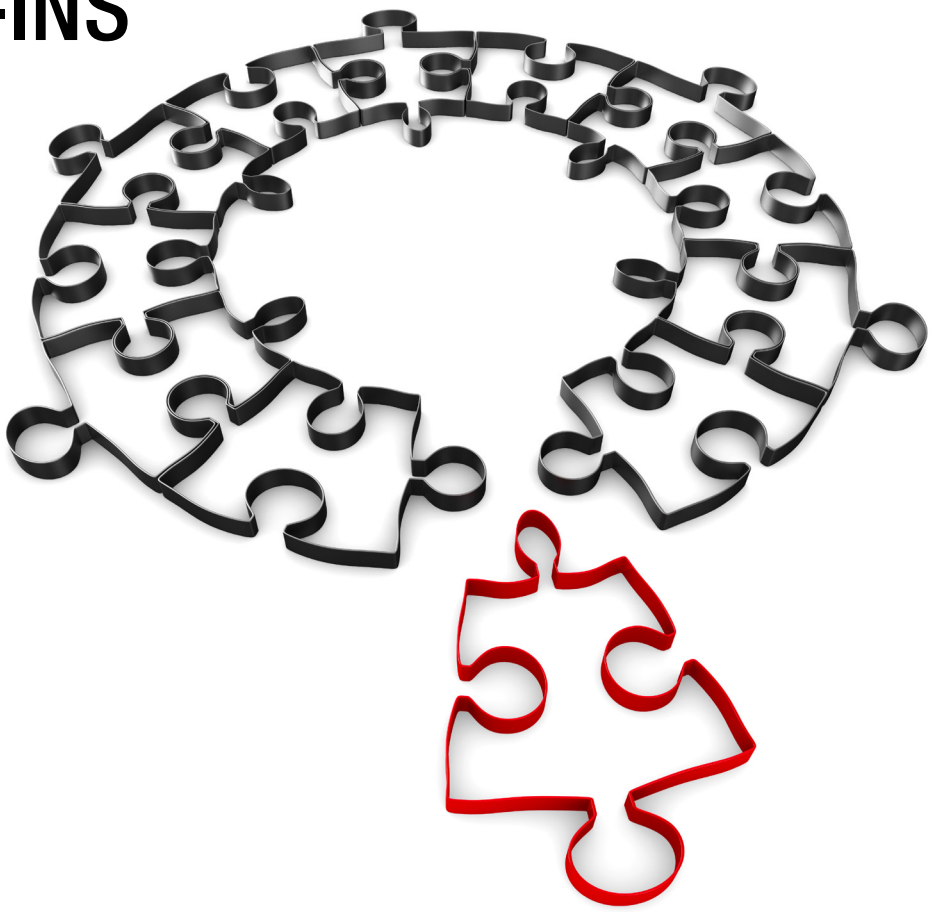
HAUFE.
Ausgabe 04/2017

ACCESS

IM UNTERNEHMEN

ACCESS-ADD-INS

Erweitern Sie Access mit praktischen .NET-Add-Ins (ab S. 2).



In diesem Heft:

E-MAILS EFFIZIENT ABRUFEN

Bringen Sie Outlook bei, nur noch in bestimmten Zeiten das Abrufen von E-Mails zu erlauben.

SEITE 33

WINDOWS EXPLORER IM FORMULAR

Erweitern Sie Ihre Anwendungen um einen Windows Explorer direkt im Formular.

SEITE 44

ZUGRIFF AUF ONENOTE-NOTZEN

Lesen Sie die Daten aus Ihren OneNote-Notizen per VBA in Access ein.

SEITE 66

Impressum

ISBN 978-3-8092-1496-0 ISSN: 1616-5535 Mat.-Nr. 01583-5102

Access im Unternehmen

Kommanditgesellschaft, Sitz Freiburg
Registergericht Freiburg, HRA 4408
Komplementäre: Haufe-Lexware Verwaltungs GmbH, Sitz Freiburg,
Registergericht Freiburg, HRB 5557; Martin Laqua
Geschäftsführung: Isabel Blank, Sandra Dittert, Markus Dränert, Jörg Frey, Birte Hackenjos, Markus Reithwiesner, Joachim Rotzinger, Dr. Carsten Thies

Beiratsvorsitzende: Andrea Haufe
Steuernummer: 06392/11008
Umsatzsteuer-Identifikationsnummer: DE 812398835

Redaktion: Dipl.-Ing. André Minhorst (Chefredakteur, V.i.S.d.P.),
Sabine Wißler (Redaktionsassistentin), Rita Klingenstein (sprachl. Lektorat), Sascha Trowitzsch (Fachlektorat)
Herausgeber: Dipl.-Ing. André Minhorst
Autoren: Dipl.-Ing. André Minhorst, Sascha Trowitzsch

Anschrift der Redaktion: Postfach 100121, 79120 Freiburg, Tel. 0761/898-0, Fax: 0761/898-3990,
E-Mail: computer@haufe.de, Internet: http://computer.haufe.de

Druck: Schätzl Druck & Medien GmbH & Co. KG, Donauwörth

Auslieferung und Vertretung für die Schweiz: H.R. Balmer AG, Neugasse 12, CH-6301 Zug, Tel. 041/711 4735, Fax: 041/711 0917

Access mit .NET-Unterstützung

Nennen wir das Kind beim Namen: Access wurde seit der Version 2010 nicht mehr weiterentwickelt, VBA seit Version 2000 nicht mehr. Das ist eine tolle Situation für alle, die mit Access im Kundenauftrag Lösungen entwickeln und einen ordentlichen Kundenstamm haben: Man muss keine Zeit in die Einarbeitung in neue Versionen oder Techniken investieren. Das hat Vor- und Nachteile.

Wer Anwendungen programmiert, die mit den vorhandenen Access-Techniken auskommen, muss sich keine Sorgen machen und kann in diesem Rahmen sehr effizient arbeiten. Eine Einarbeitung ist hier nur in den fachlichen Teil eines Projekts nötig, nicht in technische Neuerungen.

Wer mit alter Technik neue Errungenschaften nutzen möchte wie etwa den Zugriff auf Webservices, kommt in vielen Fällen noch gut zurecht – externe Bibliotheken wie etwa die XML-Bibliothek macht dies recht einfach. Liefert der Webservice die Daten hingegen im JSON-Format, steht man auf dem Schlauch – eine passende Bibliothek hierfür gibt es nicht. Hier ist man besser aufgehoben, wenn man mit moderneren Mitteln wie etwa mit Visual Studio arbeitet. Dort gibt es Bibliotheken für alle möglichen Einsatzzwecke. Der Zugriff auf ein JSON-Dokument ist damit ein Kinderspiel.

Zum Glück kann man beide verknüpfen. In dieser Ausgabe schauen wir uns dabei an, wie Sie die Office-Anwendungen durch Add-Ins erweitern, die mit Visual Studio erstellt werden. Das Thema **Add-Ins für Office** gibt es schon seit Jahren. Erfreulicherweise geht Microsoft hiermit anders um als mit Access: Vor einigen Jahren war es noch eine Qual, ein mit Visual Studio erstelltes Add-In unter einer Office-Anwendung zum Laufen zu bringen. Mit dem (noch) aktuellen Visual Studio 2015 ist dies kein Problem mehr – und in der Community-Edition ist diese Entwicklungsumgebung für kleine Unternehmen sogar kostenlos verfügbar. Es gibt passende Vorlagen für fast alle Office-Anwendungen und diese lassen sich sogar per Setup recht leicht weitergeben. Dummerweise gehört Access nicht zu den Zielanwendungen: Es gibt schlicht keine Vorlage dafür.

Aber das ist für uns von Access im Unternehmen natürlich kein Problem: Wir zeigen Ihnen, wie Sie etwa eine Vorlage für ein Excel-Add-In in eines für den Einsatz unter Access umwandeln (**Access-Add-In mit VSTO**, ab S. 2). Da Add-Ins meist über Ribbon-Einträge gestartet werden, liefert der Beitrag **COM-Add-In-Ribbons mit dem XML-Designer** ab S. 14 Informationen darüber, wie Sie das Add-In mit Ribbon-Schaltflächen ausstatten. Speziell für Access-Entwickler ist es natürlich interessant, von einem Add-In auf die Daten der Access-Datenbank zuzugreifen. Was dabei zu beachten ist, erfahren Sie unter dem Titel **Datenzugriff per VSTO-Add-In** ab S. 20. Damit haben Sie dann schon eine gute Ausgangsposition für das Erstellen eigener Add-Ins.

Damit Sie genug Zeit zum Entwickeln der Add-Ins haben, liefert der Beitrag **Effizienz-Hacks: Mailabruf erschweren** ab S. 33 eine Anleitung, wie Sie ein Add-In entwickeln, das unter Outlook das Zeitfenster zum Abrufen von E-Mails festlegt.

Sascha Trowitzsch steuert mit **Explorer Control** ab S. 44 eine umfassende Anleitung bei, wie man einem Access-Formular einen Windows Explorer unterjubelt. Und schließlich erfahren Sie im Beitrag **One Note und Access** ab S. 66 noch, wie Sie die Notizen aus One Note von Access aus einlesen können.

Und nun: Viel Spaß beim Lesen!

Ihr André Minhorst

Access-Add-In mit VSTO

Access-Add-Ins können Sie mit Access selbst erstellen oder aber auch mit Visual Studio, zum Beispiel in der kostenlosen Visual Studio 2015 Community Edition. Dieser Beitrag zeigt, wie Sie ein Access-Add-In mit Visual Studio erstellen und was dabei zu beachten ist. Dabei legen wir vor allem ein Augenmerk darauf, wie Sie dem Fehlen einer Vorlage zum Erstellen von Access-Add-Ins begegnen und ein Add-In für eine andere Office-Anwendung nutzen, um ein Access-Add-In zu programmieren.

Voraussetzungen

Die erste Voraussetzung ist das Vorhandensein von **Visual Studio Community Edition 2015**, welches Sie nach Eingabe des Namens als Suchbegriff bei Google schnell finden sollten. Dieses kommt aber in der Regel nicht direkt mit den Office Developer Tools. Diese finden Sie ebenfalls am besten, wenn Sie bei Google die Suchbegriffe Office Developer Tools eingeben. Ein guter Startpunkt ist zum Zeitpunkt der Veröffentlichung dieses Beitrags aber auch diese Adresse:

```
https://blogs.msdn.microsoft.com/visualstudio/2015/11/23/latest-microsoft-office-developer-tools-for-visual-studio-2015/
```

Sie müssen also zuerst Visual Studio 2015 Community Edition installieren und dann die aktuellsten Erweiterungen herunterladen. Danach sollten Sie, wenn Sie Visual Studio starten und ein neues Projekt anlegen, die Vorlagen wie in Bild 1 finden.

Add-In für Access

Wer weiß, dass die Elemente in alphabetischer Reihenfolge aufgelistet werden, und sieht, dass **Excel 2013 und 2016 VSTO-Add-In** der erste Eintrag ist, erkennt schnell: Es gibt keine Vorlage für Add-Ins, die mit Access arbeiten. Aber das ist kein Problem: Wir starten einfach mit einem anderen Add-In-Typ, beispielsweise für Microsoft Excel, und passen das daraus generierte Projekt dann für den Einsatz in Access an.

Start mit Excel statt mit Access

Sie erstellen also zunächst ein neues Projekt auf Basis der Vorlage **Visual C#** (oder **Visual Basic**)**|Office/SharePoint|VSTO-Add-Ins|Excel 2013 und 2016 VSTO-Add-In**. Geben Sie dabei den Projektnamen an, in unserem Beispiel schlicht **AccessAddIn**.

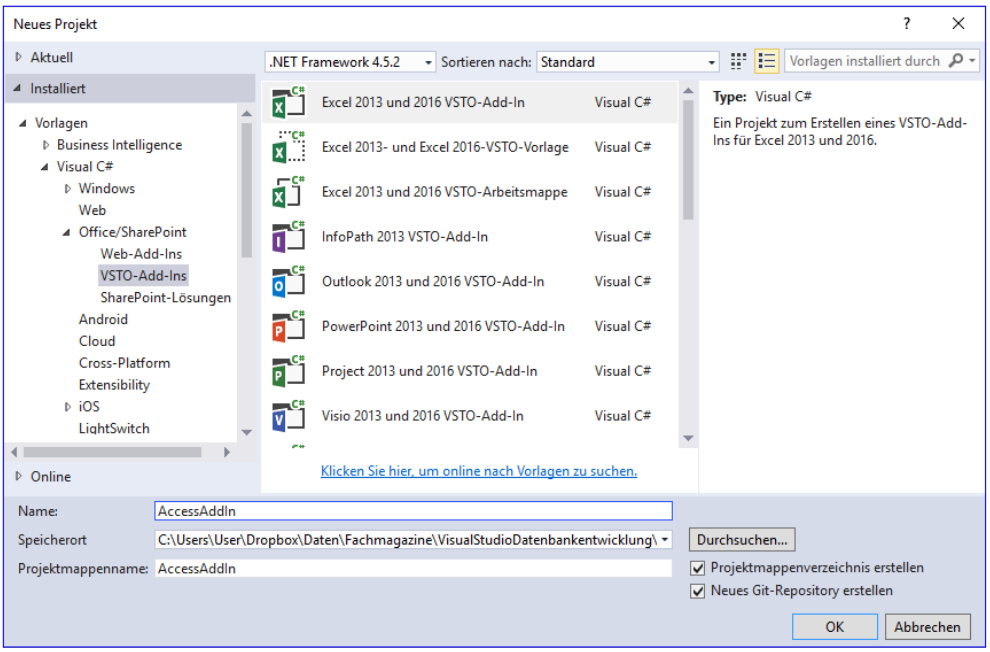


Bild 1: Keine Vorlage für ein Access-Add-In

Nach einem Klick auf **OK** erstellt Visual Studio das Projekt.

Add-In in Excel testen

Wir wollen das Add-In, das wir ja nun erstellt, aber noch nicht für den Einsatz in Access angepasst haben, zunächst einmal auf seine Tauglichkeit unter Excel testen. Dazu klicken Sie in Visual Studio einfach auf **F5** und führen das Add-In so aus. Das Resultat ist ernüchternd, denn wir erhalten direkt eine Fehlermeldung wie in Bild 2.

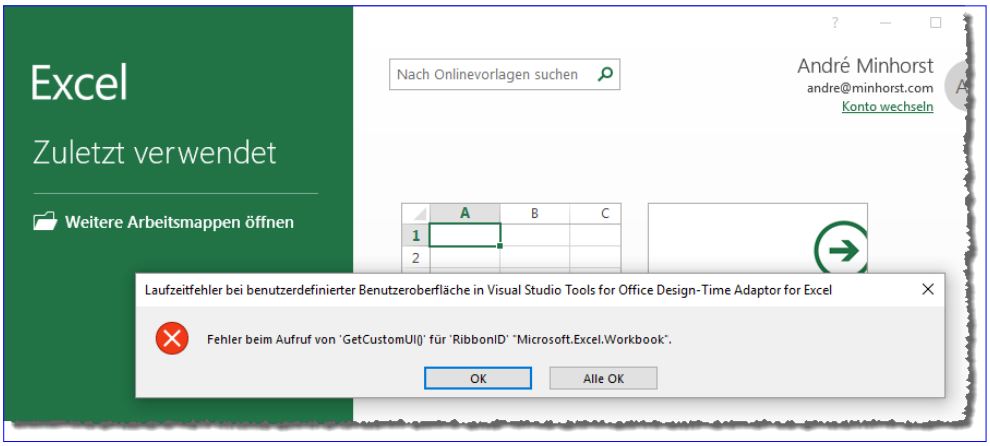


Bild 2: Fehler beim Start des Excel-Add-Ins

Um dieses erste Problem zu beheben, schließen Sie Excel und öffnen es das nächste Mal als Administrator. Dazu geben Sie Excel in das Suchen-Feld von Windows ein, klicken mit der rechten Maustaste auf den Eintrag **Excel 2016** und wählen aus dem Kontextmenü den Eintrag **Als Administrator ausführen** aus (s. Bild 3).

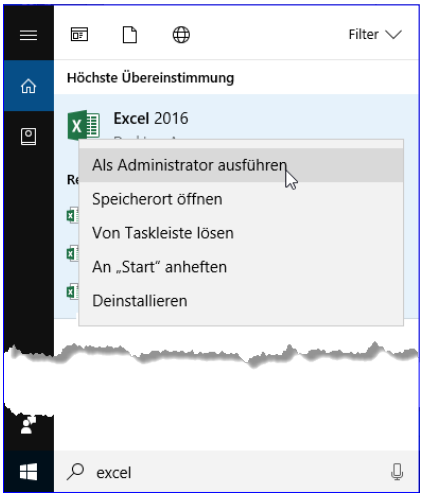


Bild 3: Excel als Administrator starten

Nach dem Öffnen von Excel und dem Erstellen oder Laden einer Datei öffnen Sie über den Backstage-Bereich die Excel-Optionen und wechseln dort zum Bereich **Menüband anpassen**. Hier aktivieren Sie nun die Hauptregisterkarte **Entwicklertools** (s. Bild 4).

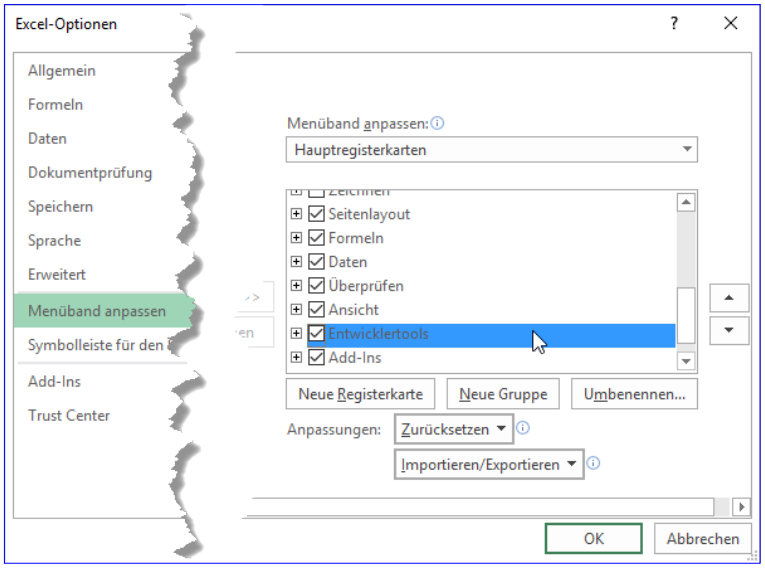


Bild 4: Entwicklertools aktivieren

Dies fügt dem Ribbon von Excel einen neuen Reiter namens Entwicklertools hinzu. In diesem finden Sie eine Schaltfläche namens **COM-Add-Ins**. Wenn Sie diese anklicken, öffnet sich die Liste der aktuell aktiverten Add-Ins. Ganz oben sehen Sie bereits den Eintrag **Access-AddIn** für unser soeben erstelltes Add-In. Ganz unten sollten Sie den Eintrag **Visual Studio Tools for Office Design-Time Adaptor for Excel** sehen. Diesen Eintrag müssen Sie deaktivieren, um den Fehler beim Aufruf von **GetCustomUI()** zu beheben (s. Bild 5).

Bevor wir uns nun an die Arbeit machen, das Add-In für den Einsatz mit Access umzurüsten, wollen wir zumindest noch eine Ribbon-Schalt-

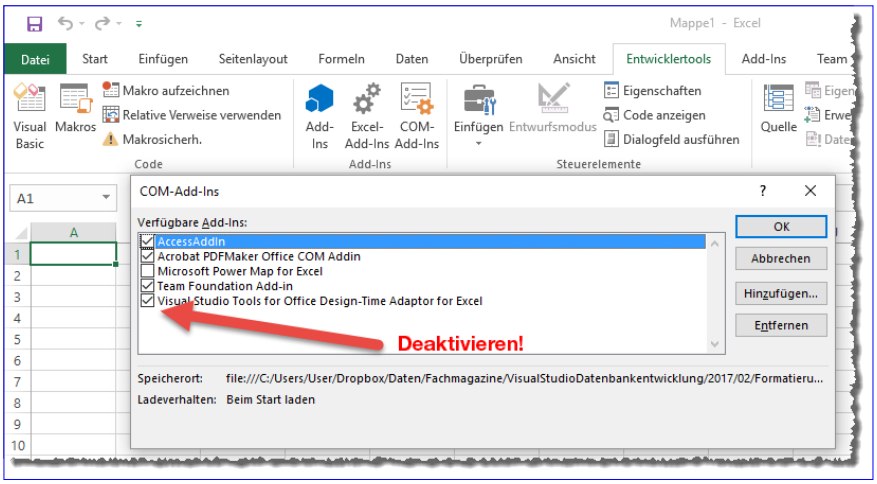


Bild 5: Fehler durch **GetCustomUI** beheben

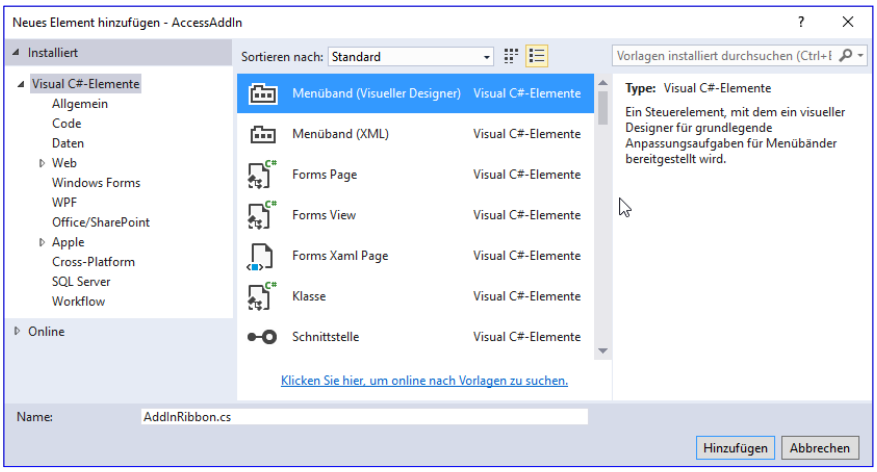


Bild 6: Hinzufügen eines Ribbons

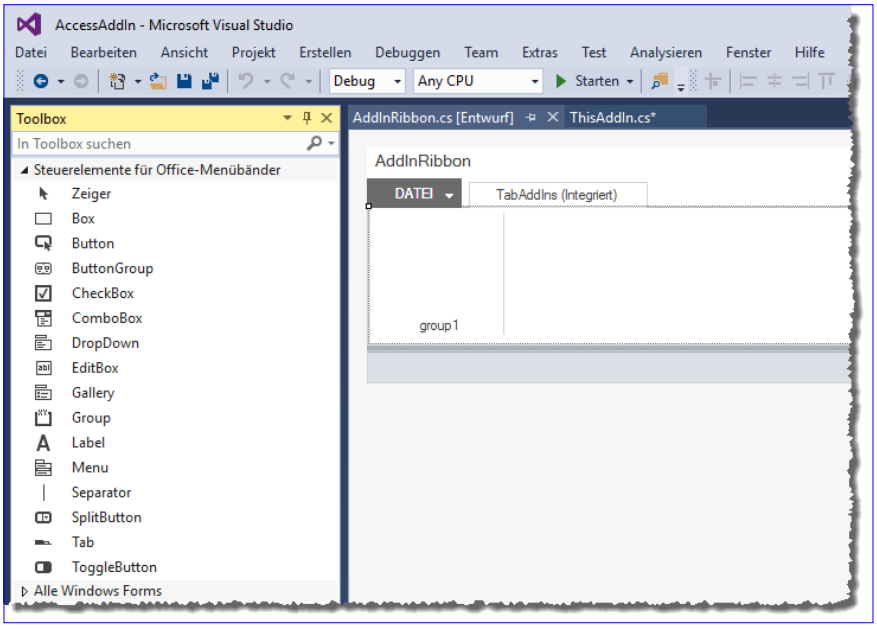


Bild 7: Bearbeiten des Ribbbons

fläche hinzufügen, welche ein Meldungsfenster anzeigt. Damit stellen wir sicher, dass das Add-In grundsätzlich funktioniert. Es wäre schade, wenn wir nun erst die ganzen Schritte für die Anpassung als Access-Add-In durchführen und dann, wenn es nicht funktionieren sollte, nicht wissen, wo der Fehler zu suchen ist.

Sobald Sie Excel nun schließen und erneut starten, erscheint die Fehlermeldung nicht mehr.

Ribbon hinzufügen

Nun wollen wir den Code zur Erstellung eines Ribbons zu unserem Add-In hinzufügen. Dazu erstellen Sie in Visual Studio eine neue Klasse, indem Sie den Kontextmenü-Eintrag des Projekts **AccessAddIn** aus dem Projektmappen-Explorer namens **Hinzufügen|Neues Element...** anklicken. Es erscheint der Dialog aus Bild 6, mit dem Sie ein Element des Typs **Menüband (Visueller Designer)** namens **AddInRibbon.cs** hinzufügen.

Es erscheint die Ansicht aus Bild 7, mit der Sie die Ribbon-Steuerelemente aus der Toolbox an die gewünschten Stellen im Ribbon einfügen können. Nun bedarf es einiger Erläuterungen, was es mit dem Element **TabAddIns (Integriert)** auf sich hat. Eines steht jedoch fest: Durch das einfache Hinzufügen eines Ribbons ändert

sich noch nichts, beim nächsten Debuggen/Starten des Add-Ins per **F5** öffnet sich zwar Excel, aber es erscheinen keine zusätzlichen Elemente im Ribbon.

Schaltfläche zum Ribbon hinzufügen

Wir fügen nun zunächst eine Schaltfläche zum Ribbon hinzu, indem wir ein **Button**-Element aus der Toolbox in den Bereich **group1** ziehen. Die Beschriftung beider Elemente, also der Gruppe und der Schaltfläche, passen wir nun an.

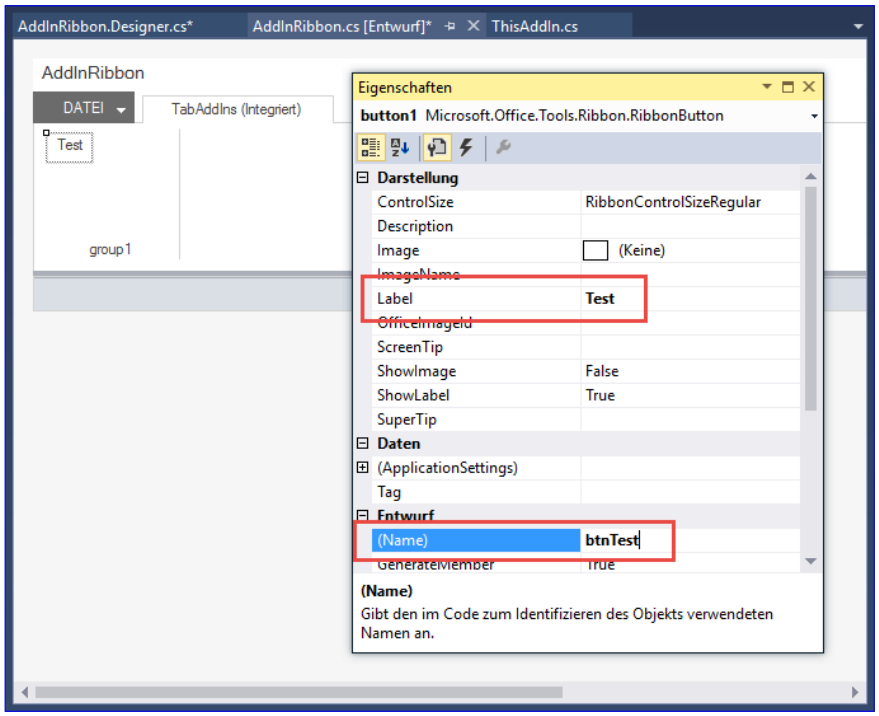


Bild 8: Bezeichnung der Schaltfläche ändern

Dazu klicken Sie auf das Element, in Bild 8 zum Beispiel auf das **Button**-Element, und stellen im Eigenschaften-Bereich die Eigenschaft **Label** auf den gewünschten Wert ein, in diesem Fall **Test**. Im gleichen Zug können Sie die Eigenschaft **Name** auf den Wert **btnTest** ändern. Für das **Group**-Element stellen Sie **Label** auf **Beispielgruppe** und **Name** auf **grpBeispiel** ein.

Nun fügen wir noch eine Ereignisprozedur zur Schaltfläche hinzu. Das Klassenmodul mit dem Code für das Ribbon-Objekt öffnen Sie, indem Sie aus dem Kontextmenü der Schaltfläche den Eintrag **Code anzeigen** auswählen (s. Bild 9). Dieser enthält jedoch noch keine Ereignisprozedur für die Schaltfläche **btnTest**. Diese legen Sie am einfachsten an, indem Sie in der Entwurfsansicht doppelt auf die Schaltfläche klicken.

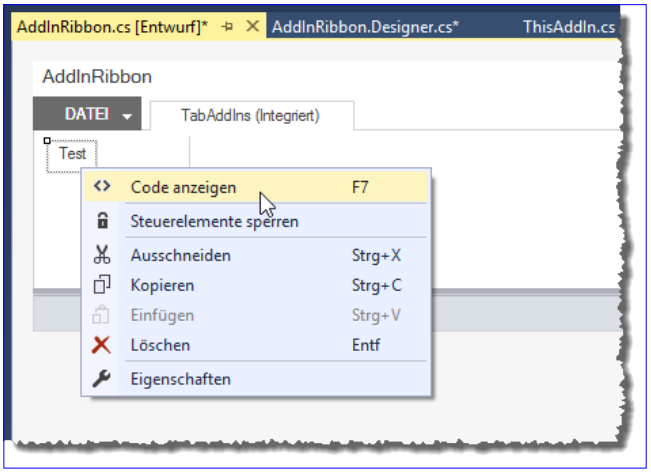


Bild 9: Anzeigen der Ereignisprozedur der Schaltfläche

Sollte nun nicht automatisch die Klasse hinter dem Ribbon-Element erscheinen, können Sie diese wiederum über den Kontextmenü-Befehl **Code anzeigen** oder die Taste **F7** öffnen. Dieses Klassenmodul enthält nun eine Methode namens **btnText_Click**. Wir wollen dieser ein einfaches Meldungsfenster hinzufügen, wozu zwei Schritte nötig sind. Als Erstes fügen Sie der Klasse oben mit der **using**-Anweisung einen Verweis auf das Namespace **System.Windows.Forms** hinzu:

```
using System.Windows.Forms;
```

Die C#-Methode füllen wir dann mit einem einfachen Aufruf der **Show**-Methode der **MessageBox**-Klasse (s. Bild 10):

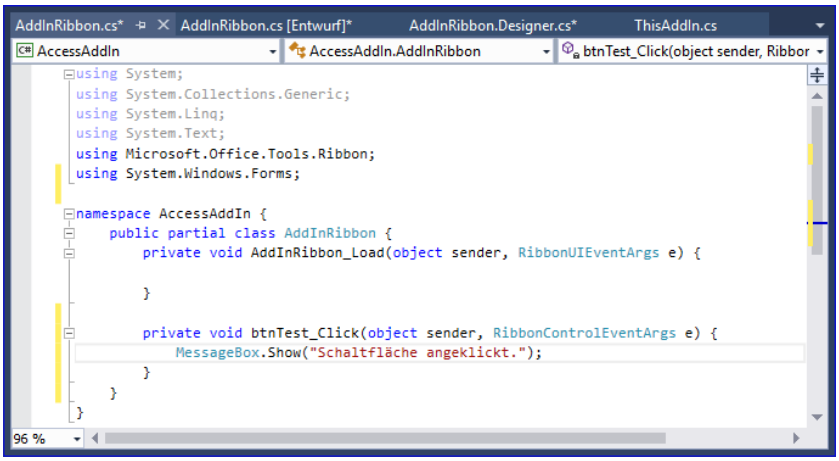


Bild 10: Hinzufügen einer **MessageBox** zur Schaltfläche

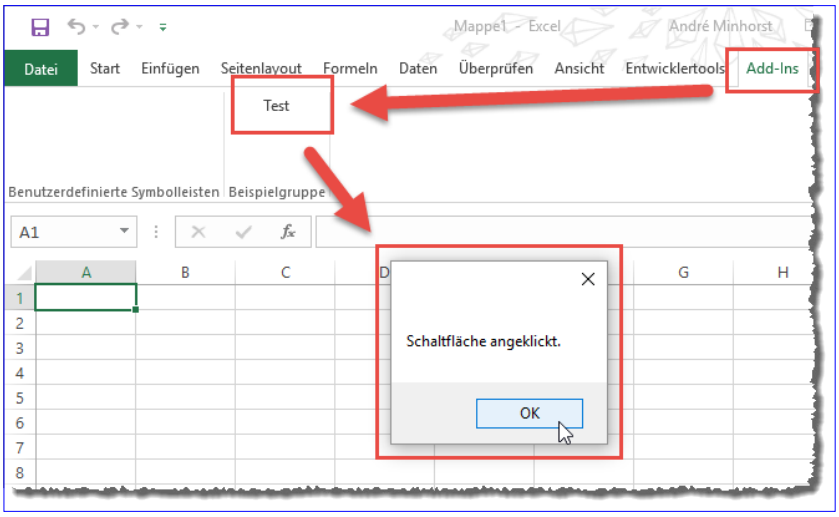


Bild 11: Die neu hinzugefügte Add-In-Schaltfläche

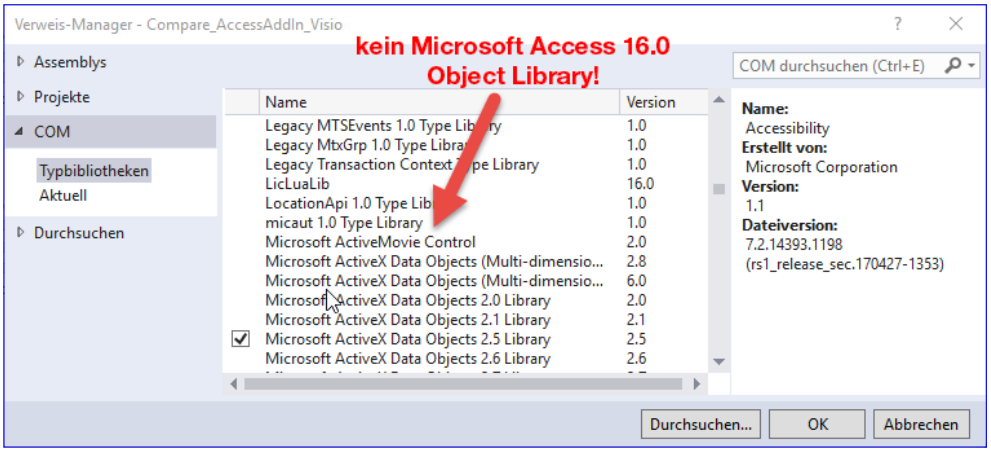


Bild 12: Der Verweis auf **Microsoft Access 16.0 Object Library** fehlt.

```
private void btnTest_Click(object sender, 7
                RibbonControlEventArgs e) {
    MessageBox.Show("Schaltfläche 7
                                angeklickt.");
}
```

Wenn Sie nun das Add-In per **F5** starten/debuggen, wird Excel geöffnet, aber auf den ersten Blick ist kein neuer Eintrag zu erkennen. Dieser erscheint erst, wenn Sie nun auf den Tab **Add-Ins** klicken. Dort finden Sie dann neben der Gruppe **Benutzerdefinierte Symbolleisten** unsere selbst definierte Gruppe namens **Beispielgruppe** und die Schaltfläche **Test**. Klicken Sie diese an, taucht auch prompt die von uns programmierte Meldung auf (s. Bild 11).

Wechsel zu Access

Nun gehen wir einen Schritt weiter und wollen das Add-In so anpassen, dass es nicht unter Excel, sondern unter Access geöffnet wird. Dazu sind einige Änderungen nötig, die wir teilweise im Text-Editor Ihrer Wahl durchführen müssen, weil die entsprechenden Dateien nicht in Visual Studio geöffnet werden können.

