6 mod 3 = 0, 10 mod 4 = 2 und 11 mod 4 = 3.

Ein Beispiel, wo man das anwenden kann (dies ist auch unser Beispiel), ist wenn wir wissen wie lange ein Vorgang in Sekunden dauert (z.B. 128 Sekunden) und wir wollen wissen, wieviele Minuten und Sekunden das sind. Die Beantwortung dieser Frage ist aber genau Integerdivision und Modulo-Bildung. Denn die Minuten sind die Beantwortung der Frage: Wie oft geht die 60[11] in die uns gegebenen Sekunden (z.B. wie oft geht die 60 in die 128) und die Sekunden sind der Rest, der übrigbleibt, wenn ich das gemacht habe. Also 128 \ 60 = 2 und 128 mod 60 = 8, daher sind 128 Sekunden 2 Minuten 8 Sekunden.

Problemstellung 7.1 Das Raketenbeispiel

Ein Programm soll 3 Werte einlesen, die den Startzeitpunkt einer Rakete in Stunden, Minuten und Sekunden angeben. Danach soll die Flugzeit (in Sekunden) eingelesen werden. Dann soll aus diesen Angaben die Ankunftszeit der Rakete berechnet werden und in einem Fenster ausgegeben werden. Als erstes soll das Programm eine Bedie-

11. Eine Minute hat 60 Sekunden :-).

nungsanleitung ausgeben, damit zukünftige Nutzer leichter mit dem Programm umgehen können. Wir machen zur Zeit noch eine Einschränkung: Start und Landung der Rakete finden am gleichen Tag statt.

Lösung:

Zunächst müssen wir uns den Algorithmus überlegen und dokumentieren. Hier lehnen wir uns zunächst einmal eng an die Algorithmen aus Kapitel 3 an. Wir überlegen uns also, was muß unser Programm ganz grob tun. Die Antwort ist: Es muß die Bedienungsanleitung ausgeben, dann die Eingaben einlesen, dann die Ankunftszeit berechnen und zum Schluß die berechnete Ankunftszeit wieder ausgeben. Der Algorithmus dargestellt als Nassi-Shneidermann Diagramm:

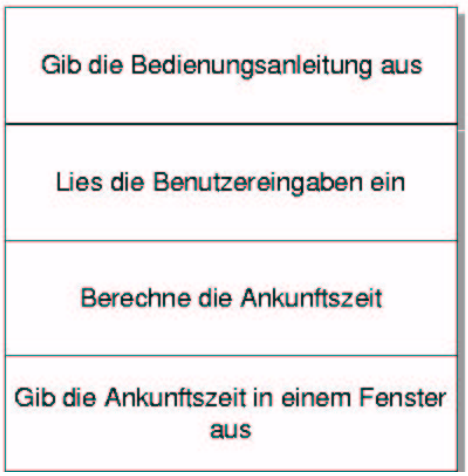

Abbildung 7.1 Nassi-Shneidermann Diagramm zur Berechnung der Raketenankunftszeit

Der Algorithmus in Pseudocode:

Pseudocode 7.1 Der Algorithmus zum Raketenbeispiel

```
Gib die Bedienungsanleitung aus
Lies die Benutzereingaben ein
Berechne die Ankunftszeit
Gib die Ankunftszeit in einem Fenster aus
```

Wir sind jetzt schon einen ganzen Schritt weiter. "Gib die Bedienungsanleitung aus" ist einfach, das regeln wir mit einer `MsgBox`. "Lies die Benutzereingaben ein" können wir auch, das machen wir mit `InputBox`-en. "Gib die Ankunftszeit in einem Fenster aus" können wir wieder mit einer `MsgBox` realisieren.

"Berechne die Ankunftszeit" ist nicht so einfach. Dies können wir zumindestens nicht 1 zu 1 in ein Programm umsetzen. Dies bedeutet, wir müssen einen Algorithmus für "Berechne die Ankunftszeit" finden. Diese Technik heißt “schrittweise Verfeinerung”. Wir beschreiben zunächst die Problemlösung in allgemeiner Form (vgl. Pseudocode 7.1). Sind Teile der Problemlösung (Zeile “Berechne die Ankunftszeit” in Pseudocode 7.1) zu allgemein, um direkt in VBA übersetzt zu werden, betrachten wir diesen Teil separat und beschreiben die Lösung des Teilproblems.

Nichtsdestotrotz müssen wir allerdings das Teilproblem lösen. Und wenn wir einen Algorithmus entwerfen wollen, aber nicht so recht weiterwissen, dann betrachten wir ein Beispiel.

Also nehmen wir an, unser Benutzer hat 13:56:12 Uhr als Abflugzeit und 4365 Sekunden als Flugzeit eingegeben. Wir haben in beiden Zeiten Sekunden, die 12 Sekunden der Abflugzeit und die 4365 Sekunden der Flugzeit. Wir könnten also diese beiden Zeiten addieren und erhalten 4377 Sekunden. Wir können jetzt mit Fug und Recht sagen:

Die Rakete landet um 13:56:4377.

Dies ist aber noch nicht recht befriedigend, weil keiner außer uns Zeiten so angibt. Wir müssen also entweder einen neuen Standard für Zeitformate setzen oder weitermachen. Wir machen weiter. Der naheliegende Schritt ist, sich zu überlegen: Wieviele Minuten sind wohl in den 4377 Sekunden?

Dies können wir aber leicht ausrechnen, denn wie wir wissen ist das die Integerdivision

4377 \ 60.

Dies ist aber 72. In unseren 4377 Sekunden sind also 72 Minuten. Als nächstes berechnen wir, wieviel Sekunden da wohl übrigbleiben, das ist das Ergebnis der Modulobildung, also

übrigbleibende Sekunden = 4377-60*4377 \60=4377-60*72=4377-4320=57

Unsere komischen 4377 Sekunden in der Ankunftszeit sind also in Wahrheit 72 Minuten und 57 Sekunden. Beachten Sie, daß wir die "wahren" Sekunden der Ankunftszeit jetzt bereits kennen, unsere Rakete kommt um 57 Sekunden an. Fragt sich nur zu welcher Stunde und Minute. Wir haben aber die Minuten in den 4377 Sekunden der Ankunftszeit bereits ausgerechnet (waren 72) und wir haben ja bereits Minuten (56) in der Ankunftszeit. Wir könnten also wie oben vorgehen und das einfach mal addieren. Das tun wir und erhalten 128 Minuten. Also behaupten wir jetzt:

Die Rakete landet um 13:128:57.

Leicht besser, aber immer noch nicht gut genug. Aber jetzt gehen wir analog zu unserer Sekunden-Problemlösung vor. Wir überlegen uns einfach: Wieviele Stunden sind in den Minuten?

Dies können wir aber leicht ausrechnen, denn wie wir wissen ist das die Integerdivision

128 \ 60[12].

Dies ist aber 2. In unseren 128 Minuten sind also 2 Stunden. Als nächstes berechnen wir, wieviel Minuten da wohl übrigbleiben, das ist das Ergebnis der Modulobildung, also

übrigbleibende Minuten = 128-60*128 \60=128-60*2=128-120=2

Unsere komischen 128 Minuten in der Ankunftszeit sind also in Wahrheit 2 Stunden und 8 Minuten. Beachten Sie, daß wir die "wahren" Minuten der Ankunftszeit jetzt be-

12.Weil eine Stunde 60 Minuten hat :-).

reits kennen, unsere Rakete kommt um 2 Minuten an. Da wir die Sekunden der Ankunftszeit auch schon kennen, können wir sogar sagen: die Rakete kommt um 2 Minuten 57 Sekunden an. Fragt sich nur, zu welcher Stunde. Wir haben aber die Stunden in den 128 Minuten der Ankunftszeit bereits ausgerechnet (waren 2) und wir haben ja bereits Stunden (13) in der Ankunftszeit. Wir könnten also wie oben vorgehen und einfach mal addieren. Das tun wir und erhalten 15 Stunden . Also behaupten wir jetzt:

Die Rakete landet um 15:02:57.

Und das ist perfekt. Wir haben also unser Problem an einem Beispiel gelöst. Nun generalisieren wir das Ganze und kommen so zu unserem Algorithmus und der Dokumentation in Pseudocode. Dies entsteht einfach dadurch, daß wir die einzelnen in unserem Zahlenbeispiel ausgeführten Arbeitsschritte aufschreiben.

Zunächst haben wir die Flugzeit in Sekunden zu den Sekunden der Abflugzeit addiert und behauptet (was ja auch stimmt), das dies die Ankunftszeit ist. Also:

> addiere die Flugzeit zu den Sekunden der Abflugzeit.

Dann haben wir die Minuten in den entstehenden Sekunden ermittelt (durch Integerdivision) und das Ergebnis den Minuten der Abflugzeit zuaddiert. Also:

> addiere die Minuten in den errechneten Sekunden (Integerdivision der errechneten Sekunden durch 60) zu den Minuten der Abflugzeit

Danach haben wir die übrigbleibenden Sekunden ermittelt (durch Modulobildung) und hatten so die Sekunden des Landezeitpunkts.

> weise den Rest obiger Integerdivision (Modulo-Bildung) den auszugebenden Sekunden zu

Daraufhin haben wir die Stunden in den entstehenden Minuten ermittelt (durch Integerdivision und das Ergebnis den Minuten der Abflugzeit zuaddiert. Also:

> addiere die Stunden in den errechneten Minuten (Integerdivision der errechneten Minuten durch 60) zu den Stunden der Abflugzeit

Im weiteren Verlauf haben wir die übrigbleibenden Minuten ermittelt (durch Modulobildung) und hatten so die Minuten des Landezeitpunkts.

> weise den Rest obiger Integerdivision (Modulo-Bildung) den auszugebenden Minuten zu

Abbildung 7.2 und Pseudocode 7.2 fassen obige Ausführungen zusammen.

Pseudocode 7.2 Berechne die Ankunftszeit

```
addiere die Flugzeit zu den Sekunden der Abflugzeit
addiere die Minuten in den errechneten Sekunden (Integerdivision der errech-
neten Sekunden durch 60) zu den Minuten der Abflugzeit
weise den Rest obiger Integerdivision (Modulo-Bildung) den auszugebenden Se-
kunden zu
addiere die Stunden in den errechneten Minuten (Integerdivision der errechneten
Minuten durch 60) zu den Stunden der Abflugzeit;
weise den Rest obiger Integerdivision (Modulo-Bildung) den auszugebenden Mi-
nuten zu
```

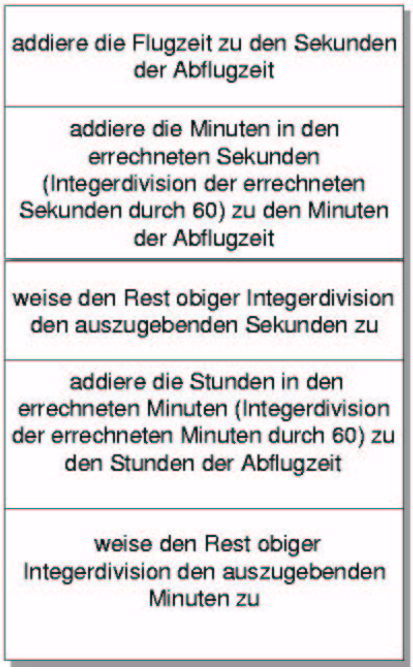

Abbildung 7.2 Nassi-Shneidermann Diagramm “Berechne die Ankunftszeit

Das einzige Problem des obigen Algorithmus entsteht, wenn die Stunden durch die Addition der Stunden in den Minuten größer als 24 werden. Das kann aber nicht passieren, da in der Problemstellung steht, daß die Rakete am gleichen Tag startet und landet.

Nun können wir die einzelnen Teile programmieren. Wir beginnen mit "Gib die Bedienungsanleitung aus".

Realisierung 7.1 Gib die Bedienungsanleitung aus

```
MsgBox("Bitte geben Sie die Startzeit der Rakete in Stunden,“ &Chr$(13) & _
                    „Minuten und Sekunden  an. Die Flugzeit wird in Sekunden
                    „& Chr$(13) & _
                     „eingegeben. Die Ausgabe erfolgt wieder in Stunden,Minuten
                    und Sekunden“)
```

Als nächstes kommt "Lies die Benutzereingaben ein".

Realisierung 7.2 Lies die Benutzereingaben ein

```
stunden = InputBox(„Geben Sie nun die Stunde der Abflugzeit ein!“)
minuten =  InputBox(„Geben Sie nun die Minuten der Abflugzeit ein!“)
sekunden =  InputBox(„Geben Sie nun die Sekunden der Abflugzeit ein!“)
flugzeit = InputBox („Geben Sie nun die Flugzeit in Sekunden ein!“)
```

Nun kommt die Realisierung von "Berechne die Ankunftszeit". Hier benutzen wir Pseudocode 7.2. Wir bilden einfach die einzelnen Anweisungen des Pseudocodes in VBA-Anweisungen ab. Um dies ganz klar zu machen wiederhole ich Pseudocode 7.2 an dieser Stelle noch einmal und schreibe die entsprechenden VBA-Anweisungen hinter die Anweisungen des Pseudocodes.

Pseudocode 7.3 Berechne die Ankunftszeit mit VBA-Anweisungen

```
addiere die Flugzeit zu den Sekunden der Abflugzeit
VBA: sekunden = sekunden + flugzeit
addiere die Minuten in den errechneten Sekunden (Integerdivision der errech-
neten Sekunden durch 60) zu den Minuten der Abflugzeit
VBA: minuten = minuten + sekunden \ 60
weise den Rest obiger Integerdivision (Modulo-Bildung) den auszugebenden Se-
kunden zu
VBA: sekunden = sekunden mod 60
addiere die Stunden in den errechneten Minuten (Integerdivision der errechneten
Minuten durch 60) zu den Stunden der Abflugzeit
VBA: stunden = stunden \ 60
weise den Rest obiger Integerdivision (Modulo-Bildung) den auszugebenden Mi-
nuten zu
VBA: minuten = minuten mod 60
```

Das heißt, wir erhalten als Realisierung:

Realisierung 7.3 Berechne die Ankunftszeit

```
sekunden = sekunden + flugzeit
minuten = minuten + sekunden \ 60
sekunden = sekunden mod 60
stunden = stunden \ 60
minuten = minuten mod 60
```

Nun folgt das Ausgeben des Ergebnisses:

Realisierung 7.4 Gib die Ankunftszeit in einem Fenster aus

```
MsgBox(„Die Rakete landet um: „ & stunden & „:“ & _
                    minuten   & „:“ &sekunden)
```

Nun müssen wir noch die Variablen deklarieren.

Realisierung 7.5 Deklaration der Variablen

```
dim stunden As Integer
dim minuten As Integer
dim sekunden As Integer
dim flugzeit As Integer
```

Durch Zusammensetzen aller Teile erhalten wir das fertige Programm.

Realisierung 7.6 Das Raketenbeispiel: Programm fertiggestellt

```
Option Explicit

Sub raketenBeispiel1()
'Dieses Programm berechnet die Ankunftszeit einer Rakete.
'Die Startzeit wird in Stunden, Minuten, Sekunden eingegeben.
'Die Flugzeit wird in Sekunden eingegeben.
'Dateiname: raketenBeispiel1

'Deklaration der Variablen

       dim stunden As Integer
       dim minuten As Integer
       dim sekunden As Integer
       dim flugzeit As Integer
```

```
'Gib die Bedienungsanleitung aus

       MsgBox("Bitte geben Sie die Startzeit der Rakete in Stunden," & Chr$(13) & _
              "Minuten und Sekunden  an. Die Flugzeit wird in Sekunden "& Chr$(13) & _
              "eingegeben. Die Ausgabe erfolgt wieder in Stunden,Minuten und " _
              " Sekunden")

'Lies die Benutzereingaben ein

       stunden = InputBox("Geben Sie nun die Stunde der Abflugzeit ein!")
       minuten =  InputBox("Geben Sie nun die Minuten der Abflugzeit ein!")
       sekunden =  InputBox("Geben Sie nun die Sekunden der Abflugzeit ein!")
       flugzeit = InputBox ("Geben Sie nun die Flugzeit in Sekunden ein!")

'Berechne die Ankunftszeit

       sekunden = sekunden + flugzeit

       'addiere die Minuten in den neu berechneten Sekunden
       'den Minuten der Abflugzeit hinzu.
       'Die Minuten in den neu berechneten Sekunden ist
       'das Ergebnis der Integerdivision der neu berechneten
       'Sekunden durch 60.

       minuten = minuten + sekunden \ 60

       'okay, die Minuten wurden aus den neu berechneten
       'Sekunden extrahiert. Berechne die uebrig bleibenden
       'Sekunden und weise sie den Ergebnissekunden zu.
       'Beachte: Das Ergebnis wird auch auf der
       'Variable sekunden abgelegt.
       'Dies ist das Ergebnis der Modulo-Operation
       'neu berechnete sekunden modulo 60

       sekunden = sekunden Mod 60

       'addiere die Stunden  in den neu berechneten Minuten
       'den Stunden  der Abflugzeit hinzu.
       'Die Stunden  in den neu berechneten Minuten ist
       'das Ergebnis der Integerdivision der neu berechneten
       'Minuten durch 60.

       stunden = stunden + minuten \ 60

       'okay, die Stunden wurden aus den neu berechneten
       'Minuten extrahiert. Berechne die uebrig bleibenden
       'Minuten und weise sie den Ergebnisminuten zu.
       'Beachte: Das Ergebnis wird auch auf der
       'Variable minuten abgelegt.
       'Dies ist das Ergebnis der Modulo-Operation
       'neu berechnete minuten modulo 60

       minuten = minuten Mod 60

       'die Rakete landet am geleichen Tag, also
       'muessen die Stunden in Ordnung sein,
       'wir sind durch

'Gib die Ankunftszeit in einem Fenster aus

       MsgBox("Die Rakete landet um: " & stunden & ":" & _
                           minuten   & ":" &sekunden)
```

```
End Sub
```

Sie sehen, daß ich in der Endfassung des Programms reichlich Gebrauch von Kommentaren gemacht habe. Zunächst habe ich die Struktur des Programmes durch Kenntlichmachung der einzelnen Teilschritte als Kommentare dokumentiert (z.B. 'Deklaration der Variablen, 'Gib die Bedienungsanleitung aus usw.). Darüberhinaus habe ich den Algorithmus zu "berechne die Ankunftszeit" ausführlich dokumentiert. Der Algorithmus ist nicht trivial und wer weiß, ob ich mich an ihn erinnere, wenn ich mir das Programm in 2 Jahren nochmal angucke, oder ob eventuell jemand anderes, der dies Programm erweitern will, versteht, was ich da programmiert habe.

Abbildung 7.3 zeigt einen beispielhaften Durchlauf des Programms.

Dieses Programm ist **strukturiert programmiert**. Wir haben zunächst den Algorithmus entwickelt. Dort wo der Algorithmus auf der ersten Abstraktionsebene zu allgemein war, haben wir mit Hilfe von "schrittweiser Verfeinerung" durch einen Teilalgorithmus ein Teilproblem gelöst. Der entwickelte Entwurf wurde dann in ein Programm übertragen.

Wie sinnvoll eine solche Vorgehensweise ist, zeigt sich an folgender Problemänderung:

Problemstellung 7.2 Das Raketenbeispiel (geändert)

Ein Programm soll 3 Werte einlesen, die den Startzeitpunkt einer Rakete in Stunden, Minuten und Sekunden angeben. Danach soll die Flugzeit eingelesen werden, aber ebenfalls in Stunden, Minuten und Sekunden. Danach soll aus diesen Angaben die Ankunftszeit der Rakete berechnet werden und in einem Fenster ausgegeben werden. Als erstes soll das Programm eine Bedienungsanleitung ausgeben, damit zukünftige Nutzer leichter mit dem Programm umgehen können. Wir machen zur Zeit noch eine Einschränkung: Start und Landung der Rakete finden am gleichen Tag statt.

Wenn wir dies mit unserem vorherigen Problem vergleichen, zeigt sich zunächst eine Veränderung im Einlesevorgang. Die ist allerdings relativ einfach. Darüberhinaus müssen wir die Bedienungsanleitung ändern. Das ist noch einfacher. Viel schlimmer ist, daß wir unsere Berechnungsroutine aus dem bisherigen Problem nicht verwenden können. Und das ist wirklich schade. Denn die Routine war nicht so einfach zu erstellen und wir haben sie darüberhinaus gut dokumentiert.

Also sollten wir uns überlegen, ob es nicht doch eine Möglichkeit gibt, die Berechnungsroutine weiter zu nutzen. Unsere Berechnungsroutine erwartet die Flugzeit in Sekunden, unser neues Programm erhält sie aber in Stunden, Minuten und Sekunden. Minimaler Aufwand ergibt sich also, wenn wir die Flugzeit in Stunden, Minuten und Sekunden einfach in Sekunden umrechnen.

Wir beschreiben den neuen Algorithmus (zunächst in einem Nassi-Shneidermann Diagramm:)

Pseudocode 7.4 Geändertes Raketenbeispiel

```
Gib die Bedienungsanleitung aus
Lies die Benutzereingaben ein
```

Abbildung 7.3 Die Bildschirmeingaben und die Ausgabe von Realisierung 7.6

```
Rechne Flugzeit in Sekunden um
Berechne die Ankunftszeit
Gib die Ankunftszeit in einem Fenster aus
```

Zur Realisierung müssen wir nun nur 3 Dinge tun:

"Gib die Bedienungsanleitung aus" ändern

- “Lies die Benutzereingaben ein” ändern.

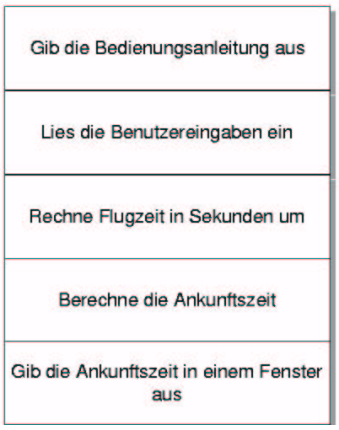

Abbildung 7.4 Nassi-Shneidermann Diagramm zur Berechnung der Raketenankunftszeit geänderte Aufgabenstellung

- “Rechne Flugzeit in Sekunden um” programmieren.

Realisierung 7.7 Änderung "Gib die Bedienungsanleitung aus"

```
MsgBox("Bitte geben Sie die Startzeit und " & Chr$(13) & _
"die Flugzeit der Rakete in Stunden, Minuten und Sekunden  an. "& Chr$(13) & _
"Auch die Ausgabe  erfolgt in Stunden,Minuten und Sekunden")
```

Realisierung 7.8 "Änderung Lies die Benutzereingaben ein"

```
stunden = InputBox("Geben Sie nun die Stunde der Abflugzeit ein!")
minuten =  InputBox("Geben Sie nun die Minuten der Abflugzeit ein!")
sekunden =  InputBox("Geben Sie nun die Sekuneden der Abflugzeit ein!")
flugzeitStunden = InputBox("Geben Sie nun die Stunden der Flugzeit ein!")
flugzeitMinuten = InputBox("Geben Sie nun die Minuten der Flugzeit ein!")
flugzeitSekunden = InputBox("Geben Sie nun die Sekunden der Flugzeit ein!")
```

Realisierung 7.9 Rechne Flugzeit in Sekunden um

```
flugzeit = flugzeitStunden * 3600 + flugzeitMinuten * 60 + flugzeitSekunden
```

Durch Zusammensetzen dieser Teile mit unseren bereits programmierten Teilen und dem Einfügen dreier neuer Variablen ensteht das fertige Programm:

Realisierung 7.10 Das Raketenprogramm zur geänderten Problemstellung

```
Option Explicit

Sub raketenBeispiel2()
'Dieses Programm berechnet die Ankunftszeit einer Rakete.
'Die Startzeit wird in Stunden, Minuten, Sekunden eingegeben.
'Die Flugzeit wird in Sekunden eingegeben.
'Dateiname: raketenBeispiel1

'Deklaration der Variablen

       dim stunden As Integer
       dim minuten As Integer
       dim sekunden As Integer
```

```
        dim flugzeitSekunden As Integer
        dim flugzeitMinuten As Integer
        dim flugzeitStunden As Integer
        dim flugzeit As Integer

'Ausgabe der Bedienungsanleitung


        MsgBox("Bitte geben Sie die Startzeit der Rakete in Stunden," & Chr$(13) & _
               "Minuten und Sekunden  an. Die Flugzeit wird in Sekunden "& Chr$(13) & _
               "eingegeben. Die Ausgabe erfolgt wieder in Stunden,Minuten und " _
               " Sekunden")

'Einlesen der Benutzereingaben

        stunden = InputBox("Geben Sie nun die Stunde der Abflugzeit ein!")
        minuten =  InputBox("Geben Sie nun die Minuten der Abflugzeit ein!")
        sekunden =  InputBox("Geben Sie nun die Sekuneden der Abflugzeit ein!")
        flugzeitStunden = InputBox("Geben Sie nun die Stunden der Flugzeit ein!")
        flugzeitMinuten = InputBox("Geben Sie nun die Minuten der Flugzeit ein!")
        flugzeitSekunden = InputBox("Geben Sie nun die Sekunden der Flugzeit ein!")

'Rechne Flugzeit in Sekunden um

        flugzeit = flugzeitStunden * 3600 + flugzeitMinuten * 60 + flugzeitSekunden

'Berechne die Ankunftszeit

        sekunden = sekunden + flugzeit

        'addiere die Minuten in den neu berechneten Sekunden
        'den Minuten der Abflugzeit hinzu.
        'Die Minuten in den neu berechneten Sekunden ist
        'das Ergebnis der Integerdivision der neu berechneten
        'Sekunden durch 60.

        minuten = minuten + sekunden \ 60

        'okay, die Minuten wurden aus den neu berechneten
        'Sekunden extrahiert. Berechne die uebrig bleibenden
        'Sekunden und weise sie den Ergebnissekunden zu.
        'Beachte: Das Ergebnis wird auch auf der
        'Variable sekunden abgelegt.
        'Dies ist das Ergebnis der Modulo-Operation
        'neu berechnete sekunden modulo 60

        sekunden = sekunden Mod 60

        'addiere die Stunden  in den neu berechneten Minuten
        'den Stunden  der Abflugzeit hinzu.
        'Die Stunden  in den neu berechneten Minuten ist
        'das Ergebnis der Integerdivision der neu berechneten
        'Minuten durch 60.

        stunden = stunden + minuten \ 60

        'okay, die Stunden wurden aus den neu berechneten
        'Minuten extrahiert. Berechne die uebrig bleibenden
        'Minuten und weise sie den Ergebnisminuten zu.
        'Beachte: Das Ergebnis wird auch auf der
        'Variable minuten abgelegt.
        'Dies ist das Ergebnis der Modulo-Operation
        'neu berechnete minuten modulo 60
```

```
        minuten = minuten Mod 60

        'die Rakete landet am geleichen Tag, also
        'muessen die Stunden in Ordnung sein,
        'wir sind durch

'Gib das Ergebnis in einem Fenster aus

        MsgBox("Die Rakete landet um: " & stunden & ":" & _
                              minuten   & ":" &sekunden)

End Sub
```

Abbildung 7.5 zeigt einen beispielhaften Durchlauf des Programms.

Abbildung 7.5 Die Bildschirmeingaben und die Ausgabe von Realisierung 7.6

8 Konstanten

Programme benötigen häufig konstante Werte (im folgenden Konstanten genannt), um Berechnungen durchführen zu können.

Eine Konstante kann explizit durch "Hinschreiben" an jeder gewünschten Stelle des Programms "definiert" werden. Dies zeigt Beispiel 8.1.

Beispiel 8.1 Konstante Werte in einem VBA-Programm

```
Option Explicit
Sub konstanten()
'Beispielprogramm fuer konstanten
'Dateiname: konstanten

       dim einkaufspreis As Double
       dim nettoVerkaufspreis As Double
       dim bruttoVerkaufspreis As Double
       einkaufspreis = InputBox ("Bitte geben Sie den Einkaufspreis ein!")
       nettoVerkaufspreis = 1.16 * einkaufspreis
       bruttoVerkaufspreis = 1.16 * nettoVerkaufspreis
       MsgBox("Netto-Verkaufspreis: " & nettoVerkaufspreis & Chr$(13) & _
                    "Brutto-Verkaufspreis: " & bruttoVerkaufspreis & Chr$(13))

End Sub
```

Abbildung 8.1 Die Bildschirmeingabe und die Ausgabe von Beispiel 8.1

Dies ist nicht immer sinnvoll:

- Die Zahl 1.16 hat geringere Aussagekraft, als das Wort Mehrwertsteuersatz.
- Die Konstante 1.16 hat in diesem Beispiel zwei Bedeutungen:
  - o Gewinnspanne
  - o Mehrwertsteuersatz

Das macht des Programm schwerer verständlich.

Ändert sich der Mehrwertsteuersatz, kann ich in Beispiel 8.1 nicht einmal mit der Funktion "Suchen und Ersetzen" den Mehrwertsteuersatz ändern. Ich würde die Gewinnspanne auch erhöhen.

Nehmen wir an, ein Programm benötigt öfter den Mehrwertsteuersatz. Eine (syntaktisch richtige) Lösung wäre, auch in diesem Fall den Mehrwertsteuersatz an jeder Stelle des Programms, wo er benötigt wird, hinzuschreiben.

Ändert der Gesetzgeber den Mehrwertsteuersatz, muß jede Stelle des Programms, wo der Mehrwertsteuersatz vorkommt, geändert werden.

VBA erlaubt es, Konstanten mit Namen zu definieren und diesen einmalig einen Wert zuzuweisen. Dies geschieht mit dem reservierten Wort `const`. Beispiel 8.2 zeigt eine abgeänderte Lösung:

Beispiel 8.2 Beispiel 8.1 mit Konstanten-Definition

```
Sub konstanten2()
'Beispielprogramm fuer konstanten
'Dateiname: konstanten2

       dim einkaufspreis As Double
       dim nettoVerkaufspreis As Double
       dim bruttoVerkaufspreis As Double
       const gewinnspanne As Double = 1.16
       const mehrwertsteuersatz As Double = 1.16

       einkaufspreis = InputBox ("Bitte geben Sie den Einkaufspreis ein!")
       nettoVerkaufspreis = 1.16 * einkaufspreis
       bruttoVerkaufspreis = 1.16 * nettoVerkaufspreis
       MsgBox("Netto-Verkaufspreis: " & nettoVerkaufspreis & Chr$(13) & _
                     "Brutto-Verkaufspreis: " & bruttoVerkaufspreis & Chr$(13))

End Sub
```

Änderungen des Mehrwertsteuersatzes implementiere ich in meinem Programm, indem ich einfach die Zeile:

```
       const mehrwertsteuersatz As Double = 1.16
```

ändere.

Konstanten unterscheiden sich von Variablen dadurch, daß sich ihr Wert während des Programmlaufs nicht ändern kann (Konstanten sind halt konstant).

Konstanten dürfen daher nie auf der linken Seite einer Zuweisung stehen.

Dies zeigt Beispiel 8.3 und Abbildung 8.2.

Beispiel 8.3 Konstante auf der linken Seite einer Zuweisung **Fehler**

```
Sub konstanten3()
'Beispielprogramm fuer konstanten
'Dateiname: konstanten3

       dim einkaufspreis As Double
       dim nettoVerkaufspreis As Double
       dim bruttoVerkaufspreis As Double
       const gewinnspanne As Double = 1.16
       const mehrwertsteuersatz As Double = 1.16

       einkaufspreis = InputBox ("Bitte geben Sie den Einkaufspreis ein!")
       nettoVerkaufspreis = gewinnspanne * einkaufspreis

'Berechnung fuer Deutschland
       bruttoVerkaufspreis = mehrwertsteuersatz * nettoVerkaufspreis
       MsgBox("Netto-Verkaufspreis: " & nettoVerkaufspreis & Chr$(13) & _
                     "Brutto-Verkaufspreis: " & bruttoVerkaufspreis & Chr$(13))
```

```
'Berechnung fuer Daenemark
       mehrwertsteuersatz = 1.25 'geraten
       bruttoVerkaufspreis = mehrwertsteuersatz * nettoVerkaufspreis
       MsgBox("Netto-Verkaufspreis: " & nettoVerkaufspreis & Chr$(13) & _
                     "Brutto-Verkaufspreis: " & bruttoVerkaufspreis & Chr$(13))

End Sub
```

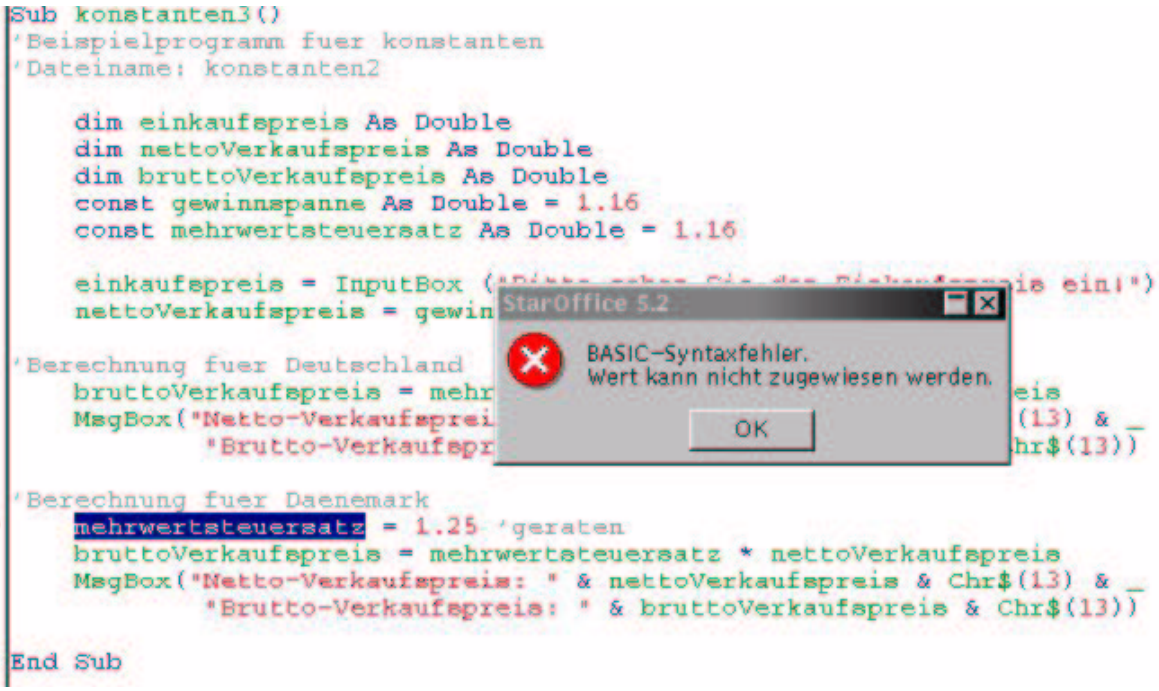

Abbildung 8.2 Fehlermeldung bei dem Versuch eine Konstante auf der linken Seite einer Zuweisung zu benutzen

Für Daten, die ihren Wert ändern können, sind nämlich, wie Sie in Kapitel 5 gelernt haben, Variablen zuständig. Beispiel 8.3 müßte also richtigerweise folgendermaßen kodiert werden:

Beispiel 8.4 Berichtigung von Beispiel 8.3

```
Option Explicit
Sub konstanten4()
'Beispielprogramm fuer konstanten
'Dateiname: konstanten4

       dim einkaufspreis As Double
       dim nettoVerkaufspreis As Double
       dim bruttoVerkaufspreis As Double
       const gewinnspanne As Double = 1.16
       dim mehrwertsteuersatz As Double

       einkaufspreis = InputBox ("Bitte geben Sie den Einkaufspreis ein!")
       nettoVerkaufspreis = gewinnspanne * einkaufspreis

'Berechnung fuer Deutschland
       mehrwertsteuersatz = 1.16
       bruttoVerkaufspreis = mehrwertsteuersatz * nettoVerkaufspreis
       MsgBox("Netto-Verkaufspreis Deutschland: " & nettoVerkaufspreis & Chr$(13) & _
                     "Brutto-Verkaufspreis: " & bruttoVerkaufspreis & Chr$(13))
```

```
'Berechnung fuer Daenemark
       mehrwertsteuersatz = 1.25 'geraten
       bruttoVerkaufspreis = mehrwertsteuersatz * nettoVerkaufspreis
       MsgBox("Netto-Verkaufspreis D‰nemark: " & nettoVerkaufspreis & Chr$(13) & _
                     "Brutto-Verkaufspreis: " & bruttoVerkaufspreis & Chr$(13))

End Sub
```

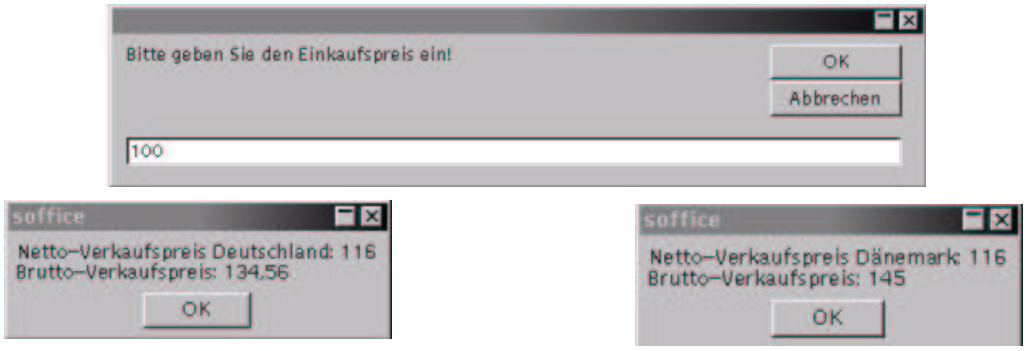

Abbildung 8.3 Die Bildschirmeingabe und die Ausgabe von Beispiel 8.4

Konstanten erhalten ihren Wert durch die `const`-Anweisung, danach ändert der Wert sich nicht mehr. Für den Konstantennamen gelten die gleichen Regeln wie für Variablennamen. Zusätzlich: Konstantennamen dürfen keine Variablennamen sein.

9 Steuerungsstrukturen (Kontrollstrukturen)

9.1 Konditionalanweisungen

9.1.1 Die if- Anweisung (Ein- und Zweiseitige Auswahl)

Unsere bisherigen Programmierkenntnisse lassen nur die Realisierung linearer Programmverläufe zu. Dies bedeutet, die Anweisungen in unseren Programmen werden von oben nach unten in genau vorgegebener Reihenfolge abgearbeitet. Wir können keine Anweisungen nur unter bestimmten Bedingungen durchführen lassen oder Programmblöcke häufiger ablaufen lassen. Abbildung 9.1 zeigt die Problematik.

```
Sub addition2()

' Programm addiert die zwei einzugebende Zahlen
' Dateiname: addition

       dim ersterSummand As Integer
       dim zweiterSummand As Integer
       dim summe As Integer

       ersterSummand = InputBox("Bitte geben Sie den ersten Summanden
       ein")
       zweiterSummand =  InputBox("Bitte geben Sie den zweiten Summanden
       ein")

       summe = ersterSummand + zweiterSummand

       MsgBox ("Summe: " & summe)
End Sub
```

Abbildung 9.1 Linearer Programmverlauf

Dies ist aber ein echter Nachteil, wie die nachfolgenden Beispiele veranschaulichen:

- Wir haben Programme geschrieben, die addieren, multiplizieren und subtrahieren können. Aber für jede Rechenart haben wir ein eigenes Programm, das ist schlecht. Viel schöner wäre es, wenn diese Dinge in einem Programm realisiert wären und der Benutzer entscheiden könnte, was er tun wollte.
- Programmieren wir ein Divisionsprogramm (wie unser Additionsprogramm) können wir Laufzeitfehler nicht vermeiden, denn wenn der Benutzer 0 als Nenner angibt, dann bricht das Programm mit einem Laufzeitfehler ab (vgl. Beispiel 9.1 und Abbildung 9.2). Besser wäre, den Benutzer auf seinen Eingabefehler aufmerksam zu machen.

Beispiel 9.1 Divsionsprogramm mit Laufzeitfehler

```
Option Explicit
Sub divisionMitLaufzeitfehler()

' Programm dividiert die erste einzugebende Zahl
' durch die zweite einzugebende Zahl
```

```
' Dateiname: divisionMitLaufzeitfehler

        dim zaehler As Double
        dim nenner As Double
        dim quotient As Double

        zaehler =InputBox("Bitte geben Sie den Zähler ein!")
        nenner = InputBox("Bitte geben Sie den Nenner ein!")

        quotient = zaehler / nenner

        MsgBox ("Quotient " & quotient)

End Sub
```

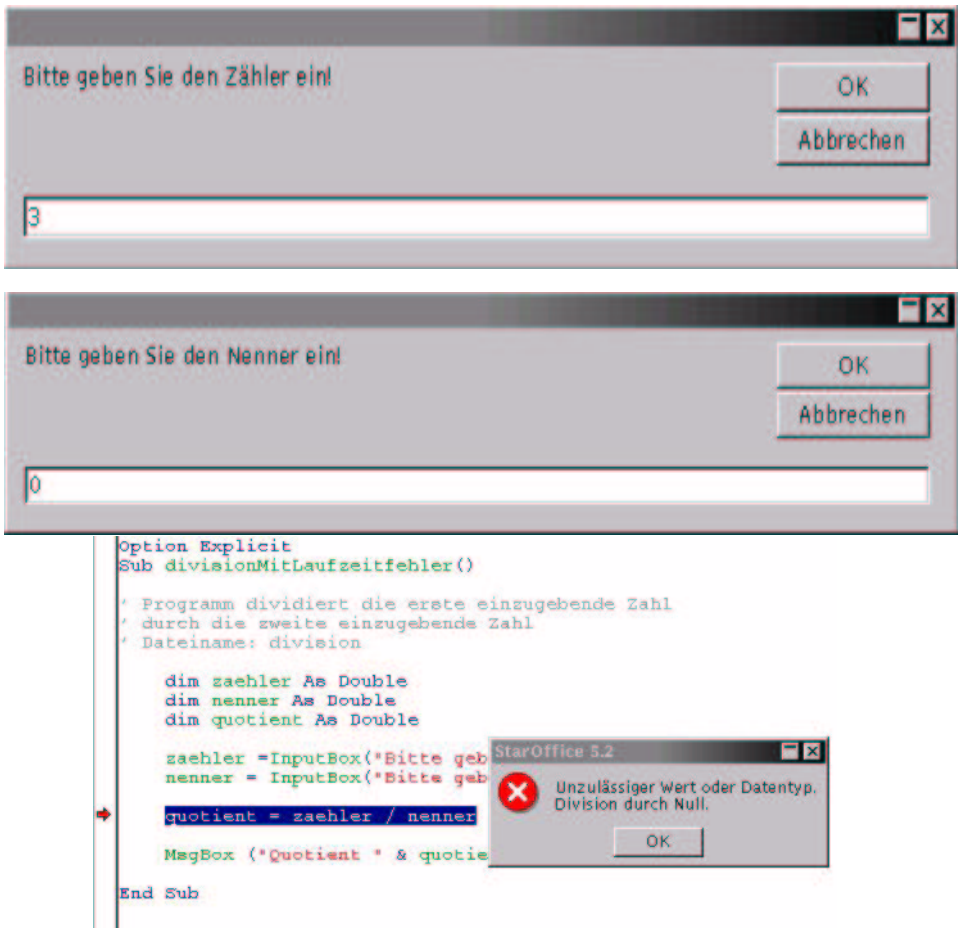

Abbildung 9.2 Die Bildschirmausgabe von Beispiel 9.1 bei Eingabe von 0

- Stellen Sie sich vor, wir wollten ein Programm schreiben, welches Provisionen für Vermittler unserer Produkte berechnet. Die Provisionen sind abhängig vom geplanten Umsatz mit uns oder vom letztjährigen Umsatz oder vom bisher mit unserem Unternehmen erzielten Umsatz. Eine solche Anwendung können wir mit unserem bisherigen Wissensstand nicht realisieren.

Lösung beider Probleme ist die `if`-Anweisung. Die `if`-Anweisung hat die Form:

```
if logischer Ausdruck Then
       anweisung1
```

```
                :
                :
        anweisungN
else
        anweisung1nachElse
                :
                :
        anweisungNnachElse
End if
```

"Logischer Ausdruck" ist ein Ausdruck, der ein "logisches Ergebnis" erzeugt.

Ein logisches Ergebnis ist:

`true` (wahr)

`false` (falsch)

Logische Ausdrücke sind daher im wesentlichen:

- Vergleiche (vgl. Kapitel 6.2). Typische Vergleiche sind:

  ```
  if nenner = 0


  if (x > 3) or (y <= 4)
  ```

  Hierzu benötigt man die Vergleichsoperatoren aus Kapitel 6.2.2 und die logischen Operatoren aus Kapitel 6.2.3.

- bool'sche Variablen (vgl. Kapitel 5.1).

Ergibt der logische Ausdruck den Wert `true` werden die Anweisungen im `then`-Teil der `if`-Anweisung , anderenfalls (der logische Ausdruck ergibt `false`) die Anweisungen im `else`-Teil ausgeführt.

Der `else`-Teil ist optional. Gibt es den `else`-Teil nicht, ist dies gleichbedeutend mit: Ansonsten tue nichts.

Ist ein `else`-Teil vorhanden, sprechen wir von zweiseitiger Auswahl, ansonsten von einseitiger Auswahl.

`if`-Anweisungen können beliebig tief geschachtelt werden.

Wir betrachten nun einige Beispiele.

Problemstellung 9.1 Division

Schreiben Sie ein Programm, das 2 reelle Zahlen einliest, die erste eingelesene Zahl durch die zweite dividiert und das Ergebnis ausgibt. Wird als zweite Zahl (Nenner) Null eingegeben, so soll die Fehlermeldung "Versuch durch 0 zu teilen" ausgegeben werden.

Pseudocode 9.1 Pseudocode zu Problemstellung 9.1

```
Lies die Benutzereingaben ein
Berechne Quotienten
Gib das Ergebnis aus
```

Nassi-Shneidermann-Diagramm zu Beispiel 2:

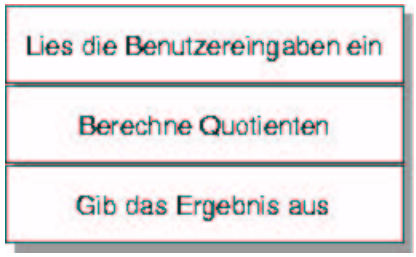

Abbildung 9.3 Nassi-Shneiderman-Diagramm zu Problemstellung 9.1

"Berechne Quotienten" müssen wir noch näher spezifizieren. Wir haben hier zunächst einmal das Problem, daß wir feststellen müssen, wann der Nenner Null ist. Das können wir aber mit einer `if`-Konstruktion in den Griff bekommen. Zum Zweiten wird nach Pseudocode 9.1 "Gib das Ergebnis aus" nach "Berechne Quotienten" durchgeführt. Wurde aber Null als Nenner eingegeben, gibt es kein Ergebnis. Demzufolge können wir auch keins ausgeben.

Eine Lösung wird dadurch möglich, daß wir ein VBA-Programm zu jeder Zeit beenden (bzw. verlassen) können. Das Kommando hierzu ist `Exit Sub`. Wenn die VBA-Umgebung auf `Exit Sub` trifft, wir das laufende VBA-Programm sofort beendet.

Wir formulieren nun den Pseudocode zu "Berechne Quotienten":

Pseudocode 9.2 Berechne Quotienten

```
if nenner ist nicht null then
       berechne den Quotienten
else
       Gib Fehlermeldung aus
       verlasse das Programm
End if
```

Abbildung 9.4 zeigt das Nassi-Shneiderman-Diagramm zu "Berechne den Quotienten". Die grafische Darstellung der `if`-Anweisung in einem Struktogramm ist (wie ich

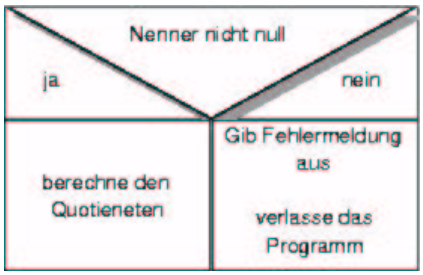

Abbildung 9.4 Nassi-Shneiderman-Diagramm zu "Berechne den Quotienten"

hoffe) selbsterklärend.

Abbildung 9.3 und Abbildung 9.4 oder Pseudocode 9.1 und Pseudocode 9.2 können wir nun, wie gehabt, leicht in ein VBA-Programm umsetzen.

Realisierung 9.1 Divisionsbeispiel zum Ersten

```
Sub division()

' Programm dividiert die erste einzugebende Zahl
' durch die zweite einzugebende Zahl
' Dateiname: division

       dim zaehler As Double
       dim nenner As Double
       dim quotient As Double

' Lies die Benutzereingaben ein

       zaehler =InputBox("Bitte geben Sie den Zähler ein!")
       nenner = InputBox("Bitte geben Sie den Nenner ein!")

' Berechne Quotienten

       if (nenner <> 0) Then
              quotient = zaehler / nenner
       else
              MsgBox ("Der Nenner ist 0!")
              Exit Sub
       End if

' Gib das Ergebnis aus

       MsgBox ("Quotient " & quotient)

End Sub
```

Wir sehen, daß wir unseren Pseudocode auch hier nach den Variablendeklarationen eins zu eins in ein VBA-Programm umsetzen konnten. Beachten Sie, daß die Anweisung

```
       MsgBox ("Quotient " & quotient)
```

nicht durchgeführt wird, wenn der `else`-Teil der `if`-Anweisung erreicht wird. Das VBA-Programm wird dort durch die Anweisung

```
       Exit Sub
```

verlassen.

