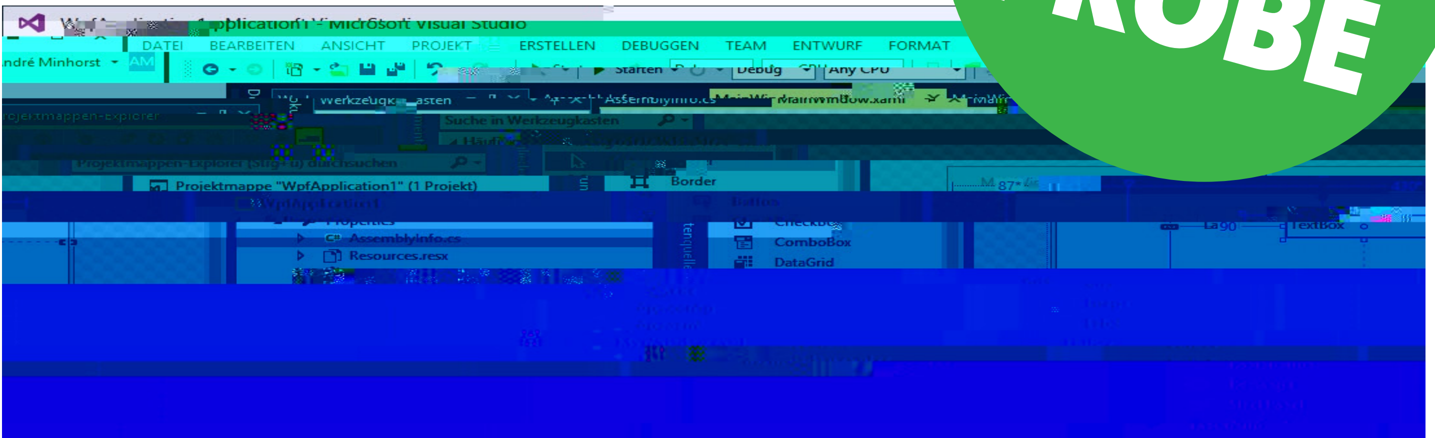
AUSGABE 02/2016

DATENBANK ENTWICKLER

Gratis LESE-PROBE

MAGAZIN FÜR DIE DATENBANKENTWICK
VISUAL STUDIO FÜR DESKTOP, WEB UN



TOP-THEMEN:

WPF-BASICS	Steuerelemente binden	SEITE 3
WPF-CONTROLS	Schaltflächen	SEITE 10
VON ACCESS ZU WPF	Fenster mit 1:n-Daten und Lookup-Feld	SEITE 27
VON ACCESS ZU WPF	Kombinationsfelder mit Daten füllen	SEITE 34
DATENZUGRIFF	SQL Server installieren	SEITE 40

André Minhorst Verlag

WPF-GRUNDLAGEN	Bindung zwischen Steuerelementen	3
BENUTZEROBERFLÄCHE MIT WPF	WPF-Controls: Schaltflächen	10
	WPF-Controls: Kombinationsfelder	15
	WPF-Controls: ToolTips	27
VON ACCESS ZU WPF	Fenster mit 1:n-Daten und Lookup-Feld	34
	Kombinationsfelder mit Daten füllen	40
DATENZUGRIFFSTECHNIK	SQL Server 2014 Express installieren	46
	Von der .mdb-Datei zum SQL Server	52
TIPPS UND TRICKS	Tipps und Tricks zu Fenstern und Steuerelementen	63
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
© 2016 André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

Das Magazin und alle darin enthaltenen Beiträge und Abbildungen sind urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmung und für die Einspeicherung in elektronische Systeme.
Wir weisen darauf hin, dass die verwendeten Bezeichnungen und Markennamen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen. Die im Werk gemachten Angaben erfolgen nach bestem Wissen, jedoch ohne Gewähr. Für mögliche Schäden, die im Zusammenhang mit den Angaben im Werk stehen könnten, wird keine Gewährleistung übernommen.

Bindung zwischen Steuerelementen

Unter VBA waren die Möglichkeiten der Datenbindung überschaubar: Formulare, Berichte und einige Steuerelemente wiesen entsprechende Eigenschaften auf, die an Tabellen, Abfragen oder auch Wertlisten oder Felder geknüpft werden konnten. Unter WPF und C# sieht es ganz anders aus. Hier stehen Objekte im Vordergrund. Dieser Artikel zeigt daher zunächst, wie Sie Elemente wie Steuerelemente an die Eigenschaften anderer Elemente binden.

Einführung

Wenn Sie zuvor mit Access programmiert haben und dieses Magazin lesen, um zu erfahren, wie das, was Sie zuvor mit Access erledigt haben, mit C#/WPF gelingt, wollen Sie vermutlich nur eins: Daten aus Tabellen oder Abfragen in Fenstern und Steuerelementen anzeigen und diese zum Bearbeiten bereitstellen. Beim Thema Datenbindung ist WPF/C# allerdings etwas breiter aufgestellt, da es nicht nur auf Datenquellen wie Tabellen oder Abfragen fixiert ist, sondern primär auf Objekte als Datenlieferant.

Daher räumen wir das Feld etwas weiter auf und beginnen nicht gleich damit, die Daten per Fenster und Steuerelementen bereitzustellen. Zunächst einmal schauen wir uns an, wie Sie verschiedene Elemente an einander binden können. Davon abgesehen haben Sie ja in vorhergehenden Artikeln der Kategorie **Datenzugriffstechnik** bereits einige Beispiele erhalten, wie dies gelingt. Grundsätzlich dient die Bindung aber dazu, nicht nur die Eigenschaften eines Elements an ein anderes Element zu binden, sondern die gebundenen Daten können auch aus Quellen wie einer XML-Datei, Auflistungen oder Datenbanken stammen.

Von Textfeld zu Textfeld

Im ersten Beispiel wollen wir schlicht und einfach, dass ein Textfeld den Inhalt eines anderen Textfeldes anzeigt. Unter Access/VBA haben Sie dazu das **Nach Aktualisierung**-Ereignis genutzt – oder auch das **Bei Änderung**-Ereignis. Das Erste hat beispielsweise auf die Eingabe- oder Tabulatortaste reagiert, Letzteres auf jede einzelne Änderung des Inhalts. Etwas eleganter konnten Sie dies erledigen, wenn Sie das erste Textfeld gleich der Eigenschaft **Steuerelementinhalt** des zweiten Textfeldes zugewiesen haben. Damit war noch nicht einmal mehr Code notwendig, um den Inhalt des ersten Textfeldes nach Änderungen in das zweite Textfeld zu übertragen.

Unter WPF/C# wollen wir uns gleich um die zweite Variante kümmern, also das zweite Textfeld an das erste Textfeld binden. Dazu fügen Sie einem Fenster eines neuen Projekts zwei Textfelder namens **txtFeld1** und **txtFeld2** hinzu:

```
<Window x:Class="Datenbindung.MainWindow" ...
Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Label x:Name="lbl1" ... />
        <Label x:Name="lbl2" ... />
        <TextBox x:Name="txtFeld1" ... />
        <TextBox x:Name="txtFeld2" ... />
    </Grid>
</Window>
```

Damit können Sie nun den Inhalt der beiden Textfelder beliebig ändern – es tut sich nichts. Dies ändern wir, indem wir die Methode **MainWindow** im C#-Modul des Fensters wie folgt um vier Zeilen erweitern:

```
public MainWindow() {
    InitializeComponent();
    Binding binding = new Binding();
    binding.ElementName = "txtFeld1";
    binding.Path = new PropertyPath("Text");
    txtFeld2.SetBinding(TextBox.TextProperty, binding);
}
```

Diese vier Zeilen binden das zweite Feld so an das erste Feld, dass dieses immer den Inhalt des ersten Feldes anzeigt (siehe Bild 1). In diesem Fall werden Änderungen am Text des ersten Textfeldes direkt in das zweite Textfeld übertragen. Änderungen am zweiten Textfeld landen erst nach dem Verlassen im ersten Textfeld – wie Sie dieses Verhalten ändern, erfahren Sie weiter unten.

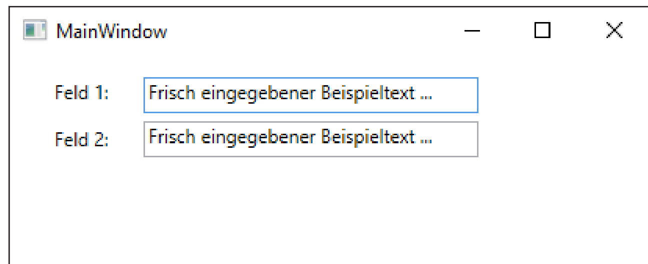


Bild 1: Das untere Textfeld ist an das obere Textfeld gebunden.

Schauen wir uns die vier Zeilen im Detail an. Die erste Zeile erzeugt ein neues Objekt des Typs **Binding**:

```
Binding binding = new Binding();
```

Die zweite Zeile weist der Eigenschaft **ElementName** des **Binding**-Objekts den Namen des ersten Textfelds **txtFeld1** zu:

```
binding.ElementName = "txtFeld1";
```

Im Gegensatz zu Access/VBA, wo sich der Steuerelementinhalt eines Steuerelements immer auf den Steuerelementinhalt des angegebenen Quellsteuerelements bezogen hat, können Sie unter WPF alle Eigenschaften des mit Source angegebenen Objekts als Quellwert nutzen. In diesem Fall ist dies die Eigenschaft **Text**, die wir als neues **PropertyPath**-Objekt der Eigenschaft **Path** des **Binding**-Objekts zuweisen:

```
binding.Path = new PropertyPath("Text");
```

Schließlich müssen wir noch festlegen, welches Steuerelement das neue **Binding**-Objekt nutzen soll und welches die Zieleigenschaft ist. Dies erledigen wir mit der Methode **SetBinding** wie in der folgenden Anweisung. Der erste Parameter ist die Zieleigenschaft, hier **TextProperty**, der zweite das **Binding**-Objekt:

```
txtFeld2.SetBinding(TextBox.TextProperty, binding);
```

Die Binding-Klasse und die Dependency Property

Die hier verwendete **Binding**-Klasse legt also fest, wie die Bindung zwischen einer Komponente wie etwa einem Steuerelement und der Datenquelle, hier die Eigenschaft eines Steuerelements, erfolgt. Dabei haben wir hier bereits die folgenden Eigenschaften genutzt:

- **ElementName**: Name des als Datenquelle verwendeten Steuerelements
- **Path**: Eigenschaft, aus der die Daten stammen

Mit dem ersten Parameter der **SetBinding**-Methode haben wir oberflächlich betrachtet einfach die Eigenschaft des Objekts **txtFeld2** festgelegt, das gebunden werden soll. Aber warum geben wir dann nicht den Namen der Eigenschaft, also **Text**, sondern **TextBox.TextProperty** als Parameter an? Weil dieser Parameter eine so genannte **Dependency Property**, zu Deutsch Abhängigkeitseigenschaft, erwartet.

Diese Dependency Properties haben die in Zusammenhang mit dem Binding interessante Eigenschaft, Automatismen für die Benachrichtigung über Änderungen bereitzustellen – in diesem Fall für den Zweck, Änderungen am Quellwert an das Ziel zu übergeben und nach Wunsch auch umgekehrt. Für Access-Nutzer ist das in etwa mit der Synchronisierung von Haupt- und Unterformular über die beiden Eigenschaften **Verknüpfen von** und **Verknüpfen nach** vergleichbar – trägt man dort die Feldnamen der verknüpften Felder der beiden Tabellen aus Haupt- und Unterformular ein, zeigt das Unterformular automatisch die zum aktuellen Datensatz im Hauptformular passenden Datensätze des Unterformulars an.

Weitere Eigenschaften der Binding-Klasse

Die **Binding**-Klasse liefert noch weitere interessante Eigenschaften:

- **Converter**: Gibt einen möglichen Konverter für die zu bindenden Daten an.
- **FallbackValue**: Wert, den das Ziel annehmen soll, falls das gebundene Objekt oder die Eigenschaft den Wert **Null** liefert.
- **IsAsync**: Legt mit **True** fest, dass die Bindung asynchron hergestellt wird.
- **Mode**: Legt fest, wie die Bindung hergestellt wird. Mögliche Werte: **Default** (übernimmt den Standardwert für die jeweilige Dependency Property), **OneTime** (nur einmalige Aktualisierung), **OneWay** (ändert nur von der Quelle zum Ziel, aber nicht umgekehrt), **OneWayToSource** (ändert nur vom Ziel zur Quelle), **TwoWay** (ändert von der Quelle zum Ziel und umgekehrt)
- **Source**: Quellobjekt der Datenbindung
- **TargetNullValue**: Wert, falls das gebundene Element keinen Wert liefert, also nicht vorhanden ist (siehe weiter unten)
- **UpdateSourceTrigger**: Gibt an, wie die Quelle in Abhängigkeit vom Inhalt des Ziels geändert werden soll. In unserem Beispiel ändert sich der Inhalt der Quelle, sobald das geänderte Zieltextfeld den Fokus verliert. Dies entspricht der Einstellung **LostFocus**. Wenn Sie den Wert **PropertyChanged** verwenden, wird auch die Quelle nach der Eingabe eines jeden Zeichens in das Zieltextfeld geändert (siehe Beispiel 5).

Textfelder binden per XAML

Das Ganze lässt sich per XAML noch einfacher abbilden. Hierzu fügen wir dem Formular zwei weitere Textfelder namens **txtFeld3** und **txtFeld4** hinzu. Für das zweite ändern Sie das Attribut **Text**, das ja zuvor eine Zeichenkette enthielt, wie folgt (siehe Beispiel 2 im Beispielprojekt):

```
<TextBox x:Name="txtFeld4" ...
    Text="{Binding ElementName=txtFeld3, Path=Text}"/>
```

Was bedeutet **{Binding ElementName=txtFeld3, Path=Text}** nun? **Binding** entspricht dem Objekt des Typs **Binding**, das wir oben per Code mit der **New**-Anweisung erzeugen mussten. Dahinter geben wir den Namen des Quell-Elements an, also **txtFeld3**. Schließlich folgt der Pfad zur zu bindenden Eigenschaft, hier also **Text** (bei einer ComboBox könnte dies beispielsweise auch **SelectedItem.Content** lauten). Es gibt noch eine alternative Schreibweise:

```
<TextBox x:Name="txtFeld4" ...
    Text="{Binding Text, ElementName=txtFeld3}" />
```

Hier geben wir mit **Text** gleich den Namen der Eigenschaft des Quellobjekts an, an das wir die Eigenschaft **Text** des Zielobjekts binden wollen. Das Quell-Element weisen wir schließlich über das Name-Wert-Paar **ElementName=txtFeld3** zu. Der Pfad zur Eigenschaft muss hier zwingend als erstes Argument hinter dem Schlüsselwort **Binding** angegeben werden.

Da wir das Binding der Eigenschaft **Text** des Textfeldes **txtFeld4** zugewiesen haben, haben wir alle Informationen zusammen, die wir weiter oben in vier Zeilen C#-Code programmiert haben. Das Ergebnis überzeugt, denn die Textfelder werden wie die C#-Version synchronisiert, wenn Sie den Inhalt von **txtFeld3** ändern. Eine übersichtlichere Schreibweise hierfür ohne geschweifte Klammern sieht so aus (siehe Beispiel 3 im Beispielprojekt):

```
<TextBox x:Name="txtFeld4" ...>
    <TextBox.Text>
        <Binding ElementName="txtFeld3" Path="Text" />
    </TextBox.Text>
</TextBox>
```

Standardwerte bei Bindungen

Es gibt zwei Attribute, mit denen wir eine Art Standardwert simulieren können. Warum gleich zwei? Weil es bei einer Bindung verschiedene Fälle gibt, die dafür sorgen, dass kein

WPF-Controls: Schaltflächen

In den ersten Ausgaben des DATENBANKENTWICKLERs haben Sie bereits einige Steuerelemente kurz kennen gelernt. In den folgenden Ausgaben wird sich dies ändern. Im vorliegenden Artikel schauen wir uns die verschiedenen Schaltflächen an, die WPF uns zur Verfügung stellt. Dabei lernen Sie die wichtigsten Unterschiede zu den von Access bekannten Schaltflächen kennen.

Allgemeines zu WPF-Steuerelementen

WPF-Fenster und auch die enthaltenen Steuerelemente wie zum Beispiel die Schaltflächen passen sich standardmäßig an das jeweils ausgewählte Windows-Design an. Dieses stellen Sie in der Systemsteuerung unter **Systemsteuerung|Darstellung und Anpassung|Anpassung** ein. In Bild 1 sehen Sie beispielsweise ein Fenster mit einer einfachen Schaltfläche unter drei verschiedenen Windows-Designs. Davon abgesehen bietet WPF unendliche Möglichkeiten zum Gestalten der Benutzeroberfläche – vor allem, wenn Sie zuvor mit Access gearbeitet haben. Die Gestaltungsmöglichkeiten unter Access haben sich zwar mit den letzten Versionen leicht verbessert, aber mit den Möglichkeiten von WPF ist dies nicht vergleichbar. Die Gestaltung spielt jedoch aktuell eine untergeordnete Rolle, sodass wir viel später darauf eingehen werden.

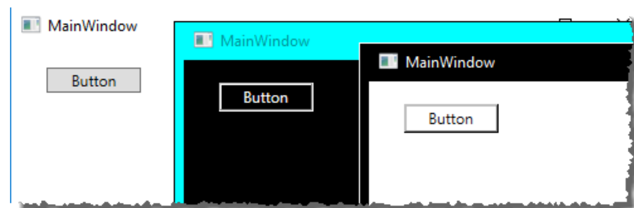


Bild 1: WPF-Fenster mit einer Schaltfläche in drei verschiedenen Designs

Alle Steuerelemente?

Die Toolbox, die zusammen mit dem Entwurf eines WPF-Fensters eingeblendet wird, zeigt im oberen Bereich die häufig verwendeten WPF-Steuerelemente an, im unteren Bereich können Sie dann alle WPF-Steuerelemente einblenden (siehe Bild 2).

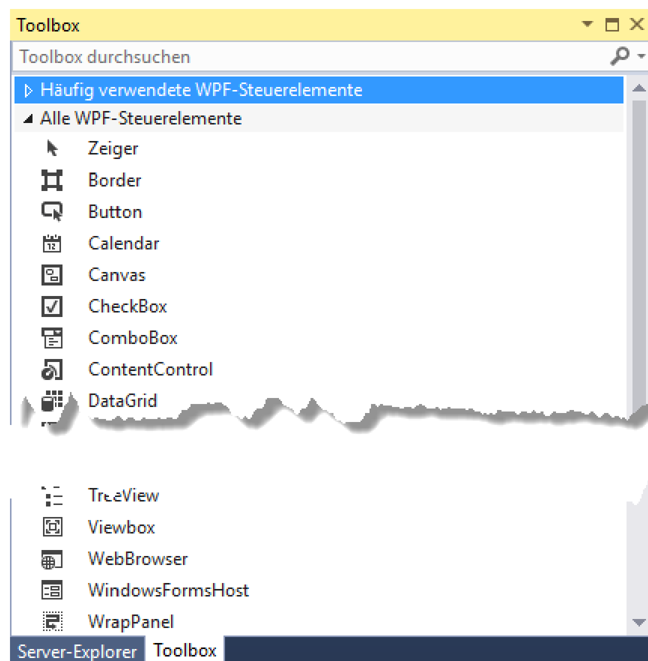


Bild 2: Toolbox mit »allen« WPF-Steuerelementen

Aber sind dies nun wirklich alle Steuerelemente? Nein: Es gibt noch einige weitere Steuerelemente wie beispielsweise den **RepeatButton** oder den **ToggleButton**, die in der Liste gar nicht erscheinen. Wie aber fügen Sie diese zu einem WPF-Fenster hinzu? Ganz einfach: Sie legen das Steuerelement über den XAML-Editor an. Geben Sie das Kleiner-Zeichen (**<**) gefolgt vom ersten Buchstaben des gesuchten Elements ein, erhalten Sie direkt eine Liste aller in Frage kommenden Steuerelemente (siehe Bild 3).

