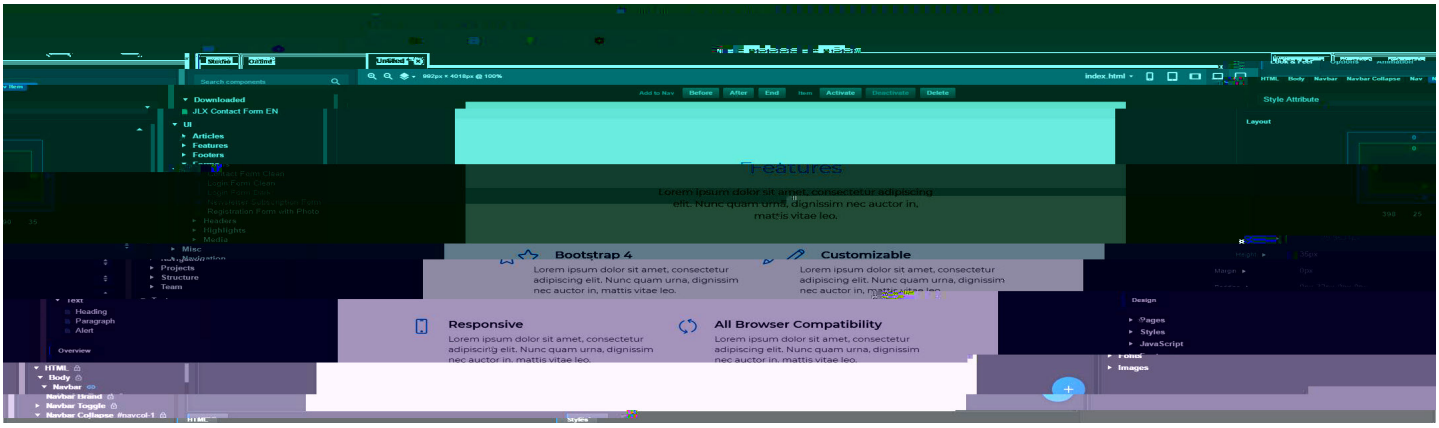
AUSGABE 04-05/2019

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER
© 2019 André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

Mehrzeilige Zeichenketten

Unter VBA war die Eingabe mehrzeiliger Zeichenketten eine Qual. Man musste sich mit dem Unterstrich und zusätzlich eingefügten Zeilenumbrüchen mit vbCrLf oder Chr(10) & Chr(13) durchschlagen. Unter Visual Basic ab der Version 14 (mit Visual Studio 2015) gibt es eine tolle Neuerung für alle, die mit Visual Basic in Visual Studio programmieren: Die Eingabe mehrzeiliger Texte wurde deutlich vereinfacht. Wie genau, lesen Sie hier.

Wenn Sie unter VBA eine mehrzeilige Zeichenkette eingeben wollten, die beispielsweise in einer Variablen landen sollte, benötigten Sie – ohne Zeilenumbrüche im Code – zumindest das folgende Konstrukt:

```
strText = "Erste Zeile" & vbCrLf & "Zweite Zeile"
```

Oder Sie haben so etwas genutzt, um die Zeilen auch im Code abzubilden:

```
strText = "Erste Zeile" & vbCrLf
strText = strText & "Zweite Zeile"
```

Etwas eleganter, aber mitunter noch unübersichtlicher war die Variante mit dem Unterstrich als Zeilenfortsetzungszeichen:

```
strText = "Erste Zeile" & vbCrLf _
    & "Zweite Zeile"
```

Sollten Sie das Zeilenumbruchszeichen **vbCrLf** weglassen, können Sie immerhin noch lange Texte im Code auf mehrere Zeilen aufteilen. Wenn Sie dann zu Visual Basic in Visual Studio wechseln, erwarten Sie eine komfortablere Eingabe solcher Zeichenketten. Allerdings waren die Möglichkeiten bis Visual Studio 2015 noch genauso aufwendig wie unter VBA. Mit Visual Studio 2015 und Visual Basic 14 hat sich hier jedoch einiges getan. In einer Zeile können Sie den Text wie schon wie unter VBA schreiben:

```
strText = "Erste Zeile" & vbCrLf & "Zweite Zeile"
```

Allerdings benötigen Sie dazu den Namespace **Microsoft.VisualBasic.Constants**, da **vbCrLf** nur noch aus Kompatibilitätsgründen vorliegt, also etwa:

```
strText = "Erste Zeile" & Microsoft.VisualBasic.Constants.vbCrLf & "Zweite Zeile"
```

Einfacher geht es mit der extra dafür vorgesehenen Konstanten der Klasse **Environment**:

```
strText = "Erste Zeile" & Environment.NewLine & "Zweite Zeile"
```

Damit betreiben wir aber schon zu viel Aufwand, denn es geht auch noch viel einfacher. Die folgende Version wird seit Einführung von Visual Basic 14 unterstützt:

String-Interpolation in Zeichenketten

Wer unter Visual Basic oder VBA komplizierte Zeichenketten wie etwa Abfragen mit vielen UNION-Verknüpfungen zusammenstellen muss, die auch noch zur Laufzeit um variable Texte wie etwa Vergleichswerte ergänzt werden sollen, macht schnell Fehler. Mit Visual Basic 14 hat Microsoft die sogenannte String-Interpolation auch in Visual Basic eingeführt. Damit wird die Lesbarkeit von Zeichenketten durch den Einsatz von Platzhaltern in geschweiften Klammern stark verbessert. Dieser Artikel zeigt die Möglichkeiten der String-Interpolation auf.

Beispiele: Die Beispiele sind in der Datei **StringinterpolationVB.linq**, die Sie in **LINQPad** ausführen können (**www.linqpad.net**).

Wenn Sie unter VBA oder unter älteren Versionen von Visual Basic Zeichenketten mit dem Inhalt aus Variablen anreichern wollten, mussten Sie die Variablen durch entsprechende Textverkettungen hinzufügen. Unter VBA sieht das etwa wie folgt aus:

```
Dim strText As String
strText = "Zwei"
Debug.Print "Eins, " & strText & ", Drei"
```

Damit erhalten wir die folgende Ausgabe im Direktbereich:

```
Eins, Zwei, Drei
```

Wie man sich leicht vorstellen kann, werden derartige Konstrukte schnell unlesbar – vor allem, wenn es sich um SQL-Abfragen handelt, die noch einige Klammern und **UNION**-Schlüsselwörter enthalten. Für diesen Fall gibt es seit Visual Basic 14 eine neue Möglichkeit, die wesentlich übersichtlicher erscheint – die sogenannte String-Interpolation. Dabei gibt es zunächst folgende wichtige Regeln:

- Die Zeichenkette, in die Texte eingefügt werden sollen, enthält vor dem ersten Anführungszeichen das Dollar-Zeichen (**$**), also beispielsweise **strText = $"..."**
- Der einzufügende Text befindet sich in einer Variablen und diese wird vor dem Zugriff auf den Text gefüllt (zum Beispiel **strText**).
- Die Zeichenkette, in die der Text eingefügt werden soll, enthält an der Stelle der Einfügung den Namen der Variablen in geschweiften Klammern, also etwa **{strText}**.

Daraus resultiert unser erstes Beispiel:

```
Dim strText As String
```

```
strText = "Zwei"
Debug.Print ($"Eins, {strText}, Drei")
```

Diese Anweisungen liefern genau das gleiche Ergebnis wie das erste Beispiel von oben (**Eins, Zwei, Drei**).

Formatierungen von interpolierten Strings

Wenn Sie mit interpolierten Strings Zahlen einfügen, können Sie diese sogar noch formatieren. Dazu geben Sie in der geschweiften Klammer als zweites Argument durch einen Doppelpunkt getrennt den Ausdruck an, mit dem der übergebene Wert formatiert werden soll. Die Zahl **2,3456** wird im folgenden Beispiel als Währung ausgegeben. Dafür sorgt die mit **C** angegebene Formatierung:

```
Dim decZahl As Decimal
decZahl = 2.3456
Debug.Print ($"Zahl mit Währungsformat: {decZahl:C}")
```

Das Ergebnis lautet:

```
Zahl mit Währungsformat: 2,35 €
```

Formatierungen

Schauen wir uns an, welche Formatierungen unter anderem möglich sind:

- Ausrichtung. Sie können durch eine Zahl, die Sie durch Komma von der Variablen getrennt angeben, die Breite des für die Ausgabe des Platzhalters reservierten Bereichs definieren, also die Feldbreite. Wenn Sie eine positive Zahl angeben, wird der Inhalt rechts ausgerichtet, bei einer negativen Zahl links. Ist der Text länger als die reservierte Zeichenanzahl, wird diese Angabe nicht berücksichtigt:

```
Debug.Print($"Ausrichtung links:   {strText,-10}|")
    Ausrichtung links:   Zwei      |
Debug.Print ($"Ausrichtung rechts: {strText, 10}|")
    Ausrichtung rechts:       Zwei|
```

- Datumsformatierung. Wenn Sie ein Datum in den String hinein interpolieren, können Sie dieses durch die Angabe verschiedener Buchstaben formatieren. Hier sind einige Beispiele, die selbsterklärend sind:

```
Debug.Print($"Datum (d): {datDatum:d}")
   Datum (d): 04.10.2019
Debug.Print($"Datum (f): {datDatum:f}")
   Datum (D): Freitag, 4. Oktober 2019
Debug.Print($"Datum (D): {datDatum:D}")
   Datum (f): Freitag, 4. Oktober 2019 21:32
```

Mit Dateien und Verzeichnissen arbeiten

Wer unter Visual Studio Anwendungen mit Visual Basic programmiert, wird früher oder später an den Punkt kommen, wo er mit Dateien und Verzeichnissen arbeitet. Dabei gibt es Aufgaben wie zu ermitteln, ob eine Datei bereits vorhanden ist, eine Datei von einem Verzeichnis in ein anderes zu kopieren, alle Dateien eines Verzeichnisses zu durchlaufen oder dies gar rekursiv für untergeordnete Verzeichnisse und Dateien zu erledigen. Natürlich bietet Microsoft passende Möglichkeiten an, die kaum noch etwas mit den von VBA gewohnten Befehlen zu tun haben und auch nicht mit den Methoden des File-SystemObjects.

Die Beispiele dieses Artikels haben wir mit dem Tool **LINQPad** durchgespielt, das die einfache Eingabe von Visual Basic- und C#-Anweisungen und deren Ausführung erlaubt. Sie müssen hier nicht immer erst das Projekt kompilieren, um die enthaltenen Anweisungen zu debuggen, sondern können dies quasi ad hoc erledigen (Download siehe **https://www.linqpad.net**).

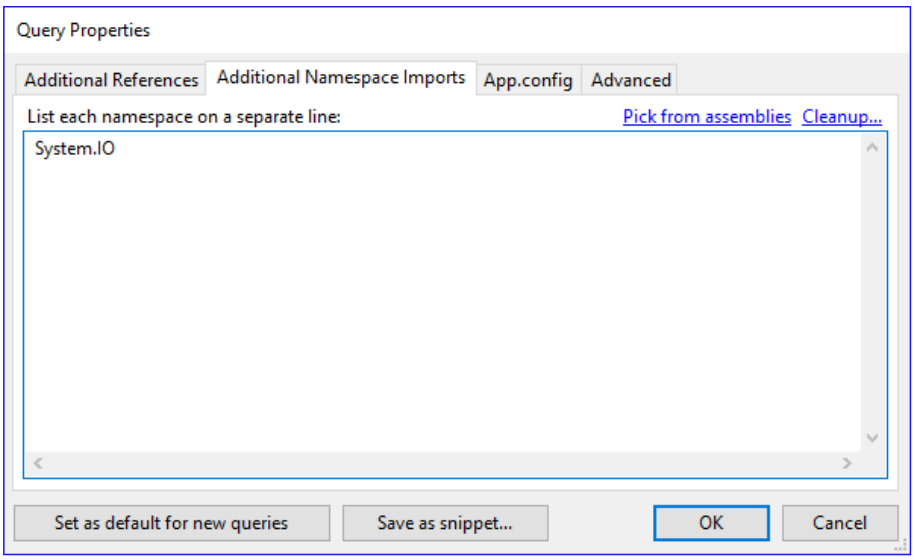


Bild 1: Hinzufügen von Namespaces

Der Namespace System.IO

Die Objekte, Methoden und Eigenschaften, die wir in diesem Artikel vorstellen, befinden sich im Namespace **System.IO**. Um diesen für eine Datei in LINQPad verfügbar zu machen, betätigen Sie die Taste **F4**. Dadurch wird der Dialog **Query Properties** geöffnet. In diesem wechseln Sie zum zweiten Registerreiter **Additional Namespace Imports**. Hier tragen Sie einfach wie in Bild 1 den folgenden Namespace ein:

```
System.IO
```

Danach stehen die Elemente dieses Namespaces in dem entsprechenden Fenster von **LINQPad** zur Verfügung.

Laufwerke

Das oberste Element der Hierarchie der Elemente des Dateisystems sind die Laufwerke. Diese können wir über die Funktion **GetDrives** auslesen und erhalten damit Objekte des Typs **DriveInfo**. Diese wiederum liefern einige Eigenschaften, mit denen wir die Informationen zum jeweiligen Laufwerk auslesen können.

Laufwerke ausgeben

Die Anweisungen, die wir ausprobieren wollen, fügen wir jeweils in die **Sub Main**-Prozedur ein. Wir steigen gleich ein und wollen alle Laufwerke des aktuellen Rechners im Direktfenster ausgeben. Dazu deklarieren wir zunächst ein Array namens **objDrives** und eine Variable namens **objDrive**, beide vom Typ **DriveInfo**. Es gibt kein Auflistungsobjekt, sondern eine Funktion namens **GetDrives**, die alle Laufwerke zurückliefert. Das Ergebnis, das wir zum Array **objDrives** hinzufügen, durchlaufen wir in einer **For Each**-Schleife. Dabei referenzieren wir das aktuelle Element jeweils mit der Variablen **objDrive** und geben den Wert seiner Eigenschaft **Name** aus:

```
Sub Main
    Dim objDrives() As DriveInfo
    Dim objDrive As DriveInfo
    objDrives = DriveInfo.GetDrives()
    For Each objDrive In objDrives
        Debug.print (objDrive.Name)
    Next objDrive
End Sub
```

Das Ergebnis sieht im Fall des hier verwendeten Rechners etwa wie in Bild 2 aus.

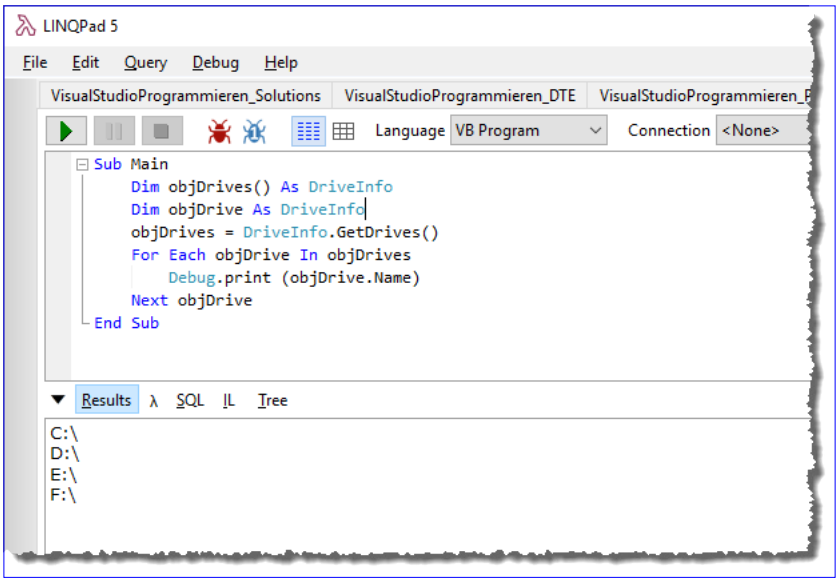


Bild 2: Ausgabe der Laufwerksnamen

Eigenschaften von Laufwerken

Das **DriveInfo**-Objekt bietet eine ganze Reihe von Eigenschaften. Mit den folgenden Anweisungen geben wir diese im Direktbereich aus:

```
For Each objDrive In objDrives
    Debug.print (objDrive.Name)
    Debug.Print("AvailableFreeSpace: " & objDrive.AvailableFreeSpace.ToString())
    Debug.Print("DriveFormat:        " & objDrive.DriveFormat.ToString())
    Debug.Print("DriveType:          " & objDrive.DriveType.ToString())
    Debug.Print("IsReady:            " & objDrive.IsReady.ToString())
    Debug.Print("RootDirectory:      " & objDrive.RootDirectory.ToString())
    Debug.Print("TotalFreeSpace:     " & objDrive.TotalFreeSpace.ToString())
    Debug.Print("TotalSize:          " & objDrive.TotalSize.ToString())
    Debug.Print("VolumeLabel:        " & objDrive.VolumeLabel.ToString())
Next objDrive
```

Die Eigenschaft **IsReady** ist beispielsweise interessant für CD/DVD-Laufwerke. Ist keine CD oder DVD eingelegt, liefert **IsReady** den Wert **False**. Ein Zugriff auf Eigenschaften wie **AvailableFreeSpace** oder **DriveFormat** ist somit nicht möglich, **Name** oder **DriveType** hingegen funktionieren. Um beim Zugriff auf alle Laufwerke Ausnahmen zu verhindern, können Sie prüfen, ob das

Laufwerk bereit ist und nur dann auf die Eigenschaften zugreifen, die bei nicht bereiten Laufwerken nicht zur Verfügung stehen. Dazu verwenden wir eine einfache **If...Then**-Bedingung:

```
For Each objDrive In objDrives
    Debug.print(objDrive.Name)
    Debug.Print("DriveType:          " & objDrive.DriveType.ToString())
    Debug.Print("IsReady:            " & objDrive.IsReady.ToString())
    Debug.Print("RootDirectory:      " & objDrive.RootDirectory.ToString())
    If (objDrive.IsReady) Then
        Debug.Print("AvailableFreeSpace: " & objDrive.AvailableFreeSpace.ToString())
        Debug.Print("DriveFormat:        " & objDrive.DriveFormat.ToString())
        Debug.Print("TotalFreeSpace:     " & objDrive.TotalFreeSpace.ToString())
        Debug.Print("TotalSize:          " & objDrive.TotalSize.ToString())
        Debug.Print("VolumeLabel:        " & objDrive.VolumeLabel.ToString())
    End If
Next objDrive
```

Praktischerweise geben Eigenschaften wie etwa **DriveFormat** nicht wie unter VBA nur die Zahlenwerte von Konstanten aus, sondern über die Funktion **ToString()** auch direkt entsprechende Zeichenketten. Benötigen Sie einmal den Zahlenwert, können Sie **ToString()** einfach weglassen.

Auf ein Verzeichnis zugreifen

Auf ein Verzeichnis greifen wir über das **DirectoryInfo**-Objekt zu. Dieses erstellen wir jeweils mit der **New**-Methode und übergeben der Konstruktor-Methode dabei den Namen des Verzeichnisses. Hier geben wir direkt die Eigenschaften als Kommentar hinter der jeweiligen Anweisung an:

```
Dim objDirectoryInfo As DirectoryInfo
objDirectoryInfo = New DirectoryInfo("c:\Beispiele")
Debug.Print (objDirectoryInfo.CreationTime.ToString)    '01.01.1601 01:00:00
Debug.Print (objDirectoryInfo.CreationTimeUtc)          '01.01.1601
Debug.Print (objDirectoryInfo.Extension)                '
Debug.Print (objDirectoryInfo.Exists)                   'False
Debug.Print (objDirectoryInfo.FullName)                 'c:\Beispiele
Debug.Print (objDirectoryInfo.Name)                     'Beispiele
Debug.Print (objDirectoryInfo.Root.FullName)            'c:\
```

Verzeichnisse eines Laufwerks lesen

Damit haben wir zwar schon einen Weg gefunden, auf ein Verzeichnis zuzugreifen, dessen Namen wir kennen. Aber wie können wir beispielsweise alle Verzeichnisse eines Laufwerks lesen und ausgeben? Es sieht nicht so aus, als ob das **DriveInfo**-Objekte Eigenschaften bereitstellt, die eine Auflistung aller untergeordneten Verzeichnisse liefert. Aber wir kommen dennoch zum Ziel, denn das **DirectoryInfo**-Objekt bietet eine Funktion namens **GetDirectories**. Und da ein Laufwerk ja prinzipiell auch ein

Verzeichnis abbilden, zum Beispiel c:\, können wir diese Funktion nutzen, um die Verzeichnisse eines Laufwerks zu ermitteln. Insgesamt sieht das etwa wie folgt aus:

```
Dim objDrive As DriveInfo
Dim objDirectoryDrive As DirectoryInfo
Dim objDirectory As DirectoryInfo
objDrive = New DriveInfo("c:")
objDirectoryDrive = New DirectoryInfo(objDrive.Name)
For Each objDirectory In objDirectoryDrive.GetDirectories
    Debug.Print(objDirectory.FullName)
Next objDirectory
```

Wir holen also das **DriveInfo**-Objekt und erstellen ein **DirectoryInfo**-Objekte mit dem Namen des **DriveInfo**-Objektes als Parameter. Dann durchlaufen wir diese Verzeichnisse in einer **For Each**-Schleife und geben mit der Eigenschaft **FullName** den Pfad der Verzeichnisse aus – zum Beispiel so:

```
c:\Dokumente und Einstellungen
c:\inetpub
c:\Intel
c:\MSOCache
...
```

Verzeichnisse suchen

Wir können der **GetDirectories**-Methode auch noch einen Suchbegriff als Parameter übergeben. Folgender Aufruf liefert alle Verzeichnisse, die mit **Windows** beginnen:

```
objDirectoryDrive.GetDirectories("Windows*")
```

Als Platzhalter können die von Access-SQL bekannten Platzhalter verwendet werden:

- Sternchen (*****): Platzhalter für beliebig viele beliebige Zeichen
- Fragezeichen (**?**): Platzhalter für ein beliebiges Zeichen

Und es gibt noch einen zweiten Parameter namens **searchOption**. Mit diesem können Sie festlegen, ob nur in der obersten Ebene gesucht werden soll oder rekursiv auch in den untergeordneten Verzeichnissen. Die Werte lauten:

- **AllDirectories** (**1**): Alle Verzeichnisse inklusive Unterverzeichnisse durchsuchen
- **TopDirectoryOnly** (**0**): Nur das aktuelle Verzeichnis durchsuchen

Wenn Sie den Parameterwert **AllDirectories** verwenden, kann es zu einem Fehler kommen, weil der Benutzer nicht ausreichende Berechtigungen für den Zugriff auf alle betroffenen Verzeichnisse hat. Der Aufruf für eine rekursive Suche sieht so aus:

```
objDirectoryDrive.GetDirectories("Windows*", searchOption.AllDirectories)
```

Besonderheit von DirectoryInfo: (Noch) nicht vorhandene Verzeichnisse

Das **DirectoryInfo**-Objekt hat eine Besonderheit, genauso wie das **DriveInfo**-Objekt und, wie wir weiter unten sehen werden, auch das **FileInfo**-Objekt. All diese Objekte können Sie mit der Angabe bestimmter Laufwerke, Verzeichnisse oder Dateien instanzieren, auch wenn die angegebenen Elemente noch gar nicht im Dateisystem vorhanden sind. Wir werden gleich als Beispiel für die Methoden von Verzeichnissen zeigen, wie Sie damit erst prüfen, ob ein Element bereits vorhanden ist und dieses gegebenenfalls erstellen können.

Methoden von Verzeichnissen

Das **DirectoryInfo**-Objekt als Repräsentant eines Verzeichnisses bietet eine ganze Reihe Methoden an, mit denen verschiedene Aufgaben rund um Verzeichnisse erledigt werden können – vom Erstellen über das Löschen und Verschieben bis hin zum Ermitteln der in einem Verzeichnis enthaltenen Daten:

- **Create**: Erstellt das im **DirectoryInfo** angegebene Verzeichnis.
- **CreateSubdirectory**: Erstellt ein Unterverzeichnis im Verzeichnis aus dem **DirectoryInfo**-Objekt.
- **Delete**: Mit der **Delete**-Methode löschen Sie das Verzeichnis, das im **DirectoryInfo**-Objekt angegeben ist.
- **EnumerateDirectories**: Liefert eine Auflistung der im referenzierten Verzeichnis enthaltenen Verzeichnisse als **IEnumerable**-Objekt.
- **EnumerateFiles**: Liefert eine Auflistung der im referenzierten Verzeichnis enthaltenen Dateien als **IEnumerable**-Objekt.
- **EnumerateFileSystemInfos**: Liefert eine Auflistung der im referenzierten Verzeichnis enthaltenen Verzeichnisse und Dateien gleichzeitig als **IEnumerable**-Objekt.
- **GetDirectories**: Liefert alle Unterverzeichnisse des referenzierten Verzeichnisses als **String**-Array.
- **GetFiles**: Liefert alle Dateien des referenzierten Verzeichnisses als **String**-Array.
- **GetFileSystemInfos**: Liefert alle Unterverzeichnisse und Dateien des referenzierten Verzeichnisses als **String**-Array.
- **MoveTo**: Verschiebt das referenzierte Verzeichnis in das als Parameter angegebene Verzeichnis.

