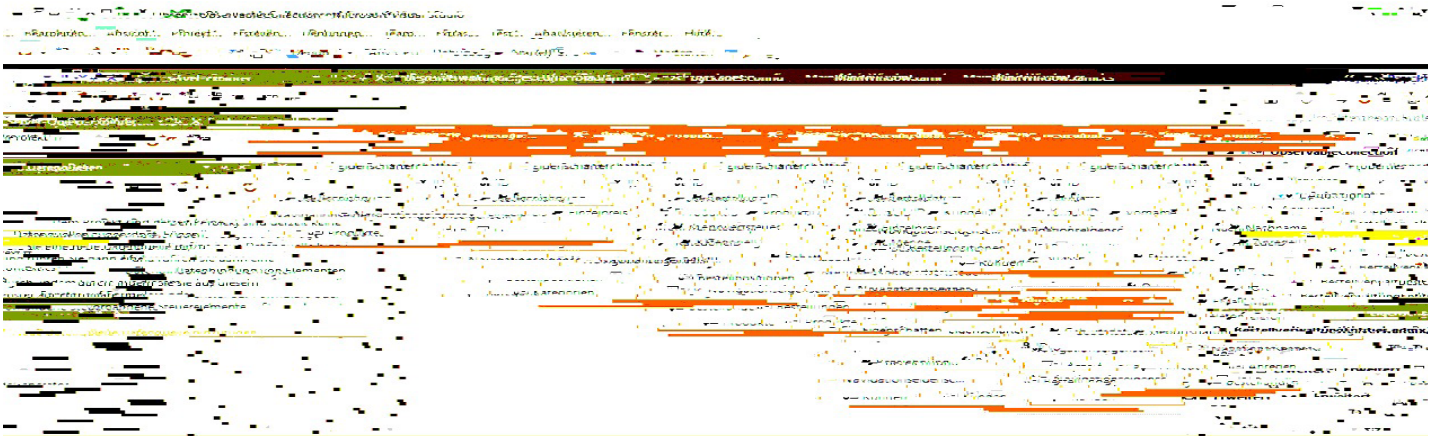
AUSGABE 05/2017

DATENBANK ENTWICKLER

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



TOP-THEMEN:

NEWS AND TOOLS	LINQPad: LINQ, C# und VB einfach ausprobieren	SEITE 1
USERINTERFACE	Drag and Drop-Grundlagen	SEITE 1
VB-BASICS	Von C# zu VB: Basics, Variablen, Operatoren	SEITE 21
INTERAKTIV	Excel-Export mit Spreadsheet Light	SEITE 47
LÖSUNGEN	Bestellverwaltung á la Visual Basic	SEITE 54

André Minhorst Verlag

NEWS UND TOOLS	LINQPad: LINQ, C# und VB einfach ausprobieren	3
WPF-GRUNDLAGEN	Property-Elemente am Beispiel Content	8
BENUTZEROBERFLÄCHE MIT WPF	Drag and Drop-Grundlagen	12
VISUAL BASIC-GRUNDLAGEN	Von C# zu VB: Basics, Variablen, Operatoren	21
C#-GRUNDLAGEN	Bubbling und Tunneling: Routed Events	31
WPF-STEUERELEMENTE	Drag and Drop mit ListBox-Elementen	38
INTERAKTIV	Excel-Export mit Spreadsheet Light	47
LÖSUNGEN	Bestellverwaltung á la Visual Basic	55
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

LINQPad: LINQ, C# und VB einfach ausprobieren

Es ist doch etwas nervig, wenn man mal eben ein paar LINQ-Abfragen testen möchte oder auch C#- oder VB-Code. Immer wieder den Code anpassen, kompilieren, starten, feststellen, dass es noch nicht wie gewünscht klappt und wieder von vorn. Unter Access/VBA ging das irgendwie einfacher: Prozedur schreiben, F5 drücken, fertig. Das Tool LINQPad erleichtert die Sache stark: Damit können Sie prima üben, ohne zum Testen immer gleich die Anwendung kompilieren zu müssen.

LINQPad

LINQPad ist eine Anwendung, die in einer abgespeckten Version kostenlos ist und nach der Installation gegen Entgelt mit einigen interessanten Features wie IntelliSense und mehr aufgerüstet werden kann. Die Bezeichnung von LINQPad heißt nicht umsonst **The .NET Programmer's Playground**: Hier können Sie viele Ausdrücke in den Sprachen C#, Visual Basic oder auch F# ausprobieren. Uns interessiert natürlich vor allem der Zugriff auf Datenbanken, denn wir wollen ja im Rahmen dieses Artikels weitere Methoden von LINQ to Entities am Beispiel unserer Datenbank **Bestellverwaltung** ausprobieren.

Den Download von LINQPad finden Sie unter der Internetadresse **www.linqpad.net**.

Nachdem Download und der Installation können Sie gleich loslegen und LINQPad starten. Die Benutzeroberfläche besteht aus drei Elementen: Links oben finden Sie einen Bereich, mit dem Sie die Verbindungen zu Ihren Datenquellen verwalten können. Links unten können Sie in bereits vorhandenen Beispielen suchen oder in dem für LINQPad definierten Bereich des Dateisystems Ihre eigenen Beispiele verwalten. Rechts finden Sie ein Register-Steuerelement, das ein oder mehrere Registerkarten mit Ihren selbst erstellen oder geladenen Skripten anbietet. Oben sind zwei wichtige Kombinationsfelder: **Language** legt nicht nur fest, mit welcher Programmiersprache Sie arbeiten, sondern auch, welche Art von Anweisungen Sie nutzen möchten (siehe Bild 1). Für unsere Zwecke, in denen wir zunächst nur LINQ-Ausdrücke testen wollen, reicht die Auswahl des Eintrags **C# Statement(s)** aus. Wenn Sie die Ausdrücke unter VB testen wollen, nutzen Sie den entsprechenden Eintrag.

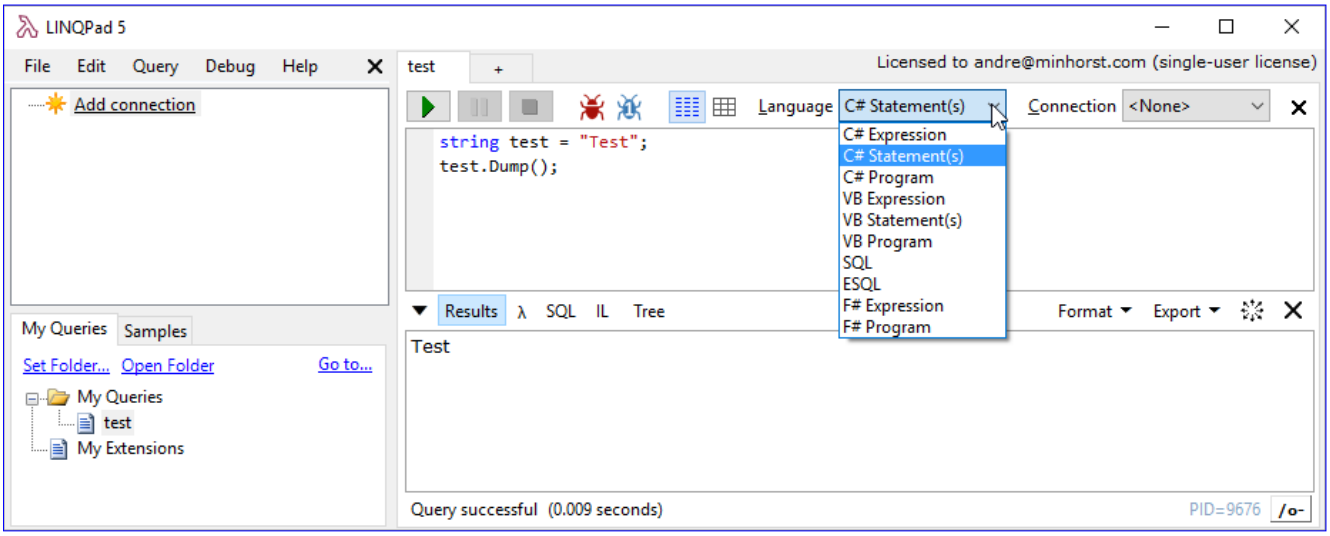


Bild 1: Ein erster Test mit LINQPad

Datenquelle wählen

Neben der Sprache benötigen wir natürlich eine Datenquelle. Um diese einzustellen, klicken Sie auf den Link **Add connection** im linken, oberen Bereich.

Dies öffnet den Dialog **Choose Data Context** (siehe Bild 2). Hier finden Sie zwei grundlegende Möglichkeiten:

- **Build data context automatically**: Erwartet die Angabe der Datenquelle, in unserem Fall zum Beispiel einer SQLite-Datenbankdatei, und erstellt die Objekte automatisch auf Basis der enthaltenen Tabellen und Beziehungen.
- **Use a typed data context from your own assembly**: Verwendet den in den Konfigurationsdateien definierten Datenbank-Kontext als Grundlage für die Entitäten. Dies hat mit SQLite in unseren Versuchen allerdings nicht geklappt.