Schritt 1: Verweis auf die Access-Bibliothek

Die erste Änderung ist das Hinzufügen eines Verweises auf die Access-Bibliothek. Dazu betätigen Sie den Menübefehl **Projekt|Verweis hinzufügen...**, der den **Verweis-Manager** öffnet. Aus unerfindlichen Gründen finden Sie den Verweis auf

COM-Add-In-Ribbons mit dem XML-Designer

Wenn Sie in Visual Studio ein COM-Add-In für Access erstellen (wie im Beitrag »Access-Add-In mit VSTO«), können Sie zwei Möglichkeiten nutzen, um Ribbons für das Add-In hinzuzufügen. Der erste Weg nutzt einen grafischen Editor und wird bereits im genannten Beitrag beschrieben. Allerdings erlaubt dieser nicht, alle verfügbaren Elemente und Techniken zu nutzen. Genau genommen können Sie damit nur Tabs, Groups und die darin enthaltenen Steuerelemente hinzufügen. Wenn Sie, wie Sie es von Access gewohnt sind, in die Vollen gehen wollen und etwa auch das Backstage anpassen oder eingebaute Ribbon-Schaltflächen mit neuen Funktionen versehen wollen, finden Sie im vorliegenden Artikel die richtigen Techniken.

Voraussetzungen

Um die in diesem Beitrag beschriebenen Techniken verwenden zu können, benötigen Sie ein COM-Add-In, das mit Access gestartet wird. Die Grundlagen dazu finden Sie im Beitrag **Access-Add-In mit VSTO** (**www.access-im-unternehmen.de/1092**). Der Beitrag beschreibt die Vorgehensweise für C#, Sie finden im Download aber auch ein für den Einsatz mit Access angepasstes COM-Add-In-Projekt. Auf diesem setzt dieser Beitrag auf. Außerdem verwenden wir Visual Studio 2015 in der Community Edition, die für einzelne Entwickler kostenlos verfügbar ist.

Hinweis

Alle wichtigen Grundlagen und Informationen sowie Listen der Callback-Funktionen für den Backstage-Bereich des Ribbons finden Sie auf der Internetseite mit dem Titel Einführung in die Office 2010-Backstage-Ansicht für Entwickler (aktueller LInk: **https://msdn.microsoft.com/de-de/library/ee691833(office.14).aspx**).

Die Grundlagen für den Ribbon-Teil finden Sie unter dem Titel **Anpassen der Multifunktionsleisten-Benutzeroberfläche von Office (2007) für Entwickler - Teil 1** und den folgenden Teilen (aktueller Link: **https://msdn.microsoft.com/de-de/library/aa338202.aspx**).

Ribbon zum Add-In hinzufügen

Wie bereits erwähnt, gibt es zwei Möglichkeiten, ein Ribbon zu einem COM-Add-In auf Basis der VSTO-Vorlagen hinzuzufügen. Der erste Eintrag namens **Menüband (Visueller Designer)** liefert einen Designer, mit dem Sie das Ribbon per Drag and Drop aus der Toolbox zusammenstellen können. Dafür stehen aber nur die wichtigsten Elemente zur Verfügung. Wer bereits unter Access Ribbons definiert hat, wird hier einiges vermissen. In diesem Fall wählt man besser den Eintrag **Menüband (XML)** und

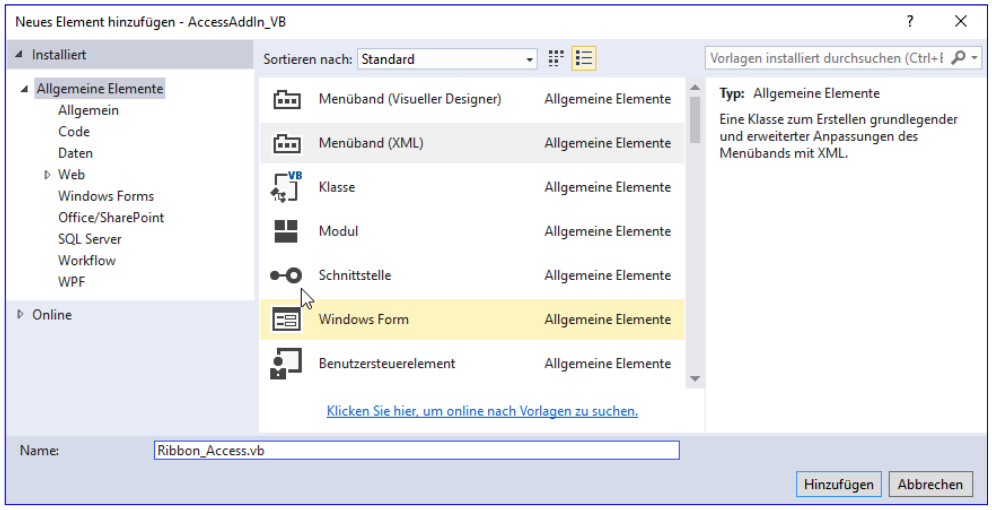


Bild 1: Hinzufügen eines Ribbons mit der Vorlage **Menüband (XML)**

gibt beispielsweise den Namen **Ribbon_Access.vb** an (s. Bild 1). Dies fügt zwei Klassen namens **Ribbon_Access.vb** und **Ribbon_Access.xml** zum Projekt hinzu. Damit die in der Datei **Ribbon_Access.xml** enthaltene Ribbon-Definition auf die Host-Anwendung, in diesem Fall Access, angewendet wird, kopieren Sie die noch auskommentierte Methode **CreateRibbonExtensibilityObject** aus dem Modul **Ribbon_Access.vb** in das Klassenmodul **ThisAddIn.vb** und entfernen die Kommentarzeichen:

```
Protected Overrides Function _
        CreateRibbonExtensibilityObject() _
        As Microsoft.Office.Core.IRibbonExtensibility
    Return New Ribbon_Access()
End Function
```

Diese Methode wird beim Starten des Add-Ins automatisch ausgeführt.

Die XML-Definition des Ribbons sieht standardmäßig wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" onLoad="Ribbon_Load">
  <ribbon>
    <tabs>
      <tab idMso="TabAddIns">
        <group id="MyGroup"
               label="My Group">
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Wenn Sie das Add-In nun starten, erscheinen allerdings keine neuen Elemente im Ribbon von Access. Der Grund ist einfach: Die automatisch erstellte Ribbon-Definition aus der Datei **Ribbon_Access.xml** versucht, eine Gruppe unterhalb eines **Tab**-Elements mit der **idMso** mit dem Wert **TabAddIns** anzulegen. Warum wird dieses Tab dann von der Vorlage als Ausgangspunkt für die Ribbon-Definition festgelegt? Ganz einfach: Weil es offiziell gar keine VSTO-Vorlage für Access-Add-Ins gibt, haben wir ja, wie im Beitrag **Access-Add-In mit VSTO** beschrieben, ein Excel-Add-In in ein Access-Add-In umgewandelt. Und unter Excel gibt es sehr wohl ein Ribbon-Tab namens **TabAddIns**. Wenn wir also einfach das eingebaute **group**-Element **TabAddIns** durch ein benutzerdefiniertes

```
<?xml version="1.0" encoding="UTF-8"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" onLoad="Ribbon_Load">
  <ribbon>
    <tabs>
      <tab id="tabAddIn" label="COM-Add-In">
        <group id="grpAnwendung" label="Anwendung">
          <button id="btnAnwendungsbefehl" label="Anwendungsbefehl" onAction="onAction" />
        </group>
        <group id="grpDatenbank" label="Datenbank">
          <button id="btnDatenbankbefehl" label="Datenbankbefehl" onAction="onAction"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Listing 1: Definition zweier Ribbon-Gruppen mit je einer Schaltfläche

Element ersetzen, sollte es funktionieren.

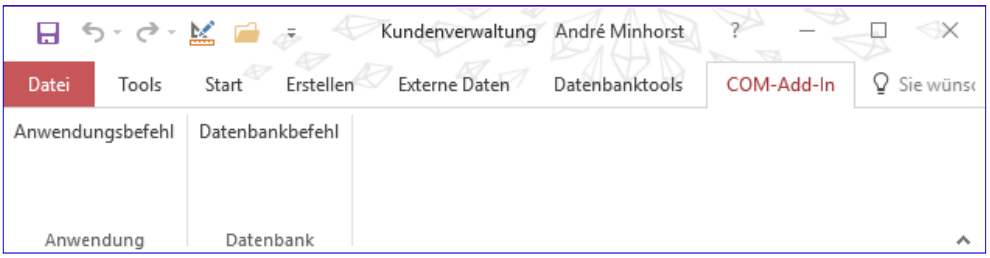


Bild 2: Die neuen Ribbon-Elemente im Tab **COM-Add-In**

Benutzerdefiniertes Tab mit Gruppe und Schaltfläche hinzufügen

Ein eingebautes **tab**-Element verwendet die für dieses Element festgelegte **idMso** (Übersicht der idMsos für alle Elemente siehe Datei **AccessControl.xlsx** aus dem Download von der Seite **http://go.microsoft.com/fwlink/?LinkID=181052**), um festzulegen, dass die darunter abgebildeten **group**-Elemente unterhalb des eingebauten Elements mit der angegebenen **idMso** angelegt werden sollen. Gegebenenfalls möchten Sie auch noch Elemente in eine eingebaute Gruppe einfügen, dann würden Sie auch für die Gruppe die entsprechende **idMso** angeben.

Wir wollen aber ein neues **tab**-Element erstellen, weshalb wir nicht die Eigenschaft **idMso** angeben, sondern mit **id** den Namen unseres benutzerdefinierten Elements. Dieses soll **tabAddIn** heißen. Darunter wollen wir zwei Gruppen anlegen, die jeweils eine Schaltfläche enthalten. Die erste Gruppe soll Befehle erhalten, die sich auf die Access-Anwendung selbst beziehen, die zweite Befehle, die sich auf die aktuell geöffnete Datenbank beziehen. Für den zweiten Teil arbeiten wir im Beitrag **Datenzugriff per VSTO-Add-In** (**www.access-im-unternehmen.de/1094**) noch eine Lösung heraus, mit der wir dafür sorgen können, dass die Elemente der zweiten Gruppe nur aktiviert sind, wenn Access gerade eine Datenbankanwendung geladen hat. Die Definition unseres Ribbons sieht somit wie in Listing 1 aus.

Wenn wir das Add-In nun starten, sollten die neuen Elemente in Access wie in Bild 2 erscheinen.

Prozedur für die Buttons anlegen

Wir legen nun zunächst eine Prozedur an, die durch einen Klick auf eine der Schaltflächen ausgelöst wird. Wie unter Access haben wir in der XML-Definition des Ribbons mit dem Attribut **onAction** den Namen der aufzurufenden Methode angegeben, in diesem Fall **onAction**.

Die Signatur der verschiedenen Callback-Funktionen, die hier nicht beschrieben werden, finden Sie in den weiter oben unter Hinweis angegebenen Quellen. Wir wollen zunächst einfach nur den Namen des aufrufenden Steuerelements in einem Meldungsfenster ausgeben. Dazu fügen Sie dem Klassenmodul **Ribbon_Access.vb** zunächst den Verweis auf folgenden Namespace hinzu:

```
Imports System.Windows.Forms
```

Danach legen wir in der Klasse **Ribbon_Access.vb** in dem Bereich, der mit **#Region "Menübandrückrufe"** gekennzeichnet ist, die folgende Methode an:

```
Sub OnAction(control As Office.IRibbonControl)
    MessageBox.Show(control.Id.ToString())
End Sub
```

Wenn Sie das Add-In nun zum Debuggen starten und auf eine der beiden Schaltflächen klicken, erscheint die entsprechende **MessageBox**. Wir wollen nun eine **Select Case**-Bedingung einbauen, die prüft, welche Schaltfläche die Methode ausgelöst hat, und davon abhängig entweder den Text **Anwendungsbefehl** oder **Datenbankbefehl** ausgeben:

```
Public Sub onAction(ctl As Office.IRibbonControl)
    Select Case ctl.Id
        Case "btnAnwendungsbefehl"
```

Datenzugriff per VSTO-Add-In

Im Beitrag »Access-Add-In mit VSTO« haben Sie erfahren, wie Sie aus einer der bestehenden Vorlagen für Outlook-, Word- oder Excel-Add-Ins ein Access-Add-In zaubern. In diesem Beitrag nun wollen wir uns ansehen, wie Sie von einem solchen Add-In auf die Objekte der Datenbank und auch auf die darin enthaltenen Daten zugreifen können. Damit erhalten Sie die Grundlage, viele interessante und praktische Add-Ins für Access zu bauen.

Wenn Sie ein Add-In für eine Access-Anwendung programmieren, wollen Sie damit nicht irgendwelche Funktionen ausführen, sondern solche, welche die Funktionen von Access oder die Daten der aktuellen Datenbank nutzen. Dazu müssen Sie innerhalb des Visual Studio-Projekts auf die Access-Instanz zugreifen können, von der aus das Add-In gestartet wurde. Darüber können Sie dann die Daten der aktuell geladenen Datenbank lesen.

Wie aber kommen wir an die Access-Instanz? Das ist, im Vergleich zu früheren Vorgehensweisen in Zusammenhang mit COM-Add-Ins für Office-Anwendungen, wesentlich leichter geworden. Genau genommen stellt die Klasse **ThisAddIn.Designer.vb** dieses Objekt über die Variable **Application** zur Verfügung (s. Bild 1). Weiter unten finden Sie dann in der gleichen Klasse die Methode Initialize, welche die Application-Eigenschaft mit einem Verweis auf die als Host verwendete Datei füllt.

Dies ist der Hintergrund, für uns ist aber vielmehr wichtig, dass wir über **Application** jederzeit die als Host dienende Access-Anwendung referenzieren können. Dies schauen wir uns an einem einfachen Beispiel an, wobei wir das im Beitrag **Access-Add-In mit VSTO** (**www.access-im-unternehmen.de/1092**) erstellte Visual Basic-Add-In als Grundlage verwenden, das Sie auch im Download zum vorliegenden Beitrag finden.

Hier fügen wir der Methode **ThisAddIn_Startup** im Klassenmodul **ThisAddIn.vb** einfach eine Anweisung hinzu, welche den Namen der mit dem **Application**-Objekt gelieferten Anwendung liefert:

```
Private Sub ThisAddIn_Startup() Handles Me.Startup
    MessageBox.Show("Application.Name: " + Application.Name)
End Sub
```

Dies zeigt beim Öffnen von Access nach dem Starten des Add-Ins (und später auch beim Öffnen ohne vorheriges Starten des Add-Ins, das ja dann weiterhin über die Registry aufgerufen wird) eine Meldung mit dem Namen **Microsoft Access** an. Wer früher einmal COM-Add-Ins für Access programmiert hat, weiß, dass damit mehr benutzerdefinierter Aufwand verbunden war.

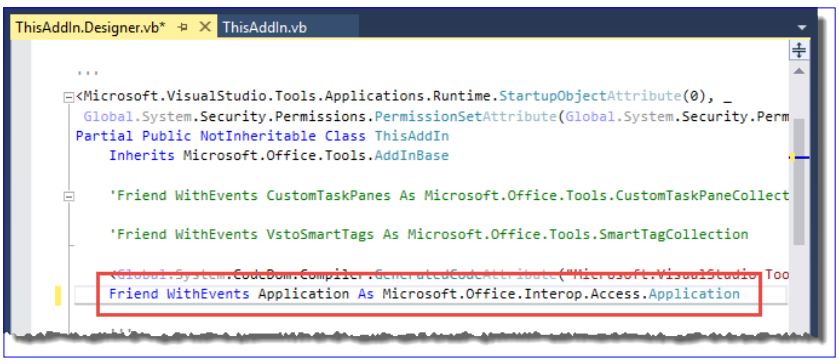


Bild 1: Variable zur Bereitstellung des **Application**-Objekts

Geöffnete Datenbank referenzieren

Etwas komplizierter wird es, wenn Sie vom Add-In aus die geöffnete Datenbank referenzieren möchten. Das Problem dabei ist, dass das Add-In ja bereits beim Starten von Access geladen wird und über die Objektvariable **Application**

verfügbar ist. Zu diesem Zeitpunkt ist aber noch keine Access-Datenbank in Access geöffnet. Dies geschieht erst später. Wir müssen also einen Weg finden, um zu erkennen, wann Access eine Datenbankdatei öffnet, und diese dann referenzieren. Wir probieren zuerst einmal aus, was geschieht, wenn wir die Methode **ThisAddIn_Startup** wie folgt ausstatten:

```
Private Sub ThisAddIn_Startup() Handles Me.Startup
    If Application.CurrentDb Is Nothing Then
        MessageBox.Show("DB ist nicht geladen.")
    Else
        MessageBox.Show("DB: " + Application.CurrentDb.Name)
    End If
End Sub
```

Damit das **MessageBox**-Objekt verfügbar ist, fügen Sie noch den folgenden Verweis auf den Namespace **System.Windows.Forms** hinzu:

```
Imports System.Windows.Forms
```

Die Prozedur prüft nun mit **CurrentDb Is Nothing**, ob es eine aktuelle Datenbankdatei gibt, und liefert eine entsprechende Meldung. Das Ergebnis: Wenn man Access ohne Datenbank öffnet, hat **CurrentDb** den Wert **Nothing**, wenn man es direkt mit einer Datenbank startet, enthält **CurrentDb** einen Verweis auf die geladene Datenbank. Dann gibt die **MessageBox.Show**-Methode den Pfad zur geladenen Datenbank aus. Wenn wir allerdings Access starten und erst dann eine Datenbank öffnen, erkennt das Add-In zwar beim Starten von Access, dass noch keine Datenbank geöffnet ist, aber wenn wir dann nachträglich eine Datenbank öffnen, löst dies kein Ereignis mehr aus, aufgrund dessen wir erkennen könnten, dass eine Datenbank geöffnet wurde.

Warum aber müssen wir so genau wissen, ob gerade eine Datenbank geladen ist oder nicht? Ganz einfach: Wir wollen ja beispielsweise über das Ribbon Funktionen anbieten, mit denen der Benutzer auf die Objekte oder Daten der aktuellen Datenbank zugreifen kann. Wir könnten zwar einfach beim Betätigen der entsprechenden Elemente prüfen, ob aktuell eine Datenbank geöffnet ist, und gegebenenfalls eine Meldung ausgeben, dass die Funktion nur bei geöffneter Datenbank zur Verfügung steht. Aber schicker wäre es natürlich, wenn solche Schaltflächen deaktiviert sind, wenn die Funktionen nicht zur Verfügung stehen.

Timer mit Datenbankerkennung

Wir wollen nun einen Timer zum Add-In hinzufügen, der in jeder Sekunde einmal prüft, ob eine Datenbank geöffnet ist. Wenn eine Datenbank geöffnet ist, soll ein Meldungsfenster erscheinen und den Pfad zur geöffneten Datenbank anzeigen. Dazu fügen wir der Klasse **ThisAddIn.vb** folgenden Verweis auf den Namespace **SystemThreading.Tasks** hinzu:

```
Imports System.Threading.Tasks
```

Wir benötigen eine **Boolean**-Variable namens **Loaded**, welche den Zustand speichert, ob eine Datenbank geladen ist oder nicht:

```
Private Loaded As Boolean
```

Schließlich erweitern wir die Methode **ThisAddIn_Startup** so, dass diese einen Timer auf Basis der gleichnamigen Klasse erstellt, der alle 1.000 Millisekunden aufgerufen werden soll. Für diese Klasse legen wir einen Handler an, der durch das Ereignis **Elapsed** des Timers ausgelöst wird und die Methode **HandleTimer** aufruft. Dann starten wir den Timer mit der **Start**-Methode:

```
Private Sub ThisAddIn_Startup() Handles Me.Startup
    Dim timer As System.Timers.Timer
    timer = New System.Timers.Timer(1000)
    AddHandler timer.Elapsed, AddressOf HandleTimer
    With timer
        .Start()
    End With
End Sub
```

Die als Ereignishandler festgelegte Prozedur **HandleTimer** sieht wie in Listing 1 aus. Für VBA-erprobte Entwickler sieht der Aufbau etwas gewöhnungsbedürftig aus. Es handelt sich dabei um eine sogenannte asynchron laufende Methode, die mehrfach aufgerufen werden kann, auch wenn die aktuelle Instanz noch nicht abgearbeitet ist. So können wir die Methode tatsächlich jede Sekunde aufrufen, auch wenn der Benutzer noch nicht die durch den vorherigen Aufruf angezeigte **MessageBox** geschlossen hat. Das kann dann schnell unübersichtlich werden, wenn man nicht sauber codiert hat und plötzlich eine MessageBox nach der anderen auf dem Bildschirm erscheint. Für den Fall, dass Ihnen das einmal geschieht, können Sie schnell zu Visual Studio wechseln und dort **Umschalt + F5** klicken – dann ist der Spuk vorbei. Falls Sie das Add-In gerade nicht im Kontext des Debuggens von Visual Studio aus gestartet haben, müssen Sie wohl den Task-Manager bemühen.

Nun zu der Methode: Sie enthält innerhalb der Anweisung **Await Task.Run** eine weitere Methode, die bei jedem Aufruf ausgeführt wird. Diese Methode prüft, ob eine Datenbank geladen ist (**Application.CurrentDb Is Nothing**). Ist das nicht der Fall, prüft sie den Wert der Variablen **Loaded** und stellt diesen im Falle des Wertes **True** auf **False** ein. Beim Öffnen von Access ohne Datenbank geschieht also nichts weiter – **CurrentDb** ist **Nothing** und **Loaded** hat den Wert **False**.

Interessant wird es, wenn eine Datenbank geöffnet wird. Dann ist **CurrentDb** nicht mehr **Nothing** und der **Else**-Teil der äußeren **If...Then**-Bedingung wird aufgerufen. **Loaded** hat bis dahin den Wert **False** und wird auf **True** eingestellt. Außerdem gibt die Methode als Zeichen, dass sie erkannt hat, dass eine Datenbank geladen wurde, den Namen der Datenbank per MessageBox aus. Beim nächsten Durchgang wird dann, weil **CurrentDb** nicht **Nothing** ist, wieder der **Else**-Teil der äußeren Bedingung angesteuert. Diesmal hat **Loaded** aber den Wert **False**, weshalb die **MessageBox** nicht erneut angezeigt wird.

Was machen wir nun mit dieser Methode, die nach dem Öffnen einer Datenbank eine MessageBox anzeigt? Wir können zum Beispiel statt des Öffnens der MessageBox Code ausführen, der dafür sorgt, dass die gewünschten Ribbon-Elemente, die wir für die Bearbeitung der Datenbank hinzufügen, aktiviert oder deaktiviert werden. Das schauen wir uns einmal an einer einzelnen Ribbon-Schaltfläche an.

```
Private Async Sub HandleTimer(sender As Object, e As EventArgs)
    Await Task.Run(
        Sub()
            If Application.CurrentDb Is Nothing Then
                If Loaded = True Then
                    Loaded = False
                End If
            Else
                If Loaded = False Then
                    Loaded = True
                    MessageBox.Show("Geladen: " + Application.CurrentDb.Name + " " + Loaded.ToString())
                End If
            End If
        End Sub
    )
End Sub
```

Listing 1: Ereignisprozedur, die nach Ablauf des Timers ausgelöst wird

Ribbon hinzufügen

Um eine Ribbon-Definition hinzuzufügen, die alle Elemente des Access-Ribbons abbilden kann, können wir leider nicht das Element **Menüband (Visueller Designer)** des Dialogs **Neues Element hinzufügen** nutzen, das wir mit dem Kontextmenü-Befehl **Hinzufügen|Neues Element...** des Projekt-Elements im Projektmappen-Explorer öffnen. Stattdessen fügen wir das Element **Menüband (XML)** hinzu, für das wir den Namen **Ribbon_Access.vb** festlegen (s. Bild 2).

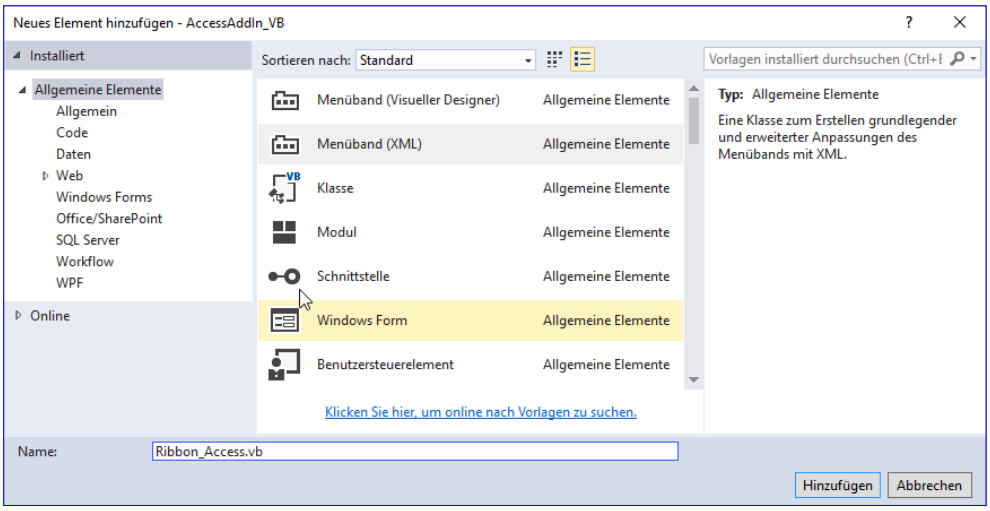


Bild 2: Anlegen des Elements **Menüband (XML)**

Aus der damit hinzugefügten Datei **Ribbon_Access.vb** kopieren wir die dort noch auskommentierte Methode **CreateRibbonExtensibilityObject** in das Klassenmodul **ThisAddIn.vb** und entfernen die Kommentarzeichen:

```
Protected Overrides Function _
        CreateRibbonExtensibilityObject() _
        As Microsoft.Office.Core.IRibbonExtensibility
    Return New Ribbon_Access()
End Function
```

Diese Methode wird beim Starten des Add-Ins automatisch ausgeführt. Einzelheiten zum Erstellen eines Ribbons über das Element **Menüband (XML)** finden Sie im Beitrag **COM-Add-In-Ribbons mit dem XML-Designer** (**www.access-im-unternehmen.de/1093**).

Funktionen hinzufügen

Unser Add-In soll nun sowohl eine Funktion enthalten, die sich auf die Anwendung selbst bezieht, als auch eine, welche auf die Daten einer geladenen Datenbank zugreift. Auf diese Weise erhalten Sie eine gute Ausgangsposition für selbst programmierte Add-Ins.

Die geplanten Funktionen sind in dem Ribbon-Tab aus Bild 3 abgebildet. Die linke Gruppe enthält zwei Schaltflächen, mit denen der Benutzer den Navigationsbereich ein- und ausblenden kann. Die rechte Gruppe enthält eine Schaltfläche, welche einen Dialog zur Anzeige aller Tabellen der aktuell geöffneten Datenbank liefern soll. Diese Schaltfläche soll natürlich nur aktiviert sein, wenn Access überhaupt eine Datenbank geladen hat.

Die Definition des Ribbons sieht wie in Listing 2 aus. Die erste Gruppe enthält die Schaltflächen **btnNavigationsbereichEinblenden** und **btnNavigationsbereichAusblenden**. Beide sind mit einer **onAction**-Callback-Eigenschaft und einem Bild ausgestattet. Die Bilder werden mit der **image**-Eigenschaft angegeben. Für das Laden der Bilder ist die im Element **customUI** im Attribut **loadImage** angegebene Methode verantwortlich (mehr zum Laden von Bildern im Beitrag **COM-Add-In-Ribbons mit dem XML-Designer**, **www.access-im-unternehmen.de/1093**). Die zweite Gruppe enthält die Schaltfläche **btnTabellen**, die ebenfalls eine **onAction**-Eigenschaft sowie zusätzlich die Callback-Eigenschaft **getEnabled** enthält.

Aktivieren und deaktivieren der Schaltfläche btnTabellen

Diese Schaltfläche soll ja nur aktiviert sein, wenn Access gerade eine geladene Datenbank enthält. Wie wir dies

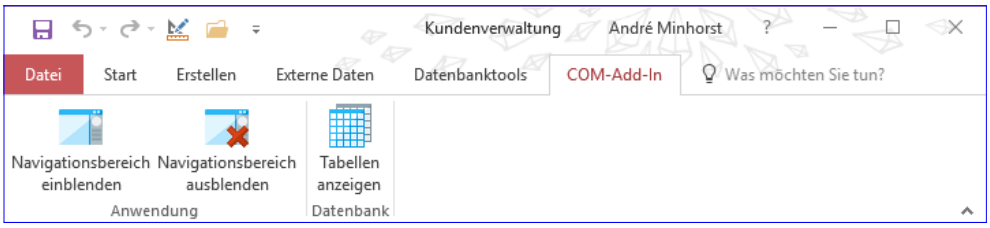


Bild 3: Unsere selbst erstellten Ribbon-Befehle

ermitteln, haben Sie weiter oben erfahren – wir starten per Timer jede Sekunde eine Prozedur, die prüft, ob **CurrentDb** den Wert **Nothing** hat. Wie können wir diese nun nutzen, um die Schaltfläche **btnTabellen** abhängig vom Ergebnis zu aktivieren oder zu deaktivieren? Zusammengefasst gehen wir so vor: Wir fügen der Klasse **Ribbon_Access.vb** eine Eigenschaft hinzu, welche den Zustand erfasst (Datenbank geladen/nicht geladen). Diese wird von der Callback-Funktion **getEnabled** ausgewertet. Noch dazu machen wir in der Klasse **Ribbon_Access.vb** aus dem **IRibbonUI**-Objekt **Ribbon** eine Eigenschaft gleichen Namens, auf die wir von außen zugreifen können. Damit können wir dann der Klasse **Ribbon_Access.vb** von außen sowohl den Zustand Datenbank geladen/nicht geladen mitgeben als auch dafür sorgen, dass die Methode **getEnabled** ausgelöst wird – nämlich durch Aufrufen der **Invalidate**-Methode des **IRibbonUI**-Objekts. Damit wir von der Klasse **ThisAddIn** allerdings überhaupt auf das **IRibbonUI**-Objekt zugreifen können, müssen wir dieses noch irgendwie innerhalb dieser Klasse referenzieren. Unter Visual Basic 2015 wäre es sogar möglich, dafür ein Modul mit einer öffentlichen Variablen anzulegen, aber diesen Weg wollen wir hier gezielt nicht gehen. Schauen wir uns den resultierenden Code an!

Ribbon in ThisAddIn.vb referenzieren

Die Klasse **ThisAddIn.vb** enthält zwei automatisch ausgelöste Methoden, nämlich **CreateRibbonExtensibility** und **ThisAddIn_Startup** – und zwar in dieser Reihenfolge. Wir deklarieren eine Variable des Typs **Ribbon_Access**, mit der wir die in **CreateRibbonExtensibility** erzeugte und als Funktionswert zurückgegebene Instanz der Klasse **Ribbon_Access.vb** speichern:

```
<?xml version="1.0" encoding="UTF-8"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" onLoad="Ribbon_Load" loadImage="loadImage">
  <ribbon>
    <tabs>
      <tab id="tabAddIn" label="COM-Add-In">
        <group id="grpAnwendung" label="Anwendung">
          <button id="btnNavigationsbereichEinblenden" label="Navigationsbereich einblenden" onAction="onAction"
              image="window_sidebar" size="large"/>
          <button id="btnNavigationsbereichAusblenden" label="Navigationsbereich ausblenden" onAction="onAction"
              image="window_sidebar_delete" size="large"/>
        </group>
        <group id="grpDatenbank" label="Datenbank">
          <button id="btnTabellen" label="Tabellen anzeigen" onAction="onAction" image="tables" size="large"
              getEnabled="getEnabled"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Listing 2: Definition des Beispielribbons

```
Private objRibbon As Ribbon_Access
```

Der Funktion **CreateRibbonExtensibility** fügen wir dann eine Anweisung hinzu, welche die neu erzeugte Instanz von **Ribbon_Access** in der Variablen **objRibbon** speichert. Der Inhalt dieser Variablen wird dann als Funktionswert zurückgegeben:

```
Protected Overrides Function _
        CreateRibbonExtensibilityObject() _
        As Microsoft.Office.Core.IRibbonExtensibility
    objRibbon = New Ribbon_Access()
    Return objRibbon
End Function
```

Der in **objRibbon** gespeicherten Instanz der Klasse **Ribbon_Access.vb** wollen wir auch einen Verweis auf die Access-Anwendung, also **Application**, übergeben. Das hat den Hintergrund, dass wir darin auch auf die Methoden von Access zugreifen wollen. Daher fügen wir in der Methode **ThisAddIn_Startup**, die ja nach dem Erstellen von **Ribbon_Access** ausgelöst wird, noch eine Zeile hinzu, welche die Eigenschaft **Application** der Klasse **Ribbon_Access.vb** mit dem Verweis aus der Objektvariablen **Application** füllt:

```
Private Sub ThisAddIn_Startup() Handles Me.Startup
    Dim timer As System.Timers.Timer
    objRibbon.Application = Application
    timer = New System.Timers.Timer(1000)
    AddHandler timer.Elapsed, AddressOf HandleTimer
    With timer
        .Start()
    End With
End Sub
```

Unsere weiter oben vorgestellte **Timer**-Methode **HandleTimer** ist auch von den Änderungen betroffen. In dem **If**-Zweig, in dem wir feststellen, dass keine Datenbank geladen ist, stellen wir die Eigenschaft **DatenbankGeladen** der in **objRibbon** gespeicherten Instanz der Klasse **Ribbon_Access** auf **False** ein. Danach rufen wir die **Invalidate**-Methode der **ribbon**-Eigenschaft von **objRibbon** auf:

```
Private Async Sub HandleTimer(sender As Object, _
        e As EventArgs)
    Await Task.Run(
        Sub()
            If Application.CurrentDb Is Nothing Then
                If Loaded = True Then
                    Loaded = False
                    objRibbon.DatenbankGeladen = False
                    objRibbon.ribbon.Invalidate()
                End If
                ...
```

Im **Else**-Teil dann, wo die Datenbank geladen ist, stellen wir **DatenbankGeladen** auf **True** ein und rufen wieder die **Invalidate**-Methode auf:

```
            ...
            Else
                If Loaded = False Then
                    Loaded = True
                    objRibbon.DatenbankGeladen = True
                    objRibbon.ribbon.Invalidate()
                End If
            End If
        End Sub
    )
End Sub
```

Das Ganze verlangt natürlich nun nach der Erläuterung der Änderungen in der Klasse **Ribbon_Access**. Hier brauchen wir erstmal den Namespace für Access:

```
Imports Access = Microsoft.Office.Interop.Access
```

Dann benötigen wir statt **Private Ribbon As Office.IRibbonUI** eine private Variable des Typs **IRibbonUI**, die wir per **Property** nach außen schreib- und lesbar machen:

Effizienz-Hacks: Mailabruf erschweren

Ich weiß nicht, wie es Ihnen geht, aber ich gestehe: Mail, Facebook und Co. sind echte Zeitfresser in meinem Leben. Wenn ich gerade mal nicht vorwärts komme, lese ich abwechselnd E-Mails (manchmal im Minutentakt) oder schaue, was es bei Facebook oder anderen sozialen Netzwerken für Neuigkeiten gibt. Wie viel Zeit könnte man sparen, wenn man diese Ablenkungen nicht direkt vor der Nase hätte – direkt neben den zu bearbeitenden Dokumenten oder Anwendungen! Dieser Beitrag zeigt, wie Sie zumindest den Umgang mit E-Mails etwas achtsamer gestalten oder diesen gleich auf bestimmte Uhrzeiten einschränken können.

Outlook ist auf meinem Desktop immer geöffnet. Kein Wunder: Ich rufe E-Mails damit ab, verwalte Kontakte damit und tracke die für meine Projekte aufgebrachte Arbeitszeit. Leider tracke ich damit nicht die Zeit, die für das Lesen von E-Mails oder für das Surfen im Internet draufgeht. Würde ich das mal vor Augen gehalten bekommen, wäre ich vielleicht ein wenig achtsamer bei diesem Thema.