Prüfen, ob ein Verzeichnis vorhanden ist

Ob ein Verzeichnis vorhanden ist, können wir mit der weiter oben bereits verwendeten Eigenschaft **Exists** des **DirecotryInfo**-Objekts prüfen:

```
If (objDirectoryInfo.Exists = True) Then
    Debug.Print(objDirectoryInfo.CreationTime.ToString)
    ...
Else
    Debug.Print("Das Verzeichnis existiert nicht.")
End If
```

Ein Verzeichnis anlegen

Um ein Verzeichnis anzulegen, erstellen Sie zunächst ein **DirectoryInfo**-Objekt mit dem Pfad des zu erstellenden Verzeichnisses als Parameter der Konstruktor-Methode. Dann prüfen wir mit der **Exists**-Funktion, ob das Verzeichnis schon vorhanden ist. Falls nicht, erstellen wir dieses mit der **Create**-Methode. In diesem Fall wird sich das Ergebnis der **Exists**-Funktion nicht automatisch ändern, nur weil das Verzeichnis nun vorhanden ist – wir müssen das **DirectoryInfo**-Objekt neu erstellen. Danach prüfen wir nochmal, ob das Verzeichnis nun existiert und geben eine entsprechende Meldung aus. Das Gleiche erledigen wir, wenn das Verzeichnis bereits vorhanden ist:

```
Dim objDirectory As DirectoryInfo
objDirectory = New DirectoryInfo("c:\Beispielverzeichnis")
If (Not objDirectory.Exists) Then
    objDirectory.Create
    objDirectory = New DirectoryInfo(objDirectory.FullName)
    If (objDirectory.Exists) Then
        Debug.Print("Das Verzeichnis " & objDirectory.FullName & " wurde erstellt.")
    End If
Else
    Debug.Print("Das Verzeichnis " & objDirectory.FullName & " existiert bereits.")
End If
```

Mehrere Verzeichnisse gleichzeitig anlegen

Wenn Sie etwa mit VBA eine Reihe verschachtelter Verzeichnisse anlegen wollten, mussten Sie diese Schritt für Schritt mit der **MkDir**-Methode erstellen und dabei auch noch jeweils prüfen, ob das Verzeichnis nicht schon vorhanden ist, weil es sonst eine Fehlermeldung gab. Hier kommt ein cooles Feature der **Create**-Methode: Sie können mehrere Verzeichnisse auf einmal angeben, die dann direkt erstellt werden. Wenn Laufwerk **C:** noch kein Verzeichnis namens **Beispielverzeichnis** enthält, können Sie mit folgenden Anweisungen direkt mehrere Verzeichnisse hinzufügen:

```
objDirectory = New DirectoryInfo("c:\Beispielverzeichnis\Test\Test\Test")
objDirectory.Create
```

Unterverzeichnis anlegen

Enthält **objDirectory** ein **DirectoryInfo**-Objekt zu einem Verzeichnis, dem Sie ein Unterverzeichnis hinzufügen wollen, können Sie dies auch mit der Methode **CreateSubdirectory** erledigen. Dieser übergeben Sie dann einfach die hinzuzufügenden Verzeichnisse als Parameter:

```
Dim objDirectory As DirectoryInfo
objDirectory = New DirectoryInfo("c:\")
objDirectory.CreateSubdirectory("Beispielverzeichnis\Test")
```

Ein Verzeichnis löschen

Die **Delete**-Methode eines **DirectoryInfo**-Objekts erlaubt das Löschen des referenzierten Verzeichnisses:

```
Dim objDirectory As DirectoryInfo
objDirectory = New DirectoryInfo("c:\Beispielverzeichnis")
objDirectory.Delete
```

Das gelingt so allerdings nur, wenn das Verzeichnis leer ist. Anderenfalls löst dies eine Ausnahme aus. Wenn Sie auch nicht leere Verzeichnisse löschen wollen, geben Sie zusätzlich den Wert **True** für den Parameter **Recursive** an:

```
objDirectory.Delete(True)
```

Damit wird das Verzeichnis ohne Rückfrage gelöscht.

Unterverzeichnisse eines Verzeichnisses durchlaufen

Um die **EnumerateDirectories**-Funktion auszuprobieren, legen wir zunächst einige Unterverzeichnisse in unserem Verzeichnis **c:\Beispielverzeichnis** an:

```
Dim i As Integer
Dim objDirectory As DirectoryInfo
Dim objSubdirectory As DirectoryInfo
For i = 1 To 10
    objDirectory = New DirectoryInfo("c:\Beispielverzeichnis\Test" & i)
    objDirectory.Create
Next i
```

Dann referenzieren wir das Verzeichnis **c:\Beispielverzeichnis** und durchlaufen in einer **For Each**-Schleife alle Elemente der durch die Funktion **EnumerateDirectories** gelieferten Liste:

```
objDirectory = New DirectoryInfo("c:\Beispielverzeichnis")
For Each objSubdirectory In objDirectory.EnumerateDirectories
    Debug.Print (objSubdirectory.FullName)
```

Textdateien mit Visual Basic

Der Namespace System.IO liefert eine ganze Reihe Befehle für den Umgang mit Laufwerken, Verzeichnissen und Dateien. Wenn es sich bei letzteren um Textdateien handelt, möchten Sie diese möglicherweise von Ihrer Anwendung aus einlesen oder Daten in die Datei schreiben. Vielleicht wollen Sie auch komplett neue Textdateien mit Texten aus einer Datenbank füllen. Dieser Artikel zeigt, wie Sie die Objekte, Methoden und Eigenschaften des System.IO-Namespaces nutzen, um lesend und schreibend auf Textdateien zuzugreifen.

Die Beispiele dieses Artikels haben wir mit dem Tool **LINQPad** durchgespielt, das die einfache Eingabe von Visual Basic- und C#-Anweisungen und deren Ausführung erlaubt. Sie müssen hier nicht immer erst das Projekt kompilieren, um die enthaltenen Anweisungen zu debuggen, sondern können dies quasi ad hoc erledigen (Download siehe **https://www.linqpad.net**). Den benötigten Namespace **System.IO** haben wir hinzugefügt, indem wir die Taste **F4** betätigt und im nun erscheinenden Dialog auf der Registerseite **Additional Namespace Imports** den Namespace **System.IO** eingetragen haben. Unter **Language** haben wir **VB Program** ausgewählt.

Methoden zum Lesen und Schreiben von Textdateien

Wer von VBA kommt, weiß, dass das Arbeiten mit Textdateien mit den Anweisungen Open, Close, Write und so weiter etwas kompliziert war. Das ändert sich in Visual Basic: Hier verschmelzen in den meisten Fällen die Anweisungen zum Öffnen, Lesen/Schreiben und Schließen in jeweils einen einzigen Befehl – je nach Anforderung.

Für den lesenden und schreibenden Zugriff gibt es die folgenden Methoden:

- **ReadAllLines**: Liest alle Zeilen einer Textdatei in ein String-Array ein.
- **ReadAllText**: Liest den kompletten Text einer Textdatei in eine String-Variable ein.
- **ReadLines**:
- **WriteAllLines**: Schreibt den Inhalt eines String-Arrays zeilenweise in eine Textdatei.
- **WriteAllText**: Schreibt den Inhalt einer String-Variablen vollständig in eine Textdatei.

Textdatei komplett in eine Variable einlesen

Wenn bereits eine Textdatei vorliegt, deren Text Sie einlesen möchten, gelingt das am schnellsten mit der **ReadAllText**-Methode der **File**-Klasse. Sie liefert einen String, den wir hier einer **String**-Variablen zuweisen:

```
Dim strText As String
Dim strFile As String
```

```
strFile = "c:\Beispielverzeichnis\Beispiel.txt"
strText = File.ReadAllText(strFile)
Debug.Print(strText)
```

Textdatei zeilenweise in ein Array einlesen

Die zweite Möglichkeit, eine Datei einzulesen, liefert die Zeilen der Textdatei als **String**-Array, wobei jede Zeile in einem eigenen Element gespeichert wird. Wir definieren also ein **String**-Array namens **strLines()** und füllen dieses mit der **ReadAllLines**-Methode, der wir den Pfad zur einzulesenden Datei übergeben. Dann durchlaufen wir die Elemente des Arrays und referenzieren das aktuelle Element jeweils mit der Variablen **strLine**, dessen Inhalt wir im Direktbereich ausgeben:

```
Dim strLine As String
Dim strFile As String
Dim strLines() As String
strFile = "c:\Beispielverzeichnis\Beispiel.txt"
strLines = File.ReadAllLines(strFile)
For Each strLine In strLines
    Debug.Print(strLine)
Next strLine
```

ReadLines

Wenn Sie nicht die ganze Datei auf einen Rutsch mit **ReadAllText** oder **ReadAllLines** einlesen möchten, sondern gegebenenfalls vor dem Einlesen einer Zeile prüfen wollen, ob Sie diese Zeile überhaupt benötigen, können Sie die Methode **ReadLines** verwenden. Diese verwendet man immer im Rahmen einer **For Each**-Schleife, wobei man die Datei im Kopf der **For Each**-Schleife öffnet und diese dann durchläuft. Im Detail sieht das wie folgt aus:

```
Dim strFile As String
Dim strLine As String
strFile = "c:\Beispielverzeichnis\Beispiel.txt"
For Each strLine In File.ReadLines(strFile)
    Debug.Print (strLine)
Next strLine
```

Der Vorteil beispielsweise gegenüber der Methode **ReadAllLines** ist, dass nicht die ganze Datei in den Speicher geladen wird, sondern jeweils nur eine Zeile. Auch wenn Sie beispielsweise nur eine bestimmte Zeile suchen, können Sie besser mit **ReadLines** arbeiten, da dieses die Operation dann nach Verlassen der **For Each**-Schleife beendet. Hier ein Beispiel, bei dem wir nach einer bestimmten Zeichenkette innerhalb einer Zeile suchen und dann die Schleife verlassen:

```
Dim strFile As String
Dim strLine As String
strFile = "c:\Beispielverzeichnis\Beispiel.txt"
For Each strLine In File.ReadLines(strFile)
```

Direkter SQL-Zugriff und SQL-Injection

Wenn Sie von Access kommen und es gewohnt sind, Abfragen mit der Entwurfsansicht über die Benutzeroberfläche anzulegen, kann schon das Zusammenstellen von SQL-Abfragen nervig sein. Aber diese kann man unter Access immerhin noch aus der SQL-Ansicht einer Abfrage ermitteln. Was aber, wenn man nun Abfragen in LINQ formulieren soll, wo es noch nicht einmal eine grafische Entwurfsansicht gibt? Nun: Es gibt auch unter Entity Framework die Möglichkeit, SQL-Befehle abzusetzen. Das ist auch deshalb interessant, weil Sie so auch gespeicherte Prozeduren ausführen können. Wie das gelingt, zeigt der vorliegende Artikel.

Wenn Sie in Entity Framework direkt mit einer SQL-Abfrage auf die Daten der zugrunde liegenden Tabellen zugreifen wollen, können Sie drei verschiedene Methoden nutzen:

- Jede **DbSet**-Auflistung bietet die Methode **SqlQuery** an, zum Beispiel: **Kunden.SqlQuery**
- Die Methode **SqlQuery** finden Sie auch für das Objekt **Database** der Kontextklasse, hier meist **dbContext** genannt: **dbContext.Database.SqlQuery**
- Die mit **dbContext** referenzierte Kontextklasse bietet über das Objekt **Database** außerdem noch die Methode **ExecuteSqlCommand** an: **dbContext.Database.ExecuteSqlCommand**

Die Methode SqlQuery der DbSet-Klasse

Wenn Sie die Methode **SqlQuery** einer der **DbSet**-Klassen des Entity Data Models aufrufen, muss die Abfrage als Ergebnis Entitäten des Typs der jeweiligen **DbSet**-Klasse liefern. Wenn Sie also etwa auf Daten der Tabelle **Kunden** zugreifen wollen, würde die folgende Methode funktionieren, die wir über eine Schaltfläche des Fensters **MainWindow** des Beispielprojekts aufrufen:

```
Private Sub btnKundenAbfragenDbSet_Click(sender As Object, e As RoutedEventArgs)
    Dim dbContext As BestellverwaltungContext
    Dim Kunden As List(Of Kunde)
    Dim Kunde As Kunde
    dbContext = New BestellverwaltungContext
    Kunden = dbContext.Kunden.SqlQuery("SELECT * FROM Kunden").ToList()
    For Each Kunde In Kunden
        Debug.Print(Kunde.ID.ToString() + " " + Kunde.Nachname)
    Next
End Sub
```

Hie definieren wir eine **List**-Variable **Kunden** für Elemente des Typs **Kunde** sowie eine **Kunde**-Variable gleichen Namens. Diese füllen wir dann mit dem Ergebnis der **SqlQuery**-Methode mit dem Parameter **SELECT * FROM Kunden**, das wir dann

mit **ToList()** noch in eine Liste konvertieren. Anschließend durchlaufen wir die Elemente in einer **For Each**-Schleife und geben die Werte der Felder **ID** und **Nachname** im Ausgabebereich aus.

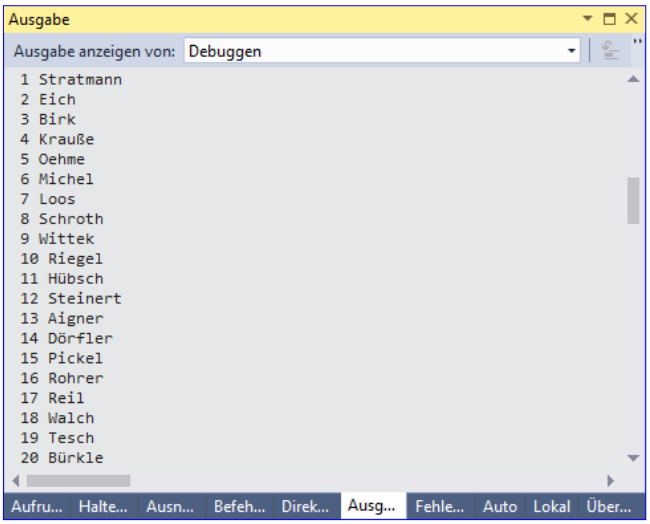


Bild 1: Ausgabe des Ergebnisses der **SqlQuery**-Methode

Abfragen mit INNER JOIN

Sie können auch weitere Tabellen zur Abfrage hinzufügen. Im folgenden Beispiel haben wir etwa die Anweisung mit der **SqlQuery**-Methode wie folgt geändert und die Tabelle **Anreden** per **INNER JOIN** hinzugefügt, um nach allen Kunden zu filtern, deren Anrede **Frau** lautet:

```
Kunden = dbContext.Kunden.SqlQuery("SELECT Kunden.* FROM
Kunden INNER JOIN Anreden ON Kunden.AnredeID = Anreden.ID
WHERE Anreden.Name = 'Frau'").ToList()
'Test
```

Das Ergebnis sehen Sie in Bild 1.

Wichtig ist nur, dass das Abfrageergebnis nur Elemente der Tabelle **Kunden** enthält. Wir probieren einmal die folgende SQL-Abfrage in der gleichen Methode aus, welche noch das Feld **Anrede** der Tabelle **Anreden** mit dem Abfrageergebnis zurückliefert:

```
SELECT Kunden.*, Anreden.Name FROM Kunden INNER JOIN Anreden ON Kunden.AnredeID = Anreden.ID WHERE Anreden.Name = 'Frau'
```

Dies liefert das gleiche Ergebnis und keinen Fehler, aber der Inhalt des Feldes **Name** der Tabelle **Anreden** ist dennoch nicht verfügbar, da wir ja über die Auflistung **Kunden** nur auf Elemente der Klasse **Kunde** zugreifen können. Aber sollte man hier nicht mit einem Fehler rechnen, da die **SELECT**-Abfrage auch Felder liefert, die so nicht in der Zielklasse **Kunde** vorkommen? Wir probieren es einmal aus und verwenden die folgende Abfrage, die neben den Feldern der Tabelle **Kunde** auch ein Feld der Tabelle **Produkte** enthält:

```
Kunden = dbContext.Kunden.SqlQuery("SELECT Kunden.*, Produkte.Name FROM (Kunden
    INNER JOIN Bestellungen ON Kunden.ID = Bestellungen.KundeID)
    INNER JOIN (Produkte INNER JOIN Bestellpositionen ON Produkte.ID =
    Bestellpositionen.ProduktID) ON Bestellungen.ID = Bestellpositionen.BestellungID").ToList()
```

Führen wir die Methode aus und schauen uns den Aufbau des Objekts **Kunde** wie in Bild 2 an, enthält dieses genau die in der Klasse definierten Elemente. Nun probieren wir auch noch den gegenteiligen Fall aus, indem wir nicht alle Elemente der Klasse **Kunde** mit Werten aus der zugrunde liegenden Tabelle **Kunden** füllen, sondern nur mit dem Feld **Kunden.ID**:

```
Kunden = dbContext.Kunden.SqlQuery("SELECT Kunden.ID, Produkte.Name FROM (Kunden
    INNER JOIN Bestellungen ON Kunden.ID = Bestellungen.KundeID)
```

```
INNER JOIN (Produkte INNER JOIN Bestellpositionen ON Produkte.ID =
Bestellpositionen.ProduktID) ON Bestellungen.ID = Bestellpositionen.BestellungID").ToList()
```

Dies sorgt dann allerdings für eine Fehlermeldung mit dem folgenden Text:

```
Der Datenleser ist mit dem angegebenen Wert für 'EDMSQL.Kunde' nicht kompatibel. Ein Element vom Typ ('Firma') weist keine entsprechende Spalte im gleichnamigen Datenleser auf.
```

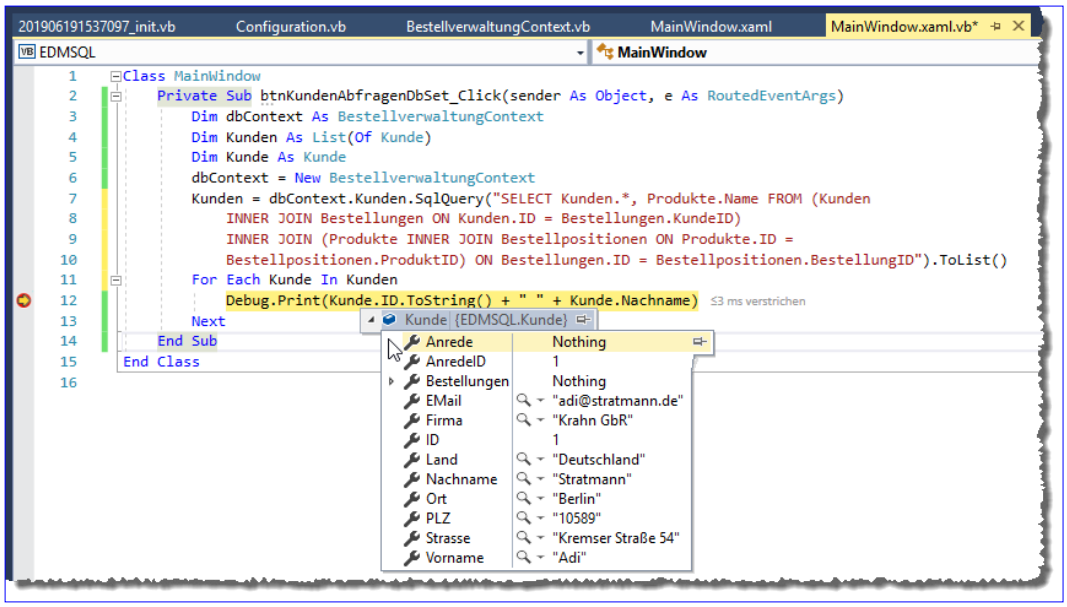


Bild 2: Die **Kunde**-Klasse enthält nur die definierten Elemente.

Die Abfrage muss also zwingend alle Eigenschaften der Zielklasse füllen, darf aber auch weitere Felder ausgeben – die dann allerdings nicht berücksichtigt werden.

Daten aus gespeicherten Prozeduren laden

Auf die gleiche Weise können Sie auch Daten aus gespeicherten Prozeduren laden. Dazu fügen wir der SQL Server-Datenbank, die in der Verbindungszeichenfolge genannt ist, eine gespeicherte Prozedur an. Das erledigen wir, in dem wir in Visual Studio den Bereich SQL Server-Objekt-Explorer einblenden (Menüpunkt **Ansicht|SQL Server-Objekt-Explorer**). Hier wählen Sie die Datenbank zu unserem Projekt aus und navigieren zum Eintrag **Gespeicherte Prozeduren**. Das Kontextmenü dieses Eintrags bietet den Befehl **Neue gespeicherte Prozedur hinzufügen...** an, den Sie nun aufrufen (siehe Bild 3).

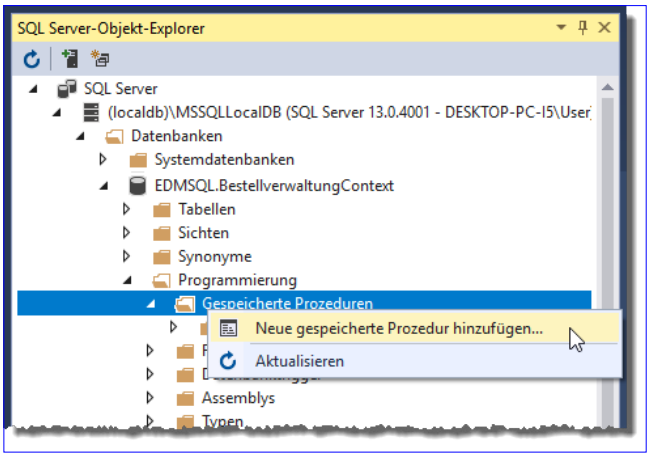


Bild 3: Anlegen einer neuen gespeicherten Prozedur

Im nun erscheinenden Bereich tragen wir den Code zum Erstellen der gespeicherten Prozedur ein und führen diesen mit **Strg + Umschalt + E** aus (im SQL Server Management Studio könnten Sie das mit **F5** erledigen, in Visual Studio jedoch startet **F5** das aktuelle Projekt):

```
CREATE PROCEDURE [dbo].[spAlleKunden]
AS
    SELECT Kunden.* FROM Kunden
RETURN 0
```

Nun fügen wir dem Fenster **MainWindow.xaml** eine neue Schaltfläche hinzu, welche die folgende Methode aufruft:

```
Private Sub btnKundenAbfragenStoredProcedure_Click(sender As Object, e As RoutedEventArgs)
    ...
    Kunden = dbContext.Kunden.SqlQuery("dbo.spAlleKunden").ToList()
    ...
End Sub
```

Damit füllen wir wieder eine Liste mit allen Datensätzen der Tabelle **Kunden** – nur über den Umweg einer gespeicherten Prozedur.

Gespeicherte Prozedur mit Parameter

Allerdings ist der Sinn einer gespeicherten Prozedur ja gerade, nicht den Inhalt einer kompletten Tabelle zurückzuliefern, sondern – unter anderem – die Daten schon auf dem Server zu filtern und nur die benötigten Daten auf den Clientrechner zu übertragen. Noch flexibler wird das Ganze natürlich, wenn Sie auch noch mit Parametern arbeiten, die beim Aufruf vom Client aus übergeben kann. Wir fügen also auch noch eine gespeicherte Prozedur hinzu, die einen Parameter erwartet. In diesem Fall soll die gespeicherte Prozedur nur den Kunden mit einer bestimmten ID zurückliefern:

```
CREATE PROCEDURE [dbo].[spKundeNachID]
    @id int
AS
    SELECT Kunden.* FROM Kunden WHERE ID = @id
RETURN 0
```

Wie übergeben wir den Wert für den Parameter beim Aufruf mit der **SqlExecute**-Methode? Wir versuchen es erst, indem wir einfach den Wert für den Parameter anhängen:

```
Kunden = dbContext.Kunden.SqlQuery("dbo.spKundeNachID", 10).ToList()
```

Dies liefert den Fehler **Procedure or function 'spKundeNachID' expects parameter '@id', which was not supplied.** Also korrigieren wir den Aufruf der Abfrage wie folgt und fügen den Namen des Parameters **@id** hinten an den Namen der gespeicherten Prozedur an:

```
Kunden = dbContext.Kunden.SqlQuery("dbo.spKundeNachID @id", 10).ToList()
```

Dies liefert wiederum den Fehler **Must declare the scalar variable "@id"**. Wo aber liegt hier der Fehler? Die Variable wird ja wie oben angegeben in der Definition der gespeicherten Prozedur deklariert.

