VBA

Automatisierungstechnik für Office-Pakete

Bernd Blümel, Volker Klingspor, Christian Metzger

Version: 20. Oktober 2011

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 evtl. langweilige Routineaufgaben, wie z.B. irgendeine Auswertung von Zahlen, in VBA zu programmieren 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 ☺
Erfahrungsgemäß ist das Verhältnis von BetriebswirtschaftsstudentInnen und (Wirtschafts-) Informatik leicht gestört. Dies beruht auf der irrigen Annahme der Studierenden das Informatik etwas sehr kompliziertes sei (so ähnlich wie Mathematik). Dem ist, zumindest im Bereich Wirtschaftsinformatik, aber nicht so. 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

[1] Visual Basic for Applications

vereinbarten Ziele erreichen oder nicht. Solche Problematiken lassen sich mit einer Tabellenkalkulation und VBA

	A	B	C	D	E	F	G	H
1	Geplanter Umsatz	1.000.000,00 €						
2	Prognose für akt. Jahr	1.548.040,00 €						
3	Differenz	548.040,00 €	Die Hochrechnung erfolgte nicht linear zum Monat März					
4								
5	Datum	Verkaufsbetrag	Provision					
6	03.01.2008	2.000,00 €	400,00 €					
7	06.01.2008	20.000,00 €	4.000,00 €					
8	01.02.2008	345.000,00 €	69.000,00 €					
9	09.02.2008	20.000,00 €	4.000,00 €					
10	01.03.2008	10,00 €	2,00 €					
11								
12								
13								
14								

Gesamt- und Durchschnittsprovision

Prognose berechnen

Plausibilitätsprüfung

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 mein Programm uns alles fertig.

	A	B	C	D	E	F	G	H	I	J
1	Punkte	100								
2	benötigte Prozente	50								
3	Name	Aufgabe1	Aufgabe 2	Aufgabe 3	Aufgabe 4	Summe	Note	Bewertungspunkte		
4	Optimal	20	30	10	21	81	2,3	17		
5	Meyer	15	15	10	2	42	5	9		
6	Müller	10	5	5	5	25	5	5		
7	Meyer	15	15	10	2	42	5	9		
8	Müller	10	5	5	40	60	3,7	12		
9	Meyer	15	15	10	40	80	2,3	16		
10	Müller	10	5	5	5	25	5	5		
11										
12										
13										
14										
15										
16										

Notenverteilung

Durchschnittsnote

Punkte darstellen

Abbildung 1.2
Programm zur Berechnung von Klausurnoten

In diesem Script wird VBA anhand von Excel und OpenOffice besprochen. Die Grundsprache VBA selber ist für Excel und OpenOffice gleich. In OpenOffice heißt die Makrosprache übrigens nicht VBA, sondern Starbasic. Im diesem Script heißt es immer VBA. Wenn sich die Realisierungen in Excel und OpenOffice unterscheiden, gibt es jeweils ein Kapitel über Excel und eins über OpenOffice. Dabei ist das Script so aufgebaut, dass nicht beide Kapitel gelesen werden müssen. Wenn Sie kein Interesse an der OpenOffice- oder an der Excel-Lösung haben, können Sie das entsprechende Kapitel ohne Informationsverlust weglassen. Dadurch entstehen natürlich Redundanzen, wenn Sie beide Lösungen lesen.

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
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 das 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 2 Zahlen addieren soll würde man dann 2 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" in einer Zelle eines Tabellenblatts ausgibt.

2.1 Ausgabe eines Strings

Beispiel 2.2 *Ausgabe eines Strings mittels einer Funktion*

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

Unser erstes Programm ist, wie alle unsere ersten Programme, eine Funktion[1]. 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,

[1] Der Funktionsbegriff wird später noch ausführlicher erklärt!

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, Zahlen 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

```
=helloWord()
```

in diese Zelle ein (aber ohne die 1). 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

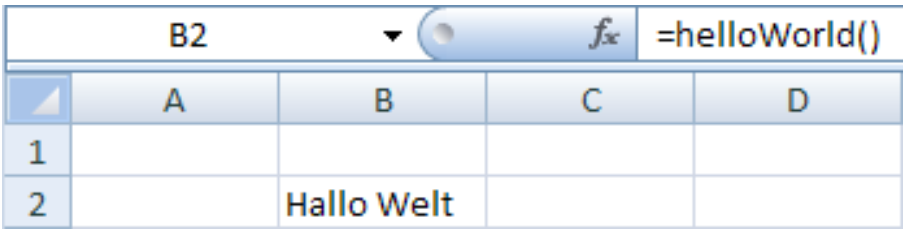


Abbildung 2.1
Einfügen der Funktion in die Tabellenkalkulation

Welt" in die Zelle 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.

2.2 Addition von Zahlen

Beispiel 2.3 *Addition mit festen Werten*

```
'Dateiname: beispiel.xls
' Programm addiert zwei Zahlen
Function addition() As Integer
        ' Variablen deklarieren
        Dim ersterSummand As Integer
        Dim zweiterSummand As Integer
        Dim summe As Integer
        '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 Integer* nach dem Programmnamen. Diese besagt aber nichts anderes, als dass diese Funktion eine ganze Zahl (Integer) in eine Zelle der Tabellenkalkulation schreiben wird. Danach jedoch sehen wir neue Dinge. Wir haben 3 relativ gleich aussehende Zeilen:

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

Diese Zeilen beginnen mit dem Schlüsselwort *dim*. *dim* leitet eine Variablendeklaration ein. 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. 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 die reservierten Worte *As Integer*. Hierdurch wird der Datentyp der Variablen festgelegt. Der Datentyp [2] 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 Integer* 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 Integer* festgelegt. Das bedeutet die Funktion gibt eine Ganzzahl zurück.
Danach folgen die :

```
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

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

```
addition = summe
```

wird der Rückgabewertder 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 das kann, nämlich zwei beliebige Zahlen addieren. Unser Benutzerinterface soll wie in Abb. 2.2 dargestellt aussehen. In die Zelle B1 wollen wir die erste zu addierende

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

Abbildung 2.2
Benutzerinterface zum Addieren zweier Zahlen

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 in VBA ist . 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 der Funktion. Über die runden Klammern kann eine Funktion Werte aus einem Arbeitsblatt[3] 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 (entweder aus einem Tabellenblatt oder einer InputBox), 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 (wieder ohne die 1):

```
= 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. Auch dieses Beispiel ist für eine Tabellenkalkulation nichts umwerfend Neues, addieren können

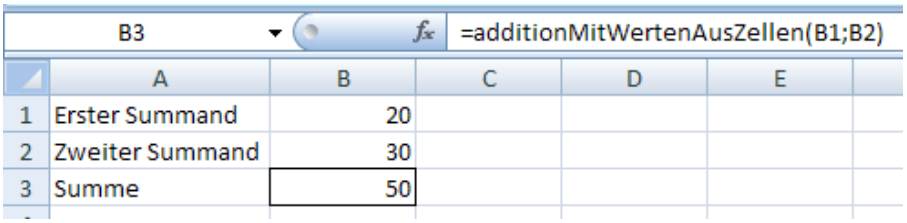


Abbildung 2.3
Einfügen einer Funktion mit Übergabewerten in die Tabellenkalkulation

sowieso schon. Sie haben aber gelernt, wie man Werte aus dem Arbeitsblatt einer Tabellenkalkulation in eigene, selbstgeschriebene Funktionen übernehmen kann. 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 z.B. 4 eingegeben, soll das Programm 1+2+3+4 rechnen. Unser Benutzerinterface soll wie in Abb. 2.4 dargestellt aussehen.

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

2.4 For-Schleife

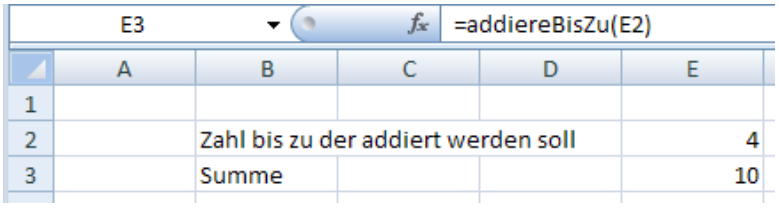


Abbildung 2.4
Addition von natürlichen Zahlen

Betrachten wir zunächst den Programmcode der Funktion:

Beispiel 2.6 *Addition bis zu einer eingegebenen Zahl*

```
'Dateiname: beispiel.xls
function addiereBisZu(endZahl as Integer) as Integer
        dim summe As Integer
        dim i As Integer
        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[4].

```
summe = 0
```

Dann folgt eine Schleife. Schleifen werden in Kap. 7 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:

[4] initialisieren bedeutet: einen Startwert zuweisen

```
summe = summe + i
```

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

```
next i
```

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

```
next i
```

VBA, zu der Anweisung

```
for i = 1 to 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.
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.

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

- 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[5], 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 Integer) as Integer
        dim summe As Integer
        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:

[5] was aber eher selten ist

- 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.

Kapitel 3

Einfügen benutzerdefinierter Funktionen

Funktionen, die wir selbst schreiben und dann in Excel einfügen, heißen benutzerdefinierte Funktionen. Unter diesem Schlagwort sind sie auch in der Hilfe von Excel bzw. OpenOffice zu finden (vgl. Abb. 3.1). 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 liegen: In OpenOffice erreichen Sie über die Menüfolge Extras⇨Makros⇨Makros verwalten⇨OpenOffice.org Basic die Entwicklungsumgebung.(vgl. Abb. 3.2)

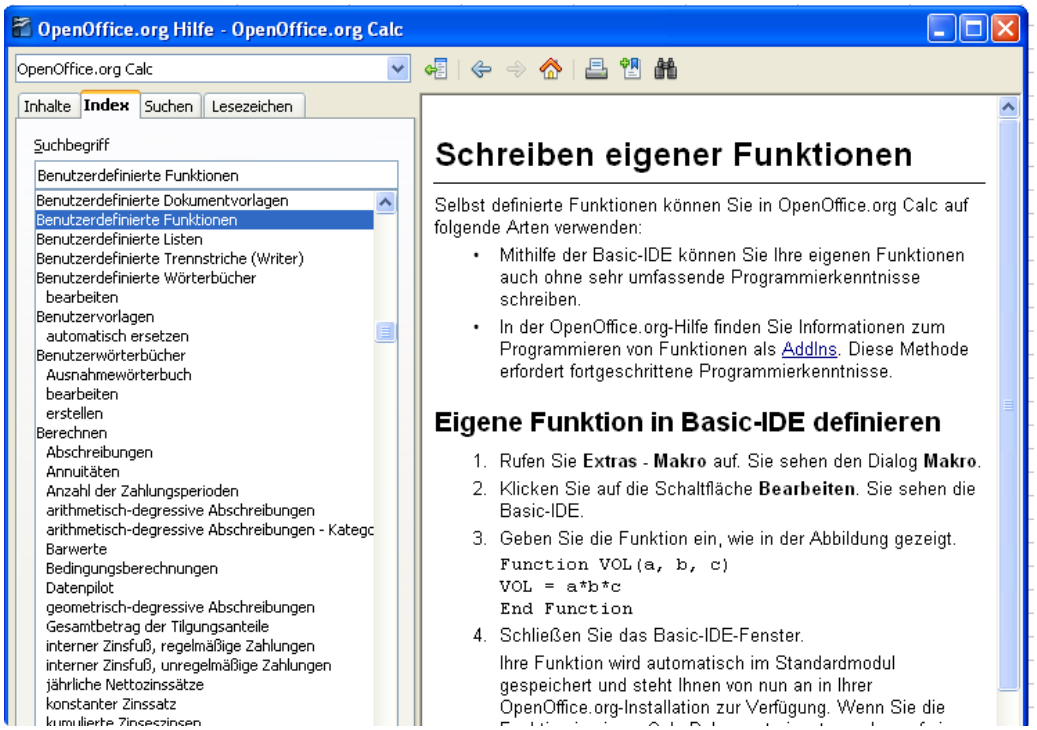


Abbildung 3.1
Open Office Hilfe

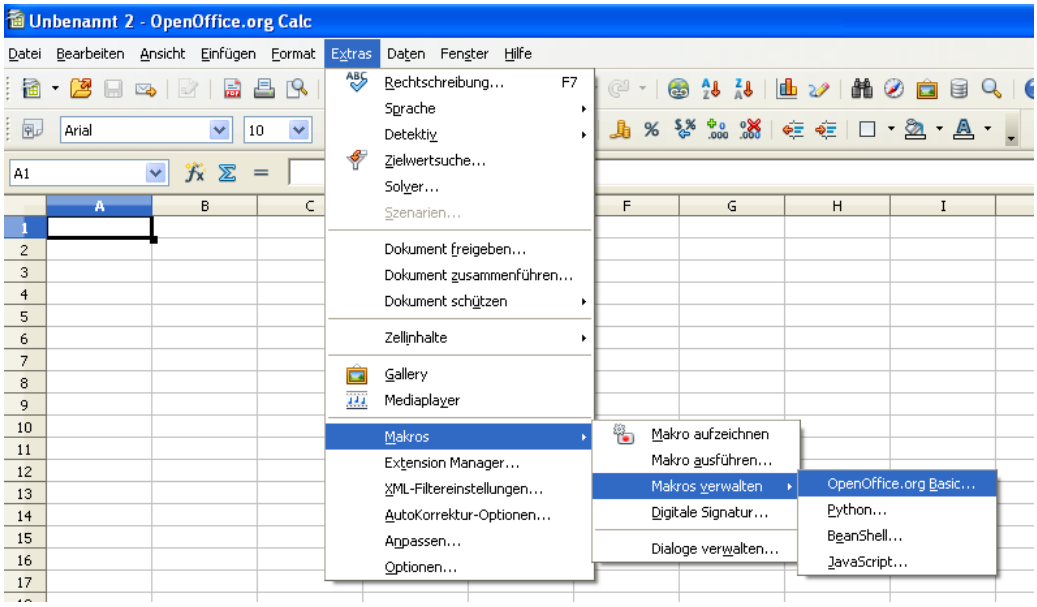


Abbildung 3.2
Aufruf der Makroverwaltung in OpenOffice

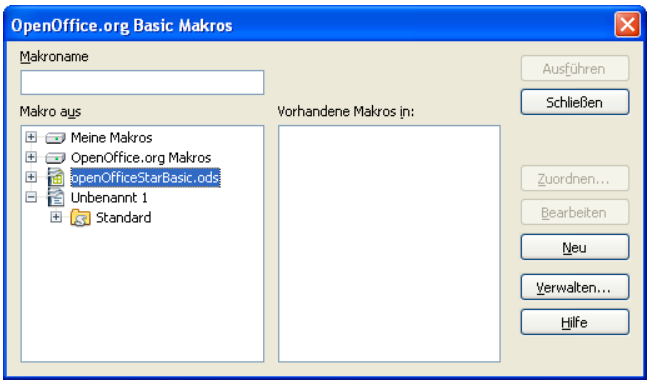


Abbildung 3.3
Übersicht Makros

Sie wählen entweder Standard unter “Meine Makros” oder openOfficeStarBasic.ods[1] aus und clicken auf Neu. OpenOffice erzeugt ein neues Modul, in dem Sie Ihre benutzerdefinierten Funktionen eingeben können. Existiert bereits ein Modul, können Sie es auswählen und über die Schaltfläche “Bearbeiten” in das Modul wechseln und weitere Funktionen eingeben, bzw. vorhandene Funktionen ändern. Funktionen, die Sie in Modulen unterhalb von “Meine Makros” erzeugen, stehen Ihnen in allen Dateien zur Verfügung. Funktionen, die Sie in Modulen unterhalb des Dateinamens erzeugen, gelten nur innerhalb des jeweiligen Dokuments, in dem Sie das Modul erzeugt haben.

[1] Falls Ihre Datei anders heißt, steht dort natürlich der von Ihnen gewählte Dateiname.

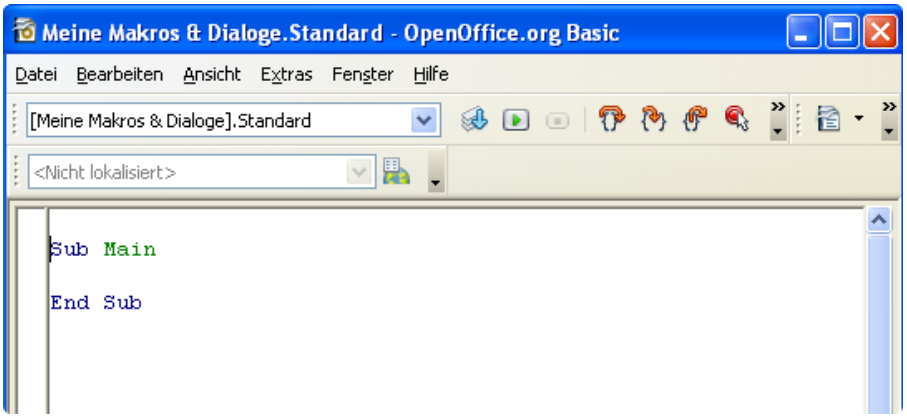


Abbildung 3.4
Entwicklungsumgebung in OpenOffice

In Excel 2007 und Excel 2010 ist die Entwicklungsumgebung zunächst versteckt. Sie aktivieren diese wie auf nachfolgenden Screenshots ersichtlich:

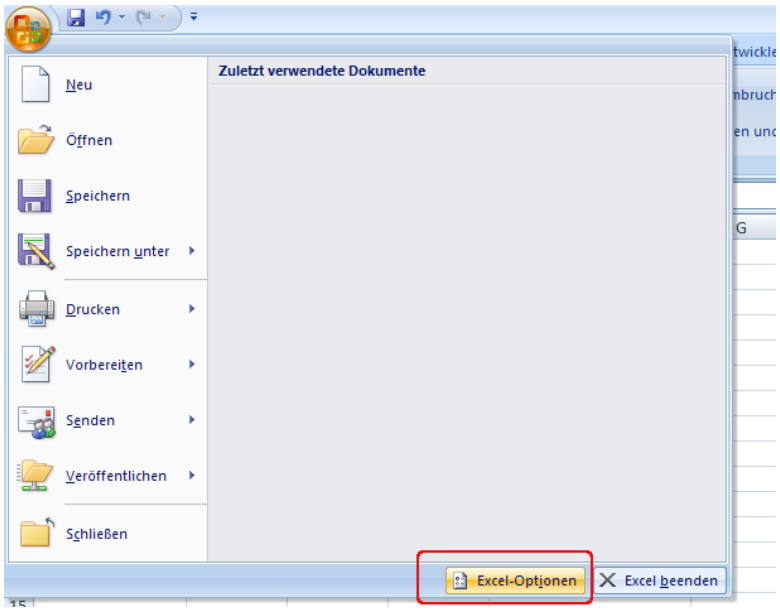


Abbildung 3.5
Excel Optionen aufrufen

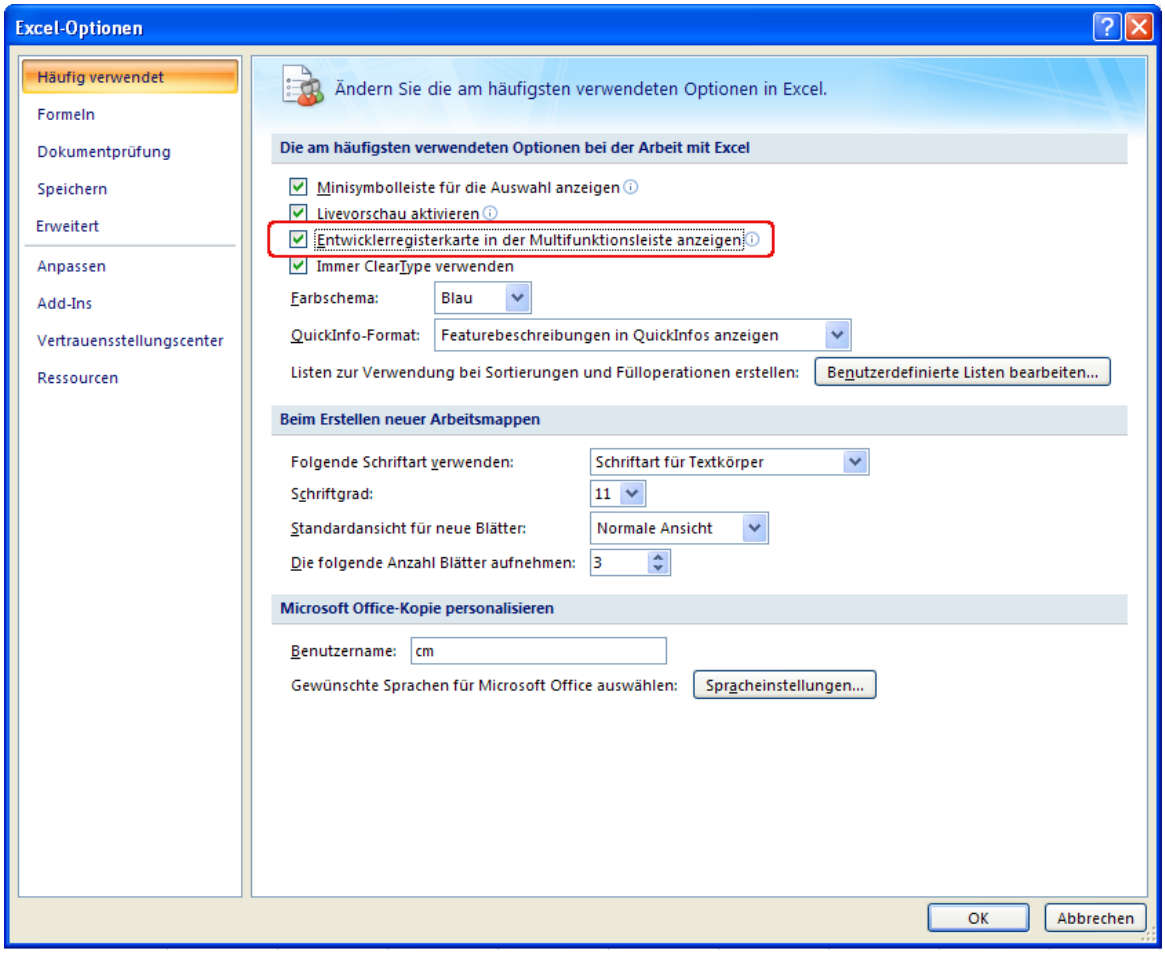


Abbildung 3.6
Excel Optionen einstellen

Beachten Sie: Sowohl in Excel, als auch in OpenOffice 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. 3.8). 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. 3.9). Nach einem weiteren Click erscheint ein neues Fenster (vgl. Abb. 3.9).
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. 3.11). In OpenOffice existiert eine solche Möglichkeit nicht. Erstaunlicherweise erscheinen Ihre selbstgeschriebenen Funktionen auch nicht in der Funktionsübersicht von OpenOffice. In diesem Zusammenhang möchten wir auf die Videoseite auf unseren Homepages verweisen. Sie finden dort hilfreiche Videos, unter anderem auch zum Thema “benutzerdefinierte Funktionen einfügen”.

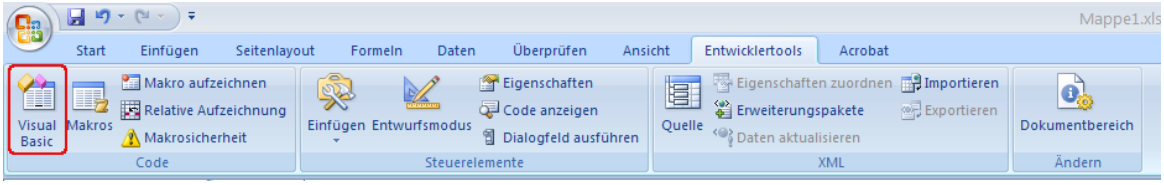


Abbildung 3.7
Excel Entwicklertab

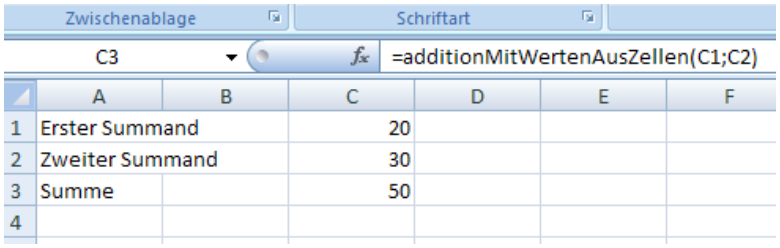


Abbildung 3.8
Excel Funktion per Eingabe einbinden

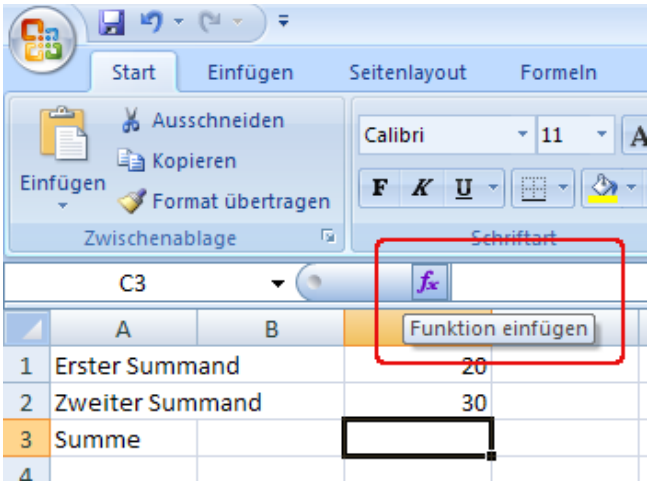


Abbildung 3.9
Das Excel Einfügen-Menü

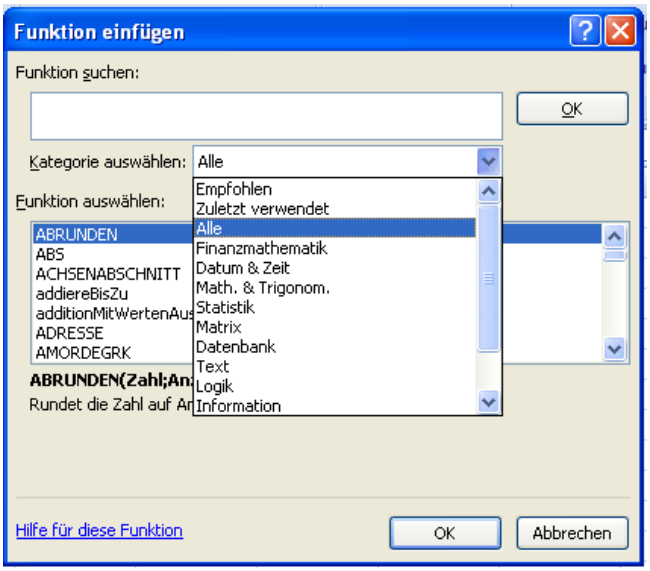


Abbildung 3.10
In Excel verfügbare interne Funktionen

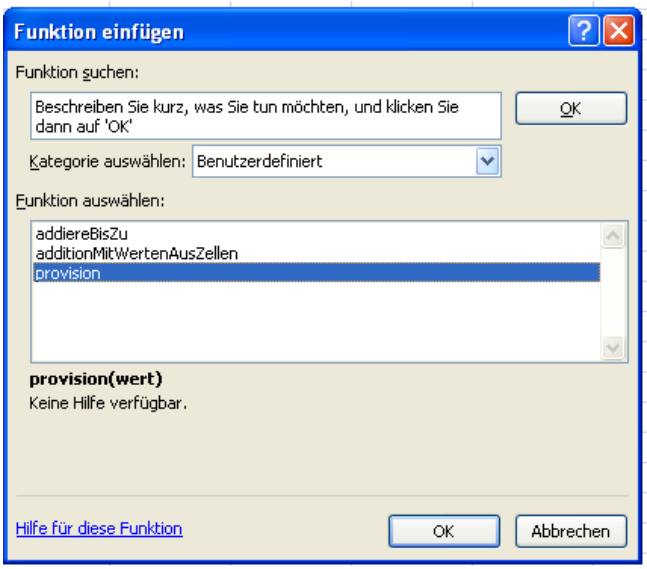


Abbildung 3.11
Die benutzerdefinierte Funktion Provision auswählen

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:

- Ganzzahlvariablen: Auf Variablen diesen Typs können ganze Zahlen z. B. -1, 1000, 3, 7, 12, -1000 gespeichert werden.
- reellen oder Fließkommazahlen: z.B. -1.2, -100.001, 3.1, 7.3, 1000.02

- Wahrheitswerten: true oder false
- Strings: Zeichenketten z.B. „Bernd Blümel“, „Christiane Schäfer“ oder auch “0815Test”.

Abb. 4.1 gibt eine Übersicht über die in VBA möglichen Datentypen. Wie wir sehen, gibt es zwei Datentypen für Fließ-

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

Abbildung 4.1
Datentypen in VBA

kommazahlen 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. 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 2 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 Integer
dim zweiterSummand As Integer
dim summe As Integer
```

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

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"
```

Auch dies funktioniert (ärgerlicherweise) ohne Fehlermeldung. 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
```

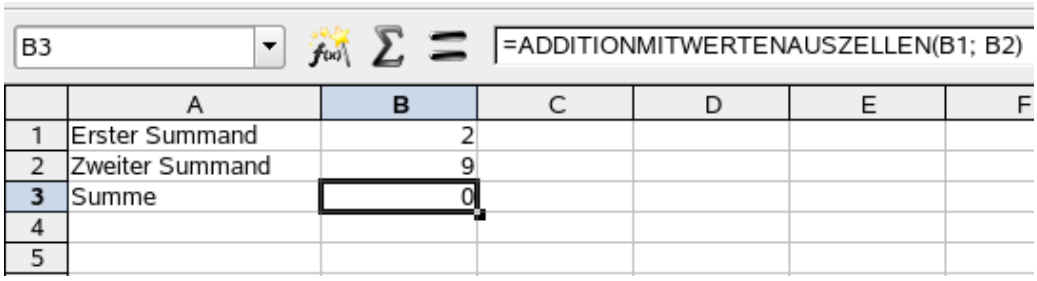


Abbildung 4.2
Addieren mit Fehler

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: variablen.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
```

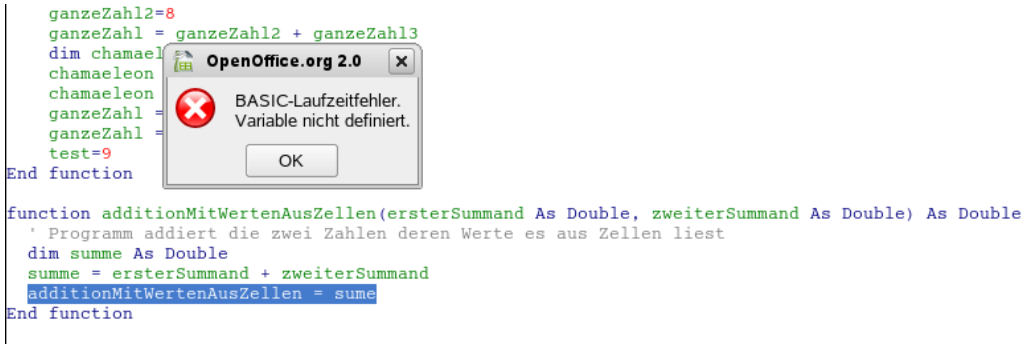


Abbildung 4.3
Fehlermeldung aus VBA

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.

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 keine Rolle (summe=Summe=suMME=SUMME).
5. Variablennamen dürfen außer dem _ keine Sonderzeichen enthalten.
6. Der Funktionsname sollte 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.

4.4 Konstanten

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

Beispiel 4.4 *Konstante Werte in einem VBA-Programm*

```
'Dateiname: konstante.xls
Option Explicit
function verkaufsPreis(einkaufspreis As Double) As Double
          dim nettoVerkaufspreis As Double
          dim bruttoVerkaufspreis As Double
          nettoVerkaufspreis = 1.19 * einkaufspreis
          bruttoVerkaufspreis = 1.19 * nettoVerkaufspreis
          verkaufsPreis = 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 *Ein Programm zur Umsatzsteuerberechnung*

```
'Dateiname: konstante.xls
Option Explicit
function umsatzsteuer(nettopreis As Double) As Double
          dim mwst As Double
          const umsatzsteuersatz As Double = 0.19
          mwst = umsatzsteuersatz * nettopreis
          umsatzsteuer = mwst
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:

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

Abbildung 4.4
Operatoren in VBA

Die arithmetischen Operatoren kennen Sie bereits. Gemeinsam mit der Möglichkeit, Klammern zu setzen, können wir also in VBA beliebige Formeln programmieren. Ein bisschen ungewohnt sind vielleicht 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 Module sowie die Integerdivision zur Berechnung benutzt:

Beispiel 4.6 *Ein Programm zur Umrechnung von Sekunden*

```
Option Explicit
Function sekundenVerarbeiten(sekunden As Long) As String
      'Deklaration der Variablen
      Dim tageinSekunden As Integer
      Dim stundenInSekunden As Integer
```

```
        Dim minutenInSekunden As Integer
        Dim restSekunden As Long
        'Deklaration der Konstanten
        Const sekundenProTag As Long = 86400
        Const sekundenProStunde As Integer = 3600
        Const sekundenProMinute As Integer = 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
        sekundenVerarbeiten = tageinSekunden & " Tag(e) " & stundenInSekunden & " Stunden " & ➥
            minutenInSekunden & " Minuten " & sekunden & " Sekunden"
    End Function
```

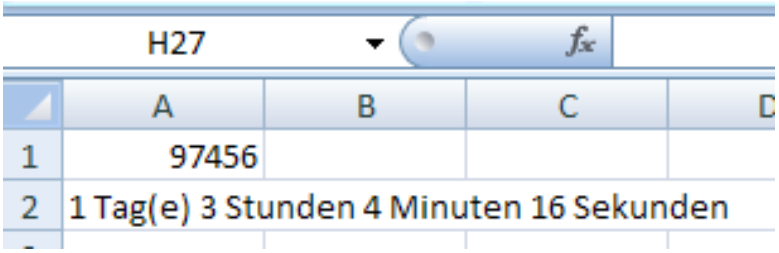


Abbildung 4.5
Screenshot Sekunden verarbeiten

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*

```
        Option Explicit
        function stringVerkettung() As String
                  dim s1 As String
                  dim s2 as String
                  s1 = "ab"
                  s2 = "cd"
                  stringVerkettung = 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.
Wir zeigen Ihnen sofort eine mögliche Realisierung dieser Problematik (Beispiel 5.1):

Beispiel 5.1 *Division durch Null*

```
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 Variblen quotient mit dem Datentyp ➥
            Double (Fließkommazahl)s
        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 folgendes Bild:
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) einleitet. Auf *if* folgt ein Vergleich. Dazu

[1] Laufzeitfehler (engl.: runtime errors) treten während des Programmablaufs auf und können verschiedene Ursachen haben, z.B. eine Division durch 0

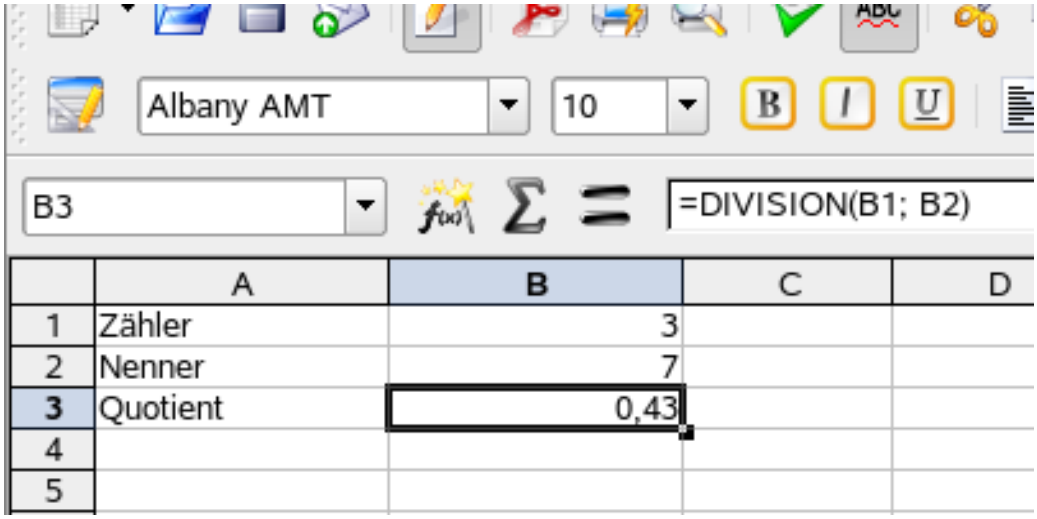


Abbildung 5.1
Das Divisionsprogramm in der Tabellenkalkulation

benötigt man die bereits in Kap. 4.5 vorgestellten Vergleichsoperatoren. Hier wird getestet, 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*

```
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.

