Microsoft Excel VBA

Formel- und Funktionssammlung

Klaus-Peter Schäfer

Inhalt

Coding

Tabellen

Abbildungen

1. Module, Formulare, Klassen

Für eigene Entwicklungen unter Microsoft Excel VBA stehen Module, Formulare und Klassen zur Verfügung. Diese müssen einzeln im Projekt angelegt werden.

1.1. Module

In den Modulen werden Funktionen und Methoden hinterlegt, auf die im Verlaufe des Programms zugegriffen werden soll. Auch Funktionen, auf die der User Zugriff haben soll, können hier hinterlegt werden.
Beim Anlegen erhält das Modul einen eindeutigen Namen, der sich aus dem Wort „Modul“ und einer fortlaufenden Nummer zusammensetzt. Solange sich das Coding nur auf das aktuelle Projekt bezieht, kann dieser Name beibehalten werden. Schwieriger wird es, wenn eine Methode oder Funktion angesprochen werden soll und diese namentlich nicht mehr eindeutig sind.

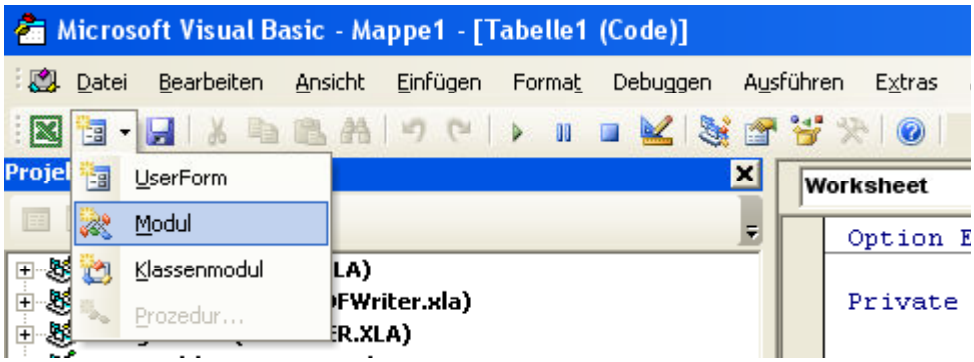


Abbildung 1: Aufruf neues Modul

Nachdem ein neues Modul durch Anklicken des entsprechenden Menüeintrages in der Dropdownliste erzeugt wurde (siehe Bild oben), steht rechts ein neues Codingfenster zur Verfügung.

1.1.1. Der Programmcode

Im Modul selbst kann nun der Code eingegeben werden. Je nach Einstellung wird ggf. der Befehl „Option Explicit“ vorgegeben. Dieser bewirkt u.a., dass alle Variablen vor ihrer Verwendung korrekt deklariert werden müssen.

1.1.2. Modulname

Module erhalten beim Anlegen einen eindeutigen Namen. Damit werden Routinen und Funktionen, die innerhalb der Module angelegt werden eine eindeutige Adressierung.
Es können mehrere Module im gleichen Projekt angelegt werden. Excel zählt dann diese einfach fortlaufend durch. Um einen sprechenden Namen für die Module zu erhalten, kann der Name eines Moduls im Eigenschaftenfenster geändert werden.

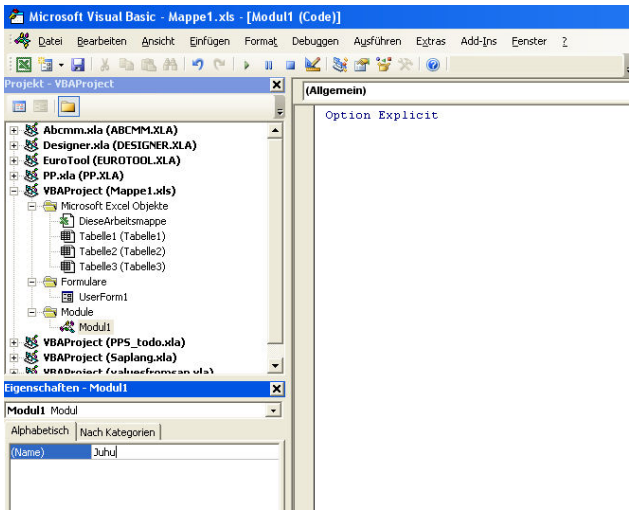


Abbildung 2: Name eines Moduls ändern

1.2. Fomulare

Mit den Formularen besteht die Möglichkeit spezielle Ein- und Ausgabefenster zu erstellen. Auch hier wird ein initialer Name vergeben, der für eine eindeutige Zuordnung besser angepasst werden sollte.
Der Aufruf ist der gleiche wie bei Modulen. Nach Aufruf erhält man in der rechten Fensterhälfte kein Codingfenster, sondern ein Formular und ein kleines, zusätzliches Werkzeugfenster.

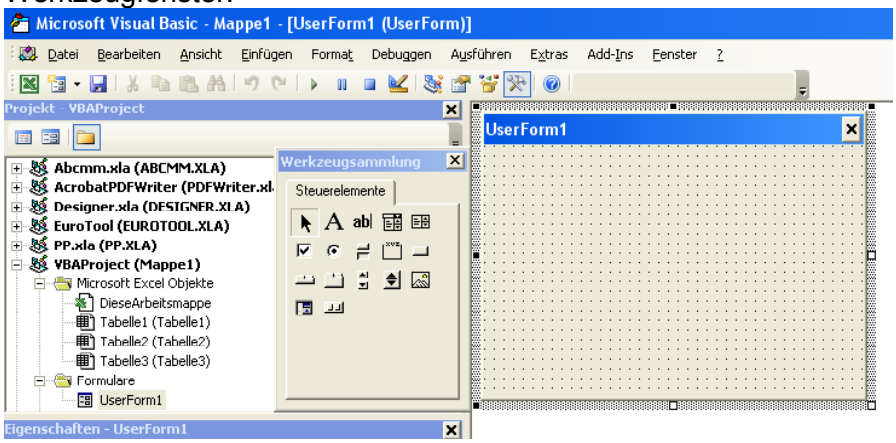


Abbildung 3: Aufruf eines Formulars

Die Objekte auf dem Werkzeugfenster können nun markiert und auf dem Formular platziert werden.
Die diverse Eigenschaften des Formulars können über das Eigenschaftenfenster eingestellt werden. Hier ein paar zur Auswahl

Eigenschaft	Bedeutung
BackColor	Hintergrundfarbe
BorderColor	Farbe des Randes
BorderStyle	Aussehen des Formulares
Caption	Text im Titel des Formulars
Font	Schriftart
ForeColor	Schriftfarbe

Tabelle 1: Auszug der Eigenschaften von Formularen

1.2.1. Steuerelemente

Auf Formularen können Steuerelemente angelegt werden. Erst hierdurch wird eine Interaktion mit dem User möglich. Die einzelnen Objekte werden durch Anklicken auf der Werkzeugsammlung aktiviert und können anschliessend durch Klicken und ziehen auf dem Formular angelegt werden.

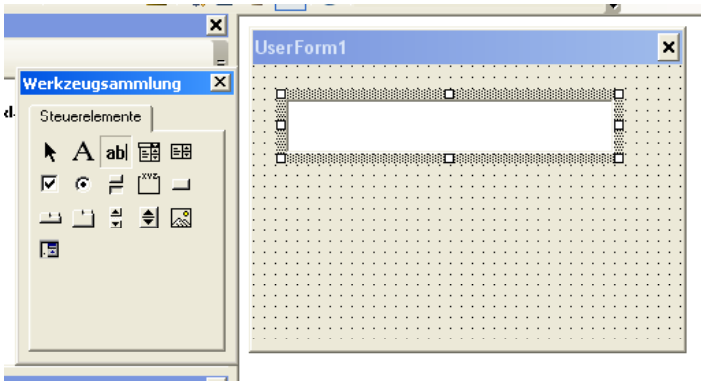


Abbildung 4: Erstellen einer Textbox auf einem Formular

Die Steuerelemente selbst verfügen über diverse Eigenschaften, die direkt im Eigenschaftenfenster eingegeben werden können.

1.2.2. Formular- Methoden und Events

Die Objekte werden im späteren Coding über Formularname.Objektname.Objekteigenschaft/-Methode angesprochen. Um das Formular anzuzeigen muss dessen Methode Show aufgerufen werden:

```
Sub schreibe_text()
   Userform1.textbox1.text=„Hallo“
   Userform1.show
End sub
```

Coding 1: Aufruf einer Userform

Über den Befehl „Unload“ lässt sich das Formular zur Laufzeit wieder entfernen.
Die wichtigsten Methoden im Überblick:

Methode	bewirkt Aktion
PrintForm	Druckt die aktuelle Form
Repaint	Zeichnet die Form neu
Show	Zeigt die Form
UndoAction	Nimmt die zuletzt durchgeführte Aktion zurück
RedoAction	Führt eine zurückgenommene Aktion wieder durch

Tabelle 2: Die wichtigsten Methoden für Formulare

Vielfach bietet es sich an, ein Coding erst dann auszuführen, nachdem ein Ereignis geschehen ist. Beispielsweise nachdem der User auf ein Formular geklickt hat. Diese Ereignisse sind als Events definiert:

```
Private Sub Userform1_click()
   Msgbox prompt:=„…klick…“
End sub
```

Coding 2: Reaktion auf ein Event

Die wichtigsten Events im Überblick:

Event	tritt ein wenn
activate	Form aktiviert wird
click	Form angeklickt wird
dblclick	Form doppelgeklickt wird
deactivate	Form deaktiviert wird
initialize	Form geladen wird
Key press	beim Drücken einer Taste
Mouse down	beim Drücken einer Maustaste
Mouse move	wenn Mauszeiger über die Form fährt
Mouse up	beim Loslassen der Maustaste
Resize	wenn Form in Grösse geändert wird

Tabelle 3: Die wichtigsten Events bei Formularen

1.3. Diese Arbeitsmappe

Als Kopfelement steht im Projektfenster „Diese Arbeitsmappe“ zur Verfügung. Hierbei handelt es sich eher um ein Modul dem Methoden und Events hinterlegt sind. Alle Aktionen beziehen sich auf das gesamte Workbook, also die gesamte Datei. Dies ist besonders für den Lade- und Entladevorgang interessant, da hier Coding hinterlegt werden kann, der dann Auswirkung auf das gesamte Dokument hat – Beispielsweise, wenn Menü’s geladen oder entladen werden sollen.

```
Private Sub Workbook_open()
   Msgbox prompt:=„…öffnen…“
End sub
```

Coding 3: Auf Workbook-Events reagieren

Aufgerufen wird das Codingfenster für „Diese Arbeitsmappe“ indem der entsprechende Eintrag im Projektfenster markiert und auf den Schalter  - „Code anzeigen“ geklickt wird. Im rechten Fensterteil erscheint ein neues, leeres Codingfenster.

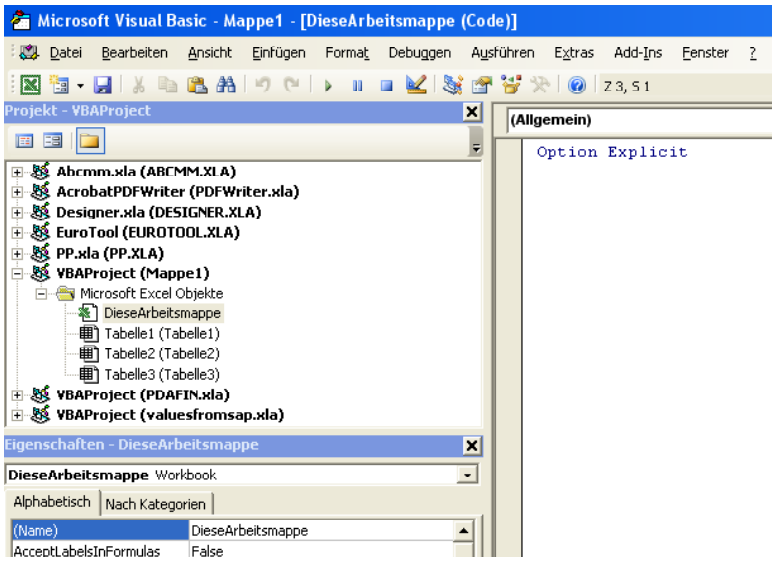


Abbildung 5: Aufruf Codingfester „Diese Arbeitsmappe“

Die wichtigsten Events im Überblick:

Event	tritt ein wenn

open	Workbook geöffnet wird
beforeclose	bevor Workbook geschlossen wird
addinstall	wenn Workbook als Addin installiert wird (*.xla)
adduninstall	wenn Workbook als Addin deinstalliert wird
beforesave	bevor gespeichert wird
Sheetchange	wenn ein Worksheet geändert wird

Tabelle 4: Die wichtigsten Events des Workbooks

1.3.1. Namen und Pfad des Workbooks ermitteln

Über folgendes Coding kann abgefragt werden, wo und unter welchem Namen die aktuelle Datei gespeichert ist:

```
Sub req_adress()
MsgBox prompt:=ThisWorkbook.FullName
End sub
```

Coding 4: Name und Pfad des Workbooks ermitteln (1)

Oder nur den Namen ermitteln:

```
Sub req_adress()
MsgBox prompt:=ThisWorkbook.FullName
End sub
```

Coding 5: Name und Pfad des Workbooks ermitteln (2)

1.3.2. Drucken und/oder Druckvorschau

Mit folgendem Coding wird eine Druckvorschau geöffnet:

```
Sub printprev()
ThisWorkbook.printpreview
End sub
```

Coding 6: Druckvorschau öffnen

Und mit diesem Coding wird gedruckt:

```
Sub print()
ThisWorkbook.print
End sub
```

Coding 7: Drucken eines Workbooks

1.4. Das Worksheet

Mit dem Erstellen eines neuen Workbooks werden drei Worksheets erstellt. Diesen kann im VBA Editor ebenfalls eigener Code hinterlegt werden. Hierzu im Projektfenster die entsprechende Tabelle (Worksheet) markieren und auf den Schalter - „Code anzeigen“ klicken. Im rechten Fensterteil erscheint ein neues, leeres Codingfenster.
Auch im Worksheet stehen spezielle Events zur Verfügung, auf die reagiert werden kann. Diese beziehen sich speziell auf das eine Tabellenblatt.

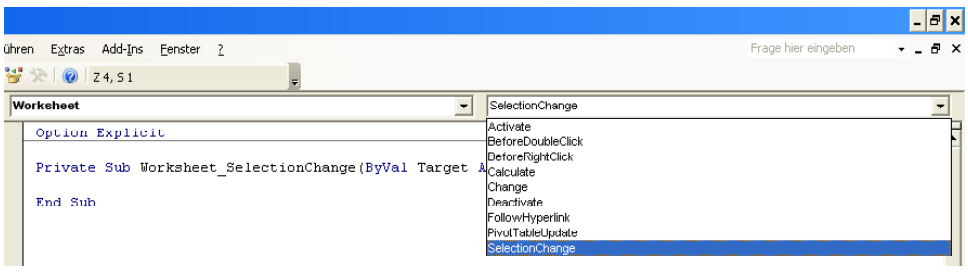


Abbildung 6: Events von „Worksheet“

Die Auswahl des Events wird wie bei „Diese Arbeitsmappe“ vorgenommen. In der linken Dropdownliste wird das Objekt Worksheet ausgewählt und in der rechten Dropdownliste die entsprechende Prozedur.

Die wichtigsten Events im Überblick:

Event	tritt ein wenn
Aktivate	Worksheet aktiviert/ausgewählt wird
Deaktivate	Worksheet deaktiviert (ein Anderes aktiviert) wird
Change	Worksheet geändert wird
Calculate	Worksheet neu berechnet wird
SelectionChange	Ein selektierter Bereiche geändert wird

Tabelle 5: Die wichtigsten Events des Worksheets

1.4.1. Globaler Name des Worksheets

Ein Worksheet kann per Programm von aussen her angesprochen werden. Entweder erfolgt dies über den Namen oder den Bezeichner. Der Name steht im Registerblatt unten und kann von jedem User beliebig geändert werden, weshalb er für die Verwendung im Coding ungeeignet ist. Der Bezeichner kann nur im Eigenschaftenfenster des VBA Editors geändert werden; dieser sollte zur Adressierung verwendet werden.

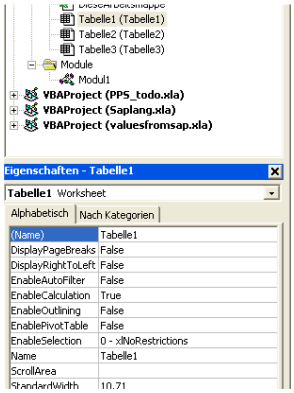


Abbildung 7: Bezeichner und Name eines Worksheets

1.4.2. Zellbereich eines Worksheets ansprechen

Um einen Zellbereich eines Worksheets aus einem Modul-Code heraus anzusprechen gibt es nun folgende Möglichkeiten:

Ansprechen per Name (sollte vermieden werden):

```
Sub read_cell()
   Sheets("Tabelle1").Cells(1, 1).Value = 1
End sub
```

Coding 8: Worksheetadressierung über Namen

Ansprechen per Bezeichner:

```
Sub read_cell()
   Tabelle1.Cells(1, 1).Value = 1
End sub
```

Coding 9: Worksheetadressierung über Bezeichner

Beim Anlegen einer neuen Exceldatei sind Name und Bezeichner für ein Worksheet immer gleich.

1.4.3. Worksheet aktivieren

Um das aktuell aktive Worksheet zu ermitteln kann folgendes Coding verwendet werden:

```
Sub is_active()
   Msgbox prompt:=activesheet.name
End sub
```

Coding 10: Name des aktiven Worksheets anzeigen

Wenn ein anderes Worksheet aktiviert werden soll:

```
Sub change_active_sheet()
   Tabelle2.activate
End sub
```

Coding 11: Anderes Worksheet aktivieren

1.4.4. Worksheet zur Laufzeit schützen

Ggf. soll ein Worksheet zur Laufzeit geschützt werden. Dies erreicht man mit folgendem Coding:

```
Sub sec_sheet()
   Tabelle1.Protect "Pass"
End sub
```

Coding 12: Worksheet schützen

Der Schutz kann mit folgendem Coding wieder entfernt werden:

```
Sub desec_sheet()
   Tabelle1.Unprotect "Pass"
End sub
```

Coding 13: Worksheet-Schutz entfernen

1.5. Klassen

In VBA können Klassen definiert werden. Diese haben allerdings in VBA gegenüber anderen Programmiersprachen einige Einschränkungen. Eine davon ist, dass sie nicht über einen normalen Constructor verfügen, sondern über die Methoden „Initialize“ und „Terminate“.

1.5.1. Anlegen einer Klasse

Eine neue Klasse wird über das Menü „Einfügen – Klassenmodul“ oder über einen Klick auf das entsprechende Icon  angelegt.

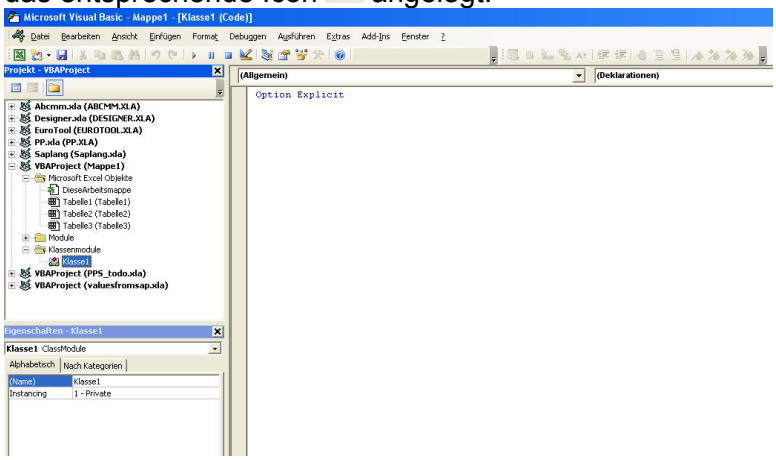


Abbildung 8: Klassenmodul

1.5.2. Code einer Klasse

Eine Klasse ist zur Laufzeit, genauso wie ein Modul, unsichtbar. Für die Methoden der Klasse gilt das gleiche wie für die der Module: Private kann von anderen Programmteilen nicht angesprochen werden, Public schon. Einzelheiten hierzu im Kapitel 8.1.

1.5.3. Die erste eigene Klasse

Nach dem Anlegen einer Klasse soll folgender Code eine Meldung ausgeben:

```
Public Sub meldung()
   MsgBox prompt:="Hallo"
End Sub
```

Coding 14: Prozedur in einer Klasse

Nun muss noch ein Modul eröffnet und die Klasse instanziert werden:

```
Sub work_with_class()
   Dim cl as new klasse1
   cl.meldung
End sub
```

Coding 15: Instanzierung einer Klasse

1.5.4. Initialize und Terminate

Wird eine Klasse mit dem SET Befehl oder durch die Deklaration mit dem NEW Befehl instanziert, so wird automatisch die Private Methode Initialize durchlaufen. Beim Beenden, durch den Befehl SET NOTHING, wird die Methode Terminate durchlaufen.

```
Dim Text

Private Sub Class_Initialize()
   Text = „Hallo“
End sub

Public Sub meldung()
   Msgbox prompt:= Text
End sub

Private Sub Class_Terminate()
   Msgbox prompt:=”Tschüss”
End Sub
```

Coding 16: Klassencode für Initialize

Aufruf durch Modul

```
Sub work_with_class()
   Dim cl as new klasse1
   cl.meldung
   set cl = nothing
End sub
```

Coding 17: Modulcode für Klassenaufruf

1.6. Der Debugger

Mit dem Debugger kann die Verarbeitung an beliebigen Punkten zur Nachverfolgung angehalten werden, oder von Anfang an per Einzelschrittverarbeitung gestartet werden.

1.6.1. Breakpoints, Einzelschritte

Der Debugger verfügt über folgende Hilfsfunktionen:

Funktion	Taste	Beschreibung
Breakpoint	F9	Setzt/Löscht einen Breakpoint, der die Verarbeitung an einer bestimmten Stelle im Code anhält.
Einzelschrittverarbeitung	F8	Startet oder setzt die Verarbeitung in Einzelschritten fort.
Ausführen (ohne Halt)	F5	Startet oder setzt die Verarbeitung, bis zum nächsten Breakpoint oder zum Ende fort.

Tabelle 6: Hilfsfunktionen des Debuggers

Breakpoints werden als roter Punkt am linken Rand des Codefensters angezeigt. Zudem wird der Code rot hinterlegt.

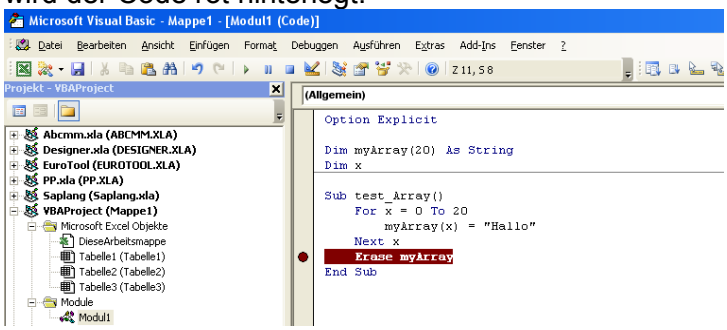


Abbildung 9: Module-Code mit Breakpoint

1.6.2. Das Überwachungsfenster

Das Überwachungsfenster stellt Variablen und Objekte sichtbar zur Verfügung. So können diese mit samt ihrem Inhalt während des Programmlaufes beobachtet werden.

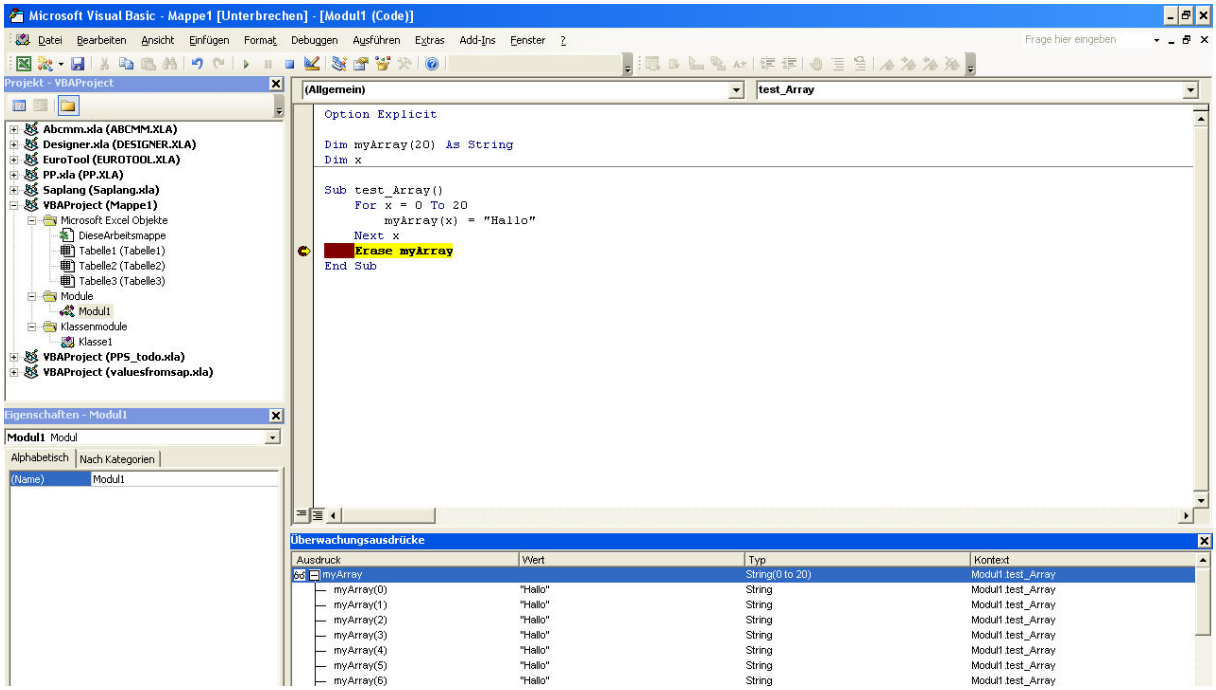


Abbildung 10: Überwachung eines Programmes zur Laufzeit

Die einzelnen Werte des Arrays werden angezeigt und können im Überwachungsfenster ggf. manuell verändert werden.

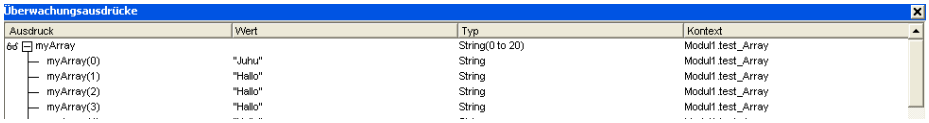


Abbildung 11: Werte einer überwachten Variablen ändern

2. Variable, Konstante, Datentypen

2.1. allgemeine Datentypen

Anbei eine kleine Übersicht, über die wohl gebräuchlichsten Datentypen:

Datentyp	Wertebereich
Boolean	True / False
Byte	0 bis 255
Integer	-32.768 bis 32.768
Long	-2.147.483.648 bis 2.147.483.647
Currency	-922.337.203.685.477,58 bis 922.337.203.685.477,58
Decimal	±79.228.162.514.264.337.593.543.950.335 ohne Nachkommastellen (mit Nachkommastellen = 28 Ziffern)
String	beliebige Zeichenkette (Buchstaben, Zahlen, Sonderzeichen)
Date	Datum / Uhrzeit
Variant	Kann ein String oder ein Double sein
Single	Gleitkommazahl mit einfacher Genauigkeit –3,402823E38 bis –1,401298E–45 1,401298E-45 bis 3,402823E38
Double	Gleitkommazahl mit doppelter Genauigkeit –1,79769313486232E308 bis –4,94065645841247E-324 4,94065645841247E–324 bis 1,79769313486232E308
Object	Referenz auf ein Objekt, eine gültige Adresse oder Nothing

Tabelle 7: Datentypen

2.1.1. Deklaration von Datentypen

Ein Datentyp für sich kann nicht verwendet werden – es muss eine Variable deklariert werden. Der Datentyp wird bei der Deklaration vererbt, d.h. die Eigenschaft des Datentyps geht auf die Variable über.

2.2. Variablen

Zur Deklaration von eingebauten oder selbst definierten Datentypen werden die Befehle DIM und PRIVATE für lokale, PUBLIC und GLOBAL für öffentliche und allgemein gültige Variablen verwendet:

```
Dim VAR1 as String
Private VAR2   as String
Public VAR3    as String
Global VAR4    as String
```

Coding 18: Deklaration von Variablen

2.2.1. abgeleitete Typänderungen

Im Programm kann es dazu kommen, dass ein Datentyp geändert werden muss. Dabei gibt es die Besonderheit, dass VBA einige Typänderungen selbständig, anhand der Logik erkennt:

```
Sub typänderung()
   Dim Alter as integer
   Alter = 10
   Alter = 10 & „ Jahre"
   Msgbox prompt:=Alter
End sub
```

Coding 19: Abgeleitete Typänderung

Aus dem Integer (einer Ganzzahl) wird durch eine logische Verknüpfung ein String.

2.2.2. Deklarierte Typänderungen

Es ist jedoch vorzuziehen, die Typänderung zu deklarieren:

```
Sub typänderung()
   Dim Alter as integer
   Alter = 10
   Alter = Cstr(10 & „ Jahre“)
   Msgbox prompt:=Alter
End sub
```

Coding 20: Deklarierte Typänderung

Folgende Funktionen stehen zur Deklarierung zur Verfügung:

Funktion	erzeugt
Cbool	Boolean
Cbyte	Byte
Ccurr	Currency
Cdate	Datum
Cdbl	Double
Cdec	Dezimal
Cint	Integer
Clng	Long
Csng	Single
Cstr	String
Cvar	Variant

Tabelle 8: Funktionen zur Typeänderung

2.3. Konstante

Die Deklaration von Konstanten erfolgt über den folgenden Befehl:

```
Const CON1 = „TEXT”
Const CON2 = 10
```

Coding 21: Deklaration von Konstanten

Die Zuweisung eines Datentypes entfällt hierbei.
Zudem stehen eine Fülle von Systeminternen Konstanten zur Verfügung:

Farben	
Konstante	**Wert**
vbBlack	Schwarz
vbBlue	Blau
vbCyan	Türkis
vbGreen	Grün
vbMagenta	Rot
vbWhite	Weiss
vbYellow	Gelb

Tabelle 9: Konstante für Farben

Funktionstasten	
Konstante	**Wert**
vbKeyF1	F1-TASTE
vbKeyF2	F2-TASTE
vbKeyF3	F3-TASTE
vbKeyF4	F4-TASTE
vbKeyF5	F5-TASTE
vbKeyF6	F6-TASTE

vbKeyF7	F7-TASTE
vbKeyF8	F8-TASTE
vbKeyF9	F9-TASTE
vbKeyF10	F10-TASTE
vbKeyF11	F11-TASTE
vbKeyF12	F12-TASTE
vbKeyF13	F13-TASTE
vbKeyF14	F14-TASTE
vbKeyF15	F15-TASTE
vbKeyF16	F16-TASTE

Tabelle 10: Konstante für Funktionstasten

Steuerzeichen	
Konstante	**Wert**
vbTab	Tabulator
vbBack	Rückschrittzeichen
vbCR	Wagenrücklauf
vbLF	Zeilenvorschub

Tabelle 11: Konstante für Steuerzeichen

2.3.1. Enumerations

Enumerations sind in Strukturen gebundene, gleichartige Konstanten. Diese eignen sich, um einen Wertebereich vorzugeben, beispielsweise bei der Parameterübergabe, innerhalb von Funktionsaufrufen.
Die Deklaration erfolgt ähnlich dem eines Arrays:

```
Private enum strcfix
   Montag = 1
   Dienstag = 2
   Mittwoch = 3
   Donnerstag = 4
   Freitag = 5
   Samstag = 6
   Sonntag = 7
End enum
```

Coding 22: Deklaration einer Enumarations

Nun kann direkt die Verwendung erfolgen. Dazu wird der Wert einfach als Datentyp der Parametervariablen einer Funktion hinterlegt:

```
Private function Testenum(Wochentag as strcfix)
   Dim wt as string
   Select case Wochentag
         Case 1
               Wt = ”Montag”
         Case 2
               Wt = ”Dienstag”
         Case 3
               Wt = „Mittwoch”
         Case 4
               Wt = „Donnerstag”
         Case 5
               Wt = „Freitag”
         Case 6
               Wt = „Samstag”
         Case 7
               Wt = „Sonntag”
   End select
   Testenum = „Der Wochentag ist „ & wt
End function

Sub call_func()
```

```
   Msgbox prompt:=testenum(3)
End sub
```

Coding 23: Code zum Test der Enumerations

Bei Eingabe der Subroutine erhält man nun nachfolgendes Bild:

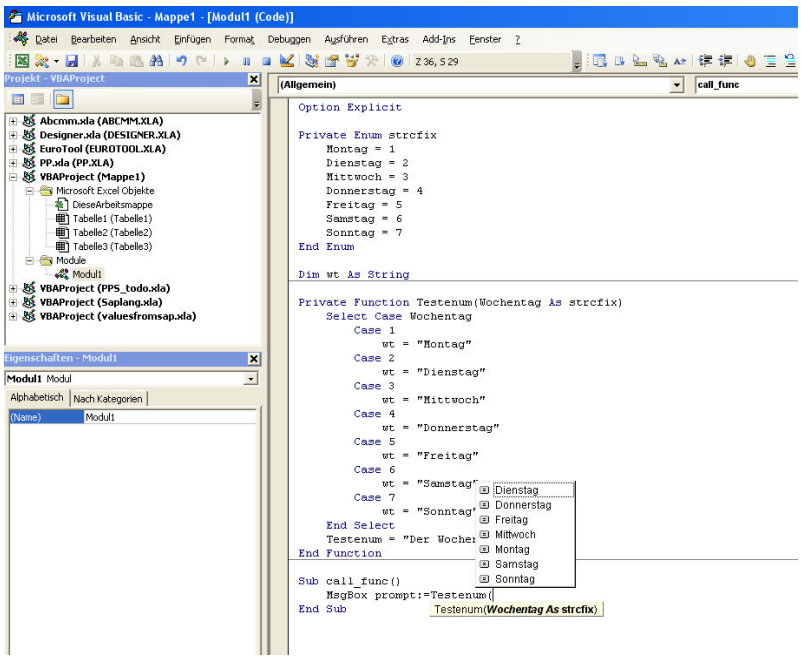


Abbildung 12: Enum Konstante beim Aufruf in einer Routine

2.4. Eigene Datentypen

2.4.1. Deklaration von UDT

Neben den internen Datentypen können noch eigene, Userdefinierte Daten Typen (UDT) angelegt werden, die wiederum nur aus den internen Datentypen bestehen können:

```
Private Type str_test
   Bezeichnung       as String
   Wertefeld         as Currency
End type

Dim VAR1 as str_test
```

Coding 24: Deklaration eines eigenen Datentypes

2.4.2. Befüllen von UDT´s

Der UDT wird über den Variablennamen und den Feldnamen angesprochen. Daten können nur auf Ebene der Feldnamen mitgegeben werden:

```
Private Type str_test
   Bezeichnung       as String
   Wertefeld         as Currency
End type

Dim VAR1 as str_test

Sub udt()
   VAR1.Bezeichnung = "Test"
   VAR1.Wertefeld = 10
End Sub
```

Coding 25: Befüllen eines UDT

2.5. Arrays

2.5.1. Allgemeines zu Arrays

Zur Definition von Datenfeldern (Arrays) werden bei Deklaration einer Variablen oder eines Datentypes die Anzahl der Datenfelder mitgegeben:

```
Sub test()
   Dim A
   A = Array(10, 20, 30)
   MsgBox prompt:=A(2)
End Sub
```

Coding 26: Ein simples Array

Üblicherweise wird ein Array oder Datentyp bereits im Vorfeld deklariert und erst später mit Daten befüllt. Dabei ist die Größe bekannt zu geben, wofür es wiederum 2 Möglichkeiten gibt (VAR3 / VAR4):

```
Private Type str_test
   VAR1        as String
   VAR2        as String
End type

Dim VAR3(1 to 100) as str_test
Dim VAR4(100)  as str_test
```

Coding 27: Deklaration eines Arrays

2.5.2. Dynamische Arrays

Es kann sein, dass die Größe des Arrays erst zur Laufzeit bekannt ist. In diesem Fall wird erst ein leeres Array erstellt und mit REDIM die Größe hinterher festgelegt:

```
Dim a()
Dim x as byte

Sub dynamisches_array()
   ReDim a(10)
   For x = 1 To 10
         a(x) = x
   Next x
End sub
```

Coding 28: Dynamisches Array

2.5.3. Daten in Array schreiben

Um einem Array Daten zu übergeben, wird einfach das entsprechende Datenfeld angesprochen und ihm ein Wert übergeben:

```
Private Type str_test
   VAR1        as String
   VAR2        as String
End type

Dim VAR3(1 to 100) as str_test
Dim x as byte

Sub test()
   For x = 1 to 100
         Var3(x).var1="Hallo"
   Next x
End sub
```

Coding 29: Werte an ein Array übergeben

2.5.4. Daten aus Array lesen

Will man lesend auf ein Array zugreifen, so wird das Coding nur umgedreht:

```
Sub test
   For x = 1 to 100
         Msgbox prompt:=Var3(x).var1
   Next x
End sub
```

Coding 30: Werte aus einem Array auslesen

2.5.5. Größe eines Array´s bestimmen

Um festzustellen, wie viele Datenfelder sich in einem Array befinden, kann der Befehl Ubound benutzt werden. Dabei wird die Obergrenze des Array´s zurückgegeben (vgl. Lbound für Untergrenze):

```
Sub test
   For x = 1 to 100
         Var3(x).var1=”Hallo”
   Next x
   Msgbox prompt:=(VAR3())
End sub
```

Coding 31: Größe eines Arrays bestimmen

2.5.6. Arrays filtern

Arrays können auch gefiltert werden. Dabei wird eine Teilmenge des Arrays gebildet, die die Suchgröße enthält:

```
Dim werte(10)
Dim fwert
Dim a As Byte

Sub fill_werte()
   For a = 0 To 4
         werte(a) = "Test"
   Next a
   For a = 5 To 9
         werte(a) = "Best"
   Next a
   fwert = Filter(werte(), "B", True)
End Sub
```

Coding 32: Arrays filtern

2.5.7. Mehrdimensionale Arrays

Manchmal wird einfach mehr benötigt, als ein schlichtes Datenfeld. Zum Beispiel dann, wenn sowohl Einzelposten als Summensatz gleichermaßen zur Verfügung stehen sollen.

```
Private Type str_m_array
   Bezeichnung as String
   Dim1              as String
   Dim2              as String
   Summe             as Currency
   EP1(1 to 100)     as String
   EW1(1 to 100)     as Currency
End type

Dim m_array(1 to 100) as str_m_array
```

Coding 33: Mehrdimensionales Array

Im obigen Coding wird ein mehrdimensionales Array definiert, das in bis zu 100 Summenzeilen jeweils bis zu 100 Einzelposten binden kann. Vorteil ist, dass mit dem Aufruf der Summenzeile auch schon die Einzelposten selektiert sind.

