AUSGABE 03/2023

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

Excel: Bilder in Worksheets einfügen

Excel ist ein leistungsstarkes und vielseitiges Tool für die Datenanalyse und -visualisierung. Es ermöglicht es den Benutzern, große Datenmengen effektiv zu verwalten und in verschiedenen Formen darzustellen. Bilder sind dabei oft eine wichtige Ergänzung, um die Analyse von Daten oder Informationen zu unterstützen. Das Einfügen von Bilddateien in Excel-Worksheets per VBA ist eine solche Aufgabe, die es Benutzern ermöglicht, Bilder in ihre Arbeitsmappen einzufügen, um die Datenanalyse und -präsentation zu verbessern. In diesem Artikel werden wir uns damit beschäftigen, wie man Bilddateien in Excel-Worksheets per VBA einfügen kann und welche Vorteile und Anwendungsfälle es gibt.

Wozu Bilder in Excel einfügen?

Für das Einfügen von Bildern in Excel-Dokumenten gibt es verschiedene Anwendungszwecke. Hier sind ein paar davon:

- Einfügen von Firmenlogos oder Produktbildern in Rechnungen oder Angebote, um sie ansprechender zu gestalten.
- Hinzufügen von Fotos von Mitarbeitern oder Teams in Organigrammen, um sie persönlicher und einladender zu gestalten.
- Einfügen von Diagrammen oder Grafiken, die auf Daten basieren, um die Visualisierung von Ergebnissen oder Trends zu verbessern.
- Hinzufügen von Screenshots oder Bildern von Webseiten, um die Analyse von Daten oder Informationen zu unterstützen.
- Einfügen von Bildern, um Anleitungen oder Arbeitsanweisungen in Excel-Tabellen zu ergänzen, um den Prozess oder das Ergebnis besser zu erklären.

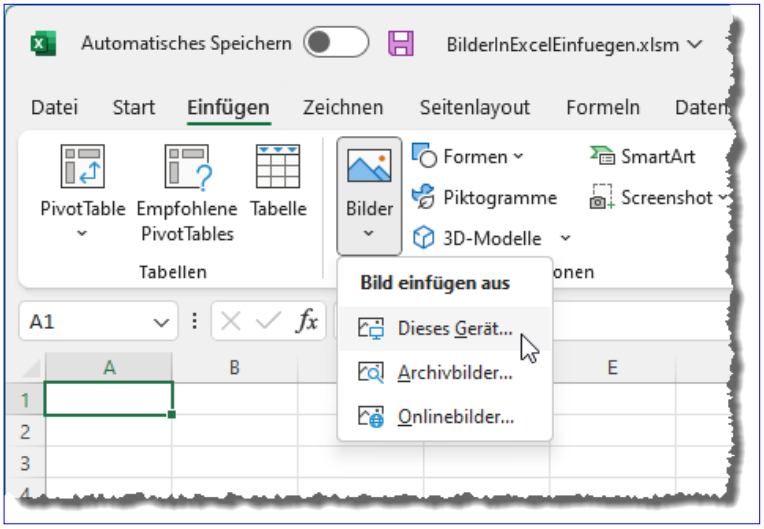

Bild 1: Bild einfügen per Benutzeroberfläche

Bild einfügen per Benutzeroberfläche

Wenn wir ein Bild in eine Zelle einfügen wollen, gelingt dies am einfachsten über die Benutzeroberfläche. Dazu markieren wir die Zielzelle und wähle aus dem Ribbon den Eintrag **Einfügen|Illustrationen|Bilder** aus und dann die Schaltfläche **Dieses Gerät... (siehe Bild 1).**

Damit aktivieren wir einen **Datei öffnen**-Dialog, mit dem wir die einzufügende Bilddatei auswählen.

Diese erscheint anschließend in der aktuell markierten Zelle (siehe Bild 2).

Klicken wir mit der rechten Maustaste auf das neu hinzugefügte Bild, finden wir im Kontextmenü beispiels-

weise die Befehle **Größe und Eigenschaften** oder **Grafik formatieren...** – mit Ersterem öffnen wir den Bereich aus Bild 3.

Damit sehen wir bereits die Optionen, die wir gegebenenfalls auch mit VBA einstellen können.

Verschiedene VBA-Methoden

Nachfolgend stellen wir zwei verschiedene Möglichkeiten vor, Bilder per VBA zu einem Excel-Worksheet hinzuzufügen:

- die **Insert**-Methode der **Pictures**-Auflistung und
- die **AddPicture**-Methode der **Shapes**-Auflistung.

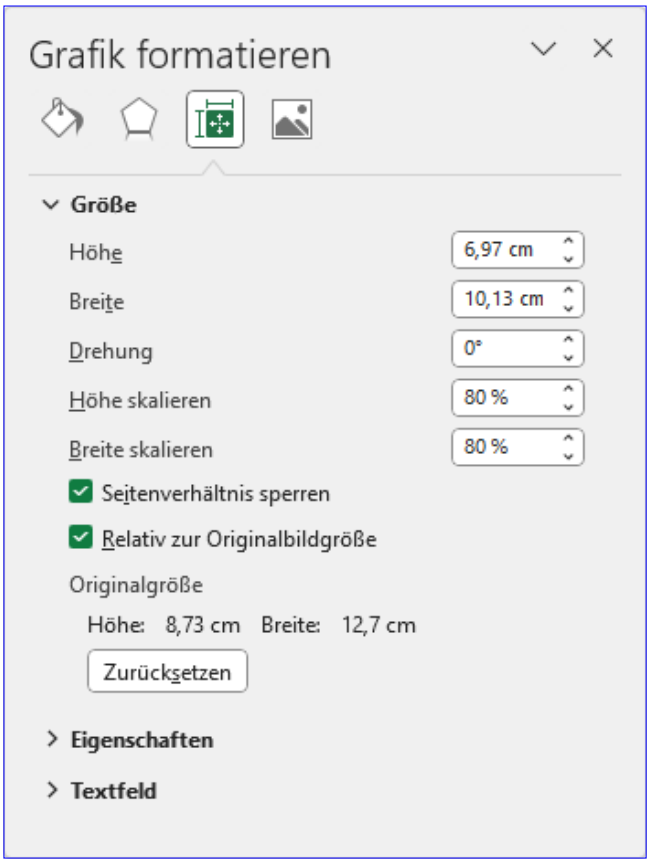

Bild 3: Einige Eigenschaften des Bildes

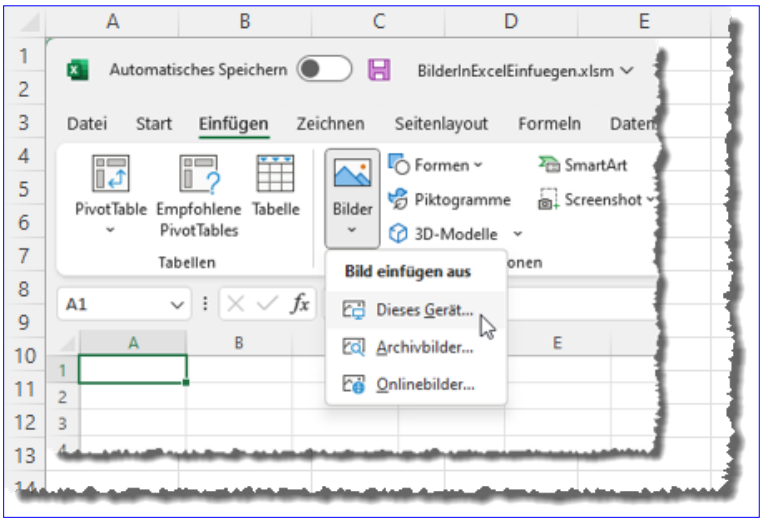

Bild 2: Ein eingefügtes Bild

Beide haben Vor- und Nachteile, die wir uns im Folgenden ansehen. Wir beginnen mit der **Insert**-Methode der **Pictures**-Auflistung.

Bilder einfügen per Pictures.Insert

Zum Einfügen von Bildern per VBA bietet das Excel-Objektmodell die Methode **Insert** der **Pictures**-Auflistung an. Diese erwartet lediglich den Pfad der einzufügenden Bilddatei als Parameter.

In folgendem Beispiel wollen wir eine Bilddatei einfügen, die sich im gleichen Verzeichnis wie das Excel-Workbook befindet, welches wir mit **ThisWorkbook.Path** ermitteln:

```
ActiveSheet.Pictures.Insert _
    ThisWorkbook.Path & "\pic001.png"
```

Aber wo wird diese Bilddatei nun eingefügt? An der Stelle der aktuell markierten Zelle. Wir müssen also, wenn wir eine Bilddatei gezielt in einem Excel-Worksheet ablegen wollen, zuvor die Zielzelle markieren.

Handelt es sich um die Zelle **B2**, gelingt das wie folgt. Wir referenzieren das aktuelle Worksheet und hier die Zelle **B2**, dann fügen wir das Bild ein:

```
Public Sub BilddateiEinfuegen()
    Dim wsh As Worksheet
    Set wsh = ActiveSheet
    wsh.Range("B2").Select
    wsh.Pictures.Insert ThisWorkbook.Path & "\pic001.png"
End Sub
```

Damit landet das Bild in Zelle **B2** (siehe Bild 4).

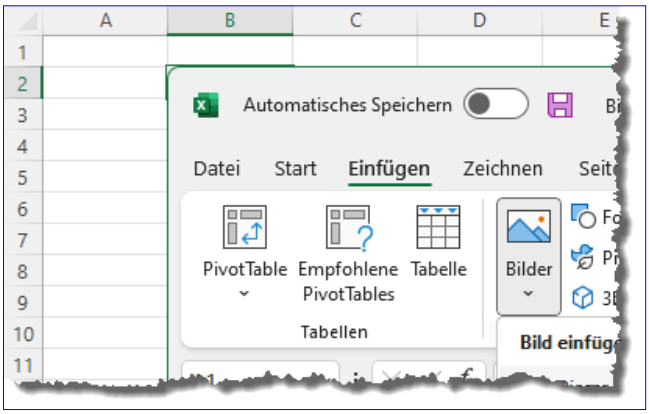

Bild 4: Bild in Zelle **B2** einfügen

Bild auswählen und in aktueller Zelle einfügen

Wir können das Bild auch zuvor per Dateiauswahl-Dialog auswählen und einfach an der aktuell markierten Zelle einfügen. Dazu verwenden wir zunächst die Methode **GetOpenFilename** und weisen das Ergebnis der **Insert**-Methode zu:

```
Public Sub BilddateiAuswaehlen()
    Dim strBild As String
    strBild = Application.GetOpenFilename(, , _
        "Bild auswählen")
    ActiveSheet.Pictures.Insert strBild
End Sub
```

Danach können wir die einzufügende Datei auswählen und diese landet mit der linken oberen Ecke in der gewünschten Zelle.

Bilder durchlaufen

In einer **For Each**-Schleife können wir die in einem Excel-Worksheet enthaltenen Bilder durchlaufen.

Im folgenden Beispiel geben wir den Namen des jeweils referenzierten Bildes aus:

```
Public Sub BilderDurchlaufen()
    Dim objPicture As Picture
    For Each objPicture In ActiveSheet.Pictures
        Debug.Print objPicture.Name
    Next objPicture
End Sub
```

Diese werden von Excel beim Einfügen automatisch vergeben und lauten zum Beispiel **Picture 8** oder **Picture 12**.

Eigenschaften von Bildern in Excel

Weiter oben haben wir bereits den Eigenschaften-Dialog für ein Bild in einem Excel-Worksheet angesehen. Die dort angezeigten Eigenschaften lassen sich auch per VBA einstellen.

Dazu referenzieren wir zuerst das gewünschte Bild, in diesem Fall das einzige vorhandene Bild (oder das erste, falls mehrere enthalten sind):

```
Dim objPicture As Picture
Set objPicture = ActiveSheet.Pictures(1)
```

Referenzieren eines Picture-Elements

Neben dem Index können wir die Bilder in der **Pictures**-Auflistung auch über einen beim Einfügen automatisch vergebenen Namen referenzieren, zum Beispiel:

```
Set objPicture = ActiveSheet.Pictures("Picture 12")
```

Umbenennen von Picture-Elementen

Und wir können die Bilder auch umbenennen, indem wir der **Name**-Eigenschaft einen neuen Wert hinzufügen:

Word-Vorlage für Anschreiben erstellen

Manchmal gibt es Situationen, da benötigt man das gute, alte Anschreiben. Adresse eintragen, Betreff hinzufügen, Ort und Datum festlegen und schließlich noch den Inhalt schreiben. Wenn man das nur hin und wieder machen muss, wie es bei mir der Fall ist, kann es schon mal sein, dass man nicht mehr weiß, wo man die Vorlage gespeichert hat, die man beim vorherigen Mal genutzt hat. Oder wo überhaupt ein Word-Dokument ist, das man bereits erstellt hat und dass man kopieren, anpassen und ausdrucken kann. Auf diese Weise braucht man nicht bei Adam und Eva anzufangen und den Briefkopf neu zu erstellen und die gute alte DIN-Norm herauszusuchen, die Informationen über die Positionen der einzelnen Elemente enthält. In diesem Artikel schauen wir uns an, wie wir eine passende Vorlage erstellen; in einem weiteren, wie wir diese aus einem Formular heraus mit den gewünschten Informationen füllen.