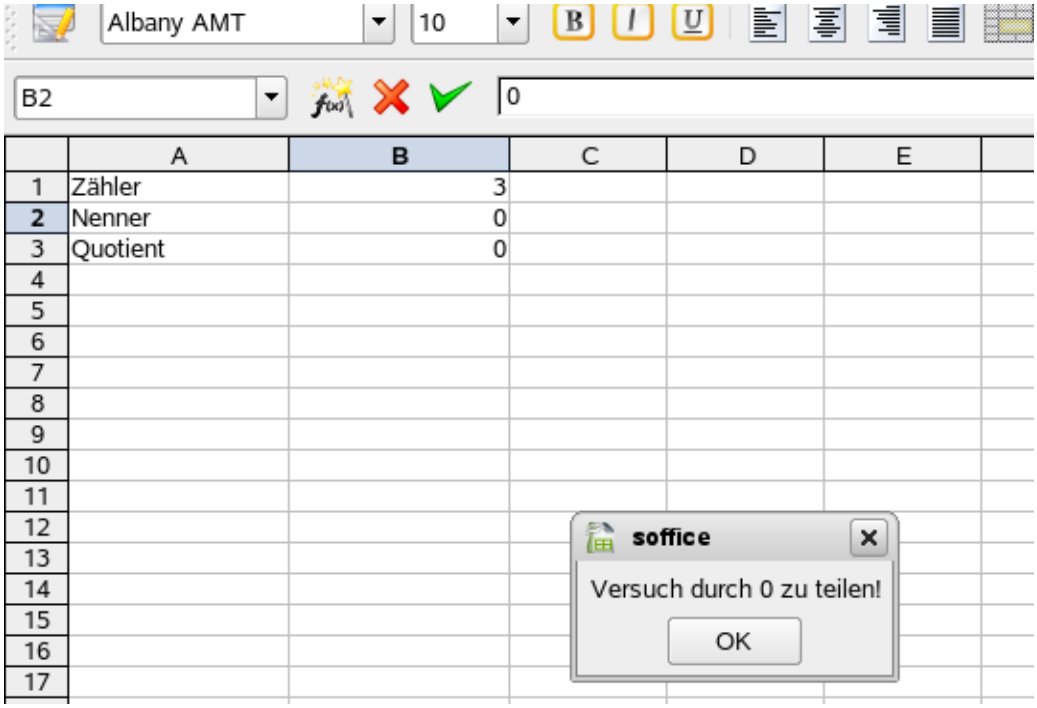


Abbildung 5.2
Das Divisionsprogramm in der Tabellenkalkulation mit Nenner 0

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
```

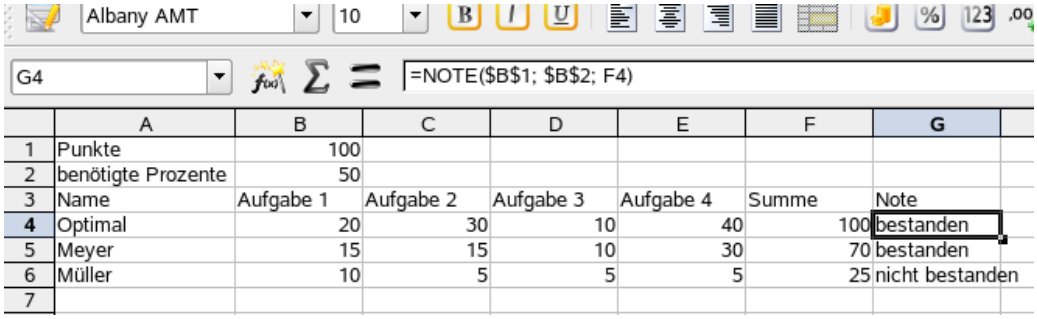
„Logischer Ausdruck“ ist ein Ausdruck, der ein „logisches Ergebnis“ erzeugt. Bei logischen Ergebnissen gibt es nur zwei Möglichkeiten. Entweder ist der Ausdruck
true oder er ist *false*.
Logische Ausdrücke sind daher im Wesentlichen:

- 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.

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 meine 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. Betrachten wir zunächst die zugehörige Pro-



	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

grammierung:

Beispiel 5.3 *Notenprogramm*

```
Option Explicit
function note(maximalpunkte As Integer, benoetigteProzente As Double, _
                              erreichtePunkte As Integer) 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 Integer, 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.

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[3]. 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.
Natürlich können wir diese Problematik mit der normalen *if*-Anweisung durch geschachtelte *if*'s lösen:

[2] Die Erweiterung auf mehr als 2 Teilnehmer bleibt Ihnen überlassen.
[3] mit weniger Alternativen

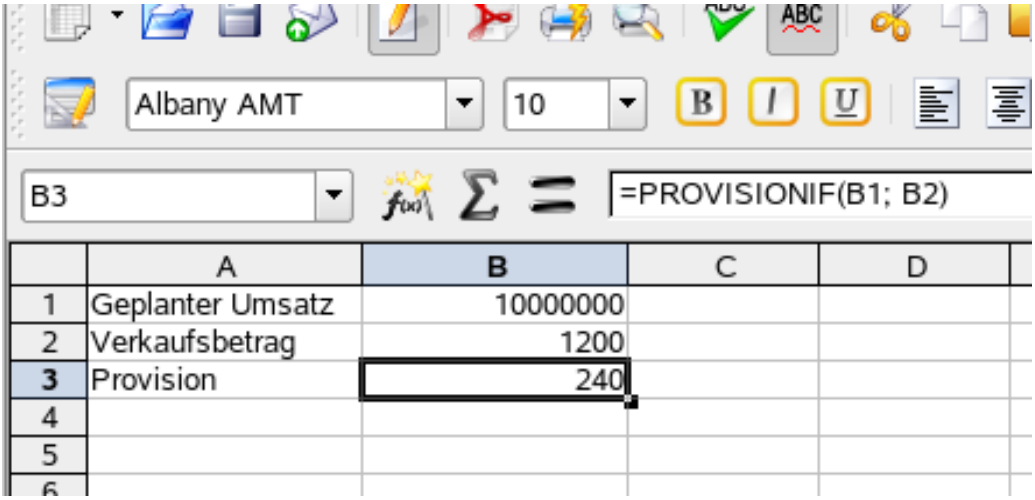


Abbildung 5.4
Das erste Provisionsprogramm mit if

Beispiel 5.4 *Provision mit if*

```
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*

```
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*

```
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 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
        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)[4]
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.

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

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)*

```
Option Explicit
function noteIfElseIF(maximalpunkte As Integer, benoetigteProzente As Double, _
                                        erreichtePunkte As Integer) As String
    dim benoetigtePunkte As Integer
    dim spanne As Integer
    dim punkteProNote As Integer
    dim punkteZweiZwischenNoten As Integer
    dim punkteDreiZwischenNoten As Integer
    dim grenze4_0 as Integer
    dim grenze3_7 as Integer
    dim grenze3_3 as Integer
    dim grenze3_0 as Integer
    dim grenze2_7 as Integer
    dim grenze2_3 as Integer
    dim grenze2_0 as Integer
    dim grenze1_7 as Integer
    dim grenze1_3 as Integer
    dim grenze1_0 as Integer

    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.

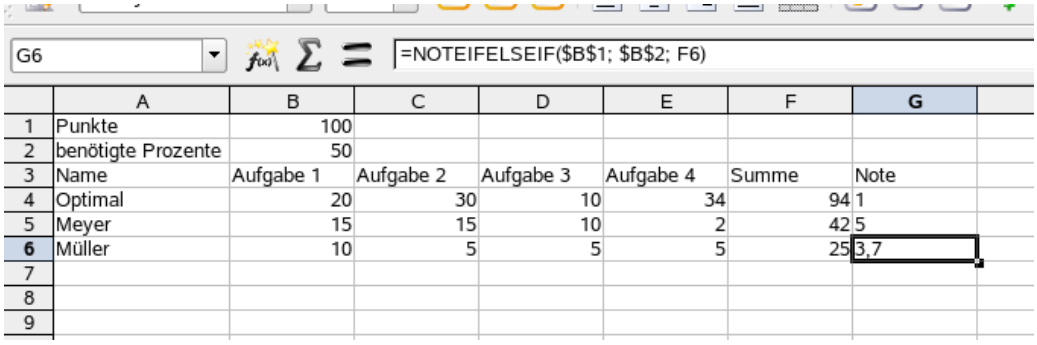
```
grenze4_0=benoetigtePunkte
grenze3_7=benoetigtePunkte+punkteZweiZwischenNoten

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[5], 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[6] dazu führen, dass alle Teilnehmer, die die Klausur bestanden haben, eine Vier erhalten.[7]
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 Teil2)

5.3.1 Beispiel und Erklärung

Neben der *if elseIf*-Anweisung gibt es in VBA eine weitere Möglichkeit, Mehrfachauswahlen zu programmieren. 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*

```
function provisionSelectCase(geplanterUmsatz as double, verkaufsbetrag as double) as Double
    dim provisionInProzent As double
    const umsatzGrenze3 as double =1000000
```

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

```
    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

    select case geplanterUmsatz
        case is >= umsatzGrenze3:
            provisionInProzent=provisionUmsatzGrenze3
        case is >= umsatzGrenze2:
            provisionInProzent=provisionUmsatzGrenze2
        case is >= umsatzGrenze1:
            provisionInProzent=provisionUmsatzGrenze1
        case else:
            provisionInProzent=provisionSonst
    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 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 beide überein oder 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*

```
Option Explicit
function noteSelectCase(maximalpunkte As Integer, benoetigteProzente As Double, _
                                                  erreichtePunkte As Integer) As String
dim benoetigtePunkte As Integer
dim spanne As Integer
dim punkteProNote As Integer
dim punkteZweiZwischenNoten As Integer
dim punkteDreiZwischenNoten As Integer
dim grenze4_0 as Integer
dim grenze3_7 as Integer
dim grenze3_3 as Integer
dim grenze3_0 as Integer
dim grenze2_7 as Integer
dim grenze2_3 as Integer
dim grenze2_0 as Integer
dim grenze1_7 as Integer
```

```
        dim grenze1_3 as Integer
        dim grenze1_0 as Integer

        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

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 6.1 *Das Programm „Hello World“ in einer Prozedur*

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

Das Programm ist selbsterklärend:
sub helloWorld() ➪ leitet die Prozedur ein,
MsgBox(„Hallo Welt!“) ➪ gibt den String „Hallo Welt“ in einer Messagebox aus und
end sub ➪ beendet das Programm wieder.

> ℹ Der wesentliche Unterschied zwischen einer Prozedur und einer Funktion besteht darin, dass eine Prozedur keinen Rückgabewert besitzt. Daher wird der Prozedur bei der Deklaration kein Datentyp zugewiesen.

Wie führen wir diese Prozedur jetzt aus?
Funktionen hatten wir in die Tabellenkalkulationen eingebunden. Dort werden sie automatisch ausgeführt, wenn sich ihre

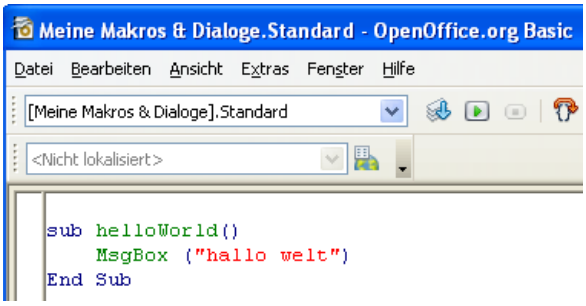


Abbildung 6.1
Prozedur Hallo Welt ausführen

Zellbezüge ändern. Mit Prozeduren geht das nicht. Unsere erste Prozedur können wir zunächst nur im VBA-Editor ausführen. In Excel bewegen wir den Cursor in die Prozedur, in OpenOffice muss es die erste Funktion oder Prozedur eines Moduls sein. Durch Betätigen des Abspielknopfs im VBA-Editor wird die Prozedur dann ausgeführt.

Tabelle 6.1
Gegenüberstellung Funktion - Prozedur

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

6.1 Wann benutzerdefinierte Funktion, wann Ereignisprozedur

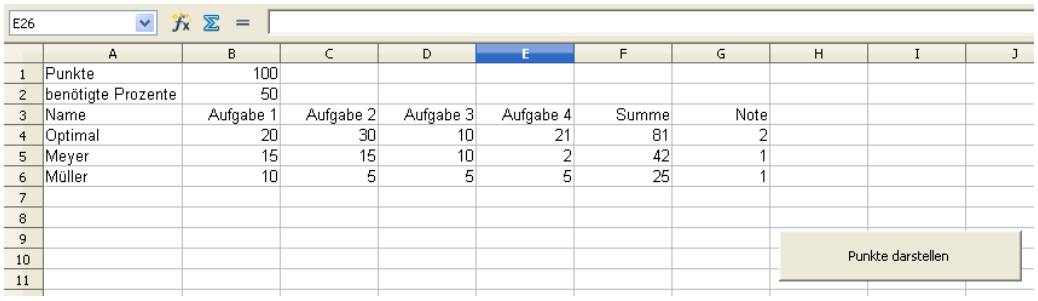
Benutzerdefinierte Funktionen benutzen wir, wenn wir den Wert einer Zelle in Abhängigkeit von den Werten anderer Zellen bestimmen wollen und wenn wir zudem wollen, dass der Wert der Zelle unserer benutzerdefinierten Funktion automatisch neu errechnet wird, wenn sich eine der Zellen, auf die sie sich 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.

6.2 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. Nicht gut, weil Arbeit! Wir könnten uns nun die Punktegrenzen, die unser Programm ja ausrechnet, in MsgBoxen ausgeben lassen und abschreiben. Das wäre zumindest 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. Nicht wirklich schön ☺.

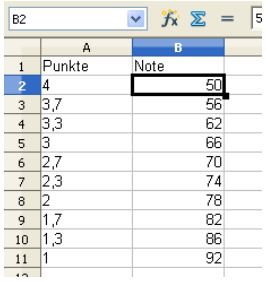
> ℹ Übrigens können benutzerdefinierte Funktionen nur in die Zelle schreiben, in die sie eingebunden wurden.

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 mir die Punktegrenzen in die Zellen meines Arbeitsblatts (vgl. Abb. 6.3). 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 aus Datenschutzgründen nicht so einfach drucken und aushängen.
Wie es aussehen könnte, zeigen Abb. 6.2 und Abb. 6.3.



	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								Punkte darstellen		
11										

Abbildung 6.2
Schaltfläche zum Erzeugen des Noten-Punkte Zusammenhangs



	A	B
1	Punkte	Note
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 6.3
Tabellenblatt Noten-Punkte darstellen

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 Ereig-

nisprozedur kommt übrigens daher, dass das Clicken auf eine Schaltfläche für eine Tabellenkalkulation ein Ereignis ist. Leider sind Ereignisprozeduren in OpenOffice und Excel unterschiedlich gelöst. Es folgt also jetzt für jede Lösung ein Kapitel:

6.3 Ereignisprozeduren in OpenOffice anhand des Notenbeispiels

[1] Zunächst müssen wir die Schaltfläche in das Tabellenblatt bekommen. Dazu benutzen wir die Steuerelemente-Toolbox. Diese erhalten wir über die Menüs:
☞Ansicht➪Symbolleisten➪Formular-Steuerelemente (vgl. Abb. 6.4)
Dort aktivieren wir den Entwurfsmodus (vgl. Abb. 6.5).

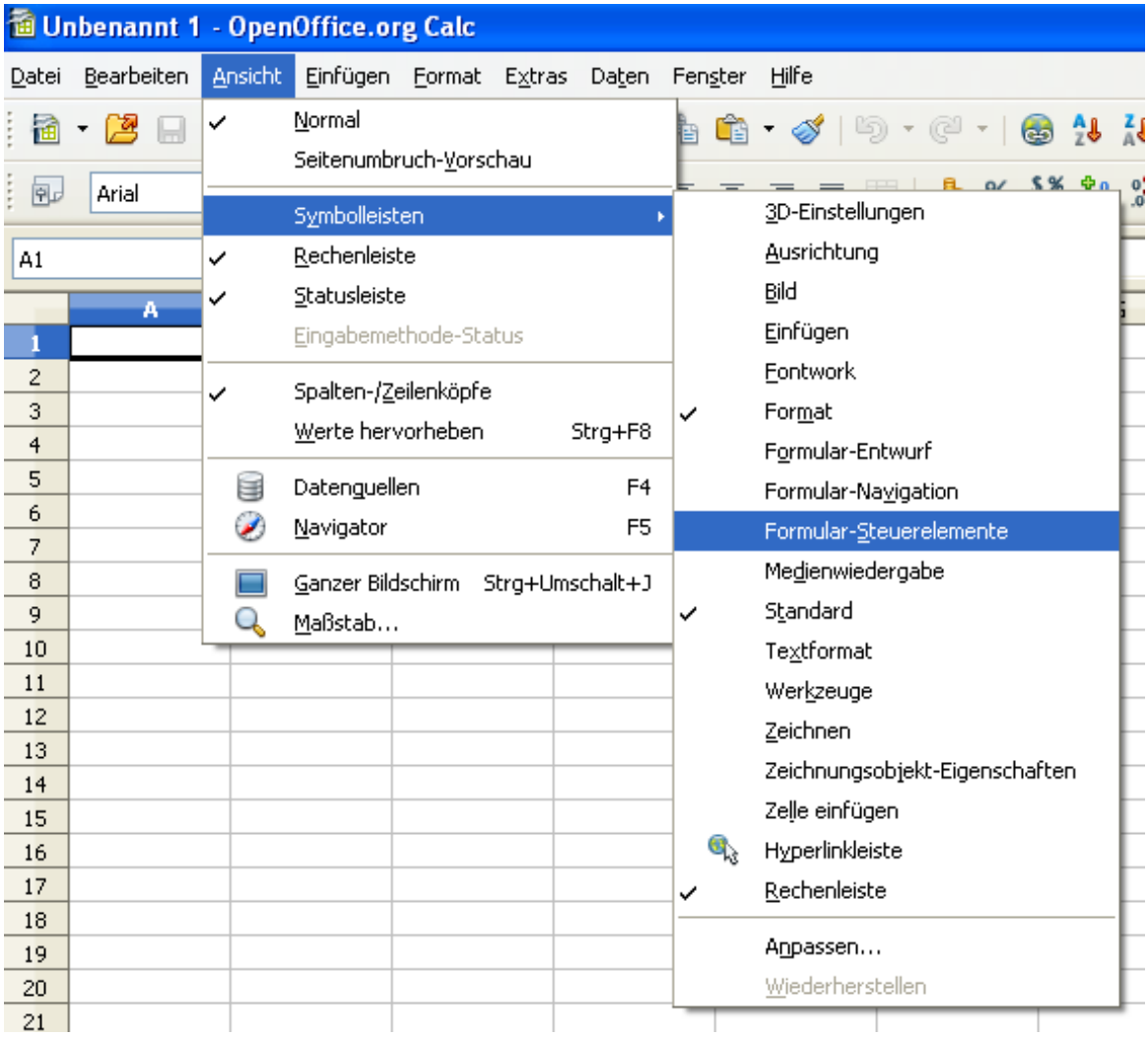


Abbildung 6.4
Menu Formular Steuerelemente

Sie clicken auf das Symbol für Schaltflächen[2] und malen die Schaltfläche in unser Tabellenblatt (vgl. Abb. 6.6).
Danach clicken Sie mit der rechten Maustaste auf die Schaltfläche und wählen „Kontrollfeld“ aus (vgl. Abb. 6.7). Sofort

[1] Excel weiter auf Seite 53
[2] Sie müssen selbst herausbekommen, welches das ist ☺.

Abbildung 6.5
Symbolleiste zum Einfügen von Formularelementen in OpenOffice

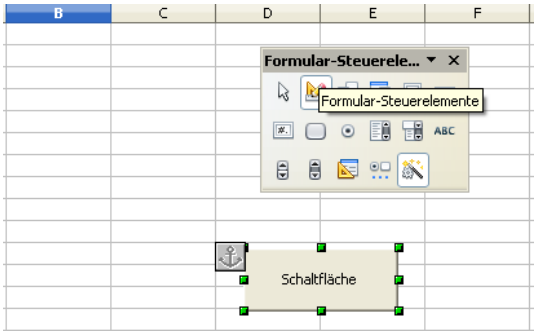


Abbildung 6.6
Button im Tabellenblatt OpenOffice

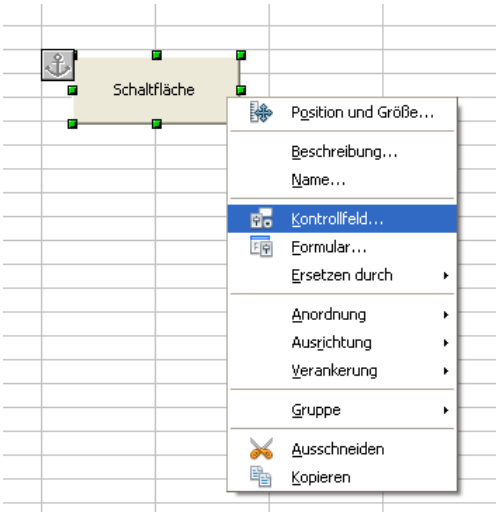


Abbildung 6.7
Auswahl Kontrollfeldoption OpenOffice

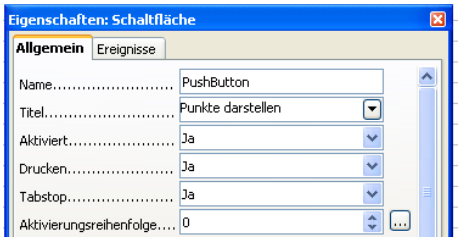


Abbildung 6.8
Allgemeine Eigenschaften der Schaltfläche in OpenOffice

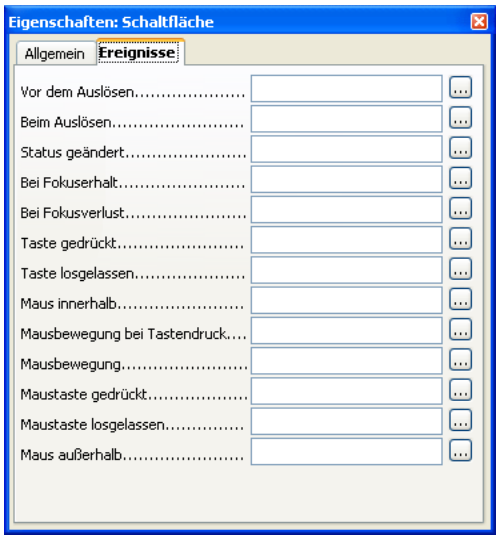


Abbildung 6.9
Ereignis-Eigenschaften der Schaltfläche in OpenOffice

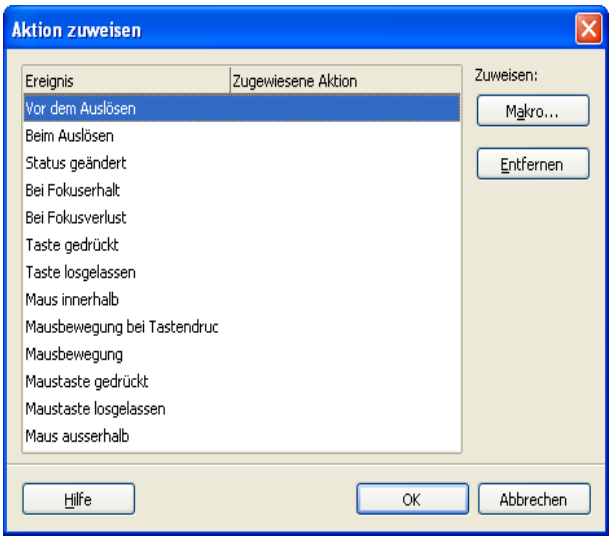


Abbildung 6.10
Makros Aktionen zuweisen

erscheint das in Abb. 6.8 dargestellte Fenster. Wenn er nicht bereits vorausgewählt ist, wählen Sie den Reiter „Allgemein“ aus. In die Eingabefläche neben Titel tragen Sie die gewünschte Beschriftung der Schaltfläche ein. Wir haben uns für „Punkte darstellen“ entschieden. Schauen Sie sich auch die anderen Möglichkeiten zur Konfiguration der Schaltfläche, die dieses Fenster bietet, an. In dem Moment, wo Sie „Punkte darstellen“ als Titel eintragen, ändert die Schaltfläche Ihre Beschriftung und „Punkte darstellen“ erscheint.
Aktivieren Sie nun den Reiter „Ereignisse“. Sie sehen das in Abb. 6.9 dargestellte Fenster. Hier sehen Sie alle Ereignisse, die OpenOffice kennt. Wir möchten die Ereignisprozedur starten, wenn die Schaltfläche ausgelöst wird. Dies ist demnach das Ereignis „Beim Auslösen“. Schauen Sie sich auch die anderen Ereignisse an. Danach clicken Sie auf die Schaltfläche mit den drei Punkten neben „Beim Auslösen“. Alternativ können Sie, wenn Sie den Namen der Ereignisprozedur, die Sie mit der Schaltfläche verbinden wollen, wissen, auch in das Eingabefeld neben „Beim Auslösen“ diesen Namen eintragen und sind direkt fertig. Wenn Sie geclickt haben, erscheint das in Abb. 6.10 dargestellte Fenster.
Clicken Sie auf Makro und wählen Sie im danach erscheinenden Fenster den Namen der Ereignisprozedur aus, die Sie der Schaltfläche zuordnen wollen. Dies bedeutet, dass Sie die Ereignisprozedur vorher geschrieben haben müssen. Wir haben das gemacht und der Ereignisprozedur den Namen *notenPunkteDarstellen* gegeben. Also wählen wir diesen Namen aus. Jedes Mal, wenn wir nun auf den Button clicken, wird die Ereignisprozedur durchgeführt. Schauen wir uns nun den Algorithmus sowie die programmierte Ereignisprozedur an:

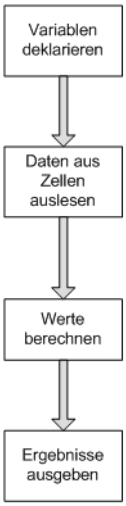


Abbildung 6.11
Programmablauf Noten-Punkte darstellen

Beispiel 6.2 *Die Ereignisprozedur für die Punktegrenzen in OpenOffice*

```
sub notenPunkteDarstellen()
'Variablen deklarieren ####################################
  dim maximalpunkte As Integer
  dim benoetigteProzente as Double
  dim benoetigtePunkte As Integer
  dim spanne As Integer
  dim punkteProNote As Integer
  dim punkteZweiZwischenNoten As Integer
  dim punkteDreiZwischenNoten As Integer
  dim grenze4_0 as Integer
  dim grenze3_7 as Integer
  dim grenze3_3 as Integer
  dim grenze3_0 as Integer
  dim grenze2_7 as Integer
  dim grenze2_3 as Integer
  dim grenze2_0 as Integer
  dim grenze1_7 as Integer
  dim grenze1_3 as Integer
  dim grenze1_0 as Integer
  dim myDoc as Object
  dim mySheet as Object
  dim cell as Object
 'Daten aus Zellen auslesen ################################
  myDoc = thisComponent
  mySheet = myDoc.sheets(0)

  cell=mySheet.getCellByPosition(1,0)
  maximalpunkte=cell.Value

  cell=mySheet.getCellByPosition(1,1)
  benoetigteProzente=cell.Value

  '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 ####################################
  mySheet = myDoc.Sheets(1)
  cell=mySheet.getCellByPosition(0,0)
  cell.String="Punkte"
  cell=mySheet.getCellByPosition(1,0)
  cell.String="Note"

  cell=mySheet.getCellByPosition(0,1)
  cell.String="4"
  cell=mySheet.getCellByPosition(1,1)
  cell.Value=grenze4_0

  cell=mySheet.getCellByPosition(0,2)
```

```
    cell.String="3,7"
    cell=mySheet.getCellByPosition(1,2)
    cell.Value=grenze3_7

    cell=mySheet.getCellByPosition(0,3)
    cell.String="3,3"
    cell=mySheet.getCellByPosition(1,3)
    cell.Value=grenze3_3

    cell=mySheet.getCellByPosition(0,4)
    cell.String="3"
    cell=mySheet.getCellByPosition(1,4)
    cell.Value=grenze3_0

    cell=mySheet.getCellByPosition(0,5)
    cell.String="2,7"
    cell=mySheet.getCellByPosition(1,5)
    cell.Value=grenze2_7

    cell=mySheet.getCellByPosition(0,6)
    cell.String="2,3"
    cell=mySheet.getCellByPosition(1,6)
    cell.Value=grenze2_3

    cell=mySheet.getCellByPosition(0,7)
    cell.String="2"
    cell=mySheet.getCellByPosition(1,7)
    cell.Value=grenze2_0

    cell=mySheet.getCellByPosition(0,8)
    cell.String="1,7"
    cell=mySheet.getCellByPosition(1,8)
    cell.Value=grenze1_7

    cell=mySheet.getCellByPosition(0,9)
    cell.String="1,3"
    cell=mySheet.getCellByPosition(1,9)
    cell.Value=grenze1_3

    cell=mySheet.getCellByPosition(0,10)
    cell.String="1"
    cell=mySheet.getCellByPosition(1,10)
    cell.Value=grenze1_0
  end sub
```

Nach der Deklaration der Prozedur finden Sie, wie immer, die Variablendeklarationen. Ganz am Ende der Variablendeklarationen erkennen Sie drei Variablen eines bisher nicht behandelten Typs, des Typs *Objekt*.

```
dim myDoc as Object
dim mySheet as Object
dim cell as Object
```

Über den Datentyp *Objekt* und über objektorientierte Programmierung werden Sie in diesem Kurs nicht so wirklich viel lernen. Für unsere jetzigen Zwecke reicht es völlig aus, wenn Sie wissen, dass wir die Arbeitsmappe an sich, die Tabelle und jede Zelle, auf die wir zugreifen wollen, auf einer Variablen benötigen. *myDoc* ist für die Arbeitsmappe, *mySheet* für die Tabelle und *cell* für die jeweilige Zelle. In der Codezeile

```
myDoc = thisComponent
```

wird VBA angewiesen, die aktuelle Arbeitsmappe auf der Variablen *myDoc* zu speichern. Die Zeile

```
mySheet = myDoc.sheets(0)
```

speichert die erste Tabelle der Arbeitsmappe auf der Variablen *mySheet*. Die Tabellen sind durchnummeriert in OpenOffice und die Nummerierung beginnt bei 0. Daher ist *sheets(0)* die erste Tabelle. Als nächstes müssen wir jetzt *maximalpunkte*

und *benoetigteProzente* aus dem Tabellenblatt lesen[3]. *maximalpunkte* steht in der Zelle B1. Zugriff auf die Zelle B1 bekommen wir in OpenOffice u.A. durch folgende Zeile:

```
cell=mySheet.getCellByPosition(1, 0)
```

Die Funktion *getCellByPosition* erwartet als Übergaben die Spalte und die Zeile der Zelle, die angesprochen werden soll. Die Zelle B1 ist zweite Spalte, erste Zeile. Die Zählung der Spalten und Zeilen beginnt bei OpenOffice bei 0, die zweite Spalte ist also Spalte 1 (erste Spalte ist Spalte 0), die erste Zeile dementsprechend Zeile 0.
Dann benötigen wir den Wert dieser Zelle. Die Zeile

```
maximalpunkte=cell.Value
```

schreibt ihn auf die Variable *maximalpunkte*.

In OpenOffice verwendet man *cell.Value*, wenn der Wert einer Zelle eine Zahl ist, *cell.String* wenn es sich um einen String handelt.

Mit den Zeilen

```
cell=mySheet.getCellByPosition(1,1)
benoetigteProzente=cell.Value
```

holen wir dann auf völlig analoge Weise *benoetigteProzente*. 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.
Mit den Zeilen

```
mySheet = myDoc.Sheets(1)
```

besetzen wir nun die Variable mySheet mit dem zweiten, noch leeren Tabellenblatt[4].
Schreiben in eine Zelle erfolgt nun völlig analog zum Lesen aus einer Zelle. Wir lesen die Zelle auf eine Variable ein:

```
cell=mySheet.getCellByPosition(0,0)
```

und setzen dann z.B. mit

```
cell.String="Punkte"
```

bzw. wenn wir eine Zahl in eine Zelle schreiben wollen, z.B. mit

