AUSGABE 04/2017

DATENBANK ENTWICKLER

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

TOP-THEMEN:

C#-BASICS	Basics: ObservableCollection	SEITE 3
USERINTERFACE	Neuer Eintrag in ComboBox	SEITE 18
USERINTERFACE	m:n-Beziehung mit Listenfeld	SEITE 25
DATENZUGRIFF	LINQ to Entities: Methodensyntax	SEITE 42
ANWENDUNGEN	EDM: Backend ändern	SEITE 53

André Minhorst Verlag

C#-GRUNDLAGEN	Basics: ObservableCollection	3
	Type Converter in WPF/C#	12
WPF-GRUNDLAGEN	Fehlerhafte Bindungen prüfen	16
BENUTZEROBERFLÄCHE MIT WPF	Neuer Eintrag in ComboBox	18
	m:n-Beziehung mit Listenfeld	25
DATENZUGRIFFSTECHNIK	LINQ to Entities-Beispiele in der Methodensyntax	42
INTERAKTIV	EDM: SQLite aktuell halten	52
ANWENDUNGSENTWICKLUNG	EDM: Backend ändern	53
SERVICE	Impressum	2

DOWNLOAD

Die Downloads zu dieser Ausgabe finden Sie unter folgendem Link:

http://www.amvshop.de

Klicken Sie dort auf **Mein Konto**, loggen Sie sich ein und wählen dann **Meine Sofortdownloads**.

Impressum

DATENBANKENTWICKLER
© 2017 André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

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

Basics: ObservableCollection

Unter WPF gibt es einige Mechanismen, welche die Bindung der Steuerelemente an die zugrunde liegenden Daten in einem gewissen Rahmen automatisieren. Diese werden durch die Programmierung bestimmt – entweder durch die Verwendung bestimmter Schnittstellen für Eigenschaften oder auch durch entsprechende Auflistungstypen, die dann als Datenquelle etwa für Listen-Elemente verwendet werden. Mit der PropertyChanged-Schnittstelle haben wir schon die Synchronisierung zwischen den Attributen der XAML-Definition und Eigenschaften in den Code-Klassen besprochen (siehe Artikel »Basics: PropertyChanged«). In diesem Artikel schauen wir uns nun den Auflistungstyp ObservableCollection an.

Beispielanwendung

Die Beispielanwendung verwendet die auch in den anderen Artikeln genutzte SQLite-Datenbank **Bestellverwaltung.db**. Wir wollen daraus die Tabelle **Kunden** nutzen, um die Unterschiede zwischen einer normalen Liste (**List**) und einer **ObservableCollection** zu demonstrieren.

Die **ObservableCollection**-Klasse ist eine Alternative beispielsweise zur **List**-Klasse. Der wesentliche Unterschied ist, dass die **ObservableCollection**-Klasse externe Objekte, die den Inhalt einer **ObservableCollection** anzeigen, über Änderungen informiert.

Änderungen können dabei etwa das Hinzufügen oder Entfernen von Elementen sein. Beispiele für Steuerelemente, die Sie an Objekte auf Basis der Klasse **ObservableCollection** binden können, sind etwa das **ListBox**-Element oder das **DataGrid**-Element.

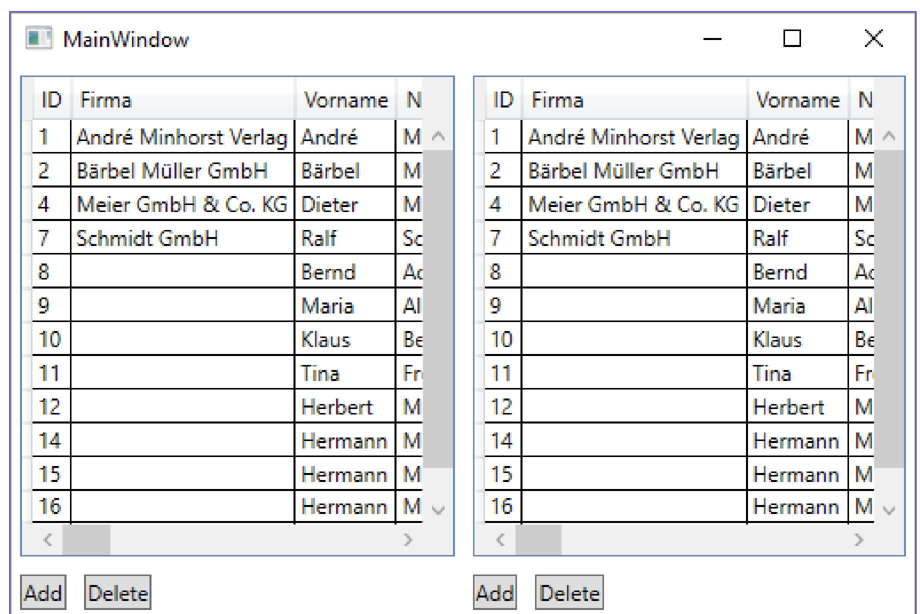

Bild 1: Beispiel für an ein **List**- und ein **ObservableCollection**-Objekt gebundene **DataGrid**-Steuerelemente

Was aber heißt überhaupt »informieren« in diesem Zusammenhang? Wenn Sie etwa unter Access ein Listenfeld an eine Tabelle gebunden und dann einen Datensatz dieser Tabelle gelöscht haben, wurde der Datensatz nach dem Aufruf der **Requery**-Methode des Listenfeldes auch aus dem Listenfeld entfernt. In einem Formular in der Datenblattansicht führte das Löschen eines Datensatzes direkt zum Entfernen dieses Eintrags aus der Ansicht – sehr praktisch, aber auch logisch, da hier direkt die Tabelle an das Formular gebunden war. Unter WPF ist das alles etwas anders – hier binden Sie ja beispielsweise nicht direkt an Tabellen

oder Abfragen, sondern an Auflistungen wie das **List**-Objekt, **Collection** oder **ObservableCollection**, die Sie zuvor noch mit den Daten aus der Datenbank füllen. Wenn Sie nun Änderungen an einem **List**- oder **Collection**-Objekt durchführen, wie es etwa geschieht, wenn Sie etwa einen neuen Eintrag hinzufügen oder einen Eintrag löschen, dann wirken sich die Änderungen zwar auf die Einträge des **List**- oder **Collection**-Objekts aus, aber sie schlagen sich nicht in der Benutzeroberfläche nieder.

Und hier tritt das **ObservableCollection**-Objekt auf den Plan: Wenn Sie diesem einen neuen Eintrag hinzufügen oder einen Eintrag entfernen, löst dies ein Ereignis aus, das automatisch von bestimmten Elementen der Benutzeroberfläche wie etwa dem **ListBox**- oder dem **DataGrid**-Element implementiert wird und dafür sorgt, dass die Ansicht aktualisiert wird. Um uns dies einmal an einem Beispiel anzusehen, haben wir das Fenster aus Bild 1 erstellt.

Die Definition dieser Steuerelemente finden Sie in Listing 1. Das erste DataGrid namens **dgList** ist an das Objekt **KundenList** der Code behind-Klasse gebunden, das zweite DataGrid namens **dgObservableCollection** an das Objekt **KundenObservable-Collection**. Zu jedem **DataGrid**-Steuerelement haben wir jeweils eine Hinzufügen- und eine Entfernen-Schaltfläche hinzugefügt, mit denen Sie per Code jeweils einen neuen Datensatz anlegen beziehungsweise den aktuell markierten Datensatz aus dem zugrunde liegenden Auflistungsobjekt entfernen. Wir werden dann später erkennen, wo die Unterschiede zwischen dem **List**- und dem **ObservableCollection**-Objekt liegen.

```
<Window x:Class="ObservableCollection.MainWindow" ...  Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <DataGrid x:Name="dgList" Grid.Column="0" Margin="5" ItemsSource="{Binding KundenList}"></DataGrid>
        <DataGrid x:Name="dgObservableCollection" Margin="5" Grid.Column="1"
            ItemsSource="{Binding KundenObservableCollection}"></DataGrid>
        <StackPanel Grid.Row="1" Grid.Column="0" Orientation="Horizontal">
            <Button x:Name="btnListAdd" Margin="5" Click="btnListAdd_Click">Add</Button>
            <Button x:Name="btnListDelete" Margin="5" Click="btnListDelete_Click">Delete</Button>
        </StackPanel>
        <StackPanel Grid.Row="1" Grid.Column="1" Orientation="Horizontal">
            <Button x:Name="btnObservableCollectionAdd" Margin="5" Click="btnObservableCollectionAdd_Click">Add</Button>
            <Button x:Name="btnObservableCollectionDelete" Margin="5"
                Click="btnObservableCollectionDelete_Click">Delete</Button>
        </StackPanel>
    </Grid>
</Window>
```

Listing 1: Definition der beiden **DataGrid**-Elemente und der Schaltflächen zum Hinzufügen und Löschen der Elemente

DataGrid-Elemente füllen

Damit die beiden **DataGrid**-Elemente gefüllt werden, haben wir der Code behind-Klasse etwas Code hinzugefügt. Im allgemeinen Teil haben wir dazu zunächst eine Objektvariable für den Datenbankkontext deklariert:

```
BestellverwaltungEntities dbContext;
```

Dann benötigen wir ein **List**-Objekt namens **kundenList**, das Elemente des Typs **Kunde** aufnehmen soll und über die öffentliche Eigenschaft **KundenList** per **get** und **set** verfügbar gemacht wird:

```
private List<Kunde> kundenList;
public List<Kunde> KundenList {
    get {
        return kundenList;
    }
    set {
        kundenList = value;
    }
}
```

Das Gleiche haben wir für ein Auflistungs-Element des Typs **ObservableCollection** durchgeführt. Dieses heißt allerdings **kundenObservableCollection** und wird wie folgt definiert:

```
private ObservableCollection<Kunde> kundenObservableCollection;
public ObservableCollection<Kunde> KundenObservableCollection {
    get {
        return kundenObservableCollection;
    }
    set {
        kundenObservableCollection = value;
    }
}
```

Damit Sie das **ObservableCollection**-Objekt überhaupt nutzen können, müssen Sie dieses zunächst über den passenden Namespace in der Klasse bekannt machen. Dies erledigen Sie mit der folgenden Zeile, die Sie zu den bestehenden **using**-Anweisungen hinzufügen:

```
using System.Collections.ObjectModel;
```

Damit kommen wir zur Konstruktor-Methode, die beim Anzeigen des Fensters ausgelöst wird. Sie erstellt das Datenbankkontext-Objekt **dbContext** und liest zunächst die Elemente der Auflistung **Kunden** des Datenbankkontexts in die Liste **kundenList** ein. Dann erledigt sie dies auch für die Liste **kundenObservableCollection**. Schließlich weist sie der Eigenschaft **DataContext**

einen Verweis auf die Code behind-Klasse selbst zu, damit die Elemente der Benutzeroberfläche die hier definierten Eigenschaften **KundenList** und **ObservableCollectionList** nutzen können, um die enthaltenen Daten anzuzeigen:

```
public MainWindow() {
    InitializeComponent();
    dbContext = new BestellverwaltungEntities();
    kundenList = dbContext.Kunden.ToList();
    kundenObservableCollection = new ObservableCollection<Kunde>(dbContext.Kunden);
    DataContext = this;
}
```

Element hinzufügen

Nun legen wir die Prozedur an, die durch das **Click**-Ereignis der ersten Schaltfläche **btnListAdd** ausgelöst wird. Diese erstellt ein neues **Kunde**-Objekt, weist diesem ein paar Eigenschaftswerte zu und fügt es dann als Parameter der **Add**-Methode zur Auflistung **kundenList** hinzu:

```
private void btnListAdd_Click(object sender, RoutedEventArgs e) {
    Kunde kunde = new Kunde();
    kunde.AnredeID = 1;
    kunde.Vorname = "Hermann";
    kunde.Nachname = "Müller";
    kundenList.Add(kunde);
}
```

Die Methode, die durch die Schaltfläche **btnObservableCollectionAdd_Click** ausgelöst wird, hat überwiegend identische Anweisungen, daher hier nur die gekürzte Fassung mit dem Aufruf der **Add**-Methode des **ObservableCollection**-Objekts **kundenObservableCollection**:

```
private void btnObservableColl-
ectionAdd_Click(object sender,
RoutedEventArgs e) {
    ...
    kundenObservableCollection.
Add(kunde);
}
```

Beide Methoden erledigen grundsätzlich die gleichen Schritte für ihr

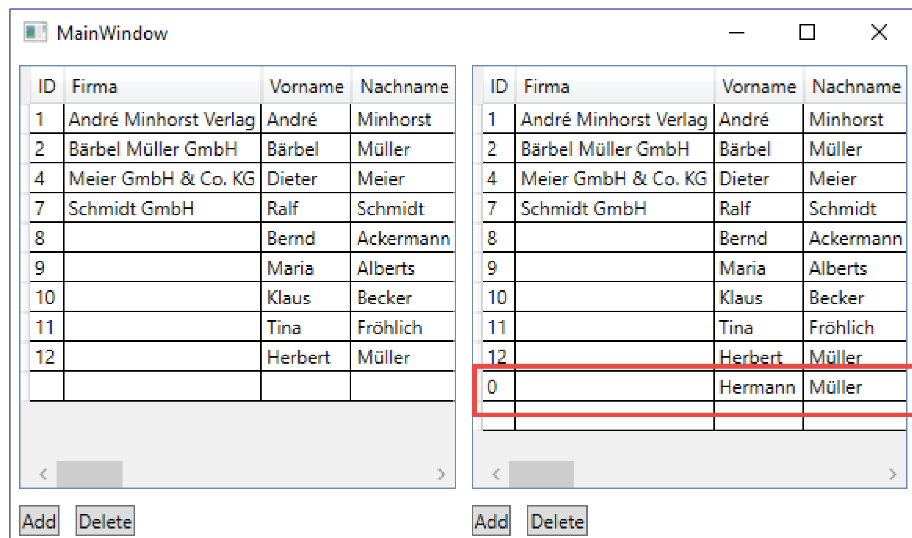

Bild 2: Nur in dem an das **ObservableCollection**-Objekt gebundene **DataGrid** taucht der neue Eintrag auf.

