Integration des VBA-Codes in Excel-Tabellen

Bernd Blümel

Version: 17. Dezember 2002

Inhaltsverzeichnis

Kapitel 1

Integration des VBA-Codes in Excel

1.1 Benutzerdefinierte Tabellenfunktionen

1.1.1 Erstes einfaches Beispiel

Der einfachste Weg, VBA-Code von Excel aus zu nutzen, sind benutzerdefinierte Tabellenfunktionen. Alle von uns geschriebenen Funktionen tauchen nämlich in Excel unter dem Menüpukt Einfügen –> Funktionen auf. Veranschaulichen wir uns dies an einem Beispiel.

Beispiel 1.1 *Allereinfachstes Provisionsbeispiel*

```
Option Explicit
Function provision(umsatz As Double) As Double
’ Funktion berechnet Provisionen abhängig
’ vom Umsatz
’ Dateiname: provision
    Dim provisionInProzent As Double

    Const umsatzGrenze As Double = 100000

    ’ Provisionen in Prozent

    Const provisionUmsatzGrenze As Double = 10

’ bestimme Provision in Prozent

    If umsatz >= umsatzGrenze Then
        provisionInProzent = provisionUmsatzGrenze
    Else
        provisionInProzent = 0
    End If

’ Berechne auszuzahlenden Betrag

    provision = (umsatz * provisionInProzent) / 100


End Function
```

Beispiel 1.1 erwartet als Eingabe einen Double-Wert und berechnet dann die Provision für das Geschäft. Nach unserem bisherigen Kenntnisstand müßten wir jetzt ein „Hauptprogramm“ schreiben, um diese Funktion zu nutzen. Doch Funktionen können auch direkt von einer Excel-Tabelle aus genutzt werden. Tippen Sie dazu den Umsatz, von dem die Provision berechnet werden soll, in eine beliebige Excel-Zelle. Als nächstes fügen Sie in eine andere Zelle (z.B. direkt darunter) das Gleichheitszeichen ein. Dann clicken Sie zunächst das Menü Einfügen auf und wählen dort Funktionen (vgl. Abb. 1.1). Nach einem weiteren Click erscheint ein neues Fenster (vgl. Abb. 1.2). Wie durch Zauber-

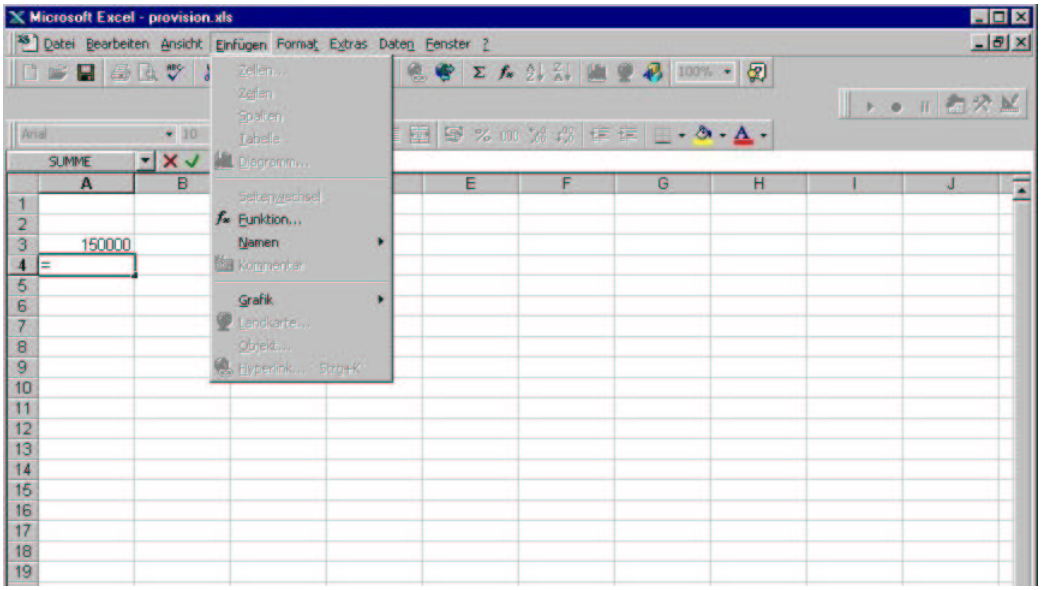

Abbildung 1.1: Das Excel Einfügen-Menü

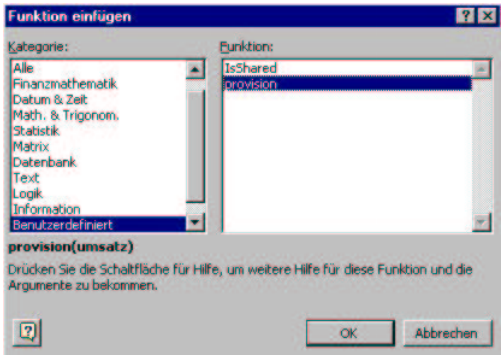

Abbildung 1.2: In Excel verfügbare Funktionen

hand existiert dort unsere Funktion provision in der Rubrik benutzerdefinierte Funktionen. Wählen Sie provision aus und bestätigen Sie (vgl. Abb. 1.3). Die Funktion provision wird in die Tabelle übernommen. Nun clicken Sie auf die Zelle, die den Umsatz enthält. Die Zelle (im Beispiel A3) wird als Parameter für die Funktion provision übernommen.

Sie lösen aus (drücken der Taste Return). Excel übergibt nun den Inhalt der Zelle, die Sie in die Parameterliste der Funktion übernommen haben (im Beispiel A3), an die Funktion. Die Funktion wird

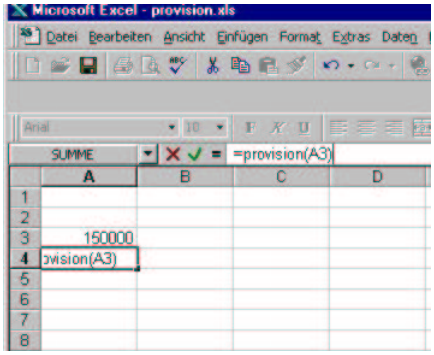

Abbildung 1.3: Provision in die Tabelle übernommen

Tabelle 1.1: Umsatz, Provisionstabelle

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

durchgeführt und der Rückgabewert der Funktion wird an Excel übergeben.

Excel stellt den Rückgabewert in der Zelle mit dem Inhalt „=provision(A3)“ dar (im Beispiel A4 der Abb. 1.3).

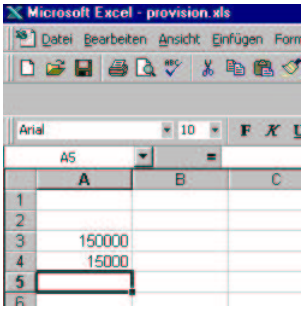

Abbildung 1.4: Provision in die Tabelle übernommen

1.1.2 Nutzung des Provisionsbeispiels aus einer Excel Tabelle

Wir ändern die Aufgabenstellung des Provisionsbeispiels folgendermaßen ab:

Problemstellung 1.1 *Provisionsberechnung aus einer Excel-Tabelle*

Ein VBA-Programm soll die zusätzliche Provision für einen Verkauf eines Vermittlers anhand des geplanten Jahresumsatzes berechnen. Die Provision soll nach Tabelle 1.1 gewährt werden: Das Programm soll als benutzerdefinierte Funktion von einer Excel-Tabelle genutzt werden. Das Benutzerinterface soll Abb. 1.5 entsprechen. Liegt der in der Zelle B2 eingegebene Verkaufsbetrag, für den die

Provision gerade berechnet werden soll, über dem geplanten Umsatz für das ganze Jahr (Eingabe Zelle B1), soll das Programm eine Fehleingabe vermuten und eine Fehlermeldung ausgeben. Ansonsten wird der Provisionbetrag errechnet und in Zelle B3 ausgegeben. Um diese Aufgabe zu lösen,

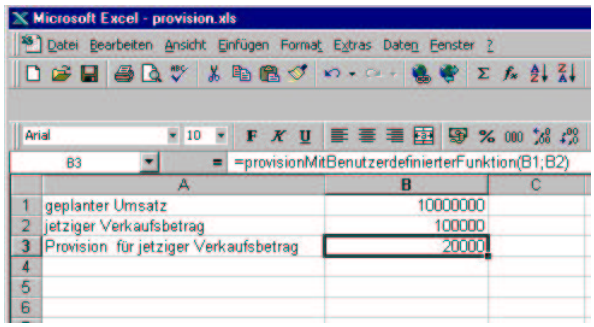

Abbildung 1.5: Das Benutzerinterface der Provisionsanwendung

betrachten wir zunächst den Pseudocode aus Kapitel 9 (VBA-Script).

Pseudocode 1.1 *Provision berechnen mit Do-While Schleife (erneut wiederholt)*

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

„Gib Programmbeschreibung aus“ können wir uns kneifen, was gemacht werden soll ist durch das in Abb. 1.5 dargestellte Benutzerinterface klar genug[1]. Um „Lies den umsatz ein“ brauchen wir uns ebenfalls kaum zu kümmern, weil die Übergabe der Parameter durch Excel erfolgt. Wir müssen nur die Prozedur „lies restliche und überprüfe alle Benutzereingaben“ in eine Funktion umwandeln. Die Übergabeparameter der neuen Funktion sind der geplante Umsatz und der jetzige Verkaufsbetrag.

Aus „lies restliche und überprüfe alle Benutzereingaben“ wird also eine Funktion, die nur die Benutzereingaben überprüft. Wir benennen sie um in: „überprüfe Benutzereingaben“.

„berechne auszuzahlenden Betrag“ ist ebenfalls ein Einzeiler. Da dies funktional eng mit „bestimme Provision in Prozent“ zusammenhängt, realisieren wir dies zusammen in einer Funktion, der wir den Namen „berechne Provision“ geben. Dies ist bereits in Kapitel 11 (VBA-Script) geschehen. Die Schleife benötigen wir nicht, da Excel bei jeder Änderung der Eingaben die benutzerdefinierte Funktion erneut ausführt.

„Gib das Ergebnis aus“ ist auch überflüssig, weil der Rückgabewert der Funktion sowieso in der Excel-Zelle, die den Funktionsaufruf enthält, dargestellt wird.

Unser Pseudocode reduziert sich also zu:

Pseudocode 1.2 *Provisionsberechnung aus einer Excel-Tabelle*

[1] So zumindest ist die Hoffnung.

```
        überprüfe Benutzereingaben
        berechne Provision
                beinhaltet:     bestimme Provision in Prozent
                                berechne auszuzahlenden Betrag
                                Formel:
                                (verkaufsbetrag*provision in prozent)/100
```

Die Funktion „berechne Provision“ haben wir bereits in Kapitel 11.9 (VBA-Script) realisiert, eine Änderung ist nicht notwendig. „überprüfe Benutzereingaben“ muß zwar aus lies restliche und überprüfe alle Benutzereingaben“ abgeleitet werden, jedoch ist dies nicht weiter schwierig, weil wir ja nur die InputBox-en streichen und dafür einen Parameter mehr übergeben müssen. Diese Gedanken führen zu folgender Realisierung:

Realisierung 1.1 *Provisionsberechnung aus einer Excel-Tabelle*

```
Function provisionMitBenutzerdefinierterFunktion(umsatzEingabe As String, _
                                                    verkaufsbetragEingabe As String)

’ Funktion berechnet Provisionen abhaengig
’ vom Umsatz
’ Dateiname: provisionMitBenutzerdefinierterFunktion

    Dim umsatz As Double
    Dim verkaufsbetrag As Double
    Dim eingabe As String

    If Not ueberpruefe_Benutzereingaben(umsatzEingabe, _
            verkaufsbetragEingabe, umsatz, verkaufsbetrag) Then Exit Function

    provisionMitBenutzerdefinierterFunktion = berechne_Provision(umsatz, _
     verkaufsbetrag)

End Function

Function ueberpruefe_Benutzereingaben(ByVal umsatzEingabe As String, _
                  ByVal verkaufsbetragEingabe As String, _
                  umsatz As Double, _
                  verkaufsbetrag As Double) _
                  As Boolean

        ueberpruefe_Benutzereingaben = True
        If Not wandle_in_Double_um(umsatzEingabe, umsatz) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der erste Parameter (umsatz) der Funktion ist keine Zahl!")
            Exit Function
        End If

        If Not wandle_in_Double_um(verkaufsbetragEingabe, verkaufsbetrag) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der zweite Parameter (verkaufsbetrag) der Funktion ist keine Zah
            Exit Function
        End If

        If verkaufsbetrag > umsatz Then
```

```
            MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
            ueberpruefe_Benutzereingaben = False
            Exit Function
        End If
End Function

 Function berechne_Provision(ByVal umsatz As Double, _
            ByVal verkaufsbetrag As Double) _
            As Double

    Dim provisionInProzent As Double

    ' Umsatzgrenzen sind DM-Betraege

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

    ' Provisionen in Prozent

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

    ' Bestimme Provision
    If umsatz >= umsatzGrenze3 Then
        provisionInProzent = provisionUmsatzGrenze3
    ElseIf umsatz >= umsatzGrenze2 Then
        provisionInProzent = provisionUmsatzGrenze2
    ElseIf umsatz >= umsatzGrenze1 Then
        provisionInProzent = provisionUmsatzGrenze1
    Else
        provisionInProzent = 0
    End If

' Berechne die Provision

    berechne_Provision = (verkaufsbetrag * provisionInProzent) / 100

End Function

Function wandle_in_Double_um(ByVal eingabe As String, rueckgabe As Double) _
                    As Boolean
    wandle_in_Double_um = True
    If Not IsNumeric(eingabe) Then
        MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
        wandle_in_Double_um = False
        Exit Function
    End If
    rueckgabe = CDbl(eingabe)
End Function
```

Wie Sie sehen, waren dies tatsächlich die einzigen Änderungen:

- Aus einer Prozedur wird eine Funktion.
- Die Überprüf-Funktion liest selber nichts mehr ein, sondern bekommt alle zu überprüfenden Variablen übergeben.
- Die Schleife verschwindet.