2.5.8. Reinitialisieren von Arrays

Arrays mit fester Grösse können mit dem Befehl ERASE wieder auf Nulllänge zurückgeset werden.

```
Dim myArray(20) As String
Dim x

Sub test_Array()
   For x= 0 to 10
         myArray(x) = "Hallo"
   Next x
   Erase myArray
End Sub
```

Coding 34: Array mit Erase auf Nulllänge setzen

3. Schleifen

3.1. Eine Schleife Programmieren

3.1.1. IF THEN

Eher eine „Halbschleife“ aber häufigsten verwendet wird die IF THEN Schleife. Diese wird wie folgt vewendet:

```
If VAR1 = „01“ then goto <Sprungmarke>
```

Coding 35: Einfache If Bedingung

Oder:

```
If VAR1 = „01” then _
   Goto <Sprungmarke1>
Else if VAR1 = „02” then _
   VAR1 = „99”
End if
```

Coding 36: Verschachtelte If Bedingung

3.1.2. DO LOOP

Beim Loop wird ein Vorgang solange ausgeführt, wie eine Bedingung gegeben ist:

```
Do while VAR1=„01“
   VAR1 = „01“
Loop
```

Coding 37: DO LOOP Schleife

3.1.3. SELECT CASE

Mit Select Case können fast alle verschachtelten „IF THEN“ Bedingungen auf einfache Weise deklariert werden:

```
Select Case VAR1
   Case „01“
         Goto <Sprungmarke1>
   Case „02“
         Goto <Sprungmarke2>
End select
```

Coding 38: Select Case Schleife

3.1.4. FOR…NEXT

Die For …next Schleife bietet die Möglichkeit eine feste Anzahl von Schleifen zu durchlaufen:

```
For x = 1 to 10
   Msgbox prompt:=x
Next x
```

Coding 39: For…next Schleife

3.1.5. For Each … Next

Hierbei werden alle Elemente eines Objektes durchlaufen – zum Beispiel alle Worksheets des aktiven Workbooks:

```
Sub test()
   Dim w As Object
   For Each w In Worksheets
         w.Cells(1, 1).Value = "Hallo"
   Next w
End Sub
```

Coding 40: For Each

3.2. Auf Ereignisse reagieren

Schleifen werden zur Prüfung oder Anreicherung von Daten verwendet. Tritt der Prüfwert ein, so wird die Schleife verlassen. Auf externe Eingaben reagiert das Programm aber nicht. Dies ändert sich mit Verwendung von

3.2.1. DoEvents

Mit DoEvents reagiert das Programm auf Benutzereingaben. Dadurch können hängende Programme verhindert werden und der Anwender kann weiterhin Eingaben tätigen.

```
Sub usereingabe()
   Dim a
   'Endlosschleife
   Do
         DoEvents
   Loop Until a <> a
End Sub
```

Coding 41: DoEvents

3.2.2. Selektives DoEventes

Ein DoEvent hält einmal pro Schleife an um nachzusehen, ob eine Eingabe erfolgt ist. Dies lässt sich performanter gestallten, indem per API gezielt abgefragt wird, ob überhaupt eine Eingabe getätigt wurde.

```
Public Declare Function GetInputState Lib "user32" () As Long

Sub usereingabe()
   Dim a
   'Endlosschleife
   Do
         'Prüfe auf Usereingabe
         If GetInputState() <> 0 Then
               DoEvents
         End If
   Loop Until a <> a
End Sub
```

Coding 42: Selektives DoEventes

4. Funktionen und Methoden

4.1. Funktionen

Funktionen sind kleine Programmteile, die von anderen Programmteilen aufgerufen werden. Dabei können Werte übergeben werden. Eine Besonderheit unter Excel ist, dass in Modulen angelegte Funktionen direkt aus einer Zelle heraus aufgerufen werden können.

```
Function add_value(Value1 as variant, value2 as variant) as variant
   Value1 = value1 + value2
   Add_value=value1
End function
```

Coding 43: Aufbau einer Funktion

Innerhalb einer Funktion stehen alle Möglichkeiten der VBA Programmierung zur Verfügung. Beispielsweise kann auf weitere Zellen oder auf Objekte (Textbox udgl.) zugegriffen werden und deren Werte in die Operation mit eingebunden werden.

```
Function add_tb(value1 as variant) as variant
   Value2 = worksheets(„Tabelle1”).textbox1.value
   Value1 = value1 + value2
   Add_tb = value1
End function
```

Coding 44: Funktion mit Zugriff auf Objekt

Für das Beispiel wird eine Textbox im Worksheet „Tabelle1“ benötigt.

4.1.1. Public / Private Function

Für die Ausführbarkeit einer Funktion in einer Excel-Worksheet-Zelle spielt es keine Rolle, ob die Funktion Public oder Private deklariert wurde. Einzig die Verfügbarkeit der Funktion von anderen Modulen oder Klassen aus, ist nicht möglich.

```
Private Function Test(text As String) As String
   Dim Ausgabe As String
   If text = "a" Then Ausgabe = "B" Else Ausgabe = "C"
   Test = Ausgabe
End Function
```

Coding 45: Verwendung einer Private Function

Eigene Funktionen finden sich automatisch im Funktionen-Auswahlfenster unter „Benutzerdefiniert“ - ausgenommen, sie wurden als Private deklariert (Taste fx drücken).

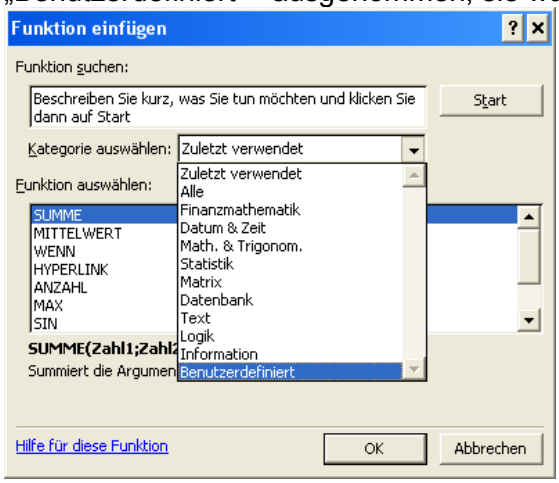


Abbildung 13: Aufruf benutzerdefinierter Funktionen

Sofern die Funktion mit allen Parametern bekannt ist, ist sie auch ausführbar:

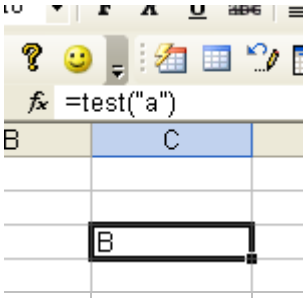


Abbildung 14: Ausfürbarkeit einer Private Funktion

4.1.2. Zusatzinformationen mitgeben

Mit einem kleinen Coding kann eine eigene Funktion recht schnell mit zusätzlichen User Informationen versorgt und im Funktions-Auswahlfenster entsprechend eingeordnet werden. Die Funktion:

```
Public Function repl(Text As String, Suchstring As String, Ersatzwert As String) As String
   Dim txtout
   txtout = Replace(Text, Suchstring, Ersatzwert)
   repl = txtout
End Function
```

Coding 46: Public Function

Die Funktion findet sich nun im Funktions-Auswahlfenster unter den benutzerdefinierten Formeln. Wird die Funktion nun in eine Zelle eingegeben und anschliessend die Formeltaste fx gedrückt, so erhält man folgenden Bildschirm (im Vergleich dazu eine Standardformel):

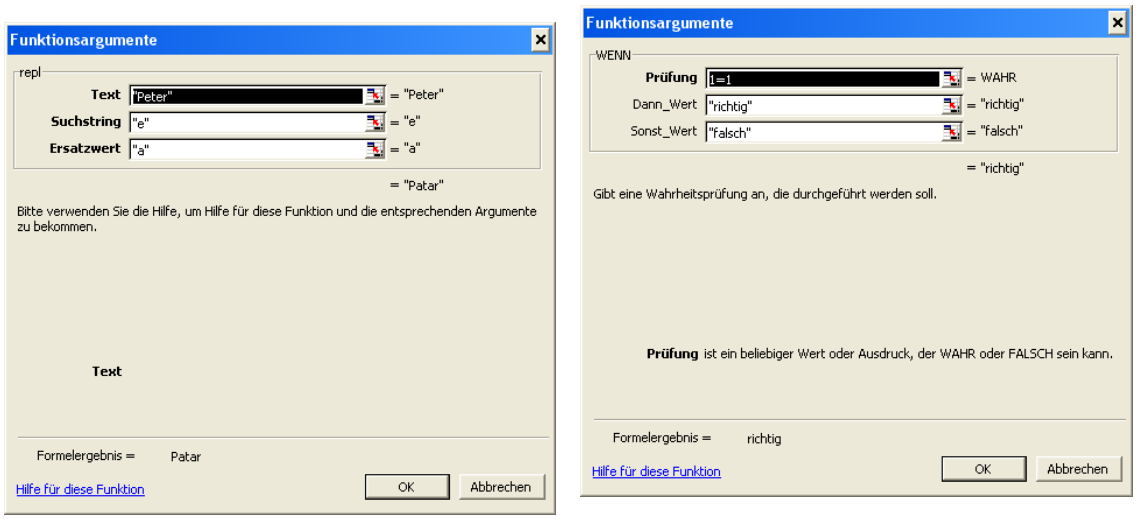


Abbildung 15: Funktionsargumente – eigene Funktion vs. Standardfunktion

Während die Standardfunktionen Eingabehilfen anzeigt, steht bei der eigenen Funktion nichts hilfreiches für den User drin. Dies durch folgende kleine Routine, die beim öffnen des Dokumentes zu durchlaufen ist, ändern:

```
Sub befor_run_function()
   Application.MacroOptions _
         macro:="repl", _
         Description:="Replace Testfunktion." & Chr$(13) & "Ersetzt alle im Text „ _
                  enthaltenen Suchstrings durch den Ersatzwert.", _
         Category:="Meine eigene Kategorie", _
         HelpFile:="c:\Hilfe.chm"
End Sub
```

Coding 47: Routine für Makrobeschreibung

Als Ergebnis wird der Beschreibungstext in der Eingabehilfe angezeigt. Zudem befindet sich die Funktion in einer eigenen Gruppe mit individueller Bezeichnung. Ausserdem wird nun beim Anklicken von „Hilfe für diese Funktion“ auch die entsprechende Hilfedatei aufgerufen.

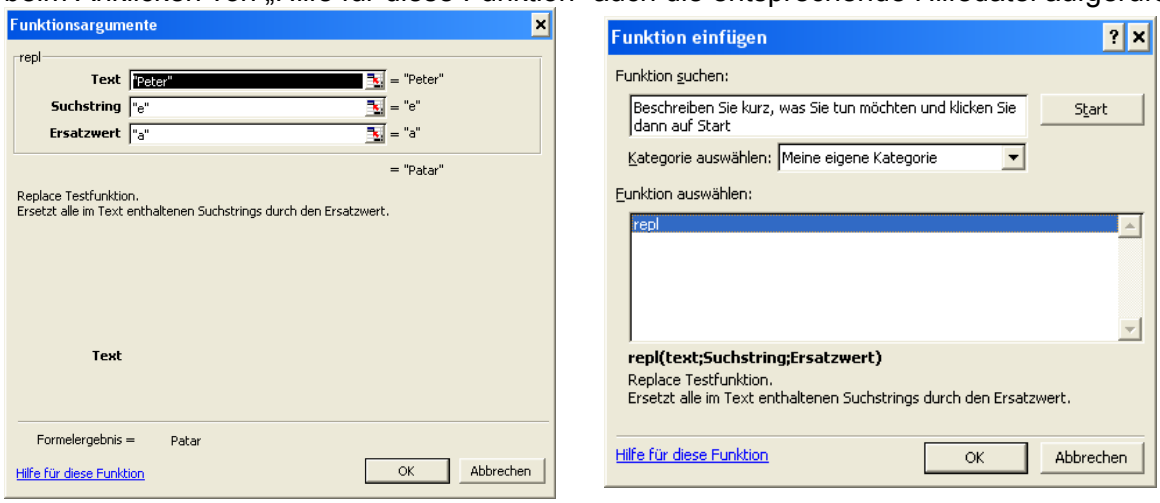


Abbildung 16: Formelbeschreibung und individuelle Funktionsgruppe

Soll die eigene Funktion in eine der anderen Formelsammlungen aufgenommen werden, so muss für den Parameter „Categorie“ lediglich eine der folgenden Zahlen mitgegeben werden:

Nr	Kategorie
1	Finanzmathematisch
2	Datum & Zeit
3	Math. & Trigonom.
4	Statistik
5	Matrix
6	Datenbank
7	Text
8	Logik
9	Information
10	Befehle
11	Benutzerorientiert
12	Makrosteuerung
13	DDE/Extern
14	Benutzerdefiniert
15	Erste benutzerdefinierte Kategorie

Tabelle 12: Formel-Kategorien

4.1.3. Neuberechnung steuern

Generell werden Formeln nur dann neu berechnet, wenn sich ein Bezugsfeld ändert, oder das Arbeitsblatt neu berechnet wird. Um eine Formel immer neu berechnen zu lassen, sobald irgendeine Zelle verändert wird, muss der Zusatz „Volatile“ mitgegeben werden:

```
Public Function calcme(wert1 as long, wert2 as long)as long
   Application.volatile
   wert1 = wert1 + wert2
   calcme = wert1
end function
```

Coding 48: Funktion immer neu berechnen lassen

4.2. Subroutinen / Methoden

Subroutinen (Methoden) lassen sich nicht per Aufruf in einer Zelle starten. Entweder sie werden durch ein Ereignis (Schalter wird gedrückt) oder eine Funktion aufgerufen.

4.2.1. Public / Private

Die Deklaration als Public oder Private hat die gleichen Konsequezen, wie bei den Funktionen. Einzig, dass Methoden nie über Excelzellen aufgerufen werden können.

4.2.2. Aufruf mit Parameterübergabe

Subroutinen können bei Aufruf Parameter mitgegeben werden.

```
Private sub msg(Byval text as string)
   Msgbox prompt:=text
End sub
```

Coding 49: Subroutine mit Parameter

Der Aufruf unterscheidet sich nicht von dem eines Function-Calls:

```
Sub programm()
   Dim Text as string
   Text = inputbox(„Bitte Text eingeben:", „Texteingabe")
   Call msg(Text)
End sub
```

Coding 50: Subroutinenaufruf mit Parameterübergabe

4.2.3. Modulübergreifende Addressierung

Soll eine Methode aus einem anderen Modul oder einer Klasse aufgerufen werden, so muss dies eindeutig addressiert sein. Dies geschiet dadurch, dass dem Methodennamen der Name des Moduls oder der Klasse vorangestellt wird.

```
Sub programm
   Dim Text as string
   Test = „inputbox(„Bitte Text eingeben:", „Texteingabe")
   Modul1.msg(Text)
End sub
```

Coding 51: Subroutinenaufruf über Adresse

5. Steuerung von Daten und Interaktionen

5.1. Dialoge

Microsoft Excel VBA verfügt über viele fest eingebaute Dialogbausteine, die sich recht einfach bedienen lassen:

5.1.1. Messagebox

Mit der Messagebox können dem Anwender nicht nur Nachrichten und Meldungen übergeben werden, man kann den Anwender reagieren lassen und diese Reaktion verarbeiten:

```
Dim reaktion As VbMsgBoxResult

Sub test()
   reaktion = MsgBox("Aussage!", vbYesNoCancel, "Titel")
   Select Case reaktion
         Case vbYes
               'User hat ja geklickt
         Case vbNo
               'User hat nein geklickt
         Case vbCancel
               'User hat abbruch geklickt
   End Select
End Sub
```

Coding 52: Interaktion auf Messagebox

5.1.2. Inputbox

Mit der Inputbox können Usereingaben abgefragt werden, wobei das Ergebnis immer ein String ist:

```
Dim eingabe As String

Sub test()
   eingabe = InputBox("Frage", "Titel", "Defaultwert")
End Sub
```

Coding 53: Userabfrage mit Inputbox

5.1.3. Datei öffnen

Für das Öffnen einer Datei steht unter Anderem die unter Application angesiedelte „Findfile“ Methode zur Verfügung:

```
Sub open_file()
   Application.findfile
End sub
```

Coding 54: Datei mit findfile öffnen

Nachteil dieser Methode ist, dass die Datei nach Anklicken von OK sofort als Exceldatei geöffnet wird. Zudem stehen keine Rückgabeparameter zur Verfügung. Deshalb:

```
Sub open_file()
   Dim datei
   datei = Application.GetOpenFilename("Excelfile (*.xls), *.xls")
   If datei <> vbFalse Then
         Workbooks.Open datei
   End If
End Sub
```

Coding 55: Datei mit Dialogfeld öffnen

5.2. Interaktionen

5.2.1. AppActivate

Eine bereits gestartete Anwendung wird durch diesen Befehl aktiviert und tritt in den Vordergrund – Beispiel für Word mit dem Dokument „Brief.doc“:

```
AppActivate(„Brief.doc - Microsoft Word“)
```

Coding 56: Fremde Application aktivieren

5.2.2. Shell

Mit dem Shell Befehl können alle im System verfügbaren Applikationen gestartet werden. Dabei wird bei erfolgreichem Start die TaskID des Programmes zurückgeliefert:

```
Shell(„C:\windows\system32\calc.exe“)
```

Coding 57: Programmaufruf mit Shell

5.2.3. Sendkeys

Der Sendkeys Befehl simuliert den Tastendruck. Dadurch wird der fremden Applikation vorgegaukelt, der Anwender würde per Tastatur Eingaben machen. Dabei ist für manche „Tasten“ ein Sonderzeichen mitzugeben:

```
' Rechner öffnen, kalkulieren und schliessen
Sub calculate()
   'Aufruf des Rechners
   Shell ("Calc.exe")
   'warten
   Application.Wait (Now + TimeValue("0:00:01"))
   'Fenster Aktivieren
   AppActivate ("Rechner")
   'Taste 3 senden
     SendKeys "3"
   'Taste Plus senden => in geschweifter Klammer,
   ' da das reine Plusszeichen eine Sonderfunktion hat
   SendKeys "{+}"
   SendKeys "5"
   Application.Wait (Now + TimeValue("0:00:01"))
   SendKeys "{=}"
   Application.Wait (Now + TimeValue("0:00:03"))
   'Sende Tastenkombination ALT+F4
   SendKeys "%{F4}"
End Sub
```

Coding 58: Fremde Application mit Sendkeys fernsteuern

Nachstehend die Tabelle mit den Codes für Sondertasten

Taste	Code
RÜCKTASTE	{BACKSPACE}, {BS} oder {BKSP}
PAUSE	{BREAK}
FESTSTELLTASTE	{CAPSLOCK}
ENTF	{DELETE} oder {DEL}
NACH-UNTEN	{DOWN}
ENDE	{END}
EINGABETASTE	{ENTER}oder ~
ESC	{ESC}
HILFE	{HELP}
POS 1	{HOME}
EINFG	{INSERT} oder {INS}
NACH-LINKS	{LEFT}
NUM-FESTSTELL	{NUMLOCK}

BILD-AB	{PGDN}
BILD-AUF	{PGUP}
DRUCK	{PRTSC}
NACH-RECHTS	{RIGHT}
ROLLEN-FESTSTELL	{SCROLLLOCK}
TAB	{TAB}
NACH-OBEN	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}
F8	{F8}
F9	{F9}
F10	{F10}
F11	{F11}
F12	{F12}
F13	{F13}
F14	{F14}
F15	{F15}
F16	{F16}
UMSCHALT	+
STRG	^
ALT	%

Tabelle 13: Tabelle mit Codes für Sondertasten

5.2.4. SendMail

Mit folgender Routine kann ein Workbook direkt aus Excel heraus per E-Mail versandt werden:

```
Sub mail_to()
   On Error GoTo ErrHandler
   Empfänger = InputBox("Mail-Empfänger:", "Exceldatei per Mail versenden")
   Betreff = InputBox("Betreff:", "Exceldatei per Mail versenden")
   Application.ActiveWorkbook.SendMail Empfänger, Betreff
   GoTo ende

ErrHandler:
   MsgBox prompt:="Der Mailempfänger konnte nicht ermittelt werden!"
ende:
End Sub
```

Coding 59: Workbook per Mail versenden

5.2.5. IIF

Wenn auf einen Zustand mit einem konkreten Wert reagiert werden soll, bietet sich die Verwendung von IIF an – dieser Befehl entspricht der Wenn-Formel:

```
Dim wert As String
Dim a As Integer

Sub test()
   a = 2
   wert = IIf(a = 1, "ja", "nein")
   MsgBox prompt:=wert
```

```
End Sub
```

Coding 60: Reaktion auf Zustand

5.3. Fehlerhandling und Errorobjekt

5.3.1. On Error goto

Beim Auftreten eines Fehlers kann gezielt reagiert werden. Beispielsweise dadurch, dass auf eine Sprungmarke verwiesen wird, an der weiter gearbeitet werden soll. Übrigends sollte die Datei C:\Datei.dat für nachfolgende Programme nicht existieren!

```
Sub testerr()
   On Error GoTo fehler
   Open "c:\Datei.dat" For Input As #1
   Close #1
   GoTo ende
fehler:
   MsgBox prompt:="Es ist ein Fehler aufgetreten"
ende:
End Sub
```

Coding 61: On Error goto

5.3.2. On Error resume next

Nun kann es auch gewollt sein, dass bei einem Fehler einfach an der nächsten Stelle im Code weiter gemacht werden soll, ohne dass eine Meldung ausgegeben wird.

```
Sub testerr()
   On Error resume next
   Open "c:\Datei.dat" For Input As #1
   Close #1
   GoTo ende
fehler:
   MsgBox prompt:="Es ist ein Fehler aufgetreten"
ende:
End Sub
```

Coding 62: On Error resume next

5.3.3. Systemfehler mit dem Error-Objekt

Mit dem Error-Objekt können die im System codierten Fehlermeldungen direkt ausgegeben werden:

```
Sub testerr()
   On Error resume next
   Open "c:\Datei.dat" For Input As #1
   Close #1
   GoTo ende
fehler:
   MsgBox prompt:=Err.Describtion
ende:
End Sub
```

Coding 63: Ausgabe der Fehlermeldung mit dem Error-Objekt

5.3.4. Eigene Fehler mit dem Error-Objekt

Nun kann es auch sein, dass eine eigene Fehlermeldung benötigt wird, beispielsweise um Usereingaben zu steuern:

```
Dim eingabe as string

Sub testerr()
   On Error GoTo errhandler
   eingabe = InputBox("Bitte geben Sie ein A ein:", "Eingabe")
   If eingabe <> "A" Then Err.raise 9999, "myprg", "Ihre Eingabe war falsch!"
   GoTo ende
errhandler:
```

```
   MsgBox prompt:=Err.Description
ende:
End Sub
```

Coding 64: Eigene Fehlermeldung mit dem Error-Objekt

5.4. Einträge in der Registry

5.4.1. Einstellungen in der Registry sichern

Um Einstellungen des Makros zu sichern – beispielsweise, die zu letzt bearbeitete Datei – kann die Registrierungsdatenbank verwendet werden. Ist der Wert noch nicht vorhanden, wird er angelegt, ansonsten wird er upgedatet.

```
Sub write_reg()
    SaveSetting "MeinProg", "Setting", "Filename", "C:\egal.txt"
End Sub
```

Coding 65: Eintrag in der Registry sichern

5.4.2. Einstellungen aus der Regedit lesen

Um nun wieder an die Daten zu gelangen, wird nun die Methode GetSettings aufgerufen.

```
Sub read_reg()
    MsgBox prompt:=GetSetting("MeinProg", "Setting", "Filename")
End Sub
```

Coding 66: Eintrag aus der Registry lesen

Zudem kann ein Defaultwert mitgegeben werden. Dies ist immer dann sinnvoll, wenn nicht gewährleistet werden kann, dass ein Wert aus der Registry geliefert wird.

```
Sub read_reg()
    MsgBox prompt:=GetSetting("MeinProg", "Setting", "Filename", „C:\test.txt”)
End Sub
```

Coding 67: Eintrag aus der Registry lesen – mit Defaultwert

5.4.3. Einstellungen aus der Registry löschen

Werden die Einstellungen nicht mehr benötigt, so können sie wie folgt gelöscht werden:

```
Sub delete_reg()
    DeleteSetting "MeinProg", "Setting"
End Sub
```

Coding 68: Löschen des Registryeintrags

5.5. Einträge in INI-Dateien

5.5.1. INI-Dateien schreiben/erzeugen

Um Daten in eine INI Datei zu schreiben, wird eine API Deklaration benötigt. Die damit zur Verfügung gestellte Funktion ermöglicht das Schreiben der INI Einträge wie folgt:

```
Private Declare Function WritePrivateProfileString Lib "kernel32" Alias _
   "WritePrivateProfileStringA"(ByVal lpApplicationname As String, ByVal lpKeyName As _
   Any, ByVal lsString As Any, ByVal lplFilename As String) As Long

Function SaveINISetting(ByVal Filename As String, ByVal Key As String, ByVal Setting As _
   String, ByVal Value As String) As Boolean

   Dim Dataout As Boolean
   On Error GoTo errhandler
   Call WritePrivateProfileString(Key, Setting, Value, Filename)
   Dataout = True
   GoTo ende
errhandler:
   Dataout = False
ende:
   SaveINISetting = Dataout
```

```
End Function


Sub INI_File_write()
   If SaveINISetting("C:\Test.ini", "Meine Daten", "Key", "3") = False Then
         MsgBox prompt:="Beim Speichern ist ein Fehler aufgetreten!"
   End If
End Sub
```

Coding 69: Daten in ein INI File schreiben

5.5.2. INI Dateien auslesen

Um nun die Daten aus einer INI Datei auszulesen, wird eine weitere API Deklaration benötigt. Diese liefert den Wert zum übergebenen Suchkey, bzw. einen Standardwert, wenn der Suchkey nicht gefunden wird.

```
Private Declare Function GetPrivateProfileString Lib "kernel32" Alias _
   "GetPrivateProfileStringA" (ByVal lpApplicationname As String, ByVal lpKeyName As _
   String, ByVal lpDefault As String, ByVal lpReturnedString As String, ByVal nSize _
   As Long, ByVal lpFileName As String) As Long

Function GetINISetting(ByVal Filename As String, ByVal Key As String, ByVal Setting As _
   String, ByVal Default As Variant) As String
   Dim Temp As String * 1024
   Call GetPrivateProfileString(Key, Setting, Default, Temp, Len(Temp), Filename)
   GetINISetting = Mid(Temp, 1, InStr(1, Temp, Chr(0)) - 1)
End Function

Sub INI_File_read()
   MsgBox prompt:=GetINISetting("C:\Test.ini", "Meine Daten", "Key", "0")
End Sub
```

Coding 70: Daten aus INI File lesen

6. Dateihandling

6.1. Dateidialoge

Um einen Dateinamen zu ermitteln, kann man sehr komfortabel auf die Application-Funktion von Microsoft Excel zurückgreifen. Dies dürfte den meisten auch bekannt sein. Dass beim Aufruf Parameter, wie z.B. ein Dateitypenfilter mitgegeben werden können, ist aber entweder nicht allen bekannt, oder es wird wenig verwendet.

6.1.1. Datei-Öffnen-Dialog

Beispiel für einen Dialag zum Öffnen einer csv Datei

```
Private Sub open_datei
   Dateiname = Application.getopenfilename(„Trennzeichen (*.csv), *.csv“)
End sub
```

Coding 71: Datei-Öffnen-Dialog

6.1.2. Datei-Speichern-Dialog

Beim Schreiben einer Datei ändert sich die Parametrisierung dahingehend, dass ein Dateiname als Vorschlagswert mitgegeben werden kann.
Bei einem User-Abbruch wird in der deutschen Version der Wert „Falsch“ sonst „False“ zurückgeliefert

Beispiel für einen Dialog zum Speichern einer csv Datei

```
Private Sub save_datei
   Dateiname = Application.getsaveasfilename(„File.csv“,“Trennzeichen (*.csv), *.csv“)
   If Dateiname = vbfalse then goto ende
   Open Dateiname for output as #1
         Print #1, „10;20;30”
   Close#1
Ende:
End sub
```

Coding 72: Datei-Speichern-Dialog: Dateiname wird durch User mitgegeben

6.1.3. Dateifilter

Der Dateifilter ist ein String, der sich aus einem beschreibenden Text und dem eigentlichen Filterkriterium, der Dateiextension zusammensetzt. Dabei wird die Extension mit einem Komma vom Text getrennt.

Beispiele:

Klassische CSV Datei	„Trennzeichen (*.csv), *.csv“
Textdatei	„Textfile (*.txt), *.txt“
Eingener Dateityp	„Meintyp (*.typ), *.typ“

Tip: Dateifilter vorab als Konstante anlegen!

Datei öffnen:

```
Const txt_file = "Textfile (*.txt), *.txt"
Const csv_file = "CSV-File (*.csv), *.csv"
Const xls_file = "Excelfile (*.xls), *.xls"
Const all_file = txt_file & "," & csv_file & "," & xls_file

Private Sub open_datei()
   Dim dateiname
   dateiname = Application.GetOpenFilename(all_file)
End Sub
```

Coding 73: Datei speichern Aufruf mit vordefinierten Dateitypen

Datei speichern:

```
Private Sub save_datei
   dim gespeichert
   gespeichert = Application.getsaveasfilename(„File.csv“,csv_file)
   If gespeichert = vbfalse then goto ende
End sub
```

Coding 74: Datei speichern Aufruf mit vordefinierten Dateitypen

6.1.4. Mehrere Dateifilter

Will man mehrere Dateifilter zur Verfügung stellen, so stellt man diese einfach in Reihe – mit Komma getrennt:

```
Const txt_file = „Textfile (*.txt), *.txt, CSV-File (*.csv), *.csv“

Private Sub save_datei
   gespeichert = Application.getsaveasfilename(„File.txt“,txt_file)
   If gespeichert = „Falsch“ then goto ende
End sub
```

Coding 75: Mehrere Dateifilter zur Verfügung stellen

6.1.5. Mehrere Dateien eines Verzeichnisses

Wenn mehrere Dateien eines Verzeichnisses ausgelesen werden sollen, hilft folgendes Coding weiter ("." und ".." sind übergeordnete Verzeichnisse und werden eleminiert):

```
Sub read_allfile()
   Dim sFile, sFiles(1000), x

   sFile = Dir("c:\winnt\*.*")
   x = 0
   Do Until sFile = ""
         If sFile <> "." And sFile <> ".." Then
               sFiles(x) = sFile
               x = x + 1
         End If
         sFile = Dir$
   Loop
End Sub
```

Coding 76: Alle Dateien eines Verzeichnisses auslesen

6.1.6. Dateien suchen

Ist der Name nicht oder nicht vollständig bekannt, so kann auch nach einer / mehreren Dateien gesucht werden:

```
Sub suche()
   Dim fs  As FileSearch
   Set fs = Application.FileSearch
   With fs
         .LookIn = "C:\"
         .Filename = "help.dat"
         If .Execute(msoSortByFileName, msoSortOrderAscending) > 0 Then
               MsgBox "There were " & .FoundFiles.Count & " file found."
               MsgBox "The filename is " & .FoundFiles.Item(1)
         Else
               MsgBox "There were no files found."
         End If
   End With
End Sub
```

Coding 77: Dateien suchen

6.2. Dateiformate

6.2.1. Dateien schreiben

Je nach Anwendung werden die Daten auf unterschiedliche Art und Weise in Dateien abgelegt. Eine Art ist das zeilenweise Wegschreiben, eine andere Art das Schreiben eines Datenwurms und letztlich die Unterscheidung, ob die Daten mit oder ohne Trennzeichen weg geschrieben werden.
Üblicherweise wird unter Microsoft Excel das CSV Format verwendet. Hierbei handelt es sich um ein zeilenweise Wegschreiben der Daten und Trennung der Datenfelder mittels eines Semikolons.
Beispiel für das Schreiben in eine CSV Datei:

```
Private Sub write_csv()
   Ausgabe = Variable1 & „;“ & Variable2
   Dateiname = Application.getsaveasfilename(,„Trennzeichen getrennt (*.csv), *.csv“)
   Open Dateiname for output as #1
         Print #1, Ausgabe
   Close #1
End sub
```

Coding 78: Schreiben einer CSV Datei

6.2.2. Dateien lesen

Beim Lesen einer CSV Datei kommt der Split – Befehl von VBA zum Einsatz:

```
Private Sub read_csv()
   Dateiname = Application.getopenfilename(„Trennzeichen getrennt (*.csv), *.csv)
   Open Dateiname for input as #1
         Input #1, Eingabe
   Close #1
   Variable1 = Split(Eingabe,”;”)(0)
   Variable2 = Split(Eingabe,”;”)(1)
End sub
```

Coding 79: Lesen einer CSV Datei

6.3. Satzarten

Wenn zu einem Datensatz mehrere Zeilen geschrieben werden müssen, so bietet sich die Verwendung von sog. Satzarten an. Dabei wird jeder Zeile eine Nummer mitgegeben, die den Aufbau des Zeileninhaltes dieser Zeile eindeutig kennzeichnet.

6.3.1. Schreiben von Dateiinhalten mit Satzart

Das Schreiben von Dateiinhalten mit Satzart ist relativ einfach. Man fasst die Datenfelder der jeweiligen Satzart in einen Datenstring zusammen und gibt diese in eine Datei aus. Damit wird das Coding im Ausgabebereich übersichtlicher.

