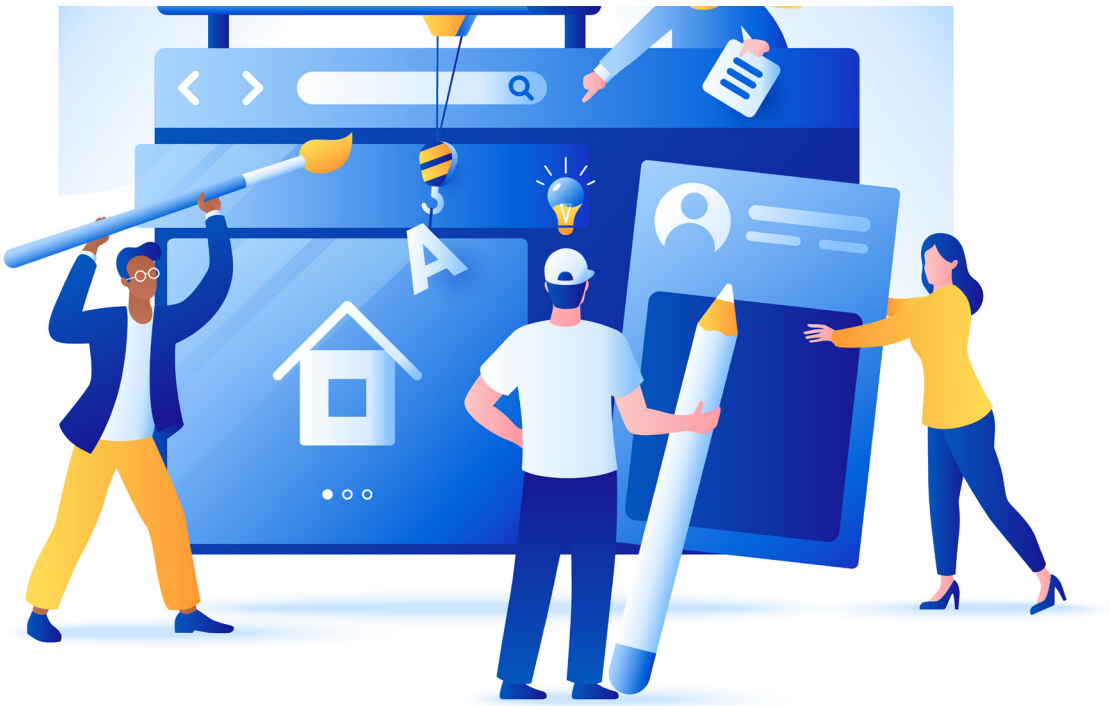
HAUFE.
Ausgabe 01/2022

ACCESS
IM UNTERNEHMEN

VBA-EDITOR PROGRAMMIEREN

Lernen Sie, den VBA-Editor per Code zu steuern und somit Aufgaben zu automatisieren (ab Seite 17).



In diesem Heft:

DATEV-EXPORT

Exportieren Sie Ihre Buchungsdaten in das DATEV-Format und leiten Sie diese direkt an Ihren Steuerberater weiter.

SEITE 60

ETIKETTEN DRUCKEN

Automatisieren Sie die Ausgabe von Adressetiketten direkt auf einen Labeldrucker.

SEITE 2

FEIERTAGE PER VBA

Ermitteln Sie die Feiertage für ein Jahr und ein Bundesland direkt über eine praktische VBA-Funktion.

SEITE 7

VBA-Editor programmieren

Wer nicht nur Access-Anwendungen programmieren möchte, sondern vielleicht auch einmal Tools entwickeln will, mit denen sich bestimmte Schritte bei der Programmierung vereinfachen lassen, kommt nicht um Kenntnisse zur Programmierung der Entwicklungsumgebung selbst herum. Um dieses Thema kümmern wir uns in einem Schwerpunkt in dieser Ausgabe. Dabei schauen wir uns die verschiedenen Bereiche und Techniken an, mit denen Sie die Elemente eines VBA-Projekts per Code selbst erstellen und manipulieren können.

Den Start macht der Beitrag **VBA-Projekt per VBA referenzieren** (ab Seite 17). Hier erfahren Sie, wie Sie das VBA-Projekt der aktuellen Access-Datenbank überhaupt referenzieren, denn das ist die Basis für alle weiteren Schritte beim Programmieren der Entwicklungsumgebung.

Der Beitrag **Zugriff auf den VBA-Editor mit der VBE-Klasse** beschreibt ab Seite 21 die Grundlagen der Klasse, die alle weiteren Elemente, Methoden und Eigenschaften für den Zugriff auf die Elemente des VBA-Editors bereitstellt. Hier lernen Sie auch die grundlegenden Elemente des VBA-Editors aus Sicht dieser Klasse kennen.

Im Beitrag **Zugriff auf VBA-Projekte per VBProject** erfahren Sie ab Seite 25, wie Sie mit der **VBProject**-Klasse auf die weiteren Elemente des VBA-Editors zugreifen können, hier speziell auf die Module, die Sie über die **VBComponents**-Auflistung erreichen. Außerdem können Sie hiermit beispielsweise die Projekteigenschaften einstellen oder ein Kennwort für das VBA-Projekt festlegen. Wie Sie mit den Modulen eines VBA-Projekts arbeiten, erfahren Sie ab Seite 30 im Beitrag **Module und Co. im Griff mit VBComponent**. Die **VBComponent**-Elemente bieten über die **CodeModule**-Eigenschaft Zugriff auf die eigentlichen Module. Sie können diese mit der **VBComponents**-Auflistung durchlaufen und darauf zugreifen sowie neue **VBComponent**-Elemente anlegen oder bestehende löschen.

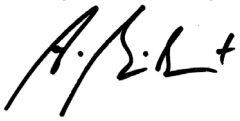
Was Sie genau mit den **CodeModule**-Elementen anfangen können, zeigt dann der Beitrag **VBA-Code manipulieren mit der CodeModule-Klasse** ab Seite 35. Das CodeModule-Element erlaubt den direkten Zugriff auf den Code der Module.

Etwas mehr in Richtung Benutzeroberfläche geht es im Beitrag **Auf VBA-Code zugreifen mit der CodePane-Klasse** ab Seite 47. Hier erfahren Sie beispielsweise, wie Sie den aktuell markierten Code im VBA-Fenster auslesen oder selbst per Code eine Markierung setzen.

Die Ausgabe hat jedoch noch mehr zu bieten: Im Beitrag **Export von Daten in das DATEV-Format** finden Sie ab Seite 60 eine spannende Lösung, mit der Sie die Daten aus Ihrer Buchhaltungsanwendung direkt in das DATEV-Format exportieren können, um es an Ihren Steuerberater weiterzuleiten.

Richtig praktisch wird es ab Seite 2 im Beitrag **Etiketten drucken mit Access**: Hier lernen Sie, wie Sie einen Bericht zum Ausdrucken auf einem Etikettendrucker erstellen und den Ausdruck vorbereiten können. Und unter **Feiertage mit VBA ermitteln** zeigen wir Ihnen ab Seite 7, wie Sie die Feiertage für ein Jahr für ein bestimmtes Bundesland ganz einfach per VBA ermitteln und sogar in eine Tabelle schreiben können.

Viel Spaß beim Ausprobieren!



Ihr André Minhorst

Etiketten drucken mit Access

Ein Kunde fragte neulich, warum ich nicht mal einen Beitrag darüber verfasse, wie man mit einem Thermodrucker beispielsweise Versandetiketten druckt. Also machen wir das einfach! Dieser Artikel zeigt am Beispiel des Brother-Druckers QL-700, wie Sie aus Access heraus Etiketten drucken können.

Adressdaten vorbereiten

Für die Ausgabe von Etiketten benötigen wir einige Daten, zum Beispiel Adressdaten, sowie einen Bericht, der die Ausgabe der Daten formatiert. Die Adressdaten verwalten wir in der Tabelle **tblAdressen**, die mit Daten wie in Bild 1 aussieht.

tblAdressen

AdresseID	Firma	Anrede	Vorname	Nachname	Strasse	PLZ	Ort
1	Krahn GbR	Herr	Adi	Stratmann	Kremser Straße 54	10589	Berlin
2	Göllner AG	Frau	Heidi	Eich	Moosstraße 30	42289	Wuppertal
3	Peukert GmbH & Co. KG	Herr	Wernfried	Birk	Wiener Straße 78	22297	Hamburg
4		Herr	Vitus	Krauße	Burgenlandstraße 77	20355	Hamburg
5	Mader KG	Frau	Jadwiga	Oehme	Peter-Rosegger-Straße	65187	Wiesbaden
6	Fleischhauer GmbH	Herr	Niko	Michel	Kindergartenstraße 42	12051	Berlin
7		Herr	Siegert	Loos	Lenaustraße 80	66115	Saarbrücken
8		Herr	Michl	Schroth	Dr. Karl Renner-Straße	01129	Dresden
9		Herr	Florentius	Wittek	Industriestraße 7	80638	München
10	Krahl KG	Herr	Gernulf	Riegel	Erzherzog-Johann-Stra	81545	München
11	Becker AG	Herr	Tristan	Hübsch	Jahnstraße 78	65193	Wiesbaden
12	Runge GmbH	Herr	Heinfried	Steinert	Kaplanstraße 60	30159	Hannover
13	Albert AG	Frau	Herma	Aigner	Burgstraße 31	22089	Hamburg
14		Frau	Mina	Dörfler	Schloßstraße 66	38118	Braunschweig
15	Quandt GmbH & Co. KG	Frau	Jo	Pickel	Kreuzstraße 29	79114	Freiburg
16		Herr	Heimbert	Rohrer	Lenaustraße 83	10785	Berlin
17	Lührs AG	Herr	Roderich	Reil	Flurstraße 100	40595	Düsseldorf

Datensatz: 23 von 100 | Kein Filter | Suchen

Bild 1: Adressdaten für die Adressetiketten

Die Adressen sind bis auf einige Adressen ohne Firma vollständig, weshalb wir im Bericht zwei verschiedene Formate vorsehen wollen:

- Adressen mit Firma: Firma in der ersten Zeile, Anrede, Vorname und Nachname in der zweiten Zeile, Straße in der dritten und PLZ und Ort in der vierten Zeile.

- Adressen ohne Firma: Anrede in der ersten Zeile, Vorname und Nachname in der zweiten Zeile, Straße in der dritten und PLZ und Ort in der vierten Zeile.

Bericht erstellen

Um den Bericht zu erstellen, der in dem gewünschten Format auf Etiketten ausgedruckt werden soll, legen wir zunächst einen einfachen Bericht in der Entwurfsansicht an. Diesem weisen wir als Datensatzquelle die Tabelle **tblAdressen** zu. Nun könnten wir jedes der Felder einzeln in den Detailbereich des Berichtsentwurfs ziehen, aber wenn wir die Darstellung mit wahlweise Firma oder Anrede in der ersten Zeile realisieren wollen, ist ein einziges Textfeld praktischer.

Nachdem wir dieses hinzugefügt haben, entfernen wir zuerst das Beschriftungsfeld des Textfeldes und stellen dann seine Eigenschaft **Steuerelementinhalt** auf den folgenden Ausdruck ein:

```
=Wenn(IstNull([Firma]);[Anrede];[Firma])
  & Zchn(13) & Zchn(10)
& Wenn(IstNull([Firma]);Null;[Anrede])+" "
  & [Vorname] & " " & [Nachname]
  & Zchn(13) & Zchn(10)
& [Strasse]
  & Zchn(13) & Zchn(10)
& [PLZ] & " " & [Ort]
```

Der Berichtsentwurf sieht danach wie in Bild 2 aus.

Der Ausdruck prüft zu Beginn, ob das Feld **Firma** für den aktuellen Datensatz **Null** ist. Falls ja, wird zuerst das Feld **Anrede** ausgeben, sonst **Firma**. **Zchn(13) & Zchn(10)** ist ein Zeilenumbruch und entspricht der Konstanten **vbCrLf**.

Der Ausdruck für die zweite Zeile prüft wieder, ob **Firma** den Wert **Null** hat. Falls ja, gibt der Wenn-Ausdruck den Wert **Null** aus, falls nein, den Wert des Feldes **Anrede**. Warum geben wir den Wert **Null** aus und nicht einfach eine leere Zeichenkette? Weil wir den Inhalt der **Wenn**-Bedingung per Plus-Zeichen (**+**) mit dem folgenden Teil **" " & Vorname & " " & Nachname** verknüpft haben. Und wenn der **Wenn**-Teil den Wert **Null** liefert, dann wird der komplette Ausdruck, der mit dem Plus-Zeichen verknüpft ist, zum Wert **Null**. Wenn die Anrede also in der zweiten Zeile nicht ausgegeben werden soll, weil sie schon in der ersten Zeile steht, würde dort sonst noch das Leerzeichen zwischen **Anrede** und dem **Vorname** erscheinen. Dies verhindern wir durch das Ausgeben von **Null** in dem Fall, dass die Anrede schon in der ersten Zeile ausgegeben wurde und die Verknüpfung mit dem Ausdruck **" "** durch ein Plus-Zeichen statt durch das Kaufmanns-Und (**&**).

Danach folgt noch eine weitere Zeile mit dem Wert des Feldes **Strasse** sowie in der letzten Zeile **PLZ** und **Ort**. Wechseln wir nun in die **Seitenansicht**, erhalten wir die Adressen wie in Bild 3.

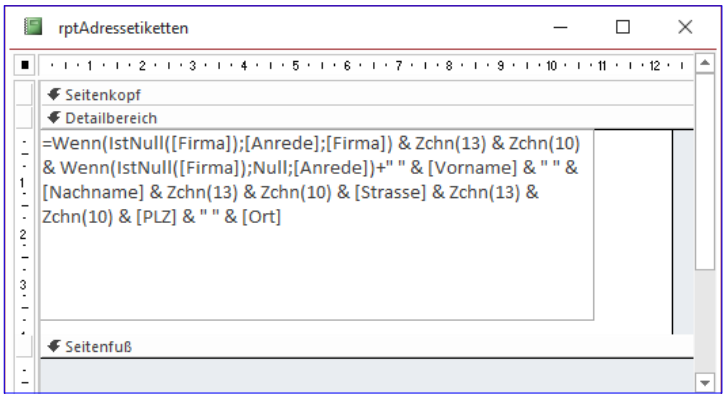


Bild 2: Erster Entwurf des Berichts

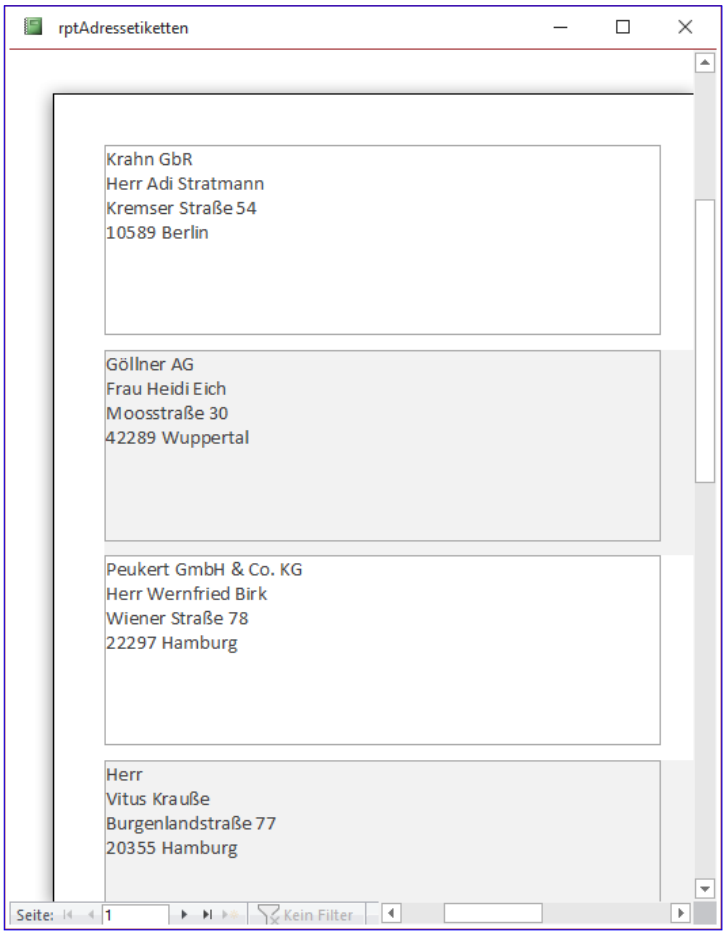


Bild 3: Adressdaten in der Seitenansicht

Bericht an die Adressetiketten anpassen

Damit folgt der interessante Teil der Aufgabe: Wir wollen den Bericht so anpassen, dass jeweils eine Adresse je Seite in der Größe der Etiketten angezeigt wird. Dazu öffnen wir den Bericht in der Entwurfsansicht und wechseln im Ribbon zum Reiter **Seite einrichten**. Hier klicken Sie auf **Seite einrichten**, was den Dialog **Seite einrichten** öffnet. In diesem wechseln Sie zur Registerseite **Seite**. Hier finden Sie die Option **Drucker für [Berichtsname]**. Wählen Sie hier statt **Standarddrucker** die Option **Spezieller Drucker** aus und klicken Sie anschließend auf die Schaltfläche **Drucker...** (siehe Bild 4).

Feiertage per VBA ermitteln

Wenn Sie mit Access Termine verwalten, werden Sie auch Feiertage berücksichtigen wollen. Die entsprechenden Datumsangaben liefert Access leider nicht frei Haus – Sie müssen selbst eine entsprechende Funktion bereitstellen. Der vorliegende Beitrag zeigt, wie Sie die Feiertage ermitteln und wie Sie schnell prüfen, ob ein Feiertag auf ein bestimmtes Datum fällt.