```
cell.Value=grenze1_7
```

den Wert der Zelle. Das ist der ganze Rest dieses Programms[5].

6.4 Ereignisprozeduren in Excel 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 (vgl. Abb. 6.12).

[3] In Beispiel 5.7 hatten wir das ja über Zellbezüge der benutzerdefinierten Funktion gelöst.
[4] Beachten Sie auch hier: Die Zählung beginnt bei 0, daher *Sheets(1)*
[5] OpenOffice weiter auf Seite 46

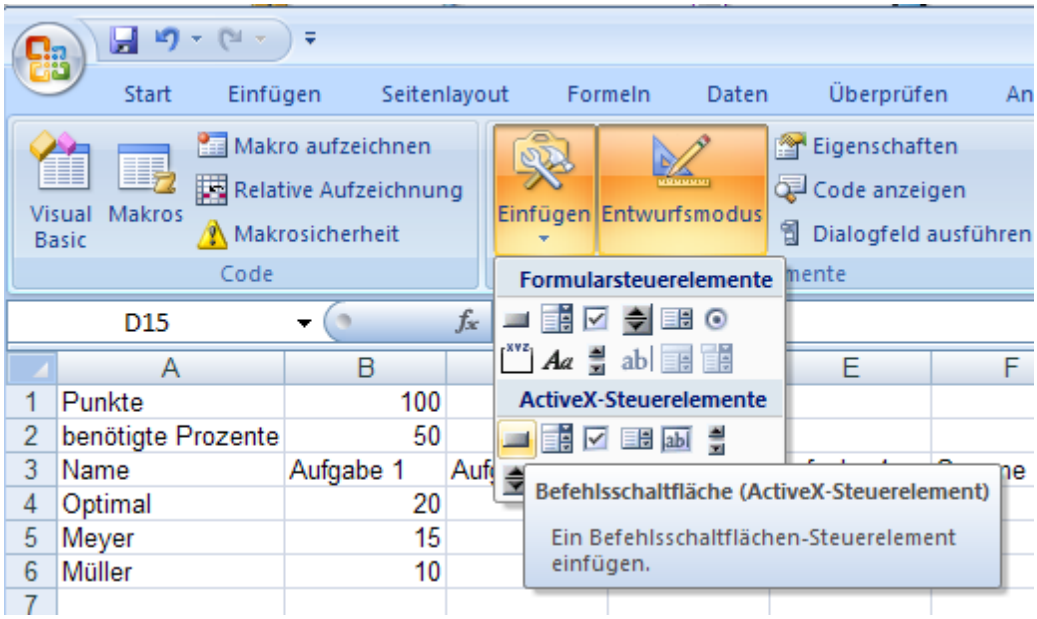


Abbildung 6.12
Öffnen der Steuerelemente Toolbox

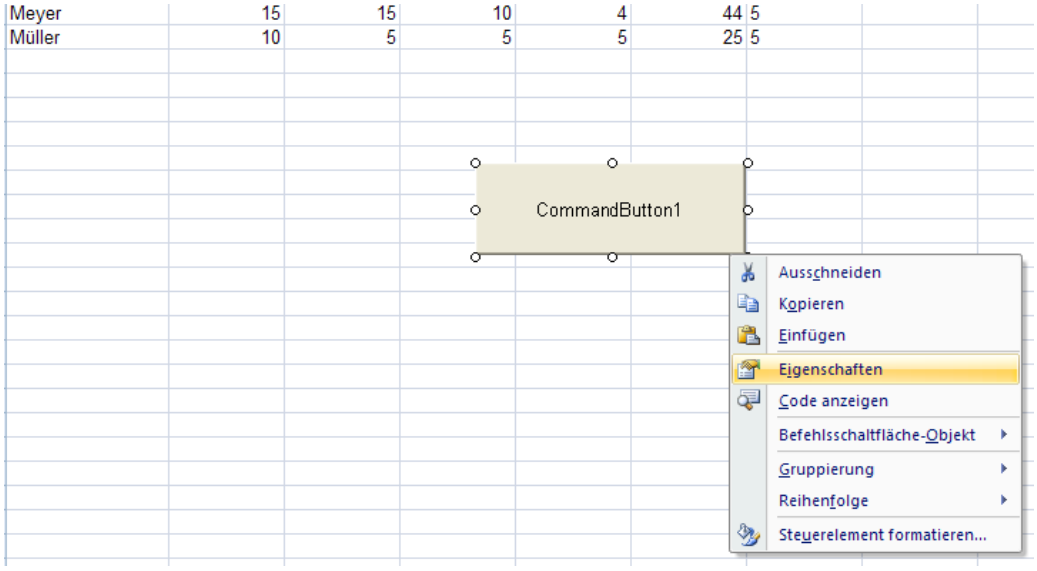


Abbildung 6.13
Eigenschaften der Schaltfläche

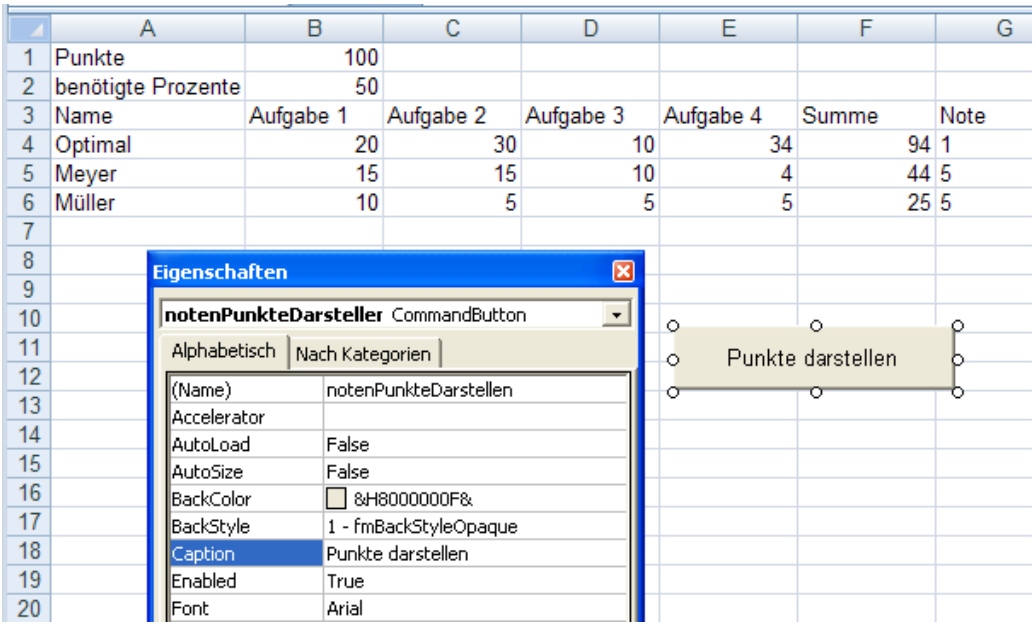


Abbildung 6.14
Eigenschaften der Schaltfläche verändern

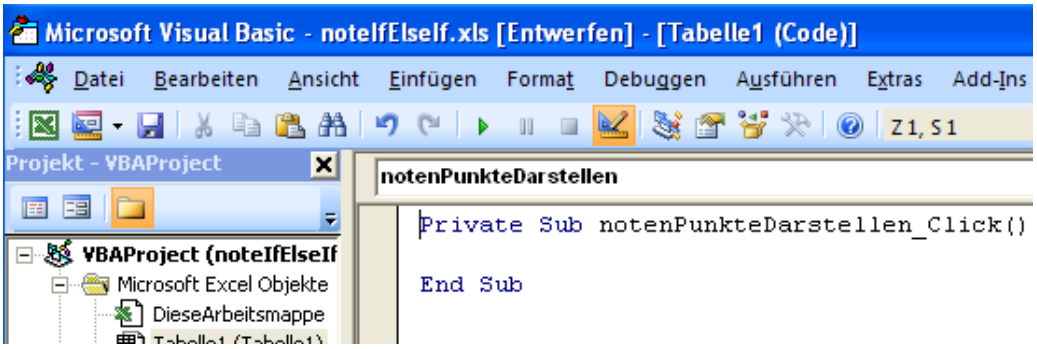


Abbildung 6.15
Code "hinter" Schaltfläche legen

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. 6.14). 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.
Doppel-Clicken Sie nun auf ihre neue Schaltfläche. Excel schaltet nun zum VBA-Editor um (vgl. Abb. 6.15). 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.

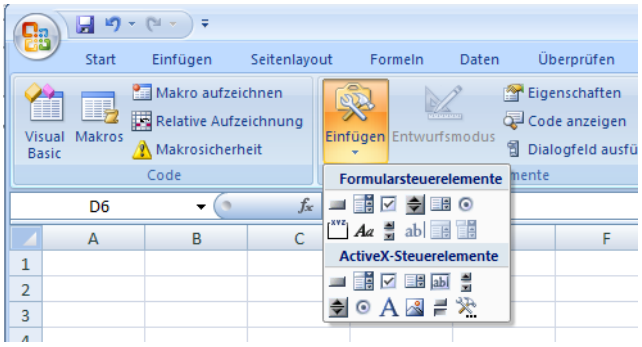


Abbildung 6.16
Steuerelement-Toolbox Excel

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. Zunächst, vor der Programmierung den Algorithmus der Aufgabe.

Betrachten wir nun den VBA-Code, um die Aufgabenstellung in Excel zu lösen:

Beispiel 6.3 *Die Ereignisprozedur für die Punktegrenzen in Excel*

```
Sub notenPunkteDarstellen_Click()
    ' Variablen deklarieren##############################################
    Dim maximalpunkte As Integer
    Dim benoetigteProzente As Double
    Dim benoetigtePunkte As Integer
    Dim spanne As Integer
    Dim punkteProNote As Integer
    Dim punkteZweiZwischenNoten As Integer
    Dim punkteDreiZwischenNoten As Integer
    Dim grenze4_0 As Integer
    Dim grenze3_7 As Integer
    Dim grenze3_3 As Integer
    Dim grenze3_0 As Integer
    Dim grenze2_7 As Integer
    Dim grenze2_3 As Integer
```

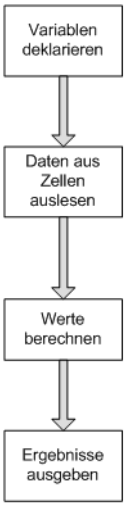


Abbildung 6.17
Programmablauf Noten-Punkte darstellen

```
Dim grenze2_0 As Integer
Dim grenze1_7 As Integer
Dim grenze1_3 As Integer
Dim grenze1_0 As Integer
' Werte auslesen#####################################################
maximalpunkte = Cells(1, 2)
benoetigteProzente = 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[6]. *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.

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

Kapitel 7

Wiederholungsanweisungen (Schleifen)

7.1 Die do-while-Anweisung

7.1.1 Motivation und Beispiel

Betrachten Sie die in Abb. 7.1 dargestellte Version unseres Provisionsbeispiels: In der ersten Zeile sehen Sie den ver-

C5 =PROVISIONIFELSEIFKONSTANTE(B1; B5)

	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 7.1
Mehrere Provisionsberechnungen in einer Tabelle

einbarten 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. 7.1). 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. 7.2 dargestellt, das Ergebnis in Abb. 7.3.

	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							

Gesamt- und Durchschnittsprovision

Abbildung 7.2
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 7.3
Aggregierte und Durchschnittsprovision in eigener Tabelle

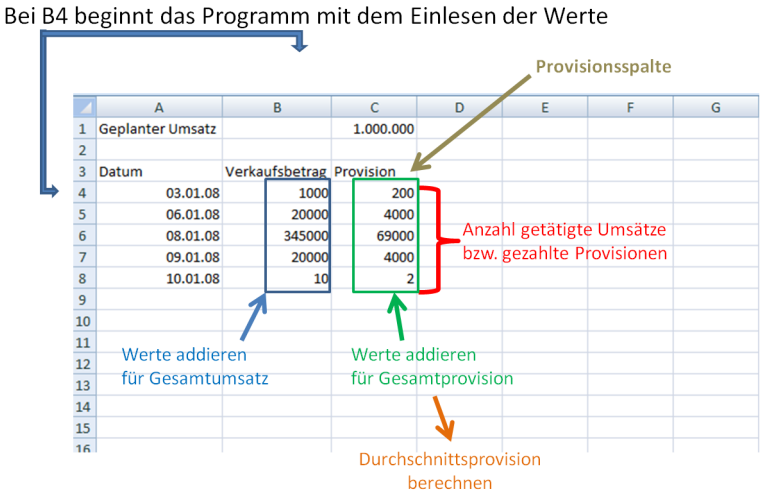


Abbildung 7.4
Bildliche Darstellung der Aufgabe

Konstanten deklarieren

Verkaufsbetragsspalte

Provisionsspalte

Erste Zeile mit Verkaufsbetrag

Variablen deklarieren

gesamtVerkaufsbetrag mit 0 initialisieren

gesamtProvision mit 0 initialisieren

zeilenzaehler den Wert aus Erste Zeile mit Verkaufsbetrag zuweisen

do while Schleife solange bis Provisionsspalte leer

gesamtumsatz= gesamtVerkaufsbetrag + Inhalt der Zelle (zeilenzaehler, VerkaufsbetragSpalte)

gesamtprovision = gesamtprovision + Inhalt der Zelle (zeilenzaehler, Provisionsspalte)

zeilenzaehler = zeilenzaehler +1

loop

anzahl der Umsaetze = zeilenzaehler – Erste Zeile mit Verkaufsbetrag

durchschnittsprovision = gesamtprovision / anzahl der Umsaetze

Berechnete Werte in Tabellenblatt schreiben

Abbildung 7.5
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 nicht, wie viele Provisionsberechnungen in unserer Tabelle sind, d.h. das Programm muss das 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. Da wir in der Tabelle navigieren und direkt auf Zelleninhalte zugreifen müssen, unterscheiden sich die Realisierungen in OpenOffice und in Excel.

Realisierung des Beispiels in OpenOffice

Beispiel 7.1 *Bisheriger Verkaufsbetrag, Gesamt- und durchschnittliche Provision in OpenOffice*[1]

```
sub berechneBisherigenVerkaufsbetragGesamtUndDurchschnittsprovision()
        '-----------------------------
        'Deklaration von Konstanten
        const VERKAUFSBETRAGSPALTE as Integer = 1
        const PROVISIONSSPALTE as Integer = 2
        const ERSTE_ZEILE_MIT_VERKAUFSBETRAG as Integer = 3
        '------------------------------
        'Deklaration von Variablen
```

[1] Excel weiter auf Seite 64

```
        dim gesamtprovision as double
        dim gesamtVerkaufsbetrag as double
        dim zeilenzaehler as Integer
        dim anzahlVerkaeufe as Integer
        dim durchschnittsprovision as Double
        dim myDoc as Object
        dim mySheet as Object
        dim cell as Object
        '------------------------------
        ' Variable den Wert 0 zuweisen
        gesamtVerkaufsbetrag = 0
        gesamtprovision = 0

        myDoc = thisComponent
        mySheet = myDoc.sheets(0)
        '--------------------------------------
        'zeilenzähler einen Wert zuweisen
        zeilenzaehler = ERSTE_ZEILE_MIT_VERKAUFSBETRAG

        cell=mySheet.getCellByPosition(PROVISIONSSPALTE, zeilenzaehler)
        '-------------------------------------------------------------
        ' solange keine leere Zelle gefunden wird wiederhole die Befehle
        do while cell.Type<>com.sun.star.table.CellContentType.EMPTY
                gesamtprovision=gesamtprovision + cell.Value
                cell=mySheet.getCellByPosition(VERKAUFSBETRAGSPALTE, zeilenzaehler)
                gesamtVerkaufsbetrag=gesamtVerkaufsbetrag + cell.Value
                zeilenzaehler=zeilenzaehler+1
                cell=mySheet.getCellByPosition(PROVISIONSSPALTE, zeilenzaehler)
        loop

        anzahlVerkaeufe=zeilenzaehler-ERSTE_ZEILE_MIT_VERKAUFSBETRAG
        if anzahlVerkaeufe=0 then
                MsgBox("Noch kein Verkauf")
                exit sub
        end if

        durchschnittsprovision=gesamtprovision/anzahlVerkaeufe

        mySheet = myDoc.Sheets(1)
        cell=mySheet.getCellByPosition(0,0)
        cell.String="Anzahl Verkäufe"
        cell=mySheet.getCellByPosition(1,0)
        cell.String="Gesamt Verkaufsbetrag"
        cell=mySheet.getCellByPosition(2,0)
        cell.String="Gesamtprovision"
        cell=mySheet.getCellByPosition(3,0)
        cell.String="Durchschnittliche Provision"

        cell=mySheet.getCellByPosition(0,1)
        cell.Value=anzahlVerkaeufe
        cell=mySheet.getCellByPosition(1,1)
        cell.Value=gesamtVerkaufsbetrag
        cell=mySheet.getCellByPosition(2,1)
        cell.Value=gesamtprovision
        cell=mySheet.getCellByPosition(3,1)
        cell.Value=durchschnittsprovision
end sub
```

Es handelt sich hier überraschenderweise um ein recht kleines Programm. Zunächst werden Variablen und Konstanten deklariert. Die Konstanten

- *VERKAUFSBETRAGSPALTE*
- *PROVISIONSSPALTE*
- *ERSTE_ZEILE_MIT_VERKAUFSBETRAG*

haben ihre jeweiligen Werte, weil ja in OpenOffice, wie bereits in Beispiel 6.2 behandelt, die Zeilen- und Spaltenzählung bei Null beginnt. In den Zeilen

```
gesamtVerkaufsbetrag = 0
gesamtprovision = 0
```

werden die Variablen, auf denen 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. Die Anzahl der Verkäufe wird auch um 1 hochgezählt. Mit den Zeilen

```
myDoc = thisComponent
mySheet = myDoc.sheets(0)
zeilenzaehler = ERSTE_ZEILE_MIT_VERKAUFSBETRAG
cell=mySheet.getCellByPosition(PROVISIONSSPALTE, anzahlVerkaeufe+1)
```

wird zunächst das Blatt der Tabellenkalkulation ansprechbar gemacht, dann wird die Variable *zeilenzaehler* mit der Zeile, in der die Verkaufsbeträge beginnen, initialisiert. Danach speichern wir die Zelle C4 auf der Variablen *cell*. Dann beginnt die . 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 cell.Type<>com.sun.star.table.CellContentType.EMPTY
```

Die Bedingung ist die Art und Weise, mit der man in OpenOffice feststellt, ob eine Zelle leer ist oder nicht. Wenn Sie eine Zelle auf der Variablen *cell* abgespeichert haben, dann ergibt der Ausdruck

```
cell.Type<>com.sun.star.table.CellContentType.EMPTY
```

true, wenn die Zelle nicht leer ist und *false*, wenn sie leer ist. Die *while*-Anweisung führt die zur Schleife gehörenden Anweisungen durch, wenn die auf *do while* folgende Bedingung *true* ergibt. In Abb. 7.2 ist dies der Fall, die Zelle C4 ist gefüllt. In der Schleife wird zunächst die Gesamtprovision durch den Wert der aktuellen Zelle (zur Zeit gerade C4) erhöht:

```
gesamtprovision=gesamtprovision + cell.Value
```

Danach erhöhen wir durch die Zeilen

```
cell=mySheet.getCellByPosition(VERKAUFSBETRAGSPALTE, zeilenzaehler)
gesamtVerkaufsbetrag=gesamtVerkaufsbetrag + cell.Value
```

den Gesamtverkaufsbetrag durch den Verkaufsbetrag der aktuellen Zeile. Wir erhöhen *zeilenzaehler* um eins.

```
zeilenzaehler=zeilenzaehler+1
```

Durch

```
cell=mySheet.getCellByPosition(PROVISIONSSPALTE, zeilenzaehler)
```

gehen wir eine Zeile nach unten. Die Anweisung *loop* bewegt VBA nun, zur Zeile mit der *do while*-Anweisung zurückzukehren. Jedesmal, wenn VBA zur *do while*-Anweisung zurückzukehrt, wird die auf *do while* folgende Bedingung ausgewertet. Ergibt die Auswertung der Bedingung *true* werden die Anweisungen der Schleife erneut durchlaufen, ansonsten setzt VBA mit der auf *loop* folgenden 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 6.2 beschrieben, in das zweite Tabellenblatt geschrieben[2].

Realisierung des Beispiels in Excel

Beispiel 7.2 *Bisheriger Verkaufsbetrag, Gesamt- und durchschnittliche Provision in Excel*

```
        Private Sub berechneGesamtUndDurchschnittsprovision_Click()
                Const VERKAUFSBETRAGSPALTE As Integer = 2
                Const PROVISIONSSPALTE As Integer = 3
                Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Integer = 4
                Dim gesamtprovision As Double
                Dim gesamtVerkaufsbetrag As Double
                Dim zeilenzaehler As Integer
                Dim anzahlVerkaeufe As Integer
                Dim durchschnittsprovision As Double

                gesamtprovision = 0
                gesamtVerkaufsbetrag = 0

                zeilenzaehler = ERSTE_ZEILE_MIT_VERKAUFSBETRAG
                Do While Not IsEmpty(Cells(zeilenzaehler, PROVISIONSSPALTE))
                        gesamtprovision = gesamtprovision + Cells(zeilenzaehler, ➥
                            PROVISIONSSPALTE)
                        gesamtVerkaufsbetrag = gesamtVerkaufsbetrag + 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(2).Cells(1, 1) = "Anzahl Verkäufe"
                Sheets(2).Cells(1, 2) = "Gesamtverkaufsbetrag"
                Sheets(2).Cells(1, 3) = "Gesamtprovision"
                Sheets(2).Cells(1, 4) = "Durchschnittliche Provision"

                Sheets(2).Cells(2, 1) = anzahlVerkaeufe
                Sheets(2).Cells(2, 2) = gesamtVerkaufsbetrag
                Sheets(2).Cells(2, 3) = gesamtprovision
                Sheets(2).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
```

[2] OpenOffice weiter auf Seite 65.

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(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. 7.2 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 + Cells(zeilenzaehler, PROVISIONSSPALTE)
gesamtVerkaufsbetrag = gesamtVerkaufsbetrag + 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 6.3 beschrieben, in das zweite Tabellenblatt geschrieben.

7.1.2 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 (abweisende Schleife). 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 (Endlosschleife). Die Anweisungen der Schleife werden unendlich oft wiederholt (dies ist selten vom Programmierer so beabsichtigt).
Der Wert von *logischer Ausdruck* muss also innerhalb der Schleife geändert werden (wenn man keine Endlosschleife zu programmieren beabsichtigt).

7.1.3 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 hat es gegeben) 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. 7.6 dargestellt, das Ergebnis in Abb. 7.7.

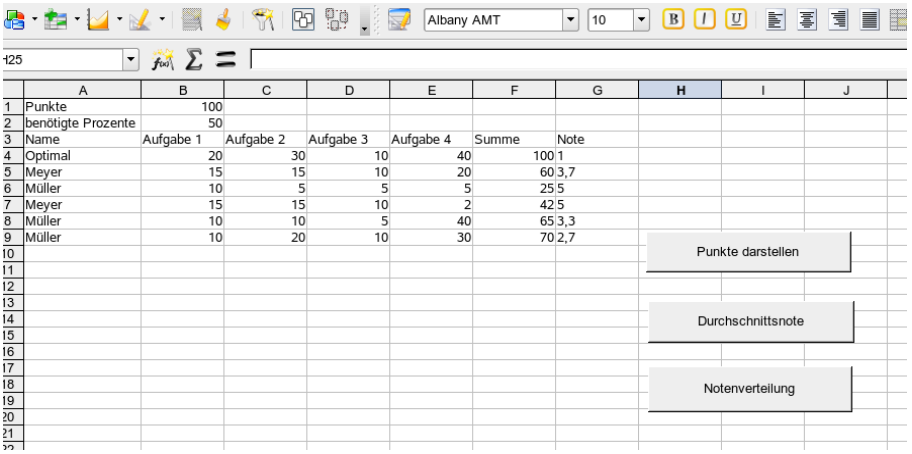


Abbildung 7.6
Statistiken für die Klausurauswertung: Benutzerinterface

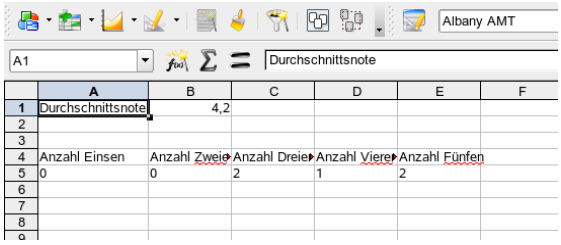


Abbildung 7.7
Statistiken für die Klausurauswertung: das Ergebnis

Hier haben wir die bereits in Kap. 7.1.1 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. Auch hier müssen wir direkt mit Zellinhalten interagieren, d.h. auch hier unterscheidet sich die OpenOffice- von der Excel-Lösung.

Realisierung des Beispiels in OpenOffice

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

Beispiel 7.3 *Durchschnittsnote in OpenOffice* [3]

```
sub berechneDurchschnittsnote
        const OPTIMALZEILE=3
        dim spaltenzaehler as integer
        dim aktuelleZeile as integer
        dim letzteBesetzteSpalte as integer
        dim anzahlTeilnehmer as integer
        dim notenSumme as double
        dim durchschnittsnote as double
        dim myDoc as Object
        dim mySheet as Object
        dim cell as Object

        spaltenzaehler=0
        anzahlTeilnehmer=0
        notenSumme=0

        myDoc = thisComponent
        mySheet = myDoc.sheets(0)
        cell=mySheet.getCellByPosition(spaltenzaehler, OPTIMALZEILE)

        do while cell.Type<>com.sun.star.table.CellContentType.EMPTY
                spaltenzaehler=spaltenzaehler+1
                cell=mySheet.getCellByPosition(spaltenzaehler, OPTIMALZEILE)
        loop

        letzteBesetzteSpalte=spaltenzaehler-1

        aktuelleZeile=OPTIMALZEILE+1
        cell=mySheet.getCellByPosition(letzteBesetzteSpalte, aktuelleZeile)

        do while cell.Type<>com.sun.star.table.CellContentType.EMPTY
                anzahlTeilnehmer=anzahlteilnehmer+1
                notenSumme=notenSumme+cdbl(cell.String)
                aktuelleZeile=aktuelleZeile+1
                cell=mySheet.getCellByPosition(letzteBesetzteSpalte, aktuelleZeile)
        loop

        durchschnittsnote=notenSumme/anzahlTeilnehmer

        mySheet = myDoc.Sheets(2)
        cell=mySheet.getCellByPosition(0,0)
        cell.String="Durchschnittsnote"
        cell=mySheet.getCellByPosition(1,0)
        cell.Value=durchschnittsnote

end sub
```

[3] Excel weiter auf Seite 69

Auch hier werden zunächst die für die Prozedur benötigten Variablen und Konstanten deklariert. Wegen des Beginns der Zeilenzählung bei 0 von OpenOffice hat die Konstante *OPTIMALZEILE* den Wert 3. *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 speichert unser Programm A4 auf der Variablen *cell* (0te Spalte, 3te Zeile). Dann wird geprüft, ob diese Zeile leer ist, genau wie im Provisionsbeispiel. Diese Zelle ist nicht leer, also führt das Programm die Schleife aus. Hier wird *spaltenzaehler* um eins erhöht. Dann wird die nächste Zelle der nächsten Spalte von *OPTIMALZEILE* in die Zelle *cell* geladen. Die Schleife beginnt von Neuem. 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 7.2 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 7.4 *Notenverteilung in OpenOffice*

```
sub bestimmeNotenverteilung
        const OPTIMALZEILE=3
        dim spaltenzaehler as integer
        dim aktuelleZeile as integer
        dim letzteBesetzteSpalte as integer
        dim anzahlEinsen as integer
        dim anzahlZweien as integer
        dim anzahlDreien as integer
        dim anzahlVieren as integer
        dim anzahlFuenfen as integer
        dim myDoc as Object
        dim mySheet as Object
        dim cell as Object

        myDoc = thisComponent
        mySheet = myDoc.sheets(0)
        spaltenzaehler=0
        anzahlEinsen=0
        anzahlZweien=0
        anzahlDreien=0
        anzahlVieren=0
        anzahlFuenfen=0

        cell=mySheet.getCellByPosition(spaltenzaehler, OPTIMALZEILE)
        do while cell.Type<>com.sun.star.table.CellContentType.EMPTY
                spaltenzaehler=spaltenzaehler+1
                cell=mySheet.getCellByPosition(spaltenzaehler, OPTIMALZEILE)
        loop
        letzteBesetzteSpalte=spaltenzaehler-1

        aktuelleZeile=OPTIMALZEILE+1
        cell=mySheet.getCellByPosition(letzteBesetzteSpalte, aktuelleZeile)
        do while cell.Type<>com.sun.star.table.CellContentType.EMPTY
                if (cell.String="1") or (cell.String="1,3") then
                        anzahlEinsen=anzahlEinsen+1
                elseif (cell.String="1,7") or (cell.String="2") or (cell.String="➥
                    2,3") then
                        anzahlZweien=anzahlZweien+1
                elseif (cell.String="2,7") or (cell.String="3") or (cell.String="➥
                    3,3") then
                        anzahlDreien=anzahlDreien+1
                elseif (cell.String="3,7") or (cell.String="4") then
                        anzahlVieren=anzahlVieren+1
                elseif cell.String="5" then
                        anzahlFuenfen=anzahlFuenfen+1
```

```
                    end if
                    aktuelleZeile=aktuelleZeile+1
                    cell=mySheet.getCellByPosition(letzteBesetzteSpalte, aktuelleZeile)
          loop

          mySheet = myDoc.Sheets(2)
          cell=mySheet.getCellByPosition(0,3)
          cell.String="Anzahl Einsen"
          cell=mySheet.getCellByPosition(1,3)
          cell.String="Anzahl Zweien"
          cell=mySheet.getCellByPosition(2,3)
          cell.String="Anzahl Dreien"
          cell=mySheet.getCellByPosition(3,3)
          cell.String="Anzahl Vieren"
          cell=mySheet.getCellByPosition(4,3)
          cell.String="Anzahl Fünfen"
          cell=mySheet.getCellByPosition(0,4)
          cell.String=anzahlEinsen
          cell=mySheet.getCellByPosition(1,4)
          cell.String=anzahlZweien
          cell=mySheet.getCellByPosition(2,4)
          cell.String=anzahlDreien
          cell=mySheet.getCellByPosition(3,4)
          cell.String=anzahlVieren
          cell=mySheet.getCellByPosition(4,4)
          cell.String=anzahlFuenfen
end sub
```

Dieses Programm ist zwar etwas länger, enthält aber keine neuen Konzepte: In einer ersten *do while*-Schleife wird, wie in Beispiel 7.3 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 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[4].

Realisierung des Beispiels in Excel

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

Beispiel 7.5 *Durchschnittsnote in Excel*

```
Sub berechneDurchschnittsnote_Click()
Const OPTIMALZEILE As Integer = 4
Dim spaltenzaehler As Integer
Dim aktuelleZeile As Integer
Dim letzteBesetzteSpalte As Integer
Dim anzahlTeilnehmer As Integer
Dim notenSumme As Double
Dim durchschnittsnote As Double

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

aktuelleZeile = OPTIMALZEILE + 1
Do While Not IsEmpty(Cells(aktuelleZeile, letzteBesetzteSpalte))
          anzahlTeilnehmer = anzahlTeilnehmer + 1
          notenSumme = notenSumme + Cells(aktuelleZeile, letzteBesetzteSpalte)
```

[4]OpenOffice weiter auf Seite 71

```
            aktuelleZeile = aktuelleZeile + 1
        Loop
        durchschnittsnote = notenSumme / anzahlTeilnehmer

        Sheets(3).Cells(1, 1) = "Durchschnittsnote"
        Sheets(3).Cells(1, 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 7.1 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 7.6 *Notenverteilung in Excel*

```
        Sub bestimmeNotenverteilung_Click()
        Const OPTIMALZEILE As Integer = 4
        Dim spaltenzaehler As Integer
        Dim aktuelleZeile As Integer
        Dim letzteBesetzteSpalte As Integer
        Dim anzahlEinsen As Integer
        Dim anzahlZweien As Integer
        Dim anzahlDreien As Integer
        Dim anzahlVieren As Integer
        Dim anzahlFuenfen As Integer

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

        Do While Not IsEmpty(Cells(OPTIMALZEILE, spaltenzaehler))
            spaltenzaehler = spaltenzaehler + 1
        Loop
        letzteBesetzteSpalte = spaltenzaehler - 1

        aktuelleZeile = OPTIMALZEILE + 1
        Do While Not IsEmpty(Cells(aktuelleZeile, letzteBesetzteSpalte))
            If (Cells(aktuelleZeile, letzteBesetzteSpalte) = "1") Or _
                                (Cells(aktuelleZeile, letzteBesetzteSpalte) = "➥
                                    1,3") Then
            anzahlEinsen = anzahlEinsen + 1
            ElseIf (Cells(aktuelleZeile, letzteBesetzteSpalte) = "1,7") Or _
                                (Cells(aktuelleZeile, letzteBesetzteSpalte) = "➥
                                    2") Or _
                                (Cells(aktuelleZeile, letzteBesetzteSpalte) = "➥
                                    2,3") Then
            anzahlZweien = anzahlZweien + 1
            ElseIf (Cells(aktuelleZeile, letzteBesetzteSpalte) = "2,7") Or _
```

```
                                                (Cells(aktuelleZeile, letzteBesetzteSpalte) = "➥
                                                    3") Or _
                                                (Cells(aktuelleZeile, letzteBesetzteSpalte) = "➥
                                                    3,3") Then
                    anzahlDreien = anzahlDreien + 1
                    ElseIf (Cells(aktuelleZeile, letzteBesetzteSpalte) = "3,7") Or _
                                                (Cells(aktuelleZeile, letzteBesetzteSpalte) = "➥
                                                    4") Then
                    anzahlVieren = anzahlVieren + 1
                    ElseIf Cells(aktuelleZeile, letzteBesetzteSpalte) = "5" Then
                    anzahlFuenfen = anzahlFuenfen + 1
                    End If
                    aktuelleZeile = aktuelleZeile + 1
          Loop

          Sheets(3).Cells(1, 1) = "Anzahl Einsen"
          Sheets(3).Cells(1, 2) = "Anzahl Zweien"
          Sheets(3).Cells(1, 3) = "Anzahl Dreien"
          Sheets(3).Cells(1, 4) = "Anzahl Vieren"
          Sheets(3).Cells(1, 5) = "Anzahl Fünfen"
          Sheets(3).Cells(2, 1) = anzahlEinsen
          Sheets(3).Cells(2, 2) = anzahlZweien
          Sheets(3).Cells(2, 3) = anzahlDreien
          Sheets(3).Cells(2, 4) = anzahlVieren
          Sheets(3).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 7.5 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.

7.2 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[5]:

Beispiel 7.7 *Fakultäten berechnen*

```
function fakultaetBerechnen(bisZu as Integer) As Integer
        dim fakultaet As Integer
        dim i As Integer
        fakultaet = 1
        for i = 1 to bisZu
                fakultaet = fakultaet * i
        next i
        fakultaetBerechnen = 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 *Integer*, sie wird also einen *Integer* 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 drei eingegeben hat, ist dies der Fall. Daher werden nun die Anweisungen der Schleife abgearbeitet. Dies sind alle Anweisungen, die sich zwischen *for i = 1 to 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
```

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:

[5] 3 Fakultät schreibt man in der Mathematik 3! und es gilt: 3!=1*2*3.

```
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) 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:

```
fakultaetBerechnen = fakultaet
```

Kapitel 8

Interne Nutzung eigener Funktionen oder Prozeduren

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 könnte ja hingehen und mein 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

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

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 sowohl für OpenOffice als auch für Excel zahllose von anderen geschriebene Funktionen, die man selbst nutzen kann.

8.1 Interne Nutzung eigener Funktionen

Beginnen wollen wir allerdings mit einem einfachen Beispiel. Bei der Darstellung der Gesamtprovision und der durchschnittlichen Provision soll neben den Provisionen auch noch die Umsatzsteuer und der Bruttobetrag ausgewiesen werden. Dies ist ein typisches Beispiel für die sinnvolle Nutzung einer Funktion. Wir können den Umsatzsteuersatz, wie bereits in Beispiel 4.5 beschrieben, zentral in einer Funktion vorhalten und haben ihn daher nicht in vielen verschiedenen Programmen verteilt. Die Implementierung dieser Funktion kennen wir bereits aus Beispiel 4.5, wir stellen sie erneut dar:

Beispiel 8.1 *Berechnung der Umsatzsteuer aus dem Nettobetrag*

```
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[2]. Kommen wir nun zur Berechnung und Darstellung der Umsatzsteuer.
Wir müssen die Funktion *berechneGesamtUndDurchschnittsprovision* aus Beispiel 7.1 bzw. Beispiel 7.2 ändern. Daher gibt es jetzt wieder getrennte Lösungen für OpenOffice und Excel.

8.1.1 Umsatzsteuerberechnung für das Provisionsbeispiel in OpenOffice

Beispiel 8.2 *Integration der Umsatzsteuer in OpenOffice*[3]

```
sub berechneBisherigenVerkaufsbetragUmsatzsteuerGesamtUndDurchschnittsprovision()
        const VERKAUFSBETRAGSPALTE as Integer = 1
        const PROVISIONSSPALTE as Integer = 2
        const ERSTE_ZEILE_MIT_VERKAUFSBETRAG as Integer = 3
        dim gesamtprovision as double
        dim gesamtVerkaufsbetrag as double
        dim zeilenzaehler as Integer
        dim anzahlVerkaeufe as Integer
        dim durchschnittsprovision as Double
        dim umsatzsteuerGesamtprovision as double
        dim bruttoGesamtprovision as double
        dim bruttoDurchschnittsprovision as double
        dim umsatzsteuerDurchschnittsprovision as double
        dim myDoc as Object
        dim mySheet as Object
        dim cell as Object

        zeilenzaehler = ERSTE_ZEILE_MIT_VERKAUFSBETRAG
        gesamtVerkaufsbetrag = 0
        gesamtprovision = 0

        myDoc = thisComponent
        mySheet = myDoc.sheets(0)

        cell=mySheet.getCellByPosition(PROVISIONSSPALTE, zeilenzaehler)
        do while cell.Type<>com.sun.star.table.CellContentType.EMPTY
```

[2] Toller Satz, oder?
[3] Excel weiter auf Seite 78

```
                    gesamtprovision=gesamtprovision + cell.Value
                    cell=mySheet.getCellByPosition(VERKAUFSBETRAGSPALTE, zeilenzaehler)
                    gesamtVerkaufsbetrag=gesamtVerkaufsbetrag + cell.Value
                    zeilenzaehler=zeilenzaehler+1
                    cell=mySheet.getCellByPosition(PROVISIONSSPALTE, zeilenzaehler)
          loop

          anzahlVerkaeufe=zeilenzaehler-ERSTE_ZEILE_MIT_VERKAUFSBETRAG
          if anzahlVerkaeufe=0 then
                    MsgBox("Noch kein Verkauf")
                    exit sub
          end if

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

          mySheet = myDoc.Sheets(1)
          cell=mySheet.getCellByPosition(0,0)
          cell.String="Anzahl Verkäufe"
          cell=mySheet.getCellByPosition(1,0)
          cell.String="Gesamtverkaufsbetrag"
          cell=mySheet.getCellByPosition(2,0)
          cell.String="Gesamtprovision"
          cell=mySheet.getCellByPosition(3,0)
          cell.String="Umsatzsteuer"
          cell=mySheet.getCellByPosition(4,0)
          cell.String="Bruttoprovision"
          cell=mySheet.getCellByPosition(5,0)
          cell.String="Durchschnittliche Provision (Netto)"
          cell=mySheet.getCellByPosition(6,0)
          cell.String="Umsatzsteuer"
          cell=mySheet.getCellByPosition(7,0)
          cell.String="Durchschnittliche Provision (Brutto)"

          cell=mySheet.getCellByPosition(0,1)
          cell.Value=anzahlVerkaeufe
          cell=mySheet.getCellByPosition(1,1)
          cell.Value=gesamtVerkaufsbetrag
          cell=mySheet.getCellByPosition(2,1)
          cell.Value=gesamtprovision
          cell=mySheet.getCellByPosition(3,1)
          cell.Value=umsatzsteuerGesamtprovision
          cell=mySheet.getCellByPosition(4,1)
          cell.Value=bruttoGesamtprovision
          cell=mySheet.getCellByPosition(5,1)
          cell.Value=durchschnittsprovision
          cell=mySheet.getCellByPosition(6,1)
          cell.Value=umsatzsteuerDurchschnittsprovision
          cell=mySheet.getCellByPosition(7,1)
          cell.Value=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. 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. Das der Name der Variablen beim Aufruf der Funktion *gesamtprovision*, bzw. *durchschnittsprovision* nicht mit dem Na-

men der Variablen bei der Deklaration der Funktion *nettobetrag* übereinstimmt, ist auch nicht wirklich überraschend. Wenn wir VBA interne Funktionen aufrufen, dann wissen wir auch nicht, wie die Microsoft- oder OpenOffice-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.

