Integration des VBA-Codes in Excel-Tabellen, Zugriff auf Datenbanken

Bernd Blümel

Version: 25. Mai 2004

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-

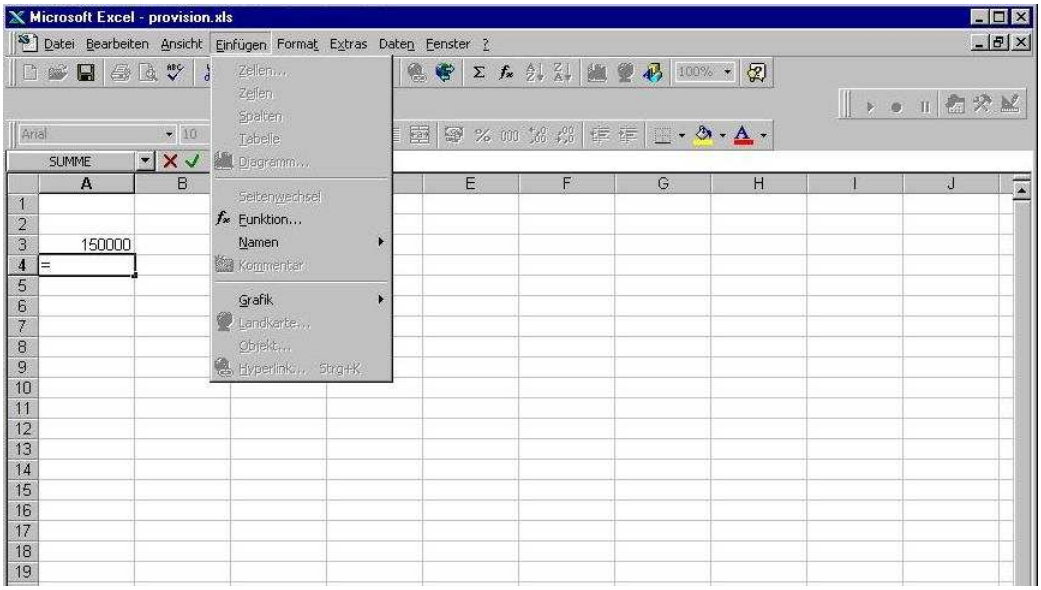

Abbildung 1.1: Das Excel Einfügen-Menü

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

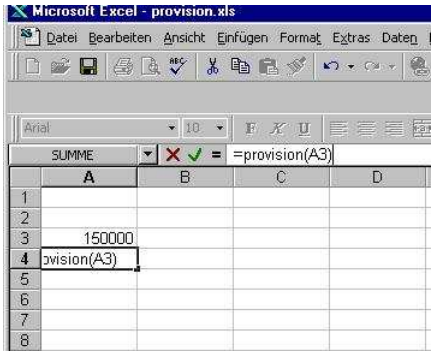

Abbildung 1.3: Provision in die Tabelle übernommen

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).

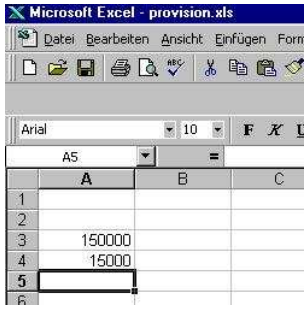

Abbildung 1.4: Provision in die Tabelle übernommen

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,

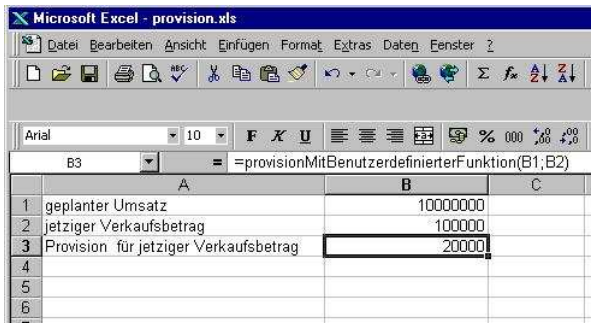

Abbildung 1.5: Das Benutzerinterface der Provisionsanwendung

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“

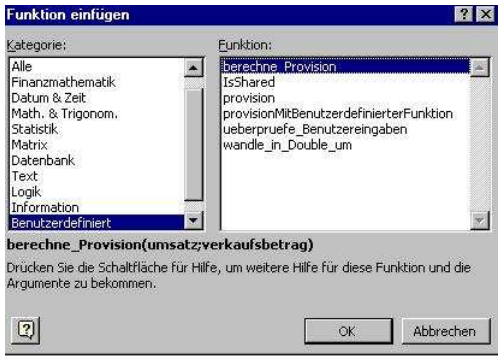

Abbildung 1.6: Die Rubrik benutzerdefinierte Funktionen nach der Implementierung von Realisierung 1.1

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

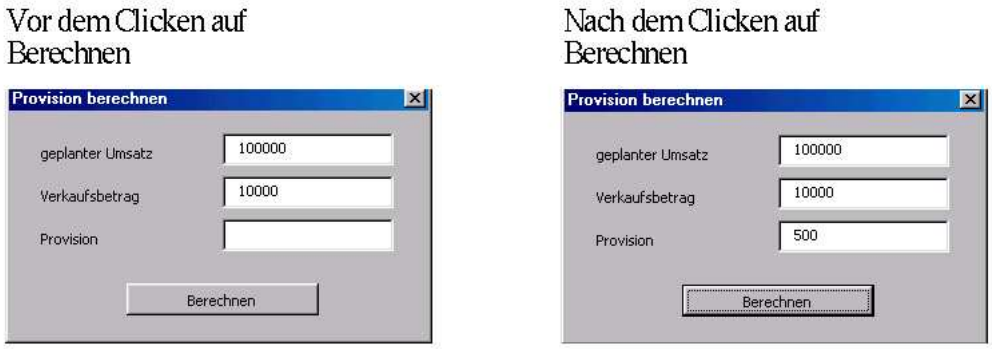

Abbildung 1.9: Ein- und Ausgabe des Provisionsbeispiels über ein Formular

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.

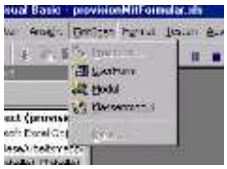

Abbildung 1.10: Erzeugen einer UserForm

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

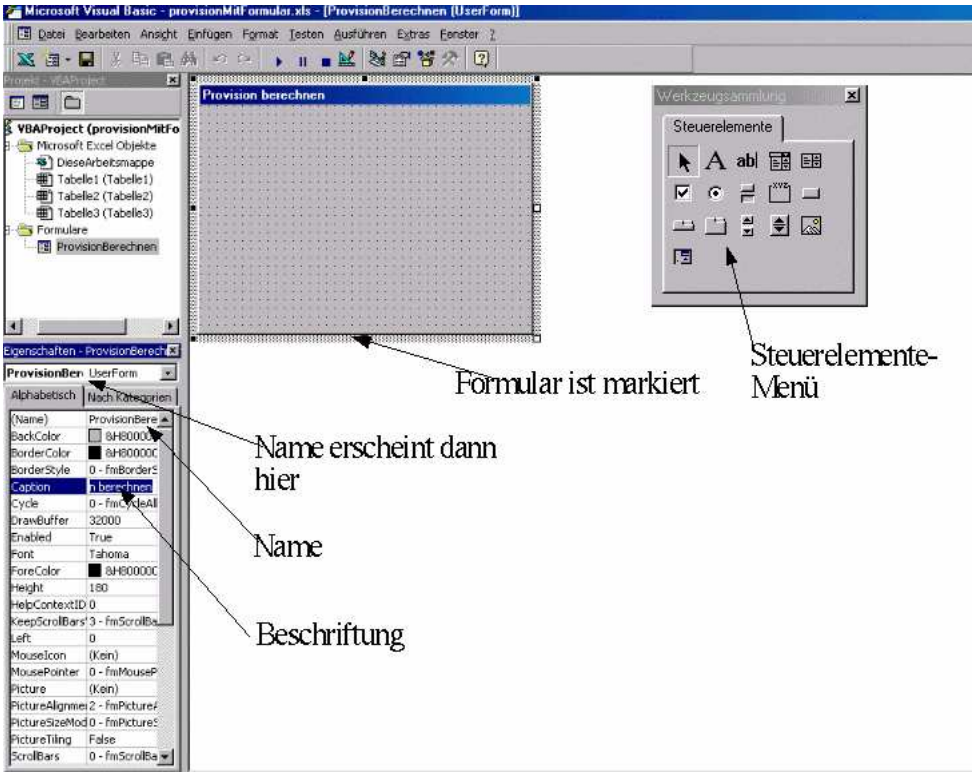

Abbildung 1.11: Erzeugen einer UserForm: Die Darstellung

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.

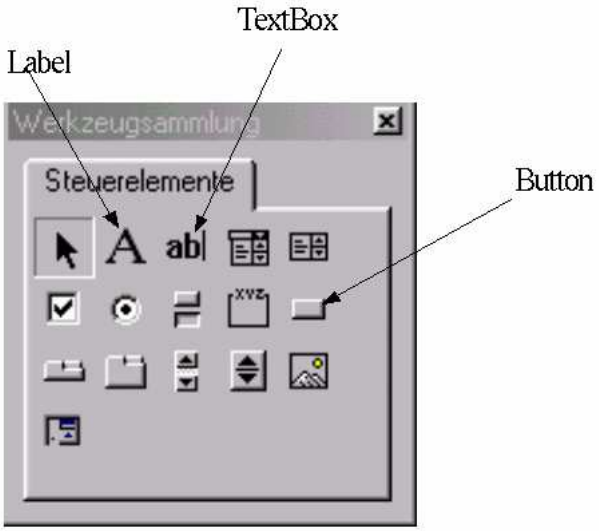

Abbildung 1.12: Die Steuerelemente-Toolbox

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!

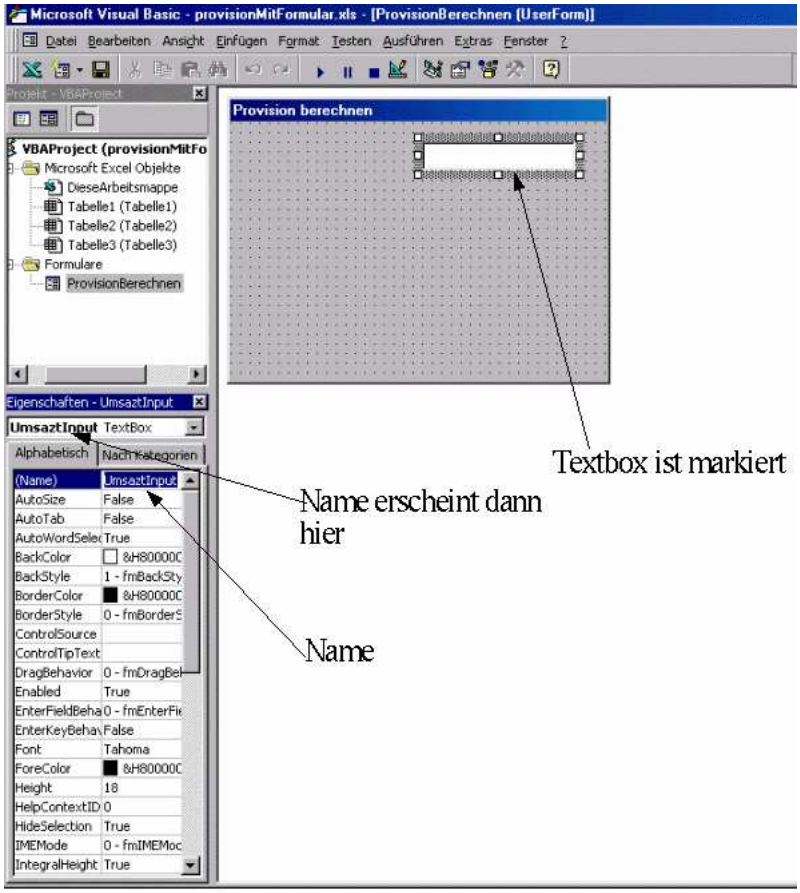

Abbildung 1.13: Hinzufügen eines Eingabefeldes (einer TextBox)

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.

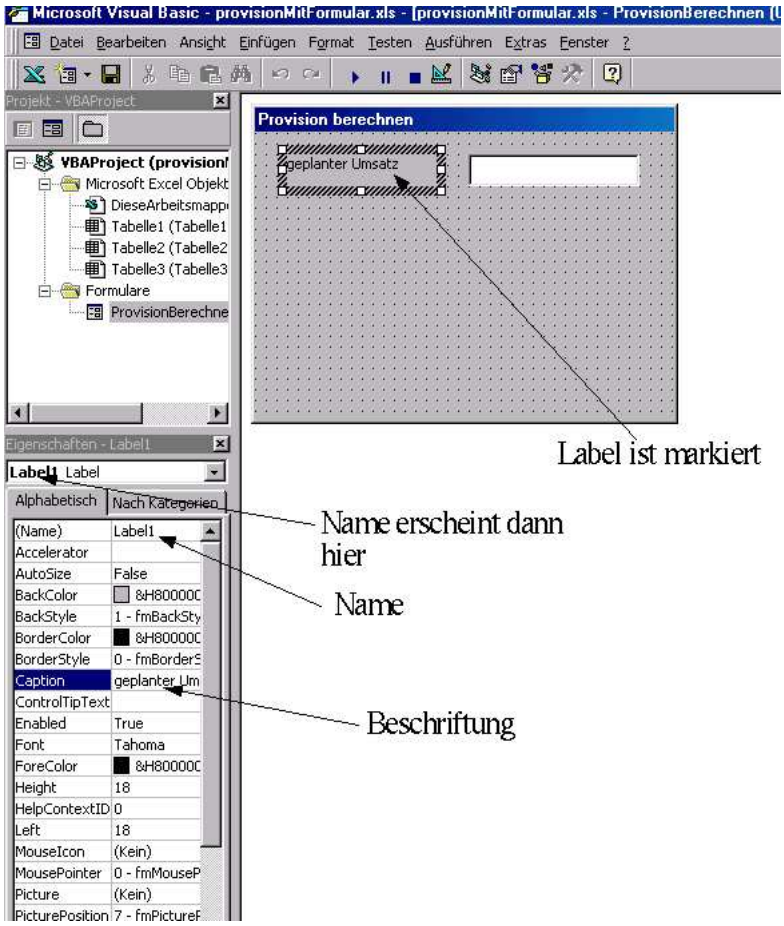

Abbildung 1.14: Hinzufügen einer Beschriftung (eines Labels)

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.

