VBA

Automatisierungstechnik für Office-Pakete

Bernd Blümel, Volker Klingspor, Christian Metzger

Version: 10. Oktober 2019

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In diesem Script werden Sie lernen, wie man in VBA[1] programmiert und wie mittels der Programmiersprache VBA die Elemente einer Tabellenkalkulation (Tabellen, Zellen, eingefügte Diagramme, etc.) manipuliert werden können. Mit Hilfe von VBA-Programmierung sind Sie in der Lage, langweilige Routineaufgaben durch VBA zu automatisieren und sich damit viele Mausclicks sowie vermutlich auch einige Minuten Zeit zu sparen. Diese gesparte Zeit können Sie dann nutzen, um entspannt eine Tasse Kaffee zu trinken ☺.
Darüberhinaus lernen Sie, Excel um eigene, in der Tabellenkalkulation noch nicht vorhandene, Funktionen zu erweitern.
Die Autoren dieses Scripts versuchen (hoffentlich erfolgreich ☺) Ihnen die Wirtschaftsinformatik mittels Beispielen sowie Übungsaufgaben so einfach wie möglich beizubringen.
Dazu werden in diesem Script durchgängig zwei Anwendungen, die Umsatzprovisionsberechnung sowie das Klausurauswertungsprogramm, erarbeitet. Die Anwendungen beginnen mit einem einfachen Programm und werden dann in jedem Kapitel um weitere Programmzeilen erweitert.
Nun zu den zwei Anwendungen:

- Stellen Sie sich folgendes Geschäftsmodell vor: Das Unternehmen, für das Sie arbeiten, vertreibt im Auftrag Anderer Produkte und erhält dafür Provisionen. Beispiel dafür sind Reisebüros oder Reisebüroketten, die für große Reiseveranstalter Reisen vermarkten. Der Provisionssatz, den Ihr Unternehmen erhält, hängt vom Umsatz, den Sie insgesamt in jedem Jahr mit dem Reiseveranstalter machen, ab. Am Anfang eines Jahres wird ein Zielumsatz definiert. Für jede Reise, die Sie vertreiben, erhalten Sie eine auf dem Zielumsatz basierende Provision. Am Ende des jeweiligen Jahres wird der Zielumsatz mit dem tatsächlich angefallenen Umsatz verglichen. Führt der tatsächlich angefallene Umsatz dazu, dass Ihr Unternehmen eigentlich einen anderen Provisionssatz hätte erhalten müssen, erhalten Sie eine Gutschrift oder eine Rechnung über den Differenzbetrag. Für diese Art von Geschäftsmodell gibt es zahllose weitere Beispiele:

  - Sie kaufen und verkaufen und Ihre Einkaufskonditionen hängen vom jährlichen Gesamtumsatz ab.
  - Sie arbeiten in einer Bank in der Unternehmensanalyse und rufen bei einer Rating-Agentur Ratings für die zu analysierenden Unternehmen ab. Die Kosten pro Rating hängen von einem vorher vereinbarten Jahreszielumsatz ab.

  Der Erfolg Ihres Unternehmens hängt in all diesen Fällen vom Erreichen einer Zielgröße, dem vereinbarten Umsatz, ab. Das bedeutet aber, dass dies im Laufe des Jahres kontrolliert werden muss. Zu jedem Zeitpunkt des Jahres muss der bisher erreichte Umsatz auf das Jahr hochgerechnet werden können, damit beurteilt werden kann, ob das Ziel erreicht wird oder nicht. Sinnvoll sind auch Szenarien wie: Wenn alles so weiter läuft wie bisher, dann wird der geplante Umsatz überschritten, wir erhalten einen besseren Provisionssatz und unser Gewinn wird sich dadurch erhöhen oder auch umgekehrt. Schön wäre auch, Planrechnung und Istrechnung in unterschiedlichen Tabellen der Arbeitsmappe gegenüber stellen zu können. Diagramme, wie sich die Istentwicklung auf das Ergebnis auswirken wird, sind immer sinnvoll, denn Manager sind ganz grelle auf bunte Bilder. Diese Betrachtungen sind natürlich auch für den Lieferanten sinnvoll, denn auch seine Geschäftsentwicklung hängt davon ab, ob seine Partner ihre mit ihm vereinbarten Ziele erreichen oder nicht. Solche Problematiken lassen sich mit einer Tabellenkalkulation und VBA

[1] Visual Basic for Applications

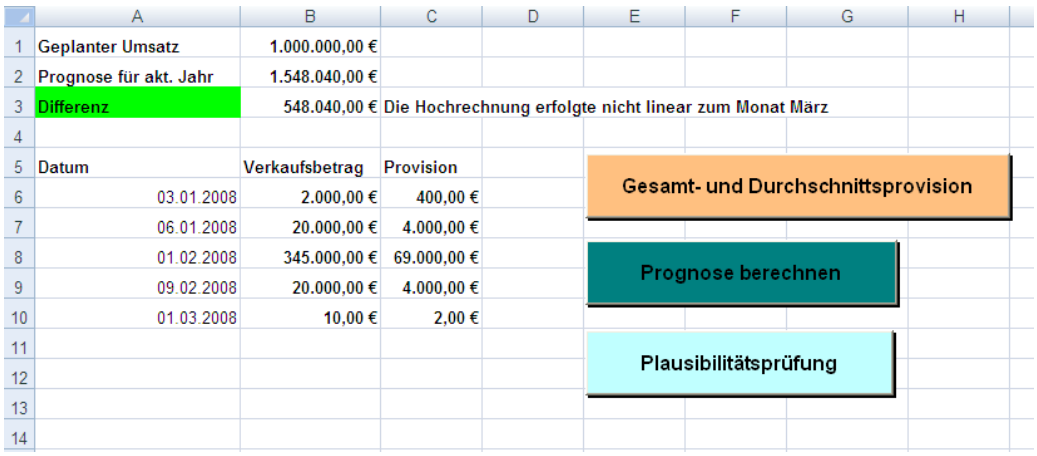


Abbildung 1.1
Programm zur Berechnung von Provisionen

auf elegante Weise lösen. Im Laufe dieses Scripts entsteht ein VBA-Programm, das genau diese Aufgabenstellung löst.

- Klausurauswertung: Wir müssen in regelmäßigen Abständen Klausuren stellen, korrigieren und bewerten. Dies geschieht dadurch, dass wir für die einzelnen gelösten oder auch nicht gelösten Aufgaben Punkte vergeben, die Punkte für alle Aufgaben addieren und dann abhängig von der Punktzahl eine Note berechnen. Dies geschieht nach festgelegten Regeln. So ist z.B. eine Minimalpunktzahl von 50 % der erreichbaren Punkte für eine 4 erforderlich. Zum Schluss erzeugen wir eine Ausgabe, wie viele Einsen, Zweien usw. es gegeben hat und wieviel Prozent aller Teilnehmer das waren. Wir berechnen die Durchschnittsnote und erzeugen Grafiken mit all diesen Informationen. Das kann man sicher alles von Hand machen. Wir haben aber ein Programm, wo wir nur die Punktzahl der Teilnehmer pro Aufgabe eingeben müssen. Dann macht das Programm uns alles fertig.

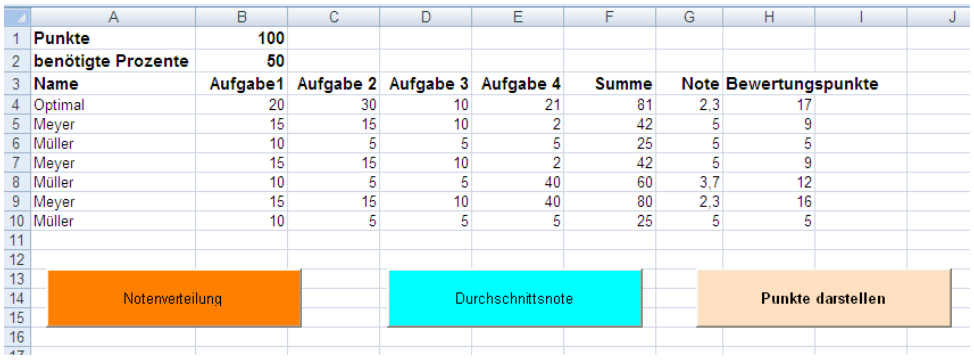


Abbildung 1.2
Programm zur Berechnung von Klausurnoten

Sobald ein Themengebiet abgeschlossen ist, vertiefen wir den behandelten Stoff anhand dreier weiterer Beispiele. Bei diesen Beispielen handelt es sich um in der Vergangenheit gestellte Klausuren. Sie können also jeweils am Ende eines Themengebietes erkennen, wie dieses Thema in der Klausur geprüft wird. Im Einzelnen handelt es sich um:

- Das Gewinnbeispiel (Klausur 2014-06-28)
- Das Schlüsseldienstbeispiel (Klausur 2014-02-01)
- Das Zuweisungsbeispiel (Klausur 2012-06-23)

Diese Klausuren finden Sie neben vielen weiteren Klausuraufgaben natürlich auch im Klausurarchiv im Downloadbereich.

Kapitel 2

Einführende Beispiele

In diesem Kapitel stellen wir einige kleinere Programme vor. Dies soll Ihnen ein Gefühl für Programmierung geben. Sie brauchen die Beispiele nicht vollständig zu verstehen. Alle Programmkonstrukte, die in diesem Kapitel vorkommen, werden in eigenen Kapiteln eingehend behandelt.
Zunächst trinken wir einen Kaffee ☕. Da wir leider nur Kaffeebohnen haben, müssen wir diese in einer Kaffeemühle mahlen. Dazu schütten wir die Kaffeebohnen in die Kaffeemühle, drehen ein paarmal an der Kurbel und erhalten Kaffeepulver. Mit diesem Pulver kann man dann Kaffee zubereiten. Angenommen, VBA könnte Kaffee mahlen, würde das als Programmcode in etwa so aussehen:

Beispiel 2.1 *Das Programm „Kaffee mahlen“*

```
function kaffeeMahlen(kaffee as bohne) as pulver
    kaffeeMahlen = pulverisiere_irgendwie_den_kaffee(kaffee)
End function
```

Wir werfen die Kaffeebohnen in die Kaffemühle und die Kaffeemühle liefert uns Kaffeepulver zurück. In den beiden runden Klammern stehen diejenigen Werte, welche in die Kaffeemühle „reingeworfen“ werden. In diesem Fall „Kaffee“. Da VBA nicht sonderlich intelligent ist, erkennt es nicht, dass wir Kaffeebohnen reinwerfen. Deswegen schreiben wir das explizit mit in die runde Klammer. Das Programm soll den Kaffee in Form von Kaffeepulver zurückgeben. Auch dies geben wir wieder an, diesmal nach den runden Klammern. Es wurde also Kaffee in Form von Bohnen in das Programm geworfen und das Programm gab Kaffeepulver zurück. Das ist auch das Prinzp bei Funktionen in VBA. Als Funktion wird in VBA ein Programm bezeichnet, in welches man etwas reinschickt (z.B: Zahlen, Texte) und das, wie auch die Kaffeemühle, irgendwas zurück gibt (z.B. auch wieder Zahlen und Texte). In eine Funktion, welche zwei Zahlen addieren soll würde man dann zwei Zahlen „reinwerfen“ und eine Zahl, nämlich die Summe, zurückerhalten. Das war doch jetzt einfach!? Nun schauen wir uns eine „echte“ Funktion an: Dazu beginnen wir mit einem Programm, das den Text „Hallo Welt“ zurückgibt und zeigen Ihnen, wie wir das Funktionsergebnis in einer Zelle eines Tabellenblatts ausgeben können.

2.1 Ausgabe eines Strings

Beispiel 2.2 *Ausgabe eines Strings mittels einer Funktion*[1]

```
'Dateiname: beispiel.xls
function helloWorld() As String
    ' Programm gibt Hello World aus
    helloWorld = "Hallo Welt!"
End function
```

[1] Alle Beispiele dieses Skripts finden Sie im Downloadbereich als Excel-Dateien, so dass Sie sie selbst ablaufen lassen können. Damit Sie sie in der Menge der von uns zur Verfügung gestellten Dateien finden, geben wir in jedem Beispiel den Dateinamen in einem Kommentar (was das ist, lernen Sie gleich) an.

Unser erstes Programm ist, wie alle unsere ersten Programme, eine Funktion[2]. Funktionen können, wie wir gleich sehen werden, ganz einfach in Arbeitsblätter der Tabellenkalkulation eingebunden werden.
Die Funktion startet in Zeile 1 mit dem reservierten Wort *function*. Reservierte Worte sind, wie der Name schon sagt, für VBA reserviert. *function* darf nur benutzt werden, um Funktionen einzuleiten. Funktionen enden mit den reservierten Worten *End function*. Auf den Befehl *function* (reservierte Worte werden häufig auch als Befehle bezeichnet) folgt der Name der Funktion. Jede VBA-Funktion 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 deswegen *helloWorld*, weil dieses Programm „Hallo Welt“ in eine Tabellenzelle schreibt. Beachten Sie, dass *helloWorld* kein Blank (Leerzeichen) enthält. Blanks sind innerhalb von Namen verboten. Auf den Funktionsnamen folgen runde Klammern. Diese Klammern können leer sein oder sogenannte Parameter enthalten. Parameter sind nichts anderes als Werte, welche in eine Funktion übergeben werden. Auf die runden Klammern folgen die reservierten Worte *As String*. Hier handelt es sich um eine Information über den Rückgabewert der Funktion. Dies klingt für den Anfang ziemlich kompliziert, ist es aber nicht. Unsere Funktion soll den Text „Hallo Welt“ in eine Zelle der Tabellenkalkulation schreiben. Texte heißen in der Informatik Strings (englisch für Zeichenkette). Strings können Buchstaben, Ziffern sowie sonstige Zeichen beinhalten, also so ziemlich alles. Die Information *As String* besagt nun, dass unsere erste Funktion einen String an die Tabellenkalkulation zurückgibt oder in anderen Worten, einen String in eine Zelle der Tabellenkalkulation schreiben wird. In Zeile 2 des Programms steht ein sogenannter Kommentar:

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

Kommentare sind dazu da, unseren Programmcode (die Textform unserer Programme heißt Programmcode, Quellcode oder Programmquelle) verständlicher zu machen. Die obige Kommentarzeile erklärt den Sinn unserer Funktion. Kommentare sind eine prima Hilfe, wenn Sie z.B. in zwei Jahren Ihr heute geschriebenes Programm verstehen möchten. Kommentare werden beim Programmablauf ignoriert. Das bedeutet, Sie könnten in eine Kommentarzeile richtigen VBA-Code schreiben, er würde nicht ausgeführt werden. Es empfiehlt sich, Programme zu kommentieren, allerdings nur solche Codezeilen, die nicht selbsterklärend sind. Wir kommentieren hier entgegen unserer Empfehlung aber so ziemlich alles. Das machen wir, damit Sie keinerlei Schwierigkeiten haben, den Programmcode zu verstehen. Im richtigen Leben kommentieren wir natürlich auch nur Programmzeilen, welche evtl. in der Zukunft nicht so einfach zu verstehen sind.
Die Zeile 3 des Programms

```
helloWorld = "Hallo Welt!"
```

belegt den Namen der Funktion mit einem Wert, nämlich der Zeichenkette "Hallo Welt". Wir erinnern uns: die Funktion hieß *helloWorld*. Dies führt dazu, dass dieser Wert (im Folgenden als Funktionswert bezeichnet) in der Zelle des Arbeitsblatts, in die die Funktion eingebunden ist, erscheint.
Wie binden wir unsere benutzerdefinierte Funktion in das Arbeitsblatt ein? Dies ist glücklicherweise auch ganz einfach. Das geht nämlich ganz genau so, wie wir die internen Funktionen der Tabellenkalkulation einbinden. Wir positionieren den Cursor in die Zelle, in der wir die Ausgabe der Funktion sehen wollen. Dann geben wir einfach

```
=helloWorld()
```

in diese Zelle ein. Die Tabellenkalkulation führt die Funktion aus. Das Ergebnis der Funktion erscheint in der Zelle (vgl. Abb. 2.1). Nun ist Beispiel 2.2 noch nicht das Intelligenteste aller Programme. Direkt die Worte „Hallo Welt“ in die Zelle

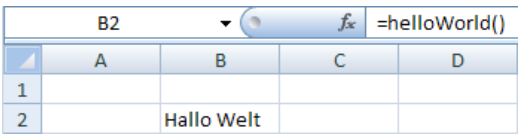


Abbildung 2.1
Einfügen der Funktion in die Tabellenkalkulation

einzugeben, hätte natürlich auch funktioniert. Darum machen wir direkt weiter mit einer neuen Funktion, die zwar auch noch nicht überwältigend intelligent ist, aber einige neue Konzepte vorstellt. Wir schreiben jetzt nämlich eine Funktion, die die Zahlen 9 und 10 addiert.

[2]Der Funktionsbegriff wird später noch ausführlicher erklärt!

2.2 Addition von Zahlen

Beispiel 2.3 *Addition mit festen Werten*

```
'Dateiname: beispiel.xls
'Programm addiert zwei Zahlen
Function addition() As Long
    ' Variablen deklarieren
    Dim ersterSummand As Long
    Dim zweiterSummand As Long
    Dim summe As Long
    ' Variablen Werte zuweisen
    ersterSummand = 9
    zweiterSummand = 10
    ' Berechnung durchführen
    summe = ersterSummand + zweiterSummand
    ' Der Funktion den Rückgabewert zuweisen
    addition = summe
End Function
```

Die ersten zwei Zeilen des Programms können wir bereits lesen und richtig interpretieren. Diese Funktion heißt *addition* und die Kommentarzeile sagt uns, dass dieses Programm die Zahlen 9 und 10 addiert. Einzig neu hier ist die Information *As Long* nach dem Programmnamen. Diese besagt aber nichts anderes, als dass diese Funktion eine ganze Zahl (Long) in eine Zelle der Tabellenkalkulation schreiben wird. Danach jedoch sehen wir neue Dinge. Wir haben drei relativ gleich aussehende Zeilen:

```
dim ersterSummand As Long
dim zweiterSummand As Long
dim summe As Long
```

Diese Zeilen beginnen mit dem Schlüsselwort *dim*. *dim* leitet eine Variablendeklaration ein. Mittels Variablen kann man Speicherplätzen im Arbeitsspeicher des Rechners ordentliche Namen zuordnen. Die Adressen der Speicherplätze in dem Arbeitspeicher eines Rechners haben sehr komische Namen wie z.B. 0X8049421. Gäbe es keine Variablen müssten Sie dann schreiben *0X8049421="Guten Morgen"*. Sicherlich könnten Sie das, aber fällt Ihnen in 10 Minuten noch diese kryptische Zeichenkette ein? Vermutlich nicht. Damit Sie sich keine komischen Zeichenketten merken müssen, wurden die Variablen erfunden. Variablennamen können Sie frei vergeben. Sie können also den äußerst treffenden Namen begruessung als Variablennamen verwenden und würden dann schreiben: *begruessung="Guten Morgen"*. Das Wort begruessung können Sie sich viel leichter merken als kryptische Zeichenketten und Sie können aufgrund des beschreibenden Variablennamens auf den möglichen Inhalt der Variablen schließen. Soviel zunächst zum Thema Variablen. Mehr zu Variablen finden Sie im Kapitel 4. Hinter den Namen der Variablen sehen wir jeweils die reservierten Worte *As Long*. Hierdurch wird der Datentyp der Variablen festgelegt. Der Datentyp[3] definiert, welche Werte die Variable aufnehmen darf. Passt der Datentyp nicht zum Wert, so kann das bis hin zu Programmabstürzen führen. Daher ist die Auswahl des richtigen Datentyps einer Variable sehr wichtig. *As Long* bedeutet, dass auf diesen drei Variablen nur ganze Zahlen abgespeichert werden dürfen. Sie erkennen die Ähnlichkeit der Datentypen der Variablen mit den Worten hinter dem Namen der Funktion. Dies ist kein Zufall und auch keine reine Ähnlichkeit, sondern Absicht. Wie die Variablen bekommt auch die Funktion selbst einen Datentyp zugewiesen. Dies wird durch *As Long* festgelegt. Das bedeutet, die Funktion gibt eine Ganzzahl zurück. Danach folgen die Zuweisungen:

```
ersterSummand = 9
zweiterSummand = 10
```

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

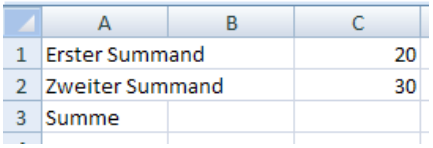
```
summe = ersterSummand + zweiterSummand
```

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 2.2. Durch

[3] Datentypen sind z.B.: Zeichenketten, Zahlen, Wahrheitswerte.

```
addition = summe
```

wird der Rückgabewert der Funktion gesetzt. Das heißt, der Wert der Variablen *summe*, in unserem Fall 19, erscheint in der Zelle, in der unsere Funktion eingebunden wird. Die Zeile *End function* beendet die Funktion.
Nun haben wir ein Programm, mit dem wir die Zahlen 9 und 10 addieren können. Das ist natürlich nicht wirklich überzeugend, insbesondere vor dem Hintergrund, dass jede beliebige Tabellenkalkulation beliebige Zahlen addieren kann. Wir wollen unser Programm jetzt so ändern, dass es zumindestens auch so etwas kann, nämlich zwei beliebige Zahlen addieren. Unser Benutzerinterface soll wie in Abb. 2.2 dargestellt aussehen.



	A	B	C
1	Erster Summand		20
2	Zweiter Summand		30
3	Summe		

Abbildung 2.2
Benutzerinterface zum Addieren zweier Zahlen

In die Zelle B1 wollen wir die erste zu addierende Zahl eingeben, in die Zelle B2 die zweite. Das Ergebnis soll in der Zelle B3 erscheinen. Damit wissen wir schon, dass wir unsere Funktion in die Zelle B3 einfügen müssen. Uns stellt sich aber folgendes Problem: Wie kommen wir an die Werte in den Zellen B1 und B2? Diese benötigen wir, um die Addition durchzuführen.
Schauen wir uns zunächst die Implementierung der Funktion an:

2.3 Addition mit Werten aus Zellen

Beispiel 2.4 *Addition mit Werten aus Zellen*

```
'Dateiname: beispiel.xls
'Programm addiert die zwei Zahlen deren Werte es aus Zellen liest
function additionMitWertenAusZellen(ersterSummand As Double,zweiterSummand As Double) As Double
    dim summe As Double
    summe = ersterSummand + zweiterSummand
    additionMitWertenAusZellen = summe
End function
```

Überraschenderweise ist der Programmcode von Beispiel 2.4 deutlich kürzer als der von Beispiel 2.3. Wie kann das sein, obwohl unsere neue Funktion doch deutlich mehr kann? Einmal fallen die Zuweisungen weg, weil wir jetzt nicht mehr feste Werte addieren, sondern die zu addierenden Zahlen aus dem Arbeitsblatt der Tabellenkalkulation holen. Darüber hinaus sind die Deklarationen der Variablen *ersterSummand* und *zweiterSummand* weggefallen. Dafür stehen diese Variablen jetzt in den runden Klammern hinter dem Funktionsnamen, allerdings ohne *dim* und mit dem Datentyp *Double*. Unsere neue Funktion soll ja nicht nur ganze Zahlen addieren können, sondern beliebige Zahlen, also auch Fließkommazahlen. Der Datentyp für Fließkommazahlen in VBA ist Double. Dadurch erklärt sich auch der neue Datentyp der Funktion selber. Dieser ist jetzt *Double*, denn wenn wir zwei Fließkommazahlen addieren, ist das Ergebnis normalerweise wieder eine Fließkommazahl. Obiges Beispiel kann noch kürzer geschrieben werden, indem die Variable *summe* nicht benutzt wird, sondern direkt der Funktion das Ergebnis der Addition zugewiesen wird.

Beispiel 2.5 *Addition mit Werten aus Zellen (kürzer)*

```
'Dateiname: beispiel.xls
'Programm addiert die zwei Zahlen deren Werte es aus Zellen liest
function additionMitWertenAusZellenKuerzer(ersterSummand As Double,zweiterSummand As Double) As ➥
    Double
    additionMitWertenAusZellenKuerzer = ersterSummand + zweiterSummand
End function
```

Die Variable *summe* war nur ein Zwischenspeicher. Dieser wurde nun weggelassen. Das Ergebnis der Addition wird direkt der Funktion zugewiesen. Kürzer kann man die Additionsfunktion nun vermutlich nicht mehr schreiben.
Kommen wir nun zu den runden Klammern. Die runden Klammern sind die Kommunikationsschnittstelle der Funktion. Über die runden Klammern kann eine Funktion Werte aus einem Arbeitsblatt[4] entgegennehmen und weiter verarbeiten. In der Zeile

```
function additionMitWertenAusZellen(ersterSummand As Double, zweiterSummand As Double) as Double
```

sagen wir, dass unsere Funktion zwei Werte erwartet, beides Fließkommazahlen. In der Funktion sind die Namen der Werte dann *ersterSummand* und *zweiterSummand*. Danach entspricht Beispiel 2.4 dann Beispiel 2.3. Wie erreichen wir aber nun, dass Inhalte von Zellen des Arbeitsblatts auf unsere Variablen *ersterSummand* und *zweiterSummand* übertragen werden? Auch dies ist glücklicherweise nicht so wirklich schwer. Wir erinnern uns: Der Wert, den wir als ersten Summanden haben wollen, steht in der Zelle B1, der Wert für den zweiten Summanden in der Zelle B2. Wir gehen jetzt folgendermaßen vor: Wir positionieren den Cursor in die Zelle B3, denn dort soll ja schließlich das Ergebnis erscheinen. In die Zelle B3 tippen wir ein:

```
=additionMitWertenAusZellen(B1; B2)
```

Die Tabellenkalkulation startet nun die Funktion *additionMitWertenAusZellen* und überträgt den Inhalt der Zelle B1 in die Variable *ersterSummand* sowie den Inhalt der Zelle B2 in die Variable *zweiterSummand*. Beachten Sie, wenn Sie die Funktion schreiben, dass die Variablen in den runden Klammern (von nun an Übergabevariablen oder Übergabeparameter genannt) durch Kommata, im Arbeitsblatt hingegen, wenn Sie die Zellen angeben, deren Werte in die Variablen übernommen werden sollen, diese durch Semikola getrennt werden müssen. Abb. 2.3 zeigt einen Screenshot der Einbindung unserer Funktion.

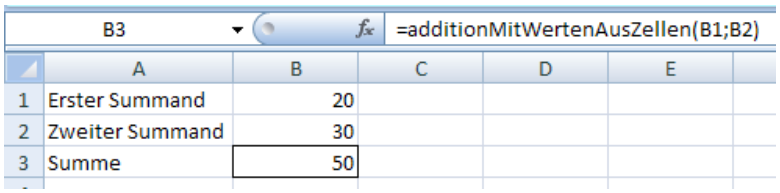


Abbildung 2.3
Einfügen einer Funktion mit Übergabewerten in die Tabellenkalkulation

Auch dieses Beispiel ist für eine Tabellenkalkulation nichts umwerfend Neues, addieren können wir in Excel eigentlich auch ohne unsere Funktion schon. Sie haben aber gelernt, wie man Werte aus dem Arbeitsblatt einer Tabellenkalkulation in eigene, selbstgeschriebene Funktionen übernehmen kann.

2.4 For-Schleife

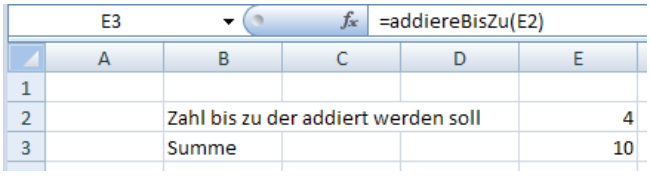


Abbildung 2.4
Addition von natürlichen Zahlen

Zum Abschluss dieses Kapitels nun ein Beispiel, das Tabellenkalkulationen so einfach nicht mehr können: Wir wollen eine Zahl eingeben und unsere Funktion soll alle natürlichen Zahlen bis zu der eingegebenen Zahl aufaddieren. Wird

[4] und, wie Sie später sehen werden, auch aus anderen Programmen und Funktionen

z.B. 4 eingegeben, soll das Programm $1 + 2 + 3 + 4$ rechnen. Unser Benutzerinterface soll wie in Abb. 2.4 dargestellt aussehen.
Betrachten wir zunächst den Programmcode der Funktion:

Beispiel 2.6 *Addition bis zu einer eingegebenen Zahl*

```
'Dateiname: beispiel.xls
function addiereBisZu(endZahl as Long) As Long
    dim summe As Long
    dim i As Long
    summe = 0
    for i = 1 to endZahl
        summe = summe + i
    next i
    addiereBisZu = summe
End function
```

Gemäß unserem Benutzerinterface tragen wir in die Zelle E3

```
= addiereBisZu(E2)
```

ein. Dies führt dazu, dass die Variable *endZahl* unserer Funktion mit dem Wert der Zelle E2 belegt wird. In unserem Beispiel ist das 4 (vgl. Abb. 2.4). Diskutieren wir nun den weiteren Code:
Nach der Deklaration der Variablen *summe* und *i* wird die Variable *summe* mit 0 initialisiert[5].

```
summe = 0
```

Dann folgt eine Schleife. Schleifen werden in Kap. 8 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 endZahl
```

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 *endZahl* ist.
Wenn der Benutzer wie in Abb. 2.4 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 endZahl
```

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

[5] Initialisieren bedeutet: einen Startwert zuweisen.

```
for i = 1 to endZahl
```

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 *endZahl* ist. Da *i* 2 ist und *endZahl* 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 *endZahl* ist. Dies ist der Fall (*i* ist 3, *endZahl* 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 der Wert von *i* immer noch kleiner oder gleich dem Wert von *endZahl* ist. Dies ist der Fall (*i* ist 4, *endZahl* 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 der Wert von *i* immer noch kleiner oder gleich dem Wert von *endZahl* ist. Dies ist diesmal nicht der Fall (*i* ist 5, *endZahl* 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:

```
addiereBisZu = summe
```

Unser berechnetes Ergebnis wird jetzt ausgegeben. *summe* hatte zum Schluss den Wert 10, also erscheint dieser auch in der Zelle E3 (vgl. Abb. 2.4). Tabelle 2.1 zeigt zusammenfassend die Entwicklung der Werte der Variablen während der Durchführung der Schleife.

Tabelle 2.1
Wertetabelle der Schleife

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

In diesem Beispiel haben Sie einige neue Dinge kennengelernt:

- Die Werte von Variablen können sich ändern (vgl. Tabelle 2.1).
- Programmteile können mehrfach durchlaufen werden.
- Variablen können auf der rechten und auf der linken Seite einer Zuweisung stehen.
- 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 inititalisiert 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.

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 2.7 *Addition bis zu einer eingegebenen Zahl mit Formel*

```
'Dateiname: beispiel.xls
function addiereBisZuFormel(endZahl as Long) as Long
    dim summe As Long
    summe = (endZahl * (endZahl + 1)) / 2
    addiereBisZuFormel = summe
End function
```

Das ist natürlich sehr viel einfacher. Die Zeile:

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

zeigt uns, dass VBA nicht nur addieren, sondern auch multiplizieren und dividieren kann. Darüber hinaus kann man Klammern setzen. Damit sind beliebige Formeln in VBA abbildbar.
Alle Beispiele dieses Kapitels sind nicht besonders praxisrelevant. Dies ist zu Beginn eines Kurses über Automatisierung in Office-Produkten auch nicht weiter verwunderlich, weil Sie ja noch nicht so wirklich viel über VBA wissen. Ab den nächsten Kapiteln beginnen aber dann die in der Einleitung vorgestellten praxisrelevanten Beispiele. 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*, was ja kürzer wäre. Programme mit „guten“ Variablennamen sind aber weitaus leichter lesbar und viel verständlicher. Einzige Ausnahme ist die Schleifenvariable *i* in Beispiel 2.6. 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.

[6] Was aber eher selten ist!

Kapitel 3

Einfügen benutzerdefinierter Funktionen

Funktionen, die wir selbst schreiben und dann in Excel einfügen, heißen benutzerdefinierte Funktionen. Unter dem Schlagwort „Schreiben einer Funktion“ sind sie auch in der Hilfe von Excel zu finden (vgl. Abb. 3.1).

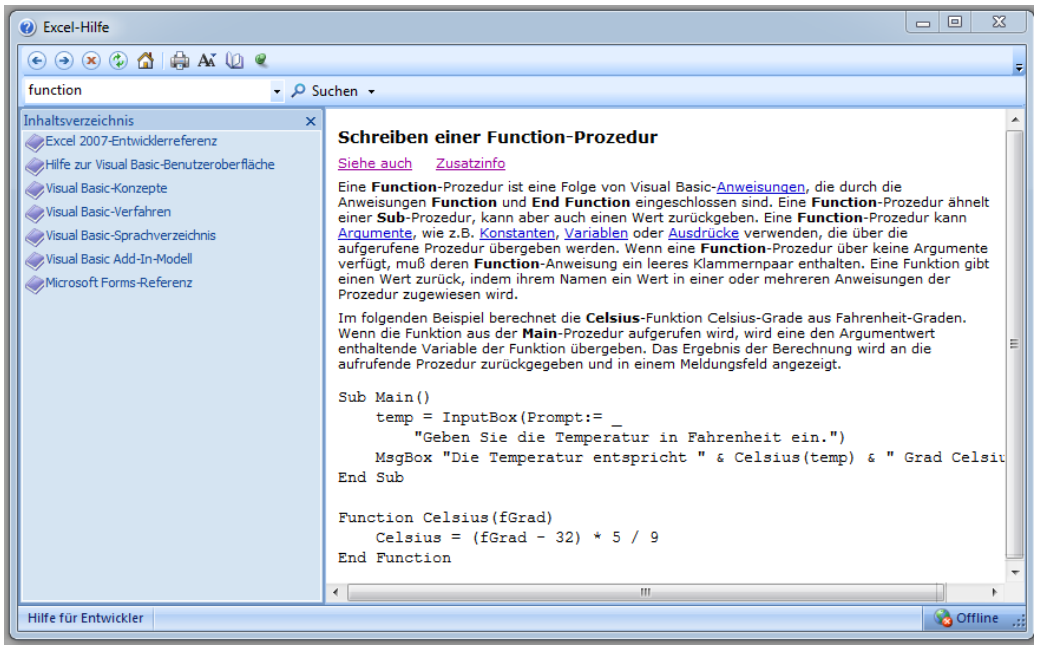


Abbildung 3.1
Open Office Hilfe

Die Funktionen, die wir in Kapitel 2 geschrieben haben, sind alles benutzerdefinierte Funktionen. Damit benutzerdefinierte Funktionen in die Tabellenkalkulation eingefügt werden können, müssen sie in bestimmten Modulen der Entwicklungsumgebung liegen. In Excel 2007 und Excel 2010 ist die Entwicklungsumgebung zunächst versteckt, und muss durch Sie aktiviert werden. Sie aktivieren diese wie auf nachfolgenden Screenshots ersichtlich[1]:

[1] Blöderweise funktioniert diese Aktivierung bei den unterschiedlichen Excel-Versionen leicht unterschiedlich, wir haben für Sie hier die Screenshots für Excel 2007 dargestellt. In unserem Webauftritt finden Sie Videos für die verschiedenen Excel-Versionen, die Ihnen anschaulich darstellen, wie Sie die Entwicklungsumgebung aktivieren können.

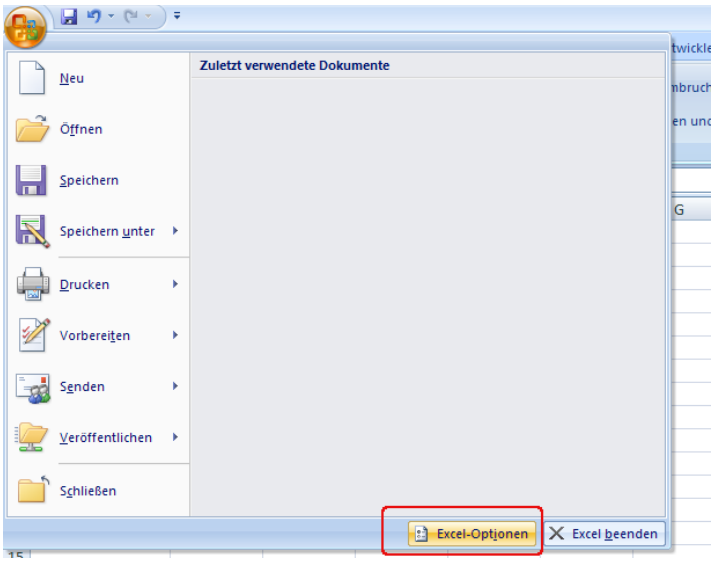


Abbildung 3.2
Excel Optionen aufrufen

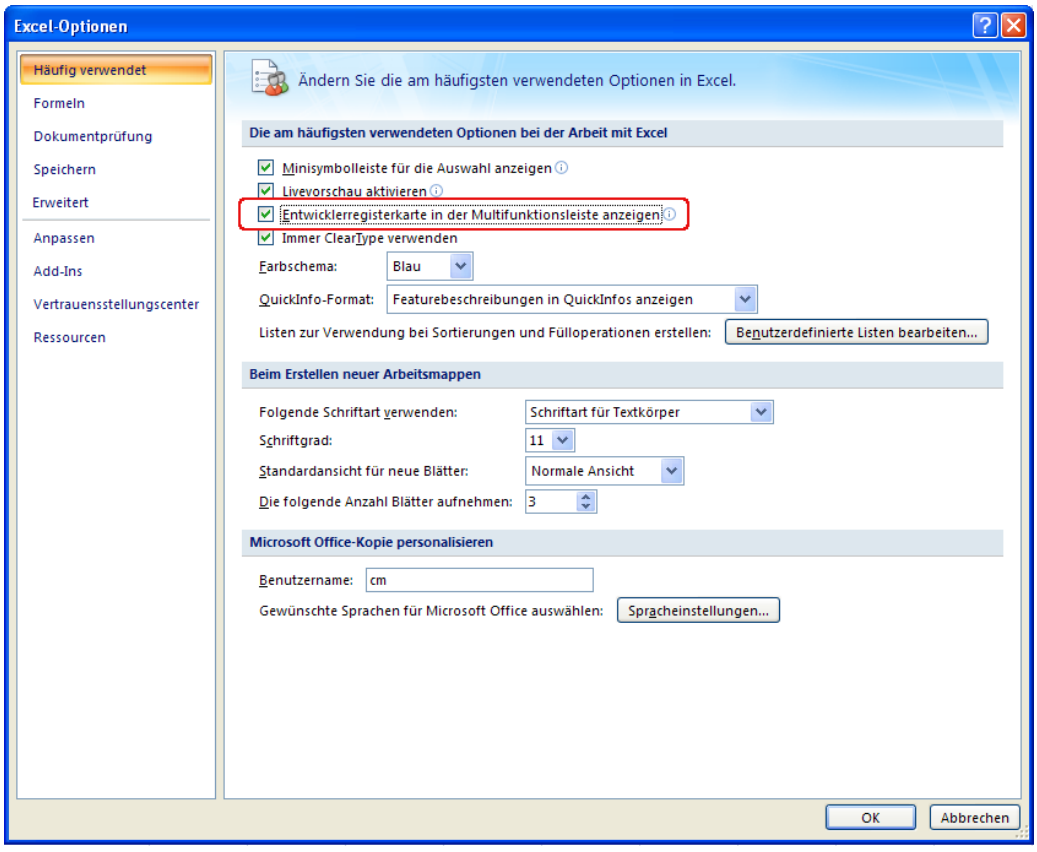


Abbildung 3.3
Excel Optionen einstellen

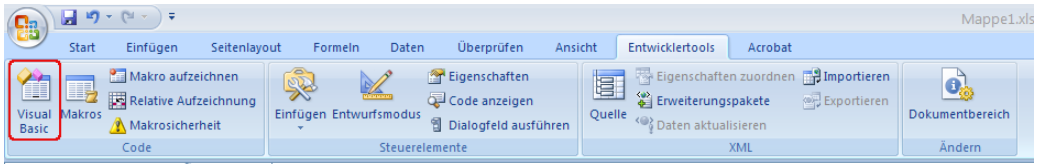


Abbildung 3.4
Excel Entwicklertab

Damit die benutzerdefinierte Funktionen von einem Excel-Tabellenblatt aus aufgerufen werden können, müssen Sie zunächst ein neues Modul anlegen (vgl. Abb. 3.5).

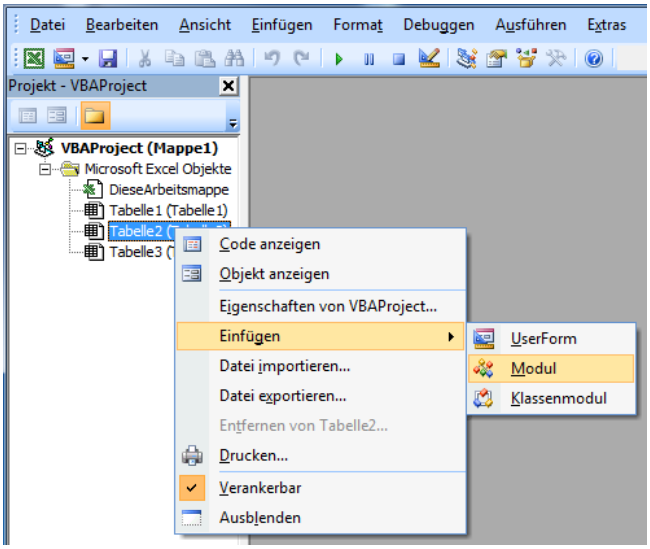


Abbildung 3.5
Einfügen eines Moduls

Dann können Sie in dem neu geöffneten Fenster Ihre eigenen Funktionen programmieren, indem Sie den Programmtext eintippen (vgl. Abb. 3.6). Sie können in einem Modul beliebig viele Funktionen programmieren, solange diese alle unterschiedlich heißen. Die Entwicklungsumgebung trennt die verschiedenen Funktionen automatisch durch eine Linie.

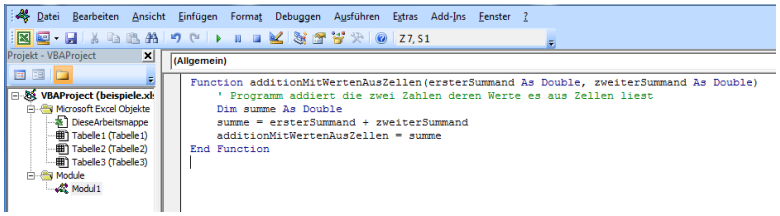


Abbildung 3.6
Unsere neue Funktion

Beachten Sie: In Excels Entwicklungsumgebung existieren weitere Fenster, in denen Sie VBA-Code eingeben können. Wenn Sie die von Ihnen geschriebenen Funktionen allerdings in der Tabellenkalkulation nutzen wollen, müssen Sie wie oben dargestellt vorgehen.
Um nun eine benutzerdefinierte Funktion zu nutzen, wechseln Sie in die Tabellenkalkulation, positionieren den Cursor in die Zelle, in der das Ergebnis der benutzerdefinierten Funktion erscheinen soll und schreiben, wie auch schon in Kap. 2 ausführlich beschrieben, nach einem Gleichheitszeichen den Namen der Funktion. Eventuelle Zellbezüge folgen durch

Semikola getrennt nach dem Namen der Funktion (vgl. Abb. 3.7).

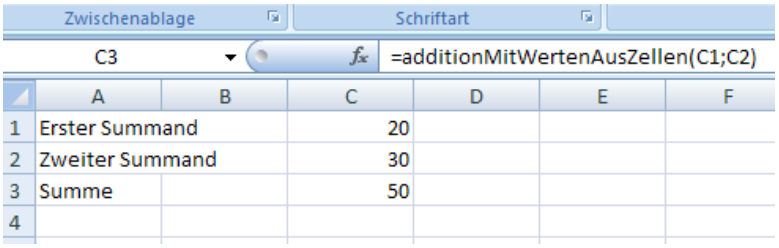


Abbildung 3.7
Excel Funktion per Eingabe einbinden

In Excel existiert eine weitere Möglichkeit, über die grafische Oberfläche benutzerdefinierte Funktionen einzubinden. Dazu schreiben Sie zunächst das Gleichheitszeichen in die Zelle, in der Sie die Funktion einfügen möchten. Dann clicken Sie zunächst auf das Menü Einfügen und wählen dort Funktionen (vgl. Abb. 3.8). Nach einem weiteren Click erscheint ein neues Fenster (vgl. Abb. 3.8).

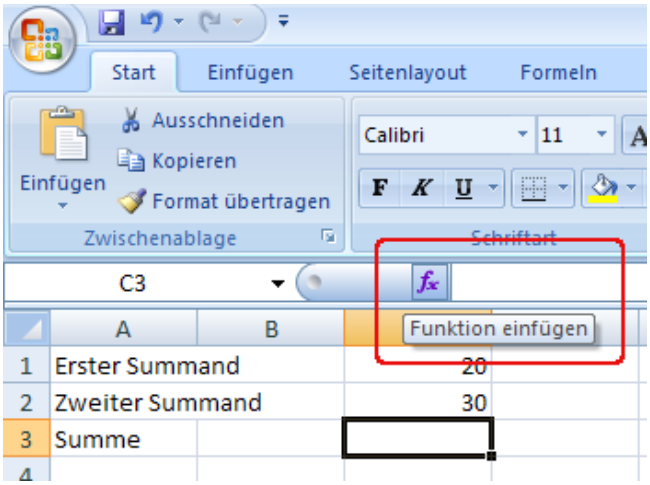


Abbildung 3.8
Das Excel Einfügen-Menü

Wie durch Zauberhand existieren dort alle unsere selbstgeschriebenen Funktionen in der Rubrik benutzerdefinierte Funktionen. Wählen Sie die gewünschte Funktion aus und bestätigen Sie (vgl. Abb. 3.10).
In diesem Zusammenhang möchten wir nochmal auf die Videoseite[2] auf unseren Homepages verweisen. Sie finden dort hilfreiche Videos, unter anderem auch zum Thema „benutzerdefinierte Funktionen einfügen“.

[2] http://www.hochschule-bochum.de/fbw/personen/bluemel/downloads/videos-grundstudium.html

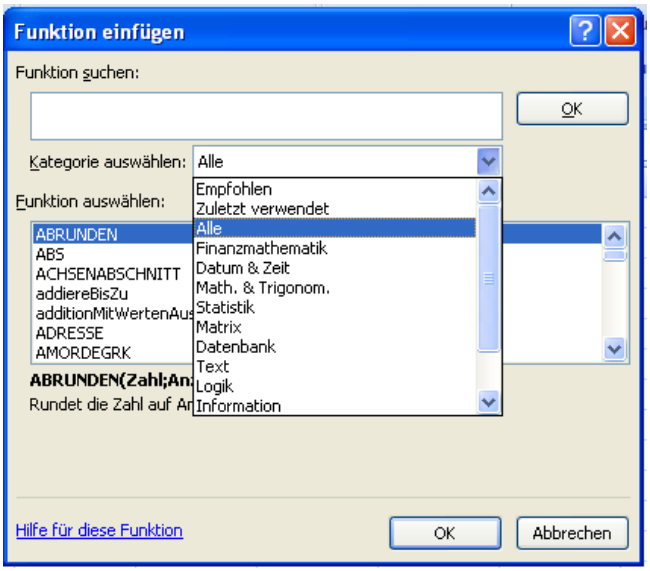


Abbildung 3.9
In Excel verfügbare interne Funktionen

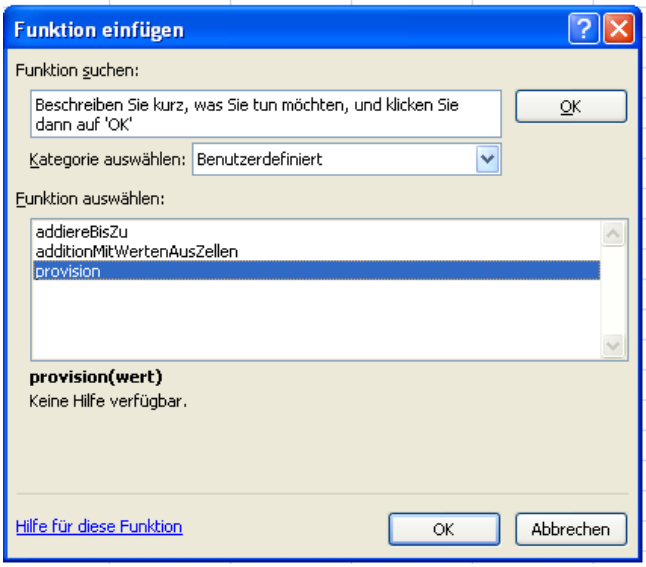


Abbildung 3.10
Die benutzerdefinierte Funktion Provision auswählen

Kapitel 4

Variablen, Datentypen, Konstanten und Operatoren

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 2.4). 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.
- *zweiterSummand*: der Name für die zweite zu addierende Zahl.
- *summe*: der Name für das Resultat.

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.
Darüber hinaus können Variablen Datentypen zugeordnet werden. Damit wird festgelegt, welche Art Werte diese Variable aufnehmen kann.
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 den Typ der Variablen. Dies wird durch erhöhten Speicherverbrauch erkauft, denn da auf einer solchen Variablen alle Datentypen gespeichert werden können, muss VBA, um auf der sicheren Seite zu sein, maximalen Speicherplatz für Variablen diesen Typs reservieren.

4.1 Datentypen in VBA

VBA unterscheidet im Wesentlichen zwischen:

- Ganzzahlen: 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
- Wahrheitswerten: true oder false

- Strings: Zeichenketten z.B. „Bernd Blümel“, „Volker Klingspor“ oder auch „0815Test“.

Abb. 4.1 gibt eine Übersicht über die in VBA möglichen Datentypen.

Name	Kürzel	Beschreibung	Größe
Integer	%	Ganze Zahlen zwischen -32768 und 32767	2 Byte
Long	&	Ganze Zahlen zwischen -2147483648 und 2147483647	4 Byte
Byte		Ganze Zahlen zwischen 0 und 255	1 Byte
Single	!	Fließkommazahlen mit 8 Stellen Genauigkeit	4 Byte
Double	#	Fließkommazahlen mit 16 Stellen Genauigkeit	8 Byte
Boolean		Wahrheitswerte	1 Byte
Char		Ein einzelnes Zeichen	2 Byte
String	$	Zeichenketten	10 Byte plus 2 Byte pro Zeichen
Date		Datum und Uhrzeit	8 Byte
Currency	@	Festkommazahlen mit 15 Stellen vor und 4 Stellen nach dem Komma	8 Byte
Variant		Standarddatentyp. Nimmt je nach Bedarf einen der obigen Datentypen an	mind. 16 Byte

Abbildung 4.1
Datentypen in VBA

Wie wir sehen, gibt es zwei Datentypen für Fließkommazahlen und drei Datentypen für ganze Zahlen. Bei Fließkommazahlen liegt der Grund hierfür 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.
Der Grund für die Existenz von drei Datentypen für ganze Zahlen liegt auf der Hand. Wenn wir wissen, wie groß die Werte der Variablen im Programmlauf werden können, können wir durch Wahl des geeigneten Variablentyps Hauptspeicher sparen.
Allerdings benötigen unsere Programme sehr wenig Speicher. Dafür besteht aber das Risiko, dass „zu klein“ deklarierte Variablen Rundungsfehler oder Programmabstürze zur Folge haben. Daher verwenden wir generell den Datenbyp Double für Fließkommazahlen und Long für ganze Zahlen.
String-Variablen benötigt man, um (trivialerweise) mit Strings zu arbeiten. Was Variablen vom Typ Boolean sind und wozu man so etwas braucht, wird später erklärt (fängt an in Kap. 5). Variablen 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 zwei Möglichkeiten:

1. Der Typ der Variablen wird durch das Schlüsselwort *As* gefolgt vom Typ der Variablen, wie in der ersten Spalte von Abb. 4.1 definiert, festgelegt. Schauen wir uns, um dies zu illustrieren, noch einmal den Code von Beispiel 2.3 an. Durch die Zeilen:

   ```
   dim ersterSummand As Long
   dim zweiterSummand As Long
   dim summe As Long
   ```

   werden also 3 Variablen vom Typ Long vereinbart. Auf jeder dieser Variablen kann eine ganze Zahl zwischen -2147483648 und 2147483647 gespeichert werden.

2. Alternativ können Variablen von Datentypen, die über ein Kürzel verfügen (zweite Spalte von Abb. 4.1), durch den Variablennamen direkt (und ohne Blank) gefolgt vom Kürzel deklariert werden. Die Variablendeklaration von Beispiel 2.3 hätten wir also auch folgendermaßen schreiben können:

```
dim ersterSummand&
dim zweiterSummand&
dim summe&
```

Betrachten wir nun einige Variablendeklarationen und Zuweisungen:

Beispiel 4.1 *Eine Funktion zur Demonstration von Variablenzuweisungen*

```
'Dateiname: variablen.xls
function variablen()
    ' Beispiele fuer Variablendeklarationen
    ' Single und Double (Fließkommazahlen) als Datentyp
    dim reelleZahl As Single
    dim reelleZahl2#
    dim reelleZahl3 As Double
    reelleZahl3 = 7
    reelleZahl2 = 4.7
    reelleZahl = reelleZahl2 + reelleZahl3
    '--------------------------------------
    ' Integer und Long (Ganzzahlen) als Datentyp
    dim ganzeZahl As Integer
    dim ganzeZahl2%
    dim ganzeZahl3 As Long
    ganzeZahl3=5
    ganzeZahl2=8
    ganzeZahl = ganzeZahl2 + ganzeZahl3
    '--------------------------------------
    ' Variant als Datentyp, weil kein Datentyp explizit zugewiesen
    dim chamaeleon
    chamaeleon = 6
    chamaeleon = "sieben"
    ganzeZahl = reelleZahl2
    ganzeZahl = "sieben"
    test=9
End function
```

In den ersten 3 Zeilen von Beispiel 4.1 werden Fließkommavariablen vereinbart. Zwei davon sind vom Typ *Double* (einmal durch die lange Schreibweise mit *As Double* und einmal durch das Anfügen des Typkürzels #), eine ist vom Typ *Single*. Die *Double*-Variablen werden addiert und die Summe der *Single*-Variablen zugewiesen. Man sieht hier, 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 zugewiesen. 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 automatisch 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. 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. Danach wird einer Ganzzahlvariablen ein String zugewiesen:

```
ganzeZahl = "sieben"
```

Dies erzeugt während des Ablaufs des Programms eine Fehlermeldung (“Typen unverträglich”). Denn es gibt keine Möglichkeit, einen String, der nicht aus Ziffern besteht, in eine Zahl umzuwandeln. 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? Wir betrachten dazu Beispiel 4.2:

Beispiel 4.2 *Addition mit Werten aus Zellen und Fehler*

```
'Dateiname: variablen.xls
'Programm addiert die zwei Zahlen deren Werte es aus Zellen liest
Function additionMitWertenAusZellen(ersterSummand As Double, zweiterSummand As Double) As Double
    Dim summe As Double
    summe = ersterSummand + zweiterSummand
    additionMitWertenAusZellen = sume '<-- falsch geschriebener Variablenname
End Function
```

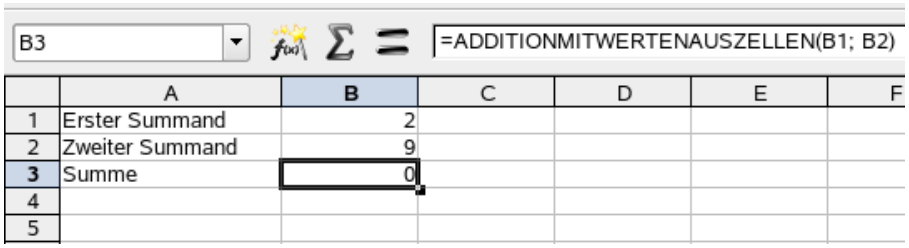


Abbildung 4.2
Addieren mit Fehler

Wie auch Abb. 4.2 zeigt, kann man nun eingeben was man will, das Ergebnis ist immer 0. Verwunderlich ist das nicht, in Beispiel 4.2 ist ja nun ein Schreibfehler. In der letzten Anweisung der Funktion steht *sume* statt *summe*. Diese Variable gibt es nicht, sie hat im ganzen Programm keinen Wert bekommen, daher belegt VBA sie mit 0. Der Rückgabewert dieser Funktion ist daher immer 0.
Glücklicherweise können wir aber in VBA die Variablenbehandlung ändern. Fügen wir in unsere Programme als erste Anweisung noch vor dem ersten *function* die Schlüsselworte *Option Explicit* ein, wird Variablendeklaration zwingend vorgeschrieben. Nicht deklarierte Variablen werden mit einer Fehlermeldung zurückgewiesen. Beispiel 4.3 zeigt das verbesserte Programm und Abb. 4.3 die ausgegebene Fehlermeldung.

Beispiel 4.3 *Addition mit Werten aus Zellen und von VBA erkanntem Fehler*

```
'Dateiname: optionExplicit.xls
Option Explicit

' Programm addiert die zwei Zahlen deren Werte es aus Zellen liest
function additionMitWertenAusZellen(ersterSummand As Double,zweiterSummand As Double)As Double
    dim summe As Double
    summe = ersterSummand + zweiterSummand
    additionMitWertenAusZellen = sume
End function
```

VBA sagt uns, dass es eine nicht deklarierte Variable gibt und verzweigt gleichzeitig in den VBA-Editor, wo die Zeile mit dem Fehler markiert wird.

```
Function additionMitWertenAusZellen(ersterSummand As Double, zweiterSummand As Double) As Double
  ' Programm addiert die zwei Zahlen deren Werte es aus Zellen liest
  Dim summe As Double
  summe = ersterSummand + zweiterSummand
  additionMitWertenAusZellen = sume
End Function
```

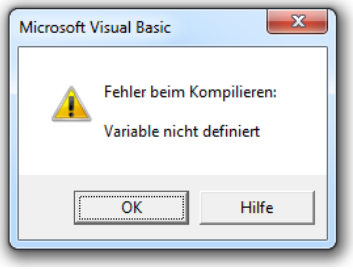


Abbildung 4.3
Fehlermeldung aus VBA

4.2 Regeln zu Variablennamen

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

1. Variablennamen beginnen mit einem Buchstaben.
2. Variablennamen können nach dem ersten Buchstaben Ziffern, Buchstaben sowie Unterstriche in beliebiger Reihenfolge enthalten.
3. Variablennamen können beliebig lang sein (naja in Wirklichkeit 255 Zeichen).
4. Groß- und Kleinschreibung spielt bei VBA keine Rolle (summe=Summe=suMME=SUMME).
5. Variablennamen dürfen außer dem _ keine Sonderzeichen enthalten.
6. Der Funktionsname darf kein Variablenname sein.
7. In VBA reservierte Worte dürfen keine Variablennamen sein.
8. Variablennamen sollten einen Bezug zu den Daten haben, die sie später aufnehmen sollen.

4.3 Regeln zu Fließkommazahlen (Wann Punkt, wann Komma)

Wenn Sie in einem VBA-Programm eine Fließkommazahl eingeben, dann ist der Dezimaltrenner der Punkt. Dies kann auch Beispiel 4.1 entnommen werden. Ein Komma als Dezimaltrenner führt zu einem VBA-Laufzeitfehler. Wenn Sie hingegen im Arbeitsblatt Eingaben tätigen, dann ist der Dezimaltrenner das Komma. Ein Punkt als Dezimaltrenner führt zu erstaunlichen und nicht immer nachvollziehbaren Ergebnissen. Manchmal wird der Punkt als Tausendertrenner interpretiert, manchmal wird der Wert in ein Datum umgewandelt.
Der Grund für dieses merkwürde unterschiedliche Verhalten ist eigentlich ganz einfach: Programmcode ist prinzipiell (bis auf unsere eigenen Namen) englisch. Somit gilt hier auch die englische Schreibweise von Zahlen mit dem Punkt als Dezimaltrenner. In der Exceltabelle wird hingegen die Schreibweise der eingestellten Länderkennung verwendet, und die ist bei unseren Excel-Installationen typischerweise Deutsch.

4.4 Konstanten

Programme benötigen häufig konstante Werte (im folgenden Konstanten genannt), um Berechnungen durchführen zu können. Eine Konstante kann explizit durch „Hinschreiben“ ihres Wertes an jeder gewünschten Stelle des Programms definiert werden. Dies zeigt Beispiel 4.4.

Beispiel 4.4 *Konstante Werte in einem VBA-Programm*

```
'Dateiname: konstante.xls
Option Explicit
function berechneVerkaufspreis(einkaufspreis As Double) As Double
    dim nettoVerkaufspreis As Double
    dim bruttoVerkaufspreis As Double
    nettoVerkaufspreis = 1.19 * einkaufspreis
    bruttoVerkaufspreis = 1.19 * nettoVerkaufspreis
    berechneVerkaufspreis = bruttoVerkaufspreis
End function
```

Dies ist nicht immer sinnvoll:

- Die Zahl 1.19 hat geringere Aussagekraft als das Wort Mehrwertsteuersatz.
- Die Konstante 1.19 hat in diesem Beispiel zwei Bedeutungen:
  - Gewinnspanne
  - Mehrwertsteuersatz

  Das macht das Programm schwerer verständlich.

Ändert sich der Umsatzsteuersatz, kann man in Beispiel 4.4 nicht einmal mit der Funktion “Suchen und Ersetzen“ den Umsatzsteuersatz ändern. Man würde die Gewinnspanne auch erhöhen.
VBA erlaubt es, Konstanten mit Namen zu definieren und diesen einmalig einen Wert zuzuweisen. Dies geschieht mit dem reservierten Wort *const*. Beispiel 4.5 zeigt die korrekte Nutzung einer Konstante.

Beispiel 4.5 *Die Berechnung des Verkaufspreises mit Konstanten*

```
'Dateiname: konstante.xls
Option Explicit
function berechneVerkaufspreisKonstante(einkaufspreis As Double) As Double
    dim nettoVerkaufspreis As Double
    dim bruttoVerkaufspreis As Double
    const UMSATZSTEUERSATZ As Double = 1.19
    const GEWINNSPANNE As Double = 1.19
    nettoVerkaufspreis = GEWINNSPANNE * einkaufspreis
    bruttoVerkaufspreis = UMSATZSTEUERSATZ * nettoVerkaufspreis
    berechneVerkaufspreisKonstante = bruttoVerkaufspreis
End function
```

Konstanten unterscheiden sich von Variablen dadurch, dass sich ihr Wert während des Programmlaufs nicht ändern kann (Konstanten sind halt konstant). Der Versuch, einer Konstanten während des Programmlaufs einen neuen Wert zu geben, führt zu einem VBA-Laufzeitfehler. Konstanten erhalten ihren Wert also ausschließlich 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.

4.5 Operatoren

Abb. 4.4 zeigt die in VBA vorhandenen Operatoren.
Die arithmetischen Operatoren kennen Sie bereits, auch wenn die Schreibweise zum Teil für Sie vielleicht ungewohnt ist. Gemeinsam mit der Möglichkeit, Klammern zu setzen, können wir also in VBA beliebige Formeln programmieren. Ein bisschen ungewohnt sind vielleicht auch die Operatoren Backslash und *mod*. Der Backslash steht für die Integerdivision. Der Modulo-Operator liefert als Ergebnis den Rest dieser Division. Ziemlich verwirrend? Hier ein kleines Beispiel zur Verwendung dieser beiden Operatoren. Sie möchten wissen wie viele Tage, Stunden, Minuten und Sekunden sich in 97456 Sekunden verbergen. Nun können Sie den Taschenrechner nehmen und ganz einfach 97456 / 86400 rechnen. Dann erhalten Sie 1,12962 als Ergebnis. Das Ergebnis ist ziemlich wertlos. Mehr Aussagekraft hat das Programm, welches Modulo und die Integerdivision zur Berechnung benutzt:

Beispiel 4.6 *Ein Programm zur Umrechnung von Sekunden*

Rang	Operatoren	Bedeutung
arithmetische Operatoren	$-$	negatives Vorzeichen
	$+, -, *, /$	Grundrechenarten
	^	Potenz
	$\backslash$	Integerdivision
	mod	Modulo-Operator
Zeichenketten-Operatoren	$+$	verbindet Zeichenketten
	&	verbindet Zeichenketten, Zahlen werden vorher in Text umgewandelt
Vergleichsoperatoren	$=$	gleich
	$<>$	ungleich
	$<, <=$	kleiner, kleiner gleich
	$>, >=$	größer, größer gleich
Logische Operatoren	And	logisches „und“
	Or	logisches „oder“
	Xor	logisches „entweder . . . oder . . . “ (nicht beides gleichzeitig)
	Not	logische Negation
Zuweisungsoperator	$=$	Zuweisung

Abbildung 4.4
Operatoren in VBA

```
'Dateiname: sekundenVerarbeiten.xls
Option Explicit

Function verarbeiteSekunden(sekunden As Long) As String
    'Deklaration der Variablen
    Dim tageinSekunden As Long
    Dim stundenInSekunden As Long
    Dim minutenInSekunden As Long
    Dim restSekunden As Long
    'Deklaration der Konstanten
    Const sekundenProTag As Long = 86400
    Const sekundenProStunde As Long = 3600
    Const sekundenProMinute As Long = 60

    '---------Wieviele Tage in Sekunden -----------
    tageinSekunden = sekunden \ sekundenProTag
    'Restsekunden
    restSekunden = sekunden Mod sekundenProTag

    '---------Wieviele Stunden in Restsekunden -----
    stundenInSekunden = restSekunden \ sekundenProStunde
    'Übriggebliebene Sekunden
    restSekunden = restSekunden Mod sekundenProStunde

    '---------Wieviele Minuten in Restsekunden -----
    minutenInSekunden = restSekunden \ sekundenProMinute
    'Übriggebliebene Sekunden
    sekunden = restSekunden Mod sekundenProMinute

    'Ergebnis der Funktion zuweisen
    verarbeiteSekunden = tageinSekunden & " Tag(e) " & stundenInSekunden & " Stunden " & ➥
        minutenInSekunden & " Minuten " & sekunden & " Sekunden"
```

```
End Function
```

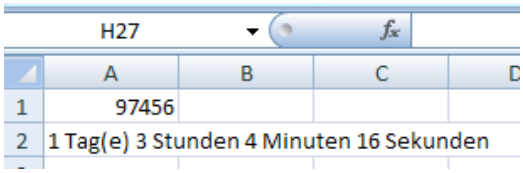


Abbildung 4.5
Screenshot „verarbeite Sekunden“

Die Stringoperatoren (& und +) dienen der Zeichenverkettung. Ihre Wirkung wird an Beispiel 4.7 schnell klar. Die Function *stringVerkettung* hat nach Durchlauf den Rückgabewert “abcd”.

Beispiel 4.7 *Stringverkettung*

```
'Dateiname: stringverkettung.xls
Option Explicit
Function verketteStrings() As String
    Dim s1 As String
    Dim s2 As String
    s1 = "ab"
    s2 = "cd"
    verketteStrings = s1 & s2
End Function
```

Vergleichs- und logische Operatoren werden wir im Zusammenhang mit Kontrollstrukturen besprechen.

Kapitel 5

Konditionalstrukturen, VBA-interne Funktionen

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

5.1.1 Beispiel und Erklärung

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 oder Programmblöcke häufiger ablaufen lassen. Dies ist aber ein echter Nachteil, wie das folgende einfache Beispiel veranschaulicht:
Programmieren wir ein Divisionsprogramm (wie unser Additionsprogramm), können wir Laufzeitfehler[1] nicht vermeiden, denn wenn der Benutzer 0 als Nenner eingibt, dann bricht das Programm mit einem Laufzeitfehler ab. Besser wäre, den Benutzer auf seinen Eingabefehler aufmerksam zu machen.
Beispiel 5.1 zeigt Ihnen sofort eine mögliche Realisierung dieser Problematik:

Beispiel 5.1 *Division durch Null*

```
'Dateiname: division.xls
Option Explicit 'damit wird die Deklaration von Variablen erzwungen
function division(zaehler As Double, nenner As Double) As Double
    dim quotient as Double 'Deklaration der Variablen quotient als Fließkommazahl
    if nenner = 0 then
        MsgBox("Versuch durch 0 zu teilen!")
        division = 0
    else
        quotient = zaehler / nenner
        division = quotient
    end if
End function
```

Wir fügen diese Funktion, wie in Abb. 5.1 dargestellt, in unsere Tabellenkalkulation ein. Wie wir sehen, funktioniert das Programm bei der Eingabe vernünftiger Werte richtig. Geben wir jedoch als Nenner 0 ein, erhalten wir folgende Situation: Ein kleines Fenster blendet auf, das uns mitteilt, dass der Nenner 0 ist. Dieses Fenster blockiert die Tabellenkalkulation, das heißt, der Benutzer kann, bevor er nicht auf OK geclickt hat, nichts machen. Gehen wir nun unser kleines Programm durch:
Neu ist die Zeile

```
if nenner = 0 then
```

if ist das reservierte Wort, dass die Konditionalstruktur (oder *if*-Anweisung, oder bedingte Anweisung) einleitet. Auf *if* folgt ein Vergleich. Dazu benötigt man die bereits in Kap. 4.5 vorgestellten Vergleichsoperatoren. Hier wird getestet,

[1] Laufzeitfehler (engl.: runtime errors) treten erst während des Programmablaufs auf und können verschiedene Ursachen haben, z.B. eine Division durch Null.

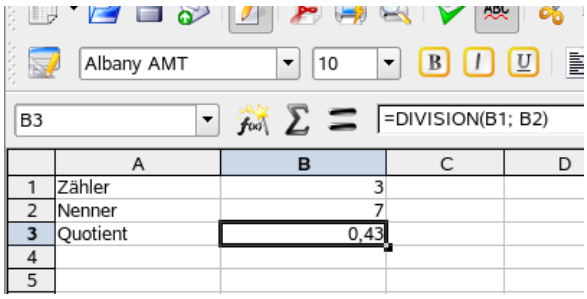


Abbildung 5.1
Das Divisionsprogramm in der Tabellenkalkulation

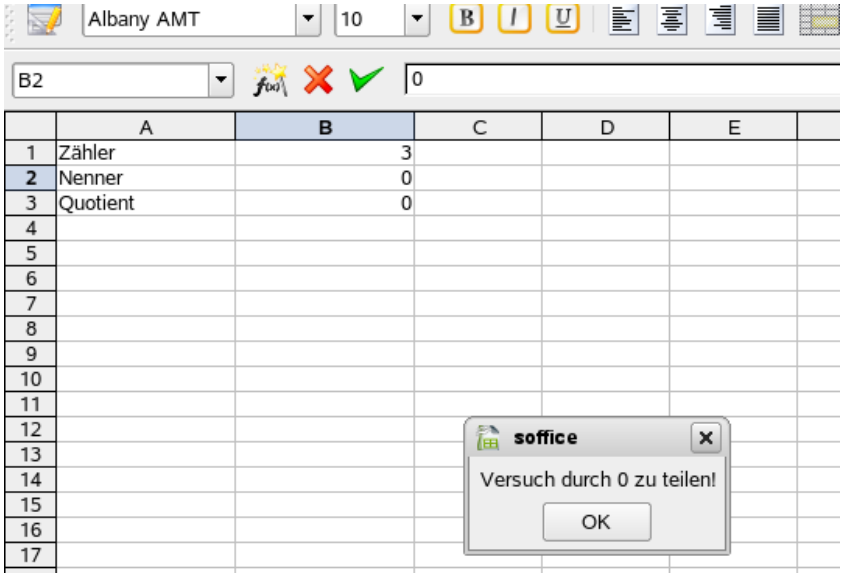


Abbildung 5.2
Das Divisionsprogramm in der Tabellenkalkulation mit Nenner 0

ob die Variable *nenner* den Wert 0 hat. Ist dies der Fall (der Vergleich ergibt *true*), werden die Anweisungen zwischen *then* und *else* durchgeführt. Dies ist die in Abb. 5.2 dargestellte Alternative. Es erscheint ein Fenster mit der Meldung, dass der Nenner 0 ist. Der Rückgabewert der Funktion wird ebenfalls auf 0 gesetzt und erscheint so, nachdem man das Meldungsfenster weggeclickt hat, in der Tabellenkalkulation.
Die Anweisung, die das Meldungsfenster aufblendet, ist:

```
MsgBox("Versuch durch 0 zu teilen!")
```

Ist der Vergleich hingegen nicht wahr (der Vergleich ergibt *false*) werden die Anweisungen zwischen *else* und *end if* durchgeführt. Dies ist die in Abb. 5.1 dargestellte Alternative. Die Division wird durchgeführt. Das Ergebnis wird in das Arbeitsblatt der Tabellenkalkulation geschrieben.
Der *else*-Teil ist optional. Ist kein *else*-Teil vorhanden, rückt das *end if* an die Stelle des *else*. Die Bedeutung ist nun: Ergibt der Vergleich *true*, wird der *then*-Teil durchgeführt, ansonsten wird das gesamt *if*-Konstrukt ignoriert. Wir veranschaulichen uns das an einer anderen Lösung des Divisionsprogramms (Beispiel 5.2):

Beispiel 5.2 *Division*

```
'Dateiname: division.xls
Option Explicit
```

```
function division2(zaehler As Double, nenner As Double) As Double
    dim quotient as Double
    if nenner = 0 then
        MsgBox("Versuch durch 0 zu teilen!")
        division2 = 0
        exit function
    end if
    quotient = zaehler / nenner
    division2 = quotient
end function
```

Neu hier ist die Anweisung *exit function*. Diese Anweisung bewirkt den sofortigen Abbruch der Funktion. Die Logik ist jetzt Folgende: Ergibt der Vergleich *true* (die Variable *nenner* hat den Wert 0), so wird der *then*-Teil des *if*-Konstrukts durchgeführt. Das bedeutet, das Fehlerfenster wird aufgeblendet, der Rückgabewert der Funktion wird auf 0 gesetzt und die Funktion wird durch *exit function* verlassen. Alle Anweisungen nach *exit function* werden nicht mehr durchgeführt, weil die Funktion ja verlassen wurde.
Ergibt der Vergleich hingegen *false* (der Wert der Variable *nenner* ist ungleich 0), so wird das komplette *if*-Konstrukt wegen des fehlenden *else*-Zweigs ignoriert. Dies bedeutet, die Funktion setzt mit der Anweisung

```
quotient=zaehler / nenner
```

fort, berechnet also den Quotienten und schreibt danach den ausgerechneten Quotienten in das Arbeitsblatt der Tabellenkalkulation.

5.1.2 Syntax

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. Bei logischen Ergebnissen gibt es nur zwei Möglichkeiten. Entweder ist der Ausdruck *true* oder er ist *false*.
Logische Ausdrücke sind daher im Wesentlichen:

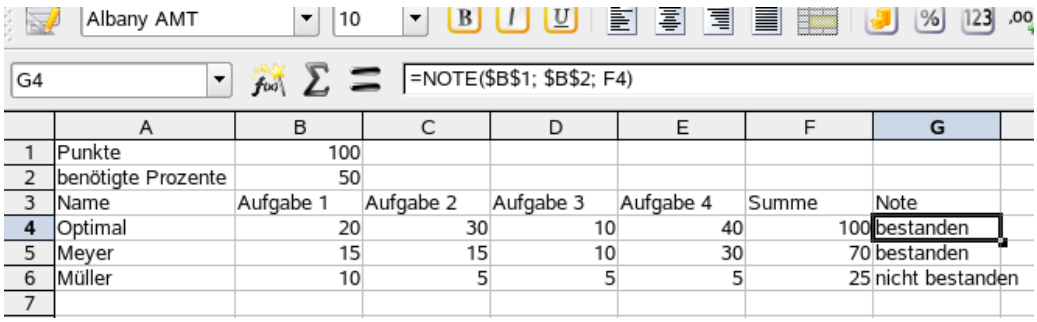
- Vergleiche (vgl. Kap. 4.5). Typische Vergleiche sind:
  if nenner = 0
  if (x > 3) or (y <= 4)
  Hierzu benötigt man die Vergleichs- und logischen Operatoren aus Kap. 4.5.
- bool’sche Variablen, also Variable, die Wahrheitswerte (*true* oder *false*) enthalten.

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.

5.1.3 Das Notenbeispiel

Als weiteres Beispiel wollen wir eine kleine Anwendung schreiben, die uns die Notenvergabe bei Klausuren erleichtert. Eingegeben werden soll die maximal erreichbare Punktzahl, der Prozentsatz, der zum Bestehen ausreicht sowie dann

in einzelnen Zeilen die erreichten Punkte pro Aufgabe und Teilnehmer. Ausgeben soll das Programm, ob der jeweilige Teilnehmer bestanden hat oder nicht. Abb. 5.3 zeigt das Benutzerinterface.



	A	B	C	D	E	F	G
1	Punkte	100					
2	benötigte Prozente	50					
3	Name	Aufgabe 1	Aufgabe 2	Aufgabe 3	Aufgabe 4	Summe	Note
4	Optimal	20	30	10	40	100	bestanden
5	Meyer	15	15	10	30	70	bestanden
6	Müller	10	5	5	5	25	nicht bestanden
7							

Abbildung 5.3
Das erste Notenprogramm

Betrachten wir zunächst die zugehörige Programmierung:

Beispiel 5.3 *Notenprogramm*

```
'Dateiname: noteIf.xls
Option Explicit
Function note(maximalpunkte As Long, benoetigteProzente As Double, _
                erreichtePunkte As Long) As String
    Dim benoetigtePunkte As Double
    benoetigtePunkte = (maximalpunkte * benoetigteProzente) / 100
    If erreichtePunkte >= benoetigtePunkte Then
        note = "bestanden"
    Else
        note = "nicht bestanden"
    End If
End Function
```

Das Programm benötigt aus dem Arbeitsblatt die Werte für *maximalpunkte*, *benoetigteProzente*, sowie *erreichtePunkte* eines jeden Teilnehmers. Zunächst werden dann die zum Bestehen nötigen Punkte ausgerechnet. Anschließend werden in einem *if*-Konstrukt die erreichten Punkte des Teilnehmers mit den zum Bestehen benötigten Punkten verglichen. Sind die erreichten Punkte größer gleich den benötigten Punkten, gibt die Funktion “bestanden” zurück, ansonsten “nicht bestanden”.
Beachten Sie bitte den _ in der Zeile:

```
function note(maximalpunkte As Long, benoetigteProzente As Double, _
```

Hier möchten wir die *function*-Anweisung in der nächsten Zeile fortsetzen, weil uns die Zeile sonst zu lang wird. U.A. passt Sie nicht mehr in das Layout des Scripts. Um eine Anweisung auf 2 Zeilen verteilen zu können, müssen wir an der Stelle, wo wir unterbrechen möchten, den _ setzen.
Betrachten wir nun wieder Abb. 5.3. Die erste Zelle, in die wir das Notenprogramm einfügen müssen, ist die Zelle G4. Hier geben wir

```
=note($B$1; $B$2; F4)
```

ein. Die Funktion schreibt sofort “bestanden” in die Zelle G4, schließlich steht dort ja das Optimalergebnis. Nun kopieren wir die Zelle G4, selektieren die Zellen G5 sowie G6 und fügen ein[2]. Sofort bewertet unsere Funktion die Teilnehmer. Beachten Sie, dass wir für die Zellen B1 und B2 absolute Bezüge verwendet haben (F4-Taste), damit die Tabellenkalkulation diese Zellbezüge beim Einfügen nicht anpasst.

[2]Die Erweiterung auf mehr als zwei Teilnehmer bleibt Ihnen überlassen.

5.2 Die if-elseif-Anweisung (Mehrseitige Auswahl)

5.2.1 Beispiel und Erklärung

In vielen Fällen reichen zwei Alternativen für die Entscheidungsfindung nicht aus. Um dies zu veranschaulichen, reicht eine nochmalige Betrachtung von Kap. 5.1.3. Normalerweise möchten wir nicht wissen, ob ein Teilnehmer einer Klausur bestanden hat, wir möchten jedem Teilnehmer eine Note geben. Hierfür gibt es nicht zwei, sondern elf Alternativen.
Wir betrachten aber zunächst ein einfacheres Beispiel mit weniger Alternativen. Wir wollen eine Funktion schreiben, die die Provision eines Vermittlers aufgrund eines geplanten Jahresumsatzes errechnet. Die Provision wird nach der in Tab. 5.1 dargestellten Staffelung berechnet:

Tabelle 5.1
Provisionsstaffelung

Umsatz	Provision (in %)
bis 100.000	0
100.000 bis 500.000	5
500.000 bis 1.000.000	10
Über 1.000.000	20

Das Benutzerinterface zeigt Abb. 5.4.

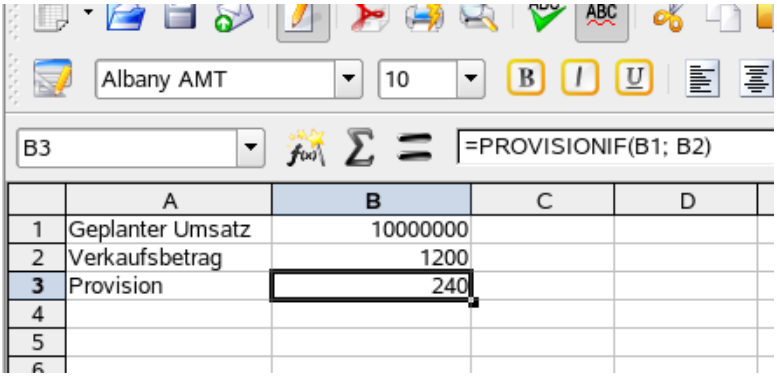


Abbildung 5.4
Das erste Provisionsprogramm mit if

Natürlich können wir diese Problematik mit der normalen *if*-Anweisung durch geschachtelte *if*'s lösen:

Beispiel 5.4 *Provision mit if*

```
'Dateiname: provision.xls
Option Explicit
Function provisionIf(geplanterUmsatz As Double, verkaufsbetrag As Double) As Double
    Dim provisionInProzent As Double
    If geplanterUmsatz >= 1000000 Then
        provisionInProzent = 0.2
    Else
        If geplanterUmsatz >= 500000 Then
            provisionInProzent = 0.1
        Else
            If geplanterUmsatz >= 100000 Then
                provisionInProzent = 0.05
            Else
                provisionInProzent = 0
            End If
        End If
    End If
```

```
    provisionIf = verkaufsbetrag * provisionInProzent
End Function
```

Durch das konsequente Einrücken ist der Code noch einigermaßen verständlich. Man kann gerade noch erkennen, welches *if* zu welchem *else* zu welchem *end if* gehört. Richtig schön ist aber anders und man sieht auch sofort, dass, wenn die Anzahl Alternativen größer wird, der Code immer unübersichtlicher wird. Für solche Fälle existiert in jeder Programmiersprache das *if elseif*-Konstrukt.
Und hier nun eine Lösung mit *if elseif*:

Beispiel 5.5 *Provision mit if elseif*

```
'Dateiname: provision.xls
Option Explicit
Function provisionIfElseIf(geplanterUmsatz As Double, verkaufsbetrag As Double) As Double
    Dim provisionInProzent As Double
    If geplanterUmsatz >= 1000000 Then
        provisionInProzent = 0.2
    ElseIf geplanterUmsatz >= 500000 Then
        provisionInProzent = 0.1
    ElseIf geplanterUmsatz >= 100000 Then
        provisionInProzent = 0.05
    Else
        provisionInProzent = 0
    End If
    provisionIfElseIf = verkaufsbetrag * provisionInProzent
End Function
```

Dieses Programm ist doch ziemlich selbsterklärend? Zunächst wird die Bedingung hinter *if* ausgewertet. Ergibt diese *true*, werden die Anweisungen zwischen *if* und dem ersten *elseif* ausgeführt. Danach wird das gesamte *if elseif* verlassen und das Programm fährt mit der auf den *if elseif*-Block folgenden Anweisung fort. Ergibt die erste Bedingung hingegen *false*, wertet VBA die auf das erste *elseif* folgende Bedingung aus. Ist diese *true*, werden die zwischen diesem und dem nächsten *elseif* stehenden Anweisungen durchgeführt und der *if elseif*-Block danach verlassen. Ergibt diese Bedingung jedoch *false*, wird –überraschenderweise – die auf das nächste *elseif* folgende Bedingung ausgewertet. Trifft keine der Bedingungen zu, wird der *else*-Teil ausgeführt. Natürlich nur, wenn es einen gibt, denn auch hier ist der *else*-Teil optional.
Zum Abschluss eine Realisierung unseres Provisionsbeispiels, wie es sein sollte, nämlich mit allen Provisionen und Provisonsgrenzen auf Konstanten. Diese Änderung des Programms ist selbsterklärend, daher verzichten wir auf weitere Erläuterungen.

