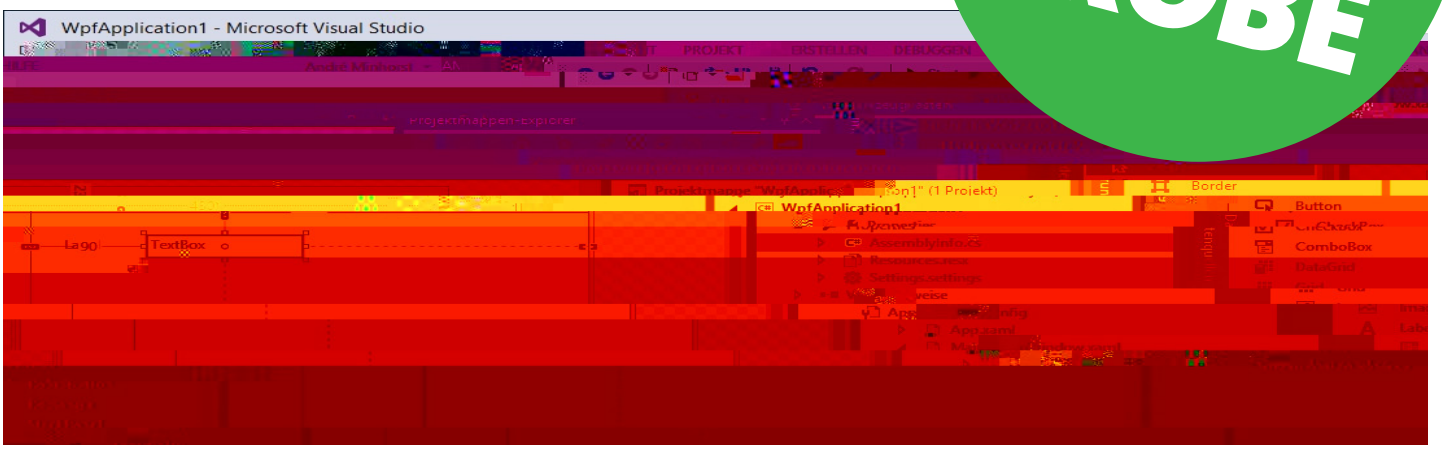
AUSGABE 01/2016

DATENBANK ENTWICKLER

Gratis LESE-PROBE

MAGAZIN FÜR DIE DATENBANKENTWIC
VISUAL STUDIO FÜR DESKTOP, WEB UN



TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER
© André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

Von VBA zu C#: Das Static-Schlüsselwort

Was bedeutet eigentlich das Schlüsselwort »static«, mit dem Sie sowohl Typen/Klassen also auch die Elemente einer Klasse wie Methoden oder Eigenschaften kennzeichnen können – und was ist bei der Erstellung und Nutzung von Typen und Elementen mit diesem Schlüsselwort zu beachten? Dies schauen wir uns im vorliegenden Artikel an und liefern einige Beispiele, welche die Regeln verdeutlichen.

Statische Klassen unter VBA

Unter VBA waren schon einige Verrenkungen nötig, wenn Sie auf die Methoden oder Eigenschaften einer Klasse zugreifen wollten, ohne diese Klasse selbst zu instanzieren – also beispielsweise so:

```
Dim objTest As clsTest
Set objTest = New clsTest
objTest.Beispielmethode
```

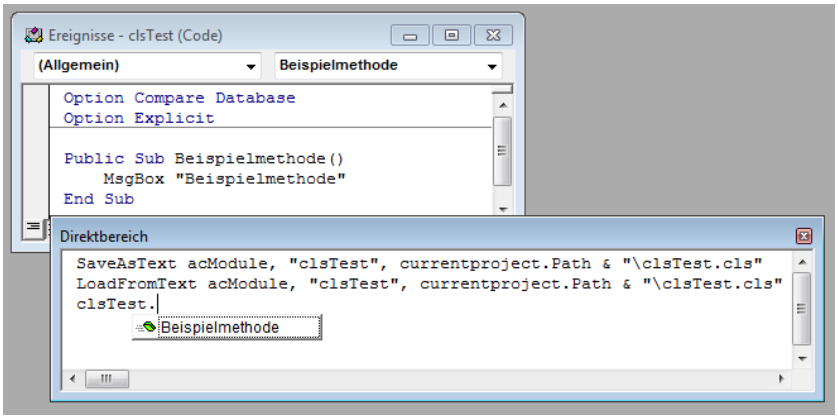


Bild 1: »Statische« Klassen unter VBA

Mit ein paar Tricks konnten Sie eine Klasse mit einer zusätzlichen, im VBA-Editor nicht sichtbaren Eigenschaft ausstatten, die es ermöglichte, direkt auf ihre Methoden zuzugreifen. Das sah dann schlicht so aus:

```
clsTest.Beispielmethode
```

Der Trick ist, die Klasse per **SaveAsText** zu exportieren und dann den Wert des Attributs **VB_PredeclaredId** in einem Texteditor auf **True** einzustellen:

```
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Compare Database
Option Explicit

Public Sub Beispielmethode()
    MsgBox "Beispielmethode"
End Sub
```

Danach konnten Sie in VBA einfach den Namen der Klasse gefolgt von einem Punkt angeben, um per IntelliSense die Member der Klasse einzublenden (siehe Bild 1). Sie konnten die Klasse aber weiterhin instanzieren und wie eine herkömmliche Klasse per Objektverweis nutzen.

Zum Glück waren die beschriebenen Schritte nur notwending, wenn Sie die gewünschte Funktion tatsächlich im Kontext einer Klasse benötigten – also etwa, um Eigenschaften einzugeben und dann eine oder mehrere Methoden aufzurufen. Für den direkten Aufruf einer Methode (beziehungsweise Prozedur, wie es unter VBA heißt) gibt es ja zum Glück das gute, alte Standardmodul.

Statische Klassen unter C#

Unter C# ist dieses Feature standardmäßig vorgesehen, und zwar in verschiedenen Stufen. Dabei hilft Ihnen das Schlüsselwort **static**, das Sie dem Namen einer Klasse oder auch einer Methode, Variable oder anderen Membern voranstellen können – allerdings nicht in beliebigen Kombinationen.

Von VBA zu C#: Steuerelement-Ereignisse

Wer unter VBA Ereignisprozeduren für Formulare, Berichte oder Steuerelemente erstellen wollte, hatte es leicht: Einfach in der Ereigniseigenschaft den Wert [Ereignisprozedur] auswählen, auf die Schaltfläche mit den drei Punkten klicken und schon konnte man die vorgefertigte Prozedur mit Anweisungen füllen. In manchen Fällen war das Implementieren von Ereignissen schon komplizierter, aber immer noch schnell machbar. Unter C# bekommt man dies auch meist mit wenigen Klicks hin, aber die notwendigen Handgriffe unterscheiden sich doch deutlich von denen unter VBA. Dieser Artikel zeigt, wie Sie gängige Varianten von Ereignisprozeduren anlegen.

Unter einem Ereignis verstehen wir dabei etwas, das durch eine bestimmte Aktion ausgelöst wird. Unter Access steht dies meist in Zusammenhang mit einem Formular oder einem Steuerelement, zum Beispiel das Öffnen oder Schließen des Formulars, das Anklicken einer Schaltfläche oder das Ändern des Inhalts eines Textfeldes.

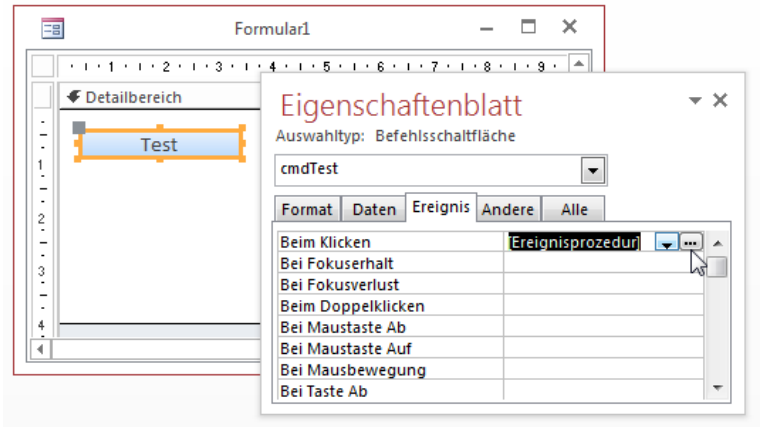


Bild 1: Anlegen einer Ereignisprozedur

Für die Implementierung einer Prozedur, die beim Eintreten des Ereignisses ausgelöst wird, sind dann zwei Schritte nötig: Erstens haben wir die entsprechende Ereigniseigenschaft aus dem Eigenschaftsfenster herausgesucht und dort den Wert **[Ereignisprozedur]** ausgewählt (siehe Bild 1).

Zweitens haben wir durch einen Klick auf die Schaltfläche mit den drei Punkten neben dem Eigenschaftswert die leere Prozedur angelegt, die dann wie in Bild 2 aussieht – die Ereignisprozedur. Wenn wir hier eine Anweisung wie **MsgBox "Test"** einfügen, in die Formularansicht wechseln und die Schaltfläche betätigen, wurde die Prozedur wie gewünscht ausgewählt.

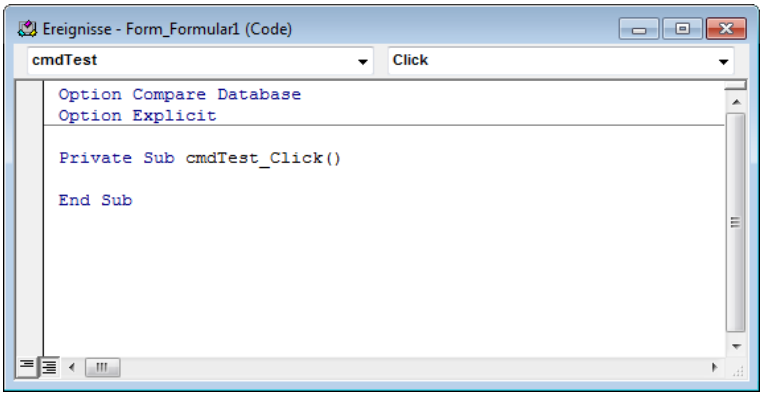


Bild 2: Eine neue Ereignisprozedur unter VBA/Access

Hier gibt es eine implizite Vereinbarung: Diese besagt, dass die Definition der Ereignisprozedur bestimmten Regeln entsprechen muss. Die erste bezieht sich auf den Namen, der aus dem Steuerelementnamen (oder Formular-/Berichtsnamen), einem Unterstrich sowie der englischen Version der Ereignisbezeichnung bestehen muss, hier beispielsweise **cmdText_Click**. Da es jedes Ereignis für jedes Objekt/Steuerelement nur einmal gibt, funktioniert dies sehr gut und

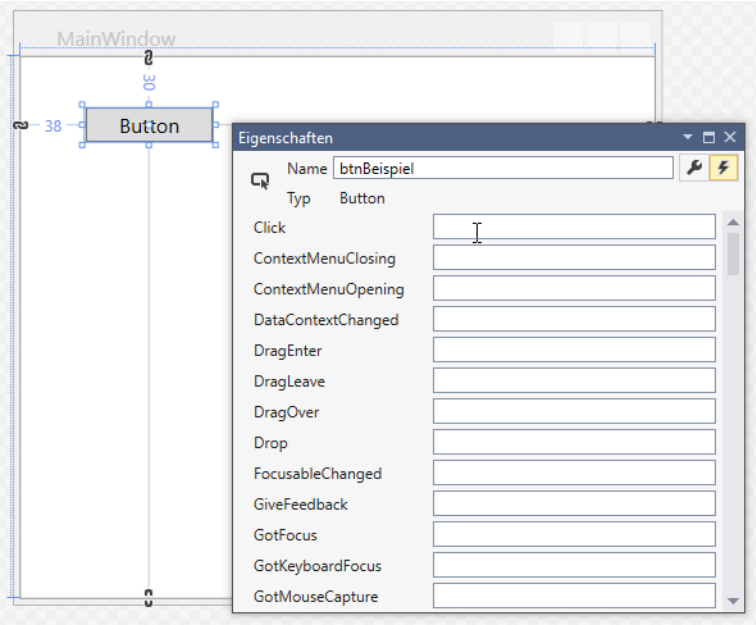


Bild 3: Anlegen einer Ereignisprozedur unter WPF

man braucht für die Ereigniseigenschaft nur den Wert **[Ereignisprozedur]** anzugeben statt etwa des Prozedurnamens.

Andersherum können Sie Ereignisprozeduren auch über den VBA-Editor anlegen. Dazu wählen Sie im linken Kombinationsfeld den Namen des Objekts aus, das Sie mit dem Ereignis ausstatten wollen, und im linken das entsprechende Ereignis. Der VBA-Editor legt dann sowohl die leere Ereignisprozedur an als auch den Eintrag in der entsprechenden Ereigniseigenschaft.

Fenster- und Steuerelementereignisse unter WPF

Unter WPF läuft es ähnlich ab. Für die folgenden Beispiele verwenden wir ein C#/WPF-Projekt namens **Ereignisse**. Um für eine Schaltfläche eine Ereignisprozedur zu implementieren, die beim Anklicken der Schaltfläche ausgelöst wird, erstellen Sie zunächst eine neue Schaltfläche, in diesem Fall direkt im Hauptfenster der Anwendung namens **MainWindow**.

Wir benennen die Schaltfläche um, indem wir das Attribut **name** des **button**-Objekts im XAML-Bereich mit dem gewünschten Namen versehen:

```
<Button x:Name="btnBeispiel" .../>
```

Neue Ereignisprozedur per Eigenschaftsfenster

Danach gibt es eine einfache Möglichkeit, wie unter Access/VBA gleichzeitig die Ereigniseigenschaft zu füllen als auch die Ereignisprozedur anzulegen. Dazu wechseln Sie im Eigenschaftsfenster, das standardmäßig die allgemeinen Eigenschaften für das ausgewählte Objekt anzeigt, auf den Bereich für die Ereignishandler. Dazu ist lediglich ein Klick auf die Schaltfläche mit dem Blitz-Symbol nötig (siehe Bild 3).

Hier klicken Sie dann einfach doppelt in die für uns interessante Eigenschaft, nämlich die mit der Bezeichnung **Click**. Das ist im Vergleich zu Access etwas weniger intuitiv, denn dort lässt sich der Wert **[Ereignisprozedur]** ja per Kombinationsfeld auswählen und die Prozedur per Klick auf die Schaltfläche mit den drei Punkten anlegen. Das Ergebnis ist, dass die Eigenschaft mit dem Wert **btnBeispiel_Click** gefüllt wird und eine neue, leere Methode in der Klasse **MainWindow** mit dem Code für das WPF-Fenster erscheint:

```
private void btnBeispiel_Click(object sender,
    RoutedEventArgs e) {
}
```

Die Methode wird bereits mit den benötigten Parametern ausgestattet (zu diesen kommen wir gleich). Der Unterschied zu Access/VBA ist nun, dass wir für die Ereigniseigenschaft nicht pauschal einen Wert wie **[Ereignisprozedur]** eintragen und der Name der durch das Ereignis ausgelösten Methode bestimmten Vorgaben entsprechen muss. Stattdessen könnten Sie auch jeden anderen Wert für die Eigenschaft **Click** eintragen, solange die auszulösende Methode entsprechend benannt wird. Aber wo speichert Visual Studio nun die Eigenschaft und den angegebenen Wert tatsächlich? Dies geschieht direkt in der XAML-Definition des Fensters, in diesem Fall mit dem Attribut **Click**:

Von VBA zu C#: Objekt-Ereignisse

Nicht nur Fenster und Steuerelemente, sondern auch Objekte, die nicht Bestandteil der Benutzeroberfläche sind, können Ereignisse auslösen. Schließlich können Sie auch selbst Klassen programmieren und diese mit Ereignishandlern versehen. Wie Sie unter C# mit diesen Möglichkeiten umgehen, zeigt dieser Artikel. Dabei schauen wir uns zunächst an, wie Sie Ereignisse für Objekte auf Basis etwa der OpenFileDialog-Klasse implementieren und erstellen dann eine benutzerdefinierte Klasse mit einem Ereignishandler.