Es gibt verschiedene Gruppen von Feiertagen: Solche, die jedes Jahr auf den gleichen Termin fallen (wie Neujahr, Weihnachten oder Tag der deutschen Einheit), Tage, die vom Datum des Ostersonntags abhängen, das auf komplizierte Art berechnet wird, und Tage, die vom Datum des vierten Advents abhängen. Außerdem müssen Sie bei der Ermittlung von Feiertagen berücksichtigen, dass es einige Feiertage nur in bestimmten Bundesländern gibt.

So ist Heilige Drei Könige am 6. Januar nur in drei Bundesländern ein Feiertag, in den übrigen Bundesländern wird gearbeitet. Außerdem ändern sich die Feiertage der verschiedenen Bundesländer gelegentlich oder es kommen neue Feiertage hinzu – so ist der Internationale Frauentag als Feiertag noch recht neu und wird nur in Berlin begangen und der Weltkindertag findet als Feiertag nur in Thüringen statt.

Tabellen oder reiner Code?

Vor dem Zusammenstellen der Lösung zu diesem Beitrag stand die Frage im Raum, ob man die Basisdaten zu den Feiertagen in Tabellen speichert oder ob man diese komplett per VBA-Code ermittelt. In Anbetracht dessen, dass die Feiertage nur alle Jubeljahre geändert werden, haben wir uns für die reine VBA-Variante entschieden – auch vor dem Hintergrund, dass Sie so nur ein einziges Standardmodul in Ihre Anwendung kopieren müssen, wenn Sie die dynamische Ermittlung der Feiertage verwenden möchten. Der Einsatz einer reinen VBA-Lösung bedeutet zunächst, dass einige grundlegende Informationen in Enumerationen gespeichert werden. Die erste Enumeration heißt **eBundesland** und nimmt alle Bundesländer in Form entsprechender Konstanten und Zahlenwerte auf. Sie sieht wie folgt aus:

```
Public Enum eBundesland
    eBadenWuerttemberg = 1
    eBayern = 2
    eBerlin = 4
    eBrandenburg = 8
    eBremen = 16
    eHamburg = 32
    eHessen = 64
    eMecklenburgVorpommern = 128
    eNiedersachsen = 256
    eNordrheinWestfalen = 512
    eRheinlandPfalz = 1024
    eSaarland = 2048
    eSachsen = 4096
    eSachsenAnhalt = 8192
    eSchleswigHolstein = 16384
    eThueringen = 32768
End Enum
```

Warum nun erhalten die Konstanten für die Bundesländer ausschließlich Zweierpotenzen als Zahlenwerte? Nun: So können wir einfach festlegen und ermitteln, welcher Feiertag in welchem Bundesland gefeiert wird. Dazu nutzen wir eine zweite Enumeration, welche die Feiertage auflistet und deren Zahlenwert Informationen darüber liefert, welche Bundesländer den jeweiligen Feiertag begehen. Diese Enumeration können Sie in Kurz- oder Langform verwenden. Hier ist eine abgekürzte Version der Langform:

```
Public Enum eFeiertageBundeslaender
    eNeujahr = eBadenWuerttemberg + eBayern + eBerlin _
        + eBrandenburg + eBremen + eHamburg + eHessen + _
            eMecklenburgVorpommern + eNiedersachsen + _
```

```
    eNordrheinWestfalen + eRheinlandPfalz + eSaarland + _
    eSachsen + eSachsenAnhalt + eSchleswigHolstein + _
                                        eThueringen
    ...
    eWeltkindertag = eThueringen
    eInternationalerFrauentag = eBerlin
End Enum
```

Wie gut zu erkennen ist, werden zu jedem Feiertag die Konstanten aller Bundesländer summiert, in denen der jeweilige Feiertag stattfindet. Da wir den Konstanten der Bundesländer Zweierpotenzen zugewiesen haben, können wir theoretisch statt der etwas längeren Ausdrücke auch einen Zahlenwert angeben. Für Feiertage, die in allen Bundesländern stattfinden, wäre dies etwa 32.768 + 16.384 + 8.192 + 4.096 + 2.048 + 1.024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1, also 65.535. Allerdings lassen sich die Informationen doch besser als Konstante prüfen.

Ermittlung des Datums des vierten Advents

Der vierte Advent ist der letzte Sonntag vor dem ersten Weihnachtstag und kann somit auch auf Heiligabend fallen. Der erste bis vierte Advent ist in allen Bundesländern Feiertag. Der erste, zweite und dritte Advent wird jeweils an den Sonntagen vor dem vierten Advent gefeiert, somit sind diese Feiertage vom Datum des vierten Advents abhängig. Auch der Buß- und Bettag, der allerdings nur in Sachsen Feiertag ist, hängt vom Datum des vierten Advents ab. Den vierten Advent berechnen wir so:

```
Function VierterAdvent(intJahr As Integer) As Date
    Dim dat As Date
    dat = CDate("24.12." & intJahr)
    Do While Not Weekday(dat, vbSunday) = vbSunday
        dat = dat - 1
    Loop
    VierterAdvent = dat
End Function
```

Diese Funktion stellt den Wert der Variablen **dat** einfach auf den 24. Dezember des Jahres ein, für welches das Datum des vierten Advents ermittelt werden soll, und wird in einer **Do While**-Schleife so lange um jeweils einen Tag in Richtung Jahresbeginn verschoben, bis es auf einen Sonntag fällt. Dies kann auch gleich beim ersten Durchlauf der Fall sein – dann fällt Heiligabend genau auf den vierten Advent. Die vom vierten Advent abhängigen Feiertage werden dann durch Subtraktion der entsprechenden Anzahl Tage ermittelt.

Ermittlung des Datums des Ostersonntags

Die Ermittlung des Ostersonntags erfolgt durch eine recht komplizierte Berechnung, die durch die folgende Funktion abgebildet wird:

```
Function Ostersonntag(intJahr As Integer) As Date
    Dim a As Integer
    Dim b As Integer
    Dim c As Integer
    Dim d As Integer
    Dim e As Integer
    Dim intTag As Integer
    Dim intMonat As Integer
    a = intJahr Mod 19
    b = intJahr Mod 4
    c = intJahr Mod 7
    d = (19 * a + 24) Mod 30
    e = (2 * b + 4 * c + 6 * d + 5) Mod 7
    intTag = 22 + d + e
    intMonat = 3
    If intTag > 31 Then
        intTag = d + e - 9
        intMonat = 4
    End If
    Ostersonntag = CDate(intTag & "." _
                                & intMonat & "." & intJahr)
End Function
```

Diese Funktion sorgt dafür, dass der Ostersonntag in Abhängigkeit von der Jahreszahl an einem Tag zwischen dem 23. März und dem 26. April liegt. Wenn man sich überlegt, dass Ostern (und die davon abhängigen Tage)

genau wie Weihnachten Jahrestage bestimmter Ereignisse sind, fragt man sich schon, warum hier keine einfachere Regelung gefunden wurde ...

Array der Feiertage zusammenstellen

Die grundlegenden Funktionen und Enumerationen haben wir nun zusammengestellt. Es fehlt noch eine Funktion, die alle Informationen zusammenführt und ein Array zurückliefert, das alle Feiertagsdaten für ein per Parameter angegebenes Jahr enthält. Ein zweiter Parameter gibt an, für welches Bundesland die Feiertage zusammengestellt werden sollen. Dies alles erledigt die Funktion **FeiertageArray** aus Listing 1. Der erste Parameter erwartet die Angabe einer Jahreszahl, also beispielsweise **2022**, der zweite Parameter eine der Konstanten der Enumeration der Bundesländer, also etwa **eNordrheinWestfalen**.

Die Funktion soll ein Array mit den Feiertagen zurückliefern, wobei der erste Wert den Namen des Feiertags enthält und der zweite das Datum des Feiertags. Die Daten der beiden Stichtage (Ostersonntag und der vierte Advent) werden in den Variablen **datOstersonntag** und **datVierterAdvent** gespeichert und später zur Berechnung der davon abhängigen Feiertage herangezogen. Die weiter oben vorgestellten Funktionen **Ostersonntag** und **VierterAdvent** füllen die beiden Datumsvariablen **datOstersonntag** und **datVierterAdvent** gleich zu Beginn der Funktion.

Dann prüft die Prozedur für jeden Feiertag, ob dieser in dem mit dem Parameter **lngBundesland** übergebenen Bundesland begangen wird. Dabei macht sich die Funktion die Enumerationen zunutze: Ein Ausdruck wie **(eNeujahr And lngBundesland) = lngBundesland** liefert so etwa den Wert **True** zurück, wenn der Feiertag im angegebenen Bundesland gefeiert wird. Wie funktioniert das? **eNeujahr** entspricht genau wie **lngBundesland** einem Zahlenwert. **eNeuJahr** hat den Zahlenwert **65.535**, das Bundesland **eNordrheinWestfalen** entspricht beispielsweise dem Wert **1**. Dadurch, dass wir den Bundesländern Zweierpotenzen als Wert zugewiesen haben, können wir durch einen Ausdruck wie **eNeujahr And lngBundesland** ermitteln, ob **eNordrheinWestfalen** in **eNeuJahr** enthalten ist. Diese arithmetische beziehungsweise binäre **Und**-Verknüpfung liefert genau den Wert der gemeinsamen Zweierpotenz, in diesem Fall **1** – dieses Bundesland begeht also diesen Feiertag. Wenn dies der Fall ist, ruft die Funktion eine weitere Funktion namens **FeiertagHinzufuegen** mit einigen Parametern auf:

```
FeiertagHinzufuegen i, arr, "Neujahr", ISODatum("1.1." &
intJahr)
```

Der Parameter **i** enthält die laufende Nummer des aktuell abgearbeiteten Feiertags, **arr** ist das Array mit der zu füllenden Liste der Feiertage, der dritte Parameter enthält die Bezeichnung des Feiertags und der vierte das mit der Funktion **ISODatum** im Format **#yyyy/mm/dd#** formatierte Datum. Die Funktion **FeiertagHinzufuegen** sieht so aus:

```
Public Function FeiertagHinzufuegen(i, arr() As String, _
                strFeiertag As String, strDatum As String)
    ReDim Preserve arr(2, i)
    arr(0, i) = strFeiertag
    arr(1, i) = strDatum
    i = i + 1
End Function
```

Die Funktion fügt lediglich den Namen des Feiertags und das Datum zum Array hinzu und erhöht den Zähler **i** um **1**. Dadurch, dass alle Parameter standardmäßig mit **ByRef** übergeben werden, spiegeln sich die Änderungen direkt in den Variablen der aufrufenden Funktion wider.

Verwenden des Arrays

Das resultierende Array können Sie auf verschiedene Arten weiternutzen. Wenn Sie beispielsweise alle Feiertage einfach im Direktfenster ausgeben möchten, verwenden Sie die folgende Prozedur:

```
Public Function FeiertageAusgeben()
    Dim arr As Variant
```

Die Eval-Funktion

Die Eval-Funktion erlaubt das Auswerten von Ausdrücken, die als Parameter an diese Funktion übergeben werden. Damit können Sie sich verschiedene Anwendungszwecke erschließen – zum Beispiel die Eingabe von Berechnungen in einfache Textfelder oder das Ermitteln von Eigenschaften der Benutzeroberflächenelemente ohne Verwendung des VBA-Editors. Dieser Beitrag zeigt die Möglichkeiten der Eval-Funktion auf.

Eval unter VBA

Eine der einfachsten Einsatzmöglichkeiten der **Eval**-Funktion ist das Berechnen eines Ausdrucks, den Sie als Parameter der **Eval**-Funktion übergeben. Damit lässt sich dann beispielsweise die Summe aus **1** und **2** ermitteln:

```
Debug.Print Eval("1+2")
 3
```

Sie sehen schon am ersten Beispiel, dass wir den zu berechnenden Ausdruck in Anführungszeichen einfassen. Das ist erforderlich, weil die **Eval**-Funktion eigentlich nur Parameter des Typs **String** entgegennimmt. Bei als numerisch zu interpretierenden Parameterwerten, zum Beispiel einfache Zahlenwerte wie **12**, interpretiert die Funktion dies korrekt. Auch Berechnungen wie **1+2** werden noch ohne Angabe von Anführungszeichen als numerische Werte erkannt und toleriert.

Wenn Sie jedoch beispielsweise Funktionen wie **Date()** aufrufen, erhalten Sie ohne Anführungszeichen eine Fehlermeldung (Laufzeitfehler **2040**, **Der von Ihnen eingegebene Ausdruck enthält eine ungültige Zahl.**).

Das Berechnen von **1+2** ist nun noch kein Hexenwerk, denn das bekommen Sie auch ohne **Eval** hin:

```
Debug.Print 1+2
 3
```

Aber vielleicht haben Sie eine Funktion, die zwei **Integer**-Zahlen addiert:

```
Public Function Addieren(int1 As Integer, _
                         int2 As Integer) As Integer
    Addieren = int1 + int2
End Function
```

Dann können Sie diese ebenfalls per **Eval** aufrufen:

```
Public Sub EvalAddieren()
    Debug.Print Eval("Addieren(1,2)")
End Sub
```

Berechnung per InputBox

Wenn Sie dem Benutzer sehr einfach eine Berechnungsmöglichkeit bereitstellen wollen, gelingt dies mit der folgenden Prozedur. Diese fragt per **InputBox** den zu berechnenden Ausdruck ab und gibt das Ergebnis aus:

```
Public Sub EvalPerInput()
    Dim strEval As String
    strEval = InputBox("Zu berechnender Ausdruck:")
```

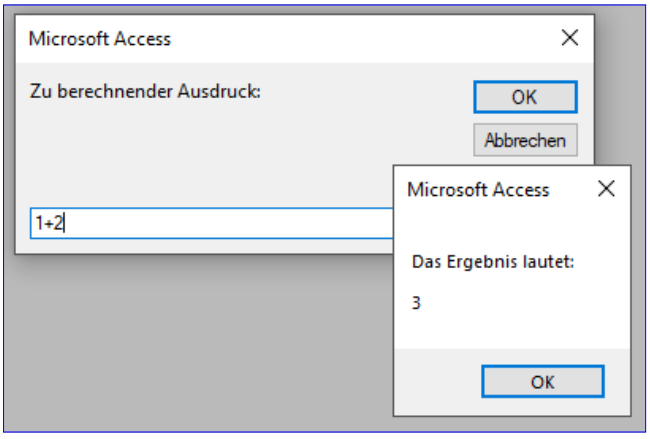


Bild 1: **Eval** per **InputBox**-Funktion

```
    MsgBox "Das Ergebnis lautet: " & vbCrLf & vbCrLf _
                                        & Eval(strEval)
End Sub
```

Ein Beispiel für die Nutzung dieser Prozedur sehen Sie in Bild 1.

Berechnungen in Textfeldern

Eine weitere Möglichkeit für den Einsatz der **Eval**-Funktion sind Textfelder. Dazu legen wir ein neues, leeres Formular an und fügen diesem ein Textfeld namens **txtAusdruck** zum Eingeben des zu berechnenden Ausdrucks hinzu. Außerdem legen wir ein zweites Textfeld namens **txtErgebnis** an. Für das Ereignis **Nach Aktualisierung** des Textfeldes **txtAusdruck** hinterlegen wir die folgende Prozedur, welche den Inhalt des Textfeldes **txtAusdruck** mit der **Eval**-Funktion berechnet und in das Textfeld **txtErgebnis** schreibt:

```
Private Sub txtAusdruck_AfterUpdate()
    Me!txtErgebnis = Eval(Me!txtAusdruck)
End Sub
```

Damit braucht der Benutzer nur den zu berechnenden Ausdruck in das erste Textfeld einzugeben und die Eingabetaste zu betätigen, um den Ausdruck zu berechnen (siehe Bild 2).

Zugriff auf Formulare, Steuerelemente und ihre Eigenschaften

Mit diesem Textfeld können Sie auch einfach auf Formulare, Steuerelemente und ihre Eigenschaften zugreifen. Um beispielsweise den Namen des aktiven Steuerelements zu ermitteln, geben Sie den folgenden Ausdruck ein:

```
Screen.ActiveControl.Name
```

Das Ergebnis sehen Sie in Bild 3.

Damit können Sie auch auf andere Formulare oder Steuerelemente zugreifen – Sie müssen lediglich die entsprechende Auflistung und die passenden Objektnamen verwenden.

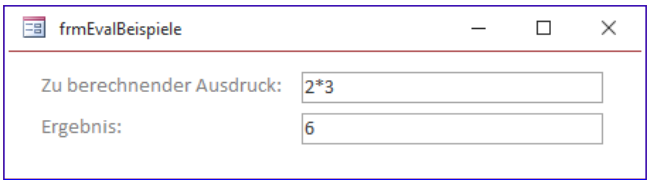


Bild 2: Ausdruck im Textfeld berechnen

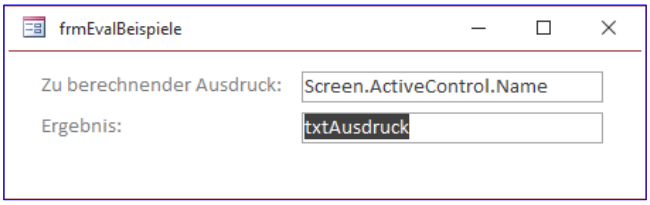


Bild 3: Ausgabe des Namens des aktiven Steuerelements

Eingebaute VBA-Funktionen nutzen

Weiter oben am Beispiel der Funktion **Addieren** haben Sie bereits gesehen, wie Sie eine benutzerdefinierte Funktion mit der **Eval**-Funktion auswerten können. Das gelingt übrigens auch mit den eingebauten VBA-Funktionen, zum Beispiel zum Ermitteln des Datums:

```
? Eval("Date()")
05.01.2022
```

Auch das ist wieder nichts Besonderes, denn dieses Ergebnis erhalten Sie auch ohne **Eval**-Funktion:

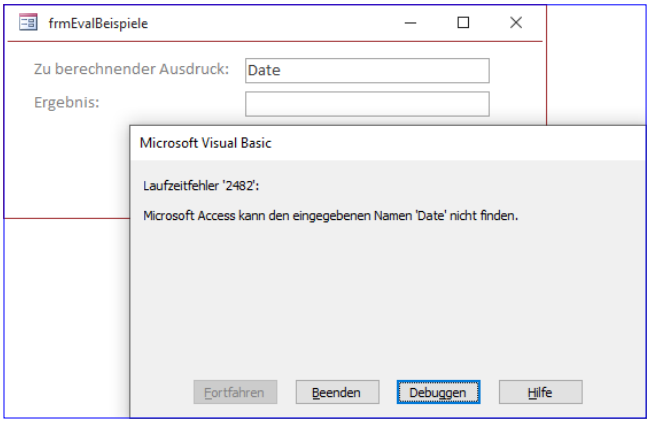


Bild 4: Fehler beim Aufruf einer Funktion ohne Klammernpaar

VBA-Projekt per VBA referenzieren

Access bietet nicht nur die Möglichkeit, Tabellen, Abfragen, Formulare und Berichte per VBA-Code zu erstellen. Sie können auch die Elemente, die Sie im VBA-Editor bearbeiten, per VBA erstellen, bearbeiten und wieder löschen. Dieser Beitrag macht den Start in eine Beitragsreihe, die sich mit den Möglichkeiten zur Programmierung des VBA-Editors und von VBA-Code beschäftigen. In diesem Teil schauen wir uns an, wie Sie überhaupt VBA-Projekte mit VBA referenzieren.