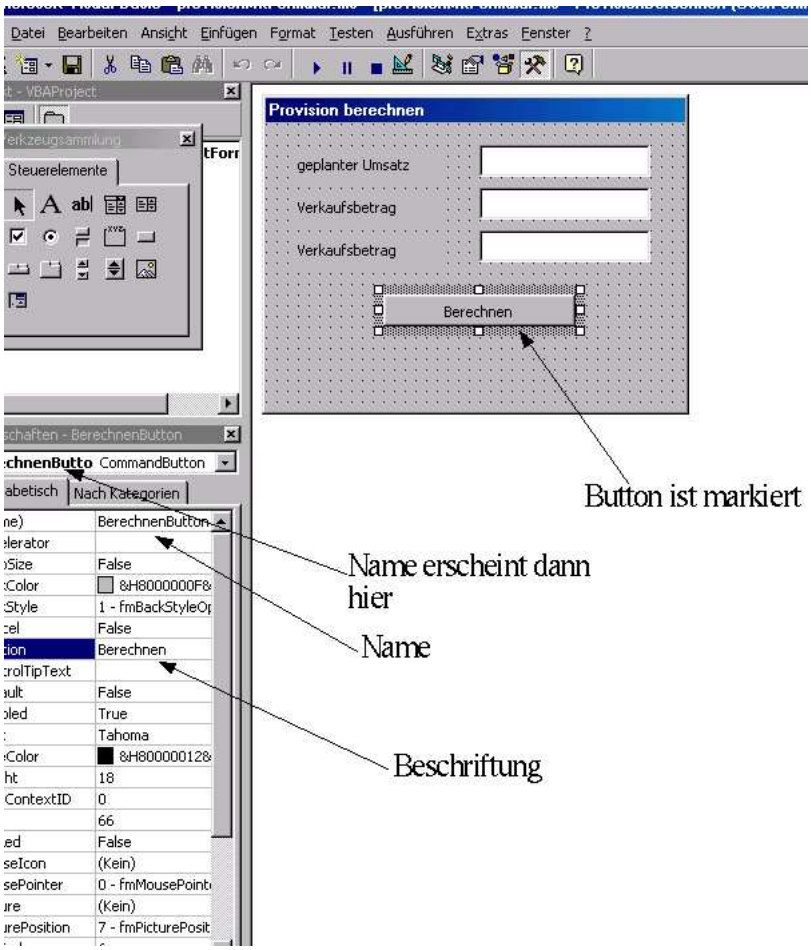

Abbildung 1.15: Hinzufügen eines Button

```
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-

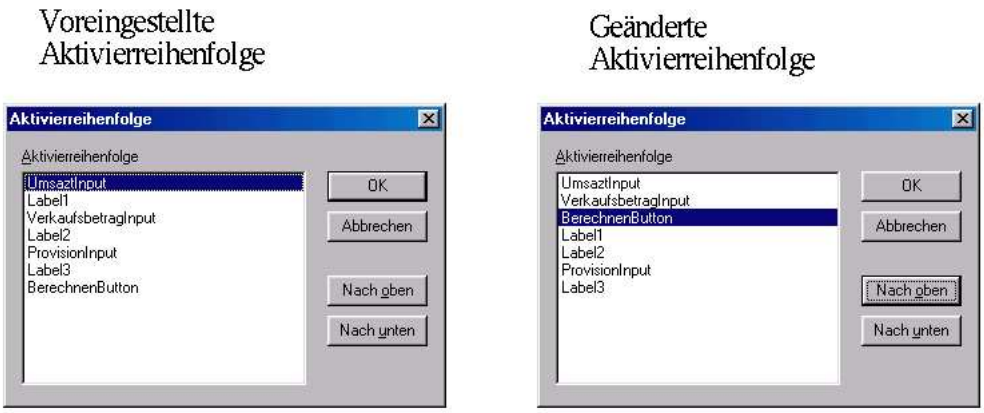

Abbildung 1.16: Änderung der Aktivierungreihenfolge

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.

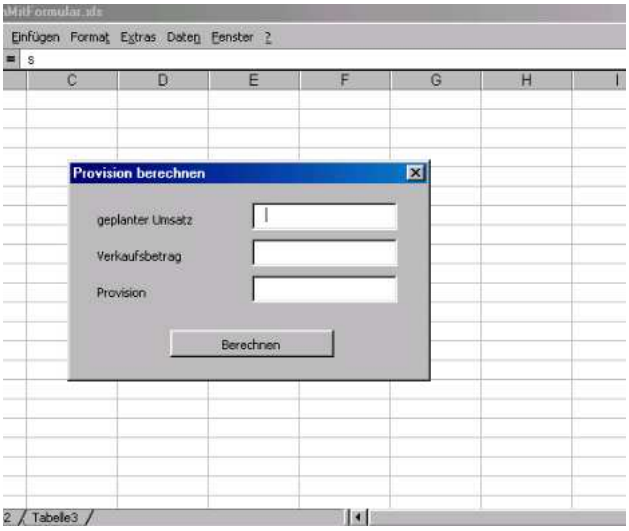

Abbildung 1.17: Unser schönes Formular

```
    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
```

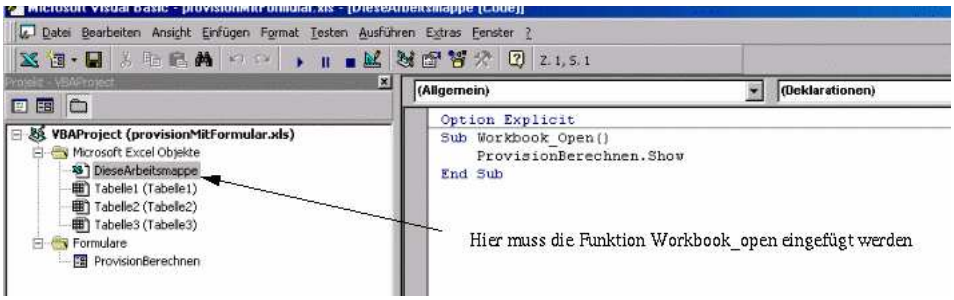

Abbildung 1.18: Das Formular im Autostart

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.

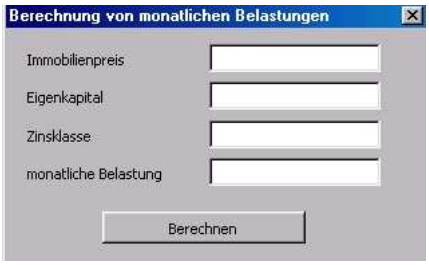

Abbildung 1.19: Das Formular der Zinsanwendung

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

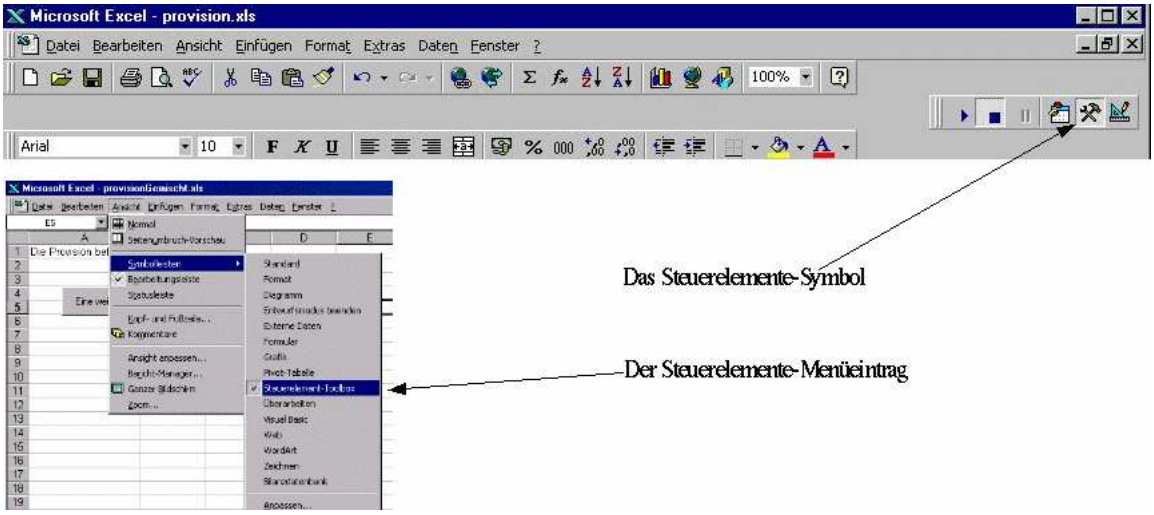

Abbildung 1.21: Öffnen der Steuerelemente Toolbox

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.

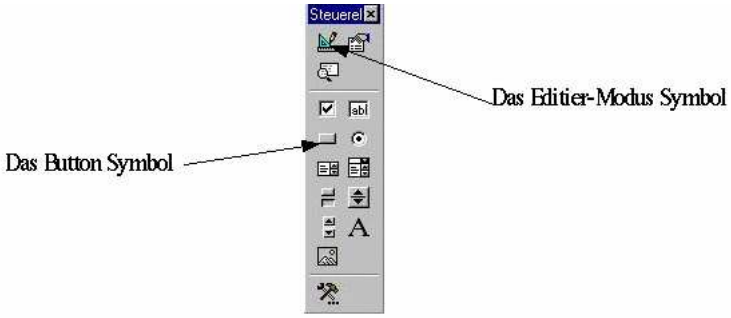

Abbildung 1.22: Die Steuerelemente-Toolbox

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.

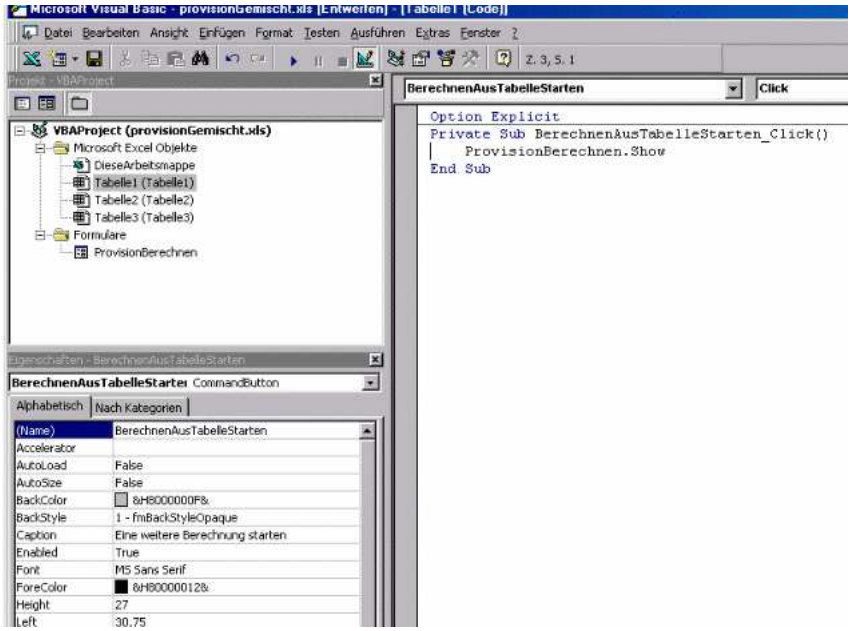

Abbildung 1.23: Umbenennen des Button und Programmierung der Ereignisprozedur

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.

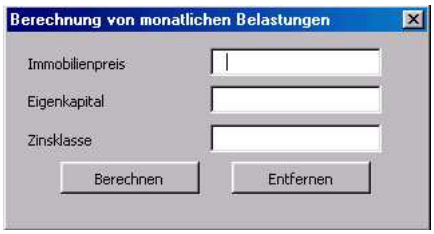

Abbildung 1.25: Das geänderte Formular des Belastungsrechnungsbeispiels

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.

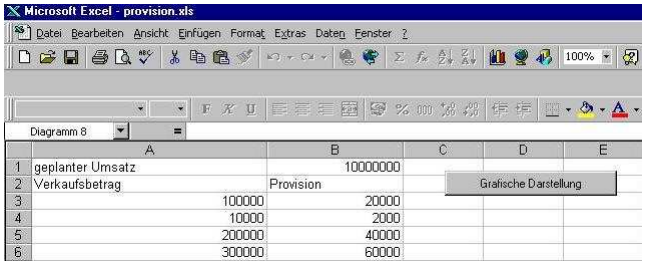

Abbildung 1.26: Provisionsbeispiel vor Darstellung der Grafik

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).

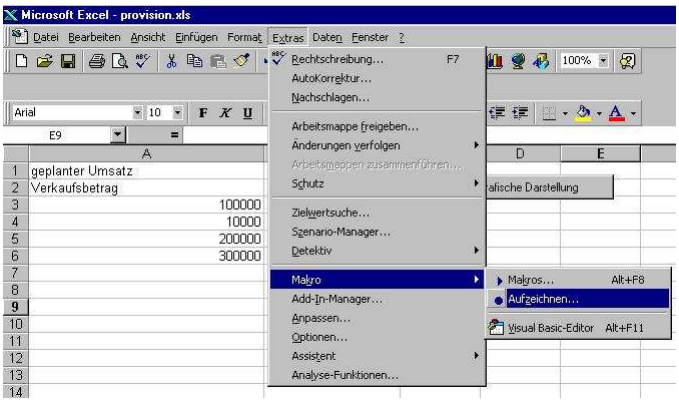

Abbildung 1.27: „Makro Aufzeichnen“ anschalten

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

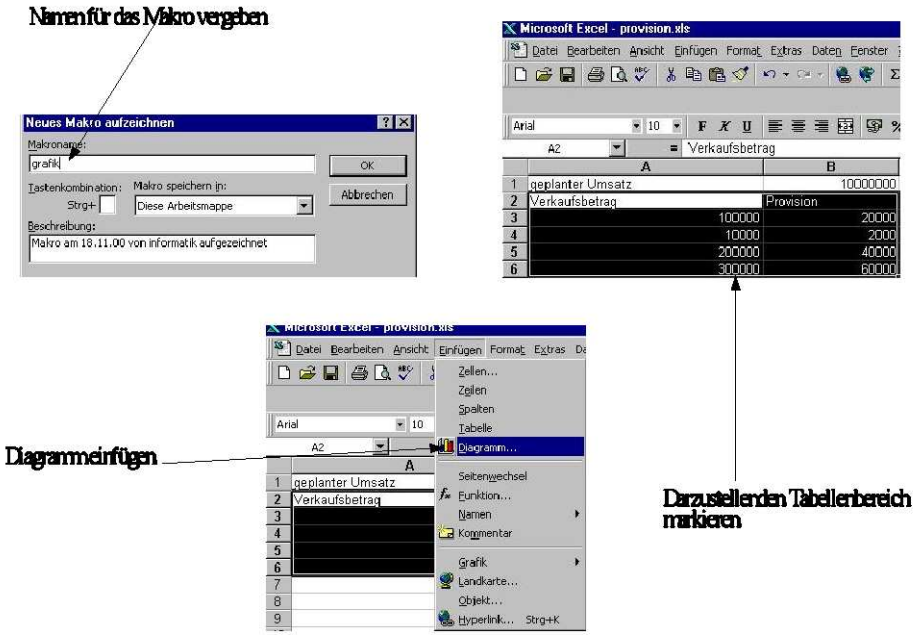

Abbildung 1.28: Erste Schritte Diagramm einfügen

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

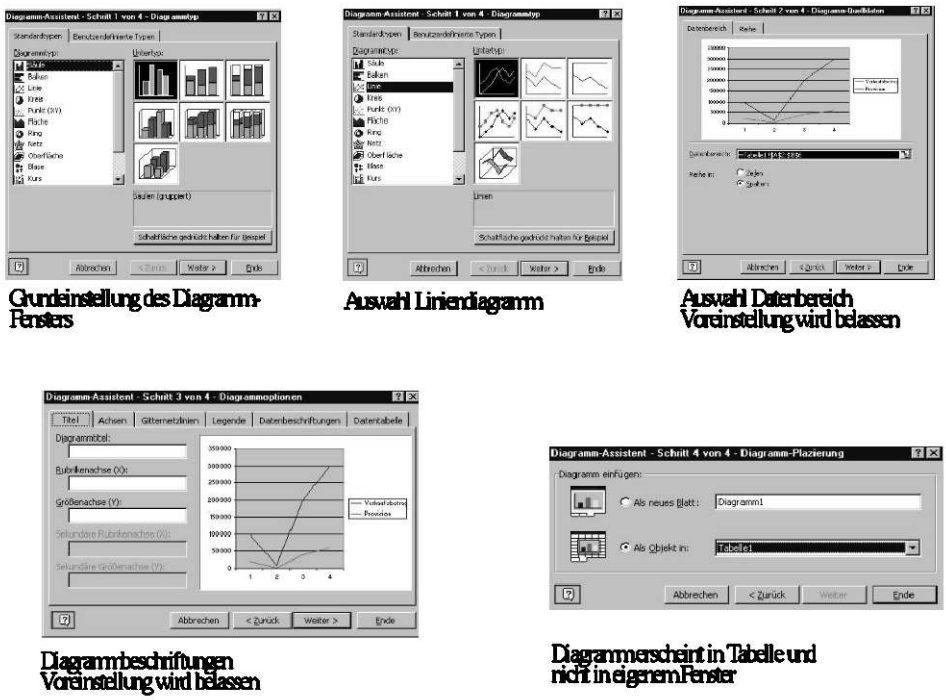

Abbildung 1.29: Dialoge zu Diagramm einfügen

(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
```

