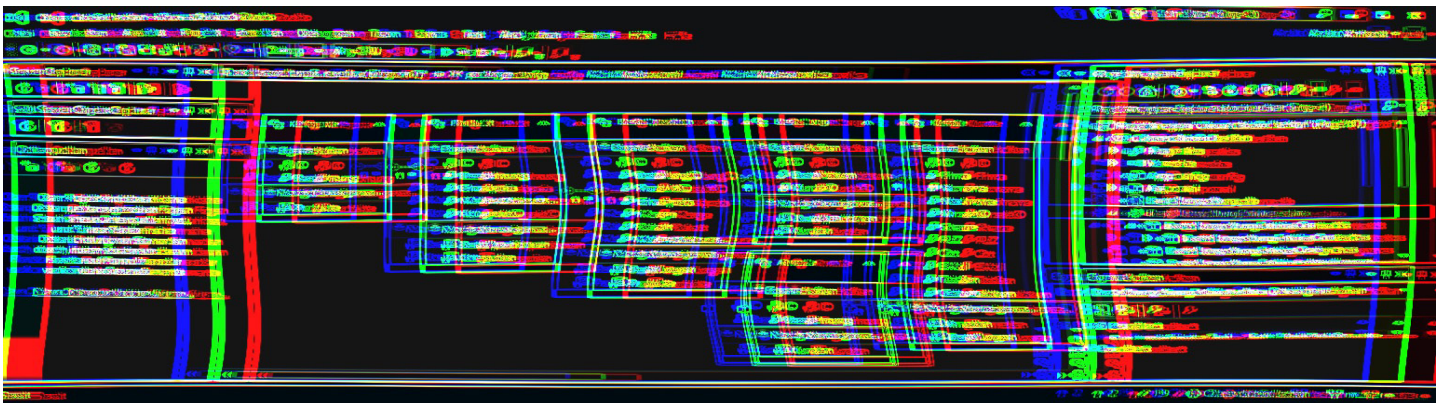
AUSGABE 06/2021

DATENBANK ENTWICKLER

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



TOP-THEMEN:

André Minhorst Verlag

TabControl: Das Registersteuerelement von WPF

Genau wie unter Microsoft Access finden Sie auch unter WPF ein Registersteuerelement – das sogenannte TabControl-Element. Sie können dieses Steuerelement vielseitig einsetzen, um beispielsweise Inhalte, die sonst nicht auf einer Bildschirmseite Platz finden, dennoch halbwegs übersichtlich darzustellen. Dieser Artikel stellt das Registersteuerelement von WPF vor und zeigt, wie Sie es in Ihren eigenen Anwendungen nutzen können.

Das leere TabControl-Element

Wenn wir einem Grid in einem XAML-Fenster ein **TabControl**-Element hinzufügen, bekommen wir auf jeden Fall wesentlich weniger geboten, als wenn wir ein solches beispielsweise einem Access-Formular hinzufügen:

```
<Grid>
    <TabControl Margin="5"></TabControl>
</Grid>
```

Wir müssen diesem schon per **Margin**-Attribut einen Abstand zum Fensterrahmen hinzufügen, damit wir erkennen können, dass es überhaupt vorhanden ist (siehe Bild 1).

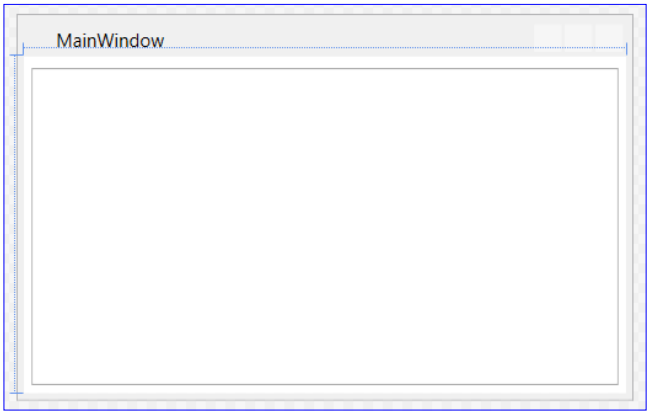


Bild 1: Ein leeres **TabControl**-Element

Ein TabItem-Element hinzufügen

Erst wenn wir dem **TabControl**-Element ein **TabItem**-Element unterordnen, erkennen wir, dass es sich bei dem angezeigten Rahmen um ein **TabControl**-Element handelt:

```
<TabControl Margin="5">
    <TabItem Header="Seite 1"></TabItem>
</TabControl>
```

Dem **TabItem**-Element weisen wir über das **Header**-Attribut den Text für den Registerreiter zu (siehe Bild 2).

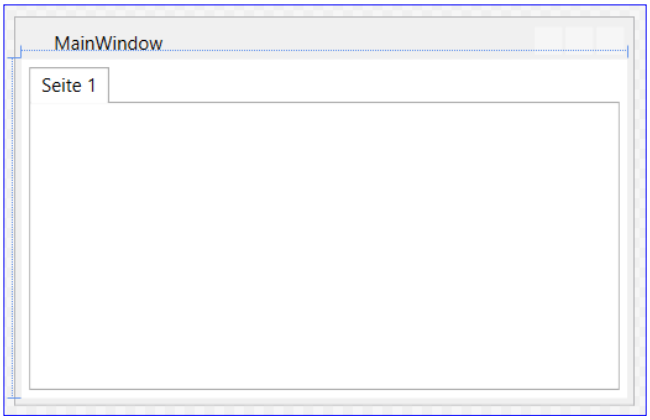


Bild 2: Ein **TabControl**-Element mit einem **TabItem**-Element

Auf die gleiche Weise legen Sie weitere Registerreiter an, die wie in Bild 3 erscheinen:

```
<TabControl Margin="5">
    <TabItem Header="Seite 1"></TabItem>
```

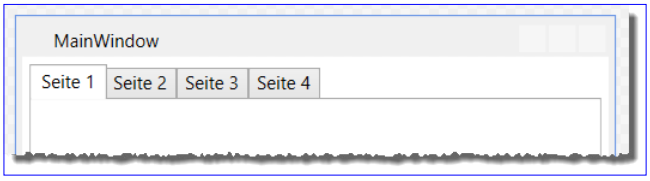


Bild 3: Mehrere **TabItem**-Elemente

```
    <TabItem Header="Seite 2"></TabItem>
    <TabItem Header="Seite 3"></TabItem>
    <TabItem Header="Seite 4"></TabItem>
</TabControl>
```

Um im Entwurf der XAML-Seite zu einer anderen Registerseite als der ersten zu wechseln, klicken Sie entweder auf den jeweiligen Registerreiter – also so, wie Sie es auch später in der laufenden Anwendung erledigen würden – oder Sie positionieren die Einfügemarke auf einem der **TabItem**-Elemente im XAML-Code.

Steuerelemente zu einem TabItem-Element hinzufügen

Allerdings können wir so nicht erkennen, ob tatsächlich der Inhalt der jeweiligen Registerseite angezeigt wird. Um dies zu erreichen, fügen wir jedem **TabItem**-Element zunächst ein **Label**-Steuerelement mit einer passenden Beschriftung hinzu:

```
<TabItem Header="Seite 1">
    <Label>Dies ist die erste Seite.</Label>
</TabItem>
<TabItem Header="Seite 2">
    <Label>Dies ist die zweite Seite.</Label>
</TabItem>
<TabItem Header="Seite 3">
    <Label>Dies ist die dritte Seite.</Label>
</TabItem>
<TabItem Header="Seite 4">
    <Label>Dies ist die vierte Seite.</Label>
</TabItem>
```

Bild 4: **TabItem**-Element mit Steuerelement

Ein Klick auf die zweite Registerseite zeigt wie erwartet das zu dieser Seite hinzugefügte **Label**-Steuerelement an (siehe Bild 4).

Nur ein Element je TabItem-Element

Dass es möglich ist, einfach Steuerelemente im **TabItem**-Element zu platzieren, liegt daran, dass das **TabItem**-Steuerelement vom **ContentControl**-Element abstammt. Das bedeutet gleichzeitig, dass Sie in jedem **TabItem**-Element aber auch nur ein einziges Steuerelement unterbringen dürfen. Wenn Sie wie in Bild 5 noch ein **Text-**

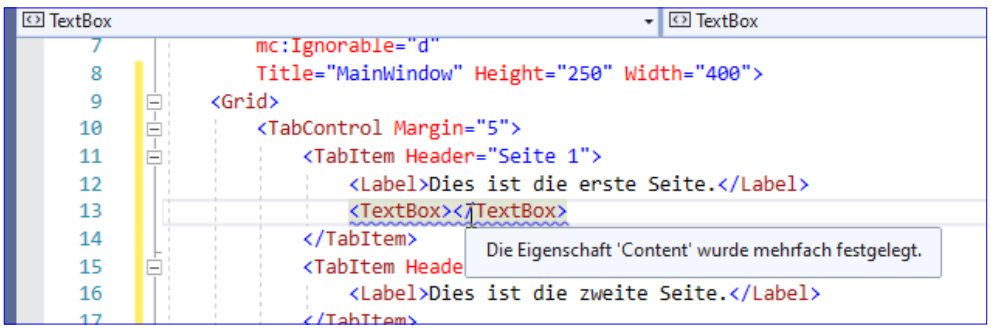


Bild 5: Das **TabItem**-Element kann nur ein Unterelement aufnehmen.

Box-Steuerelement hinzufügen möchten, erhalten Sie dementsprechend direkt eine Fehlermeldung. Das ist allerdings kein Problem, denn auch einem **Window**-Element, also dem Hauptelement eines XAML-Fensters, können Sie ja nur ein Element unterordnen. Aber wozu gibt es Elemente, die zum Organisieren mehrerer untergeordneter Elemente vorgesehen sind wie **Grid**, **StackPanel** und so weiter?

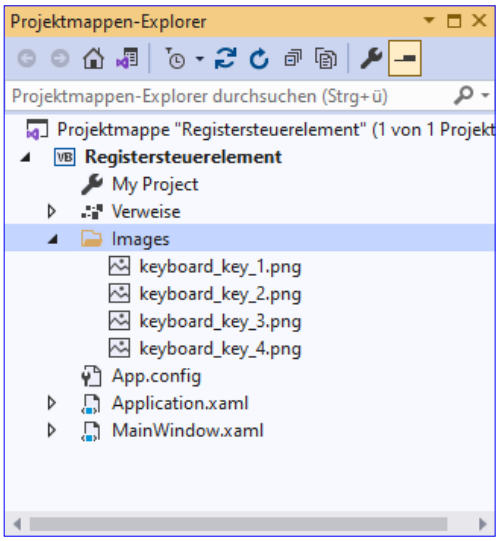


Bild 6: Images für die Registerreiter

Registerreiter anpassen

Wenn Sie von Access kommen, wird Ihnen allein die Idee, die Registerreiter anzupassen, ungewohnt vorkommen – dort war das höchste der Gefühle, den angezeigten Text und die Schriftgröße und -art einzustellen. Unter WPF bieten sich naturgemäß einige zusätzliche Möglichkeiten. Wie wäre es beispielsweise mit ein paar Icons, mit denen der Benutzer noch schneller die gewünschte Registerseite finden kann? Kein Problem: Fügen wir den Registerreitern Icons hinzu!

Registerreiter mit Icons

Um die Registerreiter mit Icons auszustatten, müssen wir dem Projekt zunächst entsprechende Ressourcen hinzufügen. Für diese legen wir einen Ordner namens **Images** an. In diesen ziehen Sie die gewünschten Bilddateien, sodass der Projektmappen-Explorer anschließend wie in Bild 6 aussieht.

