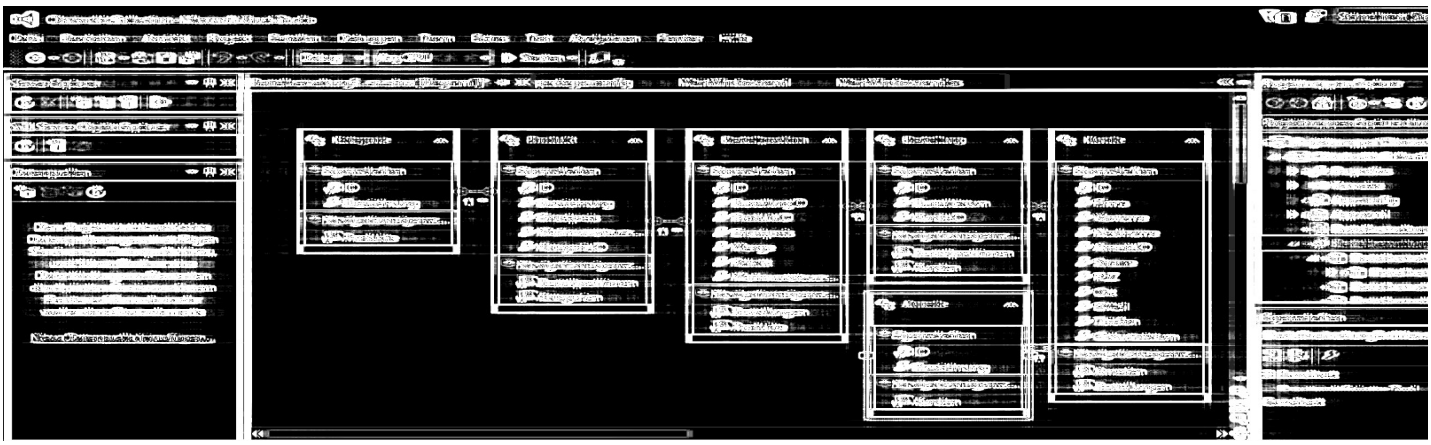
AUSGABE 01/2017

DATENBANK ENTWICKLER

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



TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER
© 2016-2017 André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

WPF-Basics: Namespaces

Bislang haben wir immer mit WPF-Projekten gearbeitet, ohne uns Gedanken um das Stammelement einer .xaml-Datei zu machen. Dieses Element, das in der Regel Window heißt, enthält zum Beispiel einige Namespace-Angaben und weitere Informationen. In diesem Artikel wollen wir erläutern, was es mit diesen Attributen auf sich hat und wie sich diese auf die Definition der vollständigen .xaml-Datei auswirken.

Window oder ...

Wie schon erwähnt, ist das Stammelement einer **.xaml**-Datei meist ein **Window**-Element – also eines, das ein Fenster beschreibt. Es gibt allerdings noch einige andere Elementtypen, zum Beispiel das in diesem Magazin auch schon verwendete **Page**-Element, mit dem man eine Seite beschreibt, die innerhalb anderer Elemente eingebunden werden kann.

Wenn Sie eine neue WPF-Anwendung anlegen, erhalten Sie direkt ein Fenster mit dem Titel **MainWindow**, dass einige für den Einsteiger kryptisch anmutende Elemente enthält (siehe Listing 1).

Das erste Attribut namens **x:Class** gibt an, welche Klasse die Code behind-Klasse der aktuellen **.xaml**-Datei ist. Der Teil vor dem Punkt gibt den Namespace an, der Teil hinter dem Punkt den Klassennamen.

Im entsprechenden Code-Modul finden Sie als oberstes Element das **Namespace**-Objekt namens **Experimente**, darunter die Klasse mit dem unter **x:Class** angegebenen Namen **MainWindow**. Der Doppelpunkt und die folgende Bezeichnung **Window** geben an, dass die Klasse **MainWindow** von der Klasse **Window** abgeleitet ist und somit alle Eigenschaften, Methoden und Ereignisse der Klasse **Window** besitzt. Die Methode **public MainWindow** ist die sogenannte Konstruktor-Methode, die automatisch beim Initialisieren der Klasse aufgerufen wird:

```
namespace Experimente {
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }
    }
}
```

Beim Starten wird in der Datei **App.xaml** die unter **StartupUri** angegebene **.xaml**-Datei eingelesen, hier **MainWindow.xaml**, die dortige unter **x:Class** angegebene Code behind-Klasse gesucht und die Konstruktor-Methode aufgerufen. Diese wiederum ruft die Methode **InitializeComponent** auf, die grob zusammengefasst aus der in der **.xaml**-Datei enthaltenen Definition das entsprechende Objekt zusammenstellt, also beispielsweise ein **Window**-Element mit all seinen Steuerelementen.

```
<Window x:Class="Experimente.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:Experimente"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Listing 1: Namespaces eines **Window**-Elements

xmlns-Attribute

Wozu sind nun die ganzen **xmlns…**-Attribute gut? Diese legen fest, welche Elemente Sie in der **.xaml**-Datei zur Definition des Aussehens etwa eines **Window**-Objekts nutzen können. Das erste **xmlns**-Attribut enthält den folgenden Wert, der jedoch nicht etwa tatsächlich eine Seite anzeigt, wenn Sie die URL in den Webbrowser eingeben:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Vermutlich kennen Sie schon die **using**-Anweisung, mit der Sie einer C#-Klasse die dort zu verwendenden Namespaces zuweisen und damit den Zugriff auf die dort enthaltenen Klassen ermöglichen.

Das oben genannte Attribut weist der **.xaml**-Datei nicht direkt einen Namespace zu wie es in einer C#-Klasse der Fall ist. Der Namespace **http://schemas.microsoft.com/winfx/2006/xaml/presentation** fasst aber einige der Klassen, die auch standardmäßig in einer C#-Klasse verfügbar gemacht werden, zusammen – zum Beispiel **System. Windows. System.Data** und noch einige mehr.

Wenn Sie dieses **xmlns**-Attribut einmal entfernen, werden Sie feststellen, dass beispielsweise das **Window**-Element in der **.xaml**-Datei als fehlerhaft unterstrichen wird. Der Fehler lautet in diesem Fall **The default namespace is not defined** (siehe Bild 1).

Wie in der Fehlermeldung ersichtlich, wird der hier fehlende Namespace auch als Standardnamespace bezeichnet. Wir benötigen diesen also, um Elemente wie **Window**, **Grid** und viele mehr für die Definition der Benutzeroberfläche heranzuziehen.

Der zweite Namespace heißt ähnlich wie der erste, aber er unterscheidet sich erstens in der Bezeichnung des Attributs, das noch ein durch einen Doppelpunkt angeführtes **x** enthält, und zweitens durch das Fehlen des letzten Teils **/presentation**:

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Der Teil hinter dem Doppelpunkt, also das **x**, heißt in diesem Zusammenhang Präfix. Wenn Sie nun ein Element einer von diesem zweiten Namespace referenzierten Klassen im **.xaml**-Code verwenden wollen, das nicht im Standardnamespace enthalten ist, müssen Sie diesem ebenfalls das Präfix voranstellen. Das ist sogar schon im obigen **.xaml**-Code der Fall - das **Class**-Attribut des **Window**-Elements ist nämlich mit einem führenden **x** versehen:

Bild 1: Fehler durch den fehlenden Standardnamespace

```
<Window x:Class="Experimente.MainWindow" …
```

Wenn Sie das **x** entfernen, erhalten Sie ebenfalls einen Fehler als wenn Sie die Definition des entsprechenden Namespace entfernen. Die Fehlermeldung lautet dann: **The namespace prefix „x“ is not defined**.

Der mit dem Präfix **d** versehene Namespace enthält Elemente, die speziell für die Entwurfszeit vorgesehen sind. Dabei handelt es sich beispielsweise um Elemente, die zur Anzeige von Daten zur Entwurfszeit in Visual Studio und Expression Blend dienen.

Das Präfix **mc** steht für einen Namespace, der Elemente für einen Kompatibilitätsmodus liefert. Die drei folgenden Namespace-Definitionen sorgen so etwa dafür, dass die Elemente, die zur Entwurfszeit interpretiert werden sollen, zur Laufzeit ignoriert werden:

```
xmlns:d="http://schemas.microsoft.com/expression/
blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compa-
tibility/2006"
mc:Ignorable="d"
```

Fehlt nur noch ein Element, dass mit dem Präfix **local** ausgestattet ist und auf einen **clr**-Namespace namens **Experimente** verweist:

```
xmlns:local="clr-namespace:Experimente"
```

Experimente ist nicht aus Zufall der Name unseres aktuell verwendeten Namespace. Er wird unter dem Namen **local** angegeben, damit Sie bei Bedarf auf Elemente des hier angegebenen Namespace zugreifen können. Wenn Sie zum Beispiel in der Code behind-Klasse **MainWindow** eine Konstante wie folgende deklarieren und den Inhalt als ToolTip-Text einer Schaltfläche anzeigen wollen, verwenden Sie den **local**-Namespace:

```
public static string ToolTipContent = "Beispiel-ToolTip";
```

Im **.xaml**-Code können Sie dann wie folgt auf den Inhalt dieser Konstanten zugreifen:

```
<Grid>
    <Button ToolTip="{x:Static local:MainWindow.ToolTip-
Content}"></Button>
</Grid>
```

Mit **Static** verwenden Sie dabei auch noch ein Element des mit dem Präfix **x** versehenen Namespace.

Zusammenfassung und Ausblick

Dieser Artikel erläutert, was es mit den Namespace-Angaben im obersten Element einer WPF-Datei auf sich hat. Sie wissen nun, wozu diese dienen und was geschieht, wenn Sie diese einfach weglassen.

Sie haben auch erfahren, dass die für die eingebauten Namespaces wie **http://schemas.microsoft.com/expression/blend/2008** angegebenen Adressen nicht wirklich existieren, sondern nur Platzhalter sind, die intern angeben, welche Namespaces hier zum Einsatz kommen sollen.

Unter C# sind die Namespaces explizit anzugeben, wozu Sie die **using**-Anweisung nutzen.

Die Schnittstelle zwischen den Namespaces in den .xaml-Dateien und den C#-Dateien sind die benutzerdefinierten Namespaces, hier zuerst der in **.xaml** unter **xmlns:local="clr-namespace:Experimente"** angegebene Namespace, der sich im C#-Code der dazu gehörenden Dateien in der folgenden Form manifestiert:

```
namespace Bestellverwaltung {
    ...
}
```

Quellcodeverwaltung mit Visual Studio und Git

Die Arbeit mit Visual Studio-Projekten bietet gegenüber Access einen interessanten Vorteil: Im Gegensatz zu Access, wo die einzelnen Objekte alle in der Access-Datei gespeichert wurden, finden Sie alle Objekte eines Visual Studio-Projekts als einzelne Dateien im Projektordner wieder. Das erleichtert vor allem die Verwaltung des Quellcodes in einer Quellcodeverwaltung. Eine solche erfasst nach Wunsch Zwischenstände der Entwicklung und bietet die Möglichkeit, später noch auf vorherige Stände zuzugreifen und diese wiederherzustellen, sollte sich die Programmierung in die falsche Richtung entwickelt haben. Dieser Artikel zeigt die Basics der Quellcodeverwaltung unter Visual Studio – vorerst für den Einsatz im stillen Kämmerlein eines Einzelentwicklers.

Bei einem üblichen WPF/C#-Projekt, wie Sie es im Rahmen dieses Magazins schon einige Male angelegt haben, taucht die Quellcodeverwaltung von Visual Studio nicht sichtbar in Erscheinung. Dies ist erst der Fall, wenn Sie das Kontextmenü der Projektmappe im Projektmappen-Explorer anzeigen – dort finden Sie den Eintrag **Projektmappe zur Quellcodeverwaltung hinzufügen** vor (siehe Bild 1).

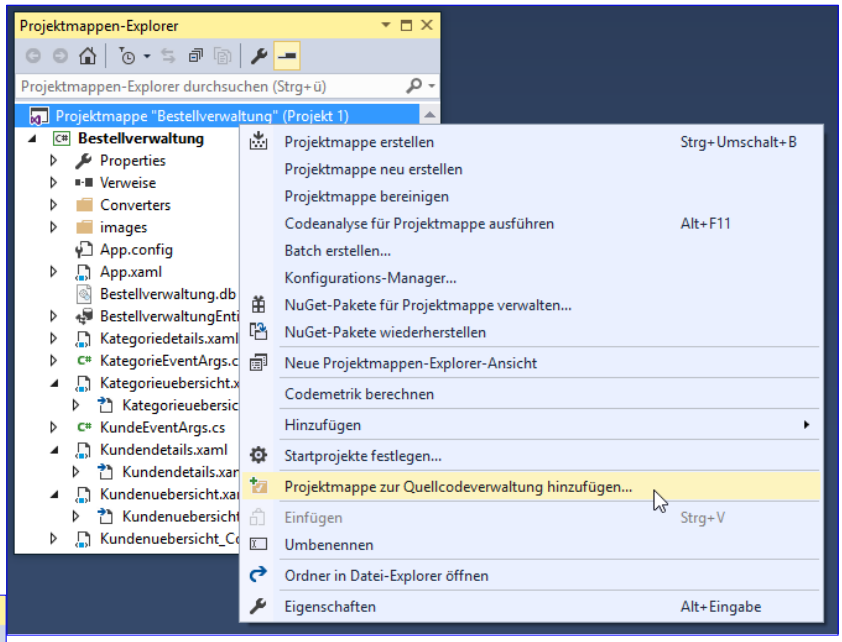


Bild 1: Hinzufügen der Projektmappe zu einer Quellcodeverwaltung

Nach dem Betätigen dieses Befehls geschieht nicht viel – außer dass die Dateien im Projektmappen-Explorer nun alle mit einem zusätzlichen Icon ausgestattet sind, dass beim Überfahren mit der Maus den Text **Eingecheckt** liefert (siehe Bild 2).

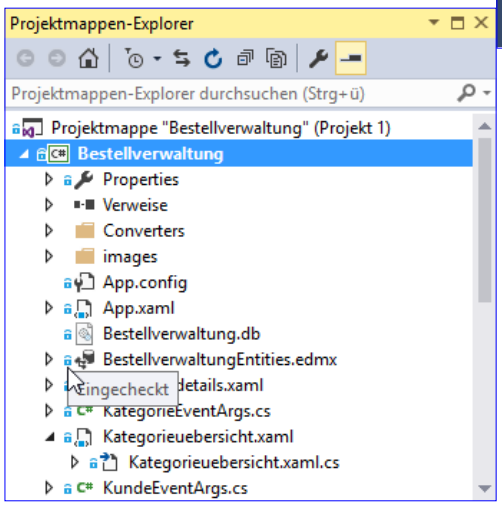


Bild 2: Nach dem Hinzufügen zur Quellcodeverwaltung

Das Hinzufügen der Projektmappe zur Quellcodeverwaltung sorgt also auch dafür, dass alle enthaltenen Dateien erstmalig eingecheckt werden. Einchecken bedeutet hierbei, dass die aktuelle Version der Dateien gespeichert wird. Dies war der Weg, um ein bestehendes Projekt zu einer Quellcodeverwaltung hinzuzufügen. Im Folgenden sehen wir uns allerdings die Vorgehensweise für ein neues, leeres WPF-Projekt an – dort werden wir dann ein paar Änderungen am Code vornehmen, einchecken, Änderungen betrachten, neue Dateien anlegen und mehr.

Neues Projekt direkt zur Quellcodeverwaltung hinzufügen

Wenn Sie über den Menüeintrag **Datei|Neu|Projekt...** ein neues Projekt erstellen, finden Sie im Dialog **Neues Projekt** rechts unten eine Option namens **Zur Quellcodeverwaltung hinzufügen** (siehe Bild 3). Aktivieren Sie diese Option und klicken dann auf **OK**, werden die neu erzeugten Projektdateien direkt zur Quellcodeverwaltung hinzugefügt.

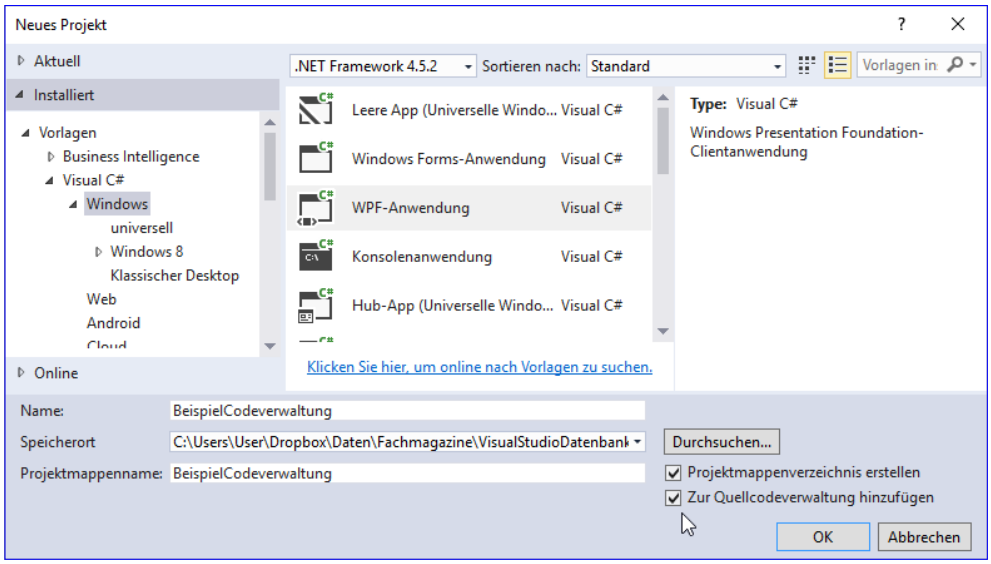


Bild 3: Anlegen eines neuen Projekts, das direkt zur Quellcodeverwaltung hinzugefügt werden soll

Der Projektmappen-Explorer zeigt dann direkt alle Dateien als eingecheckt an (siehe Bild 4).

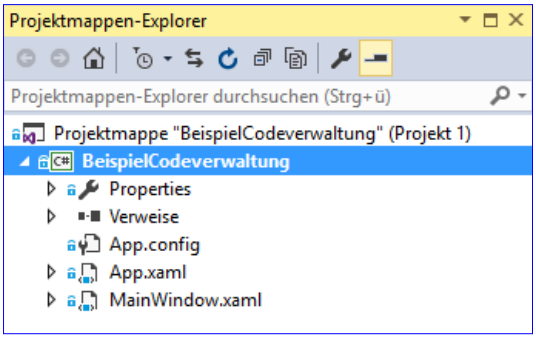


Bild 4: Ein frisches Objekt, direkt in die Quellcodeverwaltung eingecheckt

Bearbeiten einer Datei

Wenn Sie eine der eingecheckten und mit dem Schloss-Symbol versehenen Dateien bearbeiten, ändert sich das Symbol in einen roten Haken mit dem ToolTip-Text **Ausstehende Bearbeitung**.

Als Beispiel haben wir der noch unberührten Datei **MainWindow.xaml** ein neues Textfeld hinzugefügt, also von

```
<Grid>
</Grid>
```

zu

```
<Grid>
    <TextBox x:Name="txtBeispiel" Text="Beispiel"></TextBox>
</Grid>
```

Das Kontextmenü dieser Datei im Projektmappen-Explorer liefert nun einige weitere Optionen im Kontextmenü (siehe Bild 5):

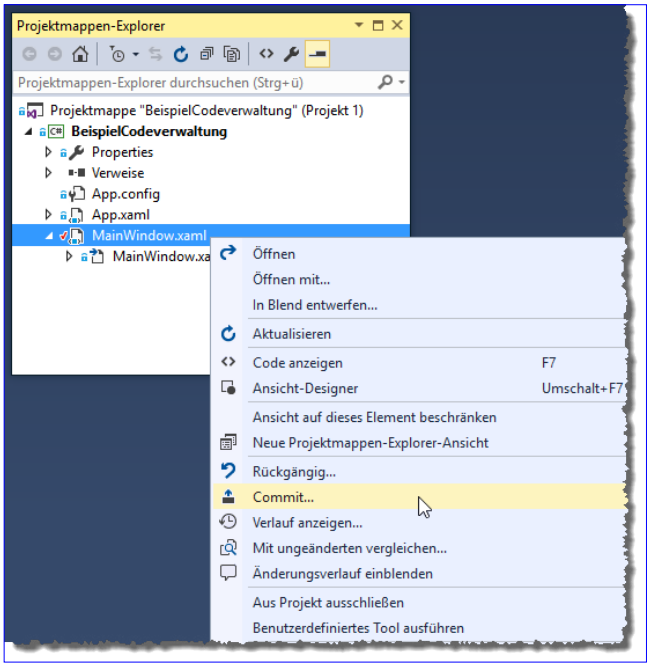


Bild 5: Optionen einer geänderten Datei

- **Commit**: Ruft den Bereich **Änderungen** auf, der alle seit dem letzten Commit durchgeführten Änderungen auflistet und die Möglichkeit bietet, eine Commitnachricht einzugeben. Dabei handelt es sich um einen benutzerdefinierten Text, mit dem Sie beispielsweise Informationen zu den getätigten Änderungen angeben können.

- **Verlauf anzeigen**: Zeigt eine Liste der bisherigen Commits für dieses Projekt an.

- **Mit ungeänderten vergleichen**: Öffnet einen Bereich, in dem die aktuelle, geänderte Version mit der Version vergleichen wird, wie sie zuvor aussah.

- **Änderungsverlauf einblenden**: Zeigt alle Änderungen in einer Ansicht an.

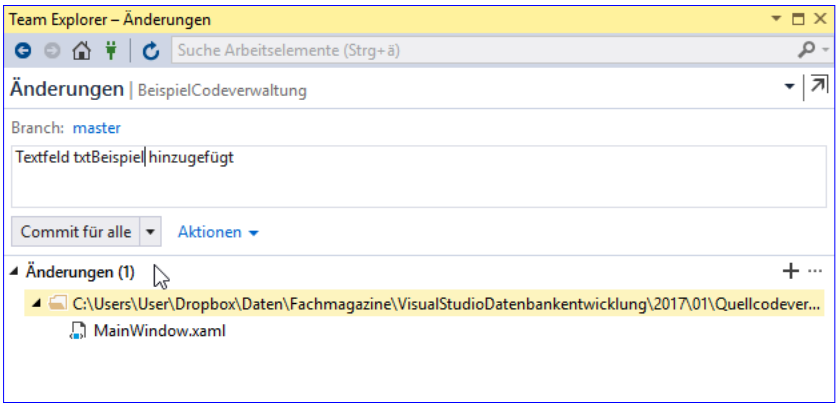


Bild 6: Angaben vor dem Commit

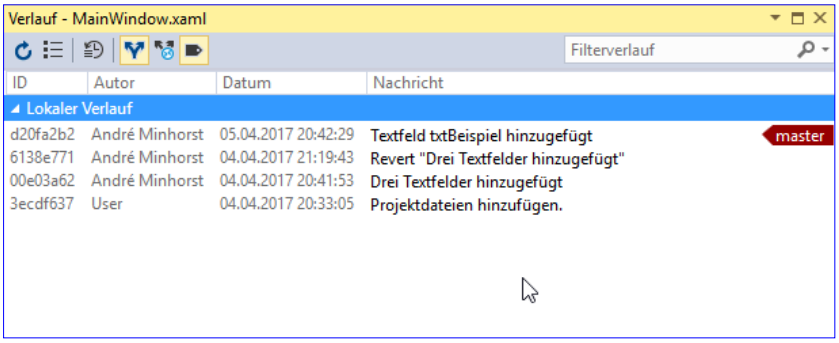


Bild 7: Änderungsverlauf einer Datei

Diese Bereiche schauen wir uns in den folgenden Abschnitten im Detail an.

Änderungen-Dialog

Der Bereich **Team-Explorer - Änderungen** zeigt jeweils die seit dem letzten Commit geänderten Dateien in einer Liste im unteren Bereich an. Diesen Bereich können Sie immer mit dem Menübefehl **Ansicht|Team Explorer** oder mit der Tastenkombination **Strg + ^, Strg + M** einblenden. Sobald eine Änderung vorliegt, können Sie eine Commitnachricht eingeben und die Änderungen mit einem Klick auf die Schaltfläche **Commit für alle** durchführen (siehe Bild 6). Das erste offensichtliche Resultat ist, dass das Icon der geänderten Dateien wieder vom roten Haken in das blaue Schloss geändert wird. Die Schaltfläche **Commit für alle** wird übrigens nur bei Vorliegen von Änderungen und bei Vorhandensein einer Commitnachricht aktiviert.

Die Schaltfläche bietet übrigens noch weitere Funktionen an, wenn Sie auf den Pfeil nach unten klicken:

- **Commit für alle**: Überträgt die Änderungen in das lokale Repository.

- **Commit für alle und Push**: Überträgt die Änderungen in das lokale Repository und »pusht« diese auch gleich an eventuell angeschlossene Remote-Repositorys (weitere Informationen weiter unten).

- **Commit für alle und Sync**: Überträgt die Änderungen in das lokale Repository, »synct« die Änderungen von einem eventuell angeschlossenen Remote-Repository und »pusht« die Änderungen dann in das Remote-Repository. Sollten dabei an der

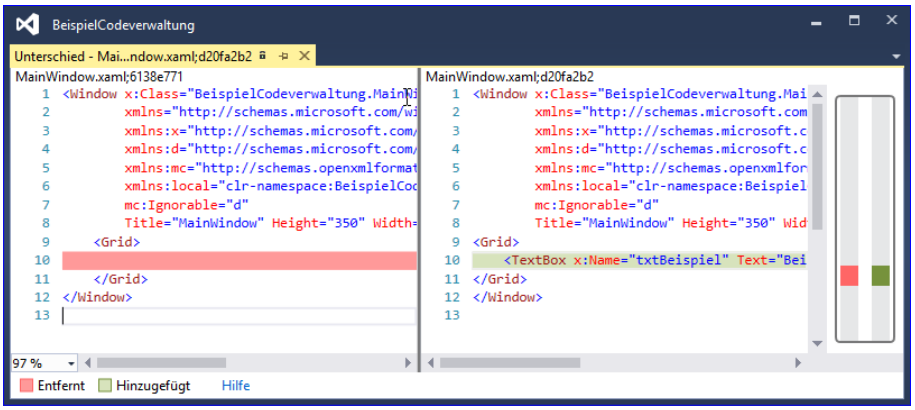


Bild 8: Vergleich der aktuellen mit der vorherigen Version

Bild 9: Änderungsverlauf der vorherigen und der ausgewählten Version

gleichen Datei auf beiden Seiten Änderungen vorgenommen worden sein, müssen eventuell auftretende Konflikte gelöst werden (siehe weiter unten).

Verlauf anzeigen

Wenn Sie für eine Datei eine oder mehrere Commits durchgeführt haben, können Sie sich den Änderungsverlauf ansehen. Dazu wählen Sie aus dem Kontextmenü der jeweiligen Datei den Eintrag **Verlauf anzeigen...** auf. Dies öffnet den Dialog **Verlauf - <Dateiname>** (siehe Bild 7). Dieser zeigt alle bisherigen Commits samt den angegebenen Commit-Nachrichten an.

Wenn Sie mit der rechten Maustaste auf einen der Commit-Einträge klicken, finden Sie wiederum einige interessante Befehle vor. Mit **Commitdetails anzeigen** öffnen Sie beispielsweise wieder den **Team Explorer**-Bereich, der dann die Details zu diesem Commit inklusive der betroffenen Dateien anzeigt. **Mit vorherigen vergleichen...** öffnet den **Unterschied**-Bereich, den Sie in Bild 8 sehen. Hier erscheinen die ausgewählte und die vorherige Version, wobei die gelöschten, geänderten und neu erstellten Zeilen jeweils farbig hervorgehoben werden.

Der Kontextmenü-Eintrag **Änderungsverlauf einblenden** zeigt die Änderungen in einer einzigen Textdatei. Das sieht für eine geänderte Zeile allein noch recht überschaubar aus, wie Bild 9 zeigt.

Die Befehle **Neuer Branch...** und **Tag erstellen** schauen wir uns weiter unten an. Der Befehl **Zurücksetzen** sowie die beiden Befehle **Zurücksetzen\|Zurücksetzen und Änderungen beibehalten** und **Zurücksetzen\|Zurücksetzen und Änderungen löschen** sorgen dafür, das nur ein einziger Commit rückgängig gemacht wird, aber nicht das ganze Projekt auf einen Stand zu diesem Commit gebracht wird. Dafür gibt es besses Methoden – siehe weiter unten unter **Branches**.

Vollständigen Verlauf des Projekts anzeigen

Wenn Sie nicht nur den Verlauf einer einzelnen Datei, sondern die Commits des kompletten Projekts ansehen wollen, öffnen Sie den **Team Explorer**, wechseln zur Startseite (mit der **Home**-Schaltfläche) und wechseln dann zum Bereich **Branches**. Auch wenn wir bisher noch nicht über Branches gesprochen haben, finden Sie dort den Eintrag **Master**. Dieser liefert über das Kontextmenü den Befehl **Verlauf anzeigen...**, mit dem Sie eine Übersicht der Commits wie für einzelne Datei öffnen können – nur eben diesmal mit allen Commits, die für die Dateien des Projekts ausgeführt wurden.

Weitere Optionen im Team Explorer-Bereich

Wenn Sie im Kontextmenü den Eintrag **Commit** für das komplette Projekt oder auch für eine einzelne Datei aufrufen, erscheint der Team Explorer-Bereich und zeigt die Seite **Änderungen** an. Der Team-Explorer hat aber auch noch weitere Bereiche, die Sie durch einen Klick auf die **Home**-Schaltfläche einblenden können (siehe Bild 10).

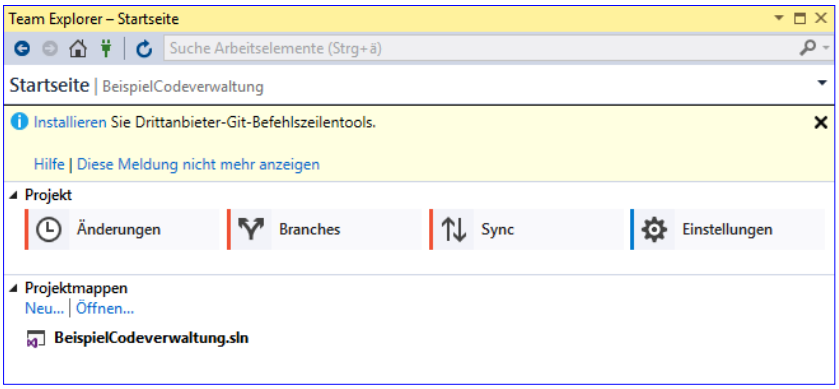


Bild 10: Die Team Explorer-Startseite mit Schaltflächen zum Aufrufen der verschiedenen Bereiche

Hier finden Sie unter Projekt die vier folgenden Schaltflächen:

- **Änderungen**: der Bereich, den Sie bereits kennen gelernt haben
- **Branches**: Zeigt die verschiedenen Branches des Projekts an. Mehr dazu gleich im Anschluss.
- **Sync**: Zeigt die Befehle an, um aktuelle Änderungen vom lokalen Repository in das Remote-Repository zu »pushen« oder Änderungen vom Remote-Repository in das lokale Repository zu »pullen«.
- **Einstellungen**: Diverse Einstellungen rund um die zu verwendenden Tools, das Remote-Repository und verschiedene Sicherheitseinstellungen, deren Erläuterung den Rahmen dieses Artikels sprengen würde.

Branch

Bei Branch denken Sie vielleicht jetzt an ausgedehnte Frühstücksorgien, aber da müssen wir Sie enttäuschen. Es handelt sich viel mehr um eine Möglichkeit, den Zustand des Projekts zum Zeitpunkt eines beliebigen Commits herzustellen und von dort aus weiterzuarbeiten, ohne Gefahr zu laufen, die danach erledigten Schritte zu verlieren. Ein Branch ist also eine Möglichkeit, zu einem bestimmten Zeitpunkt eines Projekts zu erkennen, dass man vielleicht in eine falsche Richtung gelaufen ist und vielleicht ein paar Schritte zurückgehen will, um einen anderen Weg auszuprobieren. Wenn Sie ohne Quellcodeverwaltung etwa unter Access gearbeitet haben, war es erstens gar nicht möglich, auf Zwischenstände zurückzugreifen, wenn Sie nicht regelmäßig Kopien der Access-Da-

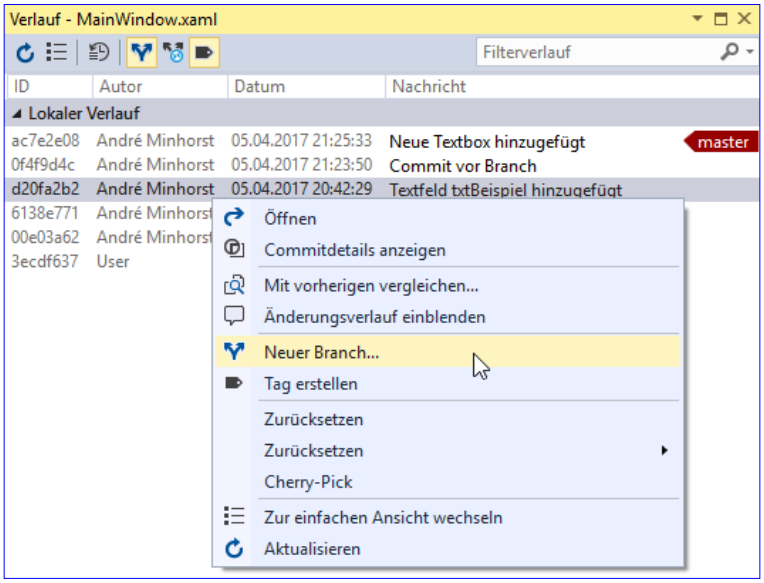


Bild 11: Anlegen eines neuen Branches

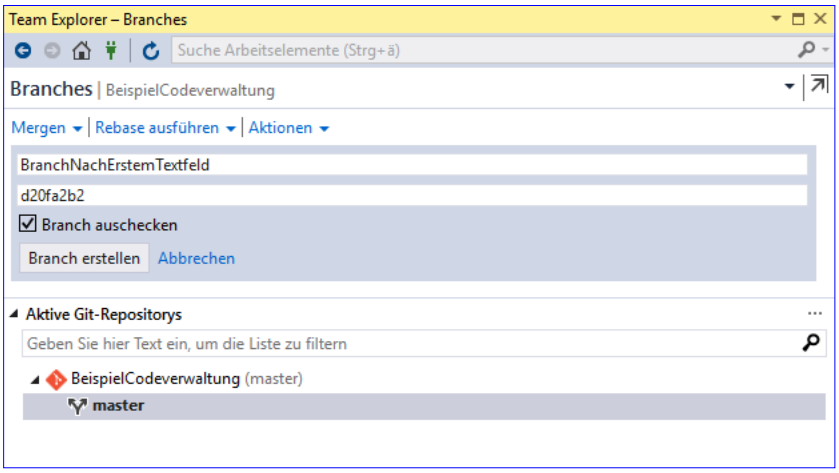


Bild 12: Eingabe der Daten des neuen Branches

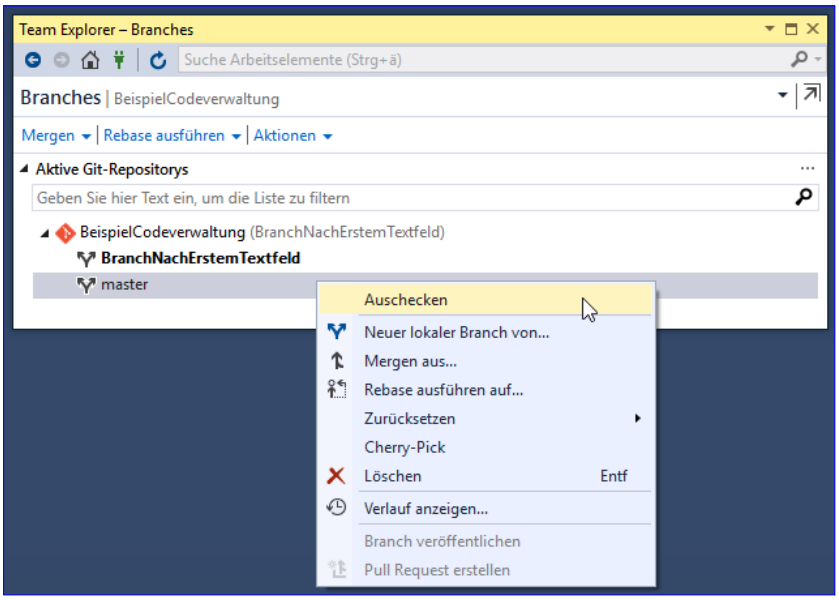


Bild 13: Anzeige des neuen Branches und Wiederherstellen des vorherigen Zustands

tenbank angelegt haben. Bei den Branches eines Projekts erhalten Sie genau diese Möglichkeit. Wir schauen uns das an einem einfachen Beispiel an. Wir haben unser Projekt nun um ein weiteres Textfeld in **MainWindow.xaml** erweitert:

```
<Grid>
    <TextBox x:Name="txtBeispiel"
Text="Beispiel"></TextBox>
    <TextBox x:Name="txtBeispiel_Branch"
Text="Branch"></TextBox>
</Grid>
```

Nun haben wir festgestellt, dass wir zuvor eine andere Variante ausprobieren wollen – ausgehend von dem Commit, den wir nach dem Anlegen des ersten, aber vor dem Anlegen des zweiten Textfeldes durchgeführt haben. Voraussetzung dafür, dass wir später wieder auf den aktuellen Stand zugreifen können, ist, dass wir einen Commit für diesen Stand ausführen.

Danach zeigen wir unsere Commits im Verlauf der Datei **MainWindow.xaml** an (oder auch im Gesamtverlauf des Branches **master**, der standardmäßig angelegt wird).

Klicken Sie mit der rechten Maustaste auf den Commit, dessen Stand Sie als Ausgangspunkt für eine alternative Weiterentwicklung nutzen wollen und wählen Sie dort den Eintrag **Neuer Branch...** aus (siehe Bild 11).

Hier geben Sie nun eine Bezeichnung für den neue Branch ein (siehe Bild 12). Wenn Sie die Option **Branch auschecken aktiviert** lassen, wird direkt dieser Stand aus der Quellcodeverwaltung ausgecheckt. Das heißt, dass alle Dateien mit dem Stand zu diesem Zeitpunkt in den Projektmappen-Explorer geladen werden. So hat beispielsweise unsere Datei **MainWindow.xaml** nur eine TextBox – nämlich die, welche wir zu Beginn angelegt haben. Alle Änderungen nach diesem Commit sind verschwunden.

Hilfe! Was ist, wenn ich nun erkenne, dass der erste eingeschlagene Weg doch der bessere war? Kein Problem: Der Team Explorer zeigt nun neben dem Branch namens **master** auch noch den neu erstellten Branch an (siehe Bild 13). Mit dem

SQLite als Backend

Der SQL Server kommt als mächtiges Installationspaket, und selbst die schlanke Variante LocalDb muss erst einmal installiert werden, bevor die Anwendung auf dem Zielsystem auf Daten zugreifen kann. Schlank und ohne eigene Installation – das wären gute Eigenschaften für ein Datenbanksystem für den Einsatz in einfachen Desktop-Anwendungen. Die Lösung lautet SQLite: Diese Programmbibliothek können Sie einfach zum Projekt hinzufügen und es mit der Anwendung auf dem Zielrechner installieren. Und das Beste: SQLite unterstützt sogar das Entity Framework.

SQLite – wozu?

Unser primärer Anlass, einmal nach Alternativen zum SQL Server oder zu LocalDb zu suchen, sind die Beispielprojekte zu diesem Magazin. Wenn diese eine Datenbank enthalten, sind immer der eine oder andere Schritt nötig, bevor Sie die Beispiele auf Ihrem Rechner ausprobieren können. Und das ist selbst der Fall, wenn bereits eines der Microsoft-Datenbanksysteme auf ihrem Rechner installiert ist. Welche Voraussetzungen sollte das gesuchte Datenbanksystem also haben? Am wichtigsten ist natürlich, dass wir damit weiterhin die Techniken vorstellen können, die Sie auch bei Verwendung von Microsoft-Datenbanken nutzen – also Beispielsweise der Einsatz des Entity Framework als Schnittstelle zwischen Anwendung und Datenbank. Der zweite Punkt ist, dass eine möglichst einfache Installation der Programmdateien des Datenbanksystems möglich ist oder am besten gar keine. Stattdessen sollte das Datenbanksystem am besten direkt im Projektordner an den Zielrechner übergeben werden und bereits fertig konfiguriert sein.

Wir hätten uns natürlich auch mit einer Access-Datenbank als Backend zufrieden gegeben, denn Access dürfte bei der Zielgruppe dieses Magazins auf dem Rechner installiert sein. Allerdings erfüllt eine Access-Datei gleich zwei Punkte nicht: Erstens ist für die Verwendung immer noch eine Access-Installation auf dem Zielrechner nötig. Dies ist zwar aufgrund der Runtime-Version, die kostenlos weitergegeben werden darf, möglich, aber besser ist natürlich ein Datenbanksystem, das komplett ohne Installation auskommt. Zweitens können wir zwar per ADO.NET auf Access-Datenbanken zugreifen, aber spätestens, wenn das Entity Framework zum Einsatz kommen soll (und das Entity Framework wird uns in diesem Magazin noch eine Weile begleiten), wird es eng – es gibt nämlich keine Implementierung, die den Zugriff auf die Tabellen einer Access-Datenbank erlaubt. Darüber sind wir dann auf SQLite gestoßen. Diese Datenbank kommt im Wesentlichen in Form einer Programmbibliothek, die außerdem auch noch nur wenige hundert Kilobyte groß ist. Und es hat eine weitere Gemeinsamkeit mit Access: Die eigentlichen Datenbanken kommen in Form einer einzigen Datei, in diesem Fall mit der Dateiendung **.db**.

SQLite ist nicht von ungefähr so kompakt: In der Regel wird es etwa in Apps für Mobiltelefone eingesetzt. Die Kompaktheit hat natürlich auch ihren Preis: SQLite enthält längst nicht alle Funktionen, die ein »großes« Datenbanksystem wie etwa SQL Server bietet. Für den Einsatz in unseren Beispieldatenbanken reichen die Funktionen aber allemal aus. In den folgenden Abschnitten erfahren Sie verschiedene Dinge über SQLite:

- mit welchem Tool Sie SQLite-Datenbanken erstellen und bearbeiten können,
- wie Sie SQL Server-Datenbanken nach SQLite migrieren,
- wie Sie SQLite in .NET-Projekten einsetzen und ein Entity Data Model auf Basis einer SQLite-Datenbank erstellen und
- wie Sie eine Anwendung mit SQLite-Datenbank weitergeben können.

SQLite-Tools

Bevor wir überhaupt mit unserer Anwendung auf eine SQLite-Datenbank zugreifen können, sollten wir uns eine solche erstellen. Dazu gibt es verschiedene Tools. Wir haben den **DB Browser for SQLite** verwendet, den Sie unter folgendem Link finden:

```
http://sqlitebrowser.org/
```

Hier laden Sie einfach den Download unter dem Link **Windows .exe** in der 32bit- oder 64bit-Version herunter und installieren die Anwendung.

Nach der Installation können Sie das Tool direkt starten. Hier bietet sich auch direkt die Schaltfläche **Neue Datenbank** zum Erstellen einer neuen Datenbank an. Wir wollen aber nicht tiefer in den Umgang mit diesem Tool einsteigen, sondern dieses nur kurz vorstellen – also Notlösung, wenn Sie einmal in den Entwurf einer Datenbank eingreifen müssen. Wir wollen aber dennoch kurz einmal einen Blick in eine echte SQLite-Datenbank werfen. Eine solche findet sich auch recht schnell, und zwar unter dem folgenden Link:

```
http://www.sqlitetutorial.net/sqlite-sample-database/
```

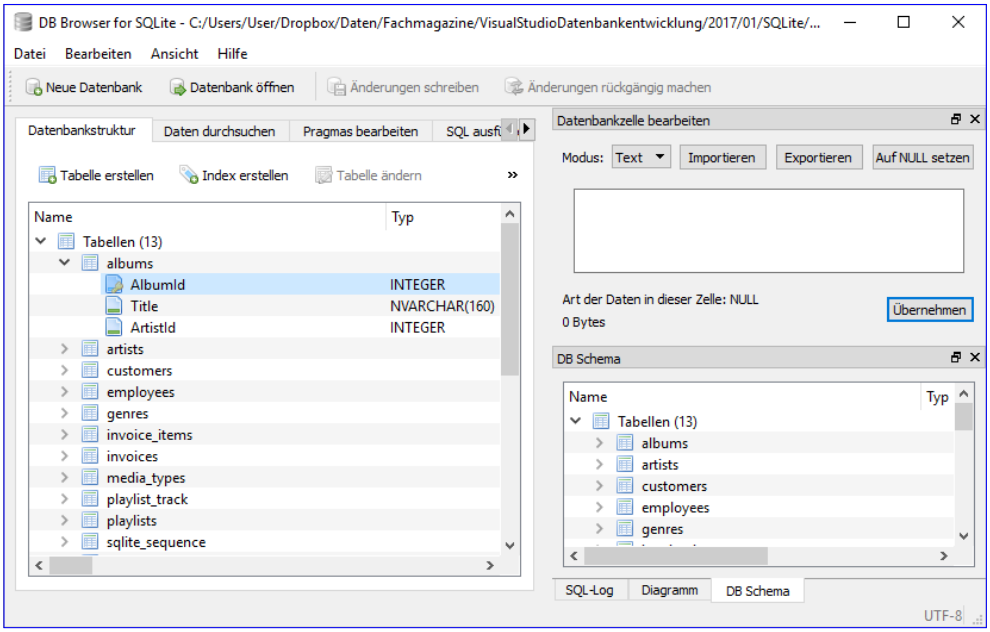


Bild 1: Fenster des SQLite-Tools **DB Browser for SQLite**

Nach dem Download finden Sie eine Datei namens **chinook.db** im Zielverzeichnis. Öffnen Sie diese nun mit dem **Tool DB Browser for SQLite**, füllt sich dieses recht schnell mit bekannt aussehenden Elementen wie Tabellen, Feldern und Felddatentypen (siehe Bild 1).

Vom SQL Server zu SQLite

Sie können diese Tabelle als Ausgangspunkt für eigene Experimente nutzen, aber wir wollen gern mit der bereits in weiteren Beispielen verwendeten Datenbank Bestellverwaltung weiter arbeiten. Wie aber machen wir aus einer SQL Server-Datenbank eine SQLite-Datenbank? Auch hierfür gibt es ein Tool, dass Sie unter diesem Link finden:

```
https://www.codeproject.com/articles/26932/convert-sql-server-db-to-sqlite-db
```

Hier müssen Sie sich allerdings anmelden, um dann die gewünschte Version, hier wohl die Binary, herunterzuladen. Sie erhalten hier eine **.zip**-Datei mit einem Verzeichnis namens **SqlConverter_v1_20**, dass Sie einfach auf Ihre Festplatte kopieren. Starten Sie die Anwendung, die als **Converter.exe** in diesem Verzeichnis steckt, erscheint das Fenster aus Bild 2.

Geben Sie hier den Servernamen des SQL Servers ein und wählen Sie dann die Quelldatenbank aus. Legen Sie außerdem den Namen der zu erstellenden **.db**-Datei fest, zum Beispiel **Bestellverwaltung_SQLite.db**. Ein Mausklick auf die Schaltfläche **Start the conversion** startet den Vorgang, bei dem Sie zunächst mit dem Dialog aus Bild 3 die zu konvertierenden Tabellen festlegen.

Danach dürfte eine Datei namens **Bestellverwal-**

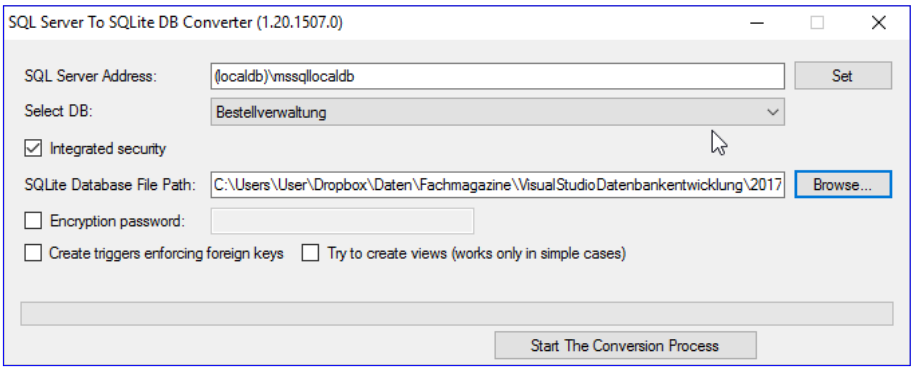


Bild 2: Tool zum Konvertieren von SQL Server-Datenbanken nach SQLite

tung_SQLite.db im gewählten Zielverzeichnis gelandet sein. Öffnen Sie diese Datei nun mit dem **DB Browser for SQLite**, können Sie schnell prüfen, ob alle Tabellen und Felder wie gewünscht in der Zieldatenbank gelandet sind (siehe Bild 4).

Vom Artikel zum Produkt

Wir wollen in diesem Rahmen auch gleich eine kleine Änderung an der Beispieldatenbank vornehmen: Im Gegensatz zu den englischen Tabellenbezeichnungen, die sich im Plural gegenüber dem Singular durch ein zusätzliches, angehängtes s unterscheiden (**customer** – **customers**), ist es im deutschen Sprachgebrauch nicht so einfach. Dummerweise schleichen sich hier mitunter Bezeichnungen wie **Artikel** ein, die im Singular wie im Plural gleich lauten. Um hier keine Probleme zu bekommen, wollen wir die Bezeichnung der SQL Server-Tabelle **Artikel** zuvor in **Produkte** ändern. Außerdem fallen entsprechende Änderungen in der Tabelle **Bestelldetails** an, das ja die Tabelle **Artikel**/**Produkte** per Fremdschlüsselfeld referenziert.

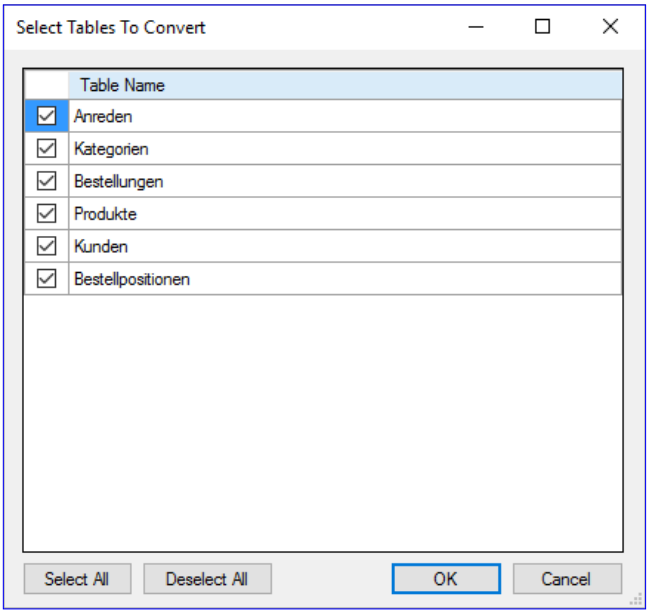


Bild 3: Auswahl der zu konvertierenden Tabellen

SQLite in .NET-Projekten

Wie greifen wir nun von einer .NET-Anwendung auf die Daten einer SQLite-Datenbank zu? Da wir den Fokus aktuell auf den Zugriff per Entity Framework setzen, ist dies unser Ansatz für den Zugriff auf die soeben umgewandelte Datenbank.

SQLite mit Entity Framework

Wir wollen also zunächst ein Entity Data Model auf Basis der SQLite-Datenbank erstellen. Dazu erstellen wir ein neues Projekt auf Basis der Vorlage **Visual C#|Windows|WPF**-

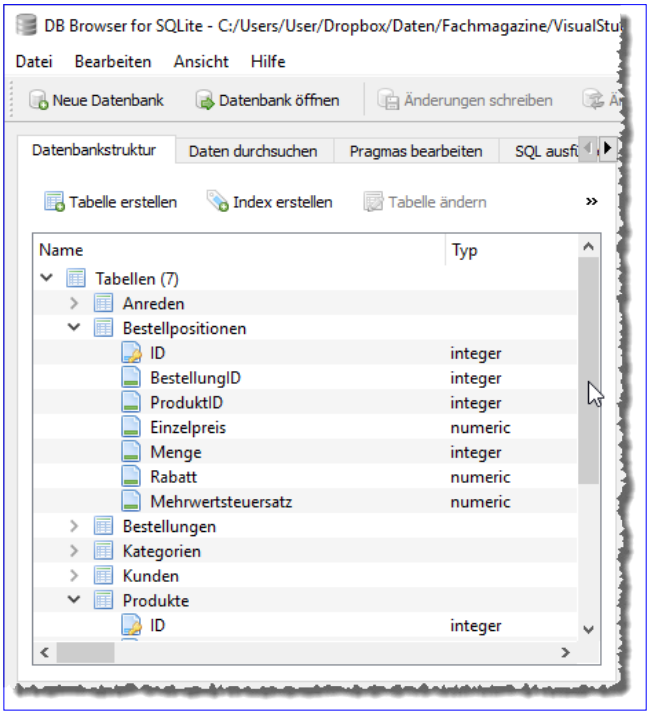


Bild 4: Anzeige der konvertierten Datei im **DB Browser for SQLite**

Anwendung namens **Bestellverwaltung_SQLite**.

Anschließend würden wir bei Verwendung des SQL Servers einfach ein neues Element des Typs **ADO.NET Entity Data Model** hinzufügen. Das geht nun nicht direkt, sondern wir müssen einen kleinen Umweg gehen. Wir müssen nämlich zuerst ein entsprechendes NuGet-Paket hinzufügen.

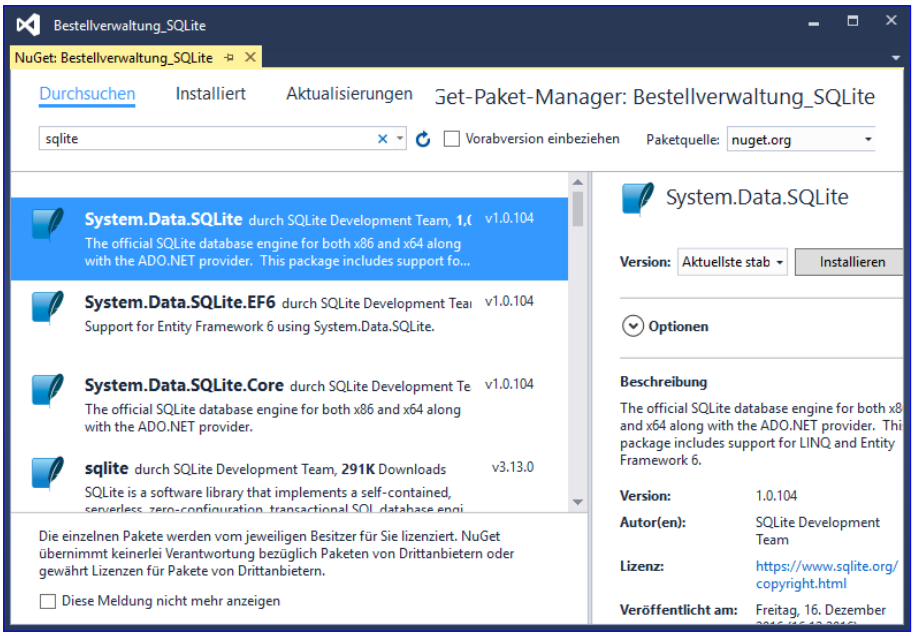


Bild 5: Hinzufügen des NuGet-Pakets für SQLite

SQLite-Erweiterung zum Projekt hinzufügen

Dazu wählen Sie den Menüeintrag **Projekt|Nuget-Pakete verwalten...** aus und wechseln im nun erscheinenden Dialog auf die Seite **Durchsuchen**.

Hier geben Sie als Suchbegriff **SQLite** ein und finden dann recht schnell das Paket **System.Data.SQLite**, welches alles enthält, was wir benötigen (siehe Bild 5). Wählen Sie diesen Eintrag aus und klicken Sie auf die Schaltfläche **Installieren**.

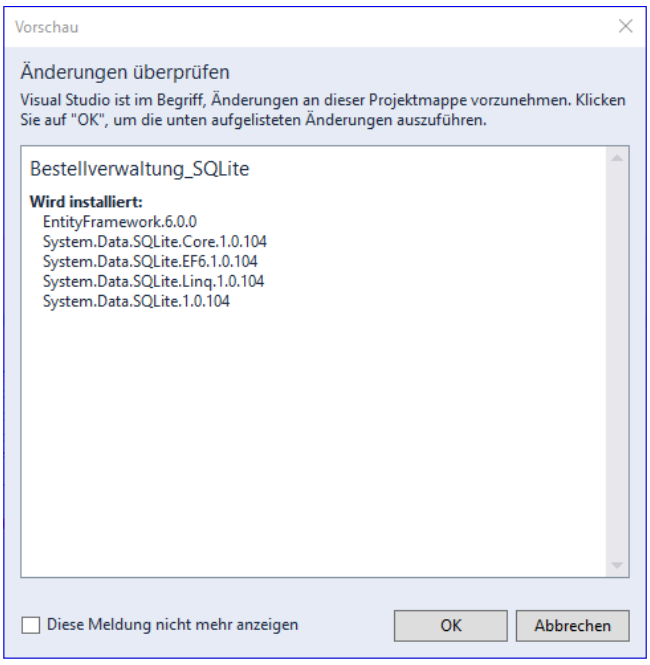


Bild 6: Bestätigen der Änderungen

Nach wenigen Augenblicken und dem Bestätigen der Meldung aus Bild 6 finden Sie im Projektmappen-Explorer einige neue Einträge unter Verweise, nämlich **System.Data.SQLite**, **System.Data.SQLite.EF6** und **System.Data.SQLite.Linq**. Damit haben Sie sich nicht nur Entity Framework-, sondern gleich auch noch Linq-Unterstützung ins Projekt geholt.

Entity Data Model hinzufügen

Damit können wir nun ein Entity Data Model auf Basis der SQLite-Datenbank erstellen. Öffnen Sie den Dialog **Neues Element hinzufügen** (zum Beispiel mit der Tastenkombination **Strg + Umschalt + A**) und wählen Sie dort den Eintrag **ADO.NET Entity Data Model** aus. Im Dialog **Neues Element hinzufügen** geben Sie als Namen für das Entity Data Model den Wert **BestellverwaltungEntities** an. Klicken Sie dann auf Hinzufügen, erscheint der **Assistent für Entity Data Model**. Hier wählen Sie den Eintrag **EF Designer aus Datenbank** aus.

Nun folgt der Schritt **Wählen Sie Ihr Datenverbindung aus**. Hier klicken Sie auf **Neue Verbindung** und öffnen damit den

Anwendung von SQL Server zu SQLite wechseln

Die aktuelle Beispieldatenbank zu diesem Magazin namens Bestellverwaltung verwendet bisher eine SQL Server-Datenbank gleichen Namens als Datenbank. Nun wollen wir für kommende Beispiele auf SQLite als Datenbanksystem wechsen, da es schlanker und für Beispiele perfekt geeignet ist. In diesem Zuge macht es Sinn, einmal zu betrachten, wie Sie das Entity Data Model für Ihre Datenbankanwendung wechseln und die Anwendung anschließend wie zuvor weiter verwenden können.

Die Ausgangsposition ist also die Beispieldatenbank, die wir in Ausgabe 6/2016 zuletzt aktualisiert haben und die per Ribbon die Bearbeitung von Kunden ermöglicht. Wie können wir für diese Datenbankanwendung, die auf einem Entity Data Model auf Bais der SQL Server-Datenbank **Bestellverwaltung** basiert, ein Entity Data Model auf Basis einer anderen Datenbank eines anderen Datenbanksystems erstellen?

Als Erstes fügen wir, weil wir mit SQLite arbeiten wollen, mit dem NuGet-Paket-Manager das Paket **System.Data.SQLite** zum Projekt hinzu (Vorgehensweise hierzu und Details zu den folgenden Schritten siehe Artikel **SQLite als Backend**).

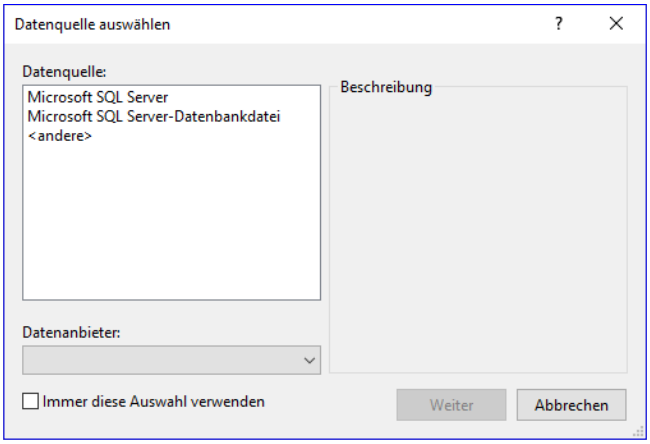


Bild 1: Der SQLite-Provider fehlt.

Prüfen, ob SQLite-Provider da ist

Die nächsten Schritte probieren Sie zunächst einmal aus, um zu ermitteln, ob der SQLite-Provider verfügbar ist – falls ja, entfernen wir vor dem tatsächlichen Hinzufügen des Entity Data Models auf Basis der SQLite-Datenbank anschließend erst das vorhandene Entity Data Model. Zum Testen fügen wir über den Dialog **Neues Element hinzufügen** (Menüeintrag **ProjektlNeues Element hinzufügen**) ein neues **Entity Data Model** namens **BestellverwaltungEntities** hinzu. Beim Schritt **Wählen Sie Ihre Datenverbindung aus** klicken Sie auf **Neue Verbindung...** und finden dann den Dialog aus Bild 1 vor, der den erwarteten SQLite-Eintrag nun eventuell nicht anzeigt.

In diesem Fall gehen Sie wie folgt vor:

- Öffnen Sie die Datei **App.config**.
- Suchen Sie den folgenden Eintrag und entfernen Sie den kursiv gedruckten Teil:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  ...
  <system.data>
    <DbProviderFactories>
```

```
      <remove invariant="System.Data.SQLite.EF6" />
      <add name="SQLite Data Provider (Entity Framework 6)" ...
          type="System.Data.SQLite.EF6.SQLiteProviderFactory, System.Data.SQLite.EF6" />
      <remove invariant="System.Data.SQLite" />
      <add name="SQLite Data Provider" ... type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
    </DbProviderFactories>
  </system.data>
</configuration>
```

- Kompilieren Sie die Anwendung.
- Versuchen Sie erneut, das Entity Data Model hinzuzufügen, was diesmal gelingen sollte.

Gelingt es immer noch nicht, haben Sie möglicherweise noch nicht alle nötigen Komponenten installiert – siehe Artikel **SQLite als Backend**. Ob dies der Fall ist, können Sie auch testen, indem Sie den Menüeintrag **Extras|Mit Datenbank verbinden...** aufrufen und im Dialog **Verbindung hinzufügen auf Ändern...** klicken. Sollte dann im nächsten Dialog wie in Bild 2 der Eintrag **System.Data.SQLite Database File** auftauchen, ist der Provider jedenfalls vorhanden.

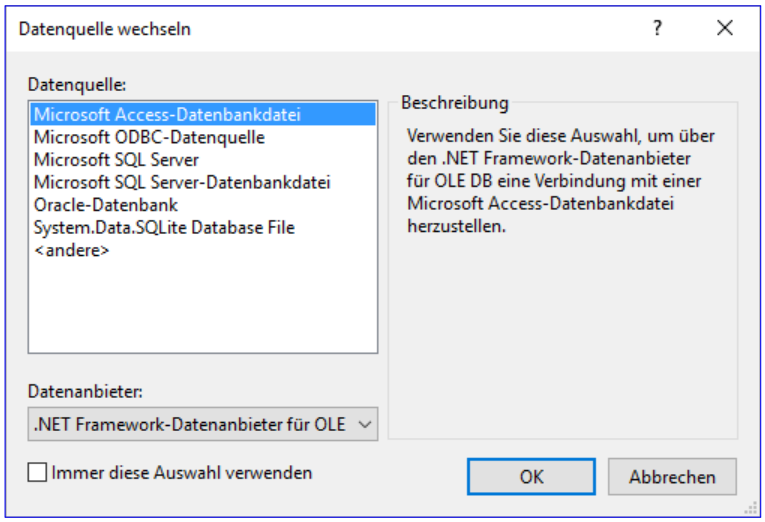


Bild 2: SQLite ist grundsätzlich verfügbar.

SQL Server-EDM löschen

Danach gehen wir recht rustikal vor und löschen den alten Eintrag **BestellverwaltungModel.edmx** einfach aus dem Projektmappen-Explorer, indem wir den Zweig **BestellverwaltungModel.edmx** entfernen. Danach löschen wir noch den folgenden Teil aus der Datei **App.config**:

```
<connectionStrings>
  <add name="BestellverwaltungEntities" connectionString="metadata=res://*/BestellverwaltungModel.csdl|res://*/Be-
stellverwaltungModel.ssdl|res://*/BestellverwaltungModel.msl;provider=System.Data.SqlClient;provider connection
string=&quot;data source=(localdb)\MSSQLLocalDB;initial catalog=Bestellverwaltung;integrated security=True;MultipleActive
ResultSets=True;App=EntityFramework&quot;" providerName="System.Data.EntityClient" />
</connectionStrings>
```

SQLite-EDM hinzufügen

Erst dann legen wir tatsächlich das Entity Data Model auf Basis der SQLite-Datenbank **Bestellverwaltung.db** an. Dieser Vorgang resultiert in der Anzeige des Diagramms der Datei **BestellverwaltungEntities.edmx**. Hier ändern Sie noch die Bezeichnungen der Entitäten, sodass diese im Singular statt im Plural angegeben sind.

Datentypen anpassen

Beim nächsten Versuch, die Anwendung zu kompilieren, erhalten wir Fehlermeldungen mit dem Text **Konvertierung von "long" in "int" nicht möglich**. In der SQL Server-Datenbank, aus der wir die SQLite-Datenbank migriert haben, waren die Primärschlüsselfelder und somit auch die Fremschlüsselfelder als **int** deklariert. Bei der Migration hat das entsprechende Tool daraus den Datentyp **long** gemacht. Dies lässt sich bei dem verwendeten Tool auch nirgends einstellen, sodass wir nun zwei Möglichkeiten haben:

- Wir ändern den Datentyp in der SQLite-Datenbank für alle Primär- und Fremdschlüsselfelder von **long** zurück in **int** und erstellen das Entity Data Model erneut.

- Oder wir passen die Datentypen der aktuell noch wenigen Klassen in unserer Anwendung an. Bei sehr großen Datenmengen kann es ohnehin sinnvoll sein, direkt mit dem Datentyp **long** für Primär- und Fremdschlüsselfelder zu arbeiten – zumal dafür kein zusätzlicher Platz reserviert werden muss.

Wir entscheiden uns an dieser Stelle für die zweite Variante und ändern einige Deklarationen (im Falle der Beispieldatenbank beim Parameter von vier verschiedenen Funktionen). Aus

```
public Kundendetails(int kundeID = 0) {
```

wird dann beispielsweise:

```
public Kundendetails(long kundeID = 0) {
```

Nun wird die Anwendung kompiliert, aber beim Versuch, auf die Daten zuzugreifen, erscheint für die Zeile

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

die folgende Fehlermeldung:

```
Der Anbietername für die Anbieterfactory vom Typ 'System.Data.SQLite.SQLiteFactory' kann nicht ermittelt werden. Stellen
Sie sicher, dass der Anbieter ADO.NET installiert oder in der config-Datei der Anwendung registriert ist.
```

Und nun kommt das Interessante: Wir müssen an dieser Stelle die folgenden Zeilen wieder zur Datei **App.config** hinzufügen:

```
<remove invariant="System.Data.SQLite" />
<add name="SQLite Data Provider" nvariant="System.Data.SQLite" description=".NET Framework Data Provider for SQLite"
  type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
```

Anschließend läuft unsere kleine Beispielanwendung genau so, wie Sie es auch bereits mit dem Entity Data Model auf Basis des SQL Servers getan hat.

Bestellverwaltung planen

Unsere Bestellverwaltung wächst weiter. Wir haben bereits eine Übersicht der Kunden und eine Detailansicht, mit der Sie Kunden anzeigen oder neue Kunden anlegen können. Nun sollen noch Übersichten und Detailansichten für Produkte, Kategorien und weitere Elemente hinzukommen. Hier sind nun einige grundsätzliche Entscheidungen bezüglich des Aufbaus der Anwendung zu treffen – zum Beispiel, wie das Ribbon erweitert werden soll oder wie nach dem Ändern oder Anlegen neuer Datensätze verfahren werden soll. Außerdem wollen wir bei der Verwaltung der Kunden noch ein paar Anpassungen durchführen, bevor wir die übrigen Ansichten nach einem ähnlichen Schema aufbauen.

Beispieldaten

Als Beispielprojekt verwenden wir weiterhin das Projekt **Bestellverwaltung_SQLite**, das Sie im Download etwa des Beitrags **EDM: 1:n-Beziehungen mit DataGrid** finden.

Aktueller Stand

Der Stand vor dieser Ausgabe enthält ein kleines Ribbon mit drei Schaltflächen, welche die Befehle zum Öffnen der Kundenübersicht, zum Öffnen eines Detailformulars zum Anlegen eines neuen Kunden und zum Löschen des aktuell in der Kundenübersicht markierten Datensatzes anzeigt. An der Schaltfläche **Kunde löschen** haben wir gezeigt, wie Sie in Abhängigkeit des im **Frame**-Objekt angezeigten **Page**-Elements Steuerelemente im Ribbon aktivieren und deaktivieren können.

Die Schaltfläche sollte aktiviert werden, wenn die Kundenübersicht im **Frame**-Element angezeigt wurde und deaktiviert werden, wenn ein anderes **Page**-Element im **Frame**-Element zu sehen war – oder gar keine.

Einen bestehenden Kunden konnten Sie anzeigen, indem Sie erst die Kundenübersicht geöffnet und dann doppelt auf einen der Einträge geklickt haben. Das nun erscheinende Detailformular enthielt auch zwei Schaltflächen namens **Speichern** und **Abbrechen** (siehe Bild 1). Die **Speichern**-Schaltfläche hat die Daten gespeichert und die **Page** geschlossen, die **Abbrechen**-Schaltfläche hat die **Page** geschlossen, ohne zu speichern. In beiden

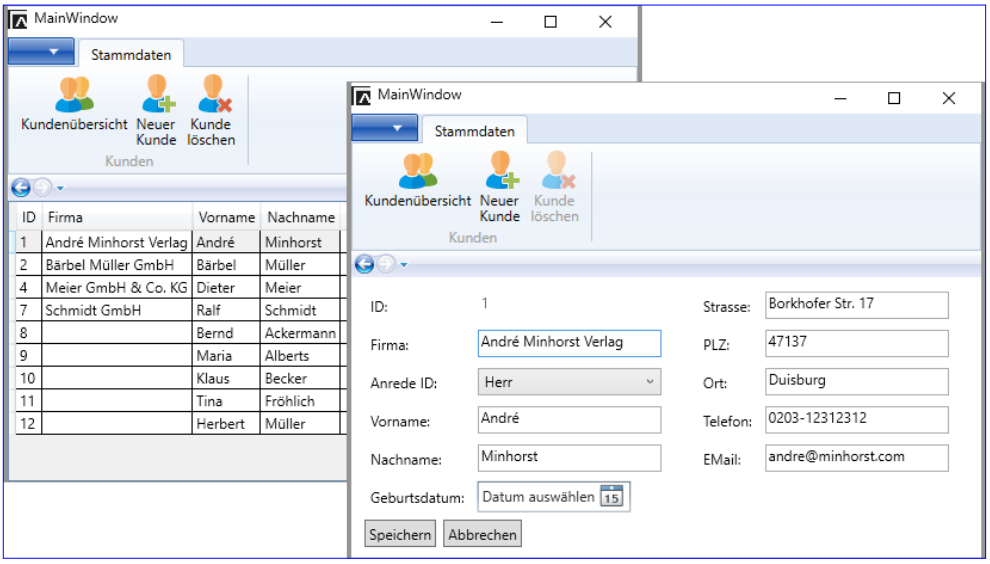


Bild 1: Übersicht der Kategorien

Fällen wurde danach wieder die Seite mit der Kundenübersicht aufgerufen, wobei der soeben bearbeitete Kunde markiert wurde.

Neue Elemente

In dieser Ausgabe wollen wir der Anwendung weitere Übersichten und Detailansichten hinzufügen. Zunächst einmal wäre da die Produktübersicht mit dem jeweiligen Detailformular zur Anzeige eines Produkts. Diese beiden **Page**-Elemente unterscheiden sich nicht von denen zur Anzeige der Kunden-Übersicht und der Kundendetails.

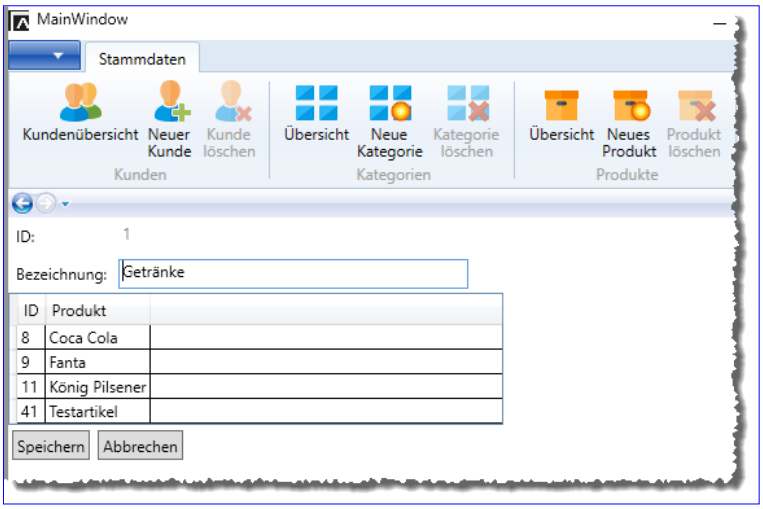


Bild 2: Kategorie-Details mit zugehörigen Produkten

Wir haben diese aber hinzugefügt, weil wir die Produkte noch in einer anderen Ansicht anzeigen möchten, und zwar in einer Kategorie-Detailansicht (mit der natürlich eine passende Kategorien-Übersicht kommt). Die Kategorie-Detailansicht soll nicht nur die Kategorie und die beiden Eigenschaften **ID** und **Bezeichnung** anzeigen, sondern auch die Produkte, die zu der jeweiligen Kategorie gehören (siehe Bild 2).

Von der Übersicht zum Detail und wieder zurück

Wir wollen nun sowohl in der Übersicht der Produkte als auch in der Liste der Produkte in der Kategorie-Detailansicht die Möglichkeit bieten, ein Produkt per Doppelklick in der Detailansicht anzuzeigen. Das ist kein Problem, wir können dabei wie schon bei der Kundenübersicht vorgehen. Was aber geschieht, wenn wir in der Produkt-Detailansicht auf die Schaltfläche **Speichern** oder **Abbrechen** klicken? Bei der Kunden-Detailansicht haben wir einfach die Kundenübersicht angezeigt und den zuletzt geöffneten Kunden markiert.

Nun haben wir aber zwei Produktübersichten (die eigentliche Übersicht und die Übersicht in den Kategorie-Details), von denen aus die Produkt-Detailansicht geöffnet worden sein könnte. Wir haben nun zum Beispiel die folgenden Möglichkeiten:

- Wir fügen in den Konstruktor des **Page**-Elements **Produktdetails.xaml** einen Parameter ein, dem wir den Namen des aufrufenden **Page**-Elements mitgeben und zeigen dieses nach dem Schließen der Produktdetails wieder an.

- Wir ändern die Funktion der beiden Schaltflächen **Speichern** und **Abbrechen**, sodass diese nicht mehr das **Page**-Element **Produktdetails.xaml** ausblenden. Die **Speichern**-Schaltfläche sollte dann nur noch das aktuell angezeigte Produkt speichern, die **Abbrechen**-Schaltfläche könnte dann entfallen. Oder man ersetzt die Abbrechen-Schaltfläche durch eine Verwerfen-Schaltfläche, welche den Zustand vor den Änderungen wieder herstellt. Hier wäre es dann sinnvoll, wenn die **Speichern**-Schaltfläche nur aktiviert wird, wenn es tatsächlich Änderungen gegeben hat.

- Wir nutzen die Eigenschaften des **Frame**-Elements, das ja das Navigieren zum nächsten oder vorherigen Element erlaubt.

Grundsätzlich sollten wir hier auch noch eine Funktion einfügen, die den Benutzer beim Wechsel zu einer anderen Ansicht ohne vorheriges Speichern darauf hinweist, dass eventuell vorgenommene Änderungen noch nicht gespeichert wurden.

Wir entscheiden uns für die folgende Vorgehensweise: Wir nutzen die Navigationsmöglichkeiten des **Frame**-Elements kombiniert mit je zwei Schaltflächen auf jeder Detailansicht, welche die Beschriftungen **Speichern** und **Verwerfen** enthalten.

Drückt der Benutzer eine dieser Schaltflächen, wird das zuvor angezeigte **Page**-Element angezeigt, was das aufrufende Element sein dürfte – im Falle der Produkt-Detailseite also etwa die Produkt-Übersicht oder die Kategorie-Detailseite mit der Liste der verknüpften Produkte.

Sollte der Benutzer die Navigationsschaltflächen zum Zurückspringen nutzen, ohne auf eine der Schaltflächen Speichern oder Verwerfen zu klicken, soll eine Meldung erscheinen, die den Benutzer fragt, ob die Änderungen gespeichert oder verworfen werden sollen. Die Schaltflächen sollen nur aktiviert werden, wenn Änderungen stattgefunden haben, dementsprechend soll die Meldung auch nur in diesem Fall erscheinen.

Markieren des zuletzt bearbeiteten Eintrags

Im bisherigen Stand des Projekts haben wir im Falle des Änderns oder Neuanlegens eines Kunden im **Page**-Element **Kundendetails**.xaml beim Schließen ein Ereignis ausgelöst, welches das Objekt mit dem aktuellen Kunden bereitstellt. Dieses haben wir im übergeordneten **MainWindow** implementiert. So konnten wir, wenn ein Kunde geändert oder hinzugefügt wurde, gleich die Kundenübersicht anzeigen und dort den geänderten oder hinzugefügten Kunden markieren. Dieses Verhalten wollen wir auch beibehalten.

Löschen-Schaltflächen

Auch über die **Löschen**-Schaltflächen sollten wir uns Gedanken machen. Wenn wir für jede Entität nun jeweils eine Schaltfläche zum Anzeigen der Übersicht, zum Anlegen eines neuen Datensatzes und zum Löschen des jeweils markierten Datensatzes hinzufügen, nimmt das erstens Platz im Ribbon weg. Zweitens müssen wir zum Aktivieren und Deaktivieren der Löschen-Schaltflächen mehr und mehr Code zum Code behind-Modul des Fensters **MainWindow.xaml** hinzufügen. Wir entscheiden an dieser Stelle, die Schaltflächen direkt in die jeweiligen **Page**-Elemente zu verlagern, sodass sie direkt an Ort und Stelle aktiviert und deaktiviert werden können.

Hinzufügen-Schaltflächen

Und letztlich nehmen wir auch etwas Komplexität aus dem Code, wenn wir auch eine Schaltfläche zum Hinzufügen eines neuen Elements direkt auf der Page mit der Übersicht anlegen. Auf diese Weise können die Detailansichten der jeweiligen Entitäten zum Hinzufügen auch nur von der entsprechenden Übersicht aus geöffnet werden, was dazu führt, dass wir nach dem Speichern oder Verwerfen der Änderungen am neuen Datensatz auf jeden Fall wieder zu der aufrufenden Übersicht zurückkehren. Gleichermaßen wollen wir aber auch die Möglichkeit bieten, ohne Umweg über die Übersicht einen neuen Datensatz der jeweiligen Entität anzulegen und behalten die entsprechenden Einträge im Ribbon bei.

Zusammenhänge zwischen XAML, C# und den verschiedenen Ansichten und Ereignissen

Auch wenn wir erst wenige Bereiche wie die Kundenübersicht und die Kundendetailansicht zu unserer Lösung hinzugefügt haben, möchten wir einen kleinen Zwischenstopp machen und uns den aktuellen Stand inklusive der oben erwähnten Änderungen ansehen. Dazu haben wir ein paar Übersichten erstellt, welche die einzelnen Abläufe in der Anwendung vorstellen. Diese zeigen wir in den nächsten Abschnitten. Auf Basis der hier beschriebenen Abläufe wollen wir auch die Seiten für weitere Elemente wie die Kategorien und die Produkte aufbauen.

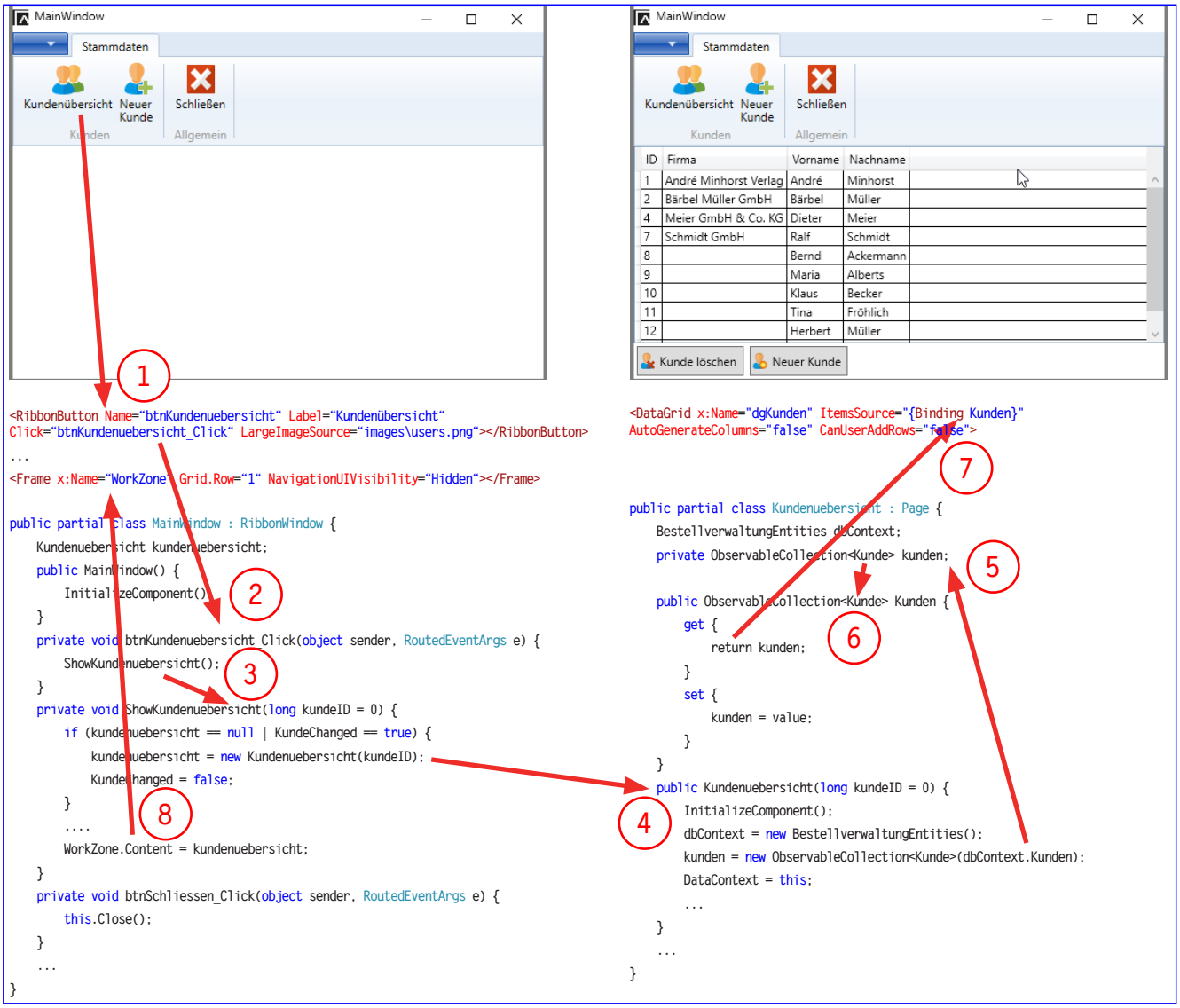


Bild 3: Vorgang beim Öffnen der Übersicht

Anzeigen der Übersicht

Der erste Schritt beim Arbeiten mit den Kunden dürfte die Anzeige der Kundenübersicht sein (siehe Bild 3). Die Anwendung startet mit der Anzeige des nackten Ribbons, das **Frame**-Element im unteren Bereich zeigt noch keine Elemente an. Wenn der Benutzer nun auf die Schaltfläche **cmdKundenuebersicht** klickt, löst er das Ereignis **cmdKundenuebersicht_Click** aus **(1)**.

Dieses haben wir in der Code behind-Klasse mit der Ereignismethode **btnKundenuebersicht_Click** implementiert **(2)**. Die Methode ruft eine weitere Methode namens **ShowKundenuebersicht** auf **(3)**. Dies hat den Hintergrund, dass wir diese Methode noch von anderen Stellen aus aufrufen wollen.

ShowKundenuebersicht prüft hier, ob **kundenuebersicht** bereits eine Kundenübersicht enthält – beispielsweise, weil wir diese zuvor schon einmal angezeigt haben. Das ist hier nicht der Fall, also erstellen wir eine neue Kundenübersicht.

Dadurch wird die Konstruktormethode **Kundenuebersicht** der gleichnamigen Klasse aufgerufen (4). Diese erstellt einen neuen DBContext und füllt die Auflistung **kunden** mit den **Kunde**-Entitäten des Entity Data Models. Außerdem weist sie die **Kunden**-Liste dann der Eigenschaft **this.DataContext** zu, wodurch diese zur Datenquelle für das **Page**-Objekt wird. Durch das Füllen des **ObservableCollection**-Objekts **kunden (5)** wird diese Liste über die öffentliche Eigenschaft **Kunden (6)** auch von außerhalb der Klasse zugreifbar.

So kann nun auch das **DataGrid**-Element der Page **Kundenuebersicht** über das Attribut **ItemsSource** und die Eigenschaft **Kunden** auf die in **kunden** gespeicherte **ObservableCollection** zugreifen **(7)**. Nachdem das **Page**-Element **Kundenuebersicht** nun initialisiert und mit den gewünschten Daten gefüllt ist, kann diese als Inhalt des **Frame**-Elements des Hauptfensters **MainWindow** angegeben werden **(8)**.

Anlegen eines neuen Kunden per Ribbon-Befehl

Das Anlegen eines neuen Kunden wird in Bild 4 schematisch dargestellt.

Wenn der Benutzer auf die Schaltfläche **btnNeuerKunde** im Ribbon klickt, löst er damit die für diese Schaltfläche definierte Ereignismethode **btnNeuerKunde_Click** aus **(1)**. Diese Ereignismethode ist im Code behind-Modul **MainWindow.xaml.cs** definiert und ruft die Methode **NeuerKunde** auf **(2)**. Die Methode erstellt im ersten Schritt eine neue Instanz der Klasse **Kundendetails**. Damit löst sie die Konstruktormethode **Kundendetails** aus, der in diesem Fall kein Parameter übergeben wird, weshalb die Variable **kundeID** den Wert **0** annimmt **(3)**. Die Methode erstellt einen neuen DBContext und weist die aktuelle Klasse als **DataContext** der **Page** zu. Außerdem prüft sie, ob **kundeID** einen Wert ungleich **0** enthält, was hier nicht der Fall ist.

Daher wird im **else**-Teil der Bedingung eine neue Instanz der **Kunde**-Klasse erstellt und in der Variablen **kunde** gespeichert, die dann über die Eigenschaft **Kunde** öffentlich verfügbar ist. Außerdem fügen wir dieses Objekt auch gleich mit der **Add**-Methode der Auflistung **dbContext.Kunden** hinzu **(4)**. Dies bedeutet noch nicht, dass der neue Kunde in der Datenbank gespeichert ist – dies geschieht erst, wenn wir die SaveChanges-Methode aufrufen, was weiter unten der Fall ist. Wenn der Benutzer die Änderung am neuen Kunden verwirft, wird einfach nicht die **SaveChanges**-Methode aufgerufen, wodurch die Lebensdauer des neuen **Kunde**-Objekts einfach gemeinsam mit **dbContext** und den übrigen lokalen Variablen endet.

Außerdem füllt diese Methode noch die Liste **anreden** mit den Elementen der Entitätsliste **Anreden** des Datenbankkontextes **(5)**. Diese steht dann über die Eigenschaft **Anreden** für den Zugriff von den Elementen von **Kundendetails.xaml** bereit **(6)**.

Die Klasse **Kundendetails.xaml.cs** stellt außerdem zwei Ereignisse namens **ItemChanged** und **ItemCancelled** bereit. Diese werden mit den vier Zeilen unter **(7)** deklariert und mit einem Delegaten ergänzt. Damit sind die Arbeiten zum Initialisieren der Seite **Kundendetails** erledigt und wir kehren zum Code behind-Modul **MainWindow.xaml.cs** zurück. Nachdem wir die Objektvariable **kundendetails** mit einem neuen Objekt auf Basis der Klasse **Kundendetails** gefüllt haben, können wir die beiden dort bereitgestellten Ereignisse implementieren. Dazu fügen zunächst entsprechende Ereignishandler hinzu **(8)**. Und schließlich sorgen wir dafür, dass das **Page**-Objekt **kundendetails** im **Frame**-Element **WorkZone** eingeblendet wird **(9)**.

Nun fehlen noch die Schritte, die auf der Page **Kundendetails** ausgeführt werden, um tatsächlich einen neuen Kunden anzulegen. Nach dem Eintragen der Eigenschaften des neuen Kunden in das leere Formular klickt der Benutzer dazu auf die Schaltfläche **btnSpeichern (10)**. Für diese Schaltfläche haben wir die Ereignismethode **btnSpeichern_Click** hinterlegt, die wie unter

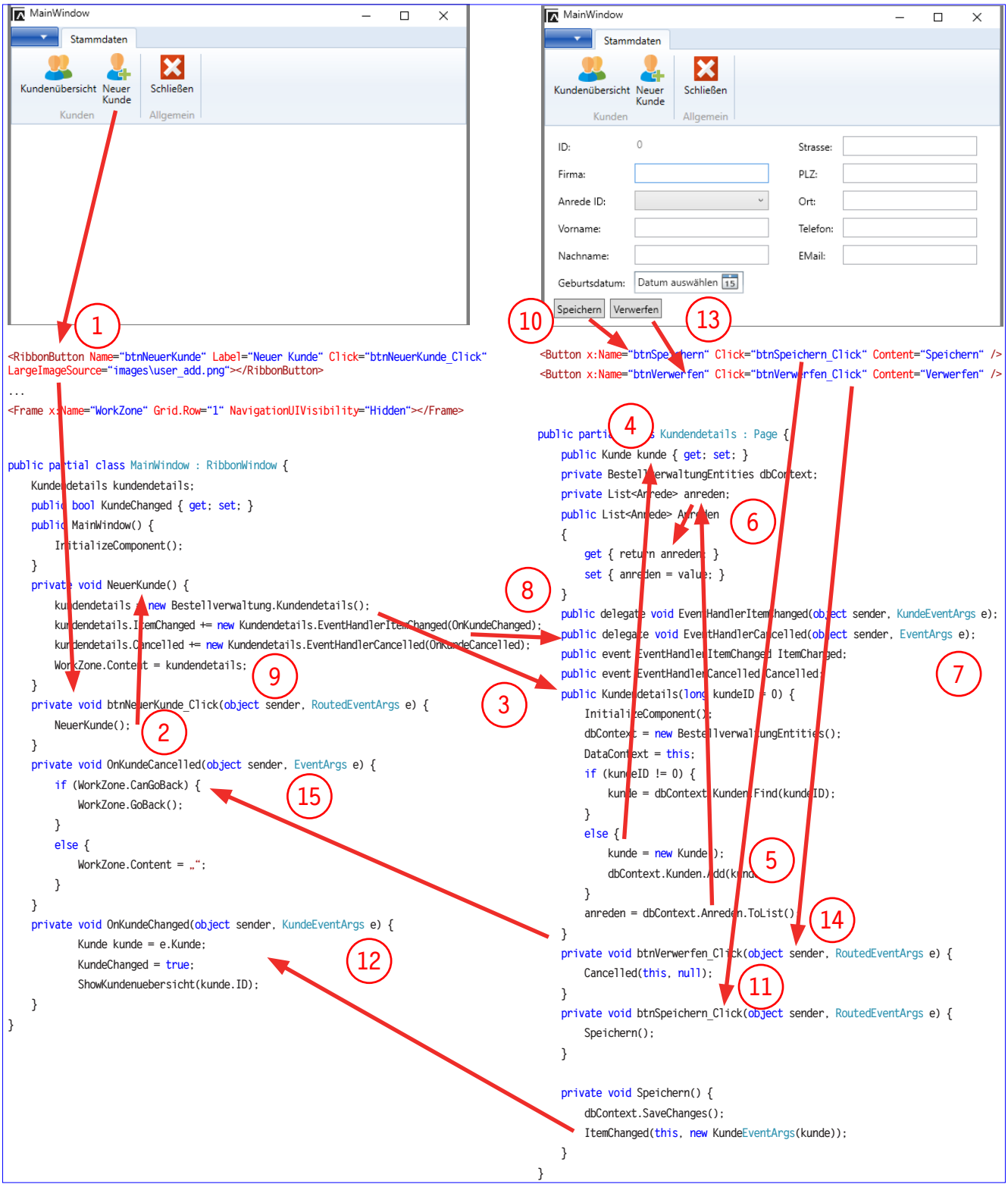


Bild 4: Vorgang beim Öffnen der Übersicht: Links **MainWindow.xaml** ohne Inhalt, recht **MainWindow.xaml** mit **Kundendetails.xaml** im Frame-Element.

EDM: 1:n-Beziehungen mit DataGrid

Unter Access haben wir 1:n-Beziehungen einfach in einem Haupt- und einem Unterformular abgebildet, wobei wir beiden einfach die Datenquellen und gebundenen Steuerelemente zugewiesen haben – den Rest hat Access automatisch erledigt. Unter C# und WPF ist das ein wenig mehr Arbeit, aber nach der Lektüre dieses Artikels haben Sie das Wissen, das für die Anzeige zweier per 1:n-Beziehung verknüpfter Tabellen in einem Fenster beziehungsweise einer Seite und einem DataGrid als Unterformular-Ersatz nötig ist.

Beispieldaten

Als Beispiel wollen wir uns die Kategorien und die damit verknüpften Produkte ansehen. Dabei sollen die beiden Felder einer Kategorie im Fenster/in der Seite selbst angezeigt werden, die dazugehörigen Produkte in einem DataGrid unter den Kategoriedaten. Wir verwenden die SQLite-Datenbank **Bestellverwaltung.db** und ein daraus abgeleitetes Entity Data Model namens **BestellverwaltungEntities.edmx** als Datenquelle. Die verwendeten Auflistungen beziehungsweise Entitäten heißen **Kategorien**, **Kategorie**, **Produkte** und **Produkt**.

Einbau in die Bestellverwaltung

In einigen weiteren Beiträgen verwenden wir die Bestellverwaltung als Beispielanwendung. Diese verwendet im Office-Stil ein Ribbon zur Auswahl der verschiedenen Bereiche und Funktionen. Die einzelnen Bereiche etwa zur Anzeige einer Kundenliste oder einer Kundendetailansicht auf Seiten (**Page**) statt auf eigenen Fenstern (**Window**) erstellt und je nach angeklickter Ribbon-Schaltfläche in einem Frame-Element eingeblendet – also etwa wie die Unterformulare in einem Unterformular-Steuerelement in Access. Wir fügen also in diesem Artikel weitere **Page**-Elemente hinzu, die wir dann nach dem Anklicken der ebenfalls noch anzulegenden Ribbon-Schaltflächen angezeigt werden sollen.

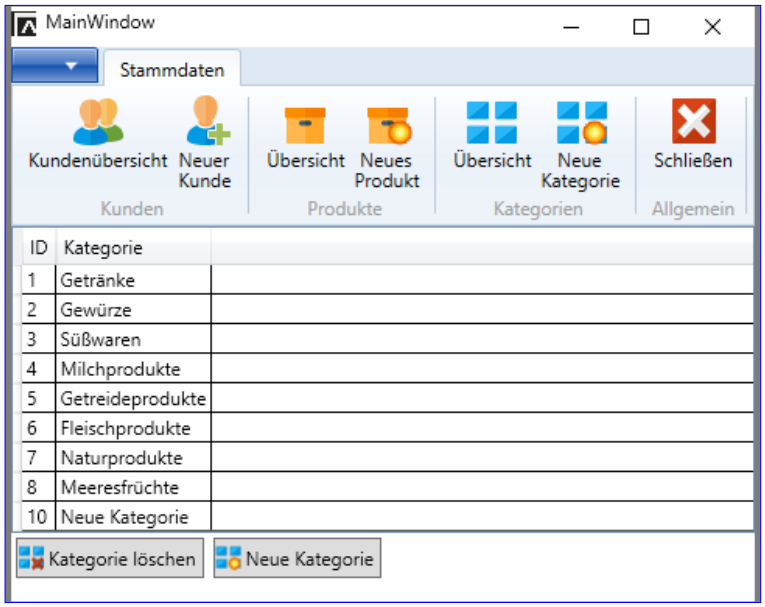


Bild 1: Übersicht der Kategorien

Ribbon-Einträge

Als Erstes legen wir die drei Ribbon-Einträge an, die Sie in Bild 1 sehen. Die Elemente für diese Ribbon-Gruppe sehen wie folgt aus:

```
<RibbonGroup Header="Kategorien">                                                    //MainWindow.xaml
    <RibbonButton Name="btnKategorieuebersicht" Label="Übersicht" Click="btnKategorieuebersicht_Click"
        LargeImageSource="images/elements4.png"></RibbonButton>
```

```
    <RibbonButton Name="btnNeueKategorie" Label="Neue Kategorie" Click="btnNeueKategorie_Click"
        LargeImageSource="images/elements4_new.png"></RibbonButton>
    <RibbonButton Name="btnKategorieLoeschen" Label="Kategorie löschen" Click="btnKategorieLoeschen_Click"
        LargeImageSource="images/elements4_delete.png"></RibbonButton>
</RibbonGroup>
```

Wir haben für jedes Element ein Bild hinterlegt, dass wir gleichzeitig zum Ordner **images** des Projekts hinzugefügt haben.

Übersichtsseite für die Kategorien

Die Kategorien werden in der Übersichtsseite **KategorieUebersicht.xaml** in einem **DataGrid**-Element aufgelistet. Das Füllen dieses **DataGrid**-Elements erfolgt genauso, wie wir es bereits im Artikel **EDM: Kunden verwalten mit Ribbon** für die Anzeige der Kunden realisiert und im Artikel **Bestellveraltung planen** angepasst haben – nur, dass wir diesmal auf die Entitätsliste der Kategorien statt der Kunden zugreifen und nur die Felder **ID** und **Bezeichnung** anzeigen.

Detailseite für die Kategorien

Der interessante Teil folgt nun, nämlich die Detailseite für eine Kategorie. Eine Kategorie enthält zwar nur die beiden Felder **ID** und **Bezeichnung**, was sich genau so leicht abbilden lässt wie in der Detailansicht für Kunden (**Kundendetails.xaml**) – nur mit weniger Feldern. Allerdings wollen wir ja zu jeder Kategorie auch noch die Liste der Produkte anzeigen, die der jeweiligen Kategorie zugeordnet sind! Und das wollen wir wiederum mit einem **DataGrid**-Element erledigen.

Imgrunde brauchen wir also eine Kombination der bereits einmal gezeigten Detailansicht für die Kunden (diesmal für die Kategorien) und einer Übersicht für die Produkte. Der Unterschied bei der Übersichtsseite diesmal ist, dass wir nicht mehr einfach alle Elemente der Produkte-Liste anzeigen können, sondern nur noch diejenigen Elemente, die der aktuellen Kategorie zugeordnet sind. Die Definition der Seite mit den Kategoriedetails und der Produktliste startet mit ein paar **Page.Resources**-Elementen, welche globale Attribute der verschiedenen Steuerelemente festlegen und die wir hier ebenso gekürzt haben wie die Definition des Grids:

```
<Page x:Class="Bestellverwaltung.Kategoriedetails" ...Title="Kategoriedetails">                    //Kategoriedetails.xaml
    <Page.Resources>...</Page.Resources>
    <Grid>
        <Grid.ColumnDefinitions>...</Grid.ColumnDefinitions>
        <Grid.RowDefinitions>...</Grid.RowDefinitions>
```

Danach folgt das Label für die **ID** der Kategorie und die **TextBox**, die wir an das Feld **kategorie.ID** binden:

```
        <Label Content="ID:" Grid.Column="0" />
        <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Text="{Binding kategorie.ID, Mode=TwoWay,
            ValidatesOnExceptions=true}" Width="50" IsEnabled="False" BorderBrush="Transparent" />
```

Die Bezeichnung erhält ebenfalls ein Label und eine Bindung an das Feld **kategorie.Bezeichnung** – beide landen in jeweils einer Spalte, genau wie die Steuerelemente für die ID:

```
        <Label Content="Bezeichnung:" Grid.Column="0" Grid.Row="1" />
        <TextBox x:Name="txtBezeichnung" Grid.Column="1" HorizontalAlignment="Stretch" Grid.Row="1"
            Text="{Binding kategorie.Bezeichnung, Mode=TwoWay, ValidatesOnDataErrors=true}" />
```

Schließlich folgt das **DataGrid**-Element namens **dgProdukte**, das in der folgenden Zeile landet und sich über zwei Spalten erstrecken soll (**Grid.ColumnSpan="2"**). Als **ItemsSource** für das **DataGrid** legen wir das Element **Produkte** fest. Das automatische Generieren der Spalten sowie das Anzeigen einer leeren Spalte zum Hinzufügen von Elementen deaktivieren wir:

```
        <DataGrid x:Name="dgProdukte" Grid.Column="0" Grid.Row="2" Grid.ColumnSpan="2" ItemsSource="{Binding Produkte}"
                AutoGenerateColumns="false" CanUserAddRows="False">
```

Die beiden Spalten des **DataGrid**-Elements binden wir an die Felder **ID** und **Bezeichnung**:

```
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding Path=ID}" Header="ID"  />
                <DataGridTextColumn Binding="{Binding Path=Bezeichnung}" Header="Produkt" />
            </DataGrid.Columns>
```

Außerdem wollen wir, wie schon bei den übrigen Übersichten, ein Öffnen der Produktdetails per Doppelklick erlauben und fügen dazu einen **EventSetter** hinzu:

```
            <DataGrid.Resources>
                <Style TargetType="DataGridRow">
                    <EventSetter Event="MouseDoubleClick" Handler="Row_DoubleClick"/>
                </Style>
            </DataGrid.Resources>
        </DataGrid>
```

Schließlich folgen noch die beiden Schaltflächen zum Speichern und Verwerfen der aktuellen Änderungen:

```
        <StackPanel Orientation="Horizontal" Grid.Row="6" Grid.ColumnSpan="4" >
            <Button x:Name="btnSpeichern" Margin="3" Padding="3" Click="btnSpeichern_Click" Height="23"
                Content="Speichern"></Button>
            <Button x:Name="btnVerwerfen" Margin="3" Padding="3" Click="btnVerwerfen_Click" Height="23"
                Content="Verwerfen"></Button>
        </StackPanel>
    </Grid>
</Page>
```

Der Entwurf sieht in der XAML-Ansicht nun wie in Bild 2 aus. Im Entwurf haben wir nun schon einige Bindungen gesehen, die wir nun in der Code behind-Klasse bereitstellen wollen. Diese Klasse enthält zunächst eine Variable für das **DBContext**-

Element, über das wir auf das Entity Data Model zugreifen (**dbContext**). Außerden finden wir hier die Definition für die Objekte, an die wir die Elemente der XAML-Seite binden wollen. Die erste ist das **Kategorie**-Objekt, das wir in der öffentlichen Variablen **kategorie** speichern.

Die zweite ist das **ObservableCollection**-Objekt **Produkte**, deren Inhalt wie in der als privat deklarierten Variablen **produkte** speichern und über entsprechende Eigenschaften für den Zugriff von außen bereitstellen:

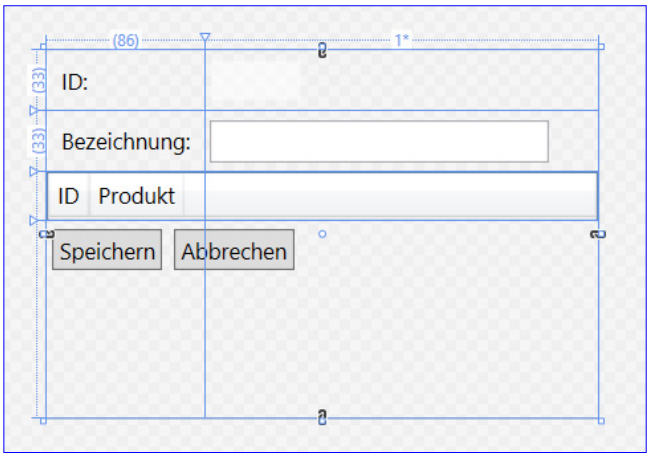


Bild 2: Entwurf der Kategoriedetail-Seite

```
public partial class Kategoriedetails : Page {
//Kategoriedetails.xaml.cs
    private BestellverwaltungEntities dbContext;
    public Kategorie kategorie { get; set; }
    private ObservableCollection<Produkt> produkte;
    public ObservableCollection<Produkt> Produkte {
        get {
            return produkte;
        }
        set {
            produkte = value;
        }
    }
    ...
}
```

Die Konstruktor-Methode **Kategoriedetails**, die beim Erstellen eines Objekts auf Basis unserer Klasse aufgerufen wird, erwartet als Parameter den Primärschlüsselwert der ID einer Kategorie. Wird dieser nicht übergeben, setzt die Methode ihn auf **0** fest – dies bedeutet, dass ein neues, leeres **Kategorie**-Element angezeigt werden soll.

Die Methode erstellt ein neues Objekt auf Basis der Klasse **BestellverwaltungEntities** und stellt den **DataContext** auf die Code behind-Klasse (**this**) ein, damit die XAML-Elemente an die per Eigenschaft zugängig gemachten Elemente der Code behind-Klasse zugreifen können. Ist die mit dem Parameter **kategorieID** übergebene **ID** der Kategorie nicht gleich **0**, sucht die **Find**-Methode in der **Kategorien**-Auflistung nach dem entsprechenden Element. Anderenfalls wird ein neues Kategorie-Element angelegt.

```
public Kategoriedetails(long kategorieID = 0) {                    //Kategoriedetails.xaml.cs
    InitializeComponent();
    dbContext = new BestellverwaltungEntities();
    DataContext = this;
```

```
    if (kategorieID != 0) {
        kategorie = dbContext.Kategorien.Find(kategorieID);
    }
    else {
        kategorie = new Kategorie();
        dbContext.Kategorien.Add(kategorie);
    }
    produkte = new ObservableCollection<Produkt>(dbContext.Produkte.Where(d => d.KategorieID == kategorie.ID));
}
```

Danach folgt der interessante Schritte: Wir füllen das DataGrid, und zwar nur mit den Entitäten, die einem bestimmten Suchkriterium entsprechen.

Dieses soll nur die Entitäten liefern, deren Feld **KategorieID** dem Wert des Feldes **ID** der angezeigten **Kategorie**-Entität entspricht. Dazu verwenden wir einen entsprechenden Linq-Ausdruck (siehe auch Artikel **LINQ to Entities: Daten abfragen**):

```
produkte = new ObservableCollection
<Produkt>(dbContext.Produkte.Where
(d=>d.KategorieID==kategorie.ID));
```

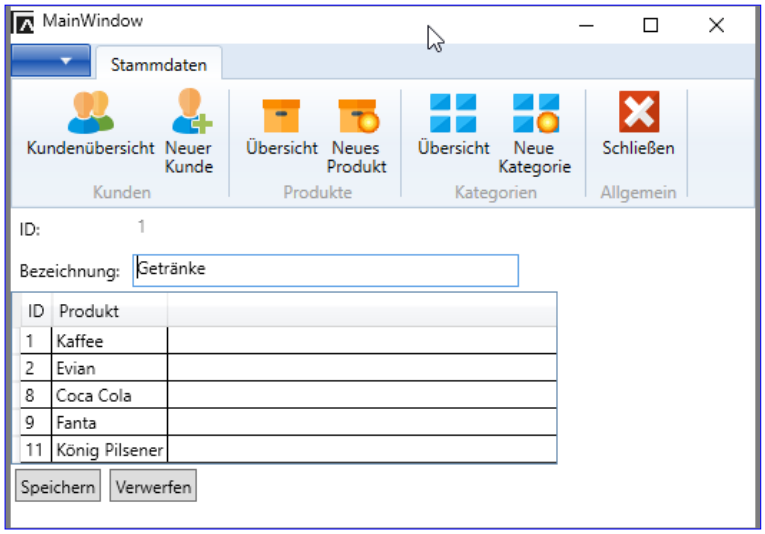


Bild 3: Anzeige von Kategorien und Produkten

Das Ergebnis sieht dann wie in Bild 3 aus.

Fokus auf Bezeichnung

Damit gleich beim Öffnen der Seite der Fokus auf das Textfeld zur Eingabe der Bezeichnung gelegt wird, geben Sie für das **Page**-Objekt den Wert **Page_Loaded** für das Attribut **Loaded** an:

```
<Page x:Class="Bestellverwaltung.Kategoriedetails" ... Loaded="Page_Loaded">                    //Kategoriedetails.xaml
```

Dies soll die folgende Ereignismethode auslösen, die lediglich den Fokus auf das Feld **txtBezeichnung** legt:

```
private void Page_Loaded(object sender, RoutedEventArgs e) {                                    //Kategoriedetails.xaml.cs
    txtBezeichnung.Focus();
}
```

Speichern der Kategorie

Ein Mausklick auf die Schaltfläche **Speichern** löst die folgende Ereignismethode aus. Diese ruft einfach eine weitere Methode namens **Speichern** auf:

```
private void btnSpeichern_Click(object sender, RoutedEventArgs e) {                    //Kategoriedetails.xaml.cs
    Speichern();
}
```

Die Methode **Speichern** speichert die aktuelle **Kategorie**-Entität mit der Methode **SaveChanges**.

Außerdem ruft sie noch das Ereignis **ItemChanged** auf und übergibt ihm eine neue Instanz der Klasse **KategorieEventArgs** mit der aktuellen Kategorie als Konstruktor-Parameter:

```
private void Speichern() {                                                             //Kategoriedetails.xaml.cs
    dbContext.SaveChanges();
    ItemChanged(this, new KategorieEventArgs(kategorie));
}
```

Die Klasse **KategorieEventArgs** implementiert die Schnittstellt **EventArgs**. Dieses bietet die Möglichkeit, die beim Auslösen des Ereignisses (hier **ItemChanged**) notwendigen Daten aufzunehmen und an die Implementierung des Ereignisses zu übergeben.

In diesem Fall soll dies lediglich das geänderte **Kategorie**-Objekt sein:

```
public class KategorieEventArgs : EventArgs {                                          //KategorieEventArgs.cs
    public KategorieEventArgs(Kategorie kategorie) {
        Kategorie = kategorie;
    }
    public Kategorie Kategorie { get; set; }
}
```

Das Ereignisse definieren wir wie folgt und legen den entsprechenden Delegaten fest:

```
public delegate void EventHandlerItemChanged(object sender, KategorieEventArgs e);    //Kategoriedetails.xaml.cs
public event EventHandlerItemChanged ItemChanged;
```

Den Fall des Abbrechens behandelt diese Methode:

```
private void btnAbbrechen_Click(object sender, RoutedEventArgs e) {                   //Kategorieudetails.xaml.cs
    Cancelled(this, null);
}
```

Bei **Cancelled** handelt es sich wieder um ein Ereignis, das samt Delegat wie folgt deklariert wird:

```
public delegate void EventHandlerCancelled(object sender, EventArgs e);               //Kategoriedetails.xaml.cs
public event EventHandlerCancelled Cancelled;
```

Die Ereignisse werden im **MainWindow**, dass ja diese Seite im **Frame**-Element anzeigt, implementiert. Die Implementierungen sorgen dort etwa dafür, dass die aktualisierten Daten direkt in der Übersicht angezeigt werden. Dies haben wir auch schon für die Seite mit den Kundendetails programmiert – die Erläuterung dazu finden Sie im Artikel **EDM: Kunden verwalten mit Ribbon** und **Bestellverwaltung planen**.

Nun in aller Kürze der Teil auf Seite des Hauptfensters – hier werden die Ereignisse **ItemChanged** und **Cancelled** in der Methode, welche die Seite **Kategoriedetails.xaml** erstellt und anzeigt, abonniert:

```
public void KategoriedetailsAnzeigen(long kategorieID = 0) {                    //MainWindow.xaml.cs
    kategoriedetails = new Kategoriedetails(kategorieID);
    kategoriedetails.ItemChanged += new Kategoriedetails.EventHandlerItemChanged(OnKategorieChanged);
    kategoriedetails.Cancelled += new Kategoriedetails.EventHandlerCancelled(OnKategorieCancelled);
    WorkZone.Content = kategoriedetails;
}
```

Beim Auslösen des Ereignisses **OnItemChanged** soll also in der Code behind-Klasse des Hauptfensters **MainWindow.xaml** die folgende Ereignismethode aufgerufen werden:

```
private void OnKategorieChanged(object sender, KategorieEventArgs e) {          //MainWindow.xaml.cs
    Kategorie kategorie = e.Kategorie;
    KategorieChanged = true;
    ShowKategorieuebersicht(kategorie.ID);
}
```

Diese entnimmt aus dem per Parameter übergebenen Objekt des Typs **KategorieEventArgs** das **Kategorie**-Objekt und speichert diese in der Variablen **kategorie**. Sie stellt die Variable **KategorieChanged** auf **True** ein und öffnet über die Methode **ShowKategorieuebersicht** die Seite **Kategorieuebersicht.xaml** im **Frame**-Element (weitere Informationen dazu am Beispiel der Kundenuebersicht siehe Artikel **Bestellverwaltung planen**).

Das Ereignis **OnCancelled** löst die Methode **OnKategorieCancelled** aus, die prüft, ob es bereits eine vorherige Seite in der Historie des **Frame**-Elements gibt und navigiert in diesem Fall zur vorherigen Seite. Anderenfalls wird das **Frame**-Element einfach geleert:

```
private void OnKategorieCancelled(object sender, EventArgs e) {                 //MainWindow.xaml.cs
    if (WorkZone.CanGoBack) {
        WorkZone.GoBack();
    }
    else {
        WorkZone.Content = "";
    }
}
```

Tipps und Tricks zu Visual Studio

Wer von Access zu Visual Studio wechselt, stößt immer wieder auf neue Herausforderungen in dieser neuen, im Vergleich zur Access-Entwicklungsumgebung viel komplexeren Welt. In der heutigen Ausgabe der Reihe Tipps und Tricks zu Visual Studio schauen wir uns einmal an, wie Sie ein Projekt umbenennen. Unter Access haben Sie dazu die Datei und gegebenenfalls noch das VBA-Projekt umbenannt. Unter Visual Studio ist eine ganze Menge mehr zu beachten: Allein, das wir nicht nur eine einzige Datei haben, sondern ganze Ordner voller Dateien und uns auch noch mit zu erstellenden .exe-Dateien und Dingen wie Namespaces herumschlagen müssen, macht die Aufgabe recht komplex ...

Komplettes Projekt umbenennen

Während der Erstellung der Beispielprojekte für dieses Magazin ist es recht häufig passiert, dass ein Projekt nicht gleich den richtigen Namen hatte. Mit dem Projektnamen legt man beim Erstellen eines Projekts natürlich auch gleich noch eine Menge weiterer Benennungen fest, zum Beispiel verschiedene Ordner und Pfade, den Namespace et cetera. Also schauen wir uns einmal an, welche Schritte nötig sind, um ein solches Projekt umzubenennen. Wir sind es zuerst einmal naiv angegangen und haben einfach den Ordner, in dem sich alle Projektdateien befinden, die **.sln**-Datei und den Ordner mit den weiteren Projektdateien in **Bestellverwaltung** umbenannt (siehe Bild 1). Den Rest werden wir schon in Visual Studio ändern können, haben wir uns gedacht.

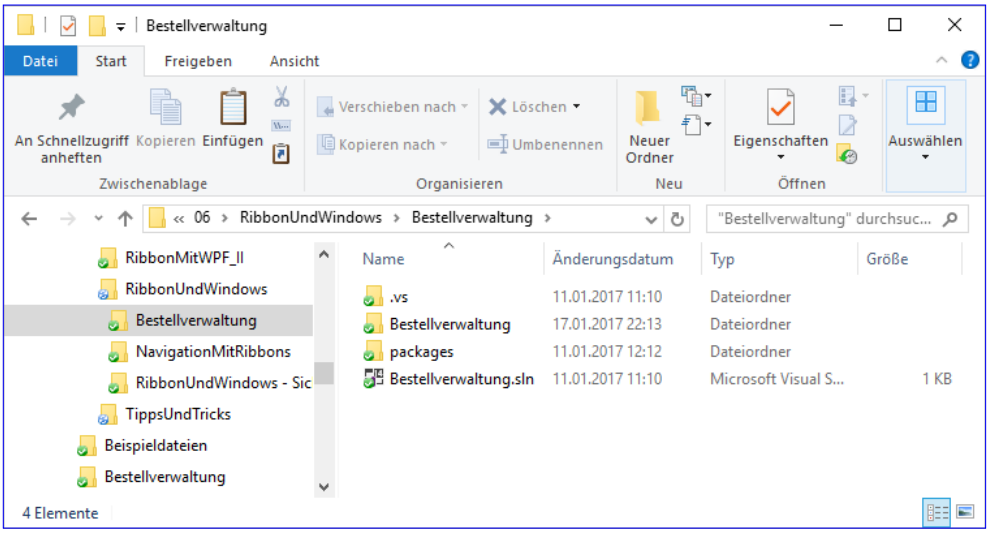


Bild 1: Projektordner mit **.sln**-Datei und Unterordnern

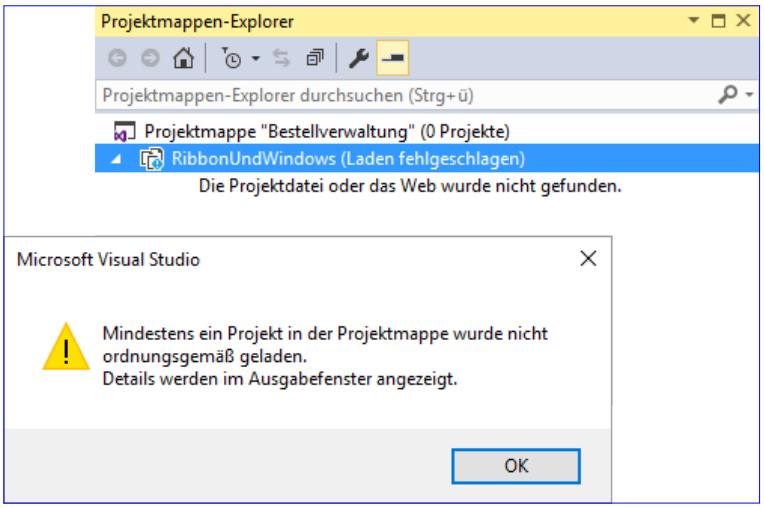


Bild 2: Probleme beim Laden eines Projekts

Aber weit gefehlt: Das Öffnen der **.sln**-Datei per Doppelklick lieferte die Meldung aus Bild 2 und anschließend einen Projektmappen-Explorer mit dem Hinweis, dass das Laden fehlgeschlagen ist. Was nun?

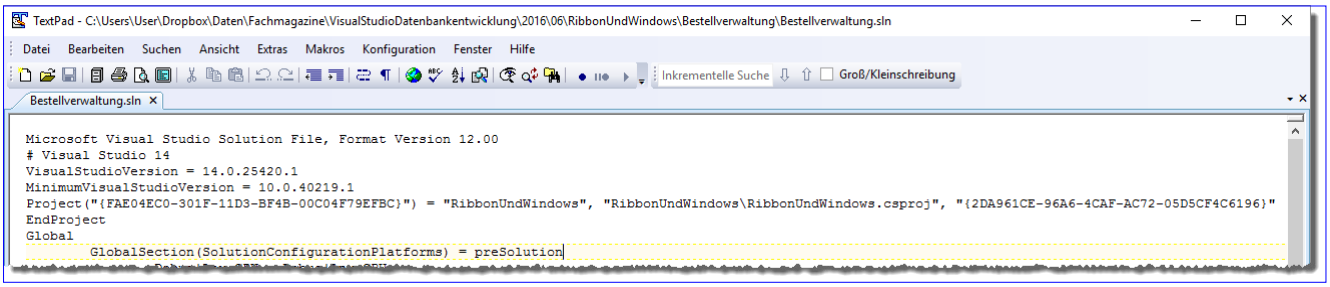


Bild 3: Falscher Pfad in der **.sln**-Datei

Wenn das Laden fehltschlägt, liegt das meist daran, dass die gewünschte Datei nicht an Ort und Stelle ist, was uns ein Blick in das Ausgabefenster dann auch bestätigte.

Da Visual Studio keine Möglichkeit bot, die **.sln**-Datei zu öffnen, haben wir diese also in einem handelsüblichen Editor angezeigt. Und hier tauchten dann auch gleich einige Erwähnungen des vorherigen Projektnamens **RibbonsUndWindows** auf (siehe Bild 3). Tauschen wir diesen also an den relevanten Stellen gegen **Bestellverwaltung** aus, sieht die betroffene Zeile wie folgt aus (auch den Projektnamen haben wir gleich angepasst):

```
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Bestellverwaltung", "Bestellverwaltung\Bestellverwaltung.csproj", "{2DA961CE-96A6-4CAF-AC72-05D5CF4C6196}"
```

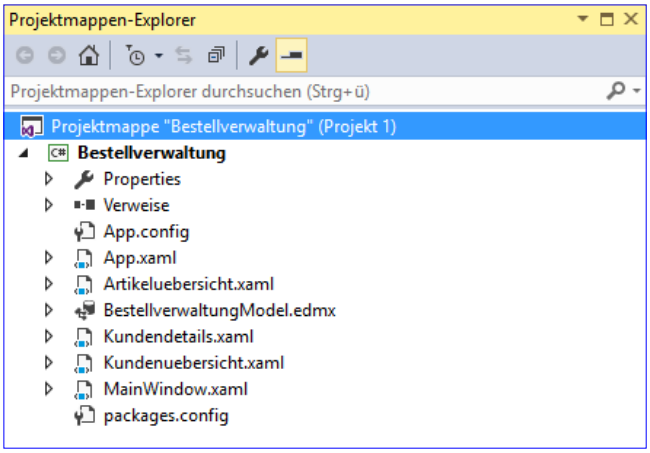


Bild 4: Erfolgreiches Laden des Projekts

Das der nächste Versuch, das Projekt über die **.sln**-Datei in Visual Studio ebenfalls fehlschlägt, hätten wir vorausahnen können: Wir haben ja in der obigen Zeile auch gleich den Namen der **.csproj**-Datei auf **Bestellverwaltung.csproj** geändert, dies aber noch nicht im Windows Explorer nachgeholt. Gegegebenenfalls müssen Sie auch noch andere Dateien umbenennen, deren Name den alten Projektnamen enthält.

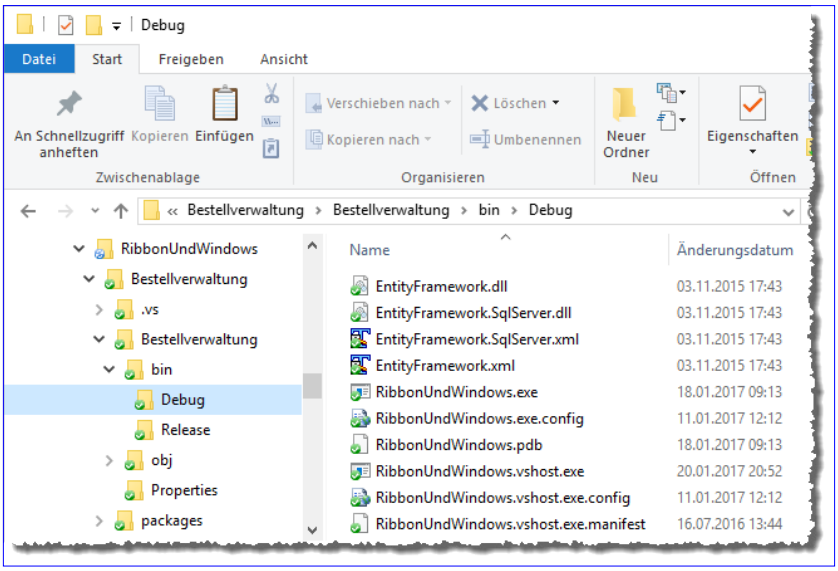


Bild 5: Die erstellten Dateien enthalten noch den alten Projektnamen

Erste wenn wir auch noch diese Datei im Windows Explorer umbenennen, lädt Visual Studio das komplette Projekt (siehe Bild 4). Dies dauert zwar ein paar Sekunden, aber vielleicht hat Visual Studio ja direkt noch ein paar Änderungen bezüglich des Projektnamens im Code vorgenommen.

Der erste Test zeigt, dass das Projekt läuft, aber ein Blick in den Code offenbart, dass

tatsächlich wohl nur die Ordner, die beiden Dateien mit der Endung **.sln** und **.csproj** sowie das Projekt selbst umbenannt wurden (siehe Bild 5). Der Namespace hat seinen Namen noch behalten.

Außerdem zeigt ein Blick in den Ordner **bin\Debug**, das auch die erstellten Dateien, also etwa die **.exe**-Datei, noch den alten Namen enthalten. Kümmern wir uns zunächst einmal um diese Dateien. Dazu brauchen Sie nur die Eigenschaften des Projekts zu öffnen, indem Sie im Projektmappen-Explorer doppelt auf den Eintrag **Properties** unterhalb des Projektnamens klicken. Hier stellen Sie direkt auf der ersten Seite **Anwendung** den Wert der Eigenschaft **Assemblyname** auf **Bestellverwaltung** ein (siehe Bild 6). Ein Test zeigt, dass die erstellten Dateien nun mit dem Namen **Bestellverwaltung** beginnen – also **Bestellverwaltung.exe** et cetera. Da liegt nun die Versuchung nahe, auch gleich die Eigenschaft **Standardnamespace** im gleichen Dialog auf **Bestellverwaltung** zu ändern. Also probieren wir dies aus und erstellen das Projekt erneut.

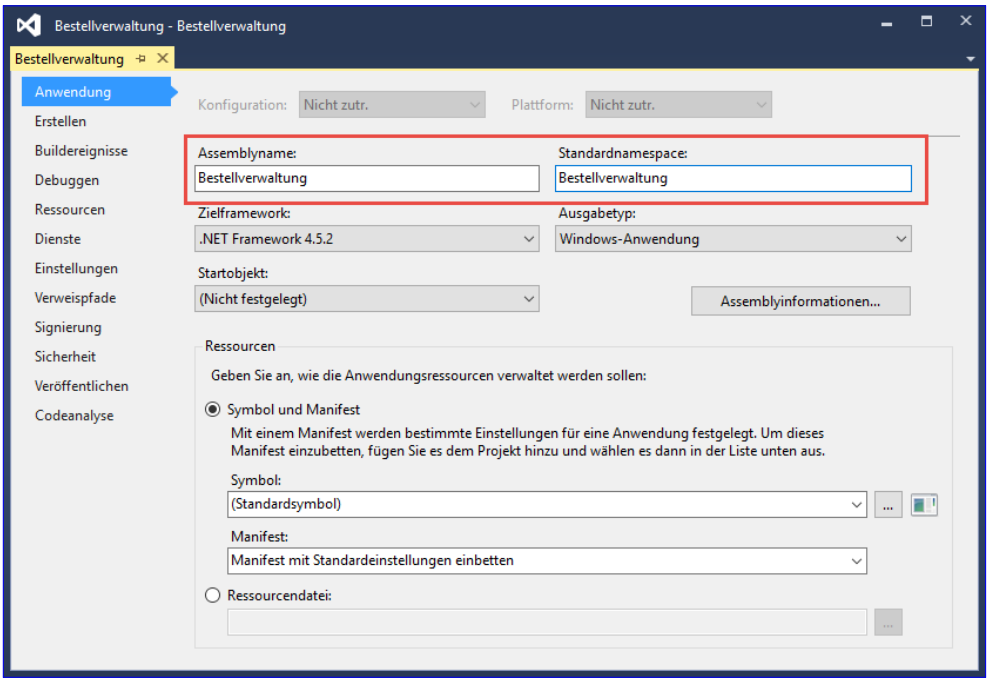


Bild 6: Ändern des Assemblynamens und des Standardnamespaces

Nun stoßen wir auf Probleme: Zwar hat diese Änderung nicht den Namespace in den durch den Benutzer erstellten geändert. Aber es gibt ja gegebenenfalls (und speziell in den Projekten, die wir in den Beispielen der aktuellen Ausgaben erstellen) auch einige Klassen und Dateien, die automatisch generiert werden. Dazu gehören die Dateien des Entity Data Models, in unserem Fall zum Beispiel die Datei **BestellverwaltungModel.Context.cs** (siehe Bild 7). Offensichtlich wird deren Neugenerierung beim Anpassen des Standardnames in den Projekteigenschaften automatisch angestoßen. Dies wird auch offensichtlich, wenn Sie dies erneut versuchen und testweise den Wert der Eigenschaft **Standardnamespace** auf **Bestellverwaltung1** ändern. Die längere

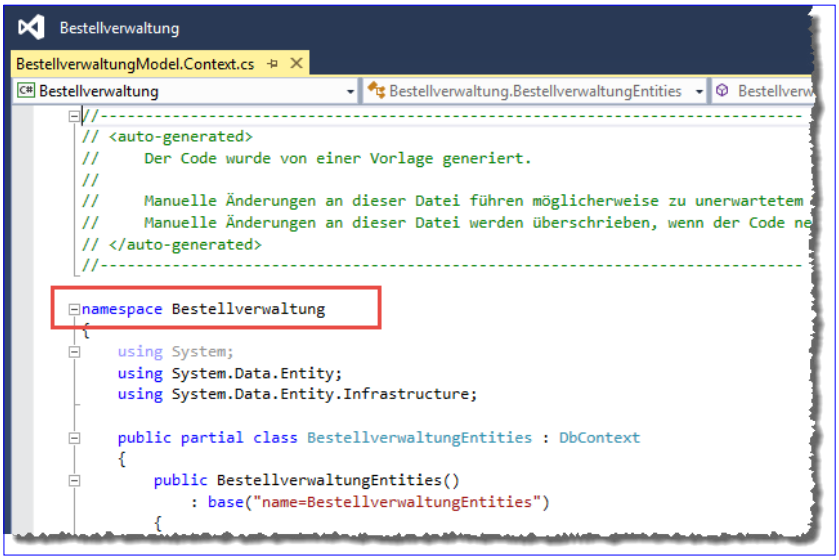


Bild 7: Einige Namespace-Bezeichnungen werden automatisch geändert.

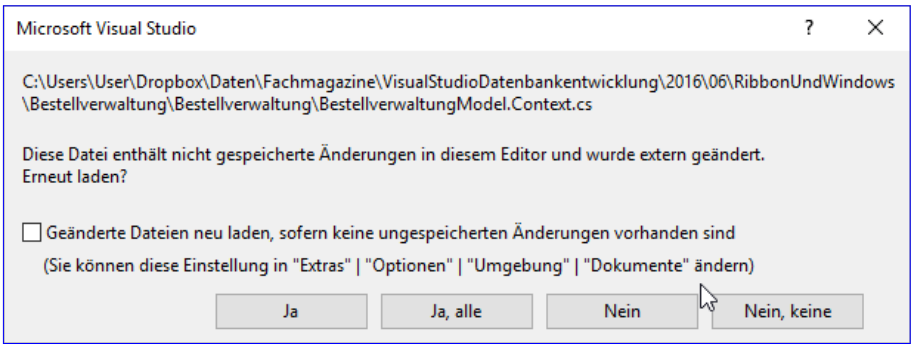


Bild 8: Information über die Änderung von Dateien

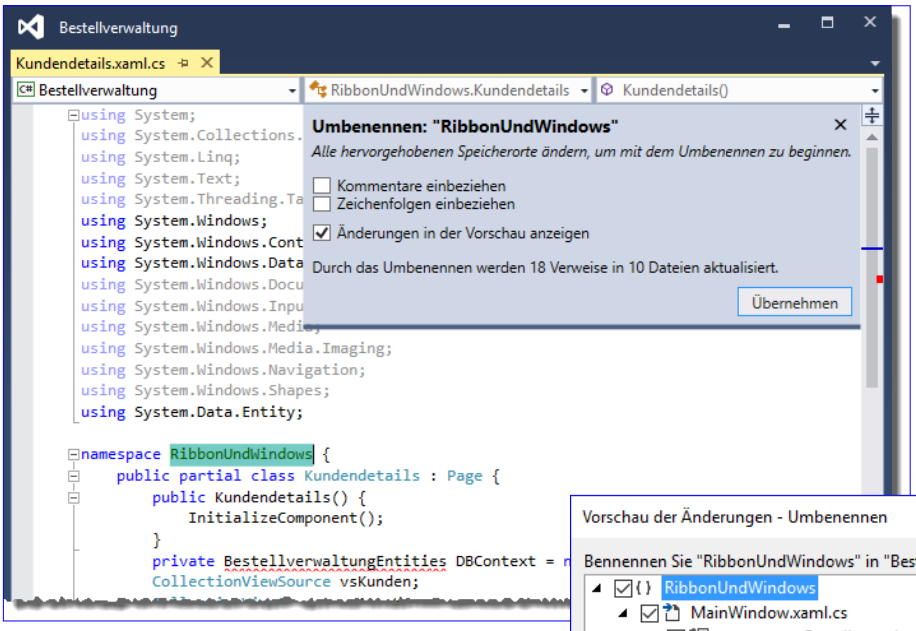


Bild 9: Aufrufen des Assistenten zum Umbenennen

Anzeige der Sanduhr deutet darauf hin, dass da im Hintergrund gearbeitet wird – eben, indem der Namespace des Entity Data Models angepasst wird. Den Beweis dafür liefert dann auch die Meldung aus Bild 8.

Probleme treten nun dadurch auf, dass die Klassen des Entity Data Models sich nicht mehr im gleichen Namespace wie die darauf zugreifenden Klassen sind und dadurch nicht mehr gefunden werden können.

Wenn wir aber nun den Namen des Namespace an den übrigen Stellen ändern, ist auch dieses Problem gelöst. Unter Visual Studio 2015 etwa können Sie nun, wenn Sie den zu ersetzenden Ausdruck an einer Stelle im Code markiert haben, einen Assistenten zum Ersetzen von Code mit der Tastenkombination **Strg + R, Strg + R** öffnen (siehe Bild 9). Den beim Öffnen des Dialogs markierten Text ersetzen Sie nun durch den gewünschten Text.

Klicken Sie dann auf **Übernehmen**, erscheint der Dialog aus Bild 10. Hier können Sie einsehen, an welchen Stellen die Änderung vorgenommen wird. Wir haben alle Vorkommen geändert und damit die Änderung vom alten zum neuen Namen inklusive Ordner, Dateien und Namespaces abgeschlossen.

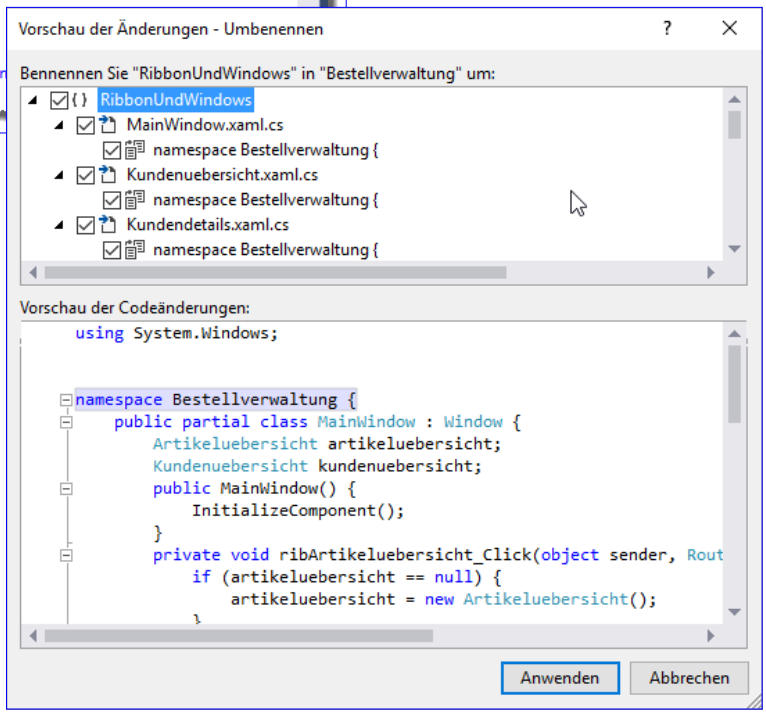


Bild 10: Der Assistent zum Umbenennen von Code