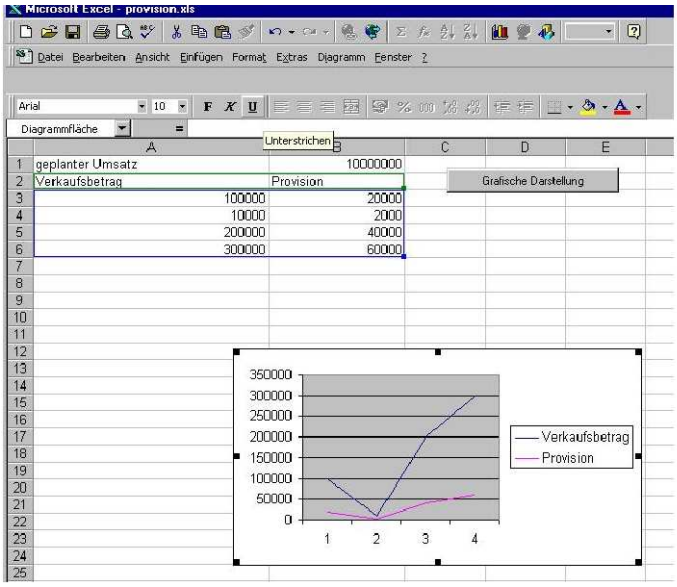

Abbildung 1.30: Die Grafik erscheint

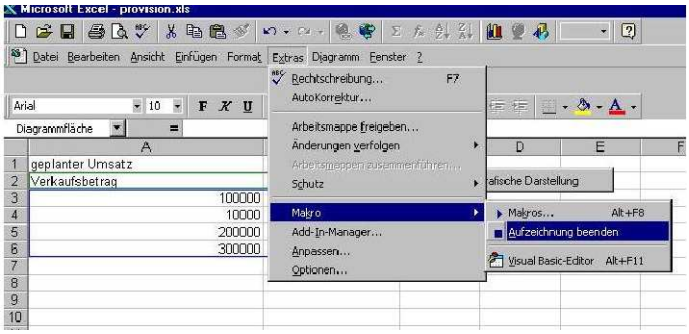

Abbildung 1.31: Der Aufzeichnungsmodus wird beendet

```
    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

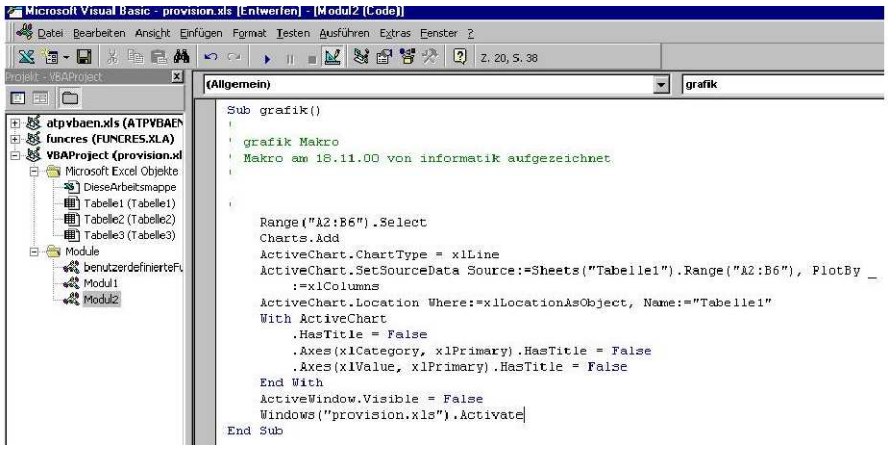

Abbildung 1.32: Das aufgezeichnete Modul

```
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.

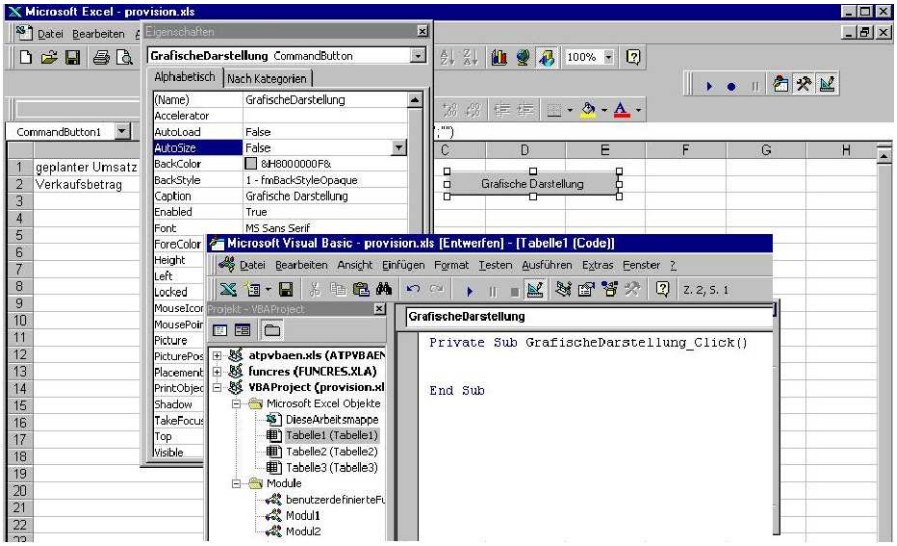

Abbildung 1.33: Schaltfläche mit von Excel erzeugtem Prozedur-Skelett

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
```

Kapitel 2

Zugriff aus VBA auf externe Datenbanken

2.1 ODBC

ODBC (Open Database Connectivity) ist ein Standard zum Zugriff auf, auf Servern liegenden, Datenbankensystemen. ODBC erlaubt selbstgeschriebenen Programmen, aber auch Anwendungen, wie 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. Danach muss eine Verbindung zur Datenbank konfiguriert werden. 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. 2.1).

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. 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. 2.2).

Die DSN ist nun benutzbar.

2.2 Nutzung der DSN von VBA

Ich demonstriere die Vorgehensweise an einem Beispiel. Wir wollen Raumnummer und -name der Vorlesungsräume unseres Fachbereichs aus der Intranetdatenbank auslesen und in eine Excel-Tabelle schreiben. Hierzu benötigen wir folgende Informationen:

- Die Datenbank heißt intranet, sie liegt auf dem Server pav050.fh-bochum.de.

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

Abbildung 2.1: Auswahl des ODBC-Treibers

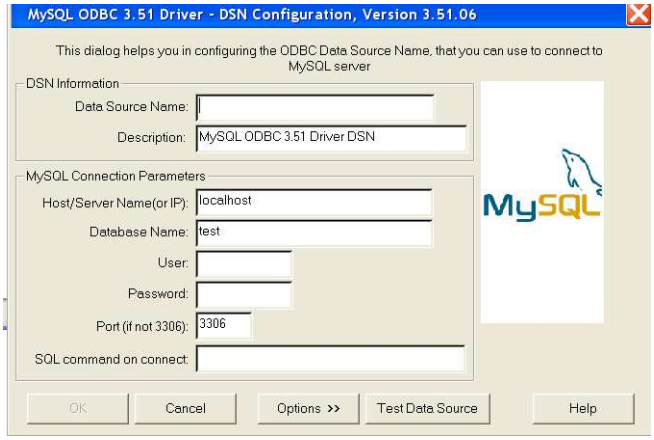

Abbildung 2.2: Einstellung der Parameter der Verbindung

- Die Tabelle der Räume heißt room und hat die in in Abb. 2.3 dargestellte Struktur.
- Die DSN heißt IntranetPav050.
- Ein Benutzer mit Zugriffsrechten auf diese Datenbank heißt iksy und hat das Passwort iksy [2].
- Die benötigten Felder in der Tabelle room sind room_nr und name.

VBA verfügt über mehrere Objektbibliotheken zum Zugriff auf ODBC-Datenbanken. Die derzeit aktuelle heißt ADO[3](ActiveX Data Objects). Vor der Benutzung muss die Biblothek 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[4].

Den Sourcecode der Anwendung zeigt Realisierung 2.1

Realisierung 2.1 *Lesen aus einer ODBC-Datenquelle*

[2]Das ist natürlich die Testdatenbank :-).

[3]Was sich bei Microsoft mit jeder Version des Office-Paketes und damit von VBA ändert. Die aktuelle heißt eigentlich ADO.net und unterscheidet sich von allen voran gegangenenen. Aber das ist bei Microsoft immer so.

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

Abbildung 2.3: Einstellung der Parameter der Verbindung

```
Attribute VB_Name = "raum"
Sub rauemeAuslesen()
    Dim conn As ADODB.Connection
    Dim ConnectionString As String
    Dim rec As ADODB.Recordset
    Dim SQL As String

    ConnectionString = "Provider=MSDASQL.1"
    ConnectionString = ConnectionString & ";Driver=MYSQL ODBC 3.51 Driver"
    ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
    ConnectionString = ConnectionString & ";Database=intranet"
    ConnectionString = ConnectionString & ";DNS=IntranetPav050"
    ConnectionString = ConnectionString & ";UID=iksy"
    ConnectionString = ConnectionString & ";PWD=iksy"

    Set conn = New ADODB.Connection
    conn.ConnectionString = ConnectionString
    conn.Open

    SQL = "Select "
    SQL = SQL & "room_nr, "
    SQL = SQL & "name "
    SQL = SQL & "from room "
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn

    'Ueberschriften
    Worksheets("Raum").Cells(1, 1) = "Raumnummer"
    Worksheets("Raum").Cells(1, 2) = "Raumname"

    ' Nun Inhalte
    i = 2
    Do While Not rec.EOF
        Worksheets("Raum").Cells(i, 1) = rec!room_nr
        Worksheets("Raum").Cells(i, 2) = rec!Name
        rec.MoveNext
        i = i + 1
    Loop
End Sub
```

Wir gehen den Code nun im Einzelnen durch.

```
Sub rauemeAuslesen()
 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[5]. 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 3.51 Driver"
ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
ConnectionString = ConnectionString & ";Database=intranet"
ConnectionString = ConnectionString & ";DNS=IntranetPav050"
ConnectionString = ConnectionString & ";UID=iksy"
ConnectionString = ConnectionString & ";PWD=iksy"
```

werden die zum Verbindungsaufbau notwendigen Informationen bereitgestellt. Die Zeile Provider=MSDASQL.1 sagt VBA, dass es sich um eine ODBC-Verbindung handeln soll[6]. MYSQL ODBC 3.51 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 die Raumnummern und -namen aus der Datenbank zu lesen, auf der Variablen SQL abgespeichert:

[5] Müßten Sie aus dem ersten Semester noch wissen!

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

```
SQL = "Select "
SQL = SQL & "room_nr, "
SQL = SQL & "name "
SQL = SQL & "from room "
SQL = SQL & "order by room_nr"
```

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. 2.4 illustriert einen Recordset.

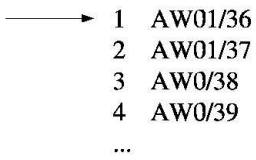

Abbildung 2.4: Grafische Darstellung eines Recordset

Ein Recordset enthält alle Datensätze, die das SQl-Kommando gefunden hat, plus einen Zeiger, der in Abb. 2.4 durch einen Pfeil veranschaulicht wird. Zu Anfang zeigt der Zeiger (wie auch in Abb. 2.4 dargestellt) auf den ersten gefundenen Datensatz. Nun stellt sich die Frage, wie wir die Ergebnisse aus dem Recordset heraus holen. Folgende Zeile[7]

```
raumname = rec!name
```

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

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 raumname nun den Wert AW01/36 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!room_nr. Wie schon gesagt beziehen sich rec!name und rec!room_nr 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. 2.5 dargestellt, aus. Mit diesem Wissen

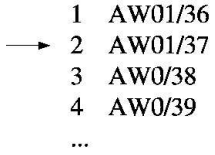

Abbildung 2.5: Unser Recordset nach rec.moveNext

können Sie jetzt den Rest des Beispiels verstehen. Die nächsten vier Zeilen sind einfach:

```
'Ueberschriften
Worksheets("Raum").Cells(1, 1) = "Raumnummer"
Worksheets("Raum").Cells(1, 2) = "Raumname"
i=2
```

Ein Kommentar, gefolgt von 2 Anweisungen, die Raumnummer, bzw. Raumname in die Zellen A1, resp. B1 des Tabelllenblatts Raum schreiben[8]. Dann wird die Variable i mit dem Wert 2 initialisiert. Die nächten Zeilen sind schwieriger:

```
Do While Not rec.EOF
    Worksheets("Raum").Cells(i, 1) = rec!room_nr
    Worksheets("Raum").Cells(i, 2) = rec!Name
    rec.MoveNext
    i = i + 1
Loop
```

Wie wir sehen, handelt es sich um eine while-Schleife. Bis auf das Abbruchkriterium können wir hier allerdings bereits alles verstehen. Beim ersten Schleifendurchlauf (i ist dann ja 2) wird die Zelle A2 mit der room_nr 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 1 geschrieben. Analoges gilt für B2 und rec!name und der Inhalt von B2 nach dem ersten Schleifendurchlauf ist AW01/36. 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 2 und B3 mit AW01/37, schiebt den Zeiger auf den dritten Datensatz und erhöht i wieder um Eins.

[8] Voraussetzung ist allerdings, das wir dem Tabellenblatt in Excel diesen Namen gegeben haben.

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 2.4 und 2.5 ist rec.EOF also false. Auch in Abb. 2.6 ist rec.EOF noch false.

Wenn wir allerdings davon ausgehen, und das tun wir jetzt, dass unsere Tabelle room und damit unser Recordset nur die dargestellten vier Datensätze enthält, dann zeigt der Zeiger des Recordsets 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[9] true.

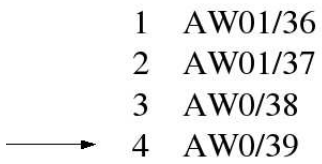

Abbildung 2.6: Unser Recordset mit dem Zeiger auf den letzten Datensatz

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

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

Kapitel 3

Das grosse Ganze

In diesem Kapitel wollen wir alles jenes, was Sie in den letzten beiden Semestern in Wirtschaftsinformatik gelernt haben, an einem Beispiel zusammenfassen. Hier sollen Sie auch erkennen, dass das, was Sie gelernt haben, durchaus sinnvoll und im beruflichen Alltag anwendbar ist[1].

Wir nehmen an, Sie arbeiten im Controlling eines Unternehmens und eine ihrer Aufgaben ist, vierteljährlich die Kennzahlen ihres Unternehmens mit Durchschnittskennzahlen der Branche, zu der ihr Arbeitgeber gehört, den Durchschnittskennzahlen der solventen Unternehmen und den Durchschnittskennzahlen der insolventen Unternehmen zu vergleichen (Benchmark). Darüber hinaus müssen Sie jederzeit in der Lage sein, solche Vergleiche für beliebige Perioden der Vergangenheit durchzuführen. Eine solche Analyse soll, wie in Abb. 3.1 dargestellt, aussehen. Dies ist, wie unschwer zu erkennen, eine Excel-Ausgabe.

Die Kennzahlen ihres Unternehmens werden jedes Vierteljahr berechnet, die Durchschnittskennzahlen werden von ihrem Arbeitgeber gekauft. Sie liegen als Tabellen in einer Datenbank vor.

3.1 Das Datenmodell

Zunächst überlegen wir uns ein Datenmodell. Es kann sein, dass Sie auf das Datenmodell keinen Einfluss haben, dann müssen die SQL-Kommandos, die wir später entwickeln, angepasst werden, es kann aber auch sein, dass Sie Einfluss nehmen können, dann nämlich, wenn Sie von Anfang in das Projekt mit einbezogen sind, und drittens kann es sein, dass Sie von Ihrer DV-Abteilung die Bereitstellung der Daten nach ihrem Modell beantragen können, was bedeuten würde, dass Sie die SQL-Kommandos nicht ändern müssen.

Wie Sie ja wissen (oder wissen sollten :-)), entwickelt man zunächst ein ERM. Das ERM könnte wie in Abb. 3.2 dargestellt, aussehen.

Jede Entity führt zu einer Tabelle, die Tabelle Kennzahl würde die Informationen über die Kennzahl enthalten z.B. Name, Formel, betriebswirtschaftliche Bedeutung etc. die Tabelle Jahr wäre eine Tabelle mit den in der Datenbank vorhandenen Jahren, die Tabelle Periode die mit den vorhandenen Perioden, sprich Quartalen, und die Tabelle Bewerteter enthielte Informationen über diejenigen, über die Kennzahlen vorliegen, in unserem Fall also das Unternehmen selber, die Branche, die solventen Unternehmen und die insolventen Unternehmen.

Dieses Datenmodell verwenden wir aber nicht, wir fassen Jahr und Periode zu einer Entity jahrPeriode zusammen und kommen zu dem in Abb. 3.3 dargestellten Modell.

Den Grund hierfür ist, dass das Datenmodell in Abb. 3.3 die SQL-Befehle, die wir später benötigen, wesentlich vereinfacht. Ich werde das, wenn wir die SQL-Befehle besprechen, erklären. Das

[1] Wäre sonst ja auch irgendwie doof, oder? :-)).

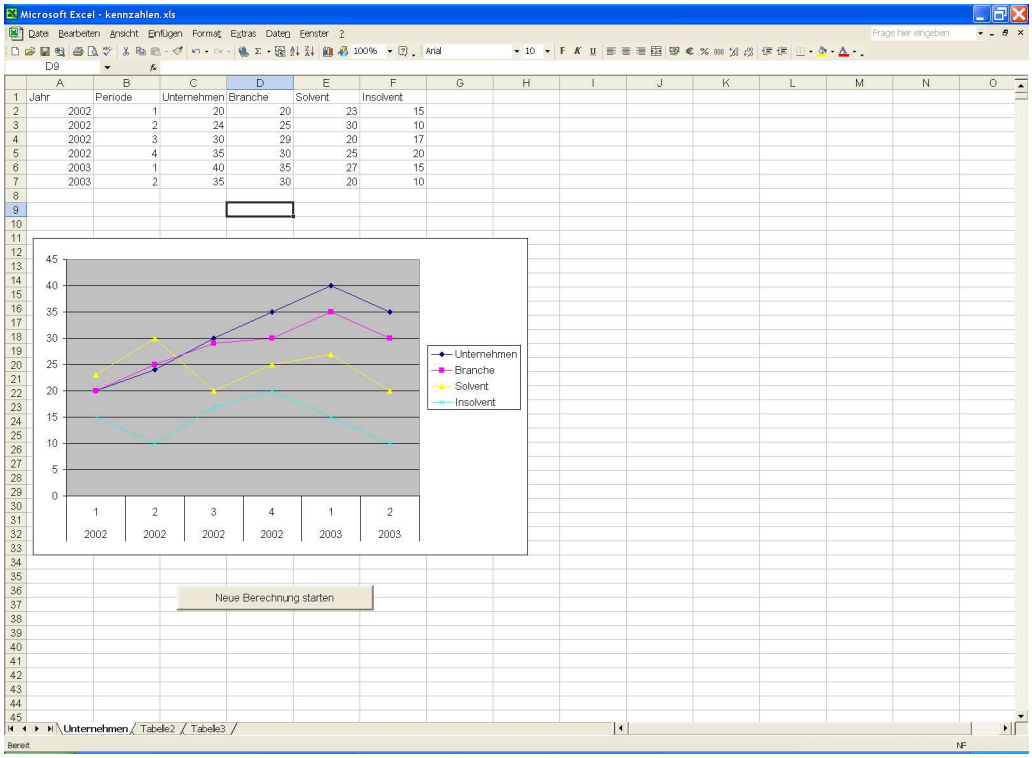

Abbildung 3.1: Der Benchmark

Datenmodell aus Abb. 3.2 ist natürlich auch richtig und könnte auch verwendet werden.

Unser Datenmodell führt zu folgender Tabellenstruktur.

Name	Primärschlüssel	Weitere Felder
Kennzahl	KennzahlNr	Name Namenskuerzel Beschreibung

Tabelle 3.1: Die Tabelle Kennzahl

Beispielhafte Datensätze in der Tabelle Kennzahl (vgl. Tab. 3.1) sehen wie in Tab. 3.2 dargestellt aus.

Beispielhafte Datensätze in der Tabelle Bewerteter (vgl. Tab. 3.3) sehen wie in Tab. 3.4 dargestellt aus.

Beispielhafte Datensätze in der Tabelle JahrPeriode (vgl. Tab. 3.5) sehen wie in Tab. 3.6 dargestellt aus.

Die n zu m zu o-Beziehung dieser Tabellen führt zu einer Verbindungstabelle. Der Verbindungstabelle geben wir den Namen kennzahlWert. Sie hat die in Tab. 3.7 dargestellten Eigenschaften.

Beispielhafte Datensätze in der Tabelle kennzahlWert sehen sehen wie in Tab. 3.8 dargestellt aus.

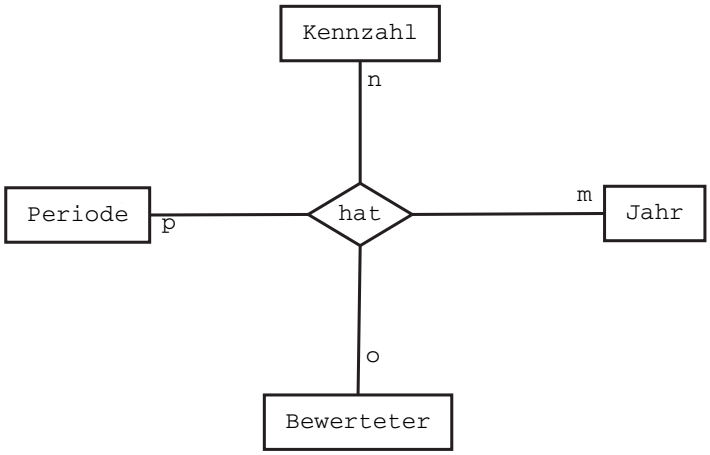

Abbildung 3.2: Ein erstes ERM

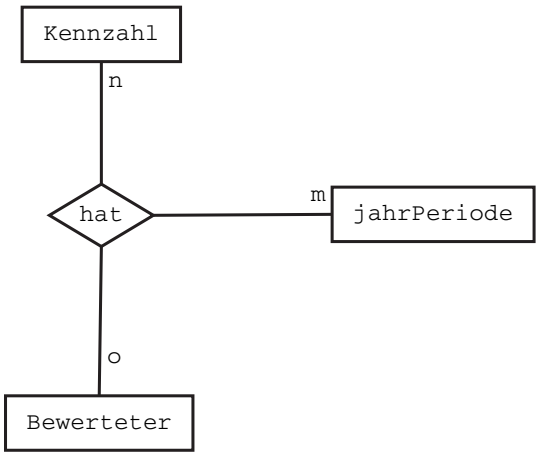

Abbildung 3.3: Ein verbessertes ERM

3.2 Das in Excel integrierte VBA-Programm - Erster Ansatz

Zunächst überlegen wir uns, welche Eingaben unsere Anwendung benötigt. Die Datenbank wird auf einem externen Datenbankserver liegen, also werden wir Benutzernamen und Paßwort eingeben müssen.

Weitere notwendige Eingaben sind:

- Name der Kennzahl
- Startjahr
- Startquartal
- Endjahr
- Endquartal

KennzahlNr	Name	Namenskuerzel	Beschreibung
1	Anlageintensität (in %)	anlin	Die Anlageintensität zeigt den ...
2	Arbeitsintensität (in %)	arbin	Die Arbeitsintensität ...

Tabelle 3.2: Die Tabelle Kennzahl im Beispiel

Name	Primärschlüssel	Weitere Felder
Bewerteter	BewerteterNr	Name

Tabelle 3.3: Die Tabelle Bewerteter

Also werden wir ein Formular entwerfen. Für Benutzername und Paßwort verwenden wir normale Textboxen. Für die restlichen fünf Eingaben bieten sich Listboxen an. Listboxen sind Rollfelder, wo unsere Benutzer aus vorgegebenen Möglichkeiten auswählen können. Und da wir der Datenbank ja entnehmen können, welche Kennzahlen es gibt und über welche Jahre und Perioden Informationen vorliegen, können wir dies aus der Datenbank lesen und die Listboxen damit vorbelegen. Wir erhalten das in Abb. 3.4 dargestellte Benutzerinterface.

Abbildung 3.4: Das Benutzerinterface

Abb. 3.5 zeigt die Steuerelemente-Toolbox mit einem Pfeil auf das neue in unserem Beispiel genutzte Steuerelement.

Nun entwickeln wir, wie in Kapitel 1 beschrieben, unser Eingabeformular. Einzig neu gegenüber Kapitel 1 ist die Benutzung der Comboboxen[2] und die farbliche Hinterlegung einiger Formular-

[2]Streng genommen verwenden wir keine Listboxen, sondern Comboboxen, die sowohl die Auswahl aus der Liste, als

BewerteterNr	Name
1	Unternehmen
2	Branche
3	solvente Unternehmen
4	insolvente Unternehmen

Tabelle 3.4: Die Tabelle Bewerteter im Beispiel

Name	Primärschlüssel	Weitere Felder
jahrPeriode	jahrPeriodeNr	Jahr Periode

Tabelle 3.5: Die Tabelle jahrPeriode

Elemente. Dies findet man aber immer in den Eigenschaften (vgl. Kapitel 1) der jeweiligen Formularelemente.

Wir vergeben folgende Namen für die Formularelemente:

- BenutzernameInput: Die Textbox zur Eingabe des Benutzernamens.
- PasswortInput: Die Textbox zur Eingabe des Passworts.
- KennzahlInput: Die Combobox zur Eingabe der Kennzahlen.
- StartjahrInput: Die Combobox zur Eingabe des Startjahrs.
- StartperiodeInput: Die Combobox zur Eingabe der Startperiode.
- EndjahrInput: Die Combobox zur Eingabe des Endjahrs.
- EndperiodeInput: Die Combobox zur Eingabe der Endperiode.
- AnmeldenBtn: Der Button, der die Berechnung startet.

Schwieriger schon ist die Vorbelegung der Comboboxen mit den Werten aus der Datenbank. Denn dies muss automatisch erfolgen, wenn das Formular geladen wird[3]. Aber auch dafür stellt VBA eine Möglichkeit zur Verfügung: Wenn wir eine Prozedur mit dem Namen