Beachten Sie allerdings, daß die Übergabeparameter beim Aufruf aus einer Excel-Tabelle durch das Semicolon getrennt werden und nicht wie in VBA selbst durch das Komma (vgl. Abb. 1.5).

Realisierung 1.1 hat allerdings noch einen Nachteil. In der Rubrik benutzerdefinierte Funktionen des Einfügen –> Funktionen Dialogs sehen wir alle Funktionen von Realisierung 1.1 (vgl. Abb. 1.6). So etwas kann diese Rubrik recht schnell füllen, speziell, wenn wir viele „Hilfsfunktionen“

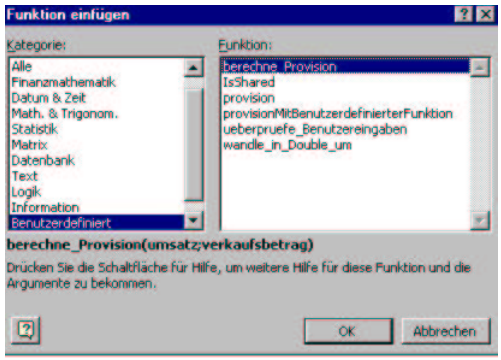

Abbildung 1.6: Die Rubrik benutzerdefinierte Funktionen nach der Implementierung von Realisierung 1.1

implementieren, von denen wir gar nicht wollen, dass sie von Excel aus zugänglich sind.

Die Sichtbarkeit von Funktionen können wir durch das Schlüsselwort Private einschränken. Funktionen, denen das Schlüsselwort Private voransteht, sind nur in dem Modul, in dem sie deklariert sind, sichtbar und erscheinen insbesondere nicht in der Rubrik benutzerdefinierte Funktionen des Einfügen –> Funktionen Dialogs. Dies entspricht den Regeln zur Sichtbarkeit von Variablen (vgl. Kapitel 11.11, VBA-Script).

Wir können also Realisierung 1.1 folgendermaßen umschreiben:

Realisierung 1.2 *Provisionsberechnung aus einer Excel-Tabelle, nur die Hauptfunktionen sichtbar*

```
Function provisionMitBenutzerdefinierterFunktion(umsatzEingabe As String, _
                                                 verkaufsbetragEingabe As String)

' Funktion berechnet Provisionen abhaengig
' vom Umsatz
' Dateiname: provisionMitBenutzerdefinierterFunktion

    Dim umsatz As Double
    Dim verkaufsbetrag As Double
    Dim eingabe As String

    If Not ueberpruefe_Benutzereingaben(umsatzEingabe, _
            verkaufsbetragEingabe, umsatz, verkaufsbetrag) Then Exit Function
```

```
    provisionMitBenutzerdefinierterFunktion = berechne_Provision(umsatz, _
     verkaufsbetrag)

End Function

Function ueberpruefe_Benutzereingaben(ByVal umsatzEingabe As String, _
                   ByVal verkaufsbetragEingabe As String, _
                   umsatz As Double, _
                   verkaufsbetrag As Double) _
                   As Boolean

        ueberpruefe_Benutzereingaben = True
        If Not wandle_in_Double_um(umsatzEingabe, umsatz) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der erste Parameter (umsatz) der Funktion ist keine Zahl!")
            Exit Function
        End If

        If Not wandle_in_Double_um(verkaufsbetragEingabe, verkaufsbetrag) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der zweite Parameter (verkaufsbetrag) der Funktion ist keine Zah
            Exit Function
        End If

        If verkaufsbetrag > umsatz Then
            MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
            ueberpruefe_Benutzereingaben = False
            Exit Function
        End If
End Function

 Function berechne_Provision(ByVal umsatz As Double, _
            ByVal verkaufsbetrag As Double) _
            As Double

    Dim provisionInProzent As Double

    ' Umsatzgrenzen sind DM-Betraege

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

    ' Provisionen in Prozent

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

    ' Bestimme Provision
    If umsatz >= umsatzGrenze3 Then
        provisionInProzent = provisionUmsatzGrenze3
    ElseIf umsatz >= umsatzGrenze2 Then
```

```
        provisionInProzent = provisionUmsatzGrenze2
    ElseIf umsatz >= umsatzGrenze1 Then
        provisionInProzent = provisionUmsatzGrenze1
    Else
        provisionInProzent = 0
    End If

' Berechne die Provision

    berechne_Provision = (verkaufsbetrag * provisionInProzent) / 100

End Function

Function wandle_in_Double_um(ByVal eingabe As String, rueckgabe As Double) _
                    As Boolean
    wandle_in_Double_um = True
    If Not IsNumeric(eingabe) Then
        MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
        wandle_in_Double_um = False
        Exit Function
    End If
    rueckgabe = CDbl(eingabe)
End Function
```

Nun wird nur noch provisionMitBenutzerdefinierterFunktion in der Rubrik benutzerdefinierte Funktionen des Einfügen –> Dialogs dargestellt. (vgl. Abb. 1.7).

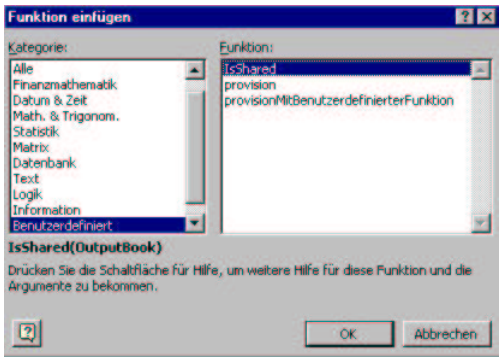

Abbildung 1.7: Die Rubrik benutzerdefinierte Funktionen nach der Implementierung von Realisierung 1.2

1.1.3 Nutzung des Zinsbeispiels aus einer Excel-Tabelle

Als nächstes wollen wir das Beispiel mit den Zinsklassen in benutzerdefinierte Funktionen umsetzen. Dabei soll ein Benutzerinterface, wie in Abb. 1.8 dargestellt, erreicht werden. Hier ist die Vorgehensweise aber ganz analog. Wenn wir die Aufgabenstellung mit der bereits erfolgten Realisierung 11.4 in Kapitel 11 (VBA-Script) vergleichen, sind die Änderungen nur marginal:

- Aus der „Steuerungsprozedur sub zinsberechnung()“ wird die Funktion

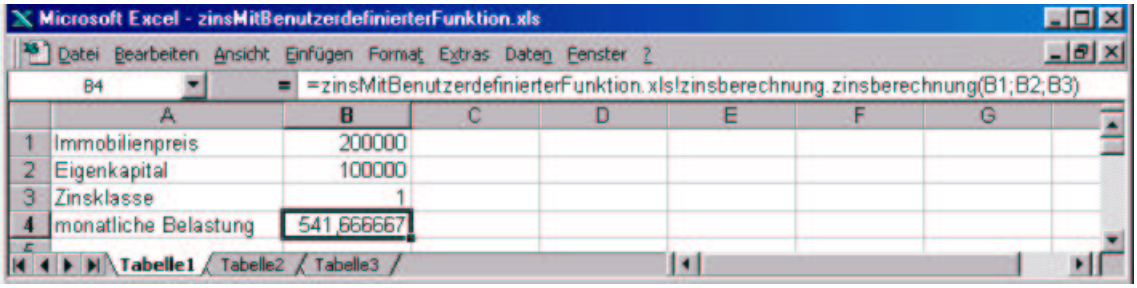

Abbildung 1.8: Zinsbeispiel mit benutzerdefinierter Funktion

```
Function zinsberechnung(immobilienpreisString As String, _
        eigenkapitalString As String, _
        zinsKlasseStString As String) _
        As Double
```

Diese Funktion erhält ihre Übergabeparameter aus der Excel-Tabelle, in die sie eingebunden wird. Das Einlesen entfällt.

- Die Überprüf-Funktion liest selber nichts mehr ein, sondern bekommt alle zu überprüfenden Variablen übergeben.
- Die Schleife verschwindet.
- Die Ausgabe erfolgt über den Funktionsnamen in der Zelle der Excel-Tabelle, in die die selbstgeschriebene Funktion eingefügt wurde.

Daher sollten Sie den nun folgenden VBA-Code verstehen können.

Realisierung 1.3 *Zinsberechnung aus einer Excel-Tabelle, nur die Hauptfunktionen sichtbar*

```
Option Explicit
Function zinsberechnung(immobilienpreisString As String, _
                    eigenkapitalString As String, _
                    zinsKlasseString As String) _
                    As Double

' Programm berechnet die monatliche Belastung
' bei einzugebendem Kaufpreis und Eigenkapital
' Dateiname: zinsMitDelbstdefinierterFunktion
    Dim eigenkapital As Double
    Dim immobilienpreis As Double
    Dim zinsKlasse As Integer
    Dim monatlicheBelastung As Double
    If Not ueberpruefe_alle_Benutzereingaben(immobilienpreisString, _
                            eigenkapitalString, zinsKlasseString, _
                                    immobilienpreis, eigenkapital, _
                                            zinsKlasse) Then Exit Function
    If Not fuehre_Berechnungen_durch(eigenkapital, immobilienpreis, _
            zinsKlasse, monatlicheBelastung) Then Exit Function
    zinsberechnung = monatlicheBelastung
End Function
```

```
Private Function berechne_Eigenkapitalquote(ByVal eigenkapital As Double, _
                ByVal immobilienpreis As Double) _
                As Boolean

    Dim eigenkapitalquote As Double
    berechne_Eigenkapitalquote = True

    eigenkapitalquote = (eigenkapital / immobilienpreis) * 100
    If eigenkapitalquote < 30 Then
        MsgBox ("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                "% ist zu niedrig! ")
        berechne_Eigenkapitalquote = False
        Exit Function
    End If
End Function
Private Function Monatliche_Belastung_berechnen(ByVal immobilienpreis As Double,

_
                  ByVal eigenkapital As Double, _
                  ByVal zins As Double) _
                  As Double
    Const tilgung As Double = 1
    Dim aufzunehmenderBetrag As Double
    Dim jahresBelastung As Double
    Dim eigenkapitalquote As Double
    aufzunehmenderBetrag = immobilienpreis - eigenkapital
    jahresBelastung = (aufzunehmenderBetrag / 100) * (zins + tilgung)
    Monatliche_Belastung_berechnen = jahresBelastung / 12
End Function
Private Function ueberpruefe_alle_Benutzereingaben _
                      (ByVal immobilienpreisString As String, _
                       ByVal eigenkapitalString As String, _
                       ByVal zinsKlasseString As String, _
                      immobilienpreis As Double, _
                      eigenkapital As Double, _
                      zinsKlasse As Integer) _
                      As Boolean

        ueberpruefe_alle_Benutzereingaben = True
        If Not wandle_in_Double_um(immobilienpreisString, immobilienpreis) Then
            ueberpruefe_alle_Benutzereingaben = False
            MsgBox ("Immobilienpreis muss eine Zahl sein!")
            Exit Function
        End If

        If Not wandle_in_Double_um(eigenkapitalString, eigenkapital) Then
            ueberpruefe_alle_Benutzereingaben = False
            MsgBox ("Eigenkapital muss eine Zahl sein!")
            Exit Function
        End If

        If Not wandle_in_Integer_um(zinsKlasseString, zinsKlasse) Then
            ueberpruefe_alle_Benutzereingaben = False
            MsgBox ("Zinsklasse muss eine Zahl sein!")
```

```
            Exit Function
        End If
End Function
Private Function wandle_in_Double_um(ByVal eingabe As String, rueckgabe As
Double) _
                    As Boolean
    wandle_in_Double_um = True
    If Not IsNumeric(eingabe) Then
        MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
        wandle_in_Double_um = False
        Exit Function
    End If
    rueckgabe = CDbl(eingabe)
End Function
Private Function wandle_in_Integer_um(ByVal eingabe As String, rueckgabe As
Integer) _
                           As Boolean
    wandle_in_Integer_um = True
    If Not IsNumeric(eingabe) Then
        MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
        wandle_in_Integer_um = False
        Exit Function
    End If
    rueckgabe = CInt(eingabe)
End Function
Private Function fuehre_Berechnungen_durch(ByVal eigenkapital As Double, _
                    ByVal immobilienpreis As Double, _
                    ByVal zinsKlasse As Integer, _
                    monatlicheBelastung As Double) _
                    As Boolean

    Const zinsKlasse1 As Double = 5.5
    Const zinsKlasse2 As Double = 5.3
    Const zinsKlasse3 As Double = 5.2
    Const zinsKlasse4 As Double = 5#
    Const zinsKlasse5 As Double = 4.5

    fuehre_Berechnungen_durch = True

    If Not berechne_Eigenkapitalquote(eigenkapital, immobilienpreis) Then
        fuehre_Berechnungen_durch = False
        Exit Function
    End If
    Select Case zinsKlasse
        Case 1
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse1)
        Case 2
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse2)
        Case 3
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse3)
        Case 4
```

```
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse4)
        Case 5
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse5)
        Case Else
            MsgBox ("Sie haben eine falsche Zinsklasse eingegeben!  " & _
                        "Zinsklasse muß kleiner gleich 5 sein! ")
            fuehre_Berechnungen_durch = False
            Exit Function
    End Select
End Function
```

1.2 Formulare

Eine weitere Möglichkeit, VBA-Code in Excel einzubinden, bilden Formulare. Mit Formularen können wir „beliebig“ komfortable Benutzerschnittstellen schaffen. Ich demonstriere dies zum besseren Verständnis sofort an einem Beispiel.

