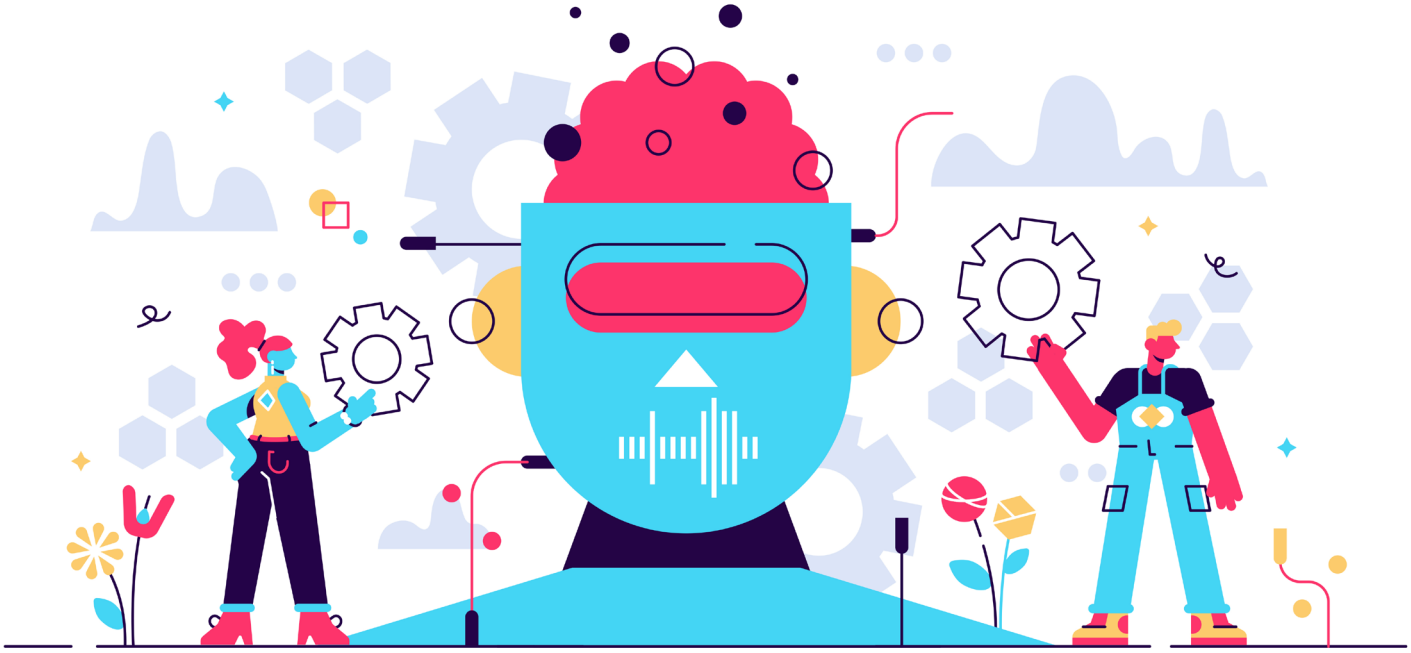
AUSGABE 02/2024

VISUAL BASIC ENTWICKLER

MAGAZIN FÜR DIE PROGRAMMIERUNG VON MICROSOFT OFFICE UND ANDEREN ANWENDUNGEN MIT VB.NET, VBA UND TWINBASIC



IN DIESEM HEFT:

André Minhorst Verlag

Programmieren mit Klasse

Softwareentwicklung mit selbst programmierten Klassen und die Verwendung von Objekten auf Basis dieser Klassen ist für viele VBA-Entwickler noch ein Buch mit sieben Siegeln. Damit wollen wir in dieser Ausgabe von Visual Basic Entwickler aufräumen und liefern gleich drei Artikel, die sich mit diesem Thema beschäftigen. Dadurch inspiriert bleiben wir im VBA-Editor und schauen uns gleich noch an, wie wir das Programmieren von Klassen codegesteuert vereinfach können. Schließlich geht es noch an die Entwicklung eines waschechten COM-Add-Ins, dass uns aufwendige Such- und Programmierarbeit abnimmt.

Die Frage, die sich bei VBA-Entwicklung in Zusammenhang mit der Programmierung eigener Klassen als Erstes stellt, ist gleichzeitig der Titel unseres ersten Artikels dieser Ausgabe: **Wozu mit Klassen programmieren?** Warum das sinnvoll sein kann, liest Du ab Seite 4.

Nachdem wir einführend über den Sinn und Unsinn geredet haben, gehen wir gleich an die Arbeit. Im Artikel **Klassen programmieren unter VBA** zeigen wir ab Seite 8, wie Klassen tatsächlich programmiert werden. Dabei erläutern wir nochmal den Unterschied zwischen Standardmodulen, Klassenmodulen von Formularen, Dokumenten und Co. und alleinstehenden Klassenmodulen, wie wir sie programmieren möchten. Dabei werfen wir einen Blick auf die Methoden und Eigenschaften von Klassen.

Nachdem die Grundlagen geklärt sind, geht es im Artikel **Ereignisse in Klassen programmieren** ab Seite 19 ans Eingemachte. Ereignisse kennen viele nur aus der Sicht des Konsumenten, indem sie Ereignisprozeduren auf der Basis von Ereignissen vorhandener Objekte programmiert haben – beispielsweise, um auf einen Mausklick, das Öffnen eines Dokuments oder das Senden einer E-Mail reagiert haben. Jetzt schauen wir uns die andere Seite an und betrachten die Programmierung von Ereignissen für unsere eigenen Klassen, die wir dann von außen in Form von Ereignisprozeduren implementieren können.

Beim Programmieren von Klassen fällt vor allem für die Definition von Eigenschaften viel Arbeit an. Für die meisten benötigen wir die Möglichkeit, dass diese von außen gelesen und geschrieben werden können. Dies regeln wir über entsprechende **Get**- und **Set**/**Let**-Prozeduren. Das ist allerdings mit viel Tipparbeit oder Copy-Paste und Anpassen verbunden. Also zeigen wir im Artikel **VBA-Editor: Quellcode-Bearbeitung automatisieren** ab Seite 25 die grundlegenden VBA-Techniken, mit denen wir solche Aufgaben vereinfachen können.

Damit wir die auf diese Weise erstellen Techniken zum Automatisieren der Quellcodebearbeitung immer im Zugriff haben, können wir beispielsweise COM-Add-Ins für den VBA-Editor nutzen. Wie wir ein solches grundsätzlich erstellen, zeigen wir im Artikel **twinBASIC: COM-Add-In für den VBA-Editor** ab Seite 41.

Und damit Du noch etwas Handfestes aus diesem Magazin mitnehmen kannst, liefern wir Dir im Artikel **Dateiauswahl-Dialog-Assistent programmieren** ab Seite 51 noch eine praktische Lösung, mit der Du mit wenigen Klicks den Code zum Anzeigen von Dateidialogen baust.

Nun viel Spaß beim Lesen!

Dein André Minhorst

Wozu mit Klassen programmieren?

Wer die Office-Anwendungen Access, Excel, Outlook, PowerPoint oder Word mit VBA programmiert, kann eine Menge erreichen, ohne jemals eine eigene Klasse zu programmieren. Wer sich die gesamten Möglichkeiten eröffnen will, kommt jedoch irgendwann nicht mehr um die Programmierung benutzerdefinierter Klassen herum. Dieser Artikel zeigt verschiedene Szenarien auf, wann man benutzerdefinierte Klassen programmieren und darauf basierende Objekte instanziieren kann und soll. Dabei geht es um Begriffe wie Eigenschaften, Methoden, Ereignisse und darum, wo und wie man Klassen einsetzt.

Wer in den VBA-Projekten von Outlook oder von Dokumenten auf Basis von Access, Excel, PowerPoint oder Word programmiert, ahnt es vielleicht manchmal nicht, aber tatsächlich programmiert man fast zwangsläufig innerhalb von Klassenmodulen. Das Modul **Projekt1**, das angezeigt wird, wenn man das VBA-Projekt von Outlook öffnet, das Modul **ThisDocument** in Word, die Module **DieseArbeitsmappe** oder **Tabelle1** in Excel oder auch die Module, die man in Access beim Anlegen von Ereignisprozeduren für Formulare, Steuerelemente oder Berichte verwendet, sind allesamt Klassenmodule.

Diese sind jedoch mit der jeweiligen Anwendung (Outlook) beziehungsweise mit dem jeweiligen Dokument oder dem Element verknüpft, in deren Kontext sie geöffnet werden. Dadurch ergeben sich einige Vorteil: Wir können nämlich in diesen direkt auf einige Elemente des jeweiligen Objekts zugreifen. Im Klassenmodul **ThisOutlookSession** von Outlook greifen wir beispielsweise direkt auf die Eigenschaften, Methoden und Ereignisse des **Application**-Objekts zu.

Im Klassenmodul eines Access-Formulars greifen wir direkt auf die Eigenschaften, Methoden und Ereignisse des jeweiligen Formulars und der enthaltenen Steuerelemente zu. Im Klassenmodul **ThisDocument** eines Word-Dokuments stehen die entsprechenden Elemente des **Document**-Objekts zur Verfügung und in Excel können wir in der Klasse **DieseArbeitsmappe** über die Variable **Workbook** auf die enthaltenen Eigenschaften, Methoden und Ereignisse zugreifen sowie in den Klassen für die einzelnen Tabellen auf das jeweilige **Worksheet**-Element.

Wenn wir bereits eine solche Vielfalt von eingebauten Elementen haben, warum sollten wir noch benutzerdefinierte Klassen entwickeln? Dazu gibt es verschiedene Gründe, die wir in den folgenden Abschnitten beschreiben. Außerdem sehen wir uns an, warum es in diesem Fällen ein Klassenmodul sein muss und warum es nicht reicht, ein Standardmodul zu verwenden.

Grund für Klassen: Eigenschaften, Methoden und Ereignisse zusammenfassen

Ein wichtiger Anwendungszweck für Klassen ist schlicht und einfach das Zusammenfassen von Elementen in einer Code-Einheit. Wann immer wir im Code allein Informationen zusammenstellen, die zusammengehören, können wir dies mit einer Klasse erledigen.

Für diese legen wir dann Eigenschaften fest, mit denen wir die Informationen erfassen können. Sobald wir ein Objekt auf Basis der Klasse erstellt haben, können wir diesem Informationen über die Eigenschaften zuweisen und die Eigenschaften wieder auslesen. Wir können auch dafür sorgen, dass das Objekt seine Eigen-

schaften selbst füllt – beispielsweise durch das Einlesen eines Datensatzes einer Access-Tabelle, der Daten einer E-Mail oder der Absätze eines Word-Dokuments.

Auf Basis dieser Eigenschaften können wir nun Methoden bereitstellen, welche die Inhalte der Eigenschaften verarbeiten – beispielsweise, um aus Daten wie Absender, Empfänger, Betreff und Inhalt eine Outlook-E-Mail zu erstellen und diese abzusenden. Oder wir bieten Funktionen an, welche die eingegebenen Eigenschaften in unterschiedlichen Arten verarbeiten.