Beispiel 5.6 *Provision mit if elseif und Konstanten*

```
'Dateiname: provision.xls
Option Explicit
Function provisionIfElseIfKonstante(geplanterUmsatz As Double, _
                verkaufsbetrag As Double) As Double
    Dim provisionInProzent As Double
    Const UMSATZGRENZE3 As Double = 1000000
    Const UMSATZGRENZE2 As Double = 500000
    Const UMSATZGRENZE1 As Double = 100000

    Const PROVISION_UMSATZGRENZE3 As Double = 0.2
    Const PROVISION_UMSATZGRENZE2 As Double = 0.1
    Const PROVISION_UMSATZGRENZE1 As Double = 0.05
    Const PROVISION_SONST As Double = 0

    If geplanterUmsatz >= UMSATZGRENZE3 Then
        provisionInProzent = PROVISION_UMSATZGRENZE3
    ElseIf geplanterUmsatz >= UMSATZGRENZE2 Then
        provisionInProzent = PROVISION_UMSATZGRENZE2
    ElseIf geplanterUmsatz >= UMSATZGRENZE1 Then
        provisionInProzent = PROVISION_UMSATZGRENZE1
    Else
        provisionInProzent = PROVISION_SONST
    End If
```

```
    provisionIfElseIfKonstante = verkaufsbetrag * provisionInProzent
End Function
```

5.2.2 Syntax

Die *if elseif*-Anweisung hat die Form:

```
if logischerAusdruck1 Then
    anweisung1
    :
    :
    anweisungN
elseif logischerAusdruck2 Then
    anweisung1
    :
    :
    anweisungN
elseif logischerAusdruckN Then
    anweisung1
    :
    :
    anweisungN
else
    anweisung1nachElse
    :
    :
    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 Anweisung hinter dem gesamten *if-elseif*-Block fortgesetzt. Wird *false* ermittelt, prüft VBA den logischen Ausdruck des nächsten *elseif*. Dies wird fortgesetzt bis:

- ein logischer Ausdruck *true* ergibt: In diesem Fall werden die Anweisungen zwischen diesem logischen Ausdruck und dem nächsten *elseif* durchgeführt. Danach setzt VBA mit der nächsten Anweisung hinter dem gesamten *if elseif*-Block fort.
- *else* erreicht wird: In diesem Fall wird der *else*-Teil durchgeführt. Danach setzt VBA mit der nächsten Anweisung hinter dem *if elseif*-Block fort.
- das Ende des *if elseif*-Blocks erreicht wird: *else* ist wie in Kap. 5.1 optional.

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

5.2.3 Das Notenbeispiel – Nutzung VBA-interner Funktionen

In diesem Kapitel werden wir das Notenprogramm so anpassen, dass nicht mehr bestanden oder nicht bestanden ausgegeben wird, sondern die wirkliche Note. Die Vorgehensweise hierbei ist:

1. Zunächst berechnen wir, wie in Kap. 5.1.3, die zum Bestehen notwendigen Punkte.
2. Die maximal erreichbaren Punkte minus die zum Bestehen notwendigen Punkte ist die Anzahl Punkte, die wir im Bereich, wo die Klausur bestanden ist, zur Verfügung haben (Beispiel: Wenn wir 240 Punkte vergeben, und 50 % zum Bestehen verlangen, dann sind 120 Punkte zum Bestehen notwendig, und die Anzahl Punkte im Bestehensbereich ist 240-120=120)[3]

[3] eigentlich nicht ganz korrekt, denn da wir bei 120 anfangen bestehen zu lassen, sind es in Wirklichkeit 121 Punkte im Bestehensbereich.

3. Es gibt vier „Hauptnoten“, mit denen man die Klausur besteht (1, 2, 3, 4). Wir haben also die in (2) errechnete Anzahl Punkte (dividiert durch vier) in jedem Notenbereich zur Verfügung (Beispiel fortgesetzt: Wenn wir, wie in (2), 120 Punkte im Bestehensbereich haben, dann haben wir 30 Punkte pro Note.). Nun können wir ausrechnen, bei welcher Punktzahl unsere „Hauptnoten“ beginnen.

   Note 4 beginnt bei den in (1) ausgerechneten zum Bestehen notwendigen Punkten.
   Note 3 beginnt bei zum Bestehen notwendigen Punkten plus Anzahl Punkten pro Note.
   Note 2 beginnt bei zum Bestehen notwendigen Punkten plus 2 * Anzahl Punkten pro Note.
   Note 1 beginnt bei zum Bestehen notwendigen Punkten plus 3 * Anzahl Punkten pro Note.

   Daraus folgt:

   Note 4 beginnt bei 120 Punkten,
   Note 3 beginnt bei 120 + 30 =150 Punkten,
   Note 2 beginnt bei 120 + 2*30 = 180 Punkten,
   Note 1 beginnt bei 120 + 3*30 = 210 Punkten.

4. Wir müssen noch die Zwischennoten festlegen. Hier gibt es zwei Fälle:

   bei den Hauptnoten 4 und 1 gibt es zwei Zwischennoten: 4 und 3,7 bzw. 1 und 1,3,
   bei den Hauptnoten 2 und 3 gibt es jeweils 3 Zwischennoten: 2,7; 3,0 und 3,3 bzw. 1,7; 2,0 und 2,3.

   Wir teilen also die in (3) ausgerechneten Punkte pro Note durch 2, um die Aufteilung bei 4 und 1 durchführen zu können sowie dann noch einmal durch 3 für die Aufteilung bei 3 und 2.

   Daraus folgt:

   30/2=15 Punkte für die Zwischennoten bei 4 und 1,
   30/3=10 für die Zwischennoten bei 3 und 2.

5. Dadurch liegen nun die Grenzen aller Noten fest:

   4,0 bei zum Bestehen notwendigen Punkten,
   3,7 bei zum Bestehen notwendigen Punkten plus Punkte pro Note durch 2,
   3,3 bei zum Bestehen notwendigen Punkten plus Punkte pro Note,
   3,0 bei Punkte 3,3 plus Punkte pro Note durch 3,
   2,7 bei Punkte 3,0 plus Punkte pro Note durch 3,
   2,3 bei zum Bestehen notwendigen Punkten plus 2*Punkte pro Note,
   2,0 bei Punkte 2,3 plus Punkte pro Note durch 3,
   1,7 bei Punkte 2,0 plus Punkte pro Note durch 3,
   1,3 bei zum Bestehen notwendigen Punkten plus 3*Punkte pro Note,
   1,0 bei Punkte 1,3 plus Punkte pro Note durch 2.

   Das bedeutet:

   4 bei 120 Punkten,
   3,7 bei 120 + 15 = 135 Punkten,
   3,3 bei 120 + 30 = 150 Punkten,
   3,0 bei 150 + 10 = 160 Punkten,
   2,7 bei 160 + 10 = 170 Punkten,
   2,3 bei 120 + 2*30 = 180 Punkten,
   2,0 bei 180 + 10 = 190 Punkten,
   1,7 bei 190 + 10 = 200 Punkten,
   1,3 bei 120 + 3*30 = 210 Punkten,
   1,0 bei 210 + 15 = 225 Punkten.

6. Sind die Ergebnisse der Divisionen nicht ganzzahlig, dann runden wir ab. Das ist teilnehmerfreundlich, denn dann beginnt ein Bereich tendenziell einen Punkt niedriger.

Ok, das war möglicherweise kompliziert ;-) Hier zur besseren Übersicht das Ganze in Tabellenform:

Tabelle 5.2
Notentabelle

Note	Punkte	Berechnung
1,0	225	120 + 30 + 30 + 30 + 15
1,3	210	120 + 30 + 30 + 30
1,7	200	120 + 30 + 30 + 20
2,0	190	120 + 30 + 30 + 10
2,3	180	120 + 30 + 30
2,7	170	120 + 30 + 20
3,0	160	120 + 30 + 10
3,3	150	120 + 30
3,7	135	120 + 15
4,0	120	120

Schauen wir uns dies nun in der Umsetzung an:

Beispiel 5.7 *Note mit if elseif (Korrekte Notenvergabe)*

```
'Dateiname: noteIfElseIf.xls
Option Explicit
Function noteIfElseIF(maximalpunkte As Long, benoetigteProzente As Double, _
                erreichtePunkte As Long) As String
    Dim benoetigtePunkte As Long
    Dim spanne As Long
    Dim punkteProNote As Long
    Dim punkteZweiZwischenNoten As Long
    Dim punkteDreiZwischenNoten As Long
    Dim grenze4_0 As Long
    Dim grenze3_7 As Long
    Dim grenze3_3 As Long
    Dim grenze3_0 As Long
    Dim grenze2_7 As Long
    Dim grenze2_3 As Long
    Dim grenze2_0 As Long
    Dim grenze1_7 As Long
    Dim grenze1_3 As Long
    Dim grenze1_0 As Long

    benoetigtePunkte = Int((maximalpunkte * benoetigteProzente) / 100)
    spanne = maximalpunkte - benoetigtePunkte
    punkteProNote = Int(spanne / 4)
    punkteZweiZwischenNoten = Int(punkteProNote / 2)
    punkteDreiZwischenNoten = Int(punkteProNote / 3)

    grenze4_0 = benoetigtePunkte
    grenze3_7 = benoetigtePunkte + punkteZweiZwischenNoten

    grenze3_3 = benoetigtePunkte + punkteProNote
    grenze3_0 = grenze3_3 + punkteDreiZwischenNoten
    grenze2_7 = grenze3_0 + punkteDreiZwischenNoten

    grenze2_3 = benoetigtePunkte + 2 * punkteProNote
    grenze2_0 = grenze2_3 + punkteDreiZwischenNoten
    grenze1_7 = grenze2_0 + punkteDreiZwischenNoten

    grenze1_3 = benoetigtePunkte + 3 * punkteProNote
    grenze1_0 = grenze1_3 + punkteZweiZwischenNoten

    If erreichtePunkte >= grenze1_0 Then
        noteIfElseIF = "1"
    ElseIf erreichtePunkte >= grenze1_3 Then
        noteIfElseIF = "1,3"
```

```
    ElseIf erreichtePunkte >= grenze1_7 Then
        noteIfElseIF = "1,7"
    ElseIf erreichtePunkte >= grenze2_0 Then
        noteIfElseIF = "2"
    ElseIf erreichtePunkte >= grenze2_3 Then
        noteIfElseIF = "2,3"
    ElseIf erreichtePunkte >= grenze2_7 Then
        noteIfElseIF = "2,7"
    ElseIf erreichtePunkte >= grenze3_0 Then
        noteIfElseIF = "3"
    ElseIf erreichtePunkte >= grenze3_3 Then
        noteIfElseIF = "3,3"
    ElseIf erreichtePunkte >= grenze3_7 Then
        noteIfElseIF = "3,7"
    ElseIf erreichtePunkte >= grenze4_0 Then
        noteIfElseIF = "4"
    Else
        noteIfElseIF = "5"
    End If
End Function
```

Dieses Programm ist nun etwas umfangreicher, aber eigentlich nicht wesentlich komplizierter als unsere vorhergehenden Programme. Zunächst haben wir eine ganze Reihe Variablendeklarationen. Dann beginnen die Berechnungen:

```
benoetigtePunkte=Int((maximalpunkte*benoetigteProzente)/100)
```

berechnet die Punktzahl, die zum Bestehen der Klausur notwendig ist. Neu hier ist die Funktion *Int*. *Int* ist eine in VBA vorhandene Funktion, die wir nutzen können. *Int* rundet ab. *Int(4.8)* ist also 4, genau wie *Int(4.1)*. Neben *Int* gibt es in VBA eine ganze Reihe von internen Funktionen (mathematische, Funktionen zur Behandlung von Zeichenketten usw.), die wir benutzen können. Interne Funktionen benutzt man immer auf die gleiche Art und Weise:

- Auf die linke Seite des Gleichheitszeichens schreiben wir die Variable, auf der wir das Ergebnis der Funktion speichern wollen.
- Auf die rechte Seite des Gleichheitszeichens dann den Namen der Funktion. In runden Klammern hinter den Funktionsnamen dann den Wert, auf den wir die Funktion anwenden wollen.

Hier wird zunächst der Term *(maximalpunkte*benoetigteProzente)/100* ausgerechnet. Auf das Ergebnis wendet VBA die Funktion *Int* an. Das heißt, das Ergebnis von *(maximalpunkte*benoetigteProzente)/100* wird abgerundet und dann auf der Variablen *benoetigtePunkte* gespeichert.

```
spanne = maximalpunkte-benoetigtePunkte
```

berechnet nun die Anzahl der Punkte im Bestehensbereich. Diese wird sodann durch 4 geteilt, um die Anzahl der Punkte pro Note zu erhalten:

```
punkteProNote=Int(spanne/4)
```

Danach errechnen wir die Werte für die Zwischennoten-Bestimmung, indem wir *punkteProNote* durch 2, resp. 3 teilen:

```
punkteZweiZwischenNoten=Int(punkteProNote/2)
punkteDreiZwischenNoten=Int(punkteProNote/3)
```

Immer dann, wenn wir nicht sicher sein können, dass das Ergebnis ganzzahlig ist, runden wir mit der Funktion *Int* ab. Dann bestimmen wir, wie oben beschrieben, die Grenzen für die einzelnen Noten.

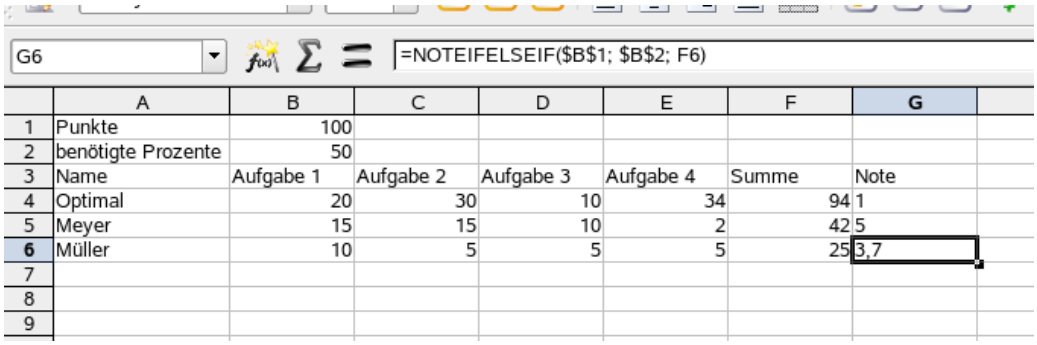
```
grenze4_0=benoetigtePunkte
grenze3_7=benoetigtePunkte+punkteZweiZwischenNoten

grenze3_3=benoetigtePunkte+punkteProNote
grenze3_0=grenze3_3+punkteDreiZwischenNoten
grenze2_7=grenze3_0+punkteDreiZwischenNoten
```

```
grenze2_3=benoetigtePunkte+2*punkteProNote
grenze2_0=grenze2_3+punkteDreiZwischenNoten
grenze1_7=grenze2_0+punkteDreiZwischenNoten

grenze1_3=benoetigtePunkte+3*punkteProNote
grenze1_0=grenze1_3+punkteZweiZwischenNoten
```

Der Rest des Programms bestimmt dann in einer großen *if elseIf*-Anweisung die Note des Teilnehmers. Dies ist aber völlig analog zu Beispiel 5.6[4], so dass wir auf eine Diskussion verzichten.
Beachten Sie bitte, dass Sie sowohl bei Beispiel 5.6, als auch hier in unserem Notenbeispiel immer mit der höchsten Alternative beginnen müssen. Denn wenn z.B. für eine Eins 210 Punkte erforderlich sind und für eine Vier 120, dann würde eine Umkehrung der Reihenfolge[5] dazu führen, dass alle Teilnehmer, die die Klausur bestanden haben, eine Vier erhalten.[6]
Das Userinterface unserer neuen Anwendung zeigt Abb. 5.5.



	A	B	C	D	E	F	G
1	Punkte	100					
2	benötigte Prozente	50					
3	Name	Aufgabe 1	Aufgabe 2	Aufgabe 3	Aufgabe 4	Summe	Note
4	Optimal	20	30	10	34	94	1
5	Meyer	15	15	10	2	42	5
6	Müller	10	5	5	5	25	3,7
7							
8							
9							

Abbildung 5.5
Das zweite Notenprogramm

Die Funktion wird exakt so, wie in Kap. 5.1.3 beschrieben, in die Tabellenkalkulation eingefügt. Einziger Unterschied: Die Funktion heißt *noteIfElseIF*.

5.3 Die select case-Anweisung (Mehrseitige Auswahl Teil 2)

5.3.1 Beispiel und Erklärung

Neben der *if elseIf*-Anweisung gibt es in VBA eine weitere Möglichkeit, Mehrfachauswahlen zu programmieren, die *select case*-Anweisung. Vom Inhaltlichen ist das jetzt nichts Neues. Es gibt nichts, was man mit *select case* machen kann und mit *if elseif* nicht. Daher zeigen wir sofort eine Implementierung der Provisionsanwendung aus Kap. 5.2.1 mit *select case*:

Beispiel 5.8 *Provision mit select case*

```
'Dateiname: provision.xls
Option Explicit
Function provisionSelectCase(geplanterUmsatz As Double, verkaufsbetrag As Double) As Double
    Dim provisionInProzent As Double
    Const UMSATZGRENZE3 As Double = 1000000
    Const UMSATZGRENZE2 As Double = 500000
    Const UMSATZGRENZE1 As Double = 100000

    Const PROVISION_UMSATZGRENZE3 As Double = 0.2
    Const PROVISION_UMSATZGRENZE2 As Double = 0.1
```

[4] Nur mit viel mehr Alternativen, wodurch das Programm ja auch so lang wird.
[5] Gemeint ist: Starten mit *if erreichtePunkte >= grenze_4 then*.
[6] Überlegen Sie sich selber, warum das so ist.

```
    Const PROVISION_UMSATZGRENZE1 As Double = 0.05
    Const PROVISION_SONST As Double = 0

    Select Case geplanterUmsatz
        Case Is >= UMSATZGRENZE3:
            provisionInProzent = PROVISION_UMSATZGRENZE3
        Case Is >= UMSATZGRENZE2:
            provisionInProzent = PROVISION_UMSATZGRENZE2
        Case Is >= UMSATZGRENZE1:
            provisionInProzent = PROVISION_UMSATZGRENZE1
        Case Else:
            provisionInProzent = PROVISION_SONST
    End Select
    provisionSelectCase = verkaufsbetrag * provisionInProzent
End Function
```

Beispiel 5.8 unterscheidet sich von Beispiel 5.6 nur durch das *select case*, das das *if elseIf* ersetzt. Im Unterschied zum *if elseif*-Konstrukt (dies besteht ja im Wesentlichen aus unabhängigen Bedingungen) wählt *select case* die auszuführenden Aktionen anhand der Werte einer einzigen Variablen aus. In Beispiel 5.8 ist dies die Variable *geplanterUmsatz*. Danach folgen die Auswahlblöcke. Jeder Auswahlblock wird mit *case* eingeleitet. Das hinter *case* stehende reservierte Wort *is* bezieht sich auf die Variable, anhand derer die Auswahl getroffen wird, hier also *geplanterUmsatz*. *is* wird durch den Wert der Variablen, also hier durch den Wert von *geplanterUmsatz* ersetzt. Die Anweisungen des ersten *case*-Blocks, deren Wert *true* ergibt, wird ausgeführt. Ergibt keine der Bedingungen *true*, wird der *case else*-Anwendungsblock ausgeführt, so er existiert.

5.3.2 Syntax

Die allgemeine Form eines *select case* -Konstrukts ist:

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

Selector ist die Variable, anhand derer die Mehrfachauswahl getroffen wird. Der Auswahlwert hinter *case* kann sein:

- ein fester Wert: Der Auswahlblock wird durchgeführt, wenn der Auswahlwert gleich dem Wert der Variablen ist. Beispiel:

```
select case quartal
    case 1:
        MsgBox("Erstes Quartal")
    case 2:
        MsgBox("Zweites Quartal")
    case 3:
        MsgBox("Drittes Quartal")
    case 4:
        MsgBox("Viertes Quartal")
    case else:
        MsgBox("Kein Quartal")
end select
```

- eine Menge oder ein Bereich. Beispiel:

```
select case monat
    case 1 to 3:
```

```
        MsgBox("Erstes Quartal")
    case 4 to 6:
        MsgBox("Zweites Quartal")
    case 7 to 9:
        MsgBox("Drittes Quartal")
    case 10 to 12:
        MsgBox("Viertes Quartal")
    case else:
        MsgBox("Kein Quartal")
end select
```

- eine Bedingung. In den Bedingungen wird die Variable, anhand der die Auswahl erfolgt, durch das Schlüsselwort *is* repräsentiert. Dies wurde schon in Beispiel 5.8 dargestellt. Hier ein weiteres Beispiel:

```
select case monat
    case is >= 10:
        MsgBox("Viertes Quartal")
    case is >= 7:
        MsgBox("Drittes Quartal")
    case is >= 4:
        MsgBox("Zweites Quartal")
    case is >= 1:
        MsgBox("Erstes Quartal")
    case else:
        MsgBox("Kein Quartal")
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 die beiden überein wird der zugehörige Auswahlblock durchgeführt. Danach wird *select case* verlassen und die Ausführung des VBA-Programms wird hinter *end select* fortgesetzt. Tritt keine Übereinstimmung auf, wird, soweit vorhanden, der Auswahlblock hinter *case else* durchgeführt. Ist *case else* nicht vorhanden, setzt die Ausführung sofort hinter *end select* fort. Auf jeden Fall ist auch hier sichergestellt, dass höchstens ein Auswahlblock durchgeführt wird.

5.3.3 Das Notenbeispiel

Hier zeigen wir nur den Code des Notenbeispiels mit einer Lösung mittels *select case* anstelle von *if elseIf*. Den Programmcode sollten Sie mit dem bisher Dargestellten ohne weitere Erläuterung verstehen.

Beispiel 5.9 *Note mit select case*

```
'Dateiname: noteSelectCase.xls
Option Explicit
Function noteSelectCase(maximalpunkte As Long, benoetigteProzente As Double, erreichtePunkte As ➥
    Long) As String
    Dim benoetigtePunkte As Long
    Dim spanne As Long
    Dim punkteProNote As Long
    Dim punkteZweiZwischenNoten As Long
    Dim punkteDreiZwischenNoten As Long
    Dim grenze4_0 As Long
    Dim grenze3_7 As Long
    Dim grenze3_3 As Long
    Dim grenze3_0 As Long
    Dim grenze2_7 As Long
    Dim grenze2_3 As Long
    Dim grenze2_0 As Long
    Dim grenze1_7 As Long
    Dim grenze1_3 As Long
    Dim grenze1_0 As Long

    benoetigtePunkte = Int((maximalpunkte * benoetigteProzente) / 100)
```

```
    spanne = maximalpunkte - benoetigtePunkte
    punkteProNote = Int(spanne / 4)
    punkteZweiZwischenNoten = Int(punkteProNote / 2)
    punkteDreiZwischenNoten = Int(punkteProNote / 3)

    grenze4_0 = benoetigtePunkte
    grenze3_7 = benoetigtePunkte + punkteZweiZwischenNoten

    grenze3_3 = benoetigtePunkte + punkteProNote
    grenze3_0 = grenze3_3 + punkteDreiZwischenNoten
    grenze2_7 = grenze3_0 + punkteDreiZwischenNoten

    grenze2_3 = benoetigtePunkte + 2 * punkteProNote
    grenze2_0 = grenze2_3 + punkteDreiZwischenNoten
    grenze1_7 = grenze2_0 + punkteDreiZwischenNoten

    grenze1_3 = benoetigtePunkte + 3 * punkteProNote
    grenze1_0 = grenze1_3 + punkteZweiZwischenNoten

    Select Case erreichtePunkte
        Case Is >= grenze1_0:
            noteSelectCase = "1"
        Case Is >= grenze1_3:
            noteSelectCase = "1,3"
        Case Is >= grenze1_7:
            noteSelectCase = "1,7"
        Case Is >= grenze2_0:
            noteSelectCase = "2"
        Case Is >= grenze2_3:
            noteSelectCase = "2,3"
        Case Is >= grenze2_7:
            noteSelectCase = "2,7"
        Case Is >= grenze3_0:
            noteSelectCase = "3"
        Case Is >= grenze3_3:
            noteSelectCase = "3,3"
        Case Is >= grenze3_7:
            noteSelectCase = "3,7"
        Case Is >= grenze4_0:
            noteSelectCase = "4"
        Case Else
            noteSelectCase = "5"
    End Select
End Function
```

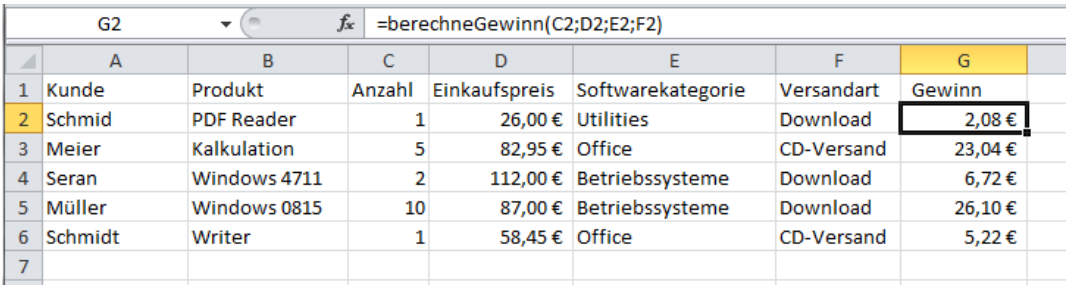
Kapitel 6

Benutzerdefinierte Funktionen: Weitere Beispiele

Mit dem bisher Gelernten können Sie nun die Funktionalität von Excel durch beliebige eigene Funktionen erweitern.

6.1 Das Gewinnbeispiel

Sie verkaufen Software und benötigen ein VBA-Programm, um die mit den Verkäufen erreichten Gewinne zu berechnen. Der Gewinn hängt vom Einkaufspreis, der gekauften Anzahl, der Softwarekategorie und der Versandart ab. Für Software der Kategorie "Betriebssysteme" kalkulieren Sie 3% des Einkaufspreises als Gewinn, für "Office"-Produkte 5% und für alle anderen Kategorien 8%. Wird die Software vom Kunden per Download erworben, entsteht kein weiterer Gewinn. Beim CD-Versand werden pauschal 2,30 Euro zusätzlicher Gewinn erzielt.
Folgende in Abb. 6.1 dargestellte Benutzerschnittstelle soll realisiert werden.



	A	B	C	D	E	F	G
1	Kunde	Produkt	Anzahl	Einkaufspreis	Softwarekategorie	Versandart	Gewinn
2	Schmid	PDF Reader	1	26,00 €	Utilities	Download	2,08 €
3	Meier	Kalkulation	5	82,95 €	Office	CD-Versand	23,04 €
4	Seran	Windows 4711	2	112,00 €	Betriebssysteme	Download	6,72 €
5	Müller	Windows 0815	10	87,00 €	Betriebssysteme	Download	26,10 €
6	Schmidt	Writer	1	58,45 €	Office	CD-Versand	5,22 €
7							

Abbildung 6.1
Benutzerdefinierte Funktionen: Gewinnbeispiel

Aus dem in Abb. 6.1 dargestellten Screenshot kann der Name der zu entwickelnden Funktion abgelesen werden. Die Funktion wird *berechneGewinn* heißen. Die Funktion muss vier Übergabeparameter akzeptieren:

- Die Anzahl der verkauften Produkte (Spalte C)
- Den Einkaufspreis (Spalte D)
- Die Softwarekategorie (Spalte E)
- Die Versandart (Spalte F)

Wir kommen nun zur Lösung:

Beispiel 6.1 *Benutzerdefinierte Funktionen: Gewinnbeispiel*

```
Function berechneGewinn(anzahl As Long, einkaufspreis As Double, kategorie As String, versandart ➥
    As String) As Double
    Const GEWINN_BETRIEBSSYSTEME As Double = 0.03
    Const GEWINN_OFFICE As Double = 0.05
    Const GEWINN_SONSTIGE As Double = 0.08

    Const GEWINN_VERSAND As Double = 2.3

    Dim gewinn As Double
    Dim gewinnProzent As Double

    If kategorie = "Betriebssysteme" Then
        gewinnProzent = GEWINN_BETRIEBSSYSTEME
    ElseIf kategorie = "Office" Then
        gewinnProzent = GEWINN_OFFICE
    Else
        gewinnProzent = GEWINN_SONSTIGE
    End If

    gewinn = anzahl * gewinnProzent * einkaufspreis

    If versandart = "CD-Versand" Then
        gewinn = gewinn + GEWINN_VERSAND
    End If

    berechneGewinn = gewinn
End Function
```

Gehen wir dieses kurze Programm im Einzelnen durch. Zunächst definieren wir Konstanten für die prozentualen Gewinne und den eventuellen zusätzlichen Gewinn durch den CD-Versand. Dies ist sehr sinnvoll, denn diese Zahlen können sich ändern. Dann brauchen wir nur die jeweilige Konstante anzupassen, ohne uns um die Logik kümmern zu müssen.

```
Const GEWINN_BETRIEBSSYSTEME As Double = 0.03
Const GEWINN_OFFICE As Double = 0.05
Const GEWINN_SONSTIGE As Double = 0.08

Const GEWINN_VERSAND As Double = 2.3
```

Wir deklarieren zwei Variablen, die wir für die Berechnung benötigen:

```
Dim gewinn As Double
Dim gewinnProzent As Double
```

Im nächsten Schritt bestimmen wir den prozentualen Gewinn in Abhängigkeit von der Versandart. Dies geschieht mit einem *if-elseif*-Konstrukt:

```
If kategorie = "Betriebssysteme" Then
    gewinnProzent = GEWINN_BETRIEBSSYSTEME
ElseIf kategorie = "Office" Then
    gewinnProzent = GEWINN_OFFICE
Else
    gewinnProzent = GEWINN_SONSTIGE
End If
```

Nun können wir den durch die Produktart, die Anzahl Verkäufe und den Einkaufspreis bestimmten Gewinn berechnen:

```
gewinn = anzahl * gewinnProzent * einkaufspreis
```

Im Falle der Versandart CD-Versand addieren wir den zusätzlichen Gewinn. Dies erfolgt durch eine einfache *if*-Anweisung:

```
If versandart = "CD-Versand" Then
    gewinn = gewinn + GEWINN_VERSAND
End If
```

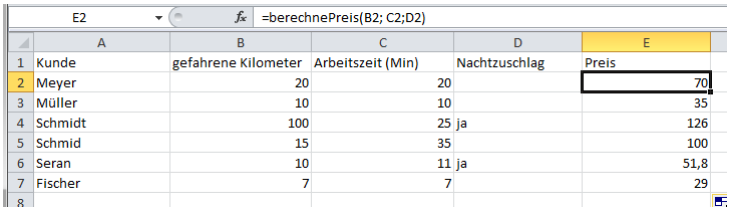
Zum Schluss belegen wir den Funktionsnamen mit dem ausgerechneten Ergebnis, damit das Resultat in der Excel-Tabelle erscheint.

```
berechneGewinn = gewinn
```

Das Euro-Zeichen hinter dem ausgerechneten Gewinn in Abb. 6.1 wird nicht vom VBA-Programm geschrieben, dies ist eine einfache Excel-Zellenformatierung.

6.2 Das Schlüsseldienstbeispiel

Sie sollen für einen Schlüsseldienst ein VBA-Programm schreiben, um den Preis eines Einsatzes zu berechnen. Der Preis hängt von den gefahrenen Kilometern, der Arbeitszeit und von einem eventuellen Nachtzuschlag ab. Sind die gefahrenen Kilometer kleiner gleich 10, wird eine Anfahrtspauschale von 15 Euro erhoben. Sind die gefahrenen Kilometer größer 10 aber kleiner gleich 20, wird eine Anfahrtspauschale von 30 Euro erhoben. Ansonsten beträgt die Anfahrtspauschale 40 Euro. Die Minute Arbeitszeit kostet 2 Euro. Wird ein Nachtzuschlag fällig (zu erkennen an dem Wert "ja" in der Spalte Nachtzuschlag), wird der bis dahin berechnete Preis um 40 Prozent erhöht.
In Abb. 6.2 dargestellte Benutzerschnittstelle soll realisiert werden.



	A	B	C	D	E
1	Kunde	gefahrene Kilometer	Arbeitszeit (Min)	Nachtzuschlag	Preis
2	Meyer	20	20		70
3	Müller	10	10		35
4	Schmidt	100	25	ja	126
5	Schmid	15	35		100
6	Seran	10	11	ja	51,8
7	Fischer	7	7		29
8					

Abbildung 6.2
Benutzerdefinierte Funktionen: Schlüsseldienst

Aus dem in Abb. 6.2 dargestellten Screenshot kann der Name der zu entwickelnden Funktion abgelesen werden. Die Funktion wird *berechnePreis* heißen. Die Funktion muss drei Übergabeparameter akzeptieren:

- Die gefahrenen Kilometer (Spalte B)
- Die Arbeitszeit (Spalte C)
- Der Nachtzuschlag (Spalte D)

Wir kommen nun zur Lösung:

Beispiel 6.2 *Benutzerdefinierte Funktionen: Schlüsseldienstbeispiel*

```
Function berechnePreis(gefahreneKilometer As Double, arbeitszeit As Double, nachtzuschlag As ➥
    String) As Double
    Const KILOMETER_GRENZE_1 As Double = 10
    Const KILOMETER_GRENZE_2 As Double = 20

    Const PREIS_KILOMETER_GRENZE_1 As Double = 15
    Const PREIS_KILOMETER_GRENZE_2 As Double = 30
    Const PREIS_KILOMETER_SONST As Double = 40

    Const PREIS_PRO_MINUTE As Double = 2
    Const NACHTZUSCHLAG_IN_PROZENT As Double = 0.4

    Dim anfahrtspreis As Double
    Dim preis As Double

    If gefahreneKilometer <= KILOMETER_GRENZE_1 Then
        anfahrtspreis = PREIS_KILOMETER_GRENZE_1
    ElseIf gefahreneKilometer <= KILOMETER_GRENZE_2 Then
```

```
        anfahrtspreis = PREIS_KILOMETER_GRENZE_2
    Else
        anfahrtspreis = PREIS_KILOMETER_SONST
    End If

    preis = anfahrtspreis + arbeitszeit * PREIS_PRO_MINUTE

    If nachtzuschlag = "ja" Then
        preis = preis * (1 + NACHTZUSCHLAG_IN_PROZENT)
    End If

    berechnePreis = preis
End Function
```

Gehen wir dieses kurze Programm im Einzelnen durch. Zunächst definieren wir Konstanten für die Grenzen der gefahrenen Kilometer, ihre jeweiligen Pauschalpreise, den Minutenpreis und den Nachtzuschlag. Dies ist sehr sinnvoll, denn diese Zahlen können sich ändern. Dann brauchen wir nur die jeweilige Konstante anzupassen, ohne uns um die Logik kümmern zu müssen.

```
Const KILOMETER_GRENZE_1 As Double = 10
Const KILOMETER_GRENZE_2 As Double = 20

Const PREIS_KILOMETER_GRENZE_1 As Double = 15
Const PREIS_KILOMETER_GRENZE_2 As Double = 30
Const PREIS_KILOMETER_SONST As Double = 40

Const PREIS_PRO_MINUTE As Double = 2
Const NACHTZUSCHLAG_IN_PROZENT As Double = 0.4
```

Wir deklarieren zwei Variablen, die wir für die Berechnung benötigen:

```
Dim anfahrtspreis As Double
Dim preis As Double
```

Im nächsten Schritt bestimmen wir Anfahrtspreis in Abhängigkeit von den gefahrenen Kilometern. Dies geschieht mit einem *if-elseif*-Konstrukt. Beachten Sie hier die Reihenfolge der Bedingungen. Da wir mit “kleiner gleich” vergleichen, müssen wir mit der kleinsten Kilometergrenze beginnen:

```
If gefahreneKilometer <= KILOMETER_GRENZE_1 Then
    anfahrtspreis = PREIS_KILOMETER_GRENZE_1
ElseIf gefahreneKilometer <= KILOMETER_GRENZE_2 Then
    anfahrtspreis = PREIS_KILOMETER_GRENZE_2
Else
    anfahrtspreis = PREIS_KILOMETER_SONST
End If
```

Nun können wir den durch die Anfahrtspauschale und die Arbeitszeit bestimmten Preis berechnen:

```
preis = anfahrtspreis + arbeitszeit * PREIS_PRO_MINUTE
```

Im Falle eines Nachtzuschlages multiplizieren wir mit dem zusätzlichen prozentualen Zuscchlag. Dies erfolgt durch eine einfache *if*-Anweisung:

```
If nachtzuschlag = "ja" Then
    preis = preis * (1 + NACHTZUSCHLAG_IN_PROZENT)
End If
```

Zum Schluss belegen wir den Funktionsnamen mit dem ausgerechneten Ergebnis, damit das Resultat in der Excel-Tabelle erscheint.

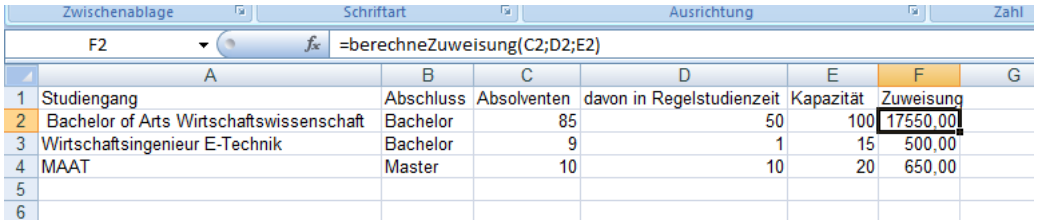
```
berechnePreis = preis
```

6.3 Das Zuweisungsbeispiel

Sie arbeiten in der Verwaltung einer Hochschule und sollen die Zuweisungen an Geld (Zahlungen) an die Hochschule für jeden Studiengang durch ein VBA-Programm ermitteln.
Die Zuweisung (Zahlung) richtet sich nach den Absolventen und der Kapazität des jeweiligen Studiengangs:

- Ist die Anzahl der Absolventen größer gleich 80% der Kapazität, so erhält die Hochschule 200 Euro pro Student, der sein Studium in Regelstudienzeit absolviert.
- Ist die Anzahl der Absolventen größer gleich 60% der Kapazität, so erhält die Hochschule 100 Euro pro Student, der sein Studium in Regelstudienzeit absolviert.
- Ansonsten erhält die Hochschule 50 Euro pro Student, der sein Studium in Regelstudienzeit absolviert.

Für jeden Studenten, der nicht in Regelstudienzeit fertig wird, wird 50% des oben ermittelten Betrages bezahlt.
Werden 50% oder mehr der Absolventen in Regelstudienzeit fertig, so wird die Zuweisung um 30% erhöht.
Die Zahlungen pro Student, die prozentualen Grenzen und die prozentuale Erhöhung der Zuweisung werden auf Konstanten abgelegt. In Abb. 6.3 dargestellte Benutzerschnittstelle soll realisiert werden.



	A	B	C	D	E	F	G
1	Studiengang	Abschluss	Absolventen	davon in Regelstudienzeit	Kapazität	Zuweisung	
2	Bachelor of Arts Wirtschaftswissenschaft	Bachelor	85	50	100	17550,00	
3	Wirtschaftsingenieur E-Technik	Bachelor	9	1	15	500,00	
4	MAAT	Master	10	10	20	650,00	
5							
6							

Abbildung 6.3
Benutzerdefinierte Funktionen: Zuweisung

Aus dem in Abb. 6.3 dargestellten Screenshot kann der Name der zu entwickelnden Funktion abgelesen werden. Die Funktion wird *berechneZuweisung* heißen. Die Funktion muss drei Übergabeparameter akzeptieren:

- Die Absolventen (Spalte C)
- Die Absolventen in Regelstudienzeit (Spalte D)
- Die Kapazität (Spalte E)

Wir kommen nun zur Lösung:

Beispiel 6.3 *Benutzerdefinierte Funktionen: Zuweisungsbeispiel*

```
Function berechneZuweisung(anzahlAbsolventen As Long, davonInRegelstudienzeit As Long, kapazitaet➥
    As Long) As Double
    Const PROZENT_GRENZE_1 As Double = 0.8
    Const PROZENT_GRENZE_2 As Double = 0.6
    Const EURO_PROZENT_GRENZE_1 As Double = 200
    Const EURO_PROZENT_GRENZE_2 As Double = 100
    Const EURO_SONST As Double = 50
    Const NICHT_REGELSTDIENZEIT_MALUS As Double = 0.5
    Const ZUSATZ_PROZENT_GRENZE As Double = 0.5
    Const ZUSATZ_PROZENT As Double = 0.3

    Dim anzahlNichtRegelstudienzeit As Long
    Dim euroProStudent As Double
    Dim zuweisung As Double

    If anzahlAbsolventen >= PROZENT_GRENZE_1 * kapazitaet Then
        euroProStudent = EURO_PROZENT_GRENZE_1
    ElseIf anzahlAbsolventen >= PROZENT_GRENZE_2 * kapazitaet Then
```

```
        euroProStudent = EURO_PROZENT_GRENZE_2
    Else
        euroProStudent = EURO_SONST
    End If

    anzahlNichtRegelstudienzeit = anzahlAbsolventen - davonInRegelstudienzeit
    zuweisung = davonInRegelstudienzeit * euroProStudent + anzahlNichtRegelstudienzeit * ➥
        euroProStudent * NICHT_REGELSTDIENZEIT_MALUS

    If davonInRegelstudienzeit >= anzahlAbsolventen * ZUSATZ_PROZENT_GRENZE Then
        zuweisung = zuweisung * (1 + ZUSATZ_PROZENT)
    End If

    berechneZuweisung = zuweisung
End Function
```

Gehen wir dieses kurze Programm im Einzelnen durch. Zunächst definieren wir Konstanten für die Prozentgrenzen, ihre jeweiligen Zuweisungen, den Malus für die Studenten, die nicht in Regelstudienzeit abgeschlossen haben sowie die Prozentgrenze und die Prozente für die eventuelle zusätzliche Zuweisung. Dies ist sehr sinnvoll, denn diese Zahlen können sich ändern. Dann brauchen wir nur die jeweilige Konstante anzupassen, ohne uns um die Logik kümmern zu müssen.

```
Const PROZENT_GRENZE_1 As Double = 0.8
Const PROZENT_GRENZE_2 As Double = 0.6
Const EURO_PROZENT_GRENZE_1 As Double = 200
Const EURO_PROZENT_GRENZE_2 As Double = 100
Const EURO_SONST As Double = 50
Const NICHT_REGELSTDIENZEIT_MALUS As Double = 0.5
Const ZUSATZ_PROZENT_GRENZE As Double = 0.5
Const ZUSATZ_PROZENT As Double = 0.3
```

Wir deklarieren drei Variablen, die wir für die Berechnung benötigen:

```
Dim anzahlNichtRegelstudienzeit As Long
Dim euroProStudent As Double
Dim zuweisung As Double
```

Im nächsten Schritt bestimmen wir die Zuweisung pro Student in Abhängigkeit von der Kapazität und der Anzahl Studenten. Dies geschieht mit einem *if-elseif*-Konstrukt. Beachten Sie hier die Reihenfolge der Bedingungen. Da wir mit "größer gleich" vergleichen, müssen wir mit der größten prozentualen Grenze beginnen:

```
If anzahlAbsolventen >= PROZENT_GRENZE_1 * kapazitaet Then
    euroProStudent = EURO_PROZENT_GRENZE_1
ElseIf anzahlAbsolventen >= PROZENT_GRENZE_2 * kapazitaet Then
    euroProStudent = EURO_PROZENT_GRENZE_2
Else
    euroProStudent = EURO_SONST
End If
```

Nun berechnen wir die Anzahl der Studenten nicht in Regelstudienzeit und können danach die Zuweisung bestimmen:

```
anzahlNichtRegelstudienzeit = anzahlAbsolventen - davonInRegelstudienzeit
zuweisung = davonInRegelstudienzeit * euroProStudent + _
        anzahlNichtRegelstudienzeit * euroProStudent * NICHT_REGELSTDIENZEIT_MALUS
```

Wenn wir genügend Studenten in Regelstudienzeit haben, multiplizieren wir mit dem zusätzlichen prozentualen Zuschlag. Dies erfolgt durch eine einfache *if*-Anweisung:

```
If davonInRegelstudienzeit >= anzahlAbsolventen * ZUSATZ_PROZENT_GRENZE Then
    zuweisung = zuweisung * (1 + ZUSATZ_PROZENT)
End If
```

Zum Schluss belegen wir den Funktionsnamen mit dem ausgerechneten Ergebnis, damit das Resultat in der Excel-Tabelle erscheint.

```
berechneZuweisung = zuweisung
```

Kapitel 7

Ereignisprozeduren

Tabellenkalkulationen kennen neben den bereits beschriebenen Funktionen noch eine weitere Möglichkeit, Code zur Verfügung zu stellen: die Prozeduren. Prozeduren unterscheiden sich nur in einem, aber wesentlichen, Punkt von Funktionen: Prozeduren können nicht als benutzerdefinierte Funktionen in Arbeitsblätter eingebunden werden, da sie über Ihren Namen keinen Wert zurückgeben.
Prozeduren werden auch nicht mit *function* eingeleitet (es sind ja auch keine Funktionen), sondern mit dem Schlüsselwort *sub*. Beendet werden sie mit *end sub*. Betrachten wir eine erste Prozedur:

Beispiel 7.1 *Das Programm „Hello World“ in einer Prozedur*

```
sub helloWorld()
    MsgBox("Hallo Welt!")
end sub
```

Das Programm ist selbsterklärend. Die Zeile

```
sub helloWorld()
```

leitet die Prozedur ein. Da Prozeduren keinen Wert zurückgeben, wird auch kein Datentyp angegeben.

```
    MsgBox("Hallo Welt!")
```

gibt den String „Hallo Welt“ in einer Messagebox aus und

```
end sub
```

legt fest, dass das Programm hier wieder endet.
Wie führen wir diese Prozedur jetzt aus?

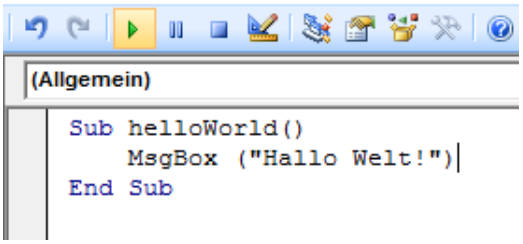


Abbildung 7.1
Prozedur Hallo Welt ausführen

Funktionen hatten wir in die Tabellenkalkulationen eingebunden. Dort werden sie automatisch ausgeführt, wenn sich ihre Zellbezüge ändern. Mit Prozeduren geht das nicht. Unsere erste Prozedur können wir zunächst nur im VBA-Editor

ausführen. Wir bewegen den Cursor in die Prozedur. Durch Betätigen des Abspielknopfs (das grüne Dreieck in Abbildung 7.1) im VBA-Editor wird die Prozedur dann ausgeführt.

Tabelle 7.1
Gegenüberstellung Funktion - Prozedur

Funktion	**Prozedur**
hat Rückgabewert	hat keinen Rückgabewert
wird eingebunden in Tabellenzelle	wird über Ereignis aufgerufen
function machwas() as string	sub machwas()
machwas="VBA Script"	MsgBox("VBA Script")
end function	end sub

7.1 Motivation (Erweiterung des Notenbeispiels)

Unser Notenbeispiel ist jetzt so weit, dass die Benotung einer Klausur automatisiert durchgeführt werden kann, was an sich schon eine gewisse Arbeitserleichterung ist. Wir sehen die Noten, erkennen aber an der Ausgabe nicht, ab wann unser Programm welche Note ausgegeben hat. Wir könnten uns die Punktegrenzen mit einem Taschenrechner selbst ausrechnen. Das ist aber nicht sonderlich schlau, weil wir dies immer wieder tun, und wir müssen, und wir ja eigentlich faul sind[1] Außerdem unterlaufen uns dann sicherlich immer mal wieder Fehler. Wir könnten uns nun die Punktegrenzen, die unser Programm ja ausrechnet, in MsgBoxen ausgeben lassen und abschreiben. Das wäre aber auch lästig. Umso mehr, wenn wir uns vorstellen, wir korrigieren gerade eine Grundstudiumsklausur mit 100 Teilnehmern und fügen daher unsere Funktion 100 mal in die Tabellenkalkulation ein. Die Funktion wird dann 100 mal ausgeführt, weil es 100 Teilnehmer zu bewerten gibt. Wenn die Funktion dann bei jeder Ausführung ihre Notengrenzen in MsgBoxen zeigt, dann wären das ca. 1000 MsgBoxen. Das ist nicht wirklich schön ☺.

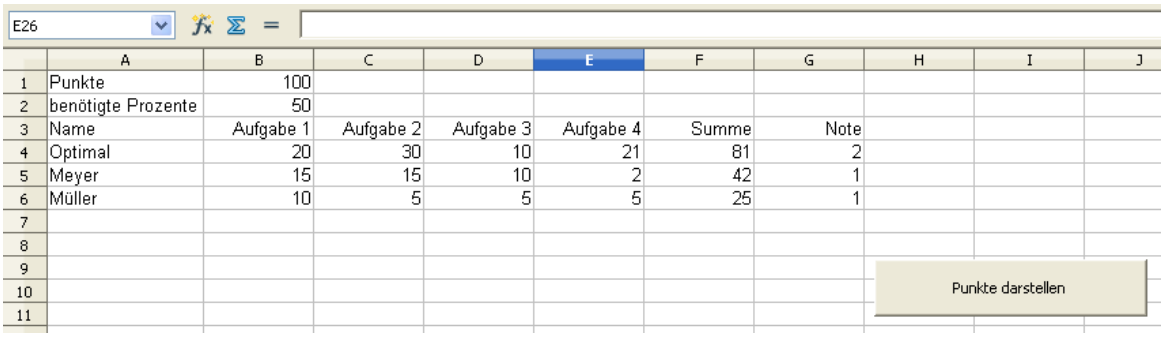
ℹ Übrigens können benutzerdefinierte Funktionen nur in die Zelle schreiben, in die sie eingebunden wurden.

Hinzu kommt, dass uns die Punktegrenzen in Wirklichkeit überhaupt nicht interessieren. Zum Zeitpunkt, wo wir bewerten, benötigen wir nur die Noten. Die Punktegrenzen sind für später, wenn die Teilnehmer wissen möchten, ab wann es welche Note gab und um wieviel sie evtl. eine bessere Note verpasst haben. Optimal für uns wäre, wenn wir in unserer Tabellenkalkulation eine Schaltfläche hätte, auf die wir einfach clicken könnten und ein dadurch gestartetes Programm schreibt die Punktegrenzen in die Zellen unseres Arbeitsblattes (vgl. Abb. 7.3). Und zwar am besten in ein zweites Arbeitsblatt der Tabellenkalkulation, dann können wir dorthin wechseln, die Punktegrenzen drucken und aushängen. Das gleiche Arbeitsblatt ist deswegen nicht so gut, weil da ja die Namen der Teilnehmer mit Noten stehen. Die dürfen wir schon aus Datenschutzgründen nicht so einfach drucken und aushängen.
Wie es aussehen könnte, zeigen Abb. 7.2 und Abb. 7.3.
Dort können Sie bereits sehen, dass das, was wir uns so vorgestellt haben, geht. Dafür gibt es nämlich Ereignisprozeduren. Wir können eine von uns geschriebene Prozedur mit einer Schaltfläche verbinden. Dann sorgt die Tabellenkalkulation dafür dass bei einem Click auf die Schaltfläche direkt die zugehörige Ereignisprozedur aufgerufen wird. Der Name Ereignisprozedur kommt übrigens daher, dass das Clicken auf eine Schaltfläche für eine Tabellenkalkulation ein Ereignis ist.

7.2 Ereignisprozeduren anhand des Notenbeispiels

Zunächst müssen wir die Schaltfläche in das Tabellenblatt bekommen. Dazu benutzen wir die Steuerelemente-Toolbox. Um dies zu tun, müssen Sie zunächst die Steuerelemente-Toolbox öffnen. Sie öffnen die Steuerelemente-Toolbox, indem Sie auf das Steuerelemente-Symbol in der Symbolleiste clicken oder indem Sie
☞Entwicklertools ⇨ Steuerelemente einfügen ⇨ ActiveX-Steuerelemente auswählen (vgl. Abb. 7.4).

[1] Zumindest bezüglich langweiligen, stupiden Tätigkeiten.



	A	B	C	D	E	F	G	H	I	J
1	Punkte	100								
2	benötigte Prozente	50								
3	Name	Aufgabe 1	Aufgabe 2	Aufgabe 3	Aufgabe 4	Summe	Note			
4	Optimal	20	30	10	21	81	2			
5	Meyer	15	15	10	2	42	1			
6	Müller	10	5	5	5	25	1			
7										
8										
9										
10										
11										

Abbildung 7.2
Schaltfläche zum Erzeugen des Noten-Punkte Zusammenhangs

	A	B
1	Note	Punkte
2	4	50
3	3,7	56
4	3,3	62
5	3	66
6	2,7	70
7	2,3	74
8	2	78
9	1,7	82
10	1,3	86
11	1	92

Abbildung 7.3
Tabellenblatt Noten-Punkte darstellen

Sie markieren das Schaltflächen-Symbol und zeichnen eine Schaltfläche auf das Tabellenblatt. Über das Kontextmenü (rechte Maustaste) können Sie die Eigenschaften der Schaltfläche verändern (vgl. Abb. 7.5). Ändern Sie dort zunächst die Eigenschaft *caption*. *caption* ist die Beschriftung der Schaltfläche. Wir ändern sie in „Punkte darstellen“. Danach ändern Sie den Namen der Schaltfläche. Sie soll *notenPunkteDarstellen* heißen (vgl. Abb. 7.6).
Doppel-Clicken Sie nun auf ihre neue Schaltfläche. Excel schaltet nun zum VBA-Editor um (vgl. Abb. 7.7). Innerhalb des Moduls mit dem Namen Ihrer Tabelle erstellt Excel das Skelett einer Prozedur.
Die Prozedur heißt *notenPunkteDarstellen_Click*.

Sie müssen übrigens immer, wenn Sie den Namen einer Schaltfläche ändern, den Namen der Ereignisprozedur anpassen. Excel macht das nicht automatisch.

Sollten Sie den Button später bearbeiten wollen, müssen Sie ihn zur Bearbeitung auswählen. Dies geschieht, indem Sie in der Steuerelemente-Toolbox den Entwurfs-Modus anwählen und dann auf den Button clicken. Selbst, wenn Sie den Button nur verschieben wollen, müssen Sie ihn in den Modus bringen.
Immer dann, wenn Sie nun auf die Schaltfläche mit dem Namen *notenPunkteDarstellen* clicken, führt Excel die Ereignisprozedur mit dem Namen *notenPunkteDarstellen_Click* aus. Schaltfläche und Ereignisprozedur werden also über Ihre Namen miteinander verbunden.
Zwischen dem Prozedurnamen und End Sub fügen Sie nun (wie immer) Ihren VBA-Code ein. Bevor wir die eigentliche Programmierung vorstellung, zeigen wir Ihnen in Abb. 7.8 den Algorithmus zu der Aufgabe.
Betrachten wir nun den VBA-Code, um die Aufgabenstellung in Excel zu lösen:

Beispiel 7.2 *Die Ereignisprozedur für die Punktegrenzen in Excel*

```
'Dateiname: noteEreignisprozedur.xls
Sub notenPunkteDarstellen_Click()
    ' Variablen deklarieren##################################################
```

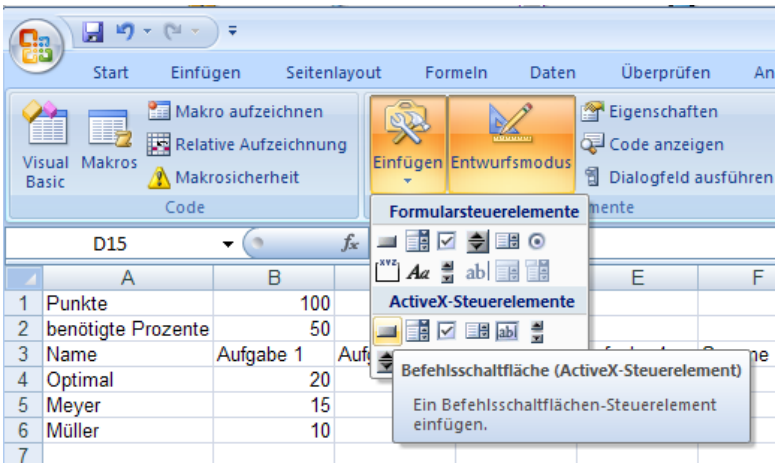


Abbildung 7.4
Öffnen der Steuerelemente Toolbox

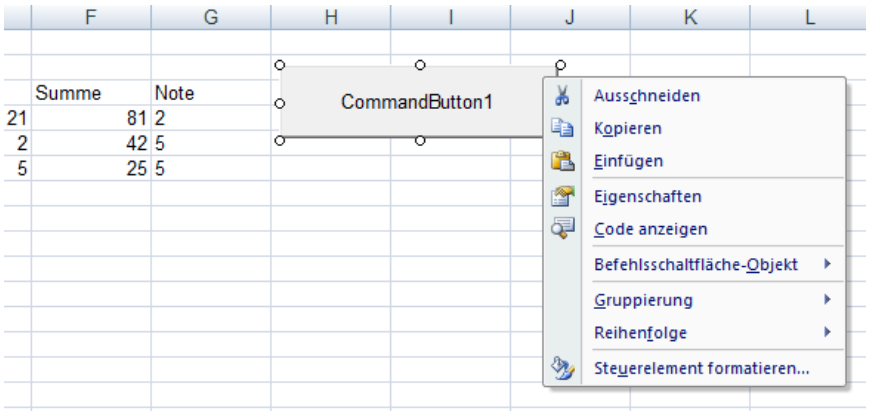


Abbildung 7.5
Eigenschaften der Schaltfläche

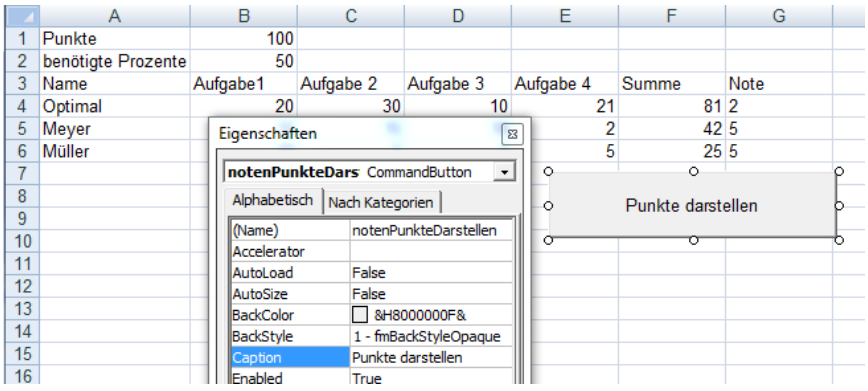


Abbildung 7.6
Eigenschaften der Schaltfläche verändern

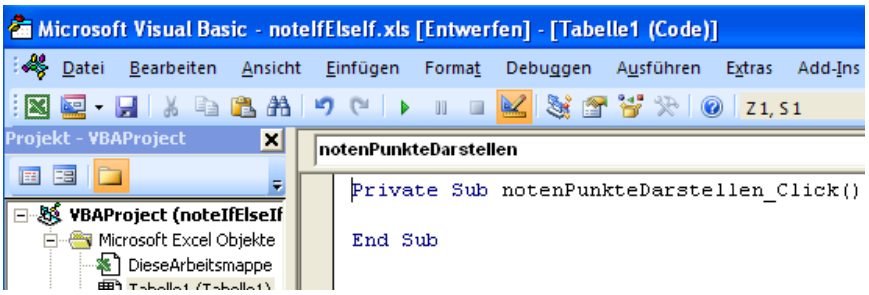


Abbildung 7.7
Code "hinter" Schaltfläche legen

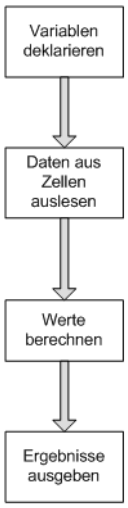


Abbildung 7.8
Programmablauf Noten-Punkte darstellen

```
    Const TABELLENBLATT As Long = 1
    Dim maximalpunkte As Long
    Dim benoetigteProzente As Double
    Dim benoetigtePunkte As Long
    Dim spanne As Long
    Dim punkteProNote As Long
    Dim punkteZweiZwischenNoten As Long
    Dim punkteDreiZwischenNoten As Long
    Dim grenze4_0 As Long
    Dim grenze3_7 As Long
    Dim grenze3_3 As Long
    Dim grenze3_0 As Long
    Dim grenze2_7 As Long
    Dim grenze2_3 As Long
    Dim grenze2_0 As Long
    Dim grenze1_7 As Long
    Dim grenze1_3 As Long
    Dim grenze1_0 As Long
    ' Werte auslesen#####################################################
    maximalpunkte = Sheets(TABELLENBLATT).Cells(1, 2)
    benoetigteProzente = Sheets(TABELLENBLATT).Cells(2, 2)
    ' Berechnungen durchführen ##########################################
    benoetigtePunkte = Int((maximalpunkte * benoetigteProzente) / 100)
    spanne = maximalpunkte - benoetigtePunkte
    punkteProNote = Int(spanne / 4)
    punkteZweiZwischenNoten = Int(punkteProNote / 2)
    punkteDreiZwischenNoten = Int(punkteProNote / 3)

    grenze4_0 = benoetigtePunkte
    grenze3_7 = benoetigtePunkte + punkteZweiZwischenNoten

    grenze3_3 = benoetigtePunkte + punkteProNote
    grenze3_0 = grenze3_3 + punkteDreiZwischenNoten
    grenze2_7 = grenze3_0 + punkteDreiZwischenNoten

    grenze2_3 = benoetigtePunkte + 2 * punkteProNote
    grenze2_0 = grenze2_3 + punkteDreiZwischenNoten
    grenze1_7 = grenze2_0 + punkteDreiZwischenNoten

    grenze1_3 = benoetigtePunkte + 3 * punkteProNote
    grenze1_0 = grenze1_3 + punkteZweiZwischenNoten
    ' Ergebnisse ausgeben ##############################################
    Sheets(2).Cells(1, 1) = "Note"
    Sheets(2).Cells(1, 2) = "Punkte"

    Sheets(2).Cells(2, 1) = "4"
    Sheets(2).Cells(2, 2) = grenze4_0

    Sheets(2).Cells(3, 1) = "3,7"
    Sheets(2).Cells(3, 2) = grenze3_7

    Sheets(2).Cells(4, 1) = "3,3"
    Sheets(2).Cells(4, 2) = grenze3_3

    Sheets(2).Cells(5, 1) = "3"
    Sheets(2).Cells(5, 2) = grenze3_0

    Sheets(2).Cells(6, 1) = "2,7"
    Sheets(2).Cells(6, 2) = grenze2_7

    Sheets(2).Cells(7, 1) = "2,3"
    Sheets(2).Cells(7, 2) = grenze2_3

    Sheets(2).Cells(8, 1) = "2"
    Sheets(2).Cells(8, 2) = grenze2_0

    Sheets(2).Cells(9, 1) = "1,7"
    Sheets(2).Cells(9, 2) = grenze1_7
```

```
    Sheets(2).Cells(10, 1) = "1,3"
    Sheets(2).Cells(10, 2) = grenze1_3

    Sheets(2).Cells(11, 1) = "1"
    Sheets(2).Cells(11, 2) = grenze1_0
End Sub
```

Nach der Deklaration der Prozedur finden Sie, wie immer, die Variablendeklarationen. Als erstes müssen wir jetzt *maximalpunkte* und *benoetigteProzente* aus dem Tabellenblatt lesen[2]. *maximalpunkte* steht in der Zelle B1. Den Wert der Zelle B1 bekommen wir in Excel u.A. durch folgende Zeile:

```
maximalpunkte = Cells(1, 2)
```

Cells erwartet als Übergaben die Zeile und die Spalte der Zelle, die angesprochen werden soll. Die Zelle B1 ist erste Zeile, zweite Spalte. Die Zählung der Spalten und Zeilen beginnt in Excel bei 1, die zweite Spalte ist also Spalte 2 (erste Spalte ist Spalte 1), die erste Zeile dementsprechend Zeile 1. Analog ermitteln wir *benoetigteProzente* aus der Zelle B2:

```
benoetigteProzente = Cells(2, 2)
```

Die nächsten Zeilen ermitteln dann nach dem bereits in Kap. 5.2.3 und Beispiel 5.7 dargestelltem Algorithmus die Punktegrenzen. Zum Schluss müssen wir noch unsere Ergebnisse in ein Tabellenblatt schreiben.
Schreiben in eine Zelle erfolgt nun völlig analog zum Lesen aus einer Zelle. Die Zeile

```
Sheets(2).Cells(1, 1) = "Punkte"
```

schreibt die Zeichenkette „Punkte“ in die Zelle A1 des zweiten Tabellenblattes. Durch *Sheets(2).Cells* sagen wir Excel, dass wir die Ausgabe nicht in das gerade aktuelle Tabellenblatt schreiben möchten, sondern in das zweite (zur Zeit noch leere) Tabellenblatt. Der ganze Rest dieses Programms schreibt also die errechneten Punktegrenzen in das zweite Tabellenblatt der Arbeitsmappe.

7.3 Wann benutzerdefinierte Funktion, wann Ereignisprozedur?

Benutzerdefinierte Funktionen benutzen wir, wenn wir den Wert einer Zelle in Abhängigkeit von den Werten anderer Zellen bestimmen wollen und wenn wir zudem wollen, dass der Wert der Zelle unserer benutzerdefinierten Funktion automatisch neu errechnet wird, wenn sich eine der Zellen, auf die sie sich bezieht, ändert. Dies ist völlig analog zu den in den Tabellenkalkulationen bereits vorhandenen Funktionen.
Ereignisprozeduren benutzen wir, wenn wir die Prozedur durch eine Aktion, z.B. einen Mausklick selber auslösen wollen.

[2] In Beispiel 5.7 hatten wir das ja über Zellbezüge der benutzerdefinierten Funktion gelöst.

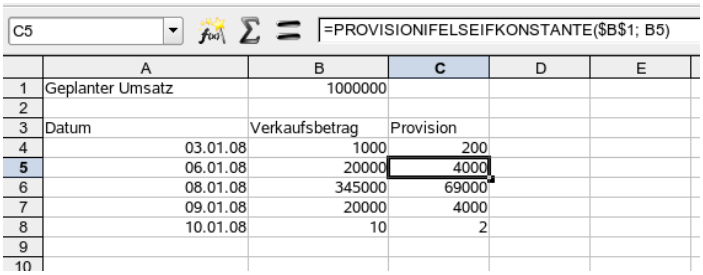
Kapitel 8

Wiederholungsanweisungen (Schleifen)

8.1 Die do-while-Anweisung

8.1.1 Motivation und Beispiel

Betrachten Sie die in Abb. 8.1 dargestellte Version unseres Provisionsbeispiels:



	A	B	C	D	E
1	Geplanter Umsatz	1000000			
2					
3	Datum	Verkaufsbetrag	Provision		
4	03.01.08	1000	200		
5	06.01.08	20000	4000		
6	08.01.08	345000	69000		
7	09.01.08	20000	4000		
8	10.01.08	10	2		
9					
10					

Abbildung 8.1
Mehrere Provisionsberechnungen in einer Tabelle

In der ersten Zeile sehen Sie den vereinbarten Jahresumsatz, der den Provisionsberechnungen zugrunde liegt. In den Zeilen darunter sind die Verkaufsbeträge der einzelnen Tage eingetragen. Unsere benutzerdefinierte Funktion zur Provisionsberechnung ist dort in die jeweiligen Zellen der Spalte C eingefügt. In die Zelle C5 ist z.B.

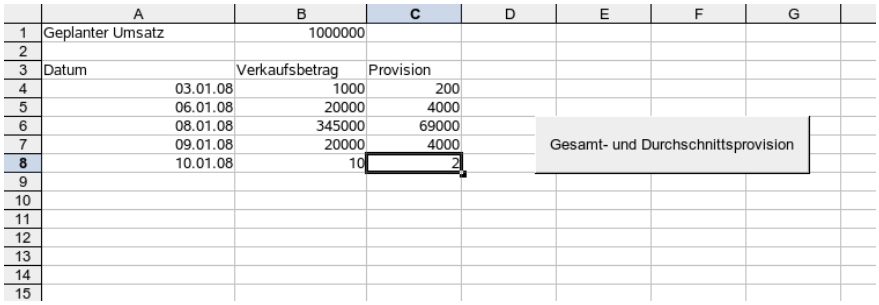
```
= provisionIfElseIfKonstante($B$1, B4)
```

eingetragen (vgl. wieder Abb. 8.1). Beachten Sie die $-Zeichen im absoluten Zellbezug des ersten Übergabeparameters. Hierdurch stellen wir sicher, dass dieser Zellbezug beim Kopieren der Formel nicht angepasst wird, so dass gewährleistet ist, dass immer der Umsatz aus Zelle B1 genutzt wird.
Nun nehmen Sie an, Sie müssen in unregelmäßigen Abständen für Ihren Abteilungsleiter

1. den bisher erzielten Umsatz,
2. die bisher erzielte Gesamtprovision,
3. die durchschnittliche Provision,
4. und die Anzahl Verkäufe

drucken.
Hier gibt es sicher viele Möglichkeiten, diese Aufgabe auch ohne Programmierung zu erledigen. Eine sehr gute Möglichkeit ist aber, eine Schaltfläche mit Ereignisprozedur in die Tabelle aufzunehmen. Wenn Ihr Abteilungsleiter nun die Ergebnisse wissen möchte, clicken Sie auf die Schaltfläche. Die Ereignisprozedur erzeugt die Ergebnisse und schreibt

diese in Tabellenblatt 2. Die Aufgabe ist ein für allemal gelöst. Die Ausführung dauert wenige Sekunden. Das Benutzerinterface der neuen Anwendung ist in Abb. 8.2 dargestellt, das Ergebnis in Abb. 8.3.

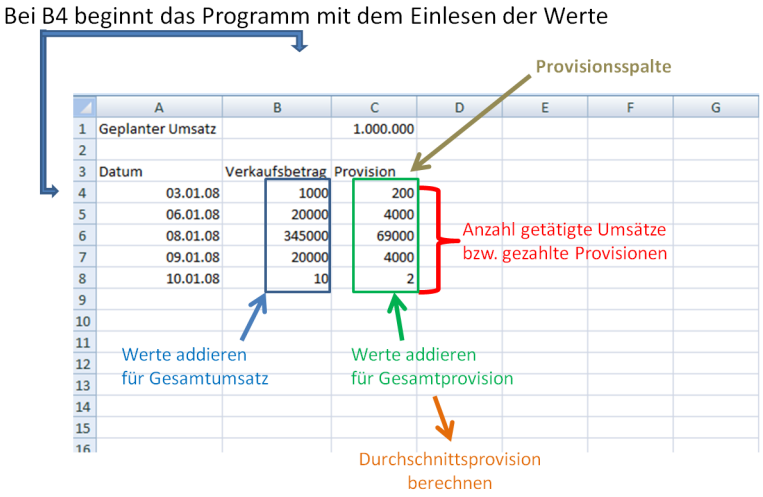


	A	B	C	D	E	F	G
1	Geplanter Umsatz	1000000					
2							
3	Datum	Verkaufsbetrag	Provision				
4	03.01.08	1000	200				
5	06.01.08	20000	4000				
6	08.01.08	345000	69000				
7	09.01.08	20000	4000				
8	10.01.08	10	2				
9							
10							
11							
12							
13							
14							
15							

Abbildung 8.2
Mehrere Provisionsberechnungen in einer Tabelle mit Schaltfläche

A	B	C	D	E
Anzahl Verkäufe	Gesamt Verkaufsbetrag	Gesamtprovision	Durchschnittliche Provision	
5	386010	77202	15440,4	

Abbildung 8.3
Aggregierte und Durchschnittsprovision in eigener Tabelle



	A	B	C	D	E	F	G
1	Geplanter Umsatz		1.000.000				
2							
3	Datum	Verkaufsbetrag	Provision				
4	03.01.08	1000	200				
5	06.01.08	20000	4000				
6	08.01.08	345000	69000				
7	09.01.08	20000	4000				
8	10.01.08	10	2				
9							
10							
11							
12							
13							
14							
15							
16							

Abbildung 8.4
Bildliche Darstellung der Aufgabe

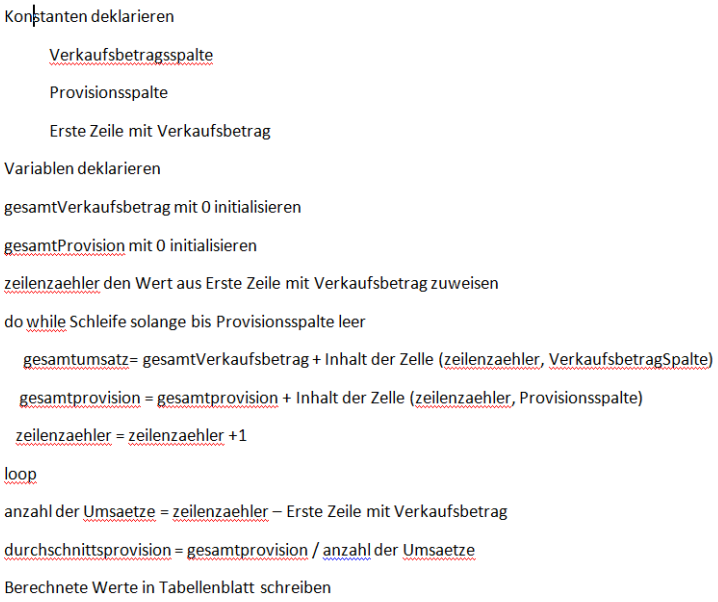


Abbildung 8.5
Algorithmus der Aufgabe

Wenn wir jetzt anfangen uns zu überlegen, wie wir das realisieren sollen, stellen wir schnell fest, dass dies mit den bisher besprochenen Mitteln nicht geht. Das Problem ist nämlich: Wir wissen während wir das Programm erstellen nicht, wie viele Provisionsberechnungen in unserer Tabelle sein werden, wenn das Programm ausgeführt wird. Daher muss das Programm dies feststellen. Wir gehen jetzt davon aus, dass in der Tabelle keine Leerzeilen zwischen den Provisionsberechnungen sind. Das heißt, wenn wir ermitteln könnten, in welcher Zeile die erste nicht gefüllte (also leere ;-)) Zelle in der Spalte C auftaucht, wäre die Problemstellung lösbar. Wir könnten dann die Inhalte aller Zellen der Spalte C bis zu dieser Zelle aufsummieren (dies ist die Gesamtprovision) und dann durch die gefundene Zeile - 3 teilen (die Verkaufsbeträge beginnen in der vierten Zeile). Unser Problem lässt sich also auf die Lösung des Problems, die erste leere Zelle in einer Spalte zu finden, reduzieren.
Aber wie soll das gehen? Wir müssen ja die Zelle C4 nehmen, testen, ob sie gefüllt ist. Wenn ja, dann müssen wir C5 testen, dann C6 und zwar solange bis wir die erste freie Zelle gefunden haben. „Solange bis“ ist das Zauberwort für Schleifen in der Informatik. Wir müssen nämlich die gleichen Programmschritte immer und immer wieder durchführen: Testen, ob eine Zelle leer ist, eine Zeile tiefer gehen, testen ob diese Zelle jetzt leer ist, eine Zeile tiefer gehen, testen ob diese Zelle jetzt leer ist, usw. solange bis wir die erste leere Zelle gefunden haben.
Wie schauen uns jetzt sofort die Realisierung an, anhand derer Sie das Konzept der *while*-Schleife sicher sofort verstehen werden.

8.1.2 Realisierung des Beispiels

Beispiel 8.1 *Bisheriger Verkaufsbetrag, Gesamt- und durchschnittliche Provision in Excel*

```
'Dateiname: provisionDoWhile.xls
Sub berechneGesamtUndDurchschnittsprovision_Click()
    Const VERKAUFSBETRAGSPALTE As Long = 2
    Const PROVISIONSSPALTE As Long = 3
    Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Long = 4
    Const TABELLENBLATT As Long = 1
    Const ERGEBNIS_TABELLENBLATT As Long = 2
    Dim gesamtprovision As Double
    Dim gesamtVerkaufsbetrag As Double
    Dim zeilenzaehler As Long
    Dim anzahlVerkaeufe As Long
```

```
    Dim durchschnittsprovision As Double

    gesamtprovision = 0
    gesamtVerkaufsbetrag = 0

    zeilenzaehler = ERSTE_ZEILE_MIT_VERKAUFSBETRAG
    Do While Not IsEmpty(Sheets(TABELLENBLATT).Cells(zeilenzaehler, PROVISIONSSPALTE))
        gesamtprovision = gesamtprovision + Sheets(TABELLENBLATT).Cells(zeilenzaehler, ➥
            PROVISIONSSPALTE)
        gesamtVerkaufsbetrag = gesamtVerkaufsbetrag + Sheets(TABELLENBLATT).Cells(zeilenzaehler, ➥
            VERKAUFSBETRAGSPALTE)
        zeilenzaehler = zeilenzaehler + 1
    Loop

    anzahlVerkaeufe = zeilenzaehler - ERSTE_ZEILE_MIT_VERKAUFSBETRAG
    If anzahlVerkaeufe = 0 Then
        MsgBox ("Noch kein Verkauf")
        Exit Sub
    End If

    durchschnittsprovision = gesamtprovision / anzahlVerkaeufe

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 1) = "Anzahl Verkäufe"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 2) = "Gesamtverkaufsbetrag"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 3) = "Gesamtprovision"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 4) = "Durchschnittliche Provision"

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 1) = anzahlVerkaeufe
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 2) = gesamtVerkaufsbetrag
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 3) = gesamtprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 4) = durchschnittsprovision
End Sub
```

Es handelt sich hier überraschenderweise um ein recht kleines Programm. Zunächst werden Variablen und Konstante deklariert. In den Zeilen

```
gesamtVerkaufsbetrag = 0
gesamtProvision = 0
```

werden die Variablen auf der wir den bisherigen Verkaufsbetrag und die Gesamtprovision speichern wollen, mit Null initialisiert, so dass wir nun bei jeder nicht leeren Zelle der Spalte C den Inhalt der Zelle auf *gesamtprovision* und den Inhalt der Zelle in der B-Spalte auf *gesamtVerkaufsbetrag* addieren können. Mit

```
zeilenzaehler = ERSTE_ZEILE_MIT_VERKAUFSBETRAG
```

wird die Variable *zeilenzaehler* mit der Zeile, in der die Verkaufsbeträge beginnen, initialisiert. Dann beginnt die Schleife. Die Schleife wird mit *do while* eingeleitet, sie endet mit *loop*. Die Anweisungen zwischen *do while* und *loop* sind die Anweisungen der Schleife. Auf *do while* folgt eine Bedingung:

```
Do While Not IsEmpty(Sheets(TABELLENBLATT).Cells(zeilenzaehler, PROVISIONSSPALTE))
```

Die Bedingung ist die Art und Weise, mit der man in Excel feststellt, ob eine Zelle leer ist oder nicht. Die Funktion *isEmpty* ist eine weitere Excel-interne Funktion, die wir, wie auch in Beispiel 5.7 beschrieben, nutzen können. *isEmpty* wird auf Zellen angewendet und gibt *true* zurück, wenn die übergebene Zelle leer ist ansonsten *false*. Die *do while*-Anweisung führt die zur Schleife gehörenden Anweisungen durch, wenn die auf *do while* folgende Bedingung *true* ergibt. In Abb. 8.2 ist dies der Fall. *zeilenzaehler* ist zur Zeit 4, *PROVISIONSSPALTE* hat den Wert 3, betrachtet wird also die Zelle C4 und diese Zelle ist gefüllt. In der Schleife werden zunächst die Gesamtprovision und der Gesamtverkaufsbetrag, das sind die Werte der aktuellen Zeile (zur Zeit gerade 4), erhöht. Dann wird *zeilenzaehler* um 1 erhöht.

```
gesamtprovision = gesamtprovision + Sheets(TABELLENBLATT).Cells(zeilenzaehler, PROVISIONSSPALTE)
gesamtVerkaufsbetrag = gesamtVerkaufsbetrag + Sheets(TABELLENBLATT).Cells(zeilenzaehler, ➥
    VERKAUFSBETRAGSPALTE)
zeilenzaehler = zeilenzaehler + 1
```

Die Anweisung *loop* bewegt VBA nun, zur Zeile mit der *do while*-Anweisung zurückzukehren. Jedesmal, wenn VBA zur *do while*-Anweisung zurückkehrt, wird die auf *do while* folgende Bedingung ausgewertet. Hier ergibt sich aber nun eine Veränderung: Da *zeilenzaehler* um 1 erhöht wurde (ist dann derzeit 5) und durch die bei Eins beginnende Zeilenzählung wird nun die Zelle C5 getestet. Wir sind also eine Zeile nach unten gegangen. Ergibt die Auswertung der Bedingung *true*, werden die Anweisungen der Schleife erneut durchlaufen, ansonsten setzt VBA mit der auf *loop* folgenden Anweisung fort. Und damit haben wir genau, was wir wollen. Die Schleife läuft solange, bis die erste leere Zelle in der Spalte C gefunden wird. Bis dahin addiert sie die Werte aller gefüllten Zellen auf *gesamtprovision* und *gesamtVerkaufsbetrag*.
Wir bestimmen durch

```
anzahlVerkaeufe=zeilenzaehler - ERSTE_ZEILE_MIT_VERKAUFSBETRAG
```

die Anzahl der bisherigen Verkäufe. Sollte noch gar nichts gewesen sein, bricht das Programm mit einer Fehlermeldung ab:

```
if anzahlVerkaeufe = 0 then
    MsgBox("Noch kein Verkauf")
    exit sub
end if
```

Ansonsten sind wir jetzt in der Lage, die Durchschnittsprovision zu berechnen:

```
durchschnittsprovision = gesamtprovision / anzahlVerkaeufe
```

Abschließend werden die ausgerechneten Werte, wie in Beispiel 7.2 beschrieben, in das zweite Tabellenblatt geschrieben.

8.1.3 Syntax

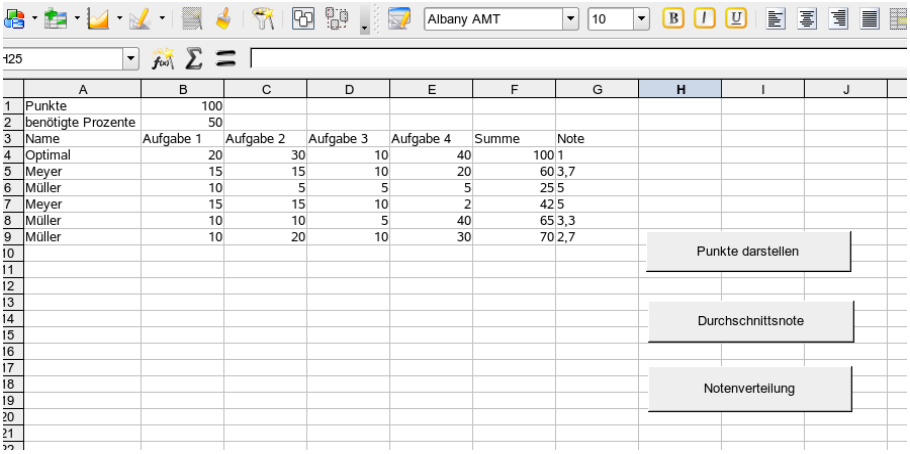
Die *Do While*-Schleife hat die Form:

```
Do While logischer Ausdruck
    anweisung1
    :
    :
    anweisungN
Loop
```

Wirkung: Wenn das Programm auf *Do While* trifft, wird zunächst *logischer Ausdruck* ausgewertet. Ergibt *logischer Ausdruck* bei der ersten Auswertung den Wert *false*, wird die gesamte Schleife ignoriert, also nicht ausgeführt[1]. 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.
Mit Schleifen lassen sich aber auch sehr interessante Effekte erzielen: Wird *logischer Ausdruck* nämlich nie *false* wird die Schleife nie abgebrochen. Die Anweisungen der Schleife werden unendlich oft wiederholt. Solche Schleifen bezeichnet man als Endlosschleifen. Dies ist selten vom Programmierer so beabsichtigt, da das Programm nur noch durch Beenden der Tabellenkalkulation mit Hilfe des Tast Managers angehalten werden kann. Der Wert von *logischer Ausdruck* muss also innerhalb der Schleife geändert werden.

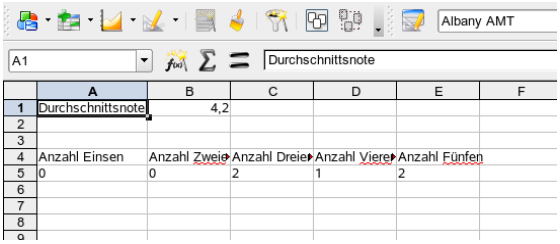
8.1.4 Das Notenbeispiel mit ersten Statistiken

Wir wollen unser Klausurauswertungsprogramm weiter verbessern. Wir möchten auf Knopfdruck die Durchschnittsnote der Klausur sowie die Notenverteilung (also wie viele Einsen, Zweien, Dreien, Vieren und Fünfen es gegeben hat) erzeugen. Die Ergebnisse schreiben wir in das dritte Tabellenblatt. Im ersten befindet sich ja die Bewertung der Klausur. Das sind personenbezogene Daten, die dürfen wir nicht drucken. Und die Durchschnittsnote und die Notenverteilung könnten wir schon mal drucken und aushängen. Tabellenblatt 2 ist auch schon belegt, da haben wir ja die Noten-Punkte-Verteilung hinein geschrieben.
Das Benutzerinterface der neuen Anwendung ist in Abb. 8.6 dargestellt, das Ergebnis in Abb. 8.7.



	A	B	C	D	E	F	G
1	Punkte	100					
2	benötigte Prozente	50					
3	Name	Aufgabe 1	Aufgabe 2	Aufgabe 3	Aufgabe 4	Summe	Note
4	Optimal	20	30	10	40	100	1
5	Meyer	15	15	10	20	60	3,7
6	Müller	10	5	5	5	25	5
7	Meyer	15	15	10	2	42	5
8	Müller	10	10	5	40	65	3,3
9	Müller	10	20	10	30	70	2,7

Abbildung 8.6
Statistiken für die Klausurauswertung: Benutzerinterface



	A	B	C	D	E	F
1	Durchschnittsnote	4,2				
2						
3						
4	Anzahl Einsen	Anzahl Zwei	Anzahl Dreier	Anzahl Vierer	Anzahl Fünfen	
5	0	0	2	1	2	

Abbildung 8.7
Statistiken für die Klausurauswertung: das Ergebnis

Hier haben wir die bereits in Kap. 8.1.1 dargestellte Problematik. Unser Programm muss die erste leere Zelle einer Spalte ermitteln, um zu wissen, wann es abbrechen muss. Darüber hinaus muss es aber zuerst die erste freie Zelle einer Zeile bestimmen, weil wir ja auch nicht wissen, wie viele Aufgaben die Klausur hatte; unser Programm soll immer funktionieren, egal wie viele Aufgaben gestellt wurden.
Das ist aber nun eigentlich auch nichts wirklich Schweres mehr. Das können wir auch mit einer *do while*-Schleife machen. Wir laufen nur nicht nach unten über die Zeilen, sondern nach rechts über die Spalten bis wir die erste leere Zelle finden.

8.1.5 Realisierung des Notenbeispiels

Beginnen wir mit dem Programm zur Berechnung der Durchschnittsnote, das ist nämlich kürzer.

