AUSGABE 02-03/2022

VISUAL BASIC ENTWICKLER

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

IN DIESEM HEFT:

VBA-BASICS: DER SCHNELLSTART

Lerne die Grundlagen von VBA kennen und sei so vorbereitet auf all die praktischen Erweiterungen, die wir in diesem Magazin vorstel-

SEITE 7

OUTLOOK PROGRAMMIEREN

Automatisiere die Outlook-Benutzeroberfläche, lerne das Application-Objekt kennen und verschicke E-Mails per Mausklick.

SEITE 28

YOUTUBE-VIDEOS VERWALTEN

Lerne die YouTube-API kennen und verwalte Deinen YouTube-Kanal und die enthaltenen Videos von Access, Excel und Co.

SEITE 14

André Minhorst Verlag

Herzlich willkommen!

Ich freue mich, dass Du Dich entschieden hast, das neue Magazin Visual Basic Entwickler als neuer Abonnent oder als bestehender Abonnent des Magazins Datenbankentwickler zu lesen. Der Visual Basic Entwickler wird einiges anders machen als der Datenbankentwickler. Der Fokus gilt nun der Programmierung von Office-Anwendungen mit verschiedenen Programmiersprachen wie VBA, VB.NET und twinBASIC. In dieser und den folgenden Ausgaben lernst Du, wie Du alle Aspekte der Office-Anwendungen wie Access, Excel, Outlook, PowerPoint und Word programmieren kannst – und sicher wird auch die eine oder andere Standalone-Anwendung entstehen, die Deinen Alltag mit Windows erleichtern soll.

Die erste Ausgabe ist direkt eine Doppelausgabe. Das ist praktisch, weil wir so für den Einstieg etwas mehr Platz haben. Was Dich in der ersten Ausgabe erwartet, liest Du übersichtlich auf der Seite U2. Dort findest Du auch die Zugangsdaten für den Download der Ausgabe im PDF-Format sowie die Beispieldateien, falls vorhanden. Das Thema dieses Editorials soll ein Ausblick darauf sein, was Dich in den folgenden Ausgaben erwartet.

Das Wichtigste vorneweg: Auch als absoluter Einsteiger bist Du bei diesem Magazin genau richtig. Wir werden in den ersten Ausgaben zunächst eine Einführung in die Entwicklung mit der Programmiersprache für die Office-Anwendungen, nämlich **Visual Basic for Applications** (**VBA**), liefern. Die hier verwendete Syntax ist fast identisch mit der von **Visual Basic .NET** und **twinBASIC**.

Außerdem schauen wir uns an, wie wir Funktionen zu den Office-Anwendungen und -Dokumenten hinzufügen können. Unter Excel, PowerPoint und Word kannst Du Dokumente um Automatismen erweitern, mit Access Funktionen zu einer Datenbank hinzufügen und auch Outlook lässt sich mit VBA programmieren und an Deine Bedürfnisse anpassen.

Um diese Anwendungen und die jeweiligen Dokumente mit VBA-Code versehen und sie damit automatisieren zu können, benötigst Du einen Überblick über die wichtigsten Elemente der jeweiligen Anwendungen und wie Du diese per VBA addressieren kannst. So erlaubt beispielsweise Outlook das programmatische Erstellen von E-Mails oder anderen Elementen und Du kannst fast alle Aktionen, die der Anwender über die Benutzeroberfläche ausführt, durch Programmierung beeinflussen. Und Du kannst natürlich neue Funktionen hinzufügen.

Damit Du alle gewünschten Erweiterungen programmieren kannst, schauen wir uns nach und nach die Objektmodelle der Office-Anwendungen an und entwickeln Lösungen, welche diese per VBA nutzen.

Für manche Erweiterung reichen die Bordmittel von Office, sprich die Programmierung per VBA in den jeweiligen VBA-Projekten, nicht aus. Dann fügen wir die benötigten Funktionen in Form von COM-Add-Ins oder COM-DLLs hinzu. Beide können wir sowohl mit VB.NET als auch mit twinBASIC programmieren. twinBASIC ist ein potenzieller Nachfolger von VB6. Mit COM-Add-Ins fügen wir einer Anwendung neue Funktionen hinzu und stellen diese beispielsweise über das Ribbon zur Verfügung. COM-DLLs bauen wir beispielsweise mit VB.NET, weil dieses viel mehr Erweiterungsmöglichkeiten bietet als VBA. Ein Beispiel dafür ist das Paket für den Zugriff auf YouTube, den wir in der aktuellen Ausgabe vorstellen.

Nun viel Spaß beim Lesen!

Ihr André Minhorst

Makros in Office aktivieren

Wenn wir in den Anwendungen eines frisch installierten Office-Pakets VBA-Code ausführen wollen, führt dies mitunter nicht zum gewünschten Ergebnis. Stattdessen erscheint eine Meldung, die uns mitteilt, dass die Makros in diesem Projekt deaktiviert sind. Der Grund ist einfach: Mit VBA-Code können wir eine Menge Schaden anrichten, zum Beispiel Dateien oder Verzeichnisse löschen. Daher ist seine Ausführung standardmäßig deaktiviert. Wie wir den VBA-Code dennoch ausführen können, zeigt dieser Artikel.

Neulich wollte ich in einer frischen Office-Installation das Ereignis ausprobieren, das beim Starten von Outlook ausgelöst wird und das wir im VBA-Projekt von Outlook (zu öffnen von Outlook aus mit der Tastenkombination **Strg + F11**) wie folgt zum standardmäßig vorhandenen Modul **ThisOutlookSession** hinzufügen:

```
Private Sub Application_Startup()
    MsgBox "Startup"
End Sub
```

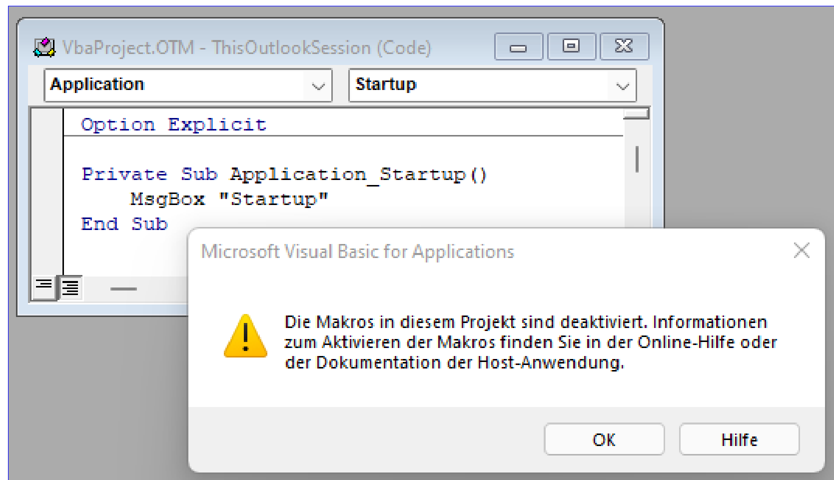

Bild 1: Meldung beim Versuch, eine VBA-Prozedur aufzurufen

Wider Erwarten führt ein Schließen und erneutes Öffnen von Outlook nicht zur Anzeige des Meldungsfensters. Also versuchte ich, die Prozedur direkt vom VBA-Editor aus aufzurufen, indem ich die Einfügemarke darin platzierte und auf **F5** drückte. Das Ergebnis war die Meldung aus Bild 1. Da die Onlinehilfe in diesem Fall recht wenig hilfreich war, ist dieser Artikel entstanden.

Ausführung von Makros aktivieren

Es gibt verschiedene Einstellungen für die Ausführung von Makros. Diese finden wir, wenn wir die Optionen für die jeweilige Office-Anwendung des aktu-

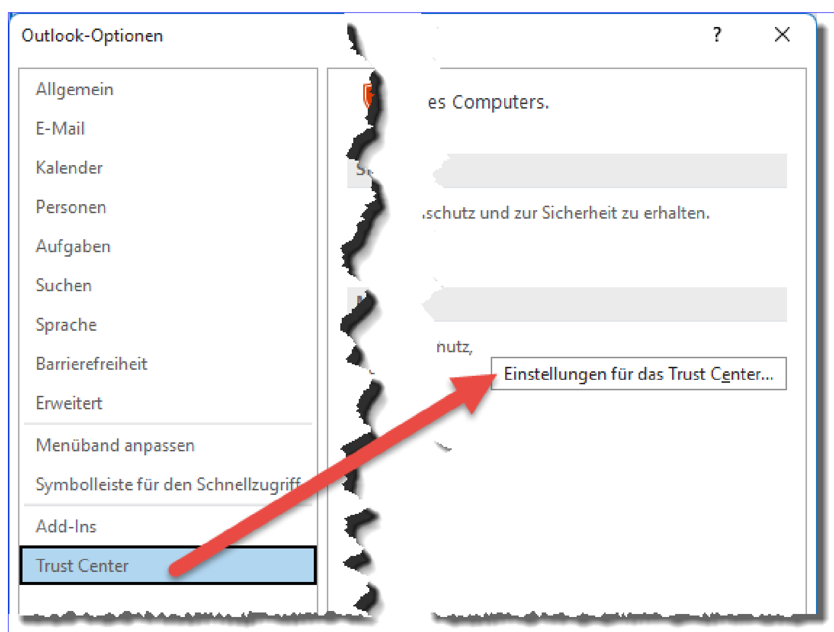

Bild 2: Anzeigen des Trust Centers

ell installierten Pakets öffnen. Dazu klicken wir im Ribbon auf **Datei** und im nun erscheinenden Bereich auf **Optionen**. Hier wechseln wir zum Bereich **Trust Center** und klicken dort auf die Schaltfläche **Einstellungen für das Trust Center...** (siehe Bild 2).

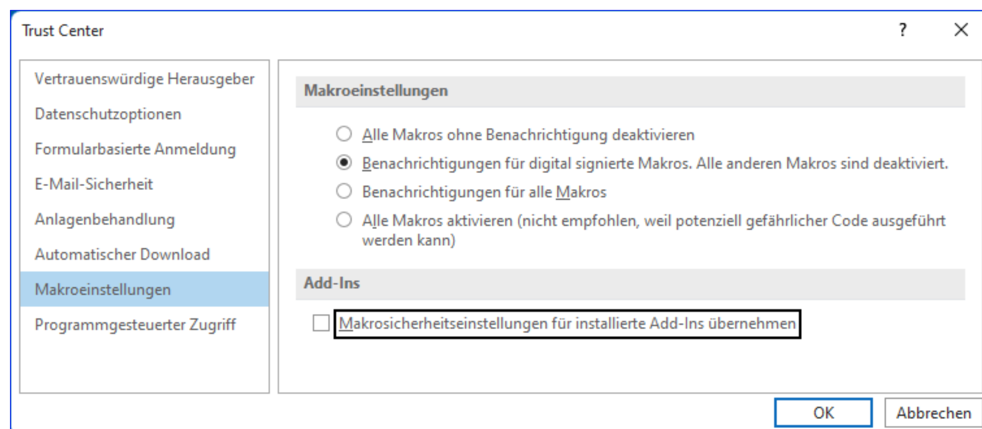

Bild 3: Die Makroeinstellungen für die Office-Anwendungen

Im nun erscheinenden Dialog **Trust Center** wechseln wir zum Bereich **Makroeinstellungen**. Hier finden wir für Microsoft Outlook beispielsweise standardmäßig die Einstellungen aus Bild 3 vor. Mit dem Wert **Benachrichtigungen für digital signierte Makros. Alle anderen Makros sind deaktiviert.** ist es kein Wunder, dass unsere VBA-Prozeduren nicht ausgeführt werden.

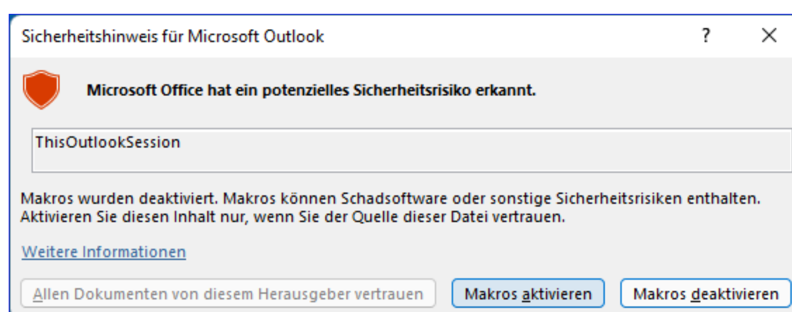

Bild 4: Diese Meldung erscheint, wenn Benachrichtigungen für alle Makros aktiviert ist und wir versuchen, eine VBA-Prozedur aufzurufen.

Es gibt zwei Alternativen, die ein Ausführen erlauben:

- **Benachrichtigungen für alle Makros**: Dies führt dazu, dass wir, wenn wir eine VBA-Prozedur aufrufen wollen, die Meldung aus Bild 4 erhalten. Bestätigen wir die Meldung mit **Makros aktivieren**, werden alle VBA-Prozeduren ausgeführt, die im Rahmen dieser Session automatisch oder manuell gestartet werden. Der Klick auf **Makros deaktivieren** führt dazu, dass innerhalb dieser Session keine VBA-Routinen ausgeführt werden können. Session bedeutet hier die Zeit vom Starten bis zum Schließen von Outlook.
- **Alle Makros aktivieren**: Mit dieser Einstellung werden alle VBA-Routinen ohne weitere Rückfragen ausgeführt.

Unterschiedliches Verhalten bei verschiedenen Anwendungen

Die verschiedenen Office-Anwendungen bieten bei Aktivierung der Option **Benachrichtigungen für alle Makros** teils unterschiedliche Möglichkeiten und zeigen unterschiedliche Verhaltensweisen:

- Unter Outlook reicht es bereits aus, das VBA-Projekt zu öffnen, um die Abfrage zu provozieren, ob die Makros aktiviert oder deaktiviert werden sollen.
- Unter Word, Excel und PowerPoint müssen die Dokumente die Dateiendung für Dokumente mit Makros aufweisen, also beispielsweise **.docm**, **.xlsm** oder **.ppxm**. Nur dann kann Code überhaupt automatisch ausgeführt werden und die Sicherheitseinstellungen für Makros greifen. In **.xlsx**-Dateien bei-

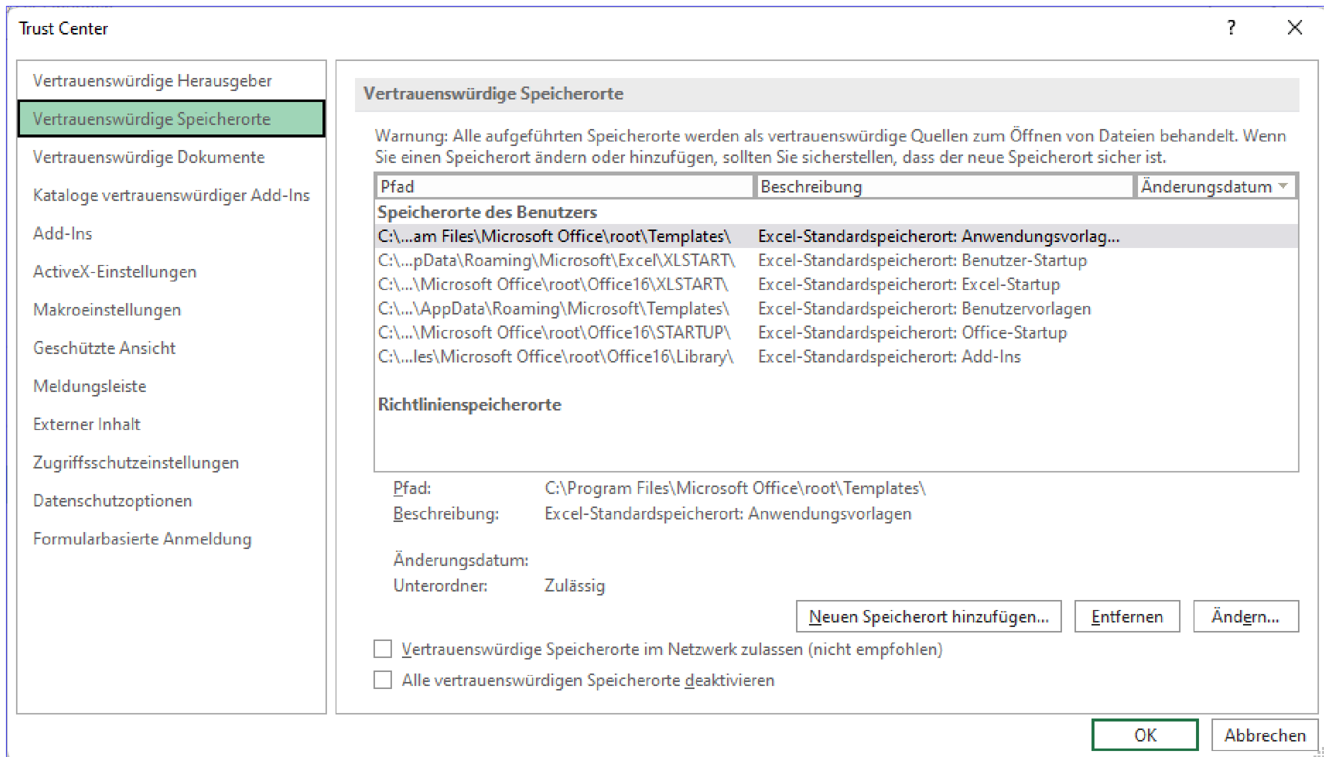

Bild 5: Verwalten vertrauenswürdiger Speicherorte

spielsweise kann VBA-Code über den VBA-Editor aufgerufen werden.

- Access-Datenbanken können immer Code ausführen, dazu muss die Datenbank nicht unter einer anderen Dateiendung gespeichert werden.

Die Einstellungen im Bereich **Makroeinstellung** werden übrigens für jede Office-Anwendung separat in der Registry gespeichert, und sie werden noch nicht einmal in jeder Anwendung gleich genannt – in Microsoft Excel lauten die Bezeichnungen etwas anders als in den übrigen Anwendungen. Wir können also für Access, Excel, Outlook, PowerPoint und Word separat festlegen, wie dort mit der Ausführung von VBA-Code umgegangen werden soll.

Vertrauenswürdige Speicherorte als Ausweg

Wenn Du nicht soweit gehen willst, die Einstellung **Alle Makros aktivieren** zu nutzen, sondern **Benachrichtigungen für alle Makros**, kannst Du den VBA-Code in Dokumenten immer noch ohne Anzeige von Sicherheitsmeldungen nutzen. Dazu fügst Du im Dialog **Trust Center** unter **Vertrauenswürdige Speicherorte** den oder die Ordner hinzu, welche die Dokumente enthalten, die ohne Rückfrage ausgeführt werden sollen (siehe Bild 5).

Sollte der gewünschte vertrauenswürdige Speicherort noch nicht in der Liste enthalten sein, kannst Du diesen mit einem Klick auf **Neuen Speicherort hinzufügen...** zur Liste hinzufügen.

VBA-Basics: Variablen

Variablen sind die Möglichkeit unter VBA, bestimmte Werte wie Zahlen, Texte, Datumsangaben oder auch komplexere Elemente wie Verweise auf Objekte der Benutzeroberfläche der Office-Anwendungen oder auf selbst erstellte Objekte für den Zeitraum der Benutzung zu speichern und wieder abrufen zu können. Kurz gesagt: Mit Variablen merken wir uns verschiedene Dinge, die wir noch mal brauchen können. Dieser Artikel stellt die verschiedenen Basisdatentypen für Variablen vor und zeigt, wie wir diese deklarieren, mit Werten oder Verweisen füllen und diese wieder abfragen. Außerdem gehen wir auch auf den Gültigkeitsbereich, die Gültigkeitsdauer und die Frage der Benennung von Variablen ein.

Voraussetzung

Im ersten Teil dieser Artikelreihe namens **VBA-Basics: Schnellstart** (**www.vbentwickler.de/318**) haben wir gezeigt, wie Du ein Modul in einem VBA-Projekt einer Office-Anwendung anlegst.

Im vorliegenden Artikel bauen wir darauf auf und gehen davon aus, dass Du in irgendeiner Office-Anwendung bereits ein Modul erstellt hast, das wir zum Experimentieren mit Variablen nutzen können.

Was sind Variablen?

Falls Du noch gar nicht weißt, was eine Variable ist, hier eine Erklärung: Eine Variable ist eine Art Behälter für einen Wert, dem wir einen Namen geben, um darüber den Behälter füllen und auslesen zu können.

Neben dem Namen legen wir in der Regel auch noch fest, welcher Art die Werte sind, welche die Variable aufnehmen können soll, also zum Beispiel Texte, Zahlen verschiedener Wertebereiche und Genauigkeiten, Datumsangaben oder Ja-/Nein-Werte.

Variablen können aber auch komplexe Objekte enthalten. Diese werden dann faktisch nicht in der Variablen gespeichert, sondern in der Variable landet nur ein Verweis auf die Speicherstelle des komplexen Objekts. Doch dazu später mehr.

Variable deklarieren

Damit wir sehen, was eine Variable kann und was nötig ist, um diese überhaupt zu nutzen, verwenden wir das folgende Beispiel. Die wichtigste Voraussetzung für die Verwendung einer Variablen ist die Deklaration. Durch die Deklaration legen wir für die Variable den Namen fest, die zulässigen Werte beziehungsweise den Datentyp sowie den Gültigkeitsbereich und in gewisser Weise auch die Gültigkeitsdauer.

Modul als Voraussetzung

Im ersten Teil dieser Artikelreihe haben wir schon gesehen, wie wir ein Modul zu einem VBA-Projekt hinzufügen. Ein solches Modul benötigen wir auf jeden Fall, um eine Variable zu deklarieren – wo sonst sollen wir diese hinschreiben?

String-Variable deklarieren

Also schreiben wir einfach einmal die erste Variable mit dem Datentyp **String** in dieses Modul. Diese Variable soll den Namen **strText** erhalten:

```
Dim strText As String
```

Dies liefert die folgenden Informationen:

- Die **Dim**-Anweisung gibt an, dass nun die Deklaration einer Variablen folgt.

- **strText** ist der Name der Variablen.
- Das Schlüsselwort **As** leitet die Angabe des Datentyps ein.
- **String** ist der Datentyp der Variablen. Das heißt, die Variable kann Zeichenketten aufnehmen.

Was können wir nun mit dieser Variablen anfangen? Wir können sie beispielsweise mit einem Wert füllen, zum Beispiel der Zeichenkette **Beispieltext**.

Da es sich hierbei um eine Zeichenkette handelt, müssen wir sie in Anführungszeichen einfassen. Dann können wir den Wert im Direktbereich des VBA-Editors der Variablen **strText** zuweisen:

```
strText = "Beispieltext"
```

Danach lassen wir uns den Wert im Direktbereich mit der **Debug.Print**-Anweisung ausgeben:

```
Debug.Print strText
```

Das Ergebnis sehen wir in Bild 1.

Private oder Public?

Um den Gültigkeitsbereich einer Variablen festzulegen, haben wir verschiedene Möglichkeiten. Die erste ist der Ort, an dem wir die Variable festlegen.

Die zweite ist das Schlüsselwort, mit dem wir die Deklaration einleiten. Wir können grundsätzlich zwischen den folgenden drei Gültigkeitsbereichen unterscheiden:

- Innerhalb einer Prozedur
- Innerhalb eines Moduls
- Im kompletten VBA-Projekt

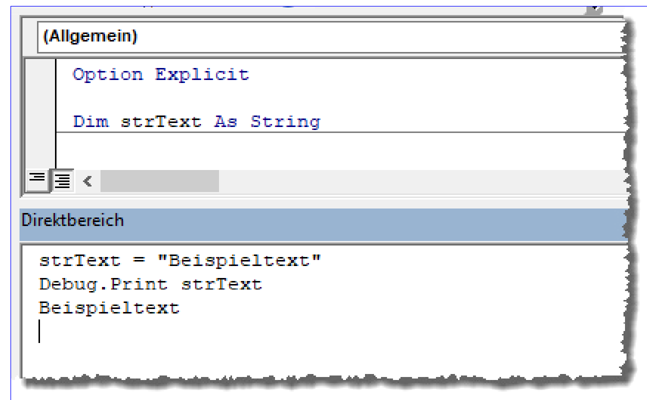

Bild 1: Testen einer Variablen

Variablen nur innerhalb von Prozeduren nutzen

Wenn wir eine Variable nur innerhalb einer Prozedur verwenden wollen, dann deklarieren wir sie genau in dieser Prozedur, also beispielsweise wie folgt:

```
Public Sub ProzedurMitVariable()
    Dim strProzedur As String
End Sub
```

Wir können nun nur noch innerhalb der Prozedur auf diese Variable zugreifen – beispielsweise, indem wir diese mit einem Wert füllen und den Wert der Variablen im Direktbereich ausgeben:

```
Public Sub ProzedurMitVariable()
    Dim strProzedur As String
    strProzedur = "Beispieltext"
    Debug.Print strProzedur
End Sub
```

Nun könntest Du versuchen, die innerhalb dieser Prozedur deklarierte Variable vom Direktbereich aus zu füllen und abzurufen, indem Du diese im Direktbereich ausgibst:

```
strProzedur = "Beispieltext"
Debug.Print strProzedur
```

Das funktioniert zwar, aber der Direktbereich hat eigene Regeln, was die Gültigkeit von Elementen angeht. Viel wichtiger ist, dass wir diese Variable nicht von anderen Prozeduren aus setzen oder lesen können.

Das probieren wir aus, indem wir in einer anderen Prozedur im gleichen Modul versuchen, diese Variable zu setzen:

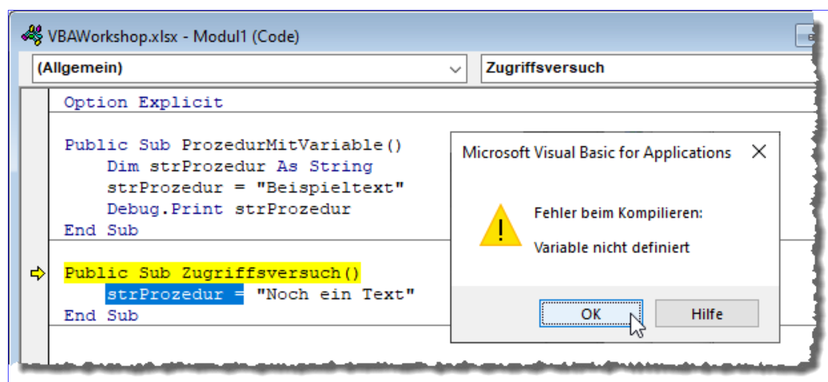

Bild 2: Versuch, auf eine private Variable zuzugreifen

```
Public Sub Zugriffsversuch()
    strProzedur = "Noch ein Text"
End Sub
```

Führen wir diese Prozedur aus, erhalten wir die Fehlermeldung aus Bild 2. Der Gültigkeitsbereich der Variablen **strProzedur** ist also auf die Prozedur selbst beschränkt.

Variablen nur innerhalb eines Moduls nutzen

Wenn wir eine Variable in einer Prozedur setzen und diese in einer anderen Prozedur auslesen wollen, dann können wir die Variable weder in der einen noch in der anderen Prozedur deklarieren. Vielmehr deklarieren wir diese dann im allgemeinen Teil des Moduls, was bedeutet: Hinter der letzten **Option...**-Anweisung und vor der ersten Prozedur.

Wir deklarieren die Variable wie folgt:

```
Dim strModulweit As String
```

Die erste Prozedur im gleichen Modul soll die Variable mit einem Wert füllen:

```
Public Sub VariableSetzen()
    strModulweit = "Modulweit"
End Sub
```

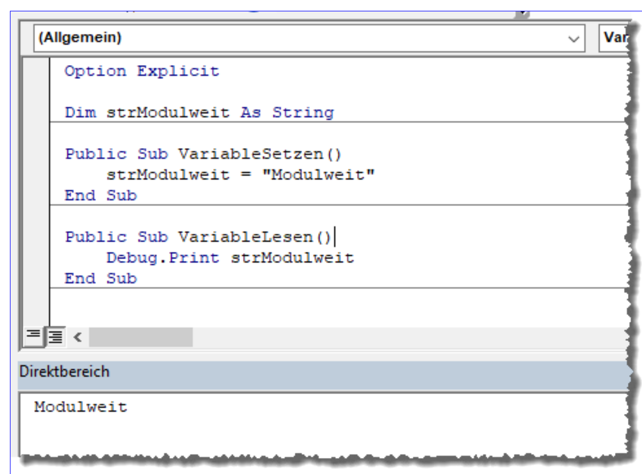

Bild 3: Einsatz einer modulweit gültigen Variablen

Die zweite Prozedur soll den Wert dieser Variablen im Direktbereich ausgeben:

```
Public Sub VariableLesen()
    Debug.Print strModulweit
End Sub
```

Wenn wir nun die erste und die zweite Prozedur nacheinander ausführen, landet der gesetzte Variablenwert wie in Bild 3 im Direktbereich.

Variablen projektweit deklarieren

Einen noch größeren Gültigkeitsbereich erreichen wir nicht, indem wir eine Variable an einem ganz speziellen Ort für projektweit deklarierte Variablen speichern

Office-Ereignisse mit VBA programmieren

Visual Basic for Applications, kurz VBA, ist eine Ableitung der Programmiersprache Visual Basic. Im Gegensatz zu diesem, das mit Visual Basic 6 vor einigen Jahren seine letzte Version erlebt hat, erfreut sich VBA immer noch reger Beliebtheit. Das liegt vor allem an seinem Hauptzweck: der Programmierung und der Steuerung von automatisierten Abläufen in den Office-Anwendungen, zum Beispiel Access, Excel, Outlook, PowerPoint oder Word. Der Kern dieser Automatisierungen sind sogenannte Ereignisse, also Prozeduren, die an einer bestimmten Stelle unter einem vordefinierten Namen hinzugefügt und dann durch das jeweilige Ereignis automatisch ausgelöst werden – beispielsweise dem Öffnen eines Dokuments oder dem Anklicken einer Schaltfläche. Dieser Artikel zeigt, welche Möglichkeiten die einzelnen Office-Anwendungen zum Einrichten von Ereignisprozeduren bieten und was bei der Automatisierung grundlegend zu beachten ist.

Öffnen des VBA-Editors

Der dazu verwendete VBA-Editor steht zum Beispiel in den oben genannten Anwendungen wie Access, Excel, Outlook, PowerPoint und Word standardmäßig zur Verfügung und lässt sich schnell mit der Tastenkombination **Alt + F11** öffnen. Unter Access gibt es sogar noch eine weitere Tastenkombination, um vom Hauptfenster der Anwendung zum VBA-Editor zu gelangen, nämlich **Strg + G**. Dies aktiviert direkt den Direktbereich des VBA-Editors.

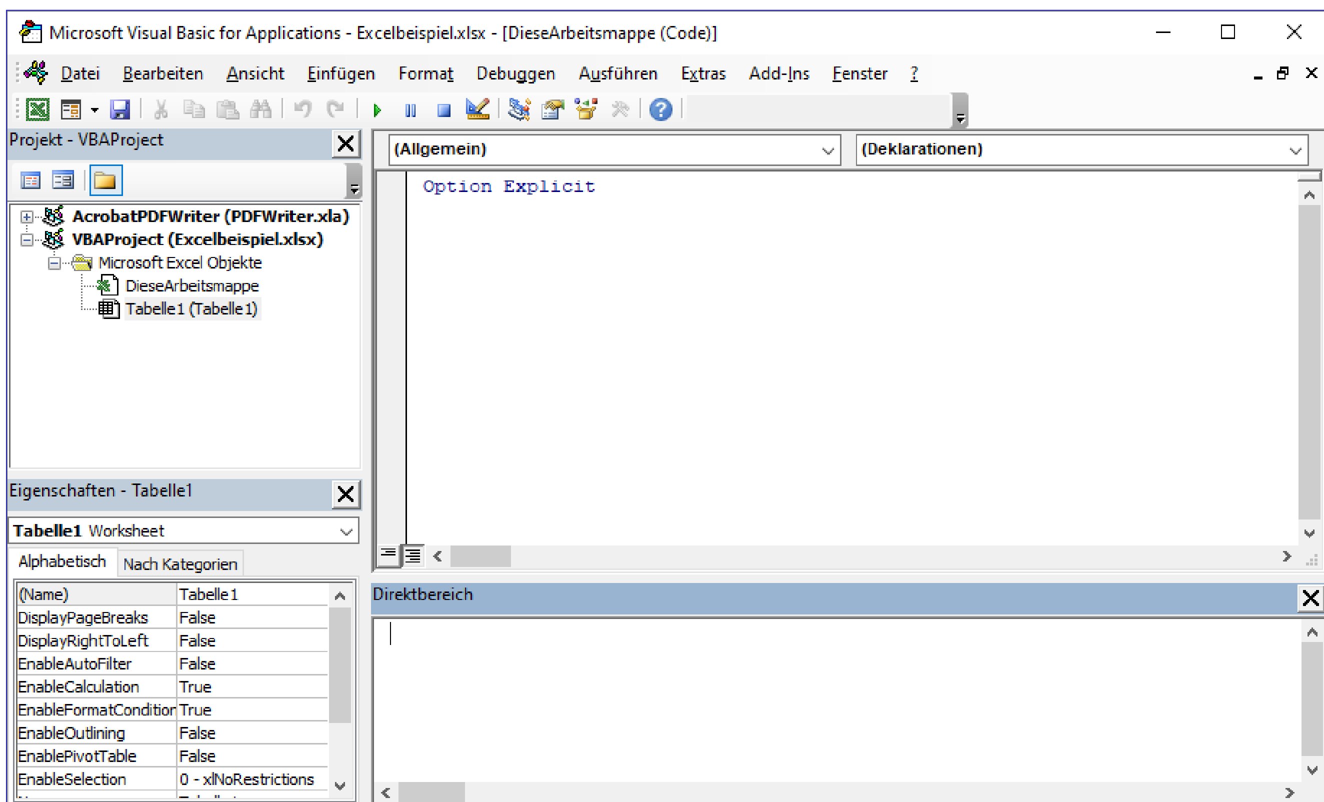

Bild 1: Der VBA-Editor

VBA-Editor für eine Excel-Datei

Der VBA-Editor sieht für alle Office-Anwendungen ähnlich aus. Bild 1 zeigt den VBA-Editor für eine frisch erstellte Excel-Datei.

Der Unterschied bei der Darstellung des VBA-Editors für die verschiedenen Office-Anwendungen liegt im Wesentlichen im Aufbau der VBA-Projekte der jeweiligen Datei im Projekt-Explorer, den wir mit dem Menüeintrag **Ansicht|Projekt-Explorer** oder der Tastenkombination **Strg + R** öffnen.

In der Abbildung sehen wir beispielsweise das VBA-Projekt für eine Excel-Datei. Das Projekt erhält unter Excel standardmäßig den Namen **VBAProject**, in Klammern sehen wir dahinter den Namen, unter dem wir die Exceldatei gespeichert haben, in diesem Fall **Excelbeispiel**.

Im VBA-Projekt einer neuen, leeren Excel-Datei finden wir zwei Elemente. Bei diesen handelt es sich um Klassenmodule, die wir wie folgt nutzen können:

- **DieseArbeitsmappe**: Dabei handelt es sich um das Klassenmodul für ein Objekt namens **Workbook**, welches das Excel-Dokument selbst repräsentiert, das wiederum eine oder mehrere Tabellen enthält. Für das **Workbook**-Objekt können wir verschiedene Ereignisse definieren, die zu verschiedenen Zeitpunkten ausgelöst werden – zum Beispiel beim Öffnen des Excel-Dokuments, beim Aktivieren von Tabellen oder auch vor dem Schließen des Dokuments.
- **Tabelle1 (Tabelle1)**: Dies ist ein Klassenmodul für die standardmäßig angelegte Tabelle namens **Tabelle1**. Dieses bietet ebenfalls verschiedene Möglichkeiten zum Definieren von Ereignisprozeduren an. Diese werden beispielsweise beim Ändern der Auswahl in der Excel-Tabelle ausgelöst oder beim Aktivieren der aktuellen Tabelle.

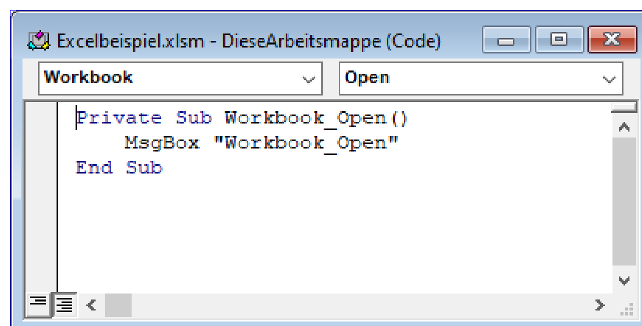

Bild 2: Ereignisprozedur für ein Excel-Workbook

Standardmäßig zeigt Excel das Element **DieseArbeitsmappe** beim Öffnen des VBA-Editors für eine neue Excel-Datei an.

Ereignis beim Öffnen eines Excel-Workbooks

Wir wollen uns an einem Beispiel ansehen, wie ein Ereignis eines Excel-Workbooks funktioniert. Dazu klicken wir doppelt auf den Eintrag **DieseArbeitsmappe** des Projekts des aktuell geöffneten Excel-Workbooks und öffnen so das Klassenmodul **DieseArbeitsmappe**.

Hier wählen wir nun mit dem linken Kombinationsfeld im Codefenster den Eintrag **Workbook** aus, was automatisch im rechten Kombinationsfeld das standardmäßig festgelegte Ereignis für diese Klasse selektiert und die Ereignisprozedur **Workbook_Open** zum Codefenster hinzufügt. Diese ergänzen wir um die folgende Anweisung:

```
Private Sub Workbook_Open()
    MsgBox "Workbook_Open"
End Sub
```

Im VBA-Editor sieht das nun wie in Bild 2 aus.

Damit wir dieses Ereignis auslösen können, müssen wir das Workbook schließen und erneut öffnen. Beim Schließen erscheint allerdings direkt eine Meldung, die besagt, dass wir das VBA-Projekt nicht mit der Ar-

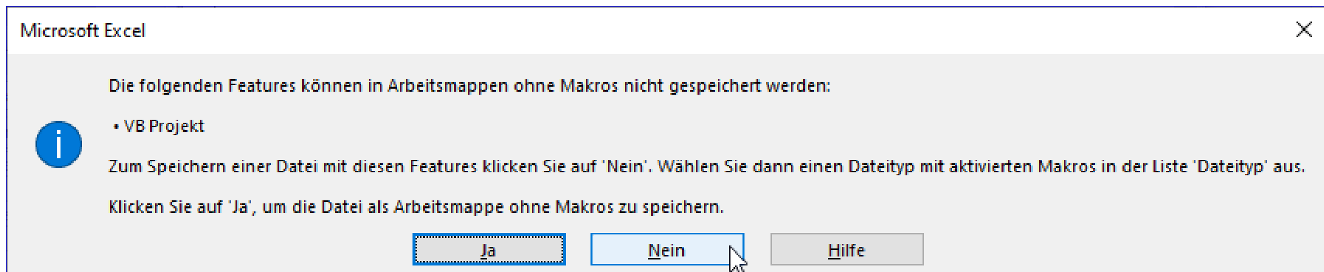

Bild 3: Meldung beim Versuch, ein Excel-Dokument mit VBA-Code zu speichern

beitsmappe speichern können (siehe Bild 3).

Das liegt allerdings nur an dem aktuellen Format, das durch die Dateiendung **.xlsx** repräsentiert wird. Also klicken wir, wie in der Meldung vorgeschlagen, auf die Schaltfläche **Nein** und wählen im anschließend erscheinenden **Speichern unter**-Dialog den Datentyp **Excel-Arbeitsmappe mit Makros (*.xlsm)** aus (siehe Bild 4).

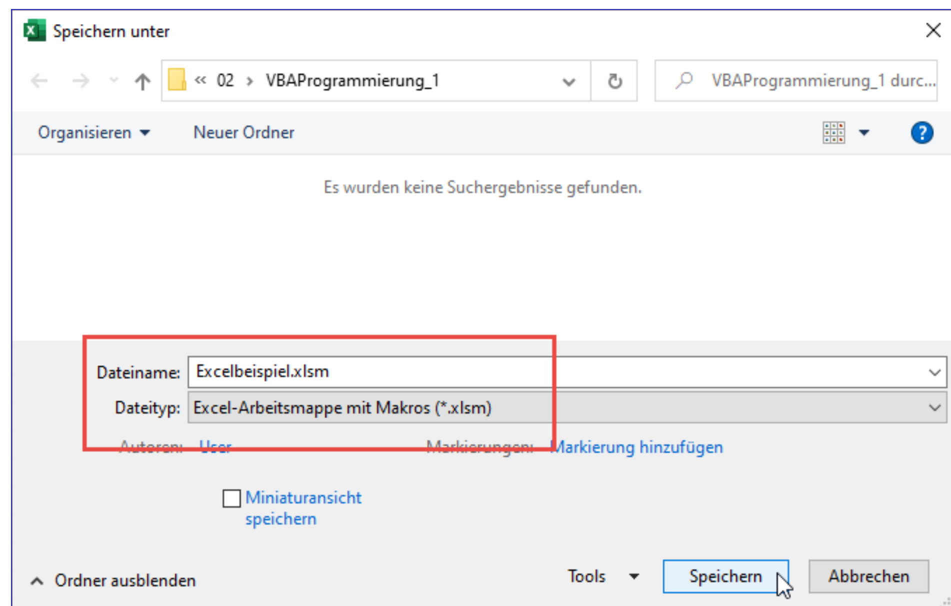

Bild 4: Auswahl eines Dateityps, der das Speichern des VBA-Projekts erlaubt

Schließen wir die Datei nun und öffnen diese anschließend erneut, zeigt Excel eine Sicherheitswarnung an (siehe Bild 5). Hier klicken wir auf Inhalt aktivieren, damit der VBA-Code im enthaltenen VBA-Projekt ausgeführt werden kann.

Danach erscheint dann die von uns weiter oben programmierte Meldungsbox mit dem Namen der auslösenden Ereignisprozedur (siehe Bild 6).

Achtung, Code-Verlust!

Wichtig ist, dass wir die Datei – genau wie bei Word oder PowerPoint – nicht in einem Dateiformat ohne Makros speichern, also wie gehabt als **.xlsx**, **.docx** oder **.pptx**, und das Dokument dann schließen. Wenn wir bereits Code zum VBA-Projekt des Dokuments hinzugefügt haben und das Dokument dann nicht als **.xlsm**, **.docm** oder **.pptm** speichern, wird das VBA-Projekt einfach verworfen.

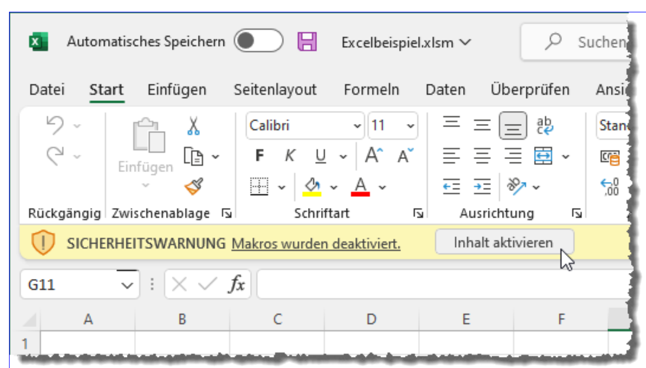

Bild 5: Sicherheitswarnung beim Öffnen eines Excel-Workbooks mit VBA-Projekt

Direkt zugreifbare Klassen

Wenn Du schon einmal mit Klassen programmiert hast, weißt Du, dass diese üblicherweise deklariert und instanziert werden müssen.

Das ist bei der Klasse **Workbook**-Klasse und auch bei der anschließend vorgestellten **Worksheet**-Klasse nicht nötig – diese sind bereits implizit initialisiert und Du kannst sie direkt nutzen.

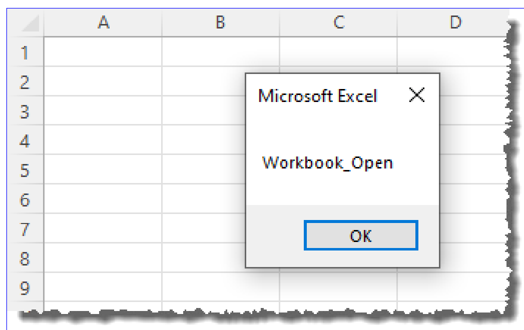

Bild 6: Meldung durch eine VBA-Prozedur beim Öffnen eines Excel-Workbooks

Ereignis für die Workbook-Klasse

Auf die gleiche Weise können wir die übrigen Ereignisprozeduren der **Workbook**-Klasse implementieren sowie die Ereignisse der einzelnen Worksheets beziehungsweise Tabellen.

Das Standardereignis der **Worksheet**-Klasse lautet übrigens **SelectionChange**. Auch dieses wollen wir noch ausprobieren. Die dafür angelegte Ereignisprozedur liefert mit einem Parameter namens **Target** ein Objekt des Typs **Range**, mit dem wir den markierten Bereich ermitteln können.

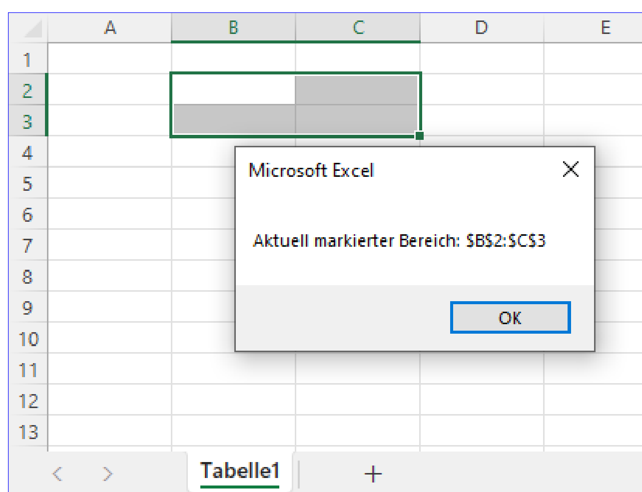

Bild 7: Meldung mit der Ausgabe des aktuell markierten Bereichs

Das **Range**-Objekt liefert mit der Eigenschaft **Address** die Angabe der Adresse des markierten Bereichs.

Diesen wollen wir uns nach jeder Auswahl im Worksheet **Tabelle 1** in einem Meldungsfenster ausgeben lassen und hinterlegen zu diesem Zweck die folgende Ereignisprozedur:

```
Private Sub Worksheet_SelectionChange(ByVal Target _
        As Range)
    MsgBox "Aktuell markierter Bereich: " & Target.Address
End Sub
```

Damit erhalten wir nach jeder neuen Auswahl im Tabellenblatt eine Meldung wie die aus Bild 7.

Natürlich können wir im VBA-Projekt eines Excel-Workbooks noch viel mehr Dinge erledigen als nur Ereignisse implementieren – dazu in späteren Artikeln mehr.

Word-Dokumente

Wenn wir ein neues Word-Dokument anlegen und den VBA-Editor von Word öffnen, sehen wir gleich zwei Projekte:

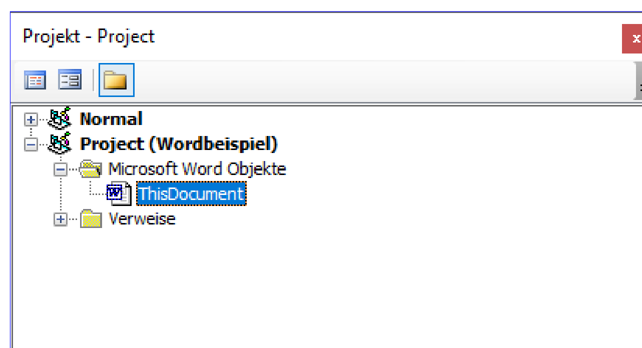

Bild 8: Projekt eines Word-Dokuments

- **Normal**: VBA-Projekt der Standardvorlage von Word
- **Project (Dateiname)**: VBA-Projekt der aktuell geöffneten Word-Datei

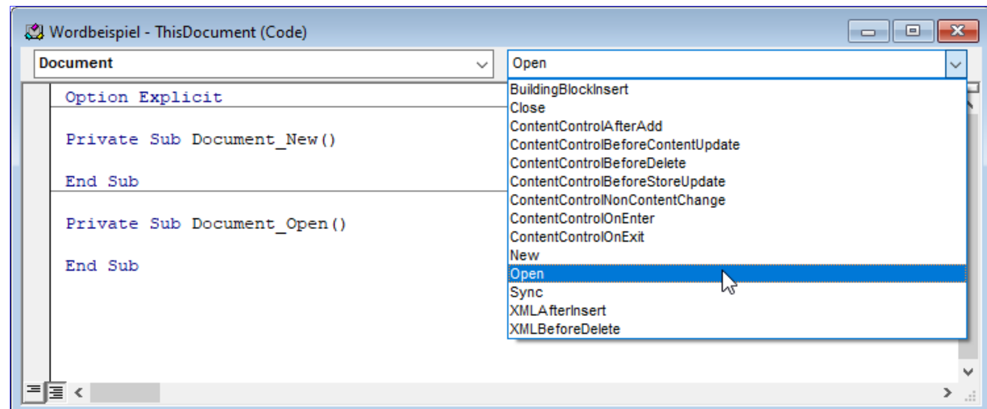

Bild 9: Anlegen eines Ereignisses für ein Word-Dokument

Wir interessieren uns zunächst für das VBA-Projekt zur aktuell geöffneten Datenbank, die in diesem Fall **Project (Wordbeispiel)** heißt. **Wordbeispiel.docx** ist der Name unseres Dokuments, **Project** ist in diesem Fall der Name des VBA-Projekts (siehe Bild 8). Diesen können wir allerdings auch jederzeit anpassen.

Im VBA-Projekt zu unserem Dokument finden wir zwei Ordner vor, von denen wir uns zunächst den namens **Microsoft Word Objekte** ansehen wollen. Hier finden wir ein Element vor, nämlich **ThisDocument**. Dies ist ein Klassenmodul, in dem wir VBA-Code einfügen können, der im Kontext dieses Dokuments ausgeführt werden soll – zum Beispiel beim Eintreten verschiedener Ereignisse.

Das Klassenmodul **ThisDocument** stellt eine Klasse namens **Document** zur Verfügung, die zum Beispiel Ereignisse anbietet, die beim Öffnen oder beim Schließen des Dokuments ausgelöst werden. Genau wie bei den beiden Excel-Klassen **Workbook** und **Worksheet** ist **Document** eine Klasse, die implizit initialisiert ist – wir können die Eigenschaften, Methoden und Ereignisse einfach so nutzen.

Wir probieren das auch hier anhand eines kleinen Beispiels aus, indem wir für das Ereignis beim Öffnen eines Dokuments eine Ereignisprozedur anlegen. Dazu öffnen wir das Modul **ThisDocument**, sofern dieses noch nicht geöffnet ist, durch einen Doppelklick auf den Eintrag **ThisDocument** im Projekt-Explorer. Es erscheint ein Codefenster mit einem komplett leeren Modul, in das wir nun Code eingeben können.

In diesem Fall wollen wir uns bei der Code-Eingabe jedoch Unterstützung holen und dazu wählen wir aus dem linken Kombinationsfeld oben im Codefenster den Eintrag **Document** aus. Dies legt automatisch die standardmäßig für diese Klasse vorgesehene Ereignisprozedur **Document_New** an.

Dieses Ereignis ist jedoch für normale Dokumente von begrenztem Nutzen: Es wird nur ausgelöst, wenn es sich bei dem Dokument um eine Vorlage handelt und wir ein neues Dokument auf Basis dieser Vorlage erstellen.

Weitere Ereignisprozeduren für die Klasse **Document** können wir dann, nachdem wir dieses im linken Kombinationsfeld selektiert haben, aus dem rechten Kombinationsfeld auswählen.

Darüber fügen wir noch eine weitere Ereignisprozedur namens **Document_Open** hinzu (siehe Bild 9).

Damit wir sehen, ob und wie diese Ereignisse funktionieren, fügen wir dem Ereignis **Document_Open** eine einfache Anweisung zum Anzeigen eines Meldungsfensters hinzu:

Outlook: Codebeispiele ausprobieren

In diesem Magazin stellen wir immer wieder Beispiele zur Programmierung von Outlook per VBA vor. Während wir solchen Beispielcode bei den anderen Anwendungen wie Word, Excel oder PowerPoint oder auch unter Access in entsprechenden Beispieldokumenten unterbringen können, müssen wir die Outlook-Beispiele in das VBA-Projekt von Outlook einfügen, um diese auszuprobieren. Wie das gelingt, zeigt der vorliegende Artikel.

Die Beispiele in unseren Artikeln zur Programmierung von Outlook über das VBA-Projekt von Outlook selbst enthalten manchmal nur einzelne Prozeduren, die Du direkt in ein beliebiges Standardmodul im VBA-Projekt von Outlook einfügen kannst – optimalerweise in ein neues, leeres Standardmodul, damit Du keine Module mit bestehendem Code beeinflusst, weil dann beispielsweise eine Variablendeklaration dort doppelt vorkommt.

Gelegentlich liefern wir aber auch komplette Module, die in zwei Typen aufgeteilt werden können:

- **Standardmodule**: Diese enthalten Routinen, die wir direkt starten können, sofern diese keine Parameter aufweisen. Routinen mit Parametern müssen wir beispielsweise über den Direktbereich starten und übergeben dabei die Werte für die Parameter.
- **Klassenmodule**: Klassenmodule müssen immer erst initialisiert und mit einer entsprechenden Objektvariablen referenziert werden, bevor wir die darin enthaltenen Methoden starten können. Eine Ausnahme ist das immer bereits vorhandene Klassenmodul **ThisOutlookSession**. Die hier aufgeführten Routinen können wir wie die in Standardmodulen direkt starten.

VBA-Routinen direkt starten

Was aber bedeutet »direkt starten« eigentlich? Damit meinen wir, dass Du die Routine (also die **Sub**- oder **Function**-Prozedur) im Code-Fenster im VBA-Editor markierst (dazu muss sich die Einfügemarke irgendwo innerhalb der Prozedur befinden) und dann entweder die Taste **F5** betätigst, den Menübefehl **Ausführen|Sub/Userform ausführen** aufrufst oder die gleichnamige Schaltfläche mit dem **Abspielen**-Symbol anklickst.

Damit kannst Du beispielsweise Prozeduren wie die Folgende Starten:

```
Public Sub Beispielaufruf
    MsgBox "Beispielaufruf"
End Sub
```

Das Ergebnis sieht wie in Bild 1 aus.

Wenn die **Sub**- oder **Function**-Prozedur jedoch einen oder mehrere Parameter enthält, kannst Du diese nicht

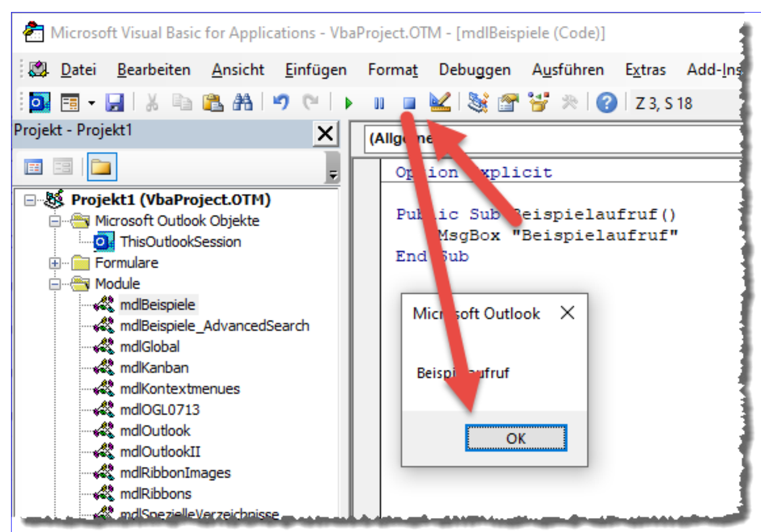

Bild 1: Starten einer einfachen **Sub**-Prozedur

mehr direkt starten. Versuchst Du es auf die gleiche Weise wie zuvor, erscheint wie in Bild 2 der Dialog **Makros**.

Hier ist nun eine von zwei Alternativen möglich:

- Wenn die Prozedur mit Parameter nur wenige Male ausprobiert werden soll, rufen wir diese über den Direktbereich auf.
- Wenn absehbar ist, dass die Prozedur mit Parametern des Öfteren aufgerufen werden soll, schreiben wir den Aufruf in eine eigene Prozedur.

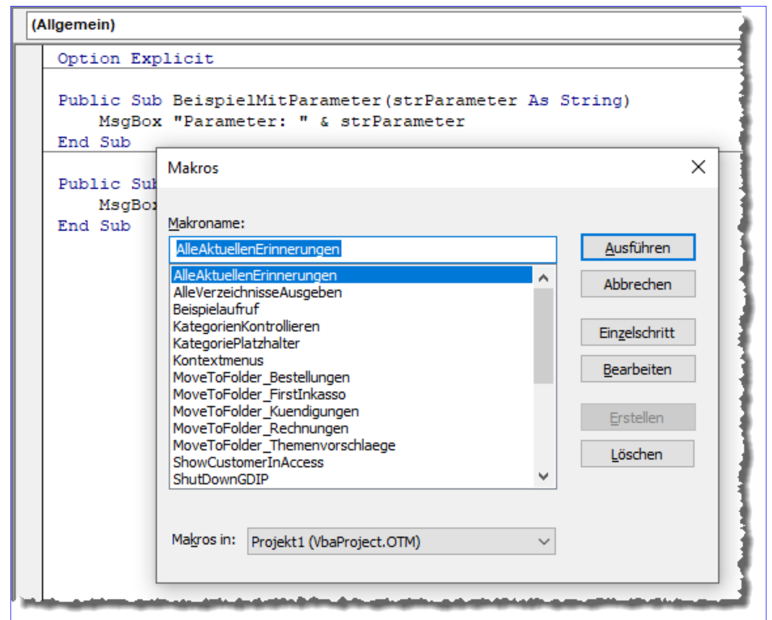

Bild 2: Fehlgeschlagener Aufruf einer Prozedur mit Parameter

Für den Aufruf über den Direktbereich schreiben wir den folgenden Befehl unter Angabe des Parameters einfach in den Direktbereich und betätigen die Eingabetaste:

```
BeispielMitParameter "Beispielparameter"
```

Das sieht beispielsweise wie in Bild 3 aus.

Wir können den Aufruf allerdings auch direkt im gleichen Modul wie die Prozedur in einer weiteren Prozedur unterbringen, die beispielsweise wie folgt aussieht:

```
Public Sub BeispielAufrufen()
    BeispielMitParameter "Bei-
spielparameter"
End Sub
```

Aufruf von Methoden in Klassenmodulen

Das gelingt allerdings beides nicht, wenn sich die Prozedur in einem Klassenmodul befindet. Wie bereits weiter oben erwähnt, müssen wir das Klassenmodul zuvor initialisieren und referenzieren, bevor wir seine Methoden (so werden Prozeduren in Klassenmodulen auch genannt) verwenden können. Wir schauen uns auch das in einem Beispiel an, da es in vielen Fällen notwendig sein wird, mit Klassenmodulen zu arbeiten.

Der Grund ist meistens, dass wir eine Objektvariable mit dem Schlüsselwort **WithEvents** deklarieren müssen, was nur in Klassenmodulen möglich ist. Warum benötigen wir das Schlüsselwort **WithEvents**? Weil wir damit sicherstellen, dass wir die Ereignisprozeduren für das jeweilige Objekt in der aktuellen Klasse imple-

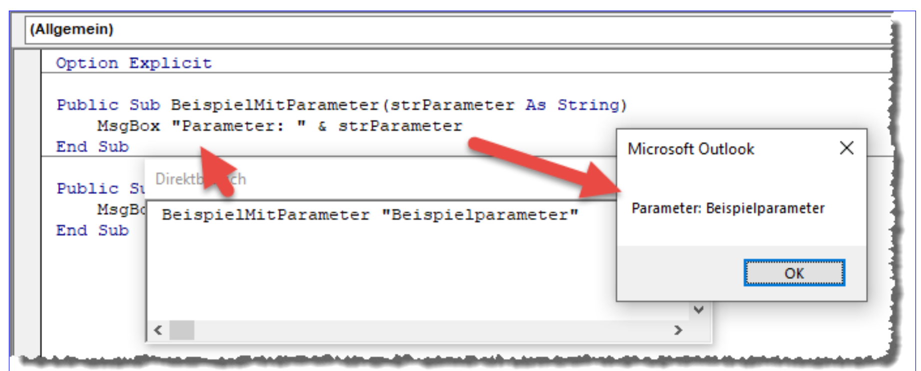

Bild 3: Aufruf einer Prozedur mit Parameter aus dem Direktbereich

Outlook: Explorer automatisieren

Outlook ist wohl die Schaltzentrale an allen Arbeitsplätzen, die mit Microsoft Office arbeiten. Es bietet schon von Haus aus sehr viele Möglichkeiten, um E-Mails, Termine, Aufgaben oder Kontakte zu verwalten. Wenn man länger damit arbeitet, kommen einem früher oder später Ideen, wie man es noch weiter automatisieren könnte. Beispielsweise, um die Inhalte von E-Mails, die man in einen bestimmten Ordner verschiebt, gleich in Aufgaben umzuwandeln, vorgefertigte Antworten für E-Mails bereitzuhalten und so weiter. Der Weg zur Umsetzung führt über die Programmierung per VBA und genauer von VBA-Ereignissen. Denn für fast jede Aktion, die wir in Outlook durchführen, ob es nun das Erstellen oder Abschicken einer E-Mail, das Anlegen einer Aufgabe, das Öffnen eines Kontaktes ist, können wir ein Ereignis definieren, für das wir eine passende Prozedur hinterlegen. Dieser Artikel zeigt die wichtigsten Elemente des Hauptelements der Benutzeroberfläche von Outlook, nämlich dem Explorer, und wie wir diese direkt im VBA-Projekt von Outlook für unsere Zwecke anpassen können.

Die Benutzeroberfläche von Outlook bietet eine Menge Möglichkeiten, mit denen der Benutzer diese bedienen kann. Und um alle denkbaren Ideen für die Automatisierungen von Outlook umzusetzen, wollen wir in der Lage sein, all diese Benutzeraktionen abzufangen, indem wir dafür entsprechende Ereignisprozeduren hinterlegen. Die Basis dafür sind die folgenden Schritte:

- Identifizieren, welcher VBA-Klasse der Outlook-Bibliothek ein Element der Benutzeroberfläche von Outlook entspricht.
- Diese Klasse in Form einer Objektvariablen deklarieren und referenzieren.
- Für die Objektvariable die gewünschten Ereignisprozeduren implementieren.

Die Basis: Die Application-Klasse

Die Grundlage für alle Zugriffe auf die Elemente der Benutzeroberfläche von Outlook ist die **Application**-Klasse. Diese beschreiben wir mit allen Eigenschaften, Methoden und Ereignissen ausführlich im Artikel **Outlook: Die Application-Klasse** (**www.vbentwickler.de/305**).

Fenstertypen unter Outlook

Bevor wir beginnen, ein Fenster unter Outlook zu referenzieren, stellen wir zunächst die verschiedenen Fenster vor. Es gibt die folgenden beiden Fenstertypen:

- **Explorer**: Eigentliches Outlook-Fenster. Es kann auch mehrere davon geben, je nachdem, wie oft der Benutzer Outlook geöffnet hat. Auch mehrere Explorer-Fenster unterliegen jedoch immer einem einzigen **Application**-Objekt. Der Aufruf der **Quit**-Methode des übergeordneten **Application**-Objekts schließt daher immer alle **Explorer**-Fenster.
- **Inspector**: Ein **Inspector**-Fenster ist eines der Fenster, mit dem Sie per Doppelklick auf eines der Elemente in den Ordnern eines **Explorer**-Fensters die Details des jeweiligen Elements anzeigen – beispielsweise für E-Mails, Termine, Kontakte et cetera.

Auf die Fenster von Outlook zugreifen

Bereits die verschiedenen Fenster von Outlook bieten einige Möglichkeiten zur Programmierung und halten dementsprechend auch einige Ereignisse bereit. Bevor wir diese vorstellen, müssen wir jedoch zunächst einmal Zugriff auf das gewünschte Fenster erhalten.

Dafür gibt es zunächst die folgenden drei Methoden:

- **ActiveExplorer**: **ActiveExplorer** liefert einen Verweis auf das oberste Element der geöffneten Outlook-**Explorer** zurück, also das Element, das zuletzt aktiviert wurde. Das gilt auch, wenn zwischenzeitlich Fenster einer anderen Anwendung aktiviert wurden.
- **ActiveInspector**: Liefert einen Verweis auf das **Inspector**-Element für das oberste **Inspector**-Objekt, also das Element, das zuletzt aktiviert wurde. Es kann auch sein, dass gar kein **Inspector**-Fenster geöffnet ist, dann liefert **ActiveInspector** den Wert **Nothing**.
- **ActiveWindow**: **ActiveWindow** liefert immer einen Verweis auf das oberste Element der geöffneten **Explorer**- und **Inspector**-Elemente. Der Typ des zurückgelieferten Elements lautet dementsprechend entweder **Explorer** oder **Inspector**.

Die folgenden Anweisungen geben jeweils den Titel des ermittelten **Explorer**- beziehungsweise **Inspector**-Elements aus:

```
? Application.ActiveExplorer.Caption
? Application.ActiveInspector.Caption
? Application.ActiveWindow.Caption
```

Wir können auch alle **Explorer**- oder alle **Inspector**-Elemente durchlaufen. Hier zunächst die Ausgabe der Titel aller aktuell geöffneten **Explorer**-Elemente:

```
Private Sub AlleExplorer()
    Dim objExplorer As Explorer
    For Each objExplorer In Application.Explorers
        Debug.Print objExplorer.Caption
    Next objExplorer
End Sub
```

Das gelingt auch für alle geöffneten **Inspector**-Elemente:

```
Private Sub AlleInspectors()
    Dim objInspector As Inspector
    For Each objInspector In Application.Inspectors
        Debug.Print objInspector.Caption
    Next objInspector
End Sub
```

Ereignis beim Öffnen eines neuen Explorers

Mit dem Implementieren von Ereignissen können wir gleich beim Öffnen eines neuen Explorers beginnen. Wenn der Benutzer beispielsweise neben dem ersten **Explorer**-Fenster ein weiteres öffnet, weil er eines mit E-Mails und eines mit Terminen nebeneinander anzeigen möchte, fangen wir das wie folgt ab. Dazu deklarieren wir im allgemeinen Teil der Klasse **ThisOutlookSession** die folgende Variable:

```
Dim WithEvents objExplorers As Explorers
```

Diese Variable müssen wir füllen, am einfachsten in der beim Start von Outlook automatisch ausgeführten Ereignisprozedur **Application_Startup**:

```
Private Sub Application_Startup()
    Set objExplorers = Application.Explorers
End Sub
```

Und für das Ereignis **NewExplorer** der Auflistung **objExplorers** hinterlegen wir dann die folgende Ereignisprozedur:

```
Private Sub objExplorers_NewExplorer(ByVal Explorer _
        As Explorer)
    MsgBox "Neuer Explorer: " & Explorer.Caption
End Sub
```

Explorer-Fenster referenzieren

Auch das **Explorer**-Fenster bietet einige Ereignisse an, die wir implementieren können. Wollen wir diese im-

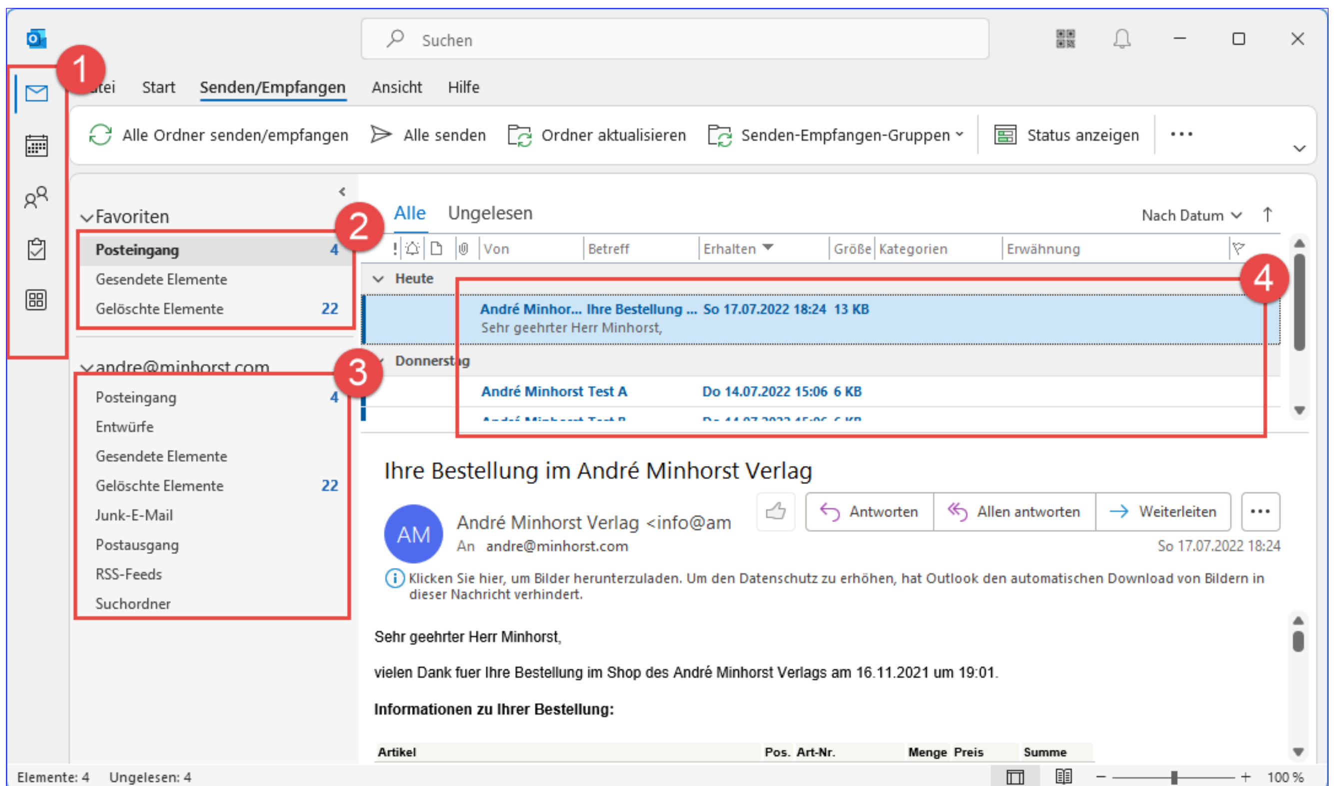

Bild 1: Die Benutzeroberfläche von Outlook mit den typischen Elementen

plementieren, benötigen wir auch für dieses Fenster eine passende Objektvariable, die wir mit dem Schlüsselwort **WithEvents** deklarieren:

```
Dim WithEvents objExplorer As Explorer
```

Eigentlich können wir bereits das vorherige Beispiel nutzen, um diese Variable zu füllen:

```
Private Sub objExplorers_NewExplorer(ByVal Explorer _
        As Explorer)
    Set objExplorer = Explorer
End Sub
```

Wenn wir hingegen direkt beim Starten von Outlook das aktuelle Explorer-Fenster mit dieser Variablen referenzieren wollen, gelingt dies wie folgt:

```
Private Sub Application_Startup()
    Set objExplorer = Explorer
End Sub
```

Auf das Wechseln des Bereichs im Explorer reagieren

In Bild 1 sehen wir einen typischen Outlook-Explorer. Anhand dieses Beispiels wollen wir uns einige Ereignisse ansehen, die der Benutzer durch verschiedene Aktionen auslösen kann. Eines vorneweg: Das Betätigen der Elemente im modernisierten Ribbon werden wir in diesem Artikel nicht dokumentieren.

Ob und wie wir auf das Betätigen der Ribbonbefehle durch den Benutzer reagieren können, zeigen wir in einem eigenen Artikel auf. Hier ermitteln und referenzieren wir zunächst einmal den aktuellen Ordner und reagieren dann darauf, wenn der Benutzer diesen Ordner wechselt. Unter Ordnern verstehen wir die in der Abbildung mit der Zahl **2** und **3** markierten Elemente.

Wenn der Benutzer beispielsweise vom Ordner **Posteingang** zu **Gesendete Elemente** wechselt, löst er damit das Ereignis **FolderSwitch** aus. Dieses implementieren wir beispielsweise wie folgt:

```
Private Sub objExplorer_FolderSwitch()
    Debug.Print "Der Ordner wurde gewechselt in: ", _
        objExplorer.CurrentFolder
End Sub
```

Dies gibt nach dem Wechseln zum Ordner **Gesendete Elemente** den Text **Gesendete Elemente** aus. Das gleiche Ereignis wird ausgelöst, wenn wir zu einem der anderen Bereiche wechseln, die in der Abbildung mit der Zahl **1** markiert sind.

Durch das Wechseln beispielsweise vom Bereich **E-Mails** zu **Kalender** zeigt Outlook nämlich automatisch den zuletzt für diesen Bereich markierten Ordner an.

Ereignisse beim Wechseln der Ansicht

Outlook bietet verschiedene Ansichten an, wenn wir im Ribbon zur Registerseite **Ansicht** wechseln und dann auf **Ansicht ändern** klicken. Wir erhalten dann die Auswahl der Ansichten aus Bild 2. Wählen wir hier einen anderen als den aktuellen Eintrag aus, lösen wir damit zwei Ereignisse aus: **BeforeViewSwitch** und **ViewSwitch**.

Die Prozedur für das Ereignis **BeforeViewSwitch** liefert zwei Parameter – die Bezeichnung der gewählten neuen Ansicht mit **NewView** und den Rückgabeparameter **Cancel**. In dieser Prozedur können wir prüfen, ob die Bedingungen für den Wechsel der Ansicht erfüllt sind und falls nicht, können wir den Wechsel durch Einstellen von **Cancel** auf den Wert **True** stoppen.

Im folgenden Beispiel geben wir jedoch nur den Namen der gewählten Ansicht aus:

```
Private Sub objExplorer_BeforeViewSwitch( _
        ByVal NewView As Variant, Cancel As Boolean)
    Debug.Print "Explorer_BeforeViewSwitch", NewView
End Sub
```

Das zweite Ereignis **ViewSwitch** wird nur ausgelöst, wenn der Wechsel der Ansicht tatsächlich erfolgt ist, also wenn der Parameter **Cancel** in der Ereignisprozedur für das Ereignis **BeforeViewSwitch** nicht auf **True** eingestellt wurde. Hier geben wir nochmals die aktuelle Ansicht aus, die wir diesmal nicht per Parameter erhalten, sondern über die Eigenschaft **CurrentView** ermitteln:

```
Private Sub objExplorer_ViewSwitch()
    Debug.Print "Die Ansicht wurde geändert in: ", _
        objExplorer.CurrentView
End Sub
```

Alle Explorer-Fenster referenzieren

Etwas komplizierter wird es, wenn wir Ereignisse für mehr als ein Explorer-Fenster nutzen wollen. In diesem Fall müssen wir für jedes Explorer-Fenster, das geöffnet ist, eine eigene Objektvariable reservieren und diese mit einem Verweis auf das jeweilige Explorer-Fenster versehen. Nun ist die Menge der zu

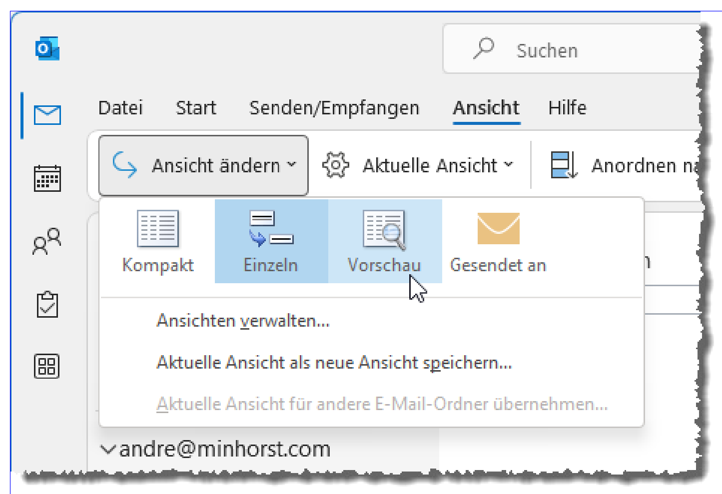

Bild 2: Ändern der Ansicht für die Elemente eines Ordners

erwartenden Explorer-Fenster in den meisten Fällen überschaubar. Ich selbst hatte genau genommen noch nie den Bedarf, mehr als einen Outlook-Explorer zu öffnen – ich bin einfach zwischen den verschiedenen Ordnern hin- und hergesprungen. Auf die Idee, beispielsweise zwei Explorer-Fenster für E-Mails und Termine nebeneinander zu platzieren, um diese gleichzeitig im Überblick zu haben, bin ich zuvor nie gekommen.

Allerdings können wir nicht ausschließen, dass es da draußen Benutzer gibt, die diese sicher durchaus praktische Darstellung nutzen und daher kümmern wir uns auch um diesen Fall. Auch wenn die meisten Benutzer vermutlich eine begrenzte Anzahl von Explorern verwenden, was dazu verleiten könnte, einige Variablen wie **objExplorer1**, **objExplorer2** und so weiter zu verwenden, so müsste man in diesem Fall doch für jedes dieser Objekte einzeln die jeweiligen Ereignisprozeduren anlegen. Und das würde zu viel Arbeit bedeuten, wenn man einmal eine der Prozeduren anpassen wollte – dann müssten diese Änderungen immer gleich mehrfach durchgeführt werden.

Also wählen wir einen alternativen Ansatz, bei dem wir eine Wrapper-Klasse für die **Explorer**-Elemente anlegen. Diese sieht wie in Listing 1 aus, wobei wir hier nur die vier bisher vorgestellten Ereignisse implementiert haben.

Schau Dir diese Konstellation genau an, denn sie wird immer und immer wieder auftreten – ob für einzelne Objekte, deren Ereignisse wir implementieren wollen oder für mehrere, die wir wie im vorliegenden Beispiel in einer Collection speichern.

Wir legen also eine neue Klasse an, die wir in diesem Fall **clsExplorerWrapper** nennen. Dazu betätigen wir als Erstes den Menübefehl **Einfügen|Klassenmodul**. Danach ändern wir den Namen des Klassenmoduls

```
Dim WithEvents m_Explorer As Outlook.Explorer

Public Property Set Explorer(obj As Outlook.Explorer)
    Set m_Explorer = obj
End Property

Private Sub m_Explorer_BeforeFolderSwitch(ByVal NewFolder As Object, Cancel As Boolean)
    Debug.Print "Folder wird zu '" & m_Explorer.CurrentFolder & "' geändert."
End Sub

Private Sub m_Explorer_BeforeViewSwitch(ByVal NewView As Variant, Cancel As Boolean)
    Debug.Print "View wird zu '" & m_Explorer.CurrentView & "' geändert."
End Sub

Private Sub m_Explorer_FolderSwitch()
    Debug.Print "Folder wurde zu '" & m_Explorer.CurrentFolder & "' geändert."
End Sub

Private Sub m_Explorer_ViewSwitch()
    Debug.Print "View wurde zu '" & m_Explorer.CurrentView & "' geändert."
End Sub
```

Listing 1: Code der Wrapperklasse **clsExplorerWrapper**

beispielsweise von **Klasse1** in **clsExplorerWrapper**, indem wir dieses im **Projekt**-Explorer markieren und dann die Eigenschaft **(Name)** im **Eigenschaften**-Bereich einstellen. Übrigens: Im Gegensatz zu den VBA-Projekten beispielsweise von Access-Datenbanken bietet Outlook für neu angelegte Module, deren Standardname wie **Modul1** oder **Klasse1** nicht durch den Entwickler geändert wurde, beim Speichern nicht die Möglichkeit, den Modulnamen zu ändern.

Wir müssen das also manuell erledigen, gerade wenn wir eine Klasse hinzufügen, auf die wir von anderen Modulen aus über den Namen zugreifen wollen (siehe Bild 3).

Die Klasse **clsExplorerWrapper** statten wir dann mit einer Objektvariablen namens **m_Explorer** mit dem Typ **Outlook.Explorer** aus. Da wir für diese Objektvariable auch Ereignisse implementieren wollen, fügen wir außerdem das Schlüsselwort **WithEvents** hinzu:

```
Dim WithEvents m_Explorer As Outlook.Explorer
```

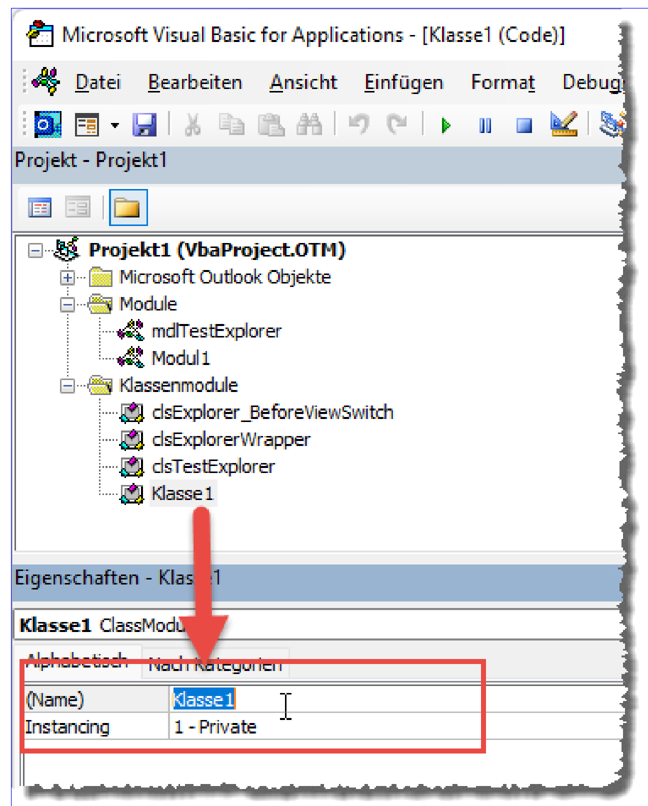

Bild 3: Ändern des Namens eines Moduls

Wenn wir diese Klasse von einem anderen Modul aus erstellen, wollen wir diese Variable mit einem Verweis auf das entsprechende **Explorer**-Element füllen. Also hinterlegen wir im Klassenmodul eine öffentliche **Property Set**-Methode, die als Parameter einen Verweis auf das **Explorer**-Element entgegennimmt und dieses der Variablen **m_Explorer** zuweist:

```
Public Property Set Explorer(obj As Outlook.Explorer)
    Set m_Explorer = obj
End Property
```

Die übrigen Zeilen der Klasse enthalten die Implementierung der vier Ereignisse **BeforeFolderSwitch**, **BeforeViewSwitch**, **FolderSwitch** und **ViewSwitch**.

Nun wollen wir dafür sorgen, dass beim Start von Outlook und somit mit dem Erstellen des ersten **Explorer**-Elements ein Verweis auf dieses in einer neuen Klasse des Typs **clsExplorerWrapper** gespeichert wird. Da es sein kann, dass zu diesem ersten Explorer-Element noch weitere hinzukommen, wollen wir auch für diese neue **clsExplorerWrapper**-Objekte erzeugen. Um diese nicht mit Variablennamen wie **objExplorer1**, **objExplorer2** und so weiter benennen zu müssen, war das Erstellen der Wrapper-Klasse **clsExplorerWrapper** nur der erste Schritt. Der zweite ist, die neu erzeugten Objekte dieser Klasse in einer Collection aufzubewahren. Diese deklarieren wir wie folgt im Modul **ThisOutlookSession**:

```
Dim colExplorers As Collection
```

Außerdem müssen wir dafür sorgen, dass das Explorer-Element gleich beim Start von Outlook in einer neuen Instanz der Klasse **clsExplorerWrapper** hinterlegt und diese zur Collection **colExplorer** hinzugefügt wird. Dazu fügen wir für die beim Start von Outlook

Outlook: Die Application-Klasse

Die Basisklasse einer jeden Office-Anwendung heißt »Application«. Diese stellt je nach Anwendung verschiedene Eigenschaften, Methoden und Ereignisse bereit. Das ist auch im Objektmodell von Outlook der Fall. Hier finden wir beispielsweise die Möglichkeit für den Zugriff auf die »Explorer«- und »Inspector«-Elemente von Outlook, auf die kompletten E-Mails, Kontakte, Termine et cetera über die »GetNamespace«-Funktion oder das wichtige Ereignis »Startup«, mit dem wir gleich beim Start von Outlook wichtige, selbst programmierte Aktionen ausführen können. Dieser Artikel stellt die verschiedenen Member der Application-Klasse vor und beschreibt, was wir mit diesen alles erledigen können.

Application-Klasse im Objektkatalog ansehen

Einen ersten Überblick über die Member einer Klasse kann man sich immer im Objektkatalog des VBA-Editors veschaffen. Dazu wechseln wir von Outlook mit der Tastenkombination **Alt + F11** zum VBA-Editor und öffnen dort mit dem Menübefehl **Ansicht|Objektkatalog** oder mit der Tastenkombination **F2** den Objektkatalog. Hier finden wir in der linken Liste mit der Überschrift **Klassen** den Eintrag **Application** vor, den wir anklicken. Dies zeigt in der rechten Liste die Eigenschaften, Methoden und Ereignisse dieser Klasse an (siehe Bild 1).

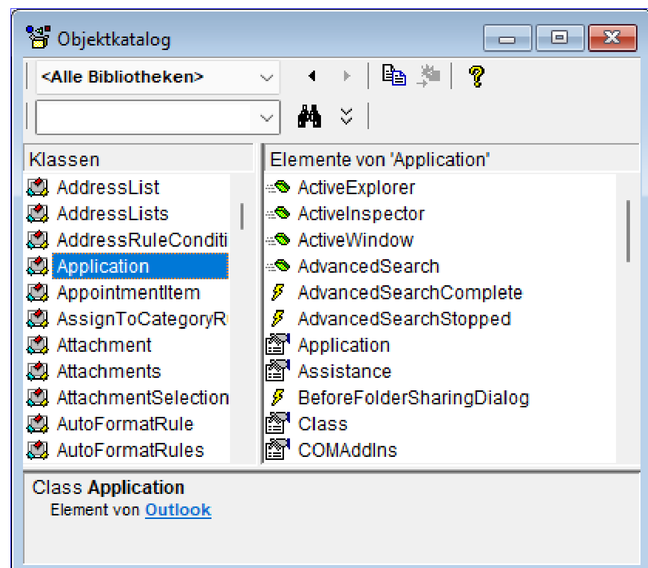

Bild 1: Der Objektkatalog des VBA-Editors von Outlook

Wir beschreiben nachfolgend nur die Elemente, die im Rahmen der in diesem Magazin vorgestellten Lösungen sinnvoll einsetzbar sind.

Eigenschaften der Application-Klasse

Die **Application**-Klasse bietet die folgenden Eigenschaften an. Um diese auszuprobieren, öffnen wir einfach den VBA-Editor und wechseln dann mit **Strg + G** zum Direktbereich, wo wir die Werte der Eigenschaften durch Voranstellen von **Debug.Print** abfragen können:

- **Application**: Liefert einen Verweis auf die **Application**-Klasse selbst.
- **Class**: Liefert einen Wert der Enumeration **olObjectClass** für das jeweilige Objekt zurück, im Falle von **Outlook.Application** den Wert **0** (**olApplication**).
- **COMAddIns**: Auflistung der COM-Add-Ins, die aktuell für diese Outlook-Instanz installiert sind. **Count** liefert die Anzahl der COM-Add-Ins, und mit der **Item**-Eigenschaft können wir auf die einzelnen COM-Add-Ins zugreifen. Der Index ist dabei **1**-basiert.

- **DefaultProfileName**: Diese Eigenschaft liefert den Namen des aktuell verwendeten Outlook-Profils zurück. Dieser wird in einem Dialog festgelegt, den wir über die Systemsteuerung öffnen, wenn wir dort **Mail (Microsoft Outlook)** aufrufen und im nun erscheinenden Dialog **Mail-Setup** auf **Profile anzeigen...** klicken (siehe Bild 2).

- **Explorers**: Auflistung der geöffneten **Explorer**-Fenster. Ein **Explorer**-Fenster ist das Fenster, in dem der Inhalt eines Outlook-Ordners angezeigt wird. **Explorers** liefert mit **Count** die Anzahl und mit **Item (Index)** die jeweiligen Instanzen von **Explorer**.

- **Inspectors**: Auflistung der geöffneten **Inspector**-Fenster. Ein **Inspector**-Fenster öffnen wir beispielsweise, wenn wir doppelt auf eine E-Mail oder einen Termin in einem der **Explorer**-Fenster klicken. **Inspectors** liefert mit **Count** die Anzahl und mit **Item(Index)** einen Verweis auf das jeweilige **Inspector**-Fenster.

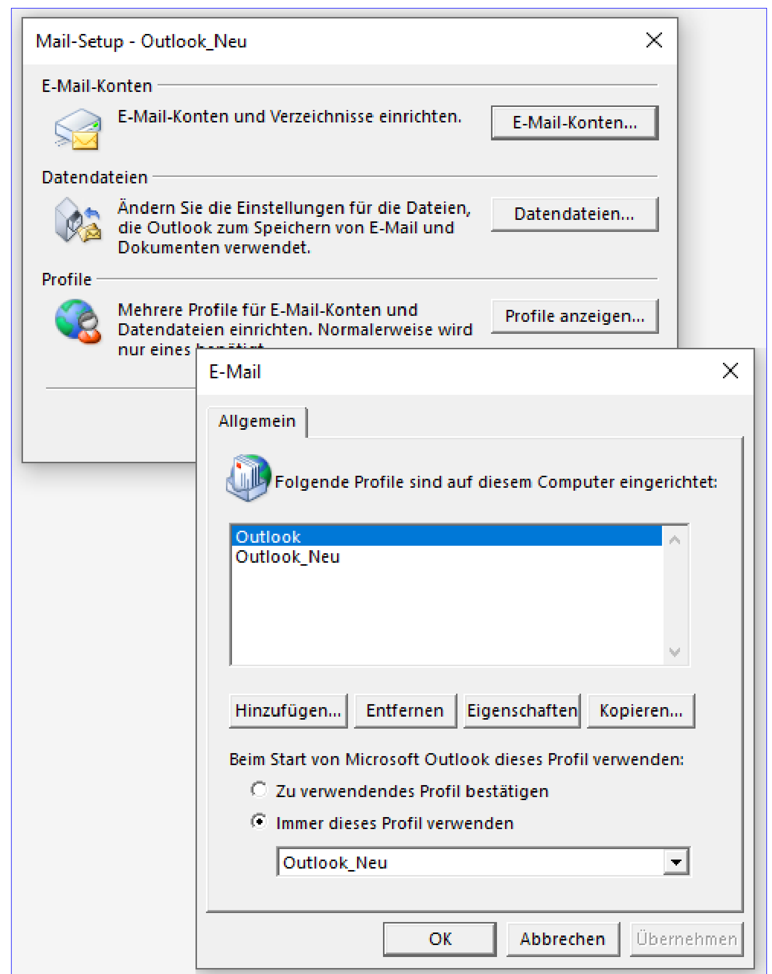

Bild 2: Verwalten von Outlook-Profilen

- **IsTrusted**: Gibt an, ob die abfragende Instanz als vertrauenswürdig eingestuft ist. Das ist nur der Fall, wenn der Aufruf vom Outlook-VBA-Projekt erfolgt oder aus einem dort registrierten COM-Add-In.

- **LanguageSettings**: Dient der Prüfung, ob eine Sprache als Bearbeitungssprache verwendet wird (zum Beispiel durch Ausgabe von **Application.LanguageSettings.LanguagePreferredForEditing(msoLanguageIDGerman)** zur Prüfung der deutschen Sprache) oder welche die aktuelle Sprache der Benutzeroberfläche ist (**Application.LanguageSettings.LanguageID(msoLanguageIDUI)** liefert für die deutsche Sprache beispielsweise den Wert **1031** gleich **msoLanguageIDGerman**)

- **Name**: Liefert im Falle von Outlook immer **Outlook**.

- **Parent**: Liefert einen Verweis auf das übergeordnete Objekt. Ist für **Application.Parent** leer und liefert beim Versuch der Ausgabe einen Fehler. Deshalb vorher beispielsweise mit **Application.Parent Is Nothing** prüfen.

- **ProductCode**: Gibt den Produktcode für Outlook zurück, zum Beispiel **{90160000-000F-0000-0000-0000000FF1CE}**.

Outlook: Elemente durchsuchen mit AdvancedSearch

Die »AdvancedSearch«-Methode der Application-Klasse von Outlook erlaubt das Suchen von Elementen nach bestimmten Kriterien in dem vorgegebenen Ordner – gegebenenfalls sogar mit Unterordnern. An ihr hängen zwei Ereignisse namens »AdvancedSearchComplete« und »AdvancedSearchStopped«, die je nach dem Ausgang der Suche ausgelöst werden. Dieser Artikel zeigt, wie wir eine Suche in Outlook-Elementen per VBA durchführen und wie wir die damit zusammenhängenden Ereignisse verwenden müssen, damit die Suche wie gewünscht funktioniert. Außerdem erfährst Du, wie Du Suchbegriff und Suchbereich definieren musst.

Im Artikel **Outlook: Die Application-Klasse** (**www.vbentwickler.de**) haben wir die AdvancedSearch-**Methode** und die beiden Ereignisse **AdvancedSearchComplete** und **AdvancedSearchStopped** zusammen mit den anderen Elementen der **Application**-Klasse von Outlook vorgestellt. Im vorliegenden Artikel nun gehen wir ins Detail und zeigen, wie Du in Outlook mit diesen Techniken suchen kannst.

Asynchrone Suche

Als Erstes wollen wir uns dabei ansehen, warum die Suche neben der Methode **AdvancedSearch** überhaupt ein weiteres Ereignis namens **AdvancedSearchComplete** benötigt.

Der Grund ist einfach: Die Methode **AdvancedSearch** ist nämlich eine asynchrone Methode, das heißt, sie wird, einmal aufgerufen, unabhängig vom aufrufenden Code ausgeführt. Das heißt, dass der aufrufende Code einfach weiterläuft. Das ist insofern ungünstig, als dass wir nach dem Aufruf von **AdvancedSearch** nicht einfach abwarten können, bis die Suche abgeschlossen ist und dann das Suchergebnis auswerten können.

Stattdessen haben wir zwei Möglichkeiten:

- Wir können beispielsweise eine Schleife definieren, die solange läuft, bis die Suche beendet ist. Wie finden wir das heraus? Wir verwenden in der Schleife ein Abbruchkriterium, das erst dann wahr wird, wenn die Suche fertig ist. Das wiederum erkennen wir, indem wir das Ereignis **AdvancedSearchComplete** implementieren und dort die als Abbruchkriterium verwendete und öffentlich deklarierte Variable einstellen. Danach können wir die Schleife verlassen und das Suchergebnis auswerten.

- Die zweite Variante verschiebt die Auswertung des Suchergebnisses direkt in die Implementierung des Ereignisses AdvancedSearchComplete. Wir können auch hier auf das Suchergebnis zugreifen, das in diesem Fall über den Parameter der Ereignisprozedur geliefert wird.

Wir schauen uns im Anschluss beide Varianten an.

Ausprobieren der Suchfunktion mit Endlosschleife

Die nächste Frage ist, wo wir die Suchfunktion ausführen wollen. In einer späteren Stufe würden wir diese beispielsweise als Teil einer Anwendung wie einer reinen VB-Anwendung oder auch einer Datenbankanwendung verwenden.

Oder wir rufen die Suchfunktion über einen benutzerdefinierten Ribboneintrag auf. Da das für die reine Vorstellung der **AdvancedSearch**-Methode etwas zu aufwendig ist, wollen wir diese zunächst gesondert betrachten, indem wir sie einfach aus einem Modul aus dem VBA-Editor von Outlook heraus aufrufen.

Als Erstes benötigen wir eine **Boolean**-Variable zum Speichern der Information, ob die Suche bereits beendet ist. Diese deklarieren wir wie folgt:

```
Public bolSearchComplete As Boolean
```

Außerdem können wir das Ereignis nicht einfach für die **Application**-Klasse implementieren, sondern wir müssen dafür eine eigene Objektvariable des Typs **Outlook.Application** deklarieren:

```
Public WithEvents objApplication As Outlook.Application
```

Damit können wir nun eine Suche wie in der Prozedur **SucheNachMailsMitBestimmtemBetreff** programmieren (siehe Listing 1).

Die Prozedur deklariert einige wichtige Variablen:

- **objSearch**: Variable für das **Search**-Objekt, mit dem wir die durch **AdvancedSearch** erstellte Suche referenzieren
- **objResults**: Objektvariable des Typs **Results** zum Referenzieren des Suchergebnisses
- **strScope**: **String**-Variable für die Angabe des zu durchsuchenden Bereichs, hier **Posteingang**.
- **strFilter**: **String**-Variable für den Ausdruck mit dem Filterkriterium

Die Prozedur referenziert zunächst das **Application**-Objekt der aktuellen Outlook-Instanz mit **objApplication**. Dann stellt es **bolSearchComplete** zunächst auf **False** ein, falls dieses durch eine vorherige Suche noch den Wert **True** enthielt.

Dann schreibt es das Filterkriterium, dazu später mehr, in die Variable **strFilter** und den zu durchsuchenden Bereich in **strScope**. Schließlich startet die Prozedur mit der Methode **AdvancedSearch** den Suchvorgang.

Neben **strScope** und **strFilter** wird für den dritten Parameter **SearchSubFolders** noch der Wert **False**

```
Sub SucheNachMailsMitBestimmtemBetreff()
    Dim objSearch As Outlook.Search
    Dim objResults As Outlook.Results
    Dim strScope As String
    Dim strFilter As String
    Dim i As Integer
    Set objApplication = Outlook.Application
    bolSearchComplete = False
    strFilter = "urn:schemas:httpmail:subject = 'Test A'"
    strScope = "Posteingang"
    Set objSearch = objApplication.AdvancedSearch(strScope, strFilter, False, "Test")
    Do While Not bolSearchComplete = True
        DoEvents
    Loop
    Set objResults = objSearch.Results
    For i = 1 To objResults.count
        Debug.Print objResults.Item(i).Subject
    Next i
End Sub
```

Listing 1: Beispiel für den Aufruf von **AdvancedSearch**

übergeben, weil wir keine Unterordner durchsuchen wollen. Und für den vierten Parameter übergeben wir einen Tag, anhand dessen wir später in der Ereignisprozedur **AdvancedSearchComplete** herausfinden können, ob das Ereignis auch durch diese Suche ausgelöst wurde – in diesem Fall schlicht **Test**.

Nun geschieht Folgendes: Die Suche wird gestartet und die Prozedur läuft direkt weiter, da der Aufruf der Suchfunktion wie oben beschrieben asynchron erfolgt.

Damit die aufrufende Prozedur dennoch nach Abschluss der Suche das Suchergebnis untersuchen kann, startet dort nun eine **Do While**-Schleife, die solange wiederholt wird, bis die Variable **bolSearchComplete** den Wert **True** erhält. Und das ist genau dann der Fall, wenn das Ereignis **AdvancedSearchComplete** ausgelöst wird und wir die folgende Ereignisprozedur dafür implementiert haben:

```
Private Sub objApplication_AdvancedSearchComplete(_
        ByVal SearchObject As Search)
    Debug.Print "The AdvancedSearchComplete Event fired"
    If SearchObject.Tag = "Test" Then
        bolSearchComplete = True
    End If
End Sub
```

Damit wir sehen, wann das Ereignis ausgelöst wird, geben wir per **Debug.Print** eine Meldung im Direktbereich aus. Danach prüfen wir, ob der Aufruf von **AdvancedSearch** mit dem Wert **Test** als vierten Parameter das Ereignis ausgelöst hat. Ist das der Fall, stellt die Prozedur **bolSearchComplete** auf **True** ein.

Danach kann die aufrufende Prozedur die **Do While**-Schleife verlassen und das Suchergebnis aus der Eigenschaft **Results** von **objSearch** mit der Variablen **objResults** referenzieren. Dieses durchlaufen wir in einer **For...Next**-Schleife über die Werte **1** bis zu der mit **objResults.Count** ermittelten Anzahl der Suchergebnisse.

Für das jeweils mit **objResults.Item(i)** ermittelte Element der Liste mit den Suchergebnissen geben wir nun den Inhalt der Eigenschaft **Subject** im Direktbereich aus.

Suchfunktion mit Auswertung in AdvancedSearchComplete

Die zweite Variante soll, wie weiter oben vorgegeben, das Suchergebnis direkt in der durch das Ereignis AdvancedSearchComplete ausgelösten Prozedur analysieren. Die auslösende Prozedur sieht viel schlanker aus und endet bereits mit dem Aufruf von **AdvancedSearch**:

```
Sub SucheImPosteingangErgebnisImEreignis()
    Dim objSearch As Outlook.Search
    Dim strScope As String
    Dim strFilter As String
    Set objApplication = Outlook.Application
    strFilter = "urn:schemas:httpmail:subject = 'Test A'"
    strScope = "Posteingang"
    Set objSearch = objApplication.AdvancedSearch(_
        strScope, strFilter, False, "Test")
End Sub
```

Auch die Variable **bolSearchComplete** benötigen wir nicht mehr. Die Ereignisprozedur **objApplication_AdvancedSearchComplete** erhält dafür einige neue Zeilen. Sie deklariert nun die Variable des Typs **Outlook.Results** und füllt diese mit der Eigenschaft **Results** des mit dem Parameter **SearchObject** gelieferten Suchobjekts.

Dieser entspricht dem Rückgabewert der Methode **AdvancedSearch** aus dem vorherigen Beispiel. Danach prüft die Ereignisprozedur wieder anhand des Tags, ob es sich um das Ereignis der zuvor aufgerufenen Suche handelt. Ist das der Fall, durchläuft die Prozedur die Elemente des Suchergebnisses in einer **For...Next**-Schleife und gibt den Wert der Eigenschaft **Subject** der gefundenen Elemente aus:

```
Private Sub objApplication_AdvancedSearchComplete(_
        ByVal SearchObject As Search)
    Dim objResults As Outlook.Results
    Dim i As Integer
    If SearchObject.Tag = "Test" Then
        Set objResults = SearchObject.Results
        For i = 1 To objResults.count
            Debug.Print objResults.Item(i).Subject
        Next i
    End If
End Sub
```

Anpassen des zu durchsuchenden Bereichs

Um den zu durchsuchenden Bereich zu definieren, können wir die Werte des ersten und dritten Parameters anpassen:

- Für den ersten Parameter **Scope** können wir statt Posteingang auch einen anderen Ordner angeben.
- Für den dritten Parameter **SearchSubFolders** können Sie mit **True** einstellen, dass nicht nur der mit Scope angegebene Ordner, sondern auch seine Unterordner durchsucht werden sollen.

Gegebenenfalls heißt der Ordner für den Posteingang auf Deinem System nicht **Posteingang**, sondern beispielsweise **Inbox**, weil Du eine andere Anwendungssprache verwendet.

In diesem Fall kannst Du den korrekten Namen für die Standardordner beispielsweise wie folgt im Direktbereich ermitteln:

```
? Application.GetNamespace("MAPI").GetDefaultFolder(_
    olFolderInbox)
Posteingang
```

Mehr als einen Ordner durchsuchen

Wenn Du mehr als einen Ordner durchsuchen möchtest, gibst Du diese einfach durch Kommata getrennt an. Dabei fassen wir die einzelnen Ordnernamen sicherheitshalber in Hochkommata ein wie in folgendem Beispiel, mit dem wir den Posteingang und die gelöschten Elemente durchsuchen:

```
strScope = "'Posteingang', 'Gelöschte Elemente'"
```

Anpassen des Suchkriteriums

Beim Suchkriterium können wir leider nicht einfach den Namen der Vergleichseigenschaft angeben, sondern verwenden einen Ausdruck wie den folgenden, der den kompletten Namespace enthält:

```
urn:schemas:httpmail:subject = 'Test A'
```

Dieser liefert alle E-Mails, deren Betreff genau **Test A** lautet. Nun benötigen wir sicher einmal andere Betreffs als Vergleichswerte und wir möchten auch einmal mit Platzhaltern arbeiten – und natürlich möchten wir auch noch andere Felder zu Vergleichszwecken nutzen.

Um all dies kümmern wir uns in den folgenden Abschnitten.

Das Schlüsselwort, nach dem Du im Web suchen musst, wenn Du Suchkriterien formulieren möchtest, lautet übrigens **DASL** (**DAV Searching and Locating**).

Namespaces für die einzelnen Objekttypen

Genau wie das oben verwendete **urn:schemas:httpmail** für die Suche in **MailItem**-Elementen finden wir auch für die übrigen Elementtypen entsprechende Namespaces. Diese lauten:

- **MailItem**-Element: **urn:schemas:httpmail**
- **AppointmentItem**-Element: **urn:schemas:calendar**
- **ContactItem**-Element: **urn:schemas:contacts**

E-Mails per VBA erstellen mit CreateItem

Wie man mit Outlook eine E-Mail erstellt, diese mit Betreff, Inhalt, Anlagen, Empfänger und so weiter füllt und diese dann verschickt, weiß mittlerweile jeder. Aber was, wenn man das Erstellen von E-Mails automatisieren möchte und dazu VBA nutzen will? Beispielsweise, um auf Knopfdruck eine Standardmail an einen Empfänger zu schicken oder auch eine Mail an eine Liste von Kunden zu senden? Der vorliegende Artikel zeigt, wie Sie per VBA neue E-Mails erstellen und diese dann entweder zum Betrachten, Nachbearbeiten und zum manuellen Senden anzeigen oder diese sofort auf den Weg zum Empfänger bringen.

Einfache E-Mails programmieren

Unabhängig von den vielen Anwendungsfällen, die wir uns für das VBA-gesteuerte Erstellen von E-Mails vorstellen können, schauen wir uns in diesem Artikel erst einmal die technische Seite an. Dabei wollen wir eine einfache E-Mail wie die aus Bild 1 erstellen – also den Empfänger festlegen und den Betreff und den Inhalt eintragen. Als Abschluss wollen wir noch ein Attachment hinzufügen.

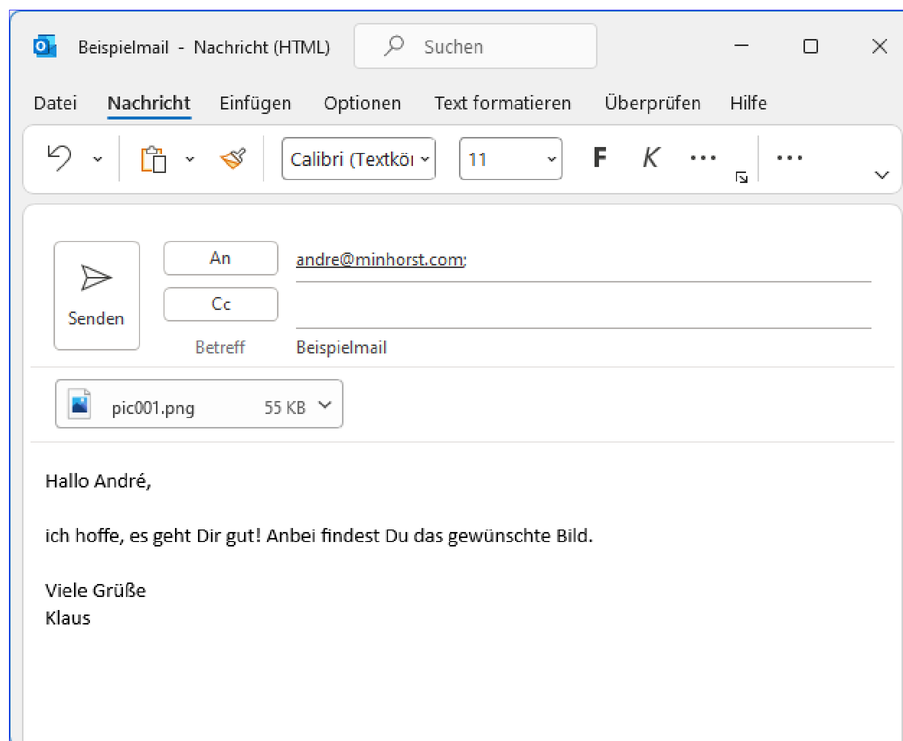

Bild 1: Zu erstellende E-Mail

Beispielcode in neuem Modul

Den Beispielcode zum Erstellen einer solchen E-Mail bringen wir in einem neuen Modul im VBA-Projekt des VBA-Editors unter, wo wir diesen leicht ausführen können. In weiteren Artikeln schauen wir uns an, wie wir solche und andere Prozeduren über das Ribbon starten können – entweder durch manuelle, benutzerdefinierte Anpassungen oder auch durch das Bereitstellen von Code und Ribbon-Anpassungen mit COM-Add-Ins.

Dazu öffnest Du den VBA-Editor (am schnellsten mit der Tastenkombination **Alt + F11**) und wählst den Menübefehl **Einfügen|Modul** aus. Damit Du den gleich erstellten Code später schnell wiederfindest, ändern wir gleich den Namen des neuen Moduls. Dazu markieren wir das Modul im **Projekt**-Explorer (**Strg + R**) und passen dann den Namen im **Eigenschaften**-Bereich an (**F4**) und geben für die Eigenschaft **(Name)** den Wert **mdlEMailsErstellen** ein (siehe Bild 2).

Leere E-Mail erstellen

Danach beginnen wir mit dem Erstellen und Anzeigen einer

neuen, leeren E-Mail. Dazu legen wir eine neue Prozedur namens **LeereEMailErstellen** an.

Hier kannst Du nun direkt mit dem Programmieren beginnen, und zwar mit der folgenden kleinen Prozedur. Diese deklariert eine Objektvariable des Typs **MailItem** und füllt diese durch den Aufruf der Methode **CreateItem**. Diese Methode können wir zum Erstellen aller in Outlook verwendeter Elemente wie E-Mails, Kontakte, Termine, Aufgaben, Notizen et cetera nutzen.

Dazu müssen wir lediglich den Typ des zu erstellenden Elements in Form einer Konstanten für den einzigen Parameter der Methode **CreateItem** übergeben. Im Falle einer E-Mail handelt es sich dabei um die Konstante **olMailItem**. Damit haben wir, auch wenn wir diese noch nicht sehen, bereits eine neue E-Mail erstellt. Um diese sichtbar zu machen, rufen wir noch die Methode **Display** der Objektvariablen mit der neuen E-Mail auf:

```
Public Sub LeereEMailErstellen()
    Dim objMailItem As MailItem
```

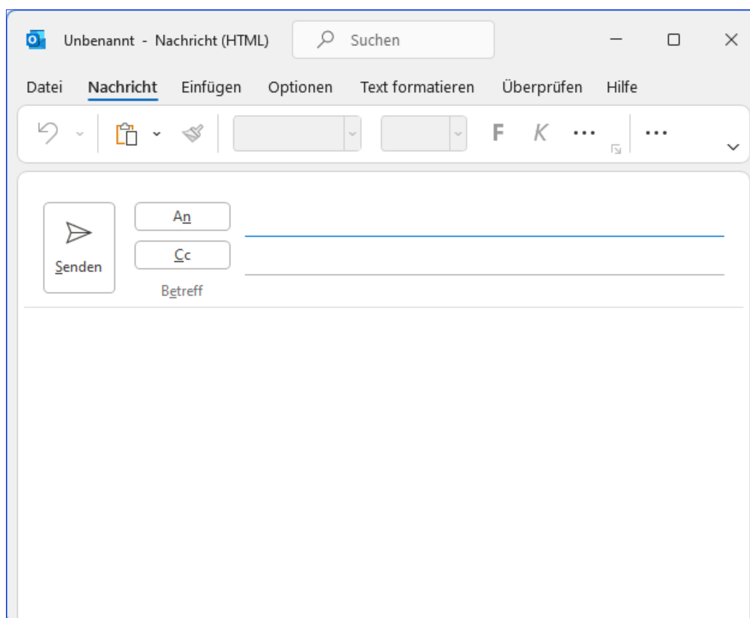

Bild 3: Eine per VBA erstellte E-Mail

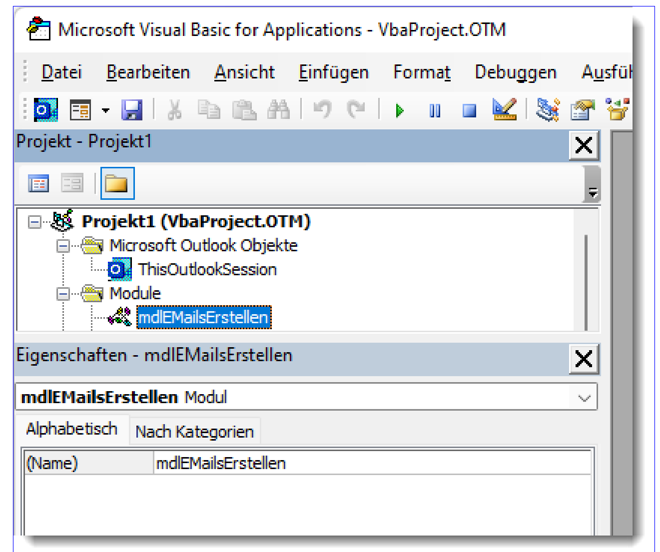

Bild 2: Umbenennen des VBA-Moduls

```
    Set objMailItem = Application.CreateItem(olMailItem)
    objMailItem.Display
End Sub
```

Um diese Prozedur auszuführen, platzieren wir einfach die Einfügemarke innerhalb der Prozedur und betätigen die Taste **F5**. Das funktioniert wie gewünscht: Es erscheint eine neue, leere E-Mail (siehe Bild 3).

Parameter der Display-Methode

Die **Display**-Methode bietet einen Parameter namens **Modal** an, der standardmäßig auf **False** eingestellt ist. Wenn Sie diesen auf True einstellen, wird die E-Mail als modales Fenster geöffnet. Das heißt, dass der Benutzer erst mit Outlook weiterarbeiten kann, wenn er die E-Mail auf irgendeine Art geschlossen hat – entweder durch Versenden oder durch anderweitiges Schließen:

```
objMailItem.Display True
```

Für Experimente mit den in diesem Artikel vorgestellten Beispielen ist die Einstellung des Para-

```
Public Sub EMailErstellen()
    Dim objMailItem As MailItem
    Dim strBody As String
    Set objMailItem = Application.CreateItem(olMailItem)
    With objMailItem
        .To = "andre@minhorst.com"
        .Subject = "Beispielmail"
        strBody = "Hallo André, " & vbCrLf & vbCrLf
        strBody = strBody & "ich hoffe, es geht Dir gut! Anbei findest Du das gewünschte Bild." & vbCrLf & vbCrLf
        strBody = strBody & "Viele Grüße" & vbCrLf
        strBody = strBody & "Klaus"
        .Body = strBody
        .Display
    End With
End Sub
```

Listing 1: Code zum Erstellen einer Mail mit Empfänger, Betreff und Inhalt

meters **Modal** auf den Wert **True** recht praktisch, da das VBA-Fenster immer den Fokus behält und gegebenenfalls geöffnete Inspector-Fenster mit E-Mails verdeckt.

Eigenschaften der E-Mail einstellen

Nun wollen wir die Eigenschaften wie den Empfänger, den Betreff und den Inhalt der E-Mail einstellen. Dazu deklarieren wir in einer neuen Prozedur namens **EMailErstellen** eine weitere Variable namens **strBody** (siehe Listing 1). In dieser stellen wir den Inhalt der E-Mail zusammen, bevor wir ihn zuweisen. Zuvor jedoch weisen wir der Eigenschaft **To** den Empfänger zu und der Eigenschaft **Subject** den Betreff der E-Mail.

Danach weisen wir der Variablen **strBody** die erste Zeile des Inhalts zu und fügen dieser mit der Konstanten **vbCrLf** zwei Zeilenumbrüche hinzu. Die nächste Anweisung greift den Inhalt von **strBody** auf und fügt diesem eine weitere Zeile samt Zeilenumbrüchen hinzu – und so stellen wir den kompletten Inhalt der E-Mail zusammen. Ist der Inhalt in **strBody** zusammengestellt, weisen wir diesen der Eigenschaft **Body** des **MailItem**-Elements zu. Schließlich zeigen wir die vollständige E-Mail wieder mit der **Display**-Methode an.

Da wir in dieser Prozedur oft auf Eigenschaften und Methoden des Elements **objMailItem** zugreifen, haben wir diese mit dem **With**-Schlüsselwort referenziert. Zwischen dieser Zeile und der Zeile **End With** können wir dann über die Punkt-Syntax auf die Eigenschaften und Methoden zugreifen, ohne jedes Mal explizit das Element **objMailItem** angeben zu müssen.

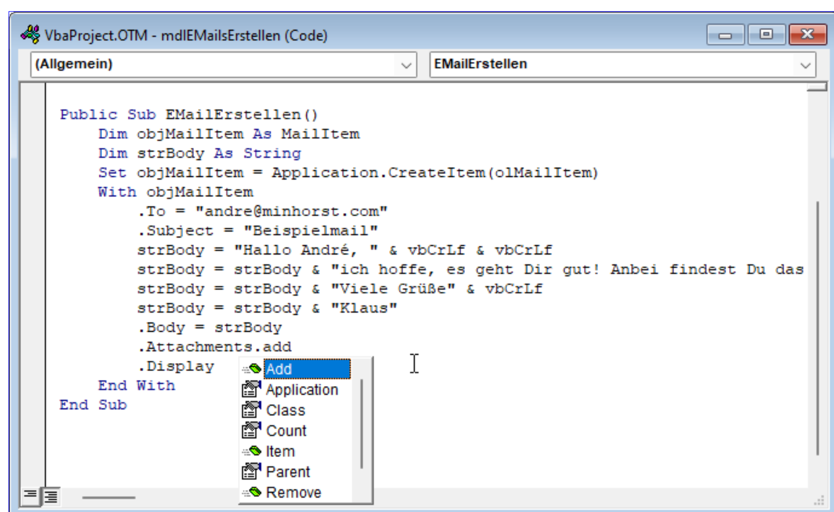

Bild 4: Methoden per IntelliSense auswählen

twinBASIC: Visual Basic für die Zukunft

twinBASIC ist eine moderne Version des klassischen Visual Basic, die auf 100% Kompatibilität zu bestehenden Visual Basic 6-Projekten ausgelegt sein wird. Es ist ein Projekt des englischen Programmierers Wayne Phillips, der bereits im Access-Umfeld mit spektakulären Lösungen auf sich aufmerksam gemacht hat. twinBASIC scheint jedoch sein Meisterwerk zu werden. Es kommt mit einer eigenen Entwicklungsumgebung, die sich nicht vor anderen modernen IDEs verstecken muss. In diesem Artikel schauen wir uns an, wie Du die twinBASIC-Entwicklungsumgebung installierst und die ersten Projekte damit erstellst. In weiteren Artikeln werfen wir einen Blick auf die verschiedenen Projekttypen und wie Du Lösungen damit erstellen kannst.

Möglichkeiten von twinBASIC

twinBASIC schickt sich an, ein vollständiger Ersatz für Visual Basic 6 zu werden – und dieses auf eine moderne Ebene zu heben. Einer der wichtigsten Faktoren, warum Visual Basic 6 nicht mehr genutzt wurde, ist die Einschränkung auf 32-Bit. twinBASIC

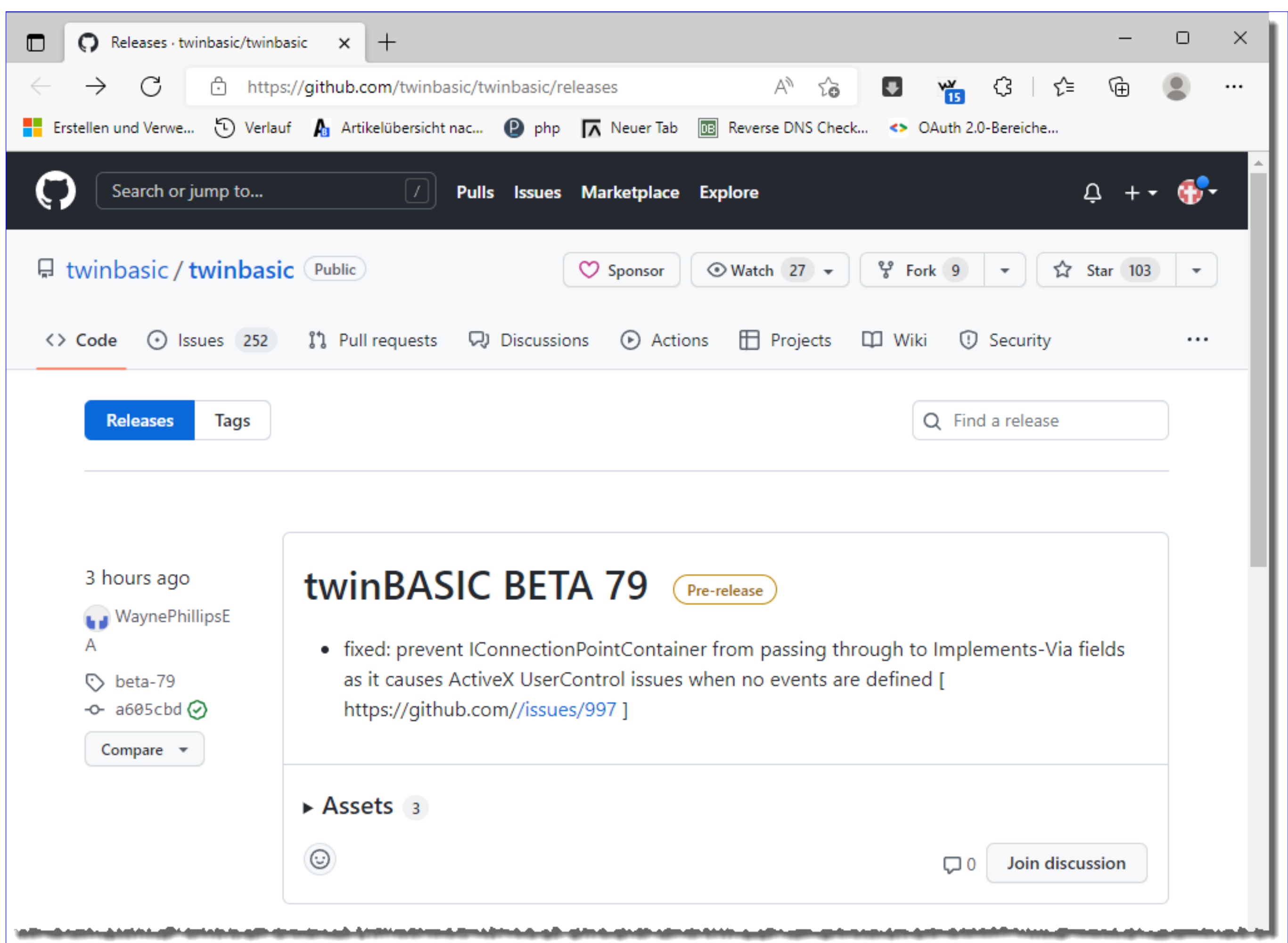

Bild 1: Download von twinBASIC in der jeweils aktuellen Version

bietet auch die Möglichkeit, beispielsweise DLLs, COM-Add-Ins und mehr in einer 64-Bit-Version zu erstellen.

Mittlerweile kann twinBASIC nicht nur reine Konsolenprojekte beziehungsweise DLLs und COM-Add-Ins, sondern auch solche mit Benutzeroberfläche erstellen.

Geplant ist auch eine vollständige Kompatibilität mit Visual Basic 6, das heißt, dass Du Deine alten Visual Basic 6-Projekte hervorholen und modernisieren kannst.

Welche Editionen bietet twinBASIC?

Zum Zeitpunkt der Erstellung dieses Artikels gibt es eine Community Edition, eine Professional Edition und eine Ultimate Edition. Die Community Edition bietet alle wichtigen Funktionen und erlaubt die kommerzielle Nutzung der damit erstellten Projekte.

Einschränkungen sind eine nicht optimierte Kompilierung sowie die Anzeige von Splash-Screens bei Projekten auf Basis von 64-Bit. Diese Edition ist komplett kostenlos – es lohnt sich also, diese auszuprobieren.

Die Professional Edition erlaubt das Kompilieren für 64-Bit ohne Anzeige von Splash-Screens in den damit erstellten Projekten. Außerdem erhält man Zugriff auf Betaversionen. Die Jahreslizenz kostet regulär 420 EUR, ist aber aktuell zum Vorbesteller-Rabatt für 248,80 EUR erhältlich (alle Preise zuzüglich MwSt.).

Die Ultimate Edition liefert zusätzlich – und das ist wirklich revolutionär – Cross-Plattform-Kompatibilität, man kann also Projekte für die Nutzung unter Linux, Mac OS oder Android erstellen. Dieses Feature ist allerdings noch nicht verfügbar. Den reduzierten Jahrespreis kann man sich dennoch jetzt schon sichern. Die Jahreslizenz kostet regulär 756 EUR und kann aktuell zum Vorbesteller-Tarif für 486 EUR bestellt werden (alle Preis zuzüglich MwSt.).

Die Möglichkeit zur Bestellung findest Du in unserem Shop unter **https://shop.minhorst.com**. Dort wechselst Du einfach zur Kategorie Access-Tools und findest die Möglichkeit zur Bestellung einer Jahreslizenz. Und noch ein Hinweis: Aktuell gibt es keine Angabe seitens des Herstellers, wie lange die Vorbestellung noch möglich und damit der reduzierte Preis noch gültig ist. Der Hersteller sichert den Vorbestellerpreis jedoch auch für die Verlängerung der Lizenz über das erste Jahr hinaus zu!

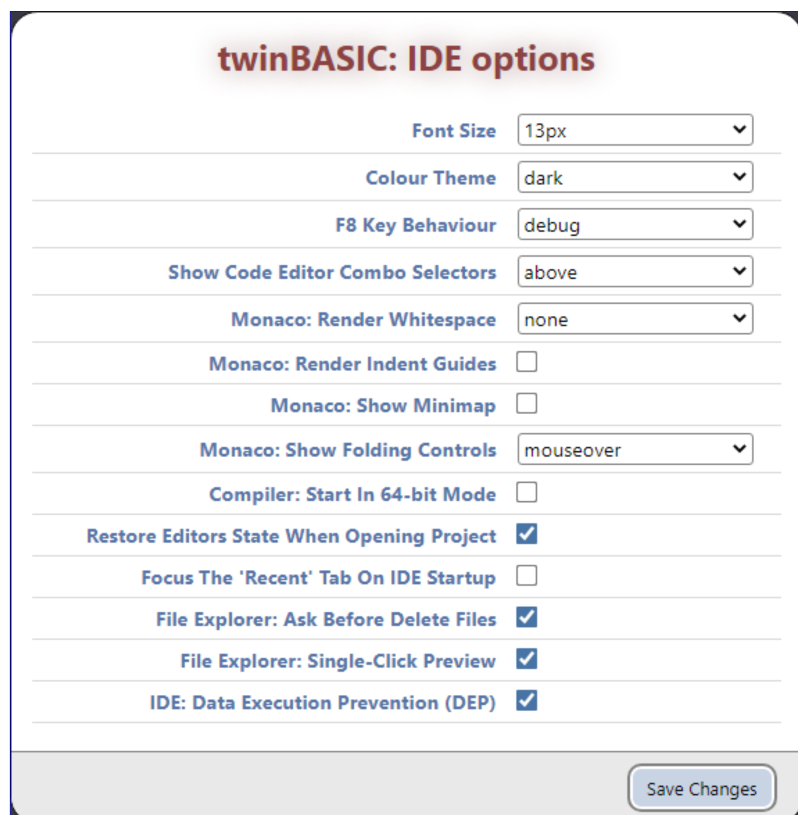

Bild 2: Einstellen der Optionen beim Starten

twinBASIC herunterladen

Den Download der Entwicklungsumgebung von twinBASIC findest Du aktuell in github.com. Da die Entwicklungsumgebung noch im Betastatus ist, werden regelmäßig neue Versionen veröffentlicht.

Diese gibt Wayne Phillips unter folgendem Link bekannt:

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

Hier wählen wir einfach den obersten Eintrag aus – zum Zeitpunkt der Erstellung dieses Artikels beispielsweise Version Beta 79 (siehe Bild 1).

twinBASIC starten

Nach dem Download der knapp 10 MB großen **.zip**-Datei extrahiert man den kompletten Inhalt in das gewünschte Verzeichnis und startet **twinBASIC** einfach durch einen Doppelklick auf die enthaltene Datei **twinBASIC.exe**. Das heißt, Du brauchst noch nicht einmal eine Installation durchzuführen!

Optionen einstellen

Beim ersten Start erscheint der Dialog **twinBASIC: IDE options**, mit dem Du die gewünschten Einstellungen vornehmen kannst. Wichtig ist hier zum Beispiel, ob Du Anwendungen im 32-Bit- oder 64-Bit-Modus erstellen möchtest. Die übrigen Optionen sind weitgehend selbsterklärend (siehe Bild 2).

Projekte neu anlegen oder öffnen

Nach dem Einstellen und dem Speichern der Optionen folgt der eigentliche Startbildschirm (siehe Bild 3).

Bild 3: Vorlagen für Projekte

Bild 4: Beispielprojekte

COM-Add-Ins mit twinBASIC

COM-Add-Ins sind Erweiterungen für Office-Anwendungen und ihre Entwicklungsumgebung, den VBA-Editor. Damit lassen sich Erweiterungen programmieren, die über die Benutzeroberfläche der jeweiligen Anwendung verfügbar gemacht werden und ihre Aufgabe mit oder ohne ein eigenes User Interface bereitstellen. COM-Add-Ins kann man jedoch nicht mit den Mitteln der Office-Anwendungen selbst programmieren. Dazu sind weitere Tools notwendig. Früher ging dies am einfachsten mit Visual Studio 6. Dieses ist jedoch spätestens seit der Einführung der 64-Bit-Versionen der Office-Anwendungen nicht mehr nutzbar, sodass Alternativen gefragt sind. Neben Visual Studio .NET, das ebenfalls das Erstellen von COM-Add-Ins erlaubt, erschien vor kurzer Zeit eine neue Alternative: twinBASIC. Ein Projekt von Wayne Phillips, das sich nicht nur anschickt, Nachfolger von Visual Studio 6 zu werden, sondern schon jetzt das Erstellen unter anderem von COM-Add-Ins ermöglicht. Dieser Artikel stellt die grundlegenden Techniken zum Erstellen eines Gerüsts für COM-Add-Ins vor, das wir in weiteren Artikeln mit praktischen Lösungen für die Erweiterung der Office-Anwendungen und auch des VBA-Editors nutzen werden.

Vorbereitungen

Wie wir **twinBASIC** herunterladen und betriebsbereit machen, beschreiben wir im Artikel **twinBASIC: Visual Basic für die Zukunft** (**www.vbentwickler.de/310**).

Nach dem Start von **twinBASIC** finden wir auf der zweiten Seite des Startdialogs namens **Samples** einige Beispielprojekte vor.

Erstellen eines COM-Add-In-Projekts

Wir starten mit einer der praktischen Vorlagen für die Erstellung verschiedener Beispielprojekte, in diesem Fall **Sample 5. MyCOMAddIn** (siehe Bild 1).

Ein Klick auf diese Schaltfläche erstellt ein neues Projekt, das zu diesem Zeitpunkt jedoch noch nicht gespeichert ist. Um dieses zu speichern, betätigen wir den Menübefehl **File|Save Project**. Dies öffnet, da das Projekt noch ungespeichert ist, einen **Save Project As...**-Dialog. Mit diesem navigieren wir zu dem gewünschten Zielordner und geben den Dateinamen für das Projekt ein, der die Dateiendung **.twinproj** trägt (siehe Bild 2).

Schauen wir dann in das Verzeichnis, in dem wir das neue Projekt angelegt haben, stellen wir fest, dass hier tatsächlich nur eine Datei vorliegt – es gibt aktuell noch keine weiteren Elemente.

Bild 1: Erstellen eines COM-Add-Ins auf Basis des passenden Beispiels

Damit kehren wir zurück zur **twinBASIC**-Entwicklungsumgebung, die das neu erstellte Projekt anzeigt.

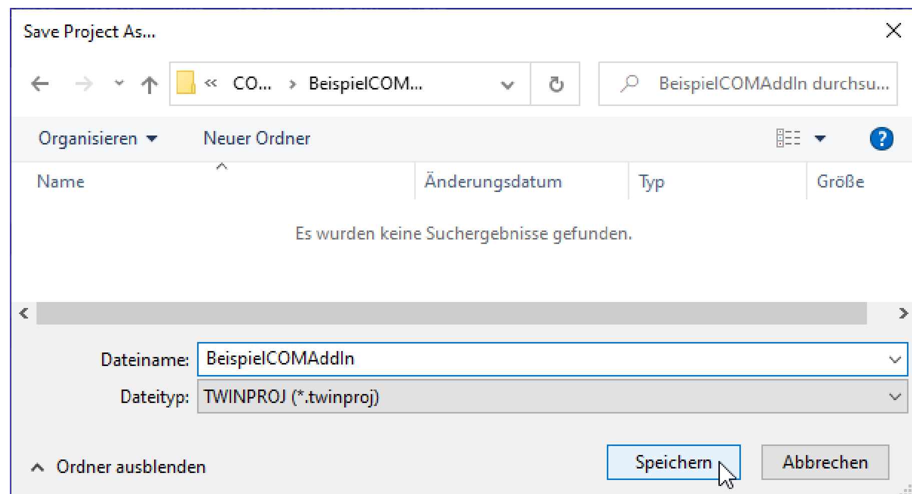

Bild 2: Festlegen des Speicherorts für die **.twinproj**-Datei

Module des Beispielprojekts

Das Projekt enthält zu diesem Zeitpunkt zwei Module:

- Das erste heißt **MyCOMAddin.twin** und wird direkt beim Start im zentralen Bereich der Entwicklungsumgebung angezeigt. Es enthält im Falle eines COM-Add-Ins den gesamten Code des Projekts. Wir schauen uns diesen weiter unten an.
- Das zweite ist das Modul **DllRegistration.twin**. Dieses enthält Informationen, die erstens beim Erstellen des Projekts zum Testen in die Registry geschrieben werden und die auch beim manuellen Registrieren mit **RegSvr32.exe** verwendet werden. Auch den Inhalt dieses Moduls schauen wir uns gleich im Detail an.

Für welche Anwendung soll das COM-Add-In eingesetzt werden?

Die wichtigste Frage, die wir beantworten müssen, bevor wir das Add-In überhaupt testweise erstellen, ist die nach der Anwendung, in der das Add-In seine Funktionen zur Verfügung stellen soll.

Aktuell gehen wir davon aus, dass es sich dabei um eine der folgenden Office-Anwendungen handelt:

- Access
- Excel
- Outlook
- PowerPoint
- Word

Oder wollen wir das COM-Add-In sogar für mehrere Office-Anwendungen gleichzeitig verfügbar machen? Das ist möglich, aber es stellt sich die Frage, ob es Anwendungsfälle gibt, die in allen Office-Anwendungen nützlich sind. Tatsächlich gehen wir davon aus, dass ein COM-Add-In eher auf eine Office-Anwendung spezialisiert ist. Dennoch zeigen wir, wie wir das COM-Add-In für mehrere Office-Anwendungen gleichzeitig bereitstellen können.

Soll das COM-Add-In für eine 32-Bit- oder für eine 64-Bit-Anwendung kompiliert werden?

Die zweite Frage, die sich stellt, ist die nach der Architektur der Office-Installation: Ist diese für 32-Bit- oder für 64-Bit ausgelegt? Wir befinden uns in einem Wandel, denn bis vor kurzer Zeit hat das Office-Setup standardmäßig die 32-Bit-Version von Office installiert. Mit Office 2016 und den entsprechenden Versionen unter Office 365 hat sich dies jedoch geändert. Hier landet nun standardmäßig die 64-Bit-Version auf dem Rechner.

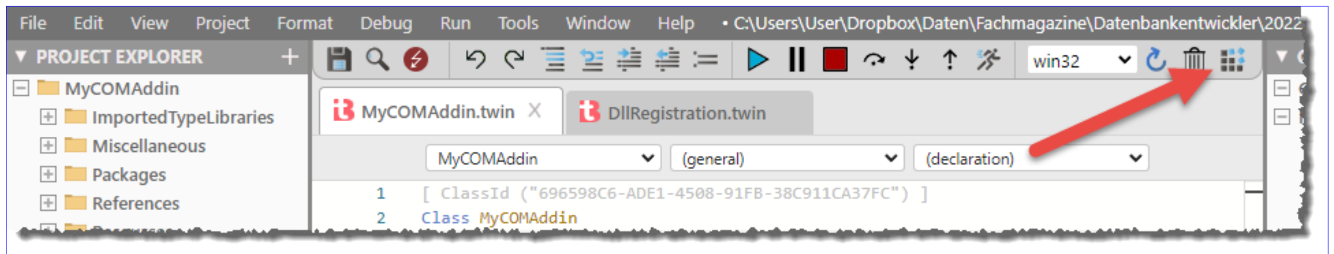

Bild 3: Erstellen des COM-Add-Ins

Das Problem ist nun, dass auch COM-Add-Ins für die entsprechende Version, also 32-Bit oder 64-Bit, kompiliert werden müssen. Unter twinBASIC ist das ein kleines Problem, wenn auch mit gewissen Einschränkungen. Wir können zwar die entsprechende Zielversion (**win32** oder **win64**) vor der Kompilierung auswählen. Allerdings ist twinBASIC nur für die 32-Bit-Version kostenlos. Wenn wir die 64-Bit-Version kompilieren, wird sowohl während der Kompilierung als auch bei Start des COM-Add-Ins jeweils ein Dialog des Herstellers mit dem **twinBASIC**-Logo eingeblendet.

Für Entwicklungszwecke und zum Ausprobieren ist das kein Problem. Wir gehen jedoch davon aus, dass wir, wenn wir mit **twinBASIC** erstellte Anwendungen beim Kunden einsetzen wollen, auch die anfallenden (überschaubaren) Lizenzgebühren gerne entrichten.

Schnellstart mit dem Beispiel-COM-Add-In

Wir wollen direkt einmal das vorgefertigte Beispiel ausprobieren und klicken nach der Auswahl der Zielversion (32-Bit oder 64-Bit) direkt auf die Schaltfläche **Build** in der Menüleiste (siehe Bild 3).

Danach können wir das COM-Add-In in Excel und auch in Access ausprobieren. Wenn wir Excel öffnen, finden wir die Möglichkeit zum Aufrufen der Funktion des COM-Add-Ins im Ribbon unter **twinBASIC Test**. Ein Klick auf die in diesem Tab enthaltene Schaltfläche zeigt ein Meldungsfenster an (siehe Bild 4).

Das gleiche Ergebnis erhalten wir mit diesem COM-Add-In auch, wenn wir Access öffnen.

Anpassung der Zielanwendung

Der erste Schritt der Anpassung des COM-Add-Ins soll sich auf die Anwendung beziehen, in welcher dieses angezeigt wird. Diese Einstellung nehmen wir im Modul **DllRegistration** vor.

Hier finden wir den Code aus Listing 1. Wir sehen ganz oben, dass unter **twinBASIC** Standardmodule in **Module [Modulname]** und **End Module** eingefasst werden müssen. Wir sehen außerdem einige Konstanten, welche Informationen enthalten, die in den beiden weiter unten definierten Funktionen zum Einsatz kommen. Die erste speichert den Namen des Projekts,

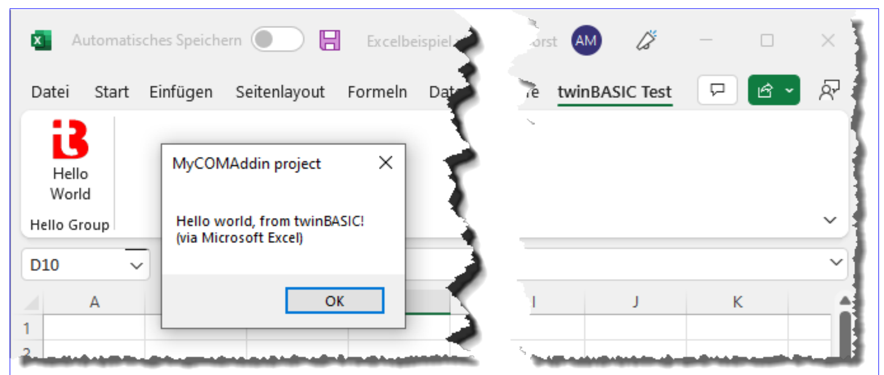

Bild 4: Aufruf des COM-Add-Ins von Excel aus

```
Module DllRegistration
    Const AddinProjectName As String = VBA.Compilation.CurrentProjectName
    Const AddinClassName As String = "MyCOMAddin"
    Const AddinQualifiedClassName As String = AddinProjectName & "." & AddinClassName
    Const RootRegistryFolder_ACCESS As String = "HKCU\SOFTWARE\Microsoft\Office\Access\Addins\" & AddinQualifiedClassName & "\"
    Const RootRegistryFolder_EXCEL As String = "HKCU\SOFTWARE\Microsoft\Office\Excel\Addins\" & AddinQualifiedClassName & "\"

    Public Function DllRegisterServer() As Boolean
        On Error GoTo RegError
        Dim wscript As Object = CreateObject("wscript.shell")
        wscript.RegWrite RootRegistryFolder_ACCESS & "FriendlyName", AddinProjectName, "REG_SZ"
        wscript.RegWrite RootRegistryFolder_ACCESS & "Description", AddinProjectName, "REG_SZ"
        wscript.RegWrite RootRegistryFolder_ACCESS & "LoadBehavior", 3, "REG_DWORD"
        wscript.RegWrite RootRegistryFolder_EXCEL & "FriendlyName", AddinProjectName, "REG_SZ"
        wscript.RegWrite RootRegistryFolder_EXCEL & "Description", AddinProjectName, "REG_SZ"
        wscript.RegWrite RootRegistryFolder_EXCEL & "LoadBehavior", 3, "REG_DWORD"
        Return True
    RegError:
        MsgBox "DllRegisterServer -- An error occured trying to write to the system registry:" & vbCrLf & _
                Err.Description & " (" & Hex(Err.Number) & ")"
        Return False
    End Function

    Public Function DllUnregisterServer() As Boolean
        On Error GoTo RegError
        Dim wscript As Object = CreateObject("wscript.shell")
        wscript.RegDelete RootRegistryFolder_ACCESS & "FriendlyName"
        wscript.RegDelete RootRegistryFolder_ACCESS & "Description"
        wscript.RegDelete RootRegistryFolder_ACCESS & "LoadBehavior"
        wscript.RegDelete RootRegistryFolder_ACCESS
        wscript.RegDelete RootRegistryFolder_EXCEL & "FriendlyName"
        wscript.RegDelete RootRegistryFolder_EXCEL & "Description"
        wscript.RegDelete RootRegistryFolder_EXCEL & "LoadBehavior"
        wscript.RegDelete RootRegistryFolder_EXCEL
        Return True
    RegError:
        MsgBox "DllUnregisterServer -- An error occured trying to delete from the system registry:" & vbCrLf & _
                Err.Description & " (" & Hex(Err.Number) & ")"
        Return False
    End Function
End Module
```

Listing 1: Beispielcode für die Registrierung des COM-Add-Ins

der aus **VBA.Compilation.CurrentProjectName** bezogen wird. Die Klasse **Compilation** ist eine **twinBASIC**-Erweiterung, die Informationen über das aktuelle Projekt liefert und deren Member wir beispielsweise wie in Bild 5 per IntelliSense abfragen können. In diesem Fall ermitteln wir den aktuellen Projektnamen.

Wie wir den Projektnamen anpassen, zeigen wir weiter unten.

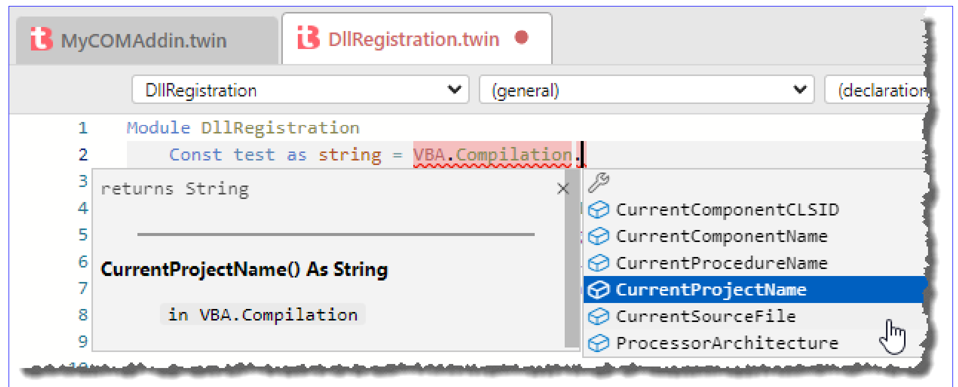

Bild 5: Die Compilation-Klasse per IntelliSense

Die zweite Konstante namens **AddinClassName** erhält den Namen der Klasse des Add-Ins. Dieser sollte mit dem Namen der Klasse im anderen Modul des Projekts übereinstimmen, in diesem Fall **MyCOMAddin**. Die nächste Konstante **AddinQualifiedClassName** fügt die beiden vorher definierten Konstanten, also **AddinProjectName** und **AddinClassName** zusammen.

Schließlich folgen noch zwei Konstanten namens **RootRegistryFolder_ACCESS** und **RootRegistryFolder_EXCEL**, welche die Pfade zu den zu erstellenden Einträgen für das COM-Add-In in der Registry aufnehmen. Für Access entsteht so beispielsweise der folgende Wert für diese Konstante:

```
HKCU\SOFTWARE\Microsoft\Office\Access\Addins\MyCOMAddin.
MyCOMAddin\
```

Wozu Registry-Einträge?

Warum machen wir in Zusammenhang mit der Installation eines COM-Add-Ins überhaupt so einen Wirbel um Registry-Einträge? Der Grund ist einfach: Die Office-Anwendungen scannen beim Starten einen bestimmten Bereich der Registry, wo die für die jeweilige Anwendung registrierten COM-Add-Ins aufgelistet werden. Die dort angegebenen DLLs werden beim Start der Anwendung ausgelesen und dort untergebrachte Schnittstellen für die Anpassung von Ribbondefinitionen angewendet.

Um zu prüfen, ob das soeben erstellte COM-Add-In seinen Platz in der Registry gefunden hat, brauchen wir eigentlich nicht die Registry zu öffnen, denn wir haben uns ja durch Starten der Anwendungen **Access** und **Excel** davon überzeugt, dass das COM-Add-In für die beiden Anwendungen installiert ist.

Dennoch schauen wir uns die Registry an, wobei wir diese mit dem Befehl **RegEdit** über das **Suchen**-Feld von Windows öffnen.

Hier navigieren wir zum folgenden Eintrag:

```
Computer\HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\Ex-
cel\Addins\MyCOMAddin.MyCOMAddin
```

Dies ist beispielsweise der Eintrag für Excel. Für diesen wurden einige Werte hinterlegt (siehe Bild 6).

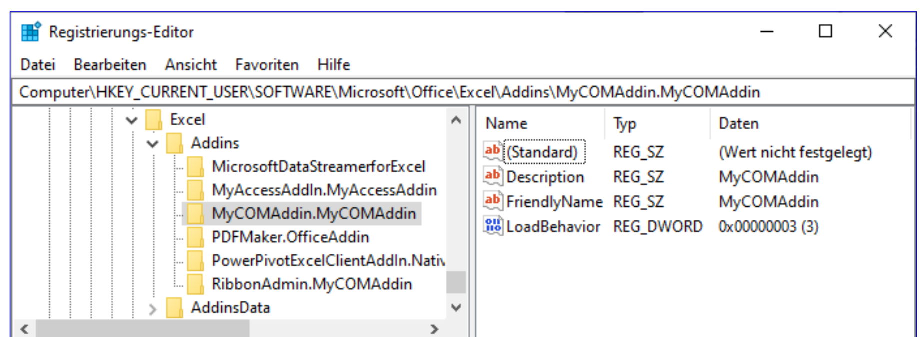

Bild 6: Das Add-In für Excel in der Registry

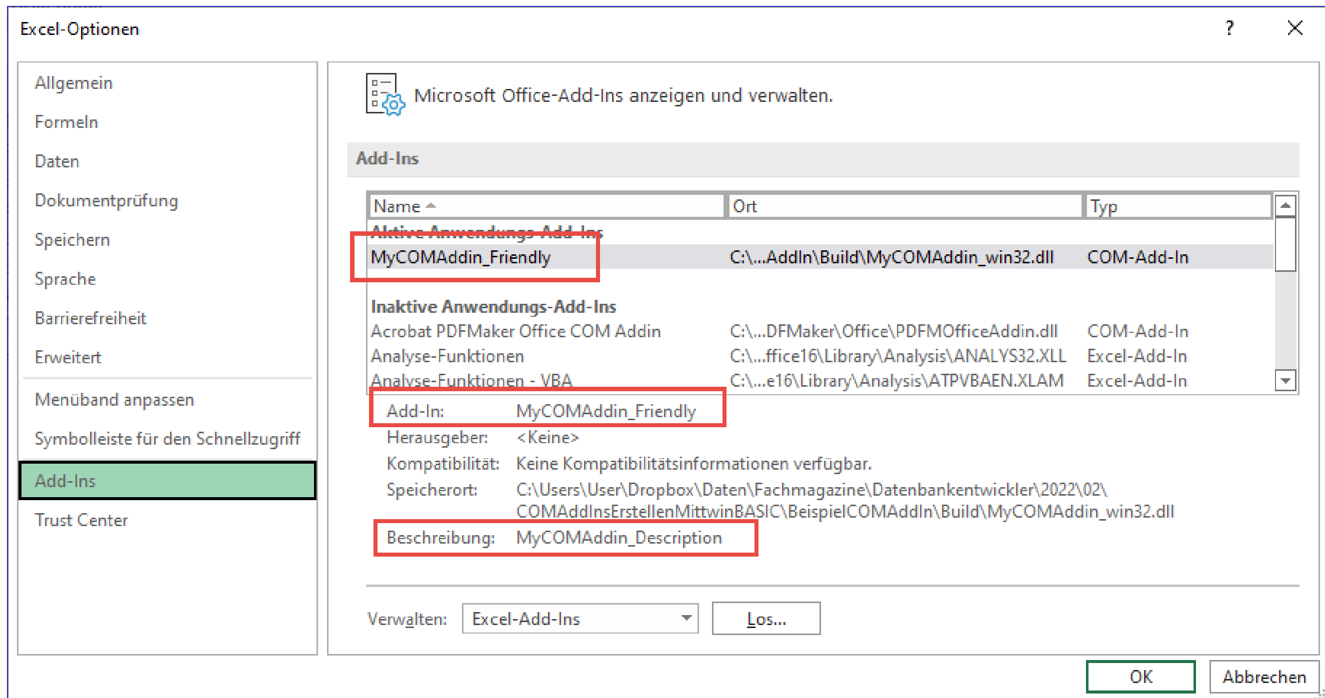

Bild 7: COM-Add-Ins in den Excel-Optionen

Diese haben die folgende Bedeutung:

- **Description**: Beschreibung, die beispielsweise im Bereich **Add-Ins** der Optionen der jeweiligen Anwendung auftaucht
- **FriendlyName**: Name des COM-Add-Ins, wie er in der Liste der COM-Add-Ins im Dialog **COM-Add-Ins** auftaucht
- **LoadBehaviour**: Verhalten beim Laden. Normalerweise soll hier der Wert **3** stehen für **Beim Start Laden**. Diesen Wert sehen wir ebenfalls im Dialog **COM-Add-Ins**.

Die Werte für **FriendlyName** und **Description** sowie den Speicherort der DLL finden wir nach der Installation des COM-Add-Ins beispielsweise für Excel im Optionen-Dialog im Bereich **Add-Ins**.

Die einzelnen Werte haben wir in Bild 7 hervorgehoben.

Wählen wir hier im unteren Bereich für das Kombinationsfeld **Verwalten** den Wert **COM-Add-Ins** aus und klicken auf die Schaltfläche **Los...**, finden wir den Dialog aus Bild 8 vor. Hier sehen wir auch den aktuellen Wert für die Eigenschaft **Ladeverhalten**, in diesem Fall **Beim Start laden**.

Funktion zum Einrichten der Registry-Einträge

Nachdem wir uns nun mit der Bedeutung des Vorhandenseins gewisser Registry-Einträge beschäftigt haben, wollen wir uns ansehen, wie diese dorthin gelangen.

Diese Aufgabe übernimmt die Funktion **DllRegisterServer** im Modul **DllRegistration**. Diese Funktion erzeugt eine Instanz des Objekt **wscript.shell**, welches beispielsweise die Methode **RegWrite** zum Schreiben von Registry-Einträgen bereitstellt. Diese nutzen wir, um die benötigten Einträge mit den Werten, die teilweise aus den oben definierten Konstanten stammen, zur Registry hinzuzufügen.

Das alles geschieht unter Deaktivierung der eingebauten Fehlerbehandlung mit **On Error Goto ReGError**. Tritt beim Anlegen der Registry-Einträge ein Fehler auf, zeigt die Funktion ein Meldungsfenster mit Details zum aufgetretenen Fehler an.

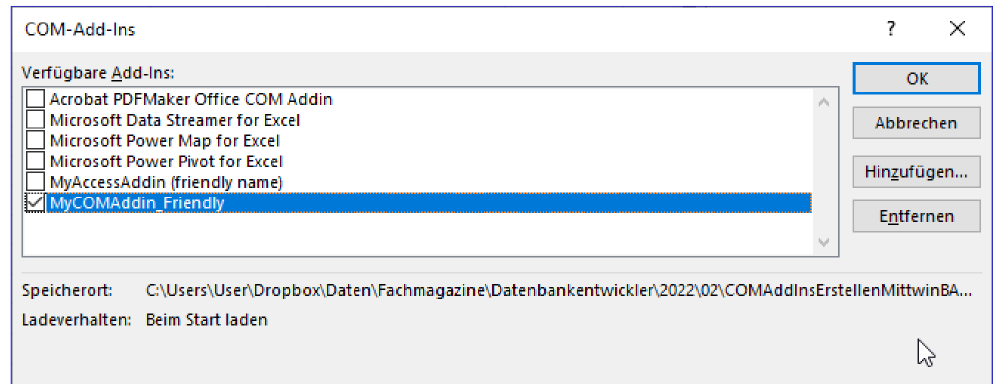

Bild 8: Der Dialog **COM-Add-Ins**

Wann wird DllRegisterServer aufgerufen?

Es gibt zwei Gelegenheiten, zu denen die Funktion **DllRegister** aufgerufen wird. Die erste ist das Betätigen der **Build**-Schaltfläche in der **twinBASIC**-Entwicklungsumgebung.

Ob sie dadurch tatsächlich aufgerufen wird, können wir anschließend am Vorhandensein der dort definierten Registry-Einträge feststellen. Wir können aber auch einfach eine **MsgBox**-Anweisung einbauen und uns beispielsweise den Pfad in der Registry ausgeben lassen, in welchen die Registry-Informationen geschrieben werden.

Der zweite Zeitpunkt, zu dem diese Funktion aufgerufen wird, ist die Registrierung auf dem jeweiligen Zielrechner mit der Anweisung **RegSvr32.exe** entweder über die Eingabeaufforderung oder auch per Batch-Datei.

Über die Eingabeaufforderung gelingt das wie folgt:

- Als Erstes starten wir die Eingabeaufforderung, und zwar im Administrator-Modus. Dazu geben wir im Suchen-Feld von Windows den Text **cmd** ein und klicken dann mit der rechten Maustaste auf den nun im Suchergebnis erscheinenden Eintrag **Eingabeaufforderung**. Im Kontextmenü wählen wir **Als Administrator ausführen** aus.
- Danach navigieren wir zu dem Verzeichnis, in dem sich die **.dll**-Datei befindet, in diesem Fall auf dem Desktop.
- Hier geben wir den Befehl **Regsvr32.exe MyCOMAddin_win64.dll** ein.
- Hier erhalten wir dann im Optimalfall die Meldung, dass die DLL erfolgreich registriert wurde (siehe Bild 9).

Wenn wir die COM-Add-In-Datei also auf einen anderen Rechner kopiert haben und diese installieren wollen, müssen wir diese mehr oder weniger von Hand registrieren, wenn wir nicht gerade ein Setup für das COM-Add-In erstellt haben. Wie wir ein solches Setup erstellen, erläutern wir noch in einem weiteren Beitrag.

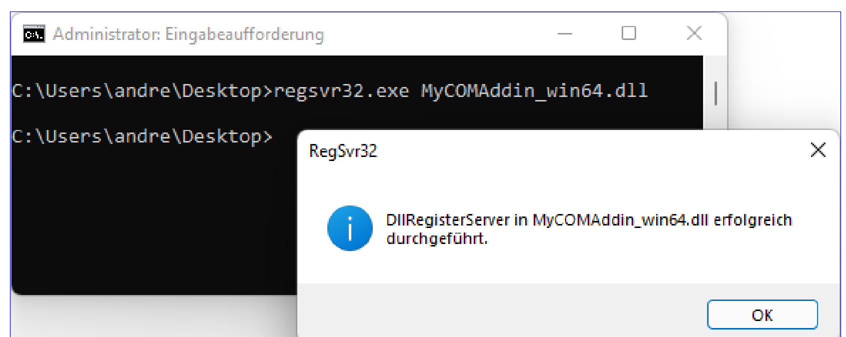

Bild 9: Installieren per Eingabeaufforderung

Wir sehen, dass wir die **twinBASIC**-Funktion **DllRegisterServer** auch von der Kommandozeile aus aufrufen können. Das eröffnet uns auch die Möglichkeit, dort weitere Mechanismen unterzubringen, die für die Verwendung des COM-Add-Ins noch hilfreich sein können – beispielsweise das Anlegen bestimmter Dateien mit Konfigurationsdateien, die Anzeige von Meldungen während der Installation et cetera.

Den Befehl **Regsvr32.exe** können wir in eine Batch-Datei schreiben und diese dann im Administratormodus aufrufen. Diese Datei nennen wir einfach **Register.bat**. Wenn wir diese Datei im gleichen Verzeichnis platzieren wie die zu registrierende Datei, brauchen wir noch nicht mal den Pfad zur zu registrierenden **.dll**-Datei anzugeben, wie es eigentlich nötig wäre, wenn wir nicht in der Eingabeaufforderung zum Verzeichnis mit der **.dll**-Datei navigiert hätten.

Funktion zum Entfernen der Registry-Einträge

Wenn wir ein Element durch Hinzufügen von Einträgen zur Registry installieren, wollen wir dieses auch wieder deinstallieren können. Dazu können wir die Funktion **DllUnregisterServer** nutzen. Der Aufbau dieser Funktion ist dem der Prozedur zum Hinzufügen der Registry-Einträge recht ähnlich. Allerdings nutzen wir hier nicht die Methode **RegWrite** der Klasse **wscript.shell**, sondern die Methode **RegDelete**.

Hier sind nicht nur drei Aufrufe nötig, sondern noch ein weiterer. Beim Registrieren werden nämlich automatisch alle Ordner angelegt, die zu den hinzuzufügenden Registry-Einträgen führen – in diesem Fall der Ordner aus der Konstanten **AddinQualifiedClassName**. Den so angelegten Ordner müssen wir beim Deinstallieren nach dem Löschen der eigentlichen Registry-Einträge separat wieder entfernen.

Der Rest läuft wie beim Anlegen der Registry-Einträge ab.

Wenn wir die Registrierung auf dem Zielrechner wieder aufheben wollen, können wir wie beim Registrieren vorgehen – wir rufen in der für den Administratorzugriff geöffneten Eingabeaufforderung den gleichen Befehl auf, diesmal allerdings mit dem Parameter **-u**:

```
Regsvr32 MyCOMAddin_win32.dll -u
```

Bedingungen für das Registrieren und Deregistrieren von COM-Add-Ins

Wenn wir ein COM-Add-In für eine Anwendung registrieren, können wir dies unabhängig davon erledigen, ob die Zielanwendung aktuell geöffnet ist oder nicht. Die einzige Konsequenz, wenn die Anwendung geöffnet ist, ist folgende: Das COM-Add-In ist dann nicht sofort, sondern erst beim nächsten Öffnen der Anwendung verfügbar.

Beim Registrieren im Rahmen des Erstellens des COM-Add-Ins von der twinBASIC-Entwicklungsumgebung aus sollten wir die Zielanwendungen jedoch immer schließen. Ansonsten kann es vorkommen, dass beim Versuch, die bestehende Version der **.dll**-Datei zu löschen und diese neu zu erstellen, ein Fehler auftritt, weil die **.dll**-Datei gerade in Verwendung ist.

Modul DLLRegistration auf die Zielanwendung anpassen

Wenn wir entschieden haben, für welche Anwendung das COM-Add-In zum Einsatz kommen soll, können wir die beiden Funktionen **DllRegister** und **DllUnregister** entsprechend anpassen. Wir können dann zuerst einen Satz von Registry-Informationen entfernen. Dann passen wir die Funktionen wie folgt an. Die Funktion **DLLRegister** sieht dann wie in Listing 2 aus.

Zu beachten ist auch, dass wir den Namen der Konstanten von **RootRegistryFolder_ACCESS** auf **RootRegistryFolder** geändert haben, da er nur noch für die jeweilige Anwendung genutzt wird.

Ribbon-Signaturen für VBA und VB6/twinBASIC

Wer Anwendungen programmiert, die das Ribbon anpassen, möchte die benutzerdefinierten Ribbon-Steuerelemente auch mit entsprechenden Callback-Prozeduren ausstatten. Dieser Artikel liefert eine Zusammenfassung der Ribbon-Signaturen für die Callback-Funktionen unter VBA und Visual Basic 6 beziehungsweise twinBASIC. Die VBA-Signaturen nutzen wir, wenn wir Ribbon-Erweiterungen direkt zu den mit den Office-Anwendungen erstellten Ribbonanpassungen hinzufügen wollen. Erstellen wir hingegen COM-Add-Ins mit VB6 oder twinBASIC, nutzen wir die alternativen Signaturen.

Wichtige Informationen

Vorab die wichtigste Information: Verwenden Sie alle Callback-Signaturen genau so wie Sie hier abgebildet sind. Manchmal kann bereits das Hinzufügen eines Datentyps zu einem Parameter, der hier keinen Datentyp aufweist, zu Problemen führen. Wichtig ist auch, auf die korrekte Verwendung von **Function/Sub** zu achten. Wenn Sie twinBASIC verwenden, können Sie das Ergebnis zum Zurückgeben entweder einer Variablen mit dem gleichen Namen wie die Funktion zuweisen, oder auch die **Return**-Anweisung nutzen. Die erste Möglichkeit lautet also:

```
Function GetText(control As IRibbonControl, ByRef text) _
        As String
    ...
    GetText = strText
End Function
```

Die Alternative ist:

```
Function GetText(control As IRibbonControl, ByRef text) _
        As String
    ...
    Return strText
End Function
```

Hinweis zu den onAction-Alternativen

Für das **button**- und das **toggleButton**-Element gibt es je zwei Definitionen für die **onAction**-Prozeduren. Du kannst mit dem **command**-Element die Funktion eingebauter Elemente, darunter auch **button**- und **toggleButton**-Elemente, überschreiben beziehungsweise ergänzen.

Wenn Du für diese eine **onAction**-Prozedur hinterlegen möchtest, musst Du die Version in der folgenden Liste verwenden, hinter der in Klammern command steht. Diese Signaturen enthalten noch einen zusätzlichen Parameter, mit dem Du festlegen kannst, ob die eigentliche Funktion des Steuerelements noch ausgeführt werden soll.

Liste der Callback-Signaturen für VBA und Visual Basic 6

```
button       getShowImage: VBA: Sub GetShowImage (control As IRibbonControl, ByRef showImage)
                           VB6: Function GetShowImage (control As IRibbonControl) As Boolean

             getShowLabel: VBA: Sub GetShowLabel (control As IRibbonControl, ByRef showLabel)
                           VB6: Function GetShowLabel (control As IRibbonControl) As Boolean

             onAction:     VBA: Sub OnAction(control As IRibbonControl)
                           VB6: Sub OnAction(control As IRibbonControl)
```

	onAction (command):	VBA: Sub OnAction(control As IRibbonControl, ByRef CancelDefault) VB6: Sub OnAction(control As IRibbonControl, ByRef CancelDefault)
checkBox	getPressed:	VBA: Sub GetPressed(control As IRibbonControl, ByRef returnValue) VB6: Function GetPressed(control As IRibbonControl) As Boolean
	onAction:	VBA: Sub OnAction(control As IRibbonControl, pressed As Boolean) VB6: Sub OnAction(control As IRibbonControl, pressed As Boolean)
comboBox	getItem-Count:	VBA: Sub GetItemCount(control As IRibbonControl, ByRef count) VB6: Function GetItemCount(control As IRibbonControl) As Integer
	getItemID:	VBA: Sub GetItemID(control As IRibbonControl, index As Integer, ByRef id) VB6: Function GetItemID(control As IRibbonControl, index As Integer) As String
	getItem-Image:	VBA: Sub GetItemImage(control As IRibbonControl, index As Integer, ByRef image) VB6: Function GetItemImage(control As IRibbonControl, index As Integer) As IPictureDisp
	getItem-Label:	VBA: Sub GetItemLabel(control As IRibbonControl, index As Integer, ByRef label) VB6: Function GetItemLabel(control As IRibbonControl, index As Integer) As String
	getItem-ScreenTip:	VBA: Sub GetItemScreenTip(control As IRibbonControl, index As Integer, ByRef screentip) VB6: Function GetItemScreentip(control As IRibbonControl, index As Integer) As String
	getItem-SuperTip:	VBA: Sub GetItemSuperTip (control As IRibbonControl, index As Integer, ByRef supertip) VB6: Function GetItemSuperTip (control As IRibbonControl, index As Integer) As String
	getText:	VBA: Sub GetText(control As IRibbonControl, ByRef text) VB6: Function GetText(control As IRibbonControl) As String
	onChange:	VBA: Sub OnChange(control As IRibbonControl, text As String) VB6: Sub OnChange(control As IRibbonControl, text As String)
customUI	loadImage:	VBA: Sub LoadImage(imageId As string, ByRef image) VB6: Function LoadImage(imageId As String) As IPictureDisp
	onLoad:	VBA: Sub OnLoad(ribbon As IRibbonUI) VB6: Function OnLoad(ribbon As IRibbonUI)
dropDown	getItem-Count:	VBA: Sub GetItemCount(control As IRibbonControl, ByRef count) VB6: Function GetItemCount(control As IRibbonControl) As Integer
	getItemID:	VBA: Sub GetItemID(control As IRibbonControl, index As Integer, ByRef id) VB6: Function GetItemID(control As IRibbonControl, index As Integer) As String
	getItem-Image:	VBA: Sub GetItemImage(control As IRibbonControl, index As Integer, ByRef image) VB6: Function GetItemImage(control As IRibbonControl, index As Integer) As IPictureDisp
	getItem-Label:	VBA: Sub GetItemLabel(control As IRibbonControl, index As Integer, ByRef label) VB6: Function GetItemLabel(control As IRibbonControl, index As Integer) As String

YouTube-Kanal mit VB.NET verwalten, Teil 1

Wie vielleicht einige wissen, füttere ich seit kurzer Zeit einen YouTube-Kanal mit Access-Videos (sicher werden bald auch welche zu den Themen aus »Visual Basic Entwickler« folgen). Um den Zuschauern der Videos den besten Service bieten zu können, möchte ich in den Beschreibungstexten der YouTube-Videos auf andere passende Videos und auch auf Artikel aus den Magazinen verweisen. Da immer wieder neue Videos und Artikel hinzukommen, die thematisch zusammenpassen, möchte ich natürlich auch die Verlinkung in den Beschreibungstexten aktuell halten. Das ist allerdings eine Menge Aufwand, wenn man das immer von Hand erledigen muss. Da bietet es sich doch an, einmal einen Blick auf die Möglichkeiten des codegesteuerten Zugriffs auf einen YouTube-Kanal zu werfen – und damit Funktionen wie das Einlesen der vorhandenen Playlisten und Videos sowie das Aktualisieren von Inhalten wie den Beschreibungstexten zu automatisieren.

YouTube-Videos verwalten in YouTube Studio

Das Verwalten eines YouTube-Kanals ist eigentlich recht unkompliziert: YouTube bietet mit YouTube Studio eine umfangreiche Plattform für das Hinzufügen, Ändern und Löschen von Videos an. In Bild 1 siehst Du die Übersicht der Videos eines Kanals in YouTube Studio – hier lassen sich die Videos zur Bearbeitung anzeigen und über den **Erstellen**-Button oben rechts fügt man schnell neue Videos hinzu. Außerdem be-

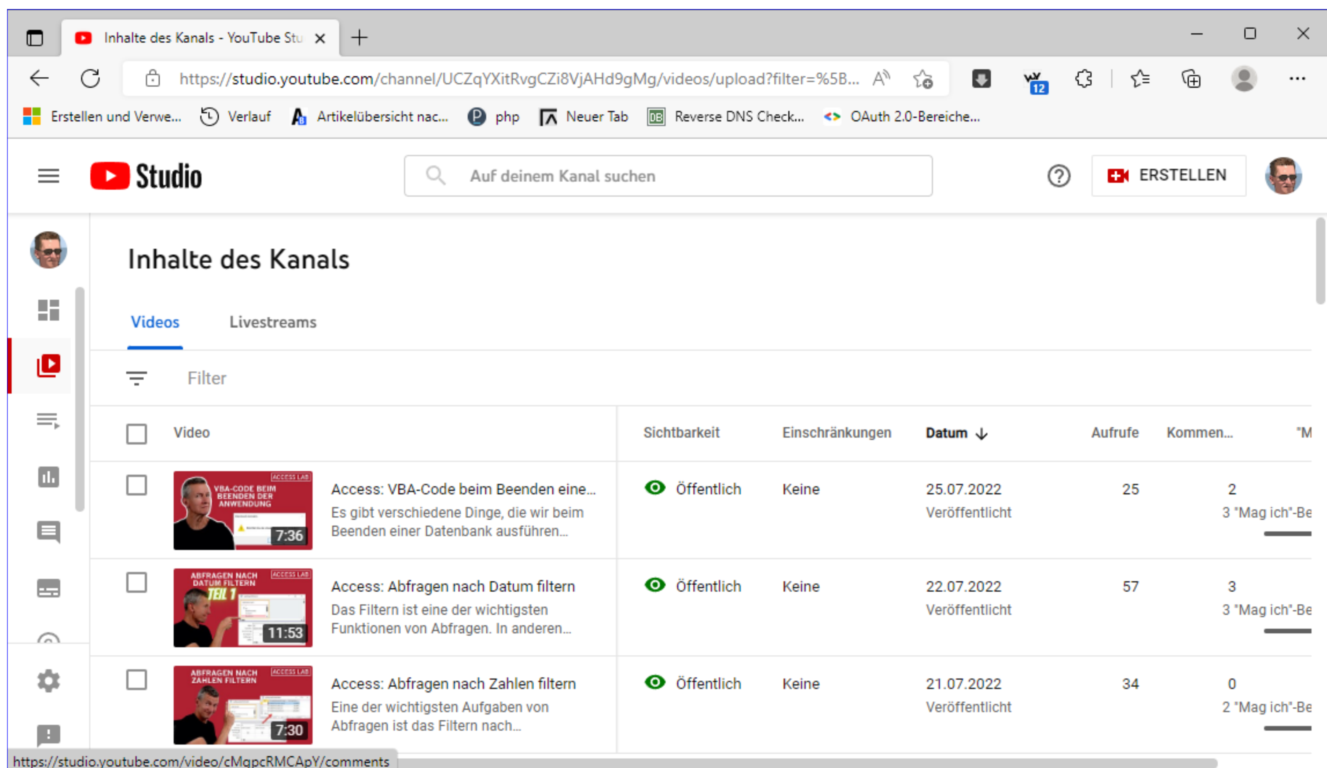

Bild 1: Verwalten eines YouTube-Kanals über YouTube Studio

kommen wir hier schnell Informationen über Zugriffe, Kommentare und so weiter.

Workflow optimieren

Vom ersten Schritt bis zur Veröffentlichung eines YouTube-Videos sind einige Aufgaben zu erledigen. Thema auswählen, eine Art Drehbuch oder zumindest Stichpunkte zum Video vorbereiten, das Video drehen, das Video schneiden und mit Intro et cetera versehen, das Video hochladen, einen Titel und einen Beschreibungstext erstellen, ein Thumbnail designen und diesen hochladen, weitere Eigenschaften in YouTube Studio einstellen und schließlich das Video veröffentlichen.

Wenn man noch nicht so viele Videos auf seinem Kanal hat, ist das Verlinken zu anderen Videos mit ähnlichen Themen noch eine überschaubare Aufgabe.

Wenn die Anzahl aber erst einmal zunimmt und man dann noch Querverweise zu thematisch passenden Artikeln aus seinem Fundus (in meinem Fall weit über 1.000 Artikel) hinzufügen möchte, wird die Arbeit aber schnell unübersichtlich.

Da hilft es dann, wenn man eine Datenbank hat, in der man Artikel und Videos samt Metadaten wie Stichwörtern, Beschreibungstexten et cetera pflegt und hier auch Verweise thematisch ähnlicher Quellen untergebracht hat. Nur: Wenn man diese dann doch händisch aus der Datenbank abfragen und in die Beschreibungstexte der Videos bei YouTube eintragen muss, kostet das dennoch einiges an Zeit.

Erste Idee: Python

Hier kam schnell die Idee auf, zumindest das Zusammenstellen und Hochladen der Beschreibungstexte auf YouTube zu automatisieren. Eine API für YouTube stellt Google zur Verfügung, aber mit welcher Programmiersprache wollen wir dies realisieren? Die ersten vielversprechenden Versuche erfolgten mit Python, aber hier fehlte die Möglichkeit, auf einfache Weise auf die in einer Access- beziehungsweise SQL Server-Datenbank gespeicherten Daten zuzugreifen, um diese mit den Daten bei YouTube abzugleichen.

YouTube-Zugriff mit .NET

Die nächste und in allen Belangen vielversprechendere Idee war, dies mit Visual Basic .NET zu programmieren. Damit können wir über die DAO-Bibliothek sehr einfach auf die Access-Datenbank mit den Video- und Artikeldaten zugreifen. Außerdem sind wir bei der Suche nach fertigen Bibliotheken für den Zugriff auf YouTube schnell fündig geworden.

Nun waren mehrere Schritte nötig:

- Einrichten eines Google-Kontos für das Betreiben des YouTube-Kanals (in unserem Fall schon erledigt)
- Einrichten eines Developer-Kontos bei Google
- Programmieren einer .NET-Lösung für den Zugriff auf den YouTube-Kanal

YouTube- beziehungsweise Google-Konto erstellen

Für das Betreiben eines YouTube-Kanals benötigst Du ein entsprechendes Konto, und da YouTube zu Google gehört, handelt es sich dabei um ein Google-Konto.

Um dieses von youtube.com aus anzulegen, klickst Du einfach rechts oben auf Anmelden. Im folgenden Dialog findest Du einen Link mit dem Text **Konto erstellen**.

In den folgenden Schritten gibst Du einige Informationen ein wie Name, E-Mail-Adresse (alternativ dazu kannst Du eine neue gmail-Adresse erstellen) und Passwort. Außerdem hinterlegst Du nach Wunsch eine Telefonnummer und einige weitere Einstellungen.

Nach dem Einrichten des Google-Kontos kannst Du Dich damit bei **youtube.com** anmelden. Dann findest Du, wenn Du rechts oben auf Dein Logo oder Deinen Anfangsbuchstaben klickst, im Kontextmenü bereits die Möglichkeiten zum Erstellen eines Kanals, zum Anzeigen von **YouTube Studio** und mehr (siehe Bild 2). Damit kannst Du nun schon erste Videos hochladen.

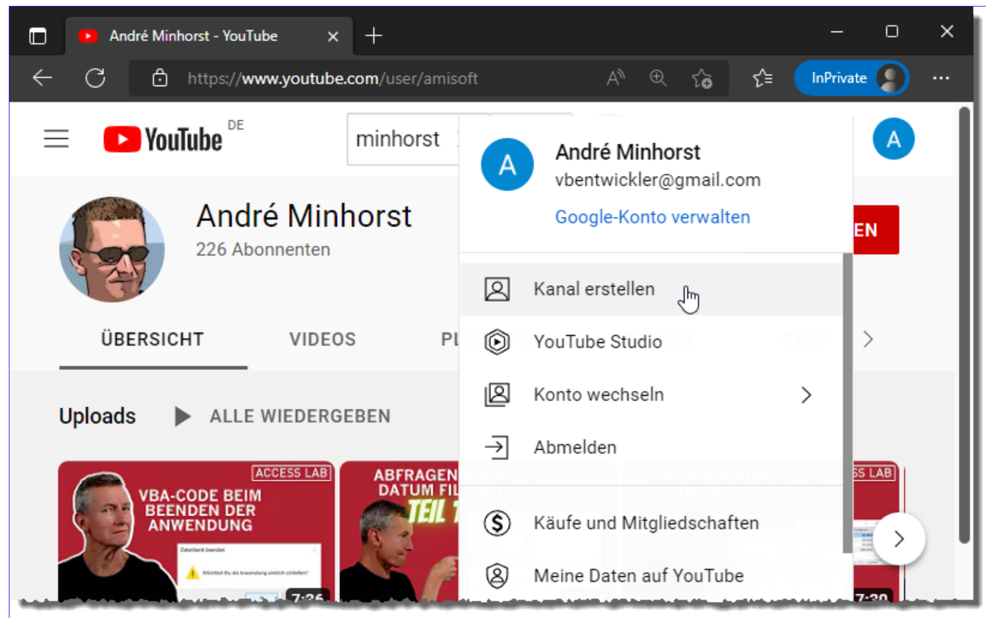

Bild 2: Nach dem Einrichten eines Google-Kontos

Developer-Konto erstellen

Der erste Schritt zu einem Google-Entwicklerkonto führt über die Seite **https://developers.google.com/youtube**. Hier klickst Du oben rechts neben Deinem Logo auf die Schaltfläche mit den drei vertikal angeordneten Punkten, was ein Popup mit der Möglichkeit zum Erstellen eines Google-Entwicklerprofils anzeigt (siehe Bild 3).

In den folgenden Schritten kannst Du ein paar Informationen zu Dir hinterlegen und das Anlegen des Entwicklerprofils abschließen.

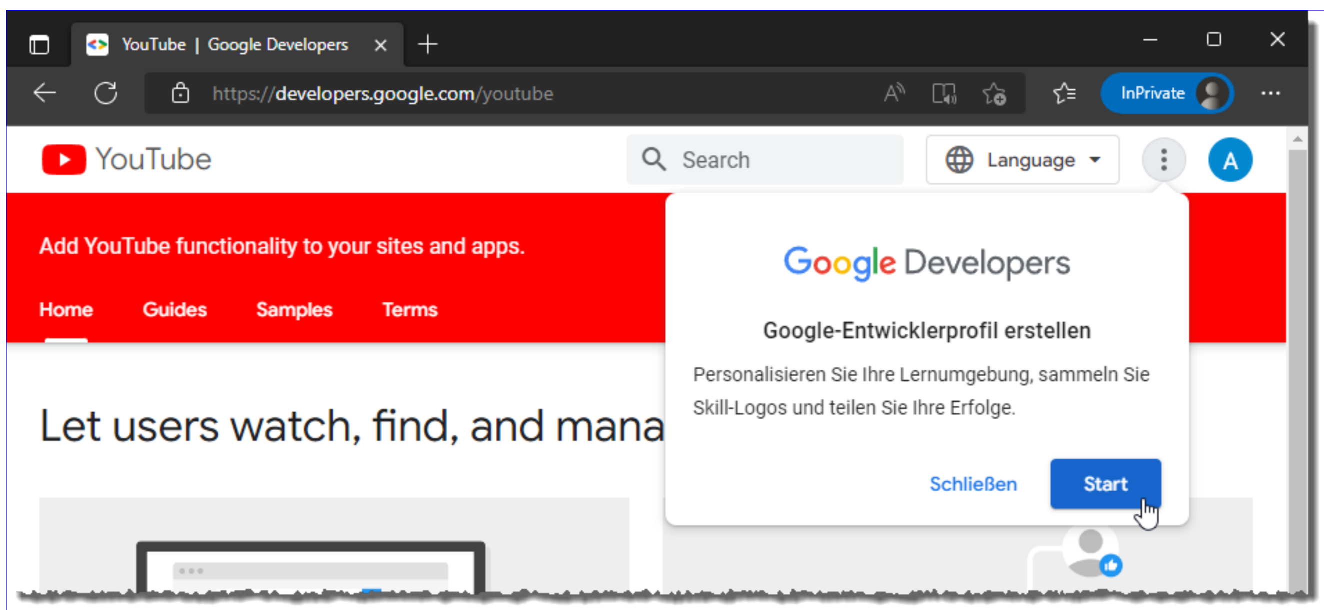

Bild 3: Anlegen eines Entwicklerprofils

Um nun einen Schritt weiterzugehen und beispielsweise die API zu nutzen, rufst Du die folgende Webseite auf:

`https://console.cloud.google.com/`

Hier stimmst Du zunächst den Nutzungsbedingungen zu.

Anschließend erscheint die Google Cloud Plattform-Konsole mit allen weiteren Möglichkeiten.

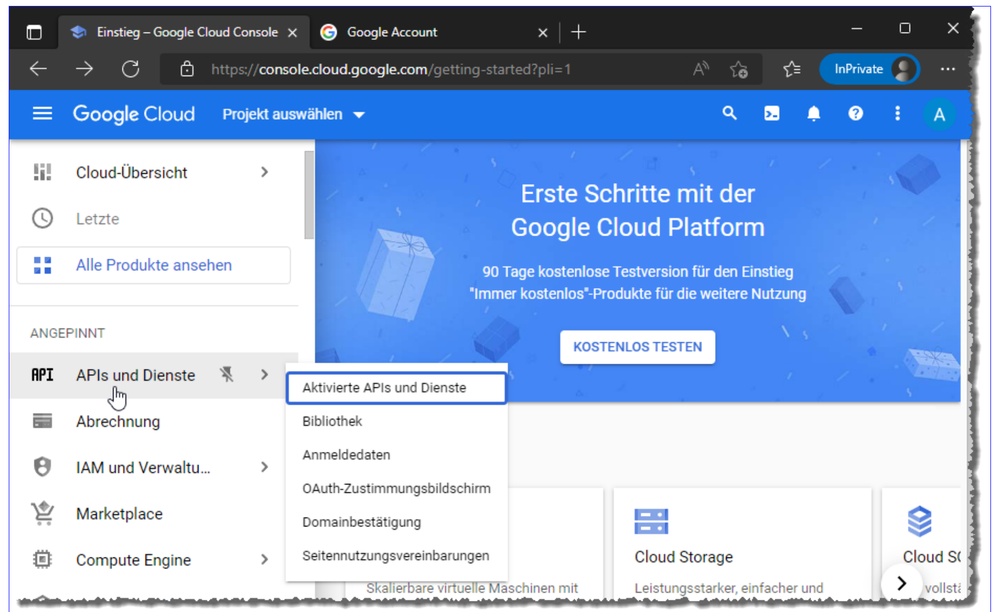

Bild 4: Die Google Cloud Plattform-Konsole

Hier klickst Du auf APIs und Dienste und wechselst zu **Aktivierte APIs und Dienste**. (siehe Bild 4).

Neues Projekt erstellen

Hier findest Du nun einen leeren Bereich, da Du in Deinem neuen Konto noch keine Dienste angelegt hast. Das kannst Du jedoch nun mit einem Klick auf **Projekt erstellen** nachholen (siehe Bild 5).

Dies öffnet das Fenster **Neues Projekt**. Hier gibst Du einen Projektnamen an, der später nicht mehr geändert werden kann. Um das Projekt anzulegen, klickst Du auf die Schaltfläche **Erstellen** (siehe Bild 6).

Dieser Vorgang dauert einige Sekunden. Anschließend erscheint eine Meldung, dass das neue Projekt angelegt wurde, und Du kannst das Projekt auswählen.

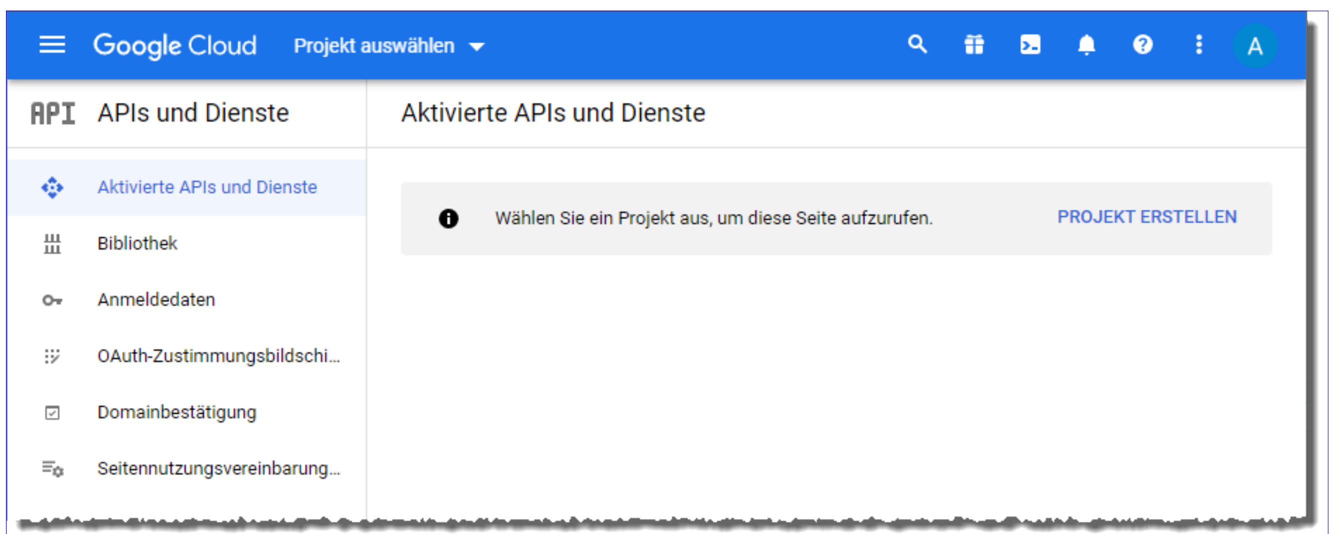

Bild 5: Erstellen eines neuen Projekts

YouTube-Kanal mit VB.NET verwalten, Teil 2

Im ersten Teil dieser Artikelreihe haben wir die Vorbereitungen auf der Seite von Google beziehungsweise YouTube erledigt. Dabei haben wir ein Google-Konto angelegt, dieses zum Entwickler-Konto erweitert und eine Anwendung für den Zugriff auf die YouTube-API registriert. Außerdem haben wir dort die für den Zugriff notwendigen Zugangsdaten erstellt, die wir in der nun zu programmierenden .NET-Lösung verwenden können. In der .NET-Lösung wollen wir verschiedene Informationen zu unserem YouTube-Kanal abrufen wie die Namen der Playlisten und Informationen zu den enthaltenen Videos. Außerdem wollen wir neue Videos anlegen und die Eigenschaften der vorhandenen Videos anpassen.

Die Ausgangsidee zu diesem Artikel war, dass ich einen YouTube-Kanal eröffnet habe und dort Access-Videos veröffentliche. Die Herausforderung dabei ist, dass ich beispielsweise im Beschreibungstext gern Links zu anderen, thematisch ähnlichen Videos hinzufügen würde und auch Links zu entsprechenden Artikeln aus meinen Magazinen. Das Problem dabei ist, dass es immer mehr Videos werden und Artikel gibt es ohnehin schon weit über 1.000 Stück. Außerdem sollen die Beschreibungstexte auch noch Links zu eventuell verfügbaren Beispieldatenbanken enthalten.

Aufwendige Pflege

Dies alles zu pflegen, indem man die Beschreibungstexte von Hand zusammenschreibt und einzeln über YouTube Studio hinzuzufügen, wäre sehr zeitintensiv und deshalb suchen wir einen alternativen Weg. Dieser sieht so aus, dass wir die notwendigen Daten lokal in einer Access-Datenbank speichern, wo sich die Informationen über YouTube-Videos, Artikel und ihre Verknüpfungen befinden. Diese sollen dann per Code zu Beschreibungstexten zusammengestellt und den Videos im YouTube-Kanal hinzugefügt werden.

Datenbankzugriff im zweiten Schritt

Den Schritt, die Daten aus der Datenbank zu beziehen, lassen wir vorerst aus – zunächst einmal wollen wir überhaupt in der Lage sein, auf die Videos des Kanals zuzugreifen, deren Daten auszulesen und Informationen wie den Titel oder den Beschreibungstext und gegebenenfalls auch weitere Eigenschaften zu aktualisieren.

Verwendete Technik

Dass wir die Lösung mit Visual Basic .NET entwickeln wollen, haben wir bereits im ersten Teil der Artikelreihe erläutert. Es gibt alle notwendigen Bibliotheken für den Zugriff auf die **YouTube Data API v3** und wir können von einer .NET-Anwendung auch auf die Daten einer Access-Datenbank zugreifen.

Gegebenenfalls ist letzteres aber auch gar nicht notwendig – wir könnten auch eine DLL programmieren, die wir in das VBA-Projekt einer Access-Datenbank einbinden. Von dort können wir dann die Methoden und Eigenschaften dieser DLL nutzen und innerhalb der Access-Datenbank die darüber zu schreibenden Daten aus den Tabellen lesen.

Welcher Projekttyp?

Deshalb stellt sich die Frage, mit welchem Projekttyp wir starten. Auch wenn wir die Funktionen später in einer .NET-DLL veröffentlichen wollen, entscheiden wir uns an dieser Stelle für eine WPF-Anwendung. Den Zugriff auf die Access-Datenbank mit den Daten für die YouTube-Videos wollen wir zunächst aussparen und stattdessen erst einmal den Zugriff auf die Daten des YouTube-Kanals realisieren sowie einige Beispieldaten in bestehende YouTube-Videos schreiben.

Projekt erstellen und Pakete hinzufügen

Im ersten Ansatz und zum Experimentieren erstellen wir ein neues Projekt des Typs **WPF-Anwendung** für **Visual Basic**. In diesem Fall haben wir die Variante für **.NET Core** verwendet. Außerdem benötigen wir zwei NuGet-Pakete. Um diese hinzuzufügen, öffnen wir mit dem Menübefehl **Projekt|NuGet-Pakete verwalten...** das **NuGet**-Fenster.

Hier wechseln wir zum Bereich **Durchsuchen** und suchen nacheinander nach den folgenden NuGet-Paketen und installieren diese jeweils mit einem Klick auf **Installieren**:

- **Google.Apis.Oauth2.v2**: Ermöglicht die Authentifizierung für den Zugriff auf die YouTube-Daten
- **Google.Apis.YouTube.V3**: Ermöglicht den Zugriff auf die YouTube-Daten.

Wenn wir nach der Installation zum Bereich **Installiert** wechseln, sollte dieser wie in Bild 1 aussehen.

Programmierung der Authentifizierung

Ziel der nachfolgend beschriebenen Authentifizierung ist es, ein Objekt des Typs **YouTubeService** zu erhalten, mit dem wir auf die Daten des YouTube-Kanals des Benutzers zugreifen können, in dessen Kontext wir uns anmelden. Diese Klasse (siehe Listing 1) bezeichnen wir mit Authentication und sie enthält nur eine Funktion namens **AuthenticateOAuth**. Diese Funktion liefert nach erfolgreicher Authentifizierung das gewünschte **YouTubeService**-Objekt zurück.

Die Frage ist nur: Woher bekommen wir die für die Authentifizierung notwendigen Daten? Dazu gehören zwei verschiedene Sätze von Informationen:

- Die Daten für die App, mit der wir uns anmelden wollen und
- die Daten für den Benutzer, in dessen Kontext die Anmeldung erfolgen soll.

Die Daten für die App haben wir im ersten Teil der Artikelreihe mit dem Titel **YouTube-Kanal mit VB.NET verwalten, Teil 1** (**www.vbentwickler.de/315**) von unserem Google-Entwicklerkonto als JSON-Datei heruntergeladen und unter dem Namen **client_secrets.json** gespeichert. Den zweiten Satz von Daten, die Daten des Benutzers, geben wir bei der eigentlichen Authentifizierung an. Zunächst benötigen wir nur die Daten aus der Datei **client_secrets.json**. Der Einfachheit halber platzieren wir diese in dem Verzeichnis, in dem beim Debuggen der Anwendung auch die dabei erstellte **.exe**-Datei landet. Dabei handelt es sich um

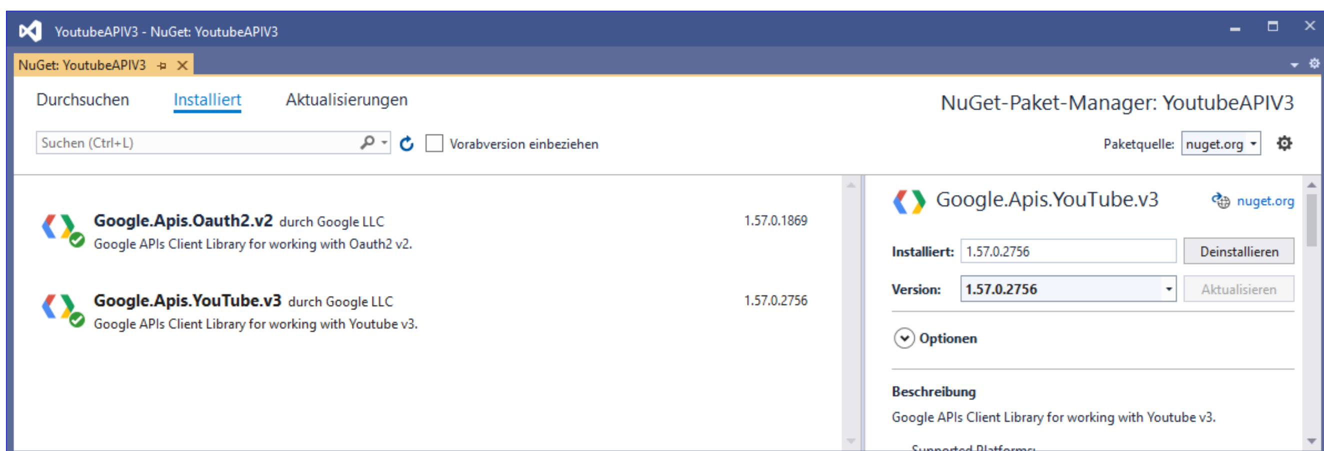

Bild 1: Die für die Lösung benötigten **NuGet**-Pakete

```
Imports System.IO
Imports System.Threading
Imports Google.Apis.Auth.OAuth2
Imports Google.Apis.Services
Imports Google.Apis.YouTube.v3

Public Class Authentication
    Public Shared Function AuthenticateOAuth() As YouTubeService
        Dim objYouTubeService As Google.Apis.YouTube.v3.YouTubeService
        Dim objUserCredential As UserCredential
        Dim objStream As FileStream
        objStream = New FileStream("client_secrets.json", FileMode.Open, FileAccess.Read)
        objUserCredential = GoogleWebAuthorizationBroker.AuthorizeAsync(
            GoogleClientSecrets.FromStream(objStream).Secrets,
            {YouTubeService.Scope.Youtube},
            "user",
            CancellationToken.None).Result
        objYouTubeService = New YouTubeService(New BaseClientService.Initializer() With {
            .HttpClientInitializer = objUserCredential
        })
        Return objYouTubeService
    End Function
End Class
```

Listing 1: Klasse zur Authentifizierung eines Google-Accounts

den Ordner **\bin\Debug\net5.0-windows** unterhalb unseres Projektordners.

Die Funktion **AuthenticateOAuth** liest diese Datei in ein **FileStream**-Objekt ein und übergibt die enthaltenen Daten als ersten Parameter der **AuthorizeAsync**-Methode der Klasse **GoogleWebAuthroizationBroker**. Daneben werden weitere Parameter übergeben. Der zweite lautet **{YouTubeService.Scope.Youtube}** und legt fest, auf welchen Bereich sich die Authentifizierung bezieht. Danach erstellen wir ein neues Objekt des Typs **YouTubeService** und übergeben diesem das **UserCredential**-Objekt als Initialisierer. Danach gibt die Funktion das neue **YouTubeService**-Objekt als Ergebnis zurück.

Alle vorhandenen Videos auflisten

Als ersten Anwendungsfall wollen wir alle Videos eines Kanals ermitteln. Der Seite **MainWindow.xaml** fügen wir dazu eine Schaltfläche wie die folgende hinzu:

```
<Button x:Name="btnAlleVideos" Click="btnAlleVideos_Click">Alle Videos</Button>
```

In der dadurch aufgerufenen Prozedur wollen wir eine Liste von Video-Elementen mit den wichtigsten Eigenschaften eines Videos füllen. Um diese Eigenschaften zu erfassen, wollen wir eine entsprechende Klasse verwenden.

Diese Klasse fügen wir hinzu, indem wir im Projektmappen-Explorer mit der rechten Maustaste auf den Namen des Projekts klicken (in diesem Fall **YouTubeAPIV3**) und dann den Kontextmenüeintrag **Hinzufügen|Klasse...** betätigen. Im folgenden Dialog geben wir einfach den Namen der Klasse an, in diesem Fall **MyVideo.vb**, und klicken dann auf **Hinzufügen**.

Anschließend füllen wir die Klasse mit dem folgenden Code:

```
Public Class MyVideo
    Public Property ID As String
    Public Property Title As String
    Public Property Description As String
    Public Property Thumbnail As String
    Public Property PublishedAt As Date
End Class
```

Wer bisher Klassen nur mit VBA oder VB6 programmiert hat: Unter VB.NET ist die explizite Angabe von **Property Get-/Let-/Set**-Prozeduren nicht notwendig, die hier vorliegende abkürzende Definition reicht für schreib- und lesbare Eigenschaften aus.

Mit der oben definierten Schaltfläche lösen wir eine Prozedur aus, die ein **List**-Objekt zum Aufnehmen der Videos erstellt. Den Typ der aufzunehmenden Elemente geben wir in Klammern an (**Of MyVideo**). Damit rufen wir eine Funktion namens **GetAllVideos** auf. Dieser übergeben wir die ID des Kanals, den wir untersuchen wollen. Wo bekommen wir den her? Dazu rufen wir die folgende Seite im Browser auf:

```
https://www.youtube.com/
account_advanced
```

Die Kanal-ID befindet sich dort unter der Nutzer-ID. (siehe Bild 2) Für den Standardkanal können wir die ID auch aus den Buchstaben **UC** und der Nutzer-ID zusammensetzen.

Nach dem Aufruf der Funktion **GetAllVideos** ist die Liste **objVideos** gefüllt und wir können diese über die Laufvariable **objVideo** in einer **For Each**-Schleife durchlaufen und ihre Eigenschaften ausgeben:

```
Private Sub btnAlleVideos_Click(sender As Object, e As
RoutedEventArgs)
    Dim objVideos As List(Of MyVideo)
    Dim objVideo As MyVideo
    Dim strChannelID As String
    strChannelID = "UCxxxxxxxxxxxxxxxxxxx"
    objVideos = GetAllVideos(strChannelID)
    For Each objVideo In objVideos
        Debug.Print(objVideo.ID)
        Debug.Print(objVideo.Title)
        Debug.Print(objVideo.Description)
        Debug.Print(objVideo.PublishedAt)
        Debug.Print(objVideo.Thumbnail)
    Next
End Sub
```

Das Ergebnis finden wir schließlich im Direktbereich von Visual Studio vor (siehe Bild 3). Wie wir hier sehen, ist die Beschreibung immer an einer bestimmten Stelle abgeschnitten und endet mit drei Punkten (**...**).

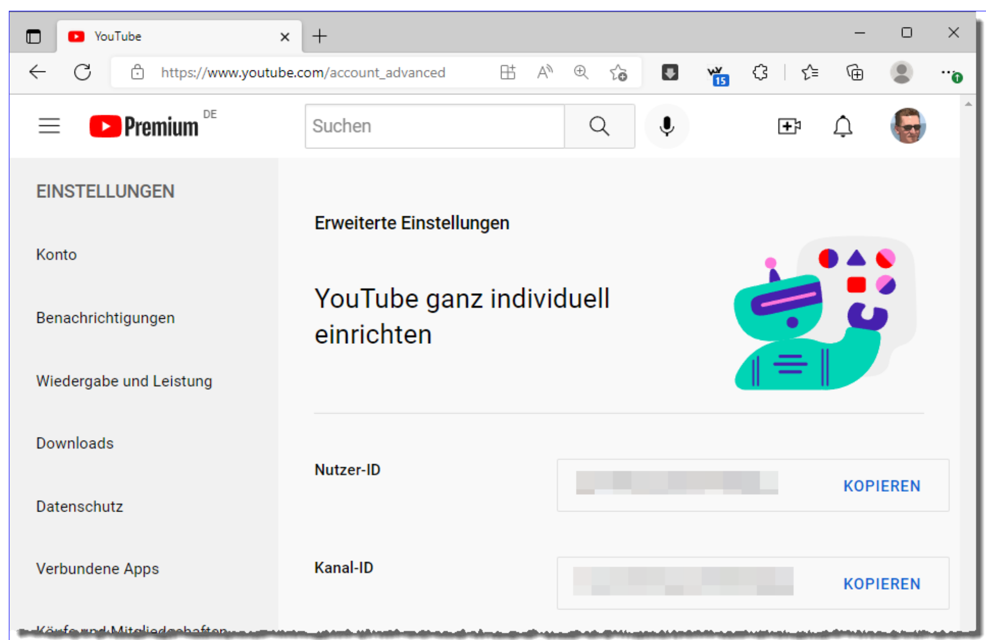

Bild 2: Abfragen der Kanal-ID

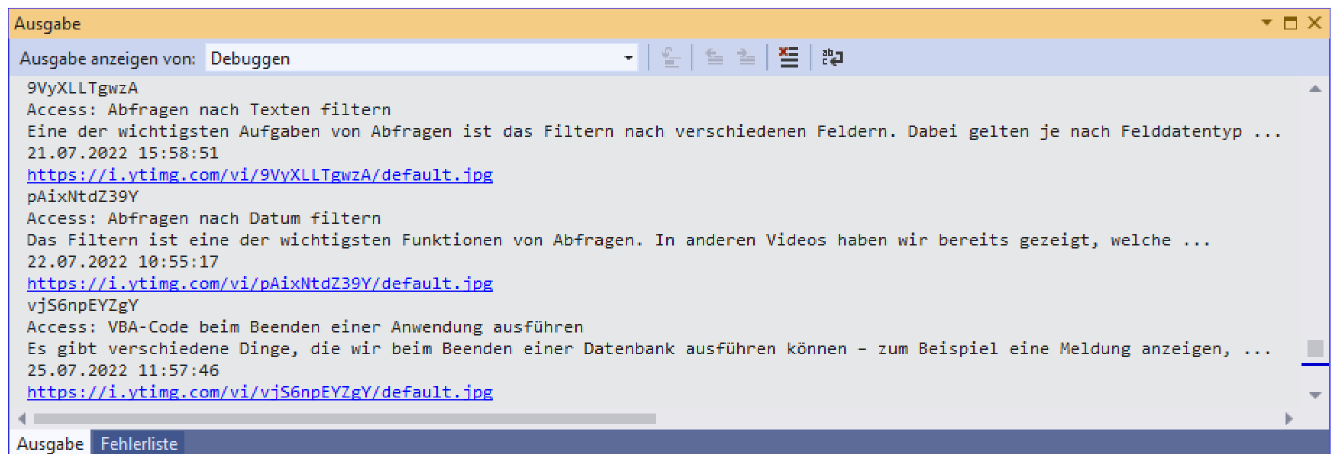

Bild 3: Ausgabe der Video-Eigenschaften im Direktbereich von Visual Studio

Die Auflistung und auch andere API-Funktionen wie etwa die Suchfunktion liefern immer nur die gekürzte Form. Um den vollständigen Beschreibungstext zu ermitteln, benötigen wir den Abruf der Video-Informationen selbst. Wie das gelingt, schauen wir uns weiter unten an – nun wollen wir erst einmal den Abruf der Videoliste genauer betrachten.

Die Funktion GetAllVideos

Die Funktion zum Ermitteln aller Videos eines Kanals erwartet die Angabe der Kanal-ID als Parameter (siehe Listing 2).

Wir deklarieren hier einige Elemente, zum Beispiel eines für den **YouTubeService**, über den der Zugriff auf YouTube abläuft. Außerdem benötigen wir:

- **objVideolistRequest**: Anfrage-Objekt der Anfrage.
- **objVideolistResponse**: Ergebnis-Objekt der Anfrage
- **objSearchResult**: Ein Element der **Items**-Auflistung von **objVideolistResponse**
- **objVideo**: Element der benutzerdefinierten Klasse **MyVideo** zum Aufnehmen der Video-Eigenschaften
- **objVideos**: **List**-Element zum Speichern und Zurückgeben der gefüllten **MyVideo**-Elemente
- **strNextPageToken**: **String**-Variable, die eine Zeichenkette für das Ermitteln der nächsten x Videos enthält

Das **YouTubeServices**-Element füllen wir mit dem Ergebnis der Funktion **AuthenticateOAuth** der oben beschriebenen **Authentication**-Klasse. In dieser Funktion erfolgt beim ersten Zugriff auf die YouTube-API auch die Anzeige des Webbrowsers mit der Aufforderung zur Anmeldung im Kontext des gewünschten Benutzers. Diese Anmeldung ist nach dem Erstellen des Tokens für eine Weile nicht mehr erforderlich. Danach steigen wir direkt in eine **Do While**-Schleife ein, die erst verlassen wird, wenn **strNextPageToken** den Wert **Nothing** enthält.

strNextPageToken ist nötig, weil wir nur maximal 50 Elemente mit einem Aufruf abrufen können (die maximale Anzahl geben wir mit der Eigenschaft **MaxResults** des Request-Objekts an). Für Kanäle mit mehr als 50 Videos rufen wir dann zunächst die ersten 50 Videos ab und erhalten mit der Eigenschaft **NextPageToken** der Antwort ein Token, den wir beim nächsten Aufruf angeben, damit dann die nächsten 50 Ergebnisse geliefert werden und nicht wieder die gleichen.

```
Private Function GetAllVideos(strChannelID As String) As List(Of MyVideo)
    Dim objYouTubeService As YouTubeService
    Dim objVideolistRequest As SearchResource.ListRequest
    Dim objVideolistResponse As SearchListResponse
    Dim objSearchResult As SearchResult
    Dim objVideo As MyVideo
    Dim objVideos As List(Of MyVideo)
    Dim strNextPageToken As String = ""

    objYouTubeService = Authentication.AuthenticateOAuth
    objVideos = New List(Of MyVideo)
    Do While Not strNextPageToken Is Nothing
        objVideolistRequest = objYouTubeService.Search.List("snippet")
        With objVideolistRequest
            .ChannelId = strChannelID
            .Type = "video"
            .MaxResults = 50
            .PageToken = strNextPageToken
        End With
        objVideolistResponse = objVideolistRequest.Execute
        For Each objSearchResult In objVideolistResponse.Items
            objVideo = New MyVideo
            With objVideo
                .ID = objSearchResult.Id.VideoId
                .Title = objSearchResult.Snippet.Title
                .Description = objSearchResult.Snippet.Description
                .Thumbnail = objSearchResult.Snippet.Thumbnails.Default__.Url
                .PublishedAt = objSearchResult.Snippet.PublishedAt
            End With
            objVideos.Add(objVideo)
        Next objSearchResult
        strNextPageToken = objVideolistResponse.NextPageToken
    Loop
    Return objVideos
End Function
```

Listing 2: Prozedur zum Ermitteln einer Liste aller Videos eines Kanals

Beim ersten Durchlauf enthält diese Variable eine leere Zeichenkette (""), was dafür sorgt, dass die ersten Videos der Liste geliefert werden. Doch zunächst weiter im Code: Hier erstellen wir in **objVideolistRequest** das Request-Objekt, indem wir es mit dem Ergebnis der **Search.List("snippet")**-Funktion füllen. Der Parameterwert **snippet** gibt hier an, welche Informationen zurückgeliefert werden sollten. Im Falle von **snippet** sind es die wesentlichen Informationen eines Videos wie Titel, Beschreibung, Veröffentlichungsdatum, Thumbnail et cetera, was in diesem Fall ausreicht – wenn wir beispielsweise die komplette Beschreibung eines Videos ermitteln wollen, benötigen wir ohnehin Zugriff auf das Video-Element über eine andere Anfrage. Und das Ändern von Eigenschaften ist über **Search.List** auch nicht möglich. Für das **objVideolistRequest**-Objekt stellen wir dann einige Eigenschaften ein:

COM-DLL für den Zugriff auf die YouTube-API

In der Artikelreihe »YouTube-Kanal mit VB.NET verwalten« haben wir Methoden vorgestellt, mit denen wir die Videos eines YouTube-Kanals auslesen und mit denen wir neue Videos hochladen oder die Eigenschaften bestehender Videos anpassen können. Diese Methoden wollen wir nun in einer DLL zusammenfassen, die wir in eine Office-Anwendung wie Excel oder Access einbinden können, um von dort aus beispielsweise Titel und Beschreibungstexte lesen und schreiben zu können. Im vorliegenden Artikel zeigen wir daher, wie wir die bereits beschriebenen Funktionen in einem neuen Projekt verwenden und daraus die gewünschte DLL erstellen. In einem weiteren Artikel zeigen wir dann, wie Du von Excel aus die Daten der Videos eines Kanals in eine Excel-Tabelle einlesen kannst.

Projekt für die COM-DLL erstellen

Wenn wir eine COM-DLL erstellen wollen, auf die wir von Office-Anwendungen aus per VBA zugreifen können, wählen wir beim Erstellen eines neuen Projekts in Visual Studio den Typ **Klassenbibliothek (.NET Framework) – Ein Projekt zum Erstellen einer Klassenbibliothek (.dll)** und nicht **Klassenbibliothek – Ein Projekt zum Erstellen einer Klassenbibliothek für .NET Standard oder .NET Core** aus (siehe Bild 1). Mit letzterer können wir aktuell keine COM-DLLs erstellen.

Im nächsten Schritt wählen wir den Speicherort aus und geben als Projektnamen **YouTubeAPIV3** an. Als Zielframework wählen wir **.NET Core 3.1** aus. Das neue Projekt erwartet uns dann in Visual Studio mit einer leeren Klasse namens **Class1.vb**.

NuGet-Pakete hinzufügen

Für den Zugriff auf die YouTube-API v3 benötigen wir zwei NuGet-Pakete. Um diese hinzuzufügen, öffnen wir mit **Projekt|NuGet-Pakete verwalten...** den **NuGet**-Bereich. Im Bereich **Durchsuchen** suchen wir

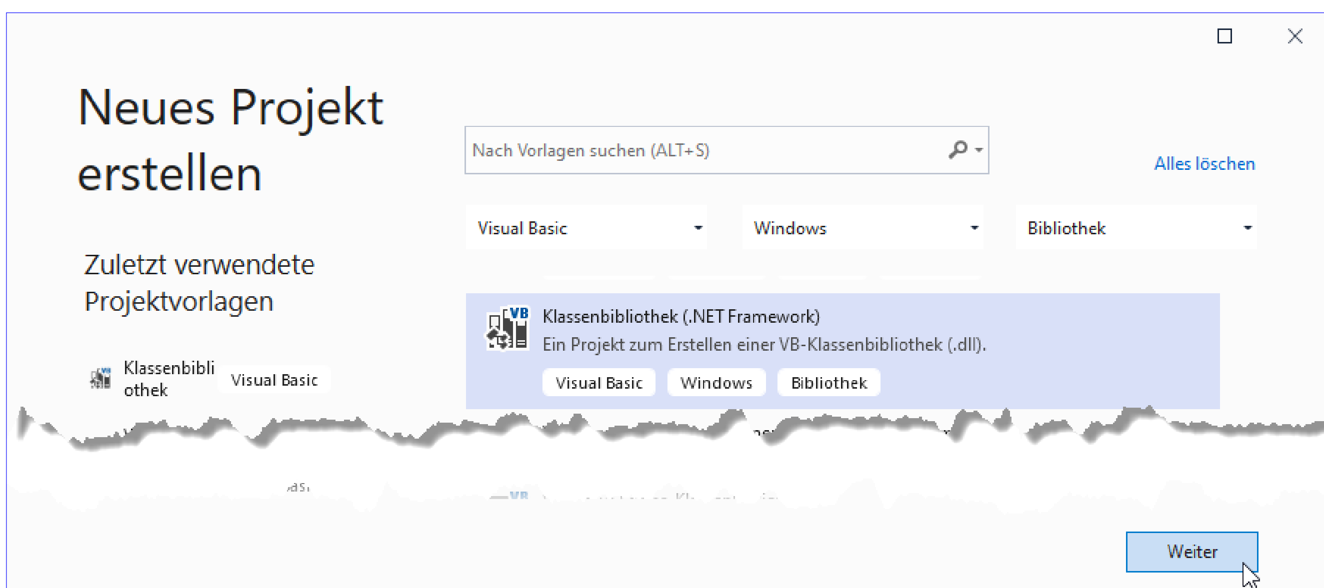

Bild 1: Erstellen eines neuen Projekts

nach den beiden Paketen **Google.Apis.YouTube.v3** und **Google.Apis.Oauth2.v2** und installieren diese.

Vorbereitungen für die COM-DLL

Damit die Klasse später als COM-DLL verwendet werden kann, nehmen wir nun einige Einstellungen in den Projekteigenschaften vor. Dazu öffnen wir die Projekteigenschaften mit einem Rechtsklick auf das Projekt im Projektmappen-Explorer und anschließender Auswahl des Eintrags **Eigenschaften** aus dem nun erscheinenden Kontextmenü.

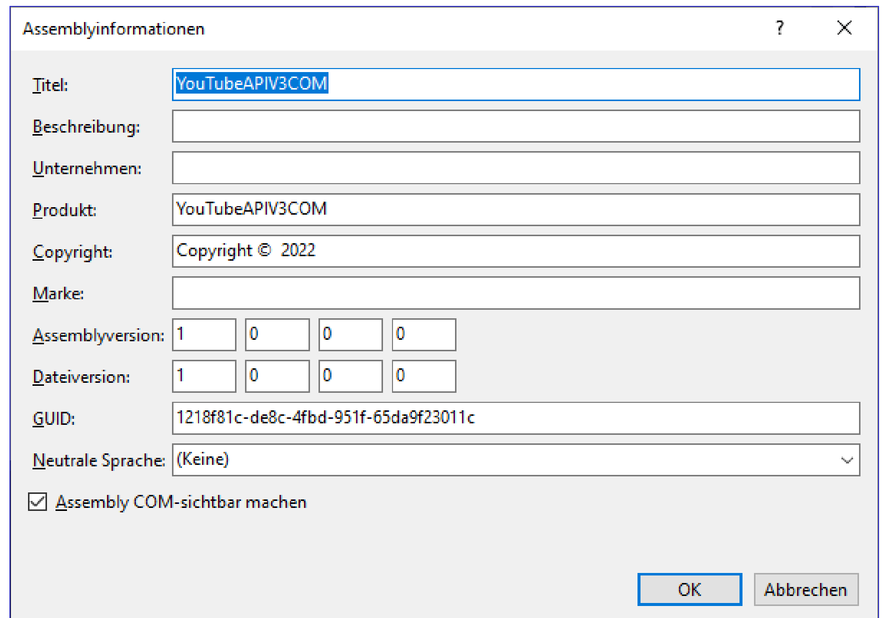

Bild 2: Aktivieren von **Assembly COM-sichtbar machen**

Die erste Einstellung finden wir, wenn wir im Bereich **Anwendung** auf die Schaltfläche **Assemblyinformationen...** klicken. Das öffnet den Dialog **Assemblyinformationen**, wo wir prüfen, ob die Option **Assembly COM-sichtbar machen** aktiviert ist und holen dies gegebenenfalls nach (siehe Bild 2).

Danach klicken wir links auf **Kompilieren**, um den entsprechenden Bereich einzublenden. Hier finden wir unten die

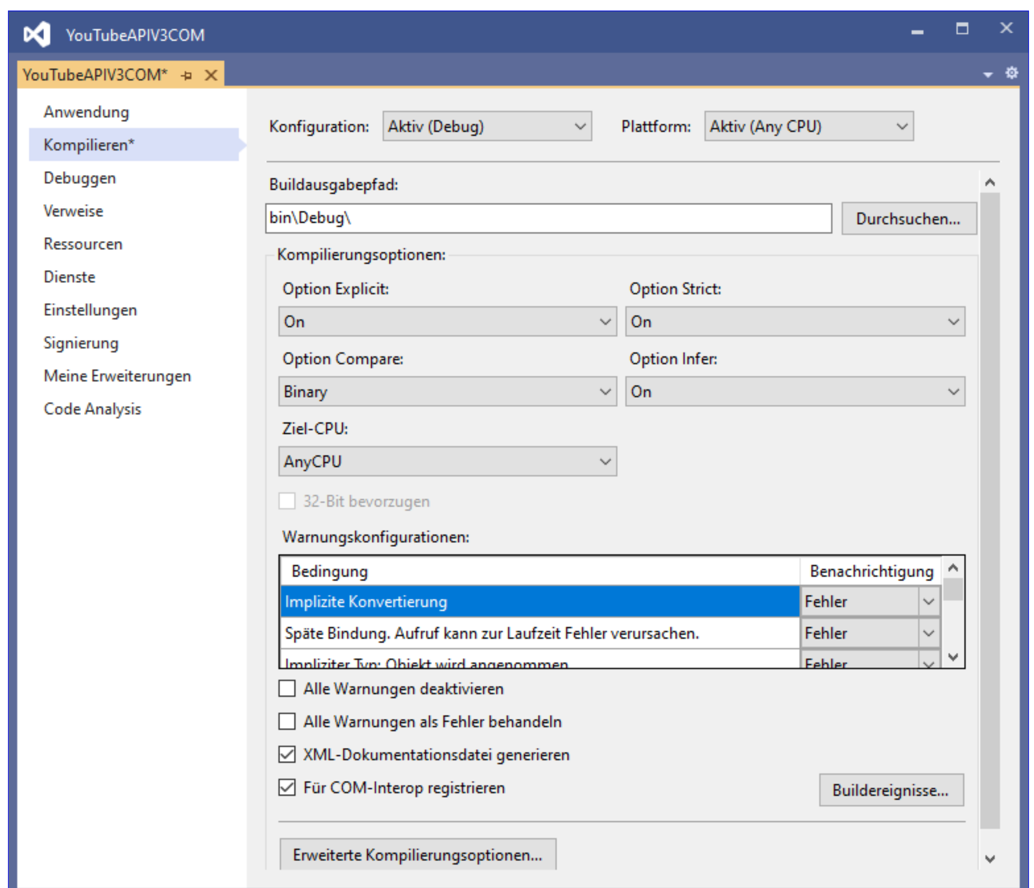

Bild 3: Aktivieren von **Für COM-Interop registrieren**

Eigenschaft **Für COM-Interop registrieren**, die wir aktivieren (siehe Bild 3).

Klasse mit der Funktion zur Authentifizierung

Für den Zugriff auf die YouTube API v3 benötigen wir eine Authentifizierung, die wir mit der im Artikel **YouTube-Kanal mit VB.NET verwalten, Teil 2** (**www.vbentwickler.de/316**) vorgestellten Klasse realisieren. Dazu fügen wir die dort beschriebene Klasse **Authentication.vb** in unser neues Projekt ein. Damit wir diese von einem VBA-Projekt aus nutzen können, sind jedoch einige Anpassungen nötig. Die erste ist eine tatsächliche Funktionserweiterung. Wir möchten das dabei gewonnene Token für den Zugriff auf die YouTube-API nämlich auch separat abrufen können, um gegebenenfalls einmal von einem VBA-Projekt aus direkt auf die YouTube-API zugreifen zu können.

Für den Betrieb als COM-DLL sind überdies noch weitere Änderungen nötig. Als Erstes benötigen wir einen Verweis auf den folgenden Namespace in jeder Klasse, die wir von VBA aus initialisieren und nutzen wollen:

```
Imports System.Runtime.InteropServices
```

Dies ermöglicht es uns, die zu veröffentlichenden Klassen so auszuzeichnen, dass wir von außen auf diese zugreifen können.

Schnittstelle für die Authentifizierungsklasse

Wenn wir eine Klasse einfach nach außen hin veröffentlichen, dann können wir beispielsweise von VBA aus immer noch auf einige .NET-Standard-Eigenschaften und -Methoden der Klasse zugreifen wie **GetHashCode**, **GetType** oder **ToString**. Wir wollen jedoch nur die von uns dafür vorgesehenen Eigenschaften und Methoden anbieten. Dazu müssen wir eine Schnittstelle erstellen, die wir dann in Form der eigentlichen Klasse implementieren.

Unsere Klasse **Authentication.vb** soll zwei Funktionen veröffentlichen: Die erste liefert das **YouTubeService**-Element und die zweite soll den beim Erstellen des **YouTubeService**-Elements generierten Token liefern und heißt **Token**. Die Schnittstelle statten wir daher genau mit der Definition dieser beiden Funktionen aus:

```
<InterfaceType(ComInterfaceType.InterfaceIsDual)>
Public Interface IAuthentication
    Function AuthenticateOAuth(strPathClientSecrets _
        As String) As YouTubeService
    Function Token() As String
End Interface
```

Wie Du siehst, zeichnen wir diese Schnittstelle mit der Annotation **InterFaceType** aus, der wir den Wert **ComInterfaceType.InterfaceIsDual** zuweisen. Was dies bedeutet, schauen wir uns in einem anderen Beitrag im Detail an.

Die Klasse **Authenticate**, die wir in der gleichen Datei wie die Schnittstelle **IAuthenticate** speichern (**Authenticate.vb**), findest Du in Listing 1. Diese Klasse kennzeichnen wir ebenfalls mit der Annotation **ClassInterface**. Für diese legen wir jedoch den Wert **ClassInterfaceType.None** fest. Das sorgt dafür, dass diese Klasse nicht öffentlich ist. Das ist kein Problem, denn ihre Elemente werden ja über die öffentliche Schnittstelle **IAuthenticate** veröffentlicht.

Außerdem müssen wir für diese Klasse den Zusatz **Implements IAuthenticate** hinzufügen, damit diese die Schnittstelle implementiert. Für die einzelnen Elemente der Schnittstelle, die wir implementieren, müssen wir ebenfalls jeweils einen Zusatz zur ersten Zeile hinzufügen. Dieser enthält zusätzlich noch den Namen des Elements, das implementiert wird – für die Funktion **AuthenticateOAuth** lautet der Zusatz beispielsweise **Implements IAuthentication.AuthenticateOAuth**.

```
<ClassInterface(ClassInterfaceType.None)>
Public Class Authentication
    Implements IAuthentication

    Private _Token As String
    Public Function AuthenticateOAuth(strPathClientSecrets As String) As YouTubeService _
            Implements IAuthentication.AuthenticateOAuth
        Dim objYouTubeService As Google.Apis.YouTube.v3.YouTubeService
        Dim objUserCredential As UserCredential
        Dim objStream As FileStream
        objStream = New FileStream(strPathClientSecrets, FileMode.Open, FileAccess.Read)
        objUserCredential = GoogleWebAuthorizationBroker.AuthorizeAsync(
            GoogleClientSecrets.FromStream(objStream).Secrets,
            {YouTubeService.Scope.Youtube},
            "user",
            CancellationToken.None).Result
        objYouTubeService = New YouTubeService(New BaseClientService.Initializer() With {
            .HttpClientInitializer = objUserCredential
        })
        _Token = objUserCredential.Token.AccessToken.ToString
        Return objYouTubeService
    End Function

    Public Function Token() As String Implements IAuthentication.Token
        Return _Token
    End Function
End Class
```

Listing 1: Definition der Klasse **Authenticate.vb**

Im Vergleich zu der Version der Funktion **AuthenticateOAuth**, die wir im Artikel **YouTube-Kanal mit VB.NET verwalten, Teil 2** (**www.vbentwickler.de/316**) vorgestellt haben, haben wir hier noch einen Parameter hinzugefügt. Wir wollen die Handhabung der Datei **client_secrets.json**, welche die Authentifizierungsdaten der App enthält, etwas ändern. Ihr Speicherort soll nicht fest im Code verdrahtet sein, sondern wir wollen diesen flexibel als Parameter übergeben. So können wir diese Datei beispielsweise auch im gleichen Verzeichnis wie die aufrufende Anwendung – beispielsweise eine Access- oder Excel-Anwendung – speichern. Wie Du die Datei **client_secrets.json** erhältst, lernst Du im Artikel **YouTube-Kanal mit VB.NET verwalten, Teil 1** (**www.vbentwickler.de/315**).

Den Pfad zu dieser Datei übergeben wir der Funktion **AuthenticateOAuth** als Parameter und nutzen diesen dann innerhalb der Funktion zum Einlesen der benötigten Daten aus der Datei.

In **AuthenticateOAuth** haben wir außerdem eine Zeile hinzugefügt, welche den Wert der Eigenschaft **AccessToken** aus dem verwendeten **UserCredential**-Objekt in die lokale Variable **_Token** schreibt. Diesen kann die aufrufende Instanz über die Funktion Token abrufen.

Die Klasse YouTubeChannel

Die Klasse mit den Funktionen für den Zugriff auf die Daten eines YouTube-Channels haben wir **YouTubeChannel** genannt. Sie enthält im Wesentlichen

```
Imports System.Runtime.InteropServices

<InterfaceType(ComInterfaceType.InterfaceIsDual)>
Public Interface IYouTubeChannel
    Function GetAllVideos(strChannelID As String) As Collection
    Sub AddPlaylist(strPlaylist As String, strDescription As String)
    Sub UploadVideo(strVideoPath As String, strTitle As String, strDescription As String, _
        Optional strThumbnail As String = "")
    Sub UploadThumbnail(strThumbnail As String, strVideoID As String)
    Function GetVideo(strID As String) As MyVideo
    Sub UpdateVideo(strID As String, Optional strTitle As String = "", Optional strDescription As String = "", _
        Optional strThumbnail As String = "")
    Property PathClientSecrets As String
End Interface
```

Listing 2: Definition der Schnittstelle **IYouTubeChannel**

die Elemente, die wir im Artikel **YouTube-Kanal mit VB.NET verwalten, Teil 2** (**www.vbentwickler.de/316**) erläutert haben. Daher gehen wir hier nicht mehr im Detail auf die enthaltenen Anweisungen ein, sondern nur noch auf die notwendigen Anpassungen für die Verwendung in einer COM-DLL.

Genau wie für die Klasse **Authenticate** erstellen wir auch für die Klasse **YouTubeChannel** eine Schnittstelle. Diese sieht wie in Listing 2 aus. Auch in der Klassendatei **YouTubeChannel.vb** fügen wir oben einen Verweis auf den Namespace **System.Runtime.InteropServices** ein, damit wir die Annotationen für die Veröffentlichungen der Klasse definieren können.

Schlüssel für die Authentifizierung an YouTubeChannel übergeben

In der Implementierung der Schnittstelle **IYouTubeChannel** deklarieren wir zunächst einige Variablen:

```
<ClassInterface(ClassInterfaceType.None)>
Public Class YouTubeChannel
    Implements IYouTubeChannel
    Private m_YouTubeService As YouTubeService
    Private m_Thumbnail As String
    Private _PathClientSecrets As String
    Private _Token As String
```

Eine davon soll den Pfad zu der Datei **client_secrets.json** aufnehmen, die wir weiter oben bereits erwähnt haben. Diese liefert die Schlüssel, die das Holen eines Tokens für den Zugriff der COM-DLL auf die YouTube-API überhaupt erst ermöglicht.

Den Pfad zu dieser Datei wollen wir über eine Eigenschaft namens **PathClientSecrets** der Klasse **YouTubeChannel** übergeben. Diese definieren wir wie folgt:

```
Public Property PathClientSecrets As String _
        Implements IYouTubeChannel.PathClientSecrets
    Get
        Return _PathClientSecrets
    End Get
    Set
        _PathClientSecrets = Value
    End Set
End Property
```

Der übergebene Pfad landet also in der Variablen **_PathClientSecrets** und wird von allen anderen Funktionen und Methoden verwendet. Es ist also wichtig, dass die aufrufende Instanz nach dem Initialisieren von **YouTubeChannel** der Eigenschaft **PathClientSecrets** den Pfad zur Datei **client_secrets.json** übergeben.