```
Private Sub Userform_initialize()
```

im zum Formular gehörenden Code-Fenster schreiben, führt VBA diese Prozedur tatsächlich immer dann aus, wenn das Formular geladen wird[4]. Nun müssen wir uns den VBA-Code dieser Prozedur überlegen. Da dies schwieriger ist, schreiben wir dazu Pseudocode:

Pseudocode 3.1 *Initialisieren des Kennzahlen-Formulars*

1. Verbindung zur Datenbank aufbauen
2. Vorhandene Kennzahlen auslesen

auch die Eingabe eigener Texte erlauben. Diesen gegenüber gibt es in VBA auch Listboxen, wo man nur aus vorgegebenen Elementen auswählen kann. Die Listboxen sehen aber einmal nicht so gut aus, was natürlich kein Argument ist, sind aber auch fehlerhaft implementiert, was ein Argument ist.

[3] Informatikdeutsch dafür ist initialisieren.

[4] initialize ist englisch und heißt initialisieren :-)) und das ist ja genau das, was wir wollen, wir wollen das Formular beim Laden initialisieren.

JahrPeriodeNr	Jahr	Periode
1	1998	1
2	1998	2
3	1998	3
4	1998	4
5	1999	1
6	1999	2

Tabelle 3.6: Die Tabelle JahrPeriode im Beispiel

Name	Primärschlüssel	Weitere Felder
kennzahlWert	KennzahlNr BewerteterNr JahrPeriodeNr	Wert

Tabelle 3.7: Die Tabelle kennzahlWert

3. Kennzahlen-Combobox mit ermittelten Kennzahlen vorbelegen
4. Vorhandene Jahre auslesen
5. Start- und Endjahr-Combobox mit ermittelten Jahren vorbelegen
6. Vorhandene Perioden auslesen
7. Start- und Endperiode-Combobox mit ermittelten Perioden vorbelegen

Natürlich wollen wir eine gut strukturierte Anwendung entwickeln und das bedeutet, wir arbeiten funktionsorientiert (vgl. Kapitel über Funktionen und Prozeduren im VBA-Script). Zumindestens den Aufbau der Verbindung zur Datenbank und das Auslesen der Kennzahlen, sowie der Jahre und Perioden, lagern wir in Prozeduren oder Funktionen aus.

Wir gehen hier sogar noch einen Schritt weiter. Für Prozeduren und Funktionen, die in irgendeiner Weise zusammengehören, erzeugen wir ein eigenes Modul (Über den Menüpunkt Einfügen -> Modul der VBA-Entwicklungsumgebung). Die Funktion für den Aufbau der Verbindung zur Datenbank schreiben wir daher in eins Modul mit dem Namen datenbankfunktionen. Und hier ist der VBA-Code dieser Funktion, deren Name connectToDB ist:

Realisierung 3.1 *Aufbau einer Verbindung zu einer ODBC-Datenquelle*

```
Function connectToDB(benutzer As String, _
                     passwort As String, _
                     conn As ADODB.Connection, _
                     fehlerNr As Long, _
                     fehlerText As String) As Boolean

' Aufbau der Verbindung zur Datenbank
' Verbindung kommt ueber den Parameter conn zurueck
' Funktion gibt bei Nichterfolg false zurueck
' Ueber fehlerNr und fehlerText werden im Fehler-Fall
' FehlerNummer und Fehler zurueckgegeben
    Dim ConnectionString As String
    On Error GoTo Fehler

    ConnectionString = "Provider=MSDASQL.1"
```

KennzahlNr	BewerteterNr	JahrPeriodeNr	Wert
1	1	1	10.45
1	1	2	12.56
1	1	3	34.44
1	1	4	34.67
1	1	5	35
1	1	6	36

Tabelle 3.8: Die Tabelle kennzahlWert im Beispiel

Abbildung 3.5: Die Steuerelemente-Toolbox

```
    ConnectionString = ConnectionString & ";Driver=MYSQL ODBC 3.51 Driver"
    ConnectionString = ConnectionString & ";Server=pav050.fh-bochum.de"
    ConnectionString = ConnectionString & ";Database=kennzahl"
    ConnectionString = ConnectionString & ";DNS=Kennzahl"
    ConnectionString = ConnectionString & ";UID=" & benutzer
    ConnectionString = ConnectionString & ";PWD=" & passwort

    Set conn = New ADODB.Connection
    conn.ConnectionString = ConnectionString
    conn.Open

    connectToDB = True
    Exit Function
Fehler:
    fehlerNr = Err
    fehlerText = Error(Err)
    connectToDB = False
End Function
```

Diese Funktion bekommt also als Eingabeparameter den Namen des Benutzers, der an der Datenbank angemeldet werden soll, sowie dessen Paßwort übergeben. Über den Parameter conn gibt sie die Verbindung zurück, auf fehlerNr bzw. fehlerText stehen Nummer und Bezeichnung des Fehlers, falls der Verbindungsaufbau zur Datenbank nicht funktioniert hat. Über ihren Namen gibt die Funktion true zurück, wenn alles geklappt hat oder false, falls die Verbindung nicht aufgebaut werden konnte.

Doch nun zum Code. Den Aufbau des ConnectionString müßten Sie nach dem in Kapitel 2.2 Gesagten verstehen. Die Funktion wird versuchen, eine ODBC-Verbindung (Provider=MSDASQL.1) zu einem MySql-Server (Driver=MYSQL ODBC 3.51 Driver) auf dem Server pav050 (Server=pav050.fh-bochum.de) zur Datenbank kennzahl (Database=kennzahl) über die Datenquelle Kennzahl (DNS=Kennzahl) mit dem der Funktion übergebenen Benutzernamen (UID=” & benutzer) und Paßwort (PWD=” & passwort) aufzubauen. Dann wird ein Verbindungsobjekt erzeugt (Set conn = New ADODB.Connection), der ConnectionString der Verbindung zugewiesen (conn.ConnectionString = ConnectionString) und die Verbindung aufgebaut (conn.Open). All dies wurde in Kapitel 2.2 erklärt.

Neu ist die Fehlerbehandlung. Die Zeile

```
    On Error GoTo Fehler
```

veranlasst VBA dann, wenn ein Fehler auftritt -was in unserem Fall nur sein kann, dass der Verbindungsaufbau zur Datenbank nicht erfolgreich war-, nach einer Zeile im Code zu suchen, die mit Fehler: beginnt. Dabei darf vor Fehler: nicht einmal ein Blanc stehen. Gibt es keine solche Zeile, erhalten Sie eine Fehlermeldung des VBA-Editors. In unserem Fall gibt es eine solche Zeile. Im Fehlerfall geht[5] VBA direkt zu dieser Zeile und führt nur noch den darauf folgenden Code aus. Dies ist:

```
Fehler:
     fehlerNr = Err
     fehlerText = Error(Err)
     connectToDB = False
End Function
```

Err ist eine VBA interne Variable und enthält die Nummer des zuletzt aufgetretenen Fehlers, Error(Err) ist eine VBA interne Funktion und gibt einen Text zu der übergebenen Fehlernummer zurück. Wir sehen also: Im Fehlerfall wird

- die Variable fehlerNr mit der Fehlernummer des aufgetretenen Fehlers besetzt,
- die Variable fehlerText mit dem Fehlertext des aufgetretenen Fehlers besetzt und
- der Wert der Funktion connectToDB auf false gesetzt.

Beachten Sie, dass die Zeile

```
    Exit Function
```

vor der Sprungstelle

```
    Fehler:
```

wichtig ist. Denn das, was nach Fehler: folgt, soll nur dann ausgeführt werden, wenn wirklich ein Fehler aufgetreten ist. VBA nimmt Sprungstellen aber nur mit dem Kommando GoTo wahr, ansonsten werden Sprungstellen ignoriert. Bedeutet natürlich, dass, wenn kein Exit Function vor der Sprungstelle steht, der Code hinter der Sprungstelle immer ausgeführt wird, was wir natürlich nicht wollen.

Soweit der Code zum Aufbau der Verbindung zur Datenbank. Beschäftigen wir uns nun mit dem Auslesen der für die Comboboxen benötigten Werte. Zunächst einmal legen wir für solche Prozeduren oder Funktionen auch wieder ein eigenes Modul an und geben diesem Modul den Namen tabelleninhalteAuslesen. Wir starten mit dem Auslesen der Kennzahlen. Und hier ist der VBA-Code dieser Prozedur deren Name getKennzahlen ist:

Realisierung 3.2 *Auslesen der in der Datenbank vorhandenen Kennzahlen*

```
Sub getKennzahlen(conn As ADODB.Connection, kennzahlen() As String)
' Holen des Inhalts der Tabelle Kennzahl
    Dim i As Integer
    Dim SQL As String
    Dim rec As ADODB.Recordset
    Dim anzahl As Integer

    SQL = "SELECT count(*) As Anzahl "
    SQL = SQL & "FROM kennzahl"
```

[5] GoTo ist Englisch, bedeutet gehe zu :-))

```
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    anzahl = rec!anzahl
    ReDim kennzahlen(anzahl)
    rec.Close
    SQL = "SELECT name "
    SQL = SQL & "FROM kennzahl"
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    i = 1
    Do While Not rec.EOF
        kennzahlen(i) = rec!Name
        rec.MoveNext
        i = i + 1
    Loop
    rec.Close
End Sub
```

Die Funktion getKennzahlen bekommt die Verbindung zur Datenbank übergeben, muss sie ja auch, da sie ja ein SQL-Kommado an die Datenbank schicken muss (vgl. Kapitel 2.2). Zurück gibt sie ein Array (vgl. VBA-Script, Kapitel über Arrays) mit den Namen der Kennzahlen. Die Funktion ermittelt mit dem ersten SQL-Befehl die Anzahl der in der Datenbank vorhandenen Kennzahlen. Dies erfolgt, indem zunächst das hierfür notwendige SQL-Kommando auf der Variablen SQL zusammengestellt wird,

```
    SQL = "SELECT count(*) As Anzahl "
    SQL = SQL & "FROM kennzahl"
```

sodann ein Recordset erzeugt und das Kommando durch das Recordset über die übergebene Verbindung an die Datenbank geschickt wird.

```
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
```

Danach wird die Dimension des Arrays kennzahlen verändert, indem wir es auf die ermittelte Anzahl setzen. Das Recordset wird wieder geschlossen.

```
    anzahl = rec!anzahl
    ReDim kennzahlen(anzahl)
    rec.Close
```

Nun stellen wir auf der Variablen SQL das für das Auslesen der Namen der Kennzahlen notwendige SQL-Kommando zusammen und schicken es durch ein neu erzeugtes Recordset an die Datenbank.

```
    SQL = "SELECT name "
    SQL = SQL & "FROM kennzahl"
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
```

Sodann füllen wir in einer Schleife unser Array mit den auf dem Recordset vorhandenen Werten:

```
    i = 1
    Do While Not rec.EOF
        kennzahlen(i) = rec!Name
        rec.MoveNext
        i = i + 1
    Loop
```

Zum Schluss schließen wir das Recordset.

```
    rec.Close
```

Als nächstes beschäftigen wir uns mit dem Auslesen der Jahre. Und hier ist der VBA-Code dieser Prozedur, deren Name getJahre ist:

Realisierung 3.3 *Auslesen der in der Datenbank vorhandenen Jahre*

```
Sub getJahre(conn As ADODB.Connection, jahre() As Integer)
' Holen der Jahre
    Dim i As Integer
    Dim SQL As String
    Dim rec As ADODB.Recordset
    Dim anzahl As Integer

    SQL = "SELECT distinct count(distinct jahr) As Anzahl "
    SQL = SQL & "FROM jahrPeriode "
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    anzahl = rec!anzahl
    ReDim jahre(anzahl)
    rec.Close
    SQL = "SELECT distinct jahr "
    SQL = SQL & "FROM jahrPeriode "
    SQL = SQL & "Order by jahr"
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    i = 1
    Do While Not rec.EOF
        jahre(i) = rec!jahr
        rec.MoveNext
        i = i + 1
    Loop
    rec.Close
End Sub
```

Der Code ist völlig analog zu dem von Realisierung 3.2. Einziger Unterschied ist das SQL-Kommando. Hier verweise ich auf das Datenbank-Script.

Zum Abschluss nun die Funktion zum Auslesen der Perioden:

Realisierung 3.4 *Auslesen der in der Datenbank vorhandenen Perioden*

```
Sub getPerioden(conn As ADODB.Connection, perioden() As Integer)
' Holen der Perioden
    Dim i As Integer
    Dim SQL As String
    Dim rec As ADODB.Recordset
    Dim anzahl As Integer

    SQL = "SELECT count(distinct periode) As Anzahl "
    SQL = SQL & "FROM jahrPeriode"
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    anzahl = rec!anzahl
```

```
    ReDim perioden(anzahl)
    rec.Close
    SQL = "SELECT distinct periode "
    SQL = SQL & "FROM jahrPeriode"
    SQL = SQL & "Order by periode"
    Set rec = New ADODB.Recordset
    rec.Open SQL, conn
    i = 1
    Do While Not rec.EOF
        perioden(i) = rec!periode
        rec.MoveNext
        i = i + 1
    Loop
    rec.Close
End Sub
```

Da wir nun die wesentlichen Prozeduren und Funktionen entwickelt haben[6] können wir uns die Implementierung der Initialisierungsprozedur ansehen:

Realisierung 3.5 *Initialisierung des Kennzahlenformulars*

```
Private Sub Userform_initialize()
    Dim conn As ADODB.Connection
    Dim fehlerNr As Long
    Dim fehlerText As String
    Dim anzahlJahre As Integer
    Dim anzahlPerioden As Integer
    Dim anzahlKennzahlen As Integer
    Dim jahre() As Integer
    Dim perioden() As Integer
    Dim kennzahlen() As String
    Dim benutzer As String
    Dim passwort As String
    Dim meldung As String
    Dim i As Integer

    benutzer = "kennzahl"
    passwort = "kennzahl"

    If Not connectToDB(benutzer, passwort, conn, fehlerNr, fehlerText) Then
        meldung = "Fehler aus Anmelden-Initialisieren "
        meldung = meldung & "ODBC-Fehler"
        Call gibFehlermeldungAus(meldung, fehlerNr, fehlerText)
        Exit Sub
    End If
    Call getKennzahlen(conn, kennzahlen)
    anzahlKennzahlen = UBound(kennzahlen, 1)
    For i = 1 To anzahlKennzahlen
        KennzahlInput.AddItem (kennzahlen(i))
    Next i
    Call getJahre(conn, jahre)
    anzahlJahre = UBound(jahre, 1)
    For i = 1 To anzahlJahre
```

[6] Wir werden sehen, dass wir noch eine weitere Prozedur benötigen.