Für Problemstellung 9.1 gibt es eine weitere Lösung unter Nutzung einer if-Anweisung ohne else-Teil. Betrachten wir dazu folgenden Pseudocode:

Pseudocode 9.3 Berechne Quotienten (Zweite Version)

```
       if nenner ist null then
              Gib Fehlermeldung aus
              verlasse das Programm
       End if
       berechne den Quotienten
```

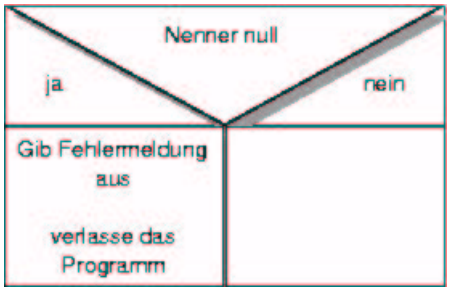

Abbildung 9.5 Nassi-Shneiderman-Diagramm zu "Berechne den Quotienten" (Zweite Version)

Abbildung 9.5 zeigt, daß ein fehlender Zweig der `if`-Anweisung in Struktogrammen durch ein leeres Rechteck dargestellt wird.

Auch in unserem neuen Algorithmus besteht der Trick darin, daß das VBA-Programm durch die Anweisung `Exit Sub` verlassen wird.

Realisierung 9.2 zeigt die Realisierung des zweiten Algorithmus:

Realisierung 9.2 Divisionsbeispiel zum Zweiten

```
Sub division2()

' Programm dividiert die erste einzugebende Zahl
' durch die zweite einzugebende Zahl
' Dateiname: division

       dim zaehler As Double
       dim nenner As Double
       dim quotient As Double

' Lies die Benutzereingaben ein

       zaehler =InputBox("Bitte geben Sie den Zähler ein!")
       nenner = InputBox("Bitte geben Sie den Nenner ein!")

' Berechne Quotienten

       if (nenner = 0) Then
              MsgBox ("Der Nenner ist 0!")
              Exit Sub
       End if

       quotient = zaehler / nenner

' Gib das Ergebnis aus

       MsgBox ("Quotient " & quotient)

End Sub
```

Abbildung 9.6 zeigt zwei beispielhafte Abläufe von Realisierung 9.1 bzw. Realisierung 9.2 (sind ja nun Programme, die identische Ein- und Ausgaben erzeugen).

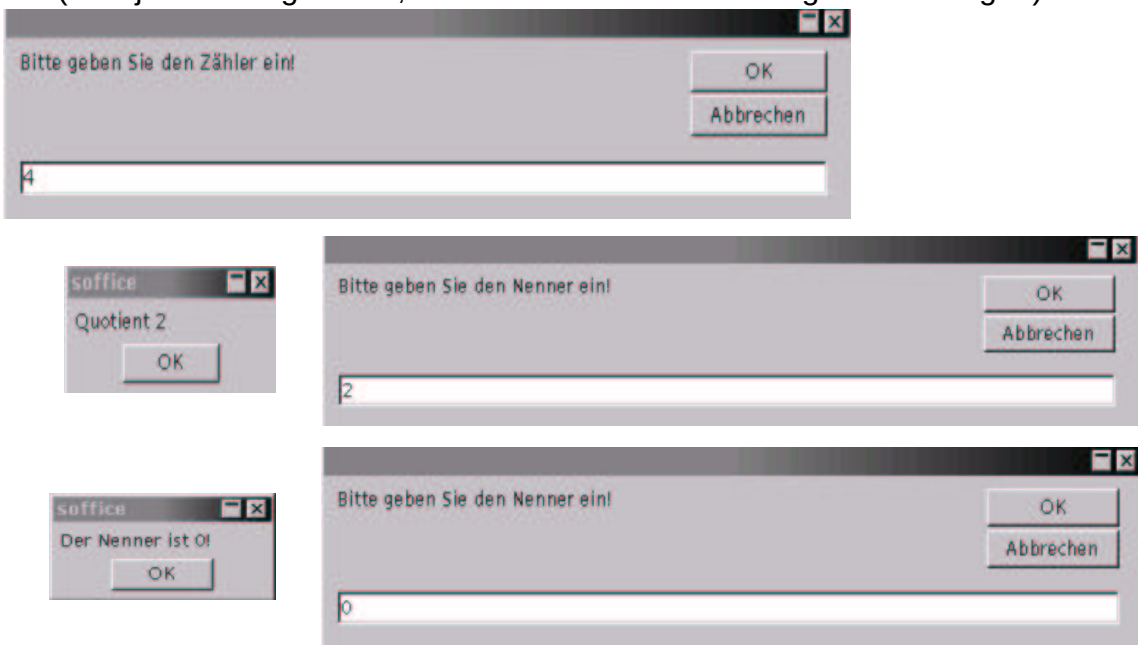

Abbildung 9.6 Realisierung 9.1 bzw. Realisierung 9.2 mit korrektem Nenner und mit Null als Nenner

Kommen wir nun zu einem zweiten Beispiel:

Problemstellung 9.2 Provision berechnen zum Ersten

Ein VBA-Programm soll die zusätzliche Provision für einen Verkauf eines Vermittlers anhand des geplanten Jahresumsatzes berechnen. Liegt der Jahresumsatz höher als DM 100.000,--, wird eine Zusatzprovision von 10 % gewährt. Darunter gibt es keine Zusatzprovision. Das Programm soll zunächst in einem Fenster eine kurze Bedienungsanleitung ausgeben, dann werden Umsatz und jetziger Verkaufsbetrag eingegeben. Danach wird die auszugebende Provision errechnet und ausgegeben.

Auch hier ermitteln wir erst den Algorithmus und stellen ihn in Pseudocode dar. Auf eine Struktogramm-Darstellung verzichte ich.

Pseudocode 9.4 Provision berechnen

```
Gib Programmbeschreibung aus
Lies die Benutzereingaben ein
bestimme Provision in Prozent
berechne auszuzahlenden betrag Formel:
(verkaufsbetrag*provision in prozent)/100
Gib das Ergebnis aus
```

Diesen Pseudocode brauchen wir bei unserem jetzigen Kenntnisstand nicht weiter zu detaillieren. "bestimme Provision in Prozent" ist eine einfache `if`-Anweisung. Die Formel für die Berechnung des auszuzahlenden Betrags haben wir in den Pseudocode aufgenommen. Wir können Sie direkt in das zu schreibende Programm übernehmen.

Realisierung 9.3 Provision berechnen

```
Sub provision()
```

```
' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provision

      dim umsatz As Double
      dim verkaufsbetrag As Double
      dim provisionInProzent As Double
      dim auszuzahlendeProvision As Double

      ' Umsatzgrenzen sind DM-Betraege

      const umsatzGrenze As Double = 100000

      ' Provisionen in Prozent

      const provisionUmsatzGrenze As Double = 10

' Gib Programmbeschreibung aus

      MsgBox("Geben Sie Umsatz und Verkaufsbetrag ein!" _
                    & chr$(13) & "Das Programm berechnet die" _
                    & " Provision des Vermittlers!")

' Lies die Benutzereingaben ein

      umsatz = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
      verkaufsbetrag = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")


' bestimme Provision in Prozent
      MsgBox(provisionUmsatzGrenze)
      if umsatz >= umsatzGrenze Then
            provisionInProzent = provisionUmsatzGrenze
      else
            provisionInProzent = 0
      End if

' Berechne auszuzahlenden Betrag

      auszuzahlendeProvision = (verkaufsbetrag * provisionInProzent)/100

' Gib das Ergebnis aus

      MsgBox ("Die Provision für dieses Geschäft ist: " _
                                 & auszuzahlendeProvision & " DM")

End Sub
```

Wir haben die Umsatzgrenze, ab der Provision gezahlt wird, und den Provisionssatz als Konstanten in das Programm aufgenommen. Dies erleichtert spätere Änderungen.

Abbildung 9.7 zeigt einen bespielhaften Programmablauf:

9.1.2 Die if-elseif-Anweisung (Mehrseitige Auswahl)

Häufig reicht die zweiseitige Auswahl nicht aus. Betrachten wir Problemstellung 9.2. Hier gibt es nur einen Provisionssatz. Dies ist nicht sehr realistisch. Normalerweise steigen die Provisionssätze mit steigendem Umsatz. Solche Anforderungen lassen

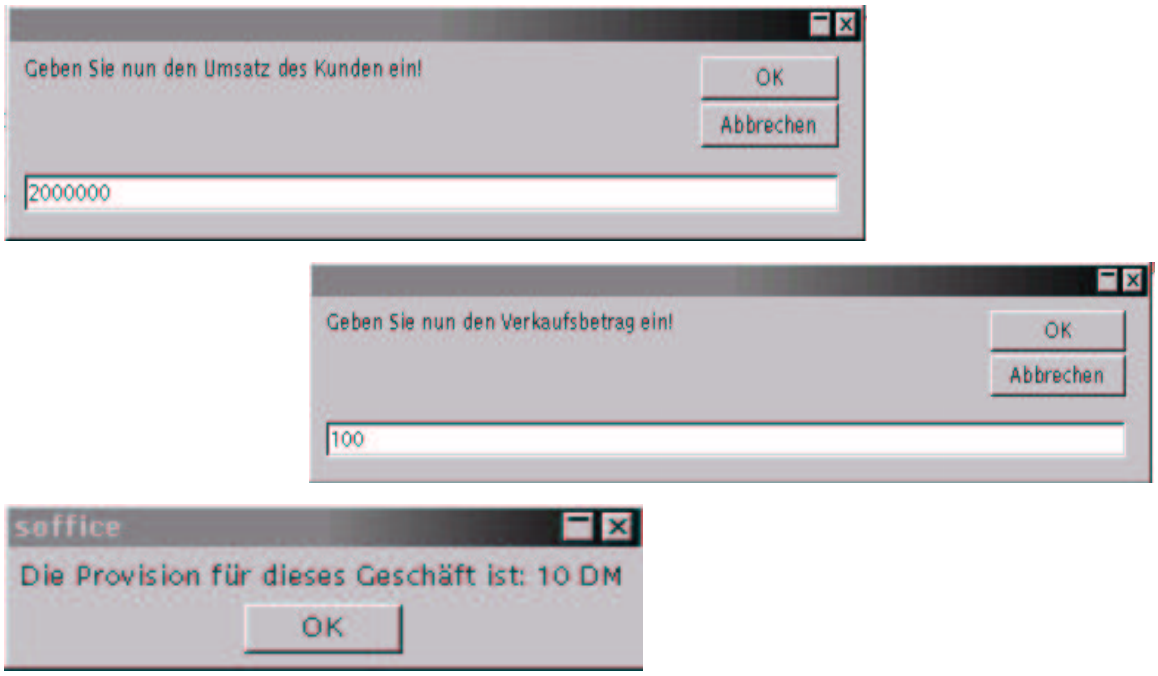

Abbildung 9.7 Bildschirmein- und -ausgabe von Realisierung 9.3

sich durch geschachtelte if-Anweisungen darstellen, der Code wird aber schnell unübersichtlich. Betrachten wir dazu:

Problemstellung 9.3 Provisionsberechnung zum Zweiten

Ein VBA-Programm soll die zusätzliche Provision für einen Verkauf eines Vermittlers anhand des geplanten Jahresumsatzes berechnen. Die Provision soll nach folgender Tabelle gewährt werden:

Umsatz	Provision in Prozent
bis 100000	0
100000 bis 500000	5
500000 bis 1000000	10
über 1000000	20

Das Programm soll zunächst in einem Fenster eine kurze Bedienungsanleitung ausgeben, dann werden Umsatz und jetziger Verkaufsbetrag eingegeben. Liegt der Verkaufsbetrag, für den die Provision gerade berechnet werden soll, über dem geplanten Umsatz für das ganze Jahr, soll das Programm eine Fehleingabe vermuten und eine Fehlermeldung ausgeben. Ansonsten wird der Provisionbetrag errechnet und ausgegeben.

Dies ist der typische Fall einer mehrwertigen[13] Entscheidungssituation. Dies ist zwar mit den Mitteln aus Kapitel 9.1.1 programmierbar, erfordert dann aber geschachtelte

13.Wieviel Entscheidungsmöglichkeiten es hier gibt, sollen Sie sich selbst überlegen :-).

`if`-Strukturen, die schnell unübersichtlich werden. Dazu betrachten wir den Pseudocode der Lösung:

Pseudocode 9.5 Provision berechnen (2ter Versuch)

```
Gib Programmbeschreibung aus
Lies die Benutzereingaben ein
überprüfe Benutzereingaben
bestimme Provision in Prozent
berechne auszuzahlenden betrag Formel:
(verkaufsbetrag*provision in prozent)/100
Gib das Ergebnis aus
```

"bestimme Provision in Prozent" ist in unserem Fall jetzt nicht ganz so einfach, daher schreiben wir Pseudocode für diesen Ablauf:

Pseudocode 9.6 Bestimme Provision in Prozent

```
if umsatz < 100000 then
        setze provision auf 0 Prozent
else
        if umsatz zwischen 100000 und 500000 then
                setze provision auf 5 Prozent
        else
                if umsatz zwischen 500000 und 1000000 then
                        setze provision auf 10 Prozent
                else
                        setze provision auf 20 Prozent
                end if
        end if
end if
```

Dies könnten wir sofort in VBA umsetzen[14], wird aber schnell unübersichtlich. Es ist nicht sofort ersichtlich, welches `else` zu welchem `if` gehört. Je mehr Fälle auftreten, desto unübersichtlicher wird die Situation. Für solche Situationen stellt VBA zwei Programmierkonstrukte zur Verfügung. In diesem Kapitel behandeln wir die `if-elseif`-Anweisung. Diese Anweisung hat die Form:

```
        if logischerAusdruck1 Then
                anweisung1
                        :
                        :
                anweisungN
        elseif logischerAusdruck2 Then
                anweisung1
                        :
                        :
                anweisungN
        ...
        elseif logischerAusdruckN Then
                anweisung1
                        :
                        :
                anweisungN
        elseif
                anweisung1nachElse
                        :
                        :
```

14. Übungsaufgabe für Sie!

```
        anweisungNnachElse
End if
```

Hierbei passiert Folgendes:

Trifft VBA auf obige Anweisung, wird zunächst "logischer Ausdruck 1" ausgewertet. Wird *true* ermittelt, werden die Anweisungen zwischen `then` und dem nächsten `elseif` oder `else` durchgeführt. Danach wird mit der nächsten Anweisungen hinter dem gesamten `if-elseif`-Block fortgesetzt. Wird *false* ermittelt, prüft VBA den logischen Ausdruck des nächsten `elseif`.

Dies wird so lange fortgesetzt,

- bis ein logischer Ausdruck *true* ergibt: In diesem werden die Anweisungen zwischen diesem logischen Ausdruck und dem nächsten `elseif` durchgeführt. Danach setzt VBA mit der nächsten Anweisungen hinter dem gesamten `if-elseif`-Block fort.
- `else` erreicht wird: In diesem Fall wird der `else`-Teil durchgeführt. Danach setzt VBA mit der nächsten Anweisungen hinter dem `if-elseif`-Block fort.
- das Ende des `if-elseif`-Blocks erreicht wird: `else` ist wie in Kapitel 9.1.1 optional.

Auf jeden Fall ist sichergestellt, daß nur ein Zweig der Konstruktion durchlaufen wird.

Mit diesem Wissen können wir Pseudocode 9.6 neu formulieren:

Pseudocode 9.7 Bestimme Provision in Prozent (mit `if-elseif`)

```
if umsatz >= 1000000 then
        setze provision auf 20 Prozent
elseif umsatz >= 500000
        setze provision auf 10 Prozent
elseif umsatz >= 100000
        setze provision auf 5 Prozent
else
        setze provision auf 0 Prozent
end if
```

Beachten Sie die Reihenfolge der Bedingungen. Wir nutzen hier aus, daß immer nur ein Zweig der gesamten Konstruktion durchlaufen wird. Denn wenn der Umsatz größer 1000000 ist, trifft die erste Bedingung zu, und die Anweisungen zwischen dieser Bedingung und dem nächsten elseif werden durchgeführt. Danach wird der gesamte `if-elseif-Block` verlassen. Dies bedeutet, daß obwohl der Umsatz in diesem Fall selbstverständlich auch größer als 500000 ist, die zugehörigen Anweisungen nicht durchlaufen werden. Überlegen Sie sich, warum man die Bedingungen nicht in einer anderen Reihenfolge hätte formulieren dürfen.

Abbildung 9.8 zeigt den Algorithmus als Struktogramm.

Auch diese Darstellung sollte selbsterkärend sein.

Wir schreiben nun noch den Pseudocode zu "überprüfe Benutzereingaben".

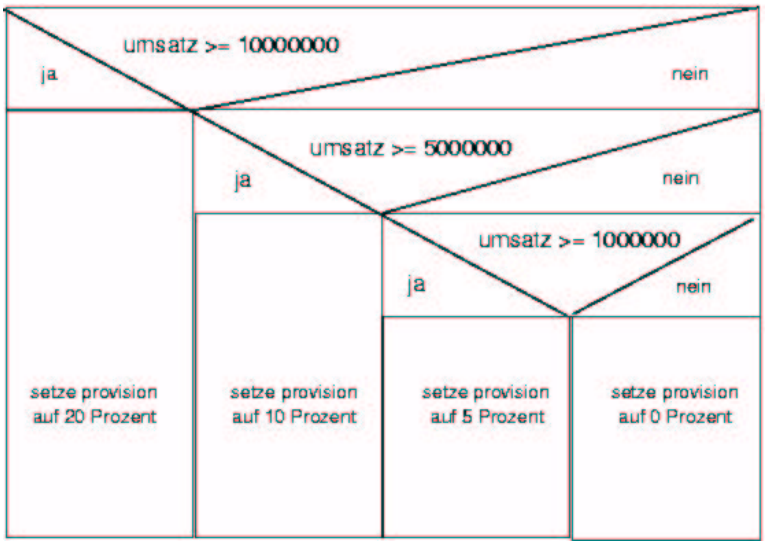

Abbildung 9.8 Struktogramm zu "Bestimme Provision in Prozent" (mit if-elseif)

Pseudocode 9.8 Überprüfe Benutzereingaben

```
if verkaufsbetrag größer umsatz Then
       gib Fehlermeldung aus
       verlasse das Programm
end if
```

Nun können wir die Lösung programmieren:

Realisierung 9.4 Problemstellung 9.3 gelöst

```
Sub provision2()

' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provision2

       dim umsatz As Double
       dim verkaufsbetrag As Double
       dim provisionInProzent As Double
       dim auszuzahlendeProvision As Double

       ' Umsatzgrenzen sind DM-Betraege

       const umsatzGrenze1 As Double = 100000
       const umsatzGrenze2 As Double = 500000
       const umsatzGrenze3 As Double = 1000000

       ' Provisionen in Prozent

       const provisionUmsatzGrenze1 As Double = 5
       const provisionUmsatzGrenze2 As Double = 10
       const provisionUmsatzGrenze3 As Double = 20

' Gib Programmbeschreibung aus

       MsgBox("Geben Sie Umsatz und Verkaufsbetrag ein!" _
```

```
                    & chr$(13) & "Das Programm berechnet die" _
                    & " Provision des Vermittlers!")

' Lies die Benutzereingaben ein

      umsatz = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
      verkaufsbetrag = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")

' ueberpruefe Benutzereingaben

      if verkaufsbetrag > umsatz Then
            MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
            Exit Sub
      End if

' bestimme Provision

      if umsatz >= umsatzGrenze3 Then
            provisionInProzent = provisionUmsatzGrenze3
      elseIf umsatz >= umsatzGrenze2 Then
            provisionInProzent = provisionUmsatzGrenze2
      elseIf umsatz >= umsatzGrenze1 Then
            provisionInProzent = provisionUmsatzGrenze1
      else
            provision = 0
      End if

' Berechne dir Provision

      auszuzahlendeProvision = (verkaufsbetrag * provisionInProzent)/100

' Gib das Ergebnis aus

      MsgBox ("Die Provision für dieses Geschäft ist: " _
                              & auszuzahlendeProvision & " DM")

End Sub
```

Auch hier wurden die Provisionssätze und Umsatzgrenzen als Konstanten in das Programm aufgenommen. Auch hier tun wir das, um das Programm bei neuen Anforderungen leichter ändern zu können. Abbildung 9.9 zeigt einen beispielhaften Ablauf des Programms:

9.1.3 Überprüfung der Benutzereingaben und Konvertierungsfunktionen

Eine Schwachstelle unserer bisherigen Programme sind die Benutzereingaben. Alle bisherigen Provisionsprogramme und auch die Rechenprogramme erwarten Zahlen als Eingaben. In keinem Fall wird aber überprüft, ob der Benutzer wirklich eine Zahl eingibt. Dies kann zu unliebsamen Ergebnissen führen. Wenn unsere Benutzer statt Zahlen Buchstaben eingeben, führt dies

- in StarCalc dazu, daß die Variablen, auf die eingelesen wird, mit Null besetzt werden. Dadurch entstehen Ergebnisse, die so nicht gemeint sein können.
- in Excel zu einem Programmabbruch mit der Fehlermeldung, daß ein nicht kompatibler Datentyp eingelesen wurde.

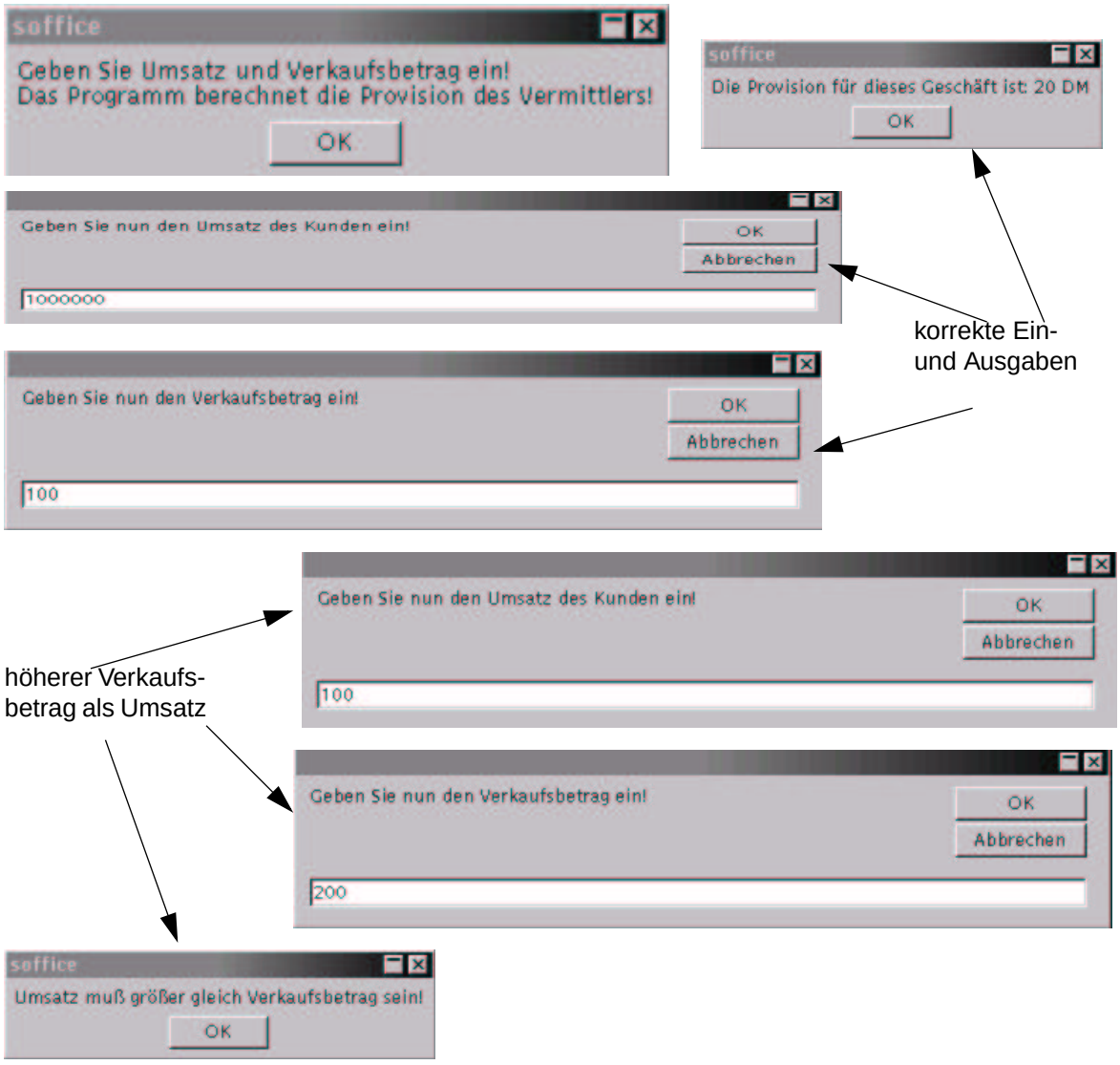

Abbildung 9.9 Realisierung 9.4 mit korrekten Eingaben und mit höherem Verkaufsbetrag als Umsatz

Beide Varianten sind so nicht erstrebenswert (wobei die Excel-Variante deutlich besser ist). Besser wäre, wir könnten prüfen, ob die Benutzer-Eingaben numerisch sind oder nicht. Bei nicht numerischen Benutzer-Eingaben könnten wir den Benutzer auf seinen Fehler aufmerksam machen und das Programm dann beenden[15]. Ich werde nun zwei Möglichkeiten vorstellen, wie dies realisiert werden kann.

Realisierung 9.5 Das Provisionsprogramm mit Eingabekontrolle: Nutzung von Variant-Variablen

```
Sub provision3()

' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provision3
```

15.Später werden wir noch Techniken lernen, wie Sie den Benutzer zu einer neuen Eingabe auffordern können.

```
	dim umsatz
	dim verkaufsbetrag
	dim provisionInProzent As Double
	dim auszuzahlendeProvision As Double

	' Umsatzgrenzen sind DM-Betraege

	const umsatzGrenze1 As Double = 100000
	const umsatzGrenze2 As Double = 500000
	const umsatzGrenze3 As Double = 1000000

	' Provisionen in Prozent

	const provisionUmsatzGrenze1 As Double = 5
	const provisionUmsatzGrenze2 As Double = 10
	const provisionUmsatzGrenze3 As Double = 20

' Gib Programmbeschreibung aus

	MsgBox("Geben Sie Umsatz und Verkaufsbetrag ein!" _
		& chr$(13) & "Das Programm berechnet die" _
		& " Provision des Vermittlers!")

' Lies die Benutzereingaben ein

	umsatz = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
	verkaufsbetrag = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")

' ueberpruefe Benutzereingaben

	if Not IsNumeric (umsatz) or Not IsNumeric (verkaufsbetrag) Then
		MsgBox ("Umsatz und Verkaufsbetrag müssen Zahlen sein!")
		Exit Sub
	End if

	'sicherstellen, dass Benutzereingaben als Double
	'interpretiert werden

	umsatz = CDbl(umsatz)
	verkaufsbetrag = CDbl(verkaufsbetrag)

	if verkaufsbetrag > umsatz Then
		MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
		Exit Sub
	End if

' bestimme Provision

	if umsatz >= umsatzGrenze3 Then
		provisionInProzent = provisionUmsatzGrenze3
	elseIf umsatz >= umsatzGrenze2 Then
		provisionInProzent = provisionUmsatzGrenze2
	elseIf umsatz >= umsatzGrenze1 Then
		provisionInProzent = provisionUmsatzGrenze1
	else
		provisionInProzent = 0
	End if

' Berechne dir Provision

	auszuzahlendeProvision = (verkaufsbetrag * provisionInProzent)/100

' Gib das Ergebnis aus
```

```
        MsgBox ("Die Provision für dieses Geschäft ist: " _
                                    & auszuzahlendeProvision & " DM")

End Sub
```

In Abweichung zu Realisierung 9.4 werden durch die Zeilen

```
        dim umsatz
        dim verkaufsbetrag
```

`umsatz` und `verkaufsbetrag` als Variant-Variablen vereinbart. Dadurch können die Einlesevorgänge

```
        umsatz = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
        verkaufsbetrag = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")
```

bedenkenlos durchgeführt werden. Egal was der Benutzer eingibt, VBA wird es als String interpretieren (vgl. Beispiel 5.2). Der nächste Schritt ist, zu prüfen, ob die Eingaben Zahlen waren oder nicht. Dafür stellt VBA die Funktion `IsNumeric` zur Verfügung. `IsNumeric` gibt `true` zurück, wenn die der Funktion übergebene Variable eine Zahl ist, `false` sonst.

Die logische Bedingung der folgenden `if`-Anweisung ergibt also `true`, wenn eine der beiden Eingaben keine Zahl war (vor `IsNumeric` steht `Not` und beide Bedingungen sind durch `or` verbunden).

```
        if Not IsNumeric (umsatz) or Not IsNumeric (verkaufsBetrag) Then
                MsgBox ("Umsatz und Verkaufsbetrag müssen Zahlen sein!")
                Exit Sub
        End if
```

In diesem Fall wird eine Fehlermeldung ausgegeben und das VBA-Programm beendet. Andernfalls (die vorstehende if-Anweisung ergab `false`) müssen wir, um Fehler wie in Beispiel 5.2 zu vermeiden, jetzt irgendwie den Variablentyp ändern. Dazu gibt es in VBA Konvertierungsfunktionen. Alle VBA-Konvertierungsfunktionen beginnen mit C und enden mit einer Drei-Buchstaben-Abkürzung des Datentyps in den umgewandelt werden soll. Die Konvertierungsfunktionen für die Umwandlung in `Double` heißt `CDbl`. Durch die Zeilen

```
        umsatz = CDbl(umsatz)
        verkaufsbetrag = CDbl(verkaufsbetrag)
```

werden also die Eingaben von Variant in `Double` gewandelt. Dies wird auf jeden Fall funktionieren, da wir durch die Überprüfung wissen, daß beide Eingaben Zahlen sind (sonst wäre das Programm nicht bis hierhin gekommen).

Der Rest des Programms entspricht Realisierung 9.4. Abbildung 9.10 zeigt eine Fehlerausgabe des Programms.

Eine zweite Möglichkeit, dies Problem zu lösen, ist, die Eingaben auf einen `String` einzulesen. Dann testen wir, ob der `String` eine Zahl ist. Wenn dies nicht der Fall ist, teilen wir dem Nutzer dies mit und beenden das Programm, ansonsten wandeln wir den `String` um und weisen das Ergebnis der Umwandlung unseren `Double`-Variablen zu. Realisierung 9.6 zeigt dies.

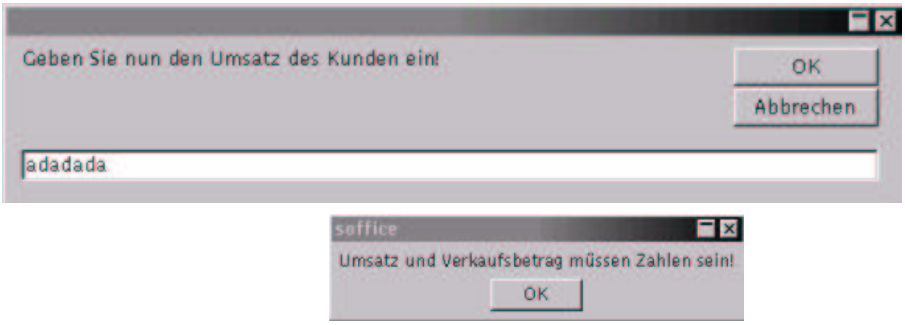

Abbildung 9.10 Realisierung 9.5 mit Eingabefehler beim Umsatz

Realisierung 9.6 Das Provisionsprogramm mit Eingabekontrolle: Nutzung einer String-Variablen

```
Sub provision4()

' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provision4

       dim umsatz As Double
       dim verkaufsbetrag As Double
       dim provisionInProzent As Double
       dim auszuzahlendeProvision As Double
       dim eingabe As String

       ' Umsatzgrenzen sind DM-Betraege

       const umsatzGrenze1 As Double = 100000
       const umsatzGrenze2 As Double = 500000
       const umsatzGrenze3 As Double = 1000000

       ' Provisionen in Prozent

       const provisionUmsatzGrenze1 As Double = 5
       const provisionUmsatzGrenze2 As Double = 10
       const provisionUmsatzGrenze3 As Double = 20

' Gib Programmbeschreibung aus

       MsgBox("Geben Sie Umsatz und Verkaufsbetrag ein!" _
                     & chr$(13) & "Das Programm berechnet die" _
                     & " Provision des Vermittlers!")

' Lies und ueberpruefe die Benutzereingaben

       eingabe = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
       if Not IsNumeric (eingabe)  Then
              MsgBox ("Umsatz muß eine Zahl sein!")
              Exit Sub
       End if
       umsatz = CDbl(eingabe)

       eingabe = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")
       if Not IsNumeric (eingabe)  Then
              MsgBox ("Verkaufsbetrag muß eine Zahl sein!")
              Exit Sub
       End if
```

```
        verkaufsbetrag = CDbl(eingabe)

        if verkaufsbetrag > umsatz Then
                MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
                Exit Sub
        End if

' bestimme Provision

        if umsatz >= umsatzGrenze3 Then
                provisionInProzent = provisionUmsatzGrenze3
        elseIf umsatz >= umsatzGrenze2 Then
                provisionInProzent = provisionUmsatzGrenze2
        elseIf umsatz >= umsatzGrenze1 Then
                provisionInProzent = provisionUmsatzGrenze1
        else
                provisionInProzent = 0
        End if

' Berechne dir Provision

        auszuzahlendeProvision = (verkaufsbetrag * provisionInProzent)/100

' Gib das Ergebnis aus

        MsgBox ("Die Provision für dieses Geschäft ist: " _
                                    & auszuzahlendeProvision & " DM")

End Sub
```

Realisierung 9.6 unterscheidet sich von Realisierung 9.5 durch folgende Zeilen:

```
        eingabe = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
        if Not IsNumeric (eingabe)  Then
                MsgBox ("Umsatz muß eine Zahl sein!")
                Exit Sub
        End if
        umsatz = CDbl(eingabe)

        eingabe = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")
        if Not IsNumeric (eingabe)  Then
                MsgBox ("Verkaufsbetrag muß eine Zahl sein!")
                Exit Sub
        End if
        verkaufsbetrag = CDbl(eingabe)
```

Hier werden die Eingaben erst auf einen `String` eingelesen, der Inhalt des `Strings` wird getestet und, wenn er eine Zahl ist, in eine `Double` umgewandelt und einer `Double`-Variablen zugewiesen. Abbildung 9.11 zeigt eine Fehlerausgabe des Programms.

9.1.4 Mehrseitige Auswahl zum Zweiten (Case Select)

VBA bietet uns eine zweite Möglichkeit, Mehrfachsauswahlen, wie in 9.1.2 zu programmieren. Ich werde dies zunächst an einem Beispiel zeigen:

Realisierung 9.7 Alternative Lösung zu Problemstellung 9.3

```
Option Explicit
```

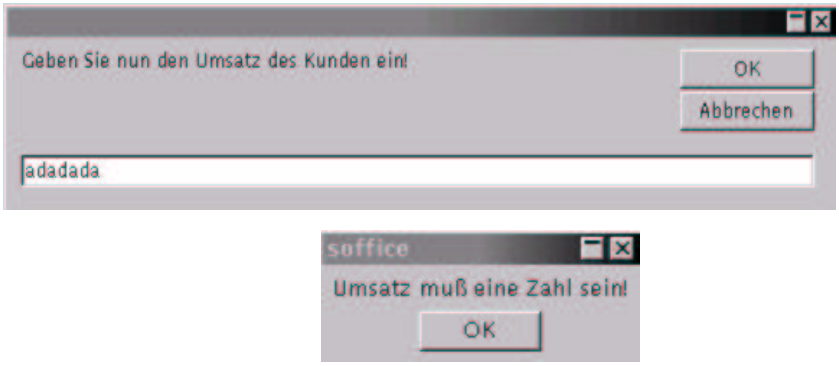

Abbildung 9.11 Realisierung 9.6 mit Eingabefehler beim Umsatz

```
Sub provision3MitCaseSelect()

' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provision3MitCaseSelect

       dim umsatz
       dim verkaufsbetrag
       dim provisionInProzent As Double
       dim auszuzahlendeProvision As Double

       ' Umsatzgrenzen sind DM-Betraege

       const umsatzGrenze1 As Double = 100000
       const umsatzGrenze2 As Double = 500000
       const umsatzGrenze3 As Double = 1000000

       ' Provisionen in Prozent

       const provisionUmsatzGrenze1 As Double = 5
       const provisionUmsatzGrenze2 As Double = 10
       const provisionUmsatzGrenze3 As Double = 20

' Gib Programmbeschreibung aus

       MsgBox("Geben Sie Umsatz und Verkaufsbetrag ein!" _
                     & chr$(13) & "Das Programm berechnet die" _
                     & " Provision des Vermittlers!")

' Lies die Benutzereingaben ein

       umsatz = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
       verkaufsbetrag = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")

' ueberpruefe Benutzereingaben

       if Not IsNumeric (umsatz) or Not IsNumeric (verkaufsBetrag) Then
              MsgBox ("Umsatz und Verkaufsbetrag müssen Zahlen sein!")
              Exit Sub
       End if

       'sicherstellen, dass Benutzereingaben als Double
       'interpretiert werden

       umsatz = CDbl(umsatz)
       verkaufsbetrag = CDbl(verkaufsbetrag)
```

```
        if verkaufsbetrag > umsatz Then
                MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
                Exit Sub
        End if

' bestimme Provision

        Select Case umsatz
                case Is >= umsatzGrenze3
                        provisionInProzent = provisionUmsatzGrenze3
                case Is >= umsatzGrenze2
                        provisionInProzent = provisionUmsatzGrenze2
                case Is >= umsatzGrenze1
                        provisionInProzent = provisionUmsatzGrenze1
                case else
                        provisionInProzent = 0
        End Select

' Berechne dir Provision

        auszuzahlendeProvision = (verkaufsbetrag * provisionInProzent)/100

' Gib das Ergebnis aus

        MsgBox ("Die Provision für dieses Geschäft ist: " _
                                & auszuzahlendeProvision & " DM")

End Sub
```

Diese Lösung unterscheidet sich von Realisierung 9.4 nur durch die Zeilen,

```
        Select Case umsatz
                case Is >= umsatzGrenze3
                        provisionInProzent = provisionUmsatzGrenze3
                case Is >= umsatzGrenze2
                        provisionInProzent = provisionUmsatzGrenze2
                case Is >= umsatzGrenze1
                        provisionInProzent = provisionUmsatzGrenze1
                case else
                        provisionInProzent = 0
        End Select
```

die das `if-elseif`-Konstrukt ersetzen. Im Unterschied zum if-elseif-Konstrukt (dies besteht ja im wesentlichen aus unabhängigen Bedingungen), wählt `Select case` die auszuführenden Aktionen anhand der Werte einer Variablen aus. In Realisierung 9.7 ist dies die Variable *umsatz*. Danach folgen die Auswahlblöcke. Jeder Auswahlblock wird mit `case` eingeleitet. Hinter `case` folgt dann

- ein fester Wert (vgl. Realisierung 9.9),
- eine Menge oder einem Bereich (vgl. Beispiel 9.2)
- oder eine Bedingung. In den Bedingungen wird die Variable, anhand der die Auswahl erfolgt (*umsatz* in unserem Beispiel), durch das Schlüsselwort `Is` repräsentiert (`Is >= umsatzGrenze3`). In Realisierung 9.7 erfolgt also die Auswahl anhand einer Bedingung.

Die Variable, anhand derer die auszuführenden Aktionen ausgewählt werden, heißt Selektor. Die auszuwählenden Aktionen heißen Auswahlblock und das, was hinter `case` steht heißt Auswahlwert. Die allgemeine Form eines `Select-Case`-Konstrukts ist daher:

```
Select Case Selector
       case Auswahlwert1
              Auswahlblock1
       case Auswahlwert2
              Auswahlblock2
       ...
       case Auswahlwertn
              Auswahlblockn
       case else
              AuswahlblockElse
End Select
```

Die Vorgehensweise, anhand derer festgestellt wird, welcher Auswahlblock ausgeführt wird, ist völlig analog zur `if-elseif`-Konstruktion. Die Auswahlwerte werden der Reihenfolge nach mit dem Wert des Selectors verglichen. Stimmen beide überein oder (falls der Auswahlwert eine Bedingung ist) ist die Bedingung erfüllt, wird der zugehörige Auswahlblock durchgeführt Danach wird `Select Case` verlassen und die Ausführung des VBA-Programms setzt hinter `Select Case` fort.

Tritt keine Übereinstimmung auf, wird, soweit vorhanden, der Auswahlblock hinter `Case Else` durchgeführt. Ist `Case Else` nicht vorhanden, setzt die Ausführung sofort hinter `Select Case` fort.

Auf jeden Fall ist auch hier sichergestellt, daß höchstens ein Auswahlblock durchgeführt wird. Beachten Sie daher auch hier die Reihenfolge der Bedingungen in Realisierung 9.7. Wir nutzen hier aus, daß immer nur ein Zweig der gesamten Konstruktion durchlaufen wird. Denn wenn der Umsatz größer 1.000.000 ist, trifft die erste Bedingung zu, und die Anweisungen des ersten Auswahlblocks werden durchgeführt. Danach wird der gesamte `Select-Case`-Block verlassen. Dies bedeutet, daß obwohl der Umsatz in diesem Fall selbstverständlich auch größer als 500000 ist, die zugehörigen Anweisungen nicht durchlaufen werden. Überlegen Sie sich, warum man die Bedingungen nicht in einer anderen Reihenfolge hätte formulieren dürfen[16].

Im Folgenden zeige ich ein Beispiel mit einer Menge als Auswahlwert.

Beispiel 9.2 Menge als Auswahlwert

```
Option Explicit
sub monatBestimmen()
' Programm zum Zeigen von Mengen
' in select Case
' Dateiname: monatBestimmen

dim monat As Integer

monat = InputBox ("Geben Sie einen Monat (Als Zahl) ein!")

select case monat
       case 1 To 3
              MsgBox("Monat liegt im ersten Quartal!")
```

16.Wem das jetzt schwer fällt, der hat in Kapitel 9.1.2 irgendwie doch gepennt!

```
        case 4 To 6
                MsgBox("Monat liegt im zweiten Quartal!")
        case 7 To 9
                MsgBox("Monat liegt im dritten Quartal!")
        case 10 To 12
                MsgBox("Monat liegt im vierten Quartal!")
End Select

End Sub
```

Sie sehen hier, daß Bereiche durch das Schlüsselwort To angegeben werden. Ein Auswahlwert kann auch aus mehreren Bereichen bestehen, diese werden dann durch Kommata voneinander getrennt (`case 1 to 3, 5 to 7, 8`). Abbildung 9.12 zeigt einen Durchlauf durch Beispiel 9.2.

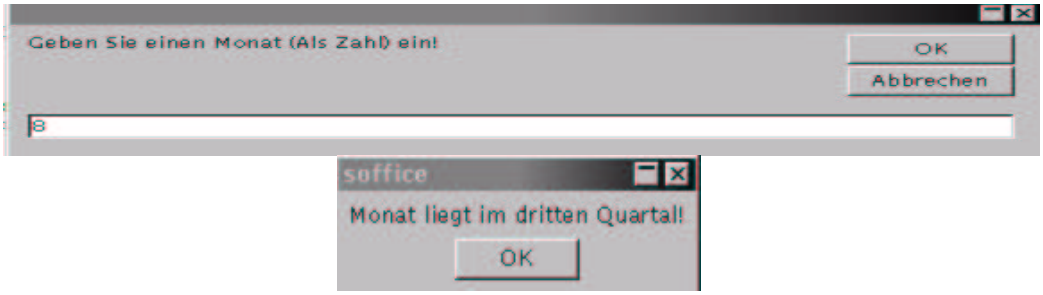

Abbildung 9.12 Bildschirmein- und ausgabe von Beispiel 9.2

Ein Beispiel mit einem einzigen Wert als Auswahlwert folgt im folgenden Kapitel.

Wenn wir mit Select Case arbeiten, sollten wir auch den Pseudocode von Realisierung 9.7 anpassen. Dies ist zwar jetzt nicht ganz die richtige Reihenfolge, denn eigentlich sollte man zunächst den Pseudocode schreiben und dann realisieren, aber hier hatten wir den Pseudocode ja eigentlich schon (durch unser Beispiel mit dem if-elseif-Konstrukt). Wir haben in diesem Kapitel nur eine alternative Realisierung gesehen.

Pseudocode 9.9 Bestimme Provision in Prozent (mit `Select case`)

```
        Select Case umsatz
                case umsatz >= 1000000
                        setze provision auf 20 Prozent
                case umsatz >= 500000
                        setze provision auf 10 Prozent
                case umsatz >= 100000
                        setze provision auf 5 Prozent
                case else
                        setze provision auf 0 Prozent
        end select
```

Abschließend zeige ich eine Darstellung des Algorithmus als Struktogramm:

9.1.5 Ein weiteres Beispiel: Das Taschenrechner-Programm

Problemstellung 9.4 Taschenrechner

Ein kleiner Taschenrechner soll programmiert werden. Der Taschenrechner soll die 4 Grundrechenarten verstehen (+, -, *, /). Das Programm soll 2 Operanden und einen

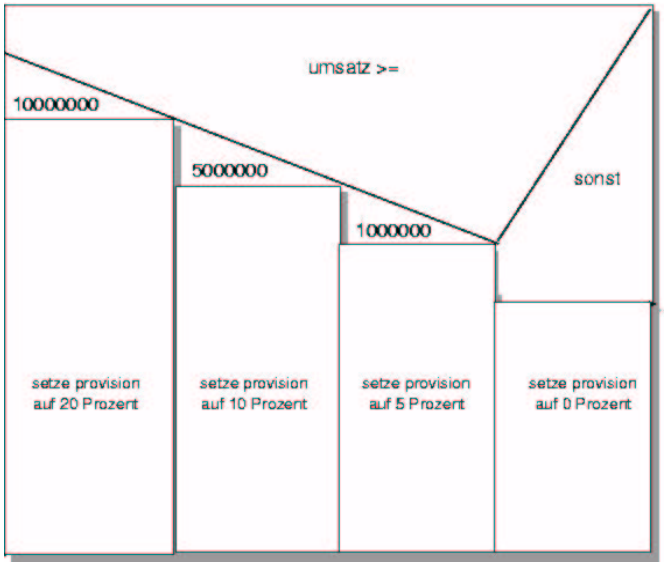

Abbildung 9.13 Struktogrammdarstellung zu "Bestimme Provision in Prozent" (mit Select case)

Operator (+, -, *, /) als Eingabe erhalten, die Berechnung durchführen und das Ergebnis ausgeben. Bei einer Falscheingabe des Operators soll eine Fehlermeldung erfolgen, beim Versuch durch 0 zu teilen ebenfalls. Beim Programmstart soll das Programm eine Beschreibung der Eingabe, die es erwartet, auf den Bildschirm ausgeben. Es soll überprüft werden, ob beide Operanden Zahlen sind.

Wir beginnen wieder mit Pseudocode:

Pseudocode 9.10 Taschenrechner

```
Gib Programmbeschreibung aus
Lies und überprüfe die Benutzereingaben
führe Berechnung durch
Gib das Ergebnis aus
```

"Gib Programmbeschreibung aus", "Lies die Benutzereingaben ein" und "Gib das Ergebnis aus" sind einfach. Dies realisieren wir mit `MsgBoxen` bzw. `InputBoxen`. "überprüfe Benutzereingaben" ist ebenfalls nicht gerade schwer, da können wir Code-Teile aus Realisierung 9.5 oder Realisierung 9.6 nutzen. Dort geht es ja gerade um solche Überprüfungen und ob ich eine Variable *umsatz* oder *ersterOperand* daraufhin überprüfe, ob sie eine Zahl enthält, ist ja ungefähr vom gleichen Schwierigkeitsgrad.

Also verfeinern wir nur noch "führe Berechnung durch".