In diesem Artikel erledigen wir die folgenden Aufgaben:

- Erstellen einer Word-Vorlage, welche die wichtigsten Informationen wie Absenderadresse, Feld für die Empfängeradresse, Betreffzeile und Inhalt enthält
- Erstellen eines Formulars, das automatisch beim Erstellen eines neuen Dokuments auf Basis dieser Vorlage erscheint und die Informationen abfragt, die in das Dokument eingetragen werden sollen.

Warum sollten wir ein Formular bemühen und nicht direkt die notwendigen Texte in das Dokument eintragen? Weil wir eine Dokumentvorlage erstellen wollen, die auch von ungeübten Benutzern einfach eingesetzt werden kann.

Und da ist es erheblich einfacher, die Texte in ein Formular einzutragen als in die einzelnen Bereiche des Word-Dokuments, die leicht verschoben oder gelöscht werden können.

Und ganz ehrlich: Word ist wegen seiner Komplexität und der vielen Funktionen nicht gerade dafür ausgelegt, das Einsteiger damit schnell einfache Dokumente erstellen. Da passiert es schnell, dass jemand lieber schnell einen Brief mit Excel schreibt, weil er da zumindest die Position bestimmter Elemente über die Spaltenbreiten und -höhen definieren kann.

Briefelemente erstellen

Ein Brief soll verschiedene Elemente enthalten. Dazu gehören:

- Briefkopf mit Absenderadresse und weiteren Informationen sowie gegebenenfalls einem Logo
- Adressfeld mit der Empfängeradresse und, falls nicht auf dem Umschlag abgedruckt, mit der Absenderadresse als Einzeiler über der Empfängeradresse
- Zeile mit dem Betreff
- Eigentlicher Inhalt des Briefs
- Gegebenenfalls eine Fußzeile für geschäftliche Informationen wie Bankverbindung, Steuernummern et cetera

Wir wollen alle Elemente mit Ausnahme des Inhalts über ein Formular erfassen. Den Inhalt selbst kann der Benutzer dann direkt im Dokument schreiben.

Das ist sinnvoller als das vorherige Eintippen des Textes im Formular, denn dort können wir nicht die vielen Formatierungsmöglichkeiten nutzen, die Word bietet.

Das fertige Dokument soll nach dem Erstellen wie in Bild 1 aussehen.

Hier ist bereits zu erkennen, dass wir hier nicht mit den üblichen Textfeldern arbeiten, sondern Tabellen verwenden.

Bild 1: So soll der fertige Brief aussehen.

Word-Vorlage erstellen

Das Erstellen einer Word-Vorlage geht immer über den Umweg des Anlegenes eines einfachen neuen Word-Dokuments. Erst nach dem Anlegen können wir das Dokument schließlich als Word-Vorlage speichern.

Wir erstellen also ein einfaches Word-Dokument und öffnen mit dann, bevor wir Änderungen vornehmen, mit **Datei|Speichern unter** und einem Klick auf den Befehl **Durchsuchen** (siehe Bild 2) den **Speichern unter**-Dialog.

Hier wählen wir unter Dateityp den Eintrag **Word Vorlage mit Makros (*.dotm)** aus, denn wir wollen eine Vorlage anlegen, die auch VBA-Code ausführen kann. Der **Speichern unter**-Dialog wählt dann selbstständig den Ordner für benutzerdefinierte Vorlagen aus, der beispielsweise hier zu finden ist:

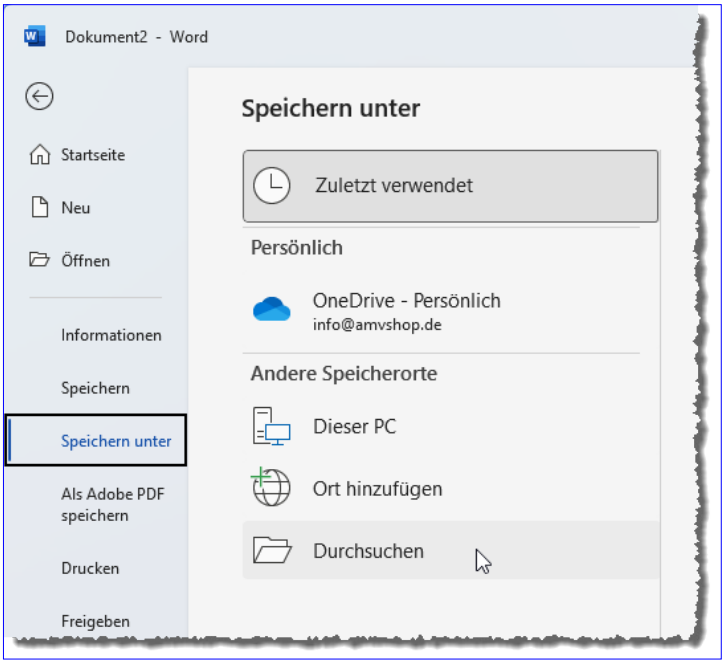

Bild 2: Speichern des Dokuments als Vorlage

```
C:\Users\[Benutzername]\Documents\Benutzerdefinierte Office-Vorlagen
```

Wir verwenden als Dateiname **Briefvorlage.dotm** (siehe Bild 3). Damit haben wir bereits die Vorlage angelegt.

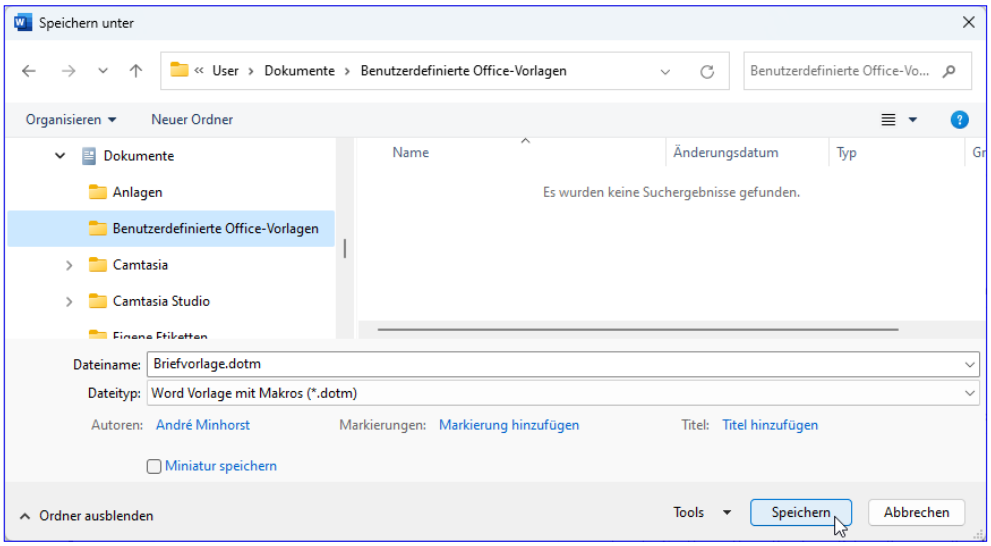

Bild 3: Auswählen des Dateinamens und des Datentyps

Dokument auf Basis der Vorlage erstellen

Wenn wir die Vorlage nun speichern und schließen und in einer neuen Word-Instanz im Begrüßungsbildschirm auf **Weitere Vorlagen** klicken, können wir in den Bereich **Persönlich** wechseln und finden dort bereits unsere neu erstellte Vorlage vor (siehe Bild 4).

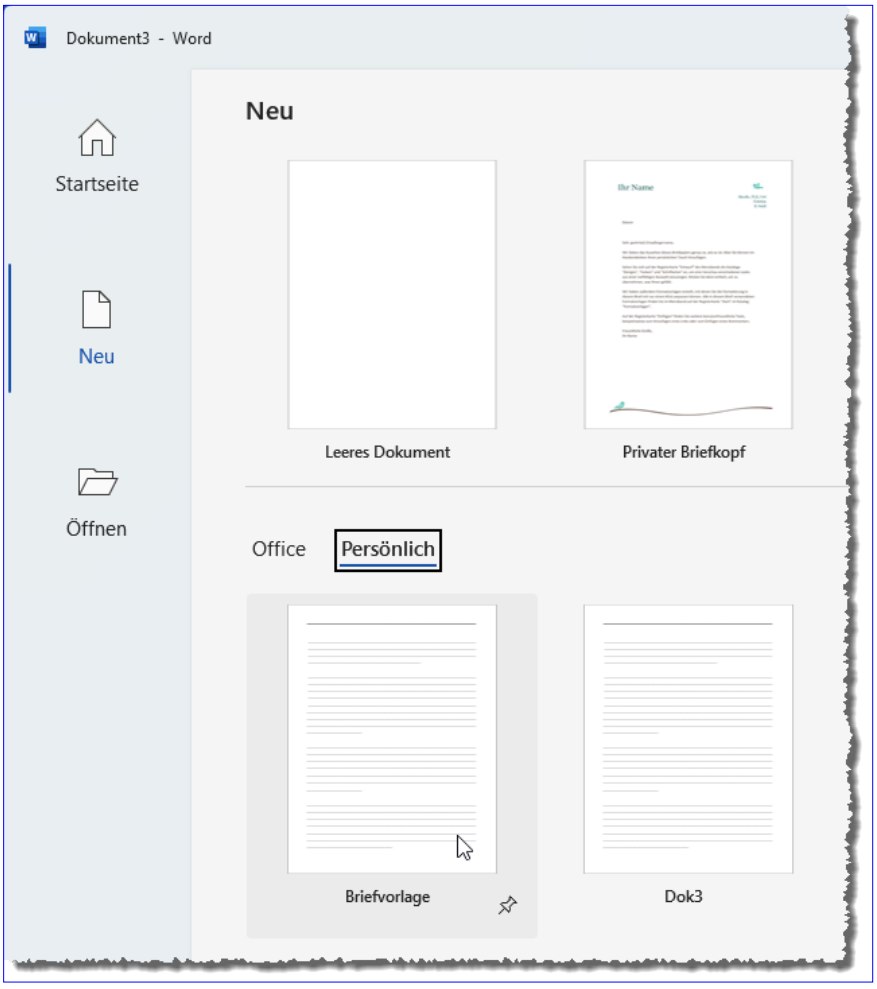

Bild 4: Die Vorlage ist nun bereits in Word verfügbar.

Klicken wir diese doppelt an, wird allerdings ein neues Dokument auf Basis dieser Vorlage erstellt.

Da wir jedoch weiter an der Vorlage arbeiten wollen, müssen wir diese anders öffnen. Auch ein Doppelklick auf die gespeicherte **.dotm**-Datei erstellt nämlich ein neues Dokument auf Basis dieser Vorlage.

Vorlage zum Bearbeiten öffnen

Um die Vorlage selbst zu öffnen, musst Du dies von Word aus erledigen. Dort wählst Du den Befehl **Datei|Öffnen**. Gegebenenfalls wird die Vorlage noch in der Liste der

zuletzt verwendeten Dateien angezeigt, dann kannst Du sie einfach anklicken.

Wenn nicht, klickst Du auf **Durchsuchen** und wählst die **.dotm**-Datei mit dem **Öffnen**-Dialog aus.

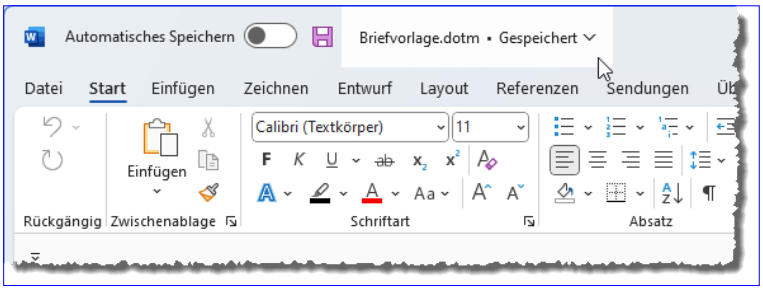

Bild 5: Geöffnete Dokumentvorlage

Woher weiß ich, ob ich die Vorlage oder ein darauf basierendes Dokument geöffnet habe? Das erkennen wir am Dokumentnamen, der in der Titelleiste von Word angezeigt wird (siehe Bild 5). Ist das der Fall, können wir uns an die eigentliche Arbeit begeben.

Seitenränder einstellen

Bevor wir die ersten Elemente zur Vorlage hinzufügen, wollen wir noch die Seitenränder einstellen. Dazu wählen wir im Ribbon den Befehl **Layout|Seite einrichten|Seitenränder** aus (siehe Bild 6).