1.2.1 Nutzung des Provisionsbeispiels mit einem Formular

Nehmen wir an, wir wollten unser Provisionsbeispiel ändern und die Benutzereingaben in der in Abb. 1.9 dargestellten Maske erfassen und die berechnete Provision ebendort ausgeben. Die Berechnung

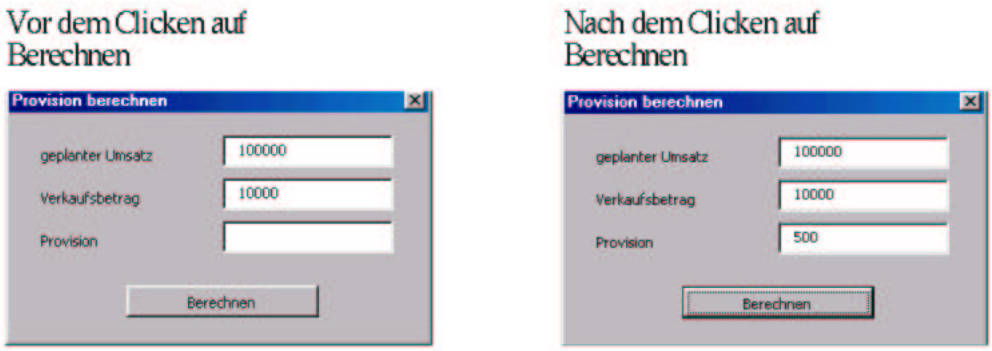

Abbildung 1.9: Ein- und Ausgabe des Provisionsbeispiels über ein Formular

soll starten, wenn der Benutzer auf den mit „Berechnen“ beschrifteten Button clickt.

Auch solche Anforderungen sind in Excel leicht realisierbar. Excel stellt dazu Möglichkeiten bereit, Formulare der in Abb. 1.9 dargestellten Art zu entwickeln und mit VBA-Code zu verbinden.

Starten wir mit dem Entwurf des Formulars. Ein Formular wird erzeugt, indem im VBA-Editor eine neue „Userform“ angelegt wird. Dies geschieht, wie Abb. 1.10 zeigt, über Einfügen–>UserForm. Es erscheint das in Abb. 1.11 dargestellte Bild. Die UserForm sollte selektiert sein. Wenn nicht selektieren Sie sie[2]. Als erstes verändern wir die Eigenschaften des Formulars[3]. Dies geschieht im

[2] Was, wie ein jeder weiß, über einen Click erfolgt.

[3] Ich verwende nun Formular und UserForm synonym.

Abbildung 1.10: Erzeugen einer UserForm

Eigenschaftsfenster. Wichtig für uns sind zunächst die Eigenschaften Name und Caption. Wir geben dem Formular den Namen ProvisionBerechnen[4]. 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 „Provision berechnen“. Zugleich mit unserem Formular erscheint die Steuerelemente-Toolbox[5]. Hier

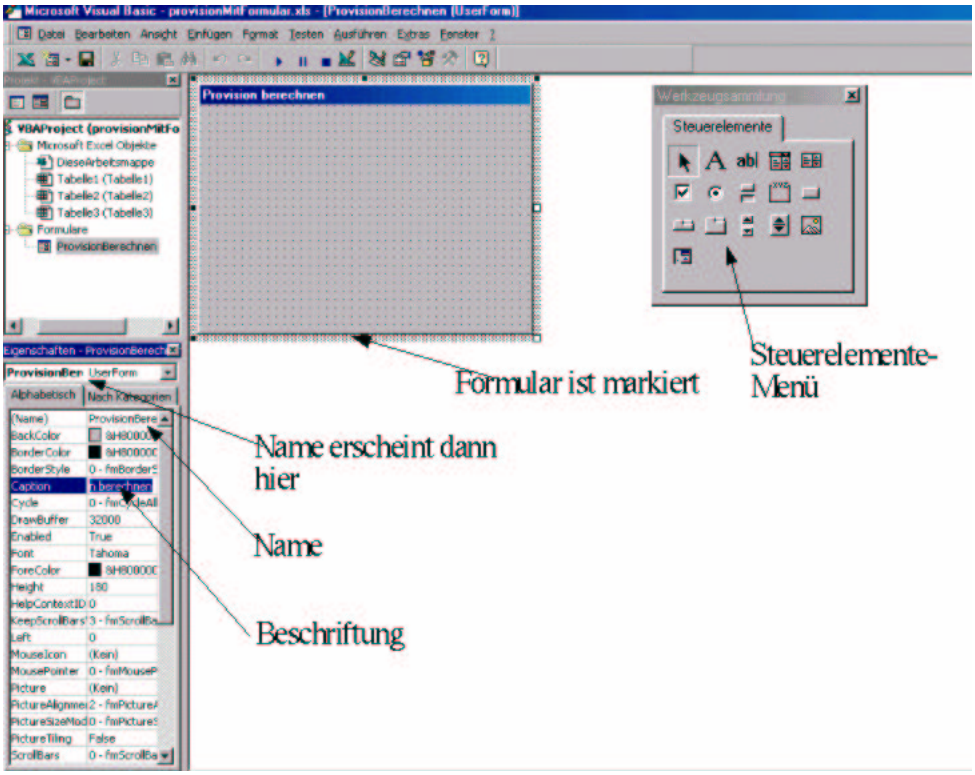

Abbildung 1.11: Erzeugen einer UserForm: Die Darstellung

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. Abb. 1.12 zeigt die Steuerelemente-Toolbox mit Pfeilen auf die in unserem Beispiel genuzten Steuerelemente. Dies sind Ein- und Ausgabefelder (TextBox), Beschriftungen (Label) und einen Button (Button), um die Berechnung anzustoßen.

Als erstes erstellen wir ein Eingabefeld. Dies geschieht, indem wir auf das TextBox-Icon in der

[4] Leerschritte sind in Namen von Formularen nicht erlaubt.

[5] Der Steuerelemente-Werkzeugkasten, das Steuerelemente-Menü.

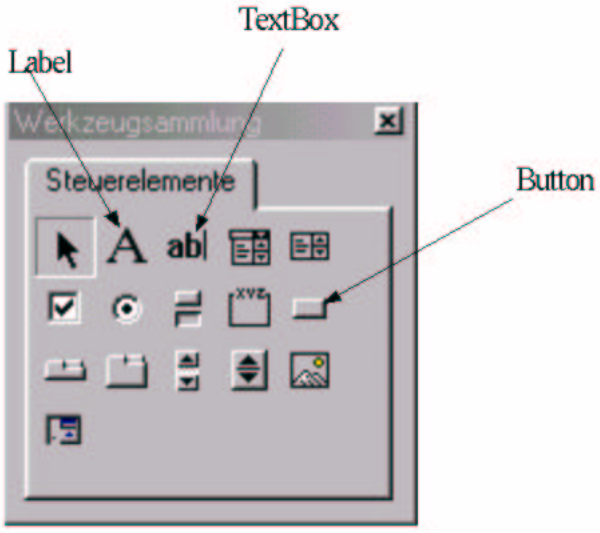

Abbildung 1.12: Die Steuerelemente-Toolbox

Steuerelemente-Toolbox clicken und dann das Eingabefeld in das Formular malen. Das Eingabefeld bleibt selektiert und das Eigenschaftsfenster zeigt nun die Eigenschaften der Textbox an[6]. Auch hier ändern wir den Namen und nennen das Eingabefeld UmsatzInput (vgl. Abb. 1.13). Völlig analog fügen wir eine Beschriftung des ersten Eingabefeldes zum Formular hinzu. Wir clicken auf das Label-Icon (vgl. Abb. 1.12) und malen das Label in das Formular. Hier müssen wir die Beschriftung ändern. Dies tun wir, indem wir die Eigenschaft Caption des Labels im Eigenschaften-Fenster auf „geplanter Umsatz“ setzen. Den Namen ändern wir nicht, da wir das Label aus dem VBA-Code nicht ansprechen werden(vgl. Abb. 1.14). Auf gleiche Art und Weise fügen wir zwei weitere Ein- bzw. Ausgabefelder[7] und Beschriftungen hinzu. Die Eingabefelder nennen wir VerkaufsbetragInput und ProvisionInput.

Als nächstes wird der Button dem Formular hinzugefügt. Dazu clicken wir auf das Button-Icon (vgl. Abb. 1.12) und malen den Button in das Formular. Hier ändern wir sowohl den Namen (Name im Eigenschaften-Fenster) in BerechnenButton, als auch die Beschriftung (Caption im Eigenschaften-Fenster) in „Berechnen“ (vgl. Abb. 1.15. Damit ist das Formular fertig. Wir sind aber noch nicht ganz zufrieden mit unserem Formular. Wenn wir das Formular jetzt testen würden,[8] gibt es noch eine „Unschönheit“. Wenn wir nämlich den Cursor in das erste Eingabefeld stellen, dort einen Umsatz eingeben und dann die Tabulator-Taste drücken, landen wir nicht, wie erwartet, im Verkaufsbetrag-Eingabefeld, der Cursor blinkt vielmehr in seiner Beschriftung. Das ist nicht wirklich schön, weil man da nichts eingeben kann. Dies kann man aber leicht ändern, indem man mit der rechten Maustaste auf das Formular clickt und den Menüpunkt Aktivierreihenfolge anwählt. Dann erscheint das in Abb. 1.16 dargestellte Fenster. Die Bedienung dieses Fensters sollte sich intuitiv erschließen[9]. Nun haben wir unser Formular erstellt. Wir können das Benutzerinterface auch bereits testen. Dies geschieht, indem wir das Formular selektieren[10] und dann die Taste F5 drücken. Excel schaltet auf die Tabellenansicht

[6] Wir könnten zu den Eigenschaften der Userform zurückkehren, indem wir irgendwo, wo kein Steuerelement ist, in die UserForm clicken.

[7] Dies macht bei Excel-Formularen keinen Unterschied

[8] Was Sie ja noch gar nicht können.

[9] Welch ein schöner Satz!!!

[10] Mit einem Click!

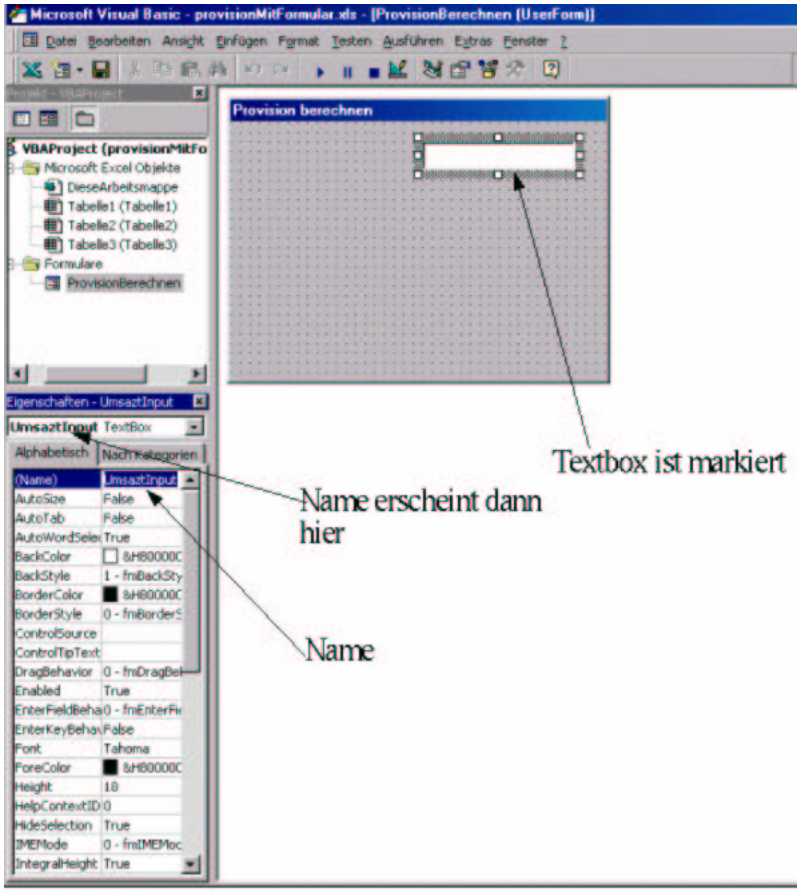

Abbildung 1.13: Hinzufügen eines Eingabefeldes (einer TextBox)

um und unser Formular erscheint vor der Tabelle (vgl. Abb. 1.17).

Zum Schluss stellt sich die Frage: Wie verbinden wir dieses Formular mit VBA-Code? Die Antwort ist: Durch ereignisgesteuerte Prozeduren. Dies klingt jetzt ziemlich schwierig, ist es aber nicht wirklich. Ereignisgesteuerte Prozeduren beruhen[11] auf Ereignissen. Und Ereignisse sind für Excel im weitetesten Sinne alle „Dinge“, die Sie in Excel durchführen.

Wenn Sie die Zahl 20 in einer Tabelle in die Zelle A1 eintragen, dann ist das für Excel das Ereignis: „Benutzer hat den Wert von Zelle A1 geändert.[12]“ Auf dieses Ereignis reagiert Excel, indem es alle Zellen, die die Zelle A1 referenzieren, anpasst. Dieses Verhalten von Excel ist auch nicht änderbar. Bei Steuerelementen verhält es sich anders. Steuerelemente können ebenfalls Ereignisse auslösen. Hier können wir aber bestimmen, was Excel machen soll, wenn ein Steuerelement ein Ereignis feststellt[13].