Pseudocode 9.11 "führe Berechnung durch" (mit `if-elseif`)

```
if eingegebener Operator = "+" then
       Operanden addieren
elseif eingegebener Operator = "-" Then
       Operanden subtrahieren
elseif eingegebener Operator = "*" Then
       Operanden multiplizieren
elseif eingegebener Operator = "/" Then
       Operanden dividieren mit Überprüfung ob als Nenner 0 eingegeben wurde
else
```

```
              MsgBox mit Hinweis falscher Operator ausgeben
              Programm verlassen
       end if
```

Pseudocode 9.12 "führe Berechnung durch" (mit `select case`)

```
       select case operator
              case "+"
                     Operanden addieren
              case "-"
                     Operanden subtrahieren
              case   "*"
                     Operanden multiplizieren
              case   "/"
                     Operanden dividieren mit Überprüfung ob als Nenner 0 eingegeben
                     wurde
              case else
                     MsgBox mit Hinweis falscher Operator ausgeben
                     Programm verlassen
       end select
```

Die Beschreibung des Algorithmus durch Struktogramme schenke ich mir. Durch den Pseudocode ergeben sich folgende Realisierungen:

Realisierung 9.8 Taschenrechner mit `if-elseif`

```
Sub taschenrechner()
' Taschenrechnerprogramm
' Dateiname: taschenrechner

dim ersterOperand As Double
dim zweiterOperand As Double
dim operator As String
dim eingabe as String
dim ergebnis As Double

' Gib Programmbeschreibung aus

       MsgBox("Geben Sie zwei Operanden und einen Operator ein!" _
                     & chr$(13) & "Das Programm verhält sich wie ein" _
                     & " Taschenrechner. Als Operatoren sind +, - / und *" _
                     & " zugelassen!")

' Lies und ueberpruefe die Benutzereingaben

       eingabe = InputBox ("Geben Sie nun den ersten Operanden ein!")
       if Not IsNumeric (eingabe)  Then
              MsgBox ("Erster Operand muß eine Zahl sein!")
              Exit Sub
       End if
       ersterOperand = CDbl(eingabe)

       operator = InputBox ("Geben Sie nun den Operator ein!")

       eingabe = InputBox ("Geben Sie nun den zweiten Operanden ein!")
       if Not IsNumeric (eingabe)  Then
              MsgBox ("Zweiter Operand muß eine Zahl sein!")
              Exit Sub
       End if
       zweiterOperand = CDbl(eingabe)

' Fuehre Berechnung durch
```

```
        if operator = "+" Then
                ergebnis = ersterOperand + zweiterOperand
        elseIf operator = "-" Then
                ergebnis = ersterOperand - zweiterOperand
        elseIf operator = "*" Then
                ergebnis = ersterOperand * zweiterOperand
        elseIf operator = "/" Then
                if (zweiterOperand = 0) Then
                        MsgBox ("Der Nenner ist 0!")
                        Exit Sub
                End if
                ergebnis = ersterOperand / zweiterOperand
        else
                MsgBox ("Der eingegebene Operator wird nicht " _
                                & "unterstützt!")
                Exit Sub
        End if

' Gib das Ergebnis aus

        MsgBox (" " & ersterOperand & " "  & operator & " " _
                        & zweiterOperand & " = " & ergebnis)

End Sub
```

Realisierung 9.9 Taschenrechner mit `case select`

```
Option Explicit
Sub taschenrechnerMitCaseSelect()
' Taschenrechnerprogramm
' Dateiname: taschenrechnerMitCaseSelect

dim ersterOperand As Double
dim zweiterOperand As Double
dim operator As String
dim eingabe as String
dim ergebnis As Double

' Gib Programmbeschreibung aus

        MsgBox("Geben Sie zwei Operanden und einen Operator ein!" _
                        & chr$(13) & "Das Programm verhält sich wie ein" _
                        & " Taschenrechner. Als Operatoren sind +, - / und *" _
                        & " zugelassen!")

' Lies und ueberpruefe die Benutzereingaben

        eingabe = InputBox ("Geben Sie nun den ersten Operanden ein!")
        if Not IsNumeric (eingabe)  Then
                MsgBox ("Erster Operand muß eine Zahl sein!")
                Exit Sub
        End if
        ersterOperand = CDbl(eingabe)

        operator = InputBox ("Geben Sie nun den Operator ein!")

        eingabe = InputBox ("Geben Sie nun den zweiten Operanden ein!")
        if Not IsNumeric (eingabe)  Then
                MsgBox ("Zweiter Operand muß eine Zahl sein!")
                Exit Sub
        End if
```

```
        zweiterOperand = CDbl(eingabe)

' Fuehre Berechnung durch

        Select Case operator
                case "+"
                        ergebnis = ersterOperand + zweiterOperand
                case "-"
                        ergebnis = ersterOperand - zweiterOperand
                case "*"
                        ergebnis = ersterOperand * zweiterOperand
                case "/"
                        if (zweiterOperand = 0) Then
                                MsgBox ("Der Nenner ist 0!")
                                Exit Sub
                        End if
                        ergebnis = ersterOperand / zweiterOperand
                case else
                        MsgBox ("Der eingegebene Operator wird nicht " _
                                        & "unterstützt!")
                        Exit Sub
        End Select


' Gib das Ergebnis aus

        MsgBox (" " & ersterOperand & " "  & operator & " " _
                                & zweiterOperand & " = " & ergebnis)

End Sub
```

Abbildung 9.14 zeigt einen beispielhaften Durchlauf durch Realisierung 9.8 bzw. Realisierung 9.9:

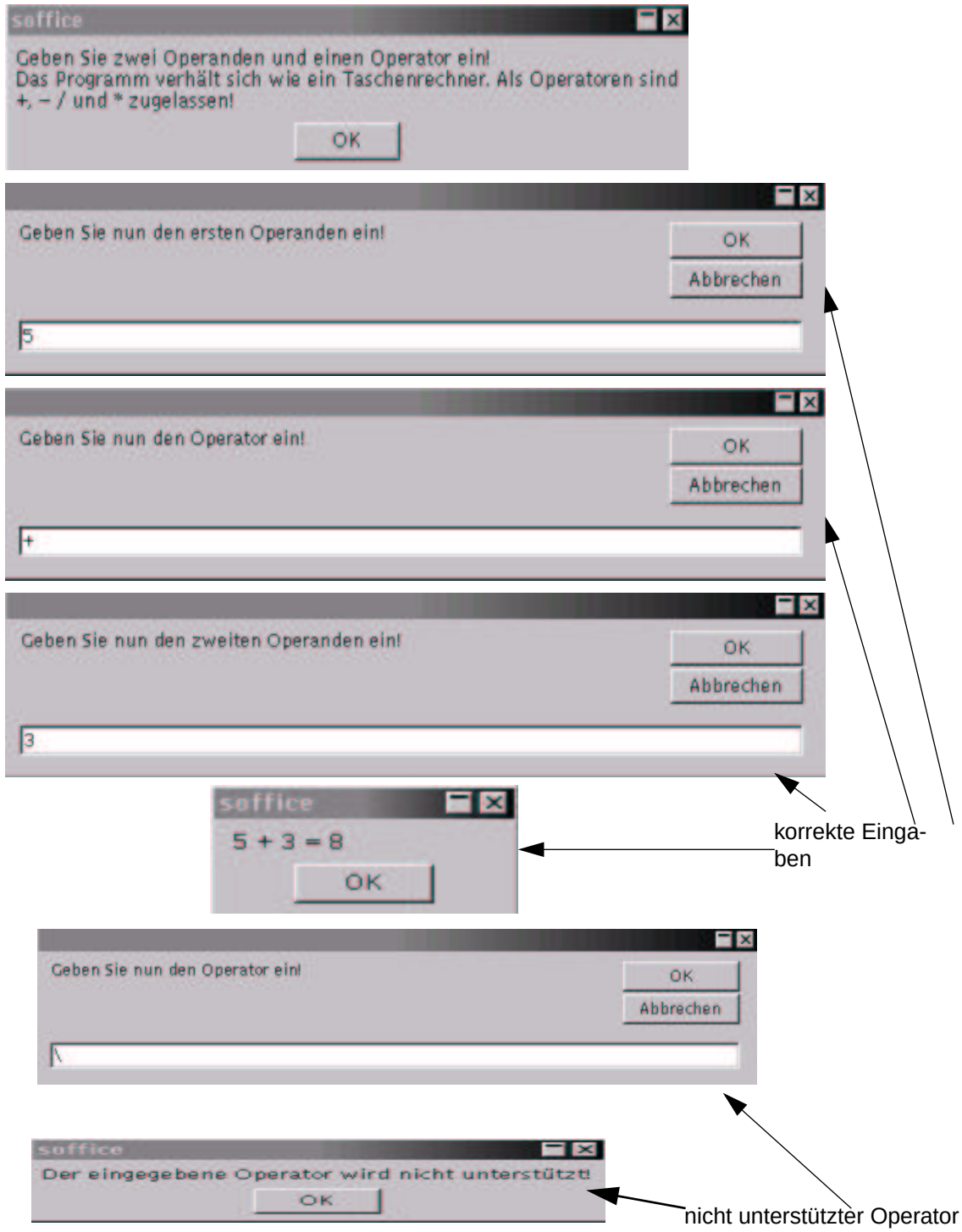

Abbildung 9.14 Realisierung 9.9 bzw. Realisierung 9.8 mit korrekte Eingaben und mit nicht unterstütztem Operator

9.2 Wiederholungsanweisungen (Schleifen)

9.2.1 Die do-while-Anweisung

Unsere Programme (Taschenrechner oder Provision berechnen) sind immer noch nicht ganz genial. Nach jeder Berechnung hören sie einfach auf. Wollen wir mehr als eine Berechnung durchführen, müssen wir das jeweilige Programm jeweils neu starten. Besser wäre es, wenn das Programm neue Eingaben erwarten würde und sich erst beendet, wenn wir dem Programm sagen: "Jetzt ist Schluß", z. B. indem wir "beenden" eingeben.

So etwas können wir aber noch nicht, da wir keine Möglichkeit haben, Anweisungsfolgen mehrfach durchführen zu lassen.

Und genau das müßten wir aber machen. Wenn das Programm die Ausgabe der ersten Berechnung auf den Bildschirm geschrieben hat, müßte es (im Taschenrechnerbeispiel) erneut 2 Zahlen und einen Operator einlesen (also die Anweisungen in '"Lies und ueberpruefe die Benutzereingaben" noch einmal durchführen), erneut eine Berechnung vornehmen (also die Anweisungen in "Fuehre Berechnung durch" noch einmal durchführen) und schließlich das neue Ergebnis ausgeben (also die Anweisungen in "Gib das Ergebnis aus" noch einmal durchführen).

Diese ganze Abfolge muß so lange wiederholt werden, bis der Anwender z.B. als Operator "beenden" eingibt.

Die Lösung eines solchen Problems erfolgt durch Schleifen. VBA kennt mehrere Schleifen-Konstrukte. Wir beginnen mit der `Do While`-Schleife.

Die `Do While`-Schleife hat die Form:

```
do while logischer Ausdruck
       anweisung1
              :
              :
       anweisungN
Loop
```

Wirkung:

Wenn das Programm auf Do While trifft, wird zunächst `logischer Ausdruck` ausgewertet. Ergibt `logischer Ausdruck` den Wert `false` wird die gesamte Schleife ignoriert (nicht ausgeführt). Das bedeutet, das Programm setzt mit der auf `Loop` folgenden Anweisung fort.

Ergibt `logischer Ausdruck` hingegen den Wert `true` werden die Anweisungen zwischen `do while` und `Loop` ausgeführt (Eintritt in die Schleife) .

Immer dann, wenn das Programm auf die Anweisung Loop der Schleife trifft, wird `logischer Ausdruck` erneut überprüft. Ergibt `logischer Ausdruck "true",` wird die Schleife erneut ausgeführt. Ergibt `logischer Ausdruck "false",` wird die Schleife abgebrochen. Das bedeutet, das Programm setzt mit der auf Loop folgenden Anweisung fort.

Ist `logischer Ausdruck` bei der ersten Auswertung `false,` werden die Anweisungen der Schleife nie ausgeführt (abweisende Schleife).

Abweisende Schleifen wie die `do while`-Schleife nennt man auch kopfgesteuerte Schleifen, weil die Überprüfung der Schleifenbedingung beim Start der Schleife durchgeführt wird.

Beispiel 9.3 Nie ausgeführte Schleife

```
Option Explicit
Sub nieschleife()

' Programm mit einer nie durchgefuehrten Schleife
' nicht sehr sinnig
' Dateiname nieschleife

       const falsch As Boolean = false
       do while falsch
              ' Hier kommen wir nie hin
              MsgBox ("Dies wird kein Benutzer jemals sehen!")
       loop
end sub
```

Nach den Kommentaren wird in Beispiel 9.3 eine bool'sche Konstante mit dem Wert `false` definiert:

```
       const falsch As Boolean = false
```

In der folgenden Zeile

```
       do while falsch
```

ist `logischer Ausdruck` der Wert der Konstanten *falsch*. Der ist aber `false`. Das bedeutet, die Schleife wird nicht ausgeführt und VBA setzt die Ausführung des Programms mit der nächsten auf die Schleife folgenden Anweisung fort. Dies ist aber bereits `end sub` und das Programm wird beendet.

Wird `logischer Ausdruck` nie `false` wird die Schleife nie abgebrochen (Endlosschleife). Die Anweisungen der Schleife werden unendlich oft wiederholt (dies ist selten vom Programmierer so beabsichtigt).

Beispiel 9.4 Endlosschleife

```
Option Explicit
Sub endlosschleife()

' Programm mit einer Endlosschleife
' nicht sehr sinnig
' Dateiname: Endlosschleife

       const wahr As Boolean = true
       do while wahr
              ' In die Endlosschleife
              MsgBox ("Jetzt müssen Sie StarOffice abschießen!")
       loop
end sub
```

Hier wird zunächst eine bool'sche Konstante mit dem Wert `true` definiert:

```
const wahr As Boolean = true
```

In der folgenden Zeile

```
do while wahr
```

ist `logischer Ausdruck` der Wert der Konstanten *wahr*. Der ist aber `true`. Das bedeutet, VBA verzweigt in die Schleife und führt die Anweisung

```
MsgBox ("Jetzt müssen Sie StarOffice abschießen!")
```

aus. Das in Abbildung 9.15 dargestellte Fenster erscheint.

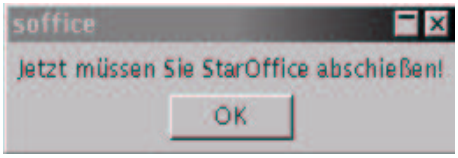

Abbildung 9.15 Bildschirmausgabe von Beispiel 9.4

Wenn Sie auf "OK" clicken, verschwindet das Fenster und VBA trifft auf die Zeile

```
loop
```

Dies veranlaßt VBA zum Beginn der Schleife zurückzukehren. Es wird also wieder die Anweisung

```
do while wahr
```

durchgeführt. Am Wert von *wahr* hat sich aber nichts geändert, Konstanten können ihren Wert ja gar nicht ändern, wahr ist immer noch `true` und VBA verzweigt in die Schleife. Also wird

```
MsgBox ("Jetzt müssen Sie StarOffice abschießen!")
```

ausgeführt und wir haben unser Fenster wieder. Wir können mit "OK" bestätigen, woraufhin

```
loop
```

ausgeführt wird. VBA kehrt somit zum Beginn der Schleife zurück. Unser Fenster erscheint usw. Wir haben nicht mehr die Chance, dieses Programm zu beenden. Da durch `MsgBox` erzeugte Fenster sofort den Fokus erhalten[17], können wir das Programm nicht einmal dadurch beenden, das wir StarOffice oder Excel beenden[18]. Die einzige Möglichkeit ist, über einen Task Manager StarOffice oder Excel abzuschießen.

Der Wert von `logischer Ausdruck` muß also innerhalb der Schleife geändert werden (wenn man keine Endlosschleife zu programmieren beabsichtigt).

Betrachten wir die `do while`-Schleife nun anhand eines einfachen Beispiels:

17.bedeutet, das Fenster ist aktiviert und ich kann, so lange das Fenster zu sehen ist, nirgendwo anders in der Anwendung Eingaben tätigen

18.Es sei denn, Sie benutzen einen alten 286 mit 16 MB RAM. Dann könnte vielleicht die Zeit ausreichen, die der Rechner benötigt, um die neue MsgBox aufzubauen.

Problemstellung 9.5 Addieren mit der Do while-Schleife

Wir wollen ein Programm zur Addition zweier Zahlen schreiben. Die Zahlen sollen über Bildschirmfenster eingegeben werden. Das Programm soll so lange laufen, bis der Benutzer als ersten Summanden 0 eingibt. Vor der ersten Berechnung soll das Programm eine Bedienungsanleitung ausgeben.

Hierzu schreiben wir zunächst Pseudocode:

Pseudocode 9.13 Summieren mit der Do while-Schleife

```
Gib Programmbeschreibung aus
Lies ersten Summanden ein
Do while erster Summand nicht null ist
       Lies zweiten Summanden ein
       Bilde die Summe
       Gib die Summe am Bildschirm aus
       Lies ersten Summanden ein
loop
```

Zunächst sehen wir, daß wir eine `Do while`-Schleife im Pseudocode einfach durch Hinschreiben der Schlüsselworte `Do while` beginnen und mit dem Schlüsselwort `loop` beenden. Dies beraubt uns nicht der Programmiersprachen-Unabhängigkeit des Pseudocodes, weil eine jede Programmiersprache ein ähnliches Konstrukt enthält. Beachten Sie auch die Einrückungen: Alle Anweisungen der Schleife sind um eine Tabulatorposition eingerückt.

Das erste Einlesen des ersten Summanden erfolgt, bevor die Schleife die Bedingung (erster Summand ungleich 0) überprüft, also vor dem Schleifenstart. Dies müssen wir so machen, damit die Überprüfung in der Schleifenbedingung sinnvoll durchgeführt werden kann. Denn sonst hätte der erste Summand keinen definierten Wert und die Überprüfung der Schleifenbedingung wäre zufällig.

In der Schleife lesen wir zunächst den zweiten Summanden ein. Dies tun wir nicht ebenfalls vor der Schleife, damit der Benutzer, wenn er als ersten Summanden "0" eingegeben hat und damit keine Berechnung wünscht, nicht mehr mit dem zweiten Eingabefenster konfrontiert wird. Dann wird die Berechnung durchgeführt und das Ergebnis ausgegeben. Am Ende der Schleife lesen wir den ersten Summanden für den nächsten Schleifendurchlauf ein. Diese Zeile nicht zu vergessen ist ziemlich wichtig, denn wenn wir den ersten Summanden nicht erneut einlesen, behält der erste Summand den Wert der ersten Eingabe.

Durch `loop` kehrt die Schleife an ihren Anfang zurück und überprüft, ob der erste Summand nicht null ist. Hätten wir also den ersten Summanden nicht erneut eingelesen, wäre der erste Summand auch nun nicht null, denn sonst hätte die Schleife nie gelaufen. Also wird in die Schleife verzweigt, der zweite Summand eingelesen usw. Wir hätten eine Endlosschleife.

Da wir aber den ersten Summanden neu einlesen, findet eine sinnvolle Überprüfung statt. Bei der Eingabe von "0" durch den Benutzer wird die Schleife verlassen, andernfalls wird eine neue Berechnung durchgeführt. Realisierung 9.10 zeigt die Realisierung des Pseudocodes in VBA. Beachten Sie auch hier die Einrückungen.

Realisierung 9.10 Summieren mit der Do while-Schleife

```
Sub additionMitDoWhile()

' Programm addiert die zwei einzugebende Zahlen
' Dateiname: additionMitDoWhile

       dim ersterSummand As Integer
       dim zweiterSummand As Integer
       dim summe As Integer

       ' Gib Programmbeschreibung aus

       MsgBox("Geben Sie Zwei Summanden ein!" _
                      & chr$(13) & "Das Programm berechnet die" _
                      & " Summe ! Sie beenden das" _
                      & " Programm durch die Eingabe 0!")

       ' Lies ersten Summanden ein

       ersterSummand = InputBox("Bitte geben Sie den ersten Summanden ein")

       do while ersterSummand <> 0

              ' Lies zweiten Summanden ein

              zweiterSummand =  InputBox("Bitte geben Sie den zweiten Summanden ein")

              ' Berechne die Summe

              summe = ersterSummand + zweiterSummand

              ' Gib das Ergebnis aus

              MsgBox ("Summe: " & summe)

              ' Lies ersten Summanden ein

              ersterSummand = InputBox("Bitte geben Sie den ersten Summanden ein")
       loop


End Sub
```

Wir erweitern nun unser Provisionsprogramm.

Problemstellung 9.6 Provisionsberechnung mit Do-While Schleife

Ein VBA-Programm soll die zusätzliche Provision für einen Verkauf eines Vermittlers anhand des geplanten Jahresumsatzes berechnen. Die Provision soll nach folgender Tabelle gewährt werden::

Umsatz	Provision in Prozent
bis 100.000	0
100.000 bis 500.000	5
500.000 bis 1.000.000	10
über 1.0000.000	20

Das Programm soll zunächst in einem Fenster eine kurze Bedienungsanleitung ausgeben, dann werden Umsatz und jetziger Verkaufsbetrag eingegeben. Liegt der Verkaufsbetrag, für den die Provision gerade berechnet werden soll, über dem geplanten Umsatz für das ganze Jahr, soll das Programm eine Fehleingabe vermuten und eine Fehlermeldung ausgeben. Ansonsten wird der Provisionbetrag errechnet und ausgegeben.

Das Programm soll die Berechnung mehrerer Provisionen erlauben. Es soll abbrechen, wenn als Umsatz "beenden" eingegeben wird.

Wie immer entwickeln wir zunächst den Pseudocode:

Pseudocode 9.14 Provision berechnen mit Do-While Schleife

```
Gib Programmbeschreibung aus
Lies den umsatz ein
do while umsatz nicht gleich "beenden"
       lies restliche und überprüfe alle Benutzereingaben
       bestimme Provision in Prozent
       berechne auszuzahlenden Betrag Formel:
       (verkaufsbetrag*provision in prozent)/100
       Gib das Ergebnis aus
       Lies den umsatz ein
Loop
```

Auch hier sehen wir, daß wir die Aktion "Lies den umsatz ein" zweimal durchführen müssen. Einmal vor der Schleife, um eine sinnvolle Überprüfung der Schleifenbedingung beim ersten Durchlauf der Schleife zu gewährleisten, und einmal am Ende der Schleife, um neue Benutzereingaben verarbeiten zu können. Abbildung 9.16 zeigt den Algorithmus als Struktogramm.

Realisierung 9.11 Provisionsberechnung mit Do While-Schleife

```
Option Explicit
Sub provisionMitDoWhile

' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provision4

       dim umsatz
```

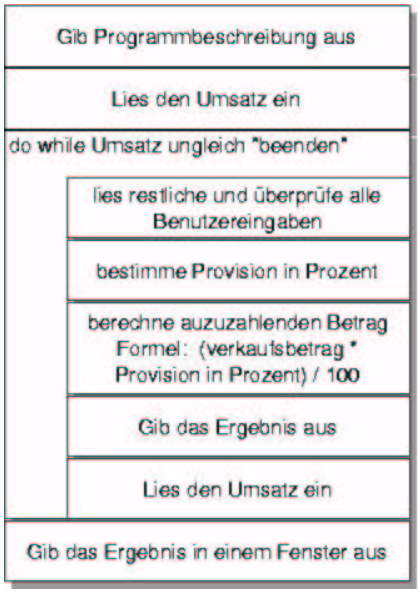

Abbildung 9.16 Struktogrammdarstellung zu "Provisionsberechnung mit Do While-Schleife"

```
dim verkaufsbetrag
dim provisionInProzent As Double
dim auszuzahlendeProvision As Double
dim eingabe As String

' Umsatzgrenzen sind DM-Betraege

const umsatzGrenze1 As Double = 100000
const umsatzGrenze2 As Double = 500000
const umsatzGrenze3 As Double = 1000000

' Provisionen in Prozent

const provisionUmsatzGrenze1 As Double = 5
const provisionUmsatzGrenze2 As Double = 10
const provisionUmsatzGrenze3 As Double = 20

' Gib Programmbeschreibung aus

MsgBox("Geben Sie Umsatz und Verkaufsbetrag ein!" _
              & chr$(13) & "Das Programm berechnet die" _
              & " Provision des Vermittlers! Sie beenden das" _
              & " Programm durch die Eingabe von: beenden!")



' Lies Umsatz ein

eingabe = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")

Do while eingabe <> "beenden"
' Lies und ueberpruefe die Benutzereingaben
      if Not IsNumeric (eingabe)  Then
            MsgBox ("Umsatz muß eine Zahl sein!")
            Exit Sub
      End if
      umsatz = CDbl(eingabe)
```

```
            eingabe = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")
            if Not IsNumeric (eingabe)  Then
                  MsgBox ("Verkaufsbetrag muß eine Zahl sein!")
                  Exit Sub
            End if
            verkaufsbetrag = CDbl(eingabe)

            if verkaufsbetrag > umsatz Then
                  MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
                  Exit Sub
            End if

            ' bestimme Provision

            if umsatz >= umsatzGrenze3 Then
                  provisionInProzent = provisionUmsatzGrenze3
            elseIf umsatz >= umsatzGrenze2 Then
                  provisionInProzent = provisionUmsatzGrenze2
            elseIf umsatz >= umsatzGrenze1 Then
                  provisionInProzent = provisionUmsatzGrenze1
            else
                  provisionInProzent = 0
            End if

            ' Berechne dir Provision

            auszuzahlendeProvision = (verkaufsbetrag * provisionInProzent)/100

            ' Gib das Ergebnis aus

            MsgBox ("Die Provision für dieses Geschäft ist: " _
                              & auszuzahlendeProvision & " DM")

            ' Lies Umsatz ein

            eingabe = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
      Loop

End Sub
```

Ich weise noch einmal auf die Reihenfolge hin:

- Zunächst wird die Variable umsatz eingelesen
  - beim ersten Mal vor der Schleife,
  - bei jedem weiteren Schleifendurchlauf als letzte Anweisung des vorherigen Schleifendurchlaufs.

    ```
    eingabe = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
    ```

- Dann wird die Schleifenbedingung überprüft.

    ```
    Do while eingabe <> "beenden"
    ```

- Erst danach können wir den `String` in einen `Double` umwandeln.

    ```
    umsatz = CDbl(eingabe)
    ```

Zum Abschluß erweitern wir das Taschenrechner-Beispiel um eine do while-Schleife.

Problemstellung 9.7 Taschenrechner mit do while-Schleife

Ein kleiner Taschenrechner soll programmiert werden. Der Taschenrechner soll die 4 Grundrechenarten verstehen (+, -, *, /). Das Programm soll 2 Operanden und einen Operator (+, -, *, /) als Eingabe erhalten, die Berechnung durchführen und das Ergebnis ausgeben. Bei einer Falscheingabe des Operators soll eine Fehlermeldung erfolgen, beim Versuch durch 0 zu teilen ebenfalls. Beim Programmstart soll das Programm eine Beschreibung der Eingabe, die es erwartet, auf den Bildschirm ausgeben. Es soll überprüft werden, ob beide Operanden Zahlen sind. Das Programm soll beliebig viele Berechnungen ermöglichen und abbrechen, wenn der Benutzer als ersten Operanden "0" eingibt.

Pseudocode 9.15 Taschenrechner mit do while-Schleife

```
Gib Programmbeschreibung aus
Lies ersten Operanden ein
do while erster Operand ist ungleich 0
       Lies restliche und überprüfe alle Benutzereingaben
       führe Berechnung durch
       Gib das Ergebnis aus
       Lies ersten Operanden ein
loop
```

Realisierung 9.12 Taschenrechner mit do while-Schleife

```
Option Explicit
Sub taschenrechnerMitDoWhile()
' Taschenrechnerprogramm
' Dateiname: taschenrechnerMitDoWhile

dim ersterOperand As Double
dim zweiterOperand As Double
dim operator As String
dim eingabe as String
dim ergebnis As Double

' Gib Programmbeschreibung aus

       MsgBox("Geben Sie zwei Operanden und einen Operator ein!" _
                     & chr$(13) & "Das Programm verhält sich wie ein" _
                     & " Taschenrechner. Als Operatoren sind +, - / und *" _
                     & " zugelassen! Das Programm endet durch die Eingabe von " _
                      & chr$(13) & "0 als erstem Summanden!")

' Lies ersten Summanden ein
       eingabe = InputBox ("Geben Sie nun den ersten Operanden ein!")

       do while eingabe <> "0"

' Lies restliche und ueberpruefe alle Benutzereingaben
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Erster Operand muß eine Zahl sein!")
                     Exit Sub
              End if
              ersterOperand = CDbl(eingabe)

              operator = InputBox ("Geben Sie nun den Operator ein!")
```

```
            eingabe = InputBox ("Geben Sie nun den zweiten Operanden ein!")
            if Not IsNumeric (eingabe)  Then
                  MsgBox ("Zweiter Operand muß eine Zahl sein!")
                  Exit Sub
            End if
            zweiterOperand = CDbl(eingabe)

' Fuehre Berechnung durch

            Select Case operator
                  case "+"
                        ergebnis = ersterOperand + zweiterOperand
                  case "-"
                        ergebnis = ersterOperand - zweiterOperand
                  case "*"
                        ergebnis = ersterOperand * zweiterOperand
                  case "/"
                        if (zweiterOperand = 0) Then
                              MsgBox ("Der Nenner ist 0!")
                              Exit Sub
                        End if
                        ergebnis = ersterOperand / zweiterOperand
                  case else
                        MsgBox ("Der eingegebene Operator wird nicht " _
                                    & "unterstützt!")
                        Exit Sub
            End Select


' Gib das Ergebnis aus

            MsgBox (" " & ersterOperand & " "  & operator & " " _
                        & zweiterOperand & " = " & ergebnis)

' Lies ersten Operanden ein

            eingabe = InputBox ("Geben Sie nun den ersten Operanden ein!")

      loop
End Sub
```

9.2.2 Die loop until-Schleife

Neben der `do while`-Schleife gibt es in VBA weitere Möglichkeiten, die wiederholte Durchführung von Anweisungen zu erzwingen. Eine davon ist die `loop until`-Schleife.

Die `loop until`-Schleife hat die Form:

```
      do
            anweisung1
                  :
                  :
            anweisungN
      Loop until logischer Ausdruck
```

Wenn das Programm auf die `loop until`-Anweisung trifft, wird `logischer Ausdruck` ausgewertet.

Wenn logischer Ausdruck `true` ist, passiert nichts weiter, das Programm fährt einfach mit der auf `loop until` folgenden Anweisung fort, als hätte es `loop until` nie gegeben.

Ist `logischer Ausdruck` jedoch `false` werden die zwischen `do` und `loop until` stehenden Anweisungen erneut durchgeführt. Jedesmal, wenn das Programm im weiteren Verlauf auf `loop until` trifft, wird `logischer Ausdruck` erneut ausgewertet. Ist logischer Ausdruck `false` wird die Schleife erneut ausgeführt, andernfalls (`true`) verlassen.

Die loop until-Schleife wird mindestens einmal durchlaufen (logischer Ausdruck wird am Ende der Schleife überprüft).

Dies ist der Unterschied zur `do while`-Schleife.

Wird logischer Ausdruck nie `true` haben wir wieder eine Endlosschleife. Will man das vermeiden muß der Wert von logischer Ausdruck in der `loop until`-Schleife geändert werden. Dies zeigt Beispiel 9.5.

Beispiel 9.5 Endlosschleife mit loop until

```
Option Explicit
Sub endlosschleifeMitLoopUntil()

' Programm mit einer Endlosschleife
' nicht sehr sinnig
' Dateiname: endlosschleifeMitLoopUntil


	const falsch As Boolean = false
	do
		' In die Endlosschleife
		MsgBox ("Jetzt müssen Sie StarOffice abschießen!")
	loop until falsch
end sub
```

Da der Wert von *falsch* nie `true` werden kann, wird auch in diesem Beispiel die Schleife nie verlassen und uns bleibt als einzige Möglichkeit die gesamte Anwendung abzuschießen.

Für den Nichteintritt in eine `loop until`-Schleife gibt es kein Beispiel, da das nun mal nicht geht.

Schleifenkonstrukte wie die `loop until`-Schleife nennt man auch fußgesteuerte Schleifen, da die Überprüfung der Schleifenbedingung am Ende der Schleife erfolgt.

Betrachten wir zunächst wieder Problemstellung 9.5. Wir wollen diesmal aber eine Realisierung mit der `loop until`-Schleife programmieren. Auch hier zunächst der Pseudocode:

Pseudocode 9.16 Summieren mit der loop until-Schleife

```
	Gib Programmbeschreibung aus
	Lies ersten Summanden ein
	Do
		Lies zweiten Summanden ein
		Bilde die Summe
```

```
        Gib die Summe am Bildschirm aus
        Lies ersten Summanden ein
    loop until erster Summand null ist
```

Zunächst sehen wir, daß wir eine `loop until`-Schleife im Pseudocode einfach durch Hinschreiben des Schlüsselwortes `Do` beginnen und mit den Schlüsselworten `loop until` beenden. Dies beraubt uns nicht der Programmiersprachen-Unabhängigkeit des Pseudocodes, weil eine jede Programmiersprache ein ähnliches Konstrukt enthält.

Das erste Einlesen des ersten Summanden erfolgt vor dem Schleifenstart. Wir könnten jetzt zwar auch in der Schleife einlesen ("Lies ersten Summanden ein" als erste Anweisung in der Schleife), da die Überprüfung der Schleifenbedingung erst am Ende erfolgt. Dies hat aber folgenden Nachteil:

In dem Moment, in dem der Benutzer das Programm beenden will und damit "0" als ersten Summanden eingibt, erwartet er sicherlich auch, daß sich das Programm beendet. Ist "Lies ersten Summanden ein" aber die erste Anweisung der Schleife, wird die gesamte Schleife noch einmal durchlaufen. Das heißt, der Benutzer wird nach dem zweiten Summanden gefragt, das Programm rechnet nochmal, gibt das Ergebnis aus und bricht dann erst ab.

In der Schleife lesen wir zunächst den zweiten Summanden ein. Dann wird die Berechnung durchgeführt und das Ergebnis ausgegeben. Am Ende der Schleife lesen wir den ersten Summanden für den nächsten Schleifendurchlauf ein. Diese Zeile nicht zu vergessen ist ziemlich wichtig, denn wenn wir den ersten Summanden nicht erneut einlesen, behält der erste Summand den Wert der ersten Eingabe (vgl. Kapitel 9.2.1).

Durch `loop until` wird nun überprüft, ob der erste Summand null ist. Ist dies der Fall, wird die Schleife verlassen, andernfalls kehrt VBA zur `do` Anweisung zurück und führt die Schleife noch einmal durch.

Der Nachteil der `loop until`-Schleife ist, daß sie auf jeden Fall einmal läuft. Startet der Benutzer unser Programm, überlegt sich aber, es doch nicht zu nutzen und gibt direkt als erste Eingabe für den ersten Summanden "0" ein, so läuft das Programm trotzdem zunächst einmal durch. Realisierung 9.13 zeigt die Realisierung des Pseudocodes in VBA.

Realisierung 9.13 Summieren mit der loop until-Schleife

```
Option Explicit
Sub additionMitDoUntil()

' Programm addiert die zwei einzugebende Zahlen
' Dateiname: additionMitDoUntil

       dim ersterSummand As Integer
       dim zweiterSummand As Integer
       dim summe As Integer

       ' Gib Programmbeschreibung aus

       MsgBox("Geben Sie Zwei Summanden ein!" _
                     & chr$(13) & "Das Programm berechnet die" _
                     & " Summe ! Sie beenden das" _
                     & " Programm durch die Eingabe 0!")
```

```
        ' Lies ersten Summanden ein

        ersterSummand = InputBox("Bitte geben Sie den ersten Summanden ein")

        do

                ' Lies zweiten Summanden ein

                zweiterSummand =  InputBox("Bitte geben Sie den zweiten Summanden ein")

                ' Berechne die Summe

                summe = ersterSummand + zweiterSummand

                ' Gib das Ergebnis aus

                MsgBox ("Summe: " & summe)

                ' Lies ersten Summanden ein

                ersterSummand = InputBox("Bitte geben Sie den ersten Summanden ein")

        loop until ersterSummand = 0


End Sub
```

Als nächstes zeige ich Pseudocode und Realisierung des Provisions- und des Taschenrechnerbeispiels mit `loop until`. Beim Provisionsbeispiel zeige ich zusätzlich die Darstellung des Algorithmus als Struktogramm. Mit ihrem jetzigen Wissen müßten die Beispiele (zumindest hoffe ich das) selbsterklärend sein.

Pseudocode 9.17 Provision berechnen mit loop until-Schleife

```
        Gib Programmbeschreibung aus
        Lies den umsatz ein
        do
                lies restliche und überprüfe alle Benutzereingaben
                bestimme Provision in Prozent
                berechne auszuzahlenden betrag Formel:
                (verkaufsbetrag*provision in prozent)/100
                Gib das Ergebnis aus
                Lies den umsatz ein
        Loop until umsatz gleich "beenden"
```

Realisierung 9.14 Provision berechnen mit loop until-Schleife

```
Option Explicit
Sub provisionMitLoopUntil()

' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provisionMitLoopUntil

        dim umsatz
        dim verkaufsbetrag
        dim provisionInProzent As Double
        dim auszuzahlendeProvision As Double
        dim eingabe As String
```

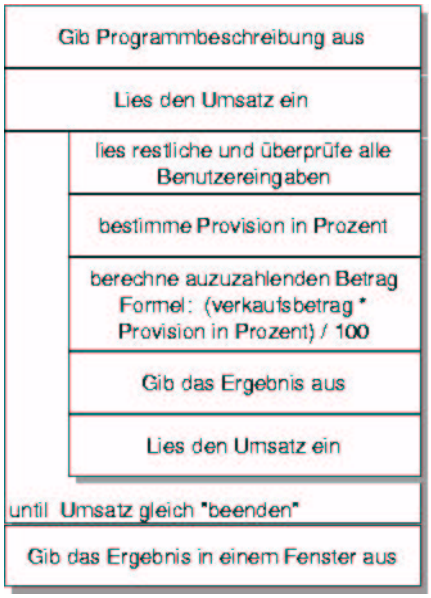

Abbildung 9.17 Struktogrammdarstellung zu "Provisionsberechnung mit loop until-Schleife"

```
' Umsatzgrenzen sind DM-Betraege

const umsatzGrenze1 As Double = 100000
const umsatzGrenze2 As Double = 500000
const umsatzGrenze3 As Double = 1000000

' Provisionen in Prozent

const provisionUmsatzGrenze1 As Double = 5
const provisionUmsatzGrenze2 As Double = 10
const provisionUmsatzGrenze3 As Double = 20

' Gib Programmbeschreibung aus

MsgBox("Geben Sie Umsatz und Verkaufsbetrag ein!" _
               & chr$(13) & "Das Programm berechnet die" _
               & " Provision des Vermittlers! Sie beenden das" _
               & " Programm durch die Eingabe von: beenden!")



' Lies Umsatz ein

      eingabe = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")

      Do

            ' Lies und ueberpruefe die Benutzereingaben

            if Not IsNumeric (eingabe)  Then
                  MsgBox ("Umsatz muß eine Zahl sein!")
                  Exit Sub
            End if
            umsatz = CDbl(eingabe)

            eingabe = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")
            if Not IsNumeric (eingabe)  Then
```

```
                MsgBox ("Verkaufsbetrag muß eine Zahl sein!")
                Exit Sub
        End if
        verkaufsbetrag = CDbl(eingabe)

        if verkaufsbetrag > umsatz Then
                MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
                Exit Sub
        End if

        ' bestimme Provision

        if umsatz >= umsatzGrenze3 Then
                provisionInProzent = provisionUmsatzGrenze3
        elseIf umsatz >= umsatzGrenze2 Then
                provisionInProzent = provisionUmsatzGrenze2
        elseIf umsatz >= umsatzGrenze1 Then
                provisionInProzent = provisionUmsatzGrenze1
        else
                provisionInProzent = 0
        End if

        ' Berechne die Provision

        auszuzahlendeProvision = (verkaufsbetrag * provisionInProzent)/100

        ' Gib das Ergebnis aus

        MsgBox ("Die Provision für dieses Geschäft ist: " _
                              & auszuzahlendeProvision & " DM")

        ' Lies Umsatz ein

        eingabe = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
    Loop until eingabe = "beenden"

End Sub
```

Pseudocode 9.18 Taschenrechner mit loop until-Schleife

```
    Gib Programmbeschreibung aus
    Lies ersten Operanden ein
    do
        Lies restliche und überprüfe alle Benutzereingaben
        führe Berechnung durch
        Gib das Ergebnis aus
        Lies ersten Operanden ein
    loop until erster Operand ist ungleich 0
```

Realisierung 9.15 Taschenrechner mit loop until-Schleife

```
Option Explicit
Sub taschenrechnerMitLoopUntil()
' Taschenrechnerprogramm
' Dateiname: taschenrechnerLoopUntil

dim ersterOperand As Double
dim zweiterOperand As Double
dim operator As String
dim eingabe as String
dim ergebnis As Double
```

```
' Gib Programmbeschreibung aus

       MsgBox("Geben Sie zwei Operanden und einen Operator ein!" _
                      & chr$(13) & "Das Programm verhält sich wie ein" _
                      & " Taschenrechner. Als Operatoren sind +, - / und *" _
                      & " zugelassen! Das Programm endet durch die Eingabe von " _
                       & chr$(13) & "0 als erstem Summanden!")

' Lies ersten Summanden ein
       eingabe = InputBox ("Geben Sie nun den ersten Operanden ein!")

       do

' Lies restliche und ueberpruefe alle Benutzereingaben
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Erster Operand muß eine Zahl sein!")
                     Exit Sub
              End if
              ersterOperand = CDbl(eingabe)

              operator = InputBox ("Geben Sie nun den Operator ein!")

              eingabe = InputBox ("Geben Sie nun den zweiten Operanden ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Zweiter Operand muß eine Zahl sein!")
                     Exit Sub
              End if
              zweiterOperand = CDbl(eingabe)

' Fuehre Berechnung durch

              Select Case operator
                     case "+"
                            ergebnis = ersterOperand + zweiterOperand
                     case "-"
                            ergebnis = ersterOperand - zweiterOperand
                     case "*"
                            ergebnis = ersterOperand * zweiterOperand
                     case "/"
                            if (zweiterOperand = 0) Then
                                   MsgBox ("Der Nenner ist 0!")
                                   Exit Sub
                            End if
                            ergebnis = ersterOperand / zweiterOperand
                     case else
                            MsgBox ("Der eingegebene Operator wird nicht " _
                                          & "unterstützt!")
                            Exit Sub
              End Select


' Gib das Ergebnis aus

              MsgBox (" " & ersterOperand & " "  & operator & " " _
                            & zweiterOperand & " = " & ergebnis)

' Lies ersten Operanden ein

              eingabe = InputBox ("Geben Sie nun den ersten Operanden ein!")

       loop until eingabe = "0"
End Sub
```

9.2.3 Die for next-Schleife

In den bisher behandelten Beispielen wußten wir im Vorhinein nicht, wie oft die Schleife durchlaufen werden soll. Dies konnten die Benutzer mit ihren Eingaben entscheiden.

In einigen Fällen ist aber bekannt, wie häufig eine Schleife laufen soll. Solche Fälle sind zwar auch mit den bisher bekannten Konstruktionen abbildbar, jede Programmiersprache besitzt für solche Fälle jedoch ein eigenes Sprachkonstrukt. In VBA ist dies die `for next`-Schleife. Die `for next`-Schleife hat die Form:

```
for zaehler = Startwert To Endwert Step Schrittweite
       anweisung1
              :
              :
       anweisungN
next zaehler
```

Die `for next`-Schleife wird durch das Schlüsselwort `for` eingeleitet. Vor dem ersten Durchlauf der Schleife wird der Wert der Variable *zaehler* auf Startwert gesetzt (Initialisierung). *zaehler* muß eine numerische Variable sein. Dann wird verglichen, ob der Wert der Variablen zaehler kleiner oder gleich dem Endwert ist.

Ist dies der Fall werden die Anweisungen in der Schleife durchgeführt. Wird die Anweisung `next zaehler` erreicht, erhöht sich der Wert der Variable *zaehler* um die Schrittweite.

Die Angabe der Schrittweite ist optional. Ist keine Schrittweite angegeben, wird der Wert der Zählvariablen um eins erhöht.

Als nächstes kehrt die Schleife zur `for`-Anweisung zurück. Die Schleife überprüft, ob der neue Wert der Variablen *zaehler* immer kleiner oder gleich dem Endwert ist. Ist dies der Fall, wird die Schleife erneut durchlaufen. Bei Erreichen der Anweisung `next zaehler` wird die Zählvariable wieder um die Schrittweite erhöht und die Schleife kehrt zur `for`-Anweisung zurück.

Hier wird wieder überprüft, ob der neue Wert der Variable *zaehler* kleiner oder gleich dem Endwert ist. Diese Vorgänge finden so lange statt, bis *zaehler* größer als der Endwert ist. Dann wird die Schleife verlassen. Dies bedeutet, daß VBA mit der auf `next zaehler` folgenden Anweisung fortsetzt.

Wir veranschaulichen uns dies an einem Programm zur Fakultätenberechnung:

Problemstellung 9.8 Fakultätenprogramm

Wir wollen ein Programm schreiben, das die Fakultät einer vom Benutzer in einem Fenster eingegebenen Zahl berechnet. Das Programm soll das Ergebnis der Berechnung in einem Bildschirmfenster ausgeben. Zu Anfang soll das Programm eine kurze Betriebsanleitung ausgeben.

Pseudocode 9.19 Das Fakultätenprogramm

```
Gib Programmbeschreibung aus
Lies die Zahl, deren Fakultät berechnet werden soll, ein
```

```
initialisiere fakultät mit 1
for i=1 To eingegebene Zahl
       fakultät = fakultät * i
next i
Gib Ergebnis in einem Fenster aus
```

Zunächst sehen wir, daß wir eine for-Schleife im Pseudocode einfach durch die Verwendung der Schlüsselworte `for next` kenntlich machen. Als Schleifenvariable wählen wir *i*. Beachten Sie auch wieder die Einrückung. zeigt das Struktogramm zum Fakultätenprogramm.

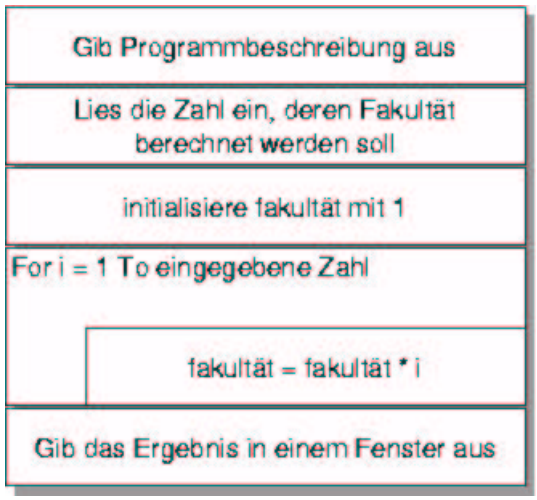

Abbildung 9.18 Struktogrammdarstellung zumFakultätenprogramm

Gehen wir den Pseudocode im Einzelnen durch:

Die ersten beiden Zeilen des Pseudocode sind wohlbekannt und klar. Nehmen wir für die weitere Diskussion an, daß der Benutzer als Zahl, bis zu der die Fakultät berechnet werden soll, 3 eingegeben hat.

Vor der Schleife wird unsere Ergebnisvariable mit 1 initialisiert:

```
initialisiere fakultät mit 1
```

Dann folgt eine Schleife. Die Schleife beginnt mit dem Befehl:

```
for i = 1 to eingegebene Zahl
```

Hier wird die Variable *i* mit dem Wert 1 initialisiert. Danach wird überprüft, ob der Wert von *i* kleiner gleich dem Wert der eingegebenen Zahl ist. Wenn der Benutzer drei eingegeben hat, ist dies der Fall. Daher werden nun die Anweisungen der Schleife abgearbeitet. Dies sind alle Anweisungen, die sich zwischen

```
for i = 1 to eingegebene Zahl
```

und

```
next i
```

befinden. In unserem Beispiel ist dies nur:

```
fakultät = fakultät * i
```

Da die Variablen *fakultät* und *i* mit dem Wert 1 initialisiert wurde, ergibt *fakultät = fakultät * i* (1 * 1) den Wert 1. Dieser neue Wert wird *fakultät* zugewiesen. *fakultät* hat also weiterhin den Wert 1. Durch die Anweisung

```
next i
```

wird der Wert von *i* um 1 erhöht. Da *i* vorher den Wert 1 hatte und 1 plus 1 zwei ergibt, hat *i* danach den Wert zwei. Des Weiteren veranlaßt

```
next i
```

VBA zu der Anweisung

```
for i = 1 to eingegebene Zahl
```

zurückzugehen. Da diese Anweisung nun zum zweiten Mal durchgeführt wird, wird die Initialisierung von *i* nicht mehr durchgeführt. Dies passiert nur bei der ersten Durchführung der for-Anweisung. *i* behält also den Wert 2. `for` überprüft nur , ob der Wert von *i* immer noch kleiner oder gleich dem Wert von eingegebene Zahl ist. Da *i* 2 ist und eingegebene Zahl 3, ist dies der Fall. Die Anweisung in der Schleife wird durchgeführt. Die Anweisung der Schleife war:

```
fakultät = fakultät * i
```

Da *fakultät* nach dem letzten Schleifendurchlauf den Wert 1 zugewiesen erhielt und *i* zur Zeit den Wert 2 hat ergibt *fakultät* * *i* (1 *2) nun 2. Dieser neue Wert wird *fakultät* zugewiesen. Durch die Zeile:

```
next i
```

wird *i* um 1 erhöht (i ist jetzt 3) und das Programm kehrt zur `for`-Anweisung zurück. Hier wird erneut überprüft, ob der Wert von *i* immer noch kleiner oder gleich dem Wert von eingegebene Zahl ist. Dies ist der Fall (*i* ist 3, eingegebene Zahl immer noch), also wird wieder die Anweisung in der Schleife durchgeführt. *fakultät* war nach dem letzten Schleifendurchlauf 2, *i* ist zur Zeit 3, also ergibt fakultät * i (2 *3) 6. Dieser neue Wert wird *fakultät* zugewiesen.

```
next i
```

erhöht *i* wieder um 1, (i ist jetzt 4) und das Programm kehrt zur `for`-Anweisung zurück. Hier wird erneut überprüft, ob der Wert von *i* immer noch kleiner oder gleich dem Wert von eingegebene Zahl ist. Dies ist diesmal nicht der Fall (*i* ist 4, eingegebene Zahl immer noch 3). Dies veranlaßt das Programmm nun, die Schleife zu beenden. Eine Schleife zu beenden bedeutet, mit der ersten Anweisung hinter der Schleife fortzufahren. Dies ist:

```
Gib Ergebnis in einem Fenster aus
```

Realisierung 9.16 Das Fakultätenprogramm

```
Option Explicit
Sub fakultaeten()

' Programm bildet die Fakultaet einer einzugebenden Zahl
' Dateiname: fakultaeten

       dim fakultaet As Integer
```

```
       dim bisZu As Integer
       dim i As Integer
' Gib Programmbeschreibung aus
       MsgBox ("Berechnung der Fakultät der Zahl, die Sie eingeben!")

'Lies die Zahl, deren Fakultät berechnet werden soll, ei

       bisZu = InputBox("Geben Sie nun die Zahl ein, deren Fakultät " & _
                                          "berechnet werden soll!")

' initialisiere fakultaet mit 1

       fakultaet = 1

' Berechnung der fakultaet

       for i = 1 to bisZu
              fakultaet = fakultaet * i
       next i

' Gib Ergebnis in einem Fenster aus

       MsgBox ("  " & bisZu & "! = "  & fakultaet)
End Sub
```

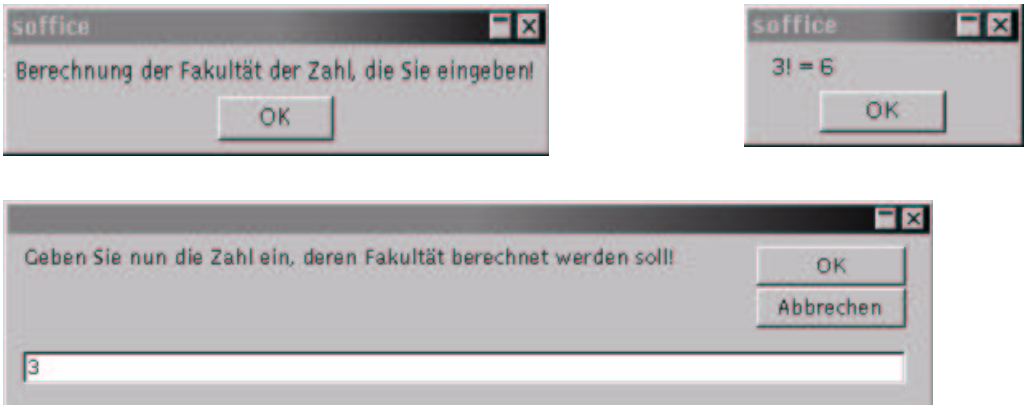

Abbildung 9.19 Bildschirmein- und ausgabe von Realisierung 9.16

10 Ein Anwendungsbeispiel (Zinsberechnung)

In diesem Kapitel wollen wir das bisher Gelernte an einem weiteren Anwendungsbeispiel wiederholen und vertiefen.