```
        StartjahrInput.AddItem (jahre(i))
        EndjahrInput.AddItem (jahre(i))
    Next i
    Call getPerioden(conn, perioden)
    anzahlPerioden = UBound(perioden, 1)
    For i = 1 To anzahlPerioden
        StartperiodeInput.AddItem (perioden(i))
        EndperiodeInput.AddItem (perioden(i))
    Next i
    conn.Close
End Sub
```

Nach der Deklaration der benötigten Variablen kommen wir zu einem ersten Problem der Anwendung. Wenn wir das in Abb. 3.1 gezeigte Benutzerinterface realisieren wollen, müssen wir ja auf die Datenbank zugreifen, bevor der Benutzer Benutzername und Paßwort eingegeben hat. Denn in dem Formular kann er seinen Benutzernamen und sein Paßwort eingeben, was wir ab da ja auch benutzen können, aber die Darstellung des Formulars erfordert einen Datenbankzugriff bevor diese Informationen zur Verfügung stehen. Weil wir ja bereits bei der Darstellung des Formulars die Comboboxen mit Inhalten füllen müssen, die wir der Datenbank entnehmen.

Dies bedeutet, wir müssen die Anmeldung mit einem Benutzer durchführen, dessen Namen und Paßwort im VBA-Quellcode stehen und damit allen, die Zugriff auf die Excel-Datei haben, offen liegen. Dies ist insofern kein Problem, wenn Sie das Programm alleine nutzen und dafür sorgen können, das kein anderer die Exceldatei öffnen kann. Ein Problem ensteht, wenn Kollegen von Ihnen dieses Programm auch nutzen sollen. Denn dann könnten diese ihre Benutzerkennung und ihr Paßwort sehen. In so einem Fall darf die Anmeldung in der Initialisierungsprozedur nicht unter ihrem Benutzernamen erfolgen. Man muss vielmehr einen Datenbanknutzer anlegen (oder von den Datenbankleuten anlegen lassen), dessen Rechte auf genau das beschränkt sind, was man im Anmeldeformular ohnehin sieht. Dies bedeutet, dieser Benutzer darf die Felder Jahr und Periode in der Tabelle JahrPeriode und das Feld Name in der Tabelle Kennzahl selektieren. Dies ist aber glücklicherweise in den meisten Datenbanksystemen problemlos möglich. Abb. 3.6 zeigt einen solchen Benutzer (den Benutzer kennzahl) für die MySql-Datenbank.

Abbildung 3.6: Ein MySql-Benutzer mit stark eingeschränkten Rechten

Damit erklären sich die ersten Zeilen der Initialisierungsprozedur

```
benutzer = "kennzahl"
passwort = "kennzahl"

If Not connectToDB(benutzer, passwort, conn, fehlerNr, fehlerText) Then
    meldung = "Fehler aus Anmelden-Initialisieren "
    meldung = meldung & "ODBC-Fehler"
    Call gibFehlermeldungAus(meldung, fehlerNr, fehlerText)
```

```
        Exit Sub
    End If
```

Der Benutzername und das Paßwort werden auf den rechtelosen Benutzer gesetzt, welchselbiger dann bei der Datenbank angemeldet wird. Beim Aufruf der Funktion connectToDB sieht man nun auch, was ich mit der fehlenden Prozedur gemeint habe. Falls ein Fehler auftritt, wird die Fehlermeldung nicht direkt ausgegeben, an Stelle dessen nutzen wir die Prozedur gibFehlermeldungAus. Dies einfach deswegen, damit Fehlermeldungen aus allen unseren Programmteilen einheitlich formattiert ausgegeben werden. Die Prozedur gibFehlermeldungAus werde ich zum Abschluss der Diskussion des Codes des Initialisierungsformulars vorstellen.

Ich benutze hier auch die alternative Syntax für Prozeduraufrufe in VBA. Anstelle

```
gibFehlermeldungAus meldung, fehlerNr, fehlerText
```

kann man auch schreiben[7]

```
Call gibFehlermeldungAus(meldung, fehlerNr, fehlerText)
```

Dies geht allerdings nur bei Prozeduren, bei Funktionen geht das nicht. Im gesamten Beispiel werden Prozeduren über die alternative Syntax aufgerufen.

War der Verbindungsaufbau zur Datenbank erfolgreich, werden die Kennzahlen geholt.

```
Call getKennzahlen(conn, kennzahlen)
```

Danach stellen wir fest, wieviele Kennzahlen vorhanden sind. Dies ist die Dimension des Arrays kennzahlen (vgl. Kapitel Arrays im VBA-Script).

```
anzahlKennzahlen = UBound(kennzahlen, 1)
```

Und nun besetzen wir unsere Kennzahlen-Combobox mit den ermittelten Kennzahlen:

```
    For i = 1 To anzahlKennzahlen
        KennzahlInput.AddItem (kennzahlen(i))
    Next i
```

Wir sehen hier, wie das funktioniert, eine Combobox mit Werten zu füllen. Wir schreiben einfach den Namen der Combobox (hier KennzahlInput), den Punkt, AddItem und dann den Text der der Combobox hinzu gefügt werden soll (kennzahlen(i)). Damit ist die Diskussion der Initialisierungsprozedur beendet, denn der Aufruf der Ausleseprozeduren für Jahre und Perioden, sowie die Belegung der entsprechden Comboboxen ist ja vollständig analog der Kennzahlengeschichte.

Bleibt also die Prozedur gibFehlermeldungAus. Für diese Funktion erzeugen wir ebenfalls ein neues Modul mit Namen fehlerbehandlung. Und hier ist der VBA-Code dieser Funktion:

Realisierung 3.6 *Darstellung von Datenbankfehlermeldungen*

```
Sub gibFehlermeldungAus(meldung As String, fehlerNr As Long, fehlerText As String)
    meldung = meldung & " Fehlernummer: " & fehlerNr & " Fehlertext" & fehlerText
    MsgBox (meldung)
End Sub
```

Die Realisierung dieser Prozedur ist nicht wirklich spektakulär. Letztlich gibt sie nur die übergebenen Fehlerinformationen in einer Messagebox aus. Dennoch ist es eine gute Idee, dies so zu machen. Denn sollten wir auf die Idee kommen, Fehlermeldungen nicht mehr auszugeben, sondern in einer Datei zu

[7] Man beachte, mal Klammern, mal nicht, sie sind sehr konsistent in dem was sie machen, die Microsoft-Entwickler :-).

sammeln, ist nur eine Änderung dieser Funktion erforderlich und alle Fehlermeldungen werden nicht mehr am Bildschirm angezeigt, sondern landen in dieser Datei.

Damit ist die Entwicklung der Initialisierungsprozedur fertig und das Benutzerinterface kann dargestellt werden. Bleibt noch die eigentliche Anwendung, nämlich das, was gemacht werden muss, wenn der Benutzer seine Eingaben getätigt hat und die Ergebnisse in die Excel-Tabelle geschrieben werden müssen.

Auch hier schreiben wir Pseudocode:

Pseudocode 3.2 *Erzeugung der Ausgabe in der Excel-Tabelle*

1. Verbindung zur Datenbank aufbauen
2. startjahrPeriodeNr aus Startjahr und Startperiode ermitteln
3. endjahrPeriodeNr aus Endjahr und Endperiode ermitteln
4. Wert des Unternehmens und jahrPeriodeNr holen
5. Durchschnittswert der Branche holen
6. Durchschnittswert der solventen Unternehmens holen
7. Durchschnittswert der insolventen Unternehmens holen
8. Zu jahrPeriodeNr das Jahr und die Periode ermitteln
9. Werte in die Excel-Tabelle schreiben
10. Grafik erzeugen

Ich zeige nun zunächst das Programm, danach werde ich es Schritt für Schritt erklären. Wenn Prozeduren oder Funktionen genutzt werden, werde ich den Sourcecode der Prozeduren oder Funktionen an der Stelle, wo sie genutzt werden, erklären.

Realisierung 3.7 *Erzeugung der Ausgabe*

```
Private Sub AnmeldenBtn_Click()
    Dim benutzer As String
    Dim passwort As String
    Dim conn As ADODB.Connection
    Dim fehlerNr As Long
    Dim fehlerText As String
    Dim kennzahl As String
    Dim startjahr As Integer
    Dim startperiode  As Integer
    Dim startjahrPeriode As Integer
    Dim endjahr As Integer
    Dim endperiode As Integer
    Dim endjahrPeriode As Integer
    Dim SQLFromWhere As String
    Dim SQL As String
    Dim i As Integer
    Dim rec As ADODB.Recordset
    Dim rec2 As ADODB.Recordset
    Dim rec3 As ADODB.Recordset
    Dim rec4 As ADODB.Recordset
    Dim rec5 As ADODB.Recordset

    benutzer = benutzernameInput.Text
```

```
passwort = passwortInput.Text

If Not connectToDB(benutzer, passwort, conn, fehlerNr, fehlerText) Then
    MsgBox ("Benutzername und Passwort stimmen nicht überein!")
    Exit Sub
End If
Call initialisiereTabelle(1, 1000, 1, 7)
kennzahl = KennzahlInput.Text
startjahr = StartjahrInput.Text
startperiode = StartperiodeInput.Text
endjahr = EndjahrInput.Text
endperiode = EndperiodeInput.Text

startjahrPeriode = getJahrPeriodeAusJahrUndPeriode(conn, _
                                                   startjahr, startperiode)
endjahrPeriode = getJahrPeriodeAusJahrUndPeriode(conn, _
                                                   endjahr, endperiode)
SQLFromWhere = macheSQLFromWhere(1, startjahrPeriode, endjahrPeriode, kennzahl)
'hole Jahr periode und Unternehmenswert
SQL = "Select "
SQL = SQL & "kennzahlWert.jahrPeriodeNr, "
SQL = SQL & "kennzahlWert.wert "
SQL = SQL & SQLFromWhere

Set rec = New ADODB.Recordset
rec.Open SQL, conn

' hole Vergleichsdaten Branche
SQLFromWhere = macheSQLFromWhere(2, startjahrPeriode, endjahrPeriode, kennzahl)
SQL = "Select "
SQL = SQL & "kennzahlWert.wert "
SQL = SQL & SQLFromWhere
Set rec2 = New ADODB.Recordset
rec2.Open SQL, conn

' hole Vergleichsdaten solvente Unternehmen
SQLFromWhere = macheSQLFromWhere(3, startjahrPeriode, endjahrPeriode, kennzahl)
SQL = "Select "
SQL = SQL & "kennzahlWert.wert "
SQL = SQL & SQLFromWhere
Set rec3 = New ADODB.Recordset
rec3.Open SQL, conn

' hole Vergleichsdaten insolvente Unternehmen
SQLFromWhere = macheSQLFromWhere(4, startjahrPeriode, endjahrPeriode, kennzahl)
SQL = "Select "
SQL = SQL & "kennzahlWert.wert "
SQL = SQL & SQLFromWhere
Set rec4 = New ADODB.Recordset
rec4.Open SQL, conn

' hole Jahr und Periode
SQL = "Select "
SQL = SQL & "jahr, periode "
```

```
    SQL = SQL & "from jahrPeriode "
    SQL = SQL & "where jahrPeriodeNr>=" & startjahrPeriode & " "
    SQL = SQL & "and jahrPeriodeNr<=" & endjahrPeriode & " "
    Set rec5 = New ADODB.Recordset
    rec5.Open SQL, conn

    ’ schreiben in excel tabelle
    i = 1
    Worksheets("Unternehmen").Cells(i, 1) = "Jahr"
    Worksheets("Unternehmen").Cells(i, 2) = "Periode"
    Worksheets("Unternehmen").Cells(i, 3) = "Unternehmen"
    Worksheets("Unternehmen").Cells(i, 4) = "Branche"
    Worksheets("Unternehmen").Cells(i, 5) = "Solvent"
    Worksheets("Unternehmen").Cells(i, 6) = "Insolvent"
    i = i + 1
    Do While Not rec.EOF
        Worksheets("Unternehmen").Cells(i, 1) = rec5!jahr
        Worksheets("Unternehmen").Cells(i, 2) = rec5!periode
        Worksheets("Unternehmen").Cells(i, 3) = rec!wert
        Worksheets("Unternehmen").Cells(i, 4) = rec2!wert
        Worksheets("Unternehmen").Cells(i, 5) = rec3!wert
        Worksheets("Unternehmen").Cells(i, 6) = rec4!wert
        ’Debug.Print rs!Rechtskreis
        rec.MoveNext
        rec2.MoveNext
        rec3.MoveNext
        rec4.MoveNext
        rec5.MoveNext
        i = i + 1
    Loop
    rec.Close
    rec2.Close
    rec3.Close
    rec4.Close
    rec5.Close
    conn.Close
    Worksheets("Unternehmen").ChartObjects.Delete
    Call maleChart(i - 1)
    Unload Me
End Sub
```

In unserem Programm werden zunächst die notwendigen Variablen deklariert. Danach lesen wir Benutzernamen und Paßwort aus den jeweiligen Textboxen.

```
    benutzer = benutzernameInput.Text
    passwort = passwortInput.Text
```

Mit diesen Eingaben des Benutzers wird sodann die Anmeldung an der Datenbank durchgeführt. Wir nutzen hier die bereits besprochene Funktion connectToDb. Danach wird die Prozedur initialisiere-Tabelle aufgerufen. Diese Prozedur dient dazu, die Excel-Tabelle zu leeren. Es können ja noch Daten aus vorherigen Läufen des Programms in den Zellen stehen, daher müssen diese vorher wieder auf keinen Inhalt gesetzt werden. Schauen wir uns die Implementierung der Prozedur an. Auch für diese Funktion erzeugen wir wieder ein eigenes Modul, wir nennen es hilfsfunktionen.

Realisierung 3.8 *Initialisieren der Excel-Tabelle*

```
Sub initialisiereTabelle(startzeile As Integer, _
                         endzeile As Integer, _
                         startspalte As Integer, _
                         endspalte As Integer)
    Dim i As Integer
    Dim j As Integer
    For i = startzeile To endzeile
        For j = startspalte To endspalte
            Worksheets("Unternehmen").Cells(i, j) = Null
        Next j
    Next i
End Sub
```

Die Prozedur erhält also eine Startzeile, eine Endzeile, eine Startspalte und eine Endspalte übergeben. In zwei ineinander geschachtelten Schleifen werden alle Zellen dieses Bereichs auf Null gesetzt. Null hat in Excel dieselbe Bedeutung wie bei den Datenbanken (vgl. Script Datenbanken). Die Zellen werden auf keinen Wert gesetzt. Durch unseren Aufruf werden also die ersten 7 Spalten der ersten 1000 Zeilen auf Null gesetzt.

Durch

```
kennzahl = KennzahlInput.Text
startjahr = StartjahrInput.Text
startperiode = StartperiodeInput.Text
endjahr = EndjahrInput.Text
endperiode = EndperiodeInput.Text
```

werden nun die restlichen Benutzereingaben aus den Comboboxen gelesen. Wie man sieht, funktioniert Lesen aus Comboboxen genau wie Lesen aus Textboxen. Im nächsten Schritt müssen wir aus den eingegebenen Start- und Endjahren, sowie Start- und Endperioden, die jahrPeriodeNr ermitteln. Diese benötigen wir später im SQL-Befehl, der die Werte für die Kennzahlen holt. Da das zweimal geschieht, lagern wir die Logik in eine Funktion aus. Hier ist der Code der Funktion:

Realisierung 3.9 *Ermittlung von jahrPeriodeNr aus Jahr und Periode*

```
Private Function getJahrPeriodeAusJahrUndPeriode(conn As ADODB.Connection, _
                                                 jahr As Integer, _
                                                 periode As Integer)
    Dim rec As ADODB.Recordset
    Dim SQL As String

    SQL = "Select "
    SQL = SQL & "jahrPeriodeNr "
    SQL = SQL & "from jahrPeriode "
    SQL = SQL & "where jahr=" & jahr & " "
    SQL = SQL & "and periode=" & periode & " "

    Set rec = New ADODB.Recordset
    rec.Open SQL, conn

    getJahrPeriodeAusJahrUndPeriode = rec!jahrPeriodeNr
    rec.Close
End Function
```

Diese Funktion müßten Sie bereits verstehen können. Die Funktion erhält die Verbindung zur Datenbank, das Jahr und die Periode übergeben. Auf der Variablen SQL wird dann das SQL-Kommando

zusammengestellt. Über ein Recordset wird die Abfrage an die Datenbank geschickt. Das Ergebnis wird über den Funktionsnamen zurück gegeben. Wie schon gesagt, rufen wir die Funktion zweimal auf und haben damit die Start- und EndjahrperiodenNr ermittelt:

```
    startjahrPeriode = getJahrPeriodeAusJahrUndPeriode
                              (conn, startjahr, startperiode)
    endjahrPeriode = getJahrPeriodeAusJahrUndPeriode
                                 (conn, endjahr, endperiode)
```

Als nächstes überlegen wir uns, wie wir an die Werte für die Kennzahlen kommen. Mit anderen Worten: Wie sieht das SQL-Kommando aus, das uns die Werte besorgt? Klar ist schon mal, dass ein Join über mehrere Tabellen nötig ist, weil der Name der Kennzahl (das ist das, was die Benutzer eingeben) in der Tabelle Kennzahl steht, in der Verbindungstabelle haben wir ja nur die KennzahlNr. Beispielhaft benutzen wir das Unternehmen. Das SQL-Kommando beginnt auf jeden Fall so:

```
Select
    kennzahlWert.wert
from kennzahl, kennzahlWert
where kennzahl.kennzahlNr=kennzahlWert.kennzahlNr
and kennzahl.name=Name, den der Anwender übergab
and kennzahl.bewerteterNr=Nummer des Unternehmens in der Tabelle bewerteter
```

Bleibt nur noch der Zeitbereich. Das ist aber ganz einfach: In der Tabelle jahrPeriode existiert ja das Feld jahrPeriodeNr. Und wir hatten die Tabelle so erzeugt, das eine spätere Periode einen höheren Wert in diesem Feld besitzt. Also müssen wir alle Datensätze ausgeben, deren Wert von jahrPeriode größer gleich der startjahrPeriode und kleiner gleich endjahrPeriode ist. Das bedeutet, wir müssen folgendes an den SQL-Befehl anhängen:

```
    and kennzahlWert.jahrPeriodeNr>=startjahrPeriode
    and kennzahlWert.jahrPeriodeNr<=endjahrPeriode
```

Um hinterher in der Ausgabe die richtige Reihenfolge in den Werten zu haben, sortieren wir nach jahrPeriodeNr.

```
    order by kennzahlWert.jahrPeriodeNr
```

Das SQL-Kommando ist damit fertig. So ein SQL-Kommando müssen wir für das Unternehmen, die Branche, die solventen Unternehmen und die insolventen Unternehmen erstellen. Weil in der Ausgabe in Excel auch das Jahr und die Periode auftauchen sollen, müssen wir zusätzlich noch zu jedem Datensatz die jahrPeriodeNr ermitteln. Um Schreibarbeit zu sparen, lagern wir die Erstellung des SQL-Kommandos in eine Funktion aus.

Realisierung 3.10 *Erstellung des SQL-Kommandos - Hilfefunktion*

```
Private Function macheSQLFromWhere(bewerterNr As Integer, _
                                   startjahrPeriode As Integer, _
                                   endjahrPeriode As Integer, _
                                   kennzahl As String)
    Dim SQLFromWhere As String
    SQLFromWhere = "from kennzahl, kennzahlWert "
    SQLFromWhere = SQLFromWhere & _
           "where kennzahl.kennzahlNr=kennzahlWert.kennzahlNr "
    SQLFromWhere = SQLFromWhere & _
           "and kennzahlWert.bewerteterNr= " & bewerterNr & " "
    SQLFromWhere = SQLFromWhere & _
           "and kennzahlWert.jahrPeriodeNr>='" & startjahrPeriode & "' "
```

```
    SQLFromWhere = SQLFromWhere & _
           "and kennzahlWert.jahrPeriodeNr<='" & endjahrPeriode & "' "
    SQLFromWhere = SQLFromWhere & "and kennzahl.name='" & kennzahl & "' "
    SQLFromWhere = SQLFromWhere & "order by kennzahlWert.jahrPeriodeNr"
    macheSQLFromWhere = SQLFromWhere
End Function
```

Diese Funktion stellt also den letzten Teil des SQL-Kommandos zusammen. Damit müßte der nun folgende Teil des Codes verständlich sein. Die SQL-Kommandos werden zusammen gestellt und über Recordsets zur Datenbank geschickt. Wir müssen hier für jede Abfrage ein eigenes Recordset herstellen, weil wir alle Recordsets hinterher zur Ausgabe benötigen. Abschließend müssen wir Jahr und Periode zu den jahrPeriodeNr's in unserer Ergebnismenge holen, weil dies ja das ist, was dargestellt werden soll.

```
    ' hole Jahr und Periode
    SQL = "Select "
    SQL = SQL & "jahr, periode "
    SQL = SQL & "from jahrPeriode "
    SQL = SQL & "where jahrPeriodeNr>=" & startjahrPeriode & " "
    SQL = SQL & "and jahrPeriodeNr<=" & endjahrPeriode & " "
    Set rec5 = New ADODB.Recordset
    rec5.Open SQL, conn
```

Nun sind alle Informationen vorhanden, wir können beginnen die Werte in die Excel-Tabelle zu schreiben. Wir starten mit den Überschriften der Spalten:

```
' schreiben in excel tabelle
    i = 1
    Worksheets("Unternehmen").Cells(i, 1) = "Jahr"
    Worksheets("Unternehmen").Cells(i, 2) = "Periode"
    Worksheets("Unternehmen").Cells(i, 3) = "Unternehmen"
    Worksheets("Unternehmen").Cells(i, 4) = "Branche"
    Worksheets("Unternehmen").Cells(i, 5) = "Solvent"
    Worksheets("Unternehmen").Cells(i, 6) = "Insolvent"
```

Wie in Realisierung 2.1 benutzen wir eine Schleife, um die Werte aus den Recordsets in die Tabelle zu schreiben. Die Schleife läuft so lange, wie noch Werte in den Recordsets sind. Für eine Diskussion des Schleifenkonstrukts verweise ich auf Realisierung 2.1.

```
    i = i + 1
    Do While Not rec.EOF
        Worksheets("Unternehmen").Cells(i, 1) = rec5!jahr
        Worksheets("Unternehmen").Cells(i, 2) = rec5!periode
        Worksheets("Unternehmen").Cells(i, 3) = rec!wert
        Worksheets("Unternehmen").Cells(i, 4) = rec2!wert
        Worksheets("Unternehmen").Cells(i, 5) = rec3!wert
        Worksheets("Unternehmen").Cells(i, 6) = rec4!wert
        rec.MoveNext
        rec2.MoveNext
        rec3.MoveNext
        rec4.MoveNext
        rec5.MoveNext
        i = i + 1
    Loop
```

Wir schließen die Recordsets und die Verbindung zur Datenbank.

```
    rec.Close
    rec2.Close
    rec3.Close
    rec4.Close
    rec5.Close
    conn.Close
```

Bleibt noch die Erzeugung der Grafik. Zunächst kann es sein, dass noch Grafiken aus früheren Programmläufen noch in die Tabelle eingebunden sind. Dann wüßte man nicht, welche nun die aktuelle ist. Außerdem sieht das ziemlich unprofessionell aus. Also entfernen wir alle Charts aus der Tabelle. Dies erledigt:

```
    Worksheets("Unternehmen").ChartObjects.Delete
```

Jetzt können wir die neue Grafik malen. Hier verfahren wir wie in Kap. 1.4. Wir nehmen ein Beispiel und erzeugen unsere Grafik in Excel. Dabei lassen wir den Makrorekorder aufzeichnen, was Excel macht, um die Grafik zu erzeugen. Dann nehmen wir den von Excel erzeugten Code und kopieren ihn in die Prozedur maleChart. Auch hierfür erstellen wir ein eigenes Modul, das Modul chartfunktionen. Wie in Realisierung 1.16 müssen wir nur noch den Bereich, der dargestellt werden soll, anpassen. Das bedeutet, wir müssen der Prozedur die Anzahl der Datensätze, die in der Tabelle sind, mitteilen. Diese kennen wir aber bereits, da in der Schleife, die die Werte in die Tabelle ausgegeben hat, diese Anzahl auf der Variablen i abgespeichert wird. Wir müssen allerdings i - 1 nehmen, da im letzten Schleifendurchlauf i einmal zu viel hochgezählt wird. Wir rufen unsere Prozedur auf:

```
    Call maleChart(i - 1)
```

Zum Schluß schauen wir uns die Grafik-Funktion an:

Realisierung 3.11 *Erstellung des Charts*

```
Sub maleChart(anzahZeilen As Integer)
    Dim darstellungsBereich As String
    Dim markierungsBereich As String
    Dim collection As String

    markierungsBereich = "A1:F" & anzahZeilen
    darstellungsBereich = "C1:F" & anzahZeilen
    collection = "=Unternehmen!R2C1:R" & anzahZeilen & "C2"

    Range(markierungsBereich).Select
    Charts.Add
    ActiveChart.ChartType = xlLineMarkers
    ActiveChart.SetSourceData Source:=Sheets("Unternehmen").Range(darstellungsBereich
        :=xlColumns
    ActiveChart.SeriesCollection(1).XValues = collection
    ActiveChart.SeriesCollection(2).XValues = collection
    ActiveChart.SeriesCollection(3).XValues = collection
    ActiveChart.SeriesCollection(4).XValues = collection
    ActiveChart.Location Where:=xlLocationAsObject, Name:="Unternehmen"
    With ActiveChart
        .HasAxis(xlCategory, xlPrimary) = True
        .HasAxis(xlValue, xlPrimary) = True
    End With
    ActiveChart.Axes(xlCategory, xlPrimary).CategoryType = xlAutomatic
End Sub
```

Wie in Realisierung 1.16 habe ich nur kleinere Anpassungen des von Excel erzeugten Codes vorgenommen. Ich habe drei neue Variablen definiert und diesen den darzustellenden Bereich zugewiesen:

```
Sub maleChart(anzahZeilen As Integer)
    Dim darstellungsBereich As String
    Dim markierungsBereich As String
    Dim collection As String

    markierungsBereich = "A1:F" & anzahZeilen
    darstellungsBereich = "C1:F" & anzahZeilen
    collection = "=Unternehmen!R2C1:R" & anzahZeilen & "C2"
```

Des weiteren habe ich überall dort, wo Excel feste Zellbezüge verwendet, diese durch meine Variablen ersetzt.

Dies ist der erste Durchlauf des Geschehens. Die Werte der ausgewählten Kennzahl und das Chart werden in der Excel-Tabelle dargestellt. Damit weitere Berechnungen durchgeführt werden können, erzeugen wir nun den in Abb. 3.4 dargestellten Button. Wir geben ihm den Namen NeueBerechnungBtn und fügen das Kommando zum Laden des Formulars in seine Ereignisprozedur ein:

Realisierung 3.12 *Die Ereignisprozedur des NeueBerechnungBtn der Excel-Tabelle*

```
Private Sub NeueBerechnungBtn_Click()
    AnmeldungKennzahlen.Show
End Sub
```

Abschließend erzeugen wir die Prozedur Workbook_Open im Modul DieseArbeitsmappe:

Realisierung 3.13 *Die Autoprozedur der Excel-Tabelle*

```
Sub Workbook_Open()
    AnmeldungKennzahlen.Show
End Sub
```

Wie bereits in Kapitel 1 dargestellt, führt dies dazu, dass das Formular beim Öffnen der Exceldatei geladen wird. Abb. 3.7 zeigt zusammenfassend die Struktur unserer Anwendung.

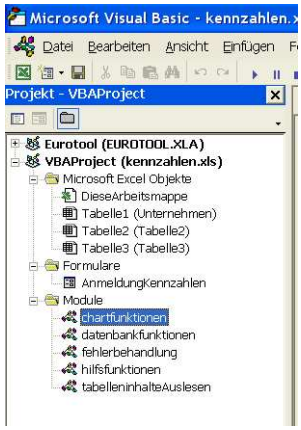

Abbildung 3.7: Die Struktur der Anwendung

3.3 Das in Excel integrierte VBA-Programm - Verbesserung durch globale Variablen

Einen Nachteil[8] hat unser Programm noch. Wenn unsere Benutzer weitere Kennzahlen aus der Datenbank lesen und plotten wollen, müssen sie sich erneut anmelden, weil wir die Anmeldeinformationen unserer Benutzer nicht speichern. In der Ereignisprozedur, die aufgerufen wird, wenn der Anmelde-Button geclickt wird, geht das auch nicht, weil Variablen, die innerhalb von Funktionen und Prozeduren deklariert werden, aus dem Hauptspeicher gelöscht werden, wenn die Prozedur oder Funktion beendet wird. Das bedeutet, in dem Moment, wenn die Ereignisprozedur beendet wird und unsere Benutzer mit Excel arbeiten, existiert keine der Variablen der Prozedur mehr. Die Variablen benutzer und passwort in Realisierung 3.7 sind demzufolge auch weg.

Wir können auch nicht mit Variablenübergabe arbeiten, denn wenn die Benutzer in der Excel-Tabelle sind und sich die Grafik und die Werte anschauen, dann läuft ja gerade gar kein Programm. Wohin also sollte man benutzer und passwort übergeben?

Es gibt allerdings eine Lösung für diese Problematik: Variablen, die außerhalb von Funktionen und Prozeduren mit dem Schlüsselwort Public deklariert werden, existieren solange, wie die Excel-Arbeitsmappe geöffnet ist. Solche Variablen sind (vgl. VBA-Script Kapitel über globale Variablen) von überall her zugreifbar. Jede Funktion oder Prozedur kann solche Variablen schreiben und lesen.

Um unsere Anwendung so zu verbessern, damit Benutzer sich nicht immer wieder neu anmelden müssen, sind einige Dinge erforderlich:

- Wir erstellen ein neues Modul mit dem Namen globaleVariablen und deklarieren dort zwei globale Variablen für Benutzernamen und Paßwort.
- Wir erstellen ein neues Formular für die weiteren Kennzahl-Grafiken. Hier lassen wir die Textboxen für Benutzernamen und Paßwort weg.
- Wir verbinden den Button in der Excel-Tabelle mit dem neuen Formular, so dass das neue Formular geladen wird, wenn der Benutzer in der Excel-Tabelle auf “Neue Berechnung starten“ clickt.
- Den Code, der die Werte der Kennzahl aus den Tabellen holt und dann in die Excel-Tabelle schreibt, lagern wir in eine Prozedur aus. So können wir ihn auch aus dem neuen Formular heraus nutzen und haben nicht zweimal gleichen Source-Code in der Anwendung.

Abb. 3.8 zeigt das neue Formular.

Dies Formular zu erstellen ist nicht wirklich schwierig. Wir wechseln in die Objekt-Ansicht des bereits bestehenden Formulars, selektieren alle Elemente, die im neuen Formular erscheinen sollen, kopieren sie, erstellen ein neues Formular und fügen unsere Elemente ein. Die Namen der Textboxen und Label können beibehalten werden. Sie sind nur in dem zum Formular gehörenden Code-Fenster bekannt, so dass keine Namenskonflikte entstehen. Den Namen des Buttons ändern wir, aber auch nur weil der Button im ersten Formular AnmeldeBtn hieß und dieser Name würde hier eventuell verwirrend wirken. Wir nennen den Button BerechnenBtn.

Wir geben unserem neuen Formular den Namen KennzahlenNeuberechnung. Damit sind wir mit dem Formular fertig.

Nun erstellen wir ein neues Modul mit dem Namen globaleVariablen. Dort fügen wir zwei Codezeilen ein:

```
Public globalBenutzer As String
Public globalPasswort As String
```

[8] Mt dem sich natürlich leben läßt.

Abbildung 3.8: Das Benutzerinterface

Diese Variablen sind jetzt von allen Code-Fenstern der VBA-Entwicklungsumgebung zugreifbar. Sie sind in allen Prozeduren und Funktionen bekannt und behalten ihre Werte, bis der Benutzer die Excelmaske schließt. Globale Variablen beginnen bei mir immer mit global. Das ist nicht vorgeschrieben, aber guter Programmierstil, denn so weiß man auf den ersten Blick, dass es sich nicht um Variablen der jeweiligen Prozedur oder Funktion handelt, in der sie gerade genutzt werden, sondern halt um globale Variablen. Was unter anderem auch bedeutet, dass eine Änderung solcher Variablen durchaus Auswirkungen auf andere Prozeduren haben kann.

Als nächstes kommen die Änderungen am Code des Formulars AnmeldungKennzahlen. Dieses Formular wird ja aufgerufen, wenn der Benutzer die Excel-Datei öffnet. An der Initialisierungsprozedur (Private Sub Userform_initialize()) ändert sich nichts, hier kennen wir den Benutzernamen und das Paßwort ja noch nicht, was bedeutet, dass sich das Programm mit dem rechtelosen Benutzer (vgl. Realisierung 3.2) an der Datenbank anmeldet und dann die Comboboxen aufbaut. Auf eine erneute Darstellung der Prozedur verzichte ich daher.

Clickt der Benutzer auf den Anmeldebutton, wird die Prozedur Private Sub AnmeldenBtn_Click() durchgeführt. Hier ergeben sich Änderungen. Zunächst benutzen wir für Benutzernamen und Paßwort die beiden globalen Variablen globalBenutzer und globalPasswort. Damit entfallen die Variablen benutzer und passwort.

Da wir das Holen der Werte der Kennzahlen und das Schreiben selbiger in die Excel-Tabelle in eine Prozedur in einem eigenen Modul auslagern, entfällt die Deklaration der hierfür benötigeten Variablen. Übrig bleibt:

Realisierung 3.14 *Erzeugung der Ausgabe bei Nutzung globaler Variablen*

```
Private Sub AnmeldenBtn_Click()
    Dim conn As ADODB.Connection
    Dim fehlerNr As Long
    Dim fehlerText As String
    Dim kennzahl As String
    Dim startjahr As Integer
    Dim startperiode  As Integer
```

```
    Dim startjahrPeriode As Integer
    Dim endjahr As Integer
    Dim endperiode As Integer

    globalBenutzer = benutzernameInput.Text
    globalPasswort = passwortInput.Text

    If Not connectToDB(globalBenutzer, globalPasswort, conn, fehlerNr, fehlerText) Th
        MsgBox ("Benutzername und Passwort stimmen nicht überein!")
        Exit Sub
    End If

    kennzahl = KennzahlInput.Text
    startjahr = StartjahrInput.Text
    startperiode = StartperiodeInput.Text
    endjahr = EndjahrInput.Text
    endperiode = EndperiodeInput.Text

    Call ausDatenbankLesenUndInTabelleSchreiben(conn, kennzahl, startjahr, _
                                            startperiode, endjahr, endperiode)
    Unload Me
End Sub
```

An den Zeilen

```
    globalBenutzer = benutzernameInput.Text
    globalPasswort = passwortInput.Text
```

sieht man, das sich globale Variablen wie normale lokale Variablen verhalten. Der einzige Unterschied ist der Geltungsbereich der Variablen.

Der Vollständigkeit halber zeige ich nun den Code der Prozedur ausDatenbankLesenUndInTabelleSchreiben. Diese Prozedur ist in das Modul rechenfunktionen ausgelagert. Den Code müßten Sie ohne weitere Diskussion verstehen können, da er Realisierung 3.7 entspricht.

Realisierung 3.15 *Die ausgelagerte Prozedur zum Zusammenstellen und Plotten der Kennzahlen*

```
Sub ausDatenbankLesenUndInTabelleSchreiben(conn As ADODB.Connection, _
                                            kennzahl As String, _
                                            startjahr As Integer, _
                                            startperiode As Integer, _
                                            endjahr As Integer, _
                                            endperiode As Integer)
    Dim startjahrPeriode  As Integer
    Dim endjahrPeriode As Integer
    Dim SQLFromWhere As String
    Dim SQL As String
    Dim i As Integer
    Dim rec As ADODB.Recordset
    Dim rec2 As ADODB.Recordset
    Dim rec3 As ADODB.Recordset
    Dim rec4 As ADODB.Recordset
    Dim rec5 As ADODB.Recordset

    Call initialisiereTabelle(1, 1000, 1, 7)
    startjahrPeriode = getJahrPeriodeAusJahrUndPeriode(conn, startjahr, startperiode)
```

```
endjahrPeriode = getJahrPeriodeAusJahrUndPeriode(conn, endjahr, endperiode)

SQLFromWhere = macheSQLFromWhere(1, startjahrPeriode, endjahrPeriode, kennzahl)
’hole Jahr periode und Unternehmenswert
SQL = "Select "
SQL = SQL & "kennzahlWert.jahrPeriodeNr, "
SQL = SQL & "kennzahlWert.wert "
SQL = SQL & SQLFromWhere

Set rec = New ADODB.Recordset
rec.Open SQL, conn

’ hole Vergleichsdaten Branche
SQLFromWhere = macheSQLFromWhere(2, startjahrPeriode, endjahrPeriode, kennzahl)
SQL = "Select "
SQL = SQL & "kennzahlWert.wert "
SQL = SQL & SQLFromWhere
Set rec2 = New ADODB.Recordset
rec2.Open SQL, conn

’ hole Vergleichsdaten solvente Unternehmen
SQLFromWhere = macheSQLFromWhere(3, startjahrPeriode, endjahrPeriode, kennzahl)
SQL = "Select "
SQL = SQL & "kennzahlWert.wert "
SQL = SQL & SQLFromWhere
Set rec3 = New ADODB.Recordset
rec3.Open SQL, conn

’ hole Vergleichsdaten insolvente Unternehmen
SQLFromWhere = macheSQLFromWhere(4, startjahrPeriode, endjahrPeriode, kennzahl)
SQL = "Select "
SQL = SQL & "kennzahlWert.wert "
SQL = SQL & SQLFromWhere
Set rec4 = New ADODB.Recordset
rec4.Open SQL, conn

’ hole Jahr und Periode
SQL = "Select "
SQL = SQL & "jahr, periode "
SQL = SQL & "from jahrPeriode "
SQL = SQL & "where jahrPeriodeNr>=" & startjahrPeriode & " "
SQL = SQL & "and jahrPeriodeNr<=" & endjahrPeriode & " "
Set rec5 = New ADODB.Recordset
rec5.Open SQL, conn

’ schreiben in excel tabelle
i = 1
Worksheets("Unternehmen").Cells(i, 1) = "Jahr"
Worksheets("Unternehmen").Cells(i, 2) = "Periode"
Worksheets("Unternehmen").Cells(i, 3) = "Unternehmen"
Worksheets("Unternehmen").Cells(i, 4) = "Branche"
Worksheets("Unternehmen").Cells(i, 5) = "Solvent"
Worksheets("Unternehmen").Cells(i, 6) = "Insolvent"
i = i + 1
```

```
    Do While Not rec.EOF
        Worksheets("Unternehmen").Cells(i, 1) = rec5!jahr
        Worksheets("Unternehmen").Cells(i, 2) = rec5!periode
        Worksheets("Unternehmen").Cells(i, 3) = rec!wert
        Worksheets("Unternehmen").Cells(i, 4) = rec2!wert
        Worksheets("Unternehmen").Cells(i, 5) = rec3!wert
        Worksheets("Unternehmen").Cells(i, 6) = rec4!wert
        'Debug.Print rs!Rechtskreis
        rec.MoveNext
        rec2.MoveNext
        rec3.MoveNext
        rec4.MoveNext
        rec5.MoveNext
        i = i + 1
    Loop
    rec.Close
    rec2.Close
    rec3.Close
    rec4.Close
    rec5.Close
    conn.Close
    Worksheets("Unternehmen").ChartObjects.Delete
    Call maleChart(i - 1)
End Sub
```

Die Funktionen getJahrPeriodeAusJahrUndPeriode und macheSQLFromWhere habe ich ebenfalls in das Modul rechenfunktionen verlegt.

Betrachten wir nun die Initialisierungsprozedur von KennzahlenNeuberechnung. Hier könnten wir die Initialisierungsprozedur des Formulars AnmeldungKennzahlen übernehmen. Ich nehme allerdings eine kleine Änderung vor. Die Anmeldung bei der Datenbank machen wir nicht mit dem rechtelosen Nutzer, sondern mit dem von unserem Benutzer bereits eingegebenen Benutzernamen und Paßwort. Dies halten wir ja auf den globalen Variablen globalBenutzer und globalPasswort vor.

Realisierung 3.16 *Initialisierung des Formulars KennzahlenNeuberechnung.*

```
Private Sub Userform_initialize()
    Dim conn As ADODB.Connection
    Dim fehlerNr As Long
    Dim fehlerText As String
    Dim anzahlJahre As Integer
    Dim anzahlPerioden As Integer
    Dim anzahlKennzahlen As Integer
    Dim jahre() As Integer
    Dim perioden() As Integer
    Dim kennzahlen() As String
    Dim meldung As String
    Dim i As Integer

    If Not connectToDB(globalBenutzer, globalPasswort, conn, fehlerNr, fehlerText) Th
        meldung = "Fehler aus Initialisieren "
        meldung = meldung & "ODBC-Fehler"
        Call gibFehlermeldungAus(meldung, fehlerNr, fehlerText)
        Exit Sub
    End If
```

```
    Call getKennzahlen(conn, kennzahlen)
    anzahlKennzahlen = UBound(kennzahlen, 1)
    For i = 1 To anzahlKennzahlen
        KennzahlInput.AddItem (kennzahlen(i))
    Next i
    Call getJahre(conn, jahre)
    anzahlJahre = UBound(jahre, 1)
    For i = 1 To anzahlJahre
        StartjahrInput.AddItem (jahre(i))
        EndjahrInput.AddItem (jahre(i))
    Next i
    Call getPerioden(conn, perioden)
    anzahlPerioden = UBound(perioden, 1)
    For i = 1 To anzahlPerioden
        StartperiodeInput.AddItem (perioden(i))
        EndperiodeInput.AddItem (perioden(i))
    Next i
    conn.Close
End Sub
```

Globale Variablen lassen sich, wie man an den Zeilen

```
    If Not connectToDB(globalBenutzer, globalPasswort, conn, fehlerNr, fehlerText) Th
        meldung = "Fehler aus Initialisieren "
        meldung = meldung & "ODBC-Fehler"
        Call gibFehlermeldungAus(meldung, fehlerNr, fehlerText)
        Exit Sub
    End If
```

erneut sehen kann, direkt in allen unseren Prozeduren und Funktionen nutzen. Auch die an den BerechnenBtn dieses Formulars geknüpfte Ereignisprozedur unterscheidet sich nicht wesentlich von der Ereignisprozedur des Formmulars AnmeldungKennzahlen. Einziger Unterschied ist, dass Benutzername und Paßwort bereits auf globalen Variablen zur Verfügung stehen und nicht mehr aus Textboxen eingelesen werden[9].

Realisierung 3.17 *Erzeugung der Ausgabe beim Formular KennzahlenNeuberechnung*

```
Private Sub BerechnenBtn_Click()
    Dim benutzer As String
    Dim passwort As String
    Dim conn As ADODB.Connection
    Dim fehlerNr As Long
    Dim fehlerText As String
    Dim kennzahl As String
    Dim startjahr As Integer
    Dim startperiode  As Integer
    Dim startjahrPeriode As Integer
    Dim endjahr As Integer
    Dim endperiode As Integer

    If Not connectToDB(globalBenutzer, globalPasswort, conn, fehlerNr, fehlerText) Th
        MsgBox ("Benutzername und Passwort stimmen nicht überein!")
        Exit Sub
```

[9] Wäre auch schwierig, dieses Formular hat ja keine mehr.

```
    End If

    kennzahl = KennzahlInput.Text
    startjahr = StartjahrInput.Text
    startperiode = StartperiodeInput.Text
    endjahr = EndjahrInput.Text
    endperiode = EndperiodeInput.Text

    Call ausDatenbankLesenUndInTabelleSchreiben(conn, kennzahl, startjahr, _
                                                startperiode, endjahr, endperiode)
    Unload Me
End Sub
```

Zum Schluss ändern wir die Ereignisprozedur des Buttons in der Excel-Tabelle, so dass Clicks auf diesen Button unser neues Formular laden.

Realisierung 3.18 *Die geänderte Prozedur des NeueBerechnungBtn der Excel-Tabelle*

```
Private Sub NeueBerechnungBtn_Click()
    KennzahlenNeuberechnung.Show
End Sub
```

Abb. 3.9 zeigt zusammenfassend die neue Struktur unserer Anwendung.

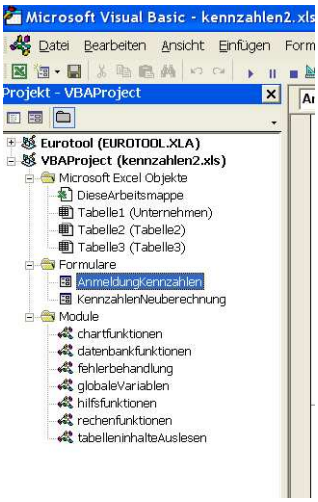

Abbildung 3.9: Die neue Struktur der Anwendung