Letztendlich funktioniert die folgende Variante:

```
Kunden = dbContext.Kunden.SqlQuery("dbo.spKundeNachID 10").ToList()
```

Wenn Sie also den Wert des Parameters zuvor ermitteln und in einer Variablen speichern, können Sie diesen wie folgt zum Parameter von **SQLQuery** hinzufügen:

```
Dim ID As Int16
ID = 10
Kunden = dbContext.Kunden.SqlQuery("dbo.spKundeNachID " + ID.ToString()).ToList()
```

Das ist allerdings nur ein Workaround. Korrekt funktioniert es so:

```
Kunden = dbContext.Kunden.SqlQuery("dbo.spKundeNachID @id",
                                   New SqlParameter("id", 10)
).ToList()
```

Wir verwenden also in der Parameterliste nicht die Parameter selbst als Parameter, sondern für jeden Parameter ein Objekt der Klasse **SqlParameter**, das den Namen und den Wert des Parameters enthält. Wenn die gespeicherte Prozedur mehr als einen Parameter enthalten sollte, sieht die Syntax des Aufrufs wie folgt aus:

```
Kunden = dbContext.Kunden.SqlQuery("dbo.spBeispiel @p1 @p2 @p3 ...",
                                   New SqlParameter("p1", 1),
                                   New SqlParameter("p2", 2),
                                   New SqlParameter("p3", 3),
                                   ...
).ToList()
```

Wichtig ist auch noch, dass Sie den folgenden Namespace angeben müssen, um die **SqlParameter**-Klasse zu nutzen:

```
Imports System.Data.SqlClient
```

Gespeicherte Prozeduren mit Datum als Parameter

Wenn die gespeicherte Prozedur ein Datum als Parameter verwendet, sind ein paar Besonderheiten zu beachten. Wir erstellen eine gespeicherte Prozedur namens **spBestellungenNachBestelldatum** mit dem Parameter **@bestelldatum** des Datentyps **date** (dies wieder im Abfragefenster des SQL Server-Objekt-Explorers von Visual Studio durch Aufrufen von **Strg + Umschalt + E**):

```
CREATE PROCEDURE [dbo].[spBestellungenNachBestelldatum]
    @bestelldatum date
AS
    SELECT Bestellungen.* FROM Bestellungen WHERE Bestelldatum = @bestelldatum
RETURN 0
```

Im gleichen Abfragefenster können wir die Abfrage mit der folgenden Abfrage ausführen, die wir dazu markieren und mit **Strg + Umschalt + E** starten:

Visual Studio erweitern: Menübefehle

Visual Studio bietet die Möglichkeit der Erweiterung durch verschiedene Elemente wie etwa Add-Ins, Templates oder Packages. Damit können Sie den Funktionsumfang von Visual Studio nach Ihren eigenen Wünschen erweitern. Ein sehr oft genutztes Beispiel ist etwa das Hinzufügen von Menübefehlen. Sie können neue Befehle zu den bestehenden Menüs des Hauptmenüs von Visual Studio hinzufügen oder auch Kontextmenüs erweitern. Sogar neue Menüs lassen sich zum Hauptmenü hinzufügen. Wie das geht und wie Sie dann auf Visual Studio zugreifen, zeigen wir Ihnen in diesem Artikel.

Extensionsentwicklung hinzufügen

Der erste Schritt hin zum ersten eigenen Add-In ist, sofern noch nicht vorhanden, das Hinzufügen des Toolsets **Visual Studio Extensionentwicklung** (siehe Bild 1). Zu diesem Fenster gelangen Sie, wenn Sie im Dialog **Neues Projekt** links auf den Link **Visual Studio Installer öffnen** klicken.

Anschließend finden Sie im Dialog **Neues Projekt** unter den gängigen Sprachen wie **Visual C#** oder **Visual Basic** die Kategorie **Extensibility** mit den folgenden drei Einträgen (siehe Bild 2):

- **Visual Basic Item Template**: Mit dieser Vorlage erstellen Sie Vorlagen für Visual-Studio-Elemente, die Sie über den Dialog **Neues Element** hinzufügen können. Wie Sie eine solche Vorlage erstellen, zeigen wir im Artikel **Visual Studio erweitern: Item Templates**.

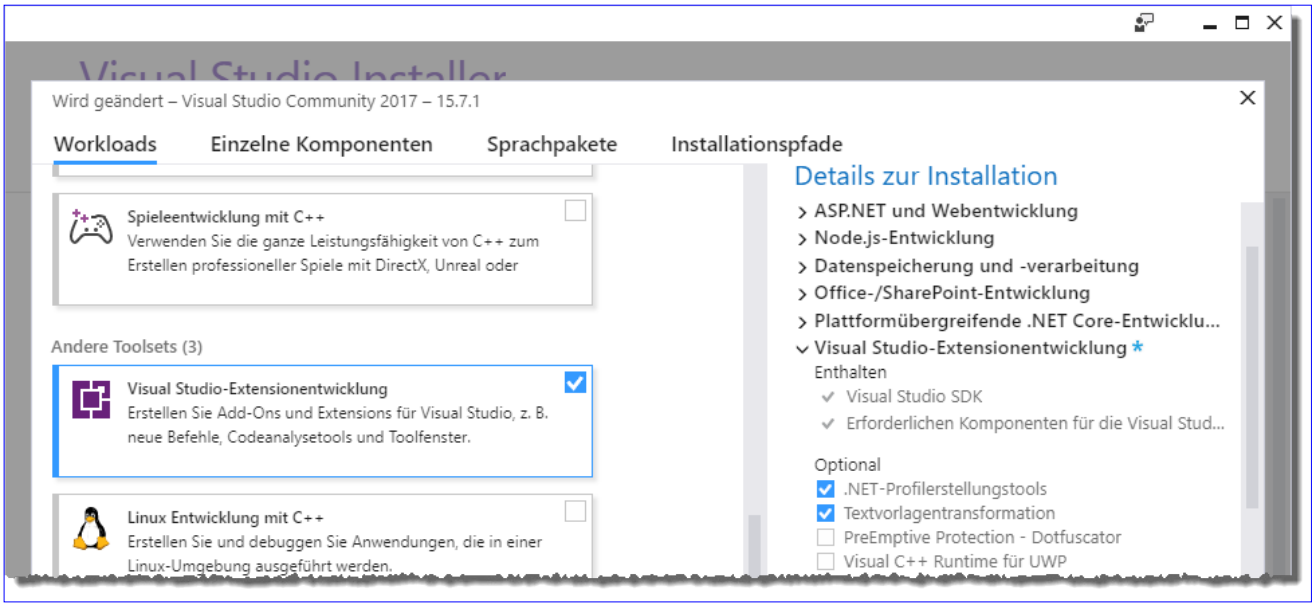


Bild 1: Erweitern von Visual Studio um die Extensionsentwicklung

- **Visual Basic Project Template**: Mit dieser Vorlage erstellen Sie komplette Projektvorlagen. Die bestehenden Vorlagen decken soweit alle Bedürfnisse ab, sodass wir diesen Vorlagentyp nicht in einem eigenen Artikel vorstellen werden.

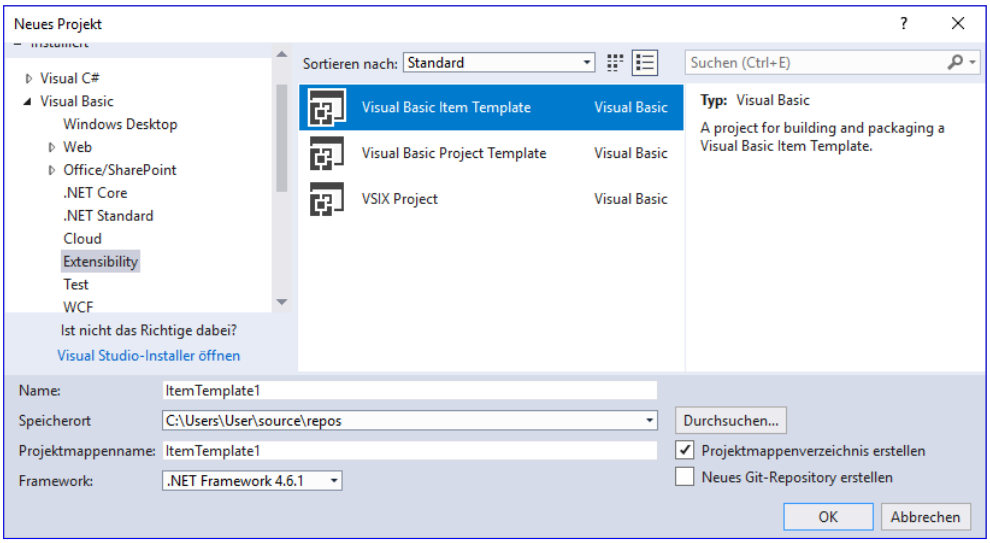


Bild 2: Verschiedene Vorlagen für das Entwickeln von Erweiterungen

- **VSIX Project**: Das sind Projekte, die Funktionserweiterungen enthalten und die in eine **.vsix**-Datei verpackt werden. Auf diese Weise können Sie die Erweiterungen leicht an andere Entwickler weitergeben. Wie Sie ein solches Projekt anlegen, um Visual Studio Menübefehle hinzuzufügen, über die Sie dann eigene Funktionen aufrufen können, zeigen wir im vorliegenden Artikel.

VSIX-Projekte

Ein VSIX-Projekt ist ein Projekt, mit dem Sie Visual Studio erweitern können, und zwar um ein oder mehrere Funktionen, die der Benutzer dann auf verschiedene Arten aufrufen kann. Typische Beispiele sind Schaltflächen, die Sie zu den Menüs von Visual Studio hinzufügen und die jederzeit aufgerufen werden können. Oder Sie erstellen Kontextmenü-Einträge, die Sie beispielsweise dem Projekt-Element im Projektmappen-Explorer oder Elementen wie einer Klasse oder einem Ordner eines Projekts zuweisen. Sie können auch sogenannte Tool Windows hinzufügen, mit denen Sie verschiedene Aufgaben erfüllen. Beispiele für Tool Windows sind die Toolbox, der Projektmappen-Explorer oder der Eigenschaften-Dialog. Wie Sie ein Tool Window erzeugen, lesen Sie Artikel **Visual Studio erweitern: Tool Windows**.

In VSIX-Projekten können Sie noch mehr Erweiterungen zu Visual Studio hinzufügen, beispielsweise Markierungen von Code im Code-Editor.

VSIX-Projekt anlegen

Wir legen wie in der obigen Abbildung ein neues Objekt auf Basis der Vorlage **VSIX Projekt** namens **VSIXBeispiel** an. Dies erstellt ein neues Visual Studio-Projekt, das im Projektmappen-Explorer zunächst nur drei Elemente enthält: **index.html**, **source.extension.vsixmanifest** und **stylesteet.css**. Wer schon einmal eine Internetseite programmiert hat, kennt die Dateiendungen **.html** und **.css** bereits. Bedeutet dies, dass die Benutzeroberfläche von VSIX-Erweiterungen auf HTML basiert? Nein: Die **.html**-Datei und die **.css**-Datei liefern nur den Inhalt für die **Getting started**-Seite, die nach dem Erstellen des Projekts erscheint. Sie können also beide Dateien löschen.

Und wozu dient die verbleibende Datei **source.extension.vsixmanifest**? Sie enthält die Eigenschaften des Projekts beziehungsweise der zu erstellenden Erweiterung. Diese erscheinen nach einem Doppelklick auf den Eintrag wie in Bild 4. Hier

können Sie beispielsweise den Produktnamen und die Beschreibung anpassen.

Einfachen Menübefehl hinzufügen

Den Ausgangspunkt einer Erweiterung bildet meist ein Element der Benutzeroberfläche wie etwa ein Menü-Eintrag. Über diesen rufen Sie dann entweder direkt die gewünschte Funktion auf oder zeigen eine Benutzeroberfläche an, mit dem Sie die gewünschte Funktion konfigurieren und starten können. Hier wollen auch wir beginnen und zunächst einmal einen Menübefehl zu Visual Studio hinzufügen.

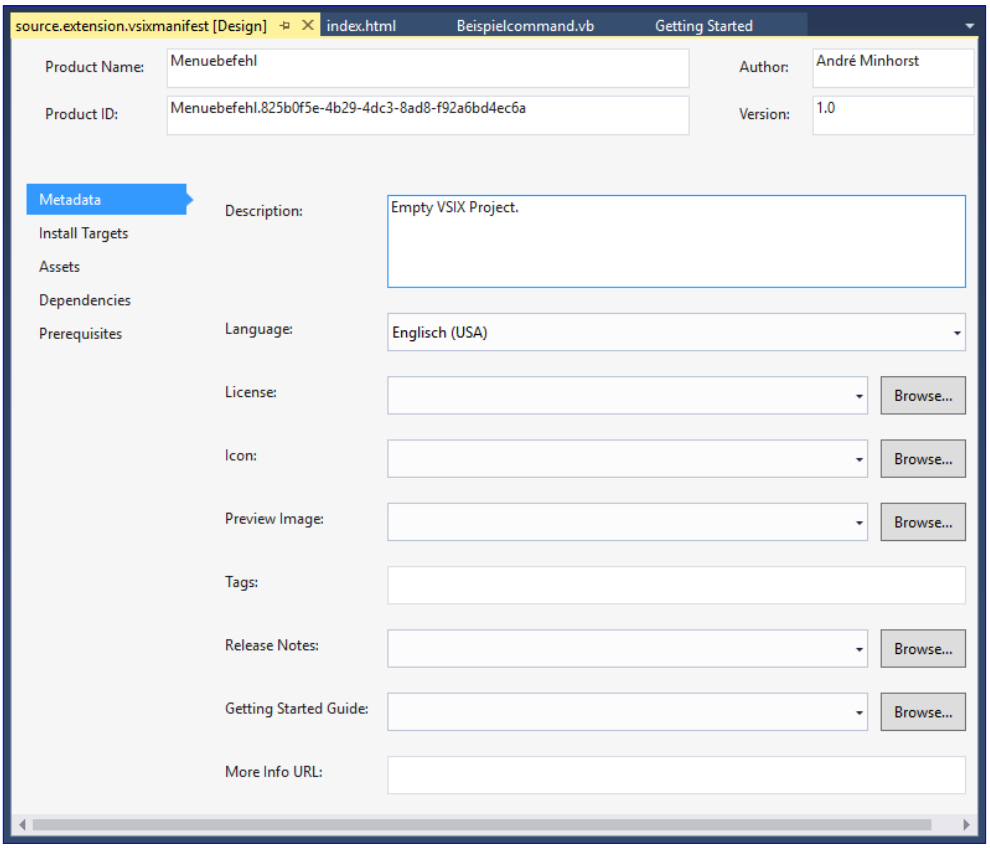


Bild 4: Eigenschaften der Erweiterung

Dazu rufen wir den Dialog **Neues Element hinzufügen** auf, der uns im Bereich **Extensibility** verschiedene neue Elemente anbietet (siehe Bild 3). Wir fügen ein Element des Typs **Custom Command** zu unserem Projekt hinzu und nennen es **Beispielcommand.vb**. Dies erstellt nach einer Weile eine neue Visual Basic-Klasse in unserem Projekt, aber nicht nur das: Wir finden auch eine ganze Reihe weiterer neuer Elemente im Projektmappen-Explorer. Es gibt einen Ordner namens **Resources**, in dem sich eine **.png**- und eine **.ico**-Datei befinden, zwei **.vb**-Dateien, einige Packages und vieles mehr.

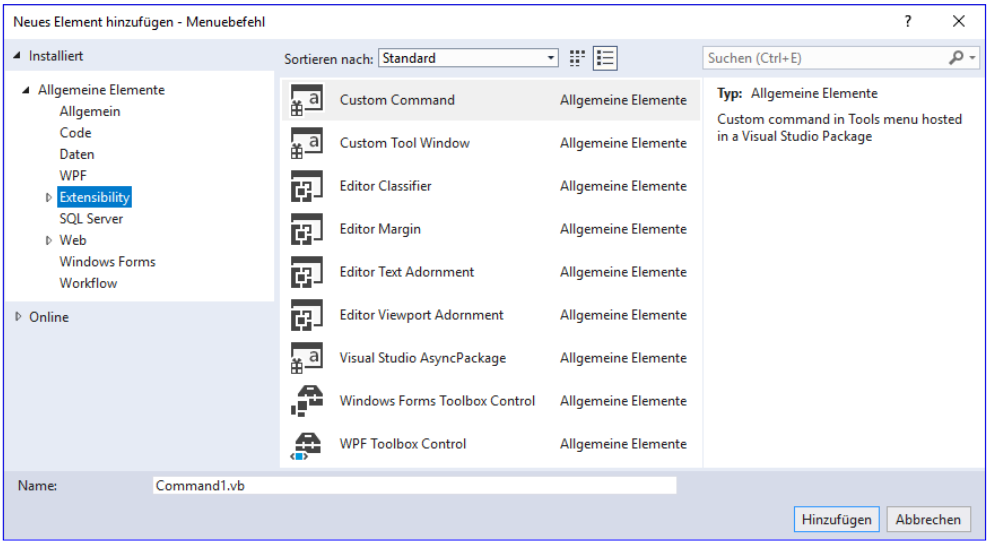


Bild 3: Elemente für Extensibility-Projekte

Bevor wir Anpassungen vornehmen, starten wir das Projekt einfach einmal, was

dazu führt, dass eine neue Instanz von Visual Studio gestartet wird. Offensichtlich handelt es sich aber nicht nur um eine neue Instanz, denn Visual Studio fragt die kompletten Informationen ab, die sonst nur nach einer Neuinstallation ermittelt werden. Wir übernehmen hier einfach die allgemeinen Einstellungen. Danach wird diese Instanz von Visual Studio vollständig gestartet. Hier finden wir allerdings keine Anzeichen einer Erweiterung – auch nicht, wenn wir ein neues Projekt anlegen.

Konfiguration der Erweiterung

An dieser Stelle ist es Zeit, einen genaueren Blick in die Datei **BeispielcommandPackage.vsct** zu werfen. Vielleicht finden wir hier einen Hinweis, wo unser Befehl zu finden ist. Bei dieser Datei handelt es sich um eine XML-Datei mit der Definition der Erweiterung und mit dem Element **CommandTable** als Stammelement:

```
<?xml version="1.0" encoding="utf-8"?>
<CommandTable xmlns="http://schemas.microsoft.com/VisualStudio/2005-10-18/CommandTable" xmlns:xs="http://www.w3.org/2001/
XMLSchema">
```

Etwas weiter unten finden wir dann die für uns wichtigen Elemente. Diese befinden sich alle unterhalb des **Commands**-Elements. Das erste ist die Auflistung **Groups** mit zunächst einem **Group**-Element:

```
  <Commands package="guidBeispielcommandPackage">
    <Groups>
      <Group guid="guidBeispielcommandPackageCmdSet" id="MyMenuGroup" priority="0x0600">
        <Parent guid="guidSHLMainMenu" id="IDM_VS_MENU_TOOLS"/>
      </Group>
    </Groups>
```

Das **Group**-Element definiert mit **guid** eine eindeutige Kennzeichnung und mit **id** den Namen des Elements, hier **MyMenuGroup**. Das Attribut gibt die Position der Gruppe unter den bereits vorhandenen Gruppen an, hier **0x0600** für die sechste Position. Darunter folgt das **Parent**-Element, welches angibt, in welches Menü die anzulegende Gruppe erstellt werden soll. In der Vorlage finden wir den Wert **IDM_VS_MENU_TOOLS**. Das entspricht dem Menü **Tools**, zu deutsch **Extras**, in Visual Studio. Hier ist eine Übersicht der möglichen Menüs:

- **IDM_VS_MENU_FILE**: Datei
- **IDM_VS_MENU_EDIT**: Bearbeiten
- **IDM_VS_MENU_VIEW**: Ansicht
- **IDM_VS_MENU_REFACTORING**: Umgestalten
- **IDM_VS_MENU_PROJECT**: Projekt
- **IDM_VS_MENU_BUILD**: Build

- **IDM_VS_MENU_FORMAT**: Format
- **IDM_VS_MENU_TOOLS**: Tools
- **IDM_VS_MENU_EXTENSIONS**: Erweiterungen
- **IDM_VS_MENU_WINDOW**: Fenster
- **IDM_VS_MENU_ADDINS**: Add-Ins
- **IDM_VS_MENU_COMMUNITY**: Community
- **IDM_VS_MENU_HELP**: Help

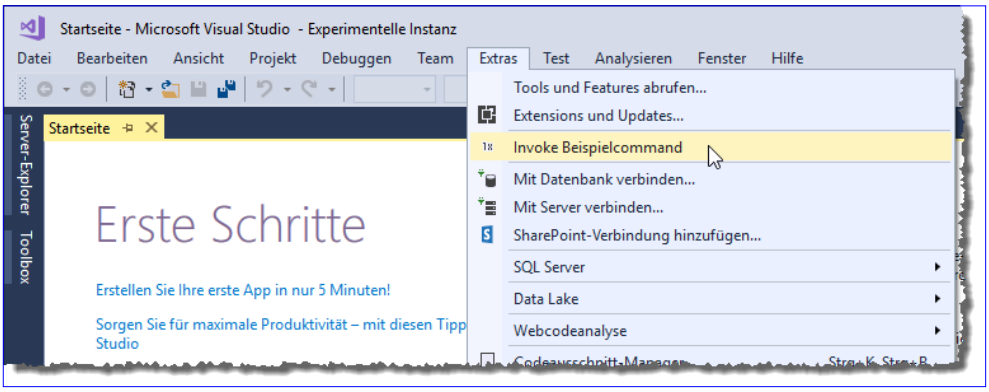


Bild 5: Unser Menü-Eintrag

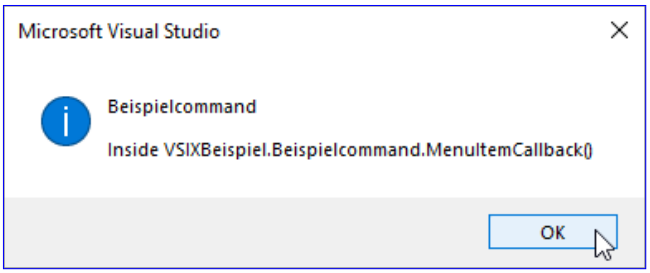


Bild 6: Durch den Menü-Eintrag angezeigte Meldung

Anhand des Wertes in unserer Extension können wir bereits ablesen, dass sich unser Befehl im Menü **Extras** befindet. Das können wir auch schnell bestätigen, wenn wir dieses Menü in der zum Debuggen gestarteten Instanz von Visual Studio öffnen. Dort finden wir direkt in der zweiten Gruppe den Eintrag (siehe Bild 5).

Klicken wir diesen Eintrag an, erscheint die Meldung aus Bild 6. Wie ist das alles in der **.vsct**-Datei abgebildet und welche Einstellungen können wir selbst für unsere Zwecke anpassen – neben dem Menü, in dem der Befehl eingefügt wird?

Dazu schauen wir uns die nächste Abteilung in dieser Datei an. Diese heißt **Buttons** und sollte dementsprechend Informationen über die Schaltflächen der Erweiterung aufweisen. In der **Buttons**-Auflistung finden wir in unserem Beispiel ein **Button**-Element mit den bereits bekannten Attributen **guid**, **id** und **priority**. Neu ist die Eigenschaft **type**, mit welcher der **Typ** auf **Button** eingestellt wird.

```
<Buttons>
  <Button guid="guidBeispielcommandPackageCmdSet" id="BeispielcommandId" priority="0x0100" type="Button">
```

Im **Button**-Element finden wir weitere Elemente. Das erste ist wieder das **Parent**-Element, welches angibt, unter welchem Element die Schaltfläche angeordnet wird – in diesem Fall in unserer Menü-Gruppe, die wir weiter oben definiert haben:

```
    <Parent guid="guidBeispielcommandPackageCmdSet" id="MyMenuGroup" />
```

Danach folgen das **Icon**-Element zur Angabe eines Icons sowie das **Strings**-Element mit der Schaltflächenbeschriftung in einem weiterem Unterelement namens **ButtonText**:

Visual Studio erweitern: Elemente hinzufügen

Wer Visual Studio erweitern möchte, muss nicht nur die verschiedenen Wege kennen, wir man benutzerdefinierten Code in Visual Studio verfügbar machen kann. Wir müssen uns auch mit dem Objektmodell beschäftigen, dass es uns erlaubt, etwa Elemente zu einem Projekt hinzuzufügen oder bestehende Elemente um Code zu erweitern. Damit legen wir dann die Grundlage für viele Anwendungsfälle – etwa, um von Visual Studio aus ein Entity Data Model auf Basis des Datenmodells einer Access-Datenbank zu erstellen. Im vorliegenden Artikel zeigen wir jedoch erst einmal, wie Sie per DTE-Objektmodell neue Elemente zu einem Visual Basic-Projekt hinzufügen.

Ein Element hinzufügen

Ein Anwendungsfall könnte sein, dass Sie ein komplett neues Element zu einem Projekt hinzufügen wollen oder dass Sie bereits über eine entsprechende Datei verfügen und diese zum Projekt hinzufügen wollen. Wir erledigen das alles über die Kontextmenü-Einträge des Projektmappen-Explorers, wie wir sie im Artikel **Visual Studio erweitern: Menübefehle** angelegt haben.

In diesem Fall fügen wir den Befehl zum Kontextmenü des Projekt-Elements zu. Wir beschreiben das hier einmal, bei den folgenden Funktionen geben wir nur noch die wichtigsten Informationen zu den neuen Befehlen an. Wir fügen unserer VSIX-Lösung (wie Sie diese erstellen, erfahren Sie im oben genannten Artikel) ein neues Element des Typs **ExtensibilityICustomCommand** hinzu und nennen dieses **NeueDateiHinzufuegen.vb**. Wenn wir das tun, findet Visual Studio automatisch eine eventuell bereits vorhandene Package-Klasse und **.vsct**-Datei und fügt letzterer die Konfiguration des neuen Elements hinzu.