Problemstellung 10.1 Das Zinsbeispiel

Für eine Bank soll ein VBA-Programm zur Zinsberechnung erstellt werden. Dies soll an Kunden der Bank verteilt und von diesen dann genutzt werden können. Zweck des Programms ist es, die monatliche Belastung des Kunden beim Eigenheim- oder Eigentumswohnungskauf bei vom Kunden einzugegebendem Eigenkapital und Preis der Immobilie zu errechnen und auszugeben.

Dabei sind folgende Randbedingungen zu beachten:

- Wenn die Eigenkapitalquote 30 % unterschreitet, soll keine Berechnung durchgeführt werden. Das Programm soll anstelle dessen einfach ausgeben, daß bei einer so geringen Eigenkapitalquote keine Finanzierung durchgeführt werden kann.
- Die Kunden der Bank sind in 5 Klassen unterteilt:
  - o Klasse 1-Kunden machen mit der Bank einen Umsatz von über 100000 DM. Sie erhalten z.Zt. einen Zinssatz von 5,5 %.
  - o Klasse 2-Kunden machen mit der Bank einen Umsatz von über 200000 DM. Sie erhalten z.Zt. einen Zinssatz von 5,3 %.
  - o Klasse 3-Kunden machen mit der Bank einen Umsatz von über 250000 DM. Sie erhalten z.Zt. einen Zinssatz von 5,2 %.
  - o Klasse 4-Kunden machen mit der Bank einen Umsatz von über 300000 DM. Sie erhalten z.Zt. einen Zinssatz von 5,0 %.
  - o Klasse 5-Kunden machen mit der Bank einen Umsatz von über 500000 DM. Sie erhalten z.Zt. einen Zinssatz von 4,5 %.

  Die Kunden geben zusätzlich zu Eigenkapital und Immoblienpreis ihre Klasse ein z.B. 1 für Klasse 1, 2 für Klasse 2 etc. (die Klasse wird allen Kunden, die das Programm nutzen dürfen, mitgeteilt). Die Berechnung wird dann mit den oben angeführten Zinssätzen durchgeführt. Gibt der Kunde eine Zahl ein, für die es keine Klasse gibt (z.B. 6) erfolgt eine Fehlerausgabe. Die Zinssätze müssen also im Programm vorhanden sein. Sie sollen dort aber so codiert sein, daß sie leicht änderbar sind, da sich auch die Zinssätze relativ häufig ändern.
- Die Tilgung beträgt immer ein Prozent.
- Die Eigenkapitalquote soll in jedem Fall ausgegeben werden.
- Das Programm soll so lange laufen, bis der Kunde / die Kundin als Immobilienpreis "beenden" eingibt.
- Alle Eingaben sollen überprüft werden.

Zunächst müssen wir den Algorithmus zur Problemlösung finden und dokumentieren. Zur Dokumentation benutzen wir Pseudocode.

Pseudocode 10.1 Zinsbeispiel

```
Gib Programmbeschreibung aus
Lies den Immobilienpreis ein
do while Immobilienpreis ungleich "beenden"
       lies restliche und überprüfe alle Benutzereingaben
       führe Berechnungen durch
       Gib Ergebnisse aus
       Lies den Immobilienpreis ein
Loop
```

"Gib Programmbeschreibung aus" ist eine einfache Anwendung einer `MsgBox`. "Lies den Immobilienpreis ein" ist eine `InputBox`. "Lies restliche und überprüfe alle Benutzereingaben" ist ebenfalls eine `InputBox` mit anschließender Überprüfung, ob das, was wir eingelesen haben, numerisch ist. "führe Berechnungen durch" hingegen ist schon schwieriger. Hier steht die gesamte Logik des Programms hinter. "führe Berechnungen durch" müssen wir also weiter verfeinern.

Pseudocode 10.2 Führe Berechnungen durch

```
berechne Eigenkapitalquote
if Eigenkapitalquote < 30 % then
       MsgBox mit Hinweis Eigenkapitalquote zu niedrig ausgeben
       verlasse das Programm
else
       select case Klasse
              case 1
                     monatliche Belastung berechnen mit Zinssatz 5,5
              case 2
                     monatliche Belastung berechnen mit Zinssatz 5,3
              case 3
                     monatliche Belastung berechnen mit Zinssatz 5,2
              case 4
                     monatliche Belastung berechnen mit Zinssatz 5,0
              case 5
                     monatliche Belastung berechnen mit Zinssatz 4,5
              case else
                     MsgBox mit Hinweis falsche Klasse ausgeben
                     verlasse das Programm
       end select
end if
```

In diesem Pseudocode sehen wir ein `if then else`- und ein `select case`-Konstrukt. Beides ist in VBA abbildbar. Bleiben noch "berechne Eigenkapitalquote" und "monatliche Belastung berechnen". Hinter diesen beiden Arbeitsschritten verbergen sich die[19] mathematischen Formeln der Aufgabenstellung. Wir müssen uns also überlegen, wie man eine Eigenkapitalquote und eine monatliche Belastung ermittelt. Dies ist glücklicherweise nicht besonders schwierig. Wir beginnen mit der Eigenkapitalquote:

19.allerdings sehr einfachen

Pseudocode 10.3 Eigenkapitalquote berechnen

```
eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
```

Hierbei handelt es sich um eine Zuweisung und die Berechnung eines Ausdrucks vermittels der Operatoren / und *. Dies ist sofort in VBA abbildbar.

Pseudocode 10.4 Monatliche Belastung berechnen mit Zinssatz zins

```
aufzunehmenderBetrag = immobilienPreis - eigenkapital
jahresBelastung = (aufzunehmenderBetrag / 100)* (zins + 1)
monatlicheBelastung = jahresBelastung / 12
```

Auch hierbei handelt es sich um Zuweisungen und um einfache Ausdrücke, die direkt in VBA abbildbar sind. "führe Berechnung durch" ist damit hinreichend dokumentiert. Vom Pseudocode der obersten Ebene bleibt also noch "Gib Ergebnisse aus". Dies können aber wieder mit einfachen `MsgBox` lösen.

Wir können daher zur Realsierung schreiten:

Realisierung 10.1 Zinsbeispiel

```
Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: zinsberechnung

      const zinsKlasse1 As Double = 5.5
      const zinsKlasse2 As Double = 5.3
      const zinsKlasse3 As Double = 5.2
      const zinsKlasse4 As Double = 5.0
      const zinsKlasse5 As Double = 4.5
      const tilgung As Double = 1

      dim eingabe As String
      dim eigenkapital As Double
      dim immobilienPreis As Double
      dim aufzunehmenderBetrag As Double
      dim eigenkapitalquote As Double
      dim jahresBelastung As Double
      dim monatlicheBelastung As Double
      dim zinsKlasse As Integer

' Gib Programmbeschreibung aus

      MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                        & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                        & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                        & chr$(13) & " indem Sie beenden als Kaufpreis angeben")

' Lies den Immobilienpreis ein

      eingabe = InputBox ("Geben Sie jetzt den Kaufpreis ein!")

      do while eingabe <> "beenden"
' Lies restliche und ueberpruefe alle Benutzereingaben

            if Not IsNumeric (eingabe)  Then
                  MsgBox ("Kaufpreis muß eine Zahl sein!")
```

```
                Exit Sub
            End if
            immobilienPreis = CDbl(eingabe)

            eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
            if Not IsNumeric (eingabe)  Then
                MsgBox ("Eigenkapital muß eine Zahl sein!")
                Exit Sub
            End if
            eigenkapital = CDbl(eingabe)

            eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
            if Not IsNumeric (eingabe)  Then
                MsgBox ("Zinsklasse muß eine Zahl sein!")
                Exit Sub
            End if
            zinsKlasse = CInt(eingabe)

' führe Berechnungen durch
            ' berechne Eigenkapitalquote
            eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
            if eigenkapitalquote < 30 then
                MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                        "% ist zu niedrig!")
                exit sub
            else
                select case zinsKlasse
                    case 1
                        'Monatliche Belastung berechnen
                        aufzunehmenderBetrag = _
                            immobilienPreis - eigenkapital
                        jahresBelastung = _
                        (aufzunehmenderBetrag/100)* (zinsKlasse1 + tilgung)
                        monatlicheBelastung = jahresBelastung / 12
                    case 2
                        'Monatliche Belastung berechnen
                        aufzunehmenderBetrag = _
                            immobilienPreis - eigenkapital
                        jahresBelastung = _
                        (aufzunehmenderBetrag/100)* (zinsKlasse2 + tilgung)
                        monatlicheBelastung = jahresBelastung / 12
                    case 3
                        'Monatliche Belastung berechnen
                        aufzunehmenderBetrag = _
                            immobilienPreis - eigenkapital
                        jahresBelastung = _
                        (aufzunehmenderBetrag/100)* (zinsKlasse3 + tilgung)
                        monatlicheBelastung = jahresBelastung / 12
                    case 4
                        'Monatliche Belastung berechnen
                        aufzunehmenderBetrag = _
                            immobilienPreis - eigenkapital
                        jahresBelastung = _
                        (aufzunehmenderBetrag/100)* (zinsKlasse4 + tilgung)
                        monatlicheBelastung = jahresBelastung / 12
                    case 5
                        'Monatliche Belastung berechnen
                        aufzunehmenderBetrag = _
                            immobilienPreis - eigenkapital
                        jahresBelastung = _
                        (aufzunehmenderBetrag/100)* (zinsKlasse5 + tilgung)
                        monatlicheBelastung = jahresBelastung / 12
                    case else
```

```
                                    MsgBox ("Sie haben eine falsche Zinsklasse" & _
                                                 " eingegeben! Zinsklasse muß" & _
                                                 " kleiner gleich 5 sein!")
                                    exit sub
                        end select
                end if

' Gib Ergebnisse aus

                MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

' Lies den Immobilienpreis ein

                eingabe = InputBox ("Geben Sie jetzt den Kaufpreis ein!")

        loop

end sub
```

Abbildung 10.1 zeigt beispielhafte Durchläufe durch Realisierung 10.1.

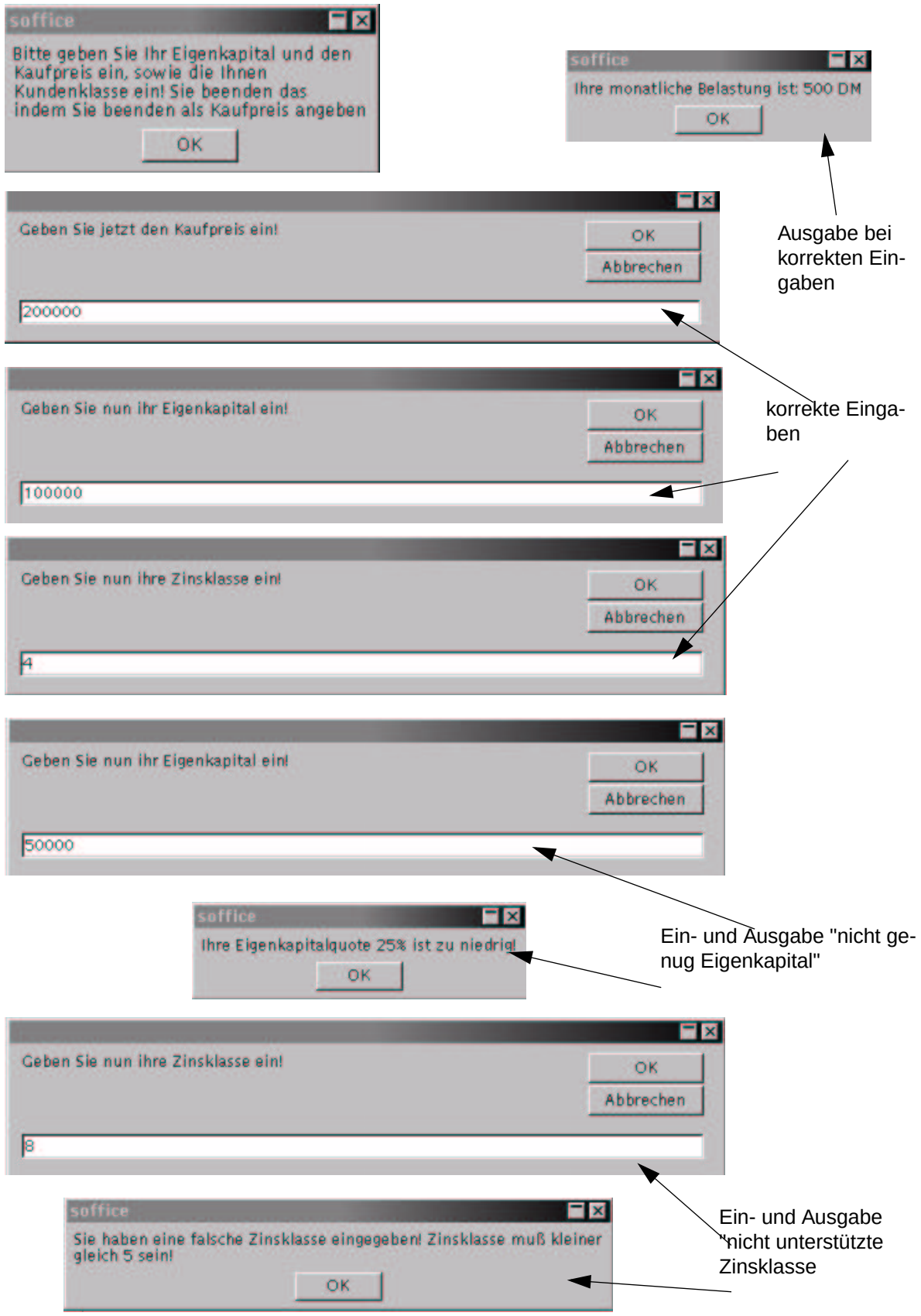

Abbildung 10.1 Realisierung 10.1 mit korrekte Eingaben, mit nicht genügender Eigenkapitalquote und nicht unterstützter Zinsklasse

11 Prozeduren und (Funktionen

11.1 Motivation:

Wenn wir uns die Problemlösung und die Vorgehensweise zur Problemlösung zur Problemstellung des vorangegangenen Kapitels noch einmal betrachten, könnten uns (leider) noch einige Schwachstellen auffallen:

- Unser Pseudocode ist strukturiert und in mehrere Ebenen zerlegt (vgl. Pfeil in Abbildung 11.1). Diese Struktur findet sich in der VBA-Implementierung in keiner Weise wieder.

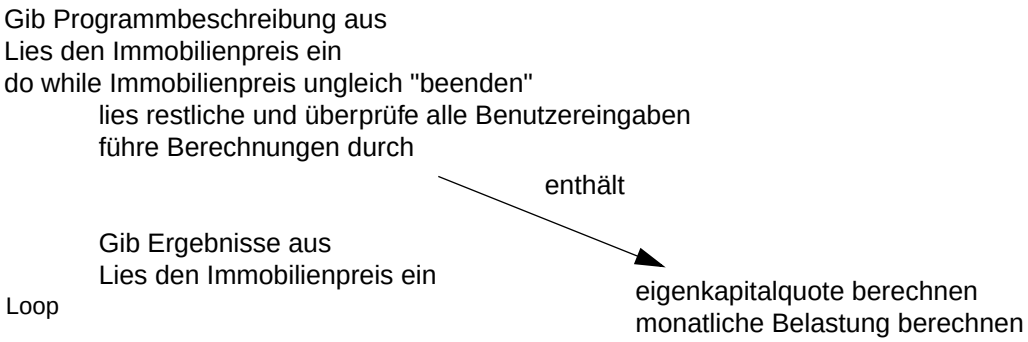

Abbildung 11.1 Strukturierung des Pseudocodes

- Wir haben während der Problemanalyse- und -lösung die Methode der schrittweisen Verfeinerung gewählt. Jede “Anweisung” der in Abbildung 11.1 dargestellten Strukturierung besteht ihrerseits in der VBA-Realisierung aus mehreren VBA-Anweisungen (vgl. Abbildung 11.2). In der VBA-Realisierung ist dies (nur) durch Kommentare kenntlich gemacht. Diese Kommentare finden sich (weil sie ja auch dorthin gehören) über den zugehörigen VBA-Codezeilen. Die Übersichtlichkeit, die in Abbildung 11.1 gegeben ist, geht so verloren.

- Im Pseudocode stehen in der `do while`-Schleife 4 “Anweisungen”. Anfang und Ende der Schleife sind also leicht überblickbar. In der Realisierung liegen zwischen Beginn und Ende der `do while`-Schleife jedoch ca. 80 Zeilen Programmcode. Der Zusammenhang zwischen Anfang und Ende ist also nur noch schwer zu identifizieren.

- Anweisungsfolgen kommen im Programm mehrfach vor. Betrachten wir die Anweisungsfolge, die zu "monatliche Belastung berechnen" gehört (vgl. Abbildung 11.2). Bis auf den zu zahlenden Zins (im Programm zinsKlasse1 bis zinsKlasse5, in Abbildung 11.2 durch zins repräsentiert) erscheint dieser Programmblock in der Realisierung 5-mal. Dies erhöht den Umfang des Programms und macht Programmänderungen und Fehlerbehebung aufwendig und schwierig. Stellen wir uns vor, wir hätten bei der Berechnung der monatlichen Belastung einen Fehler gemacht und (statt Abbildung 11.2) folgenden Pseudocode spezifiziert (Multiplizieren und Dividieren bei der Berechnung der Jahresbelastung vertauscht):

```
aufzunehmenderBetrag = immobilienPreis - eigenkapital
jahresBelastung = (aufzunehmenderBetrag/100)* (zins + 1)
monatlicheBelastung = jahresBelastung / 12
```

Abbildung 11.2 Anweisungsfolge in “monatliche Belastung berechnen”

Pseudocode 11.1 monatliche Belastung berechnen mit Zinssatz zins **(völlig falsch)**

```
aufzunehmenderBetrag = immobilienPreis - eigenkapital
jahresBelastung = (aufzunehmenderBetrag*100)/ (zins + 1)
monatlicheBelastung = jahresBelastung / 12
```

Fällt dieser Fehler vor der Umsetzung des Pseudocodes in VBA auf, muß nur eine Zeile geändert werden (die, wo die Jahresbelastung berechnet wird). Fällt der Fehler allerdings erst während der Programmtests auf[20], sind aus dem einen Fehler 5 Fehler geworden. Demzufolge muß der Fehler auch 5-mal korrigiert werden.

Neben "monatliche Belastung berechnen" kommt "Lies den Immobilienpreis ein" in unveränderter Form zweimal im VBA-Programm vor. Dies ist allerdings nicht so hoch dramatisch, da "Lies den Immobilienpreis ein" nur aus einer Anweisung besteht.

- Das VBA Programm an sich ist recht groß geworden. Die schrittweise Verfeinerung, mit deren Hilfe wir den Pseudocode erstellt haben, ließ sich im VBA Programm nicht umsetzen.

Uns fehlt also ein Hilfsmittel, um die Methode der schrittweisen Verfeinerung auch während der Implementierungsphase durchzuhalten. Schön wäre es, wenn unser Programm ähnlich wie der Pseudocode in Abbildung 11.1 aussehen würde und dann andere kleine Programme nutzen könnte, die die “Hauptfunktionen”, die in Abbildung 11.1 sichtbar sind, implementieren.

VBA bietet uns, wie (fast) alle Programmiersprachen, diese Möglichkeit. Wir können Anweisungsfolgen zu Prozeduren zusammenfassen und dieser Prozedur einen Namen geben. Für Prozedurnamen gelten die im Kapitel für Variablennamen dargestellten Regeln. Prozeduren beginnen mit dem Schlüsselwort `sub`, dann folgt der Name der Prozedur, dann eventuelle Übergabeparameter[21].

11.2 Eine erste einfache Prozedur

Ich werde dies zunächst an einem Beispiel verdeutlichen. Betrachten wir erneut das Zinsprogramm des vorangegangenen Kapitels. Wir wollen "Gib Programmbeschrei-

20.Die Formel führt zu gigantischen jährlichen und damit auch monatlichen Belastungen.
21.Übergabeparameter werden später in diesem Kapitel behandelt.

bung aus" als erste eigene Prozedur realisieren. Unsere erste Prozedur ist allerdings noch nicht so besonders sinnvoll, da sie nur aus einer Anweisung bestehen wird.

Realisierung 11.1 Zinsbeispiel mit Prozedur für "Gib Programmbeschreibung aus"

```
Option Explicit
Sub  Gib_Programmbeschreibung_aus()
       MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                             & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                             & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                             & chr$(13) & " indem Sie beenden als Kaufpreis angeben")
End Sub

Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen1

       const zinsKlasse1 As Double = 5.5
       const zinsKlasse2 As Double = 5.3
       const zinsKlasse3 As Double = 5.2
       const zinsKlasse4 As Double = 5.0
       const zinsKlasse5 As Double = 4.5
       const tilgung As Double = 1

       dim eingabe As String
       dim eigenkapital As Double
       dim immobilienPreis As Double
       dim aufzunehmenderBetrag As Double
       dim eigenkapitalquote As Double
       dim jahresBelastung As Double
       dim monatlicheBelastung As Double
       dim zinsKlasse As Integer

       Gib_Programmbeschreibung_aus


' Lies den Immobilienpreis ein

       eingabe = InputBox ("Geben Sie jetzt den Kaufpreis ein!")

       do while eingabe <> "beenden"
' Lies restliche und ueberpruefe alle Benutzereingaben

              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Kaufpreis muß eine Zahl sein!")
                     Exit Sub
              End if
              immobilienPreis = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Eigenkapital muß eine Zahl sein!")
                     Exit Sub
              End if
              eigenkapital = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Zinsklasse muß eine Zahl sein!")
                     Exit Sub
              End if
```

```
              zinsKlasse = CInt(eingabe)

' führe Berechnungen durch
              ' berechne Eigenkapitalquote
              eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
              if eigenkapitalquote < 30 then
                     MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                                   "% ist zu niedrig!")
                     exit sub
              else
                     select case zinsKlasse
                            case 1
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse1 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case 2
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse2 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case 3
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse3 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case 4
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse4 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case 5
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse5 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case else
                                   MsgBox ("Sie haben eine falsche Zinsklasse" & _
                                                 " eingegeben! Zinsklasse muß" & _
                                                 " kleiner gleich 5 sein!")
                                   exit sub
                     end select
              end if

' Gib Ergebnisse aus

              MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

' Lies den Immobilienpreis ein

              eingabe = InputBox ("Geben Sie jetzt den Kaufpreis ein!")

       loop
```

```
end sub
```

Die Prozedur wird wie unser normales Programm eingeleitet:

```
Sub  Gib_Programmbeschreibung_aus()
```

Dies nennt man die Deklaration der Prozedur. Bei der Deklaration wird ein Name für die Prozedur vergeben (hier `Gib_Programmbeschreibung_aus`) .

Danach folgen die Anweisungen der Prozedur. Die Anweisungen werden wie immer in VBA von `sub` und `end sub` eingeschlossen.

Der Aufruf der Prozedur im Hauptprogramm erfolgt über den Namen der Prozedur.

```
Gib_Programmbeschreibung_aus
```

Das Arbeiten mit Prozeduren erfolgt also in zwei Schritten. Zunächst wird die Prozedur deklariert und implementiert. Unter Deklarieren verstehen wir der Prozedur einen Namen geben und das Schlüsselwort `sub` gefolgt vom Prozedurnamen gefolgt von runden Klammern aufzuführen[22].

Unter Implementieren verstehen wir (zunächst) das Aufführen der Anweisungen der Prozedur zwischen `sub` und `end sub`[23].

Deklaration und Implementierung einer Prozedur erfolgen genau einmal (pro Prozedur) in einem Programm.

Der zweite Schritt ist der Aufruf der Prozedur. Der Aufruf erfolgt, wie wir schon gesehen haben, über den Namen der Prozedur. Beim Aufruf der Prozedur dürfen die runden Klammern an den Prozedurnamen nicht angeschlossen werden, bei der Deklaration sind sie notwendig (Excel)[24]. StarCalc erlaubt die runden Klammern auch beim Aufruf.

Eine Prozedur kann in einem Programm beliebig oft aufgerufen werden.

11.3 Prozeduren mit Übergabeparametern (Referenzparameter) und Fehlerweiterleitung an das Hauptprogramm

Nachdem die Programmierung unserer ersten Prozedur so überaus erfolgreich war, wollen wir uns nun an eine zweite Prozedur wagen. Das Einlesen des Immobilienpreises soll in eine Prozedur ausgelagert werden. Auch diese Prozedur wird nur eine Anweisung enthalten. Sie wird zweimal von unserem Hauptprogramm aufgerufen.

Wir versuchen die Implementierung der zweiten Prozedur völlig analog zum ersten Beispiel:

22.Dieser Teil der Prozedur heißt auch Prozedurkopf und die runden Klammern sind Namensbestandteil.

23.Dieser Teil der Prozedur heißt auch Prozedurrumpf.

24.Ich weiß nicht, wie dei bei Microsoft darauf gekommen sind.

Realisierung 11.2 Zinsbeispiel mit Prozedur "Gib Programmbeschreibung aus" und "Lies den Immobilienpreis ein" **(leider falsch)**

```
Option Explicit
Sub  Gib_Programmbeschreibung_aus()
       MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                             & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                             & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                             & chr$(13) & " indem Sie beenden als Kaufpreis angeben")
End Sub

Sub Lies_den_Immobilienpreis_ein()
       eingabe = InputBox ("Geben Sie jetzt den Kaufpreis ein!")
end sub

Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen2

       const zinsKlasse1 As Double = 5.5
       const zinsKlasse2 As Double = 5.3
       const zinsKlasse3 As Double = 5.2
       const zinsKlasse4 As Double = 5.0
       const zinsKlasse5 As Double = 4.5
       const tilgung As Double = 1

       dim eingabe As String
       dim eigenkapital As Double
       dim immobilienPreis As Double
       dim aufzunehmenderBetrag As Double
       dim eigenkapitalquote As Double
       dim jahresBelastung As Double
       dim monatlicheBelastung As Double
       dim zinsKlasse As Integer

       Gib_Programmbeschreibung_aus
       Lies_den_Immobilienpreis_ein

       do while eingabe <> "beenden"
' Lies restliche und ueberpruefe alle Benutzereingaben

              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Kaufpreis muß eine Zahl sein!")
                     Exit Sub
              End if
              immobilienPreis = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Eigenkapital muß eine Zahl sein!")
                     Exit Sub
              End if
              eigenkapital = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Zinsklasse muß eine Zahl sein!")
                     Exit Sub
              End if
              zinsKlasse = CInt(eingabe)
```

```
' führe Berechnungen durch
              ' berechne Eigenkapitalquote
              eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
              if eigenkapitalquote < 30 then
                     MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                                   "% ist zu niedrig!")
                     exit sub
              else
                     select case zinsKlasse
                            case 1
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse1 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case 2
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse2 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case 3
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse3 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case 4
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse4 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case 5
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse5 + tilgung)
                                   monatlicheBelastung = jahresBelastung / 12
                            case else
                                   MsgBox ("Sie haben eine falsche Zinsklasse" & _
                                                 " eingegeben! Zinsklasse muß" & _
                                                 " kleiner gleich 5 sein!")
                                   exit sub
                     end select
              end if

' Gib Ergebnisse aus

              MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

              Lies_den_Immobilienpreis_ein

       loop

end sub
```

Dies Programm sieht eigentlich ganz gut aus und ist völlig analog zu Realisierung 11.1. Aber ach! Es funktioniert nicht. Die VBA-Laufzeitumgebung gibt eine Fehlermeldung aus (vgl. Abbildung 11.3). Sie beklagt sich über die nicht deklarierte Variable *eingabe*. Der Fehler passiert in der Implementierung (im Prozedurrumpf) der Prozedur `Lies_den_Immobilienpreis_ein()` und zwar genau da, wo die Variable *eingabe* eingelesen wird.

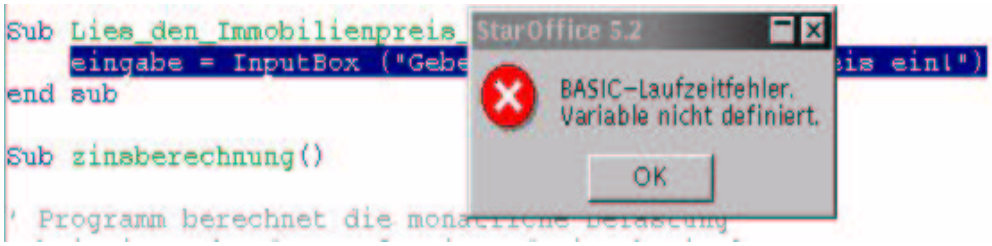

Abbildung 11.3 Fehlermeldung von Realisierung 11.2

Wie konnte das passieren? Des Rätsels Lösung ist: Es gibt doch einen Unterschied zwischen der Prozedur `Lies_den_Immobilienpreis_ein()` und der Prozedur `Gib_Programmbeschreibung_aus(). Gib_Programmbeschreibung_aus()` benutzt keine Variablen. Sie besteht nur aus einer `MsgBox` und in der `MsgBox` stehen String-Konstanten.

`Lies_den_Immobilienpreis_ein()` hingegen benutzt Variablen. Der Zweck dieser Prozedur ist ja, den Immobilienpreis einzulesen. Wenn wir uns nun den Code in Realisierung 11.2 erneut anschauen, sehen wir, daß die Deklaration der Variablen *eingabe* in unserem Hauptprogramm erfolgt.

Keine Prozedur hat Zugriff auf irgendwelche Dinge, die im Hauptprogramm oder anderen Prozeduren deklariert sind[25]. Da auch Variablen irgendwelche Dinge sind, hat `Lies_den_Immobilienpreis_ein()` keinen Zugriff auf die im Hauptprogramm deklarierte Variablen *eingabe.*

Wir könnten uns nun überlegen, die Variable *eingabe* in der Prozedur `Lies_den_Immobilienpreis_ein` noch einmal zu deklarieren. Dies bedeutet, wir ändern den Code von `Lies_den_Immobilienpreis_ein` folgendermaßen:

Realisierung 11.3 `Lies_den_Immobilienpreis_ein` mit Deklaration der Variable *eingabe*

```
Sub Lies_den_Immobilienpreis_ein()
       Dim eingabe As String
       eingabe = InputBox ("Geben Sie jetzt den Kaufpreis ein!")
end sub
```

Aber auch hier erreichen wir nicht das gewünschte Verhalten. Abbildung 11.4 zeigt die Ausgabe der VBA-Laufzeitumgebung:

25.Das ist strenggenommen nicht ganz richtig, Prozeduren können ihre Variablen anderen Prozeduren "zur Verfügung stellen" außerdem gibt es globale Variablen, aber da kommen wir später zu.

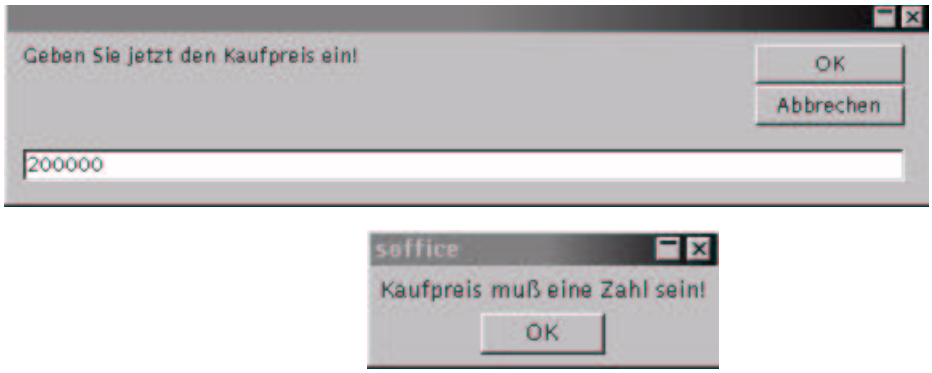

Abbildung 11.4 Falsches Verhalten von Realisierung 11.3

Woher kommt das nun? Unser Eingabefenster erscheint, wir geben auch eine Zahl ein, aber dennoch laufen wir in den `then`-Teil des folgenden Code-Stücks:

```
if Not IsNumeric (eingabe)  Then
       MsgBox ("Kaufpreis muß eine Zahl sein!")
       Exit Sub
End if
```

Dies liegt nun daran, daß die in Realisierung 11.3 deklarierte Variable *eingabe* eine lokale Variable der Prozedur `Lies_den_Immobilienpreis_ein()` ist. Sie hat nur den Namen gemeinsam mit der Variable *eingabe* des Hauptprogramms. Es handelt sich dennoch um zwei verschiedene Variablen.

Um hier dennoch zu einer Lösung[26] zu kommen, bietet VBA, wie (fast) jede andere Programmiersprache, die Möglichkeit, einer Prozedur Variablen zu übergeben.

Dies geschieht durch Nutzung der an den Prozedurnamen angefügten runden Klammern. In den Klammern werden die Übergabeparmeter (so heißen die zu übergebenden Variablen) aufgeführt. An die Übergabevariable wird wie gewohnt der Typ der Variablen mit `As` angeschlossen (vgl. Variablendeklaration). Wie immer in VBA kann ich den Variablentyp weglassen und erhalte dann eine `Variant`-Variable. Zwei Übergabevariablen werden durch ein , (Komma) getrennt.

Realisierung 11.4 Zinsbeispiel mit Prozedur "Gib Programmbeschreibung aus" und "Lies den Immobilienpreis ein" **(nun richtig)**

```
Option Explicit
Sub  Gib_Programmbeschreibung_aus()
       MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                              & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                              & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                              & chr$(13) & " indem Sie beenden als Kaufpreis angeben")
End Sub

Sub Lies_den_Immobilienpreis_ein(eingabe As String)
       eingabe = InputBox ("Geben Sie jetzt den Kaufpreis ein!")
end sub
```

26.Es gibt noch eine weitere Alternative, die ich später diskutieren werde.

```
Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen3

       const zinsKlasse1 As Double = 5.5
       const zinsKlasse2 As Double = 5.3
       const zinsKlasse3 As Double = 5.2
       const zinsKlasse4 As Double = 5.0
       const zinsKlasse5 As Double = 4.5
       const tilgung As Double = 1

       dim eingabe As String
       dim eigenkapital As Double
       dim immobilienPreis As Double
       dim aufzunehmenderBetrag As Double
       dim eigenkapitalquote As Double
       dim jahresBelastung As Double
       dim monatlicheBelastung As Double
       dim zinsKlasse As Integer

       Gib_Programmbeschreibung_aus
       Lies_den_Immobilienpreis_ein eingabe

       do while eingabe <> "beenden"
' Lies restliche und ueberpruefe alle Benutzereingaben

              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Kaufpreis muß eine Zahl sein!")
                     Exit Sub
              End if
              immobilienPreis = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Eigenkapital muß eine Zahl sein!")
                     Exit Sub
              End if
              eigenkapital = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Zinsklasse muß eine Zahl sein!")
                     Exit Sub
              End if
              zinsKlasse = CInt(eingabe)

' führe Berechnungen durch
              ' berechne Eigenkapitalquote
              eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
              if eigenkapitalquote < 30 then
                     MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                                   "% ist zu niedrig!")
                     exit sub
              else
                     select case zinsKlasse
                            case 1
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
                                   jahresBelastung = _
                                   (aufzunehmenderBetrag/100)* (zinsKlasse1 + tilgung)
```

```
                                monatlicheBelastung = jahresBelastung / 12
                        case 2
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse2 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 3
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse3 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 4
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse4 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 5
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse5 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case else
                                MsgBox ("Sie haben eine falsche Zinsklasse" & _
                                                " eingegeben! Zinsklasse muß" & _
                                                " kleiner gleich 5 sein!")
                                exit sub
                end select
        end if

' Gib Ergebnisse aus

        MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

        Lies_den_Immobilienpreis_ein eingabe

    loop

end sub
```

Bei der Deklaration der Prozedur sind also neben den Variablennamen die Variablentypen notwendig.

```
Sub Lies_den_Immobilienpreis_ein(eingabe As String)
```

Beim Prozeduraufruf werden nur noch die Variablennamen genutzt.

```
Lies_den_Immobilienpreis_ein eingabe
```

Beachten Sie, daß auch hier beim Aufruf die runden Klammern fehlen. Dies gilt nur für Excel, StarCalc gestattet runde Klammern.

Die Übergabeparameter im Prozedurkopf der Prozedurdeklaration heißen formale Parameter der Prozedur, die Variablen beim Aufruf heißen aktuelle Parameter (vgl. Abbildung 11.5.)

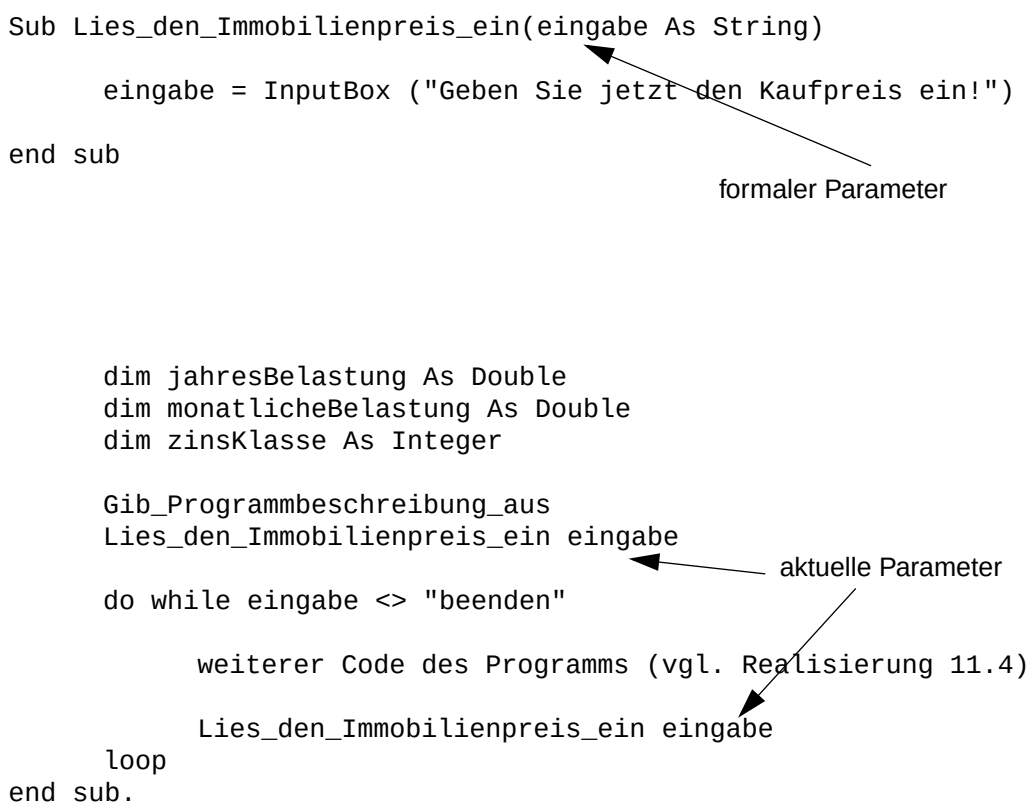

Abbildung 11.5 Unterschied formale Parameter <-> aktuelle Parameter

Merke: Formale Parameter und aktuelle Parameter müssen nicht den gleichen Namen erhalten. Die Übergabe erfolgt anhand der Reihenfolge. Der erste Aktualparameter wird an den ersten Formalparameter, der zweite Aktualparameter an den zweiten Formalparameter, usw. übergeben. Übereinstimmen müssen allerdings der Variablentyp jedes Parameterpaars und die Anzahl Parameter. Realisierung 11.2 ist also eine weitere funktionsfähige Implementierung des Zinsbeispiels.

Realisierung 11.5 Zinsbeispiel mit Prozedur "Gib Programmbeschreibung aus" und "Lies den Immobilienpreis ein" mit unterschiedlichen Namen für formale und aktuelle Parameter

```
Option Explicit
Sub  Gib_Programmbeschreibung_aus()
       MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                             & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                             & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                             & chr$(13) & " indem Sie beenden als Kaufpreis angeben")
End Sub

Sub Lies_den_Immobilienpreis_ein(immobilienPreis As String)
       immobilienPreis = InputBox ("Geben Sie jetzt den Kaufpreis ein!")
```

```
end sub

Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen4

       const zinsKlasse1 As Double = 5.5
       const zinsKlasse2 As Double = 5.3
       const zinsKlasse3 As Double = 5.2
       const zinsKlasse4 As Double = 5.0
       const zinsKlasse5 As Double = 4.5
       const tilgung As Double = 1

       dim eingabe As String
       dim eigenkapital As Double
       dim immobilienPreis As Double
       dim aufzunehmenderBetrag As Double
       dim eigenkapitalquote As Double
       dim jahresBelastung As Double
       dim monatlicheBelastung As Double
       dim zinsKlasse As Integer

       Gib_Programmbeschreibung_aus
       Lies_den_Immobilienpreis_ein eingabe

       do while eingabe <> "beenden"
' Lies restliche und ueberpruefe alle Benutzereingaben

              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Kaufpreis muß eine Zahl sein!")
                     Exit Sub
              End if
              immobilienPreis = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Eigenkapital muß eine Zahl sein!")
                     Exit Sub
              End if
              eigenkapital = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Zinsklasse muß eine Zahl sein!")
                     Exit Sub
              End if
              zinsKlasse = CInt(eingabe)

' führe Berechnungen durch
              ' berechne Eigenkapitalquote
              eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
              if eigenkapitalquote < 30 then
                     MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                                   "% ist zu niedrig!")
                     exit sub
              else
                     select case zinsKlasse
                            case 1
                                   'Monatliche Belastung berechnen
                                   aufzunehmenderBetrag = _
                                          immobilienPreis - eigenkapital
```

```
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse1 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 2
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse2 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 3
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse3 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 4
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse4 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 5
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse5 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case else
                                MsgBox ("Sie haben eine falsche Zinsklasse" & _
                                                " eingegeben! Zinsklasse muß" & _
                                                " kleiner gleich 5 sein!")
                                exit sub
                end select
        end if

' Gib Ergebnisse aus

        MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

        Lies_den_Immobilienpreis_ein eingabe

    loop

end sub
```

Zum Abschluß dieses Kapitels wollen wir das Einlesen und Überprüfen der Daten in eine Prozedur auslagern.

Hier stellt sich uns plötzlich ein weiteres Problem. Unsere Hauptprogramme sind streng genommen auch Unterprogramme. Sie werden von StarCalc, Excel oder welcher Anwendung auch immer, in der wir VBA programmieren, aufgerufen. Und das Kommando `exit sub`, das wir bisher immer benutzt haben, um unsere Anwendung bei Fehleingaben abzubrechen, bricht streng genommen nur das Unterprogramm ab, in dem es ausgeführt wurde. Wenn wir also in einem Unterprogramm unseres Hauptprogramms `exit sub` rufen, wird das Unterprogramm beendet und das Hauptpro-

gramm erhält die Kontrolle zurück. Es setzt also mit der auf den Unterprogrammaufruf folgenden Anweisung fort. Beispiel 11.1 illustriert dies:

Beispiel 11.1 Exit Sub in einem Unterprogramm

```
Option Explicit
sub verlassen(eingabe As Integer)
       MsgBox("Das Unterprogramm meldet sich!")
       if eingabe = 0 Then
              MsgBox("Das Unterprogramm wird verlassen!")
              exit sub
       end if
       MsgBox("Das Unterprogramm meldet sich wieder!")
       MsgBox("Das Unterprogramm meldet sich wieder!")
end sub

sub verlassenTest()
' Programm zu exit sub
' aus Unterprogrammen
' Dateiname: verlassenTest
       dim eingabe As Integer
       eingabe = InputBox("Was eingeben!")
       MsgBox("verlassenTest meldet sich!")
       verlassen eingabe
       MsgBox("verlassenTest meldet sich wieder!")
       MsgBox("verlassenTest meldet sich noch mal!")
end sub
```

Dies Programm erzeugt bei der Eingabe von 0 in Abbildung 11.6 dargestellte Ausgaben.

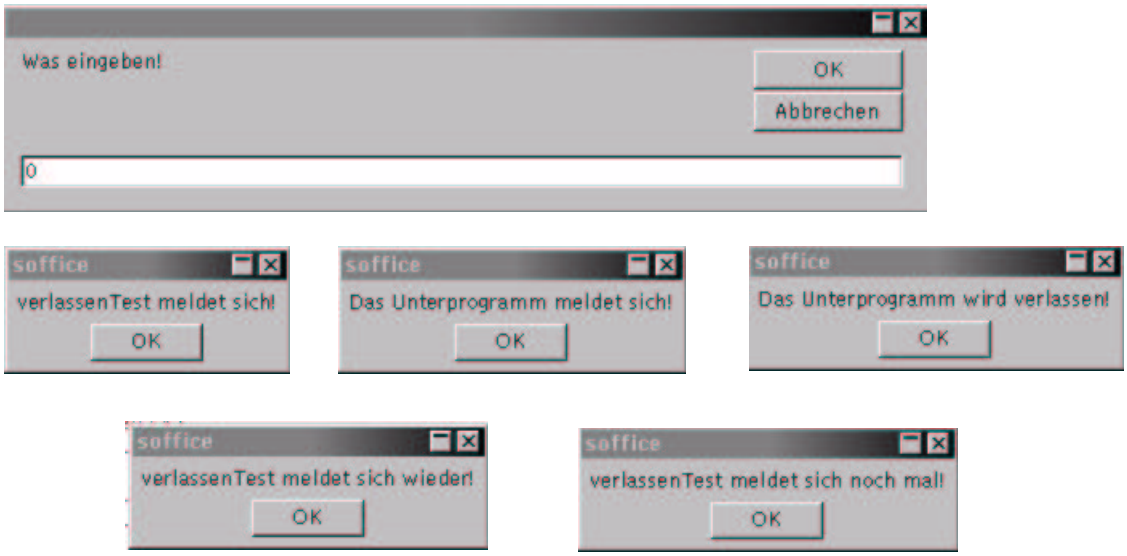

Abbildung 11.6 Ein- und Ausgaben von Beispiel 11.1

Dies ist jetzt nicht unbedingt das, was wir wollten. Durch das `exit sub` im `then` Teil von

```
       if eingabe = 0 Then
```

```
            MsgBox("Das Unterprogramm wird verlassen!")
            exit sub
      end if
```

wird das Unterprogramm offenbar verlassen. Die Anweisungen

```
      MsgBox("Das Unterprogramm meldet sich wieder!")
      MsgBox("Das Unterprogramm meldet sich wieder!")
```

werden offenbar nicht durchgeführt. Doch dummerweise erkennen wir an Abbildung 11.6, daß die Anweisungen des Hauptprogramms, die auf den Prozeduraufruf folgen, nämlich

```
      MsgBox("verlassenTest meldet sich wieder!")
      MsgBox("verlassenTest meldet sich noch mal!")
```

sehr wohl ausgeführt werden. Dies bedeutet: `exit sub` verläßt genau ein Unterprogramm. Das Hauptprogramm erhält die Kontrolle zurück und macht einfach weiter.

Damit haben wir ein Problem: Lagern wir die Kontrolle der Variablen in ein Unterprogramm aus, können wir nicht einfach mit `exit sub` das Unterprogramm verlassen und "gut ist". Dies bringt uns keinen Schritt weiter, weil dann das Hauptprogramm weiter und sogar nur Unsinn macht, weil keine Eingaben vorliegen.