Voraussetzung: Extensibility-Verweis

Wenn Sie Routinen erschaffen wollen, mit denen Sie wiederum VBA-Elemente und -Code manipulieren möchten, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**.

Diesen fügen Sie im VBA-Editor mit dem **Verweise**-Dialog hinzu, den Sie mit dem Menübefehl **Extras|Verweise** öffnen (siehe Bild 1).

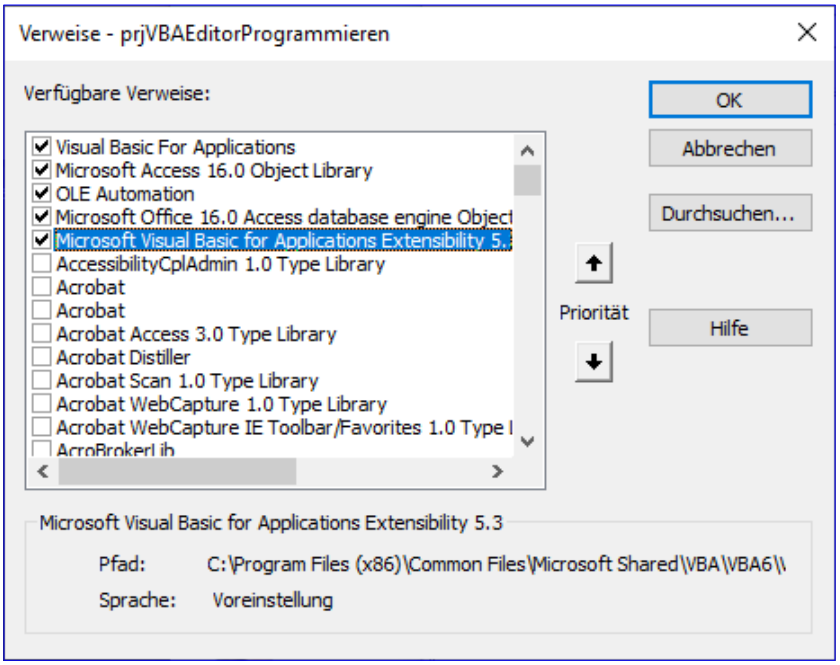


Bild 1: Verweis auf die **Extensibility**-Bibliothek

Access-Datenbank und VBA-Projekt

Eines der praktischen Dinge an Access ist, dass sowohl die Daten als auch die Elemente der Benutzeroberfläche und die Anwendungslogik normalerweise in einer einzigen Datei stecken. Natürlich gibt es Ausnahmen, wo die Tabellen in eine Backend-Access-Datei ausgelagert wurden oder wo die Tabellen Teil einer SQL Server-Datenbank sind. Und es gibt auch noch die Möglichkeit, dass Sie in einer Datenbankanwendung Elemente aus anderen Datenbankdateien nutzen – zum Beispiel, wenn Sie VBA-Code in eine Bibliotheksdatenbank auslagern.

Was jedoch immer garantiert ist, dass eine Access-Datenbankdatei auch ein VBA-Projekt enthält. Das VBA-Projekt speichert dabei die Module und Klassenmodule der Datenbankanwendung, wobei es zwei Typen von Klassenmodulen gibt – alleinstehende oder solche, die zu einem Formular oder einem Bericht gehören.

Derweil muss nicht jedes Formular und jeder Bericht ein Klassenmodul enthalten. Ein Klassenmodul zu einem Formular oder Bericht wird von Access erst durch eine der beiden folgenden Aktionen angelegt:

- wenn Sie die Eigenschaft **Enthält Modul** des Formulars oder Berichts auf **Ja** einstellen oder

- wenn Sie für das Formular oder den Bericht oder ein darin enthaltenes Steuerelement eine Ereignisprozedur definieren.

Aktuelles VBA-Projekt referenzieren

Um das aktuelle VBA-Projekt zu referenzieren, bedarf es keiner besonderen Tricks. Wir nutzen dazu die folgende Technik, nachdem wir den Namen unseres Beispielprojekts auf **prjVBAEditorProgrammieren** eingestellt haben.

Das erledigen Sie über den **Eigenschaften**-Bereich, während Sie im Projektexplorer den Eintrag für das Projekt der aktuellen Datenbank markiert haben (siehe Bild 2).

Danach probieren Sie den folgenden Code aus:

```
Public Sub VBProjectReferenzieren()
    Dim objVBProject As VBProject
    Set objVBProject = VBE.ActiveVBProject
    Debug.Print objVBProject.Name
End Sub
```

Die Prozedur deklariert eine Variable des Typs **VBProject** und füllt diese mit der Eigenschaft **ActiveVBProject** der Klasse **VBE**. Die Klasse **VBE** ist das oberste Element der Bibliothek zur Programmierung des Visual Basic Editors.

Nachdem wir die Variable **objVBProject** gefüllt haben, können wir damit beispielsweise auf den Namen des Projekts zugreifen. Diesen gibt die Prozedur im vorliegenden Fall wie in Bild 3 aus.

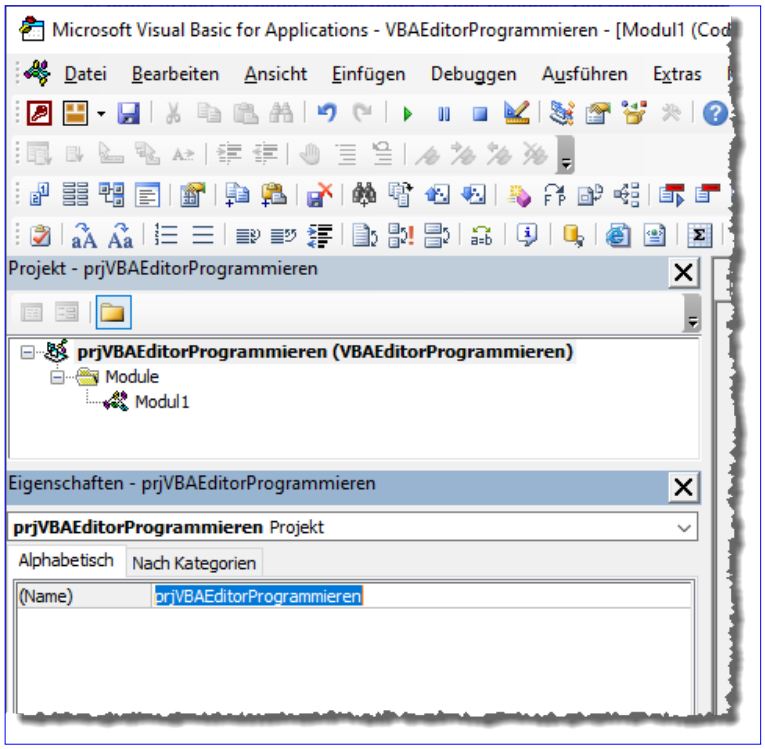


Bild 2: Ändern des Namens des VBA-Projekts

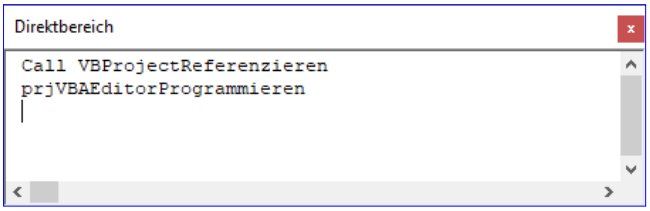


Bild 3: Ausgabe des Namens des VBA-Projekts

Mehrere VBA-Projekte im VBA-Editor?

Aber referenzieren wir mit der Eigenschaft **ActiveVBProject** auch tatsächlich immer das **VBProject**-Objekt der aktuellen Datenbank?

Nein, das ist nicht sichergestellt. Um das zu reproduzieren, wechseln Sie einmal zum Access-Fenster und legen ein neues, leeres Formular in der Entwurfsansicht an. Dann klappen Sie im Ribbon unter **Entwurf|Steuerelemente** die Liste der Steuerelemente auf und aktivieren dort die Option **Steuerelement-Assistent verwenden** (siehe Bild 4).

Danach fügen Sie beispielsweise eine neue Schaltfläche hinzu, was den dazugehörigen Assistenten aktiviert. Das war es schon auf der Access-Seite – Sie können den Assistenten nun abbrechen und wieder zum VBA-Editor wechseln.

Hier finden Sie nun plötzlich zwei VBA-Projekte vor – zusätzlich zu dem der aktuellen Access-Datenbank finden

Zugriff auf den VBA-Editor mit der VBE-Klasse

Die VBE-Klasse ist die Schaltzentrale, wenn es darum geht, die Elemente des VBA-Editors und von VBA-Projekten per VBA zu programmieren. Die Klasse ist Teil einer eigenen Bibliothek namens Microsoft Visual Basic for Applications Extensibility 5.3 Object Library. Diese stellt alle Elemente, Methoden und Eigenschaften zur Verfügung, um die im VBA-Editor bearbeitbaren Elemente zu erstellen, zu bearbeiten oder zu löschen. Dieser Beitrag stellt die Eigenschaften und Auflistungen der VBE-Klasse vor und zeigt, wo Sie weitergehende Informationen zu den einzelnen Elementen finden.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Die VBE-Klasse

Um sich einen Überblick über die Elemente einer Klasse zu verschaffen, ist der Objektkatalog (zu öffnen mit **F2**) immer eine gute Anlaufstelle. Hier wählen Sie oben den Eintrag **VBIDE** und selektieren dann links unter **Klassen** den Eintrag **VBE**. Die Elemente der Klasse erscheinen dann im rechten Bereich (siehe Bild 1). Im Einzelnen finden wir dort die folgenden Elemente vor:

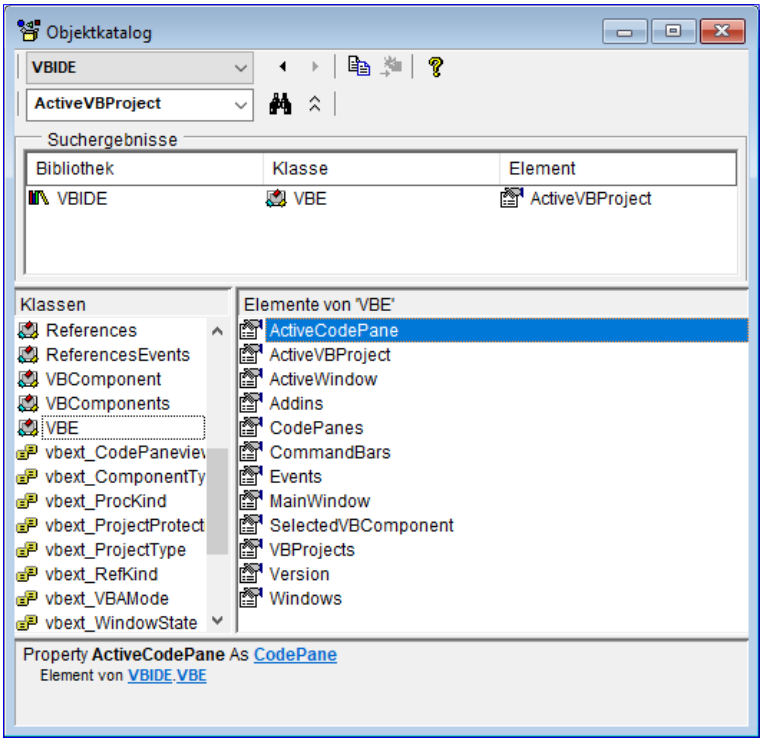


Bild 1: Elemente der VBE-Klasse im Objektkatalog

- **ActiveCodePane**: Verweis auf das **CodePane**-Element, das aktuell den Fokus hat.
- **ActiveVBProject**: Verweis auf das aktive VBA-Projekt, also das Projekt, von dem aus der Aufruf erfolgt.
- **ActiveWindow**: Verweis auf das Window-Objekt, das im VBA-Fenster aktuell den Fokus hat.
- **AddIns**: Auflistung der aktuell verfügbaren Add-Ins im VBA-Editor (nicht zu verwechseln mit den Access-Add-Ins)
- **CodePanes**: Auflistung der **CodePane**-Objekte, die aktuell geöffnet sind
- **CommandBars**: Auflistung der **CommandBar**-Objekte des VBA-Editors
- **Events**: Ermöglicht den Zugriff auf Ereignisse von **CommandBar**- und **Reference**-Elementen
- **MainWindow**: Verweis auf das VBA-Fenster

- **SelectedVBComponent**: Gibt das aktuell im Projektexplorer selektierte Element zurück, falls es sich um ein Element des Typs **VBComponent** handelt – anderenfalls lautet das Ergebnis **Nothing**.

- **VBProjects**: Liefert eine Auflistung der **VBProject**-Elemente, die aktuell im Projektexplorer des VBA-Editors angezeigt werden.

- **Version**: Gibt die Version des VBA-Editors aus.

- **Windows**: Liefert eine Auflistung der **Window**-Elemente des VBA-Editors.

Unterschied zwischen Window, CodePane und VBComponent

In der Auflistung haben wir bereits mehrere Elemente kennengelernt, die per Auflistung und auch jeweils als aktives Element ermittelt werden können. Dabei sind die **VBComponent**-Elemente unter Access die Elemente, die Sie im Projektexplorer in den Bereichen **Microsoft Access Klassenobjekte**, **Module** und **Klassenmodule** finden (siehe Bild 2).

Window-Elemente sind alle Fenster, die innerhalb des VBA-Editors angezeigt werden. Dabei handelt es sich nicht nur um die Fenster, die Code enthalten, sondern auch die **ToolWindow**-Elemente, also die Fenster, die Sie links, rechts, oben oder unten verankern können. **CodePane**-Elemente sind Container in Fenstern, die Code von **VBComponent**-Elementen enthalten. Die Zusammenhänge werden wir in den folgenden Beispielen noch aufschlüsseln.

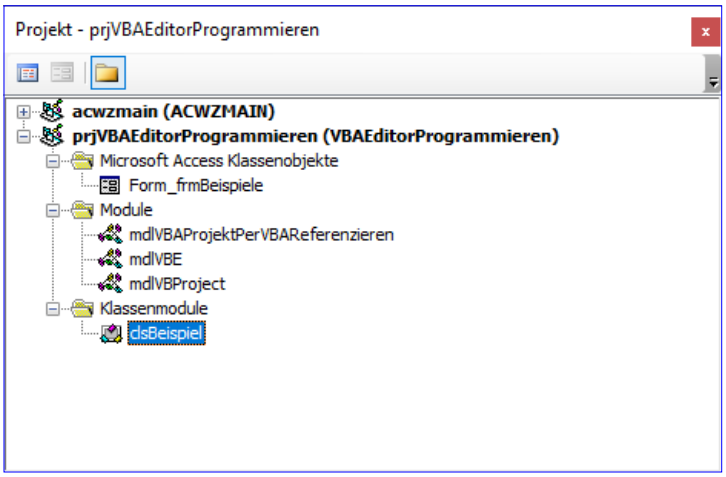


Bild 2: Die verschiedenen **VBComponent**-Elemente

Mit Window-Elementen arbeiten

Mit der **Windows**-Auflistung können wir alle **Window**-Elemente des VBA-Editors durchlaufen. Das erledigen wir in der folgenden Prozedur:

```
Public Sub WindowsAuflisten()
    Dim objWindow As Window
    For Each objWindow In VBE.Windows
        Debug.Print objWindow.Caption, objWindow.Type
    Next objWindow
End Sub
```

Dies erzeugt, wenn einige Codefenster geöffnet sind, die Ausgabe aus Bild 3. Uns interessiert nun, welche Bedeutung die Werte für die Eigenschaft **Type** haben.

Dazu ermitteln wir die Konstanten für diese Eigenschaft, was am schnellsten gelingt, wenn wir im Objektkatalog nach der Klasse **VBIDE** filtern und dort nach Elementen mit dem Teilausdruck **Type** suchen. Damit gelangen wir zur Auflistung **vbext_WindowType**, welche die Werte aus Bild 4 offenbart.

Damit können wir die Prozedur **WindowsAuflisten** wie folgt verfeinern:

```
Public Sub WindowsAuflisten()
    Dim objWindow As Window
    Dim strWindowType As String
    For Each objWindow In VBE.Windows
        Select Case objWindow.Type
            Case 0
                strWindowType = "CodeWindow"
            Case 1
```

Zugriff auf VBA-Projekte per VBProject

Die Klasse VBProject des Objektmodells zum Programmieren des VBA-Editors und der enthaltenen Elemente bietet einige interessante Eigenschaften, Methoden und Auflistungen. Diese schauen wir uns im vorliegenden Beitrag an. Hier wird deutlich, dass die VBProjects im VB-Editor nicht nur für Access-Datenbanken genutzt werden können, sondern auch noch für andere Anwendungen – es gibt nämlich einige Elemente, die unter Access nicht funktionieren.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Elemente der VBProject-Klasse

Die **VBProject**-Klasse liefert die folgenden Methoden, Auflistungen und Eigenschaften:

- **BuildFileName**: Gibt den Namen einer DLL zurück. Diese Eigenschaft hat unter Access keine Verwendung, da hier keine DLL erstellt werden kann.