8.1.2 Umsatzsteuerberechnung für das Provisionsbeispiel in Excel

Beispiel 8.3 *berechneBisherigenVerkaufsbetragUmsatzsteuerGesamtUndDurchschnittsprovision Integration der Umsatzsteuer in Excel* [4]

```
Private Sub berechneGesamtUndDurchschnittsprovision_Click()
        Const VERKAUFSBETRAGSPALTE As Integer = 2
        Const PROVISIONSSPALTE As Integer = 3
        Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Integer = 4
        Dim gesamtVerkaufsbetrag As Double
        Dim zeilenzaehler As Integer
        Dim anzahlVerkaeufe As Integer
        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(Cells(zeilenzaehler, PROVISIONSSPALTE))
                gesamtprovision = gesamtprovision + Cells(zeilenzaehler, ➥
                    PROVISIONSSPALTE)
                gesamtVerkaufsbetrag = gesamtVerkaufsbetrag + 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(2).Cells(1, 1) = "Anzahl Verkäufe"
        Sheets(2).Cells(1, 2) = "Gesamtverkäufe"
        Sheets(2).Cells(1, 3) = "Gesamtprovision"
        Sheets(2).Cells(1, 4) = "Umsatzsteuer"
        Sheets(2).Cells(1, 5) = "Bruttoprovision"
        Sheets(2).Cells(1, 6) = "Durchschnittliche Provision"
        Sheets(2).Cells(1, 7) = "Umsatzsteuer"
        Sheets(2).Cells(1, 8) = "Brutto durchschnittliche Provision"

        Sheets(2).Cells(2, 1) = anzahlVerkaeufe
```

[4] OpenOffice weiter auf Seite 79

```
                    Sheets(2).Cells(2, 2) = gesamtVerkaufsbetrag
                    Sheets(2).Cells(2, 3) = gesamtprovision
                    Sheets(2).Cells(2, 4) = umsatzsteuerGesamtprovision
                    Sheets(2).Cells(2, 5) = bruttoGesamtprovision
                    Sheets(2).Cells(2, 6) = durchschnittsprovision
                    Sheets(2).Cells(2, 7) = umsatzsteuerDurchschnittsprovision
                    Sheets(2).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. 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. 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- oder OpenOffice-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 berechneBisherigenVerkaufsbetragUmsatzsteuerGesamtUndDurchschnittsprovision 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.

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

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. 7 sehr einfach:

Beispiel 8.4 *Ermittlung der letzten belegten Zeile einer Spalte in OpenOffice*

```
          function ermittleLetzteBesetzteZeileInSpalte(spaltennummer as integer, _
                              startzeile as integer, tabellennummer as integer) as
Integer
                    dim myDoc as Object
                    dim mySheet as Object
                    dim cell as Object
                    dim zeilenzaehler as integer
                    myDoc = thisComponent
                    mySheet = myDoc.sheets(tabellennummer)
                    zeilenzaehler=startzeile
                    cell=mySheet.getCellByPosition(spaltennummer, zeilenzaehler)
                    do while cell.Type<>com.sun.star.table.CellContentType.EMPTY
                              zeilenzaehler=zeilenzaehler+1
                              cell=mySheet.getCellByPosition(spaltennummer, zeilenzaehler)
                    loop
                    ermittleLetzteBesetzteZeileInSpalte=zeilenzaehler-1
          end function
```

Durch

```
mySheet = myDoc.sheets(tabellennummer)
```

wird das Tabellenblatt festgelegt, in dem die letzte besetzte Zeile gefunden werden soll.
Bauen wir diese Funktion nun in unsere Berechnungsfunktion ein. Wir erklären nur den Berechnungsteil, die Definition der Variablen und Konstanten, sowie die Ausgabe in das Tabellenblatt der Arbeitsmappe entspricht Beispiel 8.2.

Beispiel 8.5 *Provisionsbeispiel mit Funktion zur Ermittlung der letzten besetzten Zeile*

```
sub berechneBisherigenVerkaufsbetragUmsatzsteuerGesamtUndDurchschnittsprovision()
        const VERKAUFSBETRAGSPALTE as Integer = 1
        const PROVISIONSSPALTE as Integer = 2
        const ERSTE_ZEILE_MIT_VERKAUFSBETRAG as Integer = 3

        dim gesamtprovision as double
        dim gesamtVerkaufsbetrag as double
        dim letzteBesetzteZeile as Integer
        dim i as Integer
        dim anzahlVerkaeufe as Integer
        dim durchschnittsprovision as Double
        dim umsatzsteuerGesamtprovision as double
        dim bruttoGesamtprovision as double
        dim bruttoDurchschnittsprovision as double
        dim umsatzsteuerDurchschnittsprovision as double
        dim myDoc as Object
        dim mySheet as Object
        dim cell as Object

        gesamtVerkaufsbetrag = 0
        gesamtprovision = 0

        myDoc = thisComponent
        mySheet = myDoc.sheets(0)

        letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(PROVISIONSSPALTE, _
                                ERSTE_ZEILE_MIT_VERKAUFSBETRAG, 0)
        anzahlVerkaeufe=letzteBesetzteZeile-ERSTE_ZEILE_MIT_VERKAUFSBETRAG + 1
        if anzahlVerkaeufe=0 then
                MsgBox("Noch kein Verkauf")
                exit sub
        end if
        for i= ERSTE_ZEILE_MIT_VERKAUFSBETRAG to letzteBesetzteZeile
                cell=mySheet.getCellByPosition(PROVISIONSSPALTE, i)
                gesamtprovision=gesamtprovision + cell.Value
                cell=mySheet.getCellByPosition(VERKAUFSBETRAGSPALTE, i)
                gesamtVerkaufsbetrag=gesamtVerkaufsbetrag + cell.Value
        next i

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

        mySheet = myDoc.Sheets(1)
        cell=mySheet.getCellByPosition(0,0)
        cell.String="Anzahl Verkäufe"
        cell=mySheet.getCellByPosition(1,0)
        cell.String="Gesamtverkaufsbetrag"
        cell=mySheet.getCellByPosition(2,0)
        cell.String="Gesamtprovision"
        cell=mySheet.getCellByPosition(3,0)
        cell.String="Umsatzsteuer"
        cell=mySheet.getCellByPosition(4,0)
```

```
                    cell.String="Bruttoprovision"
                    cell=mySheet.getCellByPosition(5,0)
                    cell.String="Durchschnittliche Provision (Netto)"
                    cell=mySheet.getCellByPosition(6,0)
                    cell.String="Umsatzsteuer"
                    cell=mySheet.getCellByPosition(7,0)
                    cell.String="Durchschnittliche Provision (Brutto)"

                    cell=mySheet.getCellByPosition(0,1)
                    cell.Value=anzahlVerkaeufe
                    cell=mySheet.getCellByPosition(1,1)
                    cell.Value=gesamtVerkaufsbetrag
                    cell=mySheet.getCellByPosition(2,1)
                    cell.Value=gesamtprovision
                    cell=mySheet.getCellByPosition(3,1)
                    cell.Value=umsatzsteuerGesamtprovision
                    cell=mySheet.getCellByPosition(4,1)
                    cell.Value=bruttoGesamtprovision
                    cell=mySheet.getCellByPosition(5,1)
                    cell.Value=durchschnittsprovision
                    cell=mySheet.getCellByPosition(6,1)
                    cell.Value=umsatzsteuerDurchschnittsprovision
                    cell=mySheet.getCellByPosition(7,1)
                    cell.Value=bruttoDurchschnittsprovision
          end sub
```

Ganz so, wie in Beispiel 8.2 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. TABELLE_MIT_VERKAUFSBETRAEGEN)
```

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[5].

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

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. 7 sehr einfach:

Beispiel 8.6 *Ermittlung der letzten belegten Zeile einer Spalte in Excel*

```
          Function ermittleLetzteBesetzteZeileInSpalte(spaltennummer As Integer, _
                        startzeile As Integer, tabellennummer as Integer) As Integer
                    Dim zeilenzaehler As Integer
                    zeilenzaehler = startzeile
                    Do While Not IsEmpty(Sheets(tabellennummer).Cells(zeilenzaehler, ➥
                        spaltennummer))
                        zeilenzaehler = zeilenzaehler + 1
                    Loop
                    ermittleLetzteBesetzteZeileInSpalte = zeilenzaehler - 1
          End Function
```

[5] OpenOffice weiter auf Seite 83

Durch

```
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 8.3.

Beispiel 8.7 *Provisionsbeispiel mit Funktion zur Ermittlung der letzten besetzten Zeile*

```
Private Sub berechneGesamtUndDurchschnittsprovision_Click()
        Const VERKAUFSBETRAGSPALTE As Integer = 2
        Const PROVISIONSSPALTE As Integer = 3
        Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Integer = 4

        Dim gesamtprovision As Double
        Dim gesamtVerkaufsbetrag As Double
        Dim anzahlVerkaeufe As Integer
        Dim durchschnittsprovision As Double
        Dim nettoGesamtprovision As Double
        Dim umsatzsteuerGesamtprovision As Double
        Dim umsatzsteuerDurchschnittsprovision As Double
        Dim nettoDurchschnittsprovision As Double
        Dim letzteBesetzteZeile As Integer
        Dim i As Integer

        gesamtprovision = 0
        gesamtVerkaufsbetrag = 0

        letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(PROVISIONSSPALTE, _
                                                ERSTE_ZEILE_MIT_VERKAUFSBETRAG, 1)
        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 + Cells(i, ➥
                    VERKAUFSBETRAGSPALTE)
                gesamtprovision = gesamtprovision + Cells(i, PROVISIONSSPALTE)
        Next i

        durchschnittsprovision = gesamtprovision / anzahlVerkaeufe
        umsatzsteuerGesamtprovision = berechneUmsatzsteuerAusBrutto(gesamtprovision)
        nettoGesamtprovision = gesamtprovision - umsatzsteuerGesamtprovision
        umsatzsteuerDurchschnittsprovision = berechneUmsatzsteuerAusBrutto(➥
            durchschnittsprovision)
        nettoDurchschnittsprovision = durchschnittsprovision - ➥
            umsatzsteuerDurchschnittsprovision

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

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

Ganz so, wie in Beispiel 8.3 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, 1)
```

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.

8.1.5 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[6].

8.1.6 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 in den nun folgenden tabellenkalkulationsspezifischen Kapiteln die Implementierung der Funktion *ermittleLetzteBesetzteSpalteInZeile*, die Aufrufe der beiden Funktionen, sowie die Änderung von der *while*-zur *for*-Schleife unkommentiert da.

Anpassungen in OpenOffice

[7] Zunächst die Funktion *ermittleLetzteBesetzteSpalteInZeile*:

Beispiel 8.8 *Ermittlung der letzten belegten Spalte einer Zeile in OpenOffice*

```
function ermittleLetzteBesetzteSpalteInZeile(zeilennummer as integer, _
                    startspalte as Integer, tabellennummer as Integer) as Integer
          dim myDoc as Object
          dim mySheet as Object
          dim cell as Object
          dim spaltenzaehler as integer

          myDoc = thisComponent
          mySheet = myDoc.sheets(tabellennummer)
          spaltenzaehler=     startspalte
          cell=mySheet.getCellByPosition(spaltenzaehler, zeilennummer)

          do while cell.Type<>com.sun.star.table.CellContentType.EMPTY
                    spaltenzaehler=spaltenzaehler+1
                    cell=mySheet.getCellByPosition(spaltenzaehler, zeilennummer)
          loop
```

[6] In Wirklichkeit hätten wir die Funktion zur Ermittlung der letzten besetzten Zeile gar nicht selber schreiben müssen, denn sowohl für OpenOffice als auch für Excel existieren im Internet Sammlungen von freien Makros, darunter selbstverständlich auch so grundlegende, wie das hier vorgestellte. Googeln Sie einfach „Makro OpenOffice“, bzw. „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.

[7] Excel weiter auf Seite 84

```
                    ermittleLetzteBesetzteSpalteInZeile=spaltenzaehler-1
          end function
```

Nun der Einsatz dieser Funktion, der Funktion *ermittleLetzteBesetzteZeileInSpalte* sowie der *for*-Schleife in den Ereignisprozeduren *berechneDurchschnittsnote* und *bestimmeNotenverteilung*:

Beispiel 8.9 *Änderungen in der Ereignisprozedur berechneDurchschnittsnote*

```
letzteBesetzteSpalte=ermittleLetzteBesetzteSpalteInZeile(OPTIMALZEILE, 2, 0)
letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(0, OPTIMALZEILE+1, 0)
anzahlTeilnehmer=letzteBesetzteZeile-OPTIMALZEILE

for i=OPTIMALZEILE+1 to letzteBesetzteZeile
          cell=mySheet.getCellByPosition(letzteBesetzteSpalte, i)
          notenSumme=notenSumme+cdbl(cell.String)
next i
```

Beispiel 8.10 *Änderungen in der Ereignisprozedur bestimmeNotenverteilung*

```
letzteBesetzteSpalte=ermittleLetzteBesetzteSpalteInZeile(OPTIMALZEILE, 2, 0)
letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(0, OPTIMALZEILE+1, 0)

for i=OPTIMALZEILE+1 to letzteBesetzteZeile
          cell=mySheet.getCellByPosition(letzteBesetzteSpalte, i)
          if (cell.String="1") or (cell.String="1,3") then
                    anzahlEinsen=anzahlEinsen+1
          elseif (cell.String="1,7") or (cell.String="2") or (cell.String="2,3") then
                    anzahlZweien=anzahlZweien+1
          elseif (cell.String="2,7") or (cell.String="3") or (cell.String="3,3") then
                    anzahlDreien=anzahlDreien+1
          elseif (cell.String="3,7") or (cell.String="4") then
                    anzahlVieren=anzahlVieren+1
          elseif cell.String="5" then
                    anzahlFuenfen=anzahlFuenfen+1
          end if
next i
```

Anpassungen in Excel

[8] Zunächst die Funktion *ermittleLetzteBesetzteSpalteInZeile*:

Beispiel 8.11 *Ermittlung der letzten belegten Spalte einer Zeile in Excel*

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

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

          ermittleLetzteBesetzteSpalteInZeile = spaltenzaehler - 1
End Function
```

Nun der Einsatz dieser Funktion, der Funktion *ermittleLetzteBesetzteZeileInSpalte* sowie der *for*-Schleife in den Ereignisprozeduren *berechneDurchschnittsnote* und *bestimmeNotenverteilung*:

[8] OpenOffice weiter auf Seite 85

Beispiel 8.12 *Änderungen in der Ereignisprozedur berechneDurchschnittsnote*

```
letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(OPTIMALZEILE, 3, 1)
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(1, OPTIMALZEILE + 1, 1)

anzahlTeilnehmer = letzteBesetzteZeile - OPTIMALZEILE
For i = OPTIMALZEILE + 1 To letzteBesetzteZeile
          notenSumme = notenSumme + Cells(i, letzteBesetzteSpalte)
Next i
```

Beispiel 8.13 *Änderungen in der Ereignisprozedur bestimmeNotenverteilung*

```
letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(OPTIMALZEILE, 3, 1)
letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(1, OPTIMALZEILE + 1, 1)

For i = OPTIMALZEILE + 1 To letzteBesetzteZeile
          If (Cells(i, letzteBesetzteSpalte) = "1") Or _
                    (Cells(i, letzteBesetzteSpalte) = "1,3") Then
                    anzahlEinsen = anzahlEinsen + 1
          ElseIf (Cells(i, letzteBesetzteSpalte) = "1,7") Or _
                    (Cells(i, letzteBesetzteSpalte) = "2") Or _
                    (Cells(i, letzteBesetzteSpalte) = "2,3") Then
                    anzahlZweien = anzahlZweien + 1
          ElseIf (Cells(i, letzteBesetzteSpalte) = "2,7") Or _
                    (Cells(i, letzteBesetzteSpalte) = "3") Or _
                    (Cells(i, letzteBesetzteSpalte) = "3,3") Then
                    anzahlDreien = anzahlDreien + 1
          ElseIf (Cells(i, letzteBesetzteSpalte) = "3,7") Or _
                    (Cells(i, letzteBesetzteSpalte) = "4") Then
                    anzahlVieren = anzahlVieren + 1
          ElseIf Cells(i, letzteBesetzteSpalte) = "5" Then
                    anzahlFuenfen = anzahlFuenfen + 1
          End If
Next
```

8.2 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. 6). Das macht anders auch wenig Sinn, denn Ereignisprozeduren sind ja, wie bereits gesagt, mit Schaltflächen verbunden und da ist die Aktion nur clicken[9]. 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:

8.2.1 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

[9] oder nicht clicken, aber dann passiert auch nichts.

Zelle bewegt, ein ziemlich obskurer und vielfach nichtssagender Text. Dies erschwert es, Fehler zu finden. Darüberhinaus wechselt man, wenn man den Debugger benutzt, ständig zwischen dem VBA-Editor und dem Arbeitsblatt hin und her. Wie Sie bereits in Kap.8.1 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. 6 haben Sie gelernt, dass man Prozeduren direkt im VBA-Editor ausführen kann. Es gibt neben den in in Kap. 6 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 8.1.

Beispiel 8.14 *Testprozedur für die Funktion Berechnung der Umsatzsteuer aus dem Nettobetrag*

```
sub teste()
        dim nettobetrag as double
        dim umwatzsteuer as double
        nettobetrag=129
        umsatzsteuer=berechneUmsatzsteuer(nettobetrag)
        MsgBox("Umsatzsteuer: " & umsatzsteuer)
end sub

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[10], 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.

8.2.2 Aufruf von Prozeduren aus Prozeduren oder Funktionen

Wir können Prozeduren, genau wie Funktionen, aus Prozeduren oder Funktionen aufrufen. Wir veranschaulichen dies an Beispiel 8.15. In diesem Beispiel ersetzen wir die Funktion aus Beispiel 8.1 durch eine Prozedur.

Beispiel 8.15 *Testprozedur für die Funktion Berechnung der Umsatzsteuer aus dem Nettobetrag*

```
option explicit
sub teste()
        dim nettobetrag as double
        dim umsatzsteuer as double
        nettobetrag=129
        call berechneUmsatzsteuer(nettobetrag, umsatzsteuer)
        MsgBox("Umsatzsteuer: " & umsatzsteuer)
end sub

sub berechneUmsatzsteuer(nettobetrag As Double, umsatzsteuer as Double)
        const UMSATZSTEUERSATZ as Double=0.19
        umsatzsteuer=nettobetrag*UMSATZSTEUERSATZ
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)
```

[10] hier nur einer, dem 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 8.15 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 8.15 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 8.16 und 8.17. Wir wollen hier den Bruttobetrag und die Umsatzsteuer berechnen und an das aufrufende Programm zurück geben. Zunächst die Realisierung mit einer Prozedur:

Beispiel 8.16 *Testprozedur für die Funktion Berechnung der Umsatzsteuer aus dem Nettobetrag*

```
sub teste()
        dim nettobetrag as double
        dim bruttobetrag as double
        dim umsatzsteuer as double
        nettobetrag=129
        call berechneUmsatzsteuerUndBrutto(nettobetrag, umsatzsteuer, bruttobetrag)
        MsgBox("Umsatzsteuer: " & umsatzsteuer & Chr$(13) & "Bruttobetrag: " & ➥
            bruttobetrag)
end sub

sub berechneUmsatzsteuerUndBrutto(nettobetrag As Double, _
                umsatzsteuer as Double, bruttobetrag as Double)
        const UMSATZSTEUERSATZ as Double=0.19
        umsatzsteuer=nettobetrag*UMSATZSTEUERSATZ
        bruttobetrag=nettobetrag+umsatzsteuer
end sub
```

Im Unterschied zu Beispiel 8.15 müssen wir hier 2 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 8.17 *Testprozedur für die Funktion Berechnung der Umsatzsteuer aus dem Nettobetrag*

```
sub teste()
        dim nettobetrag as double
        dim bruttobetrag as double
        dim umsatzsteuer as double
        nettobetrag=129
        umsatzsteuer=berechneUmsatzsteuerUndBrutto(nettobetrag, bruttobetrag)
        MsgBox ("Umsatzsteuer: " & umsatzsteuer & Chr$(13) & "Bruttobetrag: " & ➥
            bruttobetrag)
end sub

function berechneUmsatzsteuerUndBrutto(nettobetrag As Double, _
                        bruttobetrag as double) as Double
        const UMSATZSTEUERSATZ as Double=0.19
        dim umsatzsteuer as Double
        umsatzsteuer=nettobetrag*UMSATZSTEUERSATZ
        bruttobetrag = nettobetrag + umsatzsteuer
        berechneUmsatzsteuerUndBrutto=umsatzsteuer
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.

8.2.3 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 doch mehr als einen Rückgabewert haben. Beispiel 8.18 zeigt die Implementierung der Prozedur:

Beispiel 8.18 *Bestimmung der Notengrenzen in einer Prozedur*

```
sub ermittleNotengrenzen(maximalpunkte as Integer, benoetigteProzente as double, _
          grenze4 as Integer , grenze3_7 as Integer, _
          grenze3_3 as Integer, grenze3_0 as Integer, grenze2_7 as Integer, _
          grenze2_3 as Integer, grenze2_0 as Integer, grenze1_7 as Integer, _
          grenze1_3 as Integer, grenze1_0 as Integer)

          dim punkteZweiZwischenNoten as integer
          dim punkteDreiZwischenNoten as integer
          dim punkteProNote as integer
          dim spanne as integer
          dim benoetigtePunkte as integer

          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 8.19:

Beispiel 8.19 *Aufruf der Prozedur zur Bestimmung der Notengrenzen*

```
        function noteIfElseIF(maximalpunkte As Integer, benoetigteProzente As Double, _
                                        erreichtePunkte As Integer) As String

                dim grenze4_0 as Integer
                dim grenze3_7 as Integer
                dim grenze3_3 as Integer
                dim grenze3_0 as Integer
                dim grenze2_7 as Integer
                dim grenze2_3 as Integer
                dim grenze2_0 as Integer
                dim grenze1_7 as Integer
                dim grenze1_3 as Integer
                dim grenze1_0 as Integer

                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 erfolgt völlig analog.

Kapitel 9

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 9.1:

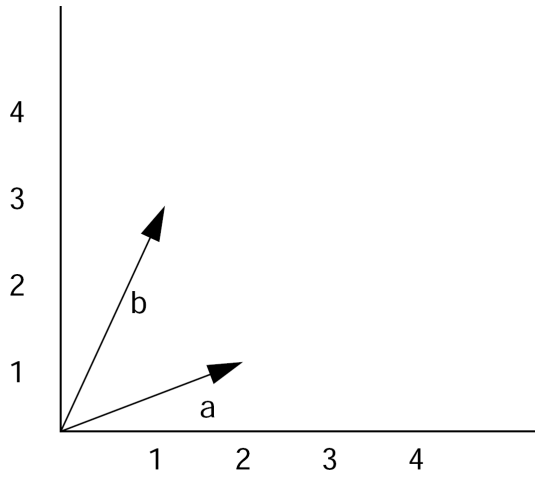


Abbildung 9.1
Koordinatensystem mit 2 Vektoren

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 Vektoren benutzen. Dann werden Sie auch erkennen, wie sinnvoll der Einsatz von Vektoren bei der Programmierung manchmal ist[1]. Vektoren heißen bei Entwicklern Arrays. Darum werde wir im Folgenden diesen Begriff 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. 8.2.1 beschrieben, auf.

[1] Zur Zeit müssen mir Sie dies einfach glauben.
[2] Außer, dass Sie lernen, wie man mit Arrays umgeht, was hier auch der Sinn ist.

Beispiel 9.1 *Arrays bei der Berechnung des arithmetischen Mittels*

```
Sub testeArithmetischesMittel()
        Dim meinErstesArray(2) As Double
        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 arithmetischesMittel As Double
        Dim i As Integer
        arithmetischesMittel = 0
        For i = 0 To 2
                arithmetischesMittel = arithmetischesMittel + dasArray(i)
        Next i
        arithmetischesMittel = arithmetischesMittel / 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. 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 Element 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
        arithmetischesMittel = arithmetischesMittel + 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 9.2 *Eine Funktion zur Ermittlung des arithmetischen Mittels bei Arrays mit beliebig vielen Elementen*

```
function berechneArithmetischesMittel(dasArray() as Double) as Double
        dim arithmetischesMittel as Double
        dim i As Integer
        dim obereGrenze as Integer
        obereGrenze=Ubound(dasArray)
        arithmetischesMittel = 0
        for i = 0 to obereGrenze
                arithmetischesMittel=arithmetischesMittel+dasArray(i)
        next i
        arithmetischesMittel=arithmetischesMittel/(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 dynamischen Indexen 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 beliebige Größen zuweisen. Wir veranschaulichen das sofort an einem Beispiel:

Beispiel 9.3 *Dynamisches Array zur Berechnung mehrerer arithmetischen Mittelwerte*

```
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 9.4 *Fester Indexbereich eines Arrays*

```
sub festeArrayGrenzen()
        dim festeGrenzenArray(1 to 4) As String
        dim i As Integer
        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 10

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 und 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 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*.

10.1 MsgBox Schaltflächen und ihre Bedeutung

Wir wollen das in Abb. 10.1 gezeigte Fragefenster implementieren. Wir zeigen Ihnen sofort das hierfür verantwortliche

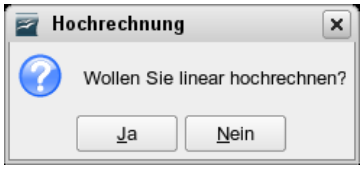


Abbildung 10.1
Fragefenster

hochkomplizierte Programm:

Beispiel 10.1 *Funktion zur Prognose des Jahresumsatzes mit Testprozedur*

```
sub frage()
        dim linear as integer
        '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, werden wir allerdings erst in eben diesem Kapitel verraten ☺.

Tabelle 10.1
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 10.2
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 6 beschrieben, eine Meldung ausgeben, man kann auch Schaltflächen auf die *MsgBox* platzieren, der *MsgBox* einen Titel geben und Grafiken, wie das Fragezeichen aus Abb. 10.1, 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 und OpenOffice verwenden eine Kodierung (und in diesem Fall sogar die Gleiche), um zu bestimmen, welche Grafiken oder Schaltflächen auf der *MsgBox* dargestellt werden. Die Tabellen 10.1 und 10.2 zeigen die gültigen Codes und ihre Bedeutung. Unser Fragefenster enthält die Schaltflächen „Ja“ und „Nein“, dem entspricht nach Tabelle 10.1 die Kodierung 4. Darüberhinaus enthält das Fragefenster ein Fragezeichen, das ist die Kodierung 32 aus Tabelle 10.2. Die Kodierungen sind dann, wie wir bereits festgestellt haben, Übergabeparameter 2 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. In Excel gibt es für die Zahlenkodierungen vordefinierte Konstanten. Diese gibt es aber in OpenOffice nicht, so dass wir auf eine Diskussion verzichten.
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 10.3 dargestellt. So erklärt sich die *if-elseif*-Anweisung aus Beispiel 10.1. Da nur die Schaltflächen „Ja“ und „Nein“ vorhanden sind, sind die möglichen Rückgabewerte 6 bzw. 7. Andere Rückgaben können nicht auftreten.

10.2 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* folgendermaßen aus: Der zugehörige Code ist:

Tabelle 10.3
Rückgabewerte der MsgBox

Rückgabewert	Schaltfläche
1	OK
2	Abbrechen
3	Abbruch
4	Wiederholen
5	Ignorieren
6	Ja
7	Nein

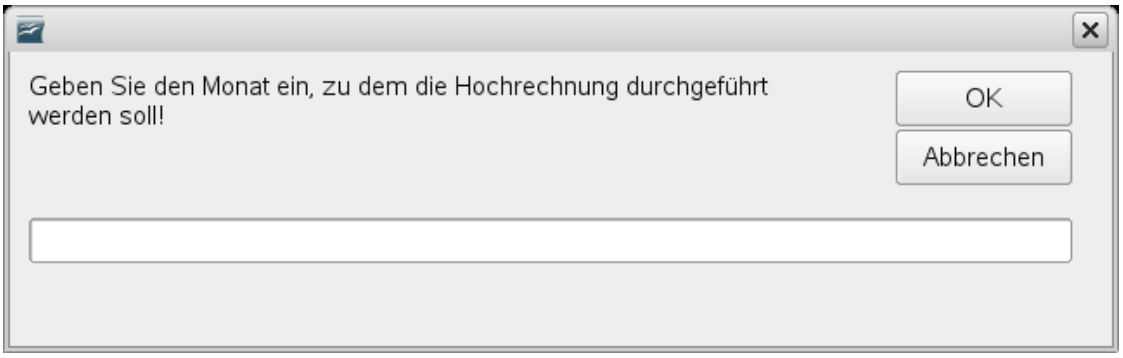


Abbildung 10.2
Eingabefenster

Beispiel 10.2 *Fragefenster und Programm zur Monatszahl in Monatsnamenumwandlung*

```
sub monatFragen()
        dim monat as Integer
        dim monatString as String
        monat=InputBox("Geben Sie den Monat ein, zu dem die Hochrechnung durchgeführt➥
             werden soll!")
        monatString=erzeugeMonatNameAusInteger(monat)
        MsgBox("Monat numerisch " & monat & "  Monat: " & monatString)
end sub


function erzeugeMonatNameAusInteger(monat As Integer) 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"

        erzeugeMonatNameAusInteger=monatArray(monat)
end function
```

Alles was in den Klammern hinter *InputBox* folgt, wird im Eingabefenster dargestellt (vgl. Abb. 10.2). 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 *erzeugeMonatNameAusInteger* 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. Und das wars auch schon, denn nun geben wir einfach *monatArray* von dem übergebenen Monat zurück.

Kapitel 11

Weitere einfache Praxisbeispiele

11.1 Aufträge nach Status farblich darstellen - Lieferantenbewertung

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 unten 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: Eine csv-Datei in eine Tabellenkalkulation einzulesen ist kein Problem.

	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

In OpenOffice geht das über *Einfügen* ⇨ *Tabelle aus Datei*, in Excel 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.
Mit Ihrem bisherigen Wissen ist dies wirklich ganz einfach:

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

- 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. 8.
- 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. 8.
- 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*

```
function ermittleFarbeAusStatusNr(statusNr As Integer) 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. In OpenOffice heißt die Ereignisprozedur *zeilenFaerben*, in Excel *zeilenFaerben_click*.

11.1.1 Realisierung in OpenOffice

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*

```
sub zeilenFaerben()
        dim statusNrSpalte As Integer
        dim tabellenblatt As Integer
        dim letzteBesetzteZeile As Integer
        dim i as Integer
        dim j as Integer
        dim statusNr As Integer
        dim farbe As Long

        dim myDoc as Object
        dim mySheet as Object
        dim myCell as Object

        letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(0, 0, 0)
        statusNrSpalte=ermittleLetzteBesetzteSpalteInZeile(0, 0, 0)
        myDoc = thisComponent
        mySheet = myDoc.sheets(0)
        for i= 0 to letzteBesetzteZeile
                myCell=mySheet.getCellByPosition(statusNrSpalte, i)
                statusNr=myCell.value
                farbe=ermittleFarbeAusStatusNr(statusNr)
                for j = 0 to statusNrSpalte
                        myCell=mySheet.getCellByPosition(j,i)
                        myCell.CellBackColor=farbe
                next j
        next i
end sub
```

Die Zeilen

```
letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(0, 0, 0)
statusNrSpalte=ermittleLetzteBesetzteSpalteInZeile(0, 0, 0)
```

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=1 to letzteBesetzteZeile
        myCell=mySheet.getCellByPosition(statusNrSpalte, i)
        statusNr=myCell.value
        farbe=ermittleFarbeAusStatusNr(statusNr)
next i
```

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. Die ermittelte Farbe wird der Variablen *farbe* zugewiesen. Dann startet eine Schleife über die besetzten Spalten der jeweiligen Zeile:

```
for j = 0 to statusNrSpalte
        myCell=mySheet.getCellByPosition(j,i)
        myCell.CellBackColor=farbe
next j
```

Jeder Zelle wird als Hintergrundfarbe nun die ermittelte Farbe zugewiesen:

```
myCell.CellBackColor=farbe
```

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

11.1.2 Realisierung in Excel

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

Beispiel 11.3 *Zellenfarben in Abhängigkeit von einem Auftragsstatus*

```
Private Sub zeilenFaerben_Click()
        Dim statusNrSpalte As Integer
        Dim tabellenblatt As Integer
        Dim letzteBesetzteZeile As Integer
        Dim i As Integer
        Dim j As Integer
        Dim statusNr As Integer
        Dim farbe As Long
        letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(1, 1, 1)
        statusNrSpalte = ermittleLetzteBesetzteSpalteInZeile(1, 1, 1)
        For i = 2 To letzteBesetzteZeile
                statusNr = Sheets(1).Cells(i, statusNrSpalte)
                farbe = ermittleFarbeAusStatusNr(statusNr)
                For j = 1 To statusNrSpalte
                        Sheets(1).Cells(i, j).Interior.Color = farbe
                Next j
        Next i
End Sub
```

Die Zeilen

```
letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(1, 1, 1)
statusNrSpalte=ermittleLetzteBesetzteSpalteInZeile(1, 1, 1)
```

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 = 2 To letzteBesetzteZeile
        statusNr = Sheets(1).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(1).Cells(i, j).Interior.Color = farbe
Next j
```

Jeder Zelle wird als Hintergrundfarbe nun die ermittelte Farbe zugewiesen:

```
Sheets(1).Cells(i, j).Interior.Color = farbe
```

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

11.2 Aufträge nach Vorgabe farblich darstellen - Lieferantenbewertung

Wir wandeln die Aufgabenstellung aus Kapitel Kap. 11.1 leicht ab. Die Farben der einzelnen Statusausprägungen sind 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 in beiden Tabellenkalkulationen jeweils 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 11.4 *Ermittlung der tabellenkalkulationsspezifischen Farbkodierung aus hexaddezimalem RGB*

```
function ermittleFarbeAusHexCode(rgbHexCode As String) as Long
        dim codeRot as Integer
        dim codeGruen as Integer
        dim codeBlau as Integer
        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 11.5 *Die Stringfunktionen*

```
sub stringfunktionen()
        dim s1 as String
        dim s2 as String
        dim s3 as String
        dim s4 as String
        dim l as Integer

        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)
                    s5=replace(s1, "a", "z") ' s5 ist zbcdefg
                    MsgBox(s5)
          end sub
```

Hier wurden zusätzlich noch die Stringfunktionen *len* und *replace* eingeführt. *len* ermittelt die Länge (Anzahl Zeichen) eines Strings, *replace* hingegen ersetzt in einem String eine Zeichenkette durch eine andere. Beispiel 11.5 sollte wegen der Kommentare selbsterklärend sein, so dass sich eine weitere Diskussion erübrigt. Kehren wir also zurück zu Beispiel 11.4. 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.

11.2.1 Realisierung in OpenOffice

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 11.6 *Zellenfarben in Abhängigkeit von einem Auftragsstatus*

```
          sub zeilenFaerben()
                    dim farbSpalte As Integer
                    dim tabellenblatt As Integer
                    dim letzteBesetzteZeile As Integer
                    dim i as Integer
                    dim j as Integer
                    dim statusNr As Integer
                    dim farbe As Long
                    dim farbeHex as String

                    dim myDoc as Object
                    dim mySheet as Object
                    dim myCell as Object
```

```
                    letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(0, 0, 0)
                    farbSpalte=ermittleLetzteBesetzteSpalteInZeile(0, 0, 0)
                    myDoc = thisComponent
                    mySheet = myDoc.sheets(0)
                    for i=1 to letzteBesetzteZeile
                              myCell=mySheet.getCellByPosition(farbSpalte, i)
                              farbeHex=myCell.String
                              farbe=ermittleFarbeAusHexCode(farbeHex)
                              for j = 0 to farbSpalte
                              myCell=mySheet.getCellByPosition(j,i)
                              myCell.CellBackColor=farbe
                              next j
                    next i
                    mySheet.columns.removeByIndex(farbSpalte, 1)
          end sub
```

Durch die Zeile

```
mySheet.columns.removeByIndex(farbSpalte, 1)
```

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

11.2.2 Realisierung in Excel

Wir zeigen Ihnen nun die Realisierung. Die Änderungen gegenüber Beipiel 11.3 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*

```
          Private Sub zeilenFaerben_Click()
                    Dim farbSpalte As Integer
                    Dim tabellenblatt As Integer
                    Dim letzteBesetzteZeile As Integer
                    Dim i As Integer
                    Dim j As Integer
                    Dim farbString As String
                    Dim farbe As Long
                    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(1, 1, 1)
                    farbSpalte = ermittleLetzteBesetzteSpalteInZeile(1, 1, 1)
                    For i = 2 To letzteBesetzteZeile
                              farbString = Sheets(1).Cells(i, farbSpalte)
                              farbe = ermittleFarbeAusHexCode(farbString)
                              For j = 1 To farbSpalte
                                        Sheets(1).Cells(i, j).Interior.Color = farbe
                              Next j
                    Next i
                    Sheets(1).Columns(farbSpalte).Delete
          End Sub
```

Durch die Zeile

```
Sheets(1).Columns(farbSpalte).Delete
```

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

11.3 Aufträge einlesen - Lieferantenbewertung

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

	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

```
"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. Das geht sowohl in Excel als auch in OpenOffice.
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,

	A	B	C	D	E	F
1	Inhalt	Spalte				
2	Kundennummer	A				
3	Name	B				
4	Straße	C				
5	PLZ	D				
6	Ort	E				
7	Ländercode	F				
8						

Csv-Datei erzeugen

Abbildung 11.3
Benutzerinterface zum Erzeugen der csv-Datei

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*

```
sub inDateiSchreiben()
        Dim fp As Integer
        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.*

```
          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. 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, das wir auf eine Diskussion verzichten.

Abbildung 11.4
Anführungszeichen ersetzen

Beispiel 11.10 *Spaltenbuchstaben in Zahlen umwandeln*

```
function wandleBuchstabenInZahl(buchstabe as String) as Integer
        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.

11.3.1 Realisierung in OpenOffice

Beispiel 11.11 *Spaltenbuchstaben in Zahlen umwandeln*

```
sub speichereAlsCsv()
          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 Integer

          dim kundenNrSpalteInteger as Integer
          dim nameSpalteInteger as Integer
          dim strasseSpalteInteger as Integer
          dim plzSpalteInteger as Integer
          dim stadtSpalteInteger as Integer
          dim laendercodeSpalteInteger as Integer

          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 Integer

          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 Integer
dim ausgabeDatei as string

dim myDoc as Object
dim mySheet as Object
dim myCell as Object

const KUNDENNUMMER_ZEILE=1
const NAME_ZEILE=2
const STRASSE_ZEILE=3
const PLZ_ZEILE=4
const ORT_ZEILE=5
const LAENDERCODE_ZEILE=5
const SPALTEN_SPALTE=1


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

myDoc = thisComponent
mySheet = myDoc.sheets(1)

myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, KUNDENNUMMER_ZEILE)
kundenNrSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, NAME_ZEILE)
nameSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, STRASSE_ZEILE)
strasseSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, PLZ_ZEILE)
plzSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, ORT_ZEILE)
stadtSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, LAENDERCODE_ZEILE)
laendercodeSpalte=myCell.String

kundenNrSpalteInteger=wandleBuchstabenInZahl(kundenNrSpalte)-1
nameSpalteInteger=wandleBuchstabenInZahl(nameSpalte)-1
strasseSpalteInteger=wandleBuchstabenInZahl(strasseSpalte)-1
plzSpalteInteger=wandleBuchstabenInZahl(plzSpalte)-1
stadtSpalteInteger=wandleBuchstabenInZahl(stadtSpalte)-1
laendercodeSpalteInteger=wandleBuchstabenInZahl(laendercodeSpalte)-1

letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(kundenNrSpalteInteger➥
    , 1, 0)
mySheet = myDoc.sheets(0)

fp = Freefile
Open ausgabeDatei For Output As #fp

for i= 1 to letzteBesetzteZeile
          kundenNr=mySheet.getCellByPosition(kundenNrSpalteInteger, i).String
          kundenName=mySheet.getCellByPosition(nameSpalteInteger, i).String
          strasse=mySheet.getCellByPosition(strasseSpalteInteger, i).String
          plz=mySheet.getCellByPosition(plzSpalteInteger, i).String
          stadt=mySheet.getCellByPosition(stadtSpalteInteger, i).String
          laendercode=mySheet.getCellByPosition(laendercodeSpalteInteger, i).➥
              String

          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
            end sub
```

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

```
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, KUNDENNUMMER_ZEILE)
kundenNrSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, NAME_ZEILE)
nameSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, STRASSE_ZEILE)
strasseSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, PLZ_ZEILE)
plzSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, ORT_ZEILE)
stadtSpalte=myCell.String
myCell=mySheet.getCellByPosition(SPALTEN_SPALTE, LAENDERCODE_ZEILE)
laendercodeSpalte=myCell.String
```

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

```
kundenNrSpalteInteger=wandleBuchstabenInZahl(kundenNrSpalte)-1
nameSpalteInteger=wandleBuchstabenInZahl(nameSpalte)-1
strasseSpalteInteger=wandleBuchstabenInZahl(strasseSpalte)-1
plzSpalteInteger=wandleBuchstabenInZahl(plzSpalte)-1
stadtSpalteInteger=wandleBuchstabenInZahl(stadtSpalte)-1
laendercodeSpalteInteger=wandleBuchstabenInZahl(laendercodeSpalte)-1
```

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=mySheet.getCellByPosition(kundenNrSpalteInteger, i).String
kundenName=mySheet.getCellByPosition(nameSpalteInteger, i).String
strasse=mySheet.getCellByPosition(strasseSpalteInteger, i).String
plz=mySheet.getCellByPosition(plzSpalteInteger, i).String
stadt=mySheet.getCellByPosition(stadtSpalteInteger, i).String
laendercode=mySheet.getCellByPosition(laendercodeSpalteInteger, i).String
```

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
```

11.3.2 Realisierung in Excel

Beispiel 11.12 *Spaltenbuchstaben in Zahlen umwandeln*

```
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 Integer

        Dim kundenNrSpalteInteger As Integer
        Dim nameSpalteInteger As Integer
        Dim strasseSpalteInteger As Integer
        Dim plzSpalteInteger As Integer
        Dim stadtSpalteInteger As Integer
        Dim laendercodeSpalteInteger As Integer

        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 Integer

        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 Integer
        Dim ausgabeDatei As String

        Const KUNDENNUMMER_ZEILE = 2
        Const NAME_ZEILE = 3
        Const STRASSE_ZEILE = 4
        Const PLZ_ZEILE = 5
        Const ORT_ZEILE = 6
        Const LAENDERCODE_ZEILE = 7
        Const SPALTEN_SPALTE = 2

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

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

                    kundenNrSpalte = Sheets(2).Cells(KUNDENNUMMER_ZEILE, SPALTEN_SPALTE)
                    nameSpalte = Sheets(2).Cells(NAME_ZEILE, SPALTEN_SPALTE)
                    strasseSpalte = Sheets(2).Cells(STRASSE_ZEILE, SPALTEN_SPALTE)
                    plzSpalte = Sheets(2).Cells(PLZ_ZEILE, SPALTEN_SPALTE)
                    stadtSpalte = Sheets(2).Cells(ORT_ZEILE, SPALTEN_SPALTE)
                    laendercodeSpalte = Sheets(2).Cells(LAENDERCODE_ZEILE, SPALTEN_SPALTE)

                    kundenNrSpalteInteger = wandleBuchstabenInZahl(kundenNrSpalte)
                    nameSpalteInteger = wandleBuchstabenInZahl(nameSpalte)
                    strasseSpalteInteger = wandleBuchstabenInZahl(strasseSpalte)
                    plzSpalteInteger = wandleBuchstabenInZahl(plzSpalte)
                    stadtSpalteInteger = wandleBuchstabenInZahl(stadtSpalte)
                    laendercodeSpalteInteger = wandleBuchstabenInZahl(laendercodeSpalte)

                    letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(➥
                        kundenNrSpalteInteger, 1, 1)

                    fp = FreeFile
                    Open ausgabeDatei For Output As #fp

                    For i = 1 To letzteBesetzteZeile
                              kundenNr = Sheets(1).Cells(i, kundenNrSpalteInteger)
                              kundenName = Sheets(1).Cells(i, nameSpalteInteger)
                              strasse = Sheets(1).Cells(i, strasseSpalteInteger)
                              plz = Sheets(1).Cells(i, plzSpalteInteger)
                              stadt = Sheets(1).Cells(i, stadtSpalteInteger)
                              laendercode = Sheets(1).Cells(i, laendercodeSpalteInteger)

                              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(2).Cells(KUNDENNUMMER_ZEILE, SPALTEN_SPALTE)
nameSpalte = Sheets(2).Cells(NAME_ZEILE, SPALTEN_SPALTE)
strasseSpalte = Sheets(2).Cells(STRASSE_ZEILE, SPALTEN_SPALTE)
plzSpalte = Sheets(2).Cells(PLZ_ZEILE, SPALTEN_SPALTE)
stadtSpalte = Sheets(2).Cells(ORT_ZEILE, SPALTEN_SPALTE)
laendercodeSpalte = Sheets(2).Cells(LAENDERCODE_ZEILE, SPALTEN_SPALTE)
```

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

```
        kundenNrSpalteInteger = wandleBuchstabenInZahl(kundenNrSpalte)
        nameSpalteInteger = wandleBuchstabenInZahl(nameSpalte)
        strasseSpalteInteger = wandleBuchstabenInZahl(strasseSpalte)
        plzSpalteInteger = wandleBuchstabenInZahl(plzSpalte)
        stadtSpalteInteger = wandleBuchstabenInZahl(stadtSpalte)
        laendercodeSpalteInteger = 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(1).Cells(i, kundenNrSpalteInteger)
kundenName = Sheets(1).Cells(i, nameSpalteInteger)
strasse = Sheets(1).Cells(i, strasseSpalteInteger)
plz = Sheets(1).Cells(i, plzSpalteInteger)
stadt = Sheets(1).Cells(i, stadtSpalteInteger)
laendercode = Sheets(1).Cells(i, laendercodeSpalteInteger)
```

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

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.

12.1 Berechnung der Umsatzprognose

12.1.1 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 $12/4 = 3$.

12.1.2 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

$$prognostizierterJahresumsatz = \frac{bisherigerUmsatz * 100}{akkumulierteprozentualeUmsatzverteilung}$$

und das Ganze mit Beispielzahlen Bisheriger Umsatz: 31.500 Euro entsprechen 23 Prozent des erwarteten Umsatzes

$$prognostizierterJahresumsatz = \frac{31.500 * 100}{23}$$

[1] dass wir noch schöner formatieren müssen und auch noch Grafiken einbauen, ist keine Frage.

$$prognostizierter Jahresumsatz = 136956,52 Euro$$

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 12.1 *Funktion zur Prognose des Jahresumsatzes mit Testprozedur*

```
sub testeHochrechnung()
        dim prozentualeVerteilungArray (1 to 12) as double
        dim monat as Integer
        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 ➥
    Integer, prozentualeVerteilungArray() as Double)
        dim i As Integer
        dim akkumulierteProzentualeUmsatzverteilung
        dim planfaktor as Double
        akkumulierteProzentualeUmsatzverteilung=0
        for i = 1 To monat
            akkumulierteProzentualeUmsatzverteilung=
                            akkumulierteProzentualeUmsatzverteilung + ➥
                                prozentualeVerteilungArray(i)
        next i
        planfaktor=100/akkumulierteProzentualeUmsatzverteilung
        prognostiziereJahresumsatz=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. Wir 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
```

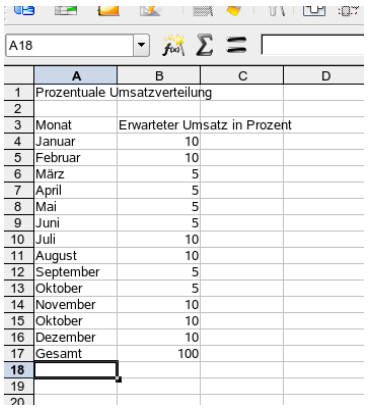
Dann berechnen wir den Faktor, mit dem der bisherige Umsatz multipliziert werden muss und abschließend erfolgt die Bestimmung der Prognose durch Multiplikation des bisherigen Umsatzes mit dem berechneten Faktor:

```
planfaktor=100/akkumulierteProzentualeUmsatzverteilung
prognostiziereJahresumsatz=planfaktor*bisherigerUmsatz
```

Und das wars.
Damit sind wir mit unserem Vorhaben auch schon fast durch. Den gesamten bisherigen Verkaufsbetrag bestimmen können Sie seit Kap. 7, die Umsatzprognose berechnen seit Kap. 12.1.
Das Einzige, was noch fehlt, ist das Einlesen der prozentualen Umsatzverteilung und die Ausgabe des Ergebnisses.

12.2 Einbau in das Provisionsbeispiel

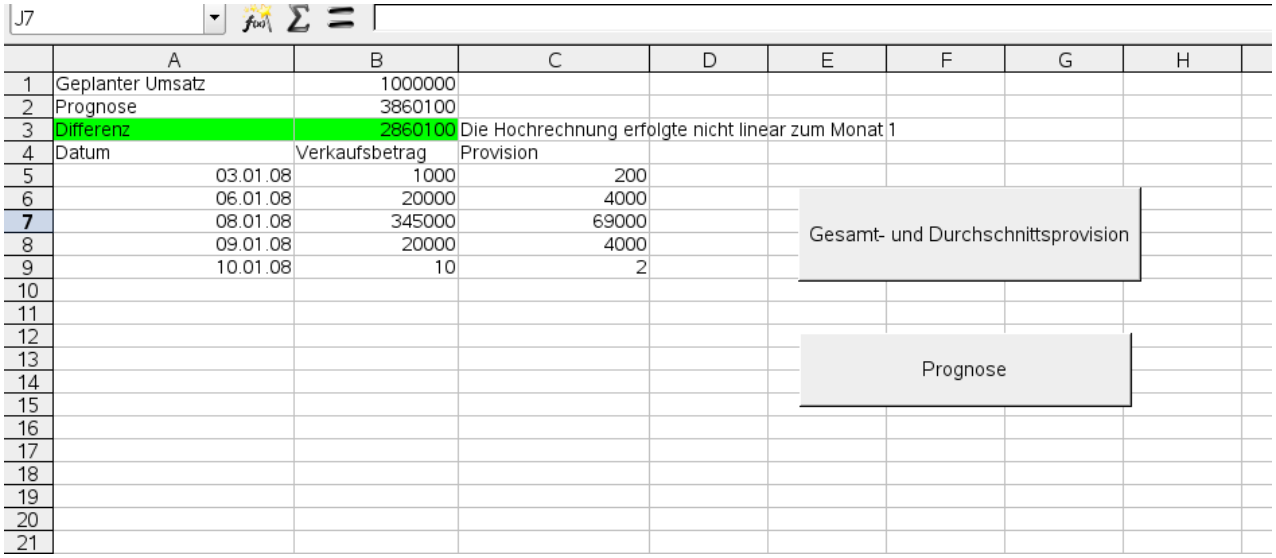
Zum Einlesen der prozentualen Umsatzverteilung benutzen wir natürlich ebenfalls die Tabellenkalkulation. Wir schreiben die prozentuale Umsatzverteilung einfach in das dritte Arbeitsblatt der Tabellenkalkulation. Wir erwarten die prozentuale Umsatzverteilung in der in Abb. 12.1 dargestellten Form: Zu Beginn der Verarbeitung fragen wir den Benutzer, ob die



	A	B	C	D
1	Prozentuale Umsatzverteilung			
2				
3	Monat	Erwarteter Umsatz in Prozent		
4	Januar	10		
5	Februar	10		
6	März	5		
7	April	5		
8	Mai	5		
9	Juni	5		
10	Juli	10		
11	August	10		
12	September	5		
13	Oktober	5		
14	November	10		
15	Oktober	10		
16	Dezember	10		
17	Gesamt	100		
18				
19				

Abbildung 12.1
Eingabe der prozentualen Umsatzverteilung

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 12.1. 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. 12.2 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. 10 beschrieben mit einer *MsgBox*.



	A	B	C	D	E	F	G	H
1	Geplanter Umsatz	1000000						
2	Prognose	3860100						
3	Differenz	2860100	Die Hochrechnung erfolgte nicht linear zum Monat 1					
4	Datum	Verkaufsbetrag	Provision					
5	03.01.08	1000	200					
6	06.01.08	20000	4000					
7	08.01.08	345000	69000					
8	09.01.08	20000	4000					
9	10.01.08	10	2					

Abbildung 12.2
Darstellung der Prognose

12.2.1 OpenOffice Einbau in das Provisionsbeispiel: Erste einfache Form

[2] Wir beginnen mit der Prozedur, die die prozentuale Umsatzverteilung einliest. Wir zeigen Ihnen sofort den Code:

Beispiel 12.2 *Prozedur zum Einlesen der prozentualen Umsatzverteilung*

```
        function liesNichtLineareUmsatzverteilungEin(tabellenblatt as Integer,
startzeile As Integer, wertespalte as integer) as Double

                    dim myDoc as Object
                    dim mySheet as Object
                    dim cell as Object
                    dim endzeile as integer
                    dim i as integer
                    dim prozentualeUmsatzverteilungArray() as Double
                    myDoc = thisComponent
                    mySheet = myDoc.sheets(tabellenblatt)
                    endzeile=startzeile+11
                    for i = startzeile to endzeile
                              cell=mySheet.getCellByPosition(wertespalte, i)
                              prozentualeUmsatzverteilungArray(i-startzeile+1)= cell.Value
                    next i
                    liesNichtLineareUmsatzverteilungEin=prozentualeUmsatzverteilungArray
          end sub
```

Die Funktion 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. die Spalte, in der die Werte stehen (*wertespalte*).

Zurückgegeben wird 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.

[2] Excel weiter auf Seite 122

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 12.3 *Ereignisprozedur zur Kontrolle der Umsatzziele in OpenOffice*

```
sub kontrolliereUmsatz()
      '### Variablen deklarieren ###
      dim myDoc as Object
      dim myFirstSheet as Object
      dim mySecondSheet as Object
      dim cell as Object
      dim prozentualeVerteilungArray() as Double
      dim linear As Integer
      dim monat as Integer
      dim letzteBesetzteZeile as Integer
      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 Integer = 0
      const ERSTE_ZEILE_MIT_VERKAUFSBETRAG as Integer = 4
      const BISHERIGER_VERKAUFSBETRAG_SPALTE=1
      const BISHERIGER_VERKAUFSBETRAG_ZEILE=1
      const PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE=3
      const PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE=1
      const GEPLANTER_UMSATZ_SPALTE=1
      const GEPLANTER_UMSATZ_ZEILE=0
      '### Objekte deklarieren###
      myDoc = thisComponent
      myFirstSheet =  myDoc.sheets(0)
      mySecondSheet = myDoc.sheets(1)
      '#######################
      '### Monat bestimmen ###
      '#######################
      letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(DATUMSSPALTE, ➥
          ERSTE_ZEILE_MIT_VERKAUFSBETRAG, 0)
      cell=myFirstSheet.getCellByPosition(DATUMSSPALTE, letzteBesetzteZeile)
      letztesVerkaufsdatum=cell.value
      monat=Month(letztesVerkaufsdatum)
      '### Sicherstellen, dass der bisherige ###
      '### Verkaufsbetrag aktuell ist        ###
      call berechneBisherigenVerkaufsbetragUmsatzsteuerGesamtUndDurchschnittsprovision()
      '###bisherigenUmsatz holen###
      cell=mySecondSheet.getCellByPosition(BISHERIGER_VERKAUFSBETRAG_SPALTE, ➥
          BISHERIGER_VERKAUFSBETRAG_ZEILE)
      bisherigerUmsatz=cell.Value
      '### Lineare Prognose ja oder nein ###
      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
                prozentualeUmsatzverteilungArray= liesNichtLineareUmsatzverteilungEin(2, ➥
                    PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, ➥
                    PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE)
                prognostizierterJahresumsatz= prognostiziereJahresumsatzNichtLinear(➥
                    bisherigerUmsatz, monat, prozentualeUmsatzverteilungArray)
                hochrechnungstext="Die Hochrechnung erfolgte nicht linear zum Monat " & monat
      end if
      '### Geplanten Umsatz einlesen ###
      cell=myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE,  GEPLANTER_UMSATZ_ZEILE)
      geplanterJahresumsatz=cell.Value
      umsatzDifferenz= prognostizierterJahresumsatz - geplanterJahresumsatz
      '### Zellen einfärben ###
```

```
        if umsatzDifferenz > 0 then
                farbe=rgb(0, 255, 0)
        else
                farbe=rgb(255, 0, 0)
        end if
      '### Werte in Tabelle schreiben###
      cell=myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE-1,  GEPLANTER_UMSATZ_ZEILE+1)
      cell.string="Prognose"
      cell=myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE,  GEPLANTER_UMSATZ_ZEILE+1)
      cell.Value=prognostizierterJahresumsatz
      cell=myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE-1,  GEPLANTER_UMSATZ_ZEILE+2)
      cell.string="Differenz"
      cell.CellBackColor=farbe
      cell=myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE,  GEPLANTER_UMSATZ_ZEILE+2)
      cell.Value=umsatzDifferenz
      cell.CellBackColor=farbe
      cell=myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE+1,  GEPLANTER_UMSATZ_ZEILE+2)
      cell.String=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[3]. 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, das haben wir in Beispiel 8.2 bereits gelöst. Das war die Ereignisprozedur, die den bisherigen Umsatz, die bisherige Provision, Umsatzsteuer usw. in das zweite Tabellenblatt schrieb. Für OpenOffice sind Ereignisprozeduren aber auch normale Prozeduren, die man auch aus eigenen Programmen aufrufen kann. Dies ist die Zeile:

```
call berechneBisherigenVerkaufsbetragUmsatzsteuerGesamtUndDurchschnittsprovision()
```

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
    prozentualeUmsatzverteilungArray= liesNichtLineareUmsatzverteilungEin(2, ➥
        PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE)
    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
```

[3] Sie sehen, wie sinnvoll Funktionen sind, denn wieder nutzen wir unsere Funktion zur Bestimmung der letzten besetzten Zeile

Zum Schluss schreiben wir die Ergebnisse in das erste Tabellenblatt und geben dabei den Zellen mit der Differenz die gewünschte Farbe. Dies erfolgt mit:[4]

```
cell.CellBackColor=farbe
```

Excel

Wir beginnen mit der Prozedur, die die prozentuale Umsatzverteilung einliest. Wir zeigen Ihnen sofort den Code:

Beispiel 12.4 *Prozedur zum Einlesen der prozentualen Umsatzverteilung*

```
Function liesNichtLineareUmsatzverteilungEin(tabellenblatt As Integer, startzeile As Integer,➥
     wertespalte As Integer)
      Dim endzeile As Integer
      Dim prozentualeUmsatzverteilungArray(1 To 12) As Double
      Dim i As Integer

      endzeile = startzeile + 11
      For i = startzeile To endzeile
                prozentualeUmsatzverteilungArray(i - startzeile + 1) = Sheets(3).Cells(i, ➥
                    wertespalte)
      Next i
      liesNichtLineareUmsatzverteilungEin=prozentualeUmsatzverteilungArray
End Function
```

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 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 12.5 *Ereignisprozedur zur Kontrolle der Umsatzziele in Excel*

```
Sub kontrolliereUmsatz_Click()
      '### Variablen deklarieren ###
      Dim prozentualeVerteilungArray() As Double
      Dim linear As Integer
      Dim monat As Integer
      Dim letzteBesetzteZeile As Integer
      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 umsatzverteilungArray() As Double
      '### Konstanten deklarieren ###
      Const DATUMSSPALTE As Integer = 1
      Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Integer = 5
      Const BISHERIGER_VERKAUFSBETRAG_SPALTE = 2
      Const BISHERIGER_VERKAUFSBETRAG_ZEILE = 2
```

[4] OpenOffice weiter auf Seite 124

```
        Const PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE = 4
        Const PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE = 2
        Const GEPLANTER_UMSATZ_SPALTE = 2
        Const GEPLANTER_UMSATZ_ZEILE = 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(2).Cells(BISHERIGER_VERKAUFSBETRAG_ZEILE, ➥
            BISHERIGER_VERKAUFSBETRAG_SPALTE)
        '### lineare Hochrechnung ja oder nein ###
        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
            prozentualeUmsatzverteilungArray=liesNichtLineareUmsatzverteilungEin(2, ➥
                PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, ➥
                PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE)
            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
        '### Zellen färben
        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[5]. 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, das haben wir in Beispiel 8.3 bereits gelöst. Das war die Ereignisprozedur, die den bisherigen Umsatz, die bisherige Provision, Umsatzsteuer usw. in das zweite Tabellenblatt schrieb. Für Excel sind Ereignisprozeduren aber auch normale Prozeduren, die man auch aus eigenen Programmen aufrufen kann. Dies ist die Zeile:

```
Call berechneGesamtUndDurchschnittsprovision_Click
```

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
```

[5]Sie sehen, wie sinnvoll Funktionen sind, denn wieder nutzen wir unsere Funktion zur Bestimmung der letzten besetzten Zeile

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
    prozentualeUmsatzverteilungArray=liesNichtLineareUmsatzverteilungEin(2, ➥
        PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, _
                PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE)
    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
```

12.2.2 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. 12.2.1 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. 12.3 zeigt 5 Verkäufe in insgesamt 3 Monaten und einen geplanten Jahresumsatz von einer Million.

provisionPrognoseMonat - OpenOffice.org Calc

	A	B	C
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

Gesamt- und Durchschnittsprovision

Prognose

Abbildung 12.3
Verkäufe und geplanter Umsatz

provisionPrognoseMonat - OpenOffice.org Calc

	A	B	C	D	E	F	G	H
1	Anzahl Verkäu	Gesamtverkaufsbe	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		
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	

Abbildung 12.4
Auswertung der ersten 3 Monate

Abb. 12.4 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 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. 12.3. 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. 12.1). 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. 12.1 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.
Sie sehen, die Anwendung, die Abb. 12.4 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. 7.1.1 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 12.6 *Berechnung des geplanten Monatsumsatzes aus dem geplanten Jahresumsatz*

```
function bisZuDiesemMonatZuErreichen(geplanterUmsatz as Double, monat as Integer, ➥
    prozentualeVerteilungArray() as Double)
        dim i As Integer
        dim akkumulierteProzentualeUmsatzverteilung
        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 12.1 entwickelt, ihren Aufruf bereits ebendort oder in den Beispielen 12.5 oder 12.3 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 12.7 *Beispiel 5.6 erneut dargestellt*

```
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:

```
prognostiziertеProvision=berechneProvision(prognostizierterJahresumsatz,➥
    prognostizierterJahresumsatz)
```

Der Code zur Erzeugung der Ausgaben sieht also folgendermaßen aus:

Beispiel 12.8 *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)
prognostiziertеProvision=berechneProvision(prognostizierterJahresumsatz, ➥
    prognostizierterJahresumsatz)
umsatzMonatsDifferenz=bisherigerUmsatz-geplanterUmsatzBisMonat
umsatzDifferenz=prognostizierterJahresumsatz-geplanterJahresumsatz
provisionsDifferenzMonat=realeProvisionBisMonat-geplanteProvisionBisMonat
provisionsDifferenz=prognostiziertеProvision-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. Zunächst die OpenOffice-Version[6]:

[6] Excel weiter auf Seite 133

Provisions- und Umsatzprognose, endgültige Form: OpenOffice

Beispiel 12.9 *Endgültige Provision-Prognoselogik in OpenOffice*

```
sub kontrolliereUmsatzMonatlich()
      dim myDoc as Object
      dim myFirstSheet as Object
      dim mySecondSheet as Object
      dim cell as Object
      dim prozentualeVerteilungArray() as Double
      dim linear As Integer
      dim monat as Integer
      dim alterMonat as Integer
      dim alterMonatAlsString as String
      dim letzteBesetzteZeile as Integer
      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 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 hochrechnungsMonat as integer
      dim bisherigeProvision as double
      dim i as Integer

      const DATUMSSPALTE as Integer = 0
      const VERKAUFSBETRAGSPALTE as Integer = 1
      const PROVISIONSSPALTE as Integer = 2
      const ERSTE_ZEILE_MIT_VERKAUFSBETRAG as Integer = 4
      const BISHERIGER_VERKAUFSBETRAG_SPALTE=1
      const BISHERIGER_VERKAUFSBETRAG_ZEILE=1
      const PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE=3
      const PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE=1
      const GEPLANTER_UMSATZ_SPALTE=1
      const GEPLANTER_UMSATZ_ZEILE=0

      myDoc = thisComponent
      myFirstSheet = myDoc.sheets(0)
      mySecondSheet = myDoc.sheets(1)

      hochrechnungsMonat=InputBox("Geben Sie den Stützmonat der Hochrechnung ein!")
      hochrechnungsMonatAlsString=erzeugeMonatNameAusInteger(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
                prozentualeUmsatzverteilungArray=liesNichtLineareUmsatzverteilungEin(2, ➥
                    PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, ➥
                    PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE)
                hochrechnungstext="Die Hochrechnung erfolgte nicht linear zum Monat " & ➥
                    hochrechnungsMonatAlsString
      end if
      cell=myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE,  GEPLANTER_UMSATZ_ZEILE)
      geplanterJahresumsatz=cell.Value
      geplanteProvision=berechneProvision(geplanterJahresumsatz, geplanterJahresumsatz)
```

```
letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(DATUMSSPALTE, ➥
    ERSTE_ZEILE_MIT_VERKAUFSBETRAG, 0)
alterMonat=1
bisherigeProvision=0
bisherigerUmsatz=0
for i= ERSTE_ZEILE_MIT_VERKAUFSBETRAG to letzteBesetzteZeile + 1
          cell=myFirstSheet.getCellByPosition(DATUMSSPALTE, i)
          if cell.Type<>com.sun.star.table.CellContentType.EMPTY then
                    datum=cell.Value
                    monat=Month(datum)
          end if
          if (monat <> alterMonat) or (i = letzteBesetzteZeile + 1) then
                    'Monatswechsel Prognose und Werte müssen geschrieben werden
                    alterMonatAlsString=erzeugeMonatNameAusInteger(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)
                    prognostiziertеProvision=berechneProvision(➥
                        prognostizierterJahresumsatz, prognostizierterJahresumsatz)
                    umsatzMonatsDifferenz=bisherigerUmsatz-geplanterUmsatzBisMonat
                    umsatzDifferenz=prognostizierterJahresumsatz-geplanterJahresumsatz
                    provisionsDifferenzMonat=realeProvisionBisMonat-➥
                        geplanteProvisionBisMonat
                    provisionsDifferenz=prognostiziertеProvision-geplanteProvision
                    if umsatzDifferenz > 0 then
                              farbe=rgb(0, 255, 0)
                    else
                              farbe=rgb(100, 0, 0)
                    end if
                    call schreibeUmsatzUndProvision(bisherigerUmsatz, ➥
                        geplanterUmsatzBisMonat, umsatzMonatsDifferenz, ➥
                        prognostizierterJahresumsatz, umsatzDifferenz, ➥
                        bisherigeProvision, geplanteProvisionBisMonat, ➥
                        realeProvisionBisMonat, provisionsDifferenzMonat, ➥
                        prognostiziertеProvision, provisionsDifferenz, alterMonat, ➥
                        alterMonatAlsString, farbe)
                    alterMonat=monat
          end if
          if (monat > hochrechnungsMonat) or (i = letzteBesetzteZeile + 1)then
                    exit for
          end if
          cell=myFirstSheet.getCellByPosition(PROVISIONSSPALTE, i)
          bisherigeProvision=bisherigeProvision + cell.Value
          cell=myFirstSheet.getCellByPosition(VERKAUFSBETRAGSPALTE, i)
          bisherigerUmsatz=bisherigerUmsatz + cell.Value
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. 12.2.2 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 cell.Type<>com.sun.star.table.CellContentType.EMPTY then
          datum=cell.Value
          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 12.8 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[7]. Ist dies der Fall, wird die *for*-Schleife beendet. *Exit For* beendet eine *for* Schleife. Der Grund für diese Bedingung ist folgender: 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.
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, 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
```

und die Werte des letzten Monats werden 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.

[7] Die mit *or* angeschlossene Bedingung besprechen wir, wie bereits gesagt, später.