Die Lösung dieser Problematik ist, dem Hauptprogramm mitzuteilen, daß ein Fehler aufgetreten ist. Dazu benutzen wir bool'sche Variablen. Das Hauptprogramm wertet die bool'sche Variable aus und kann sich anhand des Wertes der bool'schen Variablen entscheiden, ob es abbrechen oder mit der Ausführung fortfahren soll.

Ich zeige dies am folgenden Beispiel:

Beispiel 11.2 Fehlerweiterleitung aus einem Unterprogramm

```
Option Explicit
sub verlassen2(eingabe As Integer, fehler As Boolean)
      fehler = false
      MsgBox("Das Unterprogramm meldet sich!")
      if eingabe = 0 Then
            fehler = true
            MsgBox("Das Unterprogramm wird verlassen!")
            exit sub
      end if
      MsgBox("Das Unterprogramm meldet sich wieder!")
      MsgBox("Das Unterprogramm meldet sich wieder!")
end sub

sub verlassenTest2()
' Programm zu exit sub
' aus Unterprogrammen
' Dateiname: verlassenTest
      dim eingabe As Integer
      dim fehler As Boolean
      eingabe = InputBox("Was eingeben!")
      MsgBox("verlassenTest meldet sich!")
      verlassen2 eingabe, fehler
      if fehler Then
            MsgBox("Das Hauptprogramm wird verlassen!")
            exit sub
      end if
      MsgBox("verlassenTest meldet sich wieder!")
```

```
        MsgBox("verlassenTest meldet sich noch mal!")
end sub
```

Die Ausgaben können Sie Abbildung 11.7 entnehmen:

Abbildung 11.7 Ein- und Ausgaben von Beispiel 11.2

Wir deklarieren also eine zusätzliche bool'sche Variable im Hauptprogramm:

```
        dim fehler As Boolean
```

Diese wird ebenfalls dem Unterprogramm übergeben.

```
        verlassen2 eingabe, fehler
```

Natürlich mußte dafür zunächst die Deklaration der Prozedur verändert werden:

```
        sub verlassen2(eingabe As Integer, fehler As Boolean)
```

Im Falle einer Eingabe von "0" wird dann das Unterprogramm nicht nur verlassen. Darüber hinaus wird der Wert der Variablen *fehler* im Unterprogramm auf `true` gesetzt:

```
        if eingabe = 0 Then
                fehler = true
                MsgBox("Das Unterprogramm wird verlassen!")
                exit sub
        end if
```

Beachten Sie, daß die Variable *fehler* im Unterprogramm zunächst initialisiert wurde.

```
        fehler = false
```

Sonst wüßten wir nämlich nicht, welchen Wert das Unterprogramm zurückgeben würde, wenn die Bedingung zum Verzweigen in das `if`-Konstrukt nicht erfüllt wäre[27].

27.Überlegen Sie sich, warum es wenig Sinn macht, die Variable fehler im Hauptprogramm zu initialisieren.

Im Hauptprogramm prüfen wir dann, ob es im Unterprogramm zu einem Fehler gekommen ist. Ist dies der Fall, beenden wir das Programm.

```
if fehler Then
       MsgBox("Das Hauptprogramm wird verlassen!")
       exit sub
end if
```

Wenn wir dies beachten, können wir nun auch das Einlesen der weiteren Variablen und die Überprüfungen in eine eigene Prozedur auslagern. Wir werden völlig analog verfahren und eine Fehlervariable einführen, anhand derer wir dann entscheiden können, ob im Unterprogramm ein Fehler aufgetreten ist.

Ansonsten erfolgt dies vollständig analog zu Realisierung 11.2 . Wir haben fünf Übergabeparameter (*eingabe*, *immobilienPreis*, *eigenkapital*, *zinsKlasse* und *fehler*). Die ersten beiden Variablen sind vom Typ `double`, die dritte ist vom Typ `integer` und die letzte vom Typ `boolean`.

Realisierung 11.6 Zinsbeispiel mit Prozeduren für "Gib Programmbeschreibung aus", "Lies den Immobilienpreis ein" und "Lies restliche und überprüfe alle Benutzereingaben"

```
Option Explicit

Sub  Gib_Programmbeschreibung_aus()
       MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                          & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                          & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                          & chr$(13) & " indem Sie beenden als Kaufpreis angeben")
End Sub

Sub Lies_den_Immobilienpreis_ein(immobilienPreis As String)
       immobilienPreis = InputBox ("Geben Sie jetzt den Kaufpreis ein!")
end sub

sub  Lies_restliche_und_ueberpruefe_alle_Benutzereingaben(eingabe As String, _
                                        immobilienPreis As Double, _
                                        eigenKapital As Double, _
                                        zinsKlasse As Integer, _
                                        fehler As boolean)
              fehler = false
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Kaufpreis muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              immobilienPreis = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Eigenkapital muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              eigenkapital = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Zinsklasse muß eine Zahl sein!")
                     fehler = true
```

```
                Exit Sub
            End if
            zinsKlasse = CInt(eingabe)
end sub

Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen5

      const zinsKlasse1 As Double = 5.5
      const zinsKlasse2 As Double = 5.3
      const zinsKlasse3 As Double = 5.2
      const zinsKlasse4 As Double = 5.0
      const zinsKlasse5 As Double = 4.5
      const tilgung As Double = 1

      dim eingabe As String
      dim eigenkapital As Double
      dim immobilienPreis As Double
      dim aufzunehmenderBetrag As Double
      dim eigenkapitalquote As Double
      dim jahresBelastung As Double
      dim monatlicheBelastung As Double
      dim zinsKlasse As Integer
      dim fehler As boolean

      Gib_Programmbeschreibung_aus
      Lies_den_Immobilienpreis_ein eingabe

      do while eingabe <> "beenden"

            Lies_restliche_und_ueberpruefe_alle_Benutzereingaben eingabe, _
                                          immobilienPreis , _
                                          eigenkapital , _
                                          zinsKlasse, _
                                          fehler
            if fehler then
                  exit sub
            end if


' führe Berechnungen durch
            ' berechne Eigenkapitalquote
            eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
            if eigenkapitalquote < 30 then
                  MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                              "% ist zu niedrig!")
                  exit sub
            else
                  select case zinsKlasse
                        case 1
                              'Monatliche Belastung berechnen
                              aufzunehmenderBetrag = _
                                    immobilienPreis - eigenkapital
                              jahresBelastung = _
                              (aufzunehmenderBetrag/100)* (zinsKlasse1 + tilgung)
                              monatlicheBelastung = jahresBelastung / 12
                        case 2
                              'Monatliche Belastung berechnen
                              aufzunehmenderBetrag = _
                                    immobilienPreis - eigenkapital
```

```
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse2 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 3
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse3 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 4
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse4 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case 5
                                'Monatliche Belastung berechnen
                                aufzunehmenderBetrag = _
                                        immobilienPreis - eigenkapital
                                jahresBelastung = _
                                (aufzunehmenderBetrag/100)* (zinsKlasse5 + tilgung)
                                monatlicheBelastung = jahresBelastung / 12
                        case else
                                MsgBox ("Sie haben eine falsche Zinsklasse" & _
                                        " eingegeben! Zinsklasse muß" & _
                                        " kleiner gleich 5 sein!")
                                exit sub
                end select
        end if

' Gib Ergebnisse aus

        MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

        Lies_den_Immobilienpreis_ein eingabe

    loop

end sub
```

Bei dieser Realsierung müssen wir, wie immer, wenn wir mit Prozeduren und Übergabeparametern arbeiten, strikt auf die Reihenfolge der Aktualparameter achten.

Durch die Prozedurdeklartion

```
sub Lies_restliche_und_ueberpruefe_alle_Benutzereingaben(eingabe As String, _
                                        immobilienPreis As Double, _
                                        eigenKapital As Double, _
                                        zinsKlasse As Integer _
                                        fehler As Boolean)
```

ist die Reihenfolge der Variablen im Prozeduraufruf festgelegt.

Der Aufruf muß auf folgende Art geschehen:

```
    Lies_restliche_und_ueberpruefe_alle_Benutzereingaben eingabe, _
                                            immobilienPreis , _
                                            eigenKapital , _
                                            zinsKlasse _
```

```
                                                fehler
```

Sollten wir folgenden Fehler einbauen

```
        Lies_restliche_und_ueberpruefe_alle_Benutzereingaben eingabe, _
                                                eigenKapital, _
                                                immobilienPreis, _
                                                zinsKlasse
                                                fehler
```

würde *eigenkapital* auf *immobilienPreis* übergeben und umgekehrt. Die Folge wäre eine negative monatliche Belastung (die Bank muß Geld auszahlen). **Die Variablennamen spielen bei der Übergabe keine Rolle, nur die Reihenfolge zählt.**

11.4 Prozeduren mit Übergabeparametern (Wertparameter)

Schauen wir uns die Übergabeparameter unserer Prozedur `Lies_restliche_und_ueberpruefe_alle_Benutzereingaben()` noch einmal genau an.

Wir können hier 2 unterschiedliche Nutzungen von Übergabeparametern feststellen:

- *immobilienPreis*, *eigenKapital, fehler* und *zinsKlasse* werden in der Prozedur geändert. Das Hauptprogramm benötigt die geänderten Werte, um weitermachen zu können.
- *eingabe* wird in der Prozedur ebenfalls geändert. Allerdings benötigt das Hauptprogramm den geänderten Wert nicht mehr. Es ist also eigentlich unsinnig, den geänderten Wert an das Hauptprogramm zurückzugeben.

Um solche Dinge kenntlich zu machen, kennt VBA zwei Mechanismen der Variablenübergabe. Beim ersten Mechanismus erhält die Prozedur das Recht, die Aktualparameter zu ändern. Dies ist zum Beispiel beim Einlesen von Daten oder bei Initialisierungen notwendig. Diese Art der Variablenübergabe heißt “call by reference”, die so übergebenen Variablen Referenzparameter.

Beim zweiten Mechanismus kann die aufgerufene Prozedur die übergebenen Variablen nicht ändern. Die Prozedur kann in diesem Fall die übergebenen Variablen zwar scheinbar ändern, die Änderungen werden dem aufrufenden Programm aber nicht übermittelt. Diese Art der Variablenübergabe heißt “call by value”, die so übergebenen Variablen Wertparameter.

In VBA macht man Wertparameter kenntlich, indem man ByVal vor den Variablennamen schreibt. Beispiel 11.3 macht dies klar:

Beispiel 11.3 Wert- und Referenzparameter

```
Option Explicit
sub wertUndReferenz (ByVal wert As Long,  referenz As Long)


        wert = wert + referenz
        referenz = wert + referenz

        MsgBox ("Werte der Variablen in der Prozedur " _
                             & chr$(13) & "wert = " & wert _
                             & chr$(13) & "referenz = " & referenz)
```

```
End Sub


sub hauptprogramm()

' Programm zu Wert- und Referenz-
' parametern
' Dateiname: wertUndReferenz

       dim wert As Long
       dim referenz As Long

       wert = 9
       referenz = 10

       MsgBox ("Werte der Variablen vor dem Aufruf "  & _
                    chr$(13) & "wert = " & wert & _
                    chr$(13) & "referenz = " & referenz)

       wertUndReferenz(wert, referenz)

       MsgBox ("Werte der Variablen nach dem Aufruf " _
                          & chr$(13) & "wert = " & wert _
                          & chr$(13) & "referenz = " & referenz)

End sub
```

Abbildung 11.8 zeigt die Ausgaben von Beispiel 11.3.

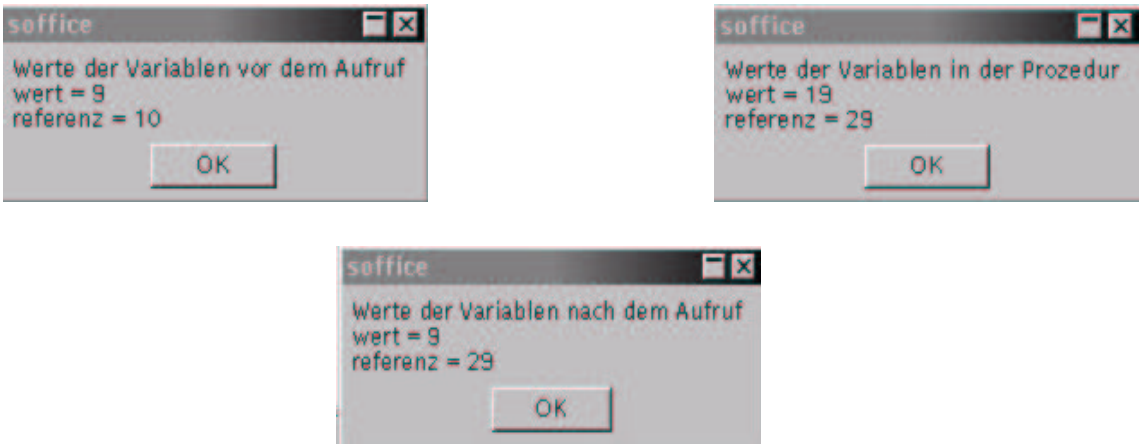

Abbildung 11.8 Unterschied Wertparameter <-> Referenzparameter

Die Prozedur `wertUndReferenz` belegt die erste übergebene Variable mit "19", die zweite mit "29"[28]. Im aufrufenden Programm ändert jedoch nur die per Referenz übergebene Variable ihren Wert.

`Lies_restliche_und_ueberpruefe_alle_Benutzereingaben()` kann also auch wie folgt programmiert werden:

28.Überlegen Sie sich, warum dies so ist.

Realisierung 11.7 Änderung der Übergabeparametertypen der Prozedur `Lies_restliche_und_ueberpruefe_alle_Benutzereingaben()`

```
sub  Lies_restliche_und_ueberpruefe_alle_Benutzereingaben(ByVal eingabe As String, _
                                        immobilienPreis As Double, _
                                        eigenKapital As Double, _
                                        zinsKlasse As Integer, _
                                        fehler As boolean)
              fehler = false
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Kaufpreis muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              immobilienPreis = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Eigenkapital muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              eigenkapital = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Zinsklasse muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              zinsKlasse = CInt(eingabe)
end sub
```

Merke: Wird einer Übergabe-Variable in der Prozedurdeklaration das Schlüsselwort `ByVal` vorangestellt, **kann** diese Variable von der Prozedur **nicht** geändert werden.

Merke: Soll eine Prozedur eine Übergabe-Variable ändern, darf dieser Variablen in der Prozedurdeklaration das Schlüsselwort `ByVal` nicht vorangestellt werden.

11.5 Prozeduren mit Übergabeparametern (Referenzparameter und Wertparameter) und lokalen Variablen

Als nächstes soll die Berechnung in eine eigene Prozedur ausgelagert werden. Auch hier müssen wir wieder überlegen, welche Parameter diese Prozedur benötigt. Die zu erstellende Prozedur benötigt *immobilienPreis*, *eigenkapital* und *zinsKlasse* als Eingaben für die Berechnung und *monatlicheBelastung* als Variable für die Ausgaben der Prozedur.

Daraus erkennen wir zunächst folgendes: *immobilienPreis*, *eigenkapital* und *zinsKlasse* können als Wertparameter übergeben werden. Sie werden von der Prozedur nicht geändert, ihre derzeitigen Werte werden aber für die Berechnungen benötigt. *monatlicheBelastung* muß als Referenzparameter übergeben werden, da das das Ergebnis der Prozedur ist.

Dann haben wir noch zwei äußerst merkwürdige Variablen: *aufzunehmenderBetrag* und *jahresBelastung*. Beide Variablen haben vor dem Aufruf der Berechnungsprozedur keinen Wert. Sie sind daher keine Eingabeparameter. Ihre Werte werden später vom Hauptprogramm auch nicht mehr genutzt, das Hauptprogramm gibt ja nur noch die montliche Belastung aus (dazu werden diese Variablen nicht benötigt). Danach wird wieder *Lies_den_Immobilienpreis_ein* aufgerufen und der nächste Durchlauf durch die Schleife beginnt. *aufzunehmenderBetrag* und *jahresBelastung* sind also auch keine Ausgabeparameter.

Was sind sie aber dann? Bei genauem Überlegen sind *aufzunehmenderBetrag* und *jahresBelastung* zwei Variablen, die die Berechnungsprozedur nur temporär zum Speichern von Zwischenergebnissen benötigt. Und es sind Variablen, die auch nur von der Berechnungsprozedur gebraucht werden.

Dennoch stehen sie bei unserer derzeitigen Lösung unter den Variablen des Hauptprogramms und müßten damit eigentlich mit übergeben werden. Da gehören sie aber nicht hin, da diese Variablen außer in der Berechnungsprozedur nirgendwo benötigt werden.

Daher bietet VBA die Möglichkeit, lokale Variablen in Prozeduren anzulegen. Diese Variablen gelten nur in der Prozedur, in der sie deklariert sind. Von außerhalb der Prozedur hat man keinen Zugriff auf die Variablen.

Wir stehen aber vor einem weiteren Problem: Auch für die Konstanten gilt dasselbe wie für die Variablen *aufzunehmenderBetrag* und *jahresBelastung*. Die Konstanten sind ebenfalls nur in der Berechnungsprozedur notwendig.

VBA stellt aus diesem Grund lokale Konstanten zur Verfügung. Sie gelten nur in der Prozedur in der sie deklariert sind.

Darüber hinaus müssen wir noch eine *fehler*-Variable aufnehmen, da ja die Eigenkapitalquote zu gering sein kann oder eine falsche Zinsklasse eingegeben werden kann.

Realisierung 11.8 Zinsbeispiel mit Prozeduren für "Gib Programmbeschreibung aus", "Lies den Immobilienpreis ein", "Lies restliche und überprüfe alle Benutzereingaben" und "führe Berechnungen durch"

```
Option Explicit
Sub  Gib_Programmbeschreibung_aus()
       MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                            & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                            & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                            & chr$(13) & " indem Sie beenden als Kaufpreis angeben")
End Sub

Sub Lies_den_Immobilienpreis_ein(immobilienPreis As String)
       immobilienPreis = InputBox ("Geben Sie jetzt den Kaufpreis ein!")
end sub

sub  Lies_restliche_und_ueberpruefe_alle_Benutzereingaben(ByVal eingabe As String, _
                                                    immobilienPreis As Double, _
                                                    eigenKapital As Double, _
                                                    zinsKlasse As Integer, _
                                                    fehler As boolean)
              fehler = false
              if Not IsNumeric (eingabe)  Then
```

```
                MsgBox ("Kaufpreis muß eine Zahl sein!")
                fehler = true
                Exit Sub
            End if
            immobilienPreis = CDbl(eingabe)

            eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
            if Not IsNumeric (eingabe)  Then
                MsgBox ("Eigenkapital muß eine Zahl sein!")
                fehler = true
                Exit Sub
            End if

            eigenkapital = CDbl(eingabe)
            eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
            if Not IsNumeric (eingabe)  Then
                MsgBox ("Zinsklasse muß eine Zahl sein!")
                fehler = true
                Exit Sub
            End if
            zinsKlasse = CInt(eingabe)
end sub


sub  fuehre_Berechnungen_durch (ByVal eigenkapital As Double, _
                            ByVal immobilienPreis As Double, _
                            ByVal zinsKlasse As Integer, _
                            monatlicheBelastung As Double, _
                            fehler As Boolean)

    const zinsKlasse1 As Double = 5.5
    const zinsKlasse2 As Double = 5.3
    const zinsKlasse3 As Double = 5.2
    const zinsKlasse4 As Double = 5.0
    const zinsKlasse5 As Double = 4.5
    const tilgung As Double = 1

    dim aufzunehmenderBetrag As Double
    dim jahresBelastung As Double
    dim eigenkapitalquote As Double

    fehler = false
    ' berechne Eigenkapitalquote
    eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
    if eigenkapitalquote < 30 then
            MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                        "% ist zu niedrig!")
            fehler = true
            exit sub
    else
            select case zinsKlasse
                case 1
                    'Monatliche Belastung berechnen
                    aufzunehmenderBetrag = immobilienPreis - eigenkapital
                    jahresBelastung = _
                            (aufzunehmenderBetrag/100)* (zinsKlasse1 + tilgung)
                    monatlicheBelastung = jahresBelastung / 12
                case 2
                    'Monatliche Belastung berechnen
                    aufzunehmenderBetrag = immobilienPreis - eigenkapital
                    jahresBelastung = _
                            (aufzunehmenderBetrag/100)* (zinsKlasse2 + tilgung)
                    monatlicheBelastung = jahresBelastung / 12
```

```
                    case 3
                           'Monatliche Belastung berechnen
                           aufzunehmenderBetrag = immobilienPreis - eigenkapital
                           jahresBelastung = _
                                  (aufzunehmenderBetrag/100)* (zinsKlasse3 + tilgung)
                           monatlicheBelastung = jahresBelastung / 12
                    case 4
                           'Monatliche Belastung berechnen
                           aufzunehmenderBetrag = immobilienPreis - eigenkapital
                           jahresBelastung = _
                                  (aufzunehmenderBetrag/100)* (zinsKlasse4 + tilgung)
                           monatlicheBelastung = jahresBelastung / 12
                    case 5
                           'Monatliche Belastung berechnen
                           aufzunehmenderBetrag = immobilienPreis - eigenkapital
                           jahresBelastung = _
                                  (aufzunehmenderBetrag/100)* (zinsKlasse5 + tilgung)
                           monatlicheBelastung = jahresBelastung / 12
                    case else
                           MsgBox ("Sie haben eine falsche Zinsklasse" & _
                                  " eingegeben! Zinsklasse muß" & _
                                  " kleiner gleich 5 sein!")
                           fehler = true
                           exit sub
             end select
      end if
end sub

Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen7

      dim eingabe As String
      dim eigenkapital As Double
      dim immobilienPreis As Double
      dim aufzunehmenderBetrag As Double
      dim monatlicheBelastung As Double
      dim zinsKlasse As Integer
      dim fehler As Boolean

      Gib_Programmbeschreibung_aus
      Lies_den_Immobilienpreis_ein eingabe

      do while eingabe <> "beenden"

            Lies_restliche_und_ueberpruefe_alle_Benutzereingaben eingabe, _
                                          immobilienPreis , _
                                          eigenKapital , _
                                          zinsklasse, _
                                          fehler
            if fehler then
                  exit sub
            end if

            Fuehre_Berechnungen_durch eigenkapital, immobilienpreis, _
                        zinsKlasse, monatlicheBelastung, fehler

            if fehler then
                  exit sub
            end if
' Gib Ergebnisse aus
```

```
            MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

            Lies_den_Immobilienpreis_ein eingabe

      loop

end sub
```

Die Deklaration lokaler Konstanten und Variablen in Prozeduren erfolgt also genauso wie wir das von VBA gewohnt sind. Variablen werden mit dem Schlüsselwort `Dim`, Konstanten mit dem Schlüsselwort `const` erzeugt. Da die Deklarationen in einem Prozedurrumpf erfolgen, handelt es sich um lokale Variablen der Prozedur.

Um das Bild von Pseudocode 10.1 zu erhalten, müßten wir jetzt noch "Gib Ergebnisse aus" in eine eigene Prozedur auslagern. Dies ist jedoch nicht sonderlich sinnvoll, da die Prozedur nur aus einer einzigen Codezeile (der `MsgBox`) bestehen würde.

Von dieser Ausnahme abgesehen, besteht unser Hauptprogramm (vgl. Realisierung 11.8) jetzt nur noch aus Prozeduraufrufen und der `do while`-Schleife. Es entspricht daher dem Pseudocode der obersten Ebene (vgl. Pseudocode 10.1).

Bis auf die Prozedur `fuehre_Berechnungen_durch` entsprechen auch alle anderen Prozeduren den Ihnen zugeordneten Pseudocodes. Was noch zu tun bleibt, ist die `Berechnung der monatlichen Belastung und der Eigenkapitalquote aus fuehre_Berechnungen_durch` auszulagern, um die in Abbildung 11.1 dargestellte Strukturierung des Pseudocodes auch in unser Programm abzubilden.

11.6 Prozedurenaufruf aus Prozeduren

Realisierung 11.9 Zinsbeispiel mit Prozeduren für "Gib Programmbeschreibung aus", "Lies den Immobilienpreis ein", "Lies restliche und überprüfe alle Benutzereingaben","führe Berechnungen durch", "Eigenkapitalquote berechnen" und "monatlichen Belastung berechnen"

```
Option Explicit

Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen8

      dim eingabe As String
      dim eigenkapital As Double
      dim immobilienPreis As Double
      dim aufzunehmenderBetrag As Double
      dim monatlicheBelastung As Double
      dim zinsKlasse As Integer
      dim fehler As Boolean

      Gib_Programmbeschreibung_aus
      Lies_den_Immobilienpreis_ein eingabe

      do while eingabe <> "beenden"
```

```
              Lies_restliche_und_ueberpruefe_alle_Benutzereingaben eingabe, _
                                          immobilienPreis , _
                                          eigenKapital , _
                                          zinsklasse, _
                                          fehler
              if fehler then
                     exit sub
              end if

              Fuehre_Berechnungen_durch  eigenkapital, immobilienpreis, _
                                   zinsKlasse, monatlicheBelastung, fehler

              if fehler then
                     exit sub
              end if


' Gib Ergebnisse aus

              MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

              Lies_den_Immobilienpreis_ein eingabe

       loop

end sub

sub berechne_Eigenkapitalquote (ByVal eigenkapital As Double, _
                           ByVal immobilienPreis As Double, _
                           fehler As Boolean)

       dim eigenkapitalquote As Double
       fehler = false

       eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
       if eigenkapitalquote < 30 then
              MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                           "% ist zu niedrig!")
              fehler = true
              exit sub
       end if
end sub

sub Monatliche_Belastung_berechnen (ByVal immobilienPreis As Double, _
                           ByVal eigenkapital As Double, _
                           ByVal zinsKlasse As Integer, _
                           monatlicheBelastung As Double)


       const tilgung As Double = 1

       dim aufzunehmenderBetrag As Double
       dim jahresBelastung As Double

       dim eigenkapitalquote As Double
       aufzunehmenderBetrag = immobilienPreis -  eigenkapital
       jahresBelastung = (aufzunehmenderBetrag/100)* (zinsKlasse + tilgung)
       monatlicheBelastung = jahresBelastung / 12

end sub

Sub  Gib_Programmbeschreibung_aus()
```

```
       MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                             & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                             & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                             & chr$(13) & " indem Sie beenden als Kaufpreis angeben")
End Sub

Sub Lies_den_Immobilienpreis_ein(immobilienPreis As String)
       immobilienPreis = InputBox ("Geben Sie jetzt den Kaufpreis ein!")
end sub

sub  Lies_restliche_und_ueberpruefe_alle_Benutzereingaben(ByVal eingabe As String, _
                                                    immobilienPreis As Double, _
                                                    eigenKapital As Double, _
                                                    zinsKlasse As Integer, _
                                                    fehler As boolean)
              fehler = false
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Kaufpreis muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              immobilienPreis = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Eigenkapital muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              eigenkapital = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Zinsklasse muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              zinsKlasse = CInt(eingabe)
end sub


sub  fuehre_Berechnungen_durch (ByVal eigenkapital As Double, _
                             ByVal immobilienpreis As Double, _
                             ByVal zinsKlasse As Integer, _
                             monatlicheBelastung As Double, _
                             fehler As Boolean)

       const zinsKlasse1 As Double = 5.5
       const zinsKlasse2 As Double = 5.3
       const zinsKlasse3 As Double = 5.2
       const zinsKlasse4 As Double = 5.0
       const zinsKlasse5 As Double = 4.5
       fehler = false
       berechne_Eigenkapitalquote eigenkapital, immobilienPreis, fehler
       if fehler Then
              exit sub
       end if
       select case zinsKlasse
              case 1
                     Monatliche_Belastung_berechnen immobilienPreis, _
                            eigenkapital, zinsKlasse1, monatlicheBelastung
              case 2
                     Monatliche_Belastung_berechnen immobilienPreis, _
```

```
                        eigenkapital, zinsKlasse2, monatlicheBelastung
            case 3
                  Monatliche_Belastung_berechnen immobilienPreis, _
                        eigenkapital, zinsKlasse3, monatlicheBelastung
            case 4
                  Monatliche_Belastung_berechnen immobilienPreis, _
                        eigenkapital, zinsKlasse4, monatlicheBelastung
            case 5
                  Monatliche_Belastung_berechnen immobilienPreis, _
                        eigenkapital, zinsKlasse5, monatlicheBelastung
            case else
                  MsgBox ("Sie haben eine falsche Zinsklasse eingegeben! " & _
                        "Zinsklasse muß kleiner gleich 5 sein!")
                  fehler = true
                  exit sub
      end select
end sub
```

Prozeduren können aus Prozeduren genauso aufgerufen werden, wie aus dem Hauptprogramm.

Des weiteren sehen wir, daß Konstanten als Aktualparameter erlaubt sind:

```
monatliche_Belastung_berechnen( monatlicheBelastung, immobilienPreis,
                                eigenkapital, zinsKlasse3);
```

zinsKlasse3 ist eine Konstante.

Als Formalparameter sind Konstanten selbstverständlich nicht erlaubt (warum nicht?).

Unser Programm entspricht unserem Pseudocode nun vollständig. Alle “Anweisungen” des Pseudocodes sind als Prozeduren realisiert und auch die in Abbildung 11.1 erkennbare Strukturierung des Pseudocodes wird in unserem Programm abgebildet. Alle in Kapitel 11.1 genannten Nachteile des ursprünglichen Entwurfs sind behoben.

Des Weiteren sehen wir, daß die Reihenfolge der Prozeduren im Modul keine Rolle spielt. `Sub zinsberechnung()` ist nun das erste Programm im Modul. Die Implementierung der Unterprogramme erfolgt später.

11.7 Testen von Prozeduren

Neben den in Kapitel 11.1 beschriebenen Vorteilen der prozeduralen Programmierung, sind aus Prozeduren zusammengesetzte Programme besser testbar. Das Programm aus Realisierung 10.1 ist nur als Ganzes testbar. Schon bei diesem Programm ist dies recht unübersichtlich. Je größer Programme werden, desto unübersichtlicher und schwieriger wird der Programmtest.

Bei der prozeduralen Programmierung ist es möglich, einzelne Prozeduren zu testen, bis sie “fehlerfrei” sind. Programme können dann aus ausgetesteten Prozeduren zusammengesetzt werden. Zum Test des Gesamtprogramms ist dann nur noch der Integrationstest notwendig, indem gezeigt werden muß, daß die Prozeduren fehlerfrei zusammenarbeiten.

Um eine Prozedur zu testen, müssen wir nur ein Hauptprogramm schreiben, in dem die Übergabevariablen deklariert werden. Wir lesen die Übergabevariablen ein, rufen die zu testende Prozedur auf und geben dann die von der Prozedur zurückgelieferten

Werte aus. Realisierung 11.10 zeigt dies am Beispiel der Prozedur Monatliche_Belastung_berechnen.

Realisierung 11.10 Programmrahmen zum Testen der Prozedur Monatliche_Belastung_berechnen

```
sub Monatliche_Belastung_berechnen (ByVal immobilienPreis As Double, _
                            ByVal eigenkapital As Double, _
                            ByVal zinsKlasse As Integer, _
                            monatlicheBelastung As Double)


       const tilgung As Double = 1

       dim aufzunehmenderBetrag As Double
       dim jahresBelastung As Double

       dim eigenkapitalquote As Double
       aufzunehmenderBetrag = immobilienPreis -  eigenkapital
       jahresBelastung = (aufzunehmenderBetrag/100)* (zinsKlasse + tilgung)
       monatlicheBelastung = jahresBelastung / 12

end sub

sub testRahmen()
' Programm um das Testen von Prozeduren zu veranschaulichen
' Dateiname funktionen9

       dim immobilienPreis As Double
       dim eigenkapital As Double
       dim zinsKlasse As Integer
       dim monatlicheBelastung As Double

       immobilienPreis = InputBox("Immobilien Preis")
       eigenkapital = InputBox("eigenkapital")
       zinsKlasse = InputBox("zinsKlasse")

       Monatliche_Belastung_berechnen immobilienPreis, _
                                   eigenkapital, _
                                   zinsKlasse, _
                                   monatlicheBelastung

       MsgBox (monatlicheBelastung)

End sub
```

11.8 Funktionen

Funktionen sind spezielle Prozeduren (oder aber: Prozeduren sind spezielle Funktionen). Funktionen zeichnen sich dadurch aus, daß sie dem aufrufenden Programm einen Wert zurückgeben. Dies tun Prozeduren auch (über Referenzparameter). Funktionen unterscheiden sich dadurch von Prozeduren, daß sie einen Wert über den Funktionsnamen zurückgeben können. Obgleich Ihnen dies erst hier richtig klar werden wird, benutzen Sie Funktionen seit Anbeginn des Kurses. Denn in fast jedem Programm haben wir InputBox-en benutzt. Und das ging z.B. so:

```
       immobilienPreis = InputBox("Immobilien Preis")
```

InputBox ist eine von VBA zur Verfügung gestellte Funktion, die einen Übergabeparameter erwartet, nämlich das, was die InputBox im Textbereich darstellen soll. InputBox gibt dann die Eingabe des Benutzers zurück. Im Unterschied zu Prozeduren, die Änderungen von Variablen nur über Referenzparameter ermöglichen, geben Funktionen einen Wert über eine Zuweisung zurück. Neben den in VBA zur Verfügung gestellten Funktionen (neben InputBox gibt es noch viele andere, so sind z.B. alle Excel-Funktionen von VBA aus nutzbar), können wir auch eigene Funktionen schreiben.

Wir machen uns das an folgendem Beispiel klar:

Realisierung 11.11 Programmrahmen zum Testen der Funktion `monatliche_Belastung_berechnen`

```
Function Monatliche_Belastung_berechnen (ByVal immobilienPreis As Double, _
                                ByVal eigenkapital As Double, _
                                ByVal zinsKlasse As Integer) As Double


       const tilgung As Double = 1

       dim aufzunehmenderBetrag As Double
       dim jahresBelastung As Double
       dim eigenkapitalquote As Double

       aufzunehmenderBetrag = immobilienPreis -  eigenkapital
       jahresBelastung = (aufzunehmenderBetrag/100)* (zinsKlasse + tilgung)
       Monatliche_Belastung_berechnen = jahresBelastung / 12
end Function

sub testRahmen()

' Programm um das Testen von Funktionen zu veranschaulichen
' Dateiname funktionen 10

       dim immobilienPreis As Double
       dim eigenkapital As Double
       dim zinsKlasse As Integer
       dim monatlicheBelastung As Double

       immobilienPreis = InputBox("Immobilien Preis")
       eigenkapital = InputBox("eigenkapital")
       zinsKlasse = InputBox("zinsKlasse")

       monatlicheBelastung = Monatliche_Belastung_berechnen (immobilienPreis, _
                                     eigenkapital, _
                                     zinsKlasse)

       MsgBox (monatlicheBelastung)

End sub
```

Unsere Prozedur `monatliche_Belastung_berechnen` aus Realisierung 11.10 gab (über einen Referenzparameter) einen Wert an das Hauptprogramm zurück, die berechnete monatliche Belastung. Solche Prozeduren können durch Funktionen ersetzt werden.

Im Unterschied zu Prozeduren werden Funktionen durch das Schlüsselwort `function` deklariert. Hiernach folgt der Funktionsname und dann die Übergabeparameter in runden Klammern.

Merke: Die Übergabeparameter von Funktionen **sollten** Wertparameter sein. Der Rückgabewert einer Funktion sollte allein über den Funktionsnamen übergeben werden. Sollte eine Funktion mehr als einen Rückgabewert erfordern, sind Prozeduren vorzuziehen. Es ist allerdings syntaktisch nicht verboten, einer Funktion Referenzparameter zu übergeben und über diese weitere Werte an das Hauptprogramm zurückzugeben. Da VBA für solche Fälle aber Prozeduren vorsieht, sind diese vorzuziehen. Eine Ausnahme von dieser Regel sind Funktionen, die neben den Berechnungen Fehlerüberprüfungen durchführen. Dann kann man das Ergebnis der Fehlerüberprüfung über den Funktionsnamen übergeben und die restlichen Ergebnisse über Referenzparameter. Kapitel 11.9 zeigt dies.

Merke: Grundsätzlich läßt sich aber sagen: Alles, was sich über Prozeduren in einer Programmiersprache lösen läßt, läßt sich ebenfalls über Funktionen lösen (und umgekehrt).

Merke: In vielen anderen Programmiersprachen (wie C, C++, Java oder Objective C) wird **nicht** zwischen Funktionen und Prozeduren unterschieden.

Nach den schließenden runden Klammern des Funktionskopfs wird der Typ des Rückgabewerts mit `As` angeschlossen.

```
Function Monatliche_Belastung_berechnen (ByVal immobilienPreis As Double, _
                                ByVal eigenkapital As Double, _
                                ByVal zinsKlasse As Integer) As Double
```

Auf die Angabe des Rückgabewertes kann man verzichten, dann wird ein `Variant` zurückgegeben.

Die Implementierung einer Funktion unterscheidet sich nur in einem Punkt von einer Prozedurimplementierung: Der Rückgabewert der Funktion wird dem Funktionsnamen zugewiesen.

```
    monatliche_Belastung_berechnen = jahresBelastung / 12
```

Ebenso wie eine Prozedur wird eine Funktion über ihren Namen aufgerufen. Im Unterschied zu Prozeduren steht der Funktionsname auf der rechten Seite einer Zuweisung. Dies ist notwendig, da der Rückgabewert der Funktion ja irgendeiner Variablen des Hauptprogramms zugewiesen werden muß. Sinnigerweise müssen die Übergabeparameter beim Aufruf einer Funktion in runde Klammern eingeschlossen werden.

```
    monatlicheBelastung = monatliche_Belastung_berechnen
                                            (immobilienPreis,
                                                  eigenkapital,zinsSatz)
```

11.9 Re-Implementierung des Zinsbeispiels mit Funktionen und Prozeduren

Hierzu schauen wir einfach nach, welche Prozeduren nur eine Variable an das Hauptprogramm zurückgeben. Solche Prozeduren schreiben wir zu Funktionen um. Im Einzelnen sind dies:

```
Sub Lies_den_Immobilienpreis_ein(immobilienPreis As String)
sub berechne_Eigenkapitalquote (ByVal eigenkapital As Double, _
                    ByVal immobilienPreis As Double, _
                    fehler As Boolean)
sub Monatliche_Belastung_berechnen (ByVal immobilienPreis As Double, _
                    ByVal eigenkapital As Double, _
                    ByVal zinsKlasse As Integer, _
                    monatlicheBelastung As Double)
```

Diese Prozeduren werden zu:

```
Function Lies_den_Immobilienpreis_ein() As String
Function berechne_Eigenkapitalquote (ByVal eigenkapital As Double, _
                    ByVal immobilienPreis As Double) _
                    As Boolean
function Monatliche_Belastung_berechnen (ByVal immobilienPreis_
                                            As Double, _
                            ByVal eigenkapital As Double, _
                            ByVal zinsKlasse As Integer) _
                            As Double
```

Damit ergibt sich:

Realisierung 11.12 Re-Implementierung des Zinsbeispiels mit Funktionen und Prozeduren

```
Option Explicit

Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen11

       dim eingabe As String
       dim eigenkapital As Double
       dim immobilienPreis As Double
       dim aufzunehmenderBetrag As Double
       dim monatlicheBelastung As Double
       dim zinsKlasse As Integer
       dim fehler As Boolean

       Gib_Programmbeschreibung_aus
       eingabe = Lies_den_Immobilienpreis_ein()

       do while eingabe <> "beenden"

              Lies_restliche_und_ueberpruefe_alle_Benutzereingaben eingabe, _
                                                    immobilienPreis , _
                                                    eigenKapital , _
                                                    zinsklasse, _
                                                    fehler
              if fehler then
```

```
                exit sub
            end if

            Fuehre_Berechnungen_durch  eigenkapital, immobilienpreis, _
                        zinsKlasse, monatlicheBelastung, fehler

            if fehler then
                exit sub
            end if


' Gib Ergebnisse aus

            MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

            eingabe = Lies_den_Immobilienpreis_ein()

        loop

end sub

Function berechne_Eigenkapitalquote (ByVal eigenkapital As Double, _
                            ByVal immobilienPreis As Double) _
                            As Boolean

        dim eigenkapitalquote As Double
        berechne_Eigenkapitalquote = false

        eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
        if eigenkapitalquote < 30 then
            MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                        "% ist zu niedrig!")
            berechne_Eigenkapitalquote = true
            exit function
        end if
end Function

function Monatliche_Belastung_berechnen (ByVal immobilienPreis As Double, _
                                    ByVal eigenkapital As Double, _
                                    ByVal zinsKlasse As Integer) _
                                    As Double


        const tilgung As Double = 1

        dim aufzunehmenderBetrag As Double
        dim jahresBelastung As Double

        dim eigenkapitalquote As Double
        aufzunehmenderBetrag = immobilienPreis -  eigenkapital
        jahresBelastung = (aufzunehmenderBetrag/100)* (zinsKlasse + tilgung)
        Monatliche_Belastung_berechnen = jahresBelastung / 12

end function

Sub  Gib_Programmbeschreibung_aus()
        MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                            & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                            & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                            & chr$(13) & " indem Sie beenden als Kaufpreis angeben")
End Sub

Function Lies_den_Immobilienpreis_ein() As String
```

```
       Lies_den_Immobilienpreis_ein = InputBox ("Geben Sie jetzt den Kaufpreis ein!")
end function

sub  Lies_restliche_und_ueberpruefe_alle_Benutzereingaben(ByVal eingabe As String, _
                                                immobilienPreis As Double, _
                                                eigenKapital As Double, _
                                                zinsKlasse As Integer, _
                                                fehler As boolean)
              fehler = false
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Kaufpreis muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              immobilienPreis = CDbl(eingabe)

              eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Eigenkapital muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if

              eigenkapital = CDbl(eingabe)
              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
              if Not IsNumeric (eingabe)  Then
                     MsgBox ("Zinsklasse muß eine Zahl sein!")
                     fehler = true
                     Exit Sub
              End if
              zinsKlasse = CInt(eingabe)
end sub


sub  fuehre_Berechnungen_durch (ByVal eigenkapital As Double, _
                           ByVal immobilienpreis As Double, _
                           ByVal zinsKlasse As Integer, _
                           monatlicheBelastung As Double, _
                           fehler As Boolean)

       const zinsKlasse1 As Double = 5.5
       const zinsKlasse2 As Double = 5.3
       const zinsKlasse3 As Double = 5.2
       const zinsKlasse4 As Double = 5.0
       const zinsKlasse5 As Double = 4.5
       fehler = false
       fehler = berechne_Eigenkapitalquote (eigenkapital, _
                                  immobilienPreis, fehler)

       if fehler Then
              exit sub
       end if
       select case zinsKlasse
              case 1
                     monatlicheBelastung = Monatliche_Belastung_berechnen( _
                                   immobilienPreis, eigenkapital, zinsKlasse1)
              case 2
                     monatlicheBelastung = Monatliche_Belastung_berechnen( _
                                   immobilienPreis, eigenkapital, zinsKlasse2)
              case 3
                     monatlicheBelastung = Monatliche_Belastung_berechnen( _
                                   immobilienPreis, eigenkapital, zinsKlasse3)
              case 4
```

```
                    monatlicheBelastung = Monatliche_Belastung_berechnen( _
                                          immobilienPreis, eigenkapital, zinsKlasse4)
              case 5
                    monatlicheBelastung = Monatliche_Belastung_berechnen( _
                                          immobilienPreis, eigenkapital, zinsKlasse5)
              case else
                    MsgBox ("Sie haben eine falsche Zinsklasse eingegeben! " & _
                                          "Zinsklasse muß kleiner gleich 5 sein!")
                    fehler = true
                    exit sub
       end select
end sub
```

11.10 Optimierung des Code, Wiederverwertbarkeit von Funktionen

Realisierung 11.12 besitzt noch einige Nachteile:

- Wir haben eine Prozedur und eine Funktion, die nur aus einer Zeile bestehen. Diese Prozedur und Funktion enstanden, weil ich einfache Beispiele für die Einführung von Prozeduren und Funktionen benötigte. Richtig sinnvoll ist das nicht. Daher nehmen wir dies zurück.

- Wir haben eine recht merkwürdige Prozedur, nämlich `Lies_restliche_und_ueberpruefe_alle_Benutzereingaben`. Diese überprüft eine ihr übergebene Benutzereingabe, wandelt diese, wenn es geht, in einen Double um, liest eine weitere Eingabe ein, wandelt auch diese, wenn es geht, in einen Double um und beendet sich dann. Diese Prozedur ist sehr spezialisiert. Sie läuft nur im Kontext unseres Zinsprogramms und macht zwei ganz unterschiedliche Dinge: Einlesen und Überprüfen. Andererseits kommt es sicherlich häufig vor (auch bei anderen Problemen), daß wir Eingaben testen und umwandeln wollen. Schön wäre es, wenn wir dann unsere im Rahmen des Zinsproblems gefundene Lösung nutzen könnten. Man spricht dann von Wiederverwertung. Wiederverwertbarkeit ist ein weiterer, wichtiger Grund für die Verwendung von Prozeduren und Funktionen. Das ist auch ganz logisch: Nehmen wir an, wir hätten eine Funktion, die eine ihr übergebene Variable überprüft und wenn es geht, diese Variable in einen Double umwandelt. Im Fehlerfall gibt die Funktion eine Fehlermeldung aus. Wollen wir die Fehlermeldung ändern, machen wir es einmal in der Funktion. Haben wir aber andererseits den Überprüfungscode durch Kopieren in viele Programme übernommen und wollen eine einheitliche, neue Fehlermeldung, so müssen wir alle diese Programme ändern. Wir lagern also die Umwandlung der Variablen in eine neue Prozedur aus. Daher verfeinern wir "Lies restliche und ueberpruefe alle Benutzereingaben". Der Pseudocode dazu lautet:

Pseudocode 11.2 Verfeinerung "Lies restliche und ueberpruefe alle Benutzereingaben"

```
wandle Immobilienpreis in Double um
lies Eigenkapital ein
wandle Eigenkapital in Double um
lies Zinsklasse ein
wandle Zinsklasse in Integer um
```

Um zur Realisierung zu kommen, müssen wir nun beschreiben, wie man einen String in einen Double bzw. Integer umwandelt:

Pseudocode 11.3 Wandle String in Double

```
if String ist nicht numerisch
       Gib Fehlermeldung aus
       Verlasse Programm
end if
konvertiere in Double
```

Pseudocode 11.4 Wandle String in Integer

```
if String ist nicht numerisch
       Gib Fehlermeldung aus
       Verlasse Programm
end if
konvertiere in Integer
```

Die Umsetzung dieser Gedanken führt zu:

Realisierung 11.13 Zinsbeispiel (verbessert)

```
Option Explicit

Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen12

       dim eingabe As String
       dim eigenkapital As Double
       dim immobilienPreis As Double
       dim aufzunehmenderBetrag As Double
       dim monatlicheBelastung As Double
       dim zinsKlasse As Integer
       dim fehler As Boolean

       MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                           & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                           & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                           & chr$(13) & " indem Sie beenden als Kaufpreis angeben")

       eingabe  = InputBox ("Geben Sie jetzt den Kaufpreis ein!")

       do while eingabe <> "beenden"

              Lies_restliche_und_ueberpruefe_alle_Benutzereingaben eingabe, _
                                                    immobilienPreis , _
                                                    eigenKapital , _
                                                    zinsklasse, _
                                                    fehler
              if fehler then
                     exit sub
              end if

              Fuehre_Berechnungen_durch eigenkapital, immobilienpreis, _
                             zinsKlasse, monatlicheBelastung, fehler

              if fehler then
                     exit sub
```

```
                end if


' Gib Ergebnisse aus

                MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

                eingabe  = InputBox ("Geben Sie jetzt den Kaufpreis ein!")

        loop

end sub

Function berechne_Eigenkapitalquote (ByVal eigenkapital As Double, _
                            ByVal immobilienPreis As Double) _
                            As Boolean

        dim eigenkapitalquote As Double
        berechne_Eigenkapitalquote = false

        eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
        if eigenkapitalquote < 30 then
                MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                            "% ist zu niedrig!")
                berechne_Eigenkapitalquote = true
                exit function
        end if
end Function

function Monatliche_Belastung_berechnen (ByVal immobilienPreis As Double, _
                                    ByVal eigenkapital As Double, _
                                    ByVal zinsKlasse As Integer) _
                                    As Double


        const tilgung As Double = 1

        dim aufzunehmenderBetrag As Double
        dim jahresBelastung As Double

        dim eigenkapitalquote As Double
        aufzunehmenderBetrag = immobilienPreis -  eigenkapital
        jahresBelastung = (aufzunehmenderBetrag/100)* (zinsKlasse + tilgung)
        Monatliche_Belastung_berechnen = jahresBelastung / 12

end function

sub  Lies_restliche_und_ueberpruefe_alle_Benutzereingaben(ByVal eingabe As String, _
                                                immobilienPreis As Double, _
                                                eigenKapital As Double, _
                                                zinsKlasse As Integer, _
                                                fehler As boolean)
                fehler = false
                wandle_in_Double_um  eingabe, immobilienPreis, fehler
                if (fehler) Then
                        exit sub
                end if

                eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein!")
                wandle_in_Double_um eingabe, eigenKapital, fehler
                if (fehler) Then
                        exit sub
                end if
```

```
              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein!")
              wandle_in_Integer_um eingabe, zinsKlasse, fehler
              if (fehler) Then
                     exit sub
              end if
end sub

sub wandle_in_Double_um(ByVal eingabe As String, rueckgabe As Double, _
                           fehler As Boolean)
       fehler = false
       if Not IsNumeric (eingabe)  Then
              MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein!")
              fehler = true
              Exit Sub
       End if
       rueckgabe = CDbl(eingabe)
end sub

sub wandle_in_Integer_um(ByVal eingabe As String, rueckgabe As Integer, _
                           fehler As Boolean)
       fehler = false
       if Not IsNumeric (eingabe)  Then
              MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein!")
              fehler = true
              Exit Sub
       End if
       rueckgabe = CInt(eingabe)
end sub

sub  fuehre_Berechnungen_durch (ByVal eigenkapital As Double, _
                           ByVal immobilienpreis As Double, _
                           ByVal zinsKlasse As Integer, _
                           monatlicheBelastung As Double, _
                           fehler As Boolean)

       const zinsKlasse1 As Double = 5.5
       const zinsKlasse2 As Double = 5.3
       const zinsKlasse3 As Double = 5.2
       const zinsKlasse4 As Double = 5.0
       const zinsKlasse5 As Double = 4.5
       fehler = false
       fehler = berechne_Eigenkapitalquote (eigenkapital, _
                                  immobilienPreis, fehler)

       if fehler Then
              exit sub
       end if
       select case zinsKlasse
              case 1
                     monatlicheBelastung = Monatliche_Belastung_berechnen( _
                                           immobilienPreis, eigenkapital, zinsKlasse1)
              case 2
                     monatlicheBelastung = Monatliche_Belastung_berechnen( _
                                           immobilienPreis, eigenkapital, zinsKlasse2)
              case 3
                     monatlicheBelastung = Monatliche_Belastung_berechnen( _
                                           immobilienPreis, eigenkapital, zinsKlasse3)
              case 4
                     monatlicheBelastung = Monatliche_Belastung_berechnen( _
                                           immobilienPreis, eigenkapital, zinsKlasse4)
              case 5
                     monatlicheBelastung = Monatliche_Belastung_berechnen( _
```

```
                                        immobilienPreis, eigenkapital, zinsKlasse5)
            case else
                  MsgBox ("Sie haben eine falsche Zinsklasse eingegeben! " & _
                                        "Zinsklasse muß kleiner gleich 5 sein!")
                  fehler = true
                  exit sub
      end select
end sub
```

Zum Abschluß führen wir noch eine weitere Verbesserung durch:

In der Bedingung des `if`-Konstrukts kann auch ein Funktionsaufruf stehen, wenn die Funktion einen Wahrheitswert (`true` oder `false`) zurückgibt. Und auch für die `if`-Anweisung gibt es eine verkürzte Form, die ich Ihnen bislang nicht verraten habe. Wenn der `if-then` Anweisungsblock aus nur einer Anweisung besteht, kann diese Anweisung direkt nach `then` in die gleiche Zeile aufgenommen werden, ein `end if` entfällt dann. Die verkürzte `if`-Anweisung hat dann die Form:

```
if Bedingung Then anweisung
```

Wir können also unsere Fehlerbehandlung wesentlich vereinfachen, indem wir aus allen unseren Prozeduren, in denen Fehler auftreten können und die deswegen eine Fehlervariable zurückgeben, bool'sche Funktionen[29] machen. Im Hauptprogramm wenden wir die verkürzte `if`-Anweisung an. Dadurch wird das Programm kürzer und besser lesbar.