So ist z.B. das Clicken auf einen Button in einem Formular ein Ereignis, auf das Excel reagieren kann. Und genau dies werden wir ausnutzen, denn das ist ja genau das, was wir haben wollen. Der Benutzer clickt auf den von uns erstellten “Berechnen“ Knopf. Dann soll VBA die Provision berechnen

[11] Welch eine Erkenntnis

[12] Auch ein Neueintrag ist eine Änderung!

[13] oder auslöst, wie auch immer man will.

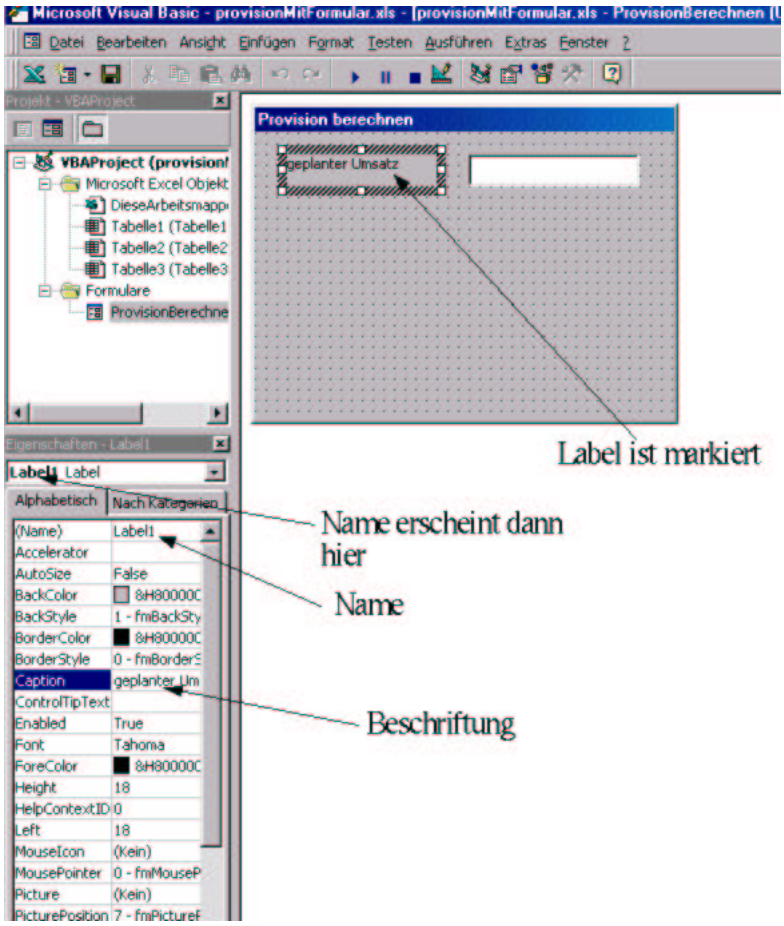

Abbildung 1.14: Hinzufügen einer Beschriftung (eines Labels)

und in das ProvisionsInput[14] wird die berechnete Provision eingetragen.

Damit ändert sich unsere Fragestellung in: Wie sagen wir Excel, reagiere auf dieses Button-Click-Ereignis, indem Du unseren VBA-Code ausführst? Dies ist aber ziemlich einfach: Wir clicken in der VBA-Entwicklungsumgebung doppelt auf unseren Button. Wie von Geisterhand geht ein Codefenster auf. Das Codefenster enthält auch bereits VBA-Code nämlich:

```
Private Sub BerechnenButton_Click()

End Sub
```

BerechnenButton war der Name, den wir dem Button bei der Erstellung des Formulars gegeben hatten. Der Zusatz "_Click()" deutet bereits darauf darauf hin, was passieren wird, wenn ein Benutzer unseren Button clickt. Dann nämlich wird der VBA-Code in der Prozedur BerechnenButton_Click() ausgeführt. Diese Prozedur müssen wir nun mit VBA-Code füllen. Die Prozedur BerechnenButton_Click() heißt übrigens Ereignisprozedur, weil sie erst dann ausgeführt wird, wenn das Ereignis „Button geclickt" eintritt.

Dazu schauen wir uns noch einmal die „Hauptfunktion"[15] unseres als benutzerdefinierte Funktion realisiertens Provisionsbeispiel an:

[14] Irreführende Bezeichnung, aber wie gesagt, Excel unterscheidet nicht zwischen Ein- und Ausgabefeldern.

[15] Damit meine ich die Funktion, die die Benutzereingabe erledigt und dann die anderen Funktionen aufruft.

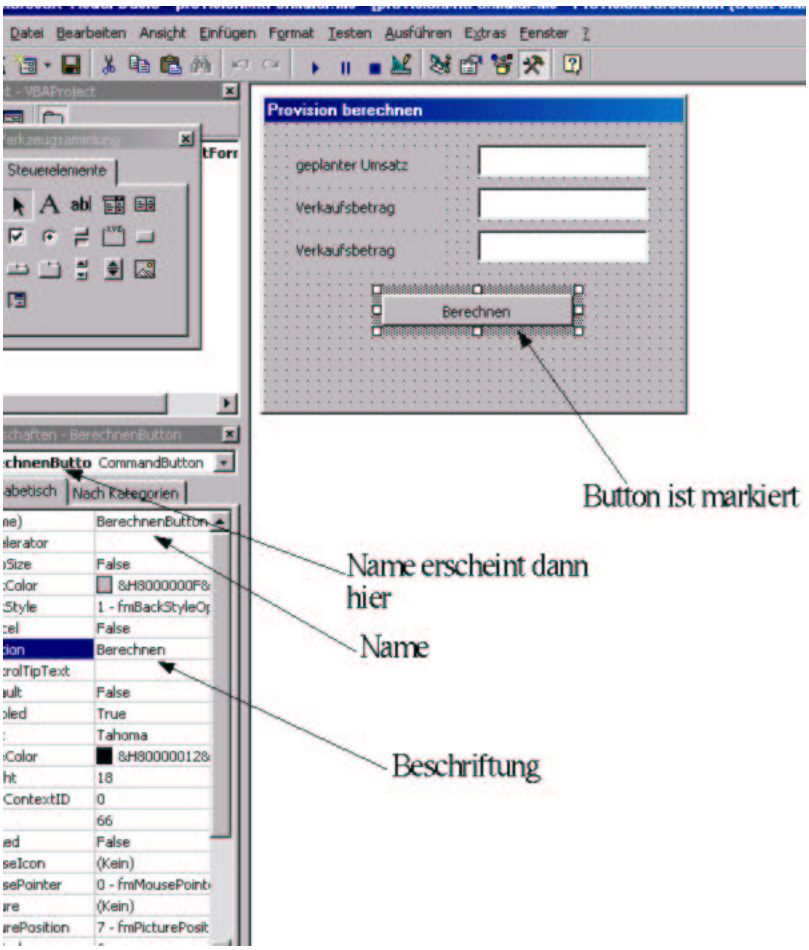

Abbildung 1.15: Hinzufügen eines Button

```
Function provisionMitBenutzerdefinierterFunktion _
                                        (umsatzEingabe As String, _
                                        verkaufsbetragEingabe As String)

' Funktion berechnet Provisionen abhängig
' vom Umsatz
' Dateiname: provisionMitBenutzerdefinierterFunktion3

    Dim umsatz As Double
    Dim verkaufsbetrag As Double

    If Not ueberpruefe_Benutzereingaben(umsatzEingabe, _
            verkaufsbetragEingabe, umsatz, verkaufsbetrag) Then Exit Function
    provisionMitBenutzerdefinierterFunktion = _
                                    berechne_Provision(umsatz, verkaufsbetrag)

End Function
```

Hier erfolgt das Einlesen der Variablen ja über die Parameterübergabe aus Zellen einer Excel-Tabelle. Danach werden die Benutzereingaben überprüft, die Provision wird berechnet und über den Funkti-

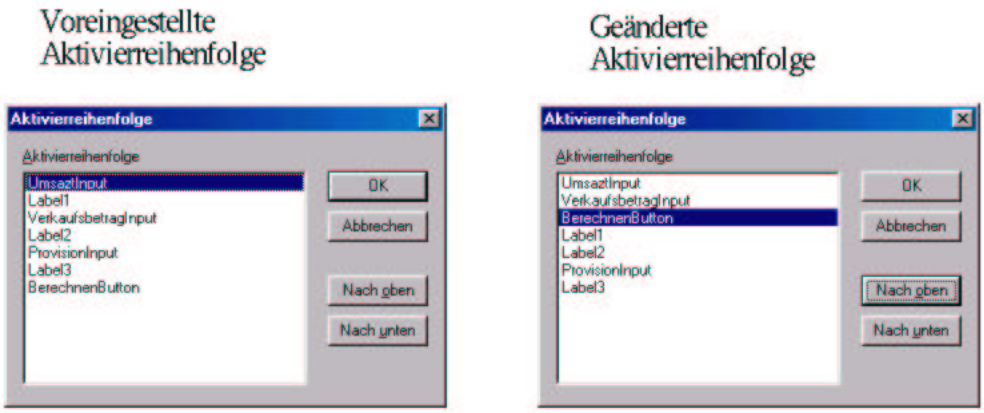

Abbildung 1.16: Änderung der Aktivierungreihenfolge

onsnamen in die Zelle, die die Funktion referenziert, eingetragen.

Bis auf das Einlesen der Benutzereingaben und das Ausgeben des Ergebnisses in eine Zelle, können wir den Code dieser Beispielanwendung nutzen. Die Funktionen ueberpruefe_Benutzereingaben und berechne_Provision können ungeändert übernommen werden.

Einlesen aus Ein- Ausgabefeldern und Schreiben in Ein- Ausgabefelder ist ziemlich einfach. Das Einlesen erfolgt einfach, indem ich eine Variable[16] auf die linke Seite einer Zuweisung schreibe. Auf der rechten Seite der Zuweisung steht der Name des Eingabefelds gefolgt von .Text. Um den Inhalt des Umsatz-Eingabefeldes auf die Variable umsatzString zu schreiben, ist also folgende Anweisung notwendig:

```
umsatzString=UmsatzInput.Text
```

UmsatzInput war ja der Name des Eingabefeldes für den Umsatz, wie wir ihn bei der Erstellung des Formulars vergeben hatten.

Um Text in ein Ein- Ausgabefeld zu schreiben, muss umgekehrt vorgegangen werden. Auf die linke Seite einer Zuweisung schreiben wir den Namen des Ein- Ausgabefeldes gefolgt von .Text. Auf der rechten Seite steht dann der darzustellende Text. Um also die Zahl 12 in das Provisions-Ausgabefeld zu schreiben, ist ist also folgende Anweisung notwendig:

```
ProvisionInput.Text = 12
```

ProvisionsInput war ja der Name des Ausgabefeldes für die Provision, wie wir ihn bei der Erstellung des Formulars vergeben hatten.

Mit diesem Wissen können wir nun unsere Ereignisprozedur realisieren:

Realisierung 1.4 *Die Ereignisprozedur BerechnenButton_Click()*

```
Private Sub BerechnenButton_Click()
    Dim umsatzEingabe As String
    Dim verkaufsbetragEingabe As String
    Dim umsatz As Double
    Dim verkaufsbetrag As Double
    umsatzEingabe = UmsatzInput.Text
```

[16] Das sollte natürlich die sein, auf die ich den Wert wirklich einlesen will.

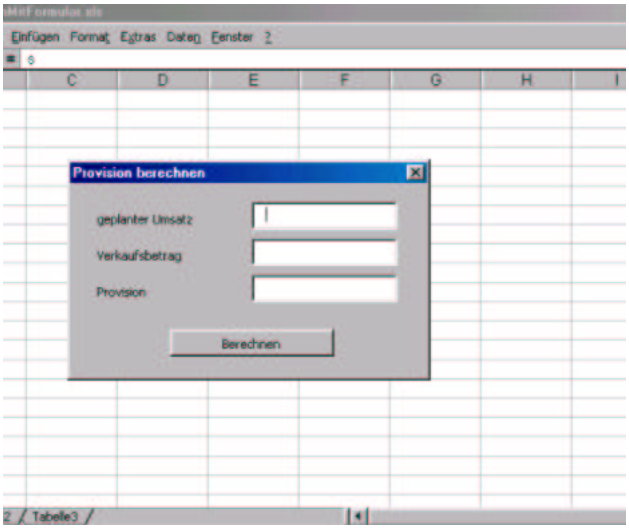

Abbildung 1.17: Unser schönes Formular

```
    verkaufsbetragEingabe = VerkaufsbetragInput.Text
    If Not ueberpruefe_Benutzereingaben(umsatzEingabe, _
            verkaufsbetragEingabe, umsatz, verkaufsbetrag) Then Exit Sub

    ProvisionInput.Text = berechne_Provision(umsatz, verkaufsbetrag)
End Sub
```

Die Änderungen zu unserem vorherigen Beispiel halten sich also in engen Grenzen. umsatzEingabe und verkaufsbetrag wandern aus der Parameterübergabe der benutzerdefinierten Funktion in die Variablendeklaration und werden dann mit den Werten der Eingabefelder besetzt. Die Aufrufe von ueberpruefe_Benutzereingaben und berechne_Provision bleiben gleich, die berechnete Provision wird allerdings nicht über den Funktionsnamen in eine Zelle einer Excel-Tabelle geschrieben, sondern durch die Zeile

```
    ProvisionInput.Text = berechne_Provision(umsatz, verkaufsbetrag)
```

in das Formular. Realisierung 1.5 zeigt zusammenfassend den VBA-Code dieser Anwendung.