Das Button-Steuerelement

Wenn Sie von Access kommen, wollen Sie sicher erstmal das Pendant zur dortigen Schaltfläche kennen lernen. Dabei han-

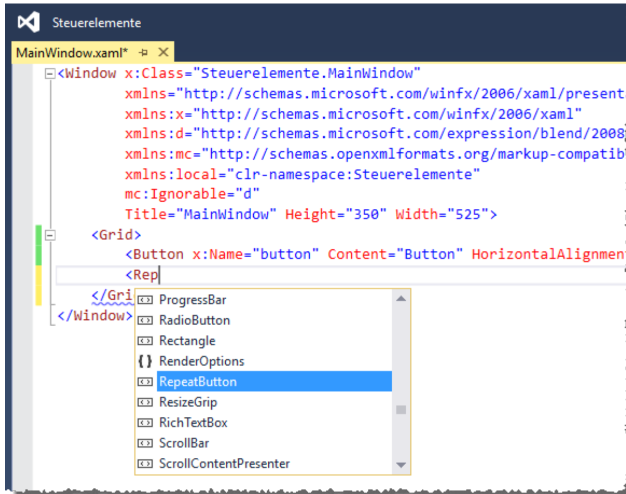


Bild 3: Anlegen nicht in der Toolbox verfügbarer Steuerelemente

delt es sich um das Button-Element, dass Sie sowohl über die Toolbox als auch direkt per XAML hinzufügen können. Da Sie per XAML immer alle Attribute manuell setzen müssen, macht es Sinn, dass Button-Element aus der Toolbox in das Fenster zu ziehen und dann seine Attribute im XAML-Editor anzupassen (oder auch per Eigenschaftsfenster, wenn Sie mögen – wir wollen hier allerdings beim XAML-Editor bleiben). Das Steuerelement wird standardmäßig mit Werten für die Attribute **Name**, **Content**, **HorizontalAlignment**, **Margin**, **VerticalAlignment** und **Width** ausgestattet. **Name** enthält den Steuerelementnamen, **Content** den angezeigten Text.

Entspricht **Content** also der Eigenschaft **Text** der Schaltfläche von Access? Mitnichten: Die Anzeige eines Textes ist nur die einfachste Variante, die Schaltfläche zu füllen. Sie können dort auch wesentlich komplexere Inhalte anzeigen – beispielsweise Icon und Text nebeneinander.

Beim Klicken

Als Nächstes schauen wir uns an, wie die WPF-Schaltfläche ein Ereignis auslöst. Unter Access haben wir die Ereigniseigenschaft **Beim Klicken** mit dem Wert **[Ereignisprozedur]** gefüllt und per Klick die passende Ereignisprozedur angelegt, die wir nur noch mit den gewünschten Anweisungen füllen mussten. Hier läuft das ganz ähnlich. Sie fügen dem **Button**-Element das Attribut **click** hinzu.

Spätestens nach Eingabe des Gleichheitszeichens erscheint ein kleines Popup und bietet den Wert **<Neuer Ereignishandler>** zur Auswahl an (siehe Bild 4).

Klicken Sie darauf, fügt Visual Studio den Wert **<Buttonname>_Click** als Wert des Attributs ein und legt automatisch die folgende Methode an, wobei **<Buttonname>** als Platzhalter für den aktuellen Namen des Buttons zu verstehen ist:

```
private void <Buttonname>_Click(object sender,
                                RoutedEventArgs e) {
    ...
}
```

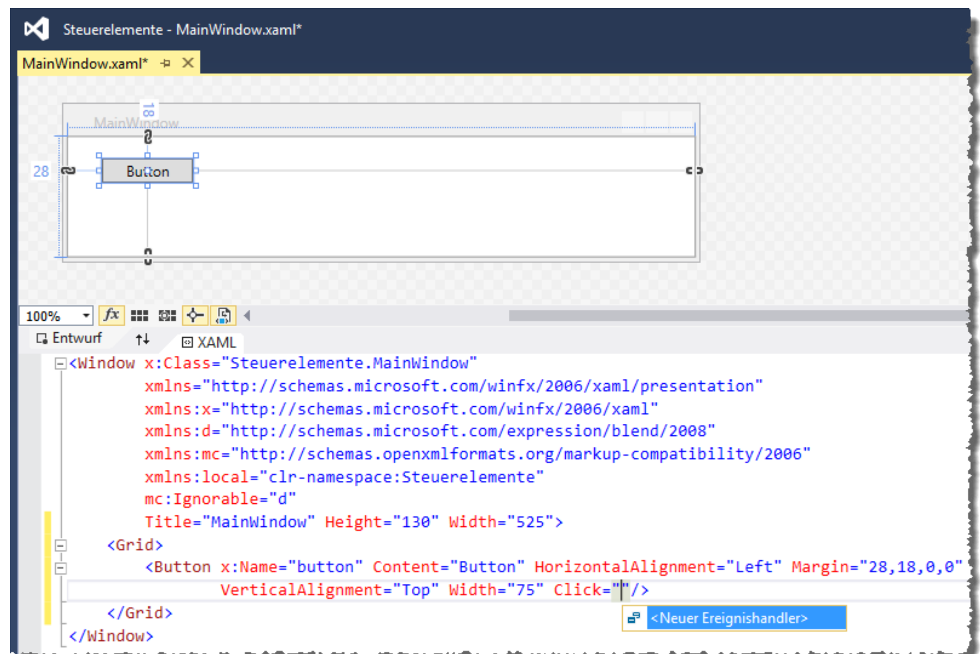


Bild 4: Hinzufügen des **Click**-Ereignisses

Es fällt auf, dass hier eine ähnliche Syntax wie bei Access/VBA verwendet wird und der Methodenname aus dem Namen des Steuerelements, einem Unterstrich und dem Ereignisbezeichner besteht.

Das ist in diesem Fall allerdings Zufall, weil Visual Studio den Namen bei Verwendung dieses Shortcuts so generiert. Sie können auch einen benutzerdefinierten Namen vergeben, wobei Sie allerdings das Eigenschaftsfenster bemühen müssen.

Dort wechseln Sie über das Blitz-Symbol rechts oben zur Liste der Ereignisse und tragen für das **Click**-Ereignis den gewünschten Namen für den Ereignishandler ein (siehe Bild 5). Nun klicken Sie doppelt auf die Eigenschaft und erhalten in der C#-Klasse zu diesem Fenster die folgende Methode:

```
private void AlternativerMethodenname(object sender,
                                      RoutedEventArgs e) {

}
```

Im Gegensatz zu Access/VBA können Sie den gleichen Ereignishandler auch mehreren Ereignissen verschiedener Steuerelemente zuweisen. Wir löschen die bisher angelegten **Button**-Steuerelemente und legen ein neues namens **btnMeldung** an. Dann fügen Sie das **Click**-Ereignis hinzu und ergänzen den Ereignishandler um eine **MessageBox.Show**-Anweisung (siehe Bild 6):

```
private void btnMeldung_Click(object sender,
        RoutedEventArgs e) {
    MessageBox.Show("Button 'button' angeklickt.",
        "WPF-Beispiele");
}
```

Apropos Klick: Das Ereignis fängt nicht nur Mausklicks mit der linken Maustaste ab (übrigens erst beim Loslassen der Maustaste), sondern auch das Betätigen der Eingabetaste und der Leertaste, wenn das **Button**-Element den Fokus hat.

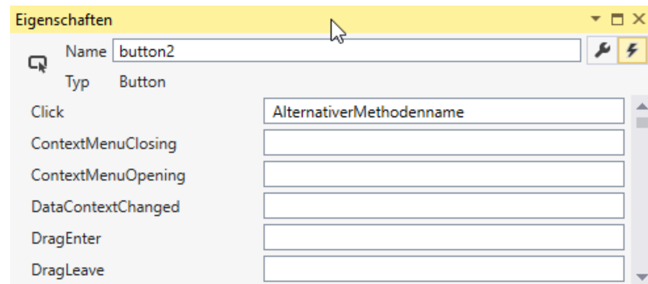


Bild 5: Ereignishandler mit benutzerdefiniertem Namen

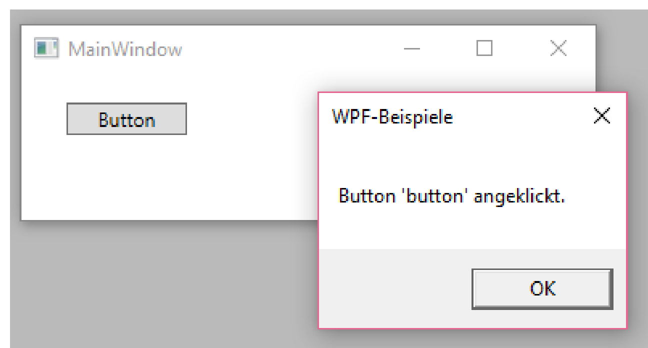


Bild 6: Beispiel für eine Schaltfläche mit Meldungsfenster

Standardschaltfläche und Abbrechen-Schaltfläche

Unter Access konnten Sie für eine Schaltfläche die beiden Eigenschaften **Standard** und **Abbrechen** jeweils auf den Wert **Ja** einstellen.

Haben Sie für eine Schaltfläche **Standard** auf **Ja** eingestellt, wurde die dafür definierte Ereignisprozedur auch beim Betätigen der Eingabetaste ausgelöst, haben Sie **Abbrechen** auf **Ja** eingestellt, feuerte das Ereignis beim Betätigen der **Escape**-Taste.

Dies gelingt auch unter WPF. Wir fügen dem Fenster zwei Schaltflächen hinzu. Das erste heißt **btnStandard** und erhält für das Attribut **IsDefault** den Wert **True** sowie den Wert **btnStandard_Click** für das Attribut **Click**:

```
<Button x:Name="btnStandard" Content="Standard" ...
Click="btnStandard_Click" IsDefault="True"/>
```

Der Ereignishandler sieht so aus und zeigt eine Meldung an, dass die Standard-Methode ausgelöst wurde:

WPF-Controls: Kombinationsfelder

In den ersten Ausgaben des DATENBANKENTWICKLERs haben Sie bereits einige Steuerelemente kurz kennen gelernt. In den folgenden Ausgaben wird sich dies ändern. Im vorliegenden Artikel schauen wir uns das ComboBox-Steuerelement an, dass Sie als Access-Entwickler ja vielleicht aus dem Eff-Eff kennen. Unter WPF läuft aber gewöhnlich alles etwas anders, sodass wir hier mit den Grundlagen einsteigen und uns dann Schritt für Schritt Themen wir der Datenbindung et cetera widmen. Außerdem dürfen wir vornewegnehmen, dass sicher hier einige Möglichkeiten mehr ergeben ...

Wenn Sie ein **ComboBox**-Steuerelement zu einem Fenster hinzufügen und das Debugging starten, erhalten Sie ein einfaches Kombinationsfeld, das beim Aufklappen keine Werte anzeigt und in das Sie auch keinen Text einfügen können. Das ist der erste Unterschied im Vergleich zu einem Access-Kombinationsfeld – dort können Sie zumindest gleich einen Wert eintippen.

Eine recht einfache Definition eines solchen **ComboBox**-Steuerelements sieht so aus:

```
<ComboBox x:Name="comboBox"
HorizontalAlignment="Left" Margin="127,10,0,0"
VerticalAlignment="Top" Width="120"/>
```

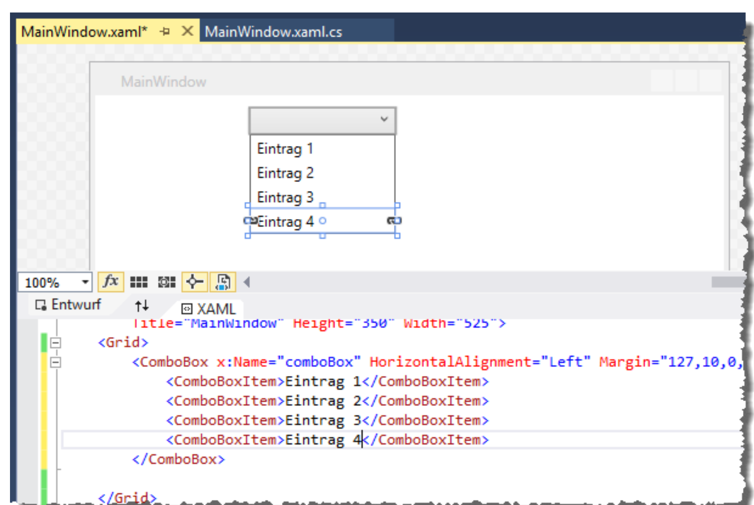


Bild 1: WPF-Fenster mit einer Schaltfläche in drei verschiedenen Designs

ComboBox mit statischen Werten

Soll die **ComboBox** beim Öffnen des Fensters immer die gleichen Werte anzeigen, können Sie diese ganz einfach durch eine entsprechende XAML-Definition hinzufügen. Dies ähnelt dann dem Anlegen von Einträgen in einem HTML-Select-Element, allerdings verwenden Sie für jeden neuen Eintrag ein **ComboBoxItem**-Element, das Sie unterhalt des **ComboBox**-Elements anlegen:

```
<ComboBox x:Name="comboBox" ... >
    <ComboBoxItem>Eintrag 1</ComboBoxItem>
    <ComboBoxItem>Eintrag 2</ComboBoxItem>
    <ComboBoxItem>Eintrag 3</ComboBoxItem>
    <ComboBoxItem>Eintrag 4</ComboBoxItem>
</ComboBox>
```

Während Sie diese Einträge unterhalb des **ComboBox**-Elements hinzufügen, zeigt der XAML-Editor direkt die Einträge in der Vorschau an (siehe Bild 1).

Im Betrieb erscheint dieses Kombinationsfeld nun wie in Bild 2. Beim Aufklappen zeigt es die vier Einträge an, die Auswahl eines Eintrags trägt diesen oben in das Steuerelement ein. Nun wollen wir dies noch erweitern, und zwar erstens durch die Eingabe von Texten durch den Benutzer.

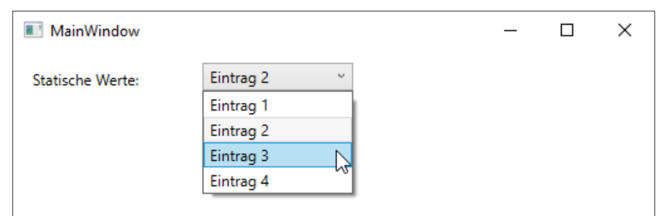


Bild 2: Statisches Kombinationsfeld mit vier Einträgen

ComboBox bedienen

Natürlich wissen Sie, dass Sie eine ComboBox mit einem Mausklick auf den Pfeil nach unten öffnen und dass Sie dort Text eingeben und Einträge auswählen können. Aber haben Sie auch schon einmal nur mit der Tastatur auf eine ComboBox zugegriffen? Sie können diese nämlich auch durch Betätigen der Taste **F4** öffnen oder schließen. Auch die Tastenkombinationen **Alt + Nach oben** oder **Alt + Nach unten** öffnen und schließen die Liste des **ComboBox**-Steuerelements. Schließlich blättern Sie auch bei nicht ausgeklappter Liste mit den Tasten **Nach oben** und **Nach unten** durch die Einträge, wobei Sie **Nach oben** vom obersten Element und **Nach unten** vom untersten Element aus keine Auswirkungen zeigen.

Eingabe von Werten aktivieren

Um das Eingabefeld des Kombinationsfeldes zu aktivieren, fügen Sie einfach nur das Attribut **IsEditable** zum **ComboBox**-Element hinzu und stellen es auf den Wert **True** ein:

```
<ComboBox x:Name="cboMitEingabe" ... IsEditable="True">
    <ComboBoxItem Content="Eintrag 1"/>
    ...
</ComboBox>
```

Damit können Sie nun Werte wie in Bild 3 eingeben. Allerdings fügt dies den Wert noch nicht zur Auswahlliste hinzu.

Wenn Sie den Anfangsbuchstaben eines der bereits in der Liste enthaltenen Einträge eingeben, wird dieser übrigens gleich als Eintrag vorgeschlagen (siehe Bild 4) – wie auch unter Access. Nach der Eingabe eines Wertes in das Textfeld prüft die ComboBox, ob dieser Wert bereits in der Liste enthalten ist. Falls ja, wird dieser Eintrag als markiert gekennzeichnet – später schauen wir uns an, wie wir dies nutzen

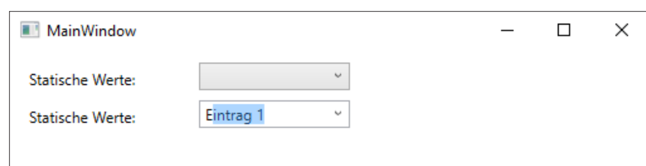


Bild 4: Autoergänzung standardmäßig

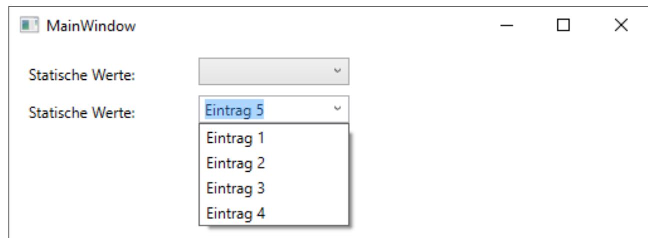


Bild 3: Hinzufügen eines eigenen Textes

können. Ist der Wert nicht vorhanden, hat die Liste keinen selektierten Eintrag. In beiden Fällen wird der Wert jedoch in die Eigenschaft **Text** des **ComboBox**-Steuerelements eingetragen.

IsEditable und IsReadOnly

Selbst, wenn Sie das Attribut **IsEditable** auf **True** eingestellt haben, können Sie die Eingabe von Texten in das Textfeld des Kombinationsfeldes unterbinden. Dazu stellen Sie die Eigenschaft **IsReadOnly** ebenfalls auf **True** ein. Der Unterschied zwischen **IsEditable=False** sowie **IsEditable=True** und **IsReadOnly=True** ist, dass der Hintergrund des Eingabefeldes nicht grau, sondern weiß dargestellt wird.

```
<ComboBox x:Name="cboMitEingabe" ... IsEditable="True"
        IsReadOnly="True">
    ...
</ComboBox>
```

Ein Element vorauswählen per XAML

Unter Access/VBA haben Sie etwa die folgende Zeile verwendet, um beispielsweise den ersten Eintrag eines Kombinationsfeldes etwa beim Öffnen des Formulars zu markieren:

```
Me!cboEintraege = Me!cboEintraege.ItemData(0)
```

Unter WPF können Sie dieses Element gleich in der Definition markieren, sofern diese überhaupt per Definition festgelegt und nicht dynamisch mit einer anderen Datenherkunft gefüllt wird. Dazu fügen Sie dem zu markierenden Eintrag das Attribut **IsSelected** mit dem Wert **True** hinzu:

```
<ComboBox x:Name="cboMitStandardwert" ...
        IsEditable="True" IsReadOnly="True">
```

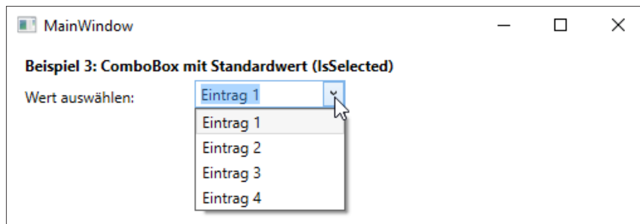


Bild 5: ComboBox mit voreingestelltem Wert

```
    <ComboBoxItem Content="Eintrag 1" IsSelected="True"/>
    <ComboBoxItem Content="Eintrag 2"/>
    <ComboBoxItem Content="Eintrag 3"/>
    <ComboBoxItem Content="Eintrag 4"/>
</ComboBox>
```

Das Ergebnis finden Sie in Bild 5. Der Text des gewählten Eintrags wird nicht nur im Eingabefeld angezeigt, sondern beim Ausklappen der Liste auch optisch hervorgehoben (siehe Beispiel 3a des Beispielprojekts).