Beispiel für das Schreiben von Dateien mit Satzart:

```
Private Sub write_sa_file()
   Ausgabe1 = „01;“ & Variable1 & „;“ & Variable2
   Ausgabe2 = „02;“ & Variable2 & „;“ & Variable1
   Dateiname = Application.getsaveasfilename(,„Trennzeichen getrennt (*.csv), *.csv“)
   Open Dateiname for output as #1
         Print #1, Ausgabe1
         Print #1, Ausgabe2
   Close #1
End sub
```

Coding 80: Schreiben von Dateien mit Satzart

Die Daten der Zieldatei haben nun folgende Struktur:

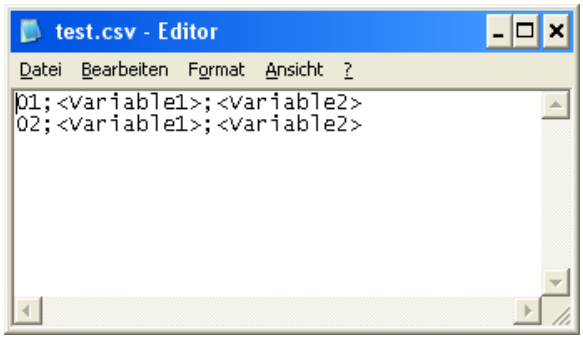


Abbildung 17: Datei mit Satzarten

6.3.2. Lesen von Dateiinhalten mit Satzart

Bei der Verarbeitung von Dateiinhalten mit Satzarten werden die Daten zeilenweise in einen String gelesen und der Inhalt des ersten Datenfeldes verarbeitet.
Je nach Satzart können die nachfolgenden Datenfelder unterschiedlich verarbeitet werden. Dabei kann sowohl ein IF THEN als auch ein SELECT CASE verwendet werden, um auf die jeweiligen Satzarten zu reagieren.

Beispiel für das Lesen einer Datei mit Satzart:

```
PrivateSub read_sa_file()
   Dateiname = Application.getopenfilename(„Trennzeichen getrennt (*.csv), *.csv“)
   Open Dateiname for input as #1
         Input #1, Eingabe
         sa = Split(Eingabe, ";")(0)
         Select Case sa
         Case "01"
               Variable1 = Split(Eingabe, ";")(1)
               Variable2 = Split(Eingabe, ";")(2)
         Case "02"
               Variable2 = Split(Eingabe, ";")(1)
               Variable1 = Split(Eingabe, ";")(2)
         End Select
   Close #1
End Sub
```

Coding 81: Lesen von Dateien mit Satzart

6.3.3. Das Dateiende

Während beim Schreibvorgang die Datei mit dem Close-Befehl einfach beendet wird, muss beim Lesen auf das Ende der Datei reagiert werden. Dies geschieht, in dem der Lesevorgang in einer DO LOOP Schleife mit der Bedingung while not eof(1) abgearbeitet wird. Die Nummer in der Klammer repräsentiert die Dateinummer.

Beispiel für das Lesen einer Datei, bis zum Dateiende:

```
Private Sub read_file_to_end()
```

```
    Dateiname = Application.getopenfilename(„Trennzeichen getrennt (*.csv), *.csv)
    Open Dateiname for input as #1
          Do while not eof(1)
                Input #1, Eingabe
                …
          Loop
    Close #1
End sub
```

Coding 82: Lesen einer Datei, bis zum Dateiende

6.3.4. Lesen von Strings mit Formatierungszeichen

Manche Textfiles tragen Formatierungszeichen zwischen den einzelnen Datenfeldern. Hier können unter Umständen die in VBA eingebauten Konstanten weiterhelfen.
Beispielsweise kann es vorkommen, dass ein Textfile geliefert wird, deren Datenfelder mit dem Zeichen □ getrennt sind. Als Trennzeichen könnte dann die System-Konstante vbBack verwendet werden.

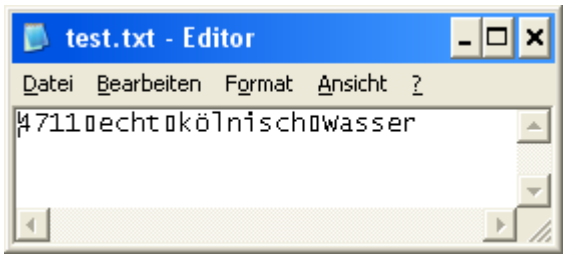


Abbildung 18: Datei mit Formatierungszeichen

Diese können nun wie folgt als Trennzeichen verwendet werden:

```
Sub test()
   Dim longstring As String
   Dim Wert As Integer
   Open "c:\test.txt" For Input As #1
         Input #1, longstring
         Wert = Split(longstring, vbTab)(0)
   Close #1
   MsgBox prompt:=Wert
End Sub
```

Coding 83: System-Konstante als Trennzeichen

Bearbeiten mehrerer Dateien
Für das Bearbeiten der Datei wurde bislang mit „#1“ die Dateinummer 1 vergeben. Will man mehrere Dateien gleichzeitig verarbeiten, so müssen die Dateien lediglich über ihre Nummer angesprochen werden.

```
Sub test()
   Open "c:\test.txt" For Input As #1
   Open „c:\test1.txt” for Output as #2
         Input #1, longstring
         Print #2, longstring
   Close #1
   Close #2
End sub
```

Coding 84: Bearbeiten mehrerer Files gleichzeitig

6.4. Lesen und Schreiben von Flatfiles

Bei der Verarbeitung von Flatfiles können die Daten nicht wie bisher gezeigt, zeilenweise gelesen oder geschrieben werden, sondern müssen über den Offset-Punkt und ihre Datensatzlänge verarbeitet werden. Dies hat zur Folge, dass die Länge eines Datensatzes immer gleich sein muss, da die Verarbeitung sonst auf Fehler läuft.

6.4.1. Öffnen eines Flatfiles

Zum Öffnen eines Flatfiles wird im Befehlssatz „open..“ immer das Schlüsselwort „random“ verwendet.

```
Private Sub lese_flat_file()
   Open Datei for Random as #1
   Close #1
End sub
```

Coding 85: Öffnen eines Flatfiles

6.4.2. Lesen und Schreiben in einem Flatfile

Der Lese- bzw. Schreibvorgang wird dann über die Behfehle „get“ bzw. „put“ durchgeführt. Mit „get“ werden die Datensätze gelesen, mit „put“ geschrieben.
Das Vorgehen soll an folgendem Beispiel demonstriert werden:
In einem Flatfile befinden sich 5 Datensätze. Diese definieren sich aus je einem 10 Zeichen langen und einem 2 Zeichen langem Feld. Die Datensatzlänge ist somit 12.

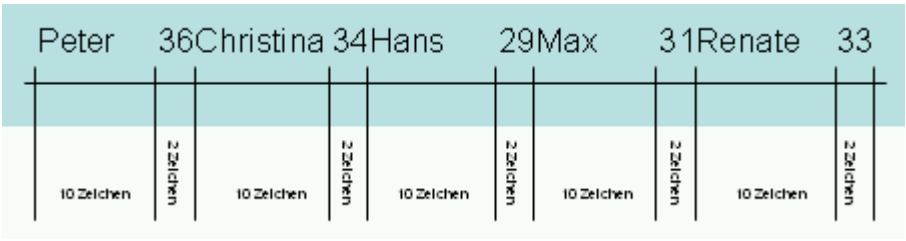


Abbildung 19: Aufbau Flatfile

Mit den Befehlen „get“ und „put“ wird nun die Datei wie folgt ausgelesen:

```
Private Sub lese_flatfile()
   Open Dateiname for random as #1
         Get #1, 2, datensatz
   Close #1
End sub
```

Coding 86: Lessen eines Flatfiles

Beim lesen wird über den Wert 2, der Satz angesprochen, der ausgelesen werden soll. Ein Datensatz hat 11 Zeichen, das System zählt also vom Beginn der Datei 11 Zeichen weiter und beginnt zu lesen.

Im Beispiel wird der 2. Datensatz eines Flatfiles ausgelesen – entsprechend wird nach dem get-Befehl die Satznummer mitgegeben. Die Länge des Datensatzes ergibt sich aus dem String, der gelesen werden soll. Dabei kann sich der String auch aus mehreren kleineren Strings zusammensetzen. Hierfür werden eigene Datentypen benötigt:

```
Private type str_ds
   Zahl        as string *3
   Wort        as string *8
End type

Dim datensatz(1 to 100) as str_ds

Private Sub lese_flatfile()
   …
   Open Dateiname for random as #1
         Get #1, 2, datensatz(2)
         …
         Put #1, 2, datensatz(2)
   Close #1
End sub
```

Coding 87: Gleichzeitiges Lesen und Schreiben in Flatfiles

Bei Flatfiles kann ein Kopfsatz mitgegeben werden. Der Aufbau eines Kopfsatzes kann von allen anderen Datensätzen abweichen. Es ist allerdings darauf zu achten, dass der Kopfsatz die gleiche Datensatzlänge hat, wie alle anderen Datensätze.

```
Private type str_ds
```

```
   Zahl        as string *3
   Wort        as string *8
End type

Private type str_head
   Wort        as string *9
   Leer        as string *2
End type

Dim datensatz(1 to 100)    as str_ds
Dim header                 as str_head

Private Sub lese_flatfile()
   …
   Open Dateiname for random as #1
         Get #1, 1, header
         Get #1, 2, datensatz(1)
   Close #1
End sub
```

Coding 88: Auslesen eines Flatfile mit Kopfsatz

6.4.3. aktuelle Lese-/Schreibposition

Bei der Verarbeitung von Flatfiles ist es wichtig die aktuelle Leseposition zu kennen. Diese lässt sich wie folgt abfragen:

```
Private Type strc_ds
   Name        As String * 10
   Alter       As String * 2
End Type

Sub test()
   Dim datei       As String
   Dim ds          As strc_ds
   datei = Application.GetOpenFilename("Datei (*.*), *.*", , "Datei suchen...", , False)
   Open datei For Random As #1
         Get #1, 1, ds
         Msgbox prompt:=Loc (1)
   Close #1
End Sub
```

Coding 89: Gibt die aktuelle Position des Datenzeigers zurück

6.4.4. Strukturierte Flatfiles

Eigentlich kann es sie gar nicht geben: „strukturierte unstrukturierte Files“. Gemeint sind damit strukturierte Files mit Feldern fester Grösse die nicht mit Trennzeichen getrennt sind. Bei dieser Art von Dateistruktur können die Daten nur zeilenweise ausgelesen und anschliessend anhand des Feldaufbaus zerlegt werden:

```
Sub test()
   Dim datei             As String
   Dim zeile             As String
   Dim myname, myalter   As String
   datei = Application.GetOpenFilename("Datei (*.*), *.*", , "Datei suchen...", , False)
   Open datei For Input As #1
         Do While Not EOF(1)
               Input #1, zeile
               myname = Left$(zeile, 10)
               myalter = Right$(zeile, 2)
         Loop
   Close #1
End Sub
```

Coding 90: Lesen von strukturierten Flatfiles

6.5. Besonderheiten beim Arbeiten mit Dateien

6.5.1. Sperren der aktuellen Datei

Um eine gleichzeitige Bearbeitung der Daten in einer Datei zu verhindern, kann die Datei über den LOCK Befehl geschützt werden. Der Close Befehl nimmt die Sperre zwar wieder weg – aber sicher ist sicher: UNLOCK

```
Private Type strc_file
   Name    As String * 10
   Alter   As String * 2
End Type

Dim data(5) As strc_file
Dim count   As Integer

Sub lese_file()
   Open "G:\test.dat" For Random As #1
         Lock 1
         Do While Not EOF(1)
               count = count + 1
               Get #1, count, data(count)
         Loop
   Close #1
   Unlock 1
End Sub
```

Coding 91: Sperren der aktuellen Datei

6.5.2. Wie groß ist die Datei

Die Größe einer Datei lässt sich wie folgt ermitteln:

```
Sub test()
   Const dtype = "Datei (*.*), *.*"
   Dim datei       As String
   datei = Application.GetOpenFilename(dtype, , "Datei suchen...", , False)
   MsgBox prompt:=FileLen(datei)
End Sub
```

Coding 92: Grösse einer Datei ermitteln

Als Rückmeldung wird eine Messagebox mit einem Wert angezeigt, der die Größe der Datei in Byte angibt.

6.5.3. Wie groß ist die geöffnete Datei

Ist die Datei mit dem OPEN Befehl geöffnet worden, so kann über den Befehl LOF deren Größe abgefragt werden:

```
Sub test()
   Dim datei       As String
   datei = Application.GetOpenFilename("Datei (*.*), *.*", , "Datei suchen...", , False)
   Open datei For Input As #1
         MsgBox prompt:=LOF(1)
   Close #1
End Sub
```

Coding 93: Auslesen der Größe einer geöffneten Datei

6.5.4. Freefile

Um mehrere Dateien gleichzeitig zu bearbeiten müssen diese durchnummeriert geöffnet werden. Um nicht mitzählen zu müssen, wird der Befehl FREEFILE verwendet, der die nächste frei Dateinummer vergibt:

```
Sub neue_datei()
   a = freefile
   Open „C:\Testfile.txt" for open as #a
         Print #1, „Hallo"
   Close #a
End sub
```

Coding 94: Nächste freie Dateinummer

6.5.5. Alle Dateien schliessen

Um alle mit open geöffneten Dateien auf einmal zu schliessen kann der Befehl Reset verwendet werden:

```
Sub close_all()
   Reste
End sub
```

Coding 95: Alle offenen Dateien schliessen

6.6. Dateiattribute und Verzeichnisse

VBA bietet eine Sammlung von Funktionen für den Umgang mit Dateien an.

6.6.1. Datei kopieren

Es ermöglicht schnelles kopieren einer Datei, wobei der Name der Zieldatei geändert werden kann:

```
Sub kopiere()
   Filecopy „C:\Testfile.txt", „D:\Testfile_kopie.txt"
End sub
```

Coding 96: Dateien mit Filecopy kopieren

6.6.2. Dateiattribute schreiben

Mit diesem Befehl können die Attribute einer Datei geändert werden:

```
Sub newattr()
   Setattr „C:\Testfile.txt", vbhidden + vbsystem
End sub
```

Coding 97: Dateiattribute ändern

Sollen mehrere Attribute vergeben werden, so werden diese mit ‚+' verkettet. Folgende Attribute stehen zur Verfügung:

Attribut	Beschreibung
VBARCHIVE	Datei wurde seit dem letzten Sichern geändert
VBSYSTEM	Systemdatei
VBHIDDEN	versteckt
VBNORMAL	normal
VBREADONLY	schreibgeschützt

Tabelle 14: Dateiattribute

6.6.3. Dateiattribute lesen

Über den Befehl Getattr können die Attribute einer Datei ausgelesen werden:

```
Sub get_attr()
   Msgbox prompt:=getattr „C:\Testfile.txt"
End sub
```

Coding 98: Attribute auslesen

6.6.4. Datei Properties

Eine Datei verfügt über eine Anzahl von Properties, wie z.B. „Author", „Titel" oder „Version". Diese lassen sich wie folgt ansprechen:

```
Dim prop            As Object

Sub test()
   Set prop = ActiveWorkbook.BuiltinDocumentProperties

   prop("Title").Value = "Testworkbook"
   prop("Subject").Value = "Test für Properties"
   prop("Author").Value = "Testmappenersteller"
   prop("Keywords").Value = "Test Testen Tests"
   prop("Comments").Value = "zum Testen"
   prop("Revision number") = "9"
   prop("Category").Value = "Testversion"
   prop("Company").Value = "Testfirma"
End Sub
```

Coding 99: Datei Properties ändern

6.6.5. Verzeichnis erstellen

Mit dem Befehl MKDIR könne Verzeichnisse erstellt werden:

```
Sub make_Dir()
   MKDIR „C:\TESTPATH"
End sub
```

Coding 100: Verzeichnis erstellen

6.6.6. Verzeichnis löschen

Mit dem Befehl RMDIR lässt sich ein Verzeichnis auch wieder entfernen:

```
Sub remove_Dir()
   RMDIR „C:\TESTPATH"
End sub
```

Coding 101: Verzeichnis löschen

6.6.7. Datei löschen

Eine Datei lässt sich mit dem Befehl Kill löschen. Was diesen Befehl besonders interessant macht ist, dass garantiert nichts von der Datei übrig bleibt – die Aktion kann definitiv nicht rückgängig gemacht werden!

```
Sub del_file()
   Kill „C:\Testfile.txt"
End sub
```

Coding 102: Datei löschen

7. Excel Application / Workbook

7.1. Application

Mit der Application werden alle Funktionen und Routinen von Excel angesprochen.

7.1.1. Der Anwendung einen eigenen Namen geben

Um den angezeigten Namen der Application zu ändern, beispielsweise um ihn gegen den der eigenen Programmierung aus zu tauschen, ist folgendes Coding notwendig:

```
Sub new_name()
   Application.Caption = „Mein Programm“
End Sub
```

Coding 103: Namen der Excelanwendung ausgeben

7.1.2. Bildschirm-Refresh verhindern

Beispielsweise kann dass heftige Bildschirmflattern beim Füllen vieler Zellen dadurch verhindert werden, dass der Bildschirm während der Befüllung nicht neu aufgebaut wird:

```
Sub ausgabe()
   Application.ScreenUpdating = False
   'Zellen ändern
   Application.ScreenUpdating = True
End Sub
```

Coding 104: Refresh während Zellenbefüllung in Excel ausschalten

7.1.3. Ausgabe einer Status - Info

Gerade wenn eine Verarbeitung länger dauert, macht es Sinn, dem User eine Rückmeldung zu geben, was gerade verarbeitet wird. Hierzu kann die Statusbar von Excel genutzt werden. Dabei wird dieser nur ein String mit der Meldung übergeben. Will man den Ursprungszustand wieder herstellen, so wird nur ein FALSE übergeben.

```
Sub meldung()
   Application.StatusBar = „Schreibe Daten in die Datei…“
End Sub
```

Coding 105: Status in der Statusbar ausgeben

7.1.4. Neuberechnen aller Zellen

Wenn sich in gespeicherten Exceldateien Worksheets mit Formeln befinden, so kann es beim erneuten Öffnen der Datei passieren, dass statt eines Formelergebnisses die Zeichenfolge „#Wert“ in der Zelle steht. Dies lässt sich mit einem Einzeiler ändern:

```
Sub refresh_all()
   Application.CalculateFull
End Sub
```

Coding 106: Refresh aller Zellen

7.1.5. Sanduhr anzeigen

Um dem Anwender zu zeigen, dass das Programm arbeitet und keine weiteren Eingaben gemacht werden sollen, kann die Sanduhr angezeigt werden:

```
Sub in_work()
   application.Cursor = xlWait
end sub
```

Coding 107: Mousecursor als Sanduhr anzeigen

Über nachfolgendes Coding wird er wieder zurückgesetzt:

```
Sub after_work()
   application.Cursor = xlDefault
end sub
```

Coding 108: Mousecursor zurücksetzen

7.2. Datenaustausch zwischen Workbook und Coding

7.2.1. Daten aus dem Tabellenblatt lesen

Im Coding muss beschrieben werden, welches Datenblatt (ggf. aus welchem Workbook) die Daten enthält. Ferner muss die Zelle unter Angabe von Zeile und Spalte adressiert werden:
Ein Worksheet kann auf zwei Arten angesprochen werden; dem internen und dem externen Namen:
Der interne Name kann nur im Coding geändert werden, der externe Name entspricht der Bezeichnung des Datenblattes auf dem Datenblattreiter. Letzterer kann jederzeit vom Anwender geändert werden, was dann zu Fehlern im Programmablauf führt.

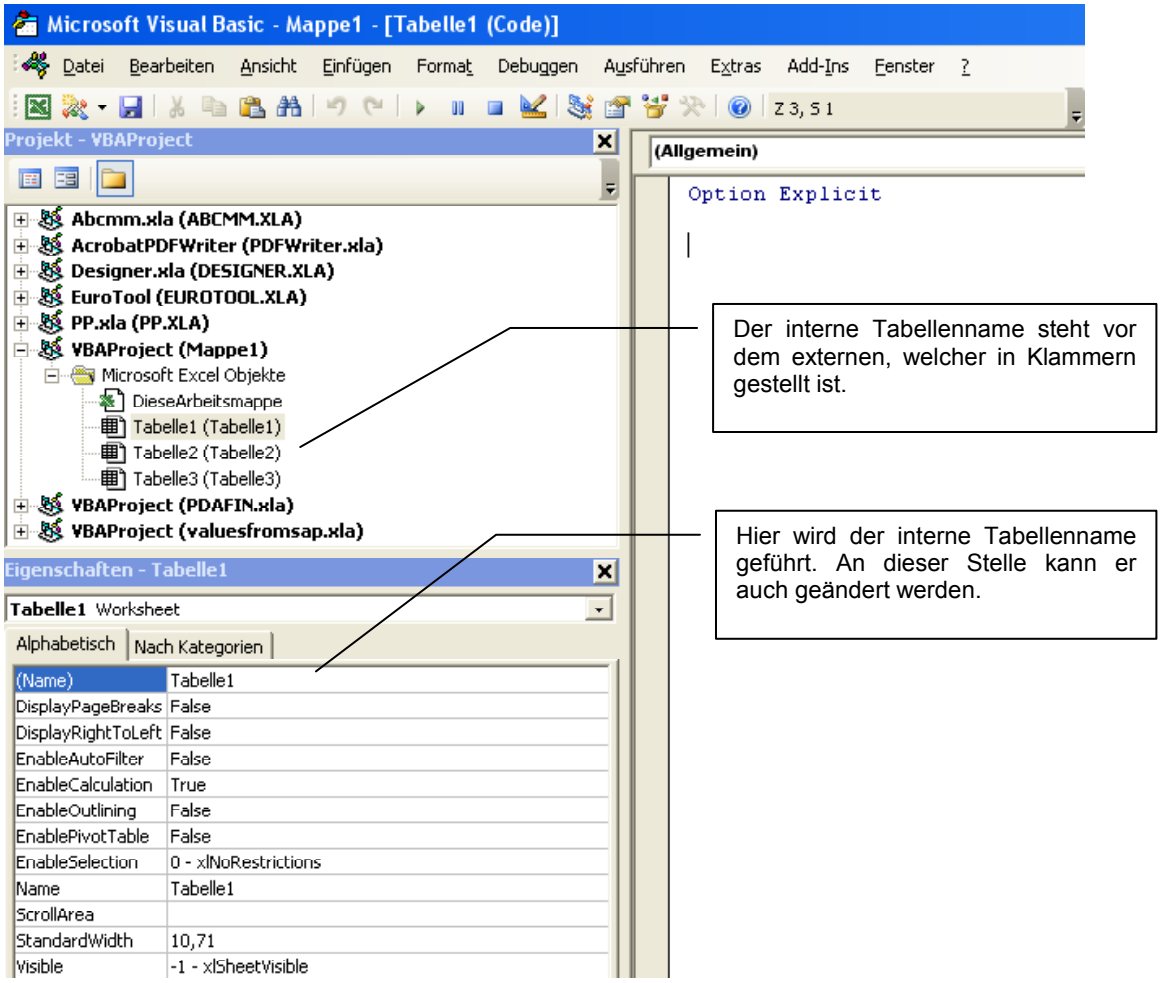


Abbildung 20: VBA Editor – interner Tabellenname

Das Ansprechen der Zellen geschieht über die Koordinaten. Beispielsweise soll die Zelle A1 angesprochen werden, so schreibt man cells(zeile,spalte):

```
Private Sub lese_ws()
   VAR1 = Worksheets(„Tabelle1“).Cells(1,1).Value
End Sub
```

Coding 109: Auslesen der Zelle A1 über externen Namen

```
Private Sub lese_ws()
   VAR1 = Worksheets(Tabelle1).Cells(1,1).Value
End Sub
```

Coding 110: Auslesen der Zelle A1 über internen Namen

7.2.2. Daten in ein Datenblatt schreiben

Beim Schreiben muss der Lesevorgang lediglich umgekehrt werden:

```
Private Sub schreibe_ws()
   Worksheets(Tabelle1).Cells(1,1).Value = VAR1
End sub
```

Coding 111: Daten in Zelle A1 schreiben

7.2.3. Datenaustausch mit Zellbereichen

Vielfach wird die Möglichkeit genutzt, Zellbereichen einen Namen zu geben. Diese Zellbereiche können dann im Coding ebenfalls mit ihrem Namen angesprochen werden.

Als Beispiel ist die Zelle A1 mit dem Namen „Bereich1“ versehen:

```
Private Sub schreibe_ws()
   Worksheets(Tabelle1).Range(„Bereich1“).Value=10
End sub
```

Coding 112: Zellen über ihren Namen ansprechen

7.2.4. Die aktive Zelle

Die Rückgabe von Werte aus selbst definierten Funktionen soll stehts in die vom User selektierte Zelle erfolgen. Hierzu übergibt man den Wert einfach an die "aktive Zelle"; diese ist, da der User ja nur eine Zelle angeklickt haben kann – global eindeutig.

```
Sub test()
   ActiveCell.Value = 10
End Sub
```

Coding 113: Aktive Zelle ansprechen

Die Position der aktiven Zellen lässt sich über die Properties "ROW" und "COLUMN" auslesen. Damit kann auch eine komplexere Ausgabe erzeugt werden:

```
Dim zeile As Integer
Dim spalte As Integer

Sub test()
   zeile = ActiveCell.Row
   spalte = ActiveCell.Column
   ActiveCell.Value = "Teil1"
   Worksheets("Tabelle1").Cells(zeile, spalte + 1).Value = "Teil2"
End Sub
```

Coding 114: Auswerten der Position der aktiven Zelle

7.2.5. Kommentare in Zelle schreiben

Eine weitere Funktionalität ist die Mitgabe von Kommentaren:

```
Sub mycomment()
   ActiveCell.addcomment „Hallo“
End sub
```

Coding 115: Kommentar einfügen

Ausgelesen wird ein Kommentar dann wie folgt:

```
Sub mycomment()
   MsgBox prompt:=ActiveCell.Comment
End Sub
```

Coding 116: Kommentar auslesen

7.2.6. Zellfarbe ändern

Um die Zellfarbe und ggf. die Hintergrundstruktur festzulegen, hilft folgendes Coding:

```
Sub change_format()
   Tabelle1.Range("A1").Interior.Color = vbRed
   With Tabelle1.Range("A2").Interior
         .Color = vbBlue
         .Pattern = xlSolid
   End With
End sub
```

Coding 117: Hintergrundfarbe einer Zelle ändern

7.2.7. Schrift ändern

Die Schriftgröße, -Farbe und –Stil lassen sich mit folgendem Coding ändern:

```
Sub change_font()
   With Tabelle1.Range("A1").Font
         .Color = vbYellow
         .Size = 14
         .Underline = True
         .Name = "Balloon"
   End With
End sub
```

Coding 118: Schrift einer Zelle ändern

7.2.8. Formel einfügen

Dass man in eine Zelle eine Formel eingeben kann ist gemein hin bekannt. Das ganze funktioniert auch über VBA:

```
Sub set_formel()
   Tabelle1.Range("A3").Formula = "=sum(A1:A2)"
End Sub
```

Coding 119: Formel in Zelle einfügen

7.3. Zellbereiche

7.3.1. Namen

Um einem Zellbereich einen eindeutigen Namen zu geben, gibt es eine entsprechende Methode für das Workbook Objekt.

```
Sub create_area()
   ActiveWorkbook.Names.Add _
         Name:="test", _
         RefersTo:="=" & ActiveSheet.Name & "!$a$1:$c$20"
End Sub
```

Coding 120: Namen für Zellbereich festlegen

Der Name kann durch nachfolgendes Coding wieder aufgehoben werden:

```
Sub delete_area()
   ActiveWorkbook.Names("test").Delete
End Sub
```

Coding 121: Namen für Zellbereich löschen

7.3.2. Zellen verbinden

Um nun Zellen zu verbinden, werden diese mit Range(…).Select angesprochen. Der Selection-Methode „MergeCells“ wird True übergeben, um die Zellen zu verbinden und False, um die Verbindung zu lösen.

```
Sub merge_range()
   ActiveSheet.Range("A1:B3").Select
   Selection.MergeCells = True
```

```
End Sub
```

Coding 122: Zellen verbinden

7.3.3. Autofilter

Über einen Autofilter kann eine Datenreihe einfach und effizient nach bestimmten Werten durchsucht werden. Das entsprechende Coding hierfür lautet:

```
Sub filtern()
   Tabelle1.Range("A1:A10") _
         .AutoFilter field:=1, Criteria1:="2"
End Sub
```

Coding 123: Autofilter

Der Autofilter kann über erneutes Aufrufen entfernt werden:

```
Sub del_filter()
   Tabelle1.Range("A1:A10").AutoFilter
End Sub
```

Coding 124: Autofilter entfernen

7.4. Verwendung von Steuerelementen

Im Worksheet selbst können Steuerelement, wie Dropdownlisten, Schalter oder Checkboxen verwendet werden. Diese werden durch Aktivierung der Steuerelemente-Toolbox angezeigt (rechter Mausklick auf die Menüleiste).

Abbildung 21: Werkzeugleiste "Steuerelemente-Toolbox"

Jedes der Objekte verfügt über Eigenschaften, die über den Schalter  bearbeitet werden können. Es könne unter anderem der Name, die Beschriftung und das Aussehen geändert werden.

7.4.1. Schalter

Auf das Anklicken eines Schalters kann, wie im Menü direkt reagiert werden:

```
Private Sub CommandButton1_click()
   Msgbox prompt:=„…klick…“
End sub
```

Coding 125: Auf Schalter Event reagieren

7.4.2. Checkbox

Die Checkbox bietet die Möglichkeit den User eine Auswahl tätigen zu lassen, ohne das eine direkte Aktion folgen muss. Selbstverständlich kann auf das Event „Click“ reagiert werden, beispielweise um die Konsistenz der Eingabe in Bezug auf andere Eingaben zu prüfen. Vorrangig geht es aber um die Statusprüfung, die über den Property „Value“ vorgenommen wird. Der Wert von Value kann auch per Coding vorgegeben werden.

```
Private Sub Checkbox1_click()
   If Checkbox1.value=true then msgbox prompt:=”…an…” else msgbox prompt:=”…aus…”
End sub
```

Coding 126: Checkbox Value

7.4.3. Combobox

Die Combobox muss vor Verwendung mit Werten befüllt werden. Anschliessend wird beim Event „Change“ der Index ausgewertet um festzustellen, welche Auswahl der User vorgenommen hat.

```
Sub fuelle_combo()
   With combobox1
         .additem „Item 1“
         .additem „Item 2”
         .additem „Item 3”
   End with
End sub
```

Coding 127: Befüllen einer Combobox

Der Einträge haben nun den Index 0, 1 bzw. 2, der beim Ändern wie folgt ausgelesen wird:

```
Private sub Combobox1_change()
   Select case combobox1.listindex
   Case 0
         Msgbox prompt:=”…eins…”
   Case 1
         Msgbox prompt:=”…zwei…”
   Case 2
         Msgbox prompt:=”…drei…”
   End select.
End sub
```

Coding 128: Auf Änderungen der Combobox reagieren

7.4.4. SpinButton

Beim Drehfeld (SpinButton) handelt es sich um einen Schalter, der selbständig rauf und runter zählen kann. Mit jedem Klick nach oben, wird raufgezählt, mit jedem Klick auf die untere Hälfte nach unten. Der Zählwert wird im Property „Value“ zurückgegeben. Der Schalter eignet sich besonders, um durch ein Datenfeld zu navigieren.

```
Private Sub Spinbutton1_change()
   Msgbox prompt:=spinbutton1.value
End sub
```

Coding 129: Mit dem SpinButton arbeiten

7.4.5. Listbox

Mit der Listbox kann dem User eine Liste mit Optionen zur Verfügung gestellt werden. Die Listbox muss genauso wie eine Dropdownliste erst befüllt werden. Die Auswahl des Users wird ebenfalls über den Listindex ausgewertet.

```
Sub fuelle_listbox()
   With listbox1
         .additem „Item1“
         .additem „Item2”
         .additem „Item3”
   End with
End sub
```

Coding 130: Befüllen einer Listbox

Eine Wahl des Users wird durch das Ereignis „Click“ angezeigt. Hier kann nun der Listindex abgefragt werden:

```
Private Sub Listbox1_Click()
   Select case Listbox1.Listindex
   Case 0
         Msgbox prompt:=”…eins…”
   Case 1
         Msgbox prompt:=”…zwei…”
   Case 2
         Msgbox prompt:=”…drei…”
   End select
End sub
```

Coding 131: Auswahl aus einer Listbox

7.4.6. Textbox

Die Verwendung einer Textbox ermöglicht die einfache Beschränkung der Eingabemöglichkeiten des Users. Zudem kann auf Ereignisse wie „Click“ oder „Chance“ reagiert werden.
Für nachfolgendes Beispiel wird eine Textbox auf einem Worksheet oder einer User benötigt:

```
Private Sub TextBox1_Change()
   Me.TextBox1.ForeColor = vbRed
End Sub

Private Sub TextBox1_LostFocus()
   Me.TextBox1.ForeColor = vbBlack
End Sub
```

Coding 132: Aussteuern einer Textbox

Bei Eingabe wird der Text in rot dargestellt, sobald das Textfeld verlassen wird, wird der Text in schwarz dargestellt.

7.4.7. Image-Box

Über die Image-Box lassen sich Bilder div. Formate laden und anzeigen.

```
Dim Bdatei

Pivate Sub Image1.Click()
   Bdatei = Application.Getopenfilename(„JPG-File (*.jpg), *.jpg, BMP-File (*.bmp), *.bmp“)
   If Bdatei <> vbfalse then
         With image1
               .PictureSizeMode = fmPictureSizeModeStretch
               .Picture = LoadPicture(Bdatei)
         End With
   End If
End sub
```

Coding 133: Laden eines Bildes in eine Imagebox

8. Objektorientierung

8.1. Klassenmodule

In Microsoft Excel VBA gibt es die Möglichkeit eigene Klassen anzulegen. Diese können in der Form „Private“ oder „PublicNotCreatable“ instanziert werden.

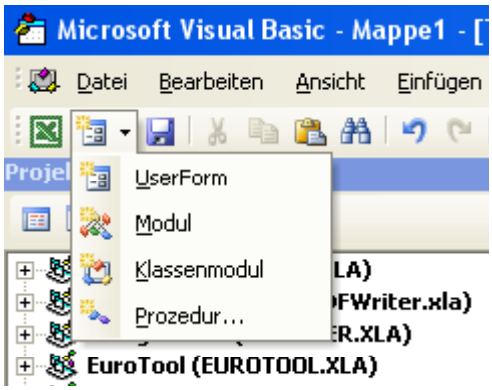


Abbildung 22: Anlegen von Klassenmodulen

8.1.1. Anlegen von Klassen

Wie bereits in 1.5.2 gezeigt, wird im Codefenster der Klasse die Routine/Methode eingegeben. Diese kann nach der Instanzierung der Klasse ausgeführt werden. Bitte hierbei beachten, dass beim späteren Aufruf der Klasse nur die öffentlichen Routinen und Funktionen zur Verfügung stehen.

```
Public Sub meldung()
   Msgbox prompt:=„Hallo“
End sub
```

Coding 134: Anlegen einer Klasse

8.1.2. Verwendung von Klassen

Bevor eine Klasse innerhalb einer Routine oder Funktion, in einem Modul verwendet werden kann, muss diese zunächst instanziert werden – dies geschieht über den Befehl „SET“:

```
Dim myklasse   as KLASSE1

Private Sub test_oo()
   Set myklasse = new KLASSE1
   myklasse.meldung
End sub
```

Coding 135: Instanzieren einer Klasse

8.1.3. Properties

Um einer Klasse vor Verarbeitung Variablen übergeben zu können steht – praktisch ausschliesslich – die Property – Methode zur Verfügung. Die Werteübergabe an das Objekt erfolgt über Property let, die Übergabe zurück mit Property get:

```
Dim VARX as String

Public Property let wVAR1(WERT as STRING)
   VARX = WERT
End property
```

Coding 136: Deklaration von Property Let

Coding für die Rückgabe des Variablenwertes

```
Public Property get rVAR1() as String
   WERT = VARX
```

```
End property
```

Coding 137: Deklaration von Property Get

Die Property Methode bietet die Möglichkeit die übergebenen Werte vor Verwendung zu prüfen und auf Fehleingaben mit einer entsprechenden Meldung zu reagieren.