- **Collection**: Liefert einen Verweis auf die Collection, in der sich das **VBProject**-Element befindet. Diese sollte normalerweise nur ein Element enthalten, nämlich das aktuelle VBA-Projekt. Wenn Sie jedoch beispielsweise seit dem Start der aktuellen Access-Session ein Access-Add-In verwendet haben, finden Sie auch dessen **VBProject**-Element in der Auflistung.

- **Description**: Diese Eigenschaft ist standardmäßig leer und kann in den Projekteigenschaften eingestellt werden (siehe weiter unten).

- **FileName**: Liefert den Pfad zu der Datei, in der das aktuelle VBProject-Objekt gespeichert ist – in der Regel also die Datenbankdatei, von der aus Sie den VBA-Editor mit dem aktuellen Projekt geöffnet haben.

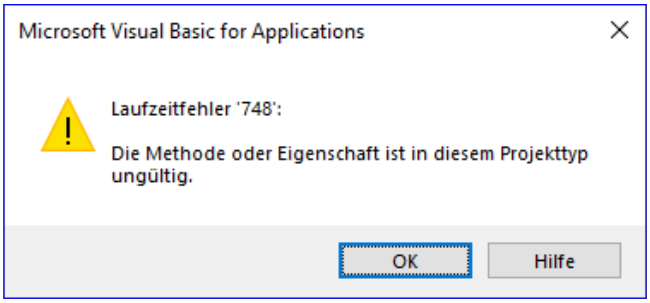


Bild 1: Fehler beim Aufruf der Methode **MakeCompiledFile**

- **MakeCompiledFile**: Führt beim Aufruf aus einem VBA-Projekt zu dem Fehler aus Bild 1 und ist laut Dokumentation zum Erstellen einer DLL für das aktuelle Projekt gedacht – was aber wohl nicht für VBA-Projekte von Access-Datenbanken gilt.

- **Mode**: Gibt den Modus des Projekts an. Kann die drei Werte **vbext_vm_Run** (**0**), **vbext_vm_Break** (**1**) oder **vbext_vm_Design** (**2**) annehmen. Auch diese Eigenschaft lässt sich in VBA-Projekten unter Access nicht sinnvoll nutzen. Wenn Sie einen Haltepunkt setzen, dieser im Code erreicht wird und Sie dann im Direktbereich **Debug.Print VBE.ActiveVBProject.Mode** ausgeben lassen, erhalten Sie dennoch den Wert **0**.

- **Name**: Liefert den Namen des VBA-Projekts, so wie er auch in den Eigenschaften festgelegt ist.

- **Protection**: Kann die Werte **vbext_pp_locked** (**1**) oder **vbext_pp_none** (**0**) annehmen. Standardmäßig ist der Wert **0** eingestellt. Diesen Wert ändern Sie durch Vergeben eines Kennworts für das VBA-Projekt.

- **References**: Liefert eine Auflistung der Verweise des aktuellen VBA-Projekts.
- **SaveAs**: Nicht für Access-VBA-Projekte verfügbar.
- **Saved**: Gibt an, ob es noch ungespeicherte Änderungen im VBA-Projekt gibt. **True** bedeutet, dass alle Änderungen gespeichert sind, **False**, dass noch nicht gespeicherte Änderungen vorliegen.
- **Type**: Gibt den Typ des VBA-Projekts zurück. Es gibt die beiden Werte **vbext_pt_HostProject** (**100**) und **vbext_pt_Standalone** (**101**).
- **VBComponents**: Liefert eine Auflistung aller **VBComponent**-Objekte.
- **VBE**: Verweis auf das übergeordnete VBE-Objekt

Projekteigenschaften einstellen

Die weiter oben erwähnte Eigenschaft **Description** ist standardmäßig leer. Sie können diese füllen, indem Sie den Dialog **[Projektname] - Projekteigenschaften** öffnen, was Sie mit dem Menüeintrag **Extras|Eigenschaften von [Projektname]...** erledigen. Hier finden Sie auf der ersten Seite gleich die Eigenschaft **Projektbeschreibung**, die Sie mit einem beliebigen Text füllen können (siehe Bild 2).

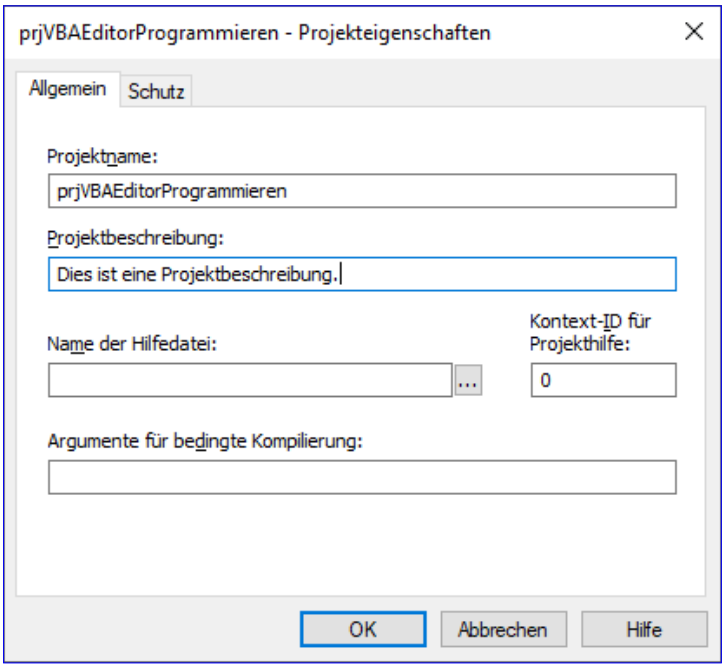


Bild 2: Einstellen der Beschreibung eines VBA-Projekts

Anschließend rufen Sie diesen Text mit der folgenden Anweisung beispielsweise im Direktbereich ab:

```
? VBE.ActiveVBProject.Description
Dies ist eine Projektbeschreibung.
```

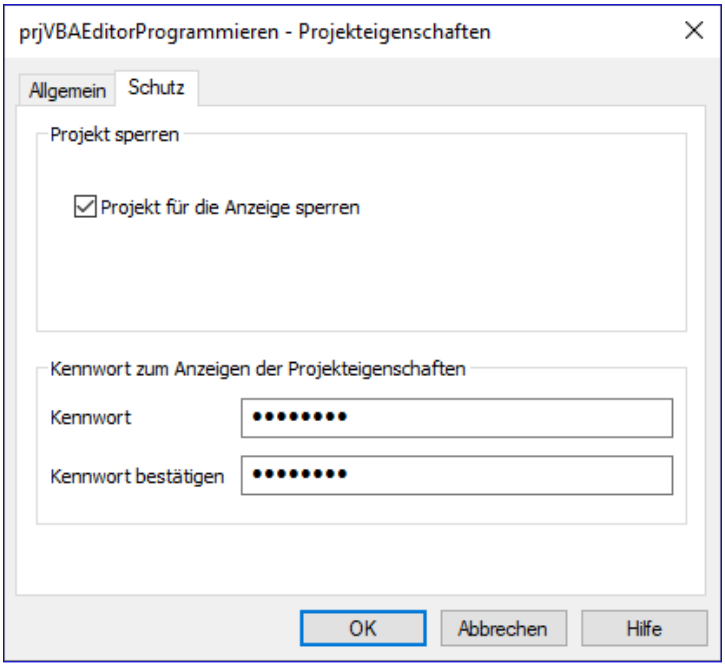


Bild 3: Festlegen des Kennwortschutzes

Kennwortschutz aktivieren und abfragen

Den Kennwortschutz aktivieren Sie im gleichen Dialog, in dem Sie auch die Projektbeschreibung eingeben – allerdings auf der zweiten Registerseite unter **Schutz**. Um den Schutz zu aktivieren, setzen Sie einen Haken für die Option **Projekt für die Anzeige** sperren (siehe Bild 3). Danach können Sie unten das Kennwort eingeben und nochmals bestätigen.

Module und Co. im Griff mit VBComponent

Bei der Programmierung des VBA-Editors per VBA ist eine der Kernkomponenten das Element VBComponent. Wir können diese mit der Auflistung VBComponents durchlaufen oder direkt über den Namen der Komponente oder den Index darauf zugreifen. Danach ergeben sich verschiedene Möglichkeiten, die erst mit dem Zugriff auf das CodeModule des VBComponent-Elements interessant werden. Bis dahin schauen wir uns aber noch an, welche Möglichkeiten das VBComponent-Element bietet.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Elemente der VBComponent-Klasse

Die **VBComponent**-Klasse und ihre Eigenschaften, Methoden und Auflistungen können Sie im Objektkatalog im Überblick ansehen, wenn Sie dort nach **VBComponent** suchen (siehe Bild 1). Die Klasse bietet folgende Elemente:

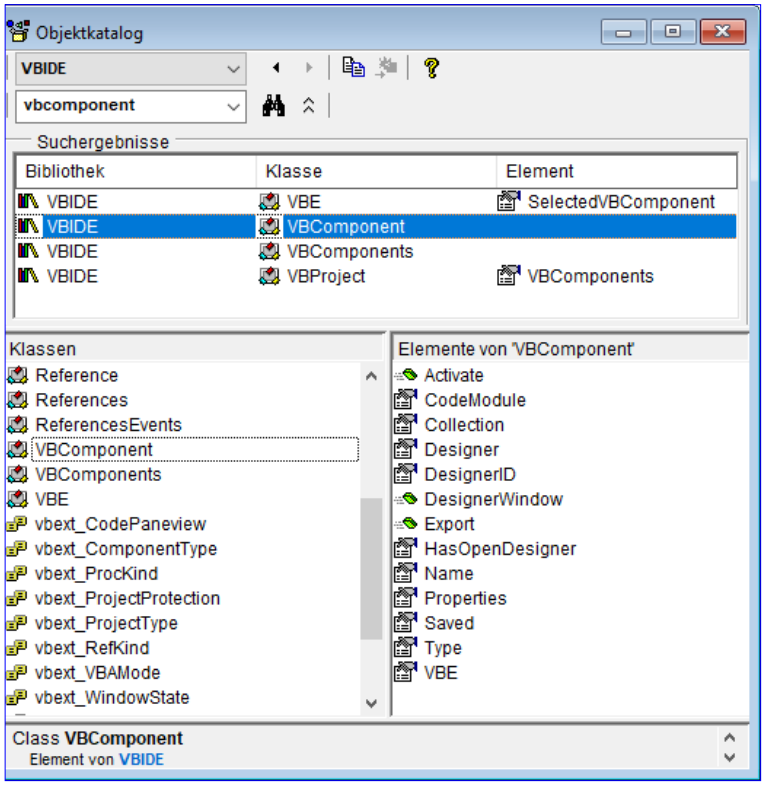


Bild 1: Das **VBComponent**-Element im Objektkatalog

- **Activate**: Aktiviert das **VBComponent**-Objekt und zeigt es in seinem Fenster an.
- **CodeModule**: Liefert einen Verweis auf das **CodeModule**-Element des **VBComponent**-Objekts.
- **Collection**: Liefert Zugriff auf die Collection mit diesem **VBComponent**-Objekt und allen anderen, die in der übergeordneten **VBComponents**-Auflistung enthalten sind.
- **Designer**: Liefert einen Verweis auf den Designer des **VBComponent**-Objekts. Dieser liefert bei den üblicherweise unter Access verwendeten **VBComponent**-Elementen immer **Nothing**. Sie können ihn nutzen, wenn Sie beispielsweise **UserForm**-Objekte programmieren wollen, was nicht Thema dieses Beitrags ist und normalerweise unter Access nicht geschieht.
- **DesignerID**: Siehe Eigenschaft **Designer**.
- **DesignerWindow**: Siehe Eigenschaft **Designer**.
- **Export**: Exportiert das Objekt in eine Textdatei.

- **HasOpenDesigner**: Siehe **Designer**. Liefert daher für die standardmäßig verwendeten Access-Module immer den Wert **False**.
- **Name**: Liefert den Namen des **VBComponent**-Objekts.
- **Properties**: Liefert eine Auflistung der **Property**-Eigenschaften des **VBComponent**-Elements.
- **Saved**: Gibt an, ob es ungespeicherte Änderungen an dem **VBComponent**-Objekt gibt.
- **Type**: Gibt den Typ des **VBComponent**-Objekts zurück.
- **VBE**: Verweis auf das VBA-Editor-Objekt des **VBComponent**-Objekts

VBComponent-Objekte durchlaufen

Die folgende Prozedur referenziert mit der Variablen **objVBProject** das aktuelle VBA-Projekt und durchläuft dann in einer **For Each**-Schleife die Elemente der Auflistung **VBComponents**. Dabei speichert sie den Verweis auf das jeweils aktuelle Objekt in der Variablen **objVBComponent**. Damit gibt die Prozedur den Namen des **VBComponent**-Objekts aus:

```
Public Sub VBComponentsDurchlaufen()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Set objVBProject = VBE.ActiveVBProject
    For Each objVBComponent In objVBProject.VBComponents
        Debug.Print objVBComponent.Name
    Next objVBComponent
End Sub
```

Das Ergebnis entspricht den Elementen, die Sie in den drei Ordnern **Microsoft Access Klassenobjekte**, **Module** und **Klassenmodule** des Projektexplorers sehen, zum Beispiel:

```
mdlVBAProjektPerVBAReferenzieren
mdlVBProject
mdlVBE
Form_frmBeispiele
clsBeispiel
mdlVBComponent
```

VBComponent-Objekte gezielt referenzieren

Wenn Sie wissen, auf welches **VBComponent**-Objekt Sie zugreifen wollen, beispielsweise um sein **CodeModule**-Objekt zu bearbeiten, können Sie dies auch über den Index oder den Namen erledigen. Der Index ist 1-basiert:

```
? VBE.ActiveVBProject.VBComponents(1).Name
mdlVBAProjektPerVBAReferenzieren
```

Das Referenzieren per Name geht so – hier mit Ausgabe des Wertes der Eigenschaft **Saved**:

```
? VBE.ActiveVBProject.VBComponents("mdlVBE").Saved
Wahr
```

Neues VBComponent-Objekt hinzufügen

Um ein neues **VBComponent**-Objekt hinzuzufügen und somit ein Klassenmodul eines Formulars oder Berichts, ein Standardmodul oder ein alleinstehendes Klassenmodul, benötigen Sie die **Add**-Methode der **VBComponents**-Auflistung. Dieser übergeben Sie lediglich den Typ der zu erstellenden Komponente (siehe Bild 2).

Das Ergebnis referenzieren wir beispielsweise mit der Variablen **objVBComponent** und bearbeiten dieses anschließend weiter. Dabei geben wir im folgenden Beispiel zunächst nur den Namen der Komponente an:

```
Public Sub VBComponentNeu()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = objVBProject.VBComponents.
                                Add(vbext_ct_StdModule)
    With objVBComponent
        .Name = "mdlNeu"
```

VBA-Code manipulieren mit der CodeModule-Klasse

Wenn Sie sich mit den anderen Beiträgen dieser Reihe bis zum VBComponent-Objekt eines Moduls vorgearbeitet haben, ist es nur noch ein Katzensprung bis zur CodeModule-Klasse. Damit können Sie dann die Inhalte eines VBA-Moduls auslesen und bearbeiten. Dieser Beitrag zeigt, welche Methoden die CodeModule-Klasse bietet und wie Sie diese für die verschiedenen Anwendungszwecke einsetzen können.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Vorbereitende Beiträge

Wenn Sie erfahren wollen, wie Sie überhaupt bis zu der hier beschriebenen Klasse gelangen, helfen die folgenden Beiträge weiter:

- **VBA-Projekt per VBA referenzieren** (**www.access-im-unternehmen.de/1337**)
- **Zugriff auf den VBA-Editor mit der VBE-Klasse** (**www.access-im-unternehmen.de/1350**)
- **Zugriff auf VBA-Projekte per VBProject** (**www.access-im-unternehmen.de/1351**)
- **Module und Co. im Griff mit VBComponent** (**www.access-im-unternehmen.de/1352**)

Elemente der VBComponent-Klasse

Die **CodeModule**-Klasse und ihre Eigenschaften, Methoden und Auflistungen können Sie im Objektkatalog im Überblick ansehen, wenn Sie dort nach **CodeModule** suchen (siehe Bild 1). Hier sehen wir neben den Elementen dieser Klasse auch, dass diese sowohl ein Unterelement von **VBComponent** als auch von **CodePane** ist. Als Element von **VBComponent** können Sie es anlegen, und **CodePane** ist die Schnittstelle zwischen dem anzeigenden **Window**-Element im VBA-Editor und dem **CodeModule**-Objekt, das einige Möglichkeiten für den Zugriff auf den Code über die Benutzeroberfläche ermöglicht.