Letztlich gibt es nur einen ganz kleinen Prozentsatz von E-Mails, die so wichtig sind, dass sie ganz schnell erledigt werden müssen. Und wer wirklich dringend etwas von einem möchte, der wird wahrscheinlich ohnehin zum Hörer greifen.

Und wenn wir ehrlich sind: Da immer mehr E-Mails, die wirklich nicht wichtig sind, mit der höchsten Priorität versendet werden, ist das rote Aufrufezeichen neben einer E-Mail auch nicht unbedingt Anlass für übertriebene Hektik (man munkelt, einige Benutzer hätten die höchste Priorität als Standard eingestellt und wüssten nicht, wie man diese Einstellung ändert ...). Nun gibt es grob drei Typen von E-Mail-Konsumenten:

- Der **Ich-will-nichts-verpassen-Konsument**: Er hat Outlook so eingestellt, dass automatisch alle 60 Sekunden neue E-Mails abgerufen werden und hechelt direkt beim Eingang des Signals für eine eingehende E-Mail zu Outlook.
- Der **Prokrastinierer**: Er erwartet nicht wirklich wichtige E-Mails und lässt nicht automatisch abrufen. Dafür ruft er selbst immer E-Mails ab, wenn die eigentliche Arbeit zu langweilig, zu stressig oder zu anstrengend ist.
- Der **Gelegenheitskonsument**: Er ist völlig entspannt und weiß, dass die Welt nicht untergeht, wenn er mal einen Tag keine E-Mails abruft. Er ruft E-Mails maximal ein bis zwei Mal pro Tag ab.

Gerade beim Prokrastinierer ist Achtsamkeit ein wichtiges Schlüsselwort: Letztlich weiß er, dass es lediglich Zeit kostet und die Erledigung der eigentlichen Aufgabe hinauszögert, wenn er E-Mails abruft. Dennoch macht er es. Das ist genau wie der Griff zur Zigarette, zur Süßigkeit oder zum Kaffee. Wie aber können wir dem Dauer-Mail-Checker helfen, von seinem Verhalten loszukommen? Dazu gibt es zwei Ansätze, die ich in den letzten Tagen testweise einsetze:

- Jedes Mal, wenn ich E-Mails abrufe und mindestens eine E-Mail heruntergeladen wird, trägt Outlook dies in den Kalender ein.
- Jedes Mal, wenn ich auf **Alle Ordner senden/empfangen** klicke, erscheint eine Meldung, gegebenenfalls mit Countdown, die mich fragt, ob ich gerade wirklich E-Mails abrufen möchte.

E-Mail-Lese-Tracker

Die erste Variante liefert im Kalender beispielsweise eine Ansicht wie in Bild 1. Jeder grüne Eintrag entspricht einem erfolgreichen Abruf einer E-Mail.

Was soll dies bewirken? Es zeigt dem Benutzer, wie oft er an diesem Tag tatsächlich E-Mails abgerufen hat. Diese Tage sind schon recht diszipliziert, teilweise wurde weniger als jede halbe Stunde auf die Taste zum Abrufen geklickt. Das könnte auch anders aussehen ...

Spätestens, wenn der Benutzer die übrigen Einträge im Kalender nicht mehr lesen kann, wird er über seinen E-Mail-Konsum nachdenken. Und falls nicht, haben wir ja später noch eine weitere Möglichkeit, die Achtsamkeit zu erhöhen.

Schauen wir uns zunächst die technische Umsetzung dieser Lösung an.

Outlook-Mailabrufe in den Kalender eintragen

Dazu fügen wir dem Modul **ThisOutlookSession** des VBA-Projekts von Outlook einige Zeilen Code hinzu. Den VBA-Editor von Outlook öffnen Sie übrigens am schnellsten mit der Tastenkombination **Alt + F11**. Dort finden Sie dann auch das genannte Modul, in dem wir zunächst die folgenden drei Variablen deklarieren:

```
Dim WithEvents objOutlook As Outlook.Application
Dim WithEvents objPosteingang As Outlook.Folder
Dim WithEvents objPosteingangItems As Outlook.Items
```

Außerdem benötigen wir noch eine Variable, in der wir das Datum der letzten E-Mail-Abfrage speichern:

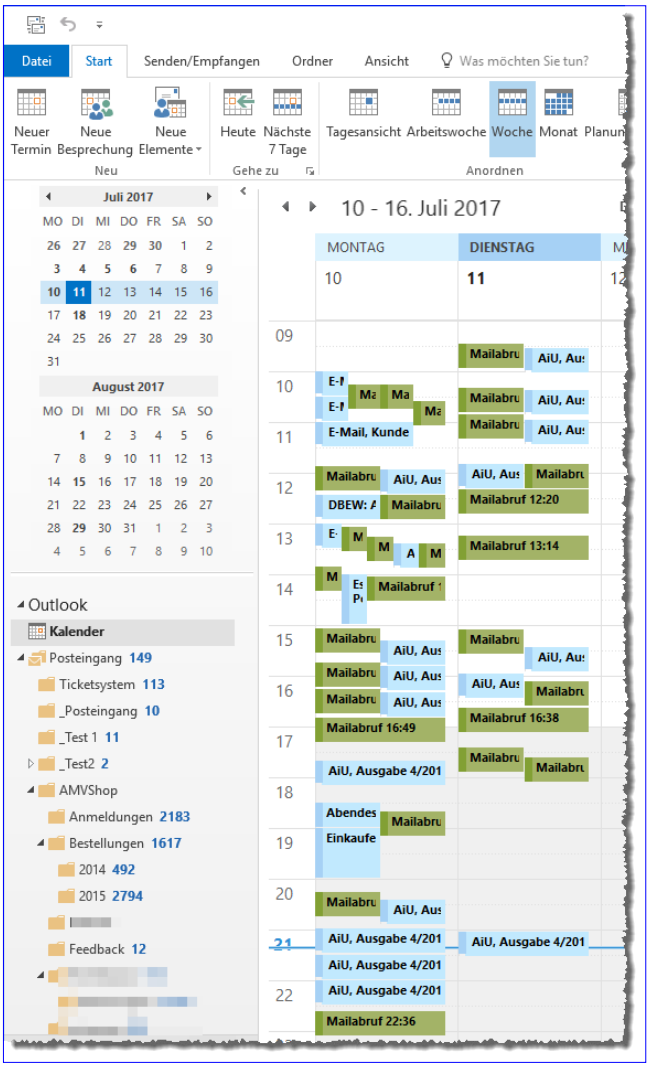


Bild 1: Jeder grüne Eintrag entspricht einem erfolgreichen Mail-Abruf.

```
Dim datLastMail As Date
```

Danach legen wir die Ereignisprozedur **Application_Startup** an, was Sie durch die Auswahl des Eintrags **Application** im linken Kombinationsfeld des Codefensters und Auswahl des Eintrags **Startup** im rechten

```
Public Sub Application_Startup()
    Set objOutlook = Application
    Set objPosteingang = objOutlook.GetNamespace("MAPI").GetDefaultFolder(olFolderInbox)
    Set objPosteingangItems = objPosteingang.Items
End Sub
```

Listing 1: Diese Methode wird beim Start von Outlook ausgeführt.

Kombinationsfeld erledigen können. Diese Prozedur füllt lediglich die drei zuvor deklarierten Objektvariablen mit den entsprechenden Objekten (s. Listing 1).

objOutlook erhält einen Verweis auf das **Application**-Objekt der aktuellen Outlook-Instanz. **objPosteingang** nimmt einen Verweis auf den Outlook-Ordner **Posteingang** auf, den wir über den MAPI-Namespace und die Methode **GetDefaultFolder** mit dem Wert **olFolderInbox** als Parameter ermitteln. Schließlich erhält die **Items**-Variable **objPosteingangItems** einen Verweis auf die **Items**-Auflistung des in **objPosteingang** gespeicherten Ordners.

Für das Objekt **objPosteingangItems** legen wir eine Ereignisprozedur an, die durch das Ereignis **ItemAdd** ausgelöst wird. Um die leere Ereignisprozedur zu erstellen, wählen wir wieder zwei Einträge in den beiden Kombinationsfeldern des VBA-Editors aus – links den Eintrag **objPosteingangItems** und rechts **ItemAdd**. Diese Ereignisprozedur füllen wir wie in Listing 2.

Die Prozedur nimmt mit dem Parameter **Item** das neue Element des Ordners entgegen, in diesem Fall eine E-Mail. Die Prozedur definiert eine Variable namens **objCalendar**, welche den Kalender referenzieren soll und eine **AppointmentItem**-Variable für den neu anzulegenden Termin. Außerdem ermitteln wir mit der **DateSerial**-Funktion eine Zeitangabe, welche das aktuelle Datum und die aktuelle Uhrzeit ermittelt. Dabei soll als Anzahl der Sekunden der Wert **0** übergeben werden, damit pro Minute nur ein Mail-Abruf aufgezeichnet werden kann.

Nun folgt eine Prüfung, ob der in der modulweit deklarierten Variablen **datLastMail** gespeicherte Wert mit dem Wert aus **datLastMailTemp** übereinstimmt. Ist dies der Fall, dann wurde in dieser Minute bereits einmal eine E-Mail abgerufen – und mehr als einen Eintrag pro Minute wollen wir dem Kalender nicht zumuten.

Anderenfalls wird die Variable **objCalendar** mit dem Kalender-Ordner gefüllt, welchen wir über die Metho-

```
Private Sub objPosteingangItems_ItemAdd(ByVal Item As Object)
    Dim objCalendar As Outlook.Folder
    Dim datLastMailTemp As Date
    Dim objAppointment As Outlook.AppointmentItem
    datLastMailTemp = DateSerial(Year(Now), Month(Now), Day(Now)) + TimeSerial(Hour(Now), Minute(Now), 0)
    If Not datLastMailTemp = datLastMail Then
        Set objCalendar = objOutlook.GetNamespace("MAPI").GetDefaultFolder(olFolderCalendar)
        Set objAppointment = objOutlook.CreateItem(olAppointmentItem)
        With objAppointment
            .Subject = "Mailabruf " & Format(Now, "hh:nn")
            .Start = Now
            .Duration = 30
            .Categories = "Mail abrufen"
            .ReminderSet = False
            .Save
            Debug.Print "Geschrieben " & .Subject
        End With
    End If
    datLastMail = datLastMailTemp
End Sub
```

Listing 2: Dieses Ereignis wird ausgelöst, wenn eine neue E-Mail in der **Items**-Liste landet.

de **GetDefaultFolder** des MAPI-Ordners mit dem Wert **olFolderCalendar** als Parameter referenzieren. **objAppointment** füllen wir mit der Methode **CreateItem**, der wir den Wert **olAppointmentItem** übergeben. So wird ein **AppointmentItem**-Element erstellt.

Danach stellt die Prozedur einige Werte für das neue **AppointmentItem**-Element ein. Der Betreff (**Subject**) soll den Text **Mailabruf** sowie die aktuelle Uhrzeit im Format **hh:nn** als Inhalt anzeigen. Die Eigenschaft **Duration** legt mit dem Wert **30** die Dauer des Elements in Minuten fest. Als Kategorie stellen wir **Mail abrufen** ein. Diese Kategorie müssen Sie noch als Kategorie mit dem Dialog **Farbkategorien** einrichten (s. Bild 2). Außerdem deaktivieren wir mit **ReminderSet = False** die Anzeige von Erinnerungen zu diesem Termin. Schließlich speichert die **Save**-Methode den Termin und sorgt so für dessen Anzeige in Outlook. Damit der Vergleich des Zeitpunkts des aktuellen Mailabrufs mit dem des vorherigen gelingt, müssen wir nun noch den Wert von **datLastMailTemp** in **datLastMail** eintragen.

Wie funktioniert es?

Wenn Sie nun direkt eine Testmail an sich selbst verschicken, Mails abrufen und sich wundern, warum kein entsprechender Termin im Kalender eingetragen wird – keine Sorge: Es fehlt nur noch ein kleiner Schritt. Sie müssen nämlich dafür sorgen, dass die Ereignisprozedur **Application_Startup** ausgelöst wird. Diese füllt erst die Variablen mit den gewünschten Objekten und sorgt dann dafür, dass die Ereignisprozedur für die Variable **objPosteingangItems** beim nächsten Eingang einer E-Mail ausgelöst wird. Dies können Sie in diesem Fall manuell erledigen, indem Sie die Einfügemarke innerhalb der Prozedur platzieren und die Taste **F5** betätigen. Oder Sie starten Outlook einfach neu, um die Prozedur automatisch auszulösen.

Achtsamkeit beim Mailabruf

Damit kommen wir nun zur zweiten Lösung, die direkt beim Klick auf die Schaltfläche **Alle Ordner senden/empfangen** ausgelöst wird und eine passende Meldungsbox

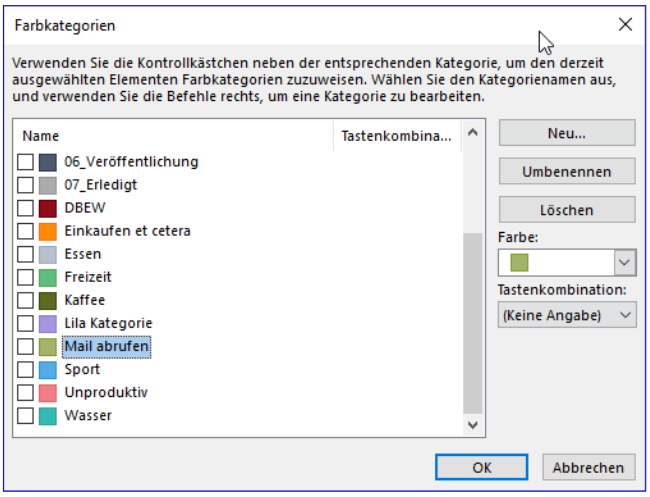


Bild 2: Einstellen der Farbe für die Kategorie **Mail abrufen**

anzeigen soll. Die schlechte Nachricht ist: Es gibt keine Möglichkeit, den Mausklick auf diese Schaltfläche des Ribbons mithilfe einer VBA-basierten Lösung abzufangen. Stattdessen benötigen wir ein COM-Add-In auf Basis von Visual Studio. Zum Glück gibt es Visual Studio 2015 in der Community-Edition kostenlos für einzelne Entwickler und kleinere Betriebe, sodass keine zusätzlichen Kosten für diese sehr interessante Programmierung einer Erweiterung von Outlook anfallen. Sie müssen sich lediglich Visual Studio 2015 Community herunterladen und dieses installieren. Außerdem benötigen Sie alle aktuellen Erweiterungen, die dann auch die für die Programmierung von COM-Add-Ins für die Office-Anwendungen benötigten Vorlagen mitliefern.

Projekt anlegen

Nach dem Starten von Visual Studio legen Sie ein neues Projekt des Typs **Visual Basic\Office/SharePoint\VSTO-Add-Ins\Outlook 2013 und 2016 VSTO-Add-In** an. Legen Sie als Name **OutlookAddIn** fest (s. Bild 3).

Visual Studio zeigt nun ein Modul mit den zwei Ereignismethoden **ThisAddIn_Startup()** und **ThisAddIn_Shutdown()** an. Diese beiden benötigen wir nicht. Wir wollen eine Ribbon-Definition hinzufügen, mit der wir die eigentliche Funktion der Schaltfläche **Alle Ordner senden/empfangen** unterbinden und eine eigene Funktion ausführen.

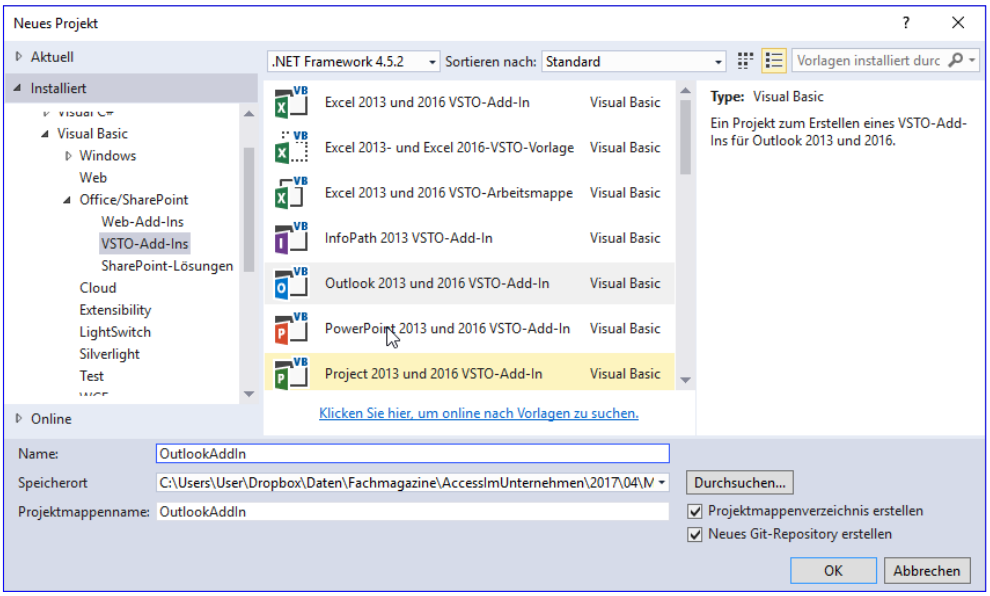


Bild 3: Anlegen eines neuen Projekts auf Basis der Outlook-Vorlage

Dazu fügen wir dem Projekt zunächst ein Ribbon hinzu. Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf den Projektnamen **OutlookAddIn** und wählen Sie den Eintrag **Hinzufügen|Neues Element**. Es erscheint der Dialog **Neues Element hinzufügen**. Hier wählen Sie den Eintrag **Menüband (Visueller Designer)** aus und geben den Namen **Outlook_Ribbon** ein (s. Bild 4).

Es erscheint der Designer für den Ribbon-Entwurf, der jeden Access-Entwickler, der sich bisher mit dem Ribbon herumschlagen durfte, neidisch werden lässt. Unser Ziel ist es nun, wie es auch unter Access möglich ist, ein Command-Element per Ribbon-Definition hinzuzufügen, das die Schaltfläche **Alle Ordner senden/empfangen** referenziert und für diese Schaltfläche eine neue Funktion hinterlegt. Dummerweise liefert die Toolbox aber überhaupt kein Element namens **Command**.