Der spätere Aufruf im Coding erfolgt dann so:

```
Dim myklasse   as Klasse1

Private Sub test_klassen_routine()
   Set myklasse = new Klasse1
   Myklasse.wvarx = „Hallihallo"
   myklasse.meldung
End sub
```

Coding 138: Aufruf einer Klassen-Routine

8.1.4. Funktionen

Für Funktionen in Klassen gilt das gleiche wie für Methoden:

```
Public Function func_in_klasse(Text as string)
   Msgbox prompt:=Text
End function
```

Coding 139: Funktion in einer Klasse

Der spätere Aufruf im Coding:

```
Dim myklasse   as Klasse1

Private Sub test_klassen_funktion()
   Set myklasse = new Klasse1
   myklasse.func_in_klasse(„Hallo")
End sub
```

Coding 140: Aufruf einer Klassen-Funktion

8.1.5. Events

Hierbei handelt sich dabei um Ereignisse, die während dem Programmablauf auftreten können und auf die per Code reagiert werden kann. Da Events nur im Sub-Classing verwendet werden können, werden nun zwei Klassen benötigt:
Klasse1 öffnet eine Datei – dies löst ein Event aus:

```
Public Event IsOpen(offen As Boolean)

Public Sub doit()
   On error goto Ende
   Open "C:\cleanup.dat" For Input As #1
         RaiseEvent IsOpen(True)
   Close #1
   Ende:
End Sub
```

Coding 141: Sub-Klasse – mit Event

Klasse2 instanziert Klasse1 und führt deren Methode aus:

```
Dim WithEvents cl1    As Klasse1

Sub machauf()
   Set cl1 = New Klasse1
   cl1.doit
End Sub

Private Sub cl1_IsOpen(offen As Boolean)
   MsgBox prompt:="offen"
End Sub
```

Coding 142: Klasse mit Eventhandle

Über ein Modul wird die Klasse2 instanziert und deren Methode ausgeführt:

```
Dim cl2    As Klasse2

Sub test()
   Set cl2 = New Klasse2
   cl2.machauf
End Sub
```

Coding 143: Modul für start des Sub-Classing

8.1.6. Collection

Neben dem Sub-Classing gibt es noch die Collection. Diese ermöglicht das parallele halten mehrerer gleicher Objekte, ähnlich einem Array. Eine Collection wird als Klasse angelegt und kann Private oder Public instanziert werden, wobei Private aus Sicherheitsgründen zu bevorzugen ist.
Folgendes Beispiel eines „Kopierprogrammes“ soll eine Collection veranschaulichen:

Als Erstes eine Klasse (cls_datei), die eine Datei öffnet, Daten liest oder schreibt und die Datei danach wieder schliesst.

```
Dim Datei, inhalt, X

Public Property Get get_datei() As String
   get_datei = Datei
End Property

Public Property Let set_datei(Dateiname As String)
   Datei = Dateiname
End Property

Public Function lese() As String
   If Datei <> vbNullString Then
         X = Freefile
         Open Datei For Input As #X
               Input #X, inhalt
         Close #X
   End If
   lese = inhalt
End Function

Public Sub schreibe(ByVal text As String)
   If Datei <> vbNullString Then
         X = Freefile
         Open Datei For Output As #X
               Print #X, text
         Close #X
   End If
End Sub
```

Coding 144: Klasse zum Lesen und Schreiben von Dateien

Als Nächstes die Collection-Klasse (cls_col_datei):

```
Private myDatei As New Collection

Public Function Add(ByVal Dateiname As String) As cls_datei
   Dim newDatei As New cls_datei
   With newDatei
         .set_datei = Dateiname
         myDatei.Add newDatei
   End With
   Set Add = New cls_datei
End Function


Public Function Count() As Long
   Count = myDatei.Count
End Function

Public Sub Delete(ByVal Index As Variant)
   myDatei.Remove Index
End Sub
```

```
Public Function Item(ByVal Index As Variant) As cls_datei
   Set Item = myDatei.Item(Index)
End Function

Public Function readfile(ByVal Index As Variant) As String
   readfile = myDatei(Index).lese
End Function

Public Sub writefile(ByVal Index As Variant, ByVal text As String)
   myDatei(Index).schreibe (text)
End Sub
```

Coding 145: Collection Klasse

Und zu guter Letzt noch ein Modul, für das Rahmenprogramm:

```
Dim Datei   As cls_col_datei

Sub egal()
   Set Datei = New cls_col_datei
   With Datei
         .Add "C:\cleanup.dat"
         .Add "C:\test.dat"
   End With
   Datei.writefile 2, Datei.readfile(1)
End Sub
```

Coding 146: Rahmenprogramm für Collection Beispiel

8.1.7. Dynamischer Aufruf mit CallByName

Im Bereich der Klassen bietet sich auch der dynamische Aufruf über CallByName an. Hierbei wird nicht die Methode einer Klasse direkt angesprochen, sondern über eine Variable, die den Namen der Methode enthält. Diese sehr dynamische Form kann auch in anderen Bereichen – ausserhalb der objektorientierten Programmierung angewandt werden.

Hier das Coding für das Modul:

```
Sub msg()
  Dim sVariable As String
  Dim sText As String
  Dim cl1 As New Klasse1

  sVariable = "ShowMessage"
  sText = "Irgend ein Text"
  CallByName cl1, sVariable, VbMethod, sText
End Sub
```

Coding 147: Modul für CallByName aufruf

Und das Coding für die Klasse:

```
Public Sub ShowMessage(ByVal sText As String)
  MsgBox sText
End Sub
```

Coding 148: Methode für CallByName aufruf

8.2. Bibliotheken

Gleichartige Klassen werden in Bibliotheken organisiert. Diese Obergruppen, meist DLL´s, ermöglichen eine gezielte Verwendung von Klassen, ohne den Speicher zu belasten, da sie erst auf Anweisung geladen werden. Eine solche Anweisung ist der Verweis auf diese Bibliothek.

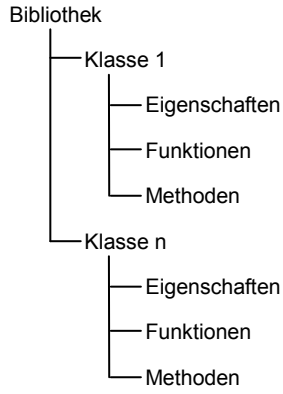


Abbildung 23: Organisation von Klassen

8.2.1. Verweise

Auf in Bibliotheken gekapselte Objekte kann von VBA aus nur zugegriffen werden, wenn auf die Bibliothek zu der das Objekt gehört, verwiesen wurde. Ein solcher Verweis auf die Bibliothek wird über das Menü „Extras“ – „Verweise“ eingerichtet.

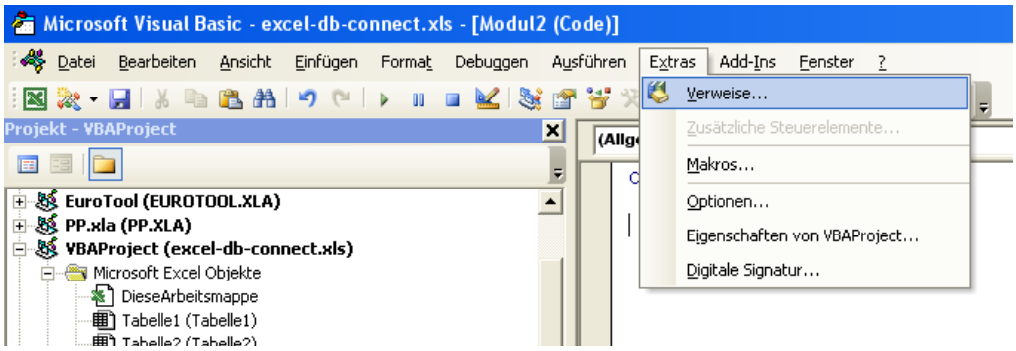


Abbildung 24: Verweis auf Bibliotheken

Im Folgenden erscheint ein Fenster, in dem die gewünschte Bibliothek ausgewählt werden kann. Durch Klicken auf OK wird die Bibliothek geladen. Es können zeitgleich auch mehrere Bibliotheken gewählt und geladen werden.

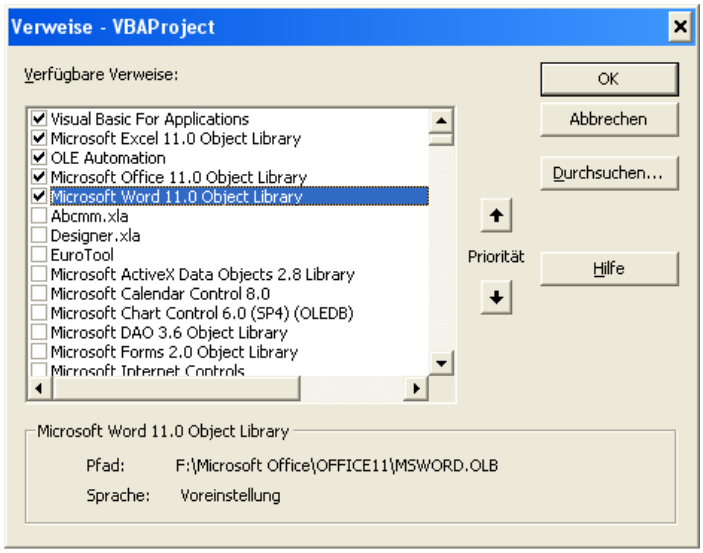


Abbildung 25: Verweis auf Bibliothek einrichten

Nach dem Bestätigen der Auswahl durch Anklicken des OK Button steht die Bibliothek zur Verfügung.

8.3. Verwendung von Bibliotheken

Ein Verweis ermöglicht es, auf bestehende Klassen oder Bibliotheken zu zugreifen. Die Deklaration im Coding erfolgt wie bei einer Variablen. Dabei kann eine Variable nur einer Klasse, niemals der gesamten Bibliothek zugewiesen werden.
Als Beispiel dient die Scripting Bibliothek. Hier muss zum Einen ein FileSystemObject, zum Anderen ein File deklariert werden. Mit der Initialisierung wird die File-Variable der FileSystemObject-Variable zugeordnet.

8.3.1. Scripting Bibliothek

Nach dem der Verweis auf die Bibliothek „Microsoft Scripting Runtime“ gelegt wurde, soll gleich die erste Datei geladen und die wichtigsten Parameter zurückgegeben werden:

```
Dim srun    As Scripting.FileSystemObject
Dim file    As Scripting.file

Sub test()
   Set srun = New Scripting.FileSystemObject
   Set file = srun.GetFile("e:\test.txt")
   With file
         Msgbox prompt:=.name
         MsgBox prompt:=.Type
         msgbox prompt:=.DateLastModified
   End With
End Sub
```

Coding 149: Laden einer Datei mit Scripting

8.3.2. MAPI Bibliothek

Um E-Mails zu versenden kann mit MAPI auf Outlook zugegriffen werden. Dazu muss ein Verweis auf die „msmapi32.ocx” gelegt werden.

```
Dim mapi    As MSMAPI.MAPISession
Dim mess    As MSMAPI.MAPIMessages

Sub mail_mit_mapi()
   Set mapi = New MSMAPI.MAPISession
   Set mess = New MSMAPI.MAPIMessages
   ' Benutzer anmelden und SessionID erzeugen
   With mapi
         ' Benutzername und Passwort (für das Mailkonto)
         '.UserName = "myusername"
         '.Password = "mypassword"
         'Outlook-Einstellung:
         .UserName = "Outlook"
         .SignOn
         ' SessionID beziehen
         mess.SessionID = .SessionID
   End With

   ' Mail senden
   With mess
         ' Neue Nachricht
         .Compose
         ' Empfänger
         .RecipAddress = "meine.mail@mail.net"
         ' Betreff
         .MsgSubject = "Test-Nachricht"
         'Datei anhängen
         .AttachmentPathName = "C:\anhang.doc"
         ' Nachricht selbst
         .MsgNoteText = "Nachrichtentext"
         ' Namen auflösen
         .ResolveName
         ' Nachricht versenden
         .Send
   End With
End Sub
```

Coding 150: E-Mail mit MAPI versenden

8.3.3. Microsoft XML Bibliothek (Arbeiten mit dem DOM Objekt)

Um eine XML Datei auszulesen oder zu schreiben, bietet sich die XML Bibliothek von Microsoft an (für das Beispiel Version 6). Für das Beispiel (Auslesen der EZB Devisenkurse) wird ein Verweis auf die Datei msxml6.dll und ein unbound ADO Tabelle (siehe)benötigt.

```
Dim rs                  As ADODB.Recordset
Dim rx                  As ADODB.Recordset
Dim doc                 As MSXML2.DOMDocument
Dim oAttrD              As IXMLDOMAttribute
```

```
Dim oAttrW                  As IXMLDOMAttribute
Dim oAttrK                  As IXMLDOMAttribute
Dim oRoot                   As IXMLDOMNodeList
Dim oElemKnoten             As IXMLDOMNode
Dim iDatum                  As Date

Const target = "http://www.ecb.europa.eu/stats/eurofxref/eurofxref-hist-90d.xml"

Sub ECB_MIT_DOM()
   'Aufbau einer Tabelle
   Set rs = New ADODB.Recordset
   With rs
         .fields.Append "Datum", adDate
         .fields.Append "Währung", adChar, 3
         .fields.Append "Kurs", adDouble
         .Open
   End With

   'XML aufrufen
   Set doc = New MSXML2.DOMDocument
   With doc
         .async = False
         Call .Load(target)
   End With

   'XML auslesen
   Set oRoot = doc.documentElement.selectNodes("Cube")
   For Each oElemKnoten In oRoot(0).childNodes
         'Datumsabfrage
         Set oAttrD = oElemKnoten.Attributes.getNamedItem("time")
         iDatum = oAttrD.nodeValue
         For Each oElemData In oElemKnoten.childNodes
               'Kursabfrage
               Set oAttrW = oElemData.Attributes.getNamedItem("currency")
               Set oAttrK = oElemData.Attributes.getNamedItem("rate")
               'Tabellenbefüllung
               With rs
                     .AddNew
                           .fields("Datum").Value = iDatum
                           .fields("Währung").Value = oAttrW.nodeValue
                           .fields("Kurs").Value = oAttrK.nodeValue
                     .Update
               End With
         Next
   Next
   'gehe zum ersten Datensatz
   rs.MoveFirst

   'schreibe die Daten ins aktuelle Sheet
   With ActiveSheet
         'Überschrift
         For spalte = 1 To 3
               .Cells(1, spalte).Value = rs.fields.Item(spalte - 1).Name
         Next spalte
         'Daten
         zeile = 1
         Do While Not rs.EOF
               For spalte = 1 To 3
                     .Cells(zeile + 1, spalte).Value = rs.fields.Item(spalte - 1).Value
               Next spalte
               rs.MoveNext
               zeile = zeile + 1
         Loop
   End With
   'Recordset schliessen
   .Close
End Sub
```

Coding 151: EZB: tägliche Devisenkurs-XML auslesen

8.3.4. Die Shell-Bibliothek

Mit der Shell Bibliothek können grundlegende System- und Dateifunktionen ausgeführt werden.

```
Dim sh      As Shell32.Shell
Dim fol     As Shell32.Folder
```

```
Dim it      As Shell32.ShellFolderItem
Dim val(30)
```

Auslesen von Dateiinformationen

```
Sub DateiDetailInfo()
    Set sh = New Shell32.Shell
    Set fol = sh.Namespace("G:\")
    Set it = fol.ParseName("IMGP0368.JPG")
    For i = 0 To 30
        val(i) = fol.GetDetailsOf(it, i)
    Next i
End Sub
```

Coding 152: Dateiinformationen mittels Shell auslesen

Systemzeit einstellen

```
Sub Zeit_einstellen()
    Set sh = New Shell32.Shell
    sh.SetTime
End Sub
```

Coding 153: Systemzeit mittels Shell neu einstellen

Windows beenden

```
Sub Windows_beenden()
    Set sh = New Shell32.Shell
    sh.ShutdownWindows
End Sub
```

Coding 154: Windows mittels Shell beenden

8.4. Andere Office-Applikationen

8.4.1. Worddokument ändern

Mit der Bibliothek Word 11 soll nun gearbeitet werden. Nachdem der Verweis erstellt wurde, müssen nun noch die Variablen angelegt werden. Anschliessend soll ein bestehendes Worddokument „C:\test.doc“ geöffnet und das zweite Wort geändert werden:

```
Dim wapp As Word.Application
Dim docs As Word.Documents
Dim doc  As Word.Document

Sub test()
   Set wapp = New Word.Application
   Set docs = wapp.Documents
   Set doc = docs.Open("c:\test.doc")

   doc.Words.Item(2) = "Teständerung"

   doc.Close True
   Set docs = Nothing
   Set wapp = Nothing
End Sub
```

Coding 155: Ändern eines Worddokumentes

8.4.2. Texte in Word schreiben

Damit man nun nicht jedes Wort einzeln bearbeiten muss, gibt es eine Methode, die den Text bei Übergabe entsprechend anlegt. Gleichzeitig kann natürlich auch der Schrifttyp und die Ausrichtung geändert werden.

```
Sub text_schreiben()
   Dim wapp As Word.Application
   Dim docs As Word.Documents
   Dim doc  As Word.Document
   Set wapp = New Word.Application
   Set docs = wapp.Documents
   Set doc = docs.Open("C:\test.doc")
   With doc.Application.Selection
   'schreibt einen Text
         .TypeText "Hallo, dies ist ein "
```

```
   'erzeugt die Eingabe von Enter
         .TypeParagraph
   'verändert den Schrifttyp
         .Font.Name = "Arial"
         .Font.Size = 18
         .Font.Bold = True
   'Textausrichtung zentriert
         .ParagraphFormat.Alignment = wdAlignParagraphCenter
   'schreibt einen Text
         .TypeText "Testtext."
   'erzeugt die Eingabe von Enter
         .TypeParagraph
   'Textausrichtung linksbündig
         .ParagraphFormat.Alignment = wdAlignParagraphLeft
   'verändert den Schrifttyp
         .Font.Name = "Times New Roman"
         .Font.Size = 12
         .Font.Bold = False
   End With
   doc.Close
   Set doc = Nothing
   Set docs = Nothing
   Set wapp = Nothing
End Sub
```

Coding 156: Text in Word schreiben

8.4.3. Kopfzeile in Worddokument einfügen

Hier noch kurz ein Beispiel, wie man eine Kopzeile in ein Worddokument einfügt:

```
Sub create_header()
   ActiveWindow.ActivePane.View.SeekView = wdSeekCurrentPageHeader
   Selection.TypeText Text:="Hallo"
   ActiveWindow.ActivePane.View.SeekView = wdSeekMainDocument
End Sub
```

Coding 157: Kopfzeile in Worddokument

8.4.4. Exceldokument ändern

Um ein Exceldokument zu ändern entfällt der Verweis, da Excel bereits vollständig geladen ist. Einzig die Deklaration der Variablen fehlt:

```
Dim xapp  As Excel.Application
Dim wbs   As Excel.Workbooks
Dim wb    As Excel.Workbook

Sub test()
   Set xapp = New Excel.Application
   Set wbs = xapp.Workbooks
   Set wb = wbs.Open("c:\test.xls")
   wb.ActiveSheet.Cells(1, 1).Value = "Test"
   wb.Close True, "e:\test.xls"
   Set wbs = Nothing
   Set xapp = Nothing
End Sub
```

Coding 158: Ändern eines Exceldokumentes

8.4.5. Adressen aus Outlook auslesen

Mit nachfolgendem Coding können die Namen der in Outlook gespeicherten Kontakte ausgelesen werden:

```
Dim wappAs Outlook.Application
Dim adrlst  As Outlook.AddressLists
Dim lst As Outlook.AddressList
Dim adrsAs Outlook.AddressEntries

Sub outl_contact()
   Set wapp = New Outlook.Application
   Set adrlst = wapp.Session.AddressLists
   Set lst = adrlst.Item(2)
   Set adrs = lst.AddressEntries
```

```
   With adrs
         For x = 1 To .Count
               MsgBox prompt:=.Item(x).Name
         Next x
   End With
End Sub
```

Coding 159: Auslesen der Kontakte aus Outlook

8.4.6. Termin in Outlook eintragen

Einen Termin kann man mit folgendem Coding, recht einfach in Outlook eintragen:

```
Dim wappAs Outlook.Application
Dim termin  As Outlook.AppointmentItem

Sub outl_reminder()
   Set wapp = New Outlook.Application
   Set termin = wapp.CreateItem(olAppointmentItem)
   With termin
         .Subject = "Testtermin"
         .AllDayEvent = False
         .ReminderSet = True
         .Start = "20.09.2009 18:30"
         .End = "20.09.2009 20:00"
         .Save
   End With
End Sub
```

Coding 160: Termin in Outlook eintragen

8.4.7. E-Mails aus Outlook auslesen

Mit folgendem Coding werden die E-Mails aus Outlook ausgelesen und in einem Excelsheet aufgelistet:

```
Dim outl       as Outlook.Application
Dim fold       as Outlook.MAPIFolder
Dim mail       as Outlook.MailItem

Dim anzMail    as Integer
Dim Zeile      as Integer

Sub outlook_mail_lesen()
   Set outl = New Outlook.Applikation
   Set fold = outl.Session.Folders.Item(1).Folders.Item(„Posteingang“)

   Zeile = 2
   With Tabelle1
         .Cells(1,1).Value = „Mail vom”
         .Cells(1,2).Value = „Sender”
         .Cells(1,3).Value = „Betreff”

         For anzMail = 1 to fold.Items.Count
               Set mail = fold.Items.Item(anzMail)
               .Cells(Zeile,1).Value = mail.ReceivedTime
               .Cells(Zeile,2).Value = mail.SenderName
               .Cells(Zeile,3).Value
               Zeile = Zeile +1
         Next anzMail
   End With
End Sub
```

Coding 161: Mails aus Outlook auslesen

8.5. Verbindungen mit anderen Programmen (fremde API's)

8.5.1. Verbindung zu HPQC aufbauen und Bugs auslesen

Hierfür wird ein Verweis auf die OTA COM TYPE LIBRARY benötigt – dieser kann auch manuell gelegt werden. Dazu die Datei „OTAClient.DLL" im Verzeichnis des HPQC's verlinken.

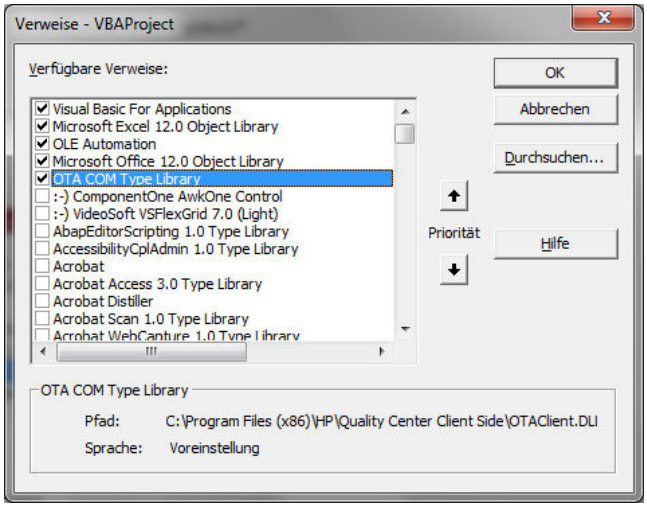


Abbildung 26: Verweis auf HPQC Bibliothek

Der Aufruf selbst verlangt die Mitgabe der URL, sowie der Projekt und der Anmeldedaten:

```
Dim ota  As TDAPIOLELib.TDConnection


Sub HPQC()
   strQCUrl = "http://si0bos98.de.firma.com:8080/qcbin"
   strDomain = "UBK"
   strProject = "AIM_UBK"
   strusername = "skl6fe"
   mypw = InputBox("Passwort bitte:", "Passwort")
   strpassword = mypw

   Dim test As Object

   Set ota = New TDAPIOLELib.TDConnection
   With ota
         .InitConnectionEx strQCUrl
         .ConnectProjectEx strDomain, strProject, strusername, strpassword

         Set o_Bfactory = .BugFactory
         Set o_BFilter = o_Bfactory.Filter
         Set o_BList = o_BFilter.newlist
   End With

   Zeile = 1
   For Each bug In o_BList
         Set o_BLink = bug.LinkFactory
         Set o_BLinkList = o_BLink.newlist("")
         If o_BLinkList.Count > 0 Then
               For Each TestLink In o_BLinkList
                     Set test = TestLink.TargetEntity
                     Tabelle1.Cells(Zeile, 1).Value = bug.ID
                     Tabelle1.Cells(Zeile, 2).Value = bug.Priority
                     Tabelle1.Cells(Zeile, 3).Value = bug.Status
                     Tabelle1.Cells(Zeile, 4).Value = TestLink.creationdate
                     Zeile = Zeile + 1
               Next
         End If
         DoEvents
   Next
   MsgBox prompt:="Bin fertig!"
End Sub
```

Coding 162: Bugs aus HPQC auslesen und in Excel anlisten

Wenn nicht alle Daten angelistet werden sollen, so können diese gefiltert werden. Hierzu wird einfach der relevante Felder gefiltert

```
o_BFilter.Filter("BG_STATUS") = "<> CLOSED"
```

Coding 163: HPQC Daten filter

9. Menüsteuerung

9.1. Kontextmenüs

Während sich eigene Funktionen in Microsoft Excel relativ leicht erstellen lassen, ist deren Bereitstellung eher etwas spartanisch. Im Formelmenü unter den Benutzerdefinierten Funktionen werden alle öffentlichen Funktionen angeboten. Für den Endanwender manchmal etwas mühsam.

9.1.1. Einrichten von Kontextmenüs

Eigene Funktionen können sehr leicht in das Kontextmenü von Excel eingebunden werden:

```
Dim cont_men As Object

Private Sub cont_menu()
   Set cont_men = CommandBars("Cell").Controls.Add
   cont_men.BeginGroup = true
   With cont_men
         .Caption = „meine Funktion”
         .onAction = „Makro1”
         .FaceId = 71
   End With
End sub
```

Coding 164: Einbinden einer Funktion in das Excel Kontextmenü

Mit dem Setzen von …BeginGroup = True wird ein Trennstrich oberhalb des Menüeintrages erzeugt. Die Angabe von …onAction muss auf ein eindeutiges Makro (Funktion) zeigen. Die FaceID ist optional und verziert den Menüeintrag mit einem Icon.

9.1.2. Löschen von Kontextmenüs

Hier gibt es zwei Möglichkeiten: Zum Einen kann das gesamte Menü resetet werden, womit dann alle Einträge auf den Excel-Standard zurückgesetzt werden, zum Anderen kann ein einzelner Eintrag gezielt gelöscht werden:

```
Private Sub clear_menu()
   Application.CommandBars(„Cell“).reset
End sub
```

Coding 165: Kontextmenü reseten

```
Private Sub del_menu_item()
   Application.CommandBars(„Cell”).Controlls(„meine Funktion”).delete
End sub
```

Coding 166: Kontextmenüeintrag löschen

Zudem kann natürlich das Standardmenü ausgeblendet werden, bevor eigene Menüeinträge erstellt werden. Dies ist besonders dann sinnvoll, wenn viele neue Einträge erfolgen sollen. Dabei wird gleich noch aufgezeigt, dass ein Menüeintrag auch über sein Index angesprochen werden kann.

```
Private Sub del_menu()
   With CommandBars("Cell")
         Do While .Controls.Count > 0
               .Controls(1).Delete
         Loop
   End with
End sub
```

Coding 167: Gesamtes Kontextmenü löschen

9.1.3. Kontextmenü als Klasse

Mit der Kapselung des Kontextmenü-Codings in einer Klasse lässt sich das Handling wesentlich vereinfachen:

```
Dim cont_menu   As Object
Dim ok  As Boolean

Function set_cont_menu(newGroup As Boolean, mCaption As String, _
                        mOnAction As String, mFaceId As Integer) As Boolean
   On Error GoTo errhandler
   Set cont_menu = CommandBars("Cell").Controls.Add
   cont_menu.BeginGroup = newGroup
   With cont_menu
         .Caption = mCaption
         .onAction = mOnAction
         .FaceId = mFaceId
   End With
   ok = True
   GoTo ende
errhandler:
   ok = False
ende:
   set_cont_menu = ok
End Function
```

Coding 168: Kontextmenü als Klasse

Im Modul-Coding muss nun nur noch die Klasse instanziert werden und die Funktion ausgeführt werden. Über den Rückgabewert „OK“ erhält man gleich ein Feedback darüber, ob die Verarbeitung fehlerfrei verlief.

9.2. Menü in der Menüleiste

9.2.1. Erstellen einer eigenen Menüleiste

Viele Routinen sollen erst per Useraktion gestartet werden. Hierzu können in Worksheet Schalter angelegt werden oder der User wird auf die Starttaste der Makro-Menüleiste verwiesen.
Wesentlich schöner und benutzerfreundlicher ist da eine eigene Menüleiste. Diese wird wieder mit dem Commandbar – Objekt aufgebaut:

```
Dim mybar   As CommandBar
Dim myentry(1 To 5) As CommandBarButton

Private Sub build_menu()
   Set mybar = Application.CommandBars.Add

   With mybar
         .Name = „newBar”
         .Visible = True
         .Position = msoBarTop
   End With

   Set myentry(1) = mybar.Controls.Add

   With myentry(1)
         .Visible = True
         .Style = msoButtonCaption
         .Caption = „Aktion”
         .onAction = „Makro1”
         .Tooltiptext = „Für die erste Aktion aus”
   End With
End sub
```

Coding 169: Erstellen einer Menübar

9.2.2. Schalter mit Icon

Nun erscheint ein Menü mit einem beschrifteten Schalter. Soll dieser über ein Icon verfügen (was wesentlich Platzsparender ist), so muss der Button wie folgt deklariert werden:

```
Set myentry(1) = mybar.Controls.Add

With myentry(1)
      .Visible = True
      .Style = msoButtonIcon
      .FaceId = 71
      .onAction = „Makro1”
      .Tooltiptext = „Für die erste Aktion aus”
End With
```

Coding 170: Schalter mit Icon

9.2.3. Schalter mit eigenem Icon

Neben den FaceID´s, die aus einer VBA-eigenen Bibliothek geladen werden, können auch eigene Bilder verwendet werden. Diese sollten für Schalter im Format BMP, 16x16 Pixel vorliegen. Das Einbinden geschiet dann wie folgt:

```
Set myentry(1) = mybar.Controls.Add

With myentry(1)
      .Visible = True
      .Style = msoButtonIcon
      .Picture = LoadPicture(„C:\Windows\Bild.bmp”)
      .onAction = „Makro1”
      .Tooltiptext = „Für die erste Aktion aus”
End With
```

Coding 171: Schalter mit eigenem Icon

9.2.4. Eigene Icons in Imagelist

Damit neben der Exceldatei nicht noch duzende von Bilddateien mitgegeben werden müssen, können die Icon´s in einem Imagelist-Objekt geparkt werden. Dieses Objekt kann auf einer beliebigen Form platziert werden – es ist zur Laufzeit nicht sichtbar und stört daher nicht. Der Aufruf wird dann wie folgt vorgenommen:

```
Set myentry(1) = mybar.Controls.Add

With myentry(1)
      .Visible = True
      .Style = msoButtonIcon
      .Picture = frm_about.ImageList1.ListImages(1).Picture
      .onAction = „Makro1”
      .Tooltiptext = „Für die erste Aktion aus”
End With
```

Coding 172: Eigenes Icon in Imagelist

9.3. Popup Menü

Das wahrscheinlich bekannteste ist das Popup-Menü. Es eignet sich für die detaillierte Programmsteuerung.

```
Dim mybar   As CommandBar
Dim myentry(1 To 5) As CommandBarpopup

Private Sub build_menu()
   Set mybar = Application.CommandBars.Add

   With mybar
         .Name = „newBar”
         .Visible = True
         .Position = msoBarTop
   End With

   Set myentry(1) = mybar.Controls.Add(type:=msocontrolpopup)
```

```
   With myentry(1)
         .Caption = „Aktion”
         .onAction = „Makro1”
         .Tooltiptext = „Für die erste Aktion aus”
   End With
End sub
```

Coding 173: Popup Menü einrichten

9.3.1. Standard Popup Menü erweitern

Mit folgendem Code kann das Standard Popupmenü von Excel um einen Eintrag erweitert werden:

```
Sub new_menu()
   With CommandBars.ActiveMenuBar.Controls
         Set newmenu = .Add(msoControlPopup, 10, , , True)
   End With
   newmenu.Caption = "Selbstdefiniert"

   With newmenu.CommandBar.Controls
         Set ctrl1 = .Add(msoControlButton, 1)
   End With

   With ctrl1
         .BeginGroup = False
         .Caption = "Eintrag1"
         .TooltipText = "Eintrag1"
         .Style = msoButtonIconAndCaption
         .FaceId = 360
   End With
End Sub
```

Coding 174: Eintrag in das Standard Popupmenü einfügen

Das entfernen funktioniert dann so:

```
Sub del_menu()
   CommandBars.ActiveMenuBar.Controls("Selbstdefiniert").Delete
End Sub
```

Coding 175: Menüeintrag entfernen

9.4. allgemeines zu Menüs

9.4.1. Neue Gruppe im Menü

Zusätzlich können vertikale Trennlinien die einzelnen Schalter gruppieren. Hierzu muss lediglich der Befehl „…BeginGroup = true“ eingefügt werden:

```
   Set myentry(1) = mybar.Controls.Add
   Myentry(1).BeginGroup = True
   With myentry(1)
         .Visible = True
         .Style = msoButtonIcon
         .FaceId = 71
         .onAction = „Makro1”
         .Tooltiptext = „Für die erste Aktion aus”
   End With
```

Coding 176: Gruppen im Menü

9.4.2. FaceId

Um einfach und schnell an die FaceId zu kommen, bietet es sich an, ein eigenes kleines Programm für Excel zu schreiben und sich dort die Icons mit ihrer Nummer, der FaceId, anzeigen zu lassen:

```
Private Sub ShowFaceIDs()
   Dim von As Integer
   Dim NewToolbar As CommandBar
   Dim NewButton As CommandBarButton
   Dim x As Integer, IDStart As Integer, IDStop As Integer
```

```
   von = InputBox("Beginn bei FaceID...", "Abgrenzung Faceid", 1)
   On Error Resume Next
   Application.CommandBars("FaceIds").Delete
   On Error GoTo 0

   Set NewToolbar = Application.CommandBars.Add _
         (Name:="FaceIds", temporary:=True)
   NewToolbar.Visible = True
   IDStart = von
   IDStop = von + 250
   For x = IDStart To IDStop
         Set NewButton = NewToolbar.Controls.Add _
               (Type:=msoControlButton, ID:=2950)
         NewButton.FaceId = x
         NewButton.Caption = "FaceID = " & x
   Next x
   NewToolbar.Width = 600
End Sub
```

Coding 177: Programm zu Anzeige der FaceId´s

Das Programm erstellt eine neue Toolbar (versucht diese vorher zu löschen) und zeigt in diesem die FaceId´s an. Damit nicht der ganze Bildschirm voller Icon´s hängt, wird bei Aufruf ein Startwert abgefragt, ab dem die Icons angezeigt werden. Die Anzeige ist auf 250 begrenzt.
Es befindet sich nicht auf jedem Wert ein Icon, aber der Bereich ist bis über 10000 (!) immer wieder mit Werten gefüllt. Für den allgemeinen Gebrauch dürften die ersten 2000 absolut ausreichend sein.

9.4.3. Combobox in der Menüleiste

Um nun mehrere Funktionen in ein Steuerelement zu kapseln, bietet sich die Combobox an. Diese kann ebenfalls relativ leicht integriert werden. Hierzu muss nachstehendes Coding in die Definition einer Commandbar eingebunden werden:

```
Dim cbo As Office.CommandBarComboBox
Set cbo = CommandBars("Test").Controls.Add(Type:=msoControlComboBox)

With cbo
   .OnAction = "Makro1"
   .TooltipText = "Tooltip text here"
   .Tag = "ShortIdName"
   .Caption = "Put the caption here"
   .Width = 100
   .DescriptionText = "Enter a description here"
   .DropDownLines = 8
   .DropDownWidth = 121
   .AddItem "Menü 1"
   .AddItem "Menü 2"
   .AddItem "Menü 3"
End With
```

Coding 178: Combobox im Menü

Beim Anklicken des Menüs wird durch folgendes Coding der Listindex ausgelesen und dann das entsprechende Coding ausgeführt:

```
Sub Makro1()
   Dim cboAction As Office.CommandBarComboBox
   Dim cboNext As Office.CommandBarComboBox
   Set cboAction = CommandBars.ActionControl
   Select Case cboAction.ListIndex
   Case 1
         MsgBox "erstes menu"
   Case 2
         MsgBox "zweites menu"
   Case Else
         MsgBox "anderes menu"
   End Select
End Sub
```

Coding 179: Combobox Event

9.4.4. Textbox in der Menüleiste

Für die direkte Eingabe kann eine Textbox eingebunden werden:

```
Dim mybar   As CommandBar
Dim xtext   As CommandBarControl

Private Sub Workbook_AddinInstall()
   Set mybar = Application.CommandBars.Add
   With mybar
         .Name = "newbar"
         .Visible = True
         .Position = msoBarTop
   End With
   Set xtext = mybar.Controls.Add(msoControlEdit, 1)
   With xtext
         .OnAction = "mymakro.xla!aktion"
   End With
End Sub
```

Coding 180: Textbox in Menü einbinden

9.5. Menü entfernen

Beim Beenden des Programmes sollte die Menüleiste wieder entfernt werden.