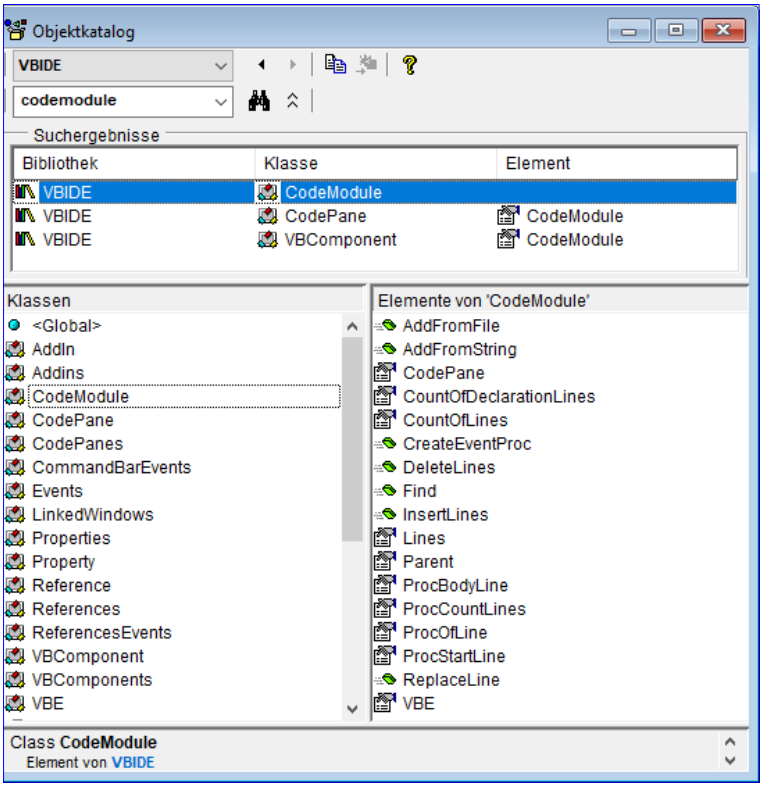


Bild 1: Die **CodeModule**-Klasse im Objektkatalog

Die Klasse **CodeModule** bietet folgende Elemente:

- **AddFromFile**: Fügt VBA-Code aus der per Parameter angegebenen Textdatei hinzu.

- **AddFromString**: Fügt VBA-Code aus der per Parameter übergebenen Zeichenkette hinzu.
- **CodePane**: Liefert einen Verweis auf das **CodePane**-Element, mit dem Sie speziell Zugriffsmöglichkeiten auf die angezeigte Version des **CodeModule**-Objekts haben – beispielsweise um Markierungen abzufragen oder zu setzen.
- **CountOfDeclarationLines**: Liefert die Anzahl der Zeilen im Deklarationsbereich des Moduls.
- **CountOfLines**: Liefert die gesamte Anzahl der Zeilen im Modul.
- **CreateEventProc**: Erstellt eine Ereignisprozedur für das mit den beiden Parametern angegebene Ereignis und Objekt.
- **DeleteLines**: Löscht die mit dem zweiten Parameter angegebene Anzahl von Zeilen ab der mit dem ersten Parameter angegebenen Zeile.
- **Find**: Sucht nach dem mit dem ersten Parameter angegebenen Ausdruck und liefert mit den folgenden Parametern die Position des Fundorts zurück.
- **InsertLines**: Fügt ab der mit dem ersten Parameter angegebenen Zeile den mit dem zweiten Parameter angegebenen Text im Modul ein.
- **Lines**: Liefert den Inhalt einer oder mehrerer Zeilen, wobei der erste Parameter die Nummer der ersten Zeile und der zweite die Anzahl der Zeilen angibt.
- **Parent**: Referenziert das übergeordnete Objekt, in diesem Fall ein Objekt des Typs **VBComponent**.
- **ProcBodyLine**: Liefert die Nummer der Zeile der angegebenen Prozedur mit dem **Sub|Function|Property**-Schlüsselwort.
- **ProcCountLines**: Liefert die Anzahl der Zeilen einer Prozedur.
- **ProcOfLine**: Gibt die Prozedur und den Typ der Prozedur für eine bestimmte Zeile zurück.
- **ProcStartLine**: Liefert die Nummer der ersten Zeile nach der letzten Zeile der vorherigen Prozedur oder des allgemeinen Deklarationsteils.
- **ReplaceLine**: Ersetzt die Zeile mit der im ersten Parameter angegebenen Nummer durch den mit dem zweiten Parameter angegebenen Text.
- **VBE**: Referenziert das VBA-Editor-Objekt des **CodeModule**-Objekts.

Das CodeModule-Objekt referenzieren

Um das **CodeModule**-Objekt eines Moduls zu referenzieren, benötigen Sie zuerst Zugriff auf das entsprechende **VBComponent**-Element. Meist wollen Sie direkt auf ein bestimmtes Objekt zugreifen, dessen Namen Sie kennen. Dann können Sie dieses über die **VBComponents**-Auflistung des **VBProject**-Elements referenzieren.

Das **VBComponent**-Element bietet dann über die **CodeModule**-Eigenschaft die Möglichkeit des Zugriffs auf das **CodeModule**-Objekt an. Im folgenden Beispiel referenzieren wir dieses und geben dann die Anzahl der Codezeilen in diesem **CodeModule**-Objekt aus:

```
Public Sub CodeModuleReferenzieren()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Dim objCodeModule As CodeModule
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = 7
                        objVBProject.VBComponents("mdlVBE")
    Set objCodeModule = objVBComponent.CodeModule
    Debug.Print objCodeModule.CountOfLines
End Sub
```

Code aus einer Textdatei hinzufügen

Mit der **Export**-Methode des **VBComponent**-Objekts können Sie eine Textdatei mit dessen Inhalt exportieren. Diese können Sie beispielsweise mit der **AddFromFile**-Methode der **CodeModule**-Klasse wieder einlesen.

Wie das gelingt, zeigen wir in folgendem Beispiel. Hier legen wir ein neues, leeres **VBComponent**-Objekt an und nutzen dann die **AddFromFile**-Methode seines **CodeModule**-Objekts, um den Inhalt des exportierten Moduls in das neue Modul einzulesen:

```
Public Sub CodeModuleAddFromFile()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Dim objCodeModule As CodeModule
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = ⤷
        objVBProject.VBComponents.Add(vbext_ct_StdModule)
    Set objCodeModule = objVBComponent.CodeModule
    objCodeModule.AddFromFile CurrentProject.Path ⤷
                                        & "\Neu.txt"
End Sub
```

Zu beachten ist hier, dass bei exportierten Modulen auch die Attribute mit exportiert werden, also zum Beispiel der Modulname.

Dieser wird beim Importieren in ein vorhandenes Modul dann für das übergeordnete **VBComponent**-Objekt verwendet.

Sie können mit **AddFromFile** jedoch auch Textdateien mit dem reinen Inhalt des Moduls laden.

Code per Zeichenkette hinzufügen

Gegebenenfalls stellen Sie den Code für ein Modul in einer Textvariablen zusammen oder lesen diesen von anderer Stelle ein, beispielsweise aus einem Feld einer Datenbanktabelle. Wenn Sie den Inhalt der Variablen schnell in ein neues, leeres Modul einfügen möchten, bietet sich die Methode **AddFromString** an. Diese schreibt den Code direkt in das **CodeModule**-Objekt.

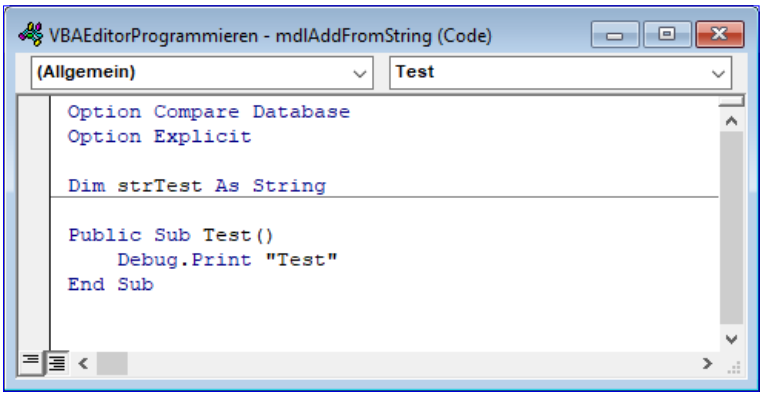


Bild 2: Per **AddFromString** initial hinzugefügter Code

Im folgenden Beispiel stellen wir den einzufügenden Code zuvor in der Variablen **strCode** zusammen und weisen diesen dann mit **AddFromString** dem **CodeModule**-Objekt zu:

```
Public Sub CodeModuleAddFromString()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Dim objCodeModule As CodeModule
    Dim strCode As String
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = objVBProject.VBComponents.⤷
                                    Add(vbext_ct_StdModule)
    objVBComponent.Name = "mdlAddFromString"
    Set objCodeModule = objVBComponent.CodeModule
    strCode = "Dim strTest As String" & vbCrLf & vbCrLf
    strCode = strCode & "Public Sub Test()" & vbCrLf
    strCode = strCode & "    Debug.Print ""Test""" ⤷
                                        & vbCrLf
    strCode = strCode & "End Sub"
    objCodeModule.AddFromString strCode
End Sub
```

Das Ergebnis finden Sie in Bild 2.

Sie können **AddFromString** auch nutzen, um Code in ein Modul einzufügen, das bereits Code enthält. Der neu einzufügende Code landet dann genau hinter der letzten Deklarationszeile im Modul.

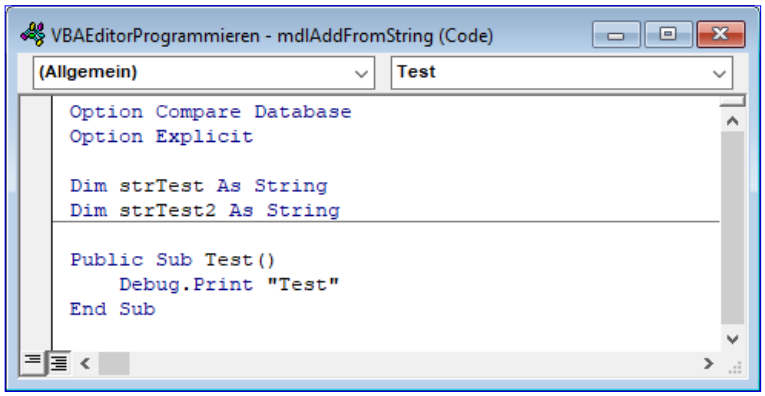


Bild 3: Per **AddFromString** nachträglich hinzugefügter Code

Das macht Sinn, denn wenn der einzufügende Code auch im oberen Bereich Deklarationszeilen und im unteren Routinen enthält, dann gibt es weiterhin eine Trennung zwischen den Deklarationen im oberen Bereich und der Programmlogik im unteren Bereich. Die folgende Prozedur fügt eine Deklarationszeile zum bestehenden Code aus dem vorherigen Beispiel hinzu:

```
Public Sub CodeModuleAddMoreFromString()
    Dim objVBProject As VBProject
    Dim objCodeModule As CodeModule
    Dim strCode As String
    Set objVBProject = VBE.ActiveVBProject
    Set objCodeModule = objVBProject.VBComponents(_
                            "mdlAddFromString").CodeModule
    strCode = "Dim strTest2 As String"
    objCodeModule.AddFromString strCode
End Sub
```

Dieser wird dann im Modul wie in Bild 3 eingefügt.

Das CodePane-Objekt eines Moduls referenzieren

Im Beitrag **Zugriff auf den VBA-Editor mit der VBE-Klasse** (**www.access-im-unternehmen.de/1350**) haben wir den Unterschied und die Zusammenhänge zwischen **VBComponent**, **Window** und **CodePane** erläutert. Das **CodePane** ist das Element, in dem das **CodeModule**-Objekt im **Window**-Objekt angezeigt wird.

Deshalb können wir vom **CodeModule**-Element über die Eigenschaft **CodePane** auch auf das betroffene **CodePane**-Element zugreifen. Das **CodePane**-Element referenzieren wir dabei wie folgt:

```
Public Sub CodePaneReferenzieren()
    Dim objVBProject As VBProject
    Dim objCodeModule As CodeModule
    Dim objCodePane As CodePane
    Set objVBProject = VBE.ActiveVBProject
    Set objCodeModule = objVBProject.VBComponents(_
                                "mdlCodeModule").CodeModule
    Set objCodePane = objCodeModule.CodePane
    '... Dinge mit dem CodePane erledigen
End Sub
```

Welche Möglichkeiten die **CodePane**-Klasse bietet, lesen Sie im Beitrag **Auf VBA-Code zugreifen per CodePane** (**www.access-im-unternehmen.de/1354**).

Schneller Zugriff auf das CodeModule per CodePane

Das **CodePane**-Objekt hat eine Eigenart, die wir uns in den folgenden Beispielen zunutze machen wollen: Sie können das aktive **CodePane**-Objekt, also den Container im aktuell aktiven VBA-Fenster mit dem **CodeModule**-Objekt, auch direkt mit der Eigenschaft **ActiveCodePane** der übergeordneten **VBE**-Klasse referenzieren.

Zeilen zählen im Modul

Es gibt einige Eigenschaften, mit denen Sie verschiedene Zeilenanzahlen ermitteln können.

Diese schauen wir uns in den nächsten Abschnitten an.

Anzahl aller Zeilen im Modul

Am einfachsten ist das Zählen aller Zeilen. Dies erledigen wir mit der Eigenschaft **CountOfLines**. Diese liefert die Anzahl der Zeilen von der ersten bis zur letzten Zeile des Moduls.

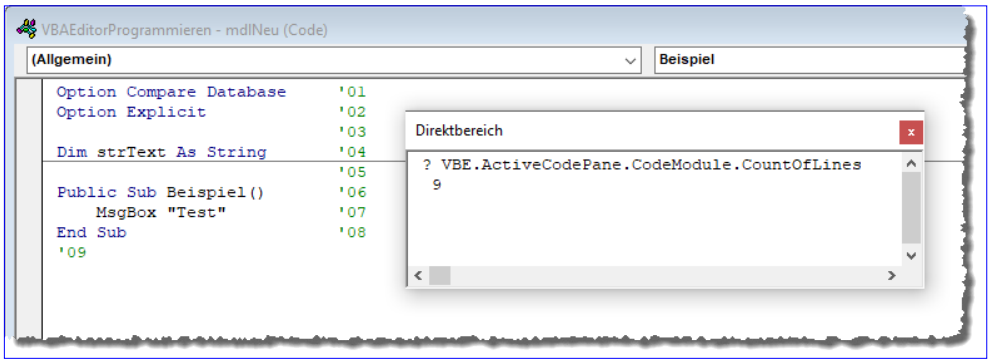


Bild 4: Abfragen von **CodeModule**-Eigenschaften per Direktbereich

Wir gehen davon aus, dass das zu untersuchende Modul geöffnet ist und dass das entsprechende VBA-Fenster den Fokus hat.

Dann können wir nämlich ganz einfach wie folgt über den Direktbereich auf zu die untersuchenden Eigenschaften zugreifen – hier auf die Anzahl der Codezeilen:

```
? VBE.ActiveCodePane.CodeModule.CountOfLines
 9
```

Die Eigenschaft **CountOfLines** liefert die Anzahl der Zeilen, wobei auch solche Zeilen ohne Inhalt mitgezählt werden. In Bild 4 ist die Zeile mit dem Kommentar **'09** die letzte Zeile. Wenn wir hinter **'09** noch einmal die Eingabetaste betätigen und somit einen Zeilenumbruch hinzufügen, liefert **CountOfLines** folglich den Wert **10**.

Beispielmodul für die folgenden Beispiele

Damit Sie Beispielmaterial haben, anhand dessen Sie die folgenden Beispiele nachvollziehen können, haben wir das Modul **mdlBeispielcode** aus Bild 5 zum Projekt hinzugefügt. Mit der **CountOfLines**-Eigenschaft erhalten Sie für dieses Modul den Wert **28**.

Anzahl der Deklarationszeilen

Die Eigenschaft **CountOfDeclarationLines** liefert die Anzahl der Zeilen bis zur letzten Deklarationsanweisung im Modul. Da Sie keine Deklarationszeilen hinter der ersten Routine mehr verwenden dürfen, können Sie beide Bereiche somit sehr gut voneinander unterscheiden.

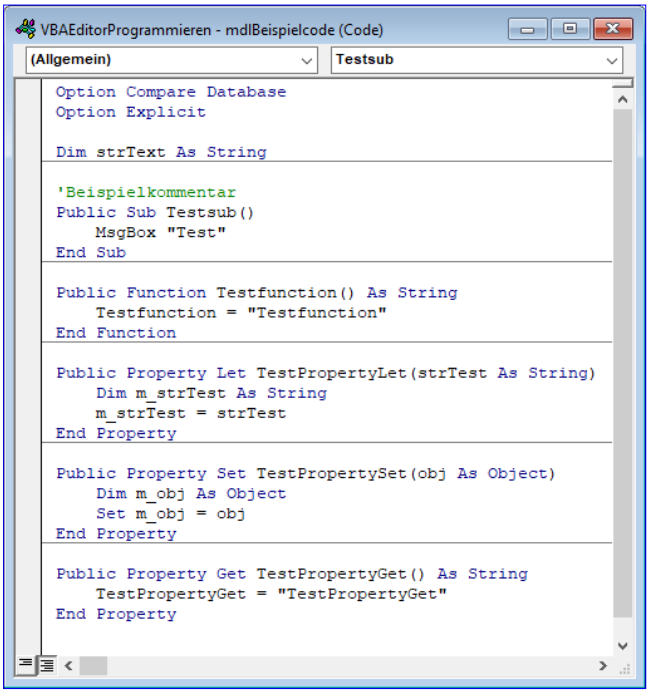


Bild 5: Beispielmodul

Im Gegensatz zu **CountOfLines** kümmert sich **CountOfDeclarationLines** situationsbedingt nicht um nachfolgende Leerzeilen:

- Wenn nach der letzten Deklarationszeile mindestens eine **Sub**-, **Function**- oder **Property**-Prozedur folgt, dann gibt **CountOfDeclarationLines** die Anzahl der Zeilen bis zur letzten Deklarationszeile aus.

- Wenn nach der letzten Deklarationszeile keine **Sub**-, **Function**- oder **Property**-Prozedur folgt, wenn das

Modul also nur einen Deklarationsteil enthält, dann liefert **CountOfDeclarationsLines** die Anzahl der Zeilen bis zur letzten Zeile des Moduls.