Beispiel 8.2 *Durchschnittsnote*

```
'Dateiname: noteEreignisprozedurDoWhile
Sub berechneDurchschnittsnote_Click()
    Const OPTIMALZEILE As Long = 4
    Const TABELLENBLATT As Long = 1
    Const ERGEBNIS_TABELLENBLATT As Long = 3
    Dim spaltenzaehler As Long
    Dim aktuelleZeile As Long
    Dim letzteBesetzteSpalte As Long
    Dim anzahlTeilnehmer As Long
    Dim notenSumme As Double
    Dim durchschnittsnote As Double
```

[1] Schleifen, deren Abbruchbedingung bereits vor dem ersten Schleifendurchlauf geprüft werden, bezeichnet man als abweisende Schleifen.

```
    spaltenzaehler = 1
    anzahlTeilnehmer = 0
    notenSumme = 0
    Do While Not IsEmpty(Sheets(TABELLENBLATT).Cells(OPTIMALZEILE, spaltenzaehler))
        spaltenzaehler = spaltenzaehler + 1
    Loop
    letzteBesetzteSpalte = spaltenzaehler - 1

    aktuelleZeile = OPTIMALZEILE + 1
    Do While Not IsEmpty(Sheets(TABELLENBLATT).Cells(aktuelleZeile, letzteBesetzteSpalte))
        anzahlTeilnehmer = anzahlTeilnehmer + 1
        notenSumme = notenSumme + Sheets(TABELLENBLATT).Cells(aktuelleZeile, letzteBesetzteSpalte➥
            )
        aktuelleZeile = aktuelleZeile + 1
    Loop
    durchschnittsnote = notenSumme / anzahlTeilnehmer

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(4, 1) = "Durchschnittsnote"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(4, 2) = durchschnittsnote
End Sub
```

Auch hier werden zunächst die für die Prozedur benötigten Variablen und Konstanten deklariert. Wegen des Beginns der Zeilenzählung bei 1 von Excel hat die Konstante *OPTIMALZEILE* den Wert 4. *OPTIMALZEILE* ist die Zeile, in der wir die optimalen Punkte eingetragen haben. Diese Zeile werden wir benutzen, um die letzte besetzte Spalte zu identifizieren. Dies ist ja die Spalte mit den Noten.
Nach den Variableninitialisierungen prüft unser Programm, ob die Zelle A4 (4te Zeile, 1te Spalte) leer ist, genau wie im Provisionsbeispiel. Diese Zelle ist nicht leer, also führt das Programm die Schleife aus. Hier wird *spaltenzaehler* um eins erhöht. Die Schleife beginnt von Neuem und prüft, ob die Zelle in der nächsten Spalte leer ist. Dies läuft so lange, bis die erste leere Zelle in der Zeile *OPTIMALZEILE* ermittelt wurde. Dann endet die Schleife. Die Spaltennummer der ersten leeren Zelle steht dann auf *spaltenzaehler*. Die Spaltennummer der letzten besetzten Zelle in *OPTIMALZEILE* ist dann natürlich *spaltenzaehler - 1*. Dies ist die Spalte mit den Noten.
Die Zeilennummer des ersten richtigen Teilnehmers ist *OPTIMALZEILE + 1*. Dies wird auf der Variablen *aktuelleZeile* gespeichert. Mit dieser Zeile starten wir dann unsere zweite *do while*-Schleife, die völlig analog zu Beispiel 8.1 die Noten aufsummiert und die Teilnehmer zählt. Nun wird nur noch die Durchschnittsnote gebildet und das Ergebnis danach in das dritte Tabellenblatt geschrieben.
Als nächstes nun das Programm für die Notenverteilung:

Beispiel 8.3 *Notenverteilung*

```
'Dateiname: noteEreignisprozedurDoWhile
Sub bestimmeNotenverteilung_Click()
    Const OPTIMALZEILE As Long = 4
    Const TABELLENBLATT As Long = 1
    Const ERGEBNIS_TABELLENBLATT As Long = 3
    Dim spaltenzaehler As Long
    Dim aktuelleZeile As Long
    Dim letzteBesetzteSpalte As Long
    Dim anzahlEinsen As Long
    Dim anzahlZweien As Long
    Dim anzahlDreien As Long
    Dim anzahlVieren As Long
    Dim anzahlFuenfen As Long
    Dim note As String

    spaltenzaehler = 1
    anzahlEinsen = 0
    anzahlZweien = 0
    anzahlDreien = 0
    anzahlVieren = 0
    anzahlFuenfen = 0

    Do While Not IsEmpty(Sheets(TABELLENBLATT).Cells(OPTIMALZEILE, spaltenzaehler))
        spaltenzaehler = spaltenzaehler + 1
    Loop
```

```
    letzteBesetzteSpalte = spaltenzaehler - 1

    aktuelleZeile = OPTIMALZEILE + 1
    Do While Not IsEmpty(Sheets(TABELLENBLATT).Cells(aktuelleZeile, letzteBesetzteSpalte))
        note = Sheets(TABELLENBLATT).Cells(aktuelleZeile, letzteBesetzteSpalte)
        If (note = "1") Or (note = "1,3") Then
            anzahlEinsen = anzahlEinsen + 1
        ElseIf (note = "1,7") Or (note = "2") Or (note = "2,3") Then
            anzahlZweien = anzahlZweien + 1
        ElseIf (note = "2,7") Or (note = "3") Or (note = "3,3") Then
            anzahlDreien = anzahlDreien + 1
        ElseIf (note = "3,7") Or (note = "4") Then
            anzahlVieren = anzahlVieren + 1
        ElseIf note = "5" Then
            anzahlFuenfen = anzahlFuenfen + 1
        End If
        aktuelleZeile = aktuelleZeile + 1
    Loop

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 1) = "Anzahl Einsen"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 2) = "Anzahl Zweien"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 3) = "Anzahl Dreien"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 4) = "Anzahl Vieren"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 5) = "Anzahl Fünfen"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 1) = anzahlEinsen
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 2) = anzahlZweien
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 3) = anzahlDreien
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 4) = anzahlVieren
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 5) = anzahlFuenfen
End Sub
```

Dieses Programm ist zwar etwas länger, enthält aber keine neuen Konzepte: In einer ersten *do while*-Schleife wird, wie in Beispiel 8.2 die letzte besetzte Spalte und damit die Spalte mit der Note ermittelt. Danach startet wieder eine Schleife über alle Teilnehmer, die bereits benotet wurden. Hier ermitteln wir mit einem *if elseif*, ob der jeweilige Teilnehmer eine Eins, Zwei, Drei, Vier oder Fünf geschrieben hat. Davon abhängig wird dann eine der Variablen *anzahlEinsen*, *anzahlZweien*, *anzahlDreien*, *anzahlVieren* oder *anzahlFuenfen* um eins erhöht. Alle diese Variablen wurden mit 0 initialisiert. Wenn die erste leere Zelle in der Notenspalte gefunden wurde, bricht die Schleife ab. Das Ergebnis wird in das dritte Arbeitsblatt der Arbeitsmappe geschrieben.

8.2 Die For Next-Schleife

Die *For Next*-Schleife behandeln wir sehr kurz. Sie wurde bereits in Beispiel 2.6 eingehend besprochen.
In den bisher behandelten Beispielen wussten wir im Vorhinein nicht, wie oft die Schleife durchlaufen werden soll. In einigen Fällen ist aber bekannt, wie häufig eine Schleife laufen soll. Solche Fälle sind zwar auch mit der *do while*-Schleife abbildbar, alle Programmiersprachen besitzen 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 derWert der Variablen *zaehler* kleiner oder gleich dem Wert der Variablen *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* kleiner oder gleich dem Wert der Variablen *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 Wert der Variablen *Endwert* ist. Diese Vorgänge finden so lange statt, bis *zaehler* größer als *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[2]:

Beispiel 8.4 *Fakultäten berechnen*

```
'Dateiname: fakultaet.xls
function berechneFakultaet(bisZu as Long) As Long
    dim fakultaet As Long
    dim i As Long
    fakultaet = 1
    for i = 1 to bisZu
        fakultaet = fakultaet * i
    next i
    berechneFakultaet = fakultaet
End function
```

Der Start der Funktion ist altbekannt. Die Funktion bekommt einen Namen (*fakultaetBerechnen*); sie erwartet einen Übergabeparameter (Zellenbezug) aus der Tabellenkalkulation. Wir nehmen an, dass irgendein Benutzer 3 in diese Zelle eingetragen hat. Der Rückgabewert ist vom Typ *Long*, sie wird also einen *Long* in die Zelle, in die sie eingebunden ist, schreiben. Dann folgen Variablendeklarationen.
Vor der Schleife wird unsere Ergebnisvariable mit 1 initialisiert:

```
fakultaet=1
```

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

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

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 3 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:

```
fakultaet = fakultaet * i
```

Da die Variablen *fakultaet* und *i* mit dem Wert 1 initialisiert wurden, ergibt

```
fakultaet = fakultaet * i
```

den Wert 1. Dieser neue Wert wird *fakultaet* zugewiesen. *fakultaet* 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 bisZu
```

[2] 3 Fakultät schreibt man in der Mathematik 3! und es gilt: 3!=1*2*3.

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* 3, ist dies der Fall. Die Anweisung in der Schleife wird durchgeführt. Die Anweisung der Schleife war:

```
fakultaet = fakultaet * i
```

Da *fakultaet* nach dem letzten Schleifendurchlauf den Wert 1 erhielt und *i* zur Zeit den Wert 2 hat, ergibt *fakultaet* * *i* (1 *2) nun 2. Dieser neue Wert wird *fakultaet* 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 *bisZu* ist. Dies ist der Fall (*i* ist 3, eingegebene Zahl immer noch 3), also wird wieder die Anweisung in der Schleife durchgeführt. *fakultaet* war nach dem letzten Schleifendurchlauf 2, *i* ist zur Zeit 3, also ergibt *fakultaet* * *i* (2 *3) den Wert 6. Dieser neue Wert wird *fakultaet* zugewiesen. Die Anweisung

```
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 der eingegebene Zahl ist. Dies ist diesmal nicht der Fall (*i* ist 4, bisZu immer noch 3). Dies veranlasst das Programm nun, die Schleife zu beenden. Eine Schleife zu beenden bedeutet, mit der ersten Anweisung hinter der Schleife fortzufahren. Dies ist:

```
berechneFakultaet = fakultaet
```

Kapitel 9

Interne Nutzung eigener Funktionen

Bei der Arbeit mit Prozeduren und Funktionen haben Sie bisher zwei Dinge gelernt:

- Wie Sie von Ihnen geschriebene Funktionen und Prozeduren aus der Tabellenkalkulation heraus nutzen können (benutzerdefinierte Funktionen, Ereignisprozeduren).
- Wie Sie VBA-Funktionen in den von Ihnen geschriebenen Prozeduren nutzen können (vgl. Kap. 5.2.3).

Noch nicht besprochen haben wir, wie man eigene Funktionen oder Prozeduren in eigenen Funktionen oder Prozeduren benutzt. Das ist aber häufig auch sehr sinnvoll und auch nicht besonders schwierig. Das geht nämlich genauso, wie die in Kap. 5.2.3 beschriebene Nutzung der VBA-internen Funktionen.
Zunächst wollen wir uns an zwei Beispielen die Sinnhaftigkeit einer solchen Vorgehensweise veranschaulichen:

- In unseren beiden Beispielen (Provisionsberechnung bzw. Notenbestimmung) mussten wir die letzte beschriebene Spalte einer Zeile bzw. die letzte beschriebene Zeile einer Spalte ermitteln, im Notenbeispiel sogar mehrfach. Der Realisierungscode steht also mehrfach in unseren Programmen. Dadurch werden die Programme länger. Haben wir einen Fehler in diesem Code, dann haben wir den Fehler mit hoher Wahrscheinlichkeit mehrfach in unseren Programmen, denn wir tippen den Realisierungscode ja nicht mehrfach ein, sondern kopieren. Und hier liegt noch eine weitere Fehlerquelle: Denn sind in der Funktion, in die wir kopieren, die Variablennamen anders[1], dann müssen sie nach dem Kopieren angepasst werden. Dabei können auch Fehler passieren. Diesen Code in eine Funktion auszulagern und diese Funktion dann aufzurufen, ist da hilfreich. Unsere Programme werden kürzer, weil anstelle des Realisierungscodes nur noch ein Funktionsaufruf in den Programmen steht. Fehler können an genau einer Stelle behoben werden, Variablennamen-Anpassungen fallen weg.
- Die Bestimmung der Notengrenzen erfolgt im Notenbeispiel zweimal. Einmal in einer benutzerdefinierten Funktion, wenn die Noten vergeben werden und einmal in einer Ereignisprozedur, wenn die Notengrenzen in ihre Tabelle zum Ausdrucken für die Studenten geschrieben werden. Wir könnten ja hingehen und unser Programm Kollegen zur Verfügung stellen; es ist ja schon eine ziemliche Vereinfachung der Notengebung. Nehmen wir weiter an, ein solcher Kollege benutzt eine andere Methodik zur Notenvergabe. Dann ist das Programm ja zu großen Teilen weiter nutzbar, nur die Bestimmung der Notengrenzen muss nach der anderen Methodik erfolgen. Und weil wir ja von Natur aus nett ☺ sind, machen wir das für ihn. Da wir aber auch fehlbar sind und unsere eigenen Programme manchmal nicht mehr im Griff haben ☺, passen wir nur die benutzerdefinierte Funktion zur Notenvergabe an, die Ereignisprozedur aber nicht. Damit wäre schon eine gewisse Verwirrung erzeugbar. Besser wäre natürlich, die Bestimmung der Notengrenzen fände in einer Prozedur statt, die immer aufgerufen wird, wenn eine Funktion die Notengrenzen benötigt. Dann kann die geschilderte Problematik gar nicht auftreten, denn wir müssten ja nur die Prozedur ändern, und alle Programme, die Notengrenzen benötigen, erhalten dann zwangsläufig die gleichen Werte.
- Nützliche Prozeduren oder Funktionen können wir sammeln und auch Anderen zur Verfügung stellen. Vielleicht gibt es ja Menschen, die selber nicht auf die Idee kommen, wie man die letzte beschriebene Spalte einer Zeile oder Ähnliches ermitteln kann. Diese könnten von uns geschriebene Funktionen und Prozeduren ja auch nutzen. Umgekehrt geht das natürlich auch. So gibt es im Internet zahllose von anderen geschriebene Funktionen, die man selbst nutzen kann.

[1] Bei Notenvergabe und Provisionsberechnung ist es schon wahrscheinlich, dass Variable anders heißen.

9.1 Funktion zur Berechnung der Umsatzsteuer

Beginnen wollen wir allerdings mit einem einfachen Beispiel. Betrachten wir die Funktion zur Berechnung der Umsatzsteuer in Beispiel 9.1. Wir können hier den Umsatzsteuersatz zentral in einer Funktion vorhalten und haben ihn daher nicht in vielen verschiedenen Programmen verteilt. Dies ist ein typisches Beispiel für die sinnvolle Nutzung einer Funktion.

Beispiel 9.1 *Berechnung der Umsatzsteuer aus dem Nettobetrag*

```
'Dateiname: umsatzSteuer.xls
function berechneUmsatzsteuer(nettobetrag As Double) as Double
    const UMSATZSTEUERSATZ as Double = 0.19
    dim umsatzsteuer as Double
    umsatzsteuer = nettobetrag  * UMSATZSTEUERSATZ
    berechneUmsatzsteuer = umsatzsteuer
end function
```

Diese Funktion können wir, wie gewohnt, direkt in Tabellen der Tabellenkalkulationsprogramme nutzen. Andererseits ist es aber auch möglich, diese Funktion, wie auch jede andere von uns geschriebene Funktion, aus von uns geschriebenen Funktionen zu nutzen. Dies schauen wir uns sofort an einem Beispiel an:
Wir wollen aus einem Nettobetrag einen Bruttobetrag ausrechnen. Und wir wissen ja:

```
  Bruttobetrag=Nettobetrag+Umsatzsteuer
```

Kommen wir nun zur Berechnung und Darstellung der Umsatzsteuer. Wir müssen die Funktion *berechneGesamtUnd-Durchschnittsprovision* aus Beispiel 8.1 ändern.

9.2 Umsatzsteuerberechnung für das Provisionsbeispiel in einer Funktion

Bei der Darstellung der Gesamtprovision und der durchschnittlichen Provision soll neben den Provisionen auch noch die Umsatzsteuer und der Bruttobetrag ausgewiesen werden.

Beispiel 9.2 *Integration der Umsatzsteuer*

```
'Dateiname: provisionFunktion.xls
Sub berechneGesamtUndDurchschnittsprovision_Click()
    Const VERKAUFSBETRAGSPALTE As Long = 2
    Const PROVISIONSSPALTE As Long = 3
    Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Long = 4
    Const TABELLENBLATT As Long = 1
    Const ERGEBNIS_TABELLENBLATT As Long = 2
    Dim gesamtVerkaufsbetrag As Double
    Dim zeilenzaehler As Long
    Dim anzahlVerkaeufe As Long
    Dim durchschnittsprovision As Double
    Dim gesamtprovision As Double
    Dim bruttoGesamtprovision As Double
    Dim umsatzsteuerGesamtprovision As Double
    Dim umsatzsteuerDurchschnittsprovision As Double
    Dim bruttoDurchschnittsprovision As Double

    anzahlVerkaeufe = 0
    gesamtprovision = 0

    zeilenzaehler = ERSTE_ZEILE_MIT_VERKAUFSBETRAG
    Do While Not IsEmpty(Sheets(TABELLENBLATT).Cells(zeilenzaehler, PROVISIONSSPALTE))
        gesamtprovision = gesamtprovision + Sheets(TABELLENBLATT).Cells(zeilenzaehler, ➥
            PROVISIONSSPALTE)
        gesamtVerkaufsbetrag = gesamtVerkaufsbetrag + Sheets(TABELLENBLATT).Cells(zeilenzaehler, ➥
            VERKAUFSBETRAGSPALTE)
        zeilenzaehler = zeilenzaehler + 1
    Loop
```

```
    anzahlVerkaeufe = zeilenzaehler - ERSTE_ZEILE_MIT_VERKAUFSBETRAG
    If anzahlVerkaeufe = 0 Then
        MsgBox ("Noch kein Verkauf")
        Exit Sub
    End If

    durchschnittsprovision = gesamtprovision / anzahlVerkaeufe
    umsatzsteuerGesamtprovision = berechneUmsatzsteuer(gesamtprovision)
    bruttoGesamtprovision = gesamtprovision + umsatzsteuerGesamtprovision
    umsatzsteuerDurchschnittsprovision = berechneUmsatzsteuer(durchschnittsprovision)
    bruttoDurchschnittsprovision = durchschnittsprovision + umsatzsteuerDurchschnittsprovision

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 1) = "Anzahl Verkäufe"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 2) = "Gesamtverkäufe"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 3) = "Gesamtprovision"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 4) = "Umsatzsteuer"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 5) = "Bruttoprovision"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 6) = "Durchschnittliche Provision"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 7) = "Umsatzsteuer"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 8) = "Brutto durchschnittliche Provision"

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 1) = anzahlVerkaeufe
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 2) = gesamtVerkaufsbetrag
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 3) = gesamtprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 4) = umsatzsteuerGesamtprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 5) = bruttoGesamtprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 6) = durchschnittsprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 7) = umsatzsteuerDurchschnittsprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 8) = bruttoDurchschnittsprovision
End Sub
```

Neu sind hier nur die Zeilen:

```
umsatzSteuerGesamtprovision = berechneUmsatzsteuer(gesamtprovision)
umsatzSteuerDurchschnittsprovision = berechneUmsatzsteuer(durchschnittsprovision)
```

Und hier sehen wir, wie wir eigene Funktionen aufrufen: Genauso, wie wir VBA-interne Funktionen aufrufen (vgl. Kap. 5.2.3). Eigene Funktionen in eigenen Funktionen zu verwenden ist also nichts Neues. Eigene Funktionen kann man auch mehrfach aufrufen, aber das ist auch nichts wesentlich Neues, auch das kannten wir schon von den VBA internen Funktionen. Während wir beim Aufruf einer Funktion aus einem Tabellenblatt Zellbezüge übergeben, übergeben wir jetzt unsere Variablen. Das der Name der Variablen beim Aufruf der Funktion *gesamtprovision*, bzw. *durchschnittsprovision* nicht mit dem Namen der Variablen bei der Deklaration der Funktion *bruttobetrag* übereinstimmt, ist auch nicht wirklich überraschend. Wenn wir VBA-interne Funktionen aufrufen, dann wissen wir auch nicht, wie die Microsoft-Entwickler die Variablen in den von Ihnen geschriebenen Funktionen nennen. Wir rufen die einfach auf und alles ist gut. Und auch, wenn wir unsere eigenen Funktionen aus der Tabellenkalkulation heraus aufrufen, dann tragen wir Zellbezüge als Übergabeparameter ein und das funktioniert auch.
Im Ausgabe-Teil der Funktion, wenn die Ergebnisse in das zweite Tabellenblatt geschrieben werden, fügen wir einfach die berechneten Umsatzsteuersätze an den richtigen Stellen ein.

9.3 Die Funktion zur Bestimmung der letzten belegten Zeile einer Spalte und ihr Einbau in das Provisionsbeispiel

Bevor wir mit der Programmierung beginnen, machen wir uns klar, welche Problemstellung wir in dieser Funktion realisieren wollen. Wir möchten die letzte belegte Zeile einer Spalte wissen. Zum einen bedeutet dies, dass unsere Funktion wissen muss, in welcher Spalte sie suchen soll. Zum zweiten kann es auch passieren, dass die Funktion nicht in der ersten Zeile der Spalte anfangen soll zu suchen, weil das Layout der Tabelle vielleicht einige leere Zeilen am Anfang einer jeden Spalte vorsieht. Der Funktion muss also mitgeteilt werden, ab welcher Zeile sie in der Spalte suchen soll. Einer Funktion etwas mitteilen, oder eine Funktion muss wissen, sind in der Programmierung andere Worte für Übergabeparameter. Die Übergabeparameter der Funktion sind also: Die Spalte, in der gesucht wird und die Zeile, ab der gesucht wird. Um die

Funktion allgemein zu halten, übergeben wir noch die Nummer des Tabellenblatts, in dem gesucht werden soll. Dann aber ist die Implementierung mit unseren Kenntnissen aus Kap. 8 sehr einfach:

Beispiel 9.3 *Ermittlung der letzten belegten Zeile einer Spalte*

```
'Dateiname: provisionFunktion2.xls
Function ermittleLetzteBesetzteZeileInSpalte(spaltennummer As Long, _
            startzeile As Long, tabellennummer as Long) As Long
    Dim zeilenzaehler As Long
    zeilenzaehler = startzeile
    Do While Not IsEmpty(Sheets(tabellennummer).Cells(zeilenzaehler, spaltennummer))
        zeilenzaehler = zeilenzaehler + 1
    Loop
    ermittleLetzteBesetzteZeileInSpalte = zeilenzaehler - 1
End Function
```

Durch das zusätzliche *Sheets(tabellennummer)* in der Zeile

```
Do While Not IsEmpty(Sheets(tabellennummer).Cells(zeilenzaehler, spaltennummer))
```

wird das Tabellenblatt festgelegt, in dem die letzte besetzte Zeile gefunden werden soll.
Bauen wir diese Funktion nun in unsere Berechnungsfunktion ein. Wir besprechen nur den Berechnungsteil, die Definition der Variablen und Konstanten, sowie die Ausgabe in das Tabellenblatt der Arbeitsmappe entspricht Beispiel 9.2.

Beispiel 9.4 *Provisionsbeispiel mit Funktion zur Ermittlung der letzten besetzten Zeile*

```
'Dateiname: provisionFunktion2.xls
Sub berechneGesamtUndDurchschnittsprovision_Click()
    Const VERKAUFSBETRAGSPALTE As Long = 2
    Const PROVISIONSSPALTE As Long = 3
    Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Long = 4
    Const TABELLENBLATT As Long = 1
    Const ERGEBNIS_TABELLENBLATT As Long = 2

    Dim gesamtprovision As Double
    Dim gesamtVerkaufsbetrag As Double
    Dim anzahlVerkaeufe As Long
    Dim durchschnittsprovision As Double
    Dim nettoGesamtprovision As Double
    Dim bruttoGesamtprovision As Double
    Dim umsatzsteuerGesamtprovision As Double
    Dim umsatzsteuerDurchschnittsprovision As Double
    Dim nettoDurchschnittsprovision As Double
    Dim bruttoDurchschnittsprovision As Double
    Dim letzteBesetzteZeile As Long
    Dim i As Long

    gesamtprovision = 0
    gesamtVerkaufsbetrag = 0

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(PROVISIONSSPALTE, _
                    ERSTE_ZEILE_MIT_VERKAUFSBETRAG, TABELLENBLATT)
    anzahlVerkaeufe = letzteBesetzteZeile - ERSTE_ZEILE_MIT_VERKAUFSBETRAG + 1
    If anzahlVerkaeufe = 0 Then
        MsgBox ("Noch kein Verkauf")
    End If
    For i = ERSTE_ZEILE_MIT_VERKAUFSBETRAG To letzteBesetzteZeile
        gesamtVerkaufsbetrag = gesamtVerkaufsbetrag + sheets(TABELLENBLATT).Cells(i, ➥
            VERKAUFSBETRAGSPALTE)
        gesamtprovision = gesamtprovision + sheets(TABELLENBLATT).Cells(i, PROVISIONSSPALTE)
    Next i

    durchschnittsprovision = gesamtprovision / anzahlVerkaeufe
    umsatzsteuerGesamtprovision = berechneUmsatzsteuer(gesamtprovision)
    nettoGesamtprovision = gesamtprovision - umsatzsteuerGesamtprovision
```

```
    umsatzsteuerDurchschnittsprovision = berechneUmsatzsteuer(durchschnittsprovision)
    nettoDurchschnittsprovision = durchschnittsprovision - umsatzsteuerDurchschnittsprovision

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 1) = "Anzahl Verkäufe"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 2) = "Gesamtverkäufe"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 3) = "Gesamtprovision"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 4) = "Umsatzsteuer"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 5) = "Bruttoprovision"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 6) = "Durchschnittliche Provision"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 7) = "Umsatzsteuer"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 8) = "Brutto durchschnittliche Provision"

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 1) = anzahlVerkaeufe
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 2) = gesamtVerkaufsbetrag
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 3) = gesamtprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 4) = umsatzsteuerGesamtprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 5) = bruttoGesamtprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 6) = durchschnittsprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 7) = umsatzsteuerDurchschnittsprovision
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 8) = bruttoDurchschnittsprovision
End Sub
```

Ganz so, wie in Beispiel 9.2 die Funktion zur Berechnung der Umsatzsteuer aufgerufen wurde, rufen wir hier unsere neue Funktion zur Bestimmung der letzten besetzten Zeile auf:

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(PROVISIONSSPALTE, ➥
    ERSTE_ZEILE_MIT_VERKAUFSBETRAG, TABELLENBLATT)
```

Nun benötigen wir natürlich im Hauptprogramm keine *while*-Schleife mehr. Wir wissen ja nun die letzte besetzte Zeile und können daher die Aufsummierungen in einer *for*-Schleife durchführen.

```
For i = ERSTE_ZEILE_MIT_VERKAUFSBETRAG To letzteBesetzteZeile
```

lässt die Schleife mit der Schleifenvariable *i* von der ersten Zeile mit Verkaufsbeträgen bis zur letzten besetzten Zeile, also über alle Daten der Tabelle, laufen. Weil die For-Schleife den Startwert von *i* automatisch setzt, benötigen wir nicht mehr die Zuweisung

```
zeilenzaehler = ERSTE_ZEILE_MIT_VERKAUFSBETRAG
```

Diese Zeile fällt bei der neuen Lösung also vollständig weg. In der Schleife wird genauso wie bisher die Gesamtprovision und der Gesamtverkaufsbetrag berechnet. Die beide Zeilen

```
gesamtprovision = gesamtprovision + Sheets(TABELLENBLATT).Cells(zeilenzaehler, PROVISIONSSPALTE)
gesamtVerkaufsbetrag = gesamtVerkaufsbetrag + Sheets(TABELLENBLATT).Cells(zeilenzaehler, ➥
    VERKAUFSBETRAGSPALTE)
```

ändern sich also im Vergleich zu Beispiel 9.2 nicht. Zuletzt kommt noch das Schleifenende

```
Next i
```

Die bisherige dritte Zeile in der Schleife aus Beispiel 9.2

```
zeilenzaehler = zeilenzaehler + 1
```

fällt wieder weg. For-Schleifen zählen die Schleifenvariable automatisch hoch, deswegen müssen wir uns darum nicht kümmern. Dies hat neben dem Vorteil, dass wir eine Zeile weniger zu tippen haben, auch den riesigen Vorteil, dass wir keine Endlosschleifen mehr programmieren können.

9.4 Anmerkung

Wir betonen, dass wir hier durch die Einführung einer Funktion zur Bestimmung der letzten besetzten Zeile keine Verkürzung des Programms bekommen. Das Programm läuft auch nicht schneller. Wir haben die Qualität unseres Programms

verbessert. Bislang waren die Bestimmung der letzten besetzten Zeile und die Aufsummierungen im Programmcode vermischt. Dies haben wir nun getrennt. Zum Zweiten gibt es nun eine weitere Funktion, die wir unserer Funktionsbibliothek hinzufügen und immer wieder in neuen Projekten nutzen können[2]. Außerdem werden Programme durch die Verwendung von For-Schleifen meist besser lesbar, und robuster gegenüber der unabsichtlichen Programmierung von Endlosschleifen.

9.5 Anpassung des Notenbeispiels

Sinnvolle Anpassungen des Notenbeispiels sind nur:

- Nutzung der Funktion *ermittleLetzteBesetzteZeileInSpalte*, sowie Umstellung der *while*-Schleife auf eine *for*-Schleife.
- Schreiben der Funktion *ermittleLetzteBesetzteSpalteInZeile*, um die Spalte zu ermitteln, in der die Klausurergebnisse stehen.

Das Schreiben der Funktion *ermittleLetzteBesetzteSpalteInZeile* ist komplett analog zu *ermittleLetzteBesetzteZeileInSpalte*, nur dass wir halt über die Spalten einer Zeile iterieren, um die letzte besetzte Spalte zu finden. Der Aufruf beider Funktionen ist ebenfalls analog zum Aufruf der Funktion *ermittleLetzteBesetzteZeileInSpalte* im Provisionsbeispiel. Daher stellen wir im folgenden die Implementierung der Funktion *ermittleLetzteBesetzteSpalteInZeile*, die Aufrufe der beiden Funktionen sowie die Änderung von der *while*-zur *for*-Schleife unkommentiert dar.
Zunächst die Funktion *ermittleLetzteBesetzteSpalteInZeile*:

Beispiel 9.5 *Ermittlung der letzten belegten Spalte einer Zeile*

```
Function ermittleLetzteBesetzteSpalteInZeile(zeilennnummer As Long, _
                    startspalte As Long, tabellennummer As Long) As Long
    Dim spaltenzaehler As Long
    spaltenzaehler = startspalte

    Do While Not IsEmpty(Sheets(tabellennummer).Cells(zeilennnummer, spaltenzaehler))
        spaltenzaehler = spaltenzaehler + 1
    Loop

    ermittleLetzteBesetzteSpalteInZeile = spaltenzaehler - 1
End Function
```

Zunächst zeigen wir die Einbindung der beiden neuen Funktionen sowie der *for*-Schleife in das Beispiel 8.2 zur Berechnung des Notendurchschnitts. Hier zunächst der vollständige Programmcode:

Beispiel 9.6 *Änderungen in der Ereignisprozedur berechneDurchschnittsnote*

```
'Dateiname: noteFunktion.xls
Sub berechneDurchschnittsnote_Click()
    Const OPTIMALZEILE As Long = 4
    Const NAME_SPALTE As Long = 1
    Const TABELLENBLATT As Long = 1
    Const ERGEBNIS_TABELLENBLATT As Long = 3
    Dim letzteBesetzteSpalte As Long
    Dim letzteBesetzteZeile As Long
    Dim anzahlTeilnehmer As Long
    Dim notenSumme As Double
    Dim durchschnittsnote As Double
    Dim i As Long

    anzahlTeilnehmer = 0
    notenSumme = 0
    letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(OPTIMALZEILE, NAME_SPALTE + 2, ➥
        TABELLENBLATT)
```

[2] In Wirklichkeit hätten wir die Funktion zur Ermittlung der letzten besetzten Zeile gar nicht selber schreiben müssen, denn für Excel existieren im Internet Sammlungen von freien Makros, darunter selbstverständlich auch so grundlegende, wie das hier vorgestellte. Googeln Sie einfach „Makro Excel“ und Sie erhalten seitenweise Trefferlisten. Bei solch einem einfachen Makro, wie dem hier vorgestellten, ist es aber sicherlich schneller, das Makro eben selber neu zu schreiben.

```
    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(NAME_SPALTE, OPTIMALZEILE + 1, ➥
        TABELLENBLATT)

    anzahlTeilnehmer = letzteBesetzteZeile - OPTIMALZEILE
    For i = OPTIMALZEILE + 1 To letzteBesetzteZeile
        notenSumme = notenSumme + Sheets(TABELLENBLATT).Cells(i, letzteBesetzteSpalte)
    Next
    durchschnittsnote = notenSumme / anzahlTeilnehmer

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 1) = "Durchschnittsnote"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 2) = durchschnittsnote
End Sub
```

Zunächst berechnen wir die Spalte, in der die Noten stehen, mit Hilfe der Funktion *ermittleLetzteBesetzteSpalteInZeile* und die letzte Zeile mit Noten durch die Funktion *ermittleLetzteBesetzteZeileInSpalte*.

```
letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(OPTIMALZEILE, NAME_SPALTE + 2, ➥
    TABELLENBLATT)
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(NAME_SPALTE, OPTIMALZEILE + 1, ➥
    TABELLENBLATT)
```

Da wir die Notenspalte jetzt durch einen einzigen Funktionsaufruf bestimmen können, fällt die erste *do while*-Schleife inklusive der dort notwendigen Initialisierung der Schleifenvariable aus Beispiel 8.2 komplett weg. Die Anzahl der Teilnehmer an der Prüfung können wir dann wie folgt berechnen:

```
anzahlTeilnehmer = letzteBesetzteZeile - OPTIMALZEILE
```

Und wie in Beispiel 9.4 ersetzen wir die *do while*-Schleife durch eine *for*-Schleife, wodurch das bisher notwendige initialisieren der Schleifenvariable und deren Hochzählen wegfällt.

```
For i = OPTIMALZEILE + 1 To letzteBesetzteZeile
    notenSumme = notenSumme + Sheets(TABELLENBLATT).Cells(i, letzteBesetzteSpalte)
Next i
```

In exakt der gleichen Weise können die die *do while*-Schleifen im Beispiel 8.3 zu Berechnung der Notenverteilung ersetzt werden. Da hier wirklich nichts Neues dazu kommen, überlassen wir dies Ihnen als Übungsaufgabe.

Kapitel 10

Datenauswertung

Ein weiterer Einsatzschwerpunkt einer Tabellenkalkulation ist die Auswertung von Daten. Wie wir bereits mehrfach gesehen haben, entspricht im Normalfall jede Zeile eines Arbeitsblattes einer Tabellenkalkulation einem Datensatz. In Beispiel 9.4 entspricht jede Zeile einem Verkaufsvorgang. Dort haben wir auch bereits erste Auswertungen durchgeführt, indem wir Gesamtumsatz, Gesamtprovision und durchschnittliche Provison berechnet haben. Je nach Daten sind aber weitergehende Auswertung möglich, wie

- Umsatz verteilt auf Regionen
- umsatzstärkste Region (Maximum)
- umsatzschwächste Region (Minimum).

Solche Aufgabenstellungen lassen sich aber leicht mit dem bereits zur Verfügung stehenden Instrumentarium lösen. Wir schauen uns die grundsätzliche Vorgehensweise an drei Beispielen an:

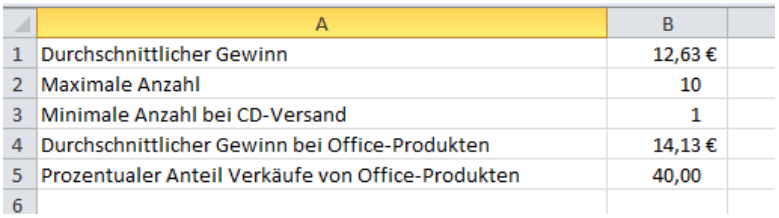
10.1 Das Gewinnbeispiel

Wir betrachten das in Abb. 10.1 dargestellte Tabellenblatt. Daraus wollen wir die in in Abb. 10.2 dargestellten Statistiken erzeugen.

	A	B	C	D	E	F	G
1	Kunde	Produkt	Anzahl	Einkaufspreis	Spoftwarekategorie	Versandart	Gewinn
2	Schmid	PDF Reader	1	26,00 €	Utilities	Download	2,08 €
3	Meier	Kalkulation	5	82,95 €	Office	CD-Versand	23,04 €
4	Seran	Windows 4711	2	112,00 €	Betriebssysteme	Download	6,72 €
5	Müller	Windows 0815	10	87,00 €	Betriebssysteme	Download	26,10 €
6	Schmidt	Writer	1	58,45 €	Office	CD-Versand	5,22 €
7							

Abbildung 10.1
Daten des ersten Beispiels

Die Schaltfläche zum Start des Statistikprogramms soll grundsätzlich in jedes Tabellenblatt einbaubar sein. Die Auswertung soll in Tabellenblatt 2 geschrieben werden. Sie in Tabellenblatt 1 zu schreiben, ist nicht so sinnvoll, da wir nicht wissen, wieviel Zeilen in diesem Tabellenblatt gefüllt sein werden.
Die grundsätzliche Vorgehensweise ist analog zu der in Kapitel 9.3. Wir bestimmen zunächst die letzte besetzte Zeile. Dafür haben wir ja bereits eine Funktion. Dann gehen wir in einer Schleife über alle Datensätze. Die Gesamtanzahl Datensätze können wir dann wie in Kapitel 9.3 oder durch mitzählen in der Schleife bestimmen. Um den durchschnittlichen Gewinn bestimmen zu können, müssen wir den Gesamtgewinn ausrechnen, dies erfolgt völlig analog der Berechnung der Gesamtprovision in Kapitel 9.3. Der durchschnittliche Gewinn ist dann der Gesamtgewinn dividiert durch die Gesamtanzahl Datensätze. Wie man ein Maximum und Minimum bestimmt, erklären wir später. Daher kommen wir nun zur



	A	B
1	Durchschnittlicher Gewinn	12,63 €
2	Maximale Anzahl	10
3	Minimale Anzahl bei CD-Versand	1
4	Durchschnittlicher Gewinn bei Office-Produkten	14,13 €
5	Prozentualer Anteil Verkäufe von Office-Produkten	40,00
6		

Abbildung 10.2
Auswertung des ersten Beispiels

Berechnung des Gesamtgewinns bei Office-Produkten. Dies funktioniert aber grundsätzlich genauso, wie die Berechnung des Gesamtgewinns. Allerdings darf die Aufsummierung hier nur dann stattfinden, wenn in der Spalte E Office steht. Dies können wir aber leicht durch ein *if*-Konstrukt lösen. Zur Berechnung des durchschnittlichen Gewinns bei Office-Produkten müssen wir zählen, wie viele Datensätze mit Office-Produkten wir haben und dann den Gesamtgewinn bei Office-Produkten durch diese Anzahl teilen. Der prozentale Anteil Verkäufe von Office-Produkten ist abschließend ja nur die Anzahl der Datensätze mit Office-Produkten dividiert durch die Gesamtanzahl der Datensätze multipliziert mit 100. Schauen wir uns nun die Lösung an:

Beispiel 10.1 *Das Programm zur Datenauswertung*

```
Sub statistiken_click()
    Const ANZAHL_SPALTE As Long = 3
    Const KATEGORIE_SPALTE As Long = 5
    Const VERSANDART_SPALTE As Long = 6
    Const GEWINN_SPALTE As Long = 7
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Const TABELLENBLATT As Long = 1
    Const TABELLENBLATT_STATISTIK As Long = 2

    Dim anzahlGesamt As Long
    Dim gewinnGesamt As Double
    Dim durchschnittlicherGewinn As Double
    Dim maxAnzahl As Long
    Dim minAnzahlCDVersand As Long
    Dim gewinnOfficeProdukte As Double
    Dim anzahlOfficeProdukte As Long
    Dim durchschnittlicherGewinnOffice As Double
    Dim prozentualerAnteilOffice As Double
    Dim gewinn As Double
    Dim anzahl As Long

    Dim i As Long
    Dim letzteBesetzteZeile As Long

    anzahlGesamt = 0
    gewinnGesamt = 0
    maxAnzahl = 0
    minAnzahlCDVersand = 1000
    gewinnOfficeProdukte = 0
    anzahlOfficeProdukte = 0

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(GEWINN_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        anzahlGesamt = anzahlGesamt + 1

        gewinn = Sheets(TABELLENBLATT).Cells(i, GEWINN_SPALTE)
        gewinnGesamt = gewinnGesamt + gewinn

        anzahl = Sheets(TABELLENBLATT).Cells(i, ANZAHL_SPALTE)
```

```
        maxAnzahl = bestimmeMaximumLong(maxAnzahl, anzahl)

        If Sheets(TABELLENBLATT).Cells(i, VERSANDART_SPALTE) = "CD-Versand" Then
            minAnzahlCDVersand = bestimmeMinimumLong(minAnzahlCDVersand, anzahl)
        End If

        If Sheets(TABELLENBLATT).Cells(i, KATEGORIE_SPALTE) = "Office" Then
            anzahlOfficeProdukte = anzahlOfficeProdukte + 1
            gewinnOfficeProdukte = gewinnOfficeProdukte + gewinn
        End If
    Next i

    durchschnittlicherGewinn = gewinnGesamt / anzahlGesamt
    prozentualerAnteilOffice = anzahlOfficeProdukte / anzahlGesamt * 100
    durchschnittlicherGewinnOffice = dividiereMitErgebnis0FuerNenner0Double(gewinnOfficeProdukte,➥
         anzahlOfficeProdukte)

    Sheets(TABELLENBLATT_STATISTIK).Cells(1, 1) = "Durchschnittlicher Gewinn"
    Sheets(TABELLENBLATT_STATISTIK).Cells(2, 1) = "Maximale Anzahl"
    Sheets(TABELLENBLATT_STATISTIK).Cells(3, 1) = "Minimale Anzahl bei CD-Versand"
    Sheets(TABELLENBLATT_STATISTIK).Cells(4, 1) = "Durchschnittlicher Gewinn bei Office-Produkten➥
        "
    Sheets(TABELLENBLATT_STATISTIK).Cells(5, 1) = "Prozentualer Anteil Verkäufe von Office-➥
        Produkten"

    Sheets(TABELLENBLATT_STATISTIK).Cells(1, 2) = durchschnittlicherGewinn
    Sheets(TABELLENBLATT_STATISTIK).Cells(2, 2) = maxAnzahl
    Sheets(TABELLENBLATT_STATISTIK).Cells(3, 2) = minAnzahlCDVersand
    Sheets(TABELLENBLATT_STATISTIK).Cells(4, 2) = durchschnittlicherGewinnOffice
    Sheets(TABELLENBLATT_STATISTIK).Cells(5, 2) = prozentualerAnteilOffice
End Sub
```

Gehen wir die Lösung im Einzelnen durch:
Zunächst definieren wir einige Konstanten. Die ersten Konstanten definieren die Spalten, die wir im weiteren Verlauf benötigen:

```
Const ANZAHL_SPALTE As Long = 3
Const KATEGORIE_SPALTE As Long = 5
Const VERSANDART_SPALTE As Long = 6
Const GEWINN_SPALTE As Long = 7
```

Dies ist sinnvoll, weil sich das Layout des Tabellenblatts verändern kann. Nehmen wir an, ein Entscheider im Unternehmen möchte die Spalte mit der Versandart vor der Spalte mit der Softwarekategorie stehen haben. Dann müssen wir nur die Werte der Konstanten *KATEGORIE_SPALTE* und *VERSANDART_SPALTE* anpassen. Aus dem gleichen Grund legen wir die Zeile, in der die Daten beginnen, das Tabellenblatt, in dem die Daten stehen, sowie das Tabellenblatt, in das die Statistiken geschrieben werden sollen, auf Konstanten.

```
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
Const TABELLENBLATT As Long = 1
Const TABELLENBLATT_STATISTIK As Long = 2
```

Dann erfolgt die Deklaration der Variablen, die wir für die Berechnung benötigen:

```
Dim anzahlGesamt As Long
Dim gewinnGesamt As Double
Dim durchschnittlicherGewinn As Double
Dim maxAnzahl As Long
Dim minAnzahlCDVersand As Long
Dim gewinnOfficeProdukte As Double
Dim anzahlOfficeProdukte As Long
Dim durchschnittlicherGewinnOffice As Double
Dim prozentualerAnteilOffice As Double
Dim gewinn As Double
Dim anzahl As Long

Dim i As Long
```

```
Dim letzteBesetzteZeile As Long
```

Es folgen die Initialisierungen.

```
anzahlGesamt = 0
gewinnGesamt = 0
maxAnzahl = 0
minAnzahlCDVersand = 1000
gewinnOfficeProdukte = 0
anzahlOfficeProdukte = 0
```

Warum diese notwendig sind, erläutern wir, wenn wir die Schleife besprechen. Wir bestimmen die letzte besetzte Zeile, also die Zeile, bis zu der unsere Schleife laufen muss.

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(GEWINN_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
```

Zur Bestimmung der letzten besetzten Zeile können wir grundsätzlich jede Spalte des Tabellenblatts bis zur Spalte G nehmen, weil alle Spalten durchgängig gefüllt sind. Die Zeile, in der wir die Bestimmung der letzten besetzten Zeile starten, ist natürlich die Zeile 2, weil hier die Daten beginnen. Diese hatten wir ja auf der Konstanten *ERSTE_ZEILE_MIT_INFORMATIONEN* abgelegt. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteZeileInSpalte* hatten wir in Kapitel 9.3 eingehend besprochen, so dass hier auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun startet unsere Schleife. Sie läuft natürlich von Zeile 2 (*ERSTE_ZEILE_MIT_INFORMATIONEN*) zur letzten besetzten Zeile, da ja jeder Datensatz behandelt werden muss.

```
For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
    anzahlGesamt = anzahlGesamt + 1
```

Die Zeile

```
    anzahlGesamt = anzahlGesamt + 1
```

zählt die Anzahl der Datensätze im Tabellenblatt. Die Variable *anzahlGesamt* wird bei jedem Durchlauf der Schleife durch obige Anweisung um 1 erhöht. Hier werden Sie jetzt auch die Problematik der Initialisierungen verstehen. Denn, wenn die Variable *anzahlGesamt* nicht initialisiert wird, dann hat diese Variable beim ersten Schleifendurchlauf keinen definierten Wert. Demzufolge hätte natürlich auch *anzahlGesamt + 1* keinen definierten Wert. Andererseits möchten wir, dass *anzahlGesamt* nach dem ersten Schleifendurchlauf den Wert 1 hat (der erste Datensatz wurde bearbeitet). Daher initialisieren wir *anzahlGesamt* mit dem Wert 0:

```
anzahlGesamt = 0
```

Dann hat *anzahlGesamt* beim Start des ersten Schleifendurchlauf den Wert 0.

```
    anzahlGesamt = anzahlGesamt + 1
```

erhöht diesen Wert dann auf 1, was genau das ist, was wir wollen. Bei jedem weiteren Durchlauf der Schleife wird *anzahlGesamt* jeweils um eins hochgezählt, so dass auf dieser Variablen am Ende der Schleife die Anzahl der Datensätze im Arbeitsblatt abgespeichert ist.
Falls Sie sich jetzt fragen, warum wir *anzahlGesamt* überhaupt berechnen, wo dies doch gar nicht gefragt, so ist die Antwort: Es ist sehr wohl gefragt, denn wir sollen den durchschnittlichen Gewinn ausrechnen. Der durchschnittliche Gewinn ist aber der Gesamtgewinn dividiert durch die Anzahl Datensätze. *anzahlGesamt* ist die Anzahl Datensätze.
Im folgenden berechnen wir den Gesamtgewinn: In den Zeilen

```
    gewinn = Sheets(TABELLENBLATT).Cells(i, GEWINN_SPALTE)
    gewinnGesamt = gewinnGesamt + gewinn
```

wird zunächst der Inhalt der Zelle der Spalte mit dem Gewinn in der jeweiligen Zeile in der Variablen *gewinn* gespeichert. Wegen der Anforderung die Schaltfläche zum Programmstart in jedes Zabellenblatt einbauen zu können, ist nicht mehr sichergestellt, dass sich Daten und Schaltfläche im gleichen Tabellenblatt befinden. Wir müssen also unter Angabe des Tabellenblattes auf die Zelle zugreifen. Daher ist die Angabe *Sheets(TABELLENBLATT).Cells(i, GEWINN_SPALTE)* notwendig. *Cells(i, GEWINN_SPALTE)* reicht nicht mehr. Diese Vorgehensweise ist natürlich auch bei allen anderen Zellzugriffen in diesem Programm notwendig.
Danach wird der Wert auf die Variable *gewinnGesamt* addiert, so dass nach dem letzten Schleifendurchlauf auf dieser Variablen die Summe aller Gewinne (der Gesamtgewinn) gespeichert ist. Aus dem gleichen Grund wie bei *anzahlGesamt*, muss die Variable *gewinnGesamt* mit Null initialisiert werden.

```
gewinnGesamt = 0
```

Generell können wir sagen:
Jede Variable, die in einer Anweisung gleichzeitig auf der rechten und der linken Seite des Zuweisungsoperators vorkommt, muss zu diesem Zeitpunkt einen definierten Wert haben. Für unsere Auswertungsprogramme bedeutet dies, dass diese Variablen initialisiert werden müssen. Und weil auf diesen Variablen in unseren Fällen immer Aufsummierungen gespeichert werden, müssen diese Variablen mit 0 initialisiert werden.
Nun folgt die Bestimmung des Maximums. Hierbei gehen wir folgendermaßen vor: Zunächst initialisieren wir die Variable, auf der das Maximum gespeichert wird, mit einer Zahl, die zu klein ist, als das sie als Maximum in Frage kommt.

```
maxAnzahl = 0
```

Hier wird *maxAnzahl* mit 0 initialisiert. Da es sich hier um die Anzahl verkaufter Produkte handelt, ist 0 kleiner als jede erlaubte Anzahl.
In der Schleife nun wird der Wert in der gerade bearbeiteten Zeile mit dem Wert, den das Maximum gerade hat, verglichen. Ist das bisherige Maximum größer als der jetzige Wert, passiert nichts. Ist das bisherige Maximum allerdings kleiner als der jetzige Wert, wird das Maximum auf den gerade behandelten Wert geändert.
In unserem Beispiel bedeutet das:

- Beim ersten Schleifendurchlauf ist der Wert von *maxAnzahl* Null, weil *maxAnzahl* so initialisiert wurde. In Zeile 2 (dies ist ja der Start der Schleife), ist die Anzahl der Produkte Eins (Spalte C). Eins ist größer als Null, die Variable *maxAnzahl* wird also auf Eins gesetzt.
- Beim zweiten Schleifendurchlauf ist der Wert von *maxAnzahl* Eins (vom vorherigen Schleifendurchlauf). In Zeile 3 ist die Anzahl der Produkte Fünf (Spalte C). Fünf ist größer als Eins, die Variable *maxAnzahl* wird also auf Fünf gesetzt.
- Beim dritten Schleifendurchlauf ist der Wert von *maxAnzahl* Fünf (vom vorherigen Schleifendurchlauf). In Zeile 4 ist die Anzahl der Produkte Zwei (Spalte C). Zwei ist nicht größer als Fünf, die Variable *maxAnzahl* behält also ihren Wert Fünf.
- Beim vierten Schleifendurchlauf ist der Wert von *maxAnzahl* Fünf (vom vorherigen Schleifendurchlauf). In Zeile 5 ist die Anzahl der Produkte Zehn (Spalte C). Zehn ist größer als Fünf, die Variable *maxAnzahl* wird also auf Zehn gesetzt
- Beim fünften Schleifendurchlauf ist der Wert von *maxAnzahl* Zehn (vom vorherigen Schleifendurchlauf). In Zeile 6 ist die Anzahl der Produkte Eins (Spalte C). Eins ist nicht größer als Zehn, die Variable *maxAnzahl* behält also ihren Wert Zehn.

Dann ist die Schleife zu Ende, Die Variable *maxAnzahl* hat den Wert 10, also des Maximum.
Wie programmieren wir dies? Die Initialisierung haben wir bereits durchgeführt, die Variable *maxAnzahl* ist bereits mit Null initialisiert. In der Schleife müssen wir nun noch den jetzigen Wert von *maxAnzahl* mit dem Wert von Anzahl der jeweiligen Zeile vergleichen und wenn der Wert in der jeweiligen Zeile größer ist, dann müssen wir den Wert von *maxAnzahl* umsetzen.
Für den Vergleich, ob der jetzige in der Zeile eingelesene Wert größer ist, als das bisherige Maximum, schreiben wir eine Funktion. Dies ist sinnvoll, denn es verkürzt den Code in der Schleife. Dadurch wird das Programm besser lesbar. Darüber hinaus ist die Bestimmung eines Maximums eine Aufgabe, die wir noch häufiger lösen werden müssen. Wir können dann auf die bereits geschriebene Funktion zurückgreifen.

Beispiel 10.2 *Die Funktion zur Maximumbestimmung für Long-Variablen*

```
Function bestimmeMaximumLong(wert1 As Long, wert2 As Long) As Long
    If wert1 >= wert2 Then
        bestimmeMaximumLong = wert1
    Else
        bestimmeMaximumLong = wert2
    End If
End Function
```

Diese Funktion rufen wir in der Schleife wie folgt auf:

```
    anzahl = Sheets(TABELLENBLATT).Cells(i, ANZAHL_SPALTE)
    maxAnzahl = bestimmeMaximumLong(maxAnzahl, anzahl)
```

Die Implementierung der Funktion sollten Sie sofort verstehen. Sie vergleicht die beiden ihr übergebenen Werte und gibt den größeren als Ergebnis zurück. Anders ausgedrückt: Sie bestimmt das Maximum ihrer Übergabeparameter.
Den Namen der Funktion finden Sie vielleicht etwas merkwürdig: *bestimmeMaximumLong*. Wenn Sie sich jedoch noch an die Einführung von Funktionen in Kap. 9 erinnern, so sind ja die Übergabevariablen einer der wesentlichen Punkte der Arbeit mit Funktionen. Die Übergabeparameter der Funktion *bestimmeMaximumLong* sind die beiden Variablen *wert1* und *wert2*. Sie sind beide vom Typ *Long*. Dies bedeutet aber, dass unsere Funktion *bestimmeMaximumLong* immer mit zwei Variablen, die beide vom Typ *Long* sind, aufgerufen werden muss. Dies ist hier auch der Fall: *maxAnzahl* und *anzahl* sind beide vom Typ *Long*. Dies schränkt die Benutzbarkeit der Funktion natürlich ein, denn schon mit *Double*-Variablen in der Übergabe, würde die Funktion nicht laufen. Diese Problemstellung existiert aber auch häufiger, so etwa in Kap. 10.2.
Unsere Funktion *bestimmeMaximumLong* heißt also so, weil sie das Maximum zweier Zahlen vom Typ *Long* bestimmt.
Gehen wir den Ablauf innerhalb der Schleife nun noch einmal anhand des oben dargestellten Programmcodes durch:

- Beim ersten Schleifendurchlauf ist der Wert von *maxAnzahl* Null, weil *maxAnzahl* so initialisiert wurde. In Zeile 2 (dies ist ja der Start der Schleife), ist die Anzahl der Produkte Eins (Spalte C). Der Variablen *anzahl* wird also der Wert Eins zugewiesen. Beim Aufruf der Funktion *bestimmeMaximumLong* wird also *wert1* mit Null (dem Wert von *maxAnzahl*) und *wert2* mit Eins (dem Wert von *anzahl*) besetzt. Die Funktion *bestimmeMaximumLong* gibt also Eins zurück. Die Variable *maxAnzahl* wird also auf Eins gesetzt.

- Beim zweiten Schleifendurchlauf ist der Wert von *maxAnzahl* Eins (vom vorherigen Schleifendurchlauf). In Zeile 3 ist die Anzahl der Produkte Fünf (Spalte C). Der Variablen *anzahl* wird also der Wert Fünf zugewiesen. Beim Aufruf der Funktion *bestimmeMaximumLong* wird also *wert1* mit Eins (dem Wert von *maxAnzahl*) und *wert2* mit Fünf (dem Wert von *anzahl*) besetzt. Die Funktion *bestimmeMaximumLong* gibt also Fünf zurück. Die Variable *maxAnzahl* wird also auf Fünf gesetzt.

- Beim dritten Schleifendurchlauf ist der Wert von *maxAnzahl* Fünf (vom vorherigen Schleifendurchlauf). In Zeile 4 ist die Anzahl der Produkte Zwei (Spalte C). Der Variablen *anzahl* wird also der Wert Zwei zugewiesen. Beim Aufruf der Funktion *bestimmeMaximumLong* wird also *wert1* mit Fünf (dem Wert von *maxAnzahl*) und *wert2* mit Zwei (dem Wert von *anzahl*) besetzt. Die Funktion *bestimmeMaximumLong* gibt also Fünf zurück. Die Variable *maxAnzahl* wird also auf Fünf gesetzt.

- Beim vierten Schleifendurchlauf ist der Wert von *maxAnzahl* Fünf (vom vorherigen Schleifendurchlauf). In Zeile 5 ist die Anzahl der Produkte Zehn (Spalte C). Der Variablen *anzahl* wird also der Wert Zehn zugewiesen. Beim Aufruf der Funktion *bestimmeMaximumLong* wird also *wert1* mit Fünf (dem Wert von *maxAnzahl*) und *wert2* mit Zehn (dem Wert von *anzahl*) besetzt. Die Funktion *bestimmeMaximumLong* gibt also Zehn zurück. Die Variable *maxAnzahl* wird also auf Zehn gesetzt.

- Beim fünften Schleifendurchlauf ist der Wert von *maxAnzahl* Zehn (vom vorherigen Schleifendurchlauf). In Zeile 6 ist die Anzahl der Produkte Eins (Spalte C). Der Variablen *anzahl* wird also der Wert Eins zugewiesen. Beim Aufruf der Funktion *bestimmeMaximumLong* wird also *wert1* mit Zehn (dem Wert von *maxAnzahl*) und *wert2* mit Eins (dem Wert von *anzahl*) besetzt. Die Funktion *bestimmeMaximumLong* gibt also Zehn zurück. Die Variable *maxAnzahl* wird also auf Zehn gesetzt.

Kommen wir nun zum nächsten Punkt: Minimale Anzahl bei CD-Versand. Ein Minimum bestimmen wir im Prinzip genauso wie ein Maximum. Initialisieren müssen wir allerdings mit einem Wert, der zu groß ist, als das er als Minimum in Frage kommt. Dies tun wir hier:

```
minAnzahlCDVersand = 1000
```

Nun wählen wir die gleiche Vorgehensweise wie bei der Maximumbestimmung. Die einzige Änderung ist, dass wir das Minimum dann umsetzen, wenn der Wert in der Zeile, in der wir uns befinden, kleiner ist als das bisherige Minimum. Dafür schreiben wir die Funktion *bestimmeMinimumLong*:

Beispiel 10.3 *Die Funktion zur Minimumbestimmung für Long-Variablen*

```
Function bestimmeMinimumLong(wert1 As Long, wert2 As Long) As Long
  If wert1 <= wert2 Then
    bestimmeMinimumLong = wert1
  Else
    bestimmeMinimumLong = wert2
  End If
End Function
```

Wir müssen eine weitere Einschränkung betrachten: Die Minimumbestimmung soll nur über die Verkäufe bei der Versandart CD-Versand gehen. Dies bedeutet, dass wir in der Schleife nur dann tätig werden müssen, wenn in der Spalte für die Versandart (Spalte F) „CD-Versand“ steht. Dies können wir aber wieder einfach durch ein *if*-Statement regeln. Daher resultiert folgender Programmcode:

```
    If Sheets(TABELLENBLATT).Cells(i, VERSANDART_SPALTE) = "CD-Versand" Then
        minAnzahlCDVersand = bestimmeMinimumLong(minAnzahlCDVersand, anzahl)
    End If
```

Es bleiben noch die Auswertungen für Office-Produkte. Um diese ausrechnen zu können, müssen wir den Gesamtgewinn bei Office-Produkten und die Anzahl Datensätze, die sich auf Office-Produkte beziehen, ermitteln. Der Quotient dieser beiden Werte ist der durchschnittlicher Gewinn bei Office-Produkten, der ja dargestellt werden soll. Das ist aber prinzipiell dieselbe Vorgehensweise, die wir bereits bei der Ermittlung des Gesamtgewinns und der Gesamtanzahl Datensätze angewendet haben. Der einzige Unterschied ist, dass wir in der Schleife nur dann tätig werden, wenn in der Spalte für die Softwarekategorie (Spalte E) „Office“ steht. Wie man so etwas macht, wissen wir aber auch schon. Dazu reicht ein einfaches *if*-Statement:

```
    If Sheets(TABELLENBLATT).Cells(i, KATEGORIE_SPALTE) = "Office" Then
        anzahlOfficeProdukte = anzahlOfficeProdukte + 1
        gewinnOfficeProdukte = gewinnOfficeProdukte + gewinn
    End If
```

Nun beenden wir die Schleife.

```
Next i
```

Nach der Schleife müssen wir die Durchschnitte und prozentualen Anteile berechnen. Der durchschnittliche Gewinn ist ja einfach der Gesamtgewinn dividiert durch die Anzahl Datensätze. Der prozentuale Anteil Verkäufe von Office-Produkten ist die Anzahl der Datensätze mit Office-Produkten dividiert durch die Gesamtanzahl Datensätze multipliziert mit 100:

```
durchschnittlicherGewinn = gewinnGesamt / anzahlGesamt
prozentualerAnteilOffice = anzahlOfficeProdukte / anzahlGesamt * 100
```

Um den durchschnittlichen Gewinn bei Office-Produkten zu berechnen, müssen wir den Gesamtgewinn bei Office-Produkten durch die Anzahl der Datensätze mit Office-Produkten teilen. Doch was passiert, wenn wir keine Office-Produkte verkauft haben? Dann ist die Anzahl der Datensätze mit Office-Produkten Null. Was bedeutet, wir würden durch Null teilen. Dies geht aber nicht, unser Programm würde abstürzen. In diesem Fall müssen wir also prüfen, ob die Anzahl der Datensätze mit Office-Produkten nicht Null ist. Nur in diesem Fall dürfen wir die Division durchführen. Ansonsten setzen wir den Durchschnitt auf 0. Wir müssten also Programmcode schreiben, wie:

```
If anzahlOfficeProdukte <> 0 Then
    durchschnittlicherGewinnOffice = gewinnOfficeProdukte / anzahlOfficeProdukte
Else
    durchschnittlicherGewinnOffice = 0
End If
```

Uns fällt natürlich sofort auf, dass wir diese Vorgehensweise häufiger anwenden werden müssen. Nämlich immer dann, wenn wir durch eine Anzahl, die wir durch eine Einschränkung ermittelt haben, teilen wollen. Denn dann kann es immer vorkommen, dass wir keine Datensätze haben, für die die Einschränkung gilt. Dann müssen wir ein Teilen durch Null verhindern. Wir wissen allerdings auch, was wir tun müssen, wenn wir Vorgehensweisen haben, die wir häufiger mal programmieren müssen. Wir schreiben eine Funktion dafür:

Beispiel 10.4 *Die Funktion zum Dividieren unter Berücksichtigung eines Nenners mit Wert Null*

```
Function dividiereMitErgebnis0FuerNenner0Double(zaehler As Double, nenner As Long) As Double
    If nenner <> 0 Then
        dividiereMitErgebnis0FuerNenner0Double = zaehler / nenner
    Else
        dividiereMitErgebnis0FuerNenner0Double = 0
    End If
End Function
```

Diese Funktion müssten Sie sofort verstehen. Wie bei der oben dargestellten Maximumberechnung bereiten wir die Lösung darauf vor, dass wir dieselbe Funktionalität eventuell auch mit anderen Datentypen zur Verfügung stellen sollen. In diesem Fall ist der Zähler vom Datentyp *Double*, der Nenner ist *Long* [1]. Wir rufen diese Funktion aus unserem Programm auf:

```
durchschnittlicherGewinnOffice = dividiereMitErgebnis0FuerNenner0Double(gewinnOfficeProdukte, ➥
    anzahlOfficeProdukte)
```

Beachten Sie, dass wir die Problematik mit den Datentypen der Übergabeparameter wie bei der Funktion *bestimmeMaximumLong* hier nicht haben. Denn in den Fällen, für die diese Funktion vorgesehen ist, dividieren wir immer durch eine Anzahl von Datensätzen. Eine Anzahl ist aber immer vom Datentyp *Long*. Das Ziel dieser Funktion ist immer, einen Durchschnitt zu berechnen. Ein Durchschnitt ist aber nur sinnvoll, wenn der Wert aus dem er gebildet wird, ein *Double*-Wert ist. Und genau so ist diese Funktion definiert, der Zähler ist vom Typ *Double*, der Nenner vom Typ *Long*. Die Funktion gibt einen *Double*-Wert zurück und Durchschnitte sind immer vom Typ *Double*.
Zum Schluss schreiben wir unsere Ergebnisse in das Tabellenblatt 2:

```
Sheets(TABELLENBLATT_STATISTIK).Cells(1, 1) = "Durchschnittlicher Gewinn"
Sheets(TABELLENBLATT_STATISTIK).Cells(2, 1) = "Maximale Anzahl"
Sheets(TABELLENBLATT_STATISTIK).Cells(3, 1) = "Minimale Anzahl bei CD-Versand"
Sheets(TABELLENBLATT_STATISTIK).Cells(4, 1) = "Durchschnittlicher Gewinn bei Office-Produkten"
Sheets(TABELLENBLATT_STATISTIK).Cells(5, 1) = "Prozentualer Anteil Verkäufe von Office-Produkten"

Sheets(TABELLENBLATT_STATISTIK).Cells(1, 2) = durchschnittlicherGewinn
Sheets(TABELLENBLATT_STATISTIK).Cells(2, 2) = maxAnzahl
Sheets(TABELLENBLATT_STATISTIK).Cells(3, 2) = minAnzahlCDVersand
Sheets(TABELLENBLATT_STATISTIK).Cells(4, 2) = durchschnittlicherGewinnOffice
Sheets(TABELLENBLATT_STATISTIK).Cells(5, 2) = prozentualerAnteilOffice
```

10.2 Das Schlüsseldienstbeispiel

Wir betrachten das in Abb. 10.3 dargestellte Tabellenblatt. Daraus wollen wir die in in Abb. 10.4 dargestellten Statistiken erzeugen.
Die Schaltfläche zum Start des Statistikprogramms soll grundsätzlich in jedes Tabellenblatt einbaubar sein. Die Auswertung soll in Tabellenblatt 2 geschrieben werden. Sie in Tabellenblatt 1 zu schreiben ist nicht so sinnvoll, da wir nicht wissen, wieviel Zeilen in diesem Tabellenblatt gefüllt sein werden.
Die grundsätzliche Vorgehensweise ist analog zu der in Kapitel 9.3. Wir bestimmen zunächst die letzte besetzte Zeile. Dafür haben wir ja bereits eine Funktion. Dann gehen wir in einer Schleife über alle Datensätze. Die Gesamtanzahl Datensätze können wir dann wie in Kapitel 9.3 oder durch mitzählen in der Schleife bestimmen. Um den durchschnittlichen Preis bestimmen zu können, müssen wir den Gesamtpreis ausrechnen, dies erfolgt völlig analog der Berechnung der

[1] Da wir in unseren Beispielen immer Durchschnitte berechnen, wird der Nenner im vom Datentyp *Long* sein, daher brauchen wir nur den Zähler zu betrachten.

	A	B	C	D	E
1	Kunde	gefahrene km	Arbeitszeit (Min)	Nachtzuschlag	Preis
2	Meyer	20	20		70
3	Müller	10	10		35
4	Schmidt	50	25	ja	126
5	Schmid	15	35		100
6	Seran	10	11	ja	51,8
7	Fischer	7	7		29
8					

Abbildung 10.3
Daten des zweiten Beispiels

	A	B
1	Durchschnittlicher Preis	68,63 €
2	Maximale Arbeitszeit	35
3	Minimaler Preis bei Nachtzuschlag	51,80 €
4	Durchschnittlicher Preis ohne Nachtzuschlag	58,50 €
5	Prozentualer Anteil Aufträge ohne Nachtzuschlag	66,6666667
6		

Abbildung 10.4
Auswertung des zweiten Beispiels

Gesamtprovision in Kapitel 9.3 oder des Gesamtgewinns in Kapitel 10.1. Der durchschnittliche Preis ist dann der Gesamtpreis dividiert durch die Gesamtanzahl Datensätze. Zur Bestimmung des maximalen Arbeitszeit und des minimalen Preises bei Nachtzuschlag verfahren wir wie in Kap. 10.1 beschrieben. Daher kommen wir nun zur Berechnung des Gesamtpreises ohne Nachtzuschlag. Dies funktioniert aber grundsätzlich genauso, wie die Berechnung des Gesamtpreises. Allerdings darf die Aufsummierung hier nur dann stattfinden, wenn in der Spalte D nicht "ja" steht. Dies können wir aber leicht durch ein *if*-Konstrukt lösen. Zur Berechnung des durchschnittlichen Preises ohne Nachtzuschlag müssen wir zählen, wieviel Datensätze ohne Nachtzuschlag wir haben und dann den Gesamtpreis ohne Nachtzuschlag durch diese Anzahl teilen. Der prozentale Anteil Aufträge ohne Nachtzuschlag ist abschließend ja nur die Anzahl Datensätze ohne Nachtzuschlag dividiert die durch Gesamtanzahl Datensätze multipliziert mit 100.
Schauen wir uns nun die Lösung an:

Beispiel 10.5 *Das Programm zur Datenauswertung (Beispiel 2)*

```
Sub statistiken_click()
    Const NACHTZUSCHLAG_SPALTE As Long = 4
    Const PREIS_SPALTE As Long = 5
    Const ARBEITSZEIT_SPALTE As Long = 3
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Const TABELLENBLATT As Long = 1
    Const TABELLENBLATT_STATISTIK As Long = 2

    Dim anzahlGesamt As Long
    Dim anzahlOhneNachtzuschlag As Long
    Dim maxArbeitszeit As Double
    Dim minPreisBeiNachtzuschlag As Double
    Dim gesamtPreis As Double
    Dim gesamtPreisOhneNachtzuschlag As Double
    Dim arbeitszeit As Double
    Dim durchschnittlicherPreis As Double
    Dim prozentualerAnteilOhneNachtzuschlag As Double
    Dim durchschnittlicherPreisOhneNachtzuschlag As Double
    Dim nachtzuschlag As String
    Dim preis As Double
    Dim i As Long
    Dim letzteBesetzteZeile As Long
```

```
    anzahlGesamt = 0
    maxArbeitszeit = 0
    minPreisBeiNachtzuschlag = 10000
    gesamtPreis = 0
    gesamtPreisOhneNachtzuschlag = 0
    anzahlOhneNachtzuschlag = 0

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(PREIS_SPALTE, ➥
        _ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        anzahlGesamt = anzahlGesamt + 1

        preis = Sheets(TABELLENBLATT).Cells(i, PREIS_SPALTE)
        gesamtPreis = gesamtPreis + preis

        arbeitszeit = Sheets(TABELLENBLATT).Cells(i, ARBEITSZEIT_SPALTE)
        maxArbeitszeit = bestimmeMaximumDouble(maxArbeitszeit, arbeitszeit)

        nachtzuschlag = Sheets(TABELLENBLATT).Cells(i, NACHTZUSCHLAG_SPALTE)
        If nachtzuschlag = "ja" Then
            minPreisBeiNachtzuschlag = bestimmeMinimumDouble(minPreisBeiNachtzuschlag, preis)
        Else
            gesamtPreisOhneNachtzuschlag = gesamtPreisOhneNachtzuschlag + preis
            anzahlOhneNachtzuschlag = anzahlOhneNachtzuschlag + 1
        End If
    Next i

    durchschnittlicherPreis = gesamtPreis / anzahlGesamt
    prozentualerAnteilOhneNachtzuschlag = anzahlOhneNachtzuschlag / anzahlGesamt * 100
    durchschnittlicherPreisOhneNachtzuschlag = dividiereMitErgebnis0FuerNenner0Double(➥
        gesamtPreisOhneNachtzuschlag, anzahlOhneNachtzuschlag)

    Sheets(TABELLENBLATT_STATISTIK).Cells(1, 1) = "Durchschnittlicher Preis"
    Sheets(TABELLENBLATT_STATISTIK).Cells(2, 1) = "Maximale Arbeitszeit"
    Sheets(TABELLENBLATT_STATISTIK).Cells(3, 1) = "Minimaler Preis bei Nachtzuschlag"
    Sheets(TABELLENBLATT_STATISTIK).Cells(4, 1) = "Durchschnittlicher Preis ohne Nachtzuschlag"
    Sheets(TABELLENBLATT_STATISTIK).Cells(5, 1) = "Prozentualer Anteil Aufträge ohne ➥
        Nachtzuschlag"

    Sheets(TABELLENBLATT_STATISTIK).Cells(1, 2) = durchschnittlicherPreis
    Sheets(TABELLENBLATT_STATISTIK).Cells(2, 2) = maxArbeitszeit
    Sheets(TABELLENBLATT_STATISTIK).Cells(3, 2) = minPreisBeiNachtzuschlag
    Sheets(TABELLENBLATT_STATISTIK).Cells(4, 2) = durchschnittlicherPreisOhneNachtzuschlag
    Sheets(TABELLENBLATT_STATISTIK).Cells(5, 2) = prozentualerAnteilOhneNachtzuschlag
End Sub
```

Zunächst definieren wir einige Konstanten. Die ersten Konstanten definieren die Spalten, die wir im weiteren Verlauf benötigen:

```
Const NACHTZUSCHLAG_SPALTE As Long = 4
Const PREIS_SPALTE As Long = 5
Const ARBEITSZEIT_SPALTE As Long = 3
```

Dies ist sinnvoll, weil sich das Layout des Tabellenblatts verändern kann. Nehmen wir an, ein Entscheider im Unternehmen möchte die Spalte mit den gefahrenen Kilometern vor der Arbeitszeit-Spalte stehen haben. Dann müssen wir nur die Werte der Konstanten *ARBEITSZEIT_SPALTE* anpassen. Aus dem gleichen Grund legen wir die Zeile, in der die Daten beginnen, das Tabellenblatt, in dem die Daten stehen, sowie das Tabellenblatt, in das die Statistiken geschrieben werden sollen, auf Konstanten.

```
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
Const TABELLENBLATT As Long = 1
Const TABELLENBLATT_STATISTIK As Long = 2
```

Dann erfolgt die Deklaration der Variablen, die wir für die Berechnung benötigen:

```
Dim anzahlGesamt As Long
Dim anzahlOhneNachtzuschlag As Long
Dim maxArbeitszeit As Double
Dim minPreisBeiNachtzuschlag As Double
Dim gesamtPreis As Double
Dim gesamtPreisOhneNachtzuschlag As Double
Dim arbeitszeit As Double
Dim durchschnittlicherPreis As Double
Dim prozentualerAnteilOhneNachtzuschlag As Double
Dim durchschnittlicherPreisOhneNachtzuschlag As Double
Dim nachtzuschlag As String
Dim preis As Double
Dim i As Long
Dim letzteBesetzteZeile As Long
```

Es folgen die Initialisierungen.

```
anzahlGesamt = 0
maxArbeitszeit = 0
minPreisBeiNachtzuschlag = 10000
gesamtPreis = 0
gesamtPreisOhneNachtzuschlag = 0
anzahlOhneNachtzuschlag = 0
```

Warum diese notwendig sind, haben wir in Kapitel 10.1 eingehend besprochen. Jede Variable, die in einer Anweisung gleichzeitig auf der rechten und der linken Seite des Zuweisungsoperators vorkommt, muss zu diesem Zeitpunkt einen definierten Wert haben. Für unsere Auswertungsprogramme bedeutet dies, dass diese Variablen initialisiert werden müssen. Und weil auf diesen Variablen in unseren Fällen immer Aufsummierungen gespeichert werden, müssen diese Variablen mit 0 initialisiert werden. Die Variable, auf der das Maximum gespeichert wird, initialisieren wir mit einer Zahl, die zu klein ist, als das sie als Maximum in Frage kommt. Daher wird *maxArbeitszeit* mit 0 initialisiert. Das Minimum initialisieren wir mit einem Wert, der zu groß ist, als das er als Minimum in Frage kommt. Hier initialisieren wir *minPreisBeiNachtzuschlag* mit 10000.
Wir bestimmen die letzte besetzte Zeile, also die Zeile, bis zu der unsere Schleife laufen muss.

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(PREIS_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
```

Zur Bestimmung der letzten besetzten Zeile können wir grundsätzlich jede Spalte des Tabellenblatts bis zur Spalte E nehmen, mit Ausnahme der Spalte D. Die Spalte D enthält leere Zellen. Das Verfahren, das uns die letzte besetzte Zeile bestimmt, würde in der Spalte D bereits in Zeile 2 abbrechen, da die Zelle D2 leer ist. Unsere Funktion zur Ermittlung der letzten besetzten Zeile würde in der Spalte D also 1 ermitteln. Wir nehmen Spalte E, die Preis-Spalte. Die Zeile, in der wir die Bestimmung der letzten besetzten Zeile starten, ist natürlich die Zeile 2, weil hier die Daten beginnen. Diese hatten wir ja auf der Konstanten *ERSTE_ZEILE_MIT_INFORMATIONEN* abgelegt. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteZeileInSpalte* hatten wir in Kapitel 9.3 eingehend besprochen, so dass wir auf eine erneute Diskussion verzichten. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun startet unsere Schleife. Sie läuft natürlich von Zeile 2 (*ERSTE_ZEILE_MIT_INFORMATIONEN*) zur letzten besetzten Zeile, da ja jeder Datensatz behandelt werden muss.

```
For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
    anzahlGesamt = anzahlGesamt + 1
```

Die Zeile

```
    anzahlGesamt = anzahlGesamt + 1
```

zählt die Anzahl der Datensätze im Tabellenblatt. Die Variable *anzahlGesamt* wird bei jedem Durchlauf der Schleife durch obige Anweisung um 1 erhöht. In den Zeilen

```
    preis = Sheets(TABELLENBLATT).Cells(i, PREIS_SPALTE)
    gesamtPreis = gesamtPreis + preis
```

wird zunächst der Inhalt der Zelle der Spalte mit dem Preis in der jeweiligen Zeile in der Variablen *preis* gespeichert. Wegen der Anforderung die Schaltfläche zum Programmstart in jedes Zabellenblatt einbauen zu können, ist nicht mehr sichergestellt, dass sich Daten und Schaltfläche im gleichen Tabellenblatt befinden. Wir müssen also unter Angabe des Tabellenblattes auf die Zelle zugreifen. Daher ist die Angabe *Sheets(TABELLENBLATT).Cells(i, PREIS_SPALTE)* notwendig. *Cells(i, PREIS_SPALTE)* reicht nicht mehr. Diese Vorgehensweise ist natürlich auch bei allen anderen Zellzugriffen in diesem Programm notwendig.
Danach wird der Wert auf die Variable *gesamtPreis* addiert, so dass nach dem letzten Schleifendurchlauf auf dieser Variablen die Summe aller Preise (der Gesamtpreis) gespeichert ist. Die Variablen *gesamtPreis* und *anzahlGesamt* müssen, da sie auf beiden Seiten des Gleichheitszeichen stehen, mit 0 initialisiert werden.
Nun folgt die Bestimmung des Maximums. Hierbei gehen wir folgendermaßen vor: Zunächst initialisieren wir die Variable, auf der das Maximum gespeichert wird, mit einer Zahl, die zu klein ist, als das sie als Maximum in Frage kommt.

```
maxArbeitszeit = 0
```

Hier wird *maxArbeitszeit* mit 0 initialisiert. Da es sich hier um eine Arbeitszeit handelt, ist 0 kleiner als jede erlaubte Arbeitszeit. In der Schleife nun wird der Wert in der gerade bearbeiteten Zeile mit dem Wert, den das Maximum gerade hat, verglichen. Ist das bisherige Maximum größer als der jetzige Wert, passiert nichts. Ist das bisherige Maximum allerdings kleiner als der jetzige Wert, wird das Maximum auf den gerade behandelten Wert geändert. Eine noch ausführlichere Beschreibung des Verfahrens findet sich in Kapitel 10.1.
Wie programmieren wir dies? Die Initialisierung haben wir bereits durchgeführt, die Variable *maxArbeitszeit* ist bereits mit Null initialisiert. In der Schleife müssen wir nun noch den jetzigen Wert von *maxArbeitszeit* mit dem Wert der Arbeitszeit der jeweiligen Zeile vergleichen und wenn der Wert in der jeweiligen Zeile größer ist, dann müssen wir den Wert von *maxArbeitszeit* umsetzen. Hierfür schreiben wir eine Funktion

Beispiel 10.6 *Die Funktion zur Maximumbestimmung für Double-Variablen*

```
Function bestimmeMaximumDouble(wert1 As Double, wert2 As Double) As Double
    If wert1 >= wert2 Then
        bestimmeMaximumDouble = wert1
    Else
        bestimmeMaximumDouble = wert2
    End If
End Function
```

und rufen diese in der Schleife für jeden Datensatz auf:

```
    arbeitszeit = Sheets(TABELLENBLATT).Cells(i, ARBEITSZEIT_SPALTE_SPALTE)
    maxArbeitszeit = bestimmeMaximumDouble(maxArbeitszeit, arbeitszeit)
```

Bereits in Kap. 10.1 hatten wir auf die Wichtigkeit der Übergabeparameter hingewiesen. Wir hatten dort eine Funktion zur Maximumbestimmung zweier *Long*-Variablen geschrieben (vgl. Beispiel 10.2). Diese können wir hier nicht verwenden, da es sich sowohl bei der Variablen *maxArbeitszeit*, als auch bei der Variablen *arbeitszeit* um *Double*-Variablen handelt. Wir benötigen also eine analoge Funktion für *Double*-Variablen. Diese erhält den Namen *bestimmeMaximumDouble* und ist in Beispiel 10.6 dargestellt.
Kommen wir nun zum nächsten Punkt: Minimaler Preis bei Nachtzuschlag. Ein Minimum bestimmen wir im Prinzip genauso wie ein Maximum. Initialisieren müssen wir allerdings mit einem Wert, der zu groß ist, als das er als Minimum in Frage kommt. Dies tun wir hier:

```
    minPreisBeiNachtzuschlag = 10000
```

Nun wählen wir die gleiche Vorgehensweise wie bei der Maximumbestimmung. Die einzige Änderung ist, dass wir das Minimum dann umsetzen, wenn der Wert in der Zeile, in der wir uns befinden, kleiner ist als das bisherige Minimum. Wir schreiben auch Hierfür eine Funktion:

Beispiel 10.7 *Die Funktion zur Minimumbestimmung für Double-Variablen*

```
Function bestimmeMinimumDouble(wert1 As Double, wert2 As Double) As Double
    If wert1 <= wert2 Then
        bestimmeMinimumDouble = wert1
    Else
        bestimmeMinimumDouble = wert2
    End If
End Function
```

Diese Funktion ist analog zu Beispiel 10.3 aus Kap. 10.1. Wir mussten sie schreiben, weil Beispiel 10.3 nur für *Long*-Variablen funktioniert und wir hier *Double*-Variablen haben.
Wir müssen eine weitere Einschränkung betrachten: Die Minimumbestimmung soll nur über die Aufträge mit Nachtzuschlag gehen. Dies bedeutet, dass wir in der Schleife nur dann tätig werden müssen, wenn in der Spalte für den Nachtzuschlag (Spalte D) „ja“ steht. Dies können wir aber wieder einfach durch ein *if*-Statement regeln. Daher resultiert folgender Programmcode:

```
    If nachtzuschlag = "ja" Then
        minPreisBeiNachtzuschlag = bestimmeMinimumDouble(minPreisBeiNachtzuschlag, preis)
```

Es bleiben noch die Auswertungen für Aufträge ohne Nachtzuschlag. Um diese ausrechnen zu können, müssen wir den Gesamtpreis ohne Nachtzuschlag und die Anzahl Datensätze, die sich auf Aufträge ohne Nachtzuschlag beziehen, ermitteln. Das ist aber prinzipiell dieselbe Vorgehensweise, die wir bereits bei der Ermittlung des Gesamtpreises und der Gesamtanzahl Datensätze angewendet haben. Der einzige Unterschied ist, dass wir in der Schleife nur dann tätig werden, wenn in der Spalte für den Nachtzuschlag (Spalte D) nicht „ja“ steht. Wie man so etwas macht, wissen wir aber auch schon, dazu reicht ein einfaches *if*-Statement. Hier haben wir es sogar etwas einfacher, denn durch die Zeile

```
    If nachtzuschlag = "ja" Then
```

haben wir bereits ein *if*-Statement, das sich auf die Nachtzuschlags-Spalte bezieht begonnen. Wir können einfach mit einem else-Teil fortfahren:

```
    Else
        gesamtPreisOhneNachtzuschlag = gesamtPreisOhneNachtzuschlag + preis
        anzahlOhneNachtzuschlag = anzahlOhneNachtzuschlag + 1
    End If
```

Nun beenden wir die Schleife.