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
   application.CommandBars("newbar").Delete
End Sub
```

Coding 181: Entfernen eines Menüs beim Beenden des Programmes

10. Sonstige Techniken

10.1. mit Datum arbeiten

10.1.1. Kalenderwoche ermitteln

Hierfür steht die interne Funktion „DatePart“ zur Verfügung. Zu berücksichtigen ist die unterschiedliche Ermittlung zwischen US Kalendern und europäischen Kalender.

```
Funktion aktuelle_kw(ByVal datum1 as Date)
   Dim dataout as byte
   dataout = DatePart("ww", datum1, vbUseSystemDayOfWeek, vbFirstFourDays)
   aktuelle_kw = dataout
End Function
```

Coding 182: Ermittlung der Kalenderwoche

10.1.2. Wochentag ermitteln

Um festzustellen, um was für einen Wochentag es sich bei einem Datum handelt hilft folgende Funktion:

```
Function weekday(ByVal datum1 As Date)
   Dim dataout As String
   dataout = DatePart("w", datum1, vbSunday, vbUseSystem)
   weekday = dataout
End Function
```

Coding 183: Ermittlung des Wochentages

Als Ergebnis erhält man nun einen Wert die wie folgt zu interpretieren ist:

Wert	Beschreibung
1	Sonntag
2	Montag
3	Dienstag
4	Mittwoch
5	Donnerstag
6	Freitag
7	Samstag

Tabelle 15: Wochentagswerte

10.2. Textzeichenvergleiche

10.2.1. Eingabe in Textbox beschränken

Manchmal ist es sinnvoll die Eingabe von Zeichen in eine Textbox zu beschränken, zum Beispiel dann, wenn nur Zahlen eingegeben werden dürfen:

```
Private Sub textbox1_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
   Zugelassen = "1234567890," & Chr$(8)
   If InStr(1, Zugelassen, Chr$(KeyAscii)) = 0 Then
         KeyAscii = 0
   End If
End Sub
```

Coding 184: Eingabe in Textbox beschränken

10.2.2. Variableninhalte beschränken

Wenn ein Variableninahlt auf unerlaubte Zeichen hin überprüft werden soll, so kann dies ebenfalls über die InStr-Funktion erledigt werden:

```
Function compare_var(byval Filename as String)as Boolean
   Dim Dataout as boolean
   Dataout = true
   Prohib = ",/\:"
   For x = 1 To Len(Filename)
```

```
         If InStr(1, Prohib, Mid$(full_Filename, x, 1)) <> 0 Then
               Msgbox promp:="Die Angabe des INI-File enthält unerlaubte Zeichen!"
               Dataout = false
         End If
   Next x
End Function
```

Coding 185: Variableninhalt beschränken

10.3. Formatieren von Strings

Vielfach ist es notwendig Daten in ein entsprechendes Format zu bringen, beispielsweise bevor diese an ein anderes System übergeben werden soll (Alphakonvertierung für SAP RFC´s).

```
VAR1 = Format(VAR1, „00000000“)
```

Coding 186: Formatieren von Strings

10.4. Unicode

10.4.1. ASCII Nummer aus Unicodetabelle ermitteln

String können recht einfach auf Unicode geprüft werden:

```
Sub test()
   Dim i As Long
   Dim x() As Byte
   x = StrConv("ABCDEFG", vbFromUnicode)
   For i = 0 To UBound(x)
         MsgBox prompt:=x(i)
   Next
End Sub
```

Coding 187: Unicode-Convertierung

10.4.2. String in Unicode wandeln

Zur Verarbeitung von Daten kann es notwendig sein, diese vorher in Unicode zu wandeln. Hierfür werden die entsprechenden Zeichen im String einfach mit Replace ersetzt:

```
Sub test()
   wert = "Häberle"
   repwert = Replace(wert, "ä", "ae")
   MsgBox prompt:=repwert
End Sub
```

Coding 188: Sting in Unicode umwandeln

10.5. String verschlüsseln

Eine relativ einfache Art und Weise, einen String zu verschlüsseln bietet nachfolgende Routine. Dabei werden die einzelnen Zeichen des Strings einfach nur über die XOR-Verknüpfung substituiert.

```
Dim dataout As String
Dim i   As Integer

Function kodierung(ByVal inhalt As String, ByVal code As Byte) As String
   For i = 1 To Len(inhalt)
         dataout = dataout & Chr(code Xor Asc(Mid(inhalt, i, 1)))
   Next i
   kodierung = dataout
End Function
```

Coding 189: Einfache Verschlüsselung von Strings

10.6. Entfernen von Leerzeichen

Manchmal können Werte nur in String übergeben werden. Dann besteht die Gefahr, dass führende oder nachstehende Leerzeichen den Wert zum „Text“ machen, sobald dieser in eine Zelle geschrieben wird.

```
LTRIM(VAR1)
```

Coding 190: Entfernen von linksbündigen Leerzeichen

```
RTRIM(VAR1)
```

Coding 191: Entfernen von rechtsbündigen Leereichen

```
Trim(VAR1)
```

Coding 192: Entfernen von links- und rechtsbündigen Leerzeichen

10.7. Gross- und Kleinbuchstaben

In manchen Fällen werden unbedingt Gross- bzw. Kleinbuchstaben benötigt. Dies kann über folgende Befehle erreicht werden:

```
VAR1 = UCASE(„Peter“)
```

Coding 193: Erzeugen einer Zeichenkette mit Grossbuchstaben

```
VAR2 = LCASE(„Peter“)
```

Coding 194: Erzeugen einer Zeichenkette mit Kleinbuchstaben

10.8. Strings vergleichen

10.8.1. Der Like Operator

Der Vergleich von Strings wird unter anderem bei der Suche in Datenfelder benötigt. Mit dem like Befehl lassen sich auch Suchmuster erstellen:

Suche nach einer Ziffer im Suchmuster:

```
If „Lieferung #“ like „Lieferung 1“ then msgbox prompt:=„gefunden“
```

Coding 195: Suche nach einer Ziffer

Suche nach einem oder mehreren Zeichen:

```
If „Lie*“ like „Lieferung“ then msgbox prompt:=„gefunden“
```

Coding 196: Suche nach Zeichen

Suche nach einem einzelnen Zeichen:

```
If „Li?ferung“ like „Lieferung“ then msgbox prompt:=„gefunden“
```

Coding 197: Suche nach einem Zeichen

Suche nach einer Zeichenfolge, die nicht im String enthalten ist:

```
If „A“ like „[!Lieferung]“ then msgbox prompt:=„ein A ist nicht enthalten“
```

Coding 198: Suche nach nicht enthaltener Zeichenfolge

Suche nach einer Zeichenfolge, die im String enthalten ist:

```
If „L“ like „[Lieferung]“ then msgbox prompt:=„ein L ist enthalten“
```

Coding 199: Suche nach enthaltener Zeichenfolge

10.8.2. Die Compare Funktion

Mit der Compare Funktion lassen sich zwei Strings auf gleichen Inhalt hin vergleichen:

```
Dim a, b, c

Sub vgl_comp()
   a = "TESTTEXT": b = "testtext"
   c = StrComp(b, a)
End Sub
```

Coding 200: Strings strcomp vergleichen

C liefert als Rückgabewert „1“. Optional kann noch die Art des Zeichenfolgenvergleichs festgelegt werden.

```
Dim a, b, c

Sub vgl_comp()
   a = "TESTTEXT": b = "testtext"
   c = StrComp(b, a, vbTextCompare)
End Sub
```

Coding 201: Strings strcomp vergleichen – Textcompare

Hierbei liefert C den Rückgabewert „0“.

10.9. Zwischenablage

10.9.1. Daten in die Zwischenablage kopieren

In manchen Fällen ist es praktisch Daten in die Zwischenablage zu stellen. Hierfür muss ein Verweis auf die Bibliothek Microsoft Forms 2.0 gelegt sein. Das Beispiel ist als Funktion aufgebaut.

```
function set_clip(wert as string)
   Dim clip As DataObject
   Set clip = New DataObject
   clip.SetText wert
   clip.PutInClipboard
End function
```

Coding 202: Daten in die Zwischenablage kopieren

10.9.2. Daten aus der Zwischenablage lesen

Um die Daten aus der Zwischenablage auslesen werden nun die Methoden GetFromClipboard und GetText aufgerufen:

```
Function get_clip() As String
   Dim clip As DataObject
   Set clip = New DataObject
   clip.GetFromClipboard
   get_clip = clip.GetText
End Function
```

Coding 203: Daten aus der Zwischenablage lese

10.9.3. Mehrere Einträge in der Zwischenanblage verwalten

Mit jedem Aufruf der Methode PutInClipboard wird der alte Wert der Zwischenablage überschrieben. Um mehrere Einträge zu verwalten wird der Methode SetText noch ein weiterer Parameter mitgegeben:

```
Dim ClipAs DataObject
Dim T(2)As String

Sub multi_write()
   Set Clip = New DataObject
   T(1) = "text string one"
   T(2) = "text string two"
   With DataObj
```

```
            .SetText T(1), "Id1"
            .PutInClipboard
            .SetText T(2), "Id2"
            .PutInClipboard
      End With
End Sub
```

Coding 204: Mehrere Einträge ins Clipboard schreiben

```
Sub multi_read()
      Set Clip = New DataObject
      With Clip
            .GetFromClipboard
            T(1) = .GetText("Id1")
            T(2) = .GetText("Id2")
      End With
End Sub
```

Coding 205: Mehrere Einträge aus dem Clipboard lesen

10.10. Arbeiten mit importierten Daten

10.10.1. Text teilen

Es kann vorkommen, dass ein gelesener String eigentlich zwei Datenfelder enthält und geteilt werden muss. Dies kann über den Splitbefehl sehr einfach realisiert werden:

```
Function Splitten(Bezug As String, Trennzeichen As String, Teil As Integer)
      Bezug = LTRIM(Bezug)
      Splitten = Split(Bezug, Trennzeichen)(Teil - 1)
End Function
```

Coding 206: Splitten eines Textes

10.10.2. Zahlen formatieren

Gerade bei importierten Zahlen stellt sich immer wieder das Problem der korrekten formatierung. Zahlen werden nicht korrekt erkannt und als Text dargestellt. Schuld daran sind führende Leerzeichen und ein Minuszeichen am falschen Platz. Folgendes Coding schafft Abhilfe:

```
Function TXTtovalue(Bezug As String) As Currency
      Bezug = LTRIM(Bezug)
      If Right$(Bezug, 1) = "-" Then _
            Bezug = Left$(Bezug, (Len(Bezug) - 1)): Bezug = 0 - Bezug
      End if
      TXTtovalue = Bezug
End Function
```

Coding 207: Zahlen konvertieren

10.10.3. Alphakonvertierung

Ein anderes Problem stellt sich beispielsweise bei der Verwendung von Werten wie Buchungskreisen ect., die mit führenden Nullen angezeigt werden sollen. Folgende Funktion führt eine Alphakonvertierung durch. Dabei ist die Datenlänge flexibel einstellbar:

```
Function nullvor(Bezug As String, Stellen As Integer) As String
      For X = 1 To (Stellen - (Len(Bezug)))
            Bezug = 0 & Bezug
      Next X
      nullvor = Bezug
End Function
```

Coding 208: Alphakonvertierung

11. Kommunikation mit Datenbanken - DAO

Eines wird wohl gerade beim Arbeiten mit Excel immer wieder vorkommen: Das Problem mit der Datenhaltung. Nicht alle Daten, die in Excel verarbeitet werden, sollen auch dort gespeichert werden.

11.1. Organisation der DAO Objektstruktur

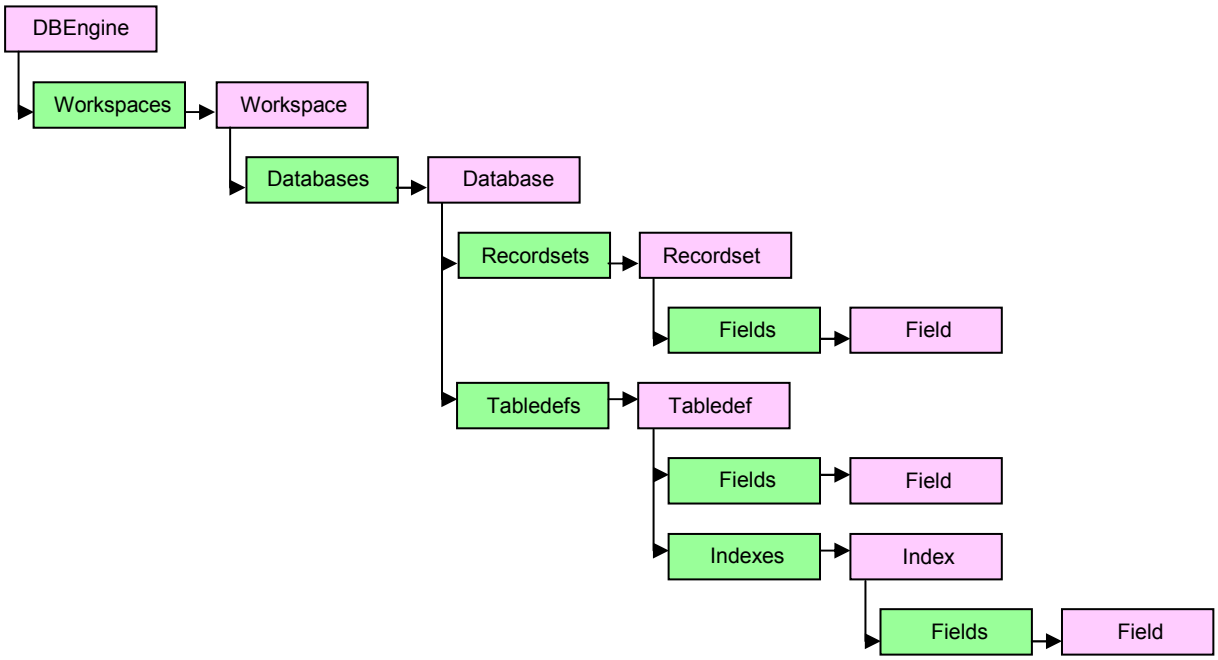


Abbildung 27: Organisation des DAO Objekts

11.2. Der DAO Connect

Einen Datenbankconnect mit DAO herzustellen ist sehr einfach, wie folgendes Beispiel zeigt:

```
Dim db   as dao.database

Sub con_dao()
   Set db = openDatabase(„d\test.mdb")
End sub
```

Coding 209: Connect via DAO

11.2.1. Daten in eine Datenbanktabelle schreiben

Nachfolgende werden über die Bibliothek DAO eine Datenbank und ein Recordset deklariert. In der Routine werden die Methodenaufrufe mit Parameterübergabe getätigt. Nach Aufruf der Methode „Addnew" findet die Parameterübergabe statt. Mit der Methode „Update" werden die übergebenen Feldinhalte weg geschrieben. Mit der Methode „Close" wird die Sitzung mit der Datenbank beendet.

```
Sub write_to_mdb()
   Dim db As DAO.Database
   Dim rs As DAO.Recordset
   Dim kto As String
   Dim bez As String

   Set db = OpenDatabase("d:\test.mdb")
   Set rs = db.OpenRecordset("Konten", dbOpenTable)
   kto = „471100"
   bez = „Testkonto"
   With rs
         .AddNew
         .Fields("Konto") = kto
```

```
            .Fields("Bezeichnung") = bez
            .Update
        End With
        rs.Close
        Set rs = Nothing
        db.Close
        Set db = Nothing
    End Sub
```

Coding 210: Daten in eine Datenbanktabelle schreiben

11.2.2. Daten aus einer Datenbank lesen

Die Deklaration und der Verbindungsaufbau ist identisch mit dem des Daten schreiben. Mit der Methode „MoveLast“ wird der letzte Datensatz aufgerufen – anschliessend werden die Felder „Konto“ und „Bezeichnung“ ausgelesen. Zum Schluss wird die Sitzung mit der Datenbank wieder beendet.

```
Sub read_from_mdb()
    Dim db As DAO.Database
    Dim rs As DAO.Recordset
    Dim kto As String
    Dim bez As String

    Set db = OpenDatabase("d:\test.mdb")
    Set rs = db.OpenRecordset("Konten", dbOpenTable)

    rs.MoveLast

    kto = rs.Fields("Konto")
    bez = rs.Fields("Bezeichnung")

    rs.Close
    Set rs = Nothing
    db.Close
    Set db = Nothing
End Sub
```

Coding 211: Daten aus einer Datenbank lesen

11.3. Methoden im Recordset

11.3.1. In Tabellen bewegen

Um sich in den Datensätzen einer Tabelle zu bewegen stehen folgende Methoden zur Verfügung:

Methode	Bedeutung
MoveFirst	gehe zu ersten Datensatz
MoveLast	gehe zum letzten Datensatz
MoveNext	gehe zum nächsten Datensatz
MovePrevious	gehe zum vorherigen Datensatz

Tabelle 16: Recordset: in Tabellen bewegen

Als Beispiel kann vorheriges Coding 211: dienen – vor Übergabe der Tabelleninhalte wurde mit `rs.movelast` auf den letzten Datensatz gemoved.

11.3.2. Gezieltes Suchen

Gerade beim Lesen will man in der Tabelle nicht nur rauf und runter scrollen, um an einen speziellen Datensatz zu gelangen. Hierzu gibt es ein paar Methoden, die bei der expliziten Suche behilflich sind. Zum einen kann nach dem ersten, nächsten, vorherigen oder letzten Datensatz mit einer Übereinstimmung gesucht werden, zum anderen kann nach einem String gesucht werden und dabei bis zu 13 Kriterien mitgegeben werden:

Methode	Bedeutung
Seek	Sucht einen Datensatz anhand von Suchkriterien
Findfirst	Sucht den ersten übereinstimmenden Datensatz
Findnext	Sucht den nächsten übereinstimmenden Datensatz
Findprevious	Sucht den vorherigen übereinstimmenden Datensatz
Findlast	Sucht den letzten übereinstimmenden Datensatz

Tabelle 17: Recordset: Daten suchen

Für ein Beispiel werden eigentlich mehr als nur ein Datensatz in der Tabelle benötigt, um den Effekt einer echten Suche zu haben:

```
Sub read_from_mdb()
   Dim db As DAO.Database
   Dim rs As DAO.Recordset
   Dim kto As String
   Dim bez As String

   Set db = OpenDatabase("d:\test.mdb")
   Set rs = db.OpenRecordset("Konten", dbOpenTable)

   With rs
         .Index = "PrimaryKey"
         .Seek "=", "471100"
   End With

   kto = rs.Fields("Konto").value
   bez = rs.Fields("Bezeichnung").value
   rs.Close
   Set rs = Nothing
   db.Close
   Set db = Nothing
End Sub
```

Coding 212: Daten in einer Datenbank finden

11.3.3. Bearbeiten von Datensätzen

Für das Anfügen, Löschen und Ändern stehen folgende Methoden zur Verfügung:

Methode	Bedeutung
AddNew	fügt Datensatz hinzu
Delete	löscht einen Datensatz
Edit	verändert einen Datensatz
Update	Änderung ausführen

Tabelle 18: Recordset: Daten anfügen, löschen, ändern

Hierzu ein Ausschnitt aus Coding 210:, mit dem ein neuer Datensatz hinzugefügt und anschliessend ein Update durchgeführt wurde.

```
   With rs
         .AddNew
         .Fields("Konto").value = kto
         .Fields("Bezeichnung").value = bez
         .Update
   End With
```

Coding 213: Methoden um einer Tabelle Daten anzufügen

11.4. Datenbank zur Laufzeit anlegen

11.4.1. Datenbanken anlegen

Mit DAO können auch Datenbanken zur Laufzeit angelegt werden. Dies ist zum Beispiel dann sinnvoll, wenn kein MS Access vorhanden ist, oder die Entscheidung zur Ablage der Daten in eine Datenbank erst zur Laufzeit fallen soll.
Auch für dieses Vorgehen muss ein Verweis auf die DAO Bibliothek gelegt werden.

```
Dim db as database

Sub db_anlegen
   Set db = DBEngine.CreateDatabase(„C:\meine.MDB", dbLangGeneral, dbEncrypt)
End Sub
```

Coding 214: Anlegen einer Datenbank zur Laufzeit

Dabei verbirgt sich hinter *dbLangGeneral* nur der Code für die Sprachen, in der die Datenbank installiert werden soll:

Sprachcode	Sprache
dbLangGeneral	Englisch, Deutsch, Französisch, Portugiesisch, Italienisch und Spanisch (Modern)
dbLangArabic	Arabisch
dbLangChineseSimplified	Vereinfachtes Chinesisch
dbLangChineseTraditional	Traditionelles Chinesisch
dbLangCyrillic	Russisch
dbLangCzech	Tschechisch
dbLangDutch	Niederländisch
dbLangGreek	Griechisch
dbLangHebrew	Hebräisch
dbLangHungarian	Ungarisch
dbLangIcelandic	Isländisch
dbLangJapanese	Japanisch
dbLangKorean	Koreanisch
dbLangNordic	Nordische Sprachen (nur Microsoft Jet-Datenbankmodul, Version 1.0)
dbLangNorwDan	Norwegisch und Dänisch
dbLangPolish	Polnisch
dbLangSlovenian	Slowenisch
dbLangSpanish	Traditionelles Spanisch
dbLangSwedFin	Schwedisch und Finnisch
dbLangThai	Thailändisch
dbLangTurkish	Türkisch

Tabelle 19: Sprachcodes für Datenbanken

Mit dem Begriff dbEncrypt wird eine Verschlüsselte Datenbank erstellt. Hierzu gibt es noch weitere Codes:

Konstante	Beschreibung
dbEncrypt	Erstellt eine verschlüsselte Datenbank.
dbVersion10	Erstellt eine Datenbank, die das Dateiformat des Microsoft Jet-Datenbankmoduls, Version 1.0, verwendet.
dbVersion11	Erstellt eine Datenbank, die das Dateiformat des Microsoft Jet-Datenbankmoduls, Version 1.1, verwendet.
dbVersion20	Erstellt eine Datenbank, die das Dateiformat des Microsoft Jet-Datenbankmoduls, Version 2.0, verwendet.
dbVersion30	(Standard) Erstellt eine Datenbank, die das Dateiformat des Microsoft Jet-Datenbankmoduls, Version 3.0 (kompatibel mit Version 3.5), verwendet.

Tabelle 20: Datenbankformate

Wird beim Erstellen der Datenbank kein Wert mitgegeben, so wird eine Datenbank im Format dbVersion30 angelegt.

11.4.2. Tabellen zur Laufzeit anlegen

Die zur Laufzeit angelegte Datenbank ist noch Leer und es werden Tabellen benötigt. Hierzu muss erst eine Tabellendefinition angelegt werden, die dann der Datenbank übergeben wird:

```
Dim tb As TableDef

Sub create_table()
   Set tb = db.CreateTableDef("DATA")
   With tb
         .Fields.Append .CreateField("Name", dbText)
   End With
   db.TableDefs.Append tb
End Sub
```

Coding 215: Tabellen zur Laufzeit anlegen

Die einzelnen Feldtypen sind dabei folgende:

Typ	Beschreibung
dbBigInt	Big Integer
dbBinary	Binary
dbBoolean	Boolean
dbByte	Byte
dbChar	Char
dbCurrency	Currency
dbDate	Date/Time
dbDecimal	Decimal
dbDouble	Double
dbFloat	Float
dbGUID	GUID
dbInteger	Integer
dbLong	Long
dbLongBinary	Long Binary (OLE Object)
dbMemo	Memo
dbNumeric	Numeric
dbSingle	Single
dbText	Text
dbTime	Time
dbTimeStamp	Time Stamp
dbVarBinary	VarBinary

Tabelle 21: Feldtypen für Datenfelder

Und nun einmal komplett – eine Datenbank, mit Tabelle und darin zwei Feldern erstellen:

```
Dim db         as DAO.Database
Dim rs_def     As TableDef

Sub erstelle_db()
   'Erstellen der Datenbankdatei
   Set db = DBEngine.CreateDatabase("e:\neuedb.mdb", dbLangGeneral)

   'Deklarieren einer Tabelle
   Set rs_def = db.CreateTableDef("Personen")

   With rs_def
         'Felder anfügen
                  'erstelle ein Feld "Name", Typ Text, 35 Zeichen lang
         .Fields.Append .CreateField("Name", dbText, 35)
                  'erstelle ein Feld "Wert", Typ Integer
         .Fields.Append .CreateField("Wert", dbInteger)
```

```
   End With

   'öffnen der Datenbank
   Set db = OpenDatabase("e:\neuedb.mdb")

   'Tabellendefinition übergeben
   db.TableDefs.Append rs_def

   'schliessen der Datenbank
   Set db = Nothing
End Sub
```

Coding 216: Datenbank mit Tabelle erstellen

11.4.3. Nachträglich Felder einfügen

Wie in eine bestehende Tabelle nachträglich Felder angefügt werden können, zeigt folgendes Coding:

```
Sub add_field()
   Dim db  As Database
   Dim tdf As TableDef
   Dim fld As Field

   Set db = opendatabase("e:\neuedb.mdb")
   Set tdf = db.TableDefs!Tabelle1
   Set fld = tdf.CreateField("Alter")
   With fld
         .Type = dbText
         .Size = 3
   End With
   tdf.Fields.Append fld
End Sub
```

Coding 217: Nachträgliches Einfügen von Feldern in Tabellen

11.4.4. Tabelle mit PrimaryKey erzeugen

Folgendes Beispiel soll zeigen, wie eine Tabelle mit einem PrimaryKey erzeugt werden kann:

```
Dim db  As DAO.Database
Dim td  As DAO.TableDef
Dim ind As DAO.index

Sub tabelle()
   Set db = OpenDatabase("e:\test.mdb")
   Set td = db.CreateTableDef("Dimension")
   Set ind = td.CreateIndex("PrimaryKey")
   With ind
         .Fields.Append .CreateField("Woche")
         .IgnoreNulls = False
         .Unique = True
         .Required = True
         .Primary = True
   End With
   With td
         .Fields.Append .CreateField("Woche", dbByte)
         .Fields.Append .CreateField("Monat", dbByte)
         .Indexes.Append ind
   End With
   db.TableDefs.Append td
End Sub
```

Coding 218: Tabelle mit PrimaryKey erstellen

11.4.5. Einen einfachen Index erzeugen

Neben dem eindeutigen PrimaryKey können noch weitere Indezes für Selektionen in der Tabelle angelegt werden. Dies ist auch bei bereits bestehenden Tabellen möglich:

```
Dim db  As DAO.Database
Dim td  As DAO.TableDef
Dim ind As DAO.index

Sub tabelle()
```

```
    Set db = OpenDatabase("e:\test.mdb")
    Set td = db.TableDefs("Dimension")
    Set ind = td.CreateIndex("Meinindex")
    With ind
          .Fields.Append .CreateField("Monat")
    End with
    td.Indexes.Append ind
End Sub
```

Coding 219: Anlegen eines einfachen Index

11.5. Auslesen der Datenbankobjekte

11.5.1. Tabellen

DAO ermöglicht aber nicht nur das Anlegen von Tabellen und Feldern, sonder auch das Auslesen, welche Tabellen und welche Felder angelegt sind.

```
Dim db  as dao.database
Dim tdefas dao.tabledefs
Dim x   as long

Sub read_all_tables()
   Set db=opendatabase(„e:\test.mdb")
   Set tdef = db.tabledefs
   For x = 0 to tdef.count
         Msgbox prompt:=tdef(x).name
   Next x
End sub
```

Coding 220: Abfrage aller Tabellen einer Datenbank

11.5.2. Tabellenfelder

Die Felder sind Teil des tabledef-Objektes. Die Abfrage muss daher lediglich ein wenig erweitert werden:

```
Dim db  as dao.database
Dim tdefas dao.tabledefs
Dim x   as long

Sub read_all_tables()
   Set db=opendatabase(„e:\test.mdb")
   Set tdef = db.tabledefs
   For x = 0 to tdef(1).fields.count -1
         Msgbox prompt:=tdef(1).Fields(x).name
   Next x
End sub
```

Coding 221: Abfrage aller Felder einer Tabellen

12. Kommunikation mit Datenbanken – ADO

Auch bei ADO gilt, erst einmal einen Verweis auf die Bibliothek einrichten: siehe Kapitel 8.2.1 Im konkreten Fall wird die Bibliothek zu Microsoft ADO 2.8 benötigt.

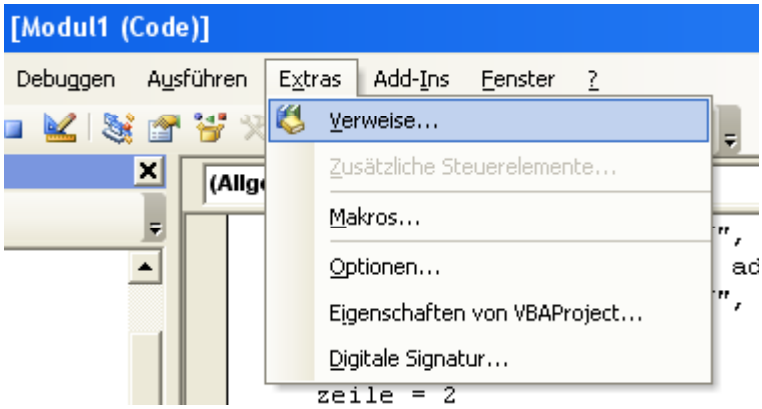


Abbildung 28: Aufruf des Verweise-Menüs

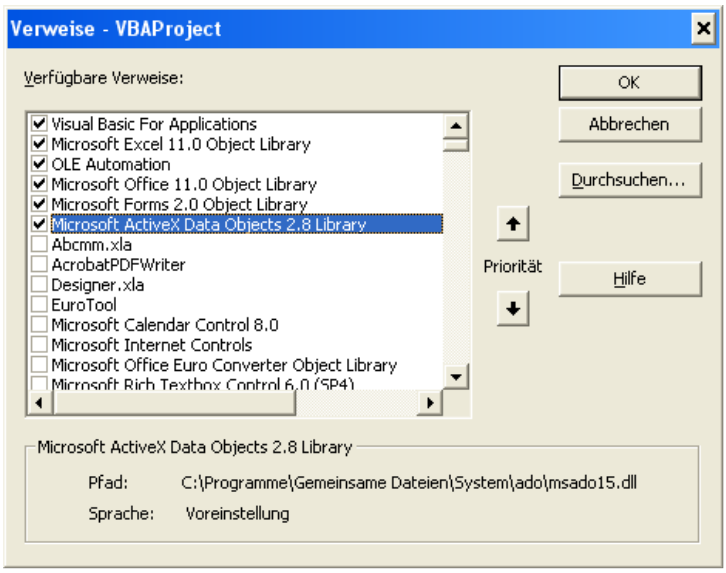


Abbildung 29: Einrichten des Verweises auf die ADO Bibliothek

12.1. Programminterne Verwendung (unbound)

12.1.1. Aufbau eines Recordsets

Grundsätzlich lassen sich Recordsets auch programmintern verwenden. Hierzu wird ein Recordset eingerichtet und mit einem Feldaufbau versehen. Als Beispiel wird eine Tabelle mit einem Feld für einen Namen (String – 20 Stellen) eingerichtet:

```
Dim mytable as Adodb.Recordset

Sub create_mytable()
   Set mytable = New Adodb.Recordset

   With mytable
         .fields.append „Name“, adchar, 20
         .fields.append „Alter“, adinteger, 2
   End with
End sub
```

Coding 222: ADO Tabelle erstellen

Die Feldtypen werden dabei wie folgt definiert:

Datentyp	Beschreibung
adBigInt	Big Integer Typ
adBinary	Binär-Typ
adBoolean	Boolescher Variant-Typ
adByRef	Refernz-Typ
adChar	Zeichenfolge mit fester Länge
adCurrency	Währungstyp
adDate	Datumstyp
adDBDate	Datumstyp im DB-Format (JJJJMMTT)
adDBTime	Zeittyp im DB-Format (hhmmss)
adDBTimeStamp	Zeitstempel im DB-Format (JJJJMMTThhmmss)
adDecimal	Dezimaler Variant - Typ
adDouble	Gleitkommazahl mit doppelter Genauigkeit
adInteger	Integer Typ
adLongVarBinary	Binär-Typ mit Long-variabler Länge
adLongVarChar	Zeichenfolge mit Long-variabler Länge
adLongVarWChar	Zeichenfolge mit Wide Long-variabler Länge
adNumeric	Nummerischer Typ
adSingle	Gleitkommazahl mit einfacher Genauigkeit
adVarBinary	Binär-Typ mit variabler Länge
adVarChar	Zeichenfolge mit variabler Länge
adVariant	Variant-Typ
adVarWChar	Zeichenfolge mit Wide-variabler länge
adWChar	Zeichenfolge mit Wide Länge

Tabelle 22: ADO Datentypen

12.1.2. Daten in die Tabelle schreiben

Als nächstes wird die Tabelle geöffnet, es werden Werte in die Tabelle geschrieben und die Tabelle wird wieder geschlossen. Beim Schreiben ist darauf zu achten, dass der Feldzeiger bei 0 zu zählen beginnt – somit ist das erste Feld = item(0).

```
Sub set_values()
   Mytable.open

   With mytable
         .addnew
               .fields.item(0).value = „Peter“
               .fields.item(1).value = 35
         .addnew
               .fields.item(0).value = „Karl“
               .fields.item(1).value = 42
         .update
   End with

   Mytable.close
End sub
```

Coding 223: ADO Tabelle mit Werten befüllen

Mit dem schliessen der Tabelle sind die Daten allerdings auch wieder weg!

12.1.3. Daten aus der Tabelle lesen

Das Lesen eines Datensatzes funktioniert wie das Schreiben, nur mit umgedrehtem Coding:

```
Dim Person as string
Dim Alter as string

Sub get_values()
   With mytable
         Person = .fields.item(0).value
         Alter = .fields.item(1).value
   End with
End sub
```

Coding 224: Datensatz aus einer Tabelle lessen

12.1.4. Den Datenzeiger bewegen

Vorab sollte man folgende Dinge wissen:
Der Recordset verfügt über die Haltepunkte BOF (Begin of File) und EOF (End of File). Der aktuelle Stand des Datenzeigers kann man mit AbsolutPosition und die Gesamtzahl aller Datensätze über RecordCount auslesen.

12.1.5. Move

Um sich innerhalb einer Tabelle zwischen den Datensätzen auf und ab zu bewegen werden folgende Anweisungen benötigt:

Anweisung	bewirkt
Move	geht zum gesuchten Datensatz
Movefirst	geht zum ersten Datensatz
Movelast	geht zum letzten Datensatz
Movenext	geht zum nächsten Datensatz
Moveprevios	geht zum vorherigen Datensatz

Tabelle 23: Move Anweisungen

Für die Anweisung MOVE wird ein Parameter benötigt. Gibt man beispielsweise nur eine Zahl mit – beispielsweise 5, so wird der Datenzeiger einfach 5 Positionen weiter gesetzt:

```
Sub test_move()
   With mytable
         .movefirst
         .move 5
   End with
   Msgbox prompt:= „Position: „ & mytable.absolutposition
End sub
```

Coding 225: Datensatzzeiger mit Move bewegen

Im obigen Beispiel wird der Datensatzzeiger mit Movefirst auf die erste Position gestellt. Anschliessend wird er mit Move 5 um 5 Positionen weiter nach unten verschoben. Als Ergebnis erhält man folgende Meldung:

Abbildung 30: Ergebnis für Move

12.1.6. Datensätze gezielt suchen - Find

Wenn die genaue Position des gesuchten Datensatzes nicht bekannt ist, so kann auf ganz einfache Art nach Ihm gesucht werden. Der Anweisung FIND wird einfach ein String als Parameter mitgegeben, der den Suchbefehl enthält:

```
Sub suche()
   With mytable
         .find „NAME like 'Peter'"
   End with
End sub
```

Coding 226: Beispiel für Find Anweisung

12.2. Datenaustausch mit Datenbanken (bound)

Primär ist ADO dafür gedacht, die Daten mit entsprechenden Datenbanken auszutauschen. Die Verbindung wird über ein Connection-Objekt hergestellt.

12.2.1. Verbindungsaufbau zu einer Access-Datenbank

```
Dim con As ADODB.Connection
Dim Db  as string

Sub Conn_to_Access()
   Db = application.getopenfilename(„Access Datenbank (*.mdb), *.mdb")
   Set con = New ADODB.Connection
   With con
         .CursorLocation = adUseClient
         .Provider = "Microsoft.Jet.OLEDB.4.0"
         .Open Db
   End With
End Sub
```

Coding 227: Verbindungsaufbau mit einer Access-Datenbank

Anfänglich muss die CursorLocation auf den Client gesetzt werden – andernfalls kann es zu Problemen kommen.
Bei MS Office ab Version 2000 läuft nichts mehr unter Version 4.0 des Jet OleDB Providers (vorher 3.5)

12.2.2. Abruf von Daten – Aufbau eines Recordset

Um nun ein paar Daten aus einer Tabelle abfragen zu können wird ein Recordset benötigt. Dieses muss nicht erst aufwendig – Feld für Feld – deklariert werden, sondern wird einfach per SQL abfrage erzeugt.