Realisierung 1.5 *Provisionsberechnung mit einem Formular*

```
' Dateiname provisionMitFormular
Private Sub BerechnenButton_Click()
    Dim umsatzEingabe As String
    Dim verkaufsbetragEingabe As String
    Dim umsatz As Double
    Dim verkaufsbetrag As Double
    umsatzEingabe = UmsatzInput.Text
    verkaufsbetragEingabe = VerkaufsbetragInput.Text
    If Not ueberpruefe_Benutzereingaben(umsatzEingabe, _
            verkaufsbetragEingabe, umsatz, verkaufsbetrag) Then Exit Sub

    ProvisionInput.Text = berechne_Provision(umsatz, verkaufsbetrag)
End Sub
```

```
Private Function ueberpruefe_Benutzereingaben(ByVal umsatzEingabe As String, _
                    ByVal verkaufsbetragEingabe As String, _
                    umsatz As Double, _
                    verkaufsbetrag As Double) _
                    As Boolean

        ueberpruefe_Benutzereingaben = True
        If Not wandle_in_Double_um(umsatzEingabe, umsatz) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der erste Parameter (umsatz) der Funktion ist keine Zahl!")
            Exit Function
        End If

        If Not wandle_in_Double_um(verkaufsbetragEingabe, verkaufsbetrag) Then
            ueberpruefe_Benutzereingaben = False
            MsgBox ("Der zweite Parameter (verkaufsbetrag) der Funktion ist
keine Zahl!")
            Exit Function
        End If

        If verkaufsbetrag > umsatz Then
            MsgBox ("Umsatz muß größer gleich Verkaufsbetrag sein!")
            ueberpruefe_Benutzereingaben = False
            Exit Function
        End If
End Function

Private Function berechne_Provision(ByVal umsatz As Double, _
            ByVal verkaufsbetrag As Double) _
            As Double

    Dim provisionInProzent As Double

    ’ Umsatzgrenzen sind DM-Betraege

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

    ’ Provisionen in Prozent

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

    ’ Bestimme Provision
    If umsatz >= umsatzGrenze3 Then
        provisionInProzent = provisionUmsatzGrenze3
    ElseIf umsatz >= umsatzGrenze2 Then
        provisionInProzent = provisionUmsatzGrenze2
    ElseIf umsatz >= umsatzGrenze1 Then
        provisionInProzent = provisionUmsatzGrenze1
    Else
        provisionInProzent = 0
```

```
    End If

' Berechne die Provision

    berechne_Provision = (verkaufsbetrag * provisionInProzent) / 100
End Function
Private Function wandle_in_Double_um(ByVal eingabe As String, rueckgabe As
Double) _
                    As Boolean
    wandle_in_Double_um = True
    If Not IsNumeric(eingabe) Then
        MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
        wandle_in_Double_um = False
        Exit Function
    End If
    rueckgabe = CDbl(eingabe)
End Function
```

Dennoch sind wir noch nicht ganz fertig. Es stellt sich die Frage, was wir tun müssen, um unseren Benutzern das Formular anzuzeigen. Wir können nämlich kaum folgende Benutzeranleitung schreiben:

- Starten Sie den VBA-Editor!
- Doppelclicken Sie auf das Formular mit dem Namen ProvisionBerechnen!
- Drücken Sie nun F5!

Besser wäre, wenn das Formular automatisch erscheint, wenn der Benutzer die Excel-Tabelle öffnet. Aber auch dies ist leicht zu erreichen. Ähnlich wie Windows über einen Autostart-Ordner verfügt, in dem ich Anwendungen referenzieren kann, die jedem Systemstart automatisch gestartet werden, verfügt Excel über eine Funktion, die, wenn sie vorhanden ist, beim Öffnen der Excel-Arbeitsmappe automatisch ausgeführt wird.

Die Funktion heißt Workbook_Open. Sie muss in das Modul DieseArbeitmappe aufgenommen werden. Und hier müssen wir programmieren, dass Excel unser Formular startet. Dies ist eine Anweisung, wir schreiben den Namen des Formulars gefolgt von .Show. DieseArbeitmappe stellt sich also wie folgt dar (vgl. auch Abb. 1.18):

Realisierung 1.6 *Das Formular wird automatisch gestartet*

```
Option Explicit
Sub Workbook_Open()
    ProvisionBerechnen.Show
End Sub
```

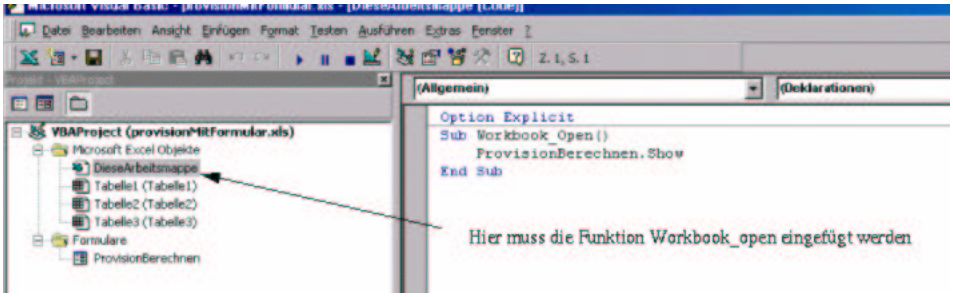

Abbildung 1.18: Das Formular im Autostart

1.2.2 Nutzung des Zinsbeispiels mit einem Formular

Wir wollen das Zinsbeispiel ebenfalls mit einer Formular-Benutzerschnittstelle ausstatten. Die Benutzerschnittstelle soll wie in Abb. 1.19 dargestellt aussehen.

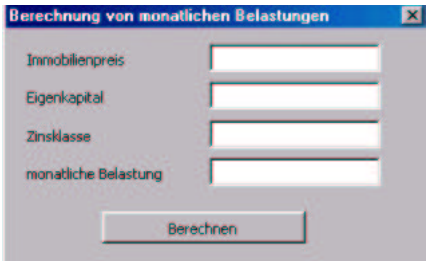

Abbildung 1.19: Das Formular der Zinsanwendung

Dazu erstellen wir, wie im vorherigen Abschnitt beschrieben, eine Userform. Das Formular heißt BelastungBerechnen, die Ein- Ausgabefelder nennen wir:

- ImmobilienpreisInput.
- EigenkapitalInput.
- ZinsklasseInput.
- MonatlicheBelastungInput.

Der Button wird BerechnenButton genannt. Durch einen Doppelclick auf den BerechnenButton erreichen wir das Code-Fenster mit dem von Excel erzeugten Prozedurrumpf.

Wie im vorherigen Beispiel muss auch hier nur das „Steuerprogramm“ geändert werden. Wir sehen uns noch einmal die Hauptfunktion aus Realisierung 1.3 an:

```
Function zinsberechnung(immobilienpreisString As String, _
                        eigenkapitalString As String, _
                        zinsKlasseString As String) _
                        As Double

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

```
        Dim eigenkapital As Double
        Dim immobilienpreis As Double
        Dim zinsKlasse As Integer
        Dim monatlicheBelastung As Double
        If Not ueberpruefe_alle_Benutzereingaben(immobilienpreisString, _
                                eigenkapitalString, zinsKlasseString, _
                                        immobilienpreis, eigenkapital, _
                                                zinsKlasse) Then Exit Function
        If Not fuehre_Berechnungen_durch(eigenkapital, immobilienpreis, _
                zinsKlasse, monatlicheBelastung) Then Exit Function
        zinsberechnung = monatlicheBelastung
    End Function
```

Die Änderungen sind marginal. Die Eingabewerte des Programms (immobilienpreisString, eigenkapitalString und zinsKlasseString) kommen in Realisierung 1.3 aus einer Excel-Tabelle. Nun holen wir sie uns über die Text-Eigenschaften der Ein- Ausgabefelder. Das Ergebnis wird in Realisierung 1.3 über den Funktionsnamen in eine Zelle der Excel-Tabelle geschrieben. Wir müssen sie nun in ein Ein- Ausgabefeld schreiben. Daraus ergibt sich folgende Ereignisprozedur:

```
Private Sub BerechnenButton_Click()
    Dim immobilienpreisString As String
    Dim eigenkapitalString As String
    Dim zinsKlasseString As String
    Dim eigenkapital As Double
    Dim immobilienpreis As Double
    Dim zinsKlasse As Integer
    Dim monatlicheBelastung As Double

    immobilienpreisString = ImmobilienpreisInput.Text
    eigenkapitalString = EigenkapitalInput.Text
    zinsKlasseString = ZinsklasseInput.Text

    If Not ueberpruefe_alle_Benutzereingaben(immobilienpreisString, _
                            eigenkapitalString, zinsKlasseString, _
                                    immobilienpreis, eigenkapital, _
                                            zinsKlasse) Then Exit Sub
    If Not fuehre_Berechnungen_durch(eigenkapital, immobilienpreis, _
            zinsKlasse, monatlicheBelastung) Then Exit Sub

    MonatlicheBelastungInput.Text = monatlicheBelastung
End Sub
```

Der gesamte Rest der Anwendung stimmt mit Realsisierung 1.3 überein. Die Anwendung ist zur Vollständigkeit in Realisierung 1.7 dargestellt.

Realisierung 1.7 *Monatliche Belastung mit einem Formular*

```
Option Explicit
Private Sub BerechnenButton_Click()
    Dim immobilienpreisString As String
    Dim eigenkapitalString As String
    Dim zinsKlasseString As String
    Dim eigenkapital As Double
    Dim immobilienpreis As Double
    Dim zinsKlasse As Integer
```

```
    Dim monatlicheBelastung As Double
    immobilienpreisString = ImmobilienpreisInput.Text
    eigenkapitalString = EigenkapitalInput.Text
    zinsKlasseString = ZinsklasseInput.Text
    If Not ueberpruefe_alle_Benutzereingaben(immobilienpreisString, _
                            eigenkapitalString, zinsKlasseString, _
                                    immobilienpreis, eigenkapital, _
                                            zinsKlasse) Then Exit Sub
    If Not fuehre_Berechnungen_durch(eigenkapital, immobilienpreis, _
            zinsKlasse, monatlicheBelastung) Then Exit Sub
    MonatlicheBelastungInput.Text = monatlicheBelastung
End Sub
Private Function berechne_Eigenkapitalquote(ByVal eigenkapital As Double, _
               ByVal immobilienpreis As Double) _
               As Boolean

    Dim eigenkapitalquote As Double
    berechne_Eigenkapitalquote = True

    eigenkapitalquote = (eigenkapital / immobilienpreis) * 100
    If eigenkapitalquote < 30 Then
        MsgBox ("Ihre Eigenkapitalquote " & eigenkapitalquote & _
                "% ist zu niedrig! ")
        berechne_Eigenkapitalquote = False
        Exit Function
    End If
End Function
Private Function Monatliche_Belastung_berechnen(ByVal immobilienpreis As Double,

_
                  ByVal eigenkapital As Double, _
                  ByVal zins As Double) _
                  As Double
    Const tilgung As Double = 1
    Dim aufzunehmenderBetrag As Double
    Dim jahresBelastung As Double
    Dim eigenkapitalquote As Double
    aufzunehmenderBetrag = immobilienpreis - eigenkapital
    jahresBelastung = (aufzunehmenderBetrag / 100) * (zins + tilgung)
    Monatliche_Belastung_berechnen = jahresBelastung / 12
End Function
Private Function ueberpruefe_alle_Benutzereingaben _
                       (ByVal immobilienpreisString As String, _
                        ByVal eigenkapitalString As String, _
                        ByVal zinsKlasseString As String, _
                       immobilienpreis As Double, _
                       eigenkapital As Double, _
                       zinsKlasse As Integer) _
                       As Boolean

        ueberpruefe_alle_Benutzereingaben = True
        If Not wandle_in_Double_um(immobilienpreisString, immobilienpreis) Then
            ueberpruefe_alle_Benutzereingaben = False
            MsgBox ("Immobilienpreis muss eine Zahl sein!")
```

```
            Exit Function
        End If

        If Not wandle_in_Double_um(eigenkapitalString, eigenkapital) Then
            ueberpruefe_alle_Benutzereingaben = False
            MsgBox ("Eigenkapital muss eine Zahl sein!")
            Exit Function
        End If

        If Not wandle_in_Integer_um(zinsKlasseString, zinsKlasse) Then
            ueberpruefe_alle_Benutzereingaben = False
            MsgBox ("Zinsklasse muss eine Zahl sein!")
            Exit Function
        End If
End Function
Private Function wandle_in_Double_um(ByVal eingabe As String, rueckgabe As
Double) _
                    As Boolean
    wandle_in_Double_um = True
    If Not IsNumeric(eingabe) Then
        MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
        wandle_in_Double_um = False
        Exit Function
    End If
    rueckgabe = CDbl(eingabe)
End Function
Private Function wandle_in_Integer_um(ByVal eingabe As String, rueckgabe As
Integer) _
                            As Boolean
    wandle_in_Integer_um = True
    If Not IsNumeric(eingabe) Then
        MsgBox ("Der von Ihnen eingegebene Wert muß eine Zahl sein! ")
        wandle_in_Integer_um = False
        Exit Function
    End If
    rueckgabe = CInt(eingabe)
End Function
Private Function fuehre_Berechnungen_durch(ByVal eigenkapital As Double, _
                    ByVal immobilienpreis As Double, _
                    ByVal zinsKlasse As Integer, _
                    monatlicheBelastung As Double) _
                    As Boolean

    Const zinsKlasse1 As Double = 5.5
    Const zinsKlasse2 As Double = 5.3
    Const zinsKlasse3 As Double = 5.2
    Const zinsKlasse4 As Double = 5#
    Const zinsKlasse5 As Double = 4.5

    fuehre_Berechnungen_durch = True

    If Not berechne_Eigenkapitalquote(eigenkapital, immobilienpreis) Then
        fuehre_Berechnungen_durch = False
        Exit Function
```