Beispiel 12.10 *Schreiben in Tabellenblatt 2*

```
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 ➥
    Integer, alterMonatAlsString as string, farbe as long)
        dim myDoc as Object
        dim mySecondSheet as Object
        dim cell as Object

        const UMSATZ_START_SPALTE as Integer = 0
        const UMSATZ_START_ZEILE as Integer = 4
        const PROVISION_START_SPALTE as Integer = 0
        const PROVISION_START_ZEILE as Integer = 20
        myDoc = thisComponent
        mySecondSheet = myDoc.sheets(1)

        'umsatz schreiben
        cell = mySecondSheet.getCellByPosition(UMSATZ_START_SPALTE,  ➥
            UMSATZ_START_ZEILE+alterMonat)
        cell.string=alterMonatAlsString
        cell = mySecondSheet.getCellByPosition(UMSATZ_START_SPALTE+1, ➥
            UMSATZ_START_ZEILE+alterMonat)
        cell.Value=bisherigerUmsatz
        cell.CellBackColor=farbe
        cell = mySecondSheet.getCellByPosition(UMSATZ_START_SPALTE+2, ➥
            UMSATZ_START_ZEILE+alterMonat)
        cell.Value=geplanterUmsatzBisMonat
        cell.CellBackColor=farbe
        cell = mySecondSheet.getCellByPosition(UMSATZ_START_SPALTE+3, ➥
            UMSATZ_START_ZEILE+alterMonat)
        cell.Value=umsatzMonatsDifferenz
        cell.CellBackColor=farbe
        cell = mySecondSheet.getCellByPosition(UMSATZ_START_SPALTE+4, ➥
            UMSATZ_START_ZEILE+alterMonat)
        cell.Value=prognostizierterJahresumsatz
        cell.CellBackColor=farbe
        cell = mySecondSheet.getCellByPosition(UMSATZ_START_SPALTE+5, ➥
            UMSATZ_START_ZEILE+alterMonat)
        cell.Value=umsatzDifferenz
        cell.CellBackColor=farbe

        'provision schreiben
        cell = mySecondSheet.getCellByPosition(PROVISION_START_SPALTE, ➥
            PROVISION_START_ZEILE+alterMonat)
        cell.string=alterMonatAlsString
        cell = mySecondSheet.getCellByPosition(PROVISION_START_SPALTE+1, ➥
            PROVISION_START_ZEILE+alterMonat)
        cell.Value=bisherigeProvision
        cell.CellBackColor=farbe
        cell = mySecondSheet.getCellByPosition(PROVISION_START_SPALTE+2,  ➥
            PROVISION_START_ZEILE+alterMonat)
        cell.Value=geplanteProvisionBisMonat
        cell.CellBackColor=farbe
        cell = mySecondSheet.getCellByPosition(PROVISION_START_SPALTE+3, ➥
            PROVISION_START_ZEILE+alterMonat)
        cell.Value=realeProvisionBisMonat
        cell.CellBackColor=farbe
        cell = mySecondSheet.getCellByPosition(PROVISION_START_SPALTE+4, ➥
            PROVISION_START_ZEILE+alterMonat)
        cell.Value=provisionsDifferenzMonat
        cell.CellBackColor=farbe
        cell = mySecondSheet.getCellByPosition(PROVISION_START_SPALTE+5, ➥
            PROVISION_START_ZEILE+alterMonat)
        cell.Value=prognostizierteProvision
        cell.CellBackColor=farbe
```

```
                cell = mySecondSheet.getCellByPosition(PROVISION_START_SPALTE+6, ➥
                    PROVISION_START_ZEILE+alterMonat)
                cell.Value=provisionsDifferenz
                cell.CellBackColor=farbe
        end sub
```

Beispiel 12.11 *Schreiben in Tabellenblatt 1*

```
        sub schreibePrognose(prognostizierterJahresumsatz as double, umsatzDifferenz as double,➥
            _
                                                                hochrechnungstext as String➥
                                                                    , farbe as long)
                dim myDoc as Object
                dim myFirstSheet as Object
                dim cell as Object

                const GEPLANTER_UMSATZ_SPALTE=1
                const GEPLANTER_UMSATZ_ZEILE=0

                myDoc = thisComponent
                myFirstSheet = myDoc.sheets(0)
                cell = myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE-1,  ➥
                    GEPLANTER_UMSATZ_ZEILE+1)
                cell.string="Prognose"
                cell = myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE,  ➥
                    GEPLANTER_UMSATZ_ZEILE+1)
                cell.Value=prognostizierterJahresumsatz
                cell = myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE-1,  ➥
                    GEPLANTER_UMSATZ_ZEILE+2)
                cell.string="Differenz"
                cell.CellBackColor=farbe
                cell = myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE,  ➥
                    GEPLANTER_UMSATZ_ZEILE+2)
                cell.Value=umsatzDifferenz
                cell.CellBackColor=farbe
                cell = myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE+1,  ➥
                    GEPLANTER_UMSATZ_ZEILE+2)
                cell.String=hochrechnungstext
        end sub
```

Provisions- und Umsatzprognose, endgültige Form: Excel

Beispiel 12.12 *Endgültige Provision-Prognoselogik in Excel*[8]

```
    Sub kontrolliereUmsatzMonatlich_Click()
        Dim prozentualeVerteilungArray() As Double
        Dim linear As Integer
        Dim monat As Integer
        Dim alterMonat As Integer
        Dim alterMonatAlsString As String
        Dim letzteBesetzteZeile As Integer
        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
```

[8] OpenOffice weiter auf Seite 137

```
Dim prognostiziertеProvision 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 Integer
Dim bisherigeProvision As Double
Dim i As Integer

Const DATUMSSPALTE As Integer = 1
Const VERKAUFSBETRAGSPALTE As Integer = 2
Const PROVISIONSSPALTE As Integer = 3
Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Integer = 5
Const BISHERIGER_VERKAUFSBETRAG_SPALTE = 2
Const BISHERIGER_VERKAUFSBETRAG_ZEILE = 2
Const PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE = 4
Const PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE = 2
Const GEPLANTER_UMSATZ_SPALTE = 2
Const GEPLANTER_UMSATZ_ZEILE = 1

hochrechnungsMonat = InputBox("Geben Sie den Stützmonat der Hochrechnung ein!")
hochrechnungsMonatAlsString = erzeugeMonatNameAusInteger(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
          prozentualeUmsatzverteilungArray= liesNichtLineareUmsatzverteilungEin(2, ➥
              PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE, ➥
              PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE)
          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, 1)
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 = erzeugeMonatNameAusInteger(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)
                            prognostiziertеProvision = berechneProvision(➥
                                prognostizierterJahresumsatz, prognostizierterJahresumsatz)
                            umsatzMonatsDifferenz = bisherigerUmsatz - geplanterUmsatzBisMonat
                            umsatzDifferenz = prognostizierterJahresumsatz - ➥
                                geplanterJahresumsatz
                            provisionsDifferenzMonat = realeProvisionBisMonat - ➥
                                geplanteProvisionBisMonat
                            provisionsDifferenz = prognostiziertеProvision - 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, ➥
                                prognostiziertеProvision, 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. 12.2.2 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 12.8 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[9]. 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.
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, 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
```

und die Werte des letzten Monats werden 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 12.13 *Schreiben in Tabellenblatt 2*

```
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 Integer, ➥
    alterMonatAlsString as string, farbe as long)

        Const UMSATZ_START_SPALTE As Integer = 1
        Const UMSATZ_START_ZEILE As Integer = 5
        Const PROVISION_START_SPALTE As Integer = 1
        Const PROVISION_START_ZEILE As Integer = 21

        'umsatz schreiben
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE) =         ➥
                    alterMonatAlsString
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE).Interior.Color = ➥
               farbe
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 1) =     ➥
                    bisherigerUmsatz
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 1).Interior.➥
            Color=farbe
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 2) =     ➥
                    geplanterUmsatzBisMonat
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 2).Interior.➥
            Color= farbe
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 3) =     ➥
                    umsatzMonatsDifferenz
```

[9]Die mit *or* angeschlossene Bedingung besprechen wir, wie bereits gesagt, später.

```
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 3).Interior.➥
            Color= farbe
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 4) =     ➥
                      prognostizierterJahresumsatz
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 4).Interior.➥
            Color= farbe
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 5) = ➥
            umsatzDifferenz
        Sheets(2).Cells(UMSATZ_START_ZEILE + alterMonat, UMSATZ_START_SPALTE + 5).Interior.➥
            Color= farbe

        'provision schreiben
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE) = ➥
            alterMonatAlsString
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, _
                                          PROVISION_START_SPALTE).Interior.Color = farbe
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 1) = ➥
            bisherigeProvision
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, _
                                    PROVISION_START_SPALTE + 1).Interior.Color = farbe
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 2) = ➥
            geplanteProvisionBisMonat
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, _
                                                PROVISION_START_SPALTE + 2).Interior.➥
                                                    Color = farbe
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 3) = ➥
            realeProvisionBisMonat
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 3).➥
            Interior.Color = farbe
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 4) = ➥
            provisionsDifferenzMonat
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 4).➥
            Interior.Color = farbe
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 5) = ➥
            prognostizierteProvision
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 5).➥
            Interior.Color = farbe
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 6) = ➥
            provisionsDifferenz
        Sheets(2).Cells(PROVISION_START_ZEILE + alterMonat, PROVISION_START_SPALTE + 6).➥
            Interior.Color = farbe
End Sub
```

Beispiel 12.14 *Schreiben in Tabellenblatt 1*

```
        Sub schreibePrognose(prognostizierterJahresumsatz As Double, umsatzDifferenz As Double,➥
             _
                                                                    hochrechnungstext➥
                                                                         As String, ➥
                                                                        farbe As ➥
                                                                        Long)
                Const GEPLANTER_UMSATZ_SPALTE = 2
                Const GEPLANTER_UMSATZ_ZEILE = 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 13

Fehlerbehandlung und Plausibilitätsprüfungen

In diesem Kapitel wollen wir über Fehler sprechen, die durch falsche Eingaben der Benutzer unserer Anwendungen entstehen. Betrachten wir dazu Abb. 13.1.

Im ersten Screenshot (Abb. 13.1) 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 verhalten sich beide hier betrachteten Tabellenkalkulationsprogramme so, als ob der Benutzer anstelle von *e* eine Null eingegeben hätte. Bei unseren selbstgeschriebenen Funktionen und Prozeduren verhält sich OpenOffice komplett identisch, während Excel, wenn die Dateitypen in der Übergabeliste nicht übereinstimmen, *#WERT!* in die Zelle mit dem Funktionsbezug schreibt, so dass der Fehler wenigstens sichtbar ist (vgl. Abb. 13.2).

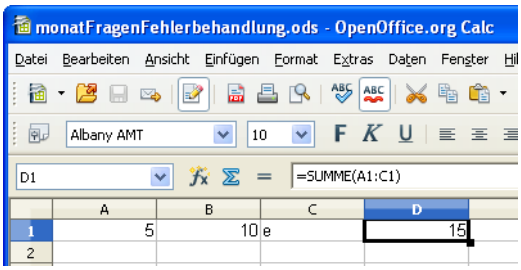


Abbildung 13.1
Fehler in Tabelleneingaben OpenOffice

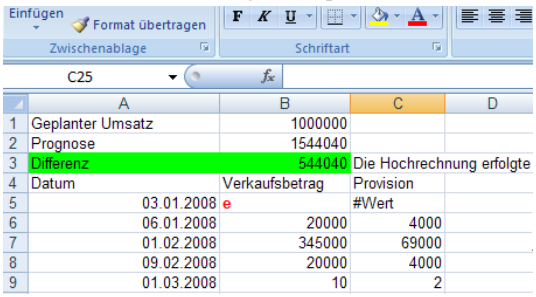


Abbildung 13.2
Fehler in Tabelleneingaben Excel

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

Neben den Benutzereingaben in den Zellen der Arbeitsblätter, kennen Sie noch zwei weitere Arten, wie unsere Programme Daten von Nutzern entgegen nehmen können:

1. über eine *MsgBox*. Hier können keine Fehleingaben auftreten, weil der Benutzer ja nur auch „OK“ oder „Abbrechen“ clicken kann. Freie, eigene Eingaben sind nicht möglich.

2. über die *Inputbox*. Und hier kann man sehr wohl Fehler machen. In unserem Provisionsbeispiel sind bei der Abfrage des Monats eigentlich nur die Zahlen 1 bis 12 erlaubt.

Abb. 13.3 zeigt die Reaktion unseres Tabellenkalkulationsprogramms auf die Eingabe von Buchstaben.
Beide Programme stürzen mit einer Fehlermeldung ab. Nun kann man sich natürlich auf den Standpunkt stellen: „Wer zu

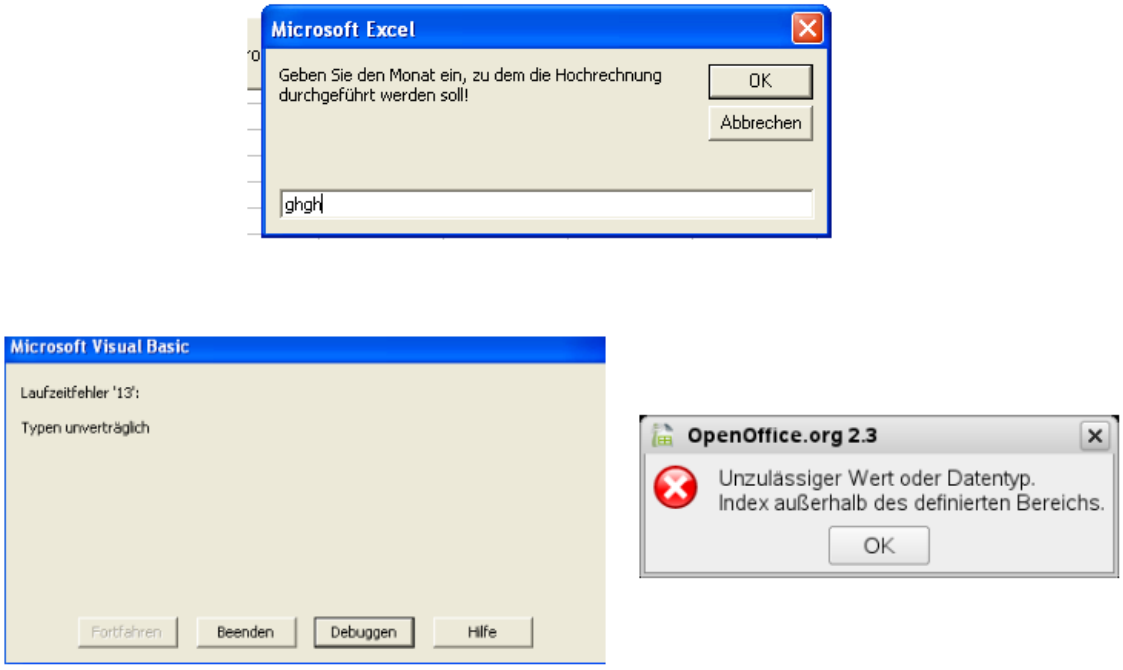


Abbildung 13.3
Fehleingabe in einer InputBox

blöd ist, eine Zahl einzugeben, wenn nach einem Monat gefragt ist, dem kann man auch einen Programmabsturz zumuten“. Andererseits ist das nicht nett ☺.
Darüber hinaus gibt es noch eine andere Fehlerquelle: Sie möchten 3 eingeben, geben aber 33 ein, weil der Finger noch mal gezuckt hat ☺, und drücken dann schnell die Return-Taste. Auch hier wird das Programm abstürzen, allerdings an einer späteren Stelle und mehr durch Zufall als durch unsere weise Voraussicht. Das Programm stürzt nämlich ab, wenn es den Namen des Monats zu seiner Nummer holen soll. In der dafür verantwortlichen Funktion (vgl. Beispiel 10.1) hatten wir ein Array mit den Monatszahlen als Index erzeugt und dann auf dieses Array mit dem übergebenen Index zugegriffen. Das funktioniert nun natürlich nicht mehr, weil 33 kein Index dieses Arrays ist und VBA dann abstürzt.
Da wäre es vielleicht doch ganz nett, wenn unser Programm nach der Eingabe prüft, ob die Eingabe so in Ordnung geht.
Wann überprüfen wir nun unsere Eingaben? Im Falle der InputBox zur Monatseingabe ist das einfach: direkt nachdem der Benutzer den Monat eingegeben hat, testen wir, ob das so sein kann 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 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.

13.1 Fehlerbehandlung und Plausibilitätsprüfungen nach Eingaben über eine *InputBox*

Sowohl in Excel als auch in OpenOffice gibt es einen Mechanismus, der das Umgehen mit Fehlern, die zu einem Programmabsturz führen, stark vereinfacht. In Excel kann man dieses Verfahren auch benutzen, um eigene Fehler zu definieren und darauf zu reagieren[1]. In OpenOffice geht das nicht. Daher und weil wir in unseren Programmen nur wenig Eingaben über *InputBoxen*[2] realisieren, stellen wir ein Verfahren vor, dass in jeder Programmiersprache, daher auch im OpenOffice-VBA geht, aber nicht ganz so schön ist.
Wir demonstrieren das Verfahren an Beispiel 13.1.

Beispiel 13.1 *Monatseingabe mit Überprüfungen*

```
sub monatFragen()
        dim monat as Integer
        dim monatString as String
        monatString=InputBox("Geben Sie den Monat (als Zahl) ein, zu dem die ➥
            Hochrechnung durchgeführt werden soll!")
        if Not wandleInIntegerUm(monatString, monat) then
                MsgBox("Die Monatseingabe muss eine Zahl sein!")
                exit sub
        end if
        if Not istGueltigerMonat(monat) then
                MsgBox("Die Monatseingabe ist nicht richtig!!")
                exit sub
        end if
        monatString=erzeugeMonatNameAusInteger(monat)
        MsgBox("Monat numerisch " & monat & "  Monat: " & monatString)
end sub

function wandleInIntegerUm(eingabe as String, rueckgabe as Integer) as Boolean
        if not isNumeric(eingabe) then
                wandleInIntegerUm=false
                exit Function
        end if
        rueckgabe=CInt(eingabe)
        wandleInIntegerUm=true
End function

function istGueltigerMonat(monat as Integer) As Boolean
        if(monat>12)or(monat<1) then
                istGueltigerMonat=false
        else
                istGueltigerMonat=true
        end if
End function

function erzeugeMonatNameAusInteger(monat As Integer) 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"
```

[1] Ein selbst definierter Fehler wäre z.B.: Eingabe negativ.
[2] Oder wie Sie in Kap. 15 lernen werden, über Dialoge.

```
                monatArray(9)="September"
                monatArray(10)="Oktober"
                monatArray(11)="November"
                monatArray(12)="Dezember"
                erzeugeMonatNameAusInteger=monatArray(monat)
        end function
```

Die erste Änderung ist: Wir lesen die Eingabe des Benutzers auf die Variable *monatString* ein. Egal was der Benutzer eingibt, VBA wird es als String interpretieren.

> Jede beliebige Zeichenkette kann als String interpretiert werden.

Ein Absturz des Programms an dieser Stelle ist nun ausgeschlossen. Unser Programm ist deswegen abgestürzt, weil wir eine Zeichenkette eingeben hatten und VBA diese Zeichenkette auf einer Variablen vom Typ Integer abspeichern wollte. Das aber führt zu einem Programmabsturz.
Wir haben in diesem Beispiel die String-Variable *monatString* zur Verfügung und können sie daher auch benutzen. Existiert in Ihrem Programm keine String-Variable die Sie zu diesem Zeitpunkt für das Einlesen nutzen können, dann deklarieren Sie einfach eine Variable für diesen Zweck und nennen diese z.B. *eingabe*.
Als nächstes wird die Funktion *wandleInIntegerUm* aufgerufen. Übergabeparameter ist zum Einen die Eingabe des Benutzers (auf der Variablen *monatString*) und zum Anderen der Monat als Zahl, wenn alles funktioniert. Die Funktion ist vom Typ *Boolean* und es wird *true* zurückgeben, wenn alles okay war, die Eingabe also eine Zahl und *false*, wenn die Eingabe keine Zahl war.
Der erste Schritt ist in der Funktion, zu prüfen, ob die Eingabe eine Zahl ist oder nicht. Dafür stellt VBA die Funktion *IsNumeric* zur Verfügung.
IsNumeric gibt:

1. *true* zurück, wenn der der Funktion übergebene Text (es wird eine Variable vom Typ String übergeben) in eine Zahl umgewandelt werden kann
2. *false* wenn der Text nicht in eine Zahl umgewandelt werden kann

Wenn *isNumeric(eingabe) false* zurückgibt, die Eingabe also keine Zahl ist, dann ist

```
if not isNumeric(eingabe) then
```

true und die Funktion verzweigt in den *Then*-Teil. Hier wird der Rückgabewert der Funktion auf *false* gesetzt und die Funktion verlassen. Ist das nicht der Fall, dann müssen wir jetzt irgendwie aus einem String eine Integer-Variable erzeugen. Dazu gibt es in VBA Konvertierungsfunktionen.

> Alle VBA-Konvertierungsfunktionen beginnen mit C und enden mit einer Drei-Buchstaben-Abkürzung des Datentyps in den umgewandelt werden soll. Die Konvertierungsfunktionen für die Umwandlung in Integer heißt *CInt* (die für Double übrigens *CDbl*).

Durch die Zeile

```
rueckgabe=CInt(eingabe)
```

wird also die Variable *rueckgabe* mit dem Wert von *eingabe*, aber im Integerformat besetzt. Dies wird auf jeden Fall funktionieren, da wir durch die Überprüfung wissen, dass auf *eingabe* eine Zahl steht (sonst wäre das Programm nicht bis hierhin gekommen). Danach wird der Rückgabewert der Funktion auf *true* gesetzt und die Funktion verlassen. Wenn also alles funktioniert hat, gibt die Funktion *true* zurück und auf dem Übergabeparameter *rueckgabe* steht der Wert des Übergabeparameters *eingabe*, allerdings im Integerformat. Ist der Wert des Übergabeparameters *eingabe* keine Zahl, gibt die Funktion *false* zurück und der Übergabeparameter *rueckgabe* ist undefiniert.
Im aufrufenden Programm werten wir den Rückgabewert der Funktion direkt in einer *if*-Anweisung aus:

```
if Not wandleInIntegerUm(monatString, monat) then
```

Ist der Rückgabewert von *wandleInIntegerUm false*, so ergibt *Not wandleInIntegerUm true*. In diesem Fall wird der *Then*-Teil ausgeführt. Der *Then*-Teil ist der Fehlerfall, wir geben also eine Fehlermeldung aus und beenden das Programm:

```
MsgBox("Die Monatseingabe muss eine Zahl sein!")
exit sub
```

Wenn *wandleInIntegerUm* hingegen *true* zurück gibt, so ist *Not wandleInIntegerUm false*. Das gesamte *if*-Konstrukt wird ignoriert und die Programmausführung wird hinter dem *if*-Konstrukt fortgesetzt. Hier wird nun die Funktion *istGueltigerMonat* aufgerufen. Diese Funktion gibt *true* zurück, wenn der ihr übergebene Wert zwischen 1 und 12 liegt, ansonsten *false*. Diese Funktion sollten Sie ohne weitere Diskussion verstehen können.
Im aufrufenden Programm wird genau wie bei *wandleInIntegerUm* verfahren, nur dass die Fehlermeldung eine andere ist.
Wir zeigen jetzt nur die Änderung des Provisionsbeispiel an der Stelle, wo wir dieses Verfahren einbauen. Die Änderung betrifft nur die Prozedur *kontrolliereUmsatzMonatlich*, bzw. *kontrolliereUmsatzMonatlich_Click* in der Excel-Lösung. Wir ersetzen dort die Zeile:

```
hochrechnungsMonat=InputBox("Geben Sie Stützmonat der Hochrechnung ein!")
```

durch:

```
hochrechnungsMonatAlsString=InputBox("Geben Sie den Monat ein, zu dem die Hochrechnung ➥
    durchgeführt werden soll!")
```

Beispiel 13.2 *Provisionsbeispiel mit Überprüfung der Monatseingabe*

```
        if Not wandleInIntegerUm(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
```

13.2 Plausibilitätsprüfungen der Eingaben im Tabellenblatt

Wie schon gesagt, regeln wir die Plausibilitätsprüfungen der Eingaben im Tabellenblatt über eine Ereignisprozedur.

13.2.1 Das Provisionsbeispiel

Beginnen wollen wir mit unserem Provisionsbeispiel. Wir erweitern die Benutzerschnittstelle, so dass sich das in Abb. 13.4 dargestellte Bild ergibt: 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 13.3 *Die Funktion istPositiveZahl*

```
        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
```

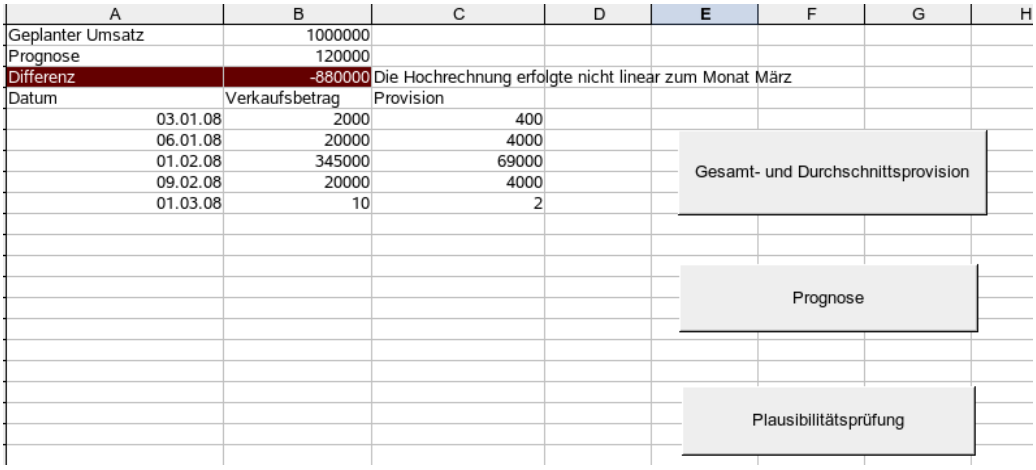


Abbildung 13.4
Erweiterung der Provisionsanwendung: Eingabeüberprüfung der Tabellendaten

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 umgeändert 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. 12 getan haben. Vielmehr werden wir die in Kap. 9 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 13.4 *Die Funktion schreibeNeuenFehlerInArray*

```
Sub schreibeNeuenFehlerInArray(fehlerString As String, fehlerArray() As String)
        Dim laengeFehlerArray As Integer
        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)
```

Realisierung in OpenOffice

[3] Wir beginnen direkt mit der Darstellung des Codes:

Beispiel 13.5 *Plausibilitätsprüfung des Provisionsbeispiels in OpenOffice*

```
sub plausibilitaeten()
        dim myDoc as Object
        dim myFirstSheet as Object
        dim cell as Object
        dim testwert as String
        dim i as Integer
        dim fehlerString as String
        dim fehlerArray() as String
        dim letzteBesetzteZeile as Integer

        const DATUMSSPALTE as Integer = 0
        const VERKAUFSBETRAGSPALTE as Integer = 1
        const ERSTE_ZEILE_MIT_VERKAUFSBETRAG as Integer = 4
        const ERSTE_ZEILE_MIT_FEHLER_INFO as Integer = 20

        Redim fehlerArray(0)
        myDoc = thisComponent
        myFirstSheet = myDoc.sheets(0)

        letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(DATUMSSPALTE, ➥
            ERSTE_ZEILE_MIT_VERKAUFSBETRAG, 0)

        for i= ERSTE_ZEILE_MIT_VERKAUFSBETRAG to letzteBesetzteZeile
                cell=myFirstSheet.getCellByPosition(DATUMSSPALTE, i)
                testwert=cell.String
                if Not isDate(testwert) then
                    fehlerString="Kein Datum in Zeile " & i + 1
                    call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
                end if
                cell=myFirstSheet.getCellByPosition(VERKAUFSBETRAGSPALTE, i)
                testwert=cell.String
                if Not istPositiveZahl(testwert) then
                    fehlerString="Keine positive Zahl als Verkaufsbetrag in Zeile ➥
                        " & i + 1
                    call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
                end if
        next i
        if Ubound(fehlerArray)>0 Then
                MsgBox("Plausibilitätsverletzungen sind aufgetreten, ein Protokoll ➥
                    wird geschrieben!")
                call schreibeFehlerArray(fehlerArray, 2, ➥
                    ERSTE_ZEILE_MIT_FEHLER_INFO, 1)
        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 Elemente eines Arrays, wenn wir nicht, wie bei dem Monatsarray aus Kap. 12, feste Indexgrenzen angeben, die Indexzählung eines Arrays immer bei Null beginnt.

[3] Excel weiter auf Seite 146

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 wir hier *i+1* als Zeile des Fehlers aufnehmen, weil OpenOffice ja, wie bereits mehrfach erwähnt, die Zeilenzählung bei 0 beginnen lässt. Wir wollen aber die wirkliche Zeile ausgeben.
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*. 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 13.6 *Die Prozedur schreibeFehlerArray in OpenOffice*

```
sub schreibeFehlerArray(fehlerArray() as String, tabellenNummer as Integer, _
                                        startzeile as Integer, spalte as Integer)
          dim laengeFehlerArray as Integer
          dim myDoc as Object
          dim mySheet as Object
          dim cell as Object
          dim i As Integer
          myDoc = thisComponent
          mySheet = myDoc.sheets(tabellenNummer)
          laengeFehlerArray=UBound(fehlerArray)

          for i = 0 to laengeFehlerArray
                    cell=mySheet.getCellByPosition(spalte, startzeile+i)
                    cell.String=fehlerArray(i)
          next i
end sub
```

Realisierung in Excel

[4] Wir beginnen direkt mit der Darstellung des Codes:

Beispiel 13.7 *Plausibilitätsprüfung des Provisionsbeispiels in Excel*

```
Sub plausibilitaeten_Click()
          Dim testwert As String
          Dim i As Integer
          Dim fehlerString As String
          Dim fehlerArray() As String
          Dim letzteBesetzteZeile As Integer

          Const DATUMSSPALTE As Integer = 1
          Const VERKAUFSBETRAGSPALTE As Integer = 2
          Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Integer = 5
          Const ERSTE_ZEILE_MIT_FEHLER_INFO As Integer = 21

          ReDim fehlerArray(0)
          letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(DATUMSSPALTE, _
                                                        ERSTE_ZEILE_MIT_VERKAUFSBETRAG➡
                                                            , 1)

          For i = ERSTE_ZEILE_MIT_VERKAUFSBETRAG To letzteBesetzteZeile
                    testwert = Cells(i, DATUMSSPALTE)
                    If Not IsDate(testwert) Then
                    fehlerString = "Kein Datum in Zeile " & i
```

[4] Excel weiter auf Seite 147

```
                                Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray)
                                End If
                                testwert = 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, 3, ➥
                                    ERSTE_ZEILE_MIT_FEHLER_INFO, 1)
                        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. 12, 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. 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*. 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.

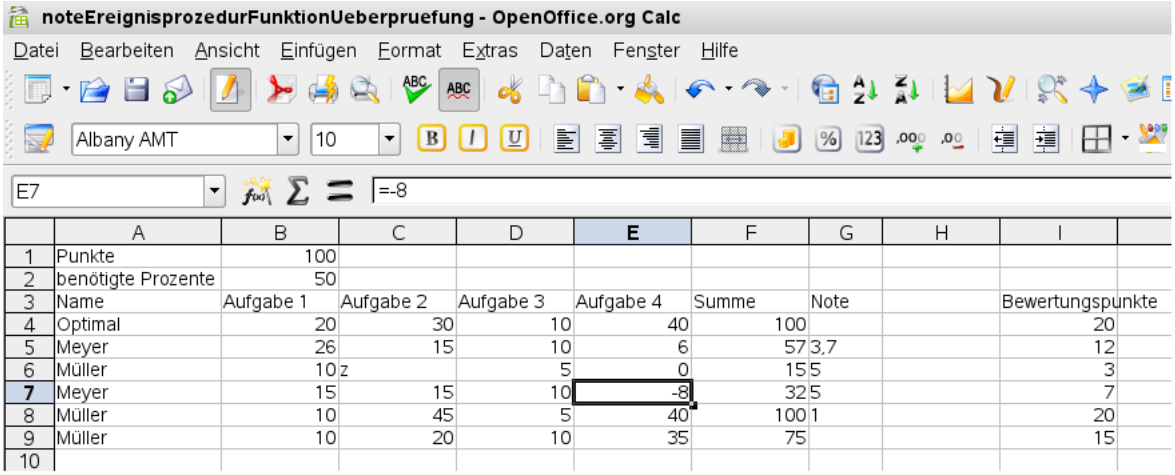
Beispiel 13.8 *Die Prozedur schreibeFehlerArray in Excel*

```
Sub schreibeFehlerArray(fehlerArray() As String, tabellenNummer As Integer, _
                                        startzeile As Integer, spalte As Integer)
          Dim laengeFehlerArray As Integer
          Dim i As Integer
          laengeFehlerArray = UBound(fehlerArray)
          For i = 0 To laengeFehlerArray
                    Sheets(tabellenNummer).Cells(startzeile + i, spalte) = fehlerArray(i)
          Next i
End Sub
```

13.2.2 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. 13.5, sollten also zu einem Fehlerprotokoll wie in Abb. 13.6 führen. Wir schreiben also



	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 13.5
Notenbeispiel mit Benutzerschnittstelle und Eingabefehlern

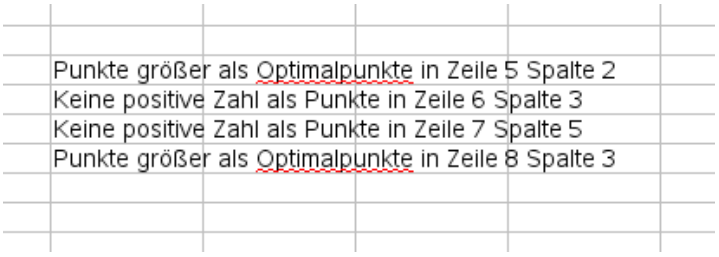


Abbildung 13.6
Fehlerprotokoll zu Abb. 13.5

zwei neue Funktionen: *istPositiveZahlNullOderLeer* und *istKleinerAlsGrenze*.

Beispiel 13.9 *Die Funktion istPositiveZahlNullOderLeer*

```
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 13.9 entspricht Beispiel 13.3.

Beispiel 13.10 *Die Funktion istKleinerAlsGrenze*

```
function istKleinerAlsGrenze(testwert as Integer, grenze as Integer) as Boolean
          if testwert <= grenze then
                    istKleinerAlsGrenze = true
          else
                    istKleinerAlsGrenze = false
          end if
end function
```

Diese Funktion ist so einfach, dass sich eine Diskussion erübrigt.

Realisierung in OpenOffice

[5]

Beispiel 13.11 *Überprüfung der Eingaben des Notenbeispiels: OpenOffice*

```
sub plausibilitaeten()
          dim myDoc as Object
          dim myFirstSheet as Object
          dim cell as Object
          dim testwert as String
          dim testwertInt as Integer
          dim i as Integer
          dim j as Integer
          dim fehlerString as String
          dim fehlerArray() as String
          dim letzteBesetzteZeile as Integer
          dim letzteBesetzteSpalte as Integer
          dim letzteSpalteMitPunkten as Integer
          dim leer as Boolean

          const OPTIMALZEILE as integer = 3
          const ERSTE_ZEILE_MIT_FEHLER_INFO as Integer = 20
```

[5] Excel weiter auf Seite 151

```
        letzteBesetzteSpalte=ermittleLetzteBesetzteSpalteInZeile(OPTIMALZEILE, 2, 0)
        letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(0, OPTIMALZEILE+1, 0)
        letzteSpalteMitPunkten=letzteBesetzteSpalte-2

        dim optimalPunkteArray(1 to letzteSpalteMitPunkten) as Integer
        Redim fehlerArray(0)
        myDoc = thisComponent
        myFirstSheet = myDoc.sheets(0)

        for i = 1 to letzteSpalteMitPunkten
                cell=myFirstSheet.getCellByPosition(i, OPTIMALZEILE)
                testwert=cell.String
                if Not istPositiveZahl(testwert) then
                        fehlerString="Keine positive Zahl als Optimalwert in ➥
                            Spalte " & i + 1
                        call schreibeNeuenFehlerInArray(fehlerString, fehlerArray➥
                            )
                else
                        optimalPunkteArray(i)=cell.Value
                end if
        next i

        for i=OPTIMALZEILE + 1 to letzteBesetzteZeile
                for j = 1 to letzteSpalteMitPunkten
                        cell=myFirstSheet.getCellByPosition(j, i)
                        testwert=cell.String
                        if Not istPositiveZahlNullOderLeer(testwert, leer) then
                                fehlerString="Keine positive Zahl als Punkte in➥
                                     Zeile " & i + 1 & " Spalte " & j + 1
                                call schreibeNeuenFehlerInArray(fehlerString, ➥
                                    fehlerArray)
                        else
                                if not leer then
                                        testwertInt=cell.Value
                                        if Not istKleinerAlsGrenze(testwert, ➥
                                            optimalPunkteArray(j)) then
                                                fehlerString="Punkte größer➥
                                                     als Optimalpunkte in ➥
                                                    Zeile " & i + 1 & " ➥
                                                    Spalte " & _
                                                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, 2, ➥
                    ERSTE_ZEILE_MIT_FEHLER_INFO, 1)
        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 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 1 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 13.5, 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 = 1 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, wenn die Zelle nicht leer war, ob der Wert der Zelle kleiner gleich dem in der Optimalzeile ist. Dazu benutzen wir das 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 nun auf eine Integervariable 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 13.5.

Realisierung in Excel

Beispiel 13.12 *Überprüfung der Eingaben des Notenbeispiels: Excel*[6]

```
Sub plausibilitaeten_Click()
        Dim testwert As String
        Dim testwertInt As Integer
        Dim i As Integer
        Dim j As Integer
        Dim fehlerString As String
        Dim fehlerArray() As String
        Dim letzteBesetzteZeile As Integer
        Dim letzteBesetzteSpalte As Integer
        Dim letzteSpalteMitPunkten As Integer
        Dim optimalPunkteArray() As Integer
        Dim leer As Boolean

        Const OPTIMALZEILE As Integer = 4
        Const ERSTE_ZEILE_MIT_FEHLER_INFO As Integer = 21

        letzteBesetzteSpalte = ermittleLetzteBesetzteSpalteInZeile(OPTIMALZEILE, 3, ➥
            1)
        letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(1, OPTIMALZEILE + ➥
            1, 1)
        letzteSpalteMitPunkten = letzteBesetzteSpalte - 2

        ReDim optimalPunkteArray(2 To letzteSpalteMitPunkten) As Integer
        ReDim fehlerArray(0)

        For i = 2 To letzteSpalteMitPunkten
                testwert = Cells(OPTIMALZEILE, i)
                If Not istPositiveZahl(testwert) Then
                        fehlerString = "Keine positive Zahl als Optimalwert in ➥
                            Spalte " & i + 1
                        Call schreibeNeuenFehlerInArray(fehlerString, fehlerArray➥
                            )
                Else
```

[6]OpenOffice weiter auf Seite 152

```
                                        optimalPunkteArray(i) = testwert
                                End End If
                        Next i

                        For i = OPTIMALZEILE + 1 To letzteBesetzteZeile
                                For j = 2 To letzteSpalteMitPunkten
                                        testwert = 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
                                                        testwertInt = Cells(i, j)
                                                        if Not istKleinerAlsGrenze(testwertInt, ➥
                                                            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, 2, ERSTE_ZEILE_MIT_FEHLER_INFO, 1)
                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 13.7, 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 = 2 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, das 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 Integervariable 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 13.7.