So können wir beispielsweise einer Klasse, die eine Adresse in Form der Eigenschaften Firma, Anrede, Vorname, Nachname, Straße, PLZ, Ort und Land aufnimmt, eine Funktion hinzufügen, welche aus diesen Informationen einen Adressblock erstellt, den wir in einem Access-Bericht oder in einem Word-Dokument ausgeben können.

Ein Beispiel hierfür stellen wir im Artikel **Klassen programmieren unter VBA** (**www.vbentwickler.de/423**) vor.

Grund für Klassen: Ereignisse bereitstellen

Wenn Du selbst Ereignisse bereitstellen möchtest,

Ein gutes Beispiel dafür ist eine Schaltfläche. Diese erledigt beim Anklicken bereits automatisch Aufgaben – zum Beispiel ändert sie ihre Ansicht in den gedrückten Zustand. Du möchtest aber für jede Schaltfläche individuelle Funktionen programmieren. So soll zum Beispiel eine **OK**-Schaltfläche in einem Formular andere Aktionen ausführen als beispielsweise eine **Abbrechen**-Schaltfläche. Deshalb stellt die Schaltflächen-Klasse für jede einzelne Instanz einer Schaltfläche eine eigene Ereignisprozedur bereit, in der Du die jeweiligen Anweisungen unterbringen kannst.

Wenn Du selbst eine Klasse programmierst, wird es für Ereignisse ebenfalls zwei Arten von Aktionen geben:

- solche, die immer ausgeführt werden sollen und
- solche, die individuell implementiert werden sollen, um die Klasse an den jeweiligen Anwendungsfall anzupassen.

Beispiel: Klasse zum Einlesen von XML-Dokumenten

Nehmen wir an, Du hast eine Klasse programmiert, die das Einlesen von XML-Dateien erledigt. Die Klasse nimmt mit einer Eigenschaft namens **Pfad** den Pfad

Klassen programmieren unter VBA

Neben den Standardmodulen gibt es in VBA-Projekten auch noch einen weiteren Typ von Modulen, nämlich die Klassenmodule. Diese kommen wiederum in zwei Ausführungen: Es gibt alleinstehende Klassenmodule und eingebaute Klassenmodule, die den Code zu bestimmten Objekten enthalten – wie beispielsweise die Klassenmodule zu Formularen und Berichten in Access, zu Word-Dokumenten für das Dokument oder in Excel für Worksheet und Workbook. Auch in Outlook gibt es Klassenmodule. Wir wollen uns an dieser Stelle jedoch auf die alleinstehenden Klassenmodule konzentrieren, also auf solche, die wir selbst anlegen müssen. Hier schauen wir uns an, warum man diese überhaupt nutzen sollte, welche Anwendungszwecke es gibt und welche Best Practices sich für uns etabliert haben.

Unterschied zwischen Standardmodulen, eingebauten Klassenmodulen und benutzerdefinierten Klassenmodulen

Zunächst einmal wollen wir den Unterschied zwischen den verschiedenen Typen von Modulen erläutern.

Standardmodule

Wir beginnen mit den Standardmodulen:

- Standardmodule sind einfache Container für VBA-Code, die in einer VBA-Projektdatei enthalten sind und die selbst hinzufügen müssen.
- Sie enthalten normalerweise Funktionen, Prozeduren und Variablen, die in verschiedenen Teilen des Projekts verwendet werden können.
- Ein Standardmodul kann Funktionen und Prozeduren enthalten, die unabhängig voneinander sind und direkt vom Rest des Codes aufgerufen werden können.
- Standardmodule werden oft verwendet, um allgemeine Funktionen und Hilfsroutinen zu speichern, die von verschiedenen Teilen des Projekts benötigt werden.

Klassenmodule allgemein

Wir schauen uns erst einmal die allgemeinen Eigenschaften an, die allen Klassenmodulen gemein sind:

- Klassenmodule ermöglichen die Definition benutzerdefinierter Datenstrukturen, die als Objekte bezeichnet werden.
- Sie dienen dazu, spezifische Arten von Objekten zu definieren, die Eigenschaften, Methoden und Ereignisse haben können.

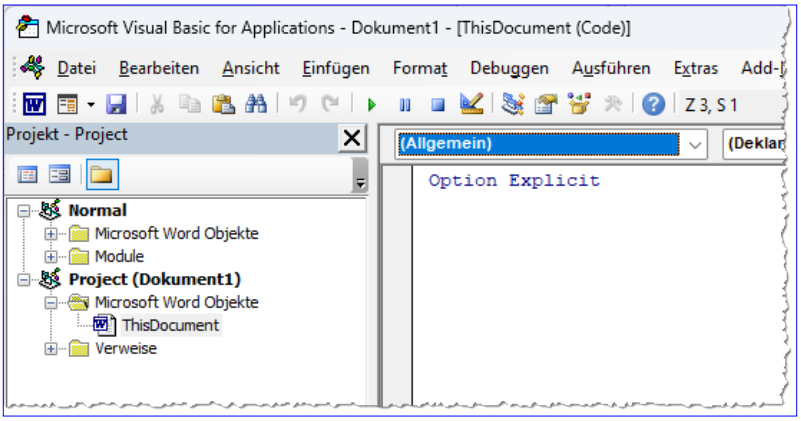


Bild 1: Ein eingebautes Klassenmodul, hier in Word.

- Klassenmodule ermöglichen die Erstellung von Objekten mit spezifischen Verhaltensweisen und Datenstrukturen, was die Codeorganisation und -wiederverwendbarkeit verbessert.

- Sowohl eingebauten als auch alleinstehenden Klassenmodulen können wir benutzerdefinierte Eigenschaften, Methoden und Ereignisse hinzufügen.

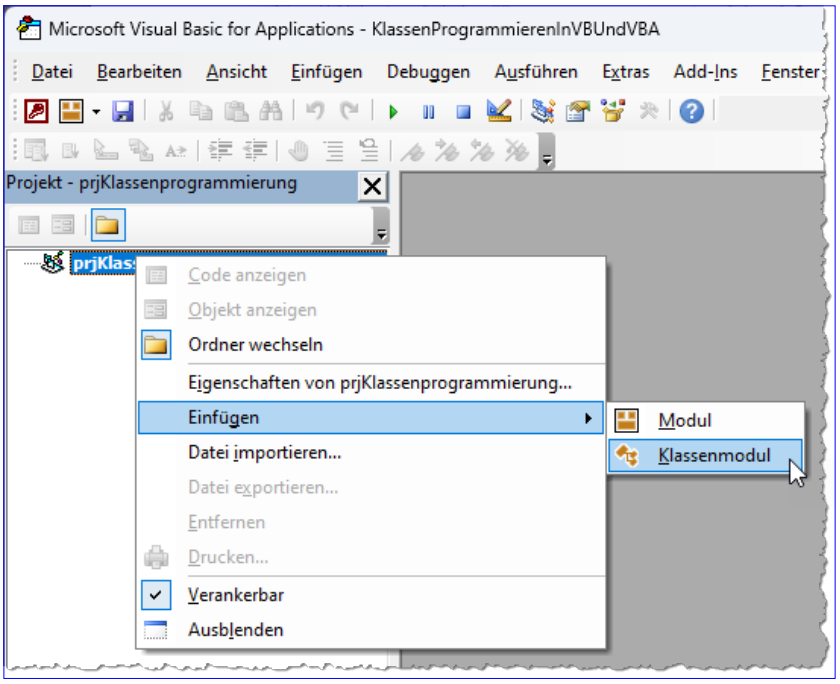


Bild 2: Hinzufügen eines Klassenmoduls

Eingebaute Klassenmodule

Unter eingebauten Klassenmodulen verstehen wir solche Klassenmodule, wie sie beispielsweise in Word, Excel, Outlook oder Access automatisch mit den dortigen Objekten angelegt werden oder dort bereits vorhanden sind, wenn wir den VBA-Editor erstmalig öffnen.

Bild 1 zeigt ein Klassenmodul eines Word-Dokuments.

- Klassenmodule können Ereignisse enthalten, die es Objekten ermöglichen, auf bestimmte Aktionen zu reagieren, wie das Öffnen oder Schließen einer Arbeitsmappe in Excel, das Öffnen eines Word-Dokuments oder das Anzeigen eines Formulars in Access.

- Sie enthalten vorgefertigte Ereignisse, auf deren Basis wir Ereignisprozeduren implementieren können.

Alleinstehende Klassenmodule

Im Gegensatz zu den eingebauten Klassenmodulen können wir beliebig viele alleinstehende Klassenmodule erstellen. Dazu nutzen wir den Kontextmenübefehl **Einfügen|Klassenmodul** des Projekt-Explorers (siehe Bild 2) oder den gleichnamigen Befehl der Menüleiste des VBA-Editors.

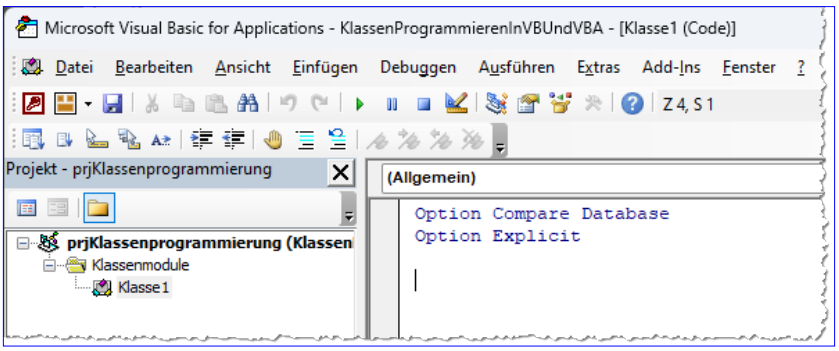


Bild 3: Hinzufügen eines Klassenmoduls

Damit erzeugen wir erst einmal einen neuen Eintrag im Projekt-Explorer, der standardmäßig **Klasse1** genannt und direkt als neues Fenster im VBA-Editor angezeigt wird (siehe Bild 3).

In dieses Fenster können wir nun Code schreiben, genau wie in einem Standardmodul. Wir können den dort enthaltenen Code aber nicht so einfach ausführen in einem Standardmodul.

Wir müssen die Klasse erst in Form eines Objekts instanziieren und dieses einer Objektvariablen zuweisen, damit wir auf seine Eigenschaften, Methoden oder Ereignisse zugreifen können.

Klassenmodul umbenennen

Als Erstes erledigen wir allerdings eine wichtige Aufgabe: Das Klassenmodul soll einen anderen Namen als **Klasse1** erhalten.

Zu Beispielzwecken wollen wir die Klasse **clsAdresse** nennen. Das Präfix **cls** steht für **Class**, also Klasse. Mit der Klasse wollen wir die typischen Eigenschaften einer Adresse aufnehmen und abfragen können. Nach dem Umbenennen sehen wir den neuen Klassennamen gleich an mehreren Stellen (siehe Bild 5).