```
Next i
```

Nach der Schleife müssen wir die Durchschnitte und prozentualen Anteile berechnen. Der durchschnittliche Preis ist ja einfach der Gesamtpreis dividiert durch die Anzahl Datensätze. Der prozentale Anteil von Aufträgen ohne Nachtzuschlag ist die Anzahl der Datensätze mit Aufträgen ohne Nachtzuschlag dividiert durch die Gesamtanzahl Datensätze multipliziert mit 100:

```
durchschnittlicherPreis = gesamtPreis / anzahlGesamt
prozentualerAnteilOhneNachtzuschlag = anzahlOhneNachtzuschlag / anzahlGesamt * 100
```

Um den durchschnittlichen Preis bei Aufträgen ohne Nachtzuschlag zu berechnen, müssen wir den Gesamtpreis bei Aufträgen ohne Nachtzuschlag durch die Anzahl der Datensätze mit Aufträgen ohne Nachtzuschlag teilen. Doch was passiert, wenn wir keine Aufträge ohne Nachtzuschlag erhalten haben? Dann ist die Anzahl der Datensätze mit Aufträgen ohne Nachtzuschlag Null. Was bedeutet, wir würden durch Null teilen. Dies geht aber nicht, unser Programm würde abstürzen. In diesem Fall müssen wir also prüfen, ob die Anzahl der Datensätze mit Aufträgen ohne Nachtzuschlag nicht Null ist. Nur in diesem Fall dürfen wir die Division durchführen. Ansonsten setzen wir den Durchschnitt auf 0. Hierfür haben wir in Kap. 10.1 bereits die Funktion *dividiereMitErgebnis0FuerNenner0Double* geschrieben (Beispiel 10.4). Wir rufen sie einfach auf (dafür muss die Funktion natürlich vorhanden sein):

```
durchschnittlicherPreisOhneNachtzuschlag = dividiereMitErgebnis0FuerNenner0Double(➥
    gesamtPreisOhneNachtzuschlag, anzahlOhneNachtzuschlag)
```

Diese Vorgehensweise müssen wir immer anwenden, wenn wir durch eine Anzahl, die wir durch eine Einschränkung ermittelt haben, teilen wollen. Denn dann kann es immer vorkommen, dass wir keine Datensätze haben, für die die Einschränkung gilt. Dann müssen wir ein Teilen durch Null verhindern.
Zum Schluss schreiben wir unsere Ergebnisse in das Tabellenblatt 2:

```
    Sheets(TABELLENBLATT_STATISTIK).Cells(1, 1) = "Durchschnittlicher Preis"
    Sheets(TABELLENBLATT_STATISTIK).Cells(2, 1) = "Maximale Arbeitszeit"
    Sheets(TABELLENBLATT_STATISTIK).Cells(3, 1) = "Minimaler Preis bei Nachtzuschlag"
    Sheets(TABELLENBLATT_STATISTIK).Cells(4, 1) = "Durchschnittlicher Preis ohne Nachtzuschlag"
    Sheets(TABELLENBLATT_STATISTIK).Cells(5, 1) = "Prozentualer Anteil Aufträge ohne ➥
        Nachtzuschlag"

    Sheets(TABELLENBLATT_STATISTIK).Cells(1, 2) = durchschnittlicherPreis
    Sheets(TABELLENBLATT_STATISTIK).Cells(2, 2) = maxArbeitszeit
    Sheets(TABELLENBLATT_STATISTIK).Cells(3, 2) = minPreisBeiNachtzuschlag
    Sheets(TABELLENBLATT_STATISTIK).Cells(4, 2) = durchschnittlicherPreisOhneNachtzuschlag
    Sheets(TABELLENBLATT_STATISTIK).Cells(5, 2) = prozentualerAnteilOhneNachtzuschlag
```

10.3 Das Zuweisungsbeispiel

Wir betrachten das in Abb. 10.5 dargestellte Tabellenblatt:

	A	B	C	D	E	F
1	Studiengang	Abschluss	Absolventen	davon in Regelstudienzeit	Kapazität	Zuweisung
2	Bachelor of Arts	Bachelor	85	50	100	17550
3	Wirtschaftsingenieur	Bachelor	9	1	15	500
4	MAAT	Master	10	10	20	650
5	Wirtschaftsinformatik	Bachelor	5	1	15	150
6						

Abbildung 10.5
Daten des dritten Beispiels

Daraus wollen wir die in in Abb. 10.6 dargestellten Statistiken erzeugen.

	A	B
1	Gesamtanzahl Datensätze	4
2	Gesamtzuweisung	18850
3	Durchschnittszuweisung	4712,5
4	Gesamtanzahl Absolventen nicht in Regelstudienzeit	47
5	Gesamtanzahl Bachelorabsolventen nicht in Regelstudienzeit	47
6	Durchschnittliche Anzahl Bachelorabsolventen nicht in Regelstudienzeit	15,6666667
7	Gesamtanzahl Masterabsolventen nicht in Regelstudienzeit	0
8	Durchschnittliche Anzahl Masterabsolventen nicht in Regelstudienzeit	0
9	Maximale Anzahl Absolventen nicht in Regelstudienzeit	35
10	Minimale Anzahl Absolventen nicht in Regelstudienzeit	0
11		

Abbildung 10.6
Auswertung des dritten Beispiels

Die Schaltfläche zum Start des Statistikprogramms soll grundsätzlich in jedes Tabellenblatt einbaubar sein. Die Auswertung soll in Tabellenblatt 2 geschrieben werden. Sie in Tabellenblatt 1 zu schreiben ist nicht so sinnvoll, da wir nicht wissen, wieviel Zeilen in diesem Tabellenblatt gefüllt sein werden.
Die grundsätzliche Vorgehensweise ist analog zu der in Kapitel 9.3. Wir bestimmen zunächst die letzte besetzte Zeile. Dafür haben wir ja bereits eine Funktion. Dann gehen wir in einer Schleife über alle Datensätze. Die Gesamtanzahl Datensätze können wir dann wie in Kapitel 9.3 oder durch mitzählen in der Schleife bestimmen. Um die durchschnittliche

Zuweisung bestimmen zu können, müssen wir die Gesamtzuweisung ausrechnen, dies erfolgt völlig analog der Berechnung der Gesamtprovision in Kapitel 9.3 oder des Gesamtgewinns in Kapitel 10.1. Die durchschnittliche Zuweisung ist dann der Gesamtzuweisung dividiert durch die Gesamtanzahl Datensätze.
Für die Berechnung der Auswertungen, die auf der Anzahl von Absolventen nicht in Regelstudienzeit beruhen, müssen wir erst in jeder Zeile diese Anzahl ausrechnen. Das ist aber auch nicht weiter kompliziert, denn die Anzahl der Absolventen nicht in Regelstudienzeit ist ja einfach die Anzahl Absolventen minus der Anzahl Absolventen in Regelstudienzeit. Nachdem wir diese Größe berechnet haben, können wir damit:

- Die Gesamtanzahl Absolventen nicht in Regelstudienzeit bestimmen, wie zuvor die Gesamtzuweisung.
- Mit dem gleichen Verfahren wie in Kap. 10.1 beschrieben, die maximale und minimale Anzahl nicht in Regelstudienzeit bestimmen.
- Durch ein *if*-Konstrukt Gesamt- und Durchschnittszahlen für Bachelor- bzw. Master-Studiengänge bestimmen.

Schauen wir uns nun die Lösung an:

Beispiel 10.8 *Das Programm zur Datenauswertung (Beispiel 3)*

```
Sub statistiken_click()
    Const ABSCHLUSS_SPALTE As Long = 2
    Const ABSOLVENTEN_SPALTE As Long = 3
    Const DAVON_REGELSTUDIENZEIT_SPALTE As Long = 4
    Const ZUWEISUNG_SPALTE As Long = 6
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Const TABELLENBLATT As Long = 1
    Const TABELLENBLATT_STATISTIK As Long = 2

    Dim anzahlGesamt As Long
    Dim zuweisung As Double
    Dim gesamtZuweisung As Double
    Dim anzahlNichtRegelstudienzeit As Long
    Dim maxAnzahlNichtRegelstudienzeit As Long
    Dim minAnzahlNichtRegelstudienzeit
    Dim gesamtAnzahlNichtRegelstudienzeit
    Dim abschluss As String
    Dim gesamtAnzahlNichtRegelstudienzeitBachelor As Long
    Dim gesamtAnzahlNichtRegelstudienzeitMaster As Long
    Dim durchschnittlicheZweisung As Double
    Dim durchschnittlicheAnzahlNichtRegelstudienzeitBachelor As Double
    Dim durchschnittlicheAnzahlNichtRegelstudienzeitMaster As Double
    Dim anzahlMaster As Long
    Dim anzahlBachelor As Long

    Dim i As Long
    Dim letzteBesetzteZeile As Long

    anzahlGesamt = 0
    gesamtAnzahlNichtRegelstudienzeit = 0
    maxAnzahlNichtRegelstudienzeit = 0
    minAnzahlNichtRegelstudienzeit = 1000
    gesamtAnzahlNichtRegelstudienzeitBachelor = 0
    gesamtAnzahlNichtRegelstudienzeitMaster = 0
    anzahlMaster = 0
    anzahlBachelor = 0

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(ZUWEISUNG_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        anzahlGesamt = anzahlGesamt + 1

        zuweisung = Sheets(TABELLENBLATT).Cells(i, ZUWEISUNG_SPALTE)
        gesamtZuweisung = gesamtZuweisung + zuweisung
```

```
        anzahlNichtRegelstudienzeit = Sheets(TABELLENBLATT).Cells(i, ABSOLVENTEN_SPALTE) - Sheets➥
            (TABELLENBLATT).Cells(i, DAVON_REGELSTUDIENZEIT_SPALTE)
        maxAnzahlNichtRegelstudienzeit = bestimmeMaximumLong(maxAnzahlNichtRegelstudienzeit, ➥
            anzahlNichtRegelstudienzeit)
        minAnzahlNichtRegelstudienzeit = bestimmeMinimumLong(minAnzahlNichtRegelstudienzeit, ➥
            anzahlNichtRegelstudienzeit)

        gesamtAnzahlNichtRegelstudienzeit = gesamtAnzahlNichtRegelstudienzeit + ➥
            anzahlNichtRegelstudienzeit
        abschluss = Sheets(TABELLENBLATT).Cells(i, ABSCHLUSS_SPALTE)
        If abschluss = "Bachelor" Then
            gesamtAnzahlNichtRegelstudienzeitBachelor = gesamtAnzahlNichtRegelstudienzeitBachelor➥
                + anzahlNichtRegelstudienzeit
            anzahlBachelor = anzahlBachelor + 1
        ElseIf abschluss = "Master" Then
            gesamtAnzahlNichtRegelstudienzeitMaster = gesamtAnzahlNichtRegelstudienzeitMaster + ➥
                anzahlNichtRegelstudienzeit
            anzahlMaster = anzahlMaster + 1
        End If
    Next i

    durchschnittlicheZweisung = gesamtZuweisung / anzahlGesamt
    durchschnittlicheAnzahlNichtRegelstudienzeitBachelor = dividiereMitErgebnis0FuerNenner0Long(➥
        gesamtAnzahlNichtRegelstudienzeitBachelor, anzahlBachelor)
    durchschnittlicheAnzahlNichtRegelstudienzeitMaster = dividiereMitErgebnis0FuerNenner0Long(➥
        gesamtAnzahlNichtRegelstudienzeitMaster, anzahlMaster)

    Sheets(TABELLENBLATT_STATISTIK).Cells(1, 1) = "Gesamtanzahl Datensätze"
    Sheets(TABELLENBLATT_STATISTIK).Cells(2, 1) = "Gesamtzuweisung"
    Sheets(TABELLENBLATT_STATISTIK).Cells(3, 1) = "Durchschnittszuweisung"
    Sheets(TABELLENBLATT_STATISTIK).Cells(4, 1) = "Gesamtanzahl Absolventen nicht in ➥
        Regelstudienzeit"
    Sheets(TABELLENBLATT_STATISTIK).Cells(5, 1) = "Gesamtanzahl Bachelorabsolventen nicht in ➥
        Regelstudienzeit"
    Sheets(TABELLENBLATT_STATISTIK).Cells(6, 1) = "Durchschnittliche Anzahl Bachelorabsolventen ➥
        nicht in Regelstudienzeit"
    Sheets(TABELLENBLATT_STATISTIK).Cells(7, 1) = "Gesamtanzahl Masterabsolventen nicht in ➥
        Regelstudienzeit"
    Sheets(TABELLENBLATT_STATISTIK).Cells(8, 1) = "Durchschnittliche Anzahl Masterabsolventen ➥
        nicht in Regelstudienzeit"
    Sheets(TABELLENBLATT_STATISTIK).Cells(9, 1) = "Maximale Anzahl Absolventen nicht in ➥
        Regelstudienzeit"
    Sheets(TABELLENBLATT_STATISTIK).Cells(10, 1) = "Minimale Anzahl Absolventen nicht in ➥
        Regelstudienzeit"

    Sheets(TABELLENBLATT_STATISTIK).Cells(1, 2) = anzahlGesamt
    Sheets(TABELLENBLATT_STATISTIK).Cells(2, 2) = gesamtZuweisung
    Sheets(TABELLENBLATT_STATISTIK).Cells(3, 2) = durchschnittlicheZweisung
    Sheets(TABELLENBLATT_STATISTIK).Cells(4, 2) = gesamtAnzahlNichtRegelstudienzeit
    Sheets(TABELLENBLATT_STATISTIK).Cells(5, 2) = gesamtAnzahlNichtRegelstudienzeitBachelor
    Sheets(TABELLENBLATT_STATISTIK).Cells(6, 2) = ➥
        durchschnittlicheAnzahlNichtRegelstudienzeitBachelor
    Sheets(TABELLENBLATT_STATISTIK).Cells(7, 2) = gesamtAnzahlNichtRegelstudienzeitMaster
    Sheets(TABELLENBLATT_STATISTIK).Cells(8, 2) = ➥
        durchschnittlicheAnzahlNichtRegelstudienzeitMaster
    Sheets(TABELLENBLATT_STATISTIK).Cells(9, 2) = maxAnzahlNichtRegelstudienzeit
    Sheets(TABELLENBLATT_STATISTIK).Cells(10, 2) = minAnzahlNichtRegelstudienzeit
End Sub
```

Zunächst definieren wir einige Konstanten. Die ersten Konstanten definieren die Spalten, die wir im weiteren Verlauf benötigen:

```
Const ABSCHLUSS_SPALTE As Long = 2
Const ABSOLVENTEN_SPALTE As Long = 3
Const DAVON_REGELSTUDIENZEIT_SPALTE As Long = 4
Const ZUWEISUNG_SPALTE As Long = 6
```

Dies ist sinnvoll, weil sich das Layout des Tabellenblatts verändern kann. Nehmen wir an, ein Entscheider im Unternehmen möchte die Spalte mit den Absolventen vor der Abschluss-Spalte stehen haben. Dann müssen wir nur die Werte der dementsprechenden Konstanten anpassen. Aus dem gleichen Grund legen wir die Zeile, in der die Daten beginnen, das Tabellenblatt, in dem die Daten stehen, sowie das Tabellenblatt, in das die Statistiken geschrieben werden sollen, auf Konstanten.

```
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
Const TABELLENBLATT As Long = 1
Const TABELLENBLATT_STATISTIK As Long = 2
```

Dann erfolgt die Deklaration der Variablen, die wir für die Berechnung benötigen:

```
Dim anzahlGesamt As Long
Dim zuweisung As Double
Dim gesamtZuweisung As Double
Dim anzahlNichtRegelstudienzeit As Long
Dim maxAnzahlNichtRegelstudienzeit As Long
Dim minAnzahlNichtRegelstudienzeit
Dim gesamtAnzahlNichtRegelstudienzeit
Dim abschluss As String
Dim gesamtAnzahlNichtRegelstudienzeitBachelor As Long
Dim gesamtAnzahlNichtRegelstudienzeitMaster As Long
Dim durchschnittlicheZweisung As Double
Dim durchschnittlicheAnzahlNichtRegelstudienzeitBachelor As Double
Dim durchschnittlicheAnzahlNichtRegelstudienzeitMaster As Double
Dim anzahlMaster As Long
Dim anzahlBachelor As Long

Dim i As Long
Dim letzteBesetzteZeile As Long
```

Es folgen die Initialisierungen.

```
anzahlGesamt = 0
gesamtAnzahlNichtRegelstudienzeit = 0
maxAnzahlNichtRegelstudienzeit = 0
minAnzahlNichtRegelstudienzeit = 1000
gesamtAnzahlNichtRegelstudienzeitBachelor = 0
gesamtAnzahlNichtRegelstudienzeitMaster = 0
anzahlMaster = 0
anzahlBachelor = 0
```

Warum diese notwendig sind, haben wir in Kapitel 10.1 eingehend besprochen. Jede Variable, die in einer Anweisung gleichzeitig auf der rechten und der linken Seite des Zuweisungsoperators vorkommt, muss zu diesem Zeitpunkt einen definierten Wert haben. Für unsere Auswertungsprogramme bedeutet dies, dass diese Variablen initialisiert werden müssen. Und weil auf diesen Variablen in unseren Fällen immer Aufsummierungen gespeichert werden, müssen diese Variablen mit 0 initialisiert werden. Die Variable, auf der das Maximum gespeichert wird, initialisieren wir mit einer Zahl, die zu klein ist, als das sie als Maximum in Frage kommt. Daher wird *maxAnzahlNichtRegelstudienzeit* mit 0 initialisiert. Das Minimum initialisieren wir mit einem Wert, der zu groß ist, als das er als Minimum in Frage kommt. Hier initialisieren wir *minAnzahlNichtRegelstudienzeit* mit 1000.
Wir bestimmen die letzte besetzte Zeile, also die Zeile, bis zu der unsere Schleife laufen muss.

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(ZUWEISUNG_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
```

Zur Bestimmung der letzten besetzten Zeile können wir jede Spalte des Tabellenblatts bis zur Spalte F nehmen. Wir nehmen Spalte F, die Zuweisungs-Spalte. Die Zeile, in der wir die Bestimmung der letzten besetzten Zeile starten, ist natürlich die Zeile 2, weil hier die Daten beginnen. Diese hatten wir ja auf der Konstanten *ERSTE_ZEILE_MIT_INFORMATIONEN* abgelegt. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteZeileInSpalte* hatten wir in Kapitel 9.3 eingehend besprochen, so dass auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.

Nun startet unsere Schleife. Sie läuft natürlich von Zeile 2 (*ERSTE_ZEILE_MIT_INFORMATIONEN*) zur letzten besetzten Zeile, da ja jeder Datensatz behandelt werden muss.

```
For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
    anzahlGesamt = anzahlGesamt + 1
```

Die Zeile

```
    anzahlGesamt = anzahlGesamt + 1
```

zählt die Anzahl der Datensätze im Tabellenblatt. Die Variable *anzahlGesamt* wird bei jedem Durchlauf der Schleife durch obige Anweisung um 1 erhöht. In den Zeilen

```
    zuweisung = Sheets(TABELLENBLATT).Cells(i, ZUWEISUNG_SPALTE)
    gesamtZuweisung = gesamtZuweisung + zuweisung
```

wird zunächst der Inhalt der Zelle der Spalte mit der Zuweisung in der jeweiligen Zeile in der Variablen *zuweisung* gespeichert. Wegen der Anforderung die Schaltfläche zum Programmstart in jedes Zabellenblatt einbauen zu können, ist nicht mehr sichergestellt, dass sich Daten und Schaltfläche im gleichen Tabellenblatt befinden. Wir müssen also unter Angabe des Tabellenblattes auf die Zelle zugreifen. Daher ist die Angabe *Sheets(TABELLENBLATT).Cells(i, ZUWEISUNG_SPALTE)* notwendig. *Cells(i, ZUWEISUNG_SPALTE)* reicht nicht mehr. Diese Vorgehensweise ist natürlich auch bei allen anderen Zellzugriffen in diesem Programm notwendig.
Danach wird der Wert auf die Variable *gesamtZuweisung* addiert, so dass nach dem letzten Schleifendurchlauf auf dieser Variablen die Summe aller Zuweisungen (die Gesamtzuweisung) gespeichert ist. Die Variablen *gesamtZuweisung* und *anzahlGesamt* müssen, da sie auf beiden Seiten des Gleichheitszeichen stehen, mit 0 initialisiert werden.
Nun rechnen wir die Anzahl der Absolventen nicht in Regelstudienzeit aus. Hier müssen wir aber einfach den Inhalt der Spalte mit den Absolventen in Regelstudienzeit vom Inhalt der Spalte mit den Absolventen in der gerade bearbeiteten Zeile abziehen:

```
    anzahlNichtRegelstudienzeit = Sheets(TABELLENBLATT).Cells(i, ABSOLVENTEN_SPALTE) - Sheets(➥
        TABELLENBLATT).Cells(i, DAVON_REGELSTUDIENZEIT_SPALTE)
```

Nun folgt die Bestimmung des Maximums. Hierbei gehen wir folgendermaßen vor: Zunächst initialisieren wir die Variable, auf der das Maximum gespeichert wird, mit einer Zahl, die zu klein ist, als das sie als Maximum in Frage kommt.

```
maxAnzahlNichtRegelstudienzeit = 0
```

Hier wird *maxAnzahlNichtRegelstudienzeit* mit 0 initialisiert. Da es sich hier um eine Anzahl Absolventen handelt, ist 0 kleiner als jede erlaubte Absolventenzahl. In der Schleife nun wird der Wert in der gerade bearbeiteten Zeile mit dem Wert, den das Maximum gerade hat, verglichen. Ist das bisherige Maximum größer als der jetzige Wert, passiert nichts. Ist das bisherige Maximum allerdings kleiner als der jetzige Wert, wird das Maximum auf den gerade behandelten Wert geändert. Eine noch ausführlichere Beschreibung des Verfahrens findet sich in Kapitel 10.1.
Wie programmieren wir dies? Die Initialisierung haben wir bereits durchgeführt, die Variable *maxAnzahlNichtRegelstudienzeit* ist bereits mit Null initialisiert. In der Schleife müssen wir nun noch den jetzigen Wert von *maxAnzahlNichtRegelstudienzeit* mit dem Wert der Absolventen nicht in Regelstudienzeit der jeweiligen Zeile vergleichen und wenn der Wert in der jeweiligen Zeile größer ist, dann müssen wir den Wert von *maxAnzahlNichtRegelstudienzeit* umsetzen. *maxAnzahlNichtRegelstudienzeit* ist vom Datentyp *Long*, *anzahlNichtRegelstudienzeit* ebenso. Für den Vergleich können wir also die Funktion *bestimmeMaximumLong* aus Kap. 10.1 nutzen. Die Implementierung finden Sie in Beispiel 10.2. Hier müssen wir die Funktion nur aufrufen (die Funktion muss dafür natürlich vorhanden sein).

```
    maxAnzahlNichtRegelstudienzeit = bestimmeMaximumLong(maxAnzahlNichtRegelstudienzeit, ➥
        anzahlNichtRegelstudienzeit)
```

Kommen wir nun zum nächsten Punkt: Minimale Anzahl Studenten nicht in Regelstudienzeit. Ein Minimum bestimmen wir im Prinzip genauso wie ein Maximum. Initialisieren müssen wir allerdings mit einem Wert, der zu groß ist, als das er als Minimum in Frage kommt. Dies tun wir hier:

```
minAnzahlNichtRegelstudienzeit = 1000
```

Nun wählen wir die gleiche Vorgehensweise wie bei der Maximumbestimmung. Die einzige Änderung ist, dass wir das Minimum dann umsetzen, wenn der Wert in der Zeile, in der wir uns befinden, kleiner ist als das bisherige Minimum. *minAnzahlNichtRegelstudienzeit* ist vom Datentyp *Long*, *anzahlNichtRegelstudienzeit* ebenso. Für den Vergleich können wir also die Funktion *bestimmeMinimumLong* aus Kap. 10.1 nutzen. Die Implementierung finden Sie in Beispiel 10.3. Hier müssen wir die Funktion nur aufrufen (die Funktion muss dafür natürlich vorhanden sein).

```
    minAnzahlNichtRegelstudienzeit = bestimmeMinimumLong(minAnzahlNichtRegelstudienzeit, ➥
        anzahlNichtRegelstudienzeit)
```

Nun müssen wir nocht die Gesamtanzahl Absolventen nicht in Regelstudienzeit bestimmen, dies ist aber völlig analog zur Gesamtzuweisung:

```
    gesamtAnzahlNichtRegelstudienzeit = gesamtAnzahlNichtRegelstudienzeit + ➥
        anzahlNichtRegelstudienzeit
```

Die Variable *gesamtAnzahlNichtRegelstudienzeit* wurde mit 0 initialisiert, da es sich um eine Variable für eine Aufsummierung handelt.
Es bleiben noch die abschlussbezogenen Auswertungen. Hier müssen wir aber nur die Gesamtanzahl Absolventen nicht in Regelstudienzeit sowie die Anzahl Datensätze für jeden Abschluss bestimmen. Dies können wir aber einfach durch ein *if-elseif*-Konstrukt realisieren:

```
    abschluss = Sheets(TABELLENBLATT).Cells(i, ABSCHLUSS_SPALTE)
    If abschluss = "Bachelor" Then
        gesamtAnzahlNichtRegelstudienzeitBachelor = gesamtAnzahlNichtRegelstudienzeitBachelor + ➥
            anzahlNichtRegelstudienzeit
        anzahlBachelor = anzahlBachelor + 1
    ElseIf abschluss = "Master" Then
        gesamtAnzahlNichtRegelstudienzeitMaster = gesamtAnzahlNichtRegelstudienzeitMaster + ➥
            anzahlNichtRegelstudienzeit
        anzahlMaster = anzahlMaster + 1
    End If
```

Nun beenden wir die Schleife.

```
Next i
```

Nach der Schleife müssen wir die Durchschnitte berechnen. Die durchschnittliche Zuweisung ist ja einfach die Gesamtzuweisung dividiert durch die Anzahl Datensätze.

```
durchschnittlicheZweisung = gesamtZuweisung / anzahlGesamt
```

Um die durchschnittlichen Anzahl Absolventen mit Abschluss Bachelor zu berechnen, müssen wir die Gesamtanzahl Absolventen mit Abschluss Bachelor durch die Anzahl der Datensätze mit Abschluss Bachelor teilen. Doch was passiert, wenn wir keine Studiengänge mit Abschluss Bachelor haben? Dann ist die Anzahl der Datensätze mit Abschluss Bachelor Null. Was bedeutet, wir würden durch Null teilen. Dies geht aber nicht, unser Programm würde abstürzen.In diesem Fall müssen wir also prüfen, ob die Anzahl der Datensätze mit Aufträgen ohne Nachtzuschlag nicht Null ist. Nur in diesem Fall dürfen wir die Division durchführen. Ansonsten setzen wir den Durchschnitt auf 0. Hierfür haben wir in Kap. 10.1 bereits die Funktion *dividiereMitErgebnis0FuerNenner0Double* geschrieben (Beispiel 10.4). Da hier der Datentyp des Zählers *Long* ist, benötigen wir eine weitere Funktion, die bis auf den geänderten Datentypen des Zähler der Funktion aus Beispiel 10.4 entspricht. Wir rufen diese einfach auf:

```
durchschnittlicheAnzahlNichtRegelstudienzeitBachelor = dividiereMitErgebnis0FuerNenner0Long(➥
    gesamtAnzahlNichtRegelstudienzeitBachelor, anzahlBachelor)
```

Das gleiche Verfahren wenden wir natürlich auch bei den Masterabschlüssen an.

```
durchschnittlicheAnzahlNichtRegelstudienzeitMaster = dividiereMitErgebnis0FuerNenner0Long(➥
    gesamtAnzahlNichtRegelstudienzeitMaster, anzahlMaster)
```

Diese Vorgehensweise müssen wir immer anwenden, wenn wir durch eine Anzahl, die wir durch eine Einschränkung ermittelt haben, teilen wollen. Denn dann kann es immer vorkommen, dass wir keine Datensätze haben, für die die Einschränkung gilt. Dann müssen wir ein Teilen durch Null verhindern.
Zum Schluss schreiben wir unsere Ergebnisse in das Tabellenblatt 2:

```
Sheets(TABELLENBLATT_STATISTIK).Cells(1, 1) = "Gesamtanzahl Datensätze"
Sheets(TABELLENBLATT_STATISTIK).Cells(2, 1) = "Gesamtzuweisung"
Sheets(TABELLENBLATT_STATISTIK).Cells(3, 1) = "Durchschnittszuweisung"
Sheets(TABELLENBLATT_STATISTIK).Cells(4, 1) = "Gesamtanzahl Absolventen nicht in Regelstudienzeit➥
    "
Sheets(TABELLENBLATT_STATISTIK).Cells(5, 1) = "Gesamtanzahl Bachelorabsolventen nicht in ➥
    Regelstudienzeit"
Sheets(TABELLENBLATT_STATISTIK).Cells(6, 1) = "Durchschnittliche Anzahl Bachelorabsolventen nicht➥
     in Regelstudienzeit"
Sheets(TABELLENBLATT_STATISTIK).Cells(7, 1) = "Gesamtanzahl Masterabsolventen nicht in ➥
    Regelstudienzeit"
Sheets(TABELLENBLATT_STATISTIK).Cells(8, 1) = "Durchschnittliche Anzahl Masterabsolventen nicht ➥
    in Regelstudienzeit"
Sheets(TABELLENBLATT_STATISTIK).Cells(9, 1) = "Maximale Anzahl Absolventen nicht in ➥
    Regelstudienzeit"
Sheets(TABELLENBLATT_STATISTIK).Cells(10, 1) = "Minimale Anzahl Absolventen nicht in ➥
    Regelstudienzeit"

Sheets(TABELLENBLATT_STATISTIK).Cells(1, 2) = anzahlGesamt
Sheets(TABELLENBLATT_STATISTIK).Cells(2, 2) = gesamtZuweisung
Sheets(TABELLENBLATT_STATISTIK).Cells(3, 2) = durchschnittlicheZweisung
Sheets(TABELLENBLATT_STATISTIK).Cells(4, 2) = gesamtAnzahlNichtRegelstudienzeit
Sheets(TABELLENBLATT_STATISTIK).Cells(5, 2) = gesamtAnzahlNichtRegelstudienzeitBachelor
Sheets(TABELLENBLATT_STATISTIK).Cells(6, 2) = ➥
    durchschnittlicheAnzahlNichtRegelstudienzeitBachelor
Sheets(TABELLENBLATT_STATISTIK).Cells(7, 2) = gesamtAnzahlNichtRegelstudienzeitMaster
Sheets(TABELLENBLATT_STATISTIK).Cells(8, 2) = durchschnittlicheAnzahlNichtRegelstudienzeitMaster
Sheets(TABELLENBLATT_STATISTIK).Cells(9, 2) = maxAnzahlNichtRegelstudienzeit
Sheets(TABELLENBLATT_STATISTIK).Cells(10, 2) = minAnzahlNichtRegelstudienzeit
```

Kapitel 11

Weitere einfache Praxisbeispiele

11.1 Aufträge nach Status farblich darstellen - Lieferantenbewertung

11.1.1 Vorüberlegungen

Sie sind bei einer Ratingagentur beschäftigt und sind Projektmanager für Lieferantenbewertungen. Einige Unternehmen haben der Sie beschäftigenden Ratingagentur den Auftrag gegeben, die Lieferanten dieser Unternehmen zu bewerten. Dies können bis zu mehreren 1000 Lieferanten sein. Jede einzelne Lieferantenbewertung ist ein Auftrag und jeder Auftrag hat einen Status. Die Aufträge werden innerhalb einer Software-Anwendung Ihres Arbeitgebers verwaltet. Sie können aus dieser Anwendung eine csv-Datei[1] exportieren.
Ein Beispiel des möglichen Inhalts einer dieser csv-Dateien ist folgend dargestellt:

```
"AuftraggeberKundenNr";"Name";"Straße";"PLZ";"Ort";"Laendercode";"Status";"StatusNr"
11;"Testunternehmen1";"Teststraße 5";44801;"Bochum";"de";"erledigt";1
3;"Testunternehmen2";"Teststraße 6";44801;"Bochum";"de";"angerufen";2
4;"Testunternehmen3";"Teststraße 7";44801;"Bochum";"de";"Recherche erforderlich";3
5;"Testunternehmen4";"Teststraße 8";44801;"Bochum";"de";"erledigt";1
```

Je nach Auftraggeber variiert der Inhalt der csv-Datei. Auftraggeber ohne Lieferanten im Ausland möchten z.B. keinen Ländercode erhalten. Die Spalte mit der StatusNr ist allerdings immer die letzte. Sie müssen diese csv-Datei in die Tabellenkalkulation einlesen. Zum Einen müssen Sie sich selber einen Überblick über die Aufträge verschaffen, zum Anderen müssen Sie einmal im Monat die Tabellenkalkulationsdateien an Ihren Vorgesetzten und an den jeweiligen Auftraggeber weiterleiten. Dabei sollen aber die Tabellenzeilen, abhängig vom Status, unterschiedlich gefärbt sein. Abb. 11.1 zeigt ein Beispiel:

	A	B	C	D	E	F	G	H
1	AuftraggeberKundenNr	Name	Straße	PLZ	Ort	Laendercode	Status	StatusNr
2	11	Testunternehmen1	Teststraße 5	44801	Bochum	de	erledigt	1
3	3	Testunternehmen2	Teststraße 6	44801	Bochum	de	angerufen	2
4	4	Testunternehmen3	Teststraße 7	44801	Bochum	de	Recherche erforderlich	3
5	5	Testunternehmen4	Teststraße 8	44801	Bochum	de	erledigt	1
6								
7								

Abbildung 11.1
Andersfarbige Zeilen je nach Status

Eine csv-Datei in eine Tabellenkalkulation einzulesen ist kein Problem. In Excel geht das über *Daten* ⇨ *externe Daten importieren* und dann den selbsterklärenden Dialogen folgen.
Das Einfärben der Zeilen läßt sich sicher auch ohne VBA erledigen. Man kann für jede Zeile einzeln Hintergrundfarben einstellen[2], man kann aber auch mit bedingten Formatierungen arbeiten. Hier hat man allerdings die Schwierigkeit, dass man nicht weiß, wie viel Zeilen und wie viel Spalten in der csv-Datei sind, so dass man jedes Mal mit Hand nacharbeiten muss. Man kann aber auch einfach eine Ereignisprozedur schreiben, die dies macht.

[1] csv=comma separated values
[2] nicht wirklich angenehm bei mehreren tausend Zeilen

Mit Ihrem bisherigen Wissen ist dies wirklich ganz einfach:

- Wir ermitteln die letzte besetzte Zeile. Dann wissen wir, bis zu welcher Zeile die Färbeaktion durchgeführt werden muss. Hierzu benutzen wir die Funktion *ermittleLetzteBesetzteZeileInSpalte* aus Kap. 9.
- Wir ermitteln die letzte besetzte Spalte in der ersten Zeile. Das ist die Spalte, in der die StatusNr eingetragen ist. Auch dies können wir bereits. Dazu schrieben wir die Funktion *ermittleLetzteBesetzteSpalteInZeile* ebenfalls in Kap. 9.
- Wir müssen eine Funktion schreiben, die in Abhängigkeit von der StatusNr die zugehörige Farbe ermittelt. Dies ist im Wesentlichen eine Mehrfachauswahl, wie sie in Kap. 5 besprochen wurde.
- Dann müssen wir nur noch in einer Schleife über die Zeilen laufen und die Zeilen färben.

Wie ermittelt man nun die Farbe? Um Zellen Farben zuzuweisen, müssen die Werte der Farben in einem bestimmten Format vorliegen. Dieses Format wird mit der VBA-Funktion *rgb* erzeugt. Diese erwartet als Übergabeparameter die Rot-, Grün- und Blau-Werte der gewünschten Farbe. Also bestimmen wir mit der Funktion *ermittleFarbeAusStatusNr* in Abhängigkeit von der StatusNr die Farbe. Betrachten wir nun die Funktion:

Beispiel 11.1 *Farbe aus Statusnummer ermitteln*

```
'Dateiname: zeilennachStatusFaerben.xls
Function ermittleFarbeAusStatusNr(statusNr As Long) As Long
    If statusNr = 1 Then
        ermittleFarbeAusStatusNr = RGB(0, 255, 0)
    ElseIf statusNr = 2 Then
        ermittleFarbeAusStatusNr = RGB(255, 0, 0)
    Else
        ermittleFarbeAusStatusNr = RGB(0, 0, 255)
    End If
End Function
```

Die der Funktion *rgb* übergebenen Farbwerte dürfen zwischen 0 und 255 variieren. StatusNr 1 führt also zu grün, StatusNr 2 zu rot, alles andere zu blau. Unsere neue Funktion ist auch leicht erweiter- oder änderbar. Kommt ein neuer Status mit einer eigenen Farbe hinzu, fügen wir einfach einen neuen *elseif*-Zweig ein. Wollen wir die Farbe eines Status ändern, so gehen wir in den entsprechenden *elseif*-Zweig und führen die Änderung durch.
Unter http://de.selfhtml.org/helferlein/farben.htm finden Sie übrigens eine Anwendung, mit der Sie die rgb-Entsprechung von Farben leicht finden können.
Um den nun folgenden Code zu vereinfachen, gehen wir davon aus, dass Sie die csv-Datei in das erste Tabellenblatt importieren. Im zweiten Tabellenblatt erstellen wir eine Schaltfläche, die wir mit einer Ereignisprozedur verbinden. Die Ereignisprozedur heißt Excel *zeilenFaerben_click*.

11.1.2 Realisierung

Wir zeigen Ihnen direkt die Realisierung. Danach gehen wir den Code wie immer Zeile für Zeile durch:

Beispiel 11.2 *Zellenfarben in Abhängigkeit von einem Auftragsstatus*

```
'Dateiname: zeilennachStatusFaerben.xls
Sub zeilenFaerben_Click()
    Const TABELLENBLATT As Long = 1
    Const KUNDENNUMMER_SPALTE As Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Dim statusNrSpalte As Long
    Dim letzteBesetzteZeile As Long
    Dim i As Long
    Dim j As Long
    Dim statusNr As Long
    Dim farbe As Long

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(KUNDENNUMMER_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
```

```
    statusNrSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1, ➥
        KUNDENNUMMER_SPALTE, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        statusNr = Sheets(TABELLENBLATT).Cells(i, statusNrSpalte)
        farbe = ermittleFarbeAusStatusNr(statusNr)
        For j = 1 To statusNrSpalte
            Sheets(TABELLENBLATT).Cells(i, j).Interior.Color = farbe
        Next j
    Next i
End Sub
```

Die Zeilen

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(KUNDENNUMMER_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
statusNrSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN -1, ➥
    KUNDENNUMMER_SPALTE, TABELLENBLATT)
```

ermitteln die Spalte mit der StatusNr, dies ist die letzte besetzte Spalte einer Zeile, sowie die letzte besetzte Zeile. Die hier benutzten Funktionen kennen Sie bereits, daher werden wir diese nicht weiter kommentieren.

```
For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
    statusNr = Sheets(TABELLENBLATT).Cells(i, statusNrSpalte)
    farbe = ermittleFarbeAusStatusNr(statusNr)
```

Hier startet eine Schleife über alle besetzten Zeilen, somit über alle Aufträge. Die *StatusNr* wird aus Ihrer Zelle ausgelesen. Mit der Funktion *ermittleFarbeAusStatusNr(statusNr)* wird die zur *StatusNr* korrespondierende Farbe ermittelt. Dann startet eine Schleife über die besetzten Spalten der jeweiligen Zeile:

```
For j = 1 To statusNrSpalte
    Sheets(TABELLENBLATT).Cells(i, j).Interior.Color = farbe
Next j
```

Jeder Zelle wird als Hintergrundfarbe nun die ermittelte Farbe zugewiesen:

```
Sheets(TABELLENBLATT).Cells(i, j).Interior.Color = farbe
```

Nun wissen Sie also, wie man in Excel VBA Zellen Hintergrundfarben zuordnet.

11.1.3 Realisierung mit den VBA-Farbkonstanten

Zusätzlich zu der Möglichkeit Farben über die VBA-interne Funktion *RGB* zu bestimmen, stellt VBA acht Farben auf Konstanten zur Verfügung. Diese Konsatnten sind: *vbBlack, vbBlue, vbCyan, vbGreen, vbMagenta, vbRed, vbWhite, vbYellow*. Die Funktion aus Beispiel 11.1 können wir daher auch so schreiben:

Beispiel 11.3 *Farbe aus Statusnummer ermitteln unter Benutzung der VBA-Farbkonstanten*

```
'Dateiname: zeilennachStatusFaerben.xls
Function ermittleFarbeAusStatusNrMitKonstanten(statusNr As Long) As Long
    If statusNr = 1 Then
        ermittleFarbeAusStatusNrMitKonstanten = vbGreen
    ElseIf statusNr = 2 Then
        ermittleFarbeAusStatusNrMitKonstanten = vbRed
    Else
        ermittleFarbeAusStatusNrMitKonstanten = vbBlue
    End If
End Function
```

11.1.4 Entfärben: Zurücksetzen auf den Excel-Defaultzustand

Um Farben wieder zu entfernen, gibt es in VBA die Konstante *xlNone*.

Beispiel 11.4 *Zellenfarben auf den Defaultwert Zurücksetzen*

```
'Dateiname: zeilennachStatusFaerben.xls{beiFarbeAusStatus}
Sub zeilenEntFaerben_Click()
    Const TABELLENBLATT As Long = 1
    Const KUNDENNUMMER_SPALTE As Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Dim letzteBestzteSpalte As Long
    Dim letzteBesetzteZeile As Long
    Dim i As Long
    Dim j As Long

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(KUNDENNUMMER_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
    letzteBestzteSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1, ➥
        KUNDENNUMMER_SPALTE, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        For j = 1 To letzteBestzteSpalte
            Sheets(1).Cells(i, j).Interior.Color = xlNone
        Next j
    Next i
End Sub
```

Dieses Programm müssten Sie sofort verstehen. Im Vergleich zu Beispiel 11.2 entfällt die Ermittlung der Farbe. Die ist ja bekannt, wir wollen den auf dser VBA-Konstanten *xlNone* liegenden Defaultfarbwert benutzen. Ansonsten stimmt der Code mit 11.2 überein.
Wir könnten uns die Frage stellen, warum wir nicht die Farbe Weiß (*vbWhite*) benutzen. Die Antwort ist: es gibt erstaunlicherweise einen Unterschied. Im Ursprungszustand sind um die Excel-Zellen dünne hellgraue Rahmen zu sehen. Durch das Färben werden diese entfernt. Eine weiß gefärbte Zelle ist also eine Zelle mit weißem Hintergrund ohne sichtbare Zellbegrenzung. Ein Setzen der Zellfarbe auf *xlNone* hingegen stellt den Ursprungszustand wieder her.

11.2 Aufträge nach Vorgabe farblich darstellen - Lieferantenbewertung

11.2.1 Vorüberlegungen - Stringfunktionen

Wir wandeln die Aufgabenstellung aus Kapitel Kap. 11.1 leicht ab. Die Farben der einzelnen Statusausprägungen sinbeiFarbeAusStatusd auftraggeberspezifisch. Sie werden in einer zentralen Datenbank vorgehalten. Die csv-Datei, die Sie aus Ihrer Anwendung exportieren können, enthält nun anstelle der StatusNr den Farbcode. Sie hat nun die im folgenden dargestellte Form:

```
"AuftraggeberKundenNr";"Name";"Straße";"PLZ";"Ort";"Laendercode";"Status";"Farbe"
11;"Testunternehmen1";"Teststraße 5";44801;"Bochum";"de";"erledigt";121212
3;"Testunternehmen2";"Teststraße 6";44801;"Bochum";"de";"angerufen";ff0000
4;"Testunternehmen3";"Teststraße 7";44801;"Bochum";"de";"Recherche erforderlich";0000ff
5;"Testunternehmen4";"Teststraße 8";44801;"Bochum";"de";"erledigt";121212
```

Sie müssen natürlich wieder die entsprechenden Zellen in der vorgegebenen Farbe darstellen. Danach soll allerdings die Spalte mit der Farbinformation gelöscht werden. Erschwerend kommt hinzu, dass die Farben nicht in dezimal sondern in hexadezimal kodiert sind. Dies bedeutet, die ersten beiden Zeichen der Farbe ist die Hexadzimalrepräsentation des Rot-Wertes, die mittleren diejenigen des Grün-Wertes und die letzten beiden dementsprechend die des Blau-Wertes. Diese Kodierung ist nicht ungewöhnlich, viele Grafikprogramme erwarten und erzeugen solch eine Kodierung. Auch die Internet-Browser erwarten für die Farbdarstellung eine hexadezimale Kodierung.
Im ersten Schritt muss also aus der Hexadezimal-Kodierung die Farbkodierung der Tabellenkalkulation abgeleitet werden. Dazu werden wir eine Funktion schreiben. Dann benutzen wir diese Funktion anstelle der Funktion *ermittleFarbeAusStatusNr* aus Kap. 11.1. Der Rest entspricht dem Code aus Kap. 11.1, mit dem einen Unterschied, dass wir die Spalte mit der

Farbinformation hinterher löschen müssen. Dies kann man jetzt natürlich auch manuell machen, insbesondere vor dem Hintergrund, dass Sie sich die Datei vor dem Weiterleiten eh anschauen müssen. Da es sich aber nur um eine Zeile Code handelt, können wir es natürlich auch in unser Programm mit aufnehmen.
Wir beginnen mit der Funktion zur Ermittlung der Farbe.

Beispiel 11.5 *Ermittlung der tabellenkalkulationsspezifischen Farbkodierung aus hexaddezimalem RGB*

```
'Dateiname: zeilenNachHexCodeFaerben.xls
Function ermittleFarbeAusHexCode(rgbHexCode As String) As Long
    Dim codeRot As Long
    Dim codeGruen As Long
    Dim codeBlau As Long
    Dim hexCode As String
    rgbHexCode = Trim(rgbHexCode)
    hexCode = Left(rgbHexCode, 2)
    codeRot = Val("&H" & hexCode)
    hexCode = Mid(rgbHexCode, 3, 2)
    codeGruen = Val("&H" & hexCode)
    hexCode = Right(rgbHexCode, 2)
    codeBlau = Val("&H" & hexCode)
    ermittleFarbeAusHexCode = RGB(codeRot, codeGruen, codeBlau)
End Function
```

Kernstück dieser Funktion sind die VBA-internen Stringfunktionen. Wir benutzen hier deren drei:

- *trim*: Entfernt Leerzeichen (Blancs) am Anfang und am Ende einer Stringvariablen.
- *left(string, anzahl)*: Extrahiert einen Teilstring bestehend aus *anzahl* Zeichen von links beginnend.
- *mid(string, startzeichen, anzahl)*: Extrahiert einen Teilstring aus der Mitte bestehend aus *anzahl* Zeichen von *startzeichen* beginnend.
- *right*: Extrahiert einen Teilstring bestehend aus *anzahl* Zeichen von rechts beginnend.

Wir veranschaulichen uns dies an einem einfachen Beispiel:

Beispiel 11.6 *Die Stringfunktionen*

```
'Dateiname: stringfunktionen.xls
Sub stringfunktionen()
    Dim s1 As String
    Dim s2 As String
    Dim s3 As String
    Dim s4 As String
    Dim s5 As String
    Dim l As Long

    s1 = " abcdefg "
    l = Len(s1) ' l hat jetzt den Wert 9 wegen der Blancs am Anfang und am Ende
    MsgBox (l)
    s1 = Trim(s1) 'Die Leerzeichen werden entfernt
    l = Len(s1) ' l hat jetzt den Wert 7,  weil die Leerzeichen weg sind
    MsgBox (l)
    s2 = Left(s1, 2) ' s2 ist ab
    MsgBox (s2)
    s3 = Mid(s1, 3, 2) 's2 ist cd
    MsgBox (s3)
    s4 = Right(s1, 3) ' s3 ist efg
    MsgBox (s4)
    l = InStr(s1, "de") 'l ist jetzt 4
    MsgBox (l)
    s5 = Replace(s1, "a", "z") ' s5 ist zbcdefg
    MsgBox (s5)
End Sub
```

Hier wurden zusätzlich noch die Stringfunktionen *len*, *InStr* und *replace* eingeführt. *len* ermittelt die Länge (Anzahl Zeichen) eines Strings. *InStr* ermittelt, an welcher Stelle ein Teilstring in einem String beginnt. In unserem Beipiel beginnt der Teilstring *de* an der vierten Stelle des Strings *abcdefg*. Kommt ein Teilstring im String nicht vor, gibt *InStr* 0 zurück. *replace* hingegen ersetzt in einem String eine Zeichenkette durch eine andere. Beispiel 11.6 sollte wegen der Kommentare selbsterklärend sein, so dass sich eine weitere Diskussion erübrigt. Kehren wir also zurück zu Beispiel 11.5. Durch die Zeile

```
rgbHexCode=trim(rgbHexCode)
```

werden zunächst irrtümlich eingegebene Leerzeichen am Anfang und am Ende der Variablen entfernt.

```
hexCode=left(rgbHexCode, 2)
```

ermittelt nun die ersten beiden Ziffern der hexadezimalen Farbkodierung. Dies ist der Farbkode der Rotkomponente der Farbe.

```
codeRot=Val("&H" & hexCode)
```

Val ist eine VBA-interne Funktion zur Zahlenkonvertierung. Beginnt der *Val* übergebene String mit *&H*, so interpretiert *Val* diesen String als eine hexadezimal kodierte Zahl und gibt den Dezimalwert dieser Zahl zurück. Das ist genau das was wir wollen. Die nächsten Zeilen

```
hexCode=mid(rgbHexCode, 3, 2)
codeGruen=Val("&H" & hexCode)
hexCode=right(rgbHexCode, 2)
codeBlau=Val("&H" & hexCode)
```

müssten nun leicht verständlich sein. Zunächst werden die beiden mittleren Zeichen der in hexadezimal kodierten Farbe ermittelt (der Grünwert) und dann in dezimal umgewandelt anschließend die letzten beiden. Durch die Zeile

```
ermittleFarbeAusHexCode=rgb(codeRot, codeGruen, codeBlau)
```

wird aus diesen Zwischenergebnissen der tabellenkalkulationsspezifische Farbkode erzeugt und über den Funktionsnamen zurückgegeben.

11.2.2 Realisierung

Wir zeigen Ihnen nun die Realisierung. Die Änderungen gegenüber Beipiel 11.2 halten sich durchaus in Grenzen. Anstelle der StatusNr wird die Farbe aus der Tabelle gelesen und anstelle der Funktion *ermittleFarbeAusStatusNr* wird die Funktion *ermittleFarbeAusHexCode* aufgerufen.

Beispiel 11.7 *Zellenfarben in Abhängigkeit von einem Auftragsstatus*

```
'Dateiname: zeilenNachHexCodeFaerben.xls
Sub zeilenFaerben_Click()
    Const TABELLENBLATT As Long = 1
    Const KUNDENNUMMER_SPALTE As Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2

    Dim farbSpalte As Long
    Dim letzteBesetzteZeile As Long
    Dim i As Long
    Dim j As Long
    Dim farbString As String
    Dim farbe As Long

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(KUNDENNUMMER_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
    farbSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1, ➥
        KUNDENNUMMER_SPALTE, TABELLENBLATT)
```

```
    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        farbString = Sheets(TABELLENBLATT).Cells(i, farbSpalte)
        farbe = ermittleFarbeAusHexCode(farbString)
        For j = 1 To farbSpalte
            Sheets(TABELLENBLATT).Cells(i, j).Interior.Color = farbe
        Next j
    Next i
    Sheets(TABELLENBLATT).Columns(farbSpalte).Delete
End Sub
```

Durch die Zeile

```
Sheets(TABELLENBLATT).Columns(farbSpalte).Delete
```

wird die Spalte mit den Farbkodierungen gelöscht. Daher wissen Sie nun auch, wie man in Excel eine Spalte löscht.

11.3 Aufträge einlesen - Lieferantenbewertung

11.3.1 Vorüberlegungen - in Datei schreiben

Die Aufträge werden von den Auftraggebern der Ratingagentur als Excel-Tabellen geliefert. Ein Beispiel zeigt Abb. 11.2.

	A	B	C	D	E	F
1	KundenNr	Name	Straße	PLZ	Ort	Laendercode
2	11	Testunterneh	Teststraße 5	44801	Bochum	de
3	3	Testunterneh	Teststraße 6	44801	Bochum	de
4	4	Testunterneh	Teststraße 7	44801	Bochum	de
5	5	Testunterneh	Teststraße 8	44801	Bochum	de
6						
7						

Abbildung 11.2
Aufträge in einer Excel-Tabelle

Die einzelnen Aufträge (Zeilen der Tabelle aus Abb. 11.2) müssen in die Software-Anwendung Ihres Arbeitgebers importiert werden. Dazu steht ein Auftraganlage-Modul zur Verfügung. Eine solche Tabelle, wie Sie sie bekommen, kann aber mehrere tausend Einträge beinhalten. Da ist die manuelle Eingabe sicher kein Spaß. Alternativ steht eine csv-Schnittstelle zur Verfügung. Die csv-Dateien müssen dann wie folgt aussehen:

```
"11";"Testunternehmen1";"Teststraße 5";"44801";"Bochum";"de"
"3";"Testunternehmen2";"Teststraße 6";"44801";"Bochum";"de"
"4";"Testunternehmen3";"Teststraße 7";"44801";"Bochum";"de"
"5";"Testunternehmen4";"Teststraße 8";"44801";"Bochum";"de"
```

Leider ist die Struktur der Excel-Tabellen von Auftraggeber zu Auftraggeber höchst unterschiedlich. Die Reihenfolge der einzelnen Spalten stimmt nicht überein, so steht die PLZ mal in der zweiten und ein anderes Mal in der vierten Spalte. Darüberhinaus sind ab und zu Felder wie Ansprechpartner oder Telefonnummer enthalten, die in der csv-Schnittstelle nicht vorkommen. Sie könnten nun die Spalten jeweils so umarrangieren, dass sie der oben dargestellten Reihenfolge entsprechen, unnötige Spalten löschen und dann aus der Tabellenkalkulation heraus eine csv-Ausgabe erzeugen.
Diese Vorgehensweise ist zeitaufwändig und kann zu Fehlern führen. Alternativ können Sie ein kleines VBA-Programm schreiben, welches diese Dinge für Sie erledigt. Abb. 11.3 zeigt das Benutzerinterface. In der Spalte B tragen Sie ein, in welcher Spalte der gelieferten Excel-Datei sich welche Information befindet. Das Clicken auf die Schaltfläche erzeugt dann die csv-Datei.
Um diese Problematik zu lösen, müssen Sie zunächst lernen, wie man aus VBA heraus eine Datei öffnet und beschreibt. Beispiel 11.8 erläutert dies:

Beispiel 11.8 *Schreiben aus VBA in eine Datei*

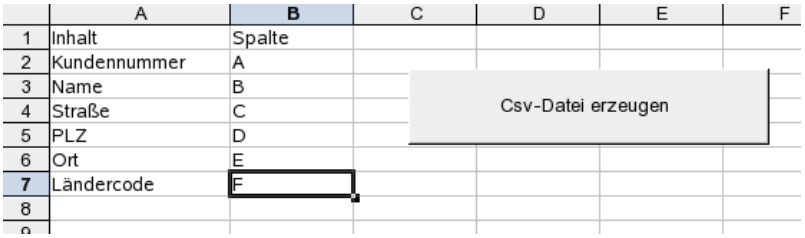


Abbildung 11.3
Benutzerinterface zum Erzeugen der csv-Datei

```
'Dateiname: schreibenInDatei.xls
Sub inDateiSchreiben()
    Dim fp As Long
    Dim ausgabeDatei As String
    Dim zeilenTrenner As String
    Dim zeile As String

    zeilenTrenner = Chr$(13)
    ausgabeDatei = "datei.txt"
    fp = FreeFile
    Open ausgabeDatei For Output As #fp

    zeile = "Dies ist die erste Zeile" & zeilenTrenner
    Print #fp, zeile
    zeile = "Dies ist die zweite Zeile" & zeilenTrenner
    Print #fp, zeile
    Close #fp
End Sub
```

Die Zeilen

```
ausgabeDatei="datei.txt"
fp = Freefile
Open ausgabeDatei For Output As #fp
```

öffnen eine Datei im Schreibmodus. Mit

```
zeile="Dies ist die erste Zeile" & zeilenTrenner
print #fp, zeile
```

wird ein Text in die Datei geschrieben. Am Ende muss die Datei geschlossen werden:

```
Close #fp
```

In der zu erstellenden csv-Datei müssen alle Werte in Anführungszeichen eingeschlossen sein. In einem Unternehmensnamen können aber selber Anführungszeichen vorkommen. Das ist aber nicht so gut, weil dann die ersten Anführungszeichen des Namens von allen Importprogrammen als Ende der einschließenden Anführungszeichen angesehen werden. Um die Funktionalität von Importprogrammen sicher zu stellen, muss Anführungszeichen im Text ihre Sonderbedeutung (das Einschließen von den Werten der csv-Datei) genommen werden. Dies geschieht, indem jedem Anführungszeichen im Text der Backslash \ vorangestellt wird.
Wie dies geschieht, zeigt Beispiel 11.9:

Beispiel 11.9 *Anführungszeichen durch Backslash Anführungszeichen ersetzen.*

```
'Dateiname: anfuehrungszeichenErsetzen.xls
Sub ersetzeAnfuehrungszeichen()
    Dim zuErsetzen As String
    Dim ersetzer As String
```

```
    Dim s As String
    Dim s1 As String
    zuErsetzen = """"
    ersetzer = "\"""
    s = "dies ist ein ""text"" mit Anführungszeichen"
    s1 = Replace(s, zuErsetzen, ersetzer)
    MsgBox (s & Chr$(13) & s1)
End Sub
```

Ein bischen „strange“ muten die Zeilen

```
zuErsetzen=""""
ersetzer="\"""
```

an. Sie sind aber leicht erklärt. Anführungszeichen haben in VBA die Sonderbedeutung „Zeichenkette beginnen“ bzw. „Zeichenkette beenden“ (oder anders ausgedrückt „Zeichenkette umschließen“). Unser Ziel ist es, Anführungszeichen gegen Backslash Anführungszeichen auszutauschen. Das erste Anführungszeichen in *zuErsetzen* beginnt die Zeichenkette. Die Zeichenkette soll aus nur einem Anführungszeichen bestehen. Ein zweites Anführungszeichen würde die Zeichenkette beenden. Um dies zu verhindern, muss dem zweiten Anführungszeichen seine Sonderbedeutung (Zeichenkette beenden) genommen werden. Einem Zeichen seine Sonderbedeutung nehmen, geschieht in VBA durch Voranstellen des Anführungszeichens ☺. Das erste Anführungszeichen beginnt also die Zeichenkette. Das zweite Anführungszeichen hebt die Sonderbedeutung des dritten Anführungszeichens (Zeichenkette beenden) auf. Das dritte Anführungszeichen ist also der eigentliche Inhalt der Zeichenkette. Das vierte Anführungszeichen schließt die Zeichenkette ☺. Alles klar? Warum die Variable *ersetzer* aussieht, wie sie aussieht, überlegen Sie sich selbst. Abb. 11.4 zeigt, das Beispiel 11.9 wie beschrieben funktioniert.

Abbildung 11.4
Anführungszeichen ersetzen

Abschließend benötigen wir eine weitere Hilfsfunktion: Die Benutzer sollen, wie in Abb. 11.3 dargestellt, eingeben, in welcher Spalte der gelieferten Excel-Datei sich welche Information befindet. Sie werden dort, wie man ebenfalls Abb. 11.3 entnehmen kann, Buchstaben eingeben. Um auf die Inhalte von Zellen zugreifen zu können, benötigen wir aber Zahlen. Das heißt, wir müssen die eingegebenen Buchstaben in die korrespondierende Spaltennummer umwandeln. Beispiel 11.10 zeigt die zugehörige Funktion. Sie ist so einfach, dass wir auf eine Diskussion verzichten.

Beispiel 11.10 *Spaltenbuchstaben in Zahlen umwandeln*

```
'Dateiname: alsCsvSpeichern.xls
Function wandleBuchstabenInZahl(buchstabe As String) As Long
    Select Case buchstabe
        Case "A", "a"
            wandleBuchstabenInZahl = 1
            Exit Function
        Case "B", "b"
            wandleBuchstabenInZahl = 2
            Exit Function
        Case "C", "c"
            wandleBuchstabenInZahl = 3
            Exit Function
        Case "D", "d"
            wandleBuchstabenInZahl = 4
            Exit Function
```

```
        Case "E", "e"
            wandleBuchstabenInZahl = 5
            Exit Function
        Case "F", "f"
            wandleBuchstabenInZahl = 6
            Exit Function
        Case "G", "g"
            wandleBuchstabenInZahl = 7
            Exit Function
        Case "H", "h"
            wandleBuchstabenInZahl = 8
            Exit Function
        Case "I", "i"
            wandleBuchstabenInZahl = 9
            Exit Function
        Case "J", "j"
            wandleBuchstabenInZahl = 10
            Exit Function
        Case "K", "k"
            wandleBuchstabenInZahl = 11
            Exit Function
        Case "L", "l"
            wandleBuchstabenInZahl = 12
            Exit Function
        Case "M", "m"
            wandleBuchstabenInZahl = 13
            Exit Function
        Case "N", "n"
            wandleBuchstabenInZahl = 14
            Exit Function
        Case "O", "o"
            wandleBuchstabenInZahl = 15
            Exit Function
        Case "P", "p"
            wandleBuchstabenInZahl = 16
            Exit Function
        Case "Q", "q"
            wandleBuchstabenInZahl = 17
            Exit Function
        Case "R", "r"
            wandleBuchstabenInZahl = 18
            Exit Function
        Case "S", "s"
            wandleBuchstabenInZahl = 19
            Exit Function
        Case "T", "t"
            wandleBuchstabenInZahl = 20
            Exit Function
        Case "U", "u"
            wandleBuchstabenInZahl = 21
            Exit Function
        Case "V", "v"
            wandleBuchstabenInZahl = 22
            Exit Function
        Case "W", "w"
            wandleBuchstabenInZahl = 23
            Exit Function
        Case "X", "x"
            wandleBuchstabenInZahl = 24
            Exit Function
        Case "Y", "y"
            wandleBuchstabenInZahl = 25
            Exit Function
        Case "Z", "z"
            wandleBuchstabenInZahl = 26
            Exit Function
    End Select
End Function
```

Mit diesem Wissen können wir uns nun die Implementierung der Ereignisprozedur zur Erstellung der csv-Datei anschauen.

11.3.2 Realisierung

Beispiel 11.11 *Eine Csv-Datei schreiben*

```
'Dateiname: alsCsvSpeichern.xls
Sub speichereAlsCsv_Click()
    Dim kundenNrSpalte As String
    Dim nameSpalte As String
    Dim strasseSpalte As String
    Dim plzSpalte As String
    Dim stadtSpalte As String
    Dim laendercodeSpalte As String
    Dim letzteBesetzteZeile As Long

    Dim kundenNrSpalteLong As Long
    Dim nameSpalteLong As Long
    Dim strasseSpalteLong As Long
    Dim plzSpalteLong As Long
    Dim stadtSpalteLong As Long
    Dim laendercodeSpalteLong As Long

    Dim kundenNr As String
    Dim kundenName As String
    Dim strasse As String
    Dim plz As String
    Dim stadt As String
    Dim laendercode As String
    Dim i As Long

    Dim zuErsetzen As String
    Dim ersetzer As String
    Dim csvString As String
    Dim csvTrenner As String
    Dim csvEinschliesser As String
    Dim zeilenTrenner As String
    Dim fp As Long
    Dim ausgabeDatei As String

    Const KUNDENNUMMER_ZEILE As Long = 2
    Const NAME_ZEILE As Long = 3
    Const STRASSE_ZEILE As Long = 4
    Const PLZ_ZEILE As Long = 5
    Const ORT_ZEILE As Long = 6
    Const LAENDERCODE_ZEILE As Long = 7
    Const SPALTEN_SPALTE As Long = 2
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 1
    Const TABELLENBLATT_DATEN As Long = 1
    Const TABELLENBLATT_SPALTENINFOS As Long = 2

    zuErsetzen = """"
    ersetzer = "\"""
    csvTrenner = ";"
    csvEinschliesser = """"
    zeilenTrenner = Chr$(13)
    ausgabeDatei = "auftrag" & Date & ".csv"

    kundenNrSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(KUNDENNUMMER_ZEILE, SPALTEN_SPALTE)
    nameSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(NAME_ZEILE, SPALTEN_SPALTE)
    strasseSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(STRASSE_ZEILE, SPALTEN_SPALTE)
    plzSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(PLZ_ZEILE, SPALTEN_SPALTE)
    stadtSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(ORT_ZEILE, SPALTEN_SPALTE)
    laendercodeSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(LAENDERCODE_ZEILE, ➥
        SPALTEN_SPALTE)
```

```
    kundenNrSpalteLong = wandleBuchstabenInZahl(kundenNrSpalte)
    nameSpalteLong = wandleBuchstabenInZahl(nameSpalte)
    strasseSpalteLong = wandleBuchstabenInZahl(strasseSpalte)
    plzSpalteLong = wandleBuchstabenInZahl(plzSpalte)
    stadtSpalteLong = wandleBuchstabenInZahl(stadtSpalte)
    laendercodeSpalteLong = wandleBuchstabenInZahl(laendercodeSpalte)

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(kundenNrSpalteLong, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT_DATEN)

    fp = FreeFile
    Open ausgabeDatei For Output As #fp

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        kundenNr = Sheets(TABELLENBLATT_DATEN).Cells(i, kundenNrSpalteLong)
        kundenName = Sheets(TABELLENBLATT_DATEN).Cells(i, nameSpalteLong)
        strasse = Sheets(TABELLENBLATT_DATEN).Cells(i, strasseSpalteLong)
        plz = Sheets(TABELLENBLATT_DATEN).Cells(i, plzSpalteLong)
        stadt = Sheets(TABELLENBLATT_DATEN).Cells(i, stadtSpalteLong)
        laendercode = Sheets(TABELLENBLATT_DATEN).Cells(i, laendercodeSpalteLong)

        kundenName = Replace(kundenName, zuErsetzen, ersetzer)
        strasse = Replace(strasse, zuErsetzen, ersetzer)
        stadt = Replace(stadt, zuErsetzen, ersetzer)

        csvString = csvEinschliesser & kundenNr & csvEinschliesser & csvTrenner & _
            csvEinschliesser & kundenName & csvEinschliesser & csvTrenner & _
            csvEinschliesser & strasse & csvEinschliesser & csvTrenner & _
            csvEinschliesser & plz & csvEinschliesser & csvTrenner & _
            csvEinschliesser & stadt & csvEinschliesser & csvTrenner & _
            csvEinschliesser & laendercode & csvEinschliesser & zeilenTrenner
        Print #fp, csvString
    Next i
    Close #fp
    MsgBox ("csv-Datei erzeugt")
End Sub
```

Der Code ist eigentlich nicht weiter schwierig: Zunächst kommt die Deklaration der benötigten Variablen und Konstanten. Durch

```
kundenNrSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(KUNDENNUMMER_ZEILE, SPALTEN_SPALTE)
nameSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(NAME_ZEILE, SPALTEN_SPALTE)
strasseSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(STRASSE_ZEILE, SPALTEN_SPALTE)
plzSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(PLZ_ZEILE, SPALTEN_SPALTE)
stadtSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(ORT_ZEILE, SPALTEN_SPALTE)
laendercodeSpalte = Sheets(TABELLENBLATT_SPALTENINFOS).Cells(LAENDERCODE_ZEILE, SPALTEN_SPALTE)
```

wird die Information, welche Spalte welchen Inhalt besitzt, eingelesen. Die Zeilen

```
kundenNrSpalteLong = wandleBuchstabenInZahl(kundenNrSpalte)
nameSpalteLong = wandleBuchstabenInZahl(nameSpalte)
strasseSpalteLong = wandleBuchstabenInZahl(strasseSpalte)
plzSpalteLong = wandleBuchstabenInZahl(plzSpalte)
stadtSpalteLong = wandleBuchstabenInZahl(stadtSpalte)
laendercodeSpalteLong = wandleBuchstabenInZahl(laendercodeSpalte)
```

wandeln diese Information in Zahlen um. Dann wird die Ausgabedatei geöffnet:

```
fp = Freefile
Open ausgabeDatei For Output As #fp
```

Nun beginnt eine Schleife über die Zeilen der Tabelle:

```
for i= 1 to letzteBesetzteZeile
```

Die für die csv-datei benötigten Informationen der Zeile werden ausgelesen:

```
kundenNr = Sheets(TABELLENBLATT_DATEN).Cells(i, kundenNrSpalteLong)
kundenName = Sheets(TABELLENBLATT_DATEN).Cells(i, nameSpalteLong)
strasse = Sheets(TABELLENBLATT_DATEN).Cells(i, strasseSpalteLong)
plz = Sheets(TABELLENBLATT_DATEN).Cells(i, plzSpalteLong)
stadt = Sheets(TABELLENBLATT_DATEN).Cells(i, stadtSpalteLong)
laendercode = Sheets(TABELLENBLATT_DATEN).Cells(i, laendercodeSpalteLong)
```

Eventuell vorhandene Anführungszeichen in Name, Straße und Stadt werden durch Backslash Anführungszeichen ersetzt.

```
kundenName=Replace(kundenName, zuErsetzen, ersetzer)
strasse=Replace(strasse, zuErsetzen, ersetzer)
stadt=Replace(stadt, zuErsetzen, ersetzer)
```

Die in die csv-Datei zu schreibende Zeile wird erzeugt

```
csvString=csvEinschliesser & kundenNr & csvEinschliesser & csvTrenner & _
            csvEinschliesser & kundenName & csvEinschliesser & csvTrenner & _
            csvEinschliesser & strasse & csvEinschliesser & csvTrenner & _
            csvEinschliesser & plz & csvEinschliesser & csvTrenner & _
            csvEinschliesser & stadt & csvEinschliesser & csvTrenner & _
            csvEinschliesser & laendercode & csvEinschliesser & zeilenTrenner
```

und in die csv-Datei geschrieben:

```
print #fp, csvString
```

Nach der Schleife wird die csv-Datei geschlossen:

```
close #fp
```

Kapitel 12

Weitere Beispiele zum Formatieren von Zellen

12.1 Grundlegende Formatierungen

Neben der in Kap. 11.1 beschriebenen Möglichkeit, Zellen aus einem VBA-Programm Hintergrundfarben zuzuordnen, kann man natürlich jede weitere Formatierung einer Zelle aus VBA heraus ändern. Beispiel 12.1 gibt uns davon eine Vorstellung.

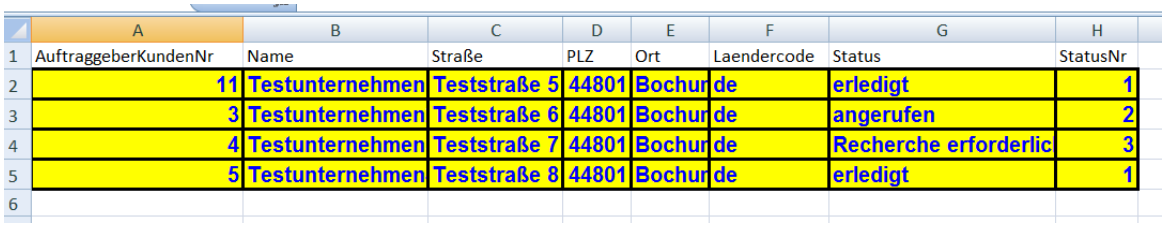
Beispiel 12.1 *Grundlegende Formatierungsmöglichkeiten für Zellen*

```
Sub zeilenFormatieren_Click()
    Const TABELLENBLATT As Long = 1
    Const KUNDENNUMMER_SPALTE As Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Dim letzteBesetzteSpalte As Long
    Dim letzteBesetzteZeile As Long
    Dim i As Long
    Dim j As Long

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(KUNDENNUMMER_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
    letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1,➥
         KUNDENNUMMER_SPALTE, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
      For j = 1 To letzteBesetzteSpalte
          Sheets(TABELLENBLATT).Cells(i, j).Interior.Color = vbYellow
          Sheets(TABELLENBLATT).Cells(i, j).Font.Color = vbBlue
          Sheets(TABELLENBLATT).Cells(i, j).Font.Bold = True
          Sheets(TABELLENBLATT).Cells(i, j).Font.Italic = False
          Sheets(TABELLENBLATT).Cells(i, j).Font.Name = "Arial"
          Sheets(TABELLENBLATT).Cells(i, j).Font.Size = 12
          Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeLeft).LineStyle = xlContinuous
          Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeLeft).Weight = xlThick
          Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeRight).LineStyle = xlContinuous
          Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeRight).Weight = xlThick
          Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeTop).LineStyle = xlContinuous
          Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeTop).Weight = xlThick
          Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeBottom).LineStyle = xlContinuous
          Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeBottom).Weight = xlThick
      Next j
    Next i
End Sub
```

Beispiel 12.1 führt zu dem in Abb. 12.1 dargestelltem Ergebnis. Neu sind die Zeilen:



	A	B	C	D	E	F	G	H
1	AuftraggeberKundenNr	Name	Straße	PLZ	Ort	Laendercode	Status	StatusNr
2	11	Testunternehmen	Teststraße 5	44801	Bochur	de	erledigt	1
3	3	Testunternehmen	Teststraße 6	44801	Bochur	de	angerufen	2
4	4	Testunternehmen	Teststraße 7	44801	Bochur	de	Recherche erforderlic	3
5	5	Testunternehmen	Teststraße 8	44801	Bochur	de	erledigt	1
6								

Abbildung 12.1
Formatierungen

```
Sheets(TABELLENBLATT).Cells(i, j).Font.Color = vbBlue
Sheets(TABELLENBLATT).Cells(i, j).Font.Bold = True
Sheets(TABELLENBLATT).Cells(i, j).Font.Italic = False
Sheets(TABELLENBLATT).Cells(i, j).Font.Name = "Arial"
Sheets(TABELLENBLATT).Cells(i, j).Font.Size = 12
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeLeft).LineStyle = xlContinuous
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeLeft).Weight = xlThick
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeRight).LineStyle = xlContinuous
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeRight).Weight = xlThick
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeTop).LineStyle = xlContinuous
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeTop).Weight = xlThick
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeBottom).LineStyle = xlContinuous
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeBottom).Weight = xlThick
```

Hier wird die Schriftfarbe auf „Blau“, der Font auf „Arial“ und die Schriftgröße auf 12 gesetzt. Die Schrift wird in Fett dargestellt:

```
Sheets(TABELLENBLATT).Cells(i, j).Font.Bold = True
```

Eventuell vorhandene Kursivschrift wird entfernt:

```
Sheets(TABELLENBLATT).Cells(i, j).Font.Italic = False
```

Die Zellen bekommen an allen vier Kanten (*Borders(xlEdgeLeft)*, *Borders(xlEdgeRight)*, *Borders(xlEdgeTop)* und *Borders(xlEdgeBottom)*) einen durchgezogenen Rahmen (*xlContinuous*). Die Rahmen werden fett dargestellt (*xlThick*). Weitere in VBA erlaubte Werte für den Linienstil sind:

- xlContinuous,
- xlDot
- xlDashDotDot,
- xlDash,
- xlSlantDashDot,
- xlDouble und
- xlNone.

Für das Gewicht des Rahmens gibt es die Konstanten:

- xlHairline,
- xlThin,
- xlMedium und
- xlThick.

Beachten Sie: Wenn Sie als Linienstil *xlNone* gewählt haben, dann darf das Rahmengewicht nicht mehr zugewiesen werden. Der folgende Code

```
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeLeft).LineStyle = xlNone
Sheets(TABELLENBLATT).Cells(i, j).Borders(xlEdgeLeft).Weight = xlThick
```

führt also zu einem Fehler. Dies ist aber eigentlich auch logisch, denn, wenn man keinen Rahmen hat, kann man dem nicht vorhandenen Rahmen schlecht ein Gewicht zuweisen.

12.2 Das Gewinnbeispiel

Schreiben Sie ein Programm, das die in Abb. 12.2 gezeigten Formatierungen durchführt:

- Zeilen mit der Softwarekategorie „Betriebssysteme“ werden gelb gefärbt.
- Zeilen mit der Softwarekategorie „Office“ werden pink gefärbt (Rotanteil 255, Grünanteil 50, Blauanteil 255).
- Alle anderen Zeilen sollen nicht gefärbt werden.

Dies gilt nicht für Verkäufe, die mittels CD-Versand verschickt werden. Diese sollen unabhängig von der Softwarekategorie rot gefärbt werden.
Bei Datensätzen (Excel-Zeilen) mit einem Gesamteinkaufspreis von mehr als 100 Euro soll der Text in „Fett“ dargestellt werden.

	A	B	C	D	E	F	G	H
1	Kunde	Produkt	Anzahl	Einkaufspreis	Spoftwarekategorie	Versandart	Gewinn	
2	Schmid	PDF Reader	1	26,00 €	Utilities	Download	2,08 €	
3	**Meier**	**Kalkulation**	**5**	**82,95 €**	**Office**	**Download**	**20,74 €**	
4	**Seran**	**Windows 4711**	**2**	**112,00 €**	**Betriebssysteme**	**Download**	**6,72 €**	
5	**Müller**	**Windows 0815**	**10**	**87,00 €**	**Betriebssysteme**	**Download**	**26,10 €**	
6	Schmidt	Writer	1	58,45 €	Office	CD-Versand	5,22 €	
7								
8								

Abbildung 12.2
Formatierungen: Gewinnbeispiel

Wie wir grundsätzlich vorgehen werden, wissen wir aus Kap. 11.1:

- Wir ermitteln die letzte besetzte Zeile. Dann wissen wir, bis zu welcher Zeile die Formatierungen zugewiesen werden müssen. Hierzu benutzen wir die Funktion *ermittleLetzteBesetzteZeileInSpalte* aus Kap. 9.
- Wir ermitteln die letzte besetzte Spalte. Auch dies können wir bereits. Dazu schrieben wir die Funktion *ermittleLetzteBesetzteSpalteInZeile* ebenfalls in Kap. 9.
- Wir laufen in einer Schleife über die Zeilen.
- In jeder Zeile
  - Ermitteln wir die Farbe.
  - Stellen wir fest, ob die Darstellung in „Fett“ erfolgen muss.
  - Dann müssen wir nur noch in einer Schleife über die Spalten laufen und die Formatierungen zuweisen.

Wir kommen nun zur Lösung:

Beispiel 12.2 *Zeilen Formatieren: Gewinnbeispiel*

```
Sub zeilenFormatieren_Click()
    Const ANZAHL_SPALTE As Long = 3
    Const EINKAUFSPREIS_SPALTE As Long = 4
    Const KATEGORIE_SPALTE As Long = 5
    Const VERSANDART_SPALTE As Long = 6
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Const TABELLENBLATT As Long = 1

    Dim i As Long
    Dim j As Long
    Dim letzteBesetzteZeile As Long
    Dim letzteBesetzteSpalte As Long
    Dim farbe As Long
    Dim fett As Boolean

    Dim kategorie As String
    Dim versandart As String
    Dim anzahl As Long
    Dim einkaufspreis As Double

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(ANZAHL_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
    letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1,➥
         ANZAHL_SPALTE, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
      kategorie = Sheets(TABELLENBLATT).Cells(i, KATEGORIE_SPALTE)
      versandart = Sheets(TABELLENBLATT).Cells(i, VERSANDART_SPALTE)
      anzahl = Sheets(TABELLENBLATT).Cells(i, ANZAHL_SPALTE)
      einkaufspreis = Sheets(TABELLENBLATT).Cells(i, EINKAUFSPREIS_SPALTE)
      farbe = ermittleFarbe(kategorie, versandart)
      fett = schreibeInFett(anzahl, einkaufspreis)
      For j = 1 To letzteBesetzteSpalte
          Sheets(TABELLENBLATT).Cells(i, j).Interior.Color = farbe
          Sheets(TABELLENBLATT).Cells(i, j).Font.Bold = fett
      Next j
    Next i
End Sub
```

Gehen wir dieses Programm im Einzelnen durch. Zunächst definieren wir Konstanten für die Spalten, deren Werte für die Formatierungen verantwortlich sind, für die Zeile, in der die Daten beginnen und für das Tabellenblatt, in dem das Programm laufen soll. Dies ist sehr sinnvoll, denn diese Zahlen können sich ändern. Dann brauchen wir nur die jeweilige Konstante anzupassen, ohne uns um die Logik kümmern zu müssen.

```
Const ANZAHL_SPALTE As Long = 3
Const EINKAUFSPREIS_SPALTE As Long = 4
Const KATEGORIE_SPALTE As Long = 5
Const VERSANDART_SPALTE As Long = 6
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
Const TABELLENBLATT As Long = 1
```

Wir deklarieren die Variablen, die wir benötigen:

- die Schleifenvariablen,
- die Variablen für die letzte besetzte Zeile und für die letzte besetzte Spalte,
- die Variablen für die Farbe und ob die Schriftdarstellung in „Fett“ erfolgen soll,
- die Variablen, die wir benötigen, um in der jeweiligen Zeile die für die Formatierung benötigten Werte einzulesen.

```
Dim i As Long
Dim j As Long
Dim letzteBesetzteZeile As Long
Dim letzteBesetzteSpalte As Long
Dim farbe As Long
Dim fett As Boolean

Dim kategorie As String
Dim versandart As String
Dim anzahl As Long
Dim einkaufspreis As Double
```

Wir bestimmen die letzte besetzte Zeile, also die Zeile, bis zu der unsere Schleife laufen muss, sowie die letzte besetzte Spalte für die Formatierung der jeweiligen Zeile.

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(ANZAHL_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1, ➥
    ANZAHL_SPALTE, TABELLENBLATT)
```

Zur Bestimmung der letzten besetzten Zeile können wir grundsätzlich jede Spalte des Tabellenblatts bis zur Spalte G nehmen, weil alle Spalten durchgängig gefüllt sind. Die Zeile, in der wir die Bestimmung der letzten besetzten Zeile starten, ist natürlich die Zeile 2, weil hier die Daten beginnen. Diese hatten wir ja auf der Konstanten *ERSTE_ZEILE_MIT_INFORMATIONEN* abgelegt. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteZeileInSpalte* hatten wir in Kapitel 9.3 eingehend besprochen, so dass auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun bestimmen wir die letzte besetzte Spalte. Dies tun wir in der Zeile mit den Überschriften. Diese Zeile ist natürlich für alle Spalten gefüllt (jede Spalte hat eine Überschrift). Die Zeile mit der Überschrift ist die Zeile über der Zeile, in der die Daten starten (also *ERSTE_ZEILE_MIT_INFORMATIONEN*) - 1). Starten können wir den Vorgang in jeder beliebigen Spalte, wir nehmen die *ANZAHL_SPALTE*, da wir diese Konstante bereits zur Verfügung haben. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteSpalteInZeile* hatten wir in Kapitel 9.3 eingehend besprochen, so dass auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun startet die äußere Schleife. Sie läuft natürlich von Zeile 2 (*ERSTE_ZEILE_MIT_INFORMATIONEN*) zur letzten besetzten Zeile, da ja jeder Datensatz behandelt werden muss:

```
For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
```

Wir lesen die für die Bestimmung der Farbe und des Schriftgewichts benötigten Werte der jeweiligen Zeile auf die von uns deklarierten Variablen.

```
kategorie = Sheets(TABELLENBLATT).Cells(i, KATEGORIE_SPALTE)
versandart = Sheets(TABELLENBLATT).Cells(i, VERSANDART_SPALTE)
anzahl = Sheets(TABELLENBLATT).Cells(i, ANZAHL_SPALTE)
einkaufspreis = Sheets(TABELLENBLATT).Cells(i, EINKAUFSPREIS_SPALTE)
```

Anschließend müssen wir Farbe und Schriftgewicht bestimmen. Diese Aufgaben werden wir in Funktionen auslagern. Wie Sie bereits gelernt haben ist dies immer sinnvoll. Das Hauptprogramm wird kürzer, die Funktionen sind besser testbar, das Programm wird leichter verständlich und besser änderbar. Wir benötigen zwei Funktionen, eine zur Bestimmung der Farbe (diese werden wir *ermittleFarbe* nennen) und eine zur Bestimmung des Schriftgewichts (diese werden wir *schreibeInFett* nennen). Die Hintergrundfarbe hängt von der Softwarekategorie und der Versandart ab, die Funktion *ermittleFarbe* benötigt also zwei Übergabeparameter, nämlich die Variablen *kategorie* und *versandart*. Das Schriftgewicht hängt von der Anzahl und dem Einkaufspreis ab, die Funktion *schreibeInFett* benötigt also ebenfalls zwei Übergabeparameter, nämlich die Variablen *anzahl* und *einkaufspreis*.

```
farbe = ermittleFarbe(kategorie, versandart)
fett = schreibeInFett(anzahl, einkaufspreis)
```

Beachten Sie, dass Sie hier nur den Aufruf der Funktionen sehen (wie immer im aufrufenden Programmteil). Die Funktionen *ermittleFarbe* und *schreibeInFett* müssen existieren, damit die Aufrufe funktionieren. Die Implementierung dieser Funktionen besprechen wir nach diesem Programm.
Nun müssen wir in einer Schleife von der ersten Spalte bis zur letzten besetzten Spalte allen Zellen die auf der Variable *farbe* abgespeicherte Farbe und die Information, ob die Darstellung in „Fett“ erfolgen soll, zuweisen. Wie dies realisiert wird, haben wir bereits in Beispiel 12.1 beschrieben.

```
For j = 1 To letzteBesetzteSpalte
    Sheets(TABELLENBLATT).Cells(i, j).Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(i, j).Font.Bold = fett
Next j
```

Wir beenden die äußere Schleife

```
Next i
```

und das Programm

```
End Sub
```

Bleiben noch die Implementierungen der beiden Funktionen, die wir aus dem Hauptprogramm aufrufen. Wir starten mit der Funktion *ermittleFarbe*:

Beispiel 12.3 *Die Funktion ermittleFarbe des Gewinnbeispiels*

```
Function ermittleFarbe(kategorie As String, versandart As String) As Long
    Dim farbe As Long
    If kategorie = "Betriebssysteme" Then
        farbe = vbYellow
    ElseIf kategorie = "Office" Then
        farbe = RGB(255, 50, 255)
    Else
        farbe = xlNone
    End If
    If versandart = "CD-Versand" Then
        farbe = vbRed
    End If
    ermittleFarbe = farbe
End Function
```

Die Funktion *ermittleFarbe* benötigt, wie bereits erwähnt zwei Übergabeparameter, nämlich *kategorie* und *versandart*. Beide sind vom Typ *String*. Die Funktion gibt einen *Long*-Wert zurück, denn dies ist, wie Sie ja wissen, der Datentyp für die interne Kodierung der Farben in Excel:

```
Function ermittleFarbe(kategorie As String, versandart As String) As Long
```

Nach der Deklaration der Variablen *farbe*

```
    Dim farbe As Long
```

bestimmen wir die Farbe in Abhängigkeit von der Softwarekategorie. Dazu benutzen wir das *if-elseif*-Konstrukt. Für „Betriebssysteme“ soll die Hintergrundfarbe gelb gewählt werden, hier können wir die VBA-interne Konstante *vbYellow* benutzen (vgl. Beispiel 11.3). Für „Office“ sind die Rot-Grün-und Blauanteile gegeben, also nutzen wir die VBA-Funktion *RGB* (vgl. Beispiel 11.1). In allen anderen Fällen soll nicht gefärbt werden. Wie Sie aus Kap. 11.1.4 wissen, gibt es eine VBA-Farbkonstante, die den VBA-Originalzustand erhält. Dies ist die Konstante *xlNone*. Diese weisen wir also in allen anderen Fällen zu:

```
    If kategorie = "Betriebssysteme" Then
        farbe = vbYellow
    ElseIf kategorie = "Office" Then
        farbe = RGB(255, 50, 255)
    Else
```

```
        farbe = xlNone
    End If
```

Damit ist die Farbe in Abhängigkeit von der Produktkategorie bestimmt. Wir müssen nun noch die Sonderregel programmieren: Wenn die Versandart „CD-Versand“ ist, so soll die Zeile, unabhängig von allem anderen, rot gefärbt werden. Hierzu reicht ein einfaches *if*-Konstrukt. Wir prüfen, ob diese Bedingung erfüllt ist, und wenn dies der Fall ist, überschreiben wir die Farbe einfach:

```
    If versandart = "CD-Versand" Then
        farbe = vbRed
    End If
```

Dann weisen wir die ermittelte Farbe dem Funktionsnamen zu und beenden die Funktion:

```
    ermittleFarbe = farbe
End Function
```

Es fehlt noch die Funktion *schreibeInFett*:

Beispiel 12.4 *Die Funktion schreibeInFett des Gewinnbeispiels*

```
Function schreibeInFett(anzahl As Long, einkaufspreis As Double) As Boolean
    Dim gesamtEinkaufspreis As Double
    Const EINKAUFSPREIS_GRENZE As Double = 100
    gesamtEinkaufspreis = anzahl * einkaufspreis
    If gesamtEinkaufspreis > EINKAUFSPREIS_GRENZE Then
        schreibeInFett = True
    Else
        schreibeInFett = False
    End If
End Function
```

Hier müssen wir, wie wir aus Kap. 12.1, *true* zurückgeben, wenn in Fett dargestellt werden soll, *false* sonst. Die Funktion muss also den Datentyp *boolean* an das aufrufende Programm zurückgeben. Die Funktion selber benötigt, wie bereits erwähnt zwei Übergabeparameter, nämlich *anzahl* und *einkaufspreis*. *anzahl* ist vom Typ *Long*, *einkaufspreis* vom Typ *Double*. Also ergibt sich:

```
Function schreibeInFett(anzahl As Long, einkaufspreis As Double) As Boolean
```

In der Funktion selber deklarieren wir eine Variable (*gesamtEinkaufspreis*) und eine Konstante für die Grenze, ab wann die Darstellung in Fett erfolgen soll. Dann wird der *gesamtEinkaufspreis* ausgerechnet:

```
    Dim gesamtEinkaufspreis As Double
    Const EINKAUFSPREIS_GRENZE As Double = 100
    gesamtEinkaufspreis = anzahl * einkaufspreis
```

Nun müssen wir nur noch prüfen, ob der ausgerechnete Gesamteinkaufspreis größer als die Grenze ist, denn dann müssen wir in „Fett“ formatieren. Dazu reicht eine einfache *if*-Anweisung. Hier verzichten wir auf eine Zwischenvariable, wir weisen das Ergebnis direkt dem Funktionsnamen zu und beenden dann die Funktion:

```
    If gesamtEinkaufspreis > EINKAUFSPREIS_GRENZE Then
        schreibeInFett = True
    Else
        schreibeInFett = False
    End If
End Function
```

Damit ist die Programmierung dieser Anforderungen erledigt.