Ereignisse von Klassen implementieren

Neben Formularen und Steuerelementen bieten unter Access auch verschiedene Klassen beziehungsweise Objekte Ereignisse an, die Sie selbst implementieren können. Als Beispiel sei etwa die Klasse **Application** anderer Office-Anwendungen wie Outlook,Word oder Excel genannt. Sie deklarieren dabei eine entsprechende Objektvariable wie etwa **objWord** mit dem Typ **Word.Application** und instanzieren Word dann mit der **New**-Methode. Wenn Sie die Deklaration mit dem Schlüsselwort **WithEvents** durchführen, können Sie anschließend nicht nur Methoden oder Eigenschaften von Word aufrufen, sondern auch noch die Ereignisse von Word durch entsprechende Ereignisprozeduren implementieren – beispielsweise, wenn ein Dokument geöffnet oder geschlossen wird:

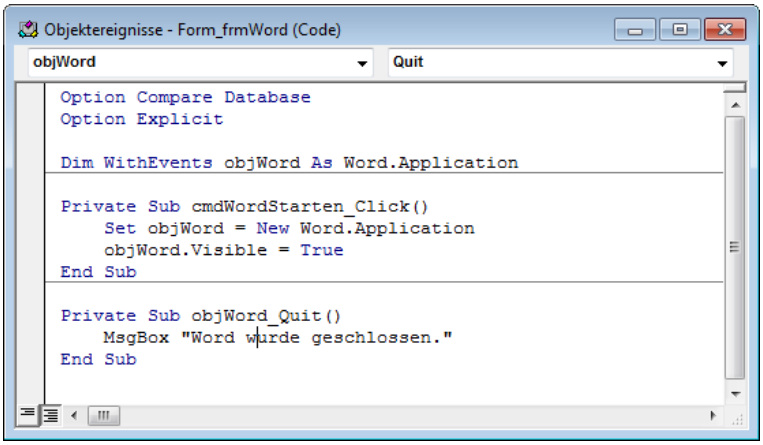


Bild 1: Implementieren eines Ereignisses eines Objekts

```
Dim WithEvents objWord As Word.Application
Set objWord = New Word.Application
```

Dazu wählen Sie dann im linken Kombinationsfeld des VBA-Fensters, in der die Variable **objWord** definiert ist, den Eintrag **objWord** aus und können dann mit dem rechten Kombinationsfeld die zur Verfügung stehenden Ereignisse selektieren – beispielsweise das **Quit**-Ereignis. Im Beispiel aus Bild 1 haben wir etwa eine Word-Instanz erstellt und unter **objWord** gespeichert. Die für das **Quit**-Ereignis erstellte Ereignisprozedur haben wir mit einer Anweisung gefüllt, die ein Meldungsfenster liefert, wenn die Word-Instanz wieder geschlossen wird.

Ereignis unter C# implementieren

Unter C# läuft dies etwas anders, aber nicht viel. Im folgenden Beispiel wollen wir die **Timer**-Klasse verwenden (der Zugriff auf Word ist von C#-Projekten aus etwas aufwändiger – dies schauen wir uns in einem späteren Beitrag an ...). Die **Timer**-Klasse erlaubt es, ein Zeitgeber-Intervall festzulegen, nach dem das Ereignis **Elapsed** ausgelöst wird.

Die Objektvariable für das Objekt, das ein Ereignis auslösen soll, deklarieren wir unter C# nicht mit einem speziellen Schlüsselwort wie **WithEvents** unter VBA, sondern einfach als private, statische Variable:

```
private static Timer timer;
```

In der gleichen Klasse legen wir unsere Methode an, welche das Objekt auf Basis der Klasse **Timer** instanziert, stellen das Zeitgeber-Intervall ein und starten den Zeitgeber. Außerdem,

und das ist der wichtigste Punkt, legen wir fest, dass beim Eintreten des Ereignisses **Elapsed** der **Timer**-Klasse die Methode **Timer_Elapsed** ausgeführt werden soll. Dies erreichen wir durch die folgende Zeile:

```
timer.Elapsed += Timer_Elapsed;
```

Die komplette Prozedur zum Erstellen des **Timer**-Objekts und zum Starten des Zeitgebers sieht nun wie folgt aus:

```
public static void TimerEreignis() {
    Console.WriteLine("Start Timer. Intervall: eine Sekunde.");
    timer = new Timer();
    timer.Elapsed += Timer_Elapsed;
    timer.Interval = 1000;
    timer.Start();
}
```

Nun benötigen wir noch die Methode, die beim Eintreten des **Elapsed**-Ereignisses ausgelöst wird – unter VBA haben wir diese Ereignisprozedur genannt. Diese erhält die ähnliche Signatur wie auch die Ereignisprozeduren etwa von Steuerelementen, also mit zwei Parametern namens **sender** und **e**:

```
private static void Timer_Elapsed(object sender,
        ElapsedEventArgs e) {
    timer.Stop();
    Console.WriteLine("Zeit abgelaufen.");
    Console.ReadLine();
}
```

e hat hier allerdings den Typ **ElapsedEventArgs**. Dieses Objekt liefert neben den Standardelementen mit der Eigenschaft **SignalTime** die Zeit, zu der das Ereignis ausgelöst wurde. Dieses nehmen wir noch zur Ausgabe hinzu:

```
Console.WriteLine("Zeit abgelaufen. {0}", e.SignalTime);
```

Eigene Ereignisse programmieren

Nun wollen wir eine Klasse mit einem eigenen Ereignis programmieren. Ausgangspunkt ist die folgende einfache Klasse namens **Person** mit einer einzigen Eigenschaft namens **Vorname** sowie den entsprechenden **get**- und **set**-Methoden:

```
public class Person {
    private string vorname;
    public string Vorname {
        get {
            return this.vorname;
        }
        set {
            this.vorname = value;
        }
    }
}
```

Die folgende Methode erstellt ein neues Objekt auf Basis dieser Klasse und füllt die Eigenschaft **Vorname** mit dem Wert **André**:

```
public static void KundeErstellen() {
    Person person = new Person();
    person.Vorname = "André";
    Console.WriteLine("Vorname: {0}", person.Vorname);
    Console.ReadLine();
}
```

Diese Klasse soll nun beim Ändern des Wertes der Eigenschaft Vorname ein Ereignis auslösen, das wir wiederum in der Klasse mit der Methode implementieren wollen. Dazu sind auf Seite der Klasse **Person** die folgenden Schritte nötig:

- Definieren eines Delegates, der festlegt, welche Parameter die zu implementierenden Ereigisprozeduren verwenden müssen (und die auch beim Auslösen gefüllt werden),
- Definieren eines Ereignisses mit dem Schlüsselwort event, dem Typ des Ereignisses (der dem zuvor definierten Delegate entspricht) und dem Namen des Ereignisses und
- die Definition einer Methode, die der Signatur des Delegates entspricht und das Ereignis auslöst, sofern dieses

abonniert wurde und die innerhalb der Klasse beim Eintreten des jeweiligen Ereignisses – hier beim Ändern des Wertes der Eigenschaft **Vorname** in der entsprechenden **Set**-Methode – aufgerufen wird.

```
public class Person {
    private string vorname;
    public string Vorname {
        get {
            return this.vorname;
        }
        set {
            this.vorname = value;
            BeiEigenschaftGeaendert(this, null);
        }
    }

    public delegate void EigenschaftGeaendertEventHandler(object sender, EventArgs e);

    public event EigenschaftGeaendertEventHandler EigenschaftGeaendert;

    protected void BeiEigenschaftGeaendert(object sender, EventArgs e) {
        if(EigenschaftGeaendert != null) {
            EigenschaftGeaendert(this, e);
        }
    }
}
```

Listing 1: Die Klasse **Person** mit einem Ereignis, das beim Ändern einer Eigenschaft feuert

Die Klasse **Person** wird nun also wie in Listing 1 erweitert. In der **get**/**set**-Methode kommt nur die Zeile hinzu, welche die Methode **BeiEigenschaftGeaendert** aufruft. Diese übergibt als ersten Parameter einen Verweis auf die Klasse selbst, als zweiten den Wert **null**. Später werden wir dies noch erweitern.

Das **delegate**-Objekt namens **EigenschaftGeaendertEventHandler** enthält die Signatur für die zu implementierende Ereignisprozedur. Die Namen aller Delegates, die als Typ eines **event**-Objekts herangezogen werden, sollen auf **EventHandler** enden.

Die Ereignis-Definition, also die Zeile mit dem Schlüsselwort **event**, enthält wie eine herkömmliche Variablendeklaration den Typ der Variablen (hier also **EigenschaftGeaendertEventHandler**) sowie den Namen des Events, hier **EigenschaftGeaendert**. Die Verwendung des **event**-Schlüsselworts bringt unter anderem den Vorteil, dass Sie in der definierenden Methode die Operatoren **+=** und **-=** verwenden können, um das Ereignis zu abonnieren oder zu entfernen.

Fehlt noch der Aufruf des Events. Diesen packen wir in eine eigene Methode namens **BeiEigenschaftGeaendert**, welche die Standardparameter für Ereignisse verwendet, also solche vom Typ **object** und **EventArgs**. Die Methode vergleicht den Wert des Events **EigenschaftGeaendert** mit dem Wert **null**. Dies dient der Prüfung, ob das Event überhaupt von der implementierenden Klasse abonniert wurde. »Abonniert« entspräche unter VBA dem Zuweisen des Wertes **[Event Procedure]** zu der jeweiligen Ereigniseigenschaft, also etwa so:

```
txt.AfterUpdate = "[Event Procedure]"
```

Unter C# gelingt dies, sofern Sie die Klasse **Person** wie oben beschrieben erweitert haben, durch einen kleinen Schritt: Dann stellen Objekte auf Basis der Klasse **Kunde** nämlich den Namen des **event**-Objekts, hier **EigenschaftGeaendert**, als Eigenschaft bereit. Diesem fügen Sie dann den Namen der Methode hinzu, mit der Sie das Ereignis implementieren wollen. Visual Studio unterstützt Sie dabei, indem es vorschlägt, die **Tabulator**-Taste zu drücken, um die Event-Eigenschaft zu füllen und das Grundgerüst des Eventhandlers anzulegen (siehe Bild 2).

Von VBA zu C#: Dateidialoge

Unter VBA musste man schon einigen Zusatzcode inklusive Api-Deklarationen zu seinem Projekt hinzufügen, um Dialoge etwa zum Auswählen einer zu öffnenden Datei, eines Verzeichnisses oder zur Angabe eines Dateinamens zum Speichern anzuzeigen. Unter .NET gibt es dazu natürlich eine vorgefertigte Klasse, die alle notwendigen Funktionen liefert. Der vorliegende Artikel zeigt, wie Sie diese Dialoge anzeigen und die damit ermittelten Daten nutzen können.

Die folgenden Beispiele verwenden eine WPF-Anwendung unter C#.

Dateien auswählen per Klasse

Unter VBA gibt es verschiedene Varianten, um beispielsweise einen **Datei öffnen**-Dialog anzuzeigen. Per API, über die versteckte Wizhook-Klasse oder auch mit der entsprechenden Klasse der Office-Bibliothek. Alle Lösungen hatten Vor- und Nachteile, aber wirklich perfekt war keine davon.

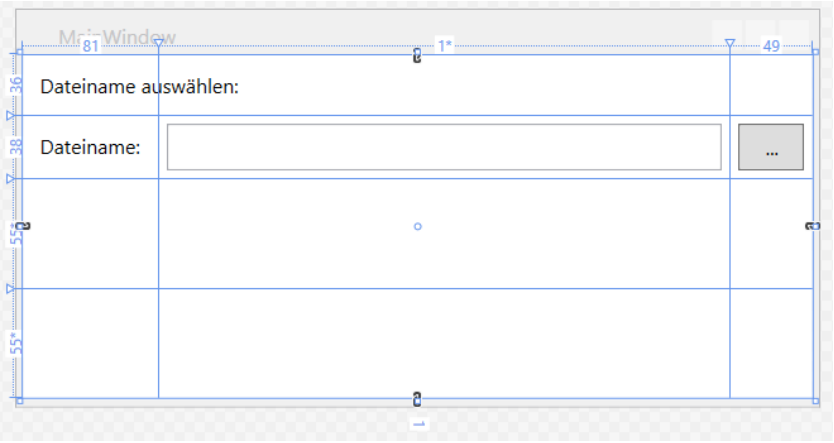


Bild 1: Entwurf des Fensters zum Auswählen einer Datei

Unter .NET und in unserem Fall unter C# gibt es eine Klasse namens **OpenFileDialog**. Diese ist Bestandteil des Namespaces **Microsoft.Win32**, den Sie der Klasse per **using**-Anweisung bekannt machen:

```
using Microsoft.Win32;
```

Danach können Sie bereits auf die **OpenFileDialog**-Klasse zugreifen. Dazu legen wir ein WPF-Formular an, das wie in Bild 1 aussieht. Das Textfeld heißt **txtDateiname**, die Schaltfläche **btnDateiAuswaehlen**. Für die Schaltfläche hinterlegen wir die folgende Ereignisprozedur:

```
private void btnDateiAuswaehlen_Click(object sender,
        RoutedEventArgs e) {
    OpenFileDialog openFileDialog = new OpenFileDialog();
    if (openFileDialog.ShowDialog()== true) {
        txtDateiname.Text = openFileDialog.FileName;
    }
}
```

Diese erstellt zunächst ein neues Objekt auf Basis der Klasse **OpenFileDialog**. Im folgenden Schritt rufen wir den **Datei öffnen**-Dialog mit der Funktion **ShowDialog** auf. Diese zeigt den **Datei öffnen**-Dialog an und erwartet die Eingabe des Benutzers (siehe Bild 2). Die Funktion liefert als Ergebnis einen Boolean-Wert zurück, den die **if**-Anweisung auswertet. Lautet das Ergebnis **true**, hat der Benutzer eine Datei ausgewählt und wir können diese über die Eigenschaft **FileName** des **OpenFileDialog**-Objekts auslesen und als Text des Textfeldes **txtDateiname** eintragen.

Den Wert **false** liefert die Funktion nur dann zurück, wenn der Benutzer den Dialog mit der **Schließen**-Schaltfläche oben rechts oder mit der **Abbrechen**-Schaltfläche unten rechts beendet.

Eigenschaften des OpenFileDialog-Objekts

Die Klasse besitzt natürlich einige Eigenschaften, mit denen Sie das Aussehen des Dialogs und auch seine Funktion

beeinflussen können. Hier ist eine Liste der wichtigsten Eigenschaften samt Kurzbeschreibung – weiter unten schauen wir uns Beispiele zu den wichtigsten Eigenschaften an:

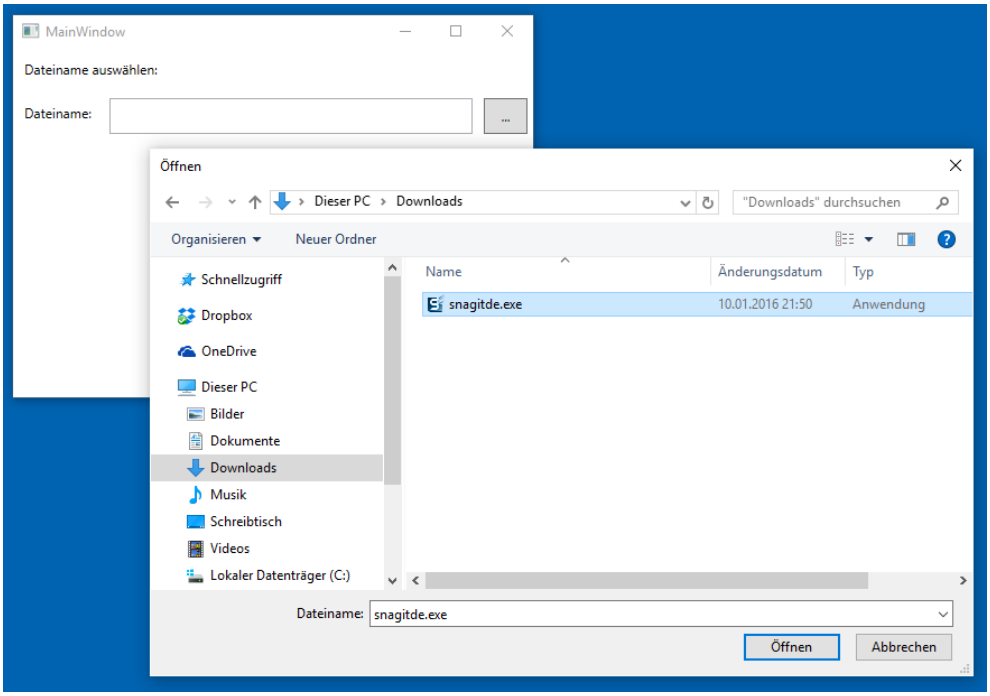


Bild 2: Ein einfacher **Datei öffnen**-Dialog

- **CheckFileExists**: Boolean-Wert, der angibt, ob das Vorhandensein der gewählten Datei geprüft werden soll. Der Wert **false** lässt auch die Angabe nicht vorhandener Dateien zu, der Wert **true** liefert eine Meldung, wenn eine nicht vorhandene Datei gewählt wird (Standardwert: **true**)

- **CustomPlaces**: Spezielle Ordner zusätzlich zu den Favoriten-Ordnern anzeigen

- **DereferenceLinks**: Legt fest, ob bei der Auswahl einer Dateiverknüpfung die Verknüpfung selbst zurückgegeben werden soll (in der Regel eine **.lnk**-Datei) oder der Pfad zu der verknüpften Datei.