Danach passen Sie den Code des **TabItem**-Elements an, indem Sie das **Header**-Attribut, das ja nun nicht mehr nur einen Text aufnehmen soll, sondern ein Icon und einen Text, entfernen und stattdessen ein Property-Element namens **TabItem.Header** zum **TabItem**-Element hinzufügen. Wie schon beim **TabItem**-Element gesehen, können Sie auch dem Property-Element **TabItem.Header** nur ein einziges Element unterordnen. Allerdings kann es sich hierbei auch beispielsweise um ein **StackPanel**-Element handeln, mit dem wir das Icon und die Beschriftung nebeneinander anordnen können. Dafür ist die Einstellung des Attributs **Orientation** des **StackPanel**-Elements auf den Wert **Horizontal** notwendig.

Die Definition für den ersten Registerreiter sieht dann wie folgt aus:

```
<TabControl Margin="5">
    <TabItem>
        <TabItem.Header>
            <StackPanel Orientation="Horizontal">
                <Image Source="Images\keyboard_key_1.png"></Image>
                <Label>Seite 1</Label>
            </StackPanel>
        </TabItem.Header>
        <Label>Dies ist die erste Seite.</Label>
```

```
    </TabItem>
    ...
</TabControl>
```

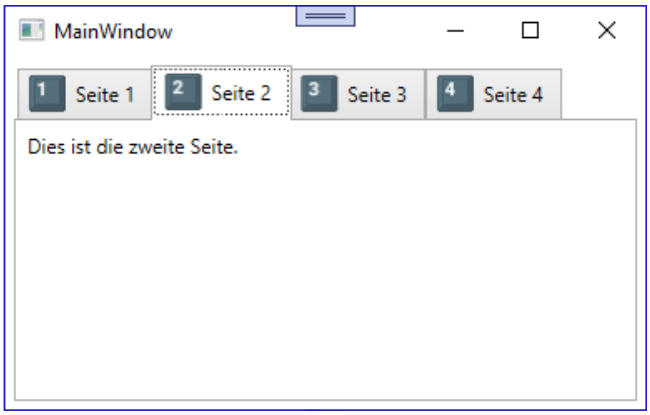


Bild 7: Registerreiter mit Icons

Wenn wir alle vier Registerreiter mit den passenden Icons ausstatten, sieht dies schließlich wie in Bild 7 aus.

Hier gibt es noch viel mehr Möglichkeiten – eigentlich können Sie alle möglichen Elemente zu einem Registerreiter beziehungsweise zu einem **TabItem**-Element hinzufügen. Ob das sinnvoll ist, sei dahingestellt, aber die Anzeige mit Icons und Beschriftung sieht bereits recht professionell aus.

Das TabControl mit Visual Basic programmieren

In den folgenden Abschnitten schauen wir uns verschiedene Möglichkeiten für den Zugriff auf das **TabControl**-Element und seine Steuerung an. Dabei behandeln wir die folgenden Themen:

- Ermitteln des Index der aktuellen Seite des **TabControl**-Elements
- Eine bestimmte Seite im **TabControl**-Element einstellen
- Weitere Informationen über das selektierte **TabItem**-Element ermitteln
- Ereignis beim Wechsel des **TabItem**-Elements
- Kein **TabItem**-Element markieren
- **TabItem**-Elemente per Code hinzufügen
- **TabItem**-Elemente per Code entfernen

Für diese Beispiele haben wir das Fenster ein wenig umstrukturiert. Wir haben dem Grid zwei **RowDefinition**-Elemente hinzugefügt, von denen das obere nun das **TabControl**-Element aufnimmt und das untere ein **StackPanel**-Element, dem wir im weiteren Verlauf einige Schaltflächen hinzufügen. Für das **TabControl**-Element haben wir außerdem den Namen **tab** eingestellt:

```
<TabControl x:Name="tab" Grid.Row="0">
```

Das **StackPanel** in der unteren Zeile des **Grid**-Elements enthält zunächst eine Schaltfläche, zu der aber noch weitere hinzukommen werden:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
    <Button x:Name="btnIndex" Click="btnIndex_Click">Aktueller Index</Button>
</StackPanel>
```

Ermitteln des Index der aktuellen Seite des TabControl-Elements

Für die Schaltfläche **btnIndex** hinterlegen wir eine Ereignismethode namens **btnIndex_Click**, die wie folgt aussieht:

```
Private Sub btnIndex_Click(sender As Object, e As RoutedEventArgs)
    MessageBox.Show("Index der aktuellen Seite: " + tab.SelectedIndex.ToString)
End Sub
```

Hier greifen wir auf die Eigenschaft **SelectedIndex** des **TabControl**-Elements namens **tab** zu und geben diese Eigenschaft in einer MessageBox aus (siehe Bild 8). Der Index ist 0-basiert und deshalb liefert die Eigenschaft **SelectedIndex** den Wert **2**, wenn die dritte Registerseite aktiviert ist.

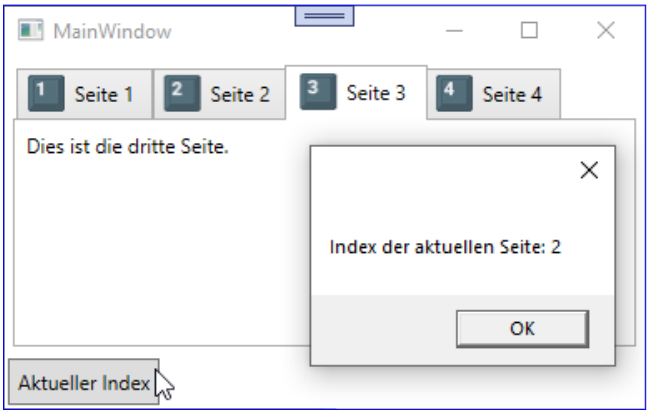


Bild 8: Ausgabe der Eigenschaft **SelectedIndex** des **TabControl**-Elements

Weitere Informationen über das selektierte TabItem-Element ermitteln

Für dieses Beispiel fügen wir eine Schaltfläche zum **StackPanel**-Element hinzu:

```
<Button x:Name="btnTabItemName" Click="btnTabItemName_Click">Aktueller Header</Button>
```

Außerdem haben wir für die **TabItem**-Elemente für das Attribut **x:Name** Namen wie **tab1**, **tab2** und so weiter angegeben:

```
<TabControl x:Name="tab" Margin="5" Grid.Row="0">
    <TabItem x:Name="tab1">...</TabItem>
    <TabItem x:Name="tab2">...</TabItem>
    ...
</TabControl>
```

Die durch das **Click**-Ereignis ausgelöste Ereignismethode referenziert nun das aktuell selektierte **TabItem**-Element mit der Variablen **objTabItem** und gibt seinen Namen in einer MessageBox aus:

```
Private Sub btnTabItemName_Click(sender As Object, e As RoutedEventArgs)
    Dim objTabItem As TabItem
    objTabItem = TryCast(tab.SelectedItem, TabItem)
```

Registerseiten im TabControl an Daten binden

Im Artikel »TabControl: Das Registersteuerelement von WPF« haben wir uns die grundlegenden Eigenschaften des Registersteuerelements von .NET und WPF angesehen. Dieses bietet ähnliche Möglichkeiten wie das Registersteuerelement, das Sie vielleicht von Microsoft Access kennen. An einigen Stellen gehen die Möglichkeiten, bedingt durch die größere Flexibilität der Beschreibungssprache XAML, darüber hinaus. So können Sie beispielsweise das TabControl und seine TabItem-Elemente an eine Datenquelle wie eine ObservableCollection binden und die Daten der Objekte dieser Collection sowohl für die Gestaltung der Registerreiter als auch für den eigentlich Inhalt einer jeden Registerseite nutzen.

Unter Microsoft Access müssen Sie, wenn Sie ein Registersteuerelement mit je einer Registerseite pro Datensatz versehen wollen, jede Registerseite per VBA-Code hinzufügen. Das gelingt zwar, aber unter .NET/WPF ist das wesentlich komfortabler zu erledigen.

Dazu legen Sie ein neues WPF-Projekt mit Visual Basic als Sprache an. Die nachfolgenden Elemente fügen wir dem Hauptfenster **MainWindow.xaml** beziehungsweise der dazu gehörenden Code behind-Klasse **MainWindow.xaml.vb** hinzu.

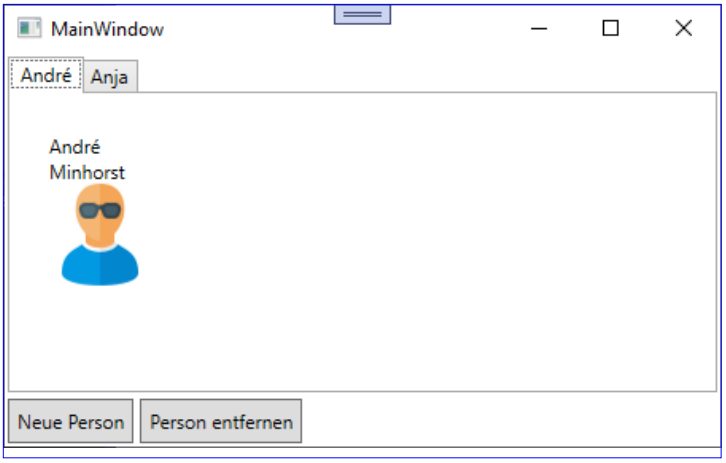


Bild 1: So soll das **TabControl**-Element aussehen

Das Ergebnis dieses Artikels soll wie in Bild 1 aussehen.

Personen-Auflistung als Datenquelle für ein TabControl

Unser Vorhaben ist es, die Elemente einer Klasse namens Person als Datenquelle für ein **TabControl**-Element zu nutzen. Diese Klasse haben wir wie folgt beschrieben:

```
Public Class Person
    Public Property ID As Long
    Public Property Vorname As String
    Public Property Nachname As String
    Public Property Bild As String
End Class
```

Der Datei **MainWindow.xaml.vb** fügen wir außerdem per **Imports**-Anweisung den Namespace **System.Collections.ObjectModel** hinzu, der die für dieses Beispiel benötigte **ObservableCollection**-Klasse beisteuert:

```
Imports System.Collections.ObjectModel
```

In der **MainWindow**-Klasse schließlich deklarieren wir zunächst die Property für die **Personen**-Liste. Diese deklarieren wir als Public, da sie als Datenquelle für das TabControl-Element dienen soll. Wir legen mit **Person** auch gleich den Typ der aufzunehmenden Objekte fest:

```
Public Class MainWindow
    Public Property Personen As ObservableCollection(Of Person)
```

Auflistung der anzuzeigenden Personen füllen

Im Konstruktor der Klasse erledigen wir dann Aufgaben, die wir schon in vielen Artikeln durchgeführt haben. Wir deklarieren ein Objekt der Klasse **Person** und initialisieren das **ObservableCollection**-Element **Personen**:

```
    Public Sub New()
        Dim Person As Person
        Personen = New ObservableCollection(Of Person)
```

Danach erstellen wir zwei **Person**-Elemente und weisen diesen die entsprechenden Eigenschaften zu. Für die Eigenschaft **Bild** geben wir den Pfad von Bilddateien an, die wir im Projektmappen-Explorer in einem separaten Ordner namens **Images** gespeichert haben.

Damit wir diese so wie hier mit **\Images\user_sunglasses.png** referenzieren können, müssen Sie für die Bilddateien die Eigenschaft **In Ausgabeverzeichnis kopieren** auf **Immer kopieren** oder **Kopieren, wenn neuer** einstellen (siehe Bild 2).

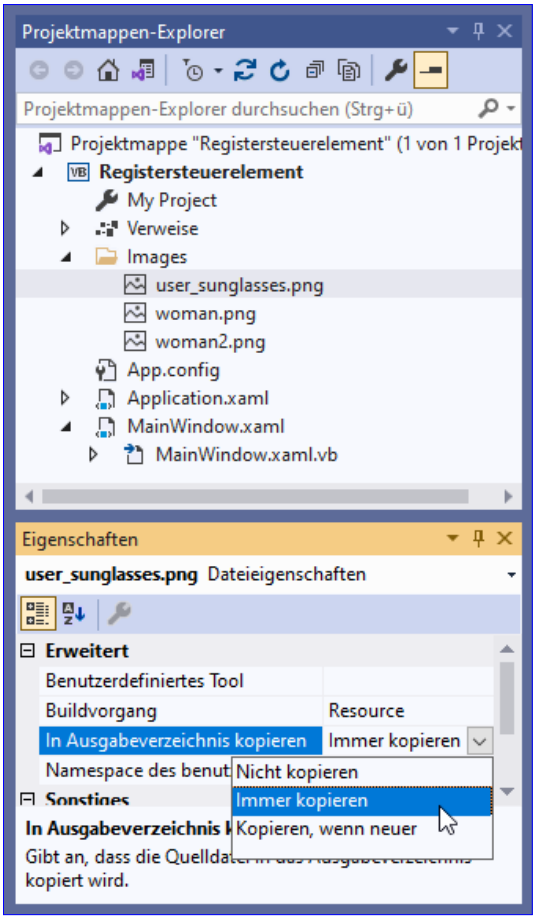


Bild 2: Einstellung, damit die Bilddateien ins Ausgabeverzeichnis kopiert werden

Der Code zum Anlegen der beiden vorerst benötigten **Person**-Elemente und zum Hinzufügen der Elemente mit der **Add**-Methode zur **Personen**-Liste sieht so aus:

```
        Person = New Person
        With Person
            .ID = 1
            .Vorname = "André"
            .Nachname = "Minhorst"
            .Bild = "\Images\user_sunglasses.png"
        End With
```

```
        Personen.Add(Person)
        Person = New Person
        With Person
            .ID = 2
            .Vorname = "Anja"
            .Nachname = "Minhorst"
            .Bild = "\Images\woman2.png"
        End With
        Personen.Add(Person)
```

Schließlich rufen wir noch die Methode **InitializeComponent** auf, damit die XAML-Definition angewendet wird, und stellen die Eigenschaft **DataContext** des Fensters auf das Code behind-Modul ein:

```
        InitializeComponent()
        DataContext = Me
    End Sub
```

Dies zeigt schön, wie die Trennung der Definition der Benutzeroberfläche und der Anwendungslogik unter WPF funktioniert. Wir haben noch keine Elemente zur Benutzeroberfläche hinzugefügt und die Anwendungslogik ist quasi schon fertig. Nun können wir uns an den Entwurf der Benutzeroberfläche begeben.

Layout der Elemente vorbereiten

Wie oben im Bild zu erkennen, wollen wir nicht nur das **TabControl**-Element, sondern auch noch zwei Schaltflächen hinzufügen, mit denen wir neue Elemente zum **TabControl**-Element hinzufügen oder vorhandene Elemente entfernen können. Damit die Schaltflächen und das **TabControl**-Element nicht genau am Rand und aneinander kleben, definieren wir **Margin**- und **Padding**-Attribute für alle Elemente dieser Typen:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="250" Width="400">
    <Window.Resources>
        <Style TargetType="Button">
            <Setter Property="Margin" Value="2"></Setter>
            <Setter Property="Padding" Value="5"></Setter>
        </Style>
        <Style TargetType="TabControl">
            <Setter Property="Margin" Value="2"></Setter>
            <Setter Property="Padding" Value="5"></Setter>
        </Style>
    </Window.Resources>
```

Dann definieren wir ein **Grid**-Element mit zwei Zeilen, von denen die obere das **TabControl**-Element und das untere ein **StackPanel**-Element mit den Schaltflächen aufnehmen soll:

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

TabControl-Element hinzufügen

Schließlich folgt das **TabControl**-Element. Dieses sieht in der nachfolgenden Definition nur in der ersten Zeile ungefähr so aus wie das mit statischen Daten gefüllte Registersteuerelement aus dem Artikel **TabControl: Das Registersteuerelement von WPF** (**www.datenbankentwickler.net/290**). Aber schon die erste Zeile verwendet das **ItemsSource**-Attribut und weist diesem eine Bindung an die Eigenschaft **Personen** zu, welche das **ObservableCollection**-Element enthält.

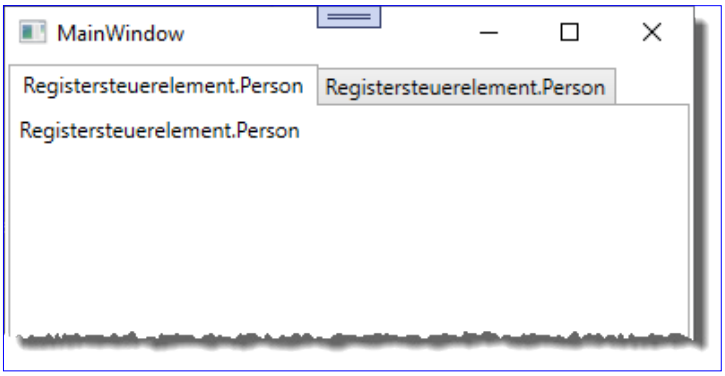


Bild 3: **TabControl**-Element nur mit Bindung, aber noch ohne Inhalte

Sie können das Projekt bereits jetzt einmal starten und finden dann die Ansicht aus Bild 3 vor. Die Bindung funktioniert also – die Registerseiten sind bereits jetzt auf **Person**-Elemente eingestellt:

```
<TabControl x:Name="tab" ItemsSource="{Binding Personen}">
    ...
</TabControl>
```

Header des TabItem-Elements füllen

Allerdings wollen wir im Header der Registerseiten den Vornamen der Person anzeigen und auf der jeweiligen Registerseite die Inhalte aller Felder der **Person**-Elemente. Für die Anzeige des Vornamens im Header fügen wir ein Property-Element namens **TabControl.ItemsContainerStyle** hinzu. Dieses erhält ein **Style**-Element, das sich auf die **TabItem**-Elemente beziehen soll. Der Name der Eigenschaft, die wir in diesem Style definieren wollen, heißt **HeaderTemplate**. Der Wert ist ein **DataTemplate**-Element, in dem wir angeben, dass der Header ein **StackPanel**-Element mit einem **TextBlock**-Element enthalten soll. Dieser ist über das Attribut **Text** an die Eigenschaft **Vorname** der übergeordneten Datenquelle gebunden:

```
<TabControl.ItemContainerStyle>
    <Style TargetType="TabItem">
        <Setter Property="HeaderTemplate">
            <Setter.Value>
                <DataTemplate>
                    <StackPanel>
```

Unit-Testing mit Visual Studio

In den bisherigen Ausgaben von Datenbankentwickler haben wir ohne moderne Entwurfsmuster programmiert und beispielsweise den Code von Fenstern im Code behind-Modul gespeichert. Das wollen wir nun ändern und das Entwurfsmuster Model-View-ViewModel (kurz MVVM) vorstellen. Das ermöglicht durch eine Zwischenschicht zwischen der Benutzeroberfläche und dem Entity Data Model das automatisierte Testen der meisten Funktionen einer Anwendung. Durch automatisiertes Testen, hier Unit-Testing genannt, können wir Tests für unseren Code definieren, die wir immer wieder per Mausklick durchführen können. So können Sie Änderungen am Code durchführen und sich blitzschnell versichern, dass der Code noch so läuft wie gewünscht. Im vorliegenden Artikel erläutern wir, was es mit Unit-Testing auf sich hat und wie Sie es unter Visual Studio einsetzen.

Tests von Softwareanwendungen sind meist aufwendig, vor allem, wenn sie manuell durchgeführt werden. Das größte Problem dabei ist: Wenn sich eine Funktion der Anwendung ändert, muss man normalerweise zumindest alle damit zusammenhängenden Funktionen erneut testen, was Zeit und Geld kostet. Eine Automatisierung der Tests über die Benutzeroberfläche ist ebenfalls kritisch zu sehen, was den Aufwand betrifft, denn auch hier können kleinste Änderungen dazu führen, dass die Tests nicht mehr funktionieren und ebenfalls angepasst werden müssen.

Wenn Sie mit einem Entwurfsmuster wie dem eingangs erwähnten MVVM arbeiten, wobei es eine Schicht gibt, welche die Anwendungslogik enthält und die Daten der Benutzeroberfläche liefert, aber keine Abhängigkeit von der Benutzeroberfläche hat, können Sie diese Schicht mit der Anwendungslogik auch ohne den Umweg über die Benutzeroberfläche testen. Dazu initialisieren Sie die entsprechenden Objekte und testen deren Methoden.

Weil sich diese Tests meist auf einzelne Klassen oder Komponenten beziehen, werden sie im Allgemeinen Unit-Tests genannt.

Beispielprojekt für die Unit-Tests erstellen

Da wir in diesem Magazin meist Desktop-Anwendungen programmieren, wollen wir auch als Beispiel für die Unit-Tests ein solches Projekt nutzen. Also legen Sie ein neues Projekt des Typs **WPF-App (.NET-Framework)** an und legen für dieses den Namen **UnitTestBeispiel** fest.

Im Anschluss an diesen Vorgang sollte Visual Studio geöffnet sein und im Projektmappen-Explorer das neu erstellte Projekt anzeigen.

Mögliche Konfigurationen für Unit-Tests

Wenn Sie Unittests durchführen wollen, haben Sie verschiedene Möglichkeiten – zum Beispiel die folgenden:

- Sie erstellen ein eigenes Projekt, das nur die Unittests enthält und das Sie gemeinsam mit dem Testprojekt in einer Projektmappe verwalten.

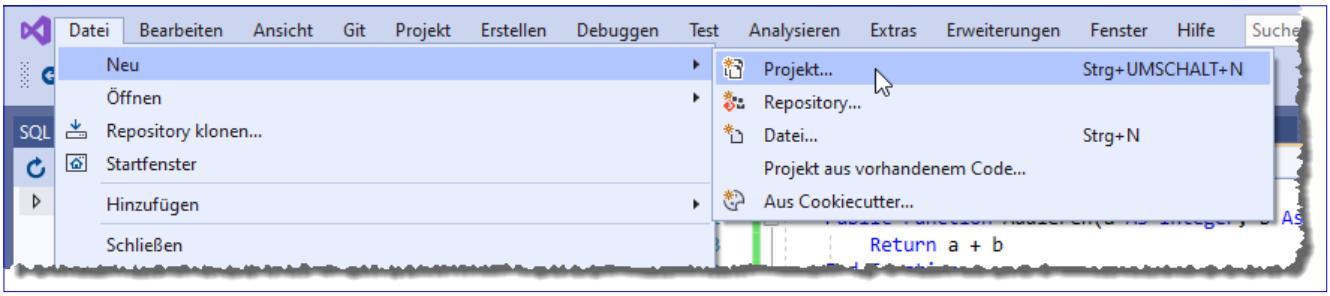


Bild 1: Anlegen eines neuen Projekts

- Sie erstellen die zum Testen notwendigen Elemente innerhalb des zu testenden Projekts.