12.3 Das Schlüsseldienstbeispiel

Schreiben Sie ein Programm, das die in Abb. 12.3 gezeigten Formatierungen durchführt:

- In Zeilen, wo das Verhältnis zwischen Anfahrtszeit und Arbeitszeit kleiner gleich 1 ist, wird die Textfarbe Grün gewählt.
- In Zeilen, wo das Verhältnis zwischen Anfahrtszeit und Arbeitszeit größer als 1 ist, wird die Textfarbe Gelb mit der Farbcodierung Rotanteil 250, Grünanteil 210, Blauanteil 100 gewählt.
- In Zeilen, wo das Verhältnis zwischen Anfahrtszeit und Arbeitszeit größer als 2 ist, wird die Textfarbe Rot gewählt.

In Ballungsgebieten wie im Ruhrgebiet wird eine Durchschnittsgeschwindigkeit von 36 km/h erreicht. Diese Angabe soll zur Ermittlung der Anfahrtszeit herangezogen werden.
Bei Datensätzen (Excel-Zeilen) mit Nachtzuschlag soll der Text in „Kursiv“ dargestellt werden.

	A	B	C	D	E
1	Kunde	gefahrene Kilometer	Arbeitszeit (Min)	Nachtzuschlag	Preis
2	Meyer	20	20		70
3	Müller	10	10		35
4	*Schmidt*	*100*	*25*	*ja*	*126*
5	Schmid	15	35		100
6	*Seran*	*10*	*11*	*ja*	*51,8*
7	Fischer	7	7		29
8					
9					
10					

Abbildung 12.3
Färben: Schlüsseldienstbeispiel

Wie wir grundsätzlich vorgehen werden, wissen wir aus Kap. 11.1:

- Wir ermitteln die letzte besetzte Zeile. Dann wissen wir, bis zu welcher Zeile die Formatierungen zugewiesen werden müssen. Hierzu benutzen wir die Funktion *ermittleLetzteBesetzteZeileInSpalte* aus Kap. 9.
- Wir ermitteln die letzte besetzte Spalte. Auch dies können wir bereits. Dazu schrieben wir die Funktion *ermittleLetzteBesetzteSpalteInZeile* ebenfalls in Kap. 9.
- Wir laufen in einer Schleife über die Zeilen.
- In jeder Zeile
  - ermitteln wir die Textfarbe.
  - stellen wir fest, ob die Darstellung in „Kursiv“ erfolgen muss.
  - Dann müssen wir nur noch in einer Schleife über die Spalten laufen und die Formatierungen zuweisen.

Wir kommen nun zur Lösung:

Beispiel 12.5 *Zeilen Formatieren: Schlüsseldienstbeispiel*

```
Sub zeilenFormatieren_Click()
    Const NACHTZUSCHLAG_SPALTE As Long = 4
    Const KILOMETER_SPALTE As Long = 2
    Const ARBEITSZEIT_SPALTE As Long = 3
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Const TABELLENBLATT As Long = 1
    Dim i As Long
    Dim j As Long
```

```
    Dim letzteBesetzteZeile As Long
    Dim letzteBesetzteSpalte As Long
    Dim farbe As Long
    Dim kursiv As Boolean

    Dim arbeitszeit As Double
    Dim nachtzuschlag As String
    Dim kilometer As Double

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(KILOMETER_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
    letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1,➥
         KILOMETER_SPALTE, TABELLENBLATT)
    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        arbeitszeit = Sheets(TABELLENBLATT).Cells(i, ARBEITSZEIT_SPALTE)
        nachtzuschlag = Sheets(TABELLENBLATT).Cells(i, NACHTZUSCHLAG_SPALTE)
        kilometer = Sheets(TABELLENBLATT).Cells(i, KILOMETER_SPALTE)
        farbe = bestimmeTextfarbe(kilometer, arbeitszeit)
        kursiv = schreibeInKursiv(nachtzuschlag)
        For j = 1 To letzteBesetzteSpalte
            Sheets(TABELLENBLATT).Cells(i, j).Font.Color = farbe
            Sheets(TABELLENBLATT).Cells(i, j).Font.Italic = kursiv
        Next j
    Next i
End Sub
```

Gehen wir dieses Programm im Einzelnen durch. Zunächst definieren wir Konstanten für die Spalten, deren Werte für die Formatierungen verantwortlich sind, für die Zeile, in der die Daten beginnen und für das Tabellenblatt, in dem das Programm laufen soll. Dies ist sehr sinnvoll, denn diese Zahlen können sich ändern. Dann brauchen wir nur die jeweilige Konstante anzupassen, ohne uns um die Logik kümmern zu müssen.

```
    Const NACHTZUSCHLAG_SPALTE As Long = 4
    Const KILOMETER_SPALTE As Long = 2
    Const ARBEITSZEIT_SPALTE As Long = 3
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Const TABELLENBLATT As Long = 1
```

Wir deklarieren die Variablen, die wir benötigen:

- die Schleifenvariablen,
- die Variablen für die letzte besetzte Zeile und für die letzte besetzte Spalte,
- die Variablen für die Textfarbe und ob die Schriftdarstellung in „Kursiv“ erfolgen soll,
- die Variablen, die wir benötigen, um in der jeweiligen Zeile die für die Formatierung benötigten Werte einzulesen.

```
    Dim i As Long
    Dim j As Long
    Dim letzteBesetzteZeile As Long
    Dim letzteBesetzteSpalte As Long
    Dim farbe As Long
    Dim kursiv As Boolean

    Dim arbeitszeit As Double
    Dim nachtzuschlag As String
    Dim kilometer As Double
```

Wir bestimmen die letzte besetzte Zeile, also die Zeile, bis zu der unsere Schleife laufen muss, sowie die letzte besetzte Spalte für die Formatierung der jeweiligen Zeile.

```
    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(KILOMETER_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
    letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1,➥
         KILOMETER_SPALTE, TABELLENBLATT)
```

Zur Bestimmung der letzten besetzten Zeile können wir jede Spalte des Tabellenblatts bis zur Spalte E nehmen, mit Ausnahme der Spalte D. Die Spalte D enthält leere Zellen. Das Verfahren, das uns die letzte besetzte Zeile bestimmt, würde in der Spalte D bereits in Zeile 2 abbrechen, da die Zelle D2 leer ist. Unsere Funktion zur Ermittlung der letzten besetzten Zeile würde in der Spalte D also 1 ermitteln. Daher müssen wir eine andere Spalte nehmen, wir haben die *KILOMETER_SPALTE* Spalte gewählt. Die Zeile, in der wir die Bestimmung der letzten besetzten Zeile starten, ist natürlich die Zeile 2, weil hier die Daten beginnen. Diese hatten wir ja auf der Konstanten *ERSTE_ZEILE_MIT_INFORMATIONEN* abgelegt. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteZeileInSpalte* hatten wir in Kapitel 9.3 eingehend besprochen, so dass auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun bestimmen wir die letzte besetzte Spalte. Dies tun wir in der Zeile mit den Überschriften. Diese Zeile ist natürlich für alle Spalten gefüllt (jede Spalte hat eine Überschrift). Die Zeile mit der Überschrift ist die Zeile über der Zeile, in der die Daten starten (also *ERSTE_ZEILE_MIT_INFORMATIONEN*) - 1). Starten können wir den Vorgang in jeder beliebigen Spalte, wir nehmen wieder die *KILOMETER_SPALTE*. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteSpalteInZeile* hatten wir in Kapitel 9.3 eingehend besprochen, so dass auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun startet die äußere Schleife. Sie läuft natürlich von Zeile 2 (*ERSTE_ZEILE_MIT_INFORMATIONEN*) zur letzten besetzten Zeile, da ja jeder Datensatz behandelt werden muss:

```
    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
```

Wir lesen die für die Bestimmung der Textfarbe und des Schriftgewichts benötigten Werte der jeweiligen Zeile auf die von uns deklarierten Variablen.

```
        arbeitszeit = Sheets(TABELLENBLATT).Cells(i, ARBEITSZEIT_SPALTE)
        nachtzuschlag = Sheets(TABELLENBLATT).Cells(i, NACHTZUSCHLAG_SPALTE)
        kilometer = Sheets(TABELLENBLATT).Cells(i, KILOMETER_SPALTE)
```

Anschließend müssen wir Farbe und Schriftstil bestimmen. Diese Aufgaben werden wir in Funktionen auslagern. Wie Sie bereits gelernt haben ist dies immer sinnvoll. Das Hauptprogramm wird kürzer, die Funktionen sind besser testbar, das Programm wird leichter verständlich und besser änderbar. Wir benötigen zwei Funktionen, eine zur Bestimmung der Textfarbe (diese werden wir *bestimmeTextfarbe* nennen) und eine zur Bestimmung des Schriftstils (diese werden wir *schreibeInKursiv* nennen). Die Textfarbe hängt von den gefahrenen Kilometern und der Arbeitszeit ab, die Funktion *bestimmeTextfarbe* benötigt also zwei Übergabeparameter, nämlich die Variablen *kilometer* und *arbeitszeit*. Der Schriftstil hängt nur vom Nachtzuschlag ab, die Funktion *schreibeInKursiv* benötigt also nur einen Übergabeparameter, nämlich die Variable *nachtzuschlag*.

```
        farbe = bestimmeTextfarbe(kilometer, arbeitszeit)
        kursiv = schreibeInKursiv(nachtzuschlag)
```

Beachten Sie, dass Sie hier nur den Aufruf der Funktionen sehen (wie immer im aufrufenden Programmteil). Die Funktionen *bestimmeTextfarbe* und *schreibeInKursiv* müssen existieren, damit die Aufrufe funktionieren. Die Implementierung dieser Funktionen besprechen wir nach diesem Programm.
Nun müssen wir in einer Schleife von der ersten Spalte bis zur letzten besetzten Spalte allen Zellen die auf der Variable *farbe* abgespeicherte Textfarbe und die Information, ob die Darstellung in „Kursiv“ erfolgen soll, zuweisen. Wie dies realisiert wird, haben wir bereits in Beispiel 12.1 beschrieben.

```
        For j = 1 To letzteBesetzteSpalte
            Sheets(TABELLENBLATT).Cells(i, j).Font.Color = farbe
            Sheets(TABELLENBLATT).Cells(i, j).Font.Italic = kursiv
        Next j
```

Wir beenden die äußere Schleife

```
    Next i
```

und das Programm

```
End Sub
```

Bleiben noch die Implementierungen der beiden Funktionen, die wir aus dem Hauptprogramm aufrufen. Wir starten mit der Funktion *bestimmeTextfarbe*:

Beispiel 12.6 *Die Funktion bestimmeTextfarbe des Schlüsseldienstbeispiels*

```
Function bestimmeTextfarbe(gefahreneKilometer As Double, arbeitszeit As Double) As Long
    Const KILOMETER_GRENZE As Double = 100
    Const DURCHSCHNITTS_GESCWINDIGKEIT As Double = 36
    Const VERHAELTNIS_GRENZE_1 As Double = 2
    Const VERHAELTNIS_GRENZE_2 As Double = 1

    Dim verhaeltnis As Double
    Dim farbe As Long
    Dim anfahrtszeit As Double

    anfahrtszeit = (gefahreneKilometer / DURCHSCHNITTS_GESCWINDIGKEIT) * 60
    verhaeltnis = anfahrtszeit / arbeitszeit

    If verhaeltnis > VERHAELTNIS_GRENZE_1 Then
        farbe = vbRed
    ElseIf verhaeltnis > VERHAELTNIS_GRENZE_2 Then
        farbe = RGB(250, 210, 100)
    Else
        farbe = vbGreen
    End If

    bestimmeTextfarbe = farbe
End Function
```

Die Funktion *bestimmeTextfarbe* benötigt, wie bereits erwähnt zwei Übergabeparameter, nämlich *gefahreneKilometer* und *arbeitszeit*. Beide sind vom Typ *Double*. Die Funktion gibt einen *Long*-Wert zurück, denn dies ist, wie Sie ja wissen, der Datentyp für die interne Kodierung der Farben in Excel:

```
Function bestimmeTextfarbe(gefahreneKilometer As Double, arbeitszeit As Double) As Long
```

Nach der Deklaration von Konstanten und Variablen müssen wir zunächst die Anfahrtszeit berechnen. Dies ist aber ziemlich leicht, da wir die gefahrenen Kilometer und die Durchschnittsgeschwindigkeit kennen.

```
    anfahrtszeit = (gefahreneKilometer / DURCHSCHNITTS_GESCWINDIGKEIT) * 60
```

Hier müssen wir mit 60 multiplizieren, weil die Durchschnittsgeschwindigkeit pro Stunde gegeben ist, wir die Anfahrtszeit aber in Minuten benötigen. Sodann bestimmen wir das Verhältnis zwischen Anfahrtszeit und Arbeitszeit:

```
    verhaeltnis = anfahrtszeit / arbeitszeit
```

Mit einem *if-elseif*-Statement können wir nun die Farbe bestimmen:

```
    If verhaeltnis > VERHAELTNIS_GRENZE_1 Then
        farbe = vbRed
    ElseIf verhaeltnis > VERHAELTNIS_GRENZE_2 Then
        farbe = RGB(250, 210, 100)
    Else
        farbe = vbGreen
    End If
```

Dann weisen wir die ermittelte Farbe dem Funktionsnmaen zu und beenden die Funktion:

```
    bestimmeTextfarbe = farbe
End Function
```

Es fehlt noch die Funktion *schreibeInKursiv*:

Beispiel 12.7 *Die Funktion schreibeInKursiv des Schlüsseldienstbeispiels*

```
Function schreibeInKursiv(nachtzuschlag As String) As Boolean
    If nachtzuschlag = "ja" Then
        schreibeInKursiv = True
    Else
        schreibeInKursiv = False
    End If
End Function
```

Hier müssen wir, wie wir aus Kap. 12.1, *true* zurückgeben, wenn in Kursiv dargestellt werden soll, *false* sonst. Die Funktion muss also den Datentyp *boolean* an das aufrufende Programm zurückgeben. Die Funktion selber benötigt, wie bereits erwähnt einen Übergabeparameter, nämlich *nachtzuschlag*. *nachtzuschlag* ist vom Typ *String*.

```
Function schreibeInKursiv(nachtzuschlag As String) As Boolean
```

Die Funktion selber ist trivial. Wenn die Variable *nachtzuschlag* den Wert „ja“ hat, muss die Funktion *true* zurückgeben, *false* sonst. Dafür reicht ein einfaches *if*-Statement. Hier verzichten wir auf eine Zwischenvariable, wir weisen das Ergebnis direkt dem Funktionsnmaen zu und beenden dann die Funktion:

```
    If nachtzuschlag = "ja" Then
        schreibeInKursiv = True
    Else
        schreibeInKursiv = False
    End If
End Function
```

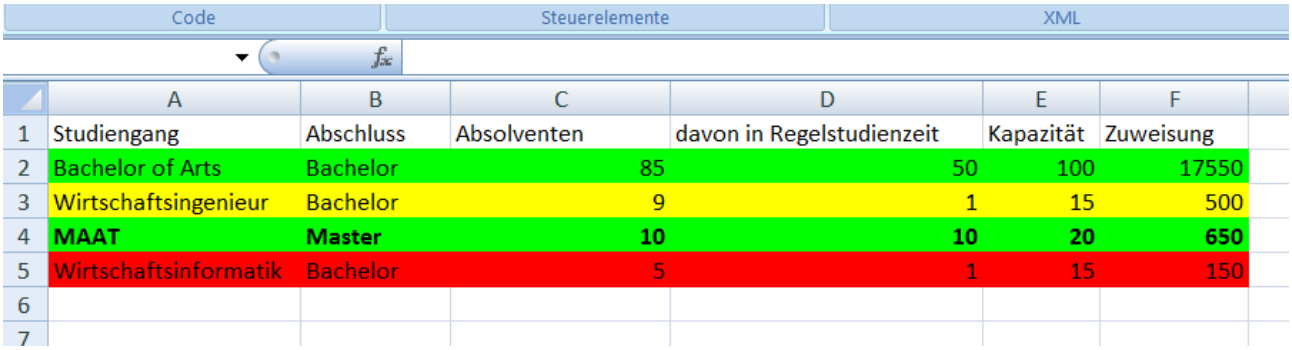
Damit ist die Programmierung dieser Anforderungen erledigt.

12.4 Das Zuweisungsbeispiel

Schreiben Sie ein Programm, das die in Abb. 12.4 gezeigten Formatierungen durchführt:

- Wenn die Absolventenquote größer als 60% ist, so soll die Zeile grün gefärbt werden.
- Zeilen mit einer Absolventenquote zwischen 40 und 60% werden gelb gefärbt.
- Alle anderen Zeilen werden rot gefärbt.

Werden jedoch alle Absolventen in Regelstudienzeit fertig, so entfällt die Gelbfärbung. Auch die Zeilen mit einer Absolventenquote zwischen 40 und 60% werden dann grün gefärbt.
Bei Datensätzen (Excel-Zeilen), die sich auf Masterstudiengänge beziehen, soll der Text in „Fett“ dargestellt werden.



	A	B	C	D	E	F
1	Studiengang	Abschluss	Absolventen	davon in Regelstudienzeit	Kapazität	Zuweisung
2	Bachelor of Arts	Bachelor	85	50	100	17550
3	Wirtschaftsingenieur	Bachelor	9	1	15	500
4	**MAAT**	**Master**	**10**	**10**	**20**	**650**
5	Wirtschaftsinformatik	Bachelor	5	1	15	150
6						
7						

Abbildung 12.4
Färben: Zuweisungsbeispiel

Wie wir grundsätzlich vorgehen werden, wissen wir aus Kap. 11.1:

- Wir ermitteln die letzte besetzte Zeile. Dann wissen wir, bis zu welcher Zeile die Formatierungen zugewiesen werden müssen. Hierzu benutzen wir die Funktion *ermittleLetzteBesetzteZeileInSpalte* aus Kap. 9.
- Wir ermitteln die letzte besetzte Spalte. Auch dies können wir bereits. Dazu schrieben wir die Funktion *ermittleLetzteBesetzteSpalteInZeile* ebenfalls in Kap. 9.
- Wir laufen in einer Schleife über die Zeilen.
- In jeder Zeile
  - ermitteln wir die Hintergrundfarbe.
  - stellen wir fest, ob die Darstellung in „Fett“ erfolgen muss.
  - Dann müssen wir nur noch in einer Schleife über die Spalten laufen und die Formatierungen zuweisen.

Wir kommen nun zur Lösung:

Beispiel 12.8 *Zeilen Färben: Zuweisungsbeispiel*

```
Sub zeilenFormatieren_Click()
    Const ABSCHLUSS_SPALTE As Long = 2
    Const ABSOLVENTEN_SPALTE As Long = 3
    Const DAVON_REGELSTUDIENZEIT_SPALTE As Long = 4
    Const KAPAZITAETS_SPALTE As Long = 5
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Const TABELLENBLATT As Long = 1

    Dim i As Long
    Dim j As Long
    Dim letzteBesetzteZeile As Long
    Dim letzteBesetzteSpalte As Long
    Dim farbe As Long
    Dim fett As Boolean

    Dim abschluss As String
    Dim absolventen As Long
    Dim davonInRegelstudienzeit As Long
    Dim kapazitaet As Long

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(ABSCHLUSS_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
    letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1,➥
         ABSCHLUSS_SPALTE, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        abschluss = Sheets(TABELLENBLATT).Cells(i, ABSCHLUSS_SPALTE)
        absolventen = Sheets(TABELLENBLATT).Cells(i, ABSOLVENTEN_SPALTE)
        davonInRegelstudienzeit = Sheets(TABELLENBLATT).Cells(i, DAVON_REGELSTUDIENZEIT_SPALTE)
        kapazitaet = Sheets(TABELLENBLATT).Cells(i, KAPAZITAETS_SPALTE)
        farbe = ermittleFarbe(absolventen, davonInRegelstudienzeit, kapazitaet)
        fett = schreibeInFett(abschluss)
        For j = 1 To letzteBesetzteSpalte
            Sheets(TABELLENBLATT).Cells(i, j).Interior.Color = farbe
            Sheets(TABELLENBLATT).Cells(i, j).Font.Bold = fett
        Next j
    Next i
End Sub
```

Gehen wir dieses kurze Programm im Einzelnen durch. Zunächst definieren wir Konstanten für die Spalte, deren Werte für das Färben verantwortlich sind, für die Zeile, in der die Daten beginnen und für das Tabellenblatt, in dem das Programm laufen soll. Dies ist sehr sinnvoll, denn diese Zahlen können sich ändern. Dann brauchen wir nur die jeweilige Konstante anzupassen, ohne uns um die Logik kümmern zu müssen.

```
    Const ABSCHLUSS_SPALTE As Long = 2
    Const ABSOLVENTEN_SPALTE As Long = 3
```

```
Const DAVON_REGELSTUDIENZEIT_SPALTE As Long = 4
Const KAPAZITAETS_SPALTE As Long = 5
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
Const TABELLENBLATT As Long = 1
```

Wir deklarieren die Variablen, die wir benötigen:

- die Schleifenvariablen,
- die Variablen für die letzte besetzte Zeile und für die letzte besetzte Spalte,
- die Variablen für die Textfarbe und ob die Schriftdarstellung in „Fett“ erfolgen soll,
- die Variablen, die wir benötigen, um in der jeweiligen Zeile die für die Formatierung benötigten Werte einzulesen.

```
Dim i As Long
Dim j As Long
Dim letzteBesetzteZeile As Long
Dim letzteBesetzteSpalte As Long
Dim farbe As Long
Dim fett As Boolean

Dim abschluss As String
Dim absolventen As Long
Dim davonInRegelstudienzeit As Long
Dim kapazitaet As Long
```

Wir bestimmen die letzte besetzte Zeile, also die Zeile, bis zu der unsere Schleife laufen muss, sowie die letzte besetzte Spalte für die Formatierung der jeweiligen Zeile.

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(ABSCHLUSS_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(ERSTE_ZEILE_MIT_INFORMATIONEN - 1,➥
     ABSCHLUSS_SPALTE, TABELLENBLATT)
```

Zur Bestimmung der letzten besetzten Zeile können wir grundsätzlich jede Spalte des Tabellenblatts bis zur Spalte F nehmen, weil alle Spalten durchgängig gefüllt sind. Die Zeile, in der wir die Bestimmung der letzten besetzten Zeile starten, ist natürlich die Zeile 2, weil hier die Daten beginnen. Diese hatten wir ja auf der Konstanten *ERSTE_ZEILE_MIT_INFORMATIONEN* abgelegt. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteZeileInSpalte* hatten wir in Kapitel 9.3 eingehend besprochen, so dass auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun bestimmen wir die letzte besetzte Spalte. Dies tuen wir in der Zeile mit den Überschriften. Diese Zeile ist natürlich für alle Spalten gefüllt (jede Spalte hat eine Überschrift). Die Zeile mit der Überschrift ist die Zeile über der Zeile, in der die Daten starten (also *ERSTE_ZEILE_MIT_INFORMATIONEN*) - 1). Starten können wir den Vorgang in jeder beliebigen Spalte, wir nehmen die *ABSCHLUSS_SPALTE*, da wir diese Konstante bereits zur Verfügung haben. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteSpalteInZeile* hatten wir in Kapitel 9.3 eingehend besprochen, so dass wir auf eine erneute Diskussion verzichten. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun startet die äußere Schleife. Sie läuft natürlich von Zeile 2 (*ERSTE_ZEILE_MIT_INFORMATIONEN*) zur letzten besetzten Zeile, da ja jeder Datensatz behandelt werden muss:

```
For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
```

Wir lesen die für die Bestimmung der Farbe und des Schriftgewichts benötigten Werte der jeweiligen Zeile auf die von uns deklarierten Variablen.

```
    abschluss = Sheets(TABELLENBLATT).Cells(i, ABSCHLUSS_SPALTE)
    absolventen = Sheets(TABELLENBLATT).Cells(i, ABSOLVENTEN_SPALTE)
    davonInRegelstudienzeit = Sheets(TABELLENBLATT).Cells(i, DAVON_REGELSTUDIENZEIT_SPALTE)
    kapazitaet = Sheets(TABELLENBLATT).Cells(i, KAPAZITAETS_SPALTE)
```

Anschließend müssen wir Farbe und Schriftgewicht bestimmen. Diese Aufgaben werden wir in Funktionen auslagern. Wie Sie bereits gelernt haben ist dies immer sinnvoll. Das Hauptprogramm wird kürzer, die Funktionen sind besser testbar, das Programm wird leichter verständlich und besser änderbar. Wir benötugen zwei Funktionen, eine zur Bestimmung der Farbe (diese werden wir *ermittleFarbe* nennen) und eine zur Bestimmung des Schriftgewichts (diese werden wir *schreibeInFett* nennen). Die Hintergrundfarbe hängt von der Absolventenanzahl, der Kapazität und der Anzahl Absolventen in Regelstudienzeit ab. Die Funktion *ermittleFarbe* benötigt also drei Übergabeparameter, nämlich die Variablen *absolventen*, *davonInRegelstudienzeit* und *kapazitaet*. Das Schriftgewicht hängt nur vom Abschluss ab, die Funktion *schreibeInFett* benötigt also nur einen Übergabeparameter, nämlich die Variable *abschluss*.

```
        farbe = ermittleFarbe(absolventen, davonInRegelstudienzeit, kapazitaet)
        fett = schreibeInFett(abschluss)
```

Beachten Sie, dass Sie hier nur den Aufruf der Funktionen sehen (wie immer im aufrufenden Programmteil). Die Funktionen *ermittleFarbe* und *schreibeInFett* müssen existieren, damit die Aufrufe funktionieren. Die Implementierung dieser Funktionen besprechen wir nach diesem Programm.
Nun müssen wir in einer Schleife von der ersten Spalte bis zur letzten besetzten Spalte allen Zellen die auf der Variable *farbe* abgespeicherte Farbe und die Information, ob die Darstellung in „Fett“ erfolgen soll, zuweisen. Wie dies realisiert wird, haben wir bereits in Beispiel 12.1 beschrieben.

```
        For j = 1 To letzteBesetzteSpalte
            Sheets(TABELLENBLATT).Cells(i, j).Interior.Color = farbe
            Sheets(TABELLENBLATT).Cells(i, j).Font.Bold = fett
        Next j
```

Wir beenden die äußere Schleife

```
    Next i
```

und das Programm

```
End Sub
```

Bleiben noch die Implementierungen der beiden Funktionen, die wir aus dem Hauptprogramm aufrufen. Wir starten mit der Funktion *ermittleFarbe*:

Beispiel 12.9 *Die Funktion ermittleFarbe des Zuweisungsbeispiels*

```
Function ermittleFarbe(anzahlAbsolventen As Long, davonInRegelstudienzeit As Long, kapazitaet As ➥
    Long) As Long
    Const GRUEN_GRENZE = 0.6
    Const GELB_GRENZE = 0.4

    Dim absolventenVerhaeltnis As Double
    Dim farbe As Long

    absolventenVerhaeltnis = anzahlAbsolventen / kapazitaet
    If absolventenVerhaeltnis > GRUEN_GRENZE Then
        farbe = vbGreen
    ElseIf absolventenVerhaeltnis > GELB_GRENZE Then
        If davonInRegelstudienzeit = anzahlAbsolventen Then
            farbe = vbGreen
        Else
            farbe = vbYellow
        End If
    Else
        farbe = vbRed
    End If
    ermittleFarbe = farbe
End Function
```

Die Funktion *ermittleFarbe* benötigt, wie bereits erwähnt drei Übergabeparameter, nämlich die Variablen *absolventen*, *davonInRegelstudienzeit* und *kapazitaet*. Alle sind vom Typ *Long*. Die Funktion gibt einen *Long*-Wert zurück, denn dies ist, wie Sie ja wissen, der Datentyp für die interne Kodierung der Farben in Excel:

```
Function ermittleFarbe(anzahlAbsolventen As Long, davonInRegelstudienzeit As Long, kapazitaet As ➥
    Long) As Long
```

Zunächst definieren wir zwei Konstante für die Grenzen, ab wann in „Grün“ bzw. in „Gelb“ gefärbt wird:

```
    Const GRUEN_GRENZE = 0.6
    Const GELB_GRENZE = 0.4
```

Als nächttes definieren wir die benötigten Variablen:

```
    Dim absolventenVerhaeltnis As Double
    Dim farbe As Long
```

Wir bestimmen das Absolventenverhältnis:

```
    absolventenVerhaeltnis = anzahlAbsolventen / kapazitaet
```

Mit einem *if-elseif*-Statement können wir nun die Farbe bestimmen:

```
    If absolventenVerhaeltnis > GRUEN_GRENZE Then
        farbe = vbGreen
    ElseIf absolventenVerhaeltnis > GELB_GRENZE Then
        If davonInRegelstudienzeit = anzahlAbsolventen Then
            farbe = vbGreen
        Else
            farbe = vbYellow
        End If
    Else
        farbe = vbRed
    End If
```

Beachten Sie, dass wir im „Gelbfall“

```
    ElseIf absolventenVerhaeltnis > GELB_GRENZE Then
```

noch überprüfen müssen, ob es nur Absolventen gibt, die in Regelstudienzeit abgeschlossen haben. Denn in diesem Fall müssen wir „grün“ und nicht „gelb“ zuweisen. So erklären sich die Zeilen:

```
        If davonInRegelstudienzeit = anzahlAbsolventen Then
            farbe = vbGreen
        Else
            farbe = vbYellow
        End If
```

Dann weisen wir die ermittelte Farbe dem Funktionsnamen zu und beenden die Funktion:

```
    bestimmeTextfarbe = farbe
End Function
```

Es fehlt noch die Funktion *schreibeInFett*:

Beispiel 12.10 *Die Funktion schreibeInFett des Zuweisungsbeispiels*

```
Function schreibeInFett(abschluss As String) As Boolean
    If abschluss = "Master" Then
      schreibeInFett = True
    Else
      schreibeInFett = False
    End If
End Function
```

Hier müssen wir, wie wir aus Kap. 12.1, *true* zurückgeben, wenn in Kursiv dargestellt werden soll, *false* sonst. Die Funktion muss also den Datentyp *boolean* an das aufrufende Programm zurückgeben. Die Funktion selber benötigt, wie bereits erwähnt einen Übergabeparameter, nämlich *abschluss*. *abschluss* ist vom Typ *String*.

```
Function schreibeInFett(abschluss As String) As Boolean
```

Die Funktion selber ist trivial. Wenn die Variable *abschluss* den Wert „ja“ hat, muss die Funktion *true* zurückgeben, *false* sonst. Dafür reicht ein einfaches *if*-Statement. Hier verzichten wir auf eine Zwischenvariable, wir weisen das Ergebnis direkt dem Funktionsnamen zu und beenden dann die Funktion:

```
    If abschluss = "Master" Then
      schreibeInFett = True
    Else
      schreibeInFett = False
    End If
End Function
```

Damit ist die Programmierung dieser Anforderungen erledigt.

Kapitel 13

Interne Nutzung eigener Prozeduren

Dieses Kapitel gibt es nur, weil in VBA im Gegensatz zu vielen anderen Programmiersprachen zwischen Funktionen und Prozeduren unterschieden wird. Sie kennen bislang nur Ereignisprozeduren. Dies sind Prozeduren, die mit einer Schaltfläche in einer Tabelle des Arbeitsblatts verbunden sind. Sie werden ausgeführt, wenn auf die mit ihnen verbundene Schaltfläche geclickt wird.
Die Übergabeliste solcher Prozeduren ist immer leer. D.h. in den runden Klammern hinter dem Prozedurnamen existieren keine Variablendeklarationen (vgl. alle Beispiele in Kap. 7). Das macht anders auch wenig Sinn, denn Ereignisprozeduren sind ja, wie bereits gesagt, mit Schaltflächen verbunden und da ist die Aktion nur clicken[1]. Irgendwelche Werte können nicht mitgegeben werden, wie bei den benutzerdefinierten Funktionen, die sich auf Zellen beziehen und damit den Inhalt „ihrer“ Zellen an die Funktion über die Parameterliste weiterreichen können.
Dies lässt sich aber auch ändern. Wir können Prozeduren aus allen unseren Funktionen und auch aus anderen Prozeduren aufrufen, genauso, wie Sie es im vorherigen Kapitel mit Funktionen gelernt haben. Und wenn man es so macht, dann kann man Prozeduren ebenfalls mit Übergabeparametern füttern.
Doch zunächst zeigen wir Ihnen eine Vereinfachung der Fehlersuche mit eigenen Prozeduren:

13.1 Prozeduren als Testrahmen für benutzerdefinierte Funktionen

Oftmals ist es ziemlich aufwendig, Fehler in benutzerdefinierten Funktionen zu finden. Wir tragen die Funktion in ein Arbeitsblatt ein, doch anstelle des Ergebnisses erscheint in der Zelle der Text *Wert* und, wenn man die Maus in die Zelle bewegt, ein ziemlich obskurer und vielfach nichtssagender Text. Dies erschwert es, Fehler zu finden. Darüberhinaus wechselt man, wenn man den Debugger[2] benutzt, ständig zwischen dem VBA-Editor und dem Arbeitsblatt hin und her.
Wie Sie bereits in Kap.9.1 gelernt haben, spielt es für Funktionen keine Rolle, ob sie aus der Tabellenkalkulation oder eben aus anderen Funktionen aufgerufen werden. Im ersten Fall schreiben sie ihr Ergebnis in eine Zelle, im zweiten Fall halt in eine Variable der aufrufenden Funktion. In Kap. 7 haben Sie gelernt, dass man Prozeduren direkt im VBA-Editor ausführen kann. Es gibt neben den in in Kap. 7 erwähnten noch eine weitere Möglichkeit, eine Prozedur direkt im VBA-Editor auszuführen, nämlich durch Drücken der Taste F8. Die Prozedur wird dann direkt im Debugger gestartet und hält in der ersten Zeile des Programms an. Wenn wir diese beiden Tatbestände kombinieren, erhalten wir ein einfaches Mittel, Funktionen schneller zu debuggen. Wir veranschaulichen dies an Beispiel 13.1.

Beispiel 13.1 *Testprozedur für die Funktion Berechnung der Umsatzsteuer aus dem Nettobetrag*

```
'Dateiname: testProzedur.xls
Sub teste()
    Dim nettobetrag As Double
    Dim umsatzsteuer As Double
    nettobetrag = 129
    umsatzsteuer = berechneUmsatzsteuer(nettobetrag)
    MsgBox ("Umsatzsteuer: " & umsatzsteuer)
End Sub
```

[1] oder nicht clicken, aber dann passiert auch nichts.
[2] Der Debugger ermöglicht es dem Programmierer, das Programm in Einzelschritten auszuführen, und dabei den Inhalt der Variablen zu kontrollieren. Dies erleicht die Fehlersuche ungemein. Ein kurze Einführung in die Benutzung des Debuggers finden Sie in den Videos auf unserer Homepage.

```
Function berechneUmsatzsteuer(nettobetrag As Double) As Double
    Const UMSATZSTEUERSATZ As Double = 0.19
    Dim umsatzsteuer As Double
    umsatzsteuer = nettobetrag * UMSATZSTEUERSATZ
    berechneUmsatzsteuer = umsatzsteuer
End Function
```

Hier schreiben wir zunächst eine Prozedur in der wir allen Variablen[3], die wir der zu testenden Funktion übergeben wollen, Werte geben. Dann rufen wir die Funktion mit diesen Variablen in der Übergabeliste auf und schauen uns das Ergebnis an. Sollten Sie debuggen wollen, platzieren Sie den Cursor in die Prozedur und drücken F8. Dann können Sie im Einzelschritt durch Prozedur und Funktion gehen. Dies machen Sie so lange, bis Sie sicher sind, dass die Funktion richtig ist.

13.2 Aufruf von Prozeduren aus Prozeduren oder Funktionen

Wir können Prozeduren, genau wie Funktionen, aus Prozeduren oder Funktionen aufrufen. Wir veranschaulichen dies an Beispiel 13.2. In diesem Beispiel ersetzen wir die Funktion aus Beispiel 13.1 durch eine Prozedur.

Beispiel 13.2 *Testprozedur für die Funktion Berechnung der Umsatzsteuer aus dem Nettobetrag*

```
'Dateiname: prozedurAusProzedur.xls
Sub teste()
    Dim nettobetrag As Double
    Dim mehrwertsteuer As Double
    nettobetrag = 129
    Call berechneMehrwertsteuer(nettobetrag, mehrwertsteuer)
    MsgBox ("Mehrwertsteuer: " & mehrwertsteuer)
End Sub

Sub berechneMehrwertsteuer(nettobetrag As Double, mehrwertsteuer As Double)
    Const MEHRWERTSTEUERSATZ As Double = 0.19
    mehrwertsteuer = nettobetrag * MEHRWERTSTEUERSATZ
End Sub
```

Der erste (und einzige) Unterschied zwischen Prozeduren und Funktionen ist, dass Prozeduren keinen Wert über ihren Namen zurückgeben können. Eine Zeile, wie

```
umsatzsteuer=berechneUmsatzsteuer(nettobetrag)
```

ist Funktionen vorbehalten, denn nur Funktionen können einen Wert über ihren Namen zurückgeben. Ein Prozeduraufruf hingegen steht immer alleine in einer Zeile, ohne Variable und Gleichheitszeichen davor. Der Prozeduraufruf in Beispiel 13.2 ist die Zeile:

```
call berechneUmsatzsteuer(nettobetrag, umsatzsteuer)
```

Prozeduraufrufe erfolgen also durch das Schlüsselwort *call* gefolgt vom Prozedurnamen und den Übergabeparametern. Und nun lernen Sie etwas Neues: Übergabeparameter kann man nicht nur nutzen, um dem aufgerufenen Programm Werte zu geben, Übergabeparameter können ihre Werte auch an das aufrufende Programm zurückgeben. In Beispiel 13.2 wird die Variable *nettobetrag* benutzt, um den Nettobetrag an die Prozedur zu übergeben, die Variable *umsatzsteuer* hingegen, um die ausgerechnete Umsatzsteuer von der Prozedur zurück zu bekommen.
Beachten Sie bitte, dass Funktionen sich gleich verhalten. Ändern Sie in einer Funktion den Wert einer Übergabevariable, so wird diese Änderung auch im Hauptprogramm durchgeführt. Wir illustrieren dies an 13.3 und 13.4. Wir wollen hier den Bruttobetrag und die Umsatzsteuer berechnen und an das aufrufende Programm zurück geben. Zunächst die Realisierung mit einer Prozedur:

Beispiel 13.3 *Testprozedur für die Funktion Berechnung der Umsatzsteuer aus dem Nettobetrag*

[3] hier nur einer, dem Nettobetrag.

```
'Dateiname: prozedurAusProzedur2.xls
Sub teste()
    Dim nettobetrag As Double
    Dim bruttobetrag As Double
    Dim mehrwertsteuer As Double
    nettobetrag = 129
    Call berechneMehrwertsteuerUndBrutto(nettobetrag, mehrwertsteuer, bruttobetrag)
MsgBox ("Mehrwertsteuer: " & mehrwertsteuer & Chr$(13) & "Bruttobetrag: " & bruttobetrag)
End Sub

Sub berechneMehrwertsteuerUndBrutto(nettobetrag As Double, mehrwertsteuer As Double, bruttobetrag➥
     As Double)
    Const MEHRWERTSTEUERSATZ As Double = 0.19
    mehrwertsteuer = nettobetrag * MEHRWERTSTEUERSATZ
    bruttobetrag = nettobetrag + mehrwertsteuer
End Sub
```

Im Unterschied zu Beispiel 13.2 müssen wir hier zwei Werte zurückgeben, die Umsatzsteuer und den Bruttobetrag. Die Erweiterung der Lösung mit der Prozedur ist ganz einfach. Wir erweitern die Übergabeliste um die Variable *bruttobetrag*, und rechnen in der Prozedur *berechneUmsatzsteuerUndBrutto* zusätzlich den Bruttobetrag aus. Das *Chr$(13)* sorgt für einen Zeilenvorschub in der *MsgBox*. Umsatzsteuer und Bruttobetrag werden also untereinander dargestellt. Betrachten wir nun die Lösung mit einer Funktion:

Beispiel 13.4 *Testprozedur für die Funktion Berechnung der Umsatzsteuer aus dem Nettobetrag*

```
'Dateiname: funktionAusProzedurZweiUebergaben.xls
Sub teste()
    Dim nettobetrag As Double
    Dim bruttobetrag As Double
    Dim mehrwertsteuer As Double
    nettobetrag = 129
    mehrwertsteuer = berechneMehrwertsteuerUndBrutto(nettobetrag, bruttobetrag)
    MsgBox ("Mehrwertsteuer: " & mehrwertsteuer & Chr$(13) & "Bruttobetrag: " & bruttobetrag)
End Sub

Function berechneMehrwertsteuerUndBrutto(nettobetrag As Double, bruttobetrag As Double) As Double
    Const MEHRWERTSTEUERSATZ As Double = 0.19
    Dim mehrwertsteuer As Double
    mehrwertsteuer = nettobetrag * MEHRWERTSTEUERSATZ
    bruttobetrag = nettobetrag + mehrwertsteuer
    berechneMehrwertsteuerUndBrutto = mehrwertsteuer
End Function
```

Bei Funktionen wird ja ein Wert über den Funktionsnamen zurückgegeben. In unserem bisherigen Beispiel war das die Umsatzsteuer. Dabei bleiben wir. Fehlt noch der Bruttobetrag. Den geben wir über die Parameterliste zurück. So entsteht der Aufruf:

```
umsatzsteuer = berechneUmsatzsteuerUndBrutto(nettobetrag, bruttobetrag)
```

Beachten Sie: Wenn Sie eine Funktion in eine Zelle der Tabellenkalkulation einbinden, dann werden die Inhalte der Zellbezüge bei Änderungen der Übergabevariablen **nicht** geändert. Hier wäre das auch nicht besonders sinnvoll. Denn wenn sich der Zellbezug einer benutzerdefinierten Funktion ändert, führt die Tabellenkalkulation die Funktion erneut aus. Ändert sich dadurch wieder der Zellbezug, führt die Tabellenkalkulation die Funktion erneut aus. Ändert sich dadurch wieder der Zellbezug, usw. Da würde dann nur der Task Manager helfen.

Generell lässt sich nicht sagen, ob und wann Prozeduren Funktionen vorzuziehen sind. Wir benutzen Funktionen, wenn genau nur ein Wert zurückgegeben wird; Prozeduren nehmen wir immer dann, wenn mehrere Werte an das aufrufende Programm geliefert werden müssen.

13.3 Einbau der Prozedur zur Bestimmung der Punktegrenzen in das Notenprogramm

Wie bereits in der Einleitung zu diesem Kapitel erwähnt, besteht eine der Schwachstellen des Notenprogramms darin, dass die Notengrenzen an zwei Stellen des Programms ausgerechnet werden:

- Bei der Bestimmung der Note in der benutzerdefinierten Funktion.
- Bei der Darstellung der Notengrenzen durch die Ereignisprozedur.

Jede Änderung an der Logik der Bestimmung der Notengrenzen muss also zweimal implementiert werden. Wodurch man natürlich die Chance ☺ erhält, Inkonsistenzen in die Software einzubauen. Wir schreiben also eine Prozedur für die Bestimmung der Notengrenzen und rufen diese zweimal auf. Dadurch wird das Programm insgesamt kürzer, Inkonsistenzen werden vermieden und alles wird gut ;-).
Wir benutzen eine Prozedur, weil wir mehr als einen Rückgabewert haben. Beispiel 13.5 zeigt die Implementierung der Prozedur:

Beispiel 13.5 *Bestimmung der Notengrenzen in einer Prozedur*

```
'Dateiname: noteFunktionProzedur.xls
Sub ermittleNotengrenzen(maximalpunkte As Long, benoetigteProzente As Double, _
    grenze4 As Long, grenze3_7 As Long, _
    grenze3_3 As Long, grenze3_0 As Long, grenze2_7 As Long, _
    grenze2_3 As Long, grenze2_0 As Long, grenze1_7 As Long, _
    grenze1_3 As Long, grenze1_0 As Long)

    Dim punkteZweiZwischenNoten As Long
    Dim punkteDreiZwischenNoten As Long
    Dim punkteProNote As Long
    Dim spanne As Long
    Dim benoetigtePunkte As Long

    benoetigtePunkte = Int((maximalpunkte * benoetigteProzente) / 100)
    spanne = maximalpunkte - benoetigtePunkte
    punkteProNote = Int(spanne / 4)
    punkteZweiZwischenNoten = Int(punkteProNote / 2)
    punkteDreiZwischenNoten = Int(punkteProNote / 3)

    grenze4 = benoetigtePunkte
    grenze3_7 = benoetigtePunkte + punkteZweiZwischenNoten

    grenze3_3 = benoetigtePunkte + punkteProNote
    grenze3_0 = grenze3_3 + punkteDreiZwischenNoten
    grenze2_7 = grenze3_0 + punkteDreiZwischenNoten

    grenze2_3 = benoetigtePunkte + 2 * punkteProNote
    grenze2_0 = grenze2_3 + punkteDreiZwischenNoten
    grenze1_7 = grenze2_0 + punkteDreiZwischenNoten

    grenze1_3 = benoetigtePunkte + 3 * punkteProNote
    grenze1_0 = grenze1_3 + punkteZweiZwischenNoten
End Sub
```

Diese Prozedur birgt weiter keine Geheimnisse. Sie benötigt die maximal zu erreichenden Punkte und zum Bestehen notwendigen Prozente als Eingangsparameter. Zurückgegeben werden die Notengrenzen. All diese Variablen finden Sie in der Übergabeliste. Dann folgt der Code zur Bestimmung der Notengrenzen. Er wurde bereits eingehend besprochen.
Den Aufruf dieser Prozedur aus der benutzerdefinierten Funktion, die die Note der Klausurteilnehmer ermittelt, zeigt Beispiel 13.6:

Beispiel 13.6 *Aufruf der Prozedur zur Bestimmung der Notengrenzen*

```
'Dateiname: noteFunktionProzedur.xls
Function noteIfElseIF(maximalpunkte As Long, benoetigteProzente As Double, erreichtePunkte As ➥
    Long) As String
    Dim benoetigtePunkte As Long
    Dim spanne As Long
    Dim punkteProNote As Long
    Dim punkteZweiZwischenNoten As Long
    Dim punkteDreiZwischenNoten As Long
    Dim grenze4_0 As Long
    Dim grenze3_7 As Long
    Dim grenze3_3 As Long
    Dim grenze3_0 As Long
    Dim grenze2_7 As Long
    Dim grenze2_3 As Long
    Dim grenze2_0 As Long
    Dim grenze1_7 As Long
    Dim grenze1_3 As Long
    Dim grenze1_0 As Long

    Call ermittleNotengrenzen(maximalpunkte, benoetigteProzente, grenze4_0, grenze3_7, _
                    grenze3_3, grenze3_0, grenze2_7, _
                    grenze2_3, grenze2_0, grenze1_7, _
                    grenze1_3, grenze1_0)

    If erreichtePunkte >= grenze1_0 Then
        noteIfElseIF = "1"
    ElseIf erreichtePunkte >= grenze1_3 Then
        noteIfElseIF = "1,3"
    ElseIf erreichtePunkte >= grenze1_7 Then
        noteIfElseIF = "1,7"
    ElseIf erreichtePunkte >= grenze2_0 Then
        noteIfElseIF = "2"
    ElseIf erreichtePunkte >= grenze2_3 Then
        noteIfElseIF = "2,3"
    ElseIf erreichtePunkte >= grenze2_7 Then
        noteIfElseIF = "2,7"
    ElseIf erreichtePunkte >= grenze3_0 Then
        noteIfElseIF = "3"
    ElseIf erreichtePunkte >= grenze3_3 Then
        noteIfElseIF = "3,3"
    ElseIf erreichtePunkte >= grenze3_7 Then
        noteIfElseIF = "3,7"
    ElseIf erreichtePunkte >= grenze4_0 Then
        noteIfElseIF = "4"
    Else
        noteIfElseIF = "5"
    End If
End Function
```

Der Code zur Bestimmung der Notengrenzen ist einfach durch die Zeilen

```
Call ermittleNotengrenzen(maximalpunkte, benoetigteProzente, grenze4_0, grenze3_7, _
                    grenze3_3, grenze3_0, grenze2_7, _
                    grenze2_3, grenze2_0, grenze1_7, _
                    grenze1_3, grenze1_0)
```

ersetzt worden. Der Aufruf in der Ereignisprozedur (Beispiel 13.7) erfolgt völlig analog:

Beispiel 13.7 *Die Ereignisprozedur für die Punktegrenzen in Excel*

```
'Dateiname: noteFunktionProzedur.xls
Sub notenPunkteDarstellen_Click()
    Const TABELLENBLATT As Long = 1
    Const ERGEBNIS_TABELLENBLATT As Long = 2
    Dim maximalpunkte As Long
    Dim benoetigteProzente As Double
```

```
    Dim benoetigtePunkte As Long
    Dim spanne As Long
    Dim punkteProNote As Long
    Dim punkteZweiZwischenNoten As Long
    Dim punkteDreiZwischenNoten As Long
    Dim grenze4_0 As Long
    Dim grenze3_7 As Long
    Dim grenze3_3 As Long
    Dim grenze3_0 As Long
    Dim grenze2_7 As Long
    Dim grenze2_3 As Long
    Dim grenze2_0 As Long
    Dim grenze1_7 As Long
    Dim grenze1_3 As Long
    Dim grenze1_0 As Long

    maximalpunkte = Sheets(TABELLENBLATT).Cells(1, 2)
    benoetigteProzente = Sheets(TABELLENBLATT).Cells(2, 2)

    Call ermittleNotengrenzen(maximalpunkte, benoetigteProzente, grenze4_0, grenze3_7, _
                         grenze3_3, grenze3_0, grenze2_7, _
                         grenze2_3, grenze2_0, grenze1_7, _
                         grenze1_3, grenze1_0)

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 1) = "Punkte"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(1, 2) = "Note"

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 1) = "4"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(2, 2) = grenze4_0

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(3, 1) = "3,7"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(3, 2) = grenze3_7

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(4, 1) = "3,3"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(4, 2) = grenze3_3

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(5, 1) = "3"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(5, 2) = grenze3_0

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(6, 1) = "2,7"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(6, 2) = grenze2_7

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(7, 1) = "2,3"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(7, 2) = grenze2_3

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(8, 1) = "2"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(8, 2) = grenze2_0

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(9, 1) = "1,7"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(9, 2) = grenze1_7

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(10, 1) = "1,3"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(10, 2) = grenze1_3

    Sheets(ERGEBNIS_TABELLENBLATT).Cells(11, 1) = "1"
    Sheets(ERGEBNIS_TABELLENBLATT).Cells(11, 2) = grenze1_0
End Sub
```

Kapitel 14

Arrays

Wie in jeder anderen Programmiersprache gibt es auch in VBA Arrays (Felder, Vektoren). Arrays fassen Variablen ähnlichen Inhalts unter einem gemeinsamen Namen zusammen. Sie kennen Arrays unter dem Namen Vektoren bereits aus der Mathematik. Betrachten Sie dazu Abbildung 14.1:

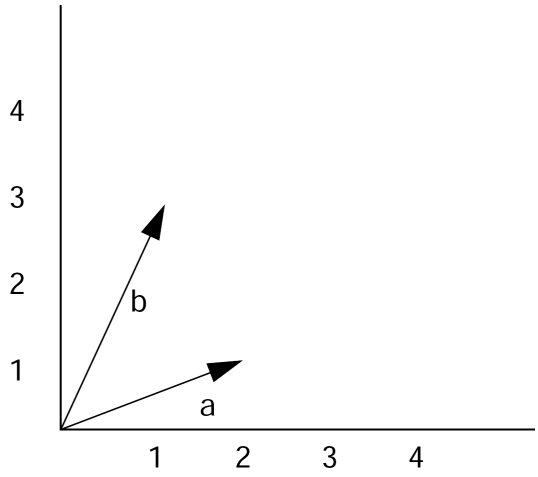


Abbildung 14.1
Koordinatensystem mit 2 Vektoren

Vektor a besitzt die Koordinaten (2, 1), b (1,3). Auf die Komponenten der Vektoren können wir über ihren Index zugreifen, so ist a(1) = 2, a(2) = 1, b(1) = 1 und b(2) = 3. Vektoren sind allerdings nicht nur in der Mathematik sinnvoll. Wir werden bei der nächsten Erweiterung unseres Provisionsbeispiels in Kapitel 17 Vektoren benutzen. Dann werden Sie auch erkennen, wie sinnvoll der Einsatz von Vektoren bei der Programmierung manchmal ist[1]. Vektoren heißen bei Entwicklern Felder oder auf Englisch Arrays. Darum werde wir im Folgenden diese Begriffe benutzen.
Doch genug der Vorrede. Schauen wir uns Arrays an einem Beispiel an. Im Beispiel schreiben wir eine Funktion, die das arithmetische Mittel dreier einzugebender Zahlen berechnet. Der Einfachheit wegen und weil diese Funktion uns auch nicht wirklich weiterbringt[2], rufen wir die neue Funktion aus einer Prozedur, wie in Kap. 13.1 beschrieben, auf.

Beispiel 14.1 *Arrays bei der Berechnung des arithmetischen Mittels*

```
'Dateiname: arrayBeispiel1.xls
Sub testeArithmetischesMittel()
    Dim meinErstesArray(2) As Double
```

[1] Zur Zeit müssen Sie dies einfach glauben.
[2] Außer, dass Sie lernen, wie man mit Arrays umgeht, was hier auch der Sinn ist.

```
    Dim arithmetischesMittel As Double
    meinErstesArray(0) = 2
    meinErstesArray(1) = 4
    meinErstesArray(2) = 0
    arithmetischesMittel = berechneArithmetischesMittel(meinErstesArray)
    MsgBox (arithmetischesMittel)
End Sub

Function berechneArithmetischesMittel(dasArray() As Double) As Double
    Dim summe As Double
    Dim arithmetischesMittel As Double
    Dim i As Long
    summe = 0
    For i = 0 To 2
        summe = summe + dasArray(i)
    Next i
    arithmetischesMittel = summe / 3
    berechneArithmetischesMittel = arithmetischesMittel
End Function
```

Arrays werden, wie einfache Variablen, mit der Anweisung *dim* erzeugt. Durch die Zeile

```
dim meinErstesArray(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 und der höchste Index angegeben wird. Dennoch verhält sich VBA hier etwas merkwürdig, weil man in anderen Programmiersprachen, wenn man drei Elemente in einem Array wünscht, dies in der Deklaration auch so sagen muss. In VBA schreibt man zwei und erhält drei. Wir sehen weiterhin, dass ein Array nur Variablen gleichen Typs aufnehmen kann, da der Typ des Feldes für das ganze Array in der Deklaration festgelegt wird. Das Array *meinErstesArray* hat also die Elemente *meinErstesArray(0)*, *meinErstesArray(1)* und *meinErstesArray(2)*. Auf die Elemente des Feldes wird über den Index zugegriffen. Wir sehen dies an den Zeilen:

```
meinErstesArray(0) = 2
meinErstesArray(1) = 4
meinErstesArray(2) = 0
```

Dann wird die Funktion zur Berechnung des arithmetischen Mittels aufgerufen. Arrays können, wie Sie hier leicht erkennen können, wie normale Variablen an Funktionen übergeben werden. Bei der Deklaration der Funktion müssen Sie ebenfalls kenntlich machen, dass der erwartete Übergabeparameter ein Array ist. In der Übergabeliste bei der Deklaration der Funktion darf allerdings nicht angegeben werden, wie viele Elemente das Array umfasst.

```
function berechneArithmetischesMittel(dasArray() As Double) As Double ' richtig
function berechneArithmetischesMittel(dasArray(2) As Double) As Double ' falsch
```

Funktionen akzeptieren immer Arrays mit beliebig vielen Elementen. Schön an Arrays ist, dass wir auf die Array-Inhalte recht einfach mit der *for-next*-Schleife zugreifen können:

```
For i = 0 To 2
    summe = summe + dasArray(i)
Next i
```

Ansonsten sollten Sie dieses kleine Programm jetzt verstehen. Zusammenfassend können wir also sagen:

Arrays sind Listen von Variablen ähnlichen Inhalts.
Dabei gilt:

- Jedes Element eines Arrays entspricht einer Variablen.
- Auf die einzelnen Elemente eines Arrays wird über einen Index zugegriffen. Der Index zählt von Null hoch.
- Einem Array kann jeder in VBA verfügbare Datentyp zugewiesen werden. Da der Datentyp aber dem Feld als Ganzem zugewiesen wird, müssen alle Elemente des Arrays von eben diesem Datentyp sein.
- Die Anzahl der Elemente des Arrays minus Eins wird bei der Deklaration des Arrays in runden Klammern an den Feldnamen angefügt. Sie können auch Arrays ohne Angabe der Anzahl der Elemente des Arrays erzeugen. Solche Arrays heißen dynamische Arrays. Dies werden wir später in diesem Kapitel besprechen.

Nicht wirklich schön an unserer Funktion ist, dass sie zwar Arrays mit beliebig vielen Elementen akzeptiert, das arithmetische Mittel aber immer nur für die ersten drei Elemente ermittelt. Aber auch das können wir ändern.
Ubound ist eine VBA-Funktion, die als Übergabeparameter ein Array erwartet und den höchsten Index zurückgibt. *Ubound(meinErstesArray)* ist also zwei.
Auch hier verhält sich VBA ziemlich merkwürdig. Andere Programmiersprachen besitzen ähnliche Funktionen, die allerdings immer die Anzahl der Elemente eines Arrays zurückgeben. Mit der Kenntnis der Funktion *Ubound* können wir nun das Programm zur Ermittlung des arithmetischen Mittels wie folgt abwandeln:

Beispiel 14.2 *Eine Funktion zur Ermittlung des arithmetischen Mittels bei Arrays mit beliebig vielen Elementen*

```
'Dateiname: arrayBeispiel2.xls
Function berechneArithmetischesMittel(dasArray() As Double) As Double
    Dim summe As Double
    Dim arithmetischesMittel As Double
    Dim i As Long
    Dim obereGrenze As Long
    obereGrenze = UBound(dasArray)
    summe = 0
    For i = 0 To obereGrenze
        summe = summe + dasArray(i)
    Next i
    arithmetischesMittel = summe / (obereGrenze + 1)
    berechneArithmetischesMittel = arithmetischesMittel
End Function
```

Durch die Zeile

```
obereGrenze = Ubound(dasArray)
```

ermitteln wir den Index des letzten Elementes des Arrays. Nun läuft unsere Schleife nicht mehr bis drei, sondern bis *obereGrenze*.

```
for i = 0 to obereGrenze
```

Beachten Sie, dass bei der Ermittlung des arithmetischen Mittels dann durch *obereGrenze + 1* geteilt werden muss, weil der Index des Arrays ja bei Null anfängt zu zählen. Wir können auch Arrays mit dynamische Größe erzeugen. Das sind Arrays, wo wir bei der Deklaration des Arrays nicht angeben, wie viele Elemente später auf dem Array abgelegt werden sollen. Solchen Arrays kann man während des Programmlaufs beliebig in der Größe verändern. Wir veranschaulichen das sofort an einem Beispiel:

Beispiel 14.3 *Dynamisches Array zur Berechnung mehrerer arithmetischen Mittelwerte*

```
'Dateiname: arrayBeispiel2a.xls
Sub testeArithmetischesMittel2()
    Dim meinErstesArray() As Double
    Dim arithmetischesMittel As Double
```

```
    ReDim meinErstesArray(2)
    meinErstesArray(0) = 2
    meinErstesArray(1) = 4
    meinErstesArray(2) = 0
    arithmetischesMittel = berechneArithmetischesMittel(meinErstesArray)
    MsgBox (arithmetischesMittel)
    ReDim Preserve meinErstesArray(4)
    meinErstesArray(3) = 7
    meinErstesArray(4) = 8
    arithmetischesMittel = berechneArithmetischesMittel(meinErstesArray)
    MsgBox (arithmetischesMittel)
End Sub
```

Hier deklarieren wir das Array zunächst ohne die Größe festzulegen. Damit ist es ein dynamisches Array und kann seine Größe während des Programmlaufs verändern:

```
dim meinErstesArray() As Double
```

Wir legen nun die Größe fest, dies geschieht mit dem Kommando *Redim*.

```
Redim meinErstesArray(2)
```

Dieses Array verhält sich jetzt genau so, als ob es mit der Anweisung

```
dim meinErstesArray(2) As Double
```

erzeugt worden wäre. Aber wir können jetzt an jeder beliebigen Stelle des Programms die Größe des Arrays verändern:

```
Redim preserve meinErstesArray(4)
```

Nun haben wir das Array vergrößert. Das Schlüsselwort *Preserve* bedeutet, dass die bisherigen Werte auf dem Array erhalten bleiben. Lassen wir es weg, werden die bisherigen Werte gelöscht. Dem Array können jetzt neue Werte hinzugefügt werden.

```
meinErstesArray(3) = 7
meinErstesArray(4) = 8
```

Es ist ebenfalls möglich, den Indexbereich eines Arrays explizit festzulegen. Wir zeigen dies an folgendem Beispiel:

Beispiel 14.4 *Fester Indexbereich eines Arrays*

```
'Dateiname: arrayBeispiel3.xls
Sub festeArrayGrenzen()
    Dim festeGrenzenArray(1 To 4) As String
    Dim i As Long
    festeGrenzenArray(1) = "Erstes Quartal"
    festeGrenzenArray(2) = "Zweites Quartal"
    festeGrenzenArray(3) = "Drittes Quartal"
    festeGrenzenArray(4) = "Viertes Quartal"
    For i = 1 To 4
        MsgBox (festeGrenzenArray(i))
    Next i
End Sub
```

Hier wird zunächst ein Array mit einem gegebenen Indexbereich definiert. Das Array *festeGrenzenArray* erhält die Indexe 1, 2, 3 und 4. Danach werden Werte auf dem Array abgespeichert. Anschließend werden die Werte in einer *for*-Schleife mittels MsgBoxen ausgegeben.

Kapitel 15

Fehlerbehandlung und Plausibilitätsprüfungen

In diesem Kapitel wollen wir über Fehler sprechen, die durch falsche Eingaben der Benutzer unserer Anwendungen entstehen. Betrachten wir dazu Abb. 15.1.

Im ersten Screenshot (Abb. 15.1) hat die Zelle *C1* den Wert *e*. In Zelle *D1* wird die Summe der Zellen *A1* bis *C1* berechnet. Obwohl der Inhalt von *C1* nicht einmal eine Zahl ist, wird dieser Fehler schlicht ignoriert und die Summe gebildet. Dabei verhält sich das Tabellenkalkulationsprogramm so, als ob der Benutzer anstelle von *e* eine Null eingegeben hätte. Bei unseren selbstgeschriebenen Funktionen und Prozeduren schreibt Excel, wenn die Dateitypen in der Übergabeliste nicht übereinstimmen, *#WERT!* in die Zelle mit dem Funktionsbezug, so dass der Fehler wenigstens sichtbar ist (vgl. Abb. 15.2).

Darüber hinaus gibt es auch Fehler, die VBA selber gar nicht erkennen kann:

- In unserem Provisionsbeispiel ergeben negative Verkaufsbeträge wenig Sinn. Dass ein Verkaufsbetrag genau so groß ist, wie der gesamte geplante Umsatz, ist ebenfalls wenig wahrscheinlich.
- Im Notenbeispiel müssen alle Eingaben positive Zahlen sein, im Gegensatz zum Provisionsbeispiel ist aber die Null erlaubt.
- Im Notenbeispiel können wir prüfen, ob die Anzahl der Punkte, die ein Student für eine Aufgabe bekommt kleiner oder gleich der Optimalanzahl an Punkten ist.

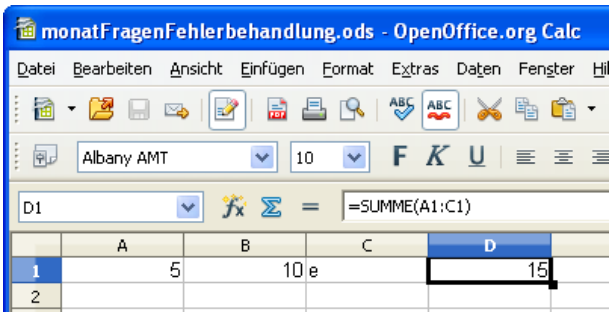


Abbildung 15.1
Fehler in Tabelleneingaben: Excel-Funktion

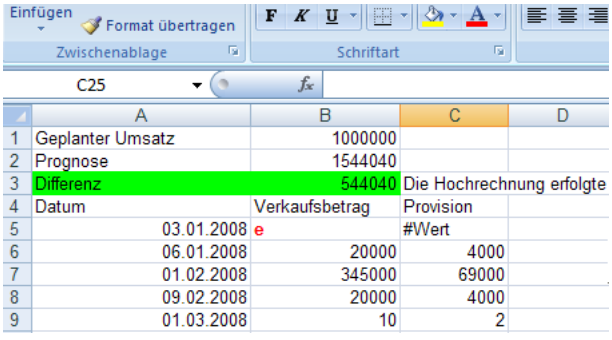


Abbildung 15.2
Fehler in Tabelleneingaben: benutzerdefinierte Funktion

Wann überprüfen wir nun unsere Eingaben? Bei dialogorientierten Programmen, bei denen der Benutzer Daten in einem Formular eingeben soll[1], wäre dies einfach: direkt nachdem der Benutzer die Daten eingegeben hat, prüfen wir die Daten und wenn die Eingabe nicht in Ordnung ist, teilen wir das dem Benutzer in einer *MsgBox* mit und beenden das Programm. Bei den Eingaben in die Zellen der Tabellenkalkulation ist das schon etwas schwieriger: Betrachten wir unser Notenbeispiel: Die Funktion, die die Note ermittelt, greift auf das Feld zu, in dem wir mit der Summenfunktion der Tabellenkalkulation die Punkte ermitteln, die der Teilnehmer erreicht hat. Da steht aber immer eine Zahl drin, egal was der Benutzer

[1] Wie das funktioniert, lernen Sie später in Kapitel 19.

eingibt. Also müssten wir den Bereich der Tabelle ermitteln, in den der Benutzer die Punkte der einzelnen Aufgaben eingetragen hat. Dazu müsste die benutzerdefinierte Funktion aber ermitteln, in welcher Zelle sie steht, damit wir die Zeile wissen, um die Punkte der einzelnen Aufgaben einzulesen. Das geht aber meines Wissens gar nicht. Und selbst wenn es gehen würde, wäre unsere Bewertungsfunktion voll von Plausibilitätsprüfungen, was sie sicher nicht übersichtlicher und schöner macht.
Wir wählen folgende Vorgehensweise: Wir schreiben eine Ereignisprozedur die die Plausibilitätsprüfungen durchführt und wenn Verstöße gegen Plausibilitäten vorkommen, diese in einem Gesamtbericht in die Tabellenkalkulation schreibt.

15.1 Plausibilitätsprüfungen der Eingaben im Tabellenblatt

Wie schon gesagt, regeln wir die Plausibilitätsprüfungen der Eingaben im Tabellenblatt über eine Ereignisprozedur.

15.1.1 Das Provisionsbeispiel

Beginnen wollen wir mit unserem Provisionsbeispiel. Wir erweitern die Benutzerschnittstelle, so dass sich das in Abb. 15.3 dargestellte Bild ergibt:

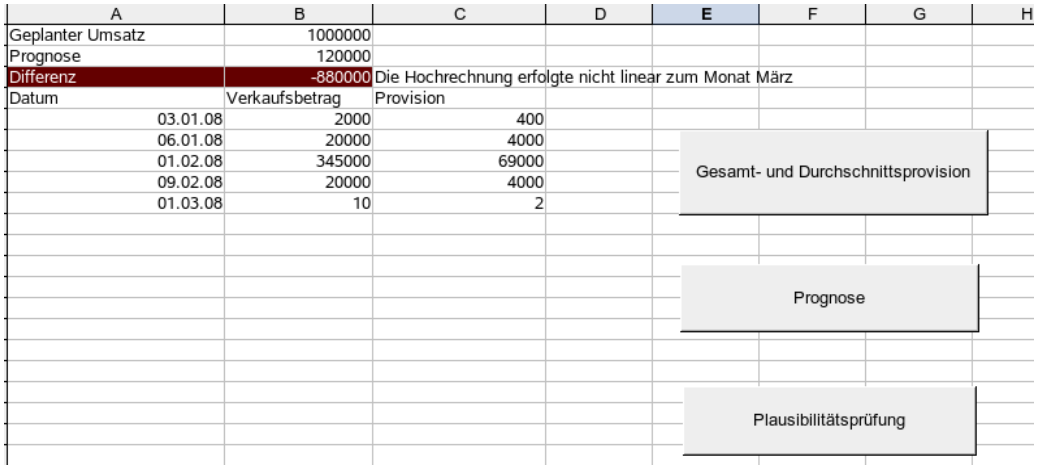


Abbildung 15.3
Erweiterung der Provisionsanwendung: Eingabeüberprüfung der Tabellendaten

Was müssen wir überprüfen? Eigentlich nur zwei Dinge: Jede Eingabe in der ersten Spalte muss ein gültiges Datum sein, in der zweiten Spalte hingegen sind nur positive Zahlen erlaubt. Das erste ist einfach zu checken, in VBA gibt es dafür die Funktion *isDate*. Für das zweite schreiben wir eine Funktion.

Beispiel 15.1 *Die Funktion istPositiveZahl*

```
'Dateiname: provisionPrognoseMonatUeberpruefung.xls
Function istPositiveZahl(testwert As String) As Boolean
    Dim testwertAlsDouble As Double
    If Not IsNumeric(testwert) Then
        istPositiveZahl = False
        Exit Function
    End If
    testwertAlsDouble = CDbl(testwert)
    If (testwertAlsDouble <= 0) Then
        istPositiveZahl = False
        Exit Function
    End If
    istPositiveZahl = True
End Function
```

Diese Funktion sollte einfach zu verstehen sein. Zunächst wird geprüft, ob der der Funktion *istPositiveZahl* übergebene Wert numerisch ist. Ist dies nicht der Fall, erhält die Funktion den Rückgabewert *false* und beendet sich. Ansonsten wird der übergebene Wert in Double umgewandelt. Nun wird überprüft, ob dieser Wert kleiner gleich 0 ist. Ist dies der Fall, wird der Rückgabewert der Funktion wieder auf *false* gesetzt und die Funktion beendet. Ansonsten, in diesem Fall war alles okay, gibt die Funktion *true* zurück.
Nun machen wir uns noch einige Gedanken darüber, wie unser Programm sich die Verstöße gegen die Plausibilitätsregeln merken soll. Das einfachste wäre, die bemerkten Fehler direkt in ein Tabellenblatt zu schreiben. Viel eleganter aber ist, alle aufgetretenen Fehler zunächst auf ein Array zu schreiben. Dieses Array können wir dann am Schluss, wenn alles überprüft ist, insgesamt ausgeben. Das ist zunächst etwas mehr Arbeit, hat aber dann den Vorteil, dass wir diese Technik in allen Tabellenblättern angepasst übernehmen können und damit dann Zeit sparen.
Nun wissen wir aber im Vorhinein nicht, wieviele falsche Eingaben getätigt worden sind. Wir können also kein Array fester Größe benutzen, wie wir das für die Monate in Kap. 17 getan haben. Vielmehr werden wir die in Kap. 14 ebenfalls beschriebenen dynamischen Arrays benutzen. In unserer Ereignisprozedur werden wir ein dynamisches Array deklarieren und dann mit der *Redim*-Anweisung ein erstes Element des Arrays erzeugen. Dann schreiben wir eine Prozedur, die den jeweilig letzten Index eines Arrays mit einem übergebenen String belegt und im Abschluss daran eine neues Element erzeugt. Betrachten wir diese Prozedur:

Beispiel 15.2 *Die Funktion schreibeNeuenFehlerInArray*

```
'Dateiname: provisionPrognoseMonatUeberpruefung.xls
Sub schreibeNeuenFehlerInArray(fehlerString As String, fehlerArray() As String)
    Dim laengeFehlerArray As Long
    laengeFehlerArray = UBound(fehlerArray)
    fehlerArray(laengeFehlerArray) = fehlerString
    ReDim Preserve fehlerArray(laengeFehlerArray + 1)
End Sub
```

Die Prozedur *schreibeNeuenFehlerInArray* erwartet zwei Übergabeparameter: Das Array mit den bisherigen Fehlern und den neuen Fehler. Die Prozedur ermittelt nun den höchsten Index des Arrays mit den Fehlern, auf diesen Index wird der neue Fehler geschrieben:

```
laengeFehlerArray = UBound(fehlerArray)
fehlerArray(laengeFehlerArray) = fehlerString
```

Dann wird das Array um eins vergrößert, wobei die alten Werte erhalten bleiben:

```
ReDim Preserve fehlerArray(laengeFehlerArray + 1)
```

Wir beginnen direkt mit der Darstellung des Codes:

Beispiel 15.3 *Plausibilitätsprüfung des Provisionsbeispiels*

```
'Dateiname: provisionPrognoseMonatUeberpruefung.xls
Sub plausibilitaeten_Click()
    Dim testwert As String
    Dim i As Long
    Dim fehlerString As String
    Dim fehlerArray() As String
    Dim letzteBesetzteZeile As Long

    Const DATUMSSPALTE As Long = 1
    Const VERKAUFSBETRAGSPALTE As Long = 2
    Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Long = 5
    Const ERSTE_ZEILE_MIT_FEHLER_INFO As Long = 21
    Const TABELLENBLATT As Long = 1
    Const TABELLENBLATT_FEHLER As Long = 3
    Const FEHLERSPALTE As Long = 3

    ReDim fehlerArray(0)
    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(DATUMSSPALTE, _
                          ERSTE_ZEILE_MIT_VERKAUFSBETRAG, TABELLENBLATT)
```

```
    For i = ERSTE_ZEILE_MIT_VERKAUFSBETRAG To letzteBesetzteZeile
        testwert = Sheets(TABELLENBLATT).Cells(i, DATUMSSPALTE)
        If Not IsDate(testwert) Then
            fehlerString = "Kein Datum in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
        testwert = Sheets(TABELLENBLATT).Cells(i, VERKAUFSBETRAGSPALTE)
        If Not istPositiveZahl(testwert) Then
            fehlerString = "Keine positive Zahl als Verkaufsbetrag in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
    Next i
    If UBound(fehlerArray) > 0 Then
        MsgBox ("Plausibilitätsverletzungen sind aufgetreten, ein Protokoll wird geschrieben!")
        Call schreibeFehlerArray(fehlerArray, TABELLENBLATT_FEHLER, ERSTE_ZEILE_MIT_FEHLER_INFO, ➥
            FEHLERSPALTE)
    Else
        MsgBox ("Alles klar")
    End If
End Sub
```

Beachten Sie hier, dass durch die beiden Anweisungen

```
dim fehlerArray() as String
```

und

```
Redim fehlerArray(0)
```

ein dynamisches Array mit einem Element angelegt wird. Dies liegt daran, dass die Indexzählung eines Arrays, wenn wir nicht, wie bei dem Monatsarray aus Kap. 17, feste Indexgrenzen angeben, immer bei Null beginnt. Die Prozedur ermittelt die letzte besetzte Zeile und geht dann in einer *for*-Schleife alle Zeilen bis dorthin durch. In jeder Zeile wird geprüft, ob in der Datumsspalte ein gültiges Datum und in der Verkaufsbetragsspalte eine positive Zahl eingetragen ist. Wenn dem nicht so ist, wird die Prozedur *schreibeNeuenFehlerInArray* aufgerufen. Diese schreibt den neuen Fehler in das *fehlerArray* und vergrößert das *fehlerArray* dann um ein Element.
Beachten Sie, dass die Inhalte der Zellen auf eine String-Variable eingelesen werden:

```
Dim testwert As String
```

und z.B. für den Verkaufsbetrag:

```
testwert = Sheets(TABELLENBLATT).Cells(i, VERKAUFSBETRAGSPALTE)
```

Dies ist notwendig, weil wir, während die Plausibilitätskontrollen laufen, noch nicht wissen, ob die Benutzer Eingabefehler gemacht haben. Ein Benutzer kann ja durchaus, durch Vertippen in eine Zelle für den Verkaufsbetrag einen Buchstaben eingeben. Sinn der Plausibilitätsprüfungen ist ja gerade, festzustellen, ob solche Fehler passiert sind und den Benutzer dann darauf hinzuweisen. Würden wir die Verkaufsbetrag-Eingaben auf eine Zahl-Variable (*Long*, *Long* oder *Double*) einlesen, würde das Programm beim Einlesen eines Buchstabens abstürzen. Das ist aber gerade nicht das, was wir wollen. Wir wollen solche Fehleingaben feststellen und den Benutzer dann darauf hinweisen. Hierzu bieten sich String-Variablen an, denn auf einer Stringvariable lassen sich beliebige Zeichenketten speichern. Die Vorgehensweise, wenn wir etwas auf numerischen Inhalt prüfen wollen, ist also immer:

- Wir lesen die Werte aus den Zellen der Tabellenkalkulation auf eine *String*-Variable ein.
- Dann checken wir, ob es sich um eine Zahl handelt.

Nach der Schleife wird überprüft, ob ein Fehler gefunden wurde. Dies ist genau dann der Fall, wenn das *fehlerArray* vergrößert wurde. Wir checken also, ob der größte Index (*UBound*) von *fehlerArray* größer als 0 ist. Ist dies der Fall, wird eine *MsgBox* aufgeblendet, die dies mitteilt und die Prozedur aufgerufen, die das *fehlerArray* in die Tabelle schreibt. Bleibt also noch die Prozedur *schreibeFehlerArray* (Abb. 15.4). Die Übergabeparameter sind das Array, das geschrieben werden soll, die Nummer des Tabellenblatts, in das die Ausgabe erfolgen soll, die erste Zeile, ab der die Ausgabe erfolgt

und die Spalte, in die geschrieben wird. Das Programm läuft dann in einer Schleife über das Array und gibt die Werte in untereinander liegenden Zellen aus.

Beispiel 15.4 *Die Prozedur schreibeFehlerArray*

```
'Dateiname: provisionPrognoseMonatUeberpruefung.xls
Sub schreibeFehlerArray(fehlerArray() As String, tabellennummer As Long, startzeile As Long, ➥
    spalte As Long)
    Dim laengeFehlerArray As Long
    Dim i As Long
    laengeFehlerArray = UBound(fehlerArray)
    For i = 0 To laengeFehlerArray
            Sheets(tabellennummer).Cells(startzeile + i, spalte) = fehlerArray(i)
    Next i
End Sub
```

15.1.2 Das Gewinnbeispiel

Als nächstes wollen wir ein etwas komplexeres Beispiel betrachten.
Wir sollen die in Abb. 15.4 dargestellten Daten überprüfen. Folgende Eingabekontrollen sollen durchgeführt werden:

	A	B	C	D	E	F	G
1	Kunde	Produkt	Anzahl	Einkaufspreis	Softwarekategorie	Versandart	Gewinn
2	Schmid	PDF Reader	300	26,00 €	Utilities	Abholung	624,00 €
3	Meier	Kalkulation	300	82,95 €	Office	CD-Versand	1.246,55 €
4	Seran	Windows 4711	2	1.300,00 €	Betriebssysteme	Download	78,00 €
5	Müller	Windows 0815	-3	87,00 €	Betriebssysteme	Download	- 7,83 €
6	Schmidt	Writer	623	58,45 €	Utilities	CD-Versand	2.915,45 €
7							

Abbildung 15.4
Gewinnbeispiel mit Eingabefehlern

- Die Anzahl muss eine positive Zahl sein.
- Die Anzahl darf höchstens 100 sein, außer bei Software der Kategorie „Utilities“. Dort darf die Anzahl höchstens 500 sein.
- Der Einkaufspreis liegt zwischen 5 und 250 Euro.
- Die Spalte Versandart ist entweder mit „Download“ oder mit „CD-Versand“ gefüllt.

Unsere Anwendung muss also das in Abb. 15.5 dargestellte Fehlerprotokoll erzeugen.

	A	B	C	D	E	F	G
1							
2							
3		Falscher Wert in der Versandartspalte in Zeile 2					
4		Anzahl liegt nicht unterhalb der zugelassenen Grenze in Zeile 3					
5		Einkaufspreis liegt nicht in den zugelassenen Grenzen in Zeile 4					
6		Anzahl nicht positiv in Zeile 5					
7		Anzahl liegt nicht unterhalb der bei Utilties zugelassenen Grenze in Zeile 6					
8							

Abbildung 15.5
Fehlerprotokoll für das Gewinnbeispiel mit Eingabefehlern

Die Ausgabe der Verstöße gegen die Plausibilitätskontrollen erfolgt in Tabellenblatt 3. Sind alle Eingaben okay, wird eine MsgBox mit der Meldung „alles okay“ aufgeblendet, ansonsten eine MsgBox mit dem Hinweis darauf, dass Verstöße

gegen die Plausibilitäten vorliegen. Die Schaltfläche zum Start des Plausibilitätenprogramms soll grundsätzlich in jedes Tabellenblatt einbaubar sein.
Schauen wir uns nun die Lösung an:

Beispiel 15.5 *Das Programm zur Plausibilitätsprüfung*

```
Sub plausibilitaeten_click()
    Const ANZAHL_SPALTE As Long = 3
    Const EINKAUFSPREIS_SPALTE As Long = 4
    Const KATEGORIE_SPALTE As Long = 5
    Const VERSANDART_SPALTE As Long = 6
    Const GEWINN_SPALTE As Long = 7

    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Const TABELLENBLATT As Long = 1

    Const TABELLENBLATT_PLAUSIBILITAETEN As Long = 3
    Const ERSTE_ZEILE_MIT_FEHLER_INFO As Long = 3
    Const SPALTE_MIT_FEHLER_INFO As Long = 2

    Const MIN_GRENZE_EINKAUFSPREIS As Double = 5
    Const MAX_GRENZE_EINKAUFSPREIS As Double = 250
    Const MAX_GRENZE_ANZAHL As Long = 100
    Const MAX_GRENZE_ANZAHL_UTILITIES As Long = 500

    Dim testwert As String
    Dim i As Long
    Dim fehlerString As String
    Dim fehlerArray() As String
    Dim letzteBesetzteZeile As Long

    ReDim fehlerArray(0)

    Dim anzahl As Long
    Dim einkaufspreis As Double

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(EINKAUFSPREIS_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        testwert = Sheets(TABELLENBLATT).Cells(i, ANZAHL_SPALTE)
        If Not istPositiveZahl(testwert) Then
            fehlerString = "Anzahl nicht positiv  in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        Else
            anzahl = CLng(testwert)
            If Sheets(TABELLENBLATT).Cells(i, KATEGORIE_SPALTE) = "Utilities" Then
                If Not istKleinerGleichGrenzeLong(anzahl, MAX_GRENZE_ANZAHL_UTILITIES) Then
                    fehlerString = "Anzahl liegt nicht unterhalb der bei Utilties zugelassenen ➥
                        Grenze in Zeile " & i
                    Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
                End If
            Else
                If Not istKleinerGleichGrenzeLong(anzahl, MAX_GRENZE_ANZAHL) Then
                fehlerString = "Anzahl  liegt nicht unterhalb der zugelassenen Grenze in Zeile " ➥
                    & i
                Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
                End If
            End If
        End If
        testwert = Sheets(TABELLENBLATT).Cells(i, VERSANDART_SPALTE)
        If Not istCDVersandOderDownload(testwert) Then
            fehlerString = "Falscher Wert in der Versandartspalte in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If

        testwert = Sheets(TABELLENBLATT).Cells(i, EINKAUFSPREIS_SPALTE)
```

```
        If Not istPositiveZahl(testwert) Then
            fehlerString = "Einkaufspreis keine positive Zahl in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        Else
            einkaufspreis = CDbl(testwert)
            If Not liegtZwischenGrenzenDouble(einkaufspreis, MIN_GRENZE_EINKAUFSPREIS, ➥
                MAX_GRENZE_EINKAUFSPREIS) Then
                fehlerString = "Einkaufspreis liegt nicht in den zugelassenen Grenzen in Zeile " ➥
                    & i
                Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
            End If
        End If
    Next i
    If UBound(fehlerArray) > 0 Then
        MsgBox ("Plausibilitätsverletzungen sind aufgetreten, ein Protokoll wird geschrieben!")
        Call schreibeFehlerArray(fehlerArray, TABELLENBLATT_PLAUSIBILITAETEN, ➥
            ERSTE_ZEILE_MIT_FEHLER_INFO, SPALTE_MIT_FEHLER_INFO)
    Else
        MsgBox ("Alles klar")
    End If
End Sub
```

Gehen wir die Lösung im Einzelnen durch. Zunächst definieren wir einige Konstanten. Die ersten Konstanten definieren die Spalten, die wir im weiteren Verlauf benötigen:

```
Const ANZAHL_SPALTE As Long = 3
Const EINKAUFSPREIS_SPALTE As Long = 4
Const KATEGORIE_SPALTE As Long = 5
Const VERSANDART_SPALTE As Long = 6
Const GEWINN_SPALTE As Long = 7
```

Dies ist sinnvoll, weil sich das Layout des Tabellenblatts verändern kann. Nehmen wir an, ein Entscheider im Unternehmen möchte die Spalte mit der Versandart vor der Spalte mit der Softwarekategorie stehen haben. Dann müssen wir nur die Werte der Konstanten *KATEGORIE_SPALTE* und *VERSANDART_SPALTE* anpassen. Aus dem gleichen Grund legen wir die Zeile, in der die Daten beginnen und das Tabellenblatt, in dem die Daten stehen auf Konstanten.

```
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
Const TABELLENBLATT As Long = 1
```

Auch das Zieltabellenblatt (also das Tabellenblatt, in dem die Plausibilitätsverstöße dargestellt werden), die Zeile in der die Darstellung der Fehler beginnen soll, sowie die Spalte in der die Fehler dargestellt werden, werden auf Konstanten abgelegt:

```
Const TABELLENBLATT_PLAUSIBILITAETEN As Long = 3
Const ERSTE_ZEILE_MIT_FEHLER_INFO As Long = 3
Const SPALTE_MIT_FEHLER_INFO As Long = 2
```

Zum Schluss legen wir die einzuhaltenden Grenzen auf Konstanten. Auch diese können sich ändern, und auch hier wollen wir dann nur Konstanten anpassen.

```
Const MIN_GRENZE_EINKAUFSPREIS As Double = 5
Const MAX_GRENZE_EINKAUFSPREIS As Double = 250
Const MAX_GRENZE_ANZAHL As Long = 100
Const MAX_GRENZE_ANZAHL_UTILITIES As Long = 500
```

Dann erfolgt die Deklaration der notwendigen Variablen:

```
Dim testwert As String
Dim i As Long
Dim fehlerString As String
Dim fehlerArray() As String
Dim letzteBesetzteZeile As Long
```

Die Variablen *fehlerString* und *fehlerArray* benötigen wir für die bereits in den Beispielen 15.3 und 15.2 dargestellte Fehlerbehandlung. Auf die Variable *testwert* lesen wir die zu überprüfenden Werte aus den Zellen ein. *testwert* ist eine *String*-Variable und kann daher beliebige Werte aufnehmen, ohne dass es zu einem Fehler kommen kann (vgl. die ausführliche Diskussion dazu in Beispiel 15.3).
Dann erfolgt die Initialisierung der Variablen *fehlerArray*.

```
ReDim fehlerArray(0)
```

Dies wurde ebenfalls in Beispiel 15.2 ausführlich diskutiert. Den Sinn der beiden Variablen

```
Dim anzahl As Long
Dim einkaufspreis As Double
```

wird erläutert, sobald sie im Programmablauf genutzt werden.
Wir bestimmen die letzte besetzte Zeile, also die Zeile, bis zu der unsere Schleife laufen muss.

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(GEWINN_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
```

Zur Bestimmung der letzten besetzten Zeile können wir grundsätzlich jede Spalte des Tabellenblatts bis zur Spalte G nehmen, weil alle Spalten durchgängig gefüllt sind. Die Zeile, in der wir die Bestimmung der letzten besetzten Zeile starten, ist natürlich die Zeile 2, weil hier die Daten beginnen. Diese hatten wir ja auf der Konstanten *ERSTE_ZEILE_MIT_INFORMATIONEN* abgelegt. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteZeileInSpalte* hatten wir in Kapitel 9.3 eingehend besprochen, so dass auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun startet unsere Schleife. Sie läuft natürlich von Zeile 2 (*ERSTE_ZEILE_MIT_INFORMATIONEN*) zur letzten besetzten Zeile, da ja jeder Datensatz behandelt werden muss.

```
For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
```

Wir beginnen mit der Überprüfung, ob es sich bei der Anzahl um eine positive Zahl handelt. Dazu lesen wir zunächst die Anzahl im gerade behandelten Datensatz auf die Variable *testwert* ein. Wie in Beispiel 15.3 ausführlich beschrieben müssen wir dies tun, um einen Programmabsturz während der Plausibilitätskontrollen zu vermeiden.

```
    testwert = Sheets(TABELLENBLATT).Cells(i, ANZAHL_SPALTE)
```

Wegen der Anforderung die Schaltfläche zum Programmstart in jedes Zabellenblatt einbauen zu können, ist nicht mehr sichergestellt, dass sich Daten und Schaltfläche im gleichen Tabellenblatt befinden. Wir müssen also unter Angabe des Tabellenblattes auf die Zelle zugreifen. Daher ist die Angabe *Sheets(TABELLENBLATT).Cells(i, ANZAHL_SPALTE)* notwendig. *Cells(i, ANZAHL_SPALTE)* reicht nicht mehr. Diese Vorgehensweise ist natürlich auch bei allen anderen Zellzugriffen in diesem Programm notwendig.
Nun prüfen wir, ob es sich bei dem eingelesenen Wert um eine positive Zahl handelt. Dazu benutzen wir die Funktion *istPositiveZahl* aus Beispiel 15.1. Die Funktion *istPositiveZahl* wurde ebenda ausführlich erläutert. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert. Sollte es sich nicht um eine positive Zahl handeln, führen wir die in Kap. 15.1.1 dargestellte Fehlerbehandlung durch. Wir schreiben den aufgetretenen Fehler und die Zeile in der der Fehler aufgetreten ist auf das Fehlerarray.

```
    If Not istPositiveZahl(testwert) Then
        fehlerString = "Anzahl nicht positiv  in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
```

Als nächstes überprüfen wir, ob die Anzahl innerhalb der vorgegebenen Grenzen liegt:

- Die Anzahl darf höchstens 100 sein, außer bei Software der Kategorie „Utilities“. Dort darf die Anzahl höchstens 500 sein.

Wir stellen sofort fest, dass diese Überprüfung nur dann Sinn ergibt, wenn es sich bei dem Wert für Anzahl im gerade behandelten Datensatz um eine Zahl handelt. Denn wenn es sich um keine Zahl handelt, dann benötigen wir auch keine weiterführende Prüfung, weil wir ja schon einen Fehler festgestellt haben. Darüberhinaus müssen wir, wenn wir auf jeden Fall richtige Vergleiche durchführen wollen, den gerade eingelesenen Wert wieder in eine Zahl umwandeln. Dies illustrieren folgende kurze Beispiele:

Beispiel 15.6 *Vergleiche 1: Stringvariable mit Longvariable*

```
Sub vergleichenStringLong()
    Dim s1 As String
    Dim i1 As Long
    s1 = "a"
    i1 = 9
    If s1 < i1 Then
        MsgBox ("9 ist größer als a")
    Else
        MsgBox ("9 ist kleiner als a")
    End If
End Sub
```

Wenn wir dieses Programm starten, erhalten wir die Fehlermeldung:

> Laufzeitfehler 12: Typen unverträglich

VBA kann also keine auf String-Variablen gespeicherten Buchstaben mit auf Long-Variablen gespeicherten Zahlen vergleichen. Dies müssen wir in unserem Plausibilitätenprogramm beachten.

Beispiel 15.7 *Vergleiche 2: Stringvariable mit Stringvariable*

```
Sub vergleichen1()
    Dim s1 As String
    Dim s2 As String
    s1 = 9
    s2 = 12
    If s2 < s1 Then
        MsgBox ("9 ist größer als 12")
    Else
        MsgBox ("12 ist größer als 9")
    End If
End Sub
```

Wenn wir dieses Programm ablaufen lassen, erhalten wir als Ergebnis:

> 9 ist größer als 12

Dies liegt an der Vorgehensweise von VBA bei Stringvergleichen: Strings werden Zeichen für Zeichen verglichen. Sobald das erste Mal eine Reihenfolge festgestellt werden kann, ist der Vergleich beendet. In Beispiel 15.7 bedeutet dies: Zuerst wird das Zeichen 9 des Strings *s1* mit dem Zeichen 1 des Strings *s2* verglichen. 9 ist aber in der bei Strings benutzten lexikalischen Reihenfolge größer als 1. Der Vergleich ist somit beendet, *s1* ist größer als *s2*.
Wir lösen dieses Problem dadurch, dass wir den Inhalt der Variablen *testwert*, bevor wir mit den Plausibilitätsprüfungen fortfahren, in eine Zahl umwandeln. Dies können wir allerdings nur dann, wenn der Inhalt der Variablen *testwert* auch wirklich eine Zahl ist. Wie machen wir das?

```
If Not istPositiveZahl(testwert) Then
    fehlerString = "Anzahl nicht positiv  in Zeile " & i
    Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
else
    anzahl = CLng(testwert)
```

Sie sehen, die Lösung ist einfach. Wir hatten ja schon (mit einer *if*-Anweisung) festgestellt, ob der Inhalt der Variablen *testwert* eine positive Zahl ist. Wir nehmen jetzt einfach einen *else*-Teil auf. In diesen verzweigt VBA, wenn der Inhalt von *testwert* eine positive Zahl ist. Hier wandeln wir zunächst den Inhalt von Testwert in einen *Long*-Wert um. Dazu dient die Anweisung *CLng*[2]. Nun sehen wir auch den Sinn der Deklaration:

[2] Abkürzung von *Convert to Long*

```
Dim anzahl As Long
```

Wir benötigen die Variable *anzahl*, um den in Long umgewandelten Wert der Spalte Anzahl des gerade bearbeiteten Datensatzes zu speichern. Nun müssen wir noch den Größenvergleich durchführen. Um unser Plausibilitätenprogramm kürzer zu gestalten und wegen der Wiederverwendung in anderen Programmen schreiben wir dazu eine Funktion:

Beispiel 15.8 *Die Funktion istKleinerGleichGrenze für Long-Variablen*

```
Function istKleinerGleichGrenzeLong(testwert As Long, grenze As Long) As Boolean
  If testwert <= grenze Then
    istKleinerGleichGrenzeLong = True
  Else
    istKleinerGleichGrenzeLong = False
  End If
End Function
```

Diese Funktion sollten Sie sofort verstehen. Wie in Beispiel 10.2 in Kap. 10.1 ausführlich diskutiert, funktioniert diese Funktion nur für *Long*-Variablen, da beide Übergabeparameter vom Typ *Long* sind. Daher heißt unsere Funktion *istKleinerGleichGrenzeLong*.
Diese Funktion müssen wir mit der Grenze *MAX_GRENZE_ANZAHL_UTILITIES* aufrufen, wenn in der Spalte der Softwarekategorien „Utilities“ steht und mit der Grenze *MAX_GRENZE_ANZAHL* sonst. Für so etwas haben wir aber das *if*:

```
    testwert = Sheets(TABELLENBLATT).Cells(i, ANZAHL_SPALTE)
    If Not istPositiveZahl(testwert) Then
        fehlerString = "Anzahl nicht positiv  in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    Else
        anzahl = CLng(testwert)
        If Sheets(TABELLENBLATT).Cells(i, KATEGORIE_SPALTE) = "Utilities" Then
            If Not istKleinerGleichGrenzeLong(anzahl, MAX_GRENZE_ANZAHL_UTILITIES) Then
                fehlerString = "Anzahl liegt nicht unterhalb der bei Utilties zugelassenen Grenze➥
                     in Zeile " & i
                Call schreibeNeuenFehlerInArray(fehlerStringLong, fehlerArray)
            End If
        Else
            If Not istKleinerGleichGrenzeLong(anzahl, MAX_GRENZE_ANZAHL) Then
                fehlerString = "Anzahl  liegt nicht unterhalb der zugelassenen Grenze in Zeile " ➥
                    & i
                Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
            End If
        End If
    End If
```

Damit sind die Plausibilitätskontrollen für den Inhalt der Spalte „Anzahl“ beendet.
Kommen wir nun zu den Plausibilitätskontrollen für den Inhalt der Spalte „Einkaufspreis“. Hier sollen wir überprüfen, ob der Einkaufspreis zwischen 5 und 150 Euro liegt. Wie Beispiel 15.6 zeigt, ist das aber nur möglich, wenn in der Spalte „Einkaufspreis“ eine Zahl steht. Sonst stürzt unser Plausibilitätenprogramm bei der Prüfung mit der Fehlermeldung

> Laufzeitfehler 12: Typen unverträglich

ab. Das ist kein erwünschtes Verhalten. Ein Plausibilitätenprogramm soll Fehler oder Plausibilitätsverletzungen melden und nicht selber bei einer solchen abstürzen. Um Abstürze zu vermeiden, müssen wir aber überprüfen, ob der Einkaufspreis im gerade bearbeiteten Datensatz eine Zahl ist. Eigentlich müssen wir genau so verfahren, wie gerade bei der Anzahl. Was bedeutet: Obwohl es in der Aufgabenstellung nicht explizit vermerkt ist, müssen wir überprüfen, ob der Einkaufspreis im gerade bearbeiteten Datensatz eine Zahl ist. Denn sonst können wir die Aufgabenstellung zu überprüfen, ob der Einkaufspreis zwischen 5 und 150 Euro liegt, gar nicht realisieren.
Allgemein können wir feststellen: Immer, wenn wir Plausibilitätsprüfungen auf Spalten, in denen Zahlen erforderlich sind, durchführen, müssen wir als Erstes prüfen, ob der gerade bearbeitete Wert überhaupt eine Zahl ist. Und dabei ist es völlig unerheblich, ob das in der Aufgabenstellung explizit verlangt wurde oder nicht.
Wie oben, schreiben wir für die Überprüfung, ob der Einkaufspreis zwischen 5 und 150 Euro liegt, eine Funktion:

Beispiel 15.9 *Die Funktion liegtZwischenGrenzen für Double-Variablen*

```
Function liegtZwischenGrenzenDouble(testwert As Double, unterGrenze As Double, oberGrenze As ➥
    Double) As Boolean
    If testwert >= unterGrenze And testwert <= oberGrenze Then
        liegtZwischenGrenzenDouble = True
    Else
        liegtZwischenGrenzenDouble = False
    End If
End Function
```

Auch diese Funktion müssten Sie sofort verstehen. Diese Funktion läuft nur für *Double*-Variablen, daher machen wir dies auch am Namen kenntlich (*liegtZwischenGrenzenDouble*).
Damit ergibt sich folgende Plausibilitätsprüfung für den Inhalt der Spalte Einkaufspreis:

```
testwert = Sheets(TABELLENBLATT).Cells(i, EINKAUFSPREIS_SPALTE)
If Not istPositiveZahl(testwert) Then
    fehlerString = "Einkaufspreis keine positive Zahl in Zeile " & i
    Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
Else
    einkaufspreis = CDbl(testwert)
    If Not liegtZwischenGrenzenDouble(einkaufspreis, MIN_GRENZE_EINKAUFSPREIS, ➥
        MAX_GRENZE_EINKAUFSPREIS) Then
        fehlerString = "Einkaufspreis liegt nicht in den zugelassenen Grenzen in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    End If
End If
```

Neu ist noch die Zeile:

```
    einkaufspreis = CDbl(testwert)
```

Hier wird die VBA-Funktion *CDbl*[3] benutzt. Sie wandelt einen String in einen Double um. Hier sehen wir auch, warum die Deklaration:

```
Dim einkaufspreis As Double
```

notwendig war.
Nun bleibt noch zu überprüfen:

- Die Spalte Versandart ist entweder mit „Download“ oder mit „CD-Versand“ gefüllt.

Auch für diese Überprüfung schreiben wir, um das Plausibilitätenprogramm kürzer unnd besser lesbar zu halten, eine eigene Funktion:

Beispiel 15.10 *Die Funktion istCDVersandOderDownload*

```
Function istCDVersandOderDownload(testwert As String) As Boolean
    If testwert = "CD-Versand" Or testwert = "Download" Then
        istCDVersandOderDownload = True
    Else
        istCDVersandOderDownload = False
    End If
End Function
```

In der Plausibilitätenanwendung erfolgt dann folgender Aufruf:

```
    testwert = Sheets(TABELLENBLATT).Cells(i, VERSANDART_SPALTE)
    If Not istCDVersandOderDownload(testwert) Then
        fehlerString = "Falscher Wert in der Versandartspalte in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    End If
```

[3] Abkürzung von *Convert to Double*

Damit sind die Plausibilitätsprüfungen beendet. Der Schluss des Programms

```
    Next i
    If UBound(fehlerArray) > 0 Then
        MsgBox ("Plausibilitätsverletzungen sind aufgetreten, ein Protokoll wird geschrieben!")
        Call schreibeFehlerArray(fehlerArray, TABELLENBLATT_PLAUSIBILITAETEN, ➥
            ERSTE_ZEILE_MIT_FEHLER_INFO, SPALTE_MIT_FEHLER_INFO)
    Else
        MsgBox ("Alles klar")
    End If
```

wurde bereits in Beispiel 15.3 erklärt.

15.1.3 Das Schlüsseldienstbeispiel

Als nächstes wollen wir ein weiteres etwas komplexeres Beispiel betrachten.
Wir sollen die in Abb. 15.6 dargestellten Daten überprüfen.

	A	B	C	D	E
1	Kunde	gefahrene km	Arbeitszeit (Min)	Nachtzuschlag	Preis
2	Meyer	-20	20	nein	55
3	Müller	10	2		19
4	Schmidt	100	25	ja	126
5	Schmid	15	35		100
6	Seran	10	230	ja	665
7	Fischer	7	7		29

Abbildung 15.6
Schlüsseldienstbeispiel mit Eingabefehlern

Folgende Eingabekontrollen sollen durchgeführt werden:

- Gefahrene Kilometer muss eine positive Zahl sein.
- Gefahrene Kilometer ist kleiner gleich 70.
- Die Arbeitszeit liegt zwischen 5 und 180 Minuten.
- Die Spalte Nachtzuschlag ist entweder gar nicht oder mit „ja“ gefüllt.

Unsere Anwendung muss also das in Abb. 15.7 dargestellte Fehlerprotokoll erzeugen. Die Ausgabe der Verstöße ge-

	A	B	C	D	E	F	G	H
1								
2		Falscher Wert in der Nachtzuschlagspalte in Zeile 2						
3		Kilometer nicht positiv oder 0 in Zeile 2						
4		Arbeitszeit liegt nicht in den zugelassenen Grenzen in Zeile 3						
5		Gefahrene Kilometer liegt nicht unterhalb der zugelassenen Grenze in Zeile 4						
6		Arbeitszeit liegt nicht in den zugelassenen Grenzen in Zeile 6						
7								

Abbildung 15.7
Fehlerprotokoll für das Schlüsseldienstbeispiel mit Eingabefehlern

gen die Plausibilitätskontrollen erfolgt in Tabellenblatt 3. Die Schaltfläche zum Start des Plausibilitätenprogramms soll grundsätzlich in jedes Tabellenblatt einbaubar sein. Sind alle Eingaben okay, wird eine MsgBox mit der Meldung „alles okay“ aufgeblendet, ansonsten eine MsgBox mit dem Hinweis darauf, dass Verstöße gegen die Plausibilitäten vorliegen. Schauen wir uns nun die Lösung an:

Beispiel 15.11 *Das Programm zur Plausibilitätsprüfung*

```
Sub plausibilitaeten_click()
    Const NACHTZUSCHLAG_SPALTE As Long = 4
    Const KILOMETER_SPALTE As Long = 2
    Const ARBEITSZEIT_SPALTE As Long = 3
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2

    Const TABELLENBLATT As Long = 1
    Const ERSTE_ZEILE_MIT_FEHLER_INFO As Long = 2

    Const TABELLENBLATT_PLAUSIBILITAETEN As Long = 3
        Const SPALTE_MIT_FEHLER_INFO As Long = 2

    Const MIN_GRENZE_ARBEISZEIT As Double = 5
    Const MAX_GRENZE_ARBEISZEIT As Double = 180
    Const MAX_GRENZE_KILOMETER As Double = 70

    Dim testwert As String
    Dim i As Long
    Dim fehlerString As String
    Dim fehlerArray() As String
    Dim letzteBesetzteZeile As Long

    ReDim fehlerArray(0)

    Dim kilometer As Double
    Dim arbeitszeit As Double

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(KILOMETER_SPALTE, ➥
        ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)

    For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
        testwert = Sheets(TABELLENBLATT).Cells(i, KILOMETER_SPALTE)
        If Not istPositiveZahl(testwert) Then
            fehlerString = "Kilometer nicht positiv in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        Else
            kilometer = CDbl(testwert)
            If Not istKleinerGleichGrenzeDouble(kilometer, MAX_GRENZE_KILOMETER) Then
                fehlerString = "Gefahrene Kilometer liegt nicht unterhalb der zugelassenen Grenze➥
                     in Zeile " & i
                Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
            End If
        End If

        testwert = Sheets(TABELLENBLATT).Cells(i, ARBEITSZEIT_SPALTE)
        If Not istPositiveZahl(testwert) Then
            fehlerString = "Arbeitszeit nicht positiv in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        Else
            arbeitszeit = CDbl(testwert)
            If Not liegtZwischenGrenzenDouble(arbeitszeit, MIN_GRENZE_ARBEISZEIT, ➥
                MAX_GRENZE_ARBEISZEIT) Then
                fehlerString = "Arbeitszeit liegt nicht in den zugelassenen Grenzen in Zeile " & ➥
                    i
                Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
            End If
        End If

        testwert = Sheets(TABELLENBLATT).Cells(i, NACHTZUSCHLAG_SPALTE)
        If istWederJaNochNichtGefuellt(testwert) Then
            fehlerString = "Falscher Wert in der Nachtzuschlagspalte in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
    Next i
    If UBound(fehlerArray) > 0 Then
        MsgBox ("Plausibilitätsverletzungen sind aufgetreten, ein Protokoll wird geschrieben!")
```

```
        Call schreibeFehlerArray(fehlerArray, TABELLENBLATT_PLAUSIBILITAETEN, ➥
            ERSTE_ZEILE_MIT_FEHLER_INFO, SPALTE_MIT_FEHLER_INFO)
    Else
        MsgBox ("Alles klar")
    End If
End Sub
```

Gehen wir die Lösung im Einzelnen durch. Zunächst definieren wir einige Konstanten. Die ersten Konstanten definieren die Spalten, die wir im weiteren Verlauf benötigen:

```
Const NACHTZUSCHLAG_SPALTE As Long = 4
Const KILOMETER_SPALTE As Long = 2
Const ARBEITSZEIT_SPALTE As Long = 3
```

Dies ist sinnvoll, weil sich das Layout des Tabellenblatts verändern kann. Nehmen wir an, ein Entscheider im Unternehmen möchte die Spalte mit der Arbeitszeit vor der Spalte mit den gefahrenen Kilometern stehen haben. Dann müssen wir nur die Werte der Konstanten *KILOMETER_SPALTE* und *ARBEITSZEIT_SPALTE* anpassen. Aus dem gleichen Grund legen wir die Zeile, in der die Daten beginnen und das Tabellenblatt, in dem die Daten stehen auf Konstanten.

```
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
Const TABELLENBLATT As Long = 1
```

Auch das Zieltabellenblatt, also das Tabellenblatt, in dem die Plausibilitätsverstöße dargestellt werden, die Zeile in der die Darstellung der Fehler beginnen soll, sowie die Spalte in der die Fehler dargestellt werden, werden auf Konstanten abgelegt:

```
Const TABELLENBLATT_PLAUSIBILITAETEN As Long = 3
Const ERSTE_ZEILE_MIT_FEHLER_INFO As Long = 3
Const SPALTE_MIT_FEHLER_INFO As Long = 2
```

Zum Schluss legen wir die einzuhaltenden Grenzen auf Konstanten. Auch diese können sich im ändern, und auch hier wollen wir dann nur Konstanten anpassen.

```
Const MIN_GRENZE_ARBEISZEIT As Double = 5
Const MAX_GRENZE_ARBEISZEIT As Double = 180
Const MAX_GRENZE_KILOMETER As Double = 70
```

Dann erfolgt die Deklaration der notwendigen Variablen:

```
Dim testwert As String
Dim i As Long
Dim fehlerString As String
Dim fehlerArray() As String
Dim letzteBesetzteZeile As Long
```

Die Variablen *fehlerString* und *fehlerArray* benötigen wir für die bereits in den Beispielen 15.2 und 15.3 dargestellte Fehlerbehandlung. Auf die Variable *testwert* lesen wir die zu überprüfenden Werte aus den Zellen ein. *testwert* ist eine *String*-Variable und kann daher beliebige Werte aufnehmen, ohne dass es zu einem Fehler kommen kann (vgl. die ausführliche Diskussion dazu in Beispiel 15.3).
Dann erfolgt die Initialisierung der Variablen *fehlerArray*.

```
ReDim fehlerArray(0)
```

Dies wurde ebenfalls in Beispiel 15.2 ausführlich diskutiert. Den Sinn der beiden Variablen

```
Dim kilometer As Double
Dim arbeitszeit As Double
```

werden wir erläutern, sobald sie im Programmablauf genutzt werden.
Wir bestimmen die letzte besetzte Zeile, also die Zeile, bis zu der unsere Schleife laufen muss.

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(KILOMETER_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
```

Zur Bestimmung der letzten besetzten Zeile können wir grundsätzlich jede Spalte des Tabellenblatts bis zur Spalte E nehmen, mit Ausnahme der Spalte D. Die Spalte D enthält leere Zellen. Das Verfahren, das uns die letzte besetzte Zeile bestimmt, würde in der Spalte D bereits in Zeile 2 abbrechen, da die Zelle D2 leer ist. Unsere Funktion zur Ermittlung der letzten besetzten Zeile würde in der Spalte D also 1 ermitteln. Die Zeile, in der wir die Bestimmung der letzten besetzten Zeile starten, ist natürlich die Zeile 2, weil hier die Daten beginnen. Diese hatten wir ja auf der Konstanten *ERSTE_ZEILE_MIT_INFORMATIONEN* abgelegt. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteZeileInSpalte* hatten wir in Kapitel 9.3 eingehend besprochen, so dass auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun startet unsere Schleife. Sie läuft natürlich von Zeile 2 (*ERSTE_ZEILE_MIT_INFORMATIONEN*) zur letzten besetzten Zeile, da ja jeder Datensatz behandelt werden muss.

```
For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
```

Wir beginnen mit der Überprüfung, ob es sich bei den gefahrenen Kilometern um eine positive Zahl handelt. Dazu lesen wir zunächst die gefahrenen Kilometer im gerade behandelten Datensatz auf die Variable *testwert* ein. Wie in Beispiel 15.3 ausführlich beschrieben müssen wir dies tun, um einen Programmabsturz während der Plausibilitätskontrollen zu vermeiden.

```
    testwert = Sheets(TABELLENBLATT).Cells(i, KILOMETER_SPALTE)
```

Wegen der Anforderung die Schaltfläche zum Programmstart in jedes Zabellenblatt einbauen zu können, ist nicht mehr sichergestellt, dass sich Daten und Schaltfläche im gleichen Tabellenblatt befinden. Wir müssen also unter Angabe des Tabellenblattes auf die Zelle zugreifen. Daher ist die Angabe *Sheets(TABELLENBLATT).Cells(i, KILOMETER_SPALTE)* notwendig. *Cells(i, KILOMETER_SPALTE)* reicht nicht mehr. Diese Vorgehensweise ist natürlich auch bei allen anderen Zellzugriffen in diesem Programm notwendig.
Nun prüfen wir, ob es sich bei dem eingelesenen Wert um eine positive Zahl handelt. Dazu benutzen wir die Funktion *istPositiveZahl* aus Beispiel 15.1. Die Funktion *istPositiveZahl* wurde ebenda ausführlich erläutert. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert. Sollte es sich nicht um eine positive Zahl handeln, führen wir die in Kap. 15.1.1 dargestellte Fehlerbehandlung. Wir schreiben den aufgetretenen Fehler und die Zeile in der der Fehler aufgetreten ist auf das Fehlerarray.

```
    If Not istPositiveZahl(testwert) Then
        fehlerString = "Kilometer nicht positiv  in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
```

Als nächstes überprüfen wir, ob die Anzahl innerhalb der vorgegebenen Grenzen liegt:

- Gefahrene Kilometer ist kleiner gleich 70.

Wir stellen sofort fest, dass diese Überprüfung nur dann Sinn ergibt, wenn es sich bei dem Wert für gefahrene Kilometer im gerade behandelten Datensatz um eine Zahl handelt. Denn wenn es sich um keine Zahl handelt, dann benötigen wir auch keine weiterführende Prüfung, weil wir ja schon einen Fehler festgestellt haben. Darüberhinaus müssen wir, wenn wir auf jeden Fall richtige Vergleiche durchführen wollen, den gerade eingelesenen Wert wieder in eine Zahl umwandeln. Der Grund hierfür ist in Beispiel 15.5 eingehend und ausführlich beschrieben.
Wir wandeln daher den Inhalt der Variablen *testwert*, bevor wir mit den Plausibilitätsprüfungen fortfahren, in eine Zahl um. Dies können wir allerdings nur dann, wenn der Inhalt der Variablen *testwert* auch wirklich eine Zahl ist. Wie machen wir das?

```
    If Not istPositiveZahl(testwert) Then
        fehlerString = "Kilometer nicht positiv in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    Else
        kilometer = CDbl(testwert)
```

Sie sehen, die Lösung ist einfach. Wir hatten ja schon (mit einer *if*-Anweisung) festgestellt, ob der Inhalt der Variablen *testwert* eine positive Zahl ist. Wir nehmen jetzt einfach einen *else*-Teil auf. In diesen verzweigt VBA, wenn der Inhalt von *testwert* eine positive Zahl ist. Hier wandeln wir zunächst den Inhalt von Testwert in einen *Double*-Wert um. Dazu dient die Anweisung *Cdbl*[4]. Nun sehen wir auch den Sinn der Deklaration:

```
Dim kilometer As Double
```

Wir benötigen die Variable *kilometer*, um den in Double umgewandelten Wert der Spalte Kilometer des gerade bearbeiteten Datensatzes zu speichern. Nun müssen wir noch den Größenvergleich durchführen. Um unser Plausibilitätenprogramm kürzer zu gestalten und wegen der Wiederverwendung in anderen Programmen benutzen wir dazu eine Abwandlung der in Beispiel 15.8 bereits geschriebene Funktion *istKleinerGleichGrenzeLong*. Hier müssen wir nur den Variablentyp anpassen, denn im Gegensatz zu Beispiel 15.8 müssen wir hier einen Größenvergleich von *double*-Variablen durchführen:

Beispiel 15.12 *Die Funktion istKleinerGleichGrenze für Double-Variablen*

```
Function istKleinerGleichGrenzeDouble(testwert As Double, grenze As Double) As Boolean
  If testwert <= grenze Then
    istKleinerGleichGrenzeDouble = True
  Else
    istKleinerGleichGrenzeDouble = False
  End If
End Function
```

Wir wenden diese Funktion an:

```
    If Not istPositiveZahlOderNull(testwert) Then
        fehlerString = "Kilometer nicht positiv oder 0 in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    Else
        kilometer = CDbl(testwert)
        If Not istKleinerGleichGrenzeDouble(kilometer, MAX_GRENZE_KILOMETER) Then
            fehlerString = "Gefahrene Kilometer liegt nicht unterhalb der zugelassenen Grenze in ➥
                Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
    End If
```

Damit sind die Plausibilitätskontrollen für den Inhalt der Spalte „Kilometer“ beendet.
Kommen wir nun zu den Plausibilitätskontrollen für den Inhalt der Spalte „Arbeitszeit“. Hier sollen wir überprüfen, ob die Arbeitszeit zwischen 5 und 180 Minuten liegt. Wie Beispiel 15.6 zeigt, ist das aber nur möglich, wenn in der Spalte „Arbeitszeit“ eine Zahl steht. Sonst stürzt unser Plausibilitätenprogramm bei der Prüfung mit der Fehlermeldung

Laufzeitfehler 12: Typen unverträglich

ab. Das ist kein erwünschtes Verhalten. Ein Plausibilitätenprogramm soll Fehler oder Plausibilitätsverletzungen melden und nicht selber bei einer solchen abstürzen. Um Abstürze zu vermeiden, müssen wir aber überprüfen, ob die Arbeitszeit im gerade bearbeiteten Datensatz eine Zahl ist. Eigentlich müssen wir genau so verfahren, wie gerade bei den gefahrenen Kilometern.
Was bedeutet: Obwohl es in der Aufgabenstellung nicht explizit vermerkt ist, müssen wir überprüfen, ob die Arbeitszeit im gerade bearbeiteten Datensatz eine Zahl ist. Denn sonst können wir die Aufgabenstellung zu überprüfen, ob die Arbeitszeit zwischen 5 und 180 Minuten liegt, gar nicht realisieren.
Allgemein können wir feststellen: Immer wenn wir Plausibilitätsprüfungen auf Spalten, in denen Zahlen erforderlich sind, durchführen, müssen wir als Erstes prüfen, ob der gerade bearbeitete Wert eine Zahl ist. Und dabei ist es völlig unerheblich, ob das in der Aufgabenstellung explizit verlangt wurde oder nicht.
Zur Überprüfung der Arbeitszeit, können wir die bereits in Beispiel 15.9 geschriebene Funktion nutzen. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit der nachfolgende Programmcode funktioniert. Damit ergibt sich folgende Plausibilitätsprüfung für den Inhalt der Spalte Einkaufspreis:

[4] Abkürzung von *Convert to Double*

```
    testwert = Sheets(TABELLENBLATT).Cells(i, ARBEITSZEIT_SPALTE)
    If Not istPositiveZahl(testwert) Then
        fehlerString = "Arbeitszeit nicht positiv in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    Else
        arbeitszeit = CDbl(testwert)
        If Not liegtZwischenGrenzenDouble(arbeitszeit, MIN_GRENZE_ARBEISZEIT, ➥
            MAX_GRENZE_ARBEISZEIT) Then
            fehlerString = "Arbeitszeit liegt nicht in den zugelassenen Grenzen in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
    End If
```

Hier sehen wir auch, warum die Deklaration:

```
Dim arbeitszeit As Double
```

notwendig war.
Nun bleibt noch zu überprüfen:

- Die Spalte Nachtzuschlag ist entweder gar nicht oder mit „ja“ gefüllt.

Auch für diese Überprüfung schreiben wir, um das Plausibilitätenprogramm kürzer unnd besser lesbar zu halten, eine eigene Funktion:

Beispiel 15.13 *Die Funktion istWederJaNochNichtGefuellt*

```
Function istWederJaNochNichtGefuellt(testwert As String) As Boolean
    If testwert <> "ja" And testwert <> "" Then
        istWederJaNochNichtGefuellt = True
    Else
        istWederJaNochNichtGefuellt = False
    End If
End Function
```

In der Plausibilitätenanwendung erfolgt dann folgender Aufruf:

```
    testwert = Sheets(TABELLENBLATT).Cells(i, NACHTZUSCHLAG_SPALTE)
    If istWederJaNochNichtGefuellt(testwert) Then
        fehlerString = "Falscher Wert in der Nachtzuschlagspalte in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    End If
```

Damit sind die Plausibilitätsprüfungen beendet. Der Schluss des Programms

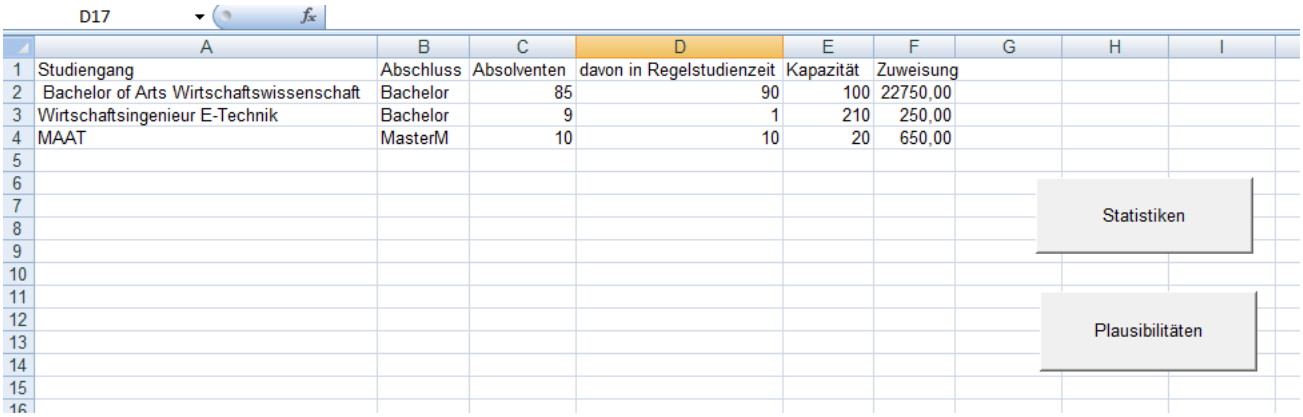
```
Next i
If UBound(fehlerArray) > 0 Then
    MsgBox ("Plausibilitätsverletzungen sind aufgetreten, ein Protokoll wird geschrieben!")
    Call schreibeFehlerArray(fehlerArray, TABELLENBLATT_PLAUSIBILITAETEN, ➥
        ERSTE_ZEILE_MIT_FEHLER_INFO, SPALTE_MIT_FEHLER_INFO)
Else
    MsgBox ("Alles klar")
End If
```

wurde bereits in Beispiel 15.3 erklärt.

15.1.4 Das Zuweisungsbeispiel

Als nächstes wollen wir ein weiteres etwas komplexeres Beispiel betrachten.
Wir sollen die in Abb. 15.8 dargestellten Daten überprüfen.
Folgende Eingabekontrollen sollen durchgeführt werden:

- Absolventen und Absolventen in Regelstudienzeit müssen jeweils positive Zahlen sein, die Null ist erlaubt.



	A	B	C	D	E	F	G	H	I
1	Studiengang	Abschluss	Absolventen	davon in Regelstudienzeit	Kapazität	Zuweisung			
2	Bachelor of Arts Wirtschaftswissenschaft	Bachelor	85	90	100	22750,00			
3	Wirtschaftsingenieur E-Technik	Bachelor	9	1	210	250,00			
4	MAAT	MasterM	10	10	20	650,00			

Abbildung 15.8
Zuweisungsbeispiel mit Eingabefehlern

- Die Kapazität muss ebenfalls eine positive Zahl sein, die Null ist nicht erlaubt.
- Die Hochschule besitzt keinen Studiengang mit einer Kapazität von mehr als 200 Studenten.
- Erlaubte Abschlüsse sind Master, Bachelor und Diplom.
- Natürlich muss die Anzahl der Absolventen größer gleich der Anzahl der Absolventen in Regelstudienzeit sein.

Unsere Anwendung muss also das in Abb. 15.9 dargestellte Fehlerprotokoll erzeugen.

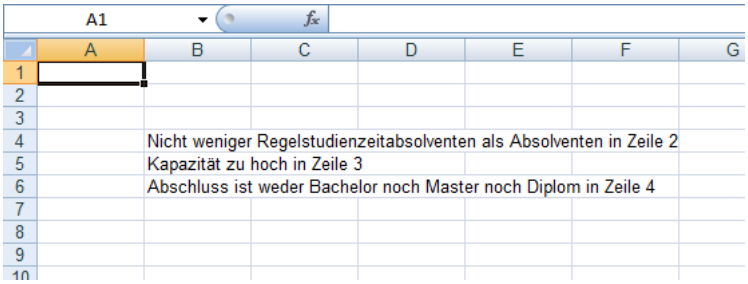


Abbildung 15.9
Fehlerprotokoll für das Zuweisungsbeispiel mit Eingabefehlern

Die Ausgabe der Verstöße gegen die Plausibilitätskontrollen erfolgt in Tabellenblatt 3. Die Schaltfläche zum Start des Plausibilitätenprogramms soll grundsätzlich in jedes Tabellenblatt einbaubar sein. Sind alle Eingaben okay, wird eine MsgBox mit der Meldung „alles okay“ aufgeblendet, ansonsten eine MsgBox mit dem Hinweis darauf, dass Verstöße gegen die Plausibilitäten vorliegen.
Schauen wir uns nun die Lösung an:

Beispiel 15.14 *Das Programm zur Plausibilitätsprüfung*

```
Sub plausibilitaeten_click()
    Const ABSCHLUSS_SPALTE As Long = 2
    Const ABSOLVENTEN_SPALTE As Long = 3
    Const DAVON_REGELSTUDIENZEIT_SPALTE As Long = 4
    Const KAPAZITAETS_SPALTE As Long = 5
    Const ZUWEISUNG_SPALTE As Long = 6

    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
    Const TABELLENBLATT As Long = 1
```

```
Const TABELLENBLATT_PLAUSIBILITAETEN As Long = 3
Const SPALTE_MIT_FEHLER_INFO As Long = 2
Const ERSTE_ZEILE_MIT_FEHLER_INFO As Long = 4

Const KAPAZITAETS_GRENZE As Long = 200

Dim testwert As String
Dim i As Long

Dim fehlerString As String
Dim fehlerArray() As String
Dim letzteBesetzteZeileabsolventen As Long
Dim absolventen As Long
Dim davonInRegelstudienzeit As Long
Dim absolventenIstZahl As Boolean
Dim davonInRegelstudienzeitIstZahl As Boolean
Dim kapazitaet As Long

ReDim fehlerArray(0)
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(ABSCHLUSS_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)

For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
    testwert = Sheets(TABELLENBLATT).Cells(i, ABSCHLUSS_SPALTE)
    If istWederBachelorNochMasterNochDiplom(testwert) Then
        fehlerString = "Abschluss ist weder Bachelor noch Master noch Diplom in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    End If

    testwert = Sheets(TABELLENBLATT).Cells(i, KAPAZITAETS_SPALTE)
    If Not istPositiveZahl(testwert) Then
        fehlerString = "Kapazität nicht positiv in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    Else
        kapazitaet = CLng(testwert)
        If Not istKleinerGleichGrenzeLong(kapazitaet, KAPAZITAETS_GRENZE) Then
            fehlerString = "Kapazität zu hoch in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
    End If

    testwert = Sheets(TABELLENBLATT).Cells(i, ABSOLVENTEN_SPALTE)
    If Not istPositiveZahlOderNull(testwert) Then
        fehlerString = "Anzahl Absolventen nicht positiv oder 0 in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        absolventenIstZahl = False
    Else
        absolventen = CLng(testwert)
        absolventenIstZahl = True
    End If

    testwert = Sheets(TABELLENBLATT).Cells(i, DAVON_REGELSTUDIENZEIT_SPALTE)
    If Not istPositiveZahlOderNull(testwert) Then
        fehlerString = "davon in Regelstudienzeit nicht positiv oder 0 in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        davonInRegelstudienzeitIstZahl = False
    Else
        davonInRegelstudienzeit = CLng(testwert)
        davonInRegelstudienzeitIstZahl = True
    End If

    If absolventenIstZahl And davonInRegelstudienzeitIstZahl Then
        If Not istKleinerGleichGrenzeLong(davonInRegelstudienzeit, absolventen) Then
            fehlerString = "Nicht weniger Regelstudienzeitabsolventen als Absolventen in ➥
                Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
```

```
        End If

    Next i
    If UBound(fehlerArray) > 0 Then
        MsgBox ("Plausibilitätsverletzungen sind aufgetreten, ein Protokoll wird geschrieben!")
         Call schreibeFehlerArray(fehlerArray, TABELLENBLATT_PLAUSIBILITAETEN, ➥
             ERSTE_ZEILE_MIT_FEHLER_INFO, SPALTE_MIT_FEHLER_INFO)
    Else
        MsgBox ("Alles klar")
    End If
End Sub
```

Gehen wir die Lösung im Einzelnen durch:
Zunächst definieren wir einige Konstanten. Die ersten Konstanten definieren die Spalten, die wir im weiteren Verlauf benötigen:

```
Const ABSCHLUSS_SPALTE As Long = 2
Const ABSOLVENTEN_SPALTE As Long = 3
Const DAVON_REGELSTUDIENZEIT_SPALTE As Long = 4
Const KAPAZITAETS_SPALTE As Long = 5
Const ZUWEISUNG_SPALTE As Long = 6
```

Dies ist sinnvoll, weil sich das Layout des Tabellenblatts verändern kann. Nehmen wir an, ein Entscheider im Unternehmen möchte die Spalte mit den Absolventen vor der Spalte mit dem Abschluss stehen haben. Dann müssen wir nur die Werte der Konstanten *ABSCHLUSS_SPALTE* und *ABSOLVENTEN_SPALTE* anpassen. Aus dem gleichen Grund legen wir die Zeile, in der die Daten beginnen und das Tabellenblatt, in dem die Daten stehen auf Konstanten.

```
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
Const TABELLENBLATT As Long = 1
```

Auch das Zieltabellenblatt, also das Tabellenblatt, in dem die Plausibilitätsverstöße dargestellt werden, die Zeile in der die Darstellung der Fehler beginnen soll, sowie die Spalte in der die Fehler dargestellt werden, werden auf Konstanten abgelegt:

```
Const TABELLENBLATT_PLAUSIBILITAETEN As Long = 3
Const ERSTE_ZEILE_MIT_FEHLER_INFO As Long = 3
Const SPALTE_MIT_FEHLER_INFO As Long = 2
```

Zum Schluss legen wir die einzuhaltende Grenze auf eine Konstante. Auch diese kann sich im ändern, und auch hier wollen wir dann nur eine Konstante anpassen.

```
Const KAPAZITAETS_GRENZE As Long = 200
```

Dann erfolgt die Deklaration der notwendigen Variablen:

```
Dim testwert As String
Dim i As Long
Dim fehlerString As String
Dim fehlerArray() As String
Dim letzteBesetzteZeile As Long
```

Die Variablen *fehlerString* und *fehlerArray* benötigen wir für die bereits in den Beispielen 15.2 und 15.3 dargestellte Fehlerbehandlung. Auf die Variable *testwert* lesen wir die zu überprüfenden Werte aus den Zellen ein. *testwert* ist eine *String*-Variable und kann daher beliebige Werte aufnehmen, ohne dass es zu einem Fehler kommen kann (vgl. die ausführliche Diskussion dazu in Beispiel 15.3).
Dann erfolgt die Initialisierung der Variablen *fehlerArray*.

```
ReDim fehlerArray(0)
```

Dies wurde ebenfalls in Beispiel 15.2 ausführlich diskutiert. Den Sinn der Variablen

```
Dim absolventen As Long
Dim davonInRegelstudienzeit As Long
Dim absolventenIstZahl As Boolean
Dim davonInRegelstudienzeitIstZahl As Boolean
Dim kapazitaet As Long
```

werde ich erläutern, sobald sie im Programmablauf genutzt werden.
Wir bestimmen die letzte besetzte Zeile, also die Zeile, bis zu der unsere Schleife laufen muss.

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(GEWINN_SPALTE, ➥
    ERSTE_ZEILE_MIT_INFORMATIONEN, TABELLENBLATT)
```

Zur Bestimmung der letzten besetzten Zeile können wir grundsätzlich jede Spalte des Tabellenblatts bis zur Spalte F nehmen, weil alle Spalten durchgängig gefüllt sind. Die Zeile, in der wir die Bestimmung der letzten besetzten Zeile starten, ist natürlich die Zeile 2, weil hier die Daten beginnen. Diese hatten wir ja auf der Konstanten *ERSTE_ZEILE_MIT_INFORMATIONEN* abgelegt. Und selbstverständlich muss der Vorgang in Tabellenblatt 1 durchgeführt werden. Hierzu hatten wir die Konstante *TABELLENBLATT* definiert. Die Funktion *ermittleLetzteBesetzteZeileInSpalte* hatten wir in Kapitel 9.3 eingehend besprochen, so dass auf eine erneute Diskussion verzichtet wird. Beachten Sie bitte, dass diese Funktion vorhanden sein muss, damit dieser Aufruf funktioniert.
Nun startet unsere Schleife. Sie läuft natürlich von Zeile 2 (*ERSTE_ZEILE_MIT_INFORMATIONEN*) zur letzten besetzten Zeile, da ja jeder Datensatz behandelt werden muss.

```
For i = ERSTE_ZEILE_MIT_INFORMATIONEN To letzteBesetzteZeile
```

Bevor wir mit der Programmierung beginnen, überlegen wir uns, was wir im Einzelnen machen müssen. Wir haben drei Plausibilitätsprüfungen, die wir relativ einfach lösen können, weil die Lösungen analog zu Kap. 15.1.2 sind. Es handelt sich um:

- Die Kapazität muss ebenfalls eine positive Zahl sein, die Null ist nicht erlaubt.
- Die Hochschule besitzt keinen Studiengang mit einer Kapazität von mehr als 200 Studenten.
- Erlaubte Abschlüsse sind Master, Bachelor und Diplom.

Die einfachste Prüfung ist hierbei:

- Erlaubte Abschlüsse sind Master, Bachelor und Diplom.

Dies ist analog zur Prüfung, ob die Spalte Versandart entweder mit „Download“ oder mit „CD-Versand“ gefüllt ist aus Kap. 15.1.2. Wie in Kap. 15.1.2 schreiben wir auch hier für diese Überprüfung, um das Plausibilitätenprogramm kürzer unnd besser lesbar zu halten, eine eigene Funktion:

```
Function istWederBachelorNochMasterNochDiplom(testwert As String) As Boolean
    If testwert <> "Bachelor" And testwert <> "Master" And testwert <> "Diplom" Then
        istWederBachelorNochMasterNochDiplom = True
    Else
        istWederBachelorNochMasterNochDiplom = False
    End If
End Function
```

Wir bauen diese Funktion in unser Plausibilitätenprogramm ein:

```
    testwert = Sheets(TABELLENBLATT).Cells(i, ABSCHLUSS_SPALTE)
    If istWederBachelorNochMasterNochDiplom(testwert) Then
        fehlerString = "Abschluss ist weder Bachelor noch Master noch Diplom in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    End If
```

Beachten Sie, dass wir hier im Vergleich zu Kap. 15.1.2 die Logik umgedreht haben. Hier heißt unsere Funktion *istWederBachelorNochMasterNochDiplom* und wird ohne ein *Not* davor aufgerufen. In Kap. 15.1.2 hieß die Funktion *istCDVersandOderDownload* und muss dann natürlich mit einem *Not* davor in der *if*-Anweisung aufgerufen werden.
Die beiden Plausibilitätskontrollen

- Die Kapazität muss ebenfalls eine positive Zahl sein, die Null ist nicht erlaubt.
- Die Hochschule besitzt keinen Studiengang mit einer Kapazität von mehr als 200 Studenten.

ist analog zur Überprüfung der Anzahl bzw. des Einkaufspreises aus Kap. 15.1.2. Wir überprüfen zunächst, ob es sich bei dem Inhalt der Spalte Kapazität um eine Zahl handelt. Ist dies nicht der Fall, behandeln wir die Plausibilitätsverletzung, ansonsten wandeln wir den auf dem String *testwert* eingelesenen Wert in einen Long um und rufen dann die in Beispiel 15.8 bereits entwickelte Funktion auf. Die Funktion muss natürlich auch hier vorhanden sein.
Daher ergibt sich:

```
    testwert = Sheets(TABELLENBLATT).Cells(i, KAPAZITAETS_SPALTE)
    If Not istPositiveZahl(testwert) Then
        fehlerString = "Kapazität nicht positiv in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
    Else
        kapazitaet = CLng(testwert)
        If Not istKleinerGleichGrenzeLong(kapazitaet, KAPAZITAETS_GRENZE) Then
            fehlerString = "Kapazität zu hoch in Zeile " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
    End If
```

Diesen Programmteil sollten Sie mit Ihren Kenntnissen aus Kap. 15.1.2 sofort verstehen können. Die Funktion *istPositiveZahl* aus Beispiel 15.1 muss natürlich im Arbeitsblatt verhanden sein.
Kommen wir nun zu den beiden noch zu realisierenden Plausibilitätsprüfungen:

- Absolventen und Absolventen in Regelstudienzeit müssen jeweils positive Zahlen sein, die Null ist erlaubt.
- Natürlich muss die Anzahl der Absolventen größer gleich der Anzahl der Absolventen in Regelstudienzeit sein.

Der erste Teil ist einfach, wir schreiben eine Funktion *istPositiveZahlOderNull*. Dies ist nur eine kleine Veränderung der Funktion *istPositiveZahl*. Dann aber wird es komplizierter. Wir müssen die Inhalte zweier Spalten vergleichen. Und es muss gewährleistet sein, dass im Datensatz, der gerade bearbeitet wird, in beiden Spalten eine Zahl steht. Denn nur dann ist ein Vergleich sinnvoll[5] und nur dann können wir die notwendigen Umwandlungen von String in eine Zahl vornehmen. Die Vorgehensweise aus dem Kap. 15.1.2 oder hier bei der Kapazität können wir nicht wählen, denn dort haben wir die weitergehende Kontrolle jeweils im *else*-Teil nach der Zahlüberprüfung gemacht. Das geht hier nicht. Denn im *else*-Teil der Zahlprüfung von der Absolventenanzahl, wissen wir nicht, ob Absolventenanzahl in Regelstudienzeit eine Zahl ist und umgekehrt. Wir schauen uns jetzt die Lösung an. Zunächst schreiben wir die neue Funktion *istPositiveZahlOderNull*:

Beispiel 15.15 *Die Funktion istPositiveZahlOderNull*

```
Function istPositiveZahlOderNull(testwert As String) As Boolean
    Dim testwertAlsDouble As Double
    If Not IsNumeric(testwert) Then
        istPositiveZahlOderNull = False
        Exit Function
    End If
    testwertAlsDouble = CDbl(testwert)
    If (testwertAlsDouble < 0) Then
        istPositiveZahlOderNull = False
        Exit Function
    End If
    istPositiveZahlOderNull = True
End Function
```

Diese Funktion ist selbsterklärend. Wir schauen uns jetzt den ersten Teil der Lösung in der Plausibilitätenprozedur an:

```
    testwert = Sheets(TABELLENBLATT).Cells(i, ABSOLVENTEN_SPALTE)
    If Not istPositiveZahlOderNull(testwert) Then
        fehlerString = "Anzahl Absolventen nicht positiv oder 0 in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
```

[5] Denn ansonsten ist eines von beiden keine Zahl und dann hat eine von unseren Zahlprüfungen den Fehler ohnehin bereits gemeldet.

```
        absolventenIstZahl = False
    Else
        absolventen = CLng(testwert)
        absolventenIstZahl = True
    End If
```

Wie immer lesen wir zunächst den zu überprüfenden Wert (den Wert in der Absolventenspalte) auf die Variable *testwert* ein. Dann überprüfen wir, ob es sich um eine positive Zahl oder Null handelt. Ist dies nicht der Fall, besetzen wir, wie immer in solchen Fällen, unsere Variable *fehlerString* mit dem korrespondierenden Fehler und schreiben sie auf das *fehlerArray*. Zusätzlich aber weisen wir der am Anfang deklarierten bool'schen Variablen *absolventenIstZahl*

```
Dim absolventenIstZahl As Boolean
```

den Wert *false* zu. Ist der Wert des gerade bearbeiteten Datensatzes in der Absolventenspalte hingegen eine Zahl, wandeln wir, auch wie immer, die auf der Stringvariablen *testwert* eingelesene Zahl in einen Long um und speichern diesen auf der zuvor deklarierten Longvariable *absolventen*. Darüberhinaus setzen wir die bool'sche Variablen *absolventenIstZahl* auf *true*. Danach haben wir folgenden Zustand erreicht:

- Ist der Wert des gerade bearbeiteten Datensatzes in der Absolventenspalte keine Zahl, so hat die bool'sche Variable *absolventenIstZahl* den Wert *false*.
- Ist der Wert des gerade bearbeiteten Datensatzes in der Absolventenspalte hingegen eine Zahl, so hat die bool'sche Variable *absolventenIstZahl* den Wert *true*. Zusätzlich ist die Variable *absolventen* mit dem in Long umgewandelten gerade eingelesenen Wert gefüllt.

Wir schauen uns jetzt den zweiten Teil der Lösung in der Plausibilitätenprozedur an:

```
    testwert = Sheets(TABELLENBLATT).Cells(i, DAVON_REGELSTUDIENZEIT_SPALTE)
    If Not istPositiveZahlOderNull(testwert) Then
        fehlerString = "davon in Regelstudienzeit nicht positiv oder 0 in Zeile " & i
        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        davonInRegelstudienzeitIstZahl = False
    Else
        davonInRegelstudienzeit = CLng(testwert)
        davonInRegelstudienzeitIstZahl = True
    End If
```

Hier lesen wir den jetzt zu überprüfenden Wert (den Wert in der „davon in Regelstudienzeit"-Spalte) auf die Variable *testwert* ein. Dann überprüfen wir, ob es sich um eine positive Zahl oder Null handelt. Ist dies nicht der Fall, besetzen wir, wie immer in solchen Fällen, unsere Variable *fehlerString* mit dem korrespondierenden Fehler und schreiben sie auf das *fehlerArray*. Zusätzlich aber weisen wir der am Anfang deklarierten bool'schen Variablen *davonInRegelstudienzeitIstZahl*

```
Dim davonInRegelstudienzeitIstZahl As Boolean
```

den Wert *false* zu. Ist der Wert des gerade bearbeiteten Datensatzes in der Absolventenspalte hingegen eine Zahl, wandeln wir, auch wie immer, die auf der Stringvariablen *testwert* eingelesene Zahl in einen Long um und speichern diesen auf der zuvor deklarierten Longvariable *davonInRegelstudienzeit*. Darüberhinaus setzen wir die bool'sche Variablen *davonInRegelstudienzeitIstZahl* auf *true*. Danach haben wir folgenden Zustand erreicht:

- Ist der Wert des gerade bearbeiteten Datensatzes in der „davon in Regelstudienzeit"-Spalte keine Zahl, so hat die bool'sche Variable *davonInRegelstudienzeitIstZahl* den Wert *false*.
- Ist der Wert des gerade bearbeiteten Datensatzes in der „davon in Regelstudienzeit"-Spalte hingegen eine Zahl, so hat die bool'sche Variable *davonInRegelstudienzeitIstZahl* den Wert *true*. Zusätzlich ist die Variable *davonInRegelstudienzeit* mit dem in Long umgewandelten gerade eingelesenen Wert gefüllt.

Jetzt können wir unser Problem lösen. Wir hatten bereits festgestellt, dass die Durchführung des Vergleichs

> Inhalt der Absolventenspalte des gerade bearbeiteten Datensatzes größer Inhalt der „davon in Regelstudienzeit"-Spalte

nur dann sinnvoll und möglich ist, wenn beides Zahlen sind. Nach unseren Vorbereitungen ist dieser Fall nun erkennbar. Die Inhalte beider Spalten im gerade bearbeiteten Datensatz sind Zahlen, genau dann, wenn die bool'schen Variablen *absolventenIstZahl* und *davonInRegelstudienzeitIstZahl* beide den Wert *true* besitzen. Wenn dies der Fall ist, sind aber auch die Long-Variablen *absolventen* und *davonInRegelstudienzeit* mit diesen Werten befüllt. Mit einer *if*-Anweisung kann unser Programm nun entscheiden, ob der kombinierte Vergleich durchgeführt werden muss. Zum Vergleich kann dann die bereits bekannte Funktion *istKleinerGleichGrenze* benutzt werden.
Hier folgt also der Problemlösung dritter Teil:

```
    If absolventenIstZahl And davonInRegelstudienzeitIstZahl Then
        If Not istKleinerGleichGrenzeLong(davonInRegelstudienzeit, absolventen) Then
            fehlerString = "Nicht weniger Regelstudienzeitabsolventen als Absolventen in Zeile " ➥
                & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
    End If
```

Hierbei ist:

```
    If absolventenIstZahl And davonInRegelstudienzeitIstZahl Then
```

nur eine abkürzende Schreibweise für:

```
    If absolventenIstZahl=true And davonInRegelstudienzeitIstZahl=true Then
```

Der folgende Teil des Programms

```
        If Not istKleinerGleichGrenze(davonInRegelstudienzeit, absolventen) Then
            fehlerString = "Nicht weniger Regelstudienzeitabsolventen als Absolventen in Zeile " ➥
                & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        End If
```

wird also nur ausgeführt, wenn die Variablen *absolventenIstZahl* und *davonInRegelstudienzeitIstZahl* den Wert *true* haben. Was bedeutet, dass im gerade bearbeiteten Datensatz sowohl in der Spalte „Absolventen", als auch in der Spalte „davon in Regelstudienzeit" Zahlen stehen. Und das ist genau das, was wir wollen. Damit sind die Plausibilitätsprüfungen beendet.
Der Schluss des Programms

```
Next i
If UBound(fehlerArray) > 0 Then
    MsgBox ("Plausibilitätsverletzungen sind aufgetreten, ein Protokoll wird geschrieben!")
    Call schreibeFehlerArray(fehlerArray, TABELLENBLATT_PLAUSIBILITAETEN, ➥
        ERSTE_ZEILE_MIT_FEHLER_INFO, SPALTE_MIT_FEHLER_INFO)
Else
    MsgBox ("Alles klar")
End If
```

wurde bereits in Beispiel 15.3 erklärt.

15.1.5 Allgemeine Vorgehensweise beim Vergleich zweier Spalten mit Zahlen

Wenn wir die Inhalte zweier Spalten mit Zahlen vergleichen wollen, müssen wir also folgendermaßen vorgehen:

- Wir führen für jede der beiden zu vergleichenden Spalten eine bool'sche Variable ein.
- Wir prüfen nacheinander, ob es sich bei den Inhalte beider Spalten um Zahlen handelt.
- Wenn es sich um eine Zahl handelt, setzen wir den Wert der korrespondierenden bool'schen Variablen auf *true*. Zusätzlich wandeln wir den auf einer String-Variablen liegenden eingelesenen Wert in eine Zahl um (Long oder Double, je nachdem) und speichern ihn auf einer korrespondierenden Variablen ab.
- Wenn es sich nicht um eine Zahl handelt, setzen wir den Wert der korrespondierenden bool'schen Variablen auf *false*.

- Nur dann, wenn beide bool'sche Variablen auf *true* stehen, führen wir den Vergleich der beiden Spalten in der gerade bearbeiteten Zeile durch.
- Dabei benutzen wir die auch nur in diesem Fall gefüllten korrespondierenden Zahlvariablen.

15.1.6 Änderung der Aufgabenstellung bei gleicher Lösung

Ich stelle noch einmal die in Kap. 15.1.4 gegebene Aufgabenstellung dar:
Folgende Eingabekontrollen sollen durchgeführt werden:

- Absolventen und Absolventen in Regelstudienzeit müssen jeweils positive Zahlen sein, die Null ist erlaubt.
- Die Kapazität muss ebenfalls eine positive Zahl sein, die Null ist nicht erlaubt.
- Die Hochschule besitzt keinen Studiengang mit einer Kapazität von mehr als 200 Studenten.
- Erlaubte Abschlüsse sind Master, Bachelor und Diplom.
- Natürlich muss die Anzahl der Absolventen größer gleich der Anzahl der Absolventen in Regelstudienzeit sein.

Diese Aufgabenstellung ist nahezu identisch mit:

- Die Hochschule besitzt keinen Studiengang mit einer Kapazität von mehr als 200 Studenten.
- Erlaubte Abschlüsse sind Master, Bachelor und Diplom.
- Natürlich muss die Anzahl der Absolventen größer gleich der Anzahl der Absolventen in Regelstudienzeit sein.

Denn wenn geprüft werden muss, ob die Kapazität kleiner gleich 200 ist, so ist für diese Prüfung Voraussetzung, dass der zu prüfende Wert eine Zahl ist (vgl. Beispiel 15.14). Das bedeutet, wir müssen erst prüfen, ob es sich um eine Zahl handelt, und können erst dann die weitergehende Prüfung durchführen.
Auch wenn wir prüfen, ob die Anzahl der Absolventen größer gleich der Anzahl der Absolventen in Regelstudienzeit ist, handelt es sich um den Vergleich zweier Zahlen. Und Zahlen können wir nur richtig vergleichen, wenn sie auf „Zahl"-Variablen gespeichert sind. Dies bedeutet, die Plausibilitätskontrolle

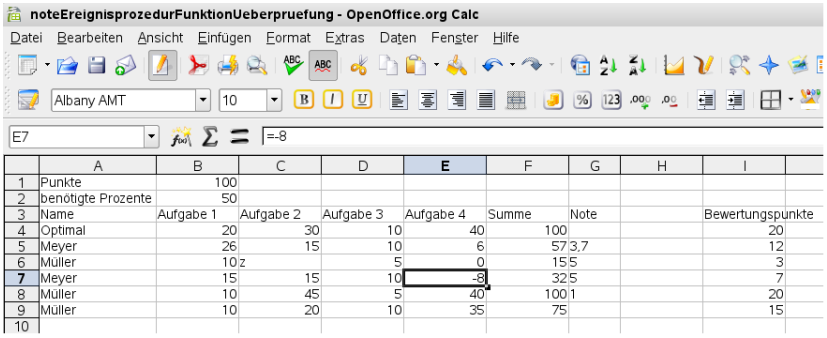
- Natürlich muss die Anzahl der Absolventen größer gleich der Anzahl der Absolventen in Regelstudienzeit sein.

beinhaltet die vorangehenden Prüfungen, ob beides Zahlen sind. Wenn ich die Werte zweier Spalten, die Zahlen beinhalten, vergleichen muss, ist es also völlig irrelevant, ob die Aufgabenstellung explizit eine Prüfung beider Spalten auf Zahlen beinhaltet, ich muss immer wie in Kap. 15.1.5 beschrieben vorgehen.

15.1.7 Das Notenbeispiel

Auch hier können und sollten wir unsere Eingaben validieren. Alle eingegebenen Punkte, auch die in der Optimalzeile, sollten positive Zahlen sein. Anders als beim Provisionsbeispiel ist hier die Null erlaubt, denn Studenten können sehr wohl Null Punkte in einer Aufgabe einer Klausur erreichen. Außerdem muss als Punktzahl auch „nichts" akzeptiert werden, dann nämlich, wenn der Student die Aufgabe nicht bearbeitet hat. Die Eingabe von Leerzeichen soll auch möglich sein, wenn jemand an Stelle von gar nichts lieber ein oder mehrere Leerzeichen eingibt, so soll ihm das gestattet sein. Die Punkte in der Optimalzeile hingegen dürfen nicht Null sein, denn Null Punkte als Höchstpunktzahl einer Klausuraufgabe wäre schon etwas schräg. Darüberhinaus müssen alle vergebenen Punkte in einer Spalte kleiner gleich den Punkten in der Optimalzeile dieser Spalte sein (kein Student kann für eine Aufgabe mehr als die Optimalpunktzahl dieser Aufgabe erhalten). Eingaben, wie in Abb. 15.10, sollten also zu einem Fehlerprotokoll wie in Abb. 15.11 führen.
Wir schreiben also zwei neue Funktionen: *istPositiveZahlNullOderLeer* und *istKleinerAlsGrenzeLong*.

Beispiel 15.16 *Die Funktion istPositiveZahlNullOderLeer*



	A	B	C	D	E	F	G	H	I
1	Punkte	100							
2	benötigte Prozente	50							
3	Name	Aufgabe 1	Aufgabe 2	Aufgabe 3	Aufgabe 4	Summe	Note		Bewertungspunkte
4	Optimal	20	30	10	40	100			20
5	Meyer	26	15	10	6	57	3,7		12
6	Müller	10	z	5	0	15	5		3
7	Meyer	15	15	10	-8	32	5		7
8	Müller	10	45	5	40	100	1		20
9	Müller	10	20	10	35	75			15
10									

Abbildung 15.10
Notenbeispiel mit Benutzerschnittstelle und Eingabefehlern

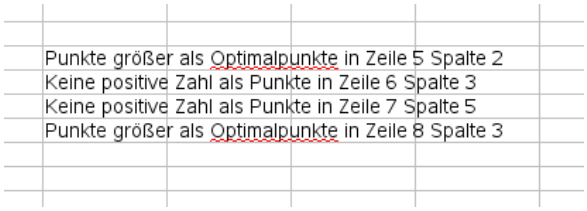


Abbildung 15.11
Fehlerprotokoll zu Abb. 15.10

```
'Dateiname: noteFunktionUeberpruefung.xls
Function istPositiveZahlNullOderLeer(testwert As String, leer As Boolean) As Boolean
    Dim testwertAlsDouble As Double
    leer = False
    If (Trim(testwert) = "") Then
        istPositiveZahlNullOderLeer = True
        leer = True
        Exit Function
    End If
    If Not IsNumeric(testwert) Then
        istPositiveZahlNullOderLeer = False
        Exit Function
    End If
    testwertAlsDouble = CDbl(testwert)
    If (testwertAlsDouble < 0) Then
        istPositiveZahlNullOderLeer = False
        Exit Function
    End If
    istPositiveZahlNullOderLeer = True
End Function
```

Unsere Funktion gibt neben dem Ergebnis der Überprüfung noch zurück, ob die überprüfte Eingabe und damit der Inhalt der Zelle, die gerade zur Überprüfung ansteht, leer war. Dies ist notwendig, weil in der weiteren Verarbeitung, sozusagen im Anschluss, geprüft werden muss, ob die eingegebenen Punkte kleiner oder gleich der Optimalpunktzahl ist. Dies ist

bei einer Eingabe von „nichts“ so nicht mehr notwendig. Überdies muss, bevor der Vergleich mit der Optimalpunktzahl durchgeführt wird, die Eingabe als Zahl behandelt werden. Leere Eingaben als Zahl zu behandeln geht aber in Excel nicht. Die Funktion setzt zunächst die Variable *leer* auf *false*. Sie testet dann, ob der Wert der ihr übergebenen Variablen „nichts“ ist. Dabei wir die Funktion *trim* eingesetzt. *trim* entfernt Leerzeichen am Anfang und Ende einer Zeichenkette. Besteht eine Zeichenkette nur aus Leerzeichen, so wird sie dadurch zu „nichts“ ☺. Wenn eine Zeichenkette also nach der Behandlung durch *trim* der leere String ist, so ist dies eine erlaubte Eingabe, die Variable *leer* wird auf *true* gesetzt, die Funktion gibt *true* zurück und wird verlassen. Der Rest der Implementierung von Beispiel 15.16 entspricht Beispiel 15.1.
Des weiteren benutzen wir die Funktion *istKleinerGleichGrenzeLong* aus Beispiel 15.8.

Beispiel 15.17 *Überprüfung der Eingaben des Notenbeispiels*

```
'Dateiname: noteFunktionUeberpruefung.xls
Sub plausibilitaeten_Click()
    Dim testwert As String
    Dim note As Long
    Dim i As Long
    Dim j As Long
    Dim fehlerString As String
    Dim fehlerArray() As String
    Dim letzteBesetzteZeile As Long
    Dim letzteBesetzteSpalte As Long
    Dim letzteSpalteMitPunkten As Long
    Dim optimalPunkteArray() As Long
    Dim leer As Boolean

    Const OPTIMALZEILE As Long = 4
    Const ERSTE_ZEILE_MIT_FEHLER_INFO As Long = 21
    Const ERSTE_SPALTE_MIT_PUNKTEN As Long = 2
    Const TABELLENBLATT As Long = 1
    Const TABELLENBLATT_FEHLER As Long = 2
    Const FEHLER_SPALTE As Long = 1
    Const NAME_SPALTE As Long = 1

    letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(OPTIMALZEILE, ➥
        ERSTE_SPALTE_MIT_PUNKTEN, TABELLENBLATT)
    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(NAME_SPALTE, OPTIMALZEILE + 1, ➥
        TABELLENBLATT)

    letzteSpalteMitPunkten = letzteBesetzteSpalte - 2

    ReDim optimalPunkteArray(ERSTE_SPALTE_MIT_PUNKTEN To letzteSpalteMitPunkten) As Long
    ReDim fehlerArray(0)

    For i = ERSTE_SPALTE_MIT_PUNKTEN To letzteSpalteMitPunkten
        testwert = Sheets(TABELLENBLATT).Cells(OPTIMALZEILE, i)
        If Not istPositiveZahl(testwert) Then
            fehlerString = "Keine positive Zahl als Optimalwert in Spalte " & i
            Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
        Else
            optimalPunkteArray(i) = CLng(testwert)
        End If
    Next i

    For i = OPTIMALZEILE + 1 To letzteBesetzteZeile
        For j = ERSTE_SPALTE_MIT_PUNKTEN To letzteSpalteMitPunkten
            testwert = Sheets(TABELLENBLATT).Cells(i, j)
            If Not istPositiveZahlNullOderLeer(testwert, leer) Then
                fehlerString = "Keine positive Zahl als Punkte in Zeile " & i & " Spalte " & j
                Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
            Else
                If Not leer Then
                    note = CLng(testwert)
                    If Not istKleinerGleichGrenzeLong(note, optimalPunkteArray(j)) Then
                        fehlerString = "Punkte größer als Optimalpunkte in Zeile " & i & " Spalte➥
                            " & j
                        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
```

```
                End If
            End If
        End If
    Next j
Next i

If UBound(fehlerArray) > 0 Then
    MsgBox ("Plausibilitätsverletzungen sind aufgetreten, ein Protokoll wird geschrieben!")
    Call schreibeFehlerArray(fehlerArray, TABELLENBLATT_FEHLER, ERSTE_ZEILE_MIT_FEHLER_INFO, ➥
        FEHLER_SPALTE)
Else
    MsgBox ("Alles klar")
End If
End Sub
```

Nach der Deklaration der benötigten Variablen und Konstanten werden zunächst die letzte besetzte Zeile und die letzte besetzte Spalte ermittelt. Denn im Gegensatz zum Provisionsbeispiel wissen wir hier ja nicht, wie viele Spalten besetzt sind, weil Klausuren ja unterschiedlich viele Aufgaben beinhalten können. Die letzte Spalte mit Punkten ist dann die letzte besetzte Spalte minus zwei, weil in der letzten Spalte die Note steht, in der vorletzten hingegen die Summe der Punkte. Dann definieren wir das Array *optimalPunkteArray*. Der Index des Arrays läuft von 1 bis *letzteSpalteMitPunkten*. Die Spalte 2 ist die erste Spalte mit Punkten. Auf diesem Array werden wir die Optimalpunkte speichern, damit wir beim Vergleich der erzielten Punkte mit den Optimalpunkten nicht immer auf das Tabellenblatt zugreifen müssen. In einer Schleife über die Aufgabenspalten testen wir dann die Optimalzeile. Wenn ein Wert keine positive Zahl ist, schreiben wir den Fehler, wie in Beispiel 15.3, auf das *fehlerArray*. Anderenfalls speichern wir den Wert auf dem *optimalPunkteArray*. Danach starten wir eine Schleife über alle besetzten Zeilen. In jeder Zeile müssen wir nun die Zellen aller Spalten bis zur letzten Spalte mit eingetragenen Punkten prüfen. Dies bedeutet, wir müssen eine innere Schleife benutzen, die nun die Zellen dieser Zeile überprüft. Dies ist die Konstruktion:

```
For i = OPTIMALZEILE + 1 To letzteBesetzteZeile
    For j = ERSTE_SPALTE_MIT_PUNKTEN To letzteSpalteMitPunkten
```

Wir holen den Inhalt jeder Zelle und prüfen, ob der Wert der Zelle positiv, Null oder leer ist. Ist dies nicht der Fall, schreiben wir den Inhalt auf das Fehlerarray. Ansonsten prüfen wir nur dann, wenn die Zelle nicht leer war, ob der Wert der Zelle kleiner gleich dem in der Optimalzeile ist. Dazu benutzen wir bei der Überprüfung der Optimalzeile gefüllte *optimalPunkteArray*. Beachten Sie, dass wir hier den Rückgabeparameter *leer* der Funktion *istPositiveZahlNullOderLeer* benötigen, um zu entscheiden, ob die zweite Überprüfung überhaupt notwendig ist. Hier ist es ebenfalls wichtig, den Inhalt der Zelle auf eine Longvariable zu legen, damit der Vergleich zu einem richtigen Ergebnis führt. Wenn die innere Schleife beendet ist, wechselt die äußere Schleife in die nächste Zeile. Die Überprüfung der Spalten dieser Zeile wird durchgeführt. Das Ende der Funktion entspricht Beispiel 15.3.

Kapitel 16

MsgBox und InputBox

In diesem Kapitel besprechen wir zwei weitere Arten der Benutzerinteraktion. Zum einen können wir in VBA Fragen stellen, die der Benutzer durch clicken beantworten kann. Sie kennen diese Fragefenster vom Betriebssystem, wenn Sie z.B. eine Datei löschen wollen. Normalerweise werden Sie gefragt, ob Sie das wirklich wollen. Durch clicken auf *OK* wird der Vorgang durchgeführt, durch Betätigen der *Abbrechen*-Schaltfläche hingegen, wie der Name schon sagt, abgebrochen. Das werden wir im Kapitel über Umsatzprognose (Kap. 17) benötigen, denn wir werden vor dem Berechnen der Prognose nachfragen, ob die Prognose linear oder nicht linear durchgeführt werden soll[1].
Zum Zweiten kann man Eingaben über *InputBoxen* tätigen. Das ist ganz einfach. Eine *InputBox* ist ein Fragefenster mit einer freien Eingabemöglichkeit. Wir beginnen mit der *MsgBox*.

16.1 MsgBox Schaltflächen und ihre Bedeutung

Wir wollen das in Abb. 16.1 gezeigte Fragefenster implementieren.

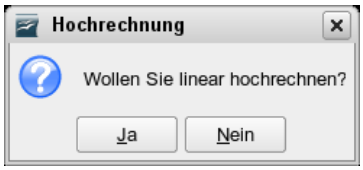


Abbildung 16.1
Fragefenster

Wir zeigen Ihnen sofort das hierfür verantwortliche hochkomplizierte Programm:

Beispiel 16.1 *Funktion zur Prognose des Jahresumsatzes mit Testprozedur*

```
'Dateiname: frage.xls
Sub frage()
    Dim linear As Long
    '4 bedeutet: Ja und Nein Button anzeigen'
    '32 bedeutet: Fragezeigen anzeigen'
    linear = MsgBox("Wollen Sie linear hochrechnen?", 4 + 32, "Hochrechnung")

    If linear = 6 Then
        MsgBox ("Sie haben ja geclickt")
    ElseIf linear = 7 Then
        MsgBox ("Sie haben nein geclickt")
    Else
        MsgBox ("Sie haben etwas Unmögliches gemacht!")
    End If
End Sub
```

[1] Was wir damit meinen, wird allerdings erst in eben diesem Kapitel verraten ☺.

Tabelle 16.1
Schaltflächen: Kodierung und Bedeutung

Kodierung	Bedeutung
0	Nur Schaltfläche OK
1	Schaltflächen OK und Abbrechen
2	Schaltflächen Abbruch, Wiederholen und Ignorieren
3	Schaltflächen Ja, Nein und Abbrechen
4	Schaltflächen Ja und Nein
5	Schaltflächen Wiederholen und Abbrechen

Tabelle 16.2
Grafiken: Kodierung und Bedeutung

Kodierung	Bedeutung
16	kritischer Fehler
32	Frage
48	Warnmeldung
64	Information

MsgBox ist, wie Sie sehen, vielseitig verwendbar. Das Fragefenster wird offenbar durch die Zeile

```
linear = MsgBox("Wollen Sie linear hochrechnen?", 4 + 32, "Hochrechnung")
```

erzeugt.
Man kann also über eine *MsgBox* nicht nur, wie in Kapitel 7 beschrieben, eine Meldung ausgeben, man kann auch Schaltflächen auf die *MsgBox* platzieren, der *MsgBox* einen Titel geben und Grafiken, wie das Fragezeichen aus Abb. 16.1, auf die *MsgBox* zaubern. Der Titel der *MsgBox* wird eindeutig durch den dritten Übergabeparameter bestimmt, der Text in der *MsgBox* ist der erste Übergabeparameter. Schaltflächen und Grafik müssen also durch den zweiten Übergabeparameter (4 + 32) erzeugt werden. Dies ist auch der Fall. Excel verwendet eine Kodierung, um zu bestimmen, welche Grafiken oder Schaltflächen auf der *MsgBox* dargestellt werden. Die Tabellen 16.1 und 16.2 zeigen die gültigen Codes und ihre Bedeutung. Unser Fragefenster enthält die Schaltflächen „Ja“ und „Nein“, dem entspricht nach Tabelle 16.1 die Kodierung 4. Darüberhinaus enthält das Fragefenster ein Fragezeichen, das ist die Kodierung 32 aus Tabelle 16.2. Die Kodierungen sind dann, wie wir bereits festgestellt haben, der zweite Übergabeparameter der Funktion. Sie können durch Pluszeichen getrennt eingegeben werden. Alternativ kann man die Addition selbst durchführen und die Summe übergeben.

```
linear=MsgBox("Wollen Sie linear hochrechnen?", 36, "Hochrechnung")
```

ist also eine weitere Möglichkeit, dieses Fenster zu erzeugen.
Die Funktion *MsgBox* gibt dann, je nachdem welche Schaltfläche betätigt wurde, einen anderen Wert zurück. Da es 7 unterschiedliche Schaltflächen gibt (ja, zwischen „Abbruch“ und „Abbrechen“ wird erstaunlicherweise unterschieden), gibt es demzufolge auch 7 unterschiedliche Rückgabewerte. Sie sind in Tabelle 16.3 dargestellt. So erklärt sich die *if-elseif*-Anweisung aus Beispiel 16.1. Da nur die Schaltflächen „Ja“ und „Nein“ vorhanden sind, sind die möglichen Rückgabewerte 6 bzw. 7. Andere Rückgaben können nicht auftreten.

16.2 Die InputBox

InputBoxen sind noch einfacher zu implementieren und auch zu verstehen. Wir werden eine *InputBox* nutzen, um Benutzer zu fragen, für welchen Monat die Prognose durchgeführt werden soll. Darum nutzen wir dies auch als Beispiel. Um das Beispiel etwas gehaltvoller zu machen, wandeln wir die eingegebene Monatszahl in den Namen des Monats um. In der Benutzerschnittstelle sieht eine *InputBox* wie in Abb. 16.2 aus.

Tabelle 16.3
Rückgabewerte der MsgBox

Rückgabewert	Schaltfläche
1	OK
2	Abbrechen
3	Abbruch
4	Wiederholen
5	Ignorieren
6	Ja
7	Nein

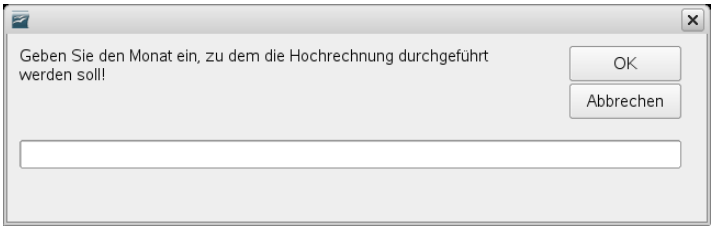


Abbildung 16.2
Eingabefenster

Der zugehörige Code ist:

Beispiel 16.2 *Fragefenster und Programm zur Monatszahl in Monatsnamenumwandlung*

```
'Dateiname: monatFragen.xls
Sub monatFragen()
    Dim monat As Long
    Dim monatString As String
    monat = InputBox("Geben Sie den Monat ein, zu dem die Hochrechnung durchgeführt werden soll!"➥
        )
    monatString = erzeugeMonatNameAusLong(monat)
    MsgBox ("Monat numerisch " & monat & "  Monat: " & monatString)
End Sub

Function erzeugeMonatNameAusLong(monat As Long) As String
    Dim monatArray(1 To 12) As String

    monatArray(1) = "Januar"
    monatArray(2) = "Februar"
    monatArray(3) = "März"
    monatArray(4) = "April"
    monatArray(5) = "Mai"
    monatArray(6) = "Juni"
    monatArray(7) = "Juli"
    monatArray(8) = "August"
    monatArray(9) = "September"
    monatArray(10) = "Oktober"
    monatArray(11) = "November"
    monatArray(12) = "Dezember"

    erzeugeMonatNameAusLong = monatArray(monat)
End Function
```

Alles was in den Klammern hinter *InputBox* folgt, wird im Eingabefenster dargestellt (vgl. Abb. 16.2). Darüberhinaus stellt InputBox ein Eingabefeld zur Verfügung. Der Inhalt dieses Eingabefeldes wird, wenn der Benutzer den OK-Button

clickt, der Variable *monat* zugewiesen. Danach wird die Funktion *erzeugeMonatNameAusLong* aufgerufen. Diese Funktion erwartet den Monat als Zahl und wandelt diese Zahl in den Namen des Monats um. Zu diesem Zweck deklarieren wir in der Funktion einfach ein Array, dessen Index die Nummer des Monats ist und der zugehörige Wert der Name des entsprechenden Monats. Und das wars auch schon, denn nun geben wir einfach *monatArray* von dem übergebenen Monat zurück.

Kapitel 17

Umsatzprognose und der Einbau in das Provisionsbeispiel

In unserem Provisionsbeispiel fehlt von der reinen Funktionalität her[1] nur noch der Vergleich der bisher erzielten Verkaufsbeträge mit dem, was wir hätten verkaufen müssen, um das vertraglich vereinbarte Umsatzziel zu erreichen. Dazu müssen wir aber den bisher erzielten Umsatz auf das Gesamtjahr hochrechnen. Wir betrachten hier zwei Möglichkeiten:

- Der Umsatzverlauf ist linear. Das bedeutet, wir verkaufen in jedem Monat in etwa gleichviel von dem überwachten Produkt. Dann können wir am Ende eines jeden Monats einfach linear hochrechnen.
- Der Umsatzverlauf ist nicht linear. Wir kennen aber die prozentuale Verteilung des Umsatzes auf die einzelnen Monate. Z.B. machen wir im Januar 10% des Umsatzes mit dem überwachten Produkt, im Februar nur 3%, im März hingegen 15% und so weiter. Natürlich muss die Summe aller prozentualen Umsätze 100 ergeben.

17.1 Berechnung der Umsatzprognose

17.1.1 Lineare Umsatzprognose

Der lineare Fall ist einfach. Wir addieren alle Umsätze einschließlich der des Monats zu dem wir die Hochrechnung durchführen wollen. Das Ergebnis multiplizieren wir mit $12/Monat$. Wenn wir also eine Hochrechnung zum März erstellen wollen, multiplizieren wir mit $12/3 = 4$. Eine Hochrechnung zum April ergibt einen Faktor von $12/4 = 3$.

17.1.2 Nicht Lineare Umsatzprognose

Ein bisschen komplizierter ist es bei einem nicht linearen Umsatzverlauf. Doch auch hier ist die Lösung nicht weit entfernt. Wir addieren zunächst die Umsatzprozente, bis zu dem Monat zu dem die Hochrechnung erstellt werden soll. Beispiel: Im Januar erwarten wir 10% des Jahresumsatzes, im Februar 5% und im März 8%. Wenn der März vorbei ist, müssen wir $10\% + 5\% + 8\% = 23\%$ des Jahresumsatzes gemacht haben. Diese Größe nennen wir akkumulierte prozentuale Umsatzverteilung. Am Ende des Jahres sind dies trivialerweise 100%. Dann aber gilt der einfache Dreisatz:

$$\frac{prognostizierterJahresumsatz}{100} = \frac{bisherigerUmsatz}{akkumulierteprozentualeUmsatzverteilung}$$

also

$$prognostizierterJahresumsatz = \frac{bisherigerUmsatz * 100}{akkumulierteprozentualeUmsatzverteilung}$$

Um das Ganze besser zu verdeutlichen, wiederholen wir die Rechnung mit Beispielzahlen:

[1] Dass wir noch schöner formatieren müssen und auch noch Grafiken einbauen, ist keine Frage.

Bisheriger Umsatz: 31.500 Euro entsprechen 23 Prozent des erwarteten Umsatzes

$$prognostizierterJahresumsatz = \frac{31.500 * 100}{23}$$

$$prognostizierterJahresumsatz = 136956,52Euro$$

Damit können wir nun auch im nicht linearen Fall den Jahresumsatz prognostizieren. Wieder addieren wir alle Umsätze einschließlich der des Monats, zu dem wir die Hochrechnung durchführen wollen. Dann berechnen wir die akkumulierte prozentuale Umsatzverteilung dieses Monats. Und zum Schluss multiplizieren wir den bisherigen Umsatz mit 100 durch die berechnete akkumulierte prozentuale Umsatzverteilung.
Im nächsten Schritt müssen wir dies programmieren. Zunächst schreiben wir die Funktion, die den Jahresumsatz prognostiziert. Als Eingangsparameter benötigt diese Funktion den bisherigen Umsatz, den Monat zu dem die Hochrechnung erstellt werden soll und die prozentuale Umsatzverteilung. Und hier kommen Arrays ins Spiel. Denn für die prozentuale Umsatzverteilung ist ein Array die natürliche Wahl.

Beispiel 17.1 *Funktion zur Prognose des Jahresumsatzes mit Testprozedur*

```
'Dateiname: umsatzhochrechnung.xls
Sub testeHochrechnung()
    Dim prozentualeVerteilungArray(1 To 12) As Double
    Dim monat As Long
    Dim bisherigerUmsatz As Double
    Dim jahresprognose As Double
    prozentualeVerteilungArray(1) = 10
    prozentualeVerteilungArray(2) = 8
    prozentualeVerteilungArray(3) = 12
    prozentualeVerteilungArray(4) = 5
    prozentualeVerteilungArray(5) = 5
    prozentualeVerteilungArray(6) = 5
    prozentualeVerteilungArray(7) = 5
    prozentualeVerteilungArray(8) = 10
    prozentualeVerteilungArray(9) = 10
    prozentualeVerteilungArray(10) = 10
    prozentualeVerteilungArray(11) = 10
    prozentualeVerteilungArray(12) = 10
    bisherigerUmsatz = 100000
    monat = 4
    jahresprognose = prognostiziereJahresumsatzNichtLinear(bisherigerUmsatz, monat, ➥
        prozentualeVerteilungArray)
    MsgBox (jahresprognose)
End Sub

Function prognostiziereJahresumsatzNichtLinear(bisherigerUmsatz As Double, monat As Long, ➥
    prozentualeVerteilungArray() As Double) As Double
    Dim i As Long
    Dim akkumulierteProzentualeUmsatzverteilung As Double
    Dim planfaktor As Double
    akkumulierteProzentualeUmsatzverteilung = 0
    For i = 1 To monat
        akkumulierteProzentualeUmsatzverteilung = akkumulierteProzentualeUmsatzverteilung + ➥
            prozentualeVerteilungArray(i)
    Next i
    planfaktor = 100 / akkumulierteProzentualeUmsatzverteilung
    prognostiziereJahresumsatzNichtLinear = planfaktor * bisherigerUmsatz
End Function
```

In der Testprozedur definieren wir zunächst die benötigten Übergabeparameter für die Funktion. Dazu gehört auch ein Array auf dem wir die prozentuale Umsatzverteilung abspeichern wollen. Der Indexbereich des Arrays liegt zwischen 1 und 12, was für Monate nicht ganz unvernünftig ist ☺. Wir belegen dann das Array mit Werten und rufen die Prognosefunktion auf.
Und hier sehen wir, wie nützlich Arrays sind. Der erste Schritt bei der Prognose des Jahresumsatzes ist, die akkumulierte prozentuale Umsatzverteilung zu errechnen. Dies tun wir einfach in einer Schleife über das Array mit der prozentualen Umsatzverteilung. Die Schleife startet bei 1 (Januar ☺) und läuft bis zu dem Monat zu dem die Prognose erstellt werden

soll. In der Schleife addieren wir einfach den prozentualen Umsatz des jeweiligen Indexes auf die vorher mit 0 initialisierte Variable.

```
akkumulierteProzentualeUmsatzverteilung = 0
For i = 1 To monat
    akkumulierteProzentualeUmsatzverteilung = akkumulierteProzentualeUmsatzverteilung + ➥
        prozentualeVerteilungArray(i)
Next i
```

Dann berechnen wir den Faktor, mit dem der bisherige Umsatz multipliziert werden muss und abschließend erfolgt die Bestimmung der Prognose durch Multiplikation des bisherigen Umsatzes mit dem berechneten Faktor:

```
planfaktor = 100 / akkumulierteProzentualeUmsatzverteilung
prognostiziereJahresumsatzNichtLinear = planfaktor * bisherigerUmsatz
```

Und das war es. Damit sind wir mit unserem Vorhaben auch schon fast durch. Den gesamten bisherigen Verkaufsbetrag bestimmen können Sie seit Kap. 8, die Umsatzprognose berechnen seit Kap. 17.1. Das Einzige, was noch fehlt, ist das Einlesen der prozentualen Umsatzverteilung und die Ausgabe des Ergebnisses.

17.2 Einbau in das Provisionsbeispiel

Zum Einlesen der prozentualen Umsatzverteilung benutzen wir natürlich ebenfalls die Tabellenkalkulation. Wir schreiben die prozentuale Umsatzverteilung einfach in das dritte Arbeitsblatt der Tabellenkalkulation. Wir erwarten die prozentuale Umsatzverteilung in der in Abb. 17.1 dargestellten Form:

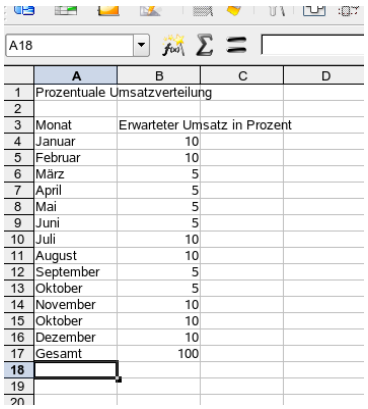


Abbildung 17.1
Eingabe der prozentualen Umsatzverteilung

Zu Beginn der Verarbeitung fragen wir den Benutzer, ob die Prognose linear oder nicht linear vorgenommen werden soll. Antwortet der Benutzer mit linear, so ignorieren wir die Einträge im dritten Tabellenblatt. Ansonsten lesen wir die prozentuale Umsatzverteilung auf ein Array, exakt in der Form, wie in Beispiel 17.1. Das hat den Vorteil, dass unsere Benutzer sich anschauen können, wie das Ergebnis bei linearer Prognose wäre, ohne die Einträge im dritten Tabellenblatt löschen zu müssen.
Den prognostizierten Umsatz schreiben wir dann unter den geplanten Umsatz in das erste Arbeitsblatt der Tabellenkalkulation, die Differenz ebenfalls. Liegen wir im Soll (der prognostizierte Umsatz ist größer gleich dem geplanten Umsatz), stellen wir die Differenzzeile grün, ansonsten rot dar. Das Ergebnis ist in Abb. 17.2 dargestellt. Natürlich benötigen wir eine Schaltfläche, um die Berechnung anzustoßen.
Das Einlesen der prozentualen Umsatzverteilung werden wir in einer eigenen Funktion vornehmen. Wir wollen nämlich in späteren Kapiteln die prozentuale Umsatzverteilung aus einer Datenbank einlesen. Die einzige Änderung, die wir dann vornehmen müssen, ist dann der Austausch der Einlesefunktion.
Beim Start des Programms fragen wir den Benutzer, ob er linear oder nicht linear hochrechnen möchte. Dies machen wir, wie in Kap. 16 beschrieben, mit einer *MsgBox*.

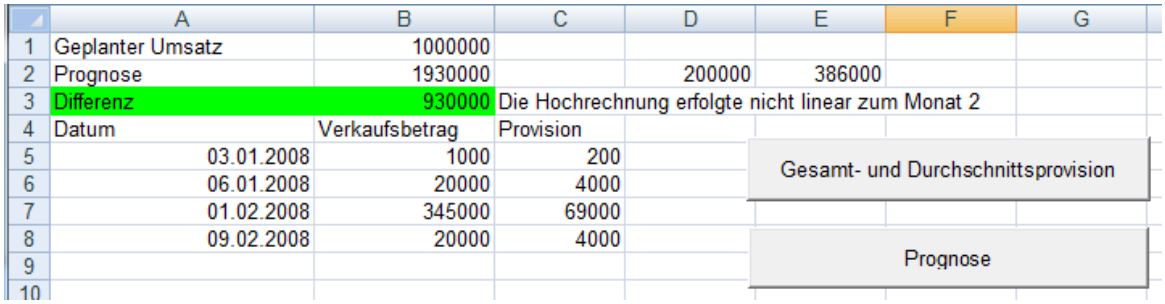


Abbildung 17.2
Darstellung der Prognose

17.2.1 Einbau in das Provisionsbeispiel: Erste einfache Form

Wir beginnen mit der Prozedur, die die prozentuale Umsatzverteilung einliest, und zeigen Ihnen sofort den Code:

Beispiel 17.2 *Prozedur zum Einlesen der prozentualen Umsatzverteilung*

```
'Dateiname: provisionPrognose.xls
Sub liesNichtLineareUmsatzverteilungEin(tabellenblatt As Long, startzeile As Long, wertespalte As➥
     Long, _
                    prozentualeUmsatzverteilungArray() As Double)
    Dim endzeile As Long
    Dim i As Long

    endzeile = startzeile + 11
    For i = startzeile To endzeile
        prozentualeUmsatzverteilungArray(i - startzeile + 1) = Sheets(tabellenblatt).Cells(i, ➥
            wertespalte)
    Next i
End Sub
```

Die Prozedur bekommt drei Informationen zur Verfügung gestellt:

1. das Tabellenblatt, in dem die prozentuale Umsatzverteilung eingegeben wurde (*tabellenblatt*)
2. die Zeile, bei der die prozentuale Umsatzverteilung beginnt (*startzeile*)
3. sowie die Spalte, in der die Werte stehen (*wertespalte*)

Zurückgegeben wird als vierter Übergabeparameter das Array mit der prozentualen Umsatzverteilung. Nachdem in der Funktion die Endzeile für die Werte bestimmt wird (beachten Sie, dass Sie hier nur 11 hinzuaddieren müssen, weil in der Startzeile selber schon der prozentuale Umsatz des Januar steht), werden in einer *for*-Schleife die Werte eingelesen und auf das Array geschrieben. Da der Index des Arrays zwischen 1 und 12 variiert, die Laufvariable der *for*-Schleife hingegen zwischen *startzeile* und *endzeile*, ist der korrekte Index des Arrays jeweils $i - startzeile + 1$.
Als nächstes folgt die Ereignisprozedur:

Beispiel 17.3 *Ereignisprozedur zur Kontrolle der Umsatzziele in Excel*

```
'Dateiname: provisionPrognose.xls
Sub kontrolliereUmsatz_Click()
    '### Variablen deklarieren ###
    Dim linear As Long
    Dim monat As Long
    Dim letzteBesetzteZeile As Long
    Dim letztesVerkaufsdatum As Date
    Dim bisherigerUmsatz As Double
    Dim prognostizierterJahresumsatz As Double
    Dim geplanterJahresumsatz As Double
```

```
    Dim prozentualeUmsatzverteilungArray(1 To 12) As Double
    Dim farbe As Long
    Dim umsatzDifferenz As Double
    Dim hochrechnungstext As String
    '### Konstanten deklarieren ###
    Const DATUMSSPALTE As Long = 1
    Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Long = 5
    Const BISHERIGER_VERKAUFSBETRAG_SPALTE As Long = 2
    Const BISHERIGER_VERKAUFSBETRAG_ZEILE As Long = 2
    Const PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE As Long = 4
    Const PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE As Long = 2
    Const PROZENTUALE_UMSATZVERTEILUNG_TABELLE As Long = 3
    Const BERECHNETE_WERTE_TABELLE As Long = 2
    Const GEPLANTER_UMSATZ_SPALTE As Long = 2
    Const GEPLANTER_UMSATZ_ZEILE As Long = 1

    ' Monat bestimmen
    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(DATUMSSPALTE, _
                    ERSTE_ZEILE_MIT_VERKAUFSBETRAG, 1)
    letztesVerkaufsdatum = Cells(letzteBesetzteZeile, DATUMSSPALTE)
    monat = Month(letztesVerkaufsdatum)
    'Sicherstellen, dass der bisherige Verkaufsbetrag aktuell ist
    Call berechneGesamtUndDurchschnittsprovision_Click
    'bisherigenUmsatz holen
    bisherigerUmsatz = Sheets(BERECHNETE_WERTE_TABELLE).Cells(BISHERIGER_VERKAUFSBETRAG_ZEILE, _
                    BISHERIGER_VERKAUFSBETRAG_SPALTE)
    linear = MsgBox("Wollen Sie linear hochrechnen?", 4 + 32, "Hochrechnung")
    If linear = 6 Then
        'linear hochrechnen
        prognostizierterJahresumsatz = (bisherigerUmsatz * 12) / monat
        hochrechnungstext = "Die Hochrechnung erfolgte linear zum Monat " & monat
    Else
        Call liesNichtLineareUmsatzverteilungEin(PROZENTUALE_UMSATZVERTEILUNG_TABELLE, _
                PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, _
                PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE, _
                prozentualeUmsatzverteilungArray)
        prognostizierterJahresumsatz = prognostiziereJahresumsatzNichtLinear( _
                bisherigerUmsatz, monat, prozentualeUmsatzverteilungArray)
        hochrechnungstext = "Die Hochrechnung erfolgte nicht linear zum Monat " & monat
    End If
    geplanterJahresumsatz = Cells(GEPLANTER_UMSATZ_ZEILE, GEPLANTER_UMSATZ_SPALTE)
    umsatzDifferenz = prognostizierterJahresumsatz - geplanterJahresumsatz

    If umsatzDifferenz > 0 Then
        farbe = RGB(0, 255, 0)
    Else
        farbe = RGB(255, 0, 0)
    End If
    Cells(GEPLANTER_UMSATZ_ZEILE + 1, GEPLANTER_UMSATZ_SPALTE - 1) = "Prognose"
    Cells(GEPLANTER_UMSATZ_ZEILE + 1, GEPLANTER_UMSATZ_SPALTE) = prognostizierterJahresumsatz
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE - 1) = "Differenz"
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE - 1).Interior.Color = farbe
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE) = umsatzDifferenz
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE).Interior.Color = farbe
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE + 1) = hochrechnungstext
End Sub
```

Nach der Deklaration der benötigten Variablen und Konstanten bestimmen wir den Monat der Hochrechnung. Dazu benötigen wir die letzte besetzte Zeile, denn dort finden wir das Datum des letzten Verkaufs[2]. Dann wird der Monat ermittelt. Dazu benutzen wir die VBA interne Funktion *Month*, die, angewendet auf ein Datum, den Monat dieses Datums zurückgibt. Anschließend wäre der bisherige Umsatz zu berechnen. Das brauchen wir aber gar nicht mehr, denn das haben wir in Beispiel 9.2 bereits gelöst. Das war die Ereignisprozedur, die den bisherigen Umsatz, die bisherige Provision, Umsatzsteuer usw. in das zweite Tabellenblatt schrieb. Für Excel sind solche Ereignisprozeduren aber auch normale Prozeduren, die man auch aus eigenen Programmen aufrufen kann. Wir können also mit der Zeile

[2] Sie sehen, wie sinnvoll Funktionen sind, denn wieder nutzen wir unsere Funktion zur Bestimmung der letzten besetzten Zeile.

```
Call berechneGesamtUndDurchschnittsprovision_Click
```

unsere Ereignisprozedur aufrufen, und damit ohne weitere Programmierung die erforderlichen Daten in das zweite Tabellenblatt schreiben.
Nun lesen wir den so berechneten bisherigen Umsatz aus dem zweiten Tabellenblatt ein. Danach wird eine *MsgBox* aufgeblendet mit der Frage, ob wir linear oder nicht linear hochrechnen wollen. Im linearen Fall multiplizieren wir den bisherigen Umsatz mit $(bisherigerUmsatz * 12)/monat$ und schreiben einen Text für die Ausgabe:

```
If linear = 6 Then
    'linear hochrechnen
    prognostizierterJahresumsatz = (bisherigerUmsatz * 12) / monat
    hochrechnungstext = "Die Hochrechnung erfolgte linear zum Monat " & monat
```

Im nicht linearen Fall lesen wir die prozentuale Umsatzverteilung aus dem dritten Tabellenblatt, rufen unsere Funktion zur Prognose des Umsatzes auf und schreiben ebenfalls einen Text für die Ausgabe.

```
Else
    Call liesNichtLineareUmsatzverteilungEin(PROZENTUALE_UMSATZVERTEILUNG_TABELLE, _
            PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, _
            PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE, _
            prozentualeUmsatzverteilungArray)
    prognostizierterJahresumsatz = prognostiziereJahresumsatzNichtLinear( _
            bisherigerUmsatz, monat, prozentualeUmsatzverteilungArray)
    hochrechnungstext = "Die Hochrechnung erfolgte nicht linear zum Monat " & monat
```

Wir lesen den geplanten Jahresumsatz aus dem ersten Tabellenblatt und subtrahieren ihn von unserer Prognose. Wir wollen ja, wenn alles in Ordnung ist, die Differenz in grün darstellen, ansonsten in rot. Um Zellen Farben zuzuweisen, müssen die Werte der Farben, wie Sie ja bereits aus Kap. 11 wissen, in einem bestimmten Format vorliegen. Dieses Format wird mit der VBA-Funktion *rgb* erzeugt. Diese erwartet als Übergabeparameter die Rot-, Grün- und Blau-Werte der gewünschten Farbe. Also bestimmen wir in Abhängigkeit von der Differenz die Farbe:

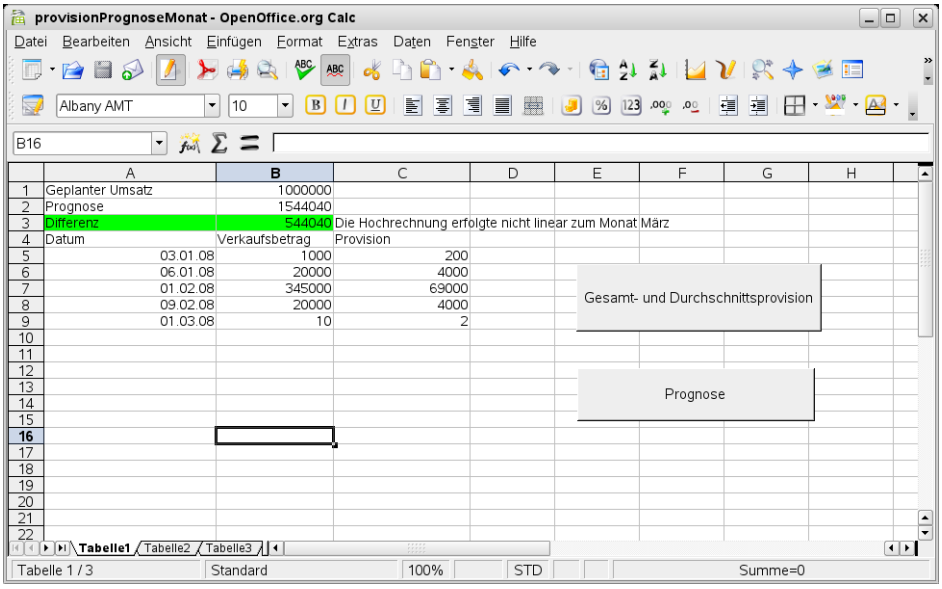
```
if umsatzDifferenz > 0 then
    farbe=rgb(0, 255, 0)
else
    farbe=rgb(255, 0, 0)
end if
```

Zum Schluss schreiben wir die Ergebnisse in das erste Tabellenblatt und geben dabei den Zellen mit der Differenz die gewünschte Farbe. Dies erfolgt mit:

```
Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE).Interior.Color = farbe
```

17.2.2 Provisions- und Umsatzprognose, endgültige Form

Unser Kontrollprogramm hat einen gravierenden Nachteil: Man kann es nur am Ende eines jeden Monats aufrufen, nachdem der letzte Verkaufsbetrag des Monats eingegeben wurde und bevor der erste Verkauf des nächsten Monats eingegeben wird, weil die Ergebnisse sonst doch stark verfälscht werden.
Außerdem reichen die Informationen, die wir in Kap. 17.2.1 erzeugen, bei weitem nicht aus. Um einen richtigen Überblick über den Geschäftsverlauf zu bekommen, benötigen wir mindestens monatliche Informationen. Der prognostizierte Jahresumsatz im Vergleich mit dem geplanten Jahresumsatz reicht darüber hinaus nicht aus. Wir brauchen zumindest noch Aussagen über die Provision. Denn hier können sich gravierende Änderungen ergeben. Wird z.B. der geplante Jahresumsatz verfehlt, so kann es sogar passieren, dass wir in eine andere (niedrigere) Provisionklasse rutschen und statt 20% Provision vielleicht nur noch 5% bekommen. Dann müssen wir am Ende des Jahres sogar Provisionen zurückzahlen. Der umgekehrte Fall kann natürlich auch eintreten. Diese Information müssen wir natürlich auch ausweisen.
Betrachten wir eine mögliche Ergebnisdarstellung. Abb. 17.3 zeigt 5 Verkäufe in insgesamt 3 Monaten und einen geplanten Jahresumsatz von einer Million.
Abb. 17.4 zeigt die Auswertung dieser Monate. Wir sehen 2 „Päckchen“, eines für die Umsätze, eines für die Provisionen. In beiden Päckchen sind die ersten 3 Zeilen gefüllt, weil unsere Auswertung sich auf den Monat März bezieht und davor



	A	B	C	D	E	F	G	H
1	Geplanter Umsatz	1000000						
2	Prognose	1544040						
3	Differenz	544040	Die Hochrechnung erfolgte nicht linear zum Monat März					
4	Datum	Verkaufsbetrag	Provision					
5	03.01.08	1000	200					
6	06.01.08	20000	4000					
7	01.02.08	345000	69000					
8	09.02.08	20000	4000					
9	01.03.08	10	2					

Abbildung 17.3
Verkäufe und geplanter Umsatz

liegen bekannterweise die Monate Januar und Februar. Betrachten wir nun die Zeilen für den Januar: Bis Ende Januar hatten wir einen Umsatz von 21.000 (Spalte Bisheriger Umsatz), geplant waren jedoch 100.000 (Spalte gepl. bis. Umsatz). Die 21.000 sind die Summe der Januar-Verkäufe aus Abb. 17.3. Die 100.000 ergeben sich aus dem geplanten Jahresumsatz von einer Million und daraus, dass wir im Januar 10% des Umsatzes machen wollen (vgl. Abb. 17.1). 100.000 sind 10% von einer Million.

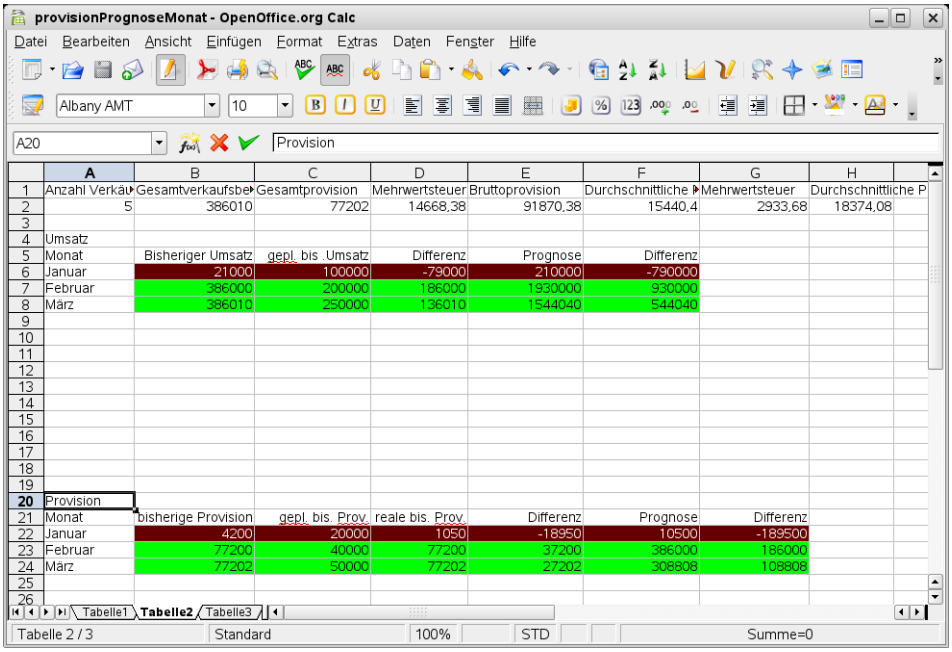
Die sich daraus ergebende Differenz für Januar ist -79.000 (erste Spalte Differenz). Rechnen wir den bisherigen Umsatz mit der Umsatzverteilung aus Abb. 17.1 ergibt sich ein prognostizierter Jahresumsatz von 210.000 (Spalte Prognose). Die Differenz zum geplanten Jahresumsatz beträgt -790.000 (zweite Spalte Differenz).

Was ergibt sich daraus für die Provisionen? Im Januar haben wir insgesamt 4.200 Euro an Provisionen eingenommen (Spalte bisherige Provisionen). Geplant waren aber 20.000 (Spalte gepl. bis. Prov.), denn der geplante Jahresumsatz beträgt eine Million. Der Provisionssatz dort ist 20 %. Die geplante Jahresprovision ist damit 200.000 und weil wir 10 % des Umsatzes und damit auch der Provision im Januar machen wollen, ergibt sich 20.000. In Wirklichkeit ist die Situation im Januar aber noch dramatischer. Aufgrund der Januar-Zahlen prognostizieren wir einen Jahresumsatz von 210.000. Damit verlieren wir den Provisionssatz von 20%. Wir bekommen nur noch 5%. Damit stehen uns keine 4.200 Euro zu, sondern nur 1.050 (Spalte reale bis. Prov.) Den Rest werden wir am Ende des Jahres zurück zahlen müssen, wenn sich nichts ändert. Wir erhalten eine Differenz von -18.950 (erste Spalte Differenz). Für das ganze Jahr prognostizieren wir eine Provision von 10.500 Euro (Spalte Prognose). Dies ist eine Differenz von -189.500 bezogen auf die geplante Provision von 200.000 (zweite Spalte Differenz).

In der Zeile für den Februar werden jetzt die kumulierten Umsätze von Januar und Februar betrachtet. Dies sind 386.000 (Bisheriger Umsatz). Geplant waren 200.000, denn in Januar und Februar wollten wir jeweils 10% des Umsatzes machen, zusammen also 20%. Wir sind jetzt im Plus und haben 186.000 mehr als geplant. Rechnen wir die kumulierten Januar- und Februar-Umsätze auf das Jahr hoch, ergibt sich ein prognostizierter Jahresumsatz von 1.930.000, 930.000 mehr als geplant.

Bei den Provisionen sieht es jetzt natürlich auch gut aus. Insgesamt ergaben sich in Januar und Februar Provisionen von 77.200. Geplant waren 40.000 (20% von den insgesamt geplanten 200.000). Die reale Provision entspricht der bisherigen Provision, denn wir wechseln durch unsere Jahresumsatzprognose den Provisionssatz nicht. Die Differenz ist nun auch beruhigend positiv. Die Provisionsprognose ergibt 386.000, so dass sich im Jahresverlauf eine positive Differenz von 186.000 zur geplanten Provision von 200.000 ergibt.

In der Märzzeile kommen dann die Umsätze von März hinzu. Beachten Sie, dass der geplante Umsatz im März nur um 50.000 steigt. Das liegt daran, dass wir im März nur 5% des jährlichen Umsatzes erwarten, nicht 10% wie jeweils im Januar und Februar.



	A	B	C	D	E	F	G	H
1	Anzahl Verkäufe	Gesamtverkaufsbetrag	Gesamtprovision	Mehrwertsteuer	Bruttoprovision	Durchschnittliche	Mehrwertsteuer	Durchschnittliche P
2	5	386010	77202	14668,38	91870,38	15440,4	2933,68	18374,08
3								
4	Umsatz							
5	Monat	Bisheriger Umsatz	gepl. bis. Umsatz	Differenz	Prognose	Differenz		
6	Januar	21000	100000	-79000	210000	-790000		
7	Februar	386000	200000	186000	1930000	930000		
8	März	386010	250000	136010	1544040	544040		
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20	Provision							
21	Monat	bisherige Provision	gepl. bis. Prov.	reale bis. Prov.	Differenz	Prognose	Differenz	
22	Januar	4200	20000	1050	-18950	10500	-189500	
23	Februar	77200	40000	77200	37200	386000	186000	
24	März	77202	50000	77202	27202	308808	108808	
25								
26								

Abbildung 17.4
Auswertung der ersten 3 Monate

Sie sehen, die Anwendung, die Abb. 17.4 erzeugt hat, ist hervorragend geeignet, solch ein Geschäftsmodell zu überwachen. Nun müssen wir uns nur noch damit beschäftigen, wie man sie programmiert. Und da gehen wir, wie bei der Diskussion des Ergebnisses, spaltenweise vor. Zur Ermittlung des „Bisherigen Umsatz“ addieren wir einfach alle Umsätze im Tabellenblatt 1. Wie das geht, haben Sie schon in Kap. 8.1.1 gelernt. Immer dann, wenn wir einen Monatswechsel haben, schreiben wir den bisherigen Umsatz in das Tabellenblatt 2, errechnen die weiteren notwendigen Informationen, schreiben diese ebenfalls in das Tabellenblatt 2. Wie man erkennt, dass ein Monatswechsel ist, erklären wir später bei der Diskussion des Quellcodes.
Als nächstes müssen wir den geplantem Umsatz berechnen. Hier aber gilt der einfache Dreisatz:

$$\frac{geplanterJahresumsatz}{100} = \frac{geplanterUmsatzbisMonat}{akkumulierteprozentualeUmsatzverteilung}$$

also

$$\frac{geplanterJahresumsatz * akkumulierteprozentualeUmsatzverteilung}{100} = geplanterUmsatzbisMonat$$

Programmieren wir dies direkt:

Beispiel 17.4 *Berechnung des geplanten Monatsumsatzes aus dem geplanten Jahresumsatz*

```
'Dateiname: provisionPrognoseMonat.xls
Function bisZuDiesemMonatZuErreichen(geplanterUmsatz As Double, monat As Long, ➥
    prozentualeVerteilungArray() As Double) As Double
    Dim i As Long
    Dim akkumulierteProzentualeUmsatzverteilung As Double
    Dim planfaktor As Double
    akkumulierteProzentualeUmsatzverteilung = 0
    For i = 1 To monat
        akkumulierteProzentualeUmsatzverteilung = akkumulierteProzentualeUmsatzverteilung + ➥
            prozentualeVerteilungArray(i)
    Next i
    planfaktor = akkumulierteProzentualeUmsatzverteilung / 100
    bisZuDiesemMonatZuErreichen = planfaktor * geplanterUmsatz
End Function
```

Diese Funktion erwartet als Eingaben den geplanten Jahresumsatz, den Monat für den das Ergebnis erzeugt werden soll, sowie das Array mit der Umsatzverteilung auf die Monate. Die Funktion errechnet nun zunächst, wieviel Prozent des Umsatzes bis zu dem gegebenen Monat geplant ist. Das Ergebnis wird durch 100 geteilt und dann mit dem geplantem Jahresumsatz multipliziert und fertig.
Der Rest vom „Umsatzpäckchen“ ist einfach. Differenzen bilden können Sie schon seit dem ersten Kapitel. Die Funktion zur Umsatzprognose haben wir in Beispiel 17.1 entwickelt, ihren Aufruf bereits ebendort oder in den Beispielen 17.3 gesehen.
Kommen wir nun zum „Provisionspäckchen“. Die bisherige Provision ermitteln wir wie den bisherigen Umsatz aus Tabellenblatt 1. Als nächstes müssen wir die geplante bisherige Provision darstellen. Wie aber kommen wir zur geplanten bisherigen Provision? Wäre uns die für das Jahr geplante Provision bekannt, dann könnten wir einfach die von uns geschriebene Funktion *bisZuDiesemMonatZuErreichen* aufrufen, wobei wir anstelle des geplanten Jahresumsatzes die geplante Jahresprovision übergeben. Die Funktion würde dann mit der geplanten monatlichen Provision antworten.
Das Problem reduziert sich also darauf, die geplante Jahresprovision auszurechnen. Aber auch das können wir. Wir stellen noch einmal die in Tabellenblatt 1 eingebundene Funktion vor, die wir seit Kap. 5.2 kennen und die zu einem gegebenen Verkaufsbetrag die Provision ausrechnet:

Beispiel 17.5 *Beispiel 5.6 erneut dargestellt*

```
'Dateiname: provisionPrognoseMonat.xls
Function berechneProvision(geplanterUmsatz As Double, verkaufsbetrag As Double) As Double
    Dim provisionInProzent As Double
    Const umsatzGrenze3 As Double = 1000000
    Const umsatzGrenze2 As Double = 500000
    Const umsatzGrenze1 As Double = 100000

    Const provisionUmsatzGrenze3 As Double = 0.2
    Const provisionUmsatzGrenze2 As Double = 0.1
    Const provisionUmsatzGrenze1 As Double = 0.05
    Const provisionSonst As Double = 0

    If geplanterUmsatz >= umsatzGrenze3 Then
        provisionInProzent = provisionUmsatzGrenze3
    ElseIf geplanterUmsatz >= umsatzGrenze2 Then
        provisionInProzent = provisionUmsatzGrenze2
    ElseIf geplanterUmsatz >= umsatzGrenze1 Then
        provisionInProzent = provisionUmsatzGrenze1
    Else
        provisionInProzent = provisionSonst
    End If

    berechneProvision = verkaufsbetrag * provisionInProzent
End Function
```

Was aber ist nun die geplante Provision? Die geplante Provision erhalten wir, wenn wir exakt den geplanten Umsatz erzielen. Das bedeutet, wenn wir der Funktion *berechneProvision* als *verkaufsbetrag* ebenfalls den geplanten Umsatz übergeben, dann wird *berechneProvision* mit der geplanten Provision antworten. Der Code:

```
geplanteProvision = berechneProvision(geplanterUmsatz, geplanterUmsatz)
```

Dass das funktioniert, können Sie sich sofort klar machen. In unserem Beispiel ist der geplante Umsatz 1 Million. Beim Aufruf ist in der Funktion dann *geplanterUmsatz* 1 Million und *verkaufsbetrag* ebenfalls 1 Million. Das *if-elseif* ermittelt dann als *provisionInProzent provisionUmsatzGrenze3*, also 0,2. In der letzten Zeile der Funktion wird dann *verkaufsbetrag* mit *provisionInProzent* also 0,2 multipliziert. Dies ergibt 200000 und ist exakt das, was an Provision geplant ist.
Als nächstes müssen wir die reale bisherige Provision berechnen. Hier überlegen wir uns: Was unterscheidet die reale bisherige Provision von dem, was wir bekommen haben? Die Antwort ist einfach: Die „normale“ Provisionsberechnung erfolgt auf Basis des geplanten Jahresumsatzes, die Berechnung der realen Provision hingegen soll aufgrund des prognostizierten Jahresumsatzes erfolgen. Unterschiede treten dann auf, wenn auf Basis des prognostizierten Jahresumsatzes ein anderer Provisionssatz ermittelt wird.
Aber dann ist die Ermittlung der realen bisherigen Provision ganz einfach: Wir rufen *berechneProvision* mit dem prognostizierten Jahresumsatzes anstelle des geplanten Jahresumsatzes auf. Folgender Code

```
realeProvisionBisMonat = berechneProvision(prognostizierterJahresumsatz, bisherigerUmsatz)
```

führt also zum richtigen Ergebnis. Fehlt nur noch die prognostizierte Provision. Die können wir komplett analog zur geplanten Provision berechnen, nur dass wir uns nicht auf den geplanten, sondern auf den prognostizierten Jahresumsatz beziehen:

```
prognostiziertePrvision = berechneProvision(prognostizierterJahresumsatz,➥
    prognostizierterJahresumsatz)
```

Der Code zur Erzeugung der Ausgaben sieht also folgendermaßen aus:

Beispiel 17.6 *Beispiel Code zur Erzeugung der in der Darstellung in Tabellenblatt 2 benötigten Informationen*

```
If linear = 6 Then
    prognostizierterJahresumsatz = (bisherigerUmsatz * 12) / alterMonat
    geplanterUmsatzBisMonat = (geplanterJahresumsatz * alterMonat) / 12
    geplanteProvisionBisMonat = (geplanteProvision * alterMonat) / 12
Else
    prognostizierterJahresumsatz = prognostiziereJahresumsatzNichtLinear(bisherigerUmsatz, ➥
        alterMonat, prozentualeUmsatzverteilungArray)
    geplanterUmsatzBisMonat = bisZuDiesemMonatZuErreichen(geplanterJahresumsatz, alterMonat, ➥
        prozentualeUmsatzverteilungArray)
    geplanteProvisionBisMonat = bisZuDiesemMonatZuErreichen(geplanteProvision, alterMonat, ➥
        prozentualeUmsatzverteilungArray)
End If
realeProvisionBisMonat = berechneProvision(prognostizierterJahresumsatz, bisherigerUmsatz)
prognostiziertePrvision = berechneProvision(prognostizierterJahresumsatz, ➥
    prognostizierterJahresumsatz)
umsatzMonatsDifferenz = bisherigerUmsatz - geplanterUmsatzBisMonat
umsatzDifferenz = prognostizierterJahresumsatz - geplanterJahresumsatz
provisionsDifferenzMonat = realeProvisionBisMonat - geplanteProvisionBisMonat
provisionsDifferenz = prognostiziertePrvision - geplanteProvision
```

Der *then*-Teil nach *if linear = 6* rechnet den Jahresumsatz, den geplanten Umsatz bis zum Monat der Hochrechnung und die bis dorthin geplante Provision linear hoch. Auf der Variablen *alterMonat* ist der Monat, zu dem hochgerechnet werden soll, abgespeichert. Nach dem *if*-Konstrukt wird zweimal *berechneProvision* aufgerufen, um die reale Provision und die prognostizierte Provision zu berechnen. Zum Abschluss werden die darzustellenden Differenzen berechnet.
Kommen wir nun zur Implementierung des Programms.

Provisions- und Umsatzprognose, endgültige Form

Beispiel 17.7 *Endgültige Provision-Prognoselogik*

```
'Dateiname: provisionPrognoseMonat.xls
Sub kontrolliereUmsatzMonatlich_Click()
    Dim prozentualeVerteilungArray() As Double
    Dim linear As Long
    Dim monat As Long
    Dim alterMonat As Long
    Dim alterMonatAlsString As String
    Dim letzteBesetzteZeile As Long
    Dim bisherigerUmsatz As Double
    Dim prognostizierterJahresumsatz As Double
    Dim geplanterJahresumsatz As Double
    Dim geplanterUmsatzBisMonat As Double
    Dim geplanteProvision As Double
    Dim geplanteProvisionBisMonat As Double
    Dim prozentualeUmsatzverteilungArray(1 To 12) As Double
    Dim farbe As Long
    Dim umsatzDifferenz As Double
    Dim realeProvisionBisMonat As Double
    Dim prognostiziertePrvision As Double
```

```
Dim umsatzMonatsDifferenz As Double
Dim provisionsDifferenzMonat As Double
Dim provisionsDifferenz As Double
Dim hochrechnungstext As String
Dim hochrechnungsMonatAlsString As String
Dim datum As Date
Dim hochrechnungsMonat As Long
Dim bisherigeProvision As Double
Dim i As Long

Const DATUMSSPALTE As Long = 1
Const VERKAUFSBETRAGSPALTE As Long = 2
Const PROVISIONSSPALTE As Long = 3
Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Long = 5
Const BISHERIGER_VERKAUFSBETRAG_SPALTE As Long = 2
Const BISHERIGER_VERKAUFSBETRAG_ZEILE As Long = 2
Const PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE As Long = 4
Const PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE As Long = 2
Const GEPLANTER_UMSATZ_SPALTE As Long = 2
Const GEPLANTER_UMSATZ_ZEILE As Long = 1
Const TABELLENBLATT_DATEN As Long = 1
Const TABELLENBLATT_UMSATZVERTEILUNG As Long = 2

hochrechnungsMonat = InputBox("Geben Sie den Stützmonat der Hochrechnung ein!")
hochrechnungsMonatAlsString = erzeugeMonatNameAusLong(hochrechnungsMonat)
linear = MsgBox("Wollen Sie linear hochrechnen?", 4 + 32, "Hochrechnung")
If linear = 6 Then
    'linear hochrechnen
    hochrechnungstext = "Die Hochrechnung erfolgte linear zum Monat " & ➥
        hochrechnungsMonatAlsString
Else
    Call liesNichtLineareUmsatzverteilungEin(TABELLENBLATT_UMSATZVERTEILUNG, ➥
        PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE,➥
         prozentualeUmsatzverteilungArray)
    hochrechnungstext = "Die Hochrechnung erfolgte nicht linear zum Monat " & ➥
        hochrechnungsMonatAlsString
End If
geplanterJahresumsatz = Cells(GEPLANTER_UMSATZ_ZEILE, GEPLANTER_UMSATZ_SPALTE)
geplanteProvision = berechneProvision(geplanterJahresumsatz, geplanterJahresumsatz)

letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(DATUMSSPALTE, ➥
    ERSTE_ZEILE_MIT_VERKAUFSBETRAG, TABELLENBLATT_DATEN)
alterMonat = 1
bisherigeProvision = 0
bisherigerUmsatz = 0
For i = ERSTE_ZEILE_MIT_VERKAUFSBETRAG To letzteBesetzteZeile + 1
    If Not IsEmpty(Cells(i, DATUMSSPALTE)) Then
        datum = Cells(i, DATUMSSPALTE)
        monat = Month(datum)
    End If
    If (monat <> alterMonat) Or (i = letzteBesetzteZeile + 1) Then
        'Monatswechsel Prognose und Werte müssen geschrieben werden
        alterMonatAlsString = erzeugeMonatNameAusLong(alterMonat)
        If linear = 6 Then
            prognostizierterJahresumsatz = (bisherigerUmsatz * 12) / alterMonat
            geplanterUmsatzBisMonat = (geplanterJahresumsatz * alterMonat) / 12
            geplanteProvisionBisMonat = (geplanteProvision * alterMonat) / 12
        Else
            prognostizierterJahresumsatz = prognostiziereJahresumsatzNichtLinear(➥
                bisherigerUmsatz, alterMonat, prozentualeUmsatzverteilungArray)
            geplanterUmsatzBisMonat = bisZuDiesemMonatZuErreichen(geplanterJahresumsatz, ➥
                alterMonat, prozentualeUmsatzverteilungArray)
            geplanteProvisionBisMonat = bisZuDiesemMonatZuErreichen(geplanteProvision, ➥
                alterMonat, prozentualeUmsatzverteilungArray)
        End If
        realeProvisionBisMonat = berechneProvision(prognostizierterJahresumsatz, ➥
            bisherigerUmsatz)
        prognostizierteProvision = berechneProvision(prognostizierterJahresumsatz, ➥
```

```
                prognostizierterJahresumsatz)
            umsatzMonatsDifferenz = bisherigerUmsatz - geplanterUmsatzBisMonat
            umsatzDifferenz = prognostizierterJahresumsatz - geplanterJahresumsatz
            provisionsDifferenzMonat = realeProvisionBisMonat - geplanteProvisionBisMonat
            provisionsDifferenz = prognostizierteProvision - geplanteProvision
            If umsatzDifferenz > 0 Then
                farbe = RGB(0, 255, 0)
            Else
                farbe = RGB(200, 0, 0)
            End If
            Call schreibeUmsatzUndProvision(bisherigerUmsatz, geplanterUmsatzBisMonat, ➥
                umsatzMonatsDifferenz, prognostizierterJahresumsatz, umsatzDifferenz, ➥
                bisherigeProvision, geplanteProvisionBisMonat, realeProvisionBisMonat, ➥
                provisionsDifferenzMonat, prognostizierteProvision, provisionsDifferenz, ➥
                alterMonat, alterMonatAlsString, farbe)
            alterMonat = monat
        End If
        If (monat > hochrechnungsMonat) Or (i = letzteBesetzteZeile + 1) Then
            Exit For
        End If
        bisherigeProvision = bisherigeProvision + Cells(i, PROVISIONSSPALTE)
        bisherigerUmsatz = bisherigerUmsatz + Cells(i, VERKAUFSBETRAGSPALTE)
    Next i
    Call schreibePrognose(prognostizierterJahresumsatz, umsatzDifferenz, hochrechnungstext, farbe➥
        )
End Sub
```

Nach der Deklaration der notwendigen Konstanten und Variablen wird zunächst der Monat, zu dem hochgerechnet werden soll, abgefragt. Zu diesem Monat bestimmen wir dann seine *String*-Darstellung für unsere Ausgaben. Anschließend stellt das Programm fest, ob linear oder nicht linear hochgerechnet werden soll. Der geplante Umsatz wird aus dem Tabellenblatt 1 gelesen und die geplante Provision, wie in Kap. 17.2.2 beschrieben berechnet. Wir ermitteln die letzte besetzte Zeile, initialisieren den bisherigen Umsatz und die bisherige Provision mit 0, sowie die Variable *alterMonat* mit 1. Dann beginnen wir unsere Schleife über alle Zeilen mit Verkäufen in Tabellenblatt 1. Warum die Schleife bis *letzteBesetzteZeile + 1* läuft, und warum wir die Zeilen

```
If Not IsEmpty(Cells(i, DATUMSSPALTE)) Then
    datum = Cells(i, DATUMSSPALTE)
    monat = Month(datum)
End If
```

benötigen, erklären wir später. In der Schleife lesen wir zunächst das Datum ein und bestimmen den Monat des aktuellen Datums.

Hat sich der Monat zu dem Wert auf der Variablen *alterMonat* geändert, bedeutet das, ein neuer Monat beginnt (wenn das das erste Mal der Fall ist, bearbeitet das Programm gerade den ersten Februar-Eintrag). Das heißt aber andererseits, die Zeilen, die zum alten Monat gehören, müssen in Tabellenblatt 2 geschrieben werden. Wir überprüfen eine mögliche Änderung im folgenden *if*. Hatten wir eine Änderung (die mit *or* angeschlossene zweite Bedingung erklären wir später), erreichen wir den *then*-Teil des *if*s. Mit den in Beispiel 17.6 dargestellten und besprochenen VBA-Zeilen werden die für Tabellenblatt 2 benötigten Informationen berechnet. Anhand der *umsatzDifferenz* wird jetzt die Farbe der Darstellung bestimmt, rot, wenn wir nicht im Soll liegen, grün sonst.

Dann wird die Prozedur *schreibeUmsatzUndProvision* aufgerufen, die nichts anderes macht, als die übergebenen Werte in Tabellenblatt 2 darzustellen. Wir benutzen hier eine weitere Prozedur, damit unsere „Hauptprozedur“ übersichtlich bleibt. Zum Schluss wird die Variable *alterMonat* auf den jetzt aktuellen Monat gesetzt. Im nächsten *if* wird überprüft, ob der jetzt aktuelle Monat größer ist als der Monat zu dem hochgerechnet werden soll[3]. Ist dies der Fall wird die *for*-Schleife beendet. *Exit For* beendet eine *for* Schleife. Der Grund für diese Bedingung ist: Nehmen wir an, wir haben den 6 April. Eine Auswertung zu April macht wenig Sinn, weil dieser Monat ja noch läuft. Eine Auswertung zu März macht Sinn und im Eingabefenster wird als Monat zu dem hochgerechnet werden soll, 3 eingegeben. Dann dürfen die April-Werte nicht mehr verarbeitet werden. Für den ersten April-Wert ist aber der aktuelle Monat größer als der Hochrechnungsmonat und über diese Bedingung wird die Schleife beendet. Zum Schluss addieren wir den Umsatz und die Provision auf die bisherigen Umsätze und Provisionen.

[3] Die mit *or* angeschlossene Bedingung besprechen wir, wie bereits gesagt, später.

Nachdem die *for*-Schleife verlassen wurde, werden die dann aktuellen Ergebnisse für den prognostizierten Umsatz, die Differenz zum geplanten Umsatz und die Grundlage der Hochrechnung in das erste Tabellenblatt geschrieben.
Warum nun die Geschichte mit *letzteBesetzteZeile + 1*? Das hat einen einfachen Grund. Nehmen wir an, wir haben Umsätze bis zum 31.03 in Tabellenblatt 1, aber noch keine April-Einträge. Wir wollen aber eine Hochrechnung zum März durchführen. Folgendes Problem tritt auf: Da wir keine April-Einträge haben, wird der aktuelle Monat nie von 3 nach 4 wechseln. Das würde aber bedeuten, die Zeile für März, die wichtigste Zeile in diesem Fall, würde nie in das Tabellenblatt 2 geschrieben. Das geht aber gar nicht ☺. Dadurch, dass wir die Schleife nun eine Zeile weiter als Einträge vorhanden sind, laufen lassen, wird die Schleife nach dem Aufsummieren des letzten Eintrags noch einmal ausgeführt. In diesem Fall ist die Zelle auf die nun zugegriffen wir, leer, d.h der Monat wird nicht neu besetzt. Nun haben wir zwar keinen neuen Monat, die zweite Bedingung ist aber erfüllt:

```
if (monat <> alterMonat) or (i = letzteBesetzteZeile + 1) then
```

Daher werden die Werte des letzten Monats in Tabellenblatt 2 übernommen. Verlassen wird die Schleife dann dadurch, dass wieder die zweite Bedingung in

```
if (monat > hochrechnungsMonat) or (i = letzteBesetzteZeile + 1)then
```

true ergibt. Wir zeigen nun die beiden Prozeduren zum Schreiben in die einzelnen Tabellenblätter. Da es sich um reines Schreiben von Variablen in Tabellenzellen handelt, bleiben sie unkommentiert.

Beispiel 17.8 *Schreiben in Tabellenblatt 2*

```
'Dateiname: provisionPrognoseMonat.xls
Sub schreibeUmsatzUndProvision(bisherigerUmsatz As Double, geplanterUmsatzBisMonat As Double, ➥
    umsatzMonatsDifferenz As Double, prognostizierterJahresumsatz As Double, umsatzDifferenz As ➥
    Double, bisherigeProvision As Double, geplanteProvisionBisMonat As Double, ➥
    realeProvisionBisMonat As Double, provisionsDifferenzMonat As Double, ➥
    prognostizierteProvision As Double, provisionsDifferenz As Double, alterMonat As Long, ➥
    alterMonatAlsString As String, farbe As Long)
    Const UMSATZ_START_SPALTE As Long = 1
    Const UMSATZ_START_ZEILE As Long = 5
    Const PROVISION_START_SPALTE As Long = 1
    Const PROVISION_START_ZEILE As Long = 21
    Const TABELLENBLATT As Long = 2

    'umsatz schreiben
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE) = ➥
        alterMonatAlsString
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE).Interior.➥
        Color = farbe
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 1) = ➥
        bisherigerUmsatz
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 1).➥
        Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 2) = ➥
        geplanterUmsatzBisMonat
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 2).➥
        Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 3) = ➥
        umsatzMonatsDifferenz
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 3).➥
        Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 4) = ➥
        prognostizierterJahresumsatz
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 4).➥
        Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 5) = ➥
        umsatzDifferenz
    Sheets(TABELLENBLATT).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 5).➥
        Interior.Color = farbe

    'provision schreiben
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE) = ➥
        alterMonatAlsString
```

```
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, _
                    PROVISION_START_SPALTE).Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 1) =➥
         bisherigeProvision
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, _
                PROVISION_START_SPALTE + 1).Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 2) =➥
         geplanteProvisionBisMonat
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, _
                        PROVISION_START_SPALTE + 2).Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 3) =➥
         realeProvisionBisMonat
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 3).➥
        Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 4) =➥
         provisionsDifferenzMonat
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 4).➥
        Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 5) =➥
         prognostizierteProvision
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 5).➥
        Interior.Color = farbe
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 6) =➥
         provisionsDifferenz
    Sheets(TABELLENBLATT).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 6).➥
        Interior.Color = farbe
End Sub
```

Beispiel 17.9 *Schreiben in Tabellenblatt 1*

```
'Dateiname: provisionPrognoseMonat.xls
Sub schreibePrognose(prognostizierterJahresumsatz As Double, umsatzDifferenz As Double, ➥
    hochrechnungstext As String, farbe As Long)
    Const GEPLANTER_UMSATZ_SPALTE As Long = 2
    Const GEPLANTER_UMSATZ_ZEILE As Long = 1

    Cells(GEPLANTER_UMSATZ_ZEILE + 1, GEPLANTER_UMSATZ_SPALTE - 1) = "Prognose"
    Cells(GEPLANTER_UMSATZ_ZEILE + 1, GEPLANTER_UMSATZ_SPALTE) = prognostizierterJahresumsatz
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE - 1) = "Differenz"
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE - 1).Interior.Color = farbe
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE) = umsatzDifferenz
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE).Interior.Color = farbe
    Cells(GEPLANTER_UMSATZ_ZEILE + 2, GEPLANTER_UMSATZ_SPALTE + 1) = hochrechnungstext
End Sub
```

Kapitel 18

Charts/Diagramme

Vielfach wollen wir die Ergebnisse unserer Berechnungen visualisieren. Bei unserem Provisionsbeispiel wäre es schön, wenn die ausgerechneten Prognosen in einem Chart dargestellt werden könnten. Natürlich können Sie das. Sie müssen nur den Bereich, der im Chart erscheinen soll, selektieren, „Einfügen ⇨Diagramm“ clicken und dann den Erstellungsdialog durchführen. Man kann aber auch eine Ereignisprozedur schreiben, die das erledigt. Abb. 18.1 auf der nächsten Seite zeigt ein Beispiel. Clicken auf die Schaltfläche „Grafik“ erzeugt die Balken- und Geradengrafik. Eine solche Vorgehensweise hat Vorteile:

- Clicken auf eine Schaltfläche ist sicher noch schneller, als ein Diagramm mit Hand einzufügen.
- Die Grafik sieht immer gleich aus. Dies erhöht den Wiedererkennungswert bei Präsentationen und Berichten.

Hinzu kommt, dass die Erstellung von Grafiken recht einfach zu programmieren ist. In den folgenden Kapiteln zeigen wir einmal eine Grafik, wie sie Excel erzeugt, wenn keine Formatisierungsanweisungen programmiert werden. Dies erledigt erfahrungsgemäß 90% aller Fälle, da die voreingestellten Formatierungen von Excel recht gut sind. In einem zweiten Beispiel zeigen wir dann im Programm vorgenommene Formatierungen des Charts. Dies zeigt beispielhaft Ihre Möglichkeiten auf, Ihrer Kreativität ☺ freien Lauf zu lassen.
Weitere Beispiele finden Sie im Internet. Darüberhinaus verfügt Excel über einen Makrorekorder. Man kann die Makroaufzeichnung anschalten, dann das Diagramm „mit Hand“ formatieren und sich danach den VBA-Code ansehen, den die Tabellenkalkulation selber erzeugen würde. Leider ist dieser Code in beiden Fällen relativ schräg, so dass man schon ein wenig Erfahrung benötigt, um dies in eigenen Programmen zu verwenden. Dennoch denken wir, dass der Inhalt dieses Kapitels bereits 95% der gewünschten Diagramme abdecken wird.
Wir zeigen sofort die Realisierung von Abb. 18.1.

Beispiel 18.1 *Grafische Darstellung der prognostizierten Umsatzentwicklung*

```
'Dateiname: provisionPrognoseMonatGrafik.xls
Sub erstelleChart_click()
    Dim letzteBesetzteZeile As Long
    Dim anzahlMonate As Long
    Dim diagrammBreite As Long
    Dim myChart As Object

    Const TABELLENBLATT As Long = 2
    Const UMSATZ_START_SPALTE As Long = 1
    Const PLOT_BEREICH_END_SPALTE As Long = 6
    Const UMSATZ_START_ZEILE As Long = 5
    Const CHART_X_KOORDINATE As Long = 250
    Const CHART_BAR_Y_KOORDINATE As Long = 450
    Const CHART_LINE_Y_KOORDINATE As Long = 750
    Const CHART_HOEHE As Long = 250
    Const BREITENMULTIPLIKATOR As Long = 150

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(UMSATZ_START_SPALTE, ➥
        UMSATZ_START_ZEILE, TABELLENBLATT)
    anzahlMonate = letzteBesetzteZeile - UMSATZ_START_ZEILE
```

	A	B	C	D	E	F	G	H	I	J
1	Anzahl Verkäu	Gesamtverkaufsbet	Gesamtprovision	Mehrwertsteuer	Bruttoprovision	Durchschnittliche	Mehrwertsteuer	Durchschnittliche Provision (Brutto)		
2	5	387010	77402	14706,38	92108,38	15480,4	2941,28	18421,68		
3										
4	Umsatz									
5	Monat	Bisheriger Umsatz	gepl. bis. Umsatz	Differenz	Prognose	Differenz				
6	Januar	22000	83333,33	-61333,33	264000	-736000				
7	Februar	387000	166666,67	220333,33	2322000	1322000				
8	März	387010	250000	137010	1548040	548040				
9										
10										
11										
12										
13										
14										
15										
16										
17										
18										
19										
20	Provision									
21	Monat	bisherige Provision	gepl. bis. Prov.	reale bis. Prov.	Differenz	Prognose	Differenz			
22	Januar	4400	16666,67	1100	-15566,67	13200	-186800			
23	Februar	77400	33333,33	77400	44066,67	464400	264400			
24	März	77402	50000	77402	27402	309608	109608			
25										
26										
27										
28										
29										
30										
31										
32										
33										
34										
35										
36										
37										
38										
39										
40										
41										
42										
43										
44										
45										
46										
47										
48										
49										
50										
51										
52										
53										
54										
55										
56										
57										
58										
59										
60										
61										
62										
63										
64										
65										
66										
67										
68										
69										
70										
71										
72										
73										
74										
75										
76										
77										
78										
79										

Abbildung 18.1
Automatische Charterzeugung

```
    diagrammBreite = anzahlMonate * BREITENMULTIPLIKATOR
    ActiveSheet.ChartObjects.Delete
    Set myChart = ActiveSheet.ChartObjects.Add(CHART_X_KOORDINATE, CHART_BAR_Y_KOORDINATE, _
                  diagrammBreite, CHART_HOEHE)
    myChart.Chart.SetSourceData Source:=Sheets(TABELLENBLATT).Range(Cells(UMSATZ_START_ZEILE, _
            UMSATZ_START_SPALTE), Cells(letzteBesetzteZeile, PLOT_BEREICH_END_SPALTE)), _
                  PlotBy:=xlColumns
    myChart.Chart.ChartType = xlColumnClustered

    Set myChart = ActiveSheet.ChartObjects.Add(CHART_X_KOORDINATE, CHART_LINE_Y_KOORDINATE, ➥
        diagrammBreite, CHART_HOEHE)
    myChart.Chart.SetSourceData Source:=Sheets(TABELLENBLATT).Range(Cells(UMSATZ_START_ZEILE, ➥
        UMSATZ_START_SPALTE), _
                  Cells(letzteBesetzteZeile, PLOT_BEREICH_END_SPALTE)), PlotBy:=xlColumns
    myChart.Chart.ChartType = xlLine
End Sub
```

Auch hier beginnt alles mit der Deklaration von Variablen und Konstanten. Neu hier ist vielleicht

```
Dim myChart As Object
```

Dies ist die Variable, auf der wir das Chart abspeichern werden. Sie muss den Typ *Object* besitzen. Durch

```
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(UMSATZ_START_SPALTE, _
                UMSATZ_START_ZEILE, TABELLENBLATT)
anzahlMonate = letzteBesetzteZeile - UMSATZ_START_ZEILE
diagrammBreite = anzahlMonate * BREITENMULTIPLIKATOR
```

wird die Breite des Charts ermittelt. Denn, wenn Sie die Prognose auf Basis von November darstellen wollen, müssen Sie die Ergebnisse von 11 Monaten anzeigen, wenn die Basis hingegen März ist, wie in Abb. 18.1 hingegen nur drei. Naturgemäß möchten Sie hier unterschiedliche Breiten des Charts realisieren können. Das bedeutet, die Breite des Charts hängt von der Anzahl Monate ab, deren Verkaufsbeträge bisher eingegeben wurden.

```
ActiveSheet.ChartObjects.Delete
```

löscht alle eventuell bereits in das Tabellenblatt eingebetteten Charts. Würden wir nicht vorher eventuell vorhandene Diagramme löschen, dann würde jeder Click auf die Schaltfläche „Grafik“ ein neues Diagramm über den alten hinzufügen. Bei uns kostet das nur Speicherplatz, weil unsere Diagramme, wenn die Anzahl der Monate steigt, immer breiter werden. Man sieht die darunter liegenden also nicht. Das muss aber nicht immer so sein, darum gewöhnen wir uns an, eventuell bereits vorhandene Charts abzuschießen.

```
Set myChart = ActiveSheet.ChartObjects.Add(CHART_X_KOORDINATE, CHART_BAR_Y_KOORDINATE, _
                diagrammBreite, CHART_HOEHE)
```

fügt ein neues Diagramm hinzu. Die Funktion *ActiveSheet.ChartObjects.Add* erwartet als Parameter die x-Koordinate, die y-Koordinate, die Breite und die Höhe des Bereiches des Tabellenblatts, wo das Diagramm dargestellt werden soll.

```
myChart.Chart.SetSourceData Source:=Sheets(TABELLENBLATT).Range(Cells(UMSATZ_START_ZEILE, _
                UMSATZ_START_SPALTE), Cells(letzteBesetzteZeile, _
                PLOT_BEREICH_END_SPALTE)), PlotBy:=xlColumns
```

wird der Zellbereich, der geplottet werden soll festgelegt. *PlotBy:=xlColumns* besagt, dass die Werte in den Spalten stehen.

```
myChart.Chart.ChartType = xlColumnClustered
```

legt den Typ des Charts fest. *xlColumnClustered* steht für Balkendiagramm.

```
Set myChart = ActiveSheet.ChartObjects.Add(CHART_X_KOORDINATE, CHART_LINE_Y_KOORDINATE, _
            diagrammBreite, CHART_HOEHE)
myChart.Chart.SetSourceData Source:=Sheets(TABELLENBLATT).Range(Cells(UMSATZ_START_ZEILE, ➥
    UMSATZ_START_SPALTE), _
            Cells(letzteBesetzteZeile, PLOT_BEREICH_END_SPALTE)), PlotBy:=xlColumns
myChart.Chart.ChartType = xlLine
```

fügt nun auf exakt die gleiche Weise ein Liniendiagramm hinzu. Sie sehen, *xlLine* ist das Excel-Wort für Liniendiagramm. Wenn Sie in Ihren Anwendungen nun eigene Diagramme erstellen wollen, müssen Sie folgendes tun:

- Den Ort wo Ihr Diagramm erscheinen soll, anpassen: Das heißt: *CHART_X_KOORDINATE*, *CHART_HOEHE*, *CHART_BAR_Y_KOORDINATE* bzw. *CHART_LINE_Y_KOORDINATE* anpassen und die Breite des Bildes bestimmen.
- Den Zellbereich, der dargestellt werden soll anpassen: Das heißt: *Cells(UMSATZ_START_ZEILE, UMSATZ_START_SPALTE), Cells(letzteBesetzteZeile, PLOT_BEREICH_END_SPALTE)* ihren Anforderungen gemäß besetzen.
- Je nachdem, was Sie erzeugen wollen, den Code für das Balkendiagramm oder für das Liniendiagramm löschen.

So wie das Diagramm jetzt erzeugt wurde, werden für die Formatierungen des Diagramms die Voreinstellungen von Excel benutzt. Im nächsten Beispiel zeigen wir Ihnen, wie Sie diese ändern können. Das Chart soll jetzt wie in Abb. 18.2 dargestellt erscheinen.

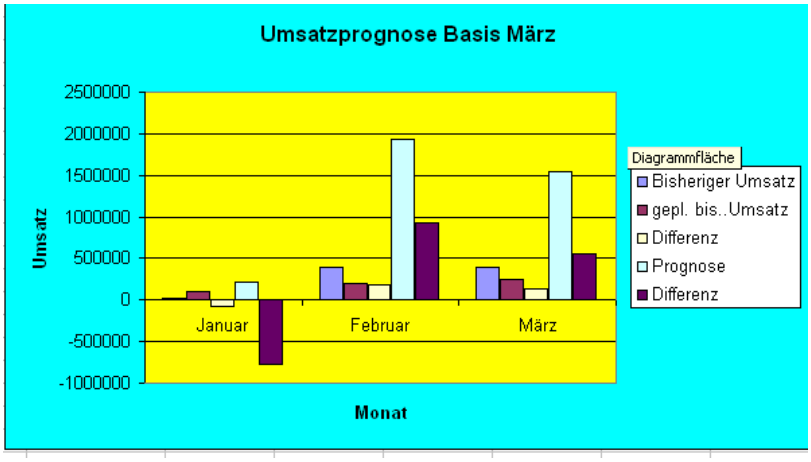


Abbildung 18.2
Eigene Formatierungen im Chart

Wir sehen eine Überschrift, Beschriftungen der x- und y-Achse sowie eine Hintergrundfarbe. Beispiel 18.2 zeigt die Programmierung:

Beispiel 18.2 *Grafische Darstellung der prognostizierten Umsatzentwicklung in Excel mit eigener Formatierung*

```
'Dateiname: provisionPrognoseMonatGrafik.xls
Sub erstelleChartMitBeispielFormatierungen_click()
    Dim letzteBesetzteZeile As Long
    Dim anzahlMonate As Long
    Dim monatAlsString As String
    Dim diagrammBreite As Long
    Dim myChart As Object

    Const TABELLENBLATT As Long = 2
    Const UMSATZ_START_SPALTE As Long = 1
    Const PLOT_BEREICH_END_SPALTE As Long = 6
    Const UMSATZ_START_ZEILE As Long = 5
    Const CHART_X_KOORDINATE As Long = 250
    Const CHART_BAR_Y_KOORDINATE As Long = 450
    Const CHART_LINE_Y_KOORDINATE As Long = 750
    Const CHART_HOEHE As Long = 250
    Const BREITENMULTIPLIKATOR As Long = 150

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(UMSATZ_START_SPALTE, ➥
        UMSATZ_START_ZEILE, TABELLENBLATT)
    anzahlMonate = letzteBesetzteZeile - UMSATZ_START_ZEILE
```

```
    diagrammBreite = anzahlMonate * BREITENMULTIPLIKATOR
    ActiveSheet.ChartObjects.Delete
    Set myChart = ActiveSheet.ChartObjects.Add(CHART_X_KOORDINATE, CHART_BAR_Y_KOORDINATE, ➥
        diagrammBreite, CHART_HOEHE)
    myChart.Chart.SetSourceData Source:=Sheets(TABELLENBLATT).Range(Cells(UMSATZ_START_ZEILE, ➥
        UMSATZ_START_SPALTE), Cells(letzteBesetzteZeile, PLOT_BEREICH_END_SPALTE)), PlotBy:=➥
        xlColumns
    myChart.Chart.ChartType = xlColumnClustered

    monatAlsString = erzeugeMonatNameAusLong(anzahlMonate)
    myChart.Chart.HasTitle = True
    myChart.Chart.ChartTitle.Characters.Text = "Umsatzprognose Basis " & monatAlsString
    myChart.Chart.Axes(xlCategory, xlPrimary).HasTitle = True
    myChart.Chart.Axes(xlCategory, xlPrimary).AxisTitle.Characters.Text = "Monat"
    myChart.Chart.Axes(xlValue, xlPrimary).HasTitle = True
    myChart.Chart.Axes(xlValue, xlPrimary).AxisTitle.Characters.Text = "Umsatz"
    myChart.Chart.ChartArea.Interior.ColorIndex = 28
    myChart.Chart.PlotArea.Interior.ColorIndex = 6
    myChart.Chart.HasLegend = True
    myChart.Chart.Legend.Position = xlBottom
    Set myChart = ActiveSheet.ChartObjects.Add(CHART_X_KOORDINATE, CHART_LINE_Y_KOORDINATE, ➥
        diagrammBreite, CHART_HOEHE)
    myChart.Chart.SetSourceData Source:=Sheets(TABELLENBLATT).Range(Cells(UMSATZ_START_ZEILE, ➥
        UMSATZ_START_SPALTE), Cells(letzteBesetzteZeile, PLOT_BEREICH_END_SPALTE)), PlotBy:=➥
        xlColumns
    myChart.Chart.ChartType = xlLine
End Sub
```

Neu hinzugekommen sind die Zeilen:

```
monatAlsString = erzeugeMonatNameAusLong(anzahlMonate)
myChart.Chart.HasTitle = True
myChart.Chart.ChartTitle.Characters.Text = "Umsatzprognose Basis " & monatAlsString
myChart.Chart.Axes(xlCategory, xlPrimary).HasTitle = True
myChart.Chart.Axes(xlCategory, xlPrimary).AxisTitle.Characters.Text = "Monat"
myChart.Chart.Axes(xlValue, xlPrimary).HasTitle = True
myChart.Chart.Axes(xlValue, xlPrimary).AxisTitle.Characters.Text = "Umsatz"
myChart.Chart.ChartArea.Interior.ColorIndex = 28
myChart.Chart.PlotArea.Interior.ColorIndex = 6
```

Diese sind aber, glauben wir, selbsterklärend. Wenn Sie die Excel-Zahlen Farben Zuordnung nicht kennen, können Sie selbstverständlich die in Kap. 17 eingeführte Excel-Funktion *rgb* benutzen. Wir haben den Makrorekorder genommen, um die Farben einzustellen, uns dann den erzeugten Code angesehen und die Farben gesucht. Diese haben wir dann in unser Programm eingefügt.

Kapitel 19

Dialoge / Formulare

Excel bietet neben den in Kap. 16 behandelten Meldungs- und Eingabefenstern weitere leistungsstarke Möglichkeiten der Kommunikation mit dem Benutzer. Wir können mit einem grafischen Editor Benutzeroberflächen mit allen, aus anderen Anwendungen geläufigen, Interaktionselementen erstellen. Wir veranschaulichen dies sofort: In unserem Provisionsbeispiel lassen wir die Benutzer zunächst über eine *InputBox* den Stützmonat der Hochrechnung eingeben. Danach fragen wir über eine MsgBox ab, ob linear oder nicht linear hochgerechnet werden soll. Der Benutzer wird vom Programm also zweimal „belästigt“. Schöner ist die in Abb. 19.1 dargestellte Benutzeroberfläche:

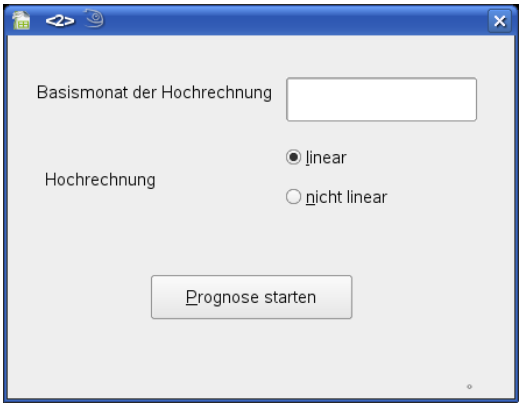


Abbildung 19.1
Der Eingabedialog des Provisionsbeispiels

Clickt der Benutzer auf die Schaltfläche „Prognose“ erscheint das in Abb. 19.1 dargestellte Fenster über der Tabellenkalkulation. Der Benutzer tätigt die benötigten Eingaben und betätigt danach die Schaltfläche „Prognose starten“. In diesem Beispiel finden Sie bereits vier Interaktionselemente: Ein Texteingabefeld (Textinput, für die Eingabe des Monates), zwei Optionsfelder (Radiobutton, für die Eingabe, auf welche Art hochgerechnet werden soll), eine Schaltfläche (Button, um die Durchführung der Berechnung zu starten) und zwei Beschriftungsfelder (Label). Fenster, wie das in Abb. 19.1 dargestellte, heißen in einer Tabellenkalkulation Dialog oder Formular. Wir werden beide Namen synonym verwenden.
Um den Dialogeditor von Excel zu starten, wechseln Sie zunächst in den Makro-Editor. Dort wählen Sie Einfügen ⇨ UserForm aus. Dialoge heißen auf microsoftisch UserForm. Es erscheint das in Abb. 19.2 dargestellte Bild.
Die UserForm sollte selektiert sein. Wenn nicht selektieren Sie sie[1]. Als erstes verändern wir die Eigenschaften des Formulars[2]. Dies geschieht im Eigenschaftsfenster. Wichtig für uns sind zunächst die Eigenschaften Name und Caption. Wir geben dem Formular den Namen *prognoseForm*[3]. Caption ist die Beschriftung des Formulars, also das, was im oberen blauen Balken des Formulars stehen wird. Da dies das ist, was unsere Benutzer sehen werden, erklären wir hier, zu was das Formular gut sein soll. Wir beschriften unser Formular mit „Eingaben zur Prognoserechnung“.

[1] Was, wie ein jeder weiß, über einen Click erfolgt.
[2] Wir verwenden nun Dialog, Formular und UserForm synonym.
[3] Leerschritte sind in Namen von Formularen nicht erlaubt.

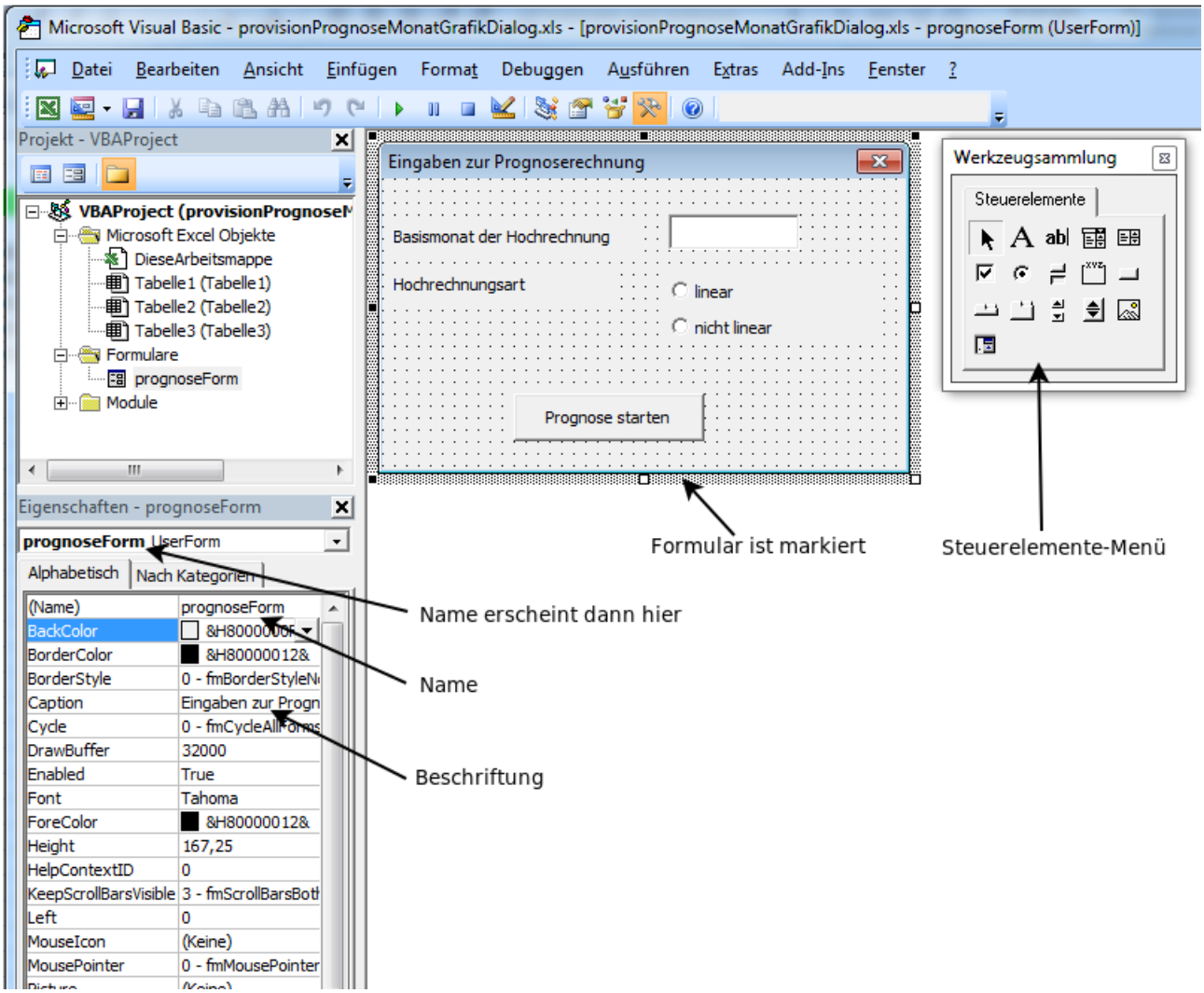


Abbildung 19.2
Der Dialogeditor in Excel

Zugleich mit unserem Formular erscheint die Steuerelemente-Toolbox[4]. Hier sind die Steuerelemente zu sehen, die in ein Excel-Formular integriert werden können. Unter Steuerelemente versteht Microsoft mögliche Elemente eines Formulars, wie Eingabefelder, Radio-Buttons, Auswahlfelder, Beschriftungsfelder etc. Sie kennen Sie schon aus Kap. 7. Dort sieht sie zwar komplett anders aus[5], hat aber den gleichen Inhalt. Denn die Steuerelemente, die wir hier betrachten, lassen sich auch direkt in die Tabellen einfügen. Wenn Sie mit der Maus über die Symbole der Steuerelemente-Toolbox fahren, wird Ihnen angezeigt, um welches Interaktionselement es sich handelt. Wählen Sie das Textfeld-Icon aus und führen Sie die Maus über das noch leere Formular. Dort können Sie nun das Textfeld an die von Ihnen gewünschte Position des Formulars malen. Das Eingabefeld bleibt selektiert und das Eigenschaftsfenster zeigt nun die Eigenschaften der Textbox an[6]. Auch hier ändern wir den Namen und nennen das Eingabefeld *monatInput*. Wir malen nun zwei Optionsfelder, zwei Beschriftungsfelder und eine Schaltfläche in das Formular. Die Optionsfelder bekommen die Namen *linearOption* bzw. *nichtLinearOption*. Bei allen neu eingefügten Feldern müssen wir die Beschriftung ändern. Dies geschieht in Excel durch Änderung der Eigenschaft „Caption“ im Eigenschaftsfenster. Wir vergeben sie, wie in Abb. 19.1 dargestellt.
Die Optionsfelder müssen weiter angepasst werden. Von ihnen darf ja nur eins auswählbar sein. Dies erreichen wir, indem

[4] Der Steuerelemente-Werkzeugkasten, das Steuerelemente-Menü.
[5] Wahrscheinlich, damit man sie leichter *nicht* wieder erkennt ☺.
[6] Wir könnten zu den Eigenschaften der Userform zurückkehren, indem wir irgendwo, wo kein Steuerelement ist, in die UserForm clicken.

wir der Eigenschaft „GroupName“ bei beiden denselben Wert (im Beispiel „Hochrechnungsart“) geben. Abb. 19.3 zeigt das Eigenschaftsfenster des *linearOption* Optionsfelds.

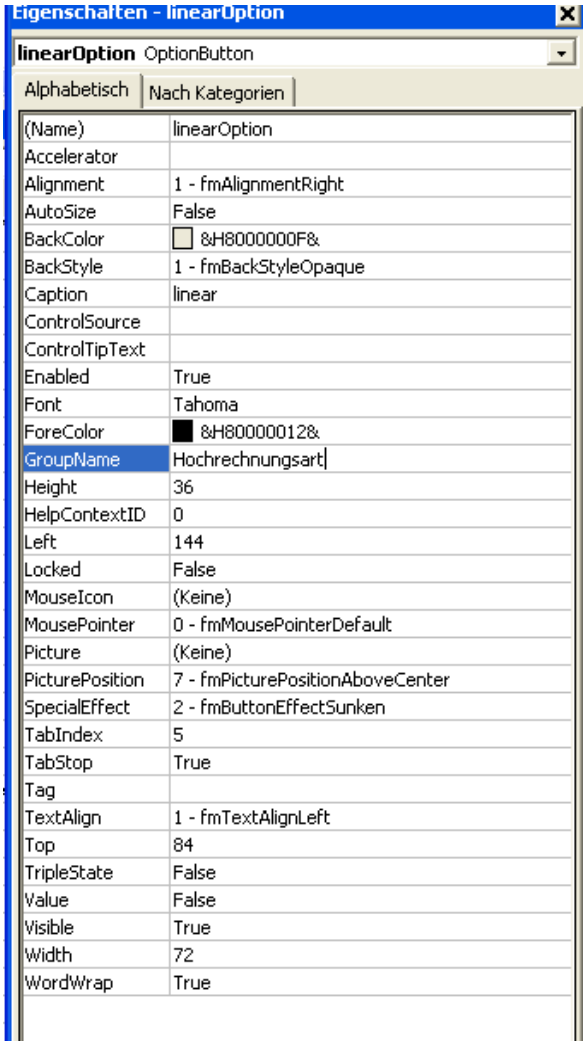


Abbildung 19.3
Das Eigenschaftsfenster eines Optionsfelds

Die Schaltfläche bekommt ebenfalls einen Namen, wir vergeben den Namen *berechnenButton*. Damit wissen wir auch bereits (aus Kap. 7), wie diese Schaltfläche mit VBA-Code verbunden wird: Wir clicken doppelt auf den Button und VBA verzweigt in ein Code Fenster und erzeugt die Prozedur *berechnenButton_Click*. So weit wollen wir aber zur Zeit nicht gehen ☺. Zunächst legen wir die Reihenfolge fest, in der die Steuerelemente angesprungen werden, wenn ein Benutzer mit der Tabulator-Taste durch das Formular navigiert. Dazu clicken wir mit der rechten Maustaste irgendwo in das Formular. Im nun erscheinenden Kontextmenü wählen wir „Aktivierreihenfolge“ aus. Das in Abb. 19.4 dargestellte Fenster erscheint. Die Bedienung dieses Fensters sollte sich intuitiv erschließen[7]. Damit haben wir alles, was sich mit dem Dialogeditor erzeugen läßt, fertig. Eine Schwachstelle hat unser Entwurf allerdings noch: Wenn sich der Dialog öffnet, ist keines der Optionsfelder ausgewählt. Dies ist aus zwei Gründen nicht schön:

- Normalerweise weiß man, was häufiger vorkommt: lineare oder nicht lineare Prognose. Darum sollte diese Option auch voreingestellt sein.
- Präsentieren wir dem Benutzer die Optionsfelder ohne Vorbelegung, so kann der Benutzer, nachdem er einmal

[7] Welch ein schöner Satz!

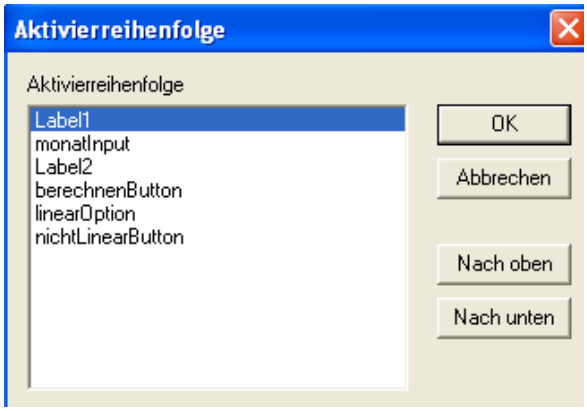


Abbildung 19.4
Aktivierreihenfolge in Excel

eine Auswahl getroffen hat, den Ursprungszustand nicht wieder herstellen. So etwas sollte man beim Design einer Benutzeroberfläche nie tun.

Aus nicht wirklich ersichtlichen Gründen kann man das im Dialogeditor nicht machen. Dazu müssen wir programmieren. Um Code für ein Formular zu schreiben, clicken wir mit der rechten Maustaste in das Formular und wählen im erscheinenden Kontextmenü „Code anzeigen“ aus. Ein Programmierfenster mit dem zum Formular gehörigen Code öffnet sich[8]. Das erscheinende Codefenster ist leer. Wir schreiben folgende Prozedur:

Beispiel 19.1 *Vorbelegung des Optionsfelds in Excel: Userform_initialize*

```
'Dateiname: provisionPrognoseMonatGrafikDialog.xls
Sub Userform_initialize()
    linearOption.Value = True
End Sub
```

Existiert in einem zu einem Formular gehörigen Codefenster die Prozedur *Userform_initialize*, so wird diese, wenn das Formular geladen wird, durchgeführt. Unserem Optionsfeld für die lineare Prognose hatten wir den Namen *linearOption* gegeben. Durch die Anweisung

```
linearOption.Value = True
```

sorgen wir dafür, dass dieses Optionsfeld vorausgewählt ist. Nun ist das Formular fertiggestellt und wir könnten es uns zumindest schon einmal anschauen. Das Formular soll ja angezeigt werden, wenn der Benutzer im Tabellenblatt auf die Schaltfläche „Prognose“ clickt. Mit dieser Schaltfläche ist unsere Ereignisprozedur *kontrolliereUmsatzMonatlich_Click* verbunden. Diese müssen wir so verändern, dass das Formular angezeigt wird. Wir benennen die ursprüngliche Prozedur *kontrolliereUmsatzMonatlich_Click* um in *kontrolliereUmsatzMonatlichBerechnung*, so dass wir zur Zeit über gar keine Ereignisprozedur verfügen und die Schaltfläche „Prognose“ damit funktionslos ist. Diesen Zustand beenden wir sofort dadurch, dass wir eine neue Ereignisprozedur schreiben:

Beispiel 19.2 *Die neue Ereignisprozedur der „Prognose“ Schaltfläche: Der Dialog wird aufgeblendet*

```
'Dateiname: provisionPrognoseMonatGrafikDialog.xls
Sub kontrolliereUmsatzMonatlich_Click()
    prognoseForm.Show
End Sub
```

[8] Oh ja, Microsoft ist ziemlich freigiebig mit Codefenstern, es gibt eins für jede Tabelle, eins für die Arbeitsmappe, eins für jedes Formular und dann noch die Module. Irgendwer wird wissen, warum.

Dem Dialog hatten wir den Namen *prognoseForm* gegeben. Einen Dialog blendet man also auf, in dem man an den Namen Dialogs *.Show* anhängt. Und das ist die ganze neue Ereignisprozedur. Denn die weitere Verarbeitung wird ja dadurch ausgelöst, dass die Benutzer im Formular auf die Schaltfläche mit der Beschriftung „Prognose starten“ clicken. Die zugehörige Ereignisprozedur müssen wir jetzt erstellen.
Durch einen Click auf den Formularnamen im Projektfenster kehren wir in den Dialogeditor zurück. Ein Doppelclick auf die Schaltfläche veranlasst Excel, in das Codefenster zurückzukehren. Wie Sie das schon aus Kap. 7 kennen, hat Excel nun eine Ereignisprozedur für die Schaltfläche angelegt. Da wir die Schaltfläche *berechnenButton* genannt haben, heißt die Ereignisprozedur *berechnenButton_Click*. Sie wird durchgeführt, wenn der Benutzer auf die Schaltfläche clickt. Was muss die Ereignisprozedur tun? Sie muss

- den Monat aus *monatInput* auslesen,
- prüfen, ob die Eingabe des Monats korrekt war und
- feststellen, welches Optionsfeld ausgewählt ist.

Dann sollte Sie das Formular entfernen und zum Schluss die Prognoserechnung anstoßen.
Die Implementierung zeigt Beispiel 19.3:

Beispiel 19.3 *Die Ereignisprozedur des Dialogs in Excel*

```
'Dateiname: provisionPrognoseMonatGrafikDialog.xls
Sub berechnenButton_Click()
    Dim linear As Boolean
    Dim hochrechnungsMonatAlsString As String
    Dim hochrechnungsMonat As Long
    hochrechnungsMonatAlsString = monatInput.Text
    If Not wandleInLongUm(hochrechnungsMonatAlsString, hochrechnungsMonat) Then
        MsgBox ("Die Monatseingabe muss eine Zahl sein!")
        Exit Sub
    End If
    If Not istGueltigerMonat(hochrechnungsMonat) Then
        MsgBox ("Die Monatseingabe ist nicht richtig!!")
        Exit Sub
    End If
    linear = linearOption.Value
    Call kontrolliereUmsatzMonatlichBerechnung(hochrechnungsMonat, linear)
    Unload Me
End Sub

Function wandleInLongUm(eingabe As String, rueckgabe As Long) As Boolean
    If Not IsNumeric(eingabe) Then
        wandleInLongUm = False
        Exit Function
    End If
    rueckgabe = CLng(eingabe)
    wandleInLongUm = True
End Function

Function istGueltigerMonat(monat As Long) As Boolean
    If (monat > 12) Or (monat < 1) Then
        istGueltigerMonat = False
    Else
        istGueltigerMonat = True
    End If
End Function
```

Durch

```
hochrechnungsMonatAlsString = monatInput.Text
```

wird in Excel der Inhalt eines Textfeldes geholt. Wir sehen, wir benötigen hier nur den Namen des Textfeldes. Das Textfeld hatten wir *monatInput* genannt. *monatInput.Text* enthält dann den Inhalt des Textfelds als String. In den nächsten Zeilen werden Plausibilitätsprüfungen und die Typumwandlung durchgeführt.