```
    End If
    Select Case zinsKlasse
        Case 1
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse1)
        Case 2
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse2)
        Case 3
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse3)
        Case 4
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse4)
        Case 5
            monatlicheBelastung = Monatliche_Belastung_berechnen _
                (immobilienpreis, eigenkapital, zinsKlasse5)
        Case Else
            MsgBox ("Sie haben eine falsche Zinsklasse eingegeben!  " & _
                        "Zinsklasse muß kleiner gleich 5 sein! ")
            fuehre_Berechnungen_durch = False
            Exit Function
    End Select
End Function
```

Zum Abschluss fügen wir die Autostart-Funktion in das Modul DieseArbeitsmappe ein (vgl. Realisierung 1.8).

Realisierung 1.8 *Das Formular zur Berechnung der monatlichen Belastung wird automatisch gestartet*

```
Option Explicit
Sub Workbook_Open()
    BelastungsBerechnung.Show
End Sub
```

1.3 Gemischte Realisierungen: Excel-Tabellen und Formulare

1.3.1 Eine gemischte Realisierung des Provisionsbeispiels

Wir ändern die Aufgabenstellung zum Provisionsbeispiel ein wenig ab:

1. Die Eingaben sollen weiterhin über ein Formular erfolgen (vgl. Abb. 1.20).
2. Die Ausgaben sollen direkt in eine Excel-Tabelle geschrieben werden.
3. Wenn die Ausgabe in die Excel-Tabelle geschrieben wurde, soll das Formular verschwinden, damit der Benutzer mit der Excel-Tabelle arbeiten kann.

4. Das Formular muss jederzeit wieder geladen werden können, damit der Benutzer weitere Provisionsberechnungen durchführen kann.

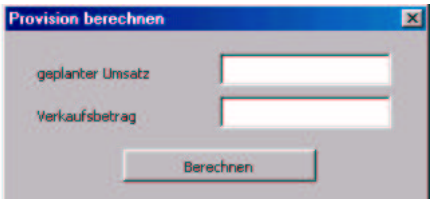

Abbildung 1.20: Das geänderte Formular

Punkt (1) ist relativ leicht zu lösen. Wir löschen einfach das überflüssige Label und das nicht mehr benötigte Ausgabefeld. Für Punkt (2) müssen wir nur wissen, wie man von VBA aus in Zellen schreibt. Dafür gibt es zwei Möglichkeiten:

```
Cells(1,1) = 37
Range("A1") = 37
```

Beide VBA-Code-Zeilen schreiben die Zahl 37 in die Zelle A1[17].

Für Punkt (3) muss man eigentlich nur wissen, dass die VBA-Anweisung

```
Unload me
```

ein Formular entfernt. Damit werden die Punkte (1) bis (3) durch folgenden VBA-Code implementiert:

Realisierung 1.9 *Anforderungen (1) bis (3) realisiert*

```
Option Explicit
Private Sub BerechnenButton_Click()
    Dim umsatzEingabe As String
    Dim verkaufsbetragEingabe As String
    Dim umsatz As Double
    Dim verkaufsbetrag As Double
    umsatzEingabe = UmsaztInput.Text
    verkaufsbetragEingabe = VerkaufsbetragInput.Text
    If Not ueberpruefe_Benutzereingaben(umsatzEingabe, _
            verkaufsbetragEingabe, umsatz, verkaufsbetrag) Then Exit Sub
    Cells(1, 1) = "Die Provision beträgt:"
    Cells(1, 2) = berechne_Provision(umsatz, verkaufsbetrag)
    Unload Me
End Sub
```

Die aufgerufenen Funktionen sind natürlich mit Realsierung 1.7 identisch. Sie werden daher nicht mehr dargestellt.

Bleibt Punkt (4). Punkt (4) realisieren wir mit einem Button. Steuerelemente können nämlich nicht nur in Formularen auftreten, man kann Steuerelemente auch direkt in ein Tabellenblatt einfügen. Um dies zu tun, müssen Sie zunächst die Steuerlemente-Toolbox öffnen. Dies ist nicht die Steuerlemente-Toolbox der VBA-Umgebung. Sie sieht auch anders aus, hat aber im Wesentlichen die gleichen Funktionen[18]. Sie öffnen die Steuerelemente-Toolbox, indem Sie auf das Steuerelemente-Symbol in der Symbolleiste clicken oder indem Sie Ansicht –> Symbolleisten –> Steuerelement-Toolbox

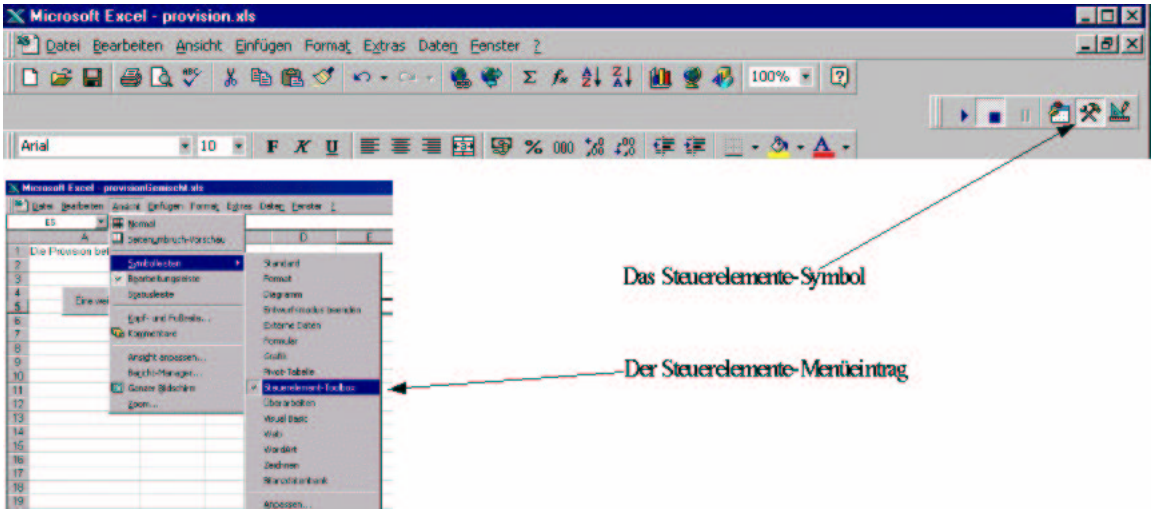

Abbildung 1.21: Öffnen der Steuerelemente Toolbox

auswählen (vgl. Abb. 1.21). Die Steuerelemente Toolbox erscheint (vgl. Abb. 1.22). Sie markieren das Schaltflächen-Symbol und zeichnen eine Schaltfläche auf das Tabellenblatt. Über das Kontextmenü (rechte Maustaste) können Sie die Eigenschaften der Schaltfläche verändern.

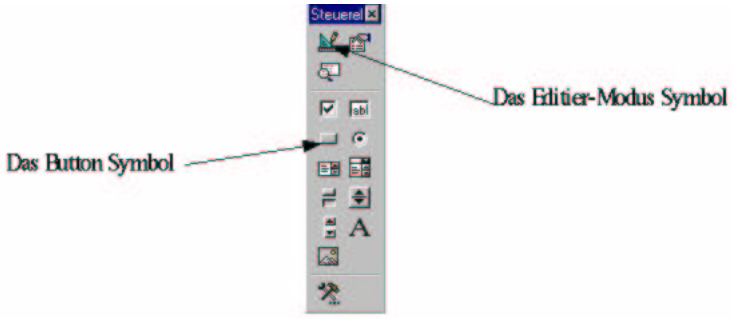

Abbildung 1.22: Die Steuerelemente-Toolbox

Doppel-Clicken Sie nun auf ihre neue Schaltfläche. Excel schaltet nun zum VBA-Editor um. Innerhalb des Moduls mit dem Namen Ihrer Tabelle erstellt Excel das Skelett einer Prozedur. Die Prozedur heißt CommandButton1_Click(). Sie ändern Name und Beschriftung des Button über den Ihnen schon bekannten Eigenschaften Dialog. Wir nennen den Button BerechnenAusTabelleStarten. Sie müssen nun den Namen der Ereignisprozedur in BerechnenAusTabelleStarten_Click() ändern. Sie müssen übrigens immer, wenn Sie den Namen eines Button ändern, den Namen der Ereignisprozedur anpassen. Excel macht das nicht automatisch.

Sollten Sie den Button später bearbeiten wollen, müssen Sie ihn zur Bearbeitung auswählen. Dies geschieht, indem Sie in der Steuerelemente-Toolbox den Editor-Modus anwählen (vgl. Abb.1.22) und dann auf den Button clicken. Selbst wenn Sie den Button nur verschieben wollen, müssen Sie ihn in den Editiermodus bringen.

[17] Wie man etwas in die Zelle A2 oder irgendeine andere Zelle schreibt, müssen Sie sich jetzt selbst überlegen :-).

[18] Warum dies so ist, weiß nur Microsoft (wenn die es wissen).

Zwischen dem Prozedurname und End Sub fügen Sie nun (wie immer) Ihren VBA-Code ein. Dies ist hier nur eine Anweisung, denn es soll ja nur das Formular wieder eingeblendet werden. Das Formular hat den Namen ProvisionBerechnen, die Anweisung, um das Formular wieder einzublenden heißt also ProvisionBerechnen.Show. Die Implementierung der Ereignisprozedur ist also wie folgt:

Realisierung 1.10 *Ereignisprozedur zum Aufblenden des Formulars*

```
Option Explicit
Private Sub BerechnenAusTabelleStarten_Click()
    ProvisionBerechnen.Show
End Sub
```

Abb. 1.23 fasst die Vorgehensweise zusammen, Abb. 1.24 zeigt den erstellten Button in der Excel-Tabelle.

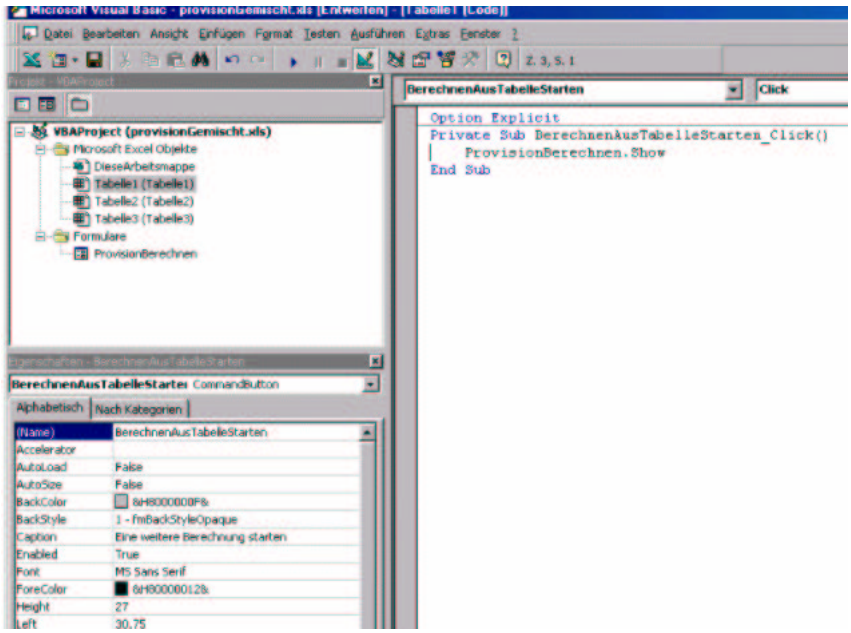

Abbildung 1.23: Umbenennen des Button und Programmierung der Ereignisprozedur

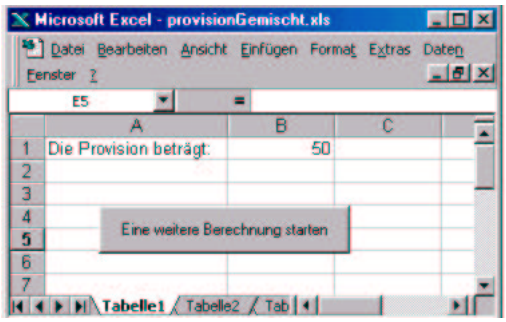

Abbildung 1.24: Der Button in der Excel-Tabelle

1.3.2 Eine gemischte Realisierung des Zinsbeispiels

Wir ändern die Aufgabenstellung zum Zinsbeispiel ein wenig ab:

1. Die Eingaben sollen weiterhin über ein Formular erfolgen (vgl. Abb. 1.25).
2. Die Ausgaben sollen direkt in eine Excel-Tabelle geschrieben werden. Allerdings sollen bereits vorhandene Berechnungen nicht überschrieben werden, VBA soll neue Ergebnisse in eine neue Zeile der Tabelle schreiben. Dies soll selbst dann der Fall sein, wenn der Benutzer die Tabelle speichert und beim nächsten Mal neu öffnet.
3. Das Formular soll einen zweiten Knopf erhalten, mit dem man es entfernen kann. Ansonsten soll es im Vordergrund bleiben und neue Eingaben erwarten.
4. Das Formular muss jederzeit wieder geladen werden können, damit der Benutzer weitere Belastungsberechnungen durchführen kann.

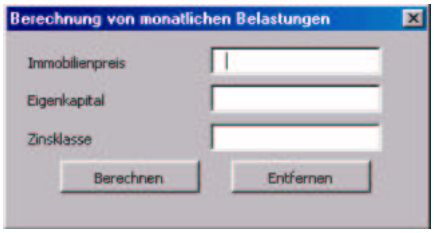

Abbildung 1.25: Das geänderte Formular des Belastungsrechnungsbeispiels

Die Punkte (1) und (4) fallen uns leicht. Wie im vorherigen Kapitel löschen wir im Formular einfach das überflüssige Ausgabefeld. In die Excel-Tabelle fügen wir mit Hilfe der Steuerelemente-Toolbox einen Button ein, doppelclicken den neuen Button, benennen ihn um und schreiben die Ereignisprozedur, die wieder nur aus einem Befehl besteht. Realisierung 1.11 zeigt die Implementierung.

Realisierung 1.11 *Ereignisprozedur zum Aufblenden des Zins-Formulars*

```
Option Explicit
Private Sub BerechnenAusTabelleStarten_Click()
    BelastungsBerechnung.Show
End Sub
```

Auch die neue Anforderung (3) ist nicht wirklich schwierig. Wir erzeugen diesmal mit der Steuerelemente-Toolbox der VBA-Entwicklungsumgebung einen neuen Button in unserem Formular, geben ihm den Namen EntfernenButton und beschriften ihn mit „Entfernen“. Dann doppelclicken wir den Button, um in seine Ereignisprozedur zu kommen und fügen in diese die Zeile „Unload Me“ ein. Realisierung 1.12 zeigt dies.

Realisierung 1.12 *Ereignisprozedur zum Entfernen des Zins-Formulars*

```
    Private Sub EntfernenButton_Click()
        Unload Me
    End Sub
```

Anforderung (2) hat es allerdings in sich. Wir wissen nämlich so ohne weiteres nicht, was die nächste freie Zeile der Excel-Tabelle ist. Eine einfache Lösung wäre, einfach mitzuzählen, wieviele Berechnungen durchgeführt wurden. Dies geht aber nicht, weil der Benutzer die Tabelle ja abspeichern kann. Wir dürfen beim nächsten Öffnen der Seite dann nicht wieder in die erste Zeile schreiben, sondern müssen die Berechnungen, die bereits in der Tabelle vorhanden sind, berücksichtigen. Es hilft also nichts, wir müssen bevor wir in die Tabelle schreiben, feststellen, was die nächste freie Zeile ist. Ich zeige jetzt zunächst den zugehörigen VBA-Code und erläutere ihn dann.

Realisierung 1.13 *Feststellen der nächsten freien Zeile*

```
    Private Function naechsteFreieZeile() As Integer
        Dim zelleBelegt As Boolean
        Dim i As Integer
        zelleBelegt = True
        i = 1
        Do While zelleBelegt
            If IsEmpty(Cells(i, 1)) Then
                naechsteFreieZeile = i
                zelleBelegt = False
            End If
            i = i + 1
        Loop
    End Function
```

In den ersten beiden Zeilen der Funktion nach den Variablendeklarationen werden die Variablen zelleBelegt und i initialisiert und zwar mit true bzw. 1. Dann startet eine while-Schleife. Die Schleifenbedingung ist der Wert der logischen Variablen zelleBelegt. zelleBelegt wurde mit true initialisiert, so dass VBA, wenn es zum ersten Mal auf die Schleife trifft, diese auch durchführt. Die erste Anweisung der Schleife ist ein if-statement. Die Bedingung des if-statements ist ein Aufruf der Funktion isEmpty, die Sie noch nicht kennen. isEmpty erwartet als einzigen Übergabeparameter eine Zelle einer Excel-Tabelle. Ist die Zelle leer, gibt isEmpty true zurück, ansonsten false. Und dies ist auch schon der ganze Trick. Unsere Schleife startet mit dem Wert 1 für i. Dann stellt isEmpty fest, ob die Zelle (1,1), also A1 belegt ist oder nicht. Ist sie belegt, dann wird das if-statement ignoriert, i wird um 1 erhöht (durch die Zeile i=i+1) und die Schleife läuft ein nächstes Mal, da der Wert von zelleBelegt ja nicht geändert wurde, daher also immer noch true ist. Dies passiert solange, bis isEmpty bei einer Zelle true zurück gibt. Dann haben wir die erste freie Zeile gefunden, die Funktion gibt über ihren Namen diese Zeile zurück und zelleBelegt wird auf false gesetzt. Dies beendet die Schleife und im Anschluss daran die Funktion.

Damit sollte dann die Codierung der Ereignisprozedur des BerechnenButton für Sie verständlich sein.

Realisierung 1.14 *Die neue Ereignisprozedur*

```
Option Explicit
Private Sub BerechnenButton_Click()
    Dim immobilienpreisString As String
    Dim eigenkapitalString As String
    Dim zinsKlasseString As String
    Dim eigenkapital As Double
    Dim immobilienpreis As Double
    Dim zinsKlasse As Integer
    Dim monatlicheBelastung As Double
```

```
    Dim i As Integer

    immobilienpreisString = ImmobilienpreisInput.Text
    eigenkapitalString = EigenkapitalInput.Text
    zinsKlasseString = ZinsklasseInput.Text
    If Not ueberpruefe_alle_Benutzereingaben(immobilienpreisString, _
                            eigenkapitalString, zinsKlasseString, _
                                    immobilienpreis, eigenkapital, _
                                            zinsKlasse) Then Exit Sub
    If Not fuehre_Berechnungen_durch(eigenkapital, immobilienpreis, _
            zinsKlasse, monatlicheBelastung) Then Exit Sub

    i = naechsteFreieZeile()
    Cells(i, 1) = "Ihre monatliche Belastung beträgt"
    Cells(i, 2) = monatlicheBelastung
End Sub
```

1.4 Nutzung von Excel-Funktionen

In diesem Kapitel wollen wir uns ansehen, wie man in Excel existierende Funktionalitäten aus VBA heraus benutzen kann. Wir machen dies an unserem Provisionsbeispiel. Wir wollen verschiedene Verkaufsbeträge eines Kunden eingeben, das System soll die Provision berechnen. Wenn wir auf einen mit „Grafische Darstellung“ beschrifteten Button drücken, soll die zu programmierende Ereignisprozedur eine grafische Darstellung der realisierten Umsätze und Provisionen erzeugen. Insgesamt soll die Excel-Tabelle vor dem Drücken des Button Abb. 1.26 entsprechen.

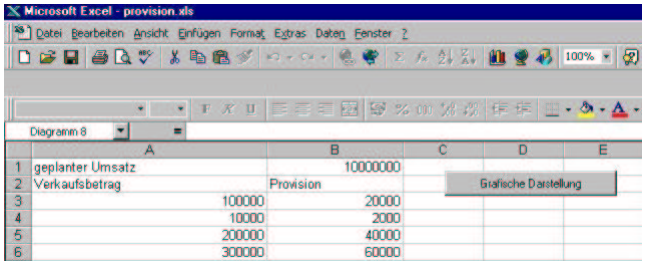

Abbildung 1.26: Provisionsbeispiel vor Darstellung der Grafik

Die Realisierung der Berechnung ist einfach. Wir werden einfach in der Spalte B unsere benutzerdefinierte Funktion zur Berechnung der Provision einfügen und jeweils in den Übergabeparametern die entsprechende Zelle der Spalte A und die Zelle B1 referenzieren.

Dann haben wir aber noch das Problem der Erzeugung des Diagramms. Dazu müssen wir auf das in Excel für Diagramme zuständige Objekt zugreifen. Ich habe ja bereits bereits kurz angesprochen, was Objekte und was Methoden und Eigenschaften von Objekten sind. Allerdings wissen wir weder den Namen des Diagramm-Objekts, noch kennen wir seine Methoden[19]. Theoretisch müssten wir jetzt in den diversen Excel-Hilfen suchen oder bei Microsoft anrufen. Beides ist nicht so unbedingt erstrebenswert. Glücklicherweise gibt es eine elegantere Lösung. Wir können uns von Excel zeigen lassen, wie man Diagramme programmiert. Dazu gibt es die Excel Funktion „Makro Aufzeichen“. Wir schalten „Makro Aufzeichen“ an. Dann erstellen wir „von Hand“ unser Diagramm. Wir beenden

[19] Methoden sind ein Synonym für Funktionen.

„Makro Aufzeichen“. Excel erstellt für alle unsere Aktionen VBA-Kommandos und speichert diese als Prozedur in einem neuen Modul der Excel-Arbeitsmappe ab.

Doch gehen wir den Vorgang im Einzelnen durch. Zunächst schalten wir „Makro Aufzeichen“ an (vgl. Abb. 1.27), Excel Kommandofolge Extras –> Makro –> Aufzeichnen).

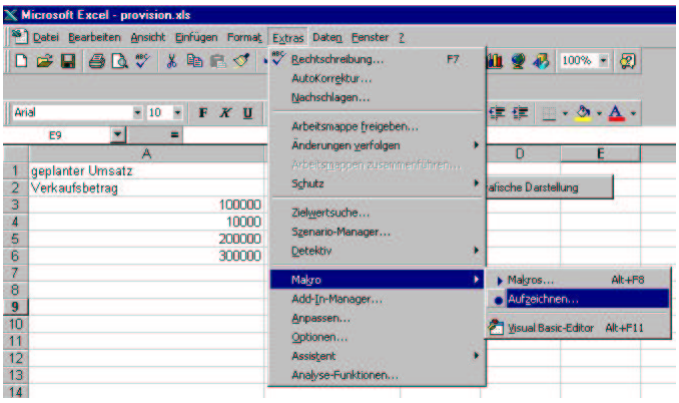

Abbildung 1.27: „Makro Aufzeichnen“ anschalten

Wir vergeben einen Namen für das Makro, markieren die Zellen, die in das Diagramm aufgenommen werden sollen und fügen das neue Diagrammm ein (vgl. Abb. 1.28), Excel Kommandofolge Einfügen –> Diagramm). In den nächsten vier Excel-Eingabefenstern geben wir die für die Erstellung

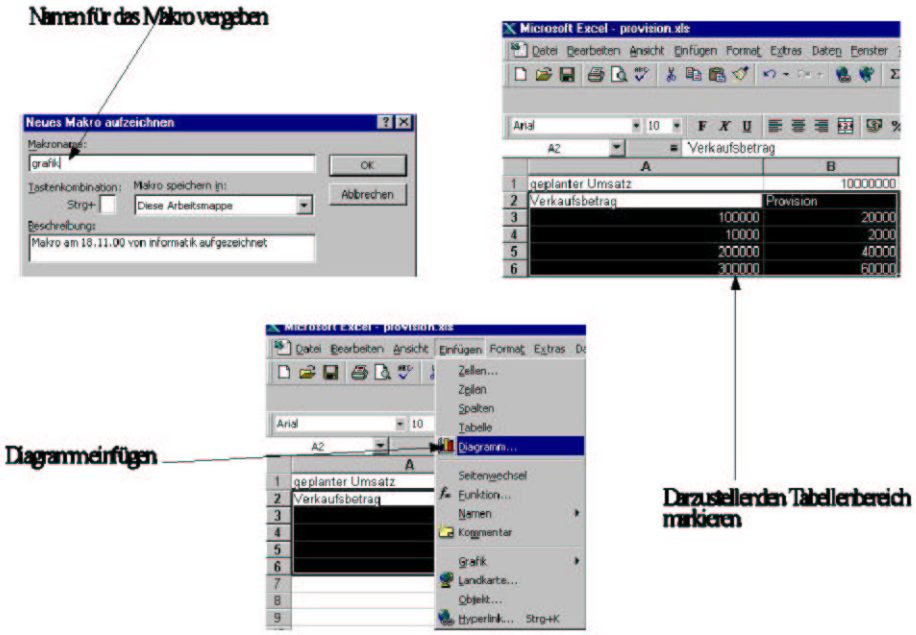

Abbildung 1.28: Erste Schritte Diagramm einfügen

des Diagramms notwendigen Informationen ein. Wir wählen ein Liniendiagramm und belassen sonst eigentlich die Voreinstellungen (vgl. Abb. 1.29). Daraufhin erscheint das Diagramm im Tabellenblatt

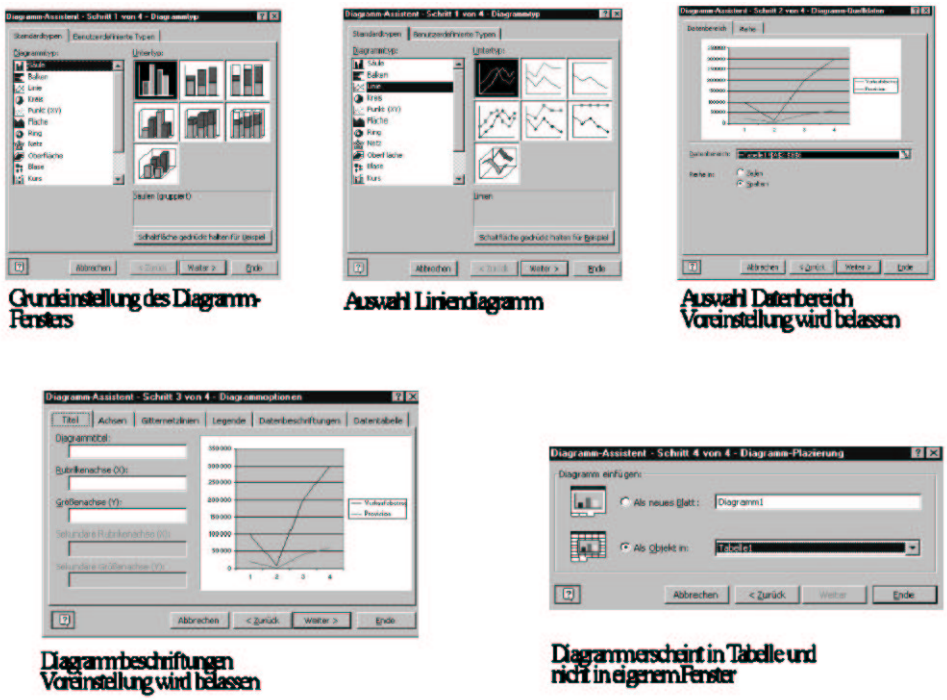

Abbildung 1.29: Dialoge zu Diagramm einfügen

(vgl. Abb. 1.30). Wir beenden die Aufzeichnung (vgl. Abb. 1.31). Durch das Aufzeichnen das Makros entstand eine Prozedur mit dem Namen des Makros in einem von Excel neu angelegten Modul (vgl. Abb. 1.32). Nun erzeugen wir, wie bereits besprochen, eine neue Schaltfläche mit dem Namen GrafischeDarstellung und der Beschriftung „Grafische Darstellung“. Damit ist auch das in Abb. 1.33 dargestellte Code-Skelett vorhanden. Im letzten Schritt kopieren wir den aufgezeichneten Code in das von Excel erzeugte Code-Skelett für unsere Schaltfläche. Wir löschen das Modul mit dem aufgezeichneten Code. Jedesmal, wenn wir jetzt unsere Schaltfläche betätigen, wird die Grafik neu erzeugt. Dies ist von jetzt an der übliche Weg, wenn wir erstmals mit Excel-Objekten, die wir auch aus Menüs oder ähnlichem erreichen können, arbeiten wollen. Wir lassen uns von Excel Beispiel-Code erzeugen und passen diesen an. Jetzt wollen wir aber das erzeugte Programm im Einzelnen durchgehen:

Realisierung 1.15 *Mit VBA ein Diagramm mit festen Grenzen erzeugen*

