HANSER

Leseprobe

zu

„Visual C# 2019“

von Jürgen Kotz

Print-ISBN: 978-3-446-45802-4
E-Book-ISBN: 978-3-446-46099-7
E-Pub-ISBN: 978-3-446-46253-3

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-45802-4
sowie im Buchhandel

Inhalt

Vorwort

C# ist die momentan von Microsoft propagierte Sprache, sie bietet die Möglichkeiten und Flexibilität von C++ und erlaubt trotzdem eine schnelle und unkomplizierte Programmierpraxis wie einst bei Visual Basic. C# ist (fast) genauso mächtig wie C++, wurde aber komplett neu auf objektorientierter Basis geschrieben.

Damit ist C# das ideale Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework, beginnend bei Windows Forms- über WPF-, ASP.NET- und mobilen Anwendungen (mittlerweile auch für Android und iOS) bis hin zu systemnahen Applikationen.

Das vorliegende Buch ist ein Angebot für künftige wie auch für fortgeschrittene C#-Programmierer. Seine Philosophie knüpft an die vielen anderen Titel an, die in dieser Reihe in den vergangenen zwanzig Jahren zu verschiedenen Programmiersprachen erschienen sind:

- Programmieren lernt man nicht durch lineares Durcharbeiten eines Lehrbuchs, sondern nur durch unermüdliches Ausprobieren von Beispielen, verbunden mit ständigem Nachschlagen in der Referenz.
- Der Umfang einer modernen Sprache wie C# in Verbindung mit Visual Studio ist so gewaltig, dass ein seriöses Programmierbuch das Prinzip der Vollständigkeit aufgeben muss und nach dem Prinzip „so viel wie nötig“ sich lediglich eine „Initialisierungsfunktion“ auf die Fahnen schreiben kann.

Gegenüber anderen Büchern zur gleichen oder ähnlichen Thematik nimmt dieses Buch für sich in Anspruch, gleichzeitig Lehr- und Übungsbuch zu sein.

Zum Buchinhalt

Wie Sie bereits dem Buchtitel entnehmen können, wagt das vorliegende Werk den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in C# 7.3 zu liefern oder all die Informationen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation (MSDN) ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die **F1**-Taste trennt.

- Für den *Einsteiger* möchte ich den einzig vernünftigen und gangbaren Weg beschreiten, nämlich nach dem Prinzip „so viel wie nötig“ eine schmale Schneise durch den Urwald der .NET-Programmierung mit C# schlagen, bis er eine Lichtung erreicht hat, die ihm erste Erfolgserlebnisse vermittelt.

- Für den *Profi* möchte ich in diesem Buch eine Vielzahl von Informationen und Know-how bereitstellen, wonach er bisher in den mitgelieferten Dokumentationen, im Internet bzw. in anderen Büchern vergeblich gesucht hat.

Die Kapitel des Buches und die Onlinekapitel sind in fünf Themenkomplexe gruppiert:

1. Grundlagen der Programmierung mit C# (Buch)
2. WPF-Anwendungen (Buch)
3. Technologien (Buch)
4. Windows Forms-Anwendungen (online)
5. Bonuskapitel (online)

Die Kapitel innerhalb eines Teils bilden einerseits eine logische Aufeinanderfolge, können andererseits aber auch quergelesen werden. Im Praxisteil eines jeden Kapitels werden anhand realer Problemstellungen die behandelten Programmiertechniken im Zusammenhang demonstriert.

Zu den Codebeispielen

Auf vielfachen Wunsch habe ich mich dazu entschlossen, ab dieser Auflage des Buches die Beispiele auch mit WPF zu erstellen. Der Einfachheit halber werden im ersten Teil die Beispiele weiter mit Windows Forms erstellt. Der zweite Teil des Buches behandelt dann ausführlich WPF und im dritten Teil „Technologien“ werden die Beispiele dann mit WPF dargestellt. Alle Beispieldaten dieses Buches und das Bonuskapitel zu Windows Forms können Sie unter der folgenden Adresse herunterladen:

www.hanser-fachbuch.de

Geben Sie bitte im Suchfeld „Visual C# 2019“ ein und klicken Sie auf der Buchdetailseite auf die Registerkarte *Extras*.

Beim Nachvollziehen der Buchbeispiele beachten Sie bitte Folgendes:

- Kopieren Sie die Buchbeispiele auf die Festplatte. Wenn Sie auf die Projektmappendatei (**.sln*) klicken, wird Visual Studio in der Regel automatisch geöffnet und das jeweilige Beispiel wird in die Entwicklungsumgebung geladen, wo Sie es z. B. mittels **F5**-Taste kompilieren und starten können.
- Für einige Beispiele ist ein installierter Microsoft SQL Server Express LocalDB oder jegliche andere Instanz eines SQL-Servers erforderlich.
- Beachten Sie die zu einigen Beispielen beigefügten *Liesmich.txt*-Dateien, die Sie auf besondere Probleme hinweisen.

Nobody is perfect

Sie werden in diesem Buch nicht alles finden, was C# bzw. das .NET Framework 4.7.2 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel noch besser oder ausführlicher beschrieben. Aber Sie halten mit diesem Buch einen überschaubaren Breitband-Mix in den Händen, der sowohl vertikal vom Einsteiger bis zum Profi als auch horizontal von den einfachen Sprachelementen bis hin zu komplexen Anwendungen jedem etwas bietet, ohne dabei den Blick auf das Wesentliche im .NET-Dschungel zu verlieren.

Wenn Sie Vorschläge oder Fragen zum Buch haben, können Sie mich gerne kontaktieren:

juergen.kotz@primetime-software.de

Ich hoffe, dass ich Ihnen mit diesem Buch einen nützlichen Begleiter bei der .NET-Programmierung zur Seite gestellt habe, der es verdient, seinen Platz nicht im Regal, sondern griffbereit neben dem Computer einzunehmen.

Jürgen Kotz

München, im Sommer 2019

11 WPF-Datenbindung

Nachdem wir schon an der einen oder anderen Stelle auf Datenbindung zurückgegriffen haben, wollen wir uns jetzt direkt mit dieser Thematik beschäftigen.

Im Unterschied zu den Windows-Forms-Anwendungen sind Sie bei der Datenbindung nicht auf spezielle Controls oder Eigenschaften angewiesen. In einer WPF-Anwendung kann fast jede Eigenschaft (Abhängigkeitseigenschaft) an andere Eigenschaften gebunden werden.

Als Datenquelle können Sie beispielsweise

- Eigenschaften anderer WPF-Controls (Elemente),
- Ressourcen,
- XML-Elemente oder
- beliebige Objekte (auch ADO.NET-Objekte, z. B. *DataTable*)

verwenden.

11.1 Grundprinzip

Zunächst wollen wir Ihnen das Grundprinzip der Datenbindung in WPF an einem recht einfachen Beispiel demonstrieren.

Beispiel 11.1: Datenbindung zwischen *Slider* und *ProgressBar*

XAML

Fügen Sie in ein *Window,* eine *ProgressBar* und einen *Slider* ein. Mit dem *Slider* soll der aktuelle Wert der *ProgressBar* direkt und ohne zusätzlichen Quellcode verändert werden.

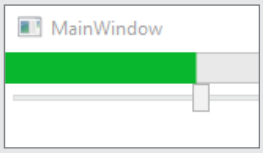
```
<StackPanel>
```

Hier sehen Sie auch schon den Ablauf: Das Ziel (*ProgressBar*) bindet seine Eigenschaft *Value* an die Quelle (*Slider*) mit deren Eigenschaft *Value*.

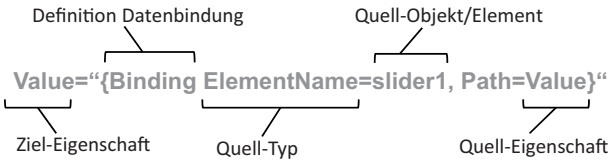
```
  <ProgressBar Height="20" Name="progressBar1" Maximum="100"
               Value="{Binding ElementName=slider1, Path=Value}"/>
 Separator Height="10"/>
  <Slider Name="slider1" Maximum="100" />
</StackPanel>
```

Ergebnis

Zur Laufzeit können Sie den *Slider* beliebig verändern, die *ProgressBar* passt sofort ihren Wert an:



Sehen wir uns noch einmal die Syntax im Detail an:



HINWEIS: Kann die Quelleigenschaft nicht automatisch in den Datentyp der Zieleigenschaft konvertiert werden, können Sie zusätzlich einen Typkonverter angeben (siehe dazu Abschnitt 11.6.1, „IValueConverter“).

11.1.1 Bindungsarten

Das vorhergehende Beispiel zeigte bereits recht eindrucksvoll, wie einfach sich Eigenschaften verschiedener Objekte miteinander verknüpfen lassen. Doch das ist noch nicht alles. Über ein zusätzliches Attribut *Mode* lässt sich auch bestimmen, in welche Richtungen die Bindung aktiv ist, d. h. ob die Werte nur von der Quelle zum Ziel oder auch umgekehrt übertragen werden. Die folgende Tabelle zeigt die möglichen Varianten:

Typ	Beschreibung
OneTime	Mit der Initialisierung wird der Wert einmalig von der Quelle zum Ziel kopiert. Danach wird die Bindung aufgehoben.
OneWay	Der Wert wird nur von der Quelle zum Ziel übertragen (readonly). Ändert sich der Wert des Ziels, wird die Bindung aufgehoben.
OneWayToSource	Der Wert wird vom Ziel zur Quelle übertragen (writeonly). Ändert sich der Wert der Quelle, bleibt die Bindung erhalten, eine Wertübertragung findet jedoch nicht statt.
TwoWay	(meist Defaultwert[1]) Werte werden zwischen Quelle und Ziel in beiden Richtungen übertragen.

[1] Bei Bindung an eine *ItemsSource* wird per Default OneWay-Binding verwendet.

Beispiel 11.2: Testen der verschiedenen Bindungsarten

XAML

```
...
   <Grid Margin="10">
      <Grid.ColumnDefinitions>
          <ColumnDefinition></ColumnDefinition>
          <ColumnDefinition></ColumnDefinition>
          <ColumnDefinition></ColumnDefinition>
      </Grid.ColumnDefinitions>
      <StackPanel>
          <Label Height="30">One Time</Label>
          <Label Height="30">One Way</Label>
          <Label Height="30">One Way To Source</Label>
          <Label Height="30">Two Way</Label>
      </StackPanel>
      <StackPanel Grid.Column="1">
          <Slider Name="sl1" Maximum="100" Height="30"/>
          <Slider Name="sl3" Maximum="100" Height="30"/>
          <Slider Name="sl5" Maximum="100" Height="30"/>
          <Slider Name="sl7" Maximum="100" Height="30"/>
      </StackPanel>
      <StackPanel Grid.Column="2">
          <Slider Name="sl2" Maximum="100" Height="30"
           Value="{Binding ElementName=sl1, Path=Value, Mode=OneTime}"/>
          <Slider Name="sl4" Maximum="100" Height="30"
           Value="{Binding ElementName=sl3, Path=Value, Mode=OneWay}"/>
          <Slider Name="sl6" Maximum="100" Height="30"
       Value="{Binding ElementName=sl5, Path=Value, Mode=OneWayToSource}"/>
          <Slider Name="sl8" Maximum="100" Height="30"
           Value="{Binding ElementName=sl7, Path=Value, Mode=TwoWay}"/>
      </StackPanel>

     </Grid>
```

Ergebnis

Verschieben Sie ruhig einmal die *Slider* im Testprogramm. Jeweils der linke und der rechte *Slider* bilden eine Datenbindung und sollten auch das entsprechende Verhalten zeigen:

11.1.2 Wann eigentlich wird die Quelle aktualisiert?

Im obigen Beispiel scheint alles ganz einfach zu sein, Sie ziehen an einem Schieberegler und der andere bewegt sich mit (oder auch nicht, wie bei *OneTime*). Doch was ist, wenn Sie beispielsweise eine *TextBox* in einer Datenbindung verwenden? Hier stellt sich die Frage, **wann** der „gewünschte" Wert wirklich in der *TextBox* steht.

Eine eingegebene Ziffer ist vielleicht nicht der richtige Wert, sie kann aber schon als gültiger Inhalt interpretiert werden. Nicht in jedem Fall möchte man deshalb sofort einen Datenaustausch zwischen Ziel und Quelle zulassen (bei *TwoWay* oder *OneWayToSource*).

Über das optionale Attribut *UpdateSourceTrigger* haben Sie direkten Einfluss darauf, wann die Aktualisierung **der Quelle** durchgeführt wird. Vier Varianten bieten sich dabei an:

- *Default*
  Meist wird das *PropertyChanged*-Ereignis für die Datenübernahme genutzt, bei einigen Controls kann es auch *LostFocus* sein.
- *Explicit*
  Die Datenübernahme muss „manuell" per *UpdateSource*-Methode ausgelöst werden.
- *LostFocus*
  Die Datenübernahme erfolgt bei Fokusverlust des Ziels.
- *PropertyChanged*
  Die Datenübernahme erfolgt mit jeder Werteänderung. Dies kann bei komplexeren Abläufen zu Problemen führen, da der Abgleich, z. B. bei einem Schieberegler/einer Scrollbar, recht häufig vorgenommen wird.