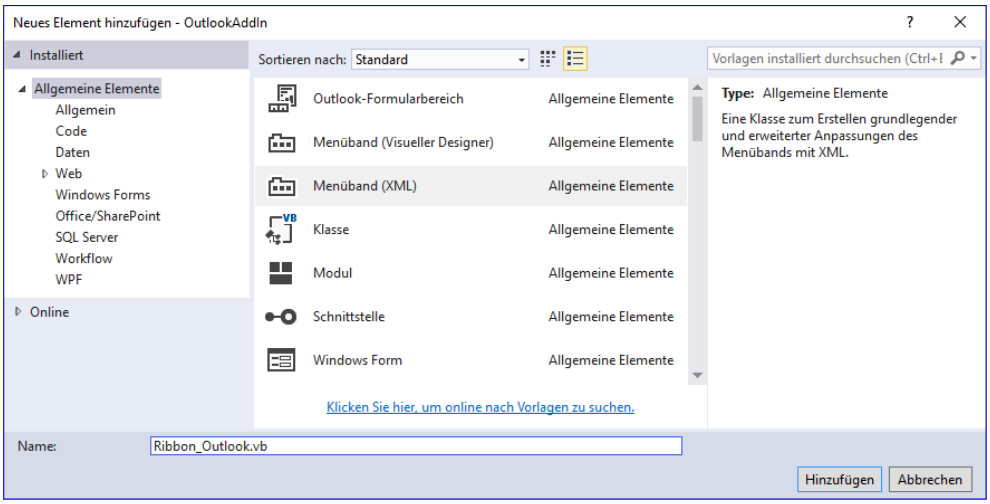


Bild 4: Hinzufügen eines Ribbons

XML-Definition erstellen

Können wir also von einem VSTO-Add-In gar keine eingebauten Schaltflächen mit neuen Funktionen versehen? Doch, wir sind nur etwas in die falsche Richtung gelaufen. Wir können nämlich durchaus auch mit einer klassischen XML-Definition für das Ribbon arbeiten. Dazu fügen wir einfach ein neues Objekt mit der Bezeichnung **Menüband (XML)** zum Projekt hinzu (wie weiter oben – rechter Mausklick auf das Projekt-Element im Projektmappen-Explorer, dann den Eintrag **Hinzufügen|Neues Element...** auswählen und im nun erscheinenden Dialog **Menüband (XML)** wählen).

Legen Sie den Namen **Ribbon_Outlook.vb** für das neue Element fest.

Explorer Control

Für manche Aufgaben benötigen auch Datenbanken die Anzeige oder Auswahl von Dateien oder Ordnern des Betriebssystems. Manche davon lassen sich mit dem Dateiauswahldialog von Office erledigen. Hin und wieder ist aber eine unmittelbare Einsicht in das Dateisystem nützlich. Der Windows Explorer lässt sich mit Einschränkungen fernsteuern, doch schicker wäre ein Explorer direkt in einem Formular. Und das geht!

Explorer Browser Control aus Windows

Den Explorer komplett nachzubilden, ist in der Regel mit gewaltigem Aufwand verbunden. Hier sind unzählige **Shell**-Objekte zu berücksichtigen. Wer etwa das **Access 2007 Praxisbuch** kennt, findet in der zentralen Beispielanwendung **dmsBase** ein **ActiveX-Steuerelement ShellExplorer.ocx**, das zur Laufzeit weitgehend dem Windows Explorer gleicht. Die Programmierung dieses Controls benötigte über 20.000 Zeilen Code.

Diese Zeiten sind seit **Windows Vista** vorbei! Microsoft hat hier ein neues Control in den **Shell**-Komponenten verbaut, welches sich **ExplorerBrowser** nennt. Dabei handelt es sich um einen kompletten Explorer mit allem Zubehör als einbettbares Fenster. In gewisser Weise ähnelt diese Komponente dem **Webbrowser Control**, welches den **Internet Explorer** als ActiveX-Steuerelement zur Verfügung stellt. Nur gibt es die **ExplorerBrowser**-Komponente eben leider nicht als ActiveX-Control. Sie muss stattdessen über **COM-Schnittstellen** erzeugt und gesteuert werden. Das wäre weiter nicht tragisch, enthielte denn irgendeine Windows-Bibliothek, die sich in die **Verweise** von VBA eintragen ließe, die entsprechenden Schnittstellendefinitionen. Das ist jedoch nicht der Fall.

OLEEXP als Nachfolger von OLELIB

Zum Glück gibt es Typbibliotheken, die diesem Umstand abhelfen. Der Microsoft **MVP** für **Visual Basic 6**, **Eduardo Morcillo**, stellte einst die geniale Bibliothek **olelib.tlb** bereit, die in großem Umfang **COM**-Schnittstellen von Windows-Komponenten definiert. Mit dieser können Dinge erreicht werden, die sich auch mit Unmengen von **API**-Funktionen nicht realisieren lassen. Das Problem mit dieser Bibliothek aus dem Jahre 2003 ist, dass sie ziemlich ins Alter gekommen ist. Sie kann nur abbilden, was bis zu diesem Zeitpunkt bekannt war – und das sind maximal die Schnittstellen von **Windows XP**.

Das fiel auch anderen Programmierern auf. Es gibt immer noch rastlose Geister, die das altehrwürdige **VB6**, auf dessen Stand das Pendant **VBA** nun mal leider stehenblieb, weiterentwickeln. Viele kompetente Mitstreiter finden sich im Forums **vbforums.com**. Der User **fafalone** hat sich an die Arbeit gemacht, die **olelib** mit neuem Leben zu füllen, und in die abgeleitete neue Bibliothek **oleexp.tlb** Schnittstellendefinitionen eingebracht, die bis **Windows 7** reichen. Und einer der Neuen ist das Interface zum **ExplorerBrowser**.

Es handelt sich, wohlgemerkt, nicht um eine aktive Komponente! Eine **tlb** ist lediglich eine Typbibliothek, die sich auf aktive Komponenten bezieht. Es ist deshalb in Ihrem Unternehmen nicht zu befürchten, dass sie bei der Rasterfahndung nach gefährdenden Komponenten durchfällt, wie manche andere ActiveX-Dateien. Sie können sie getrost in die Verweise Ihrer Datenbank laden, wozu sie noch nicht einmal zwingend registriert werden muss. Die Methode **References.LoadFromFile** erlaubt auch die Integration zur Laufzeit.

Registrieren der OLEEXP

Freilich ist es dennoch sinnvoll, die Bibliothek im System zu registrieren, damit sie in Zukunft auch für andere VBA-Projekte zur Verfügung steht. Im Verzeichnis zur Beispieldatenbank **explorerctl.accdb** finden Sie neben der **oleexp.tlb** noch die ausführbare Datei **regtlibv12.**

exe, die solche Typbibliotheken registrieren kann. Das ist ein Kommandozeilen-Tool von Microsoft, das Sie wahrscheinlich auch auf Ihrem System als Bestandteil des **NET-Frameworks 4** vorfinden. Der Einfachheit halber haben wir die Datei in das Verzeichnis der Datenbank kopiert.

Damit Sie nicht umständlich in der Windows-Kommandozeile herumhantieren müssen, ruft eine weitere Datei **register_tlb.bat** die **regtlibv12** auf und übergibt ihr als Parameter die **oleexp.tlb**. Die Registrierung verlangt über die **UAC** automatisch nach erhöhten Rechten. Die Registrierung der Bibliothek erfolgt danach ganz im Stillen. Nach diesem Vorgang können Sie die Bibliothek über die Verweise von VBA und den Eintrag **oleexp - olelib with modern interfaces by fafalone** in Ihr Projekt laden.

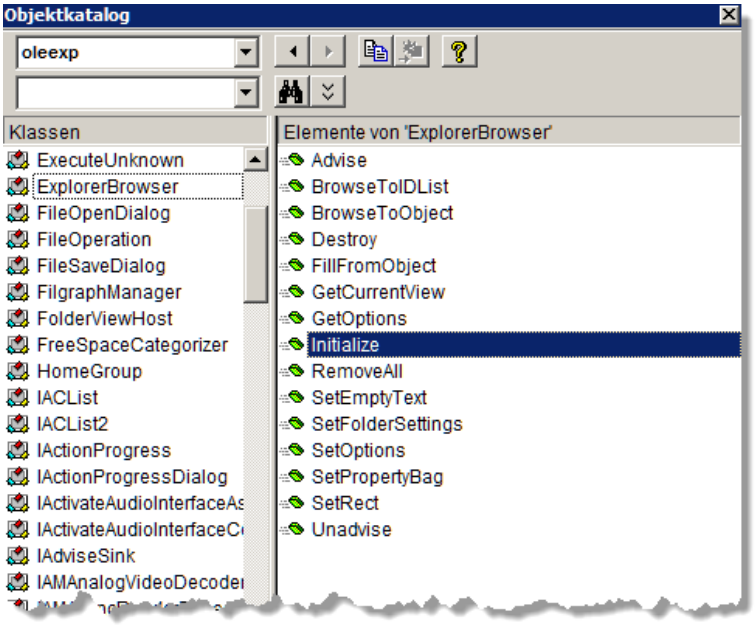


Bild 1: Die Klasse **ExplorerBrowser** und ihre Methoden im VBA-Objektkatalog

Die Klasse ExplorerBrowser

Im VBA-Objektkatalog stellen Sie links oben die Bibliothek **oleexp** ein und navigieren danach zur Klasse **ExplorerBrowser**. Der Objektkatalog gibt leider keine Auskunft darüber, ob eine Klasse instanzierbar ist oder lediglich eine Interface-Definition darstellt. Sie behelfen sich damit, dass Sie das Schlüsselwort **New** ins VBA-Direktfenster schreiben und die gewünschte Bibliothek anfügen. Etwa so:

```
New oleexp.
```

Sobald Sie den Punkt anfügen, zeigt **Intellisense** alle Klassen an, die in der **Library** instanzierbar sind. Beim **ExplorerBrowser** ist das tatsächlich der Fall. Eine Instanz kann also per **New**-Operator erzeugt werden, und die sonst häufig benötigten API-Aufrufe mit **CoCreateInstance** et cetera entfallen. Bild 1 zeigt die recht überschaubaren und dennoch mächtigen Methoden der Klasse, auf die wir im Einzelnen noch zu sprechen kommen.

Sie erzeugen den Explorer im Formular also, indem Sie eine modulweit gültige Objektvariable deklarieren und ihr eine neue Instanz der Klasse zuweisen:

```
Dim oExplBrowser As oleexp.ExplorerBrowser

Private Sub Form_Load()
Set oExplBrowser = New oleexp.ExplorerBrowser
...
```

Damit tut sich indessen noch nichts. Erst müssen noch einige Methoden des Objekts aufgerufen werden, die grundlegende Einstellungen anweisen. Die wichtigste davon ist die **Initialize**-Methode, welche dem Objekt unter anderem sagt, wo der Explorer zu platzieren ist. Dazu gleich mehr. Davor zeigt Ihnen aber Bild 2, wie das Resultat aussieht. Auf der linken Seite findet sich der Verzeichnisbaum mit allen Elementen. Nicht nur das Dateisystem ist vorhanden, sondern etwa auch Netzlaufwerke, Bibliotheken und die Systemsteuerung! Rechts zeigen sich die Elemente des Verzeichnisses. Die Spaltenköpfe können auf gleiche Weise manipuliert werden, wie sonst auch. Der Toolbar oben enthält dieselben Elemente, wie der **Windows Explorer**. Die Ansicht und

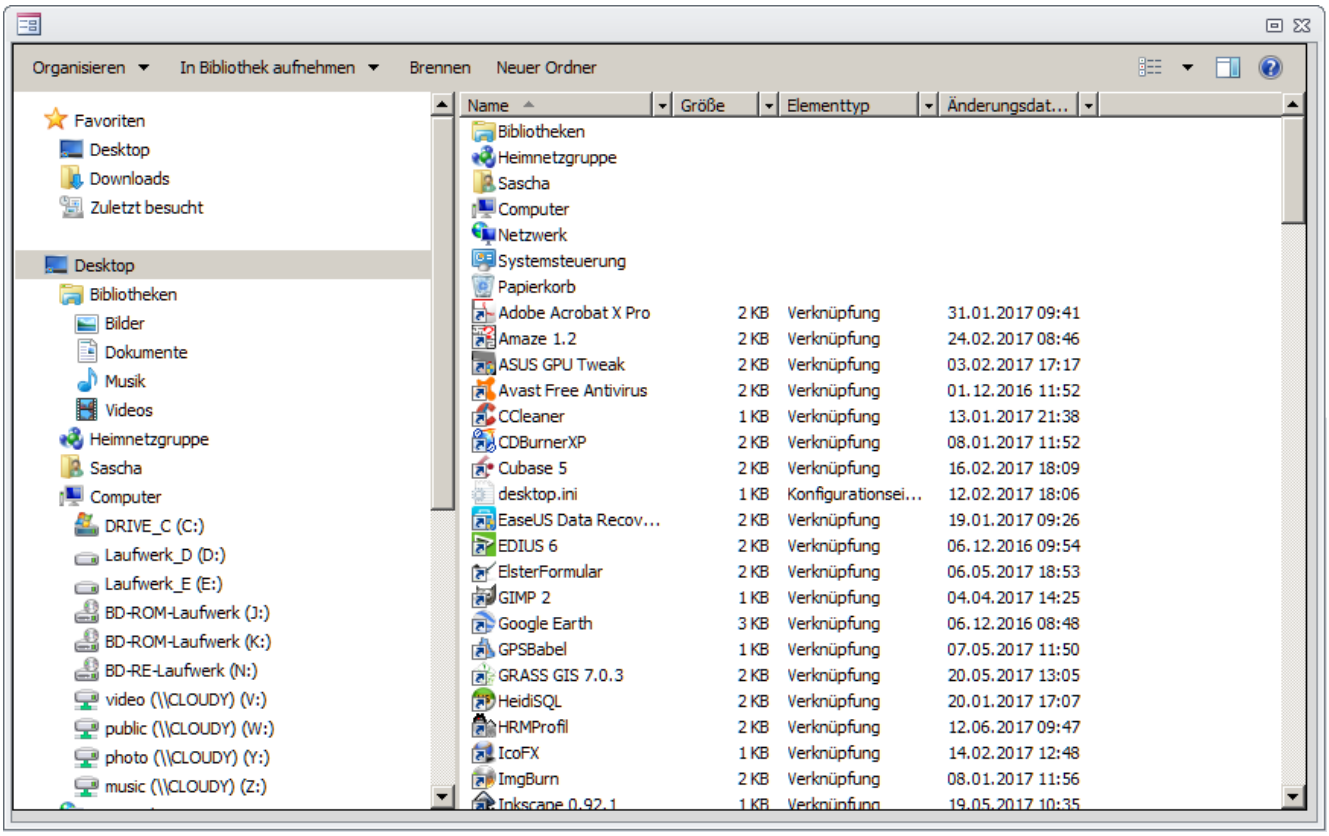


Bild 2: Im Formular **fmExplorerSimple** ist nur ein **Windows Control** im Detailbereich untergebracht, welches den Explorer abbildet

das Layout können also darüber gesteuert werden. Selbstverständlich funktionieren auch **Drag And Drop** und alle Kontextmenüs samt gegebenenfalls installierten **Shell Extensions**.

Ein Blick in die Entwurfsansicht des Formulars (s. Bild 3) verblüfft möglicherweise etwas. Hier finden Sie tatsächlich kein einziges Steuerelement, sondern nur den Detailbereich vor. Es ist Aufgabe der **Initialize**-Methode,

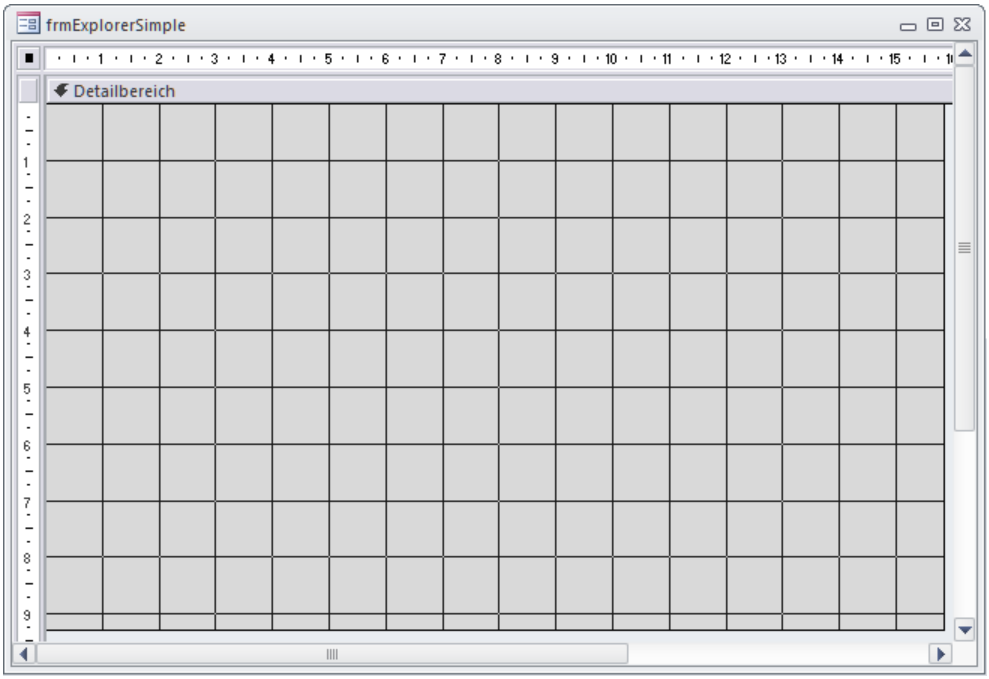


Bild 3: Die Entwurfsansicht des Formulars **frmExplorerSimple** stellt sich in der Tat äußerst simpel dar

auf dem Detailbereich das **ExplorerBrowser Control** zu rendern.

Initialisieren des ExplorerBrowsers
Die **Initialize**-Funktion hat folgende Syntax:

```
Initialize (hwnd As Long, rct As RECT, _
                          pfs As FOLDERSETTINGS) As Long
```

Der Aufruf dieser einen Funktion führt schon zur Anzeige des Explorers im Formular – wenn denn die Parameter korrekt übergeben werden.

Das wäre zunächst das **Fenster-Handle hwnd**, das angibt, auf welchem Fenster das Control angebracht werden soll. Der Typ **RECT** für die Variable **rct** ist ebenfalls in der **oleexp**-Bibliothek definiert und steuert die Positionierung des Controls:

```
Type RECT
    Bottom As Long
    Left As Long
    Right As Long
    Top As Long
End Type
```

Left ist der horizontale Abstand zum linken Rand des Host-Fensters, **Top** der vertikale zum oberen Rand. Die Breite und Höhe des Controls ergeben sich aus **Right** und **Bottom**. Ist **Left** etwa **20** und **Right 300**, so wird die Breite des Controls **280** sein. Mit **30** für **Top** und **200** für **Bottom** entsteht eine Höhe von 170. Alle Angaben in Pixel.

Der letzte Parameter **pfs** muss eine Variable vom Typ **oleexp.FOLDERSETTINGS** sein. Der Typ setzt zwei Enumerationskonstanten zusammen:

```
Type FOLDERSETTINGS
    fFlags As FOLDERFLAGS
    ViewMode As FOLDERVIEWMODE
End Type
```

FOLDERFLAGS und **FOLDERVIEWMODE** sind Konstanten, die über den **Or**-Operator zusammengesetzt werden können. Beispiel:

```
Dim pfs As oleexp.FOLDERSETTINGS
pfs.fFlags = FWF_ALIGNLEFT Or FWF_NOWEBVIEW
pfs.ViewMode = FVM_DETAILS
```

Damit stellt das Control die Dateien dann einerseits auf **Links angeordnet** ein und verbietet die Web-Ansicht (**fFlags**), andererseits wird die Detail- beziehungsweise Report-Ansicht angeordnet (**ViewMode**). **FVM_SMALLICON** etwa würde zur Ansicht **Kleine Symbole** führen, **FVM_THUMBNAIL** zu **Große Symbole**.

Schon beim Initialisieren können also grundlegende Eigenschaften des Controls gesetzt werden.

Kopfzerbrechen bereitet indessen das benötigte Fenster-**Handle hwnd**. Zwar weist auch das **Access**.**Form**-Objekt eine Eigenschaft **hwnd** auf, doch dieses **Handle** bezieht sich auf das gesamte Fenster, nicht aber auf einen Formularbereich, wie den Detailbereich. Dieser, wie auch Formularkopf oder Formularfuß, sind tatsächlich eigene Fenster, die als Kindfenster des Hauptfensters fungieren. Zwar kann einer dieser Bereiche über die **Section**-Funktion des Formulars ermittelt werden (**Form.Section (acDetail)** etwa), doch dieses zurückgegebene **Section**-Objekt kennt kein Fenster-Handle. Wir kommen hier nicht umhin, **API**-Funktionen zu bemühen. Zum Glück ist der Code hierfür nicht sonderlich kompliziert. Listing 1 zeigt eine Hilfsroutine, die bei uns zum Einsatz kommt.

Sie geht vom Handle des Hauptfensters (**Me.hwnd**) aus und befragt dann dessen Kindfenster mit der API-Funktion **GetWindow**. Dazu gehören allerdings nicht nur die Bereiche des Formulars, sondern etwa auch andere Fensterelemente, wie die Navigationsleiste. Die Bereiche lassen sich aber dadurch identifizieren, dass sie alle den Fensterklassennamen **OFormSub** tragen, den die Funktion **GetClassName** zurückgibt. Grundsätzlich

sind sowohl Formularkopf wie -fuß vorhanden. Sind diese ausgeblendet, so existieren die Fenster zwar, haben jedoch die Höhe **0**. Das zweite Kindfenster ist also immer der Detailbereich. Die Schleife identifiziert den Klassennamen und bricht ab, wenn der korrekte Name gefunden wurde. Da dann bereits das nächste Kindfenster über **GetWindow** mit **GW_HWNDNEXT** enumeriert wurde, steht in **hwnd** das richtige Handle, welches als Rückgabewert der Funktion dient.

```
Private Const GW_CHILD As Long = 5
Private Const GW_HWNDNEXT As Long = 2
PrivateDeclareFunctionGetClassNameLib"user32.dll"Alias"GetClassNameA"(ByValhwndAsLong,_
                                ByVal lpClassName As String, ByVal nMaxCount As Long) As Long
PrivateDeclareFunctionGetWindowLib"user32.dll"(ByValhwndAsLong,ByValwCmdAsLong)AsLong

Private Function GetDetailHwnd() As Long
    Dim hwnd As Long
    Dim ret As Long
    Dim rct As RECT
    Dim sClass As String

    hwnd = GetWindow(Me.hwnd, GW_CHILD)
    Do
        sClass = String(255, 0)
        ret = GetClassName(hwnd, sClass, 255)
        sClass = Left(sClass, ret)
       hwnd = GetWindow(hwnd, GW_HWNDNEXT)
    Loop Until sClass = "OFormSub"
    GetDetailHwnd = hwnd
End Function
```

Listing 1: Hilfsfunktion mit API zum Ermitteln des **Fenster-Handles** des Detailbereichs eines Formulars

Damit haben wir bereits alle nötigen Bestandteile beisammen, um das Formular mit eingebettetem Explorer zum Laufen zu bringen. Den Code des Beispielformulars mit dem kleinsten Aufwand (**frmExplorerMinimal**) demonstriert Listing 2. Die ganze Instanzierung des Explorer Controls findet im Ereignis **Form_Load** statt.

Erst werden die zwei Teile der **FOLDERSETTINGS**-Variablen **pfs** mit Vorgabewerten bestückt. Selbst das könnte im Prinzip entfallen, da beide dann eben den Wert **0** enthielten. Zwar gibt es eigentlich keine von Microsoft deklarierte **ViewMode**-Konstante, die diesem Wert entspricht, ein Fehler ereignet sich dabei aber nicht. Die Ansicht stellt sich bei Auskommentieren der beiden Zeilen einfach auf **Große Symbole** ein. Die **Flags**-Konstanten sind ohnehin alle optional.

Das Setzen der Abmessungen des Controls über die **RECT**-Variable **rct** allerdings ist zwingend erforderlich. Für **Left** und **Right** bleibt es bei **0**, was das Control in der linken oberen Ecke des Detailbereichs anordnet. Die Höhe in Bottom ergibt sich aus der Höhe des Detailbereichs (**Section acDetail**), wobei der Wert durch Division mit **15** von **Twips** nach **Pixel** umgewandelt wird. Die Breite des Controls ergibt sich aus der des Formulars selbst (**InsideWidth**). Schließlich wird für die per **New** erzeugte Instanz **oExplBrowser** die **Initialize**-Methode aufgerufen und das Fenster-Handle des Detailbereichs über die Hilfsfunktion **GetDetailHwnd** (Listing 1) übergeben.

Nicht erwähnt haben wir bisher die Methode **SetOptions** des **ExplorerBrowser**-Objekts. Die hier anzugebenden Konstanten steuern das Gesamtlayout des Controls. Wichtig etwa ist das Setzen von **EBO_SHOWFRAMES**. Erst diese Zuweisung sagt dem Control, dass es den Explorer komplett nachbilden soll, und Treeview sowie Toolbar anzeigen soll. Unterbleibt das, so zeigt es lediglich die Dateiansicht an, wie in Bild 4. Für diese gibt es sicher ebenfalls Anwendungsfälle.

Unterschlagen haben wir auch, dass das Control erst wissen muss, welches Verzeichnis des Systems es überhaupt anzeigen soll. Dies übernimmt seine Methode **BrowseToIDList**. Lässt man sie weg, so zeigt das Control nur eine weiße Fläche.

Die Methode erwartet eine sogenannte **ID-List**. Das ist in der **Shell**-Welt im Grunde eine eindeutige **ID** eines Verzeichnisses oder einer Datei. Einen String für einen Verzeichnispfad kann man hier nicht angeben, sondern müsste diesen erst über eine Hilfsfunktion in eine **ID** umwandeln. Das erläutern wir später. Die andere hier verwendete Option ist das Ermitteln einer **ID** für einen der Standardordner von Windows über die API-Funktion **SHGetSpecialFolderLocation**. Ihr übergibt man die Konstante eines der Standardordner, wobei **CLSID_DESKTOP** den Desktop symbolisiert. Möchten Sie zu den Laufwerken navigieren, wie in Bild 4, so geben Sie stattdessen die Konstante **CLSID_DRIVES** an. Die möglichen Konstanten sehen Sie im Objektkatalog zur Bibliothek **oleexp** unter der Liste **CSIDLs** ein.

Die ermittelte **ID** speichert die formularglobale Variable **IPIDL**. Sie wird **BrowseToIDList** übergeben. Der zweite Parameter **SBSP_ABSOLUTE** sagt ihr zusätzlich, dass ein absoluter Sprung zum Ordner erfolgen soll. Möglich wäre auch ein relativer Sprung, den wir hier aber nicht erklären, weil dies zu deutlich höherem Aufwand führen würde.

Mehr gibt es nicht zu tun. Die gerade 40 Zeilen des Formularcodes zeigen, wie einfach ein Explorer anzulegen ist. Beim Schließen des Formulars (**Form_Unload**) muss allerdings der Explorer wieder abgebaut werden, damit es nicht zu Speicherlecks oder Abstürzen kommt. Die **Destroy**-Methode veranlasst dies. Außerdem sollte die erhaltene Ordner-ID in **IPIDL** noch freigegeben werden, was über die API-Funktion **CoTaskMemFree** geschieht. Wir haben damit nun einen funktionsfähigen Explorer im Formular. Wünschenswert wäre allerdings noch eine erweiterte Einflussnahme auf seine Gestalt und sein Verhalten, wie auch zusätzlich Interaktion mit dem Control und den angezeigten Ordnern und Dateien. Die aufgebohrte Version mit allerlei Zusatz-Features nennt sich in der Beispieldatenbank **frmExplorerCtl**.

Das Ultra-Komplettbeispiel

Bild 5 zeigt dieses Formular zur Laufzeit. Beschreiben wir kurz seine Funktionalität. Alle Bezeichnungen sind in Englisch gehalten, damit der Bezug zu den Methoden im VBA-Code etwas deutlicher wird.

Zunächst sind im Formular auch Kopf und Fuß angezeigt. Das Explorer Control befindet sich ausfüllend im Detailbereich. Im Formularkopf bewirkt ein Klick auf die Schaltfläche **Reset to 'Computer'** den Sprung zu eben diesem Spezialordner, also der Laufwerksübersicht. Haben Sie

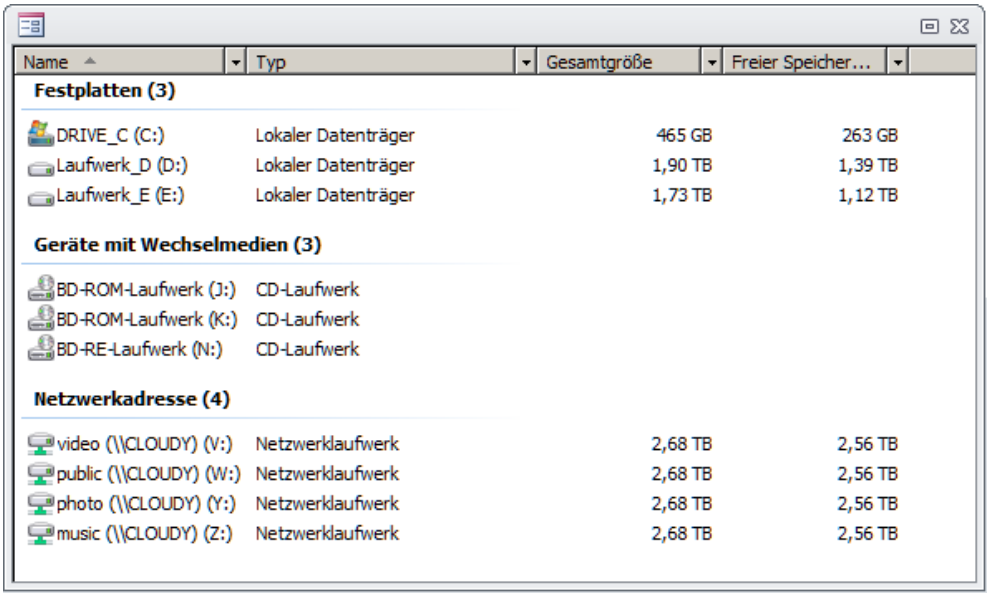


Bild 4: Ist **EBO_SHOWFRAMES** nicht gesetzt, so zeigt das Control nur den rechten Teil des Explorers

sich im Explorer durch diverse Verzeichnisse gehangelt, so gelangen Sie über den Button **Back** zum vorigen Verzeichnis, mit **Forward** zum nächsten, so wie bei der Navigation im richtigen Explorer auch. **Dir Up** schaltet eine Verzeichnisebene aufwärts, falls möglich. Beim Desktop klappt das natürlich nicht. Die Textbox oben rechts zeigt das aktuelle Verzeichnis an. Dies entspricht der Adresszeile des Explorers. Eine Eingabe ist hier allerdings nicht möglich. Bei speziellen Elementen, wie denen der Systemsteuerung etwa, bleibt die Textbox leer.

Rechts unten gibt es eine mehrzeilige Textbox, die der Ausgabe von durch das Control ausgelösten Ereignissen dient wie auch der weiterer Informationen. In der Abbildung ist zu erkennen, dass etwa die Navigation zu Verzeichnissen abgefangen werden kann. Auch die Markierung von Dateien bleibt dem Formularcode nicht verborgen.

Im linken Bereich des Formularfußes können über zahlreiche Checkboxen und Optionsfelder Einstellungen für das Control vorgenommen werden. Diese beziehen sich allerdings alle auf die Dateiansicht im rechten Rahmen, der sogenannten **Folder View**. Dabei können Sie mit einem Klick auf die Schaltfläche **Get** die momentane Konfiguration der **Folder View** ermitteln. Die Checkboxen werden dann gemäß der Ansicht markiert. Ändern Sie einzelne Markierungen, so wirken sich die neuen Einstellungen erst nach Klick auf den roten Button **Apply view settings** aus.

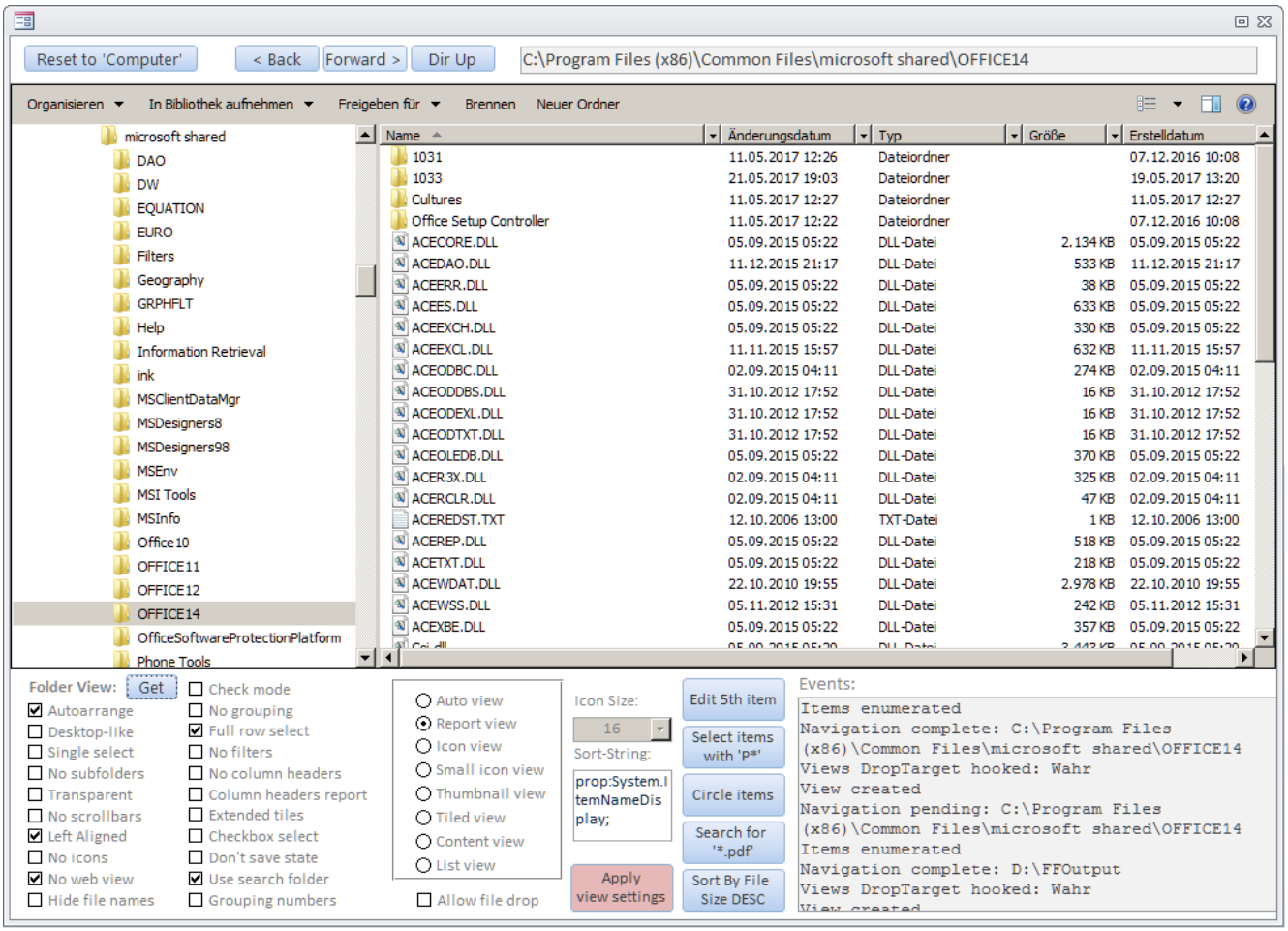


Bild 5: Im Formular **frmExplorerCtl** können unten verschiedene Einstellungen des **Explorer Controls** zur Laufzeit vorgenommen werden

OneNote 2016 und Access

Vor ein paar Ausgaben haben wir uns den Zugriff auf das Notizprogramm Evernote von Access aus angesehen. Leider bietet dieses populäre Programm keine echte VBA-Schnittstelle. Das ist beim Pendant des Herstellers Microsoft natürlich anders: Hier gibt es eine Objektbibliothek, die Sie in ein VBA-Projekt einbinden und mit dem Sie auf die in OneNote gespeicherten Daten zugreifen können. Im vorliegenden Beitrag schauen wir uns an, wie Sie per VBA auf die in OneNote gespeicherten Informationen zugreifen können.