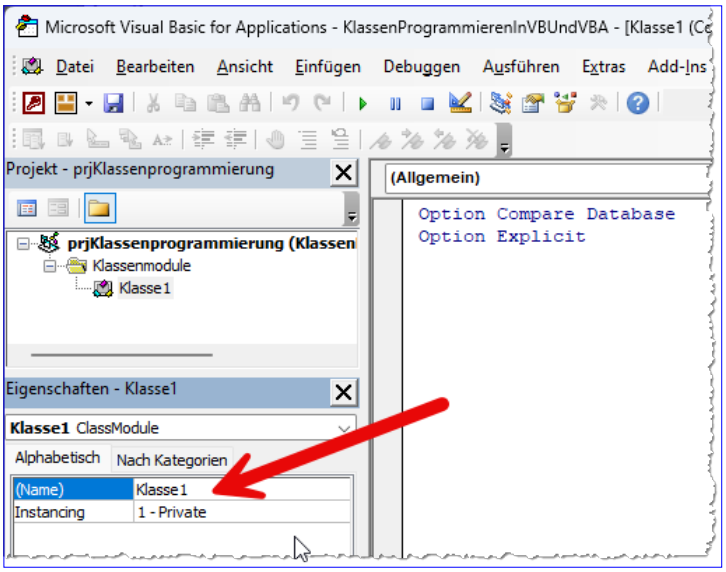


Bild 4: Umbenennen eines Klassenmoduls

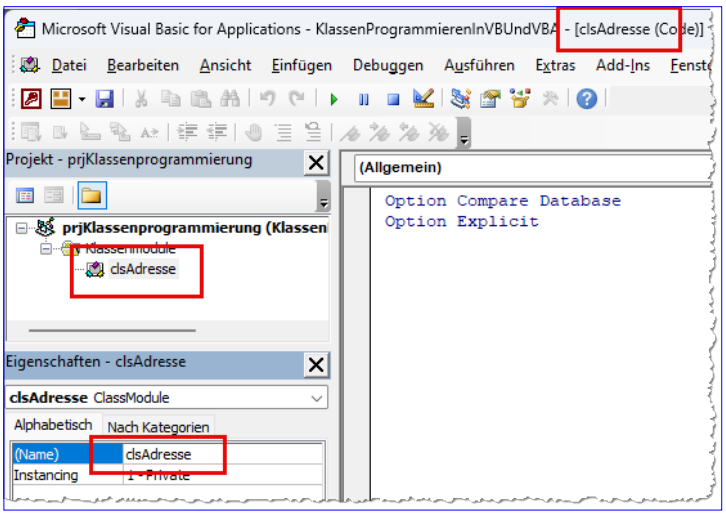


Bild 5: Der neue Klassenname tritt gleich an drei Stellen in Erscheinung.

Welche Elemente machen eine Klasse nach außen hin aus?

Es gibt drei verschiedene Elemente, die wir von außen nutzen können, die eine Klasse charakterisieren:

- **Eigenschaften**: Eigenschaften erscheinen als lesbare und/oder schreibbare Eigenschaften. Wir können für Klassen beliebig viele Eigenschaften festlegen und durch bestimmte Schreibweisen angeben, ob man diese bei der Programmierung der Klasse lesend, schreibend oder lesend und schreibend zugreifbar möchte.

- **Methoden/Funktionen**: Dies sind im Prinzip **Sub**- und **Function**-Prozeduren wie in Klassenmodulen, nur dass sie nur aufgerufen werden können, wenn es eine Objektvariable mit einem Verweis auf eine Instanz der Klasse gibt, über die wir auf diese Methoden und Funktionen zugreifen können. Was Instanzen und Objektvariablen sind, erläutern wir weiter unten.

- **Ereignisse**: Ereignisse sind eine echte Besonderheit von Klassenmodulen gegenüber Standardmodulen. Die Ereignisse von eingebauten Standardmodulen werden meist durch Benutzeraktionen ausgelöst. Wir können dann Ereignisprozeduren programmieren, die als Reaktion auf solche Benutzeraktionen gestartet werden sollen – zum Beispiel beim Öffnen eines Dokuments, beim Anklicken einer Schaltfläche, beim Absenden einer E-Mail et cetera. Bei Klassenmodulen können wir allerdings selbst genau

festlegen, wann ein solches Ereignis ausgelöst werden soll. Wie das gelingt und wozu wir das nutzen können, zeigen wir ebenfalls weiter unten.

Welche Elemente gibt es noch innerhalb einer Klasse?

Innerhalb von Klassenmodulen können wir all die von herkömmlichen Modulen bekannten Elemente nutzen – Prozeduren, Funktionen, Variablen, Konstanten und so weiter.

Es gibt allerdings einen wichtigen Unterschied zwischen den Eigenschaften, Methoden und Ereignissen, die wir bei Vorhandensein einer Instanz der Klasse nutzen können, und den übrigen Elementen: Die Eigenschaften, Methoden und Ereignisse sind öffentliche Elemente, wir müssen diese daher mit dem Schlüsselwort **Public** deklarieren. Die Elemente, die wir nur innerhalb des Klassenmoduls nutzen wollen, deklarieren wir mit dem Schlüsselwort **Private**. Diese sind nicht von außerhalb sichtbar.

Instanziieren einer Klasse

Da unsere Klasse noch keine Elemente enthält, also keine Eigenschaften, Methoden oder Ereignisse, behelfen wir uns für die Beschreibung der Instanziierung einer Klasse einer eingebauten Klasse. In diesem Fall wollen

Objekts gespeichert. Das Objekt selbst befindet sich an der angegebenen Stelle im Speicher.

Doch schreiten wir zur Tat. Wir benötigen zwei Schritte, um eine Objektvariable mit einer neuen Instanz eines Objekts zu füllen:

- das Deklarieren der Objektvariablen und
- das Instanziieren des Objekts auf Basis der Klasse und Zuweisen an die Objektvariable.

Bei der Deklaration verwenden wir den Namen der Klasse als Datentyp:

```
Dim objCollection As Collection
```

Das Instanziieren erfolgt mit dem Schlüsselwort **New** mit dem Typ des zu erzeugenden Elements, hier **Collection**, und dem Zuweisen an die Objektvariable:

```
Set objCollection = New Collection
```

Wir können diese beiden Anweisungen sogar in einer einzigen Anweisung zusammenfassen:

```
Dim objCollection As New Collection
```

Ereignisse in Klassen programmieren

Eines der wichtigsten Features vieler eingebauter Klassen in den Office-Anwendungen sind die Ereignisse. Damit können wir Ereignisprozeduren implementieren, mit denen wir beispielsweise auf das Anklicken von Schaltflächen, dem Öffnen oder Schließen von Dokumenten oder dem Wechseln eines Tabellenblatts in einem Excel-Arbeitsblatt reagieren können. Wenn man selbst Klassen programmiert, findet sich früher oder später ein Anlass, dieser ein eigenes Ereignis hinzuzufügen, dass durch eine bestimmte Aktion ausgelöst wird. Dieser Artikel zeigt, wir wie solche Ereignisse programmieren und auslösen und wie wir diese in Form von Ereignisprozeduren in den Klassen implementieren, welche das entsprechende Objekt instanziiert haben.

Ereignisse in Klassenmodulen

Wer schon einmal eine Ereignisprozedur beispielsweise für das Anklicken einer Schaltfläche, das Öffnen eines Word-Dokuments, das Markieren eines Ranges in Excel et cetera programmiert hat, kennt zumindest schon einmal die Verbraucherseite von Ereignissen.

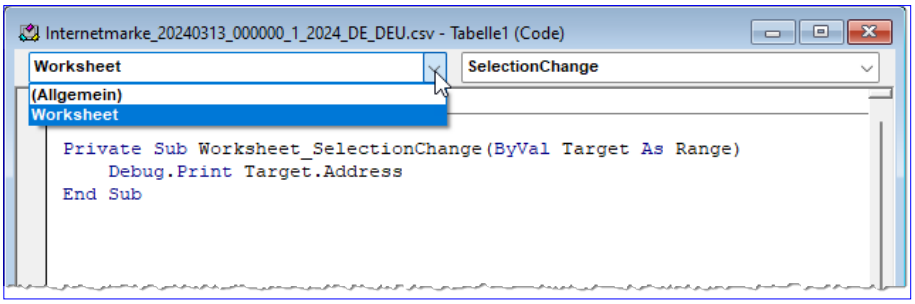


Bild 1: Ereignisprozedur beim Ändern der Markierung in einem Excel-Arbeitsblatt

Hier können wir zum Beispiel in Excel in der Klasse **Tabelle1** eines Excel-Dokuments im Codefenster im linken Kombinationsfeld den Eintrag **Worksheet** auswählen, was im rechten Kombinationsfeld automatisch die Standardmethode **SelectionChange** selektiert. Dadurch wird automatisch die Ereignisprozedur **Worksheet_SelectionChange** angelegt, mit der wir in diesem Fall die Adresse des neu ausgewählten Bereichs im Direktbereich des VBA-Editors ausgeben (siehe Bild 1).

Nun schauen wir uns die andere Seite an, auf der wir Ereignisse definieren, die wir dann wiederum als Verbraucher nutzen können. Wozu aber überhaupt Ereignisse? Weil wir benutzerdefiniert auf bestimmte Ereignisse reagieren können wollen. Stellen wir uns beispielsweise vor, wir wollen den Benutzer unserer Klasse **clsAdresse** darüber informieren, wenn die Methode **Leeren** erfolgreich aufgerufen wurde. Dann können wir das einfach erledigen, indem wir der Methode **Leeren** eine entsprechende Meldung hinzufügen:

```
Public Sub Leeren()
    ...
    m_Land = ""
    MsgBox "Die Eigenschaften der Adresse wurden geleert.", _
        vbOKOnly + vbExclamation, "Adresse geleert"
End Sub
```

Rufen wir die Methode nun von einer anderen Prozedur aus auf, erscheint diese Meldung (siehe Bild 2).

Nun stellen wir uns vor, wir wollen die Meldung dieser Klasse nicht innerhalb der Klasse verdrahten, sondern dem Entwickler, der diese Klasse in seinem Code ver-

wendet (meist wir selbst), die Möglichkeiten geben wollen, individuell auf dieses Ereignis zu reagieren.

Das ist ein Beispieleinsatzzweck für die Bereitstellung eines Ereignisses. Und das gelingt in zwei Schritten:

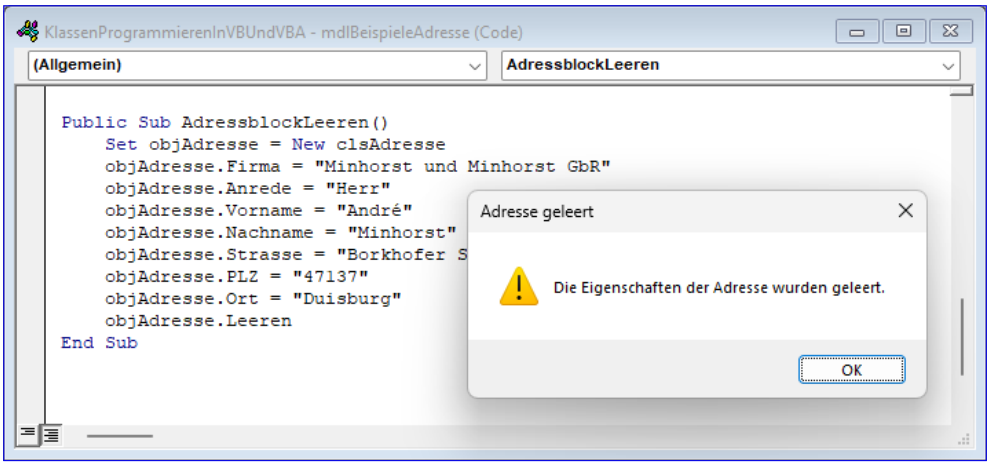


Bild 2: Meldung beim Aufruf einer Methode

- Als Erstes definieren wir das Ereignis mit dem Schlüsselwort **Event**, dem Namen des Ereignisses und gegebenenfalls den zu verwendenden Parametern.

