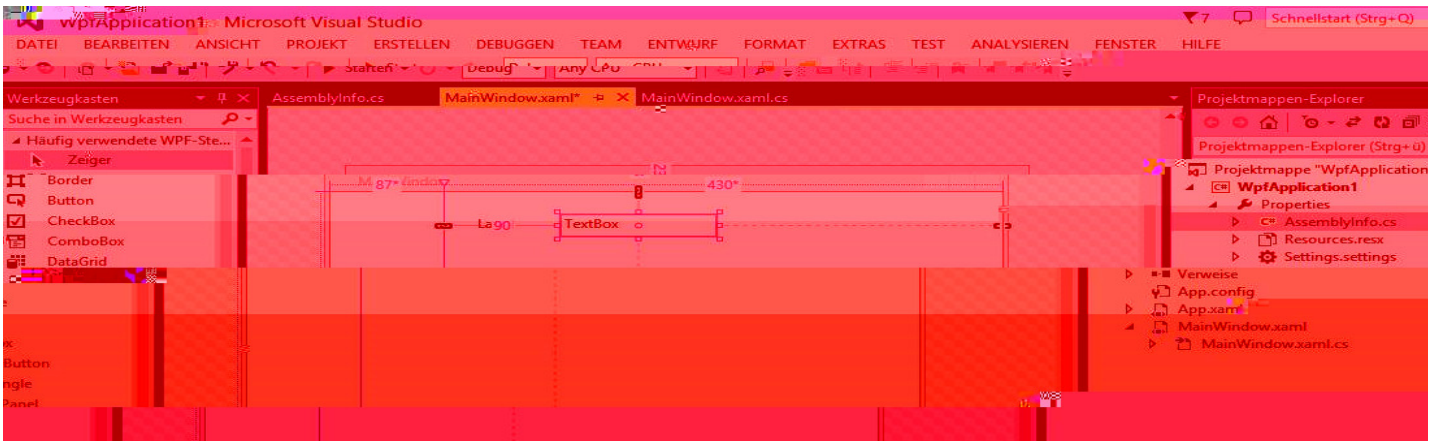
AUSGABE 06/2016

DATENBANK ENTWICKLER

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



TOP-THEMEN:

André Minhorst Verlag

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

Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

WPF-Steuerelemente: Das DockPanel

Das DockPanel-Element ist eines der Steuerelemente, mit dem sich untergeordnete Steuerelemente ausrichten lassen. Das interessante an diesem Steuerelement ist, dass die enthaltenen Elemente jeweils an der angegebenen Seite des DockPanels angedockt werden. Damit lassen sich interessante Anordnungen erzielen. Welche das sind und wie Sie diese realisieren, erfahren Sie in diesem Artikel.

Grundlagen

Neben dem **DockPanel** gibt es noch einige weitere Elemente zum Anordnen der untergeordneten Elemente. Das bekannteste ist wohl das **Grid**-Element, das ja auch standardmäßig als einziges untergeordnetes Element in neuen **Window**- oder **Page**-Elementen angezeigt wird. Während sich mit dem **Grid**-Steuerelement ein Raster, bestehend aus Zeilen und Spalten, realisieren lässt, bietet das **DockPanel** ganz andere Möglichkeiten.

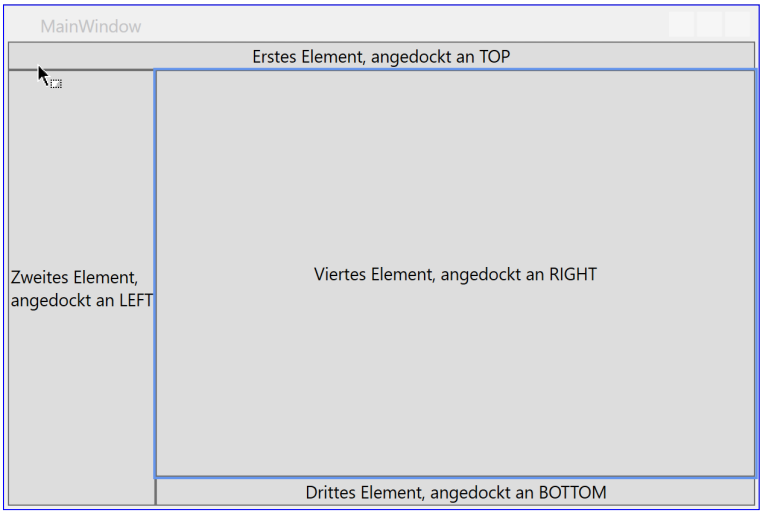


Bild 1: Beispiel für vier an den verschiedenen Seiten angedockte Steuerelemente

Sie können diesem beliebig viele Unterelemente zuweisen, für die Sie jeweils angeben, ob diese oben, unten, links oder rechts innerhalb des DockPanels angedockt werden sollen. Das angegebene Element nimmt dann, wenn es links oder rechts angeordnet wurde, jeweils die gesamte Höhe ein, und wenn es oben oder unten angeordnet wurde, die gesamte Breite.

Die Reihenfolge entscheidet dann darüber, welchem Element welcher Platz zufällt. Wie dies aussieht, zeigen wir anhand von Beispielen. Im ersten Fall haben wir vier **Button**-Elemente angeordnet, die über die Eigenschaft **DockPanel.Dock** mit den Werten **Top**, **Left**, **Bottom** und **Right** versehen wurden (siehe Beispielprojekt unter **Beispiel #1**) – siehe Listing 1.

Bild 1 zeigt, wie das Ergebnis im Fenster aussieht. Das erste Element landet oben und nimmt die komplette Breite ein. Das zweite Element landet links, erhält aber nur noch die Höhe, die das erste Element übrig lässt. Die folgenden Elemente führen dies fort, bis das vierte Element den verbleibenden Platz einnimmt. Das Verhalten des letzten Elements lässt sich übrigens durch das Attribut **LastChildFill** des **DockPanel**-Elements beeinflussen. Stellen Sie dieses auf den Wert **False** ein, nimmt das letzte Element nur den Platz ein, den der Inhalt erfordert (siehe Beispielprojekt unter **Beispiel #2**):

```
<DockPanel>
    <Button DockPanel.Dock="Top" Content="Erstes Element, angedockt an TOP"></Button>
    <Button DockPanel.Dock="Left" Content="Zweites Element, &#xa;angedockt an LEFT"></Button>
    <Button DockPanel.Dock="Bottom" Content="Drittes Element, angedockt an BOTTOM"></Button>
    <Button DockPanel.Dock="Right" Content="Viertes Element, angedockt an RIGHT"></Button>
</DockPanel>
```

Listing 1: Code für vier Elemente, die an den vier Seiten eines Fensters angedockt werden

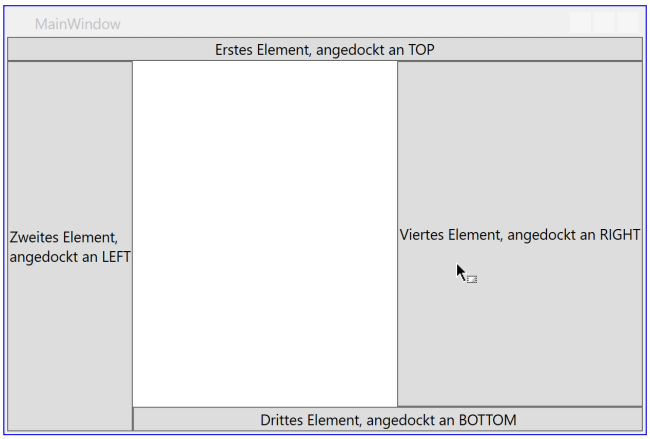


Bild 2: Leerraum durch das Attribut **LastChildFill**

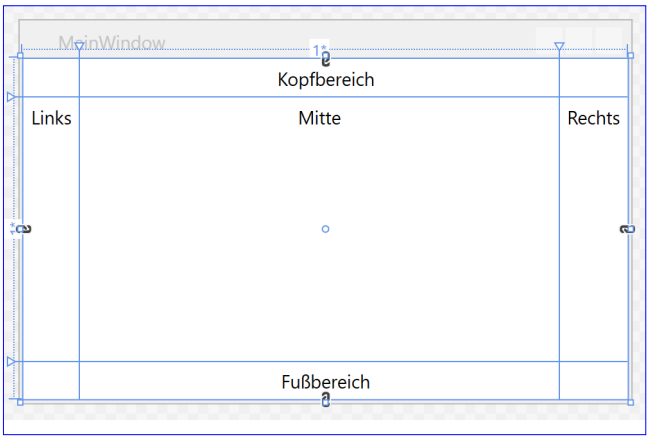


Bild 3: Herstellung eines ähnlichen Layouts mit dem **DockPanel**-Element

```
<DockPanel LastChildFill="False">
    ...
</DockPanel>
```

Dadurch entsteht ein Leerraum wie in Bild 2.

Struktur durch passende Reihenfolge

Vielleicht möchten Sie einmal eine Anordnung erstellen, die zum Beispiel einen Kopfbereich und einen Fußbereich, jeweils über die vollständige Fensterbreite, und dazwischen drei Spalten über die verbleibende Höhe erhalten. Wenn Sie dies mit einem **Grid**-Element erledigen wollen, sieht der Code wie in Listing 2 aus (siehe Beispielprojekt, **Beispiel #3**).

Das Ergebnis erhalten Sie in Bild 3.

Mit dem **DockPanel**-Element lässt sich eine ähnliche Anordnung mit viel weniger XAML-Code herstellen (siehe Listing 3 und im Beispielprojekt unter **Beispiel #4**).

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Grid.ColumnSpan="3" HorizontalAlignment="Stretch"
        Height="25" HorizontalContentAlignment="Center">Kopfbereich</Label>
    <Label Grid.Row="2" Grid.ColumnSpan="3" HorizontalAlignment="Stretch"
        Height="25" HorizontalContentAlignment="Center">Fußbereich</Label>
    <Label Grid.Row="1" Grid.Column="0" HorizontalAlignment="Center">Links</Label>
    <Label Grid.Row="1" Grid.Column="1" HorizontalAlignment="Center">Mitte</Label>
    <Label Grid.Row="1" Grid.Column="2" HorizontalAlignment="Center">Rechts</Label>
</Grid>
```

Listing 2: Code für die Herstellung einer bestimmten Anordnung per **Grid**-Element

Hier werden also schlicht zunächst ein **Label**-Element an der oberen Seite, eines an der unteren Seite und dann drei Elemente an der linken Seite des **DockPanel**-Elements angedockt. Je nach Anforderung können Sie Anordnungen mit dem **DockPanel**-Element also leichter als mit alternativen Elementen herstellen. Das Ergebnis zeigt Bild 4.

Wenn Sie hier nun noch eine dynamische Aufteilung der Breite der Spalten zwischen dem Kopfbereich und dem Fußbereich erhalten wollen, können Sie das

WPF-Trigger

Trigger kennt der geneigte Access-Anwender höchstens vom SQL Server, wo diese die Möglichkeit boten, auf Änderungen in den Tabellen zu reagieren, für die sie angelegt wurden. Ein Konstrukt namens Trigger gibt es aber auch unter WPF. Dort sind Trigger nicht an Tabellen gebundene Algorithmen, sondern Definitionen von Aktionen, die in Zusammenhang mit Datenänderungen ausgelöst werden. Dieser Artikel stellt Trigger und ihre Anwendungsmöglichkeiten vor.

Trigger-Arten

Es gibt drei Arten von Triggern, von denen zwei aktuell für uns interessant sind. Der Vollständigkeit halber sollen hier jedoch dennoch alle drei Arten aufgeführt werden:

- Eigenschaftstrigger (Property Trigger): Werden ausgelöst, wenn sich der Wert einer Eigenschaft ändert. Arbeitet nur mit Dependency Properties. Kann nur Eigenschaften des betroffenen Elements ändern und dabei nur auf Eigenschaften des gleichen Elements zugreifen.

- Datentrigger (Data Trigger): Wie Eigenschaftstrigger, allerdings auch für andere Eigenschaften als Dependency Properties. Als Quelle kann auch ein anderes Element als das mit dem Trigger versehene Element dienen. Die wichtigste Eigenschaft von Dependency Properties, die hier interessant ist, ist die automatische Information von daran gebundenen Elementen über einen geänderten Zustand des Wertes der Property.

- Ereignistrigger (Event Trigger): Werden zum Beispiel verwendet, um Animationen zu starten und zu beenden, wenn bestimmte Ereignisse eintreten. Um diese Art von Triggern kümmern wir uns in einem späteren Artikel.

Eigenschaftstrigger (Property Trigger)

Gleich zu Beginn zeigen wir Ihnen, dass Sie Trigger ganz einfach statt einfacher C#-Ereignismethoden einsetzen können. Im Beispiel wollen wir die aktive Schaltfläche jeweils mit einem hellgelben Hintergrund versehen. Dazu definieren wir unseren XAML-Code wie folgt (siehe Beispielprojekt, Seite **TriggerPerCSharpEreignis.xaml**):

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <Label Content="Vorname:" Margin="5"></Label>
    <TextBox x:Name="txtVorname" ... GotFocus="txtVorname_GotFocus" LostFocus="txtVorname_LostFocus"></TextBox>
    <Label Content="Nachname:" Margin="5" Grid.Row="1"></Label>
```

```
    <TextBox x:Name="txtNachname" ... GotFocus="txtNachname_GotFocus" LostFocus="txtNachname_LostFocus"></TextBox>
</Grid>
```

Für die vier für die beiden Textfelder definierten Ereignisattribute hinterlegen wir die folgenden vier Ereignismethoden im Code behind-Modul:

```
private void txtVorname_GotFocus(object sender, RoutedEventArgs e) {
    txtVorname.Background = Brushes.LightYellow;
}
private void txtVorname_LostFocus(object sender, RoutedEventArgs e) {
    txtVorname.Background = null;
}
private void txtNachname_GotFocus(object sender, RoutedEventArgs e) {
    txtNachname.Background = Brushes.LightYellow;
}
private void txtNachname_LostFocus(object sender, RoutedEventArgs e) {
    txtNachname.Background = null;
}
```

Dies sorgt dafür, dass das jeweilige Textfeld bei Fokuserhalt jeweils auf die Hintergrundfarbe **LightYellow** eingestellt wird. Bei Fokusverlust leeren wir die Eigenschaft **Background** wieder, indem wir diese auf den Wert **null** einstellen.

Dies ergibt ein Fenster wie in Bild 1.

Bild 1: Fenster mit zwei Textfeldern, von denen das aktive jeweils einen gelben Hintergrund erhält

Eigenschaftsänderung per XAML

Dies können Sie auch ohne C#-Code erreichen, allerdings mit etwas mehr Schreibaufwand. Ändern Sie den Code für die erste TextBox wie folgt (Beispielprojekt, Seite **PropertyTrigger.xaml**):

```
<TextBox x:Name="txtVorname" Grid.Row="1" Grid.Column="1" Margin="5">
    <TextBox.Style>
        <Style TargetType="{x:Type TextBox}">
            <Style.Triggers>
                <Trigger Property="IsFocused" Value="True">
                    <Setter Property="Background" Value="LightYellow"></Setter>
                </Trigger>
            </Style.Triggers>
        </Style>
    </TextBox.Style>
</TextBox>
```

Die wesentlichen Elemente hier sind die folgenden:

```
<Trigger Property="IsFocused" Value="True">
    <Setter Property="Background" Value="LightYellow"></Setter>
</Trigger>
```

Wir setzen einen Trigger, der ausgelöst werden soll, wenn die Eigenschaft **IsFocused** den Wert **True** enthält. Dann soll eine weitere Eigenschaft namens **Background** auf den Wert **LightYellow** eingestellt werden. Das Drumherum benötigen wir, um den Trigger für die Textbox verfügbar zu machen. In diesem Fall legen wir ein **Style**-Objekt für die **Style**-Eigenschaft der Textbox an (**TextBox.Style**), der wir in der **Triggers**-Auflistung den obigen Trigger bekannt machen.

Wenn wir der zweiten Textbox genau den gleichen Trigger unterschieben, erhalten wir genau das gleiche Verhalten wie beim ersten Beispiel, wo wir die Änderung des Hintergrunds noch mit C#-Ereignismethoden realisiert haben.

Welche Vorteile liefert uns diese Vorgehensweise? Aktuell nur den, dass wir das Code behind-Modul nicht aufblasen. Stattdessen haben wir aber einigen Code mehr im **.xaml**-Modul. Bei beiden Varianten hätten wir noch viel mehr Code, wenn wir dieses Verhalten für weitere Textfelder implementieren wollten. Das können wir aber in beiden Fällen ändern.

Ein weiterer Vorteil ist, dass die Eigenschaften immer überwacht werden und der Zustand auch wieder rückgängig gemacht wird, wenn die Bedingung nicht mehr erfüllt ist. Verliert das erste **TextBox**-Element also den Fokus, ist **IsFocused** nicht mehr **True**, also wird auch der Wert für die dadurch geänderte Eigenschaft **Background** wieder zurückgenommen.

Property-Trigger per Ressource

Wenn wir davon ausgehen, dass alle **TextBox**-Elemente eines Fensters (oder, wie in diesem Fall, einer Seite) das gleiche Verhalten implementieren sollen, können wir den Style mit dem Trigger auch in den Ressourcen des Fensters/der Seite festlegen. Die Trigger-Definitionen für die einzelnen Steuerelemente können Sie dann entfernen. Der Vorteil ist, dass Sie nun beliebig viele Textfelder hinzufügen können – alle werden beim Fokuserhalt gelb hinterlegt. Der benötigte Code sieht so aus (siehe Beispielprojekt, Seite **PropertyTrigger_Resource.xaml**) und wird direkt unterhalb des **Page**-Elements eingefügt:

```
<Page.Resources>
    <Style TargetType="{x:Type TextBox}">
        <Style.Triggers>
            <Trigger Property="IsFocused" Value="True">
                <Setter Property="Background" Value="LightYellow"></Setter>
            </Trigger>
        </Style.Triggers>
    </Style>
</Page.Resources>
```

Data Trigger

Gleich, ob Sie die oben vorgestellten Trigger direkt für ein Steuerelement angegeben oder diese für alle Steuerelemente eines bestimmten Typs hinterlegt haben, haben Sie eine entscheidende Einschränkung: Sie können nur die Eigenschaften des

Steuerelements abfragen, für das Sie den Trigger definiert haben. Es ist also nicht möglich, beispielsweise den Inhalt eines anderen Steuerelements zu ermitteln und in Abhängigkeit davon eine Eigenschaft eines anderen Steuerelements einzustellen.

Außerdem gelingt der Einsatz von Property Triggern nur mit bestimmten Eigenschaften, und zwar mit Dependency Properties. Wir aber wollen uns nun ein Beispiel ansehen, bei dem wir einen Data Trigger einsetzen.

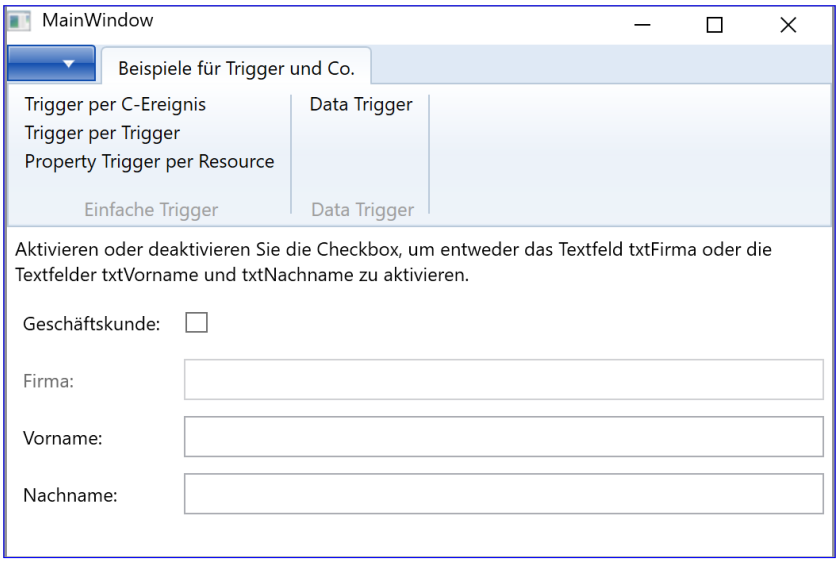


Bild 2: Von einer Checkbox abhängige Textfelder

Dabei soll eine Checkbox aktiviert oder deaktiviert werden, um anzugeben, ob es sich bei einem Kunden um einen Geschäftskunden handelt. Ist die Checkbox aktiviert, soll das Feld **Firma** aktiviert sein und die beiden Felder **Vorname** und **Nachname** nicht und umgekehrt (siehe Bild 2 und Beispielprojekt, Seite **DataTrigger.xaml**).

Man könnte dies per C# lösen, indem man für die Checkbox das Ereignis **Checked** implementiert und dort die Eigenschaft **IsEnabled** der betroffenen Bezeichnungsfelder und Textfelder anpasst. Das widerspricht aber der gewünschten Vorgehensweise, wonach die Benutzeroberfläche und die Anwendungslogik voneinander entkoppelt sein sollen. Aber mit Data Triggern lässt sich die hier gewünschte Lösung sogar abbilden, ohne überhaupt eine Zeile C#-Code zu benötigen. Bisher sehen die Definitionen der Steuerelemente in gekürzter Form so aus:

```
<Label Content="Geschäftskunde:" ... />
<CheckBox x:Name="chkGeschaeftskunde" ... />
<Label Content="Firma:" ... />
<TextBox x:Name="txtFirma" ... />
<Label Content="Vorname:" ... />
<TextBox x:Name="txtVorname" ... />
<Label Content="Nachname:" ... />
<TextBox x:Name="txtNachname" ... />
```

Nun wollen wir ein paar Zeilen XAML-Code hinzufügen, damit die Felder **Firma**, **Vorname** und **Nachname** bei Änderungen des Wertes der Checkbox **chkGeschaeftskunde** aktiviert beziehungsweise deaktiviert werden. Die erste Textbox **txtFirma** ergänzen wir dazu um die folgenden Elemente:

```
<TextBox x:Name="txtFirma" Grid.Row="2" Grid.Column="1" Margin="5">
    <TextBox.Style>
        <Style TargetType="TextBox">
            <Style.Triggers>
```

Abhängige Eigenschaften per Binding

Im Artikel Trigger haben Sie erfahren, dass Sie Eigenschaften von Elementen abhängig von der Änderung anderer Eigenschaften ebenfalls ändern können. Dies gelingt, mitunter über kleine Umwege, auch mithilfe von Bindungen zwischen den Steuerelementen. So können Sie beispielsweise ganz einfach definieren, dass eine Eigenschaft den Wert True erhält, wenn eine Eigenschaft eines anderen Elements auch diese Eigenschaft annimmt. Soll hingegen der Wert False übernommen werden, wenn die andere Eigenschaft True lautet, wird es kompliziert – dann kommt ein Converter ins Spiel. Dieser Artikel zeigt die Möglichkeiten für den Ersatz von Triggern durch Binding auf.

Beispielprojekt

Wir verwenden hier wieder das Beispiel, das auch im Artikel **Trigger** zum Einsatz kam: Wir wollen also in Abhängigkeit vom Zustand eines **CheckBox**-Elements drei Textfelder aktivieren/deaktivieren. Der Aufbau der vier beteiligten Steuerelemente sieht, bevor wir die benötigten Bindungen hinzufügen, wie folgt aus (siehe Bild 1):

Bild 1: Abhängigkeit per Binding realisieren

```
<CheckBox x:Name="chkGeschaeftskunde" Margin="5" Grid.Column="1" VerticalAlignment="Center" />
<TextBox x:Name="txtFirma" Grid.Row="2" Grid.Column="1" Margin="5" />
<TextBox x:Name="txtVorname" Grid.Row="3" Grid.Column="1" Margin="5" />
<TextBox x:Name="txtNachname" Grid.Row="4" Grid.Column="1" Margin="5" />
```

Binding für den gleichen Boolean-Wert

Ganz einfach ist die Lösung, wenn das Ändern einer Eigenschaft eines Elements eine Eigenschaft des anderen Elements mit dem gleichen Datentyp auslösen soll und die Werte der beiden Eigenschaften gleich sein sollen. Das ist bei uns der Fall, wenn wir das **CheckBox**-Element **chkGeschaeftskunde** und das **TextBox**-Element **txtFirma** betrachten: Wenn die Eigenschaft **IsChecked** von **chkGeschäftskunde** den Wert **True** hat, soll auch die Eigenschaft **IsEnabled** von **txtFirma** den Wert **True** aufweisen und umgekehrt. Beide Attribute haben den Datentyp **Boolean**. In diesem Fall können wir den Wert der Eigenschaft **IsChecked** von **chkGeschaeftskunde** direkt per Binding der Eigenschaft **IsEnabled** von **txtFirma** zuweisen:

```
<TextBox x:Name="txtFirma" ... IsEnabled="{Binding ElementName=chkGeschaeftskunde, Path=IsChecked}" />
```

Wann immer der Benutzer nun das Kontrollkästchen mit einem Haken versieht, wird auch das Textfeld aktiviert.

Binding für negierte Boolean-Werte

Bei dem Kontrollkästchen und den beiden übrigen Textfeldern wird es etwas komplizierter: Wir wollen ja nicht den gleichen Wert zuweisen, sondern den negierten Wert – hat die Eigenschaft **IsChecked** von **chkGeschaeftskunde** den Wert **True**, soll **IsEnabled** von **txtVorname** und **txtNachname** den Wert **False** aufweisen und umgekehrt. In einem Binding können wir aber nicht einfach wie unter C# den Operator **Not** verwenden, um einen **Boolean**-Wert umzukehren. Hier benötigen wir einen Converter,

der in Form einer eigenen Klasse mit zwei Methoden daherkommt.

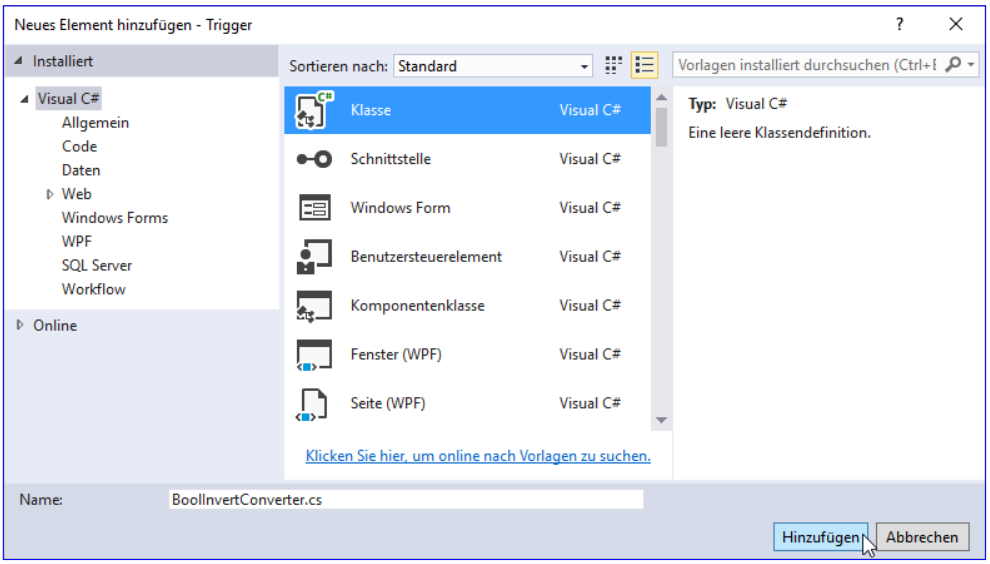


Bild 2: Anlegen der Converter-Klasse

Ein Converter kann einfache Aufgaben übernehmen wie etwa das Umwandeln von **True** in **False** und umgekehrt, aber auch komplexere Operationen durchführen. In diesem Fall wollen wir eine **Converter**-Klasse namens **BoolInvertConverter** erstellen, indem wir per **Strg + Umschalt + A** den Dialog **Neues Element hinzufügen** öffnen, dort den Eintrag **Klasse** auswählen und den Namen **BoolInvertConverter.cs** angeben, bevor wir auf **Hinzufügen** klicken (siehe Bild 2). Nach dem Hinzufügen legen Sie in der Klasse noch einen Verweis auf einen Namespace an, der die gleich verwendete Schnittstelle **IValueConverter** nutzt:

```
using System.Windows.Data;
```

Für die Implentierung eines Converters gibt es bestimmte Regeln, die sich in Form einer Schnittstellendefinition niederschlagen. In der neu erstellten Klasse fügen Sie nun also durch einen Doppelpunkt getrennt hinter dem Klassennamen die Schnittstelle **IValueConverter** hinzu:

```
public class BoolInvertConverter : IValueConverter {
}
```

IValueConverter wird nun rot unterstrichen dargestellt, was daran liegt, dass wir die Member der Schnittstelle noch nicht implementiert haben. Das holen Sie am einfachsten nach, indem Sie mit der rechten Maustaste auf den Namen der Schnittstelle klicken, den Kontextmenü-Eintrag **Schnellaktionen und Refactorings...** auswählen und im nun erscheinenden Popup auf **Schnittstelle implementieren** klicken. Dies fügt die folgenden beiden Methoden zur Klasse hinzu:

```
public object Convert(object value, Type targetType, object parameter, CultureInfo culture) {
    throw new NotImplementedException();
}
public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture) {
    throw new NotImplementedException();
}
```

Wir benötigen nur die erste Methode **Convert** und ergänzen den Code dieser Methode wie folgt:

```
public object Convert(object value, Type targetType, object parameter, CultureInfo culture) {
    bool boolValue = (bool)value;
    return !boolValue;
}
```

Hier nutzen wir den ersten Parameter der Methode namens **value**, welcher standardmäßig den zu konvertierenden Wert liefert, und konvertieren den Inhalt mit **(bool)value** in einen **Boolean**-Wert. Dieser landet in der **Boolean**-Variablen **boolValue**. Diesen negieren wir mit dem **!**-Operator und geben diesen mit **return** zurück. Damit können Sie die Klasse bereits speichern und schließen.

Converter verfügbar machen

Bevor wir den Converter einsetzen können, fehlt noch ein kleiner Schritt: Wir müssen die Converter-Klasse in der XAML-Datei, die darauf zugreifen soll, bekannt machen. Das können wir zum Beispiel gleich in der betroffenen Datei machen, in diesem Fall in der Datei **MainWindow.xaml**. Im Beispiel befindet sich die **Converter**-Klasse **BoolInvertConverter** im gleichen Namespace wie der Rest der Projektdateien, daher reicht der hier vorliegende Verweis auf den Namespace per **xmlns:local** in Verbindung mit der Angabe der Klasse unter **Window.Resources** aus. Wir binden die Klasse hier mit **local:BoolInvertConverter** ein und weisen dieser mit **x:Key="BoolInvertConvert"** einen gleichnamigen Schlüssel zu, über den wir später auf den Converter zugreifen können:

```
<Window x:Class="BindingConverters.MainWindow" ... xmlns:local="clr-namespace:BindingConverters"... >
    <Window.Resources>
        <local:BoolInvertConverter x:Key="BoolInvertConverter" />
    </Window.Resources>
    ...
</Window>
```

Boolean-Converter einsetzen

Der Einsatz dieser **Converter**-Klasse ist sehr einfach. Wir binden beispielsweise das Steuerelement **txtVorname** wie bereits zuvor **txtFirma** über das Attribut **IsChecked** (**Path=IsChecked**) an das Element **chkGeschaeftskunde** (**ElementName=chkGeschaeftskunde**). Zusätzlich geben wir über das Attribut **Converter** noch den folgenden Ausdruck an:

```
Converter={StaticResource BoolInvertConverter}}
```

Die beiden Steuerelemente **txtVorname** und **txtNachname** sehen nun also wie folgt aus und werden so auch wie gewünscht aktiviert und deaktiviert:

```
<TextBox x:Name="txtVorname"
IsEnabled="{Binding ElementName=chkGeschaeftskunde, Path=IsChecked, Converter={StaticResource BoolInvertConverter}}" />
<TextBox x:Name="txtNachname"
IsEnabled="{Binding ElementName=chkGeschaeftskunde, Path=IsChecked, Converter={StaticResource BoolInvertConverter}}" />
```

Basics: PropertyChanged

Unter Access/VBA haben Sie Tabellen direkt an Formulare und Steuerelemente gebunden, Änderungen an den Daten wurden regelmäßig auch im Frontend aktualisiert. Unter C#/WPF sieht das ganz anders aus: Hier landen die Daten aus der Tabelle erstmal in Objekten und deren Eigenschaften werden mit Steuerelementen wie TextBox, ComboBox und so weiter angezeigt. Damit sich eine Änderung am zugrunde liegenden Objekt auch in der Benutzeroberfläche manifestiert, sind ein paar zusätzliche Handgriffe nötig.

Beispiel: Einfache Kundenklasse

Um uns diese Handgriffe anzusehen, entwerfen wir ein ganz einfaches Beispiel. In diesem legen Sie in einem neuen, leeren Projekt des Typs **Visual C#\WPF-Anwendung** namens **PropertyChanged** eine neue Klasse namens **Kunde.cs** an und füllen das dortige **Namespace**-Element **PropertyChanged** wie folgt:

```
public class Kunde {
    string vorname;
    string nachname;
    public string Vorname {
        get { return vorname; }
        set { vorname = value; }
    }
    public string Nachname {
        get { return nachname; }
        set { nachname = value; }
    }
}
```

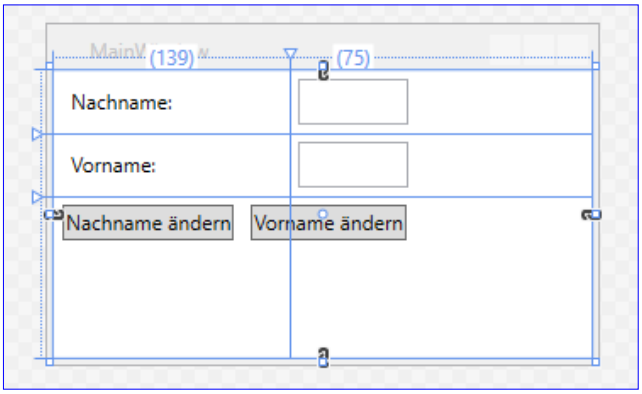


Bild 1: Beispielfenster

Dem Grid des Fensters **MainWindow.xaml** fügen wir in zwei Spalten und drei Zeilen einige Elemente hinzu, die wie folgt definiert werden und die im Entwurf wie in Bild 1 aussehen:

```
<Grid>
    --- Grid-Definition ...
    <Label Content="Vorname:" Grid.Row="1" Margin="5"></Label>
    <TextBox Text="{Binding kunde.Vorname}" Margin="5" Grid.Row="1" Grid.Column="1"></TextBox>
    <Label Content="Nachname:" Grid.Row="0" Margin="5"></Label>
    <TextBox Text="{Binding kunde.Nachname}"  Grid.Row="0" Grid.Column="1" Margin="5"></TextBox>
    <StackPanel Orientation="Horizontal" Grid.Row="2" Grid.ColumnSpan="2">
        <Button x:Name="btnNachnameAendern" Margin="5" Content="Nachname ändern" Click="btnNachnameAendern_Click"></Button>
        <Button x:Name="btnVornameAendern" Margin="5" Content="Vorname ändern" Click="btnVornameAendern_Click"></Button>
    </StackPanel>
</Grid>
```

Die beiden **TextBox**-Elemente sind jeweils über das Attribut **Text** an die Eigenschaften **kunde.Vorname** und **kunde.Nachname** gebunden. Das Objekt **kunde** des Typs **Kunde** enthält, wie aus der Klassendefinition oben ersichtlich, die beiden Eigenschaften

Vorname und **Nachname** und wird beim Erstellen des Fensters **MainWindows** erzeugt. Dafür sorgt die Konstruktor-Methode **MainWindow()**, die beim Erstellen des Objekts ausgelöst wird. Die Klasse enthält außerdem noch eine öffentliche Variable des Typs **Kunde** namens **kunde**, die in der Konstruktor-Methode gefüllt wird:

```
//Code der Klasse MainWindow.xaml.cs
public partial class MainWindow : Window {
    public Kunde kunde { get; set; }          //öffentliche Variable für das anzuzeigende Kunde-Objekt

    public MainWindow() {                     //Konstruktor-Methode
        InitializeComponent();                //Initialisiert das Fenster auf Basis des .xaml-Codes
        kunde = new Kunde();                  //Erstellt ein neues Kunde-Objekt
        kunde.Vorname = "André";              //Füllt die Eigenschaft Vorname
        kunde.Nachname = "Minhorst";          //Füllt die Eigenschaft Nachname
        DataContext = this;                   //Weist die Klasse als Datenquelle für das Fenster zu
    }
...
}
```

Dadurch, dass **DataContext** auf **this** eingestellt ist, können wir im XAML-Code auf alle öffentlichen Elemente dieser Klasse zugreifen. Nach dem Starten sieht die Anwendung nun wie in Bild 2 aus.

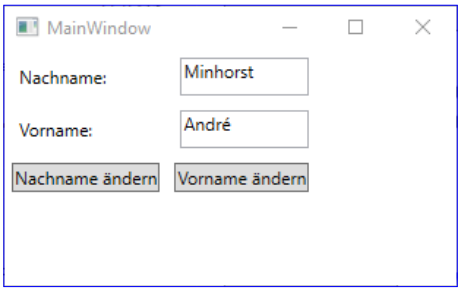


Bild 2: Fenster mit gefüllten Steuerelementen

Eigenschaft aktualisieren

Wozu brauchen wir nun den **PropertyChanged**-Mechanismus? Wir füllen nun die Ereignismethode für die Schaltfläche **btnNachnameAendern** mit folgendem Code:

```
private void btnNachnameAendern_Click(object sender, RoutedEventArgs e) {
    kunde.Nachname = "Neuer Nachname";
    MessageBox.Show("Vorname: '" + kunde.Vorname + "', Nachname: '" + kunde.Nachname + "'", "Neuer Nachname:");
}
```

Dies ändert den Wert der Eigenschaft **Nachname** des angezeigten Objekts auf einen neuen Wert. Außerdem gibt ein Meldungsfenster den neuen Inhalt der beiden Eigenschaften **Vorname** und **Nachname** aus. Für die zweite Schaltfläche hinterlegen wir analog eine Ereignismethode, welche den Wert von **kunde.Vorname** auf einen neuen Wert einstellt und ebenfalls die beiden aktuellen Werte der beiden Eigenschaften ausgibt. Im aktuellen Zustand werden die **MessageBox**-Objekte die neuen Werte anzeigen, aber diese werden nicht in die an die Eigenschaften gebundenen Textfelder übernommen (siehe Bild 3):

```
private void btnVornameAendern_Click(object sender, RoutedEventArgs e) {
    kunde.Vorname = "Neuer Vorname";
    MessageBox.Show("Vorname: '" + kunde.Vorname + "', Nachname: '" + kunde.Nachname + "'", "Neuer Vorname:");
}
```

Events in der Praxis

Im Artikel »Von VBA zu C#: Objekt-Ereignisse« haben wir bereits die Grundlagen zur Programmierung und Implementierung benutzerdefinierter Ereignisse gelegt. Dies wollen wir nun ausbauen, indem wir uns zwei praktische Beispiele ansehen. Dabei wollen wir von einem Hauptfenster aus verschiedene Ansichten in einem Frame anzeigen, darunter eine Kundenübersicht und eine Kundendetailansicht. Beim Anzeigen sollen verschiedene Dinge geschehen, die wir über die Implementierung von Ereignissen lösen wollen – und zwar über eingebaute sowie über benutzerdefinierte Ereignisse.

Beispielprojekt

Im Beispiel geht es um ein Hauptfenster, das über zwei Ribbon-Buttons verschiedene **Page**-Elemente in einem Frame anzeigen soll. Die Schaltfläche **Kundenübersicht** zeigt das **Page**-Objekt **Kundenuebersicht** an, die Schaltlfäche **Neuer Kunde** das **Page**-Objekt **Kundendetails**. Diese Seite wird ebenfalls aufgerufen, wenn der Benutzer doppelt auf einen der Einträge der Kundenübersicht klickt (siehe Bild 1).

Im ersten Beispiel wollen wir dafür sorgen, dass eine dritte Ribbon-Schaltfläche zum Löschen von Elementen der Kundenübersicht nur aktiviert wird, wenn die Kundenübersicht angezeigt wird. Das heißt, dass diese Schaltfläche deaktiviert werden soll, wenn eine andere Ansicht als die Kundenübersicht erscheint. Dazu wollen wir ein Ereignis implementieren, das beim Seitenwechseln im Frame-Element ausgelöst wird.

Im zweiten Beispiel wollen wir zwei eigene Events programmieren und implementieren. Die Kundenübersicht soll nicht jedes Mal, wenn der Benutzer von einer anderen Seite zurück auf diese Seite wechselt, neu erstellt werden beziehungsweise ihre Daten neu aus der Datenbank einlesen. Dies soll nur geschehen, wenn entweder einer der angezeigten Einträge auf der Seite **Kundendetails** geändert oder ein neuer Eintrag hinzugefügt wurde. Dazu wollen wir ein Ereignis in der Code behind-Klasse der Seite **Kundendetails** programmieren, die wir dann in der Code behind-Klasse des **MainWindow**-Objekts implementieren.

So können wir dann direkt im **MainWindow** den relevanten Code ausführen und müssen nicht von der untergeordneten Seite mit den Kundendetails auf das übergeordnete Fenster zugreifen – den Hintergrund erläutern wir weiter unten.

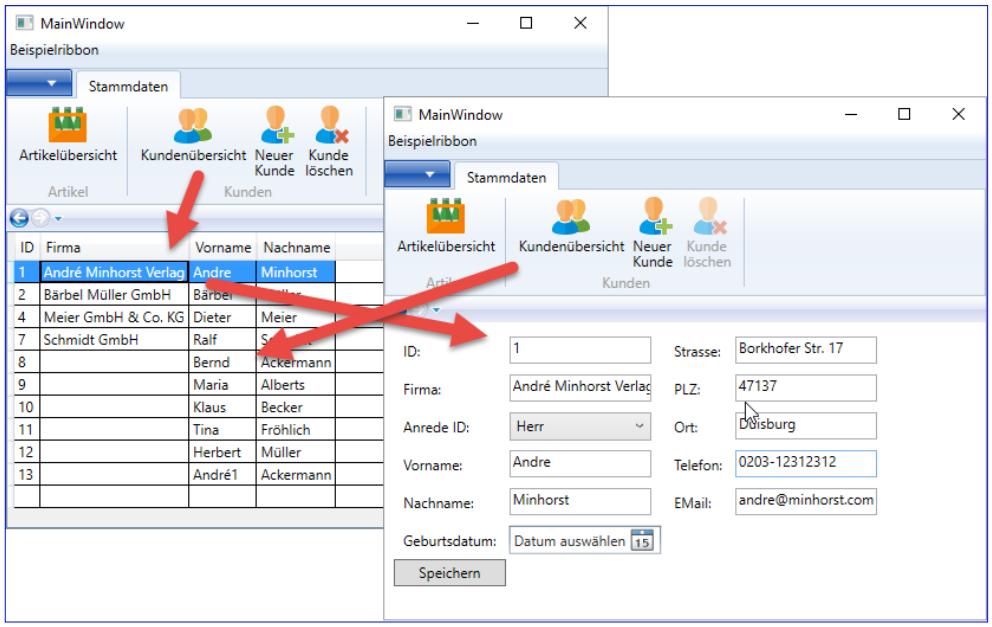


Bild 1: Interaktion zwischen den einzelnen Seiten

Eingebaute Events nutzen

Im ersten Beispiel wollen wir also ein **Ribbon**-Steuerelement im **MainWindow** aktivieren oder deaktivieren, je nachdem welches **Page**-Element das **Frame**-Element anzeigt. Wenn es die Seite **Kundenuebersicht** anzeigt, soll die Schaltfläche **btnKundeLoeschen** aktiviert werden, wenn es eine andere Seite anzeigt, soll der Benutzer diese Schaltfläche nicht anklicken können.

Dazu legen wir für das Frame-Element im XAML-Code, also in der Klasse **MainWindow.xaml**, das Attribut **Navigated** an und füllen es mit dem Wert **WorkZone_Navigated**:

```
<Window x:Class="Bestellverwaltung.MainWindow" ...
        Title="MainWindow" Height="450" Width="525">
    <Grid>
        ...
        <Frame x:Name="WorkZone" Grid.Row="1"
            Navigated="WorkZone_Navigated"></Frame>
    </Grid>
</Window>
```

Für dieses Ereignis hinterlegen wir die folgende Methode in der Code behind-Datei von **MainWindow**:

```
private void WorkZone_Navigated(object sender,
        System.Windows.Navigation.NavigationEventArgs e) {
    switch (WorkZone.Content.ToString()) {
        case "Bestellverwaltung.Kundenuebersicht":
            btnKundeLoeschen.IsEnabled = true;
            break;
        default:
            btnKundeLoeschen.IsEnabled = false;
            break;
    }
}
```

Diese Methode wird nun beim Wechseln des im Frame-Objekts angezeigten Objekts ausgelöst. Mit einer **switch**-Bedingung prüfen wir den Namen des angezeigten Elements, hier mit der Eigenschaft **Content** ermittelt. Hat diese den Wert **Bestellverwaltung.Kundenuebersicht**, stellt die Methode den Wert der Eigenschaft **IsEnabled** des Elements **btnKundeLoeschen** auf **true** ein. Für jeden anderen Inhalt erhält **IsEnabled** den Wert **false**.

Was ist nun der Unterschied zu den Ereignissen etwa einer Schaltfläche, die wir nun schon in vielen Beispielen implementiert haben? In den bisherigen Beispielen haben wir meist selbst die Ereignisse ausgelöst, zum Beispiel durch einen Mausklick auf eine Schaltfläche. Diesmal wurde das Ereignis indirekt durch unsere Aktion ausgelöst, nämlich durch einen Klick auf eine Ribbon-Schaltfläche, die wiederum den Inhalt des **Frame**-Objekts geändert hat. Und das hat dann unser hier implementiertes Ereignis hervorgerufen.

Benutzerdefinierte Events

Wie oben gezeigt, gibt es eine große Anzahl eingebauter Ereignisse, die Sie für Ihre Zwecke nutzen können. Nicht immer jedoch werden Ereignisse zum gewünschten Zeitpunkt ausgelöst. Ein Beispiel ist das Speichern eines Kunden, der neu angelegt oder geändert wurde. Unter Access, wo alles auf die Anzeige und Bearbeitung von Daten ausgelegt ist, finden Sie entsprechende Ereignisse zuhauf – zum Beispiel das Ereignis **Nach Aktualisierung**. Unter WPF/C# sorgen wir selbst dafür, dass die Änderungen an den angezeigten Daten auch in die Datenbank geschrieben werden. Dementsprechend müssen wir auch selbst Ereignisse definieren, die zu einem solchen Zeitpunkt ausgelöst werden.

In unserem Fall wollen wir ein solches Ereignis nutzen, um festzulegen, ob eine Änderung oder eine Neuanlage eines Kunden erfolgt ist. Abhängig davon soll die Kundenübersicht, wenn sie das nächste Mal eingeblendet wird, entweder neu erzeugt werden (falls Änderungen vorliegen) oder mit den alten Daten eingeblendet werden.

Dazu soll das Hauptfenster eine Variable namens **KundenChanged** vorhalten, die lesend und schreibend zugreibar ist und wie folgt in der Klasse **MainWindow** deklariert wird:

```
public bool KundeChanged { get; set; }
```

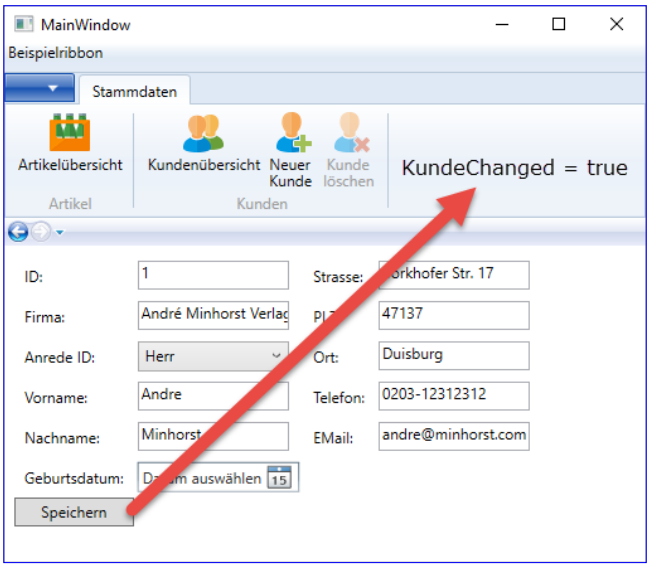


Bild 2: Ein Klick auf **Speichern** stellt die Eigenschaft **KundeChanged** im MainWindow auf **true** ein.

Die einfache Variante

Nun könnten wir im **Page**-Objekt **Kundendetails** die Schaltfläche **Speichern** mit einer Anweisung ausstatten, welche die Eigenschaft **KundeChanged** einfach auf den Wert **true** einstellt, wenn der Benutzer die Schaltfläche betätigt hat (siehe Bild 2).

In diesem einfachen Fall sieht die Methode, die beim Anklicken der **Speichern**-Schaltfläche ausgelöst wird, wie in Listing 1 aus. Die erste Anweisung prüft, ob es sich um einen neuen Kunden handelt (in diesem Fall ist die **ID** noch **0**, da noch nicht vom Datenbanksystem vergeben). Ist der Kunde neu, wird er im ersten Teil der **If**-Bedingung hinzugefügt und in der Datenbank gespeichert. Außerdem referenzieren wir das **MainWindow**-Objekt über **Window.GetWindow(this)** und stellen anschließend seine Eigenschaft **KundeChanged**, die wir weiter oben als öffentliche Variable des Fensters **MainWindow** deklariert haben, auf den Wert **True** ein. Handelt es sich um einen vorhandenen Kunden, der geändert wurde, speichern wir ebenfalls die Änderungen in der Datenbank und stellen dann wie oben den Wert der Eigenschaft **KundeChanged** auf **True** ein.

Neu erstellen oder nicht?

Wenn der Benutzer nun im **MainWindow** auf die Ribbon-Schaltfläche **btnKundenuebersicht** klickt, löst er die Ereignismethode **btnKundenuebersicht_Click** aus Listing 2 aus. Diese prüft im ersten Teil der **if**-Bedingung vor dem Oder-Zeichen (**|**), ob **kundenuebersicht** überhaupt schon existiert, also nicht den Wert **null** enthält – in diesem Fall wird dieses **Page**-Objekt auf jeden Fall neu erstellt. Der Teil hinter dem Oder-Zeichen prüft, ob die Boolean-Variable **KundeChanged** den Wert **true** enthält.

```
private void btnSpeichern_Click(object sender, RoutedEventArgs e) {
    bool kundeCreated = kundeTemp.ID == 0;
    if (kundeCreated) {
        dbContext.Kunden.Add(kundeTemp);
        dbContext.SaveChanges();
        MainWindow wnd =
            (MainWindow)Window.GetWindow(this);
        wnd.KundeChanged = true;
    }
    else {
        dbContext.SaveChanges();
        MainWindow wnd =
            (MainWindow)Window.GetWindow(this);
        wnd.KundeChanged = true;
    }
}
```

Listing 1: Beispiel für das Aktualisieren einer Variablen durch direkten Zugriff

```
private void btnKundenuebersicht_Click(object sender, RoutedEventArgs e) {
    if (kundenuebersicht == null | KundeChanged == true) {
        kundenuebersicht = new Kundenuebersicht();
        KundeChanged = false;
    }
    WorkZone.Content = kundenuebersicht;
}
```

Listing 2: Prüfung, ob Seite neu geladen werden muss

Auch dies kann ein Indikator sein, dass die Seite **kundenuebersicht** neu geladen werden muss, da in diesem Fall ein neuer Datensatz angelegt oder ein bestehender Datensatz geändert wurde. Wird die Seite nun neu erstellt und später über die **Content**-Eigenschaft in das **Frame**-Objekt **WorkZone** geladen, stellt die Methode auch gleichzeitig die Variable **KundeChanged** wieder auf den Wert **false** ein, damit nun nicht bei jedem Aufruf ein Neuerstellen der Seite erfolgt. Perfekt – warum lassen wir es nicht einfach so?

Weil wir so zu viele Abhängigkeiten zwischen den einzelnen Elementen der Benutzeroberfläche schaffen. Wenn die Seite zum Anzeigen der Kundendetails wiederverwendbar sein soll, also wenn wir die Seite auch in anderen Fenstern oder Zusammenhängen anzeigen wollen, dann ist es wichtig, dass die Seite nicht von bestimmten Eigenschaften der übergeordneten Instanz – hier des Fensters mit dem Frame-Element – abhängt. Und genau das geschieht, wenn wir im MainWindow eine öffentliche Eigenschaft anbieten, die von einem untergeordneten Element wie der Seite bei bestimmten Ereignissen geändert oder auch abgefragt werden soll.

Welche Alternativen haben wir also? Wir wollen dafür sorgen, dass das MainWindow davon erfährt, wenn in dem im Frame angezeigten **Page**-Element etwas geschieht, ohne vom untergeordneten Element direkt auf die Eigenschaften des übergeordneten Elements zuzugreifen. Die Lösung ist, in dem untergeordneten Element ein Ereignis zu definieren (oder auch ein vorhandenes Ereignis zu nutzen),

Für Quereinsteiger: Wie es unter Access/VBA läuft

Die Syntax zum Deklarieren öffentlicher Ereignisse sah für diejenigen, die bisher mit Access/VBA gearbeitet haben, recht einfach aus – zum Beispiel so:

```
Public Event Ereignisname(Parameter1 As <Datentyp>, Parameter2 As <Datentyp>, ...)
```

Um das Ereignis auszulösen, war schlicht der Aufruf dieses Ereignisses erforderlich – zum Beispiel wie folgt innerhalb einer Prozedur:

```
Private Sub cmdOK_Click
    Call Ereignisname("Wert1", "Wert2", ...)
End Sub
```

Die Klasse hat man dann in einer anderen Klasse mit dem Schlüsselwort **WithEvents** deklariert:

```
Dim WithEvents objAusloeser As clsAusloeser
```

Danach braucht man nur noch den Steuerelementnamen und den Ereignisnamen aus den beiden Kombinationsfeldern im VBA-Editor auszuwählen, um die Implementierung des Ereignisses wie folgt zu erzeugen:

```
Private Sub objAusloeser_Ereignisname(Parameter 1 As _
        <Datentyp>, Parameter2 As <Datentyp>, ...)
    ...
End Sub
```

Unter C# sieht das Ganze wesentlich komplizierter aus, eröffnet dadurch aber natürlich auch mehr Flexibilität.

Der Ablauf

Wenn wir zwei benutzerdefinierte Ereignisse definieren, die durch das Anlegen eines neuen Kunden beziehungsweise das Ändern eines vorhandenen Kunden ausgelöst werden, geschieht erst einmal nichts bei diesen beiden Aktionen. Die Ereignisse werden ausgelöst, aber wenn diese nicht implementiert werden, passiert auch nichts. Erst wenn wir wie in diesem Beispiel in **MainWindow.xaml.cs** Ereignismethoden definieren, welche die Ereignisse **ItemChanged** oder **ItemAdded** implementieren, können wir die Ereignisse nutzen. Den allgemeinen Ablauf zeigt Bild 3.

Die Ereignisse werden dann in **Kundendetails.xaml.cs** ausgelöst, was die Ereignismethoden in **MainWindow.xaml.cs** auf den Plan ruft. Deren Implementierung stellt schließlich die Eigenschaft **KundeChanged** auf **true** ein, sodass die Kundenübersicht beim nächsten Anzeigen neu erstellt wird. Wir schauen uns nun zwei verschiedene Varianten zum Definieren und Nutzen von benutzerdefinierten Events an.

EDM: Ausnahmen beim Speichern behandeln

Die Validierung bei der Eingabe von Daten ist eines der wichtigsten Themen bei der Erstellung benutzerfreundlicher Anwendungen. In diesem ersten Artikel zu diesem Thema wollen wir uns darauf beschränken, solche Eingabefehler abzufangen, welche durch die Restriktionen im Datenmodell und entsprechende Fehleingaben entstehen. Das sind beispielsweise Fehler, die auftreten, weil der Benutzer keinen Wert in ein Feld eingibt, das nicht leer sein darf, oder der Datentyp des Feldes nicht mit dem eingegebenen Wert korrespondiert.

Beispielanwendung

Wie bei den meisten Artikeln dieser Ausgabe wollen wir unser Know-how an der Beispielanwendung **Bestellverwaltung** ausprobieren. Diese hat in einigen Tabellen Restriktionen, gegen die wir über die Eingabe in den beiden vorgestellten Seiten **Kundenuebersicht** oder **Artikeluebersicht** verstoßen können. Das wären beispielsweise Folgende:

- Das Feld **Bezeichnung** in der Tabelle **Artikel** darf nicht **Null** sein, muss also einen Wert enthalten.

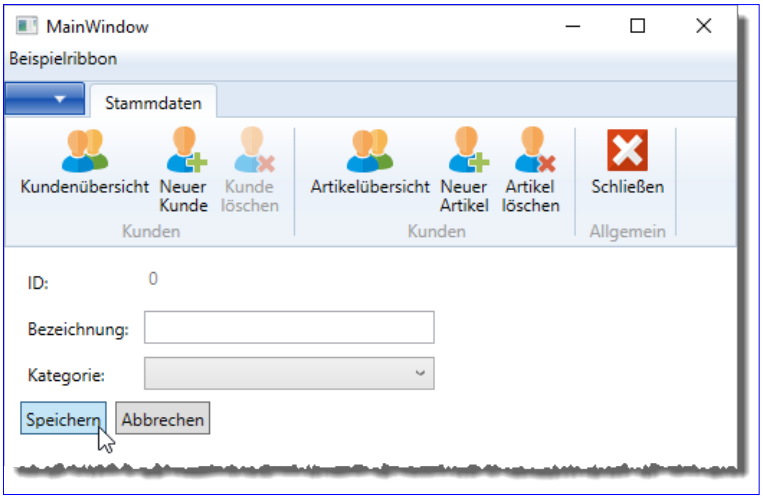


Bild 1: Speichern eines Datensatzes ohne Angabe der Bezeichnung

- Für die Fremdschlüsselfelder **AnredeID** der Tabelle **Kunden** oder **KategorieID** der Tabelle **Artikel** muss ein Wert angegeben werden.

- Andere Felder sind grundsätzlich aufgrund ihres Datentyps auf die Eingabe bestimmter Werte beschränkt – zum Beispiel können Sie keinen Text in ein Zahlen- oder Datumsfeld eingeben.

Ablauf ohne Validierung

Schauen wir uns an, was geschieht, wenn wir keine Vorsichtsmaßnahmen treffen und davon ausgehen, dass der Benutzer die korrekten Werte in die Felder eingibt und auch alle Pflichtfelder ausfüllt. Nehmen wir also an, dass der Benutzer einen neuen Artikel anlegt und vor dem Klick auf die Schaltfläche **Speichern** keine Bezeichnung eingibt und auch keine Kategorie auswählt (siehe Bild 1).

Dann liefert Visual Studio die Ausnahme aus Bild 2. Die hier angegebene Information, es sei eine Ausnahme des Typs **System.Data.Entity.Validation.DbEntityValidationException** liefert noch keine echte Hilfe beim Beheben des Problems.

Also klicken Sie auf den Link **Details anzeigen...** und erhalten eine weitere Meldung, die diesmal den Hinweis **Fehler bei der Überprüfung einer oder mehrerer Entitäten** liefert. Wir erfahren noch, dass die Methode **SaveChanges** den Fehler ausgelöst hat, aber das war es dann auch.

Wir können an dieser Stelle nur die Meldung ausblenden und die Ausführung beenden. Selbst wenn Sie den Link **Ausnahmedetail in die Zwischenablage kopieren** anklicken und den Inhalt der Zwischenablage anschließend in einen Editor Ihrer Wahl einfügen, erhalten Sie hier keine

nennenswerten weiteren Informationen.

Einzelnen Fehler provozieren

Schauen wir uns nun an, was geschieht, wenn wir nur die **Bezeichnung** auslassen. In diesem Fall erhalten wir den gleichen Fehler, als wenn wir die Bezeichnung und die Kategorie weglassen. Unter Messsage erhalten wir dann, wenn wir in die Tiefe gehen, die folgende Meldung:

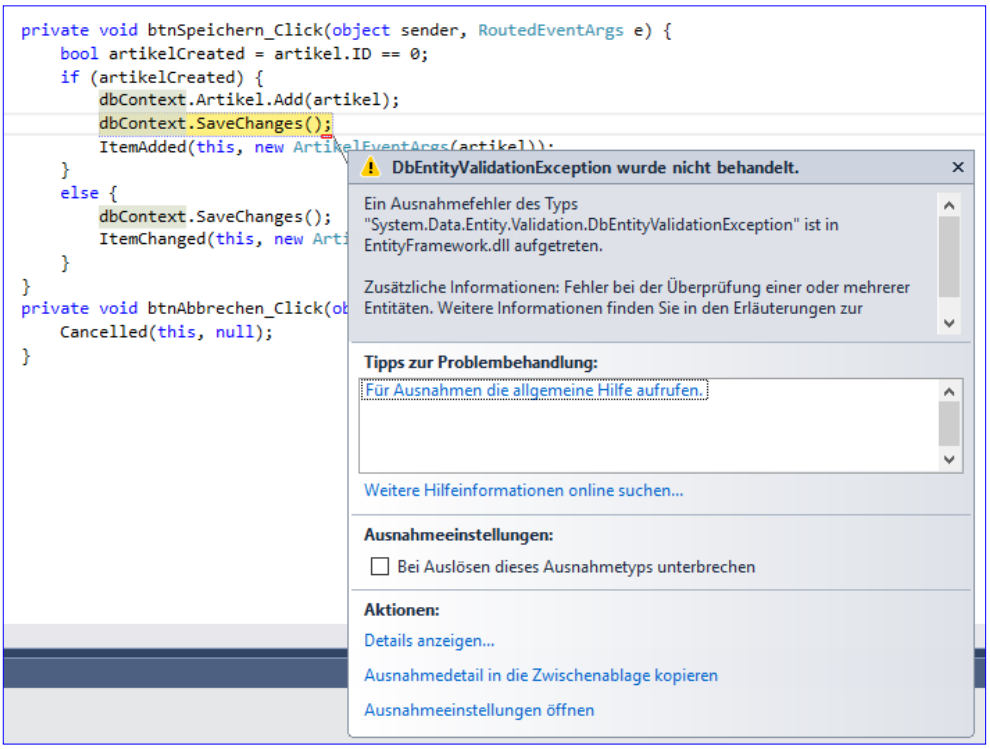


Bild 2: Ausnahme beim Versuch, Daten ohne Füllen eines Feldes zu speichern, das keine Nullwerte akzeptiert

```
Fehler bei der Überprüfung einer oder mehrerer Entitäten. Weitere Informationen finden Sie in den Erläuterungen zur EntityValidationErrors-Eigenschaft.
```

Um den Fehler weiter zu untersuchen, fassen wir die **SaveChanges**-Methode in einen **Try...Catch**-Block ein, der wie folgt aussieht:

```
try {
    dbContext.SaveChanges();
}
catch (Exception ex) {
    message = string.Format("Fehler: {0}", ex.Message);
    Debug.WriteLine(message);
    message = string.Format("Typ: {0}", ex.GetType());
    Debug.WriteLine(message);
}
```

Wir fangen also eine einfache Exception ab und geben im Ausgabefenster den Inhalt der Eigenschaften **Message** und **GetType** des **Exception**-Objekts aus. Das Ergebnis sieht wie folgt aus:

```
Fehler: Fehler bei der Überprüfung einer oder mehrerer Entitäten. Weitere Informationen finden Sie in den Erläuterungen zur EntityValidationErrors-Eigenschaft.
Typ: System.Data.Entity.Validation.DbEntityValidationException
```

Da wir nun wissen, dass es sich um eine Exception des Typs **DbEntityValidationException** handelt, können wir nun gezielt diese Exception abfangen.

DbEntityValidationException abfangen

Die Abfrage der Exception **DbEntityValidationException** bauen wir wie in Listing 1 in die Methode **btnSpeichern_Click** ein. Den bestehenden **Catch**-Zweig mit der Standardexception behalten wir bei, damit auch andere Fehler behandelt werden. Der neue **Catch**-Zweig gibt zunächst eine Meldung aus, dass ein Fehler aufgetreten ist. Dann durchläuft er in einer **foreach**-Schleife alle Elemente des Typs **DbEntityValidationResult** der Auflistung **EntityValidationErrors** und speichert diese in der Variablen **item**.

```
try {
    dbContext.SaveChanges();
}
catch (DbEntityValidationException ex) {
    Debug.WriteLine("Fehler beim Speichern:");
    Debug.WriteLine("=====================================");
    foreach (DbEntityValidationResult item in ex.EntityValidationErrors) {
        DbEntityEntry entry = item.Entry;
        string entityTypeName = entry.Entity.GetType().Name;
        foreach (DbValidationError subItem in item.ValidationErrors) {
            string message = string.Format("Fehlerbeschreibung: '{0}'", subItem.ErrorMessage);
            Debug.WriteLine(message);
            message = string.Format("Tabelle/Entität: {0}", entityTypeName);
            Debug.WriteLine(message);
            message = string.Format("Feld/Eigenschaft: {0}", subItem.PropertyName);
            Debug.WriteLine(message);
        }
    }
    Debug.WriteLine("=====================================");
}
```

Listing 1: Analysieren eines Fehlers

Die Variable **entry** mit dem Typ **DbEntityEntry** erhält den Inhalt der **Entry**-Eigenschaft des in **item** gespeicherten Elements. Der Zugriff auf die Eigenschaft **Name** der Funktion **GetType()** des **Entity**-Objekts aus dem **DBEntityEntry**-Element liefert den Namen der Entität, hier **Artikel**. Da für das Speichern eines Objekts dieser Entität in der Datenbank mehrere Fehler gleichzeitig aufgetreten sein können, durchlaufen wir die Auflistung **ValidationErrors** des **DBEntityEntry**-Objekts aus **item** – und dies wiederum in einer **foreach**-Schleife. Diese liefert mit den beiden Eigenschaften **ErrorMessage** und **PropertyName** zwei wichtige Informationen, nämlich die Fehlermeldung und die Eigenschaft, deren Speichern den Fehler ausgelöst hat. Das Ausgabe-Fenster zeigt die gewünschten Informationen nun wie in Bild 3 an. Wir wissen nun also, dass die Eingabe eines Wertes für das Feld **Bezeichnung** erforderlich ist und könnten mit den Informationen, die hier geliefert wurden, auch eine entsprechende Validierungsmeldung ausgeben.

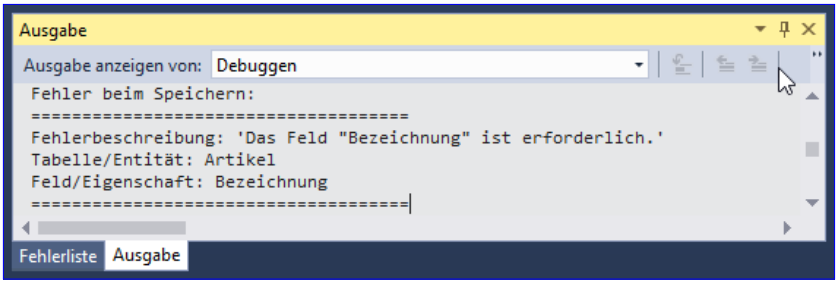


Bild 3: Ausgabe der Informationen bei der **DBEntityValidationException**

Fehlender Fremdschlüsselwert

Schauen wir uns nun an, ob wir auch das Problem mit dem Fremdschlüsselfeld auf diese Weise in den Griff bekommen. Wenn Sie also erneut einen neuen Artikel anlegen wollen und diesmal zwar die Bezeichnung eingeben, aber keinen Wert für das Fremdschlüsselfeld **KategorieID** auswählen, greift wieder die allgemeine Exception im letzten **Case**-Zweig. Dies gibt die folgende Meldung aus, die ohne Exception-Behandlung wie in Bild 4 aussehen würde:

```
Fehler: Fehler beim Aktualisieren der
Einträge. Weitere Informationen finden
Sie in der internen Ausnahme.
Typ: System.Data.Entity.Infrastructure.
DbUpdateException
```

EDM: Validieren von Entitäten mit IDataErrorInfo

Die Validierung bei der Eingabe von Daten ist eines der wichtigsten Themen bei der Erstellung benutzerfreundlicher Anwendungen. Nachdem der Artikel »EDM: Ausnahmen beim Speichern behandeln« gezeigt hat, wozu Sie im Rahmen der Validierung die durch Restriktionen im Datenmodell auftretenden Exceptions nutzen können, schauen wir uns nun einen einfachen Weg an, um Validierungsregeln in Entitätsklassen zu definieren und beim Fehlschlagen der Validierung entsprechende Meldungen in der Benutzeroberfläche auszugeben. Dabei zeigen wir hier den Umgang mit der Schnittstelle »IDataErrorInfo«.

Beispielanwendung

Wie im oben genannten Artikel nutzen wir wieder die Beispielanwendung **Bestellverwaltung**. Bevor die im Datenmodell dieser Anwendung festgelegten Restriktionen greifen und bei Fehleingaben Exceptions auslösen, wollen wir Validierungen programmieren, um Fehleingaben des Benutzers zu vermeiden. In diesem Fall schauen wir uns die Seite **Kundendetails.xaml.cs** an, um die Programmierung der Validierung zu veranschaulichen. Dies soll im Ergebnis wie in Bild 1 aussehen.

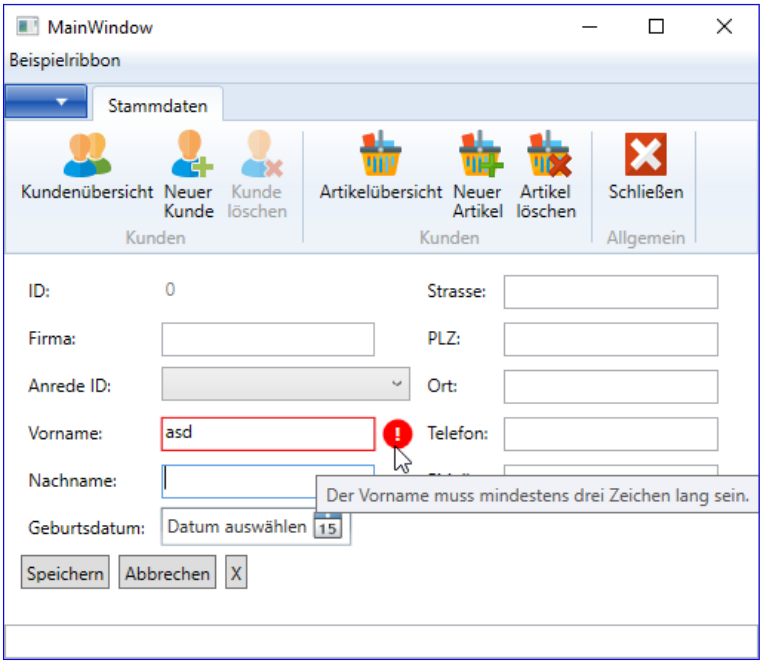


Bild 1: Beispiel für eine einfache Validierungsmeldung

Validierung

Die hier vorgestellten Techniken zur Validierung von Benutzereingaben beziehen sich auf die Untersuchung der jeweiligen Entität – ist ein Wert in einem Feld mit einem eindeutigen Index bereits vorhanden? Darf ein Feld überhaupt leer sein? Hat der Benutzer auch ein gültiges Datum eingegeben? Liegt der Zahlenwert im zulässigen Bereich?

Validierung auf Feldebene

Validierungen können Sie auf mehrere Arten durchführen. Der Artikel **EDM: Ausnahmen bei Speichern behandeln** zeigt, wie Sie erst beim Auftreten von Fehlern bei Speichern auf fehlerhafte Eingaben des Benutzers reagieren können. Davor gibt es noch mindestens zwei weitere Methoden:

- Validieren direkt nach der Eingabe eines Wertes in ein Steuerelement und Hinweis auf die fehlerhafte Eingabe
- Validieren, nachdem der Benutzer den Speichervorgang initiiert hat, aber bevor das Speichern selbst stattfindet. Hier können dann nicht nur einzelne Felder isoliert, sondern auch mehrere Felder im Zusammenhang untersucht werden. Wie dies gelingt, zeigt zum Beispiel der Artikel **EDM: Validieren von Entiten mit IValidatableObject**.

In diesem Artikel schauen wir uns die einfachere Validierung direkt nach der Eingabe eines Wertes an.

Für den Benutzer hat dies beispielsweise den Vorteil, dass er Fehleingaben direkt erkennt und diese direkt erkennen kann.

Einfache Validierung

In vielen Fällen reicht eine einfache Validierung der Eingaben für Steuerelemente aus. Damit meinen wir, dass nur die Werte einzelner Steuerelemente geprüft werden sollen, und zwar direkt nach der Eingabe. So können Sie beispielsweise abfragen, ob ein Wert eine bestimmte Mindestlänge erreicht und bei Nichterfüllung dieser Vorgabe einen optischen Hinweis einblenden. Dieser Artikel zeigt, wie Sie eine solche Validierung zu einer Anwendung hinzufügen können.

Beispielanwendung

Die Beispielanwendung heißt **Bestellverwaltung**. Die im Fenster **MainWindow** eingeblendete Seite **Kundendetails.xaml** soll zeigen, wie Sie die hier vorgestellte Art der Validierung implementieren.

Ablauf der Validierung

Die Validierung mit der hier vorgestellten Methode geschieht immer gleich nach dem Abschluss der Eingabe in ein Feld. Sie löst dann die Anzeige des Steuerelements mit der fehlerhaften Eingabe mit rotem Rahmen und mit einem Warnsymbol aus. Das Überfahren des Warnsymbols mit der Maus führt zur Anzeige eines ToolTip-Textes, der weitere Informationen zum vorliegenden Fehler liefert.

Aufbau der Validierung

Die Validierung basiert auf zwei verschiedenen Elementen. Das erste Element ist die Klasse, welche die Entität der zu validierenden Objekte beschreibt, in unserem Fall also etwa **Kunde.cs**. Hier legen wir in Form einer Methode die Regeln fest, nach der das soeben geänderte Steuerelement geprüft werden soll. Damit erhalten wir eine Basis, die wir für alle Zugriffe auf Objekte dieser Klasse nutzen können, was die Wiederverwendbarkeit dieser Klasse erhöht. Der zweite Teil der Validierung ist das Hinzufügen einer bestimmten Eigenschaft zum Binding des Steuerelements, im Falle eines Textfeldes also beispielsweise zum Attribut **Text**. Außerdem müssen wir noch definieren, wie sich das Aussehen des Textfeldes ändern soll, wenn die Validierung fehlgeschlagen ist. Immerhin wollen wir den Benutzer ja auch informieren, wenn dieser eine Fehleingabe tätigt. Dies erledigen wir durch eine entsprechende Vorlage für das **TextBox**-Steuerelement. Schauen wir uns nun die drei Elemente der Validierung an.

Anpassen der Klasse Kunde.cs

Unser Entity Data Model, dass wir auf Basis der SQL Server-Datenbank **Bestellverwaltung** erstellt haben, enthält auch eine Klasse namens **Kunde.cs**, welche die Eigenschaften eines Kunden abbildet. Diese erweitern wir nun um die Schnittstelle **IDataErrorInfo**. Da diese in der Bibliothek **System.ComponentModel** beschrieben wird, fügen wir zunächst einen Verweis auf diese Schnittstelle per **using**-Schlüsselwort hinzu:

```
using System.ComponentModel;
```

Danach legen wir in der Kopfzeile der Klasse fest, dass diese die Schnittstelle **IDataErrorInfo** implementieren soll. Die Zeile sieht dann so aus:

```
public partial class Kunde : IDataErrorInfo {
...
```

Um die Member dieser Schnittstelle schnell zu implementieren, wählen Sie den Kontextmenü-Eintrag **Schnellaktionen und Refactorings** von **IDataErrorInfo** aus. Im nun erscheinenden Popup wählen Sie **Schnittstelle implementieren** aus und erstellen so die beiden Methoden dieser Schnittstelle. Diese sehen zunächst wie folgt aus:

```
public string Error {
    get {
        throw new NotImplementedException();
    }
}

public string this[string columnName] {
    get {
        throw new NotImplementedException();
    }
}
```

Hier fügen wir nun für die Methode **this** eine erste Validierung ein. Diese soll sicherstellen, dass die Eingabe in das Feld **Vorname** nicht leer ist.

```
public string this[string columnName] {
    get {
        string errorMessage = "";
        switch (columnName) {
            case "Vorname":
                {
                    if (String.IsNullOrEmpty(Vorname)) {
                        errorMessage = "Bitte geben Sie einen Vornamen ein.";
                    }
                    break;
                }
        }
        return errorMessage;
    }
}
```

Listing 1: Implementierung der Schnittstelle **IDataErrorInfo** in der Klasse **Kunde.cs**

Die Methode **this** nimmt mit dem Parameter **columnName** immer den Namen des Feldes entgegen, dass untersucht werden soll – in diesem Fall **Vorname**. Im **get**-Teil der Methode deklarieren wir die Variable **result** als **string** und prüfen in einer **if**-Bedingung, ob **columnName** den Wert **Vorname** enthält. Falls ja, prüfen wir, ob **Vorname** den Wert **null** enthält oder leer ist (**IsNullOrEmpty**). Ist das der Fall, stellen wir den Rückgabewert mit der Variablen **errorMessage** auf den Text **Bitte geben Sie einen Vornamen ein.** ein (siehe Listing 1).

.xaml-Code erweitern

Wenn Sie das Projekt nun starten, einen neuen Kunden anlegen und einen Vornamen mit maximal drei Zeichen angeben, tut sich jedoch erst einmal nichts. Das ist auch kein Wunder, denn wir haben ja im **.xaml**-Code auch noch nicht angegeben, dass es eine Validierung gibt.

Damit die Methode **this** der Klasse **Kunde.cs** überhaupt aufgerufen wird, müssen wir der Bindung des Textfeldes Vorname das Attribut **ValidatesOnDataErrors** hinzufügen und dafür den Wert **true** festlegen:

```
<TextBox x:Name="txtVorname" Grid.Column="1" Grid.Row="3"
HorizontalAlignment="Stretch"
Text="{Binding kunde.Vorname, Mode=TwoWay,
ValidatesOnDataErrors=True}" />
```

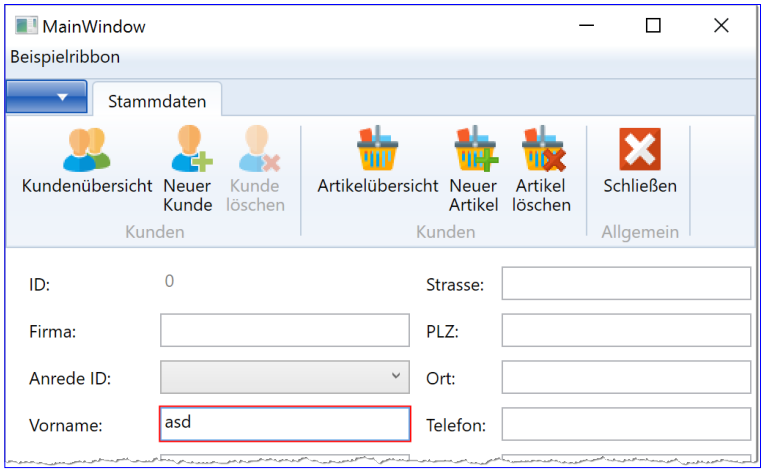


Bild 2: Eine fehlerhafte Eingabe wird durch einen roten Rahmen markiert.

Ein erneuter Start der Anwendung und die Eingabe eines ungültigen Ausdrucks zeigt nun Wirkung: Das fehlerhafte Feld wird mit einem roten Rahmen versehen (siehe Bild 2).

Validierungshinweis anbringen

Nun fehlt noch der Hinweis für den Benutzer, damit er weiß, was er bei der Eingabe falsch gemacht hat. Dazu wollen wir rechts vom Textfeld ein Warnsymbol anzeigen, dass beim Überfahren mit der Maus einen ToolTip-Text einblendet. Dazu ist nicht mehr nötig als die Änderung einiger Eigenschaften des **TextBox**-Elements. Da wir diese nicht nur für eine **TextBox** benötigen, sondern für alle, definieren wir direkt

ein entsprechendes **Style**-Element in den Ressourcen des übergeordneten Elements, hier also für ein **Page**-Element.

Also fügen Sie, soweit noch nicht vorhanden, direkt unterhalb des **Page**-Elements ein **Page.Resources**-Element wie das aus Listing 2 hinzu (wenn Sie ein **Window** zur Anzeige der zu validierenden Daten nutzen, handelt es sich um das Element **Window.Resources**).

Das **Style**-Element wird für jeden Elementtyp einmal angelegt und enthält alle Definitionen, die alle Steuerelemente dieses Typs betreffen. In diesem Fall legen wir diese mit dem Attribut **TargetType** für den Typ **TextBox** fest.
Die ersten drei **Setter**-Elemente definieren die Höhe, den linken, oberen, rechten und unteren Abstand des eigentlichen Elements zu dem dafür reservierten Bereich und die vertikale Ausrichtung.

Das vierte **Setter**-Element liefert den Inhalt für die Eigenschaft **Validation.ErrorTemplate**. Da es sich hier nicht um einen einfachen Attribut-Wert handelt, sondern um eine komplette Struktur, legen wir diese Struktur in Form von Unterelementen unterhalb des Elements **Setter.Value** an. Dass die darunter angegebenen Elemente nur bei einem Validierungsfehler im gebundenen Feld des betroffenen Steuerelements angezeigt werden, liegt am Namen der Property, also **Validation.ErrorTemplate**.

Damit unser Warnsymbol, übrigens ein einfacher roter Kreis mit einem Ausrufezeichen in der Mitte, sichtbar wird, fügen wir zunächst ein **DockPanel**-Element hinzu. Darin bringen wir ein **Border**-Element unter, dass im **DockPanel** rechts verankert wird und die Höhe und Breite 20 hat. Mit dem Wert 10 für das Attribut **CornerRadius** wird das bisherige, 20x20 große Viereck zu einem Kreis. Diesem verleihen wir noch

```
<Page.Resources>
    ...
    <Style TargetType="{x:Type TextBox}">
        <Setter Property="Height" Value="23" />
        <Setter Property="Margin" Value="3,3,27,0" />
        <Setter Property="VerticalAlignment" Value="Center" />
        <Setter Property="Validation.ErrorTemplate">
            <Setter.Value>
                <ControlTemplate>
                    <DockPanel>
                        <Border Background="Red" DockPanel.Dock="right" Margin="5,0,0,0" Width="20" Height="20" CornerRadius="10"
                ToolTip="{Binding ElementName=customAdorner, Path=AdornedElement.(Validation.Errors)[0].ErrorContent}">
                            <TextBlock Text="!" VerticalAlignment="center" HorizontalAlignment="center"
                                FontWeight="Bold" Foreground="white" />
                        </Border>
                        <AdornedElementPlaceholder Name="customAdorner" VerticalAlignment="Center" >
                            <Border BorderBrush="red" BorderThickness="1" />
                        </AdornedElementPlaceholder>
                    </DockPanel>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</Page.Resources>
```

Listing 2: Hinzufügen der Elemente zur Anzeige eines Warnsymbols und eines ToolTip-Textes

EDM: Kunden verwalten mit Ribbon

Bisher haben wir in diesem Magazin nur einzelne Beispiele zur Darstellung von Daten aus Tabellen geliefert. Diesmal wollen wir einen Schritt weiter gehen: Wir erstellen eine WPF-Anwendung, die ein eigenes Ribbon enthält und mit diesem die Steuerung einiger Funktionen zur Auflistung von Kunden sowie zum Bearbeiten, Hinzufügen und Löschen von Kundendatensätzen ermöglicht. Dabei nutzen wir als Container für die angezeigten Seiten mit der Kundenliste und den Kundendetails ein Frame-Objekt. Damit können wir, wenn mehrere Benutzer geöffnet sind, sogar durch die entsprechenden Seiten navigieren.

Dabei wollen wir zunächst eine Kundenübersicht und später eine Seite zum Anlegen eines neuen Kunden über entsprechende Ribbon-Steuerelemente sichtbar machen. Eine weitere Schaltfläche soll den aktuell in der Kundenübersicht markierten Kunden löschen.

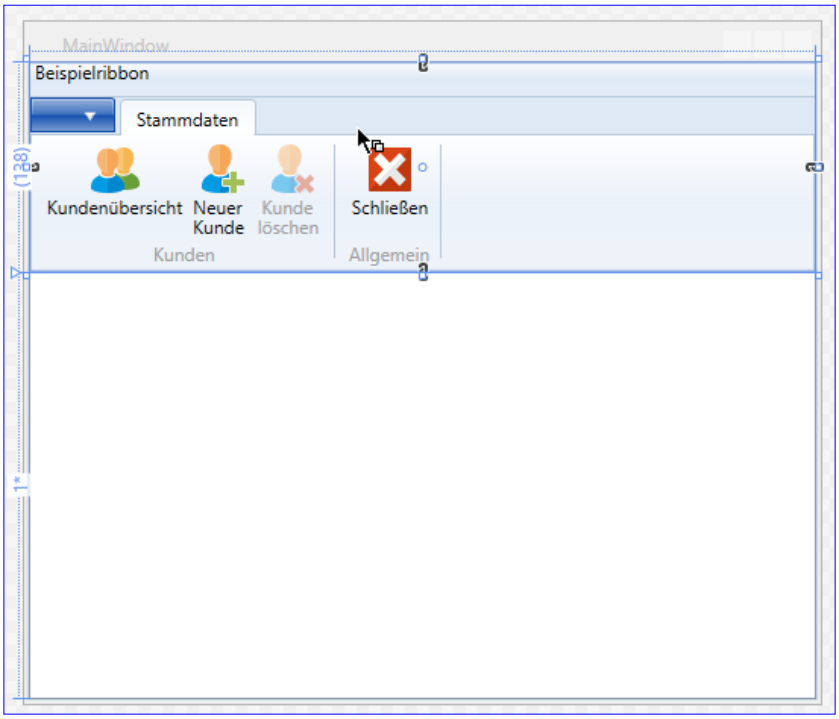


Bild 1: Erster Entwurf des Ribbons

Das heißt, dass wir zunächst ein Ribbon im Fenster **MainWindow** unserer Beispielanwendung benötigen, das drei Schaltflächen für die gewünschte Navigation in den Kundendatensätzen und eine Schaltfläche zum Schließen des Fensters enthält. Dies sollte später so wie in Bild 1 aussehen.

Um dies zu realisieren, legen Sie zunächst ein neues Projekt des Typs **Visual C#\WPF-Anwendung** namens **NavigationMitRibbons** an. Um die Ribbons zu definieren, benötigen Sie einen neuen Verweis, den Sie über den **Verweis-Manager** (Menü-Eintrag **Projekt\Verweise**) hinzufügen. Der Verweis heißt **System.Windows.Controls.Ribbons**.

Da wir dem **Window**-Element im Kopf ein Ribbon und darunter die Steuerelemente zur Anzeige der Artikel- und der Kundenübersicht hinzufügen wollen, legen wir zunächst ein Grid mit zwei Zeilen an. Diese definieren wir wie folgt:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition></RowDefinition>
</Grid.RowDefinitions>
```

Anschließend fügen wir dem Grid ein **Ribbon**-Element mit zwei Schaltflächen in verschiedenen Gruppen hinzu, das wir über das Attribut **Grid.Row="0"** der ersten Ribbonzeile zuweisen (siehe Listing 1). Diesem fügen wir ein **RibbonTab**-Element mit zwei **RibbonGroup**-Elementen hinzu. Die

```
<Ribbon Name="rbnMain" Title="Beispielribbon" Grid.Row="0">
    <RibbonTab Name="Tab1" Header="Stammdaten" KeyTip="A">
        <RibbonGroup Header="Kunden">
            <RibbonButton Name="btnKundenuebersicht" Label="Kundenübersicht" Click="btnKundenuebersicht_Click"
                LargeImageSource="images\users.png"></RibbonButton>
            <RibbonButton Name="btnNeuerKunde" Label="Neuer Kunde" Click="btnNeuerKunde_Click"
                LargeImageSource="images\user_add.png"></RibbonButton>
            <RibbonButton Name="btnKundeLoeschen" Label="Kunde löschen" Click="btnKundeLoeschen_Click"
                LargeImageSource="images\user_delete.png" IsEnabled="false"></RibbonButton>
        </RibbonGroup>
        <RibbonGroup Header="Allgemein">
            <RibbonButton Name="btnSchliessen" Label="Schließen" Click="btnSchliessen_Click"
                LargeImageSource="images\close.png"></RibbonButton>
        </RibbonGroup>
    </RibbonTab>
</Ribbon>
```

Listing 1: Code für ein Ribbon mit Schaltflächen in verschiedenen Gruppen

erste Gruppe statten wir mit drei **RibbonButton**-Elementen aus und die zweite mit einem **RibbonButton**-Element. Die Schaltflächen versehen wir mit Attributen, die auf die jeweiligen Ereignismethoden verweisen. Außerdem legen wir mit der Eigenschaft **LargeImageSource** den Namen jeweils einer Bilddatei fest, die im Ribbon für die jeweilige Schaltfläche angezeigt werden soll. Diese Bilder legen wir in einem Unterordner namens **images** im Projektmappen-Explorer ab.

Seiten erstellen

Nun wollen wir die beiden Seiten erstellen, die durch einen Mausklick auf die beiden Ribbon-Einträge angezeigt werden sollen. Dazu legen Sie zunächst ein erstes neues **Page**-Element an und nennen es **Kundenuebersicht** (**Strg + Umschalt + A**, dann **Seite (WPF)** auswählen und den Namen unten angeben) – siehe Bild 2.

Das Ergebnis ist ein transparentes, rechteckiges Objekt, für das weder Höhe noch Breite mit den üblichen Attributen **Height** oder **Width** festgelegt sind. Stattdessen finden wir die beiden Attribute **d:DesignHeight** und **d:DesignWidth**. Aber was sind das für Eigenschaften? Eigenschaften, die mit **d:** beginnen, sind Eigenschaften für die Design-Ansicht, die nur die Größe des **Page**-Elements für die Entwurfsansicht markieren.

Zur Laufzeit wird das **Page**-Element ja ohnehin in das **Frame**-Element eingebettet und nimmt dessen Größe an. Dementsprechend finden Sie im **Page**-Element die beiden Eigenschaften **d:DesignHeight** und **d:DesignWidth**. Das Attribut

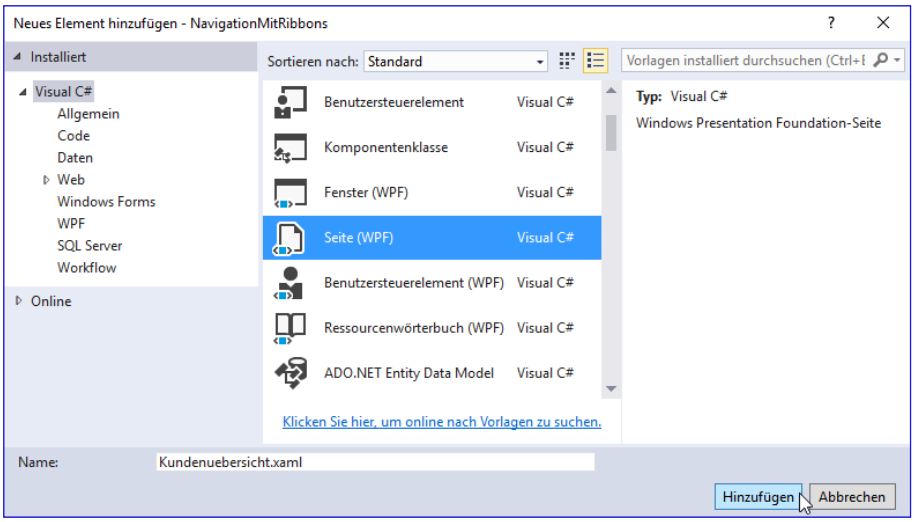


Bild 2: Hinzufügen eines **Page**-Elements

```
public partial class Kundenuebersicht : Page {
    BestellverwaltungEntities dbContext;
    private List<Kunde> kunden;
    public List<Kunde> Kunden {
        get {
            return kunden;
        }
        set {
            kunden = value;
        }
    }
    public Kundenuebersicht(int kundeID = 0) {
        InitializeComponent();
        dbContext = new BestellverwaltungEntities();
        kunden = new List<Kunde>(dbContext.Kunden);
        DataContext = this;
        if (kundeID != 0) {
            Kunde currentKunde = dbContext.Kunden.Find(kundeID);
            dgKunden.SelectedItem = currentKunde;
            dgKunden.ScrollIntoView(currentKunde);
        }
    }
}
```

Listing 2: Code behind-Datei der **Page**-Klasse **Kundenuebersicht.xaml**

mc:Ignorable="d" gibt dem Interpreter zu verstehen, dass Attribute mit führendem **d** nicht interpretiert werden sollen:

```
<Page x:Class="NavigationMitRibbons.Kundenuebersicht"
    ...
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300"
    Title="Kundenuebersicht">
```

Datenquelle

Als Datenquelle für dieses Beispiel verwenden wir wieder die Datenbank **Bestellverwaltung**, die Sie bereits in weiteren Artikeln in diesem Magazin kennen gelernt haben. Dazu haben wir wieder ein Entity Data Model namens **BestellverwaltungEntities** angelegt.

Page mit Kunden füllen

Damit das **Page**-Element die Kunden in einer entsprechenden Liste, hier in Form eines **DataGrid**-Elements, anzeigt, stellen wir zunächst eine entsprechende Datenquelle in Form eines **List**-Objekts in der Code behind-Datei **Kundenuebersicht.xaml.cs** zur Verfügung, die wir innerhalb der Konstruktor-Methode der Klasse füllen (siehe Listing 2).

In der Konstruktor-Methode der Klasse **Kundenuebersicht** wird zunächst die Methode **InitializeComponent** ausgeführt, um das Fenster entsprechend der XAML-Definition aufzubauen. Danach erstellen wir das Entity Data Model-Objekt **dbContext** auf Basis der Klasse **BestellverwaltungEntities**. Die Liste **kunden**, die wir zuvor als privates **List**-Objekt deklariert haben, füllen wir dann mit der Auflistung **Kunden** des **dbContext**-Objekts:

```
kunden = new List<Kunde>(dbContext.Kunden);
```

Danach weisen wir dem **Page**-Objekt mit der **DataContext**-Eigenschaft als Datenherkunft die Code behind-Klasse selbst zu. Damit die Seite nun auf die Kunden-Liste aus der Variablen **kunde** zugreifen kann, stellen wir diese noch mit einer öffentlichen Variablen namens **Kunden** zur Verfügung:

```
public List<Kunde> Kunden {
    ...
}
```

Desweiteren haben Sie sicher bemerkt, dass wir einen optionalen Parameter namens **kundeID** für die Konstruktormethode angelegt haben. Diese wird in der **if**-Bedingung interessant: Ist **kundeID** nämlich nicht **0**, was geschieht, wann immer eine Kunden-ID übermittelt wird, dann liest die Methode den Kunden mit dieser ID in die Variable **currentKunde** ein. Dieser wird dann im DataGrid als aktueller Kunde

```
<Page x:Class="Bestellverwaltung.Kundenuebersicht" ...  xmlns:local="clr-namespace:Bestellverwaltung"
      mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="300" Title="Kundenuebersicht">
    <Grid>
        <DataGrid x:Name="dgKunden" ItemsSource="{Binding Kunden}" AutoGenerateColumns="false">
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding Path=ID}" Header="ID"  />
                <DataGridTextColumn Binding="{Binding Path=Firma}" Header="Firma" />
                <DataGridTextColumn Binding="{Binding Path=Vorname}" Header="Vorname"  />
                <DataGridTextColumn Binding="{Binding Path=Nachname}" Header="Nachname" />
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Page>
```

Listing 3: Code für die Definition des **Page**-Elements zur Anzeige der Kundenübersicht

markiert. Die Methode **ScrollInto-View** sorgt dann noch dafür, dass dieser Datensatz auch noch in den sichtbaren Bereich verschoben wird. Letzteres ist für Access/VBA-Programmierer natürlich ein Traum – dort waren für eine solche Aktion durchaus größere Verrenkungen nötig.

DataGrid-Element in Page anlegen

Schließlich wollen wir ein **DataGrid**-Element im **Page**-Element anlegen, das wie in Listing 3 definiert wird. Die Seite hat ja als Datenherkunft bereits die Code behind-Klasse erhalten. Damit brauchen wir für das **DataGrid**-Element namens **dgKunden** nur noch das Attribut **ItemsSource** auf den Wert **{Binding Kunden}** einzustellen.

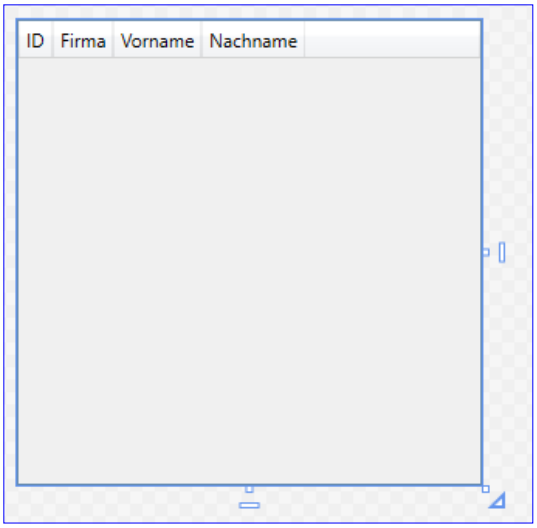


Bild 3: Das **DataGrid** im **Page**-Element in der Entwurfsansicht

Es verwendet dann die Auflistung **Kunden** aus der Klasse **Kunden-uebersicht.xaml.cs** als Datenquelle. Die Spalten sollen nicht automatisch erstellt werden, daher erhält **AutoGenerateColumns** den Wert **false**. Die Spalten legen wir unterhalb des Elements **DataGrid.Columns** an, und zwar als **DataGridTextColumn**-Elemente. Hier erhalten diese per **Binding** eine Verknüpfung zum jeweiligen Feld der Datenherkunft sowie per **Header** eine Spaltenüberschrift. Das Ergebnis soll anschließend wie in Bild 4 aussehen.

```
<Window x:Class="NavigationMitRibbons.MainWindow" ... xmlns:local="clr-namespace:NavigationMitRibbons"
        mc:Ignorable="d" Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>...</Grid.RowDefinitions>
        <Ribbon Name="rbnMain" Title="Beispielribbon" Grid.Row="0">...</Ribbon>
        <Frame x:Name="WorkZone" Grid.Row="1"></Frame>
    </Grid>
</Window>
```

Listing 4: Gekürzte Fassung der Definition von **MainWindow.xaml**, jetzt mit dem **Frame**-Objekt

Wäre das **Page**-Objekt nun ein **Window**-Objekt, könnten wir es einfach öffnen und es würde die enthaltenen Daten anzeigen. Allerdings ist es kein **Window**-Objekt und soll als **Page**-Objekt in einem **Frame**-Element angezeigt werden.

Frame-Element zum Window hinzufügen

Deshalb fügen wir nun zunächst noch ein **Frame**-Element zum Window **MainWindow.xaml** hinzu. **MainWindow.xaml** sieht in gekürzter Form nun wie in Listing 4 aus. Beachten Sie, dass wir das **Frame**-Element mit dem Namen **WorkZone** versehen haben. Dies ist nötig, damit wir später auf dieses Element zugreifen und ihm den anzuzeigenden Inhalt zuweisen können.

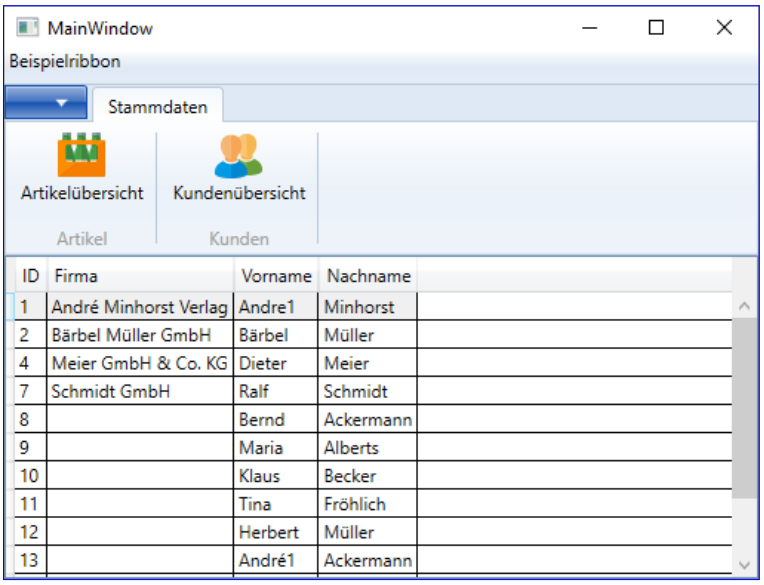


Bild 4: Das **Page**-Element mit den Kundendaten im **Frame**-Element des **Window**-Objekts

Frame-Element mit Page füllen

Fehlt noch der letzte Schritt: Ein Klick auf den Ribbon-Eintrag **Kundenübersicht** soll das **Page**-Element im **Frame**-Element einblenden und mit den Kundendaten füllen (siehe Bild 3).

Die Code behind-Klasse des Fensters **MainWindow.xaml** sehen Sie in Listing 5. Hier sind neben der Konstruktor-Methode noch zwei weitere Methoden zu sehen, die durch die beiden Schaltflächen des Ribbons ausgelöst werden. Die zweite davon heißt **btnKundenuebersicht_Click**. Wir haben diese bereits mit der einzigen Anweisung gefüllt, die eine weitere Methode namens **ShowKundenuebersicht** aufruft – auch diese ist im Quellcode abgebildet (den Parameter **kundeID** benötigen wir hier noch nicht, daher wird der Standardwert **0** verwendet).

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }
    public bool KundeChanged { get; set; }
    Kundenuebersicht kundenuebersicht;
    private void btnKundenuebersicht_Click(object sender, RoutedEventArgs e) {
        ShowKundenuebersicht();
    }
    private void ShowKundenuebersicht(int kundeID = 0) {
        if (kundenuebersicht == null | KundeChanged == true) {
            kundenuebersicht = new Kundenuebersicht(kundeID);
            KundeChanged = false;
        }
        WorkZone.Content = kundenuebersicht;
    }
}
```

Listing 5: Code behind-Datei der Klasse **MainWindow.xaml**

Die Methode prüft, ob die Objektvariable **kundenuebersicht** bereits gefüllt ist. Damit sorgen wir dafür, dass die Kundenübersicht nur neu erstellt wird, wenn sie noch nicht vorhanden ist. Ein anderer Grund, die Kundenübersicht neu zu erstellen, ist eine Änderung in den zugrunde liegenden Daten. Wenn also beispielsweise nach dem erstmaligen Füllen der Kundenübersicht einer der Kundendatensätze geändert oder ein neuer Kundendatensatz hinzugefügt wurde, soll die Kundenübersicht auch neu geladen werden. Dies prüfen wir über die

EDM: Kundendetails verwalten

In unserer Lösung zur Verwaltung von Bestellungen spielt die Kundenverwaltung natürlich eine große Rolle. In diesem Artikel wollen wir uns ansehen, wie wir einen neuen Kunden anlegen oder einen vorhandenen Kunden bearbeiten können – eingebettet natürlich in unser Hauptfenster, das wir im Artikel »Kunden verwalten« beleuchten. Daher benötigen wir auch kein eigenes Fenster, um die Kundendetails anzuzeigen, sondern erstellen ein Page-Objekt mit den relevanten Informationen. Dieses nutzen wir dann sowohl zum Anlegen neuer Kunden als auch zum Bearbeiten vorhandener Kunden.

Die Seite zum Bearbeiten und Erstellen von Kunden soll in Aktion wie in Bild 1 aussehen. Wenn der Benutzer auf den Ribbon-Eintrag **Kundenübersicht** klickt, soll die Kundenliste erscheinen, über die per Doppelklick einer der Kunden in der hier gezeigten Detailansicht erscheinen soll. Wenn der Benutzer auf die Schaltfläche **Neuer Kunde** klickt, erscheint die gleiche Seite, allerdings mit einem leeren **Kunden**-Objekt.

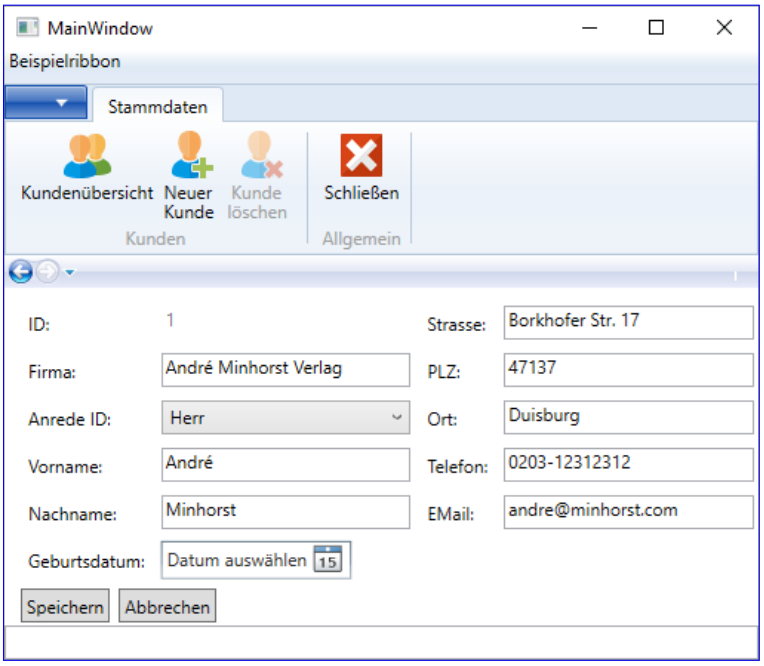


Bild 1: Die Seite zum Bearbeiten und Anlegen von Kunden

Kundendetails-Seite anlegen

Um die Kunden zu bearbeiten, legen Sie in unserem Projekt **Bestellverwaltung** ein Objekt des Typs **Seite** an. Wir wollen diese Seite in ein **Frame**-Element einblenden, wann immer die Bearbeitung eines Kunden nötig ist – oder wenn ein neuer Kunde angelegt werden soll. Erstmal müssen wir die Seite jedoch erstellen und mit den entsprechenden Steuerelementen sowie dem Code zum Füllen der Steuerelemente mit den Daten ausstatten. Die Seite soll im Entwurf wie in Bild 2 aussehen. Die zweite und die vierte Spalte haben wir so ausgelegt, dass sie sich beim Verbreitern des **Page**-Elements ebenfalls im gleichen Maße verbreitern – allerdings nur bis zu einer bestimmten maximalen Breite.

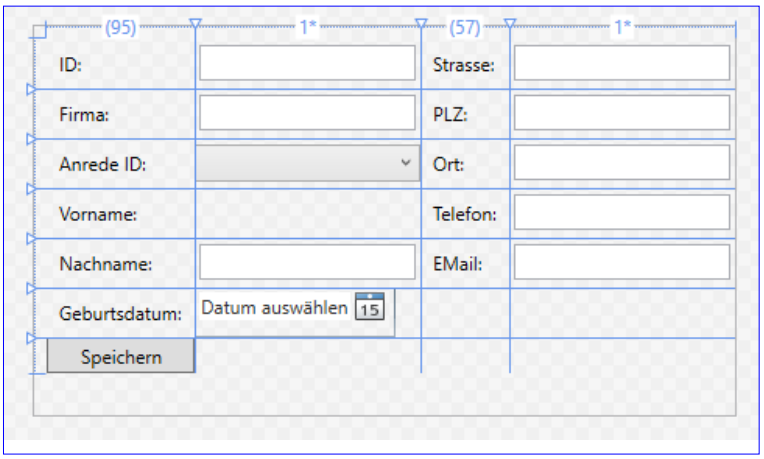


Bild 2: Die Seite zum Bearbeiten und Anlegen von Kunden im Entwurf

Standardwerte für Steuerelementeigenschaften

Den WPF-Code für diese Seite finden Sie in Listing 1. Zu Beginn des Codes haben wir im Breich **Page.**

```
<Page x:Class="Bestellverwaltung.Kundendetails"
      ...
      d:DesignHeight="250" d:DesignWidth="450"
      Title="Kundendetails">
    <Page.Resources>
        <Style TargetType="Label">
            <Setter Property="HorizontalAlignment" Value="Left"></Setter>
            <Setter Property="Margin" Value="3"></Setter>
            <Setter Property="VerticalAlignment" Value="Center"></Setter>
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="Height" Value="23"></Setter>
            <Setter Property="Margin" Value="3"></Setter>
            <Setter Property="VerticalAlignment" Value="Center"></Setter>
        </Style>
    </Page.Resources>
    <Grid x:Name="grid1" HorizontalAlignment="Stretch" Margin="8,8,0,0" VerticalAlignment="Top">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" MaxWidth="300"/>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*" MaxWidth="300"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            ...
        </Grid.RowDefinitions>
        <Label Content="ID:" Grid.Column="0" />
        <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Text="{Binding kunde.ID, Mode=TwoWay,
            NotifyOnValidationError=true, ValidatesOnExceptions=true}" Width="50" IsEnabled="False" BorderBrush="Transparent"/>
        <Label Content="Firma:" Grid.Column="0" Grid.Row="1" />
        <TextBox x:Name="txtFirma" ... Text="{Binding kunde.Firma, ...}" />
        <Label Content="Anrede ID:" Grid.Column="0" Grid.Row="2" />
        <ComboBox x:Name="cboAnreden" Grid.Column="1" HorizontalAlignment="Stretch" Height="23" Grid.Row="2"
            ItemsSource="{Binding Anreden}" SelectedItem="{Binding kunde.Anreden}" DisplayMemberPath="Bezeichnung"
            SelectedValuePath="ID" VerticalAlignment="Center" Margin="3"></ComboBox>
        <Label Content="Vorname:" ... /><TextBox x:Name="txtVorname" ... Text="{Binding kunde.Vorname, ...}" />
        <Label Content="Nachname:" ... /><TextBox x:Name="txtNachname" ... Text="{Binding kunde.Nachname, ...}" />
        <Label Content="Geburtsdatum:" .../>
        <DatePicker x:Name="dpGeburtsdatum" ... SelectedDate="{Binding kunde.Geburtsdatum, ...}" Margin="3"/>
        <Label Content="Strasse:" ... /><TextBox x:Name="txtStrasse" ... Text="{Binding kunde.Strasse, ...}" />
        <Label Content="PLZ:" ... /><TextBox x:Name="txtPLZ" Text="{Binding kunde.PLZ, ...}" />
        <Label Content="Ort:" ... /> <TextBox x:Name="txtOrt" ... Text="{Binding kunde.Ort, ...}" />
        <Label Content="Telefon:" ... /><TextBox x:Name="txtTelefon" ... Text="{Binding kunde.Telefon, ...}" />
        <Label Content="EMail:" ... /><TextBox x:Name="txtEMail" ... Text="{Binding kunde.EMail, ...}" />
        <Button x:Name="btnSpeichern" Click="btnSpeichern_Click" Grid.Row="6" Grid.Column="0" Height="23"
            Content="Speichern"></Button>
    </Grid>
</Page>
```

Listing 1: WPF-Code für die Seite **Kundendetails**

Resources einige **Style**-Elemente hinzugefügt. **Style**-Elemente dienen der Festlegung von Werten für bestimmte Attribute von WPF-Elementen. Das erste verwendet für das Attribut **TargetType** den Wert **Label**, das heißt, dass es Standardwerte für **Label**-Elemente vorgibt. In diesem Fall legen wir mit den untergeordneten Elementen des Typs **Setter** jeweils die Eigenschaft (**Property**) und den Wert (**Value**) der Voreinstellung fest. Das Attribut **HorizontalAlignment** soll den Standardwert **Left** erhalten, **Margin** den Wert **3** und **VerticalAlignment** den Wert **Center**. Diese Werte brauchen wir dann nicht mehr explizit für die Steuerelemente des Typs **Label** festzulegen. Sollten wir an einer bestimmten Stelle einmal andere Werte benötigen, tragen Sie diese einfach für das entsprechende **Label**-Element ein.

Auch für das **TextBox**-Steuerelement haben wir drei Attribute vordefiniert, nämlich **Height** (Wert: **23**), **Margin** (**3**) und **VerticalAlignment** (**Center**).

Damit erhalten wir nicht nur den Vorteil, diese Werte nicht mehr explizit einstellen zu müssen. Wir brauchen außerdem auch Änderungen, die alle Elemente des gleichen Typs betreffen, nur noch an einer Stelle vorzunehmen.

Das Grid

Unser Grid soll vier Spalten und sieben Zeilen enthalten. Für die Spalten 1 und 3 haben wir die Breite **Auto** festgelegt, für die Spalten 2 und 4 den Wert ***** sowie die maximale Breite von **300**.

Die einzelnen Steuerelemente weisen wir den Zellen des Grids zu, indem wir die Eigenschaften **Grid.Column** und **Grid.Row** auf die entsprechenden 0-basierten Werte einstellen.

Die gebundenen Steuerelemente

Wir haben drei verschiedene Arten gebundener Steuerelemente: **TextBox**, **ComboBox** und **DatePicker**. Das erste **TextBox**-Element heißt **txtID** und soll die Eigenschaft **ID** der Klasse **kunde** aufnehmen. Woher die Klasse **kunde** und die enthaltenen Eigenschaften bezogen werden, stellen wir später in der Code behind-Datei ein.

Hier ist zunächst wichtig, dass **kunde.ID** als Inhalt des Attributs **Text** innerhalb eines **Binding**-Elements angegeben wird:

```
Text="{Binding kunde.ID,
Mode=TwoWay,
NotifyOnValidationError=true,
ValidatesOnExceptions=true}"
```

Mit **Mode=TwoWay** legen wir außerdem fest, dass Änderungen nicht nur von der Eigenschaft zum Steuerelement, sondern auch in die umgekehrte Richtung weitergegeben werden. Um die übrigen beiden Attribute **NotifyOnValidationError** und **ValidatesOnExceptions** kümmern wir uns später.

Beim ersten **TextBox**-Element **txtID** haben wir außerdem das Attribut **IsEnabled** auf **False** eingestellt und **BorderBrush** auf **Transparent**. Dadurch kann der Benutzer den Inhalt nicht bearbeiten, der transparente Rahmen weist optisch darauf hin, dass es sich nicht um ein editierbares Feld handelt.

Die übrigen Textfelder referenzieren im Wert für das Attribut **Text** jeweils die entsprechende Eigenschaft der Klasse **kunde**, also etwa **kunde.Vorname**, **kunde.Nachname** und so weiter. Die Benennung der Textfelder beginnt jeweils mit **txt**.

ComboBox zur Auswahl der Anreden

Das **ComboBox**-Steuerelement **cboAnreden** soll die aktuelle Anrede des Kunden-Datensatzes anzeigen sowie die Auswahl der verfügbaren Anreden ermöglichen. Dazu binden wir es zunächst über das Attribut **ItemsSource** an eine Klasse namens **Anreden**. Der ausgewählte Eintrag wird über die Eigenschaft **SelectedItem** angegeben, welche die Eigenschaft **Anreden** des Objekts **kunde** referenziert. Das Attribut **DisplayMemberPath** gibt an, welche Eigenschaft im Kombinationsfeld angezeigt werden soll (**Bezeichnung**), **Selected-**

ValuePath legt fest, welches die gebundene Eigenschaft des Quellobjekts ist – hier das Feld **ID**.

DatePicker für das Geburtsdatum

Fehlt noch das Steuerelement **dpGeburtsdatum**, das als **DatePicker** die Auswahl des Geburtstages des jeweiligen Kunden vereinfacht. Es ist an das Feld **kunde.Geburtsdatum** geknüpft.

Speichern-Schaltfläche

Fehlt noch die **Speichern**-Schaltfläche **cmdSpeichern**, welche als Wert für das Attribut **cmdSpeichern_Click** den Namen der Methode angibt, die beim Anklicken der Schaltfläche aufgerufen werden soll.

Abbrechen-Schaltfläche

Soll das Anlegen des neuen Kunden oder die Änderung am angezeigten Kunden abgebrochen werden, klickt der Benutzer auf die **Abbrechen**-Schaltfläche. Diese schließt die Seite **Kundendetails** wieder, ohne darin vorgenommene Änderungen zu speichern.

Reihenfolge der Steuerelemente

Wenn der Benutzer die Daten eingibt, sollte er per Tabulator-Taste von Eingabefeld zu Eingabefeld springen können, und zwar in diesem Fall erst in der linken Spalte von oben nach unten und dann auf der rechten Spalte. Damit dies gewährleistet ist, legen Sie einfach die Elemente in der **.xaml**-Datei in der richtigen Reihenfolge an. Auf diese Weise brauchen Sie zusätzliche Steuerelemente einfach nur an der entsprechenden Stelle im Code einzufügen.

Die zweite Möglichkeit wäre, das Attribut **TabIndex** der betroffenen Steuerelemente mit einem nullbasierten Index zu versehen. Wenn Sie hier mittendrin ein Steuerelement einfügen wollen, müssen Sie die Indexwerte der folgenden Elemente allerdings nachträglich anpassen.

Erstes Steuerelement aktivieren

Beim Öffnen der Seite, gleichwohl ob diese einen neuen Kunden oder einen vorhandenen Kunden anzeigt, soll das erste Feld, also **Firma**, den Fokus erhalten – so kann der Benutzer dort direkt mit der Eingabe oder Änderung der Daten beginnen. Dies erledigen wir in der Ereignismethode, die beim Laden der Seite ausgelöst wird. Zuerst die Angabe der Ereignismethode im **Page**-Element:

```
<Page x:Class="Bestellverwaltung.Kundendetails" ...
      Title="Kundendetails" Loaded="Page_Loaded">
```

Und hier ist die Methode mit der einzigen Anweisung, welche lediglich die **Focus**-Methode für das Steuerelement **txtFirma** aufruft:

```
private void Page_Loaded(object sender, RoutedEventArgs e) {
    txtFirma.Focus();
}
```

Code behind-Modul

Das Code behind-Modul namens **Kundendetails.xaml.cs** referenziert die folgenden Bibliotheken:

```
using System.Windows.Controls;
using System.Windows;
using System.Collections.Generic;
using System.Linq;
```

Den Rest der Klasse, der sich wie der Rest des Codes des Beispielprojekts im Namespace **Bestellverwaltung** befindet, sieht im Überblick wie in Listing 2 aus. Die enthaltene Klasse erbt vom Objekt **Page**.

Die ersten sechs Zeilen definieren Ereignisse, die beim Speichern eines Kunden oder beim Abbrechen der Eingabe in die Seite ausgelöst werden und die von anderen Klassen wie etwa dem **Window**-Objekt, welches unser **Page**-Objekt **Kundendetails.xaml** anzeigt, implementiert werden können, um auf das Speichern von geänderten oder neu angelegten Kunden zu reagieren. Die Ereignisse werden von der Methode **btnSpeichern_Click**, die durch einen Klick auf die Schaltfläche **btnSpeichern** ausgelöst wird, angestoßen oder von der Methode **btnAbbrechen_Click**. Eine genaue

PropertyChanged in der Praxis

Der Artikel »Basics: PropertyChanged« zeigt, wie die Schnittstelle INotifyPropertyChanged funktioniert. In unserer Beispielanwendung Bestellverwaltung verwenden wir diese Schnittstelle an einer Stelle, wo wir einen Ribbon-Button in Abhängigkeit vom Wert einer Eigenschaft aktivieren und deaktivieren, die das PropertyChanged-Ereignis auslöst. Im vorliegenden Artikel zeigen wir, wie dies im Detail funktioniert.

Hintergrund

Unter Access war es üblich, den Zustand von Steuerelementen direkt per Code durch Ändern der entsprechenden Eigenschaft zu ändern. Das ist natürlich auch unter WPF/C# möglich. Allerdings gehört es dort zum guten Ton, die Definition der Benutzeroberfläche mit XAML und die Anwendungslogik in C#-Klassen voneinander zu trennen, und zwar in der Form, dass der Zustand der Benutzeroberfläche entweder direkt in XAML definiert wird oder aber durch die Bindung an Elemente der Anwendungslogik ermittelt wird. Keinesfalls jedoch sollten die Elemente der Anwendungslogik auf jene der Benutzeroberfläche zugreifen. Nun wollen wir in unserem Beispiel eine Schaltfläche im Ribbon, mit der man einen in einer Liste markierten Eintrag löschen kann, nur aktivieren, wenn die Liste auch im unteren Bereich des Fensters angezeigt wird. Ist das nicht der Fall, weil beispielsweise gerade die Details eines Kundendatensatzes dort abgebildet werden, soll die Schaltfläche deaktiviert sein. Unter Access hätte man nun ein Ereignis, das beim Einblenden der Kundenliste ausgelöst wird, genutzt, um die Schaltfläche zum Löschen eines Listeneintrags zu aktivieren oder zu deaktivieren.

Hier wollen wir nun aber moderner vorgehen und Benutzeroberfläche und Anwendungslogik so weit wie möglich voneinander trennen. Dazu legen wir im Code behind-Modul eine öffentliche Eigenschaft an, an die wir das **IsEnabled**-Attribut der betroffenen Ribbon-Schaltfläche binden. Den Wert dieser Eigenschaft stellen wir an einer geeigneten Stelle ein, in diesem Fall dem Ereignis **Navigated** des **Frame**-Objekts namens **Workzone**, das unsere verschiedenen **Page**-Objekte wie die Kundendetails oder die Kundenliste einblendet.

Umsetzung

Für die Umsetzung deklarieren wir zunächst eine **Boolean**-Variable namens **kundeLoeschbar**:

```
bool kundeLoeschbar;
```

Für diese legen wir eine öffentliche Eigenschaft namens **KundeLoeschbar** an und statten diese mit dem üblichen Getter und Setter aus. Dem Setter fügen wir gleich noch den Aufruf der Methode **OnPropertyChanged** hinzu und übergeben dieser mit einem neuen Objekt des Typs **PropertyChangedEventArgs** den Namen der Eigenschaft:

```
public bool KundeLoeschbar {
    get { return kundeLoeschbar; }
    set {
        kundeLoeschbar = value;
        OnPropertyChanged(new PropertyChangedEventArgs("KundeLoeschbar"));
    }
}
```

Um die Schnittstelle **INotifyPropertyChanged** zu implementieren, fügen wir das entsprechende Element zur Definition der Klasse **MainWindow** hinzu:

```
public partial class MainWindow : RibbonWindow, INotifyPropertyChanged {
```

Das einzige zu implementierende Element dieser Schnittstelle sieht so aus:

```
public event PropertyChangedEventHandler PropertyChanged;
```

Fehlt noch die Methode **OnPropertyChanged**, die zwischen den Setter und das Auslösen des Ereignisses gesetzt wird:

```
protected virtual void OnPropertyChanged(PropertyChangedEventArgs e) {
    if (PropertyChanged != null) {
        PropertyChanged(this, e);
    }
}
```

In der Ribbon-Definition stellen wir nun das Attribut **IsEnabled** so ein, dass es an die Eigenschaft **KundeLoeschbar** gebunden wird. **KundeLoeschbar** ist ein Element der Code behind-Klasse des **Window**-Elements. Diese Bindung können wir auf verschiedene Arten herstellen – wir stellen zwei davon vor. Die erste geht davon aus, dass Sie dem **Window**-Element einen Namen gegeben haben, in diesem Fall **wndMain**:

```
<RibbonWindow x:Class="Bestellverwaltung.MainWindow" ... Title="MainWindow" Height="450" Width="525" x:Name="wndMain">
```

Dann können wir **wndMain** als **ElementName** angeben und die Eigenschaft **KundeLoeschbar** als **Path**:

```
<RibbonButton Name="btnKundeLoeschen" Label="Kunde löschen" ... IsEnabled="{Binding ElementName=wndMain,
Path=KundeLoeschbar}"></RibbonButton>
```

Alternativ referenzieren wir das **Window**-Objekt über das Attribut **RelativeSource**. Hier ist keine Benennung von **MainWindow** nötig:

```
<RibbonButton Name="btnKundeLoeschen" Label="Kunde löschen" ... IsEnabled="{Binding Path=KundeLoeschbar,
RelativeSource={RelativeSource Mode=FindAncestor, AncestorType=Window}}" ></RibbonButton>
```

Der Ausdruck **{Binding Path=KundeLoeschbar, RelativeSource={RelativeSource Mode=FindAncestor, AncestorType=Window}}** sucht nach dem Wert einer Eigenschaft namens **KundeLoeschbar**, wobei das Quellobjekt das nächste übergeordnete Element (**FindAncestor**) des Typs **Window** ist. Nun fehlt noch der Schritt, an dem wir die Eigenschaft ändern, damit sich die Änderung auf die Aktivierung der Ribbon-Schaltfläche auswirkt. Dies erledigen wir in dem Ereignis, das durch das Ändern der angezeigten Seite ausgelöst wird:

```
private void WorkZone_Navigated(object sender, System.Windows.Navigation.NavigationEventArgs e) {
    switch (WorkZone.Content.ToString()) {
```