Im Fall des Moduls aus dem Beispiel erhalten wir also folgendes Ergebnis:

```
? VBE.ActiveCodePane.CodeModule.CountOfDeclarationLines
 4
```

Anzahl der Zeilen einer Prozedur

Wenn Sie die Anzahl der Zeilen einer speziellen Prozedur ermitteln wollen, nutzen Sie die Eigenschaft **ProcCountLines**. Diese Eigenschaft erwartet zwei Parameter:

- **ProcName**: Name der zu untersuchenden Prozedur
- **ProcKind**: Art der zu untersuchenden Prozedur. Mögliche Werte: **vbext_pk_Get** (**Property Get**-Prozedur), **vbext_pk_Let** (**Property Let**-Prozedur), **vbext_pk_Proc** (**Sub**- oder **Function**-Prozedur) oder **vbext_pk_Set** (**Property Set**-Prozedur)

Um diese Eigenschaft nutzen zu können, müssen Sie sowohl den Namen als auch den Typ der Prozedur kennen. Ein Beispielaufruf aus dem Direktbereich heraus lautet:

```
? VBE.ActiveCodePane.CodeModule.ProcCountLines("Testsub",
vbext_pk_Proc)
 5
```

Warum erscheint hier der Wert **5**? Weil sowohl die Kommentarzeile als auch die leeren Zeilen unmittelbar vor der untersuchten Prozedur mitgezählt werden – also alle seit dem allgemeinen Deklarationsteil oder der vorherigen Prozedur inklusive Kommentarzeilen.

Alle Zeilen eines Moduls ausgeben

Wenn Sie eine oder mehrere Zeilen eines Moduls ausgeben wollen, referenzieren Sie als erstes das **CodeModule**-Objekt. Dann können Sie mit der **Lines**-Methode gezielt den Inhalt einer oder mehrerer Zeilen gleichzeitig abfragen.

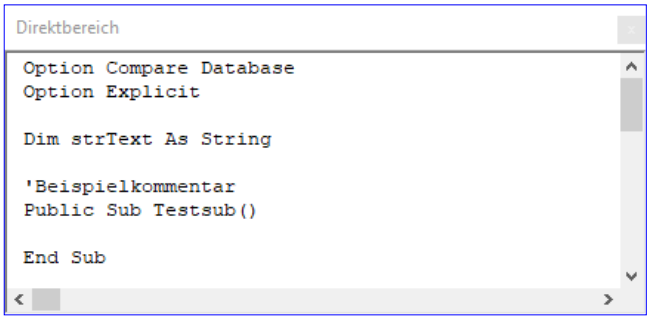


Bild 6: Ausgabe aller Zeilen

Im ersten Beispiel geben wir für das erste Argument der **Lines**-Methode den Wert **1** für die erste Zeile und den Wert der Eigenschaft **CountOfLines** des **CodeModule**-Objekts für den zweiten Parameter an. **Lines** sollte hier also alle Zeilen in einem Rutsch liefern:

```
Public Sub AlleZeilenGleichzeitigAusgeben()
    Dim objCodeModule As CodeModule
    Set objCodeModule = VBE.ActiveVBProject.
                VBComponents("mdlBeispielcode").CodeModule
    Debug.Print objCodeModule.Lines(1,
                                objCodeModule.CountOfLines)
End Sub
```

Das Ergebnis finden Sie in Bild 6.

Nun wollen wir die Prozedur inklusive Zeilennummern ausgeben. Dazu referenziert die folgende Prozedur das Modul **mdlBeispielcode** und durchläuft in einer **For Next**-Schleife die Zahlen von **1** bis zu der mit der Eigenschaft **CountOfLines** ermittelten Anzahl der Zeilen.

Dabei gibt sie jeweils den Inhalt der aktuellen Zeile im Direktbereich aus. Hier fügen wir vorn noch die Nummer der aktuellen Zeile im Format **00** an:

```
Public Sub AufEineZeileZugreifen()
    Dim objCodeModule As CodeModule
```

Auf VBA-Code zugreifen mit der CodePane-Klasse

In den vorherigen Beiträgen dieser Beitragsreihe haben wir uns bereits angesehen, wie Sie auf die Module im VBA-Editor zugreifen, neue Module erstellen und den enthaltenen Code bearbeiten. Es fehlt allerdings noch eine wichtige Schnittstelle zwischen Benutzer und der automatisierten Bearbeitung von VBA-Code: Die Klasse CodePane, die unter anderem die Möglichkeit bietet, vom Benutzer gesetzte Markierungen im Code auszulesen und solche zu setzen. Letzteres können Sie beispielsweise nutzen, um per VBA gesuchte Stellen im Code zu markieren, damit der Benutzer diese erkennen kann. Dieser Beitrag stellt die CodePane-Klasse und ihre Möglichkeiten vor.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Vorbereitende Beiträge

Wenn Sie erfahren wollen, wie Sie überhaupt bis zu der hier beschriebenen Klasse gelangen, helfen die folgenden Beiträge weiter:

- **VBA-Projekt per VBA referenzieren** (**www.access-im-unternehmen.de/1337**)
- **Zugriff auf den VBA-Editor mit der VBE-Klasse** (**www.access-im-unternehmen.de/1350**)
- **Zugriff auf VBA-Projekte per VBProject** (**www.access-im-unternehmen.de/1351**)
- **Module und Co. im Griff mit VBComponent** (**www.access-im-unternehmen.de/1352**)
- **VBA-Code manipulieren mit der CodeModule-Klasse** (**www.access-im-unternehmen.de/1353**)

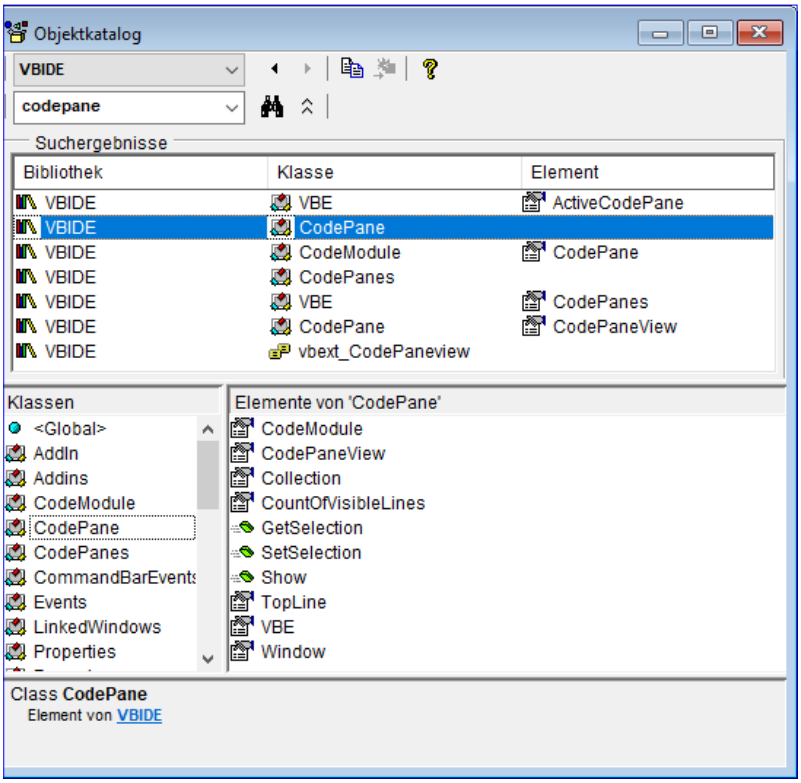


Bild 1: Die **CodePane**-Klasse im Objektkatalog

Elemente der CodePane-Klasse

Die **CodePane**-Klasse und ihre Eigenschaften, Methoden und Auflistungen können Sie im Objektkatalog (zu öffnen mit der Taste **F2**) im Überblick ansehen, wenn Sie dort nach **CodePane** suchen (siehe Bild 1). Die **CodePane**-Klasse hat folgende Eigenschaften, Methoden und Auflistungen:

- **CodeModule**: Verweis auf das **CodeModule**-Objekt, das im **CodePane**-Objekt enthalten ist
- **CodePaneView**: Gibt die aktuelle Darstellung im CodePane wieder – entweder vollständiges Modul (**1**) oder nur einzelne Prozeduren (**0**).
- **Collection**: Erlaubt den Zugriff auf alle geöffneten **CodePane**-Objekte.
- **CountOfVisibleLines**: Liefert die Anzahl der sichtbaren Zeilen im Fenster.
- **GetSelection**: Liefert den aktuell markierten Bereich mit den vier Rückgabeparametern.
- **SetSelection**: Setzt eine Markierung anhand von vier Parametern.
- **Show**: Versieht das **CodePane**-Objekt, für das diese Methode aufgerufen wurde, mit dem Fokus.
- **TopLine**: Gibt die Zeilennummer der oben im Fenster angezeigten Zeile aus und erlaubt auch das Einstellen dieser Zeile.
- **VBE**: Verweis auf die **VBE**-Klasse
- **Window**: Verweis auf das übergeordnete **Window**-Objekt

Auf das CodePane-Objekt zugreifen

Um auf ein **CodePane**-Objekt zuzugreifen, gibt es mehrere Möglichkeiten. Der Zugriff erfolgt entweder von der Auflistung **CodePanes** der **VBE**-Klasse, vom untergeordneten **CodeModule**-Element oder über die Eigenschaft **ActiveCodePane** der **VBE**-Klasse.

Wenn Sie den Namen eines Moduls kennen und auf sein **CodePane**-Objekt zugreifen wollen, nutzen Sie den Weg über **CodeModule**. Dazu referenzieren Sie zuerst das **CodeModule**-Objekt, das sie über die gleichnamige Eigenschaft des entsprechenden Elements der **VBComponents**-Auflistung des aktuellen VB-Projekts erhalten. Dieses bietet über die **CodePane**-Eigenschaft Zugriff auf das **CodePane**-Objekt.

Damit wir sehen, ob wir das richtige **CodePane**-Objekt referenzieren, zeigen wir sein Fenster mit der **Show**-Methode an:

```
Public Sub CodePanePerCodeModule()
    Dim objCodeModule As CodeModule
    Dim objCodePane As CodePane
    Set objCodeModule = VBE.ActiveVBProject.
                VBComponents("mdlBeispielcode").CodeModule
    Set objCodePane = objCodeModule.CodePane
    objCodePane.Show
End Sub
```

Damit können Sie auf alle **CodePane**-Elemente zugreifen, auch die von aktuell noch geschlossenen Modulen. Nur die aktuell in **Window**-Elementen angezeigten **CodePane**-Elemente können Sie über die Auflistung **CodePanes** der **VBE**-Klasse ansprechen.

Die folgende Prozedur durchläuft in einer **For Each**-Schleife alle Elemente der **CodePanes**-Auflistung und referenziert das jeweils aktuelle Element mit der Variablen **objCodePane**:

```
Public Sub CodePanePerWindow()
    Dim objCodePane As CodePane
    For Each objCodePane In VBE.CodePanes
        Debug.Print objCodePane.CodeModule.Parent.Name
    Next objCodePane
End Sub
```

Da das **CodePane**-Objekt selbst keine **Name**-Eigenschaft enthält, greifen wir über **objCodePane.CodeModule.Parent** auf das **VBComponent**-Objekt dieses **CodePane**-Objekts zu und geben seinen Namen aus. Das liefert nur die Namen der Module, die aktuell geöffnet sind.

Setup für Access-Applikationen, Restarbeiten

Autor: Christoph Jüngling, https://www.juengling-edv.de

In diesem Teil widmen wir uns einigen Restarbeiten für das Erstellen eines Setups für Access-Applikationen. Diese Arbeiten sind zwar keineswegs unbedingt notwendig, runden aber unser Setup ab und sorgen daher beim Anwender oder Administrator für ein »gutes Gefühl«. Wir wollen zunächst sicherstellen, dass unsere Applikation nicht läuft, wenn wir sie updaten wollen. Dann unterscheiden wir bei der Installation zwischen Beta- und finaler Version. Und letztlich wollen wir unser Setup dann noch digital signieren.

Läuft die Applikation?

Wie können wir sicherstellen, dass die Access-Applikation nicht läuft, wenn wir sie updaten wollen?

Prinzip eines "Mutex"

Sicher haben Sie schon bei Setups bemerkt, dass eine Meldung wie diese kam:

```
Das Setup hat festgestellt, dass die Applikation Xyz noch
läuft. Bitte beenden Sie diese und starten Sie dann das
Setup erneut.
```

Dem Anwender ist das natürlich egal, aber der Entwickler fragt sich unweigerlich: »Woher weiß das Setup das eigentlich?« Die Frage ist berechtigt, denn einfach mal »nachschauen« ist halt für eine Software nicht ganz so einfach wie für uns.

Das Geheimnis heißt »Mutex«. Das ist die Kurzform von »mutual exclusion«, auf Deutsch »gegenseitiger Ausschluss«, eine beeindruckend gute Bezeichnung für diesen Mechanismus.

Denn die Setupdurchführung und das laufende Programm schließen sich gegenseitig aus. Während die ACCDB aktiv ist, sollte man sie nicht per Setup austauschen.

Um dies zu nutzen, benötigen wir also sowohl in der Applikation als auch im Setup jeweils einen Mechanismus, der den Mutex setzen, löschen und überprüfen kann. Unsere Access-Applikation erzeugt den Mutex beim Start und löscht ihn kurz vor dem Ende.

Das Setup wiederum überprüft nur, ob es ihn gibt. Falls ja, kommt eine Meldung wie oben gezeigt, falls nein, läuft das Setup einfach weiter.

Das lässt sich in InnoSetup sogar soweit automatisieren, dass ein von der Accdb angestoßenes Update automatisch abläuft.

Mutex-Klasse (VBA)

Mit Hilfe eines kurzen Codesegments schaffen wir es, den Mutex beim Starten unserer Access-Applikation zu erzeugen. Das übernimmt in meinen Projekten immer eine Klasse, die ich eigens dafür geschrieben habe.

Es müsste natürlich keine Klasse sein, eine Sub-Prozedur täte es auch, aber mit der Klasse haben wir den Vorteil, dass wir mit einer Einheit beide Aktionen erledigen können: Erzeugen und Löschen des Mutex wird innerhalb der Klasse durchgeführt – und zwar über den Konstruktor **Initialize** und den Destruktor **Terminate** der Klasse.

Zur Integration in die Applikation müssen neben der Klasse selbst nur noch eine Deklaration und zwei Zeilen Code eingefügt werden.

Schauen wir uns zunächst die Mutex-Klasse an. Besonders aufwändig ist sie nicht (siehe Listing 1).

Die Erzeugung passiert in **Class_Initialize**, die Zerstörung in **Class_Terminate**. Der einzige Bezug in unsere

```
''
' Manage an Application Mutex for the current application
' @remarks  Mutex name = APP_MUTEX
' @author   Christoph Juengling <christoph@juengling-edv.de>
' @link   https://gitlab.com/juengling/vb-and-vba-code-library
Option Explicit
#If VBA7 Then
Private Declare PtrSafe Function CreateMutex Lib "kernel32" Alias "CreateMutexA" (lpMutexAttributes As Any, _
     ByVal bInitialOwner As Long, ByVal lpName As String) As LongPtr
Private Declare PtrSafe Function CloseHandle Lib "kernel32" (ByVal hObject As LongPtr) As Long
Private Declare PtrSafe Function ReleaseMutex Lib "kernel32" (ByVal hMutex As LongPtr) As Long
#Else
Private Declare Function CreateMutex Lib "kernel32" Alias "CreateMutexA" (lpMutexAttributes As Any, _
    ByVal bInitialOwner As Long, ByVal lpName As String) As Long
Private Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long
Private Declare Function ReleaseMutex Lib "kernel32" (ByVal hMutex As Long) As Long
#End If
Private m_lMutexHandle As Long
Private Sub Class_Initialize()
    m_lMutexHandle = 0
    CreateMyMutex
End Sub

Private Sub Class_Terminate()
    ReleaseMyMutex
End Sub

' CreateMutex the Mutex
Public Sub CreateMyMutex()
    m_lMutexHandle = CreateMutex(ByVal CLng(0), CLng(1), APP_MUTEX)
End Sub

''
' ReleaseMutex the Mutex and close handle
'
Public Sub ReleaseMyMutex()
    If m_lMutexHandle > 0 Then
        ReleaseMutex m_lMutexHandle
        CloseHandle m_lMutexHandle
        m_lMutexHandle = 0
    End If
End Sub
```

Listing 1: Klasse zum Nutzen eines Mutex

Applikation ist hier die Nutzung der globalen Konstanten **APP_MUTEX**. Diese deklariert den Namen des Mutex:

```
Public Const APP_MUTEX = "Mein Name ist Hase"
```

Die Bezeichnung sollte sinnvollerweise dem Namen der Applikation entsprechen, so dass man natürlich auch **APP_NAME** verwenden kann.

Wichtig ist die Einzigartigkeit im Hinblick auf andere Applikationen! Eine solche Deklaration muss natürlich namensgleich später auch im InnoSetup-Skript enthalten sein, damit Setup und Applikation über dieselbe Bezeichnung verfügen.

Nun müssen wir nur noch dafür sorgen, dass

- eine Modul-Variable für die Instanz der Klasse existiert,
- diese beim Start instanziiert wird und
- vor dem Beenden der Applikation zerstört wird.

Das ist ebenfalls trivial. In einem Modul, in dem vielleicht noch weitere globale Deklarationen stehen, fügen wir diese Zeile ein:

```
Public mutex As clsMutex
```

In dem Code für die Initialisierung:

```
Set mutex = New clsMutex
```

Und für das Beenden:

```
Set mutex = Nothing
```

Das war doch einfach, oder? Streng genommen ist die letzte Aktion nicht nötig, da der Mutex mit dem Beenden der Applikation automatisch gelöscht wird.

Den Mutex in InnoSetup eintragen

In InnoSetup ist die Sache sogar noch einfacher, denn der ganze Mechanismus zum Überprüfen und Reagieren ist dort bereits enthalten, Code müssen wir dafür nicht schreiben.