- **FileName**: Liefert den Dateinamen der gewählten Datei zurück.

- **FileNames**: Liefert ein Array mit den Dateinamen der gewählten Dateien zurück. Sowohl **FileName** als auch **FileNames** funktionieren bei der Einfachauswahl als auch bei der Mehrfachauswahl (siehe Eigenschaft **MultiSelect**).

- **FileOk**: Dies ist ein Ereignis, das ausgelöst wird, wenn der Benutzer die Schaltfläche **Öffnen** des **Datei öffnen**-Dialogs betätigt hat.

- **Filter**: Erwartet einen Ausdruck, der die zu filternden Dateien angibt. Beispiel: **Access-Datenbanken (*.mdb, *.accdb)|*.mdb;*.accdb|Alle Dateien (*.*)|*.***

- **FilterIndex**: Legt fest, welcher **Filter**-Eintrag angezeigt werden soll (**1** für den ersten Eintrag, **2** für den zweiten und so weiter)

- **InitialDirectory**: Gibt das beim Einblenden des Dialogs anzuzeigende Verzeichnis an, entweder als Zeichenkette mit führendem @-Zeichen (**@"c:\"**) oder als Ausdruck, zum Beispiel **AppDomain.CurrentDomain.BaseDirectory** für das Verzeichnis der aktuellen Anwendung.

- **MultiSelect**: **Boolean**-Eigenschaft, die festlegt, ob nur eine (**false**) oder mehrere Dateien gleichzeitig ausgewählt werden können (**true**).

- **SafeFileName**: Liefert den Dateinamen ohne Verzeichnis.

- **SafeFileNames**: Liefert bei Mehrfachauswahl die Dateinahmen ohne Verzeichnis.

- **ShowDialog**: Zeigt den Dialog an und liefert den Wert **true** zurück, wenn der Benutzer den Dialog mit der **Öffnen**-Schaltfläche schließt.

- **Title**: Legt den Fenstertitel fest.

Mehrere Dateien ermitteln

Wenn Sie mit der **OpenFileDialog**-Klasse nicht nur eine, sondern gegebenenfalls auch einmal mehrere Dateien gleichzeitig ermitteln möchten, bedarf es nur weniger Änderungen (siehe Listing 1). Dazu stellen Sie zunächst einmal den Wert der Eigenschaft **MultiSelect** der Objektvariablen **openFileDialog** auf den Wert **true** ein. Danach zeigen Sie den Dialog wie gewohnt mit der Methode **ShowDialog** an.

```
private void btnDateienAuswaehlen_Click(object sender, RoutedEventArgs e) {
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.InitialDirectory = AppDomain.CurrentDomain.BaseDirectory;
    openFileDialog.Multiselect = true;
    if (openFileDialog.ShowDialog() == true) {
        foreach(string filename in openFileDialog.FileNames) {
            lstDateinamen.Items.Add(filename);
        }
        txtDateiname.Text = openFileDialog.FileName;
    }
}
```

Listing 1: Einlesen mehrerer Dateinamen per **OpenFileDialog**

Hat der Benutzer die gewünschten Dateien ausgewählt und die **Öffnen**-Schaltfläche betätigt (siehe Bild 3), gehen wir allerdings etwas anders vor als bei einer einzigen Datei. In diesem Fall soll die Methode nämlich eine **foreach**-Schleife über alle **string**-Objekte der Auflistung **FileNames** des **OpenFileDialog**-Objekts durchlaufen. Innerhalb der Schleife fügt die Methode der Auflistung **Items** des Listenfeldes namens **lstDateinamen** einen neuen Eintrag mit der **Add**-Methode hinzu.

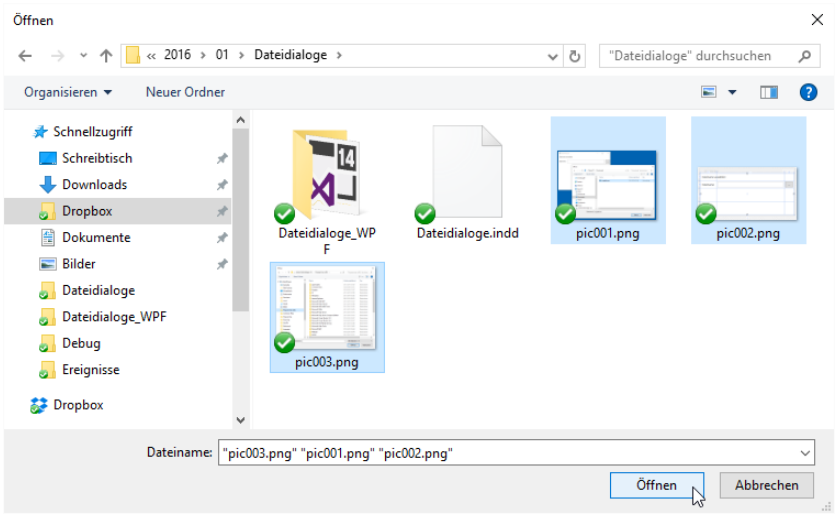


Bild 3: Markieren mehrerer Dateien gleichzeitig

Das Ergebnis sieht wie in Bild 4 aus. Der Methode haben wir noch eine weitere Anweisung hinzugefügt, welche den Wert der Eigenschaft **FileName** in das Textfeld **txtDateiname** einträgt. Damit zeigen wir, dass auch bei der Einstellung **MultiSelect = true** der erste Eintrag über die **FileName**-Eigenschaft ermittelt werden kann.

Textdatei einlesen

Wenn wir schon Dateien auswählen, können wir deren Inhalt auch direkt einmal in ein Textfeld einlesen – zumindest, wenn es sich um eine Textdatei handelt.

Dazu verwenden wir die Schaltfläche **btnDateiEinlesen**, welche die Methode aus Listing 2 auslöst. Wir

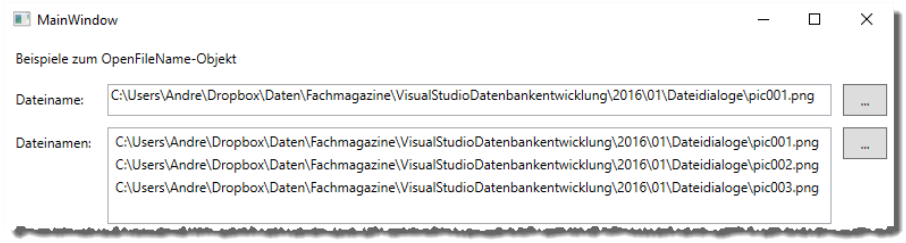


Bild 4: Anzeige der Dateien im Listenfeld

```
private void btnDateiEinlesen_Click(object sender, RoutedEventArgs e) {
    OpenFileDialog openFileDialog = new OpenFileDialog();
    StreamReader streamReader;
    openFileDialog.Filter = "Text-Dateien (*.txt)|*.txt|XML-Dateien (*.xml)|*.xml|Alle Dateien (*.*)|*.*";
    openFileDialog.FilterIndex = 1;
    if (openFileDialog.ShowDialog() == true) {
        string dateiname = openFileDialog.FileName;
        streamReader = new StreamReader(dateiname);
        this.txtDateiinhalt.Text = streamReader.ReadToEnd();
        this.txtDateiname.Text = dateiname;
    }
}
```

Listing 2: Einlesen einer Textdatei nach der Auswahl mit dem **OpenFileDialog**

wollen an dieser Stelle gleich ein Beispiel für einen Filter liefern. Der **Öffnen**-Dialog soll entweder nur Dateien mit der Dateiendung **.txt**, mit der Endung **.xml** oder alle Dateien anzeigen (***.***) – siehe Bild 5. Dazu geben wir der Eigenschaft **Filter** einen entsprechenden Ausdruck mit und stellen den zu Beginn anzuzeigenden Filtereintrag mit der Eigenschaft **FilterIndex** ein.

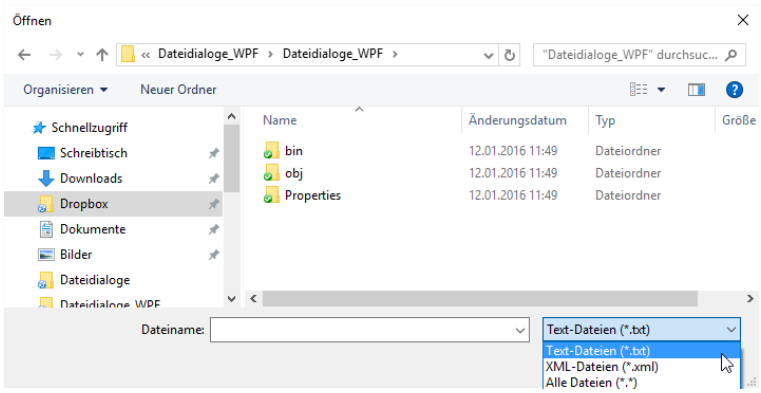


Bild 5: Einlesen von Text- oder XML-Dateien per Filter

Nach dem Anzeigen des **Öffnen**-Dialogs und der Auswahl der Datei speichert die Methode den Dateinamen in der Variablen **dateiname**. Dann erstellt sie ein neues Objekt des Typs **StreamReader** und übergibt dem Konstruktor den Dateinamen. Die Methode **ReadToEnd** ermittelt schließlich den Inhalt der Textdatei und trägt diesen in das Textfeld **txtDateiinhalt** ein. Der Dateiname landet schließlich noch im Textfeld **txtDateiname**. Das Ergebnis sieht dann wie in Bild 6 aus.

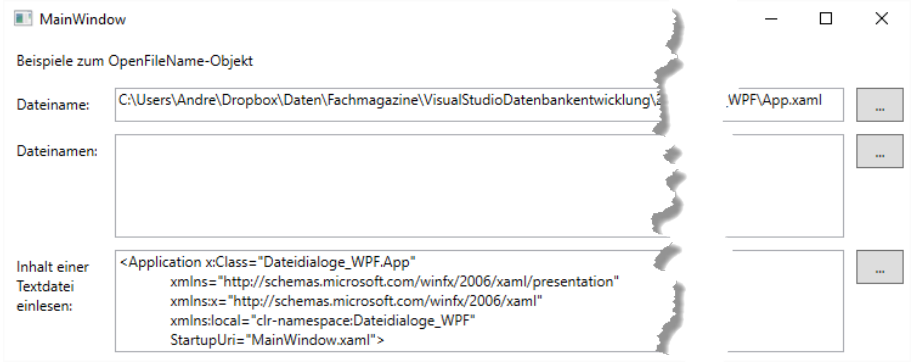


Bild 6: Das Beispielformular nach dem Einlesen einer XML-Datei

Ereignis nach Auswahl auslösen

Die Klasse **OpenFileDialog** bietet auch ein Ereignis an, das Sie implementieren können. Dazu legen wir eine neue Schaltfläche namens **btnEreignis-FileOk** an. Die dadurch ausgelöste Methode finden Sie in Listing 3. Diese zeigt wieder ein **OpenFileDialog**-Fenster an. Diesmal fügt sie jedoch der Eigenschaft **FileOk** den Namen der auszulösenden Methode hinzu, nämlich **OpenFileDialog1_FileOk**. Dazu verwenden wir den **+=**-Operator. Nach der Eingabe dieses Operators bietet Visual Studio an, per **Tabulator**-Taste automatisch den Wert **OpenFileDialog1_FileOk**

Delegates

Delegates sind Typen, denen Sie eine Methode zuweisen, die sie dann über das Delegate ausführen können. Ein Delegate legt dabei nur fest, welchen Rückgabewert und welche Parameter die zu verwendende Methode nutzt – Sie können dem Delegate zur Laufzeit dann beliebige Methoden zuweisen, die dieser Signatur entsprechen. Delegates benötigen Sie aber auch, wenn Sie Klassen mit benutzerdefinierten Ereignissen programmieren möchten. In diesem Fall nutzen Sie ein Delegate, um die beim Eintreten des Ereignisses auszuführende Methode festzulegen.

In diesem Artikel schauen wir uns zunächst an, wie ein Delegate funktioniert, und erläutern erst im Anschluss, welchen Nutzen dies hat. Angenommen, Sie haben zwei Methoden, welche die gleichen Parameter nutzen, aber verschiedene Dinge erledigen. Ein schönes Beispiel ist dabei das Addieren und das Multiplizieren zweier Zahlen. Die beiden Methoden würden dann etwa wie folgt aussehen:

```
public static int Summe(int a, int b) {
    return a + b;
}
public static int Produkt(int a, int b) {
    return a * b;
}
```

Wollen Sie nun eine dieser Methoden aufrufen, würden Sie dies beispielsweise wie folgt erledigen:

```
Console.WriteLine("Summe aus 2 und 3: {0}", Summe(2, 3));
```

Zum Multiplizieren rufen Sie einfach die Methode **Produkt** auf:

```
Console.WriteLine("Produkt aus 2 und 3: {0}", Produkt(2, 3));
```

Wenn Sie beim Erstellen des Projekts noch nicht wissen, welche der Methoden an einer bestimmten Stelle zum Einsatz kommen wird, fragen Sie beispielsweise den Benutzer danach und codieren die Aufrufe dann direkt in die Bedingung:

```
public static void Test_SummeOderProdukt() {
    Console.WriteLine("Möchten Sie multiplizieren (*)
        oder addieren (+)?");
    string operation = Console.ReadLine();
    if (operation == "*") {
        Console.WriteLine("Produkt aus 2 und 3: {0}",
            Produkt(2, 3));
    }
    else {
        Console.WriteLine("Summe aus 2 und 3: {0}",
            Summe(2, 3));
    }
    Console.ReadLine();
}
```

Aufruf per Delegate

Diese beiden Methoden wollen wir nun über eine einzige Methode aufrufen, wobei wir dieser Methode zuvor den Namen der zu verwendenden Methode (also **Summe** oder **Produkt**) übergeben, damit diese weiß, welche Methode sie nutzen soll. Diese gemeinsame Methode heißt dann allerdings nicht mehr Methode, sondern Delegate, und es ist auch keine Methode mehr, sondern ein Objekt. Dieses deklarieren wir wie folgt:

```
public delegate int Berechnung(int a, int b);
```

Die Unterschiede zu einer herkömmlichen Methode sind offensichtlich: Erstens verwenden wir bei der Deklaration das Schlüsselwort **delegate**. Zweitens hat dieses Delegate im Gegensatz zu einer Methode nur eine Deklaration, aber keinen in geschweiften Klammern befindlichen Programmcode. Nun wollen wir über das Delegate **Berechnung** zunächst

Anwendungskonfigurationsdateien

Während Sie etwa in einer Access-Anwendung leicht die Konfigurationsdaten in einer eigens dafür vorgesehenen Tabelle speichern können oder in einer benutzerdefinierten Text- oder XML-Datei, gibt es unter C# andere Möglichkeiten. Wie für alles finden Sie unter C# auch für das Verwalten von Konfigurationsdaten einen eigenen Namespace – in diesem Fall heißt dieser ConfigurationManager. Dieser Artikel zeigt, wie Sie damit Daten für Desktop-Anwendungen speichern und wieder abrufen können.

Beispielanwendung

Als Beispielanwendung legen wir ein C#-Projekt des Typs Konsolenanwendung an und speichern es unter dem Namen **Konfigurationsdateien**.

Ausprobieren

Wenn Sie die Methoden dieses Artikels selbst in ein neues Projekt eingeben und ausprobieren möchten, fügen Sie diese zur Klasse **Program.cs** hinzu und tragen den Aufruf der Methoden in die Methode **static void Main** ein – etwa wie in folgendem Beispiel:

```
static void Main(string[] args) {
    KonfigurationsdatenLesen
}
```

Alternativ nutzen Sie die im Artikel **Methodenstarter als Vorlage** vorgestellte Variante zum Aufrufen von Beispielmethoden.

Wozu Konfigurationsdateien?

Wozu nutzt man überhaupt Konfigurationsdateien? Darin speichert man Daten, die nicht unbedingt zu den Geschäftsdaten der Anwendung gehören (die in Tabellen landen), sondern die Informationen zur Anwendung wie etwa die Verbindungszeichenfolge enthalten. Diese könnten wir zwar für Anwendungen, die nur auf dem eigenen Rechner laufen und jederzeit neu kompiliert werden können, auch etwa in einer Konstanten speichern. Sobald Sie eine Anwendung jedoch weitergeben möchten, sollten Sie Mechanismen vorsehen, mit denen Konfigurationsdaten an Orten gespeichert werden, die auch der Benutzer zur Laufzeit ändern kann.

Ein gutes Beispiel ist die Verbindungszeichenfolge einer Datenbankanwendung. Ob es sich dabei um eine Verbindung zu einem SQL Server handelt oder auch nur die Angabe des Speicherpfades einer als Backend verwendeten Access-Datenbank, spielt dabei keine Rolle. Tatsache ist, dass es während der Laufzeit geschehen kann, dass der Benutzer diese Daten ändern können sollte, um beispielsweise den Servernamen einer SQL Server-Verbindung zu ändern oder auch einen neuen Speicherpfad einer Access-Datenbank einzugeben.

Eine solche Konfigurationsdatei mit einer Verbindungszeichenfolge für eine Access-Datenbank sieht beispielsweise wie folgt aus:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <connectionStrings>
    <add name="Accessdatenbank"
         connectionString="c:\Suedsturm.mdb" />
  </connectionStrings>
</configuration>
```

Speicherort der Konfigurationsdateien

Für eine C#-Anwendung legt Visual Studio drei Konfigurationsdateien an, die Sie im Projektverzeichnis finden. Direkt im Hauptverzeichnis landet die Datei **App.config**. Im Verzeichnis **bin\Debug** finden Sie beim Debuggen die beiden Dateien **<Anwendungsname>.exe.config** und **<Anwendungsname>.vshost.exe.config**. Später, wenn Sie die Anwendung erstellen, ist zunächst keine **.config**-Datei vorhanden. Diese wird erstellt, sobald die Anwendung Konfigurationsinforma-

tionen speichert – und zwar ebenfalls unter dem Namen **<Anwendungsname>.exe.config**.

Wenn Sie eine Anwendungskonfigurationsdatei zu einem Projekt hinzufügen möchten, das nach der Installation bereitsteht, legen Sie die gewünschte Datei im XML-Format an und speichern diese im Anwendungsverzeichnis.

Konfigurationsdatei manuell hinzufügen

Wenn Sie eine Konfigurationsdatei hinzufügen möchten, statt diese per Code zu erzeugen, erledigen Sie dies am einfachsten über den Menüeintrag **Projekt|Neues Element hinzufügen**. Dies öffnet den Dialog **Neues Element hinzufügen**, wo Sie den Eintrag **Anwendungskonfigurationsdatei** auswählen (siehe Bild 1). Geben Sie den gewünschten Namen für die Anwendungskonfigurationsdatei ein und klicken Sie auf die Schaltfläche **Hinzufügen**.

Die neue Anwendungskonfigurationsdatei sieht nun wie in Bild 2 aus und wird auch im Projektmappen-Explorer angezeigt. Prinzipiell benötigen Sie aber keine

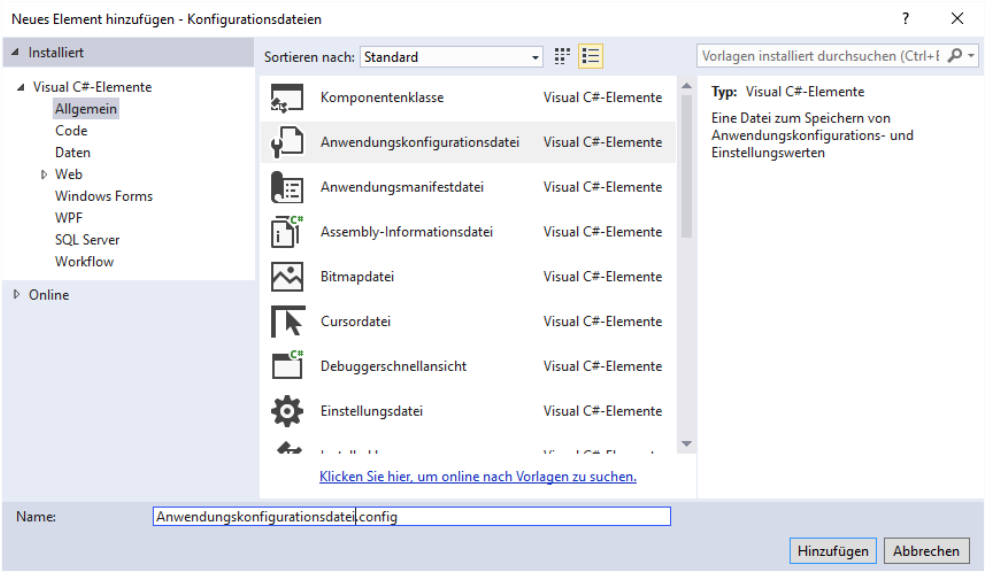


Bild 1: Anlegen einer Anwendungskonfigurationsdatei

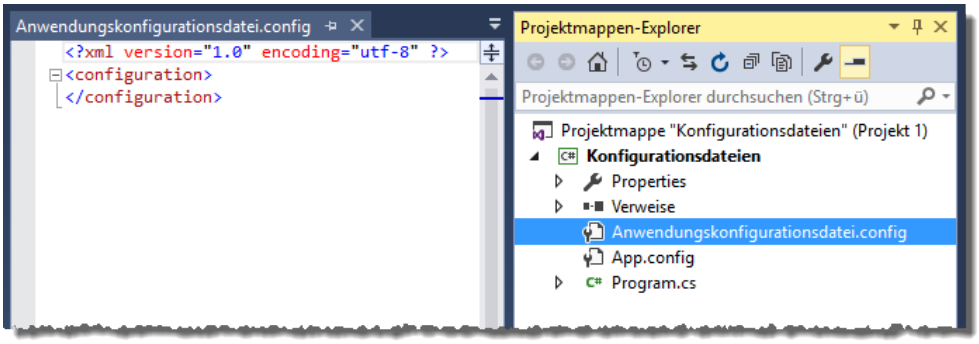


Bild 2: Die neue Anwendungskonfigurationsdatei

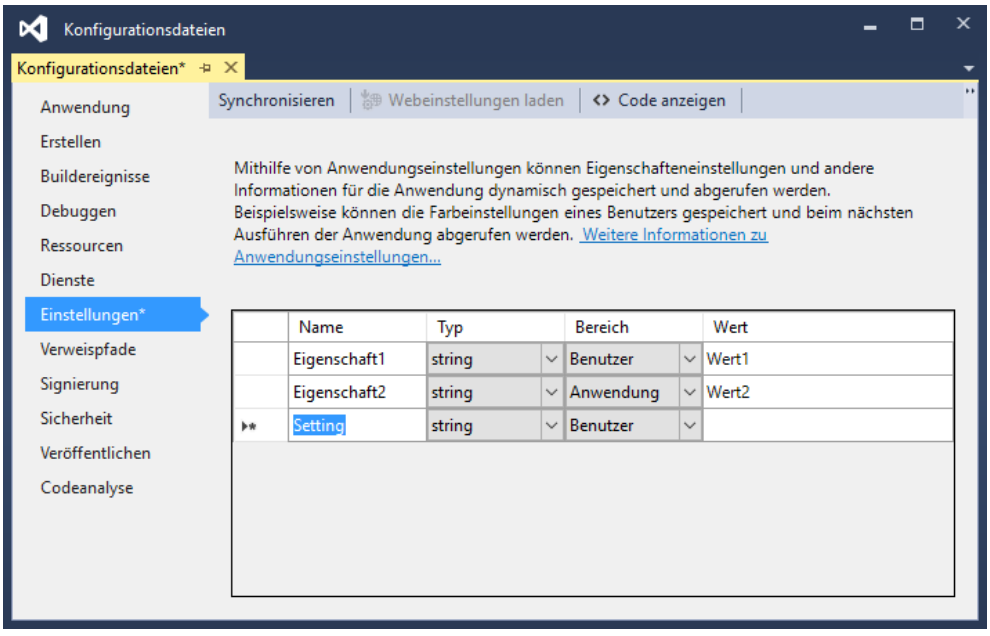


Bild 3: Einstellungen zu einer Anwendung hinzufügen

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="userSettings" ... >
            <section name="Konfigurationsdateien.Properties.Settings" ... />
        </sectionGroup>
        <sectionGroup name="applicationSettings" ... >
            <section name="Konfigurationsdateien.Properties.Settings" ... />
        </sectionGroup>
    </configSections>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
    </startup>
    <userSettings>
        <Konfigurationsdateien.Properties.Settings>
            <setting name="Eigenschaft1" serializeAs="String">
                <value>Wert1</value>
            </setting>
        </Konfigurationsdateien.Properties.Settings>
    </userSettings>
    <applicationSettings>
        <Konfigurationsdateien.Properties.Settings>
            <setting name="Eigenschaft2" serializeAs="String">
                <value>Wert2</value>
            </setting>
        </Konfigurationsdateien.Properties.Settings>
    </applicationSettings>
</configuration>
```

Listing 1: Konfigurationsdatei **app.config** mit einer Benutzer- und einer Anwendungseinstellung

neue Konfigurationsdatei, denn das Projekt bringt mit **app.config** bereits eine solche mit. Sollten Sie diese Datei jedoch einmal versehentlich löschen, können Sie diese mit der oben beschriebenen Vorgehensweise wieder herstellen.

Konfigurationsdaten anlegen

Um Konfigurationsdateien mit Eigenschaften auszustatten, verwenden Sie den Einstellungen-Editor von Visual Studio. Um diesen zu öffnen, klicken Sie doppelt auf den Eintrag **Properties** im Projektmappen-Explorer. Wechseln Sie dann zum Bereich **Einstellungen** und fügen Sie die benötigten Eigenschaften samt Datentyp, Bereich (**Benutzer** oder **Anwendung**) sowie den Standardwert fest (siehe Bild 3). Warum Standardwert? Weil die Konfigurationsdatei erst beim Erstellen der Anwendung erzeugt und mit Werten gefüllt wird.

Um dies zu prüfen, erstellen Sie die Anwendung zum Debuggen (**F5**) und schauen sich anschließend den Inhalt der Datei **app.config** im Hauptverzeichnis des Projekts an. Diese sieht nun in gekürzter Form wie in Listing 1 aus. Im Bereich **userSettings** finden Sie die Eigenschaft mit der Bezeichnung **Eigenschaft1** und dem Wert **Wert1**, unter **applicationSettings** die Eigenschaft namens **Eigenschaft2** mit dem Wert **Wert2**.

Einlesen der Konfigurationsdateien

Wenn Sie die so angelegten Einstellungen von einer C#-Anwendung aus einlesen möchten, erledigen Sie das etwa mit einer Methode wie der folgenden:

```
public static void KonfigurationsdatenLesen() {
    Settings settings = new Settings();
```

```
    Console.WriteLine("Wert von Eigenschaft1: {0}",
        settings.Eigenschaft1);
    Console.WriteLine("Wert von Eigenschaft2: {0}",
        settings.Eigenschaft2);
    Console.ReadLine();
}
```

Bei der Eingabe dieser Methode wird Visual Studio meckern, weil es das Objekt **Settings** nicht kennt. **Settings** ist eine Klasse, die automatisch angelegt wird, sobald Sie im Bereich **Properties** der Anwendungseinstellungen eine Eigenschaft angelegt haben. Diese Klasse sehen Sie auch im Projektmappen-Explorer (siehe Bild 4). Damit Sie im Code komfortabel auf die Elemente der **Settings**-Klasse zugreifen können, fügen Sie dem Klassenmodul mit der **using**-Direktive einen Verweis auf das Element **<Anwendungsname>.Properties** hinzu:

```
using Konfigurationsdateien.Properties;
```

Danach ist der Zugriff auf die Klasse **Settings** per Code problemlos möglich, und Sie können die Werte der per Benutzeroberfläche hinzugefügten Einstellungen über **settings.Eigenschaft1** und **settings.Eigenschaft2** ausgeben. Die Klasse **Settings** wird automatisch bei Änderungen an den Eigenschaften neu erstellt und arbeitet wie eine benutzerdefinierte Klasse – sie stellt die Eigenschaften per **Set**- und **Get**-Methode zum Lesen und Schreiben zur Verfügung. Ein Blick in den Code dieser Klasse zeigt jedoch direkt, dass die Einstellungen des Bereichs **Benutzer** gelesen und geändert, die Einstellungen des Bereichs **Anwendung** jedoch nur gelesen werden können.

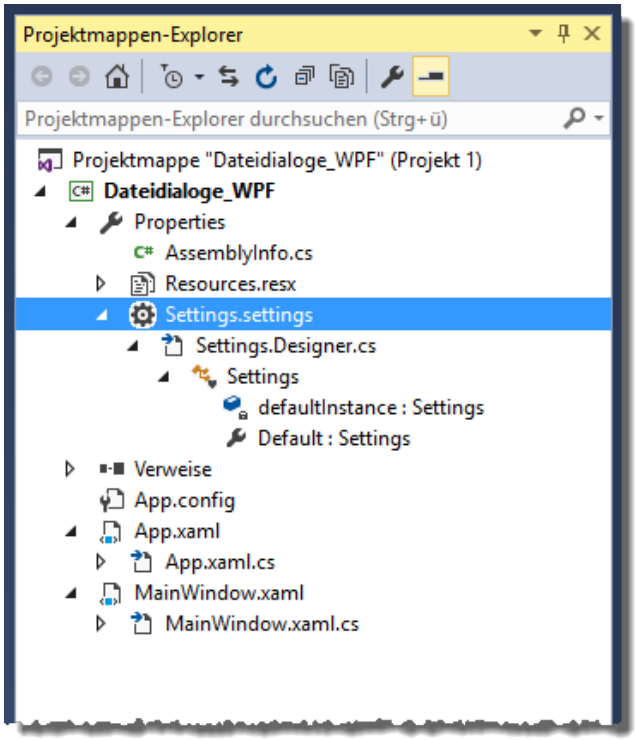


Bild 4: Die Klasse **Settings** stellt die benutzerdefinierten Einstellungen bereit.

Benutzereinstellungen ändern

Wenn Sie also per Code Einstellungen ändern möchten, gelingt dies ohne Weiteres nur bei den Benutzereinstellungen – und zwar beispielsweise wie folgt:

```
public static void KonfigurationsdatenAendern() {
    Settings settings = new Settings();
    settings.Eigenschaft1 = "Test1_Neu";
    settings.Save();
    Console.ReadLine(); }
```

Dies ändert den Wert von **Eigenschaft1** auf **Test1_Neu**. Wenn Sie diese Methode aufrufen und anschließend nochmals die Methode **KonfigurationsdatenLesen**, wird der neue Wert ausgegeben.

Speicherort nach der Weitergabe

Wo sich die Konfigurationsdatei beim Debuggen befindet, haben wir bereits weiter oben erläutert. Aber wo finden Sie die entsprechenden Dateien, wenn Sie die **.exe**-Datei weitergeben und auf einem anderen Rechner ausführen? Genau genommen handelt es sich in diesem Fall um zwei Konfigurationsdateien. Die erste enthält die Benutzereinstellungen, die zweite die Anwendungseinstellungen. Die Benutzereinstellungen finden Sie etwa unter Windows 7 in einem Verzeichnis, das sich wie folgt zusammensetzt:

```
C:\Users\<Benutzername>\AppData\Local\<Hersteller>\<Anwendungsname>.exe_<Evidence Type>_<Evidence Hash>\<Version>\user.config
```

Wenn Sie für **<Hersteller>** keine Angabe in der Datei **AssemblyInfo.cs** gemacht haben, verwendet die Anwendung die Bezeichnung des Namespace (hier **nsKonfigurationsdateien**). **<Evidence Type>** und **<Evidence Hash>** werden vom System generiert. Diese Datei enthält nur den Bereich **userSettings** der weiter oben vorgestellten Konfigurationsdatei:

```
<configuration>
    <userSettings>
        <Konfigurationsdateien.Properties.Settings>
            <setting name="Eigenschaft1" serializeAs="String">
                <value>Test1_Neu</value>
            </setting>
        </Konfigurationsdateien.Properties.Settings>
    </userSettings>
</configuration>
```

Wie nun können wir etwa per Code ermitteln, wo sich diese Datei befindet? Dazu fügen Sie zunächst einen Verweis auf **System.Configuration** zum Projekt hinzu (siehe Bild 5) und erstellen dann Sie ein Objekt des Typs **Configuration**:

```
Configuration config =
    ConfigurationManager.OpenExeConfiguration(
    ConfigurationUserLevel.PerUserRoamingAndLocal);
```

Dieses liefert mit der Eigenschaft **FilePath** den Pfad zur Konfigurationsdatei:

```
Console.WriteLine("Speicherort der
    Benutzerkonfigurationsdatei: {0}",
    config.FilePath);
```

Diese beiden Zeilen können Sie beispielsweise in der Methode **KonfigurationsdatenLesen** unterbringen.

Ändern per Konfigurationsdatei

Sie oder der Benutzer können die Einstellungen der Datei **user.config** natürlich auch direkt in der Datei ändern. Diese stehen anschließend direkt in der Anwendung zur Verfügung.

Anwendungseinstellungen ändern

Wie Sie weiter oben erfahren haben, können Sie Einstellungen im Bereich **applicationSettings** nicht wie die im Bereich **userSettings** per Code ändern. Wie aber können wir dennoch anwendungsweite Einstellungen pflegen, die alle Benutzer betreffen? Dazu gibt es einen weiteren Bereich namens **appSettings**. Die Inhalte dieses Bereichs können Sie nicht so einfach über die Benutzeroberfläche von Visual Studio vorbereiten wie bei den beiden Bereichen **userSettings** und **applicationSettings**, die Sie weiter oben kennen gelernt haben. In der **.config**-Datei sieht der Bereich wie folgt aus, hier mit zwei Beispieleigenschaften:

```
<configuration>
    <configSections>...</configSections>
    ...
    <appSettings>
        <add key="eigenschaft1" value="wert1" />
        <add key="eigenschaft2" value="wert2" />
    </appSettings>
    ...
</configuration>
```

Die Einträge dieses Bereichs lesen Sie beispielsweise mit der Methode aus Listing 2. Die Methode verwendet die Eigenschaft **Count** der Auflistung **AppSettings** des **ConfigurationManager**-Objekts, um die Anzahl der Einstellungen zu ermitteln und in der Variablen **count** zu speichern. Ist **count**

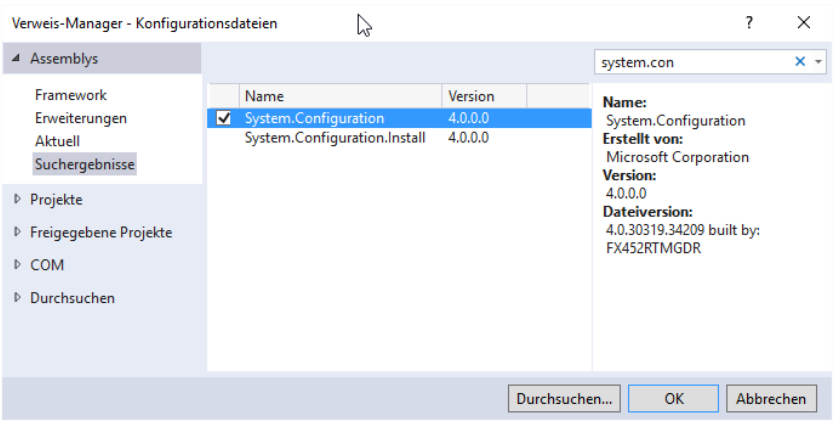


Bild 5: Die Klasse **Settings** stellt die benutzerdefinierten Einstellungen bereit.

Datenzugriff mit ADO.NET, Teil 2

Im ersten Teil unserer Artikelreihe zum Thema ADO.NET haben wir uns angesehen, wie Sie mit dem DataReader-Objekt auf die Daten einer Tabelle zugreifen und wie Sie mit dem Command-Objekt Aktionsabfragen durchführen können. Im vorliegenden Teil sehen wir uns an, wie Sie mit dem DataAdapter-Objekt auf die Daten einer Datenbank zugreifen und das DataSet- und das DataTable-Objekt einsetzen, um verbindungslos mit den Daten einer Datenbank zu arbeiten.

Die im ersten Teil dieser Artikelreihe vorgestellten Techniken für den Zugriff auf die Daten einer Datenbank erforderten eine geöffnete Verbindung. So etwas ist in einer Desktopdatenbank etwa auf Basis von Microsoft Access kein Problem. Etwas kritischer wird es schon, wenn Sie Access in einer Mehrbenutzerumgebung einsetzen – und vor allem dann, wenn die Anwendung über das Internet über die anzuzeigenden oder zu bearbeitenden Daten zugreift. In Zeiten der Mehrbenutzer-, Internet- und mobilen Anwendungen benötigen wir daher verbindungslose Zugriffstechnologien.

Die unterstützt ADO.NET durch einige weitere Objekte, welche nach Wunsch mit den Daten einer oder mehrerer Tabellen oder Abfragen der Datenbank gefüllt werden. Der Benutzer zeigt die Daten an, löscht, bearbeitet oder erweitert sie und stößt dann den Speichervorgang für die Daten an. Daraufhin wird erneut eine Verbindung geöffnet und die geänderten Daten werden zurück in die Datenbank geschrieben. Wie dies gelingt, schauen wir uns weiter unten an. Vorher jedoch noch einige Ergänzungen zum ersten Teil dieser Artikelreihe.

Verbindungszeichenfolge einfach festlegen

Wer sich mit dem Zusammenstellen der Zeichenkette für die Verbindungszeichenfolge schwer tut und/oder Eingabefehler bei der Benennung der Parameter vermeiden will, kann auch ein Objekt namens **ConnectionStringBuilder** verwenden. Dieses gibt es ebenso wie die Objekte **Connection**, **Command** et cetera jeweils mit entsprechendem Präfix für die verschiedenen ADO.NET-Klassen wie **System.Data.OLEDB**, **System.Data.ODBC**, **System.Data.SQL** und so weiter. Da wir aktuell noch eine Access-Datenbank als Datenquelle für unsere Beispiele nutzen, heißt die entsprechende Klasse also **OLEDBConnectionStringBuilder**. Die folgende Prozedur erstellt ein neues Objekt auf Basis dieser Klasse und speichert es in der Variablen **ConnectionStringBuilder**. Die Objektvariable stellt alle für die aktuelle Verbindung möglichen Parameter als Eigenschaften zur Verfügung, sodass Sie die gewünschten Elemente einfach per IntelliSense auswählen wollen.

Im vorliegenden Fall weisen wir so den Parameter **DataSource** und den Parameter **Provider** zum Objekt **ConnectionStringBuilder** hinzu und stellen so die Verbindungszeichenfolge zusammen, die wir dann über die Eigenschaft **ConnectionString** abfragen und beim Erstellen des **OLEDBConnection**-Objekts verwenden können:

```
public static void ConnectionstringBuilder() {
    OleDbConnectionStringBuilder ConnectionStringBuilder =
        new OleDbConnectionStringBuilder();
    ConnectionStringBuilder.DataSource = "Suedsturm.mdb";
    ConnectionStringBuilder.Provider =
        "Microsoft.Jet.OLEDB.4.0";
    OleDbConnection cnn = new OleDbConnection(
        ConnectionStringBuilder.ConnectionString);
    try {
        cnn.Open();
        Console.WriteLine("Provider: {0}", cnn.Provider);
        Console.WriteLine("DataSource: {0}",
            cnn.DataSource);
        Console.ReadLine();
        cnn.Close();
    }
```

```
    catch(Exception e) {
        Console.WriteLine("Fehler: {0}", e.Message);
        Console.ReadLine();
    }
}
```

Im Folgenden haben wir die Ausgabe einiger Eigenschaften des **Connection**-Objekts in eine rudimentäre Fehlerbehandlung gesteckt. Diese greift, wenn innerhalb des **try**-Blocks ein Fehler auftritt. Dann wird der **catch**-Block ausgelöst, der eine Fehlermeldung in der Konsole ausgibt. Mehr zum Thema Fehlerbehandlung lesen Sie im Artikel **Fehlerbehandlung mit C#**.

Verbindungszeichenfolge speichern

Wenn Sie unter Access eine Backend-Datenbank verwenden, wird der Pfad zu den verknüpften Tabellen in der Systemtabelle **MSysObjects** gespeichert. Sollten die verknüpften Tabellen nicht an der vorgesehen Stelle verfügbar sein, löst dies einen Fehler aus. Dies fängt man unter Access ab, indem man beim Öffnen der Datenbank prüft, ob die verknüpfte Datenbank sich am vorgesehenen Ort befindet. Falls nicht, blendet man einen Dialog ein, der dem Benutzer die Auswahl des neuen Speicherorts der Backend-Datenbank ermöglicht und verknüpft die Tabellen dann erneut.

Dieses Verhalten wollen wir unter C# für Desktop-Anwendungen ähnlich abbilden – zumindest so, dass es für den Benutzer so aussieht. Intern läuft es natürlich etwas anders: Es gibt ja keine Verknüpfung zu einer Datenbank, sondern wir verwenden eine Zeichenkette, welche den Speicherort der zu nutzenden Datenbank angibt. Zumindest in den bisherigen Beispielen (und der Einfachheit halber auch in den meisten weiteren) legen wir die Datenbank direkt im Verzeichnis der **.exe**-Datei ab. Spätestens, wenn Sie einmal eine Anwendung für den Mehrbenutzerbetrieb auslegen und dabei eine Datei wie eine Access-Datenbank, aber möglicherweise auch eine Excel- oder XML-Datei als Datenquelle nutzen, sollten Sie eine Möglichkeit vorsehen, das Vorhandensein der Quelldatei zu prüfen und gegebenenfalls den Speicherort der Datei neu zu ermitteln.

Hierzu besteht die erste Aufgabe darin, den Pfad zur Quelldatei an einem Ort zu speichern, der außerhalb der kompilierten Anwendung liegt. Anderenfalls müssten Sie die Anwendung ja jedes Mal neu kompilieren, wenn sich der Pfad ändert. Das können Sie bei selbst genutzten Anwendungen tun, aber sicher nicht bei solchen Anwendungen, die Sie an Kunden oder andere Benutzer weitergeben.

Unter Access hätten wir für einen solchen Zweck einfach eine lokale Optionentabelle verwendet oder alternativ eine Textdatei. Eine C#-Desktop-Anwendung jedoch enthält ja keine eigenen Tabellen zum Speichern von Daten, also müssen wir schon auf eine Art Konfigurationsdatei zurückgreifen. Es gibt jedoch gute Nachrichten: C# sieht für Desktop-Anwendungen eine Möglichkeit vor, Konfigurationsdateien zu pflegen und die enthaltenen Daten einfach per Code einzulesen und auch zu ändern.

Wie Sie die Verbindungszeichenfolge in einer Konfigurationsdatei speichern, erfahren Sie im Artikel **Anwendungskonfigurationsdateien**.

Der DataAdapter

Genau wie die übrigen ADO.NET-Objekte gibt es auch den **DataAdapter** wieder in verschiedenen Klassen. Dementsprechend heißt der **DataAdapter** beispielsweise **SqlDataAdapter**, **OdbcDataAdapter** oder **OleDbDataAdapter** – je nachdem, mit welcher Klasse Sie auf welchen Datenbanktyp zugreifen möchten.

Wir beschäftigen uns der Einfachheit halber immer noch mit unserer Access-Beispieldatenbank **Suedsturm.mdb**, also nutzen wir den **OleDbDataAdapter**. Im Folgenden reden wir jedoch neutral von **DataAdapter**, nur im Beispielcode finden Sie entsprechend die Objektbezeichnung **OleDbDataAdapter**.

Was bietet uns der **DataAdapter**? Mit dem **Command**- und dem **DataReader**-Objekt konnten wir ja immerhin schon einmal Auswahlabfragen ausführen und die Ergebnisse vorwärts durchlaufen oder Aktionsabfragen anstoßen. Der

DataReader erlaubt es aber beispielsweise nicht, einen der enthaltenen Datensätze zu ändern und die geänderte Version zurück in die zugrunde liegenden Tabellen zu schreiben. Genauso wenig, wie Sie damit durch die Datensätze navigieren können – es geht nur vorwärts.

Außerdem sind **Command** und **DataReader** Objekte, die eine Verbindung zur Datenbank benötigen. Gerade dies ist natürlich für Web-Anwendungen oder Anwendungen auf mobilen Endgeräten nicht sinnvoll, und auch für Desktop-Anwendungen in Mehrbenutzerumgebungen können nicht endlos viele Verbindungen gleichzeitig offen gehalten werden. Daher sieht ADO.NET einige Objekte vor, mit denen Sie die Daten aus der Datenbank einlesen, bearbeiten und wieder zurückschreiben können. Dabei ist nur für das Einlesen und das Zurückschreiben eine Verbindung nötig, zwischendurch wird die Verbindung getrennt.

Und hier kommt der **DataAdapter** ins Spiel (je nach Anwendungsfall als **SqlDataAdapter**, **OdbcDataAdapter**, **OleDbDataAdapter** et cetera): Er stellt die Verbindung zwischen Client und Server her, um die benötigten Daten aus den Tabellen der Datenbank in lokale Objekte zu übertragen und diese nach Änderungen wieder zurückzuschicken. Dabei stellt der **DataAdapter** nur Verbindungen zum Server her, wenn tatsächlich Daten bewegt werden müssen.

Connection- und Command-Objekte

Die beiden Objekte **Connection** und **Command**, die Sie bereits kennen gelernt haben, dürfen Sie jedoch nicht aus Ihrem Gedächtnis streichen: Das **Connection**-Objekte benötigen wir nämlich nach wie vor, auch um die Verbindungen des **DataAdapters** herzustellen.

Und das **Command**-Objekt ist immer noch gefragt, wenn es darum geht, Änderungen an den Daten einer Tabelle vorzunehmen, für die wir die vorhandenen Daten nicht erst einlesen und analysieren müssen. Eine Aktionsabfrage etwa zum Anfügen eines neuen Datensatzes oder zum Löschen oder Bearbeiten vorhandener Datensätze ist per **Command**-Objekt immer noch schneller, als wenn Sie die entsprechenden Tabellen über den **DataAdapter** in weitere Objekte laden, die Daten dort ändern und die Änderungen dann zurückschreiben.

DataTable und DataRow

Um mit den über den **DataAdapter** gewonnenen Daten zu arbeiten und etwa in einer Schleife alle Datensätze zu durchlaufen und anzuzeigen, benötigen Sie noch entsprechende Objekte, um diese zu speichern. Das **DataTable**-Objekt nimmt dabei das Ergebnis einer Tabelle oder Abfrage auf. Seine **Rows**-Eigenschaft erlaubt den Zugriff auf die einzelnen Datensätze. Diese können Sie wiederum mit dem **DataRow**-Objekt referenzieren, um gezielt auf die Inhalte zuzugreifen.

Im Gegensatz zu den Objekten **Connection**, **Command** und **DataReader**, die ja je nach Anforderung etwa aus einem der Namespaces **System.Data.Sql**, **System.Data.Odbc** oder **System.Data.OleDb** stammen, kommen die beiden Objekte **DataTable** und **DataRow** aus dem Namespace **System.Data**. Diesen fügen Sie also noch über folgende Anweisung zur Klasse hinzu:

```
using System.Data;
```

DataTable füllen und durchlaufen

Damit kommen wir nun endlich zum ersten handfesten Beispiel dieses Artikels. Dabei wollen wir ein **DataTable**-Objekt über einen **DataAdapter** mit den Daten einer Tabelle füllen und die Datensätze durchlaufen und in der Konsole ausgeben (siehe Listing 1).

Die Methode speichert zunächst die Verbindungszeichenfolge und die SQL-Abfrage in den beiden Variablen **strConnection** und **strSQL**.

Dann erzeugt sie ein neues Objekt des Typs **OleDbConnection** und übergibt dabei gleich die zu verwendende Verbindungszeichenfolge, mit der in diesem Beispiel auf die Access-Datenbank **Suedsturm.mdb** im Verzeichnis der C#-Anwendung selbst zugegriffen werden soll.

```
public static void DataTableFuellen() {
    string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    string strSQL = "SELECT * FROM tblKategorien";
    OleDbConnection cnn = new OleDbConnection(strConnection);
    OleDbDataAdapter da = new OleDbDataAdapter(strSQL, cnn);
    DataTable dt = new DataTable();
    da.Fill(dt);
    for (int i = 0; i < dt.Rows.Count; i++) {
        DataRow row = dt.Rows[i];
        Console.WriteLine("{0} {1}", row[0], row[1]);
    }
    Console.ReadLine();
}
```

Listing 1: Füllen und Durchlaufen einer Tabelle per **DataTable**

Den **OleDbDataAdapter** speichert die Methode nach dem Erstellen und der Übergabe der SQL-Anweisung mit **strSQL** und dem **Connection**-Objekt in der Variablen **da**.

Hier fällt auf, dass die Methode die Verbindung nicht explizit öffnet, indem Sie die Open-Methode aufruft. Dies ist nicht nötig, da die Fill-Methode des DataAdapter-Objekts diese Aufgabe für uns übernimmt. Sollten Sie für andere Aktionen in diesem Zusammenhang eine offene Verbindung benötigen, können Sie diese mit Open öffnen, müssen dann aber auch mit Close für das Schließen der Verbindung sorgen.

Das **DataTable**-Objekt erzeugt die Methode zunächst einfach nur und speichert es in der Variablen **dt**. Das Füllen dieses Objekts erfolgt nicht über eine Methode des Objekts selbst, sondern das Objekt wird als Parameter der **Fill**-Methode des **DataAdapter**-Objekts übergeben:

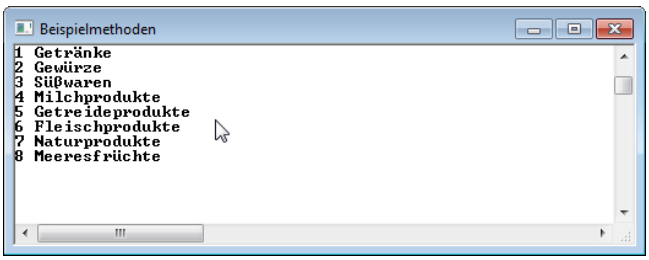


Bild 1: Ausgabe der Daten der Tabelle **tblKategorien**

```
da.Fill(dt);
```

Nun enthält das **DataTable**-Objekt bereits alle Datensätze mit allen Feldern der Tabelle **tblKategorien**. Diese durchläuft die Methode in einer **for**-Schleife mit der Laufvariablen **i**, welche die Werte von **0** bis zur Anzahl der Datensätze minus eins durchläuft. Die Anzahl der Datensätze ermitteln wir hier mit dem Ausdruck **dt.Rows.Count**. **Rows** ist die Auflistung aller Datensätze.

Über diese Auflistung erhalten wir mit dem entsprechenden Indexwert ein **DataRow**-Objekt, das wir mit jedem Schleifendurchlauf neu mit der Variablen **row** referenzieren:

```
DataRow row = dt.Rows[i];
```

Über den Index der Felder gibt die Methode nun die Inhalte der ersten beiden Felder in der Konsole aus:

```
Console.WriteLine("{0} {1}", row[0], row[1]);
```

In dieser Anweisung könnten wir auch konkret auf die Feldnamen zugreifen:

```
Console.WriteLine("{0} {1}", row["KategorieID"],
row["Kategoriename"]);
```

Anschließend gibt die Methode die Liste der Kategorien aus der Tabelle **tblKategorien** in der Konsole aus (siehe Bild 1).

Wir haben nun das **OleDbConnection**-Objekt und die SQL-Anweisung per Konstruktor an das **OleDbDataAdapter**-Objekt übergeben. Sie können auch zunächst ein **OleDbCommand**-Objekt erstellen und dieses dann als Konstruktor beim Erstellen des **OleDbDataAdapter**-Objekts übergeben:

```
string strConnection = "Provider=Microsoft.Jet.
OLEDB.4.0;Data Source=Suedsturm.mdb";
string strSQL = "SELECT * FROM tblKategorien";
OleDbConnection cnn = new OleDbConnection(strConnection);
OleDbCommand cmd = new OleDbCommand(strSQL, cnn);
OleDbDataAdapter da = new OleDbDataAdapter(cmd);
```

Oder Sie verwenden beim Erstellen des **DataAdapter**-Objekts die konstruktorlose Variante und weisen das **OleDbCommand**-Objekt erst im Anschluss über die Eigenschaft **SelectCommand** hinzu:

```
...
OleDbCommand cmd = new OleDbCommand(strSQL, cnn);
OleDbDataAdapter da = new OleDbDataAdapter();
da.SelectCommand = cmd;
```

Warum ist hier von **SelectCommand** die Rede, und nicht einfach von **Command**? Weil der **DataAdapter** ja zunächst einmal Daten liefern soll. Daher soll das **Command**-Objekt eine **Select**-Anweisung enthalten und nicht etwa eine Aktionsabfrage.

Anzahl der Datensätze per Fill

Wenn Sie ein **DataTable**-Objekt mit der **Fill**-Methode des **DataAdapter**-Objekts füllen, liefert die **Fill**-Methode einen Rückgabewert, den Sie hier direkt nutzen können: Es handelt sich nämlich um die Anzahl der zurückgelieferten Datensätze. Im folgenden Ausschnitt der Methode **DataTableFuellenIV** schreiben wir das Ergebnis der **Fill**-Methode in die Variable **anzahl**, die wir dann als Vergleichswert des zweiten Parameters der **for**-Schleife nutzen:

```
...
DataTable dt = new DataTable();
int anzahl = da.Fill(dt);
for (int i = 0; i < anzahl; i++)
{
    DataRow row = dt.Rows[i];
    Console.WriteLine("{0} {1}", row[0], row[1]);
}
...
```

DataTable per foreach durchlaufen

Das Durchlaufen der Datensätze des **DataTable**-Objekts ist eine Variante. Etwas eleganter sieht es mit der **foreach**-Schleife aus. Dazu verwenden wir ein **DataRow**-Objekt als Laufvariable und ermitteln dieses jeweils aus der **Rows**-Auflistung des **DataTable**-Objekts:

```
da.Fill(dt);
foreach (DataRow datensatz in dt.Rows) {
    Console.WriteLine("{0} {1}", datensatz[0],
        datensatz[1]);
}
```

Das DataSet-Objekt

Ein **DataTable**-Objekt nimmt das Ergebnis einer SQL-Abfrage auf. Was aber, wenn man mehrere verknüpfte Tabellen referenzieren und deren Inhalte anzeigen, bearbeiten, ergänzen oder löschen möchte? Wenn Sie etwa einen Artikel zur Tabelle **tblArtikel** hinzufügen möchten und gleichzeitig eine neue Kategorie für diesen Artikel in der Tabelle **tblKategorien** anlegen möchten, können sie dies natürlich über zwei einzelne **DataTable**-Objekte realisieren. Wenn Sie jedoch deren Beziehung automatisch berücksichtigen möchten, nutzen Sie das **DataSet**-Objekt, das ein oder mehrere Tabellen berücksichtigt.

Typisierte und untypisierte DataSets

DataSets kommen in zwei Ausführungen: Im Folgenden besprechen wir zunächst die sogenannten »untypisierten« DataSets. Diese liefern nur die Standardmethoden, -eigenschaften und -ereignisse und sind mit einem Recordset

unter Access vergleichbar. Wenn Sie etwa die Felder referenzieren möchten, um darauf zuzugreifen, geben Sie entweder den Index oder den Namen des jeweiligen Feldes als Parameter in eckigen Klammern an. Unter Access hätten Sie etwa für den Zugriff auf das Feld **Kategoriename** der Tabelle **tblKategorien** in einem Recordset namens **rstKategorien** den Ausdruck **rstKategorien!Kategoriename** oder **rstKategorien("Kategoriename")** verwendet. Sie hätten aber nicht etwa per IntelliSense das Feld **Kategoriename** als Eigenschaft des Objekts **rstKategorien** eingeben können.

So ähnlich ist es bei untypisierten **DataSet**-Objekten: Hier greifen Sie etwa über den Ausdruck **KategorieRow[0]["Kategoriename"]** auf das Feld **Kategoriename** zu. Bei einem typisierten **DataSet**-Objekte legt Visual Studio für Sie eine Klassenstruktur an, die es ermöglicht, per IntelliSense auf die Felder der Tabelle zuzugreifen. Hier sähe der Zugriff dann etwa so aus: **KategorieRow[0].Kategoriename**. Damit erhalten Sie Typsicherheit: Sie können so zum Beispiel nicht etwa durch einen Schreibfehler einen falschen Feldnamen angeben, da die Feldnamen über eine Klasse alle fest definiert sind.

DataSet mit einer Tabelle füllen

Wir schauen uns zunächst an, wie dies mit einer einzelnen Tabelle funktioniert (siehe Listing 2). Dazu verwenden wir wieder die gleiche Verbindungszeichenfolge und SQL-Anweisung wie zuvor, und erstellen ein **Connection**-Objekt und ein **DataAdapter**-Objekt. Dann jedoch legen wir ein neues **DataSet**-Objekt namens **ds** an, dem wir keinen Konstruktor übergeben. Wir füllen es genauso wie das **DataTable**-Objekt, indem wir es der Methode **Fill** des **DataAdapter**-Objekts übergeben:

```
DataSet ds = new DataSet();
int anzahl = da.Fill(ds);
```

Die **Fill**-Methode liefert die korrekte Anzahl der Datensätze zurück, die wir dann in der **for**-Schleife zum Durchlaufen der Datensätze verwenden. Hier gibt es nun in Ermangelung des **DataTable**-Objekts eine kleine Änderung: Wir greifen nämlich auf das erste Element der **Tables**-Auflistung des **DataSet**-Objekts zu und erhalten wiederum genau ein **DataTable**-Objekt, auf dessen Feldinhalte wir über den Index zugreifen können:

```
DataRow row = ds.Tables[0].Rows[i];
Console.WriteLine("{0} {1}", row[0], row[1]);
```

Wir haben hier also den Fall abgebildet, dass ein **DataSet**-Objekt genau ein **DataTable**-Objekt enthält.

Später kommen wir darauf zurück, wie Sie mehrere verknüpfte Tabellen in ein **DataSet** einbetten.

```
public static void DataSetFuellenI() {
    string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    string strSQL = "SELECT * FROM tblKategorien";
    OleDbConnection cnn = new OleDbConnection(strConnection);
    OleDbDataAdapter da = new OleDbDataAdapter(strSQL, cnn);
    DataSet ds = new DataSet(); //DataSet statt DataTable
    int anzahl = da.Fill(ds);
    for (int i = 0; i < anzahl; i++) {
        DataRow row = ds.Tables[0].Rows[i]; //Deshalb hier Zugriff auf ds.Tables[0] statt auf dt wie zuvor
        Console.WriteLine("{0} {1}", row[0], row[1]);
    }
    Console.ReadLine();
}
```

Listing 2: Füllen und Durchlaufen einer Tabelle per **DataSet**

Fenster mit einfachen Tabellendaten

Unter Access haben Sie mit wenigen Mausklicks die Daten einer Tabelle in einem Formular angezeigt – Datenherkunft festlegen, Felder aus der Feldliste in das Formular zeihen, fertig war das Formular. Unter WPF ist es erstens etwas aufwendiger, und zweitens gibt es verschiedene Möglichkeiten, dies zu realisieren. In diesem Artikel schauen wir uns die einfachste Methode an, die Daten einer einfachen Tabelle in einem Fenster anzuzeigen, in den Datensätzen zu navigieren und Änderungen zu speichern.

Wie unter Access gibt es auch unter C# einfache und schnelle Wege, ein Ziel zu erreichen – nämlich mit Unterstützung der Entwicklungsumgebung, meist in Form von Assistenten. Wer später Änderungen am Entwurf seiner Benutzeroberfläche vornehmen möchte, sollte jedoch wissen, was der Assistent dort im Einzelnen gezaubert hat.

Deshalb schauen wir uns in diesem Artikel zunächst an, wie Sie die Daten einer Tabelle so schnell wie möglich in einem Fenster anzeigen. Später erfahren Sie dann, was dort alles geschehen ist und wie Sie ein solches Fenster von Hand erstellen und auch anpassen können.

Voraussetzung

Voraussetzung für die Beispiele dieses Artikels ist das Vorhandensein der Beispieldatenbank **Suedsturm.mdb**, einer Access-Datenbank. Diese sollte im Projektverzeichnis liegen und auch dem Projekt im Projektmappen-Explorer zugewiesen sein, damit dieses beim Debuggen automatisch in den **Debug**-Ordner kopiert wird und dort für den Zugriff beim Testen zur Verfügung steht. Ist dies nicht der Fall, ziehen Sie die Datenbankdatei **Suedsturm.mdb** einfach aus dem Windows Explorer auf den Namen des Projekts im Projektmappen-Explorer von Visual Studio.

Da wir im Folgenden auch Änderungen an Datensätzen vornehmen wollen, soll bei jedem neuen Debuggen die Original-Datenbank erneut in das **Debug**-Verzeichnis kopiert werden. Dazu stellen Sie die Eigenschaft In Ausgabeverzeichnis kopieren des Elements **Suedsturm.mdb** auf den Wert **Immer kopieren** ein (siehe Bild 1). So erscheinen gelöschte oder geänderte beim nächsten Debuggen immer wieder im Ausgangszustand.

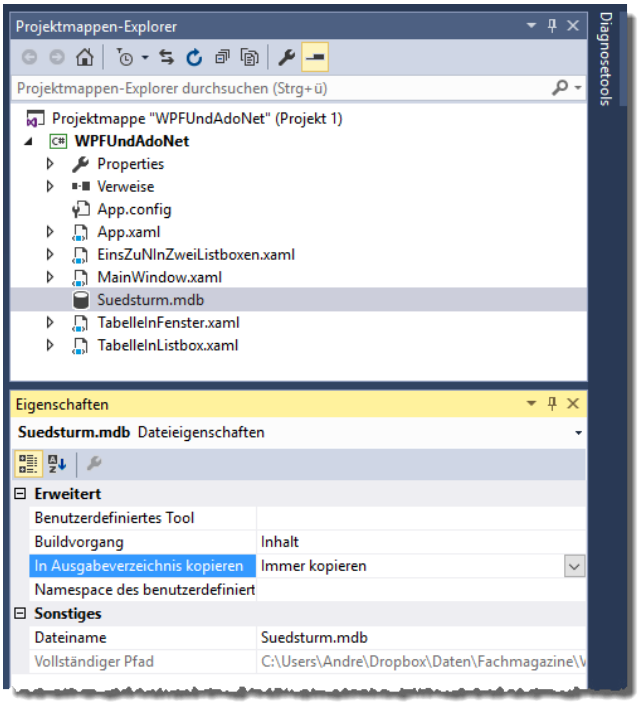


Bild 1: Die Datenbank soll beim Debuggen jeweils erneut ins Debug-Verzeichnis kopiert werden.

Schnelle Datenanzeige

Die einfachste Disziplin beim Anzeigen von Daten unter Access war ein Formular, das die Daten in der Formularansicht anzeigt. Dazu mussten Sie einfach nur die Tabelle markieren, deren Daten Sie anzeigen wollten, und den entsprechenden Assistenten aufrufen. Im Ergebnis erhielten Sie ein neues Formular mit allen Feldern der Datenherkunft plus einer Navigationsleiste, die fest im Formular verbaut war und die

Navigation zwischen den Datensätzen sowie das Anlegen und Löschen erleichterte.

Unter WPF, soviel sei vorab verraten, gibt es keine vorgefertigten Formulare mit Navigationsleisten oder gar einem Datensatzmarkierer wie unter Access. Hier müssen Sie alle Elemente selbst hinzufügen, allerdings hilft die Entwicklungsumgebung bei dem einen oder anderen Schritt.

Datenquelle hinzufügen

Mit dem Menüeintrag **Projekt|Neue Datenquelle hinzufügen...** öffnen Sie den Dialog **Assistent zum Konfigurieren von Datenquellen**. Wählen Sie hier im ersten Schritt den Eintrag **Datenbank** aus (siehe Bild 2).

Im folgenden Schritt haben Sie nur eine sehr eingeschränkte Auswahl (siehe Bild 3). Sie brauchen also nur auf die Schaltfläche **Weiter** zu klicken.

Im folgenden Dialog wählen Sie dann die Datenverbindung aus. In unserem Fall ist dies einfach: Wir müssen einfach nur die bereits als mögliche Datenquelle erkannte Access-Datenbank namens **Suedsturm.mdb** zu bestätigen. Mit einem Klick auf das Plus-Zeichen im unteren Bereich können Sie noch die Verbindungszeichenfolge für diese Verbindung anzeigen lassen (siehe Bild 4).

Schließlich können Sie noch festlegen, dass die Verbindungszeichenfolge in der Anwendungskonfigurationsdatei gespeichert wird (siehe Bild 5).

Im folgenden Schritt wird es interessant. Hier analysiert der Assistent die angegebene Datenbank und stellt alle Objekte der Typen **Tabelle**

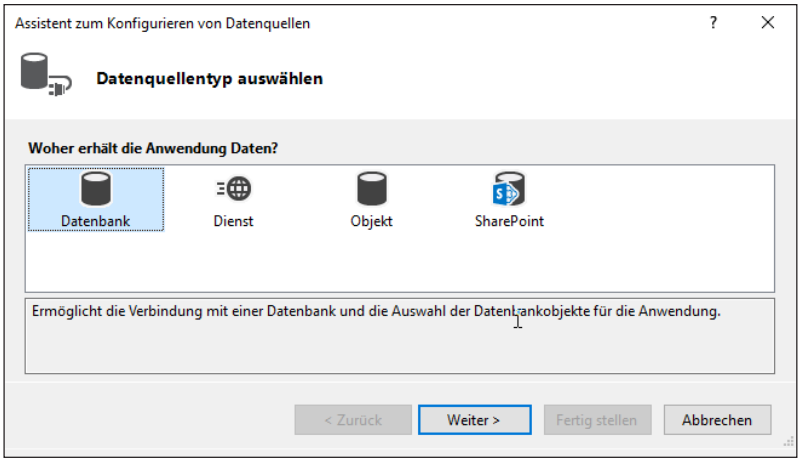


Bild 2: Der erste Schritt im **Assistent zum Konfigurieren von Datenquellen**

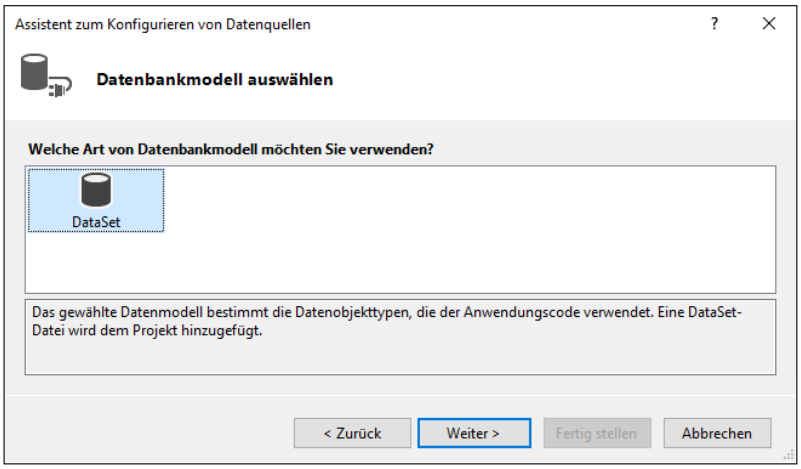


Bild 3: Auswahl der Art des Datenbankmodells

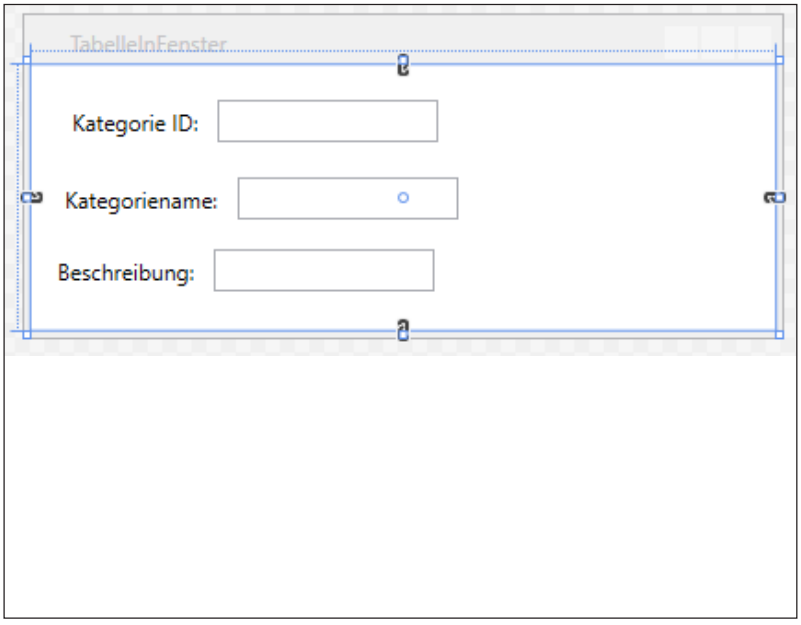


Bild 4: Festlegen der Datenverbindung

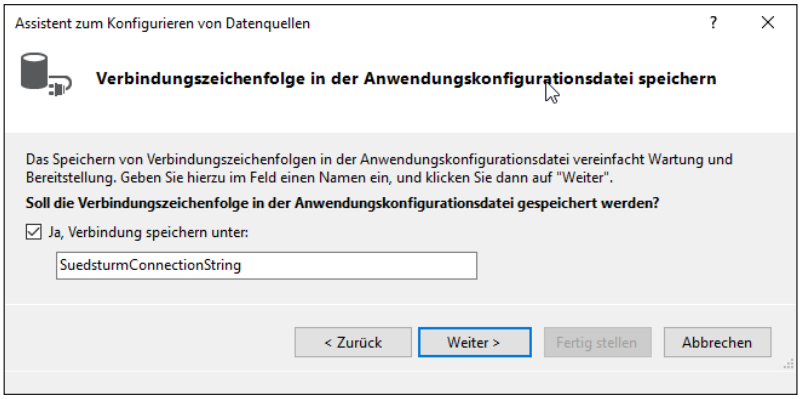


Bild 5: Speichern der Datenverbindung

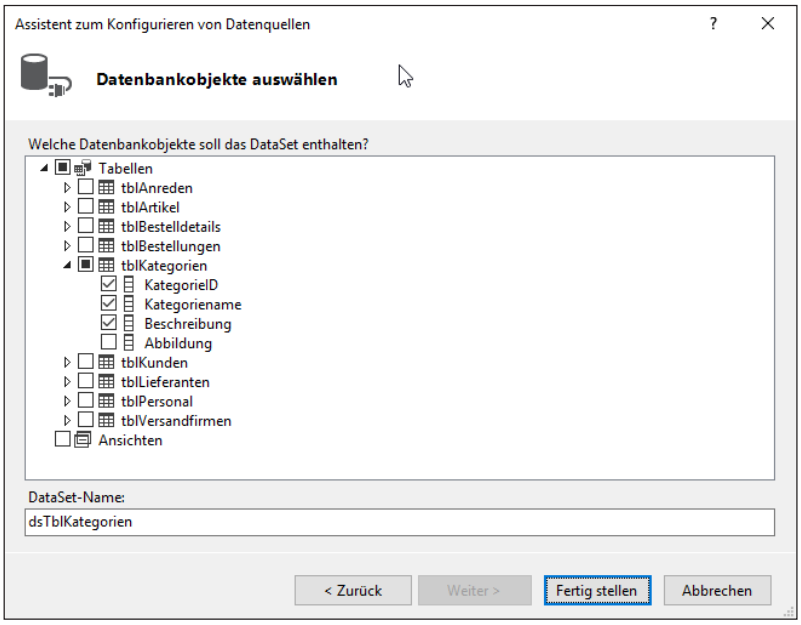


Bild 6: Auswahl der Tabellen/Datenbankobjekte für ein **DataSet**

und **Abfrage** zur Auswahl. Zu jedem Objekt können Sie die Felder auswählen, die zum **DataSet** hinzugefügt werden sollen. Überhaupt kommt hier das erste Mal der Begriff **DataSet** zum Einsatz (siehe Bild 6).

Fürs Erste wollen wir nun die drei Felder **KategorieID**, **Kategoriename** und **Beschreibung** der Tabelle **tblKategorien** zum **DataSet** hinzufügen. Dieses **DataSet** wollen wir dann unter dem Namen **dsTblKategorien** speichern.

Was ist nun geschehen? Der Assistent hat nicht nur einen neuen Eintrag zum Bereich **Datenquellen** hinzugefügt, sondern Sie finden auch im Projektmappen-Explorer ein neues Element. Dieses heißt **dsTblKategorien.xsd** und wird in Visual Studio grafisch wie in Bild 7 dargestellt. Dort finden Sie oben ein Objekt für die einzige Tabelle des **DataSet**-Objekts sowie ein weiteres namens **tblKategorienTableAdapter**, das die beiden Methoden **Fill** und **GetData()** anzeigt. Die untergeordneten Objekte enthalten, kurz gesagt, eine Menge automatisch generierten C#-Codes.

Das Ergebnis der Auswahl der Verbindungszeichenfolge landet übrigens in der Datei **Settings.settings** (siehe Bild 8). Wenn Sie die Verbindungszeichenfolge also entfernen möchten, müssen Sie also hier Hand anlegen.

Nach Abschluss des Assistenten finden Sie außerdem die neue Datenquelle im Bereich **Datenquellen** von Visual Studio vor (siehe Bild 9).

Felder zum Fenster hinzufügen

Mit den erzeugten Elementen können wir nun schon eine Menge erledigen. Der erste Schritt ist, dass Sie ein neues Fenster namens **TabelleInFenster.xaml** anlegen (Menüpunkt **ProjektIFenster hinzufügen**). Das Fenster ist nun noch leer. Nun aktivieren Sie den Bereich **AnsichtIWeitere FensterIDatenquellen** beziehungsweise Tastenkombination **Umschalt + Alt + D** (siehe Bild 10).

Hier haben Sie eine Reihe von Möglichkeiten. Die erste ist, einfach das Objekt **tblKategorien** in das Fenster zu ziehen. Dies würde ein **DataGrid**-Steuerelement zum Fenster hinzufügen, das die drei Felder der Tabelle **tblKategorien** anzeigt. Dies behandeln wir allerdings in einem weiteren Artikel – hier wollen wir uns um die Detailansicht kümmern.

Dazu ziehen Sie die drei Felder **KategorieID**, **Kategoriename** und **Beschreibung** einzeln in den Entwurf des Fensters,

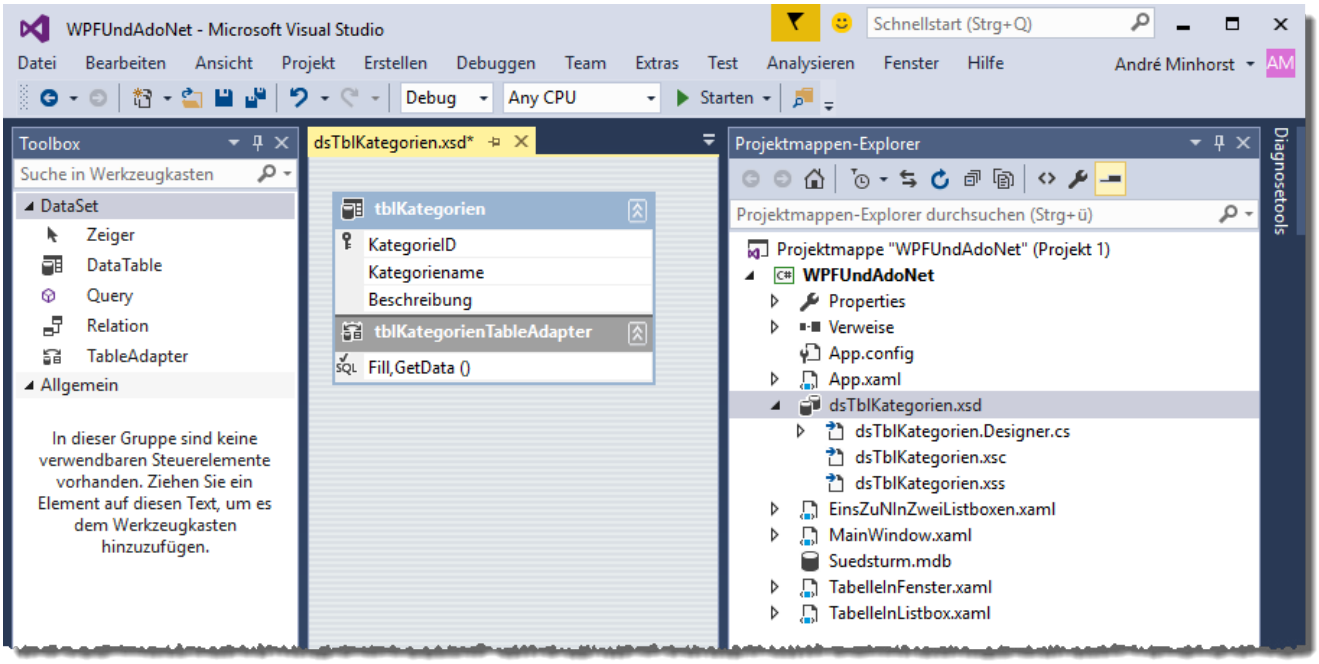


Bild 7: Neue Objekte im Projektmappen-Explorer

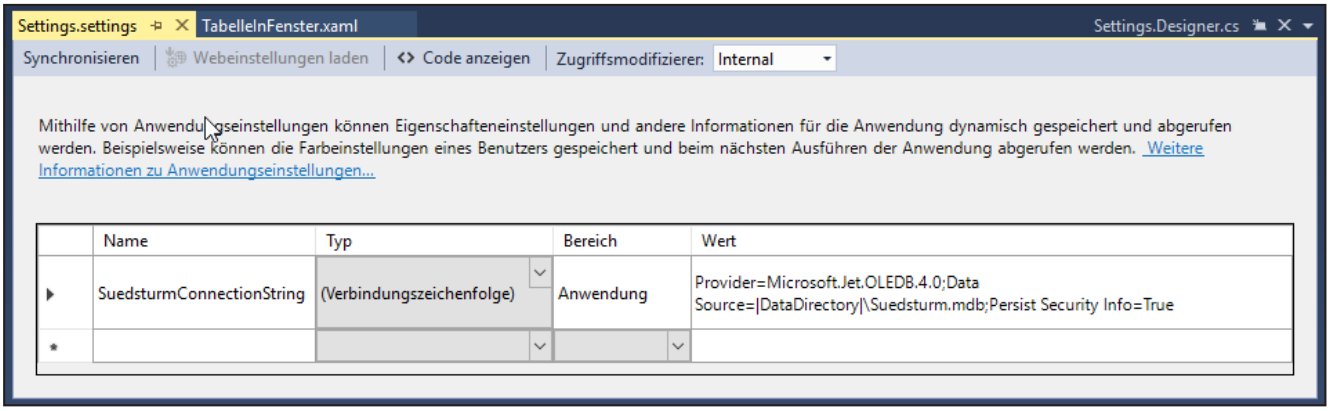


Bild 8: Die Verbindungszeichenfolge wird im Bereich **Settings.settings** gespeichert.

sodass sein Inhalt etwa wie in Bild 11 aussieht. Was ist nun geschehen?

- Visual Studio hat dem **.xaml**-Code eine Verbindung zum soeben erstellen und gespeicherten DataSet **dsTblKategorien** sowie eine **CollectionViewSource**, deren Funktion wir weiter unten erläutern.

- Das **Window**-Element wurde um das Ereignisattribut **Loaded** erweitert: **<Window ... Loaded="Window_Loaded">**. Passend dazu wurde in der Datei **TabelleInFenster.xaml.cs** eine entsprechende Methode angelegt, die beim Eintreten des Ereignisses ausgelöst wird.

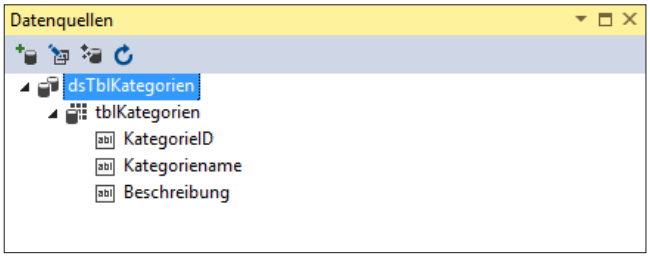


Bild 9: Die neue Datenquelle im Bereich **Datenquellen** von Visual Studio

- Außerdem wurden natürlich die drei **TextBox**-Steuerelemente plus entsprechender **Label**-Elemente angelegt – allerdings jedes in einem eigenen **Grid**-Konstrukt, die keine vernünftige Ausrichtung erlaubt. Dies passen wir noch an.

Steuerelemente organisieren

Schauen wir uns erst die Steuerelemente und die **Grid**-Elemente an, in denen diese angelegt wurden. Alle wurden mit dem Attribut **DataContext="{StaticResource tblKategorienViewSource}"** ausgestattet. Es reicht, wenn wir dieses einmal dem umschließenden **Grid**-Element zuweisen.

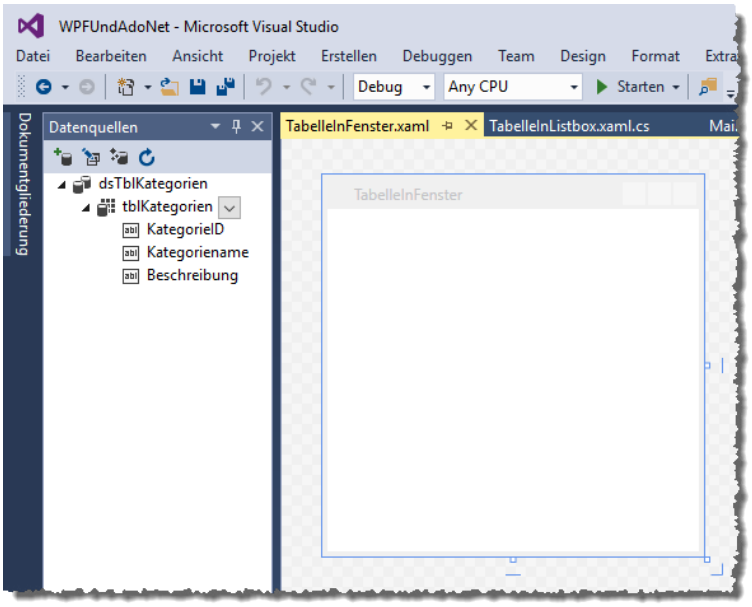


Bild 10: Die Datenquellen und das noch leere Fenster

Dann stellen wir das übergeordnete **Grid**-Steuerelement auf drei Zeilen und zwei Spalten ein. Die übrigen **Grid**-Elemente samt **Grid.ColumnDefinitions**-, **ColumnDefinition**-, **Grid.RowDefinitions**- und **RowDefinition**-Elementen löschen wir.

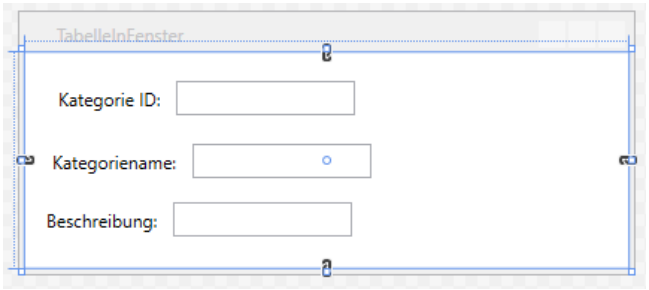


Bild 11: Fenster mit Steuerelementen

Die verbleibenden **Label**- und **TextBox**-Steuerelemente fügen wir in die **Grid**-Zellen ein, indem wir die Attribute **Grid.Column** und **Grid.Row** entsprechend mit den nullbasierten Indizes der Zellen auszeichnen. Außerdem passen wir das Layout noch an. Das Ergebnis im Code sehen Sie in Listing 2.

Datensatz im Fenster anzeigen

Wenn Sie nun das Debugging starten (**F5**), erhalten Sie das Fenster aus Bild 12. Es funktioniert! Das Fenster zeigt den ersten Datensatz der festgelegten Datenquelle an. Das ging nicht so schnell wie mit Access und wir haben ja auch noch keine Navigationsschaltflächen hinzugefügt, aber immerhin sehen wir schon einmal Daten aus der Datenbank.

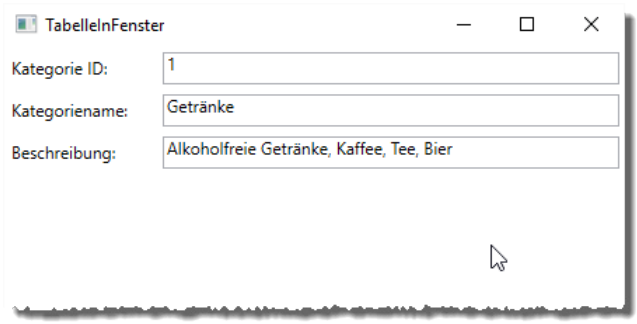


Bild 12: Tabelleninhalte im WPF-Fenster

```
<Window.Resources>
    <WPFUndAdoNet:dsTblKategorien x:Key="dsTblKategorien"/>
    <CollectionViewSource x:Key="tblKategorienViewSource"
        Source="{Binding tblKategorien, Source={StaticResource dsTblKategorien}}"/>
</Window.Resources>
```

Listing 1: Ressourcen-Informationen durch Hinzufügen des ersten Steuerelements zum Fenster

```
<Window xmlns:WPFUndAdoNet="clr-namespace:WPFUndAdoNet" x:Class="WPFUndAdoNet.Properties.TabelleInFenster"
        mc:Ignorable="d" Title="TabelleInFenster" Height="175" Width="412" Loaded="Window_Loaded">
    <Window.Resources>
        <WPFUndAdoNet:dsTblKategorien x:Key="dsTblKategorien"/>
        <CollectionViewSource x:Key="tblKategorienViewSource" Source="{Binding tblKategorien,
            Source={StaticResource dsTblKategorien}}"/>
    </Window.Resources>
    <Grid DataContext="{StaticResource tblKategorienViewSource}">
        <Grid.RowDefinitions>
            <RowDefinition Height="30"/>
            <RowDefinition Height="30"/>
            <RowDefinition Height="30"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="110"/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Label Content="Kategorie ID:" Grid.Column="0" HorizontalAlignment="Left" Grid.Row="0"
            VerticalAlignment="Center"/>
        <TextBox x:Name="kategorieIDTextBox" Grid.Column="1" HorizontalAlignment="Stretch" Height="23" Margin="3,3,3,3"
            Grid.Row="0" Text="{Binding KategorieID, Mode=TwoWay, NotifyOnValidationError=true,
            ValidatesOnExceptions=true}" VerticalAlignment="Center" />
        <Label Content="Kategoriename:" Grid.Column="0" HorizontalAlignment="Left" Grid.Row="1"
            VerticalAlignment="Center"/>
        <TextBox x:Name="kategorienameTextBox" Grid.Column="1" HorizontalAlignment="Stretch" Height="23" Margin="3,3,3,3"
            Grid.Row="1" Text="{Binding Kategoriename, Mode=TwoWay, NotifyOnValidationError=true,
            ValidatesOnExceptions=true}" VerticalAlignment="Center"/>
        <Label Content="Beschreibung:" Grid.Column="0" HorizontalAlignment="Left" Grid.Row="2"
            VerticalAlignment="Center"/>
        <TextBox x:Name="beschreibungTextBox" Grid.Column="1" HorizontalAlignment="Stretch" Height="23" Margin="3,3,3,3"
            Grid.Row="2" Text="{Binding Beschreibung, Mode=TwoWay, NotifyOnValidationError=true,
            ValidatesOnExceptions=true}" VerticalAlignment="Center"/>
    </Grid>
</Window>
```

Listing 2: Code des WPF-Fensters

Navigationsschaltflächen hinzufügen

Damit stellen wir uns natürlich nicht zufrieden, denn wer zeigt schon immer den ersten Datensatz in einem Fenster an? Wir benötigen Schaltflächen, mit denen Sie zum ersten und zum letzten Datensatz navigieren können sowie zum vorherigen und zum folgenden Datensatz. Und damit kommen wir auch zu dem bereits von Visual Studio beim Hineinziehen des ersten Feldes aus der Datenquelle angelegten Code in der Datei **TabellenInFenster.xaml.cs**.

Wenn Sie sich dort die Methode **Window_Loaded** ansehen, finden Sie einige Anweisungen, die – um einige führende Angaben erleichtert – wie in Listing 3 aussehen.

Die erste Anweisung erzeugt ein Objekt auf Basis der Klasse **dsTblKategorien**, die in der beim Erstellen der Datenquelle erzeugten Klasse **dsTblKategorien.Designer.cs** steckt. Die Klasse erbt von der Klasse **DataSet** des Namespaces **Windows.Data** und wird hier genau wie ein **DataSet** verwendet.

Methodenstarter als Vorlage

In Ausgabe 2/2015 haben wir unter dem Titel »**Experimentieren mit der Konsole**« eine Technik vorgestellt, mit der Sie die Methoden einer Klasse direkt in der Konsole auflisten und dort zur Ausführung auswählen können. Der Nachteil war, dass Sie damit nur eine Klasse mit fest vorgegebenem Klassennamen referenzieren konnten. Dies haben wir nun erweitert: mit einer neuen »Main«-Methode, die beim Start der Anwendung aufgerufen wird und alle Methoden aller Klassen außer der Klasse »Program« selbst auflistet und so ermöglicht, diese zu starten.

Im vorliegenden Beitrag nun stellen wir Ihnen eine neue Version der Technik aus Experimentieren mit der Konsole vor, mit der Sie die zu testenden Methoden bequem in anderen Klassen anlegen, die sich optimalerweise auch noch in einer anderen Datei befinden. Damit wird das Ausprobieren von Code über einfache C#-Methoden noch einfacher als zuvor.

Alles, was Sie tun müssen, ist das Erstellen eines neuen Projekts als Konsolen-Projekt, dessen Datei **Program.cs** Sie mit dem Code aus Listing 1 füllen.

Listing 2 liefert den Rest des notwendigen Programmcodes. Anschließend brauchen Sie nur noch ein neues Klassenmodul anzulegen und dort die zu testende Methode einzufügen – beispielsweise die folgende:

```
[Beschreibung("Dies ist die erste Testmethode.")]
public static void Test1()
{
    Console.WriteLine("Test erfolgreich!");
    Console.ReadLine();
}
```

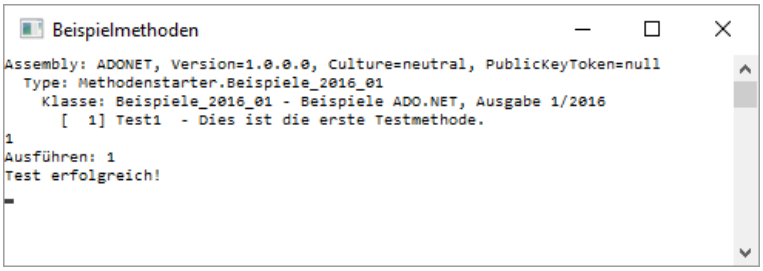


Bild 1: Anzeige der Beispielmethoden einer Klasse

Wenn Sie in Ihrem neuen Klassenmodul den gleichen Namespace verwenden, in dem sich auch die Klasse **Program** mit der Methode **Main** befindet (in diesem Beispiel also den Namespace namens **Methodenstarter**), sind Sie nun bereits fertig und können das Projekt zum Debuggen starten. Das Ergebnis sieht dann wie in Bild 1 aus.

Sollten Sie im neuen Klassenmodul hingegen einen neuen Namespace verwenden, müssen Sie noch den Namespace mit der **Main**-Methode etwa per **using**-Direktive bekanntgeben:

```
using Methodenstarter;
```

Schnelles Testprojekt erstellen

Nun möchten Sie vielleicht gelegentlich auf die Schnelle ein neues Projekt erstellen, um mal eben ein paar C#-Befehle zu testen oder auch um die Beispiele aus einem Artikel aus dem **DATENBANKENTWICKLER** auszuprobieren.

In diesem Fall haben Sie die hier vorgestellten Methoden vielleicht gerade nicht greifbar. Damit Sie immer auf die Methode **Main** zugreifen können, legen wir einfach eine Vorlage an, die genau dies liefert. Die neue Vorlage finden Sie dann im **Neues Projekt**-Fenster.

Aber wie erstellen Sie eine Projektvorlage? Das ist gar nicht so schwer: Als Erstes müssen Sie das Projekt erstellen, das alle Elemente für die gewünschte Vorlage enthält, und dieses Projekt dann in eine Vorlage umwandeln. Dazu wählen Sie einfach den

```
using System;
using System.Reflection;
using System.Collections;

namespace Methodenstarter {
    class Program {
        static void Main(string[] args) {
            int j = 0; int i = 1; int intType = 1;
            string strMethode = ""; string strBeschreibung = "";
            ArrayList Methoden = new ArrayList(); ArrayList Typen = new ArrayList();
            Type[] types; Type currentType = null;
            do {
                i = 1;
                Assembly assembly = Assembly.GetEntryAssembly();
                Console.WriteLine("Assembly: {0}", assembly.FullName);
                types = FindAllExportedTypesInNamespaceFor(assembly);
                Array.ForEach(types, delegate (Type type) {
                    if ((type.Name!="Program") & (type.Name!="Beschreibung")) {
                        intType = ++intType;
                        currentType = type;
                        Console.WriteLine("  Type: {0}", type.FullName);
                        Beschreibung beschreibung = (Beschreibung)type.GetCustomAttribute(typeof(Beschreibung));
                        if (beschreibung != null) { strBeschreibung = " - " + beschreibung.Text; }
                        else { strBeschreibung = ""; }
                        Console.WriteLine("    Klasse: {0}{1}", type.Name, strBeschreibung);
                        MethodInfo[] myArrayMethodInfo = type.GetMethods(BindingFlags.Public | BindingFlags.Static |
                            BindingFlags.Instance | BindingFlags.DeclaredOnly);
                        strMethode = "";
                        foreach (MethodInfo methodInfo in myArrayMethodInfo) {
                            beschreibung = (Beschreibung)methodInfo.GetCustomAttribute(typeof(Beschreibung));
                            if (beschreibung != null) { strBeschreibung = " - " + beschreibung.Text; }
                            else { strBeschreibung = ""; }
                            strMethode = methodInfo.Name;
                            Console.WriteLine("      [{0,3}] {1} {2}", i++, strMethode, strBeschreibung);
                            Methoden.Add(strMethode);
                            Typen.Add(type);
                        }
                    }
                });
                String strMethodeNummer = Console.ReadLine();
                if (int.TryParse(strMethodeNummer, out j)) {
                    if (j < i & j > 0) {
                        Console.WriteLine("Ausführen: {0}", j);
                        strMethode = (String)Methoden[j - 1];
                        currentType = (Type)Typen[j - 1];
                        currentType.InvokeMember(strMethode, BindingFlags.InvokeMethod, null, currentType, null);
                    }
                }
                else {
                    j = 0;
                }
            } while (j > 0);
        }
```

Listing 1: Methode, die alle übrigen Klassen nach Methoden durchsucht und diese zum Ausführen auflistet (Teil 1)