```
    If Not wandleInLongUm(hochrechnungsMonatAlsString, hochrechnungsMonat) Then
        MsgBox ("Die Monatseingabe muss eine Zahl sein!")
        Exit Sub
    End If
    If Not istGueltigerMonat(hochrechnungsMonat) Then
        MsgBox ("Die Monatseingabe ist nicht richtig!!")
        Exit Sub
    End If
```

Die hinter den Plausibilitätsprüfungen stehenden Ideen entsprechenden den Prüfungen aus Kapitel 15, nur dass wir hier die Fehler direkt in einer MsgBox ausgeben.
In der Zeile

```
linear = linearOption.Value
```

wird der Zustand des Optionsfelds *linearOption* geholt. *linearOption.Value* gibt *true* zurück, wenn das Optionsfeld selektiert ist und *false*, wenn es nicht selektiert ist[9]. Die letzte Zeile der Prozedur

```
Unload Me
```

entfernt das Formular. Durch

```
Call kontrolliereUmsatzMonatlichBerechnung(hochrechnungsMonat, linear)
```

wird die Prognoserechnung angestoßen. Sie erinnern sich, wir hatten unsere alte Ereignisprozedur *kontrolliereUmsatzMonatlich_Click* in *kontrolliereUmsatzMonatlichBerechnen* umbenannt. Natürlich hatten wir zu diesem Zeitpunkt noch keine Übergabeparameter definiert, weil wir ja nur den Namen *kontrolliereUmsatzMonatlich_Click()* für unsere neue Ereignisprozedur benötigten. Nun müssen wir *kontrolliereUmsatzMonatlichBerechnen* umschreiben. Diese Prozedur wird zwei Übergabeparameter bekommen, nämlich *hochrechnungsMonat* (Datentyp Long) und *linear* (Datentyp Boolean). Wir löschen dann im Deklarationsteil die Variablen *hochrechnungsMonat* und *linear*, weil sie ja nun schon in der Übergabeliste stehen. Sodann entfernen wir die Aufrufe der *MsgBox* und der *InputBox*, weil die Eingaben ja über das Formular getätigt werden und über die Übergabeliste in die Funktion kommen. Und zum Schluss verändern wir beide Zeilen, in denen über den Vergleich

```
if linear=6 then
```

entschieden wird, ob die lineare oder die nicht lineare Verarbeitung durchgeführt wird in:

```
if linear then
```

Die Variable *linear* hat in unserer neuen Konstruktion den Wert *true*, wenn linear hochgerechnet werden soll. Das aber sind alle Veränderungen an der neuen Prozedur *kontrolliereUmsatzMonatlichBerechnen*. Sie ist in Beispiel 19.4 dargestellt.

Beispiel 19.4 *Die Berechnung der Prognose aus dem Formular in Excel*

```
'Dateiname: provisionPrognoseMonatGrafikDialog.xls
Sub kontrolliereUmsatzMonatlichBerechnung(hochrechnungsMonat As Long, linear As Boolean)
    Dim prozentualeVerteilungArray() As Double
    Dim monat As Long
    Dim alterMonat As Long
    Dim alterMonatAlsString As String
    Dim letzteBesetzteZeile As Long
    Dim bisherigerUmsatz As Double
    Dim prognostizierterJahresumsatz As Double
    Dim geplanterJahresumsatz As Double
    Dim geplanterUmsatzBisMonat As Double
    Dim geplanteProvision As Double
    Dim geplanteProvisionBisMonat As Double
    Dim prozentualeUmsatzverteilungArray(1 To 12) As Double
```

[9] Dies steht auch im Einklang mit der gerade geschriebenen Funktion *Userform_initialize*.

```
    Dim farbe As Long
    Dim umsatzDifferenz As Double
    Dim realeProvisionBisMonat As Double
    Dim prognostizierteProvision As Double
    Dim umsatzMonatsDifferenz As Double
    Dim provisionsDifferenzMonat As Double
    Dim provisionsDifferenz As Double
    Dim hochrechnungstext As String
    Dim hochrechnungsMonatAlsString As String
    Dim datum As Date
    Dim bisherigeProvision As Double
    Dim i As Long

    Const DATUMSSPALTE As Long = 1
    Const VERKAUFSBETRAGSPALTE As Long = 2
    Const PROVISIONSSPALTE As Long = 3
    Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Long = 5
    Const BISHERIGER_VERKAUFSBETRAG_SPALTE As Long = 2
    Const BISHERIGER_VERKAUFSBETRAG_ZEILE As Long = 2
    Const PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE As Long = 4
    Const PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE As Long = 2
    Const GEPLANTER_UMSATZ_SPALTE As Long = 2
    Const GEPLANTER_UMSATZ_ZEILE As Long = 1
    Const TABELLENBLATT As Long = 1

    hochrechnungsMonatAlsString = erzeugeMonatNameAusLong(hochrechnungsMonat)
    If linear Then
        'linear hochrechnen
        hochrechnungstext = "Die Hochrechnung erfolgte linear zum Monat " & ➥
            hochrechnungsMonatAlsString
    Else
        Call liesNichtLineareUmsatzverteilungEin(2, PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, _
                PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE, prozentualeUmsatzverteilungArray)
        hochrechnungstext = "Die Hochrechnung erfolgte nicht linear zum Monat " & ➥
            hochrechnungsMonatAlsString
    End If
    geplanterJahresumsatz = Cells(GEPLANTER_UMSATZ_ZEILE, GEPLANTER_UMSATZ_SPALTE)
    geplanteProvision = berechneProvision(geplanterJahresumsatz, geplanterJahresumsatz)

    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(DATUMSSPALTE, ➥
        ERSTE_ZEILE_MIT_VERKAUFSBETRAG, TABELLENBLATT)
    alterMonat = 1
    bisherigeProvision = 0
    bisherigerUmsatz = 0
    For i = ERSTE_ZEILE_MIT_VERKAUFSBETRAG To letzteBesetzteZeile + 1
        If Not IsEmpty(Cells(i, DATUMSSPALTE)) Then
            datum = Cells(i, DATUMSSPALTE)
            monat = Month(datum)
        End If
        If (monat <> alterMonat) Or (i = letzteBesetzteZeile + 1) Then
            'Monatswechsel Prognose und Werte müssen geschrieben werden
            alterMonatAlsString = erzeugeMonatNameAusLong(alterMonat)
            If linear Then
                prognostizierterJahresumsatz = (bisherigerUmsatz * 12) / alterMonat
                geplanterUmsatzBisMonat = (geplanterJahresumsatz * alterMonat) / 12
                geplanteProvisionBisMonat = (geplanteProvision * alterMonat) / 12
            Else
                prognostizierterJahresumsatz = prognostiziereJahresumsatzNichtLinear(➥
                    bisherigerUmsatz, alterMonat, prozentualeUmsatzverteilungArray)
                geplanterUmsatzBisMonat = bisZuDiesemMonatZuErreichen(geplanterJahresumsatz, ➥
                    alterMonat, prozentualeUmsatzverteilungArray)
                geplanteProvisionBisMonat = bisZuDiesemMonatZuErreichen(geplanteProvision, ➥
                    alterMonat, prozentualeUmsatzverteilungArray)
            End If
            realeProvisionBisMonat = berechneProvision(prognostizierterJahresumsatz, ➥
                bisherigerUmsatz)
            prognostizierteProvision = berechneProvision(prognostizierterJahresumsatz, ➥
                prognostizierterJahresumsatz)
```

```
            umsatzMonatsDifferenz = bisherigerUmsatz - geplanterUmsatzBisMonat
            umsatzDifferenz = prognostizierterJahresumsatz - geplanterJahresumsatz
            provisionsDifferenzMonat = realeProvisionBisMonat - geplanteProvisionBisMonat
            provisionsDifferenz = prognostizierteProvision - geplanteProvision
            If umsatzDifferenz > 0 Then
                farbe = RGB(0, 255, 0)
            Else
                farbe = RGB(200, 0, 0)
            End If
            Call schreibeUmsatzUndProvision(bisherigerUmsatz, geplanterUmsatzBisMonat, ➥
                umsatzMonatsDifferenz, prognostizierterJahresumsatz, _
                            umsatzDifferenz, bisherigeProvision, geplanteProvisionBisMonat, ➥
                                realeProvisionBisMonat, _
                            provisionsDifferenzMonat, prognostizierteProvision, ➥
                                provisionsDifferenz, _
                            alterMonat, alterMonatAlsString, farbe)
            alterMonat = monat
        End If
        If (monat > hochrechnungsMonat) Or (i = letzteBesetzteZeile + 1) Then
            Exit For
        End If
        bisherigeProvision = bisherigeProvision + Cells(i, PROVISIONSSPALTE)
        bisherigerUmsatz = bisherigerUmsatz + Cells(i, VERKAUFSBETRAGSPALTE)
    Next i
    Call schreibePrognose(prognostizierterJahresumsatz, umsatzDifferenz, hochrechnungstext,farbe)
End Sub
```

Sie müssen hier allerdings noch folgendes beachten: Ursprünglich stand die Ereignisprozedur *kontrolliereUmsatzMonatlich_Click* im zu Tabelle 1 gehörenden Codefenster. Das muss auch so sein, denn alle Ereignisprozeduren, die sich auf Tabelle 1 beziehen, müssen sich dort befinden. Die Ereignisprozedur *berechnenButton_Click* befindet sich aber im zum Formular *prognoseForm* gehörigen Codefenster. Von diesem hat man leider keinen Zugriff auf die Funktionen und Prozeduren im Codefenster zu Tabelle 1. Daher müssen wir alle Prozeduren und Funktionen, außer den Ereignisprozeduren, aus dem Codefenster zu Tabelle 1 in ein Codefenster unter Module kopieren, denn die Prozeduren und Funktionen dort sind von überall zugänglich.

Kapitel 20

Zugriff auf Datenbanken

20.1 ODBC

ODBC (Open Database Connectivity) ist eine offene Datenbankschnittstelle. Mittels ODBC ist es möglich, auf Datenbankensysteme zuzugreifen, welche entweder lokal oder auf entfernten Servern vorhanden sind. ODBC verwendet zum Manipulieren von Datenbanken/Tabellen SQL.
ODBC erlaubt selbstgeschriebenen Programmen, aber auch Anwendungen wie z.B. MS Access, über ein Netzwerk auf einen Datenbankserver zuzugreifen und Daten in dort verhandenen Datenbanken zu lesen und zu schreiben[1]. ODBC wird von allen führenden Datenbankherstellern unterstützt. Hersteller von Office-Paketen unterstützen ebenfalls ODBC.
Um von einem Windows-PC mit ODBC auf einen Server zuzugreifen, muss zunächst ein ODBC-Treiber für das anzusprechende Datenbanksystem installiert werden[2]. Um zu testen, ob die Installation erfolgreich war, ist es sinnvoll, anschließend ist eine Verbindung zur Datenbank zu konfigurieren[3]. Dies geschieht, indem man eine DSN (Data Source Name) anlegt. Hierzu wählt man in der Windows-Systemsteuerung den Punkt Verwaltung und in dem dann aufgehenden Fenster den Punkt Datenquellen (ODBC). In dem dann aufgeblendeten Fenster muss der Punkt „Hinzufügen“ angeclickt werden. In dem nun erscheinenden Fenster wird der für das anzusprechende Datenbanksystem verantwortliche Treiber ausgewählt (vgl. Abb. 20.1).

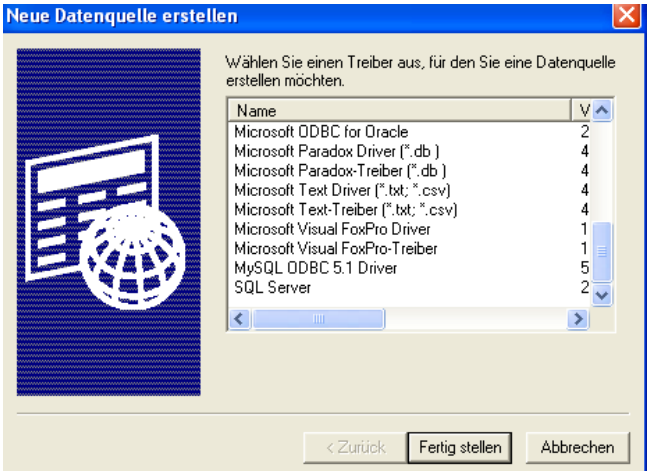


Abbildung 20.1
Auswahl des ODBC-Treibers

[1] Dies natürlich nur, wenn man die notwendigen Rechte besitzt.
[2] Das dies von dem verwendenten Datenbanksystem und dem entsprechenden Treiber abhängt, erklären wir hier nicht, wie man dies genau macht. Es finden sich aber zu jedem Treiber unzählige Anleitungen im Internet.
[3] Je nach Konfiguration des Computers, muss anscheinend eine solche Verbindung einmalig nach der Installation erstellt werden, damit die VBA-Programme den Treiber später verwenden können.

Nach Betätigen der Schaltfläche „Fertig stellen“ vergeben wir im nächsten Fenster einen Namen für die Verbindung (Feld: Data Source Name) und stellen den gewünschten Rechner und die gewünschte Datenbank ein. Nach Eingabe von User und Password wird der Datenbank-Dropdown mit den Datenbanken, für die der User Rechte hat, gefüllt. Zu diesem Zeitpunkt wissen wir bereits, dass die Installation erfolgreich war, denn die dargestellten Datenbanken sind bereits eine Antwort vom Datenbankserver. Sie clicken nun auf die Schaltfläche „Test“ (vgl. Abb. 20.2). Ein kleines Fenster mit der Meldung „connection successful“ erscheint. Die Installation des Treibers war erfolgreich .

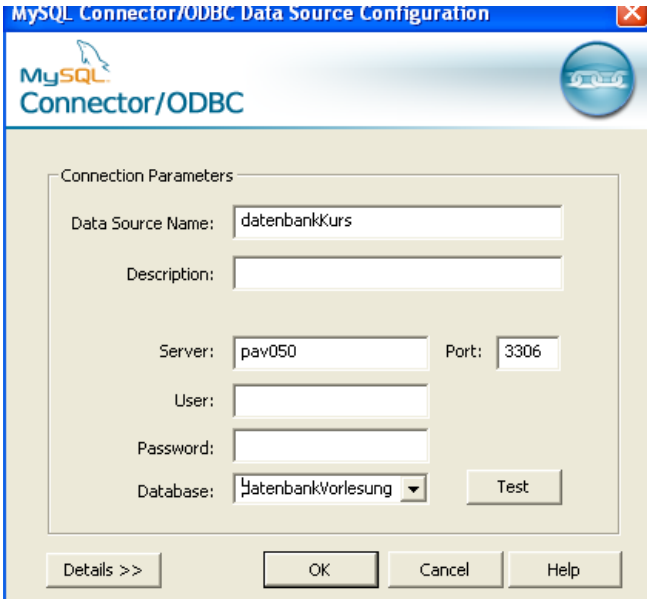


Abbildung 20.2
Einstellung der Parameter der Verbindung

20.2 Lesen aus Datenbanken von Excel

20.2.1 Erstes einfaches Beispiel

Wir demonstrieren die Vorgehensweise an einem Beispiel. Wir betrachten die Tabelle „Kunde“, deren Struktur in Abb. 20.3 dargestellt ist[4]. Wir wollen die Namen und PLZ der Kunden auslesen und in eine Excel-Tabelle schreiben. Hierzu benötigen wir folgende Informationen:

- Die Datenbank heißt datenbankVorlesung, sie liegt auf dem Server pav050.fh-bochum.de.
- Die Tabelle der Kunden heißt Kunde und hat die in in Abb. 20.3 dargestellte Struktur.
- Ein Benutzer mit Zugriffsrechten auf diese Datenbank heißt wiInf und hat das Passwort wiInf.
- Die benötigten Felder in der Tabelle Kunde sind Name und PLZ.

VBA verfügt über mehrere Objektbibliotheken zum Zugriff auf ODBC-Datenbanken. Die derzeit aktuelle heißt ADO[5] (ActiveX Data Objects). Vor der Benutzung muss die Bibliothek in VBA angemeldet werden. Dazu wählt man im VBA-Editor den Unterpunkt Verweise im Menü Extras aus, und setzt im sich dann öffnenden Fenster ein Häkchen an die Microsoft ActiveX Data Objects Library[6].
Den Sourcecode der Anwendung zeigt Beispiel 20.1.

[4] Die Tabelle haben Sie bereits in in der Datenbankvorlesung kennen gelernt.
[5] Was sich bei Microsoft mit jeder Version des Office-Paketes und damit von VBA ändert. Die aktuelle heißt ADO.net und unterscheidet sich von allen voran gegangenenen. Aber das ist bei Microsoft immer so.
[6] Der Punkt kommt unter M! :-))

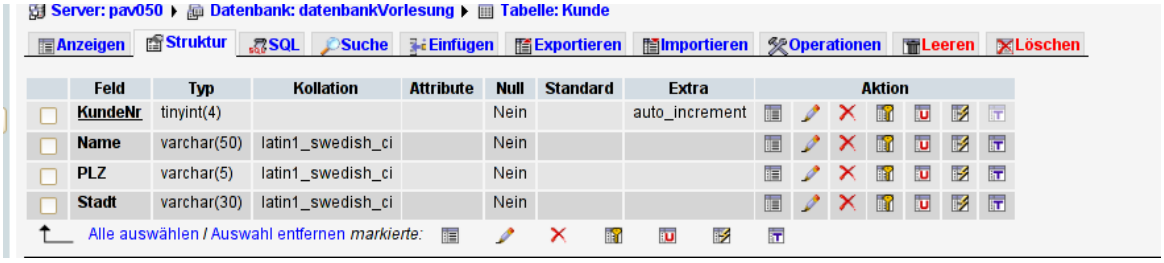


Abbildung 20.3
Die Tabelle Kunde aus der Datenbankvorlesung

Beispiel 20.1 *Lesen aus einer ODBC-Datenquelle*

```
'Dateiname: dbConnect.xls
Sub kundenAuslesen()
    Dim conn As ADODB.Connection
    Dim ConnectionString As String
    Dim rec As ADODB.Recordset
    Dim SQL As String
    Dim i As Long
    Const TABELLENBLATT As Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2

    ConnectionString = "Driver=MYSQL ODBC 8.0 Unicode Driver"
    ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
    ConnectionString = ConnectionString & ";Database=datenbankVorlesung"
    ConnectionString = ConnectionString & ";UID=wiInf"
    ConnectionString = ConnectionString & ";PWD=wiInf"
    Set conn = New ADODB.Connection
    conn.ConnectionString = ConnectionString
    conn.Open
    SQL = "Select "
    SQL = SQL & "Name, "
    SQL = SQL & "PLZ "
    SQL = SQL & "from Kunde "
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    'Ueberschriften
    Sheets(TABELLENBLATT).Cells(1, 1) = "Name"
    Sheets(TABELLENBLATT).Cells(1, 2) = "PLZ"
    'Nun Inhalte
    i = ERSTE_ZEILE_MIT_INFORMATIONEN
    Do While Not rec.EOF
        Sheets(TABELLENBLATT).Cells(i, 1) = rec!Name
        Sheets(TABELLENBLATT).Cells(i, 2) = rec!PLZ
        rec.MoveNext
        i = i + 1
    Loop
End Sub
```

Wir gehen den Code nun im Einzelnen durch.

```
Dim conn As ADODB.Connection
Dim ConnectionString As String
Dim rec As ADODB.Recordset
Dim SQL As String
Dim i As Long
Const TABELLENBLATT As Long = 1
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
```

In unserem Programm definieren wir fünf Variablen und eine Konstante. Zunächst die eher einfachen. Um Informationen aus einer Datenbank zu bekommen, müssen wir SQL-Anweisungen an die Datenbank schicken. Die SQL-Anweisung

setzen wir auf der String-Variable SQL zusammen. Beim Verbindungsaufbau mit der Datenbank müssen wir Parameter in einer bestimmten Form übergeben. Dazu dient die String-Variable ConnectionString. Die beiden nächsten Variablen sehen schwieriger aus. Bei ihnen handelt es sich um Objekte, was wir ja nicht besprochen haben. Für unsere jetzigen Zwecke reicht es aber aus, wenn Sie sich merken, dass Verbindungen zur Datenbank auf Variablen abgespeichert werden müssen, und dass diese Variablen den Datentyp ADODB.Connection haben müssen. Die Ergebnisse unserer Abfragen erhalten wir ebenfalls auf einer Variablen und diese muss vom Typ ADODB.Recordset sein. Weiterhin benötigen wir eine Variable, um das Suchergebnis zu durchlaufen. Dafür benutzen wir die Variable *i*. Und in der Konstanten *TABELLEBLATT* wird festgelegt, in welches Tabellenblatt die Daten geschrieben werden sollen.
Doch gehen wir nun weiter. In den Zeilen

```
ConnectionString = "Driver=MYSQL ODBC 8.0 Unicode Driver"
ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
ConnectionString = ConnectionString & ";Database=datenbankVorlesung"
ConnectionString = ConnectionString & ";UID=wiInf"
ConnectionString = ConnectionString & ";PWD=wiInf"
```

werden die zum Verbindungsaufbau notwendigen Informationen bereitgestellt. Die Zeile Provider=MSDASQL.1 sagt VBA, dass es sich um eine ODBC-Verbindung handeln soll[7]. MYSQL ODBC 8.0 UnicodeDriver ist der Name des installierten ODBC-Treibers[8]. Die restlichen Zeilen sollten selbsterklärend sein. Alle Angaben sind notwendig, um die Datenbankverbindung aufzubauen.
Als nächstes müssen wir eine Variable vom Typ ADODB.Connection erzeugen. Variablen, die Objekte enthalten, müssen immer erzeugt werden. Aber auch hier müssen Sie nicht verstehen, warum das so ist, sondern sie müssen nur wissen, das das so ist und wie man das macht. Die nächste Zeile zeigt, wie man eine Variable vom Typ ADODB.Connection erzeugt:

```
Set conn = New ADODB.Connection
```

Mit der Zeile

```
conn.ConnectionString = ConnectionString
```

werden die Verbindungsinformationen der Variablen conn zugewiesen, das Kommando

```
conn.open
```

öffnet die Verbindung zur Datenbank. In den nächsten Zeilen des Programms wird das SQL-Kommando, das wir benötigen, um Namen und PLZ aus der Tabelle Kunde der Datenbank zu lesen, auf der Variablen SQL abgespeichert:

```
SQL = "Select "
SQL = SQL & "Name, "
SQL = SQL & "PLZ "
SQL = SQL & "from Kunde "
```

Dies Kommando sollte jeder verstehen können. Anderenfalls sehen Sie im Datenbank-Script nach. Nun müssen wir

1. dieses SQL-Kommando über unsere Datenbank-Verbindung an das Datenbank-System schicken,
2. das Datenbanksystem anweisen, das SQL-Kommando auszuführen und
3. das Resultat der Abfrage über unsere Verbindung zurück erhalten.

Dies geschieht durch die Kommandos:

```
Set rec = New ADODB.Recordset
rec.Open SQL, conn
```

Die erste Anweisung erzeugt ein Recordset. Ein Recordset in VBA ist die Lösung der oben angesprochenen Punkte. Ein Recordset kann SQL-Anweisungen über eine Verbindung an eine Datenbank schicken, und die von der Datenbank erzielten Ergebnisse zurück bekommen. In den Variablendeklarationen dieses Programms hatten wir ja bereits eine Variable vom Typ ADODB.Recordset deklariert. Wie ebenfalls bereits angesprochen sind Recordsets Objekte und müssen erzeugt werden.

[7] Wie die bei Microsoft auf diesen Namen gekommen sind, ist uns auch nicht ganz klar.
[8] Den Namen des Treibers kann bei Ihrer Installation anders sein. Sie können ihn aus dem Dialog in Abb. 20.1 ermitteln.

```
Set rec = New ADODB.Recordset
```

erzeugt nun ein neues Objekt vom Typ ADODB.Recordset. Auch hier müssen Sie nicht begreifen, warum das so ist, Sie müssen nur behalten, das man das so machen muss.

```
rec.Open SQL, conn
```

schickt nun unser SQL-Kommando an die Datenbank und holt sich das Ergebnis zurück. Um das weitere verstehen zu können, müssen wir uns jetzt veranschaulichen, wie das von rec.Open geholte Ergebnis aussieht. Abb. 20.4 illustriert einen Recordset.

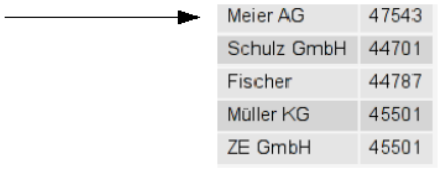


Abbildung 20.4
Grafische Darstellung eines Recordset

Ein Recordset enthält alle Datensätze, die das SQL-Kommando gefunden hat, plus einen Zeiger, der in Abb. 20.4 durch einen Pfeil veranschaulicht wird. Zu Anfang zeigt der Zeiger (wie auch in Abb. 20.4 dargestellt) auf den ersten gefundenen Datensatz. Nun stellt sich die Frage, wie wir die Ergebnisse aus dem Recordset heraus holen. Folgende Zeile[9]

```
name = rec!Name
```

holt den Inhalt des Datenbankattributs Name aus dem recordset und zwar aus dem Datensatz, auf den der Zeiger gerade deutet. Da dies zu Anfang der erste Datensatz ist, würde die Variable *name* nun den Wert „Meier AG“ haben.
Allgemein kann man sagen: Um ein Feld aus einem Recordset zu holen, schreibt man Name des Recordsets (in unserem Fall *rec*), ein Ausrufungszeichen und dann den Namen des Attributs. Verwenden kann man die Attribute, die über das Select definiert sind, in unserem Beispiel gibt es also neben rec!Name noch rec!PLZ. Wie schon gesagt beziehen sich rec!Name und rec!PLZ immer auf den Datensatz im Recordset, auf den der Zeiger gerade gerichtet ist.
Bislang können wir also nur den Inhalt des ersten Datensatzes aus dem Recordset ansprechen. Um den nächsten Datensatz bearbeiten zu können, muss man den Zeiger einen Datensatz weiter nach unten schieben. Dies geschieht durch das Kommando

```
rec.moveNext
```

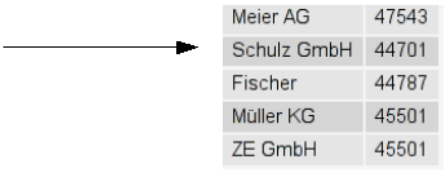


Abbildung 20.5
Unser Recordset nach rec.moveNext

[9] Vorausgesetzt eine Variable *name* vom Typ String ist deklariert.

Nach diesem Kommando sieht unser Recordset, wie in Abb. 20.5 dargestellt, aus.
Mit diesem Wissen können Sie jetzt den Rest des Beispiels verstehen. Die nächsten vier Zeilen sind einfach:

```
'Ueberschriften
Sheets(TABELLENBLATT).Cells(1, 1) = "Name"
Sheets(TABELLENBLATT).Cells(1, 2) = "PLZ"
i=ERSTE_ZEILE_MIT_INFORMATIONEN
```

Ein Kommentar, gefolgt von 2 Anweisungen, die Name, bzw. PLZ in die Zellen A1, resp. B1 des ersten Tabellenblatts schreiben. Dann wird die Variable i mit dem, in der Konstanten ERSTE_ZEILE_MIT_INFORMATIONEN abgelegten, Wert 2 initialisiert. Die nächsten Zeilen sind schwieriger:

```
Do While Not rec.EOF
    Sheets(TABELLENBLATT).Cells(i, 1) = rec!Name
    Sheets(TABELLENBLATT).Cells(i, 2) = rec!PLZ
    rec.MoveNext
    i = i + 1
Loop
```

Wie wir sehen, handelt es sich um eine while-Schleife. Bis auf das Abbruchkriterium können wir hier allerdings bereits alles verstehen. Beim ersten Schleifendurchlauf (*i* ist dann ja 2) wird die Zelle A2 mit dem Namen des Datensatzes, auf den der Zeiger gerade deutet, besetzt. Da der Zeiger noch nicht bewegt wurde, ist dies der erste Datensatz und in die Zelle A2 wird „Meier AG“ geschrieben. Analoges gilt für B2 und *rec!PLZ* und der Inhalt von B2 nach dem ersten Schleifendurchlauf ist „47543“. Dann wird der Zeiger des Recordsets durch *rec.moveNext* einen Datensatz weiter nach unten geschoben (zeigt nun auf den zweiten Datensatz) und die Variable *i* wird um 1 erhöht. Der nächste Schleifendurchlauf besetzt also A3 mit „Schulz GmbH“ und B3 mit „44701“, schiebt den Zeiger auf den dritten Datensatz und erhöht *i* wieder um Eins.
Wie lange wird unsere Schleife aber laufen? Sinnvoll wäre natürlich, alle Datensätze des Recordsets zu durchlaufen, denn gerade dies ist unsere Aufgabenstellung. Und genau das macht (überraschenderweise) unsere Schleife. *rec.EOF* kann zwei Werte annehmen, *true* oder *false*. *rec.EOF* ist solange false, wie der Zeiger auf einen Datensatz des Recordsets zeigt. In den Abbildungen 20.4 und 20.5 ist *rec.EOF* also *false*. Auch in Abb. 20.6 ist *rec.EOF* noch false.

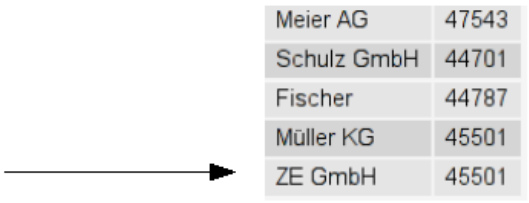


Abbildung 20.6
Unser Recordset mit dem Zeiger auf den letzten Datensatz

Der Zeiger des Recordsets zeigt jetzt auf den letzten Datensatz. Ein weiteres rec.moveNext verschiebt den Zeiger jetzt hinter den letzten Datensatz. Er zeigt damit auf keinen Datensatz mehr. Und genau dann ist *rec.EOF*[10] *true*.
Unsere Schleife läuft also so lange, bis der Zeiger des Recordsets hinter den letzen Datensatz geschoben wird und das ist genau das, was wir wollen.

20.2.2 Weitere einfache Beispiele

Im folgenden wollen wir drei weitere Beispiele zu der in Kap. 20.2.1 beschriebenen Vorgehensweise betrachten.

[10] EOF bedeutet übrigens End of File, EOR=End of Recordset wäre irgendwie passender gewesen.

Das Gewinnbeispiel

Die Kunden- und Produktnamen, Anzahlen, Einkaufspreise, Versandarten und Kategorien sollen aus einer Datenbank ausgelesen werden und in eine Excel-Tabelle wie in Abb. 6.1 übertragen werden. Das Programm startet nach clicken auf eine Schaltfläche mit dem Namen „tabelleFuellen“.
Die Datenbank heißt „Gewinnbeispiel“, der Tabellenname ist „verkauf“. Der Name des Servers ist „db.softwaregmbh.de“, Benutzer ist „hmaier“, das Passwort „ganzGeheim“. Die Struktur der zugrundeliegenden Tabelle entnehmen Sie Abb. 20.7.
Die Einkaufspreise berechnen sich aus den Datenbankinformationen wie folgt: Für die Rabattgruppe „gold“ werden 40% Rabatt auf den Katalogpreis gewährt, für „silber“ 25% Rabatt und für „bronze“ 15% Rabatt. Alle anderen Rabattgruppen erhalten keinen Rabatt auf den Katalogpreis. Die Versandarten sind in der Datenbank als Zahlen kodiert: die Versandart „CD-Versand“ mit einer 1, „Download“ mit dem Wert 0.

verkaufsNr	kundeNr	kundeName	produkt	anzahl	katalogpreis	rabatt	kategorie	versandart
1	1	Schmid	PDF Reader	1	43.33	gold	Utilities	0
2	2	Meier	Kalkulation	5	97.59	bronze	Office	1
3	2	Seran	Windows 4711	2	131.76	bronze	Betriebssysteme	0
4	3	Müller	Windows 0815	10	116	silber	Betriebssysteme	0
5	4	Schmidt	Writer	1	97.42	gold	Office	1

Abbildung 20.7
Screenshot der Tabelle für das Gewinnbeispiel

Wenn Sie die Excel-Tabelle in Abb. 6.1 betrachten, können Sie feststellen, dass wir den Einkaufspreis und die Versandart noch berechnen müssen. Schauen wir uns als Erstes die Berechnung des Einkaufspreises an:

Beispiel 20.2 *Lesen aus einer ODBC-Datenquelle: Die Funktion „bestimmeEinkaufspreis“*

```
Function bestimmeEinkaufspreis(katalogpreis As Double, rabatttext As String) As Double
    Const RABATT_BRONZE As Double = 0.15
    Const RABATT_SILBER As Double = 0.25
    Const RABATT_GOLD As Double = 0.4

    If rabatttext = "bronze" Then
        bestimmeEinkaufspreis = katalogpreis * (1 - RABATT_BRONZE)
    ElseIf rabatttext = "silber" Then
        bestimmeEinkaufspreis = katalogpreis * (1 - RABATT_SILBER)
    ElseIf rabatttext = "gold" Then
        bestimmeEinkaufspreis = katalogpreis * (1 - RABATT_GOLD)
    Else
        bestimmeEinkaufspreis = katalogpreis
    End If
End Function
```

Da die Berechnung auf dem Katalogpreis und dem Rabatt basiert, hat die Funktion genau diese beiden Parameter:

```
Function bestimmeEinkaufspreis(katalogpreis As Double, rabatttext As String) As Double
```

Die Datentypen müssen den Daten aus der Tabelle entsprechen. Der Katalogpreis ist eine Zahl mit Nachkommastellen, der Rabatt ein Text. Das Rechenergebnis ist wieder ein Währungsbetrag, also auch vom Typ Double. Die eigentliche Berechnung ist relativ einfach: es wird mit einem *if*-Befehl anhand des Rabatttextes der richtige Rabatt ausgewählt, und direkt vom Katalogpreis abgezogen.
Auch die Funktion zur Bestimmung des Versandarttextes ist nicht sonderlich schwierig:

Beispiel 20.3 *Lesen aus einer ODBC-Datenquelle: Die Funktion „bestimmeVersandarttext“*

```
Function bestimmeVersandarttext(versandart As Long) As String
    Const VERSANDART_CDVERSAND As Long = 1

    If versandart = VERSANDART_CDVERSAND Then
        bestimmeVersandarttext = "CD-Versand"
    Else
        bestimmeVersandarttext = "Download"
    End If
End Function
```

Da die Bestimmung von der als Zahl kodierten Versandart aus der Datenbanktabelle abhängt, hat die Funktion genau diesen Parameter:

```
Function bestimmeVersandarttext(versandart As Long) As String
```

Das Funktionsergebnis ist ein Text, also vom Datentyp String. Auch hier ist die eigentliche Berechnung einfach, weswegen wir sie hier nicht weiter ausführen.
Der Rest der Aufgabenstellung entspricht in weiten Teilen Beispiel 20.1. Ändern müssen wir nur die Verbindungsinformationen für den Datenbankserver, das *SQL*-Kommando und die Namen der Felder in der Ausgabe. Außerdem müssen wir natürlich die fehlenden Daten mit Hilfe der zuvor programmierten Funktionen berechnen.
Wir betrachten die Lösung:

Beispiel 20.4 *Lesen aus einer ODBC-Datenquelle: Das Gewinnbeispiel*

```
Sub tabelleFuellen_Click()
    Dim conn As ADODB.Connection
    Dim ConnectionString As String
    Dim rec As ADODB.Recordset
    Dim SQL As String
    Dim i As Long
    Const TABELLENBLATT as Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2

    Dim katalogpreis As Double
    Dim rabatt As String
    Dim versandart As Long

    ConnectionString = "Driver=MYSQL ODBC 8.0 Unicode Driver"
'  Im Pool lauffähige Lösung
    ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
    ConnectionString = ConnectionString & ";Database=wiInf_Gewinnbeispiel"
    ConnectionString = ConnectionString & ";UID=wiInf"
    ConnectionString = ConnectionString & ";PWD=wiInf"

'   Lösung nach Aufgabenstellung
'    ConnectionString = ConnectionString & ";Server=db.softwaregmbh.de"
'    ConnectionString = ConnectionString & ";Database=Gewinnbeispiel"
'    ConnectionString = ConnectionString & ";UID=hmaier"
'    ConnectionString = ConnectionString & ";PWD=ganzGeheim"

    Set conn = New ADODB.Connection
    conn.ConnectionString = ConnectionString
    conn.Open (ConnectionString)
    SQL = "Select kundeName, produkt, anzahl, katalogpreis, rabatt, kategorie, versandart  "
    SQL = SQL & "from verkauf"
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    'Ueberschriften
    Sheets(TABELLENBLATT).Cells(1, 1) = "Kunde"
    Sheets(TABELLENBLATT).Cells(1, 2) = "Produkt"
    Sheets(TABELLENBLATT).Cells(1, 3) = "Anzahl"
    Sheets(TABELLENBLATT).Cells(1, 4) = "Einkaufspreis"
    Sheets(TABELLENBLATT).Cells(1, 5) = "Softwarekategorie"
    Sheets(TABELLENBLATT).Cells(1, 6) = "Versandart"
    Sheets(TABELLENBLATT).Cells(1, 7) = "Gewinn"
```

```
    'Nun Inhalte
    i = ERSTE_ZEILE_MIT_INFORMATIONEN
    Do While Not rec.EOF
        katalogpreis = rec!katalogpreis
        rabatt = rec!rabatt
        versandart = rec!versandart

        Sheets(TABELLENBLATT).Cells(i, 1) = rec!kundeName
        Sheets(TABELLENBLATT).Cells(i, 2) = rec!produkt
        Sheets(TABELLENBLATT).Cells(i, 3) = rec!anzahl
        Sheets(TABELLENBLATT).Cells(i, 4) = bestimmeEinkaufspreis(katalogpreis, rabatt)
        Sheets(TABELLENBLATT).Cells(i, 5) = rec!kategorie
        Sheets(TABELLENBLATT).Cells(i, 6) = bestimmeVersandarttext(versandart)
        rec.MoveNext
        i = i + 1
    Loop
End Sub
```

Zunächst definieren wir die notwendigen Variablen (vgl. Beispiel 20.1) und Konstanten.

```
Dim conn As ADODB.Connection
Dim ConnectionString As String
Dim rec As ADODB.Recordset
Dim SQL As String
Dim i As Long
Const TABELLENBLATT as Long = 1
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
```

Zum Zwischenspeichern der Daten, die wir für die Berechnung der fehlenden Werte benötigen, deklarieren wir weitere Variablen:

```
Dim katalogpreis As Double
Dim rabatt As String
Dim versandart As Long
```

Danach stellen wir die Informationen zum Verbindungsaufbau auf der Variable *ConnectionString* bereit (vgl. Beispiel 20.1). Beachten Sie, dass wir hier zwei Lösungen bereitstellen:

- Eine Lösung nach Aufgabenstellung. Diese Lösung ist jedoch in den Räumen an der Hochschule nicht lauffähig. Wir haben keinen Server mit der Adresse „db.softwaregmbh.de“. Darum sind die Verbindungsinformationen für die Datenbank nach Aufgabenstellung auskommentiert.
- Eine Lösung mit Verbindungsinformationen für die Datenbank, die hier an der Hochschule lauffähig ist, so dass Sie das Programm testen und zum besseren Verständnis in Einzelschritten ausführen können.

```
ConnectionString = "Driver=MYSQL ODBC 8.0 Unicode Driver"
'  Im Pool lauffähige Lösung
ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
ConnectionString = ConnectionString & ";Database=wiInf_Gewinnbeispiel"
ConnectionString = ConnectionString & ";UID=wiInf"
ConnectionString = ConnectionString & ";PWD=wiInf"

'   Klausurlösung
'    ConnectionString = ConnectionString & ";Server=db.softwaregmbh.de"
'    ConnectionString = ConnectionString & ";Database=Gewinnbeispiel"
'    ConnectionString = ConnectionString & ";UID=hmaier"
'    ConnectionString = ConnectionString & ";PWD=ganzGeheim"
```

Die Verbindung zur Datenbank wird aufgebaut, eine ausführliche Diskussion findet sich in Beispiel 20.1.

```
Set conn = New ADODB.Connection
conn.ConnectionString = ConnectionString
conn.Open (ConnectionString)
```

Wir erstellen das *SQL*-Kommando zum Auslesen der Daten aus der Tabelle (vgl. Beispiel 20.1),

```
SQL = "Select kundeName, produkt, anzahl, katalogpreis, rabatt, kategorie, versandart "
SQL = SQL & "from verkauf"
```

schicken das *SQL*-Kommando zur Datenbank und öffnen das Resultat (vgl. Beispiel 20.1).

```
Set rec = New ADODB.Recordset
rec.Open SQL, conn
```

Wir schreiben die Spaltenüberschriften in die Excel-Tabelle, so wie sie bereits in Abb. 6.1 dargestellt wurden:

```
Sheets(TABELLENBLATT).Cells(1, 1) = "Kunde"
Sheets(TABELLENBLATT).Cells(1, 2) = "Produkt"
Sheets(TABELLENBLATT).Cells(1, 3) = "Anzahl"
Sheets(TABELLENBLATT).Cells(1, 4) = "Einkaufspreis"
Sheets(TABELLENBLATT).Cells(1, 5) = "Softwarekategorie"
Sheets(TABELLENBLATT).Cells(1, 6) = "Versandart"
Sheets(TABELLENBLATT).Cells(1, 7) = "Gewinn"
```

Die Überschrift für die Gewinnspalte wird der Vollständigkeit halber ausgegeben. Die Daten dieser Spalte können wir nicht aus der Datenbank lesen – um den Gewinn zu berechnen haben wir ja in Kap. 6.1 eine eigene Funktion geschrieben. Danach schreiben wir in einer Schleife die aus der Datenbank ausgelesenen Informationen. Eine ausführliche Diskussion der Funktionsweise der folgenden Schleife findet sich in Beispiel 20.1.

```
i = ERSTE_ZEILE_MIT_INFORMATIONEN
    Do While Not rec.EOF
        katalogpreis = rec!katalogpreis
        rabatt = rec!rabatt
        versandart = rec!versandart

        Sheets(TABELLENBLATT).Cells(i, 1) = rec!kundeName
        Sheets(TABELLENBLATT).Cells(i, 2) = rec!produkt
        Sheets(TABELLENBLATT).Cells(i, 3) = rec!anzahl
        Sheets(TABELLENBLATT).Cells(i, 4) = bestimmeEinkaufspreis(katalogpreis, rabatt)
        Sheets(TABELLENBLATT).Cells(i, 5) = rec!kategorie
        Sheets(TABELLENBLATT).Cells(i, 6) = bestimmeVersandarttext(versandart)
        rec.MoveNext
        i = i + 1
    Loop
```

Da nicht alle Daten aus der Datenbank direkt ausgegeben werden können, müssen wir noch ein wenig Arbeit in die Lösung stecken. Der Berechnung des Einkaufspreises liegt laut Aufgabenstellung der Katalogpreis und der Rabatt zugrunde. Die Festlegung des Versandarttextes basiert auf der Tabellenspalte Versandart. Daher haben wir für diese Daten zuvor Variablen deklariert, und kopieren nun die Werte aus der Datenbank in diese Variablen.

```
        katalogpreis = rec!katalogpreis
        rabatt = rec!rabatt
        versandart = rec!versandart
```

Für die Berechnung von Einkaufspreis und Versandarttext rufen wir die zuvor programmierten Funktionen auf, und schreiben die Funktionsergebnisse direkt in die Zielzellen:

```
        Sheets(TABELLENBLATT).Cells(i, 4) = bestimmeEinkaufspreis(katalogpreis, rabatt)
        ...
        Sheets(TABELLENBLATT).Cells(i, 6) = bestimmeVersandarttext(versandart)
```

Das Schlüsseldienstbeispiel

Die Kundennamen, gefahrenen Kilometer, Arbeitszeiten und ein eventueller Nachtzuschlag sollen aus einer Datenbank ausgelesen werden und in eine Excel-Tabelle wie in Abb. 6.2 übertragen werden. Das Programm startet nach clicken auf eine Schaltfläche mit dem Namen „tabelleFuellen“.

Die Datenbank heißt „Schluesseldienst“, der Tabellenname ist „AuftragLang“. Der Name des Servers ist „db.schluesseldienst.de“, Benutzer ist „hmaier“, das Passwort „strengGeheim“. Die Struktur der zugrundeliegenden Tabelle entnehmen Sie Abb. 20.8.
Die Arbeitszeiten berechnen sich aus der Differenz zwischen Arbeitsende und Arbeitsbeginn, der Nachtzuschlag wird gewährt, wenn der Arbeitsbeginn zwischen 20 Uhr und 6 Uhr war.

KundeNr	Kunde	gefahreneKilometer	BeginnStunde	BeginnMinute	EndeStunde	EndeMinute
1	Meyer	20	13	26	13	46
2	Müller	10	16	56	17	15
3	Schmidt	100	22	13	22	38
4	Schmid	15	7	48	8	23
5	Seran	10	4	23	4	34
6	Fischer	7	12	8	12	15

Abbildung 20.8
Screenshot der Tabelle für das Schlüsseldienstbeispiel

Wenn Sie die Excel-Tabelle in Abb. 6.2 betrachten, können Sie feststellen, dass wir die Arbeitszeit und den Nachtzuschlag noch berechnen müssen. Schauen wir uns als Erstes die Berechnung der Arbeitszeit an:

Beispiel 20.5 *Lesen aus einer ODBC-Datenquelle: Die Funktion „berechneArbeitszeit“*

```
Function berechneArbeitszeit(beginnStunde As Long, beginnMinute As Long, endeStunde As Long, ➥
    endeMinute As Long) As Long
    Dim arbeitszeit As Long
    Dim stunden As Long

    If (endeStunde < beginnStunde) Then
        stunden = endeStunde - beginnStunde + 24
    Else
        stunden = endeStunde - beginnStunde
    End If

    arbeitszeit = stunden * 60 + endeMinute - beginnMinute

    berechneArbeitszeit = arbeitszeit
End Function
```

Da die Berechnung auf dem Beginn und Ende der Arbeitszeit basiert, hat die Funktion genau diese vier Parameter:

```
Function berechneArbeitszeit(beginnStunde As Long, beginnMinute As Long, endeStunde As Long, ➥
    endeMinute As Long) As Long
```

Die Datentypen müssen den Daten aus der Tabelle entsprechen. Die Stunden und Minuten sind ganze Zahlen, also Long. Das Rechenergebnis ist wieder eine ganze Zahl, also auch vom Typ Long. Bei der Berechnung muss mit Hilfe eines *if*-Befehls geprüft werden, ob die Arbeit über Mitternacht hinweg ging, da dann 24 Stunden hinzugefügt werden müssen. Ansonsten sollte die Berechnung nicht so schwierig sein.
Auch die Funktion zur Bestimmung, ob ein Nachtzuschlag erhoben wird, ist nicht sonderlich schwierig:

Beispiel 20.6 *Lesen aus einer ODBC-Datenquelle: Die Funktion „bestimmeNachtzuschlag“*

```
Function bestimmeNachtzuschlag(beginnStunde As Long) As String
    If beginnStunde >= 20 Or beginnStunde < 6 Then
        bestimmeNachtzuschlag = "ja"
    Else
        bestimmeNachtzuschlag = ""
    End If
End Function
```

Da die Bestimmung nur vom Beginn der Tätigkeit, und dort auch nur von der Stunde abhängt, hat die Funktion genau diesen Parameter:

```
Function bestimmeNachtzuschlag(beginnStunde As Long) As String
```

Das Funktionsergebnis ist ein Text, also vom Datentyp String. Auch hier ist die eigentliche Berechnung einfach, weswegen wir sie hier nicht weiter ausführen.
Der Rest der Aufgabenstellung entspricht in weiten Teilen Beispiel 20.1. Ändern müssen wir nur die Verbindungsinformationen für den Datenbankserver, das *SQL*-Kommando und die Namen der Felder in der Ausgabe. Außerdem müssen wir natürlich die fehlenden Daten mit Hilfe der zuvor programmierten Funktionen berechnen.
Wir betrachten die Lösung:

Beispiel 20.7 *Lesen aus einer ODBC-Datenquelle: Das Schlüsseldienstbeispiel*

```
Sub tabelleFuellen_Click()
    Dim conn As ADODB.Connection
    Dim ConnectionString As String
    Dim rec As ADODB.Recordset
    Dim SQL As String
    Dim i As Long
    Const TABELLENBLATT as Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2

    Dim beginnStunde As Long
    Dim beginnMinute As Long
    Dim endeStunde As Long
    Dim endeMinute As Long

    ConnectionString = "Driver=MYSQL ODBC 8.0 Unicode Driver"
    '  Im Pool lauffähige Lösung
    ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
    ConnectionString = ConnectionString & ";Database=wiInf_Schluesseldienst"
    ConnectionString = ConnectionString & ";UID=wiInf"
    ConnectionString = ConnectionString & ";PWD=wiInf"

    '  Lösung nach Aufgabenstellung
    'ConnectionString = ConnectionString & ";Server=db.schluesseldienst.de"
    'ConnectionString = ConnectionString & ";Database=Schluesseldienst"
    'ConnectionString = ConnectionString & ";UID=hmaier"
    'ConnectionString = ConnectionString & ";PWD=strengGeheim"

    Set conn = New ADODB.Connection
    conn.ConnectionString = ConnectionString
    conn.Open (ConnectionString)
        SQL = "Select Kunde, gefahreneKilometer, BeginnStunde, BeginnMinute, "
        SQL = SQL & "EndeStunde, EndeMinute "
        SQL = SQL & "from AuftragLang "
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    'Ueberschriften
    Sheets(TABELLENBLATT).Cells(1, 1) = "Kunde"
    Sheets(TABELLENBLATT).Cells(1, 2) = "gefahrene Kilometer"
    Sheets(TABELLENBLATT).Cells(1, 3) = "Arbeitszeit (Min)"
    Sheets(TABELLENBLATT).Cells(1, 4) = "Nachtzuschlag"
    Sheets(TABELLENBLATT).Cells(1, 5) = "Preis"
    'Nun Inhalte
    i = ERSTE_ZEILE_MIT_INFORMATIONEN
    Do While Not rec.EOF
        beginnStunde = rec!BeginnStunde
        beginnMinute = rec!BeginnMinute
        endeStunde = rec!EndeStunde
        endeMinute = rec!EndeMinute

        Sheets(TABELLENBLATT).Cells(i, 1) = rec!Kunde
        Sheets(TABELLENBLATT).Cells(i, 2) = rec!gefahreneKilometer
```

```
        Sheets(TABELLENBLATT).Cells(i, 3) = berechneArbeitszeit(beginnStunde, beginnMinute, ➥
            endeStunde, endeMinute)
        Sheets(TABELLENBLATT).Cells(i, 4) = bestimmeNachtzuschlag(beginnStunde)
        rec.MoveNext
        i = i + 1
    Loop
End Sub
```

Zunächst definieren wir die notwendigen Variablen (vgl. Beispiel 20.1) und Konstanten.

```
    Dim conn As ADODB.Connection
    Dim ConnectionString As String
    Dim rec As ADODB.Recordset
    Dim SQL As String
    Dim i As Long
    Const TABELLENBLATT as Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
```

Zum Zwischenspeichern der Daten, die wir für die Berechnung der fehlenden Daten benötigen, deklarieren wir weitere Variablen:

```
    Dim beginnStunde As Long
    Dim beginnMinute As Long
    Dim endeStunde As Long
    Dim endeMinute As Long
```

Danach stellen wir die Informationen zum Verbindungsaufbau auf der Variable *ConnectionString* bereit (vgl. Beispiel 20.1). Beachten Sie, dass wir hier zwei Lösungen bereitstellen:

- Eine Lösung nach Aufgabenstellung. Diese Lösung ist jedoch in den Räumen an der Hochschule nicht lauffähig. Wir haben keinen Server mit der Adresse „db.schluesseldienst.de“. Darum sind die Verbindungsinformationen für die Datenbank nach Aufgabenstellung auskommentiert.
- Eine Lösung mit Verbindungsinformationen für die Datenbank, die hier an der Hochschule lauffähig ist.

TABELLENBLATT

```
ConnectionString = "Driver=MYSQL ODBC 8.0 Unicode Driver"
'  Im Pool lauffähige Lösung
ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
ConnectionString = ConnectionString & ";Database=wiInf_Schluesseldienst"
ConnectionString = ConnectionString & ";UID=wiInf"
ConnectionString = ConnectionString & ";PWD=wiInf"

'  Lösung nach Aufgabenstellung
'ConnectionString = ConnectionString & ";Server=db.schluesseldienst.de"
'ConnectionString = ConnectionString & ";Database=Schluesseldienst"
'ConnectionString = ConnectionString & ";UID=hmaier"
'ConnectionString = ConnectionString & ";PWD=strengGeheim"
```

Die Verbindung zur Datenbank wird aufgebaut, eine ausführliche Diskussion findet sich in Beispiel 20.1.

```
Set conn = New ADODB.Connection
conn.ConnectionString = ConnectionString
conn.Open (ConnectionString)
```

Wir erstellen das *SQL*-Kommando zum Auslesen der Daten aus der Tabelle (vgl. Beispiel 20.1),

```
SQL = "Select Kunde, gefahreneKilometer, BeginnStunde, BeginnMinute, "
SQL = SQL & "EndeStunde, EndeMinute "
SQL = SQL & "from AuftragLang "
```

schicken das *SQL*-Kommando zur Datenbank und öffnen das Resultat (vgl. Beispiel 20.1.

```
Set rec = New ADODB.Recordset
rec.Open SQL, conn
```

Wir schreiben die Spaltenüberschriften in die Excel-Tabelle:

```
Sheets(TABELLENBLATT).Cells(1, 1) = "Kunde"
Sheets(TABELLENBLATT).Cells(1, 2) = "gefahrene Kilometer"
Sheets(TABELLENBLATT).Cells(1, 3) = "Arbeitszeit (Min)"
Sheets(TABELLENBLATT).Cells(1, 4) = "Nachtzuschlag"
Sheets(TABELLENBLATT).Cells(1, 5) = "Preis"
```

Die Überschrift für die Preisspalte wird der Vollständigkeit halber ausgegeben. Die Daten dieser Spalte können wir nicht aus der Datenbank lesen – um den Preis zu berechnen haben wir ja in Kap. 6.2 eine eigene Funktion geschrieben.
Danach schreiben wir in einer Schleife die aus der Datenbank ausgelesenen Informationen. Eine ausführliche Diskussion der Funktionsweise der folgenden Schleife findet sich in Beispiel 20.1.

```
    i = ERSTE_ZEILE_MIT_INFORMATIONEN
    Do While Not rec.EOF
        beginnStunde = rec!BeginnStunde
        beginnMinute = rec!BeginnMinute
        endeStunde = rec!EndeStunde
        endeMinute = rec!EndeMinute

        Sheets(TABELLENBLATT).Cells(i, 1) = rec!Kunde
        Sheets(TABELLENBLATT).Cells(i, 2) = rec!gefahreneKilometer
        Sheets(TABELLENBLATT).Cells(i, 3) = berechneArbeitszeit(beginnStunde, beginnMinute, ➥
            endeStunde, endeMinute)
        Sheets(TABELLENBLATT).Cells(i, 4) = bestimmeNachtzuschlag(beginnStunde)
        rec.MoveNext
        i = i + 1
    Loop
```

Da nicht alle Daten aus der Datenbank direkt ausgegeben werden können, müssen wir noch ein wenig Arbeit in die Lösung stecken. Der Berechnung der Arbeitszeit liegt laut Aufgabenstellung der Beginn und Ende der Tätigkeit zugrunde. Die Festlegung des Nachtzuschlags basiert auf der Tabellenspalte „BeginnStunde“. Daher haben wir für diese Daten zuvor Variablen deklariert, und kopieren nun die Werte aus der Datenbank in diese Variablen.

```
        beginnStunde = rec!BeginnStunde
        beginnMinute = rec!BeginnMinute
        endeStunde = rec!EndeStunde
        endeMinute = rec!EndeMinute
```

Für die Berechnung von Arbeitszeit und Nachtzuschlag rufen wir die zuvor programmierten Funktionen auf, und schreiben die Funktionsergebnisse direkt in die Zielzellen:

```
        Sheets(TABELLENBLATT).Cells(i, 3) = berechneArbeitszeit(beginnStunde, beginnMinute, ➥
            endeStunde, endeMinute)
        Sheets(TABELLENBLATT).Cells(i, 4) = bestimmeNachtzuschlag(beginnStunde)
```

Das Zuweisungsbeispiel

Studiengang, Abschluss, Absolventen gesamt und in Regelstudienzeit sowie die Kapazität sollen aus einer Datenbank ausgelesen werden und in eine Excel-Tabelle wie in Abb. 6.3 übertragen werden. Das Programm startet nach clicken auf eine Schaltfläche mit dem Namen „tabelleFuellen“.
Der Datenbankserver hat den Namen db.zuweisung.de, der User heißt zuweisung, die Tabelle heißt Studiengang, die Datenbank wieder zuweisung und das Passwort jkummk. Die Struktur der zugrundeliegenden Tabelle entnehmen Sie Abb. 20.9.
Der Abschluss ist in der Datenbank als Zahl codiert, eine 1 bedeutet „Bachelor“, eine 2 „Master“. Andere Zahlen sind als „unbekannter Abschluss“ zu kennzeichnen. Die Kapazität berechnet sich aus den Lehreinheiten und den Finanzmitteln. Je Studienplatz und Lehreinheit werden Finanzmittel in Höhe von 850 Euro benötigt.

StudiengangNr	Studiengang	Abschlussart	Absolventen	Regelstudienzeit	Finanzmittel	Lehreinheiten
1	Bachelor of Arts	1	85	50	4590000	54
2	Wirtschaftsingenieur	1	9	1	650250	51
3	MAAT	2	10	10	459000	27

Abbildung 20.9
Screenshot der Tabelle für das Zuweisungsbeispiel

Wenn Sie die Excel-Tabelle in Abb. 6.3 betrachten, können Sie feststellen, dass wir die Kapazität und den Abschluss noch berechnen müssen. Schauen wir uns als Erstes die Berechnung der Kapazität an:

Beispiel 20.8 *Lesen aus einer ODBC-Datenquelle: Die Funktion „berechneKapazitaet“*

```
Function berechneKapazitaet(finanzmittel As Double, lehreinheiten As Long) As Long
    Dim kapazitaet As Long
    Const MITTEL_PRO_LEHREINHEIT As Double = 850

    kapazitaet = Int(finanzmittel / (lehreinheiten * MITTEL_PRO_LEHREINHEIT))

    berechneKapazitaet = kapazitaet
End Function
```

Da die Berechnung auf den Finanzmitteln und den Lehreinheiten basiert, hat die Funktion genau diese beiden Parameter:

```
Function berechneKapazitaet(finanzmittel As Double, lehreinheiten As Long) As Long
```

Die Datentypen müssen den Daten aus der Tabelle entsprechen. Finanzmittel sind ein Währungsbetrag, also Double, die Lehreinheiten eine ganze Zahl, und somit Long. Das Rechenergebnis ist wieder eine ganze Zahl, also auch vom Typ Long. Die eigentliche Berechnung ist relativ einfach: die Finanzmittel werden durch die Kosten der Lehreinheiten geteilt. Damit das Ergebnis auf jeden Fall ganzzahlig ist, wird mit dem *Int*-Befehl das Ergebnis gerundet.
Auch die Funktion zur Bestimmung des Abschlusses ist nicht sonderlich schwierig:

Beispiel 20.9 *Lesen aus einer ODBC-Datenquelle: Die Funktion „bestimmeAbschluss“*

```
Function bestimmeAbschluss(abschlussart As Long) As String
    If abschlussart = 1 Then
        bestimmeAbschluss = "Bachelor"
    ElseIf abschlussart = 2 Then
        bestimmeAbschluss = "Master"
    Else
        bestimmeAbschluss = "unbekannter Abschluss"
    End If
End Function
```

Da die Bestimmung von der als Zahl kodierten Abschlussart aus der Datenbanktabelle abhängt, hat die Funktion genau diesen Parameter:

```
Function bestimmeAbschluss(abschlussart As Long) As String
```

Das Funktionsergebnis ist ein Text, also vom Datentyp String. Auch hier ist die eigentliche Berechnung einfach, weswegen wir sie hier nicht weiter ausführen.
Der Rest der Aufgabenstellung entspricht in weiten Teilen Beispiel 20.1. Ändern müssen wir nur die Verbindungsinformationen für den Datenbankserver, das *SQL*-Kommando und die Namen der Fehler in der Ausgabe. Außerdem müssen wir natürlich die fehlenden Daten mit Hilfe der zuvor programmierten Funktionen berechnen.
Wir betrachten die Lösung:

Beispiel 20.10 *Lesen aus einer ODBC-Datenquelle: Das Zuweisungsbeispiel*

```
Sub tabelleFuellen_Click()
    Dim conn As ADODB.Connection
    Dim ConnectionString As String
    Dim rec As ADODB.Recordset
    Dim SQL As String
    Dim i As Long
    Const TABELLENBLATT as Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2

    Dim lehreinheiten As Long
    Dim finanzmittel As Double
    Dim abschlussart As Long

    ConnectionString = "Driver=MYSQL ODBC 8.0 Unicode Driver"
    '  Im Pool lauffähige Lösung
    ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
    ConnectionString = ConnectionString & ";Database=wiInf_Zuweisung"
    ConnectionString = ConnectionString & ";UID=wiInf"
    ConnectionString = ConnectionString & ";PWD=wiInf"

    '  Lösung nach Aufgabenstellung
    'ConnectionString = ConnectionString & ";Server=db.zuweisung.de"
    'ConnectionString = ConnectionString & ";Database=Zuweisung"
    'ConnectionString = ConnectionString & ";UID=zuweisung"
    'ConnectionString = ConnectionString & ";PWD=jkummk"

    Set conn = New ADODB.Connection
    conn.ConnectionString = ConnectionString
    conn.Open (ConnectionString)
    SQL = "Select Studiengang, Abschlussart, Absolventen, Regelstudienzeit, Finanzmittel, ➥
        Lehreinheiten "
    SQL = SQL & " from Studiengang "
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn

    'Ueberschriften
    Sheets(TABELLENBLATT).Cells(1, 1) = "Studiengang"
    Sheets(TABELLENBLATT).Cells(1, 2) = "Abschluss"
    Sheets(TABELLENBLATT).Cells(1, 3) = "Absolventen"
    Sheets(TABELLENBLATT).Cells(1, 4) = "davon in Regelstudienzeit"
    Sheets(TABELLENBLATT).Cells(1, 5) = "Kapazität"
    Sheets(TABELLENBLATT).Cells(1, 6) = "Zuweisung"

    'Nun Inhalte
    i = ERSTE_ZEILE_MIT_INFORMATIONEN
    Do While Not rec.EOF
        lehreinheiten = rec!lehreinheiten
        finanzmittel = rec!finanzmittel
        abschlussart = rec!abschlussart

        Sheets(TABELLENBLATT).Cells(i, 1) = rec!studiengang
        Sheets(TABELLENBLATT).Cells(i, 2) = bestimmeAbschluss(abschlussart)
        Sheets(TABELLENBLATT).Cells(i, 3) = rec!absolventen
        Sheets(TABELLENBLATT).Cells(i, 4) = rec!regelstudienzeit
        Sheets(TABELLENBLATT).Cells(i, 5) = berechneKapazitaet(finanzmittel, lehreinheiten)
        rec.MoveNext
        i = i + 1
    Loop

End Sub
```

Zunächst definieren wir die notwendigen Variablen (vgl. Beispiel 20.1) und Konstanten.

```
Dim conn As ADODB.Connection
Dim ConnectionString As String
Dim rec As ADODB.Recordset
Dim SQL As String
Dim i As Long
```

```
Const TABELLENBLATT as Long = 1
Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2
```

Zum Zwischenspeichern der Daten, die wir für die Berechnung der fehlenden Werte benötigen, deklarieren wir weitere Variablen:

```
Dim katalogpreis As Double
Dim rabatt As String
Dim versandart As Long
```

Danach stellen wir die Informationen zum Verbindungsaufbau auf der Variable *ConnectionString* bereit (vgl. Beispiel 20.1). Beachten Sie, dass wir hier zwei Lösungen bereitstellen:

- Eine Lösung nach Aufgabenstellung. Diese Lösung ist jedoch in den Räumen an der Hochschule nicht lauffähig. Wir haben keinen Server mit der Adresse „db.zuweisung.de“. Darum sind die Verbindungsinformationen für die Datenbank nach Aufgabenstellung auskommentiert.
- Eine Lösung mit Verbindungsinformationen für die Datenbank, die hier an der Hochschule lauffähig ist.

```
ConnectionString = "Driver=MYSQL ODBC 8.0 Unicode Driver"
'  Im Pool lauffähige Lösung
ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
ConnectionString = ConnectionString & ";Database=wiInf_Zuweisung"
ConnectionString = ConnectionString & ";UID=wiInf"
ConnectionString = ConnectionString & ";PWD=wiInf"

'  Lösung nach Aufgabenstellung
'ConnectionString = ConnectionString & ";Server=db.zuweisung.de"
'ConnectionString = ConnectionString & ";Database=zuweisung"
'ConnectionString = ConnectionString & ";UID=zuweisung"
'ConnectionString = ConnectionString & ";PWD=jkummk"
```

Die Verbindung zur Datenbank wird aufgebaut, eine ausführliche Diskussion findet sich in Beispiel 20.1.

```
Set conn = New ADODB.Connection
conn.ConnectionString = ConnectionString
conn.Open (ConnectionString)
```

Wir erstellen das *SQL*-Kommando zum Auslesen der Daten aus der Tabelle (vgl. Beispiel 20.1),

```
SQL = "Select Studiengang, Abschlussart, Absolventen, Regelstudienzeit, Finanzmittel, ➥
    Lehreinheiten "
SQL = SQL & " from zuweisung "
```

schicken das *SQL*-Kommando zur Datenbank und öffnen das Resultat (vgl. Beispiel 20.1.

```
Set rec = New ADODB.Recordset
rec.Open SQL, conn
```

Wir schreiben die Spaltenüberschriften in die Excel-Tabelle:

```
Sheets(TABELLENBLATT).Cells(1, 1) = "Studiengang"
Sheets(TABELLENBLATT).Cells(1, 2) = "Abschluss"
Sheets(TABELLENBLATT).Cells(1, 3) = "Absolventen"
Sheets(TABELLENBLATT).Cells(1, 4) = "davon in Regelstudienzeit"
Sheets(TABELLENBLATT).Cells(1, 6) = "Zuweisung"
```

Die Überschrift für die Zuweisungsspalte wird der Vollständigkeit halber ausgegeben. Die Daten dieser Spalte können wir nicht aus der Datenbank lesen – um die Zuweisung zu berechnen haben wir ja in Kap. 6.3 eine eigene Funktion geschrieben.
Danach schreiben wir in einer Schleife die aus der Datenbank ausgelesenen Informationen. Eine ausführliche Diskussion der Funktionsweise der folgenden Schleife findet sich in Beispiel 20.1.

```
i = ERSTE_ZEILE_MIT_INFORMATIONEN
Do While Not rec.EOF
    lehreinheiten = rec!lehreinheiten
    finanzmittel = rec!finanzmittel
    abschlussart = rec!abschlussart

    Sheets(TABELLENBLATT).Cells(i, 1) = rec!studiengang
    Sheets(TABELLENBLATT).Cells(i, 2) = bestimmeAbschluss(abschlussart)
    Sheets(TABELLENBLATT).Cells(i, 3) = rec!absolventen
    Sheets(TABELLENBLATT).Cells(i, 4) = rec!regelstudienzeit
    Sheets(TABELLENBLATT).Cells(i, 5) = berechneKapazitaet(finanzmittel, lehreinheiten)
    rec.MoveNext
    i = i + 1
Loop
```

Da nicht alle Daten aus der Datenbank direkt ausgegeben werden können, müssen wir noch ein wenig Arbeit in die Lösung stecken. Der Berechnung der Kapazität liegen laut Aufgabenstellung die Finanzmittel und die Lehreinheiten zugrunde. Die Festlegung des Abschlusses basiert auf der Tabellenspalte „Abschlussart“. Daher haben wir für diese Daten zuvor Variablen deklariert, und kopieren nun die Werte aus der Datenbank in diese Variablen.

```
    lehreinheiten = rec!lehreinheiten
    finanzmittel = rec!finanzmittel
    abschlussart = rec!abschlussart
```

Für die Berechnung von Abschluss und Kapazität rufen wir die zuvor programmierten Funktionen auf, und schreiben die Funktionsergebnisse direkt in die Zielzellen:

```
    Sheets(TABELLENBLATT).Cells(i, 2) = bestimmeAbschluss(abschlussart)
    ...
    Sheets(TABELLENBLATT).Cells(i, 5) = berechneKapazitaet(finanzmittel, lehreinheiten)
```

20.2.3 Verbessertes praxisnahes Beispiel: Erweiterung Beispiel 20.1

Beispiel 20.1 hat noch Nachteile:

- Benutzername und Passwort sind im VBA-Quellcode hinterlegt. Jeder, der Zugriff auf die Excel-Datei hat, hat damit auch eine Zugriffsberechtigung auf die zugrundeliegende Datenbank. Besser und sicherer wäre, wenn die Benutzer Benutzernamen und Passwort eingeben müßten. Das können wir aber arrangieren: Wir erzeugen ein Formular mit Eingabemöglichkeiten für Benutzername und Passwort und melden den Benutzer damit an.
- Das Programm lässt sich zur Zeit nur aus dem VBA-Editor aufrufen. Das ist natürlich auch nicht ganz so günstig. Da können wir aber auch schnell Abhilfe schaffen. Wir erzeugen eine Schaltfläche in der Excel-Tabelle und starten die ganze Geschichte, wenn der Benutzer auf diese Schaltfläche clickt.
- Name und PLZ aus einer Datenbank auszulesen ist für Excel sicher auch nicht ganz so adäquat, damit kann man schließlich nicht rechnen. Wir überlegen uns hier auch etwas praxisnäheres: Wir holen uns die Umsätze aller Aufträge des laufenden Monats, geordnet nach den Kundennamen. Danach können wir dieses mit den üblichen Excel-Funktionalitäten auswerten.

Zunächst das Benutzerinterface: Zu Beginn sieht die Datei, wie in Abb. 20.10 dargestellt, aus. Clicken auf die Schaltfläche „zur Datenbank“ öffnet das in Abb. 20.11 dargestellte Formular. Abb. 20.12 zeigt das Ergebnis nach dem Ausfüllen und Abschicken des Formulars. Als nächstes überlegen wir uns das für die Selektion der Daten benötigte SQL-Kommando. Wir zeigen noch einmal das ERM (Abb. 20.13) und die Tabellenstruktur der Auftragsdatenbank aus dem Datenbank-Script.

Name	Primärschlüssel	Weitere Felder
Kunde	KundeNr	Name, PLZ, Stadt
Auftrag	AuftragNr	AuftragDatum, Lieferdatum, *KundeNr*
Produkt	ProduktNr	Name, Preis
auftragProdukt	*AuftragNr*, *ProduktNr*	Anzahl

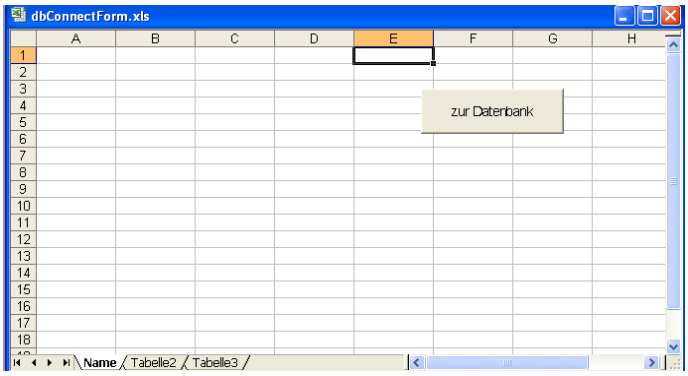


Abbildung 20.10
Startbildschirm der Anwendung

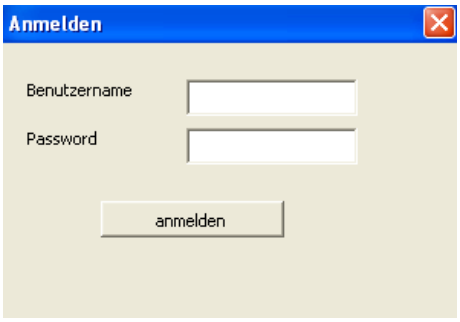


Abbildung 20.11
Das Anmeldeformular

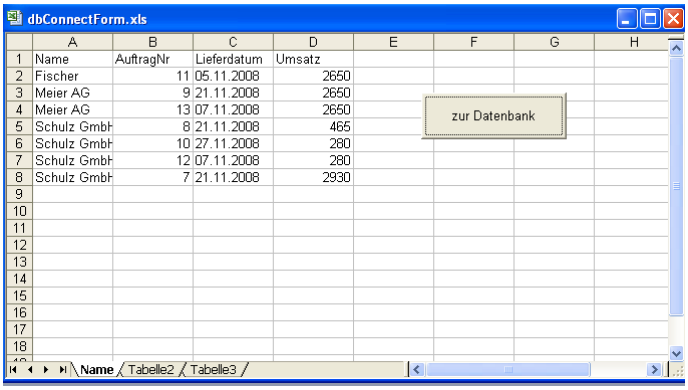


Abbildung 20.12
Die Tabelle nach Ausführung des VBA-Programms

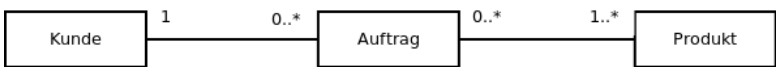


Abbildung 20.13
Das ERM der Auftragsdatenbank aus dem Datenbankscript

Die Tabellen zu 20.13

Das SQL-Kommando, um die benötigten Daten für den November 2008 aus den Tabellen zu selektieren, ist nun[11]:

Beispiel 20.11 *Das SQL-Kommando für die Umsätze*

```
SELECT
    Kunde.Name,
    Auftrag.AuftragNr,
    Auftrag.Lieferdatum,
    sum(auftragProdukt.Anzahl * Produkt.Preis) AS Umsatz
FROM
    Kunde,
    Auftrag,
    auftragProdukt,
    Produkt
WHERE
    Auftrag.Lieferdatum LIKE '2008-11-%'
AND
    Auftrag.KundeNr = Kunde.KundeNr
AND
    Auftrag.AuftragNr = auftragProdukt.AuftragNr
AND
    auftragProdukt.ProduktNr = Produkt.ProduktNr
GROUP BY
    Auftrag.AuftragNr
ORDER BY
    Kunde.Name
```

Das müssten Sie mit Ihren SQL-Kenntnissen verstehen ☺, ansonsten verweisen wir auf das Datenbank-Script.
Nun können wir mit der Entwicklung beginnen. Als erstes erzeugen wir mit dem Formular-Editor das in Abb. 20.11 dargestellte Formular. Wir geben dem Formular den Name *anmeldeForm*. Im Formular selber vergeben wir folgende Namen:

- *benutzernameInput* für das Textinput-Feld zur Eingabe des Benutzernamens.
- *passwortInput* für das Textinput-Feld zur Eingabe des Passworts.
- *anmeldenButton* für die Schaltfläche.

Dann malen wir die Schaltfläche zum Aufblenden des Anmeldeformulars in die Tabelle. Wir geben der Schaltfläche die Caption „zur Datenbank“ und den Namen „databaseConnectButton“. Der Code der Ereignisprozedur ist dann ziemlich einfach:

Beispiel 20.12 *Die Ereignisprozedur zum Aufblenden des Anmeldeformulars*

```
'Dateiname: dbConnectForm.xls
Sub databaseConnectButton_Click()
    anmeldeForm.Show
End Sub
```

Als nächstes schreiben wir die Ereignisprozedur für die Schaltfläche auf dem Formular:

Beispiel 20.13 *Die Ereignisprozedur des Anmeldebuttons des Anmeldeformulars*

```
'Dateiname: dbConnectForm.xls
Sub anmeldenButton_Click()
    Dim benutzername As String
    Dim passwort As String
    benutzername = benutzernameInput.Text
    passwort = passwortInput.Text
    Call umsatzAuslesen(benutzername, passwort)
    Unload Me
End Sub
```

[11] Beachten Sie die englische Schreibweise des Datums.

Hier werden zunächst die Eingaben der Benutzer aus dem Formular gelesen, dann wird eine Prozedur mit den Benutzereingaben als Parameter aufgerufen. Zum Abschluss wird das Formular geschlossen. Nun bleibt nur noch die Prozedur *umsatzAuslesen*:

Beispiel 20.14 *Die Prozedur Umsatzauslesen*

```
'Dateiname: dbConnectForm.xls
Sub umsatzAuslesen(benutzername As String, passwort As String)
    Dim conn As ADODB.Connection
    Dim ConnectionString As String
    Dim rec As ADODB.Recordset
    Dim SQL As String
    Dim monat As Long
    Dim jahr As Long
    Dim monatJahrString As String
    Dim i As Long
    Const TABELLENBLATT As Long = 1
    Const ERSTE_ZEILE_MIT_INFORMATIONEN As Long = 2

    ConnectionString = "Driver=MYSQL ODBC 8.0 Unicode Driver"
    ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
    ConnectionString = ConnectionString & ";Database=datenbankVorlesung"
    ConnectionString = ConnectionString & ";UID=" & benutzername
    ConnectionString = ConnectionString & ";PWD=" & passwort
    Set conn = New ADODB.Connection
    conn.ConnectionString = ConnectionString
    conn.Open
    monat = Month(Date)
    jahr = Year(Date)
    If monat < 10 Then
        monatJahrString = jahr & "-0" & monat & "-\%"
    Else
        monatJahrString = jahr & "-" & monat & "-\%"
    End If
    SQL = "Select Kunde.Name, "
    SQL = SQL & "Auftrag.AuftragNr, "
    SQL = SQL & "Auftrag.Lieferdatum, "
    SQL = SQL & "sum(auftragProdukt.Anzahl * Produkt.Preis) AS Umsatz "
    SQL = SQL & "FROM Kunde, Auftrag, auftragProdukt, Produkt "
    SQL = SQL & "WHERE Auftrag.Lieferdatum LIKE '" & monatJahrString & "' "
    SQL = SQL & "AND Auftrag.KundeNr = Kunde.KundeNr "
    SQL = SQL & "AND Auftrag.AuftragNr = auftragProdukt.AuftragNr "
    SQL = SQL & "AND auftragProdukt.ProduktNr = Produkt.ProduktNr "
    SQL = SQL & "GROUP BY Auftrag.AuftragNr "
    SQL = SQL & "ORDER BY Kunde.Name"
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    'Ueberschriften
    Sheets(TABELLENBLATT).Cells(1, 1) = "Name"
    Sheets(TABELLENBLATT).Cells(1, 2) = "AuftragNr"
    Sheets(TABELLENBLATT).Cells(1, 3) = "Lieferdatum"
    Sheets(TABELLENBLATT).Cells(1, 4) = "Umsatz"
    'Nun Inhalte
    i = ERSTE_ZEILE_MIT_INFORMATIONEN
    Do While Not rec.EOF
        Sheets(TABELLENBLATT).Cells(i, 1) = rec!Name
        Sheets(TABELLENBLATT).Cells(i, 2) = rec!AuftragNr
        Sheets(TABELLENBLATT).Cells(i, 3) = rec!Lieferdatum
        Sheets(TABELLENBLATT).Cells(i, 4) = rec!Umsatz
        rec.MoveNext
        i = i + 1
    Loop
End Sub
```

Diese Prozedur ähnelt Beispiel 20.1 sehr. Im Unterschied zu Beispiel 20.1 benutzt Beispiel 20.14 keine festen Werte für Benutzename und Passwort, sondern die übergebenen Variablen *benutzername* und *passwort* und damit die Eingaben der Benutzer:

```
ConnectionString = ConnectionString & ";UID=" & benutzername
ConnectionString = ConnectionString & ";PWD=" & passwort
```

Die Prozedur ermittelt außerdem das aktuelle Jahr und den aktuellen Monat. Diese Informationen werden mit nachgestelltem Prozentzeichen auf der Variablen *monatJahrString* abgespeichert. Falls der aktuelle Monat einstellig ist (Januar-September), müssen wir darauf achten, dass wir den Monat mit einer 0 auffüllen.

```
monat = Month(Date)
jahr = Year(Date)
If monat < 10 Then
    monatJahrString = jahr & "-0" & monat & "-\%"
Else
    monatJahrString = jahr & "-" & monat & "-\%"
End If
```

In den nächsten Zeilen wird das SQL-Kommando aus Beispiel 20.11 auf die Variable *SQL* geschrieben. Anstelle '2008-11-%' für November 2008 wird der auf der Variablen *monatJahrString* abgespeicherte aktuelle Monat benutzt.

```
SQL = SQL & "WHERE Auftrag.Lieferdatum LIKE '" & monatJahrString & "' "
```

Danach wird das SQL-Kommando an die Datenbank geschickt und das Ergebnis in die Excel-Tabelle geschrieben. Der Code hierfür entspricht Beispiel 20.1, so dass wir auf eine erneute Diskussion verzichten.

Stichwortverzeichnis