Ein Klick auf **Benutzerdefinierte Seitenränder** öffnet den Dialog **Seite einrichten** (siehe Bild 7). Hier legen wir die Seitenränder auf die gängigen Werte fest:

- Oben: 1 cm
- Unten: 2,5 cm
- Links: 2,5 cm
- Rechts 1,5 cm

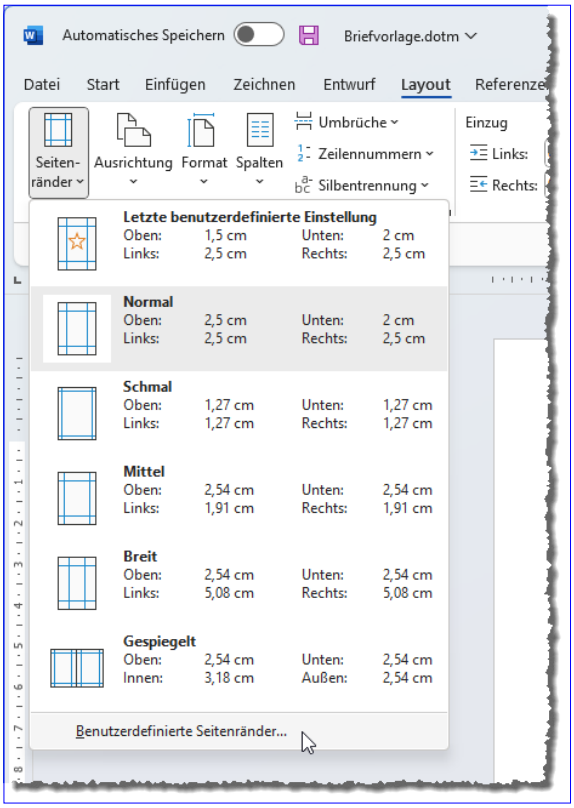

Bild 6: Aufruf der Optionen für die Seitenränder

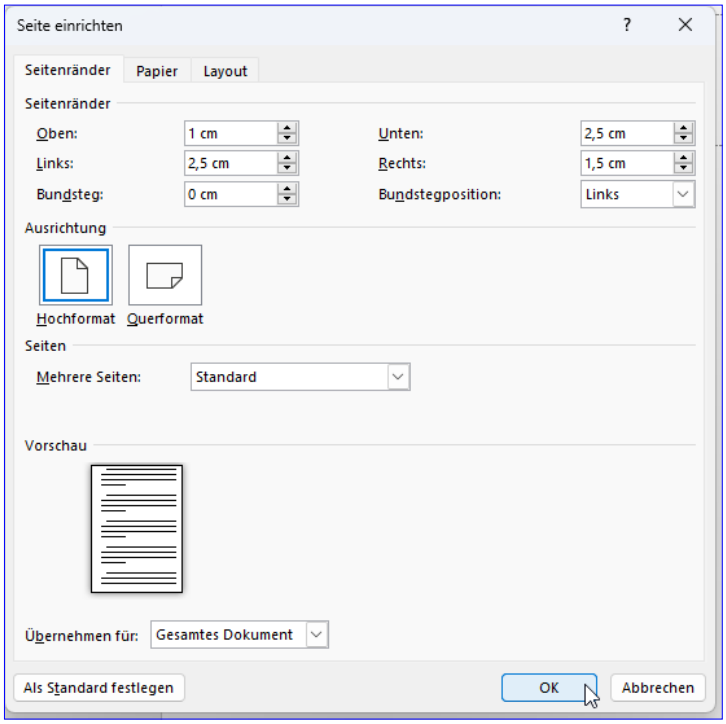

Bild 7: Einstellen der Seitenränder

Kontextmenüs per VBA programmieren

Vor sehr langer Zeit (2003) gab es für die Office-Anwendungen noch die Möglichkeit, Menüleisten, Symbolleisten und Kontextmenüs über die Benutzeroberfläche anzupassen. Die Menüleisten und Symbolleisten sind seit der Version 2007 Geschichte, und damit ist auch der Editor für die Gestaltung dieser Elemente verschwunden. Allerdings gibt es noch die Kontextmenüs, und diese lassen sich auch noch anpassen – zum Beispiel, um eigene Befehle hinzuzufügen. Diese könnten beispielsweise Teil von Add-Ins, COM-Add-Ins oder auch von Dokumenten sein. In diesem Artikel zeigen wir, wie man vorhandene Kontextmenüs bearbeitet oder eigene Kontextmenüs anlegt und diese bei Bedarf aufruft.

Unterschiede zwischen Outlook und den übrigen Office-Anwendungen

Zwischen Outlook und den übrigen Office-Anwendungen gibt es einen entscheidenden Unterschied für die individuelle Anpassungen von Kontextmenüs: Während wir unter Word, Excel, Access oder PowerPoint das Objektmodell aus der Bibliothek **Microsoft Office x.0 Object Library** nutzen können, um die gewünschten Anpassungen vorzunehmen, ist dies in Outlook nicht möglich. Hier sind die Kontextmenüs bereits Bestandteil des Ribbons. Änderungen der Kontextmenüs müssen wir in Outlook daher über die Ribbondefinition vornehmen, und zwar über die Einträge im Unterelement **contextMenus**. Dieses Thema behandeln wir im vorliegenden Artikel nicht, aber wir gehen im Detail darauf in einem weiteren Artikel namens **Kontextmenüs in Outlook anpassen** (**www.vbentwickler.de/369**) ein.

Beispieldateien

Da die CommandBars sich in den verschiedenen Office-Anwendungen Access, Excel und Word leicht unterschiedlich verhalten, haben wir jeweils eine Beispieldatei für jede Anwendung bereitgestellt:

- Access: **KontextmenuesPerVBA.accdb**
- Excel: **KontextmenuesPerVBA.xlsm**
- Word: **KontextmenuesPerVBA.docm**

Objektmodell verfügbar machen

Um die Kontextmenüs anzupassen, benötigen wir Elemente der Bibliothek **Microsoft Office x.0 Object Library**. Diese ist in beispielsweise in den VBA-Projekten von Word und Excel bereits referenziert, nicht jedoch unter Access oder in anderen Projekten auf Basis von Visual Studio oder twinBASIC. Damit wir die Bibliothek etwa in Access unter Einsatz von IntelliSense nutzen können, fügen wir einen Verweis auf diese Bibliothek hinzu. Dazu öffnen wir zunächst den

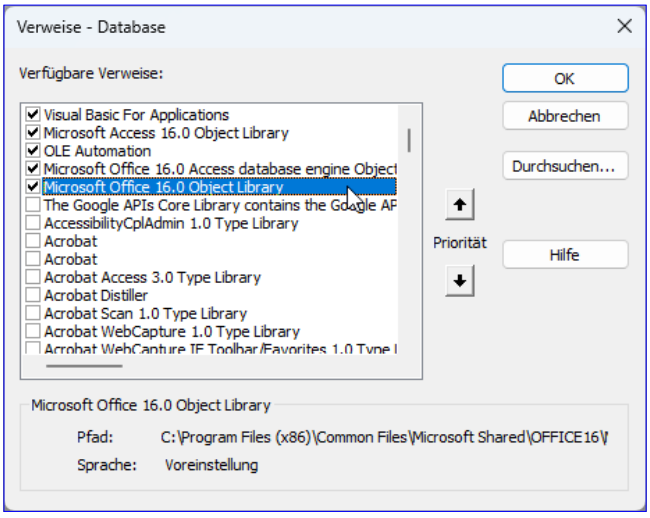

Bild 1: Aktivieren des Verweises auf die Bibliothek **Microsoft Office x.0 Object Library**

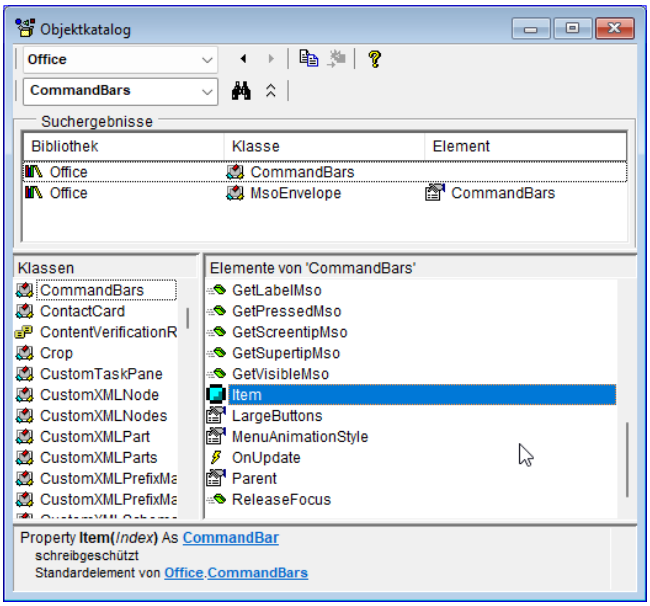

Bild 2: Die **CommandBars**-Auflistung im Objektkatalog

VBA-Editor der jeweiligen Anwendung, beispielsweise mit der Tastenkombination **Alt + F11**. Im VBA-Editor zeigen wir mit dem Menübefehl **Extras|Verweise** den **Verweise**-Dialog an. Hier selektieren Sie den Eintrag aus Bild 1 und fügen diese so hinzu.

Das Objektmodell zum Anpassen von Kontextmenüs

Zeigen wir nun mit **F2** den Objektkatalog an, können wir dort die Bibliothek **Office** auswählen und finden dort beispielsweise die Auflistung **CommandBars** vor (siehe Bild 2). Diese bietet bereits einige interessante Elemente an. Unter anderem können wir damit die Anzahl aller aktuell verfügbaren **CommandBar**-Elemente ausgeben:

```
Debug.Print "Anzahl Commandbars: " _
    & Application.CommandBars.Count
```

Das liefert beispielsweise für Access die Anzahl von 247 **CommandBar**-Elemente, unter Excel immerhin noch 199 Einträge. Ob es sich dabei ausschließlich um Kontextmenüs handelt, finden wir gleich heraus. Unter Access können wir direkt auf die **CommandBars**-Auflistung zugreifen, unter Excel und Word müssen wir einen Verweis auf die **Application**-Klasse voranstellen. Bei dem Zugriff von COM-Add-Ins aus ist dies ohnehin obligatorisch, mehr dazu zeigen wir in einem Beispiel in einem weiteren Artikel namens **Kontextmenü per COM-Add-In anpassen** (**www.vbentwickler.de/370**).

CommandBar-Elemente durchlaufen

Mit den folgenden Zeilen aus der Prozedur **CommandBarsDurchlaufen** geben wir die Namen aller **CommandBar**-Elemente im Direktbereich des VBA-Editors aus:

```
Dim cbr As CommandBar
For Each cbr In Application.CommandBars
    Debug.Print cbr.Name
Next cbr
```

Das Gleiche gelingt auch per **For...Next**-Schleife über den 1-basierten Index:

```
Dim cbr As CommandBar
Dim i As Integer
For i = 1 To Application.CommandBars.Count
    Set cbr = Application.CommandBars.Item(i)
    Debug.Print cbr.Name
Next i
```

Die Ausgabe können wir noch verfeinern, indem wir zwei weitere Eigenschaften ausgeben lassen:

```
Debug.Print cbr.Name, cbr.Type, cbr.BuiltIn
```

Typ eines CommandBar-Elements ermitteln

Die erste hier verwendete Eigenschaft lautet **Type**. **Type** kann einen der folgenden drei Werte liefern:

- **0** (**msoBarTypeNormal**): Das **CommandBar**-Element ist eine Symbolleiste.

- **1** (**msoBarTypeMenuBar**): Das **CommandBar**-Element ist eine Menüleiste.

- **2** (**msoBarTypePopup**): Das **CommandBar**-Element ist ein Kontextmenü.

Die ersten beiden Typen sind für unsere Zwecke nicht relevant. Bei diesen handelt es sich um die Typen der bis Office 2003 verwendeten Elemente, die dort angezeigt wurden, wo jetzt das Ribbon erscheint. Interessant wird es, wenn die Eigenschaft **Type** den Wert **2** liefert. Geben wie also nun alle Kontextmenüs aus und lassen uns, da wir deren Anzahl nicht über die **Count**-Eigenschaft bestimmen können, mit der Zählervariablen **i** den Index ausgeben:

```
Dim cbr As CommandBar
Dim i As Integer
For Each cbr In Application.CommandBars
    If cbr.Type = msoBarTypePopup Then
        i = i + 1
        Debug.Print i, cbr.Name, cbr.BuiltIn
    End If
Next cbr
```

Lassen wir die Prozedur unter Access laufen, erhalten wir 138 Kontextmenüs, unter Excel nur 67.

Eingebaute oder benutzerdefinierte Kontextmenüs

Mit der bereits verwendeten Eigenschaft **BuiltIn** können wir einen **Boolean**-Wert ausgeben, der angibt, ob es sich um ein eingebautes Kontextmenü handelt. Damit können wir die eingebauten Kontextmenüs von denen unterscheiden, die wir selbst per VBA hinzufügen.