Sie sollten die Anwendung der verkürzten `if`-Anweisung allerdings auf diese Fälle (Programm im Fehlerfall verlassen) beschränken. Hier ist sichergestellt, daß der `if-then` Anweisungsblock bei Programmänderungen nicht größer wird (bedeutet, daß Anweisungen hinzukommen). In Fällen wo man dies nicht weiß, ist es immer besser, die normale Form der `if`-Anweisung zu benutzen. Dies ist einfach änderungsfreundlicher.

Um unser Programm leichter lesbar zu machen, führen wir noch eine weitere Änderung durch. Zur Zeit geben unsere Unterprogramme, in denen Fehlerüberprüfungen durchgeführt werden, `true` zurück, wenn ein Fehler auftritt. Dies ist auch völlig okay, weil unsere Fehlervariable *fehler* heißt und ein jeder intuitiv erwartet, daß diese Variable auf `true` steht, wenn ein Fehler aufgetreten ist. Würden wir dies beibehalten, führt es beispielsweise zu folgendem Code:

```
if wandle_in_Double_um (eingabe, immobilienPreis) then exit sub
```

Das ist nun nicht ganz so prickelnd. Denn normal denkende Menschen erwarten, daß eine Funktion `true` zurückgibt, wenn alles richtig funktioniert hat und `false` im Fehlerfall. Bei uns ist es umgekehrt. Weil wir auch normal denkende Menschen sind (oder vielleicht diesen Eindruck erwecken wollen), machen wir dies auch so. Wir lassen unsere Funktionen `false` zurückgeben, wenn ein Fehler aufgetreten ist und `true`, wenn alles vorschriftsmäßig funktioniert hat. Dies führt zu folgendem besser verständlichem Code:

```
if Not wandle_in_Double_um (eingabe, immobilienPreis) then exit sub
```

29. Funktionen, deren Rückgabewert vom Typ Boolean ist.

Aus der vormaligen Prozedur `wandle_in_Double_um` wird somit folgende Funktion::

```
Function wandle_in_Double_um(ByVal eingabe As String, rueckgabe As Double) _
                                As Boolean
       wandle_in_Double_um = true
       if Not  IsNumeric (eingabe)  Then
              MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
              wandle_in_Double_um = false
              Exit function
       End if
       rueckgabe = CDbl(eingabe)
end function
```

Zunächst wird der Rückgabewert der Funktion mit `true` initialisiert

```
       wandle_in_Double_um = true
```

Im Fehlerfall wir der Rückgabewert auf false gesetzt:

```
       if Not  IsNumeric (eingabe)  Then
              MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
              wandle_in_Double_um = false
              Exit function
       End if
```

Der Rest der Implementierung bleibt "as is".

Durch diese Vorgehensweise besteht unsere Realisierung jetzt aus kleinen autarken Prozeduren und Funktionen, die getrennt getestet und zu einem großen Ganzen zusammengesetzt werden. Insbesondere das Hauptprogramm ist gut überschaubar.

Realisierung 11.14 Das Zinsbeispiel weiter verbessert

```
Option Explicit

Sub zinsberechnung()

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: funktionen13

       dim eingabe As String
       dim eigenkapital As Double
       dim immobilienPreis As Double
       dim aufzunehmenderBetrag As Double
       dim monatlicheBelastung As Double
       dim zinsKlasse As Integer
       dim fehler As Boolean

       MsgBox ("Bitte geben Sie Ihr Eigenkapital und den "  _
                            & chr$(13) & "Kaufpreis ein, sowie die Ihnen zugeteilte"  _
                            & chr$(13) & " Zinsklasse ein! Sie beenden das Programm,"  _
                            & chr$(13) & " indem Sie beenden als Kaufpreis angeben")

       eingabe  = InputBox ("Geben Sie jetzt den Kaufpreis ein! ")

       do while eingabe <> "beenden"

              if Not Lies_restliche_und_ueberpruefe_alle_Benutzereingaben(eingabe, _
                     immobilienPreis ,eigenKapital, zinsklasse) then exit sub

              if Not Fuehre_Berechnungen_durch (eigenkapital, immobilienpreis, _
```

```
                    zinsKlasse, monatlicheBelastung) then exit sub

' Gib Ergebnisse aus

              MsgBox("Ihre monatliche Belastung ist: " & monatlicheBelastung & " DM")

              eingabe  = InputBox ("Geben Sie jetzt den Kaufpreis ein! ")

       loop

end sub

Function berechne_Eigenkapitalquote (ByVal eigenkapital As Double, _
                           ByVal immobilienPreis As Double) _
                           As Boolean

       dim eigenkapitalquote As Double
       berechne_Eigenkapitalquote = true

       eigenkapitalquote = (eigenkapital / immobilienPreis) * 100
       if eigenkapitalquote < 30 then
              MsgBox("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                           "% ist zu niedrig! ")
              berechne_Eigenkapitalquote = false
              exit function
       end if
end Function

function Monatliche_Belastung_berechnen (ByVal immobilienPreis As Double, _
                                  ByVal eigenkapital As Double, _
                                  ByVal zinsKlasse As Integer) _
                                  As Double


       const tilgung As Double = 1

       dim aufzunehmenderBetrag As Double
       dim jahresBelastung As Double

       dim eigenkapitalquote As Double
       aufzunehmenderBetrag = immobilienPreis -  eigenkapital
       jahresBelastung = (aufzunehmenderBetrag/100)* (zinsKlasse + tilgung)
       Monatliche_Belastung_berechnen = jahresBelastung / 12

end function

Function  Lies_restliche_und_ueberpruefe_alle_Benutzereingaben(ByVal eingabe _
                                  As String, immobilienPreis As Double, _
                                  eigenKapital As Double, _
                                  zinsKlasse As Integer) _
                                  As boolean

              Lies_restliche_und_ueberpruefe_alle_Benutzereingaben = true
              if Not wandle_in_Double_um (eingabe, immobilienPreis) Then
                     Lies_restliche_und_ueberpruefe_alle_Benutzereingaben = false
                     exit function
              end if

              eingabe = InputBox ("Geben Sie nun ihr Eigenkapital ein! ")
              if Not wandle_in_Double_um(eingabe, eigenKapital) Then
                     Lies_restliche_und_ueberpruefe_alle_Benutzereingaben = false
                     exit function
              end if
```

```
              eingabe = InputBox ("Geben Sie nun ihre Zinsklasse ein! ")
              if Not wandle_in_Integer_um (eingabe, zinsKlasse) Then
                     Lies_restliche_und_ueberpruefe_alle_Benutzereingaben = false
                     exit function
              end if
end Function

Function wandle_in_Double_um(ByVal eingabe As String, rueckgabe As Double) _
                                   As Boolean
       wandle_in_Double_um = true
       if Not  IsNumeric (eingabe)  Then
              MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
              wandle_in_Double_um = false
              Exit function
       End if
       rueckgabe = CDbl(eingabe)
end function

function wandle_in_Integer_um(ByVal eingabe As String, rueckgabe As Integer) _
                                                  As Boolean
       wandle_in_Integer_um = true
       if Not  IsNumeric (eingabe)  Then
              MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
              wandle_in_Integer_um = false
              Exit Function
       End if
       rueckgabe = CInt(eingabe)
end function

Function  fuehre_Berechnungen_durch (ByVal eigenkapital As Double, _
                                   ByVal immobilienpreis As Double, _
                                   ByVal zinsKlasse As Integer, _
                                   monatlicheBelastung As Double) _
                                   As Boolean

       const zinsKlasse1 As Double = 5.5
       const zinsKlasse2 As Double = 5.3
       const zinsKlasse3 As Double = 5.2
       const zinsKlasse4 As Double = 5.0
       const zinsKlasse5 As Double = 4.5

       fuehre_Berechnungen_durch = true

       if  Not berechne_Eigenkapitalquote (eigenkapital, immobilienPreis) Then
              fuehre_Berechnungen_durch = false
              exit Function
       end if
       select case zinsKlasse
              case 1
                     monatlicheBelastung = Monatliche_Belastung_berechnen _
                            (immobilienPreis, eigenkapital, zinsKlasse4)
              case 2
                     monatlicheBelastung = Monatliche_Belastung_berechnen _
                            (immobilienPreis, eigenkapital, zinsKlasse4)
              case 3
                     monatlicheBelastung = Monatliche_Belastung_berechnen _
                            (immobilienPreis, eigenkapital, zinsKlasse4)
              case 4
                     monatlicheBelastung = Monatliche_Belastung_berechnen _
                            (immobilienPreis, eigenkapital, zinsKlasse4)
              case 5
                     monatlicheBelastung = Monatliche_Belastung_berechnen _
```

```
                              (immobilienPreis, eigenkapital, zinsKlasse4)
            case else
                  MsgBox ("Sie haben eine falsche Zinsklasse eingegeben!  " & _
                                    "Zinsklasse muß kleiner gleich 5 sein! ")
                  fuehre_Berechnungen_durch = false
                  exit Function
      end select
end Function
```

11.11 Gültigkeit von Variablen, Funktionen und Prozeduren

Unser bisheriger Code ist in Module gegliedert. In den Modulen haben wir mit Funktionen und Prozeduren eine Struktur erzeugt. Variablen haben wir bislang immer innerhalb der Prozeduren und Funktionen deklariert. Dies bedeutete, daß wir nur innerhalb der Prozedur oder Funktion, in der die Variable deklariert war, hatten. Wenn wir eine Variable an eine andere Prozedur oder Funktion weitergeben wollten, haben wir Übergabeparameter benutzt.

VBA bietet noch eine weitere Möglichkeit, Variablen für mehrere Prozeduren oder Funktionen zugänglich zu machen. Variablen können nach `Option Explicit` und vor der ersten Funktions- oder Prozedurimplementierung deklariert werden[30]. Solche Variablen heißen globale Variablen. Alle Funktionen und Prozeduren des Moduls haben Zugriff auf solche Variablen und können sie ändern oder auslesen. Beispiel 11.4 zeigt diesen Sachverhalt:

Beispiel 11.4 Globale Variablen eines Moduls

```
Option Explicit
'Prozeduren zur Demonstration globaler Variablen
' Dateiname: globaleVariable

dim globaleVariable As Integer

sub steuerung()
      ersteProzedur
      zweiteProzedur
end sub


sub ersteProzedur()
      globaleVariable = 5
      MsgBox ("ErsteProzedur, globaleVariable = " & _
                                    globaleVariable)
end Sub

sub zweiteProzedur()
      globaleVariable = globaleVariable + 5
      MsgBox ("zweiteProzedur, globaleVariable = " & _
                                    globaleVariable)
end Sub
```

30. Strenggenommen ist das so auch nicht richtig, Variablen können überall außerhalb von Prozeduren und Funktionen deklariert werden.

In Beispiel 11.4 wird zunächst außerhalb jeder Prozedur eine Variable erkärt.

```
dim globaleVariable As Integer
```

Dann sehen wir unser Hauptprogramm. Es ruft nacheinander zwei Prozeduren auf.

```
sub steuerung()
       ersteProzedur
       zweiteProzedur
end sub
```

Die erste Prozedur weist der Variablen `globaleVariable` den Wert 5 zu und gibt die Variable aus.

```
globaleVariable = 5
MsgBox ("ErsteProzedur, globaleVariable = " & _
                              globaleVariable)
```

5 wird ausgegeben. Dann beendet sich die erste Prozedur. Da es sich bei der Variable `globaleVariable` um eine globale Variable[31] handelt, behält die Variable ihren Wert, obwohl die Prozedur beendet ist. Die zweite Prozedur addiert 5 zum Wert von `globaleVariable` hinzu und gibt `globaleVariable` erneut aus. Nun wird 10 ausgegeben. Beispiel 11.4 zeigt die Ausgaben von Beispiel 11.4.

Abbildung 11.9 Ausgabe von Beispiel 11.4

Globale Variable sollte man nur in Ausnahmefällen benutzen. Funktionen, die die benötigten Variablen als Übergabeparameter erhalten, sind in sich abgeschlossene Einheiten. Funktionen, die auf globalen Variablen basieren, benötigen zum korrekten Funktionieren halt außerdem noch die globalen Variablen. Dies senkt die Wiederverwendungsmöglichkeiten dieser Funktionen. Außerdem verliert der Code in gewissem Sinne seine Struktur. Er wird schwerer zu überschauen.

Variablen und Konstanten können sogar über Modulgrenzen hinweg zur Verfügung gestellt werden. Variablen werden dazu, wie die globalen Variablen eines Moduls, außerhalb einer Funktion oder Prozedur deklariert. Zur Deklaration wird jedoch das Schlüsselwort `public` anstelle von `dim` verwendet. Konstantendeklarationen wird das Schlüsselwort `public` vorangestellt.

Beispiel 11.5 Modulübergreifende Variablen und Konstanten

```
Option Explicit
'Prozeduren zur Demonstration globaler Variablen
' und Konstanten
' Dateiname: globaleVariable2

dim globaleVariable As Integer
```

31.Man beachte die unglaublich intelligente Namensgebung der Variablen.

```
' Definition einer Variablen und einer Konstanten,
' die auch in anderen Modulen genutzt werden
' lönnen.

public const tester As Integer = 4
public superVariable As String


sub steuerung()
       ersteProzedur
       zweiteProzedur
end sub


sub ersteProzedur()
       globaleVariable = 5
       MsgBox ("ErsteProzedur, globaleVariable = " & _
                                    globaleVariable)
end Sub

sub zweiteProzedur()
       globaleVariable = globaleVariable + 5
       MsgBox ("zweiteProzedur, globaleVariable = " & _
                                    globaleVariable)
end Sub
```

Während es für modulübergreifende Konstanten mehrere Anwendungsmöglichkeiten gibt, z.B. um innerhalb eines Projekts die Fehlermeldungen zu vereinheitlichen, gibt es für modulübergreifende Variablen außer um Verwirrung zu stiften m.E. keine sinnvollen Anwendungsmöglichkeiten.

Beispiel 11.6 Anwendung modulübergreifender Konstanten: Das Definitionsmodul

```
Option Explicit

'Prozeduren zur Demonstration globaler Konstanten
' Dateiname: globaleKonstante

public const KEINE_ZAHL = "Dieses Eingabefeld erfordert eine Zahl!"
```

Beispiel 11.7 Anwendung modulübergreifender Konstanten: Nutzung der Konstanten

```
Option Explicit

'Prozeduren zur Demonstration globaler Konstanten
' Dateiname: globaleKonstante2

sub nutzungGlobalerKonstanter()
       MsgBox(KEINE_ZAHL)
end sub
```

Hier wird im ersten Modul eine modulübergreifende Konstante deklariert. Alle weiteren Module in der Arbeitsmappe haben Zugriff auf diese Konstante und können sie benutzen. Abbildung 11.10 zeigt die Ausgabe von Beispiel 11.6 und Beispiel 11.7.

Abbildung 11.10 Ausgabe von Beispiel 11.7

Prozeduren und Funktionen sind in anderen Modulen verfügbar.

11.12 Optionale Parameter

Bislang stimmte die Anzahl der Parameter in der Prozedurdeklaration und im Aufruf überein. Im Regelfall ist das auch so. VBA erlaubt uns allerdings, optionale Parameter zu deklarieren. Dies erfolgt über das Schlüsselwort `Optional`. Wir machen uns den Sachverhalt an einem Beispiel klar. Hier soll eine Prozedur erstellt werden, die eine Variable um einen ihr übergebenen Betrag erhöht oder um Eins, falls kein Erhöhungsbetrag übergeben wurde.

Beispiel 11.8 Optionale Parameter zum Ersten

```
sub optionaleParameter()
'Programm demonstriert optionale Parameter
' Dateiname: optionaleParameter
       dim zahl As Integer
       dim schritt As Integer

       zahl = InputBox ("Geben Sie die zu erhöhende Zahl ein!")
'      Aufruf mit nur einem Parameter
        zahl
       MsgBox("Zahl zum Ersten, um Eins erhöht: " & zahl)
       erhoehen zahl

       schritt = InputBox ("Geben Sie die Erhöhung ein!")
'      Aufruf mit zwei Parametern
       erhoehen zahl, schritt
       MsgBox("Zahl zum Zweiten, um Ihre Eingabe erhöht: " & zahl)
End Sub

sub erhoehen (zuErhoehendeZahl As Integer, Optional increment)
       if isMissing (increment) Then
              zuErhoehendeZahl = zuErhoehendeZahl + 1
       else
              increment = CInt(increment)
              zuErhoehendeZahl = zuErhoehendeZahl + increment
       End If
End Sub
```

Die Prozedur `erhoehen` überprüft zunächst, ob der optionale Parameter übergeben wurde:

```
       if isMissing (increment) Then
```

Dazu stellt VBA die interne Funktion `isMissing` zur Verfügung. `isMissing` gibt `true` zurück, wenn der optionale Parameter nicht mit übergeben wird. Wir sehen darüber hinaus, daß Funktionsaufrufe durchaus als Bedingung zulässig sind (sofern sie

vom Typ Boolean sind). `isMissing` funktioniert allerdings nur mit `Variant`-Variablen, daher auch die `Variant`-Deklaration des optionalen Parameters im Prozedurkopf.

Wir müssen also, wenn der optionale Parameter mit übergeben wird, diesen in einen Integer konvertieren, damit die Addition einwandfrei funktioniert:

```
increment = CInt(increment)
```

Die Prozedur erhoehen kann auf zwei Weisen aufgerufen werden, einmal mit einem Parameter:

```
erhoehen zahl
```

oder aber mit zwei Parametern:

```
erhoehen zahl, schritt
```

Beide Möglichkeiten werden im Hauptprogramm demonstriert. Abbildung 11.11 zeigt einen beispielhaften Durchlauf von Beispiel 11.8.

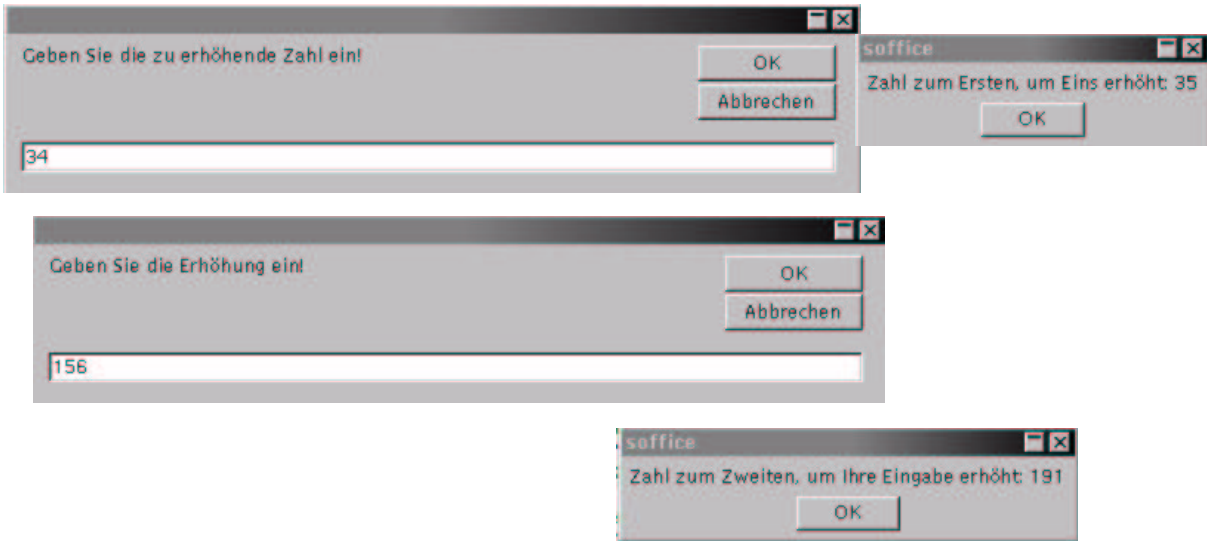

Abbildung 11.11 Ausgabe von Beispiel 11.8

Für Beispiel 11.8 gibt es eine weitere Lösung. Nicht-`Variant`-Variablen können zwar nicht mit `isMissing` auf ihr Vorhandensein getestet werde, sie werden aber von VBA auf definierte Werte gesetzt. Nocht vorhandene Variablen der Zahlentypen sind in der Prozedur Null, Strings der leere String[32]. Wir können also in der Prozedur den optionalen Parameter increment gegen Null testen. Beispiel 11.9 zeigt diese Lösung.

Beispiel 11.9 Optionale Parameter zum Zweiten

```
Option Explicit
sub optionaleParameter2()
'Programm demonstriert optionale Parameter
```

32.Dies ist keinesfalls eine besonders sinnreiche Lösung. Intuitiv würde man erwarten, daß es nicht übergebene optionale Parameter im Unterprogramm nicht gibt, so wie das bei Variant-Variablen der Fall ist. So kann ich mir vorstellen, daß Microsoft dieses Verhalten in späteren VBA-Versionen ändert. Code wie in Beispiel 11.9 würde dann nicht mehr laufen.

```
' Dateiname: optionaleParameter2
       dim zahl As Integer
       dim schritt As Integer

       zahl = InputBox ("Geben Sie die zu erhöhende Zahl ein!")
'      Aufruf mit nur einem Parameter
       inkrementieren2 zahl
       MsgBox("Zahl zum Ersten, um Eins erhöht: " & zahl)

       schritt = InputBox ("Geben Sie die Erhöhung ein!")
'      Aufruf mit zwei Parametern
       inkrementieren2 zahl, schritt
       MsgBox("Zahl zum Zweiten, um Ihre Eingabe erhöht: " & zahl)
End Sub

sub inkrementieren2(zuErhoehendeZahl As Integer, _
                                    Optional increment As Integer)
       if increment = 0 Then
              zuErhoehendeZahl = zuErhoehendeZahl + 1
       else
              zuErhoehendeZahl = zuErhoehendeZahl + increment
       End If
End Sub
```

Alle nicht-optionalen Parameter müssen vor den Optionalen deklariert werden. Oder aber andersrum: Sobald ein Übergabe-Parameter optional ist, müssen auch alle folgenden Parameter optional sein.

Ein weiteres Problem stellt sich uns: Wenn wir mehrere optionale Parameter deklarieren und wir wollen den letzten optionalen Parameter übergeben, alle andern aber nicht, wie realisieren wir so etwas? Auch dies zeige ich an einem Beispiel: Stellen wir uns vor, wir wollten Beispiel 11.8 so erweitern, daß ein weiterer Parameter übergeben wird, mit dem die eingegebene Zahl, nachdem sie inkrementiert wurde, multipliziert wird. Wird kein Inkrementierungsbetrag eingegeben, so wird nur multipliziert. Beim Fehlen des Multiplikators wird nur addiert. Fehlen beide, so wird um Eins erhöht[33]. zeigt die Realisierung:

Beispiel 11.10 Mehrere optionale Parameter

```
Option Explicit
sub optionaleParameter3()
'Programm demonstriert optionale Parameter
' Dateiname: optionaleParameter
       dim zahl As Integer
       dim schritt As Integer
       dim multiplikator As Integer

       zahl = InputBox ("Geben Sie die zu erhöhende Zahl ein!")
'      Aufruf mit nur einem Parameter
       erhoehen3 zahl
       MsgBox("Zahl zum Ersten, um Eins erhöht: " & zahl)

       schritt = InputBox ("Geben Sie die Erhöhung ein!")
'      Aufruf mit zwei Parametern
       erhoehen3 zahl, schritt
```

33. Das ist jetzt nicht das praxisorientierteste Beispiel von allen, aber Sie sollen sowieso nicht mit Funktionen mit variabler Parameterzahl arbeiten. In allen wichtigen Programmiersprachen geht das eh nicht. Wir müssen das aber durchnehmen, da viele interne VBA-Funktionen optionale Parameter besitzen.

```
        MsgBox("Zahl zum Zweiten, um Ihre Eingabe erhöht: " & zahl)

        multiplikator = InputBox ("Geben Sie nun den Multiplikator ein!")
'       Aufruf mit drei Parametern
        erhoehen3 zahl, schritt, multiplikator
        MsgBox("Zahl zum Dritten, auch noch multipliziert: " & zahl)

' jetzt wollen wir noch mal multiplizieren, aber ohne vorherige Addition

        erhoehen3 zahl, , multiplikator
        MsgBox("Zahl zum Vierten, noch mal multipliziert: " & zahl)
End Sub

sub erhoehen3(zuErhoehendeZahl As Integer, Optional increment, _
                                    Optional multiplikator)
        if isMissing(increment) Then
                        if isMissing(multiplikator) Then
                                zuErhoehendeZahl = zuErhoehendeZahl + 1
                        else
                                multiplikator = CInt(multiplikator)
                                zuErhoehendeZahl = zuErhoehendeZahl * multiplikator
                        End if
        else
                increment = CInt(increment)
                if isMissing(multiplikator) Then
                        zuErhoehendeZahl = zuErhoehendeZahl + increment
                else
                        multiplikator = CInt(multiplikator)
                        zuErhoehendeZahl=(zuErhoehendeZahl + increment)*multiplikator
                End if
        End If
End Sub
```

An der Zeile:

```
        erhoehen3 zahl, , multiplikator
```

sehen wir, wie der Aufruf bei fehlenden optionalen Parametern funktioniert. Sie werden einfach weggelassen[34] und um klarzumachen, das dieser optionale Parameter fehlt, wird ein Komma gesetzt. Abbildung 11.12 zeigt einen beispielhaften Durchlauf von Beispiel 11.9.

Den Rest des Codes müßten Sie eigentlich verstehen können. Wir sehen allerdings, daß das Arbeiten mit optionalen Parametern zu vielen Fallunterscheidungen in den betroffenen Prozeduren führt. Schließlich muß jede mögliche Kombination der optionalen Parameter überprüft werden. Dies führt nicht unbedingt zu klarerem Programmcode und ist daher auch nur bedingt zu empfehlen.

11.13Benannte Parameter

VBA verfügt über interne Funktionen mit teilweise ziemlich vielen optionalen Parametern. Dies würde zu Prozeduraufrufen wie

```
        optional parameter1,,,,,parameter2,,,,,,,,parameter3,parameter4,,,,parameter5
```

34.Was irgendwie logisch ist.

Abbildung 11.12 Ausgabe von Beispiel 11.9

führen. So etwas sieht nicht nur ziemlich blöde aus, es ist auch noch schwer nachzuvollziehen. Man muß Kommas zählen, um festzustellen, welcher Parameter weggelassen wurde.

Aus diesem Grund kennt VBA eine weitere Möglichkeit um festzulegen, welche Variable an welche übergeben wird. Man kann die Variablennamen in der Prozedurdeklaration nutzen. Beispiel 11.11 zeigt dies.

Beispiel 11.11 Nutzung der Namen von Parametern bei der Übergabe

```
Option Explicit
sub parameterMitNamen()
'Programm demonstriert Parameter mit Namen
' Dateiname: parameterMitNamen

       dim zahl As Integer
       dim schritt As Integer
       dim multiplikator As Integer

       zahl = InputBox ("Geben Sie die zu erhöhende Zahl ein!")
'      Aufruf mit nur einem Parameter
       erhoehen3 zuErhoehendeZahl:=zahl
       MsgBox("Zahl zum Ersten, um Eins erhöht: " & zahl)

       schritt = InputBox ("Geben Sie die Erhöhung ein!")
'      Aufruf mit zwei Parametern
       erhoehen3 zuErhoehendeZahl:=zahl, increment:=schritt
       MsgBox("Zahl zum Zweiten, um Ihre Eingabe erhöht: " & zahl)

       multiplikator = InputBox ("Geben Sie nun den Multiplikator ein!")
'      Aufruf mit drei Parametern
       erhoehen3 zuErhoehendeZahl:= zahl, increment:=schritt, _
                                  multiplikator:=multiplikator
       MsgBox("Zahl zum Dritten, auch noch multipliziert: " & zahl)

' jetzt wollen wir noch mal multiplizieren, aber ohne vorherige Addition
```

```
        erhoehen3 zuErhoehendeZahl:=zahl, multiplikator:=multiplikator
        MsgBox("Zahl zum Vierten, noch mal multipliziert: " & zahl)
End Sub

sub erhoehen3(zuErhoehendeZahl As Integer, Optional increment, _
                                Optional multiplikator)
        if isMissing(increment) Then
                if isMissing(multiplikator) Then
                        zuErhoehendeZahl = zuErhoehendeZahl + 1
                else
                        multiplikator = CInt(multiplikator)
                        zuErhoehendeZahl = zuErhoehendeZahl * multiplikator
                End if
        else
                increment = CInt(increment)
                if isMissing(multiplikator) Then
                        zuErhoehendeZahl = zuErhoehendeZahl + increment
                else
                        multiplikator = CInt(multiplikator)
                        zuErhoehendeZahl=(zuErhoehendeZahl + increment)*multiplikator
                End if
        End If
End Sub
```

Die Übergabe an einen Übergabeparameter einer Prozedur kann also auch erfolgen, indem man beim Aufruf den Namen des Parameters im Prozedurkopf, gefolgt von `:=` gefogt von der Variable, die übergeben wird, hinschreibt.

Dies sieht zwar ziemlich lustig aus, wenn beide gleich sind, wie bei

```
        erhoehen3 zuErhoehendeZahl:= zahl, increment:=schritt, _
                                multiplikator:=multiplikator
```

oder

```
        erhoehen3 zuErhoehendeZahl:=zahl, multiplikator:=multiplikator
```

zeigt aber ein weiteres Mal, daß Variablen in Haupt- und Unterprogrammen, selbst wenn sie den gleichen Namen haben, unabhängig voneinander sind. Denn

```
        multiplikator:=multiplikator
```

bedeutet nichts anderes als: Verbinde die Variable *multiplikator* des Hauptprogramms mit der Variablen *multiplikator* des Unterprogramms.

11.14Zwei weitere Beispiele: Das Provisionsprogramm und der Taschenrechner mit Prozeduren und Funktionen

Zum Abschluß dieses Kapitels wollen wir unsere beiden anderen Beispiele in eine strukturierte Form überführen. Wir beginnen mit dem Provisionsbeispiel. Dazu rekapitulieren wir den Pseudocode.

Pseudocode 11.5 Provision berechnen mit Do-While Schleife (wiederholt)

```
        Gib Programmbeschreibung aus
        Lies den umsatz ein
        do while umsatz nicht gleich "beenden"
                lies restliche und überprüfe alle Benutzereingaben
```

```
            bestimme Provision in Prozent
            berechne auszuzahlenden Betrag Formel:
            (verkaufsbetrag*provision in prozent)/100
            Gib das Ergebnis aus
            Lies den umsatz ein
      Loop
```

Bei der Umsetzung des Pseudocode zu einer prozeduralen Programmierung werden die Anweisungen des Pseudocode zu Prozeduren. Diese Prozeduren müssen wir implementieren. Glücklicherweise ist dies zu einem großen Teil bereits geschehen. Bei der Realisierung des Provisionsbeispiels hatten wir ja immer die “Pseudocode-Anweisung” als Kommentar über den entsprechenden Codezeilen aufgenommen. Überlegen müssen wir uns im wesentlichen nur, für welche Anweisungen des Pseudocode wir Funktionen schreiben müssen. Darüber hinaus müssen wir die Übergabe- und lokalen Variablen der Prozeduren oder Funktionen bestimmen.

Zunächst stellen wir fest: "Gib Programmbeschreibung aus", "Lies den umsatz ein" und "Gib das Ergebnis aus" sind mit jeweils einer Anweisung realisierbar. Müssen wir also nichts machen. "lies restliche und überprüfe alle Benutzereingaben" werden wir völlig analog zu Realisierung 11.14 programmieren. Wir haben es jetzt sogar noch besser, weil wir `wandle_in_Double_um` nicht mehr programmieren müssen. Wir rufen einfach unsere bereits programmierte Funktion auf. Obwohl ich das neue Provisionsprogramm in einem eigenen Modul entwickle, sind, wie in Kapitel 11.11 dargestellt, Funktionen modulübergreifend verfügbar.

"berechne auszuzahlenden Betrag" ist ebenfalls ein Einzeiler. Da dies funktional eng mit "bestimme Provision in Prozent" zusammenhängt, realisieren wir dies zusammen in einer Funktion, der wir den Namen `berechne_Provision` geben. Die Konstantendefinitionen benötigen wir nur in dieser Funktion, daher lagern wir sie aus dem Hauptprogramm aus. Sie werden zu lokalen Konstanten in `berechne_Provision`. Auch die Variable *provisionInProzent* wird nur in berechne_Provision benötigt. Sie wird also zur lokalen Variablen der Funktion. Die Übergabeparameter sind auch klar: es sind *umsatz* und *verkaufsbetrag*. Beide werden in berechne_Provision nicht verändert, also benutzen wir Wertübergabe. Wir erhalten:

Realisierung 11.15 Das Provisionsbeispiel mit Funktionen

```
Option Explicit

Sub provisionMitFunktionen()

' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provisionMitFunktionen

      dim umsatz As Double
      dim verkaufsbetrag As Double
      dim auszuzahlendeProvision As Double
      dim eingabe As String



      ' Gib Programmbeschreibung aus

      MsgBox("Geben Sie Umsatz und Verkaufsbetrag ein!" _
                  & chr$(13) & "Das Programm berechnet die" _
                  & " Provision des Vermittlers! Sie beenden das" _
```

```
                    & " Programm durch die Eingabe von: beenden!")


' Lies Umsatz ein

       eingabe = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")

       Do while eingabe <> "beenden"
              if Not Lies_und_ueberpruefe_Benutzereingaben(eingabe, _
                     umsatz ,verkaufsbetrag) then exit sub

              auszuzahlendeProvision = berechne_provision(umsatz, _
                                                          verkaufsbetrag)

              ' Gib das Ergebnis aus
              MsgBox ("Die Provision für dieses Geschäft ist: " _
                                 & auszuzahlendeProvision & " DM")
              ' Lies Umsatz ein
              eingabe = InputBox ("Geben Sie nun den Umsatz des Kunden ein!")
       Loop

End Sub

Function  Lies_und_ueberpruefe_Benutzereingaben(ByVal eingabe As String, _
                                 umsatz As Double, _
                                 verkaufsbetrag As Double) _
                                 As boolean

              Lies_und_ueberpruefe_Benutzereingaben = true
              if Not wandle_in_Double_um (eingabe, umsatz) Then
                     Lies_und_ueberpruefe_Benutzereingaben = false
                     exit function
              end if

              eingabe = InputBox ("Geben Sie nun den Verkaufsbetrag ein!")
              if Not wandle_in_Double_um (eingabe, verkaufsbetrag) Then
                     Lies_und_ueberpruefe_Benutzereingaben = false
                     exit function
              end if

              if verkaufsbetrag > umsatz Then
                     MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
                     Lies_und_ueberpruefe_Benutzereingaben = false
                     Exit Function
              End if
end Function

Function berechne_Provision (ByVal umsatz As Double, _
                     ByVal verkaufsbetrag As Double) _
                     As Double

       dim provisionInProzent As Double

       ' Umsatzgrenzen sind DM-Betraege

       const umsatzGrenze1 As Double = 100000
       const umsatzGrenze2 As Double = 500000
       const umsatzGrenze3 As Double = 1000000

       ' Provisionen in Prozent

       const provisionUmsatzGrenze1 As Double = 5
```

```
        const provisionUmsatzGrenze2 As Double = 10
        const provisionUmsatzGrenze3 As Double = 20

        ' Bestimme Provision
        if umsatz >= umsatzGrenze3 Then
                provisionInProzent = provisionUmsatzGrenze3
        elseIf umsatz >= umsatzGrenze2 Then
                provisionInProzent = provisionUmsatzGrenze2
        elseIf umsatz >= umsatzGrenze1 Then
                provisionInProzent = provisionUmsatzGrenze1
        else
                provisionInProzent = 0
        End if

' Berechne die Provision

        berechne_Provision = (verkaufsbetrag * provisionInProzent)/100

end Function
```

Kommen wir nun zum Taschenrechnerprogramm. Auch hier sehen wir uns den Pseucode noch einmal an:

Pseudocode 11.6 Taschenrechner mit do while-Schleife

```
        Gib Programmbeschreibung aus
        Lies ersten Operanden ein
        do while erster Operand ist ungleich 0
                Lies restliche und überprüfe alle Benutzereingaben
                führe Berechnung durch
                Gib das Ergebnis aus
                Lies ersten Operanden ein
        loop
```

Zunächst stellen wir fest: "Gib Programmbeschreibung aus", "Lies ersten Operanden ein" und "Gib das Ergebnis aus" sind mit jeweils einer Anweisung realisierbar. Müssen wir also nichts machen. "lies restliche und überprüfe alle Benutzereingaben" werden wir völlig analog zu Realisierung 11.14 programmieren. Wir haben es jetzt sogar noch besser, weil wir `wandle_in_Double_um` nicht mehr programmieren müssen. Wir rufen einfach unsere bereits programmierte Funktion auf. Obwohl ich das neue Taschenrechnerprogramm in einem eigenen Modul entwickle, sind, wie in Kapitel 11.11 dargestellt, Funktionen modulübergreifend verfügbar.

Bleibt noch "führe Berechnung durch". Dieser Funktion müssen *ersterOperand*, *zweiterOperand* und *operator* übergeben werden. Alle drei Variablen werden in der Funktion nicht geändert. Wir werden also Wertübergabe benutzen. Die Variable *ergebnis* kommt von der Funktion zurück. Wir übergeben daher "by reference". Lokale Variablen benötigt die Funktion nicht. Wir erhalten:

Realisierung 11.16 Taschenrechner mit Funktionen

```
Option Explicit
Sub taschenrechnerMitFunktion()
' Taschenrechnerprogramm mit Funktionen
' Dateiname: taschenrechnerMitFunktion

dim ersterOperand As Double
```

```
dim zweiterOperand As Double
dim operator As String
dim eingabe as String
dim ergebnis As Double

' Gib Programmbeschreibung aus

       MsgBox("Geben Sie zwei Operanden und einen Operator ein!" _
                     & chr$(13) & "Das Programm verhält sich wie ein" _
                     & " Taschenrechner. Als Operatoren sind +, - / und *" _
                     & " zugelassen! Das Programm endet durch die Eingabe von " _
                      & chr$(13) & "0 als erstem Summanden!")

' Lies ersten Summanden ein
       eingabe = InputBox ("Geben Sie nun den ersten Operanden ein!")

       do while eingabe <> "0"
              if Not Lies_und_ueberpruefe_Benutzereingaben(eingabe, _
                     ersterOperand ,operator, zweiterOperand) then exit sub

              if Not Fuehre_Berechnung_durch (ersterOperand, _
                     operator, zweiterOperand, ergebnis) then exit sub


' Gib das Ergebnis aus

              MsgBox (" " & ersterOperand & " "  & operator & " " _
                            & zweiterOperand & " = " & ergebnis)

' Lies ersten Operanden ein

              eingabe = InputBox ("Geben Sie nun den ersten Operanden ein!")

       loop
End Sub

Function  Lies_und_ueberpruefe_Benutzereingaben(ByVal eingabe As String, _
                                   ersterOperand As Double, _
                                   operator  As String, _
                                   zweiterOperand As Double) _
                                   As boolean

              Lies_und_ueberpruefe_Benutzereingaben = true
              if Not wandle_in_Double_um (eingabe, ersterOperand) Then
                     Lies_und_ueberpruefe_Benutzereingaben = false
                     exit function
              end if
              operator = InputBox ("Geben Sie nun den Operator ein!")

              eingabe = InputBox ("Geben Sie nun den zweiten Operanden ein!")

              if Not wandle_in_Double_um (eingabe, zweiterOperand) Then
                     Lies_und_ueberpruefe_Benutzereingaben = false
                     exit function
              end if

end Function

Function Fuehre_Berechnung_durch (ByVal ersterOperand As Double, _
                           ByVal operator As String, _
                           ByVal zweiterOperand As Double, _
                           ergebnis As Double) _
                           As Boolean
```

```
        Fuehre_Berechnung_durch = true
        Select Case operator
                case "+"
                        ergebnis = ersterOperand + zweiterOperand
                case "-"
                        ergebnis = ersterOperand - zweiterOperand
                case "*"
                        ergebnis = ersterOperand * zweiterOperand
                case "/"
                        if (zweiterOperand = 0) Then
                                MsgBox ("Der Nenner ist 0!")
                                Fuehre_Berechnung_durch = False
                                Exit Function
                        End if
                        ergebnis = ersterOperand / zweiterOperand
                case else
                        MsgBox ("Der eingegebene Operator wird nicht " _
                                & "unterstützt!")
                        Fuehre_Berechnung_durch = False
                        Exit Function
        End Select
End Function
```

12 Arrays (Felder, Vektoren)

12.1 Statische Felder

Wie in jeder anderen Programmiersprache, gibt es auch in VBA Felder (Arrays). Felder fassen Variablen ähnlichen Inhalts unter einem gemeinsamen Oberbegriff zusammen. Sie kennen Felder bereits aus der Mathematik. Betrachten Sie dazu Abbildung 12.1:

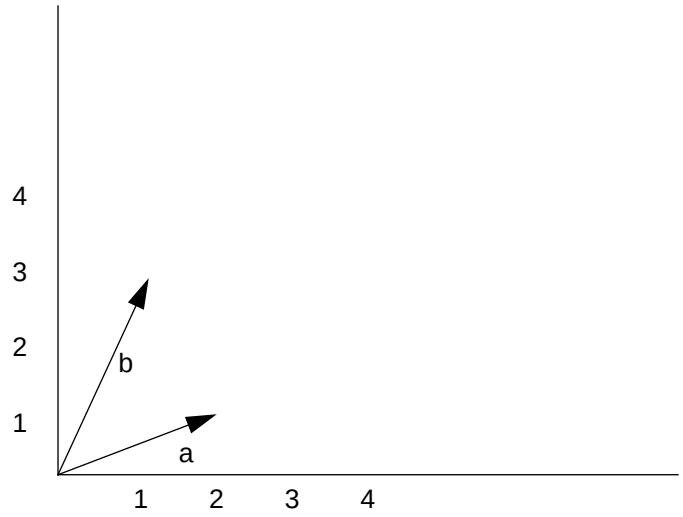

Abbildung 12.1 Koordinatensystem mit 2 Vektoren

Vector a besitzt die Koordinaten (2, 1), b (1,3). Man schreibt dies:

$$a = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$b = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

Auf die Komponenten der Vectoren können wir über ihren Index zugreifen, so ist a(1) = 2, a(2) = 1, b(1) = 1 und b(2) = 3.

Felder sind allerdings nicht nur in der Mathematik sinnvoll. Betrachten wir noch einmal unser Provisionsbeispiel. Wir verändern hier die Aufgabenstellung so, daß die Provision durch den durchschnittlichen Umsatz, gemittelt über die letzten fünf Jahre, bestimmt wird. Ohne Felder benötigen wir fünf Variablen für die Umsätze der vergangenen Jahre. Mit Feldern definieren wir einen Vector mit fünf Elementen und speichen die Umsätze der vergangenen Jahre auf den Elementen des Feldes.

Doch genug der Vorrede. Schauen wir uns Felder an einem Beispiel an. Im Beispiel soll das arithmetische Mittel von drei einzugebenden Zahlen berechnet werden. Zum Schluß soll das arithmetische Mittel zusammen mit den eingegebenen Zahlen ausgegeben werden. Da wir die Zahlen, deren arithmetisches Mittel wir berechnen, mit ausgeben sollen, müssen die Eingaben im Programm abgespeichert werden. Dafür nutzen wir ein Feld.

Beispiel 12.1 Berechnung des arithmetischen Mittels mit einem Feld

```
sub arithmetischesMittelBerechnen1()
' Programm berechnet arithmetischesMittel
' von drei eingegebenen Zahlen
' Dateiname: arithmetischesMittelBerechnen1

       dim eingaben(2) As Double
       dim i As Integer
       dim summe As Double
       dim arithmetischesMittel As Double
       dim ausgabe As String

       MsgBox("Dieses Programm berechnet das arithmetische Mittel von " _
              & chr$(13) & "drei eingegebenen Zahlen!")
       summe = 0
       for i = 0  To 2
              eingaben(i)=InputBox("Geben Sie nun die " & i + 1 & "-te Zahl ein!")
              summe = summe + eingaben(i)
       next i
       arithmetischesMittel = summe / 3

'bastle Ausgabestring zusammen
       ausgabe = "Das arithmetische Mittel der Zahlen: "

'jetzt das Feld an die Ausgabe anfügen
       for i = 0 To 2
              ausgabe = ausgabe & chr$(13) & eingaben(i)
       next i

'vervollständigen der Ausgabe
       ausgabe = ausgabe &  chr$(13) & "ist " & arithmetischesMittel & " !"
       MsgBox(ausgabe)
End Sub
```

Durch die Zeile

```
       dim eingaben(2) As Double
```

wird ein Feld mit drei Elementen erzeugt. Drei Elemente deshalb, weil der Feldindex in VBA, wie in den meisten anderen Programmiersprachen, von Null hochgezählt wird. Das Feld *eingaben* hat also die Elemente *eingaben(0)*, *eingaben(1)* und *eingaben(2)*. Auf die Elemente des Feldes wird über den Index zugegriffen. Wir sehen dies an den Zeilen:

```
       eingaben(i)=InputBox("Geben Sie nun die " & i + 1 & "-te Zahl ein!")
       summe = summe + eingaben(i)
```

Das Schöne an Feldern ist, daß wir, wenn wir alle Elemente eines Feldes bearbeiten wollen, dies recht einfach mit der for next-Schleife realisieren können:

```
       for i = 0  To 2
              eingaben(i)=InputBox("Geben Sie nun die " & i + 1 & "-te Zahl ein!")
              summe = summe + eingaben(i)
       next i
```

Gehen wir nun den Code zeilenweise durch. Zunächst erfolgen die Variablendeklarationen:

```
       dim eingaben(2) As Double
```

```
dim i As Integer
dim summe As Double
dim arithmetischesMittel As Double
dim ausgabe As String
```

Die Deklaration eines Feldes ist völlig analog zur Deklaration einfacher Variablen. Der einzige Unterschied besteht darin, daß die Anzahl der Elemente eines Feldes in runden Klammern an den Namen des Feldes angefügt wird. Strengenommen ist das nicht ganz richtig, da, weil der Feldindex bei Null beginnt, ein Feld ein Element mehr, als die Zahl in der Deklaration angibt, aufnehmen kann. Wir sehen weiterhin, daß ein Feld nur Variablen gleichen Typs aufnehmen kann, da der Typ des Feldes für das ganze Feld in der Deklaration festgelegt wird. Die Zeile

```
MsgBox("Dieses Programm berechnet das arithmetische Mittel von " _
       & chr$(13) & "drei eingegebenen Zahlen!")
```

informiert den Benutzer mit einer `MsgBox` über den Zweck des Programms. Dann beginnt die eigentliche Verarbeitung. Durch

```
summe = 0
```

wird die Variable *summe* mit 0 initialisiert. Dann beginnt eine `for next`-Schleife:

```
for i = 0  To 2
```

In der Schleife werden zunächst die Zahlen, deren arithmetisches Mittel gebildet werden soll, eingelesen.

```
eingaben(i)=InputBox("Geben Sie nun die " & i + 1 & "-te Zahl ein!")
```

Darüber hinaus bilden wir die Summe der eingegebenen Zahlen:

```
summe = summe + eingaben(i)
```

Die Schleife läuft also dreimal. Sie besetzt *eingaben(0)* mit der ersten eingegebenen Zahl, *eingaben(1)* mit der zweiten und *eingaben(2)* mit der dritten.

Dann berechnen wir das arithmetische Mittel:

```
arithmetischesMittel = summe / 3
```

Nun müssen wir das Ergebnis und die Zahlen, die zum Ergebnis führten, ausgeben. Das ist ein bißchen trickreich. Wir benutzen eine `String`-Variable (*ausgabe*), um den Ausgabestring zusammenzustellen. Durch

```
ausgabe = "Das arithmetische Mittel der Zahlen: "
```

wird *ausgabe* mit dem String "Das arithmetische Mittel der Zahlen: " vorbelegt. In einer `for next`-Schleife fügen wir den Inhalt der Elemente des Feldes an den `String` an. Dies geschieht durch den &-Operator (vgl. Kapitel 6.2.4). Gleichzeitig erzeugen wir dort für jedes Element des Feldes einen Zeilenvorschub in der `MsgBox`:

```
for i = 0 To 2
       ausgabe = ausgabe & chr$(13) & eingaben (i)
next i
```

Dann fügen wir das berechnete arithmetische Mittel an den `String` an:

```
ausgabe = ausgabe &  chr$(13) & "ist " & arithmetischesMittel & " !"
```

Zum Schluß erzeugen wir die `MsgBox`.

```
MsgBox(ausgabe)
```

Abbildung 12.2 zeigt einen beispielhaften Durchlauf durch Beispiel 12.1.

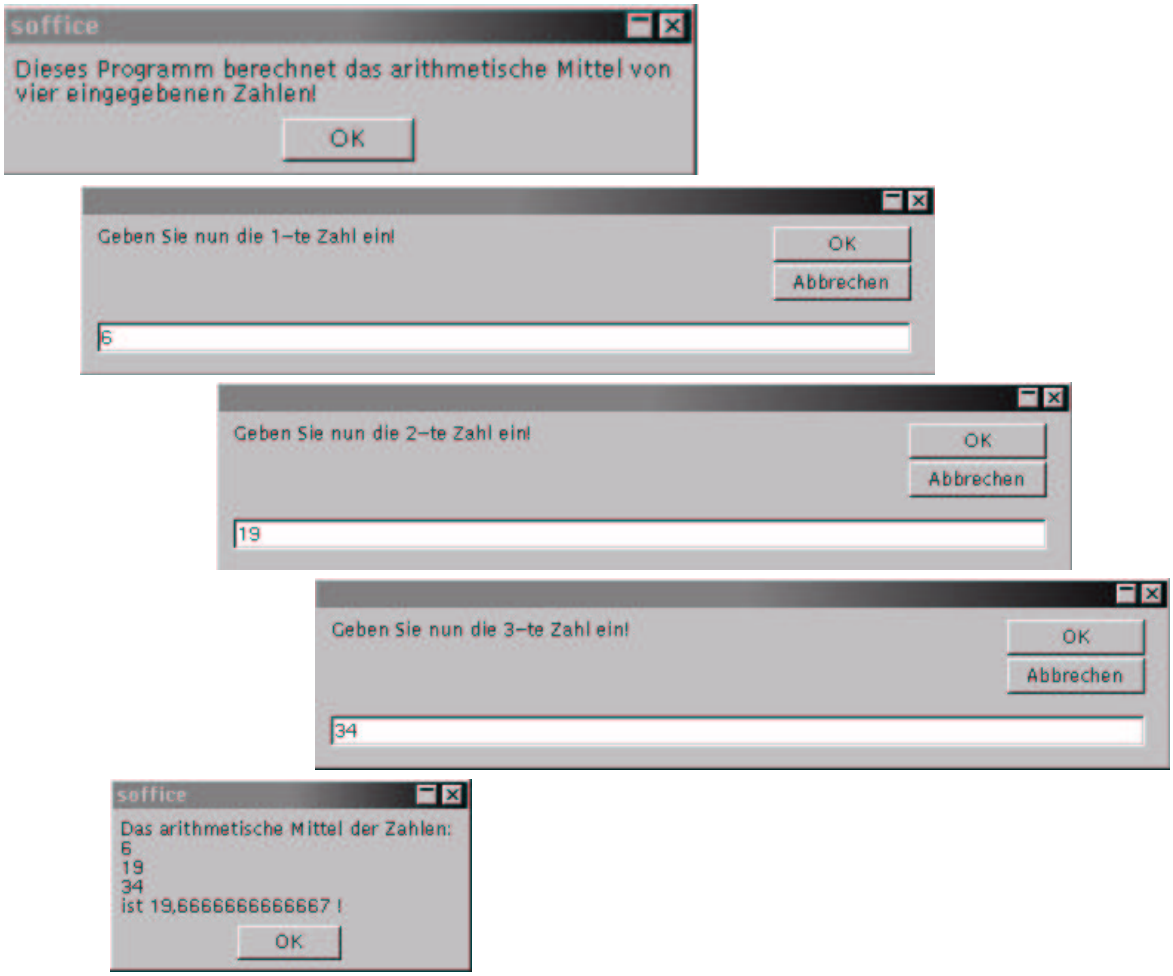

Abbildung 12.2 Ein- und Ausgabe von Beispiel 12.1

Zusammenfassend können wir also sagen:

Felder sind Listen von Variablen ähnlichen Inhalts. Dabei gilt:

- Jedes Element eines Feldes entspricht einer Variablen.
- Auf die einzelnen Elemente wird über einen Index zugegriffen. Der Index zählt von Null hoch.
- Einem Feld kann jeder in VBA verfügbare Datentyp zugewiesen werden. Da der Datentyp aber dem Feld als Ganzem zugewiesen wird, müssen alle Elemente des Feldes von eben diesem Datentyp sein.

- Die Anzahl der Elemente des Feldes minus Eins kann bei der Deklaration des Feldes in runden Klammern an den Feldnamen angefügt werden[35].

12.2 Dynamische Felder

In Kapitel 12.1 wird die Größe des Feldes zu Beginn des Programms festgelegt.

```
dim eingaben(2) As Double
```

Das Feld `eingaben` kann danach drei Elemente aufnehmen. Nehmen wir an, die Benutzeranforderungen ändern sich. Wir sollen einlesen, von wieviel Zahlen das arithmetische Mittel gebildet werden soll. Dann sollen wir es berechnen und anschließend gemeinsam mit den Eingaben ausgeben.

Diese Anforderung stellt uns vor Probleme. Wir wissen die Größe des Feldes nämlich erst nach der ersten Eingabe des Benutzers. Für solche Problematiken stellt VBA das Schlüsselwort `redim` zur Verfügung. Wir veranschaulichen uns die Vorgehensweise an Beispiel 12.2.

Beispiel 12.2 Berechnung des arithmetischen Mittels mit einem dynamischen Feld

```
sub arithmetischesMittelBerechnen2()
' Programm berechnet arithmetischesMittel
' von beliebig vielen eingegebenen Zahlen
' Dateiname: arithmetischesMittelBerechnen2

       dim eingaben() As Double
       dim i As Integer
       dim summe As Double
       dim arithmetischesMittel As Double
       dim grenze As Integer
       dim ausgabe As String

       MsgBox("Dieses Programm berechnet das arithmetische Mittel von " _
              & chr$(13) & "beliebig vielen eingegebenen Zahlen!")
       grenze = InputBox("Von wieviel Zahlen soll das arithmetische Mittel "_
                                          & "berechnet werden?")
       redim eingaben(grenze-1) As Double
       summe = 0
       for i = 0  To grenze - 1
              eingaben(i)=InputBox("Geben Sie nun die " & i + 1 & "-te Zahl ein!")
              summe = summe + eingaben(i)
       next i
       arithmetischesMittel = summe / grenze

'bastle Ausgabestring zusammen
       ausgabe = "Das arithmetische Mittel der Zahlen: "

'jetzt das Feld an die Ausgabe anfügen
       for i = 0 To grenze - 1
              ausgabe = ausgabe & chr$(13) & eingaben (i)
       next i

'vervollständigen der Ausgabe
       ausgabe = ausgabe &  chr$(13) & "ist " & arithmetischesMittel & " !"
       MsgBox(ausgabe)
End Sub
```

35.Kapitel 12.2. zeigt eine andere Möglichkeit, Felder zu deklarieren und ihre Größe festzulegen.

Das Feld *eingaben* wird zunächst ohne Größenangabe deklariert.

```
dim eingaben() As Double
```

Nachdem die Anzahl Zahlen eingegeben wurde, deren arithmetisches Mittel wir berechnen wollen,

```
grenze = InputBox("Von wieviel Zahlen soll das arithmetische Mittel "_
                                    & "berechnet werden?")
```

erzeugen wir ein Feld mit der richtigen Größe:

```
redim eingaben(grenze-1) As Double
```

Beachten Sie, daß von der eingegebenen Größe des Feldes Eins abgezogen werden muß, da der Index der Felder von Null hochgezählt wird. Auch unsere Einlese- und Additionsschleifen laufen jetzt bis zu der vom Benutzer eingegebenen Grenze.

```
for i = 0  To grenze - 1
```

Der Rest des Programms entspricht Beispiel 12.1.

Abbildung 12.2 zeigt einen beispielhaften Durchlauf durch Beispiel 12.2

Da wir in VBA an beliebigen Stellen im Programm deklarieren können, benötigen wir `redim` eigentlich nicht. Folgende Realisierung erzeugt dieselben Ergebnisse:

Beispiel 12.3 Berechnung des arithmetischen Mittels von beliebig vielen Werten ohne `redim`

```
sub arithmetischesMittelBerechnen3()
' Programm berechnet arithmetischesMittel
' von beliebig vielen eingegebenen Zahlen
' Dateiname: arithmetischesMittelBerechnen3


       dim i As Integer
       dim summe As Double
       dim arithmetischesMittel As Double
       dim grenze As Integer
       dim ausgabe As String

       MsgBox("Dieses Programm berechnet das arithmetische Mittel von " _
              & chr$(13) & "beliebig vielen eingegebenen Zahlen!")
       grenze = InputBox("Von wieviel Zahlen soll das arithmetische Mittel "_
                                    & "berechnet werden?")
       dim eingaben(grenze-1) As Double
       summe = 0
       for i = 0  To grenze - 1
              eingaben(i)=InputBox("Geben Sie nun die " & i + 1 & "-te Zahl ein!")
              summe = summe + eingaben(i)
       next i
       arithmetischesMittel = summe / grenze

'bastle Ausgabestring zusammen
       ausgabe = "Das arithmetische Mittel der Zahlen: "

'jetzt das Feld an die Ausgabe anfügen
       for i = 0 To grenze - 1
```

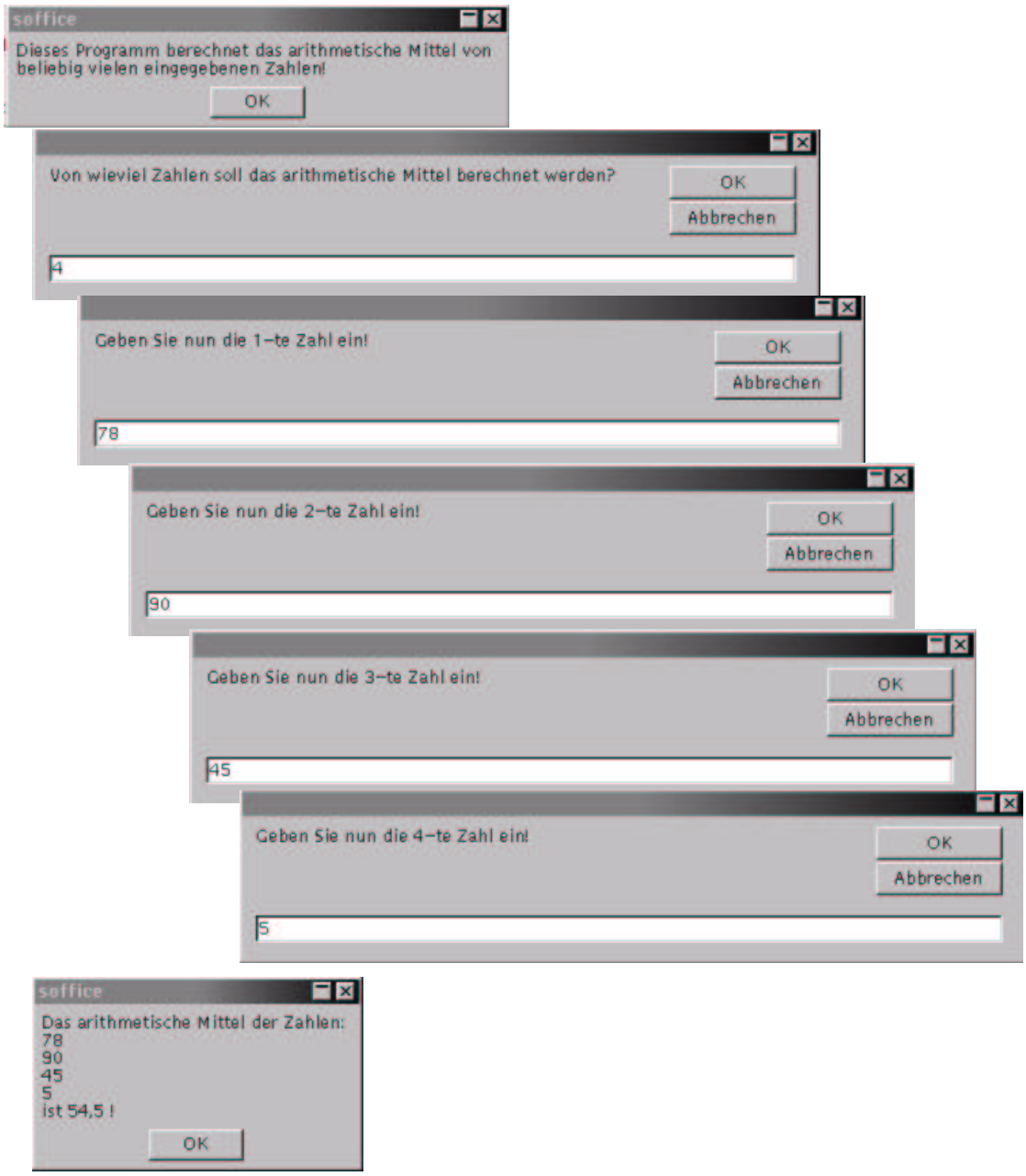

Abbildung 12.3 Ein- und Ausgabe von Beispiel 12.2

```
            ausgabe = ausgabe & chr$(13) & eingaben (i)
        next i

'vervollständigen der Ausgabe
        ausgabe = ausgabe &  chr$(13) & "ist " & arithmetischesMittel & " !"
        MsgBox(ausgabe)
End Sub
```

Hier verschieben wir einfach die Deklaration des Feldes hinter das Einlesen der Anzahl der Zahlen, von denen wir das arithmetische Mittel berechnen wollen:

```
        grenze = InputBox("Von wieviel Zahlen soll das arithmetische Mittel "_
                                        & "berechnet werden?")
```

```
dim eingaben(grenze-1) As Double
```

Der Rest von Beispiel 12.3 stimmt mit Beispiel 12.2 überein. Beispiel 12.3 sieht einfacher aus. Außerdem ist es kürzer. Der Unterschied liegt im Verhalten der Funktion `erase`. `erase` ist eine von VBA zur Verfügung gestellte Funktion, die wir noch nicht kennengelernt haben. `erase` löscht den Inhalt eines Feldes.

Felder, die ohne Größenangabe deklariert werden, heißen dynamische Felder. `erase`, angewendet auf ein dynamisches Feld, löscht das gesamte Feld und gibt den Patz im Hauptspeicher, den das Feld belegt hat, wieder frei. Vor einer erneuten Benutzung muß es mit `redim` neu erzeugt werden.

Felder, die mit Größenangabe deklariert werden, heißen statische Felder. `erase`, angewendet auf ein statisches Feld, setzt die Elemente des Feldes auf Standardwerte zurück. Elemente numerischer Felder werden auf Null zurückgesetzt, Elemente von `String`-Feldern auf den leeren `String` "". Statische Felder können sofort wieder benutzt werden.

12.3 Übergabe von Feldern und die Funktionen LBound und UBound

Felder können ebenso wie Variablen an Prozeduren oder Funktionen übergeben werden. Wir veranschaulichen uns dies an folgendem Beispiel: Die Berechnung eines arithmetischen Mittels ist sicher etwas, was man in bestimmten Anwendungsbereichen häufiger benötigt. So etwas ist mithin ein guter Kandidat für eine Funktion, die wir dann immer wieder verwenden können. Ziel ist es also, die Berechnung des arithmetischen Mittels in Beispiel 12.3 in eine Funktion auszulagern. Die Funktion soll möglichst allgemein verwendbar sein. Dies bedeutet, wir möchten der Funktion nur das Feld mit den Zahlen, über die das arithmetische Mittel gebildet wird, übergeben und erwarten dann das arithmetische Mittel als Rückgabewert der Funktion.

Schauen wir uns die Realisierung an:

Beispiel 12.4 Berechnung des arithmetischen Mittels mit einer Funktion

```
sub arithmetischesMittelBerechnen4()
' Programm berechnet arithmetischesMittel
' von beliebig vielen eingegebenen Zahlen
' Dateiname: arithmetischesMittelBerechnen4


       dim i As Integer
       dim arithmetischesMittel As Double
       dim grenze As Integer
       dim ausgabe As String

       MsgBox("Dieses Programm berechnet das arithmetische Mittel von " _
              & chr$(13) & "beliebig vielen eingegebenen Zahlen!")
       grenze = InputBox("Von wieviel Zahlen soll das arithmetische Mittel "_
                                          & "berechnet werden?")
       dim eingaben (grenze-1) As Double

       for i = 0  To grenze - 1
              eingaben(i)=InputBox("Geben Sie nun die " & i + 1 & "-te Zahl ein!")
       next i
```

```
        arithmetischesMittel = berechneArithmetischesMittel(eingaben())

'bastle Ausgabestring zusammen
        ausgabe = "Das arithmetische Mittel der Zahlen: "

'jetzt das Feld an die Ausgabe anfügen
        for i = 0 To grenze - 1
                ausgabe = ausgabe & chr$(13) & eingaben (i)
        next i

'vervollständigen der Ausgabe
        ausgabe = ausgabe &  chr$(13) & "ist " & arithmetischesMittel & " !"
        MsgBox(ausgabe)
End Sub

Function  berechneArithmetischesMittel(feld() As Double)
        dim untereGrenze As Integer
        dim obereGrenze As Integer
        dim i As Integer
        dim summe As Double

        untereGrenze=Lbound (feld)
        obereGrenze=Ubound (feld)
        summe = 0

        for i = untereGrenze  To obereGrenze
                summe = summe + feld(i)
        next i

        berechneArithmetischesMittel = summe/(obereGrenze - untereGrenze + 1)
End Function
```

Beginnen wir mit der Funktion `berechneArithmetischesMittel`. In der Deklaration der Funktion taucht das Feld wie ein gewöhnlicher Übergabeparameter auf:

```
Function  berechneArithmetischesMittel(feld() As Double)
```

Wie bei der Deklaration eines Feldes innerhalb einer Funktion oder Prozedur werden dem Namen des Feldes runde Klammern hinzugefügt. Anschließend folgt der Typ des Feldes. Beachten Sie, daß Sie hier die Größe des Feldes nicht angeben können. Wir wissen schließlich nicht, wie groß das Feld sein wird, das der Funktion übergeben wird. Und wir wollen ja nun auch eine Funktion schreiben, die mit Feldern beliebiger Größe umgehen kann.

Zu Beginn der Funktion werden die lokalen Variablen der Funktion deklariert.

```
        dim untereGrenze As Integer
        dim obereGrenze As Integer
        dim i As Integer
        dim summe As Double
```

Danach müßten wir mit der Berechnung des arithmetischen Mittels beginnen. Wir haben jedoch ein kleines Problem. Wir wissen nicht, wie groß das uns übergebene Feld ist.

Hierfür stellt uns VBA aber die beiden Funktionen `LBound` (Lower Bound) und `UBound` (Upper Bound) zur Verfügung. `UBound` gibt den größtmöglichen oberen Index eines

Feldes zurück, `LBound` den Kleinstmöglichen[36]. Wie wenden also diese Funktionen auf unser Feld an:

```
untereGrenze=Lbound (feld)
obereGrenze=Ubound (feld)
```

Jetzt können wir auch unsere Schleife programmieren:

```
for i = untereGrenze  To obereGrenze
       summe = summe + feld(i)
next i
```

Beachten Sie, das *untereGrenze* in unseren bisherigen Beispielen immer Null war und *obereGrenze* der tatsächlich größtmögliche Index ist, so daß wir hier nicht Eins abziehen müssen.

Durch

```
berechneArithmetischesMittel = summe/(obereGrenze - untereGrenze + 1)
```

weisen wir der Funktion ihren Rückgabewert zu.

Wenden wir uns nun dem Hauptprogramm zu. Der Beginn des Programms bis zur `for next`-Schleife entspricht Beispiel 12.3. In der `for next`-Schleife lesen wir das Feld nur ein, da wir die Berechnung des arithmetischen Mittels ja in eine Funktion ausgelagert haben:

```
for i = 0  To grenze - 1
       eingaben(i)=InputBox("Geben Sie nun die " & i + 1 & "-te Zahl ein!")
next i
```

Dann erfolgt der Aufruf der Funktion:

```
arithmetischesMittel = berechneArithmetischesMittel(eingaben())
```

Hier werden dem Namen des Feldes nur runde Klammern angefügt, um anzudeuten, daß dieser Übergabeparameter ein Feld ist. Selbstverständlich müssen, wie bei allen Übergabeparametern, die Typen von Aktualparameter (der Typ des Feldes beim Aufruf des Feldes) und Formalparameter (der Typ des Feldes in der Funktionsdeklaration) übereinstimmen.

Der Rest von Beispiel 12.4 entspricht wieder Beispiel 12.3.

Felder werden übrigens immer by reference übergeben. Das Schlüsselwort `ByVal` ist bei Feldern nicht erlaubt.

12.4 Felder mit beliebigen Indexen

Die Indices von Feldern müssen nicht von Null ausgehend nummeriert werden. Man kann bei der Deklaration des Feldes den Bereich, in dem der Index variieren kann angeben. Dies zeigt

Beispiel 12.5 Felder mit beliebigen Indexwerten

36.Der ist nach Ihrem jetzigen Wissensstand Null, da der Index von Feldern von Null ab hochgezählt wird. In Kapitel 12.4 werde ich aber zeigen, wie man das ändern kann.

```
sub arithmetischesMittelBerechnen5()
' Programm berechnet arithmetischesMittel
' von beliebig vielen eingegebenen Zahlen
' Dateiname: arithmetischesMittelBerechnen5


       dim i As Integer
       dim arithmetischesMittel As Double
       dim grenze As Integer
       dim ausgabe As String

       MsgBox("Dieses Programm berechnet das arithmetische Mittel von " _
              & chr$(13) & "beliebig vielen eingegebenen Zahlen!")
       grenze = InputBox("Von wieviel Zahlen soll das arithmetische Mittel "_
                                          & "berechnet werden?")
       dim eingaben(1 To grenze) As Double

       for i = 1  To grenze
              eingaben(i)=InputBox("Geben Sie nun die " & i & "-te Zahl ein!")
       next i

       arithmetischesMittel = berechneArithmetischesMittel(eingaben())

'bastle Ausgabestring zusammen
       ausgabe = "Das arithmetische Mittel der Zahlen: "

'jetzt das Feld an die Ausgabe anfügen
       for i = 1 To grenze
              ausgabe = ausgabe & chr$(13) & eingaben (i)
       next i

'vervollständigen der Ausgabe
       ausgabe = ausgabe &  chr$(13) & "ist " & arithmetischesMittel & " !"
       MsgBox(ausgabe)
End Sub


Function  berechneArithmetischesMittel(feld() As Double)
       dim untereGrenze As Integer
       dim obereGrenze As Integer
       dim i As Integer
       dim summe As Double

       untereGrenze=Lbound (feld)
       obereGrenze=Ubound (feld)
       summe = 0

       for i = untereGrenze  To obereGrenze
              summe = summe + feld(i)
       next i

       berechneArithmetischesMittel = summe/(obereGrenze - untereGrenze + 1)
End Function
```

Die Grenzen eines Index werden also einfach durch die Eingabe des kleinst- und größtmöglichen Indexwertes getrennt durch das Schlüselwort To festgelegt

```
       dim eingaben(1 To grenze) As Double
```

Dies hat natürlich Konsequenzen für die for next-Schleifen im Hauptprogramm, da diese ja vom kleinst- zum größtmöglichen Indexwert laufen müssen. Wir sehen dies anhand der Einleseschleife

```
for i = 1  To grenze
       eingaben(i)=InputBox("Geben Sie nun die " & i & "-te Zahl ein!")
next i
```

und der Ausgabeschleife:

```
for i = 1 To grenze
       ausgabe = ausgabe & chr$(13) & eingaben (i)
next i
```

In unserer Berechnungsfunktion müssen wir nichts ändern, da diese sich ja mit den beiden Funktionen LBound und UBound die Indexgrenzen selbsttätig besorgt.

Merke: Es ist keine gute Idee, die Indexgrenzen von Feldern zu verschieben. Obwohl die Behandlung der Indices von Feldern in Schleifen einfacher wird, wenn man Indexgrenzen von Feldern mit eins starten läßt, gewöhnt man sich damit inkompatiblen Programmierstil an, da Indices von Feldern in fast allen Programmiersprachen (auf jeden Fall in den wichtigen) mit Null beginnen.

12.5 Das Provisionsbeispiel mit Umsatzhistorie

Wie zu Anfang des Kapitels angekündigt, ändern wir nun die Aufgabenstellung des Provisionsbeispiels wie folgt:

Problemstellung 12.1 Provisionsberechnung mit Umsatzhistorie

Ein VBA-Programm soll die zusätzliche Provision für einen Verkauf eines Vermittlers anhand der über die letzten fünf Jahre gemittelten (arithmetisches Mittel) Jahresumsätze berechnen. Die Provision soll nach folgender Tabelle gewährt werden:

gemittelter Umsatz über die letzten 5 Jahre	Provision in Prozent
bis 100.000	0
100.000 bis 500.000	5
500.000 bis 1.000.000	10
über 1.0000.000	20

Das Programm soll zunächst in einem Fenster eine kurze Bedienungsanleitung ausgeben, dann werden die Umsätze und der jetzige Verkaufsbetrag eingegeben. Danach wird der Provisionbetrag errechnet und ausgegeben.

Das Programm soll die Berechnung mehrerer Provisionen erlauben. Es soll abbrechen, wenn als Verkaufsbetrag "beenden" eingegeben wird.

Wie immer entwickeln wir zunächst den Pseudocode:

Pseudocode 12.1 Provision berechnen mit Umsatzhistorie

```
Gib Programmbeschreibung aus
Lies den verkaufsbetrag ein
do while verkaufsbetrag nicht gleich "beenden"
       lies restliche und überprüfe alle Benutzereingaben
       berechne arithmetisches Mittel der Umsätze
       bestimme Provision in Prozent
       berechne auszuzahlenden Betrag Formel:
       (verkaufsbetrag*provision in prozent)/100
       Gib das Ergebnis aus
       Lies den verkaufsbetrag ein
Loop
```

Durch unsere strukturierte Programmierung halten sich die Änderungen in Grenzen. Wir müssen den Verkaufsbetrag vor der Schleife einlesen, weil ja das Abbruchkriterium nun vom Verkaufsbetrag abhängt. "lies restliche und überprüfe alle Benutzereingaben" muß auch geändert werden, weil dort fünf Jahresumsätze auf ein Feld eingelesen werden sollen. Wie das geht, wissen wir aber auch. Das machen wir mit einer `for next`-Schleife, wie in den anderen Beispielen dieses Kapitels. "berechne arithmetisches Mittel der Umsätze" ist noch einfacher, da benutzen wir die von uns in Beispiel 12.4 programmierte Funktion.

Ja und danach bleibt alles beim Alten. "bestimme Provision in Prozent" muß nicht geändert werden. Dieser Funktion übergeben wir unseren gemittelten Umsatz. Damit wird sie genausogut funktionieren, wie vorher mit dem geplanten Umsatz. Die Funktionen "berechne auszuzahlenden Betrag" und "Gib das Ergebnis aus" benötigen den Umsatz gar nicht, sie bleiben daher auch unverändert. Wir erhalten:

Realisierung 12.1 Provision berechnen mit Umsatzhistorie

```
Sub provisionMitFeldern()

' Programm berechnet Provisionen abhängig
' vom Umsatz der letzten 5 Jahre
' Dateiname: provisionMitFeldern

       dim umsatzFeld(4) As Double
' Der Index eines Feldes wird von 0 hochgezaehlt
       dim umsatzMittel As Double

       dim verkaufsbetrag As Double
       dim auszuzahlendeProvision As Double
       dim eingabe As String



       ' Gib Programmbeschreibung aus

       MsgBox("Geben Sie die Umsätze der letzten 5 Jahre " _
                     & chr$(13) & "und den Verkaufsbetrag ein!" _
                     & chr$(13) & "Das Programm berechnet die" _
                     & " Provision des Vermittlers! Sie beenden das" _
                     & " Programm durch die Eingabe von: beenden" _
                     & chr$(13) & " als Verkaufsbetrag!")



' Lies Verkaufsbetrag ein
```

```
	eingabe = InputBox ("Geben Sie nun den Verkaufsbetrag des Kunden ein!")

	Do while eingabe <> "beenden"

		if Not Lies_und_ueberpruefe_Benutzereingaben(eingabe, _
			umsatzFeld() ,verkaufsbetrag) then exit sub
		umsatzMittel = berechneArithmetischesMittel(umsatzFeld())
		auszuzahlendeProvision = berechne_provision(umsatzMittel, _
					verkaufsbetrag)

		' Gib das Ergebnis aus
		MsgBox ("Die Provision für dieses Geschäft ist: " _
				& auszuzahlendeProvision & " DM")
		' Lies Verkaufsbetrag ein
		eingabe = InputBox ("Geben Sie nun den Verkaufsbetrag des Kunden ein!")
	Loop

End Sub

Function  berechneArithmetischesMittel(feld() As Double)
	dim untereGrenze As Integer
	dim obereGrenze As Integer
	dim i As Integer
	dim summe As Double

	untereGrenze=Lbound (feld)
	obereGrenze=Ubound (feld)
	summe = 0

	for i = untereGrenze  To obereGrenze
		summe = summe + feld(i)
	next i

	berechneArithmetischesMittel = summe/(obereGrenze - untereGrenze + 1)
End Function

Function  Lies_und_ueberpruefe_Benutzereingaben(ByVal eingabe As String, _
					umsatz() As Double, _
					verkaufsbetrag As Double) _
					As boolean
		dim untereGrenze As Integer
		dim obereGrenze As Integer
		dim i As Integer

		Lies_und_ueberpruefe_Benutzereingaben = true
		if Not wandle_in_Double_um (eingabe, verkaufsbetrag) Then
			Lies_und_ueberpruefe_Benutzereingaben = false
			exit function
		end if

		' Einlesen des Feldes
		untereGrenze=Lbound (umsatz)
		obereGrenze=Ubound (umsatz)
		for i = untereGrenze  To obereGrenze
			eingabe = InputBox ("Geben Sie nun den Umsatz ein!")
			if Not wandle_in_Double_um (eingabe, umsatz(i)) Then
				Lies_und_ueberpruefe_Benutzereingaben = false
				exit function
			end if
		next i
end Function

Function berechne_Provision (ByVal umsatz As Double, _
```

```
                              ByVal verkaufsbetrag As Double) _
                              As Double

        dim provisionInProzent As Double

        ' Umsatzgrenzen sind DM-Betraege

        const umsatzGrenze1 As Double = 100000
        const umsatzGrenze2 As Double = 500000
        const umsatzGrenze3 As Double = 1000000

        ' Provisionen in Prozent

        const provisionUmsatzGrenze1 As Double = 5
        const provisionUmsatzGrenze2 As Double = 10
        const provisionUmsatzGrenze3 As Double = 20

        ' Bestimme Provision
        if umsatz >= umsatzGrenze3 Then
               provisionInProzent = provisionUmsatzGrenze3
        elseIf umsatz >= umsatzGrenze2 Then
               provisionInProzent = provisionUmsatzGrenze2
        elseIf umsatz >= umsatzGrenze1 Then
               provisionInProzent = provisionUmsatzGrenze1
        else
               provisionInProzent = 0
        End if

' Berechne die Provision

        berechne_Provision = (verkaufsbetrag * provisionInProzent)/100

end Function
```

Wir definieren zunächst ein Feld mit fünf Elementen, um die Umsätze der zurückliegenden Jahre dort abzuspeichern und eine `Double`-Variable, um dort das arithmetische Mittel vorzuhalten. Diese Variable übergeben wir später an `berechne_provision`:

```
        dim umsatzFeld(4) As Double
' Der Index eines Feldes wird von 0 hochgezaehlt
        dim umsatzMittel As Double
```

Der Rest der Deklarationen bleibt unverändert. Die nächste Änderung ergibt sich beim Aufruf der Funktion `Lies_und_ueberpruefe_Benutzereingaben`. Dieser Funktion wird das *umsatzFeld* übergeben:

```
               if Not Lies_und_ueberpruefe_Benutzereingaben(eingabe, _
                      umsatzFeld() ,verkaufsbetrag) then exit sub
```

Dadurch ändert sich natürlich auch die Implementierung von `Lies_und_ueberpruefe_Benutzereingaben`. Ich werde hier nur die Änderungen gegenüber der Vorversion besprechen.

Zunächst ändert sich die Deklaration der Funktion, weil die neue Funktion ein Feld von Double-Variablen zurückgeben muß.

```
Function  Lies_und_ueberpruefe_Benutzereingaben(ByVal eingabe As String, _
                                        umsatz() As Double, _
```

```
                                        verkaufsbetrag As Double) _
                                        As boolean
```

In der Funktion heißt das Feld für die Umsätze *umsatz*, im Hauptprogramm *umsatzfeld*. Daran erkennen wir, daß die Namen der übergebenen Felder, wie bei allen Übergabeparametern, nicht übereinstimmen müssen. Welche Variable an welche übergeben wird, wird allein durch die Reihenfolge in der Parameterliste bestimmt. Der Typ der Variablen bzw. Felder muß natürlich übereinstimmen.

Darüber hinaus benötigen wir drei neue Variablen, um die Umsätze in einer Schleife einzulesen. Dies ist der Start- und Endwert der Schleife, sowie die Schleifenvariable.

```
dim untereGrenze As Integer
dim obereGrenze As Integer
dim i As Integer
```

Zum Einlesen des Feldes bestimmen wir zunächst den Start- und Endwert der Schleife und beginnen daraufhin die Schleife:

```
' Einlesen des Feldes
untereGrenze=Lbound (umsatz)
obereGrenze=Ubound (umsatz)
for i = untereGrenze  To obereGrenze
```

Dies hätten wir zwar bedeutend einfacher haben können, indem wir die Variablen *untereGrenze* und *obereGrenze* einfach weggelassen und die Schleife mit

```
for i = 0 To 4
```

gestartet hätten. Unsere jetzige Implementierung hat aber den Vorteil, daß wir die Größe des Feldes nur an einer Stelle im Programm codieren, nämlich bei der Deklaration des Umsatzfeldes im Hauptprogramm. An allen anderen Stellen im Programm wird die Größe des Feldes aus dem Umsatzfeld selbst ermittelt. Das erleichtert Änderungen. Ändert sich nämlich irgendwann einmal der Zeitrahmen, über den der Umsatz gemittelt werden soll, so muß diese Änderung nur an einer Stelle des Programmes vorgenommen werden, der ganze Restcode muß nicht betrachtet werden, da ja ab der Deklaration die Größe des Feldes immer aus dem Feld selbst bestimmt wird.

Dann lesen wir den ersten Umsatz ein und überprüfen, ob wir die Eingabe in einen `Double` umwandeln können. Wir sehen hier, daß wir Elemente eines Feldes genau wie Variable behandeln können. Der Funktion `wandle_in_Double_um` wird *umsatz(i)* wie eine normale Variable übergeben. Dies ist auch überhaupt nicht abwegig, denn *umsatz(i)* ist eine normale Variable.

```
        eingabe = InputBox ("Geben Sie nun den Umsatz ein!")
        if Not wandle_in_Double_um (eingabe, umsatz(i)) Then
               Lies_und_ueberpruefe_Benutzereingaben = false
               exit function
        end if
next i
```

Der Rest des Codes stimmt mit unserer bisherigen Implementierung überein.

Zurück im Hauptprogramm rufen wir die bereits in Beispiel 12.4 programmierte Funktion `berechneArithmetischesMittel`. Dies ist ein schönes Beispiel von Wieder-

verwendung. Von nun an können wir die Variable *umsatzMittel* so benutzen, wie in der Vorversion die Variable *umsatz*.

```
umsatzMittel = berechneArithmetischesMittel(umsatzFeld())
```

Damit sind die notwendigen Änderungen am Programm vorgenommen. Abbildung 12.4 zeigt einen beispielhaften Ablauf von Realisierung 12.1.

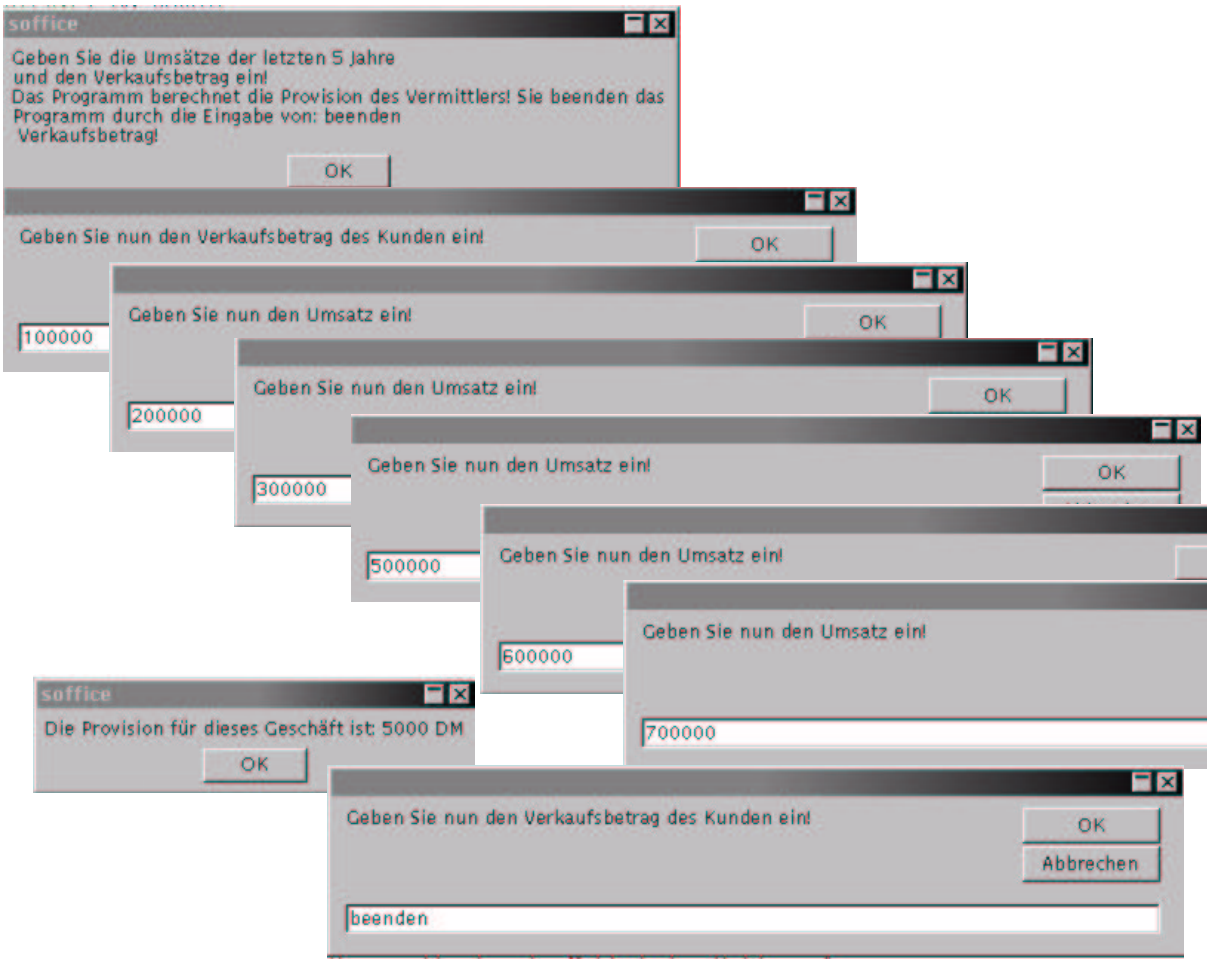

Abbildung 12.4 Ein- und Ausgabe von Realisierung 12.1

13 Integration in Excel

13.1 Benutzerdefinierte Tabellenfunktionen

13.1.1 Erstes einfaches Beispiel

Der einfachste Weg, VBA-Code von Excel aus zu nutzen, sind benutzerdefinierte Tabellenfunktionen. Alle von uns geschriebenen Funktionen tauchen nämlich in Excel unter dem Menüpukt Einfügen -> Funktionen auf. Veranschaulichen wir uns dies an einem Beispiel.

Beispiel 13.1 Allereinfachstes Provisionsbeispiel

```
Option Explicit
Function provision(umsatz As Double) As Double
' Funktion berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provision
    Dim provisionInProzent As Double

    Const umsatzGrenze As Double = 100000

    ' Provisionen in Prozent

    Const provisionUmsatzGrenze As Double = 10

' bestimme Provision in Prozent

    If umsatz >= umsatzGrenze Then
        provisionInProzent = provisionUmsatzGrenze
    Else
        provisionInProzent = 0
    End If

' Berechne auszuzahlenden Betrag

    provision = (umsatz * provisionInProzent) / 100


End Function
```

Beispiel 13.1 erwartet als Eingabe einen Double-Wert und berechnet dann die Provision für das Geschäft. Nach unserem bisherigen Kenntnisstand müßten wir jetzt ein "Hauptprogramm" schreiben, um diese Funktion zu nutzen. Doch Funktionen können auch direkt von einer Excel-Tabelle aus genutzt werden.

Tippen Sie dazu den Umsatz, von dem die Provision berechnet werden soll, in eine beliebige Excel-Zelle. Als nächstes fügen Sie in eine andere Zelle (z.B. direkt darunter) das Gleichheitszeichen ein. Dann clicken Sie zunächst das Menü Einfügen auf und wählen dort Funktionen (vgl. Abbildung 13.1). Nach einem weiteren Click erscheint ein neues Fenster (Abbildung 13.2).

Wie durch Zauberhand existiert dort unsere `Funktion` provision in der Rubrik benutzerdefinierte Funktionen. Wählen Sie `provision` aus und bestätigen Sie (Abbildung 13.3).

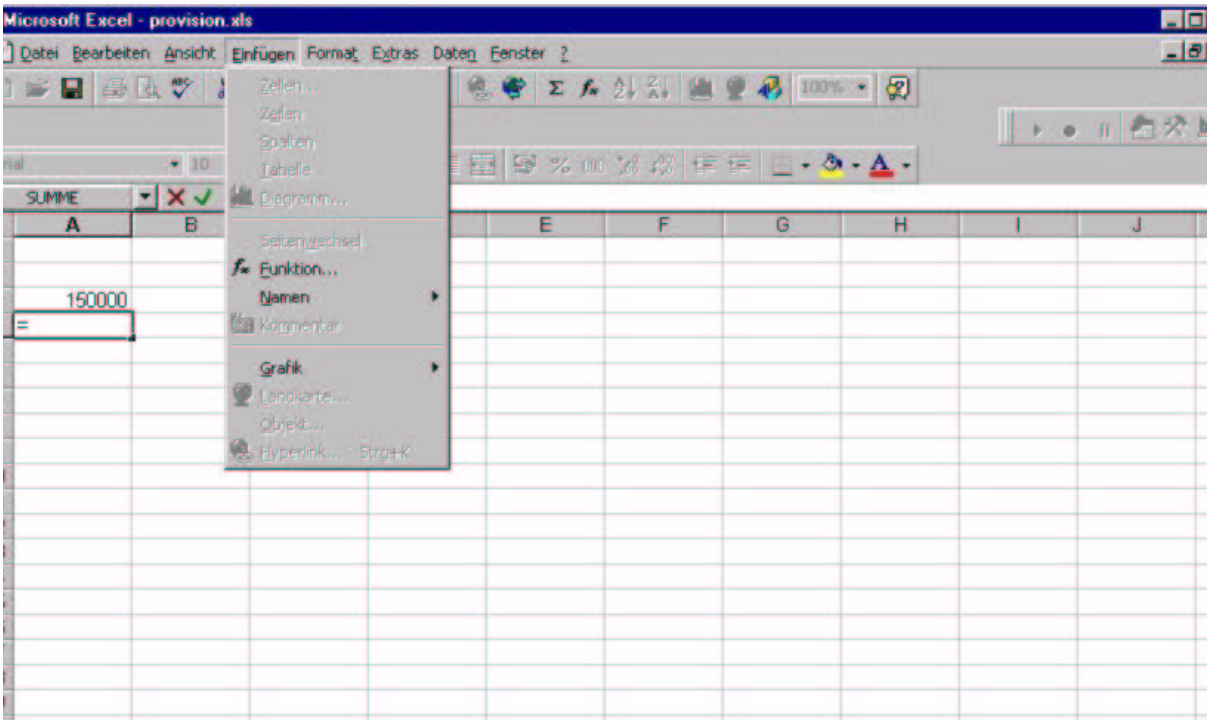

Abbildung 13.1 Das Excel Einfügen-Menü

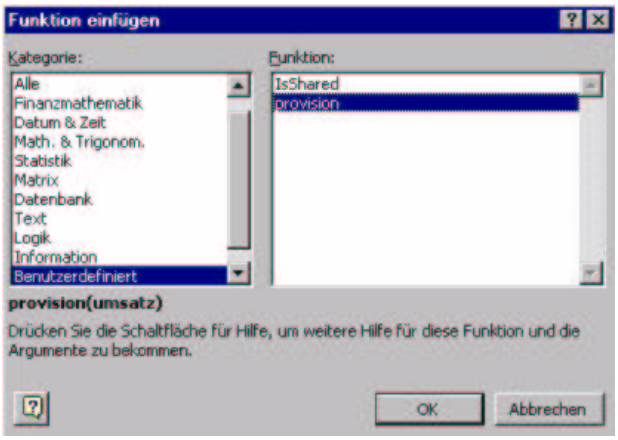

Abbildung 13.2 In Excel verfügbare Funktionen

Die Funktion `provision` wird in die Tabelle übernommen. Nun clicken Sie auf die Zelle, die den Umsatz enthält. Die Zelle (im Beispiel A3) wird als Parameter für die Funktion `provision` übernommen.

Sie lösen aus (drücken der Taste Return). Excel übergibt nun den Inhalt der Zelle, die Sie in die Parameterliste der Funktion übernommen haben (im Beispiel A3) an die Funktion. Die Funktion wird durchgeführt und der Rückgabewert der Funktion wird an Excel übergeben.

Excel stellt den Rückgabewert in der Zelle mit dem Inhalt "=provision(A3)" dar (im Beispiel A4 vgl. Abbildung 13.4).