Beispiel 11.3: Explizite Datenübernahme nur per **Enter**-Taste

XAML

```
<StackPanel>
  <TextBox Name="txt1">Hallo</TextBox>
  <TextBox Name="txt2"
      Text="{Binding ElementName=txt1, Path=Text,
             UpdateSourceTrigger=Explicit}"
      KeyDown="TextBox_KeyDown"/>
</StackPanel>
```

C#

```
private void TextBox_KeyDown(object sender, KeyEventArgs e)
{
   if (e.Key== Key.Enter)
   {
      txt2.GetBindingExpression(TextBox.TextProperty).UpdateSource();
   }
}
```

HINWEIS: Da die Bindung im XAML-Code vorgenommen wurde, müssen wir im C#-Code erst mit *GetBindingExpression* das *BindingExpression*-Objekt abrufen, um die *UpdateSource*-Methode aufzurufen.

11.1.3 Geht es auch etwas langsamer?

Am obigen Beispiel konnten Sie es ja schon beobachten: Sie verschieben den *Slider* und der zweite *Slider* reagiert sofort. So weit, so gut. Was aber, wenn Sie erst nach einiger Zeit auf die Veränderung reagieren wollen?

Hier hilft die mit WPF 4.5 eingeführte Eigenschaft *Delay* weiter. Diese verzögert die Datenübergabe um den angegebenen Wert (Millisekunden). Das heißt, erst wenn die Zeit nach einer Änderung verstrichen ist, wird der Wert weitergegeben. Jede Änderung in dieser Zeitspanne setzt den internen Timer zurück und lässt die Zeit erneut laufen. Bewegen Sie also den *Slider* dauernd hin und her, passiert nichts, erst nach der letzten Bewegung und dem Ablaufen der Zeit wird auch die Änderung berücksichtigt.

Beispiel 11.4: Zeitverzögerung bei Datenbindung

XAML

```
...
    <Slider Name="sl0" Maximum="100" Height="30"
            Value="{Binding ElementName=sl9,
            Path=Value, Mode=TwoWay, Delay=500}"/>
...
```

Was dem einen oder anderen als Spielerei vorkommen mag, ist ein fast unverzichtbares Feature im Zusammenhang mit größeren Datenmengen oder langsamen Datenverbindungen. Folgende Szenarien sind denkbar:

- **1:n-Beziehung**
  Änderungen in einer *ListBox* sollen sich nicht sofort auf die Detaildaten auswirken, sondern erst nachdem sich der Anwender für einen Datensatz final entschieden hat (Scrollen per Tastatur durch die Liste). Andernfalls kann es schnell zum Ruckeln oder Springen zwischen den Datensätzen kommen.
- **Texteingaben**
  Nutzen Sie laufende Eingaben als Filter oder Suchwert, kann gerade bei großen Ergebnismengen eine Verzögerung bei der Eingabe auftreten (ein kurzer Filter mit Platzhalter hat meist große Ergebnismengen zur Folge).
- **Anzeige großer Datenmengen**
  Mit der Auswahl in einer Liste soll eine größere Grafik angezeigt werden. Jede Änderung, z. B. beim Scrollen, führt im Normalfall zum Laden der Grafik. Hier ist eine entsprechende Verzögerung sinnvoll.

Alle obigen Fälle lassen sich natürlich auch mit einem eigenen *Timer* realisieren, aber warum kompliziert, wenn es jetzt auch wesentlich einfacher geht?

Eine Einschränkung sollten Sie allerdings beachten, auch wenn sie meist nicht von Bedeutung ist:

HINWEIS: Die Verzögerung gilt nur für eine Richtung der Datenbindung, d. h. nur für das Control, dem auch die Verzögerung zugeordnet ist. Ziehen Sie deswegen im obigen Beispiel auch einmal den rechten Slider, der linke reagiert sofort darauf.

11.1.4 Bindung zur Laufzeit realisieren

Nicht immer werden Sie mit den schon zur Entwurfszeit definierten Datenbindungen auskommen. Es ist aber auch kein Problem, die Datenbindung erst zur Laufzeit per C#-Code zu realisieren. Alles, was Sie dazu benötigen, ist ein *Binding*-Objekt, dessen Konstruktor Sie bereits den *BindingPath* zuweisen können. Legen Sie anschließend noch die *BindingSource* sowie gegebenenfalls den *Mode* (z. B. *OneWay*) fest. Letzter Schritt ist das eigentliche Binden mit der *SetBinding*-Methode des jeweiligen Controls.

Beispiel 11.5: Bindung zur Laufzeit realisieren

XAML

Unsere Testoberfläche:

```
<Window x:Class="DatenbindungStart.MainWindow"
...
        Title="MainWindow" Height="300" Width="500" Loaded="Window_Loaded">
...
    <StackPanel>
        <Label Name="Label1"></Label>
        <Button "Button1" Click="Button_Click">Test</Button>
    </StackPanel>
</Window>
```

C#

Mit dem Laden des Fensters erzeugen wir die Bindung wie oben beschrieben:

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

Wir binden an einen Button:

```
            Binding binding = new Binding("Content");
            binding.Source = button1;
            binding.Mode = BindingMode.OneWay;
```

Binden an den Content:

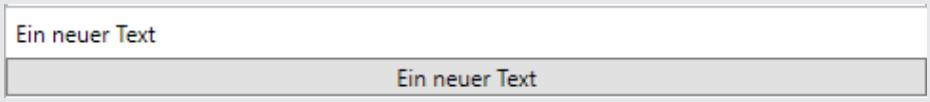
```
            label1.SetBinding(Label.ContentProperty, binding);
        }
```

Und hier verändern wir die Beschriftung des Buttons:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            button1.Content = "Ein neuer Text";
        }
    }
```

Test

Nach dem Start dürfte im Label zunächst „Test“ stehen, die ursprüngliche Button-Beschriftung. Nach einem Klick auf die Schaltfläche ändern sich sowohl die Button-Beschriftung als auch die Label-Beschriftung.



Die Bindung selbst können Sie recht einfach wieder aufheben, indem Sie der Ziel-Eigenschaft der Bindung einen neuen Wert zuweisen.

Beispiel 11.6: Bindung zur Laufzeit aufheben

C#

Entweder so:

```
label1.Content = "Bindung beendet";
```

Oder so:

```
label1.ClearValue(Label.ContentProperty);
```

11.2 Binden an Objekte

Nachdem wir uns bereits mit dem Binden an Oberflächenelemente vertraut gemacht haben, wollen wir jetzt den Schritt hin zu selbstdefinierten Objekten gehen.

Prinzipiell bieten sich zwei Varianten der Instanziierung von Objekten an:

- Sie instanziieren die Objekte in XAML (in einem Ressource-Abschnitt).
- Sie instanziieren wie bisher die Objekte im Quellcode.

HINWEIS: Von der Möglichkeit, Objekte im XAML-Code zu instanziieren, ist abzuraten. Einerseits wird mit Klassen gearbeitet, die per Code definiert und verarbeitet werden, andererseits wird die Instanz in der Oberfläche, d. h. im XAML-Code, erzeugt. Das ist sicher nicht der Weisheit letzter Schluss. Gerade die üble Vermischung von Code und Oberfläche sollte eigentlich vermieden werden.

Fragwürdig werden Beispielprogramme dann, wenn im C#-Quellcode das zunächst in XAML erzeugte Objekt per *FindResource* gesucht wird (siehe folgender Abschnitt). Das pervertiert doch jede Form der sauberen Programmierung.

Wohlgemerkt wollen wir nicht die komplette Datenbindung im Code realisieren. Das ist sicher zu aufwendig und auch nicht notwendig. Doch aus Sicht des Entwicklers sollte nicht die Oberfläche (XAML), sondern der Code im Mittelpunkt des Programms stehen.

11.2.1 Objekte im XAML-Code instanziieren

Erster Schritt nach der Definition der Klasse ist das Importieren des entsprechenden Namespace in die XAML-Datei, andernfalls können Sie auch nicht darauf Bezug nehmen.

Beispiel 11.7: Import des aktuellen Namespace *Datenbindung* in die XAML-Datei

XAML

```
<Window x:Class="ObjectBinding.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:ObjectBinding"
...
```

Nachdem in XAML die entsprechende Klasse bekannt ist, kann diese auch verwendet werden, um eine eigene Instanz zu erzeugen.

Beispiel 11.8: Erzeugen der Instanz im XAML-Code (wir nutzen eine Klasse *Person*)

C#

```
public class Person
{
    public string Nachname { get; set; }
    public string Vorname { get; set; }
}
```

XAML

```
...
 <Window.Resources>
   <local:Person x:Key="pers" Nachname="Mooshammer" Vorname="Elisabeth" />
 </Window.Resources>
```

Die Werte im Einzelnen:

- *local*: Der Bezug auf den Namespace-Alias
- *Person*: Der Klassenname
- *x:Key*: Der Schlüssel, unter dem die Instanz verwendet werden kann
- *Nachname*, *Vorname*: Das Setzen einzelner Eigenschaften für die Instanz von *Person*

Letzter Schritt: Wir nutzen die Möglichkeiten der Datenbindung und binden zwei *TextBox*en an die Eigenschaften *Nachname* und *Vorname*.

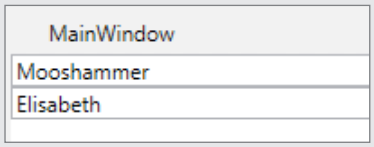
Beispiel 11.9: Bindung an das neue Objekt erzeugen

XAML

```
 <StackPanel>
   <TextBox Text="{Binding Source={StaticResource pers}, Path=Nachname}" />
   <TextBox Text="{Binding Source={StaticResource pers}, Path=Vorname}" />
 </StackPanel>
```

Ergebnis

Schon zur Entwurfszeit dürfte in den beiden *TextBox*en der gewünschte Inhalt auftauchen:



Wem das zu viel Schreibarbeit ist, der kann mit dem *DataContext* auch eine alternative Variante der Zuweisung nutzen. Diese Eigenschaft bietet zunächst eine Alternative zur Zuweisung von *Source*, hat jedoch zusätzlich die Fähigkeit, von übergeordneten auf untergeordnete Elemente vererbt zu werden. Damit können Sie beispielsweise einem *Panel* oder sogar dem gesamten *Window* einen *DataContext* zuweisen und diesen in allen enthaltenen Elementen nutzen.

Beispiel 11.10: Vereinfachung durch Verwendung eines *DataContext*

XAML

```
<StackPanel DataContext="{StaticResource pers}">
  <TextBox Text="{Binding Path=Nachname}" />
  <TextBox Text="{Binding Path=Vorname}" />
...
```

Sie sparen sich so die Angabe von *Source* bei jedem einzelnen Element.

11.2.2 Verwenden der Instanz im C#-Quellcode

Sicher nicht ganz abwegig ist der Wunsch, zur Laufzeit per C#-Code auch mit dem Objekt zu arbeiten, um z. B. die Werte mit einer *MessageBox* anzuzeigen.

Hier wird die Programmierung dann schon recht windig, müssen Sie doch zunächst die entsprechende Ressource des *Window* suchen und typisieren; möglichen Fehlern ist Tür und Tor geöffnet.

Beispiel 11.11: Anzeige der Werte eines per XAML instanziierten Objekts

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Person pers = (Person)FindResource(«pers»);
    MessageBox.Show($»{pers.Nachname}, {pers.Vorname}»);
}
```

Aus Sicht eines Programmierers sieht das doch ziemlich merkwürdig aus, auch wenn sich hier der XAML-Profi freut, dass er sogar eine Instanz plus Wertzuweisung per XAML-Code realisiert hat.

Doch was passiert eigentlich mit der Datenbindung, wenn wir der Instanz ein paar neue Werte zuweisen? Ein Test ist schnell realisiert:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Person pers = (Person)FindResource("pers");
    pers.Nachname = "Eberhofer";
}
```

Der nachfolgende Blick auf die Oberfläche dürfte in den meisten Fällen für Ernüchterung sorgen: Haben Sie Ihre .NET-Klasse (in diesem Fall *Person*) nicht entsprechend angepasst, passiert überhaupt nichts und in den *TextBox*en stehen nach wie vor die alten Werte.

11.2.3 Anforderungen an die Quell-Klasse

Was ist hier schiefgelaufen? Eigentlich nichts, die neuen Werte stehen wirklich im Objekt, sie werden aber nicht angezeigt, weil die darstellenden Elemente von einer Wertänderung nichts mitbekommen haben. Wir müssen diese quasi „wecken“, und was eignet sich dafür besser als ein Ereignis?

Auch hier gibt es bereits eine fertige Lösung:

HINWEIS: Implementieren Sie in Ihrer Klasse das Interface *INotifyPropertyChanged* (Namespace *System.ComponentModel*).

Beispiel 11.12: Unsere Klasse *Schüler* mit implementiertem *NotifyPropertyChanged*-Ereignis

C#

```
using System.ComponentModel;

namespace ObjectBinding
{
    public class Person : INotifyPropertyChanged
    {
        string nachname;
        string vorname;

        public event PropertyChangedEventHandler PropertyChanged;

        public string Vorname
        {
            get { return vorname; }
            set
            {
                vorname = value;
                NotifyPropertyChanged(nameof(Vorname));
            }
        }
```

```
        public string Nachname
        {
            get { return nachname; }
            set
            {
                nachname = value;
                NotifyPropertyChanged(nameof(Nachname));
            }
        }

        public override string ToString()
        {
            return $"{Nachname}, {Vorname}";
        }

        private void NotifyPropertyChanged(string info)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(info));
            }
        }
    }
}
```

HINWEIS: Alternativ können Sie natürlich auch Abhängigkeitseigenschaften definieren. Diese verfügen „ab Werk“ über die erforderliche Benachrichtigung an die gebundenen Elemente, sie erfordern aber einen höheren Programmieraufwand.

HINWEIS: Damit die Klasse auch im XAML-Code instanziiert werden kann, muss diese über einen parameterlosen Konstruktor verfügen.

11.2.4 Instanziieren von Objekten per C#-Code

Eigentlich könnten wir Ihnen an dieser Stelle noch weitere Möglichkeiten zeigen, wie Sie in XAML Objekte erzeugen bzw. zuweisen können, aber dies ist weder sinnvoll noch besonders übersichtlich. Wir wollen uns stattdessen mit der Vorgehensweise bei vorhandenen, d. h. per Code erzeugten, .NET-Objekten beschäftigen.

Zunächst bleiben wir bei unserem einfachen Beispiel mit der Instanz der Klasse *Person*.

Beispiel 11.13: Verwendung von instanziierten Objekten in XAML

C#

Zunächst die Instanziierung:

```
..
    public partial class MainWindow : Window
    {
        public Person Person { get; set;}

        public MainWindow()
        {
            InitializeComponent();
```

Instanz erzeugen und Werte zuweisen:

```
            Person = new Person()
                { Nachname = "Birkenberger", Vorname = "Rudi"};
```

Hier legen wir per C#-Code den *DataContext* fest:

```
            stackPanel1.DataContext = Person;
        }
```

Die spätere Abfrage des Objekts stellt jetzt überhaupt kein Problem dar, die Instanz liegt ja bereits vor:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show($"{Person.Nachname}, {Person.Vorname}");
        }
...
```

XAML

Der XAML-Code:

```
...
<StackPanel Name="stackPanel1">
   <TextBox Text="{Binding Path=Nachname}" />
   <TextBox Text="{Binding Path=Vorname}" />
   <Button Click="Button_Click">Name anzeigen</Button>
 </StackPanel>
</Window>
```

Der Vorteil dieser Vorgehensweise: Sie entscheiden, wie und wann die Instanz erzeugt wird, können vorher noch diverse Methoden aufrufen, profitieren von der Syntaxprüfung und haben einen lesbaren Code.