Wozu die eingebauten Kontextmenüs nutzen?

Wozu sollten wir überhaupt wissen wollen, welche eingebauten Kontextmenü die jeweiligen Anwendungen anbieten? Weil es geschehen kann, dass wir nicht gleich ein vollständiges Kontextmenü mit neuen Funktionen hinzufügen wollen, sondern nur einzelne Funktionen, die in bestehenden Kontextmenüs angezeigt werden sollen.

Wenn wir beispielsweise unter Excel eine Funktion hinzufügen wollen, die nur im Zusammenhang mit einer Zelle angezeigt werden soll – also wenn der Benutzer mit der rechten Maustaste auf eine Zelle klickt – dann müssen wir den Namen dieses Kontextmenüs herausfinden und passen dann dieses Kontextmenü an.

Namen eines Kontextmenüs herausfinden

Den Namen eines Kontextmenüs kann man auf mehrere Arten ermitteln. Die erste ist die Ausgabe der Namen aller Kontextmenüs der jeweiligen Anwendung. Dazu führen wir die obige Prozedur zum Durchlaufen aller Kontextmenüs in der jeweiligen Anwendung aus. In vielen Fällen können wir schon am Namen des Kontextmenüs erkennen, wann dieses angezeigt wird.

Im Falle von Excel und einem Kontextmenü für eine Zelle werden wir schnell fündig – das Kontextmenü heißt **Cell** (siehe Bild 3).

Unter anderen Anwendungen wie Access ist es teilweise schwierig, das richtige Kontextmenü aufgrund

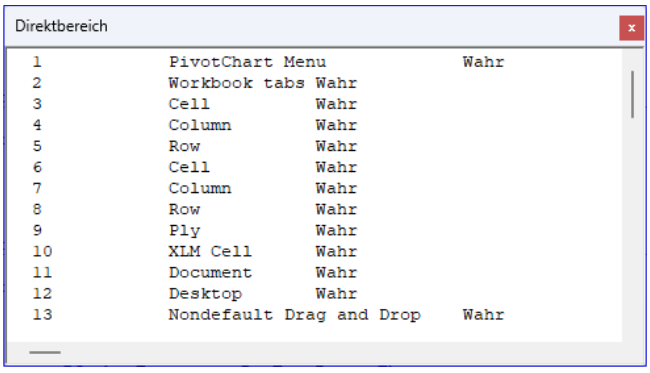

Bild 3: Ausgabe der Kontextmenüs von Excel

der Benennung zu finden. In diesem Fall kommt eine zweite Technik zum Zuge.

Beispielkontextmenüs in den anderen Anwendungen

Unter Excel nutzen wir zu Beispielzwecken wie beschrieben die Kontextmenüs **Cell** oder **Column**. In der Beispieldatei für Word schauen wir uns in den Beispielen das Kontextmenü **Text** an, das beim Rechtsklick auf normalen Text im Dokument erscheint. In der Beispieldatei für Access verwenden wir das Kontextmenüs eines Elements des Navigationsbereichs namens **Navigation Pane object Pop-up**.

Name des Kontextmenüs in das Kontextmenü schreiben

Dabei greifen wir einer Technik vor, die wir weiter unten noch genauer erläutern. Wir fügen jedem Kontextmenü eine Schaltfläche hinzu, für deren Beschriftung wir den Namen des **CommandBar**-Elements festlegen. Dabei durchlaufen wir wieder alle Elemente der **CommandBars**-Auflistung und rufen jeweils für die **Controls**-Auflistung die **Add**-Methode auf. Dieser übergeben wir als ersten Parameter den Typ des zu erstellenden Steuerelements, in diesem Fall **msoControlButton** für eine Schaltfläche. Außerdem hinterlegen wir für den letzten Parameter **Temporary** den Wert **True**. Dies sorgt dafür, dass die angelegten Steuerelemente nur für die aktuelle Session der jeweiligen Anwendung angezeigt werden und bei einem weiteren Start nicht erneut erscheinen:

```
Public Sub NameInKontextmenuesSchreiben()
    Dim cbr As CommandBar
    Dim cbb As CommandBarButton
    Dim i As Integer
    For Each cbr In Application.CommandBars
        If cbr.Type = msoBarTypePopup Then
            Set cbb = cbr.Controls.Add( _
                msoControlButton, , , , True)
            cbb.Caption = cbr.Name
        End If
    Next cbr
End Sub
```

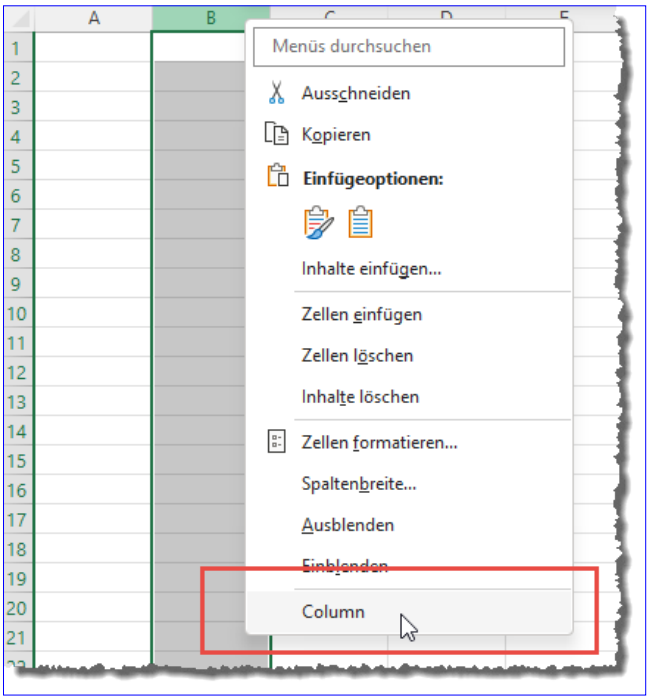

Bild 4: Name eines **CommandBar**-Elements als Schaltflächenbeschriftung

Führen wir diese Prozedur aus und zeigen mit der rechten Maustaste eines der Kontextmenüs der Anwendung an, liefert dieses in einem neuen Eintrag den Namen des Kontextmenüs (siehe Bild 4).

Mit dem Namen eines Kontextmenüs können wir gezielt auf das jeweilige Kontextmenü zugreifen und dessen Elemente bearbeiten oder neue Elemente hinzufügen.

Elemente von Kontextmenüs ausgeben

Wir schauen uns dies zunächst beim Ausgeben der Steuerelemente eines Kontextmenüs an. Die folgende Prozedur referenziert das Kontextmenü **Column** von Excel (wenn Du den in einer anderen Anwendung ausprobieren möchtest, gib den Namen eines dort vorhandenen Steuerelements an). Dann durchläuft die

Prozedur alle Elemente der **Controls**-Auflistung des **CommandBar**-Elements. Wichtig ist, dass wir die Laufvariable **ctl** mit dem Typ **Object** deklarieren. Der Hintergrund ist, dass die **Controls**-Auflistung verschiedene Steuerelementtypen enthalten kann. Deshalb geben wir im Folgenden direkt den Wert der Eigenschaft **Type** des jeweiligen Steuerelements aus. Noch eine bessere Methode, um den Typ zu ermitteln, ist die Ausgabe von **TypeName(ctl):**

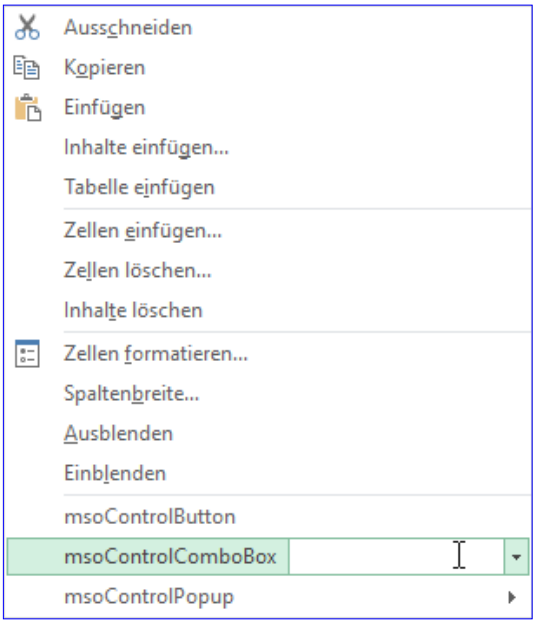

Bild 5: Kontextmenüleiste mit drei verschiedenen neuen Elementen

```
Public Sub ElementeEinesKontextmenuesAusgeben()
    Dim cbr As CommandBar
    Dim ctl As Object
    Set cbr = Application.CommandBars("Column")
    For Each ctl In cbr.Controls
        Debug.Print ctl.Caption, ctl.Type, TypeName(ctl)
    Next ctl
End Sub
```

Die Ausgabe für das **Column**-Kontextmenü lautet:

```
Auss&chneiden           1       CommandBarButton
K&opieren               1       CommandBarButton
Einfü&gen               1       CommandBarButton
Inhalte einfü&gen...    1       CommandBarButton
Tabelle e&infügen       1       CommandBarButton
&Datentyp               10      CommandBarPopup
Zellen &einfügen        1       CommandBarButton
Zellen l&öschen         1       CommandBarButton
Inhal&te löschen        1       CommandBarButton
Zellen &formatieren...  1       CommandBarButton
Spalten&breite...       1       CommandBarButton
&Ausblenden             1       CommandBarButton
Einb&lenden             1       CommandBarButton
&Links entfernen        1
CommandBarButton
Änderun&gen anzeigen    1
CommandBarButton
```

Es gibt eine ganze Reihe von Steuerelementtypen, die Du im Objektkatalog unter der Auflistung **msoControlType** einsehen kannst. Wir können jedoch nur einige wenige nutzen. Unter den eingebauten Steuerelementen finden wir eine ganze Reihe verschiedener Typen, aber wir können nur drei davon mit der Add-Methode zur Controls-Auflistung hinzufügen:

- **1** (**msoControlButton**): Schaltfläche
- **4** (**msoControlComboBox**): Kombinationsfeld
- **10** (**msoControlPopup**): Untermenü

Das erledigen wir beispielhaft mit der folgenden Prozedur. Hier fügen wir nicht nur die drei verschiedenen Typen hinzu, sondern zeigen gleich noch das angepasste Kontextmenü mit der Methode **ShowPopup** an:

```
Public Sub ElementeAnlegen()
    Dim cbr As CommandBar
    Dim ctl As Object
    Set cbr = Application.CommandBars("Column")
    Set ctl = cbr.Controls.Add(1, , , , True)
    ctl.Caption = "msoControlButton"
    Set ctl = cbr.Controls.Add(4, , , , True)
    ctl.Caption = "msoControlComboBox"
    Set ctl = cbr.Controls.Add(10, , , , True)
    ctl.Caption = "msoControlPopup"
    cbr.ShowPopup
End Sub
```

Outlook: Ribbon per COM-Add-In anpassen

Wenn wir Outlook um eigene Funktionen erweitern wollen, stellt sich eine Frage: Wie wollen wir diese auslösen? Es gibt einige Ereignisse, die wir bereits im Artikel Outlook: Explorer automatisieren (www.vbentwickler.de/307) erläutert haben. Diese werden beispielsweise durch Benutzeraktionen wie das Verschieben einer E-Mail in einen anderen Ordner ausgelöst. Aber wie können wir eigene Funktionen über die Benutzeroberfläche starten? Dazu bietet sich das Ribbon an. Hier können wir eigene Bereiche definieren, in denen wir unsere Funktionsaufrufe unterbringen. Der vorliegende Artikel erläutert, wie wir das Ribbon unter Outlook anpassen. Dabei sind einige Dinge zu berücksichtigen – zum Beispiel, dass es nicht wie bei den übrigen Office-Anwendungen nur ein Fenster gibt, das eine eigene Ribbondefinition verwendet.

Ribbon-Elemente im Überblick

Wir beginnen direkt mit dem eingangs erwähnten Umstand, dass Outlook nicht nur eine Ribbondefinition verwendet wie die übrigen Office-Anwendungen.

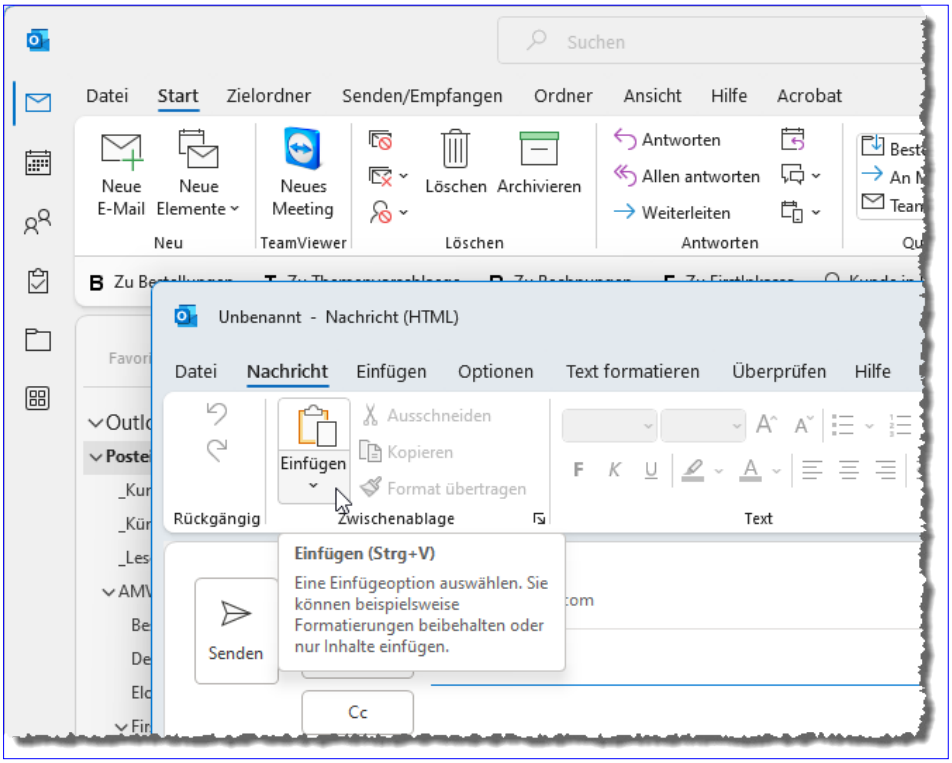

Bild 1: Outlook hat verschiedene Fenster mit Ribbons

Das wird offensichtlich, wenn wir beispielsweise den Ribbonbefehl **Start|Neu|Neue E-Mail** aufrufen. Nicht nur das Outlook-Hauptfenster, sondern auch das Fenster zum Erstellen einer neuen E-Mail weist ein Ribbon auf (siehe Bild 1). Das macht die Sache im Gegensatz zu den übrigen Office-Anwendungen komplizierter. Woher wissen wir, für welches Fenster wir gerade das Ribbon anpassen? Und wie passen wir es in Outlook überhaupt an?

Ribbon per Benutzeroberfläche anpassen

Genau wie die übrigen Office-Anwendungen bietet auch Outlook die Möglichkeit, das Ribbon über die Benutzeroberfläche anzupassen. Dazu öffnen wir die Outlook-Optionen und wechseln zum Bereich **Menüband anpassen** (siehe Bild 2). Hier finden wir in der rechten Liste einige Einträge mit den Hauptregisterkarten. Je nachdem, welcher Objekttyp gerade im Explorer angezeigt wird, also beispielsweise E-Mail oder Termin, erscheint ein anderes **Start**-Tab. Diese werden

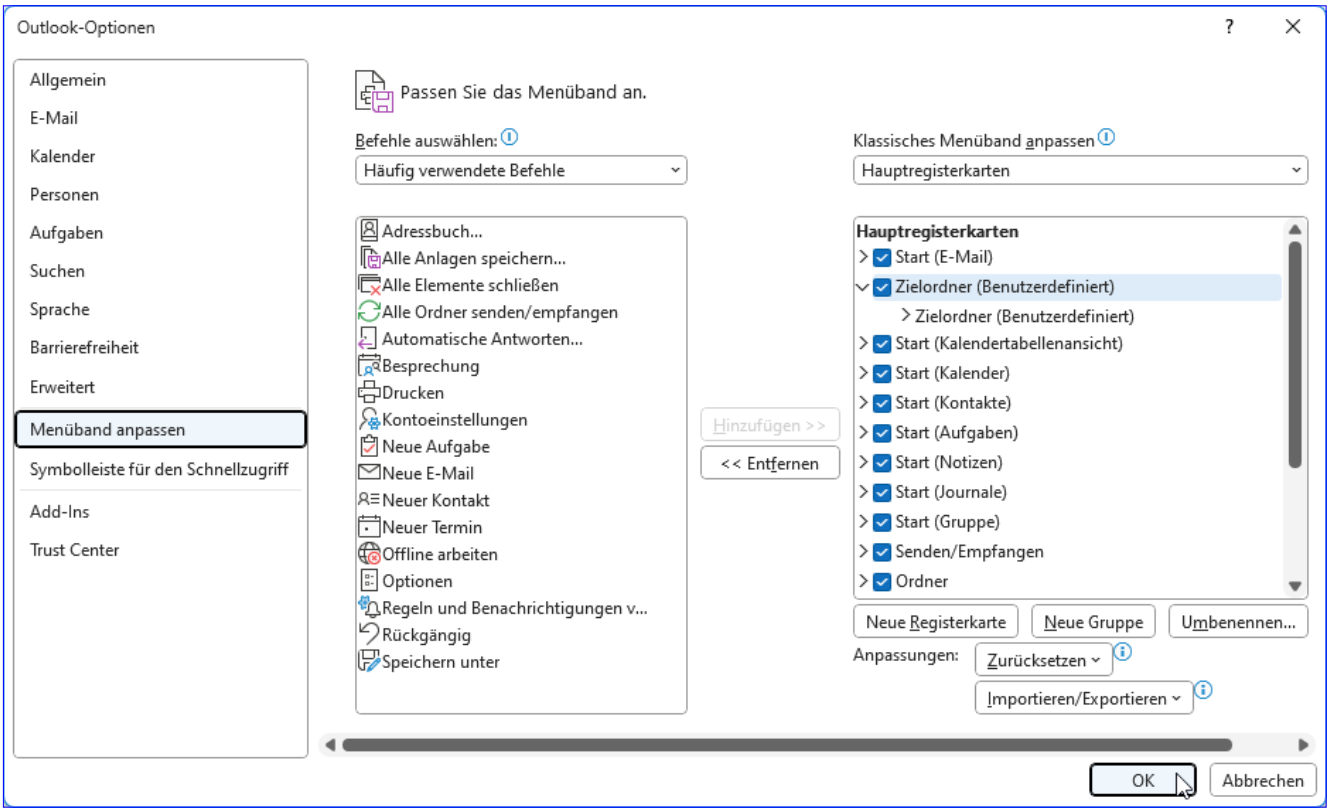

Bild 2: Anpassen des Ribbons im Optionen-Dialog von Outlook

auch in der Liste der Hauptregisterkarten dargestellt – zum Beispiel **Start (E-Mail)** oder **Start (Kalender)**.

Ribbondefinition anpassen

Welche Möglichkeiten haben wir, das Ribbon von Outlook und seiner Explorer wie beispielsweise zum Erstellen von E-Mails anzupassen, wenn wir nicht die Benutzeroberfläche nutzen wollen?

Die einzige Möglichkeit ist die Programmierung eines COM-Add-Ins. Zum Glück kennen wir mit twinBASIC ein praktisches Tool, um COM-Add-Ins zu programmieren. Im Artikel **COM-Add-Ins mit twinBASIC** (**www.vbentwickler.de/311**) haben wir uns bereits die Grundlagen der Erstellung von COM-Add-Ins mit twinBASIC angesehen. Darauf bauen wir auf und erstellen mit twinBASIC ein neues COM-Add-In auf Basis der Vorlage **Samples|MyCOMAddin**. Dieses passen wir wie nachfolgend beschrieben an, um einen ersten eigenen Ribbon-Eintrag in Outlook zu erzeugen.

Projekteinstellungen anpassen

Nach dem Erstellen des Projekts ändern wir die Projekteinstellungen und fügen einen Verweis auf die Outlook-Bibliothek hinzu. Dazu klicken wir im **Project Explorer** auf den Eintrag **Settings**. Hier tragen wir für **Project: Name**, **Project: Description** und **Project: Application Title** jeweils den Wert **OutlookRibbonAnpassen** ein (siehe Bild 3).

Danach fügen wir im gleichen Bereich weiter unten unter **COM Type Library / ActiveX References** den Verweis **Microsoft Outlook 16.0 Object Library** hinzu.

Danach passen wir die Klasse **MyCOMAddin** an, indem wir diese in **OutlookRibbonAnpassen** umbe-

nennen. Entsprechend ändern wir auch den Namen der Klasse im Code. Der allgemeine Teil des Codes der Klasse sieht nun wie folgt aus. Hier sehen wir die **ClassId**, mit der das COM-Add-In in der Registry einigetragen wird. Darunter folgt die eigentliche Definition der Klasse, die zwei Schnittstellen implementiert. **IDTExtensibility2** enthält die Elemente, die für das Laden des COM-Add-Ins notwendig sind. **IRibbonExtensibility** liefert die Elemente, die für das Anwenden von Anpassungen der Ribbondefinition nötig sind. Außerdem definieren wir hier eine Objektvariable namens **objOutlook** mit dem Typ **Outlook.Application**. Diese füllen wir gleich im Anschluss mit einem Verweis auf die Outlook-Instanz, welche das COM-Add-In lädt:

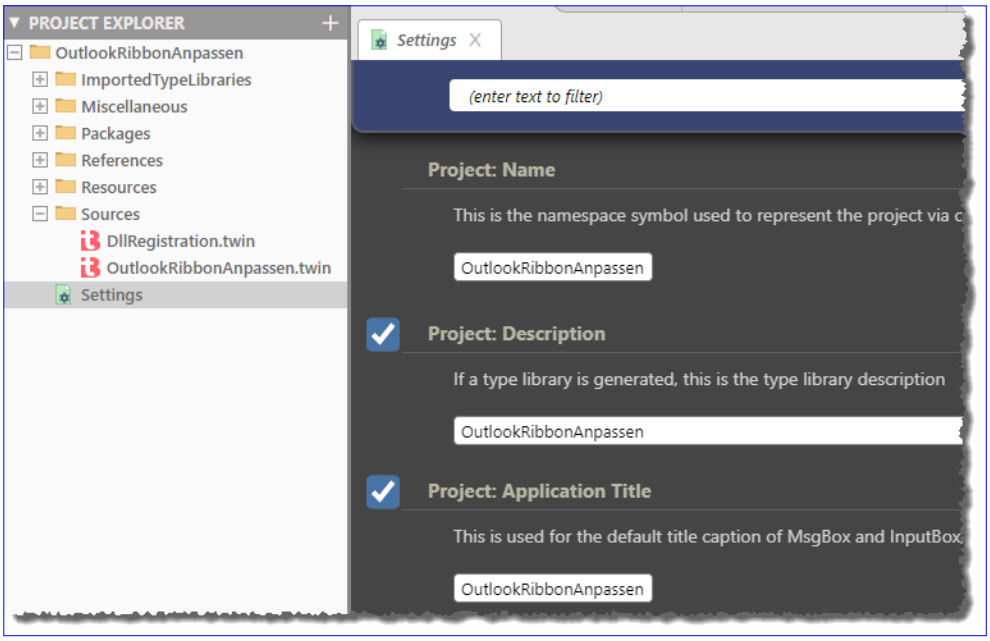

Bild 3: Anpassen der Projekteinstellungen

```
[ ClassId ("2933A563-3C51-4120-AC8A-BA10CC656CA7") ]
Class OutlookRibbonAnpassen
    Implements IDTExtensibility2
    [ WithDispatchForwarding ]
    Implements IRibbonExtensibility

    Private objOutlook As Outlook.Application
    ...
End Class
```

Für die Schnittstelle **IDTExtensitility2** müssen wir die folgenden fünf Ereignisprozeduren implementieren, von denen wir nur die erste mit einer Anweisung füllen. Das Ereignis **OnConnection** wird beim Verbinden der Anwendung mit dem COM-Add-In ausgelöst und liefert unter anderem mit dem Parameter **Application** einen Verweis auf die aufrufende Anwendung, in diesem Fall Outlook. Den Inhalt dieses Parameters weisen wir der Variablen **objOutlook** zu. Diese und die übrigen Ereignisprozeduren sehen wir in Listing 1.

Ribbondefinition anpassen

Zum Anpassen der Ribbondefinition beginnen wir mit dem Hinzufügen eines Icons, das wir im Ribbon für eine neu zu erstellende Schaltfläche anzeigen wollen. Dieses fügen wir zum Projekt hinzu, indem wir im Bereich Ressources des Projektexplorers mit der rechten Maustaste auf den Eintrag **ICON** klicken und **Add|Import File...** auswählen. Im nun erscheinenden Dialog wählen wir die gewünschte **.ico**-Datei aus. Diese wird anschließend wie in Bild 4 angezeigt.

Damit kommen wir zu der Funktion, die durch die Schnittstelle **IRibbonExtensibility** beim Laden des COM-Add-Ins ausgelöst wird. Diese heißt **GetCustomUI** und enthält einen Parameter namens **RibbonID**. Dieser wird gleich noch eine entscheidende Rolle spielen. Erst einmal gehen wir so vor, wie wir es auch bei COM-Add-Ins für die übrigen Office-An-

```
Sub OnConnection(ByVal Application As Object, ByVal ConnectMode As ext_ConnectMode, ByVal AddInInst As Object, _
        ByRef custom As Variant()) Implements IDTExtensibility2.OnConnection
    Set objOutlook = Application
End Sub

Sub OnDisconnection(ByVal RemoveMode As ext_DisconnectMode, ByRef custom As Variant()) _
        Implements IDTExtensibility2.OnDisconnection

End Sub

Sub OnAddInsUpdate(ByRef custom As Variant()) Implements IDTExtensibility2.OnAddInsUpdate

End Sub

Sub OnStartupComplete(ByRef custom As Variant()) Implements IDTExtensibility2.OnStartupComplete

End Sub

Sub OnBeginShutdown(ByRef custom As Variant()) Implements IDTExtensibility2.OnBeginShutdown


End Sub
```

Listing 1: Implementierung der Schnittstelle **IDTExtensibility2**

wendungen getan haben. Dazu legen wir die Funktion **GetCustomUI** wie in Listing 2 an. Die Funktion stellt den Code der Ribbondefinition zusammen und gibt diesen als Funktionsergebnis zurück. Die Ribbondefinition enthält zunächst nur ein **tab**-Element mit einem **group**-Element und einem **button**-Element. Für das übergeordnete **customUI**-Element haben wir das Attribut **loadImage** mit dem Wert **LoadImage** hinterlegt. Diese Callbackprozedur hinterlegen wir ebenfalls in der Klasse:

```
Function LoadImage(imageId As String) As IPictureDisp
    Return LoadResPicture(imageId, _
        vbResBitmapFromIcon, 32, 32)
End Function
```

Sie ist dafür verantwortlich, für jedes Element, welches das Attribut **image** enthält, das dort angegebene Bild aus den Ressourcen zu laden. Das Attribut **image** haben wir mit dem Wert **mail.ico** für das **button**-Element hinterlegt, also mit dem Namen der zuvor importierten Icon-Datei.

Für das Attribut **onAction** des **button**-Elements haben wir schließlich noch die folgende Prozedur angelegt.

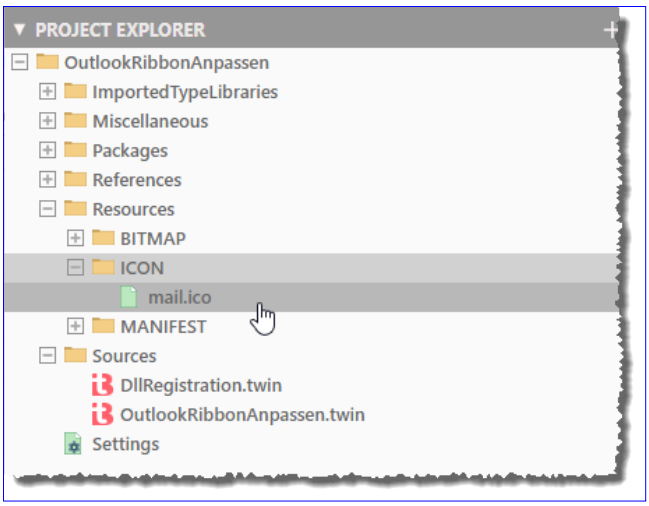

Bild 4: Ein neues Icon im Projektexplorer

Outlook: Kontextmenüs anpassen

In den Office-Anwendungen Word, Excel oder Access passen wir vorhandene Kontextmenüs per VBA über das Objektmodell von Office an. Auch das Hinzufügen und die Anzeige benutzerdefinierter Kontextmenüs erledigen wir auf diese Weise. Unter Outlook sieht die Situation anders aus: Hier wurde die Definition von Kontextmenüs bereits in die Ribbondefinition integriert. Wir haben dort einen eigenen Abschnitt namens »contextMenu« mit dem wir vorhandene Kontextmenüs anpassen und erweitern können. In diesem Artikel schauen wir uns an, wie wir solche Anpassungen vornehmen und welche Möglichkeiten sich daraus ergeben.

Kontextmenüs in Outlook

Outlook bietet genau wie die übrigen Office-Anwendungen eine Reihe von Kontextmenüs, die für die verschiedensten Elemente oder Bereiche aufgerufen werden können. In Bild 1 sehen wir beispielsweise das Kontextmenü für eine E-Mail.

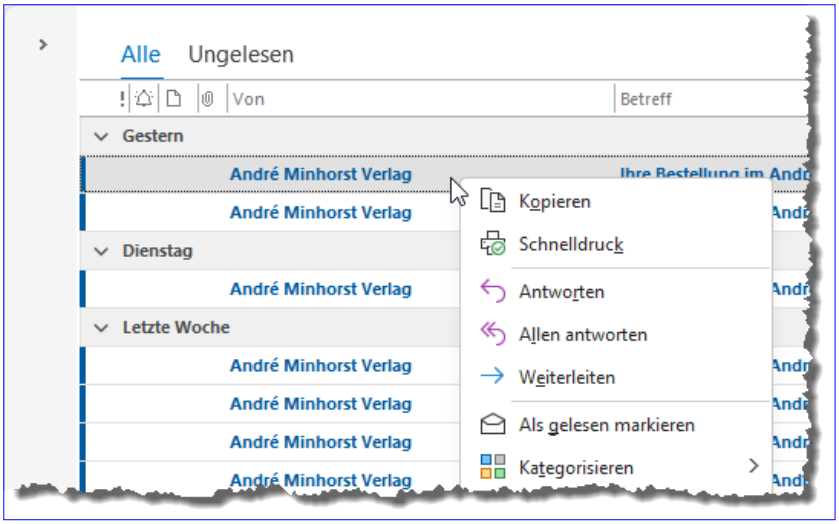

Bild 1: Ein Kontextmenü in Outlook

Technik zum Anpassen

Die schlechte Nachricht ist: Kontextmenüs unter Outlook lassen sich nur per COM-Add-In anpassen. Die gute ist: Die Basis dafür haben wir bereits in einem weiteren Artikel namens **Outlook: Ribbon per COM-Add-In anpassen** (**www.vbentwickler.de/376**) gelegt. Dort haben wir mit dem Tool **twinBASIC** bereits ein COM-Add-In erstellt, mit dem wir die normalen Ribbonelemente anpassen können, also die **tab**-, **group**- oder **button**-Elemente im Ribbon von Outlook. Dort haben wir auch bereits die Besonderheit hervorgehoben, dass Outlook als einzige Office-Anwendung mehrere Fenster mit jeweils einem eigenen Ribbon hat. Auch dies ist beim Anpassen von Kontextmenüs zu berücksichtigen.

Anpassen oder auch neue Kontextmenüs definieren?

Die nächste Frage, die wir uns stellen müssen: Können wir nur bestehende Kontextmenüs erweitern oder auch neue Kontextmenüs erstellen? Unter Access beispielsweise kann man auf das Betätigen der rechten Maustaste per Ereignisprozedur reagieren und per VBA ein neues Kontextmenü zusammenstellen und dieses auch anzeigen. Probieren wir also aus, ob wir auch in Outlook benutzerdefinierte Kontextmenüs per VBA erstellen können. Dazu fügen wir die folgende Prozedur in ein neues Modul im VBA-Editor von Outlook ein und führen sie aus:

```
Public Sub KontextmenueAnlegen()
    Dim cbr As CommandBar
    Dim cbb As CommandBarButton
    Set cbr = Application.CommandBars.Add( _
        "Benutzerdefiniert", msoBarPopup, , True)
```

```
Private Function GetCustomUI(ByVal RibbonID As String) As String Implements IRibbonExtensibility.GetCustomUI
    Dim strXML As String
    strXML &= "<customUI xmlns=""http://schemas.microsoft.com/office/2009/07/customui"" loadImage=""LoadImage"">" _
        & vbCrLf
    strXML &= "  <contextMenus>" & vbCrLf
    strXML &= "    <contextMenu idMso=""ContextMenuText"">" & vbCrLf
    strXML &= "      <button id=""btn"" label=""Beispielbutton"" onAction=""onAction"" image=""add.ico""/>" & vbCrLf
    strXML &= "    </contextMenu>" & vbCrLf
    strXML &= "  </contextMenus>" & vbCrLf
    strXML &= "</customUI>" & vbCrLf
    Return strXML
End Function
```

Listing 1: Zusammenstellen einer Ribbondefinition zur Erweiterung eines Kontextmenüs

```
    ...
    cbr.ShowPopup
End Sub
```

Die Prozedur scheitert bereits bei Verwendung der **CommandBars**-Auflistung. Diese liefert den Fehler **Objekt unterstützt diese Eigenschaft oder Methode nicht**.

Kontextmenü anpassen

Wir müssen uns beim Anpassen des Kontextmenüs also offensichtlich auf das Erweitern oder Ändern von eingebauten Elementen beschränken. Wir schauen uns an einem Beispiel an, wie das grundsätzlich funktioniert. Dabei gehen wir davon aus, dass wir schon ein COM-Add-In wie im Artikel **Outlook: Ribbon per COM-Add-In anpassen** (**www.vbentwickler.de/376**) beschrieben erstellt haben.

Für dieses brauchen wir zunächst nur die Funktion **GetCustomUI** anzupassen, die den Code für die Ribbondefinition zusammenstellt. In diesem Fall fügen wir statt des **ribbon**-Elements das Element **contextMenus** zur Ribbondefinition hinzu (siehe Listing 1). Wir können natürlich auch beide gleichzeitig verwenden. Unterhalb dieses Elements legen wir für jedes Kontextmenü, das wir anpassen wollen, ein **contextMenu**-Element an.

Um festzulegen, welches Kontextmenü wir anpassen wollen, benötigen wir die **idMso** des Elements. In diesem Fall nutzen wir den Wert **ContextMenuText**. Dieses Kontextmenü erscheint beispielsweise, wenn wir mit der rechten Maustaste in den Entwurf einer E-Mail klicken (wie wir an den Namen der idMso für ein Kontextmenü kommen, beschreiben wir weiter unten).

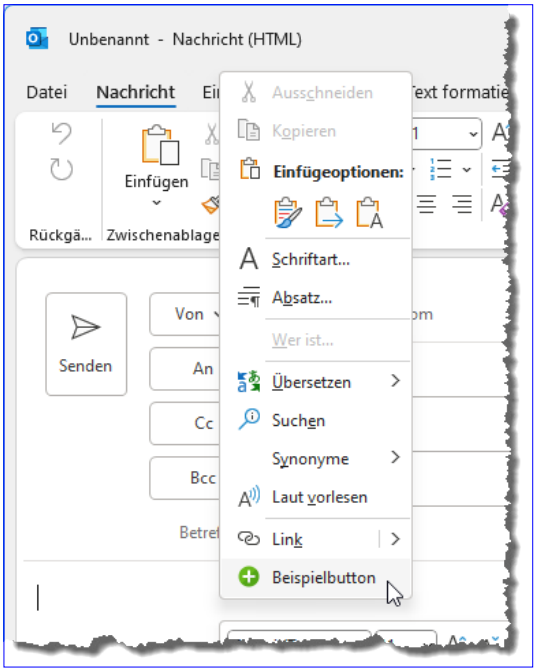

Bild 2: Benutzerdefinierter Button im Kontextmenü

Wenn wir nun noch ein **button**-Element wie in hinzufügen und das COM-Add-In erstellen, sieht das entsprechende Kontextmenü wie in Bild 2 aus. Das Icon haben wir wie ebenfalls in dem oben genannten Artikel beschrieben

hinzugefügt. Für das **button**-Element haben wir mit dem **onAction**-Attribut festgelegt, welche Prozedur beim Anklicken des Buttons aufgerufen werden soll. Diese sieht so aus:

```
Public Sub OnAction(control As IRibbonControl)
    MsgBox "Steuerelement: " & control.Id
End Sub
```

Dies zeigt beim Betätigen des Kontextmenüs die angegebene Meldung an und gibt den Namen des angeklickten Steuerelements aus.

Kontextmenüs identifizieren

Wenn wir ein Kontextmenü erweitern wollen, wissen wir erst einmal nicht, wie die **idMso** für das entsprechende Element heißt. Dazu können wir die Excel-Dateien mit **idMso**-Werten nutzen, die Microsoft bereitgestellt hat und die wir dem Download zu diesem Artikel beigefügt haben. Diese sind nicht aktuell – Miccrosoft stellt nicht regelmäßig neue Versionen dieser Dateien bereit. Da sich jedoch auch die Ribbons keinen umfangreichen Änderungen unterliegen, ist das erst einmal kein Problem.

Die Namen der meisten Kontextmenüs finden wir in der Datei **outlookexplorercontrols.xlsx**. Hier haben wir für das Feld **ControlType** einen Filter mit dem Wert **contextMenu** definiert und erhalten so die Bezeichnungen aller in der Tabelle enthaltenen Kontextmenüs (siehe Bild 3).

Wie können wir nun, abgesehen von der intuitiven Zuordnung der Bezeichnungen zu den Bereichen, herausfinden, welches contextMenu-Element in welchem Kontext angezeigt wird?