In der **.vsct**-Datei ändern wir das Element **CommandsIGroupsIGroup** für das neue Element wie folgt, wir ändern also das Attribut **id** im **Parent**-Element in **IDM_VS_CTXT_PROJNODE**:

```
<Group guid="guidBeispielcommandPackageCmdSet1"
id="MyMenuGroup" priority="0x0600">
  <Parent guid="guidSHLMainMenu"
    id="IDM_VS_CTXT_PROJECTNODE" />
</Group>
```

So erscheint der neue Befehl als Kontextmenü-Eintrag des Projekt-Elements im Projektmappen-Explorer. Außerdem ändern wir die Beschriftung des Befehls in der gleichen Datei für das Unterelement **ButtonText** des **Button**-Elements wie folgt:

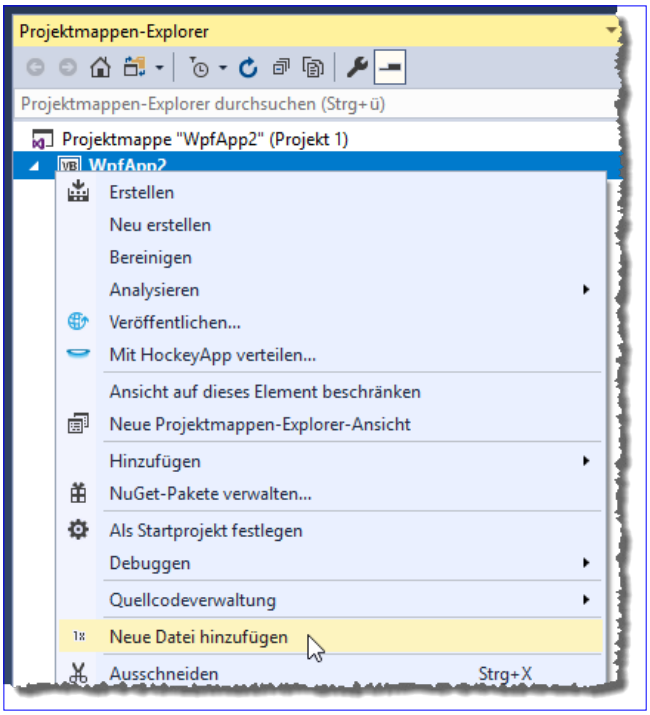


Bild 1: Neuer Kontextmenü-Eintrag

```
<ButtonText>Neue Datei hinzufügen</ButtonText>
```

Damit erhalten wir, wenn wir die Anwendung zum Debuggen starten, in der nun erscheinenden Visual Studio-Instanz mit dem Titel **Experimentelle Instanz** einen neuen Kontextmenü-Eintrag wie in Bild 1. Bevor wir mit der Programmierung der Methode **Execute** beginnen, fügen wir der Klasse **NeueDateiEinfuegen.vb** die beiden folgenden Namespaces hinzu:

```
Imports EnvDTE
Imports EnvDTE80
```

Neues Element zum Projekt hinzufügen

Der Befehl, um ein komplett neues Element zu einem Projekt hinzuzufügen, lautet **AddNewItem** und gehört zur Auflistung **ItemOperations**, die wiederum direkt dem **DTE**-Objekt untergeordnet ist. Um das **DTE**-Objekt zu referenzieren, fügen wir dem Kopf der Methode **Execute**, die beim Betätigen des Menübefehls ausgelöst wird, das Schlüsselwort **Async** hinzu:

```
Private Async Sub Execute(sender As Object, e As EventArgs)
    ThreadHelper.ThrowIfNotOnUIThread()
```

Dann deklarieren wir die zunächst benötigten Variablen, füllen **objDTE** mit einem Verweis auf das **DTE**-Objekt und prüfen, ob es nicht leer ist:

```
    Dim objItemOperations As ItemOperations
    Dim objDTE As DTE2
    objDTE = TryCast(Await package.GetServiceAsync(GetType(DTE)).ConfigureAwait(False), DTE2)
    If objDTE IsNot Nothing Then
```

Dann fügen wir der Variablen **objItemOperations** das Objekt **ItemOperations** zu und rufen dessen Methode **AddNewItem** auf – hier noch mit einem Platzhalter:

```
        objItemOperations = objDTE.ItemOperations
        objItemOperations.AddNewItem(...)
    End If
End Sub
```

Für die drei Punkte (...) im Aufruf der Methode **AddNewItem** geben wir den Typ des hinzuzufügenden Elements an. Dieser ist nicht so leicht zu ermitteln, denn es handelt sich um eine Zeichenkette, welche die Elemente des Dialogs **Neues Element hinzufügen** enthält.

Standardmäßig schlägt IntelliSense hier in der englischen Version **General/Textfile** vor (siehe Bild 2). Damit wollen wir im ersten Versuch auch arbeiten und geben zusätzlich noch den Namen **Neue Textdatei** an:

```
objItemOperations.AddNewItem("General\Text File", "Neue Textdatei.txt")
```

Visual Studio mit LINQPad: Project und ProjectItems

In den Artikeln der Reihe Visual Studio erweitern greifen wir von Visual Studio-Erweiterungen aus auf Elemente von Visual Studio zu. Leider ist das Debuggen dieser Erweiterungen äußerst mühsam. Zum Debuggen müssen Sie nämlich immer erst eine weitere Visual Studio-Instanz erstellen, welche dann die Erweiterungen enthält und zum Testen bereitstellt. Fällt eine Änderung an, müssen Sie das Debuggen beenden, die Änderungen einarbeiten und erneut starten. Da wäre es doch praktisch, wenn man einfach nur eine Instanz von Visual Studio öffnen und die darauf zugreifenden Methoden direkt testen könnte. Genau das realisieren wir in diesem Artikel, wo wir das Tool LINQPad nutzen. Dieses haben wir bereits in früheren Artikeln verwendet.

Wer von Access und dem VBA-Editor kommt, findet das Debuggen unter Visual Studio bisweilen anstrengend. Im Visual Basic Editor war das alles viel einfacher: Man konnte einfach eine Prozedur in ein Standardmodul schreiben, die Einfügemarke in der Prozedur platzieren und diese mit der Taste **F5** starten. Damit konnte man sogar während der Entwicklung auf Elemente des Visual Studio Editors selbst zugreifen! Und genau das wollen wir mit Visual Studio auch tun, um uns die Objekte, Methoden und Eigenschaften, die für die Programmierung von Visual Studio notwendig sind, besser kennenzulernen und schnell ausprobieren zu können.

Eine Möglichkeit dazu finden wir über das Tool LINQPad. Dieses erlaubt die Eingabe von Prozeduren und die direkte Ausführung mit der Taste **F5** – genau so, wie Sie es auch von Access und dem VBA-Editor kennen.

LINQPad können Sie in einer kostenlosen Version unter **https://www.linqpad.net/** herunterladen. Wenn Sie damit einfache VB-Prozeduren ausführen wollen, wechseln Sie im Auswahlfeld **Language** auf **VB Program**. Dadurch wird direkt eine leere Prozedur namens **Main** angelegt, die wir mit einer einfachen **Debug.Print**-Anweisung füllen. Diese ist übrigens automatisch verfügbar, ohne dass Sie zuvor einen Verweis auf den Namespace **System.Diagnostics** hinzufügen müssen. Wenn Sie dann die Einfügemarke in der Prozedur platzieren und auf die Start-Schaltfläche klicken oder die Taste **F5** betätigen, führt LINQPad die Prozedur aus und das Ergebnis landet im **Results**-Bereich von LINQPad (siehe Bild 1).

Visual Studio fernsteuern

Nun wollen wir von LINQPad aus die aktuelle Visual Studio-Instanz fernsteuern. Dazu benötigen wir zunächst zwei Verweise.

Diese fügen wir über einen Dialog hinzu, den wir mit der Taste **F4** öffnen. Der nun erscheinende Dialog **Query Properties** enthält eine Schaltfläche namens **Add...**, mit der wir einen weiteren Dialog namens **Add Custom Assembly References** anzeigen.

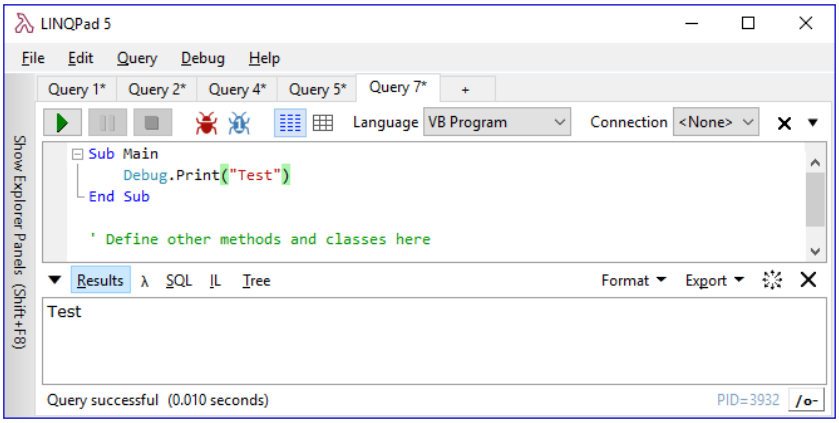


Bild 1: Einfache Eingabe von Prozeduren

Hier können Sie die angezeigte Liste aller verfügbaren Verweise schnell durch die Eingabe der Buchstaben **env** filtern, sodass nur noch die Einträge aus Bild 2 erscheinen. Hier wählen Sie die beiden Einträge **EnvDTE.dll** und **EnvDTE80.dll** aus und schließen den Dialog mit der **OK**-Schaltfläche.

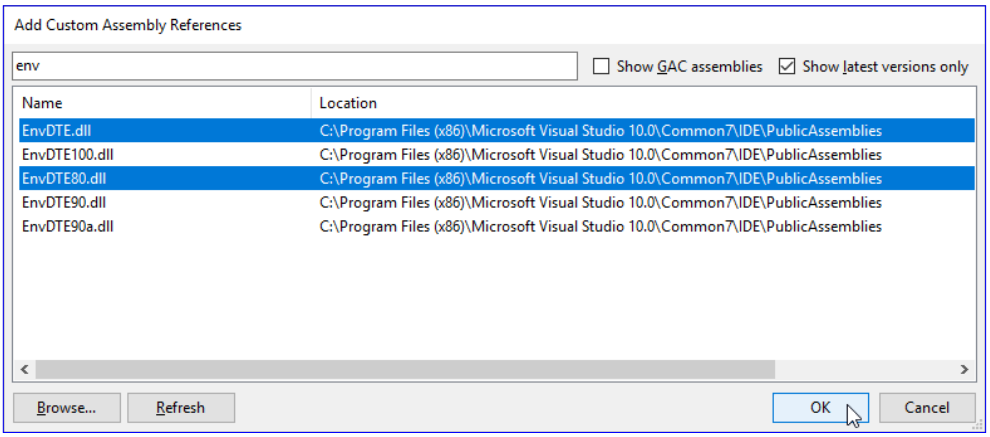


Bild 2: Hinzufügen von Verweisen

Danach zeigt der Dialog **Query Properties** die hinzugefügten Verweise an (siehe Bild 3). Damit können Sie auch diesen Dialog schließen.

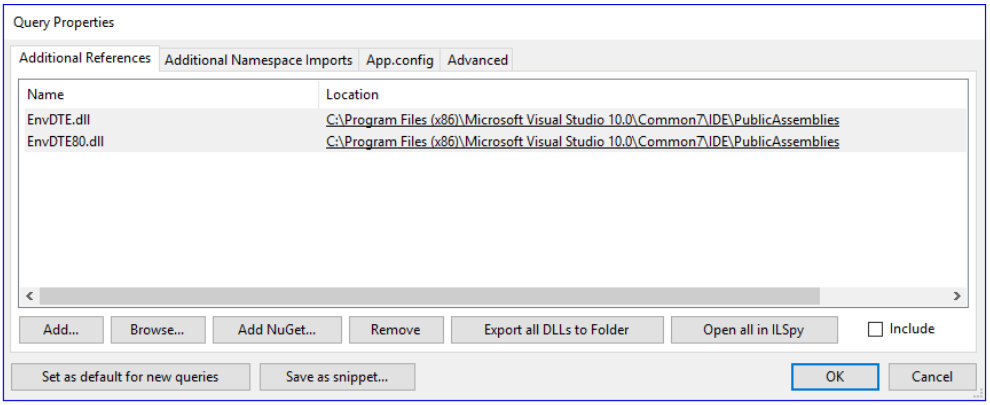


Bild 3: Neu hinzugefügte Verweise

Referenzieren des Visual Studio-Objekts

Das oberste Element der Bibliothek zum Arbeiten mit Visual Studio ist das **DTE2**-Objekt.

Dieses deklarieren wir in unserer Prozedur wie folgt:

```
Dim objDTE As EnvDTE80.DTE2
```

Dann referenzieren wir es über die Methode **GetActiveObject**:

```
objDTE = System.Runtime.InteropServices.
Marshal.GetActiveObject("VisualStudio.
DTE.15.0")
```

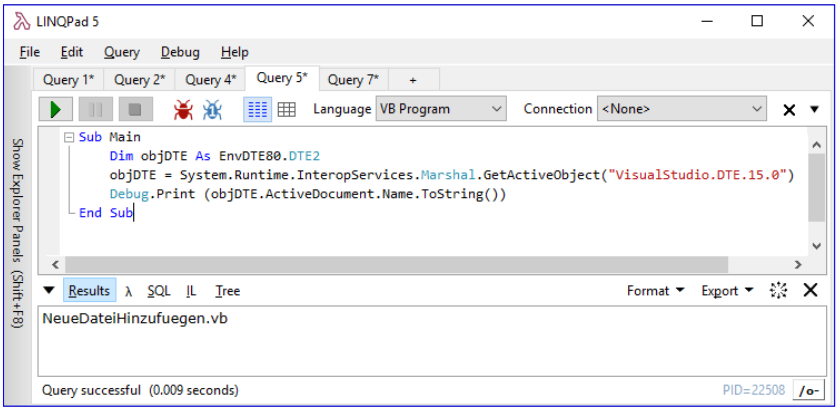


Bild 4: Zugriff auf Visual Studio

Hier ist zu beachten, dass Sie die richtige Version angeben – hier etwa **15.0** für Visual Studio 2017. Danach können wir mit der folgenden Anweisung etwa das aktuell angezeigte Dokument in Visual Studio ausgeben lassen:

```
Debug.Print (objDTE.ActiveDocument.Name.ToString())
```

Das Ergebnis sieht schließlich wie in Bild 4 aus. Gemessen daran, wie lange es dauert, einen Zugriff auf Elemente der Benutzeroberfläche von Visual Studio auszuprobieren, wenn wir diese innerhalb einer Visual Studio-Erweiterung testen, geschieht dies außerdem rasend schnell. Damit haben wir also ein Werkzeug, um uns komfortabel die Elemente des Objektmodells für den Zugriff auf Visual Studio anzusehen.

Elemente des DTE2-Objekts

Das **DTE2**-Objekt hat folgende für uns interessante Eigenschaften:

- **ActiveDocument**: Liefert einen Verweis auf das aktuell aktive Codefenster (Beispiel siehe oben)
- **ActiveSolutionProjects**: Liefert eine Auflistung aller aktuell markierten Projekte, also auch solche Projekte, von denen mindestens ein Element markiert ist:

```
For Each objProject In objDTE.ActiveSolutionProjects
    Debug.Print(objProject.Name)
Next objProject
```

- **ActiveWindow**: Liefert einen Verweis auf das aktive Fenster – arbeitet ähnlich wie **ActiveDocument**.
- **CommandBars**: Enthält eine Auflistung aller Menüs von Visual Studio.

```
Dim i As Integer
For i = 1 To objDTE.CommandBars.Count()
    Debug.Print(objDTE.CommandBars.Item(i).Name)
Next i
```

- **Documents**: Auflistung aller geöffneten Dokumente, die gezählt oder deren Einträge ausgegeben werden können:

```
Debug.Print (objDTE.Documents.Count.ToString())
Dim objDocument As EnvDTE.Document
For Each objDocument In objDTE.Documents
    Debug.Print (objDocument.Name)
Next objDocument
```

- **ItemOperations**: Objekt, das einige interessante Methoden für den Zugriff auf die Elemente des Projekts liefert, zum Beispiel **AddExistingItem** zum Hinzufügen einer vorhandenen Datei zum Projekt oder **AddNewItem** zum Erstellen eines neuen Elements auf Basis des gewünschten Typs.
- **SelectedItems**: Liefert eine Liste der markierten Einträge des Projektmappen-Explorers:

```
Dim objItem As EnvDTE.SelectedItem
```

```
For Each objItem In objDTE.SelectedItems
    Debug.Print (objItem.Name)
Next objItem
```

- **Solution**: Liefert das **Solution**-Objekt aus dem Objektmappen-Explorer:

```
Debug.Print (objDTE.Solution.FullName.ToString())
```

- **StatusBar**: Liefert den aktuellen Inhalt der Statusleiste oder erlaubt das Einstellen des Inhalts. Wenn Sie einmal längere Operationen in Visual Studio per Erweiterung erledigen, können Sie den Benutzer in der Statusleiste darüber informieren:

```
Debug.Print (objDTE.StatusBar.Text)
objDTE.StatusBar.Text = "Fertig."
```

- **ToolWindows**: Ermöglicht den Zugriff auf die ToolWindows von Visual Studio, und zwar über jeweils eigene Eigenschaften für jedes der verfügbaren Toolwindows **CommandWindow**, **ErrorList**, **OutputWindow**, **SolutionExplorer**, **TaskList** und **ToolBox**:

```
'Text an CommandWindow senden
objDTE.ToolWindows.CommandWindow.SendInput("Test", False)
'Anzahl der markierten Objekte im Projektmappen-Explorer ausgeben
Debug.Print (objDTE.ToolWindows.SolutionExplorer.SelectedItems.Length.ToString())
'Ausgabe des aktiven Tabs in der Toolbox
Debug.Print (objDTE.ToolWindows.GetToolWindow("ToolBox").ActiveTab.Name)
```

Die Beispiele hierzu finden Sie in der Datei **VisualStudioProgrammieren_DTE.linq**.

ItemOperations: Elemente hinzufügen und mehr

Die **ItemOperations**-Eigenschaft des **DTE2**-Objekts bietet zum Beispiel verschiedene Möglichkeiten, Elemente zu einem Projekt hinzuzufügen:

- **AddExistingItem**: Fügt eine bereits auf der Festplatte existierende Datei zu einem Projekt hinzu, die als Parameter angegeben werden muss und arbeitet somit wie der Kontextmenü-Befehl **Hinzufügen|Vorhandenes Element...** des Projekt-Elements. Wenn mehr als ein Element im Projektmappen-Explorer markiert ist, wird ein Fehler ausgelöst:

```
objDTE.ItemOperations.AddExistingItem("C:\...\Beispieldatei.txt")
```

- **AddNewItem**: Fügt ein neues Element zum aktuell markierten Projekt beziehungsweise zu dem Projekt des aktuell markierten Elements im Projektmappen-Explorer hinzu und ersetzt somit den Kontextmenü-Befehl **Hinzufügen|Neues Element...** des Projekt-Elements. Den mit diesem Kontextmenü-Eintrag aufgerufenen Dialog benötigen Sie aber dennoch, um genaue Bezeichnung für das hinzuzufügende Element herauszufinden. Diesen als ersten Parameter übergebenen Ausdruck stellen

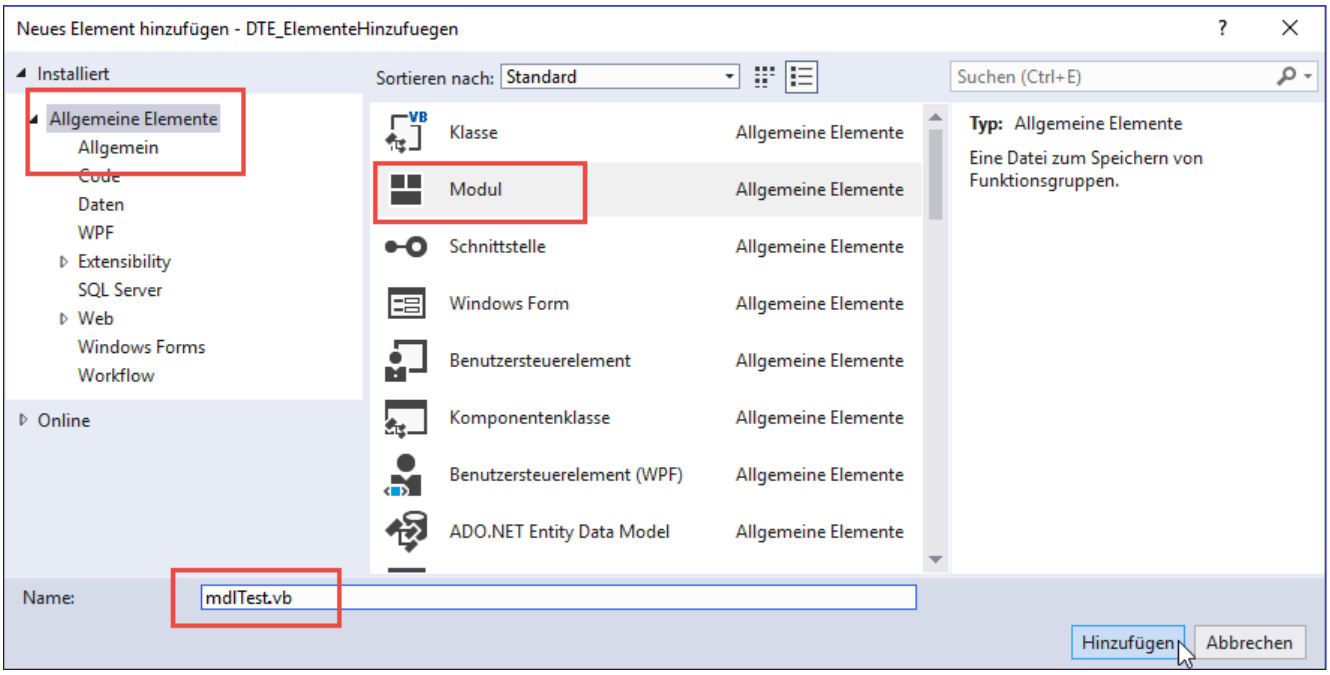


Bild 5: Diese Elemente geben Sie auch bei der **AddNewItem**-Methode an.

Sie aus der Kategorie und der Vorlage zusammen, also zum Beispiel wie in Bild 5 **Allgemeine Elemente\Code\Modul**. Sie stellen also einen durch das Backslash-Zeichen (\) getrennt die Kategorie oder Kategorien und die Elementbezeichnung zusammen. Das zweite Argument ist der Name des hinzuzufügenden Elements:

```
objDTE.ItemOperations.AddNewItem("Allgemeine Elemente\Code\Modul", "mdlBeispiel.vb")
```

- **IsFileOpen**: Diese Funktion liefert den Wert **True**, wenn die als Parameter angegebene Datei in Visual Studio geöffnet ist, aber nicht im Projektmappen-Explorer angezeigt wird. Sie liefert den Wert **False**, wenn die Datei entweder nicht in Visual Studio geöffnet ist oder aber dort geöffnet ist und im Projektmappen-Explorer erscheint. Beispiel für einen Aufruf:

```
Debug.Print(objDTE.ItemOperations.IsFileOpen("C:\...\Beispieldatei.txt"))
```

- **Navigate**: Navigiert in einem neuen Tab in Visual Studio zu der angegebenen Adresse. Hier können Sie eine Internetadresse angeben, aber auch den Pfad zu einer Datei. Dabei wird immer das aktuelle Fenster in Visual Studio als Ziel verwendet:

```
objDTE.ItemOperations.Navigate("http://www.access-im-unternehmen.de")
objDTE.ItemOperations.Navigate("C:\...\Beispieldatei.txt")
```

- **NewFile**: Legt ein neues Element in Visual Studio an, ohne es auf der Festplatte zu speichern. Der erste Parameter erwartet die Angabe der Vorlage (wie bereits bei **AddNewItem** beschrieben), der zweite den Namen. Wenn Sie keinen Parameter anlegen, wird eine Textdatei namens **TextFile1.txt** angelegt:

```
objDTE.ItemOperations.NewFile()
```

- **OpenFile**: Öffnet eine Datei von der Festplatte in Visual Studio. Die Parameter stimmen mit denen von **NewFile** überein.

Die Beispiele zu diesem Abschnitt finden Sie in der Datei **VisualStudioProgrammieren_ItemOperations.linq**.

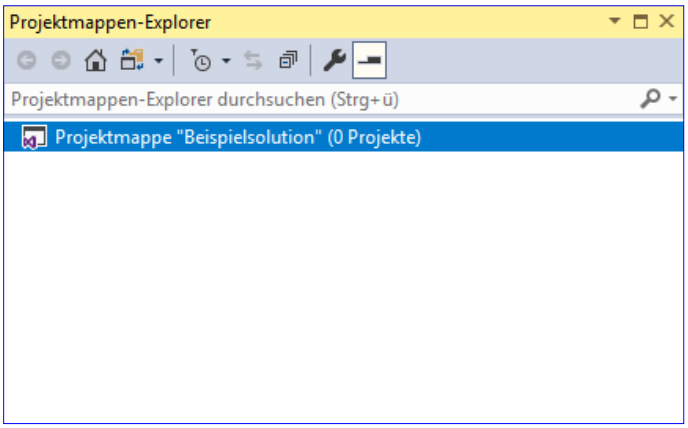


Bild 6: Anlegen eines Solution-Objekts

Mit Projekten arbeiten

Weiter oben haben Sie bereits die Solution-Eigenschaft des **DTE**-Objekts kennengelernt, mit dem Sie die aktuelle Solution referenzieren können. Wenn Sie Visual Studio geöffnet und noch kein Projekt geöffnet haben, ist der Projektmappen-Explorer noch leer. Nun führen Sie die folgenden Anweisungen aus:

```
Dim objSolution As EnvDTE.Solution
objSolution = objDTE.Solution
objSolution.Create("c:\Beispiele", "Beispielsolution")
```

Das Ergebnis ist eine leere Projektmappe im Projektmappen-Explorer (siehe Bild 6). Im Windows Explorer finden wir im hier angegebenen Verzeichnis **c:\Beispiele** allerdings noch keine neuen Einträge vor. Nun wollen wir ein Projekt hinzufügen. Das erledigen wir mit den folgenden Anweisungen, wobei wir allerdings erst einmal herausfinden müssen, wie die Parameter **TemplateName** und **Language** der Methode **GetProjectTemplate** gefüllt werden müssen.

