AUSGABE 02/2020

# DATENBANK ENTWICKLER

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

## TOP-THEMEN:



André Minhorst Verlag

datenbankentwickler.net

# Das Expander-Steuerelement

**Manchmal wird es eng in einem Fenster mit vielen oder großen Steuerelementen. Oder Sie haben Steuerelemente, die nicht unbedingt immer sichtbar sein müssen. Dann ist das Expander-Steuerelement genau das Richtige: Mit diesem können Sie Bereiche im Fenster definieren, die bei Bedarf ausgeklappt werden können. Dieser Artikel zeigt die grundlegenden Techniken des Expander-Steuerelements.**

Mit dem Kombinationsfeld haben Sie schon ein Steuerelement kennengelernt, mit dem Sie Platz in Ihren Fenstern sparen können – dort zeigen wir die zur Verfügung stehenden Einträge in einem Auswahlfeld an, das nach der Auswahl wieder verschwindet.

Das **Expander**-Steuerelement arbeitet ganz ähnlich. Hier definieren Sie allerdings weitere Steuerelemente, die durch Aufklappen des **Expander**-Steuerelements angezeigt werden können. Das **Expander**-Steuerelement kann allerdings nur ein einziges untergeordnetes Steuerelement aufnehmen.

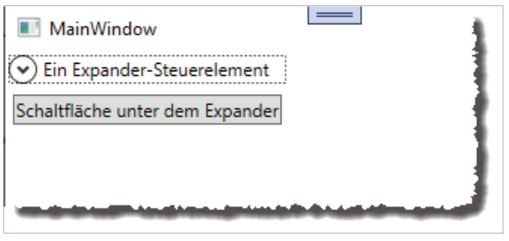


**Bild 1:** Eingeklapptes **Expander**-Steuerelement

Wenn Sie dennoch mehrere Steuerelemente anzeigen wollen, verwenden Sie eines der verfügbaren Container-Steuerelemente wie beispielsweise das **StackPanel**-Steuerelement. Darin können Sie dann die gewünschten Steuerelemente anlegen. Sie können aber auch ein **Grid**-Steuerelement als Inhalt des **Expander**-Steuerelements angeben.

Ein **Expander**-Steuerelement sieht im eingeklappten Zustand wie in Bild 1 aus. Es zeigt dann nur eine Schaltfläche mit einem Pfeil nach unten an sowie einen Text. Dieser liefert optimalerweise einen Hinweis darauf, welche Steuerelemente sich dahinter verbergen.

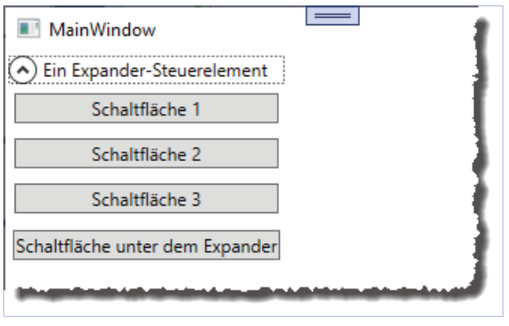


**Bild 2:** Ausgeklapptes **Expander**-Steuerelement

Nach dem Anklicken des Pfeils nach unten werden die enthaltenen Steuerelemente angezeigt. Die darunter befindlichen Steuerlemente werden dann nach unten verschoben (siehe Bild 2). Außerdem ändert sich das Symbol der Schaltfläche in einen nach oben zeigenden Pfeil.

Der Code hinter diesem **Expander**-Steuerelement sieht wie folgt aus:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Grid>
        <!--Grid-Definitionen-->
        <Expander Header="Ein Expander-Steuerelement">
            <StackPanel>
```

```
            <Button x:Name="btn1">Schaltfläche 1</Button>
            <Button x:Name="btn2">Schaltfläche 2</Button>
            <Button x:Name="btn3">Schaltfläche 3</Button>
        </StackPanel>
    </Expander>
    <Button Grid.Row="1">Schaltfläche unter dem Expander</Button>
</Grid>
</Window>
```

## Header mit Text oder anderen Inhalten

Statt des hier verwendeten Textes für die Anzeige im Header können Sie auch beliebige anderen Elemente dort anzeigen.

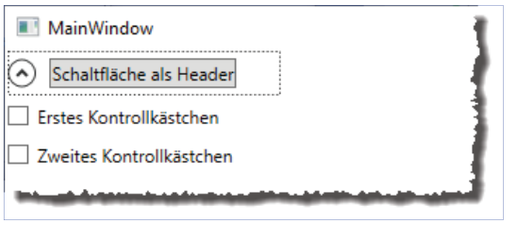


**Bild 3: Expander**-Element mit Schaltfläche als Header

Sie können dort beispielsweise eine Schaltfläche unterbringen:

```
<Expander Grid.Row="2">
    <Expander.Header>
        <Button>Schaltfläche als Header</Button>
    </Expander.Header>
    <StackPanel Orientation="Vertical">
        <StackPanel Orientation="Horizontal">
            <CheckBox x:Name="chk1"></CheckBox>
            <Label>Erstes Kontrollkästchen</Label>
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <CheckBox x:Name="chk2"></CheckBox>
            <Label>Zweites Kontrollkästchen</Label>
        </StackPanel>
    </StackPanel>
</Expander>
```

Das Ergebnis finden Sie in Bild 3.

## Ausklapp-Richtung festlegen

Mit dem Attribut **ExpandDirection** legen Sie fest, in welche Richtung die enthaltenen Inhalte ausgeklappt werden sollen. Der folgende Expander wird nach oben ausgeklappt:

```
<Expander Grid.Row="3" Header="Ein Expander-Steue-
relement" ExpandDirection="Up">
```

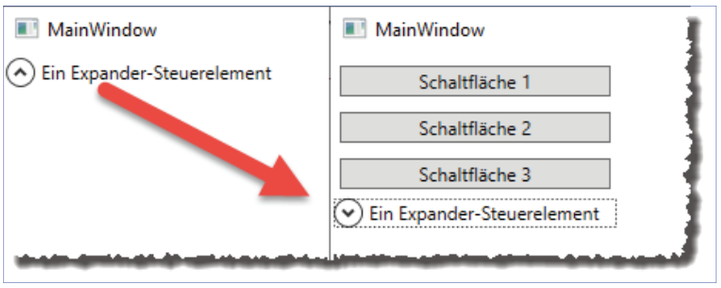


**Bild 4: Expander**-Inhalt nach oben ausklappen

# Das TextBox-Steuerelement

**Das wohl am meisten verwendete Steuerelement ist das TextBox-Steuerelement. Es dient zur Eingabe von Texten und bietet eine Menge von Eigenschaften, Ereignissen und Methoden, um damit zu arbeiten. Sie können ungebundene Textfelder nutzen, um Text einzugeben, der auf bestimmte Weise verarbeitet werden soll oder Textfelder über das Entity Data Model an die Datenquelle binden, um Datenbankinhalte anzuzeigen und zu bearbeiten. Dieser Artikel liefert die Grundlagen zum TextBox-Steuerelement unter WPF.**

Ein Textfeld ist unter WPF schnell erstellt. Sie fügen einfach das öffnende und das schließende **TextBox**-Element in das Grid-Element ein und fügen dazwischen einen Text hinzu:

```
<Grid>
    <TextBox>Dies ist ein Beispieltext.</TextBox>
</Grid>
```

Das Textfeld nimmt dann allerdings den gesamten Platz im Grid ein, sodass wir noch ein paar Anpassungen vornehmen. Entweder wir legen die Position und die Größe absolut mit den Eigenschaften **Width**, **Height**, **Top** und **Left** fest oder wir fügen zum Grid einige Zeilen und Spalten hinzu, damit wir gleich mehrere Steuerelemente gleichmäßig anordnen können. Wir wählen letztere Alternative, die Sie im Quellcode ansehen können und die im Entwurf wie in Bild 1 aussieht. Hier haben wir noch ein **Label**-Element in die erste Spalte der ersten Zeile eingefügt und das **TextBox**-Element in die zweite Spalte der ersten Zeile:

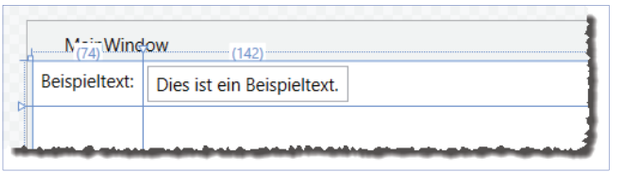


**Bild 1:** Textfeld mit Bezeichnungsfeld

```
<Label>Beispieltext:</Label>
<TextBox Grid.Column="1">Dies ist ein Beispieltext.</TextBox>
```

## Formatierter Text?

Unterschiedliche Textformatierungen lassen sich mit dem **TextBox**-Steuerelement nicht anwenden oder anzeigen. Dazu benötigen wir beispielsweise das **RichTextBox**-Steuerelement.

## Padding: Abstand des Inhalts zum Textfeldrand

Neben den Standardeigenschaften **Width**, **Height**, **Top** und **Left** benötigen Sie vermutlich noch die Eigenschaften **Margin** und **Padding**. Mit **Padding** stellen Sie den Abstand des Inhalts, also in der Regel des Textes, von den Begrenzungen des Steuerelements ein.

Soll der Abstand oben, unten, rechts und links gleich groß sein, legen Sie es zum Beispiel mit **Padding="3"** fest. Wenn Sie unterschiedliche Abstände wünschen, geben Sie die Werte durch Kommata getrennt an, wobei die Reihenfolge links, oben, rechts und unten lautet.

### Margin: Abstand zu den umgebenden Steuerelementen

Mit der Eigenschaft **Margin** hingegen geben Sie den Abstand an, den das Steuerelement zu den umgebenden Steuerelementen aufweisen soll. Wenn Sie also etwa ein Grid verwenden, um die Steuerelemente anzuordnen, und keinen Wert für **Margin** angeben, gibt es keinen Abstand zwischen den Steuerelementen. Um einen solchen Abstand herzustellen, legen Sie also etwa **Margin="3"** fest – so hat das Steuerelement jeweils einen Abstand von **3** zum nächsten angrenzenden Element, in diesem Fall zu den gedachten Linien des **Grid**-Elements.

### Textfeld ausrichten

Für Textfelder gibt es die üblichen Möglichkeiten zur Ausrichtung. Für die horizontale Ausrichtung im Grid verwenden Sie die Eigenschaft **HorizontalAlignment** mit den Werten **Center**, **Left**, **Right** und **Stretch**. Für die vertikale Ausrichtung nutzen Sie die Eigenschaft **VerticalAlignment** mit den Werten **Bottom**, **Center**, **Stretch** und **Top**.

### Text ausrichten

Natürlich können Sie auch den Text im Textfeld ausrichten. Dazu verwenden Sie die Eigenschaft **TextAlignment** mit den Werten **Center**, **Justify**, **Left** (Standardwert) und **Right**:

```
<TextBox Text="Mittig zentrierter Text" TextAlignment="Center"></TextBox>
```

### Inhalt des Steuerelements

Den Inhalt des Steuerelements können Sie, wie oben gesehen, einfach zwischen das öffnende und schließende **TextBox**-Element einfügen. Dies entspricht der Verwendung der Eigenschaft **Text**:

```
<TextBox Text="Dies ist ein Beispieltext."></TextBox>
```

### Eingebautes Kontextmenü

Das **TextBox**-Steuerelement liefert standardmäßig ein Kontextmenü mit den drei Befehlen **Ausschneiden**, **Kopieren** und **Einfügen** mit, das Sie wie üblich mit einem Klick der rechten Maustaste auf das Steuerelement anzeigen (siehe Bild 2). Dieses Kontextmenü können Sie anpassen. Wie das geht, erklären wir in einem weiteren Artikel namens **Kontextmenüs unter WPF** (**www.datenbankentwickler.net/214**).

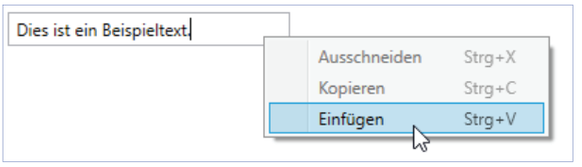


**Bild 2:** Kontextmenü eines Textfeldes

Weitere Einträge kommen zum Kontextmenü hinzu, wenn die Rechtschreibprüfung aktiviert ist – siehe nächster Abschnitt.

### Rechtschreibprüfung

Standardmäßig ist keine Rechtschreibprüfung für **TextBox**-Elemente aktiviert. Sie können diese jedoch leicht anschalten, indem Sie die Eigenschaft **SpellCheck.IsEnabled** auf den Wert **True** einstellen. Wenn Sie dann ein falsch geschriebenes

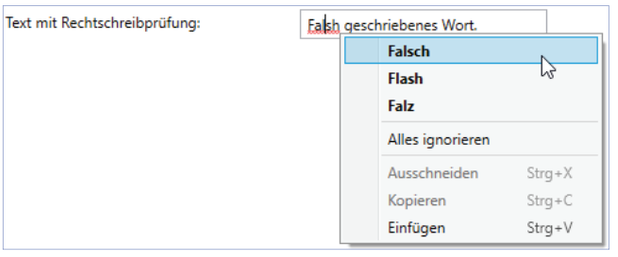


**Bild 3:** Rechtschreibprüfung im Textfeld mit Verbesserungsvorschlägen

Wort einfügen, wird dieses rot unterstrichen. Klicken Sie dann mit der rechten Maustaste auf das Wort, zeigt das Kontextmenü Verbesserungsvorschläge an (siehe Bild 3).

### Eingabe kontrollieren

Wenn Sie sicherstellen wollen, dass nicht mehr als eine bestimmte Anzahl Zeichen in einem Textfeld landen, können Sie die Anzahl mit der Eigenschaft **MaxLength** begrenzen. Geben Sie einfach die maximale Anzahl als Wert an:

```
<TextBox MaxLength="50"></TextBox>
```

### Textumbruch aktivieren

Normalerweise wird Text, der die Breite des Textfeldes übersteigt, einfach nicht komplett angezeigt (Voraussetzung dafür ist, dass sich das Textfeld nicht automatisch verbreitern kann – also wenn entweder eine feste Breite festgelegt oder die Breite des Fensters begrenzt ist). Das gilt auch für Textfelder, deren Höhe für die Darstellung von mehr als einer Zeile ausreicht. Wenn sie möchten, dass der Text am Ende des Textfeldes umgebrochen wird, können Sie die Eigenschaft **TextWrapping** verwenden. Diese bietet drei mögliche Werte an:

- **NoWrap**: Standardwert, hier erfolgt kein Zeilenumbruch.
- **Wrap**: Der Text wird am Ende des Textfeldes umgebrochen. Wenn es keine Möglichkeit durch ein Leerzeichen oder ähnliches Zeichen gibt, den Text am Wortende zu umbrechen, dann wird der Umbruch mitten im Wort eingefügt.
- **WrapWithOverflow**: Der Text wird am Ende des Textfeldes umgebrochen. Wenn es keine Möglichkeit durch ein Leerzeichen oder ähnliches Zeichen gibt, den Text am Wortende zu umbrechen, dann erfolgt für dieses Wort kein Umbruch, sondern es wird abgeschnitten angezeigt.

```
<Label Grid.Row="1">Text mit TextWrapping=Wrap:</Label>
<TextBox Grid.Row="1" Grid.Column="1" Height="80" Width="200" TextWrapping="Wrap">
    Dies ist ein Beispieltext mit automatischem Zeilenumbruch undeinemsehrsehrsehrsehrlangenWort.</TextBox>
<Label Grid.Row="2">Text mit TextWrapping=WrapWithOverflow:</Label>
<TextBox Grid.Row="2" Grid.Column="1" Height="80" Width="200" TextWrapping="WrapWithOverflow">
    Dies ist ein Beispieltext mit automatischem Zeilenumbruch undeinemsehrsehrsehrsehrlangenWort.</TextBox>
```

Beispiele sehen Sie in Bild 4.

### Zeilenumbrüche mit der Eingabetaste einfügen

Wenn Sie längere Texte im Textfeld eingeben möchten, wollen Sie gegebenenfalls auch Zeilenumbrüche nutzen, die Sie selbst per Eingabetaste festlegen. Auch dafür gibt es eine Eigenschaft. Sie heißt **AcceptsReturn** und erwartet einen der Werte **True** oder **False** (Standardwert). Die folgende Textbox verarbeitet das Betätigen der Eingabetaste als Zeilenumbruch:

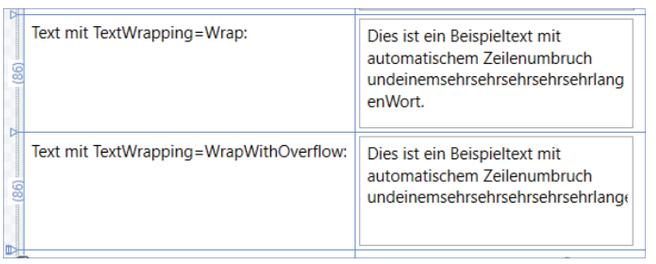


**Bild 4:** Beispiele für **TextWrapping**-Einstellungen

```
<TextBox Height="80" Width="200" TextWrapping="WrapWithOverflow" AcceptsReturn="True"></TextBox>
```

Damit können Sie beispielsweise Texte wie in Bild 5 erzeugen.

### Tabulator aktivieren

Auf ähnliche Weise können Sie die Eingabe von Tabulator-Tastenanschlägen beeinflussen. Normalerweise führt das Betätigen der Tabulator-Taste innerhalb eines Textfeldes dazu, dass der Fokus auf das nächste Steuerelement verschoben wird. Wenn Sie jedoch Tabulator-Zeichen in den Text eingeben möchten, müssen Sie dies aktivieren. Dazu stellen Sie die Eigenschaft **AcceptsTab** auf den Wert **True** ein:

```
<TextBox AcceptsTab="True">
```

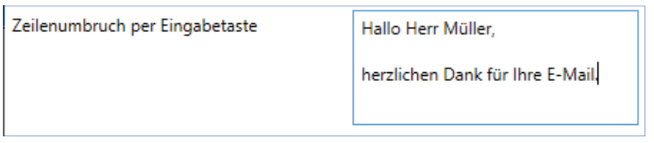


**Bild 5:** Beispiel für die Eingabe von Zeilenumbrüchen per Eingabetaste

### Bildlaufleisten aktivieren

Wenn Sie in einem **TextBox**-Element mehrzeilige Texte anzeigen wollen, kann es sein, dass die Höhe des Textfeldes nicht für die Anzeige des Inhalts ausreicht. In diesem Fall aktivieren Sie die Anzeige der vertikalen Bildlaufleiste, und zwar mit der Eigenschaft **VerticalScrollBarVisibility**:

```
<TextBox Height="80" Width="200" TextWrapping="Wrap" AcceptsReturn="True" VerticalScrollBarVisibility="Auto">Dies ist
ein Beispieltext mit automatischem Zeilenumbruch, der zu lang ist, um komplett angezeigt zu werden. Daher haben wir für
dieses Textfeld die vertikale Bildlaufleiste aktiviert.</TextBox>
```

Die Eigenschaft bietet die folgenden Werte an:

- **Auto**: Die Bildlaufleiste wird eingeblendet, sobald der Text nicht mehr komplett in das **TextBox**-Element passt.
- **Disabled**: Die Bildlaufleiste wird nicht angezeigt und Sie können auch nicht nach unten scrollen, auch wenn der Text so lang ist, dass ein Teil nicht mehr angezeigt werden kann.
- **Hidden**: Die Bildlaufleiste wird nicht angezeigt (Standardeinstellung). Aber Sie können dennoch durch den Text bis nach unten scrollen.
- **Visible**: Die Bildlaufleiste wird immer angezeigt, auch wenn die Höhe des Textfeldes noch nicht die Höhe des Textfeldes überschreitet.

Bild 6 zeigt ein **TextBox**-Element mit Bildlaufleiste.

### Textfeld für die Bearbeitung sperren

Wenn Sie nicht möchten, dass ein Benutzer den Inhalt eines Textfeldes bearbeiten kann (zum Beispiel die ID einer Entität), dann können Sie die Bearbeitung durch Einstellen der Eigenschaft **IsEnabled** auf den Wert **False** sperren:

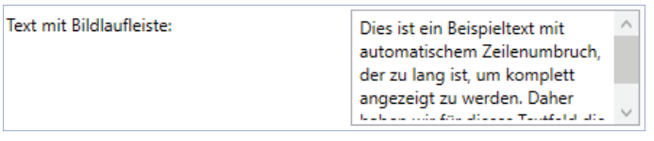


**Bild 6:** TextBox mit Bildlaufleiste

# Kontextmenüs mit WPF

**Wer einmal Kontextmenüs unter Access programmiert hat, weiß, wie viel Arbeit das ist. Unter WPF könnte das anders sein. Ob das der Fall ist und wie wir dort Kontextmenüs programmieren, zeigt dieser Artikel. Dabei beleuchten wir, wie Sie ein Kontextmenü für ein bestimmtes Element erzeugen, wie Sie Befehle hinzufügen, diese mit Icons ausstatten, welche Steuerelemente Sie darin unterbringen können und wie Sie die Einträge aktivieren und deaktivieren können. Außerdem erfahren Sie natürlich, wo Sie den Code unterbringen, der durch die Befehle von Kontextmenüs aufgerufen wird.**

## Das ContextMenu-Element

Kontextmenüs bilden wir unter WPF mit dem **ContextMenu**-Element ab. Dieses können Sie zu beliebigen Elementen hinzufügen – also zum Beispiel zu einem Fenster, einer Seite oder einem dort enthaltenen Steuerelement.

Im einfachsten Fall weisen wir direkt einem **Grid**-Element ein Kontextmenü zu. Der WPF-Code für das **Grid**-Element sieht dann wie folgt aus:

```
<Grid>
    <Grid.ContextMenu>
        <ContextMenu>
            <MenuItem Header="Meldung anzeigen" Click="MenuItem_Click"/>
        </ContextMenu>
    </Grid.ContextMenu>
</Grid>
```

Wer denkt, dass nun beim Rechtsklick auf das **Grid**-Element ein Kontextmenü erscheint, irrt sich: Es geschieht nichts. Und wir haben sichergestellt, dass das Grid sich über das gesamte übergeordnete **Window**-Element erstreckt. Der Clou ist: Jedes Element, das ein Kontextmenü anzeigen soll, muss einen Hintergrund besitzen.

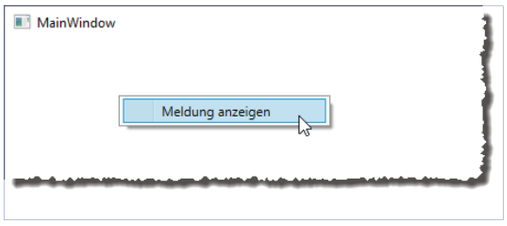


**Bild 1:** Das erste Kontextmenü, hier für ein **Grid**-Element

Das **Grid**-Element ist schlicht ein Rahmen ohne Inhalt, also können Sie darin auch kein Kontextmenü anzeigen. Das lässt sich allerdings leicht ändern, indem wir dem **Grid**-Element einen Hintergrund zuweisen. Und der darf interessanterweise auch transparent sein. Wir ändern das **Grid**-Element also wie folgt:

```
<Grid Background="Transparent">
```

Danach erscheint beim Rechtsklick das Kontextmenü wie in Bild 1. Zusammengefasst: Wir benötigen ein Property-Element (also eine Eigenschaft, die nicht als Attribut angegeben wird, sondern als untergeordnetes Element) in der Form **<Element-**

**typ>.ContextMenu**, darin ein **ContextMenu**-Element und darin ein oder mehrere **MenuItem**-Elemente. Mit der Eigenschaft **Header** legen wir den Text des Menüeintrags fest, mit der **Click**-Eigenschaft die Ereignismethode, die beim Anklicken des Eintrags aufgerufen werden soll.

In unserem Fall wollen wir einfach eine Meldung anzeigen und gestalten die Methode wie folgt:

```
Private Sub MenuItem_Click(sender As Object, e As RoutedEventArgs)
    MessageBox.Show("Meldung per Kontextmenü")
End Sub
```

## Mehrere Kontextmenü-Einträge

Wollen Sie mehrere Kontextmenü-Einträge anzeigen, fügen Sie einfach die gewünschten Elemente unter dem bereits vorhandenen **MenuItem** ein:

```
<ContextMenu>
    <MenuItem Header="Meldung anzeigen" Click="MenuItem_Click"/>
    <MenuItem Header="Noch ein Eintrag" Click="MenuItem_Click"/>
    <MenuItem Header="Letzter Eintrag" Click="MenuItem_Click"/>
</ContextMenu>
```

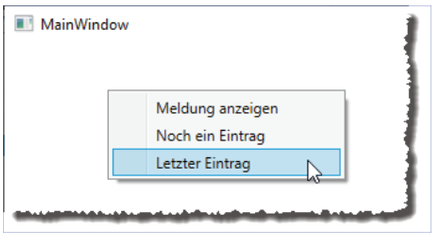


**Bild 2:** Kontextmenü mit drei Einträgen

Bild 2 zeigt das Ergebnis.

## Untermenüs

Wenn Sie Kontextmenü-Einträge unterhalb eines der Einträge benötigen, fügen Sie einfach weitere **MenuItem**-Elemente unterhalb eines der **MenuItem**-Elemente der ersten Ebene ein. Das Ergebnis finden Sie in Bild 3.

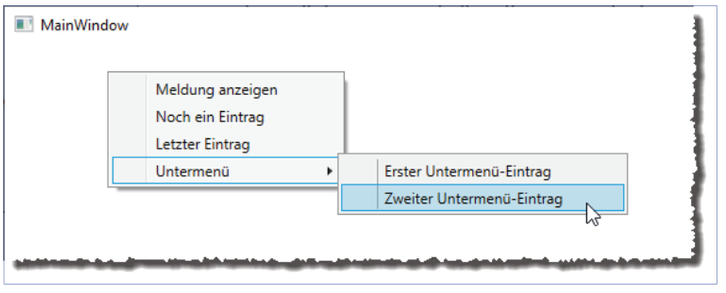


**Bild 3:** Kontextmenü mit Untermenü-Einträgen

Dem **MenuItem**-Element mit den untergeordneten Elementen können Sie zwar das **Click**-Attribut zuordnen, das Ereignis wird jedoch nicht ausgelöst – der Mausklick bewirkt lediglich das Anzeigen der untergeordneten Elemente:

```
<ContextMenu>
    <MenuItem Header="Meldung anzeigen" Click="MenuItem_Click"/>
    <MenuItem Header="Noch ein Eintrag" Click="MenuItem_Click"/>
    <MenuItem Header="Letzter Eintrag" Click="MenuItem_Click"/>
    <MenuItem Header="Untermenü">
        <MenuItem Header="Erster Untermenü-Eintrag"></MenuItem>
        <MenuItem Header="Zweiter Untermenü-Eintrag"></MenuItem>
    </MenuItem>
</ContextMenu>
```

## Click-Ereignisse anlegen

Wenn Sie mehrere Kontextmenü-Einträge in einem Kontextmenü anlegen, haben Sie zwei Möglichkeiten für das Anlegen der dadurch ausgelösten **Click**-Ereignisse: Entweder Sie vergeben jedem Kontextmenü-Eintrag einen eigenen Namen, so wie wir es sonst auch machen. Dann legen sie für jedes Kontextmenü-Element eine eigene Ereignismethode an. Den Namen vergeben Sie wie üblich bei Elementen unter WPF – wir verwenden das Präfix **itm**:

```
<MenuItem x:Name="itmMeldung" Header="Meldung anzeigen" Click="ItmMeldung_Click"/>
```

Die Ereignismethode erhält dementsprechend den Namen **ItmMeldung_Click**. Wenn Sie keine Namen für die Kontextmenü-Einträge vergeben wollen, müssen Sie zumindest Werte mit unterschiedlichen Namen für das **Click**-Attribut festlegen und entsprechend je ein Ereignis für jeden Kontextmenü-Eintrag festlegen.

## Trennstriche

Gegebenenfalls möchten Sie zwei Gruppen von Kontextmenü-Einträgen durch einen Trennstrich voneinander abgrenzen. Diesen Trennstrich realisieren wir mit einem Separator-Element (siehe Bild 4):

```
...
<MenuItem Header="Letzter Eintrag" Click="MenuItem_Click"/>
<Separator/>
<MenuItem Header="Untermenü" Click="MenuItem_Click">
...
```

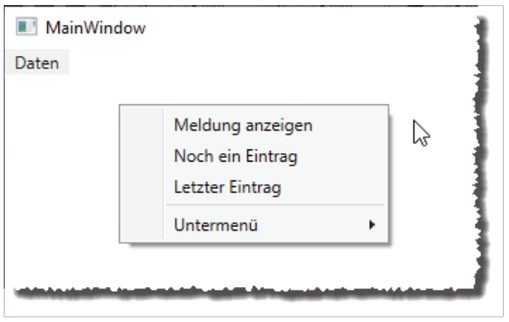


**Bild 4:** Kontextmenü mit Trennstrich

## Kontextmenüs für andere Steuerelemente

Bisher haben wir alle Kontextmenüs direkt dem **Grid**-Element hinzugefügt. Kontextmenüs für andere Steuerelemente fügen Sie auf genau die gleiche Weise hinzu – nur, dass Sie in den meisten Fällen nicht mehr das **Background**-Attribut des jeweiligen Elements anpassen müssen. Hier soll das Kontextmenü beim Rechtsklick auf ein Textfeld angezeigt werden (siehe Bild 5):

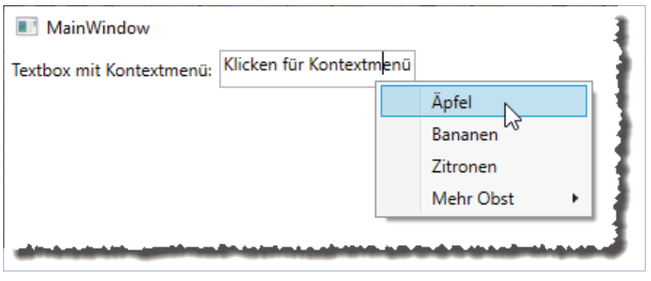


**Bild 5:** Kontextmenü für ein Textfeld

```
<TextBox Text="Klicken für Kontextmenü">
    <TextBox.ContextMenu>
        <ContextMenu>
            <MenuItem Header="Äpfel"></MenuItem>
            <MenuItem Header="Bananen"></MenuItem>
            <MenuItem Header="Zitronen"></MenuItem>
            <MenuItem Header="Mehr Obst">
                <MenuItem Header="Orange"></MenuItem>
                <MenuItem Header="Ananas"></MenuItem>
            </MenuItem>
```

```
        </ContextMenu>
    </TextBox.ContextMenu>
</TextBox>
```

## Icons im Kontextmenü

Bisher sehen die Elemente etwas trist aus. Es wäre doch schön, wenn wir mit passenden Icons ein wenig Farbe und Abwechslung hineinbringen könnten.

Auch dazu gibt es ein passendes Attribut – aber zuerst müssen Sie die Icons, die angezeigt werden sollen, zum Projekt hinzufügen. Wir fügen diese in der Regel in ein Unterverzeichnis namens **Images** ein (siehe Bild 6).

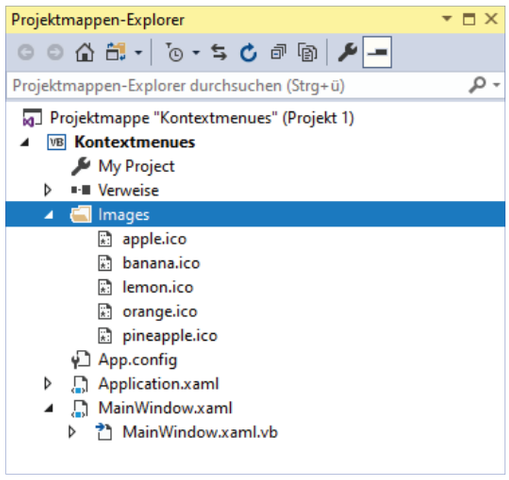


**Bild 6:** Bilddateien für die Kontextmenü-Einträge

Danach passen wir die Kontextmenü-Einträge an. Dazu benötigen wir wieder ein **Property**-Element, diesmal namens **MenuItem.Image** mit untergeordnetem **Image**-Element.

Dieses legen wir für verschiedene Einträge an und stellen für die **Source**-Eigenschaft den Namen des anzuzeigenden Icons ein. Das Ergebnis sehen Sie in Bild 7:

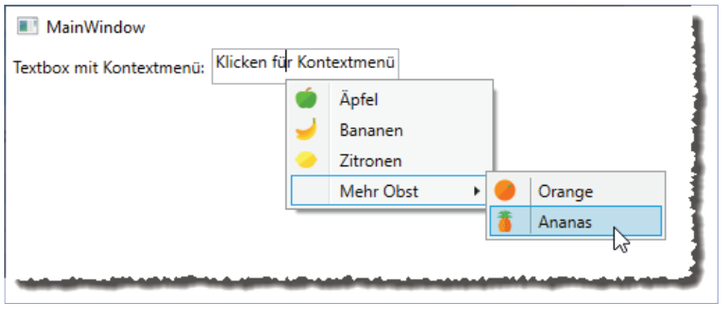


**Bild 7:** Kontextmenü mit Bildern

```
<TextBox Grid.Column="1" Text="Klicken für Kontextmenü">
    <TextBox.ContextMenu>
        <ContextMenu>
            <MenuItem Header="Äpfel">
                <MenuItem.Icon>
                    <Image Source="Images/apple.ico"></Image>
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Header="Bananen">
                <MenuItem.Icon>
                    <Image Source="Images/banana.ico"></Image>
                </MenuItem.Icon>
            </MenuItem>
            ...
        </ContextMenu>
    </TextBox.ContextMenu>
</TextBox>
```

## Kontextmenü-Einträge mit Haken

Das Kontextmenü bietet sich auch an, um einfache Optionen zu aktivieren oder zu deaktivieren. Dazu zeigt das **MenuItem**-Element dann einen Haken links vom Eintrag an – oder eben nicht. Damit das **MenuItem**-Element diese Option überhaupt anbietet, fügen wir diesem den Wert **True** für das Attribut **IsCheckable** hinzu:

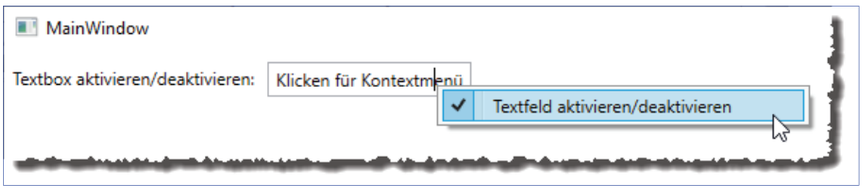


**Bild 8:** Checkbox im **MenuItem**-Element

```
<MenuItem IsCheckable="True" Header="MenuItem
mit CheckBox" />
```

Das Element sieht anschließend wie in Bild 8 aus.

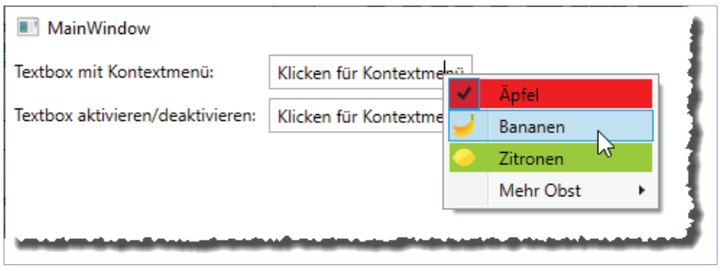


**Bild 9:** **MenuItem**-Element mit Icon und Checkbox

Wenn wir ein **MenuItem**-Element, das bereits ein Icon anzeigt, mit einer Checkbox versehen, ist diese im nicht selektierten Modus zunächst nicht sichtbar. Erst wenn Sie auf das **MenuItem**-Element klicken, erscheint bei der nächsten Anzeige der Haken der Checkbox (siehe Bild 9).

## Design von Kontextmenü-Einträgen

Eine der besonderen Stärken von WPF ist es, dass Sie jedem Element über die Attribute fast beliebige Designs zuweisen können. Im Falle von Kontextmenü-Einträgen können Sie so beispielsweise die Schriftart, -größe oder -dicke anpassen, Sie können die Hintergrundfarbe einstellen und vieles mehr.

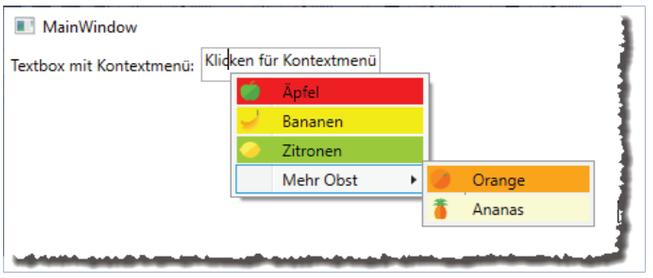


**Bild 10:** Verschiedene Hintergrundfarben

Für individuelle Anpassungen einzelner Elemente legen Sie die gewünschten Werte für die Attribute des jeweiligen Elements fest. Für ein Element mit einer anderen Hintergrundfarbe gelingt das wie folgt (siehe Bild 10):

```
<ContextMenu>
    <MenuItem Header="Äpfel" Background="Red" />
    <MenuItem Header="Bananen" Background="Yellow" />
    <MenuItem Header="Zitronen" Background="YellowGreen" />
    <MenuItem Header="Mehr Obst">
        <MenuItem Header="Orange" Background="Orange" />
        <MenuItem Header="Ananas" Background="LightGoldenrodYellow" />
    </MenuItem>
</ContextMenu>
```

# Steuerelemente mit Stil

**Visual Studio und WPF bieten verschiedene Möglichkeiten, um das Aussehen von Steuerelementen zu beeinflussen. Sie können die Eigenschaften über das Eigenschaftsfenster ändern, die Werte der Attribute direkt für das jeweilige Element im XAML-Code definieren oder auch Stilvorlagen festlegen, die entweder für alle betroffenen Elemente in der Anwendung gültig sind oder auch nur für ein bestimmtes Objekt wie ein Fenster oder eine Seite. Dieser Artikel beschreibt die verschiedenen Möglichkeiten, wie Sie die Steuerelemente Ihrer Anwendung anpassen können.**

## Eigenschaften per Eigenschaftsfenster

Der offensichtlichste Weg, um Eigenschaften für Steuerelemente einzustellen, ist das Eigenschaftsfenster. Bei einem Button-Element benötigen Sie meist die Eigenschaften der Kategorie Layout (siehe Bild 1). Sie können ein oder mehrere Steuerelemente markieren, deren Eigenschaften Sie einstellen möchten – die eingegebenen Werte werden dann auf alle aktuell markierten Steuerelemente angewendet. Weisen Steuerelemente unterschiedliche Werte für eine der Eigenschaften auf, werden die Eigenschaftswerte nicht angezeigt.

## Eigenschaften je Steuerelement per XAML

Die Änderungen der Werte im Eigenschaftsfenster wirken sich unmittelbar auf den XAML-Code des Steuerelements aus. Der XAML-Code enthält nach dem Einfügen eines Steuerelements zunächst einen minimalen Satz von Eigenschaften. Alle Eigenschaften, die nicht als Attribut mit dem Steuerelement aufgeführt werden, enthalten implizit den Standardwert.

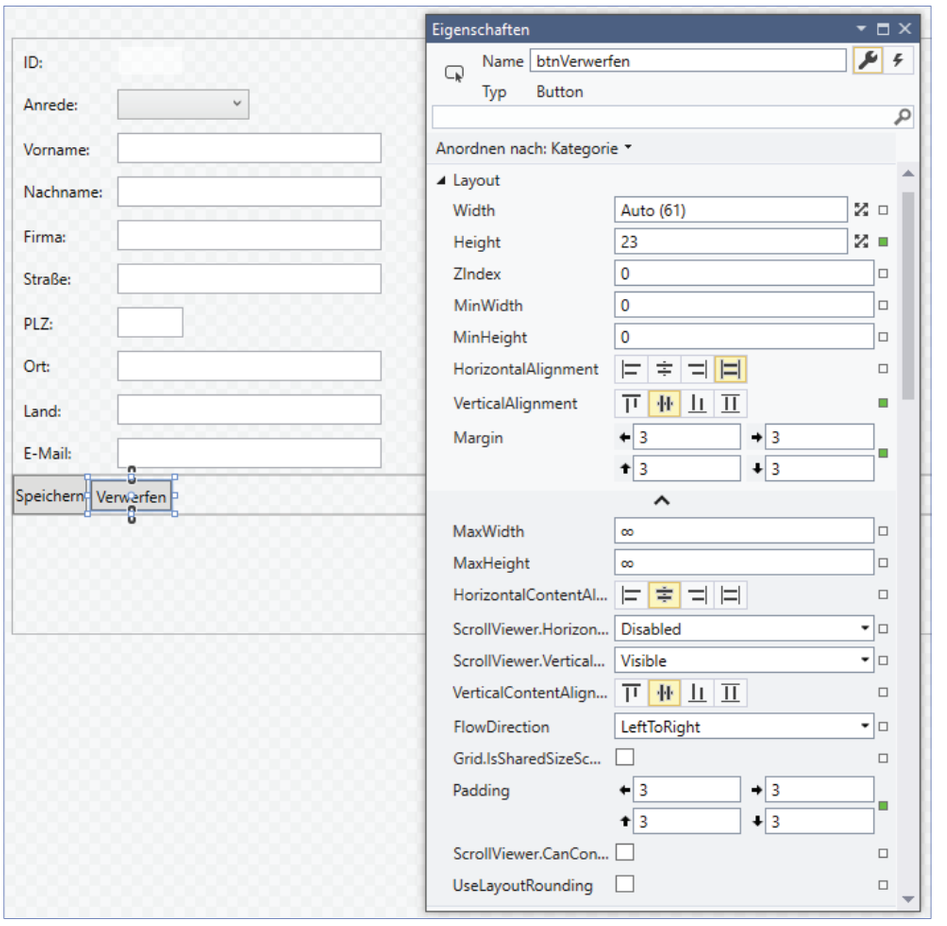


**Bild 1:** Einstellen per Eigenschaftsfenster

### Einheitliche Eigenschaften für mehrere Steuerelemente

Sie können die Attribute natürlich auch direkt im XAML-Code anpassen. Damit erreichen Sie dann umgekehrt auch eine Anpassung der Werte im Eigenschaftsfenster. Bevor Sie ein Attribut für genau ein Steuerelement anpassen, prüfen Sie, ob diese Anpassung wirklich nur für dieses Steuerelement relevant ist. Wenn Sie etwa die **Padding**-Eigenschaft ändern, die den Abstand vom Inhalt zu den Begrenzungen des Steuerelements festlegt, sollten Sie sicher sein, dass diese Eigenschaft wirklich nur explizit für ein Steuerelement geändert werden soll. Alternativ können Sie in Erwägung ziehen, dass es mehrere gleichartige Steuerelemente gibt, für die Sie diese Eigenschaft einheitlich gestalten wollen. Natürlich gibt es Eigenschaften, die tatsächlich nur ein Steuerelement betreffen – zum Beispiel die Beschriftung oder die Namen von Ereignisroutinen, die Sie für die Ereignisattribute angeben. Und auch der Steuerelementname ist natürlich ein individuelles Attribut, das Sie, wenn nötig, für jedes Steuerelement einzeln festlegen.

### Das Style-Element

Wenn es jedoch Attribute gibt, die Sie für eine Gruppe von Steuerelementen festlegen wollen, dann können Sie ein mächtiges Feature von WPF nutzen: das **Style**-Element. Der wichtigste Nutzen von **Style**-Elementen ist, Attribute zu definieren, die Sie Steuerelementen zuweisen können, statt die Attribute für jedes Steuerelement einzeln zu definieren. Wenn wir zwei Schaltflächen mit den Attributen **Padding** und **Margin** ausstatten wollen, können wir das für jede Schaltfläche individuell erledigen:

```
<StackPanel Orientation="Horizontal">
    <Button Padding="3" Margin="3">Schaltfläche 1</Button>
    <Button Padding="3" Margin="3">Schaltfläche 2</Button>
</StackPanel>
```

Der direkteste Weg, Attribute für mehrere Steuerelemente zu definieren, ist ein Style, den Sie im übergeordneten Element festlegen. Im folgenden Beispiel haben wir dem **StackPanel**-Element, in dem sich die betroffenen Steuerelemente befinden, ein **StackPanel.Resources**-Element hinzugefügt. Dieses enthält ein **Style**-Element, für das wir den **x:Key** namens **buttonStyle** angegeben haben. Damit kann der Style referenziert werden. Hier haben wir zwei **Setter**-Elemente eingefügt. Das **Setter**-Element gibt mit **Property** das Attribut an und mit **Value** den zu verwendenden Wert. Wichtig bei dieser Variante ist, dass wir dem Attributnamen die **Control**-Klasse voranstellen (**Control.Margin**).

Damit die Attribute auf die **Button**-Elemente angewendet werden, weisen wir diesen den Style explizit mit **Style="{StaticResource buttonStyle}"** zu:

```
<StackPanel Grid.Row="1" Orientation="Horizontal">
    <StackPanel.Resources>
        <Style x:Key="buttonStyle">
            <Setter Property="Control.Margin" Value="3" />
            <Setter Property="Control.Padding" Value="3" />
        </Style>
    </StackPanel.Resources>
    <Button Style="{StaticResource buttonStyle}">Schaltfläche 1</Button>
    <Button Style="{StaticResource buttonStyle}">Schaltfläche 2</Button>
```

```
</StackPanel>
```

Die Schaltflächen sehen nun genauso aus wie die aus dem vorherigen Beispiel. Wenn wir wie in diesem Beispiel nur zwei Schaltflächen mit zwei Attributen ausstatten wollen, scheint dies übertriebener Aufwand zu sein. Allerdings brauchen Sie schon die erste Änderung nur noch an einer Stelle auszuführen. Umso mehr Attribute es gibt und je mehr Steuerelementen Sie diese zuweisen, umso mehr lohnt sich das Anlegen von **Style**-Elementen.

## Styles vererben

Wenn Sie beispielsweise zwei verschiedene Typen von Schaltflächen in Ihren Fenstern verwenden wollen, von denen einige die Beschriftung in normaler Breite anzeigen, andere jedoch eine fette Schrift verwenden sollen, müssen Sie keine zwei **Style**-Elemente anlegen, wobei das zweite **Style**-Element nochmal alle Eigenschaften enthält, die das erste bereits definiert hat. Stattdessen gehen wir wie folgt vor und behalten das **Style**-Element namens **buttonStyle** bei. Außerdem legen wir ein weiteres **Style**-Element namens **buttonStyleBold** an, das nur die Eigenschaft **FontWeight** auf den Wert **Bold** festlegt. Damit dieses **Style**-Element auf dem **Style**-Element **buttonStyle** aufbaut, fügen wir diesem das Attribut **BasedOn** hinzu und stellen es auf den Namen des ersten **Style**-Elements ein, also **buttonStyle**. Den beiden Schaltflächen weisen wir dann einmal **buttonStyle** und einmal **buttonStyleBold** zu:

```
<StackPanel Grid.Row="2" Orientation="Horizontal">
    <StackPanel.Resources>
        <Style x:Key="buttonStyle">
            <Setter Property="Control.Margin" Value="3" />
            <Setter Property="Control.Padding" Value="3" />
        </Style>
        <Style x:Key="buttonStyleBold" BasedOn="{StaticResource buttonStyle}">
            <Setter Property="Control.FontWeight" Value="Bold" />
        </Style>
    </StackPanel.Resources>
    <Button Style="{StaticResource buttonStyle}">Schaltfläche 1</Button>
    <Button Style="{StaticResource buttonStyleBold}">Schaltfläche 2</Button>
</StackPanel>
```

Das Ergebnis sehen Sie in Bild 2. Auf diese Weise können Sie schnell den gewünschten Style für ein Steuerelement festlegen und Änderungen der Eigenschaften in den aufeinander aufbauenden **Style**-Elementen sind nach wie vor nur an einer Stelle nötig.

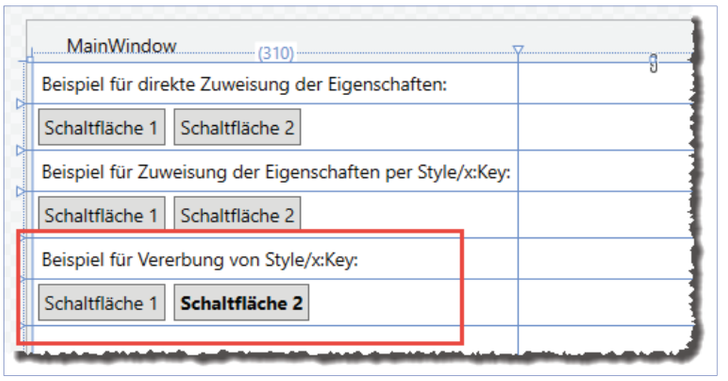


**Bild 2:** Vererbung eines Stils

## Style für bestimmte Objekttypen

Nun haben wir den Style zwar mit **buttonStyle** benannt, aber wir können diesen auch anderen

Steuerelementen zuweisen. Dabei gilt: Ein Steuerelement muss nicht alle im Style aufgelisteten Eigenschaften aufweisen. Eigenschaften, die ein Steuerelement nicht besitzt, werden dann einfach nicht gesetzt. Wenn wir also nun ein **TextBox**-Steuerelement mit dem Style **buttonStyleBold** versehen, dann zeigt auch dieses seinen Text in fetter Schrift und mit den durch Margin und Padding festgelegten Abständen an:

```
<TextBox Style="{StaticResource buttonStyle}">TextBox 1</TextBox>
<TextBox Style="{StaticResource buttonStyleBold}">TextBox 2</TextBox>
```

## Style lokal überschreiben

Wenn Sie für ein einziges Steuerelement eine Eigenschaft auf einen anderen Wert festlegen müssen als durch die Styles festgelegt, dann können Sie das ohne weiteres tun – fügen Sie einfach das entsprechende Attribut mit dem gewünschten Wert zum Element hinzu. Hier haben wir den Wert für **Margin** für die zweite Schaltfläche geändert, indem wir den linken Rand auf **0** eingestellt haben. Anderenfalls würde der Abstand zwischen den beiden Schaltflächen **6** betragen, weil die linke Schaltfläche einen rechten Rand von **3** und die rechte Schaltfläche einen linken Rand von **3**:

```
<Button Style="{StaticResource buttonStyleBold}" Margin="0,3,3,3">Schaltfläche 2</Button>
```

## Style nur für einen bestimmten Elementtyp

Sie können auch definieren, dass ein Style nur für einen speziellen Elementtyp verwendet wird. Dazu weisen Sie dem **Style**-Element über das Attribut **TargetType** den Zieltyp zu:

```
<Style x:Key="buttonStyle" TargetType="{x:Type Button}">
    <Setter Property="Control.Margin" Value="3" />
    <Setter Property="Control.Padding" Value="3" />
</Style>
```

Wenn Sie diesen Style dann einem Element zuweisen, das nicht diesem Typ entspricht, erhalten Sie direkt Feedback von Visual Studio (siehe Bild 3). Die passende Fehlermeldung lautet **TargetType 'Button'" entspricht nicht dem Typ des Elements "TextBox"**. Die Fehlermeldung verschwindet, sobald Sie das **Style**-Attribut des **TextBox**-Elements entfernen – und damit auch die durch den Style definierten Attribute.

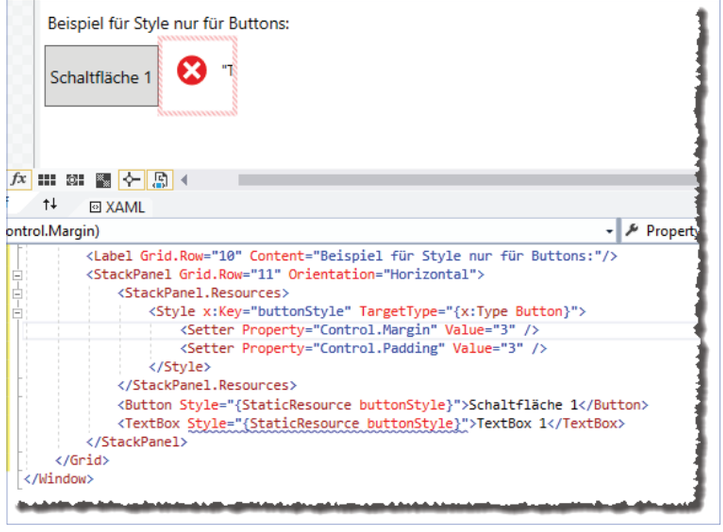


**Bild 3:** Fehler beim Zuweisen eines Styles an ein Objekt, für das dieser Style nicht vorgesehen ist.

## Styles ohne explizite Zuweisung

Sobald Sie einen Elementtypen für **TargetStyle** festgelegt haben, brauchen Sie den Style nicht mehr explizit zuzuweisen. Der Style wird dann automatisch auf alle Elemente angewendet, die

# Visual Basic: AndOr und OrElse

**Es gibt oft verschachtelte Bedingungen unter VB. Wer mit VBA programmiert hat, kennt den einzigen Weg, dies zu implementieren: indem man zwei oder mehr If...Then-Bedingungen verschachtelt. Unter Visual Basic nutzen Sie dazu die AndAlso-Verkettung. Und auch bei Ausdrücken mit Oder können Sie mit einem neuen Schlüsselwort die Performance verbessern.**

Wenn Sie unter VBA eine Aktion nur durchführen wollen, wenn zwei Bedingungen wahr sind, dann gelingt das am einfachsten mit der folgenden Schreibweise:

```
If Bedingung1 = True And Bedingung2 = True Then
    'Aktion ausführen
End If
```

Zunächst einmal lässt sich dies noch vereinfachen, denn es reicht, die Bedingungen anzugeben – diese werden ja schon auf den Wert **True** geprüft:

```
If Bedingung1 And Bedingung2 Then
    'Aktion ausführen
End If
```

Nur, wenn Sie prüfen wollen, ob der Ausdruck **Bedingung1** oder **Bedingung2** falsch ist, müssen Sie zum Beispiel **Bedingung 1 = False** angeben. Manchmal kommen Sie mit dieser Schreibweise jedoch nicht weiter – zum Beispiel, wenn **Bedingung2** nur auswertbar ist, wenn **Bedingung1** wahr ist. Wenn Sie also in der ersten Bedingung testen, ob ein bestimmtes XML-Element vorhanden ist und in der zweiten dann prüfen, ob der Wert des XML-Elements einen bestimmten Wert hat, funktionierte das nur fehlerfrei, wenn die erste Bedingung erfüllt ist. Ist das nicht der Fall, ist XML-Element nicht vorhanden und die zweite Bedingung löst beim Zugriff auf das XML-Element einen Fehler aus. In diesem Fall verwenden wir zwei verschachtelte **If...Then**-Bedingungen:

```
If Bedingung1 = True Then
    If Bedingung2 = True Then
        'Aktion ausführen
    End If
End If
```

Unter VB können Sie dies viel einfacher formulieren. Dazu nutzen wir das Schlüsselwort **AndAlso**:

```
If Bedingung1 = True AndAlso Bedingung2 = True Then
    'Aktion ausführen
End If
```

Das **AndAlso**-Schlüsselwort bewirkt, dass die zweite Bedingung nur dann ausgewertet wird, wenn die erste Bedingung wahr ist. Allein aus Performance-Gründen sollte man **AndAlso** also dem **And**-Operator vorziehen. Außerdem können Sie auch die Boolean-Ausdrücke an anderen Stellen mit dem AndAlso-Operator verknüpfen – also zum Beispiel in der Abbruchbedingung von Schleifen:

```
Do While Bedingung1 = False AndAlso Bedingung2 = False
...
```

## Or oder OrElse?

Neben der Variante **AndAlso** für das **And**-Schlüsselwort gibt es auch noch den **OrElse**-Operator, den Sie alternativ zum **Or**-Operator einsetzen können. Bei einer **Or**-Verknüpfung wie der folgenden werden immer alle Ausdrücke ausgewertet:

```
If Bedingung1 Or Bedingung2 Then
    'Aktion ausführen
End If
```

Wenn Sie hingegen das Schlüsselwort **OrElse** verwenden, wird zunächst **Bedingung1** ausgewertet und nur, wenn diese nicht wahr ist, wird **Bedingung2** herangezogen:

```
If Bedingung1 OrElse Bedingung2 Then
    'Aktion ausführen
End If
```

## Testen, ob das stimmt

Wenn Sie sicherstellen wollen, ob **AndAlso** und **OrElse** wie angegeben funktionieren, können Sie als Bedingungen einfach zwei Funktionen verwenden, die beim Aufruf eine Meldung ausgeben. So sehen Sie, ob die zweite Bedingung abhängig von der ersten Bedingung ausgewertet wird.

Die beiden Funktionen definieren wir wie folgt:

```
Public Function Bedingung1 As Boolean
    Debug.Print("Aufruf Bedingung1")
    Return False
End Function

Public Function Bedingung2 As Boolean
    Debug.Print("Aufruf Bedingung2")
    Return True
End Function
```

Die folgende Methode verknüpft die beiden Bedingungen mit dem **And**-Schlüsselwort:

# Programmieren der Zwischenablage mit VB

**Unter VBA war das Zugreifen auf die Inhalte der Zwischenablage mit dem Einsatz einiger API-Funktionen verbunden. Unter VB und .NET gelingt das wesentlich einfacher. Dieser Artikel zeigt, wie Sie Inhalt in die Zwischenablage kopieren, den Inhalt der Zwischenablage auslesen und weitere Funktionen nutzen wie etwa das Ermitteln des Typs des Inhalts der Zwischenablage.**

## Die ClipBoard-Klasse

Die **ClipBoard**-Klasse ist das Herzstück der nachfolgend vorgestellten Techniken. Da die Zwischenablage computerweit genutzt wird, also auch von anderen Programmen und auch von Windows selbst, referenzieren wir die **ClipBoard**-Klasse über **My.Computer**. **My** ist eine Funktion, die einen einfachen Zugriff auf verschiedene Klassen bietet, die sich auf den Computer, die Anwendung, Einstellungen, Ressourcen und so weiter beziehen.

Wir referenzieren über **My** die **Computer**-Klasse, die uns auch die **ClipBoard**-Klasse zur Verfügung stellt. Die **ClipBoard**-Klasse bietet wiederum die folgenden Möglichkeiten – in alphabetischer Reihenfolge:

- **Clear**: Leert den Inhalt der Zwischenablage. Vorsicht, da die Zwischenablage ja auch von anderen Anwendungen genutzt wird!
- **ContainsAudio**: Gibt an, ob die Zwischenablage **WaveAudio**-Inhalt enthält.
- **ContainsData(<String>)**: Gibt an, ob die Zwischenablage Daten des mit dem Parameter **String** angegebenen Typs enthält. Beispielwerte für den Parameter sind **Bitmap**, **StringFormat**, **WaveAudio** et cetera.
- **ContainsFileDropList**: Gibt an, ob die Zwischenablage Daten im **FileDrop**-Format enthält.
- **ContainsImage**: Gibt an, ob sich in der Zwischenablage Daten im Bitmap-Format befinden.
- **ContainsText**: Gibt an, ob die Zwischenablage Text enthält.
- **ContainsText(Optional <TextDataFormat>)**: Überladung von **ContainsText**. Gibt an, ob die Zwischenablage Text enthält. Der optionale Parameter ermöglicht eine genauere Prüfung des Inhalts. Hier gibt es die folgenden Werte: **Text** (**0**), **UnicodeText** (**1**), **Rtf** (**2**), **Html** (**3**), **CommaSeparatedValue** (**4**)
- **GetAudioStream**: Liefert einen in der Zwischenablage gespeicherten Audiostream.
- **GetData(<String>)**: Liefert den Inhalt der Zwischenablage in dem mit dem Parameter **String** angegebenen Format.
- **GetDataObject**: Ruft die Daten der Zwischenablage ab – unabhängig vom Typ.

- **GetFileDropList**: Ruft einen oder mehrere Dateinamen aus der Zwischenablage ab.
- **GetImage**: Ruft ein Bild aus der Zwischenablage ab.
- **GetText**: Ruft Text aus der Zwischenablage ab.
- **GetText(<TextDataFormat>)**: Überladung von **GetText**. Ruft Text im angegebenen Textformat ab. Hier gibt es die folgenden Werte: **Text** (**0**), **UnicodeText** (**1**) , **Rtf** (**2**), **Html** (**3**), **CommaSeparatedValue** (**4**)
- **SetAudio(<Byte[]>)**: Fügt ein Byte-Array im WaveAudio-Format als Stream hinzu.
- **SetAudio(<Stream>)**: Überladung von **SetAudio**. Fügt den mit dem Parameter referenzierten Stream im WaveAudio-Format hinzu.
- **SetData(<String>, <Object>)**: Fügt die Daten aus dem Parameter **Object** im mit dem Parameter **String** angegebenen Format in die Zwischenablage ein.
- **SetDataObject(<Object>)**: Fügt ein Objekt in die Zwischenablage ein. Dieses Objekt ist nur temporär bis zum Beenden der auslösenden Anwendung in der Zwischenablage verfügbar.
- **SetDataObject(<Object>, <Boolean>)**: Überladung von **SetDataObject**. Der Parameter **Boolean** gibt an, ob die in die Zwischenablage eingefügten Daten auch nach dem Beenden der Anwendung noch in der Zwischenablage verfügbar sein sollen.
- **SetDataObject(<Object>, <Boolean>, <Int32>, <Int32>)**: Überladung von **SetDataObject**. Die beiden **Int32**-Parameter geben an, wie oft versucht wird, **<Object>** in die Zwischenablage zu übertragen und in welchem zeitlichen Abstand.
- **SetFileDropList(StringCollection)**: Fügt eine Liste von einem oder mehreren Dateipfaden in einer **StringCollection** zur Zwischenablage hinzu.
- **SetImage(<Image>)**: Fügt ein Objekt des Datentyps **Bitmap** zur Zwischenablage hinzu.
- **SetText(<String>)**: Fügt einen Text zur Zwischenablage hinzu.
- **SetText(<String>, <TextDataFormat>)**: Überladung von **SetText**. Fügt einen Text mit dem in **TextDataFormat** angegebenen Format zur Zwischenablage hinzu.

### Text in der Zwischenablage

Das Hinzufügen von Text in die Zwischenablage ist schnell erledigt. In einem Beispielprojekt haben wir dazu ein Textfeld namens **txtText** und eine Schaltfläche namens **btnInZwischenablage** hinzugefügt:

```
<Button x:Name="btnInZwischenablage" Click="BtnInZwischenablage_Click">In Zwischenablage</Button>
```

```
<Label>Text:</Label>
<TextBox x:Name="txtZwischenablage" Width="200"></TextBox>
```

Ein Klick auf die Schaltfläche **btnInZwischenablage** soll den aktuellen Inhalt des Textfeldes **txtZwischenablage** in die Zwischenablage kopieren, was wir so realisieren:

```
Private Sub BtnInZwischenablage_Click(sender As Object, e As RoutedEventArgs)
    Clipboard.SetText(txtZwischenablage.Text)
End Sub
```

Das Ergebnis sehen Sie in Bild 1.

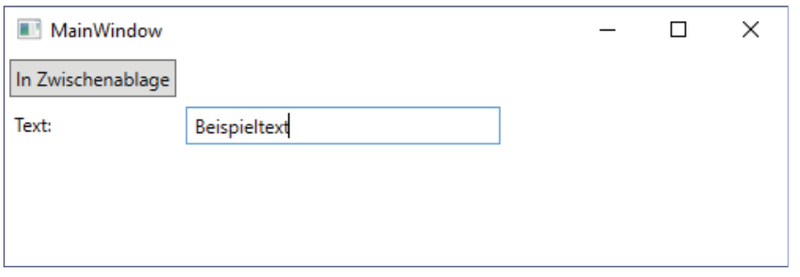


**Bild 1:** Text aus einem Textfeld in die Zwischenablage kopieren

## Text aus der Zwischenablage abrufen

Wenn Sie den soeben in der Zwischenablage gespeicherten Text oder einen anderen in der Zwischenablage gespeicherten Text auslesen wollen, verwenden Sie die Methode **GetText**.

Eine zweite Schaltfläche namens **btnAusZwischenablage** soll den Inhalt der Zwischenablage in das Textfeld kopieren. Dazu verwenden wir schlicht die Methode **GetText** der **ClipBoard**-Klasse:

```
Private Sub BtnAusZwischenablage_Click(sender As Object, e As RoutedEventArgs)
    txtZwischenablage.Text = Clipboard.GetText
End Sub
```

Damit wird allerdings der komplette Inhalt des Textfeldes überschrieben. Um den Inhalt der Zwischenablage an der Position der Einfügemarke einzufügen beziehungsweise anstelle des aktuell markierten Textes, verwenden Sie die folgende Anweisung:

```
txtZwischenablage.SelectedText = Clipboard.GetText()
```

## Prüfen, ob die Zwischenablage Text enthält

Wenn die Zwischenablage keinen Text enthält, wird auch nichts in die Zwischenablage eingefügt. Wenn Sie vor dem Einfügen von Text prüfen wollen, ob die Zwischenablage überhaupt Text enthält, können Sie die **ContainsText**-Methode wie folgt verwenden:

```
If Clipboard.ContainsText Then
    txtZwischenablage.SelectedText = Clipboard.GetText()
Else
    MessageBox.Show("Zwischenablage enthält keinen Text.")
```

```
End If
```

### Zwischenablage leeren

Mit der **Clear**-Methode der **Clipboard**-Klasse leeren Sie die Zwischenablage:

```
Private Sub BtnZwischenablageLeeren_Click(sender As Object, e As RoutedEventArgs)
    Clipboard.Clear()
End Sub
```

### Zwischenablage und Dateien

Wie genau können wir die Zwischenablage mit Dateien nutzen? Wir können beispielsweise eine Liste von Dateien im Windows Explorer markieren, per **Strg + C** in die Zwischenablage kopieren und die Namen dieser Dateien über eine Schaltfläche dann aus der Zwischenablage weiterverarbeiten. Wir fügen wieder zwei Schaltflächen hinzu, welche zum Hinzufügen von Daten zur Zwischenablage und zum Auslesen der Zwischenablage dienen. Außerdem legen wir ein Listenfeld an, in dem wir die Daten aus der Zwischenablage anzeigen wollen:

```
<Button x:Name="btnInZwischenablageDatei" Click="BtnInZwischenablageDatei_Click">In Zwischenablage</Button>
<Button x:Name="btnAusZwischenablageDatei" Click="BtnAusZwischenablageDatei_Click">Aus Zwischenablage</Button>
<Label>Text:</Label>
<ListBox x:Name="lstZwischenablageDatei" Width="300" Height="100"></ListBox>
```

Dann gehen wir davon aus, dass der Benutzer eine oder mehrere Dateien im Windows Explorer markiert und per Strg + C in die Zwischenablage kopiert hat und dann die Schaltfläche **btnAusZwischenablageDatei** betätigt. Dies löst die folgende Methode aus. Für die dort verwendete Variable des Typs **StringCollection** müssen wir der Code behind-Klasse noch einen Verweis auf den folgenden Namespace hinzufügen:

```
Imports System.Collections.Specialized
```

Die **StringCollection** benötigen wir, da die **GetFileDropList** ein Ergebnis dieses Typs zurückliefert. Dementsprechend deklarieren wir eine Variable dieses Typs und noch eine **String**-Variable namens **strDatei**, um die einzelnen Dateien dieser Auflistung zu durchlaufen:

```
Private Sub BtnAusZwischenablageDatei_Click(sender As Object, e As RoutedEventArgs)
    Dim strDateien As StringCollection
    Dim strDatei As String
```

Danach prüft die Prozedur, ob die Zwischenablage überhaupt eine **FileDropList** enthält:

```
    If Clipboard.ContainsFileDropList Then
```

Falls ja, ermitteln wir diese mit **GetFileDropList** und weisen sie der Variablen **strDateien** zu:

```
        strDateien = Clipboard.GetFileDropList
```

Außerdem leeren wir das Listenfeld für den Fall, dass dieses bereits Daten enthält:

```
        lstZwischenablageDatei.Items.Clear()
```

Schließlich durchlaufen wir diese Liste in einer **For Each**-Schleife und fügen die einzelnen Elemente jeweils dem Listenfeld als neue Einträge hinzu:

```
        For Each strDatei In strDateien
            lstZwischenablageDatei.Items.Add(strDatei)
        Next
    Else
        MessageBox.Show("Keine FileDropList")
    End If
End Sub
```

Das Ergebnis sehen Sie in Bild 2. Aber bekommen wir die Auflistung auch wieder in diesem Format in die Zwischenablage zurück? Das untersuchen wir nun.

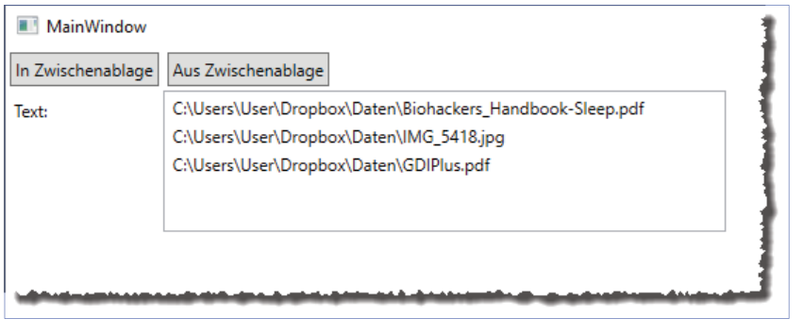


**Bild 2:** Aus dem Windows Explorer kopierte Pfade

## Dateien per ListBox in die Zwischenablage

Wir wollen untersuchen, ob wir die Liste von Dateipfaden in die Zwischenablage kopieren können und ob wir damit dann die Dateien an anderer Stelle über den Windows Explorer einfügen können – im Prinzip also so, als ob wir die Dateien direkt von einem Verzeichnis in das nächste im Windows Explorer kopieren. Der Unterschied wäre nur, dass wir die Pfade der zu kopierenden Dateien dann ohne Verwendung des Windows Explorers zusammenstellen und dann in die Zwischenablage kopieren könnten.

Das erledigen wir mit der folgenden Methode, die zunächst ein neues **StringCollection**-Objekt erstellt:

```
Private Sub BtnInZwischenablageDatei_Click(sender As Object, e As RoutedEventArgs)
    Dim i As Integer
    Dim strDatei As String
    Dim strDateien As StringCollection
    strDateien = New StringCollection
```

Dann durchläuft sie alle Einträge des Listenfeldes **lstZwischenablageDatei** über dessen **Items**-Auflistung und fügt diese mit der **Insert**-Methode zum **StringCollection**-Objekt hinzu:

# Access zu EDM: Dateien erstellen

**In vorangegangenen Artikeln haben wir Prozeduren erstellt, mit denen wir das Datenmodell einer Access-Datenbank einlesen und daraus ein Entity Data Model erstellen können. Zusätzlich haben wir auch noch die enthaltenen Daten ausgelesen und Code erzeugt, mit denen eine auf Basis des Entity Data Models erstellte SQL Server-Datenbank gefüllt werden kann. In diesem Artikel wollen wir einen Schritt weitergehen: Bisher haben wir den Code in die Zwischenablage kopiert, sodass der Benutzer diesen noch in die entsprechenden Module des Visual Studio-Projekts kopieren musste. Nun wollen wir direkt die passenden Module als Dateien erstellen, die nur noch in das Projekt gezogen werden müssen.**

Die Artikel, in denen wir das Erstellen des Entity Data Models auf Basis des Datenmodells einer Access-Datenbank beschrieben haben, heißen **Von Access zu Entity Framework: Datenmodell** (**www.datenbankentwickler.net/148**), **Von Access zu Entity Framework: Daten** (**www.datenbankentwickler.net/149**) und **Von Access zu Entity Framework: Update 1** (**www.datenbankentwickler.net/164**).

Die hier beschriebenen Prozeduren im VBA-Projekt einer Access-Datenbank lesen die Struktur des Datenmodells der Datenbank aus, die migriert werden soll, und schreibt die ermittelten Codezeilen in die Zwischenablage. Von dort aus sollen diese dann in die betroffenen Zielmodule des Visual Studio-Projekts eingefügt werden. Das ist noch etwas unkomfortabel, sodass wir die Lösung upgraden wollen – und zwar so, dass wir im Verzeichnis der Access-Datenbank neue Dateien und Verzeichnisse erhalten, die wir direkt in den Projektmappen-Explorer von Visual Studio ziehen können.

## Wie soll die Lösung aussehen?

Der Ausgangspunkt ist wieder, dass Sie in dem Projekt mit dem zu erstellenden Entity Data Model ein neues Element des Typs **ADO.NET Entity Data Model** hinzufügen (siehe Bild 1). Hier geben Sie direkt die Bezeichnung der Context-Klasse an, die wir später beim Zugriff über das Entity Data Model auf die in der Datenbank gespeicherten Daten verwenden werden – in diesem Fall **RechnungsverwaltungContext**.

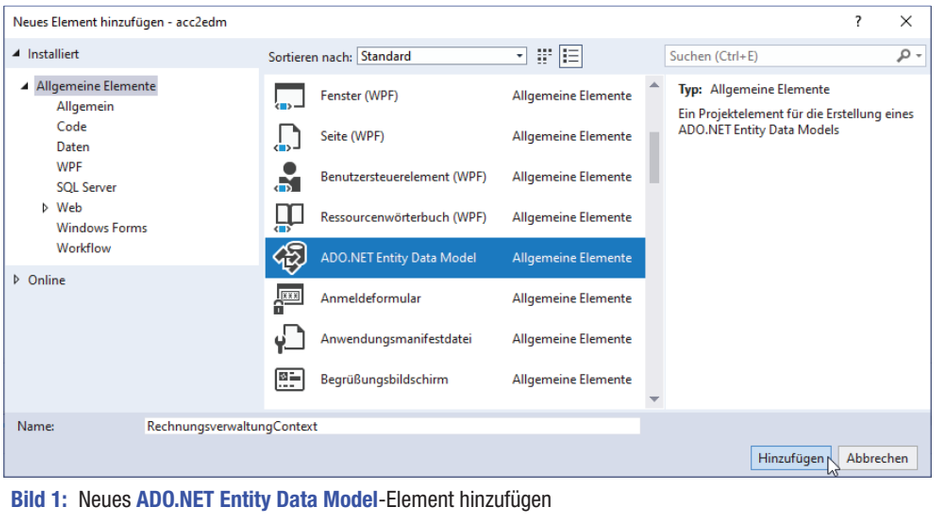


**Bild 1:** Neues **ADO.NET Entity Data Model**-Element hinzufügen

Im nächsten Schritt wählen wir dann den Eintrag **Leeres Code First-Modell** aus (siehe Bild 2). Damit erscheint im Projektmappen-Explorer zunächst ein neues Element

mit dem Namen, den wir dem neuen **ADO.NET Entity Data Model**-Element zugewiesen haben.

Für die Erstellung des Entity Data Models auf Basis des Datenmodells einer Access-Datenbank merken wir uns diese Bezeichnung.

Die Erstellung erfolgt dann durch einen einmaligen Aufruf einer Prozedur namens **EDMDateienErstellen**. Diese erwartet drei Parameter:

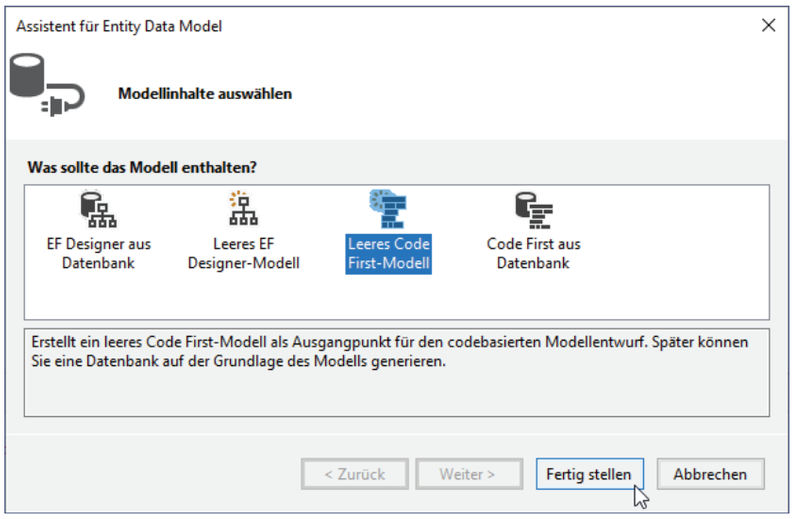


**Bild 2:** Die Wahl fällt auf **Leeres Code First-Modell**.

- **bolEineDatei**: Gibt an, ob die Entitätsklassen direkt in die Contextklasse geschrieben werden sollen oder als einzelne Dateien in ein Unterverzeichnis.
- **strContextname**: Erwartet den Namen des zuvor erstellten **ADO.NET Entity Data Model**-Elements.
- **strUnterverzeichnisEntities**: Name des Unterverzeichnisses, in das die einzelnen Klassendateien mit den Entitäten geschrieben werden sollen.

Ein Beispielaufruf sieht etwa wie folgt aus:

```
EDMDateienErstellen False, "RechnungsverwaltungContext", "DataModel"
```

Das Ergebnis finden Sie in Bild 3. Die Datei **RechnungsverwaltungContext.vb** enthält die Anweisungen, die Sie auch in der vom Assistenten erstellten gleichnamigen Datei vorfinden – erweitert um die Definition der **DbSet**-Auflistungen wie zum Beispiel die folgende:

```
Public Overridable Property Anreden() As DbSet(Of Anrede)
```

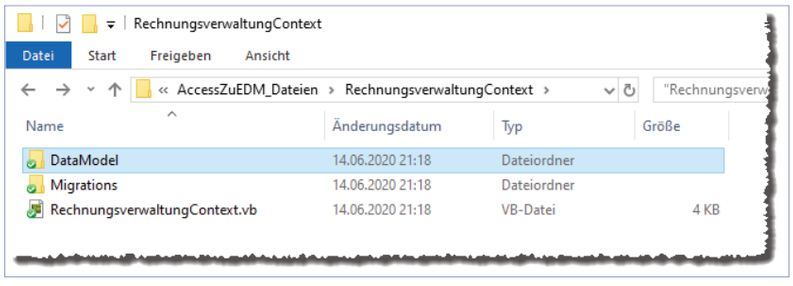


**Bild 3:** Ergebnis nach dem Aufruf der Prozedur **EDMDateienErstellen**

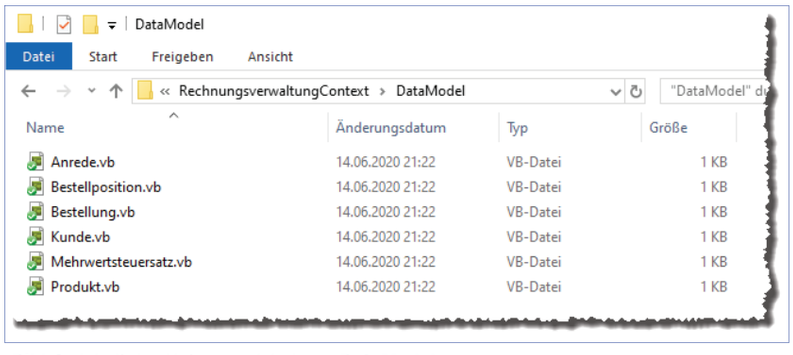


**Bild 4:** Inhalt des Unterverzeichnisses **DataModel**

Das Verzeichnis **DataModel** enthält die Dateien aus Bild 4. Jede dieser Dateien enthält die Definition einer Entität des Entity Data Models. Die für die Tabelle **tblAnreden** sieht etwa wie folgt aus:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

<Table("Anreden")>
Public Partial Class Anrede
    Public Property ID As System.Int32
    <Index(IsUnique:=true)>
    <StringLength(255)>
    <Required>
    Public Property Name As System.String
End Class
```

Das Ergebnis entspricht also inhaltlich dem aus den in den weiter oben referenzierten Artikeln beschriebenen Prozeduren – mit dem Unterschied, dass diese nun direkt in einem handlichen Verzeichnis liegen. Auf die gleiche Weise werden auch die übrigen Entitätsklassen in jeweils einer Datei in das Verzeichnis kopiert.

## Dateien zum Projekt hinzufügen

Nun folgt der wichtigste Schritt: Wir wollen die Dateien, die wir soeben erstellt haben, zum Projekt hinzufügen. Dazu öffnen Sie das neu erstellte Verzeichnis, dass die Unterverzeichnisse **DataModel** und **Migrations** sowie die Datei **RechnungsverwaltungContext.vb** enthält und markieren die enthaltenen Elemente. Diese ziehen Sie dann in den Projektmappen-Explorer auf das Projekt-Element. Der Projektmappen-Explorer sieht danach wie in Bild 5 aus.

Da der Ordner **Migrations** bereits vorhanden ist, brauchen Sie in der Paket-Manager-Konsole nun nur noch zwei der sonst üblichen drei Anweisungen auszuführen, damit die Anweisungen zum Erstellen der Datenbank hinzugefügt und die Datenbank erstellt wird. Die Anweisung **enable-migrations** entfällt beziehungsweise ihr Aufruf liefert nur den Hinweis, dass die Migrationen für dieses Projekt bereits aktiviert wurden.

Die verbleibenden Anweisungen lauten:

```
add-migration init
update-database
```

Erstere stellt die Methoden zusammen, mit denen die Datenbank gefüllt werden soll, letztere ruft diese auf und erstellt so die Datenbank.

Damit erhalten wir dann die Datenbank aus Bild 6.

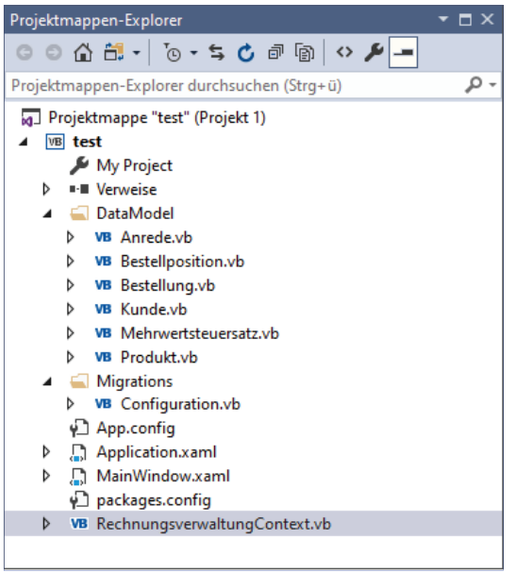


**Bild 5:** Neue Elemente im Projektmappen-Explorer

## Programmierung der Prozeduren zum Erstellen der Klassen

Die Hauptprozedur heißt **EDMDateienErstellen** und erwartet die bereits weiter oben beschriebenen Parameter. Außerdem verwendet sie einige Variablen, die wie folgt deklariert werden:

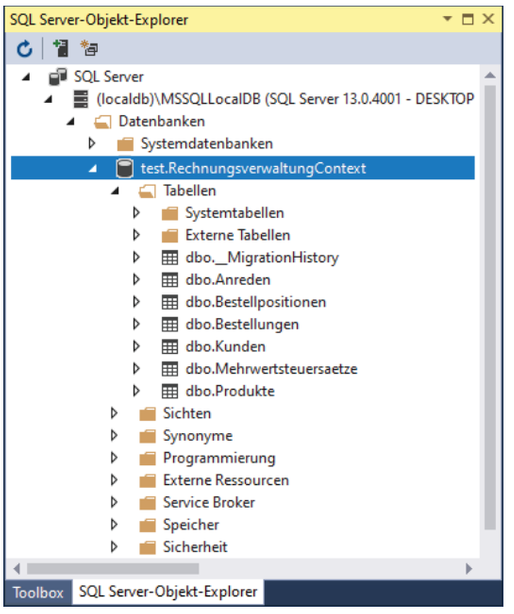


**Bild 6:** Die angelegte Datenbank

```
Public Sub EDMDateienErstellen(bolEineDatei As Boolean, strContextna-
me As String, strUnterverzeichnisEntities As String)
    Dim strDBSets As String
    Dim strEntities() As String
    Dim strEntityNames() As String
    Dim strAllEntities As String
    Dim strContextklasse As String
    Dim strNamespacesEntity As String
    Dim colMappings As Collection
    Dim objMapping As clsMapping
    Dim strSeed As String
    Dim strAddOrUpdate As String
    Dim i As Integer
    Dim strBasispfad As String
```

Als Erstes legen wir einen Basispfad fest, der aus dem aktuellen Datenbankverzeichnis sowie dem Unterverzeichnis, das nach dem Wert des Parameters **strContextname** benannt wird. Damit rufen wir eine Prozedur namens **MakeDirectory** auf, die lediglich einen **MkDir**-Aufruf bei deaktivierter Fehlerbehandlung durchführt – auf diese Weise unterbinden wir Fehler, die beim Erstellen von bereits vorhandenen Verzeichnissen gemeldet werden:

```
    strBasispfad = CurrentProject.Path & "\" & strContextname
    MakeDirectory strBasispfad
```

Die Informationen über die Mappings von den Datenbanktabellen auf die Entitäten holen wir mit einer Funktion namens **EntityDataModelZusammenstellen**. Diese hat im Wesentlichen den Aufbau wie die Prozedur **EDMErstellen**, die wir bereits in den eingangs genannten Artikeln beschrieben haben. Wir haben diese nur so umgebaut, dass der dort erstellt Quellcode nicht mehr als Ganzes in die Zwischenablage kopiert wird, sondern mit den drei Variablen **strDBSets**, **strEntites** und **strEntitiyNames** (die letzten beiden als Array) zurückgeliefert wird. Außerdem liefert die Funktion **EntityDataModelZusammenstellen** als Funktionswert eine Collection zurück, welche die Basisinformationen über die Mappings enthält, die wir nochmal benötigen:

```
    Set colMappings = EntityDataModelZusammenstellen(strDBSets, strEntities, strEntityNames)
```

Der Rückgabeparameter **strDBSets** liefert die Anweisungen, welche die **DbSet**-Objekte deklarieren. Diese übergeben wir an die Methode **ContextklasseZusammenstellen**, die diese Anweisungen um den Rest der Context-Klasse anreichert. Diese Funktion beschreiben wir weiter unten. Das Ergebnis speichern wir zunächst in der Variablen **strContextklasse**:

# Access zu EDM: INotifyPropertyChanged integrieren

**Im Beitrag »Access zu EDM: Dateien erstellen« haben wir gezeigt, wie Sie mit einer einfachen Access-Prozedur aus einem Access-Datenmodell die für ein Entity Data Model notwendigen Dateien erzeugen können. Das Entity Data Model enthält nur die reinen Eigenschaften, welche die Felder der jeweiligen Tabellen repräsentieren. Manchmal benötigen Sie allerdings mehr als nur diese Eigenschaften – dann soll zum Beispiel die Schnittstelle INotifyPropertyChanged in der Entitätsklasse implementiert sein, um Änderungen in den Eigenschaften schnell in die Anzeige der Daten übermitteln zu können. Wie Sie die Access-Prozedur zum Erstellen des Entity Data Models entsprechend erweitern, zeigen wir Ihnen in diesem Artikel.**

Unsere Entitätsklassen, die wir mit der Prozedur **EDMDateienErstellen** erzeugen, bestehen nur aus den reinen Feldern samt Attributen:

```
<Table("Anreden")>
Public Partial Class Anrede
    Public Property ID As System.Int32
    <Index(IsUnique:=true)>
    <StringLength(255)>
    <Required>
    Public Property Name As System.String
End Class
```

Für die Implementierung der **INotifyPropertyChanged**-Schnittstelle sind einige Erweiterungen notwendig, die wir in den folgenden Abschnitten hinzufügen.

## Namespace hinzufügen

Um von der Form der Entitätsklasse, wie wir sie in der bisher verwendeten Prozedur erstellt haben, zu einer Form mit Implementierung der Schnittstelle **IPropertyNotifyChanged** zu gelangen, müssen wir einige Elemente zur Entitätsklasse hinzufügen. Das erste ist der Namespace, in der diese Schnittstelle definiert wird. Dazu passen wir die Funktion **NamespacesFuerEntitaeten** an, indem wir dieser einen Parameter namens **bolINotifyPropertyChanged** hinzufügen. Diesen werten wir in der Funktion aus und fügen, wenn diese den Wert **True** hat, noch die folgende Zeile hinzu:

```
Imports System.ComponentModel
```

In der Funktion sieht das wie folgt aus:

```
Public Function NamespacesFuerEntitaetenZusammenstellen(Optional bolINotifyPropertyChanged As Boolean) As String
    Dim strNamespaces As String
```

```
    If bolINotifyPropertyChanged Then
        strNamespaces = strNamespaces & "Imports System.ComponentModel" & vbCrLf
    End If
    ...
    NamespacesFuerEntitaetenZusammenstellen = strNamespaces
End Function
```

### Implements-Anweisung hinzufügen

Als Nächstes wollen wir der Entitätsklasse die **Implements**-Anweisung hinzufügen, mit der wir festlegen, dass die Klasse die Schnittstelle **INotifyPropertyChanged** implementiert. Dieser Teil erfolgt in der Funktion **EntityDataModelZusammenstellen**, die wir von der Hauptprozedur **EDMDateienErstellen** aus aufrufen. Dazu fügen wir dem Aufruf die Übergabe des Parameters **bolINotifyPropertyChanged** hinzu:

```
Public Sub EDMDateienErstellen(bolEineDatei As Boolean, strContextname As String, _
        strUnterverzeichnisEntities As String, Optional bolINotifyPropertyChanged As Boolean)
    ...
    Set colMappings = EntityDataModelZusammenstellen(strDBSets, strEntities, strEntityNames, bolINotifyPropertyChanged)
    ...
End Sub
```

In der Funktion **EntityDataModelZusammenstellen** nehmen wir diesen Parameter im Funktionskopf entgegen und prüfen dann wieder per **If...Then**-Bedingung, ob **bolINotifyPropertyChanged** den Wert **True** hat. In diesem Fall fügen wir der Definition der Klasse in der Variablen **strEntities(i)** die Zeile **Implements INotifyPropertyChanged** hinzu:

```
Public Function EntityDataModelZusammenstellen(strDBSets As String, strEntities() As String, _
        strEntityNames() As String, Optional bolINotifyPropertyChanged As Boolean) As Collection
    ...
    For Each objMapping In colMappings
        With objMapping
            ...
            strEntities(i) = "<Table(""" & .Entities & """)>" & vbCrLf
            strEntities(i) = strEntities(i) & "Public Partial Class " & .Entity & vbCrLf
            If bolINotifyPropertyChanged Then
            If bolINotifyPropertyChanged Then
                strEntities(i) = strEntities(i) & "    Implements INotifyPropertyChanged" & vbCrLf
                strEntities(i) = strEntities(i) & vbCrLf
                strEntities(i) = strEntities(i) & "    Public Event PropertyChanged As PropertyChangedEventHandler " _
                    & "Implements INotifyPropertyChanged.PropertyChanged" & vbCrLf
                strEntities(i) = strEntities(i) & vbCrLf
                strEntities(i) = strEntities(i) & "    Protected Overridable Sub OnPropertyChanged(propname As String)" _
                   & vbCrLf
```

```
                strEntities(i) = strEntities(i) _
                    & "        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propname))" & vbCrLf
                strEntities(i) = strEntities(i) & "    End Sub" & vbCrLf
                strEntities(i) = strEntities(i) & vbCrLf
            End If
    ...
End Function
```

Wie Sie sehen, fügen wir auf die gleiche Weise auch gleich die Implementierung dieser Schnittstelle hinzu. Dabei handelt es sich um die folgenden Zeilen, die unabhängig vom Klassennamen immer gleich lauten:

```
Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged

Protected Overridable Sub OnPropertyChanged(propname As String)
    RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propname))
End Sub
```

### Eigenschaften um den OnPropertyChanged-Aufruf erweitern

Im letzten Schritt müssen wir noch die als automatisch implementierten Eigenschaften in der langen Form schreiben, damit wir den Aufruf der Methode **OnPropertyChanged** in den Setter einfügen können. Bisher sieht eine Eigenschaft beispielsweise so aus:

```
   <Required>
    Public Property Bezeichnung As System.String
```

Sie soll nun aber so aussehen:

```
    Private _Bezeichnung As String
    Public Property ID As System.Int32
    <StringLength(255)>
    Public Property Bezeichnung As System.String
        Get
            Return _Bezeichnung
        End Get
        Set
            _Bezeichnung = Value
        End Set
        OnPropertyChanged("Bezeichnung")
    End Property
```

# Access zu EDM: Validierung

**Im Beitrag »Access zu EDM: Dateien erstellen« haben wir gezeigt, wie Sie mit einer einfachen Access-Prozedur aus einem Access-Datenmodell die für ein Entity Data Model notwendigen Dateien erzeugen können. Wenn Sie nun noch passende Klassen mit den Grundfunktionen für die Validierung hinzufügen wollen, können Sie die Erweiterungen aus diesem Artikel dazu verwenden. Wir legen für jede Entitätsklasse des Entity Data Models auf Basis des Datenmodells einer Access-Datenbank zusätzlich eine weitere Klasse in einem Unterordner namens Validierung an, der die grundlegenden Funktionen für die Validierung enthält. Diese müssen allerdings noch angepasst werden. An welchen Stellen, lesen Sie im Folgenden.**

Die grundlegenden Techniken zum Validieren von Daten in Benutzeroberflächen, die Daten aus einem Entity Data Model nutzen, haben wir im Artikel **Validieren mit VB und EDM** beschrieben (**www.datenbankentwickler.net/175**). Dort zeigen wir eine Methode, die in verschiedenen Bereichen implementiert werden muss. Der erste dort beschriebene Teil erläutert, wie Sie die Entitätsklassen um eine Schnittstelle erweitern, die das Auslösen von Ereignissen beim Eintreten von Validierungsfehlern erlaubt. Diese Schnittstelle implementieren wir nicht direkt in der Klasse, zum Beispiel **Kunde**, sondern machen uns zunutze, dass man Klassen auf mehrere Dateien aufteilen kann. Das hat den Vorteil, dass automatisch erzeugte neue Versionen des Entity Data Models nach Änderungen im Datenmodell sich nicht auf die Klassen mit der Validierung auswirken.

Wir erstellen also eine neue partielle Klasse, welche die für die Validierung notwendigen Elemente aufnimmt und das ist genau das, was wir automatisch auf Basis des Datenmodells der Access-Datenbank erledigen wollen.

Die übrigen Schritte, die im oben genannten Artikel beschrieben werden, beziehen sich auf die Änderungen, die per XAML für die Benutzeroberfläche umgesetzt werden müssen.

## Benötigter Code für die partielle Klasse mit der Validierung

Die partiellen Klassen, welche den Code für die Validierung enthalten, wollen wir wie gesagt in jeweils einer eigenen Klassendatei unterbringen, die in einem Unterverzeichnis namens **Validation** liegen. Eine solche Klasse soll wie folgt aussehen und zunächst den Namespace importieren, der die benötigte Schnittstelle enthält:

```
Imports System.ComponentModel
```

Dann folgt die Klasse, welche die Schnittstelle **IDataErrorInfo** implementiert:

```
Partial Public Class Kunde
    Implements IDataErrorInfo
```

Die Implementierung besteht aus zwei öffentlichen Eigenschaften, von denen uns nur die erste interessiert – die zweite muss aber dennoch implementiert werden. Die erste sieht wie folgt aus:

```
    Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
        Get
            Dim strErrorMessage As String = ""
            Select Case columnName
                Case "Vorname"
                    If (String.IsNullOrEmpty(Vorname)) Then
                        strErrorMessage = "Bitte geben Sie einen Vornamen ein."
                    End If
                ...
            End Select
            Return strErrorMessage
        End Get
    End Property
```

Die Methode erwartet den Namen der zu untersuchenden Eigenschaft als Parameter und unterscheidet diese in einer **Select Case**-Bedingung. Hier prüft sie die Bedingung und trägt dann die Validierungsmeldung in die Variable strErrorMessage ein, die dann auch zurückgegeben wird.

Den Rest der Schnittstelle fügen wir wie folgt an:

```
    Public ReadOnly Property [Error] As String Implements IDataErrorInfo.Error
        Get
            Throw New NotImplementedException()
        End Get
    End Property

End Class
```

Nun benötigen wir eine VBA-Prozedur analog zu der im Artikel **Access zu EDM: Dateien erstellen** (**www.datenbankentwickler.net/219**), die uns diesen Code auf Basis des Datenmodells erstellt. Da wir in den dort vorgestellten Prozeduren und Funktionen schon sehr viele der auch für die Erstellung der Validierungsklassen benötigten Informationen zusammentragen, werden wir diese nochmals erweitern. Das geschieht so, dass die ermittelten Informationen genutzt werden, in weiteren Variablen die Inhalte der Validierungsklassen zusammenzustellen, die wir dann von der Hauptprozedur aus in die Validierungsdateien in einem neuen Verzeichnis namens **Validation** ausgeben.

Die Prozedur **EDMDateienErstellen** erweitern wir dazu zunächst wieder um einen Parameter, diesmal namens **bolValidation**:

```
Public Sub EDMDateienErstellen(bolEineDatei As Boolean, strContextname As String, strUnterverzeichnisEntities _
        As String, Optional bolINotifyPropertyChanged As Boolean, Optional bolValidation As Boolean)
```

Außerdem fügen wir die folgende Variable hinzu:

```
    Dim strValidation() As String
```

Die Variable **strValidation** übergeben wir als neuen Parameter beim Aufruf der Funktion **EntityDataModelZusammenstellen**:

```
    ...
    Set colMappings = EntityDataModelZusammenstellen(strDBSets, strEntities, strEntityNames, strValidation, _
        bolINotifyPropertyChanged, bolValidation)
    ...
```

Die fertigen partiellen Klassen mit dem Validierungscode landen dann in der Array-Variablen **strValidation**.

Weiter unten prüfen wir dann, ob **bolValidation** den Wert **True** hat. In diesem Fall erstellen wir ein neues Unterverzeichnis namens **Validation** und fügen dort in einer Schleife die von der Funktion **EntityDataModelZusammenstellen** gelieferten und im Array **strValidation(i)** gespeicherten Klassen in jeweils eine Datei im soeben erstellten Verzeichnis:

```
    ...
    If bolValidation = True Then
        MakeDirectory strBasispfad & "\Validation"
        For i = LBound(strEntities) To UBound(strEntities)
            TextdateiErstellen strBasispfad & "\Validation\" & strEntityNames(i) & "_Validation.vb", strValidation(i)
        Next i
    End If
End Sub
```

Die größte Arbeit übernimmt wieder die Funktion **EntityDataModelZusammenstellen**. Sie nimmt die neuen Parameter **strValidation** und **bolValidation** entgegen, von denen letztere entscheidet, ob eine Validierung erfolgen soll. Außerdem benötigen wir einige neue Variablen:

```
Public Function EntityDataModelZusammenstellen(strDBSets As String, strEntities() As String, _
        strEntityNames() As String, Optional strValidation As Variant, Optional bolINotifyPropertyChanged As Boolean, _
        Optional bolValidation As Boolean) As Collection
    ...
    Dim strObject As String
    Dim i As Integer
    Dim strValidationBase As String
    Dim strValidationCode() As String
```

Das Ermitteln des Mappings erfolgt wie in den vorherigen Artikeln beschrieben. Beim Durchlaufen eines jeden **clsMapping**-Objekts in einer **For Each**-Schleife wird dann jeweils geprüft, ob die Variable **bolValidation** den Wert **True** hat. Ist das der Fall, wird in der untergeordneten **For Each**-Schleife über alle Felder der Tabelle/Entität für jedes Feld der Validierungscode angelegt.