Ein Element vorauswählen per C#

Natürlich können Sie ein Element auch per Code auswählen wie unter Access/VBA. Dazu nutzen wir die Ereignismethode, die durch das Ereignis **Loaded** des Fensters ausgelöst wird. Damit diese bekannt ist, fügen wir die Eigenschaft **Loaded** zum **Window**-Element hinzu (dazu den Attributnamen eingeben, Tabulator-Taste drücken und den dann erscheinenden Eintrag **<Neuer Ereignishandler>** auswählen, um die Methode gleich automatisch anzulegen):

```
<Window x:Class="ComboBoxBeispiele.MainWindow" ...
Title="MainWindow" ... Loaded="Window_Loaded">
```

Diese Methode füllen Sie im C#-Modul zum Fenster wie folgt mit einer einzigen Anweisung auf:

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    cboMitStandardwertCSharp.SelectedIndex = 0;
}
```

Diese stellt den Index des aktivierten Eintrags des **ComboBox**-Steuerelements auf den Wert **0** ein, was dem ersten Eintrag entspricht (siehe Beispiel 3b des Beispielprojekts)

Vorauswahl mit Loaded-Ereignis der ComboBox

Das **ComboBox**-Steuerelement besitzt selbst ein **Loaded**-Attribut. Dieses wird beim Laden des Steuerelements ausgelöst. Um die **Loaded**-Methode des **Window**-Elements nicht zu überfrachten, können Sie auch die des **ComboBox**-Elements nutzen, um dessen Standardwert zu definieren:

```
<ComboBox x:Name="cboMitStandardwertLoaded" ...
    Loaded="cboMitStandardwertLoaded_Loaded">
```

Die **Loaded**-Methode enthält prinzipiell die gleiche Anweisung

Ereignisse beim Auf- und Zuklappen

Im Gegensatz zu Access, wo Sie zum Beispiel auf die Auswahl eines Elements reagieren konnten, gibt es bei der WPF-ComboBox zwei Ereignisse, die durch das Auf- und Zuklappen der Liste ausgelöst werden:

- **DropDownOpened**: Wird beim Aufklappen der Liste ausgelöst.
- **DropDownClosed**: Wird beim Zuklappen der Liste ausgelöst.

Außerdem gibt es mit **IsDropDownOpen** noch eine Eigenschaft, mit der Sie auslesen können, ob die Liste gerade auf- oder zugeklappt ist. Um diese zu testen, fügen wir dem **ComboBox**-Element die Namen der Methoden zu, die beim Eintreten der Ereignisse **DropDownClosed** und **DropDownOpened** ausgeführt werden sollen. Das **ComboBox**-Element sieht dann so aus:

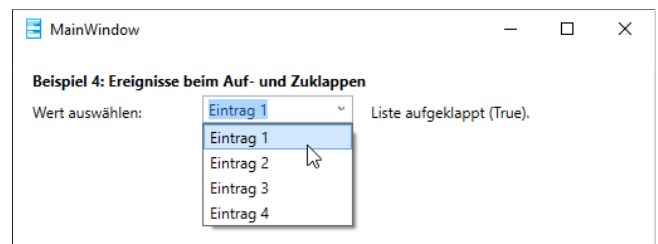


Bild 6: Auslösen von Ereignissen beim Auf- und Zuklappen und Ermitteln des Status

```
<ComboBox x:Name="cboMitEreignissen" ...
        DropDownClosed="cboMitEreignissen_DropDownClosed"
        DropDownOpened="cboMitEreignissen_DropDownOpened">
    <ComboBoxItem Content="Eintrag 1" IsSelected="True"/>
    ...
</ComboBox>
```

Dann wechseln Sie zur Datei mit dem C#-Code für das Fenster (hier **MainWindow.xaml.cs**) und ergänzen die dort bereits angelegten Methoden mit den folgenden Anweisungen:

```
private void cboMitEreignissen_DropDownClosed(...) {
    lblAufZu.Content = "Liste zugeklappt ("
        + cboMitEreignissen.IsDropDownOpen + ").";
}

private void cboMitEreignissen_DropDownOpened(...) {
    lblAufZu.Content = "Liste aufgeklappt ("
        + cboMitEreignissen.IsDropDownOpen + ").";
}
```

Dies fügt einem **Label**-Steuerelement namens **lblAufZu**, das wir rechts von der ComboBox platzieren, den Wert **Liste aufgeklappt (True)** oder **Liste zugeklappt (False)** hinzu. Dabei haben wir die Werte **True** und **False** der Eigenschaft **IsDropDownOpen** entnommen (siehe Bild 6).

Einträge hinzufügen per C#

Wenn Sie einer ComboBox Einträge zur Laufzeit hinzufügen wollen, erledigen Sie das wieder über die **Loaded**-Methode des **ComboBox**-Elements, die Sie wie folgt zum Element hinzufügen:

```
<ComboBox x:Name="cboEintragPerCSharp" ...
    Loaded="cboEintragPerCSharp_Loaded"/>
```

Die dadurch ausgelöste Methode sieht so aus:

```
private void cboEintragPerCSharp_Loaded(...) {
    cboEintragPerCSharp.Items.Add("Eintrag 1");
    cboEintragPerCSharp.Items.Add("Eintrag 2");
    ...
}
```

Hier greifen wir direkt über den Namen auf das Steuerelement zu. Dabei wird das auslösende Objekt des Ereignisses doch über den Parameter **sender** an die Methode übergeben. Diesen können wir mit einer Objektvariablen des entsprechenden Typs referenzieren und dann dessen **Items**-Auflistung mit der **Add**-Methode füllen:

```
private void cboEintragPerCSharp_Loaded(object sender,
        RoutedEventArgs e) {
    ComboBox cbo = sender as ComboBox;
    cbo.Items.Add("Eintrag 1");
    cbo.Items.Add("Eintrag 2");
    ..
}
```

Dies hat zumindest den Vorteil, dass Sie den Code leicht in eine Methode übertragen können. Noch mehr Vorteile erhalten Sie, wenn Sie möglicherweise mehrere Steuerelemente mit diesen Daten füllen wollen. Dann können Sie der Methode einen allgemeinen Namen geben und diese dem Attribut **Loaded** der betroffenen Steuerelemente zuweisen.

Einträge aus einem Array

Die Einträge für ein Kombinationsfeld können Sie auch aus einem Array ermitteln. In diesem Beispiel erledigen wir dies wieder beim Laden des Kombinationsfeldes und legen dazu das Attribut **Loaded** an:

```
<ComboBox x:Name="cboEintraegeAusArray"
Loaded="cboEintraegeAusArray_Loaded"/>
```

Die passende Methode stellt dann zunächst ein Array in der Variablen **eintraege** zusammen, das aus vier **String**-Objekten besteht. Danach legt sie diese Variable als Wert der Eigenschaft **ItemsSource** des **ComboBox**-Steuerelements fest und stellt **SelectedIndex** wieder auf **0** ein, damit das erste Element gleich angezeigt wird:

WPF-Controls: ToolTips

Unter Access/VBA fristeten die ToolTips ein Schattendasein. Sie konnten dafür lediglich einen Text angeben, der beim Überfahren des jeweiligen Steuerelements eingeblendet wurde und dann wieder verschwand. Es gab keinerlei Möglichkeiten, das Aussehen des ToolTips zu beeinflussen. Unter WPF sieht dies ganz anders aus: Sie können nicht nur das Aussehen und den Inhalt vielfältig gestalten, sondern auch noch bestimmen, wann und wie der ToolTip ein- und ausgeblendet wird.

ToolTip für Button-Steuerelemente

Von Access/VBA haben Sie mit der Eigenschaft **SteuerelementTip-Text** (unter VBA **ControlTipText**) festgelegt, welcher Text angezeigt werden soll, wenn der Benutzer mit dem Mauszeiger über dem Steuerelement verweilt. Eine solche Eigenschaft gibt es natürlich auch unter WPF. Am einfachsten fügen Sie einen solchen Hinweistext hinzu, indem Sie einfach das entsprechende Attribut des betroffenen Steuerelements verwenden – in diesem Fall für ein **Button**-Element:

```
<Button x:Name="btnToolTip" Content="Button mit ToolTip"
... ToolTip="Ich bin ein Tooltip für einen Button!"/>
```

Das Ergebnis sieht dann etwa wie in Bild 1 aus.

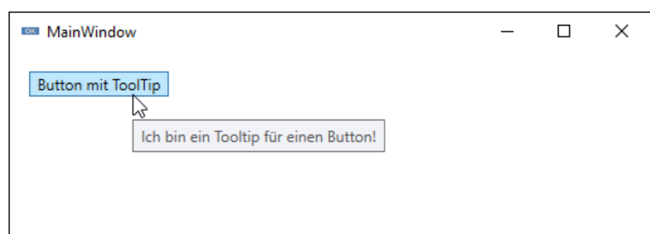


Bild 1: Schaltfläche mit einem einfachen **ToolTip**-Text

Das **ToolTip**-Element ist tatsächlich ein echtes Steuerelement, das Sie mit beliebigen Inhalten füllen können. Wie das aussehen kann, sehen Sie in Bild 2.

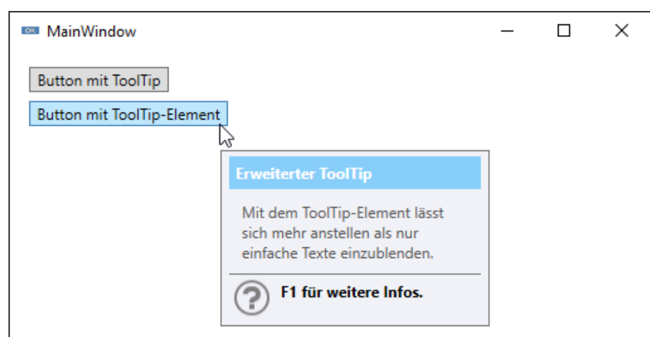


Bild 2: Erweiterter **ToolTip**-Text

Der notwendige Code ist in Listing 1 abgebildet. Hier ist zu beachten, dass wir nicht mehr das Attribut **ToolTip** verwenden. Stattdessen fügen wir dem **Button**-Element ein schließendes **Button**-Element hinzu (**</Button>**) und fügen einige weitere Elemente innerhalb von **<Button>** und **</Button>** ein.

Dabei handelt es sich zunächst um das **Button.ToolTip**-Element, das ohne eigene Attribute kommt, sondern nur den Rahmen für die anzuzeigenden Inhalte liefert. Darin bringen wir ein **StackPanel**-Element unter. Dieses haben Sie bisher in diesem Magazin noch nicht kennen gelernt, daher in aller Kürze: Es handelt sich dabei wie beim Grid um ein Element zur einfachen Anordnung von Inhalten in vertikaler oder horizontaler Ausrichtung.

Das erste **StackPanel**-Element enthält einige untereinander darzustellende Elemente: Ein **Label**, einen **TextBlock**, ein Linie (**Line**) und ein weiteres **StackPanel**. Dieses soll seine Unterelemente nun horizontal ausrichten, also nebeneinander – und zwar ein **Image**-Element und ein **Label**-Element.

Damit das dem **Image**-Element zugewiesene Bild angezeigt wird, müssen Sie dieses dem Projekt hinzufügen. Im Beispielprojekt ist dieses bereits enthalten.

```
<Button x:Name="btnToolTipElement" Content="Button mit ToolTip-Element" ...>
    <Button.ToolTip>
        <StackPanel>
            <Label Content="Erweiterter ToolTip" FontWeight="Bold" Background="LightSkyBlue" Foreground="White" />
            <TextBlock Padding="10" TextWrapping="WrapWithOverflow" Width="200">
               Mit dem ToolTip-Element lässt sich mehr anstellen als nur einfache Texte einzublenden.
            </TextBlock>
            <Line Stroke="Black" StrokeThickness="1" X2="200"/>
            <StackPanel Orientation="Horizontal">
                <Image Margin="2" Source="question_32.png" />
                <Label FontWeight="Bold" Content="F1 für weitere Infos." />
            </StackPanel>
        </StackPanel>
    </Button.ToolTip>
</Button>
```

Listing 1: Einen ausgefeilten **ToolTip** zu einem **Button**-Element hinzufügen

Zeitvorgaben für Einblenden, Ausblenden et cetera

Sie können mit drei Eigenschaften das Verhalten festlegen, wie schnell ToolTip-Texte mit weiteren Informationen ein- und ausgeblendet werden.

Diese sind Eigenschaften des Objekts **ToolTipService**:

- **InitialShowDelay**: Gibt die Anzahl der Millisekunden an, bis der ToolTip nach Positionieren des Mauszeigers auf dem Steuerelement eingeblendet wird.
- **ShowDuration**: Gibt die Anzahl der Millisekunden an, bis der ToolTip wieder ausgeblendet wird.
- **BetweenShowDelay**: Betrifft Fenster mit mehreren ToolTips. Wenn Sie für ein Steuerelement **ToolTipService.BetweenShowDelay** etwa auf **3.000** einstellen, dann wird innerhalb dieser drei Sekunden jeder **ToolTip** anderer Steuerelemente unmittelbar nach dem Überfahren mit der Maus angezeigt – egal, welchen Wert die Eigenschaft **InitialShowDelay** aufweist.

Die folgenden drei Schaltflächen zeigen, wie Sie die zeitbezogenen Attribute festlegen. Beachten Sie, dass die Attribute nicht direkt als Attribute des übergeordneten Elements, sondern als Attribute des Elements **ToolTipService** angegeben werden:

```
<Button x:Name="btnVerzoegerung" Content="Verzögerung beim 
Anzeigen" ... ToolTip="ToolTip mit verzögerter Anzeige 
(drei Sekunden)" ToolTipService.InitialShowDelay="3000" />

<Button x:Name="btnAusblenden" Content="Ausblenden nach 5 
Sekunden" ... ToolTip="ToolTip mit schnellem Ausblenden 
(eine Sekunde)" ToolTipService.ShowDuration="1000"/>

<Button x:Name="btnBetweenShowDelay" Content="Andere 
schnell anzeigen" ... ToolTip="Innerhalb von drei Sekun-
den werden ToolTips, auch wenn InitialShowDelay größer 
ist, direkt angezeigt." ToolTipService.BetweenShowDe-
lay="3000"/>
```

Platzierung der ToolTip-Elemente

Normalerweise erscheint der ToolTip immer dort, wo sich der Mauszeiger gerade befindet. Das Objekt **ToolTipService** liefert jedoch einige Eigenschaften, mit denen Sie die Position des ToolTips sehr genau festlegen können. Sie sind dabei noch nicht einmal auf eine Position abhängig vom Mauszeiger beschränkt, sondern können auch eine absolute Position vom linken oberen Bildschirmrand oder vom Steuerelement aus festlegen.

DATENBANK ENTWICKLER

Fenster mit 1:n-Daten und Lookup-Feld

Im vorherigen Artikel der Kategorie »Von Access zu WPF« haben wir Daten einer einfachen Tabelle in einem Fenster abgebildet. Nun gehen wir einen Schritt weiter und schauen uns die Abbildung zweier per 1:n-Beziehung verknüpfter Tabellen an, wobei die Daten der einen Tabelle per Kombinationsfeld als Lookup-Wert ausgewählt werden sollen. Eines der bekanntesten Beispiele für eine solche Lookup-Tabelle ist die Tabelle tblAnreden. Den anderen Teil übernimmt selbstverständlich die Tabelle tblKunden.

Fenster erstellen

Wir wollen also ein Fenster erstellen, dass die Daten der Tabelle **tblKunden** anzeigt und zur Auswahl einer Anrede für den jeweils angezeigten Kunden ein Kombinationsfeld zur Auswahl eines Datensatzes der Tabelle **tblAnreden** zur Verfügung stellt.

Dazu fügen wir einem Projekt auf Basis der Vorlage **Visual C#|WPF-Vorlagen** zunächst ein neues Fenster namens **TabelleMitLookupDaten** hinzu.

Datenbank hinzufügen

Wir wollen, wie in anderen Beispielen auch, wieder mit der Access-Datenbank **Suedsturm.mdb** arbeiten. Fügen Sie diese daher zum Projekt hinzu. Das gelingt am einfachsten, indem Sie die Datenbank aus dem Windows Explorer einfach auf den Namen des Projekts im Projektmappen-Explorer ziehen.

DataSet hinzufügen

Wie im Beispiel zur Anzeige der Daten einer einfachen Tabelle wollen wir auch hier wieder ein typisiertes DataSet erstellen, das heißt: Wir lassen uns von Visual Studio eine Menge Arbeit abnehmen, indem wir einen Assistenten nutzen.

Um diesen zu starten, wählen Sie den Menüeintrag **Projekt|Neue Datenquelle hinzufügen**.

Im nun erscheinenden Assistent zum Konfigurieren von Datenquellen klicken Sie im ersten Schritt unter der Überschrift **Datenquellentyp auswählen** auf den Eintrag **Datenbank**.

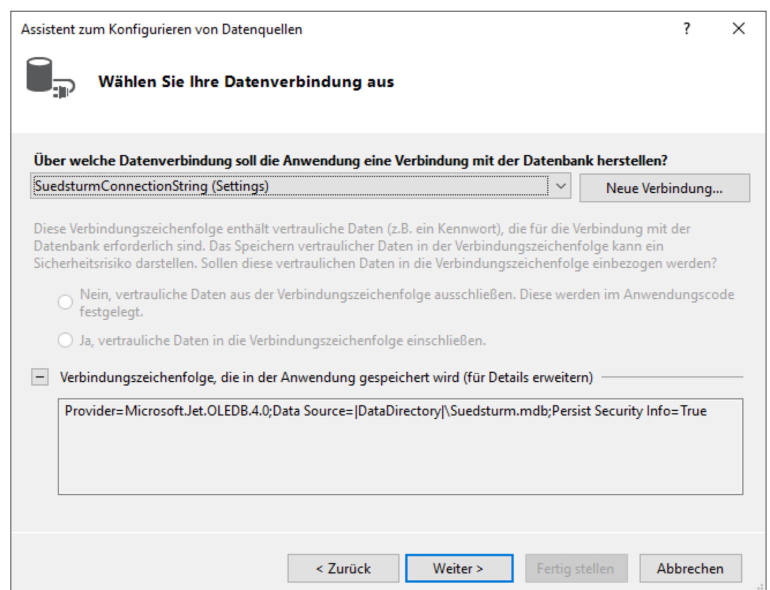


Bild 1: Auswahl einer vorhandenen Datenverbindung

Damit gelangen wir schnell zum zweiten Schritt namens **Datenbankmodell auswählen**. Hier gibt es nicht viel Auswahl, nämlich nur den Eintrag **DataSet**, den Sie folgerichtig ebenfalls doppelt anklicken, um auch diesen Schritt schnell zu beenden.

Sollten Sie das neue Beispiel in der Beispieldatenbank aus dem Artikel **Fenster mit einfachen Tabellendaten** anlegen, sieht der folgende Dialogschritt wie in aus: Es liegt nämlich bereits eine Datenverbindung namens **SuedsturmConnectionString** vor, die Sie beibehalten können (siehe Bild 1). Wenn Sie mit einem neuen, leeren Projekt begonnen haben – kein Problem! Sofern Sie, wie oben angegeben, die Datenbank **Suedsturm.mdb** zum Projekt hinzugefügt haben,

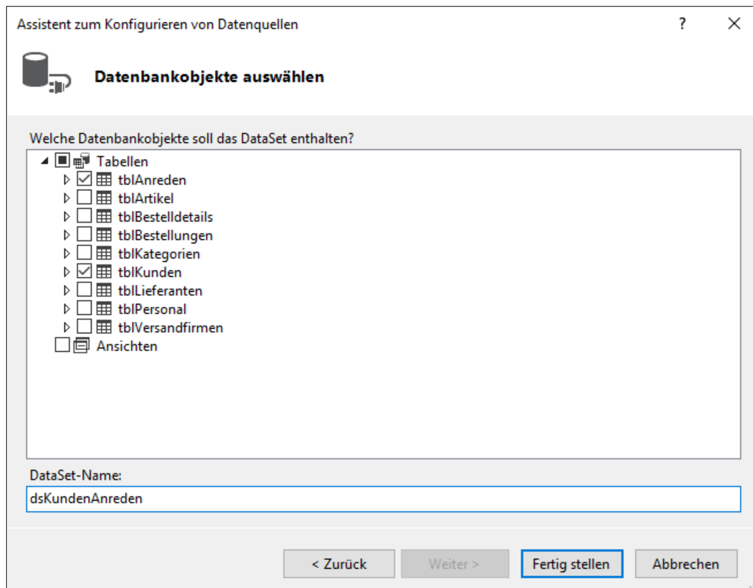


Bild 2: Auswahl der Tabellen und Angabe des Namens für das DataSet

erscheint diese als Auswahlmöglichkeit im genannten Dialog. Klicken Sie hier auf **Weiter**, fragt der Assistent Sie noch, ob Sie die Verbindung unter dem Namen **SuedsturmConnectionString** speichern wollen, was Sie akzeptieren. Beim Anlegen des nächsten DataSets können Sie dann auch auf den Eintrag **SuedsturmConnectionString** zugreifen.

Tabellen für das DataSet zusammenstellen

Nun wollen wir die Tabellen festlegen, die wir im DataSet benötigen. Anders als unter Access benötigen wir hier nicht nur die die Tabelle **tblKunden**, die dort, ein entsprechendes Nachschlagefeld vorausgesetzt, gleich die Daten der Tabelle **tblAnreden** mitliefert, sondern fügen beide Tabellen zum DataSet hinzu. Nach der Auswahl der Tabellen **tblKunden** und **tblAnreden** geben Sie als Namen für das DataSet den Wert **dsKundenAnreden** an (siehe Bild 2).

Damit legt Visual Studio wieder ein Element namens **dsKundenAnreden.xsd** im Projektmappen-Explorer an, dass die Definition des typisierten DataSets und auch die entsprechenden Objekte, Eigenschaften und Methoden bereithält.

Daten der Tabelle tblKunden zum Fenster hinzufügen

Nun fügen wir die Felder der Tabelle **tblKunden** zum Fenster **TabelleMitLookupdaten** hinzu.

Dazu aktivieren Sie zunächst das Datenquellen-Fenster (**Alt + Umschalt + D** oder **Ansicht|Fenster|Datenquellen**). Dieses zeigt nun (gegebenenfalls neben anderen) das DataSet **dsKundenAnreden** mit den beiden untergeordneten Tabellen **tblKunden** und **tblAnreden** an. Wir wollen ein Detailformular für die Darstellung eines Kunden anlegen, also bereiten wir den Eintrag **tblKunden** im **Datenquellen**-Fenster nun darauf vor. Dazu klicken Sie auf **tblKunden**, sodass sich der Eintrag in ein Kombinationsfeld ändert und öffnen dieses. Hier legen Sie den Wert **Details** fest (siehe Bild 3).

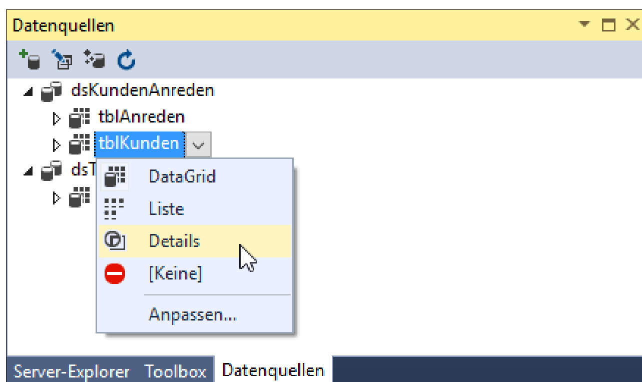


Bild 3: Einstellen der Darstellung der Felder der Tabelle **tblKunden**

Der Eintrag **tblKunden** hat nun ein neues Icon erhalten, was auf die geänderte Ansicht beim Hinzufügen in ein Fenster hinweist (siehe Bild 4). Wenn Sie die Tabelle nun in das Fenster ziehen, erscheinen bereits die gewünschten Felder

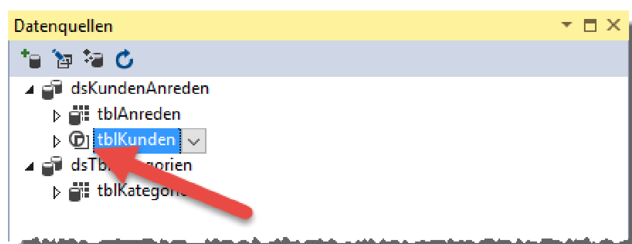


Bild 4: **tblKunden** mit neuem Ansichtstyp

samt Bezeichnungsfeldern – allerdings alle als **TextBox**-Steuerelement. Wir möchten aber doch ein Kombinationsfeld zur Auswahl der Anreden anlegen!

Doch auch dafür sieht der **Datenquellen**-Dialog eine Möglichkeit vor. Klappen Sie dort den Eintrag **tblKunden** auf und klicken Sie dann auf **AnredeID**. Auch hier erscheint ein Kombinationsfeld, aus dem Sie den Eintrag **ComboBox** auswählen (siehe Bild 5).

Damit haben Sie den Steuerelement-Typ für dieses Feld geändert, wie auch die anschließende Anzeige eines neuen Icons für den Eintrag bestätigt (siehe Bild 6).

Wenn Sie nun die Tabelle **tblKunden** in das WPF-Fenster ziehen, erscheint möglicherweise die Fehlermeldung aus Bild 7. In diesem Fall erstellen Sie das Projekt einfach einmal mit dem Menüeintrag **Erstellen|<Projektname> erstellen** (**<Projektname>** durch den aktuellen Projektnamen ersetzen). Im nächsten Anlauf sollte es dann gelingen.

Ziehen Sie dann die Tabelle **tblKunden** in die linke, obere Ecke des WPF-Fenster wie in Bild 8 dargestellt. Die Darstellung im Entwurf sieht zunächst etwas missglückt aus, was aber daran liegt, dass die Inhalte noch nicht initialisiert wurden.

Sie können das Projekt beispielsweise einmal starten oder auch nur eine Eigenschaft des WPF-Fenster aktualisieren (zum Beispiel die Höhe), um die Inhalt der **Label**- und **TextBox**-Steuerelemente korrekt darzustellen.

> Das Beispielprojekt startet das Fenster **MainWindow**, von dem Sie die einzelnen WPF-Beispiele per Mausklick öffnen können.

Nach dem Starten der Anwendung sieht das WPF-Fenster schon recht gut aus – allein beim Feld **AnredeID** gibt es Probleme: Es zeigt nur die gespeicherten Werte an, also die Primärschlüsselwerte der Tabelle **tblAnreden**. Allerdings nicht nur **1** und **2**, sondern beide Werte mehrfach und in

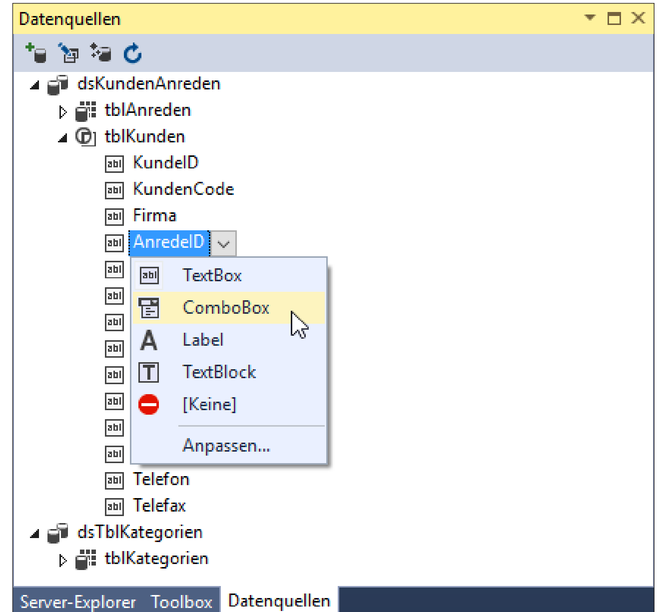


Bild 5: Ändern des Typs des Steuerelements für das Feld **AnredeID**

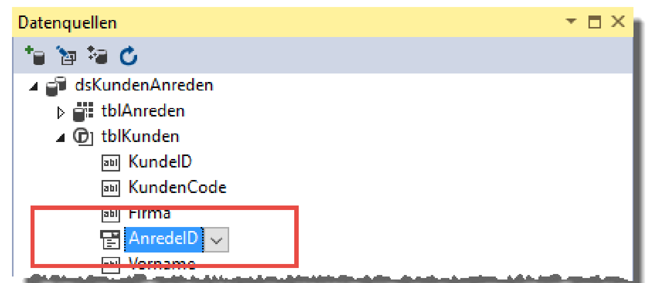


Bild 6: Das Feld **AnredeID** sollte nun als Kombinationsfeld im Fenster landen.

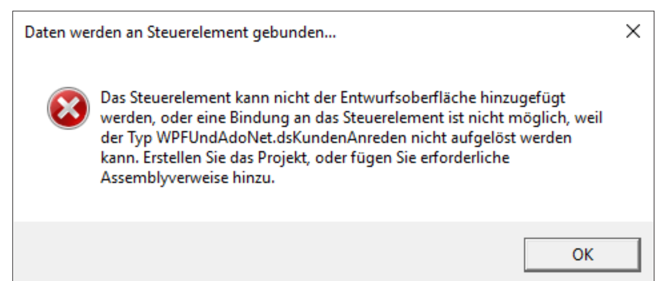


Bild 7: Eventuelle Fehlermeldung beim Versuch, Elemente aus dem Datenquellen-Dialog in ein WPF-Fenster zu ziehen

willkürlicher Reihenfolgt. Es scheint so, als ob dort nicht die Werte des Feldes **AnredeID** der Tabelle **tblAnreden**, sondern die der Tabelle **tblKunden** angezeigt werden (siehe Bild 9).

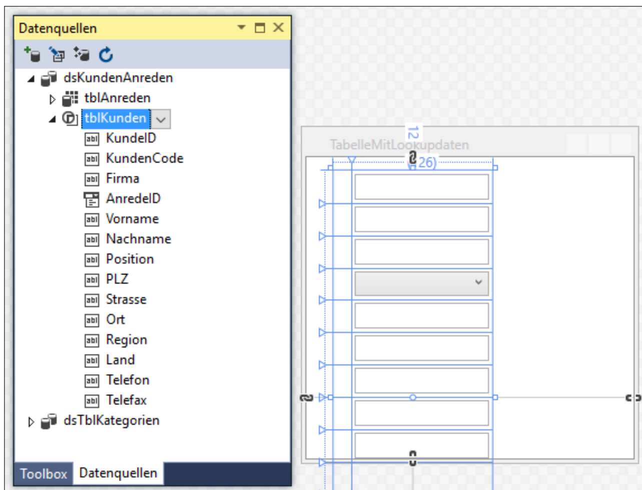


Bild 8: Hinzufügen der Tabelle **tblKunden** zum WPF-Fenster

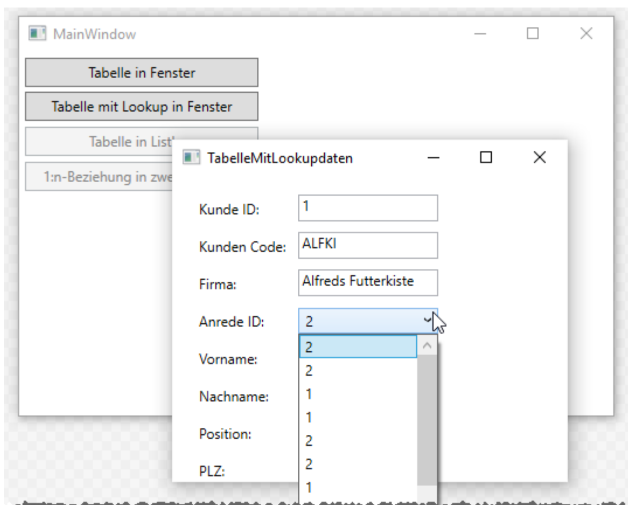


Bild 9: Die Auswahl der Anreden ist verbesserungswürdig ...

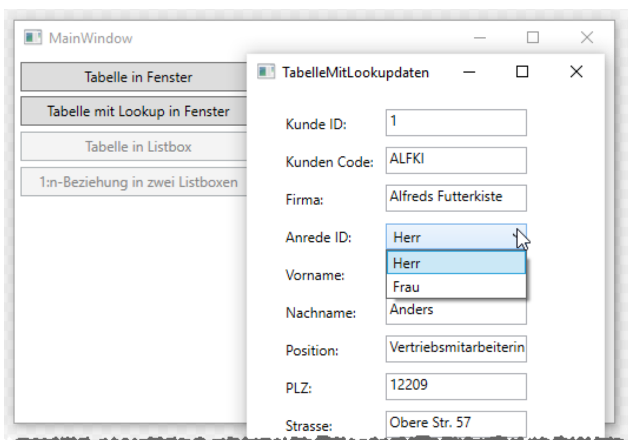


Bild 10: Im zweiten Anlauf stimmt jedoch zumindest die Datenquelle des **ComboBox**-Steuerelements.

Hier müssen wir also wohl nochmal nacharbeiten. Aber wie soll das geschehen? Schauen wir uns doch die Definition des **ComboBox**-Steuerelements an. Diese liefert für **ItemsSource** lediglich den Wert **{Binding}** ohne weitere Angaben. **DisplayMemberPath** enthält den Wert **AnredeID**, das ist definitiv nicht korrekt:

```
<ComboBox x:Name="anredeIDComboBox" Grid.Column="1"
DisplayMemberPath="AnredeID" ... ItemsSource="{Binding}">
    <ComboBox.ItemsPanel>
        <ItemsPanelTemplate>
            <VirtualizingStackPanel/>
        </ItemsPanelTemplate>
    </ComboBox.ItemsPanel>
</ComboBox>
```

Ändern wir dies in **Anrede**, also das anzuzeigende Feld, liefert das nächste Starten der Anwendung lediglich eine ComboBox ohne angezeigte Einträge – die Daten der Tabelle **tblAnreden** scheinen hier also noch nicht berücksichtigt zu werden.

Die Lösung ist recht einfach: Sie ziehen einfach das Feld **Anrede** aus dem Eintrag **dsKundenAnredenltblAnreden** des **Datenquellen**-Dialogs genau auf das **ComboBox**-Steuerelement zur Auswahl der Anreden. Dies bewirkt zunächst nichts Sichtbares, aber wenn Sie dann die Anwendung starten (**F5**), sehen Sie schnell, dass das Kombinationsfeld nun wie gewünscht funktioniert (siehe Bild 10). Allerdings zeigt der Datensatz die Anrede **Herr** an, obwohl es sich hier um einen weiblichen Kunden handelt. Dies sehen wir uns später an.

Und auch am XAML-Code für das **ComboBox**-Steuerelement hat sich etwas geändert:

```
<ComboBox x:Name="anredeIDComboBox"
    DisplayMemberPath="Anrede"
    ItemsSource="{Binding
    Source={StaticResource tblAnredenViewSource}}" ... >
    ...
</ComboBox>
```

Kombinationsfelder mit Daten füllen

Unter Access ist das einfach: Fremdschlüsselfeld als Nachschlagefeld definieren, Tabelle an Formular binden, Fremdschlüsselfeld in den Entwurf ziehen – fertig ist das Kombinationsfeld zur Auswahl von Werten einer Lookup-Tabelle. Unter C#/WPF sieht das etwas anders aus. Dieser Grundlagenartikel zeigt, wie Sie ein Kombinationsfeld mit den Daten einer Lookup-Tabelle füllen und wie Sie auf wichtige Informationen wie etwa den Primärschlüsselwert des gewählten Datensatzes zugreifen – und das auf Basis von typisierten und untypisierten DataSets.

Unter Access/VBA konnten Sie selbst auf Basis einer reinen Tabelle schnell ein Kombinationsfeld erstellen, dass etwa das Feld **Anrede** aus **tblAnreden** anzeigte und das Primärschlüsselfeld **AnredeID** als gebundenes Feld nutzte. Unter WPF/C# sind dazu schon ein paar Zeilen Code nötig, damit es funktioniert.

Tabelle in ComboBox per C#

Die erste Variante, die wir uns dabei ansehen, basiert komplett auf C# (siehe Bild 1). Der XAML-Code, der das Aussehen des **ComboBox**-Steuerelements liefert, weiß nichts davon, dass das Steuerelement Daten aus einer Tabelle einer Datenbank anzeigen soll.

Dementsprechend kommt die Definition dieses **ComboBox**-Steuerelements sehr sparsam daher:

```
<ComboBox x:Name="cboAnreden" HorizontalAlignment="Left"
Margin="95,45,0,0" VerticalAlignment="Top" Width="145" />
```

Der C#-Code, der beim Laden des Fensters ausgeführt werden soll und der für das Füllen des **ComboBox**-Steuerelements verantwortlich ist, fällt dafür umso größer aus (siehe Listing 1). Damit diese Methode aufgerufen wird, fügen wir dem **Window**-Element die Eigenschaft **Loaded** mit dem Wert **Window_Loaded** hinzu (**Loaded** eingeben, zwei Mal auf die Tabulator-Taste drücken). Das **Window**-Element sieht dann etwa so aus:

```
<Window Title="MainWindow" ... Height="350" Width="525" Wi
ndowStartupLocation="CenterScreen" Icon="data_table.ico"
Loaded="Window_Loaded">
    ...
</Window>
```

Die Methode **Window_Loaded** erledigt zunächst nichts Anderes, als ein untypisiertes DataSet zu erstellen – und zwar auf Basis der Tabelle **tblAnreden** der Access-Datenbank namens **Suedsturm.mdb**, die im gleichen Verzeichnis wie die **.exe**-Datei der erstellten Anwendung liegen soll.

> Im Beitrag **Datenzugriff mit ADO.NET, Teil 2** erfahren Sie, wie untypisierte **DataSet**-Objekte funktionieren und wie Sie **DataTable**-Objekte nutzen.

Wichtig ist, dass wir am Ende ein **DataTable**-Objekt mit den gewünschten Daten der Tabelle **tblAnreden** erhalten. Dieses liefert über die Eigenschaft **DefaultView** eine Form, in der wir es an die Eigenschaft **ItemsSource** des **ComboBox**-Steuerelements übergeben können. Dies entspricht grundsätzlich dem Zuweisen einer Tabelle an die Eigenschaft **RecordSource** unter Access/VBA. Dort haben Sie für ein Kombinationsfeld über die Eigenschaften **Gebundene Spalte**, **Spaltenanzahl** und **Spaltenbreiten** festgelegt, welche

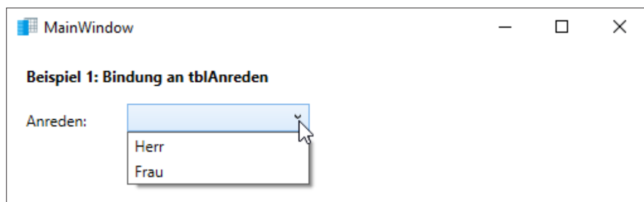


Bild 1: Einfaches Kombinationsfeld mit den Daten der Tabelle **tblAnreden**

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    DataTable dtAnreden = new DataTable();
    string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    string strSQL = "SELECT * FROM tblAnreden";
    OleDbConnection cnn = new OleDbConnection(strConnection);
    OleDbCommand cmd = new OleDbCommand();
    OleDbDataAdapter da;
    DataSet ds = new DataSet();
    DataTable dt;
    cmd.Connection = cnn;
    cmd.CommandText = strSQL;
    da = new OleDbDataAdapter(strSQL, cnn);
    da.Fill(ds, "tblAnreden");
    dt = ds.Tables["tblAnreden"];
    cboAnreden.ItemsSource = dt.DefaultView;
    cboAnreden.DisplayMemberPath = "Anrede";
    cboAnreden.SelectedValuePath = "AnredeID";
}
```

Listing 1: Füllen einer ComboBox per C#

Spalte als Wert des Kombinationsfeldes dienen und welche angezeigt werden sollte. Das ist unter C# etwas gezielter formuliert: Hier nutzen Sie die Eigenschaft **DisplayMemberPath**, um den Namen der anzuzeigenden Spalte zu übergeben, in diesem Fall das Feld **Anrede**.

Die Eigenschaft **SelectedValuePath** hingegen nimmt das Feld entgegen, das man unter Access mit dem Index der gebundenen Spalte festgelegt hätte – in diesem Fall **AnredeID**.

Angezeigten Wert und tatsächlichen Wert auslesen

Damit ist das Kombinationsfeld bereits ausgestattet. Schauen wir uns nun an, wie Sie per Code auf die beiden Werte für den aktuell markierten Eintrag des **ComboBox**-Steuerelements zugreifen können. Dazu fügen Sie der XAML-Definition des **ComboBox**-Steuerelements das Attribut **SelectionChanged** hinzu und legen gleich noch die entsprechende Methode an:

```
<ComboBox x:Name="cboAnreden" ...
SelectionChanged="cboAnreden_SelectionChanged"/>
```

Die Methode ergänzen Sie wie in Listing 2. Dort deklarieren wir zunächst eine Variable des Typs **ComboBox** und weisen dieser den mit dem Parameter **sender** gelieferten Verweis auf das auslösende Steuerelement zu.

Dann deklariert die Methode ein Objekt des Typs **DataRowView**, was einem Datensatz der **DataView** entspricht,

```
private void cboAnreden_SelectionChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e) {
    ComboBox cbo = sender as ComboBox;
    DataRowView vw = cbo.SelectedItem as DataRowView;
    string anrede = vw["Anrede"].ToString();
    MessageBox.Show("Angezeigter Wert: " + cboAnreden.SelectedValue.ToString() + "\nAngezeigter Text: " + anrede);
}
```

Listing 2: Ausgabe von Wert und angezeigtem Text für den ausgewählten Eintrag eines Kombinationsfeldes

die Sie dem **ComboBox**-Steuerelement als **ItemsSource** zugewiesen haben. Diese füllen Sie mit dem Objekt, das die Eigenschaft **SelectedItem** des Kombinationsfeldes liefert – zuvor mit **as DataRowView** explizit in den entsprechenden Datentyp konvertiert.

Den angezeigten Wert erhalten Sie dann über das Feld **Anrede** des **DataRowView**-Objekts, den tatsächlichen Wert des mit **SelectedValuePath** angegebenen Feldes liefert die Eigenschaft **SelectedValue** des Kombinationsfeldes. Beides gibt die Methode in einem Meldungsfenster aus (siehe Bild 2).

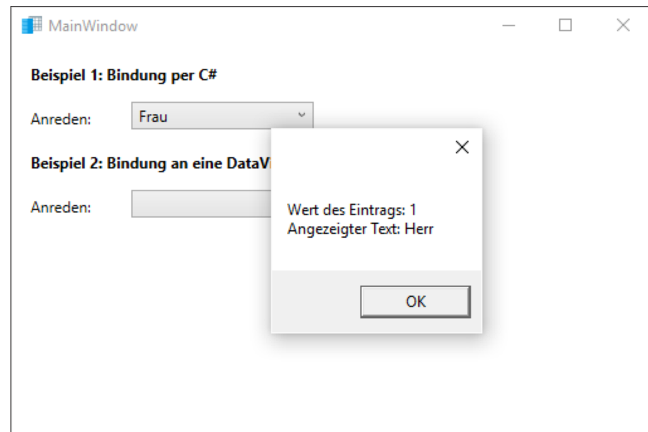


Bild 2: Ausgabe des Wertes und des angezeigten Textes eines Kombinationsfeldes per Meldungsfenster

ComboBox aus Eigenschaft füllen

Im zweiten Beispiel wollen wir so viele XAML-Attribute wie möglich verwenden, um das **ComboBox**-Steuerelement mit Daten zu füllen, und so die Funktionalität vom C#-Code nach WPF verlagern.

Dazu fügen Sie einem neuen Element dieses Typs namens **cboAnreden_WPF** hinzu und ergänzen einige Attribute:

```
<ComboBox x:Name="cboAnreden_DataView" ...
    DisplayMemberPath="Anrede"
    SelectedValuePath="AnredeID"
    ItemsSource="{Binding Path=dvAnreden}"/>
```

DisplayMemberPath und **DisplayValuePath** übernehmen wir einfach von der C#-Version des vorherigen Beispiels. **ItemsSource** benötigen wir auch, allerdings können wir

```
public DataView dvAnreden {
    get {
        DataTable dtAnreden = new DataTable();
        string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
        string strSQL = "SELECT * FROM tblAnreden";
        OleDbConnection cnn = new OleDbConnection(strConnection);
        OleDbCommand cmd = new OleDbCommand();
        OleDbDataAdapter da;
        DataSet ds = new DataSet();
        DataTable dt;
        cmd.Connection = cnn;
        cmd.CommandText = strSQL;
        da = new OleDbDataAdapter(strSQL, cnn);
        da.Fill(ds, "tblAnreden");
        dt = ds.Tables["tblAnreden"];
        return dt.DefaultView;
    }
}
```

Listing 3: Eigenschaft, um ein **ComboBox**-Element mit den Daten eines **DataTable**-Objekts zu füllen

diesem natürlich nicht einfach ein **DataView**-Objekt wie im C#-Code mit der **DefaultView** des **DataTable**-Objekts zuweisen – ein solches ist in dem Kontext ja noch gar nicht bekannt. Stattdessen fügen wir dem Code behind-Modul, also dem C#-Modul mit dem Code des Fensters, eine öffentlich deklarierte Eigenschaft namens **dvAnreden** hinzu, die eine **DataView** auf Basis der **DataTable** mit der Tabelle **tblAnreden** liefern soll.

Dazu erhält die Eigenschaft den **get**-Zweig, in die wir prinzipiell den gleichen Code zum Erstellen des **DataTable**-Objekts auf Basis eines untypisierten DataSets hinzufügen, den wir schon in der Methode **Window_Loaded** des ersten Beispiels genutzt haben. Die Unterschiede sind, dass wir hier das mit **DefaultView** ermittelte **DataView**-Objekt als Eigenschaftswert zurückliefern (siehe Listing 3). Diese weist der XAML-Code dann über das Attribut **ItemsSource** dem **ComboBox**-Element zu, und zwar per Binding und dessen Attribut **Path**:

```
ItemsSource="{Binding Path=dvAnreden}"
```

Damit zeigt das **ComboBox**-Steuerelement allerdings leider noch keine Datensätze der Tabelle **tblAnreden** an. Das fehlende Glied ist die Zuweisung der Code behind-Klasse als Datenquelle für die Steuerelemente des Fensters. Diese Zuweisung nehmen wir in der beim Anzeigen des Fensters ausgelösten Methode **MainWindow** vor – gleich hinter der Zeile **InitializeComponents**:

```
public MainWindow() {
    InitializeComponent();
    DataContext = this;
}
```

Datenquelle per WPF zuweisen

Das Attribut **DataContext** finden Sie allerdings auch im **Window**-Element des XAML-Codes. Wenn Sie dort Folgendes definieren, haben Sie noch eine Zeile weniger C#-Code:

```
<Window Title="MainWindow" ... DataContext="{Binding
RelativeSource={RelativeSource Self}}">
```

ComboBox aus typisiertem DataSet füllen

Dies war ein Beispiel, wie Sie die Daten eines untypisierten DataSets in einem **ComboBox**-Steuerelement anzeigen. Wenn Sie ein typisiertes DataSet verwenden, sieht dies etwas anders aus. Die Unterschiede beschränken sich jedoch auf die Angabe des Wertes für das Attribut **ItemsSource**.

Während wir beim untypisierten DataSet im vorliegenden Fall eine Eigenschaft nutzen, die ein **DataView**-Objekt liefert, verweisen wir beim typisierten DataSet per Binding auf eine Ressource, die wiederum an die entsprechende Tabelle des DataSets gebunden ist.

Wenn Sie das gleiche Kombinationsfeld wie oben aus einem typisierten DataSet füllen wollen, benötigen Sie zunächst ein entsprechendes **DataSet**-Objekt, dass zumindest die Tabelle **tblAnreden** der Quelldatenbank enthält.

> Wie Sie ein DataSet auf Basis einer Access-Datenbank erzeugen, erfahren Sie unter anderem im Beitrag **Fenster mit einfachen Tabellendaten anlegen**.

Aktivieren Sie dann beispielsweise mit der Tastenkombination **Alt + Umschalt + D** den **Datenquellen**-Dialog.

Hier finden Sie nun alle zum DataSet hinzugefügten Tabellen, unter denen sich auch die Tabelle **tblAnreden** befinden sollte. Klicken Sie auf den Eintrag, verwandelt sich dieses

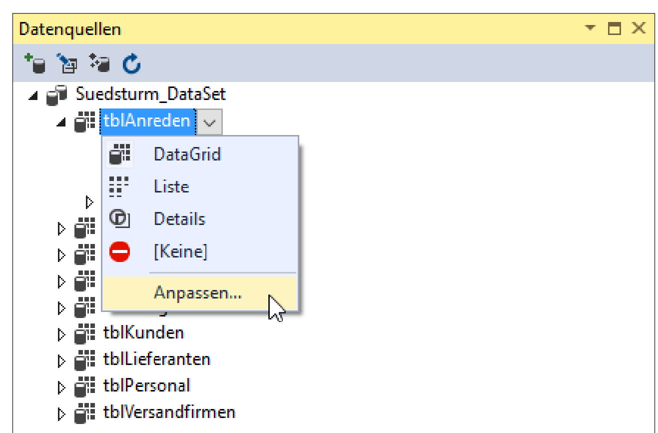


Bild 3: Aufrufen des **Anpassen**-Dialogs

SQL Server 2014 Express installieren

Eine .NET-Anwendung kann man als Frontend für eine Access-Datenbank nutzen. Den vollen Nutzen erhält man allerdings erst, wenn man beispielsweise den SQL Server 2014 als Backend verwendet. Also schauen wir uns den SQL Server 2014, die derzeit aktuelle Version des relationalen Datenbankmanagement-Systems von Microsoft, etwas genauer an. In diesem Artikel erfahren Sie, welche Editionen es gibt und wie Sie den SQL Server 2014 auf Ihrem Rechner installieren.

SQL Server 2014-Editionen

Der SQL Server 2014 kommt in den folgenden Editionen:

- SQL Server 2014 Express: Diese Version ist kostenlos und kann beliebig eingesetzt werden. Sie hat allerdings ein paar Einschränkungen. Auf der technischen Seite sind dies beispielsweise die Begrenzung auf den nutzbaren Arbeitsspeicher von einem Gigabyte, vier Prozessorkerne und eine maximale Datenbankgröße von zehn Gigabyte. Gegenüber den zwei Gigabyte einer Access-Datenbank ist dies schon eine beachtliche Steigerung.
- SQL Server 2014 Enterprise, Business Intelligence und Standard: Enthalten je nach Version weitere Features, die wir aber im Rahmen dieses Magazins vorerst nicht benötigen.

Weitere Informationen zu den verschiedenen Editionen finden Sie unter **https://www.microsoft.com/de-de/server-cloud/products/sql-server-editions/overview.aspx**.

Damit ist schnell klar: Wenn Sie den SQL Server 2014 benötigen, um mit Datenbankanwendungen auf Basis von .NET zu experimentieren, reicht die kostenlose SQL Server 2014 Express-Edition völlig aus. Deren Installation beschreiben wir in den folgenden Abschnitten.

SQL Server 2014 Express-Versionen

Den Download finden Sie, wenn Sie von oben genanntem Link aus unter **SQL Server 2014 Express** auf **Weitere Informationen** klicken. Hier folgt eine weitere Herausforderung in Form verschiedener Versionen:

- **SQL Server Express mit Tools**: Enthält nur die Verwaltungstools.
- **SQL Server Management Studio**: Enthält nur die Management-Tools für den SQL Server, nicht aber das Datenbankmanagement-System selbst.
- **SQL Server LocalDB (MSI-Installationsprogramm)**: Schnell zu installierende Version des SQL Servers in einer einfachen Variante.
- **SQL Server Express mit Advanced Services**: Paket mit SQL Server 2014, Reporting Services, SQL Server Management Studio und mehr.
- **SQL Server Express**: SQL Server ohne Tools.

Die Developer Edition

Neben den oben genannten Editionen gibt es auch noch die Developer Edition. Dabei handelt es sich um eine voll ausgestattete SQL Server-Version, die Sie zu einem recht günstigen Preis erwerben können (maximal um die 100 EUR). Diese darf jedoch nur zum Entwickeln von Anwendungen genutzt werden. Sollten Sie also damit experimentieren wollen und Sie benötigen Funktionen, welche die Express Edition nicht liefert, können Sie auf die Developer Edition zurückgreifen.

Download von SQL Server 2014 Express

Wir gehen davon aus, dass Sie den bequemen Weg über den Download der gewünschten Version des SQL Servers wählen. Alternativ haben Sie diese möglicherweise schon als DVD vorliegen und installieren den SQL Server von dort aus.

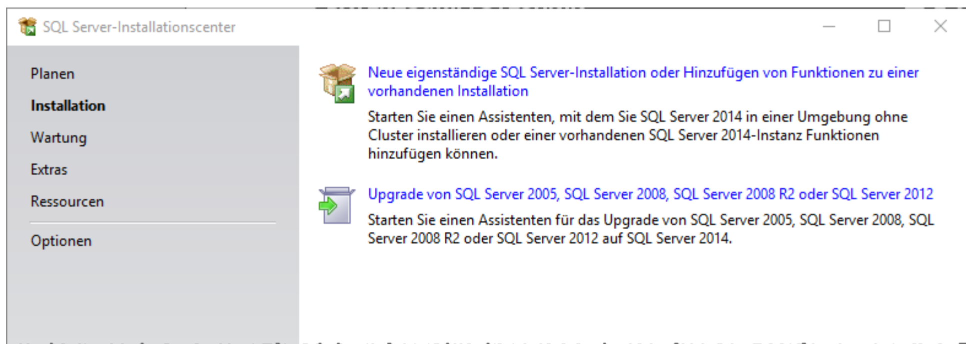


Bild 1: Neu oder Update?

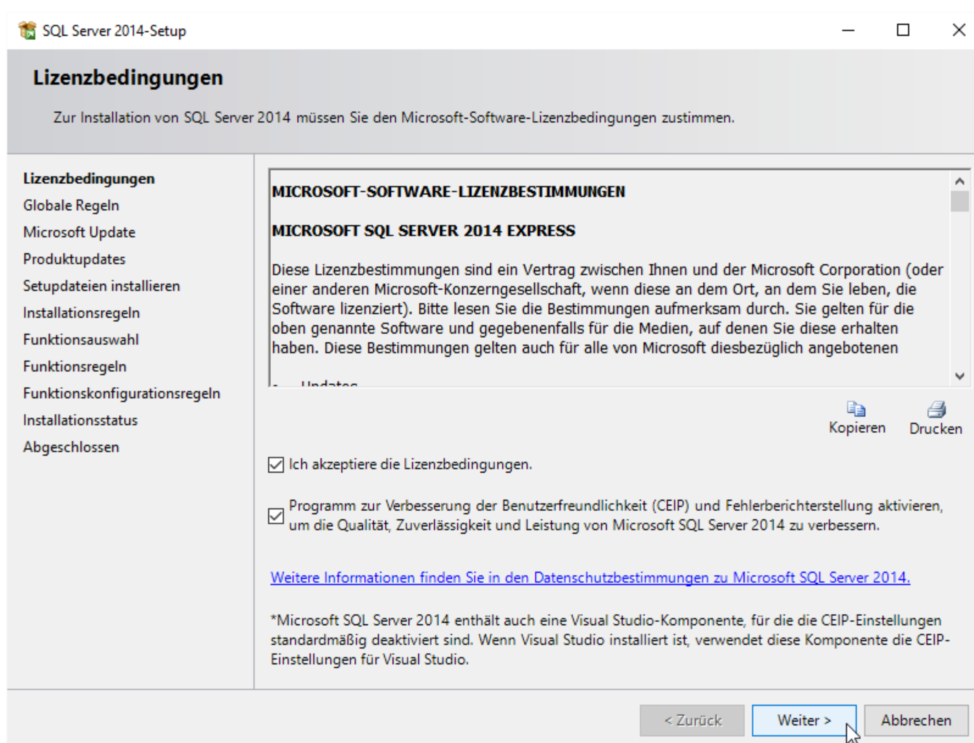


Bild 2: Akzeptieren der Lizenzbedingungen und Übersicht

Bevor Sie mit dem Download beginnen, müssen Sie sich mit Ihrem Microsoft-Konto einloggen oder ein neues Konto anlegen. Danach wählen Sie die gewünschte Version aus.

Wir gehen auf Nummer sicher und verwenden die Version **SQL Server 2014 Express with Advanced Services 64bit** (für Rechner mit 64bit-Betriebssystem). Dies lädt, sofern Sie die deutsche Version gewählt haben, die Datei **SQLEXPRADV_x64_DEU.exe** herunter – mit immerhin rund 1.200 Megabyte Größe.

Das Setup fragt zunächst, ob Sie eine neue Installation vornehmen möchten oder ob Sie ein Update einer bereits auf dem Rechner vorhandenen Version durchführen möchten (siehe Bild 1).

Im zweiten Schritt akzeptieren Sie die Lizenzbedingungen und entscheiden sich nach Wunsch dafür, Microsoft durch entsprechendes Feedback bei der Weiterentwicklung des SQL Servers zu unterstützen (siehe Bild 2).

Der Schritt **Microsoft Update** prüft, ob Aktualisierungen für die aktuelle Windows-Installation und auch für die zu installierende SQL Server-Software vorliegen und fragt, ob Sie diese herunterladen wollen.

Installationsoptionen

Direkt im Anschluss folgt der wichtige Schritt aus Bild 3. Hier legen Sie unter anderem fest, mit welchen Funktionen SQL Server installiert werden soll. Wenn Sie genügend Speicherplatz haben, sollten Sie direkt alle Optionen, die standardmäßig angeboten werden, beibehalten. Auf die **SQL Server Replikation** und die **Volltext- und semantische Extraktion für die Suche** können Sie gegebenenfalls verzichten, zumindest werden wir in diesem Magazin wohl nicht auf diese Funktionen eingehen. Die **Reporting Services** werden wir früher oder später sicher ansprechen, da – wenn Sie von Access zu .NET wechseln – ja auch die von Access gewohnten Berichte

wegfallen. Von den übrigen Funktionen ist vor allem der Eintrag **Verwaltungstools – Vollständig** interessant, da Sie damit nicht nur das Management Studio erhalten, sondern auch den Profiler und Unterstützung für die Reporting Services.

Klicken Sie auf die einzelnen Einträge unter **Funktionen**, damit unter **Funktionsbeschreibung** Details zu der jeweiligen Funktion eingeblendet werden.

Schließlich legen Sie unter **Instanzstammverzeichnis** noch fest, in welchem Bereich des Dateisystems die Installation erfolgen soll.

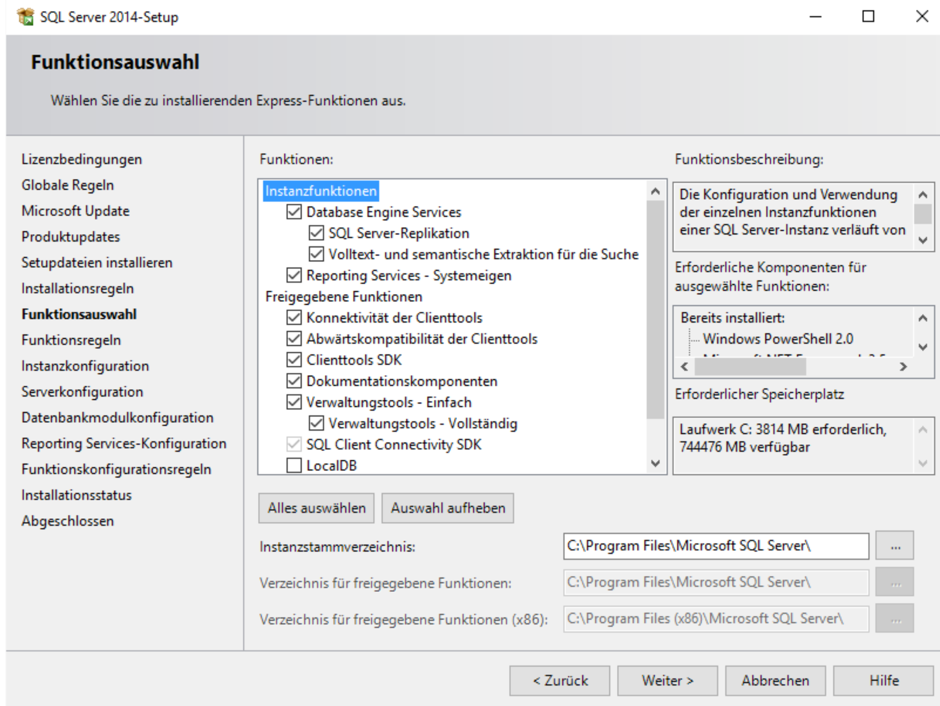


Bild 3: Festlegen der Installationsoptionen

Standardinstanz oder benannte Instanz

Nach ein paar Sekunden Verschnaufpause meldet sich der Installationsassistent wieder zurück und stellt den Dialog aus Bild 4 bereit.

Hier legen Sie nun fest, ob Sie den SQL Server als Standardinstanz oder als benannte Instanz installieren möchten. Im Gegensatz zu früheren Varianten stellt der Installationsassistent den Standardwert auf **Benannte Instanz** ein und schlägt als Namen **SQLEXPRESS** vor. Dies ist sinnvoll, wenn Sie mehrere Instanzen des SQL Servers auf einem Rechner laufen lassen möchten. Wenn Sie also etwa schon einen SQL Server betreiben und für Experimente mit

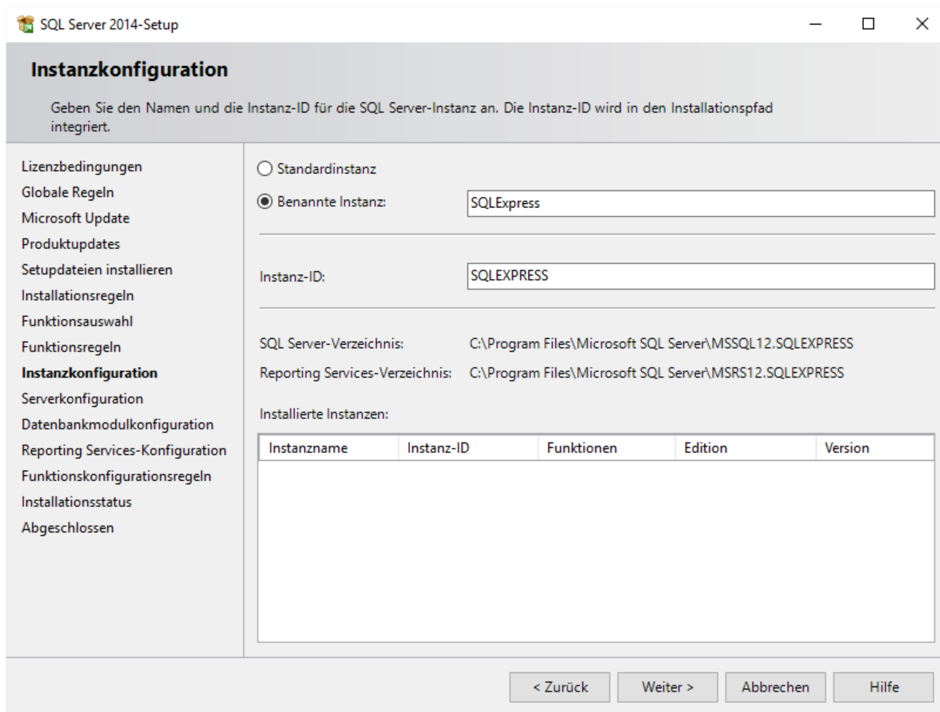


Bild 4: Installieren einer Standardinstanz oder einer benannten Instanz

Von der .mdb-Datei zum SQL Server

Der Datenzugriff von .NET auf eine .mdb-Datei ist möglich, aber die vollen Möglichkeiten erlaubt erst beispielsweise eine Datenbank, die mit dem SQL Server verwaltet wird. Da viele Leser wohl von der Access-Schiene kommen, dürfte der eine oder andere seine Datenbank zum SQL Server transferieren wollen, um damit von einer .NET-Anwendung aus zu experimentieren. Dieser Artikel zeigt, wie Sie eine Migration der Tabellen und Abfragen einer Anwendung schnell realisieren.

Voraussetzung für die Durchführung der hier vorgestellten Techniken ist das Vorhandensein einer zu migrierenden Access-Datenbank sowie einer Instanz des SQL Servers.

Außerdem wollen wir den **SQL Server Migration Assistent for Access** (kurz **SSMA**) nutzen, um die Migration durchzuführen. Diesen laden Sie von der folgenden Webseite herunter:

https://www.microsoft.com/en-us/download/confirmation.aspx?id=43690

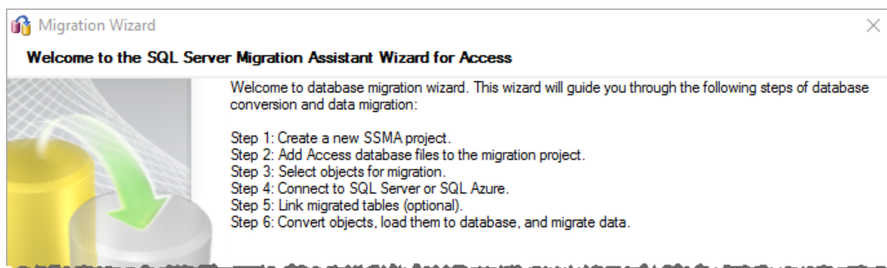


Bild 1: Startdialog des SSMA-Assistenten

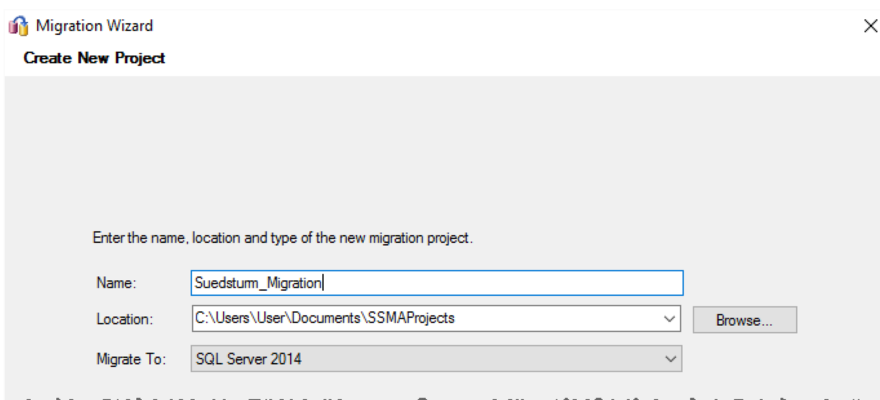


Bild 2: Angabe des Namens des Migrationsprojekts und des Speicherortes

Nach dem Download starten Sie direkt den Setup-Assistenten des SSMA. Die dazu nötigen Schritte sind selbsterklärend.

Start des SQL Server Migration Assistants

Der Assistent wird standardmäßig mit dem SSMA aufgerufen und liefert vorab eine Liste der anstehenden Aufgaben (siehe Bild 1).

Im ersten Schritt fragt der Assistent dann direkt den Namen des zu erstellenden SSMA-Projekts ab (nicht zu verwechseln mit dem Namen der zu erstellenden Datenbank) sowie den Speicherort der Projektdatei (siehe Bild 2).

Quelldatenbank auswählen

Der folgende Schritt des Assistenten fragt dann den Namen der Access-Datenbank ab, deren Tabellen in eine SQL Server-Datenbank übertragen werden sollen (siehe Bild 3). Hier wählen wir die Datei **Suedsturm.mdb** aus, da wir deren Tabellen für folgende Artikel als Beispieldatenbank verwenden wollen.

Danach folgt die Auswahl der zu migrierenden Tabellen und Abfragen. Wir wollen an dieser Stelle nur Tabellen übertragen, also aktivieren wir nur den Haken des Eintrags **Tabellen** und erweitern die darunter

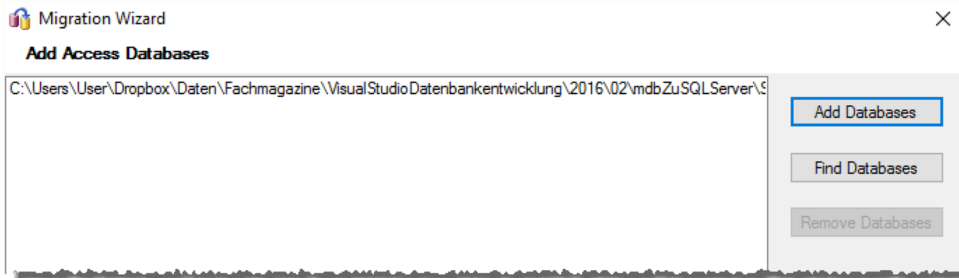


Bild 3: Hinzufügen der zu migrierenden Access-Datenbank

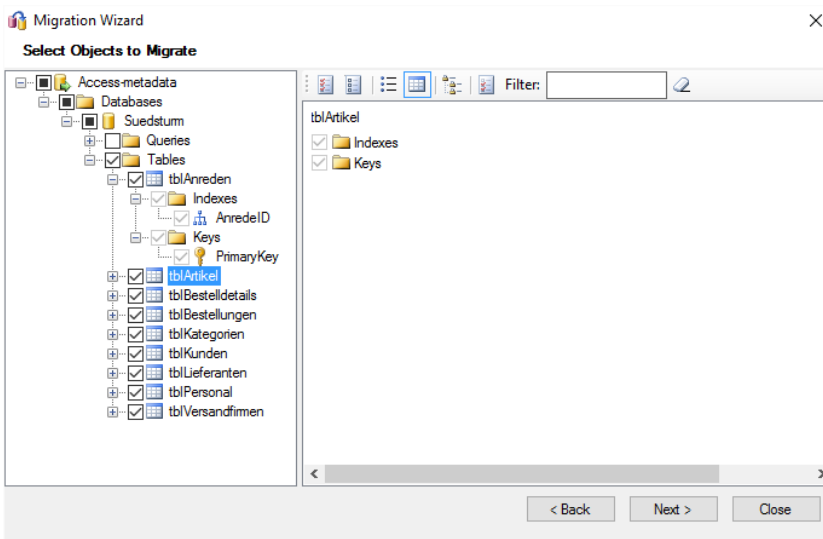


Bild 4: Auswahl der zu migrierenden Tabellen und Felder sowie ihrer Eigenschaften

befindlichen Elemente (die Migration von Abfragen ist auch nicht sonderlich ergiebig). Im rechten Bereich sehen Sie dann die Eigenschaften der Tabellen, insbesondere der Indizes und Schlüsselfelder.

Mögliche Probleme

Bei meiner Konstellation (Windows 10, Office 2016 32bit) gab es ein Problem beim Zugriff auf die zu migrierende Access-Datenbank, das sich im Auftauchen eines Ausrufezeichens wie in Bild 5 zeigte. Dieses tritt manchmal auf, wenn man die falsche Version des SSMA verwendet, also die 32bit-Variante mit dem 64bit-Office oder die 64bit-Variante mit dem 32bit-Office. Sollten bei Ihnen im Migrationsassistenten keine Tabellen angezeigt werden und ein Ausrufezeichen im Icon der Datenbank erscheinen, ist dies möglicherweise der Fall. Achten Sie dann darauf, die Version des SSMA zu starten, die zu Ihrem Office-Paket passt (also etwa SSMA 32bit zu Office 32bit).

Das Problem kann noch eine weitere Ursache haben. Dabei fehlen bestimmte Komponenten beziehungsweise sie liegen nicht in der richtigen Version vor. Laden Sie dann die Komponente **Microsoft Access Database Engine 2010 Redistributable** von folgender Internetadresse herunter:

https://www.microsoft.com/de-DE/download/details.aspx?id=13255

Anschließend sollte der Fehler nicht mehr auftreten. Die Komponente ist zwar für 2010 ausgezeichnet, aber es funktioniert bei mir auch unter Access 2016.

Ziel festlegen

Im folgenden Schritt geben Sie den Namen der SQL Server-Instanz ein, in der die neue Datenbank erstellt werden soll. Leider zeigt der Dialog aus Bild 6 in unserem Fall keine Liste der verfügbaren SQL Server an, sodass wir diesen manuell eintragen mussten. Auch erschien nach der Eingabe keine Liste der vorhandenen Datenbank, was aber keine Rolle spielte, da wir diese ja ohnehin neu erstellen wollten. Wir haben der Datenbank den Namen **Suedsturm_SQL** gegeben.

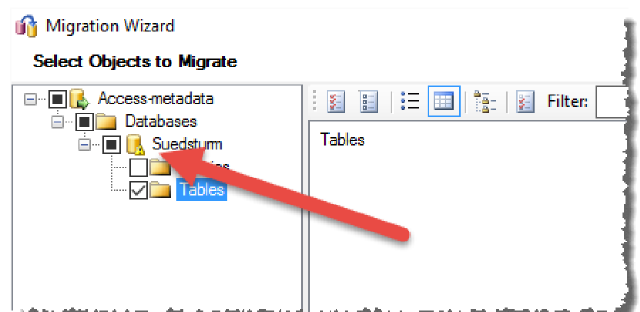


Bild 5: Problem mit dem Einlesen der Datenbank

Da die Datenbank zu diesem Zeitpunkt noch nicht angelegt wurde, fragt der SSMA-Assistent noch, ob diese neu erstellt werden soll (siehe Bild 7).

Verknüpfungen erstellen?

Auf Wunsch können Sie nun festlegen, ob in der Access-Datenbank Verknüpfungen zu den in der SQL Server zu erstellenden Tabellen hinzufügt werden sollen. Eigentlich ist das nicht nötig, da wir ja von .NET-Anwendungen auf die Tabellen zugreifen wollen.

Allerdings kann es auch nicht schaden, gelegentlich über ein Access-Frontend auf die Tabellen zuzugreifen – etwa, um zu prüfen, ob eine Änderung an den Daten wie gewünscht durchgeführt wurde. In diesem Fall aktivieren Sie die Option **Link Tables** des Dialogs aus Bild 8.

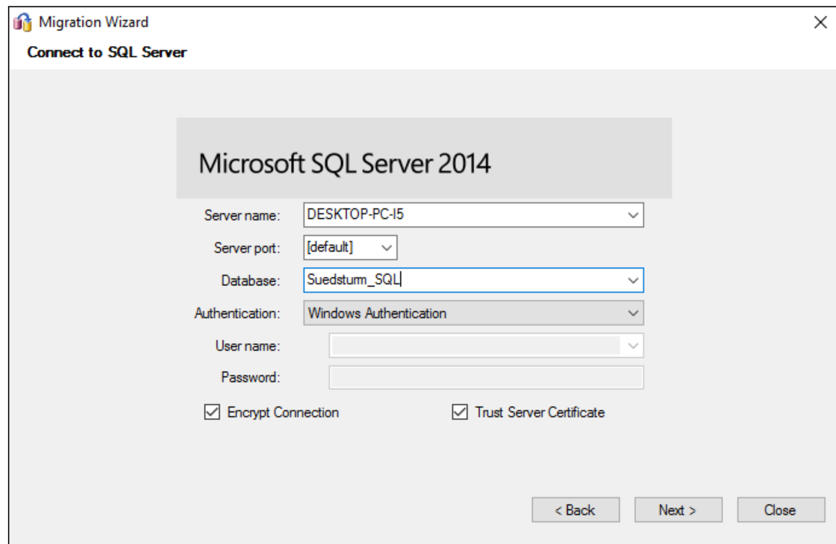


Bild 6: Festlegen der SQL Server-Instanz und der Zieldatenbank

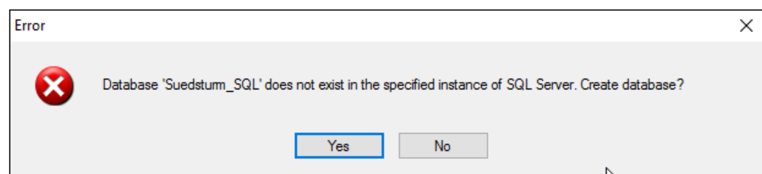


Bild 7: Soll die Datenbank neu erstellt werden?

Migration durchführen

Danach beginnt die eigentliche Migration. Das Ergebnis zeigt der Dialog **Migration Status** (siehe Bild 9). Im Falle der Migration der Beispieldatenbank traten keinerlei Fehler und lediglich einige Warnungen und Hinweise auf. Die vier dargestellten Bereiche haben dabei folgende Bedeutung:

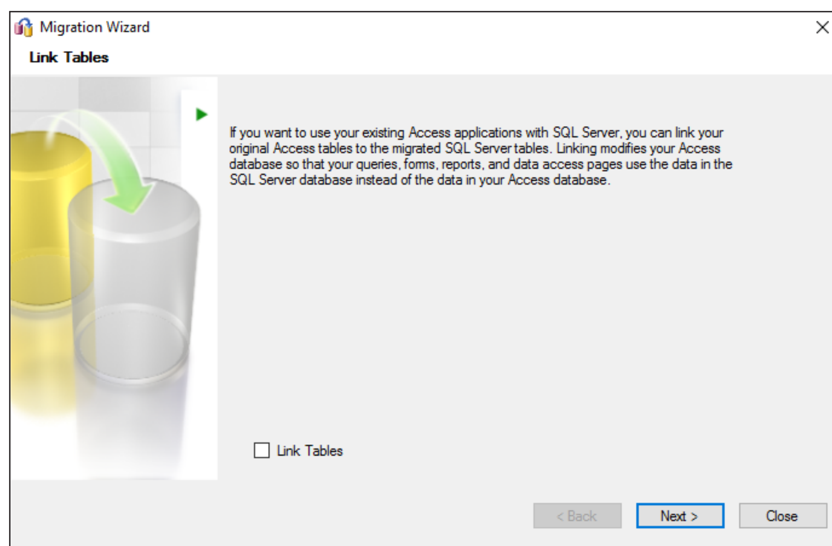


Bild 8: Sollen die Tabellen verknüpft werden?

- **Convert selected objects**: Umwandlung der Objekte in Form eines Skripts zur Definition von Tabellen- und Abfragestruktur
- **Load converted objects into the target database**: Tabellen und Sichten in der SQL Server-Datenbank erstellen
- **Migrate data for selected objects**: Migration der in den Tabellen enthaltenen Daten
- **Link converted tables**: Erstellen der Verknüpfungen zu den neu erstellten Tabellen in der Access-Datenbank

ADO.NET: Typisierte und untypisierte DataSets

Unter ADO.NET gibt es typisierte und untypisierte DataSets. In den bisherigen Artikeln haben Sie meist mit untypisierten DataSets gearbeitet. Das heißt, dass Sie alle benötigten Objekte per Code erzeugt und dabei auch Verbindungseigenschaften, SQL-Anweisungen et cetera übergeben haben. Bei typisierten DataSets definieren Sie die meisten Eigenschaften des später im Code verwendeten DataSets über einen Assistenten. Dies gelingt durch einige Klassen, die vom Assistenten automatisch erzeugt werden und mit deren Hilfe Sie später direkt per IntelliSense auf die enthaltenen Tabellen und Felder zugreifen können.

Doch eins nach dem anderen: Wir wollen zunächst einmal den zuständigen Assistenten bemühen, um ein typisiertes DataSet auf Basis zweier Tabellen der Beispieldatenbank **Suedsturm.mdb** zu erzeugen. Später zeigen wir dann, welche Möglichkeiten das typisierte DataSet gegenüber seinem nicht typisierten Pendant liefert.

Datenquelle hinzufügen

Wir wollen die beiden Tabellen **tblArtikel**, **tblKategorien** und **tblLieferanten** zum DataSet hinzufügen. Um dies zu erledigen, zeigen Sie zunächst mit dem Menüeintrag **Ansicht|Server**-Explorer den Server-Explorer an, der auch einen Eintrag namens **Datenverbindungen** enthält. Wählen Sie den Eintrag **Verbindung hinzufügen...** aus dem Kontextmenü dieses Eintrags aus.

Es erscheint nun der Dialog Datenquelle auswählen, der verschiedene Datenquellentypen anzeigt. Wählen Sie hier den Typ **Microsoft Access-Datenbankdateien** aus und klicken Sie auf **Weiter** (siehe Bild 1).

Im folgenden Schritt geben Sie die konkrete Access-Datenbankdatei an. Testen Sie die Verbindung, indem Sie auf Testverbindung klicken. Liefert dies keinen Fehler, beenden Sie den Assistenten mit einem Klick auf die Schaltfläche **OK** (siehe Bild 2).

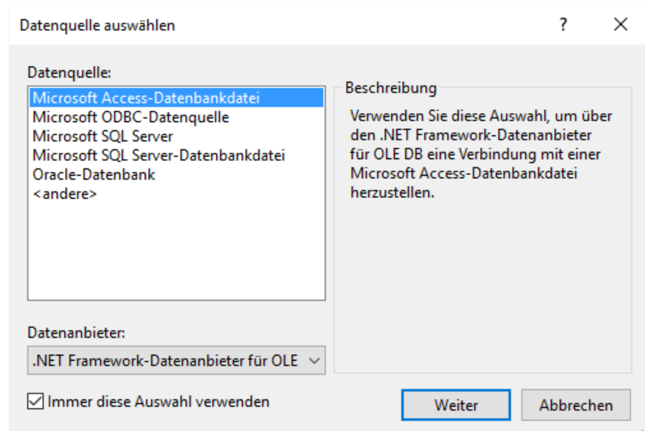


Bild 1: Auswählen des Typs der Datenquelle

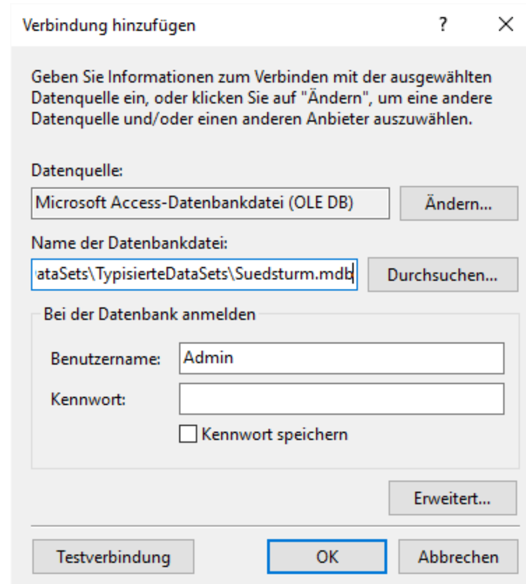


Bild 2: Auswählen der Datenquelle

Im Anschluss finden Sie im Server-Explorer unter Datenverbindungen einen neuen Eintrag namens **Suedsturm.mdb** vor. Wenn Sie diesen Eintrag erweitern, finden Sie schnell heraus, dass auch alle enthaltenen Tabellen zur Verfügung stehen (siehe Bild 3).

Typisiertes DataSet Erstellen

Wählen Sie nun den Menüeintrag **Projekt|Neues Element hinzufügen...** aus, erscheint der Dialog

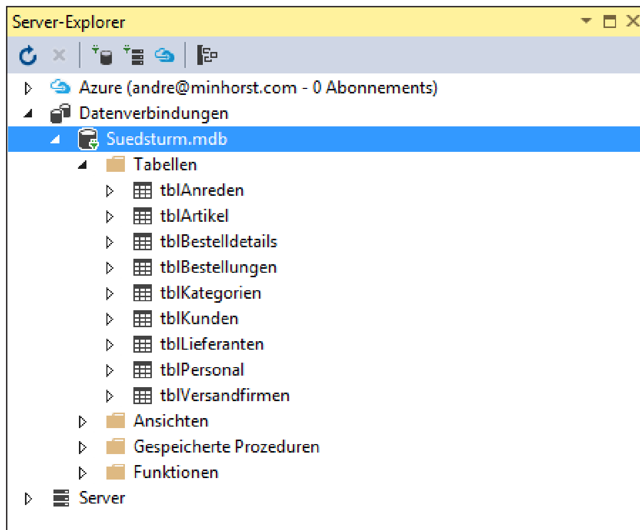


Bild 3: Die hinzugefügte Datenquelle im Server-Explorer

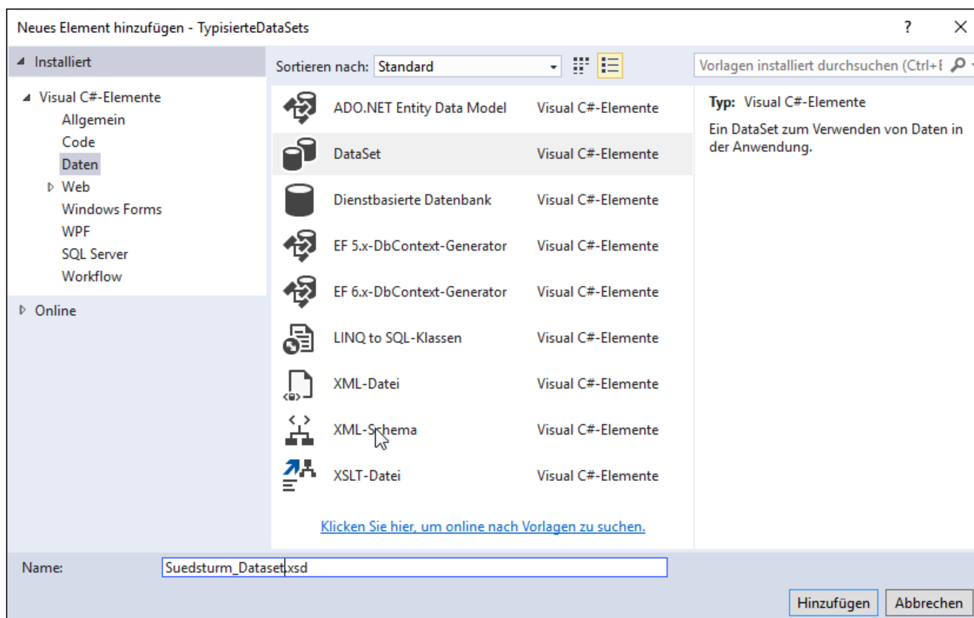


Bild 4: Hinzufügen eines **DataSet**-Objekts zum Projekt

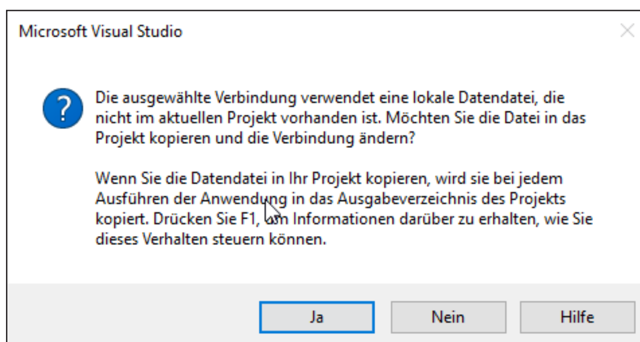


Bild 5: Meldung beim Hinzufügen der ersten Tabelle der Datenquelle

Neues Element hinzufügen. Mit diesem wählen Sie nun die Kategorie **Visual C#-Elemente|Daten** aus und klicken dann auf das Element **DataSet**. Tragen Sie als Name **Suedsturm_DataSet.xsd** ein und klicken Sie auf **Hinzufügen** (siehe Bild 4).

Das neue Objekt **Suedsturm_DataSet.xsd** wird nun im Designer in Visual Studio angezeigt. Aktivieren Sie nun den Server-Explorer, sodass dieser neben dem neuen Objekt **Suedsturm_DataSet.xsd** zu sehen ist.

Nun ziehen Sie die Tabellen **tblArtikel**, **tblLieferanten** und **tblKategorien** aus dem Server-Explorer in den Entwurf des DataSets. Direkt beim Versuch, die erste Tabelle hinüberzuziehen, sollte die Meldung aus Bild 5 erscheinen. Wenn Sie hier auf Ja klicken, wird die verwendete Datenbankdatei **Suedsturm.mdb** zum Projekt hinzugefügt.

Danach geht es los: Ziehen Sie die drei Tabellen in den Entwurf des DataSets und Sie erhalten die Konstellation aus Bild 6. Die Tabellen werden mit allen enthaltenen Feldern genau wie im Beziehungs-Fenster von Access angezeigt – nicht einmal die Beziehungspfeile fehlen. Ein wichtiger Unterschied ist jedoch, dass jede Tabelle unten ein weiteres Objekt etwa namens **tblArtikelTableAdapter** enthält, dass die beiden Methoden **Fill** und **GetData** anbietet.

Nicht alle Felder einer Tabelle verwenden

Wie Sie sehen, wurden die Tabellen vollständig in das DataSet übernommen – wie soll es auch anders sein, wenn Sie die kompletten Tabellen hinüber ziehen? Andererseits kann es ja auch einmal vorkommen, dass Sie nicht alle Felder

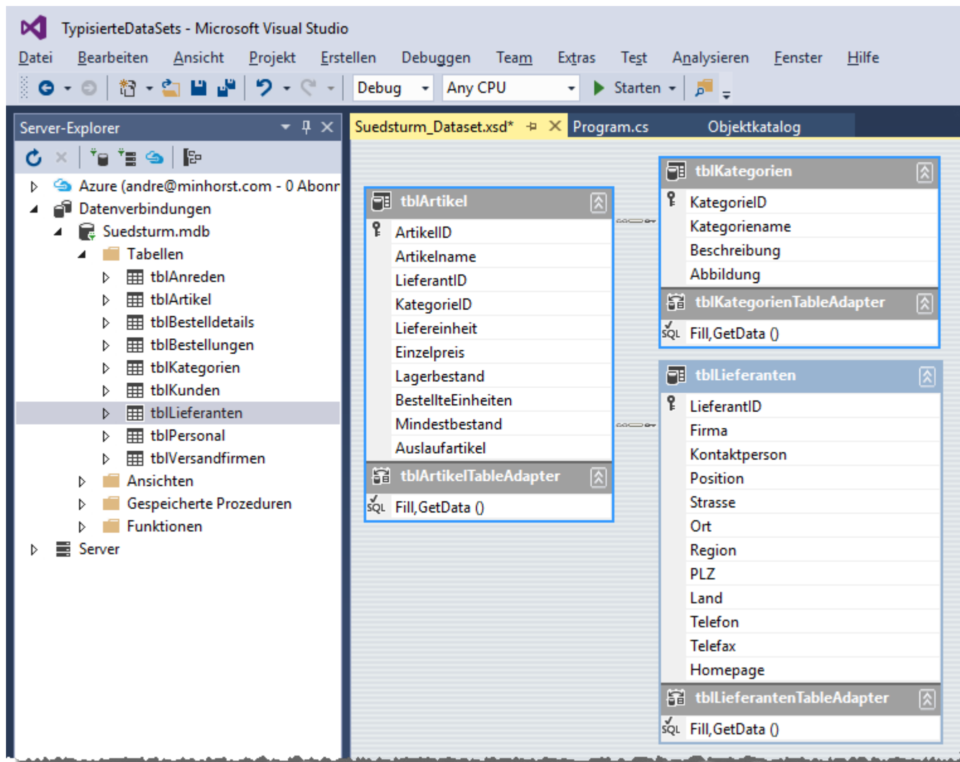


Bild 6: Das DataSet mit allen drei benötigten Tabellen

einer Tabelle verwenden möchten. In diesem Fall klappen Sie das Tabellen-Element im Server-Explorer so auf, dass Sie die einzelnen Feldnamen der Tabelle sehen. Hier wählen Sie nun entweder bei gedrückter Umschalt- oder **Strg**-Taste die gewünschten Felder aus. Bei gedrückter Umschalttaste markieren Sie alle Felder zwischen dem zuerst und dem zuletzt angeklickten Feld, mit gedrückter **Strg**-Taste aktivieren oder deaktivieren Sie einzelne Einträge. In diesem Fall wollen wir jedoch alle Felder der Tabellen nutzen.

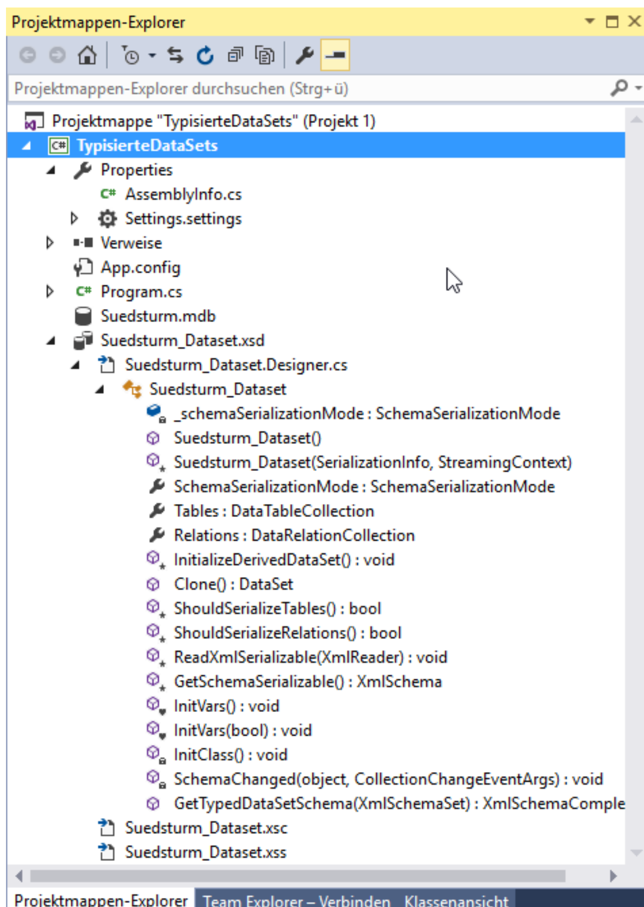


Bild 7: Neue Elemente im Projektmappen-Explorer

Vom Assistenten erzeugte Elemente

Damit ist das typisierte DataSet bereits fertiggestellt – nun schauen wir uns an, was der Assistent im Hintergrund alles für uns erledigt hat. Dazu brauchen Sie den Blick nur zum Projektmappen-Explorer zu wenden (siehe Bild 7). Hier finden Sie nun einen neuen Eintrag namens **Suedsturm_DataSet.xsd** mit dem Unterelement **Suedsturm_DataSet.Designer.cs**. Mit einem Doppelklick auf ersteren gelangen Sie später zum Designer zurück, um beispielsweise Tabellen oder Felder hinzuzufügen oder zu entfernen.

Der Eintrag **Suedsturm_DataSet.Designer.cs** zeigt auf die neu erstellte Klasse, welche im aktuellen Zustand zunächst die ein paar der Elemente bereithält, die auch ein untypisiertes **DataSet**-Objekt liefert – beispielsweise die **Tables**- oder die **Relations**-Auflistung oder auch einige Methoden. Allerdings ist hier noch nichts zu sehen von den für unsere Zusammenstellung der drei Tabellen **tblArtikel**, **tblKategorien** und **tblLieferanten** typischen Elemente. Es gibt weder einen DataAdapter noch DataTables. Das ändert sich allerdings

schlagartig, sobald Sie das typisierte DataSet beispielsweise mit der Tastenkombination **Strg + S** speichern. Dann wird Visual Studio nämlich aktiv und erweitert die Klasse um eine ganze Reihe von Elementen. Die wichtigsten zeigt Bild 8:

- **Suedsturm_Dataset** ist eine Klasse, welche von der Klasse **DataSet** abgeleitet ist. Das heißt, es enthält alle Methoden, Eigenschaften und Ereignisse dieser Klasse, fügt aber eigene Elemente hinzu oder überschreibt Elemente der Klasse. Durch das Hinzufügen oder Überschreiben von Elementen, die sich speziell auf unsere Tabellen beziehen, erhalten wir ja eben das typisierte **DataSet**-Objekt.

- Die Klassen **tblArtikelTableAdapter**, **tblKategorienTableAdapter** und **tblLieferantenTableAdapter** sind von der Klasse **TableAdapter** abgeleitet.

Die Klasse **Suedsturm_DataSet** enthält noch die drei Klassen **tblArtikelDataTable**, **tblKategorienDataTable** und **tblLieferantenDataTable**. Für den Zugriff auf die einzelnen Zeilen der Tabellen stellen die drei Klassen **tblArtikelRow**, **tblKategorienRow** und **tblLieferantenRow** für jedes Feld der ursprünglichen Tabelle jeweils eine Eigenschaft zur

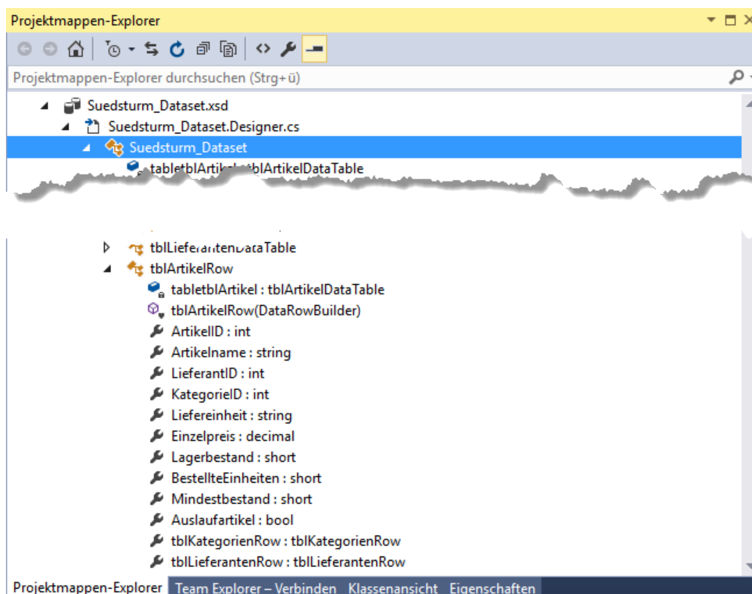


Bild 9: Für jedes einzelne Feld der beteiligten Tabellen gibt es eine in der jeweiligen Klasse definierte Eigenschaft.

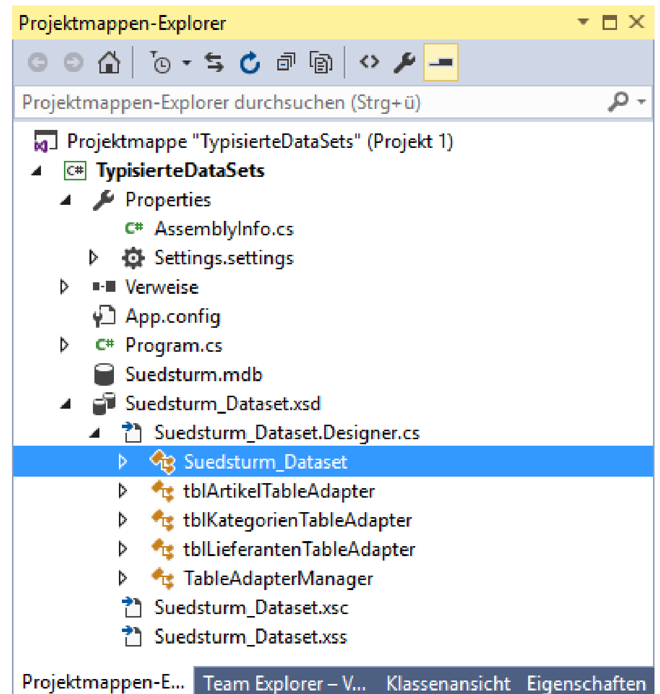


Bild 8: Die Hauptelemente des typisierten DataSets

Verfügung, über die Sie per Code auf die Feldinhalte des aktuellen Datensatzes zugreifen können (siehe Bild 9).

Beispiel der Ausgabe der Daten eines untypisierten DataSets

Schauen wir uns nun an, wie der Zugriff auf ein typisiertes DataSet im Gegensatz zu dem auf ein untypisiertes DataSet aussieht. Bei einem untypisierten DataSet würden Sie den Code aus Listing 1 verwenden. Hier stehen eine Reihe von Aufgaben an:

- Definieren des Connection-Strings

- Festlegen der SQL-Abfrage

- Deklarieren und gegebenenfalls Initialisieren von **Connection**-, **Command**-, **DataAdapter**-, **DataSet**- und **DataTable**-Objekten

- Zuweisen der **Connection**- und der **CommandText**-Eigenschaften des **Command**-Objekts

```
public static void DataSetUntypisiert() {
    string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    string strSQL = "SELECT * FROM tblArtikel";
    OleDbConnection cnn = new OleDbConnection(strConnection);
    OleDbCommand cmd = new OleDbCommand();
    OleDbDataAdapter da;
    DataSet ds = new DataSet();
    DataTable dt;
    cmd.Connection = cnn;
    cmd.CommandText = strSQL;
    da = new OleDbDataAdapter(strSQL, cnn);
    da.Fill(ds, "tblArtikel");
    dt = ds.Tables["tblArtikel"];
    foreach (DataRow dr in dt.Rows) {
        Console.WriteLine("{0} {1}", dr["ArtikelID"], dr["Artikelname"]);
    }
    Console.ReadLine();
}
```

Listing 1: Ausgabe der Daten eines untypisierten DataSets

```
public static void DataSetTypisiert() {
    Suedsturm_Dataset ds = new Suedsturm_Dataset();
    tblArtikelTableAdapter ta = new tblArtikelTableAdapter();
    ta.Fill(ds.tblArtikel);
    foreach (Suedsturm_Dataset.tblArtikelRow row in ds.tblArtikel) {
        Console.WriteLine("{0} {1}", row.ArtikelID, row.Artikelname);
    }
    Console.ReadLine();
}
```

Listing 2: Ausgabe der Daten eines typisierten DataSets

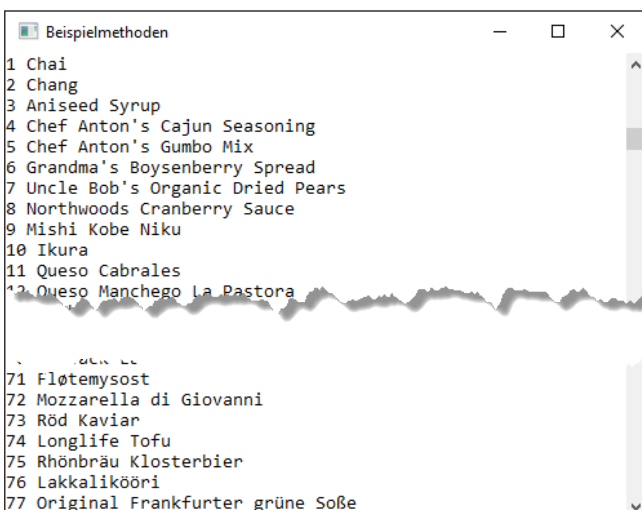


Bild 10: Ausgabe der Datensätze der Tabelle **tblArtikel** in der Console

- Erstellen und Füllen des **DataAdapter**-Objekts mit der Tabelle **tblArtikel**
- Zuweisen der Tabelle **tblArtikel** an die untypisierte **DataSet**-Variable
- Durchlaufen der **DataRow**-Objekte und Ausgabe der Werte in der Konsole wie in Bild 10 dargestellt

Beispiel der Ausgabe des typisierten DataSets

Listing 2 zeigt, wie Sie die gleiche Ausgabe der Daten der Tabelle **tblArtikel** in der Konsole mit dem typisierten DataSet erreichen. Hier haben wir mit dem Assistenten schon einige

Tipps und Tricks zu Fenstern und Steuerelementen

In dieser Reihe unserer Tipps und Tricks erfahren Sie Einiges über Fenster und Steuerelemente: Wie setzen Sie den Fokus auf ein Steuerelement? Wie legen Sie die Aktivierreihenfolge von Steuerelementen fest? Wie zentrieren Sie ein Fenster? Wie öffnen Sie ein Fenster relativ zum aufrufenden Fenster? Wie übergeben Sie Parameter beim Öffnen an ein Fenster? Wie legen Sie ein Anwendungssymbol fest, und wie eines für ein einzelnes Fenster? Mehr dazu auf den folgenden Seiten!

Fokus auf ein Steuerelement setzen

Im Gegensatz zu Access-Formularen landet der Fokus unter WPF nicht unbedingt sichtbar auf einem Steuerelement.

Daher schauen wir uns nun an, wie Sie den Fokus etwa beim Öffnen des Fensters auf das gewünschte Steuerelement verschieben. Dazu gibt es mindestens zwei Methoden: eine per XAML und eine per C#.

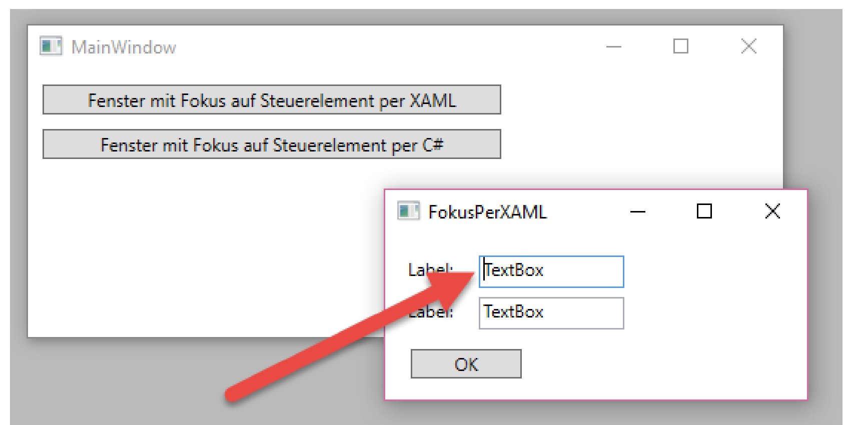


Bild 1: Öffnen eines Fensters und Setzen des Fokus

Um beide auszuprobieren, fügen wir dem Fenster **MainWindow** zwei Schaltflächen hinzu, mit denen wir die beiden weiteren Fenster **FokusPerXAML** und **FokusPerCSharp** öffnen. Beide sollen den Fokus jeweils auf das erste Textfeld verschieben (siehe Bild 1).

Unter XAML legen Sie in einem übergeordneten Element des zu betroffenen Elements fest, welches Element den Fokus erhalten soll – in diesem Fall im **Grid**-Element. Dort fügen Sie das Attribut **FocusManager.FocusedElement** hinzu und tragen als Wert **"{Binding ElementName=textBox}"** ein. Alles lässt sich einfach per IntelliSense hinzufügen:

```
<Window x:Class="TippsControls.FokusPerXAML" ...
                                    Title="FokusPerXAML">
    <Grid FocusManager.FocusedElement=
                         "{Binding ElementName=textBox}">
        <Label x:Name="label" Content="Label:" .../>
        <TextBox x:Name="textBox" Text="TextBox" .../>
        ...
    </Grid>
</Window>
```

Das Textfeld **textBox** erhält dann gleich nach dem Öffnen des Fensters den Fokus.

Unter C# ist es noch etwas einfacher. Hier definieren Sie mit dem Attribut **Loaded** den Ereignishandler, der beim Laden des Fensters aufgerufen werden soll:

```
<Window ... Loaded="Window_Loaded">
    ...
</Window>
```

Dann füge Sie der entsprechenden Methode eine einzige Anweisung hinzu, welche die Methode **Focus** des Steuerelements aufruft:

```
private void Window_Loaded(...) {
    this.textBox.Focus();
```

```
}
```

Damit verschiebt auch diese Variante beim Öffnen den Fokus auf das erste Textfeld.

Aktivierreihenfolge festlegen

Unter Access haben Sie die Aktivierreihenfolge in einem speziell dafür vorgesehenen Dialog festgelegt. Unter WPF läuft das etwas anders. Wenn Sie nichts weiter tun, entspricht die Aktivierungsreihenfolge der Reihenfolge, in der Sie die Elemente im XAML-Code angelegt haben. Solange diese Reihenfolge der gewünschten Aktivierreihenfolge entspricht, ist alles okay. Wenn Sie aber nun beispielsweise ein neues Textfeld plus Bezeichnungsfeld (**Label**) aus der Toolbox zwischen zwei vorhandene Textfelder einfügen, funktioniert die Aktivierreihenfolge nicht mehr wie gewünscht. Nun wird erst das erste, dann das dritte und schließlich das zweite Textfeld aktiviert. Um dies zu ändern, müssten Sie also die Reihenfolge der Elemente im XAML-Code anpassen.

Davon abgesehen, dass ein wenig Ordnung im XAML-Code nicht schaden kann, kann es natürlich Gründe geben, eine Aktivierreihenfolge vorzugeben, die nicht von der Gestaltung des XAML-Codes abhängt. Für solche Fälle bietet WPF das Attribut **KeyboardNavigation.TabIndex** an, dass Sie für alle relevanten Steuerelemente hinzufügen:

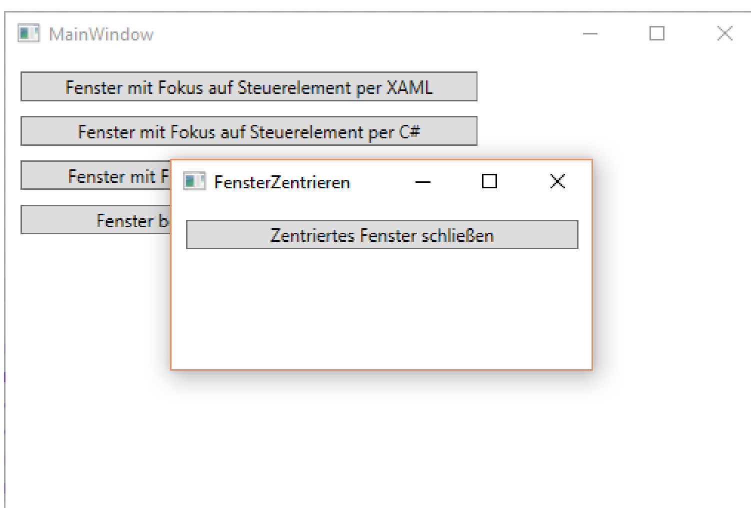


Bild 2: Ein zentriert geöffnetes Fenster

```
<Window x:Class="TippsControls.Aktivierreihenfolge" ...>
    <Grid>
        <Label x:Name="label1" .../>
        <TextBox x:Name="txt1" ...
            KeyboardNavigation.TabIndex="1" />
        <Label x:Name="label2" .../>
        <TextBox x:Name="txt2" ...
            KeyboardNavigation.TabIndex="2"/>
        ...
    </Grid>
</Window>
```

Soll ein Steuerelement gar nicht in die Aktivierreihenfolge einbezogen werden, stellen Sie das Attribut **IsTabStop** auf den Wert **False** ein:

```
<TextBox x:Name="txt3" ... IsTabStop="False" />
```

Der Standardwert dieses Attributs lautet **True**.

Fenster zentrieren

Ohne weiteres Zutun öffnen sich die Fenster einer WPF-Anwendung irgendwo, zum Beispiel im linken, oberen Bereich des Bildschirms. Wenn Sie möchten, dass sich Fenster genau in der Mitte des Bildschirms öffnen, können Sie dies mit einer einfachen Einstellung erreichen. Dazu legen Sie im XAML-Code einfach den Wert des Attributs **WindowStartupLocation** für das **Window**-Element auf **CenterScreen** fest:

```
<Window ... Title="FensterZentrieren"
    Height="150" Width="300"
    WindowStartupLocation="CenterScreen">
```

Wenn Sie dies auch noch für ein vom Hauptfenster geöffnetes Fenster erledigen, wird dieses zentriert über dem anderen Fenster geöffnet (siehe Bild 2).

Das Attribut **WindowStartupLocation** hat drei mögliche Werte:

- **CenterScreen**: Fenster wird zentriert bezogen auf den Bildschirm angezeigt.
- **CenterOwner**: Fenster wird zentriert bezogen auf das aufrufende Fenster geöffnet.
- **Manual**: Fenster wird entsprechend der Werte für **Left** und **Top** geöffnet.

Fenster relativ zum Hauptfenster öffnen

Unter Access und VBA war das Öffnen eines Formulars relativ zur Position des aufrufenden Formulars erst nachträglich durch den Einsatz der Methode **DoCmd.MoveSize** möglich. Unter C# und WPF gelingt dies wesentlich einfacher. Das liegt daran, dass wir hier im Gegensatz zur unter Access üblichen Vorgehensweise mit der **DoCmd.OpenForm**-Methode das zu öffnende Fenster als Objekt betrachten und dieses erst deklarieren und initialisieren und dann erst öffnen. Unter Access war dies zwar auch mit Formularen möglich, aber wenig verbreitet.

Dadurch, dass wir ein Fenster unter C# zunächst deklarieren und initialisieren, können wir über die Objektvariable auf seine Eigenschaften zugreifen und dementsprechend auch direkt etwa seine Position oder Größe einstellen. Hinzu kommt, das Hauptfenster einer WPF-Anwendung sowie die davon geöffneten weiteren Fenster jeweils eigene Windows-Fenster sind, und nicht wie unter Access Objekte innerhalb des Access-Fensters. Dies erleichtert das ermitteln einer relativen Position für das zu öffnende Fenster. Schauen wir uns an, wie dies gelingt – in diesem Fall durch einen Klick auf die Schaltfläche **btnFensterMitPositionRelativ1**:

```
private void btnFensterMitPositionRela-
tiv1_Click(
        object sender, RoutedEventArgs
e) {
    FensterMitRelativerPosition wnd =
        new FensterMitRelativerPosition();
    wnd.Left = this.Left + 200;
    wnd.Top = this.Top + 100;
    wnd.ShowDialog();
}
```

Die Methode deklariert und intialisiert ein Objekt auf Basis der Fensterklasse **FensterMitRelativerPosition**, dass wir zuvor als einfaches, leeres Fenster erstellen, und speichert den Objektverweis in der Variablen **wnd**. Dann weist es den beiden Eigenschaften **Left** und **Top** jeweils einen Wert zu, der um **200** größer als die entsprechenden Eigenschaften des aufrufenden Fensters sind. Danach öffnet die Methode das Fenster mit der **ShowDialog**-Methode (**Show** würde es an dieser Stelle auch tun). Das Ergebnis sehen Sie in Bild 3.

Eine zweite Variante unterscheidet sich in der Art der Zuweisung der Werte zu den Eigenschaften **Top** und **Left**. Diese hinterlegen wir für die Schaltfläche **btnFensterMitPositionRelativ2**:

```
private void btnFensterMitPositionRelativ2_Click(
        object sender, RoutedEventArgs e) {
```

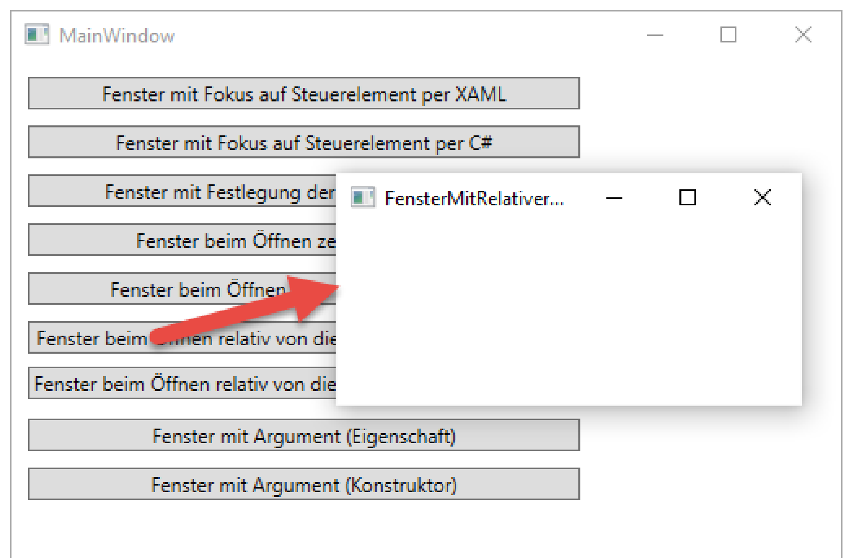


Bild 3: Öffnen eines Fensters, das um 200 Punkte nach rechts und um 100 Punkte nach unten verschoben ist