Kapitel 14

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. 14.1 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 von OpenOffice, bzw. Excel erzeugt wird, wenn keine Formatisierungsanweisungen programmiert werden. Dies erledigt m. E. 90% aller Fälle, da die voreingestellten Formatierungen von OpenOffice, bzw. 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 oder für Openoffice u.a. in [Martin 2005] bzw. für Excel in [Spona 2006]. Darüberhinaus verfügen beide Tabellenkalkulationen 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.

14.1 Charts in OpenOffice

[1] Zunächst möchten wir hier eine Übersicht der in dem Abschnitt “Charts in OpenOffice” verwendeten Befehle aufführen. Zum besseren Verständnis werden wir teilweise auch mehrere Befehle im Zusammenhang erläutern:
Wir zeigen sofort die Realisierung von Abb. 14.1.

Beispiel 14.1 *Grafische Darstellung der prognostizierten Umsatzentwicklung in OpenOffice*

```
    sub erstelleChart
          dim letzteBesetzteZeile as Integer
          dim anzahlMonate As Integer
          dim diagrammBreite As Integer
####### Hier die Variablen für den Chart
          dim myDoc as Object # Object anlegen
          dim mySheet as Object
          dim myDiagrams As Object
          dim myBarChartArea as New com.sun.star.awt.Rectangle
          dim myLineChartArea as New com.sun.star.awt.Rectangle
          dim myCellRange(0) as New com.sun.star.table.CellRangeAddress
```

[1] Excel weiter auf Seite 157

```
        dim myChart as Object

        const UMSATZ_START_SPALTE as Integer = 0
        const PLOT_BEREICH_END_SPALTE as Integer = 5
        const UMSATZ_START_ZEILE as Integer = 4
        const CHART_X_KOORDINATE as Integer = 10000
        const CHART_BAR_Y_KOORDINATE as Integer = 15000
        const CHART_LINE_Y_KOORDINATE as Integer = 25000
        const CHART_HOEHE As Integer = 8000
        const BREITENMULTIPLIKATOR As Integer = 5000
#######
        myDoc=thisComponent
        mySheet=myDoc.sheets(1)
        myDiagrams=mySheet.Charts

        letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(UMSATZ_START_SPALTE, _
                                                                UMSATZ_START_ZEILE➡
                                                                    , 1)
        anzahlMonate=letzteBesetzteZeile-UMSATZ_START_ZEILE
        diagrammBreite=anzahlMonate*BREITENMULTIPLIKATOR

        myBarChartArea.X=CHART_X_KOORDINATE
        myBarChartArea.Y=CHART_BAR_Y_KOORDINATE
        myBarChartArea.Width=diagrammBreite
        myBarChartArea.Height=CHART_HOEHE

        myLineChartArea.X=CHART_X_KOORDINATE
        myLineChartArea.Y=CHART_LINE_Y_KOORDINATE
        myLineChartArea.Width=diagrammBreite
        myLineChartArea.Height=CHART_HOEHE

        myCellRange(0).Sheet=1
        myCellRange(0).Startcolumn=UMSATZ_START_SPALTE
        myCellRange(0).StartRow=UMSATZ_START_ZEILE
        myCellRange(0).EndColumn=PLOT_BEREICH_END_SPALTE
        myCellRange(0).EndRow=letzteBesetzteZeile

        if myDiagrams.hasByName("barChart") then
                  myDiagrams.removeByName("barChart")
        end if
        myDiagrams.addNewByName("barChart", myBarChartArea, myCellRange, True, True)

        if myDiagrams.hasByName("lineChart") then
                  myDiagrams.removeByName("lineChart")
        end if
        myDiagrams.addNewByName("lineChart", myLineChartArea, myCellRange, True, True)
        myChart=myDiagrams.getByName("lineChart").embeddedObject
        myChart.Diagram=myChart.createInstance("com.sun.star.chart.LineDiagram")

    End sub
```

Auch hier beginnt alles mit der Deklaration von Variablen und Konstanten. Neu und vielleicht erschreckend sind die Deklarationen der für das Chart notwendigen Variablen. Aber das müssen Sie nicht verstehen, das müssen Sie nur genau so machen. Für jedes Chart benötigt man eine Fläche, in der das Chart angezeigt wird. Dies sind die Variablen *myBarChartArea* und *Chart!OpenOffice!myLineChartArea*[2]. Diese müssen genau so deklariert und erzeugt werden, wie es in Beispiel 14.1 dargestellt ist. Darüberhinaus ist natürlich auch der Zellbereich, der im Diagramm erscheinen soll notwendig. Dafür ist *myCellRange(0)* zuständig.

```
myDiagrams=mySheet.Charts
```

bereitet den Zugriff auf die in das Tabellenblatt bereits (oder auch nicht) eingebetteten Charts vor. Mit den Zeilen

```
  letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(UMSATZ_START_SPALTE, _
                                                          UMSATZ_START_ZEILE, 1)
```

[2]Wir erzeugen eine Balken- und eine Liniengrafik. Die sollen naturgemäß an anderen Orten erscheinen.

```
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. 14.1 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. Durch

```
myBarChartArea.X=CHART_X_KOORDINATE
myBarChartArea.Y=CHART_BAR_Y_KOORDINATE
myBarChartArea.Width=diagrammBreite
myBarChartArea.Height=CHART_HOEHE

myLineChartArea.X=CHART_X_KOORDINATE
myLineChartArea.Y=CHART_LINE_Y_KOORDINATE
myLineChartArea.Width=diagrammBreite
myLineChartArea.Height=CHART_HOEHE
```

werden die Orte im Tabellenblatt angegeben, an denen die beiden Diagramme angezeigt werden.

- *myBarChartArea.X* ist der Abstand vom linken Rand, *myBarChartArea.Y* selbiger vom rechten.
- *Width* und *Height* sind englische Worte, die jeder von Ihnen übersetzen können sollte.

```
myCellRange(0).Sheet=1
myCellRange(0).Startcolumn=UMSATZ_START_SPALTE
myCellRange(0).StartRow=UMSATZ_START_ZEILE
myCellRange(0).EndColumn=PLOT_BEREICH_END_SPALTE
myCellRange(0).EndRow=letzteBesetzteZeile
```

legt den Zellbereich fest, der dargestellt werden soll. Auch hier handelt es sich wieder um englische Worte die selbsterklärend sind.

```
if myDiagrams.hasByName("barChart") then
  myDiagrams.removeByName("barChart")
end if
```

schaut, ob es bereits ein Diagramm mit dem Namen *barChart* gibt. Wenn ja, wird es entfernt. In der nächsten Zeile wird durch

```
myDiagrams.addNewByName("barChart", myBarChartArea, myCellRange, True, True)
```

ein neues Diagramm mit dem Namen *barChart* hinzugefügt. Würden wir nicht vorher ein eventuell vorhandenes löschen, dann würde jeder Click auf die Schaltfläche „Grafik“ ein neues Diagramm über dem 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, aber das muss nicht immer so sein, darum gewöhnen wir uns an, eventuell bereits vorhandene Charts abzuschießen.
Die Funktion *myDiagrams.addNewByName* erwartet als Parameter den Namen des Diagramms, den Bereich des Tabellenblatts, wo es dargestellt werden soll (*myBarChartArea*), den Zellbereich, der geplottet werden soll (*myCellRange*) und zweimal *true*. Wofür diese beiden *true* stehen wissen wir zu diesem Zeitpunkt nicht, es interessiert uns aber auch nur peripher ☺. Balkendiagramme sind der voreingestellte Diagrammtyp und darum erscheint jetzt das in Abb. 14.1 dargestellte Balkendiagramm. Mit

```
if myDiagrams.hasByName("lineChart") then
        myDiagrams.removeByName("lineChart")
end if
```

löschen wir nun ein eventuell vorhandenes Diagramm mit dem Namen *lineChart*. Danach erzeugen wir durch

```
myDiagrams.addNewByName("lineChart", myLineChartArea, myCellRange, True, True)
```

ein neues Chart mit diesem Namen. Dies ist zur Zeit leider ein Balkendiagramm, darum wandeln wir es durch die Zeilen

```
myChart=myDiagrams.getByName("lineChart").embeddedObject
myChart.Diagram=myChart.createInstance("com.sun.star.chart.LineDiagram")
```

in ein Liniendiagramm um. 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: *myCellRange(0).Startcolumn* usw. 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 OpenOffice benutzt. Im nächsten Beispiel zeigen wir Ihnen, wie Sie diese ändern können. Das Chart soll jetzt wie in Abb. 14.2 dargestellt erscheinen.
Wir sehen eine Überschrift, Beschriftungen der x- und y-Achse, sowie eine Hintergrundfarbe. Beispiel 14.2 zeigt die Programmierung:

Beispiel 14.2 *Grafische Darstellung der prognostizierten Umsatzentwicklung in OpenOffice mit eigener Formatierung*

```
sub erstelleChartMitBeispielFormatierungen
      dim letzteBesetzteZeile as Integer
      dim anzahlMonate As Integer
      dim diagrammBreite As Integer
      dim monatAlsString as String
      dim myDoc as Object
      dim mySheet as Object
      dim myDiagrams As Object
      dim myBarChartArea as New com.sun.star.awt.Rectangle
      dim myCellRange(0) as New com.sun.star.table.CellRangeAddress
      dim myChart as Object

      const UMSATZ_START_SPALTE as Integer = 0
      const PLOT_BEREICH_END_SPALTE as Integer = 5
      const UMSATZ_START_ZEILE as Integer = 4
      const CHART_X_KOORDINATE as Integer = 10000
      const CHART_BAR_Y_KOORDINATE as Integer = 15000
      const CHART_HOEHE As Integer = 8000
      const BREITENMULTIPLIKATOR As Integer = 5000


      myDoc=thisComponent
      mySheet=myDoc.sheets(1)
      myDiagrams=mySheet.Charts

      letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(UMSATZ_START_SPALTE, _
                                                                  UMSATZ_START_ZEILE➡
                                                                      , 1)
      anzahlMonate=letzteBesetzteZeile-UMSATZ_START_ZEILE
      diagrammBreite=anzahlMonate*BREITENMULTIPLIKATOR

      myBarChartArea.X=CHART_X_KOORDINATE
      myBarChartArea.Y=CHART_BAR_Y_KOORDINATE
      myBarChartArea.Width=diagrammBreite
      myBarChartArea.Height=CHART_HOEHE

      myCellRange(0).Sheet=1
      myCellRange(0).Startcolumn=UMSATZ_START_SPALTE
      myCellRange(0).StartRow=UMSATZ_START_ZEILE
      myCellRange(0).EndColumn=PLOT_BEREICH_END_SPALTE
```

```
        myCellRange(0).EndRow=letzteBesetzteZeile

        if myDiagrams.hasByName("barChart") then
                myDiagrams.removeByName("barChart")
        end if
        myDiagrams.addNewByName("barChart", myBarChartArea, myCellRange, True, True)
        myChart=myDiagrams.getByName("barChart").embeddedObject
        myChart.HasMainTitle=true
        myChart.Title.String="Umsatzprognose"
        myChart.HasSubTitle=true
        monatAlsString=erzeugeMonatNameAusInteger(anzahlMonate)
        myChart.Subtitle.String="zum Monat " & monatAlsString
        myChart.Area.FillColor=RGB(255, 255, 0)
        myChart.diagram.HasXAxisTitle=true
        myChart.diagram.HasYAxisTitle=true
        myChart.diagram.XAxisTitle.String="Monat"
        myChart.diagram.YAxisTitle.String="Umsatz"
    End sub
```

Neu hinzugekommen sind die Zeilen:

```
myChart=myDiagrams.getByName("barChart").embeddedObject
myChart.HasMainTitle=true
myChart.Title.String="Umsatzprognose"
myChart.HasSubTitle=true
monatAlsString=erzeugeMonatNameAusInteger(anzahlMonate)
myChart.Subtitle.String="zum Monat " & monatAlsString
myChart.Area.FillColor=RGB(255, 255, 0)
myChart.diagram.HasXAxisTitle=true
myChart.diagram.HasYAxisTitle=true
myChart.diagram.XAxisTitle.String="Monat"
myChart.diagram.YAxisTitle.String="Umsatz"
```

Diese sind aber, glauben wir, selbsterklärend.

14.2 Charts in Excel

[3] Wir zeigen sofort die Realisierung von Abb. 14.1.

Beispiel 14.3 *Grafische Darstellung der prognostizierten Umsatzentwicklung in Excel*

```
    Sub erstelleChart_click()
        Dim letzteBesetzteZeile As Integer
        Dim anzahlMonate As Integer
        Dim diagrammBreite As Integer
        Dim myChart As Object

        Const UMSATZ_START_SPALTE As Integer = 1
        Const PLOT_BEREICH_END_SPALTE As Integer = 6
        Const UMSATZ_START_ZEILE As Integer = 5
        Const CHART_X_KOORDINATE As Integer = 250
        Const CHART_BAR_Y_KOORDINATE As Integer = 450
        Const CHART_LINE_Y_KOORDINATE As Integer = 750
        Const CHART_HOEHE As Integer = 250
        Const BREITENMULTIPLIKATOR As Integer = 150

        letzteBesetzteZeile =
              ermittleLetzteBesetzteZeileInSpalte(UMSATZ_START_SPALTE, UMSATZ_START_ZEILE, 2)
        anzahlMonate = letzteBesetzteZeile - UMSATZ_START_ZEILE
        diagrammBreite = anzahlMonate * BREITENMULTIPLIKATOR

        ActiveSheet.ChartObjects.Delete
```

[3] OpenOffice weiter auf Seite 160

```
        Set myChart = ActiveSheet.ChartObjects.Add(CHART_X_KOORDINATE, CHART_BAR_Y_KOORDINATE, ➥
            diagrammBreite, CHART_HOEHE)
        myChart.Chart.SetSourceData Source:=Sheets(2).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(2).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➥
                                                                        , 2)
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. 14.1 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, aber das muss 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(2).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(2).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. 14.3 dargestellt erscheinen.
Wir sehen eine Überschrift, Beschriftungen der x- und y-Achse sowie eine Hintergrundfarbe. Beispiel 14.4 zeigt die Programmierung:

Beispiel 14.4 *Grafische Darstellung der prognostizierten Umsatzentwicklung in Excel mit eigener Formatierung*

```
Sub erstelleChartMitBeispielFormatierungen_click()
      Dim letzteBesetzteZeile As Integer
      Dim anzahlMonate As Integer
      Dim monatAlsString As String
      Dim diagrammBreite As Integer
      Dim myChart As Object

      Const UMSATZ_START_SPALTE As Integer = 1
      Const PLOT_BEREICH_END_SPALTE As Integer = 6
      Const UMSATZ_START_ZEILE As Integer = 5
      Const CHART_X_KOORDINATE As Integer = 250
      Const CHART_BAR_Y_KOORDINATE As Integer = 450
      Const CHART_LINE_Y_KOORDINATE As Integer = 750
      Const CHART_HOEHE As Integer = 250
      Const BREITENMULTIPLIKATOR As Integer = 150

      letzteBesetzteZeile = ermittleLetzteBesetzteZeileInSpalte(UMSATZ_START_SPALTE, ➥
          UMSATZ_START_ZEILE, 2)
      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(2).Range(Cells(UMSATZ_START_ZEILE, ➥
          UMSATZ_START_SPALTE), Cells(letzteBesetzteZeile, _
                                      PLOT_BEREICH_END_SPALTE)), PlotBy:=xlColumns
      myChart.Chart.ChartType = xlColumnClustered

      monatAlsString = erzeugeMonatNameAusInteger(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

      Set myChart = ActiveSheet.ChartObjects.Add(CHART_X_KOORDINATE, CHART_LINE_Y_KOORDINATE,➥
                      diagrammBreite, CHART_HOEHE)
      myChart.Chart.SetSourceData Source:=Sheets(2).Range(Cells(UMSATZ_START_ZEILE, ➥
          UMSATZ_START_SPALTE), Cells(letzteBesetzteZeile, PLOT_BEREICH_END_SPALTE)), PlotBy➥
          :=xlColumns
      myChart.Chart.ChartType = xlLine
End Sub
```

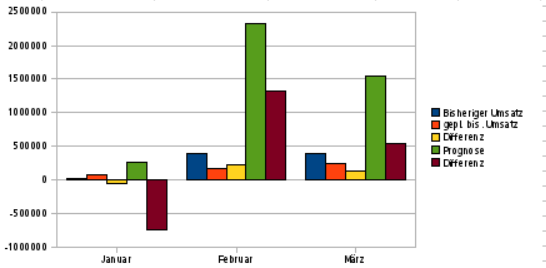
Neu hinzugekommen sind die Zeilen:

```
monatAlsString = erzeugeMonatNameAusInteger(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. 12 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.

	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			

Grafik



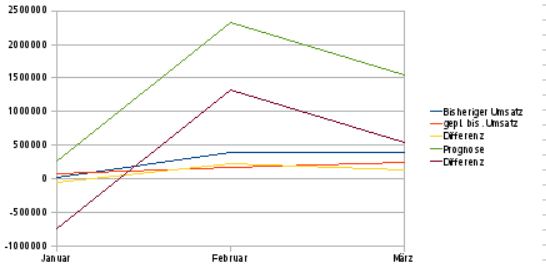


Abbildung 14.1
Automatische Charterzeugung

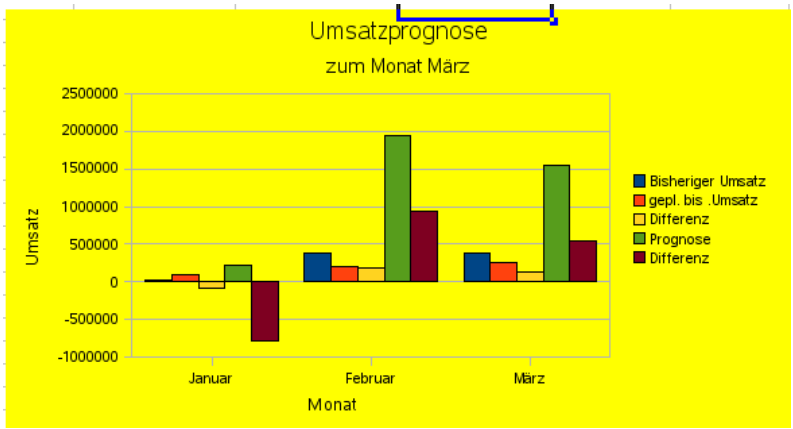


Abbildung 14.2
Eigene Formatierungen im Chart: OpenOffice

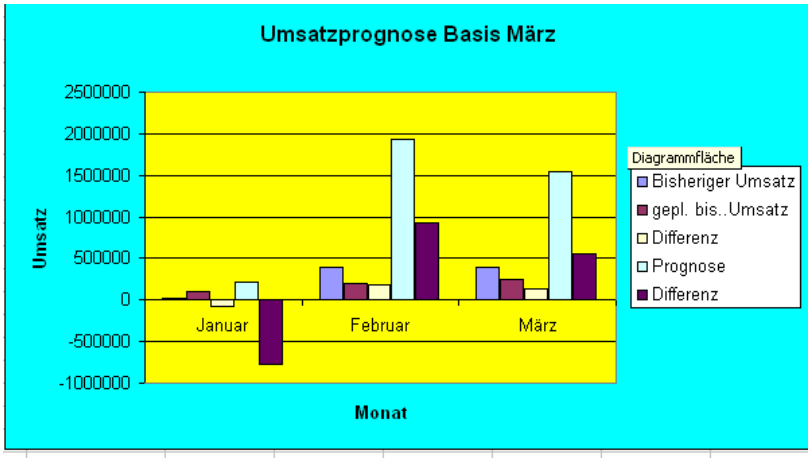


Abbildung 14.3
Eigene Formatierungen im Chart: Excel

Kapitel 15

Dialoge / Formulare

OpenOffice und Excel bieten neben den in Kap. 10 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. 15.1 dargestellte Benutzeroberfläche: Clickt der Benutzer auf

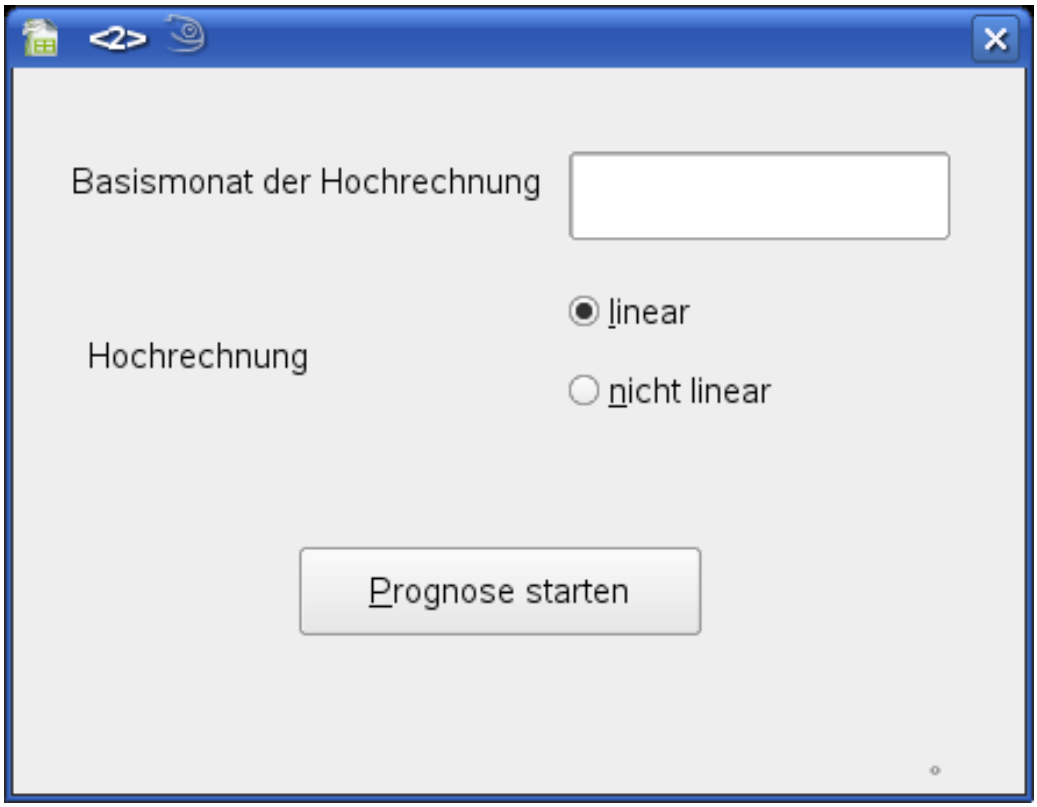


Abbildung 15.1
Der Eingabedialog des Provisionsbeispiels

die Schaltfläche „Prognose“ erscheint das in Abb. 15.1 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 (Ra-

diobutton, 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. 15.1 dargestellte, heißen in einer Tabellenkalkulation Dialog oder Formular. Wir werden beide Namen synonym verwenden.

15.1 Dialoge / Formulare in OpenOffice

[1] Um den Dialogeditor zu starten, wählen Sie zunächst Extras ➪ Makros ➪ Dialoge verwalten aus. Im nun erscheinenden Fenster clicken Sie auf das Plus neben dem Dateinamen, wählen „Standard“ aus und clicken dann auf „Neu“. Sie werden nach dem Namen des Dialogs gefragt. Geben Sie nun einen aussagekräftigen Namen ein, denn über diesen Namen wird der Dialog später angesprochen. Für unser Beispiel haben wir den Namen *prognoseDialog* vergeben. Der von Ihnen gewählte Namen erscheint unter „Standard“ und ist markiert. Clicken Sie nun auf „Bearbeiten“. Das in Abb. 15.2 dargestellte Bild erscheint. Auf der linken Seite sehen Sie die Steuerelemente-Toolbox. Sie kennen Sie schon aus Kap. 6. Denn die

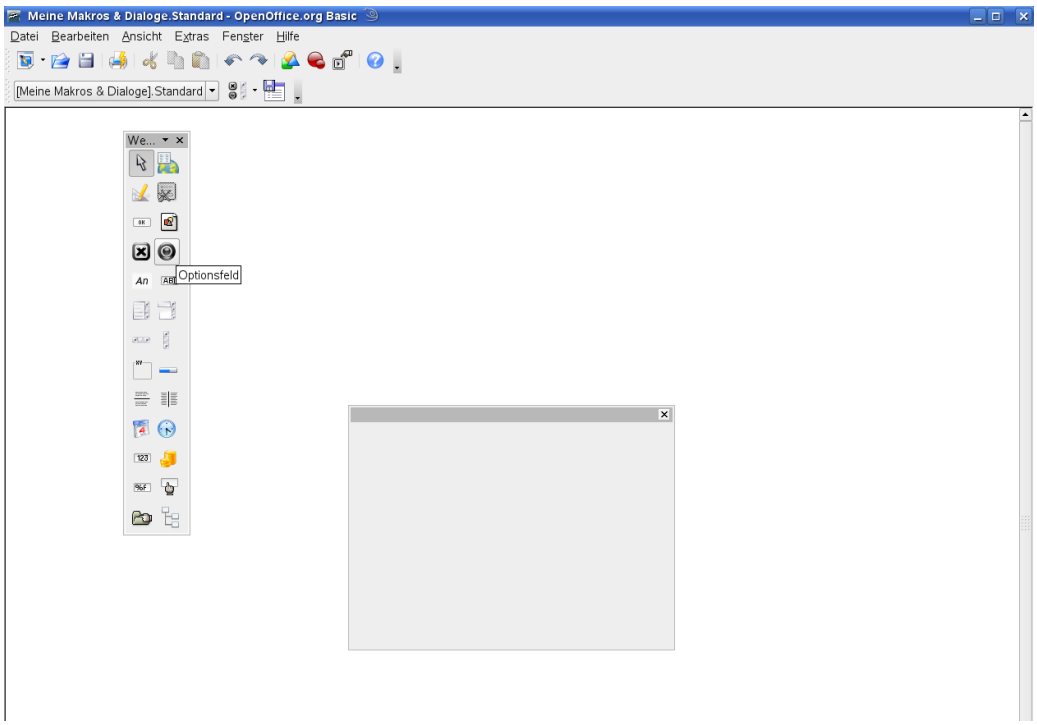


Abbildung 15.2
Der Dialogeditor in OpenOffice

Steuerelemente, die wir hier betrachten, lassen sich auch direkt in die Tabellen einfügen. Wenn Sie mit der Maus über die Symbole 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 aufgemalte Textfeld ist markiert. Sie clicken nun mit der rechten Maustaste auf das markierte Textfeld und das in Abb. 15.3 dargestellte Fenster erscheint. So etwas hatten Sie unter dem Namen Kontrollfeld bereits in Kap. 6 kennengelernt. Und in der Tat, es ist dasselbe, nur unter einem anderen Namen ☺. Die Eigenschaften des Textfelds sollten selbsterklärend sein. Für uns ist der Name wichtig. Wir nennen das Textfeld *monatInput*. Wir malen nun zwei Optionsfelder, zwei Beschriftungsfelder und eine Schaltfläche in das Formular. Die Optionsfelder bekommen die Namen *linearOption* bzw. *nichtLinearOption*. Bei den Optionsfeldern müssen wir drei weitere Eigenschaften beachten:

- Titel: Der Text neben dem Auswahlknopf. Hier vergeben wir „linear“ für das Optionsfeld *linearOption* und „nicht linear“ für das Optionsfeld *nichtLinearOption*.

[1] Excel weiter auf Seite 168

- Aktivierungsreihenfolge: Die Eigenschaft Aktivierungsreihenfolge besagt grundsätzlich in welcher Reihenfolge die Interaktionselemente den Fokus erhalten, wenn der Benutzer mit der Tabulator-Taste durch das Formular navigiert[2] Bei Optionsfelder hat die Aktivierungsreihenfolge jedoch noch eine zweite Bedeutung: Von Optionsfeldern mit unmittelbar aufeinanderfolgender Aktivierungsreihenfolge kann nur eins ausgewählt werden. Wir vergeben die Aktivierungsreihenfolgen zwei und drei.
- Status: Hier wird festgelegt, ob das Interaktionselement ausgewählt ist, wenn das Formular erscheint. Das „lineare“ Optionsfeld setzen wir auf „ausgewählt“, das „nicht lineare“ auf „Nicht ausgewählt“.
- Aktiviert: Diese Eigenschaft muss auf „Ja“ stehen, denn sonst kann man das Optionsfeld nicht aktivieren.

Bei den Beschriftungsfeldern und der Schaltfläche ist nur die Eigenschaft „Titel“ von Interesse. Dies ist die Beschriftung. Wir vergeben sie, wie in Abb. 15.1 dargestellt. Damit ist das Formular fertiggestellt.
Das Formular soll erscheinen, wenn der Benutzer die mit Prognose beschriftete Schaltfläche clickt. Das bedeutet, die mit der Prognose-Schaltfläche verbundene Ereignisprozedur *kontrolliereUmsatzMonatlich* muss geändert werden. Wir gehen hier den ganz konsequenten Weg und geben unserer alten Ereignisprozedur den neuen Namen *kontrolliereUmsatzMonatlichBerechnen*. Als nächstes definieren wir eine globale Variable. So etwas haben wir bisher nicht behandelt, weil das Konzept der globalen Variablen die Eigenständigkeit von Funktionen gefährdet. Globale Variablen sind Variablen, auf die jede Funktion Zugriff hat. Wir erzeugen eine globale Variable, indem wir sie direkt am Anfang des Moduls unterhalb der Anweisung *Option Explicit* vor der Deklaration einer jeden Funktion oder Prozedur deklarieren:

```
Option Explicit
dim meinDialog as Object
```

Die Variable *meinDialog* wird die Variable sein, auf der wir den Dialog speichern. Wir müssen hier eine globale Variable erzeugen, weil wir diese Variable sowohl in der Prozedur, in der wir den Dialog erzeugen, als auch in der Ereignisprozedur, die wir mit der „Berechnen“-Schaltfläche verbinden werden, benötigen. Jetzt können wir die Ereignisprozedur *kontrolliereUmsatzMonatlich* neu programmieren:

Beispiel 15.1 *Der Dialog wird aufgeblendet*

```
    sub kontrolliereUmsatzMonatlich()
          DialogLibraries.LoadLibrary("Standard")
          meinDialog=CreateUnoDialog(DialogLibraries.Standard.prognoseDialog)
          meinDialog.Execute()
    end sub
```

Diesen Code wollen wir nicht weiter hinterfragen. Er blendet einen Dialog auf. Das einzige was hier zum Aufblenden eines anderen Dialog geändert werden muss, ist der Name des Dialogs in der Zeile:

```
meinDialog=CreateUnoDialog(DialogLibraries.Standard.prognoseDialog)
```

Hier muss dann anstelle von *prognoseDialog* der Name eines anderen Dialogs stehen. Sie erinnern sich: Wir hatten unserem Dialog bei der Erzeugung den Namen *prognoseDialog* gegeben. Die Ereignisprozedur *kontrolliereUmsatzMonatlich* blendet nun also nur noch den in Abb. 15.1 dargestellten Dialog auf. Als nächstes müssen wir die Ereignisprozedur schreiben, die ausgeführt werden soll, wenn unsere Benutzer auf die „Prognose starten-Schaltfläche“ clicken.

Beispiel 15.2 *Die Ereignisprozedur des Dialogs in OpenOffice*

```
    sub berechnenButtonClicked()
          dim hochrechnungsMonatAlsString as String
          dim linear as Boolean
          dim hochrechnungsMonat As Integer
          hochrechnungsMonatAlsString=meinDialog.getControl("monatInput").getText()
          if Not wandleInIntegerUm(hochrechnungsMonatAlsString, hochrechnungsMonat) then
                    MsgBox("Die Monatseingabe muss eine Zahl sein!")
                    exit sub
          end if
```

[2] halt die Aktivierungsreihenfolge.

```
        if Not istGueltigerMonat(hochrechnungsMonat) then
                MsgBox("Die Monatseingabe ist nicht richtig!!")
                exit sub
        end if
        linear=meinDialog.getControl("linearOption").State
        meinDialog.endExecute()
        call kontrolliereUmsatzMonatlichBerechnen(hochrechnungsMonat, linear)
    end sub
```

Durch

```
hochrechnungsMonatAlsString=meinDialog.getControl("monatInput").getText()
```

wird in OpenOffice der Inhalt eines Textfeldes geholt. Wir sehen, wir benötigen hier die Variable auf der der Dialog gespeichert ist (*meinDialog*), sowie den Namen des Textfeldes. Das Textfeld hatten wir *monatInput* genannt. In den nächsten Zeilen werden die schon aus Kap. 13 bekannten Plausibilitätsprüfungen durchgeführt. In der Zeile

```
linear=meinDialog.getControl("linearOption").State
```

wird der Zustand des Optionsfelds *linearOption* geholt. *State* gibt *true* zurück, wenn das Optionsfeld selektiert ist, *false*, wenn es nicht selektiert ist. Dann wird die Prozedur *kontrolliereUmsatzMonatlichBerechnen* aufgerufen. Sie erinnern sich, wir hatten unsere alte Ereignisprozedur *kontrolliereUmsatzMonatlich* in *kontrolliereUmsatzMonatlichBerechnen* umbenannt. Natürlich hatten wir zu diesem Zeitpunkt noch keine Übergabeparameter definiert, weil wir ja nur den Namen *kontrolliereUmsatzMonatlich* für unsere neue Ereignisprozedur benötigten. Nun müssen wir *kontrolliereUmsatzMonatlichBerechnen* umschreiben. Diese Prozedur wird zwei Übergabeparameter bekommen. nämlich *hochrechnungsMonat* (Datentyp Integer) 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. Beispiel 15.3 zeigt den geänderten Code.

Beispiel 15.3 *Die Berechnung der Prognose aus dem Formular in OpenOffice*

```
    sub kontrolliereUmsatzMonatlichBerechnen(hochrechnungsMonat as Integer, linear as Boolean)
        dim myDoc as Object
        dim myFirstSheet as Object
        dim mySecondSheet as Object
        dim cell as Object
        dim prozentualeVerteilungArray() as Double
        dim monat as Integer
        dim alterMonat as Integer
        dim alterMonatAlsString as String
        dim letzteBesetzteZeile as Integer
        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 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 Integer

        const DATUMSSPALTE as Integer = 0
        const VERKAUFSBETRAGSPALTE as Integer = 1
        const PROVISIONSSPALTE as Integer = 2
        const ERSTE_ZEILE_MIT_VERKAUFSBETRAG as Integer = 4
        const BISHERIGER_VERKAUFSBETRAG_SPALTE=1
        const BISHERIGER_VERKAUFSBETRAG_ZEILE=1
        const PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE=3
        const PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE=1
        const GEPLANTER_UMSATZ_SPALTE=1
        const GEPLANTER_UMSATZ_ZEILE=0

        myDoc = thisComponent
        myFirstSheet = myDoc.sheets(0)
        mySecondSheet = myDoc.sheets(1)

        hochrechnungsMonatAlsString=erzeugeMonatNameAusInteger(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
        cell=myFirstSheet.getCellByPosition(GEPLANTER_UMSATZ_SPALTE,  GEPLANTER_UMSATZ_ZEILE)
        geplanterJahresumsatz=cell.Value
        geplanteProvision=berechneProvision(geplanterJahresumsatz, geplanterJahresumsatz)

        letzteBesetzteZeile=ermittleLetzteBesetzteZeileInSpalte(DATUMSSPALTE, _
                                                                  ERSTE_ZEILE_MIT_VERKAUFSBE
                                                                      , 0)
        alterMonat=1
        bisherigeProvision=0
        bisherigerUmsatz=0
        for i= ERSTE_ZEILE_MIT_VERKAUFSBETRAG to letzteBesetzteZeile + 1
                cell=myFirstSheet.getCellByPosition(DATUMSSPALTE, i)
                if cell.Type<>com.sun.star.table.CellContentType.EMPTY then
                          datum=cell.Value
                          monat=Month(datum)
                end if
                if (monat <> alterMonat) or (i = letzteBesetzteZeile + 1) then
                          'Monatswechsel Prognose und Werte müssen geschrieben werden
                          alterMonatAlsString=erzeugeMonatNameAusInteger(alterMonat)
                          if linear then
                                    prognostizierterJahresumsatz=(bisherigerUmsatz*12)/➥
                                        alterMonat
                                    geplanterUmsatzBisMonat=(geplanterJahresumsatz*alterMonat➥
                                        )/12
                                    geplanteProvisionBisMonat=(geplanteProvision*alterMonat)➥
                                        /12
                          else
                                    prognostizierterJahresumsatz=➥
                                        prognostiziereJahresumsatzNichtLinear(➥
                                        bisherigerUmsatz, _
```

```
                                                                    alterMonat, ➥
                                                                        prozentualeUmsatzverteilungArra
                                                                        )
                                        geplanterUmsatzBisMonat=bisZuDiesemMonatZuErreichen(➥
                                            geplanterJahresumsatz, _
                                                                alterMonat, ➥
                                                                    prozentualeUmsatzverteilungArray➥
                                                                    )
                                        geplanteProvisionBisMonat=bisZuDiesemMonatZuErreichen(➥
                                            geplanteProvision, alterMonat, ➥
                                            prozentualeUmsatzverteilungArray)
        end if
        realeProvisionBisMonat=berechneProvision(prognostizierterJahresumsatz, bisherigerUmsatz➥
            )
        prognostizierteProvision=berechneProvision(prognostizierterJahresumsatz, _
                                                                                prognostizierterJahresums
                                                                                    )
        umsatzMonatsDifferenz=bisherigerUmsatz-geplanterUmsatzBisMonat
        umsatzDifferenz=prognostizierterJahresumsatz-geplanterJahresumsatz
        provisionsDifferenzMonat=realeProvisionBisMonat-geplanteProvisionBisMonat
        provisionsDifferenz=prognostizierteProvision-geplanteProvision
        if umsatzDifferenz > 0 then
                farbe=rgb(0, 255, 0)
        else
                farbe=rgb(100, 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
        cell=myFirstSheet.getCellByPosition(PROVISIONSSPALTE, i)
        bisherigeProvision=bisherigeProvision + cell.Value
        cell=myFirstSheet.getCellByPosition(VERKAUFSBETRAGSPALTE, i)
        bisherigerUmsatz=bisherigerUmsatz + cell.Value
        next i
        call schreibePrognose(prognostizierterJahresumsatz, umsatzDifferenz, hochrechnungstext,➥
             farbe)
end sub
```

Zum Abschluss wechseln wir noch einmal in den Dialogeditor, um die Schaltfläche mit der Ereignisprozedur zu verbinden. Dies geschieht aber auf die gleiche Weise, wie in Kap. 6. Wir selektieren die Schaltfläche, clicken mit der rechten Maustaste und wählen „Eigenschaften“ aus. Dann wechseln wir in den Reiter *Ereignisse* und verbinden die Schaltfläche mit der in Kap. 6 beschriebenen Weise mit der Prozedur *berechnenButtonClicked*.

15.2 Dialoge / Formulare in Excel

[3] 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. 15.4 dargestellte Bild. Die UserForm sollte selektiert sein. Wenn nicht selektieren Sie sie[4]. Als erstes verändern wir die Eigenschaften des Formulars[5]. Dies geschieht im Eigenschaftsfenster. Wichtig für uns sind

[3] OpenOffice weiter auf Seite 173
[4] Was, wie ein jeder weiß, über einen Click erfolgt.
[5] Wir verwenden nun Dialog, Formular und UserForm synonym.

zunächst die Eigenschaften Name und Caption. Wir geben dem Formular den Namen *prognoseForm*[6]. 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“. Zugleich mit unserem Formular erscheint die Steuerelemente-Toolbox[7]. 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. 6. Dort sieht sie zwar komplett anders aus[8], 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[9]. 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. 15.1 dargestellt.
Die Optionsfelder müssen weiter angepasst werden. Von ihnen darf ja nur eins auswählbar sein. Dies erreichen wir, indem wir der Eigenschaft „GroupName“ bei beiden denselben Wert (im Beispiel Hochrechnungsart) geben. Abb. 15.5 zeigt das Eigenschaftsfenster des *linearOption* Optionsfelds. Die Schaltfläche bekommt ebenfalls einen Namen, wir vergeben den Namen *berechnenButton*. Damit wissen wir auch bereits (aus Kap. 6), 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. 15.6 dargestellte Fenster erscheint. Die Bedienung dieses Fensters sollte sich intuitiv erschließen[10]. 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 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[11]. Das erscheinende Codefenster ist leer. Wir schreiben sofort folgende Prozedur:

Beispiel 15.4 *Vorbelegung des Optionsfelds in Excel: Userform_initialize*

```
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

[6] Leerschritte sind in Namen von Formularen nicht erlaubt.
[7] Der Steuerelemente-Werkzeugkasten, das Steuerelemente-Menü.
[8] Wahrscheinlich, damit man sie leichter nicht wieder erkennt ☺.
[9] Wir könnten zu den Eigenschaften der Userform zurückkehren, indem wir irgendwo, wo kein Steuerelement ist, in die UserForm clicken.
[10] Welch ein schöner Satz!!!
[11] 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.

```
linearOption.Value = True
```

sorgen wir dafür, dass dieses Optionsfeld vorausgewählt ist. Nun ist das Formular fertiggestellt und wir könnten es und 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 15.5 *Die neue Ereignisprozedur der „Prognose“ Schaltfläche: Der Dialog wird aufgeblendet*

```
    Sub kontrolliereUmsatzMonatlich_Click()
          prognoseForm.Show
    End Sub
```

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. 6 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 sodann feststellen, welches Optionsfeld ausgewählt ist. Dann sollte Sie das Formular entfernen und zum Schluss die Prognoserechnung anstoßen. Die Implementierung zeigt Beispiel 15.6:

Beispiel 15.6 *Die Ereignisprozedur des Dialogs in Excel*

```
    Sub berechnenButton_Click()
          Dim linear As Boolean
          Dim hochrechnungsMonatAlsString As String
          Dim hochrechnungsMonat As Integer
          hochrechnungsMonatAlsString = monatInput.Text
          If Not wandleInIntegerUm(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
```

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 die schon aus Kap. 13 bekannten Plausibilitätsprüfungen und die Typumwandlung durchgeführt. In der Zeile

```
linear = linearOption.Value
```

wird der Zustand des Optionsfelds *linearOption* geholt. *linearOption.Value* gibt *true* zurück, wenn das Optionsfeld selektiert ist, *false*, wenn es nicht selektiert ist[12]. 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 Integer) 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 15.7 dargestellt.

Beispiel 15.7 *Die Berechnung der Prognose aus dem Formular in Excel*

```
    Sub kontrolliereUmsatzMonatlichBerechnung(hochrechnungsMonat As Integer, linear As Boolean)
          Dim prozentualeVerteilungArray() As Double
          Dim monat As Integer
          Dim alterMonat As Integer
          Dim alterMonatAlsString As String
          Dim letzteBesetzteZeile As Integer
          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 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 Integer

          Const DATUMSSPALTE As Integer = 1
          Const VERKAUFSBETRAGSPALTE As Integer = 2
          Const PROVISIONSSPALTE As Integer = 3
          Const ERSTE_ZEILE_MIT_VERKAUFSBETRAG As Integer = 5
          Const BISHERIGER_VERKAUFSBETRAG_SPALTE = 2
```

[12]Dies steht auch im Einklang mit der gerade geschriebenen Funktion *Userform_initialize*

```
Const BISHERIGER_VERKAUFSBETRAG_ZEILE = 2
Const PROZENTUALE_UMSATZVERTEILUNG_START_ZEILE = 4
Const PROZENTUALE_UMSATZVERTEILUNG_WERTE_SPALTE = 2
Const GEPLANTER_UMSATZ_SPALTE = 2
Const GEPLANTER_UMSATZ_ZEILE = 1

hochrechnungsMonatAlsString = erzeugeMonatNameAusInteger(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, 1)
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 = erzeugeMonatNameAusInteger(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, _
                                                                                    bisherigerUmsat
                                                                                        )➥

          prognostiziertеProvision = berechneProvision(prognostizierterJahresumsatz, _
                                                                  prognostizierterJahresumsatz➥
                                                                      )
          umsatzMonatsDifferenz = bisherigerUmsatz - geplanterUmsatzBisMonat
          umsatzDifferenz = prognostizierterJahresumsatz - geplanterJahresumsatz
          provisionsDifferenzMonat = realeProvisionBisMonat - geplanteProvisionBisMonat
          provisionsDifferenz = prognostiziertеProvision - 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.

Abbildung 15.3
Die Eigenschaften des Textfelds

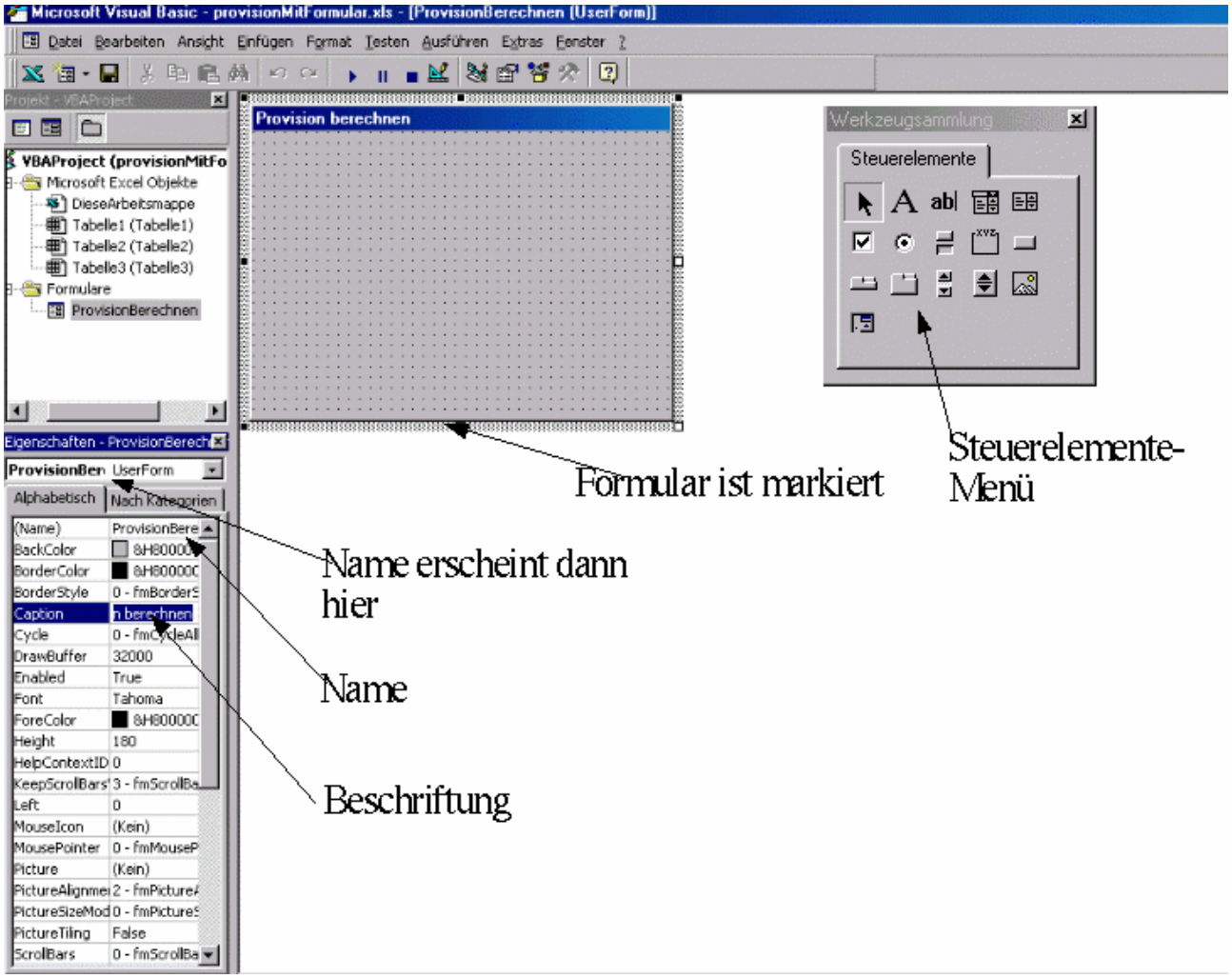


Abbildung 15.4
Der Dialogeditor in Excel

Eigenschaften - linearOption

linearOption OptionButton

Alphabetisch | Nach Kategorien

(Name)	linearOption
Accelerator	
Alignment	1 - fmAlignmentRight
AutoSize	False
BackColor	&H8000000F&
BackStyle	1 - fmBackStyleOpaque
Caption	linear
ControlSource	
ControlTipText	
Enabled	True
Font	Tahoma
ForeColor	&H80000012&
GroupName	Hochrechnungsart
Height	36
HelpContextID	0
Left	144
Locked	False
MouseIcon	(Keine)
MousePointer	0 - fmMousePointerDefault
Picture	(Keine)
PicturePosition	7 - fmPicturePositionAboveCenter
SpecialEffect	2 - fmButtonEffectSunken
TabIndex	5
TabStop	True
Tag	
TextAlign	1 - fmTextAlignLeft
Top	84
TripleState	False
Value	False
Visible	True
Width	72
WordWrap	True

Abbildung 15.5
Das Eigenschaftsfenster eines Optionsfelds

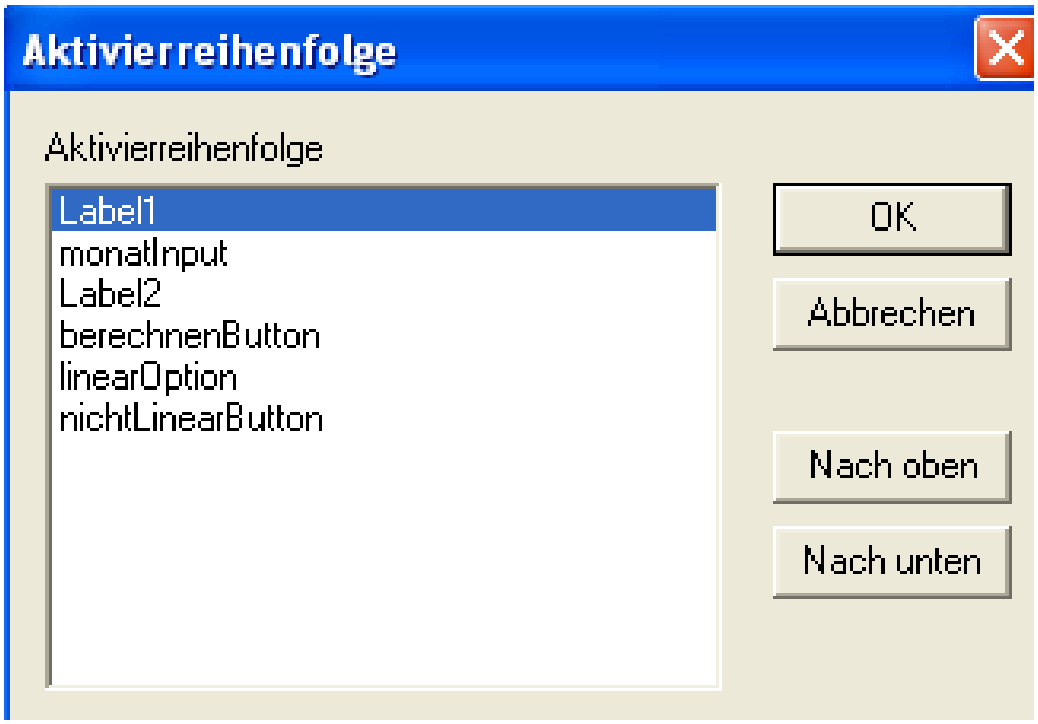


Abbildung 15.6
Aktivierreihenfolge in Excel

Kapitel 16

Zugriff auf Datenbanken

16.1 Datenbanken und OpenOffice

16.2 Datenbanken und Excel

16.2.1 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-Paket (Microsoft Office oder OpenOffice) 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. Anschließend ist eine Verbindung zur Datenbank zu konfigurieren. 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. 16.1). Nach dem 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. Wenn die Eingabefelder User und Password gefüllt werden, meldet Windows alle diese DSN benutzenden Programme unter dem voreingestellten Namen an. Ist dies nicht der Fall, muss Name und Passwort beim Verbindungsaufbau angegeben werden (vgl. Abb. 16.2). Die DSN ist nun benutzbar.

16.2.2 Nutzung der DSN von VBA

Erstes einfaches Beispiel

Wir demonstrieren die Vorgehensweise an einem Beispiel. Wir betrachten die Tabelle Kunde aus der Datenbankvorlesung des vergangenen Semesters. Die Struktur der Tabelle ist in Abb. 16.3 dargestellt. 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. 16.3 dargestellte Struktur.
- Die DSN heißt datenbankKurs.

[1] Dies natürlich nur, wenn man die notwendigen Rechte besitzt.

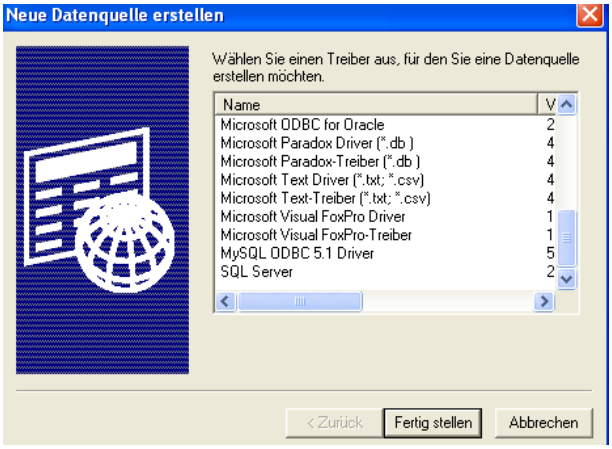


Abbildung 16.1
Auswahl des ODBC-Treibers

- 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[2] (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[3].

[2] 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.

[3] Der Punkt kommt unter M! :-))

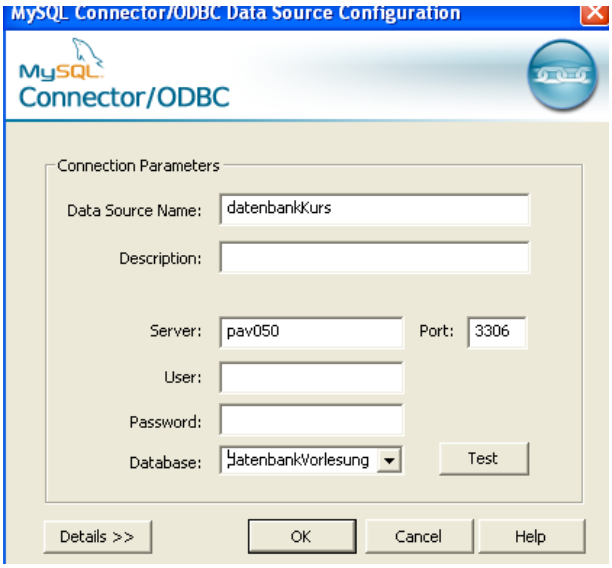


Abbildung 16.2
Einstellung der Parameter der Verbindung

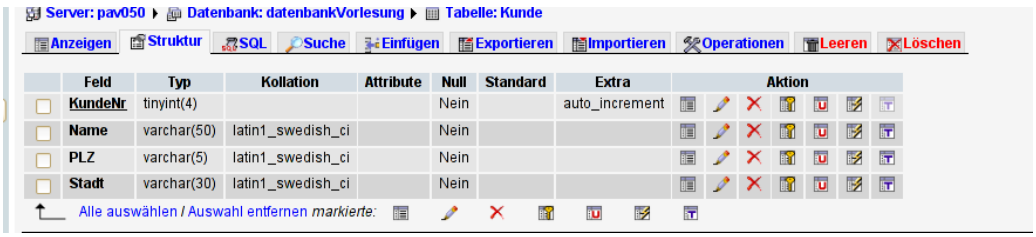


Abbildung 16.3
Die Tabelle Kunde aus der Datenbankvorlesung

Den Sourcecode der Anwendung zeigt Beispiel 16.1

Beispiel 16.1 *Lesen aus einer ODBC-Datenquelle*

```
Sub kundenAuslesen()
      Dim conn As ADODB.Connection
      Dim ConnectionString As String
      Dim rec As ADODB.Recordset
      Dim SQL As String
      Dim i As Integer

      ConnectionString = "Provider=MSDASQL.1"
      ConnectionString = ConnectionString & ";Driver=MYSQL ODBC 5.1 Driver"
      ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
      ConnectionString = ConnectionString & ";Database=datenbankVorlesung"
      ConnectionString = ConnectionString & ";DNS=datenbankKurs"
      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(1).Cells(1, 1) = "Name"
      Sheets(1).Cells(1, 2) = "PLZ"
      'Nun Inhalte
      i = 2
      Do While Not rec.EOF
                Sheets(1).Cells(i, 1) = rec!Name
                Sheets(1).Cells(i, 2) = rec!PLZ
                rec.MoveNext
                i = i + 1
      Loop
End Sub
```

Wir gehen den Code nun im Einzelnen durch.

```
Sub kundenAuslesen()
  Dim conn As ADODB.Connection
  Dim ConnectionString As String
  Dim rec As ADODB.Recordset
  Dim SQL As String
```

In unserem Programm definieren wir vier Variablen. Zunächst die eher einfachen. Um Informationen aus einer Datenbank zu bekommen, müssen wir SQL-Anweisungen an die Datenbank schicken[4]. 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 übrigen 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.
Doch gehen wir nun weiter. In den Zeilen:

```
ConnectionString = "Provider=MSDASQL.1"
ConnectionString = ConnectionString & ";Driver=MYSQL ODBC 5.1 Driver"
ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
```

[4]Müßten Sie aus dem vergangenen Semester noch wissen!

```
    ConnectionString = ConnectionString & ";Database=datenbankVorlesung"
    ConnectionString = ConnectionString & ";DNS=datenbankKurs"
    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[5]. MYSQL ODBC 5.1 Driver ist der Name des von mir installierten ODBC-Treibers. 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.

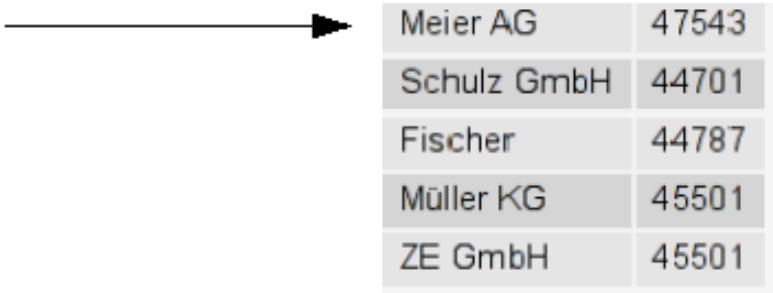
```
Set rec = New ADODB.Recordset
```

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. 16.4 illustriert einen Recordset. Ein Recordset enthält alle Datensätze, die das SQl-Kommando gefunden hat, plus einen Zeiger, der in

[5] Wie die bei Microsoft auf diesen Namen gekommen sind, ist mir auch nicht ganz klar.



Meier AG	47543
Schulz GmbH	44701
Fischer	44787
Müller KG	45501
ZE GmbH	45501

Abbildung 16.4
Grafische Darstellung eines Recordset

Abb. 16.4 durch einen Pfeil veranschaulicht wird. Zu Anfang zeigt der Zeiger (wie auch in Abb. 16.4 dargestellt) auf den ersten gefundenen Datensatz. Nun stellt sich die Frage, wie wir die Ergebnisse aus dem Recordset heraus holen. Folgende Zeile[6]

```
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
```

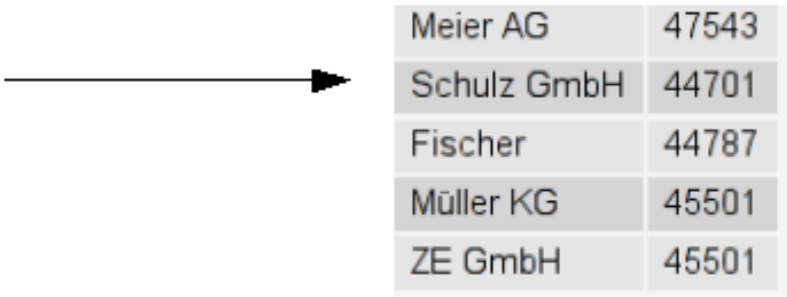
Nach diesem Kommando sieht unser Recordset, wie in Abb. 16.5 dargestellt, aus. Mit diesem Wissen können Sie jetzt den Rest des Beispiels verstehen. Die nächsten vier Zeilen sind einfach:

```
'Ueberschriften
Sheets(1).Cells(1, 1) = "Name"
Sheets(1).Cells(1, 2) = "PLZ"
i=2
```

Ein Kommentar, gefolgt von 2 Anweisungen, die Name, bzw. PLZ in die Zellen A1, resp. B1 des ersten Tabelllenblatts schreiben. Dann wird die Variable i mit dem Wert 2 initialisiert. Die nächsten Zeilen sind schwieriger:

```
Do While Not rec.EOF
          Sheets(1).Cells(i, 1) = rec!Name
          Sheets(1).Cells(i, 2) = rec!PLZ
          rec.MoveNext
          i = i + 1
Loop
```

[6] Vorausgesetzt eine Variable raumname vom Typ String ist deklariert.



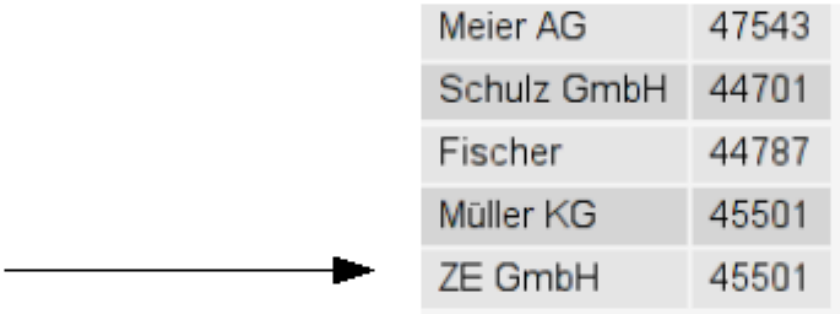
Meier AG	47543
Schulz GmbH	44701
Fischer	44787
Müller KG	45501
ZE GmbH	45501

Abbildung 16.5
Unser Recordset nach rec.moveNext

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 16.4 und 16.5 ist *rec.EOF* also *false*. Auch in Abb. 16.6 ist *rec.EOF* noch false.

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*[7] *true*. Unsere Schleife



Meier AG	47543
Schulz GmbH	44701
Fischer	44787
Müller KG	45501
ZE GmbH	45501

Abbildung 16.6
Unser Recordset mit dem Zeiger auf den letzten Datensatz

läuft also so lange, bis der Zeiger des Recordsets hinter den letzen Datensatz geschoben wird und das ist genau das, was wir wollen.

[7] EOF bedeutet übrigens End of File, EOR=End of Recordset wäre irgendwie passender gewesen.

Verbessertes praxisnahes Beispiel

Unser bisheriges Beispiel 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. 16.7 dargestellt, aus.

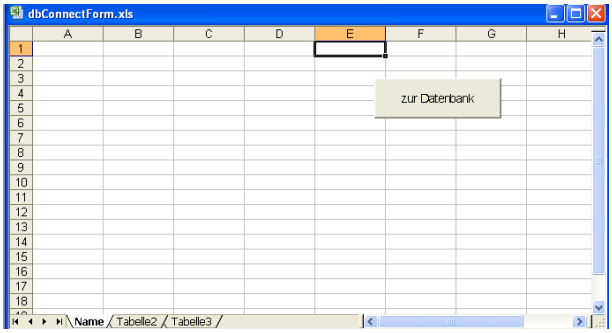


Abbildung 16.7
Startbildschirm der Anwendung

Clicken auf die Schaltfläche „zur Datenbank“ öffnet das in Abb. 16.8 dargestellte Formular.

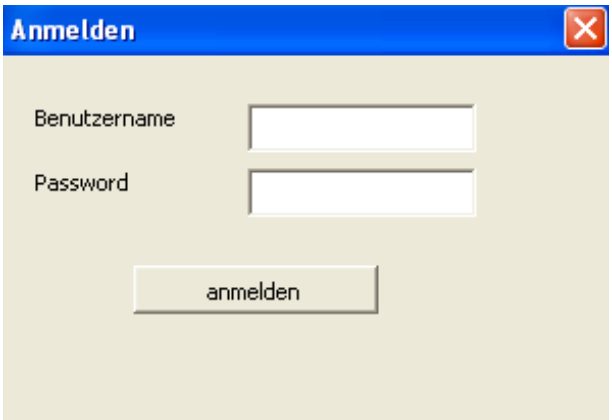
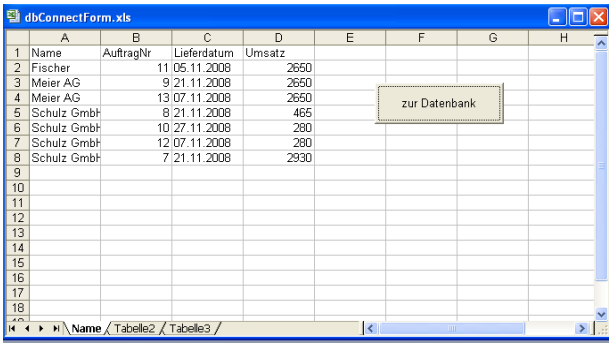


Abbildung 16.8
Das Anmeldeformular

Abb. 16.9 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. 16.10) und die Tabellenstruktur der Auftragsdatenbank aus dem Datenbank-Script.



	A	B	C	D
1	Name	AuftragNr	Lieferdatum	Umsatz
2	Fischer	11	05.11.2008	2650
3	Meier AG	9	21.11.2008	2650
4	Meier AG	13	07.11.2008	2650
5	Schulz GmbH	8	21.11.2008	465
6	Schulz GmbH	10	27.11.2008	280
7	Schulz GmbH	12	07.11.2008	280
8	Schulz GmbH	7	21.11.2008	2930

Abbildung 16.9
Die Tabelle nach Ausführung des VBA-Programms

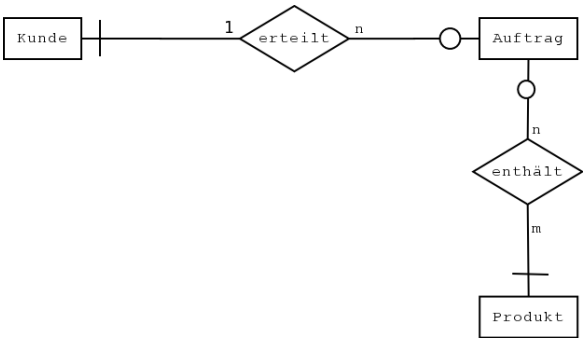


Abbildung 16.10
Das ERM der Auftragsdatenbank aus dem Datenbankscript

Name	Primärschlüssel	Weitere Felder
Kunde	KundeNr	Name, PLZ, Stadt
Auftrag	AuftragNr	AuftragDatum, Lieferdatum, *KundeNr*
Produkt	ProduktNr	Name, Preis
auftragProdukt	*AuftragNr*, *ProduktNr*	Anzahl

Die Tabellen zu 16.10

Das SQL-Kommando, um die benötigten Daten für den November 2008 aus den Tabellen zu selektieren, ist nun:

Beispiel 16.2 *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 '\%11.2008'
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 leicht 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. 16.8 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 16.3 *Die Ereignisprozedur zum Aufblenden des Anmeldeformulars*

```
Private Sub databaseConnectButton_Click()
  anmeldeForm.Show
End Sub
```

Als nächstes schreiben wir die Ereignisprozedur für die Schaltfläche auf dem Formular:

Beispiel 16.4 *Die Ereignisprozedur des Anmeldebuttons des Anmeldeformulars*

```
Private 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
```

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 16.5 *Die Prozedur Umsatzauslesen*

```
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 Integer
      Dim jahr As Integer
      Dim monatJahrString As String
      Dim i As Integer

      ConnectionString = "Provider=MSDASQL.1"
      ConnectionString = ConnectionString & ";Driver=MYSQL ODBC 5.1 Driver"
      ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
      ConnectionString = ConnectionString & ";Database=datenbankVorlesung"
      ConnectionString = ConnectionString & ";DNS=datenbankKurs"
      ConnectionString = ConnectionString & ";UID=" & benutzername
      ConnectionString = ConnectionString & ";PWD=" & passwort
```

```
        Set conn = New ADODB.Connection
        conn.ConnectionString = ConnectionString
        conn.Open
        monat = Month(Date)
        jahr = Year(Date)
        monatJahrString = "\%" & monat & "." & jahr
        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(1).Cells(1, 1) = "Name"
        Sheets(1).Cells(1, 2) = "AuftragNr"
        Sheets(1).Cells(1, 3) = "Lieferdatum"
        Sheets(1).Cells(1, 4) = "Umsatz"
        'Nun Inhalte
        i = 2
        Do While Not rec.EOF
                Sheets(1).Cells(i, 1) = rec!Name
                Sheets(1).Cells(i, 2) = rec!AuftragNr
                Sheets(1).Cells(i, 3) = rec!Lieferdatum
                Sheets(1).Cells(i, 4) = rec!Umsatz
                rec.MoveNext
                i = i + 1
        Loop
    End Sub
```

Diese Prozedur ähnelt Beispiel 16.1 sehr. Im Unterschied zu Beispiel 16.1 benutzt Beispiel 16.5 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 das aktuelle Jahr und den aktuellen Monat. Diese Informationen werden mit vorangestelltem Prozentzeichen auf der Variablen *monatJahrString* abgespeichert.

```
monat = Month(Date)
jahr = Year(Date)
monatJahrString = "\%" & monat & "." & jahr
```

In den nächsten Zeilen wird das SQL-Kommando aus Beispiel 16.2 auf die Variable *SQL* geschrieben. Anstelle '%11.2008' 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 16.1, so dass wir auf eine erneute Diskussion verzichten.

Stichwortverzeichnis