Dazu fügen wir einfach einem COM-Add-In allen **contextMenu**-Elementen einen button hinzu, der den Namen des jeweiligen **contextMenu**-Elements enthält. Dazu nutzen wir unsere Kenntnisse der Excel-VBA-Programmierung und durchlaufen in einer Schleife alle Zeilen des Excel-Dokuments (siehe Listing 2).

Dabei prüfen wir jeweils, ob die zweite Spalte den Eintrag **contextMenu** enthält. In diesem Fall stellen wir drei Codezeilen für das COM-Add-In zusammen, die im Ergebnis beispielsweise wie folgt aussehen (jeweils in einer Zeile):

```
strXML &= "    <contextMenu idMso=""ContextMenuFolder"">"
& vbcrlf
strXML &= "      <button id=""btnContextMenuFolder""
label=""ContextMenuFolder"" onAction=""onAction"" ima-
ge=""add.ico""/>" & vbcrlf
strXML &= "    </contextMenu>" & vbcrlf
```

Solch ein Konstrukt erstellen wir für alle contextMenu-Elemente und fügen diese dann der obigen getCustomUI-Funktion hinzu. Wenn wir das COM-Add-In

	A	B	C	D
1	Control Name	Control Type	Tab Set	Tab
1432	ContextMenuMailMoreActions	contextMenu	None (Context Menu)	None (Context Menu)
1438	ContextMenuCalendarMoreActions	contextMenu	None (Context Menu)	None (Context Menu)
1439	ContextMenuContactsMoreActions	contextMenu	None (Context Menu)	None (Context Menu)
1446	ContextMenuTasksMoreActions	contextMenu	None (Context Menu)	None (Context Menu)
1447	ContextMenuNotesMoreActions	contextMenu	None (Context Menu)	None (Context Menu)
1448	ContextMenuJournalMoreActions	contextMenu	None (Context Menu)	None (Context Menu)
1449	ContextMenuAttachMoreActions	contextMenu	None (Context Menu)	None (Context Menu)
1450	MenuMailNewItem	contextMenu	None (Context Menu)	None (Context Menu)
1471	MenuCalendarNewItem	contextMenu	None (Context Menu)	None (Context Menu)
1489	MenuContactsNewItem	contextMenu	None (Context Menu)	None (Context Menu)
1504	MenuTasksNewItem	contextMenu	None (Context Menu)	None (Context Menu)
1518	MenuNotesNewItem	contextMenu	None (Context Menu)	None (Context Menu)
1532	MenuJournalNewItem	contextMenu	None (Context Menu)	None (Context Menu)
1546	ContextMenuMailItem	contextMenu	None (Context Menu)	None (Context Menu)
1630	ContextMenuMultipleItems	contextMenu	None (Context Menu)	None (Context Menu)
1687	ContextMenuGroupHeader	contextMenu	None (Context Menu)	None (Context Menu)

outlookexplorercontrols

Bild 3: Namen der **ContextMenu**-Elemente

Outlook: Anhang speichern per Kontextmenü

Das Speichern von Anhängen in E-Mails gelingt in Outlook recht einfach: Man öffnet die E-Mail, klickt mit der rechten Maustaste auf den Anhang und wählt aus dem Kontextmenü den Eintrag »Speichern unter« aus. Danach allerdings fragt Outlook den Speicherort für den Anhang ab und hier startet man immer im gleichen Verzeichnis – in der Regel das Dokumente-Verzeichnis des aktuellen Benutzers. Dieser Artikel zeigt, wie wir dieses Verzeichnis auf ein anderes Verzeichnis einstellen können, aber das reicht in vielen Fällen nicht aus: Rechnungen sollen in ein bestimmtes Verzeichnis gespeichert werden, Anfragen von Kunden in einem bestimmten Verzeichnis für den jeweiligen Kunden et cetera. Diese Aufgaben werden wir mit einem Tool vereinfachen, das gleich im Kontextmenü die Möglichkeit zum Speichern in verschiedenen Verzeichnissen bietet. Und noch mehr: Wir wollen das Tool so programmieren, dass der Benutzer selbst eintragen kann, welche Kontextmenü-Einträge zum Speichern in verschiedenen Verzeichnissen genutzt werden können.

Ausgangssituation: Viel Arbeit mit dem Dateidialog

Standardmäßig bietet Outlook zwei Aufrufmöglichkeiten zum Speichern von Anhängen an:

- beim Rechtsklick auf einen der Anhänge in der Mail (siehe Bild 1) oder
- beim Markieren des Anhangs im Ribbon (siehe Bild 2).

Beim Anklicken dieser Einträge erscheint ein **Anlage speichern**-Dialog zur Auswahl des Speicherortes, der standardmäßig das Dokumente-Verzeichnis des aktuellen Benutzers anzeigt. Für viele Anwendungen wäre es schon ein Gewinn, wenn sich dieser Dialog das zuletzt verwendete Verzeichnis merken würde, aber das ist nicht der Fall. Für eine einfache Lösung schauen wir uns daher im ersten Schritt an, wie man dieses Verzeichnis dauerhaft umstellen kann.

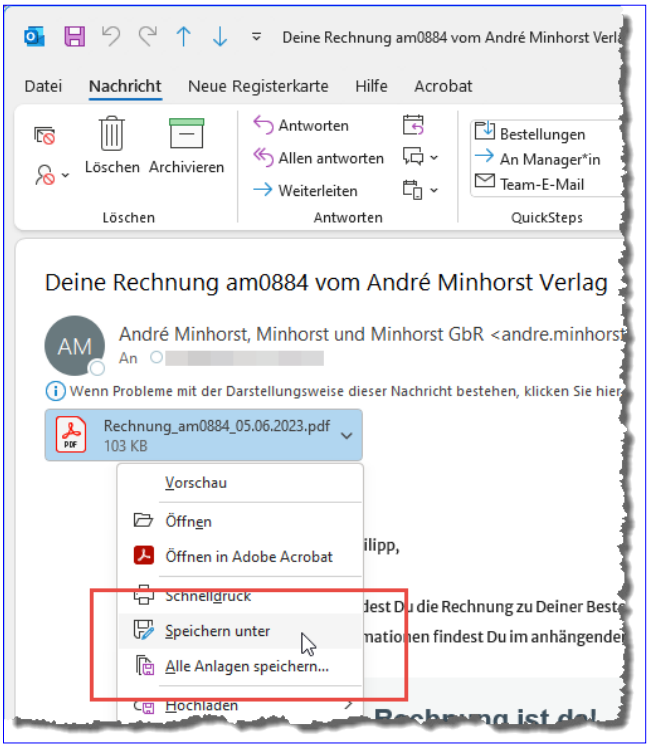

Bild 1: Kontextmenü-Einträge zum Speichern von Anhängen

Zielverzeichnis per Registry einstellen

Dazu öffnen wir die Registry von Windows (in der Window-Suche **RegEdit**

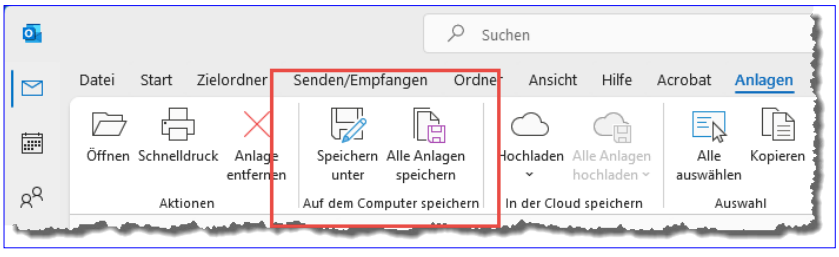

Bild 2: Ribbon-Einträge zum Speichern von Anhängen

eingeben) und navigieren dort zu dem folgenden Eintrag:

```
Computer\HKEY_CURRENT_USER\Software\Microsoft\Office\16.0\Outlook\Options
```

Hier legen wir einen neuen Eintrag mit dem Typ **Zeichenfolge** und dem Namen **DefaultPath** an und tragen für diesen das gewünschte Verzeichnis ein (siehe Bild 3).

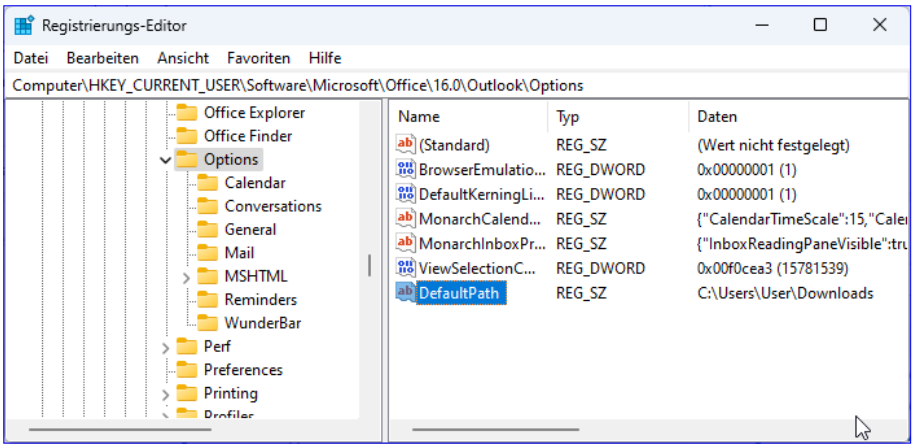

Bild 3: Neues Standardverzeichnis zum Speichern von Attachments

Nach dem nächsten Neustart bietet Outlook beim Speichern von Attachments den dort hinterlegten Ordner an.

Attachments komfortabel speichern per COM-Add-In

In einem weiteren Artikel namens **Kontextmenüs in Outlook anpassen** (**www.vbentwickler.de/369**) haben wir bereits beschrieben, wie wir die Kontextmenüs von Outlook anpassen können.

Ein weiterer Artikel mit dem Titel **Outlook: Ribbon per COM-Add-In anpassen** (**www.vbentwickler.de/376**) zeigt, wie wir das Ribbon nach unseren Bedürfnissen gestalten können.

Die dort vorgestellten Techniken verwenden wir als Grundlage für die nachfolgend vorgestellte Lösung. Im ersten genannten Beitrag haben wir bereits herausgefunden, dass es in der **Attachments**-Auflistung für Outlook-E-Mails nicht nur die Attachments gibt, die offensichtlich als Datei an die E-Mail angehängt wurden, sondern auch solche Dateien berücksichtigt werden, die in die E-Mail eingebettet wurden – beispielsweise als Bilder. Die dort vorgestellten Funktionen **HasAttachedFiles** und **IsFileAttached** verwenden wir auch in diesem Artikel.

Wie wollen wir die Funktionen zum Speichern von Attachments genau gestalten? Das hängt von den Anforderungen ab. Für den einen reicht es, wenn wir einen eigenen Menüpunkt zum Speichern von Rechnungen in einem speziellen Verzeichnis hinzufügen, damit diese von der Buchhaltung weiterverarbeitet werden können.

Der andere hat fünf feste Kunden, für die er immer wieder neue Anwendungen oder Erweiterungen programmiert und hat für jeden dieser Kunden ein Verzeichnis, in dem die von den Kunden geschickten Dokumente mit Anforderungen oder anderen Informationen gespeichert werden können.

Beide Anforderungen könnte man fest in ein COM-Add-In programmieren. Früher oder später wird sich jedoch irgendein Parameter ändern – beispielsweise, weil man den Rechner neu aufsetzt und sich dadurch die Verzeichnisstruktur ändert oder weil ein fester Kunde wegfällt und ein anderer hinzukommt.

Wir sind also gut beraten, wenn wir die zu programmierende Funktion so weit wie möglich flexibel gestalten. Wie können wir das machen? Eine Idee wäre, eine Text- oder XML-Datei zu diesem Zweck zu hinterlegen, welche die benötigten Informationen enthält. Welche Informationen werden das genau sein?

- Kontextmenü-Einträge
- Verzeichnisse, in welche die Attachments beim Betätigen der Kontextmenü-Einträge gespeichert werden

Grundsätzlich wollen wir die Idee aufgreifen, wie im Artikel **Kontextmenüs in Outlook anpassen** (**www.vbentwickler.de/369**) ein **dynamicMenu**-Element zum Kontextmenü hinzuzufügen. Dieses können wir ebenfalls an der entsprechenden Stelle zum Ribbon hinzufügen.

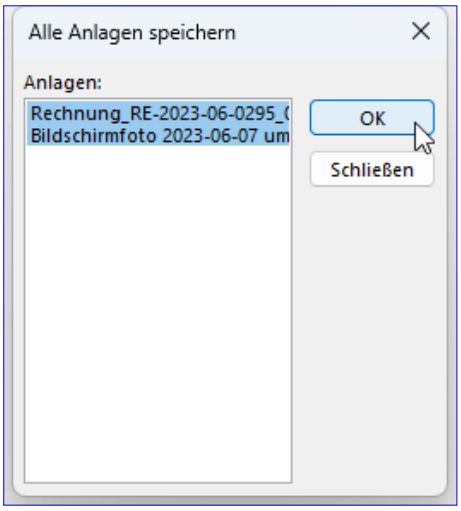

Bild 4: Dialog zur Auswahl, welche Attachments gespeichert werden sollen

Der Clou dabei wäre, dass wir dann direkt das **menu**-Element mit den Untermenüpunkten in eine Textdatei schreiben könnten, die beim Aufrufen des dynamischen Untermenüs eingelesen und verarbeitet wird. Und die Verzeichnisse schreiben wir einfach in das **tab**-Attribut der jeweiligen Elemente. Damit haben wir schon einmal diesen Teil abgedeckt.

Verschiedene Kontextmenüs

Die üblichen Befehle zum Speichern der Attachments einer E-Mail befinden sich im Kontextmenü des mit der rechten Maustaste angeklickten Attachments.

Hier finden wir zwei Einträge, die sich wie folgt verhalten

- **Speichern unter**: Zeigt den **Anlage speichern**-Dialog für diese Datei an.
- **Alle Anlagen speichern**: Öffnet den Dialog aus Bild 4, der alle enthaltenen Anlagen zum Auswählen anzeigt. Nach einem Klick auf **OK** kann man noch das Zielverzeichnis auswählen, in das dann alle Anlagen gespeichert werden.

Die gleichen Befehle finden wir in dem Ribbon-Tab, der beim Aktivieren eines der Attachments aktiviert wird.

Genau in diesen Bereich würden wir unser Untermenü zum gezielten Speichern ebenfalls unterbringen.

Außerdem wollen wir diesen Eintrag auch direkt im Ribbon-Tab namens **Start** anbieten und in dem Kontextmenü, das beim Rechtsklick auf eine E-Mail in der Liste der E-Mails erscheint.

Strategie zum Speichern

Bevor wir diese Einträge anlegen, benötigen wir jedoch noch eine Strategie für die Schritte, die nach dem Anklicken eines der Zielverzeichnis im Menü auszuführen sind.

Die erste Frage ist: Wollen wir dem Benutzer die Möglichkeit geben, den Dateinamen anzupassen? Das würde bedeuten, dass wir einen **Datei speichern**-Dialog anzeigen, der das über den Kontextmenü-Eintrag gewählte Verzeichnis als Standardverzeichnis anzeigt und den Dateinamen des Attachments.

Dieser kann dann ebenso noch angepasst werden wie das Verzeichnis. Letztlich liefert diese Vorgehensweise maximale Flexibilität und erfordert nur einen zusätzlichen Mausklick auf die **Speichern**-Schaltfläche, wenn keine Änderungen an den Dateinamen durchgeführt werden sollen.

Diese Strategie funktioniert super, wenn der Benutzer nur ein Attachment speichern möchte, beziehungsweise wenn er alle speichern will, aber ohnehin nur ein Attachment vorhanden ist.

Was aber, wenn die E-Mail mehr als ein Attachment enthält und der Benutzer wählt den Befehl zum Speichern der Attachments für die aktuell markierte E-Mail aus? Dann lassen wir ihn nur das Zielverzeichnis für diesen Kontextmenü-Eintrag bestätigen.

Für den Fall, dass eine Datei gleichen Namens bereits vorhanden ist, können wir immer noch einmal den **Datei speichern**-Dialog mit dem konkreten Dateinamen anzeigen, damit der Benutzer wählen kann, ob die vorhandene Datei überschrieben werden soll.

Namen der Kontextmenüs ermitteln

Nun schreiten wir zur Tat und ermitteln erst einmal die Namen der Kontextmenüs, die wir mit den zusätzlichen Befehlen ausstatten wollen. Dazu verwenden wir die im Artikel **Kontextmenüs in Outlook anpassen** (**www.vbentwickler.de/369**) vorgeschlagene Strategie, einfach jedem Kontextmenü ein **button**-Element mit dem **idMso**-Namen des jeweiligen Kontextmenüs hinzuzufügen. Darüber finden wir hieraus, dass die **idMso** für das Kontexmenü einer E-Mail im Outlook Explorer **ContextMenuMailItem** lautet. Die **idMso** für das Attachment einer im Inspektor für E-Mails angezeigten E-Mail lautet **ContextMenuAttachments**.

Namen der Ribbon-Elemente ermitteln

Die Namen der Ribbon-Tabs und -Groups, denen wir die Befehle hinzufügen wollen, ermitteln wir über die Excel-Tabelle **outlookexplorercontrols.xlsm**. Hier finden wir schnell den Bereich mit dem Tab namens **TabAttachments**. Gleichen wir die dortigen Elemente mit dem Ribbon-Tab ab, finden wir allerdings schnell heraus, dass die Excel-Datei nicht mehr auf dem aktuellen Stand ist – zumindest nicht für Outlook 2016.

Also gehen wir einen alternativen Weg. Wir öffnen die Outlook-Optionen über den Befehl **Datei|Optionen** und wechseln dort zum Bereich **Menüband anpassen**. Hier stellen wir rechts oben den Eintrag **Alle Registerkarten** ein und suchen dann nach dem Eintrag **Anlagen|Auf dem Computer speichern** (siehe Bild 5). Diesen markieren wir, klicken auf die Schaltfläche **Umbenennen** und ändern den Namen, beispielswei-

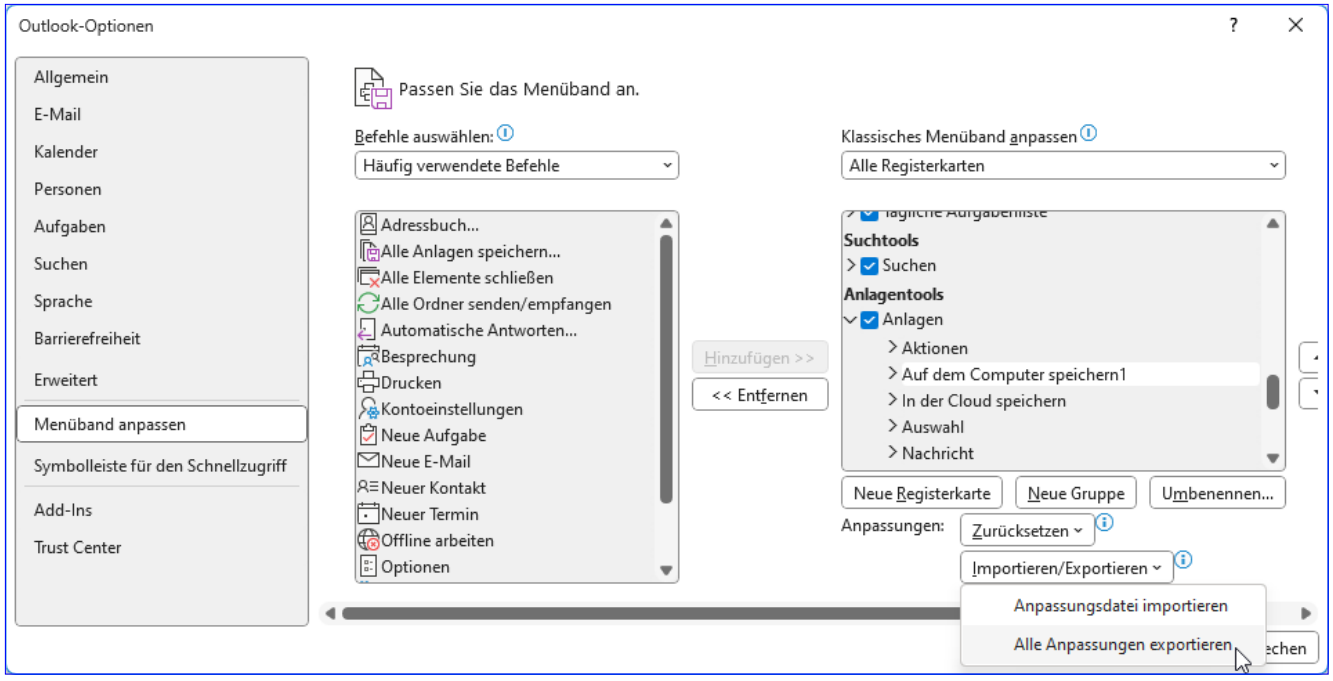

Bild 5: Ermitteln von Elementnamen, Teil 1

se durch Anfügen der Zahl **1**. Wichtig ist nur, dass wir diesen ändern.

Wo wir schon gerade hier sind, klappen wir auch gleich noch den Eintrag auf und fahren nacheinander mit der Maus über die Befehle **Speichern unter** und **Alle Anlagen speichern**. Outlook blendet per Tooltip-Text die Beschreibung des Befehls ein, die mit der in Klammern eingefassten **idMso** abgeschlossen wird (siehe Bild 6). Die Schaltfläche **Speichern unter** heißt **SaveAttachAs** und die Schaltfläche **Alle Anlagen speichern** heißt **SaveAttachments**.

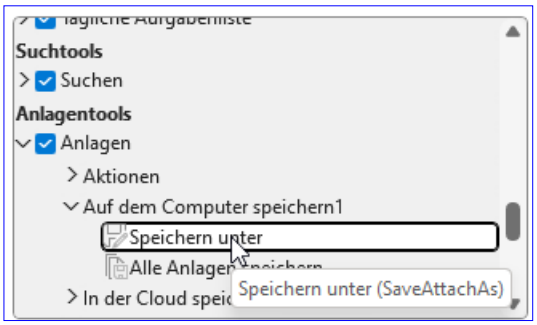

Bild 6: Ermitteln von Elementnamen, Teil 2

Wozu müssen wir die Namen der **button**-Elemente kennen? Diese brauchen wir nicht unbedingt. Es hängt davon ab, ob wir eine eigene Gruppe für unser **dynamicMenu**-Element erstellen wollen. Ist das der Fall, können wir diese einfach neu erstellen.

Wenn wir unseren Befehl jedoch in der Gruppe **Auf dem Computer speichern** unterbringen wollen, können wir das nur über einen Umweg erledigen. Wir müssen die eingebaute Gruppe ausblenden und diese nachbauen.

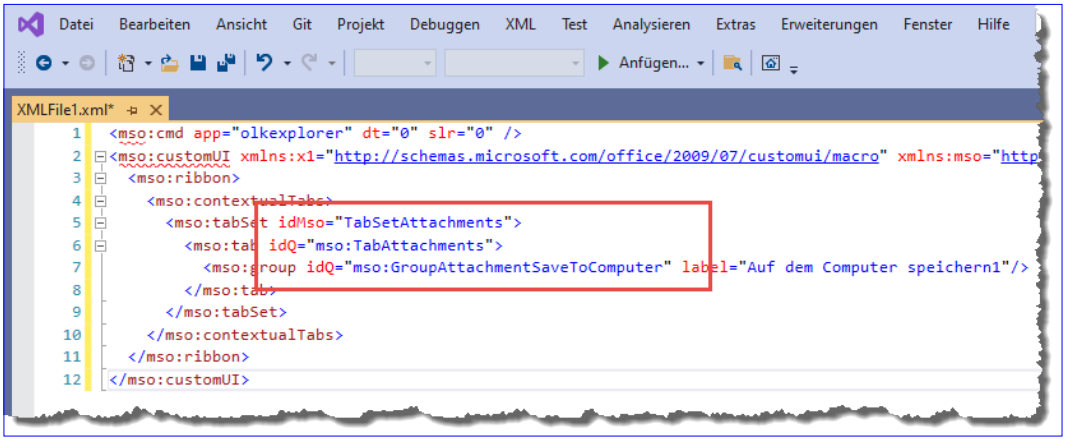

Bild 7: Ermitteln von Elementnamen, Teil 3

Zunächt brauchen wir jedoch noch den Namen des nachzubauenden **group**-Elements. Dieses erhalten wir nun, wenn wir im Optionen-Dialog den Befehl **Importieren/Exportieren|Alle Anpassungen exportieren** auswählen. Dies speichert eine XML-Datei mit allen Änderungen an der Standardkonfiguration im angegebenen Ordner. Den entscheidenden Teil für unser Vorhaben haben wir in Bild 7 durch einen Rahmen markiert.

Programmieren des COM-Add-Ins

Die Programmierung eines COM-Add-Ins für Microsoft Outlook sowie die notwendigen Schritte zum Anpassen haben wir im Artikel **Outlook: Ribbon per COM-Add-In anpassen** (**www.vbentwickler.de/376**) beschrieben.

Nun schauen wir uns die Änderungen an, die wir explizit für die vorliegende Lösung benötigen. Die Lösung findest Du als **.twinproj**-Datei im Download zu diesem Artikel.

Ribbon-Definition laden

Die Funktion **GetCustomUI** wird beim Laden des COM-Add-Ins geladen. Diese nimmt mit dem Parameter **RibbonID** einen Wert entgegen, der angibt, von welchem Outlook-Fenster aus die Funktion **GetCustomUI** aufgerufen wurde. Wir müssen hier eine Unterscheidung