Der einzige Nachteil: Sie haben keine Wertanzeige zur Entwurfszeit. Im XAML-Code ist es nicht sofort erkennbar, welches Objekt zugeordnet wird. Dies ist allerdings auch gleich wieder der Vorteil, mit einem Klick können Sie einen neuen *DataContext* zuweisen und eine andere Instanz bearbeiten.

11.3 Binden von Collections

Die bisherigen Ausführungen dürften zwar schon das Potenzial der Datenbindung demonstriert haben, doch nach der Pflicht kommt jetzt die Kür, d. h. die Arbeit mit einer Reihe von Objekten (Collections). Diese sind vor allem dann interessant, wenn Sie Objekte von Datenbanken abrufen, um sie in Eingabedialogen oder gleich in Listenfeldern darzustellen. Ausgangspunkt können hier Geschäftsobjekte, LINQ-Abfragen etc. sein.

HINWEIS: Im vorliegenden Abschnitt werden wir uns zunächst auf eine „selbstgestrickte" Collection beziehen (wir verwenden das *Person*-Objekt aus dem vorhergehenden Abschnitt). Ab Abschnitt 11.5 geht es dann mit Datenbindung in Verbindung mit *LINQ to SQL*-Abfragen weiter.

11.3.1 Anforderung an die Collection

Wie auch bei der Klassendefinition für das einzelne Objekt, werden auch an die Collection einige Anforderungen gestellt. Zwar können die WPF-Elemente durch die Verwendung der *INotifyPropertyChanged*-Schnittstelle auf Änderungen einzelner Objekteigenschaften reagieren, das Hinzufügen oder Löschen von ganzen Objekten ist davon aber nicht betroffen. Aus diesem Grund bietet WPF auch hier ein genormtes Interface für die Rückmeldung an: *INotifyCollectionChanged*.

HINWEIS: Grundvoraussetzung für die Anzeige von Listen ist die Verwendung des *IEnumerable*-Interfaces.

Wollen Sie es sich leicht machen, können Sie direkt Objekte der Klasse *ObservableCollection* (Namespace *System.Collections.ObjectModel*) erzeugen.

Beispiel 11.14: *(Fortsetzung)* Erzeugen und Verwenden einer geeigneten Klasse für die Datenbindung von Collections

C#

```
using System.Collections.ObjectModel;
...
    public partial class MainWindow : Window
    {
```

Eine Collection von Personen:

```
        public ObservableCollection<Person> Abteilung { get; set; }
```

Im Konstruktor des *Window* erzeugen wir eine Instanz und füllen diese mit einigen Datensätzen:

```
        public MainWindow()
        {
            Abteilung = new ObservableCollection<Person>();
            Abteilung.Add(new Person()
                { Nachname = "Eberhofer", Vorname = "Franz" });
            Abteilung.Add(new Person()
                { Nachname = "Birkenberger", Vorname= "Rudi" });
            Abteilung.Add(new Person()
                { Nachname = "Simmerl", Vorname ="Max" });
            InitializeComponent();
```

Hier dürften Sie die Verbindung zur bisherigen Vorgehensweise sehen, die Collection wird als *DataContext* für das Fenster und damit für alle untergeordneten Elemente ausgewählt:

```
            DataContext = Abteilung;
        }
```

11.3.2 Einfache Anzeige

Damit können wir uns zunächst der einfachen Anzeige, z. B. in *TextBox*en, widmen.

Beispiel 11.15: *(Fortsetzung)* Binden von TextBoxen an die Collection

XAML

```
   <StackPanel Background="Aqua">
     <Label Content="Nachname:" />
```

Hier werden die *TextBox*en an die Eigenschaften der Collection bzw. an das aktive Objekt der Collection gebunden (dies ist durch eine View bestimmt, siehe ab Abschnitt 11.4):

```
     <TextBox Text="{Binding Path=Nachname}" />
     <Label Content="Vorname:" />
```

Beachten Sie auch diese mögliche Kurzsyntax, die auf die Angabe von *Path* verzichtet:

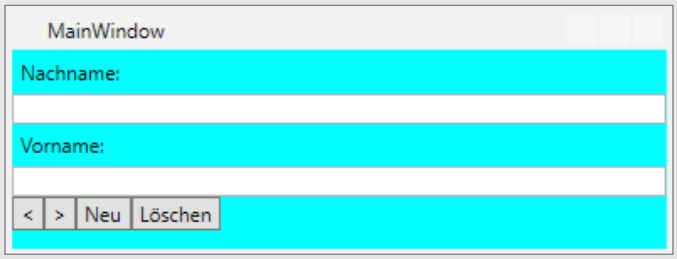
```
     <TextBox Text="{Binding Vorname}" />
```

Einige Schaltflächen definieren:

```
     <StackPanel Orientation="Horizontal">
       <Button Content=" &lt; " Click="ButtonPrevious_Click" />
       <Button Content=" &gt; " Click="ButtonNext_Click"/>
       <Button Content=" Neu " Click="ButtonNeu_Click"/>
       <Button Content=" Löschen " Click="ButtonLoeschen_Click"/>
     </StackPanel>
   </StackPanel>
```

Ergebnis

Das erzeugte Formular:



Nach dem Start dürfte schon etwas in den Textfeldern angezeigt werden, ein Navigieren zwischen den einzelnen Datensätzen (Objekten) ist allerdings noch nicht möglich.

11.3.3 Navigieren zwischen den Objekten

HINWEIS: An dieser Stelle müssen wir etwas vorgreifen, Abschnitt 11.4 geht auf dieses Thema im Detail ein.

Navigation zwischen Datensätzen bedeutet, dass auch irgendwo ein aktueller Datensatz gespeichert wird und entsprechende Navigationsmethoden zur Verfügung stehen. Auch bei intensiver Suche werden Sie aber derartige Eigenschaften zunächst nicht finden.

WPF erzeugt beim Binden von Collections automatisch eine Sicht auf die eigentliche Collection. Diese Sicht verwaltet den aktuellen Datensatz, bietet Navigationsmethoden an und ermöglicht das Filtern und Sortieren der Daten[2].

Diese automatisch erzeugte Sicht können Sie mit der Methode *CollectionViewSource.GetDefaultView* für eine spezifische Collection abrufen.

Beispiel 11.16: *(Fortsetzung)* Abrufen und Verwenden der *DefaultView* für unsere Collection

C#

Wir erweitern die Liste der lokalen Variablen, um die Sicht zu speichern:

```
        private ICollectionView view;
...
        public MainWindow()
        {
            Abteilung = new ObservableCollection<Person>();
...
```

[2] Derartige Sichten können Sie auch selbst erstellen und quasi als Schicht zwischen Daten und *DataContext* schieben.

Vergessen Sie nicht, einen Verweis auf den Namespace *System.ComponentModel* zu setzen.

Im Konstruktor rufen wir die Sicht ab:

```
            view = CollectionViewSource.GetDefaultView(Abteilung);
        }
```

Jetzt können wir mit dieser Sicht auch die Navigation zwischen den einzelnen Elementen der Collection realisieren.

Nächstes Objekt:

```
        private void ButtonNext_Click(object sender, RoutedEventArgs e)
        {
            view.MoveCurrentToNext();
            if (view.IsCurrentAfterLast)
            {
                view.MoveCurrentToLast();
            }
        }
```

Vorhergehendes Objekt:

```
        private void ButtonPrevious_Click(object sender, RoutedEventArgs e)
        {
            view.MoveCurrentToPrevious();
            if (view.IsCurrentBeforeFirst)
            {
                view.MoveCurrentToFirst();
            }
        }
```

Wir fügen zum Testen ein neues Objekt zur Laufzeit in die Collection ein:

```
        private void Button_Neu(object sender, RoutedEventArgs e)
        {
            Abteilung.Add(
              new Person() { Nachname = "Moratschek", Vorname = "Willi" });
        }
```

Auch das Löschen von Objekten ist auf diesem Wege möglich:

```
        private void ButtonDelete_Click(object sender, RoutedEventArgs e)
        {
            Abteilung.Remove(view.CurrentItem as Person);
        }
```

Nach dem Start des Beispiels können Sie zwischen den Objekten „navigieren“, Objekte hinzufügen und diese auch wieder löschen. Das Ganze kommt Ihnen sicherlich unter dem Stichwort „Datenbanknavigator“ bekannt vor.

11.3.4 Einfache Anzeige in einer ListBox

Das Anzeigen von Einzeldatensätzen ist ja schon ganz gut, wie aber steht es mit dem Füllen von ganzen Listenfeldern?

Auch hier können Sie, dank Datenbindung, schnell zu brauchbaren Ergebnissen kommen.

Beispiel 11.17: *(Fortsetzung)* Anbinden einer *ListBox* an unsere Collection

C#

Es genügt zunächst die einfache Zuweisung von *"{Binding}"* an die *ItemsSource*:

```
<ListBox Height="100" IsSynchronizedWithCurrentItem="True"
         Name="listBox1" ItemsSource="{Binding}"/>
```

Der Hintergrund: Da die Collection bereits direkt an das Formular gebunden ist, brauchen wir hier nicht weitere Eigenschaften zu spezifizieren. Alternativ könnten Sie hier auch die Collection per *DataContext* zuweisen.

Und wofür ist das Attribut *IsSynchronizedWithCurrentItem* verantwortlich? Hier sollten Sie sich an unsere Sicht erinnern, die auch den aktuellen „Satzzeiger" verwaltet. Nur wenn Sie das Attribut auf *True* setzen, wird das aktuelle Item mit dem „Satzzeiger" synchronisiert (dies gilt für beide Richtungen).

Ergebnis

Die angezeigte *ListBox* zur Laufzeit:

HINWEIS: Die *ItemsSource*-Eigenschaft kann nur verwendet werden, wenn die *Items*-Collection eines *ItemsControl* leer ist. Falls nicht, wird Ihre Anwendung eine *InvalidOperationException* auslösen.

Doch woher „weiß" die *ListBox* eigentlich, welche Eigenschaften des *Schüler*-Objekts in der Liste darzustellen sind? Antwort: Sie weiß es nicht und verwendet in diesem Fall einfach die *ToString*-Methode des betreffenden Objekts. Wenn Sie jetzt mal kurz in Abschnitt 11.2.3, „Anforderungen an die Quell-Klasse", nachschlagen, werden Sie feststellen, dass wir in weiser Vorahnung bereits die *ToString*-Methode überschrieben haben und damit eine Kombination aus *Nachname* und *Vorname* zurückgeben (siehe oben).

Verwendung von DisplayMemberPath

Natürlich ist das Überschreiben der *ToString*-Methode nicht der Weisheit letzter Schluss und so ist es sicher sinnvoll, noch einen anderen Weg zur Auswahl des anzuzeigenden Members

zu unterstützen. Genau für diesen Zweck wird die *DisplayMemberPath*-Eigenschaft angeboten. Diese bestimmt, welcher Member für den Text des Listeneintrags verwendet wird.

Beispiel 11.18: Verwendung von *DisplayMemberPath* für die Auswahl der anzuzeigenden Eigenschaft

XAML

```
<ListBox Height="100"  IsSynchronizedWithCurrentItem="True"
          ItemsSource="{Binding}" DisplayMemberPath="Nachname"/>
```

Leider genügt jedoch auch diese Version der Anzeigeformatierung nicht immer und so landen wir unweigerlich bei den *DataTemplates*.

11.3.5 DataTemplates zur Anzeigeformatierung

Obige Art der Datenbindung dürfte in vielen Fällen wohl kaum genügen. Die WPF-Entwickler haben aber auch für für diesen Fall vorgesorgt und mit dem *DataTemplate* ein mächtiges Werkzeug geschaffen.

Das Prinzip: Jeder *ListBox/ComboBox* können Sie ein *DataTemplate* zuweisen, das dafür verantwortlich ist, wie das einzelne Item aufgebaut ist (quasi eine Schablone in die die Daten eingefügt werden). Und da WPF im Content eines Items fast jede Zusammenstellung von Elementen akzeptiert, können Sie hier Formatierungen beliebiger Art erzeugen (natürlich im Rahmen der XAML-Vorgaben).

Beispiel 11.19: *(Fortsetzung)* Wir wollen in der *ListBox* eine zweispaltige Anzeige realisieren (links der Nachname, rechts der Nachname und der Vorname).

XAML

In den Ressourcen (z. B. Window) erzeugen Sie das erforderliche *DataTemplate*:

```
<Window.Resources>
  <DataTemplate x:Key="AbteilungsListTemplate">
```

Das Layout bestimmen Sie:

```
    <StackPanel Orientation="Horizontal">
```

Bei der Zuweisung von Inhalten können Sie jetzt direkt auf die Eigenschaften zugreifen:

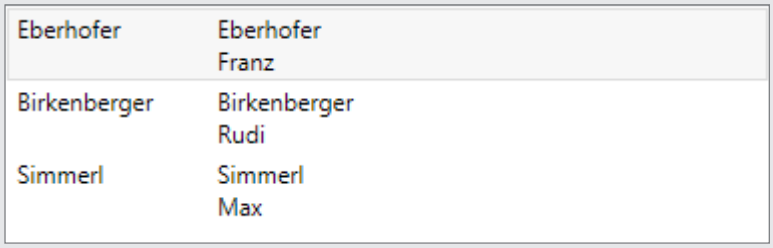
```
      <TextBlock VerticalAlignment="Top" Width="100"
                 Text="{Binding Path=Nachname}" />
      <StackPanel>
        <TextBlock Text="{Binding Path=Nachname}" />
        <TextBlock Text="{Binding Path=Vorname}" />
      </StackPanel>
    </StackPanel>
  </DataTemplate>
</Window.Resources>
...
```

Last, but not least, müssen Sie der *ListBox* auch noch das Template zuweisen:

```
<ListBox Height="120" IsSynchronizedWithCurrentItem="True"
         Name="listBox1" ItemsSource="{Binding}"
         ItemTemplate="{StaticResource AbteilungsListTemplate}"/>
```

Ergebnis

Die erzeugte *ListBox*:



Dass Sie hier auch mit Grafiken, optischen Effekten, Kontextmenüs etc. arbeiten können, sollte nach den Darstellungen der vorhergehenden Kapitel klar sein.

11.3.6 Mehr zu List- und ComboBox

An dieser Stelle wollen wir uns noch einige spezielle Eigenschaften von *List-* und *ComboBox* ansehen, die in der täglichen Programmierpraxis von Bedeutung sind.

SelectedIndex

Möchten Sie Einträge in der *ListBox* auswählen bzw. bestimmen, der wievielte Eintrag (Index) in der Liste markiert ist, können Sie die *SelectedIndex*-Eigenschaft verwenden.

Beispiel 11.20: Auswahl des zweiten Eintrags

C#

```
private void ButtonZweitenSatzAuswaehlen_Click(
        object sender, RoutedEventArgs e)
{
    listBox1.SelectedIndex = 1;
}
```

SelectedItem/SelectedItems

Möchten Sie das markierte Listenelement selbst abrufen bzw. das damit verbundene Objekt, verwenden Sie die *SelectedItem*-Eigenschaft. Alternativ können Sie auch eine Liste der markierten Einträge mit *SelectedItems* abrufen.

HINWEIS: Die Collection *SelectedItems* steht Ihnen nur zur Verfügung, wenn Sie *SelectionMode* auf *Multiple* festgelegt haben.

Beispiel 11.21: Verwendung *SelectedItem*/*SelectedItems*

C#

Wir nutzen unsere überschriebene *ToString*-Methode:

```
MessageBox.Show(listBox1.SelectedItem.ToString());
```

Wir greifen direkt auf einen Member (typisieren nicht vergessen) zu:

```
MessageBox.Show((listBox1.SelectedItem as Person)?.Nachname);
```

Wir zeigen alle markierten Einträge an:

```
foreach (Person p in listBox1.SelectedItems)
{
    MessageBox.Show(p.Nachname);
}
```

SelectedValuePath und SelectedValue

Mit *SelectedValuePath* können Sie festlegen, welcher Member von der Eigenschaft *SelectedValue* zurückgegeben wird. Dies ist im Zusammenhang mit Datenbanken meist der Primärindex der Tabelle, mit dem Sie einen Datensatz eindeutig identifizieren können.

HINWEIS: Ist *SelectedValuePath* nicht festgelegt, gibt *SelectedValue* das komplette Objekt zurück (entspricht *SelectedItem*).

Beispiel 11.22: Verwendung *SelectedValuePath* und *SelectedValue*

XAML

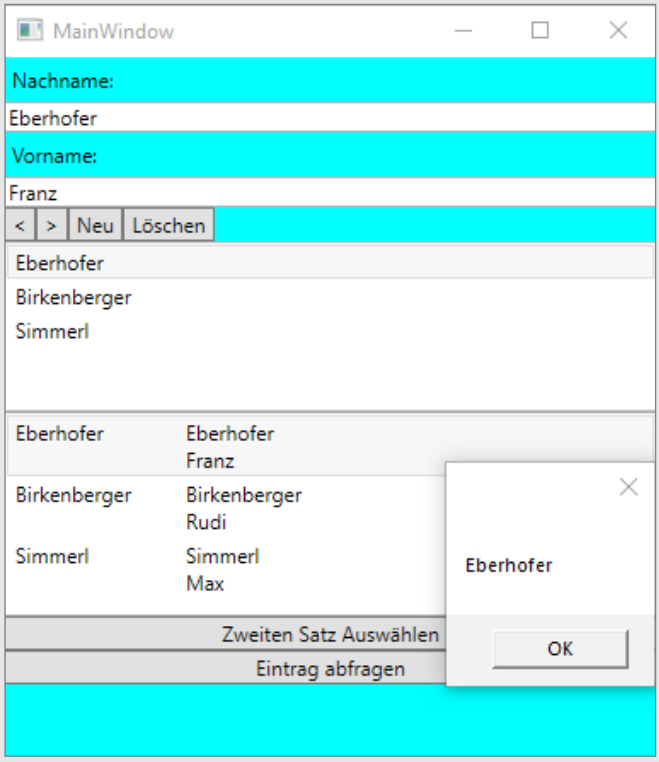
```
<ListBox  IsSynchronizedWithCurrentItem="True" Name="listBox1"
          ItemsSource="{Binding}" DisplayMemberPath="Nachname"
          SelectedValuePath="Nachname"/>
```

C#

```
private void Button2_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(listBox1.SelectedValue.ToString());
}
```

Ergebnis

Das Ergebnis zur Laufzeit:



11.3.7 Verwendung der ListView

Im vorhergehenden Kapitel hatten wir die *ListView* ja bereits kurz gestreift (Trockenschwimmen), an dieser Stelle zeigen wir Ihnen die *ListView* „in Action".

Einfache Bindung

Prinzipiell ist die *ListView* der *ListBox* recht ähnlich. Die Anbindung der Einträge erfolgt ebenfalls per *ItemsSource*, die Auswahl bzw. Bestimmung (*SelectedItem*, *SelectedValue* etc.) der markierten Einträge ist analog realisiert.

Neu ist, dass die *ListView* über Spaltenköpfe verfügt, die Sie getrennt konfigurieren können (*GridViewColumnHeader*) und gegebenenfalls auch für das Sortieren (siehe 2. Beispiel) verwenden können.

Ein weiterer Unterschied ist die Unterstützung von verschiedenen Ansichten, von denen jedoch nur die *GridView* vordefiniert ist. Im Weiteren werden wir uns auch nur auf diese Ansicht beschränken.

Beispiel 11.23: *(Fortsetzung)* Anzeige der Collection-Daten in einer *ListView*

XAML

Zuweisen der Datenquelle (Übernahme von *Window.DataContext*):

```
<ListView Name="listView1" Height="100" IsSynchronizedWithCurrentItem="True"
          ItemsSource="{Binding}">
  <ListView.View>
```

Hier wird die *GridView* definiert:

```
    <GridView>
```

Die einzelnen Spalten definieren:

```
      <GridView.Columns>
```

Und jetzt wird es einfach, binden Sie lediglich die gewünschten Eigenschaften an die einzelnen Spalten der *GridView*:

```
        <GridViewColumn Header="Name"
                DisplayMemberBinding="{Binding Path=Nachname}" />
        <GridViewColumn Header="Vorname"
                DisplayMemberBinding="{Binding Path=Vorname}" />
      </GridView.Columns>
    </GridView>
  </ListView.View>
</ListView>
```

Ergebnis

Das Endergebnis zur Laufzeit:

Name	Vorname
Eberhofer	Franz
Birkenberger	Rudi
Simmerl	Max

Sortieren der Einträge

Wie schon erwähnt, können Sie die Spaltenköpfe auch für das Sortieren der Einträge nutzen. Ein einfaches Beispiel zeigt die Vorgehensweise:

Beispiel 11.24: Sortieren nach Klick auf den jeweiligen Spaltenkopf

XAML

Unsere Änderung in der Seitenbeschreibung:

```
<ListView Name="listView1" IsSynchronizedWithCurrentItem="True"
          ItemsSource="{Binding}">
  <ListView.View>
    <GridView>
```

```
        <GridView.Columns>
          <GridViewColumn DisplayMemberBinding="{Binding Path=Nachname}" >
             <GridViewColumnHeader Click="SortClick" Content="Nachname" />
          </GridViewColumn>
          <GridViewColumn DisplayMemberBinding="{Binding Path=Vorname}" >
             <GridViewColumnHeader Click="SortClick" Content="Vorname" />
          </GridViewColumn>
        </GridView.Columns>
      </GridView>
    </ListView.View>
  </ListView>
```

C#

Der Quellcode fällt recht kurz aus:

```
using System.ComponentModel;
using System.Collections.ObjectModel;
...
       private void SortClick(object sender, RoutedEventArgs e)
       {
```

Zunächst die betreffende Spalte bestimmen:

```
            GridViewColumnHeader spalte = sender as GridViewColumnHeader;
```

Die Default-View bestimmen:

```
            ICollectionView view =
                CollectionViewSource.GetDefaultView(listView1.ItemsSource);
```

Eine neue Sortierfolge festlegen:

```
            view.SortDescriptions.Clear();
            view.SortDescriptions.Add(
                 new SortDescription(spalte.Content.ToString(),
                                     ListSortDirection.Ascending));
```

Und aktualisieren:

```
            view.Refresh();
       }
```

Auf weitere Experimente mit der *ListView* verzichten wir an dieser Stelle, mit dem *DataGrid* steht uns ein wesentlich mächtigeres Control zur Verfügung. Mehr dazu in Abschnitt 11.7.

Wie Sie auch größere Collections bändigen, zeigt das Praxisbeispiel in Abschnitt 11.8.

11.4 Noch einmal zurück zu den Details

Nachdem wir in den bisherigen Abschnitten schon mehrfach vorgreifen mussten, wollen wir an dieser Stelle noch einmal kurz auf einige Details der Datenbindung eingehen.

Interessant für den Datenbankprogrammierer ist vor allem eine Zwischenschicht, die vom WPF quasi zwischen die Daten (Collections) und die reinen Anzeige-Controls (z. B. *ListView*) geschoben wird, um einige datenbanktypische Operationen zu ermöglichen:

- Verwaltung des aktuellen Satzzeigers
- Navigation zwischen den Datensätzen
- Sortierfunktion
- Filterfunktion

Die Rede ist von der Klasse *CollectionView*, um deren Erzeugung Sie sich nicht selbst kümmern müssen, da Sie diese automatisch erstellte View recht einfach abrufen können.

Beispiel 11.25: Abrufen der *CollectionView*

C#

```
...
        NwDataContext db = new NwDataContext();
        ICollectionView view;
...
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            lvOrder.DataContext = db.Orders;
            view = CollectionViewSource.GetDefaultView(lvOrder.DataContext);
        }
```

Mit dieser *CollectionView* stellt es jetzt kein Problem mehr dar, die oben gewünschten Datenbankfunktionen zu implementieren.

11.4.1 Navigieren in den Daten

Wie schon in den vorhergehenden Abschnitten gezeigt, ist eine der Hauptaufgaben der *CollectionView* die Verwaltung des „Satzzeigers“. Dazu steht Ihnen zunächst die Eigenschaft *CurrentItem* zur Verfügung, die das aktuell ausgewählte Element der gebundenen Collection zurückgibt.

Weitere interessante Eigenschaften:

Eigenschaft	Beschreibung
CurrentItem	Aktuelles Element der Auflistung.
CurrentPosition	Ordinalposition des aktuellen Elements in der Auflistung.
IsCurrentAfterLast	Befindet sich der „Satzzeiger“ hinter dem Ende der Auflistung?
IsCurrentBeforeFirst	Befindet sich der „Satzzeiger“ vor dem Beginn der Auflistung?

Die eigentliche Navigation realisieren Sie mit den folgenden Methoden:

Methode	Beschreibung
MoveCurrentTo	Das übergebene Element wird als *CurrentItem* festgelegt.
MoveCurrentToFirst	„Satzzeiger“ auf das erste Element verschieben.
MoveCurrentToLast	„Satzzeiger“ auf das letzte Element verschieben.
MoveCurrentToNext	„Satzzeiger“ auf das folgende Element verschieben.
MoveCurrentToPosition	„Satzzeiger“ auf den angegebenen Index verschieben.
MoveCurrentToPrevious	„Satzzeiger“ auf das vorhergehende Element verschieben.

Beispiel 11.26: Navigationstasten für „Vor“ und „Zurück“

C#

Der folgende Aufwand ist nötig, um nicht hinter bzw. vor dem letzten bzw. ersten Datensatz zu landen:

```
private void ButtonNext_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToNext();
    if (view.IsCurrentAfterLast)
    {
        view.MoveCurrentToLast();
    }
}

private void ButtonPrevious_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToPrevious();
    if (view.IsCurrentBeforeFirst)
    {
        view.MoveCurrentToFirst();
    }
}
```

Beispiel 11.27: Verwendung von *CurrentItem*

C#

Löschen eines Listeneintrags per *CurrentItem* und Typisierung:

```
private void ButtonLoeschen_Click(object sender, RoutedEventArgs e)
{
    Abteilung.Remove(view.CurrentItem as Person);
}
```

11.4.2 Sortieren

Dass sich die *CollectionView* auch zum Sortieren eignet, haben wir ja bereits am Beispiel der *ListView* gezeigt, wo durch Klicken auf den Spaltenkopf die Collection nach der jeweiligen Spalte sortiert wurde.

Zum Einsatz kommt die Collection *SortDescriptions*, die neben den Membernamen auch die Sortierfolge enthält. Da es sich um eine Collection handelt, können Sie auch mehrere Elemente angeben:

Beispiel 11.28: Sortieren einer Collection

C#

```
private void SortClick(object sender, RoutedEventArgs e)
{
   GridViewColumnHeader spalte = sender as GridViewColumnHeader;
```

CollectionView abrufen:

```
   ICollectionView view =
         CollectionViewSource.GetDefaultView(listView1.ItemsSource);
```

Bisherige Sortiervorgaben löschen:

```
   view.SortDescriptions.Clear();
```

Eine neue Sortierfolge (Spaltenname, aufsteigend) festlegen:

```
   view.SortDescriptions.Add(new SortDescription(
         spalte.Content.ToString(),
         ListSortDirection.Ascending));
```

Ansicht aktualisieren:

```
   view.Refresh();
}
```

11.4.3 Filtern

Auch wenn Sie mit dieser Variante vorsichtig sein sollten (Daten werden eigentlich vor der Anzeige gefiltert, um unnötigen Traffic zu vermeiden), so besteht doch die Möglichkeit, zur Laufzeit gezielt Daten aus der gebundenen Collection herauszufiltern. Nutzen Sie dazu die *Filter*-Eigenschaft, der Sie eine selbst zu definierende Methode zuweisen.

Beispiel 11.29: Filter festlegen

C#

Zunächst unsere Filterfunktion (alle Vornamen die mit „R“ beginnen):

```
protected bool FilterVorname(object value)
{
    Person p = value as Person;
    return p.Vorname.StartsWith("R");
}
```

Und hier wird der Filter zugewiesen (ein Aktualisieren ist nicht nötig):

```
private void ButtonFilter_Click(object sender, RoutedEventArgs e)
{
    ICollectionView view =
       CollectionViewSource.GetDefaultView(listBox1.ItemsSource);
    view.Filter += FilterVorname;
}
```

HINWEIS: Möchten Sie den Filter wieder löschen, weisen Sie der Eigenschaft einfach *null* zu.

11.4.4 Live Shaping

Die vorhergehenden Funktionen zum Sortieren, Filtern und auch Gruppieren funktionieren recht gut, auch wenn Sie zum Beispiel neue Einträge zur Collection hinzufügen. Alternativ können Sie auch die *Refresh*-Methode der *CollectionView* aufrufen.

Doch was ist, wenn Sie lediglich ein Objekt der Collection bearbeiten und sich so z. B. die Filterbedingung für dieses Objekt ändert? In diesem Fall werden Sie schnell feststellen, dass Anzeige und Inhalt der Collection nicht mehr übereinstimmt.

Beispiel 11.30: *(Fortsetzung)* Fehlende Aktualisierung

C#

```
...
```

Filtern Sie die Daten und rufen Sie folgende Methode auf, passiert nichts:

```
private void ButtonFilterChange_Click(
        object sender, RoutedEventArgs e)
{
    Person person = Abteilung.FirstOrDefault(
           p => p.Vorname == "Rudi");
    p.Vorname = "Rudolf";
```

Erst nach einem Refresh sind die gefilterten Daten auch aktuell:

```
            //ICollectionView view = CollectionViewSource.GetDefaultView(
                                     listBox1.ItemsSource);
            //view.Refresh();
        }
...
```

Hier hilft Ihnen Live Shaping weiter. Über das Interface *ICollectionViewLiveShaping* können Sie bestimmte Spalten zur Überwachung anmelden.

Jeweils drei neue Member sind für die weitere Arbeit interessant:

- Die Eigenschaft *CanChangeLiveFiltering* (*… Sorting*, *… Grouping*) bestimmt, ob die Überwachung möglich ist.
- Die Eigenschaft *IsLiveFiltering* (*… Sorting*, *… Grouping*) bestimmt, ob die Überwachung eingeschaltet ist.
- Die Collection *LiveFilteringProperties* (*… Sorting…*, *… Grouping…*) enthält die Namen der zu überwachenden Eigenschaften.

Beispiel 11.31: Filtern mit Live Shaping

C#

```
...
        private void ButtonLiveShaping_Click(
                object sender, RoutedEventArgs e)
        {
```

Filter wie bekannt festlegen:

```
           view.Filter += FilterVorname;
```

Wir rufen das neue Interface ab:

```
            ICollectionViewLiveShaping viewls =
                  view as ICollectionViewLiveShaping;
```

Test auf Interface:

```
            if (viewls == null)
            {
                MessageBox.Show("Nicht unterstützt!");
            }
```

Überprüfen ob eine Überwachung möglich ist?

```
            if (viewls.CanChangeLiveFiltering)
            {
```

Feld *Vorname* soll überwacht werden:

```
                viewls.LiveFilteringProperties.Add(nameof(Person.Vorname));
                viewls.IsLiveFiltering = true;
            }
        }
...
```

HINWEIS: Da diese Form der Überwachung recht ressourcenintensiv ist, sollten Sie davon nur Gebrauch machen, wenn es unbedingt nötig ist.

11.5 Anzeige von Datenbankinhalten

Anzeige eigener Collections gut und schön, aber wir wollen auch noch kurz einen Blick aufs große Ganze werfen und damit sind wir schon bei der „Königsdisziplin“, den Datenbanken, angelangt.

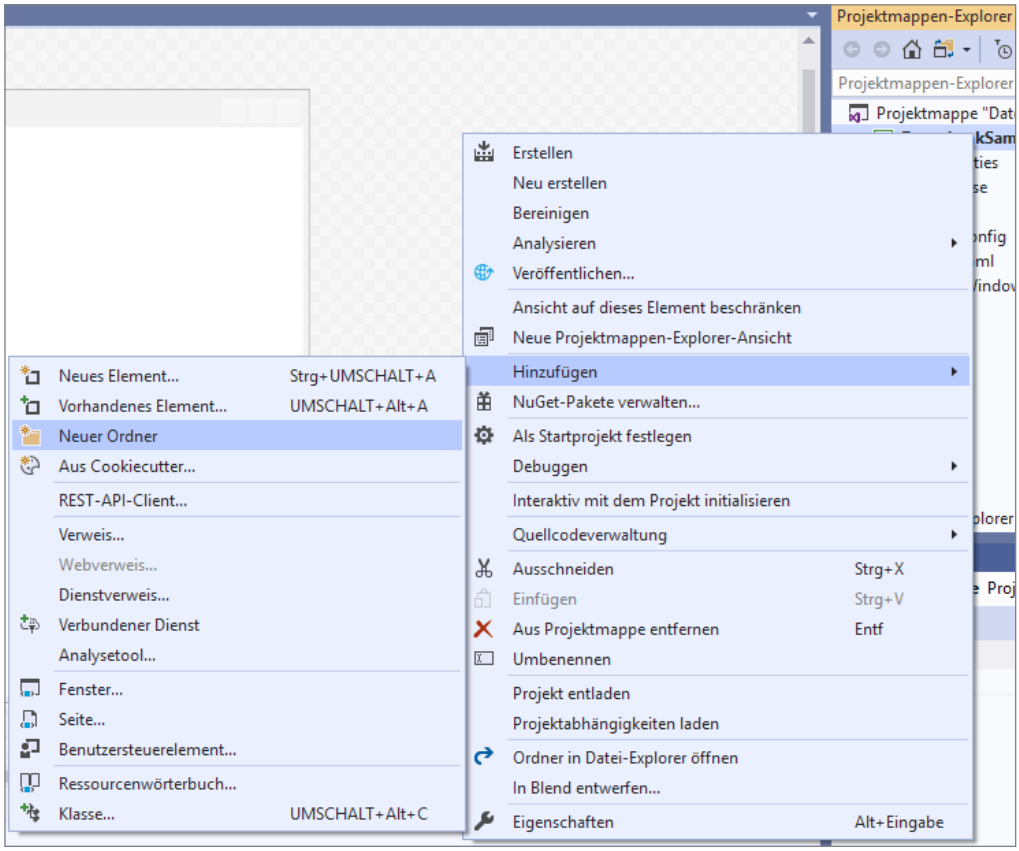
HINWEIS: Wer jetzt Berge von ADO.NET-Quellcode erwartet, den werden wir enttäuschen. Für den Zugriff auf unsere *Northwind*-Beispieldatenbank werden wir das Entity Framework verwenden.

HINWEIS: Wenn Sie bislang noch nicht mit Datenbanken gearbeitet haben, sollten Sie das Kapitel 18 durcharbeiten, bevor Sie mit diesem Abschnitt fortfahren.

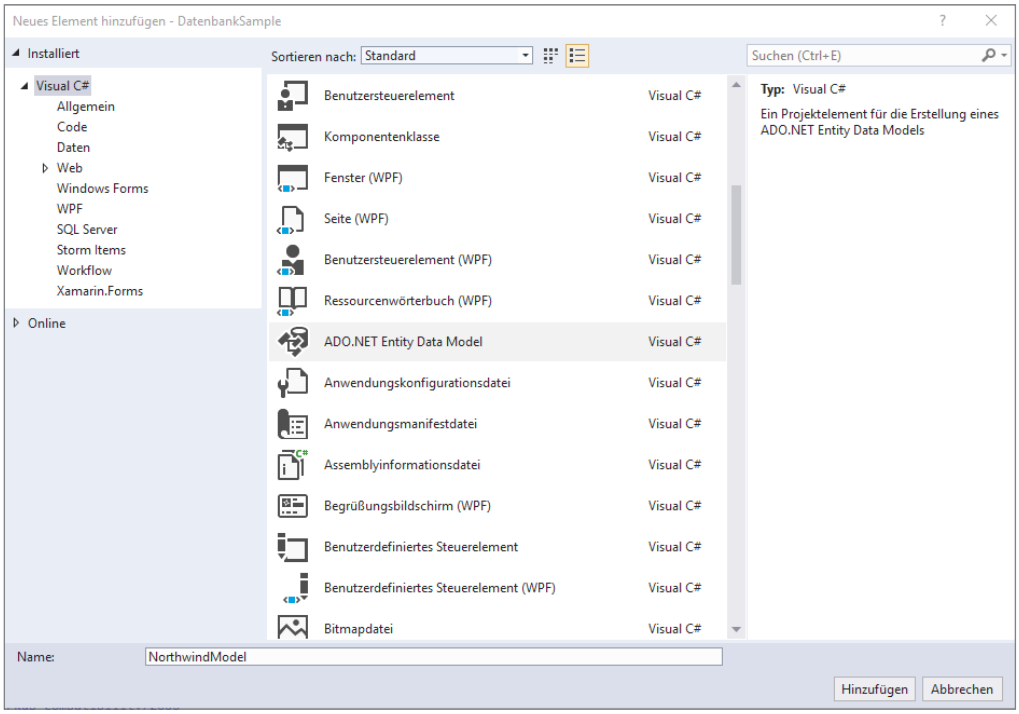
11.5.1 Datenmodell per EF-Designer erzeugen

Fügen Sie Ihrem WPF-Projekt zuerst einen neuen Ordner hinzu und nennen diesen *Model*.

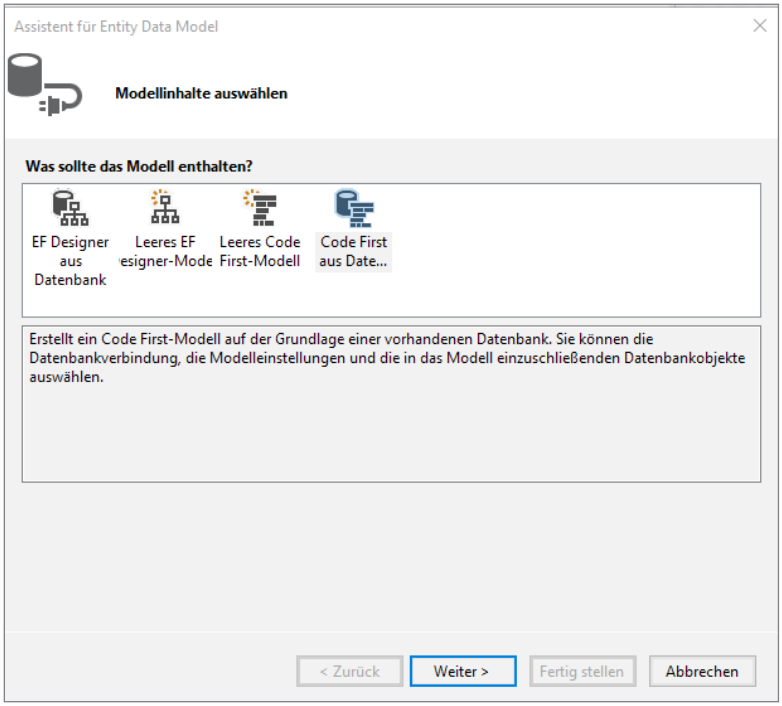
HINWEIS: Sollten Sie die Northwind-Beispieldatenbank nicht installiert haben, können Sie unter *https://github.com/Microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs* das Installationskript herunterladen.



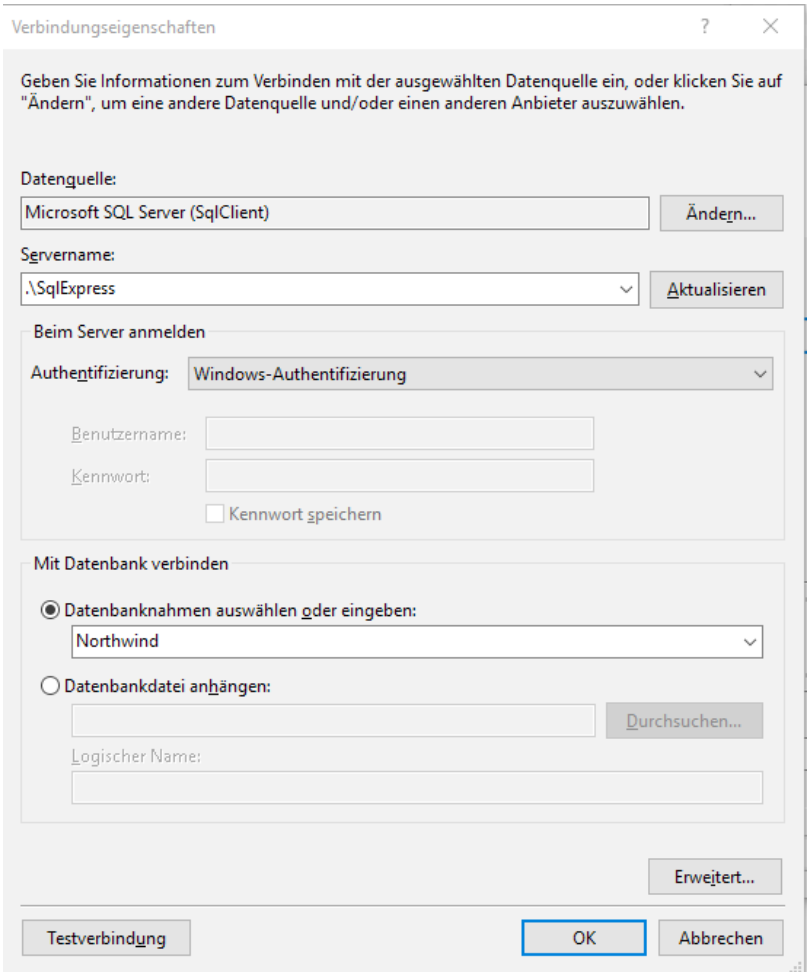
Dann fügen Sie im Ordner *Model* eine neue Klasse hinzu und wählen als Vorlage *ADO.NET Entity Data Model*. Vergeben Sie *NorthwindModel* als Name.



Im nächsten Dialog wählen Sie dann den Eintrag *CodeFirst aus Datenbank*.



Im nächsten Schritt wählen Sie die Verbindungseigenschaften zu Ihrer *Northwind*-Datenbank aus. Bei mir läuft die Datenbank in einer SQL Server Express-Instanz. Beim nächsten Dialog übernehmen Sie dann die vorgeschlagenen Einstellungen. Ändern Sie lediglich den Namen für die Verbindungszeichenfolge auf *Northwind_Model*, um eine Namenseindeutigkeit sicherzustellen. Die Verbindungszeichenfolge wird dann in der Konfigurationsdatei *app.config* der Applikation gespeichert.



Wählen Sie dann die drei Tabellen *Orders*, *Order_Details* und *Products* aus und machen Sie ein Häkchen bei der CheckBox *Generierte Objektnamen in den Singular oder Plural*. Dies bewirkt, dass die erzeugten Objektnamen im Singular (also ohne endendes „-s“) benannt werden.

Der Designer erstellt nachfolgend automatisch die erforderlichen C#-Entitätsklassen für die einzelnen Tabellen sowie deren Beziehungen. Damit sind wir aber auch schon wieder bei den schon bekannten Collections angekommen, die weitere Vorgehensweise dürfte Ihnen also bekannt vorkommen.

HINWEIS: Sie können neben reinen Tabellen auch Views dem Model hinzufügen. Views werden dabei wie Tabellen behandelt.

11.5.2 Die Programmoberfläche

Nach Schließen des Designers wollen wir uns mit einem einfachen WPF-Projekt von der Funktionsfähigkeit überzeugen.

```
<Window x:Class="DatenbankSample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:DatenbankSample"
        mc:Ignorable="d"
```

Eine Ereignisprozedur beim Öffnen des Window:

```
    Title="MainWindow" Height="450" Width="800" Loaded="Window_Loaded">
```

Zweispaltiges Layout per *Grid*:

```
 <Grid>
   <Grid.ColumnDefinitions>
     <ColumnDefinition Width="75" />
     <ColumnDefinition Width="*" />
   </Grid.ColumnDefinitions>
```

Hier die *ListView* mit den vorhandenen Bestellungen (Tabelle *Order*):

```
   <ListView Grid.Column="0" Name="lvOrder" IsSynchronizedWithCurrentItem="True"
             ItemsSource="{Binding}" SelectionChanged="lvOrder_SelectionChanged"
             HorizontalAlignment="Left" >
```

Die eigentlich Bindung erfolgt per *DataContext*-Zuweisung im C#-Code. Über das *SelectionChanged*-Ereignis werden wir die Detaildaten in die zweite *ListView* „zaubern“.

```
     <ListView.View>
       <GridView>
         <GridView.Columns>
```

Wir zeigen nur die Spalte mit der Bestellnummer an:

```
           <GridViewColumn Header="OrderID"
                DisplayMemberBinding="{Binding OrderID}" />
         </GridView.Columns>
       </GridView>
     </ListView.View>
   </ListView>
```

Die *ListView* für die Detaildaten (die *DataContext*-Eigenschaft setzen wir im *SelectionChanged*-Ereignis der obigen *ListView*):

```
   <ListView Grid.Column="1" Name="lvOrderDetails"
             IsSynchronizedWithCurrentItem="True" ItemsSource="{Binding}" >
     <ListView.View>
       <GridView>
         <GridView.Columns>
           <GridViewColumn Header="OrderId"
                DisplayMemberBinding="{Binding OrderID}" />
           <GridViewColumn Header="ID"
                DisplayMemberBinding="{Binding ProductID}" />
```

Wer jetzt erwartet, dass wir uns jetzt noch die Mühe machen und noch eine dritte *GridView* für die Artikelnamen einbinden, hat nicht mit der Leistungsfähigkeit von LINQ und dem Entity Framework gerechnet. Es genügt die Abfrage der untergeordneten Collection *Product*:

```
           <GridViewColumn Header="Artikelname"
                           DisplayMemberBinding="{Binding Product.ProductName}" />
```

Ein sinnvoller Vorteil von objektrelationalen Mapper-Klassen!

```
         </GridView.Columns>
       </GridView>
     </ListView.View>
   </ListView>
 </Grid>
</Window>
```

11.5.3 Der Zugriff auf die Daten

Jetzt müssen wir noch den erforderlichen C#-Code erstellen, um die Daten auch aus der Datenbank abzurufen.

```
...
    public partial class MainWindow : Window
    {
```

Die meiste Arbeit nimmt uns der *Entity Framework DataContext* ab, den wir gleich zu Beginn instanziieren:

```
        NorthwindModel context = new NorthwindModel();
```

Da wird das *Model* in einem eigenen Ordner *Model* gespeichert haben, benötigen wir noch einen Verweis auf den Namespace *DatenbankSample.Model*.

Wir wollen auch die Default-View zwischenspeichern, da wir diese für das Abrufen der Detaildatensätze benötigen:

```
        ICollectionView view;
...
```

Auch hierfür benötigen wir wieder den Namespace *System.ComponentModel*.

Beim Laden des Fensters:

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

Dem *DataContext* der linken *ListView* wird die Tabelle *Orders* zugewiesen:

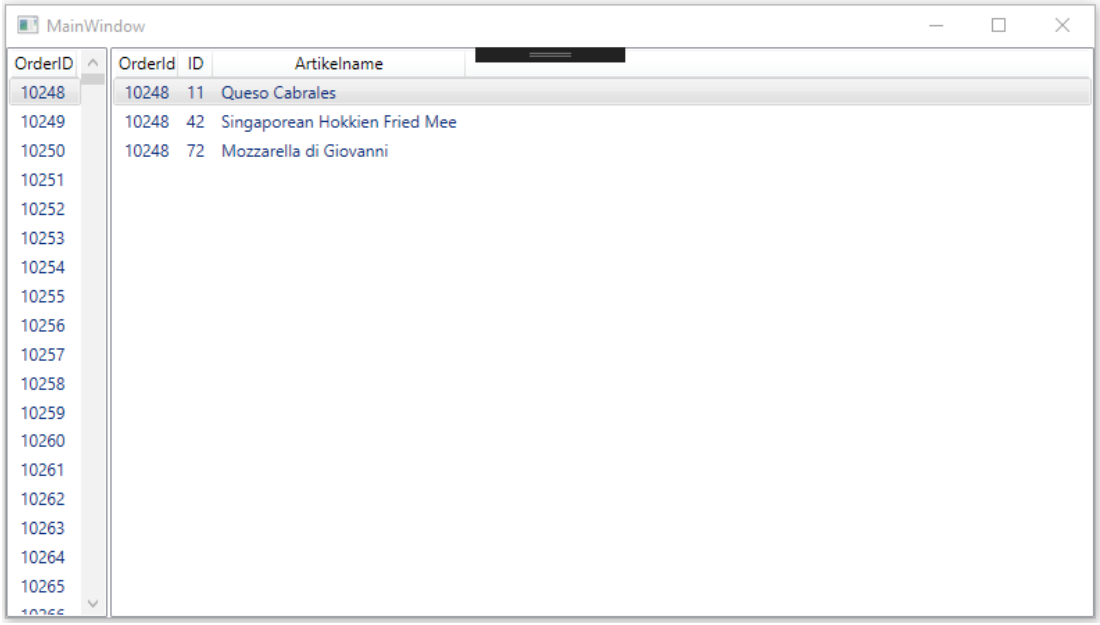
```
            lvOrder.DataContext = context.Orders.ToList();
```

Die Default-View abrufen:

```
            view = CollectionViewSource.GetDefaultView(lvOrder.DataContext);
        }
```

So, das wäre schon alles, wenn da nicht noch die Detaildatenanzeige fehlen würde. Im *SelectionChanged*-Ereignis der linken *ListView* kümmern wir uns zunächst um das Abrufen des aktuellen Datensatzes und leiten aus diesem Objekt die *OrderDetails* ab (als Collection enthalten):

```
        private void lvOrder_SelectionChanged(object sender,
                                              SelectionChangedEventArgs e)
        {
            lvOrderDetails.DataContext = (view.CurrentItem as Order).Order_Details;
        }
    }
```



Ach ja, wie kommen eigentlich neue Datensätze in die Tabellen? Hier genügt es, wenn Sie z. B. ein neues *Product*-Objekt erstellen und es an die *Products*-Collection anhängen. Mit einem *SaveChanges*-Aufruf des *Entity-Framework-DataContext*-Objekts ist die Änderung dann auch schon zum Server übertragen.

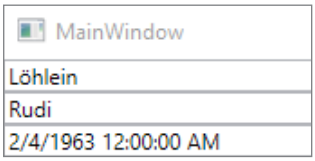
HINWEIS: In Kapitel 18 werden wir die Datenbankkonzepte noch mal ausführlich betrachten.

11.6 Formatieren von Werten

In unseren Beispielen haben wir uns bislang erfolgreich davor gedrückt, Datumswerte, Währungen etc. in einem sinnvollen Format anzuzeigen bzw. zu formatieren.

Wir erweitern unsere Personenklasse aus den vorigen Beispielen um eine Eigenschaft *Geburtstag* vom Datentyp *DateTime*.

Binden Sie beispielsweise einen Datumswert an eine *TextBox*, wird zunächst das Standardformat angezeigt:



Das sieht aus deutscher Sicht zunächst wenig erfreulich aus, aber mit dem *Language*-Attribut können Sie hier etwas nachhelfen.

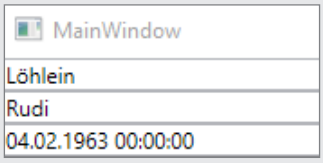
Beispiel 11.32: Verwendung *Language*-Attribut

XAML

```
<TextBox Text="{Binding Path=Geburtstag}" Language="de"/>
```

Ergebnis

Nachfolgend sollte zumindest ein deutscher Datumswert angezeigt werden:



Doch auch dies ist noch nicht der Weisheit letzter Schluss.

11.6.1 IValueConverter

Mithilfe der WPF-Wertkonvertierer können Sie jede beliebige Konvertierung zwischen Quelle und Ziel einer Datenbindung realisieren. Dazu erstellen Sie eine Klasse, die das *IValueConverter*-Interface unterstützt. Diese Klasse muss zwei Methoden implementieren:

- *Convert* (von der Quelle zum Ziel)
- *ConvertBack* (vom Ziel zur Quelle)

Sicher können Sie sich denken, dass die *ConvertBack*-Methode den höheren Programmieraufwand erfordert, hat doch hier der User die Möglichkeit, zunächst beliebige Werte in die Textfelder einzugeben, die Sie dann mühsam in den geforderten Datentyp umwandeln müssen.

Beispiel 11.33: Implementieren und Verwenden eines Wert-Konvertierers

C#

An dieser Stelle wollen wir allerdings nicht das Rad neu erfinden, sondern ein Beispiel aus dem Microsoft MSDN darstellen.

Erstellen Sie dazu eine neue Klasse *DateConverter*:

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

namespace Formatierungen
{
```

Hier die neue Klasse *DateConverter*, die Sie mit entsprechenden Attributen versehen sollten:

```
    [ValueConversion(typeof(DateTime), typeof(String))]
    public class DateConverter : IValueConverter
    {
```

Konvertieren von der Quelle zum Ziel (übergeben werden die Quelleigenschaft, der Zieleigenschaft-Typ, ein Konverter-Parameter sowie die aktuellen Landeseinstellungen):

```
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            DateTime date = (DateTime)value;
            return date.ToShortDateString();
        }
```

Konvertieren vom Ziel (z. B. *TextBox*) zur Quelle (z. B. Objekt):

```
        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            string strValue = value.ToString();
            DateTime resultDateTime;
            if (DateTime.TryParse(strValue, out resultDateTime))
            {
               return resultDateTime;
```

```
            }
            return value;
        }
    }
```

XAML

Die Verwendung im XAML-Code:

```
<Window x:Class="Formatierungen.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

Zunächst den lokalen Namespace einbinden:

```
    xmlns:local="clr-namespace:Formatierungen"
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width="800" Loaded="Window_Loaded">
```

Die Einbindung der Klasse erfolgt per Ressource:

```
 <Window.Resources>
   <local:DateConverter x:Key="dateConverter"/>
 </Window.Resources>
 <StackPanel>
   <TextBox Text="{Binding Path=Nachname}" />
   <TextBox Text="{Binding Path=Vorname}" />
```

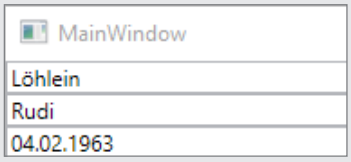
Sollten Sie nach diesen Zeilen eine Fehlermeldung bekommen, dass der Typ *DateConverter* nicht im Namespace *Formatierungen* existiert, dann kompilieren Sie die Anwendung noch mal neu (**F5**).

Und hier verwenden wir den Konverter bei der Bindung:

```
   <TextBox Text="{Binding Path=Geburtstag,
                   Converter={StaticResource dateConverter}}" />
  </StackPanel>
</Window>
```

Ergebnis

Das neue Ergebnis sieht schon viel ansprechender aus:



HINWEIS: Zusätzlich besteht die Möglichkeit, vorhandene Wertkonvertierer per Eigenschafteneditor (siehe folgende Abbildung) zuzuweisen. Der Eigenschafteneditor erstellt, falls erforderlich, die entsprechenden Einträge im <*Window.Resources*>-Abschnitt des Formulars und weist das Attribut *Converter* zu.

11.6.2 BindingBase.StringFormat-Eigenschaft

Nachdem Sie sich durch unser obiges Beispiel gequält haben, wollen wir Ihnen auch nicht die dritte Variante zur Formatierung von Werten vorenthalten.

Werfen Sie doch einmal einen Blick auf die *BindingBase.StringFormat*-Eigenschaft, welche die Verwendung eines *IValueConverter*s in vielen Standardfällen überflüssig macht.

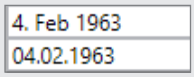
Beispiel 11.34: Zwei verschiedene Datumsformate zuweisen

XAML

```
...
<TextBox Text="{Binding Path=Geburtstag, StringFormat= d. MMM yyyy}" />
...
<TextBox Text="{Binding Path=Geburtstag, StringFormat= dd.MM.yyyy }" />
...
```

Ergebnis

Das Ergebnis:



11.7 Das DataGrid als Universalwerkzeug

Seit der WPF-Version 4 wird auch ein *DataGrid* regulär unterstützt, ohne zusätzliche Toolkits etc. laden zu müssen. Wie die schon besprochene *ListView* erlaubt auch das *DataGrid* die Anzeige von Collections im Tabellenformat. Zusätzlich werden Funktionen zum Editieren, Löschen, Auswählen und Sortieren angeboten.

HINWEIS: Anhand einiger Fallbeispiele wollen wir Ihnen eine Übersicht des Funktionsumfangs geben. Für eine komplette Beschreibung aller Eigenschaften bzw. Möglichkeiten fehlt hier jedoch der Platz und wir verweisen auf die recht umfangreiche Hilfe zum *DataGrid*-Control.

11.7.1 Grundlagen der Anzeige

Wie fast nicht anders zu erwarten, erfolgt die Anbindung an die Datenquelle mittels *ItemsSource*-Eigenschaft. Wir erzählen Ihnen an dieser Stelle also nichts Neues und verweisen auf die vorhergehenden Abschnitte.

Im Unterschied zu den bereits beschriebenen Controls bietet uns das *DataGrid* einen wesentlichen Vorteil: Sie brauchen sich nicht um das Erstellen der einzelnen Spalten zu kümmern, dank *AutoGenerateColumns*-Eigenschaft werden automatisch die erforderlichen Spalten erzeugt.

Erzeugen Sie, wie in Abschnitt 11.5.1 bereits erläutert, noch mal das *NorthwindModel* in einem Unterordner *Model*.

Beispiel 11.35: Anbinden des *DataGrid*s an die Entity-Framework-Daten

XAML

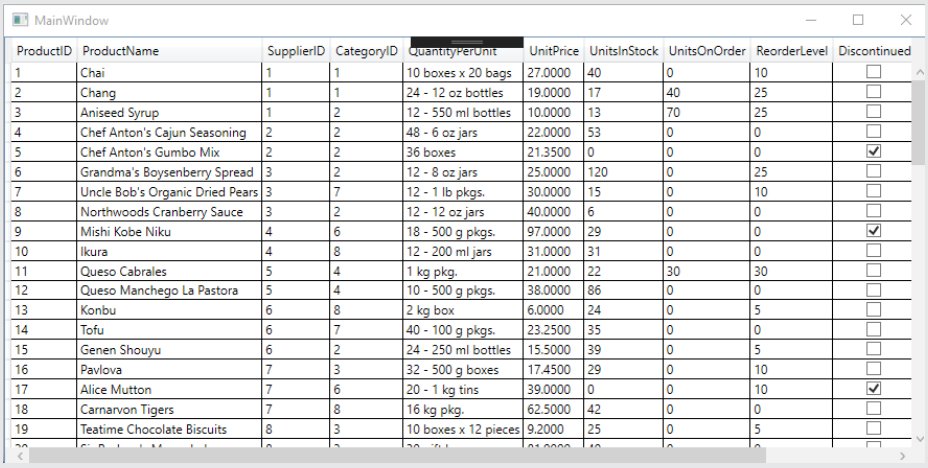
```
<DataGrid Name="dataGrid1" />
```

C#

```
    public partial class MainWindow : Window
    {
        NorthwindModel context = new NorthwindModel();
...

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            dataGrid1.ItemsSource = context.Products.ToList();
        }
```

Ergebnis



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
1	Chai	1	1	10 boxes x 20 bags	27.0000	40	0	10	☐
2	Chang	1	1	24 - 12 oz bottles	19.0000	17	40	25	☐
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000	13	70	25	☐
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22.0000	53	0	0	☐
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.3500	0	0	0	☑
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.0000	120	0	25	☐
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30.0000	15	0	10	☐
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40.0000	6	0	0	☐
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000	29	0	0	☑
10	Ikura	4	8	12 - 200 ml jars	31.0000	31	0	0	☐
11	Queso Cabrales	5	4	1 kg pkg.	21.0000	22	30	30	☐
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38.0000	86	0	0	☐
13	Konbu	6	8	2 kg box	6.0000	24	0	5	☐
14	Tofu	6	7	40 - 100 g pkgs.	23.2500	35	0	0	☐
15	Genen Shouyu	6	2	24 - 250 ml bottles	15.5000	39	0	5	☐
16	Pavlova	7	3	32 - 500 g boxes	17.4500	29	0	10	☐
17	Alice Mutton	7	6	20 - 1 kg tins	39.0000	0	0	10	☑
18	Carnarvon Tigers	7	8	16 kg pkg.	62.5000	42	0	0	☐
19	Teatime Chocolate Biscuits	8	3	10 boxes x 12 pieces	9.2000	25	0	5	☐

Die Verwendung der *AutoGenerateColumns*-Eigenschaft ist sicher recht praktisch, doch haben Sie in diesem Fall keinen Einfluss auf Anzahl, Reihenfolge und Aussehen der Spalten.

Das *DataGrid* selbst unterscheidet in diesem Fall lediglich zwischen Spalten der Typen *DataGridTextColumn* und *DataGridCheckBoxColumn*, deren Bedeutung sich bereits durch den Namen erklärt.

11.7.2 UI-Virtualisierung

Sicher interessiert es Sie auch, wie leistungsfähig das *DataGrid* ist. Erstellen Sie ruhig einmal eine Collection mit 1 000 000 Datensätzen und weisen Sie diese als *ItemsSource* zu. Sie werden feststellen, dass das Erzeugen der Collection wesentlich länger dauert als die Anzeige der Daten. Der Grund für dieses Verhalten basiert auf der UI-Virtualisierung, die mithilfe eines *VirtualizingStackPanel* als Layoutpanel innerhalb des *DataGrid* (auch *ListView*, *ListBox* etc.) verwendet wird.

Das *VirtualizingStackPanel* sorgt dafür, dass nur die gerade sichtbaren Einträge (bzw. die dazu notwendigen Controls) erzeugt werden. Was passiert, wenn dies nicht so ist, können Sie ganz einfach ausprobieren. Es genügt, wenn Sie das folgende Attribut in die Elementedefinition einfügen:

```
<DataGrid VirtualizingStackPanel.IsVirtualizing="False" Name="dataGrid1" />
```

Bitte besorgen Sie sich rechtzeitig eine Zeitung und eine Kanne Kaffee, wenn Sie versuchen wollen, eine große Collection an das *DataGrid* zu binden. Im extremsten Fall kommt es zur Meldung, dass der verfügbare Arbeitsspeicher nicht ausreicht. Die Ursache dürfte schnell klar werden, wenn Sie sich vorstellen, dass für jede erforderliche Zeile und alle angezeigten Spalten die entsprechenden Anzeige-Controls generiert werden müssen. Selbst bei unserer kleinen Datenbank wird man schon einen zeitlichen Unterschied feststellen.

11.7.3 Spalten selbst definieren

Gehen Ihnen die Möglichkeiten von *AutoGenerateColumns* nicht weit genug, können Sie alternativ auch selbst Hand anlegen und die einzelnen Spalten frei definieren. Setzen Sie in diesem Fall das Attribut *AutoGenerateColumns* auf *false* und fügen Sie die Spaltendefinitionen der *Columns*-Eigenschaft hinzu (die Reihenfolge der Definition entscheidet über die Anzeigereihenfolge).

Wir machen es uns im folgenden Beispiel etwas einfacher und definieren nur vier Spalten. Der Rest, denke ich, ist dann klar, wie es geht.

Beispiel 11.36: *DataGrid* mit einzeln definierten Spalten

XAML

```
    <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
            ItemsSource="{Binding}" Name="dataGrid1" >
```

Und hier folgen die Definitionen der einzelnen Spalten:

```
            <DataGrid.Columns>
```

Eine Textspalte erzeugen, Bindung an den Member *ProductID* (bitte auf komplette Groß-/Kleinschreibung achten) herstellen, die Kopfzeile mit „ID“ beschriften und eine Größenanpassung vornehmen:

```
        <DataGridTextColumn Binding="{Binding Path=ProductID}"
               Header="ID" Width="SizeToHeader" />
```

Gleiches für den Produktnamen (ohne Größenanpassung):

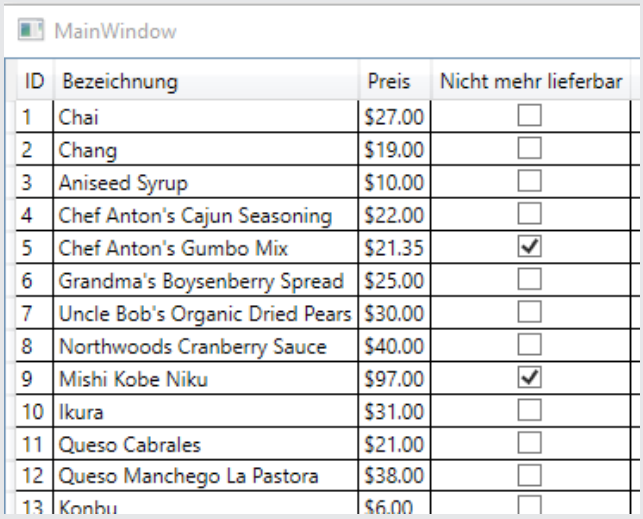
```
        <DataGridTextColumn Binding="{Binding Path=ProductName}"
                Header="Bezeichnung" />
```

Dann legen wir eine Spalte für den *UnitPrice* an. Diese Spalte wollen wir auch als Währung formatieren (*StringFormat=C*). Und zu guter Letzt die Spalte *Discontinued*, weil wir hier eine *DataGridCheckBox* verwenden wollen:

```
        <DataGridTextColumn Binding="{Binding UnitPrice, StringFormat=C}"
                Header="Preis" />
        <DataGridCheckBoxColumn Binding="{Binding Discontinued}"
                Header="Nicht mehr lieferbar" />
    </DataGrid.Columns>
  </DataGrid>
</StackPanel>
```

Ergebnis

Das erzeugte *DataGrid*:



Anmerkung

Im obigen Beispiel ist der Preis in Dollar angegeben. Wenn Sie die aktuelle Währung aus der Systemeinstellung wählen wollen, müssen Sie noch folgenden Code hinzufügen. Am besten machen Sie das in der überschriebenen *OnStart*-Methode der *App*, denn dann gilt das für die gesamte Anwendung:

```
public partial class App : Application
{
   protected override void OnStartup(StartupEventArgs e)
   {
      base.OnStartup(e);
      FrameworkElement.LanguageProperty.OverrideMetadata(
                  typeof(FrameworkElement),
                  new FrameworkPropertyMetadata(
```

```
                    XmlLanguage.GetLanguage(
                    CultureInfo.CurrentCulture.IetfLanguageTag)));
    }
}
```

HINWEIS: Sie können die Spaltenbreite auch mit „*“ angeben, in diesem Fall verwendet die Spalte den restlichen verfügbaren Platz.

Außer den Standard-Spaltentypen

- *DataGridTextColumn*,
- *DataGridCheckBoxColumn*,
- *DataGridComboBoxColumn*,
- *DataGridHyperlinkColumn*

steht auch die recht flexible *DataGridTemplateColumn* zur Verfügung. Welche Controls Sie hier einbinden (*Image*, *Chart*, *RichTextBox* etc.), bleibt Ihrer Fantasie überlassen. Auf diese Art und Weise kann man dann auch den Preis zum Beispiel rechtsbündig darstellen.

Weitere Gestaltungsmöglichkeiten bieten sich mit dem Ein- und Ausblenden der Trennlinien, der Konfiguration der Spaltenköpfe per Template usw.

11.7.4 Zusatzinformationen in den Zeilen anzeigen

Nicht alle Informationen sollen immer gleich in einem Grid sichtbar sein, vielfach werden Detailfenster etc. eingeblendet, um nach der Auswahl eines Datensatzes weitere Informationen anzuzeigen. An dieser Stelle bietet das *DataGrid* mit dem *RowDetailsTemplate* ein recht interessantes Feature, versetzt Sie dieses Template doch in die Lage, unter bestimmten Umständen (*RowDetailsVisibilityMode*-Eigenschaft) zusätzliche Inhalte einzublenden.

Beispiel 11.37: Verwendung von *RowDetailsTemplate*

XAML

Zunächst müssen Sie bestimmen, wann die Details eingeblendet werden sollen:

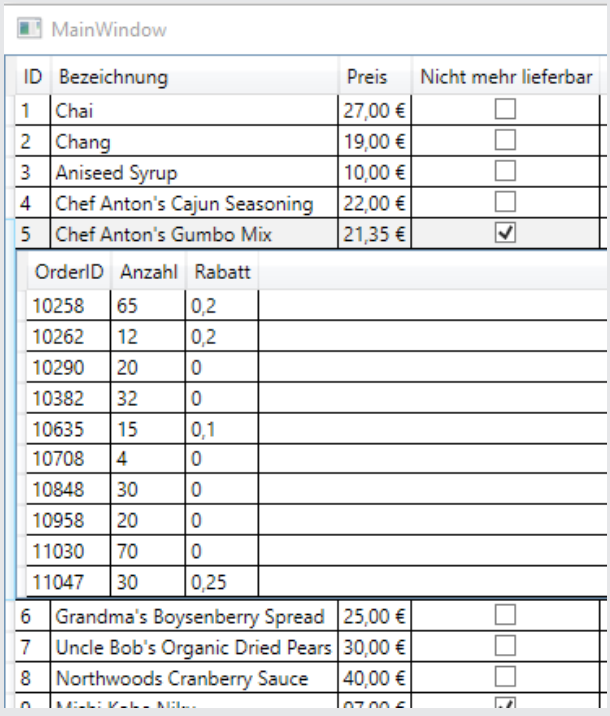
```
<DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
          ItemsSource="{Binding}"
          RowDetailsVisibilityMode="VisibleWhenSelected" >
 <DataGrid.Columns>
...
 </DataGrid.Columns>
```

Nach der Spaltendefinition können Sie das *RowDetailsTemplate* einfügen und mit den gewünschten Informationen füllen. Wir wollen hier alle Bestellungsdetails mit diesem Produkt anzeigen:

```
 <DataGrid.RowDetailsTemplate>
  <DataTemplate>
```

```
    <DataGrid ItemsSource="{Binding Order_Details}"
              AutoGenerateColumns="False">
      <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding OrderID}" Header="OrderID"/>
        <DataGridTextColumn Binding="{Binding Quantity}" Header="Anzahl"  />
        <DataGridTextColumn Binding="{Binding Discount}" Header="Rabatt" />
      </DataGrid.Columns>
    </DataGrid>
  </DataTemplate>
 </DataGrid.RowDetailsTemplate>
</DataGrid>
```

Ergebnis



Mit *RowDetailsVisibilityMode* bestimmen Sie, wie die Zeilendetails angezeigt werden. Standardwert ist *Collapsed* (nicht sichtbar), alternativ steht *Visible* (immer sichtbar) oder *VisibleWhenSelected* zur Verfügung (nur die aktuelle Zeile).

11.7.5 Vom Betrachten zum Editieren

Auch wenn die umfangreichen Anzeigeoptionen das *DataGrid* für diverse Aufgaben prädestinieren, eine Hauptaufgabe dürfte in den meisten Fällen auch das Editieren der Inhalte sein.

Grundsätzlich entscheidet zunächst die übergreifende Eigenschaft *IsReadOnly* über die Fähigkeit, Inhalte des *DataGrid*s zu editieren oder nur zu betrachten. Gleiches gilt auch auf Spaltenebene, auch hier können Sie mit *IsReadOnly* darüber entscheiden, welche Spalten editierbar sind und welche nicht. Zusätzlich unterstützen Sie diverse Ereignisse vor, während und nach dem Editiervorgang (*BeginningEdit*, *PreparingCellForEdit*, *CellEditEnding*, …).

11.8 Praxisbeispiel – Collections in Hintergrundthreads füllen

In den vorhergehenden Beispielen haben wir es uns recht einfach gemacht. Eine Collection wurde erzeugt, gefüllt und angezeigt. So weit, so gut, aber was, wenn das Erzeugen der Collection etwas länger dauert? Ein kleines Beispielprogramm zeigt das Problem und natürlich auch die Lösung dafür. Dabei trennen wir aber zwischen der bisherigen Lösung und einer mit .NET 4.5 eingeführten Neuerung.

Oberfläche

Ein einfaches *Window* mit einigen Schaltflächen und einer *ListView* zur Anzeige der Daten:

```
<Window x:Class="PraxisbeispielDatenbindung.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:PraxisbeispielDatenbindung"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <DockPanel>
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <Button Click="ButtonVT_Click">Laden Vordergrund-Thread</Button>
            <Button Click="ButtonHT_Click">Laden Hintergrund-Thread</Button>
            <Button Click="ButtonHL_Click">Laden Hintergrund Lösung</Button>
            <Button Click="ButtonLN_Click">Laden neu</Button>
        </StackPanel>
        <ListView Name="listView1" IsSynchronizedWithCurrentItem="True"
                  ItemsSource="{Binding}" VirtualizingPanel.IsVirtualizing="True"
/>
    </DockPanel>
</Window>
```

Das Problem

Stellen Sie sich folgendes Szenario vor: Sie füllen eine Liste von *Person*-Objekten[3], leider dauert der Abruf jedes einzelnen Objekts etwas länger:

[3] Definition siehe Abschnitt 11.2.3.

```
public partial MainWindow : Window
{
    public ObservableCollection<Person> Abteilung { get; set; }

    public MainWindow()
    {
        InitializeComponent();
    }

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        Abteilung = new ObservableCollection<Person>();
        DataContext = Abteilung;
    }

    private void Datenabrufen()
    {
        Abteilung.Clear();
        for (int i = 0; i < 100; i++)
        {
```

Hier simulieren wir eine Zeitverzögerung, z. B. eine langsame Datenverbindung:

```
            System.Threading.Thread.Sleep(100);
            Abteilung.Add(new Person
            {
                Nachname = $"Müller{i}",
                Vorname = "Thomas"
            });
        }
    }

    private void ButtonVT_Click(object sender, RoutedEventArgs e)
    {
        Datenabrufen();
    }
```

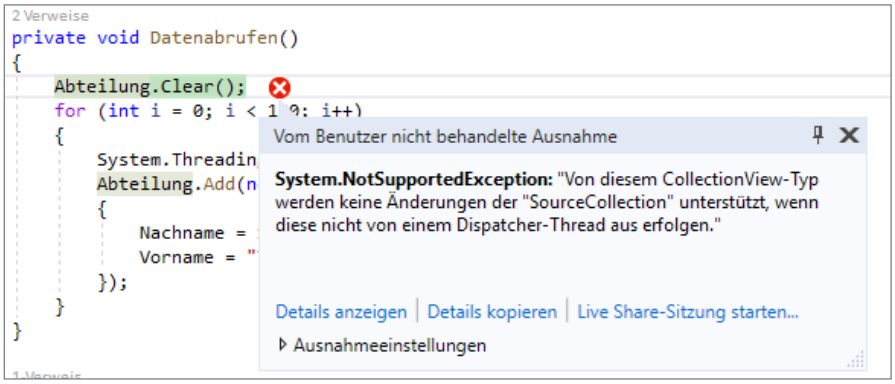
Starten Sie die Anwendung, werden Sie nach einem Klick auf die Schaltfläche feststellen, dass Ihre Anwendung „einfriert“. Diese Lösung wollen Sie dem Endanwender sicher nicht zumuten. Was liegt also näher, als diese Aufgabe in einen Hintergrundthread zu verlagern.

HINWEIS: Eine ausführliche Erläuterung zu Tasks und Threads folgt in den Kapiteln 14 und 15.

Gesagt, getan, wir kapseln obigen Methodenaufruf in einem extra Thread:

```
    private void ButtonHT_Click(object sender, RoutedEventArgs e)
    {
        Task.Run(Datenabrufen);
    }
```

Doch nach einem Start der Anwendung werden Sie schnell wieder auf den Boden zurückgeholt:



Sie können auf die Collection nicht per Hintergrundthread zugreifen. Das ist erst mal ein Show-Stopper. Doch es gibt zwei Lösungen:

- Laden einer extra Collection im Hintergrund und kopieren dieser Collection in den Vordergrund. Nachfolgend Abgleich mit der gebundenen Collection.
- Laden der Daten im Hintergrund, Einfügen der einzelnen Einträge durch jeweiligen Wechsel in den Vordergrundthread.

Die zweite Lösung ist mit häufigen Threadwechseln verbunden, wir sehen uns also die erste Lösung näher an.

Lösung (bis .NET 4.0)

Wir lagern das Laden der Daten in eine Funktion aus, die eine komplette Liste zurückgibt:

```
        private ObservableCollection<Person> Datenabrufen_alteLoesung()
        {
            ObservableCollection<Person> threadAbteilung =
                 new ObservableCollection<Person>();
            for (int i = 0; i < 100; i++)
            {
                // simuliert Laden aus der Quelle
                System.Threading.Thread.Sleep(100);
                threadAbteilung.Add(new Person
                {
                    Nachname = $"Müller{i}",
                    Vorname = "Thomas"
                });
            }
            return threadAbteilung;
        }
```

Unser Aufruf:

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

Anzeige, dass der Nutzer warten soll:

```
            Cursor = Cursors.Wait;
```

In einem extra Thread werden die Daten geladen:

```
            Task.Factory.StartNew <ObservableCollection<Person>>(
                                Datenabrufen_alteLoesung).ContinueWith(t =>
            {
```

Ist dies erfolgt, kopieren wir die Daten in die gebundene Liste:

```
                Abteilung.Clear();
                foreach (var s in t.Result)
                    klasse.Add(s);
```

Und blenden die Sanduhr aus:

```
                this.Cursor = null;
            }, TaskScheduler.FromCurrentSynchronizationContext());
        }
```

Test

Nach dem Start wird die „Sanduhr“ angezeigt, nach einigen Sekunden ist die Liste gefüllt. Wie Sie sehen, muss sich der Nutzer auch hier gedulden, die Oberfläche bleibt in dieser Zeit aber voll bedienbar.

Lösung (ab .NET 4.5)

.NET 4.5 bietet die Möglichkeit, Collections für die gleichzeitige Bearbeitung in Threads quasi anzumelden. Nutzen Sie dazu einen Aufruf der Methode *EnableCollectionSynchronization*.

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

Anmelden der Collection – wir übergeben noch das aktuelle Window-Objekt als Sperrobjekt[4]:

```
            BindingOperations.EnableCollectionSynchronization(Abteilung, this);
```

Wir rufen die Daten per extra Thread ab:

```
            Task.Run(Datenabrufen);
        }
```

Test

Nach dem Start werden Sie feststellen, dass die Oberfläche beweglich bleibt und dass die Daten „tröpfchenweise“ in die Liste geladen werden. Sie können beim Füllen quasi zusehen – eine einfache und recht elegante Lösung für das Einlesen größerer Datenmengen.

HINWEIS: Eine alternative Lösung wäre auch noch die Verwendung des *async-await*-Musters. Dazu mehr in Abschnitt 14.9.

[4] Auch jede andere Objekt-Instanz ist geeignet.

Teil III: Technologien

- Zugriff auf das Dateisystem (Kapitel 12)
- Dateien lesen und schreiben (Kapitel 13)
- Asynchrone Programmierung (Kapitel 14)
- Die Task Parallel Library (Kapitel 15)
- Fehlersuche und Behandlung (Kapitel 16)
- JSON und XML in Theorie und Praxis (Kapitel 17)
- Einführung in ADO.NET und Entity Framework (Kapitel 18)
- Weitere Techniken (Kapitel 19)

Index

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z