Eines der Ziele beim Unit-Testing ist, den Code des zu testenden Projekts nicht aufzublähen. Deshalb fügen wir die Unit-Tests auch nicht zu dem Projekt hinzu, das wir testen wollen, sondern legen ein eigenes Projekt dafür an. Dieses Projekt soll allerdings in einer Projektmappe mit dem zu testenden Projekt landen. Deshalb gehen wir wie folgt vor, um das Testprojekt zu erstellen:

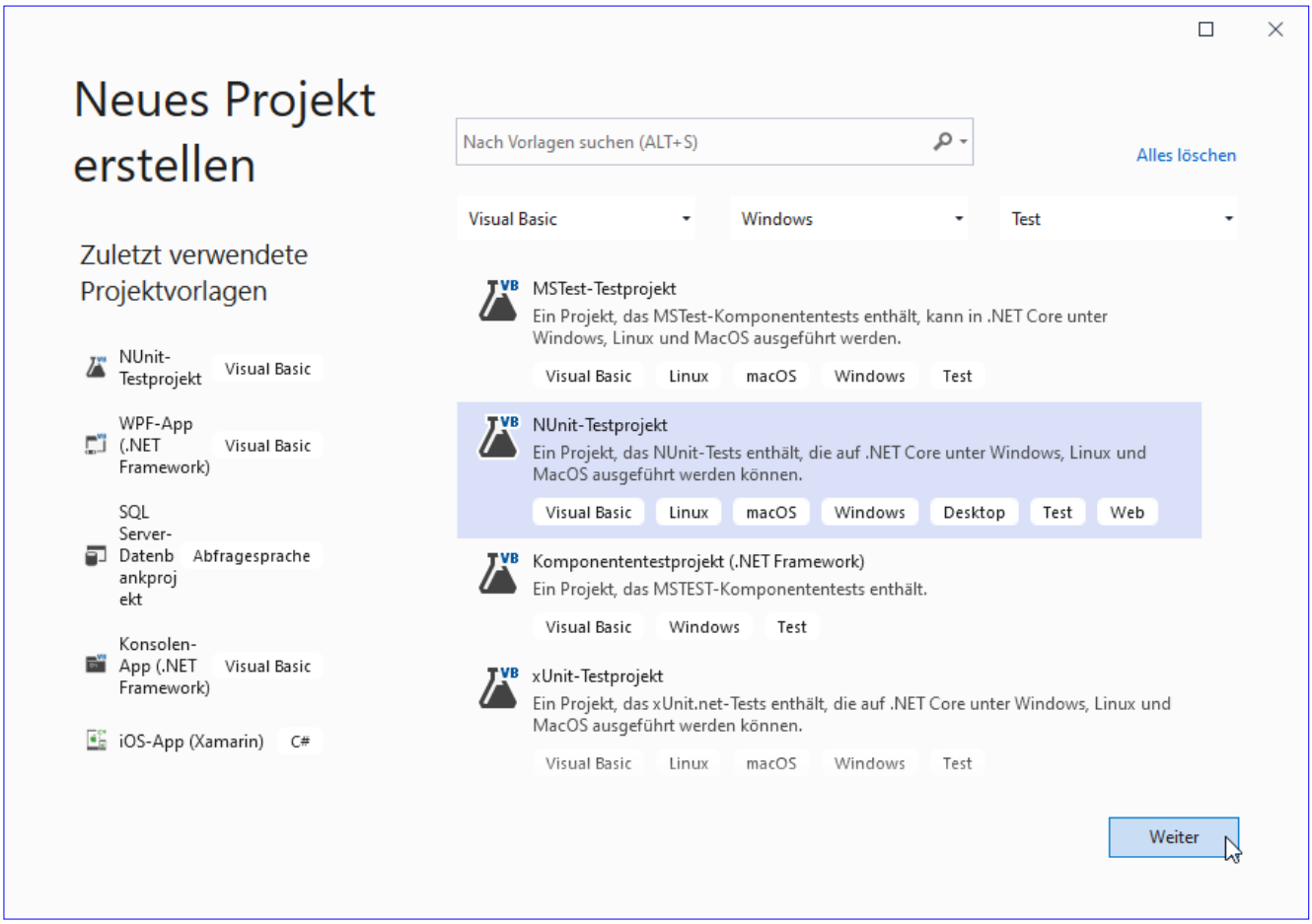


Bild 2: Auswahl des Projekttyps

- Betätigen Sie den Menübefehl **Datei|Neu|Projekt...** (siehe Bild 1).

- Wählen Sie im Dialog **Neues Projekt erstellen** mit dem linken Auswahlfeld den Eintrag **Visual Basic** und mit dem rechten Auswahlfeld den Eintrag **Test** aus und finden Sie die verfügbaren Testprojekte in der Liste. Hier wählen wir die Projektvorlage **NUnit-Testprojekt** aus (siehe Bild 2).

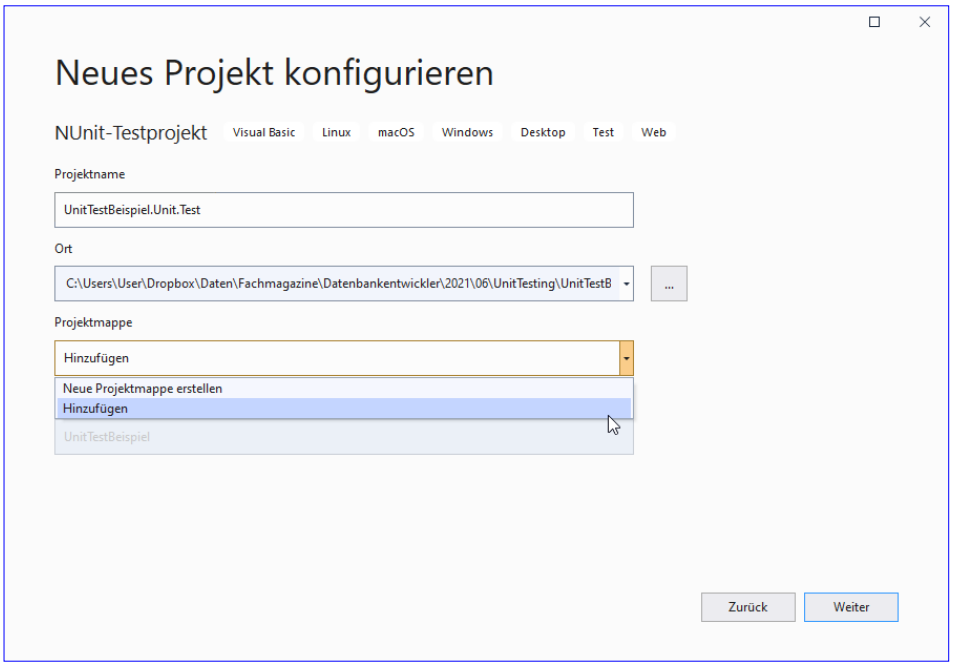


Bild 3: Hinzufügen des Projekts zur bestehenden Projektmappe

- Im folgenden Schritt legen Sie den neuen Projektnamen fest. Diesen wählen wir so, dass er mit dem Namen des zu testenden Projekts beginnt und hängen die Zeichenfolge **.Unit.Test** an. Außerdem stellen wir hier ein, dass das neue Projekt der bestehenden Projektemappe hinzugefügt werden soll (siehe Bild 3).

- Schließlich legen Sie noch das Zielframework für das neue Projekt fest.

Das gleiche Ergebnis erhalten Sie übrigens auch, wenn Sie den Menübefehl **Datei|Hinzufügen|Neues Projekt...** betätigen – Sie brauchen dann im Dialog **Neues Projekt konfigurieren** nicht mehr festzulegen, dass das neue Projekt zur aktuellen Projektmappe hinzugefügt werden soll.

Das neue Unit-Test-Projekt

Das Anlegen des Projekts hat zwei sichtbare Änderungen in unserem Projekt bewirkt. Die erste ist, dass wir im Projektmappen-Explorer unter unserem eigentlichen Projekt auch das neu angelegte Unit-Test-Projekt vorfinden (siehe Bild 4).

Die zweite Änderung ist das Klassenmodul **UnitTest1.vb**, das wir im Codefenster von Visual Studio vorfinden (siehe Bild 5). Die Bestandteile schauen wir uns gleich genauer an.

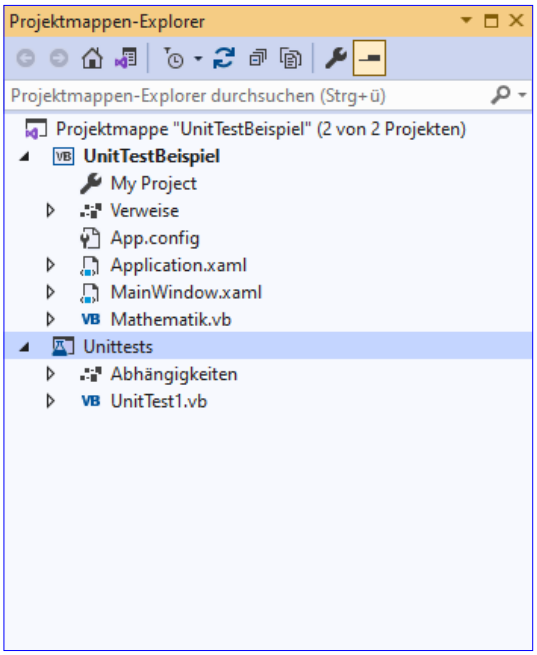


Bild 4: Das neue Testprojekt in der Projektmappe des zu testenden Projekts

Verweis zum Unit-Test-Projekt hinzufügen

Bevor wir uns die Unit-Test-Vorlage ansehen, die zum Testprojekt hinzugefügt wurde, wollen wir noch einen Verweis zum Testprojekt hinzufügen.

Dieser soll auf das zu testende Projekt verweisen. Um das zu erledigen, klicken Sie im Projektmappen-Explorer unter dem Testprojekt mit der rechten Maustaste auf den Eintrag **Abhängigkeiten** und wählen den Kontextmenübefehl **Projektverweis hinzufügen...** aus (siehe Bild 6).

Der nun erscheinende Verweis-Manager zeigt direkt die Projektmappe des zu testenden Projekts an. Hier setzen Sie einen Haken vor den Eintrag **UnitTestBeispiel** und schließen den Dialog wieder (siehe Bild 7).

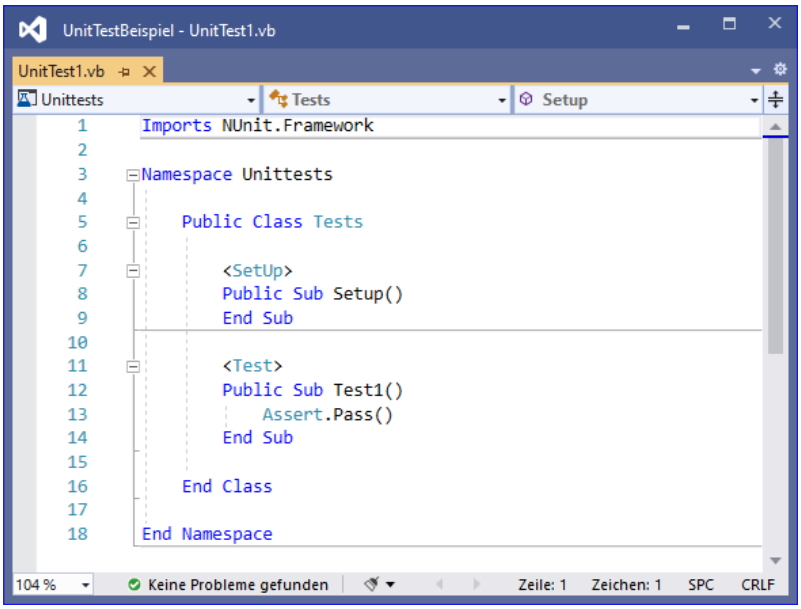


Bild 5: Die Vorlage für eine Testklasse

Die Unit-Test-Vorlage

Die unter dem Namen **UnitTest1.vb** hinzugefügte Klasse importiert zunächst den Namespace **NUnit.Framework**:

```
Imports NUnit.Framework
```

Dann deklariert sie einen Namespace namens **Unittests**, der eine erste Testklasse namens **Tests** enthält:

```
Namespace Unittests
    Public Class Tests
```

Hier finden wir einen Bereich, der mit dem Attribut **SetUp** versehen ist und eine Methode namens **Setup** enthält.

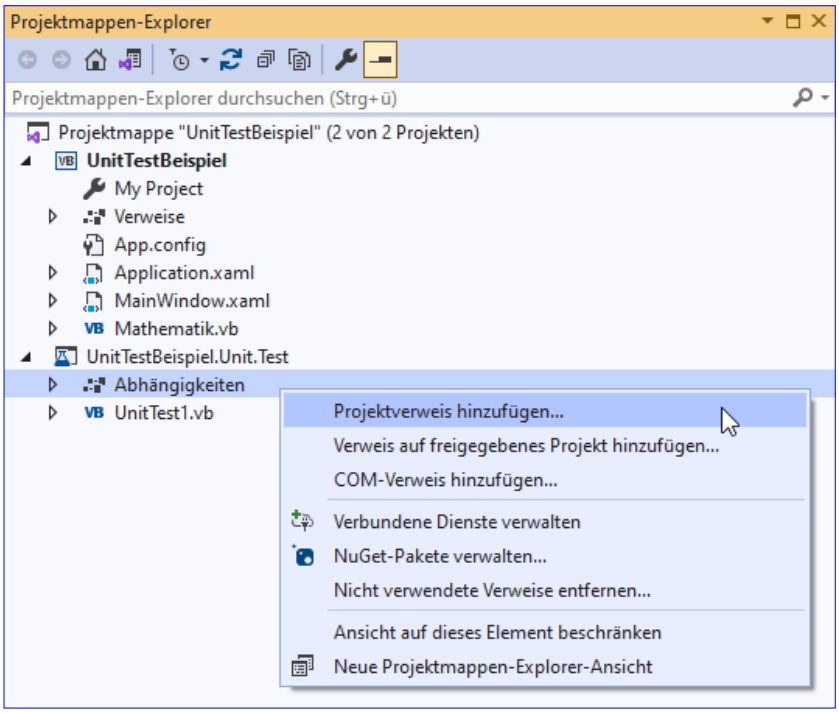


Bild 6: Hinzufügen eines Projektverweises

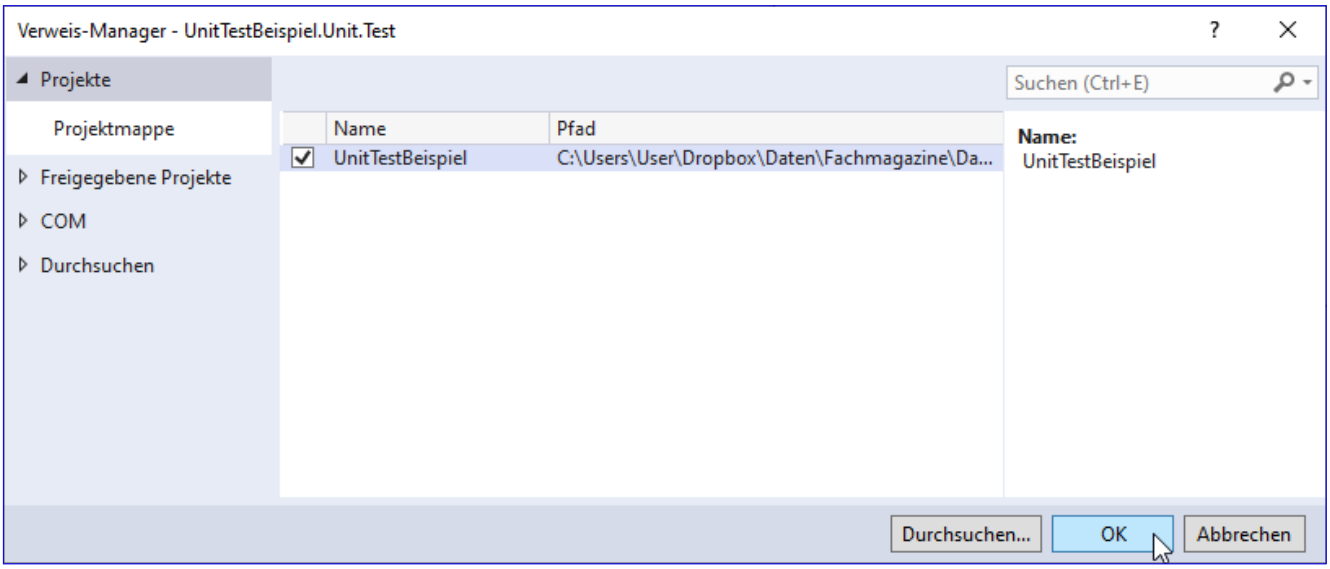


Bild 7: Hinzufügen des Verweises auf das zu testende Projekt

Dieser Methode können Sie Anweisungen hinzufügen, welche vor dem Aufruf einer jeden Testmethode ausgeführt werden:

```
        <SetUp>
        Public Sub Setup()
        End Sub
```

Danach folgen die eigentlichen Testmethoden. Diese werden mit dem Attribut **Test** versehen und sehen wie herkömmliche Methoden aus. Sie benötigen allerdings den Aufruf einer Methode der **Assert**-Klasse. Im Beispiel der Testvorlage lautet diese Methode **Pass**. **Pass** heißt, dass der Test bestanden wurde:

```
        <Test>
        Public Sub Test1()
            Assert.Pass()
        End Sub
    End Class
End Namespace
```

Den ersten Unit-Test durchführen

Um diesen Unit-Test zu starten, nutzen Sie die dafür vorgesehene Benutzeroberfläche.

Diese erscheint, wenn Sie den Menübefehl **Test|Alle Tests ausführen** aufrufen, automatisch (siehe Bild 8).

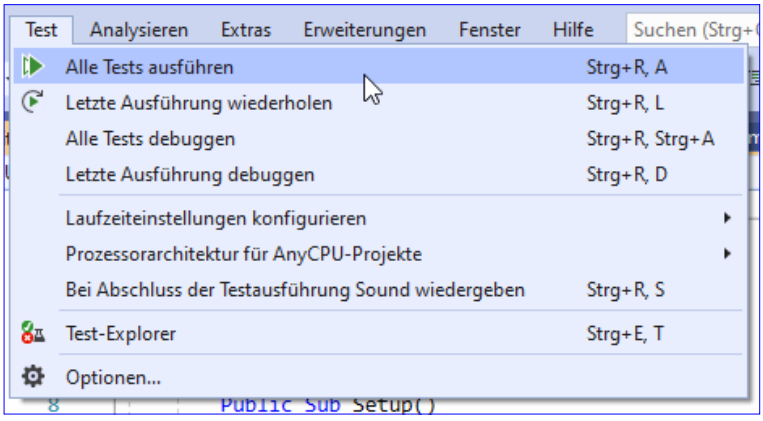


Bild 8: Starten der Tests

Sie sehen dann im Bereich **Test-Explorer** den Ablauf und das Ergebnis des ersten Tests. Dieser verläuft mit dem Ergebnis **Bestanden**, was vorhersehbar war – **Assert.Pass** liefert immer einen bestandenen Test (siehe Bild 9).

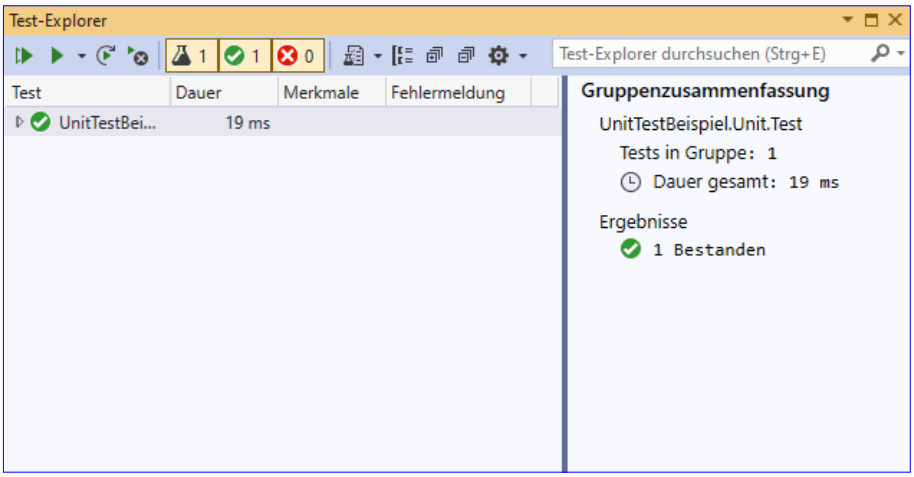


Bild 9: Ergebnis des ersten Tests

Einen echten Test programmieren

Damit können wir einen Schritt weitergehen und einen ersten echten Test programmieren.
Wir gehen hier direkt so vor, wie man es beim Test Driven Development machen würde: Das heißt, wir schreiben erst einen Test und fügen erst dann die Funktionalität hinzu, welche getestet werden soll.

Was soll der Unit-Test testen?

Was wir dazu schon einmal wissen sollten, ist der Name der Klasse, welche die zu testenden Methoden enthält. Für ein erstes Beispiel wollen wir eine Klasse namens **Mathematik.vb** programmieren, welche Methoden zum Addieren und zum Subtrahieren zweier Zahlen bereitstellt. Diese sollen **Addieren** und **Subtrahieren** heißen.

Die Funktionen sind, wie soeben erwähnt, zunächst einmal leer und liefern dementsprechend den Standardwert für den Datentyp des Funktionsergebnisses zurück, im Falle des Datentyps **Integer** den Wert **0**:

```
Public Class Mathematik
    Public Function Addieren(a As Integer, b As Integer) As Integer

    End Function

    Public Function Subtrahieren(a As Integer, b As Integer) As Integer

    End Function
End Class
```

Erstellen der Testklasse

Der Name der Testklasse soll mit dem Namen der zu testenden Klasse übereinstimmen und den Zusatz **Test** enthalten. Folglich nennen wir die Testklasse **MathematikTest.vb**. Analog erhalten die Methoden, mit denen die eigentlichen Methoden getestet werden sollen, den gleichen Namen wie die Methoden selbst, erweitert mindestens mit dem Zusatz **Test** – also beispielsweise **AddierenTest**. In den meisten Fällen werden Sie mit einem Test

Seminarverwaltung IVa: Kunden und Seminare

Die Arbeit mit der Seminarverwaltung startet an dem Punkt, wo wir die online eingegangenen Bestellungen einlesen und in der Seminarverwaltung Kunden und Bestellungen in Kunden und Seminarteilnahmen umwandeln. Dazu benötigen wir eine Möglichkeit, die online erfolgten Bestellungen in die Anwendung einzulesen – siehe Artikel »Seminarverwaltung VI: Daten vom Shopsystem«. Außerdem wollen wir in diesem Artikel zeigen, wie Sie die Kunden und die Seminarteilnahmen verwalten können. Hier kann es auch vorkommen, dass ein Kunde auf einem anderen Wege als im Onlineshop bestellt – daher benötigen wir Möglichkeit zum Anlegen von Kunden und Seminarteilnahmen über die Benutzeroberfläche.

Wir haben diesen Artikel gegenüber der ursprünglichen Planung dreigeteilt. Den ersten Teil finden Sie hier vor, den zweiten und dritten Teil beschreiben wir gleich im Anschluss. In diesem Teil der Artikelreihe zeigen wir, wie Sie die Seiten zum Verwalten von Kunden und Seminaren und der Zuordnung von Kunden zu Seminaren und umgekehrt realisieren können. Dazu legen wir jeweils **Page**-Elemente an, und zwar die folgenden:

- **KundenUebersicht.xaml**: Diese Seite zeigt eine Liste aller Kunden an und bietet die Möglichkeit, die Kunden nach verschiedenen Kriterien zu durchsuchen. Außerdem enthält sie Schaltflächen zum Anlegen neuer Kunden, zum Löschen von Kunden und zum Bearbeiten eines Kunden.

- **Kundendetails.xaml**: Diese Seite zeigt die Details zu einem Kunden an. Dazu gehören neben den Daten der Tabelle **tblKunden** auch die von diesem Kunden gebuchten Seminare. Die Seite soll auch die Möglichkeit bieten, Seminare zum Kunden hinzuzufügen oder zu entfernen.

In der ersten Fortsetzung dieses Artikels namens **Seminarverwaltung IVb: Kunden und Seminare** (**www.datenbankentwickler.net/294**) schauen wir uns die Seiten zur Darstellung der Seminare in der Übersicht und in der Einzelansicht an. Die dort beschriebenen Seiten lauten:

- **SeminareUebersicht.xaml**: Die Übersichtsseite für die Seminare soll eine Liste aller Seminare anzeigen. Hier soll der Benutzer eine Suchfunktion vorfinden sowie Schaltflächen zum Anlegen neuer Seminare und zum Bearbeiten oder Löschen von Seminaren.

- **Seminardetails.xaml**: Die Detailseite eines Seminars zeigt neben den Details des Seminars auch eine Liste der Kunden an, die dieses Seminar gebucht haben. Sie können hier manuell Kunden hinzufügen oder aus der Liste der Teilnehmer entfernen. Außerdem finden Sie hier Funktionen, um den Kunden Informationen zum Seminar zuzusenden – beispielsweise E-Mails mit dem Link zur Teams-Sitzung, in der das Seminar stattfindet, oder für die Aufzeichnung des Seminars.

Schließlich folgt noch der dritte Teil, den wir **Seminarverwaltung IVc: Kunden und Seminare** (**www.datenbankentwickler.net/295**) genannt haben. In diesem beschreiben wir, wie wir den in den ersten beiden Teilen

erläuterten Seiten noch die Elemente hinzufügen, mit denen Sie die Zuordnung von Kunden zu Seminaren verwalten können.

Die Kundenübersicht

Die Kundenübersicht soll alle Kunden in einem **ListView**-Steuerelement anzeigen. Außerdem wollen wir folgende Funktionen bereitstellen:

- Öffnen der Details zu einem Kunden per Doppelklick auf den jeweiligen Eintrag
- Öffnen der Details zu einem Kunden durch Auswahl und anschließendes Betätigen der **Bearbeiten**-Schaltfläche
- Löschen des aktuell markierten Kunden per Schaltfläche
- Anlegen eines neuen Kunden per Schaltfläche
- Suche nach Kunden über den Vornamen und den Nachnamen

Der Entwurf dieser Seite namens **Kundendetails.xaml** sieht wie in Bild 1 aus.

Im XAML-Code haben wir ein Grid definiert, das drei Zeilen aufnimmt. Die erste enthält ein StackPanel mit den Suchfeldern, die zweite das **ListView**-Steuerelement und die dritte die Steuerelemente zum Verwalten der angezeigten Kunden:

```
<Page x:Class="KundenUebersicht" ... Title="KundenUebersicht">
    ...
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
```

Bild 1: Entwurf der Seite zur Anzeige der Kundenübersicht

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

Das StackPanel für die Suche definieren wir mit zwei Textfeldern zur Eingabe von Vorname und/oder Nachname und einer Schaltfläche zum Ausführen der Suche:

```
    <StackPanel Orientation="Horizontal">
        <Label>Vorname:</Label>
        <TextBox x:Name="txtSucheVorname" Width="100"></TextBox>
        <Label>Nachname:</Label>
        <TextBox x:Name="txtSucheNachname" Width="100"></TextBox>
        <Button x:Name="btnSuchen" Click="btnSuchen_Click">
            <StackPanel Orientation="Horizontal">
                <Image Source="images/find_text.png" Width="24" Height="24"></Image>
            </StackPanel>
        </Button>
    </StackPanel>
```

Das **ListView**-Steuerelement namens **lvwKunden** ist an die Auflistung **Kunden** aus dem Code behind-Modul gebunden. Es erlaubt das Auswählen jeweils eines Elements und bei einem Doppelklick soll eine Ereignismethode ausgelöst werden. Die einzelnen anzuzeigenden Spalten definieren wir mit **GridViewColumn**-Elementen:

```
    <ListView x:Name="lvwKunden" Grid.Row="1" ItemsSource="{Binding Kunden}" SelectionMode="Single"
            MouseDoubleClick="lvwKunden_MouseDoubleClick">
        <ListView.View>
            <GridView>
                <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ID}" />
                <GridViewColumn Header="Firma" DisplayMemberBinding="{Binding Firma}" />
                <GridViewColumn Header="Vorname" DisplayMemberBinding="{Binding Vorname}" />
                <GridViewColumn Header="Nachname" DisplayMemberBinding="{Binding Nachname}" />
                <GridViewColumn Header="Straße" DisplayMemberBinding="{Binding Strasse}" />
                <GridViewColumn Header="PLZ" DisplayMemberBinding="{Binding PLZ}" />
                <GridViewColumn Header="Ort" DisplayMemberBinding="{Binding Ort}" />
                <GridViewColumn Header="Land" DisplayMemberBinding="{Binding Land}" />
                <GridViewColumn Header="E-Mail" DisplayMemberBinding="{Binding EMail}" />
            </GridView>
        </ListView.View>
    </ListView>
```

Die dritte Zeile des **Grid**-Elements enthält ein weiteres **StackPanel**-Element mit den Steuerelementen zum Verwalten der Kunden – hier mit Schaltflächen zum Schließen des Bereichs (**btnOK**), zum Anlegen neuer Kunden

(**btnNeu**), zum Anzeigen des aktuellen Kunden (**btnDetails**) und zum Löschen des aktuell markierten Kunden (**btnLoeschen**):

```
        <StackPanel Orientation="Horizontal" Grid.Row="2">
            <Button x:Name="btnOK" Click="btnOK_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/ok.png" Width="24" Height="24"></Image>
                    <Label>OK</Label>
                </StackPanel>
            </Button>
            <Button x:Name="btnNeu" Click="btnNeu_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/add.png" Width="24" Height="24"></Image>
                    <Label>Neu</Label>
                </StackPanel>
            </Button>
            <Button x:Name="btnDetails" Click="btnDetails_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/pencil.png" Width="24" Height="24"></Image>
                    <Label>Bearbeiten</Label>
                </StackPanel>
            </Button>
            <Button x:Name="btnLoeschen" Click="btnLoeschen_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/delete.png" Width="24" Height="24"></Image>
                    <Label>Löschen</Label>
                </StackPanel>
            </Button>
        </StackPanel>
    </Grid>
</Page>
```

Anzeigen der Kunden im ListView-Steuerelement

Im Code behind-Modul benötigen wir den folgenden Namespace-Verweis:

```
Imports System.Collections.ObjectModel
```

Außerdem deklarieren wir hier die folgenden Variablen:

```
Private dbContext As New SeminarverwaltungContext
Private _kunden As ObservableCollection(Of Kunde)
```

```
Private objFrame As Frame
Private objListCollectionView As ListCollectionView
```

Beim Initialisieren der Seite wird die Konstruktor-Methode ausgelöst. Diese erwartet als Parameter einen Verweis auf das übergeordnete **Frame**-Element, beispielsweise zum Leeren des **Frame**-Elements, wenn die Seite geschlossen wird. Diesen Verweis speichert die Prozedur in der Variablen **objFrame**. Außerdem initialisiert die Methode die Komponente (**InitializeComponent**), füllt die Liste der Kunden, weist sich selbst dem **DataContext** zu und referenziert mit **objListCollectionView** eine Sicht auf die Auflistung **Kunden**. Schließlich stellt sie die Methode **Kundenfilter** als auszuführende Methode beim Aufruf des Filters der **ListCollectionView** ein:

```
Public Sub New(fra As Frame)
    objFrame = fra
    InitializeComponent()
    Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
    DataContext = Me
    objListCollectionView = CollectionViewSource.GetDefaultView(Kunden)
    objListCollectionView.Filter = AddressOf Kundenfilter
End Sub
```

Die öffentliche Eigenschaft **Kunden** erfasst die Auflistung der Kunden und stellt diese als Quelle für das **ListView**-Steuerelement zur Verfügung:

```
Public Property Kunden As ObservableCollection(Of Kunde)
    Get
        Return _kunden
    End Get
    Set(value As ObservableCollection(Of Kunde))
        _kunden = value
    End Set
End Property
```

Filtern nach dem Vornamen und dem Nachnamen

Die Steuerelemente im oberen Bereich der Kundenübersicht erlauben die Eingabe eines Vornamens und einen Nachnamens, nach dem gefiltert werden soll. Für die Eigenschaft **Filter** des **ListCollectionView**-Elements haben wir weiter oben bereits die Funktion **Kundenfilter** zugewiesen. Die Funktion **Kundenfilter** wird beim Filtern der **ListCollectionView** für jeden Datensatz einmal aufgerufen. Dabei erfasst sie den Vornamen und den Nachnamen des aktuellen **Kunde**-Elements und schreibt diese beziehungsweise gegebenenfalls eine leere Zeichenkette in die Variablen **strVorname** und **strNachname**. Dann prüft sie, ob Nachname und Vorname mit den Werten aus **txtSucheNachname** beziehungsweise **txtSucheVorname** beginnen. Falls nicht, wird der Wert **False** zurückgegeben. Ist beides erfüllt, gibt die Funktion den Wert **True** zurück und des jeweilige Element wird angezeigt:

```
Private Function Kundenfilter(item As Object) As Boolean
    Dim objKunde As Kunde
    Dim strVorname As String
    Dim strNachname As String
    objKunde = TryCast(item, Kunde)
    strVorname = IIf(objKunde.Vorname = Nothing, "", objKunde.Vorname)
    strNachname = IIf(objKunde.Nachname = Nothing, "", objKunde.Nachname)
    If Not (strNachname.StartsWith(txtSucheNachname.Text, StringComparison.OrdinalIgnoreCase)) Then
        Return False
    End If
    If Not (strVorname.StartsWith(txtSucheVorname.Text, StringComparison.OrdinalIgnoreCase)) Then
        Return False
    End If
    Return True
End Function
```

Damit der Filter angewendet wird, muss der Benutzer noch die Schaltfläche **btnSuchen** betätigen. Diese ruft die Methode **Refresh** der **ListCollectionView** auf, wodurch der Filter neu gesetzt wird:

```
Private Sub btnSuchen_Click(sender As Object, e As RoutedEventArgs)
    objListCollectionView.Refresh()
End Sub
```

Einen Eintrag löschen

Die Schaltfläche **btnLoeschen** soll den aktuell markierten Eintrag löschen. Die dadurch ausgelöste Methode sieht wie folgt aus:

```
Private Sub btnLoeschen_Click(sender As Object, e As RoutedEventArgs)
    Dim ZuLoeschenderKunde As Kunde
    ZuLoeschenderKunde = lvwKunden.SelectedItem
    If Not ZuLoeschenderKunde Is Nothing Then
        With dbContext
            .Kunden.Remove(ZuLoeschenderKunde)
            .SaveChanges()
        End With
        Kunden.Remove(ZuLoeschenderKunde)
    Else
            MessageBox.Show("Markieren Sie den zu löschenden Eintrag.", "Kein Eintrag markiert", _
                MessageBoxButton.OK, MessageBoxImage.Information)
    End If
End Sub
```

Die Methode ermittelt zunächst über die Eigenschaft **SelectedItem** das aktuell markierte Element im **ListView**-Steuerelement und referenziert es mit der Variablen **ZuLoeschenderKunde**. Dann prüft sie, ob überhaupt ein Eintrag markiert ist.

Falls ja, entfernt die Methode diesen zunächst aus der Auflistung **Kunden** des **dbContext**-Elements und überträgt die Änderung mit der **SaveChanges**-Methode in die zugrunde liegende Datenbank. Anschließend entfernt sie das Element auch noch aus der Auflistung **Kunden** und somit auch aus dem **ListView**-Steuerelement.

Hat der Benutzer zu diesem Zeitpunkt keinen Eintrag im **ListView**-Steuerelement markiert, erscheint eine entsprechende Meldung.

Details zu einem Kunden anzeigen

Betätigt der Benutzer die Schaltfläche **btnDetails** oder klickt er doppelt auf einen der Einträge im **ListView**-Steuerelement, soll die Seite **Kundendetails.xaml** die Details zum aktuellen Eintrag der Liste anzeigen. Beim Betätigen der Schaltfläche **btnDetails** prüft die folgende Methode, ob ein Kunde markiert ist, bevor sie die Methode **KundendetailsAnzeigen** aufruft. Anderenfalls folgt ein entsprechender Hinweis per Meldung:

```
Private Sub btnDetails_Click(sender As Object, e As RoutedEventArgs)
    If Not lvwKunden.SelectedItem Is Nothing Then
        KundendetailsAnzeigen()
    Else
        MessageBox.Show("Markieren Sie den zu anzuzeigenden Eintrag.", "Kein Eintrag markiert", _
            MessageBoxButton.OK, MessageBoxImage.Information)
    End If
End Sub
```

Beim Doppelklick auf einen der Listeneinträge ruft die folgende Ereignismethode die Routine **KundendetailsAnzeigen** auf:

```
Private Sub lvwKunden_MouseDoubleClick(sender As Object, e As MouseButtonEventArgs)
    KundendetailsAnzeigen
End Sub
```

Die Methode zum Anzeigen des aktuell markierten Kunden sieht nun wie folgt aus:

```
Private Sub KundendetailsAnzeigen()
    Dim pgeKundendetails As Kundendetails
    Dim MarkierterKunde As Kunde
    MarkierterKunde = lvwKunden.SelectedItem
    pgeKundendetails = New Kundendetails(objFrame, dbContext, MarkierterKunde)
    AddHandler pgeKundendetails.KundeChanged, AddressOf Kundendetails_KundeChanged
```

Seminarverwaltung IVb: Kunden und Seminare

In diesem Teil der Artikelreihe zur Seminarverwaltung zeigen wir, wie Sie die Seminare verwalten. Dazu benötigen wir zwei Seiten – eine zur Anzeige der Übersicht aller Seminare und eine zur Anzeige der Details eines einzelnen Seminars beziehungsweise zum Anlegen eines neuen Seminars. Dabei müssen wir noch eine kleine Anpassung am Entity Date Model durchführen, und außerdem legen wir noch die Methoden im Hauptfenster an, mit denen Sie die beiden Seiten zur Anzeige und Bearbeitung der Seminare darstellen

Wir haben diesen Artikel gegenüber der ursprünglichen Planung dreigeteilt. Den zweiten Teil finden Sie hier vor. Im ersten Teil namens **Seminarverwaltung IVa: Kunden und Seminare** (**www.datenbankentwickler.net/273**) stellen wir die beiden Seiten zur Anzeige der Übersicht und der Details der Kunden vor.

Im dritten Teil zeigen wir unter dem Titel **Seminarverwaltung IVc: Kunden und Seminare** (**www.datenbankentwickler.net/295**), wie Sie den Kunden Seminare zuordnen und umgekehrt.

Im aktuellen Teil der Artikelreihe schauen wir uns die Seiten zur Darstellung der Seminare in der Übersicht und in der Einzelansicht an. Die hier beschriebenen Seiten lauten:

- **SeminareUebersicht.xaml**: Die Übersichtsseite für die Seminare soll eine Liste aller Seminare anzeigen. Hier soll der Benutzer eine Suchfunktion vorfinden sowie Schaltflächen zum Anlegen neuer Seminare und zum Bearbeiten oder Löschen von Seminaren.

- **Seminardetails.xaml**: Die Detailseite eines Seminars zeigt neben den Details des Seminars auch eine Liste der Kunden an, die dieses Seminar gebucht haben. Sie können hier manuell Kunden hinzufügen oder aus der Liste der Teilnehmer entfernen. Außerdem finden Sie hier Funktionen, um den Kunden Informationen zum Seminar zuzusenden – beispielsweise E-Mails mit dem Link zur Teams-Sitzung, in der das Seminar stattfindet, oder für die Aufzeichnung des Seminars.

Änderungen am Entity Data Model

Die Entitäten für die Seminare enthalten ein Datumsfeld. Damit dieses beim Speichern in der Datenbank korrekt verarbeitet werden kann, fügen wir der Methode **OnModelCreating** eine Anweisung hinzu, welche den Datentyp für die Eigenschaft **Seminartermin** auf **datetime2** festlegt:

```
Protected Overrides Sub OnModelCreating(modelBuilder As DbModelBuilder)
    MyBase.OnModelCreating(modelBuilder)
    ...
    modelBuilder.Entity(Of Seminar).
        Property(Function(s) s.Seminartermin).HasColumnType("datetime2").HasPrecision(0)
End Sub
```

Aufruf der Seminarübersicht über das Ribbon

Im Artikel **Seminarverwaltung II: Ribbon und Frame** (**www.datenbankentwickler.net/272**) haben wir bereits den Aufbau des Ribbons für diese Anwendung beschrieben. Was noch fehlt, sind die Ereignismethoden, die durch die Schaltflächen **btnSeminaruebersicht** und **btnSeminaranlegen** ausgelöst werden. Mit diesen Schaltflächen wollen wir die noch zu erstellenden Seiten **SeminareUebersicht** und **Seminardetails** aufrufen. Die Methode, die durch die Schaltfläche **btnSeminaruebersicht** ausgelöst werden soll, erstellt eine neue Seite auf Basis des **Page**-Elements **SeminareUebersicht** und übergibt dieser einen Verweis auf das **Frame**-Objekt und auf den Datenbankkontext aus **dbContext**. Dann weist sie der Eigenschaft **Content** des **Frame**-Elements das neue **Page**-Element zu:

```
Private Sub btnSeminaruebersicht_Click(sender As Object, e As RoutedEventArgs)
    Dim pgeSeminaruebersicht As New SeminareUebersicht(fra, dbContext)
    fra.Content = pgeSeminaruebersicht
End Sub
```

Aufruf der Seite zum Anlegen eines neuen Seminars über das Ribbon

Ähnlich sieht der Aufruf für die Schaltfläche **btnSeminarAnlegen** aus:

```
Private Sub btnSeminarAnlegen_Click(sender As Object, e As RoutedEventArgs)
    Dim pgeSeminardetails As New Seminardetails(fra, dbContext)
    fra.Content = pgeSeminardetails
End Sub
```

Validierung der Seminare bei der Eingabe

Bevor wir die Benutzeroberfläche und die Anwendungslogik für das Anlegen neuer Seminare und das Bearbeiten vorhandener Seminare kreieren, benötigen wir noch eine Klasse, welche die Validierungsregeln für die **Seminar**-Elemente enthält. Dafür legen wir eine Klasse namens **Seminar.vb** an, in der wir die Schnittstelle **IDataErrorInfo** implementieren. Diese enthält die beiden Eigenschaften **Item** und **Error**. Für die Eigenschaft **Item** hinterlegen wir die Validierungsregeln, die Eigenschaft Error brauchen wir nicht anzupassen.

Die beiden Felder **Titel** und **Inhalt** dürfen einfach nicht leer sein, was wir durch die Prüfung des Inhalts dieser Felder mit **String.IsNullOrEmpty** prüfen. Der Inhalt des Feldes **Preis** darf nicht leer sein und weder einen negativen noch einen nicht-numerischen Wert enthalten. Das Feld **Seminartermin** darf ebenfalls nicht leer sein und muss ein Datum enthalten:

```
Imports System.ComponentModel

Public Class Seminar
    Implements IDataErrorInfo
    Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
        Get
```

```
            Dim strErrorMessage As String = ""
            Select Case columnName
                Case "Titel"
                    If (String.IsNullOrEmpty(Titel)) Then
                        strErrorMessage = "Bitte geben Sie einen Titel ein."
                    End If
                Case "Inhalt"
                    If (String.IsNullOrEmpty(Inhalt)) Then
                        strErrorMessage = "Bitte geben Sie einen Inhalt ein."
                    End If
                Case "Preis"
                    If (Preis < 0 Or String.IsNullOrEmpty(Preis)) Then
                        strErrorMessage = "Bitte geben Sie einen Preis ein."
                    End If
                Case "Seminartermin"
                    If (String.IsNullOrEmpty(Seminartermin) Or Not IsDate(Seminartermin)) Then
                        strErrorMessage = "Bitte geben Sie einen Seminartermin ein."
                    End If
            End Select
            Return strErrorMessage
        End Get
    End Property

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

Detailseite zum Anlegen und Bearbeiten von Seminaren

Damit gehen wir gleich über zu der Seite, mit der wir neue Seminare anlegen und vorhandene Seminare bearbeiten wollen. Im Teil mit den **Resources** für das **Page**-Element definieren wir wieder einige allgemeine Eigenschaften für die Steuerelemente wie **Margin** und **Padding** für die Elemente **DatePicker** und **Button**. Für das Element **TextBox** legen wir außerdem noch einen Wert für das Attribut **Validation.ErrorTemplate** fest, mit dem wir das Aussehen der **TextBox**-Elemente für den Fall definieren, dass der Inhalt nicht erfolgreich validiert werden konnte. Diese Teile finden Sie im Beispielprojekt, wir haben sie an dieser Stelle aus Platzgründen ausgespart:

```
<Page x:Class="Seminardetails"... Title="Seminardetails">
    <Page.Resources>
        ...
```

Ein weiteres **Style**-Element definiert die Regeln, nach denen **Button**-Elemente, die diese Regel aufnehmen, deaktiviert werden sollen. Diese Regeln richten sich nach der Validierung, genauer danach, ob die Validierung für ein bestimmtes Steuerelement fehlgeschlagen ist.

```
    <Style x:Key="EnableOnValidation" TargetType="{x:Type Button}">
        <Style.Triggers>
            <DataTrigger Binding="{Binding ElementName=txtTitel, Path=(Validation.HasError)}" Value="True">
                <Setter Property="IsEnabled" Value="False"></Setter>
            </DataTrigger>
            <DataTrigger Binding="{Binding ElementName=txtInhalt, Path=(Validation.HasError)}" Value="True">
                <Setter Property="IsEnabled" Value="False"></Setter>
            </DataTrigger>
            <DataTrigger Binding="{Binding ElementName=txtPreis, Path=(Validation.HasError)}" Value="True">
                <Setter Property="IsEnabled" Value="False"></Setter>
            </DataTrigger>
            <DataTrigger Binding="{Binding ElementName=txtSeminartermin, Path=(Validation.HasError)}"
                    Value="True">
                <Setter Property="IsEnabled" Value="False"></Setter>
            </DataTrigger>
        </Style.Triggers>
    </Style>
</Page.Resources>
```

Anschließend folgt die Definition für die **RowDefinition**- und die **ColumnDefinition**-Elemente für das **Grid** dieses **Page**-Elements, die wir hier ebenfalls nicht abbilden.

Steuerelemente der Seite Seminardetails.xaml

Interessanter wird es wieder bei der Definition der Steuerelemente der Seite **Seminardetails.xaml**, die wie in Bild 1 aussehen sollen. Hier finden wir zwei Spalten, von denen die erste jeweils das **Label**-Element aufnimmt und die zweite die eigentlichen gebundenen Steuerelemente.

Damit die Steuerelemente die Werte des jeweiligen Datensatzes beziehungsweise des jeweiligen Elements des Entity Data Models anzeigen, werden wir im Anschluss das Code behind-Modul mit einer Eigenschaft

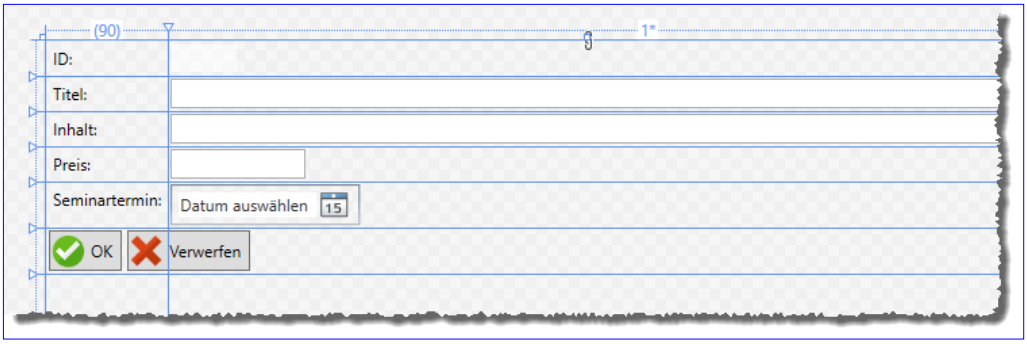


Bild 1: Entwurf der Seite zur Anzeige der Seminardetails

namens **AktuellesSeminar** füllen, welches wir beim Anlegen eines neuen Seminars oder beim Bearbeiten eines vorhandenen Seminars jeweils mit dem entsprechenden Element füllen.

Für die zu bindende Eigenschaft geben wir daher einen Ausdruck wie **{Binding AktuellesSeminar.ID, Mode=TwoWay}** an – hier für das Binden an das Feld **ID**. Für die Elemente mit Validierung definieren wir zusätzlich noch das Attribut **ValidatesOnDataErrors** wie in **{Binding AktuellesSeminar.Titel, Mode=TwoWay, ValidatesOnDataErrors=True}**.

Die ID bilden wir in einem **TextBox**-Element ab, das wir mit dem Wert **False** für das Attribut **IsEnabled** für die Eingabe deaktivieren. Die Felder **Titel** und **Inhalt** werden jeweils mit **TextBox**-Elementen dargestellt, die über das Attribut **Text** an diese Felder gebunden sind.

Für das Feld **Preis** legen wir noch zur Anzeige des Wertes als Währung noch ein spezielles Format fest, nämlich **StringFormat**: **{Binding AktuellesSeminar.Preis, Mode=TwoWay, ValidatesOnDataErrors=True, StringFormat=C}**.

Das Datum zeigen wir mit einem **DatePicker**-Steuerelement an, die im Feld **Seminartermin** enthaltene Uhrzeit liefern wir mit einer separaten TextBox, die wir ebenfalls an das Feld **Seminartermin** binden. Allerdings legen wir hier mit **StringFormat='t'** fest, dass nur die Uhrzeit angezeigt wird.

Die gebundenen Felder definieren wir insgesamt wie folgt:

```
<Grid>
    ...
    <Label Content="ID:" Grid.Column="0" />
    <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Text="{Binding
        AktuellesSeminar.ID, Mode=TwoWay}" Width="50" IsEnabled="False" BorderBrush="Transparent" />
    <Label Content="Titel:" Grid.Column="0" Grid.Row="1" />
    <TextBox x:Name="txtTitel" Grid.Column="1" Grid.Row="1" HorizontalAlignment="Stretch"
        Text="{Binding AktuellesSeminar.Titel, Mode=TwoWay, ValidatesOnDataErrors=True}" />
    <Label Content="Inhalt:" Grid.Row="2" />
    <TextBox x:Name="txtInhalt" Grid.Column="1" Grid.Row="2" HorizontalAlignment="Stretch"
        Text="{Binding AktuellesSeminar.Inhalt, Mode=TwoWay, ValidatesOnDataErrors=True}" />
    <Label Content="Preis:" Grid.Row="3" />
    <TextBox x:Name="txtPreis" Grid.Column="1" Grid.Row="3" Width="100" Text="{Binding
        AktuellesSeminar.Preis, Mode=TwoWay, ValidatesOnDataErrors=True, StringFormat=C}" />
    <Label Content="Seminartermin:" Grid.Row="4" />
    <DatePicker x:Name="dpSeminartermin" Grid.Column="1" Grid.Row="4" Width="140"
        HorizontalAlignment="left" SelectedDate="{Binding AktuellesSeminar.Seminartermin, Mode=TwoWay,
        ValidatesOnDataErrors=True, TargetNullValue=''}"></DatePicker>
   <Label Content="Uhrzeit:" Grid.Row="5" />
```

```
        <TextBox x:Name="txtUhrzeit" Grid.Column="1" Grid.Row="5" Width="140" HorizontalAlignment="left"
            Text="{Binding AktuellesSeminar.Seminartermin, Mode=TwoWay, ValidatesOnDataErrors=True, Tar-
            getNullValue='', StringFormat='t'}"></TextBox>
```

Schaltflächen zum Speichern und zum Verwerfen eines Seminars

Damit fehlen noch die beiden Schaltflächen, die wir in einem **StackPanel**-Element nebeneinander anordnen. Die Schaltfläche **btnSpeichern** soll nur aktiviert werden, wenn die in der weiter oben definierten statischen Ressource **EnableOnValidation** gestellten Bedingungen erfüllt sind.

Deshalb legen wir im **Style**-Element der Schaltfläche mit dem **BasedOn** fest, das die dortigen Bedingungen für den Wert des Attributs **Enabled** geprüft werden sollen. Die beiden Schaltflächen enthalten wiederum jeweils ein **StackPanel**-Element, welches nebeneinander das Icon und den Schaltflächentext beinhaltet:

```
        <StackPanel Orientation="Horizontal" Grid.Row="6" Grid.ColumnSpan="4" >
            <Button x:Name="btnSpeichern" Click="btnSpeichern_Click">
                <Button.Style>
                    <Style TargetType="{x:Type Button}" BasedOn="{StaticResource EnableOnValidation}">
                        <Style.Setters>
                            <Setter Property="Margin" Value="2"></Setter>
                        </Style.Setters>
                    </Style>
                </Button.Style>
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/ok.png" Width="24" Height="24"></Image>
                    <Label>OK</Label>
                </StackPanel>
            </Button>
            <Button x:Name="btnVerwerfen" Click="btnVerwerfen_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/delete.png" Height="24" Width="24"></Image>
                    <Label>Verwerfen</Label>
                </StackPanel>
            </Button>
        </StackPanel>
    </Grid>
</Page>
```

Anwendungslogik in der Datei Seminarverwaltung.xaml.vb

Bevor wir die Seite testen können, benötigen wir noch die Anwendungslogik hinter der Seite. Dazu hinterlegen wir in der Datei **Seminarverwaltung.xaml.vb** zunächst zwei **Namespace**-Importe:

```
Imports System.Collections.ObjectModel
Imports System.Data.Entity
```

Die Klasse Seminardetails deklariert dann Variablen für den Datenbankkontext, das aktuell dargestellte Seminar, das umgebende **Frame**-Element und die Auflistung aller Seminare. Außerdem eine Variable namens **bolVonUebersicht**, die angibt, ob die Seite von der Übersichtsseite aus geöffnet wurde:

```
Class Seminardetails
    Private dbContext As SeminarverwaltungContext
    Public Property AktuellesSeminar As Seminar
    Private fra As Frame
    Private Seminare As ObservableCollection(Of Seminar)
    Private bolVonUebersicht As Boolean
```

Außerdem deklarieren wir ein öffentliches Ereignis, das ausgelöst wird, wenn das Seminar geändert wurde:

```
    Public Event SeminarChanged(ByVal sender As Object, ByVal e As EventArgs)
```

Konstruktoren von Seminardetails.xaml.vb

Die Seite Seminardetails erhält zwei Konstruktoren. Der erste erwartet neben dem Verweis auf das umschließende **Frame**-Objekt und dem Datenbankkontext gegebenenfalls noch ein **ObservableCollection**-Objekt mit den Seminaren der Anwendung. Dieser Konstruktor wird verwendet, wenn mit der Seite ein neues **Seminar**-Element angelegt werden soll. Die Konstruktormethode initialisiert die Seite und weist der Variablen **fra** das übergebene **Frame**-Objekt zu. Dann erstellt sie ein neues **Seminar**-Element und referenziert es mit der Variablen **AktuellesSeminar**. Die Auflistung **objSeminare** enthält bereits eine ObservableCollection, wenn die Seite von der Übersichtsseite aufgerufen wurde, die wir im Anschluss besprechen. Anderenfalls wird die **ObservableCollection** erstellt und mit den Elementen der Auflistung **Seminare** gefüllt. Das Element **AktuellesSeminar** wird der Auflistung **Seminare** hinzugefügt und auch der entsprechenden **DbSet**-Auflistung des Datenbankkontextes. Schließlich stellt der Konstruktor den **DataContext** der Seite auf das Code behind-Modul selbst ein:

```
    Public Sub New(objFrame As Frame, Optional dbC As SeminarverwaltungContext = Nothing, _
            Optional objSeminare As ObservableCollection(Of Seminar) = Nothing)
        InitializeComponent()
        fra = objFrame
        AktuellesSeminar = New Seminar
        dbContext = dbC
        If objSeminare Is Nothing Then
            Seminare = New ObservableCollection(Of Seminar)(dbContext.Seminare)
        Else
            Seminare = objSeminare
        End If
```