Wir müssen nur dafür sorgen, dass InnoSetup den Namen des Mutex erfährt. Wie schon erwähnt, muss diese Deklaration natürlich identisch mit der Konstanten aus unserer Applikation sein, und das betrifft auch die Groß-/Kleinschreibung!

Im Deklarationsbereich zu Beginn des Skriptes schreiben wir also:

```
#define MyAppMutex "Mein Name ist Hase"
```

Und in der Gruppe **[Setup]**:

```
AppMutex={#MyAppMutex}
```

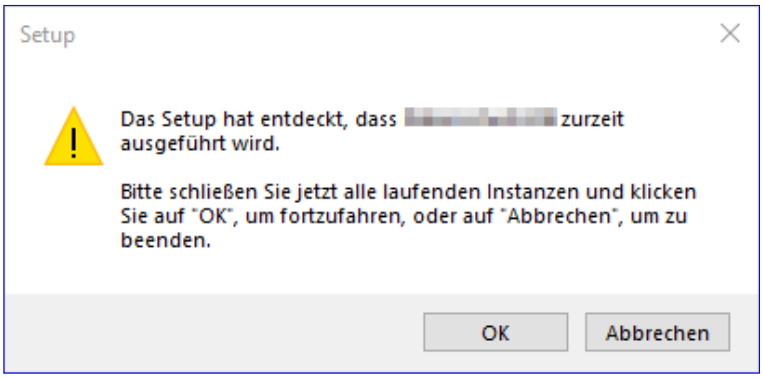


Bild 1: Mutex-Meldung

Mehr ist nicht nötig. Und wie funktioniert das nun?

Zum einen genau wie oben beschrieben. Wenn das Setup bei der Ausführung entdeckt, dass der Mutex bereits existiert, zeigt es die Meldung aus Bild 1 an.

Nun kann der Anwender entsprechend darauf reagieren. Besonders elegant arbeitet InnoSe-

tup jedoch, wenn es für die Installation mit dem Kommandozeilen-Argument **silent** aufgerufen wurde.

Dann nämlich verzichtet es auf alle Rückfragen mit Benutzerinteraktion. Für den Mutex-Fall bedeutet das, dass das Setup solange abwartet, bis der Mutex verschwunden ist. Dann wird das Setup mit Standardeinstellungen fortgeführt.

Dieser Mechanismus hat den besonderen Reiz, dass die Applikation selbst per **Shell**-Execute die Ausführung des Setups anstarten und sich dann in aller Ruhe beenden kann. Erst wenn der Mutex gelöscht ist, läuft das Setup durch.

Beta oder Final?

Nicht immer ist ein Programm sofort fertig, auch wenn der Entwickler noch so überzeugt von seiner Arbeit ist. Daher kann es sinnvoll sein, ausgewählte User in einen Beta-Test einzubeziehen.

Dabei sollte jedoch immer auf eine strenge Trennung nicht nur der Programminstallation, sondern auch bezüglich der Daten geachtet werden. Eine Möglichkeit dabei ist eine Kombination aus dem Setup-Skript und dem Code, der für die Tabelleneinbindung sorgt. Worauf müssen wir achten, und wie machen wir uns die Arbeit möglichst einfach?

Beginnen wir mit der Frage, bezüglich welcher Einstellungen sich Beta- und Final-Version unterscheiden:

- Installationspfad
- Beta-Hinweis oder Lizenzvereinbarung
- Pfad zum Backend

Installationspfad

Nehmen wir (wie schon im ersten Teil dargelegt) an, dass wir unser »normales« Programm unter **C:\Users\USERNAME\AppData\Local\Programs** installieren wollen. Dort wird sicherlich für jedes Programm ein eigenes Unterverzeichnis verwendet werden.

Es bietet sich also an, dies für unser eigenes Programm ebenfalls zu tun, und zwar getrennt für Beta- und Final-Version.

Dazu müssen wir also die Setup-Einstellung **DefaultDirName** angemessen verändern. Bisher steht dort **{userpf}\{#MyAppName}**. Es spricht also nichts dagegen, zum Beispiel **{userpf}\{#MyAppName}**-Beta zu verwenden.

Doch mir widerstrebt es, jedesmal mitten im Skript eine Änderung vorzunehmen. Da kann man sich leicht vertun, und es wird auch nicht die einzige Änderung bleiben. Eine Lösung für diesen Fall sind wieder einmal die Konstanten, die wir ja bereits kennengelernt haben. Ich füge also zunächst eine weitere hinzu:

```
#define MyAppStatus "Beta"
```

oder

```
#define MyAppStatus "Final"
```

Damit haben wir wieder eine Einstellung, die wir nur am Anfang des Skriptes verändern müssen, wodurch hoffentlich alle anderen Settings entsprechend angepasst werden.

Damit das funktioniert, nutzen wir eine weitere Möglichkeit von InnoSetup, die uns als Entwickler natürlich vertraut ist: Die **If-Then-Else**-Anweisung. Sie funktioniert im Prinzip wie von VBA her bekannt, nur sieht die Syntax ein wenig anders aus.

Im folgenden Beispiel erweitere ich bei einer Betaversion den Standard-Installationspfad und die Startmenü-Gruppe zum Beispiel durch den Zusatz **-Beta**:

Export von Daten in das DATEV-Format

Um Daten Ihrer eigenen Software so zu exportieren, dass Ihr Steuerberater diese in das DATEV-System einlesen kann, benötigen Sie gar nicht mal so viel Know-how. Die wesentlichen Informationen finden wir auf der Webseite von DATEV. Dieser Beitrag zeigt, wie Sie Daten im DATEV-Format exportieren und was dabei zu beachten ist. Die so entstandene Datei kann der Steuerberater dann per Importfunktion in das DATEV Rechnungswesen einlesen.

Die grundlegenden Informationen finden Sie in der öffentlich verfügbaren Webseite unter folgender Adresse:

```
https://developer.datev.de/portal/de/dtvf
```

Dieser Beitrag beschränkt sich auf die Zusammenstellung einer Datei mit den **Header**-Daten und Daten im Format **Buchungsstapel**.

Grundlegender Aufbau einer DATEV-Datei

Die DATEV-Datei enthält immer eine Zeile mit Daten im Format **Header**. Danach folgt eine Zeile mit den Spaltenüberschriften für die Buchungsdaten und schließlich eine oder mehrere Zeilen mit den eigentlichen Buchungsdaten.

Verschiedene Formate

Das DATEV-Format besteht aus verschiedenen Formatbeschreibungen. Verschiedene Formate deshalb, weil es unterschiedliche Informationen gibt, die Sie damit beschreiben können. Wir wollen uns auf die wesentlichen Elemente beschränken – die übrigen können Sie mit dem Know-how aus diesem Beitrag und der Dokumentation dann leicht selbst realisieren.

Es gibt die folgenden Formatsätze:

- **Header**: Erste Zeile der CSV-Datei mit den Informationen zur Verarbeitung der Datei
- **Buchungsstapel**: Enthält die eigentlichen Buchungen, in der Regel für einen bestimmten Zeitraum je Datei, zum Beispiel monatlich.
- Weitere Formatsätze, die dieser Beitrag nicht behandelt: **Wiederkehrende Buchungen**, **Debitoren/Kreditoren**, **Sachkontenbeschriftungen**, **Zahlungsbedingungen** und **Diverse Adressen**.

Zusammenstellen der Header-Zeile

Die Beschreibung des Satzaufbaus für die Header-Zeile finden Sie hier:

```
https://developer.datev.de/portal/de/dtvf/formate/header
```

Die Tabelle auf dieser Seite enthält die Überschriften und die Beschreibung der möglichen Werte als regulärer Ausdruck.

Eine Header-Zeile sieht beispielsweise wie folgt aus:

```
"EXTF";700;21;"Buchungsstapel";12;20211007000000000;4;"Buchungen_Konto1";1;"EUR"
```

Dabei entsprechen die einzelnen Elemente diesen Feldern:

- **Kennzeichen**: **EXTF** oder **DTVF** (hier **EXTF** für Export aus einer 3rd-Party-Anwendung)
- **Versionsnummer**: aktuell **700** (dient der Sicherstellung von Abwärtskompatibilität)
- **Formatkategorie**: Gibt an, in welchem Format die folgenden Daten sind (im Falle des Buchungsstapels, mit dem wir uns hier beschäftigen, nutzen wir den Wert **21**).

- **Formatname**: Name des Formats, hier **Buchungsstapel**

- **Formatversion**: Auch hier kommt ein Zahlenwert für Buchungsstapel zum Einsatz, in diesem Fall **12**.

- **Erzeugt am**: Datum, an dem die Datei erzeugt wurde, im Format **YYYYMMDDHHMMSSFFF**.

Es gibt noch einige weitere Felder, die wir weiter unten berücksichtigen.

Tabelle für Headerdaten

Wie speichern wir die Headerdaten am einfachsten? Wenn Sie aus ihrer selbstprogrammierten Buchhaltungsdatenbank die Buchungsdaten für den Import in die DATEV-Software Ihres Steuerberaters exportieren wollen, werden Sie das regelmäßig erledigen – beispielsweise monatlich oder einmal je Quartal.

Die Felder des Headers enthalten Daten, die wir über eine Tabelle namens **tblHeader** erfassen (Entwurf siehe Bild 1). Diese enthält nicht genau die Felder, die in der Header-Datei gespeichert werden sollen. Die Felder **ID** und **Referenzbezeichnung** sind das Primärschüsselfeld sowie ein Feld für eine interne Bezeichnung, zum Beispiel **Export 12/2021**. Die übrigen Felder nehmen meist direkt die benötigten Daten auf, einige jedoch sind Fremdschlüsselfelder, welche zur Auswahl von Daten aus Lookup-Tabellen dienen.

tblHeader

Feldname	Felddatentyp	Beschreibung (optional)
ID	AutoWert	ID des Headersatzes
Dateiname	Kurzer Text	Name der zu erstellenden Datei ohne Präxix und Suffix)
Kennzeichen	Kurzer Text	Kennzeichen, entweder EXTF (Export aus 3rd Party App) oder DTVF (Export aus Datev App)
Versionsnummer	Kurzer Text	Versionsnummer des Headers
FormatkategorieID	Zahl	siehe Tabelle tblFormatkategorien
Formatversion	Zahl	Formatversion
ErzeugtAm	Datum/Uhrzeit	Erzeugungsdatum
7_Reserviert	Kurzer Text	Leerfeld
8_Reserviert	Kurzer Text	Leerfeld
9_Reserviert	Kurzer Text	Leerfeld
10_Reserviert	Kurzer Text	Leerfeld
Beraternummer	Kurzer Text	Beraternummer
Mandantennummer	Kurzer Text	Mandantennummer
Wirtschaftsjahresbeginn	Datum/Uhrzeit	Beginn des Wirtschaftsjahres
Sachkontenlaenge	Zahl	Nummernlänge der Sachkonten
DatumVon	Datum/Uhrzeit	Startdatum
DatumBis	Datum/Uhrzeit	Enddatum
Bezeichnung	Kurzer Text	Bezeichnung des Stapels
Diktatkuerzel	Kurzer Text	Kürzel des Bearbeiters
Buchungstyp	Zahl	siehe Tabelle tblBuchungstypen
Rechnungslegungszweck	Zahl	siehe Tabelle tblRechnungslegungszwecke
Festschreibung	Zahl	siehe Tabelle tblFestschreibungen
Waehrungskennzeichen	Kurzer Text	ISO-Code der verwendeten Währung
23_Reserviert	Kurzer Text	Leerfeld
Derivatskennzeichen	Kurzer Text	Leerfeld
25_Reserviert	Kurzer Text	Leerfeld
26_Reserviert	Kurzer Text	Leerfeld
Sachkontenrahmen	Kurzer Text	Sachkontenrahmen, der verwendet wurde
IDDerBranchenLoesung	Kurzer Text	Falls eine spezielle Branchenlösung verwendet wurde
29_Reserviert	Kurzer Text	Leerfeld
30_Reserviert	Kurzer Text	Leerfeld
Anwendungsinformationen	Kurzer Text	Verarbeitungskennzeichen der abgebenden Anwendung

Bild 1: Entwurf der Tabelle **tblHeader**

Diese Lookup-Tabellen stellen wir in Bild 2 übersichtlich dar.

Die Beziehungen zwischen der Tabelle **tblHeader** und den übrigen Tabellen haben wir jeweils als Nachschlagefeld definiert.

Formular zur Eingabe der Headerdaten

Damit der Benutzer die Headerdaten einfach eingeben kann, haben wir dazu ein eigenes Formular bereitgestellt. Dieses heißt **frmHeader** und verwendet die Tabelle **tblHeader** als Datensatzquelle. Wir haben alle Felder der Tabelle aus der Feldliste in den Formularentwurf gezogen, die in der Dokumentation nicht als Leerfeld gekennzeichnet sind (siehe Bild 3).

Hier sehen Sie bereits, dass die Felder, für die wir Lookup-Tabellen angelegt haben, auch im Formularentwurf als Nachschlagefelder angelegt werden.

Nach dem Wechsel in die Formularansicht können Sie die Daten direkt in das Formular eingeben beziehungsweise mit den Nachschlagefeldern auswählen (siehe Bild 4).

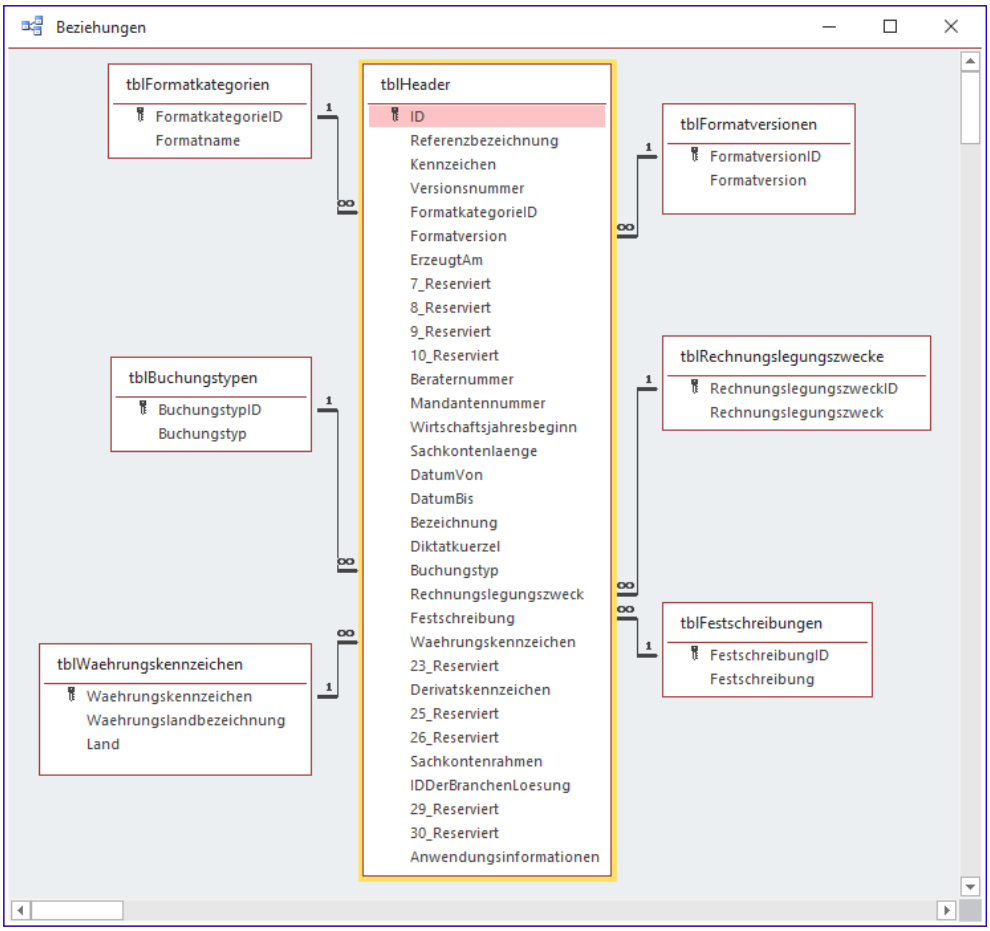


Bild 2: Übersicht des Datenmodells

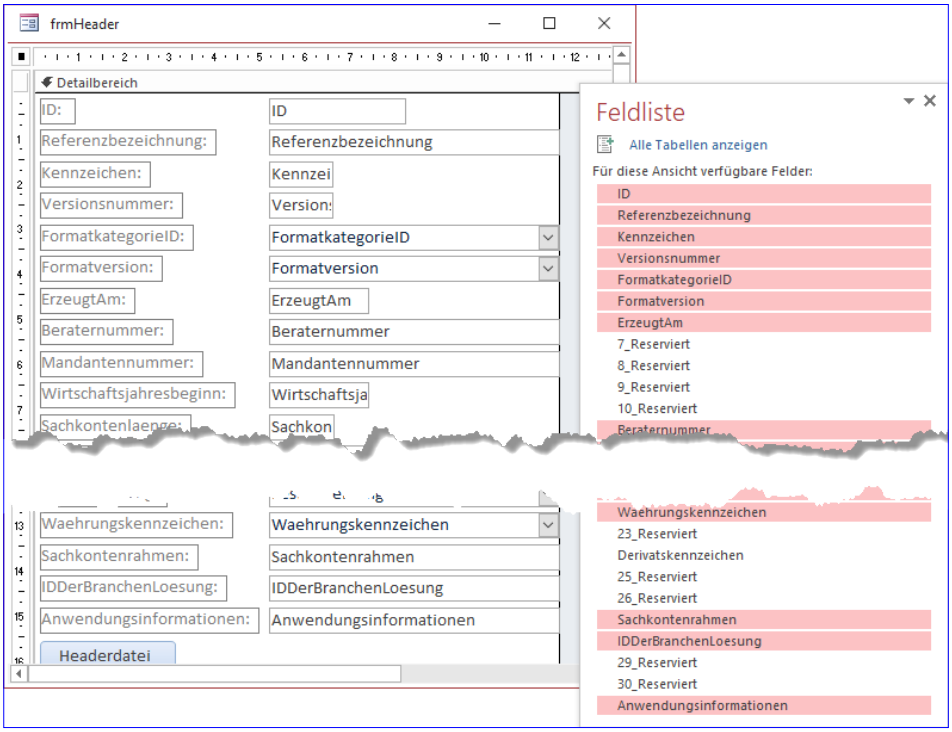


Bild 3: Entwurf des Formulars zum Eingeben der Headerdaten

Nachdem wir diese Eingabemöglichkeit geschaffen haben, wollen wir daraus noch die Headerdatei exportieren. Dazu sehen wir eine Schaltfläche namens **cmdHeaderdateiErzeugen** vor, die wir unten im Formular platzieren.

Diese Schaltfläche ruft die Prozedur aus Listing 1 auf.

Die Prozedur deklariert zwei Variablen, von denen eine die zu speichernde Headerzeile aufnimmt und die andere den Dateinamen.

Den Dateinamen setzt die Prozedur aus dem Pfad der aktuellen Datenbankdatei, dem Präfix **DTVF_**, dem Inhalt des Feldes **Dateiname** und der Dateiendung **.csv** zusammen.

Bevor wir richtig loslegen, löscht die Prozedur eine eventuell bereits vorhandene Datei mit dem Namen aus **strDateiname**. Dabei deaktivieren wir die eingebaute Fehlerbehandlung, da die **Kill**-Anweisung beim Fehlen der angegebenen Datei einen Fehler auslöst.

Die folgenden Zeilen setzen den Inhalt der Headerdatei zusammen.

Dabei sind folgende Besonderheiten zu beachten (in allen anderen Fällen landet einfach der eingegebene Wert zwischen zwei Semikola):

- Das Kennzeichen wird in Anführungszeichen eingefasst.
- Den Wert des Feldes **Formatname** lesen wir aus der Lookup-Tabelle **tblFormatversionen** ein.
- Den Wert für das Feld **Erzeugt am** formatieren wir im Format **YYYYMMDDHHNNSS000**.
- Den Wert für das Feld **Wirtschaftsjahresbeginn** formatieren wir mit dem Format **YYYYMMDD**.
- Die Werte für die Felder **Datum von** und **Datum bis** formatieren wir ebenfalls mit **YYYYMMDD**.

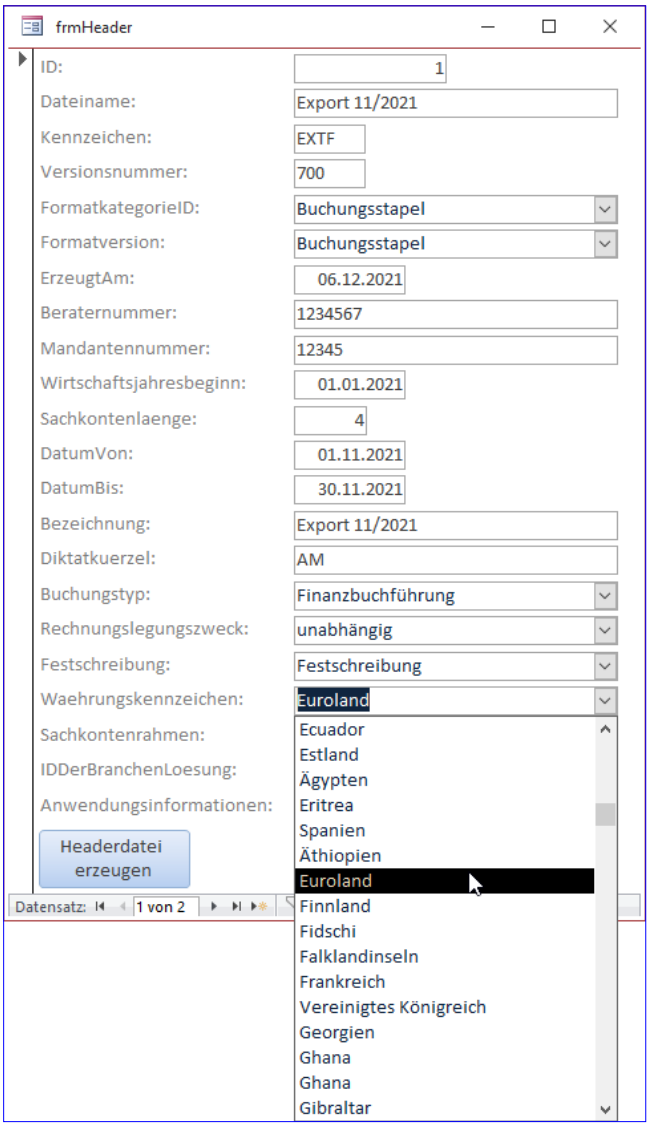


Bild 4: Eingabe der Headerdaten

Anschließend öffnen wir mit **Open** eine Datei mit dem Namen aus **strDateiname** und schreiben mit der **Print**-Anweisung den Inhalt der Variablen **strHeader** in die Datei, bevor wir diese mit der **Close**-Anweisung wieder schließen.

Das Ergebnis sieht für unser Beispiel wie folgt aus:

```
"EXTF";700;21;Buchungsstapel;12;20211206000000000;;;
;;1234567;12345;20210101;4;20211101;20211130;Export
11/2021;AM;1;0;1;EUR;;;;;03;;;;
```

Damit können wir nun einen Schritt weitergehen – und das Ergebnis mit einem speziell für diesen Zweck vorgesehenen Tool prüfen. Wie das gelingt, erfahren Sie im nächsten Schritt.

```
Private Sub cmdHeaderdateiErzeugen_Click()
    Dim strHeader As String
    Dim strDateiname As String
    strDateiname = CurrentProject.Path & "\DTVF_" & Me!Dateiname & ".csv"
    On Error Resume Next
    Kill strDateiname
    On Error GoTo 0
    strHeader = strHeader & """ & Me!Kennzeichen & """;"
    strHeader = strHeader & Me!Versionsnummer & ";"
    strHeader = strHeader & Me!FormatkategorieID & ";"
    strHeader = strHeader & DLookup("Formatversion", "tblFormatversionen", "FormatversionID = " & Me!Formatversion) & ";"
    strHeader = strHeader & Me!Formatversion & ";"
    strHeader = strHeader & Format(Me!ErzeugtAm, "YYYYMMDDHHNNSS000") & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & Me!Beraternummer & ";"
    strHeader = strHeader & Me!Mandantennummer & ";"
    strHeader = strHeader & Format(Me!Wirtschaftsjahresbeginn, "YYYYMMDD") & ";"
    strHeader = strHeader & Me!Sachkontenlaenge & ";"
    strHeader = strHeader & Format(Me!DatumVon, "YYYYMMDD") & ";"
    strHeader = strHeader & Format(Me!DatumBis, "YYYYMMDD") & ";"
    strHeader = strHeader & Me!Bezeichnung & ";"
    strHeader = strHeader & Me!Diktatkuerzel & ";"
    strHeader = strHeader & Me!Buchungstyp & ";"
    strHeader = strHeader & Me!Rechnungslegungszweck & ";"
    strHeader = strHeader & Me!Festschreibung & ";"
    strHeader = strHeader & Me!Waehrungskennzeichen & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & Me!Sachkontenrahmen & ";"
    strHeader = strHeader & Me!IDDerBranchenLoesung & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & Me!Anwendungsinformationen
    Open strDateiname For Append As #1
    Print #1, strHeader
    Close #1
End Sub
```

Listing 1: Prozedur zum Erzeugen der Headerdatei

Datei im DATEV-Format prüfen

Unter dem folgenden Link finden Sie ein Tool zum Prüfen der Daten im DATEV-Format:

`https://developer.datev.de/portal/de/dtvf/tools`

Mit diesem können Sie die erstellte Datei öffnen und prüfen. Das Tool zeigt dann an, welche Elemente fehlen oder das falsche Format haben. Es ist daher sehr hilfreich, den Export initial auf Basis von Daten aus der Datenbank zu programmieren und gleich zu testen.

Um die soeben erstellte Headerdatei zu prüfen, wählen Sie im Prüfprogramm den Menübefehl **Datei|Öffnen** aus.

Im nun erscheinenden Dialog **Datev Format Datei öffnen** wählen Sie die soeben erstellte Datei aus und bestätigen die Auswahl mit der Schaltfläche **Öffnen**.

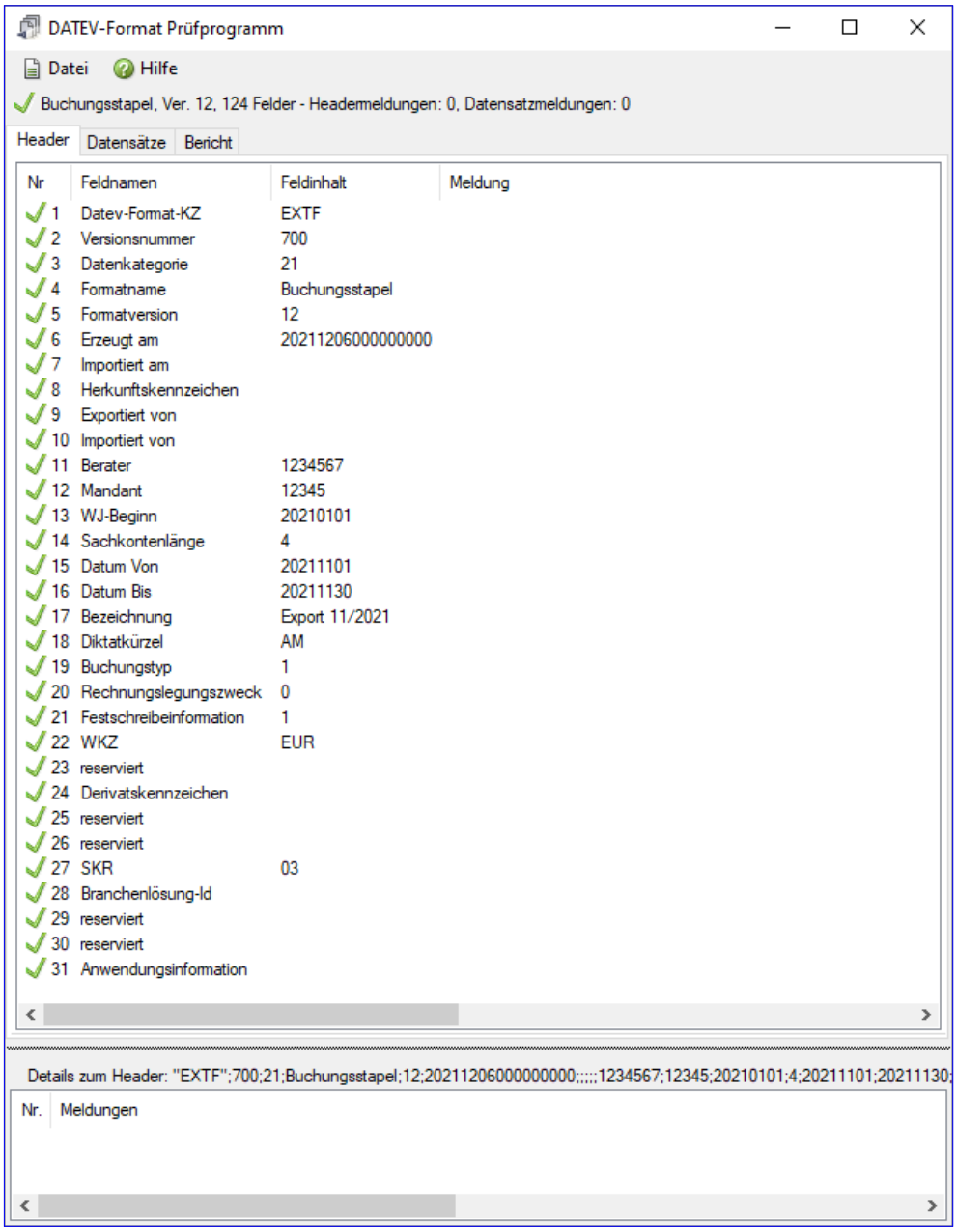


Bild 5: Prüfung des Headers

Bild 5 zeigt das Ergebnis für die Eingaben aus dem Beispiel von oben. Hier sehen Sie, dass alle Eingaben korrekt sind. Wäre dies nicht der Fall, würden Sie in der Spalte **Meldung** hilfreiche Informationen finden, mit denen Sie den Export schnell anpassen und funktionstüchtig machen könnten.

Export der Buchungsdaten programmieren

Der Export der Buchungsdaten ist grundsätzlich aufwändiger, weil eine Zeile viel mehr Informationen enthalten kann als die Headerzeile, aber wir reduzieren die auszugebenden Daten auf einige wenige Pflichtdaten.

Als Erstes legen wir wieder eine Tabelle an, welche die zu exportierenden Daten enthält. Diese nennen wir **tblBu-**

chungsstapel. Die Tabelle sieht im Entwurf wie in Bild 6 aus.

In der Datenblattansicht zeigt die Tabelle **tblBuchungsstapel** ihre Daten wie in Bild 7 an. Diese wollen wir nun in geeigneten Formularen anzeigen.

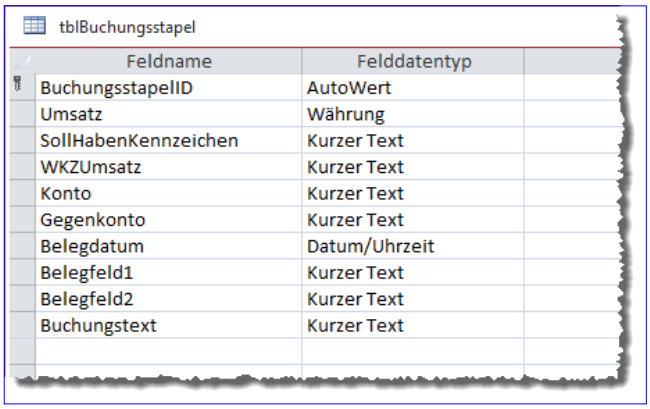


Bild 6: Entwurf der Tabelle zum Speichern der Buchungsdaten

Formulare für die Eingabe und den Export der Buchungssätze

Die Headerdaten und die Buchungsdaten sind nicht miteinander verknüpft. Die Buchungsdaten werden aber über das Datum der jeweiligen Headerdatei zugeordnet, sprich: Mit einem Header werden alle Buchungen exportiert, die innerhalb der Datumsangaben der Felder **DatumVon** und **DatumBis** liegen.

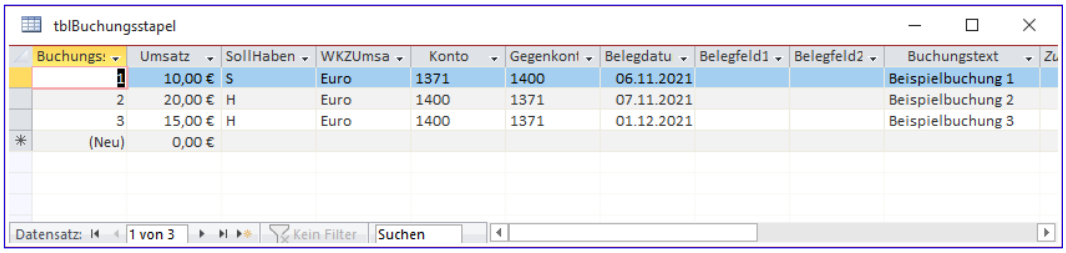


Bild 7: Beispieldaten in der Tabelle **tblBuchungsstapel**

Wir wollen in dem Formular, mit dem wir den Export steuern wollen, ein Kombinationsfeld zur Auswahl eines Headers anbieten, über das der Benutzer die zu exportierenden Buchungsdatensätze filtern kann.

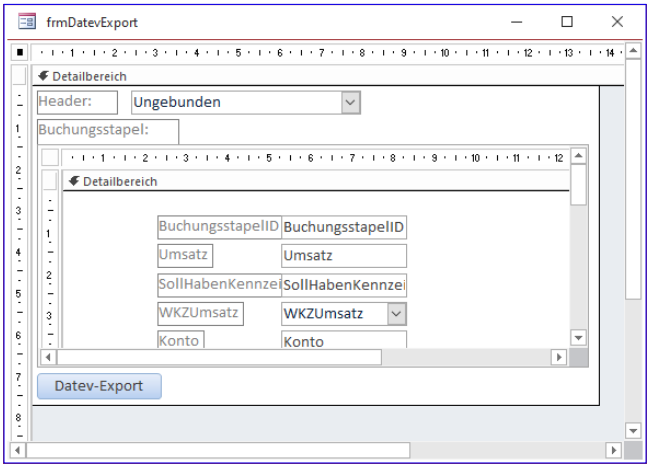


Bild 8: Entwurf des Formulars zum Steuern des Datev-Exports

Ein Unterformular soll die Datensätze der Tabelle **tblBuchungsstapel** anzeigen, die zu dem Datumsbereich des selektierten Headers passen. Der Entwurf dieser beiden Formulare sieht wie in Bild 8 aus.

Das Kombinationsfeld **cboHeader** verwendet die folgende **UNION**-Abfrage als Datensatzherkunft:

```
SELECT 0 AS ID, 'Alle anzeigen' AS Dateiname
FROM tblHeader
UNION
SELECT tblHeader.ID, tblHeader.Dateiname
FROM tblHeader;
```

Dadurch zeigt das Kombinationsfeld den Eintrag **Alle anzeigen** sowie die Werte des Feldes **Dateiname** der Tabelle **tblHeader** an (siehe Bild 9).

Damit das Kombinationsfeld gleich beim Laden des Formulars den Eintrag **Alle anzeigen** liefert, fügen wir für dieses die folgende Ereignisprozedur hinzu: