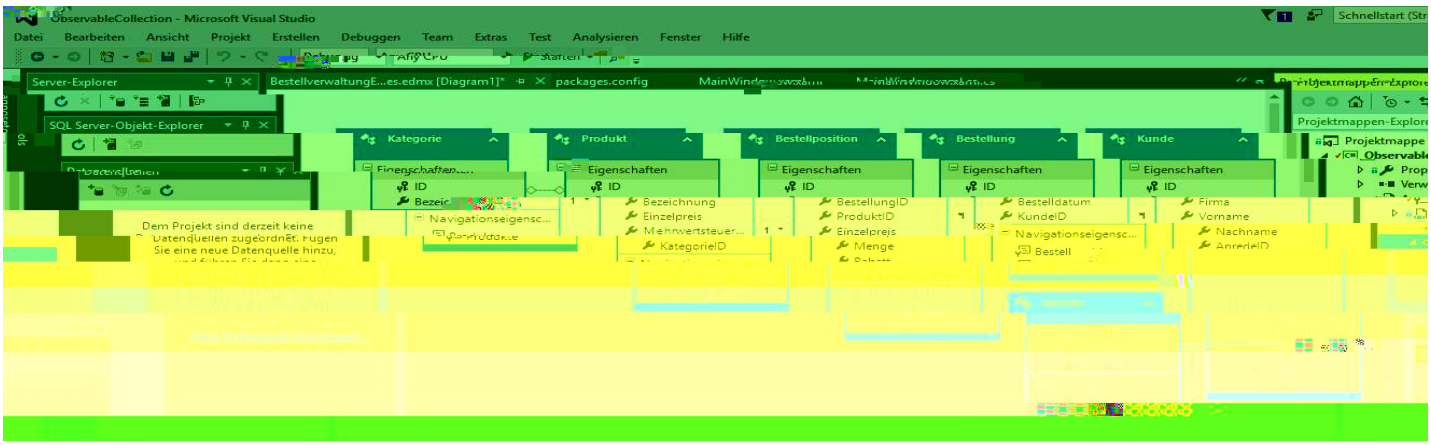
AUSGABE 01/2018

DATENBANK ENTWICKLER

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



TOP-THEMEN:

WPF-CONTROLS	Das DatePicker-Steuerelement	SEITE 3
WPF-CONTROLS	Das ListView-Steuerelement	SEITE 11
VB-BASICS	Mit Klassen programmieren	SEITE 23
DATENZUGRIFF	EDM: Der Code First-Ansatz	SEITE 36
LÖSUNGEN	Adressen verwalten	SEITE 55

André Minhorst Verlag

WPF-STEUERELEMENTE	Das DatePicker-Steuerelement	3
	Das ListView-Steuerelement	11
	ListView: Sortierungen	18
VISUAL BASIC-GRUNDLAGEN	Visual Basic: Mit Klassen programmieren	23
DATENZUGRIFFSTECHNIK	EDM: Der Code First-Ansatz	36
	EDM: Code First – Datenbank erweitern	46
LÖSUNGEN	Adressen verwalten	55
SERVICE	Impressum	2
DOWNLOAD	Die Downloads zu dieser Ausgabe finden Sie unter folgendem Link: **http://www.amvshop.de** Klicken Sie dort auf **Mein Konto**, loggen Sie sich ein und wählen dann **Meine Sofortdownloads**.	

Impressum

DATENBANKENTWICKLER
© 2015-2018 André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

Das DatePicker-Steuerelement

Für die Eingabe von Datumsangaben gibt es unter WPF ein spezielles Steuerelement – das sogenannte DatePicker-Control. Es ist recht schlicht gehalten, bietet aber eine gute und zuverlässige Möglichkeit, Datumsangaben schnell auszuwählen. Da es ein spezialisiertes Steuerelement ist, offeriert es jedoch auch einige individuelle Attribute, die Sie kennen sollten, um das Element effizient zu nutzen. Anderenfalls kommt es schnell zu Problemen ...

In einer Datenbankanwendung ist die Wahrscheinlichkeit sehr hoch, dass Sie früher oder später Datumsangaben eingeben müssen. Unter Access wurde für Felder, die auf einem Datumsfeld basierten, ein Textfeld angelegt, das beim Fokuserhalt eine Schaltfläche zum Öffnen eines kleinen Popups zur Datumsauswahl präsentierte. Bei dem **DatePicker**-Steuerelement von WPF ist diese kleine Schaltfläche immer sichtbar, sodass der Benutzer direkt weiß, dass er bei der Datumseingabe Unterstützung findet. Ein **DatePicker**-Element ist schnell hinzufügt – entweder über die ToolBox, wo es unter **Alle WPF-Steuerelemente** zu finden ist oder durch das Hinzufügen eines neuen Elements des Typs **DatePicker** zum XAML-Code – im einfachsten Fall wie folgt:

```
<DatePicker />
```

Das Steuerelement sieht dann im Entwurf wie in Bild 1 aus.

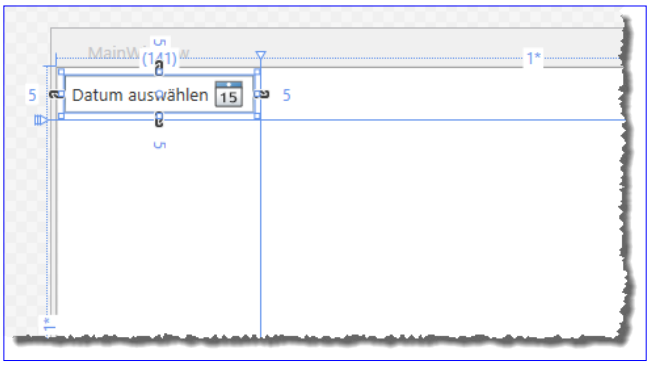


Bild 1: Das **DatePicker**-Element in der Entwurfsansicht

Wenn Sie die Anwendung starten, liefert das **DatePicker**-Steuerelement den Text **Datum auswählen**. Ein Klick auf die Schaltfläche rechts neben dem Eingabefeld zeigt dann ein Popup mit der Ansicht des aktuellen Monats an (siehe Bild 2).

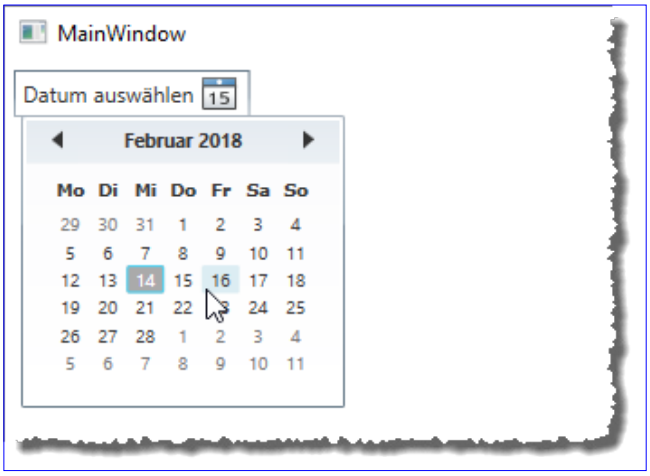


Bild 2: Das **DatePicker**-Element in Aktion

Besondere Eigenschaften des DatePicker-Elements

Das **DatePicker**-Element liefert einige Eigenschaften, die für die Programmierung interessant sind.

- **SelectedDateChanged**: Diese Ereignis wird ausgelöst, wenn sich der Wert des Datums ändert.
- **IsDropDownOpen**: Gibt den aktuellen Zustand des Kalender-Popups wieder und erlaubt auch die Einstellung der Eigenschaft.

- **SelectedDateFormat**: Gibt das angezeigte Datumsformat aus.
- **CalendarOpened**: Wird ausgelöst, wenn der Kalender geöffnet wird.
- **CalendarClosed**: Wird ausgelöst, wenn der Kalender geschlossen wird.
- **DateValidationError**: Wird ausgelöst, wenn ein ungültiges Datum eingegeben wird.

Ereignis beim Ändern des Datums

Wenn der Benutzer das Datum im **DatePicker**-Element anpasst, löst er damit das Ereignis **SelectedDateChanged** aus, welches wir mit der folgenden Methode implementieren können. Das neue Datum wird dann in einem Meldungsfenster angezeigt:

```
Private Sub dpDatepicker_SelectedDateChanged(sender As Object, e As SelectionChangedEventArgs)
    Dim datePicker As DatePicker
    datePicker = sender
    MessageBox.Show("Das neue Datum lautet: " + datePicker.Text)
End Sub
```

Zuvor legen wir das Attribut für das **DatePicker**-Element wie folgt an:

```
<DatePicker x:Name="dpDatepicker" SelectedDateChanged="DatePicker_SelectedDateChanged"></DatePicker>
```

Datumsformat ermitteln

Die **MessageBox** der obigen Ereignismethode können wir noch erweitern. Sie soll nun auch noch das Format des angezeigten Datums ausgeben:

```
MessageBox.Show("Das neue Datum lautet: " + datePicker.Text + vbCrLf + "und hat das Format: " + datePicker.SelectedDate-
Format.ToString)
```

Mögliche Werte sind **Short** und **Long** – hier mit Beispielen:

- **Short**: **15.02.2018**
- **Long**: **Donnerstag, 15. Februar 2018**

Kalender per Code aufklappen

Vielleicht möchten Sie den Kalender automatisch aufklappen – zum Beispiel, wenn sich das Fenster öffnet oder wenn der Benutzer ein ungültiges Datum eingegeben hat. Dann können Sie die Eigenschaft **IsDropDownOpen** nutzen. Diese liefert sowohl den aktuellen Zustand als auch die Möglichkeit, den Zustand durch Zuweisen eines **Boolean**-Wertes zu ändern. Die Schaltfläche, die wir im Fenster neben dem **DatePicker** untergebracht haben, macht genau das – sie klappt den Kalender auf:

```
Private Sub btnKalenderAnzeigen_Click(sender As Object, e As RoutedEventArgs)
    dpCatepicker.IsDropDownOpen = True
End Sub
```

Ereignisse bei Öffnen und Schließen des Kalenders

Die beiden Ereignisse **CalendarOpened** und **CalendarClosed** werden ausgelöst, wenn der Kalender angezeigt oder ausgeblendet wird. Sie werden wie folgt zum Element hinzugefügt:

```
<DatePicker x:Name="dpDatepicker" SelectedDateChanged="dpDatepicker_SelectedDateChanged" CalendarOpened="dpDatepicker_CalendarOpened" CalendarClosed="dpDatepicker_CalendarClosed"></DatePicker>
```

Die beiden Ereignisse programmieren wir wie folgt, um den aktuellen Zustand des Kalenders im Ausgabe-Bereich auszugeben:

```
Private Sub dpDatepicker_CalendarOpened(sender As Object, e As RoutedEventArgs)
    Debug.WriteLine("Der Kalender wurde geöffnet.")
End Sub

Private Sub dpDatepicker_CalendarClosed(sender As Object, e As RoutedEventArgs)
    Debug.WriteLine("Der Kalender wurde geschlossen.")
End Sub
```

Ungültige Eingaben abfangen

Wenn Sie in der Standardkonfiguration ein ungültiges Datum wie etwa **30.2.2018** eingeben, wird das Textfeld ohne Kommentar wieder geleert. Das überhaupt etwas geschieht, erfahren Sie im Ausgabe-Bereich – hier erscheint die folgende Meldung:

```
Ausnahme ausgelöst: "System.FormatException" in mscorlib.dll
```

Diese Ausnahme können Sie mit einer herkömmlichen Fehlerbehandlung (**Try...Catch**) nicht abfangen. Allerdings bietet das **DatePicker**-Element eine eigene Ereignismethode an, um auf solche Fehleingaben zu reagieren, und zwar das Ereignis **DateValidationError**:

```
<DatePicker x:Name="dpDatepicker" ... DateValidationError="dpDatepicker_DateValidationError"></DatePicker>
```

Mit der ersten Version der Ereignismethode zeigen wir nur die Fehlerart an, in diesem Fall **System.FormatException**:

```
Private Sub dpDatepicker_DateValidationError(sender As Object, e As DatePickerDateValidationErrorEventArgs)
    MessageBox.Show("Fehler: " + e.Exception.GetType.ToString)
End Sub
```

Geben wir in der Methode den Inhalt der Eigenschaft **Message** aus, erhalten wir die Meldung aus Bild 3.

Bestimmten Datumsbereich anzeigen

Normalerweise zeigt der Kalender folgenden Monat an:

- den aktuellen Monat, wenn noch kein Wert für das Steuerelement ausgewählt wurde oder
- den Monat, in dem sich der im Steuerelement angegebene Tag befindet.

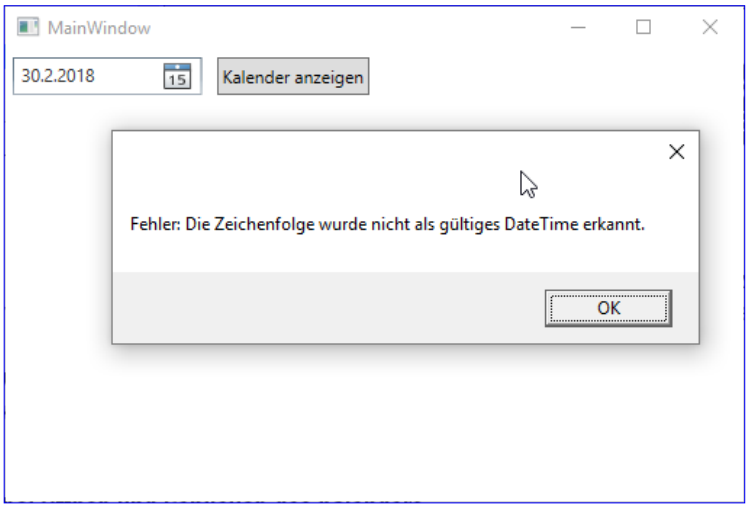


Bild 3: Fehler bei ungültigem Datumsformat

Sie können dies jedoch auch variieren, indem Sie die Eigenschaften **DisplayDateStart** und **DisplayDateEnd** nutzen. Damit können Sie nicht nur den gewünschten Monat einstellen, sondern auch kürzere Bereiche. Das folgende Beispiel zeigt nur eine einzige Woche wie in Bild 4 an:

```
Private Sub btnKalenderMitAnderemDatum_Click( _
        sender As Object, e As RoutedEventArgs)
    With dpDatepicker
        .DisplayDateStart = "22.01.2018"
        .DisplayDateEnd = "28.01.2018"
        .SelectedDate = "23.01.2018"
        .IsDropDownOpen = True
    End With
End Sub
```

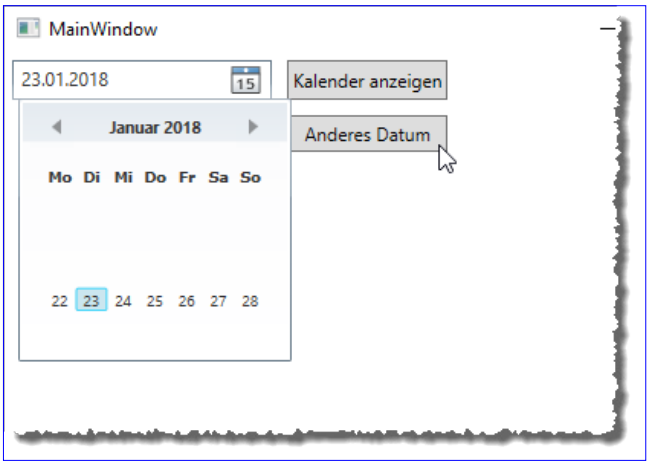


Bild 4: Anzeige nur einer einzigen Woche

Sie können jedoch maximal einen Zeitraum von sechs Wochen anzeigen.

Bestimmte Tage sperren

Sie können auch bestimmte Tage oder auch Datumsbereiche als gesperrt markieren. Wenn Sie etwa den **23.1.2018** sperren wollen, weil Sie dann Geburtstag haben, öffnen Sie den Kalender wie folgt:

```
Private Sub btnTageDeaktivieren_Click(sender As Object, e As RoutedEventArgs)
    With dpDatepicker
        .BlackoutDates.Add(New CalendarDateRange(New DateTime(2018, 1, 23)))
        .DisplayDateStart = "01.01.2018"
        .DisplayDateEnd = "31.01.2018"
        .IsDropDownOpen = True
    End With
End Sub
```

Das ListView-Steuerelement

Es gibt unter WPF sechs Steuerelemente, die eine Auswahl untergeordnete Elemente ermöglichen – ComboBox, ListBox, ListView, DataGrid, TabControl und Ribbon. In diesem Artikel schauen wir uns das ListView-Steuerelement an, das gegebenüber dem ListBox-Element einige Möglichkeiten zur einfacheren Darstellung der enthaltenen Daten liefert, und zwar über die View-Eigenschaft. Im Vergleich zur ListBox bietet die ListView außerdem standardmäßig die erweiterte Auswahl an – Sie können also direkt mehrere Einträge statt nur einem einzigen selektieren. Dieser Artikel zeigt, wie Sie das ListView-Steuerelement zur Anzeige von Daten aus einer Collection von Entitäten anzeigen können.

Das **ListView**-Steuerelement bietet eine ziemlich einfache Möglichkeit, die Daten etwa aus einer **ObservableCollection** zur Auswahl anzubieten. Für die Anzeige einiger Daten am Beispiel von Adressdaten haben wir die folgende Klasse für das Fenster **MainWindow.cs** eines frischen WPF-Projekts angelegt. Dieses hält eine **ObservableCollection** mit Elementen des Typs **Adresse** namens **Adressen** als öffentliche Eigenschaft bereit und füllt diese Collection in der Konstruktor-Methode mit einigen Elementen:

```
Class MainWindow
    Public Property Adressen As ObservableCollection(Of Adresse)
    Public Sub New()
        InitializeComponent()
        Adressen = New ObservableCollection(Of Adresse)
        Adressen.Add(New Adresse With {.Anrede = "Herr", .EMail = "andre@minhorst.com", .Firma = "André Minhorst Verlag", _
             .ID = 1, .Land = "Deutschland", .Nachname = "Minhorst", .Ort = "Duisburg", .PLZ = "47137", _
             .Strasse = "Borkhofer Str. 17", .Telefon = "0123/123456", .Vorname = "André"})
        '...weitere Adressen
        DataContext = Me
    End Sub
End Class
```

Die Klasse **Adresse.vb** sieht etwa wie folgt aus:

```
Public Class Adresse
    Public Property ID As Integer
    Public Property Firma As String
    Public Property Anrede As String
    Public Property Vorname As String
    Public Property Nachname As String
    Public Property Strasse As String
```

```
    Public Property PLZ As String
    Public Property Ort As String
    Public Property Land As String
    Public Property EMail As String
    Public Property Telefon As String
End Class
```

Damit können wir schon das **Window**-Element mit dem **ListView**-Element bestücken. Damit dieses die Daten der Auflistung in der **ObservableCollection** anzeigt, benötigen wir ein paar Unterelemente. Als Erstes stellen wir jedoch für das **ListView**-Element die Datenherkunft ein, und zwar mit dem Attribut **ItemsSource**.

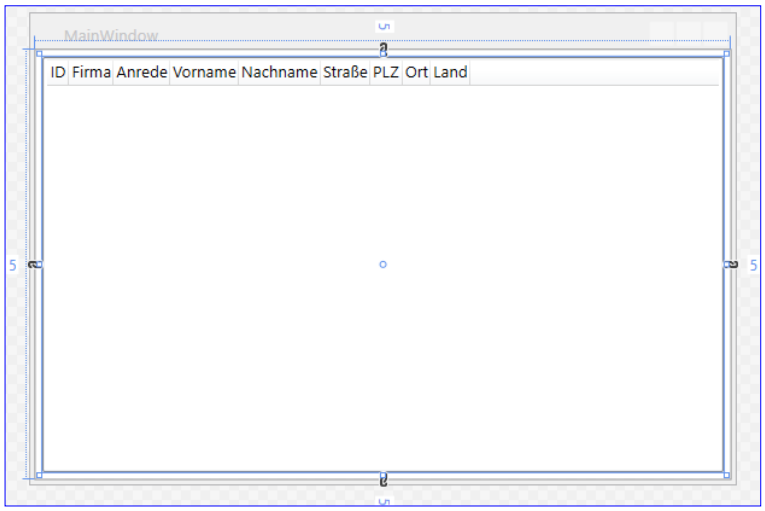


Bild 1: Entwurf des **ListView**-Elements

Dieses legen wir auf den Wert **{Binding Adressen}** ein, was eine Bindung zu der mit der öffentlichen Eigenschaft **Adressen** der Code behind-Klasse erzeugt. Dann fügen wir diesem über ein Property Element die Eigenschaft **View** hinzu (**<ListView.View>**). Dieses erhält schließlich das Element **GridView** und die untergeordneten Spaltenbeschreibungen **GridViewColumn**.

In diesen Elementen legen wir mit der Eigenschaft **Header** die Spaltenüberschriften und mit der Eigenschaft **DisplayMemberBinding** und dem Wert **{Binding <Feldname>}** den Inhalt des **ListView**-Elements fest:

```
<Window x:Class="MainWindow" ...        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <ListView ItemsSource="{Binding Adressen}" Margin="5">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ID}" />
                    <GridViewColumn Header="Firma" DisplayMemberBinding="{Binding Firma}" />
                    <GridViewColumn Header="Anrede" DisplayMemberBinding="{Binding Anrede}" />
                    <GridViewColumn Header="Vorname" DisplayMemberBinding="{Binding Vorname}" />
                    <GridViewColumn Header="Nachname" DisplayMemberBinding="{Binding Nachname}" />
                    <GridViewColumn Header="Straße" DisplayMemberBinding="{Binding Strasse}" />
                    <GridViewColumn Header="PLZ" DisplayMemberBinding="{Binding PLZ}" />
                    <GridViewColumn Header="Ort" DisplayMemberBinding="{Binding Ort}" />
                    <GridViewColumn Header="Land" DisplayMemberBinding="{Binding Land}" />
                </GridView>
            </ListView.View>
        </ListView>
    </Grid>
</Window>
```

Das Ergebnis sieht dann im Entwurf zunächst wie in Bild 1 aus.

Starten Sie nun die Anwendung, erhalten Sie bereits die mit den Adressen gefüllte Liste aus Bild 2.

Hier ein kurzer Hinweis darauf, dass wir für das Anlegen der **GridViewColumn**-Elemente die Standardeigenschaft des **GridView**-Elements nutzen, nämlich **Columns**. Ausgeschrieben sähe der Code in gekürzter Form wie folgt aus:

```
<GridView>
    <GridView.Columns>
        <GridViewColumn Header="ID"
DisplayMemberBinding="{Binding ID}" />
    </GridView.Columns>
</GridView>
```

Eigenschaften des ListView-Steuerelements

Damit können wir uns den Eigenschaften und Möglichkeiten des **ListView**-Steuerelements zuwenden. Mit dem Steuerelement in der Standardversion wie mit obigem Code erzeugt können wir noch nicht allzu viel anfangen – außer den folgenden Aktionen:

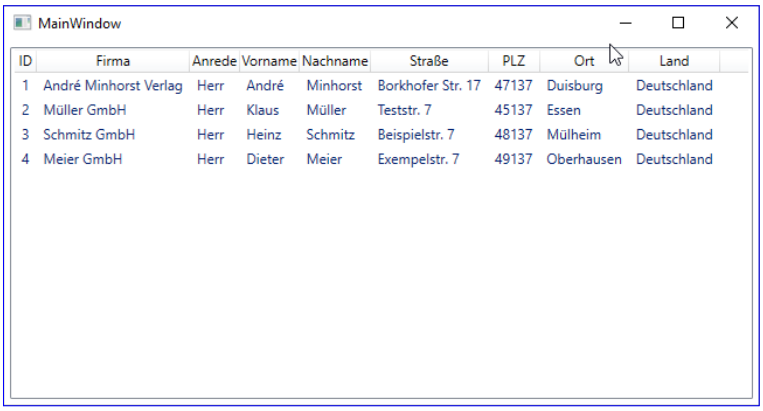


Bild 2: Das **ListView**-Element in Aktion

Bild 3: Änderung der Sortierung der Spalten

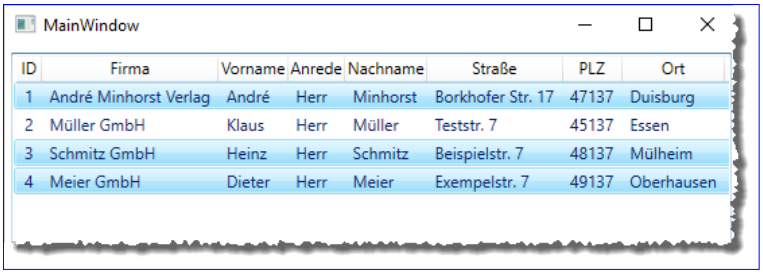


Bild 4: Erweiterte Mehrfachmarkierung

- die Spaltenbreiten durch Ziehen der vertikalen Striche zwischen den Spalten,
- Ändern der Reihenfolge der Spalten durch ziehen einer Spalten per Drag and Drop an die gewünschten Stelle (siehe Bild 3) und
- Einstellen der optimalen Spaltenbreite durch einen Doppelklick auf den vertikalen Strich rechts von der anzupassenden Spalte.

Davon abgesehen ergeben sich keine Möglichkeiten ohne zusätzliche Programmierung. Praktischerweise passt das Steuerelement die Spaltenbreiten auch automatisch an die Inhalte an. Und, wie oben bereits erwähnt, können Sie im **ListView**-Element ohne weiteres Zutun mehrere Elemente gleichzeitig auswählen – im Falle von Bild 4 durch Anklicken der Einträge bei gedrückter **Strg**-Taste.

Inhalte formatieren mit dem DataTemplate-Element

Mit dem **GridViewColumn**-Element sind die Gestaltungsmöglichkeiten beschränkt. Angenommen, wir möchten noch den Inhalt des Feldes **EMail** hinzufügen und diesen wie einen Hyperlink formatieren. Dann können wir das mit dem folgenden XAML-Code realisieren:

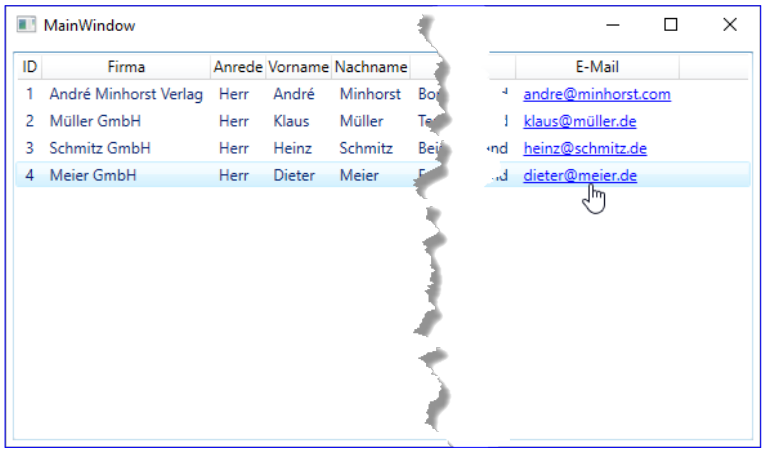


Bild 5: Hervorhebung des Inhalts einer Spalte

Statt der einfachen **GridViewColumn**-Zeile mit der Eigenschaft **DisplayMemberBinding** legen wir nun ein paar Unterelemente an, zuletzt mit einem **TextBlock**-Element, dessen **Text**-Attribut wir mit der Bindung an die Objekteigenschaft **EMail** füllen:

```
<GridViewColumn Header="E-Mail">
    <GridViewColumn.CellTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding EMail}" TextDecorations="Underline" Foreground="Blue" Cursor="Hand"></TextBlock>
        </DataTemplate>
    </GridViewColumn.CellTemplate>
</GridViewColumn>
```

Die in Bild 5 abgebildete Formatierung des Textes in der Spalte **E-Mail** erhalten wir über die drei Attribute **TextDecorations**, **Foreground** und **Cursor**.

Spaltenüberschriften ausrichten

Wenn Sie nicht zufrieden sind mit der Optik der Spaltenüberschriften, können Sie diese leider nicht direkt über Eigenschaften in den **GridViewColumn**-Elementen anpassen, da diese die entsprechenden Eigenschaften nicht anbieten. Allerdings können wir eine Vorgehensweise nutzen, die wir bisherigen Artikel dazu genutzt haben, gleich für mehrere Elemente die gleichen Standardwerte für Eigenschaften vorzubelegen – und zwar mit der Element Property **Resources** des **ListView**-Elements. Im folgenden Code fügen wir dem **ListView**-Element ein **ListView.Resources**-Element hinzu, in dem wir in einem **Style**-Element die Eigenschaft **Left** für die Eigenschaft **HorizontalContentAlignment** definieren und somit die Ausrichtung der Spaltenüberschriften auf Links festlegen:

```
<ListView ItemsSource="{Binding Adressen}" Margin="5">
    <ListView.Resources>
        <Style TargetType="GridViewColumnHeader">
            <Setter Property="HorizontalContentAlignment" Value="Left"></Setter>
        </Style>
    </ListView.Resources>
```

ListView: Sortierungen

Das ListView-Steuerelement bietet standardmäßig keine Möglichkeit, die angezeigten Einträge nach dem Inhalt zu sortieren. Wie Sie eine Funktion zum Sortieren der Daten hinzufügen, zeigen wir Ihnen anhand zweier Beispiele. Das erste nutzt einfache Schaltflächen, um die Daten nach dem Inhalt einer bestimmten Spalte zu sortieren. Die zweite sieht etwas professioneller aus und fügt jeder Spalte die Möglichkeit hinzu, die enthaltenen Daten per Mausklick auf den Spaltenkopf abwechselnd auf- oder absteigend zu sortieren.

Voraussetzung

Wir setzen auf dem Beispielprojekt an, das wir mi Artikel Das **ListView**-Steuerelement beschrieben haben. Hier haben wir ein **ListView**-Element mit den Daten einer Auflistung namens **Adressen** mit Elementen des Typs **Adresse** gefüllt.

Sortierung per Klick auf die Spaltenköpfe

Das Listenfeld im Windows Explorer liefert die Möglichkeit, per Klick auf die Spaltenköpfe in der Dateiliste die Sortierung anzupassen (siehe Bild 1). Dies wollen wir für das **ListView**-Element auch realisieren. Dazu nutzen wir eine neue Schaltfläche namens **btnSortByFirstName**, die wir wie folgt definieren:

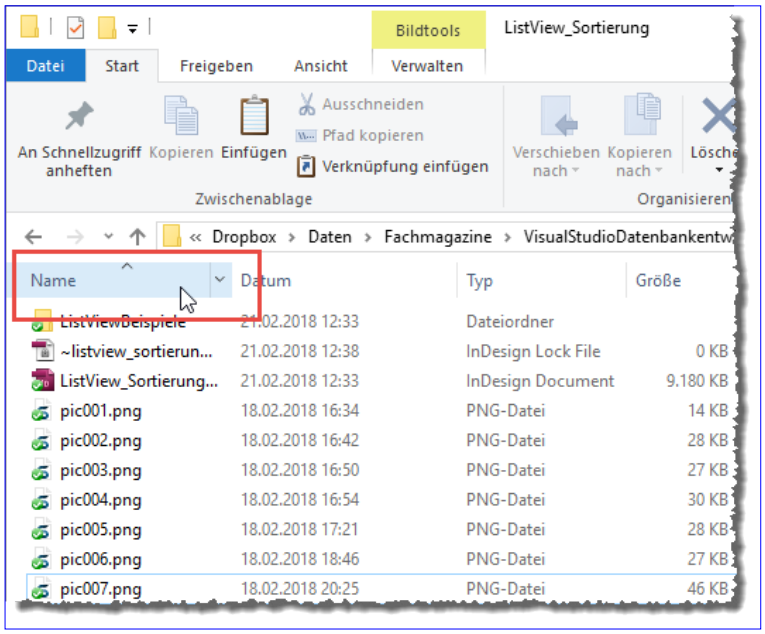


Bild 1: Sortierung im Windows Explorer

```
<Button x:Name="btnSortByFirstNameAsc" Click="btnSortByFirstNameAsc_Click">Aufsteigend nach Vorname sortieren</Button>
```

Und wir legen direkt noch eine Schaltfläche für die umgekehrte Sortierung an:

```
<Button x:Name="btnSortByFirstNameDesc" Click="btnSortByFirstNameDesc_Click">Absteigend nach Vorname sortieren</Button>
```

Für die Schaltflächen legen wir die folgenden beiden Ereignismethoden an:

```
Private Sub btnSortByFirstNameAsc_Click(sender As Object, e As RoutedEventArgs)
    Dim sortDescription As SortDescription
    sortDescription = New SortDescription("Vorname", ListSortDirection.Ascending)
    lvwAdressen.Items.SortDescriptions.Clear()
    lvwAdressen.Items.SortDescriptions.Add(sortDescription)
```

```
End Sub

Private Sub btnSortByFirstNameDesc_Click(sender As Object, e As RoutedEventArgs)
    Dim sortDescription As SortDescription
    sortDescription = New SortDescription("Vorname", ListSortDirection.Descending)
    lvwAdressen.Items.SortDescriptions.Clear()
    lvwAdressen.Items.SortDescriptions.Add(sortDescription)
End Sub
```

Diese legen ein neues Element des Typs **SortDescription** und übergeben diesem den Namen der zu sortierenden Spalte und die Sortierreihenfolge. Dann leeren sie die bisherigen Sortierungen in der Auflistung **SortDescriptions** einer weiteren Auflistung namens **Items** und fügen das neue Objekt über die Variable **sortDescription** zur Auflistung **SortDescriptions** hinzu. Als Ergebnis können Sie die Sortierreihenfolge der Spalte **Vorname** nun in beiden Richtungen festlegen (siehe Bild 2).

Mehrere Kriterien

Sie können nun auch mehrere Kriterien festlegen. Dazu erstellen Sie einfach immer weitere Objekte des Typs **SortDescription** und fügen diese der **SortDescriptions**-Auflistung hinzu. Wenn Sie möchten, dass die Daten erst nach dem Nachnamen und dann nach dem Vornamen aufsteigend sortiert werden, was bedeutet, dass erst nach dem Nachnamen sortiert wird und erst dann nach dem Vornamen, wenn es mehrere gleiche Nachnamen gibt, verwenden Sie etwa den folgenden Code:

```
Dim view As CollectionView
Dim sortDescription As SortDescription
view = CollectionViewSource.GetDefaultView(lvwAdressen.ItemsSource)
view.SortDescriptions.Clear()
sortDescription = New SortDescription("Nachname", ListSortDirection.Ascending)
view.SortDescriptions.Add(sortDescription)
```

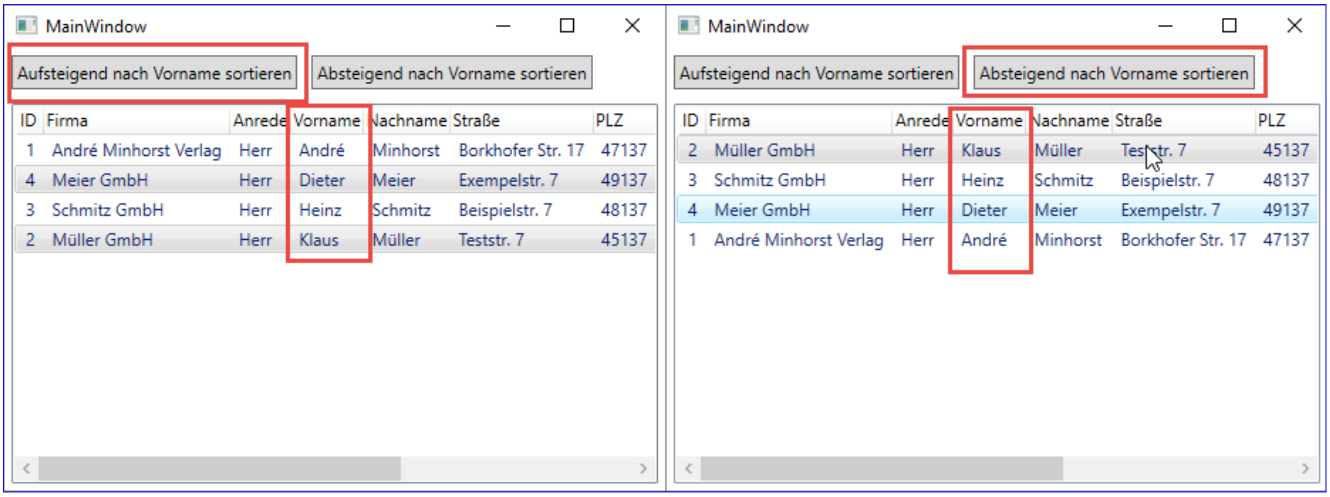


Bild 2: Zwei verschiedene Sortierungen

```
sortDescription = New SortDescription("Vorname", ListSortDirection.Ascending)
view.SortDescriptions.Add(sortDescription)
```

Sortieren über die Spaltenköpfe

Nun müssten wir allerdings eine Menge Schaltflächen einbauen, wenn wir flexibel nach allen Feldern des **ListView**-Elements sortieren wollten. Das machen wir etwas eleganter, indem wir die Spaltenköpfe dazu nutzen. Dazu müssen wir diese zunächst per XAML etwas besser greifbar machen:

```
<ListView x:Name="lvwAdressen" ItemsSource="{Binding Adressen}" ...>
    <ListView.View>
        <GridView>
            <GridView.Columns>
                <GridViewColumn DisplayMemberBinding="{Binding ID}" >
                    <GridViewColumn.Header>
                        <GridViewColumnHeader Tag="ID" Click="lvwAdressenHeader_Click">ID</GridViewColumnHeader>
                    </GridViewColumn.Header>
                </GridViewColumn>
                ...
                <GridViewColumn>
                    <GridViewColumn.Header>
                        <GridViewColumnHeader Tag="EMail" Click="lvwAdressenHeader_Click">E-Mail</GridViewColumnHeader>
                    </GridViewColumn.Header>
                    <GridViewColumn.CellTemplate>
                        <DataTemplate>
                            <TextBlock Text="{Binding EMail}" TextDecorations="Underline" Foreground="Blue"
                                Cursor="Hand"></TextBlock>
                        </DataTemplate>
                    </GridViewColumn.CellTemplate>
                </GridViewColumn>
            </GridView.Columns>
        </GridView>
    </ListView.View>
</ListView>
```

Dazu haben wir hier die einfachen Anweisungen wie **<GridViewColumn Header="ID" DisplayMemberBinding="{Binding ID}" />** etwas aufgebohrt. Dazu haben wir die Eigenschaft **Header** als Element Property ausgeführt, also als **GridViewColumn.Header**.

Dieser haben wir dann ein **GridViewColumnHeader**-Element hinzugefügt, für das wir das **Tag**-Attribut mit dem Namen des referenzierten Feldes der Datenherkunft gefüllt haben. Außerdem haben wir die Ereigniseigenschaft **Click** mit einem Verweis auf die Methode **lvwAdressenHeader_Click** gefüllt. Auf diese Weise können wir nun eine Ereignismethode anlegen, die per

Visual Basic: Mit Klassen programmieren

Wer bisher mit VBA oder C# gearbeitet hat und zu VB wechseln möchte, sieht sich bei der Entwicklung von WPF-Anwendungen einigen Änderungen gegenüber. Dieser Artikel liefert die Grundlagen zum Umgang mit Namespaces und zur Programmierung von Klassen. Zum Experimentieren mit den Beispielen nutzen wir das Tool LINQPad 5.

Klassen

Im Gegensatz zu C# werden Klassen unter Visual Basic nicht durch eine geschweifte Klammer eingeleitet und beendet, sondern wir arbeiten mit dem für Visual Basis typischen Start- und Endzeilen:

```
Class Beispiel
    ---
End Class
```

Class-Elemente können dabei mit folgenden Modifizierern ausgestattet werden:

- **Public**: Die Klasse ist innerhalb der Assembly und auch von anderen Assemblies aus sichtbar.
- **Friend**: Die Klasse ist nur innerhalb der Assembly sichtbar, zu der sie gehört. Dies ist der Standard-Modifizierer, wenn Sie dem Schlüsselwort **Class** weder **Public** noch **Friend** oder einen anderen Modifizierer voranstellen!

Klasse erstellen

Eine Klasse wird typischerweise in einer eigenen Code-Datei angelegt. Wenn Sie ein neues WPF-Fenster anlegen, fügt Visual Studio beispielsweise eine eigene Datei mit der Dateiendung **.xaml.vb** hinzu, welche als Code behind-Modul für das WPF-Fenster dient.

Wenn Sie eine neue Klasse benötigen, die nicht als Code behind-Modul für ein WPF-Element dient, erledigen Sie dies normalerweise über den Dialog **Neues Element hinzufügen...**, den Sie am einfachsten durch Markieren des Projekts im Projektmappen-Explorer und anschließendes Betätigen der Tastenkombination **Strg + Umschalt + A** aufrufen.

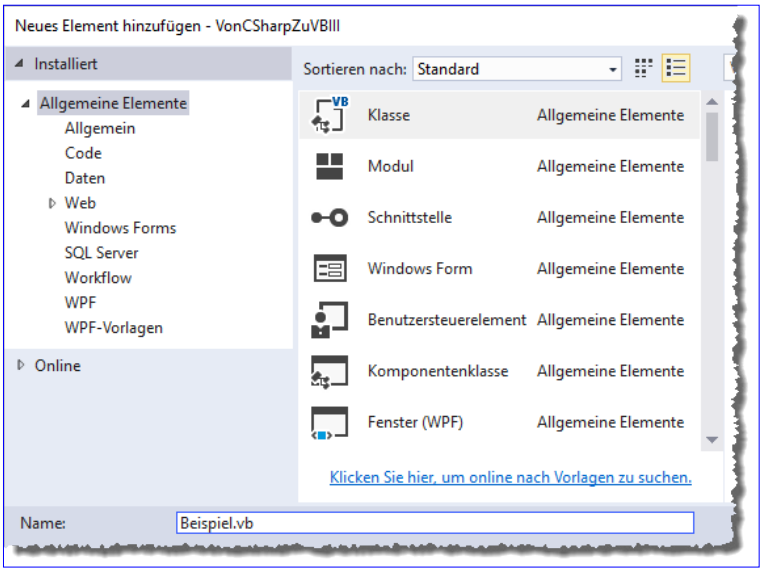


Bild 1: Einfügen einer neuen Klassendatei

Hier wählen Sie dann den Eintrag **Klasse** aus und geben den gewünschten Namen an, in diesem Fall **Beispiel** (siehe Bild 1).

Die neue Klasse wird dann mit dem **Public**-Schlüsselwort ausgezeichnet. Wenn Sie nicht wollen, dass die Klasse in anderen Assemblies genutzt werden kann, ändern Sie **Public** also in **Friend**:

```
Public Class Beispiel

End Class
```

Objekte auf Basis von Klassen initialisieren

Es gibt verschiedene Arten, eine Klasse zu instanzieren. Wenn Sie den Zeitpunkt der Deklaration und der Initialisierung trennen-möchten, nutzen Sie zwei Anweisungen:

```
Dim Beispielklasse As Beispiel
Beispielklasse = New Beispiel
```

Anderenfalls haben Sie die Wahl zwischen den folgenden beiden Schreibweisen. Die erste kennen Sie von VBA:

```
Dim Beispielklasse As New Beispiel
```

Die zweite ist eine Kombination aus dem Zweizeiler:

```
Dim Beispielklasse As Beispiel = New Beispiel
```

Weitere Klassen anlegen

Wenn Sie weitere Klassen anlegen wollen, haben Sie die Wahl: Sie können jeweils eine neue Datei mit jeder Klasse erstellen, was wie oben beschrieben funktioniert. Theoretisch können Sie aber auch mehrere Klassen in einer Datei speichern. Sie können neue Klassen auch innerhalb bestehender Klassen anlegen. Das sieht dann etwa wie folgt aus:

```
Public Class Beispiel
    Public Class UntergeordneteKlasse
    End Class
End Class
```

Mit Klassen experimentieren

Die schönste Benutzeroberfläche auch zum Experimentieren mit Klassen ist das Tool **LINQPad**. Damit können Sie, was für die Beispiele dieses Artikels sinnvoll ist, unter Language den Eintrag **VB Program** auswählen. Es wird dann automatisch eine **Sub Main** eingerichtet, unterhalb derer Sie die Klassen dieses Artikels definieren können. In **Sub Main** greifen Sie dann ganz einfach auf die Klassen zu (siehe Bild 2). Wenn Sie auf die **Start**-Schaltfläche klicken, wird **Sub Main** ausgeführt. Wir fügen der untergeordneten Klasse einfach eine öffentliche Eigenschaft hinzu:

```
Public Class Beispiel
    Public Class UntergeordneteKlasse
        Public Property Test As String = "bla"
    End Class
End Class
```

Die untergeordnete Klasse können Sie etwa wie folgt deklarieren und initialisieren:

```
Dim UntergeordneteKlasse As Beispiel.UntergeordneteKlasse
UntergeordneteKlasse = New Beispiel.UntergeordneteKlasse
Debug.WriteLine (UntergeordneteKlasse.Test)
```

Für solche untergeordneten Klassen gibt es neben den Modifizierern **Public** und **Friend** noch drei weitere mögliche Modifizierer. Diese Modifizierer gelten natürlich auch für die anderen Elemente einer Klasse, also beispielsweise Methoden, Eigenschaften oder Ereignisse:

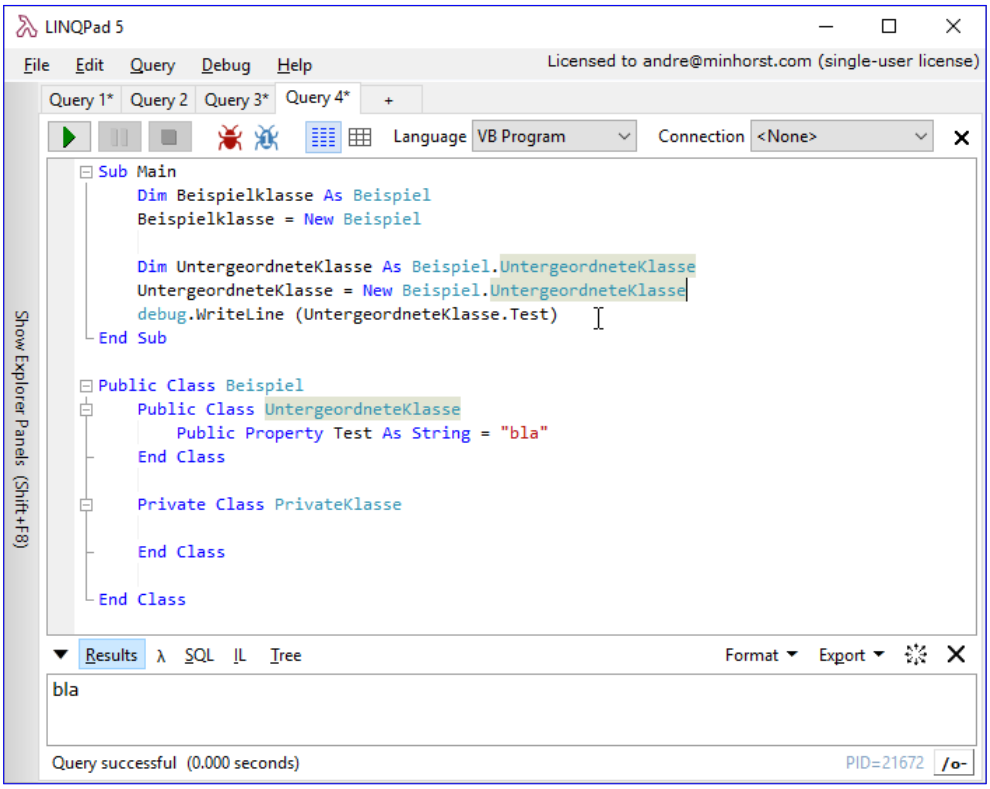


Bild 2: Experimentieren mit Klassen in LINQPad

- **Private**: Die Klassen, Methoden, Ereignisse und Eigenschaften können nur innerhalb des Objekts genutzt werden, in der diese auch definiert wurde. Außerhalb der übergeordneten Klasse ist sie nicht sichtbar. Wenn Sie die Klasse **UntergeordneteKlasse** von oben als **Private** deklarieren würden, könnten Sie nicht von der Methode **Public Main** darauf zugreifen.

- **Protected**: Die Klasse, Methoden, Ereignisse und Eigenschaften können nur innerhalb des Objekts genutzt werden, indem diese definiert wurden, und in Objekten auf Basis von Klassen, die von der ursprünglichen Klasse abgeleitet sind. Ableitungen von Klassen stellen wir später noch vor.

- **Protected Friend**: Die Klasse, Methoden, Ereignisse und Eigenschaften können nur innerhalb des Objekts und der Assembly genutzt werden, indem diese definiert wurden, und in Objekten auf Basis von Klassen, die von der ursprünglichen Klasse abgeleitet sind.

Felder von Klassen

Sie können einer Klasse sogenannte Felder hinzufügen, indem Sie direkt unterhalb der **Class**-Ebene Variablen deklarieren. Dies können Variablen mit dem Modifizierer **Public**, **Private** oder einem der anderen Modifizierer sein. **Public**-Felder können Sie

auch von außerhalb lesen und schreiben. Die Klasse im folgenden Beispiel hat eine öffentliche und eine private Variable. Die öffentliche ist von außen per IntelliSense verfügbar und kann sowohl geschrieben als auch abgefragt werden:

```
Sub Main
    Dim KlasseMitFeld As KlasseMitFeld
    KlasseMitFeld = New KlasseMitFeld
    KlasseMitFeld.Zeichenkette = "Test"
    Debug.WriteLine(KlasseMitFeld.Zeichenkette)
End Sub

Public Class KlasseMitFeld
    Public Zeichenkette As String
    Private Zahl As Integer
End Class
```

Felder als nur lesbar markieren

Wenn ein Feld nur lesenden, aber keinen schreibenden Zugriff ermöglichen soll, deklarieren Sie es mit dem Schlüsselwort **ReadOnly**:

```
Public ReadOnly ZeichenketteNurLesbar As String
```

Diese liefert dann beim Zugriff von außen den Wert **null** zurück. Schreibend können Sie von außen nicht auf dieses Feld zugreifen, und von innerhalb der Klasse, in der es definiert ist, auch nur sehr eingeschränkt. Sie können es in der Deklarationszeile mit dem gewünschten Wert initialisieren:

```
Public ReadOnly ZeichenketteNurLesbar As String = "Nur lesbar"
```

Die Konstruktor-Methode

Die einzige weitere Alternative bei schreibgeschützten Feldern ist die Verwendung der Konstruktor-Methode. Also schauen wir uns diese gleich hier an. Die Konstruktor-Methode ist wie unter C# die Methode, die beim Initialisieren eines Objekts auf Basis der Klasse automatisch ausgeführt wird.

Unter C# heißt diese wie die Klasse selbst, unter Visual Basic ist dies ganz anders: Hier lautet der Name der Konstruktor-Methode schlicht **New**. Die Konstruktor-Methode der folgenden Klasse **KlasseMitFeld** weist dem Feld **ZeichenketteNurLesbar** eine Zeichenkette zu. Die Methode **Sub Main** initialisiert die Klasse und gibt dann als Beweis die zugewiesene Zeichenkette aus:

```
Sub Main
    Dim KlasseMitFeld As New KlasseMitFeld
    Debug.WriteLine(KlasseMitFeld.ZeichenketteNurLesbar)
End Sub
```

```
Public Class KlasseMitFeld
    Public ReadOnly ZeichenketteNurLesbar As String
    Public Sub New
        ZeichenketteNurLesbar = "Wert von der Konstruktor-Methode zugewiesen"
    End Sub
End Class
```

Die Konstruktor-Methode ist ein von vielen VBA-Entwicklern schmerzlich vermisstes Feature: Damit können Sie einer Klasse direkt beim Instanzieren wichtige Informationen übermitteln, die dann auch direkt verarbeitet werden können – beispielsweise, um einer **Kunde**-Klasse direkt den Namen und die Adresse des Kunden zu übermitteln.

Eigenschaften

Felder können fast wie Eigenschaften genutzt werden – es gibt schreib- und lesbar Felder oder auch nur lesbare Felder (**Read-Only**). Eigenschaften erlauben aber eine genauere Steuerung der Berechtigungen und erlauben auch noch das Ausführen weiterer Aktionen, wenn der Eigenschaftswert geschrieben oder gelesen wird. Eine Eigenschaft enthält in der Regel eine lokale Variable, die den Eigenschaftswert speichert (die sogenannte Member-Variable), sowie ein öffentliches **Property**-Konstrukt, welches jeweils einen sogenannten Getter und einen Setter enthält. Das sieht dann insgesamt wie folgt aus:

```
Public Class KlasseMitEigenschaften
    Private _eigenschaft As String
    Public Property Eigenschaft As String
        Get
            Return _eigenschaft
        End Get
        Set(ByVal value As String)
           _eigenschaft = value
        End Set
    End Property
End Class
```

Damit sieht die Notation erheblich anders aus als unter C#, und auch wer von VBA kommt, muss sich etwas umorientieren. Unter VBA gab es noch jeweils eine **Property Get**- und eine **Property Set**-Eigenschaft, wobei die **Property Get**-Eigenschaft den Rückgabewert an eine Variable mit dem Namen der Eigenschaft übergeben hat. Der schreibende und lesende Zugriff auf diese Eigenschaft erfolgt dann etwa wie folgt:

```
Dim klasseMitEigenschaften = New KlasseMitEigenschaften
klasseMitEigenschaften.Eigenschaft = "Test"
Debug.WriteLine(klasseMitEigenschaften.Eigenschaft)
```

Wenn Sie eine einfache schreib- und lesbare Eigenschaft wie oben definieren wollen, können Sie das übrigens auch per Einzeiler erledigen – als sogenannte Auto-Implemented Property:

```
Public Property Eigenschaft As String
```

Genau genommen ist das die schnellste und übersichtlichste Art, eine einfache Property zu definieren. Sie brauchen die andere Schreibweise im Grunde nur, wenn Sie entweder für den Setter oder den Getter zusätzlich auszuführende Anweisungen benötigen, die beim Setzen oder Lesen der Eigenschaft benötigt werden. Oder wenn Sie die Property mit eingeschränkten Zugriffsberechtigungen versehen wollen – dazu später mehr.

Selbst, wenn Sie innerhalb der Klasse auf den Inhalt der privaten Variablen zugreifen wollen, welche den der Property zugewiesenen Wert enthält, brauchen Sie nur die einfache Variante zu nutzen. Im folgenden Beispiel definieren wir eine Klasse mit einer Property namens **Test** als Auto-Implemented Property.

Eine zweite Property namens **Test2** soll den Wert der Variablen **_Test** zurückliefern. **_Test** ist normalerweise die private Membervariable zum Speichern des Wertes der Property **Test**. Sie ist hier nicht explizit deklariert. Dennoch können Sie über diese Variable nach dem Zuweisen eines Wertes den Wert der Property **Test** lesen und beispielsweise über die Property **Test2** zurückgeben:

```
Sub Main
    Dim klasseMitProperty As New KlasseMitProperty
    klasseMitProperty.Test = "bla"
    Debug.WriteLine(klasseMitProperty.Test2)
End Sub

Public Class KlasseMitProperty
Public Property Test As String
    Public Readonly Property Test2 As String
        Get
            Return _Test
        End Get
    End Property
End Class
```

Property mit Getter und Setter schnell anlegen

Visual Studio hat eine tolle Vereinfachung für das Anlegen von Properties mit **Get**- und **Set**-Element. Dabei gehen Sie so vor:

- Schreiben Sie die Property in der kurzen Schreibweise, also also Auto-Implemented Property, in die erste Zeile:
  Public Property Test As String

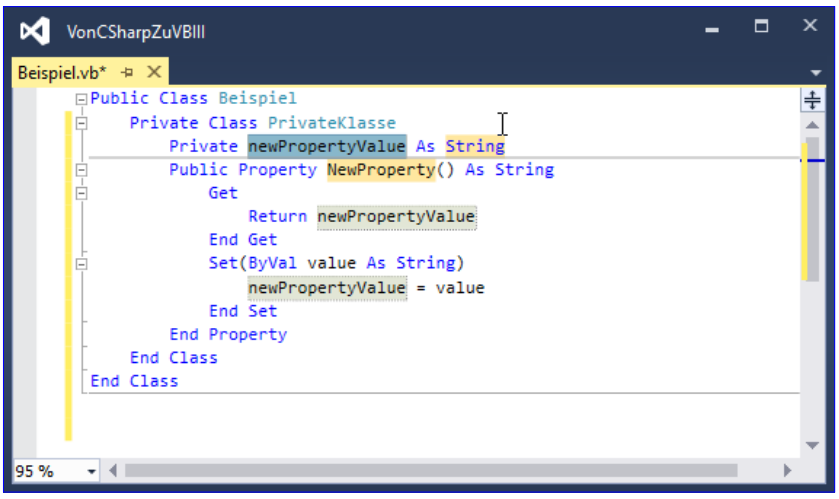


Bild 3: Vorlage für eine Property

EDM: Der Code First-Ansatz

In den bisherigen Ausgaben haben wir immer mit dem Database First-Ansatz gearbeitet, das heißt, dass wir unser Entity Data Model auf Basis einer bestehenden Datenbank im SQL Server oder SQLite generiert haben. Das geht auch andersherum: Sie erstellen ein paar Klassen, geben eine Verbindungszeichenfolge an und beim ersten Versuch, auf die Daten der Datenbank zuzugreifen, erstellt die Anwendung automatisch die Datenbank im angegebenen SQL Server. Wie das gelingt und wie die Klassen und die Verbindungszeichenfolge aussehen müssen, erfahren Sie in diesem Beitrag.

Als Beispiel wählen wir unsere Anwendung zum Erstellen von Anschreiben mit Word von einem WPF-Fenster aus. Dieser haben wir im ersten Schritt ja nur einfache Textfelder als Eingabemöglichkeit hinzugefügt, in die der Benutzer Anschrift, Datum, Betreff und Inhalt des Anschreibens eintragen kann, bevor daraus ein Word-Dokument auf Basis der gewählten Vorlage erstellt wird. Nun wollen wir dies erweitern – und zwar um die Möglichkeit, sowohl verschiedene Anschriften zu verwalten als auch um die Anschreiben abzuspeichern, um diese gegebenenfalls später noch einmal aufzurufen oder auch als Vorlage für neue Anschreiben zu verwenden.

Unter Access würden wir das Formular einfach an eine Tabelle zum Speichern der Daten des Anschreibens binden und gegebenenfalls noch eine Tabelle für die verschiedenen Anschriften hinzufügen. Allerdings wollen wir ja in diesem Magazin lernen, mit den unter WPF gängigen Techniken umzugehen – wie etwa dem Entity Data Model als Möglichkeit, Daten zwischen Benutzeroberfläche und Datenbank hin- und herzuschieben.

In den vorherigen Ausgaben haben wir meist eine vorhandene Datenbank genutzt und mit dem dafür vorgesehenen Assistenten ein Entity Data Model erstellen lassen. Dieses enthält die Klassen und Auflistungen, mit denen wir dann von der Benutzeroberfläche beziehungsweise von der Anwendungslogik aus auf die Daten der Datenbank zugreifen konnten. In diesem Beispiel wollen wir es einmal andersherum versuchen: Wir wollen zuerst die Klassen für die beiden Entitäten programmieren, also für die Anschreiben und die Adressen, und daraus dann ein Datenmodell generieren. Auch dieser Fall ist vom Entity Framework als Möglichkeit vorgesehen und nennt sich Code First. Wir erstellen also erst den Code und lassen daraus dann die Datenbank erzeugen. Das

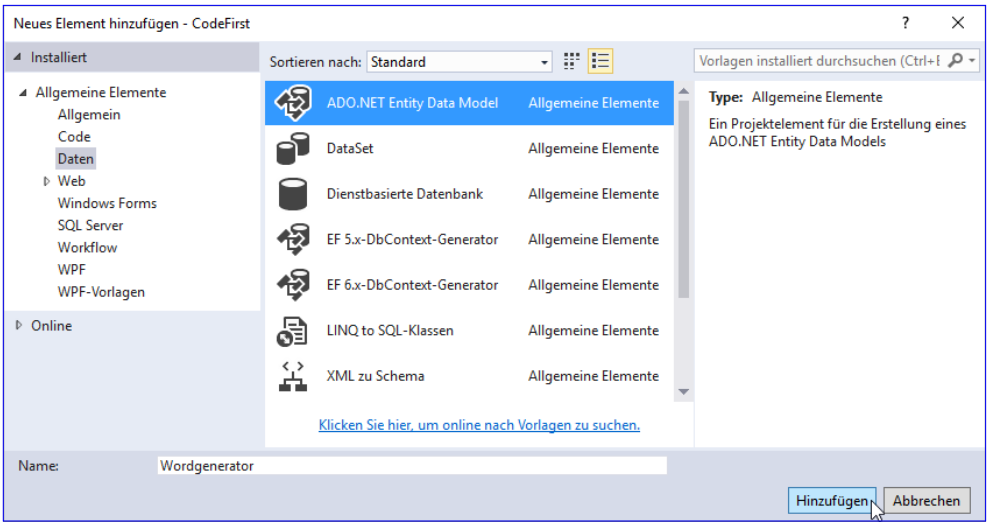


Bild 1: Entity Data Model zum Projekt hinzufügen

wir für das erste Beispiel eine relativ einfache Konstellation mit nur zwei Entitäten nutzen (okay, mit Anreden für die Adressen vielleicht drei), wird die Komplexität ein wenig vermindern.

Vorbereitungen

Als Erstes erstellen wir ein neues, leeres Projekt namens **CodeFirst** – und zwar mit der Vorlage **Visual Basic\Windows\WPF-Anwendung** oder **Visual C#\Windows\WPF-Anwendung**. Diesem Projekt fügen Sie nach Markierung des Eintrags mit dem Projektnamen **CodeFirst** im Projektmappen-Explorer ein Element des Typs **ADO.NET Entity Data Model** hinzu. Den dazu nötigen Dialog **Neues Element hinzufügen...** öffnen Sie mit der Tastenkombination **Strg + Umschalt + A**. Hier wählen Sie den passenden Elementtyp aus und legen als Name den Wert **Wordgenerator** fest (siehe Bild 1).

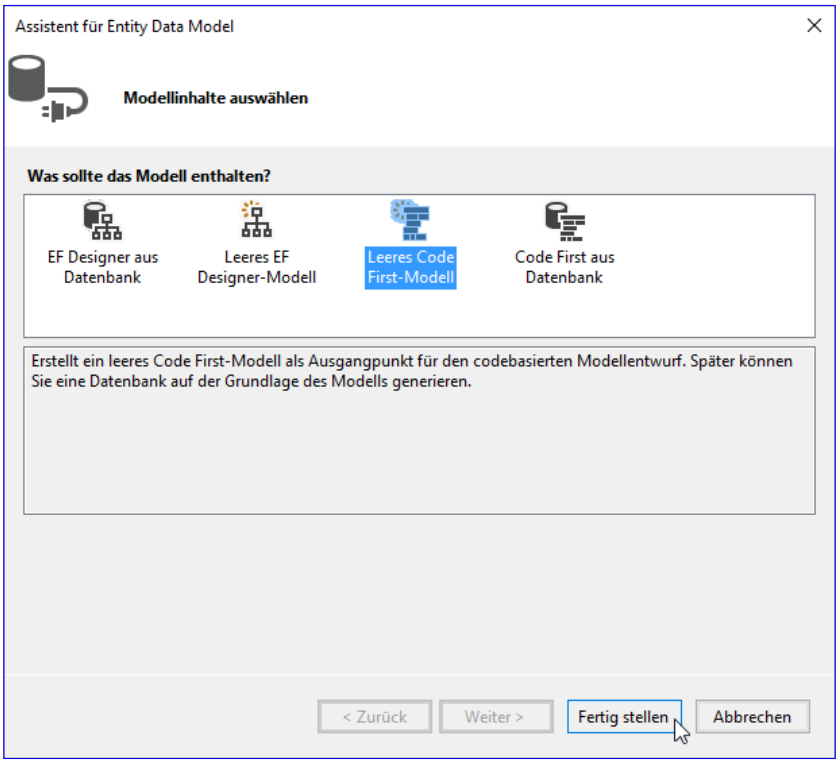


Bild 2: Modelltyp **Leeres Code First-Modell** auswählen

Im folgenden **Assistent für Entity Data Model** wählen Sie den Eintrag **Leeres Code First-Modell** aus (siehe Bild 2). Im Gegensatz zu der Variante, die wir sonst gewählt haben, nämlich **EF Designer aus Datenbank**, ist dies gleichzeitig schon der letzte Schritt in diesem Assistenten, den wir mit einem Klick auf die Schaltfläche **Fertigstellen** beenden.

Diesmal geschieht auch gar nicht viel in unserem Projekt. Die augenscheinlichste Änderung ist, dass der Assistent ein Klassenmodul namens **Wordgenerator.vb** (beziehungsweise **Wordgenerator.cs** unter C#) angelegt hat (siehe Bild 3). Die Klasse Wordgenerator erbt von der Klasse **DbContext**.

Modell programmieren

Nun programmieren wir die Klassen für unser Modell, auf dessen Basis wir später eine Datenbank erstellen lassen wollen.

Dazu legen wir drei neue Klassen an – jeweils wieder über die Tastenkombination **Strg + Umschalt + A** und den dadurch aufgerufenen Dialog **Neues Element einfügen...**. Hier wählen wir nun jeweils den Eintrag Klasse aus und legen die Namen **Anschreiben.vb**, **Adresse.vb** und **Anrede.vb** fest. Die Klasse **Anrede.vb** sieht wie folgt aus:

```
Public Class Anrede
    Public Property ID As Integer
    Public Property Anrede As String
```

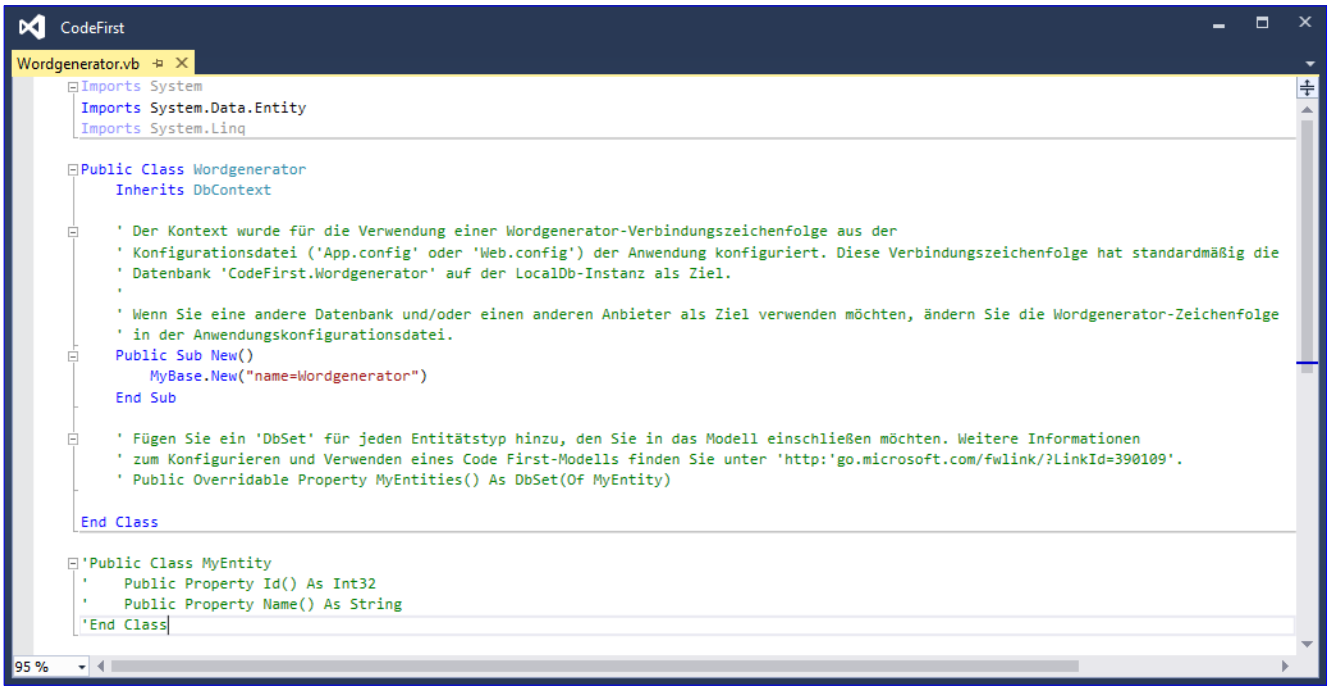


Bild 3: Durch den Assistenten angelegte Klasse

```
End Class
```

Für die Klasse Adresse legen wir den folgenden Code fest, wobei wir als Typ der Eigenschaft Anrede auch gleich die frisch angelegte Klasse gleichen Namens verwenden:

```
Public Class Adresse
    Public Property ID As Integer
    Public Property Firma As String
    Public Property Anrede As Anrede
    Public Property Vorname As String
    Public Property Nachname As String
    Public Property Strasse As String
    Public Property PLZ As String
    Public Property Ort As String
    Public Property Land As String
    Public Property EMail As String
End Class
```

Nachdem wir die Klasse **Adresse** erzeugt haben, die wir auch als Collection der Klasse **Anrede** zuordnen wollen – jede Anrede gehört ja zu keiner, einer oder mehreren Adressen –, können wir die Klasse **Anrede** noch um die Auflistung **Adressen** mit dem Typ **Adresse** erweitern:

```
Public Class Anrede
```

```
    ...
    Public Property Adressen As ICollection(Of Adresse)
End Class
```

Die Klasse **Anschreiben** erhält schließlich die folgenden Eigenschaften:

```
Public Class Anschreiben
    Public Property ID As Integer
    Public Property Adresse As Adresse
    Public Property Datum As Date
    Public Property Betreff As String
    Public Property Inhalt As String
End Class
```

Auch hier können wir nun die Klasse **Adresse** um die Auflistung der Anschreiben ergänzen:

```
Public Class Adresse
    ...
    Public Property Anschreiben As ICollection(Of Anschreiben)
End Class
```

Mit der Angabe der beiden **ICollections** in den Klassen **Anschreiben** und **Adresse** legen wir praktisch die 1:n-Beziehung zwischen den Klassen fest. Damit kann dann eine Adresse mehreren Anschreiben zugeordnet werden und eine Anrede mehreren Adressen.

Auflistungen in der DBContext-Klasse definieren

Nun gehen wir zu der eingangs erwähnten Klasse namens Wordgenerator zurück, die ja vom Assistenten automatisch angelegt wurde und die von der Klasse **DBContext** erbt. Hier legen wir nun noch Auflistungen eines speziellen Typs, nämlich **DBSet**, für die drei Entitäten **Anrede**, **Adresse** und **Anschreiben** fest. Das sieht für die ersten beiden Elemente wie folgt aus:

```
Public Property Anreden() As DbSet(Of Anrede)
Public Property Adressen() As DbSet(Of Adresse)
```

Beim Versuch, das dritte Element für die Anschreiben einzutragen, stellen wir fest, dass wir mit **Anschreiben** eine schlechte Bezeichnung gewählt habe – diese ist nämlich für Plural und Singular gleich, was eine Unterscheidung unmöglich macht. Also ändern wir den Namen der Klasse **Anschreiben** noch in **Brief**. Am einfachsten geht dies, indem Sie die Klasse schnell neu unter dem gewünschten Namen anlegen und den Inhalt in die neue Klasse kopieren:

```
Public Class Brief
    ...
End Class
```

Die Eigenschaft **Anschreiben** in der Klasse **Adresse** müssen wir dementsprechend wie folgt anpassen:

```
Public Property Briefe As ICollection(Of Brief)
```

Die alte Klasse **Anschreiben** können wir dann aus dem Projekt löschen. Außerdem tragen wir noch die fehlende Zeile für die Briefe in der Klasse **Wordgenerator** nach:

```
Public Property Briefe() As DbSet(Of Brief)
```

Damit haben wir die grundlegende Definition der Klassen abgeschlossen und schauen uns an, wie wir damit eine Datenbank erstellen können.

Verbindungszeichenfolge festlegen

Dazu benötigen wir als Erstes eine geeignete Verbindungszeichenfolge. Diese hat der Assistent vermutlich bereits für uns angelegt. Ob das der Fall ist, erfahren wir, indem wir uns die Datei **App.Config** des Projects ansehen. Diese sieht in unserem Fall wie in Bild 4 aus. Hier finden wir auch bereits einen Eintrag namens **<connectionStrings>** mit einem Untereintrag **add**. Dieser enthält die Eigenschaft connectionString mit der Verbindungszeichenfolge. Diese gibt aktuell als Server **(LocalDb)\MSSQLLocalDB** an und nennt die Datenbank **CodeFirst.Wordgenerator**. Diesen Namen ändern wir in **WordGenerator**. Die Bezeichnung der Verbindungszeichenfolge aus dem Attribut **Name** lautet **Wordgenerator** und wird auf Basis der Bezeichnung der Klasse benannt, die von **DbContext** erbt. Diese Einstellung wollen wir vorerst beibehalten. **LocalDb** ist die SQL Server-Datenbank, die mit Visual Studio installiert wird und während der Entwicklung die einfachste Möglichkeit für den Zugriff auf Datenbanken bietet. Die Verbindungszeichenfolge lautet nun:

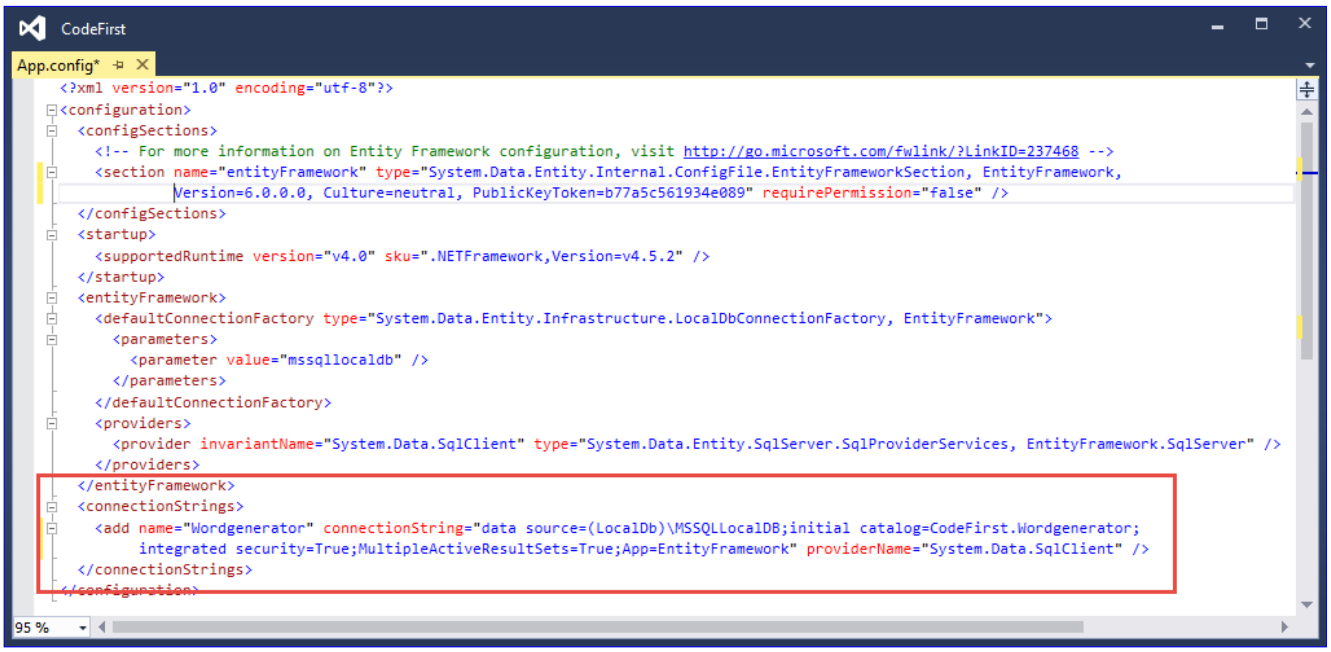


Bild 4: Die vom Assistenten automatisch angelegte Verbindungszeichenfolge

EDM: Code First - Datenbank erweitern

Im Artikel »EDM: Der Code First-Ansatz« haben wir uns angesehen, wie Sie auf Basis eines frisch erstellten Entity Data Models eine Datenbank samt den nötigen Tabellen erstellen können – und zwar automatisch beim ersten Zugriff auf die noch nicht vorhandene Datenbank. Nun gehen wir einen Schritt weiter und zeigen, wie Sie auch noch Änderungen am Entity Data Model auf bestehende Datenbanken mit älterem Versionsstand übertragen und somit ein kombiniertes Update von Anwendung und Datenbank ausliefern können.

Aktualisierungen am bestehenden Datenmodell

Wir haben während der Entwicklung der Anwendung, wir wir es im Artikel **EDM: Der Code First-Ansatz** gelernt haben, die Freiheit, nach Lust und Laune die Zieldatenbank zu löschen und neu zu erstellen. Wohlgemerkt: während der Entwicklung! Sobald die Datenbank jedoch einmal mit Daten gefüllt ist oder sogar beim Kunden läuft, wollen Sie Aktualisierungen am Datenmodell sicher etwas eleganter und ohne Datenverlust übermitteln. Auch dazu bietet Code First Möglichkeiten.

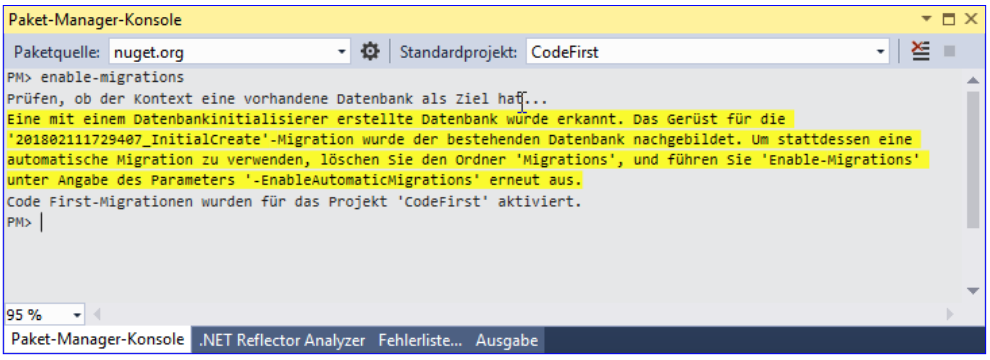


Bild 8: Aktivieren der Migrationsfunktion

Um Migrationen mit Code First durchzuführen, also die Änderungen erst am bestehenden Modell der Entitäten durchzuführen und diese dann auf die Tabellen der Datenbank zu übertragen, müssen wir zunächst die Migrationsfunktion aktivieren.

Dazu benötigen wir die Paket-Manager-Konsole, die Sie mit dem Menübefehl **Ansicht|Weitere Fenster|Paket-Manager-Konsole** einblenden. Ist dieser eingeblendet, wählen Sie oben rechts unter Standardprojekte den Namen des betroffenen Projekts aus, in diesem Fall **CodeFirst**. Dann geben Sie den folgenden Befehl ein:

```
PK> enable-migrations
```

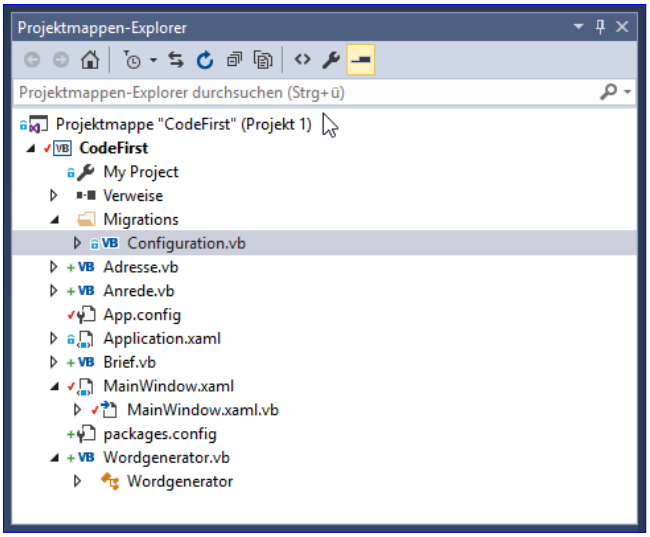


Bild 9: Aktivieren der Migrationsfunktion

Das Ergebnis finden Sie in Bild 1 vor.

Im Projekt haben sich nun einige Änderungen ergeben, die Sie direkt im Projektmappen-Explorer ablesen können. Dort finden Sie nun nämlich einen neuen Ordner namens **Migrations** (siehe Bild 2). Dieser enthält eine Datei namens **Configuration.vb**.

Diese Datei enthält vor allem die **Seed**-Methode. Dieser können Sie Anweisungen zum Schreiben von Daten in die Tabellen der Datenbank hinzufügen. Die **Seed**-Methode wird später beim Update des Datenbank-Backends ausgelöst. Deshalb fügen wir beispielsweise die folgenden Codezeilen zu dieser Methode in der Klasse **Configuration.vb** hinzu:

```
Namespace Migrations
    Friend NotInheritable Class Configuration
        Inherits DbMigrationsConfiguration(Of Wordgenerator)
        Public Sub New()
            AutomaticMigrationsEnabled = False
            ContextKey = "CodeFirst.Wordgenerator"
        End Sub
        Protected Overrides Sub Seed(context As Wordgenerator)
            Dim anrede As New Anrede With {.Bezeichnung = "Herr"}
            context.Anreden.AddOrUpdate(Function(p) p.Bezeichnung, anrede)
            anrede = New Anrede With {.Bezeichnung = "Frau"}
            context.Anreden.AddOrUpdate(Function(p) p.Bezeichnung, anrede)
            anrede = New Anrede With {.Bezeichnung = "Firma"}
            context.Anreden.AddOrUpdate(Function(p) p.Bezeichnung, anrede)
        End Sub
    End Class
End Namespace
```

Damit wollen wir sicherstellen, dass zumindest die grundlegenden Daten der Anwendung, nämlich die Anreden, bereits in die Tabelle **Anreden** geschrieben werden – hier also die Datensätze mit den Werten **Herr**, **Frau** und **Firma** im Feld **Bezeichnung**.

Anschließend geben Sie einen weiteren Befehl in den Bereich **Paket-Manager-Konsole** ein (siehe Bild 3):

```
PM> Add-Migration Initial
```

Dadurch wird eine weitere Datei zum Ordner **Migrations** hinzugefügt, diesmal nach dem Schema **201802121707380_Initial.vb**. Diese Klasse enthält eine Methode namens **Up**, die

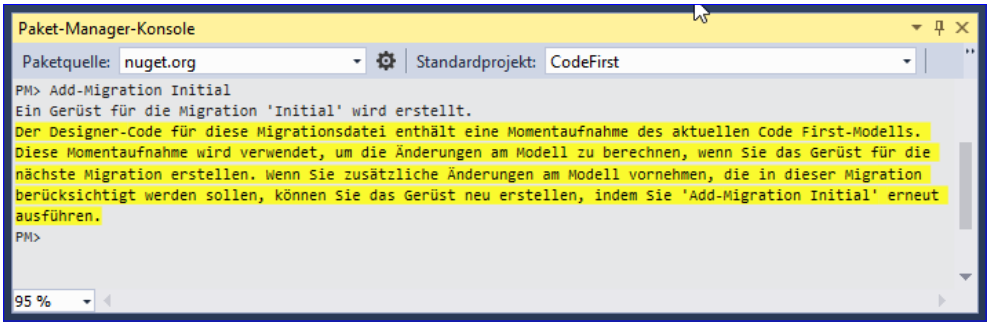


Bild 1: Initiale Migration

drei **CreateTable**-Anweisungen enthält, mit denen die Tabellen entsprechend der Entitäten **Anrede**, **Adresse** und **Brief** erstellt werden – hier in gekürzter Form nur mit den Anweisungen für die Tabelle **Anreden**:

```
Public Overrides Sub Up()
    ...
    CreateTable(
        "dbo.Anreden",
        Function(c) New With
            {
                .ID = c.Int(nullable := False, identity := True),
                .Bezeichnung = c.String()
            }) _
        .PrimaryKey(Function(t) t.ID)
    ...
End Sub
```

Zusätzlich enthält die Klasse noch eine Methode namens **Down**, welche die bestehenden Schlüssel, Indizes und Tabellen löscht:

```
Public Overrides Sub Down()
    DropForeignKey("dbo.Briefe", "Adresse_ID", "dbo.Adressen")
    DropForeignKey("dbo.Adressen", "Anrede_ID", "dbo.Anreden")
    DropIndex("dbo.Briefe", New String() { "Adresse_ID" })
    DropIndex("dbo.Adressen", New String() { "Anrede_ID" })
    DropTable("dbo.Briefe")
    DropTable("dbo.Anreden")
    DropTable("dbo.Adressen")
End Sub
```

Außerdem setzen Sie im gleichen Bereich noch den folgenden Befehl ab:

```
PM> Update-Database
```

Damit rufen Sie zunächst die Methoden der Klasse **201802121707380_Initial.vb** auf, welche die Datenbank und das Datenmodell anlegt, wenn diese noch nicht vorhanden ist, und dann mit der **Seed**-Methode die initialen Daten in die Tabelle **Anreden** schreibt.

Änderungen am Datenmodell

Nun nehmen wir eine kleine Änderung am Datenmodell vor. Wohlgemerkt: Die aktuelle Version, die durch die Datei **201802121707380_Initial.vb** beschrieben wird, haben wir durch die **Update-Database**-Methode in der **Paket-Manager-Konsole** bereits angewendet und damit eine neue Datenbank erzeugt. Nun fügen wir der Entität **Anreden** eine Eigenschaft namens **Briefanrede** hinzu:

```
<Table("Anreden")>
Public Class Anrede
    Public Property ID As Integer
    Public Property Bezeichnung As String
    Public Property Briefanrede As String
    Public Property Adressen As ICollection(Of Adresse)
End Class
```

Nun rufen wir wieder eine neue Anweisung in der **Paket-Manager-Konsole** auf:

```
add-migration add_Anrede_Briefanrede
```

Dies erstellt eine neue VB-Klasse namens **201802121730445_add_Anrede_Briefanrede.vb**, welche die Änderungen im Datenmodell seit der letzten Änderung enthält, und die wieder in Form der beiden Methoden **Up** und **Down** eingetragen werden. Das sieht dann so aus:

```
Public Partial Class add_Anrede_Briefanrede
    Inherits DbMigration
    Public Overrides Sub Up()
        AddColumn("dbo.Anreden", "Briefanrede", Function(c) c.String())
    End Sub
    Public Overrides Sub Down()
        DropColumn("dbo.Anreden", "Briefanrede")
    End Sub
End Class
```

Die **Up**-Methode wird ausgeführt, wenn die Migration angewendet werden soll, die **Down**-Methode, wenn diese rückgängig gemacht werden soll, also eine vorheriger Stand wiederhergestellt werden soll (mehr dazu weiter unten).

Damit rufen wir nun wieder den Befehl **update-database** in der **Paket-Manager-Konsole** auf, aktualisieren die im SQL Server-Objekt-Explorer angezeigten Datenbanken und navigieren wieder zu unserer Datenbank **Wordgenerator**.

Dort überzeugen uns davon, dass das neue Feld **Briefanrede** auch in der Tabelle **Anreden** vorhanden ist – was der Fall ist (siehe Bild 4).

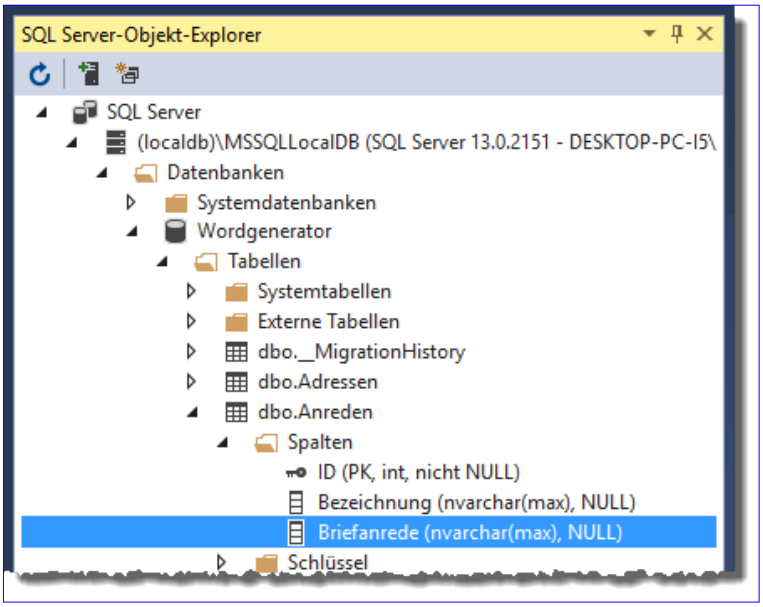


Bild 2: Das neue Feld ist in der Datenbank angekommen.

Adressen verwalten

Für unsere Lösung zum Erstellen und Verwalten von Word-Dokumenten aus dem Artikel »Briefe mit Word erstellen« benötigen wir eine Möglichkeit zum Anlegen und Verwalten von Adressen. Dazu wollen wir ein eigenes Fenster erstellen, dass eine Übersicht der Adressen in einer Liste, eine kleine Suchfunktion sowie die Details der aktuell in der Liste ausgewählten Adresse oder einer neuen Adresse anzeigt. Die Lösung soll natürlich über ein Entity Data Model an eine entsprechende Tabelle gebunden werden.

Zur Anzeige der Adressen fügen wir der Lösung ein neues Fenster namens **Adressen.xaml** hinzu. Für die Anordnung der benötigten Steuerelemente wollen wir dem Grid ein Raster von vier Spalten und acht Zeilen hinzufügen. Die erste und die dritte Spalte weisen als Spaltenbreite den Wert **Auto** auf, damit sich die Breite an den breitesten enthaltenen Steuerelementen orientiert, in diesem Fall an den Inhalten der **Label**-Elemente mit den Feldbeschriftungen. Die zweite und die vierte Spalte sollen sich den Rest der Breite aufteilen, damit der Benutzer das Fenster verbreitern und somit Inhalte der Textfelder, die gegebenenfalls nicht vollständig angezeigt werden, sichtbar machen kann.

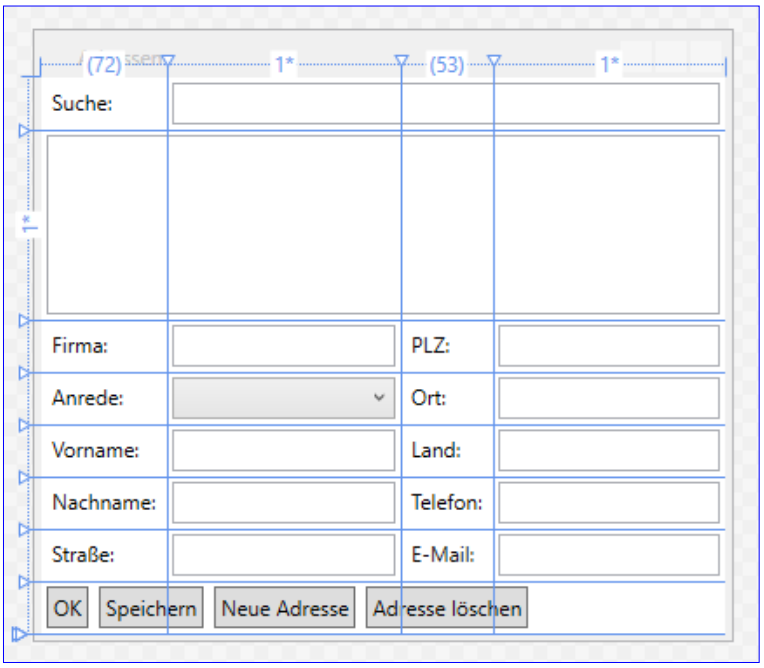


Bild 1: Fenster zum Verwalten von Adressen

Die Zeilen werden mit einer Ausnahme alle mit der Zeilenhöhe **Auto** ausgestattet, damit sich diese an der für die enthaltenen Elemente notwendigen Höhe orientieren können. Die Ausnahme ist die zweite Zeile, welche das Listenfeld zur Anzeige der Adressen in der Übersicht anzeigen soll. Diese Zeile erhält für die Eigenschaft **Height** den Wert *, was dazu führt, dass sie den verbleibenden Platz einnimmt und beim Vergrößern der Höhe des Fensters ebenfalls vergrößert wird. Die Definition des Grundgerüsts des Fensters mit XAML-Code sieht wie folgt aus und erscheint im Entwurf wie in Bild 1:

```
<Window x:Class="Adressen" ... Title="Adressen" Height="350" Width="400">
    <Window.Resources>
        <Style TargetType="TextBox">
            <Setter Property="Margin" Value="3"></Setter>
            <Setter Property="Padding" Value="3"></Setter>
        </Style>
        ...
    </Window.Resources>
```

```
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            ...
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Label>Suche:</Label>
        <Label Grid.Row="2">Firma:</Label>
        <Label Grid.Row="3">Anrede:</Label>
        ... weitere Label
        <Label Grid.Row="2" Grid.Column="2">PLZ:</Label>
        <Label Grid.Row="3" Grid.Column="2">Ort:</Label>
        ... weitere Label
        <TextBox x:Name="txtSuchbegriff" Grid.Column="1" Grid.ColumnSpan="3"></TextBox>
        <ListBox x:Name="lstAdressen" Grid.Row="1" Grid.ColumnSpan="4" Margin="3" Padding="3"></ListBox>
        <TextBox x:Name="txtFirma" Grid.Row="2" Grid.Column="1"></TextBox>
        <ComboBox x:Name="cboAnreden" Grid.Row="3" Grid.Column="1"></ComboBox>
        <TextBox x:Name="txtVorname" Grid.Row="4" Grid.Column="1"></TextBox>
        ... weitere TextBox-Elemente
        <StackPanel Orientation="Horizontal" Grid.Row="7" Grid.ColumnSpan="4">
            <Button x:Name="btnOK">OK</Button>
            <Button x:Name="btnSpeichern">Speichern</Button>
            <Button x:Name="btnNeu">Neue Adresse</Button>
            <Button x:Name="btnLoeschen">Adresse löschen</Button>
        </StackPanel>
    </Grid>
</Window>
```

Im unteren Bereich findet sich noch ein **StackPanel**, welches ebenfalls über die gesamten vier Spalten geht und in horizontaler Ausrichtung vier **Button**-Elemente anzeigt. Für die Steuerelementtypen **Label, TextBox**, **ListBox**, **ComboBox** und **Button** haben wir im oberen Bereich unter **Window.Resources** einige allgemeine Eigenschaften wie **Margin** und **Padding** festgelegt. Diese greifen bei den Steuerelementen, die per **ColumnSpan** über mehrere Spalten angelegt werden, nicht – daher müssen wir zum Beispiel die Eigenschaft **Margin** für das Textfeld zur Eingabe des Suchbegriffs und das Listenfeld direkt mit dem Steuerelement definieren.

Entity Data Model

Das Entity Data Model für die Lösung haben wir bereits im Beitrag **EDM: Die Code First-Methode** und **EDM: Code First - Datenbank erweitern** besprochen. Der Teil, der für die hier zu verwaltenden Adressen verantwortlich ist, sind die beiden Klassen **Adresse** und **Anrede**. Wir benötigen für das Listenfeld eine Auflistung der Adressen mit den wichtigsten Feldern, für das Kombinationsfeld **cboAnreden** eine Auflistung der Anreden und für die Details der aktuell bearbeiteten Adresse ein Objekt auf Basis der Klasse **Adresse**. Um die Steuerelemente des Fensters an die Daten der beiden Klassen zu binden, fügen wir zunächst eine öffentliche Eigenschaft für ein Objekt auf Basis der Klasse Adresse zur Code behind-Klasse hinzu:

```
Public Property AktuelleAdresse As Adresse
```

Außerdem benötigen wir eine Variable zum Speichern der Objektvariablen für das **DbContext**-Element:

```
Private WordbriefContext As WordbriefEntities
```

Für die Liste der Adressen im **ListView**-Steuerelement sehen wir die folgende Variable vor:

```
Public Property Adressenliste As ObservableCollection(Of Adresse)
```

Fehlt noch eine Collection für die Anreden, die wir so deklarieren:

```
Public Property Anredenliste As ObservableCollection(Of Anrede)
```

Dann erstellen wir das Grundgerüst für die Konstruktor-Methode, also die Methode, die beim Öffnen des Fensters ausgelöst wird. Nun können wir das Fenster auf zwei Arten öffnen: Erstens mit der Schaltfläche zum Bearbeiten des aktuell im Hauptfenster **MainWindow.xaml** angezeigten Datensatzes und zweitens mit der Schaltfläche zum Erstellen eines neuen Adress-Objekts. Das heißt also, dass wir entweder eine ID aus dem aufrufenden Fenster übergeben müssen oder, wenn ein neues Element angelegt werden soll, eben nicht. Das ist eine gute Gelegenheit, einmal das Überladen der Konstruktor-Methode zu demonstrieren. Das bedeutet, dass wir zwei verschiedene Konstruktor-Methoden erstellen, von denen die eine keinen Parameter enthält – das ist die zum Anlegen eines neuen Elements. Die andere soll einen bestehenden Adress-Datensatz anzeigen, also übergeben wir dieser Konstruktor-Methode die ID des anzuzeigenden Datensatzes.

Dazu legen wir einen entsprechenden Parameter an. Grundsätzlich sehen die beiden Methoden also wie folgt aus:

```
Public Sub New()
    InitializeComponent()
    ...
End Sub
Public Sub New(AdresseID As Integer)
    InitializeComponent()
    ...
End Sub
```

Sie brauchen übrigens immer nur den Kopf mit **Public Sub New(...)** einzugeben, Visual Studio erzeugt dann immer den Rest mit der Anweisung **InitializeComponent** und den üblichen Kommentarzeilen – das gilt auch, wenn Sie Überladungen der **New**-Methode angeben.

Konstruktor-Methoden füllen

Damit begeben wir uns an die Konstruktor-Methoden. Die Methode **New** ohne Parameter soll dafür sorgen, dass das Fenster **AdressenVerwalten.xaml** einen neuen, leeren Datensatz anzeigt, den der Benutzer dann mit Daten füllen und speichern kann. Dazu benötigen wir die folgende Konstruktor-Methode:

```
Public Sub New()
    InitializeComponent()
    WordbriefContext = New WordbriefEntities
    Adressenliste = New ObservableCollection(Of Adresse)(WordbriefContext.Adressen)
    AktuelleAdresse = New Adresse
    Anredenliste = New ObservableCollection(Of Anrede)(WordbriefContext.Anreden)
    DataContext = Me
End Sub
```

Die Methode füllt **WordbriefContext** mit dem Entity Data Model. Außerdem liest sie die Adressen und Anreden in die Collections **Adressen** und **Anreden** ein. Die Variable **Adresse** wird einem neuen Element des Typs **Adresse** gefüllt. Die zweite Konstruktor-Methode erwartet den Parameter **SelektierteAdresse** des Typs **Adresse**, der vom aufrufenden Fenster übergeben wird und in den Steuerelementen des Fensters **AdressenVerwalten** als aktive Adresse angezeigt werden soll:

```
Public Sub New(selektierteAdresse As Adresse)
    InitializeComponent()
    WordbriefContext = New WordbriefEntities
    Adressenliste = New ObservableCollection(Of Adresse)(WordbriefContext.Adressen)
    Anredenliste = New ObservableCollection(Of Anrede)(WordbriefContext.Anreden)
    If Not selektierteAdresse Is Nothing Then
        AktuelleAdresse = selektierteAdresse
        lvwAdressen.SelectedValue = AktuelleAdresse.ID
    End If
    DataContext = Me
End Sub
```

Diese Methode unterscheidet sich nicht nur durch den Parameter von der ersten Konstruktor-Methode, sondern auch durch eine zusätzliche **If...Then**-Bedingung.

Diese prüft, ob der Parameter **selektierteAdresse** einen Wert enthält. Falls ja, wird die Eigenschaft **AktuelleAdresse** auf diese Adresse eingestellt. Außerdem soll das **ListView**-Steuerelement, dass ja auch mit der Liste der **Adress**-Objekte gefüllt wird, das im Parameter übergebene Element selektieren.

```
Private Sub btnOK_Click(sender As Object, e As RoutedEventArgs)
    If bolAdresseAusgewaehlt Then
        RaiseEvent AdresseAusgewaehlt(Me, New AdresseEventArgs(AktuelleAdresse))
    End If
    Me.Close()
End Sub
```

Die Ereignisse des Fensters

Das Fenster definiert drei Ereignisse, die von den verschiedenen Methoden ausgelöst werden. Diese deklarieren wir wie folgt:

```
Public Event AdresseHinzugefuegt(ByVal sender As Object, ByVal e As AdresseEventArgs)
Public Event AdresseGeaendert(ByVal sender As Object, ByVal e As AdresseEventArgs)
Public Event AdresseAusgewaehlt(ByVal sender As Object, ByVal e As AdresseEventArgs)
```

Die hier als Parameter verwendete Klasse **AdresseEventArgs** definieren wir in einer eigenen Datei wie folgt:

```
Public Class AdresseEventArgs
    Inherits EventArgs
    Private m_Adresse As Adresse
    Public ReadOnly Property Adresse As Adresse
        Get
            Return m_Adresse
        End Get
    End Property
    Public Sub New(Adresse As Adresse)
        m_Adresse = Adresse
    End Sub
End Class
```

Wie wir diese Ereignisse in der aufrufenden Klasse implemetieren, schauen wir uns im Artikel **Brief mit Word erstellen, Teil II** an.

Zusammenfassung und Ausblick

Das in diesem Artikel vorgestellte Fenster soll die Verwaltung von Adressen erlauben. Diese wollen wir in unserer Lösung aus dem Artikel **Briefe mit Word erstellen** nutzen, die wir im Artikel **Briefe mit Word erstellen, Teil II** in der Ausgabe 2/2018 erweitern. In diesem Artikel beschreiben wir auch, wie Sie das Fenster **AdressenVerwalten.xaml** vom Fenster **MainWindow.xaml** des Beispielprojekts aus öffnen. Im Beispielprojekt können Sie sich dies vorab schon ansehen.