Die erste Option bietet leider keinen passenden Treiber für unsere SQLite-Datenbank **Bestellverwaltung.db**. Allerdings finden wir unten im Dialog noch eine Schaltfläche mit der Beschriftung **Browse...**, mit der wir den Dialog aus Bild 3 öffnen.

Hier finden wir dann gleich ganz oben den Eintrag, der uns die Treiber unter anderem für SQLite nachreicht. Mit einem Klick auf **Download & Enable Driver** fügen wir den fehlenden Treiber hinzu.

Dieser erscheint dann nach dem Schließen des Dialogs auch in der oberen Liste des Dialog **Choose Data Context** (siehe Bild 4).

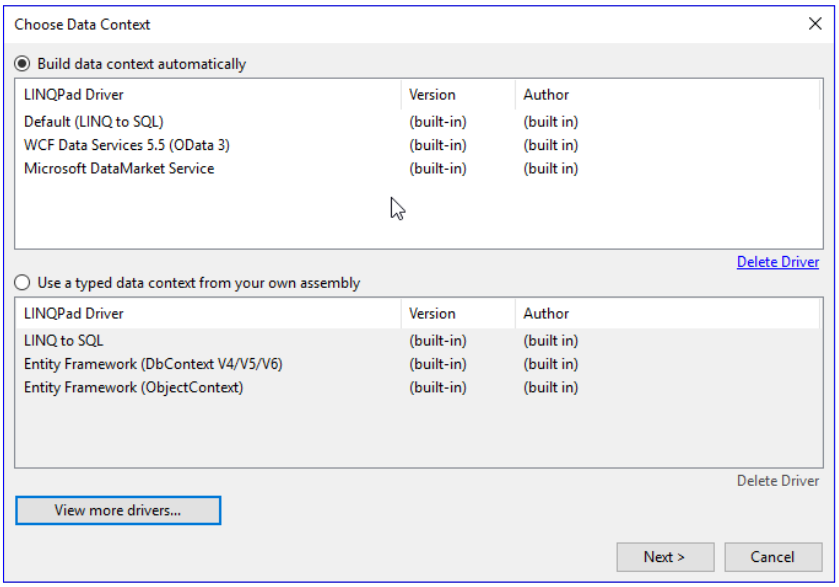


Bild 2: Auswählen des Datenbank-Kontexts

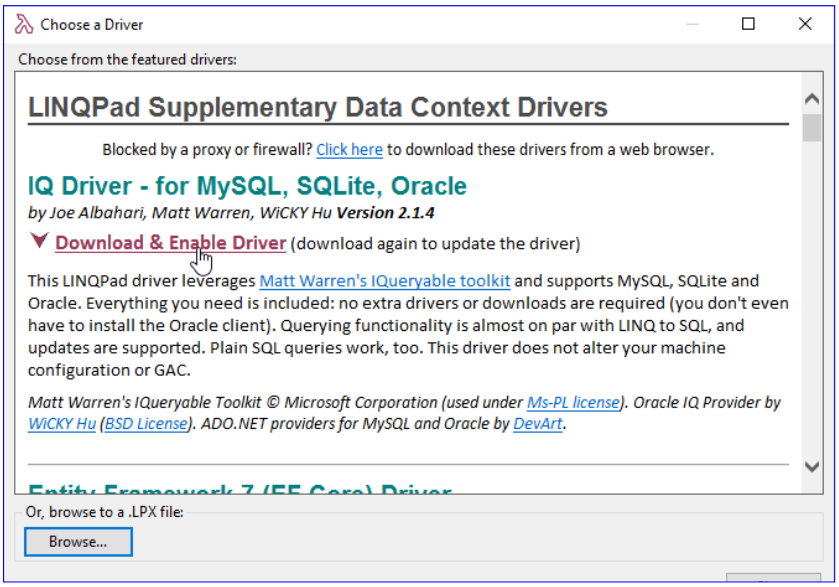


Bild 3: Herunterladen des Treibers für MySQL, SQLite und Oracle

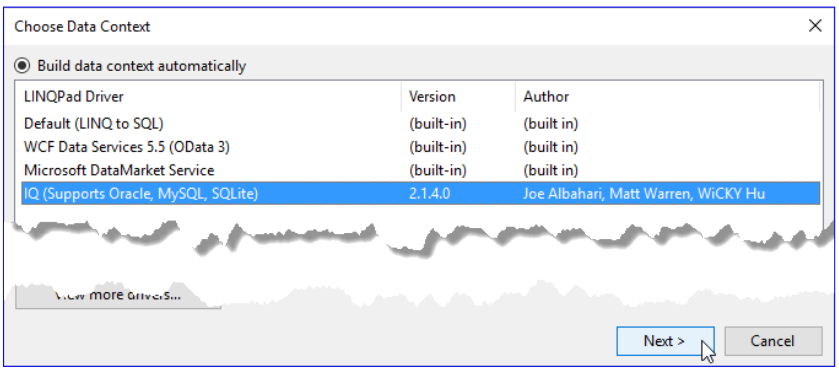


Bild 4: Auswählen des SQLite-Treibers

Property-Elemente am Beispiel Content

Unter Access konnten Sie beispielsweise in einer Schaltfläche lediglich einen Text als Inhalt unterbringen. In neueren Versionen kamen dann noch einige Features zur Gestaltung von Rand, Hintergrund, Bildern et cetera hinzu. Unter WPF sieht das ganz anders aus. Eine Schaltfläche bringt zwar mit der Content-Eigenschaft die Möglichkeit mit, einfache Texte einzugeben und liefert auch noch einige weitere Attribute, mit denen sich andere gängige Eigenschaften wie Rahmen, Hintergrund und so weiter einstellen lassen. Aber die Content-Eigenschaft nimmt nicht nur reine Texte, sondern nahezu beliebige Inhalte entgegen und zeigt diese innerhalb des Buttons an. Der Button ist dabei übrigens nur ein Beispiel – es gibt noch weitere Steuerelemente, welche die Content-Eigenschaft anbieten.

Einfache Texte

Wenn Sie wie beispielsweise unter Access einfach nur einfache Texte in einer Schaltfläche anzeigen möchten, weisen Sie diese einfach der Eigenschaft **Text** zu. Unter WPF gibt es dafür das Attribut **Content**, das wir im folgenden Beispiel mit dem Text **Beispielbutton** füllen (Ergebnis siehe Bild 1):

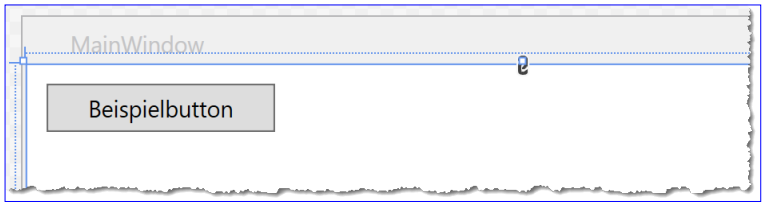


Bild 1: Button mit einfachem Text

```
<Button Height="25" Width="120" VerticalAlignment="Top" HorizontalAlignment="Left" Margin="10"
Content="Beispielbutton"></Button>
```

Content mit anderem Objekt füllen

Wenn Sie nun nicht nur einen einfachen Text als **Content** verwenden möchten, sondern beispielsweise ein anderes Objekt wie einen Text mit einem Bild, scheint dies mit dem **Content**-Attribut allein nicht möglich. Unter C# könnten Sie das aber beispielsweise wie folgt lösen:

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
        Button btn = new Button();
        btn.Width = 150;
        btn.Height = 32;
        StackPanel stp = new StackPanel();
        Image img = new Image();
        img.Source = new BitmapImage(new Uri(@"/ContentProperty;component/images/close.png", UriKind.Relative));
        img.Width = 24;
        img.Height = 24;
```

```
        img.Margin = (Thickness)System.ComponentModel.TypeDescriptor.GetConverter(typeof(Thickness)).
            ConvertFromInvariantString("5,0,5,0");
        TextBlock txt = new TextBlock();
        txt.Text = "Button mit Bild";
        txt.VerticalAlignment = VerticalAlignment.Center;
        stp.Children.Add(img);
        stp.Children.Add(txt);
        stp.Orientation = Orientation.Horizontal;
        btn.Content = stp;
        btn.HorizontalContentAlignment = HorizontalAlignment.Left;
        MyGrid.Children.Add(btn);
    }
}
```

Zusätzlich müssen Sie das **Grid**-Element im **.xaml**-Code noch entsprechend benennen, damit wir über die Bezeichnung **MyGrid** darauf zugreifen können:

```
<Grid x:Name="MyGrid">
```

Die Methode **MainWindow** erzeugt nun ein neues **Button**-Element, dem es zunächst eine Breite und eine Höhe zuweist. Dann erstellt sie ein neues **StackPanel**-Element, das die beiden enthaltenen Elemente, also das Bild und den Text nebeneinander anordnen soll. Das Bild bringen wir zunächst in der Variablen **img** unter und weisen seiner **source**-Eigenschaft über **new BitmapImage** ein im Projekt gespeichertes Bild zu. Danach stellt die Methode Höhe und Breite des Bildes ein sowie die horizontale Ausrichtung.

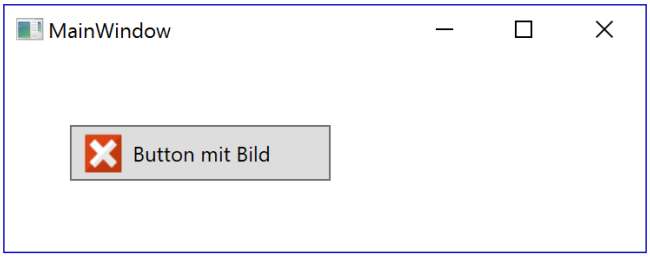


Bild 2: Per Code erstellte Schaltfläche mit Bild und Text

Den Abstand des **image**-Elements zum linken Rand des StackPanels und zum rechts daneben liegenden **TextBlock**-Element, das gleich noch erstellt wird, wollen wir mit dem **Margin**-Attribut einstellen. Was unter **.xaml** schnell per Zuweisung gelingt, erfordert hier den Einsatz einer Struktur des Typs **Thickness**, der wir für die Attribut **Left** und **Right** die entsprechenden Zahlenwerte zuweisen. Nun erstellen wir noch den **TextBlock**, der schließlich den gewünschten Text aufnimmt. Der Text soll vertikal zentriert eingeblendet werden. Nun können wir über die Methode **Add** der **Children**-Auflistung des **StackPanel**-Elements den Button und den TextBlock hinzufügen. Das StackPanel weisen wir dann der Eigenschaft **Content** des **Button**-Elements zu, das wir noch linkszentriert ausrichten, bevor wir das **Button**-Element noch im Grid einsetzen. Das Ergebnis sieht dann wie in Bild 2 aus.

Komplexer Content per .xaml

Wie nun wollen wir diese aufwendig per C# zusammengestellten Elemente per **.xaml** zu einem Window hinzufügen, wo wir doch nur das **Content**-Attribut zur Verfügung haben? Der Clou ist, dass es für **Button**- und andere Elemente noch eine weitere Möglichkeit gibt, den Content anzugeben, nämlich einfach in Form von Unterelementen. Das bedeutet, dass wir das **Content**-

Drag and Drop-Grundlagen

Unter Access fehlen einige Features, die in anderen Programmiersprachen und Entwicklungsumgebungen zum guten Ton gehören. Eines davon ist die Drag and Drop-Funktionalität, die sich nur aufwendig abbilden ließ – und auch nur mit bestimmten ActiveX-Steuerelementen. Die eingebauten Steuerelemente wie Textfelder oder Listenfelder ließen leider kein natives Drag and Drop zu. Unter WPF und den .NET-Programmiersprachen sieht das ganz anders aus. Dieser Artikel liefert Grundlagen zu Drag and Drop.

Wozu braucht man aber Drag and Drop? In der Tat lassen sich die meisten Tätigkeiten, die Sie damit ausführen können, auch auf andere Weise realisieren. Allerdings ist Drag and Drop und somit das einfache Ziehen von Elementen der Benutzeroberfläche mit der Maus doch ein sehr intuitiver Weg, um Aktionen durchzuführen. Ein Beispiel sind die beiden Listenfelder aus dem Artikel **m:n-Beziehung mit Listenfeld** (siehe Bild 1). Im Artikel haben wir Schaltflächen bereitgestellt, mit denen ein oder alle Artikel von einem Listenfeld zum anderen übertragen werden können. Nun wollen wir uns weiter vortasten und Drag and Drop-Funktionalität zu diesem Fenster hinzufügen. Bevor wir uns das im Artikel **Drag And Drog mit ListBox**-Elementen ansehen, schauen wir uns einige grundlegende Beispielen an.

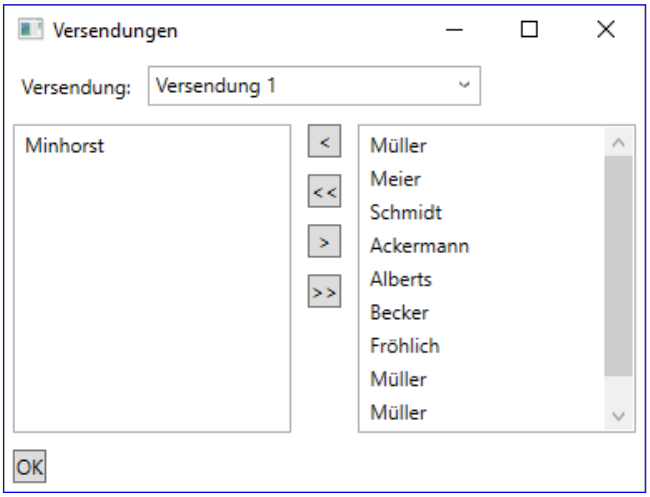


Bild 1: Für das Hin- und Herziehen von Einträgen zweier Listenfelder wäre Drag and Drop eine interessante Alternative.

Was ist Drag and Drop?

Drag and Drop nennen wir den Vorgang, bei dem der Benutzer ein Element der Benutzeroberfläche mit der linken Maustaste anklickt und diese dabei gedrückt hält. Dann wird die Maus mit dem Element zum Zielelement bewegt und das zu bewegende Element dort durch Loslassen der linken Maustaste an das Zielelement übergeben.

Dabei müssen Start- und Zielelement gar nicht unbedingt zwei verschiedene Elemente sein wie beim Beispiel mit den Listenfeldern. Es kann auch nur ein einziges Listenfeld sein, dessen Einträge Sie per Drag and Drop umsortieren wollen. Oder Sie bewegen die Elemente in einem **TreeView**-Steuerelement von einem übergeordneten Element zum nächsten.

Dazu benötigen wir ein paar Informationen. Zunächst müssen wir für das abgebende Element festlegen, dass es das Ziehen der enthaltenen Elemente mit der Maus überhaupt erlaubt. Dann wollen wir uns irgendwie merken, welches Element wir mit der Maus angepackt haben, um es zu verschieben (oder auch zu kopieren). Schließlich soll das aufnehmende Element mit der Möglichkeit zum Aufnehmen des zu übergebenden Elements ausgestattet werden, also als Drag and Drop-Ziel verfügbar gemacht werden. Da man unter Windows eine ganze Menge Dinge durch die Gegend ziehen kann – zum Beispiel von Windows Explorer auf verschiedene andere Ziele – müssen Sie dem Zielelement auch noch mitteilen, auf welche zu übergebenden Elemente es überhaupt reagieren soll, indem es das Symbol als mögliches Zielelement einblendet. Schließlich benötigen wir verschiedene

Ereignisse, für die wir passende Methoden implementieren, um das Verhalten von Drag and Drop-Quelle und -Ziel festzulegen.

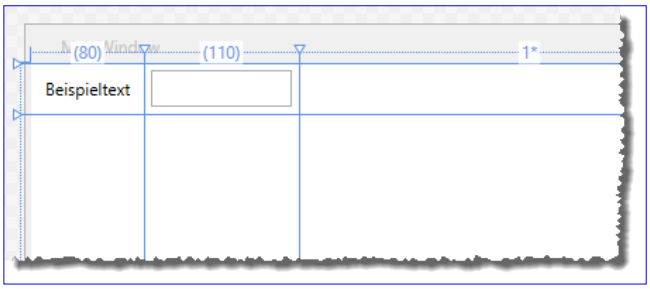


Bild 2: Label und TextBox als erstes Drag and Drop-Beispiel

Drag and Drop ist auch eine Alternative zum Kopieren/Ausschneiden und Einfügen. Der wesentliche Unterschied neben den dazu notwendigen Techniken (Ziehen mit der Maus bei Drag and Drop auf der einen, Tastenkombinationen wie **Strg + X**, **V** oder **C** und Kontextmenü-Einträge auf der anderen Seite) ist der Ort, an dem wir uns merken, welches Element von A nach B bewegt werden sollen. Beim Kopieren/Ausschneiden und Einfügen ist dies die Zwischenablage. Bei Drag and Drop benötigen wir eine andere Möglichkeit, in diesem Fall ein Element des Typs **DataObject**.

Einfaches Beispiel: Bezeichnungsfeld in Textfeld ziehen

Wir schauen uns zu Beginn ein einfaches Beispiel an, bei dem wir die Beschriftung eines Bezeichnungsfeldes einfach in ein Textfeld ziehen wollen. Dazu legen wir entsprechende Steuerelemente in einem neuen Fenster namens **MainWindow.xaml** eines neuen Projekts an (s. Bild 2). Die beiden Steuerelemente definieren wir wie wie folgt:

```
<TextBox x:Name="txtDrop" Grid.Column="1" Grid.Row="1" Margin="5" Drop="txtDrop_Drop" Width="100" />
<Label x:Name="lblDrag" Content="Beispieltext" Grid.Column="0" Grid.Row="1" Margin="5" MouseDown="lblDrag_MouseDown" />
```

Unter C# sieht die Ereignismethode für das Ereignis **MouseDown** des Bezeichnungsfeldes wie folgt aus:

```
//C#
private void lblDrag_MouseDown(object sender, MouseButtonEventArgs e) {
    DragDrop.DoDragDrop(lblDrag, lblDrag.Content, DragDropEffects.Copy);
}
```

Wir benötigen also nur eine einzige Zeile, um den Vorgang zu starten! Dabei rufen wir die Methode **DoDragDrop** der Klasse **DragDrop** auf. Diese erwartet drei Parameter:

- **dragSource**: Quelle des Drag and Drop-Vorgangs, also beispielsweise ein Steuerelement wie in diesem Fall das Bezeichnungsfeld
- **data**: Die per Drag and Drop bewegten Daten werden mit dem Parameter **data** in einen Zwischenspeicher ähnlich der Zwischenablage gegeben und können dort beim Drop wieder abgerufen werden.
- **allowedEffects**: Effekte sind verschiedene Darstellungsarten des Mauszeigers.

Im vorliegenden Fall übergeben wir für den ersten Parameter einen Verweis auf das Bezeichnungsfeld **lblLabel**. Der zweite Parameter nimmt den Inhalt der Eigenschaft **Content** des Bezeichnungsfeldes auf, hier eine einfache Beschriftung. Da es sich

hierbei um ein reines Kopieren von Daten handelt, nämlich das Kopieren der Beschriftung des Bezeichnungsfeldes in das Textfeld, soll nur das Symbol für das Kopieren mit der Maus angezeigt werden (siehe Bild 3).

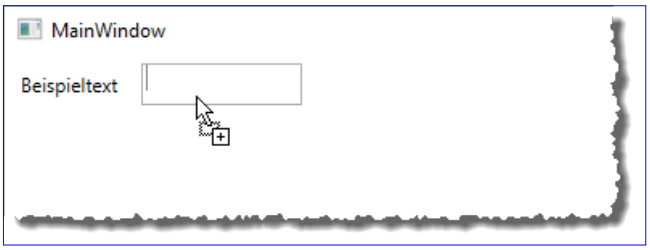


Bild 3: Veränderter Mauszeiger beim Drag and Drop

Damit kommen wir Fallenlassen des zu kopierenden Inhalts über dem Zielobjekt. Hier verwenden wir nun das Ereignis Drop des Textfeldes, das wir wie folgt implementieren:

```
//C#
private void txtDrop_Drop(object sender, DragEventArgs e) {
    string str;
    str = (string)e.Data.GetData(DataFormats.StringFormat);
    txtDrop.Text = str;
}
```

Die Methode liefert die gewohnten beiden Parameter, wobei der zweite den Typ **DragEventArgs** hat (mehr dazu weiter unten). In diesem Fall nutzen wir das Objekt **data** der mit dem Parameter e übergebenen **DragEventArgs**-Klasse, dessen Methode **GetData** mit dem Parameter **DataFormats.StringFormat** den gespeicherten Text zurückgibt. Diesen speichern wir in der String-Variablen **str** und weisen diese dann der **Text**-Eigenschaft des Textfeldes zu. Das Ergebnis sieht schließlich wie in Bild 4 aus.

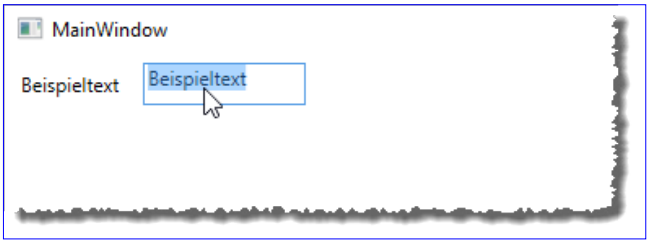


Bild 4: Erfolgreicher Drag and Drop-Vorgang

Zusammengefasst haben wir für den Start von Drag and Drop das Ereignis **MouseDown** genutzt, dort den Drag and Drop-Vorgang gestartet und dadurch auch das Ereignis **Drop** aktiviert. Ohne den Aufruf von **DoDragDrop** können Sie nämlich mit der Maus ziehen und fallenlassen, was sie möchten – das **Drop**-Ereignis wird nicht ausgelöst.

Verfeinerung: Maustaste prüfen

Im aktuellen Zustand können Sie auch mit der rechten Maustaste den Drag and Drop-Vorgang ausführen. Das liegt daran, dass wir in der Methode **lblDrag_MouseOver** nicht geprüft haben, welche Maustaste gerade gedrückt wurde. Dies holen wir nun nach und passen die Methode wie folgt an:

```
private void lblDrag_MouseDown(object sender, MouseButtonEventArgs e) {
    if (e.LeftButton == MouseButtonState.Pressed) {
        DragDrop.DoDragDrop(lblDrag, lblDrag.Content, DragDropEffects.Copy);
    }
}
```

Hier prüfen wir vorab, ob die linke Maustaste gedrückt ist. Damit wird das Drag and Drop mit gedrückter rechter Maustaste unterbunden.

Drag and Drop aus anderen Anwendungen

Wenn Sie einen Text aus einer anderen Anwendung in ein Textfeld einer WPF-Anwendung ziehen wollen, klappt das mit der gegebenen Konfiguration ebenfalls. Sie können nun beispielsweise Text aus einem Text-Editor oder aus Word markieren und diesen dann in das Textfeld ziehen. Aber: Dazu wären die bisherigen Schritte gar nicht nötig gewesen – Sie können auch einfach so den Text aus einer externen Anwendung in das Textfeld ziehen. Dazu müssen Sie noch nicht einmal ein Attribut des Textfeldes anpassen. Interessant ist noch die Frage: Feuert denn das Drop-Ereignis des Textfeldes, wenn Text aus einer externen Anwendung abgelegt wird? Die Antwort lautet nein.

Drag and Drop in andere Anwendungen

Bei Texten gelingt das Drag and Drop aus einem Textfeld in eine andere Anwendung, welche als Drop-Ziel ausgelegt ist wie etwa Microsoft Word, ohne weiteres Zutun.

Auch wenn man einen Drag and Drop-Vorgang in der WPF-Anwendung so beginnt, dass der zu verschiebende Inhalt in die Drag and Drop-Zwischenablage übertragen wird, können Sie den Inhalt etwa in eine Textverarbeitung übertragen. Auf diese Weise können Sie etwa den Text der Schaltfläche aus dem ersten Beispiel in eine Textverarbeitung statt in das Textfeld ziehen.

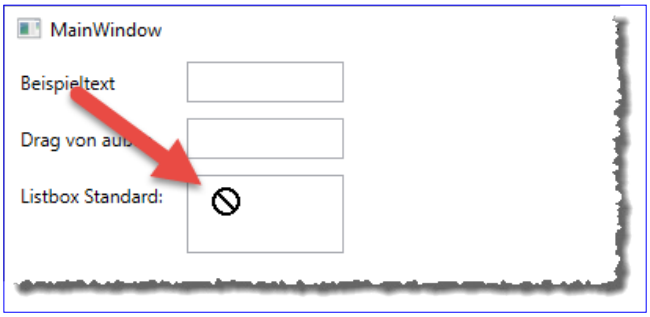


Bild 5: Drag and Drop auf eine Standard-ListBox

Drag and Drop in eine ListBox

Textfelder sind also immer für Drag and Drop empfänglich, wenn es sich um Daten im passenden Typ handelt, also um Texte. Was aber, wenn wir einen Text auf ein anderes Steuerelement droppen – beispielsweise ein **ListBox**-Steuerelement? Dazu fügen wir unserem Beispielformular eine ListBox hinzu und nennen diese **lstDrop**:

```
<ListBox x:Name="lstDropStandard" Grid.Column="1" Grid.Row="3" Margin="5" Width="100" Height="50" />
```

Wenn wir den Text des Bezeichnungsfeldes **lblDrag** auf das **ListBox**-Element ziehen, wird der Mauszeiger in ein Symbol umgewandelt, dass deutlich macht, dass das **ListBox**-Element kein valides Ziel für die Drag and Drop-Anweisung ist (siehe Bild 5). Also stellen wir nun das Attribut **AllowDrop** auf **True** ein:

```
<ListBox x:Name="lstDropStandard" ... AllowDrop="True" />
```

Damit erscheint schon einmal das richtige Symbol, wenn Sie das **ListBox**-Element beim Drag and Drop mit der Maus überfahren. Beim Fallenlassen geschieht jedoch nichts. Kein Wunder: Ein **ListBox**-Element nimmt seine Inhalte ja auch über ganz andere Wege entgegen – zum Beispiel über die **Add**-Methode der **Items**-Auflistung. Diese implementieren wir nun für die **Drop**-Methode des **ListBox**-Element, das wir dazu als neues Element wie folgt erweitern:

Von C# zu VB: Basics, Variablen, Operatoren

Wer bisher mit C# gearbeitet hat und zu VB wechseln möchte, sieht sich bei der Entwicklung von WPF-Anwendungen einigen Änderungen gegenüber. Dieser Artikel fasst die wichtigsten Elemente der Sprache Visual Basic 2015 für den Entwickler in Bezug auf die bisher in diesem Magazin unter C# durchgeführten Programmierungen zusammen. Zum Experimentieren mit den Beispielen nutzen wir das Tool LINQPad 5. Außerdem schauen wir uns in diesem Artikel die Grundlagen der Sprache an, die wir im Artikel »Von VBA zu C#: Erste Anwendung und Variablen« betrachtet haben.

WPF-Fenster: Wo ist der Code im Code behind-Modul?

Wer zuvor mit C# gearbeitet hat, weiß, dass ein Code behind-Modul eines WPF-Fensters bereits einigen Code enthält. Wenn Sie hingegen ein VB/WPF-Projekt erstellen und sich das Code behind-Fenster ansehen, erblicken Sie eine ziemliche Leere.

Dies ist der vollständige Inhalt des Moduls, hier am Beispiel von **MainWindow.xaml.vb**:

```
Class MainWindow
End Class
```

Konstruktor hinzufügen

Den liebgewonnenen Konstruktor, also die Methode, die automatisch beim Initialisieren des Fensters ausgelöst wird und die Sie auch so anpassen können, dass Sie beim Aufruf benutzerdefinierte Parameter übergeben können, fehlt gänzlich. Wie also wollen Sie beim Aktionen programmieren, die beim Initialisieren des Fensters durchgeführt werden geschweige denn eigene Parameter definieren?

Das geht einfacher, als Sie denken – Sie fügen einfach die Zeile **Public Sub New()** hinzu und erhalten dann die wie folgt automatisch ergänzte Methode:

```
Public Sub New()
        ' Dieser Aufruf ist für den Designer erforderlich.
        InitializeComponent()
        ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.
End Sub
```

Hier können Sie dann eigene, beim Initialisieren auszuführende Anweisungen eintragen. Außerdem können Sie die **New()**-Methode natürlich auch um benutzerdefinierte Parameter erweitern.

Groß- und Kleinschreibung

Was als Nächstes auffällt, ist die Großschreibung von Schlüsselwörtern. Dies ist einer der Unterschiede zwischen C# und Visual Basic 2015.

Case Sensitive oder nicht?

Case Sensitive bedeutet, dass die Groß- und Kleinschreibung einen Unterschied macht – **a** ist also ungleich **A**. Das heißt auch, dass Sie unter C# beispielsweise die folgenden beiden **String**-Variablen deklarieren können:

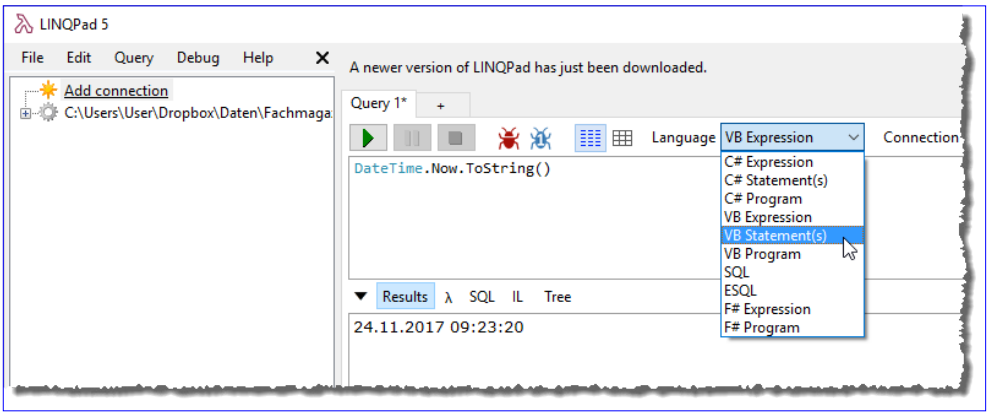


Bild 1: Auswahl der verschiedenen Eingabemöglichkeiten

```
string a;
string A;
```

Unter Visual Basic erhalten Sie für die zweite Zeile einen Fehler:

```
Dim a As String
Dim A As String
```

Variablen und Typen/Klassen können jedoch in beiden Sprachen mit unterschiedlicher Groß-/Kleinschreibung genutzt werden, also etwa wie folgt, wo wir **Test1** als Klassennamen und **test1** als Objektname auf Basis dieser Klasse nutzen:

```
Class Test1
    '...
End Class

Class Test2
    Private Sub Test()
        Dim test1 As Test1
    End Sub
End Class
```

Beispiele mit LINQPad 5

Für die folgenden, einführenden Beispiele in Visual Basic 2015 nutzen wir **LINQPad 5**, wo wir deutlich einfacher experimentieren können als in Visual

Bild 2: VB-Ausdrucks, dessen Ergebnis dann im Ausgabebereich landet

Bild 3: Eingabe einiger Anweisungen, die nacheinander ausgeführt werden.

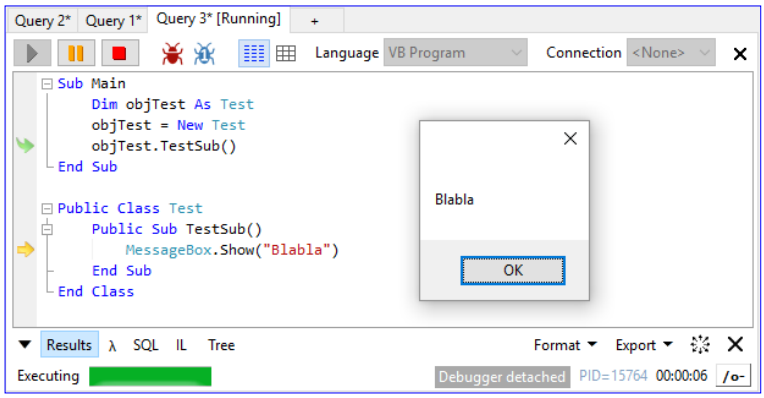


Bild 4: Ausführen der Methode einer Klasse

Studio, wo wir zumindest eine Konsolenanwendung brauchen, die wir immer wieder ausführen müssen.

Den Download von **LINQPad 5** finden Sie unter der Internetadresse **www.linqpad.net**.

Nach der Installation und dem Start von LINQPad 5 finden Sie im Hauptbereich ein Kombinationsfeld namens **Language**, mit der Sie die drei folgenden Optionen für Visual Basic einstellen können (siehe Bild 1):

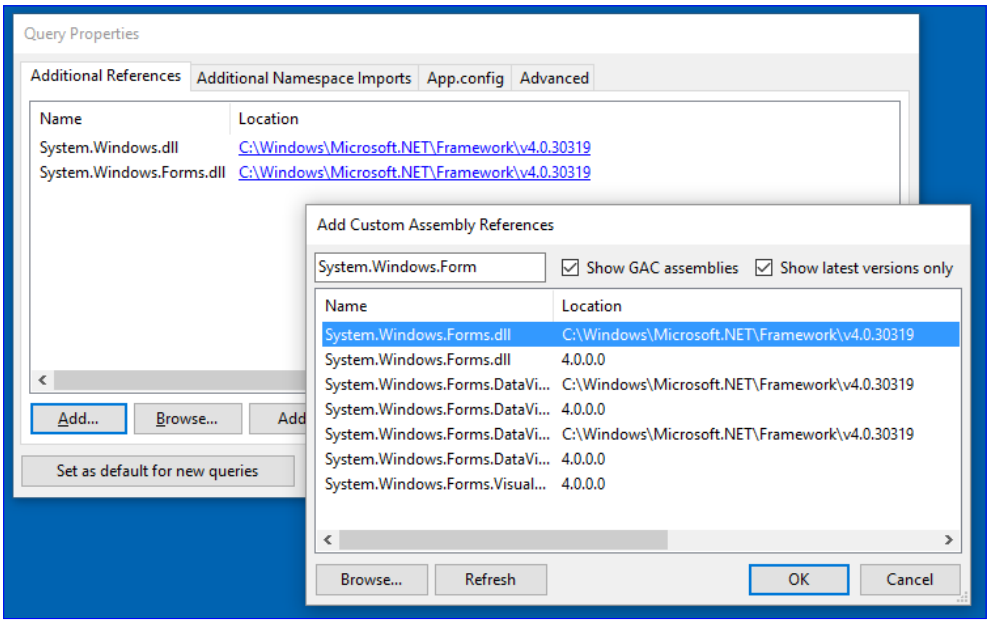


Bild 5: Hinzufügen von Bibliotheken

- **VB Expression**: Testen einfacher Ausdrücke wie etwa **DateTime.Now.ToString()** durch Eingabe und anschließende Betätigung von **F5** (siehe Bild 2)

- **VB Statement(s)**: Führt mehrere Anweisungen aus, also etwa den Inhalt einer Methode – nur eben ohne die übliche **Public Sub .../End Sub**-Zeilen (siehe Bild 3). Dazu ist wiederum das Betätigen der Taste **F5** beziehungsweise der Starten-Schaltfläche nötig.

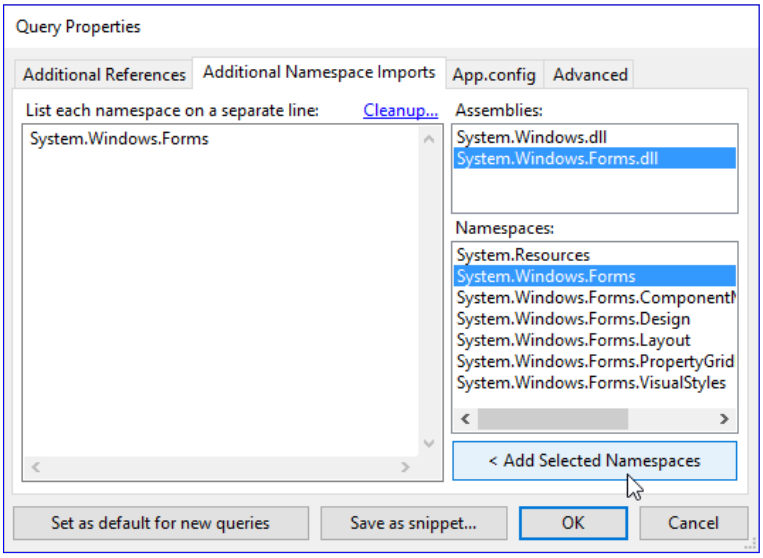


Bild 6: Hinzufügen von Namespaces

- **VB Program**: Hier können Sie komplette Methoden, Klassen und so weiter einbinden.
  Eine neue Registerseite dieses Typs blendet die Methode **Sub Main** ein, die beim Ausführen mit **F5** gestartet wird. Darunter können Sie beliebige weitere Elemente einbinden (siehe Bild 4). Mehr dazu in den folgenden Beispielen!

Namespaces hinzufügen

Unter C# fügen Sie zusätzliche Namespaces mit der **using**-Anweisung hinzu. Visual Basic nutzt stattdessen die **Imports**-Anweisung:

```
Imports System.Windows
```

In **LINQPad 5** können Sie diese Zeile in der aktuellen Version noch nicht nutzen. Dazu verwenden Sie einen kleinen Umweg:

- Klicken Sie mit der rechten Maustaste auf den Tab-Reiter und wählen Sie den Kontextmenü-Eintrag **Query Properties...** aus.
- Klicken Sie auf der Registerseite **Additional References** auf **Add...** und wählen Sie die hinzuzufügende Bibliothek aus, zum Beispiel **System.Windows.Forms.dll** (siehe Bild 5).
- Um dann den Namespace für den Code in **LINQPad 5** nutzen zu können, wechseln Sie auf die Seite **Additional Namespace Imports**. Klicken Sie hier rechts oben auf den Link **Pick from assemblies**. Dadurch wird ein weiterer Bereich eingeblendet, aus dem Sie die zu nutzenden Namespaces auswählen und mit der Schaltfläche **Add Selected Namespace** zur aktuellen Seite hinzufügen können (siehe Bild 6). Damit können Sie nun beispielsweise die **MessageBox**-Klasse nutzen.

Namespaces im Code definieren

Um einen Namespace im Code zu definieren, verwenden Sie genau dieselbe Anweisung wie unter C#:

```
Namespace Beispielnamespace
```

Unter LINQPad 5 können Sie diese Anweisung allerdings nicht nutzen – aber wenn Sie soweit sind, können Sie dies ja auch gleich im entsprechenden Projekt-Kontext testen.

Sub und Function

Genau wie unter VBA und im Gegensatz zu C# nutzen wir in Visual Basic 2015 die beiden Schlüsselwörter **Sub** und **Function**, um Prozeduren (und Methoden) sowie Funktionen zu definieren – das Schlüsselwort **void** von C# fällt hier also komplett weg. Beiden können Sie kein, ein oder mehrere Parameter übergeben. Die Funktion erhält am Ende wie unter VBA ein **As**-Schlüsselwort mit der Angabe des Datentyps für den Rückgabewert. Dieser gibt ihren Funktionswert zurück:

```
Public Function TestFunction() As String
    TestFunction = "bla"
End Function
```

Die Parameter werden wie unter VBA in einer kommaseparierten Liste in Klammern hinter dem Methodennamen übergeben, wobei dahinter das Schlüsselwort **As** und der Datentyp folgen:

```
Public Function TestFunction(strTest As String) As String
```

Datentypen

Unter Visual Basic 2015 gibt es, wie unter VBA, Werttypen, aber auch Verweistypen. Auch die einfachen Datentypen werden unter .NET bereits wie Objekte behandelt – zumindest, da diese eigene Eigenschaften bereitstellen. Hier sind die wichtigsten Datentypen für Visual Basic 2015 (in Klammern der Datentyp im .NET-Framework):

- **SByte**: -128 bis 127 (**System.SByte**), 8-Bit-Ganzzahl

- **Byte**: 0 bis 255 (**System.Byte**), vorzeichenlose 8-Bit-Ganzzahl
- **Short**: -32.768 bis 32.767 (**System.Int16**), 16-Bit-Ganzzahl
- **UShort**: 0 bis 65.535 (**System.UInt16**), vorzeichenlose 16-Bit-Ganzzahl
- **Integer**: -2.147.483.648 bis -2.147.483.647 (**System.Int32**), 32-Bit-Ganzzahl
- **UInteger**: 0 bis 4.294.967.295 (**System.UInt32**), vorzeichenlose 32-Bit-Ganzzahl
- **Long**: -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807 (**System.Int64**), 64-Bit-Ganzzahl
- **ULong**: 0 bis 18.446.744.073.709.551.615 (**System.UInt64**), vorzeichenlose 64-Bit-Ganzzahl
- **Char**: Unicode-Zeichen: U+0000 bis U+ffff (**System.Char**)

Das führende **u** in den Datentypen steht hier jeweils für **unsigned**, also ohne Vorzeichen. Bei **sbyte** ist dies inkonsequenterweise anders. Es gibt kein **int** wie in C#, dafür **Int32**. Statt **Short** können Sie auch **Int16** verwenden, statt **Long** nutzen Sie auch **Int32**. Im Gegensatz zu C# werden die Datentypen hier mit führendem Großbuchstaben geschrieben (bei denen mit Präfix **U** oder **S** auch mit zwei großen Buchstaben). Dies sind die Datentypen für Gleitkommazahlen:

- **Single**: Wird unter Visual Basic 2015 statt des Datentyps **Float** von C# verwendet. -3,402823E38 bis 3,402823E38.
- **Double**: –1,79769E308 bis 1,79769E308
- **Decimal**: -7,9E28 bis 7,9E28

Für **Ja/Nein**-Werte stellt Visual Basic 2015 den Datentyp **Bool** zur Verfügung, der die Werte **True** und **False** entgegennimmt (aber im Gegensatz zu C# auch die Zahlenwerte -1 und 0). Der Datentyp **String** nimmt Zeichenketten auf. Der Datentyp für Datumsangaben wird unter C# mit **Datetime** bezeichnet, unter Visual Studio 2015 verwenden wir **Date** wie unter VBA:

```
Dim datDatum As Date
```

Der Datentyp **Variant** wird in Visual Basic nicht unterstützt. Wenn Sie also wie in VBA mehrere Variablen in einer Zeile und durch Kommata getrennt deklarieren, werden diese auch mit dem am Ende der Zeile angegebenen Datentyp deklariert:

```
Dim intX, intY As Integer
```

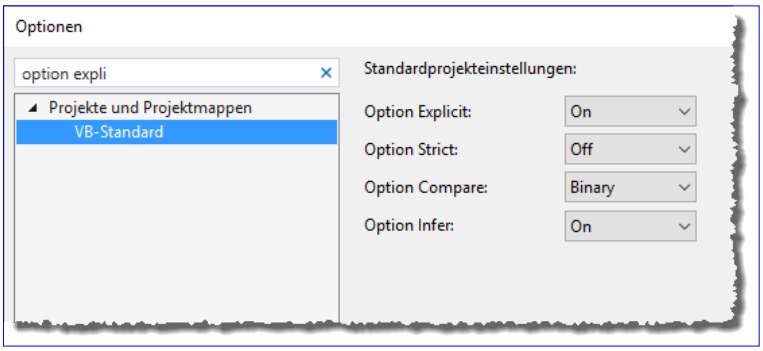


Bild 7: Variablendeklaration erzwingen

Bubbling und Tunneling: Routed Events

Im Vergleich zu VBA, wo jedes Ereignis für das Steuerelement behandelt wurde, welches es auch ausgelöst hat, gibt es unter WPF einige Erweiterungen. Es gibt dort auch solche Ereignisse, aber in vielen Fällen werden Ereignisse an übergeordnete Elemente weitergeleitet. Das hört sich erstmal so an, als ob man es nicht unbedingt benötigt. Dennoch wollen wir das Prinzip anhand eines Beispiels erläutern, damit Sie mitunter auftretendes unerwartetes Verhalten von Code interpretieren können.

Unter WPF sind Elemente verschachtelt, das heißt, es gibt übergeordnete und untergeordnete Elemente. Das **Window**-Element ist normalerweise das Hauptelement, welches dann ein **Grid** enthält, in dem sich wiederum ein **Stackpanel** mit einer Schaltfläche befinden könnte:

```
<Window x:Class="BubblingAndTunneling.MainWindow" ... Title="MainWindow" Height="350" Width="525">
    <Grid>
        <StackPanel>
            <Button x:Name="btn" Content="Klick mich!" ... Click="btn_Click"></Button>
        </StackPanel>
    </Grid>
</Window>
```

Direkte Ereignisse/Direct Events

Wenn der Benutzer nun auf die Schaltfläche namens **btn** klickt, wird das Ereignis **Click** ausgelöst, welches üblicherweise durch eine Ereignismethode etwa namens **btn_Click** implementiert wird:

```
private void btn_Click(object sender, RoutedEventArgs e) {
    MessageBox.Show("Klick!");
}
```

Bei dem **Click**-Ereignis handelt es sich um ein direktes Ereignis. Dieses wird nur von dem Element ausgelöst, für das es implementiert wurde.

Tunneling Events

Bei Tunneling Events handelt es sich um alle Ereignisse, die mit dem Präfix **Preview** beginnen, also beispielsweise **PreviewMouseDown**. Was ist die Besonderheit dieser Ereignisse? Sie werden immer vom Element der obersten Ebene aus zu den unteren Ebenen abgearbeitet. Wenn Sie also das Ereignis **PreviewMouseDown** für alle vier Elemente des obigen Beispiels definieren, sieht das wie folgt aus:

```
<Window x:Class="BubblingAndTunneling.MainWindow" ... Title="MainWindow" Height="350" Width="525" MouseDown="Window_
MouseDown" PreviewMouseDown="Window_PreviewMouseDown">
```

```
    <Grid MouseDown="Grid_MouseDown" PreviewMouseDown="Grid_PreviewMouseDown" >
        <StackPanel MouseDown="StackPanel_MouseDown" PreviewMouseDown="StackPanel_PreviewMouseDown">
            <Button x:Name="btn" Content="Klick mich!" ... MouseDown="btn_MouseDown"
                PreviewMouseDown="btn_PreviewMouseDown"></Button>
        </StackPanel>
    </Grid>
</Window>
```

Für diese Ereignisse hinterlegen wir die folgenden Methoden (für **Debug.WriteLine** müssen Sie den passenden Namespace mit **using System.Diagnostics** hinzufügen):

```
private void Window_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("Window_PreviewMouseDown");
}
private void Window_MouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("Window_MouseDown");
}
private void Grid_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("Grid_PreviewMouseDown");
}
private void Grid_MouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("Grid_MouseDown");
}
private void StackPanel_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("StackPanel_PreviewMouseDown");
}
private void StackPanel_MouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("StackPanel_MouseDown");
}
private void btn_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("btn_PreviewMouseDown");
}
private void btn_MouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("btn_MouseDown");
}
```

Wenn wir nun die Anwendung starten und mit der linken Maustaste auf die Schaltfläche klicken, erhalten wir das Ergebnis aus Bild 1. Es werden also offensichtlich nur die Tunneling-Methoden ausgelöst, also diejenigen, die mit **Preview...** beginnen.

Immerhin ist hier gut zu erkennen, dass dies ausgehend von der obersten Ebene (**Window**) bis runter zum **Button**-Element geschieht. Aber warum werden die **MouseDown**-Ereignisse nicht ausgelöst?

Dies ist wiederum der Fall, wenn wir auf den freien Bereich neben der Schaltfläche klicken. Dann erhalten wir die folgende Ausgabe:

```
Window_PreviewMouseDown
Grid_PreviewMouseDown
StackPanel_PreviewMouseDown
StackPanel_MouseDown
Grid_MouseDown
Window_MouseDown
```

Hier werden also korrekt erst die Tunneling-Ereignisse ausgelöst und dann in umgekehrte Reihenfolge die Bubbling-Ereignisse.

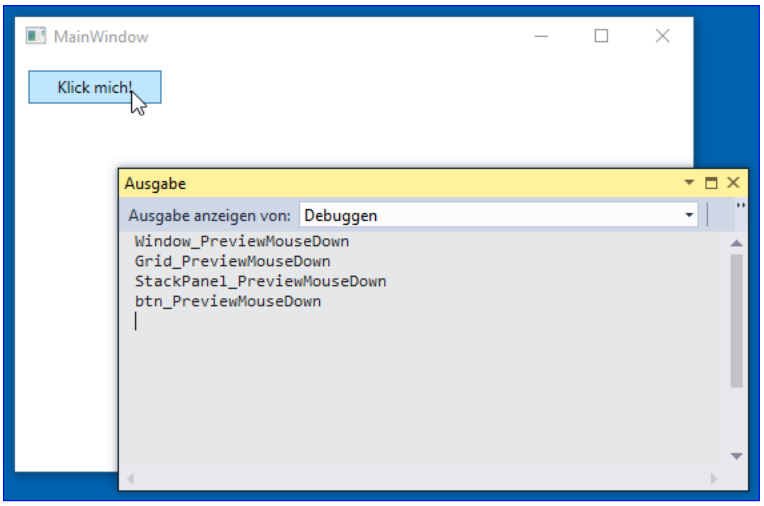


Bild 1: Es feuert nur die Hälfte der angegebenen Ereignisse.

Bubbling Events

Bevor wir uns darum kümmern, warum beim Anklicken der Schaltfläche nur die Tunneling-Events ausgelöst werden, noch kurz die Erläuterung der Bubbling-Ereignisse. Dies sind die Ereignisse, die in umgekehrter Reihenfolge wie die Tunneling-Ereignisse ausgelöst werden – also ausgehend vom auslösenden Element bis hin zum obersten Element der Hierarchie (wie gut im Beispiel oben zu sehen, wo wir nicht die Schaltfläche, sondern das **StackPanel**-Element angeklickt haben). Diese werden allerdings immer nach den Tunneling-Ereignissen abgearbeitet.

Warum kein MouseDown?

Nun schauen wir uns an, warum die Bubbling Events nicht ausgeführt werden, wenn wir das **Button**-Element mit der linken Maustaste anklicken. Der Grund ist einfach: Wenn Sie mit der linken Maustaste auf einen Button klicken, wird dessen **Click**-Ereignis ausgelöst, was dafür sorgt, dass das **MouseDown**-Ereignis nicht mehr feuert – und somit auch nicht die **MouseDown**-Ereignisse der in der Hierarchie über dem **Button**-Element liegenden Elemente. Dies gilt beispielsweise auch für das Textfeld.

Damit haben wir auch eine Erklärung, warum im Artikel **Drag and Drop** im Beispiel **Drag and Drop mit Dateien** die Bubbling Events nicht ausgeführt wurden und wir mit den Tunneling Events arbeiten mussten.

Elemente der Klasse RoutedEventArgs

Dem Kopf der Ereignismethoden können Sie als zweiten Parameter die Klasse **RoutedEventArgs** ausmachen. Diese liefert die folgenden Eigenschaften:

- **Handled**: Wenn Sie diesen Parameter auf den Wert **True** einstellen, wird das Routing der Ereignisse in der entsprechenden Ereignismethode unterbrochen. Ein Beispiel dazu finden Sie weiter unten.
- **RoutedEvent**: Liefert die **RoutedEvent**-Instanz.
- **Source**: Element, auf dem die Aktion ausgeführt wurde, die zum Routing führte.

- **OriginalSource**: Manchmal hat das Element aus **Source** noch Unterelemente, die über das **ControlTemplate** festgelegt werden. Dieses wird von **OriginalSource** ausgegeben.

Ereignis als behandelt markieren

Sie können, wie Sie oben gesehen haben, das gleiche Ereignis auf allen Ebenen als Tunneling- und Bubbling-Ereignis implementieren und es wird in der Regel auch ausgeführt (Ausnahme: wenn die Aktion auch das **Click**-Ereignis oder andere Ereignisse auslöst, welche das Auslösen der Bubbling-Ereignisse unterbindet). Gegebenenfalls wollen Sie aber, dass ein solches Ereignis nur für eine bestimmte Ebene feuert. Dann können Sie die Eigenschaft **Handled** der Klasse **RoutedEventArgs** auf **True** einstellen. Wenn Sie dies beispielsweise im obigen Beispiel für das Ereignis **PreviewMouseDown** des **Window**-Elements erledigen, sieht die Methode wie folgt aus (hier allerdings mit der Eigenschaft **Handled** der Klasse **MouseButtonEventArgs** und nicht von **RoutedEventArgs**):