```
Dim rs(1 to 5) as adodb.recordset

Sub deklarieren()
   For x = 1 to 5
         Set rs(x) = new adodb.recordset
   Next x
End sub
```

Coding 228: Erzeugen von Recordsets

Nun wird der Satzzeiger auf den ersten Datensatz gestellt und anschliessend die Daten, vom Ersten bis zum Letzten (RecordCount = Anzahl der Datensätze) Datensatz ausgelesen.

```
Private Function load_user()
   rs(1).Open "Select * from AG_User", con, adOpenKeyset, adLockOptimistic
   rs(1).MoveFirst
   For count = 1 To rs(1).RecordCount
         Msgbox prompt:= rs(1).Fields("User").value
         rs(1).MoveNext
   Next count
End Function
```

Coding 229: Lesen eines Recordset über „OPEN" Befehl

Es ist auch möglich, den Recordset über den EXECUTE Befehl zu öffnen:

```
Sub read_table()
   Dim rs      as new recordset
   Rs.execute („select * from TABELLE order by KEY“)
End sub
```

Coding 230: Recordset über Execute lessen

12.2.3. Nur ersten Datensatz lesten

Die Übergabe von Werten an den Recordset funktioniert in genau umgedrehter Richtung, wie das Lesen. Danach werden die Daten mit dem Befehl „Update“ in die Datenbank geschrieben und der Recordset mit „Close“ geschlossen.

```
Private Function read_first_count()
   With rs(1)
         .open "Select Top 1 * from TABELLE", con, adOpenKeyset, adLockOptimistic
   End with
End Function
```

Coding 231: Liest nur den ersten Datensatz

12.2.4. Daten schreiben

Die Übergabe von Werten an den Recordset funktioniert in genau umgedrehter Richtung, wie das Lesen. Danach werden die Daten mit dem Befehl „Update“ in die Datenbank geschrieben und der Recordset mit „Close“ geschlossen.

```
Private Function write_user()
   With rs(1)
         .Addnew
               .Fields("User").Value = „Peter”
         .update
         .Close
   End with
End Function
```

Coding 232: Schreiben von Daten über Feldbezeichner

Die Daten können auch über deren Feldfolge abgefüllt werden. Zu beachten ist, dass die Reihenfolge mit 0 beginnt:

```
Sub write_data()
   With rs(1)
         .addnew
               .fields(0).value = „Peter“
         .update
         .close
   End with
End sub
```

Coding 233: Schreiben von Daten über Feldfolgennummer

12.2.5. Daten Auswählen Select und Select where

Daten werden mit Select ausgelesen. Dies kann über die Where Klausel zusätzlich eingeschränkt werden:

```
Private function sel_user
   Rs(1).open „Select * from AG_User where User = ‘Peter’”, _
         con, adOpenKeyset, adLockOptimistic
End Function
```

Coding 234: Datensätze selektieren / mit Where-Klausel

12.2.6. Sortieren mit Order by

Um die Daten in der richtigen Reihenfolge zu erhalten muss das Select-Statement um den Befehl Order erweitert werden:

```
Private function sel_user
```

```
   Rs(1).open „Select * from AG_User oder by User”, _
         con, adOpenKeyset, adLockOptimistic
End Function
```

Coding 235: Datensätze sortiert selektieren

12.2.7. Datensätze gruppieren

Sobald mehrere Datensätze zum Selektionskriterium gefunden warden, kann eine Gruppierung Sinn machen:

```
Sub group_field_ado()
   With rs
         .Open "Select Pers from mytab group by Pers", con, adOpenKeyset, adLockOptimistic
   End With
End Sub
```

Coding 236: Felder im Select-Statement gruppieren

12.2.8. Summenbildung

Meist soll neben der Gruppierung auch eine Summierung stattfinden. Beispiel hierfür wäre eine Verkaufsliste mit Artikeln und Mengen. Mehrere Artikel wurden mehrfach verkauft und sollen gruppiert werden. Nun müsste zusätzlich die Anzahl der Artikel summiert werden:

```
Sub groupsum_field_ado()
   With rs
         .Open "Select Artikel, sum(Menge) from mytab group by Artikel" &_
               , con, adOpenKeyset, adLockOptimistic
   End With
End Sub
```

Coding 237: Gruppierung und Summierung im Selecct-Statement

12.2.9. Datensätze löschen

Datensätze lassen sich am Besten über ein SQL Statement löschen:

```
Private Function del_user()
   rs(1).Open "Delete * from AG_User", con, adOpenKeyset, adLockOptimistic
End Function
```

Coding 238: Datensätze löschen

12.2.10. Tabelle erstellen per SQL Befehl

Der schnellste Weg eine Tabelle zu erstellen ist der über den SQL Befehl „Create Table“.

```
Sub erstelle_tabelle()
   Set rs = new adodb.recordset
   Rs.open „Create Table Testtabelle“,con
End sub
```

Coding 239: Tabelle erstellen per SQL

12.2.11. Feld in bestehende Tabelle einfügen

Auch mit ADO können, wie im Beispiel 11.4.3 für DAO gezeigt, Felder in eine bestehende Tabelle eingefügt werden:

```
Sub add_field_ado()
   With rs
         .Open "Alter Table Tabelle1 add Column Age Char(3)", con
   End With
End Sub
```

Coding 240: Nachträglich Felder in Tabelle einfügen

Statt der unter Tabelle 21: verwendeten Datentypen müssen nachstehende verwendet werden:

Typ	Beschreibung

Binary	Binary
Boolean	Boolean
Byte	Byte
Char	Char
Currency	Currency
Date	Date
Decimal	Decimal
Double	Double
Float	Float
Int	Integer
Long	Long
Numeric	Numeric
Single	Single
Text	Text
Time	Time

Tabelle 24: Datentypen bei Verwendung des SQL „Alter"-Statements

12.2.12. Löschen eines Tabellenfeldes

Mit dem Alter-Befehl lassen sich auch Felder aus einer bestehenden Tabelle entfernen:

```
Sub drop_field_ado()
   With rs
         .Open "Alter Table Tabelle1 drop Column Age", con
   End With
End Sub
```

Coding 241: Feld aus bestehender Tabelle löschen

12.2.13. Kopieren in neue Tabelle: Select Into

Die selektierten Daten sollen nun in eine neue Tabelle geschrieben werden:

```
Sub Select_into()
   rs.Open "Select * into Table2 from Table1", con, adOpenKeyset, adLockOptimistic
End Sub
```

Coding 242: Daten mit Select Into kopieren – SQL Server/Access

12.2.14. Kopieren in bestehende Tabelle: Insert Into

Um selektierte Daten von einer Tabelle in die Andere zu kopieren wird der Befehl Insert Into verwendet:

```
Sub Insert_into()
   rs.Open "Insert into Table2 Select * from Table1", con, adOpenKeyset, adLockOptimistic
End Sub
```

Coding 243: Daten mit Insert into kopieren- SQL Server/Access

12.2.15. Löschen einer ganzen Tabelle

Das Löschen mit Delete entfernt nur die Daten, nicht aber die Tabelle. Diese kann mit dem Behfehl Drop gelöscht werden.

```
Sub Delete_Table()
   rs.Open „Drop table Table2", con, adOpenKeyset, adLockOptimistic
End Sub
```

Coding 244: Ganze Tabelle aus Datenbank löschen

12.2.16. Felder nachträglich einfügen

Für den SQL Server muss der im Beispiel 12.2.11 für ADO gezeigte Befehl nochmals angepasst werden:

```
Sub add_field_ado()
   With rs
         .Open "Alter Table Tabelle1 add Age Char(3)", con
   End With
End Sub
```

Coding 245: Nachträglich Felder in Tabelle einfügen

12.2.17. Einfachen Index erzeugen

Zur Vermeidung von doppelten Datensätzen macht ein Index Sinn.

```
Sub create_single_index()
   With rs
         .Open "Create Unique Index myindex on Table1 (Feld1)", con
   End With
End Sub
```

Coding 246: Einfacher Index

Mit dem Zusatz DESC nach dem Feldnamen wird eine absteigende Sortierreihenfolge erzielt.

12.2.18. Zusammengesetzen Index erzeugen

Der zusammengesetzte Index verhindert, dass Datensätze mit gleichem Inhalt aus mehreren Feldern doppelt gesichert werden können.

```
Sub create_multi_index()
   With rs
         .open „Create Unique Index myindex on Table1 (Feld1, Feld2)“, con
   End with
End sub
```

Coding 247: Zusammengesetzter Index

12.2.19. Index löschen

Ein Index wird, egal ob einfach oder zusammengesetzt, über den Befehl Drop gelöscht.

```
Sub del_index()
   With rs
         .open „Drop Index myindex on Table1“, con
   End with
End sub
```

Coding 248: Index löschen

12.2.20. Verbidnungsaufbau zu einem SQL Server (ADO-JET)

Neben den klassischen Access-Datenbanken lassen sich auch SQL Server und Co. Verwenden. Einen SQL Server kann man wie folgt ansprechen:

```
Sub init()
   Set con = New ADODB.Connection
   With con
         .CursorLocation = adUseClient
         .Open "Provider=sqloledb;Data Source=Composti;Initial Catalog=Master;” &_
         „User Id=SA;Password=Passwort"
   End With
End Sub
```

Coding 249: Verbindungsaufbau zu einem SQL Server

12.3. ADO Extension – was sonst nur DAO kann

12.3.1. Datenbank erstellen (ADOX)

Um mit ADOX arbeiten zu können, muss ein Verweis auf die Bibliothek „Microsoft ADO Ext." erstellt werden. Anschliessend kann mit folgendem Coding eine Datenbank erstellt werden:

```
Sub CreateDatabase()
   Dim Kat As New ADOX.Catalog
   Kat.Create "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\my.mdb"
End Sub
```

Coding 250: Datenbank mit ADOX erstellen

12.3.2. Tabellen erstellen (ADOX)

Eine Tabelle lässt sich wie folgt erstellen:

```
Sub make_table()
   Dim tbl As New Table
   Dim kat As New ADOX.Catalog

   kat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
         "Data Source=c:\my.mdb;"
   With tbl
         .Name = "newTable"
         .Columns.Append "Spalte1", adInteger
         .Columns.Append "Spalte2", adInteger
         .Columns.Append "Spalte3", adVarWChar, 50
   End with
   kat.Tables.Append tbl
   kat.ActiveConnection = Nothing
End Sub
```

Coding 251: Tabelle mit ADOX erstellen

12.3.3. Tabelle mit Index erstellen (ADOX)

Soll ein Index gelegt werden, so kann dies über folgendes Coding realisert werden:

```
Sub make_index()

   Dim tbl As New Table
   Dim idx As New ADOX.Index
   Dim kat As New ADOX.Catalog

   kat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
         "Data Source=c:\my.mdb;"
   Set tbl = kat.Tables("newTable")
   With idx
         .Name = "mehrspaltenidx"
         .Columns.Append "Spalte1"
         .Columns.Append "Spalte2"
   End with
   tbl.Indexes.Append idx
   kat.ActiveConnection = Nothing
End Sub
```

Coding 252: (Mehrspaltigen) Index auf Tabelle legen mit ADOX

12.3.4. Primärschlüssel erstellen (ADOX)

Ein Primärschlüssel kann ein oder mehrere Felder beinhalten. Der Primärschlüssel ist immer eindeutig – es können keine Datensätze mit dem gleichen Inhalt, für die dem Primärschlüssel zugehörigen Felder angefügt werden.

```
Public Sub Table_with_PKey()
   Dim tdefAs new ADOX.table
   Dim ikeyAs new ADOX.key
   Dim cat As new ADOX.Catalog
   With tdef
         .Name = Tablename
         Set .ParentCatalog = cat
         .Columns.Append "Buchungsdatum", adDate
```

```
            .Columns.Append "Saldo", adDouble, 14
            .Columns.Append "Text1", adWChar, 150
            .Columns.Append "Text2", adWChar, 200
            .Columns.Append "Betrag", adDouble, 14
      End With
      With ikey
            .Name = "Primary"
            .Type = adKeyPrimary
            .Columns.Append "Buchungsdatum", adDate
            .Columns.Append "Saldo", adDouble, 14
      End With
      cat.Tables.Append tdef
      cat.Tables.Item(Tablename).Keys.Append ikey
End Sub
```

Coding 253: Tabelle mit Primary Key erstellen

12.3.5. Fremdschlüssel erstellen (ADOX)

Um einen Fremdschlüssel zu erstellen kann folgendes Coding verwendet werden:

```
Sub make_Key()

   Dim f_key As New ADOX.Key
   Dim kat   As New ADOX.Catalog

   kat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
         "Data Source=c:\my.mdb;"
   With f_key
         .Name = "CustOrder"
         .Type = adKeyForeign
         .RelatedTable = "DIM_MONAT"
         .Columns.Append "Monat"
         .Columns("Monat").RelatedColumn = "Monat"
         .UpdateRule = adRICascade
   End With
   kat.Tables("newTable").Keys.Append f_key
End Sub
```

Coding 254: Fremschlüssel erstellen mit ADOX

12.4. Mit ADO Daten aus einem Excelsheet auslesen

Mit dieser Methode lassen sich Exceldaten auslesen, ohne dass das Excelsheet geöffnet werden muss.

```
Dim cn as ADODB.Connection
Dim rs as ADODB.Recordset

Sub connect_to_excel()
   Set cn = New ADODB.Connection
   Set rs = New ADODB.Recordset
   With cn
         .Provider = "Microsoft.Jet.OLEDB.4.0"
         .ConnectionString = "Data Source=C:\MyFolder\MyWorkbook.xls;" & _
               "Extended Properties=Excel 8.0;"
         .Open
   End With
   Rs.open „Select * from [Tabelle1$A1:B2]", cn, adOpenKeyset, adLockOptimistic
   Rs.movefirst
   Rs.close
   Cn.close
End sub
```

Coding 255: Connect zu einer Exceldatei

12.5. ODBC – Verbindungen

Zum Einstieg altbewährtes – Excelsheet auslesen:

```
Dim con As ADODB.Connection
Dim rs  As ADODB.Recordset

Sub test()
   Set con = New ADODB.Connection
   Set rs = New ADODB.Recordset
   With con
         .Provider = "MSDASQL"
         .ConnectionString = "DRIVER={Microsoft Excel Driver (*.xls)};" & _
               "DBQ=G:\Mappe2.xls"
         .Open
   End With
   With rs
         .Open "Select * from [Tabelle1$A1:B2]", con, adOpenKeyset, adLockOptimistic
         .MoveFirst
   End With
End Sub
```

Coding 256: ODBC Verbindung – Excelsheet auslesen

12.5.1. Weitere ODBC Treiber

ODBC Treiber
Microsoft Access Driver (*.mdb)
Microsoft Excel Driver (*.xls)
Microsoft dBase Driver (*.dbf)
Microsoft FoxPro Driver (*.dbf)
Microsoft Paradox Driver (*.db)
Microsoft Text Driver (*.txt;*.csv)
SQL Server
Microsoft ODBC for Oracle

Tabelle 25: Mögliche ODBC Treiber

Im Einzelnen muss der jeweilige Rechner daraufhin überprüft werden, welcher Treiber dort installiert ist.

12.5.2. Providerstrings

Bei einer OLE DB Verbindung werden für den Connect die Providernamen benötigt.

Datenbank-Typ	Providerstring
Microsoft Access	Provider=Microsoft.Jet.OLEDB.4.0;Data Source=<Dateiname mit Pfad>
Microsoft Excel	Provider=Microsoft.Jet.OLEDB.4.0;Data Source=<Dateiname mit Pfad>
Oracle Datenbank	Provider=MSDAORA;Data Source=<Dateiname mit Pfad>
Paradox	Provider=MSDASQL;Data Source=<Dateiname mit Pfad>
Microsoft Visual FoxPro	Provider=MSDASQL;Data Source=<Dateiname mit Pfad>
Microsoft SQL Server	Provider=MSDASQL;Data Source=<Dateiname mit Pfad>

Tabelle 26: Providerstrings

12.5.3. Connectionstrings

Bei einer ODBC Verbindung wird der Connectionstring benötigt. Nachfolgende Liste zeigt einige auf, wobei die genauen Parameter der Beschreibung des jeweilig aktuellen Treibers zu entnehmen sind:

Datenbank-Typ	Connectionstring
Microsoft Access	ConnectionString = "DRIVER={Microsoft Access (*.xls)};Data Source=<Dateiname mit Pfad>"
Microsoft Excel	ConnectionString = "DRIVER={Microsoft Excel (*.xls)};Data Source=<Dateiname mit Pfad>;Extended Properties=Excel 8.0;"
Oracle Datenbank	DRIVER={Microsoft ODBC for Oracle};SERVER=<Dateiname mit Pfad>
Paradox	Driver={Microsoft Paradox Driver (*.db)};DBQ=<Dateiname mit Pfad>;DriverID=26
Microsoft Visual FoxPro	Driver={Microsoft Visual FoxPro Driver};SourceType=DBC;SourceDb=<Dateiname mit Pfad>
Microsoft SQL Server	DRIVER={SQL Server};SERVER=<Dateiname mit Pfad>

Tabelle 27: Connectionstrings

12.5.4. DNS Verbindung

Unter Data Source kann statt dem Dateinamen (mit kompletten Pfad) auch eine DNS angegeben werden. Diese muss vorher im System angelegt werden.

Hierfür wird in der Systemsteuerung unter Verwaltung die ODBC-Datenquellen-Administrator gestartet:

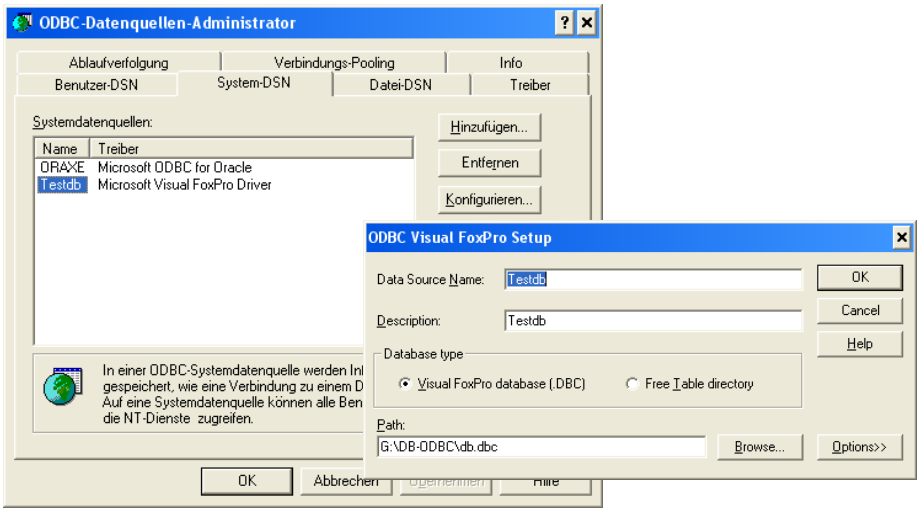


Abbildung 31: ODBC –Datenquellen-Administrator

Die Verbindung wird je nach Treiber unterschiedlich ausgeprägt. Der Aufruf im Coding geschiet dann wie folgt:

```
Dim con As ADODB.connection
Dim rs  As ADODB.Recordset

Sub dbopen()
   Set con = New ADODB.connection
   Set rs = New ADODB.Recordset
   With con
         .ConnectionString = "Data Source=Testdb"
         .Open
   End With
   With rs
         .Open "Select * from Person", con, adOpenKeyset, adLockOptimistic
   End With
   con.Close
   Set con = Nothing
End Sub
```

Coding 257: ODBC Verbindung über DNS

12.5.5. Connectionstring für Visual FoxPro

Unterschiedliche Datenbanken benötigen unterschiedliche Parameter. Hier ein Beispiel, wie eine Visual FoxPro Datenbank angesprochen werden kann:

```
Sub dbopen()
   Dim con As ADODB.connection
   Dim rs  As ADODB.Recordset

   Set con = New ADODB.connection
   Set rs = New ADODB.Recordset
   With con
         .ConnectionString = "DRIVER={Microsoft Visual FoxPro-Treiber};" &_
               SourceType=DBC;SourceDb=g:\db-odbc\db.dbc"
         .Open
   End With
   With rs
         .Open "Select * from Person", con, adOpenKeyset, adLockOptimistic
   End With
   con.Close
   Set con = Nothing
End Sub
```

Coding 258: ODBC Connect für Visual FoxPro Datenbank

12.5.6. SQL Server via ODBC

Nun ein kleines Beispiel für eine Verbindung zu einem SQL Server.

```
Dim con As ADODB.Connection
Dim rs  As ADODB.Recordset

Sub test()
   Set con = New ADODB.Connection
   Set rs = New ADODB.Recordset
   With con
         .Provider = "MSDASQL"
         .ConnectionString = "DRIVER={SQL Server};SERVER=SYSCON-MOBIL;" & _
               „Database=Master;UID=;pwd=;"
         .Open
   End With
   With rs
         .Open "Select * from Table1", con, adOpenKeyset, adLockOptimistic
         .AddNew
               .Fields("NAME").Value = "Peter"
               .Fields("AGE").Value = "36"
         .Update
         .close
   End With
End Sub
```

Coding 259: SQL Server per ODBC Verbindung öffnen

12.6. Verbindung zu einer MySQL Datenbank

12.6.1. Der Connectionstring

Hier wieder ein Beispiel für eine ODBC Verbindung. Ohne Angabe des Servers wird diese am Localhost vermutet.

```
Sub open_DB()
Dim con As New ADODB.Connection
With con
.Provider = "MSDASQL"
.ConnectionString = "DRIVER={MySQL ODBC 5.1 Driver};" & _
"Database=Home;UID=root;pwd=Crusader;"
.Open
End With
End sub
```

Coding 260: ODBC Verbindung zu einer MySQL Datenbank

12.6.2. Datenbank unter MySQL erstellen

Um unter MySQL eine Datenbank zu erstellen wird das Connection-Objekt mit der Methode „Execute“ aufgerufen und der entsprechende SQL Befehl als String übergeben. Das Connection-Objekt wurde vorher natürlich ohne Datenbank instanziert.

```
Sub CreateDB()
Dim con As New ADODB.Connection
With con
.Provider = "MSDASQL"
.ConnectionString = "DRIVER={MySQL ODBC 5.1 Driver};" & _
"UID=root;pwd=Crusader;"
.Open
con.Execute "Create Database Home"
End With
End Sub
```

Coding 261: Erstellen einer Datenbank unter MySQL

12.6.3. Tabellen unter MySQL erstellen

Beim Anlegen neuer Tabellen unter MySQL muss die Anmeldung wie in 12.6.1 gezeigt, an der Datenbank erfolgen. Anschliessend wird ebenfalls mit der Execute-Methode ein SQL Befehl abgesetzt.

```
Sub CreateTable()
With con
.Provider = "MSDASQL"
.ConnectionString = "DRIVER={MySQL ODBC 5.1 Driver};" & _
"Database=Home;UID=root;pwd=Crusader;"
.Open
con.Execute "Create Table Pers(pname CHAR(25) NOT NULL, age SMALLINT)"
End With
End Sub
```

Coding 262: Erstellen einer Tabelle unter MySQL

12.6.4. Daten in eine Tabelle schreiben

Das Schreiben von Daten funktioniert ähnlich dem bei JET Datenbanken. Ein Recordset wird instanziert, einer Tabellenstruktur zugewiesen und mit der Methode „AddNew“ befüllt. Über die Methode „Update“ werden die Daten auf die Datenbank geschrieben.

```
Sub WriteData()
Dim rs As New ADODB.Recordset
With rs
.Open "Select single * from Pers", con, adOpenStatic, adLockOptimistic
.AddNew
.Fields("pname").Value = "Peter"
.Fields("age").Value = 37
.Update
End With
End sub
```

Coding 263: Daten in Tabelle schreiben - MySQL

12.7. Das Datengrid

12.7.1. Unbound Recordset

Für die Verwendung von OCX Komponenten, wie z.B. dem Datengrid muss auf dem Rechner, mit dem programmiert wird, eine entsprechende Entwicklerlizenz vorhanden sein. Dies ist zum Beispiel der Fall, wenn das kostenlose VB 5.0 CCE installiert wurde.
Für folgendes Beispiel wird eine Userform mit einem MS-DATAGRID bestück und anschliessend mit einem unbound ADO Recordset befüllt:

```
Private Sub UserForm_Initialize()
   Dim rsAs ADODB.Recordset
   Set rs = New ADODB.Recordset
   With rs
         .Fields.Append "Name", adChar, 25
         .Fields.Append "Alter", adInteger, 3
         .Open
         .AddNew
               .Fields.Item("Name").Value = "Peter"
               .Fields.Item("Alter").Value = 36
         .Update
         .AddNew
               .Fields.Item("Name").Value = "Christina"
               .Fields.Item("Alter").Value = 34
         .Update
         .MoveFirst
   End With
   Set DataGrid1.DataSource = rs
   grid.Refresh
End Sub
```

Coding 264: Anbindung eines ADO Recordset an ein Datengrid

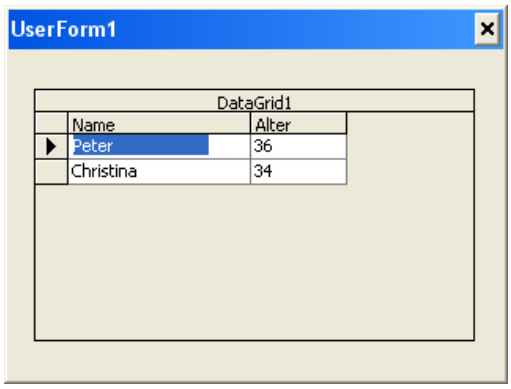


Abbildung 32: Userform mit gefülltem Datengrid

Alle Änderungen der Daten im Grid werden auch im Recordset geändert.

12.7.2. Bound Recordset

Im nachfolgenden Beispiel wird eine Verkaufstabelle ausgewertet. Die Felder der Tabelle „Verkäufe“ sind „Kunden_ID“, „Produkt_ID“ und „Menge“. Kunden_ID und Produkt_ID werden gruppiert, die Menge wird summiert. Für die Ausgabe wird eine Userform mit einem Datengrid benötigt.

```
Dim con As ADODB.Connection
Dim rs  As ADODB.Recordset

Sub test()
   Set con = New ADODB.Connection
   Set rs = New ADODB.Recordset
   With con
         .Provider = "MSDASQL"
         .ConnectionString = "DRIVER={SQL Server};SERVER=SYSCON-MOBIL;" & _
               "Database=Master;UID=;pwd=;"
         .Open
   End With
   With rs
         .Open "Select kunden_id, produkt_id, sum(menge) from Verkäufe " & _
               "group by kunden_id, produkt_id", con, adOpenKeyset, adLockOptimistic
   End With
   Set UserForm1.DataGrid.DataSource = rs
   UserForm1.Show
   con.Close
End Sub
```

Coding 265: Datengrid mit Bound Recordset

Das Ergebnis müsste nun wie folgt aussehen:

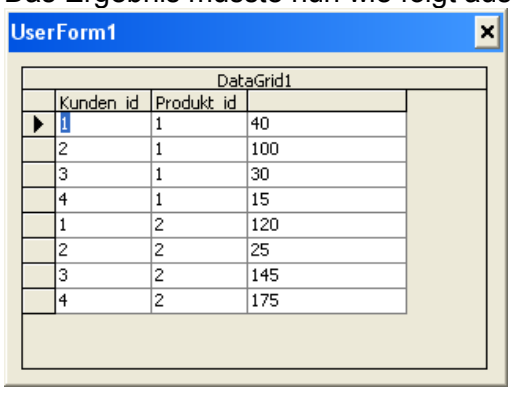


Abbildung 33: Datagrid mit Bound Recordset

Änderungen an den Datensätzen können über das Datengrid nicht vorgenommen werden.

13. VBE

Über die Visual Basic Engine lassen sich einige Aktionen zur Laufzeit realisieren, die sonst das Eingreifen des Programmierers erfordern.

13.1. Aufruf des Codefensters

Um ein Codefenster anzuzeigen muss normal erst der VisualBasic Editor gestartet werden, dann das korrekte Projekt und anschliessend das jeweilige Modul geöffnet werden. Folgende Routine erledigt dass ganz automatisch:

```
Sub Code_anzeigen()
   Application.VBE.ActiveCodePane.Show
End Sub
```

Coding 266: Codefenster öffnen

13.2. Einfügen eines Verweises zur Laufzeit

Ein Verweis kann auch zur Laufzeit angelegt werden. Das kann zur Verbesserung der Performance, aber auch als Schnittstelle genutzt werden.

```
Sub referenz_zur_laufzeit()
   Const datei = "e:\programme\gemeinsame dateien\microsoft shared\dao\dao360.dll"
   Application.VBE.ActiveVBProject.References.AddFromfile datei
End Sub
```

Coding 267: Verweis zur Laufzeit anlegen

Alternativ kann dies auch lokal, über das aktive Workbook ausgeführt werden:

```
Sub referenz_zur_laufzeit()
   Const datei = "e:\programme\gemeinsame dateien\microsoft shared\dao\dao360.dll"
   ThisWorkbook.VBProject.References.AddFromFile datei
End Sub
```

Coding 268: Lokalen Verweis zur Laufzeit anlegen

13.3. Code in laufendes Projekt einfügen

Mit der VBE kann auch der Code eines Moduls zur Laufzeit verändert werden. Dabei muss man allerdings die exakte Zeile angeben können, wo der Code eingebaut werden soll.

```
Sub Code_in_Modul_einfügen()
   Application.VBE.ActiveCodePane.CodeModule.ReplaceLine 1, "'Kommentar"
End Sub
```

Coding 269: Code zur Laufzeit einfügen

13.3.1. Code in andere VB-Komponente einfügen

Um den Code an einer anderen Stelle, beispielsweise in „DieseArbeitsmappe“ einzufügen muss die VBE wie folgt adressiert werden:

```
Dim myline(5) As String
Dim x

Sub set_myline()
   myline(1) = "Private Sub Workbook_Open()"
   myline(2) = "   msgbox prompt:=" & Chr$(34) & "Hallo" _
   & Chr$(34) & " & application.UserName"
   myline(3) = "end sub"
End Sub

Sub test_vbe()
   set_myline
   With Application.VBE.ActiveVBProject.VBComponents
         For x = 1 To 3
               .Item("DieseArbeitsmappe").CodeModule.InsertLines (25 + x), myline(x)
   Next x
```

```
   End With
End Sub
```

Coding 270: Code zu Laufzeit in ThisWorkbook einfügen

13.4. VBIDE-Objekte anfügen

Zur Laufzeit können ebenfalls VBIDE-Objekte, wie Module, Klassen ect. angefügt werden. Vor Verwendung muss für das laufende Projekt ein Verweis auf die Bibliothek „Microsoft Visual Basic for Applications Extensibility“ gelegt werden.

```
Sub Modul_anfügen()
   Application.VBE.ActiveVBProject.VBComponents.Add vbext_ct_StdModule
End Sub
```

Coding 271: Anfügen eines Moduls

Objekt	Beschreibung
vbext_ct_ClassModul	Klasse
vbext_ct_StdModul	Modul
vbext_ct_MSForm	Form

Tabelle 28: VBIDE Objekte

14. Windows API

Die Windows API ermöglicht, per VBA Coding auf Windows Funktionen zu zugreifen. Hierdurch sind Abfragen nach dem Windows-User, dem Windows-Pfad udgl. möglich.

14.1. Windows-User abfragen

Mit folgendem Coding kann der Name des am Windows-System angemeldeten Users ermittelt werden:

```
Declare Function GetUserName Lib "advapi32.dll" _
   Alias "GetUserNameA" (ByVal lpBuffer As String, _
   nSize As Long) As Long

Public Function GetBenutzer() As String
   Dim UserName As String
   Dim Result As Long

   UserName = Space$(256)
   Result = GetUserName(UserName, Len(UserName))

   If InStr(UserName, Chr$(0)) > 0 Then _
   UserName = Left$(UserName, InStr(UserName, Chr$(0)) - 1)

   GetBenutzer = UserName
End Function
```

Coding 272: Windows User abfragen

14.2. Windows-Pfad ermitteln

Zum Lesen und Schreiben von Daten wird vielfach der Pfad des Windows-Systems benötigt:

```
Private Declare Function GetWindowsDirectory Lib "kernel32" Alias _
"GetWindowsDirectoryA" (ByVal lpBuffer As String, _
ByVal nSize As Long) As Long

Public Function Get_WindowsDir() As String
   Dim Temp As String
   Dim lResult As Integer

   Temp = Space$(256)
   lResult = GetWindowsDirectory(Temp, Len(Temp))
   Temp = Left$(Temp, lResult)
   If Right$(Temp, 1) <> "\" Then Temp = Temp + "\"
   Get_WindowsDir = Temp
End Function
```

Coding 273: Windows Pfad ermitteln

14.3. Ein Tempfile erzeugen

Um temporäre Daten abzuspeichern wird unter Windows mit TMP-Files gearbeitet. Um diese am korrekten Platz zu speichern eignet sich nachfolgendes Coding. Der Funktion können dabei noch 2 Buchstaben un eine Zahl mitgegeben werden.

```
Private Declare Function GetTempFileName Lib "kernel32" Alias _
   "GetTempFileNameA" (ByVal lpszPath As String, _
   ByVal lpPrefixString As String, ByVal wUnique As Long, _
   ByVal lpTempFileName As String) As Long

Private Declare Function GetTempPath Lib "kernel32" Alias _
   "GetTempPathA" (ByVal nBufferLength As Long, ByVal _
   lpBuffer As String) As Long
```

```
Public Function set_Tempfile(fName As String, Ident As Integer) As String
   Dim p, d
   p = Space(260)
   d = Space(260)
   GetTempPath 255, d
   GetTempFileName d, "$" & fName, Ident, p
   set_Tempfile = p
End Function
```

Coding 274: Tempfile erzeugen

14.4. Programmabbruch mit Logfile-Eintrag

Für schwerwiegende Ausnahmen kann mit dem folgenden Coding ein Programmabbruch erzeugt werden. Dieser wird im Eventlog protokolliert.