Vor- und Nachteile

OneNote bietet keine offensichtlichen Nachteile im Vergleich zu Evernote, sondern eher noch Vorteile. Der gravierendste Vorteil ist, dass Sie, sofern Sie Office (zum Beispiel über Office 365) besitzen, keine zusätzlichen Kosten haben und der Speicherplatz für Ihre Notizen/Dokumente nicht begrenzt ist. Wenn Sie bisher mit Evernote gearbeitet haben, müssen Sie allerdings eine anders gestaltete Benutzeroberfläche in Kauf nehmen. Diese ist natürlich eher an die übrigen Office-Anwendungen angelehnt. Das hat auch einen gewissen Charme, denn so gewöhnen Sie sich recht schnell daran. Als Evernote-Benutzer fragen Sie sich vermutlich auch, was nun mit allen bisher in Evernote angelegten Notizen und Dokumenten geschieht. Das ist kein Problem, denn Microsoft bietet mit dem OneNote Importer ein Tool an, mit dem Sie beispielsweise Daten aus Evernote importieren können. Bei der von mir genutzten Evernote-Notizzettelsammlung, die beispielsweise auch Hunderte von Artikeln, Magazinen und Büchern im PDF-Format enthält, wurden nur einige wenige Elemente nicht importiert, die dann auch noch in einer Übersicht angezeigt werden. Die fehlenden Elemente kann man dann leicht selbst hinzufügen. In der Regel scheiterte der Import daran, dass die Elemente zu groß waren.

Ein Nachteil scheint jedoch die Wahl des Speicherortes zu sein: Während bei Evernote alle Informationen auch lokal auf dem Rechner gespeichert werden und etwa die angehängten PDF-Dateien in einem eigenen Ordner liegen, scheint dies bei OneNote nicht möglich zu sein.

Organisation in OneNote

In OneNote gibt es verschiedene Ebenen. Die oberste Ebene ist die der Notizbücher. Notizbücher legen Sie an und wählen Sie aus über den linken Reiter des mittleren Bereichs der Anwendung. Klappen Sie diesen auf, erhalten Sie den Befehl **Notizbuch hinzufügen** sowie eine Liste der aktuell geöffneten Notizbücher (s. Bild 1). Außerdem finden Sie dort einen Eintrag namens **Andere Notizbücher öffnen**, mit dem Sie die aktuell zwar in OneDrive gespeicherten, aber nicht geöffneten Notizbücher öffnen können. Wenn Sie aus Gründen der Übersicht ein Notizbuch schließen möchten, gelingt das am besten über

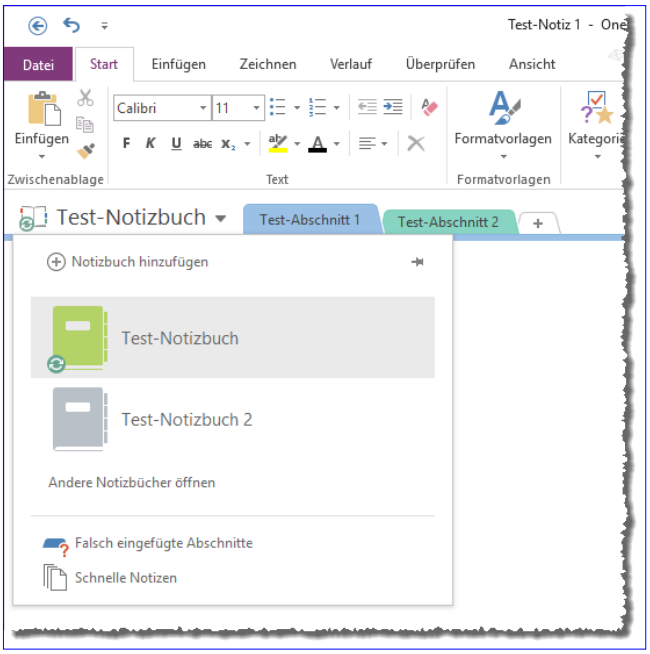


Bild 1: Notizbücher unter OneNote

den Kontextmenüeintrag **Notizbuch schließen** des Notizbuchs in dieser Ansicht.

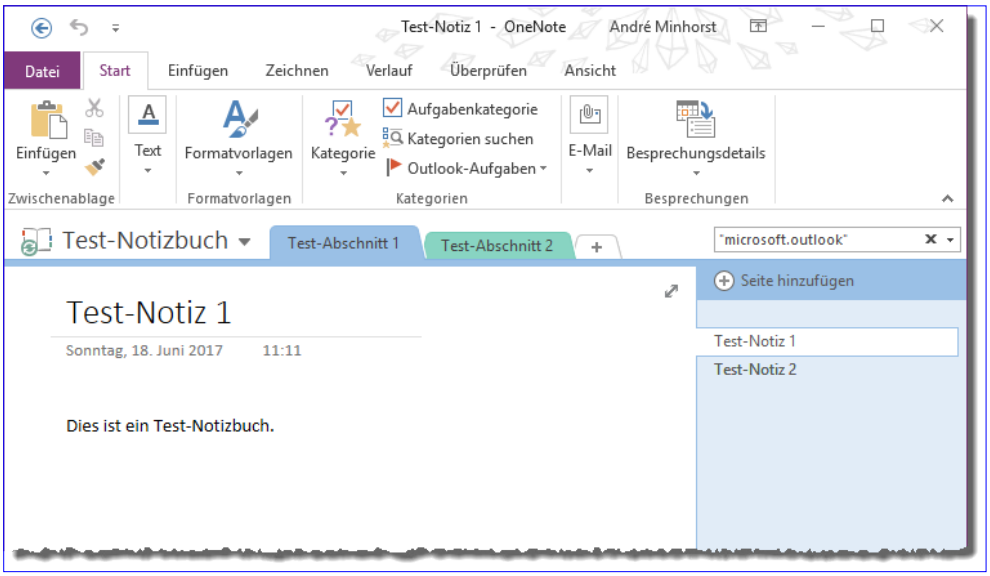


Bild 2: Bereiche eines Notizbuchs

Haben Sie ein Notizbuch geschlossen und öffnen dieses wieder, dauert dies je nach der Menge der enthaltenen Daten recht lange. Das liegt daran, dass OneNote wohl keine lokale Kopie der Daten bereithält und dass alle Inhalte immer aus dem Web geladen werden müssen. Hier arbeitet Evernote besser – die auf dem Rechner angelegten Daten werden lokal gespeichert und von dort in die Cloud synchronisiert, damit sie auch auf anderen Geräten abgerufen werden können. Zum Test des Zugriffs auf die Elemente von OneNote haben wir zunächst zwei Notizbücher namens **Test-Notizbuch 1** und **Test-Notizbuch 2** angelegt.

Abschnitte im Notizbuch

Die zweite Ebene der Notizen sind die Abschnitte. Diese erreichen Sie über die übrigen Registerkarten, die jeweils den Abschnittsnamen anzeigen. Für das Notizbuch **Test-Notizbuch** haben wir zwei Abschnitte namens **Test-Abschnitt 1** und **Test-Abschnitt 2** angelegt (s. Bild 2). Wenn Sie einen dieser Registerreiter anklicken, erscheint die jeweils erste zu diesem Abschnitt gespeicherte Notiz. Diese wird auch in der Liste aller Notizen dieses Abschnitts in der Liste auf der rechten Seite angezeigt. Einen neuen Abschnitt legen Sie per Mausklick auf den Registerreiter mit dem Plus-Symbol an **(+)**. Um den Namen einzustellen, wählen Sie das Kontextmenü des Registerreiters aus und klicken dort auf den Eintrag **Umbenennen**.

Notizen

Die Notizen bilden also die dritte Ebene in der Hierarchie von OneNote. Eine neue Notiz fügen Sie durch einen Klick auf den Befehl **Seite hinzufügen** im rechten Bereich des OneNote-Fensters hinzu. Die Notizen benennen Sie einfach durch Anpassen der Überschrift um, die über dem Datum angezeigt wird. Um einen Inhalt einzugeben, klicken Sie einfach auf den Bereich unter dem Datum und beginnen zu tippen. Sie können einer Notiz auch einfach Bilder oder andere Elemente wie PDF-Dokumente hinzufügen – oder Sie erstellen einfach eine Skizze (s. Bild 3).

Zugriff per VBA

Damit wären die Vorbereitungen erledigt und wir haben eine kleine Beispielstruktur für den Zugriff per VBA erstellt. Um tatsächlich mit VBA auf OneNote zuzugreifen, benötigen Sie noch einen Verweis auf die entsprechende Objektbibliothek.

Diesen fügen Sie im VBA-Projekt der Datenbankdatei, mit der Sie auf OneNote zugreifen wollen, über den **Verweise**-Dialog hinzu (Menüeintrag **Extras|Verweise**). Der Verweis heißt **Microsoft OneNote x.0 Object Library** (s. Bild 4).

Liste der Notizbücher einlesen

Damit kommen wir gleich zum ersten Beispiel – dem Einlesen der Liste aller Notizbücher von OneNote. Der dafür verwendete Code sieht wie folgt aus:

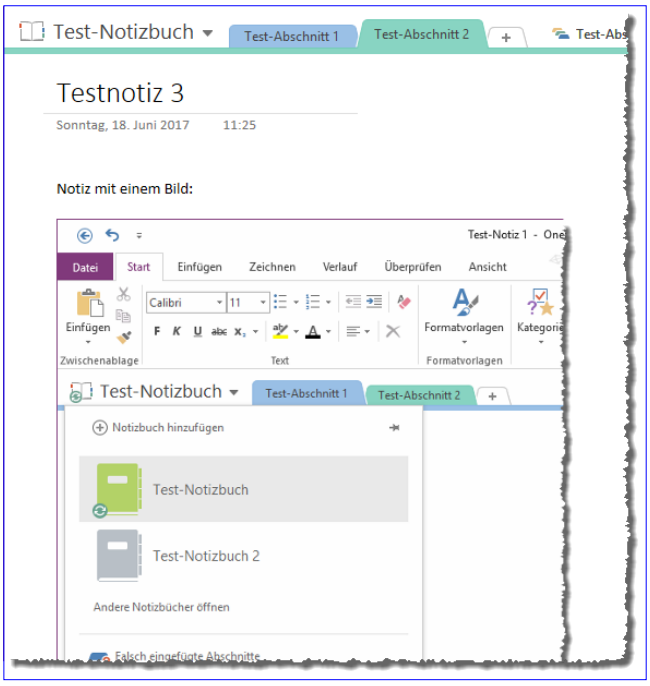


Bild 3: Notizbuch mit Bild

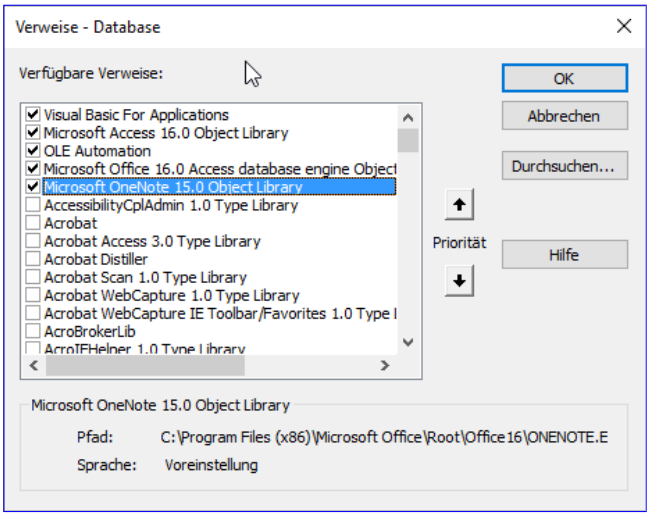


Bild 4: Verweis für den Zugriff auf OneNote

```
Public Sub OneNoteNotizbuecher()
    Dim objOneNote As OneNote.Application
    Dim strHierarchy As String
    Set objOneNote = New OneNote.Application
    objOneNote.GetHierarchy "", _
        OneNote.HierarchyScope.hsNotebooks, strHierarchy
    Debug.Print FormatXML(strHierarchy)
End Sub
```

Hier sind ein paar Erläuterungen notwendig. Wir erstellen als Erstes ein Objekt des Typs **OneNote.Application** und speichern es in einer Objektvariablen namens **objOneNote**. Dann rufen wir die Methode **GetHierarchy** dieses Objekts auf. Diese erwartet drei Parameter:

- **bstrStartNodeID**: Gibt die ID des übergeordneten Elements an. Wir geben eine leere Zeichenkette an, damit alle Notizbücher geliefert werden.
- **hsScope**: Gibt den Bereich an, der durchsucht werden soll. Wir wollen uns die Notebooks ansehen, also verwenden wir den Eintrag **hsNotebooks** der Auflistung **OneNote.HierarchyScope**.
- **pbstrHierarchyXMLOut**: Erwartet einen **String**-Parameter, mit dem das Ergebnis des Aufrufs zurückgegeben wird.

Das Ergebnis der Abfrage finden wir in Listing 1. Es handelt sich um ein XML-Dokument, das wir mithilfe der Funktion **FormatXML** in ein etwas besser lesbares Format gebracht haben (der eigentliche Rückgabewert enthält keine Zeilenumbrüche).

Nun stellen wir fest, dass das XML-Dokument lediglich zwei Einträge enthält. Dies ist für unseren Fall, da wir nur zwei Notizbücher angelegt haben, erwartungsgemäß, aber tatsächlich liegen auf dem Entwicklungsrechner einige Notizbücher mehr vor. Diese sind allerdings aktuell nicht geladen, was darauf hindeutet, dass nur die geladenen Notizbücher geliefert werden. Wenn wir nun auf die Attribute der einzelnen Elemente zugreifen wollen, müssen wir den VBA-Code etwas erweitern, wie Listing 2 zeigt. Hier verwenden wir Elemente der noch per Verweis einzubindenden Bibliothek **Microsoft XML 6.0 Object Library**, um auf die Inhalte des XML-Dokuments zuzugreifen.

Die Prozedur **OneNoteNotizbuecherXML** liest wie zuvor mit der Methode **GetHierarchy** das XML-Dokument mit den Notizbüchern in die Variable **strHierarchy** ein. Diese

```
<?xml version="1.0"?>
<one:Notebooks xmlns:one="http://schemas.microsoft.com/office/onenote/2013/onenote">
    <one:Notebook name="Test-Notizbuch 2" nickname="Test-Notizbuch 2" ID="{D6DCD364-35B7-4E68-ADE6-B4033A1C3EF2}{1}{B0}"
        path="https://d.docs.live.net/4cea18f6eeca8019/Documents/Test-Notizbuch 2/"
        lastModifiedTime="2017-06-18T09:35:38.000Z" color="#9595AA" isCurrentlyViewed="true"/>
    <one:Notebook name="Test-Notizbuch 1" nickname="Test-Notizbuch 1" ID="{68E5FF92-706E-4FD5-8218-40A4CE8D552E}{1}{B0}"
        path="https://d.docs.live.net/4cea18f6eeca8019/Documents/Test-Notizbuch 1/"
        lastModifiedTime="2017-06-8T10:16:39.000Z" color="#EE9597"/>
    <one:UnfiledNotes ID="{0B18F8FC-9EC3-4FF4-89E8-A15BBA1F7FDD}{1}{B0}"/>
</one:Notebooks>
```

Listing 1: Auflistung der Notizbücher im XML-Format

landet allerdings direkt über die Methode **loadXML** im Objekt **objXML** des Typs **DOMDocument60**. Hier müssen wir, da das komplette XML-Dokument den Namespace **one:** verwendet, diesen Namespace über die Eigenschaft **SelectionNamespaces** festlegen – anderenfalls können wir nicht mit der weiter unten verwendeten Methode **SelectNodes** auf die Elemente des XML-Dokuments zugreifen. Die Zuweisung erledigen wir mit der Methode **SetProperty**, der wir als erstes Argument den Namen der zu setzenden Eigenschaft, hier **SelectionNamespaces**, und zweitens die Definition des Namespaces übergeben, wie sie im Kopf des XML-Dokuments angegeben ist.

Danach können wir leicht in einer **For Each**-Schleife alle Elemente der Auflistung von XML-Elementen mit dem Namen **one:Notebook** durchlaufen. Diese ermitteln wir mit der Methode **selectNodes("//one:Notebook")**, welche zuverlässig alle Elemente mit dem Namen **one:Notebook** ermittelt. Innerhalb dieser Schleife geben wir dann die mit der Methode **getAttribut** ermittelten Eigenschaftswerte des jeweiligen Elements im Direkt-

```
Public Sub OneNoteNotizbuecherXML()
    Dim objOneNote As OneNote.Application
    Dim strHierarchy As String
    Dim objXML As MSXML2.DOMDocument60
    Dim objElement As MSXML2.IXMLDOMElement
    Dim strNamespace As String
    Set objOneNote = New OneNote.Application
    objOneNote.GetHierarchy "", OneNote.HierarchyScope.hsNotebooks, strHierarchy
    Set objXML = New MSXML2.DOMDocument60
    objXML.loadXML strHierarchy
    strNamespace = "xmlns:one=""http://schemas.microsoft.com/office/onenote/2013/onenote"""
    objXML.SetProperty "SelectionNamespaces", strNamespace
    For Each objElement In objXML.selectNodes("//one:Notebook")
        With objElement
            Debug.Print .getAttribute("name"), .getAttribute("ID"), .getAttribute("path"), _
                .getAttribute("lastModifiedTime"), .getAttribute("color"), .getAttribute("isCurrentlyViewed")
        End With
    Next objElement
End Sub
```

Listing 2: Auflistung der Notizbücher und Ausgabe der einzelnen Attribute