jeweiliges Auflistungs-Objekt. Dennoch sieht das Fenster nach dem Anklicken der beiden **Add**-Schaltflächen wie in Bild 2 aus. Das zeigt: Offensichtlich sorgt das **DataGrid**-Element nur im Zusammenspiel mit dem **ObservableCollection**-Objekt für eine direkte Aktualisierung der Anzeige nach der Änderung der enthaltenen Daten. Übrigens: Wir könnten noch mit den folgenden beiden Anweisungen am Ende der Methode **btnObservableCollectionAdd_Click** dafür sorgen, dass statt der **0** im Feld **ID** direkt der Wert erscheint, der für den neuen Datensatz in der Datenbank angelegt wird:

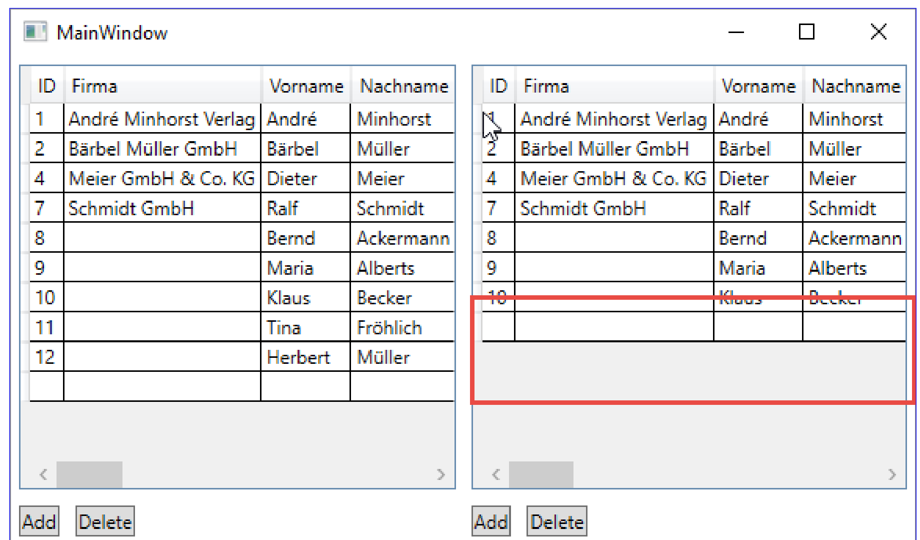

Bild 3: Nur in dem an das **ObservableCollection**-Objekt gebundene **DataGrid** werden die aus der Auflistung gelöschten Einträge als gelöscht angezeigt.

```
dbContext.Kunden.Add(kunde);
dbContext.SaveChanges();
```

Aktualisierungen beim Löschen von Einträgen

Um das Verhalten auch noch beim Entfernen von Einträgen aus der jeweils zugrunde liegenden Liste zu überprüfen, legen wir noch die beiden Ereignismethoden für die Schaltflächen **btnListDelete** und **btnObservableCollectionDelete** an. Die für das **List**-Auflistungselement sieht so aus:

```
private void btnListDelete_Click(object sender, RoutedEventArgs e) {
    Kunde delete = (Kunde)dgList.SelectedItem;
    kundenList.Remove(delete);
}
```

Und hier die Methode für die Schaltfläche **btnObservableCollectionDelete**:

```
private void btnObservableCollectionDelete_Click(object sender, RoutedEventArgs e) {
    Kunde delete = (Kunde)dgObservableCollection.SelectedItem;
    kundenObservableCollection.Remove(delete);
}
```

Auch hier bestätigt sich das bereits vom Hinzufügen bekannte Verhalten. Das Löschen von einigen Elementen aus der den **DataGrid**-Elementen zugrunde liegenden Auflistungsobjekten wird nur in dem an das **ObservableCollection**-Objekt gebundene **DataGrid**-Element angezeigt (siehe Bild 3).

Type Converter in WPF/C#

Manche Eigenschaften eines .xaml-Dokuments sind sehr flexibel. Für die Eigenschaft Margin beispielsweise können Sie ganz verschiedenen Werte angeben – zum Beispiel einfach den Wert 5, um Abstände in alle vier Richtungen zu erhalten oder auch die Zeichenkette 5,0,5,0, um nur einen linken und einen rechten Abstand abzubilden. Auch andere Eigenschaften nehmen durchaus unterschiedliche Werte entgegen, zum Beispiel solche zur Angabe von Farben. Wenn man genauer hinsieht, stellt man schnell fest, dass man da Attributen, die völlig anderen Typs sind, ein String-Literal zuweist. Warum das hier gelingt, beschreibt der vorliegende Artikel.

Type Converter am Beispiel von Abständen

Ein Beispiel für die Angabe eines Strings für ein Attribut mit einem anderen Datentyp als String ist das Attribut **Margin**.

Wenn wir einmal einen Button nur mit einem Rand ausstatten, der laut unserer Angabe den Abstand **5** von der linken und rechten und den Abstand **10** von der oberen und unteren Kante des übergeordneten Elements aufweist, dann erhalten wir einen Button wie in Bild 1. Der Code sieht indes so aus:

```
<Button Margin="5,10,5,10">
```

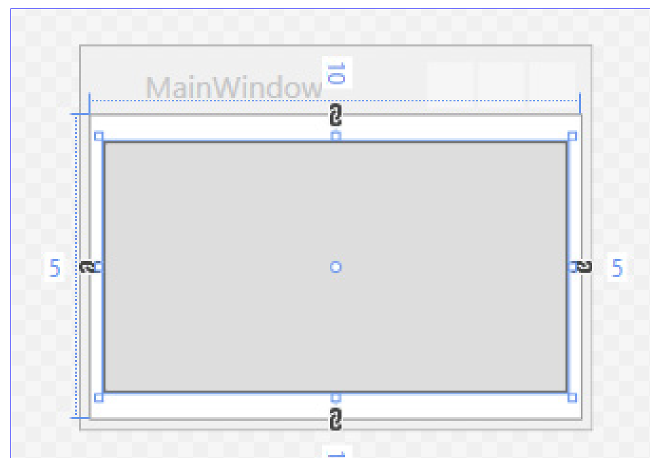

Bild 1: Ein **Button**-Element mit verschiedenen **Margin**-Werten

Wenn Sie im **.xaml**-Code einmal mit der rechten Maustaste auf das Attribut **Margin** klicken und dann den Eintrag **Gehe zu Definition** aus dem Kontextmenü auswählen, erscheint der Dialog aus Bild 2. Hier wird schnell deutlich, dass Margin keinesfalls den Datentyp **String** aufweist, sondern vielmehr den Datentyp **System.Windows.Thickness**. Wir müssten dieser Eigenschaft also normalerweise eine Struktur des Typs **Thickness** übergeben! Diese

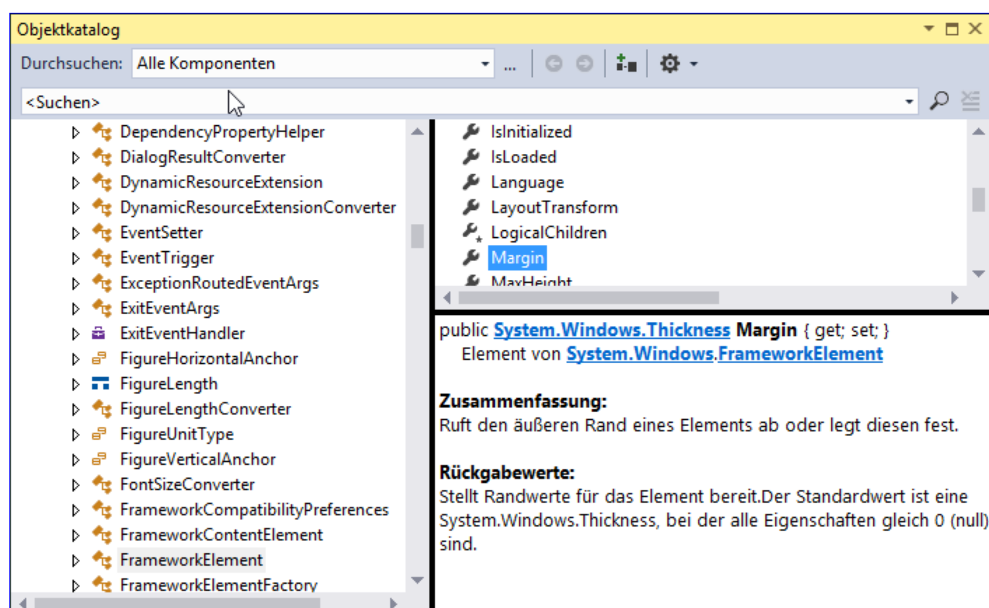

Bild 2: Definition der Eigenschaft **Margin**

Struktur weist wiederum die vier Eigenschaften **Bottom**, **Left**, **Right** und **Top** auf.

Margin unter C#

Um zu verdeutlichen, dass wir hier grundsätzlich einen nicht passenden Wert übergeben, wenn wir einen Zahlenwert oder auch einen String wie **5,10,5,10** angeben, schauen wir uns das Anlegen eines Buttons mit dieser Eigenschaft unter C# an.

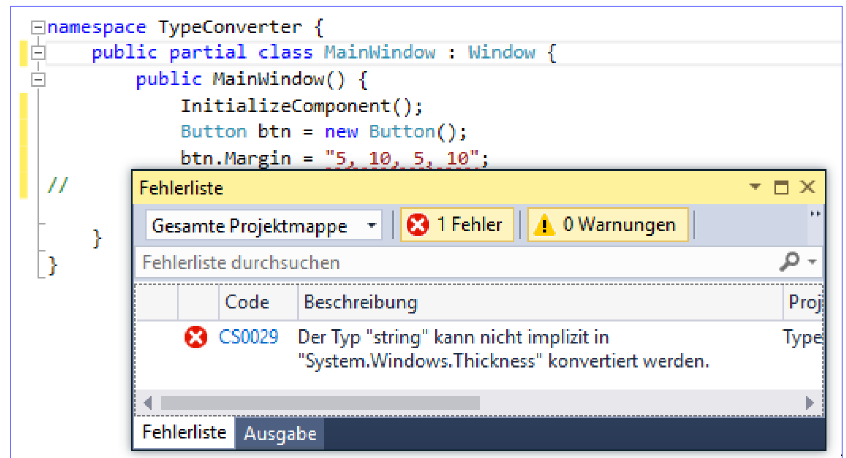

Bild 3: Unter C# kann der Eigenschaft **Margin** kein String übergeben werden.

Dazu erstellen wir mit der **new**-Anweisung ein neues Objekt des Typs **Button** und referenzieren dieses mit der Variablen **btn**. Danach weisen wir diesem genau wie in im **.xaml**-Code den Wert **„5, 10, 5, 10“** für die Eigenschaft **Margin** zu. Der gemeldete Fehler zeigt, dass **String** nicht in **Thickness** konvertiert werden kann (siehe Bild 3).

Was aber ist nun unter C# anders als im **.xaml**-Code? Der Unterschied ist, dass im **.xaml**-Code automatisch ein sogenannter Type Converter agiert und den angegebenen String in die gewünschten Werte umwandelt und zuweist.

Logisch, dass Sie hier nun auch nicht irgendeine Zeichenkette angeben können, sondern dass diese durchaus einen bis vier durch Kommata getrennte Zahlenwerte enthalten muss.

Wie geben wir die Werte nun unter C# ein? Dies erledigen wir, indem wir zunächst ein Objekt des Typs **Thickness** erstellen und seinen Eigenschaften **Bottom**, **Top**, **Left** und **Right** die gewünschten Zahlenwerte zuweisen.

Danach weisen wir der Eigenschaft **Margin** das Objekt **margin** mit dem Typ **Thickness** (damit wir den Button als Kindelement des Grids festlegen können, müssen wir diesem mit **x:Name=“MyGrid“** noch einen Namen zuweisen):

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
        Button btn = new Button();
        Thickness margin = new Thickness();
        margin.Bottom = 10;
        margin.Top = 10;
        margin.Left = 5;
        margin.Right = 5;
        btn.Margin = margin;
        MyGrid.Children.Add(btn);
    }
}
```

Fehlerhafte Bindungen prüfen

Wenn Sie mit C# programmieren, können Sie beim Debuggen bequem herausfinden, warum etwas nicht so funktioniert, wie Sie es sich vorstellen. Bei der Datenbindung von WPF-Elementen wird es komplizierter: Wenn etwa ein an ein Feld eines Objekts gebundenes Steuerelement nicht den gewünschten Wert anzeigt, kann dies verschiedene Gründe haben. Wenn das gebundene Element nicht gefunden werden kann, gibt es keine Fehlermeldung, und es wird auch schwierig, herauszufinden, woran es liegt. Dieser Artikel stellt ein paar Möglichkeiten vor, Licht ins Dunkel der Bindung zu bringen.

Verweis ins Leere

Manchmal ist man einfach betriebsblind oder unkonzentriert und gibt bei einer Bindung den Namen der falschen Eigenschaft ein. Dann bleibt das entsprechende Steuerelement schlicht leer! Wie aber finden Sie heraus, was genau nicht funktioniert? Der erste Ansatz ist: Die Ausgabe auf eventuelle Meldungen untersuchen. Nehmen wir den folgenden falschen Code zur Definition eines Grid-Elements mit Textfeldern – wie Sie sehen, kann schon ein Fehler bei der Groß-/Kleinschreibung zum Fehler führen:

```
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=VorName}"></TextBox>
```

Als Ergebnis bleibt das Steuerelement leer, denn die Eigenschaft der an das Fenster gebundenen Klasse heißt nicht **VorName**, sondern **Vorname** (siehe Bild 1):

```
public class Kunde {
    public string Vorname { get; set; }
    ...
}
```

Bild 1: Das Feld **Vorname** bleibt erwartungsgemäß leer.

Nun stellen wir uns dumm und wollen herausfinden, warum das Feld keine Daten anzeigt. Eine Ausnahme ist auch nicht aufgetreten. Wertvolle Informationen liefert in diesem Fall das Ausgabe-Fenster (siehe Bild 2). Es liefert die folgende Meldung:

```
System.Windows.Data Error: 40 : BindingExpression path error: 'VorName' property not found on 'object' ''Kunde' (HashCode=16049999)'. BindingExpression:Path=VorName; DataItem='Kunde' (HashCode=16049999); target element is 'TextBox' (Name=''); target property is 'Text' (type 'String')
```

Hier erfahren wir also ziemlich genau, dass es die Eigenschaft **VorName** nicht im Objekt **Kunde** gibt.

Mehr Informationen ausgeben

Durch eine kleine Ergänzung des XAML-Codes können Sie noch eine Reihe weiterer Informationen im Ausgabefenster ausgeben. Dazu erweitern Sie zunächst das Window-Element um einen weiteren Namespace:

```
<Window x:Class="BindungenUntersuchen.MainWindow" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

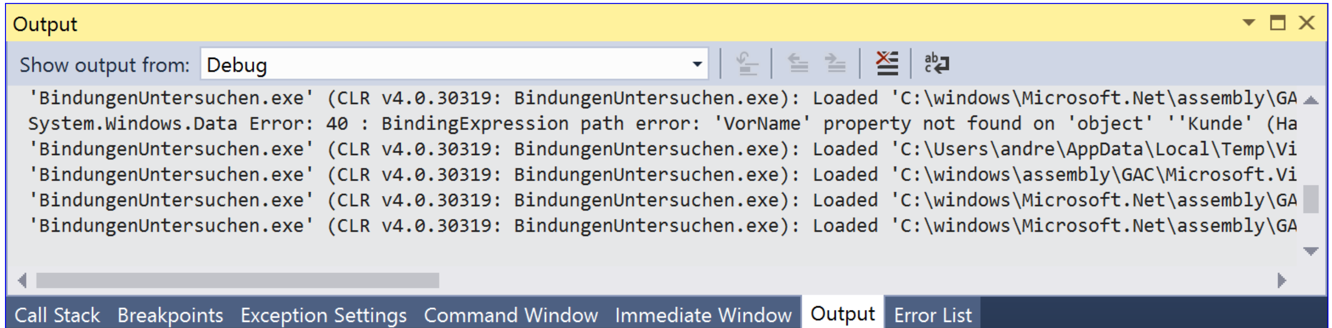

Bild 2: Das Ausgabe-Fenster mit der gesuchten Meldung

```
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:BindungenUntersuchen" xmlns:diag="clr-namespace:System.Diagnostics;assembly=WindowsBase"
mc:Ignorable="d" Title="MainWindow" Height="350" Width="525">
```

Außerdem fügen wir dem Element, das wir untersuchen wollen, ein zusätzliches Attribut zum Binding hinzu:

```
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=VorName, diag:PresentationTraceSources.TraceLevel=High}"></TextBox>
```

Danach sieht die Ausgabe im Ausgabefenster wie folgt aus:

```
System.Windows.Data Warning: 56 : Created BindingExpression (hash=20031746) for Binding (hash=28278595)
System.Windows.Data Warning: 58 :   Path: 'VorName'
System.Windows.Data Warning: 60 : BindingExpression (hash=20031746): Default mode resolved to TwoWay
System.Windows.Data Warning: 61 : BindingExpression (hash=20031746): Default update trigger resolved to LostFocus
System.Windows.Data Warning: 62 : BindingExpression (hash=20031746): Attach to System.Windows.Controls.TextBox.Text
(hash=10598606)
System.Windows.Data Warning: 67 : BindingExpression (hash=20031746): Resolving source
System.Windows.Data Warning: 70 : BindingExpression (hash=20031746): Found data context element: TextBox (hash=10598606)
(OK)
System.Windows.Data Warning: 71 : BindingExpression (hash=20031746): DataContext is null
System.Windows.Data Warning: 65 : BindingExpression (hash=20031746): Resolve source deferred
```

Ausnahme bei Bindungsfehlern?

Aber könnte man solche fehlenden Informationen nicht etwas prominenter darstellen als in der mitunter recht langen Ausgabeliste einer Anwendung – beispielsweise als Ausnahme, wie sie sonst bei Fehlern angezeigt wird? Nein, dazu gibt es leider keine gängige Methode.

Zusammenfassung

Sollte sich einmal ein Steuerelement nicht mit den erwarteten Daten füllen, wissen Sie nun zumindest, an welcher Stelle Sie weitere Informationen zum Fehlen der Daten finden können.

Neuer Eintrag in ComboBox

Manchmal zeigen Kombinationsfelder einfach nur die verknüpften Werte einer Lookup-Tabelle an – wie beispielsweise bei der Kategorie eines Produktes. Die Tabelle mit den Produkten ist dann über ein Fremdschlüsselfeld mit der Lookup-Tabelle mit den Kategorien verknüpft, worüber dann die gewünschte Kategorie ausgewählt wird. Was aber, wenn man mal eben einen neuen Wert in die Lookup-Tabelle schreiben will? Lässt sich das überhaupt erledigen, ohne dass der Benutzer einen Dialog zur Eingabe der Daten der neuen Kategorie öffnet? Dieser Artikel bringt die Antwort auf diese Frage.

Unter Access ist dies ein sehr häufig genutztes Feature: Wenn ein Kombinationsfeld die Daten eines Lookup-Feldes zur Auswahl angeboten hat und der Benutzer einen neuen Eintrag zu der zugrunde liegenden Tabelle hinzufügen wollte, hat er einfach den gewünschten Wert in das Kombinationsfeld eingetragen und schon war der neue Datensatz mit den angegebenen Wert in der Tabelle gespeichert. Dazu gab es ein praktisches Ereignis namens **Bei nicht in Liste**, welches dann etwa mit Code gefüllt werden konnte, der den Benutzer fragte, ob der neue Eintrag tatsächlich in der Datenbank gespeichert und als neuer Wert genutzt werden sollte. Auch unter WPF und C# sollte es doch eine Möglichkeit geben, dies zu realisieren.

In unserer Beispielanwendung **Bestellverwaltung** haben wir dies bisher etwas komplizierter für den Benutzer gemacht. Die Daten der Tabelle **Kategorien** konnte der Benutzer für ein Produkt zwar komfortabel per Kombinationsfeld auswählen, aber wenn er einen neuen Eintrag zur Tabelle **Kategorien** hinzufügen wollte, müsste er schon noch einen eigenen Dialog über die entsprechende Ribbon-Schaltfläche aufrufen (siehe Bild 1).

Wir wollen nun sehen, ob wir das Kombinationsfeld so anpassen können, dass wir auch durch direkte Eingabe der Bezeichnung der Kategorie in das Kombinationsfeld einen neuen Eintrag zur Tabelle **Kategorien** hinzufügen können.

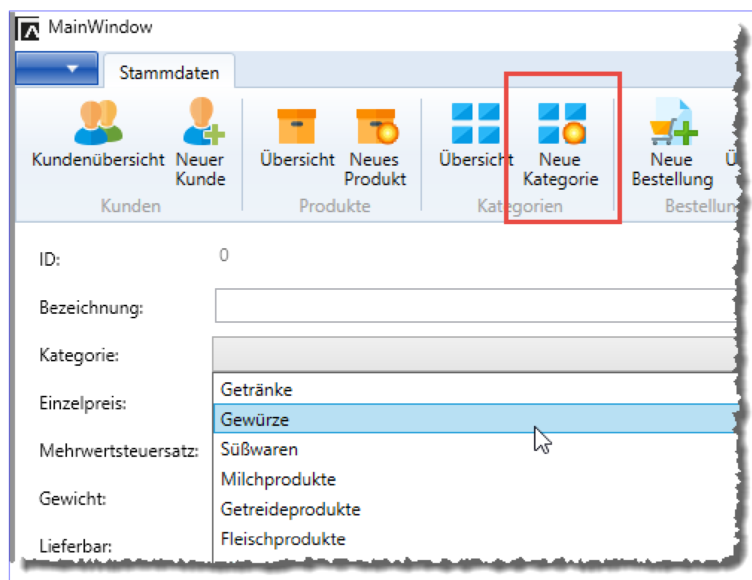

Bild 1: Neue Einträge fügt man über einen separaten Dialog hinzu.

Dazu fügen wir dem Ribbon eine Schaltfläche hinzu, mit der wir ein neues Fenster zum Anlegen eines neuen Produkts öffnen wollen. Wir könnten das auch mit dem bereits bestehenden **Page**-Objekt **Produktdetails.xaml** erledigen, aber ein eigenes Fenster ohne die Ereignisse und weiteren Code wird das Beispiel schlicht übersichtlicher. Das Fenster wollen wir zum Anzeigen der Details eines bestehenden Produkts und zum Anlegen eines neuen Produkts öffnen. Die Schaltflächen, welche das Fenster öffnen, bringen wir unten in der Seite **Produktuebersicht.xaml** unter. Die erste Schaltfläche heißt **btnNeuesProdukt_Window**, die zweite **btnProduktbearbeiten_Window**.

Also legen wir ein neues Fenster namens **Produktdetails_Kombi.xaml** an. Zur Definition dieses Fensters verwenden wir zunächst den gleichen **.xaml**-Code wie für das **Page**-Element **Produktdetails.xaml**:

```
<Window x:Class="Bestellverwaltung.Produktdetails_Kombi" ... Title="Produktdetails_Kombi" Height="300" Width="300">
    <Window.Resources>...</Window.Resources>
    <Grid x:Name="grid1" HorizontalAlignment="Stretch" Margin="8,8,0,0" VerticalAlignment="Top">
        //... Grid-Definition der Zeilen und Spalten ...
        <Label Content="ID:" Grid.Column="0" />
        <TextBox x:Name="txtID" Text="{Binding produkt.ID, Mode=TwoWay, ValidatesOnExceptions=true}" ... />
        <Label Content="Bezeichnung:" ... />
        <TextBox x:Name="txtBezeichnung" Text="{Binding produkt.Bezeichnung, Mode=TwoWay, ValidatesOnDataErrors=true}" ... />
        <Label Content="Kategorie:" ... />
        <ComboBox x:Name="cboKategorien" Grid.Column="1" HorizontalAlignment="Stretch" TabIndex="0"
                  Height="24" Grid.Row="2"
                  ItemsSource="{Binding Kategorien}"
                  SelectedItem="{Binding produkt.Kategorien, ValidatesOnDataErrors=True}"
                  DisplayMemberPath="Bezeichnung"
                  SelectedValuePath="ID"
                  VerticalAlignment="Center" Margin="1,4,29,4"></ComboBox>
        <Label Content="Einzelpreis:" ... />
        <TextBox x:Name="txtEinzelpreis" Text="{Binding produkt.Einzelpreis, Mode=TwoWay, ValidatesOnDataErrors=True}" ... />
        <Label Content="Mehrwertsteuersatz:" ... />
        <TextBox x:Name="txtMehrwertsteuersatz"
            Text="{Binding produkt.Mehrwertsteuersatz, Mode=TwoWay, ValidatesOnDataErrors=True}" ... />
        <Label Content="Gewicht:" ... />
        <TextBox x:Name="txtGewicht" Text="{Binding produkt.Gewicht, Mode=TwoWay, ValidatesOnDataErrors=True}" ... />
        <Label Content="Lieferbar:" ... />
        <CheckBox x:Name="chkLieferbar" IsChecked="{Binding produkt.Lieferbar, Mode=TwoWay, ValidatesOnDataErrors=True}" ... />
        <StackPanel Orientation="Horizontal" ... >
            <Button x:Name="btnSpeichern" Click="btnSpeichern_Click" Content="Speichern" ... />
            <Button x:Name="btnVerwerfen" Click="btnVerwerfen_Click" Content="Verwerfen" ... />
        </StackPanel>
    </Grid>
</Window>
```

Das Fenster sieht somit wie in Bild 2 aus. Um das Fenster zu öffnen, implementieren wir die beiden **Click**-Ereignisse für die auf der Seite **Produktuebersicht.xaml** neu angelegten Schaltflächen. Die Methode zum Öffnen des Formulars zum Anlegen eines neuen Produkts sieht wie folgt aus:

```
private void btnNeuesProdukt_Window_Click(object sender, RoutedEventArgs e) {
    Produktdetails_Kombi wnd = new Produktdetails_Kombi();
```

```
    wnd.ShowDialog();
}
```

Die Prozedur erstellt also einfach nur eine neue Instanz des Fensters **Produktdetails_Window** und öffnet dieses als modalen Dialog. Die Methode, die nach der Auswahl eines der Einträge der Produktübersicht und dem Anklicken der Schaltfläche **btnProduktAnzeigen** ausgelöst wird, ermittelt zunächst die **ID** des aktuell markierten Produkt-Datensatzes. Dann erstellt sie ebenfalls eine neue Instanz des Fensters **Produktdetail_Window** und übergibt die **ID** des markierten Produkts als Parameter für die Konstruktor-Methode:

```
private void btnProduktAnzeigen_Window_Click(object sender,
        RoutedEventArgs e) {
    Produkt produkt = (Produkt)dgProdukte.SelectedItem;
    int produktID = (int)produkt.ID;
    Produktdetails_Kombi wnd = new Produktdetails_Kombi(produktID);
    wnd.ShowDialog();
}
```

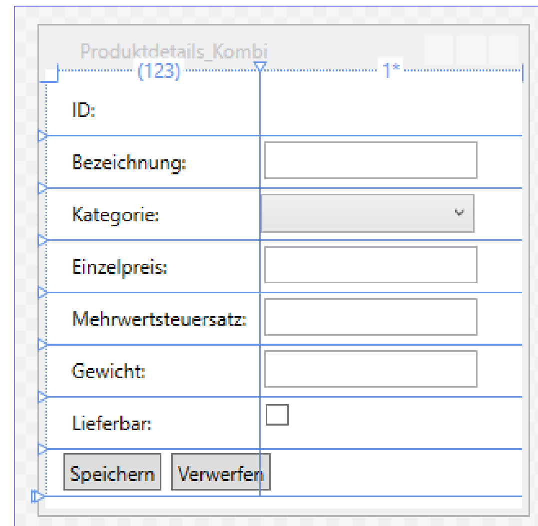

Bild 2: Entwurf des Fensters **Produktdetails_Kombi.xaml**

Damit steigen wir in den überschaubaren Code der Code behind-Klasse des Fensters **Produktdetails_Kombi** ein. Diese deklariert zunächst den Datenbankkontext für den Zugriff auf die Daten im Entity Data Model:

```
BestellverwaltungEntities dbContext = new BestellverwaltungEntities();
```

Außerdem benötigen wir ein Objekt für das anzuzeigende Produkt:

```
public Produkt produkt { get; set; }
```

Die Deklaration der Variablen zum Aufnehmen der Kategorien, die dann im Kombinationsfeld **cboKategorien** landen sollen, sieht hingegen etwas anders als im **Page**-Element **Produktdetails.xaml.cs** aus. Während wir dort noch ein einfaches **List**-Objekt verwendet haben, nutzen wir hier nun eine **ObservableCollection** sowohl für die private Variable als auch für die öffentliche Eigenschaft:

```
private ObservableCollection<Kategorie> kategorien;
public ObservableCollection<Kategorie> Kategorien {
    get { return kategorien; }
    set { kategorien = value; }
}
```

Damit die **ObservableCollection** zur Verfügung steht, benötigen Sie im Kopf des Moduls noch eine Anweisung, welche den entsprechenden Namespace verfügbar macht:

```
using System.Collections.ObjectModel;
```

Die Konstruktor-Methode erwartet als optionalen Parameter die **ID** des anzuzeigenden Produkts. Wird diese übergeben, ermittelt die Methode das **Produkt**-Element mit der entsprechenden **ID**, sonst wird ein neues **Produkt**-Element erstellt. Außerdem füllt die Methode die **ObservableCollection** mit den Kategorien:

```
public Produktdetails_Kombi(long produktID = 0) {
    InitializeComponent();
    DataContext = this;
    if (produktID != 0) {
        produkt = dbContext.Produkte.Find(produktID);
    }
    else {
        produkt = new Produkt();
        dbContext.Produkte.Add(produkt);
    }
    kategorien = new ObservableCollection<Kategorie>(dbContext.Kategorien);
}
```

Die Methoden zum Speichern, Verwerfen und beim Laden des Fensters verbergen keine außergewöhnlichen Neuigkeiten:

```
private void btnSpeichern_Click(object sender, RoutedEventArgs e) {
    Speichern();
    this.Close();
}
private void btnVerwerfen_Click(object sender, RoutedEventArgs e) {
    this.Close();
}
private void Speichern() {
    dbContext.SaveChanges();
}
private void Window_Loaded(object sender, RoutedEventArgs e) {
    txtBezeichnung.Focus();
}
```

Änderungen zum Bearbeiten der Inhalte des Kombinationsfeldes

Nun kommt der interessante Teil. Eine Änderung haben wir ja bereits gegenüber der Variante ohne das Hinzufügen neuer Kategorien per Kombinationsfeld vorgenommen, und zwar das Ersetzen des Typs **List** durch den Typ **ObservableCollection**.

m:n-Beziehung mit Listenfeld

Wir haben in der Beispieldatenbank Bestellverwaltung bereits eine m:n-Beziehung über ein Fenster realisiert, das an eine Tabelle gebunden ist und ein DataGrid enthält, das die verknüpfte Elemente anzeigt. Dabei handelt es sich um die Abbildung von Bestellungen, Bestelldetails und Produkte, wo zusätzlich zur Verknüpfung noch weitere Daten wie der Einzelpreis für die Bestellposition gespeichert werden. Im vorliegenden Artikel werden wir uns ansehen, wie wir eine m:n-Beziehung ohne weitere Daten in der Verknüpfungstabelle verwalten können. Dazu wollen wir Kunden über eine Verknüpfungstabelle einer Tabelle mit Versendungen etwa zu Werbe- oder Informationszwecken verknüpfen.

Notwendige Tabellen/ Erweiterungen

Die Datenbank **Bestellverwaltung.db** auf Basis von SQLite, die wir in den bisherigen Beispielen verwendet haben, müssen wir für diesen Artikel um zwei Tabellen erweitern.

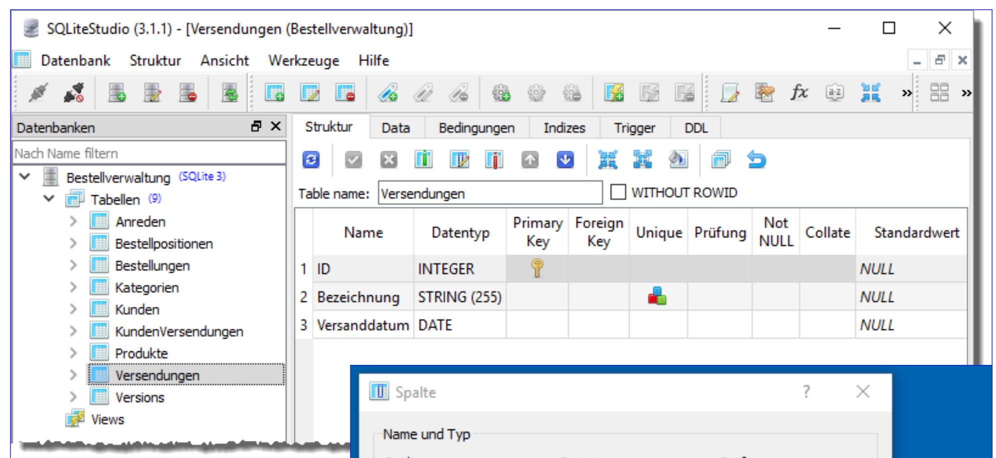

Bild 1: Anlegen der neuen Tabelle **Versendungen**

Das erledigen wir schnell über das Verwaltungsprogramm SQLite Studio, mit dem wir die Datei **Bestellverwaltung.db** öffnen und als Erstes die Tabelle **Versendungen** anlegen – und zwar mit den Feldern wie in Bild 1. Speichern Sie die Tabelle nach dem Anlegen der Felder und des Tabellennamens mit einem Klick auf die Schaltfläche **Commit structure changes**.

Danach folgt die zweite Tabelle **KundenVersendungen** als Verknüpfungstabelle. Diese Tabelle enthält neben dem Primärschlüsselfeld **ID** noch die beiden Fremdschlüsselfelder **KundeID** und **VersendungID**, welche die beiden Tabellen **Kunden** und **Versendungen** referenzieren.

Wenn Sie die Fremdschlüsselfelder anlegen, aktivieren Sie im Dialog **Spalte** die Option **Fremdschlüssel** und klicken dann auf die Schaltfläche **Konfigurieren**. Dies zeigt den Dialog

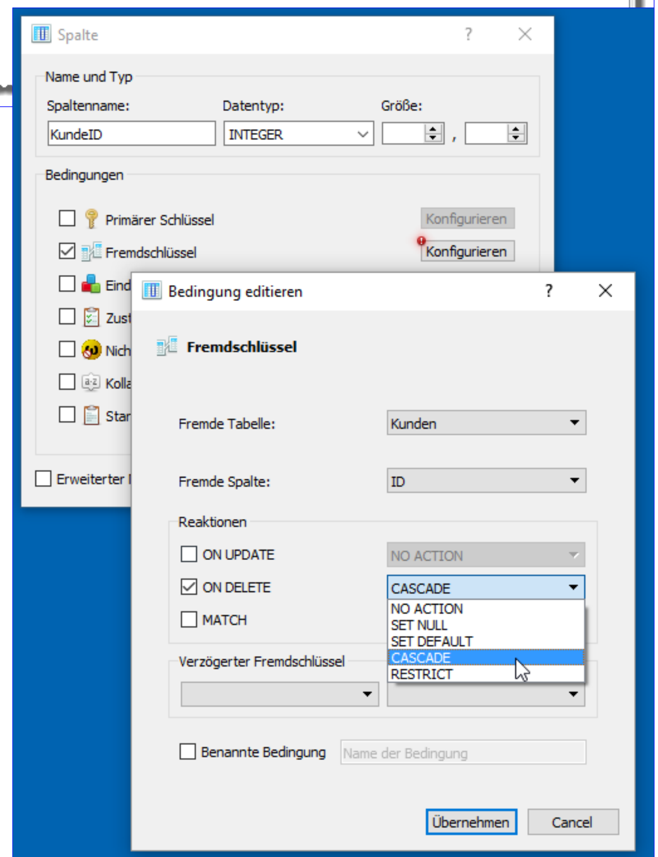

Bild 2: Anlegen der neuen Tabelle **KundenVersendungen**

Bedingung editieren an, wo Sie als **Fremde Tabelle** die Tabelle **Kunden** und als **Fremde Spalte** die Spalte **ID** auswählen (siehe Bild 2). Hier aktivieren Sie außerdem die Option **ON DELETE** und wählen dafür die Aktion **CASCADE** aus.

Dies bewirkt, dass wenn Sie einen Datensatz der Tabelle **Kunden löschen**, für den in der Tabelle **KundenVersendungen** verknüpfte Datensätze vorliegen, diese auch gelöscht werden. Das Gleiche führen wir auch für das Fremdschlüsselfeld zum Verknüpfen mit der Tabelle **Versendungen** durch. Speichern Sie die Tabelle mit der Schaltfläche **Commit structure changes**.

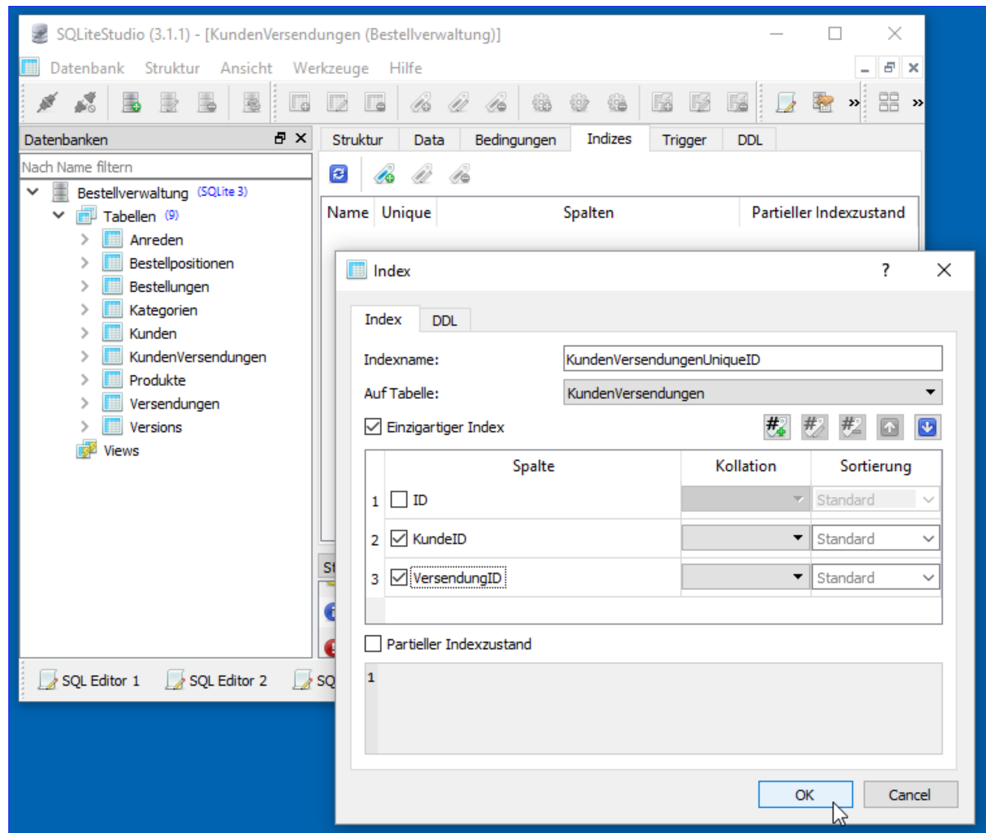

Bild 3: Anlegen eines zusammengesetzten, eindeutigen Schlüssels über die beiden Felder **KundeID** und **VersendungID**

Danach legen wir noch einen zusammengesetzten, eindeutigen Schlüssel für die beiden Fremdschlüsselfelder **KundeID** und **VersendungID** an, damit jede Versendung jedem Kunden nur einmal zugeordnet werden kann. Dazu klicken Sie oben über dem Tabellenentwurf auf den Reiter **Indizes**.

Klicken Sie dann auf die Schaltfläche **Create index (ins)**. Dann tragen Sie im nun erscheinenden Dialog **Index** den Wert **KundenVersendungenUniqueIndex** als Name ein. Wählen Sie die beiden Felder **KundeID** und **VersendungID** aus und aktivieren Sie die Option **Einzigartiger Index** (okay, die Übersetzung ist zumindest gut gemeint von den Entwicklern des Tools). Dies soll dann wie in Bild 3 aussehen, bevor Sie auf **OK** klicken und anschließend das Anlegen des Indizes bestätigen.

Änderungen im Entity Data Model

Nachdem Sie die beiden neuen Tabellen angelegt haben, benötigen wir in der Anwendung die entsprechenden Entitäten für die beiden Tabellen. Dazu öffnen Sie die Anwendung **Bestellverwaltung** und klicken im Projekt-Explorer doppelt auf das Objekt **Bestellverwaltung.edmx**.

Klicken Sie mit der rechten Maustaste in das Diagramm und wählen Sie aus dem Kontextmenü den Eintrag **Modell aus der Datenbank aktualisieren...** aus. Gegebenenfalls müssen Sie nun die Verbindung für das Entity Data Model festlegen. Danach

erscheint dann der Dialog, der auch beim Erstellen eines Entity Data Models angezeigt wird. Hier wählen Sie die beiden einzigen angezeigten Tabellen **KundenVersendungen** und **Versendungen** aus und klicken auf **Fertig stellen**.

Danach sollten die neuen Tabellen wie in Bild 4 im Entity Data Model angezeigt werden, sodass wir nun darauf zugreifen können. Genau wie bei den übrigen Entitäten haben wir auch bei diesen beiden Entitäten die Benennung in den Singular geändert, also von **KundenVersendungen** auf **KundeVersendung** und von **Versendungen** auf **Versendung**.

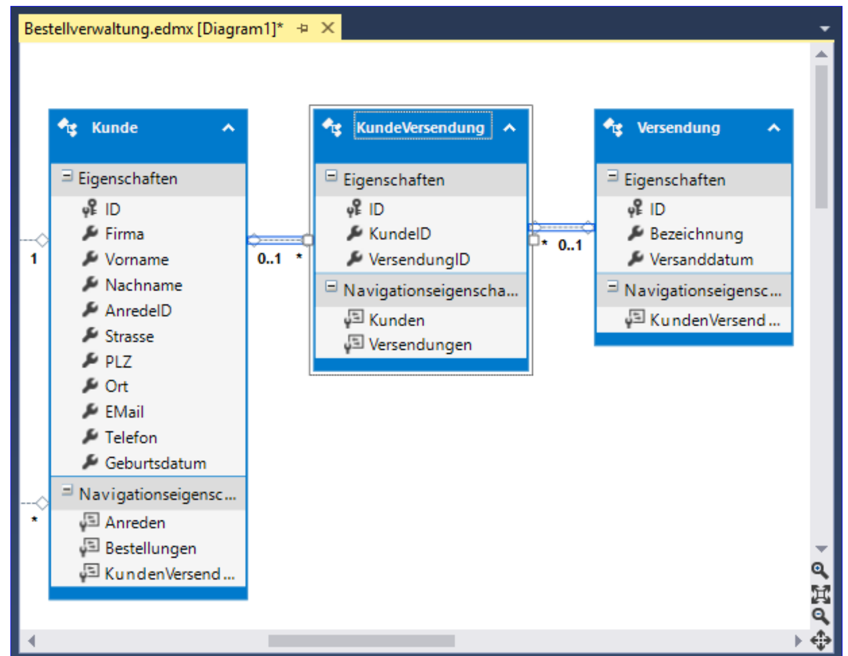

Bild 4: Die beiden neuen Entitäten im Entity Data Model, hier bereits in der Singular-Benennung

Wenn Sie nun noch den neuen Zustand etwa mit der Tastenkombination **Strg + S** speichern, werden auch die dahinter liegenden Objekte umbenannt.

Fenster zum Zuordnen von Kunden und Versendungen

Das Fenster, in dem wir einer Versendung die entsprechenden Kunden zuordnen wollen, soll im oberen Bereich ein Kombinationsfeld enthalten, welches die Versendungen abbildet. Im unteren Bereich wollen wir zwei Listenfelder darstellen. Das linke enthält alle Kunden, die der Versendung bereits zugeordnet wurden und das rechte als übrigen Kunden. Dazwischen platzieren wir vier Schaltflächen für die folgenden Funktionen:

- **<**: Den aktuell markierten Kunden aus dem rechten Listenfeld zur Liste der Empfänger hinzufügen.
- **>**: Den aktuell markierten Kunden aus dem linken Listenfeld aus der Liste der Empfänger entfernen.
- **<<**: Alle Kunden aus dem rechten Listenfeld zur Liste der Empfänger hinzufügen.

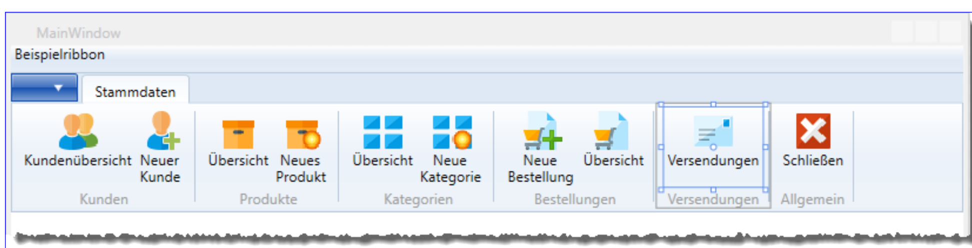

Bild 5: Neue Ribbon-Schaltfläche zum Anzeigen des Fensters mit den Zuweisungen der Versendungen

- **>>**: Alle Kunden aus dem linken Listenfeld aus der Liste der Empfänger entfernen.

Fenster anlegen

Das neue Fenster wollen wie diesmal einmal nicht in die bestehende Struktur einbinden, sondern über den Ribbon-Eintrag **Versendungen** als eigenes Fenster öffnen. Das Fenster legen Sie im Projekt unter dem Namen **Versendungen.xaml** an. Dem Fenster **MainWindow.xaml** fügen wir eine neue Ribbon-Schaltfläche wie in Bild 5 hinzu. Der Code dafür sieht wie folgt aus:

```
<RibbonGroup Header="Versendungen">
    <RibbonButton Name="btnVersendungen" Label="Versendungen" Click="btnVersendungen_Click"
        LargeImageSource="images\mail2.png"></RibbonButton>
</RibbonGroup>
```

Für die Schaltfläche hinterlegen wir die folgende Ereignismethode:

```
private void btnVersendungen_Click(object sender,
        RoutedEventArgs e) {
    Versendungen wnd = new Versendungen();
    wnd.ShowDialog();
}
```

Damit schauen wir uns nun den Entwurf des Fensters **Versendungen.xaml** an (siehe Bild 6). Das Fenster enthält drei Spalten und drei Zeilen. Die erste Zeile enthält lediglich das Kombinationsfeld **cboVersendungen** zur Auswahl der gewünschten Versendung. Der Inhalt befindet sich in einem horizontal ausgerichteten **StackPanel**-Element, dass sich über alle drei Spalten erstreckt (**Grid.ColumnSpan="3"**).

Die zweite Zeile enthält in der ersten Spalte das erste Listenfeld, in der zweiten Spalte ein vertikales **StackPanel**-Element mit den vier Schaltflächen und die dritte Spalte das zweite Listenfeld. Die dritte Spalte schließlich enthält eine **OK**-Schaltfläche zum Schließen des Fensters. Der **.xaml**-Code des Fensters sieht in gekürzter Form wie folgt aus:

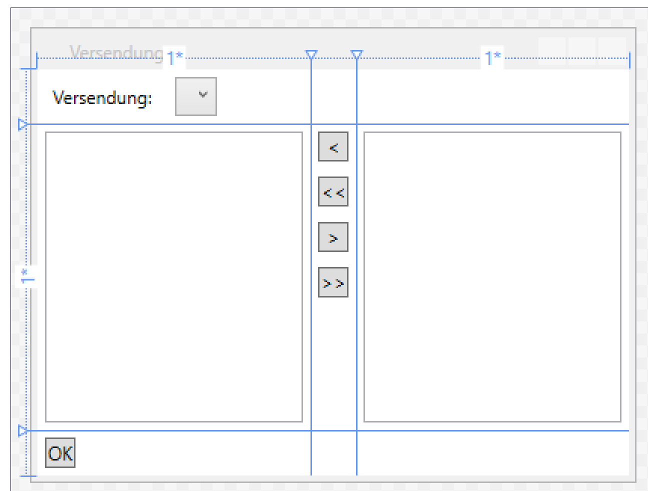

Bild 6: Entwurf des Fensters **Versendungen.xaml**

```
<Window x:Class="Bestellverwaltung.Versendungen" ... Title="Versendungen" Height="300" Width="400">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
```

```
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <StackPanel Orientation="Horizontal" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3">
            <Label Margin="5">Versendung:</Label>
            <ComboBox x:Name="cboVersendungen" Margin="5"></ComboBox>
        </StackPanel>
        <ListBox Grid.Column="0" Grid.Row="1" Margin="5"></ListBox>
        <StackPanel Orientation="Vertical" Grid.Column="1" Grid.Row="1">
            <Button x:Name="btnEinenHinzufuegen" Margin="5" Content="&lt;"></Button>
            <Button x:Name="btnAlleHinzufuegen" Margin="5" Content="&lt;&lt;"></Button>
            <Button x:Name="btnEinenEntfernen" Margin="5" Content="&gt;"></Button>
            <Button x:Name="btnAlleEntfernen" Margin="5" Content="&gt;&gt;"></Button>
        </StackPanel>
        <ListBox Grid.Column="2" Grid.Row="1" Margin="5"></ListBox>
        <StackPanel Orientation="Horizontal" Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="3">
            <Button x:Name="btnOK" Margin="5" Click="btnOK_Click">OK</Button>
        </StackPanel>
    </Grid>
</Window>
```

Interessant ist hier, wie wir die Breite und die Höhe der Spalten und Zeilen eingestellt haben. Damit die mittlere Spalte genau die für die vier Schaltflächen in dem Stackpanel benötigte Spaltenbreite annimmt, haben wir die Eigenschaft **Width** auf den Wert **Auto** eingestellt. Die linke und die rechte Spalte sollen sich den übrigen Platz gleichmäßig aufteilen, daher erhält die Eigenschaft **Width** hier den Wert *.

Die obere Zeile und die untere Zeile sollen wie die mittlere Spalte nur den benötigten Platz einnehmen, also stellen wir die Eigenschaft **Height** für beide auf den Wert **Auto** ein. Die mittlere Zeile soll den Rest der Höhe beanspruchen, war wir durch den Wert * für die Eigenschaft **Height** erreichen.

Die Definition enthält bislang nur die reinen Steuerelemente ohne jegliche Bindung.

Kombinationsfeld mit den Versendungen füllen

Um das Kombinationsfeld mit den Datensätzen der Tabelle **Versendungen** in alphabetischer Reihenfolge zu füllen, müssen zunächst im Code behind-Modul eine entsprechende Auflistung erzeugen und aus dem Entity Data Model füllen.

Dazu erstellen wir zunächst einen Datenbank-Kontext im allgemeinen Teil der Klasse, der somit direkt beim Erstellen angelegt wird:

```
BestellverwaltungEntities dbContext = new BestellverwaltungEntities();
```

Für den Inhalt des Kombinationsfeldes legen wir eine ObservableCollection an, die wir als private Variable namens **versendungen** sowie als öffentliche Variable namens **AlleVersendungen** definieren (**AlleVersendungen**, weil wir bereits das Fenster **Versendungen** genannt haben):

```
private ObservableCollection<Versendung> versendungen;
public ObservableCollection<Versendung> AlleVersendungen{
    get { return versendungen; }
    set { versendungen = value; }
}
```

In der Konstruktor-Methode des Fensters füllen wir dann zunächst die ObservableCollection, die als Datenquelle für das Kombinationsfeld dienen soll und weisen dem Fenster die Code behind-Klasse als **DataContext** zu:

```
public Versendungen() {
    InitializeComponent();
    versendungen = new ObservableCollection<Versendung>(dbContext.Versendungen);
    DataContext = this;
}
```

Nun passen wir in **Versendungen.xaml** die Definition des **ComboBox**-Elements an, damit es die Daten der **ObservableCollection** anzeigt:

```
<ComboBox x:Name="cboVersendungen" ... IsEditable="True"
    Text="{Binding NewEntry, UpdateSourceTrigger=LostFocus}"
    ItemsSource="{Binding AlleVersendungen}"
    DisplayMemberPath="Bezeichnung"
    SelectedValuePath="ID"
    VerticalAlignment="Center"></ComboBox>
```

Hier haben wir gleich die Vorbereitungen getroffen, dass der Benutzer neue Versendungen direkt in das Kombinationsfeld eingeben kann. Wie das im Detail funktioniert, erläutern wir im Artikel **Neuer Eintrag in ComboBox**. Hier nur die Kurzfassung: **IsEditable="True"** sorgt dafür, dass der Benutzer Text in das Kombinationsfeld eintippen kann. Das Binding für das Attribut **Text** sorgt dafür, dass eine Eigenschaftsmethode namens **NewEntry** neue Werte entgegennimmt und **UpdateSourceTrigger="LostFocus"** sorgt dafür, dass die Methode **NewEntry** beim Fokusverlust ausgelöst wird. Die Methode **NewEntry** sieht so aus:

```
public string NewEntry {
    set {
        if (cboVersendungen.SelectedIndex == -1) {
```

```
                if (!string.IsNullOrEmpty(value)) {
                    MessageBoxResult result = MessageBox.Show("Eintrag '" + value + "' zu den Versendungen hinzufügen?",
                        "Neuer Eintrag", MessageBoxButton.YesNo);
                    if (result == MessageBoxResult.Yes) {
                        Versendung newVersendung = new Versendung();
                        newVersendung.Bezeichnung = value;
                        versendungen.Add(newVersendung);
                        dbContext.Versendungen.Add(newVersendung);
                        dbContext.SaveChanges();
                    }
                }
            }
        }
}
```

Sie prüft, ob der Inhalt des eingegebenen Textes, der wie üblich über den Parameter **value** geliefert wird, nicht leer ist und fragt in diesem Fall den Benutzer, ob er den Wert als neuen Eintrag hinzufügen möchte (siehe Bild 7). Falls ja, legt die Methode ein neues Objekt des Typs **Versendung** an und weist **value** als **Bezeichnung** hinzu. Dann fügt sie das neue Objekt zur **ObservableCollection** und zur entsprechenden Liste des Entity Data Models hinzu und speichert die Änderung in der Datenbank.

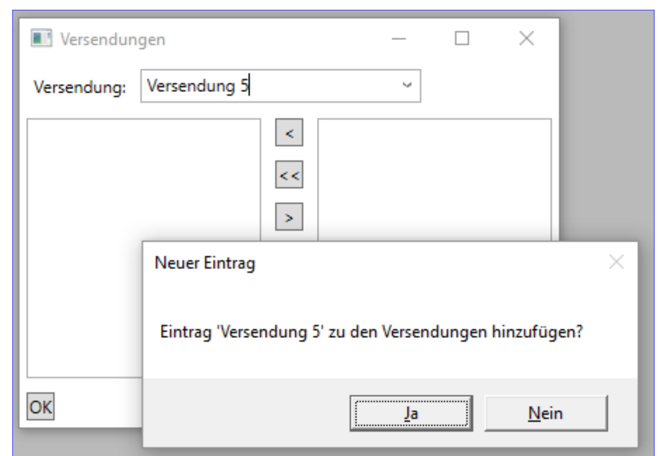

Bild 7: Anlegen eines neuen Eintrags im Kombinationsfeld

Listenfelder füllen

Nun wollen wir die Listenfelder mit den Kunden füllen, die einer Versendung zugewiesen wurden oder auch nicht. Dabei kommt die m:n-Beziehung zwischen den Versendungen und den Kunden ins Spiel. Das linke Listenfeld soll alle Kunden anzeigen, die über die m:n-Verknüpfungstabelle **KundenVersendungen** der aktuell im Kombinationsfeld **cboVersendungen** zugeordnet sind, das rechte Listenfeld alle Kunden, die der Versendung nicht zugeordnet sind. Diese Anzeige soll jeweils beim Auswählen eines neuen Eintrags im Kombinationsfeld aktualisiert werden. Außerdem sorgen wir gleich noch dafür, dass beim Öffnen des Fensters der erste Eintrag des Kombinationsfeldes vorausgewählt wird – und damit auch die entsprechenden Einträge in den Listenfeldern. Die Konstruktor-Methode erweitern wir also wie folgt:

```
public Versendungen() {
    InitializeComponent();
    versendungen = new ObservableCollection<Versendung>(dbContext.Versendungen);
    Versendung versendung = versendungen.First();
    cboVersendungen.SelectedIndex = 0;
```

```
        int intVersendungID = (int)versendung.ID;
        ausgewaehlteKunden = new ObservableCollection<Kunde>(dbContext.Kunden.Where(c => c.KundenVersendungen.Any(d =>
            d.VersendungID == intVersendungID)));
        ObservableCollection<Kunde>alleKunden = new ObservableCollection<Kunde>(dbContext.Kunden);
        abgewaehlteKunden = new ObservableCollection<Kunde>(alleKunden.Except(ausgewaehlteKunden));
        DataContext = this;
    }
```

Was geschieht hier? Als Erstes ermitteln wir die Auflistung **versendungen** für die Einträge im Kombinationsfeld, der wir alle Einträge der Liste **Versendungen** des Datenbank-Kontextes zuweisen. Dann ermitteln wir das erste Element der Auflistung und speichern es in der Variablen **versendung** des Typs **Versendung**.

Den Index des Kombinationsfeldes stellen wir auf **0** ein, also auf den ersten Eintrag. Die **ID** dieser Versendung ermitteln wir über die entsprechende Eigenschaft und speichern sie in der Variablen **intVersendungID**.

Damit erstellen wir nun die ObservableCollection **ausgewaehlteKunden**, der wir alle Elemente der Kunden-Liste zuweisen, für die es ein Objekt in der Auflistung **KundenVersendungen** gibt, dessen Wert im Feld **VersendungID** den von uns ermittelten Wert in der Variablen **intVersendungID** enthält. Das sind dann alle Kunden, die bereits einer Versendung zugeordnet wurden, und zwar durch das Hinzufügen eines entsprechenden Datensatzes zur Tabelle **KundenVersendungen**. Diese ObservableCollection landen in der Variablen **ausgewaehlteKunden** und wird über die öffentliche Eigenschaft **AusgewaehlteKunden** der ListBox namens **lstAusgewaehlteKunden** zugewiesen.

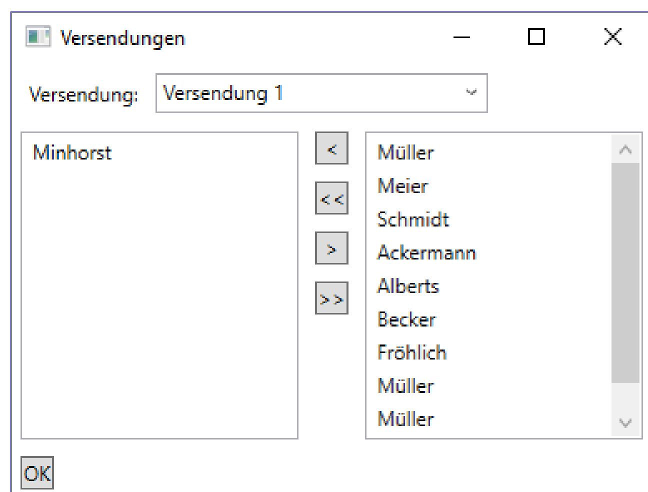

Bild 8: Anzeige einer Versendung mit einem Empfänger

Nun folgt der schwierigere Teil – zumindest war das der Programmierung einer Lösung wie dieser hier unter Access der Fall. Die rechte ListBox soll nun nämlich alle Einträge der Tabelle **Kunden** auflisten, die nicht in der linken Listbox enthalten sind – also alle Einträge der Tabelle **Kunden**, die nicht über einen Eintrag in der Tabelle **KundenVersendungen** mit der Tabelle **Versendungen** verknüpft sind.

Unter Access benötigten wir eine Abfrage mit einer **IN**-Klausel dazu, welche alle Einträge der Tabelle Kunden lieferte, die nicht in der für das erste Listenfeld enthaltenen Abfrage enthalten waren. Unter C# und LINQ for Entities gelingt das ein wenig eleganter. Wir definieren dazu zunächst in der Variablen **alleKunden** eine neue ObservableCollection, die schlicht alle Elemente der Tabelle **Kunden** aufnimmt:

```
ObservableCollection<Kunde>alleKunden = new ObservableCollection<Kunde>(dbContext.Kunden);
```

Dann verwenden wir einfach die **Except**-Methode dieses Objekts und geben als Parameter die ObservableCollection **ausgewaehlteKunden** an. Dies liefert alle Elemente aus **alleKunden** außer denen, die sich in **ausgewaehlteKunden** befinden:

LINQ to Entities-Beispiele in der Methodensyntax

LINQ to Entities ist die Abfragesprache für den Zugriff auf Daten in den Entitäten eines Entity Data Models. Im vorliegenden Artikel sehen wir uns einige Beispiele für den Zugriff per LINQ auf die Daten unserer Beispielanwendung »Bestellverwaltung« an. Dabei verwenden wir in diesem Artikel die Methoden-Syntax von LINQ.

Voraussetzungen

Für die hier vorgestellten Beispiele verwenden wir das Entity Data Model der Anwendung **Bestellverwaltung**. Die Ergebnisse sollen jeweils in einem DataGrid ausgegeben werden. Dazu erstellen wir ein neues Fenster namens **LINQBeispiele**, dessen Entwurf wie in Bild 1 aussieht. Die Definition ist einfach gehalten. Die wichtigsten Elemente ist das Kombinationsfeld **cboBeispiele**, mit dem Sie die Beispiele auswählen können, die wir anschließend per C#-Code zum Code behind-Modul hinzufügen, sowie das **DataGrid**-Steuerelement **db**, welches das Ergebnis der jeweiligen Abfrage anzeigen soll. Das **ComboBox**-Steuerelement ist an die Eigenschaft **Beispiele** gebunden und löst das Ereignis **SelectionChanged** aus:

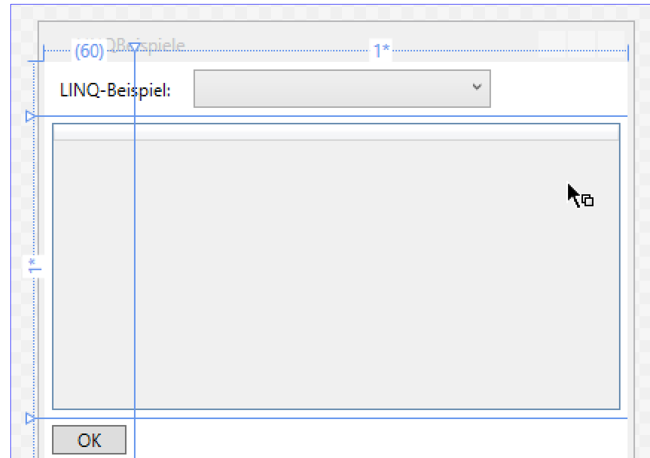

Bild 1: Fenster zur Ausgabe der Beipiele

```
<Window x:Class="Bestellverwaltung.LINQBeispiele"
        Title="LINQBeispiele" Height="300" Width="400">
    <Grid>
        ... Grid-Definition ...
        <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Orientation="Horizontal">
            <Label Margin="5">LINQ-Beispiel:</Label>
            <ComboBox x:Name="cboBeispiele" Margin="5" Width="200" SelectionChanged="cboBeispiele_SelectionChanged"
                ItemsSource="{Binding Beispiele}"></ComboBox>
        </StackPanel>
        <DataGrid x:Name="dg" Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" Margin="5"></DataGrid>
        <Button x:Name="btnOK" Grid.Row="2" Margin="5" Width="50" Click="btnOK_Click">OK</Button>
    </Grid>
</Window>
```

In der Code behind-Klasse erstellen wir ein Datenbank-Kontext-Objekt mit Zugriff auf das Entity Data Model sowie eine **ObservableCollection**, welche die Liste der Beispiele für das Kombinationsfeld enthält:

```
BestellverwaltungEntities dbContext = new BestellverwaltungEntities();
private ObservableCollection<string> beispiele;
```

```
public ObservableCollection<string> Beispiele {
    get { return beispiele; }
    set { beispiele = value; }
}
```

In der Konstruktor-Methode erstellen wir die **ObservableCollection** und füllen diesem mit Einträgen wie hier am Beispiel **Alle Kunden**:

```
public LINQBeispiele() {
    InitializeComponent();
    beispiele = new ObservableCollection<string>();
    beispiele.Add("Alle Kunden");
    DataContext = this;
}
```

Für das konkrete Beispiel, alle Kunden anzuzeigen, definieren wir auch wieder eine **ObservableCollection**:

```
private ObservableCollection<Kunde> kunden;
public ObservableCollection<Kunde> Kunden {
    get { return kunden; }
    set { kunden = value; }
}
```

Diese wird schließlich gefüllt und angezeigt, wenn der Benutzer den Eintrag **Alle Kunden** aus dem Kombinationsfeld auswählt. Dies löst das Ereignis **SelectionChanged** aus, für das wir die folgende Ereignismethode hinterlegt haben:

```
private void cboBeispiele_SelectionChanged(object sender, SelectionChangedEventArgs e) {
    ComboBox cbo = (ComboBox)sender;
    string beispiel = (string)cbo.SelectedItem;
    switch (beispiel) {
        case "Alle Kunden":
            kunden = new ObservableCollection<Kunde>(dbContext.Kunden);
            dg.ItemsSource = Kunden;
            break;
        default:
            break;
    }
}
```

Für den Eintrag **Alle Kunden** füllt die Methode nun die **ObservableCollection** namens **kunden** mit den über den LINQ-Ausdruck **dbContext.Kunden** ermittelten Kunden. Die Eigenschaft **Kunden** wird dann noch dem DataGrid **dg** zugewiesen, damit

dieses die Kundenliste wie in Bild 2 anzeigt. Das DataGrid ist ja flexibel, das heißt, Sie können ihm beliebige Auflistungen zuweisen. Wir fügen daher, wenn wir in den folgenden Beispielen andere **ObservableCollections** als **Kunden** nutzen, einfach entsprechende neue ObservableCollections hinzu und weisen die entsprechende Collection der Eigenschaft **ItemsSource** des **DataGrid**-Elements zu.

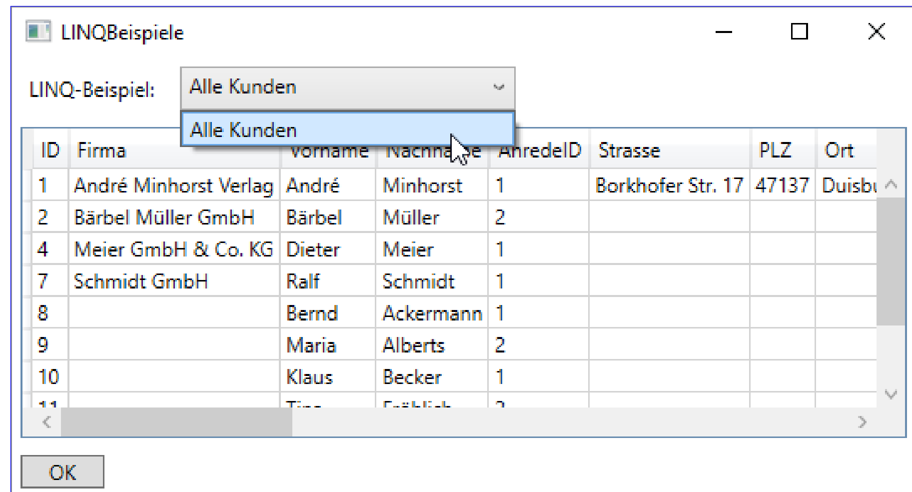

Bild 2: Ausgabe aller Kunden des Entity Data Models

Alle Elemente einer Klasse

Das erste Beispiel haben Sie ja schon kennengelernt. Um alle Elemente einer Klasse beziehungsweise Auflistung zu liefern, brauchen Sie diese einfach nur gemeinsam mit dem **dbContext**-Objekt als Inhalt der neuen ObservableCollection anzugeben:

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

Dies liefert alle Elemente der **Kunden**-Auflistung des Entity Data Models. Alle Elemente von anderen Klassen können Sie ebenso leicht holen, dazu müssen Sie nur die entsprechende **ObservableCollection** definieren und dem DataGrid zuweisen. Wenn Sie nun also alle Produkte ausgeben wollen, legen Sie erst eine neue ObservableCollection samt privater und öffentlicher Variable an:

```
private ObservableCollection<Produkt> produkte;
public ObservableCollection<Produkt> Produkte {
    get { return produkte; }
    set { produkte = value; }
}
```

Dann fügen Sie einen neuen Eintrag zur **ComboBox** hinzu:

```
public LINQBeispiele() {
    ...
    beispiele.Add("Alle Produkte");
    ...
}
```

Schließlich noch der neue **case**-Zweig in der **switch**-Bedingung:

```
case "Alle Produkte":
    produkte = new ObservableCollection<Produkt>(dbContext.Produkte);
```

EDM: SQLite aktuell halten

SQLite erscheint regelmäßig in neuen Versionen. Das betrifft sowohl die NuGet-Pakete als auch die Komponenten, die für die Arbeit in Visual Studio wichtig sind. Wenn Sie nicht beide auf dem gleichen Stand halten, sind Probleme vorprogrammiert. Dieser Artikel beschreibt, welche Schritte zu beachten sind.

Fehler bei nicht passenden Versionen

Einer der Fehler, die auftreten können, wenn das NuGet-Paket in einem Projekt und die Runtime Library, die Sie von der Seite **https://system.data.sqlite.org/index.html/doc/trunk/www/downloads.wiki** beziehen können, nicht die gleiche Version aufweisen, wird in Bild 1 angezeigt.

Der Fehler tritt beispielsweise auf, wenn Sie in der Übersicht des Entity Data Models einer Anwendung den Kontextmenü-Eintrag **Modell aus der Datenbank aktualisieren...** aufrufen oder auch ein neues Entity Data Model aus einer Datenbank generieren wollen.

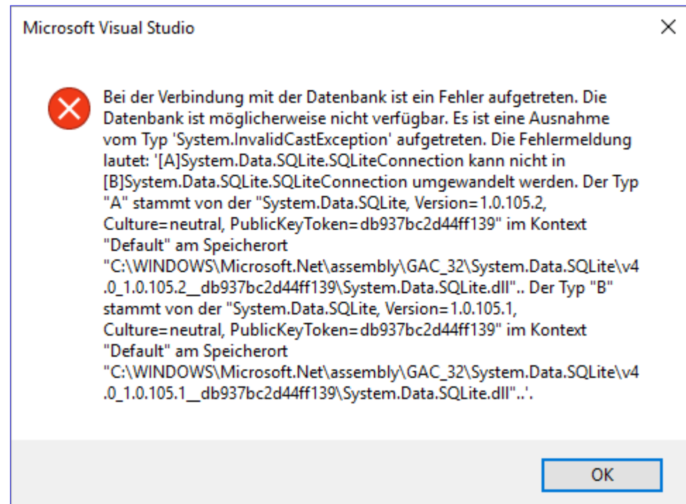

Bild 1: Fehlermeldung beim Aktualisieren des Entity Data Models

Das Problem tritt auf einem Entwicklungsrechner beispielsweise dann auf, wenn Sie für eine früher angelegte Anwendung eine bestimmte Version des NuGet-Pakets und der Runtime Library verwendet haben, die natürlich zusammenpassen mussten, und dann später eine neue Anwendung unter Einsatz des Entity Frameworks mit dem SQLite-Treiber erstellen – wobei Sie für die neue Anwendung dann logischerweise die aktuelle Version des Pakets **System.Data.SQLite** verwenden. Die neue Anwendung mag dann auch laufen, aber erst, wenn Sie auch die passende Version der Runtime Library heruntergeladen und installiert haben.

Andersherum haben Sie nun eine aktuelle Version der Runtime Library auf dem Rechner. Öffnen Sie nun aber die eingangs erwähnte Anwendung, die Sie noch unter den alten Versionen der **Runtime Library** und des NuGet-Pakets **System.Data.SQLite** erstellt haben, lösen Sie ebenfalls einen Versionskonflikt aus. Diesmal ist der Grund allerdings die veraltete Version des NuGet-Pakets, welches Sie aber zum Glück leicht aktualisieren können. Dazu rufen Sie einfach den Menü-Befehl **Projekt|NuGet-Pakete verwalten...** auf und klicken für das Paket **System.Data.SQLite** auf die Schaltfläche **Aktualisieren** (siehe Bild 2). Nach einem Neustart von Visual Studio sollten alle Funktionen wieder ohne Versionskonflikte abrufbar sein.

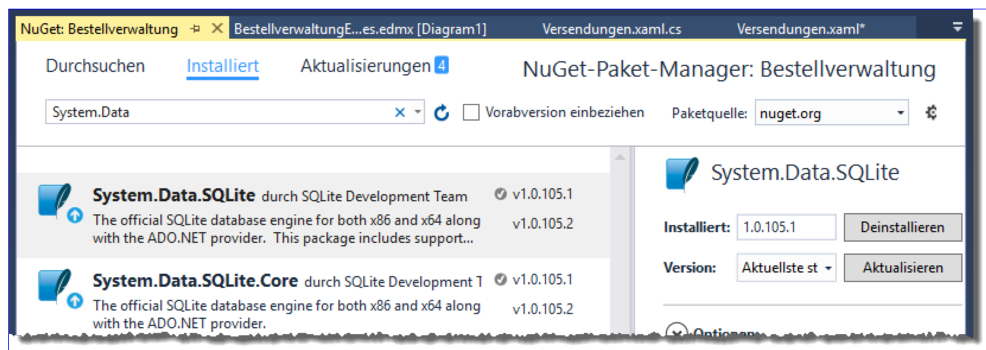

Bild 2: Aktualisieren des NuGet-Pakets

EDM: Backend ändern

Unsere aktuellen Beispieldatenbanken haben ein WPF/C#-Frontend, ein SQLite-Backend und das Entity Data Model als Zwischenschicht. Was geschieht nun, wenn wir einmal die Datenbank ändern wollen – etwa, weil wir neue Tabellen oder neue Felder in bestehenden Tabellen benötigen? Wie ist die genaue Vorgehensweise, die auch berücksichtigt, dass der Benutzer eine neue Version installieren möchte, ohne die bereits im vorhandenen Backend gespeicherten Daten zu verlieren? Wie dies gelingt, zeigt der vorliegende Artikel.

Bei der Entwicklung würden wir so vorgehen, dass wir das Datenmodell der SQLite-Datenbank ändern, indem wir die gewünschten Tabellen oder Felder hinzufügen. Dann wechseln wir zu Visual Studio und aktualisieren das Entity Data Model, gegebenenfalls müssen wir noch die Benutzeroberfläche beziehungsweise die Code behind-Klassen anpassen.

Aber was, wenn die Anwendung bereits beim Kunden im Produktiveinsatz ist? Dann würden wir die Entwicklung natürlich genau so betreiben, wie wir es oben beschrieben haben. Allerdings können wir natürlich nicht so einfach die neue Version des Datenbankbackends beim Kunden installieren, denn dieser hat ja bereits ein Backend mit gegebenenfalls neuen Daten, die wir dann überschreiben würden. Also was tun? Die neue Anwendung exklusive Backend können wir ja bei Kunden über die alte Version installieren, aber für das Backend müssen wir uns etwas einfallen lassen. Denn das neue Frontend wird auch spätestens beim Versuch, auf eines der neuen Felder per Entity Framework zuzugreifen, Probleme verursachen.

Allerdings gibt es ja die Möglichkeit, Datenmodelle per Code zu ändern. Wir können zum Beispiel neue Tabellen anlegen oder bestehende Tabellen entfernen, Felder zu Tabellen hinzufügen oder entfernen oder auch Einschränkungen ändern. Allerdings will dies gut geplant sein, denn vielleicht wollen wir das Datenmodell des Backends nicht nur einmal ändern, sondern im Laufe der Zeit je nach den geänderten Anforderungen auch öfter.

Versionsstand prüfen

In diesem Fall müssen wir in irgendeiner Form einen Versionsstand des Backends speichern, damit wir wissen, welche Aktualisierungen in Abhängigkeit vom Versionsstand durchzuführen sind. Da wir mit einem Datenbankbackend arbeiten, bietet sich dazu natürlich eine Tabelle an, die wir etwa **Versions** nennen. Die Tabelle braucht lediglich ein Feld namens **Version**, in das wir dann den Versionsstand der Tabelle eintragen.

Wir würden dann eine neue Version des Frontends installieren, diese starten und nach dem Start soll das Frontend dann prüfen, ob das Backend die dem Frontend entsprechende Version aufweist. Dazu soll das Frontend dann die Tabelle **Versions** öffnen und den Wert des Feldes **Version** prüfen. Wenn die Version kleiner als die geforderte Version ist, soll das Frontend die benötigten Änderungen am Backend durchführen und schließlich den Versionsstand in der Tabelle **Versions** aktualisieren.

Natürlich müssen wir auch einen umgekehrten Mechanismus vorsehen, der reagiert, wenn der Versionsstand des Backends aktueller ist als der des Frontends. Das kann etwa der Fall sein, wenn der Kunde das Frontend an mehr als einem Arbeitsplatz einsetzt und dieses nur an einem Arbeitsplatz auf den aktuellen Versionsstand bringt. Sollte dann ein Benutzer auf einem ande-

ren Arbeitsplatz ein veraltetes Frontend starten, sollte dieses mit einer entsprechenden Meldung darauf hinweisen, dass das Frontend aktualisiert werden soll.

Dazu gibt es natürlich noch weiterführende Techniken, bei denen das Frontend beim Start automatisch über das Internet prüft, ob es eine aktuellere Version des Frontends gibt, doch das soll nicht das Thema dieses Artikels sein.

Versionstabelle hinzufügen

Wir wollen die Schritte dieses Artikels anhand unserer Beispieldatenbank **Bestellverwaltung.db** auf Basis von SQLite erläutern. Diese enthält im aktuellen Zustand noch gar keine Tabelle namens **Versions**, sodass der erste Schritt sein wird, diese Tabelle dem Entwicklungsbackend hinzuzufügen und das Frontend um eine Funktion zu ergänzen, die beim Starten des Frontends prüft, ob die Tabelle **Versions** im Backend vorhanden ist oder nicht. Falls nicht, soll diese einfach ergänzt werden. Der Einfachheit halber finden Sie die Lösung in einer stark reduzierten Version der in den übrigen Artikeln verwendeten Anwendung, die nur das Fenster **MainWindow** enthält.

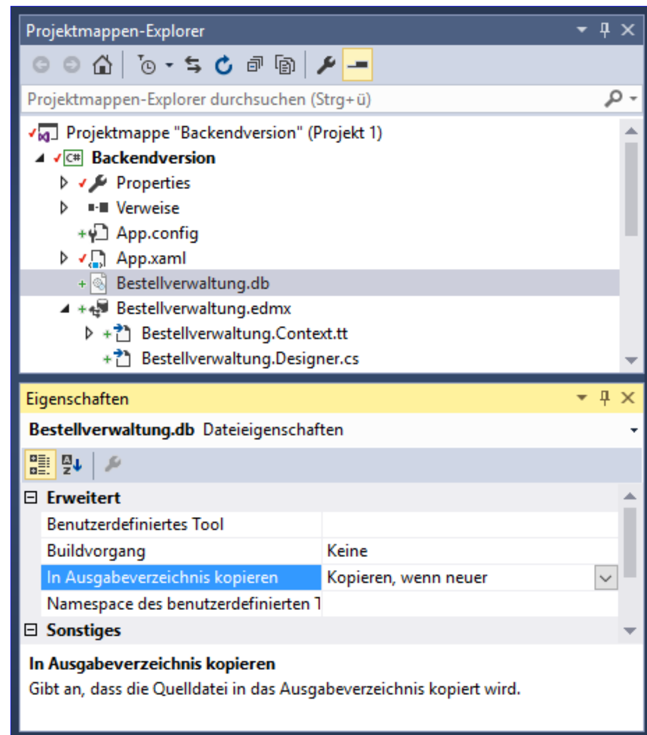

Bild 1: Eigenschaften der Datenbankdatei

Ort von Backendänderungen bei der Entwicklung

Wenn Sie, wie wir in diesem Artikel, mit einem Backend auf Dateibasis arbeiten (also mit SQLite und nicht etwa mit SQL Server), müssen Sie sich entscheiden, an welcher Stelle Sie die Änderungen durchführen. Wir führen eine Version des Backends direkt im Projektordner, also im gleichen Verzeichnis wie etwa die **.xaml**- und **.cs**-Dateien (später werden wir diese gegebenenfalls auch einmal in eigenen Ordnern strukturieren, aber aktuell reicht ein Verzeichnis dazu aus). Dieses fügen wir auch zum Projektmappen-Explorer hinzu. Dann stellen wir die Eigenschaft In **Ausgabeverzeichnis kopieren** für diesen Eintrag auf **Kopieren, wenn neuer** ein (siehe Bild 1). Die Datenbankdatei wird dann nur erneut in das Ausgabeverzeichnis kopiert, wenn die Ausgangsdatei aktualisiert wurde und somit ein neueres Änderungsdatum als die im Ausgabeverzeichnis enthaltene Datei enthält.

Hinzufügen der Tabelle Versions

Nun starten wir **SQLite Studio**, um die Tabelle **Versions** zur Datenbank hinzuzufügen. Hier wählen Sie den Menübefehl **Datenbank|Datenbank hinzufügen** und wählen im folgenden Dateiauswahl-Dialog die Datei **Bestellverwaltung.db** aus unserem Projektverzeichnis aus. Klicken Sie dann doppelt auf den Eintrag für diese Datenbank, sodass die enthaltenen Tabellen angezeigt werden. Klicken Sie mit der rechten Maustaste auf den Eintrag **Tabellen** und wählen Sie den Kontextmenü-Eintrag **Tabelle erstellen** aus. Geben Sie oben als Tabellenname **Versions** an und fügen Sie mit der Schaltfläche **Add Column** eine neue Spalte zur Tabelle hinzu. Dieses soll das Primärschlüsselfeld mit dem Spaltennamen **VersionID** und dem Datentyp **INTEGER** sowie automatischem Hinzufügen der Feldwerte sein. Für das zweite Feld legen Sie als Spaltenname **Version** fest und geben als Datentyp **String** mit der Länge **50** an (sicher ist sicher). Danach sieht die Definition der Tabelle wie in Bild 2 aus

und Sie können mit einem Klick auf die Schaltfläche **Commit structure changes** die Änderungen in die Datei schreiben.

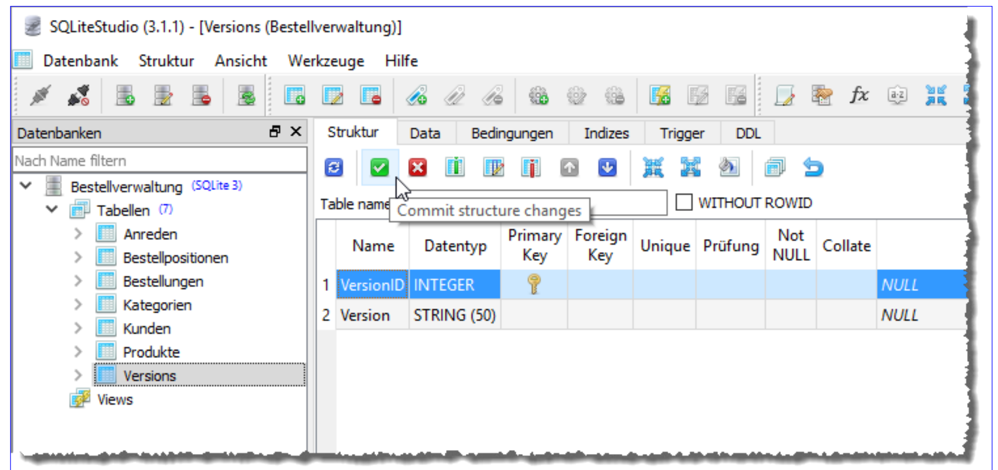

Bild 2: Anlegen der Tabelle **Versions**

Es erscheint dann der Dialog aus Bild 3 mit dem SQL-Code der auszuführenden Abfrage. Das ist recht praktisch, denn so können Sie den Code direkt kopieren und diesen später hervorholen, wenn Sie das Anlegen dieser Tabelle per C# programmieren wollen.

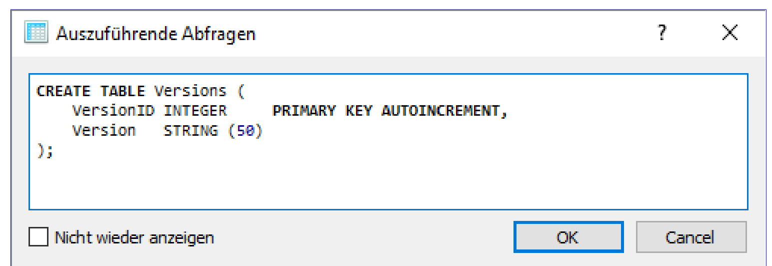

Bild 3: SQL-Code zum Erstellen der neuen Tabelle

Das waren schon alle Schritte, die wir von hier aus erledigen wollen.

Geändertes Datenmodell in Entity Data Model übernehmen

Danach müssen wir das Entity Data Model an das neue Datenmodell anpassen. Dazu öffnen Sie dieses per Doppelklick auf de Eintrag **Bestellverwaltung.edmx** im Projektmappen-Explorer. Dies öffnet die Übersicht, die per Klick mit der rechten Maustaste den Kontextmenü-Befehl **Modell aus der Datenbank aktualisieren...** offenbart (siehe Bild 4). Es erscheint dann nach einigen Sekunden der Update-Assistent und bittet um die Auswahl der Datenverbindung (eventuell erkennt er auch die korrekte Verbindung und zeigt direkt den folgenden

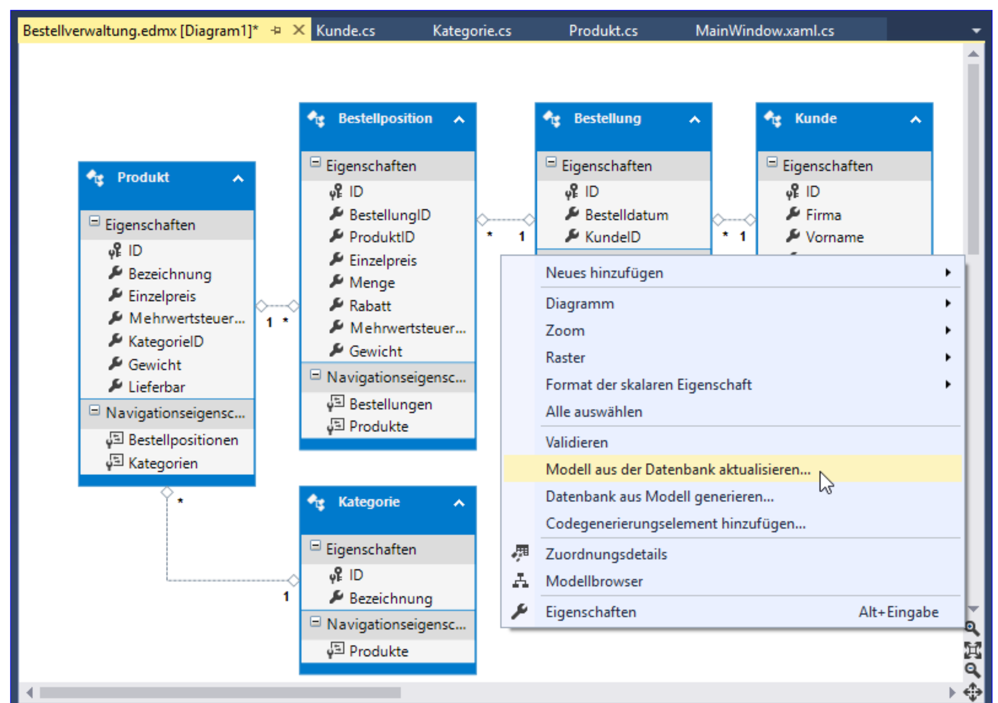

Bild 4: Aktualisieren des Modells aus der Datenbank

Dialog an). Die vorhandene Einstellung können Sie in der Regel beibehalten und mit einem Klick auf die Schaltfläche **Weiter** den nächsten Schritt aufrufen.

Nun kann es zu folgendem kommen: Es erscheint der Schritt mit der Überschrift **Wählen Sie Ihre Datenbankobjekte und Einstellungen.**, aber es lässt sich keine der Optionen **Tabellen**, **Sichten** oder **Gespeicherte Prozeduren und Funktionen** aktivieren (siehe Bild 5). Der Grund für diesen Zustand ist vermutlich, dass es tatsächlich keine zu aktualisierenden Objekte gibt. Das kann zwei Ursachen haben:

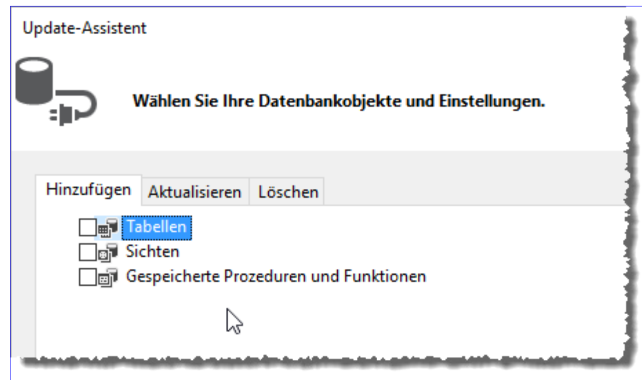

Bild 5: Es stehen keine Objekte zur Aktualisierung zur Verfügung.

- Sie haben die Änderungen nicht in die Datenbank übernommen. Prüfen Sie, ob zum Beispiel die neue Tabelle Versions in der Liste der Tabellen in **SQLite Studio** angezeigt wird.
- Sie haben die Änderungen an einer anderen Datenbank vorgenommen, als an der, die Sie nun zum Übertragen der Änderungen in das Entity Data Model verwenden wollen.

Wenn Sie den ersten Fall ausschließen können, sollten Sie im Update-Assistenten für das Entity Data Model nochmal zum ersten Schritt zurückkehren. Hier stellen Sie dann unter Umständen fest, dass Sie mehrere Verbindungen über das Kombinationsfeld auswählen können und vielleicht die falsche Datenbank ausgewählt haben. Hier sollten Sie dann sichergehen, dass Sie die richtige gewählt haben – beispielsweise, indem Sie unten unter Verbindungszeichenfolge prüfen, ob dort auch tatsächlich die Datenbankdatei im richtigen Verzeichnis angegeben ist (siehe Bild 6).

Achtung: Wenn Sie mit den Beispieldateien aus dem Download arbeiten, müssen Sie auf jeden Fall eine neue Verbindung anlegen und die Quelldatenbank neu auswählen – außer, Sie verwenden zufällig genau die gleiche Verzeichnisstruktur wie wir.

Wenn Sie danach einen Schritt weitergehen, erscheint die Tabelle **Versions** auch in der Liste der zur Aktualisierung bereitstehenden Tabellen und Sie können diese nun auswählen (siehe Bild 7). Klicken Sie dann auf **Fertigstellen**, erscheint schon kurz danach die Tabelle **Versions** im Entity Data Model.

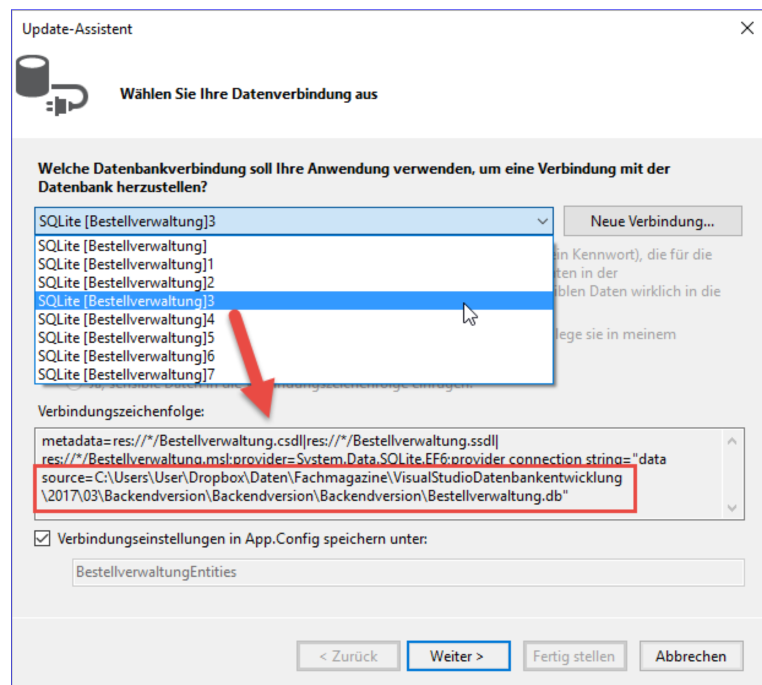

Bild 6: Prüfen der korrekten Datenverbindung anhand der Verbindungszeichenfolge

Sie können somit auch über das Entity Data Model auf die Tabelle und ihre Inhalte zugreifen.

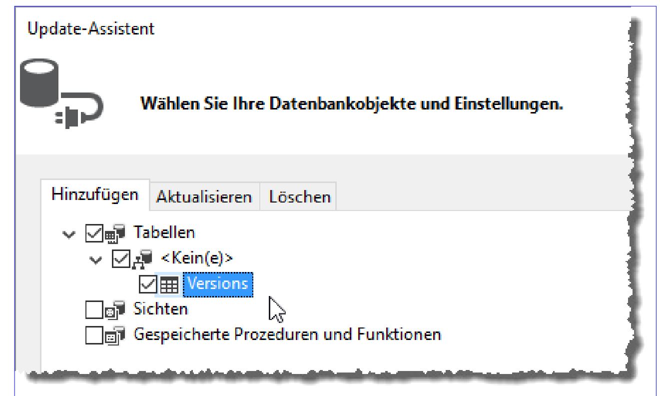

Bild 7: Einbinden der Tabelle **Versions** in das Entity Data Model

Code zum Prüfen der Version des Backends

Nun wollen wir Code zur Anwendung hinzufügen, der beim Öffnen prüft, ob die richtige Version des Backends vorliegt. Dabei unterscheiden wir zwei Fälle:

- Das Backend enthält noch gar keine Tabelle **Versions**. Dann muss diese zunächst angelegt werden. Danach sollen alle Updates des Backends bis zum aktuellen Versionsstand ausgeführt werden.
- Die Tabelle **Versions** liegt bereits vor. Dann prüfen wir den Versionsstand des Backends und führen alle Updates des Backends bis zum aktuellen Versionsstand aus.

Ort des Update-Codes

Bevor wir mit der Programmierung beginnen, müssen wir uns überlegen, wohin wir den Code schreiben, der die Version prüfen und eventuell notwendige Updates durchführt. Im Artikel **Entity Framework: SQLite verknüpfen** haben wir ja bereits eine Funktion programmiert, die beim Anzeigen von **MainWindow.xaml** ausgeführt wird und prüft, ob das Backend sich überhaupt an Ort und Stelle befindet und den Benutzer gegebenenfalls bittet, dieses auszuwählen. Dies ist ein geeigneter Ort, um auch die nachfolgend vorgestellten Methoden aufzurufen.

Prüfen, ob die Tabelle Versions vorhanden ist

Um zu prüfen, ob die Tabelle vorliegt, versuchen wir einfach, den Wert des Feldes **Version** zu lesen. Dabei kann es drei Fälle geben:

- Die Tabelle ist nicht vorhanden, was einen Fehler auslöst.
- Die Tabelle ist vorhanden, aber enthält keinen Datensatz oder einen Datensatz ohne einen Wert im Feld **Version**.
- Die Tabelle ist vorhanden und enthält einen Datensatz mit einem Wert.

Im ersten Fall wollen wir die Tabelle neu erstellen und die initiale Versionsnummer eintragen. Der zweite Fall ist kompliziert, weil wir nicht wissen, welche Version das Backend hat. Wir geben dann eine Meldung aus, dass die Versionsummer für das Backend nicht definiert ist und verweisen auf den Hersteller der Anwendung. Im dritten Fall ziehen wir die Versionsnummer heran, um noch nicht ausgeführte Updates zu erledigen. Der entsprechende **Try...Catch**-Block sieht wie folgt aus:

```
try {
    var versionCount = dbContext.Versions.Count();
    if (versionCount==0) {
        MessageBox.Show("Tabelle Versions vorhanden, aber leer.");
```

```
    }
    else {
        var versions = dbContext.Versions.First();
        string version = versions.Version;
        MessageBox.Show("Version gefunden: " + version);
    }
}
catch (System.Data.Entity.Core.EntityCommandExecutionException e) {
    MessageBox.Show("Tabelle nicht vorhanden.");
}
```

Diese greift mit **dbContext.Versions.Count()** auf die Anzahl der Einträge in der Entität **Versions** zu. Ist diese Anzahl gleich **0**, erscheint die erste Meldung, dass die Tabelle zwar vorhanden, aber leer ist. Anderenfalls liegt ein Datensatz vor, wobei wir den Wert des Feldes **Version** auslesen und diesen ausgeben. Den **Try...Catch**-Block haben wir erstellt, weil es sein kann, dass die Tabelle **Versions** gar nicht im Backend vorhanden ist. Dies löst dann einen Fehler des Typs **EntityCommandExecutionException** aus, woraufhin wir ebenfalls eine entsprechende Meldung liefern.

Tabelle Versions anlegen, falls noch nicht vorhanden

Nun wollen wir die einzelnen Verzweigungen der **Try...Catch**-Fehlerbehandlung und der darin integrierten **if...else**-Bedingung mit Leben füllen. Dabei schauen wir uns zunächst den letzten Fall, also den **Catch**-Zweig an. Hier wollen wir die Tabelle **Versions** neu anlegen. Den dazu notwendigen SQL-Befehl haben Sie vielleicht noch vom Anlegen der Tabelle im **SQLite Studio** in der Zwischenablage liegen:

```
CREATE TABLE Versions (
    VersionID INTEGER     PRIMARY KEY AUTOINCREMENT,
    Version   STRING (50)
);
```

Damit ersetzen wir nun in der oben vorgestellten Struktur die **MessageBox**-Anweisungen durch die Anweisungen zum Erstellen beziehungsweise Ergänzen der Tabelle **Versions**. Dazu deklarieren wir zunächst eine Variable, welche die Version aufnehmen soll:

```
string version;
```

Dann steigen wir in den **Try**-Teil ein, wo wir die Anzahl der Datensätze ermitteln. Ist diese **0**, aber es gibt keinen Fehler beim Zugriff auf die Tabelle über die Entität, fügen wir mit einer **INSERT INTO**-Abfrage als Parameter der **ExecuteSqlCommand**-Methode des **Database**-Objekts von **dbContext** einen Datensatz zur Tabelle hinzu, dessen Feld **Version** den Wert **1** enthält. Außerdem stellen wir die Variable **Version** ebenfalls auf den Wert **1** ein:

```
try {
    var versionCount = dbContext.Versions.Count();
    if (versionCount==0) {
```