```
Private Declare Sub FatalAppExit Lib "kernel32" Alias "FatalAppExitA" _
   (ByVal uAction As Long, ByVal lpMessageText As String)

Public Sub app_error_exit()
   FatalAppExit 0, "Fehler! Programm wird abgebrochen!"
End Sub
```

Coding 275: Programmabbruch mit Eventlogeintrag

14.5. Windows herunterfahren

Um das Betriebssystem herunter zu fahren, den User abzumelden oder einen Reboot durch zu führen hilft nachfolgendes Coding:

```
Private Declare Function ExitWindowsEx Lib "user32" _
   (ByVal uFlags As Long, ByVal dwReserved As Long) As Long

Public Enum dowin
   Logof = 0
   Shutdown = 1
   Reboot = 2
   Force = 4
End Enum

Public Sub exit_windows()
   Dim x
   x = ExitWindowsEx(dowin.Shutdown, 0)
End Sub
```

Coding 276: Windows herunterfahren

14.6. Herunterfahren verhindern

Um einen Systemneustart, bzw. ein Herunterfahren von Windows zu verhindern wird folgendes Coding benötigt:

```
Private Declare Function AbortSystemShutdown Lib _
   "advapi32.dll" Alias "AbortSystemShutdownA" _
   (ByVal lpMachineName As String) As Long

Private Declare Function GetComputerName Lib _
   "kernel32" Alias "GetComputerNameA" _
   (ByVal lpBuffer As String, nSize As Long) As Long

Public Sub abortshutdown()
   Dim x
   Dim strString As String
   strString = String(255, Chr$(0))
   GetComputerName strString, 255
   strString = Left$(strString, InStr(1, strString, Chr$(0)))
   x = AbortSystemShutdown(strString)
End Sub
```

Coding 277: Herunterfahren verhindern

14.7. MIDI File abspielen

Um ein wenig Musik abzuspielen kann folgende Funktion verwendet werden:

```
Private Declare Function mciSendString Lib "winmm.dll" Alias "mciSendStringA" _
   (ByVal lpstrCommand As String, ByVal lpstrReturnString As Any, _
   ByVal uReturnLength As Long, ByVal hwndCallback As Long) As Long

Dim ret, file

Private Sub play_midi(ByVal song As String, ByVal foreground As Boolean)
   ret = mciSendString("open " & song & " type sequencer alias canyon", 0&, 0, 0)
   'bei foreground = true wird die Verarbeitung erst nach Beendigung des Songs fortgesetzt
   If foreground = True Then
         ret = mciSendString("play canyon wait", 0&, 0, 0)
   Else
         ret = mciSendString("play canyon", 0&, 0, 0)
   End If
End Sub

Sub stop_midi()
   ret = mciSendString("close canyon", 0&, 0, 0)
End Sub

Sub start_midi()
   file = Application.GetOpenFilename("Midi (*.mid), *.mid", , "Musik öffnen", , False)
   play_midi file, False
End Sub
```

Coding 278: Abspielen von MIDI-Files

14.8. Code als String übergeben und ausführen

Mit nachfolgender API Funktion ist es möglich, einen Code, welcher als String vorliegt, direkt ausführen zu lassen. Üblicherweise werden damit etwas weniger destruktive Aktionen ausgeführt:

```
Private Declare Function EbExecuteLine Lib "vba332.dll" _
(ByVal StringToExec As Long, ByVal Any1 As Long, _
ByVal Any2 As Long, ByVal CheckOnly As Long) As Long

Function ExecuteCode(Code As String, Optional CheckOnly As Boolean) As Boolean
   ExecuteCode = EbExecuteLine(StrPtr(Code), 0&, 0&, CheckOnly) = 0
End Function

Sub test()
    ExecuteCode "kill c:\test.ini", False
End Sub
```

Coding 279: ExecuteCode: Code als String

15. Kommunikation mit SAP R/3

15.1. Kommunikationsaufbau

Zur Herstellung der Verbindung werden generell einige Objekte benötigt. Die Verbindung selbst kann dann aber auf zwei Arten erfolgen. Einmal mit Systemauswahl und Passwortabfrage und einmal mit dem sogenannten Silent-Login, bei dem keine Parameter abgefragt werden.

15.1.1. Der User-Login

Hierbei erfolgt vor Aufbau der Verbindung eine System- und Passwortabfrage. Der User erhält dabei das SAP Logon angezeigt, wählt ein System aus und gibt seinen User und sein Passwort in das Abfragefenster ein.

```
Global connect, func As Object
Global result As Boolean
Global myfunc As Object

Private Sub anmelden()
   Set func = CreateObject("SAP.Functions")
   Set connect = func.Connection

   If func.Connection.logon <> True Then
         func.RemoveAll
         MsgBox prompt:="Anmeldung fehlgeschlagen! Bitte verständigen Sie den _
                     Administrator (Datei wdtfuncs.ocx registriert?)!"
         Exit Sub
   End If
End sub
```

Coding 280: Anmelderoutine mit Userabfrage

Hierzu gibt es noch die Variante über das SAP Logon-Pad:

```
Dim log        As Object
Dim con        As Object

Sub anmelden()
   Set log = CreateObject("SAP.Logoncontrol.1")
   Set con = log.newconnection

   If con.logon(0, False) = False Then
         MsgBox prompt:="Anmeldung fehlgeschlagen!"
   End If
End Sub
```

Coding 281: Anmelderoutine über SAP Logon-Pad

15.1.2. Der Silent-Login

Beim Silent Login erfolgt die Anmeldung am SAP anhand von im Coding hinterlegten Anmeldedaten. Dabei wird dem User keinerlei Auswahlmöglichkeit gelassen, an welchem System er sich anmelden kann. Deshalb und da die Parameter – incl. Passwort – fest im Coding hinterlegt sind, eignet sich diese Methode eher für Systemanbindungen.

```
Sub anmelden()
   Set func = CreateObject("SAP.Functions")
   Set connect = func.Connection

   connect.client = "020"
   connect.user = "SAP*"
   connect.Password = "PASS"
   connect.Language = "DE"

   If func.connection.logon = true then
         func.RemoveAll
         MsgBox prompt:="Anmeldung fehlgeschlagen! Bitte verständigen Sie den_
                     Administrator (Datei wdtfuncs.ocx registriert?)!"
         Exit Sub
   End If
```

```
End Sub
```

Coding 282: Anmelderoutine Silent-Login

Die Anmeldesprache ist optional und macht natürlich nur bei Dialoganwendungen Sinn.

15.2. Aufruf von SAP Funktionsbausteinen

15.2.1. Aufruf eines RFC fähigen Funktionsbausteines

In der Funktion wird ein Objekt mitgegeben und die Variablen für die Parameterübergabe deklariert. Diese sollten, wo es möglich ist, entsprechend in Form gebracht werden (Alphakonvertierung). Anschliessend wird der Funktionsaufruf durchgeführt und die Ergebnisse abgefangen.

Hier wird ein kundendefinierter Funktionsbaustein angesprochen. Er liest Werte aus dem Feld „manueller Vermögenswert“ des Anlagestammsatzes.

```
Function get_verswert(Anlage As String) As String
   On Error GoTo errhandler
   Set myfunc1 = func.Add("YF_CTA_AA_STAMMDATEN")

   Dim mwerta As Variant
   Dim errcode As Integer
   Dim ergebnis As String
   Anlage = Format(Anlage, "000000000000")

   myfunc1.exports("anlage") = Anlage

   result = myfunc1.Call
   errcode = myfunc1.imports("OKCODE")
   mwerta = myfunc1.imports("E_MWERTA")
   GoTo ende

errhandler:
   If result = False Then Call anmelden

ende:
   If errcode = 0 Then ergebnis = mwertg _
         Else: ergebnis = "Es ist ein Fehler aufgetreten!"
   get_verswert = ergebnis
End Function
```

Coding 283: Funktionsaufruf an SAP

Der Funktionsbaustein übergibt zusätzlich zu den selektierten Daten einen Fehlercode. Statt eine Exception auszuwerten wird nur auf den Fehlercode geschaut. Dies ist aufgrund der Tatsache, dass man eh nur wissen will ob Daten gelesen wurden oder nicht, wesentlich einfacher.

Im ersten Beispiel wurden nun einzelne Werte übergeben. Es können aber auch komplette Tabellen übergeben werden. Im nächsten Beispiel werden alle Konten eines Buchungskreises zusammen mit dem Kontentext übergeben. Dabei ist das Übergabeelement ein Objekt, dass die Tabelle „ET_STXT“ enthält. Der Rowcount wird ausgelesen und die Daten an ein Array mit eigenem Datentypen übergeben.

Nachstehend wird abermals ein Kundenbaustein verwenden, der die Tabelle SKAT ausliest und die Daten als Tabelle an Excel zurück gibt.

```
Private type strc_sako
   Konto                  as string
   Bezei                  as string
End type

Dim Sako(1 to 10000)      as object
Dim myfunc                as object
Dim myresult              as object

Sub lese_sakonten()
   Set myfunc = func.Add("YGTFXX_SAKO_SKAT")

   Dim BUKRS As String * 4
   Dim SPRAS As String * 2
```

```
   BUKRS = mod_allgfunc.get_bukrs
   SPRAS = "DE"

   If BUKRS = "" Then
         MsgBox prompt:="Bitte selektieren Sie erst einen Buchungskreis!"
         GoTo ende
   End If

   Dim Zeilen As Integer

   new_status "Verbinde mit SAP..."
   myfunc.exports("I_BUKRS") = BUKRS
   myfunc.exports("I_SPRAS") = SPRAS

   result = myfunc.Call
   Set myresult = myfunc.tables.Item("ET_STXT")

   Zeilen = myresult.RowCount

   uf_sako.lb_Konto.Clear

   new_status "Übergebe Konten an Listfeld..."
   For y2 = 1 To Zeilen
         SAKO(y2).Konto = myresult(y2, "SAKNR")
         SAKO(y2).Bezei = myresult(y2, "TXT50")
   Next y2
End Sub
```

Coding 284: Übergabe von ganzen Tabellen

15.2.2. Abmelden vom SAP R/3 System

Bei diesem Vorgang wird die Verbindung zum R/3 System gekappt. Dies zu programmieren ist eigentlich nicht notwendig, da die Verbindung mit dem Beenden von Excel sowie so beendet wird, soll aber der Vollständigkeit halber auch mit aufgeführt sein:

```
Sub abmelden()
   Set func = nothing
   Set connect = nothing
End sub
```

Coding 285: Abmelden vom SAP System

15.2.3. Tabellenübergabe von SAP

Nun kann es vorkommen, dass der RFC Funktionsbaustein nicht einzelne Werte, sondern komplette Tabellen zurückgibt. Hierfür muss der Datenempfang etwas modifiziert werden. Als Empfängerelement dient nun nicht mehr eine einfache Variable, sondern Objektvariablen.

Beispiel für das Auslesen einer Tabelle über den BAPI "RFC_READ_TABLE":

```
Global connect, func As Object
Global result As Boolean
Global myfunc As Object
Global data As Object
Global fields As Object

Private Sub anmelden()
   Set func = CreateObject("SAP.Functions")
   Set connect = func.Connection

   If func.Connection.logon <> True Then
         func.RemoveAll
               MsgBox prompt:="Anmeldung fehlgeschlagen! Bitte verständigen Sie den_
                     Administrator (Datei wdtfuncs.ocx registriert?)!"
         Exit Sub
   End If
End Sub
```

Coding 286: BAPI "RFC_READ_TABLE": Variablendeklaration und Anmelderoutine

```
Public Sub daten_lesen()
   call anmelden
   On Error GoTo errhandler
   Set myfunc = func.Add("RFC_READ_TABLE")

   myfunc.exports("QUERY_TABLE") = "T003"
   myfunc.exports("DELIMITER") = ","

   result = myfunc.Call
   Set fields = myfunc.tables.Item("FIELDS")
   Set data = myfunc.tables.Item("DATA")
   GoTo ende
errhandler:
   msgbox prompt:="Bei der Verarbeitung ist ein Fehler aufgetreten!"
ende:
   call create_table
End Sub
```

Coding 287: BAPI "RFC_READ_TABLE": Leseroutine für Bapi

Die Daten liegen nun in den Objekten "fields" und "data". Diese müssen nun lediglich abgeloopt werden – die entsprechenden Count-Werte können dem Objekt direkt entnommen werden.

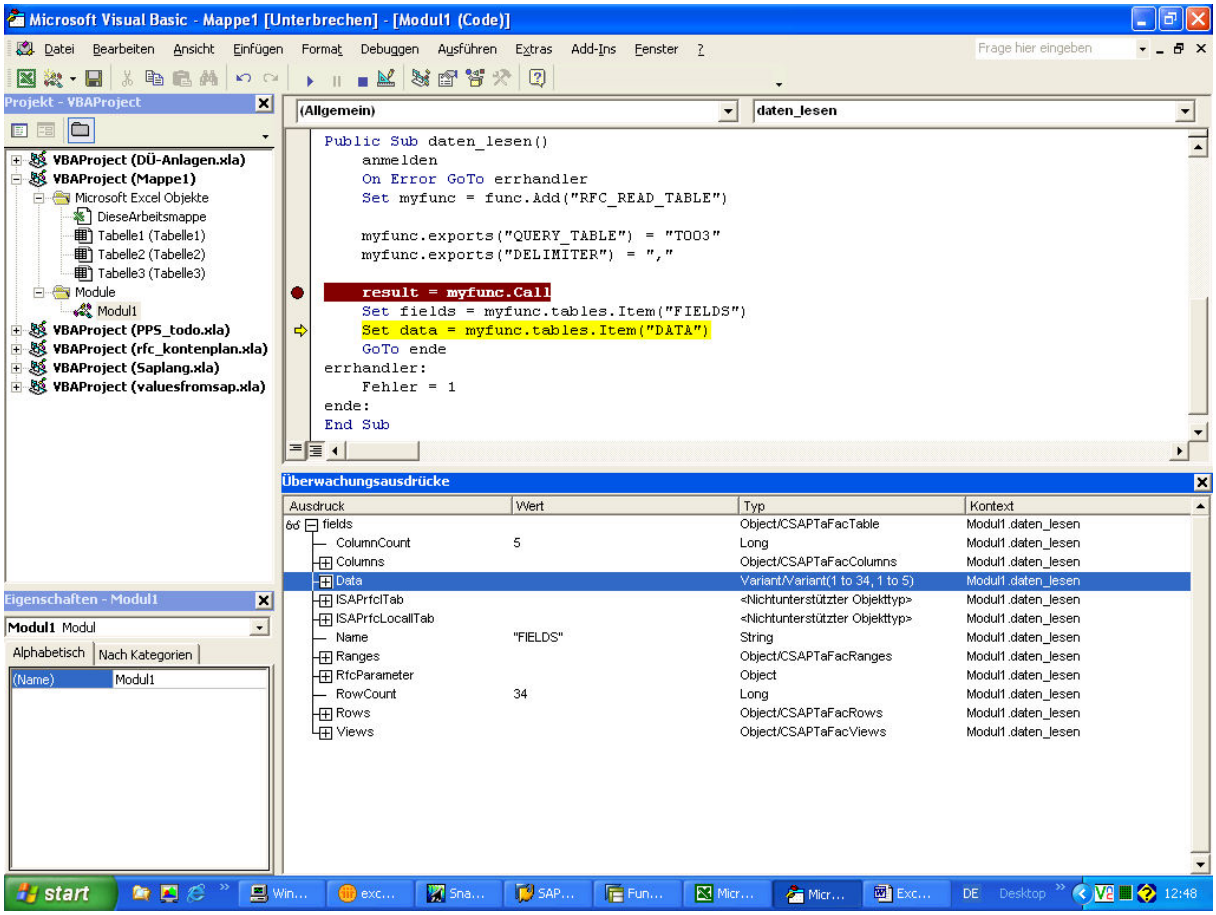


Abbildung 34: Aufbau des "field" Objektes

Die Tabelle enthält 5 Spalten und 34 Zeilen. Jede Zeile definiert sich in die Spalten Feldnamen, den Offset, der Feldlänge, dem Feldtyp und dem Feldtext.

Mit diesen Angaben lässt sich prima ein ADO Recordset aufbauen:

```
Public mytab As ADODB.Recordset
Dim fname As String
Dim flang As Integer
Dim ftype As String
Dim sa   As Long
Dim cc   As Long

Public Sub create_table()
   Set mytab = New ADODB.Recordset
   For sa = 1 To fields.RowCount
         fname = fields(sa, 1)
         flang = fields(sa, 3)
         With mytab
               Select Case fields(sa, 4)
                     Case "C"
                           .fields.Append fname, adChar, flang
                     Case "N"
                           .fields.Append fname, adNumeric, flang
                     Case "D"
                           .fields.Append fname, adDate, flang
                     Case "P"
                           .fields.Append fname, adDouble, flang
                     End Select
         End With
   Next sa
   mytab.Open
   For sa = 1 To data.RowCount
         For cc = 0 To (UBound(Split(data(1, 1), ",")))
               With mytab
                     .AddNew
                           .fields.Item(cc).Value = Split(data(sa, 1), ",")(cc)
                     .Update
               End With
         Next cc
   Next sa
End Sub
```

Coding 288: Aufbau einer ADO-Tabelle

Und zu guter Letzt noch abmelden:

```
Private Sub Abmelden()
   Set func = Nothing
   Set connect = Nothing
End Sub
```

Coding 289: Abmelden vom BAPI

15.2.4. Tabellenübergabe von Excel

Umgekehrt kann nun auch eine Tabelle von Excel nach SAP übergeben werden. Dabei wird erst das Table-Item-Objekt instanziert und anschliessend die Daten, zeilenweise über ein Row.Add Objekt in das Table-Item-Objekt geschrieben.

```
Private Sub write_table()
   With myfunc
         .exports("CO_AREA") = fix_kokrs
         .exports("POSTGDATE") = "01." & fix_perio & "." & fix_gjahr
         .exports("DOC_HDR_TX") = fix_text
         .exports("VAL_FISYEAR") = fix_gjahr
         .exports("VAL_PERIOD") = fix_perio
         'Tabellenobjekt instanzieren
         Set extab = .tables.Item("I_POS")
         'über Anzahl der Datensätze
         For x = 1 To count
               'Anfügen des Datensatzes
               Set exline = extab.Rows.Add
               exline(5) = pos(x).cc
               exline(1) = pos(x).skf
               exline(2) = pos(x).qua
               exline(4) = pos(x).txt
         Next x
         result = .call
         Set myresult = .tables.Item("RETURN")
```

```
   End With
End sub
```

Coding 290: Tabellen anfügen

15.3. Umsetzung weiterer RFC Zugriffe

15.3.1. VBA Klasse für SAP RFC Connection

Da der Verbindungsaufbau zu SAP immer gleich aufgebaut ist, soll er nun in der Klasse „sap_connect“ gekapselt werden. Auf die Object-Variable „fns“ muss später, im Modul verwiesen werden können, daher muss diese auf Public gesetzt werden.

```
Dim conn  As Object
Public fns As Object

Public Sub Anmelden()
   Set fns = CreateObject("SAP.Functions")
   Set conn = fns.Connection
   If fns.Connection.logon <> True Then
         fns.RemoveAll
         MsgBox prompt:="Anmeldung fehlgeschlagen! Bitte verständigen Sie den „ & _
                        „Administrator (Datei wdtfuncs.ocx registriert?)!"
         Exit Sub
   End If
End Sub

Public Sub Abmelden()
   Set fns = Nothing
   Set conn = Nothing
End Sub
```

Coding 291: VBA Klasse für SAP RFC Connection

15.3.2. Meldungen versenden mit TH_POPUP

Der Baustein TH_POPUP versendet SAPintern Meldungen. Dem Baustein werden Username und Meldungstext übergeben und der Funktionsbaustein versendet diese.
Zur Umsetzung muss nun neben der o.g. Klasse noch ein Modul angelegt werden:

```
Dim con As sap_connect
Dim myfunc  As Object
Dim result  As Boolean

Function send_sap_msg(Empfänger As String, Meldung As String)
   Set con = New sap_connect
   With con
         .Anmelden
         Set myfunc = .fns.Add("TH_POPUP")
         myfunc.exports("CLIENT") = .conn.client
         myfunc.exports("USER") = Empfänger
         myfunc.exports("MESSAGE") = Meldung
         result = myfunc.call
         .Abmelden
   End With
   Select Case result
         Case vbTrue
               send_sap_msg = "gesendet"
         Case vbFalse
               send_sap_msg = "Fehler"
   End Select
End Function
```

Coding 292: Systemmeldungen versenden mit TH_POPUP

Da es sich bei diesem Beispiel um eine Funktion handelt, erfolgt der Aufruf über einen Zelleintrag „=send_sap_msg(User; Meldung)“.

16. Sonstige Techniken (ohne Programmierung)

16.1. Gültigkeitsprüfung einsetzen

Über die Gültigkeitswerte kann die Eingabe in eine Zelle beschränkt werden. Eingerichtet wird die Gültigkeit wie folgt:

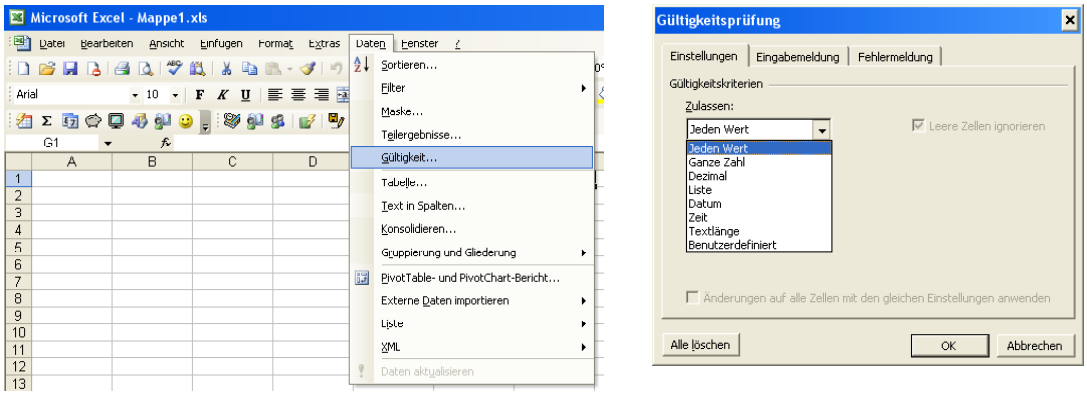


Abbildung 35: Gültigkeitsprüfung einrichten

Nun kann die Auswahl auf entsprechende Werte beschränkt werden.

16.1.1. Gültigkeitsprüfung „Ganze Zahl“

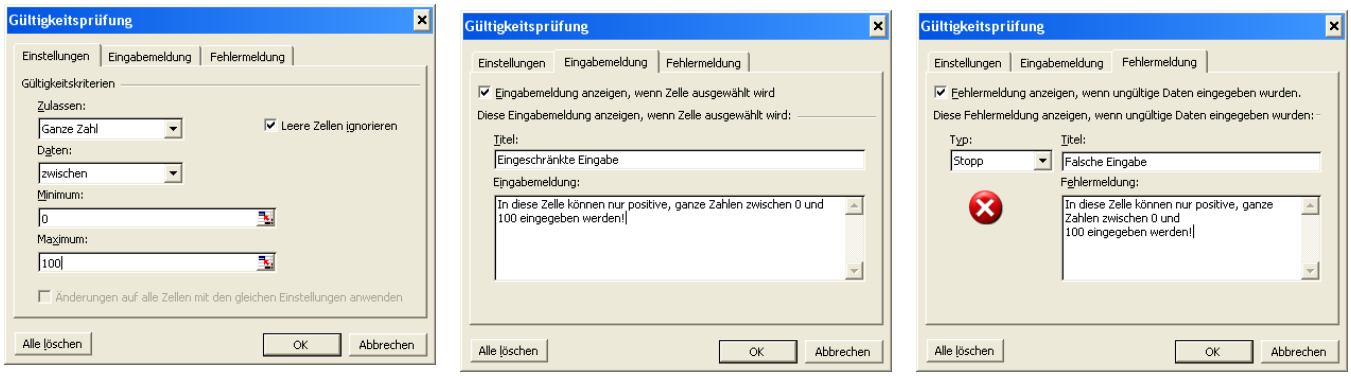


Abbildung 36: Einrichten einer Gültigkeitsprüfung mit Eingabe- und Fehlermeldung

Als Ergebnis erhält man einen Hinweis, wenn die Zelle aktiviert wird

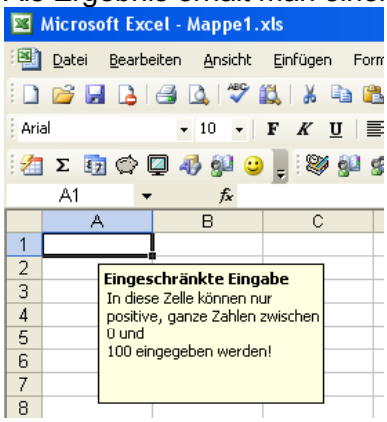


Abbildung 37: Hinweistext bei Gültigkeitsprüfung

und eine Fehlermeldung bei Falscheingabe:

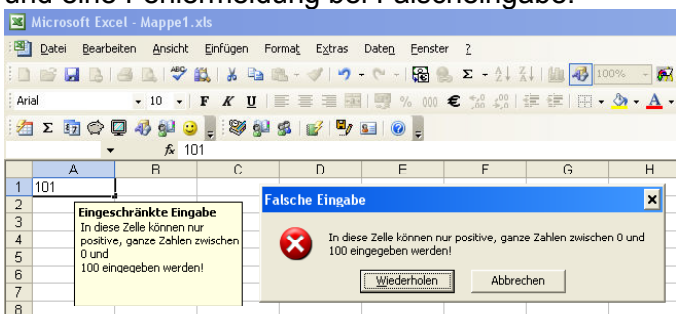


Abbildung 38: Fehlermeldung

16.1.2. Gültigkeitsprüfung „Liste“

Bei der Gültigkeitsprüfung auf Liste wird geprüft, ob der Wert in einer Liste vorhanden ist. Die Eingabe wird durch Dropdownauswahl erleichtert. Die Liste ist ein Zellbezug, der sich auf dem gleichen Worksheet befindet, wie die Zelle mit der Gültigkeitsprüfung:

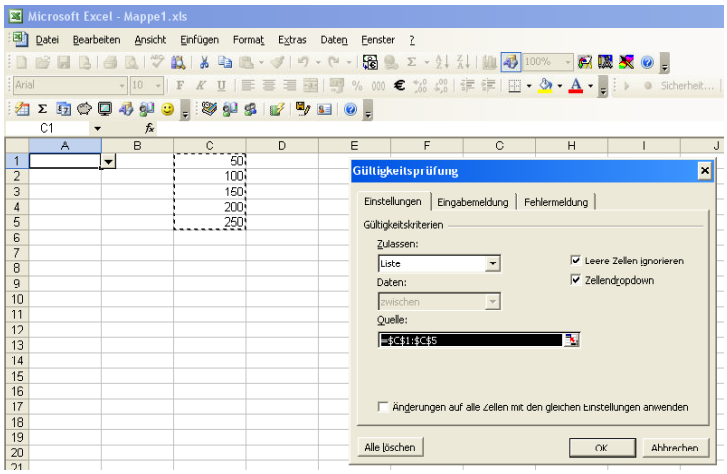


Abbildung 39: Liste als Gültigkeitswert

Die zu prüfende Zelle enthält nun eine Dropdownauswahl mit den gültigen Werten.

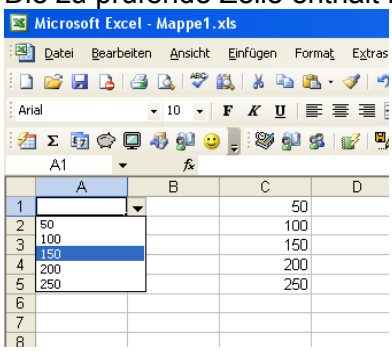


Abbildung 40: Gültigkeitsprüfung mit Dropdown-Auswahl

16.2. Datenauswertungen mit MS Query

16.2.1. Einfache Tabellenabfragen mit MS Query

Um die für Auswertungen benötigten Daten aus einer Datenbank heraus nach Excel zu bekommen, muss man nicht programmieren können. Hierfür gibt es MS Query.
In Excel wird der Menüpunkt: „Daten – externe Daten importieren – neue Abfrage erstellen“ aufgerufen. Für ein erstes Beispiel werden die Daten einer einzelnen Tabelle ausgewertet, in der die Namen und das Alter von Personen stehen:

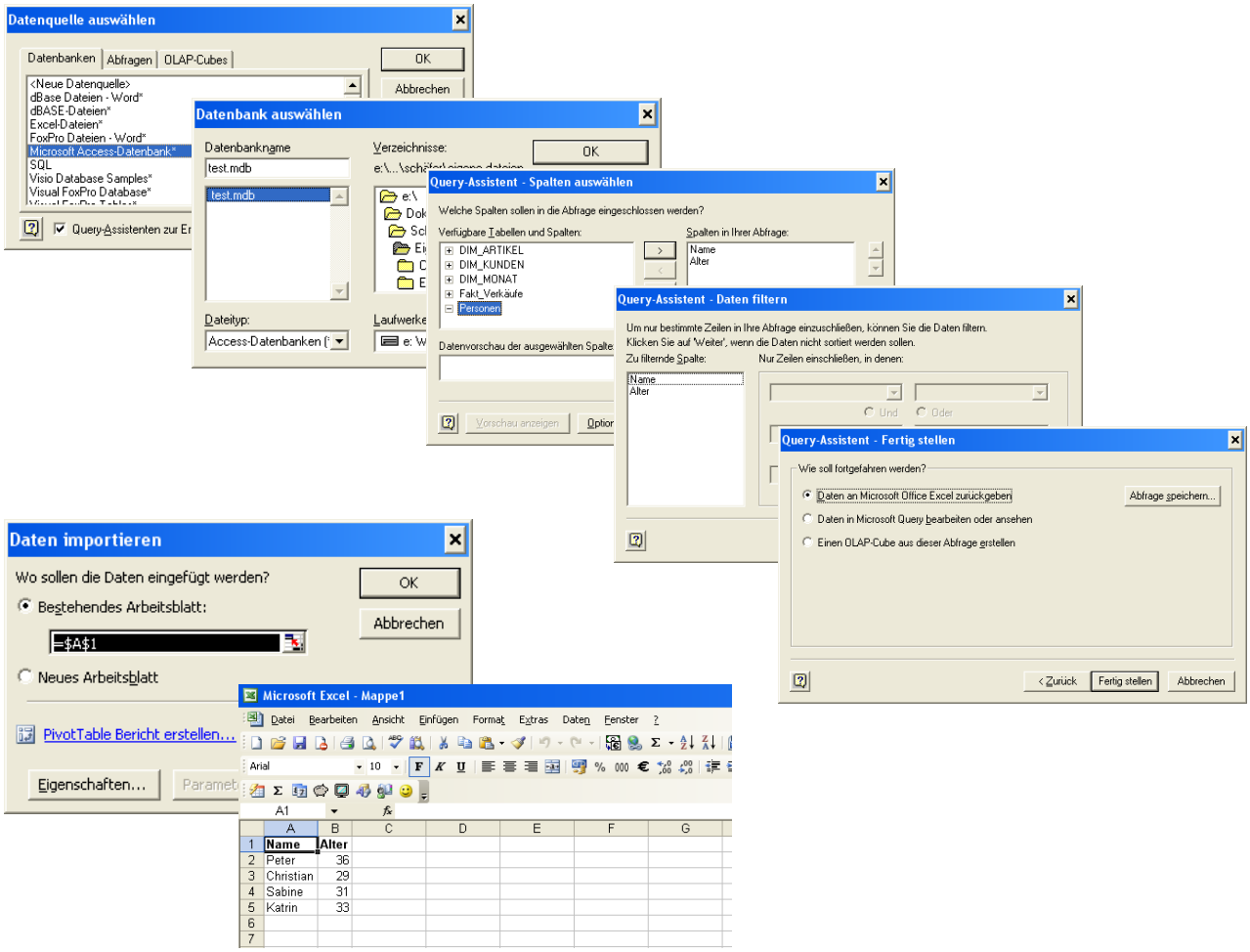
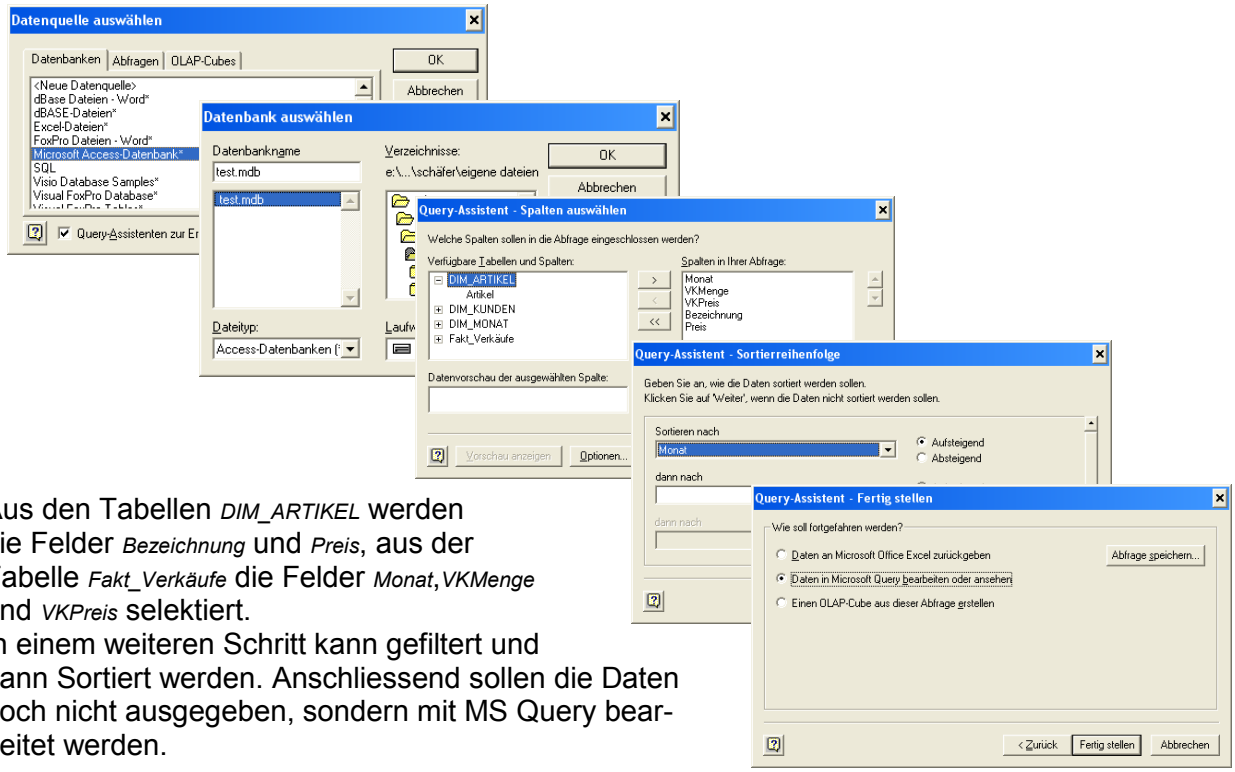


Abbildung 41: Abfrage mit MS Query erstellen

Die Rückgabe erfolgt ebenfalls tabellarisch.
MS Query kann auch zur Abfrage von SQL Server Tabellen, anderen Exceldateien udgl. verwendet werden.

16.2.2. Join Abfragen mit MS Query

Mit MS-Query können Daten auch über mehrere Tabellen hinweg abgefragt werden. Beispiel hierfür wäre, wenn die eine Tabelle (Faktentabelle), in der die Daten aus den monatlichen Verkäufen stehen, die verkauften Artikel jedoch nur mit der Artikelnummer. Alle Ausprägungen des Artikels befinden sich für dieses Beispiel in einer eigenen Tabelle (Dimensionstabelle).



Aus den Tabellen *DIM_ARTIKEL* werden die Felder *Bezeichnung* und *Preis*, aus der Tabelle *Fakt_Verkäufe* die Felder *Monat*,*VKMenge* und *VKPreis* selektiert.
In einem weiteren Schritt kann gefiltert und dann Sortiert werden. Anschliessend sollen die Daten noch nicht ausgegeben, sondern mit MS Query bearbeitet werden.

Abbildung 42: Abfrage über mehrere Tabellen

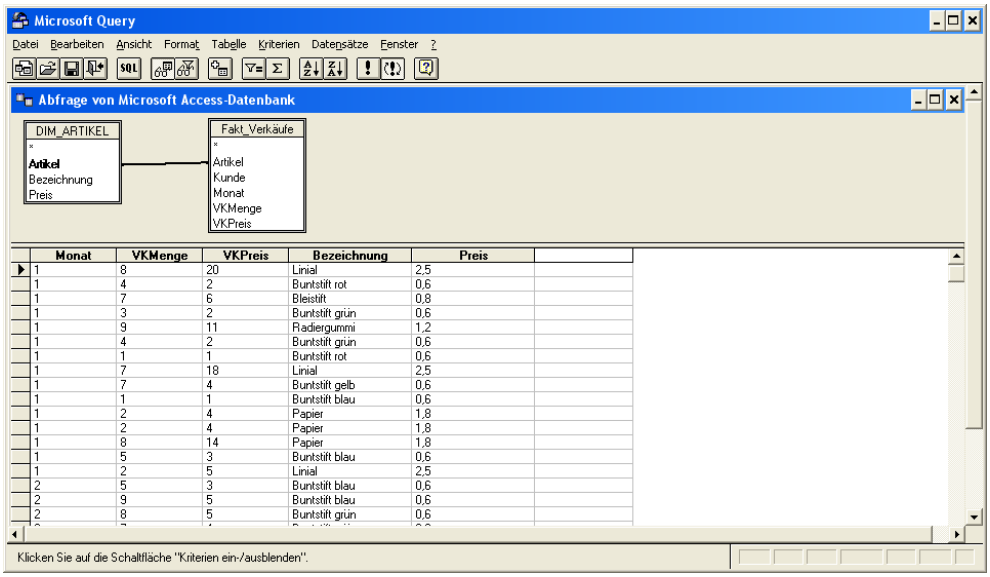


Abbildung 43: MS Query Arbeitsfenster

Im oberen Teil des Child - Fenster werden die Tabellen mit ihrer Beziehung gezeigt. Die Beziehung kann mit einem Doppelklick auf die Verbindungslinie bearbeitet werden.
Über das Menü lassen sich weitere Tabellen hinzufügen oder Filtereinstellungen vornehmen.

Die Daten werden über den Button an Excel zurückgegeben.

16.3. Arbeiten mit Cube's

16.3.1. Cube in Access erstellen

Ein Cube besteht aus mehreren Dimensionstabellen, die in fester Beziehung (mit referenzieller Integrität) zu einer oder mehreren Faktentabellen stehen. Bindeglieder sind die sog. Measures, die Keyfelder der Dimensionstabellen.
Eine Dimensionstabelle kann neben einem Keyfeld noch weitere Ausprägungen enthalten, die eine differenzierte Auswertung der Faktentabelle ermöglicht. Beispielweise wäre es möglich, die Verkäufe eines Unternehmens in einer Faktentabelle zu sammeln und als zeitliche Dimension auf die Kalenderwoche zu sichern. Eine Kalenderwoche kann nun wiederum auf einen Monat oder ein Quartal referenzieren, so dass die Verkäufe auch je Monat oder Quartal auswertbar sind.
Ein Blick auf das Entity Realisationship Model der Datenbank für das nachfolgende Beispiel:

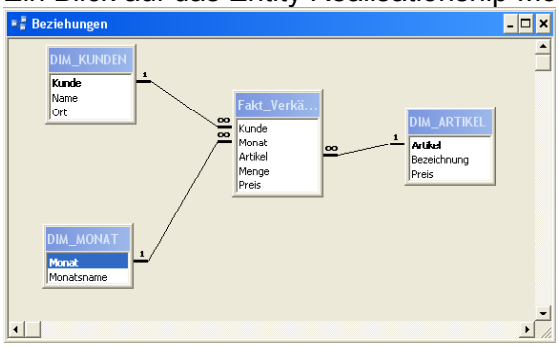


Abbildung 44: ERM eines einfachen Cube's

Die Faktentabelle wird in den Feldern Kunde, Monat und Artikel mit Referenzwerten zu den entsprechenden Dimensionstabellen gefüllt. Menge und Preis sind reine Füllwerte (der Preis wäre eine rechnerische Größe, die hier nicht berücksichtigt werden muss).
In den Dimensionstabellen stehen die Keyfelder in referenzieller Integrität zu den entsprechenden Feldern der Faktentabelle. Die Dimensionstabellen enthalten zusätzliche Ausprägungen, die in späteren Auswertungen verwendet werden können.

16.3.2. Auswertung des Cube's in Excel

Um die Daten entsprechend anzeigen zu können, wird eine Pivot Tabelle benötigt. Als Datenquelle wird „Externe Datenquelle" angegeben:

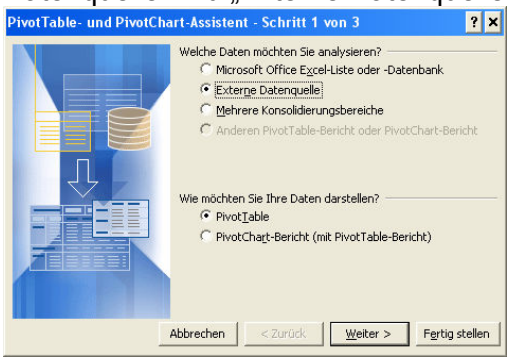


Abbildung 45: Aufruf Pivot Assistent

Anschliessend wird über einen Klick auf den Schalter „Daten importieren…" die MS Query aktiviert:

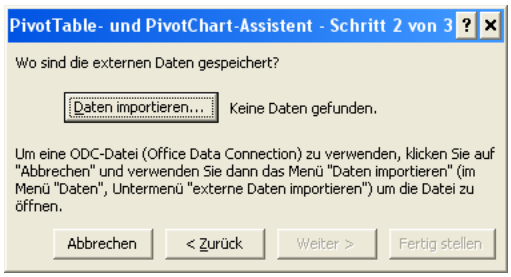


Abbildung 46: Aktivieren von MS Query

Das Selektieren von Daten mit MS Query kann unter 16.2 nachgelesen werden.
Für diese Auswertung wird nun aus der Dimension Artikel die *Bezeichnung*, aus der Dimension Kunde der *Name* und der *Ort*, aus der Dimension Monat der *Monatsname* und aus der Faktentabelle die *Menge* und der *Preis* benötigt.
Da die Datenbasis bereits entsprechend strukturiert ist, erfolgt sofort die Ausgabe in Excel:

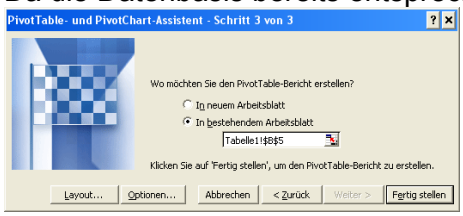


Abbildung 47: Abfrage der Ausgabezelle

Eine mögliche Ausgabe wäre dann diese:

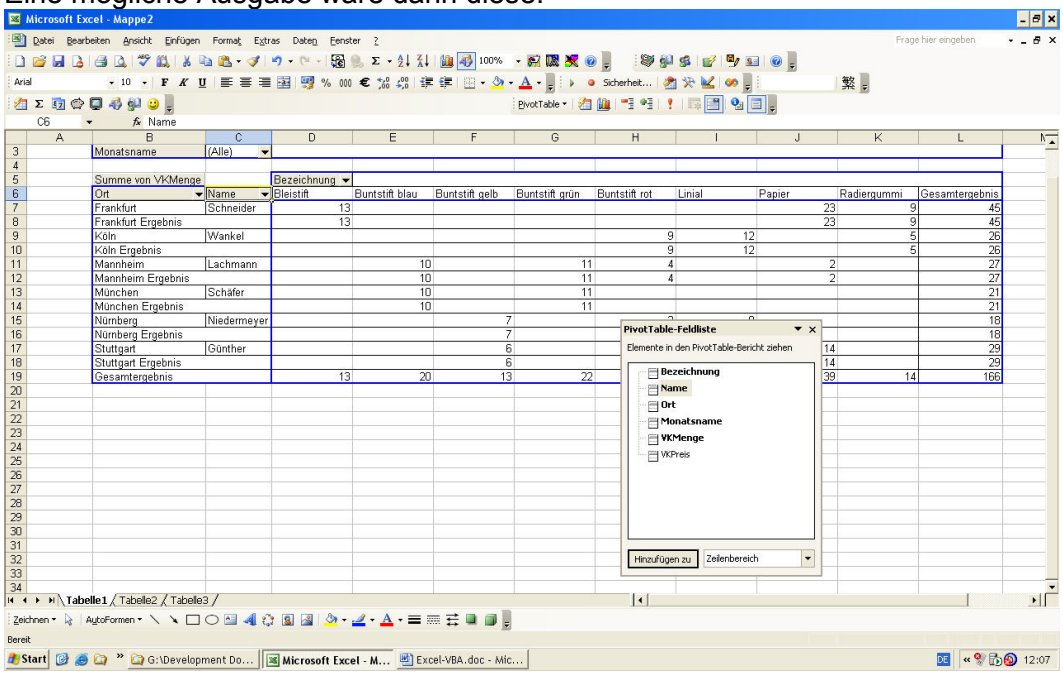


Abbildung 48: Ausgabe als Pivot Tabelle

16.3.3. Erstellen einer Cube Datei

Ein weiterer Weg ist, insbesondere bei weniger schön strukturierten Daten, eine Cube Datei zu erstellen. Dazu wird nach der Selektion der Spalten im Query Assistenten die Option „OLAP Cube erstellen“ ausgewählt:

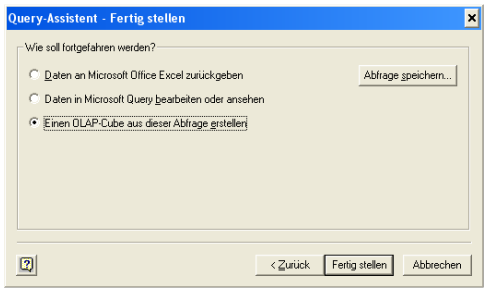


Abbildung 49: Erstellen eines OLAP Cube

Nach Anklicken des „Fertig stellen“ Button wird der OLAP-CUBE Assisten gestartet:

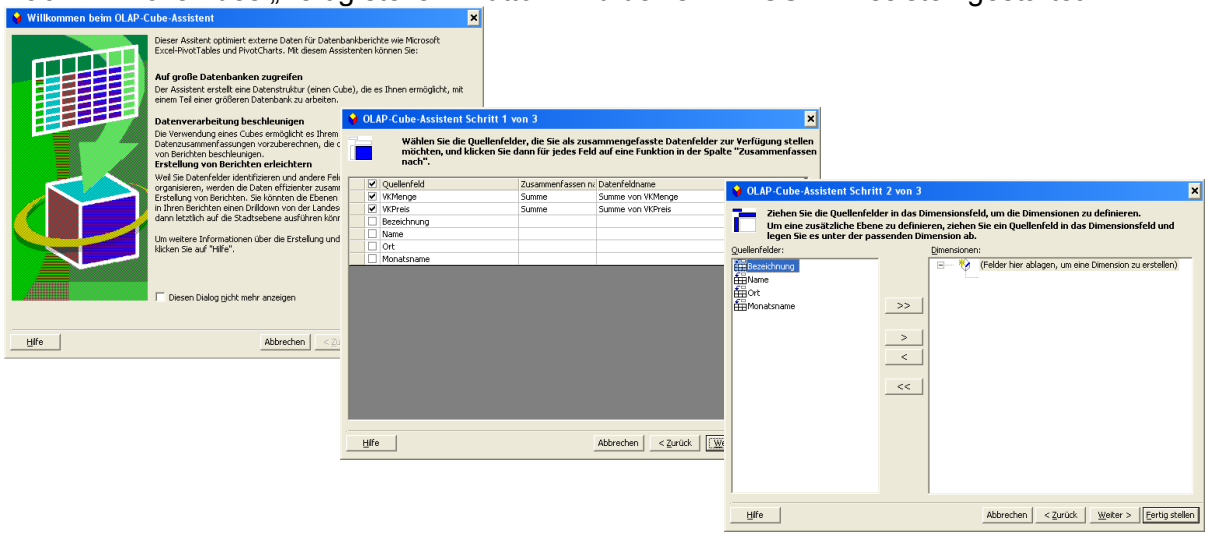


Abbildung 50: Erstellen einer Cube Datei

Im ersten Schritt werden die Quellfelder definiert. In Schritt Zwei folgen dann die Dimensionen, die in unserem Beispiel bereits die benötigte Struktur aufweise und 1:1 übernommen werden können. Als Letztes erfolgt das Sichern der Cube Datei und die Ausgabe im Excelsheet:

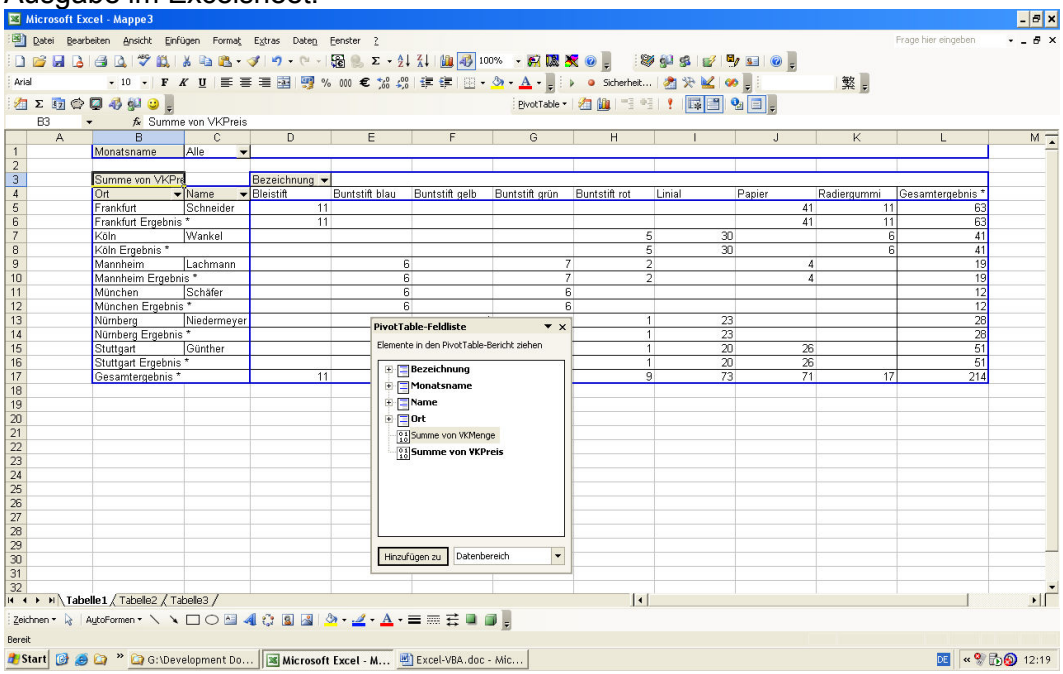


Abbildung 51: Pivot Tabelle aus Cube Datei

16.4. Arbeiten mit XML Dateien

XML (Extensible Marup Language) ist ein sehr flexibles Format. Mit XML ist es möglich, Daten in hierarchische Strukturen zu speichern.
Ein XML Dokument benötigt einen sog. Parser für die Verarbeitung. Dieser ist in den meisten Browsern enthalten. Der Parser prüft beim Öffnen, ob die XML Struktur fehlerhaft ist oder nicht und gibt ggf. eine entsprechende Meldung aus.
Die Bestandteile eines XML Dokumentes sind:

XML Deklaration
Verarbeitungsanweisungen
Kommentare
Elemente
Attribute
Text

Tabelle 29: XML Strukturen

16.4.1. XML Deklaration

Ein XML Dokument beginnt mit der Deklaration. Diese ist zwar optional, da aber an dieser Stelle Codierungsparameter gesetzt werden, sollte diese immer mitgegeben werden.

