AUSGABE 04/2023

VISUAL BASIC ENTWICKLER

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

IN DIESEM HEFT:

André Minhorst Verlag

Installieren geht über Studieren

Wenn man, wie wir es in diesem Magazin tun, öfter mal Tools wie COM-Add-Ins, COM-DLLs oder auch Add-Ins für die verschiedenen Office-Anwendungen wie Access, Excel, Outlook oder Word programmiert, möchte man diese auch zuverlässig auf die Rechner der Benutzer bringen. Dieser sollte dann nicht noch umfangreiche manuelle Schritte zur Installation der jeweiligen Erweiterung durchführen müssen. Also schauen wir uns in dieser Ausgabe ein Tool namens Inno Setup an, das kostenlos und seit Jahren für die Installation von verschiedensten Anwendungen erprobt ist.

Dabei haben wir gleich drei Artikel zu diesem Thema auf Lager. Im ersten Beitrag namens **Installation mit Inno Setup: Die Basics** schauen wir uns ab Seite **30** an, wo wir Inno Setup herunterladen und es installieren und wie wir damit ein erstes Setup erstellen. Mit diesen Setups können wir im einfachsten Fall die Dateien unserer Erweiterung auf dem Rechner des Benutzers installieren, aber auch komplexe Vorgänge durchführen, die Installation mehrerer Dateien, Registry-Einträge und mehr enthalten können.

Nach dem Motto »Das Auge isst mit« schauen wir uns im Artikel **Installation mit Inno Setup: Bilder und Images** ab Seite 41 an, wie wir die standardmäßig von Inno Setup verwendeten Bilder durch unsere eigenen Icons und Bilder ersetzen können. Damit geben wir unseren Setup-Routinen ein individuelles Look and Feel, mit dem wir zum Beispiel das Logo unseres Unternehmens einbauen können.

Sicherheit ist wichtig, und deshalb wollen wir mit unseren Setups keinesfalls den Benutzer verunsichern. Das geschieht aber regelmäßig, wenn wir ausführbare Dateien wie **Setup.exe** aus dem Internet herunterladen und ausführen. Das gelingt bei der einen **.exe**-Datei, bei der anderen aber nicht, weil die Virenscanner solche Dateien auch im Hinblick auf den Publisher untersuchen. Wird dieser nicht durch eine Signatur ausgewiesen, schlagen Virenscanner an. Mit einer validierten Zertifizierung erzeugen wir beim Benutzer daher keine Unsicherheit durch unnötige Hinweise. Mehr dazu im Artikel **Setup signieren mit Inno Setup** ab Seite 45.

Wer viel mit Outlook programmiert, füllt sein VBA-Projekt schnell mit Code. Wenn man dann mal ein neues Tool so testen will, wie es auch auf dem Rechner des Benutzers aussehen würde, ist ein neues, leeres VBA-Projekt praktisch. Wie wir ein solches nutzen können, zeigen wir unter Outlook: Alternatives VBA-Projekt verwenden ab Seite 4.

Ein spannendes Tool stellen wir in den beiden Artikeln **Word: PDF per VBA erzeugen** (ab Seite 7) und **COM-Add-In für Word: PDF-Export** (ab Seite 58) vor. Damit können wir komplette Dokumente, einzelne Seiten, Bereiche und mehr aus einem Word-Dokument komfortabel in ein PDF-Dokument exportieren.

Unter **Office-VBA per COM-DLL mit VB.NET** erweitern zeigen wir ab Seite 15, wie Du die Möglichkeiten von VB.NET in Deine Office-Anwendung bringen kannst. Dazu brauchst Du nur eine COM-DLL zu programmieren, deren öffentlichen Methoden, Eigenschaften und Co. dann im Office-VBA-Projekt per Verweis verfügbar gemacht werden können.

Wenn wir schon COM-DLLs oder COM-Add-Ins unter VB.NET erstellen, wollen wir diese auch ordentlich debuggen. Wie das funktioniert, zeigen wir im Artikel **COM-DLLs und Add-Ins unter VB.NET debuggen** ab Seite 23.

Nun viel Spaß beim Lesen!

Dein André Minhorst

Outlook: Alternatives VBA-Projekt verwenden

Wenn Du eine neue VBA-Anwendung für den Einsatz in Outlook programmierst, möchtest Du vielleicht in einem frischen, leeren VBA-Projekt arbeiten. Bei mir ist das standardmäßig verwendete VBA-Projekt in Outlook recht voll mit allen möglichen produktiven und zum Testen verwendeten Ereignisprozeduren und weiteren Routinen. Eine neue Funktion so zu testen, als wenn diese auf dem Rechner eines Nutzers mit einem leeren Outlook-VBA-Projekt landet, ist so kaum möglich. Es gibt jedoch gute Nachrichten: Wir können recht fix ein neues VBA-Projekt für Outlook erstellen, das wir alternativ zu dem produktiv genutzten VBA-Projekt nutzen. Noch besser: Es gibt die Möglichkeit, beim Start anzugeben, welches VBA-Projekt wir aktuell nutzen möchten.

Frische und umfangreiche VBA-Projekte

Das VBA-Projekt von Outlook öffnen wir mit der Tastenkombination **Alt + F11**. Wenn Du noch keine Änderungen an diesem VBA-Projekt vorgenommen hast, sieht dieses wie in Bild 1 aus. Das ist recht aufgeräumt – es gibt lediglich das Klassenmodul ThisOutlookSession.

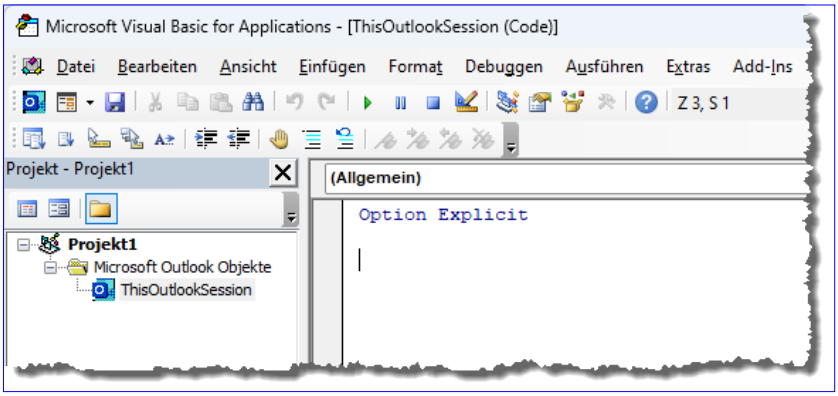

Bild 1: Ein unbeflecktes VBA-Projekt unter Outlook

Mein VBA-Projekt sieht etwas anders aus – allein der Projektexplorer ist bereits recht voll (siehe Bild 2). Hier befinden sich einige tausend Zeilen Code, die teilweise produktiv im Einsatz sind und teilweise Überreste von Experimenten und Beispielen für Artikel sind.

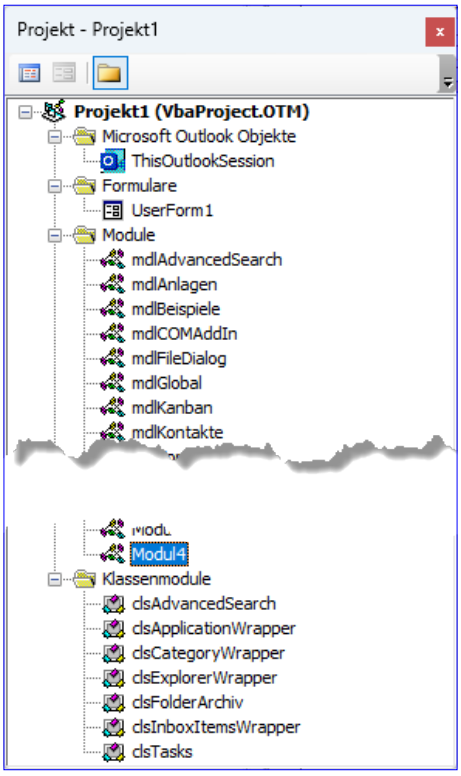

Bild 2: Das VBA-Projekt des Autors

VBA-Projekt »zurücksetzen«

Wenn Du bei Dir eine ähnliche Situation im VBA-Projekt von Outlook vorfindest wie ich beim mir, wünscht Du Dir vielleicht zu manchen Gelegenheiten, einmal ein leeres VBA-Projekt nutzen zu können. Das ist in wenigen Schritten erledigt:

- Wir benennen das bestehende VBA-Projekt um.
- Wir öffnen Outlook, welches das VBA-Projekt nicht mehr findet und dieses neu erstellt.

Fertig! Nun können wir mit einen neuen Outlook-VBA-Projekt arbeiten. Nun wollen wir dieses allerdings nur gelegentlich zum Experimentieren nutzen und standardmäßig unser

bewährtes VBA-Projekt nutzen. Dazu sind ein paar weitere Schritte erforderlich, die wir weiter unten beschreiben. Zunächst einmal schauen wir uns im Detail an, wie wir ein neues VBA-Projekt erstellen können.

Neues VBA-Projekt erstellen

Die ausführliche Version sieht vor, dass wir erst einmal Outlook schließen. Dann öffnen wir den Windows Explorer und navigieren zu dem Ordner, in dem sich die Datei mit dem VBA-Projekt von Outlook befindet. Das Verzeichnis lautet üblicherweise:

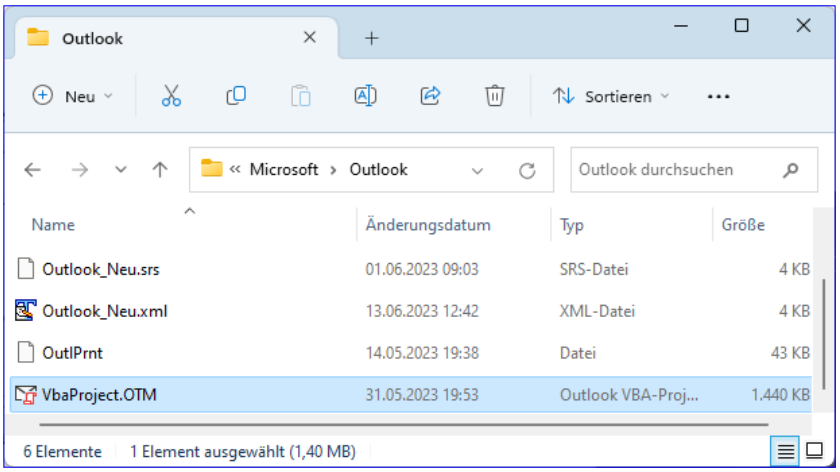

Bild 3: Das VBA-Projekt im Windows Explorer

```
C:\Users\[Benutzername]\AppData\Roaming\Microsoft\Outlook
```

Hier finden wir die Datei **VbaProject.OTM** vor (siehe Bild 3).

Diese Datei benennen wir nun beispielsweise in **VbaProject_Produktiv.OTM** um.

Anschließend öffnen wir Outlook erneut. Dabei können wir den Windows Explorer einmal im Auge behalten. Es geschieht noch nichts, solange wir nur Outlook und anschließend den VBA-Editor öffnen. Erst wenn wir das VBA-Projekt speichern, legt Outlook ein neues Projekt namens **VbaProject.OTM** an.

Zurück zum produktiven VBA-Projekt

Nun wollen wir dieses neue VBA-Projekt allerdings nur nutzen, um neue Routinen zu testen oder COM-DLLs aufzurufen. Deshalb soll beim nächsten Start von Outlook wieder das produktive VBA-Projekt verwendet werden. Das neue, leere VBA-Projekt wollen wir nur bei Bedarf nutzen.

Der erste Schritt ist, dass wir Outlook wieder schließen. Dann benennen wir zuerst das neue, leere VBA-Projekt um, beispielsweise in **VbaProject_Leer.OTM**. Das produktive VBA-Projekt erhält dann wieder den Namen **VbaProject.OTM**.

Beim nächsten Start verwendet Outlook wieder das produktive VBA-Projekt und löst gegebenenfalls auch vorhandene Ereignisprozeduren aus.

Alternatives VBA-Projekt nutzen

Bleibt noch der letzte Schritt: Der Aufruf von Outlook mit einem alternativen VBA-Projekt, ohne dass wir das standardmäßig verwendete VBA-Projekt umbenennen müssen.

Das erledigen wir mit einer Befehlszeilenoption, also einer Option, die beispielsweise in einer Dateiverknüpfung auf Outlook angeben. Diese Option heißt **/altvba**.

Wenn wir eine herkömmliche Verknüpfung nutzen wollen, erstellen wir diese mit einem Rechtsklick in das gewünschte Zielverzeichnis im Windows Explorer und anschließender Auswahl des Befehls **Neu|Verknüpfung**.

Diese Verknüpfung gestalten wir wie folgt. Als Ziel geben wir beispielsweise folgenden Ausdruck an:

Word: PDF per VBA erzeugen

Wenn man ein Dokument per VBA zusammengestellt hat, möchte man dieses gegebenenfalls auch noch weiterverarbeiten. Zum Beispiel, indem man es unter einen bestimmten Dateinamen speichert. Eine weitere Anforderung könnte sein, dass man das Dokument direkt als PDF-Dokument sichern möchte. Wie das gelingt, schauen wir uns in diesem Artikel an. Und wir gehen noch einen Schritt weiter und betrachten, wie wir bestimmte Bereiche oder auch jede einzelne Seite in einem PDF-Dokument speichern können.

In vielen Fällen sollen Word-Dokumente nicht im Format **.docx** weitergegeben werden. Einer der Gründe ist, dass wir nicht wollen, dass der Empfänger das Dokument bearbeiten oder die Inhalte auf einfache Weise extrahieren kann. Eine gute Lösung ist dann die Erstellung eines PDF-Dokuments. Dieses kann man auf allen Systemen mit entsprechenden Readern anzeigen und auch im Webbrowser erscheinen diese ohne Probleme.

Während man meist komplette Dokumente in eine PDF-Datei exportieren möchte, kommt es auch vor, dass ein Word-Dokument bereits mehrere Teildokumente enthält, die einzelnen exportiert werden sollen. Und ein weiterer Fall ist, dass wir nur bestimmte Seiten aus einem Dokument extrahieren wollen, andere jedoch nicht.

Dokument als .dotm speichern

Um einem eventuellen Verlust des nachfolgend erstellten VBA-Codes vorzubeugen, haben wir das Dokument direkt als **.dotm**-Dokument gespeichert. Die Prozeduren haben wir in das Modul **ThisDocument** eingefügt.