```
Private Sub GrafischeDarstellung_Click()
    Range("A2:B6").Select
    Charts.Add
    ActiveChart.ChartType = xlLine
    ActiveChart.SetSourceData Source:=Sheets("Tabelle1").Range("A2:B6"), PlotBy _
        :=xlColumns
    ActiveChart.Location Where:=xlLocationAsObject, Name:="Tabelle1"
    With ActiveChart
        .HasTitle = False
        .Axes(xlCategory, xlPrimary).HasTitle = False
        .Axes(xlValue, xlPrimary).HasTitle = False
    End With
    ActiveWindow.Visible = False
```

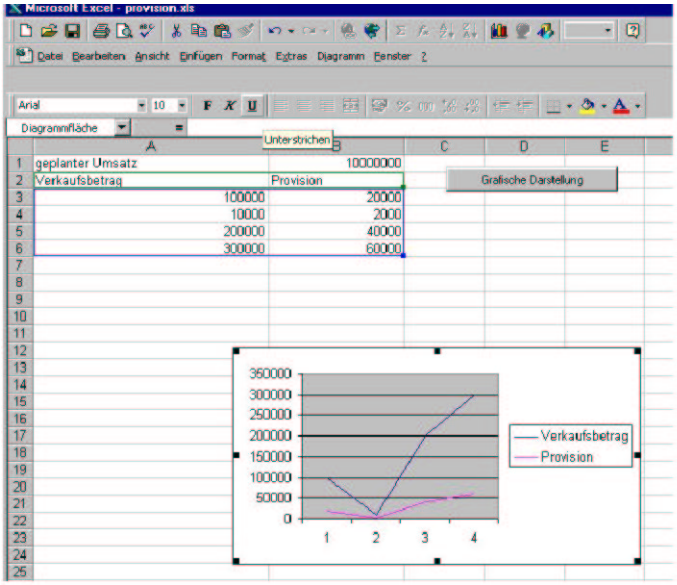

Abbildung 1.30: Die Grafik erscheint

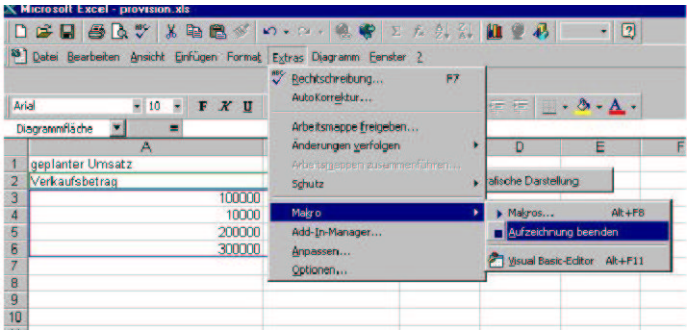

Abbildung 1.31: Der Aufzeichnungsmodus wird beendet

```
    Windows("provision.xls").Activate
End Sub
```

Die erste Zeile des Programms wurde ja von Excel bei der Erzeugung der Schaltfläche angelegt. Sie ist die normale Ereignisprozedurdeklaration.

Range definiert offenbar ein Objekt. Man sieht dies daran, daß mittels eines Punktes eine Methode (Select) angeschlossen wird. Range(„A2:B6“) ist der Zellbereich zwischen A2 und B6. Die Methode Select markiert ihn.

```
    Range("A2:B6").Select
```

Charts ist eine Objektliste, nämlich die Liste aller Diagramme (engl. Charts) der Arbeitsmappe. Charts.Add erzeugt ein neues Diagramm. Das zuletzt erzeugte (oder ausgewählte) Diagramm ist das aktuelle oder aktive Diagramm. Es kann über ActiveChart angesprochen werden. ActiveChart ist also der Name des gerade aktiven Chart-Objekts[20]. Durch ActiveChart.ChartType wird der Typ des Diagramms festgelegt. Wir hatten ein Liniendiagramm erzeugt. xlLine ist der Excel interne Name für Liniendiagramme.

[20]Ich werde Chart und Chart-Objekt von nun an als Synonyme benutzen

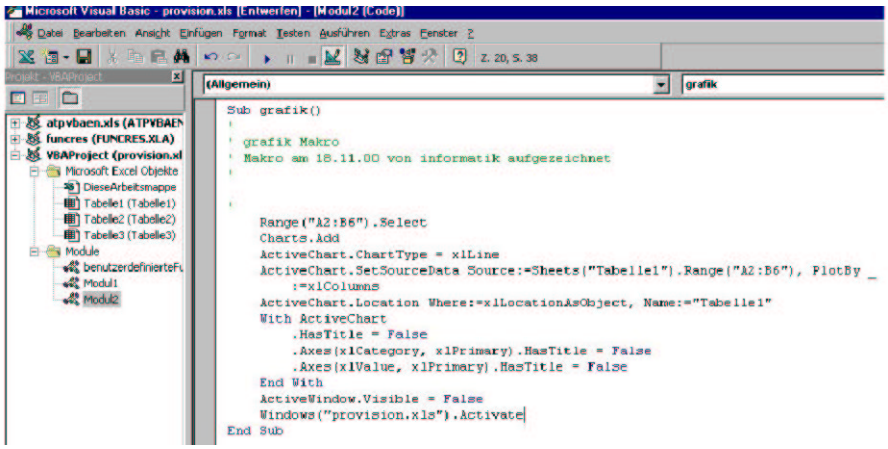

Abbildung 1.32: Das aufgezeichnete Modul

```
ActiveChart.ChartType = xlLine
```

Als nächstes wird die Methode[21] SetSourceData des ActiveChart-Objekts gerufen. SetSourceData verfügt über Unmengen von Übergabeparameteren. Wir haben nur sehr wenige davon benutzt, wir haben ja überall die Defaulteinstellungen belassen. Die Makro-Aufzeichnung benutzt also Parameter mit Namen (vgl. Kapitel 11.13), um der Methode die notwendigen Parameter zu übergeben. Der erste Parameter legt die Quelle der Daten fest (die Daten, die im Diagramm dargestellt werden sollen). Wir sehen, dass es eine Objektliste namens Sheets gibt. Dies sind (überraschenderweise :-)) die Tabellenblätter der Arbeitsmappe. Excel kann ja hier beliebig viele von verwalten. Sheets(„Tabelle1“) ist also das erste Tabellenblatt. Danach wird der Zellenbereich angegeben. Das sind die Zellen von A2 bis B6 und das hatten wir ja schon weiter oben. PlotBy legt fest, ob Spalten oder Zeilen geplottet werden sollen. Wir hatten Spalten ausgewählt. xlColumns ist der Excel interne Name dafür.

```
ActiveChart.SetSourceData Source:=Sheets("Tabelle1").Range("A2:B6"), PlotBy _
:=xlColumns
```

In der nächsten Zeile wird durch

```
ActiveChart.Location Where:=xlLocationAsObject, Name:="Tabelle1"
```

festgelegt, dass das Diagramm in der Excel-Tabelle mit dem Namen Tabelle1 eingefügt werden soll.

Nun sehen wir ein uns bis jetzt neues VBA-Schlüsselwort. With ist einfach eine abkürzende Schreibweise. In allen Zeilen zwischen With und End With wird einfach das auf With folgende Wort dem Punkt vorangestellt. Natürlich muß jede Zeile zwischen With und End With mit einem Punkt beginnen. Der Code

```
With ActiveChart
    .HasTitle = False
    .Axes(xlCategory, xlPrimary).HasTitle = False
    .Axes(xlValue,xlPrimary).HasTitle = False
    End With
```

entspricht also:

```
ActiveChart.HasTitle  = False
ActiveChart.Axes(xlCategory,xlPrimary).HasTitle = False
ActiveChart.Axes(xlValue, xlPrimary).HasTitle = False
```

[21] Auch Funktion und Methode werde ich von nun an synonym benutzen.

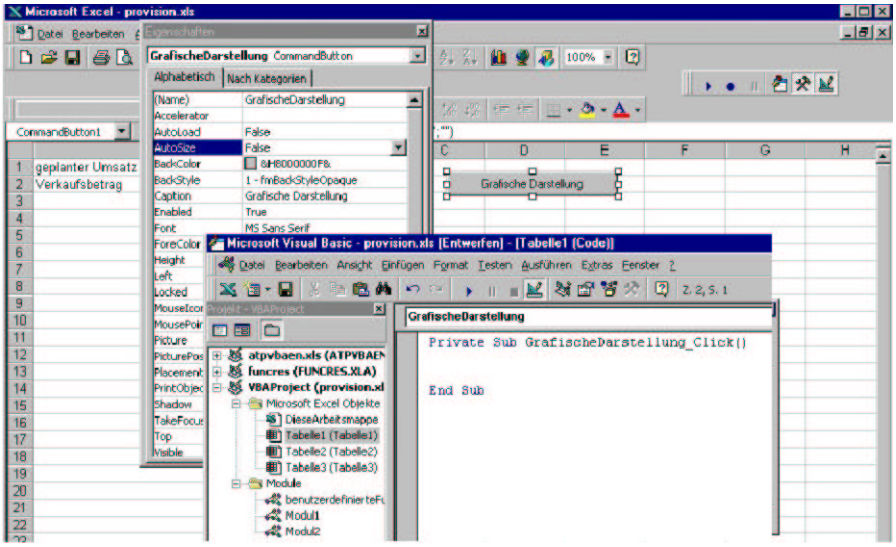

Abbildung 1.33: Schaltfläche mit von Excel erzeugtem Prozedur-Skelett

Insgesamt setzt der Code drei Eigenschaften des ActiveChart-Objekts. Er legt fest, ob es Überschriften für das Diagramm, die x-Achse und die Y-Achse gibt. Wir hatten dies bei der Erzeugung des Diagramms alles verneint. Die Eigenschaften werden also alle auf false gesetzt.

```
ActiveWindow.Visible = False
```

Diese Zeile setzt das zur Zeit aktive Fenster auf unsichtbar. Das zur Zeit aktive Fenster ist die VBA-Entwicklungsumgebung innerhalb derer der Code läuft. Die soll ja keiner sehen. Die letzte Zeile macht das Excel-Fenster wieder zum aktiven Fenster.

```
Windows("provision.xls").Activate
```

So ist auch das kurze Bildschirmflackern während des Laufens des Makros zu erklären. Die VBA-Entwicklungsumgebung wird das aktive Fenster und eigentlich sichtbar. Der Code läuft jedoch zu schnell durch, um das Fenster wirklich sichtbar werden zu lassen und durch die letzten beiden Zeilen bleibt (oder richtiger wird wieder) die Excel-Tabelle sichtbar.

Unser Programm hat jetzt noch ein Problem: Wir haben die grafische Darstellung für den festen Bereich A2B6 gelöst. Das ist aber nicht ganz das, was wir wollen. Wir wollen unseren Button Provisionen und Umsätze darstellen lassen, wann immer der Benutzer clickt und wieviele Zeilen er auch immer erfasst hat. Das ist aber auch nicht das Problem. Im letzten Kapitel hatten wir die Funktion naechsteFreieZeile() geschrieben. Sie gibt[22] die nächste freie Zeile zurück. Wir benötigen in diesem Beispiel die letzte besetzte Zeile. Das ist aber nächste freie Zeile minus 1.

Folgender VBA-Code setzt also den richtigen Bereich:

```
letzteBesetzteZeile = naechsteFreieZeile() - 1
bereich = "A2B" & letzteBesetzteZeile
```

Und damit erzeugt Realsierung 1.16 das gewünschte Ergebnis:

Realisierung 1.16 *Mit VBA ein Diagramm mit variablen Grenzen erzeugen*

[22] Wie der Name schon sagt.

```
Private Sub GrafischeDarstellung_Click()
    dim letzteBesetzteZeile As Integer
    dim bereich As String

    letzteBesetzteZeile = naechsteFreiZeile() - 1
    bereich = "A2B" & letzteBesetzteZeile

    Charts.Add
    ActiveChart.ChartType = xlLine
    ActiveChart.SetSourceData Source:=Sheets("Tabelle1").Range(bereich), PlotBy _
        :=xlColumns
    ActiveChart.Location Where:=xlLocationAsObject, Name:="Tabelle1"
    With ActiveChart
        .HasTitle = False
        .Axes(xlCategory, xlPrimary).HasTitle = False
        .Axes(xlValue, xlPrimary).HasTitle = False
    End With
    ActiveWindow.Visible = False
    Windows("provision.xls").Activate
End Sub
```