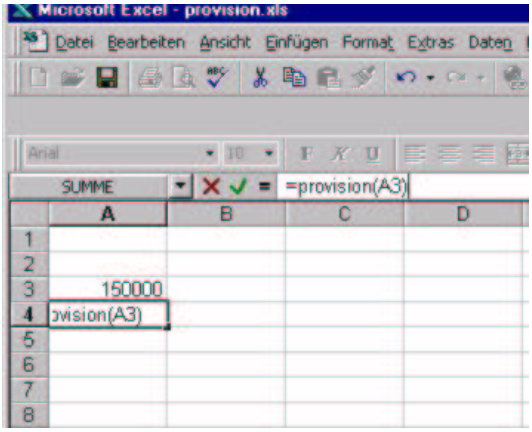

Abbildung 13.3 provision in die Tabelle übernommen

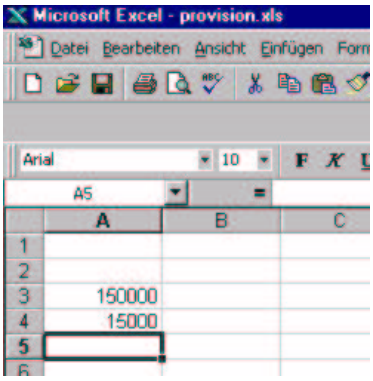

Abbildung 13.4 Das von provision berechnete Ergebnis wird in der Tabelle dargestellt

13.1.2 Nutzung unseres Provisionsbeispiels aus einer Excel-Tabelle

Wir ändern die Aufgabenstellung des Provisionsbeispiels folgendermaßen ab:

Problemstellung 13.1 Provisionsberechnung aus einer Excel-Tabelle

Ein VBA-Programm soll die zusätzliche Provision für einen Verkauf eines Vermittlers anhand des geplanten Jahresumsatzes berechnen. Die Provision soll nach folgender Tabelle gewährt werden:

Umsatz	Provision in Prozent
bis 100.000	0
100.000 bis 500.000	5
500.000 bis 1.000.000	10
über 1.0000.000	20

Das Programm soll als benutzerdefinierte Funktion von einer Excel-Tabelle genutzt werden. Das Benutzerinterface soll Abbildung 13.5 entsprechen. Liegt der in der Zelle B2 eingegebene Verkaufsbetrag, für den die Provision gerade berechnet werden soll,

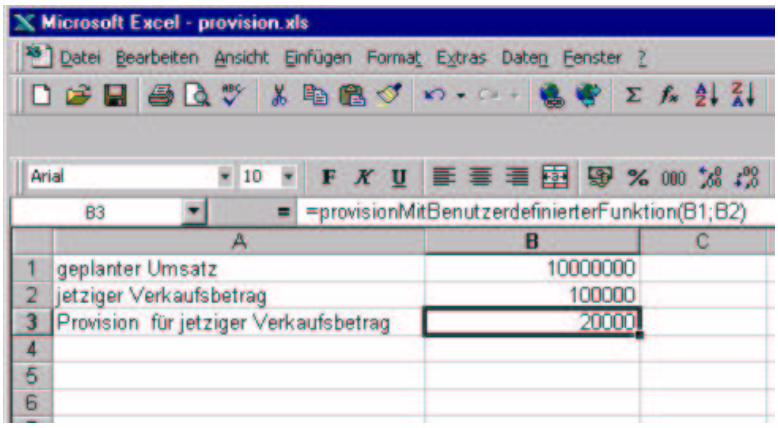

Abbildung 13.5 Das Benutzerinterface der Provisionsanwendung

über dem geplanten Umsatz für das ganze Jahr (Eingabe Zelle B1), soll das Programm eine Fehleingabe vermuten und eine Fehlermeldung ausgeben. Ansonsten wird der Provisionbetrag errechnet und in Zelle B3 ausgegeben.

Um diese Aufgabe zu lösen, betrachten wir zunächst den Pseudocode aus Kapitel 9.

Pseudocode 13.1 Provision berechnen mit Do-While Schleife (erneut wiederholt)

```
Gib Programmbeschreibung aus
Lies den umsatz ein
do while umsatz nicht gleich "beenden"
       lies restliche und überprüfe alle Benutzereingaben
       bestimme Provision in Prozent
       berechne auszuzahlenden Betrag Formel:
       (verkaufsbetrag*provision in prozent)/100
       Gib das Ergebnis aus
       Lies den umsatz ein
Loop
```

"Gib Programmbeschreibung aus" können wir uns kneifen, was gemacht werden soll ist durch das in Abbildung 13.5 dargestellte Benutzerinterface klar genug[37]. Um "Lies den umsatz ein" brauchen wir uns ebenfalls kaum zu kümmern, weil die Übergabe der Parameter durch Excel erfolgt. Wir müssen nur die Prozedur "lies restliche und überprüfe alle Benutzereingaben" in eine Funktion umwandeln. Die Übergabeparameter der neuen Funktion sind der geplante Umsatz und der jetzige Verkaufsbetrag.

Aus "lies restliche und überprüfe alle Benutzereingaben" wird also eine Funktion, die nur die Benutzereingaben überprüft. Wir benennen Sie um in: "überprüfe Benutzereingaben".

"berechne auszuzahlenden Betrag" ist ebenfalls ein Einzeiler. Da dies funktional eng mit "bestimme Provision in Prozent" zusammenhängt, realisieren wir dies zusammen in einer Funktion, der wir den Namen `berechne_Provision` geben. Dies ist bereits in Kapitel 11 geschehen. Die Schleife benötigen wir nicht, da Excel bei jeder Änderung der Eingaben die benutzerdefinierte Funktion erneut ausführt.

37.So zumindest ist die Hoffnung.

"Gib das Ergebnis aus" ist auch überflüssig, weil der Rückgabewert der Funktion sowieso in der Excel-Zelle, die den Funktionsaufruf enthält, dargestellt wird.

Unser Pseudocode reduziert sich also zu:

Pseudocode 13.2 Provisionsberechnung aus einer Excel-Tabelle

```
überprüfe Benutzereingaben
berechne Provision
       beinhaltet:   bestimme Provision in Prozent
                     berechne auszuzahlenden Betrag Formel:
                     (verkaufsbetrag*provision in prozent)/100
```

Die Funktion `berechne_Provision` haben wir bereits in Kapitel 11.9 realisiert, eine Änderung ist nicht notwendig. `überprüfe_Benutzereingaben` muß zwar aus `lies_restliche_und_überprüfe_alle_Benutzereingaben` abgeleitet werden, jedoch ist dies nicht weiter schwierig, weil wir ja nur die `InputBox`-en streichen und dafür einen Parameter mehr übergeben müssen. Diese Gedanken führen zu folgender Realisierung:

Realisierung 13.1 Provisionsberechnung aus einer Excel-Tabelle

```
Function provisionMitBenutzerdefinierterFunktion(umsatzEingabe As String, _
                                                 verkaufsbetragEingabe As String)

' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provisionMitBenutzerdefinierterFunktion

    Dim umsatz As Double
    Dim verkaufsbetrag As Double
    Dim eingabe As String

    If Not ueberpruefe_Benutzereingaben(umsatzEingabe, _
            verkaufsbetragEingabe, umsatz, verkaufsbetrag) Then Exit Function

    provisionMitBenutzerdefinierterFunktion = berechne_Provision(umsatz, _
                                                        verkaufsbetrag)

End Function

Function ueberpruefe_Benutzereingaben(ByVal umsatzEingabe As String, _
                    ByVal verkaufsbetragEingabe As String, _
                    umsatz As Double, _
                    verkaufsbetrag As Double) _
                    As Boolean

        ueberpruefe_Benutzereingaben = True
        If Not wandle_in_Double_um(umsatzEingabe, umsatz) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der erste Parameter (umsatz) der Funktion ist keine Zahl!")
            Exit Function
        End If

        If Not wandle_in_Double_um(verkaufsbetragEingabe, verkaufsbetrag) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der erste Parameter (umsatz) der Funktion ist keine Zahl!")
            Exit Function
        End If

        If verkaufsbetrag > umsatz Then
```

```
            MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
            ueberpruefe_Benutzereingaben = False
            Exit Function
        End If
End Function

 Function berechne_Provision(ByVal umsatz As Double, _
            ByVal verkaufsbetrag As Double) _
            As Double

    Dim provisionInProzent As Double

    ' Umsatzgrenzen sind DM-Betraege

    Const umsatzGrenze1 As Double = 100000
    Const umsatzGrenze2 As Double = 500000
    Const umsatzGrenze3 As Double = 1000000

    ' Provisionen in Prozent

    Const provisionUmsatzGrenze1 As Double = 5
    Const provisionUmsatzGrenze2 As Double = 10
    Const provisionUmsatzGrenze3 As Double = 20

    ' Bestimme Provision
    If umsatz >= umsatzGrenze3 Then
        provisionInProzent = provisionUmsatzGrenze3
    ElseIf umsatz >= umsatzGrenze2 Then
        provisionInProzent = provisionUmsatzGrenze2
    ElseIf umsatz >= umsatzGrenze1 Then
        provisionInProzent = provisionUmsatzGrenze1
    Else
        provisionInProzent = 0
    End If

' Berechne die Provision

    berechne_Provision = (verkaufsbetrag * provisionInProzent) / 100

End Function

Function wandle_in_Double_um(ByVal eingabe As String, rueckgabe As Double) _
                    As Boolean
    wandle_in_Double_um = True
    If Not IsNumeric(eingabe) Then
        MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
        wandle_in_Double_um = False
        Exit Function
    End If
    rueckgabe = CDbl(eingabe)
End Function
```

Wie Sie sehen, waren dies tatsächlich die einzigen Änderungen:

- Aus einer Prozedur wird eine Funktion.
- Die Überprüf-Funktion liest selber nichts mehr ein, sondern bekommt alle zu überprüfenden Variablen übergeben.

- Die Schleife verschwindet.

Beachten Sie allerdings, daß die Übergabeparameter beim Aufruf aus einer Excel-Tabelle durch das Semicolon getrennt werden und nicht wie in VBA selbst durch das Komma (vgl. Abbildung 13.5).

Realisierung 13.1 hat allerdings noch einen Nachteil. In der Rubrik benutzerdefinierte Funktionen des Einfügen --> Funktionen Dialogs sehen wir alle Funktionen von Realisierung 13.1 (vgl. Abbildung 13.6).

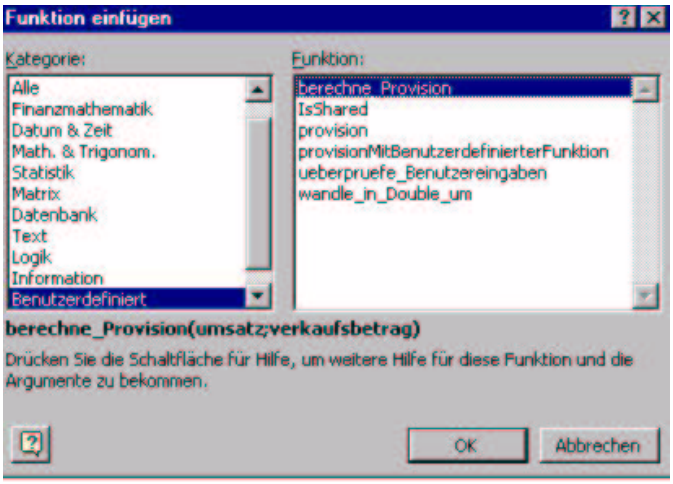

Abbildung 13.6 Die Rubrik benutzerdefinierte Funktionen nach der Implementierung von Realisierung 13.1

So etwas kann diese Rubrik recht schnell füllen, speziell, wenn wir viele "Hilfsfunktionen" implementieren, von denen wir gar nicht wollen, daß sie von Excel aus zugänglich sind.

Die Sichtbarkeit von Funktionen können wir durch das Schlüsselwort `Private` einschränken. Funktionen, denen das Schlüsselwort `Privat` voransteht, sind nur in dem Modul, in dem sie deklariert sind, sichtbar und erscheinen insbesondere nicht in der Rubrik benutzerdefinierte Funktionen des Einfügen --> Funktionen Dialogs. Dies entspricht den Regeln zur Sichtbarkeit von Variablen (vgl. Kapitel 11.11).

Wir können also Realisierung 13.1 folgendermaßen umschreiben:

Realisierung 13.2 Provisionsberechnung aus einer Excel-Tabelle, nur die Hauptfunktion ist sichtbar

```
Function provisionMitBenutzerdefinierterFunktion(umsatzEingabe As String, _
                                                 verkaufsbetragEingabe As String)

' Programm berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provisionMitBenutzerdefinierterFunktion

    Dim umsatz As Double
    Dim verkaufsbetrag As Double
    Dim eingabe As String

    If Not ueberpruefe_Benutzereingaben(umsatzEingabe, _
           verkaufsbetragEingabe, umsatz, verkaufsbetrag) Then Exit Function
```

```
    provisionMitBenutzerdefinierterFunktion = berechne_Provision(umsatz, verkaufsbe-
       trag)

End Function

Private Function ueberpruefe_Benutzereingaben(ByVal umsatzEingabe As String, _
                    ByVal verkaufsbetragEingabe As String, _
                    umsatz As Double, _
                    verkaufsbetrag As Double) _
                    As Boolean

        ueberpruefe_Benutzereingaben = True
        If Not wandle_in_Double_um(umsatzEingabe, umsatz) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der erste Parameter (umsatz) der Funktion ist keine Zahl!")
            Exit Function
        End If

        If Not wandle_in_Double_um(verkaufsbetragEingabe, verkaufsbetrag) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der erste Parameter (umsatz) der Funktion ist keine Zahl!")
            Exit Function
        End If

        If verkaufsbetrag > umsatz Then
            MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
            ueberpruefe_Benutzereingaben = False
            Exit Function
        End If
End Function

 Private Function berechne_Provision(ByVal umsatz As Double, _
            ByVal verkaufsbetrag As Double) _
            As Double

    Dim provisionInProzent As Double

    ' Umsatzgrenzen sind DM-Betraege

    Const umsatzGrenze1 As Double = 100000
    Const umsatzGrenze2 As Double = 500000
    Const umsatzGrenze3 As Double = 1000000

    ' Provisionen in Prozent

    Const provisionUmsatzGrenze1 As Double = 5
    Const provisionUmsatzGrenze2 As Double = 10
    Const provisionUmsatzGrenze3 As Double = 20

    ' Bestimme Provision
    If umsatz >= umsatzGrenze3 Then
        provisionInProzent = provisionUmsatzGrenze3
    ElseIf umsatz >= umsatzGrenze2 Then
        provisionInProzent = provisionUmsatzGrenze2
    ElseIf umsatz >= umsatzGrenze1 Then
        provisionInProzent = provisionUmsatzGrenze1
    Else
        provisionInProzent = 0
    End If

' Berechne die Provision
```

```
    berechne_Provision = (verkaufsbetrag * provisionInProzent) / 100

End Function

 Private Function wandle_in_Double_um(ByVal eingabe As String, rueckgabe As Double) _
                    As Boolean
    wandle_in_Double_um = True
    If Not IsNumeric(eingabe) Then
        MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
        wandle_in_Double_um = False
        Exit Function
    End If
    rueckgabe = CDbl(eingabe)
End Function
```

Nun wird nur noch `provisionMitBenutzerdefinierterFunktion` in der Rubrik benutzerdefinierte Funktionen des Einfügen --> Funktionen Dialogs dargestellt (vgl. Abbildung 13.7)

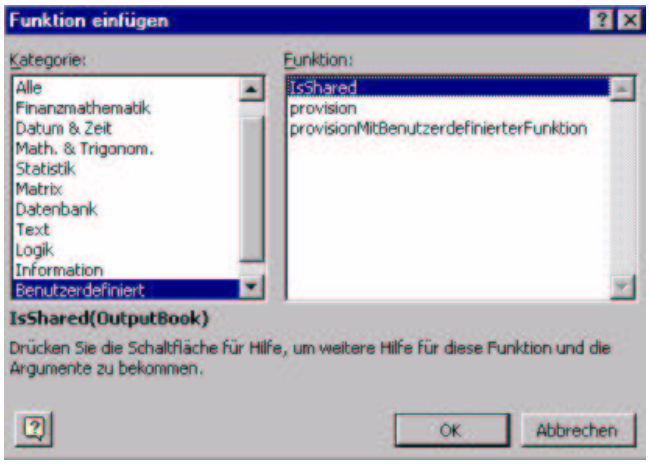

Abbildung 13.7 Die Rubrik benutzerdefinierte Funktionen nach der Implementierung von Realisierung 13.2

13.2 Ereignisgesteuerte Prozeduren

13.2.1 Erster ereignisgesteuerter Prozeduraufruf

Neben der Nutzung benutzerdefinierter Funktionen gibt es eine weitere Möglichkeit, VBA-Code in Excel ablaufen zu lassen. Excel kann auf Ereignisse reagieren. Ereignisse sind im weitesten Sinne alle Dinge, die Sie in Excel durchführen können. Wenn Sie einen Menüeintrag auswählen oder auf ein Symbol in einer Symbolleiste clicken, so ist das für Excel ein Ereignis. Solche Ereignisse können mit VBA-Code (in diesem Fall mit Prozeduren) verbunden werden. Der Code wird dann ausgeführt, wenn das Ereignis eintritt.

Sie können sich aber auch eigene Ereignisse definieren und mit VBA-Code verbinden. Zu diesem Zweck gibt es in Excel Steuerelemente. Das einfachste Excel-Steuerelement ist eine Schaltfläche (Button). Eine solche Schaltfläche können Sie auf ein Tabellenblatt positionieren. Dann können Sie sie beschriften. Sie verbinden die Schaltfläche

mit VBA-Code. Von nun an wird immer, wenn der Benutzer auf die Schaltfläche clickt, der zugehörige Code ausgeführt.

Um eine Schaltfläche zu erzeugen, müssen Sie zunächst das Steuerelemente Menü öffnen. Sie tun dies, indem Sie in der Visual Basic-Symbolleiste das Steuerelemente Symbol anclicken (vgl. Abbildung 13.8).

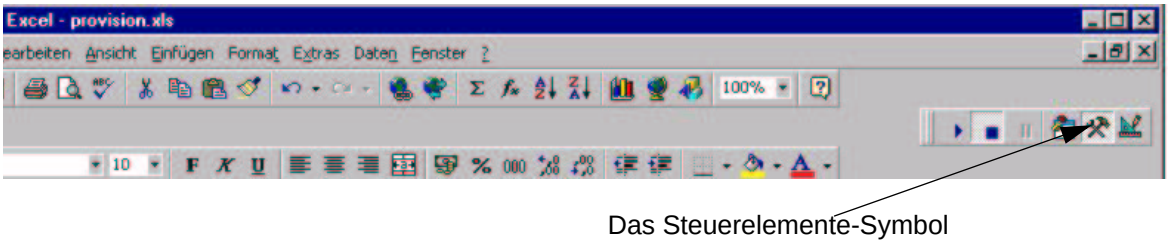

Abbildung 13.8 Erzeugung des Steuerelemente-Menüs

Die Steuerelemente Toolbox erscheint (vgl. Abbildung 13.9).

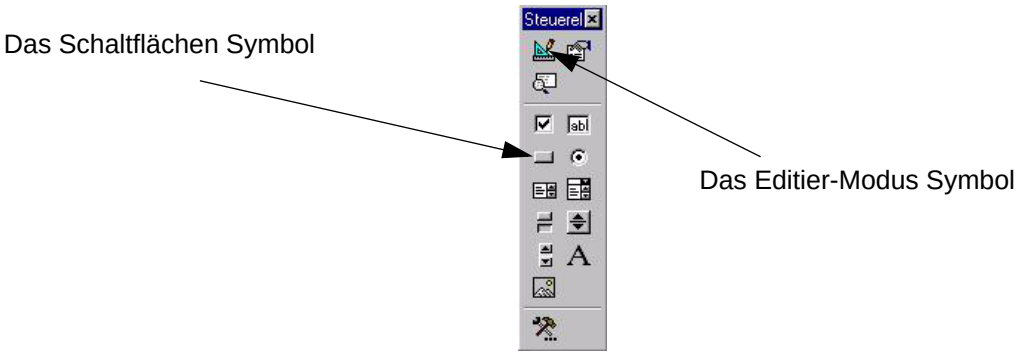

Abbildung 13.9 Das Steuerelemente-Menü mit Schaltflächen Symbol

Sie markieren das Schaltflächen-Symbol und zeichnen eine Schaltfläche auf das Tabellenblatt. Über das Kontextmenü (rechte Maustaste) oder F4 können Sie die Eigenschaften der Schaltfläche verändern. Das wichtigste ist der Name der Schaltfläche. Voreingestellt ist CommandButton1. Sie ändern dies in einen aussagekräftigen Namen[38].

Doppel-Clicken Sie nun auf ihre neue Schaltfläche. Excel erzeugt ein neues VBA-Modul für Ihre Arbeitsmappe. Innerhalb des Moduls erstellt Excel das Skelett einer Prozedur. Die Prozedur heißt `ihrName_Click()`. Zwischen dem Prozedurname und `End Sub` fügen Sie nun (wie immer) Ihren VBA-Code ein (vgl. Abbildung 13.10). Immer wenn Sie im Tabellenblatt auf die Schaltfläche clicken, wird die zugehörige Prozedur durchgeführt.

Merke: Wenn Sie irgendetwas an der Schaltfläche verändern wollen und sei es, daß Sie die Schaltfläche auf dem Tabellenblatt verschieben möchten, dann

38.Sie wissen ja, alle Namen die wir in der Anwendungsentwicklung vergeben, sollen Rückschlüsse auf den Sinn des Objekts, das den Namen erhalten hat, erlauben.

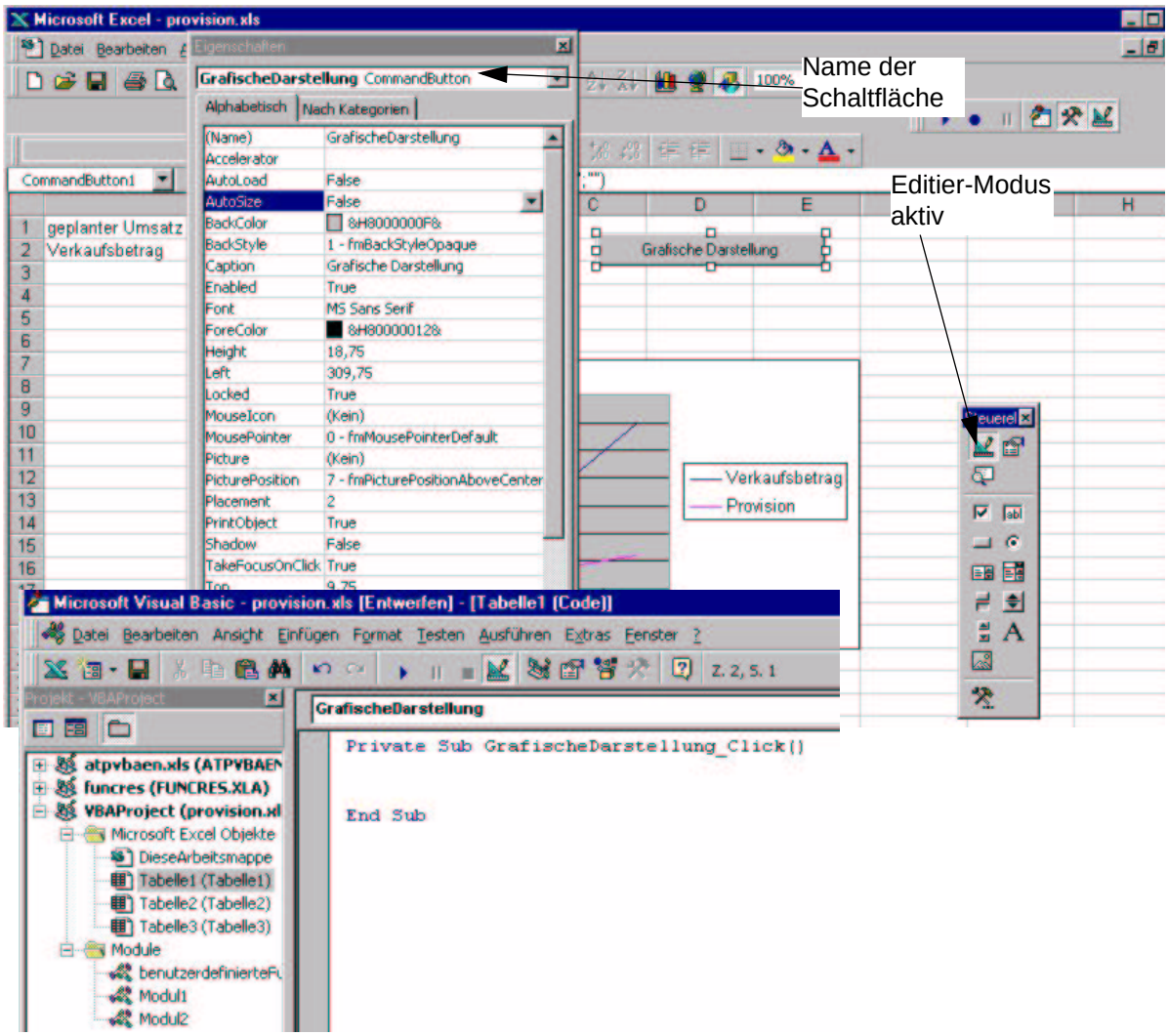

Abbildung 13.10 Schaltfläche mit von Excel erzeugtem Prozedur-Skelett

müssen Sie die Schaltfläche vorher in den Editiermodus bringen. Dies tun Sie indem Sie das Icon des Editiermodus aktivieren (vgl. Abbildung 13.9).

Veranschaulichen wir uns das bisher Gesagte an einem Beispiel. Nehmen wir an, Sie haben mit Hilfe Ihrer benutzerdefinierten Funktion aus Kapitel 13.1.2 einige Provisionen berechnet. Sie möchten nun Umsätze und zugehörige Provisionen grafisch darstellen.

Um dies für alle Zukunft zu automatisieren, möchten Sie eine Schaltfläche erstellen. Immer wenn Sie auf die Schaltfläche clicken, soll die grafische Darstellung erfolgen (vgl. Abbildung 13.11).

Hier haben wir zwei Probleme:

- Erzeugung der Schaltfläche und verbinden mit VBA-Code: Dies habe ich gerade dargestellt. Ich denke, daß Sie diese Schritte durch Probieren doch hinbekommen würden.

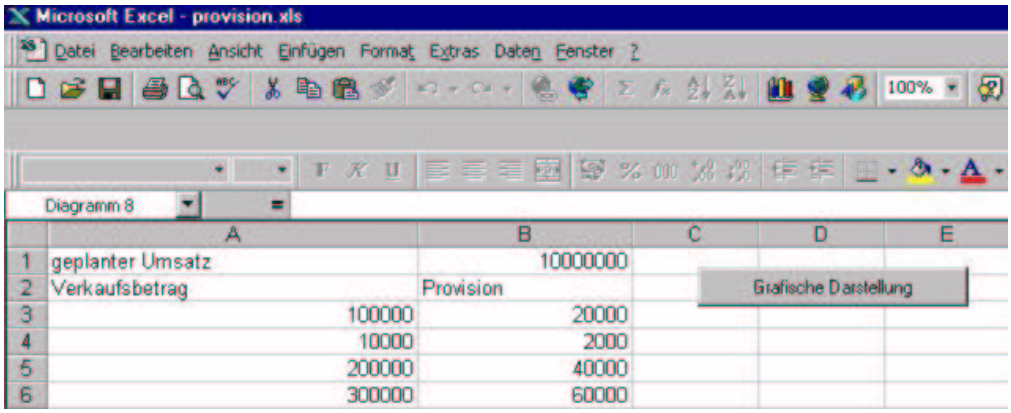

Abbildung 13.11 Schaltfläche zur Erzeugung eines Diagramms

- Programmierung der Diagramm-Darstellung: Dazu müssen wir auf das in Excel für Diagramme zuständige Objekt zugreifen. Sie haben ja bereits gelernt, was Objekte und was Methoden und Eigenschaften von Objekten sind. Allerdings wissen wir weder den Namen des Diagramm-Objekts, noch kennen wir seine Methoden. Theoretisch müßten wir jetzt in den diversen Excel-Hilfen suchen oder bei Microsoft anrufen. Beides ist nicht so unbedingt erstrebenswert. Glücklicherweise gibt es eine elegantere Lösung. Wir können uns von Excel zeigen lassen, wie man Diagramme programmiert. Dazu gibt es die Excel Funktion "Makro Aufzeichen". Wir schalten "Makro Aufzeichen" an. Dann erstellen wir "von Hand" unser Diagramm. Wir beenden "Makro Aufzeichen". Excel erstellt für alle unsere Aktionen VBA-Kommandos und speichert diese als Prozedur in einem neuen Modul der Excel-Arbeitsmappe ab.

Doch gehen wir den Vorgang im Einzelnen durch. Zunächst schalten wir "Makro Aufzeichen" an (vgl. Abbildung 13.12, Excel Kommandofolge Extras --> Makro --> Aufzeichnen).

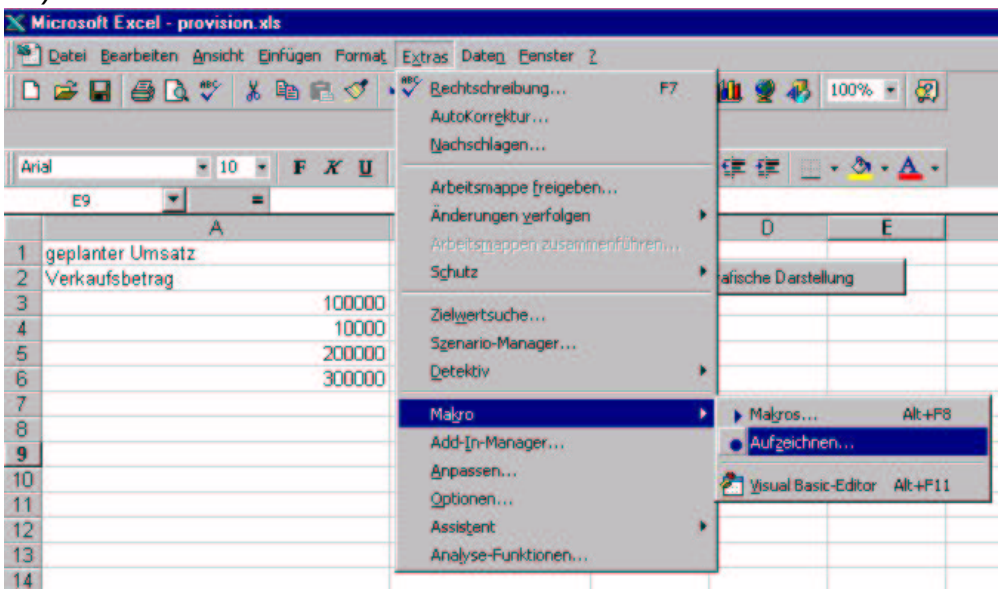

Abbildung 13.12 "Makro Aufzeichen" anschalten

Wir vergeben einen Namen für das Makro (vgl. Abbildung 13.13)..

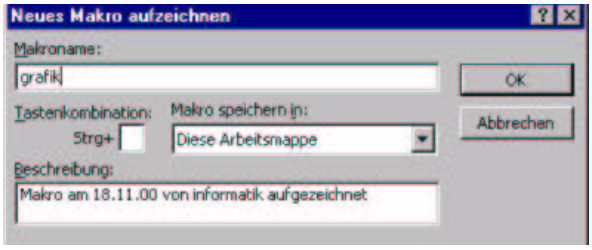

Abbildung 13.13 Namen für das Makro vergeben

Nun markieren wir die Zellen, die in das Diagramm aufgenommen werden sollen (vgl. Abbildung 13.14).

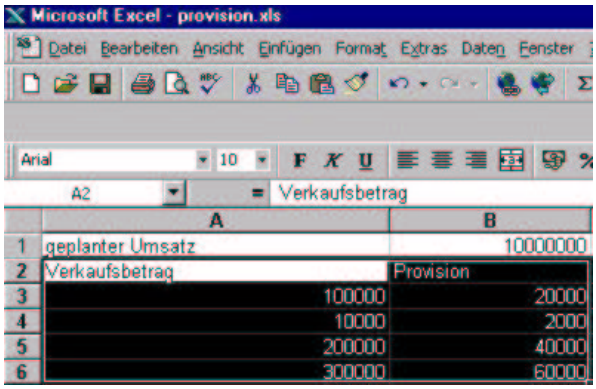

Abbildung 13.14 Darzustellenden Tabellenbereich markieren

Wir fügen ein neues Diagrammm ein (vgl. Abbildung 13.15, Excel Kommandofolge Einfügen --> Diagramm).

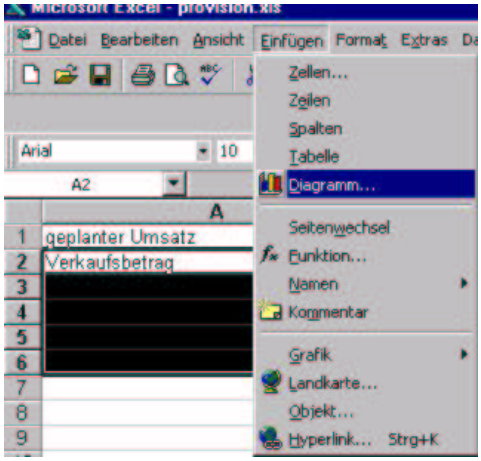

Abbildung 13.15 Diagramm einfügen

In den nächsten vier Excel-Eingabefenstern geben wir die für die Erstellung des Diagramms notwendigen Informationen ein. Wir wählen ein Liniendiagramm und belassen sonst eigentlich die Voreinstellungen (vgl. Abbildung 13.16).

Daraufhin erscheint das Diagramm im Tabellenblatt (vgl .Abbildung 13.17).

Wir beenden die Aufzeichnung (vgl. Abbildung 13.18).

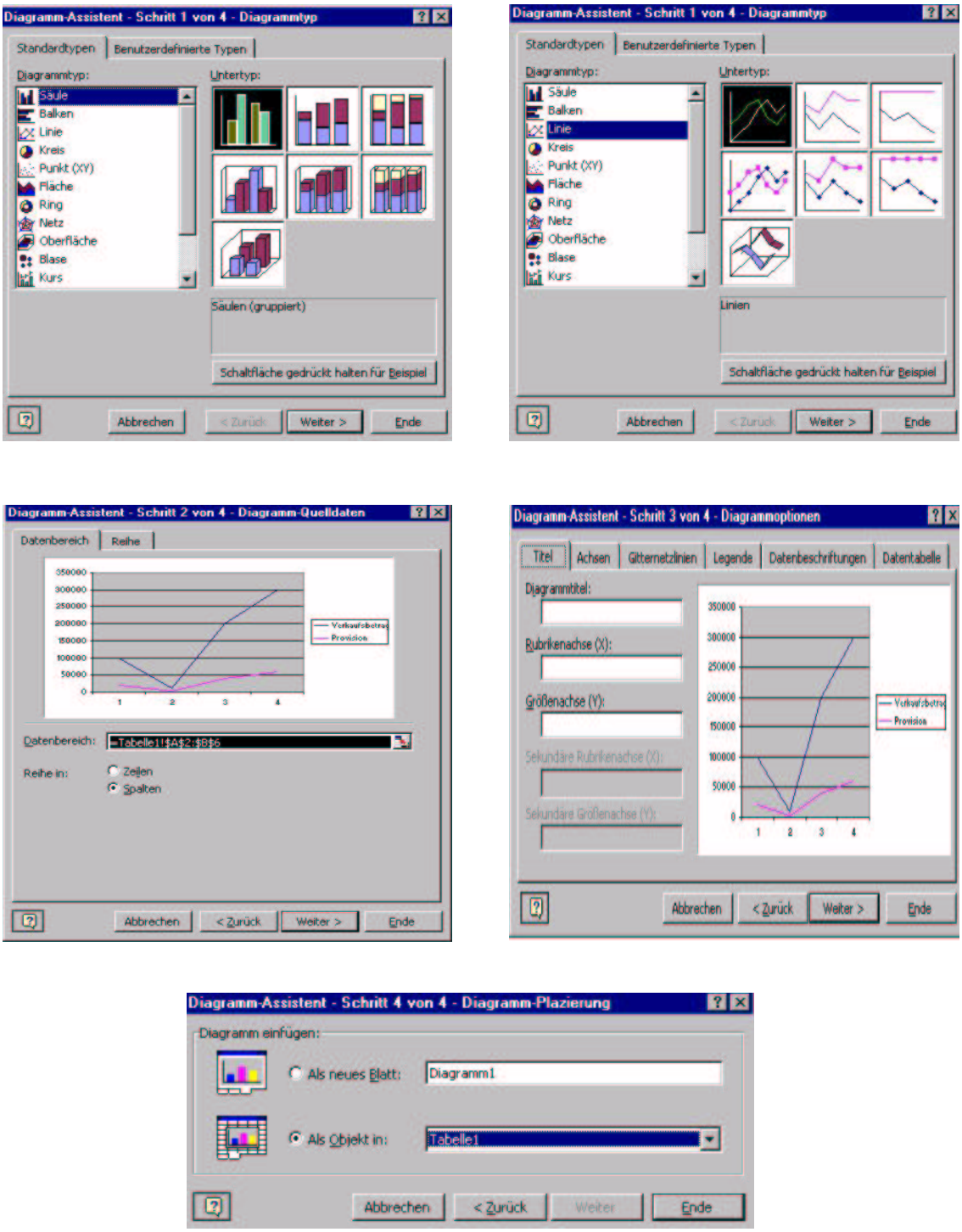

Abbildung 13.16 Dialoge zu Diagramm einfügen

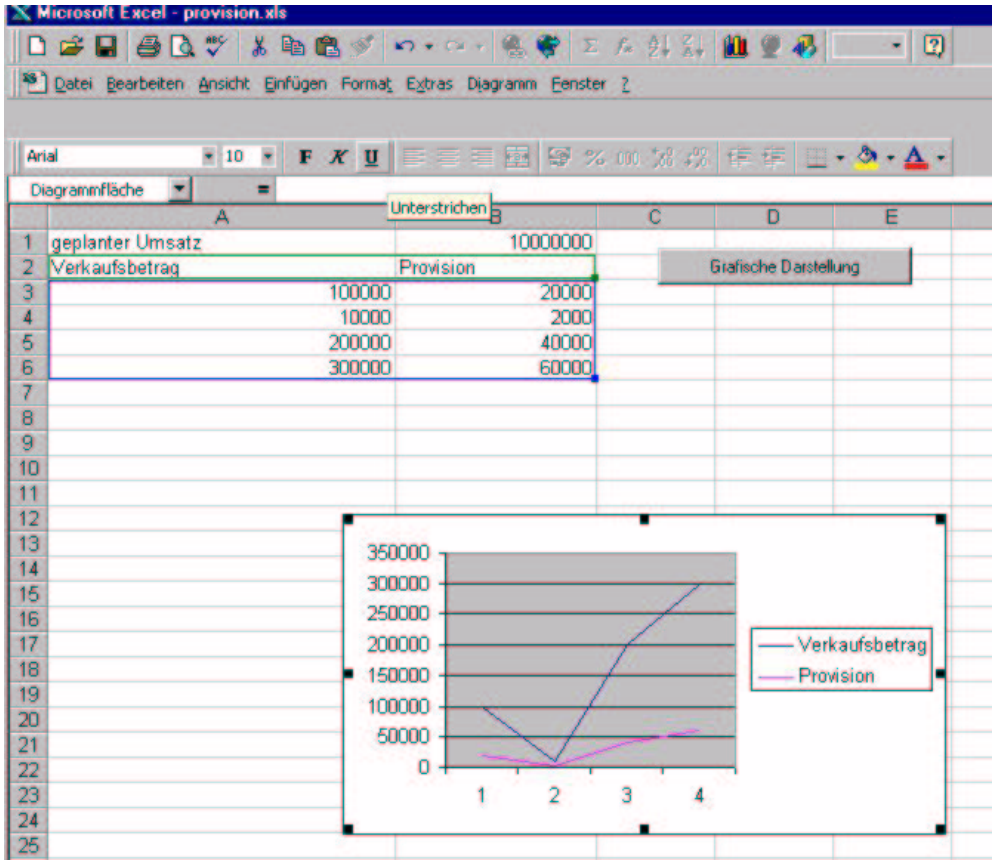

Abbildung 13.17 Das Diagramm erscheint

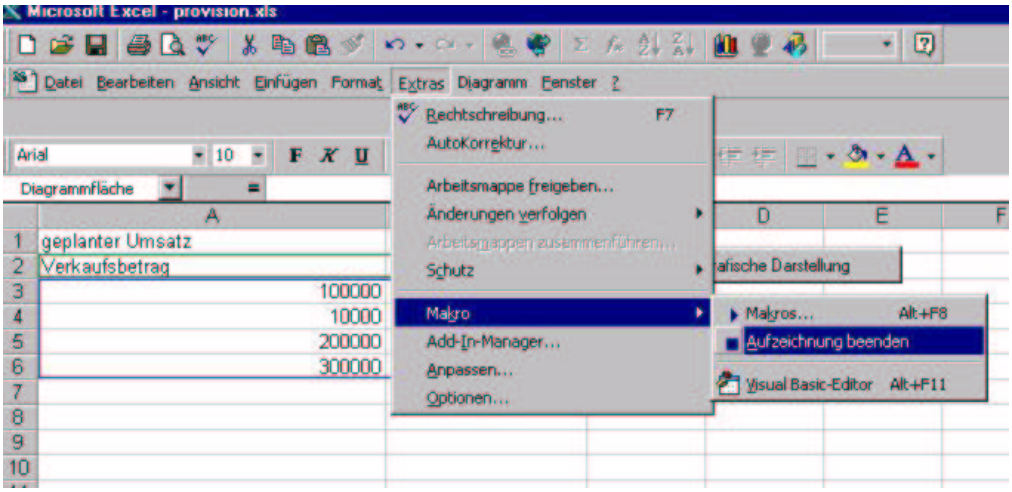

Abbildung 13.18 "Makro Aufzeichen" beenden

Vorher haben wir, wie in Abbildung 13.10 dargestellt, eine Schaltfläche mit dem Namen `GrafischeDarstellung` erzeugt. Damit ist auch das in Abbildung 13.10 dargestellte Code-Skelett vorhanden.

Durch das Aufzeichnen das Makros entstand eine Prozedur mit dem Namen des Makros in einem von Excel neu angelegten Modul (vgl. Abbildung 13.19).

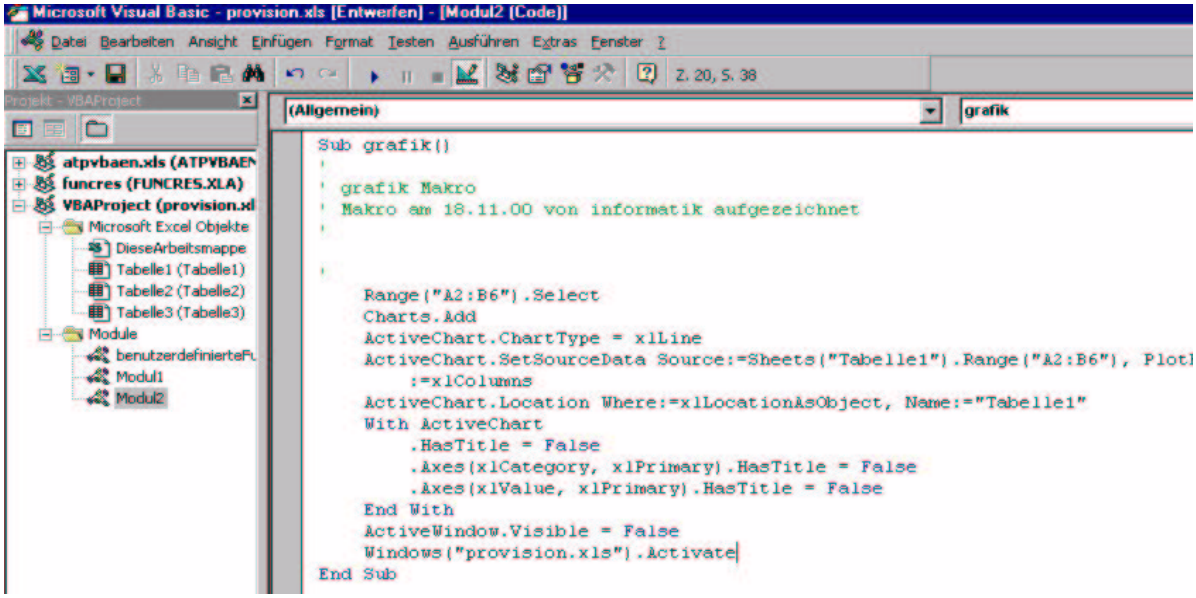

Abbildung 13.19 Das aufgezeichnete Modul

Im letzten Schritt kopieren wir den aufgezeichneten Code in das von Excel erzeugte Code-Skelett für unsere Schaltfläche. Wir löschen das Modul mit dem aufgezeichneten Code.

Jedesmal, wenn wir jetzt unsere Schaltfläche betätigen, wird die Grafik neu erzeugt.

Dies ist von jetzt an der übliche Weg, wenn wir erstmals mit Excel-Objekten, die wir auch aus Menüs oder ähnlichem erreichen können, arbeiten wollen. Wir lassen uns von Excel Beispiel-Code erzeugen und passen diesen an.

Jetzt wollen wir aber das erzeugte Programm im Einzelnen durchgehen:

Beispiel 13.2 Mit VBA ein Diagramm erzeugen

```
Private Sub GrafischeDarstellung_Click()
'
'
'
    Range("A2:B6").Select
    Charts.Add
    ActiveChart.ChartType = xlLine
    ActiveChart.SetSourceData Source:=Sheets("Tabelle1").Range("A2:B6"), PlotBy _
        :=xlColumns
    ActiveChart.Location Where:=xlLocationAsObject, Name:="Tabelle1"
    With ActiveChart
        .HasTitle = False
        .Axes(xlCategory, xlPrimary).HasTitle = False
        .Axes(xlValue, xlPrimary).HasTitle = False
    End With
```

```
    ActiveWindow.Visible = False
    Windows("provision.xls").Activate
End Sub
```

Die erste Zeile des Programms wurde ja von Excel bei der Erzeugung der Schaltfläche angelegt. Sie ist eine normale Prozedurdeklaration. Das Schlüsselwort Private sagt aus, daß die Prozedur nur innerhalb des Moduls, in dem sie deklariert ist, gesehen wird (aufgerufen werden kann). Das ist hier kein Nachteil, weil die Prozedur ja sowieso von der Excel-Ereignisverwaltung gerufen wird[39].

```
Private Sub GrafischeDarstellung_Click()
```

`Range` definiert offenbar ein Objekt. Man sieht dies daran, daß mittels eines Punktes eine Methode (`Select`) angeschlossen wird. `Range("A2:B6")` ist der Zellbereich zwischen A2 und B4. Die Methode `Select` markiert ihn.

```
Range("A2:B6").Select
```

`Charts` ist eine Objektliste, nämlich die Liste aller Diagramme (engl. Charts) der Arbeitsmappe. `Charts.Add` erzeugt ein neues Diagramm.

```
Charts.Add
```

Das zuletzt erzeugte (oder ausgewählte) Diagramm ist das aktuelle oder aktive Diagramm. Es kann über `ActiveChart` angesprochen werden. `ActiveChart` ist also der Name des gerade aktiven Chart-Objekts. Durch `ActiveChart.ChartType` wird der Typ des Diagramms festgelegt. `ChartType` ist also eine Eigenschaft des Chart-Objekts. Wir hatten ein Liniendiagramm erzeugt (vgl. Abbildung 13.16). `xlLine` ist der Excel interne Name für Liniendiagramme.

```
ActiveChart.ChartType = xlLine
```

Als nächstes wird die Methode `SetSourceData` des `ActiveChart`-Objekts gerufen. `SetSourceData` verfügt über Unmengen von Übergabeparameteren. Wir haben nur sehr wenige davon benutzt, wir haben ja überall die Defaulteinstellungen belassen. Die Makro-Aufzeichnung benutzt also Parameter mit Namen (vgl. Kapitel 11.13), um der Methode die notwendigen Parameter zu übergeben. Der erste Parameter legt die Quelle der Daten fest (die Daten, die im Diagramm dargestellt werden sollen). Wir sehen, daß es eine Objektliste namens `Sheets` gibt. Dies sind (überraschenderweise :-)) die Tabellenblätter der Arbeitsmappe. Excel kann ja hier beliebig viele von verwalten. `Sheets("Tabelle1")` ist also das erste Tabellenblatt. Danach wird der Zellenbereich angegeben. Das sind die Zellen von A2 bis B4 und das hatten wir ja schon weiter oben. `PlotBy` legt fest, ob Spalten oder Zeilen geplottet werden sollen. Wir hatten Spalten ausgewählt. `xlColumns` ist der Excel interne Name dafür.

```
ActiveChart.SetSourceData Source:=Sheets("Tabelle1").Range("A2:B6"), PlotBy _
 :=xlColumns
```

Nun sehen wir ein uns bis jetzt neues VBA-Schlüsselwort. `With` ist einfach eine abkürzende Schreibweise. In allen Zeilen zwischen `With` und `End With` wird einfach

39.Zu Sichtbarkeiten von Funktionen und Variablen vgl. Kapitel 11.11.

das auf `With` folgende Wort dem Punkt vorangestellt[40]. Natürlich muß jede Zeile zwischen `With` und `End With` mit einem Punkt beginnen. Der Code

```
With ActiveChart
       .HasTitle = False
       .Axes(xlCategory, xlPrimary).HasTitle = False
       .Axes(xlValue, xlPrimary).HasTitle = False
End With
```

entspricht also:

```
ActiveChart.HasTitle = False
ActiveChart.Axes(xlCategory, xlPrimary).HasTitle = False
ActiveChart.Axes(xlValue, xlPrimary).HasTitle = False
```

Insgesamt setzt der Code drei Eigenschaften des `ActiveChart`-Objekts. Er legt fest, ob es Überschriften für das Diagramm, die x-Achse und die Y-Achse gibt. Wir hatten dies bei der Erzeugung des Diagramms alles verneint. Die Eigenschaften werden also alle auf `false` gesetzt.

```
ActiveWindow.Visible = False
```

Diese Zeile setzt das zur Zeit aktive Fenster auf unsichtbar. Das zur Zeit aktive Fenster ist die VBA-Entwicklungsumgebung innerhalb derer der Code läuft. Die soll ja keiner sehen. Die letzte Zeile macht das Excel-Fenster wieder zum aktiven Fenster.

```
Windows("provision.xls").Activate
```

So ist auch das kurze Bildschirmflackern während des Laufens des Makros zu erklären. Die VBA-Entwicklungsumgebung wird das aktive Fenster und eigentlich sichtbar. Der Code läuft jedoch zu schnell durch, um das Fenster wirklich sichtbar werden zu lassen und durch die letzten beiden Zeilen bleibt (oder richtiger wird wieder) die Excel-Tabelle sichtbar[41].

40. Hier sehen wir auch, daß Excel keinen optimalen Code erzeugt. Warum?
41. Auch das hätte man besser (kürzeres Flackern) lösen können. Wie?