- Als zweites fügen wir in diesem Beispiel der Methode Leeren Code hinzu, der dieses Ereignis auslöst. Dabei verwenden wir das Schlüsselwort **RaiseEvent** und übergeben die Werte für die Parameter, welche das Verbraucher des Ereignisses erhalten soll.

In der Klasse fügen wir oben die Deklaration des Ereignisses ein:

```
Public Event NachDemLeeren()
```

Der Prozedur **Leeren** fügen wir am Ende den Befehl **RaiseEvent** hinzu und geben den Namen des Ereignisses als Parameter an:

```
Public Sub Leeren()
    ...
    RaiseEvent NachDemLeeren
End Sub
```

Dieses wird, wie in Bild 3 dargestellt, per IntelliSense angeboten. Dadurch geschieht nun erst einmal nichts. Wir können die obige Prozedur **AdressblockLeeren** starten und die Methode **Leeren** aufrufen, aber wir erhalten keinen sichtbaren Effekt. Das ist logisch, denn wir haben das Ereignis noch nicht implementiert.

Keine Ereignisprozeduren ohne das Schlüsselwort WithEvents

Der Schritt ist auch in dieser Konstellation nicht möglich, denn um die Ereignisse einer Klasse zu implementieren, müssen die Klasse, die das Ereignis enthält, mit einem bestimmten Schlüsselwort deklarieren. Dieses heißt **WithEvents**. Es gibt noch weitere Bedingungen: Diese Deklaration kann nicht innerhalb einer Prozedur erfolgen, sondern muss im Kopf des Moduls hinterlegt werden. Außerdem können wir das Schlüsselwort **WithEvents** nicht innerhalb von Standardmodulen verwenden, sondern nur in Klassenmodulen. Versuchen wir dies dennoch, erhalten

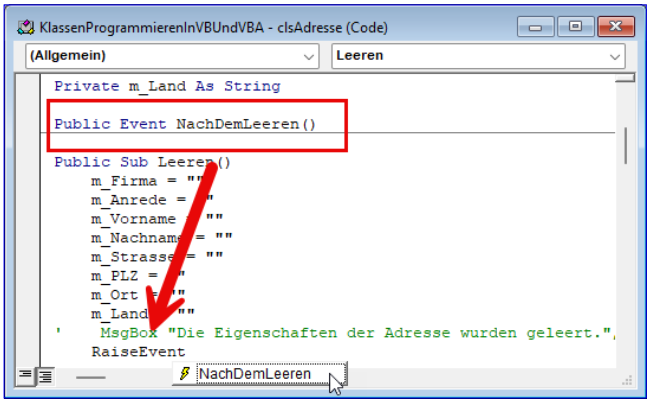


Bild 3: Deklarieren und Auslösen eines Ereignisses

wir direkt einen Kompilierfehler wie in Bild 4.

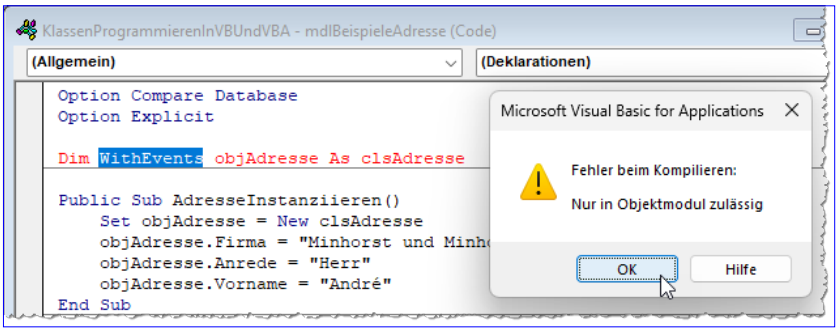


Bild 4: Fehler beim Versuch, **WithEvents** in einem Standardmodul zu verwenden

Warum aber benötigen wir dieses Schlüsselwort überhaupt? Wenn wir in Word auf die Ereignisse des **Document**-Objekts zugreifen oder in Access auf ein **Form**- oder **Button**-Element, können wir das auch einfach ohne **WithEvents**-Schlüsselwort erledigen.

Das ist dann allerdings reiner Zufall: Bei Word waren wir dann vermutlich im Klassenmodul zum Word-Dokument und in Access im Klassenmodul des Formulars, in dem sich auch die Schaltfläche befand, deren Ereignisse wir implementiert haben. Da es sich um das Klassenmodul des Objekts selbst handelt, stehen dort alle Ereignisse standardmäßig zur Verfügung.

Würden wir jedoch die Ereignisse dieses Word-Dokuments oder Access-Formulars von einem anderen Klassenmodul aus implementieren wollen, müssten wir dazu auch erst einmal eine Objektvariable anlegen, die wir mit dem Schlüsselwort **WithEvents** deklarieren. Und wir müssten diese auch noch mit einem Verweis auf das entsprechende Objekt versehen.

diesem Zweck ein neues Klassenmodul namens **clsAdressenVerwalten**.

Diese fügen wir die Deklaration der Objektvariablen ein, der wie die Instanz der Klasse **clsAdresse** zuweisen wollen:

```
Private WithEvents objAdresse As clsAdresse
```

Außerdem fügen wir eine Methode hinzu, die das Objekt erstellt und seine Eigenschaften füllt:

```
Public Sub AdresseErstellen()
    Set objAdresse = New clsAdresse
    With objAdresse
        .Anrede = "Herr"
        .Vorname = "André"
```

VBA-Editor: Quellcode-Bearbeitung automatisieren

Der VBA-Editor ist die Gemeinsamkeit der Office-Anwendungen wie Access, Excel, Outlook, PowerPoint und Word. Wer eine oder mehrere dieser Anwendungen programmiert, um dem Benutzer die Arbeit damit zu erleichtern, kennt sich mehr oder weniger mit VBA aus. In der Regel wird im Arbeitsalltag eines Entwicklers jede Codezeile von Hand neu programmiert. Gegebenenfalls kopiert man bestehenden Code und passt diesen an den jeweiligen Anwendungszweck an. Aber warum nicht einen Schritt weitergehen und VBA-Code zur Erstellung von VBA-Code selbst nutzen? Visual Basic bietet eigens zum Zweck der Programmierung der Elemente und des Codes im VBA-Editor eine eigene Bibliothek namens Microsoft Visual Basic for Applications Extensibility 5.3. Welche Möglichkeiten diese allgemein bietet und welche Elemente, Eigenschaften und Methoden sie bereitstellt, schauen wir uns in diesem Artikel an.

Der VBA-Editor ist die Entwicklungsumgebung für die Programmierung der verschiedenen Office-Anwendungen und der darin enthaltenen Dokumente und Elemente. Er zeigt im Projekt-Explorer die programmierbaren Elemente wie Standardmodule und Klassenmodule an und erlaubt es, den Code dieser Elemente im Code-Editor anzuzeigen und zu bearbeiten.

Die Codeeingabe selbst ist bis auf wenige Hilfestellungen durch IntelliSense Aufgabe des Entwicklers – hier ist viel Tipparbeit angezeigt oder gegebenenfalls noch das Kopieren und Anpassen bereits vorhandenen Codes.

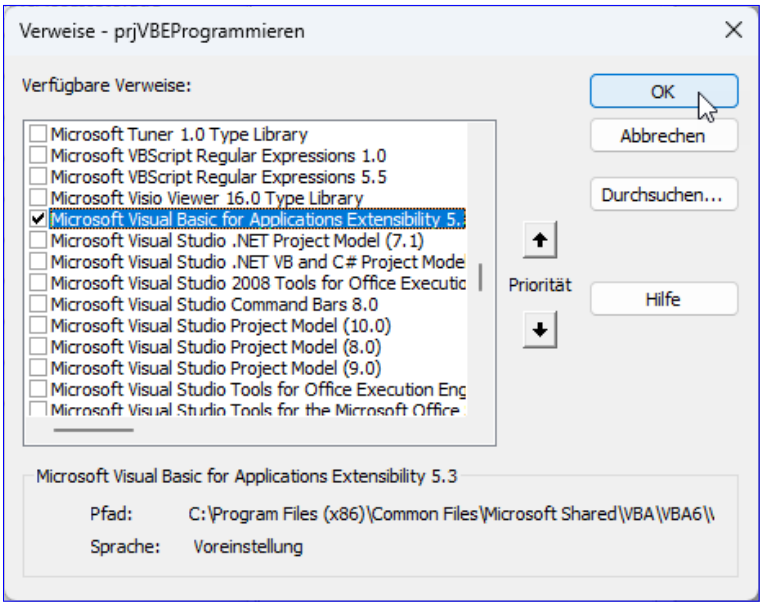


Bild 1: Setzen eines Verweises auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3**

IntelliSense unterstützt uns allerdings auch nur dabei, den Datentyp für Variablen auszuwählen oder die Eigenschaften oder Methoden von Objektvariablen zu selektieren. Wenn wir mehr aus der Entwicklungsumgebung herausholen wollen, müssen wir uns also selbst helfen.

Dabei können wir bereits mit den üblichen Mitteln von VBA Einiges erreichen: So können wir die Zeichenkettenfunktionen in Kombination mit Schleifen nutzen, um bestimmte Abfolgen von Anweisungen im Direktbereich des VBA-Editors auszugeben, die wir dann kopieren und in unseren Code einfügen.

Wir können jedoch noch viel mehr erreichen, wenn wir eine Bibliothek namens **Microsoft Visual Basic for Applications Extensibility 5.3** einbinden. Diese fügen wir über den **Verweise**-Dialog, den wir mit dem

Menübefehl **Extras|Verweise** öffnen, zum aktuellen VBA-Projekt hinzu. Dazu versehen wir den Eintrag wie in Bild 1 mit einem Haken.

Die Klassen und Elemente der VBE-Bibliothek

Anschließend können wir uns im Objektkatalog, zu öffnen mit der Taste **F2**, die Klassen und Enumerationen dieser Bibliothek ansehen (siehe Bild 2).

Damit nur die Elemente der hier **VBIDE** genannten Bibliothek angezeigt werden, haben wir diesen Eintrag im Kombinationsfeld ganz oben ausgewählt.

Danach sehen wir in der linken Liste alle Klassen und Enumerationen dieser Bibliothek und rechts die Elemente des aktuell markierten Eintrags der linken Liste, hier der Hauptklasse **VBE**.

Und hier ist eine kurze Beschreibung der Elemente, die wir direkt über die Hauptklasse erreichen können:

- **ActiveCodePane**: Aktueller Codebereich. Codebereich ist erklärungsbedürftig, darauf gehen wir weiter unten ein.
- **ActiveVBProject**: Aktuelles VBA-Projekt. Ermittelt das aktive von gegebenenfalls mehreren aktuell geöffneten VBA-Projekten im VBA-Editor.
- **ActiveWindow**: Aktuelles Fenster. Zu den Fenstern zählen nicht nur die Codefenster, sondern auch der Direktbereich, der Projekt-Explorer et cetera.
- **Addins**: Auflistung der COM-Add-Ins, die aktuell für den VBA-Editor registriert sind.
- **CodePanes**: Auflistung aller **CodePane**-Elemente.
- **CommandBars**: Auflistung aller **CommandBar**-Elemente, also aller Menüleisten des VBA-Editors.

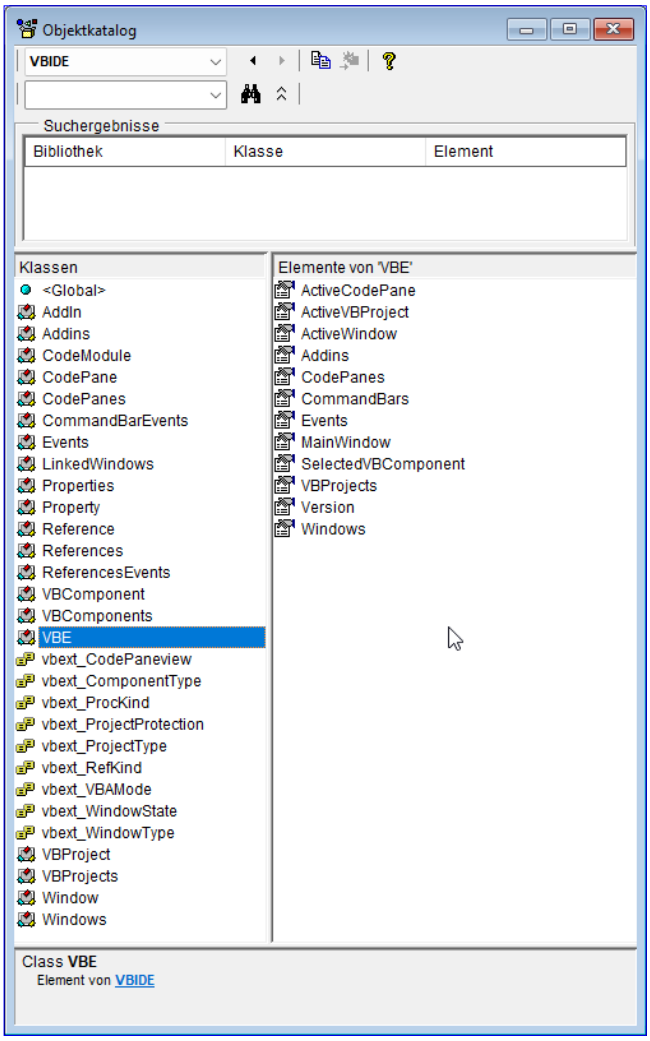


Bild 2: Die Elemente der Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3**

- **Events**: Auflistung aller **Event**-Auflistungen, enthält die beiden Auflistungen **CommandBarEvents** und **ReferencesEvents**.
- **MainWindow**: Verweis auf das Hauptfenster des VBA-Editors
- **SelectedVBComponent**: Verweis auf die aktuell im Projekt-Explorer selektierte Komponente, also beispielsweise ein Modul
- **VBProjects**: Auflistung aller VBA-Projekte, die aktuell im Projekt-Explorer angezeigt werden.

- **Version**: Version von VBA, aktuell **7.01**
- **Windows**: Auflistung aller Fenster des VBA-Editors. Dabei werden auch einige nicht geöffnete Fenster aufgelistet.

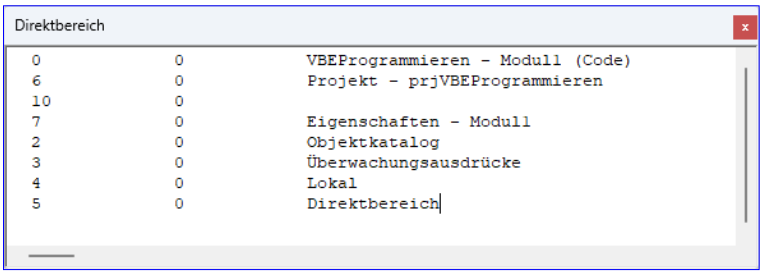


Bild 3: Ausgabe der Fenster mit einigen Eigenschaftswerten

Deklarieren und Referenzieren

Beim Deklarieren der Elemente der VB-Editor-Bibliothek können wir die übergeordnete Klasse **VBIDE** verwenden. Damit erhalten wir dann alle Elemente dieser Bibliothek per IntelliSense:

```
Dim objWindow As VBIDE.Window
```

Bei Referenzieren nutzen wir hingegen einfach **VBE**, zum Beispiel:

```
Set objWindow = VBE.ActiveWindow
```

Fenster im VBA-Editor programmieren

Die Fenster im VBA-Editor können wir mit der **Windows**-Auflistung durchlaufen. Dabei verwenden wir als Laufvariable der **For Each**-Schleife eine Variable des Typs **Window**:

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

Hier geben wir den Typ, den Fensterstatus und die Beschriftung aus, was bei einem geöffneten Modul wie in Bild 3 aussieht.

Was die einzelnen Zahlen für die Eigenschaft **Type** bedeuten, können wir der Enumeration **vbext_WindowType** entnehmen, die wir im Objektkatalog finden.

0 steht beispielsweise für **vbext_wt_CodeWindow**. Die anderen Werte stehen für die jeweiligen Fenstertypen wie Direktbereich, Projekt-Explorer und so weiter.

Wenn wir einmal die Fenster und ihre Position steuern wollen, können wir diese und andere Eigenschaften der **Window**-Objekte nutzen. Diese liefern beispielsweise auch die Position und Größe der Fenster.

VBA-Projekte

Der VBA-Editor kann durchaus einmal mehrere VBA-Projekte gleichzeitig im Projekt-Explorer anzeigen, beispielsweise wenn aktuell ein Add-In geladen ist.

Diese können wir über die Auflistung **VBProjects** durchlaufen:

```
Public Sub ProjekteDurchlaufen()
    Dim objVBProject As VBIDE.VBProject
    For Each objVBProject In VBE.VBProjects
        Debug.Print objVBProject.Name
    Next objVBProject
End Sub
```

Das Element **VBProject** liefert neben dem Namen noch einige weitere interessante Eigenschaften:

- **Filename**: Liefert den Namen der Datei, in dem sich das VBA-Modul befindet. Das kann hilfreich sein, um das VBA-Modul zum aktuell geöffneten Dokument zu ermitteln.

- **VBComponents**: Liefert eine Auflistung aller **VBComponent**-Elemente, also der VBA-Module.

Aktuelles VBA-Projekt ermitteln

Wenn mehrere VBA-Projekte geladen sind, liefert **ActiveVBProject** abhängig von dem Ort, wo es ausgeführt wird, unterschiedliche Ergebnisse. Wenn wir es im Direktbereich aufrufen, erhalten wir einen Verweis auf das VBA-Projekt, das gerade im Projekt-Explorer den Fokus hat. Wenn wir es aus einem Modul heraus aufrufen, erhalten wir mit **ActiveVBProject** einen Verweis auf das VBA-Projekt, in dem sich das aufrufende Modul befindet.

VBA-Projekt zu einem bestimmten Dokument ermitteln

Manchmal kann es interessant sein, genau das VBA-Projekt zu ermitteln, das zu einem bestimmten Dokument oder einer Datenbank gehört. Voraussetzung dafür ist, dass wir den Pfad des aktuellen Dokuments kennen. Dann können wir einfach die Liste der VBA-Projekte im VBA-Editor durchlaufen und prüfen, ob der mit der Eigenschaft **Filename** gelieferte Pfad mit dem Pfad des Dokuments übereinstimmt.

Die VBA-Komponten oder: VBComponents

Die Auflistung **VBComponents** des **VBProject**-Objekts liefert alle **VBComponent**-Elemente des VBA-Projekts. Im folgenden Beispiel durchlaufen wir alle Elemente und lassen uns den Namen und den Typ ausgeben:

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

Die verschiedenen Typen können wir der Enumeration **vbext_ComponentType** entnehmen. Hier finden wir:

- **11**: **vbext_ct_ActiveXDesigner**
- **2**: **vbext_ct_ClassModule**
- **100**: **vbext_ct_Document**
- **3**: **vbext_ct_MSForm**
- **1**: **vbext_ct_StdModule**

Neue VB-Komponente anlegen

Für das Bearbeiten von VBA-Projekten kann es hilfreich sein, neue Module anzulegen.

Module sind **VBComponent**-Objekt, also nutzen wir dazu die Methode **Add** der **VBComponents**-Auflistung.

Dieser übergeben wir für ein neues Standardmodul den Wert **vbext_ct_StdModule** als Parameter und referenzieren das neue Modul mit der Variablen **objVBComponent**. Diesem geben wir anschließend direkt einen neuen Namen:

```
Public Sub NeuesModul()
    Dim objVBComponent As VBIDE.VBComponent
    Set objVBComponent = VBE.ActiveVBProject. _
        VBComponents.Add(vbext_ct_StdModule)
    objVBComponent.Name = "mdlNeuesModul"
End Sub
```

Würden wir beispielsweise ein Klassenmodul erstellen wollen, ginge das mit folgender Anpassung:

```
Set objVBComponent = VBE.ActiveVBProject. _
    VBComponents.Add(vbext_ct_ClassModule)
objVBComponent.Name = "clsNeueKlasse"
```

Von der Komponente zum Modul

Mit dem neuen **VBComponent**-Objekt können wir noch nicht allzu viel anfangen, wenn wir seinen Code anpassen wollen. Dazu müssen wir das **CodeModule**-Objekt referenzieren, dass sich hinter der gleichnamigen Eigenschaft des **VBComponent**-Elements verbirgt.

CodeModule referenzieren

Es gibt allerdings verschiedene Wege, ein **CodeModule**-Element zu referenzieren. Der erste ist der Weg über das **VBComponent**-Element, in dem sich das Modul befindet. Wenn wir den Namen des **VBComponent**-Elements kennen, gelingt das beispielsweise wie folgt:

```
Public Sub MitModulArbeiten()
    Dim objCodeModule As VBIDE.CodeModule
    Set objCodeModule = VBE.ActiveVBProject. _
        VBComponents("mdlNeuesModul").CodeModule
    Debug.Print objCodeModule.Lines(1, 10)
End Sub
```

Um zu prüfen, ob wir das richtige **CodeModule**-Element referenziert haben, geben wir die mit der **Line**-Methode ermittelten ersten zehn Zeilen des Moduls im Direktbereich des VBA-Editors aus.

Objekt. Auf dieses Objekt kommen wir gleich im Anschluss zu sprechen.

Wichtig ist, dass dieses genau wie **VBComponent** ebenfalls die Eigenschaft **CodeModule** bietet, mit der wir auf das **CodeModule**-Element zugreifen können.

Mit dem Code arbeiten

Die **CodeModule**-Klasse bietet uns den Großteil der Möglichkeiten, um mit dem Code zu arbeiten – einige weitere finden wir noch im **CodePane**-Element.

Wir schauen uns nun zunächst die Möglichkeiten von **CodeModule** an und kommen dann auf **CodePane** zu sprechen.

- **AddFromFile**: Fügt den Inhalt der als Parameter angegebenen Textdatei hinten an den bestehenden Code des Moduls an.
- **AddFromString**: Fügt den als Parameter angegebenen Inhalt hinten an den bestehenden Code des Moduls an.
- **CodePane**: Verweist auf das **CodePane**-Objekt, das zu diesem **CodeModule**-Element gehört.

twinBASIC: COM-Add-In für den VBA-Editor

Wer den VBA-Editor mit eigenen Tools erweitern möchte, kommt um die Programmierung von COM-Add-Ins kaum herum – zumindest nicht, wenn er eine schicke Benutzeroberfläche und die Integration in Menüs, Symbolleisten und Kontextmenüs wünscht. In diesem Artikel zeigen wir daher, wie wir die Basis für ein solches COM-Add-In mit twinBASIC programmieren. Ausgehend davon kannst Du direkt loslegen und Deine gewünschten Funktionen einbauen – es sind nur jeweils wenige Anpassung notwendig. Wir erklären Schritt für Schritt, wie die Basis des COM-Add-Ins aufgebaut ist und welche Anpassungen Du vornehmen musst, um ein COM-Add-In für Deine eigenen Anwendungen zu bauen.

Start mit Vorlage

Für den Start kannst Du das Beispielprojekt **VBECOMAddInBasis.twinproj** aus dem Download zu diesem Artikel verwenden. Diese Datei öffnest Du in der Entwicklungsumgebung twinBASIC, deren jeweils aktuelle Fassung Du unter dem folgenden Link findest:

```
https://github.com/twinbasic/twinbasic/releases
```

Dort klickst Du auf Assets und kannst dann wie in Bild 1 die **.zip**-Datei mit der Entwicklungsumgebung herunterladen. Eine Installation ist nicht notwendig, es reicht das Entpacken im gewünschten Verzeichnis.

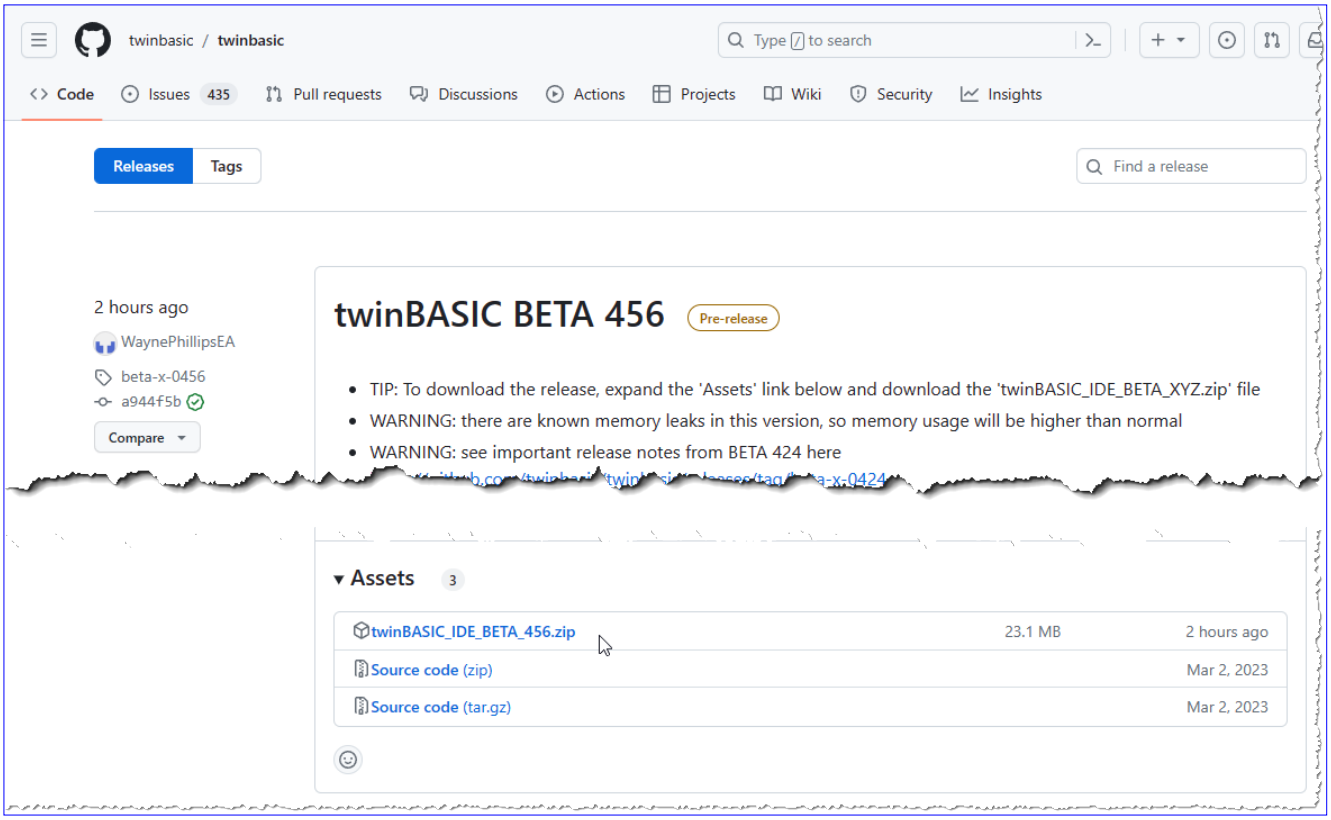


Bild 1: Download der neuesten twinBASIC-Version

Damit kannst Du direkt die Datei **VBECOMAddInBasis.twinproj** öffnen. Diese besteht aus verschiedenen Komponenten, die wir für eine eigene Lösung ändern müssen:

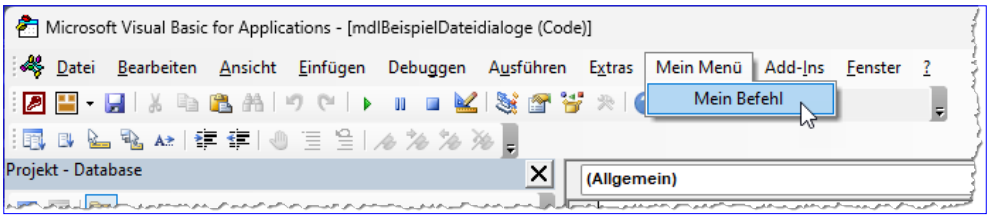


Bild 2: Hier soll der Menübefehl erscheinen.

- Code des COM-Add-Ins im Modul **MeinComAddIn.twin**, das die eigentliche Funktionalität enthält
- Code des Moduls **dllREgistration.twin**, das den Code für die Registrierung des Tools in der Registry enthält
- Eigenschaften im Bereich **Settings**, zum Beispiel um den Projektnamen und Verweise anzupassen

Code der Hauptklasse

Die Hauptklasse enthält bereits alles, was zum Erstellen eines funktionsfähigen COM-Add-Ins für den VBA-Editor enthält.

Im Kopf sehen wir eine **ClassId**, die wir für die Registrierung benötigen. Diese müssen wir jeweils auf eine neue, auf dem jeweiligen System eindeutige GUID anpassen:

```
[ ClassId ("95FA1EA0-DF9F-4505-A22E-E7CFDCD18949") ]
```

Darunter beginnt gleich die Klassendefinition. Den Namen der Klasse passen wir, genau wie den Namen des Moduls, auf den gleichen Wert an. Aktuell heißen die Klasse und die Moduldatei **MeinComAddIn**.

Die Klasse implementiert die Schnittstelle **IDTExtensibility2**:

```
Class MeinComAddIn 'auf eigenes Add-In anpassen
    Implements IDTExtensibility2
```

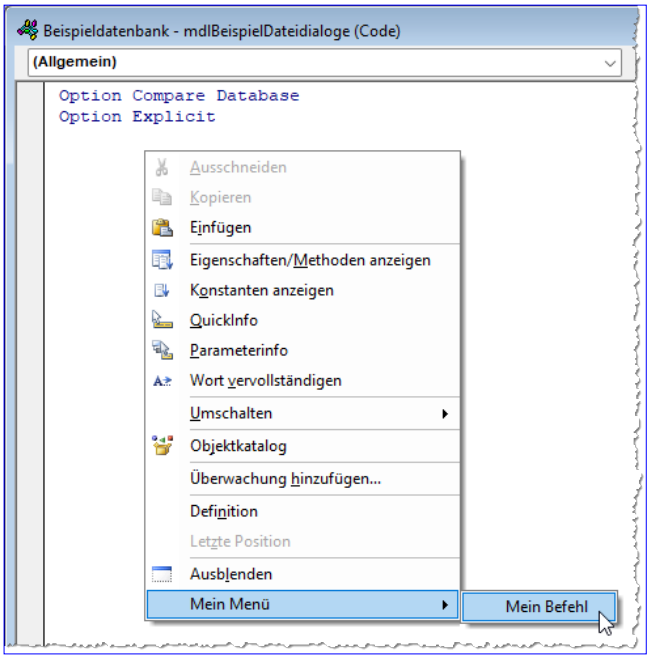


Bild 3: Der Kontextmenübefehl wird hier eingeblendet.

Konstanten für das COM-Add-In

Bevor wir diese erläutern, werfen wir einen Blick auf die benötigten Konstanten und Variablen. Hier definieren wir zunächst zwei Konstanten. Die erste enthält den Namen des Hauptmenüpunktes für Dein COM-Add-In, der zweite den Namen des Befehls, der sich in diesem Menü befindet:

```
Private Const cStrMenu As String = "Mein Menü"
Private Const cStrCommandButton As String = "Mein Befehl"
```

Der folgende Code wird dafür sorgen, dass diese beiden Konstanten beim Anlegen der Menübefehls und des Kontextmenü-Eintrags berücksichtigt werden. Der Menübefehl wird dann wie in Bild 2 auftauchen.

Den Kontextmenü-Eintrag finden wir, wenn wir mit der rechten Maustaste in das Code-Fenster klicken (siehe Bild 3).