Komplettes Dokument als PDF speichern

Wir beginnen mit dem einfachsten Fall: Wir wollen alle Seiten des Dokuments in einer PDF-Datei speichern. Diesen Fall haben wir in der Prozedur **PDFSpeichern** aus Listing 1 abgedeckt. Hier deklarieren wir eine Variable des Typs **Word.Document**, um das zu extrahierende Dokument zu referenzieren. Das erledigen wir anschließend über die Eigenschaft **ActiveDocument**, die einen Verweis auf das aktuell angezeigte Dokument liefert.

Schließlich wollen wir noch den Namen der zu erstellenden Datei definieren. Die neue Datei soll im gleichen Verzeichnis wie das Original landen. Dieses ermitteln wir mit **doc.Path**. Fügen wir noch **\Export.pdf** an, erhalten wir den vollständigen Pfad.

Die eigentliche Arbeit übernimmt die Methode **ExportAsFixedFormat** des **Document**-Objekts. Dieser

```
Public Sub PDFSpeichern()
    Dim doc As Word.Document
    Dim strDateipfad As String
    Set doc = ActiveDocument
    strDateipfad = doc.Path & "\Export.pdf"
    doc.ExportAsFixedFormat OutputFileName:=strDateipfad, ExportFormat:=wdExportFormatPDF
    Set doc = Nothing
End Sub
```

Listing 1: Speichern des kompletten Dokuments

```
Public Sub PDFSpeichernGleicherName()
    Dim doc As Word.Document
    Dim strDateipfad As String
    Dim strDateiname As String
    Set doc = ActiveDocument
    strDateiname = Replace(doc.Name, ".docx", ".pdf")
    strDateiname = Replace(strDateiname, ".docm", ".pdf")
    strDateipfad = doc.Path & "\" & strDateiname
    doc.ExportAsFixedFormat OutputFileName:=strDateipfad, ExportFormat:=wdExportFormatPDF
    Set doc = Nothing
End Sub
```

Listing 2: Speichern des kompletten Dokuments mit dem gleichen Namen und der Endung .pdf

übergeben wir für den Parameter **OutputFileName** den Pfad aus der Variablen **strDateipfad** und als **ExportFormat** den Wert **wdExportFormatPDF**.

Das Ausführen der Prozedur sorgt für das Anlegen der Datei im gleichen Verzeichnis wie die Originaldatei.

PDF unter gleichem Namen speichern

Wir können noch ein wenig verfeinern, wenn wir wollen, dass das Dokument unter dem gleichen Namen wie das Original gespeichert wird – nur mit der Dateiendung **.pdf** statt**.docx**.

Dazu ermitteln wir mit **doc.Name** den Namen des Dokuments und ersetzen mit der **Replace**-Methode die Dateiendung **.docx** durch **.pdf**.

Damit wir den Code überhaupt ausführen können, verwenden wir eine Word-Datei mit der Dateiendung **.docm**. Also ersetzen wir auch diese Dateiendung noch durch **.pdf** (siehe Listing 2).

Die ExportAsFixedFormat-Methode

Bevor wir weitere Anwendungsfälle wie das Extrahieren einzelner Seiten oder eines bestimmten Bereichs betrachten, wollen wir einen Blick auf die Parameter der Methode **ExportAsFixedFormat** werfen.

Diese lauten:

- **OutputFileName** (**String**): Erforderlich. Der Pfad und der Dateiname für die PDF-Datei.
- **ExportFormat** (**WdExportFormat**): Erforderlich. Das Format, in das exportiert werden soll (**wdExportFormatPDF** oder **wdExportFormatXPS**).
- **OpenAfterExport** (**Boolean**): Optional. Standardmäßig auf **False** gesetzt. Gibt an, ob die PDF-Datei nach dem Export geöffnet werden soll.
- **OptimizeFor** (**wdExportOptimizeFor**): Optional. Gibt an, ob für die Darstellung auf dem Bildschirm oder für den Druck optimiert werden soll (**wdOptimizeForOnScreen** oder **wdOptimizeForPrint**).
- **Range** (**wdExportRange**): Optional. Legt fest, welcher Bereich exportiert werden soll. Werte: **wdExportAllDocument** – Exportiert das vollständige Dokument, **wdExportCurrentPage** – Exportiert die aktuelle Seite, **wdExportFromTo** – Exportiert einen Seitenbereich, **wdExportSelection** – Exportiert die aktuelle Markierung.
- **From** (**Long**): Optional. Gibt die erste Seite des zu exportierenden Bereichs an.
- **To** (**Long**): Optional. Gibt die letzte Seite des zu exportierenden Bereichs an.

- **Item**: Optional. Das zu exportierende Element (zum Beispiel **wdExportDocumentContent** oder **wdExportDocumentWithMarkup**).
- **IncludeDocProps** (**Boolean**): Optional. Gibt an, ob Dokumenteigenschaften in die PDF-Datei eingefügt werden sollen.
- **KeepIRM** (**Boolean**): Optional. Gibt an, ob die Information Rights Management (IRM)-Berechtigungen beibehalten werden sollen.
- **CreateBookmarks** (**WdExportCreateBookmarks**): Optional. Gibt an, ob Lesezeichen erstellt werden sollen. Die Werte für diesen Parameter lauten: **wdExportCreateHeadingBookmarks**, **wdExportCreateNoBookmarks**, **wdExportCreateWordBookmarks**
- **DocStructureTags** (**Boolean**): Optional. Gibt an, ob Strukturinformationen in das PDF eingefügt werden sollen.
- **BitmapMissingFonts** (**Boolean**): Optional. Gibt an, ob fehlende Schriftarten als Bitmaps in das PDF eingebettet werden sollen.
- **UseISO19005_1** (**Boolean**): Optional. Gibt an, ob das PDF im ISO-19005-1-kompatiblen (PDF/A) Format erstellt werden soll.

Damit erhalten wir bereits eine ganze Menge Möglichkeiten für den Export von Word-Dokumenten. Für uns sind in den nächsten Schritten vor allem die beiden Parameter **StartPage** und **EndPage** interessant.

Jede Seiten als eigenes PDF-Dokument extrahieren

Die erste Lösung soll jede Seite einzeln extrahieren. Dazu müssen wir die Anzahl der Seiten ermitteln und danach jede Seite einzeln durchlaufen und exportieren.

Das erledigen wir in der Prozedur aus Listing 3. Hier ermitteln wir die Seitenzahl mit **doc.ComputeStatistics(wdStatisticsPages)**. Außerdem legen wir den Ba-

```
Public Sub PDFSpeichernEinzelneSeite()
    Dim doc As Document
    Dim lngSeiten As Long
    Dim lngSeite As Integer
    Dim strDateipfad As String
    Dim strDateiname As String
    Set doc = ActiveDocument
    lngSeiten = doc.ComputeStatistics(wdStatisticPages)
    strDateiname = Replace(doc.Name, ".docx", ".pdf")
    strDateiname = Replace(strDateiname, ".docm", ".pdf")
    For lngSeite = 1 To lngSeiten
        strDateipfad = doc.Path & "\" & Replace(strDateiname, ".pdf", "_" & Format(lngSeite, "00") & ".pdf")
        doc.ExportAsFixedFormat OutputFileName:=strDateipfad, _
            ExportFormat:=wdExportFormatPDF, _
            Range:=wdExportFromTo, _
            From:=lngSeite, _
            To:=lngSeite
        Next lngSeite
End Sub
```

Listing 3: Speichern aller Seiten des Dokuments als einzelne PDF-Dateien

Office-VBA per COM-DLL mit VB.NET erweitern

VBA ist, formulieren wir es einmal freundlich, seit einiger Zeit nicht mehr aktualisiert worden. Hier und da gibt es kleine Anpassungen in den Objektbibliotheken, aber der Sprachumfang an sich hat keine großen Schritte gemacht. Auch VB.NET bringt nicht täglich neue Sprachkonstrukte hervor. Aber dafür gibt es beispielsweise zahllose Erweiterungen in Form von NuGet-Paketen, die man leicht in einem VB.NET-Projekt verfügbar machen kann. Aus VB.NET-Projekten kann man aber auch eine COM-DLL erzeugen, die wir wiederum von einem VBA-Projekt aus referenzieren und nutzen können. Und somit können wir auch den Funktionsumfang von Word, Excel, Outlook, Access und Co. erweitern. Dieser Artikel zeigt die Grundlagen zur Erstellung eines COM-Add-Ins, das wir von unseren Office-Anwendungen aus nutzen können.

Voraussetzung: Visual Studio .NET

Die einzige Voraussetzung, die wir neben der zu erweiternden Office-Anwendung brauchen, ist eine Version von Visual Studio .NET. Microsoft bietet eine für viele Fälle kostenlos nutzbare Version an, die Du beispielsweise hier findest:

```
https://visualstudio.microsoft.com/de/vs/community/
```

Visual Studio als Administrator öffnen

Wenn wir die geplante DLL erstellen und testen wollen, benötigen wir in der Regel Administrator-Rechte. Dazu öffnen wir Visual Studio direkt als Administrator. Dazu klicken wir mit der rechten Maustaste auf das Visual Studio-Icon und wählen dort den Eintrag **Als Administrator ausführen** aus. Die anschließende Meldung bestätigen wir.

DLL-Projekt erstellen

Anschließend erstellen wir ein DLL-Projekt. Aus dem Startbildschirm von Visual Studio wählen wir dazu den Eintrag **Neues Projekt erstellen** aus. Im Dialog Neues Projekt erstellen finden wir einige Auswahlfelder, mit denen wir die Auswahl aus Bild 1 treffen (**Visual Basic**, **Windows**, **Bibliothek**) und dann den Eintrag **Klassenbibliothek (.NET Framework)** selektieren.

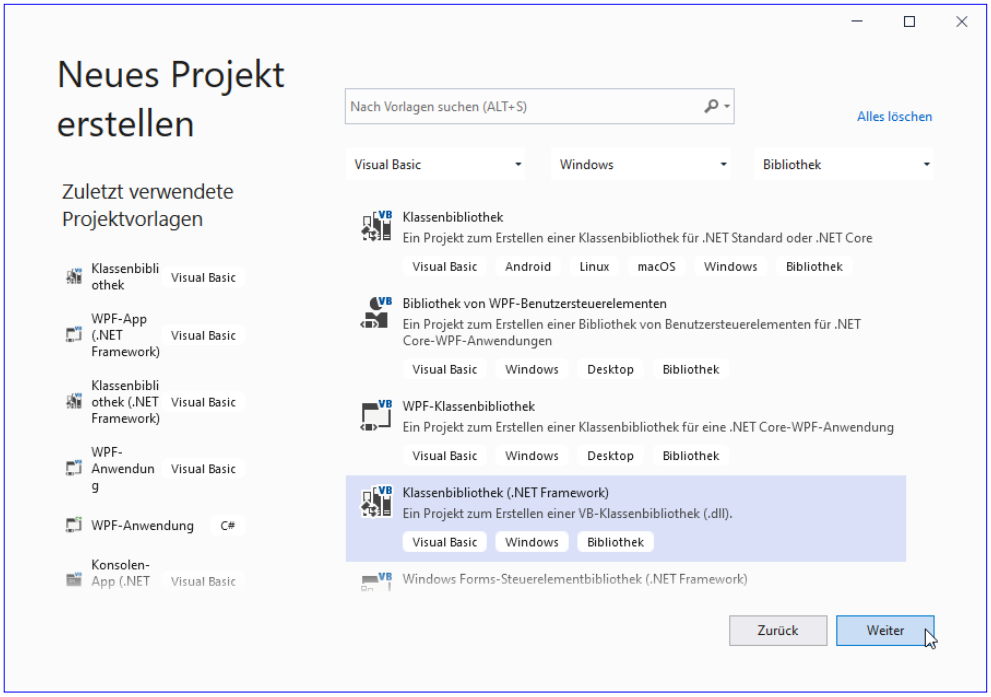

Bild 1: Auswahl des Typs für das neue Projekt

Im nächsten Dialog namens Neues Projekt konfigurieren geben wir einen Projektnamen ein, in diesem Fall COMDLL-Beispiel.

Außerdem wählen wir den Speicherort für das Projekt aus und legen als **Framework** den Eintrag **.NET Framework 4.7.2** fest (siehe Bild 2).

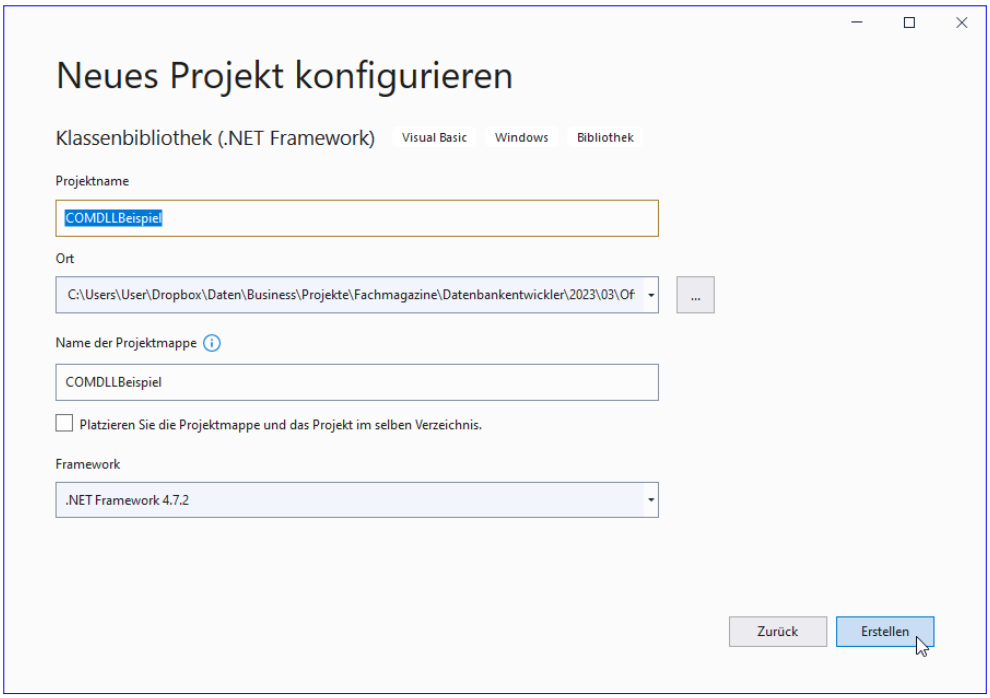

Bild 2: Eingabe von Projektname und Speicherort

Visual Studio nimmt sich nun einige Augenblicke Zeit, um das Projekt anzulegen und dieses anzuzeigen. Das Ergebnis sehen wir in Bild 3. Wir finden im Projektmappen-Explorer eine neue Projektmappe mit einer Klasse namens **Class1.vb** vor. Diese wollen wir als Erstes umbenennen.

Umbenennen der Klasse

Das ist an sich kein Hexenwerk, wir wollen nur kurz klären, wo wir den Namen dieser Klasse sehen werden. Wenn wir die COM-DLL später im VBA-Editor einer

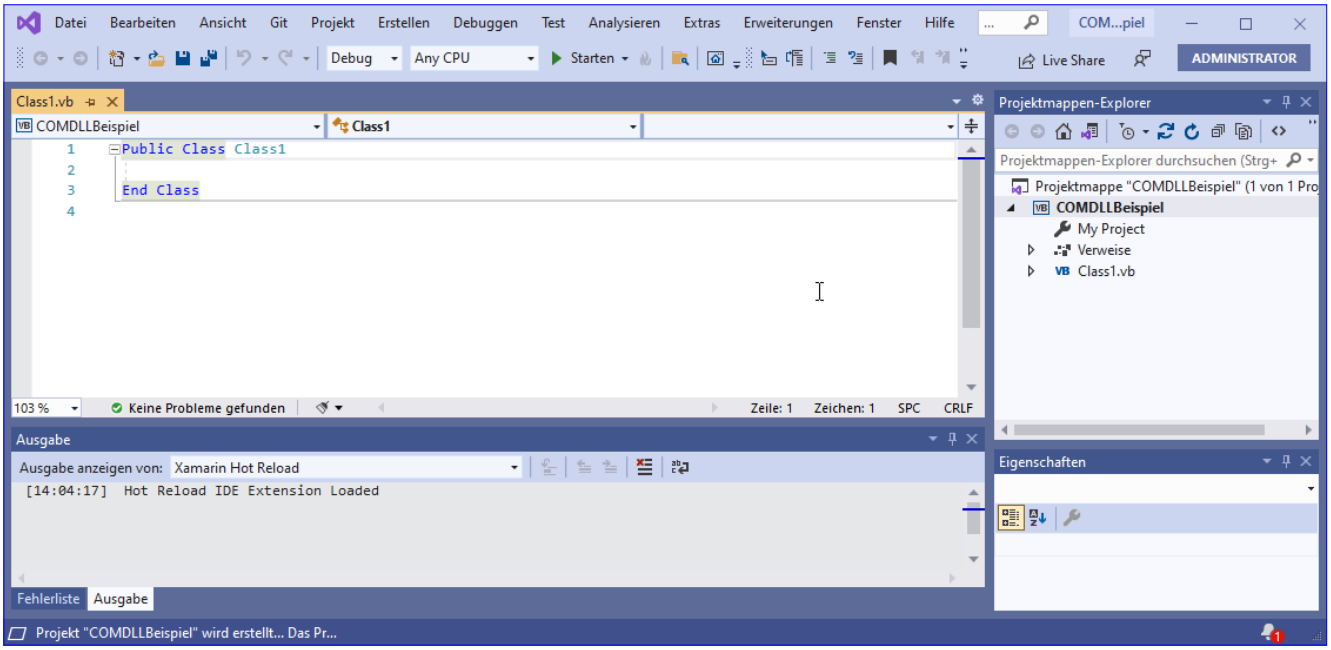

Bild 3: Das neue Projekt in Visual Studio

Office-Anwendung deklarieren und referenzieren, geschieht das in der folgenden Form:

```
Dim obj As [Assemblyname].[Klassenname]
```

Neben dem Assemblynamen sollten wir also auch den Klassennamen so auswählen, dass wir nachher auch erkennen, was diese Klasse für Funktionen liefert.

In unserem Beispiel spielt das allerdings keine große Rolle und wir verwenden einfach **Beispielklasse**. Diesen Wert tragen wir im Projektmappen-Explorer **Beispielklasse.vb** statt **Class1.vb** ein.

Visual Studio fragt nach, ob wir auch die Verweise auf diese Elemente umbenennen möchten, was wir mit **Ja** beantworten (siehe Bild 4). Dadurch wird auch direkt der Klassenname im Code geändert:

```
Public Class Beispielklasse

End Class
```

Vorbereitungen in den Projekteigenschaften

Damit kommen wir zu den anzupassenden Einstellungen in den Projekteigenschaften. Diese öffnen wir mit einem Doppelklick auf den Eintrag **My Project** im Projektmappen-Explorer.

Hier können wir nun, falls gewünscht, noch den Namen der Assembly anpassen – dieser Name erscheint später im **Verweise**-Dialog des VBA-Editors (siehe Bild 5). Wichtiger ist jedoch ein Klick auf die Schaltfläche **Assemblyinformationen...**, mit dem wir einen weiteren Dialog öffnen.

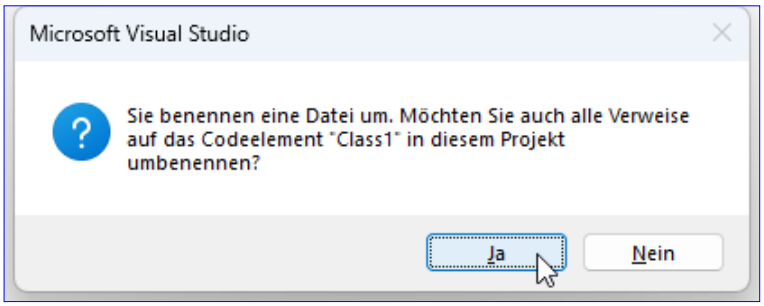

Bild 4: Umbenennen von Datei und Klasse

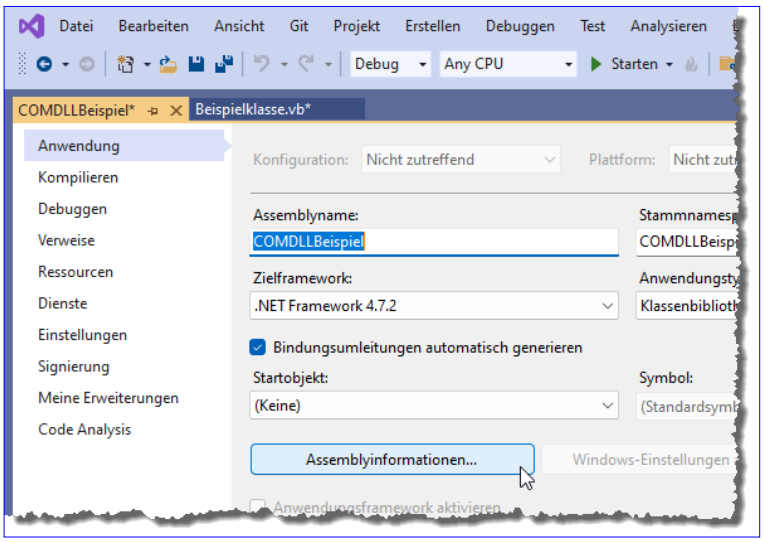

Bild 5: Einstellungen in den Projekteigenschaften

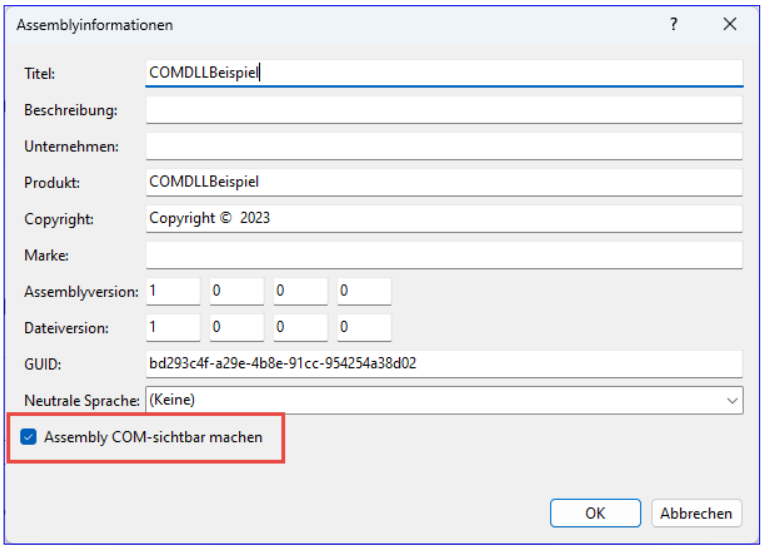

Bild 6: Die Assemblyinformationen

COM-DLLs und Add-Ins unter VB.NET debuggen

Im Artikel »Office-VBA per COM-DLL mit VB.NET erweitern« und in einigen weiteren Artikel haben wir bereits gezeigt, wie man Visual Studio zum Erstellen von COM-DLLs und COM-Add-Ins nutzen kann. Im vorliegenden Artikel ergänzen wir die dort vorstellten Techniken zum Erstellen von COM-DLLs und COM-Add-Ins noch um die Vorgehensweise zum Debuggen des Codes dieser Projekte. Während WPF-Projekte oder Konsolenanwendungen sich von Visual Studio .NET starten und auch debuggen lassen, gelingt das bei COM-DLLs und COM-Add-Ins nicht so leicht. Es handelt sich dabei nicht um ausführbare Dateien, also müssen wir einen kleinen Umweg gehen – und diesen beschreiben wir im vorliegenden Artikel.

Wir beziehen uns in diesem Artikel zunächst auf die Beispiellösung aus dem Artikel **Office-VBA per COM-DLL mit VB.NET erweitern** (**www.vbentwickler.de/378**). Später schauen wir uns noch an, wie das Debugging mit einem COM-Add-In aussieht.

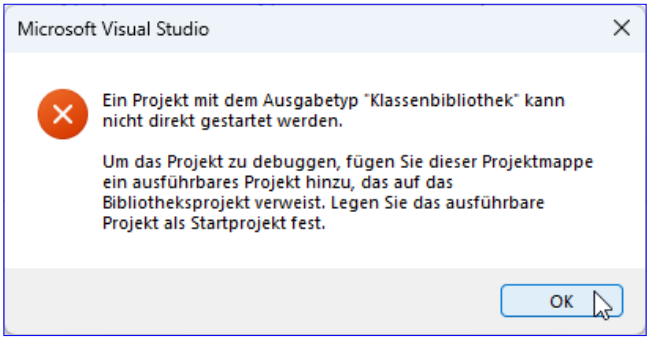

Bild 1: Fehlermeldung beim Versuch, ein nicht ausführbares Projekt zu starten

Wenn wir eine COM-DLL wie die aus dem genannten Artikel in Visual Studio .NET öffnen und sie starten wollen, indem wir auf die **Starten**-Schaltfläche klicken oder die Taste **F5** betätigen, erhalten wir die Meldung aus Bild 1. Wir müssen also einen anderen Weg finden, um unseren Code zu debuggen.

Die Debuggen-Eigenschaften

Auch wenn die Anzahl der Programmierer, die COM-DLLs und COM-Add-Ins vergleichsweise überschaubar sein dürfte, so bietet Visual Studio auch für diese eine passende Lösung an. Diese finden wir in den Projekteigenschaften, die wir mit einem Doppelklick auf

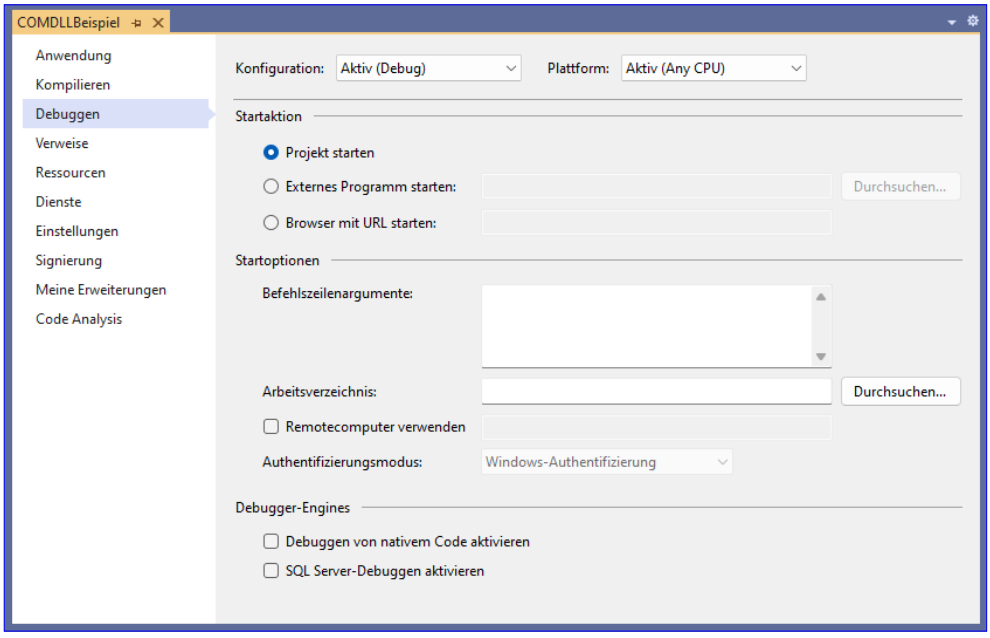

Bild 2: Der Bereich **Debuggen** in den Projekteigenschaften

den Eintrag **My Project** im Projektmappen-Explorer öffnen. Dort wechseln wir anschließend zum Bereich **Debuggen**, der zunächst wie in Bild 2 aussieht.

Konfiguration auswählen

Hier sehen wir verschiedene Optionen. Sollte die Eigenschaft **Konfiguration** einen anderen Wert als **Aktiv (Debug)** aufweisen, stellen wir diese nun auf diesen Wert ein.

Zu startende Anwendung festlegen

Im Bereich **Startaktion** geben wir an, welche Aktion durchgeführt werden soll. Für uns sind die folgenden beiden interessant:

- **Projekt starten**: Mit dieser Option wird das aktuelle Projekt gestartet, was nur bei ausführbaren Projekten sinnvoll ist.

- **Externes Programm starten**: Wenn das Projekt im Kontext einer anderen Anwendung verwendet werden soll, was bei COM-DLLs und COM-Add-Ins der Fall ist, geben wir den Pfad zu dieser Anwendung an.

Wir können zum Beispiel Access, Excel, Outlook oder Word als externes Programm angeben. Dazu benötigen wir den Pfad zu dieser Anwendung, der sich je nach verwendeter Version unterscheiden kann.

Die einfachste Methode, den Pfad zur ausführbaren Datei für eine der Office-Anwendungen zu finden, ist ein Rechtsklick auf das Icon in der Taskleiste oder auf den Namen im Suchen-Bereich von Windows und das anschließende Auswählen des Befehls **Eigenschaften** (siehe Bild 3).

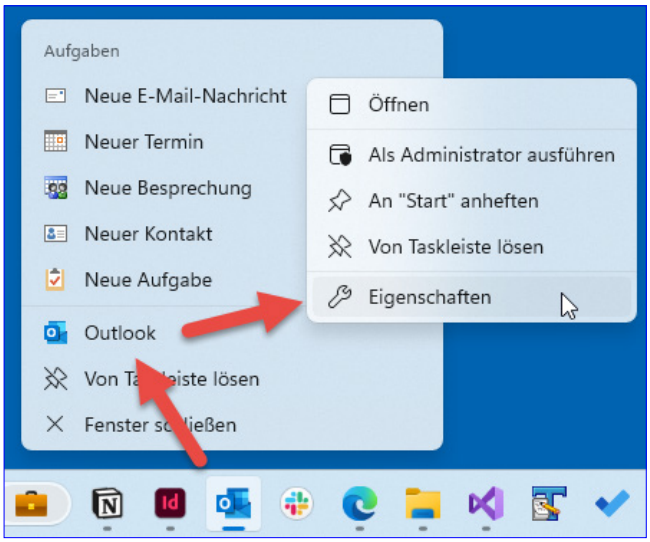

Bild 3: Anzeigen der Eigenschaften einer Office-Anwendung

Anschließend können wir dem **Eigenschaften**-Dialog aus dem Textfeld **Ziel** den Pfad zur Anwendung entnehmen (siehe Bild 4).

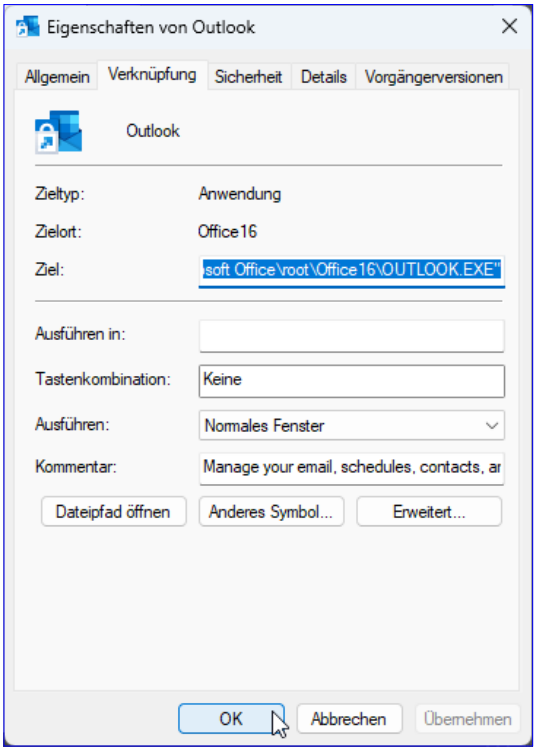

Bild 4: Eigenschaften einer Office-Anwendung

Für Outlook lautet dieser beispielsweise:

```
C:\Program Files (x86)\Microsoft
Office\root\Office16\
OUTLOOK.EXE
```

Die Pfade zu den übrigen Office-Anwendungen finden wir auf die gleiche Weise.

Debuggen starten

Damit haben wir bereits einen großen Schritt zum Debuggen getan. Wir können nun nämlich das Projekt starten, indem wir die Anwendung starten.

Visual Studio trägt für das Projekt nun die Informationen in die Registry ein, die dazu notwendig sind, dass die aufgerufene Anwendung auf die DLL zugreifen kann.

Wenn wir beispielsweise Outlook angeben, wird anschließend Outlook gestartet und die DLL steht für die Nutzung bereit. Um diese einzusetzen, öffnen wir nun unter Outlook den VBA-Editor (zum Beispiel mit **Alt + F11**). Dann können wir die zu debuggende COM-DLL im **Verweise**-Dialog auswählen (siehe Bild 5).

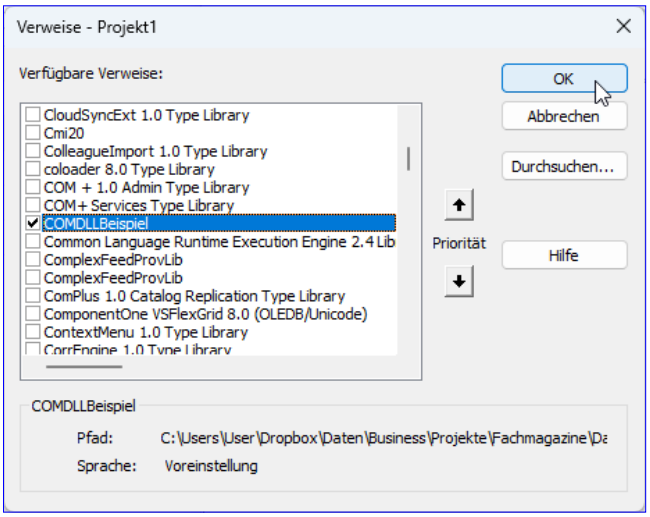

Bild 5: Die COM-DLL im **Verweise**-Dialog

Anschließend stehen die Funktionen der COM-DLL in einer neuen Prozedur in einem neuen Modul wie in Bild 6 zur Verfügung.

Haltepunkt setzen

Bevor wir nun ausprobieren, ob das Debugging funktioniert, setzen wir einen Haltepunkt für die Methode **Beispielprozedur** im VB.NET-Projekt (siehe Bild 7). Das gelingt genau wie im VBA-Editor durch Anklicken des grauen Bereichs auf Höhe der Zeile, die den Haltepunkt erhalten soll.

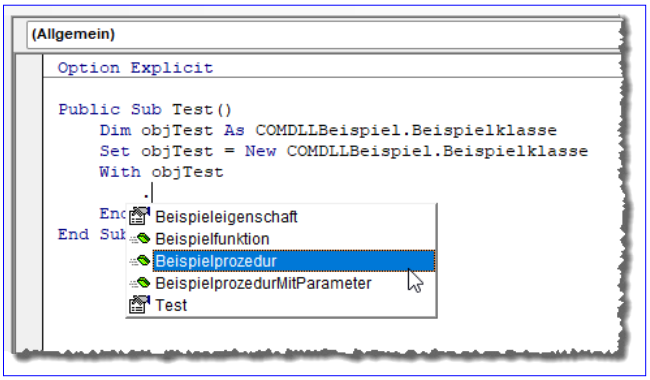

Bild 6: Nutzen der Elemente der COM-DLL

Haltepunkt ausprobieren

Nun starten wir das Projekt genau wie ein ausführbares Projekt mit **F5** oder der Schaltfläche **Starten**. Dies öffnet Outlook. Hier wechseln wir zu dem Modul, dem wir einen Aufruf der Methode **Beispielprozedur** hinzugefügt haben.

Führen wir die Prozedur nun aus, landen wir schnell am Haltepunkt in **Visual Studio .NET** (siehe Bild 8).

Bevor wir uns die Debugging-Techniken in Visual Studio .NET ansehen, werfen wir noch einen Blick auf die **Debugging**-Eigenschaften.

Debugging beenden

Den aktuellen Debugging-Vorgang beenden wir mit der Schaltfläche **Debugger beenden** mit dem roten

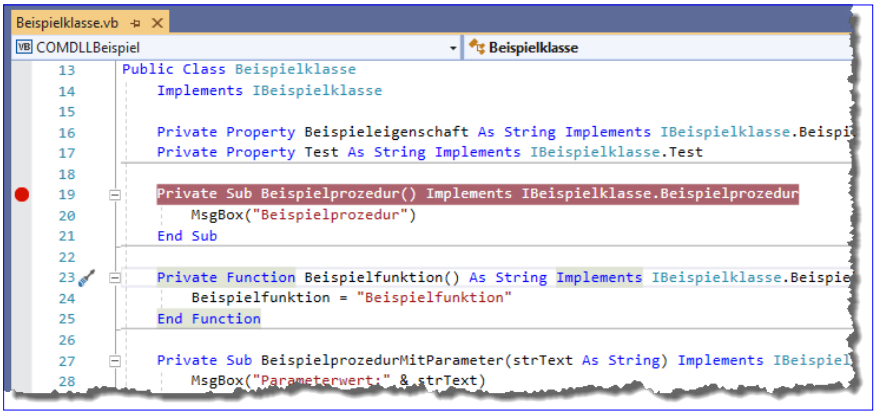

Bild 7: Setzen eines Haltepunktes in Visual Studio

Installation mit Inno Setup: Die Basics

Inno Setup ist ein zu verlässiges Tool zum Erstellen von Setups für verschiedene Anwendungen. Es funktioniert, wenn Du einfach nur eine Datenbank, eine Excel-Arbeitsmappe oder ein Word-Dokument in das gewünschte Verzeichnis beim Benutzer kopieren möchtest. Genauso kann es kompliziertere Vorgänge abbilden, die das Anlegen von Registry-Einträgen enthalten. Dieser Artikel zeigt, wie Du Inno Setup installierst und welche Möglichkeiten es gibt, die verschiedenen Einstellungen vorzunehmen. Neben dem textbasierten Editor gibt es nämlich auch noch eine professionelle Benutzeroberfläche. In weiteren Teilen schauen wir uns dann Setups und Einstellungen für verschiedene Zwecke an.

Warum Setups? Die folgende Auflistung zeigt, welche Aufgaben wir damit erledigen können:

- **Dateien kopieren**: Das Setup kopiert die erforderlichen Dateien von den Installationsquellen auf den Zielcomputer. Dies kann Programmdateien, Konfigurationsdateien, Bibliotheken, Grafiken, Dokumentationen und so weiter umfassen.

- **Verzeichnisse erstellen**: Das Setup kann Verzeichnisse auf dem Zielsystem erstellen, um die benötigten Dateien zu organisieren. Dies können Installationsverzeichnisse, Konfigurationsordner, temporäre Ordner et cetera sein.

- **Registry-Einträge anlegen**: Setups können Registrierungseinträge in der Windows-Registrierung erstellen, um Informationen über die installierte Anwendung zu speichern. Dies können Einstellungen, Dateiverknüpfungen, DLL-Pfade und andere Informationen sein, die von der Anwendung verwendet werden.

- **Verknüpfungen erstellen**: Setups können Verknüpfungen im Startmenü, auf dem Desktop oder in anderen Ordnern erstellen, um den Benutzern den einfachen Zugriff auf die installierte Anwendung zu ermöglichen.

- **Systemkonfiguration**: Einrichten von Systemkonfigurationen oder -einstellungen, die für die ordnungsgemäße Funktion der Anwendung erforderlich sind. Dies könnte die Konfiguration von Diensten, Umgebungsvariablen oder anderen Systemkomponenten beinhalten.

- **Abhängigkeiten verwalten**: Wenn die Anwendung von bestimmten Bibliotheken oder Komponenten abhängt, können diese Abhängigkeiten während der Installation eingerichtet werden. Dies umfasst beispielsweise das Installieren von Laufzeitbibliotheken oder anderen notwendigen Komponenten.

- **Benutzerinteraktion**: Einrichtung von Dialogfeldern, um den Benutzer während der Installation zu führen. Das können Lizenzvereinbarungen, Auswahl von Installationskomponenten, Festlegen von Installationspfaden und so weiter sein.

- **Deinstallation**: Neben der Installation kann das Setup auch Mechanismen zum Deinstallieren der Anwendung bereitstellen. Das beinhaltet das Entfernen von Dateien, Verzeichnissen, Registry-Einträgen und Verknüpfungen.

- **Updates und Patches**: Einrichtung von Mechanismen zur Aktualisierung oder Patchen der Anwen-

dung. Das können Updates sein, die bereits auf dem System installiert sind, oder Patches, die nachträglich angewendet werden.

- **Benachrichtigungen und Protokollierung**: Setups können Benutzer über den Fortschritt der Installation informieren und Protokolldateien erstellen, um Informationen über die Installation für spätere Überprüfung zu speichern.

All diese Funktionen werden normalerweise in einer Setup-Datei zusammengeführt, die als **.exe**-Datei geliefert wird und den Benutzer nach dem Starten durch den Installationsvorgang führt.

Bei der Installation umfangreicherer Anwendungen wie Office-Paketen et cetera enthält diese Datei meist nicht alle notwendigen Ressourcen, sondern lädt diese aus dem Internet nach.

Inno Setup

Zum Erstellen dieser Setup-Dateien kann man verschiedene Anwendungen nutzen, darunter kostenlose und kostenpflichtige Produkte. Im Laufe der Zeit hat sich für den Einsatz mit Setups rund um Microsoft Access Inno Setup als praktisches Tool erwiesen, das überdies kostenlos ist. Wir haben damit erfolgreich verschiedene Setups erstellt:

- Kopieren von Access-Datenbanken mit den zugehörigen Komponenten
- Zusätzliche Installation der Access-Runtime
- Installation von Add-Ins für Access, Excel, Word et cetera
- Installation von COM-Add-Ins für verschiedene Office-Anwendungen
- Installation von COM-DLLs für verschiedene Office-Anwendungen

Download und Installation von Inno Setup

Inno Setup finden wir im Internet unter der folgenden Adresse:

```
https://jrsoftware.org/isinfo.php
```

Hier gelangen wir schnell zum Downloa der aktuellen stabilen Version. Zum Zeitpunkt der Erstellung dieses Artikels war die Version 6.2.2 die aktuelle Version.

Das Installationspaket ist nur knapp fünf Megabyte groß. Bei der Installation wählen wir die Sprache aus, bestätigen die Lizenzbedingungen und legen fest, ob wir ein Desktop-Symbol anzeigen wollen. Außerdem legen wir fest, dass **.iss**-Dateien mit Inno Setup geöffnet werden sollen.

Nach der Zusammenfassung der gewählten Optionen starten wir die Installation und sind wenige Sekunden später bereits fertig.

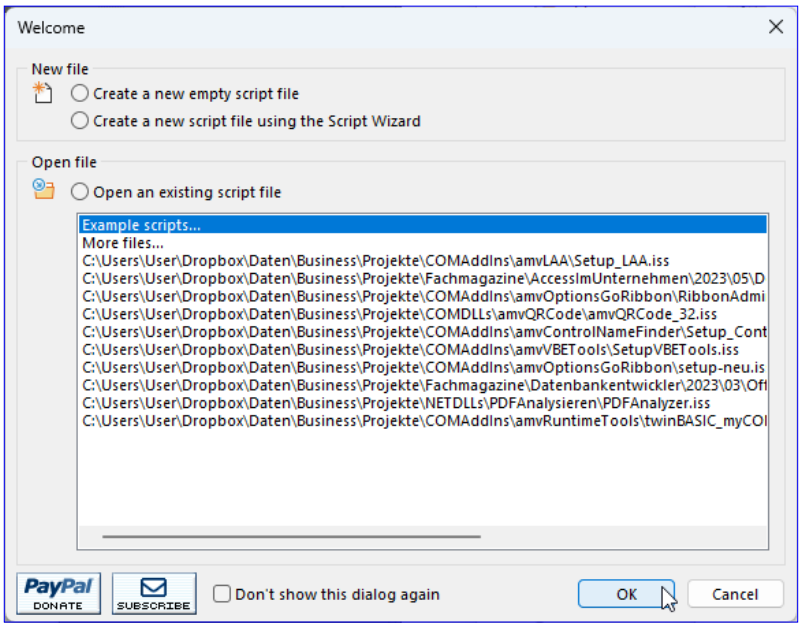

Bild 1: Erstellen eines neuen Setups

Beispiel-Setup

In diesem Artikel wollen wir an einem kleinen Beispiel demonstrieren, wir ein Setup mit dem Assistenten anlegen können. Dazu haben wir eine **.exe**-Datei erstellt, die wir auf dem Zielsystem installieren wollen.

Inno Setup starten

Nach dem Start von Inno Setup finden wir einen **Welcome**-Dialog vor, der uns verschiedene Optionen anbietet. Neben der Möglichkeit, vorhandene Setups zu öffnen (im Screenshot sind bereits einige vorhanden, weil ich Inno Setup zuvor schon genutzt habe), können wir ein neues, leeres Skript erstellen oder den Script Wizard benutzen (siehe Bild 1).

Hier bekommen wir nochmals Gelegenheit, ein neues, leeres Skript zu erstellen, was wir aber dankend ablehnen und mit einem Klick auf die Schaltfläche **Next** zum nächsten Schritt springen (siehe Bild 2).

Name, Version, Hersteller, Webseite

Dort geben wir die grundlegenden Informationen zum Setup ein, nämlich den Namen der Anwendung, die Version, den Hersteller und die Webseite der Anwendung (siehe Bild 3). Wir wählen hier die Bezeichnung **amvSetup** und passen auch die übrigen Einstellungen entsprechend an.

Zielordner definieren

Im folgenden Schritt legen wir fest, in welchen Ordner die Anwendung installiert werden soll. Die Vorauswahl ist der Ordner **Program Files Folder**. Daneben

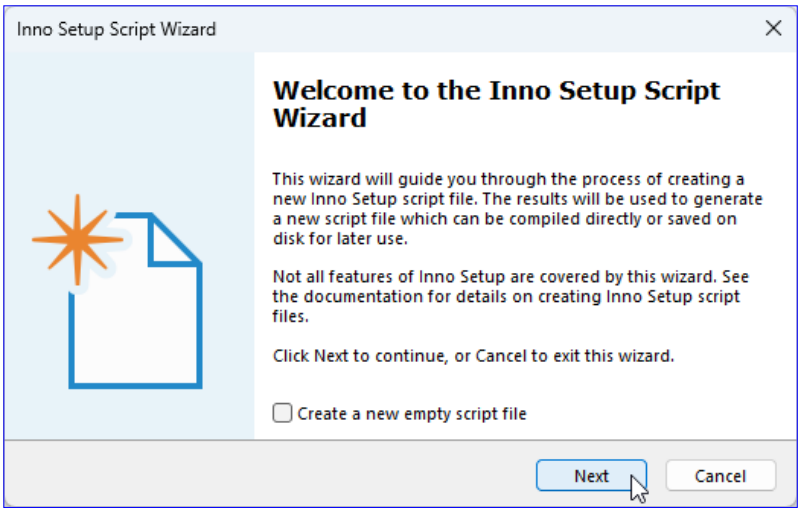

Bild 2: Start des Assistenten zum Anlegen eines neuen Skripts

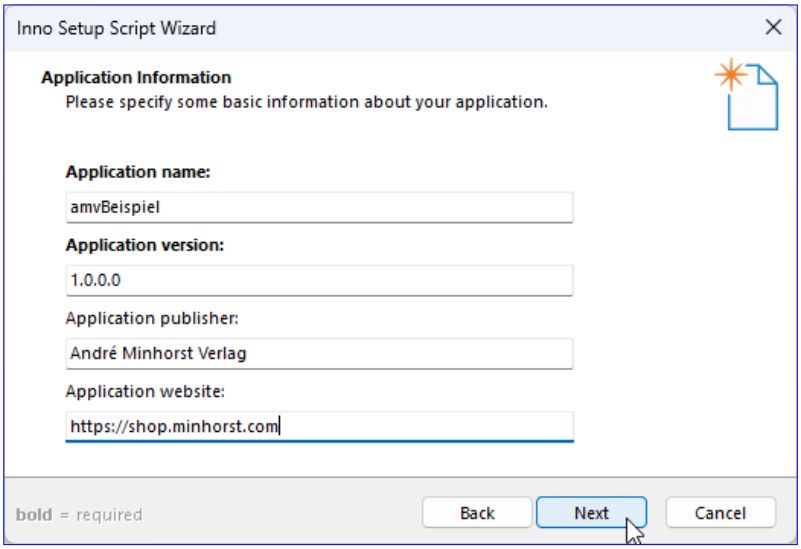

Bild 3: Abfrage der grundlegenden Informationen

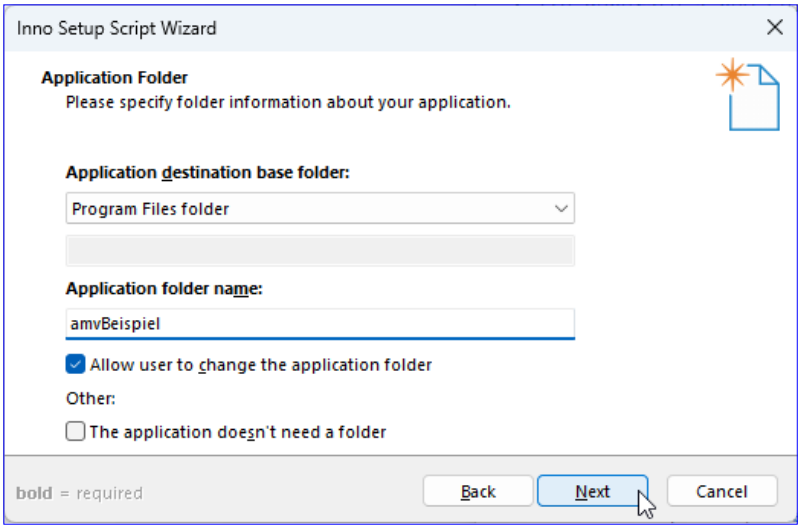

Bild 4: Abfrage des Zielverzeichnisses und Name des Anwendungsordners

finden wir nur noch den Eintrag **Custom**. Dieser erlaubt die manuelle Eingabe des Ordners. Die Möglichkeiten sind hier stark eingeschränkt, wir können im Skript selbst noch weit mehr Optionen angeben.

Der Ordner **Program Files Folder** ist aber ausreichend für uns, unsere Beispielanwendung soll nämlich genau dort installiert werden. Später sehen wir, wo die Anwendung dann landet.

Danach geben wir den Namen des neuen Ordners unterhalb von **Program Files Folder** an, in dem unsere Dateien landen sollen. Wir können außerdem einstellen, ob der Benutzer das Zielverzeichnis anpassen können soll und ob überhaupt ein neuer Ordner für die Anwendung angelegt werden soll (siehe Bild 4).

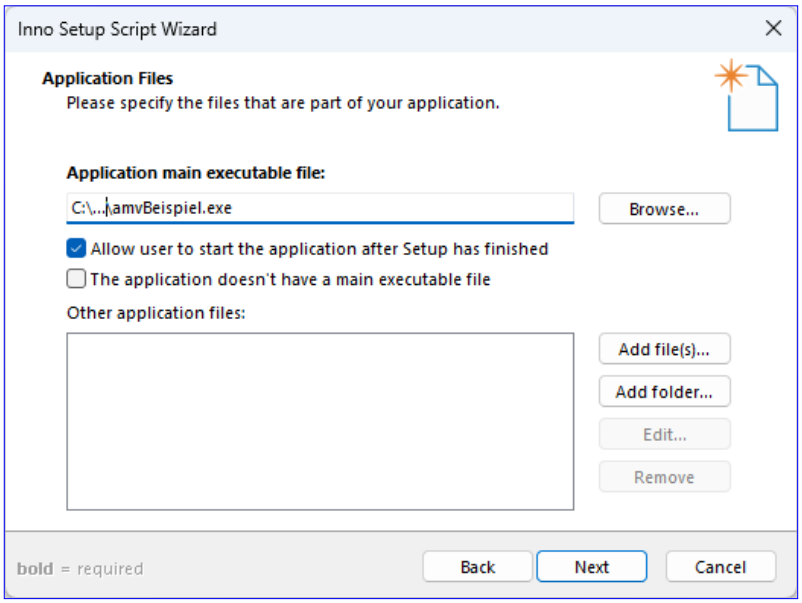

Bild 5: Auswahl der zu installierenden Dateien

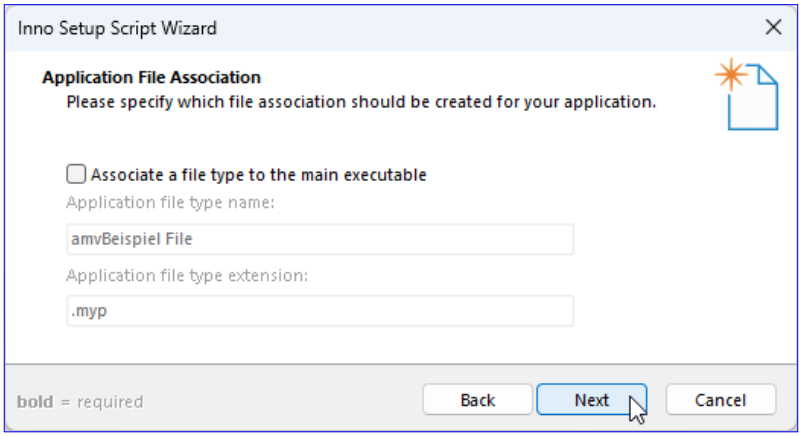

Bild 6: Angabe zu assoziierender Dateiendungen

Zu installierende Dateien auswählen

Danach wählen wir die Dateien aus, die installiert werden sollen. Dabei können wir als Erstes die ausführbare Datei angeben und angeben, ob diese nach dem Setup automatisch gestartet werden soll.

Hier wählen wir die zu Beispielzwecken erstellte **.exe**-Datei **amvBeispiel.exe** aus. Wir können aber auch angeben, dass es keine spezielle ausführbare Datei gibt. Darunter fügen wir gegebenenfalls weitere Dateien hinzu, die mit dem Setup installiert werden sollen – dies ist allerdings in diesem Fall nicht notwendig (siehe Bild 5).

Dateiendungen assoziieren

Im nächsten Schritt aus Bild 6 können wir noch angeben, ob wir eine spezielle Dateiendung mit unserer Anwendung assoziieren wollen. Das ist nicht der Fall, da wir ja einfach nur eine **.exe**-Datei installieren wollen.

Shortcut hinzufügen

Nun können wir noch entscheiden, ob wir einen Shortcut hinzufügen wollen (siehe Bild 7). Damit gehen einige weitere Optionen einher. Wir können entweder einfach einen vorgegebenen Eintrag zum Startmenü hinzufügen oder wir konfigurieren diesen genauer. Dazu können wir dem Benutzer die Möglichkeit geben, den Ordnernamen zu ändern, das Anlegen des Shortcuts zu unterbinden und mehr.

Installation mit Inno Setup: Bilder und Images

Wie heißt es so schön: Das Auge isst mit. Das gilt nicht nur bei der Nahrungsaufnahme, sondern auch beim Konsum der Optik eines Setups. Mit Inno Setup können wir an verschiedenen Stellen Icons und Images unterbringen. Wie das gelingt, welche Tricks man dabei anwenden kann und wie wir eventuelle Stolpersteine umgehen, schauen wir uns im vorliegenden Artikel an. Dabei stoßen wir an unerwarteter Stelle an die Grenzen, die uns der Windows Defender auferlegt. Doch eins nach dem anderen: Erst einmal schauen wir, wo wir in einen mit Inno Setup erstellten Setup Icons und Bilder unterbringen können.

Voraussetzungen

In diesem Artikel zeigen wir, wie man beim Erstellen von Setup-Routinen mit dem kostenlosen Tool Inno Setup Installationsroutinen mit Bildern versehen kann. Die Grundlagen zu Inno Setup stellen wir im Artikel **Installation mit Inno Setup: Die Basics** (**www.vbentwickler.de/382**) vor.

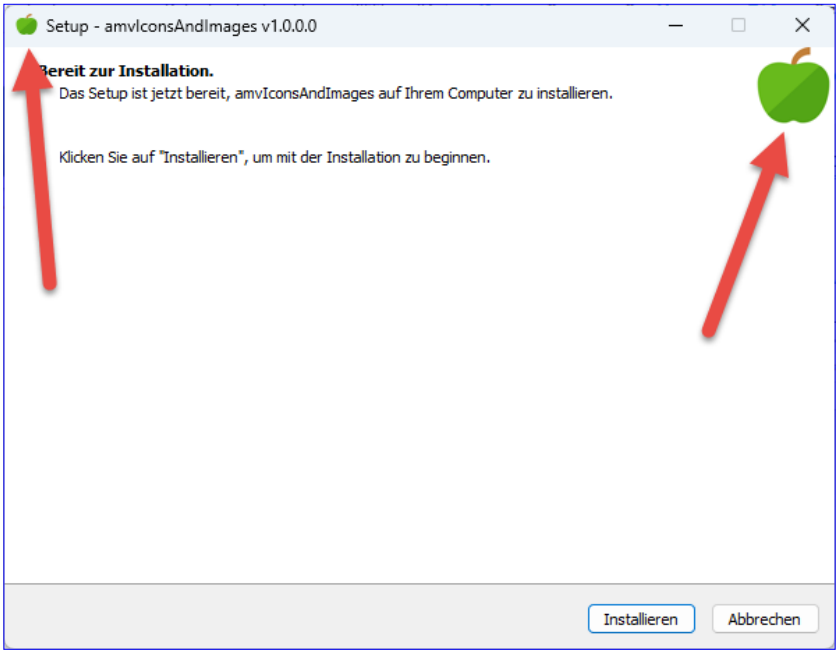

Bild 1: Bilder im Setup

Bitmaps und Icons

Eines vorweg: Wenn wir ein Setup wirklich mit hübschen Bildern versehen wollen, die auch noch einheitlich daherkommen, benötigen wir entweder ein Set von passenden Bildern oder wir investieren Zeit und Mühe in das Erstellen der verschiedenen Formate. Auch wenn es dazu ausreichend Tools gibt – wir bevorzugen fertige Icons und Bilder.

Im konkreten Fall nutzen wir en Produkt von **www.iconexperience.com**, das wir vor vielen Jahren erworben haben und das uns nach wie vor gute Dienste leistet. Hier finden wir sowohl **.ico**-Dateien, die jeweils verschiedene Größen enthalten, sowie **.png**-Dateien in den Größen 16x16, 24x24, 32x32, 48x48, 64x64, 128x128, 256x256 und 512x512. Im Falle verschiedener Bilder für ein mit Inno Setup erstelltes Setup benötigen wir allerdings **.bmp**-Dateien. Auch das ist mit dem angesprochenen Produkt kein Problem, denn es kommt mit einem Tool, mit dem wir die **.png**-Dateien in andere Formate wie **.bmp** oder **.gif** umwandeln können.

Im Setup angezeigte Bilder

Im Setup sehen wir drei verschiedene Bilder. Eine für die Eigenschaft **SetupIconFile** angegebene **.ico**-Datei wird als Icon links oben in der Titelleiste eingeblendet. Diese erscheint auf jeder Seite des Setups. Ein zweites

Bild erscheint rechts oben im eigentlich Inhaltsbereich. Dieses geben wir für die Eigenschaft **WizardSmallImageFile** an (siehe Bild 1).

Dieses Bild erscheint auf allen Seiten mit Ausnahme der letzten Seite.

Auf der letzten Seite des Setups finden wir außerdem noch eine weitere Bilddatei, die wir für die Eigenschaft **WizardImageFile** angeben (siehe Bild 2).

Bild 2: Großes Bild auf der letzten Seite

Benötigte Formate

Wir benötigen verschiedene Formate für die verschiedenen Anzeigeorte:

- **SetupIconFile**: Hier geben wir eine **.ico**-Datei mit den Abmessungen **16x16**, **32x32**, **48x48**, **64x64** oder **256x256** an.
- **WizardImageFile**: Für die Eigenschaft **WizardImageFile** geben wir den Namen der Datei an, die groß auf der letzten Seite des Setup-Assistenten angezeigt wird und die im Format **.bmp** vorliegen muss. Dafür können wir beispielsweise die folgenden Größen verwenden: **164x314**, **192x386**, **246x459**, **273x556**, **328x604**, **355x700**, **410x797**
- **WizardSmallImageFile**: Hier hinterlegen wir ebenfalls eine **.bmp**-Datei, diesmal in einem der Formate **55x55**, **64x68**, **83x80**, **92x97**, **110x106**, **119x123**, **138x140**. Größere Bilder werden heruntergeskaliert.
- **UninstallDisplayIcon**: Damit geben wir eine **.ico**-Datei an, die als Icon der Uninstall-Datei verwendet wird (siehe Bild 3).

Aufbau eines Setups mit allen Icons und Bildern

Wenn wir alle genannten Icons und Bilder in einem Setup nutzen wollen, können wir beispielsweise die minimale Konfiguration aus Listing 1 nutzen.

Hier sehen wir, dass alle **Bild**-Eigenschaften im **[Setup]**-Bereich untergebracht werden.

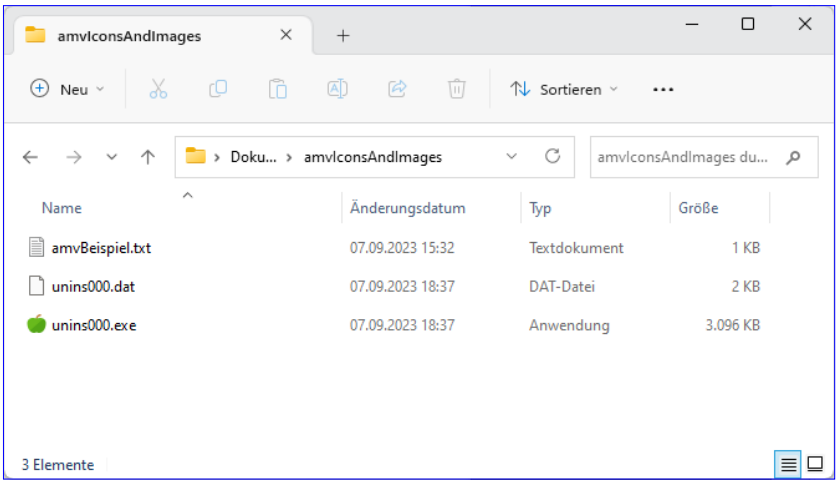

Bild 3: Icon für die Uninstall-Anwendung

Setup signieren mit Inno Setup

Die Windows-internen Sicherheitsfunktionen und die verschiedenen Virenscanner von Drittanbietern werden immer aufmerksamer, wenn es um das Herunterladen oder Installieren von Dateien geht – insbesondere .exe-Dateien sind grundsätzlich suspekt. Das ist uns in letzter Zeit häufig beim Bereitstellen von Setups von Tools aufgefallen. Wir wollen dies umgehen und sicherstellen, dass der Benutzer die von uns bereitgestellten Tools ohne Probleme installieren kann und sich nicht verunsichern lässt. Da wir bereits im Artikel »Installation mit Inno Setup: Die Basics« erläutert haben, wie wir Setups mit Inno Setup erstellen, schauen wir uns auch gleich an, wir dieses Tool zum Signieren unserer Setups nutzen können. Außerdem brauchen wir noch ein

Signieren

Signieren bedeutet, dass wir nachweisen, dass unsere Software aus einer vertrauenswürdigen Quelle stammt und vor allem auf dem Weg von der Herstellung zum Rechner des Kunden nicht manipuliert wurde. Wir können Dateien mit Tools wie dem von Microsoft bereitgestellten **signtool.exe** signieren, aber es gibt auch noch eine Reihe anderer Möglichkeiten. Wir wollen uns jedoch zuerst einmal auf **signtool.exe** konzentrieren, da dies eine kostenlose Möglichkeit bietet.

Hier gibt es wiederum verschiedene Möglichkeiten. Wir können eine bereits vorhandene **.exe**-Datei zuerst erstellen und diese dann per Aufruf von **signtool.exe** über die Kommandozeile signieren. Wenn wir eine Setup-Datei signieren können, geht dieser Weg auch – aber noch praktischer ist es, die Signierung direkt in das Tool zum Erstellen des Setups zu integrieren. Das gelingt mit Inno Setup recht einfach. In diesem Artikel schauen wir uns beide Möglichkeiten an. Zunächst wollen wir jedoch das benötigte **signtool.exe** installieren.

Zertifikat

Neben **signtool.exe** benötigen wir noch ein Zertifikat. Zertifikatsanbieter gibt es viele, die meisten bieten Zertifikate für Webseiten an. Wir suchen nach einem Zertifikat, das explizit für das Code Signing geeignet ist. Solche gibt es zu Preisen ab 200 EUR pro Jahr. Wer seine Softwareprodukte Kunden über das Internet zur Verfügung stellen möchte, tut vermutlich gut daran, diese Investition zu tätigen. Anderenfalls werden die Kunden immer wieder durch Warnmeldungen verunsichert, die beim Download oder bei der Installation der Dateien angezeigt werden.

Das ist vermutlich der aufwendigste Schritt. Welches Zertifikat ist das Richtige? Worauf soll ich achten? Welchen Preis zahle ich für das Zertifikat? Wie erhalte ich es? Wie viele Anwendungen kann ich damit zertifizieren? Diese und andere Fragen werden wir in diesem Artikel beantworten.

Ein Zertifikat erwerben oder selbst erstellen?

Wie können ein Zertifikat kaufen oder eines selbst erstellen. Wenn wir eines kaufen, haben wir die Wahl zwischen verschiedenen Anbietern und zwischen zwei verschiedenen Arten von Zertifikat. Als Erstes sollte man sich nicht in die Irre führen lassen und ein kostenloses oder sehr günstig erscheinendes Zertifikat von Anbietern wir Let's Encrypt et cetera kaufen. Diese sind in der Regel nicht für das Signieren von Code gedacht, sondern für die Verschlüsslung von Webseiten. Das sind jedoch zwei völlig unterschiedliche Dinge. Was wir suchen, nennt sich Code Signing Certificate

und wird nach unseren Recherchen nur in den USA angeboten. Dort finden wir Anbieter wie Certera, Comodo, Sectigo oder digicert. Forschen wir etwas weiter, sehen wir, dass es verschiedene Code Signing Certificates gibt, deren Preise sich unterscheiden. In Bild 1 finden wir beispielsweise zwei verschiedene Produkte namens **Certera Code Signing** und **Certera EV Code Signing**. Ersteres ist ein **Standard Validation Code Signing Certificate**, während Letzteres ein **Extended Validation Code Signing Certificate** ist. Der Name ist Programm – die EV-Version hat zusätzliche Funktionen.

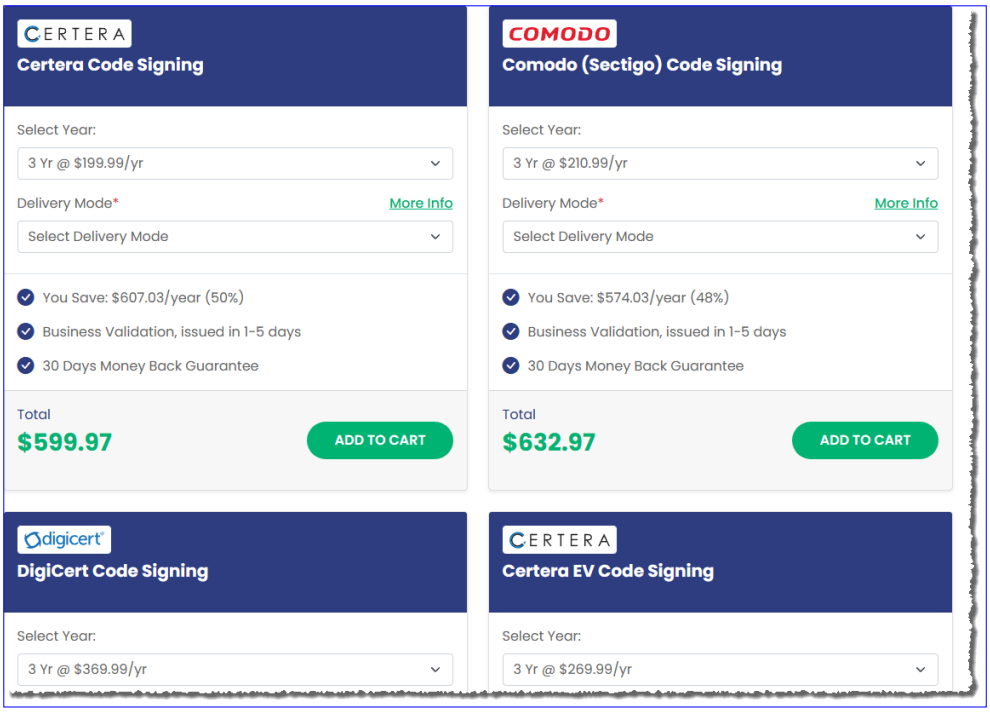

Bild 1: Verschiedene Zertifikate von unterschiedlichen Anbietern

Bei den Standard Validation Certificates gibt es wiederum OV-Zertifikate (für Organizational) und individuelle Zertifikate. Hier ist der Unterschied, dass wir beim OV-Zertifikat ein Unternehmen angeben und validieren lassen müssen, während dies für ein individuelles Zertifikat nicht nötig ist – hier muss nur die Person, auf die das Zertifikat ausgestellt wird, validiert werden. Das gelingt in der Regel über den Personalausweis oder ein ähnliches Dokument.

Die wichtigsten Unterschiede schauen wir uns in den folgenden Abschnitten an.

Validierung des Identitätsnachweises

- **Standard Validation Code Signing**: Bei der Standardvalidierung wird die Identität des Entwicklers oder Herausgebers des Codes auf Basis von grundlegenden Überprüfungen verifiziert, wie zum Beispiel das Überprüfen der Domaininhaberschaft oder des Organisationsnamens. Diese Validierung erfolgt schneller und erfordert weniger umfangreiche Dokumente.
- **Extended Validation Code Signing**: Die erweiterte Validierung erfordert eine umfassendere Überprüfung der Identität des Herausgebers. Dazu gehören strengere Identitätsnachweise, wie zum Beispiel rechtliche Dokumente und eingehende Überprüfungen, um sicherzustellen, dass die Identität des Herausgebers verifiziert ist. Dies führt zu einem höheren Maß an Vertrauen, da die Benutzer sicher sein können, dass der Code von einem vertrauenswürdigen Quellen stammt.

Sichtbarkeit des Zertifikats

- **Standard Validation Code Signing**: Zertifikate mit Standardvalidierung enthalten weniger ausführliche Informationen über den Herausgeber und die Organisation, was bedeutet, dass Benutzer weniger Informationen über die Authentizität des Codes haben.

- **Extended Validation Code Signing**: Zertifikate mit erweiterter Validierung enthalten zusätzliche Informationen über den Herausgeber und die Organisation. Dies wird durch eine spezielle Kennzeichnung im Zertifikat selbst angezeigt, was dem Benutzer eine höhere Sicherheitsebene signalisiert.

Vertrauenswürdigkeit und Anzeige:

- **Standard Validation Code Signing**: Code, der mit einem Standardvalidierungszertifikat signiert ist, wird oft als sicherer betrachtet als nicht signierter Code. Benutzer könnten jedoch weniger Vertrauen haben, da die Identität des Herausgebers nicht so stark überprüft wurde.

- **Extended Validation Code Signing**: Durch die strengere Validierung und die sichtbaren Indikatoren wird der Code mit erweiterter Validierung gegebenenfalls als als vertrauenswürdiger und sicherer angesehen. Benutzer sind eher bereit, auf solchen Code zuzugreifen.

Kosten und Aufwand

- **Standard Validation Code Signing**: Die Kosten und der Aufwand für die Erlangung eines Zertifikats mit Standardvalidierung sind in der Regel geringer, da die Anforderungen weniger streng sind. Außerdem ist die Standardvalidierung günstiger.

- **Extended Validation Code Signing**: Die erweiterte Validierung erfordert mehr Dokumentation und Zeit, was zu höheren Kosten und einem größeren Aufwand führen kann. Die erweiterte Validierung ist auch in der Anschaffung teurer.

Technische Unterschiede beim praktischen Einsatz

Im praktischen Einsatz gibt es keine nennenswerten Unterschiede zwischen **Standard Validation Code Signing** und **Extended Validation Code Signing**. Dabei beziehen wir uns darauf, wie Sicherheitsmechanismen von Windows auf die unterschiedlich signierten Dateien reagieren. Nach unseren Erfahrungen gibt es hier keine Unterschiede, da die Sicherheitsmechanismen solche Dateien in beiden Fällen nicht blockieren.

Tatsächlich müsste der Benutzer also schon die Signatur einsehen, um zwischen der Standardversion und der erweiterten Version unterschieden zu können.

Voraussetzungen für die Validierung

Die größte Hürde beim Erlangen eines Code Signing Certificates ist die Validierung. Diese ist, wie bereits erwähnt, bei der erweiterten Version wesentlich aufwendiger – genauso aufwendig wie bei der Organizational-Variante der Standardversion. Hier muss ein Unternehmen angegeben und validiert werden, was

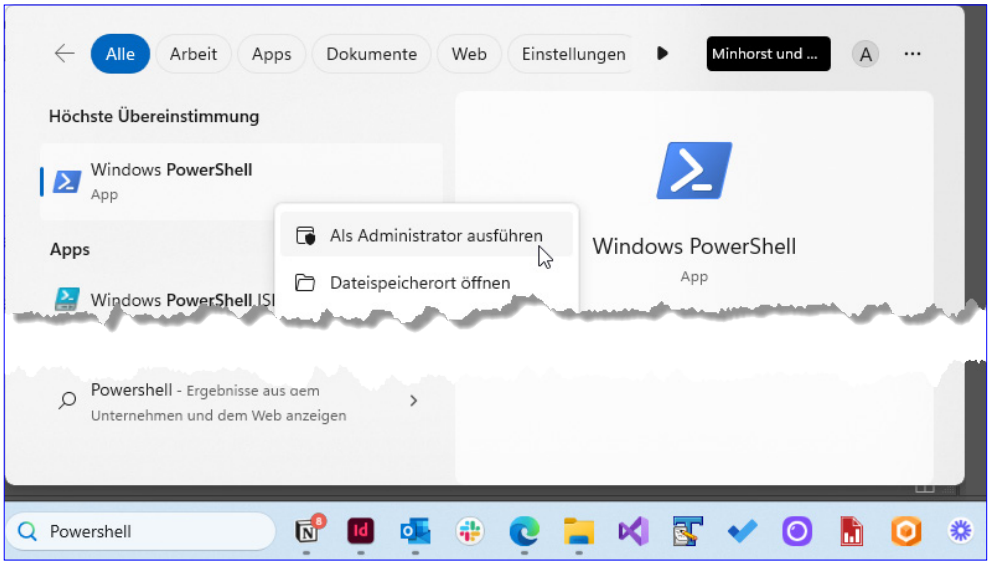

Bild 2: Starten der **PowerShell** als Administrator

zum Beipiel bei einer GbR oder einer ähnlichen Rechtsform, die nicht behördlich abgefragt werden kann, schwierig ist. Wenn wir hingegen ein Standardzertifikat mit Individual Validation wählen, reicht die Validierung über ein Dokument wie den Personalausweis aus.

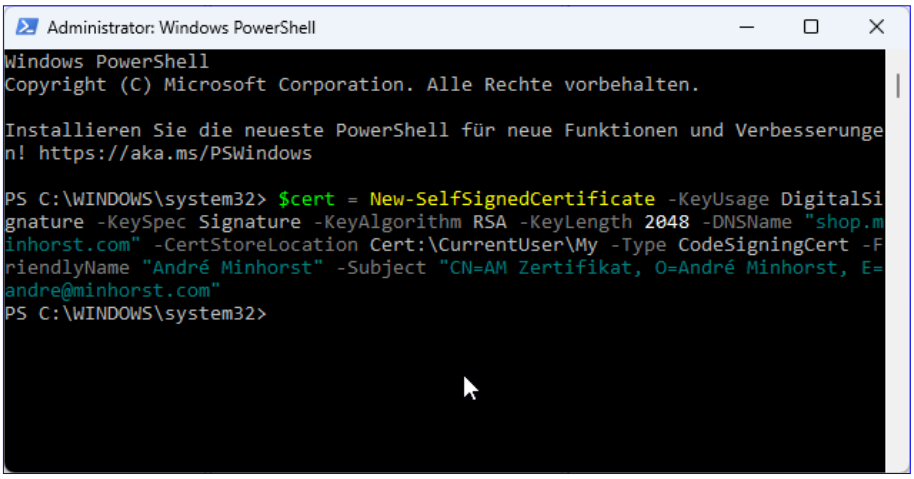

Bild 3: Starten der **PowerShell** als Administrator

Aushändigung des Zertifikats

Für alle genannten Arten von Zertifikaten erfolgt die Aushändigung über ein USB Token. Wer schnell mal ein Zertifikat benötigt, wird spätestens hier ausgebremst: Da alle uns bekannten Anbieter sich in den USA befinden, dauert allein die Zusendung einige Tage bis Wochen.

Self Signing

Die kostenlose, aber auch viel weniger wirkungsvolle Art der der Zertifizierung heißt Self Signing. Hier erstellt man selbst ein Zertifikat, das im Zertifikatsspeicher des eigenen Rechners gespeichert wird. Dieses ist dann beispielsweise von Inno Setup aus erreichbar.

Ein eigenes Zertifikat erstellen

Bevor wir uns den Einsatz eines echten Zertifikats mit USB-Token ansehen, werden wir zu Testzwecken ein Self Signing Certifikate erstellen. Dazu können wir die Power Shell verwenden, die wir als Administrator öffnen. Dazu geben wir die Zeichenkette **PowerShell** in das Suchfenster von Windows ein und klicken mit der rechten Maustaste auf den nun erscheinenden Eintrag **Windows PowerShell**. Aus dem Kontextmenü wählen wir dann den Eintrag **Als Administrator ausführen** aus (siehe Bild 2).

Danach geben wir den folgenden Befehl ein:

```
$cert = New-SelfSignedCertificate -KeyUsage DigitalSignature -KeySpec Signature -KeyAlgorithm RSA -KeyLength 2048 -DNSName "shop.minhorst.com" -CertStoreLocation Cert:\CurrentUser\My -Type CodeSigningCert -FriendlyName "André Minhorst" -Subject "CN=AM Zertifikat, O=André Minhorst, E=andre@minhorst.com"
```

Das Feedback der PowerShell-Konsole fällt recht sparsam aus (siehe Bild 3).

Zertifikat in der Microsoft Management Konsole einsehen

Wie also sehen wir nun, ob wir ein Zertifikat angelegt haben – und wo finden wir es? Dazu nutzen wir die

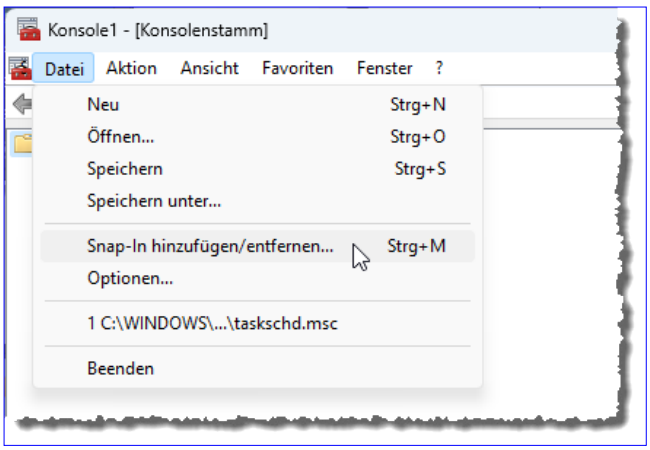

Bild 4: Hinzufügen eines neuen Snap-Ins zur Management Console

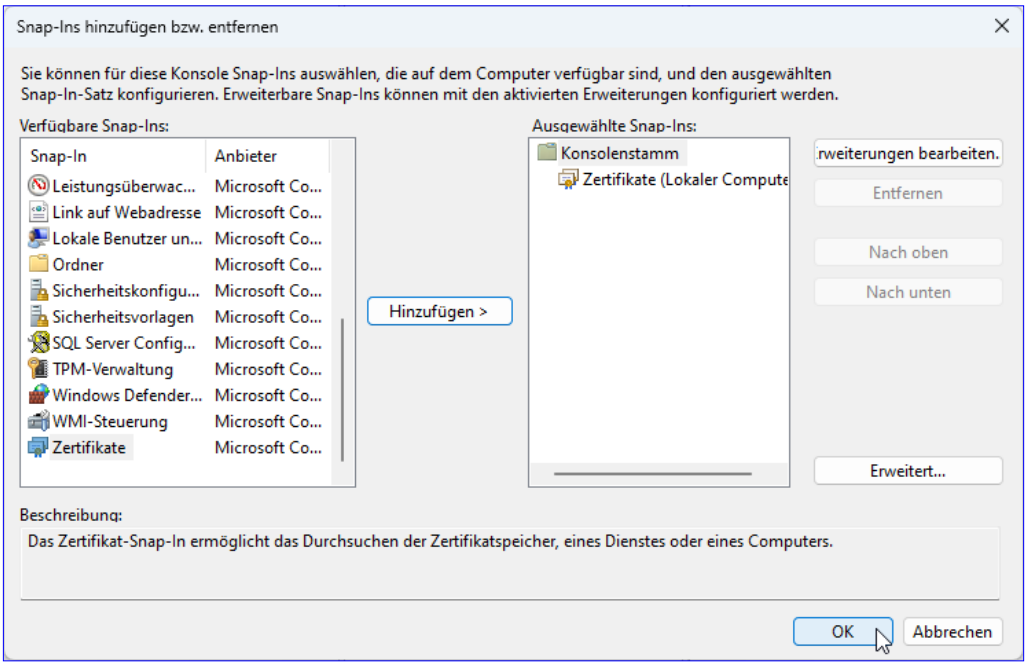

Bild 5: Hinzufügen des Snap-Ins für die Zertifikate

Im nun erscheinenden Dialog wählen wir in der linken Liste den Eintrag **Zertifikate** aus und klicken auf **Hinzufügen** (siehe Bild 5). Dies öffnet einen weiteren Dialog, wo wir im ersten Schritt die Option **Eigenes Benutzerkonto** und im zweiten die Einstellungen beibehalten.

Danach schließen wir den Dialog zum Hinzufügen von Snap-Ins wieder.

Microsoft Management Konsole, die wir durch Eingabe von **mmc** in die Windows-Suchfunktion finden und starten können.

Nach dem Start wählen wir hier den Befehl **Datei|Snap-In hinzufügen/entfernen...** aus (siehe Bild 4).

Anschließend ist die Management Konsole bereits besser gefüllt und wir können den Eintrag **Konsolenstamm|Zertifikate - Aktueller Benutzer|Eigene Zertifikate|Zertifikate** aufklappen. Hier finden wir unser soeben angelegtes Zertifikat (siehe Bild 6).

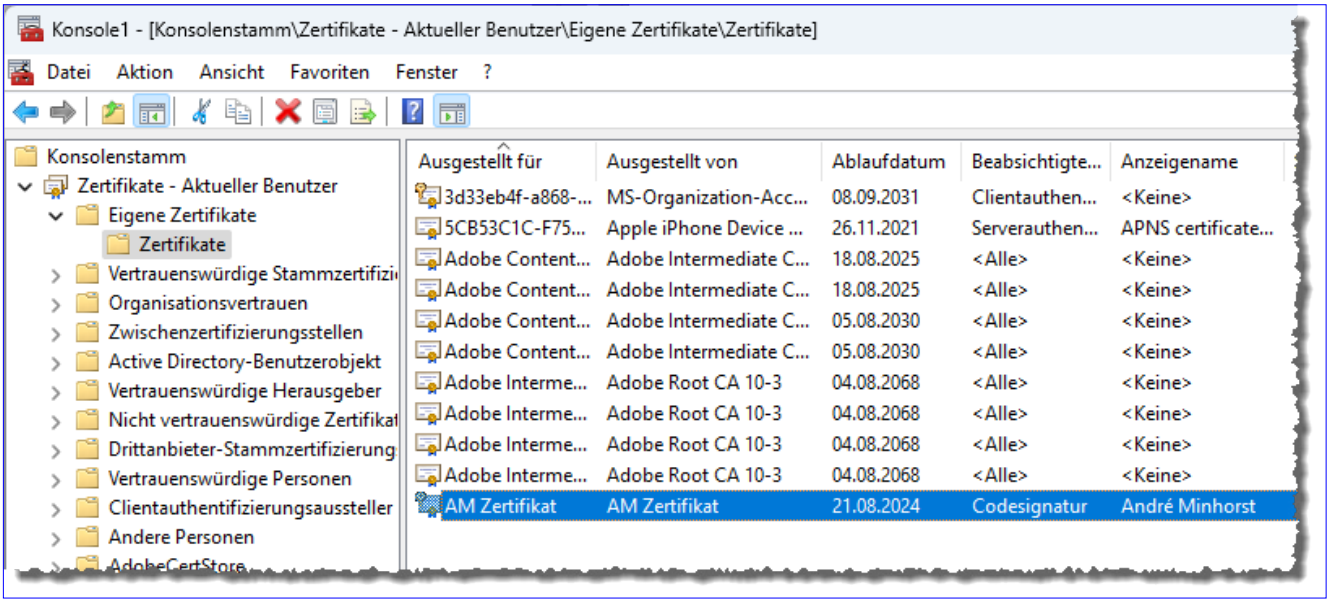

Bild 6: Unser Zertifikat in der Management Konsole

Per Doppelklick auf den Eintrag zeigen wir nun seine Details an. Auf der zweiten Seite finden wir die Einstellungen, die wir beim Anlegen des Zertifikats festgelegt haben (siehe Bild 7). Hier können wir auch Einstellungen ändern.

Damit haben wir bereits ein Zertifikat angelegt.

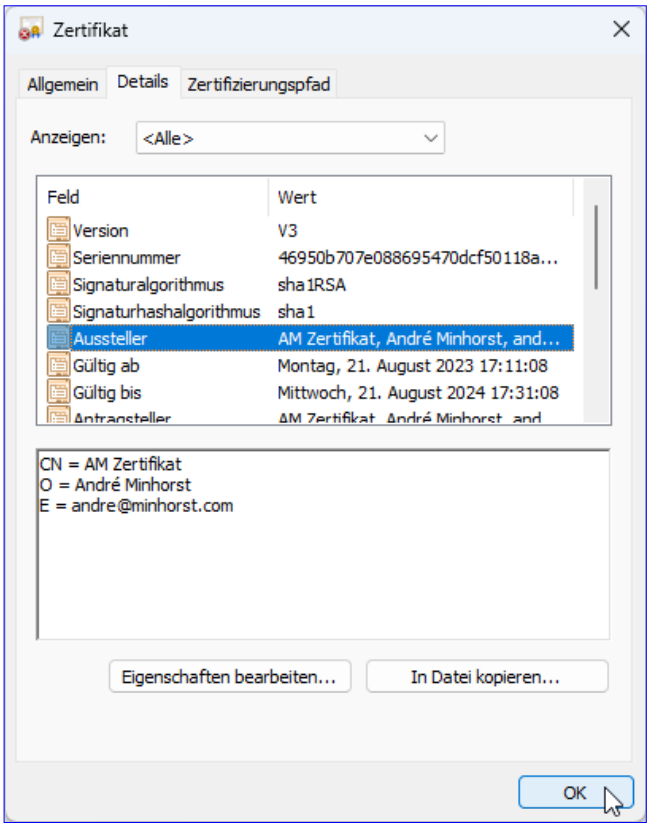

Bild 7: Details eines Zertifikats

signtool.exe installieren

Als Erstes benötigen wir das Programm **signtool.exe**. Dieses können wir über die Installation des Windows 10 SDK beziehen. Den Download finden wir beispielsweise unter dem folgenden Link:

```
https://go.microsoft.com/fwlink/?LinkID=698771
```

Alternativ suchen wir einfach auf Google nach **Windows 10 SDK Download**. Nach dem Download starten wir das Setup und wählen die direkte Installation aus (siehe Bild 8).

Wir benötigen nicht das vollständige SDK, sondern nur den Teil namens **Windows Software Development Kit** (siehe Bild 9).

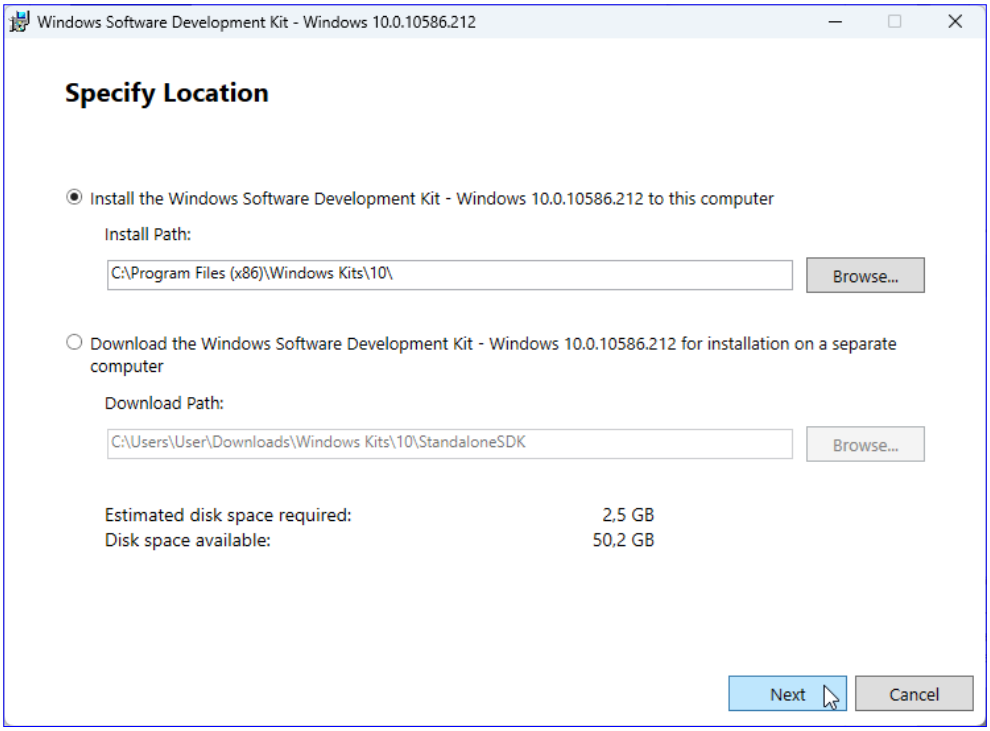

Bild 8: Start des Setups von Windows 10 SDK

Die Installation dauert eine Weile. Anschließend finden wir das Programm **signtool.exe** im Ordner **C:\Program Files (x86)\Windows Kits\10\bin\x86** (siehe Bild 10).

signtool.exe in Inno Setup nutzen

Inno Setup ist gut vorbereitet auf die Nutzung eines solchen Zertifikats.

COM-Add-In für Word: PDF-Export

Nachdem wir bereits gezeigt haben, wie man COM-Add-Ins für die Verwendung in anderen Office-Anwendungen nutzen kann, wollen wir nun auch einmal eines für Word programmieren. Die Aufgabe lautet, die verschiedenen Funktionen, die wir im Artikel »Word: PDF per VBA erzeugen« definiert haben, in Word für alle Dokumente bereitzustellen. Dazu erstellen wir ein neues COM-Add-In mit der Entwicklungsumgebung twinBASIC. Das COM-Add-In soll seine Funktionen per Ribbon bereitstellen, sodass der Benutzer mit wenigen Mausklicks die gewünschten Exporte erzeugen kann. Dazu gehört der Export des vollständigen Dokuments, aller Seiten einzeln, eines bestimmten Bereichs, der aktuellen Seite oder auch von Bereichen, die durch bestimmte Formatierungen eingeleitet werden.

Im Artikel **Word: PDF per VBA erzeugen** (**www.vbentwickler.de/384**) haben wir einige Methoden gezeigt, wie wir den Inhalt von Word-Dokumenten in PDF-Dokumente exportieren können. Diese sehen wie folgt aus:

- Komplettes Dokument als PDF speichern
- Jede Seite als eigenes PDF-Dokument extrahieren
- Bereiche nach Seitenzahl exportieren
- Exportieren der aktuellen Seite
- Aufteilen nach Eigenschaften wie Überschriften
- Speichern des markierten Textes in ein eigenes PDF-Dokument

Diese Funktionen wollen wir nun in ein COM-Add-In integrieren, das die Funktionen für alle Dokumente in einem eigenen Tab im Ribbon anzeigt (siehe Bild 1). Hier sehen wir auch die Möglichkeit, die Seitenzahlen zum Exportieren eines Bereichs zu exportieren. Außerdem können wir angeben, ob das PDF nach dem Export direkt geöffnet werden soll.

Voraussetzungen

COM-Add-Ins wie dieses erstellen wir mit dem Tool **twinBASIC**, das für die Erstellung von 32-Bit-Anwendungen kostenlos verfügbar ist. Mehr dazu findest Du

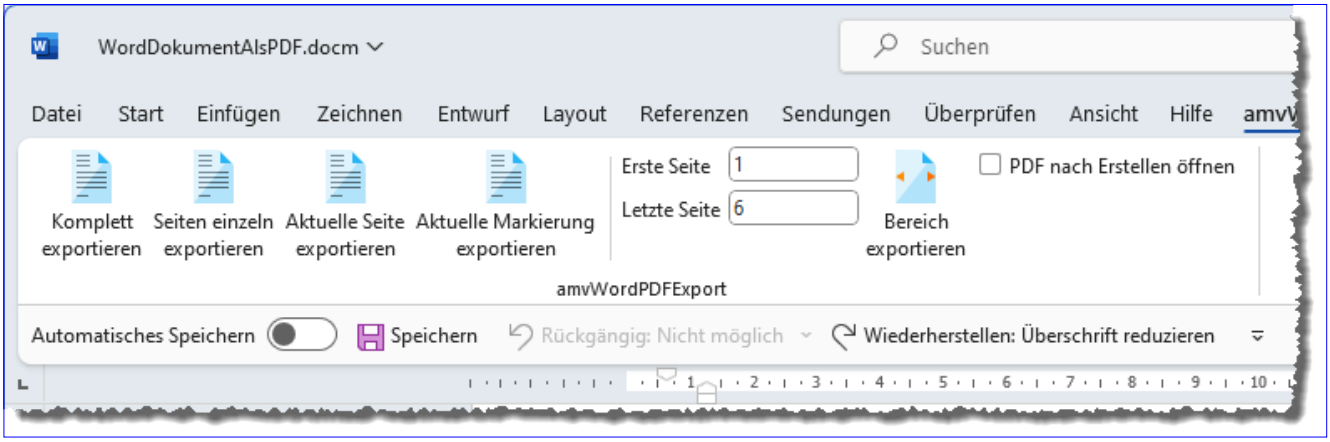

Bild 1: Das Ribbon-Tab für die Exportfunktionen

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

Listing 1: Diese Prozedur wird beim Starten von Word aufgerufen

in Artikeln wie **twinBASIC: Visual Basic für die Zukunft** (**www.vbentwickler.de/310**) und **COM-Add-Ins mit twinBASIC** (**www.vbentwickler.de/311**).

Da dort die Basics zum Erstellen von COM-Add-Ins bereits ausführlich dokumentiert wurden, konzentrieren wir uns hier auf die für unseren Anwendungszweck spezifischen Teile der Programmierung.

Starten des COM-Add-Ins

Beim Starten des COM-Add-Ins wird die Methode **OnConnection** der Schnittstelle **IDTExtentibility2** aufgerufen (siehe Listing 1).

Diese dient normalerweise dazu, einer Objektvariablen wie der folgenden einen Verweis auf die Anwendung hinzuzufügen, die das Add-In aufgerufen hat, in diesem Fall **Word.Application**. Diese landet in einem Standardmodul namens **mdlWord.twin**, weil wir von verschiedenen Modulen aus auf diese Variable zugreifen wollen:

```
Public objWord As Word.Application
```

Andererseits benötigen wir in diesem Fall aber auch die Möglichkeit, beim Öffnen eines neuen Dokuments oder beim Wechseln zu einem anderen Dokument auf das entsprechende Ereignis zu reagieren. Das ist notwendig, weil wir in die beiden Felder **Erste Seite** und **Letzte Seite** des Ribbons die erste und die letzte Seite des aktuellen Dokuments eintragen wollen. Dazu deklarieren wir also noch eine Variable namens **objWordEvents**, diesmal mit dem Schlüsselwort **WithEvents**:

```
Private WithEvents objWordEvents As Word.Application
```

Diese füllen wir und gleichzeitig implementieren wir die beiden Ereignisse **DocumentOpen** und **DocumentChange** für diese Variable:

```
Private Sub objWordEvents_DocumentOpen(ByVal Doc As _
        Word.Document)
    RibbonAktualisieren
End Sub

Private Sub objWordEvents_DocumentChange()
    RibbonAktualisieren
End Sub
```

Beide rufen die folgende Prozedur auf, welche das aktuelle Dokument referenziert, sofern dieses überhaupt vorhanden ist. Dann ermitteln wir die erste und die letzte Seite und schreiben diese in die Variablen **lngErsteSeite** und **lngLetzteSeite**. Diese Variablen deklarieren wir wiederum im Standardmodul **mdlWord.twin**.

Schließlich rufen wir die **Invalidate**-Methode der mit **objRibbon** referenzierten Ribbon-Definition auf, damit diese die anzuzeigenden Werte mit den entsprechenden **get...**-Callbackfunktionen erneut aufruft:

```
Private Sub RibbonAktualisieren()
    Dim objDocument As Word.Document
    If Not objRibbon Is Nothing Then
        On Error Resume Next
        Set objDocument = objWord.ActiveDocument
        On Error GoTo 0
        If Not objDocument Is Nothing Then
            lngErsteSeite = 1
            lngLetzteSeite = objWord.ActiveDocument. _
```

```
Private Function GetCustomUI(ByVal RibbonID As String) As String Implements IRibbonExtensibility.GetCustomUI
    Dim strXML As String
    strXML &= "<customUI xmlns=""http://schemas.microsoft.com/office/2006/01/customui"" loadImage=""loadImage" _
        & """ onLoad=""onLoad"">" & vbCrLf
    strXML &= "  <ribbon startFromScratch=""false"">" & vbCrLf
    strXML &= "    <tabs>" & vbCrLf
    strXML &= "      <tab id=""tab"" label=""amvWordPDFExport"">" & vbCrLf
    strXML &= "        <group id=""grp"" label=""amvWordPDFExport"">" & vbCrLf
    strXML &= "          <button id=""btnExportKomplett"" size=""large"" label=""Komplett exportieren"" image=" _
        & """word_pdf_book.ico"" onAction=""onAction""/>" & vbCrLf
    strXML &= "          <button id=""btnExportEinzelneSeiten"" size=""large"" label=" _
        & """Alle Seiten einzeln exportieren"" image=""word_book_pdfs.ico"" onAction=""onAction""/>" & vbCrLf
    strXML &= "          <button id=""btnExportAktuelleSeite"" size=""large"" label=""Aktuelle Seite exportieren"" " _
        & "image=""word_pdf.ico"" onAction=""onAction""/>" & vbCrLf
    strXML &= "          <button id=""btnExportAktuelleMarkierung"" size=""large"" label=" _
        & """Aktuelle Markierung exportieren"" image=""word_pdf_range.ico"" onAction=""onAction""/>" & vbCrLf
    strXML &= "          <separator id=""sep1""/>" & vbCrLf
    strXML &= "          <editBox id=""txtErsteSeite"" label=""Erste Seite"" onChange=""onChange"" getText=" _
        & """getText""/>" & vbCrLf
    strXML &= "          <editBox id=""txtLetzteSeite"" label=""Letzte Seite"" onChange=""onChange"" getText=" _
        & """getText""/>" & vbCrLf
    strXML &= "          <button id=""btnExportBereich"" size=""large"" label=""Bereich exportieren"" image=" _
        & """word_pdf_pages.ico"" onAction=""onAction""/>" & vbCrLf
    strXML &= "          <checkBox id=""chkOeffnen"" label=""PDF nach Erstellen öffnen"" getPressed=" _
        & """GetPressedCheckBox"" onAction=""onActionCheckBox""/>" & vbCrLf
    strXML &= "        </group>" & vbCrLf
    strXML &= "      </tab>" & vbCrLf
    strXML &= "    </tabs>" & vbCrLf
    strXML &= "  </ribbon>" & vbCrLf
    strXML &= "</customUI>" & vbCrLf
    Return strXML
End Function
```

Listing 2: Definition des Ribbons

```
            ComputeStatistics(wdStatisticPages)
        objRibbon.Invalidate
    End If
  End If
End Sub
```

Ribbon des COM-Add-Ins definieren

Damit kommen wir zu einem wichtigen Element, nämlich der Schnittstelle zum Aufrufen der Funktionen des COM-Add-Ins. Diese definieren wir in der Funktion **GetCustomUI**, die beim Starten von Word automatisch aufgerufen wird. Hier setzen wir in der Variablen **strXML** die Ribbon-Definition zusammen und geben diese dann zurück (siehe Listing 2).

Das Element **customUI** gibt die beiden Prozeduren namens **LoadImage** und **OnLoad** an, die beim Laden beziehungsweise für Element mit **image**-Element aufgerufen werden sollen. **OnLoad** soll beim Laden des Ribbons einen Verweis auf dieses in der Variablen **objRibbon** speichern, die im Modul **mdlWord** deklariert wird: