AUSGABE 06/2020

DATENBANK ENTWICKLER

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

TOP-THEMEN:

André Minhorst Verlag

Das GridSplitter-Steuerelement

Das Grid-Steuerelement dürfte Ihnen als Leser dieses Magazins bekannt sein – wir verwenden es sehr oft, um die Position von Steuerelementen wie Bezeichnungsfeldern, Textfeldern oder Schaltflächen einzustellen. Im Zusammenhang mit dem Grid-Steuerelement müssen Sie ein weiteres Steuerelement unbedingt kennenlernen: das GridSplitter-Steuerelement. Dieses unterteilt das Grid zwischen zwei bestimmten Zeilen oder Spalten und bietet die Möglichkeit, dass Sie das Verhältnis der Höhe beziehungsweise Breite der Steuerelemente über/unter beziehungsweise links/rechts vom GridSplitter-Steuerelement einstellen können. Damit vergrößern Sie dann den jeweils benötigten Bereich.

Vielleicht kennen Sie ähnliche Lösungen von Access: Dort mussten Sie ein Rectangle-Steuerelement zwischen den zu vergrößernden/verkleinernden Steuerelementen platzieren. Mit dem Ereignis **Bei Mausbewegung** wurde dann beim Ziehen des Rechtecks zu einer Seite die Position des Rechtecks geändert beziehungsweise die Größe der Steuerelemente, die sich auf einer der beiden Seiten vom Rechteck befunden haben. Das Problem: Man musste die Größe und Position jedes Steuerelements, das durch die Größenänderung betroffen war, per Code einstellen.

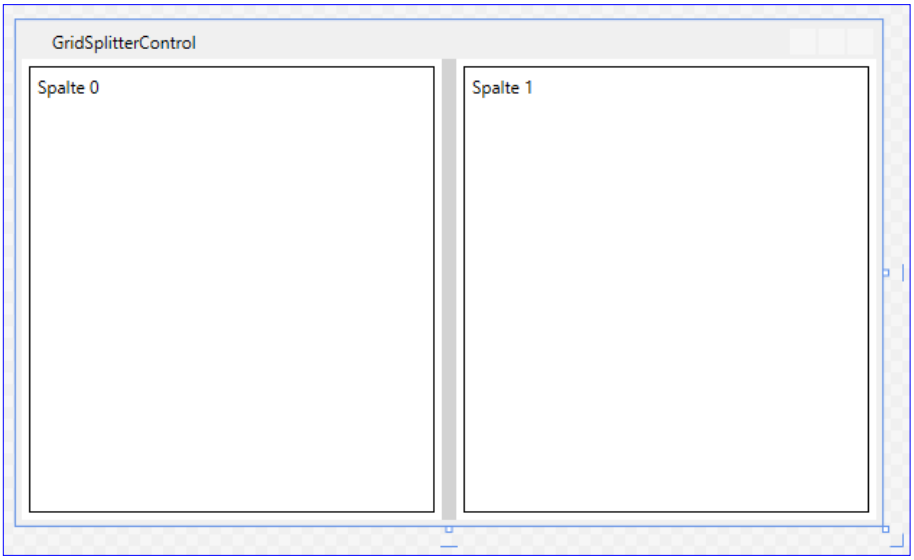

Bild 1: Das **GridSplitter**-Steuerelement zwischen zwei Steuerelementen

Vertikaler GridSplitter

Die Variante, die Sie mit dem **GridSplitter**-Steuerelement kennenlernen, ist ungleich komfortabler. Sie setzt allerdings voraus, dass Sie die Steuerelemente in den Spalten und Zeilen eines **Grid**-Steuerelements organisiert haben. Das **GridSplitter**-Steuerelement platzieren Sie dabei in einer eigens dafür vorgesehenen Spalte oder Zeile im Grid. Schauen wir uns das am Beispiel eines vertikalen GridSplitters an, der die Bereiche rechts und links vergrößern oder verkleinern soll – je nachdem, in welche Richtung Sie diesen bewegen. Der einfache Entwurf sieht wie in Bild 1 aus.

Den entsprechenden XAML-Code finden Sie hier: Wir haben mit den Elementen **Grid.ColumnDefinitions** und **ColumnDefinition** drei Spalten in einem Grid definiert. Die Spalten links und rechts erhalten für die Eigenschaft **Width** den Wert ***, was bedeutet, dass diese sich den Platz in der Breite aufteilen, der nicht von Spalten mit fest vorgegebener Breite eingenommen wird. Eine

solche Spalte legen wir in der Mitte an und weisen ihr die Breite **10** zu. Damit hat diese beim Öffnen die Breite 10 und die beiden äußeren Spalten teilen sich den restlichen Platz auf.

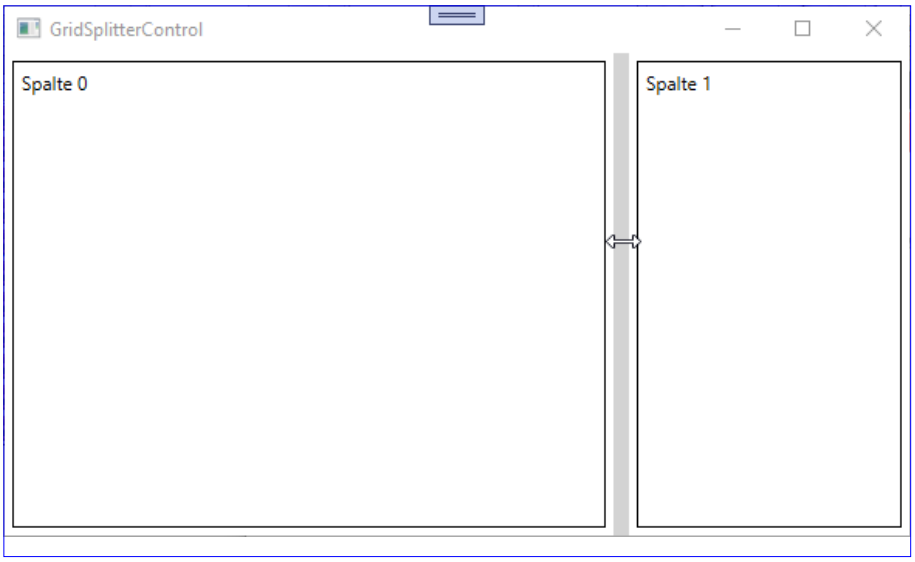

Bild 2: Einstellen der Größe der Bereiche links und rechts vom vertikalen **GridSplitter**-Steuerelement

Der linken und rechten Spalte fügen wir jeweils ein **Label**-Steuerelement zu, welche den kompletten Platz in den Spalten einnehmen. Der mittleren Spalte weisen wir ein **GridSplitter**-Steuerelement zu, das wir mit einem hellgrauen Hintergrund versehen und das durch den Wert **Stretch** für die Eigenschaft **HorizontalAlignment** die gesamte Breite der mittleren Spalte des Grids einnimmt. Außerdem legen wir vorsichtshalber den Wert des Attributs **ResizeDirection** auf **Columns** fest:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"></ColumnDefinition>
        <ColumnDefinition Width="10"></ColumnDefinition>
        <ColumnDefinition Width="*"></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label Grid.Column="0" BorderBrush="Black" BorderThickness="1">Spalte 0</Label>
    <GridSplitter Grid.Column="1" ResizeDirection="Columns" Background="LightGray" HorizontalAlignment="Stretch"></GridSplitter>
    <Label Grid.Column="2" BorderBrush="Black" BorderThickness="1">Spalte 1</Label>
</Grid>
```

Starten wir das Projekt, können wir den Gridsplitter per Herunterdrücken der linken Maustaste anfassen und durch Bewegen der Maus nach links oder rechts verschieben. Die Steuerelemente, die sich in der **Grid**-Spalte links und rechts davon befinden, werden dann entsprechend der Breite der jeweiligen Spalte angepasst (siehe Bild 2).

Horizontaler GridSplitter

Auf ähnliche Weise arbeiten wir mit einem **GridSplitter**-Steuerelement, mit dem wir das Grid in einen Teil über und einen Teil unter dem GridSplitter einteilen. Dazu stellen wir nun mehrere Zeilen her, von denen die mittlere das **GridSplitter**-Steuerelement aufnimmt:

```
<Grid>
    <Grid.RowDefinitions>
```

```
        <RowDefinition Height="*"></RowDefinition>
        <RowDefinition Height="10"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <Label Grid.Row="0" BorderBrush="Black" BorderThickness="1">Zeile 0</Label>
    <GridSplitter Grid.Row="1" ResizeDirection="Rows" Background="LightGray" HorizontalAlignment="Stretch"></GridSplitter>
    <Label Grid.Row="2" BorderBrush="Black" BorderThickness="1">Zeile 1</Label>
</Grid>
```

Wir brauchen an dem Beispiel gar nicht so viel zu ändern – wir legen statt des **Grid.ColumnDefinitions**-Elements ein **Grid.RowDefinitions**-Element an, ordnen diesen statt **ColumnDefinition**-Elementen **RowDefinition**-Elemente zu, die wir überdies statt mit der Eigenschaft **Width** mit der Eigenschaft **Height** ausstatten. Die oberste und die unterste Zeile erhalten den Wert ***** für die Höhe und die mittlere wieder den Wert **10**.

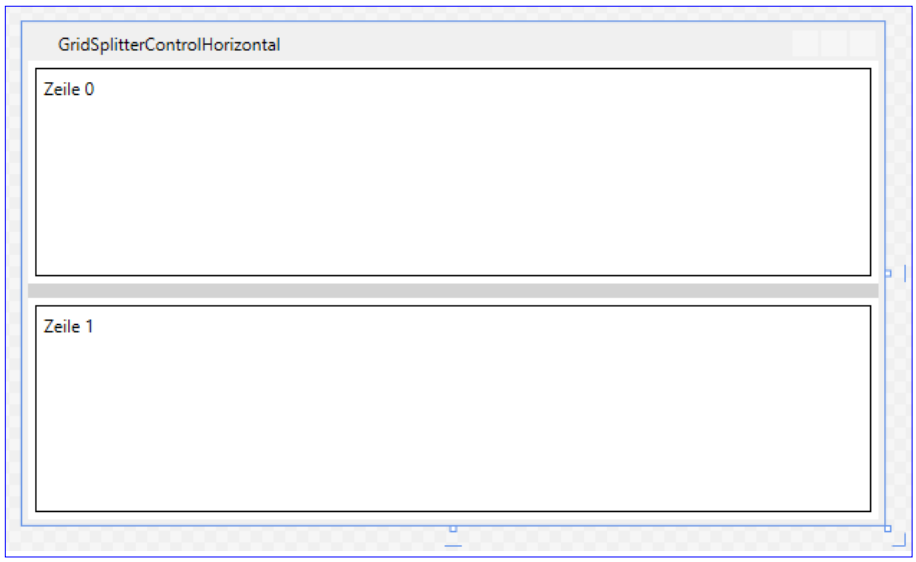

Bild 3: Entwurf des Grids mit dem horizontalen **GridSplitter**

Schließlich weisen wir die beiden Bezeichnungsfelder den Zeilen mit dem Index **0** beziehungsweise **2** und das **GridSplitter**-Element der Zeile mit dem Index **1** zu. Dieses statten wir außerdem mit dem Wert **Rows** für das Attribut **ResizeDirection** aus. Damit erhalten wir den Entwurf aus Bild 3.

Wenn wir die Anwendung starten, sieht das Ergebnis wie in Bild 4 aus.

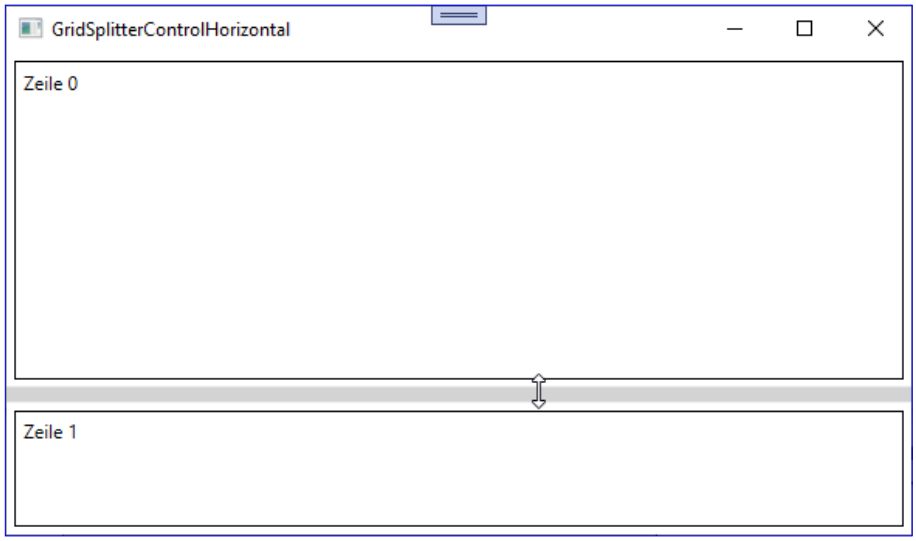

Bild 4: Einstellen der Höhe der Bereiche unter und über dem horizontalen **GridSplitter**

Das Attribut **ResizeDirection** ist standardmäßig auf den Wert **Auto** festgelegt, was in der Regel auch ausreicht – wir haben das Attribut hier nur auf die Werte **Columns** beziehungsweise **Rows** gesetzt, um seine Funktion zu dokumentieren.

GridSplitter mit Vorschau

In den bisherigen Beispielen wurden der GridSplitter und die dadurch beeinflussten Steuerelemente jeweils direkt durch das Ziehen mit der Maus angepasst.

Sie können auch festlegen, dass der Gridsplitter sich beim Ziehen mit der Maus bewegt, aber noch ohne seine Position oder die Größe der beteiligten Steuerelemente anzupassen (siehe Bild 5). Für diesen Fall stellen Sie das Attribut **ShowsPreview** auf den Wert **True** ein:

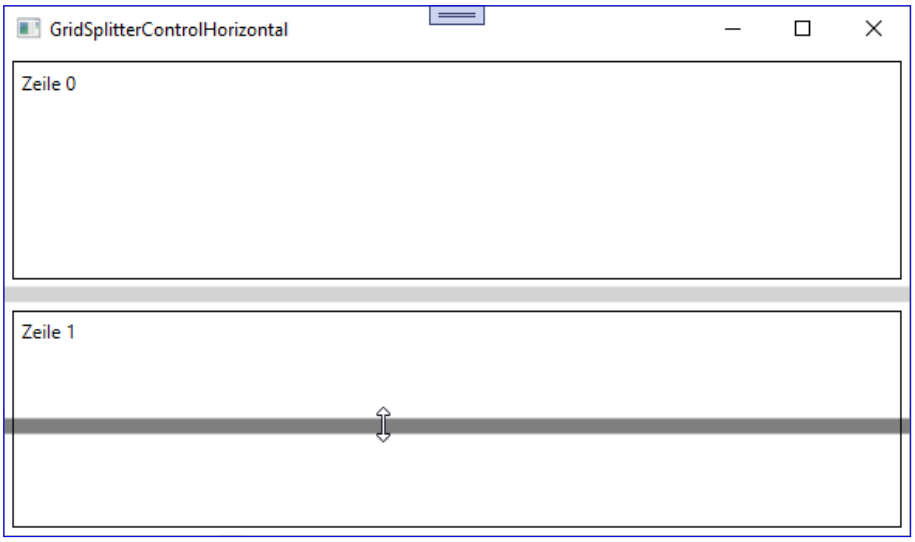

Bild 5: **GridSplitter**-Steuerelement mit Vorschau

```
<GridSplitter Grid.Row="1" ResizeDirection="Rows" Background="LightGray" HorizontalAlignment="Stretch"
ShowsPreview="True"></GridSplitter>
```

Mehrere GridSplitter

Sie können auch mehrere **GridSplitter**-Steuerelemente in einem Fenster nutzen. Im ersten Beispiel schauen wir uns an, wir Sie mehrere **GridSplitter**-Steuerelemente in einer Ebene nutzen können.

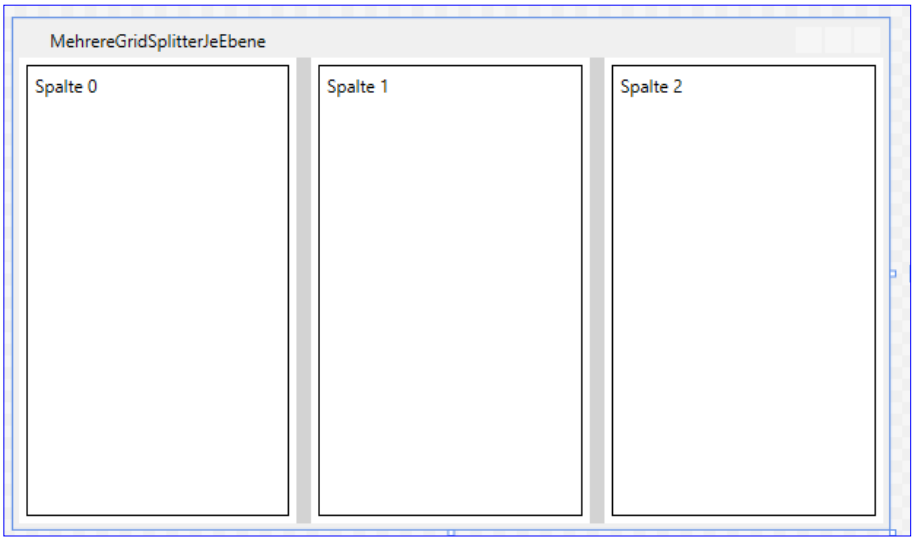

Bild 6: Mehrere **GridSplitter**-Steuerelement in einer Ebene, Entwurfsansicht

Dabei fügen wir zwei **ColumnDefinition**-Elemente zum Element **Grid.ColumnDefinitions** hinzu, von denen die letzte wieder * als Breite erhält und die vorletzte die Breite **10**. Außerdem legen wir für die letzte Spalte ein weiteres Label-Steuerelement an und für die vorletzte ein zweites **GridSplitter**-Steuerelement (Entwurf siehe Bild 6):

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"></ColumnDefinition>
        <ColumnDefinition Width="10"></ColumnDefinition>
        <ColumnDefinition Width="*"></ColumnDefinition>
```

Das Kalendersteuerelement

.NET bietet schon in der Grundausstattung sehr viele Steuerelemente für WPF-Anwendungen – hinzu kommen noch solche von Drittanbietern. Wir wollen nach und nach die WPF-Steuerelemente von .NET vorstellen, die wir noch nicht in den vorherigen Ausgaben beleuchtet haben. Und dort haben wir uns bisher ja nur um Standard-Steuerelemente wie TextBox, Button, ComboBox oder die diversen Steuerelemente zum Strukturieren anderer Steuerelemente gekümmert. Im vorliegenden Artikel schauen wir uns die Grundlagen zum Kalendersteuerelement an

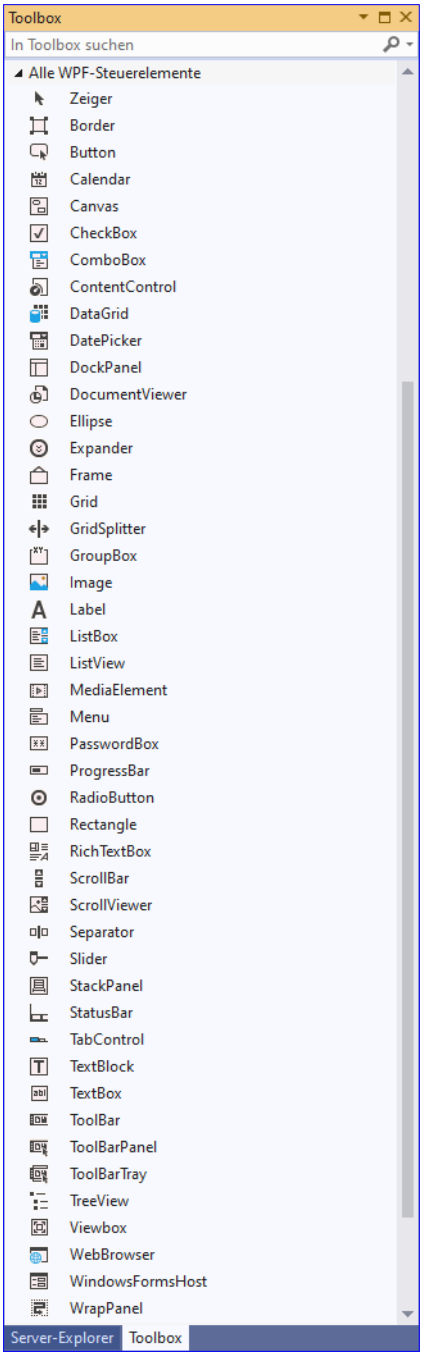

Bild 1: .WPF-Steuerelemente

Wenn Sie die Entwurfsansicht einer XAML-Datei anzeigen und die Toolbox aktivieren, sehen Sie im oberen Bereich namens **Häufig verwendete WPF-Steuerelemente** die oft verwendeten Steuerelemente wie **Button**, **CheckBox**, **ComboBox** oder **TextBox**. Der Bereich **Alle WPF-Steuerelemente** ist oft ausgeblendet, liefert aber noch viele weitere Steuerelemente, die Sie ohne weitere Handgriffe in Ihre Anwendungen integrieren können. Diesen Bereich sehen Sie in Bild 1.

Im vorliegenden Artikel wollen wir mit dem **Calendar**-Steuerelement beginnen. Dieses ziehen Sie im einfachsten Fall per Drag and Drop aus der Toolbox auf den aktuell geöffneten XAML-Entwurf (siehe Bild 2).

Im WPF-Code erscheint dann folgende Zeile:

```
<Calendar HorizontalAlignment="Left" Margin="33,27,0,0"
VerticalAlignment="Top"/>
```

In der laufenden Anwendung zeigt das **Calendar**-Steuerelement standardmäßig das aktuelle Datum an. Sie können andere Tage markieren, das aktuelle Datum bleibt jedoch farbig hinterlegt. Das **Calendar**-Steuerelement zeigt immer sechs Zeilen mit den Tagen jeweils einer Woche an. Da kein Monat 42 Tage enthält, werden die Tage des vorherigen und des folgenden Monats eingeblendet. Klicken Sie einen Tag an, der nicht zum aktuellen Monat gehört, wechselt die Ansicht zum Monat dieses Tages und markiert diesen

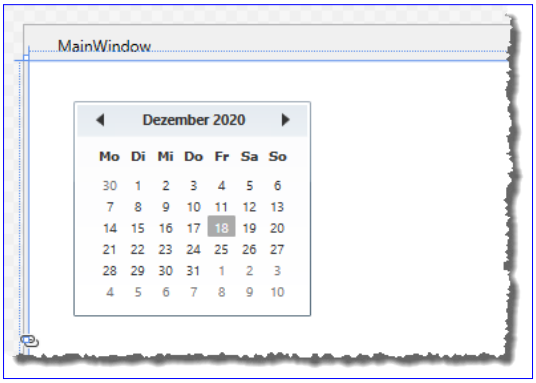

Bild 2: Das **Calendar**-Steuerelement

Tag. Eine andere Art, zum vorherigen oder zum nachfolgenden Monat zu wechseln, ist ein Mausklick auf einen der Pfeile links und rechts vom Monatsnamen.

Größe anpassen

Im Gegensatz zu vielen anderen Steuerelementen passt sich die Größe des **Calendar**-Steuerelements nicht automatisch an das umliegende Element an. Wenn Sie beispielsweise einem Grid ein **Button**-Element hinzufügen, nimmt dieses direkt den vollständigen Platz im Grid ein, solange Sie dies nicht mit entsprechenden Eigenschaften einschränken. Das Calendar-Steuerelement wird schlicht an der Stelle, an der es eingefügt wird, in der Standardgröße angezeigt.

Startdatum einstellen

Wenn Sie wollen, dass das **Calendar**-Steuerelement ein anderes Datum als das des heutigen Tages anzeigt, können Sie dies mit der Eigenschaft **DisplayDate** erreichen. Diese stellen Sie beispielsweise auf ein fixes Datum ein:

```
<Calendar DisplayDate="1.1.2021"></Calendar>
```

Bereits in der Entwurfsansicht zeigt das **Calendar**-Steuerelement dann den Monat **Januar 2021** an. Starten wir die Anwendung dann, bleibt es auch dabei – das **Calendar**-Steuerelement zeigt den **Januar 2021** an, aber markiert nicht den **1.1.2021**.

Liest man in der Dokumentation nach, erfährt man schnell, dass mit **DisplayDate** tatsächlich nur der Monat des mit **DisplayDate** angegebenen Datums eingestellt werden kann.

Wie aber können wir tatsächlich das Datum festlegen, das beim Öffnen markiert werden soll? Das erledigen Sie mit der Eigenschaft **SelectedDate**:

```
<Calendar SelectedDate="1.1.2021"></Calendar>
```

Damit wird das aktuelle Datum aber nach wie vor grau hinterlegt, während das mit **SelectedDate** angegebene Datum einen hellblauen Hintergrund erhält (siehe Bild 3).

Bild 3: Datum einstellen mit **SelectedDate**

Aktuelles Datum nicht markieren

Wenn Sie nicht möchten, dass das aktuelle Datum durch einen farbigen Hintergrund markiert wird, stellen Sie die Eigenschaft **IsTodayHighlighted** auf **False** ein:

```
<Calendar IsTodayHighlighted="False"></Calendar>
```

Anderen Wochentag als ersten Tag anzeigen

Normalerweise zeigt das **Calendar**-Steuerelement den Montag in der ersten Spalte an. Wenn Sie einen anderen Tag anzeigen wollen, beispielsweise den Samstag, legen Sie dies mit dem Attribut **FirstDayOfWeek** fest, das die englische Wochentagsbezeichnung erwartet – hier beispielsweise **Saturday**:

```
<Calendar FirstDayOfWeek="Saturday"></Calendar>
```

Einzelnes Datum auswählen

Das Attribut **SelectionMode** legt fest, ob einzelne Datumsangaben oder Datumsbereiche ausgewählt werden können. Dazu stellt diese Eigenschaft die folgenden Werte zur Verfügung:

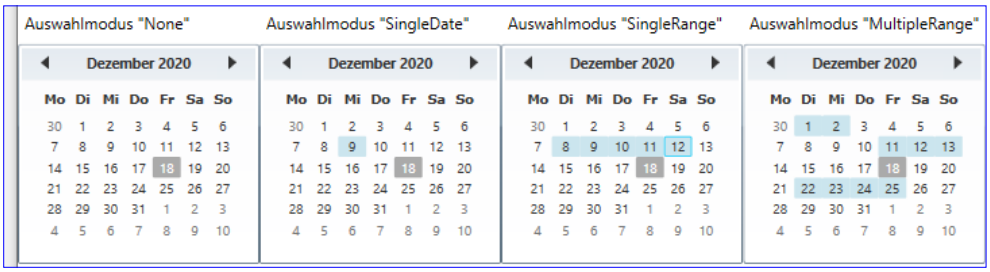

Bild 4: Verschiedene Auswahlmodi des **Calendar**-Steuerelements

- **None**: Es kann gar kein Datum ausgewählt werden.
- **SingleDate**: Es kann ein einzelnes Datum ausgewählt werden, und zwar per Mausklick auf das gewünschte Datum.
- **SingleRange**: Es kann ein einfacher Datumsbereich ausgewählt werden.
- **MultipleRange**: Es können mehrere Datumsbereiche ausgewählt werden.

Die Möglichkeiten dieser vier Einstellungen sehen Sie in Bild 4.

Einzelnen Datumsbereich auswählen

Wenn Sie die Einstellung **SingleRange** für die Eigenschaft **SelectionMode** angeben, können Sie einen oder mehrere zusammenhängende Tage auswählen. Das gelingt auf verschiedene Arten:

- Durch Anklicken des ersten Tages und Anklicken des letzten Tages bei gedrückter Umschalttaste.
- Durch Mausklick auf den ersten Tag und Ziehen der Markierung bei gedrückter Maustaste bis zum letzten Tag.

Wenn Sie einen Zeitraum wählen wollen, der sich über den aktuell angezeigten Monat hinaus erstreckt, können Sie den ersten Tag markieren, dann mit den **Nach links**- oder **Nach rechts**-Schaltflächen zum gewünschten Monat wechseln und dann bei gedrückter Umschalttaste den letzten Tag des Datumsbereichs markieren.

Oder Sie markieren den ersten Tag und klicken dann bei gedrückter Umschalttaste jeweils auf einen der Tage, die zwar im Kalender angezeigt werden, aber nicht zum aktuellen Monat gehören und wechseln so zum vorherigen oder folgenden Monat. Haben Sie den Zielmonat erreicht, klicken Sie – immer noch bei gedrückter Umschalttaste – auf den gewünschten Tag.

Mehrere Datumsbereiche auswählen

Wählen Sie die Einstellung **MultipleRange** für **SelectionMode** aus, können Sie sogar mehrere Bereiche markieren. Für den ersten Bereich klicken Sie auf das erste Startdatum und ziehen dann die Maus bei gedrückter Maustaste bis zum Ende des ersten Bereichs. Für die folgenden Bereiche drücken Sie die **Strg**-Taste und fügen die weiteren Bereiche auf die gleiche Weise

Commands verwenden

Wenn Sie unter Access und VBA Ereignisse programmiert haben, hat beispielsweise jedes Steuerelement, jede Ribbon-Schaltfläche und jeder Kontextmenü-Eintrag ein eigenes Ereignis erhalten – auch, wenn diese genau die gleiche Funktion auslösen sollten. Das war schon durch die unterschiedlichen Schnittstellen nötig. Unter WPF können Sie ein und dieselbe Funktion von verschiedenen Steuerelementen aus auslösen. Das gelingt allerdings nicht mit herkömmlichen Ereignismethoden, sondern mit sogenannten Commands. Was das ist und wie Sie es in Ihre Anwendungen einbauen können, zeigt der vorliegende Artikel.

Vorteil: Alles an einem Ort

Neben der Möglichkeit, benutzerdefinierte Ereignisse durch Commands abzubilden und diese von überall in der Anwendung aufzurufen, bieten Commands noch weitere Möglichkeiten: So gibt es auch eingebaute Commands, die etwa Basisfunktionalitäten der Zwischenablage wie Kopieren, Ausschneiden oder Einfügen bereitstellen. Wenn Sie einmal mit Access programmiert haben, wissen Sie, dass diese Funktionen immer in Form von API-Funktionen selbst bereitgestellt werden mussten und Sie nicht einfach die Systemfunktionen nutzen konnten. Diese würden Sie dann von jedem Ort, wo diese Funktion benötigt wird, aufrufen müssen, und zwar von der dafür vorgesehenen Ereignisporzedur aus. Wir schauen uns also in diesem Artikel zwei verschiedene Typen von Commands an: benutzerdefinierte und eingebaute Commands.

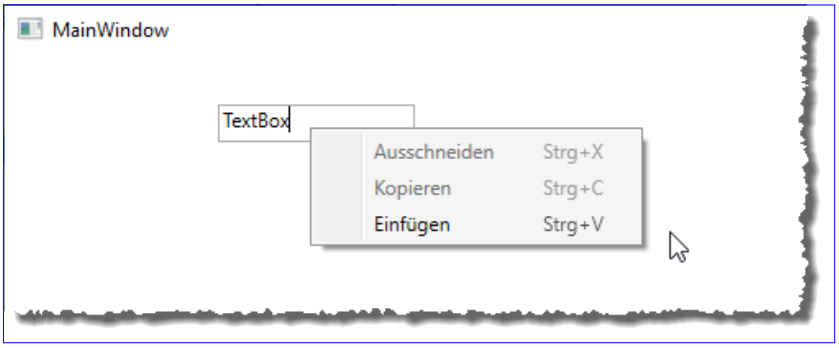

Bild 1: Aktivierte und deaktivierte Kontextmenü-Einträge

Vorteil: Tastenkombinationen

Ein weiterer Vorteil von Commands ist, dass Sie für ein benutzerdefiniertes Command eine Tastenkombination definieren können beziehungsweise, dass die Tastenkombinationen für eingebaute Commands bereits vorhanden sind – wie beispielsweise **Strg + C**, **Strg + X** oder **Strg + V** für **Kopieren**, **Ausschneiden** und **Einfügen**.

Vorteil: Aktivieren und Deaktivieren

Commands wie die zum Kopieren, Ausschneiden oder Einfügen des aktuell markierten Inhalts werden automatisch aktiviert oder deaktiviert – je nachdem, ob gerade ein Inhalt zum Kopieren oder Ausschneiden markiert beziehungsweise ob die Zwischenablage überhaupt Inhalt enthält, der aktuell eingefügt werden kann (siehe Bild 1). Auch für benutzerdefinierte Commands können Sie an einer Stelle festlegen, ob und wann diese aktiviert oder deaktiviert sind – abhängig vom jeweiligen Kontext. Während das für die Funktionen der Zwischenablage automatisch funktioniert, schauen wir uns später auch an, wie Sie benutzerdefinierte Commands von einer Stelle aus aktivieren und deaktivieren.

Definition von Commands

Wenn Sie ein Command definieren, erledigen Sie das unter Verwendung der **ICommand**-Schnittstelle. Diese sieht vor, dass Sie die folgenden beiden Methoden implementieren:

- **Execute**: Enthält die Anweisungen, die beim Aufruf des Commands ausgeführt werden sollen.
- **CanExecute**: Enthält Code, der festlegt, ob das Command gerade aktiviert oder deaktiviert ist.

Commands aufrufen

Um ein solches Command aufzurufen, müssen Sie für das jeweilige Steuerelement über ein sogenanntes Command binding festlegen, welches Command für das jeweilige Ereignis aufgerufen werden soll. Für eingebaute Commands ist dies einfacher als für benutzerdefinierte Commands. Wir schauen uns das für ein eingebautes Command an einem einfachen Beispiel an.

Dabei wollen wir die drei Befehle zum Ausschneiden, Kopieren und Einfügen in drei Schaltflächen unterbringen und mit diesen den Inhalt eines Textfeldes bearbeiten können. Der WPF-Code dafür sieht wie folgt aus:

```
<StackPanel Orientation="Horizontal">
    <Button x:Name="cmdAusschneiden" Command="ApplicationCommands.Cut">Ausschneiden</Button>
    <Button x:Name="cmdKopieren" Command="ApplicationCommands.Copy">Kopieren</Button>
    <Button x:Name="cmdEinfuegen" Command="ApplicationCommands.Paste">Einfügen</Button>
</StackPanel>
<TextBox x:Name="txtText" Grid.Row="1" Width="200"/>
```

Wir haben hier herkömmliche Schaltflächen angelegt, für die wir allerdings nicht das **Click**-Ereignis definiert haben, sondern mit dem **Command**-Attribut ein Command definiert haben, das beim Anklicken der Schaltfläche ausgelöst werden soll.

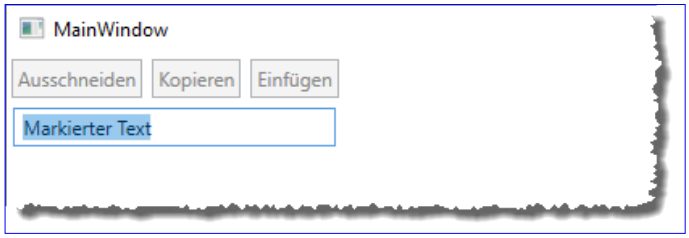

Bild 2: Schaltflächen, die nicht aktiviert werden

Das Ergebnis sieht wie in Bild 2 aus. Leider werden die Schaltflächen nicht aktiviert oder deaktiviert, wenn man einen Teil des Textes im Textfeld markiert. Dazu fehlt freilich noch ein Attribut – wir haben nämlich noch nicht festgelegt, auf welches Element der Benutzeroberfläche sich die Commands beziehen sollen. Das legen wir mit dem Attribut **CommandTarget** fest, welches per Binding das entsprechende Steuerelement referenziert:

```
<Button x:Name="cmdAusschneiden" Command="ApplicationCommands.Cut"
    CommandTarget="{Binding ElementName=txtText}">Ausschneiden</Button>
<Button x:Name="cmdKopieren" Command="ApplicationCommands.Copy"
    CommandTarget="{Binding ElementName=txtText}">Kopieren</Button>
<Button x:Name="cmdEinfuegen" Command="ApplicationCommands.Paste"
    CommandTarget="{Binding ElementName=txtText}">Einfügen</Button>
```

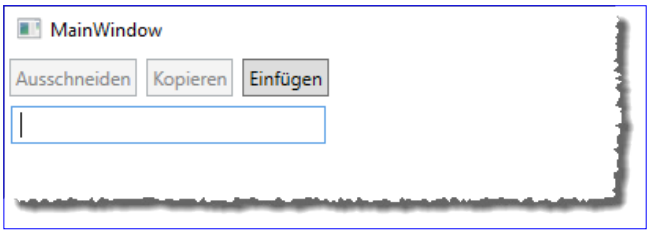

Bild 3: Aktivierte **Einfügen**-Schaltfläche

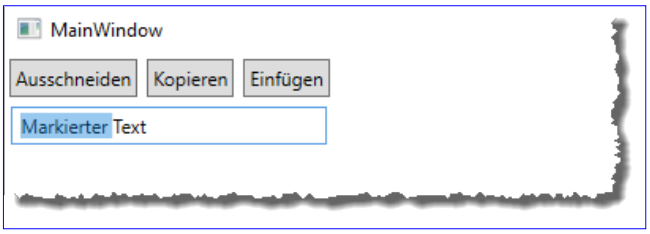

Bild 4: Durch Markierung aktivierte **Ausschneiden**- und **Einfügen**-Schaltflächen

Wenn das Textfeld leer ist, die Zwischenablage jedoch einen Wert enthält, der in das Textfeld eingefügt werden kann, wird nur die **Einfügen**-Schaltfläche aktiviert – und sie fügt den Inhalt der Zwischenablage auch per Mausklick ein (siehe Bild 3). Wenn sich beispielsweise ein Objekt wie eine Datei in der Zwischenablage befindet, wird die Einfügen-Schaltfläche nicht aktiviert, denn ein solches Objekt kann nicht in das Textfeld eingefügt werden. Gleiches gilt, wenn die Zwischenablage komplett leer ist.

Hat der Benutzer zusätzlich noch mindestens ein Zeichen im Textfeld **txtText** markiert, werden auch die beiden Schaltflächen zum Ausschneiden und Kopieren aktiviert (siehe Bild 4). Und auch das Ausschneiden und Kopieren mit den Schaltflächen funktioniert ohne Probleme.

Hier geschehen also einige Dinge automatisch – sowohl die Aktionen, die durch die Schaltflächen ausgeführt werden sollen, werden durch die Zuweisung des entsprechenden Wertes zum Attribut **Command** automatisch ausgeführt als auch die Aktivierung oder Deaktivierung abhängig vom Status des mit **CommandTarget** angegebenen Steuerelements.

Commands mit IntelliSense

Weiter oben haben wir einfach das Command als Wert der **Command**-Eigenschaft angegeben. Wenn Sie das auch gemacht haben, ist Ihnen vielleicht aufgefallen, dass es hier keine IntelliSense-Unterstützung gab, was für Visual Studio eher ungewöhnlich ist. Mit einer etwas anderen Schreibweise gelangen Sie allerdings schnell zur Anzeige der möglichen Befehle per IntelliSense. Diese sieht wie folgt aus – hier am Beispiel der Ausschneiden-Schaltfläche:

```
<Button x:Name="cmdAusschneiden" Command="{x:Static ApplicationCommands.Cut}"
    CommandTarget="{Binding ElementName=txtText}">Ausschneiden</Button>
```

Eingebaute Commands überschreiben

Wenn Sie schon eingebaute Commands nutzen können, um von verschiedenen Steuerelementen darauf zugreifen zu können, dann wäre es doch praktisch, wenn man diese eingebauten Commands auch überschreiben könnte. Damit meinen wir beispielsweise, dass Sie die Ausschneiden-Aktion **ApplicationCommands.Cut** durch eine eigene Methode ersetzen, die vor dem Ausschneiden noch nachfragt, ob der Inhalt wirklich ausgeschnitten werden soll.

Dazu können Sie mit dem Executed-Attribut eine statt der eigentlichen Funktion einzusetzende Methode angeben und diese dann im Code behind-Modul des WPF-Fensters implementieren. Dazu fügen wir in der XAML-Datei ein Element namens **TextBox.CommandBindings** mit dem Unterelement **CommandBinding** hinzu, das die drei Attribute **Command**, **CanExecute** und **Executed** enthält – und zwar für das Textfeld, für das wir das Command anpassen wollen:

```
<TextBox x:Name="txtText" Grid.Row="1" Width="200">
    <TextBox.CommandBindings>
        <CommandBinding Command="{x:Static ApplicationCommands.Cut}" CanExecute="CommandBinding_CanExecute"
            Executed="CommandBinding_Executed" />
    </TextBox.CommandBindings>
</TextBox>
```

Hier tragen wir ein, dass es sich um das **Command**-Element **ApplicationCommands.Cut** handelt und wie die auszuführenden Methoden für **CanExecute** und **Executed** heißen.

Den Rest der Definition für die Schaltfläche **cmdAusschneiden** behalten wir bei:

```
<Grid>
    <StackPanel Orientation="Horizontal">
            <Button x:Name="cmdAusschneiden" Command="{x:Static ApplicationCommands.Cut}"
                CommandTarget="{Binding ElementName=txtText}" Content="Ausschneiden" >
            </Button>
        ...
</Grid>
```

In der Code behind-Klasse fügen wir die beiden für die Attribute **CanExecute** und **Executed** angegebenen Methoden hinzu. Um das möglichst einfach zu gestalten, erledigen wir das wie folgt:

- Tippen Sie den Namen des Attributs ein, also beispielsweise **Executed**.
- Geben Sie das Gleichheitszeichen ein. Es erscheint ein Popup mit zwei Befehlen, von denen Sie den ersten namens **<Neuer Ereignishandler>** wählen.
- Das legt automatisch die Ereignismethode mit den passenden Parametern in der Code behind-Datei an.

Die Code behind-Klasse sieht nach dem Anlegen der beiden Ereignismethoden **CommandBinding_CanExecute** und **CommandBinding_Executed** wie folgt aus:

```
Class MainWindow
    Private Sub CommandBinding_CanExecute(sender As Object, e As CanExecuteRoutedEventArgs)
        e.CanExecute = True
    End Sub

    Private Sub CommandBinding_Executed(sender As Object, e As ExecutedRoutedEventArgs)
        MessageBox.Show("Überschriebenes Ausschneiden-Command")
    End Sub
```

Anwendungseinstellungen unter VB nutzen

Es gibt verschiedene Gründe, Konfigurationsdaten einer Anwendung zu speichern – um den zuletzt verwendeten Dateipfad zu speichern, die Position und Größe von Fenstern zu sichern oder auch um die Verbindungsdaten zu einer Datenbank, die sich gelegentlich ändern, zu hinterlegen. Es gibt dafür einen eigenen Bereich in einem Projekt, der in einer speziellen Datei gespeichert wird. Dieser ist auch per VB zugänglich. Wie Sie Konfigurationsdateien mit Visual Basic anlegen und diese wieder abrufen, zeigt der vorliegende Artikel.

Es gibt einige mögliche Orte zum Speichern von Konfigurationsdaten – eine Textdatei, eine XML-Datei, eine Datenbank oder auch die vom Projekt bereitgestellte Möglichkeit der Konfigurationsdateien. Diese wollen wir in diesem Artikel vorstellen, denn sie ist die einfachste Variante. Diese finden Sie, wenn Sie im Projektmappen-Explorer doppelt auf den Eintrag **My Project** klicken. Es erscheint dann der Dialog aus Bild 1, in dem Sie auf die Seite **Einstellungen** wechseln.

Hier finden Sie eine Liste, in die Sie beliebig viele Einträge schreiben können. Jeder Eintrag der Liste enthält vier Informationen:

- **Name**: Bezeichnung der Einstellung
- **Typ**: Datentyp der Einstellung. Abhängig von der Auswahl finden Sie anschließend verschiedene Möglichkeiten im Feld **Wert**, um den jeweiligen Wert einzufügen. Für den Wert **Boolean** können Sie **True** oder **False** aus einem Auswahlfeld wählen, für

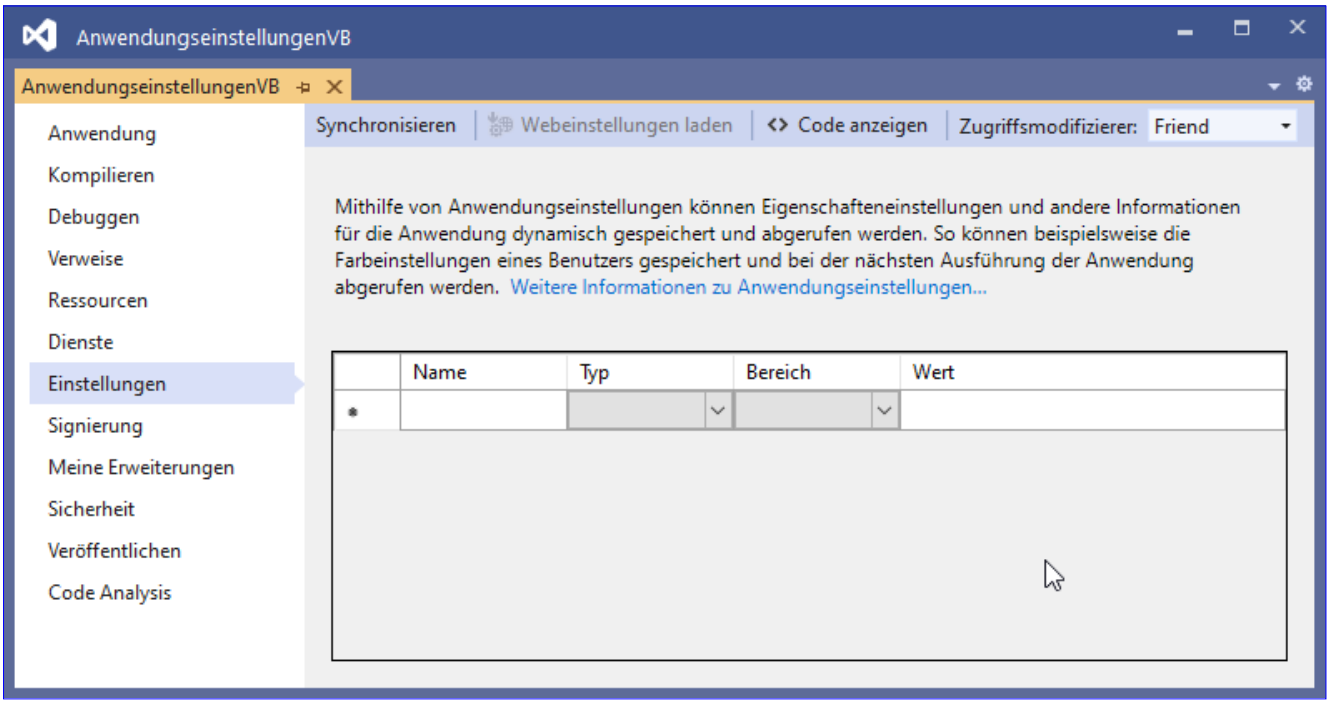

Bild 1: Der Bereich zum Verwalten von Anwendungseinstellungen

Date wird ein DatePicker angezeigt, bei Farben ein Farbdialog und so weiter (siehe Bild 2).

- **Bereich**: Gültigkeitsbereich mit den möglichen Werten **Anwendung** oder **Benutzer**.

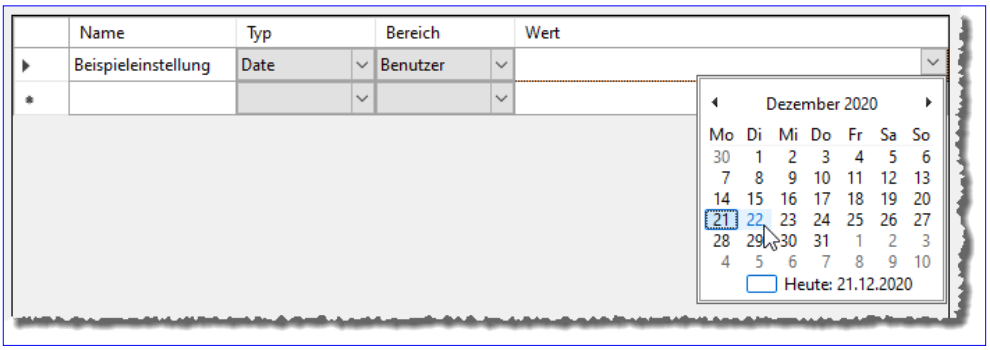

Bild 2: Verschiedene Eingabemöglichkeiten je nach Datentyp

- **Wert**: Wert der Einstellung

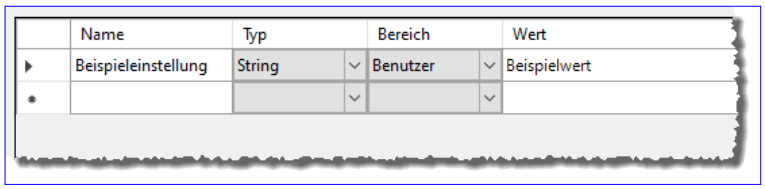

Bild 3: Beispiel für Anwendungseinstellungen

Nun wollen wir diesen Bereich nur initial nutzen, um die beim ersten Start einer Anwendung benötigten Voreinstellungen zu hinterlegen. Wir legen als Beispiel einfach eine Einstellung namens **Beispieleinstellung** mit dem Datentyp **String** und dem Wert **Beispielwert** an. Diese Werte sind einfach festzulegen (siehe Bild 3).

Bereich: Anwendung oder Benutzer?

Einen genaueren Blick werfen wir auf die Spalte **Bereich**. Hier können Sie zwischen den folgenden Werten wählen:

- **Anwendung**: Diese Einstellungen betreffen die Anwendung im Allgemeinen. Sie können nicht durch den Benutzer geändert werden.

- **Benutzer**: Diese Einstellungen sind benutzerspezifisch – zum Beispiel zum Speichern von Verzeichnissen oder Größe und Position von Fenstern. Diese Werte können dementsprechend vom Benutzer geändert werden.

Da wir zeigen wollen, wie wir die Einstellungen lesen und schreiben, legen wir für unsere erste Beispieleinstellung den Bereich **Benutzer** fest.

Speicherort der Einstellungen zur Design-Zeit

Bevor wir den Zugriff auf die Einstellungen per Code zeigen, schauen wir uns noch den Speicherort der Anwendungseinstellungen an. Dieser ist die Datei **app.config** im Projektverzeichnis. Die Datei sieht wie in Bild 5 aus und speichert die Anwendungseinstellungen für den Benutzer im Unterelement **userSettings**.

Dort finden Sie im Unterelement **<AnwendungseinstellungenVB.MySettings>** (wobei **AnwendungseinstellungenVB** der Projektname ist) für jede Einstellung einen Eintrag namens **setting**. **setting** speichert die im Anwendungseinstellungen-Dialog abgefragten ersten beiden Spalten in den Attributen **name** und **serializeAs** (Typ). Der Bereich ist ja bereits durch **userSettings** gegeben. Der Wert wird im Unterelement **value** gespeichert. Die Anwendungseinstellungen auf Anwendungsebene werden im Unterelement **applicationSettings** gespeichert.

Das Menu-Steuerelement

Wie Sie einer Anwendung beziehungsweise einem Fenster einer WPF-Anwendung ein Ribbon hinzufügen können, haben wir an anderer Stelle bereits erläutert. Wer nicht auf die eher platzraubenden Ribbons steht, kann sich auch der klassischen Menüs bedienen. Diese lassen sich unter WPF genauso leicht definieren wie Ribbons. Dieser Artikel zeigt, wie Sie einem Fenster ein Menü hinzufügen und welche Steuerelemente sich dort unterbringen lassen. Damit die Optik nicht leidet, bauen wir auch hier passende Icons ein.

Grundstruktur des Menüs

Normalerweise enthält ein **Window**-Element immer ein **Grid**-Element zur Strukturierung der enthaltenen Steuerelemente. Wenn Sie ein Menü hinzufügen wollen, strukturieren wir den XAML-Code ein wenig um. Als oberstes Element unterhalb des **Window**-Elements verwenden wir dann nämlich ein **DockPanel**-Element. Wenn wir diesem ein **Menu**-Element und ein **Grid**-Element hinzufügen, werden beide untereinander angeordnet und Sie können im **Grid**-Element wie üblich die weiteren Steuerelemente anordnen. Im folgenden Beispiel fügen wir erst einmal ein ganz einfaches Menü hinzu, das Sie im Entwurf in Bild 1 sehen. Das **DockPanel**-Element enthält zuerst ein **Menu**-Element, das Hauptelement eines Menüs. Diesem fügen Sie beliebige **MenuItem**-Elemente zur obersten Ebene hinzu, die alle oben in der Menüleiste erscheinen. Fügen Sie den **MenuItem**-Elementen der obersten Ebene weitere **MenuItem**-Elemente hinzu, werden diese darunter angeordnet. Zur Laufzeit erscheinen diese nur, wenn der Benutzer den **MenuItem**-Eintrag der obersten Ebene öffnet. Die Beschriftung der Menüeinträge stellen wir mit dem **Header**-Attribut ein:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="Datei">
                <MenuItem Header="Beenden"></MenuItem>
            </MenuItem>
        </Menu>
        <Grid>

        </Grid>
    </DockPanel>
</Window>
```

Starten wir die Anwendung, erscheint das Menü wie in Bild 2.

Bild 1: Einfaches Menü

Menüeinträge per Alt-Tastenkombination steuern

Von Windows-Anwendungen, die noch mit dem herkömmlichen Menüsystem arbeiten, kennen Sie sicher die Möglichkeit, ein Menü aufzuklappen oder einen Menüpunkt auszuführen, indem Sie eine bestimmte Tastenkombination betätigen. Diese besteht

aus der Taste Alt und dem Buchstaben der Bezeichnung des gewünschten Menüeintrags, der unterstrichen dargestellt wird.

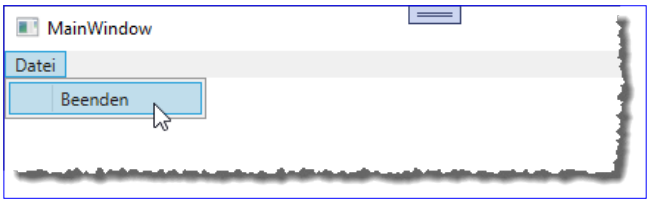

Bild 2: Einfaches Menü zur Laufzeit

Solche Menüeinträge mit unterstrichenen Buchstaben lassen sich auch für die WPF-Menüs realisieren. Dazu fügen Sie nur vor dem Buchstaben, der für die Tastenkombination genutzt werden soll, einen Unterstrich ein wie in den folgenden Beispieleinträgen:

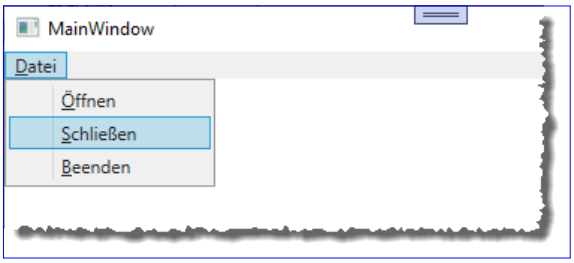

Bild 3: Menü mit Tastenkombinationen

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="_Datei">
        <MenuItem Header="_Öffnen"></MenuItem>
        <MenuItem Header="_Schließen"></MenuItem>
        <MenuItem Header="_Beenden"></MenuItem>
    </MenuItem>
</Menu>
```

Die Hauptmenüpunkte sehen danach zunächst wie üblich aus. Wenn Sie jedoch die **Alt**-Taste herunterdrücken und gedrückt halten, werden die Buchstaben mit dem führenden Unterstrich unterstrichen dargestellt. Betätigen Sie also in diesem Fall bei gedrückter **Alt**-Taste noch den Buchstaben **D**, wird das Untermenü wie in Bild 3 geöffnet. Dann können Sie mit dem entsprechenden weiteren Buchstaben einen der Untermenüpunkte auswählen. Drücken Sie im initialen Zustand nur die **Alt**-Taste und lassen diese wieder los, wird der erste Menüpunkt blau markiert und die mit dem Unterstrich versehenen Buchstaben in den Bezeichnungen werden ebenfalls nun unterstrichen dargestellt. Von nun an können Sie mit der Taste **Nach unten** die Untermenüpunkte anzeigen und dann mit den Tasten **Nach unten** und **Nach oben** einen der Menüpunkte auswählen und dann mit der Eingabetaste aufrufen. Wenn das Hauptmenü bereits mehrere Einträge hätte, könnten Sie nach dem Betätigen der **Alt**-Taste mit den Tasten **Nach links** und **Nach rechts** zwischen den Menüpunkten des Hauptmenüs navigieren.

Icons zu den Menüeinträgen hinzufügen

Ein Menü ist nur halb so hübsch, wenn es keine Icons aufweist. Also fügen wir dem Menü ein paar Icons hinzu. Diese müssen Sie zunächst zum Projekt hinzufügen. Dazu legen wir im Projektmappen-Explorer ein neues Verzeichnis an. Diesem fügen wir dann die benötigten Dateien hinzu.

Die Icons lassen sich nicht so eben per Attribut zum Menüeintrag hinzufügen. Dazu legen wir zunächst ein Unterelement namens **MenuItem.Icon** an. Dieses enthält ein weiteres Unterelement namens **Image**, für dessen **Source**-Attribut wir schließlich den Verweis auf das zu verwendende Bild angeben. Dazu geben Sie den Pfad zu der Datei an, wie er im Projektmappen-Explorer vorliegt, also zum Beispiel **/Icons/add.png**:

```
<MenuItem Header="Einträge mit Icons">
    <MenuItem Header="_Hinzufügen">
        <MenuItem.Icon>
```

```
            <Image Source="/Icons/add.png"></Image>
        </MenuItem.Icon>
    </MenuItem>
    <MenuItem Header="_Apfel">
        <MenuItem.Icon>
            <Image Source="/Icons/apple.png"></Image>
        </MenuItem.Icon>
    </MenuItem>
    <MenuItem Header="_Ballon">
        <MenuItem.Icon>
            <Image Source="/Icons/balloon.png"></Image>
        </MenuItem.Icon>
    </MenuItem>
</MenuItem>
```

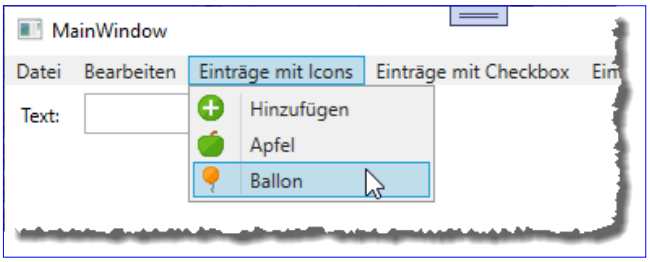

Bild 4: Menüs mit Icons

Die Icons werden dann wie in Bild 4 angezeigt.

Weitere Steuerelementtypen

Wie beim Ribbon gibt es noch weitere verwendbare Steuerelementtypen, aber bei weitem nicht so viele. Das meistverwendete neben dem herkömmlichen Menüeintrag dürfte der Trennstrich sein, der als Steuerelement die Bezeichnung **Separator** trägt. Den Trennstrich fügen Sie einfach zwischen zwei Menüeinträgen ein:

```
<MenuItem Header="_Schließen"></MenuItem>
<Separator />
<MenuItem Header="_Beenden"></MenuItem>
```

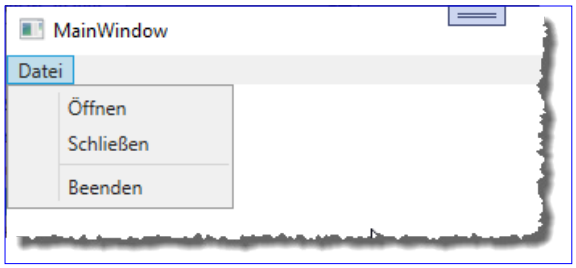

Bild 5: Menü mit Trennstrich

Er erscheint dann als waagerechter Strich zwischen den beiden anderen Einträgen (siehe Bild 5).

Menüeinträge mit Checkbox

Über Menüeinträge können Sie im kleinen Maßstab auch Optionen festlegen. Dazu bieten die **MenuItem**-Elemente die Eigenschaft **IsCheckable**. Hat diese Eigenschaft den Wert **True**, können Sie per Mausklick wie in Bild 6 einen Haken vor dem jeweiligen Eintrag einfügen. Um einen Eintrag direkt beim Anzeigen als angehakt darzustellen, weisen Sie außerdem dem Attribut **IsChecked** den Wert **True** zu. Hier ein Beispiel:

```
<MenuItem Header="Einträge mit Checkbox">
    <MenuItem Header="Erster Eintrag" IsCheckable="True" IsChecked="True"></MenuItem>
    <MenuItem Header="Zweiter Eintrag" IsCheckable="True"></MenuItem>
    <MenuItem Header="Dritter Eintrag" IsCheckable="True"></MenuItem>
</MenuItem>
```

Symbolleisten mit dem ToolBar-Steuerelement

Bei Anwendungen, die eine Menüleiste statt des Ribbons verwenden, finden Sie meist auch noch Symbolleisten vor. Davon gibt es im Gegensatz zur Menüleiste manchmal nicht nur eine, sondern mehrere. Symbolleisten zeichnen sich außerdem dadurch aus, dass Sie diese verschieben können und dass sie keine Untermenüs enthalten. Außerdem finden Sie die Befehle meist in Form einfacher Schaltflächen mit entsprechenden Symbolen vor, damit diese wenig Platz wegnehmen. Auch für eine WPF-Anwendung können Sie Symbolleisten hinzufügen. Wie das geht, zeigt der vorliegende Artikel.

Im Artikel **Das Menu-Steuerelement** (**www.datenbankentwickler.net/244**) haben wir bereits gezeigt, wie Sie eine Menüleiste zum Fenster einer Anwendung hinzufügen. Es gibt immer nur eine Menüleiste je Fenster, dem Sie ein oder mehrere Elemente in der obersten Ebene hinzufügen können. Diese stellen normalerweise per Mausklick eine aufklappbare Liste von Befehlen oder weiteren Untermenüs zur Verfügung. Bei den Symbolleisten ist es etwas anders: Sie können auch mehr als eine Symbolleiste hinzufügen, die nach dem Wunsch des Benutzers angeordnet werden können. In den Beispielen zu diesem Artikel verwenden wir wieder ein übergeordnetes **DockPanel**-Element, dem wir eine Menüleiste und eine Symbolleiste hinzufügen.

Aufbau einer Symbolleiste

Bei der Menüleiste ist der Aufbau eindeutig: Es gibt ein **Menu**-Element, das im **DockPanel**-Element oben angedockt wird und das die benötigten Menüelemente enthält. Bei der Symbolleiste verwenden wir noch ein zusätzliches Element, nämlich das **ToolBarTray** in der obersten Ebene. Dieses ordnen Sie wie das **Menu**-Element dem **DockPanel**-Element unter (siehe Beispielprojekt, **MainWindow.xaml**, **Beispiel 1**):

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="350" Width="600">
    <DockPanel>
        <Menu DockPanel.Dock="Top">...</Menu>
        <ToolBarTray DockPanel.Dock="Top">
```

Das **ToolBarTray**-Element kann dann ein oder mehrerere **ToolBar**-Elemente enthalten, die wiederum **Button**-Elemente mit dem Befehlen bereithalten:

```
            <ToolBar>
                <Button Command="Cut" Content="Ausschneiden"></Button>
                <Button Command="Copy" Content="Kopieren"></Button>
                <Button Command="Paste" Content="Einfügen"></Button>
```

Zum Abschluss werden die geöffneten Elemente **ToolBar** und **ToolBarTray** wieder geschlossen und es folgt beispielsweise ein Grid-Element, das die übrigen Steuerelemente des Fensters enthält:

```
            </ToolBar>
```

```
        </ToolBarTray>
        <Grid>...</Grid>
    </DockPanel>
</Window>
```

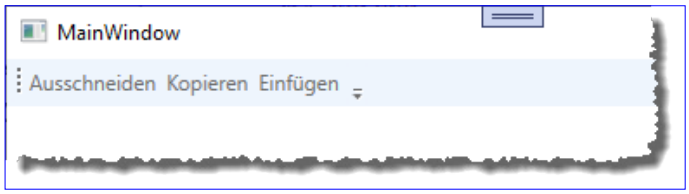

Bild 1: Fenster mit Menü und Symbolleiste

In Bild 1 sehen Sie da resultierende Menü und darunter die Symbolleiste. In diesem Fall haben wir einige eingebaute Befehle über das **Command**-Attribut verfügbar gemacht – **Cut**, **Copy** und **Paste**. Leider erkennt die Anwendung nicht, dass diese hier in einer Symbolleiste abgebildet werden sollen und zeigt keine Icons, sondern nur den Text an. Das können wir ändern!

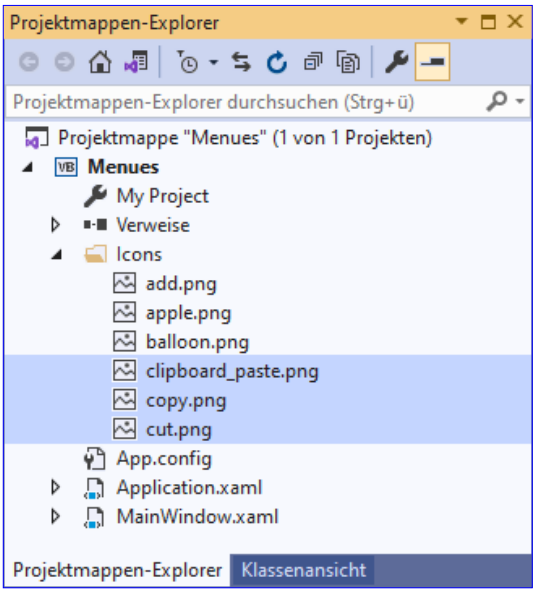

Bild 2: Hinzufügen von Icons zum Projekt

Symbolleiste mit Icons

Wenn wir statt der Beschriftungen für die eingebauten Befehle **Cut**, **Copy** und **Paste** nur Icons anzeigen wollen, müssen wir diese zunächst zum Projekt hinzufügen. Dazu legen wir im Projektmappen-Explorer ein neues Verzeichnis namens **Icons** an und fügen diesem die benötigten Icons hinzu (siehe Bild 2).

Danach bearbeiten wir die Steuerelemente der Symbolleiste in der XAML-Definition so, dass sie statt Text die Icons anzeigen. Dazu entfernen wir **Content**-Attribute und fügen jeweils ein **Image**-Element hinzu. Dieses erhält für das Attribut **Source** den Pfad zu der Bilddatei im Projektverzeichnis (siehe Beispielprojekt, **MainWindow.xaml**, **Beispiel 2**):

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        <Button Command="Cut">
            <Image Source="Icons/cut.png"></Image>
        </Button>
        <Button Command="Copy">
            <Image Source="Icons/copy.png"></Image>
        </Button>
        <Button Command="Paste">
            <Image Source="Icons/clipboard_paste.png"></Image>
        </Button>
    </ToolBar>
</ToolBarTray>
```

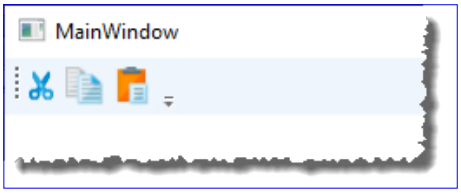

Bild 3: Symbolleiste mit Icons

Das Ergebnis sehen Sie in Bild 3. Interessanterweise werden die Schaltflächen mit den Icons im Gegensatz zu den reinen Text-Schaltflächen nicht in Abhän-

gigkeit vom Kontext als aktiviert oder deaktiviert gekennzeichnet. So sollte zum Beispiel die **Ausschneiden**-Schaltfläche nur markiert sein, wenn im Textfeld darunter ein Text markiert ist. Das wollen wir später nachholen – erst einmal schauen wir uns an, wie wir eine weitere Symbolleiste hinzufügen und wie wir Icons und Texte auf einer Schaltfläche kombinieren können.

Eine zweite Symbolleiste hinzufügen

Zum Hinzufügen einer zweiten Symbolleiste legen Sie einfach ein weiteres **ToolBar**-Element im **ToolBarTray**-Element an. Je nachdem, ob die zweite Symbolleiste vor oder hinter der ersten erscheinen soll, fügen Sie das neue **ToolBar**-Element vor oder hinter einem bereits bestehenden **ToolBar**-Element ein (siehe Beispielprojekt, **MainWindow.xaml**, **Beispiel 3**):

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        ... Steuerelemente
    </ToolBar>
    <ToolBar>
        ... Steuerelemente
    </ToolBar>
</ToolBarTray>
```

In diesem Fall werden die beiden Symbolleisten direkt nebeneinander angezeigt (siehe Bild 4). Es gibt jedoch die Möglichkeit, diese auch untereinander auszugeben.

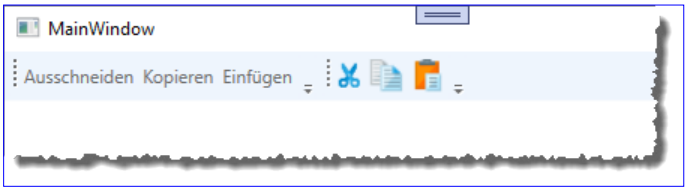

Bild 4: Zwei Symbolleisten nebeneinander

Der einfachste Weg ist, dass der Benutzer die Symbolleisten einfach per Drag and Drop so positioniert, wie er sich diese wünscht (siehe Bild 5).

Beim nächsten Start der Anwendung werden die Symbolleisten allerdings wieder so wie im XAML-Code definiert ausgegeben. Wie also können wir die Symbolleisten direkt beim Starten untereinander anzeigen?

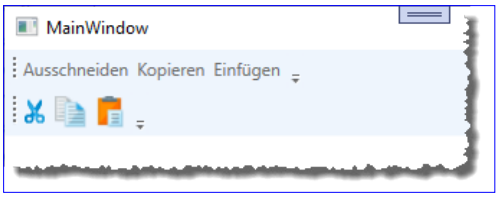

Bild 5: Zwei Symbolleisten untereinander

Symbolleistenposition relativ zu anderen Symbolleisten

Wenn Sie mehr als eine Symbolleiste verwenden, also mehr als ein **ToolBar**-Element innerhalb des **ToolBarTray**-Elements einfügen, werden diese wie oben beschrieben einfach in der gleichen Reihenfolge nebeneinander angeordnet. Wenn Sie die Reihenfolge ändern möchten, ohne die Reihenfolge im Code zu ändern, können Sie das mit der Eigenschaft **BandIndex** erledigen. **BandIndex** ist ein **0**-basierter Index (siehe Beispielprojekt, **MainWindow.xaml**, **Beispiel 4**). Wenn Sie also für das erste **ToolBar**-Element innerhalb des **ToolBarTray**-Elements den Wert **0** angeben, dann verbleibt es an der Position, die es vorher innehatte. Im folgenden Beispiel erhalten wir die Anordnung aus Bild 6:

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        <Button>Button in Toolbar 1</Button>
```

```
    </ToolBar>
    <ToolBar>
        <Button>Button in Toolbar 2</Button>
    </ToolBar>
    <ToolBar>
        <Button>Button in Toolbar 3</Button>
    </ToolBar>
</ToolBarTray>
```

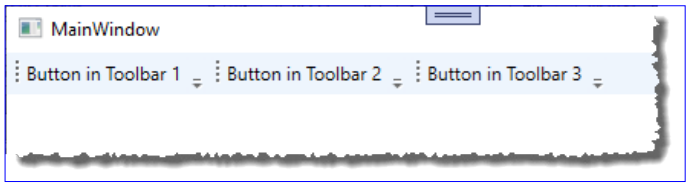

Bild 6: Standardreihenfolge

Stellen wir nun für das zuerst aufgeführte Element den Wert des Attributs **BandIndex** auf **2** ein, erhalten wir die Anordnung aus Bild 7.

```
<ToolBar BandIndex="2">
    <Button>Button in Toolbar 1</Button>
</ToolBar>
```

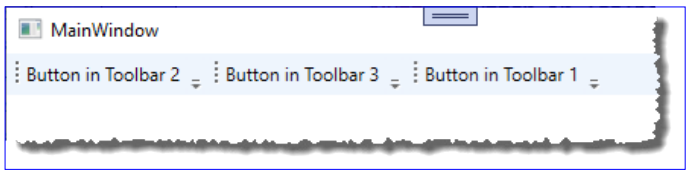

Bild 7: Per **BandIndex** angepasste Reihenfolge

Damit können Sie nun bereits die Reihenfolge innerhalb der gleichen Zeile anpassen. Nun wollen wir noch eines der **ToolBar**-Elemente in die zweite Zeile verschieben. Das erledigen wir mit dem Attribut **Band**. Behalten wir die übrigen Einstellungen bei und legen für das Attribut **Band** des ersten **ToolBar**-Elements den Wert **1** fest, landet diese Symbolleiste in der zweiten Zeile (siehe Bild 8):

```
<ToolBar Band="1">
    <Button>Button in Toolbar 1</Button>
</ToolBar>
```

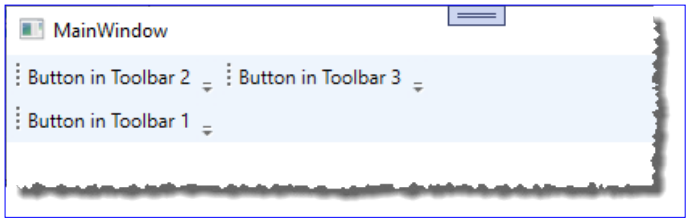

Bild 8: Verschieben einer Symbolleiste in eine andere Zeile mit dem **Band**-Attribut

Übrigens sind die Werte für **Band** und **BandIndex** relativ zu betrachten. Der Standardwert lautet **0**. Sie müssen aber nicht die Werte **0**, **1** und **2** verwenden, wenn die Symbolleisten in verschiedenen Zeilen landen sollen, sondern es funktioniert auch mit **2**, **4** und **6**.

Schaltflächen mit Icon und Text

Wollen Sie einer Symbolleiste nicht nur Schaltflächen mit Text oder Icon, sondern mit beidem hinzufügen, können Sie das mit den üblichen Techniken von WPF erledigen. Dabei fügen wir einfach ein **StackPanel**-Element als Inhalt des **Button**-Elements ein, dem wir dann mehrere weitere Steuerelemente hinzufügen können. Für unsere drei Schaltflächen sieht dies dann wie folgt aus (siehe Bild 9):

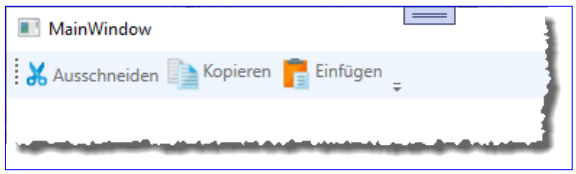

Bild 9: Symbolleiste mit Icons und Beschriftung

```
<ToolBar >
    <Button Command="Cut">
```

Symbolleiste speichern und wiederherstellen

Mit dem ToolBar-Element können Sie einem Fenster auf einfache Weise eine Symbolleiste zuweisen. Je nach Umfang der Anwendung können so einige Symbolleisten zusammenkommen. Diese kann der Benutzer normalerweise selbst organisieren. Allerdings werden die Symbolleisten bei jedem Start wieder so hergestellt, wie es in der XAML-Definition vorgesehen ist. Ändert der Benutzer also die Position von Symbolleisten oder blendet er diese ein oder aus, halten diese Anpassungen immer nur bis zum nächsten Start der Anwendung. Außer natürlich, wir sehen eine Funktion vor, die den aktuellen Stand speichert und beim nächsten Start wiederherstellt.

Zu Beispielzwecken haben wir zwei **ToolBar**-Elemente zu einem **ToolBarTray** hinzugefügt. Diese haben wir wie folgt programmiert (Auszüge aus dem Code, den Sie im Modul **MainWindow.xaml** der Beispielanwendung finden):

```
<DockPanel>
    <ToolBarTray x:Name="tbt" DockPanel.Dock="Top">
        <ToolBar>
            <Button Command="Cut">
                <StackPanel Orientation="Horizontal">
                    <Image Source="Icons/cut.png"></Image>
                    <TextBlock Margin="3,0,0,0">Ausschneiden</TextBlock>
                </StackPanel>
            </Button>
            ...
        </ToolBar>
        <ToolBar>
            <Button>
                <Image Source="Icons/add.png"></Image>
            </Button>
            ...
        </ToolBar>
    </ToolBarTray>
</DockPanel>
```

Die Symbolleiste sieht nun wie in Bild 1 aus.

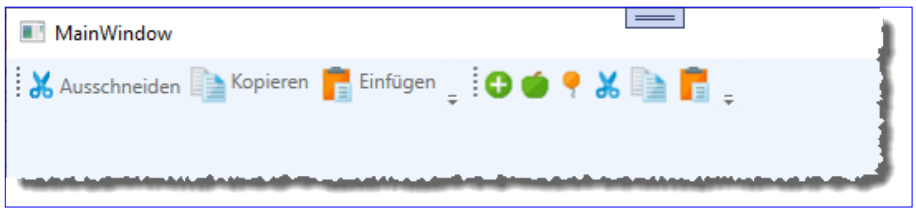

Bild 1: Beispielsymbolleisten

Der Benutzer hat nun die Möglichkeit, die Anordnung der Symbolleiste zu ändern. Zum Beispiel kann er die

Reihenfolge der beiden Symbolleisten vertauschen (siehe Bild 2).

Oder er verschiebt eine der beiden Symbolleisten in die zweite Zeile (siehe Bild 3).

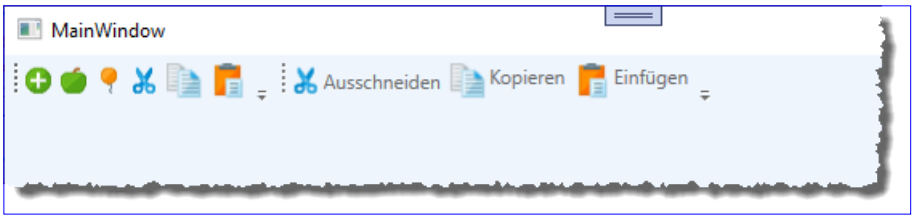

Bild 2: Vertauschte Beispielsymbolleisten

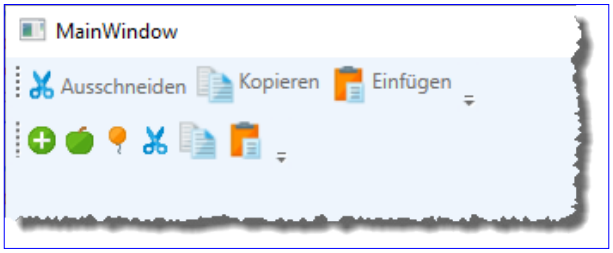

Bild 3: Beispielsymbolleisten in zwei Zeilen

Erste Idee: XAML-Definition des ToolBarTray-Elements speichern

Die erste Idee war, dass Änderungen der Position sich auch im Zustand der XAML-Definition der ToolBar-Elemente wiederspiegeln. Für diesen Fall wollten wir einfach die XAML-Definition des **ToolBarTray**-Elements speichern und diese beim nächsten Öffnen wiederherstellen. Die folgende Methode sollte durch ein Ereignis ausgelöst werden, das wir für das Element **Window** definieren:

```
<Window x:Class="MainWindow" ...  Title="MainWindow" Height="450" Width="800" Closing="Window_Closing">
```

Die dadurch ausgelöste Methode sieht wie folgt aus: Sie definiert ein **FileStream**-Objekt, das eine neue Datei namens **ToolBarTray.txt** erstellt und dann das ToolBarTray namens **tbt** mit der Variablen **objToolBarTray** referenziert. Danach nutzen wir die **Save**-Methode der **XamlWriter**-Klasse. Diese stellen wir über den folgenden Namespace bereit:

```
Imports System.Windows.Markup
```

Die **Save**-Methode erwartet einen Verweis auf das Root-Element des zu speichernden XAML-Inhalts sowie ein Ziel als Parameter. Nach dem Speichern schließen wir das **FileStream**-Objekt wieder:

```
Private Sub Window_Closing(sender As Object, e As ComponentModel.CancelEventArgs)
    Dim objToolBarTray As ToolBarTray
    Dim objFileStream As Stream
    objFileStream = New FileStream("ToolBarTray.txt", FileMode.Create)
    objToolBarTray = tbt
    XamlWriter.Save(objToolBarTray, objFileStream)
    objFileStream.Close()
End Sub
```

Nach dem Starten und Beenden der Anwendung finden wir im Anwendungsverzeichnis (hier **bin/debug**) eine Textdatei, die auch die Definition des gewünschten XAML-Bereichs enthält. Da wir zwischen Starten und Beenden noch keine Änderungen an der Konfiguration der **ToolBar**-Elemente vorgenommen haben, sieht der Code genau wie erwartet aus:

```
<ToolBarTray Name="tbt" DockPanel.Dock="Top" ...>
```

XML schreiben mit dem Document Object Model

Im Artikel »XML-Dokumente erstellen mit XmlWriter« haben wir beschrieben, wie Sie XML-Dokumente mit der XmlWriter-Klasse schreiben. Dort können Elemente nur sequenziell hinzugefügt werden. Die Methoden des Object Document Models erlauben eine wesentlich flexiblere Vorgehensweise. Dieser Artikel zeigt, wie Sie mit dem Document Object Model neue XML-Dokumente erstellen und Elemente der verschiedenen Typen hinzufügen und mit Daten füllen.

Beispieldaten

Die Beispiele aus dem ersten Teil des Artikels finden Sie im Beispielprojekt **XmlBeispiele** aus dem Download.

Erstellen eines neuen XML-Dokuments

Im Artikel **XML lesen mit dem Document Object Model** (**www.datenbankentwickler.net/237**) haben Sie neue XML-Dokument-Objekte erstellt und diese mit **Load** oder **LoadXml** befüllt. Nun wollen wir ein neues, leeres XML-Dokument erstellen und dieses speichern. Die Mindestanforderung ist dabei, dass das Dokument ein Stammelement enthält. Um das mit VB zu realisieren, benötigen wir wie in den vorherigen Artikeln einen Verweis auf den Namespace **System.Xml**:

```
Imports System.Xml
```

Danach können wir mit der folgenden Methode ein neues **XmlDocument** erstellen. Bevor wir dieses speichern können, fügen wir das Stammelement in zwei Schritten hinzu:

- Erstellen eines neuen Elements des Typs **XmlElement** mit der **CreateElement**-Methode des **XmlDocument**-Objekts und
- Hinzufügen dieses Elements mit der **AppendChild**-Methode an das XML-Dokument aus **objXml**.

Danach können wir das Dokument mit **Save** und dem gewünschten Dateipfad speichern:

```
Private Sub btnDokumentErstellenDOM_Click(sender As Object, e As RoutedEventArgs)
    Dim objXml As XmlDocument
    Dim objElement As XmlElement
    Dim strDocument As String
    strDocument = "C:\...\BestellverwaltungDOM.xml"
    objXml = New XmlDocument
    objElement = objXml.CreateElement("Bestellverwaltung")
    objXml.AppendChild(objElement)
    objXml.Save(strDocument)
End Sub
```

Das Dokument enthält nun schlicht folgenden Text:

```
<Bestellverwaltung />
```

Prolog hinzufügen

Für manche Anwendungen ist es wichtig, dass die XML-Datei einen Prolog enthält, also folgende Zeile:

```
<?xml version="1.0" encoding="utf-8"?>
```

Diese nennt man auch Processing Instruction. Und genau so ein Objekt gibt es auch im Namespace **System.Xml**. Um dieses zu erstellen, deklarieren wir eine Variable des Typs **XmlProcessingInstruction** sowie eine für den Inhalt dieses Objekts:

```
Dim objPI As XmlProcessingInstruction
Dim strPI As String
```

Dann schreiben wir den Text in **strPI** und erstellen mit der Methode **CreateProcessingInstruction** das **XmlProcessingInstruction**-Element. Der erste Parameter ist der Name, der hinter **<?** erscheinen soll, also **xml**, der zweite die Angabe der gewünschten Name-Wert-Paare. Danach hängen wir das Element wie herkömmliche Elemente mit der **AppendChild**-Methode an das Dokument an und speichern es anschließend:

```
strPI = "version=""1.0"" encoding=""utf-8"""
objPI = objXml.CreateProcessingInstruction("xml", strPI)
objXml.AppendChild(objPI)
...
objXml.Save(strDocument)
```

Das Ergebnis lautet, kombiniert mit den Anweisungen des ersten Beispiele, wie folgt:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung />
```

Da wir eine solche Processing Instruction allen XML-Dokumenten voranstellen, gliedern wir die notwendigen Anweisungen in eine eigene Methode aus, die wie folgt aussieht:

```
Public Sub AddProcessingInstruction(ByVal objXml As XmlDocument)
    Dim objPI As XmlProcessingInstruction
    Dim strPI As String
    strPI = "version=""1.0"" encoding=""utf-8"""
    objPI = objXml.CreateProcessingInstruction("xml", strPI)
    objXml.AppendChild(objPI)
End Sub
```

Den Aufruf können wir dann so in die eigentliche Methode zum Erstellen des XML-Dokuments einstreuen:

```
AddProcessingInstruction(objXml)
```

Unterelemente hinzufügen

Unser XML-Dokument besteht nun gerade mal aus der Processing Instruction und dem Stammelement **Bestellverwaltung**. Weitere Elemente müssen wir zwingend dem Stammelement unterordnen. Dieses haben wir aber bereits mit **objElement** referenziert, sodass wir eigentlich direkt loslegen können. Allerdings machen wir uns direkt an dieser Stelle Gedanken über die Benennung der **XmlElement**-Variablen, denn wir werden ja vielleicht noch weitere verschachtelte Elemente wie Kunden, Bestellungen und Bestellpositionen einfügen wollen und wir benötigen für alle zumindest eine Objektvariable je Typ. Also ändern wir den Namen der Objektvariablen für das Stammelement in **objBestellverwaltung**. Die folgenden Elemente wollen wir dann **objKunden**, **objKunde**, **objBestellungen**, **objBestellung** und so weiter nennen. Als Erstes wollen wir nur das Unterelement **Kunden** hinzufügen. Dazu legen wir die Variable **objKunden** mit dem Typ **XmlElement** an.

Dieser weisen wir dann das mit der **CreateElement** von **objXml** erstellte Objekt mit dem Namen **Kunden** zu. Allerdings hängen wir das **Kunden**-Element nicht an das XML-Dokument aus **objXml** an, sondern an das Element aus **objBestellverwaltung**:

```
Private Sub btnDokumentErstellenDOM_Unterelemente_Click(sender As Object, e As RoutedEventArgs)
    ...
    Dim objKunden As XmlElement
    ...
    objBestellverwaltung = objXml.CreateElement("Bestellverwaltung")
    objXml.AppendChild(objBestellverwaltung)
    objKunden = objXml.CreateElement("Kunden")
    objBestellverwaltung.AppendChild(objKunden)
    objXml.Save(strDocument)
End Sub
```

Das Ergebnis sieht dann genau wie gewünscht aus:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
  <Kunden />
</Bestellverwaltung>
```

Elemente mit Werten hinzufügen

Nun nähern wir uns den Elementen, die selbst Werte enthalten, also zum Beispiel das folgende:

```
  <Kunden>
    <Kunde>
      <Firma>André Minhorst Verlag</Firma>
    </Kunde>
  </Kunden>
```

Texte wie hier innerhalb des Elements **Firma** sind Elemente des Typs **XmlText**. Neben den übrigen neu hinzugekommenen Elementen deklarieren wir dieses entsprechend. Danach legen wir unterhalb des Elements aus **objKunden** weitere neue Elemente namens **objKunde** und **objFirma** an, beide des Typs **XmlElement**. Schließlich erstellen wir ein neues Element des Typs **XmlText** und weisen diesem mit dem Parameter **text** den gewünschten Inhalt zu. Dieses Objekt hängen wir an **objFirma** an:

```
Dim objKunde As XmlElement
Dim objFirma As XmlElement
Dim objText As XmlText
...
objKunde = objXml.CreateElement("Kunde")
objKunden.AppendChild(objKunde)
objFirma = objXml.CreateElement("Firma")
objKunde.AppendChild(objFirma)
objText = objXml.CreateTextNode("André Minhorst Verlag")
objFirma.AppendChild(objText)
objXml.Save(strDocument)
```

Damit hätten wir die Werkzeuge zum Zusammenstellen der Grundstruktur vollständig und können uns anderen Elementen zuwenden.

Attribute hinzufügen

Die Elemente **Kunde**, **Bestellung** und **Bestellposition** sollen jeweils um ein Attribut namens **ID** ergänzt werden:

```
<Kunde ID="123">
```

Dazu deklarieren wir eine Variable des Typs **XmlAttribute**:

```
Dim objID As XmlAttribute
```

Nach dem Erstellen des Elements, das wir mit dem Attribut ausstatten wollen, erzeugen wir das **XmlAttribute**-Objekt. Dann weisen wir diesem über das Attribut **Value** den gewünschten Wert zu und hängen das Attribut über die Methode **Append** der Auflistung **Attributes** an das **XmlElement**-Objekt an. Schließlich fügen wir das **XmlElement**-Objekt zum übergeordneten Objekt zu:

```
objKunde = objXml.CreateElement("Kunde")
objID = objXml.CreateAttribute("ID")
objID.Value = "123"
objKunde.Attributes.Append(objID)
objKunden.AppendChild(objKunde)
```

Sie können das Attribut allerdings auch erst nach dem Anhängen des **XmlElement**-Objekts an das übergeordnete Element hinzufügen.

CDATA-Block hinzufügen

Mit der Methode **CreateCDataSection** erstellen Sie ein Element mit einem **CDATA**-Block. Den Inhalt des **CDATA**-Blocks übergeben Sie als Parameter der Methode **CreateCDataSection**. Heraus kommt ein Objekt des Typs **XmlCDataSection**, das Sie wie gewohnt mit der **AppendChild**-Methode an das übergeordnete Element anfügen:

```
Dim objCData As XmlCDataSection
...
objCData = objXml.CreateCDataSection("Dies ist ein CDATA-Block mit Sonderzeichen wie <>&.")
objBestellverwaltung.AppendChild(objCData)
```

Das Ergebnis sieht so aus:

```
<Bestellverwaltung>
  <![CDATA[Dies ist ein CDATA-Block mit Sonderzeichen wie <>&.]]>
```

Kommentare hinzufügen

Kommentare legen Sie als Objekte des Typs **XmlComment** an. Die Methode **CreateComment** erstellt den Kommentar und nimmt dabei den Text des Kommentars als Parameter entgegen. Dann wird der Kommentar mit der **AppendChild**-Methode an das übergeordnete Objekt angehängt, in diesem Fall das Dokument selbst:

```
Dim objComment As XmlComment
...
objComment = objXml.CreateComment("Dies ist ein Kommentar.")
objXml.AppendChild(objComment)
```

Dadurch erscheint der Kommentar direkt unterhalb des Prologs:

```
<?xml version="1.0" encoding="utf-8"?>
<!--Dies ist ein Kommentar.-->
```

Element mit Standardnamespace hinzufügen

Wenn Sie Namespaces für die Elemente angeben wollen, verwenden Sie jeweils die Überladungen der **Create...**-Methoden. Für einen Standardnamespace geben Sie diesen als zweiten Parameter der Methode an. Soll das Element **Bestellverwaltung** beispielsweise den Standardnamespace **www.minhorst.com/ns/bestellverwaltung** erhalten, sieht der Aufruf wie folgt aus:

```
objBestellverwaltung = objXml.CreateElement("Bestellverwaltung", "www.minhorst.com/ns/bestellverwaltung")
objXml.AppendChild(objBestellverwaltung)
```

Hier ist das Ergebnis:

```
<Bestellverwaltung xmlns="www.minhorst.com/ns/bestellverwaltung" />
```