```
<?xml version=„1.0“ encoding=„ISO-8859-1“?>
```

XML gibt es in den Versionen 1.0 und 1.1. Die wesentlichen Unterschiede sind, dass in Version 1.1 neuere Unicode Versionen unterstützt werden, zusätzliche Zeichen in Namen erlaubt sind und zustätzliche Zeichen-Referenzen für Kontrollzeichen erlaubt sind.

Als Kodierungsdeklaration können folgende Parameter mitgegeben werden:

„UTF-8“	Unicode-Zeichensatz
„UTF-16“	Unicode-Zeichensatz
„ISO-8859-1“	ASCII-Zeichensatz (incl. Umlaute äöü)

Tabelle 30: Kodierungsdeklarationen

16.4.2. Verarbeitungsanweisungen

Die Verarbeitungsanweisungen ist optional und steuert, mit welcher Applikation das Dokument verarbeitet werden soll.

```
<?mso-application progid="Excel.Sheet"?>
```

Programm ID’s sind z.B.:

„Excel.Sheet”	Microsoft Excel
„Word.Document”	Microsoft Word

Tabelle 31: Programm ID’s

16.4.3. Kommentare

Kommentare werden nicht verarbeitet, sie dienen nur der Information für den Benutzer.

```
<!--Dies ist ein Kommentar.-->
```

16.4.4. Elemente

Elemente sind die Datenträger des XML Dokumentes. Ein Element bestehen aus einem Start-Tag und einem End-Tag. Bei der Schreibweise der Tags ist auf absolute Exaktheit zu achten – alle Buchstaben müssen identisch sein (Klein-/Grossschreibung). Zusätzlich ist zu beachten, dass ein Tag nur mit einem Buchstaben oder einem Unterstrich beginnen muss. Zahlen und sonstige Sonderzeichen sind nicht erlaubt.

Zwischen dem Start-Tag und dem End-Tag können die Inhalte in Form von Zahlen und/oder Zeichen geschreiben werden.

```
<Wert>20</Wert>
```

Ein Element kann aber auch ohne Inhalt bleiben – sog. leeres Element.

```
<Wert></Wert>
```

Zudem kann die Schreibweise gekürzt werden:

```
<Wert />
```

XML Elemente können verschachtelt werden. Dabei muss das Element, dass den Start-Tag des nächsten Elementes enthält, auch dessen End-Tag enthalten:

```
<Wert>20
   <Einheit>kg</Einheit>
</Wert>
```

In der Chronologie wird in jedem XML Dokument ein sog. Wurzelelement angelegt, das alle anderen Elemente enthält. Die Beziehung zum nächsten, verschachtelten Element wird als Eltern-Kind Beziehung und die Elemente entsprechend als Eltern- bzw. Kind-Element bezeichnet.
Die Beziehung zwischen irgendeinem verschachtelten Element wird zu irgendeinem übergeordneten Element wird als Vorfahr-Nachfahr Beziehung bezeichnet – die Elemente entsprechen als Vorfahr bzw. Nachfahr.

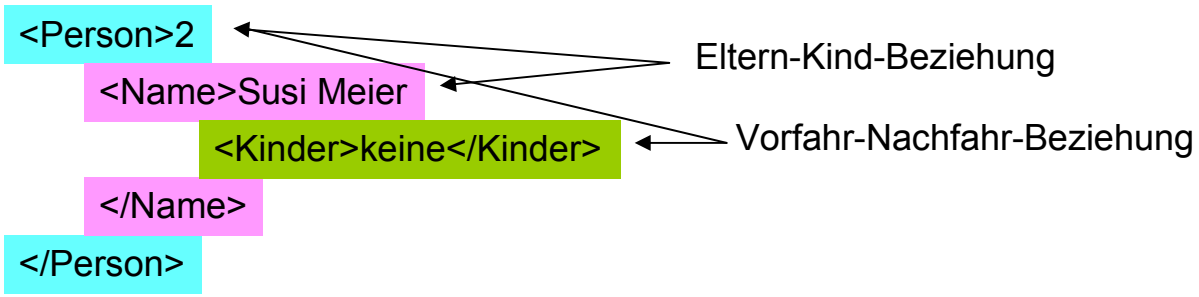


Abbildung 52: Beziehungen zwischen den Elementen

16.4.5. Attribute

Ein Attribut besteht aus einem Bezeichner und einem Wert. Jeder Start-Tag kann mit beliebig vielen Attributen versehen werden. Attribute werden nach dem Element-Namen vergeben und sind von diesem durch ein Leerzeichen getrennt. Auf einen Attribut-Bezeichner folgt ein Gleichheitszeichen und ein in Anführungstrichen gesetzter Wert.

```
<Wert Einheit=„KG“>20</Wert>
```

Attribute lassen sich typisieren:

CDATA	beiebiger Text
(Aufzählung)	Liste mit möglichen Inhalten (rot|grün|gelb)
NMTOKEN	nur Zeichen, die in XML-Tag-Namen erlaubt sind
NMTOKENS	Liste mehrerer NMTOKEN – Leerzeichen sind Trennzeichen
ID	wie NMTOKEN, wobei ID im Dokument einmalig sein muss
IDREF	Bezug auf eine ID
IDREFS	Liste mit Bezügen auf ID’s
ENTITY	Verweis auf ein definiertes Entity (nur in DTD erlaubt)
ENTITIES	Liste mit Verweisen auf definierte Entities
NOTATION	spezieller DTD Konstrukt zur Abkürzung auf SYSTEM-ID’s

Tabelle 32: Attributtypen

16.4.6. Text

Text ist die eigentliche Füllung eines XML Dokuments. Dieser wird in den Elementen gespeichert. Dabei ist zu beachten, dass nicht alle Zeichen in einem Text erlaubt sind, bzw. durch Entities ersetzt werden müssen:

Kaufmannsund - &	&amp;
Apostroph - '	&apos;
Grösser als Zeichen - >	&gt;
Kleiner als Zeichen - <	&lt;
Anführungszeichen - =	&quot;

Tabelle 33: Vordefinierte Entities

Zusätzlich ist zu beachten, dass bei der Eingabe von Text nur diejenigen Zeichen verwendet werden, die im Encoding-Zeichensatz enthalten sind. Die Mitgabe eines „ü“ bei vorheriger Deklaration des UTF-8 Encoding Zeichensatze führt zu einer Fehlermeldung.

```
<Nachricht>
   <Text>Dies ist der Text.</Text>
</Nachricht>
```

16.4.7. Aufbau eines einfachen XML Dokumentes

Nachstehend ein Beispiel für ein XML Dokument. Die Eingabe des Codes erfolgt im Editor – die Datei sollte auf dem Desktop unter dem Namen „test-xml.xml“ erfolgen.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!--Beispieldatei für Vertriebsdaten in einer XML Datei-->
<Vertrieb Jahr="2009">
   <Umsatz Region="Nord">
         <Jan>20</Jan>
         <Feb>15</Feb>
         <Mrz>25</Mrz>
   </Umsatz>
   <Umsatz Region="Mitte">
         <Jan>25</Jan>
         <Feb>10</Feb>
         <Mrz>30</Mrz>
   </Umsatz>
   <Umsatz Region="Süd">
         <Jan>35</Jan>
         <Feb>25</Feb>
   <Mrz>35</Mrz>
   </Umsatz>
</Vertrieb>
```

In der ersten Zeile steht die XML Deklaration. Als Encoding wurde ein ISO Zeichensatz gewählt – der Attributenwert „Süd” würde sonst einen Fehler verursachen.

16.4.8. DTD – Documenttyp Definitionen

Eine DTD definiert die Struktur des XML Dokumentes. Die DTD kann sowohl im XML Dokument (sog. interne Teilmenge) als auch als eigenständige Datei (sog. externe Teilmenge) existieren. Dabei bestimmt die DTD Art und Umfang der Elemente und Attribute einer XML Datei.
Das Beispiel aus 16.4.7 kann mit einer internen Teilmenge folgender Massen aufgebaut werden:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!--Beispieldatei für Vertriebsdaten in einer XML Datei-->
<!DOCTYPE Vertrieb [
<!ATTLIST Umsatz Region („Nord” | „Mitte” | „Süd”) „Nord” >
]>
<Vertrieb Jahr="2009">
   <Umsatz Region="Nord">
         <Jan>20</Jan>
         <Feb>15</Feb>
         <Mrz>25</Mrz>
   </Umsatz>
```

```
   <Umsatz Region="Mitte">
         <Jan>25</Jan>
         <Feb>10</Feb>
         <Mrz>30</Mrz>
   </Umsatz>
   <Umsatz Region="Süd">
         <Jan>35</Jan>
         <Feb>25</Feb>
   <Mrz>35</Mrz>
   </Umsatz>
</Vertrieb>
```

Für das Attribut Region, des Elements Umsatz wurde eine Auswahlliste erstellt. Nun muss das Attribut einen der Werte der Liste enthalten. Als Default-Wert wurde „Nord“ gesetzt.
Mit einer externen Teilmenge sähe das Beispiel dann wie folgt aus:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE Vertrieb SYSTEM "test.dtd">
<Vertrieb Jahr="2009">
   <Umsatz Region="Nord">
         <Jan>20</Jan>
         <Feb>15</Feb>
         <Mrz>25</Mrz>
   </Umsatz>
   <Umsatz Region="Mitte">
         <Jan>25</Jan>
         <Feb>10</Feb>
         <Mrz>30</Mrz>
   </Umsatz>
   <Umsatz Region="Süd">
         <Jan>35</Jan>
         <Feb>25</Feb>
   <Mrz>35</Mrz>
   </Umsatz>
</Vertrieb>
```

Zusätzlich wird nun eine Datei mit der Bezeichnung „test.dtd” benötigt. Deren Inhalt entspricht wiederum einer xml-Deklaration, gefolgt von Anweisungen über die Attributliste, die nachstehend als Mussangabe (Required) und als Auswahlliste deklariert wird:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!ATTLIST Umsatz Region NMTOKEN #REQUIRED>
<!ATTLIST Umsatz Region (Nord|Mitte|Süd) "Nord">
```

16.4.9. Einfügen von XML Elementen in Excel

Um die Datei nun in Excel lesen zu können, wird ein Link auf die XML Datei gelegt:

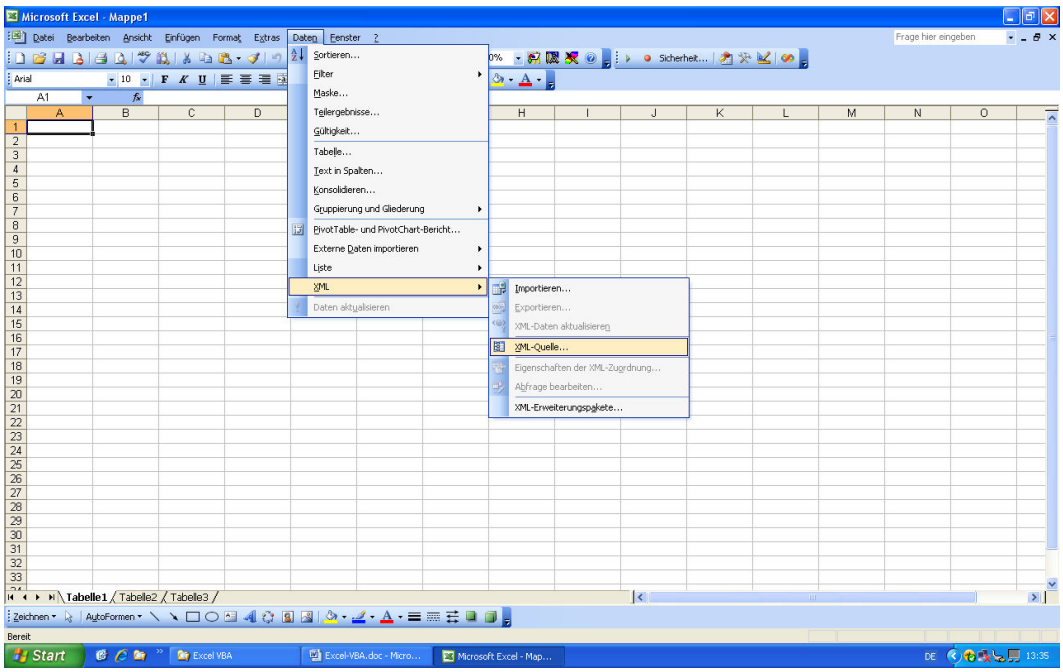


Abbildung 53: Menü Daten-XML aufrufen

Im Bildschirm rechts erscheint ein Rollupmenü für XML-Quellen. Hier wird als nächstes der Button „XML-Verknüpfungen…“ angeklickt und es erscheint ein Fenster mit XML Zuordnungen. Über den Button „Hinzufügen…“ kommt man auf das Auswahlfenster für XML Quellen. Dort muss nun die oben benannte Datei ausgewählt werden und die Eingabe mit dem Button „Öffnen“ bestätigt werden.

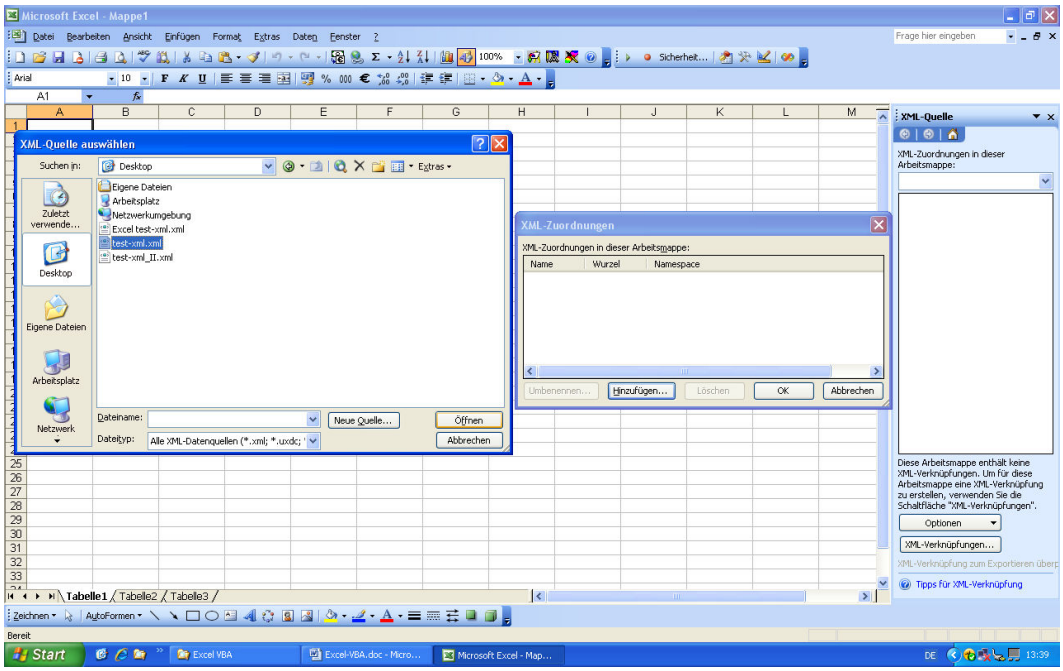


Abbildung 54: Auswahlfenster für XML Quellen

Die Quelle erscheint im XML-Zuordnungs-Fenster. Mit einem Klick auf den OK Button wird die Auswahl übernommen.

Abbildung 55: XML-Zuordnungsfenster

Nun stehen die XML Daten im Rollupmenü zur Verfügung.
Die Elemente können nun einzeln oder als gesamte Hirarchie, per Drag & Drop in das Excelsheet übernommen werden.

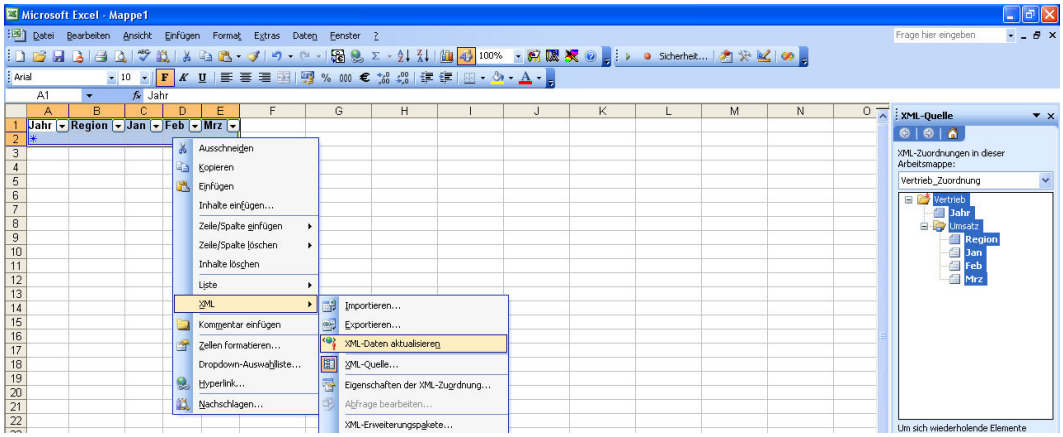


Abbildung 56: XML Struktur im Sheet platzieren

Über das Kontextmenü XML – XML-Daten aktualisieren werden die Daten angezeigt.

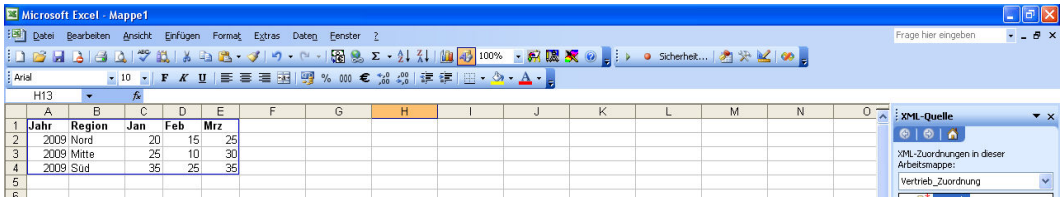


Abbildung 57: XML Daten anzeigen

16.4.10. Deklaration einer XML für Excel

Mit der zusätzlichen Verarbeitungsanweisungen für Excel (Zeile 3) wird die Datei direkt mit Excel verknüpft.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!--Beispieldatei für Vertriebsdaten in einer XML Datei-->
<?mso-application progid="Excel.Sheet"?>
<Vertrieb Jahr="2009">
   <Umsatz Region="Nord">
         <Jan>20</Jan>
         <Feb>15</Feb>
         <Mrz>25</Mrz>
   </Umsatz>
   <Umsatz Region="Mitte">
         <Jan>25</Jan>
         <Feb>10</Feb>
         <Mrz>30</Mrz>
   </Umsatz>
   <Umsatz Region="Süd">
         <Jan>35</Jan>
```

```
        <Feb>25</Feb>
    <Mrz>35</Mrz>
    </Umsatz>
</Vertrieb>
```

Wird das XML Dokument nun per Doppelklick gestartet, so öffnet sich eine Excelinstanz im Browserfenster.

Abbildung 58: Sicherheitsabfrage (je nach Browsereinstellung)

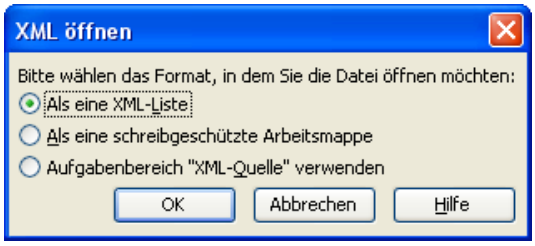


Abbildung 59: Format-Abfrage

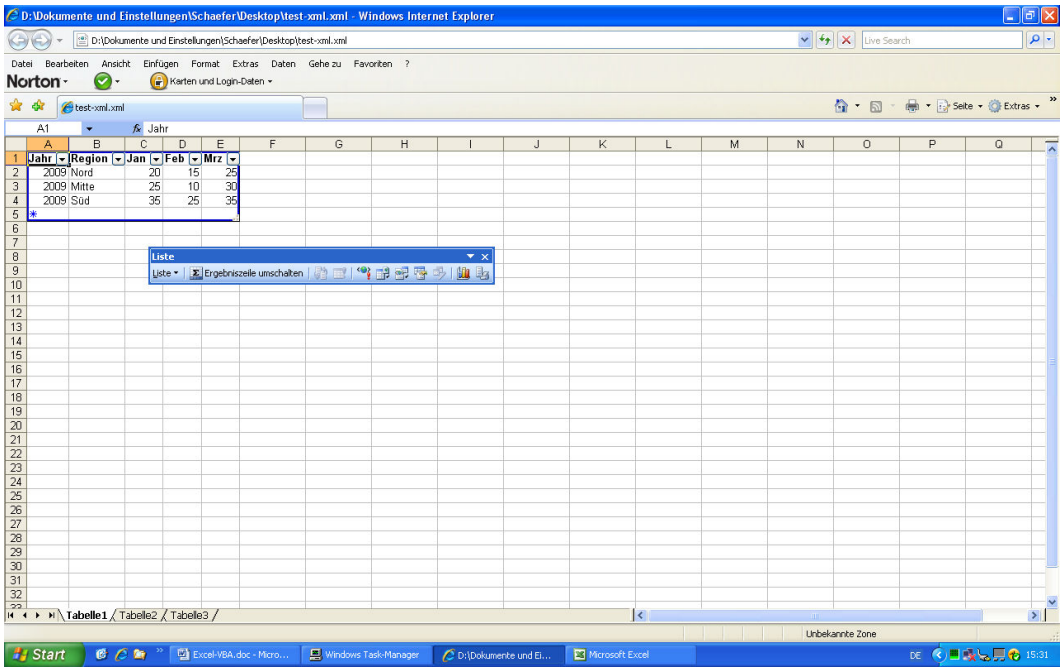


Abbildung 60: Excelapplikation im Browserfenster