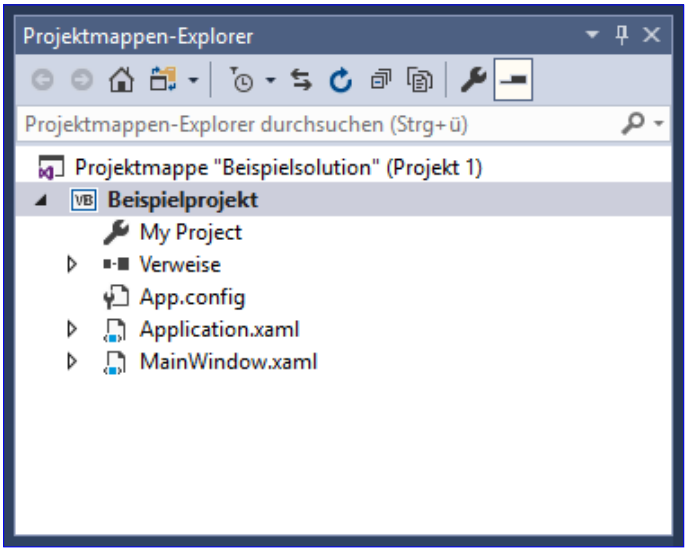


Bild 7: Ein per Code angelegtes Projekt

In diesem Beispiel verwenden wir zunächst **WPFApplication** als **TemplateName** und **VisualBasic** als Sprache. Die Funktion **GetProjectTemplate** liefert uns damit beispielsweise den folgenden Pfad, den wir in der Variablen **strTemplate** speichern:

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\
Community\Common7\IDE\ProjectTemplates\VisualBasic\Win-
dows\1031\WPFApplication\wpfapplication.vstemplate
```

Damit können wir dann die **AddFromTemplate**-Methode aufrufen und dieser den Pfad zur Vorlage, das Projektverzeichnis und den Namen des Projekts übergeben:

```
Dim objSolution As EnvDTE80.Solution2
objSolution = Trycast(objDTE.Solution, EnvDTE80.Solution2)
```

Klassen, Methoden und Co. per Code generieren

Im Beitrag »Visual Studio mit LINQPad: Project und ProjectItems« haben wir uns angesehen, wie Sie per Code auf Project-, Solution- und ProjectItem-Elemente zugreifen. Nun wollen wir uns den Umgang mit den ProjectItem-Objekten ansehen und herausfinden, wie wir beispielsweise den enthaltenen Code manipulieren können und Klassen, Prozeduren und Variablen anzulegen. Dazu benötigen wir einen Verweis auf das FileCodeModel-Objekt des jeweiligen ProjectItem-Elements. Wie wir das holen und was wir damit anstellen können, zeigt der vorliegende Artikel.

Nachdem wir im oben genannten Artikel gelernt haben, wie wir die Elemente eines Projekts referenzieren, durchlaufen und erstellen, wollen wir uns nun um den Inhalt der Elemente selbst kümmern, genau genommen um die Elemente, die Code enthalten. Für diese Zwecke gibt es das **FileCodeModel**, dass wir über die gleichnamige Eigenschaft eines jeden **ProjectItem**-Elements referenzieren können. Die folgende Funktion soll uns das **FileCodeModel**-Element der Klasse liefern, dessen Namen wir mit dem Parameter **strItem** übergeben. Die Funktion holt mit der im Artikel **Visual Studio mit LINQPad: Project und ProjectItems** vorgestellten Funktion **GetDTE** einen Verweis auf das **DTE**-Objekt zur Programmierung von Visual Studio und mit der Funktion **GetCurrentOrNewProject** aus dem gleichen Artikel einen Verweis auf das aktuell in Visual Studio geöffnete Projekt. Ist aktuell kein Projekt geöffnet, wird ein neues Projekt erstellt. Danach ermitteln wir über die **ProjectItems**-Auflistung des Projekts das **ProjectItem**-Objekt mit dem Namen aus dem Parameter **strItem**. Schließlich ermitteln wir mit der Eigenschaft **FileCodeModel** einen Verweis auf das gewünschte **FileCodeModel**-Objekt, das wir mit der **Return**-Anweisung an die aufrufende Methode zurückgeben:

```
Public Function GetFileCodeModel(strItem As String) As EnvDTE80.FileCodeModel2
    Dim objDTE As EnvDTE80.DTE2
    Dim objProject As EnvDTE.Project
    Dim objProjectItem As EnvDTE.ProjectItem
    Dim objFileCodeModel As EnvDTE80.FileCodeModel2
    If GetDTE(objDTE) = True Then
        If GetCurrentOrNewProject(objProject) = True Then
            objProjectItem = objProject.ProjectItems.Item(strItem)
            Debug.Print(objProjectItem.Name)
            objFileCodeModel = Trycast(objProjectItem.FileCodeModel, EnvDTE80.FileCodeModel2)
        End If
    End If
    Return objFileCodeModel
End Function
```

Der Aufruf könnte dann etwa wie folgt aussehen, wobei wir die zu verwendende Klasse als Parameter übergeben. Danach können wir mit dem **FileCodeModel** experimentieren:

```
Dim objFileCodeModel As EnvDTE.FileCodeModel
objFileCodeModel = GetFileCodeModel("NeueKlasse.vb")
Debug.Print (objFileCodeModel.CodeElements.Count)
```

Beispiele zu diesem Artikel

Die Beispiele finden Sie in der Datei **VisualStudioErweitern_Items.linq**. Sie können die einzelnen Prozeduren aufrufen, indem Sie die Aufrufe in der **Sub Main**-Prozedur auskommentieren und **Sub Main** ausführen. Zum Ausprobieren der Prozeduren verwenden Sie das Tool LINQPad, dass Sie unter **http://www.linqpad.net** herunterladen können.

Eigenschaften und Methoden der FileCodeModel-Klasse

Die **FileCodeModel**-Klasse hat die folgenden Eigenschaften. Genau genommen schauen wir uns hier nicht die **FileCodeModel**-Klasse an, sondern das **FileCodeModel2**-Interface, das etwas mehr Möglichkeiten bietet und mit dem Namespace **EnvDTE80** kommt:

- **CodeElements**: Auflistung der Code-Elemente. Was genau Code-Elemente sind, schauen wir uns weiter unten an, nachdem wir einige angelegt haben.
- **DTE**: Verweis auf das übergeordnete **DTE**-Objekt.
- **Language**: Gibt eine GUID aus, welche die verwendete Sprache dieses **FileCodeModel**-Elements repräsentiert.

```
Debug.Print (objFileCodeModel.Language)
  {B5E9BD33-6D3E-4B5D-925E-8A43B79820B4}
```

- **Parent**: Ermittelt das übergeordnete Objekt.

Dies sind die Methoden der **FileCodeModel**-Klasse:

- **AddAttribute**: Fügt ein Attribut wie **<Assembly: NeuesAttribut(Wert)>** zu einer Klasse hinzu.
- **AddClass**: Fügt eine neue Klasse zum **FileCodeModel** hinzu.
- **AddDelegate**: Fügt ein Delegate zum **FileCodeModel** hinzu.
- **AddEnum**: Fügt eine Enumeration zum **FileCodeModel** hinzu.
- **AddFunction**: Hinzufügen von Methoden, Funktionen, Property Get/Let/Set-Methoden und so weiter
- **AddImport**: Füge eine **Imports**-Anweisung hinzu.
- **AddInterface**: Hinzufügen eines Interfaces.

- **AddNamespace**: Fügt einen Namespace hinzu.
- **AddStruct**: Fügt eine Struktur hinzu.
- **AddVariable**: Fügt eine Variable hinzu.
- **CodeElementFromPoint**:
- **Remove**:

Die meisten der hier genannten **Add...**-Methoden gibt es auch in der Klasse **CodeClass**. Das heißt, Sie erstellen erst ein **CodeClass**-Element, also eine Klasse, referenzieren diese mit einer Objektvariablen wie **objClass** und nutzen dann Methoden wie **AddAttribute**, **AddBase**, **AddClass**, **AddDelegate**, **AddEnum**, **AddFunction**, **AddImplementedInterface**, **AddProperty**, **AddStruct** oder **AddVariable**, um die entsprechenden Elemente zu einer Klasse hinzuzufügen. Darunter sind Methoden wie **AddImplementedInterface** oder **AddProperty**, die nicht für das **FileCodeModel**-Objekt bereitstehen.

In den folgenden Abschnitten erläutern wir die Methoden mal für das **FileCodeModel**-Objekt und mal für das **CodeClass**-Objekt.

Attribut hinzufügen mit AddAttribute

Mit der folgenden Anweisung fügen wir ein neues Attribut zu einem **FileCodeModel** hinzu:

```
objFileCodeModel.AddAttribute("NeuesAttribut","Wert")
```

Der Code in der Datei **NeueKlasse.vb** sieht danach wie folgt aus:

```
<Assembly: NeuesAttribut(Wert)>
Public Class NeueKlasse

End Class
```

Hinzufügen einer Klasse mit AddClass

Mit der Methode **AddClass** und mindestens dem Klassennamen als Parameter fügen Sie eine neue Klasse hinzu:

```
objFileCodeModel.AddClass("WeitereKlasse")
```

Das Ergebnis mit diesem einfachen Aufruf sieht so aus:

```
Public Class NeueKlasse

End Class
```

Visual Studio erweitern: VSIX-Projekt weitergeben

Wenn Sie eine Visual Studio-Erweiterung programmiert haben, möchten Sie diese in der Praxis einsetzen. Danach können Sie die Erweiterung nutzen, ohne diese beispielsweise wie beim Entwickeln erst zum Debuggen starten zu müssen. Sie können die Erweiterung dann also etwa über das dafür vorgesehene Kontextmenü oder die Tastenkombination aufrufen. Dazu sind nur noch wenige Schritte notwendig, die wir in diesem Artikel erläutern.

Wenn die VSIX-Erweiterung einmal fertigprogrammiert ist, benötigen Sie nur noch einige Schritte, bis Sie diese in Visual Studio verwenden können.

Erstellen der Erweiterung

Der erste Schritt ist das Erstellen des Projekts mit dem Menüeintrag **Erstellen|Projektmappe erstellen**. Bevor Sie das erledigen, passen Sie noch die Produkteigenschaften wie Produktname, Beschreibung, Hersteller und so weiter ein. Dies erledigen Sie in der Datei **source.extension.vsixmanifest** (siehe Bild 1).

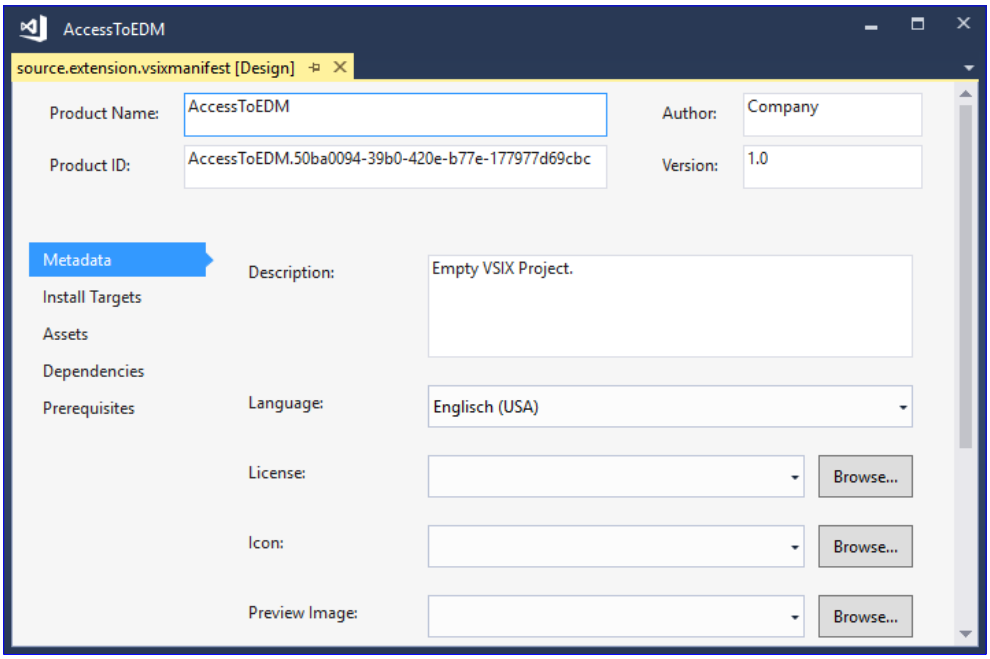


Bild 1: Einstellen der Produkteigenschaften

Danach stellen Sie in der Konfiguration für das Kompilieren den Wert Release ein (siehe Bild 2).

Danach können Sie das Projekt mit dem Menübefehl **Erstellen|Projektmappe erstellen** erstellen. Sie finden das neu erstellte Projekt dann im Verzeichnis **Release** unterhalb des Projektord-

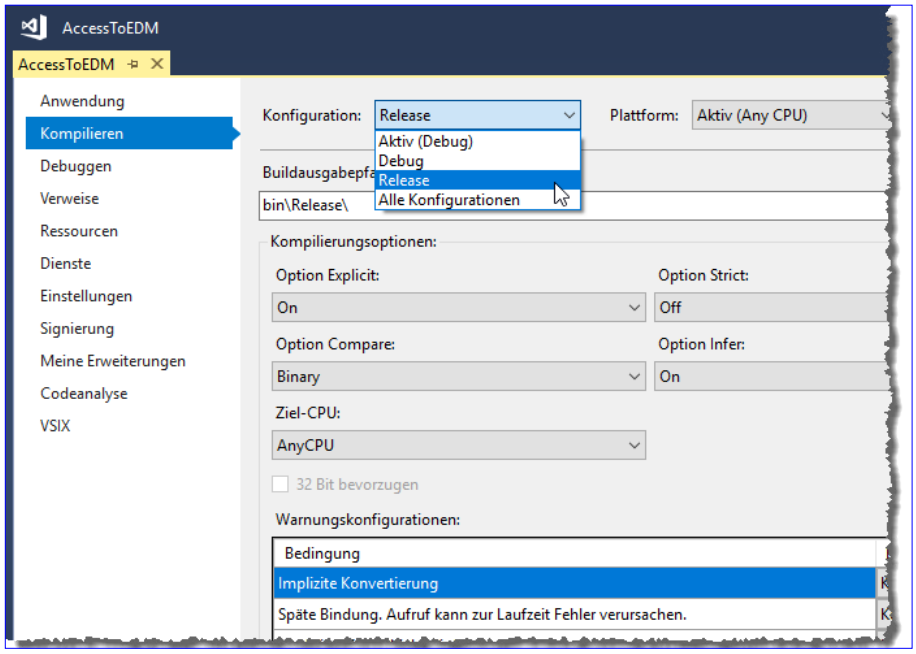


Bild 2: Einstellen des Kompiliermodus

ners. Dieser Ordner enthält eine Datei mit der Dateiendung **.vsix** vor. Bevor Sie diese installieren, schließen Sie zunächst alle geöffneten Instanzen von Visual Studio.

Installieren der Erweiterung

Klicken Sie dann doppelt auf die Datei mit der Endung **.vsix**. Es kann eine Weile dauern, dann erscheint der Dialog aus Bild 3.

Nach wenigen Augenblicken wird der Abschluss der Installation bestätigt (siehe Bild 4).

Beim nächsten Start von Visual Studio sollte der Menü- oder Kontextmenü-Eintrag, den Sie zum Starten der Funktion der Erweiterung vorgesehen haben, an der gewünschten Stelle erscheinen.

Verwalten von Extensions

Wenn Sie den Menübefehl **Extras|Extensions und Updates...** aufrufen, erscheint der Dialog aus Bild 5. Hier finden Sie auch einen Eintrag für die frisch hinzugefügte Extension vor. Sie können die Extension hier deaktivieren und deinstallieren.

Speicherort der Extension

Wo auf dem lokalen Rechner befinden sich die physischen Dateien der Extension? Unsere Extension befand sich nach dem Installieren durch doppeltes Anklicken der **.vsix**-Datei im Verzeichnis **C:\Users\User\AppData**

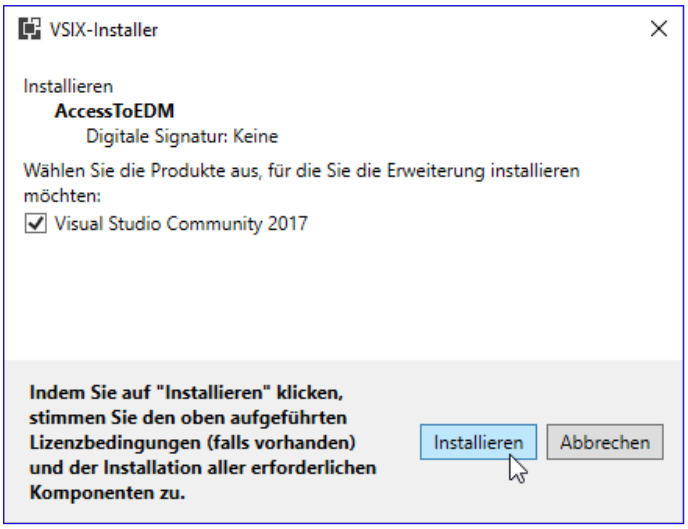


Bild 3: VSIX-Installer

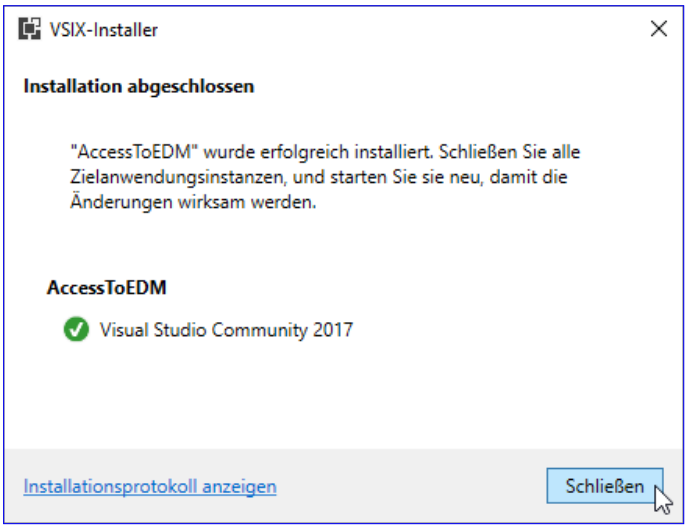


Bild 4: Abschluss der Installation

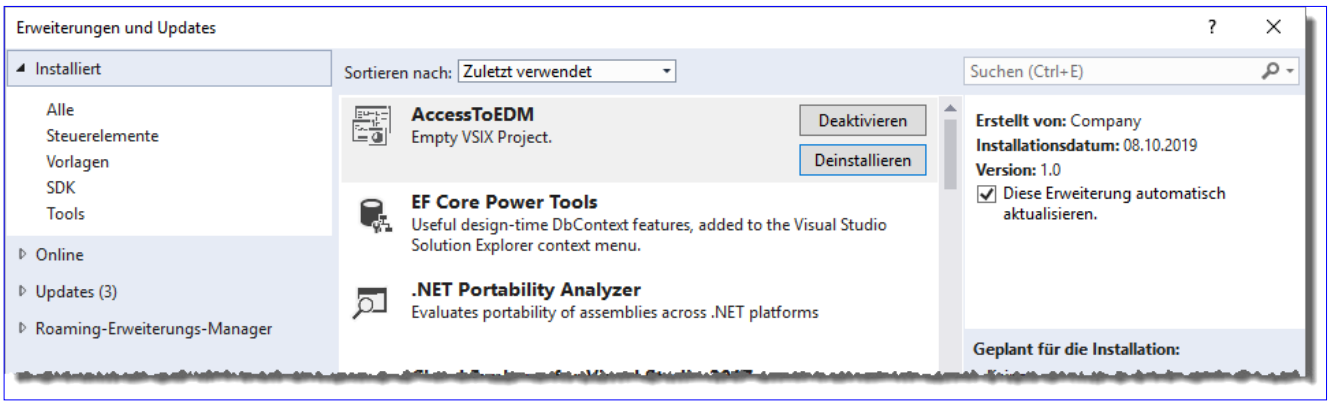


Bild 5: Das installierte Element

Von Access zum EDM per Kontextmenü

Nach den vorbereitenden Artikel zum Thema Erweitern von Visual Studio, Anlegen von Elementen im Projektmappen-Explorer und Hinzufügen von Elementen wie Klassen, Prozeduren et cetera per Code können wir uns an ein praktisches Beispiel begeben: Eine Anwendung, die Sie per Menüeintrag aus dem Kontextmenü eines Projekts aus aufrufen können und die nach der Auswahl der Access-Datenbank mit dem zu verwendenden Datenmodell ein Entity Data Model auf Basis dieser Datenbank erstellt. Damit können Sie dann mit wenigen Handgriffen auch direkt eine SQL Server-Datenbank generieren.

Grundlagen

Die Grundlagen zu den hier verwendeten Techniken finden Sie in den Artikeln **Visual Studio erweitern: Menübefehle**, **Visual Studio erweitern: Elemente hinzufügen**, **Visual Studio mit LINQPad: Project und ProjectItems** und **Klassen, Methoden und Co. per Code generieren**. Außerdem haben wir den größten Teil des Codes, den wir hier in eine Extension stecken wollen, in den Artikeln **Von Access zu Entity Framework: Datenmodell**, **Von Access zu Entity Framework: Daten**, **Von Access zu EF: Step by Step** und **Von Access zu Entity Framework: Update 1** beschrieben.

Projekt erstellen

Die Erweiterung, die unsere Funktionalität in Visual Studio zur Verfügung stellen soll, basiert auf einem Projekt mit der Vorlage **Visual Basic|Extensibility|VSIX Projekt**, die wir **Access2EDM** nennen. Die beiden Elemente **index.html** und **stylesheet.css**, die im nun erstellen Projekt vorliegen, löschen wir direkt.

Dafür fügen wir über den Kontextmenü-Eintrag **Hinzufügen|Neues Element...** des Projekt-Elements im Projektmappen-Explorer ein neues Element des Typs **Extensibility|Custom Command** hinzu und nennen dieses **cmdAccess2EDM**.

Damit der standardmäßig im **Extras**-Menü angezeigte Befehl nun im Kontextmenü des Projekt-Elements im Projektmappen-Explorer erscheint, ändern wir eine Zeile in der Datei **cmdAccess2EDMPackage.vsct**, wo wir **IDM_VS_MENU_TOOLS** durch **IDM_VS_CTXT_PROJNODE** ersetzen, was folgende Zeile ergibt:

```
<Parent guid="guidSHLMainMenu" id="IDM_VS_CTXT_PROJNODE"/>
```

Außerdem ändern wir in der Zeile zur Definition des Elements **ButtonText** die Beschriftung wie folgt:

```
<ButtonText>EDM aus Access-Datenbank</ButtonText>
```

Icon hinzufügen

Als Icon wollen wir ein Access-Logo verwenden. Dieses finden wir in einer Ressourcen-Datei **accicons.exe**. Wir extrahieren die **.ico**-Datei und fügen diese zum Ordner **Resources** des Projekts hinzu. Außerdem ändern wir noch eine weitere Zeile in der bereits bearbeiteten Datei wie folgt ab, um die neu hinzugefügte Bilddatei zu verwenden:

```
<Bitmap guid="guidImages" href="Resources\Access.ico"/>
```

Danach finden wir den Eintrag **EDM aus Access-Datenbank** wie Bild 1 in im Kontextmenü des Projekt-Elements im Projektmappen-Explorer der beim Debuggen gestarteten Instanz von Visual Studio vor.

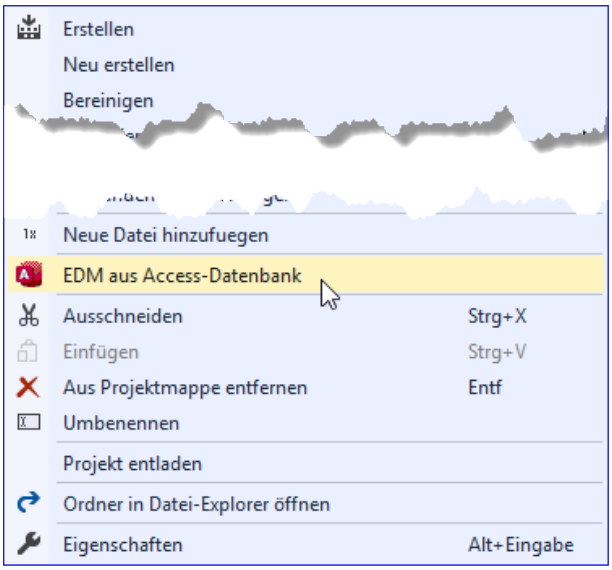


Bild 1: Erster Test des neuen Kontextmenü-Eintrags

Damit sind die Arbeiten an der Datei **cmdAccess2EDMPackage.vsct** beendet und Sie können diese Datei speichern und schließen.

Wechsel zu LINQPad

Ab dieser Stelle sind wir für die Programmierung komplett auf LINQPad umgeschwenkt. Das ständige Warten beim Debuggen der Anwendung zum Testen zerstört jeden Programmierfluss. Nur zwischendurch zum Testen und am Ende fügen wir den produzierten Code in das VSIX-Projekt ein.

Die Basis: Entity Data Model

Basis für das Erstellen eines Entity Data Model auf Basis des Datenmodells einer Access-Datenbank ist das Hinzufügen eines neuen Elements des Typs **ADO.NET Entity Date Model**, was normalerweise über den Dialog **Neues Element hinzufügen** geschieht. Dort wird dann auch der Name des neuen Entity Data Models eingegeben (siehe Bild 2).

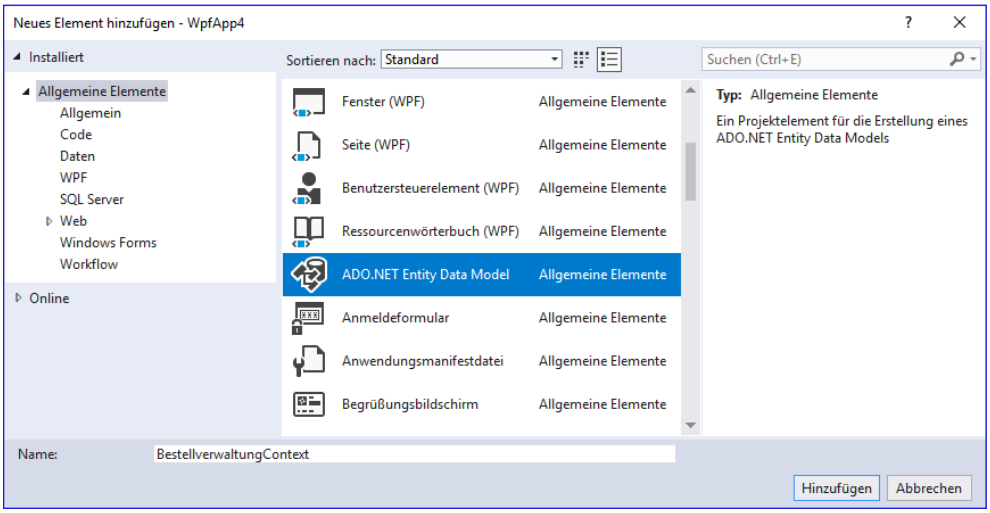


Bild 2: Hinzufügen des Entity Data Models auf herkömmlichem Wege

Diesen Schritt wollen wir bereits per Code erledigen und dazu müssen wir den Pfad der Vorlage ermitteln. Diese finden wir mit einer kleinen Prozedur heraus, die wir im Tool **LINQPad** ausführen können und die wie folgt aussieht:

```
Public Sub Main
    Dim objDTE As EnvDTE80.DTE2 = Nothing
    Dim objSolution As EnvDTE80.Solution2
    Dim strTemplate As String
    If GetDTE(objDTE) = True Then
```

```
        objSolution = Trycast(objDTE.Solution, EnvDTE80.Solution2)
        strTemplate = objSolution.GetProjectItemTemplate("Daten\ADO.NET Entity Data Model", "VisualBasic")
        Debug.Print(strTemplate)
    End If
End Sub
```

Die hier verwendete Hilfsfunktion **GetDTE** finden Sie in der Datei **Access2EDM.linq**. In dieser Datei entwickeln wir innerhalb der Benutzeroberfläche von LINQPad auch den Quellcode dieser Lösung. Der Hintergrund ist, dass das Debuggen von Extensions mit Visual Studio äußerst zeitaufwendig ist, da das Starten zum Debuggen wegen der dazu benötigten zusätzlichen Instanz von Visual Studio sehr lange dauert.

Mit der oben genannten Prozedur finden wir den Pfad der Vorlage für das **ADO.NET Entity Data Model** schnell heraus:

```
C:\Program Files (x86)\Microsoft
Visual Studio\2017\Community\
Common7\IDE\ItemTemplates\Visual-
Basic\Data\1033\EF_EDM\ModelObjec-
tItemVB.vstemplate
```

Sie können sich diese Datei auch einmal ansehen.

ADO.NET Entity Data Model-Vorlage hinzufügen

Wenn wir die oben begonnene Prozedur wie folgt erweitern, sollten eigentlich automatisch die entsprechenden Elemente für das Entity Data Model hinzugefügt werden:

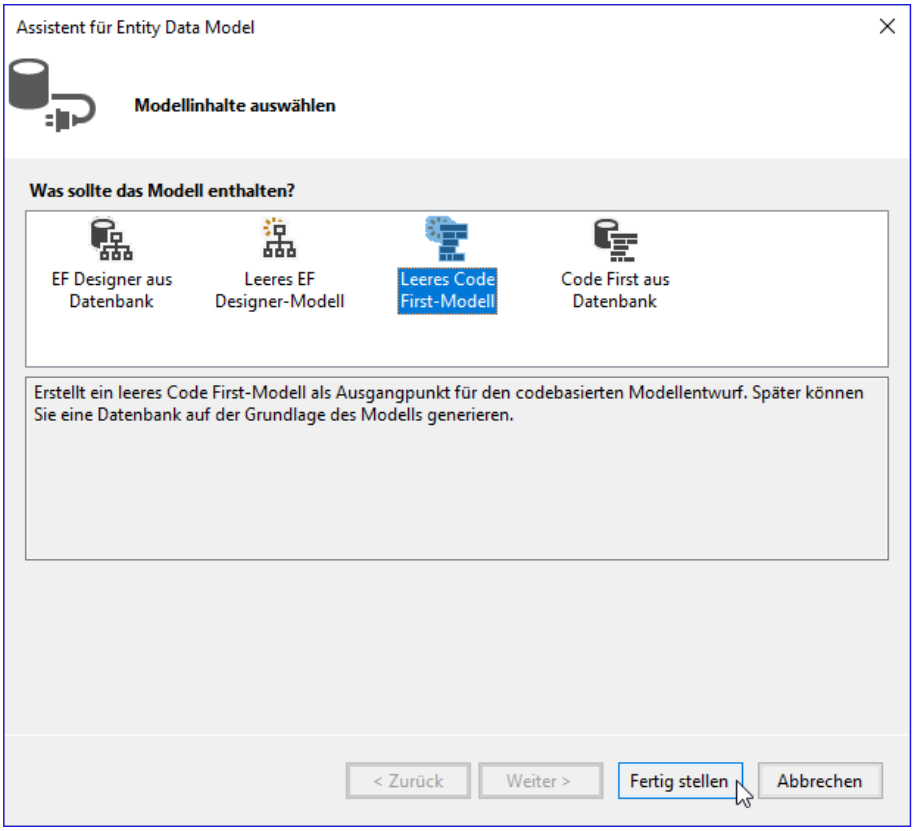


Bild 3: Entity Data Model hinzufügen

```
If GetCurrentOrNewProject(objProject) = True Then
    objProject.ProjectItems.AddFromTemplate(strTemplate, "BestellverwaltungContext")
End If
```

Allerdings ruft dies noch den Dialog aus Bild 3 auf den Plan. Hier müssen wir noch die Option **Leeres Code First-Modell auswählen** und auf die Schaltfläche **Fertigstellen** klicken, damit die Elemente des leeren Entity Data Models zum Projekt hinzugefügt werden. Leider ist und keine Möglichkeit bekannt, wir diesen Dialog umschiffen können. Daher fügen wir vorher ein Meldungsfenster ein, das den Benutzer darauf hinweist, die Option **Leeres Code First-Modell** auszuwählen.

In der Datei **ModelObjectItemVB.vstemplate** sehen wir dann auch, dass dort noch ein Assistent namens **ModelObjectItemWizard** geöffnet im Spiel ist, der vermutlich den soeben beschriebenen Dialog erscheinen lässt.

Wenn wir uns nun allerdings ansehen, welche Elemente und Änderungen der Assistent in unser Projekt gebracht hat, stellt sich die Frage, ob wir nicht selbst eine passende Erweiterung aus dem Ärmel schütteln können. Doch das würde aktuell den Rahmen sprengen, sodass wir uns dieses Thema in einer späteren Ausgabe vornehmen werden.

Wir haben nun jedenfalls die grundlegenden Elemente des Entity Data Models im Projekt:

- die Context-Klasse, die aus dem von uns eingegebenen Namen für die Anwendung und dem Zusatz **Context.vb** besteht, also etwa **BestellverwaltungContext.vb**,
- die Datei **App.Config**, in der im Bereich **connectionStrings** die Verbindungszeichenfolge für die zu erstellende SQL Server-Datenbank hinzugefügt wurde sowie
- wie in der Datei **packages.config** zu erkennen, die hinzugefügten Pakete mit den Funktionen des Entity Frameworks.

Diese müssen wir nun erweitern, so wie wir es zuvor wie in den eingangs genannten Artikeln von Access aus erledigt haben. Dort haben wir auch per Code die Tabellendefinitionen durchlaufen und die passenden Entitätsklassen sowie Entitätsauflistungen definiert und die Anweisungen zum Einfügen der in den Tabellen enthaltenen Daten zusammengestellt. Das wollen wir nun von unserer Erweiterung aus machen.

Access-Datenbank ermitteln

Dazu müssen wir als erstes herausfinden, welche Access-Datenbank als Grundlage für die Umsetzung des Datenmodells in ein Entity Data Model dienen soll. Dazu zeigen wir dem Benutzer einen **Datei auswählen**-Dialog an, was wir mit folgendem Code bewerkstelligen. Wir benötigen einen Verweis auf den Namespace **Microsoft.Win32**:

```
Imports Microsoft.Win32
```

Außerdem verwenden wir die folgende Funktion, um den Pfad mit einem Dateiauswahl-Dialog zu ermitteln:

```
Public Function GetDatabasePath() As String
    Dim objOpenFileDialog As OpenFileDialog
    Dim strDatabasePath As String = ""
    objOpenFileDialog = New OpenFileDialog
    With objOpenFileDialog
        .Filter = "Access-Datenbanken (*.mdb,*.accdb)|*.mdb;*.accdb|Alle Dateien (*.*)|*.*"
        .FilterIndex = 1
        If (.ShowDialog = DialogResult.OK) Then
            strDatabasepath = .FileName
        End If
```

```
    End With
    Return strDatabasePath
End Function
```

Wie wir diese aufrufen, zeigen wir gleich im Gesamtkontext der aufrufenden Methode.

Code zum Erzeugen des Datenmodells in das Visual Studio-Projekt kopieren

In den zu Beginn erwähnten Artikel haben wir bereits den Code beschrieben, der zum Erzeugen eines Entity Data Models aus einem Access-Datenmodell erforderlich ist. Diesen Code wollen wir nun zunächst aus dem Modul **mdlEDM** der Datenbank **Access2EDM.accdb** in ein neues Modul unseres Visual Basic-Projekts einfügen.

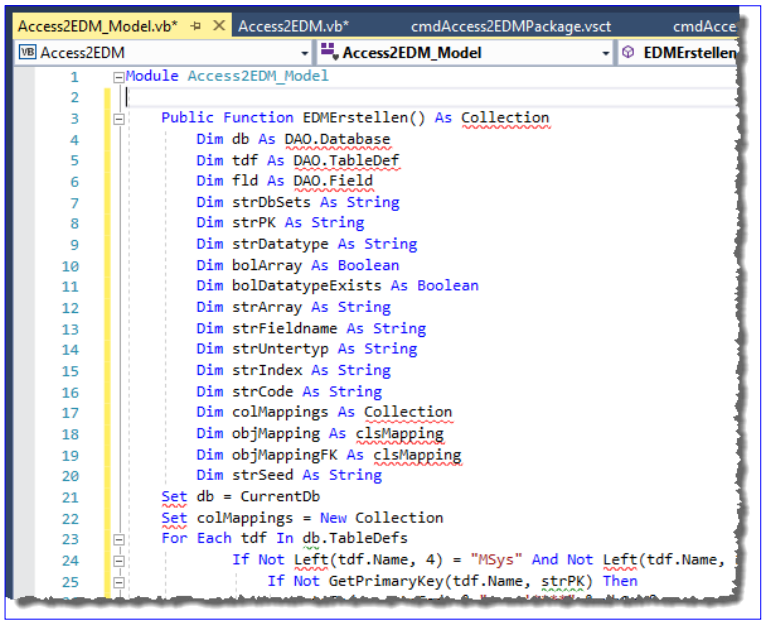


Bild 4: Fehler beim Einfügen des VBA-Codes

Das noch zu erstellende Modul soll **mdlAccess2EDM_Model** heißen. Den Code aus dem Modul **mdlEDM** der Access-Datenbank fügen wir einfach dort ein.

Damit erhalten wir natürlich den einen oder anderen Fehler, wie auch der Ausschnitt aus Bild 4 zeigt. Das wir VBA-Code nicht problemlos in ein Visual Basic-Project übernehmen können, war allerdings abzusehen.

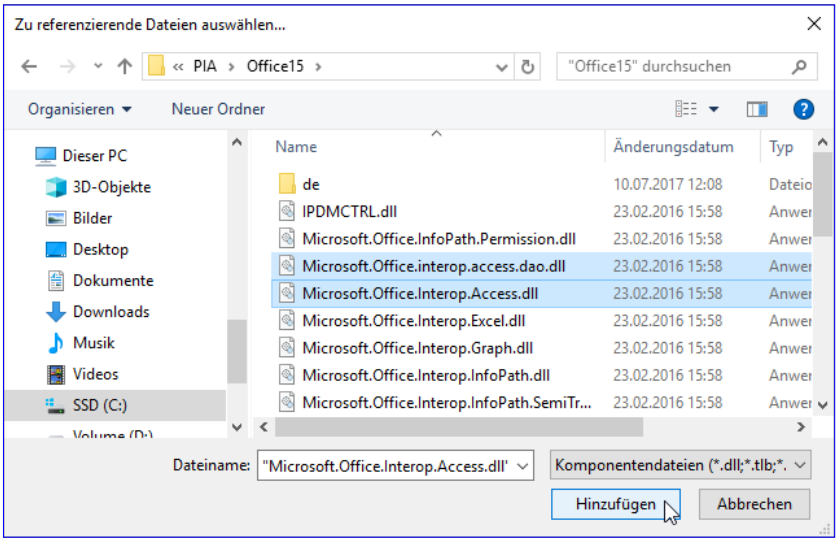


Bild 5: Hinzufügen der **Access Database Engine**-Bibliothek

Die ersten Fehler durch das Fehlen der DAO-Objektbibliothek beheben wir, indem wir den fehlenden Objektverweis hinzufügen. Dazu öffnen Sie den **Verweis-Manager** von Access, klicken auf Durchsuchen und navigieren zum Ordner **C:\Program Files (x86)\Microsoft Visual Studio 14.0\Visual Studio Tools for Office\PIA\Office15**. Hier fügen Sie die beiden Bibliotheken aus Bild 5 hinzu.

Danach fügen Sie der Datei mit dem Modul **Access2EDM_Model** die folgenden Anweisungen hinzu:

```
Imports Microsoft.Office.Interop.Access
Imports Microsoft.Office.Interop.Access.Dao
```

Nun führen Sie folgende Schritte durch, um die VBA-Routine aus der Access-Datenbank Visual Basic-kompatibel zu machen:

- Entfernen der **Set**-Anweisungen vor Zuweisungen an Objekte

- Ersetzen der **MsgBox**-Anweisung durch **MessageBox.Show** und Anpassen der Parameter sowie gegebenenfalls Hinzufügen des Verweises auf **System.Windows.Forms** und dem entsprechenden Namespace:

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

- Einfassen von Parametern von Funktionsaufrufen oder Methoden jeweils in Klammern

- Wo wir mit **CurrentDb** auf die aktuelle Datenbank zugreifen, müssen wir mit **DAO.DbEngine.OpenDatabase** erst die Datenbank öffnen und geben diese dann per Parameter an aufgerufene Funktionen weiter, die auch auf diese Datenbank zugreifen:

```
Dim objDBEngine As Dao.DBEngine
objDBEngine = New Dao.DBEngine
db = objDBEngine.OpenDatabase(strDatabasePath)
```

- Rückgabewerte von Funktionen werden nicht mehr der Variablen mit dem Namen der Funktion zugewiesen, sondern mit **Return <Wert>**.

- Ganz wichtig: Parameter, die Sie zum Ändern an eine Funktion übergeben, die also von der aufrufenden Methode anschließend weiterverwendet werden sollen, müssen Sie mit dem Schlüsselwort **ByRef** deklarieren, in folgendem Beispiel den dritten Parameter:

```
Public Function GetPrimaryKeys(db As Dao.database, strTable As String, ByRef strPKs As String) As Boolean
```

Größere Umbauarbeiten müssen wir an der Klasse **clsMappings** vornehmen, da die Gestaltung von Klassen mit Properties unter Visual Basic anders aussieht als unter VBA. Stellvertretend für die übrigen Member hier der neue Aufbau für die Eigenschaft **Tabelle**:

```
Public Class clsMapping
    Private m_Tabelle As String
    ...
    Public Property Tabelle As String
        Get
            Return m_Tabelle
        End Get
        Set(value As String)
            m_Tabelle = value
        End Set
    End Property
```

```
    ...
End Class
```

DBSet-Anweisungen zur Context-Klasse hinzufügen

Die automatisch hinzugefügte Context-Klasse enthält einen auskommentierten Eintrag, der vorgibt, wie die Definitionen für die **DbSet**-Auflistungen der Entitäten aussehen sollen:

```
' Public Overridable Property MyEntities() As DbSet(Of MyEntity)
```

Diese wollen wir als Erstes hinzufügen. Dazu benötigen wir allerdings erst einmal die notwendigen Informationen, in diesem Fall eine Collection mit Elementen der Klasse **clsMapping**. Diese enthalten Infos über die ursprünglichen Daten wie den Tabellennamen und das Primärschlüsselfeld, aber auch den Namen der zu erstellenden Entitätsklasse und des **DbSet**-Elements. Die folgende Funktion ermittelt diese Daten aus dem Datenmodell.

Dabei wird der Teil des Tabellennamens ohne **tbl** als Name der **DbSet**-Auflistung (bei **tblKunden** also **Kunden**) und der Name des Primärschlüsselfeldes ohne abschließendes **ID** als Name der Entität verwendet (bei **KundeID** also **Kunde**). Manchmal sind beide gleich, dann wird der Benutzer aufgefordert, unterschiedliche Bezeichnungen einzugeben – beispielsweise bei **tblArtikel** und **ArtikelID** kann man dann **Produkte** und **Produkt** angeben. Sind alle Tabellen durchlaufen, werden die Informationen in einer Collection zurückgegeben:

```
Public Function GetMappings(strDatabasePath As String) As Collection
    Dim db As DAO.Database, tdf As DAO.TableDef
    Dim strDbSets As String = ""
    Dim strPK As String = ""
    Dim strDatatype As String = ""
    Dim strCode As String = ""
    Dim colMappings As Collection
    Dim objMapping As clsMapping = Nothing
    Dim strSeed As String = ""
    Dim objDBEngine As Dao.DBEngine
    objDBEngine = New Dao.DBEngine
    db = objDBEngine.OpenDatabase(strDatabasePath)
    colMappings = New Collection
    For Each tdf In db.TableDefs
        If Not Left(tdf.Name, 4) = "MSys" And Not Left(tdf.Name, 1) = "~" Then
            If Not GetPrimaryKey(db, tdf.Name, strPK) Then
            MessageBox.Show($"Mehrere Primärschüssel in der Tabelle {tdf.Name}. Die Tabelle wird nicht berücksichtigt.")
            Else
                objMapping = New clsMapping
                With objMapping
                    .Entity = Replace(strPK, "ID", "")
```

```
                .Entity_Original = .Entity
                .Entities = Replace(tdf.Name, "tbl", "")
                .Entities_Original = .Entities
                If .Entity = .Entities Then
                    Do While .Entity = .Entities
                        MessageBox.Show("Plural und Singular der Bezeichnung der Entitäten (hier """ & .Entity _
                            & """) im Tabellennamen scheinen gleich zu sein." & vbCrLf & vbCrLf & "Geben Sie " _
                            & "nachfolgend eine Bezeichnung für den Singular der Entität ein und eine für den Plural.")
                        .Entity = InputBox("Bezeichnung für den Singular/den Klassennamen der Entität """ ?
                            & .Entity & """:", "Singular bestimmen", .Entity)
                        .Entities = InputBox("Bezeichnung für den Plural/den Namen der Auflistung der Entität """ _
                            & .Entities & """:", "Plural bestimmen", .Entities)
                    Loop
                End If
                .ID = "ID"
                .PK = strPK
                .Tabelle = tdf.Name
            End With
        End If
        colMappings.Add(objMapping)
    End If
Next tdf
Return colMappings
End Function
```

Diese Informationen rufen wir von der Hauptmethode aus etwa mit dieser Anweisung ab:

```
colMappings = GetMappings(strDatabasePath)
```

Entitätslisten in die Konstruktor-Methode integrieren

Mit dieser Collection ausgestattet können wir beginnen, den Code der Context-Klasse, die in unserem Fall **Bestellverwaltung-Context** heißt, zu ergänzen. Die Klasse enthält bisher nur die Konstruktor-Methode, sodass wir uns austoben können und die **DbSet**-Definitionen einfach am Ende der Klasse einfügen.

Diese Aufgabe übernimmt die folgende Methode, die einen Verweis auf das zu ändernde Projekt, den Namen des Context-Objektes, hier **Bestellverwaltung** und die Collection mit den Mappings als Parameter erhält:

```
Public Sub AddDbSets(objProject As envDTE.Project, strContextName As String, colMappings As Collection)
    Dim objItem As EnvDTE.ProjectItem
    Dim objFileCodeModel As EnvDTE80.FileCodeModel2
    Dim objCodeElement As EnvDTE.CodeElement
```

```
    Dim objClass As EnvDTE.CodeClass = Nothing
    Dim objEditPoint As EnvDTE80.EditPoint2
    Dim objMapping As clsMapping
```

Die Methode ermittelt zunächst das Projektelement **BestellverwaltungContext.vb** und holt dessen **FileCodeModel**:

```
    objItem = objProject.ProjectItems.Item(strContextName & "Context.vb")
    objFileCodeModel = objItem.FileCodeModel
```

Dann durchläuft Sie alle **CodeElement**-Elemente, bis sie die Klasse **BestellverwaltungContext** gefunden hat und referenziert diese mit der Objektvariablen **objClass** (leider gibt es keine direkte Funktion, um Elemente per Name zu finden – daher die selbst gebaute Lösung):

```
    For Each objCodeElement In objFilecodemodel.CodeElements
        If (objCodeElement.Name = strContextName & "Context") Then
            objClass = objCodeElement
            Exit For
        End If
    Next objCodeElement
```

Dann ermitteln wir als **EditPoint** mit **GetEndPoint** den Endpunkt der Klasse, also direkt vor **End Class**:

```
    objEditPoint = objClass.GetEndPoint.CreateEditPoint
```

Hier durchlaufen wir dann alle Elemente aus **colMapping**, ermitteln daraus jeweils das **clsMapping**-Objekt und fügen jeweils eine neue Zeile vor der Zeile **End Class** mit je einer **DBSet**-Deklaration hinzu:

```
    For Each objMapping In colMappings
        objEditPoint.StartOfLine
        objEditPoint.Indent(Nothing, 1)
        objEditPoint.Insert($"Public Overridable Property {objMapping.Entities}() As DbSet(Of {objMapping.Entity})")
        objEditPoint.InsertNewLine
    Next objMapping
End Sub
```

Der Aufruf erfolgt von der Hauptmethode mit der folgenden Anweisung:

```
AddDBSets(objProject, strContextName, colMappings)
```

Das Ergebnis sieht dann anschließend wie in Bild 6 aus. Da hier die Typen für die Entitätsklassen als fehlerhaft markiert werden, kümmern wir uns als nächstes um das Anlegen der entsprechenden Klassen.

PayPal-Kontostand und Umsätze

PayPal ist ein beliebtes Zahlungsmittel. Gut, wenn man auf einfache Weise den Überblick über den Stand seines PayPal-Kontos und über die erfolgten Umsätze erhält – zum Beispiel über die Webseite oder eine App für das Smartphone. Uns interessiert natürlich, wie wir von der Benutzeroberfläche einer WPF-Anwendung auf diese Informationen zugreifen können. Deshalb schauen wir uns in diesem Artikel an, wie Sie die notwendigen Sicherheitsinformationen für den Zugriff erhalten und wie Sie eine Anwendung um Funktionen zum Abrufen von Kontostand und Umsätzen erweitern.

Um die für den Zugriff nötigen Daten zu erhalten, öffnen Sie im Webbrowser die folgende URL: **https://www.paypal.com/businessmanage/credentials/apiAccess**. Hier ist zunächst eine Anmeldung nötig, danach erscheint die angeforderte Seite. Hier gibt es weiten unten einen Link mit der Beschriftung **API-Berechtigung verwalten**.

Damit gelangen Sie dann, wenn Sie die API-Daten bereits einmal angefordert haben, zu der Seite aus Bild 1, wo Sie die drei Informationen **API-Benutzername**, **API-Passwort** und **Signatur** finden.

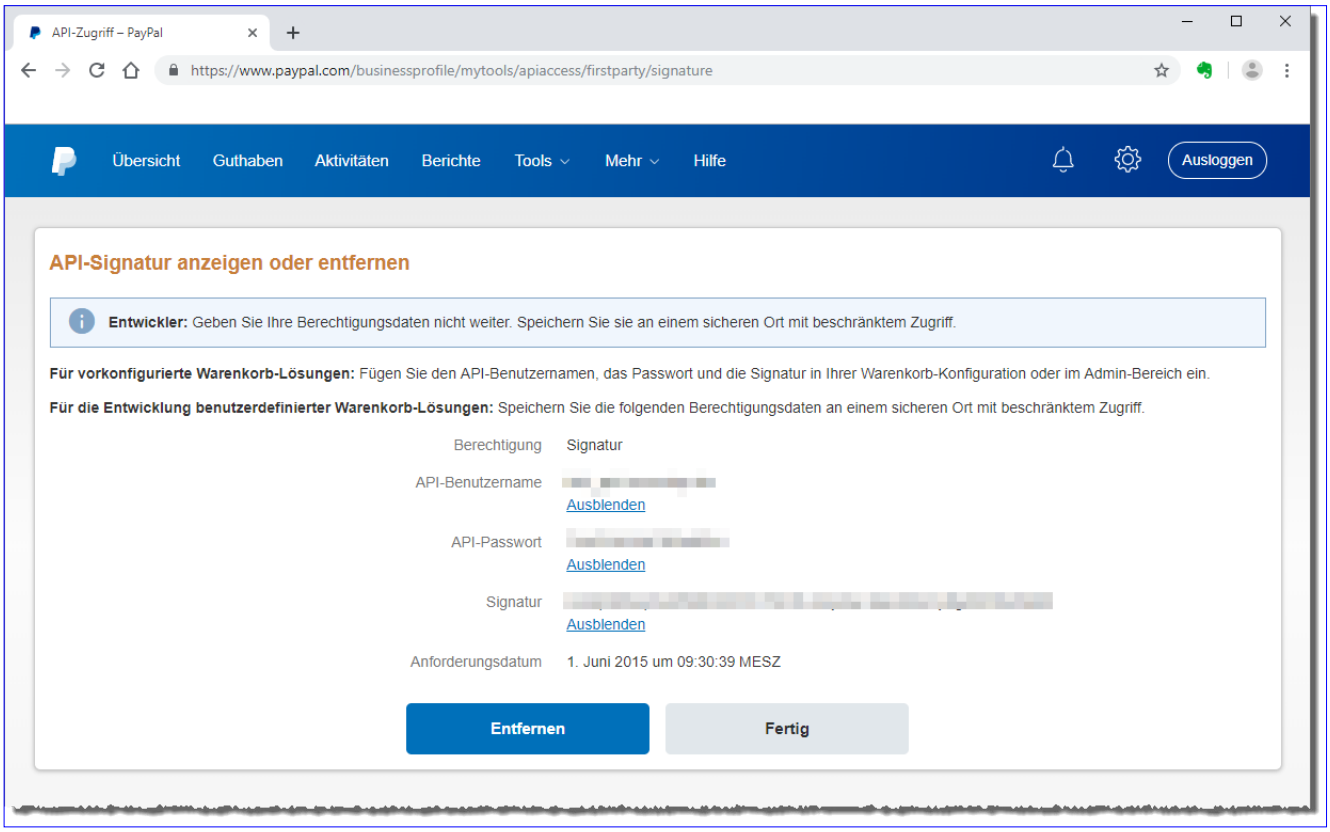


Bild 1: Abrufen der Zugangsdaten

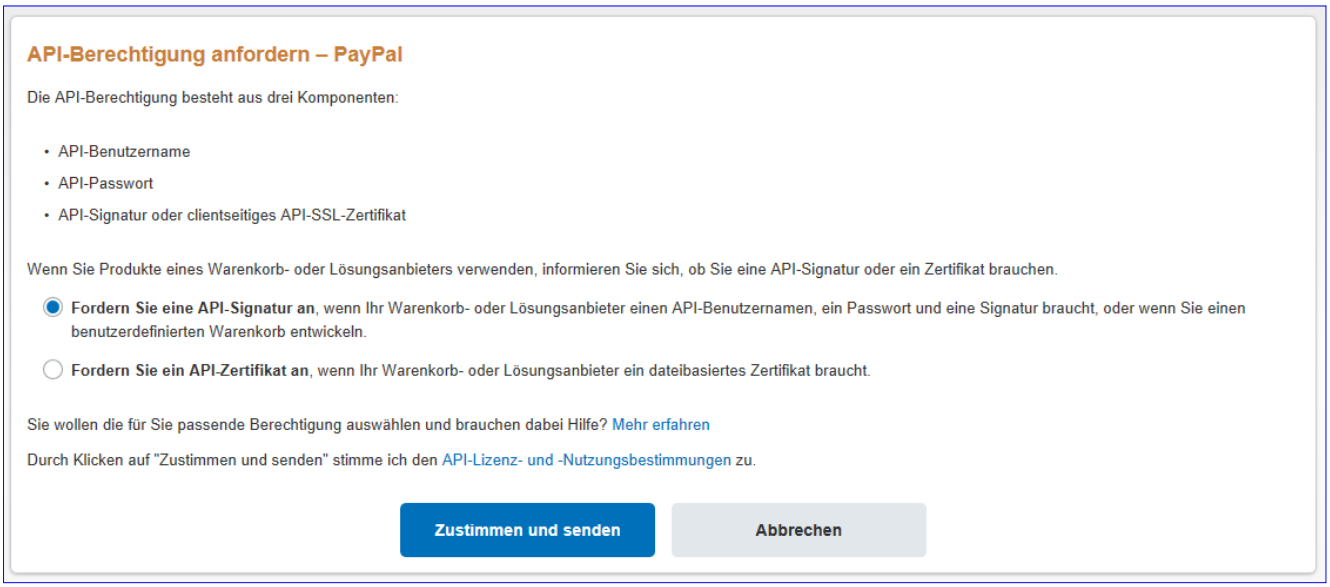


Bild 2: Anfordern der API-Berechtigung

Wenn Sie die Daten noch nicht angefordert haben, können Sie diese nun anfordern. Dann sieht der Bereich wie in Bild 2 aus. Hier wähle Sie die erste Option zum Anfordern einer API-Signatur aus. Die neu angeforderten Daten werden dann wie zuvor beschrieben angezeigt.

Projekt anlegen

Wir verwenden für den Zugriff auf das PayPal-Konto ein Projekt auf Basis der Vorlage **Visual Basic|Windows Desktop|WPF-App**.

Klassenmodell für den Zugriff auf die API

Nun wollen wir ein Klassenmodell entwickeln, mit dem wir auf den Webservice von PayPal zugreifen können. Dazu öffnen Sie mit dem Befehl **Hinzufügen|Dienstverweis** des Kontextmenüs des Projekt-Elements im Projektmappen-Explorer den Dialog **Dienstverweis hinzufügen**. Hier tragen unter Adresse den folgenden Wert ein:

```
https://www.paypalobjects.com/wsdl/PayPalSvc.wsdl
```

Dann klicken Sie auf die Schaltfläche **Gehe zu** und warten einige Augenblicke. Dann erscheint in der Liste **Dienste** der Eintrag **PayPalAPIInterfaceService** mit zwei Untereinträgen (siehe Bild 3). Wenn Sie auf einen der Untereinträge klicken, erschei-

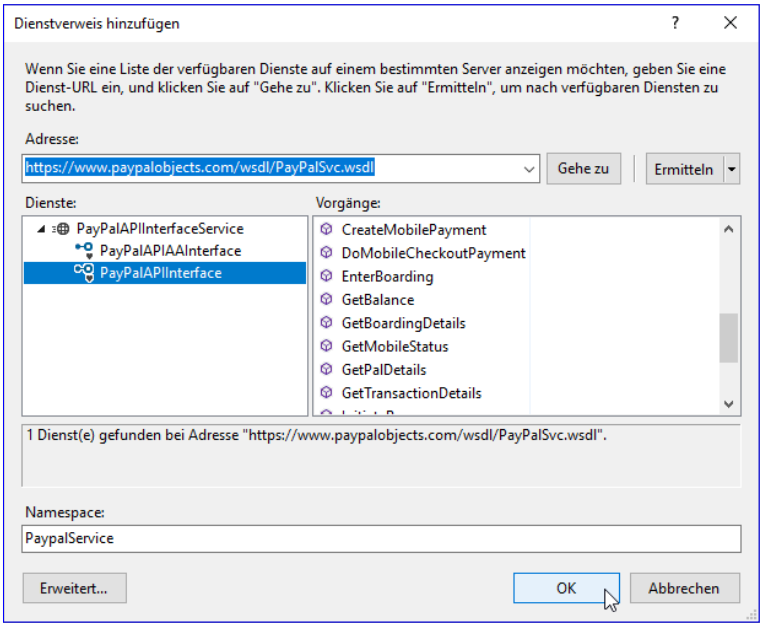


Bild 3: Hinzufügen eines Dienstverweises

nen unter **Vorgänge** einige API-Befehle. Wir tragen unten in das Feld **Namespace** den Wert **PayPalService** ein. Danach können Sie Klassen für den Zugriff auf den Webservice mit einem Klick auf die Schaltfläche **OK** erstellen lassen.

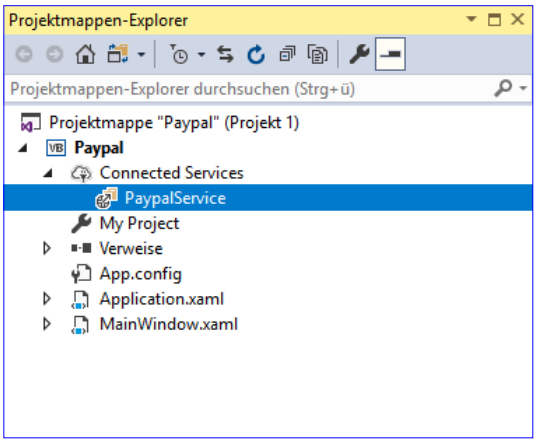


Bild 4: Der PayPal-Service im Projektmappen-Explorer

Den PayPal-Service finden wir danach im Projektmappen-Explorer in einem neuen Bereich namens **Connected Services** vor (siehe Bild 4). Hier brauchen wir keine weiteren Schritte vorzunehmen, wir nutzen gleich einfach die Befehle des Service.

Webservice auf Produktiv-Konto umstellen

Beim Einrichten des Webservice wurden in der Datei **App.config** einige Einstellungen vorgenommen. Darunter finden Sie im Bereich **client** auch zwei Elemente namens **endpoint**. Diese enthalten aktuell die Adresse der Sandbox-Zugänge (siehe Bild 5).

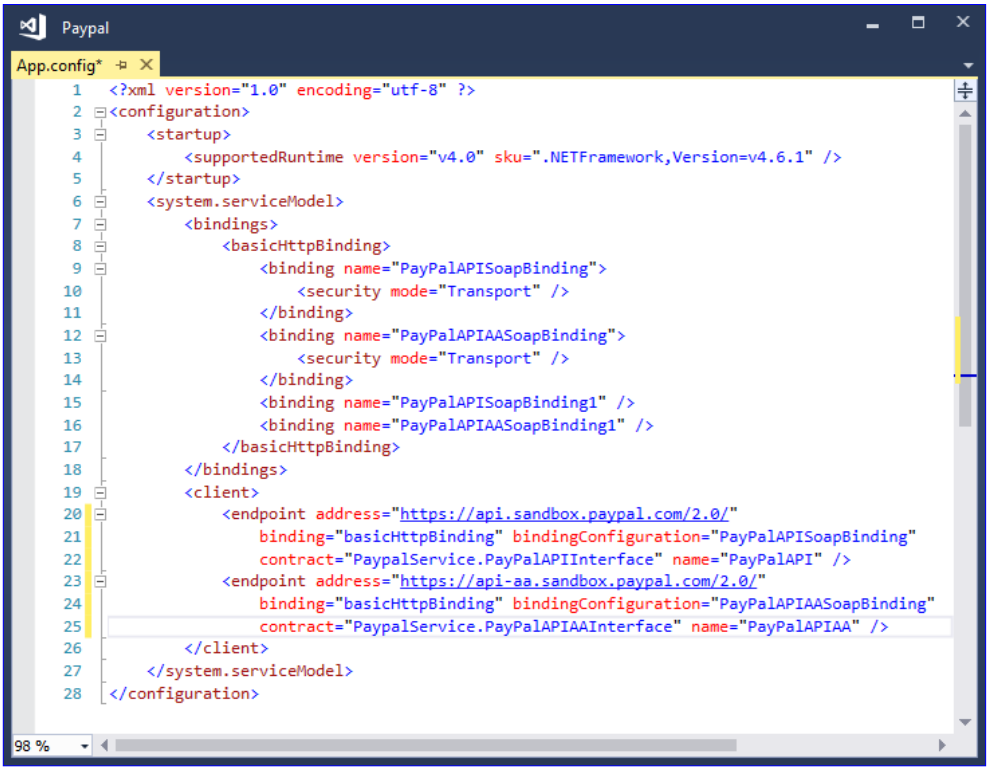


Bild 5: Einstellungen für den PayPal-Webservice in der Datei **App.config**

Wenn Sie ein Sandbox-Konto eingerichtet haben oder einrichten möchten, können Sie damit arbeiten. Wir wollen diese Schritte an dieser Stelle aussparen und direkt auf das Produktiv-Konto zugreifen.

Dazu ersetzen Sie für diese Elemente die Werte für das Attribut **address** mit den folgenden Werten:

```
https://api-3t.paypal.com/2.0/
https://api-aa-3t.paypal.com/2.0
```

Beachten Sie, dass Sie sich die Adressen für die beiden **endpoint**-Elemente geringfügig unterscheiden – genau wie die Werte der **name**-Attribute. Das zweite enthält jeweils die Buchstaben **aa**. Ersetzen Sie also so, dass das Ergebnis anschließend wie folgt aussieht:

```
<endpoint address="https://api-3t.paypal.com/2.0/" binding="basicHttpBinding" bindingConfiguration="PayPalAPISoapBinding"
    contract="PaypalService.PayPalAPIInterface" name="PayPalAPI" />
<endpoint address="https://api-aa-3t.paypal.com/2.0/" binding="basicHttpBinding"
    bindingConfiguration="PayPalAPIAASoapBinding" contract="PaypalService.PayPalAPIAAInterface" name="PayPalAPIAA" />
```

Schaltfläche für den Abruf des aktuellen Saldos

Nun fügen wir dem Fenster **MainWindow.xaml** eine Schaltfläche namens **btnSaldo** hinzu und fügen dieser über das Attribut **Click** eine neue Ereignismethode namens **btnSaldo_Click** hinzu:

```
<Button x:Name="btnSaldo" Content="Saldo abrufen" ... Click="btnSaldo_Click"/>
```

Die Ereignismethode im Code behind-Modul **MainWindow.xaml.vb** ist noch leer, wird aber gleich gefüllt:

```
Private Sub btnSaldo_Click(sender As Object, e As RoutedEventArgs)

End Sub
```

Zugriffsdaten als Konstanten abspeichern

Nun hinterlegen wir zunächst die Zugriffsdaten für den PayPal-Account als Konstanten im Kopf des Code behind-Moduls:

```
Class MainWindow
    Const cStrBenutzername As String = "********************"
    Const cStrKennwort As String = "******************"
    Const cStrSignatur As String = "***************************************************"
    ...
End Class
```

Saldo abrufen

Die Methode **btnSaldo_Click** füllen wir nun mit den benötigten Anweisungen zum Abrufen des Saldos auf. Als Erstes deklarieren wir einige Variablen:

```
Private Sub btnSaldo_Click(sender As Object, e As RoutedEventArgs)
    Dim objPayPalAPIInterfaceClient As PaypalService.PayPalAPIInterfaceClient
    Dim objCustomSecurityHeaderType As PaypalService.CustomSecurityHeaderType
    Dim objUserIDPasswordType As PaypalService.UserIdPasswordType
    Dim objBalanceReq As PaypalService.GetBalanceReq
    Dim objGetBalanceResponseType As PaypalService.GetBalanceResponseType
```

Danach füllen wir einige Objektvariablen mit den entsprechenden Verweisen auf neu erstellte Objekte, darunter auch das Objekt **objUserIDPasswordType** (wir haben die Objekte der Übersicht halber nach dem Objekttyp mit vorangestelltem **obj** benannt).

Darunter auch das Objekt **objUserIDPasswordType**, dem wir über die Eigenschaften **Username**, **Password** und **Signature** die entsprechenden Daten aus den zuvor festgelegten Konstanten zuweisen:

```
    objPayPalAPIInterfaceClient = New PaypalService.PayPalAPIInterfaceClient
    objCustomSecurityHeaderType = New PaypalService.CustomSecurityHeaderType
    objUserIDPasswordType = New PaypalService.UserIdPasswordType
    With objUserIDPasswordType
        .Username = cStrBenutzername
        .Password = cStrKennwort
        .Signature = cStrSignatur
    End With
    objCustomSecurityHeaderType.Credentials = objUserIDPasswordType
```

Danach erstellen wir ein **GetBalanceReq**-Objekt, dem wir ein neues **GetBalanceRequestType**-Objekt zuweisen sowie die Version des zu verwendenden Requests (hier **204.0**):

```
    objBalanceReq = New PaypalService.GetBalanceReq
    With objBalanceReq
        .GetBalanceRequest = New PaypalService.GetBalanceRequestType
        .GetBalanceRequest.Version = "204.0"
    End With
```

Das **GetBalanceReq**-Objekt übergeben wir dann dem Aufruf der Methode **GetBalance** als zweiten Parameter – der erste ist der Parameter **objCustomSecurityHeaderType** mit den Zugangsdaten:

```
    objGetBalanceResponseType = objPayPalAPIInterfaceClient.GetBalance(objCustomSecurityHeaderType, objBalanceReq)
```

Schließlich geben wir den Wert der **Balance**-Eigenschaft und der **Balance.currencyID**-Eigenschaft in einem Meldungsfenster aus:

```
    MessageBox.Show(objGetBalanceResponseType.Balance.Value.ToString & " " _
        & objGetBalanceResponseType.Balance.currencyID.ToString)
End Sub
```

Umsätze ausgeben

Neben dem Kontostand ist für uns natürlich interessant, wie dieser zustande kommt – und dazu lesen wir die Umsätze ein.

Dazu verwenden wir als grundlegendes Beispiel die folgende Prozedur, die durch einen Klick auf die Schaltfläche **btnUmsaetze** ausgelöst wird:

```
Private Sub btnUmsaetze_Click(sender As Object, e As RoutedEventArgs)
```

Onlinebanking mit DDBAC: Saldo und Umsätze

Onlinebanking ist aktuell in aller Munde, da es für erhöhte Sicherheit einige Änderungen in den Abläufen gegeben hat. Viele Leser kennen die DDBAC-Bibliothek bereits, denn wir haben in Access im Unternehmen schon darüber berichtet und es ist auch ein Buch zu diesem Thema im André Minhorst Verlag erschienen. Wer diese Bibliothek nutzt, braucht neben einer aktuellen Version auch noch eine kleine Änderung, ohne die Vorgänge nicht mehr durchgeführt werden können. Auch aus diesem Anlass wollen wir in diesem Artikel das Thema Homebanking mit Visual Basic und der DDBAC-Bibliothek einmal im Detail vorstellen.

Voraussetzungen

Voraussetzung für das Umsetzen der Lösung dieses Artikels ist das Vorhandensein einer DDBAC-Lizenz. Die damit nutzbaren Komponenten sind leider nicht mehr kostenlos verfügbar, sondern müssen lizensiert werden, bevor man diese in seine Produkte einbauen kann. Lizenzen finden Sie im Onlineshop unter **https://shop.minhorst.com/access-tools/295/ddbac-jahreslizenz?c=78**.

Download und Installation

Nach dem Erwerben einer Lizenz erhalten Sie einen Download in ihrem Kundenkonto. Dort finden sie eine Datei namens **DDBACNetWrapper-Version-5-7-65-0.zip** (oder neuer), die Sie entpacken. Darin enthalten ist eine **DDBACSDK.exe**, die Sie auf Ihrem Rechner installieren. Anschließend finden Sie im Verzeichnis **C:\Program Files (x86)\DataDesign\DDBACSDK** die installierten Elemente.

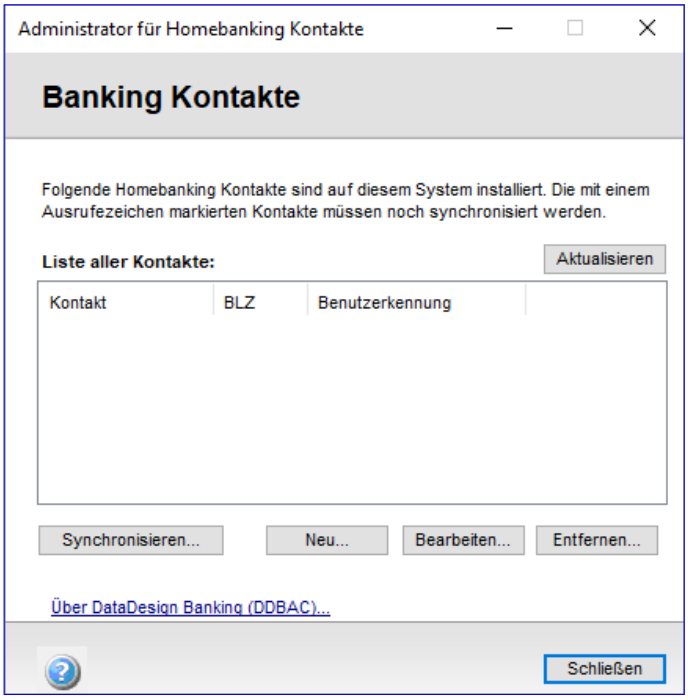


Bild 1: Verwaltung der Homebanking-Kontakte

Homebanking-Kontakte anlegen

Nachdem Sie die DDBAC-Komponenten installiert haben, finden Sie in der Systemsteuerung einen neuen Eintrag namens **Homebanking Administrator (32-bit)** vor. Mit diesem können Sie einen sogenannten Homebanking-Kontakt anlegen, was gleichbedeutend mit der Verbindung zu einer Bank für ein spezielles Konto ist. Nach dem Start erscheint der Administrator für Homebanking-Kontakte zunächst wie in Bild 1.

Um eine neue Bankverbindung hinzuzufügen, klicken Sie auf die Schaltfläche **Neu...** und finden den Dialog aus Bild 2 vor. Geben Sie nun die Bankleitzahl der gewünschten Bank ein, wird im Auswahlfeld darunter direkt die passende Bank angezeigt. Die beiden Optionen unterhalb des Auswahlfeldes können Sie ignorieren. Klicken Sie dann auf **Weiter >**, um diesen Schritt abzuschließen.

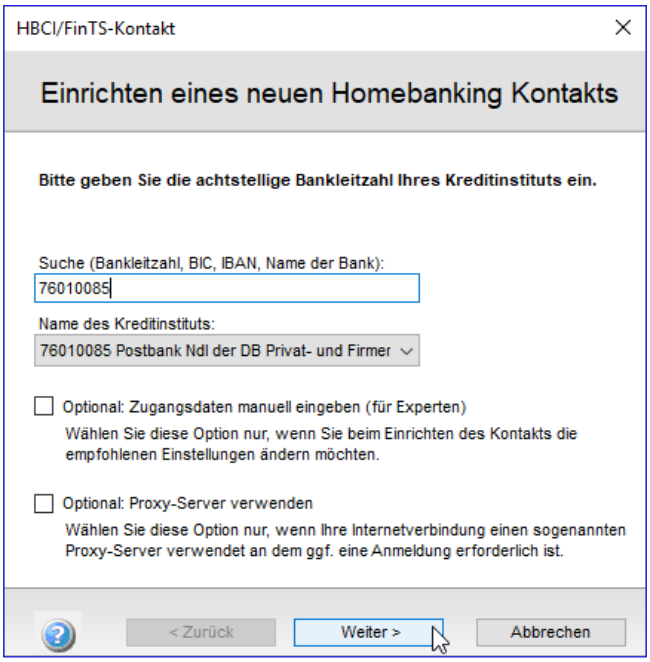


Bild 2: Einrichten eines neuen Homebanking-Kontakts

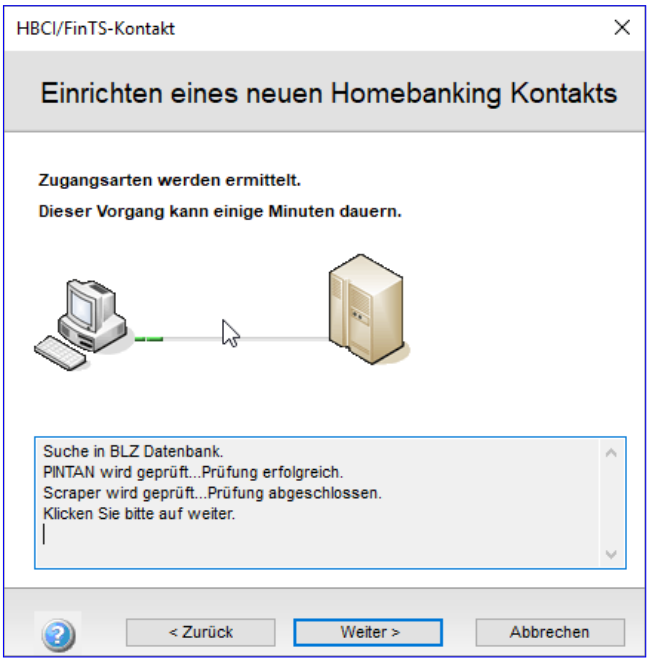


Bild 3: Ermitteln weiterer Informationen

Die nächste Seite des Dialogs bestätigt, das Zugangsarten ermittelt wurden (siehe Bild 3).

Damit geht es dann weiter zur Auswahl der Zugangsart. In diesem Beispiel wird nur PIN/TAN angeboten, was aber üblich ist (siehe Bild 4).

Danach wird es interessant, denn hier stellt sich heraus, ob Ihnen die richtigen Daten für den Zugriff auf Ihr Konto vorliegen. Im Fall der Postbank benötigen wir zunächst die Postbank-ID, bei anderen Kreditinstituten heißt die zu verwendende ID anders.

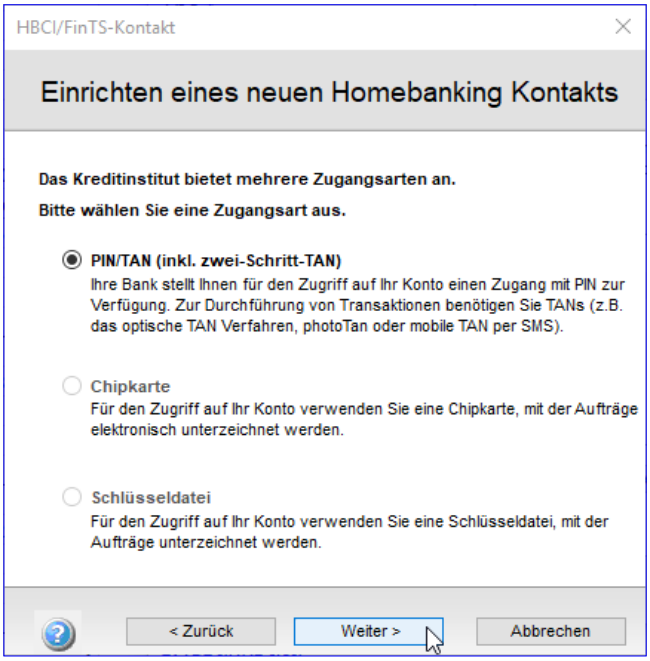


Bild 4: Auswahl der Zugangsart

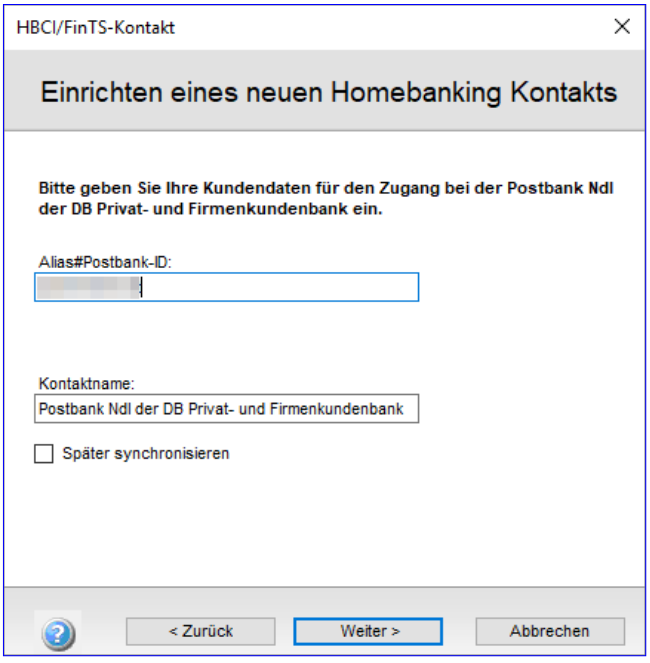


Bild 5: Eingabe der Kundendaten für den Zugang

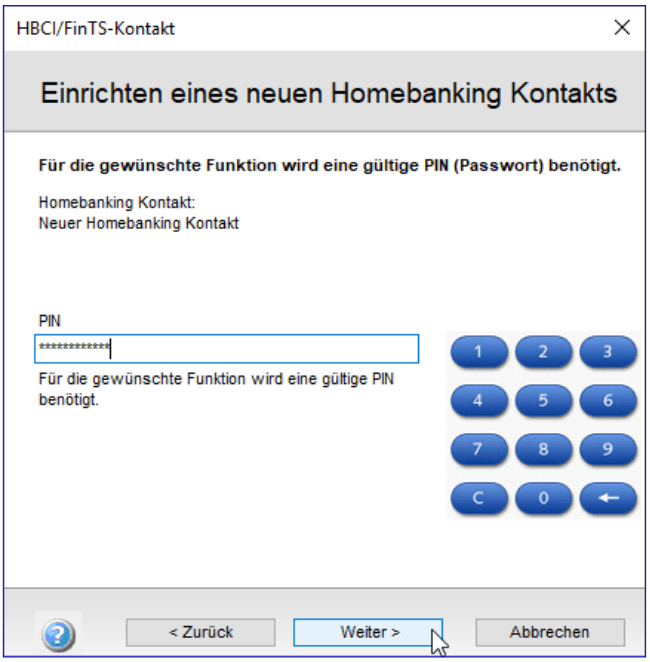


Bild 6: Eingabe des Kennworts

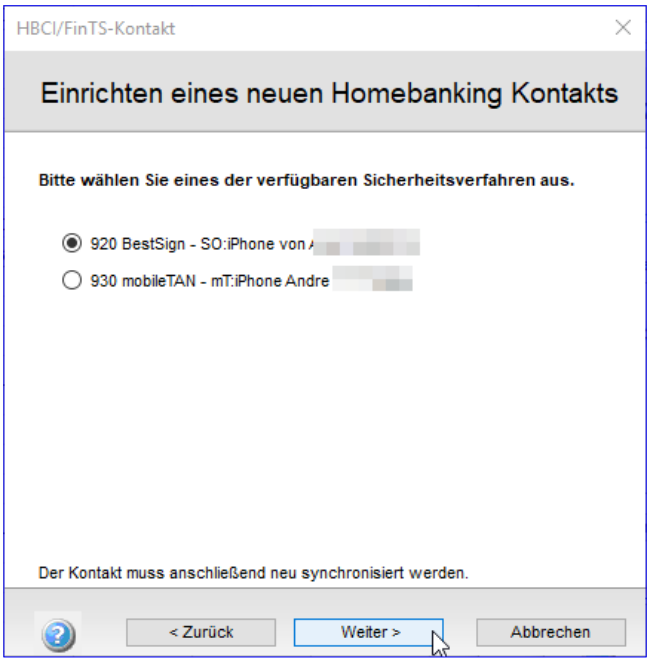


Bild 7: Synchronisierung der Zugangsdaten

Geben Sie diese im Dialog aus Bild 5 ein und passen Sie gegebenenfalls noch den Wert des Feldes **Kontaktname** auf der gleichen Seite an.

Danach fehlt nur noch die Eingabe des PIN beziehungsweise des Kennworts für diesen Onlinebanking-Zugang, den Sie im nächsten Schritt eingeben (siehe Bild 6). Lassen Sie sich nicht vom Zahlenblock irritieren, es kann auch sein, dass Ihr PIN alphanumerische Zeichen enthält.

Ein Klick auf **Weiter >** liefert den nächsten Schritt des Assistenten, wo die Synchronisation durchgeführt wird – diesen Dialog haben wir nicht abgebildet. Klicken Sie nochmals auf **Weiter >**, erscheint der Dialog aus Bild 7. Hier können Sie nun eines der verfügbaren Sicherheitsverfahren auswählen. Bei der Postbank heißt ein Verfahren beispielsweise BestSign. Sie können aber hier auch mobileTAN auswählen. Schließen Sie den Dialogschritt dann mit der Schaltfläche **Weiter >** ab.

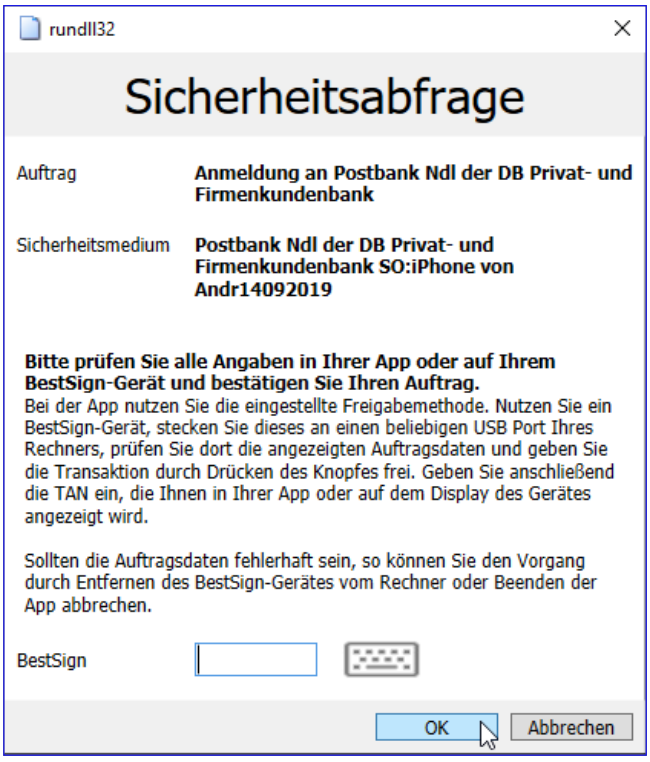


Bild 8: Kontaktaufnahme via BestSign

Nun folgt der erste interaktive Schritt: Das ausgewählte Sicherheitsverfahren wird genutzt, um die Synchronisierung abzuschließen. Dazu erhalten Sie je nach gewähltem Sicherheitsverfahren beispielsweise eine Anfrage an Ihr Smartphone geschickt, die Sie auffordert, sich an der Homebanking-App auf dem Smartphone anzumelden und eine von dort generierte

PIN in einem Dialog wie dem namens **Sicherheitsabfrage** aus Bild 8 einzugeben. Damit ist die Synchronisierung auch abgeschlossen und Sie können diesen Bankkontakt nun nutzen, um mit der DDBAC-Bibliothek zu experimentieren.

Wo werden die ermittelten Daten gespeichert?

Die hier von Ihnen eingegebenen und von der Bank ergänzten Daten werden auf Ihrem Rechner gespeichert, und zwar in der Regel im folgenden Verzeichnis:

```
C:\Users\<Benutzername>\AppData\Roaming\DataDesign\DDBAC.
```

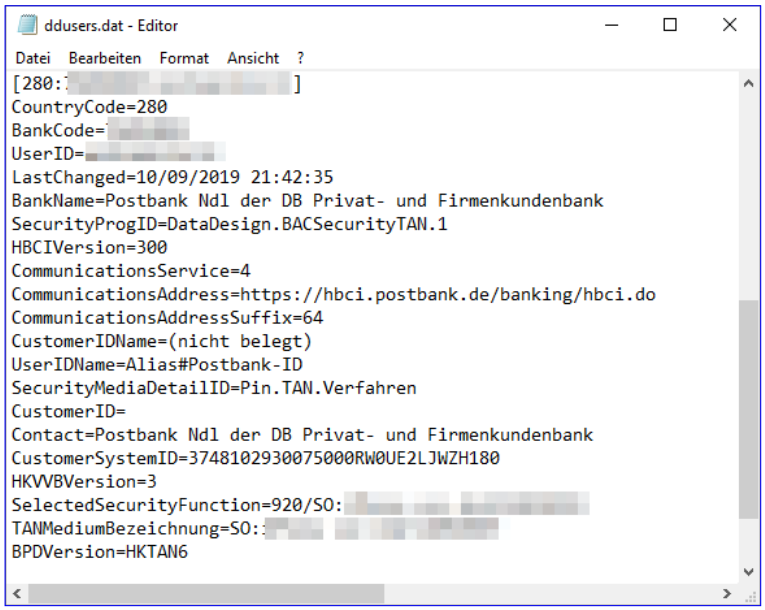


Bild 9: Textdatei mit den Konfigurationsdaten

Die Datei sieht wie in Bild 9 aus. Es kann auch sein, dass die Daten in einem anderen Verzeichnis liegen, dann ist dieses in der Registry unter einem der folgenden Schlüssel angegeben, und zwar unter dem Eintrag **DataDir**:

```
HKEY_CURRENT_USER\Software\DataDesign\DDBAC
HKEY_LOCAL_MACHINE\Software\DataDesign\DDBAC
```

Um die Registry nach diesen Werten zu durchsuchen, öffnen Sie diese mit dem Befehl **RegEdit** und navigieren dann zum entsprechenden Element. Warum müssen Sie diese Datei kennen? Weil Sie diese, wenn Sie einmal einen neuen Rechner aufsetzen und dort die Homebanking-Kontakte weiter nutzen möchten, einfach auf den neuen Rechner kopieren können.

Was kann man mit DDBAC anstellen?

Die DDBAC-Bibliothek bietet eine ganze Reihe möglicher Vorgänge an, zu denen die folgenden gehören:

- Ermittlung von Kontoständen
- Abruf von Kontoumsätzen
- Durchführung von SEPA-Überweisungen
- Durchführen von Terminüberweisungen
- Anlegen von Daueraufträgen
- Durchführen von SEPA-Lastschriften

- Durchführen von SEPA-Sammellastschriften

Wir werden uns in diesem Artikel und in Fortsetzungen den einen oder anderen Vorgang ansehen.

Objektmodell der DDBAC

Die DDBAC-Komponente **DataDesign DDBAC HBCI Banking Application Components** liefert die wichtigsten Objekte:

- **BACBanking**: Dies ist das Hauptobjekt beim Arbeiten mit der DDBAC-Bibliothek. Sie können damit auf die Onlinebanking-Kontakte zugreifen (**BACCustomer**) und diese anlegen oder löschen, den über die Systemsteuerung verfügbaren Dialog zum Verwalten von Kontakten öffnen oder Optionen einstellen.

- **BACCustomer**: Das **BACCustomer**-Objekt repräsentiert einen Kontakt, also eine Kunde-Bank-Beziehung. Die Identifizierung eines solchen Kontaktes erfolgt über die Länderkennung (**280** für Deutschland), die Bankleitzahl und die UserID. Letztere kann mit einer Kontonummer identisch sein, allerdings kann es zu jedem Kontakt auch mehrere verschiedene Konten geben – die wiederum verschiedenen Typs sein können.

- **BACDialog**: Dieses Objekt stellt die Verbindung zwischen Client und Server her, wobei Ihre Anwendung dem Client und der Bankserver dem Server entspricht. Mit dem **BACDialog**-Objekt können Sie den Dialog starten und beenden und Nachrichten oder Segmente mit dem Bankserver austauschen.

- **BACMessage**: Dieses Objekt liefert die Antwort auf die Ausführung eines Geschäftsvorgangs und enthält weitere Objekte mit den benötigten Informationen.

- **BACTransaction**: Das **BACTransaction**-Objekt liefert die eigentlichen Transaktions-Informationen, also beispielsweise den aktuellen Kontostand oder die Konto-Umsätze.

- **BACSegment**: Das **BACSegment**-Objekt ist ein Objekt, das einem HBCI-Datensatz entspricht. Sie können diesem die Parameter zuweisen, die das Segment beinhalten soll.

Testen mit LINQPad

Die folgenden Experimente können Sie auch in Visual Studio innerhalb eines Projects durchführen, aber um ein Gefühl für die einzelnen Elemente der Bibliothek zu bekommen, wollen wir mit dem Tool **LINQPad** arbeiten. Dieses erlaubt die schnelle Programmierung und Ausführung von Visual Basic-Prozeduren und Sie benötigen nicht immer so viel Zeit, um eine Anwendung zum Debuggen zu starten. In LINQPad wählen wir zunächst als **Language** den Wert **VB Program** aus (siehe Bild 10).

Dann fügen wir einen Verweis auf die benötigte Bibliothek hinzu. Dazu betätigen Sie die Taste **F4**. Es erscheint der Dialog **QueryProperties**, mit dem Sie die Verweise einstellen können, die für den Code im aktuellen Fenster zur Verfügung stehen sollen. Unter **Add...** finden wir die gewünschte Bibliothek nicht, aber wenn Sie auf **Browse...** klicken und im folgenden Verzeichnis suchen, werden wir fündig (siehe Bild 11):

```
C:\Program Files (x86)\Common Files\DataDesign\DDBAC
```

Hier fügen wir die Datei **Interop.BankingApplicationComponents.dll** hinzu und schließen den **Datei öffnen**-Dialog.

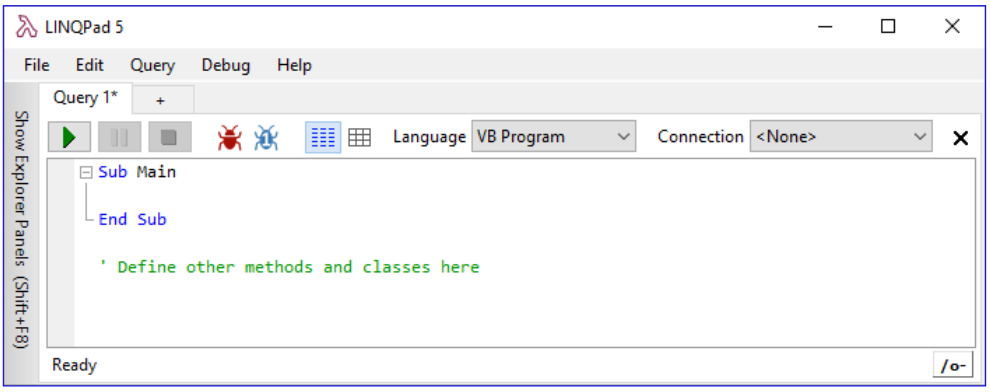


Bild 10: Testen mit LINQPad

Um nun noch das Äquivalent einer **Imports**-Anweisung hinzuzufügen, damit wir beim Deklarieren und Instanzieren nicht immer den Namespace vor den Klassennamen angeben müssen, wechseln wir im Dialog **Query Properties** zur Registerseite **Additional Namespace Imports** und tragen dort folgende Zeile ein (siehe Bild 12):

```
BankingApplicationCompo-
nents
```

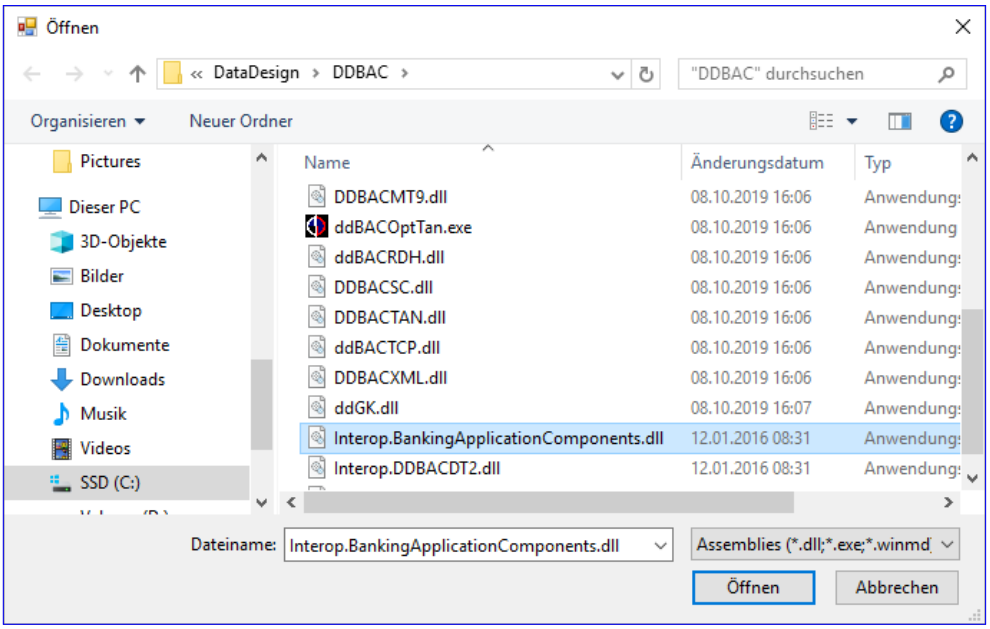


Bild 11: Auswählen der richtigen DLL als Verweis

Außerdem benötigen wir noch diese Namespaces:

```
Microsoft.VisualBasic
Microsoft.Win32
```

Nun können wir im Query-Fenster von LINQPad bereits auf die Elemente der **DDBAC**-Bibliothek zugreifen. Und da wir auch schon einen Homebanking-Kontakt angelegt haben, können wir auch gleich richtig loslegen.

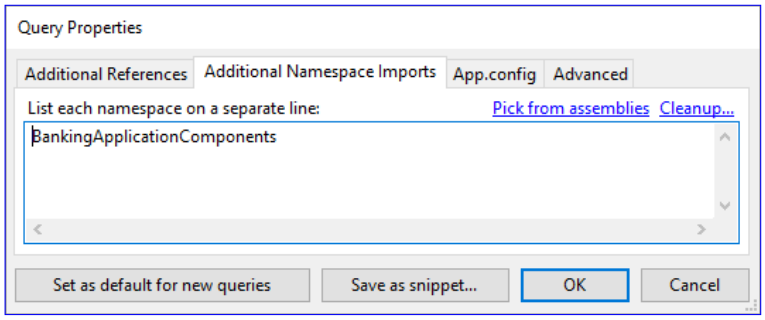


Bild 12: Ersatz für eine **Imports**-Anweisung

BACBanking-Objekt schnell verfügbar machen

Da wir das **BACBanking**-Objekt öfter benötigen werden, schreiben wir eine kleine Funktion, die uns schnell eine neue Instanz liefert, wenn wir eine brauchen. Haben wir einmal eine Instanz erstellt, soll diese in einer Variablen gespeichert werden und dann soll die Funktion auf Anfrage diese Instanz liefern. Die Variable und die Funktion füge Sie einfach hinter der noch leeren Prozedur **Sub Main** ein:

```
Private m_Banking As BACBanking

Public Function GetBanking() As BACBanking
    If m_Banking Is Nothing Then
        m_Banking = New BACBanking
    End If
    GetBanking = m_Banking
End Function
```

Damit starten wir dann zum Beispiel schnell den Dialog zum Verwalten der Homebanking-Kontakte per Code, indem wir mit der Funktion **GetBanking** einen Verweis auf ein Objekt des Typs **BACBanking** holen und dessen Methode **RunContactsCPL** aufrufen:

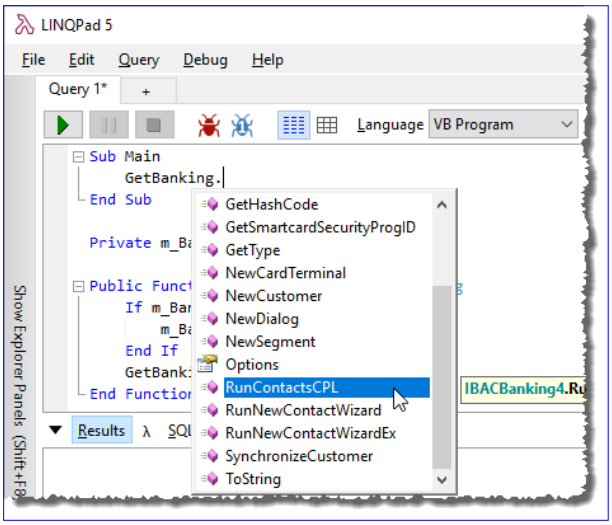


Bild 13: IntelliSense für **BACBanking**

```
Sub Main
    GetBanking.RunContactsCPL
End Sub
```

Beim Eingeben des Befehls zeigt IntelliSense sogar alle Methoden und Eigenschaften des Objekts an (siehe Bild 13). Hier sind die wichtigsten Eigenschaften des **BACBanking**-Objekt:

- **Customers**: Liefert eine Liste der Klasse **BACCustomers**, mit der Sie auf die **BACCustomer**-Objekte zugreifen können. Damit lassen sich die Daten des **BACCustomer**-Objekts ausgeben, zum Beispiel die Bankleitzahl:

```
Debug.Print (GetBanking.Customers.Count) 'Anzahl
Debug.Print (GetBanking.Customers.Item(1).BankCode) 'Bankleitzahl
```

- **Options**: Gibt die Liste der Optionen aus, zum Beispiel für die Version:

```
Debug.Print(GetBanking.Options("Version"))
```

- **DeleteCustomer**: Erwartet den **CountryCode**, den **BankCode** und die **UserID** als Parameter, um den angegebenen Benutzer zu löschen.
- **NewCustomer**: Erwartet **CountryCode**, **BankCode** und **UserID** und legt damit einen neuen Benutzer an. Das Ergebnis ist ein **BACCustomer**-Objekt.
- **NewDialog**: Erstellt ein neues **BACDialog**-Objekt. Wir nutzen dafür aber die **NewDialogUI**-Methode.
- **NewSegment**: Erstellt nach Angabe des Segmenttyps und wahlweise der Version ein neues **BACSegment**-Objekt.