Variablen für das COM-Add-In

Für die Basisversion des COM-Add-Ins verwenden wir die folgenden Variablen:

- **objVBE**: Nimmt einen Verweis auf den Visual Basic Editor auf.
- **objAddIn**: Referenziert das Add-In.
- **cbc**: Referenziert das **CommandBarButton**-Objekt für den Button in der Menüleiste.
- **cbcContext**: Referenziert das **CommandBarButton**-Objekt für den Button in der Kontextmenü-Leiste.
- **cbcEvents**: Nimmt die Events für den Button in der Menüleiste auf.
- **cbcEventsContext**: Nimmt die Events für den Button in der Kontextmenü-Leiste auf.

Die Deklaration sieht wie folgt aus:

```
Private objVBE As VBIDE.VBE
Private objAddIn As VBIDE.AddIn
Private cbc As CommandBarButton
Private cbcContext As CommandBarButton
Private WithEvents cbcEvents As VBIDE.CommandBarEvents
Private WithEvents cbcEventsContext As VBIDE.CommandBar-
Events
```

Schließlich benötigen wir noch eine Variable, mit der wir die Information speichern, ob das Add-In geladen ist:

```
Private isConnected As Boolean
```

Beim Laden/Verbinden des COM-Add-Ins

Weiter oben haben wir geschrieben, dass die Klasse **MeinComAddIn** die Schnittstelle **IDTExtensibility2** implementiert.

Das bedeutet, dass wir einerseits bestimmte Ereignisprozeduren implementieren müssen. Andererseits können wir aber auch sichergehen, dass diese Ereignisprozeduren ausgelöst werden, wenn der VBA-Editor für eine der Office-Anwendungen gestartet wird und unser COM-Add-In in der Registry eingetragen ist.

Die erste und wichtigste Prozedur, die dadurch aufgerufen wird, heißt **OnConnection**. Sie sieht wie folgt aus:

```
Sub OnConnection(ByVal Application As Object, _
        ByVal ConnectMode As ext_ConnectMode, _
        ByVal AddInInst As Object, _
        ByRef custom As Variant()) _
        Implements IDTExtensibility2.OnConnection
    Set objVBE = Application
    Set objAddIn = AddInInst
    isConnected = True
    CreateToolBar()
End Sub
```

Die Prozedur hat vor allem eine Aufgabe: Einen Verweis auf die Klasse des VBA-Editors entgegenzunehmen, der mit dem Parameter **Application** geliefert wird, und diesen mit der Variablen **objVBE** zu referenzieren.

Diese Variable ist der Ausgangspunkt für jegliche Aktionen, die wir im VBA-Editor codegesteuert durch unser Add-In durchführen wollen.

Außerdem referenzieren wir noch die Add-In-Instanz selbst, stellen **isConnected** auf **True** ein und rufen die Prozedur **CreateToolBar** auf, die wiederum die Menü- und Kontextmenü-Leisten erstellt.

Schaltflächen zu Menü und Kontextmenü hinzufügen

In der Regel wollen wir durch COM-Add-Ins im VBA-Editor Operationen am Code durchführen. Deshalb gehen wir in dieser Basisversion davon aus, dass wir den gleichen Befehl einmal in der Menüleiste und ein-

```
Private Sub CreateToolBar()
    Dim cbr As CommandBar
    Dim cbp As CommandBarPopup
    Dim cbpContext As CommandBarPopup
    'Eintrag in Menüleiste anlegen
    Set cbr = objVBE.CommandBars("Menüleiste")
    On Error Resume Next
    Set cbp = cbr.Controls(cStrMenu)
    On Error GoTo 0
    If cbp Is Nothing Then
        Set cbp = cbr.Controls.Add(msoControlPopup, Temporary:=True)
        With cbp
            .Move Before:=8
            .Caption = cStrMenu
            Set cbb =.Controls.Add(msoControlButton, Temporary:=True)
        End With
    Else
        Set cbb = cbp.Controls.Add(msoControlButton, temporary:=True)
    End If
    With cbb
        .Caption = cStrCommandButton
        .BeginGroup = True
        Set cbbEvents = objVBE.Events.CommandBarEvents(cbb)
    End With
    'Eintrag in Kontextmenü anlegen
    On Error Resume Next
    Set cbpContext = objVBE.CommandBars("Code Window").Controls(cStrMenu)
```

Dateiauswahl-Dialog-Assistent programmieren

Dateidialoge benötigt man immer wieder. Ob man nun Dateien zum Öffnen oder Bearbeiten auswählen möchte, ob man einen Pfad zum Speichern einer Datei braucht oder ob man ein Verzeichnis selektieren will – am einfachsten geht das mit den praktischen Dateidialogen. Die Office-Bibliothek bietet sogar alle benötigten Varianten über die FileDialog-Klasse an. Dumm ist nur, dass man nicht ständig Filedialoge programmiert, sondern nur alle paar Wochen, Monate oder sogar Jahre. Dann muss man sich immer wieder einarbeiten, um die verschiedenen Parameter – Titel, Schaltflächenbeschriftungen, Standardverzeichnis und -dateiname, Filter, Dateiendungen und so weiter zu definieren. Wie schön wäre es doch, wenn wir solche Dateidialoge mit einem kleinen Assistenten zusammenstellen könnten. Also machen wir uns ans Werk und schaffen einen solchen Wizard!

Wann, wie und wo nutzen wir den Dateidialog-Assistenten?

Die erste Frage, die sich stellt, ist: Wo und wie wollen wir diesen Assistenten aufrufen und was genau soll dieser eigentlich produzieren? Als Erstes fällt uns ein, dass er eine Funktion liefern soll, die den Dateidialog aufruft und uns die gewünschte Datei oder das Verzeichnis zurückliefert.

Wir könnten uns also darauf beschränken, diesen Assistenten für das Zusammenstellen des benötigten Codes zu nutzen und diesen dann beispielsweise an der aktuellen Stelle im Codefenster des VBA-Editors einzufügen oder diesen in die Zwischenablage zu schreiben, damit wir diesen selbst einfügen können.

In vielen Fällen mag das ausreichen: Dann ruft man den Dateidialog aus einer anderen Prozedur heraus auf und verwendet die gelieferten Informationen in den folgenden Codezeilen für den gewünschten Zweck.

In anderen Fällen spielt die Benutzeroberfläche eine Rolle: Dann möchte man vielleicht in einem Access-Formular ein Verzeichnis etwa zum Exportieren von Daten anzeigen und auswählen. Dazu fügt man diesem ein Textfeld oder ein Bezeichnungsfeld zur Anzeige des Pfades hinzu und legt daneben eine Schaltfläche an, mit der man den Dateiauswahl-Dialog aufruft.

Letztlich würde auch hier erst einmal der Assistent zum Zusammenstellen von Funktionen zum Anzeigen von Dateidialogen ausreichen.

Man könnte die benötigten Steuerelemente und die Programmlogik zum Aufruf der Funktionen dann selbst hinzufügen – immerhin ist das der wesentlich weniger aufwendige Teil dieser Aufgabe.

Wir werden uns also zunächst darauf beschränken, den Code für die Anzeige von Dateidialogen abhängig von den gewünschten Parametern zu produzieren.

Wo wollen wir diese Funktionalität bereitstellen? Sinnvoll ist ein COM-Add-In für den VBA-Editor, denn dort werden wir den zu erstellenden Code auch nutzen.

Wo dort wollen wir die Funktion aufrufen? Hier gibt es drei mögliche Stellen:

- einen Menüeintrag, beispielsweise unterhalb des Eintrags **Add-Ins** oder in einem eigenen Hauptmenü

- ein Eintrag in der Symbolleiste
- ein Eintrag im Kontextmenü, das erscheint, wenn wir mit der rechten Maustaste in das Codefenster klicken

Werkzeug für das Erstellen des COM-Add-Ins

Die Entwicklungsumgebung und Programmiersprache twinBASIC hat sich als zuverlässiges Tool für die Programmierung von COM-Add-Ins und COM-DLLs für Office-Anwendungen und den VBA-Editor erwiesen, sodass wir diesen auch in diesem Artikel für die Erstellung des gewünschten Assistenten nutzen werden.

COM-Add-In bis zur eigentlichen Funktion

Im Artikel **twinBASIC: COM-Add-In für den VBA-Editor** (**www.access-im-unternehmen.de/421**) stellen wir eine Vorlage für die Programmierung von COM-Add-Ins für den VBA-Editor vor – und erläutern, welche Schritte nötig sind, um diese Vorlage in ein eigenes, neues COM-Add-In umzuwandeln.

Genau das haben wir gemacht und so ein neues, fast leeres COM-Add-In erzeugt. Die einzigen Elemente, die dieses COM-Add-In enthält, dienen zur Anzeige von Menüeinträgen in einem neuen Menüpunkt im Hauptmenü des VBA-Editors sowie eines Eintrags eines Kontextmenüs, das erscheint, wenn wir mit der rechten Maustaste in das Codefenster klicken.

Informationen zum Erstellen des Codes ermitteln

Bevor wir starten, stellen wir eine Liste der Informationen zusammen, die wir vor dem Erstellen des Codes für die Anzeige eines Dateiauswahl-Dialogs vom Entwickler ermitteln müssen.

Was benötigen wir also alles?

- Um was für einen Dateidialog soll es sich handeln? Es gibt Dialoge für die Auswahl von bestehenden Dateien, für zu erstellende Dateien und für Verzeichnisse. Dies können wir mit einer Optionsgruppe oder einem Kombinationsfeld abfragen. Außerdem gibt es noch den Typ zum direkten Öffnen von Dateien, der in Word oder Excel zum Einsatz kommen kann.
- Einfache oder Mehrfachauswahl. Soll der Benutzer keine, eine oder sogar mehrere Dateien auswählen können? Wenn er mehrere Dateien auswählen können soll, müssen wir uns überlegen, wie wir diese Dateien übergeben.
- Beschriftung des Dialogtitels
- Beschriftung der Schaltfläche zum Auswählen
- Ordner, der beim Öffnen des Dialogs angezeigt werden soll, einstellen
- Datei, die beim Öffnen ausgewählt oder angegeben werden soll
- Leeren vorhandener Filter, Erstellen von Filtern und Auswahl eines der Filter als Standard

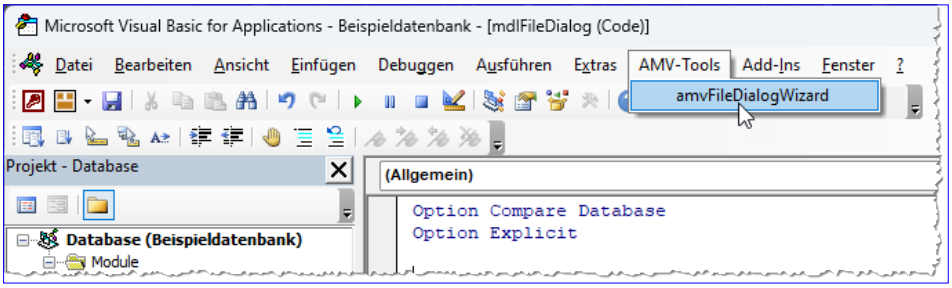


Bild 1: Aufruf per Menüeintrag

Auf all diese Elemente gehen wir in den folgenden Abschnitten ein.

Der FileDialog-Wizard in Aktion

Den Wizard wollen wir auf zwei Arten aufrufen können. Die erste ist der Menüeintrag **AMV-Tools|amvFileDialogWizard** (siehe Bild 1).

Die zweite und bevorzugte Möglichkeit ist der Aufruf über das Kontextmenü des Codefensters (siehe Bild 2).

In Bild 3 zeigen wir, wie der fertige Wizard aussehen soll. Hier sehen wir Eingabefelder für den Titel, den Funktionsnamen, den Buttontext und den beim Öffnen anzuzeigenden Pfad. Für Dialogtyp haben wir ein Kombinationsfeld hinterlegt.

Ob der Dialog die Mehrfachauswahl unterstützen soll, legen wir mit einer CheckBox fest. Schließlich kann der Benutzer die Beschreibung und die Dateiendung für Filter über zwei Textfelder eingeben, die mit der Plus-Schaltfläche zur Liste der anzuzeigenden Filter hinzugefügt werden. Mit der Minus-Schaltfläche kann der Benutzer den markierten Filtereintrag wieder entfernen.

Jede Änderung, die der Benutzer an einem der Steuerelemente vornimmt, wirkt sich direkt auf den angezeigten Code aus. Dadurch sieht der Benutzer jeder-

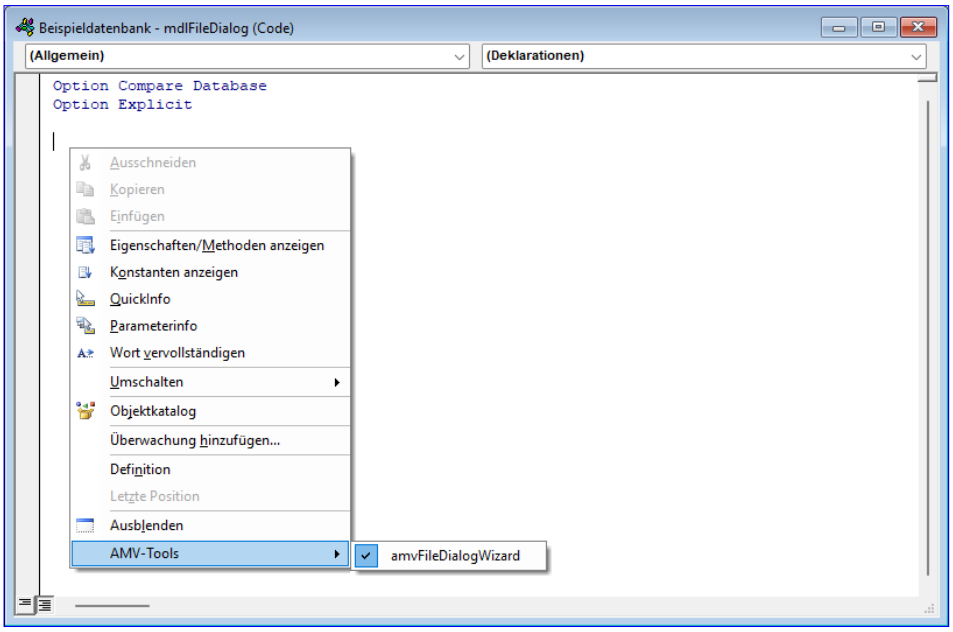


Bild 2: Aufruf per Kontextmenü

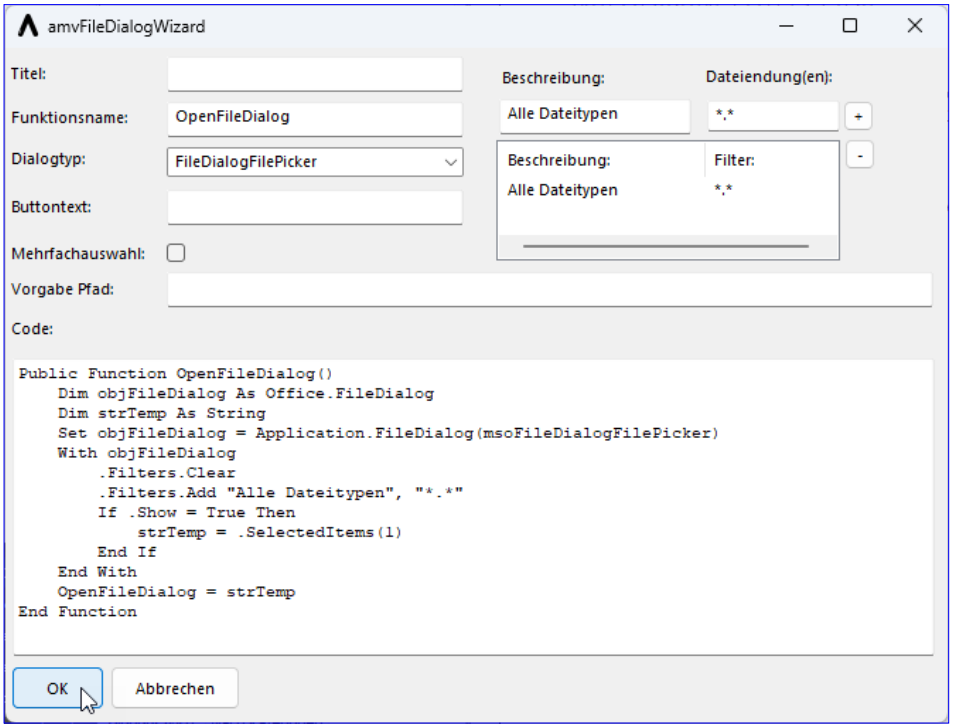


Bild 3: Der FileDialog-Wizard in Aktion

zeit, welcher Code eingefügt wird, wenn er auf die **OK**-Schaltfläche klickt.

Programmierung des COM-Add-Ins zur Bereitstellung des FileDialog-Wizards

Diesen Wizard programmieren wir in den folgenden Abschnitten.

Neues Projekt erstellen

Als Erstes erstellen wir ein neues Projekt auf Basis der Vorlage, die wir im Artikel **twinBASIC: COM-Add-In für den VBA-Editor** (**www.vbentwickler.de/421**) beschrieben haben. In diesem Artikel haben wir auch gezeigt, an welchen Stellen Änderungen vorgenommen werden müssen, wenn wir ein neues COM-Add-In auf der Basis der Vorlage erstellen wollen. Der erste Schritt ist, dass wir die Hauptklasse in **FileDialogWizard** umbenennen und das auch für einige weitere Elemente im Projekt erledigen.

Viele der Elemente können wir zunächst beibehalten. Dazu gehört auch die Prozedur **CreateToolbars**. Hier haben wir lediglich die beiden Konstanten geändert, welche für die Anzeige des Menü- und des Kontextmenüeintrags verantwortlich sind. Diese lauten nun wie folgt:

```
Private Const cStrMenu As String = "AMV-Tools"
Private Const cStrCommandButton As String = "amvFileDialogWizard"
```

Für die beiden Ereignisprozeduren, die durch die beiden Schaltflächen im Menü und im Kontextmenü ausgelöst werden, haben wir jeweils den Aufruf der Prozedur **LaunchFileDialogWizard** eingetragen:

```
Private Sub cbbEvents_Click(...) Handles _
        cbbEvents.Click
    LaunchFileDialogWizard
    ...
End Sub

Private Sub cbbEventsContext_Click(...) _
        Handles cbbEventsContext.Click
    LaunchFileDialogWizard
    ...
End Sub
```

Damit sind die Arbeiten am Modul **FileDialogWizard** bereits erledigt.

Die Methode LaunchFileDialogWizard

Die durch die Menübefehle aufgerufene Methode **LaunchFileDialogWizard** finden wir in Listing 1. Sie ist dafür verantwortlich, das Fenster unseres Assisten-

```
Public Sub LaunchFileDialogWizard()
    Dim frm As frmFileDialogWizard
    Dim strCode As String
    Dim lngStartLine As Long
    Dim lngStartColumn As Long
    Dim lngEndLine As Long
    Dim lngEndColumn As Long
    Dim lngCurrentLine As Long
    Dim objCodepane As CodePane
    Dim objCodemodule As CodeModule
    Dim lngHwnd As Long
    Set frm = New frmFileDialogWizard
    lngHwnd = frm.hWnd
    frm.Show vbModal
    If IsWindow(lngHwnd) = 0 Then
        Exit Sub
    Else
        strCode = frm.txtCode
        UnloadObjectByHandle(lngHwnd)
        Set objCodepane = objVBE.ActiveCodePane
        Set objCodemodule = objCodepane.CodeModule
        objCodepane.GetSelection(lngStartLine, lngStartColumn, _
            lngEndLine, lngEndColumn)
        objCodemodule.InsertLines(lngStartLine, strCode)
    End If
End Sub
```

Listing 1: Die Prozedur **LaunchFileDialogWizard**

ten anzuzeigen und den Code anzuhalten, bis dieses Fenster entweder geschlossen oder ausgeblendet wird. Das Fenster initialisieren wir auf Basis des **Form**-Objekts **frmFileDialogWizard**, das wir weiter unten erläutern. An dieser Stelle ist erst einmal wichtig, wie wir es aufrufen, wie wir reagieren, wenn es den Fokus verliert und was dann geschieht. Bevor wir es anzeigen, speichern wir sein Handle in der Variablen **lngHwnd**. Dann öffnen wir es mit der **Show**-Methode und übergeben dieser den Wert **vbModal** als Parameter, damit der aufrufende Code angehalten wird, bis der Benutzer die Arbeit mit dem Fenster abschließt.

Dann prüfen wir mit der Funktion **IsWindow**, ob das Fenster noch geöffnet ist oder nicht. Im ersten Fall hat der Benutzer das Fenster mit der **Abbrechen**-Schaltfläche oder der **Schließen**-Schaltfläche geschlossen. Dann gibt es nichts weiter zu tun, als die Prozedur zu beenden.

Anderenfalls lesen wir den im Textfeld **txtCode** enthaltenen Code in die Variable **strCode** ein und schließen das Formular mit der Funktion **UnloadObjectByHandle**, wobei wir das Handle des Fensters übergeben.

Danach referenzieren wir das aktuelle **Codepane**- und dann das **CodeModule**-Objekt. Über die Methode

Klasse zum Erstellen des FileDialog-Codes

Die Klasse **clsFileDialogWizard** ist das Arbeitstier des COM-Add-Ins. Die Klasse nimmt alle notwendigen Informationen entgegen und erstellt den Code der Funktion zum Anzeigen des Dateidialogs. Dazu deklarieren wir in der Klasse zunächst die folgenden lokalen Variablen:

```
Private m_Title As String
Private m_Type As MsoFileDialogType
Private m_FunctionName As String
Private m_AllowMultiSelect As Boolean
Private m_Separator As String
Private m_ButtonName As String
Private m_InitialFileName As String
Private m_Filter As String
```

Diese füllen wir über entsprechende **Property Let**-Prozeduren, die nach außen als Eigenschaften präsentiert werden. Für die meisten Eigenschaften ist das recht einfach – sie nehmen einfach den Wert entgegen und speichern diesen in der jeweiligen Variablen:

```
Public Property Let Title(str As String)
    m_Title = str
End Property
```