```
private void Window_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("Window_PreviewMouseDown");
    e.Handled = true;
}
```

Wenn Sie dann noch auf eine freie Stelle neben der Schaltfläche klicken, wird auch tatsächlich nur noch die Ereignismethode **Window_PreviewMouseDown** ausgelöst – danach endet das Tunneling durch Setzen der Eigenschaft **Handled** auf den Wert **True**.

Welche Ereignisse werden nicht »gebubblet«?

Bleibt noch die Frage, welche Ereignisse von welchen Steuerelementen dafür sorgen, dass Bubbling Events nicht ausgelöst werden. Wir haben jetzt schon das **Click**-Ereignis entdeckt, welches beispielsweise das **MouseDown**- und das **MouseUp**-Ereignis schluckt. Oder das Überfahren des Zielelements beim Drag and Drop sorgt dafür, dass das **DragEnter**- oder das **DragOver**-Ereignis nicht ausgelöst werden – stattdessen müssen Sie die entsprechenden Tunneling-Ereignisse wie **PreviewDragEnter** oder **PreviewDragOver** nutzen.

Woran aber können wir – wenn nicht durch eigene Erfahrung – erkennen, wo es Sinn macht, die Bubbling Events zu implementieren und wo nicht? Genau genommen gibt es keine einheitliche Regel dafür. Beim **Click**-Ereignis einer Schaltfläche erscheint es noch logisch, beim Drag and Drop und dem **DragOver**-Ereignis nicht mehr.

Wozu Bubbling?

Nachdem wir uns angeschaut haben, wie Bubbling und Tunneling, auch Routed Events genannt, funktionieren, müssen wir uns die Frage stellen: Wozu das Ganze? Ein wichtiger Punkt ist, dass Sie mit WPF ja ganz einfach Elemente verschachteln können. So können Sie Schaltflächen definieren, denen Sie als Unterelemente ein StackPanel und darin ein Image- und ein Label-Element hinzufügen. Unter Access hätten wir ein Click-Ereignis für die Schaltfläche definiert, der wir als Eigenschaften das Bild und die Beschriftung hinzugefügt hätten. Hier ist dann klar: Der Mausklick löst das **Click**-Ereignis der Schaltfläche aus. Wenn wir allerdings explizit Elemente wie Image oder Label zu einem **Button**-Element hinzufügen, die wiederum **Click**-Ereignisse bereitstellen, für welches Element wollen wir dann das **Click**-Ereignis implementieren? Der Einfachheit halber doch wohl für das **Button**-Element. Aber wenn wir auf das **Image**- oder das **Label**-Element klicken, wie bekommt der Button dann etwa davon

Drag and Drop mit ListBox-Elementen

Im Artikel Drag and Drop-Grundlagen haben wir uns die grundlegenden Techniken für die Implementierung von Drag and Drop-Funktionen in WPF-Benutzeroberflächen angesehen. Nun gehen wir einen Schritt weiter und wollen Drag and Drop für das Fenster namens Versendungen unserer Beispielanwendung Bestellverwaltung umsetzen. Hier geht es dann nicht nur um einfaches Bewegen von Elementen per Maus, sondern auch um die Anpassung der dahinter stehenden Daten beziehungsweise Tabellen.

Wozu braucht man aber Drag and Drop so dringend? In der Tat lassen sich die meisten Tätigkeiten, die Sie damit ausführen können, auch auf andere Weise realisieren. Allerdings ist Drag and Drop und somit das einfache Ziehen von Elementen der Benutzeroberfläche mit der Maus doch ein sehr intuitiver Weg, um Aktionen durchzuführen. Ein Beispiel sind die beiden Listenfelder aus dem Artikel **m:n-Beziehung mit Listenfeld** (siehe Bild 1). Im Artikel haben wir Schaltflächen bereitgestellt, mit denen ein oder alle Artikel von einem Listenfeld zum anderen übertragen werden können. Nun wollen wir uns weiter vortasten und Drag and Drop-Funktionalität zu diesem Fenster hinzufügen. Zuvor schauen wir uns das jedoch an einigen grundlegenden Beispielen an.

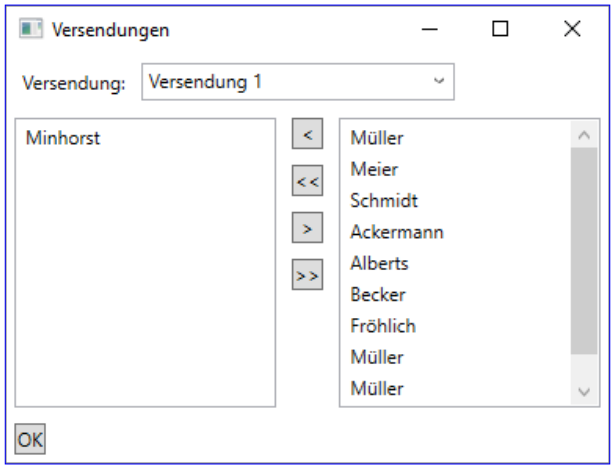


Bild 1: Für das Hin- und Herziehen von Einträgen zweier Listenfelder wäre Drag and Drop eine interessante Alternative.

Drag and Drop mit Listenfeldern

Nun schauen wir uns an, wie wir Drag and Drop in unser Beispiel mit den Versendungen einbauen. In der Beispielanwendung **Bestellverwaltung_SQLite** fügen wir dazu eine neue **Window**-Klasse namens **Versendungen_DragDrop** hinzu und kopieren den XAML-Code der Klasse **Versendungen** dort hinein. Passen Sie danach das **Window**-Element so an, dass die Klasse **Bestellverwaltung.Versendungen_DragDrop** referenziert wird und der Titel des Fensters angepasst ist:

```
<Window x:Class="Bestellverwaltung.Versendungen_DragDrop"
        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:Bestellverwaltung"
        mc:Ignorable="d"
        Title="Versendungen Drag and Drop" Height="300" Width="400">
```

Dann kopieren Sie auch den Code der Code behind-Klasse von **Versendungen.xaml.cs** in **Versendungen_DragDrop.xaml.cs**. Ändern Sie hier zunächst in der folgenden Zeile Versendungen in **Versendungen_DragDrop**:

```
public partial class Versendungen_DragDrop : Window, INotifyPropertyChanged {
```

Der gleiche Schritt ist auch nochmal in der ersten Zeile der Konstruktor-Methode nötig:

```
public Versendungen_DragDrop() {
```

Danach können wir die Funktionen für das Drag and Drop hinzufügen.

Drag and Drop-Vorgang starten

Wir beginnen mit dem linken **ListBox**-Element. Hier legen wir gar nicht erst die Methode für das Ereignis **MouseDown** an, von dem wir dank der Experimente im Artikel **Drag and Drop-Grundlagen** schon wissen, dass es nicht feuert. Stattdessen legen wir direkt das entsprechende Tunneling-Ereignis **PreviewMouseDown** an:

```
<ListBox x:Name="lstAusgewaehlteKunden" Grid.Column="0" Grid.Row="1" Margin="5"
    ItemsSource="{Binding AusgewaehlteKunden}"
    DisplayMemberPath="Nachname"
    SelectedValuePath="ID" MouseDoubleClick="lstAusgewaehlteKunden_MouseDoubleClick"
        PreviewMouseDown="lstAusgewaehlteKunden_PreviewMouseDown"/>
```

Dafür hinterlegen wir nun die folgende Ereignismethode:

```
private void lstAusgewaehlteKunden_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    if (e.LeftButton == MouseButtonState.Pressed) {
        DragDrop.DoDragDrop(lstAusgewaehlteKunden, lstAusgewaehlteKunden.SelectedItem, DragDropEffects.Copy);
    }
}
```

Wenn wir die Anwendung nun starten, das Fenster **Versendungen_DragDrop** aufrufen und einen der Einträge der linken List-Box ziehen wollen, erhalten wir den Fehler aus Bild 2.

Da der Fehler nicht direkt an Ort und Stelle gemeldet wird, müssen wir eine Fehlerbehandlung hinzufügen:

```
private void lstAusgewaehlteKunden_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    try {
        if (e.LeftButton == MouseButtonState.Pressed) {
            DragDrop.DoDragDrop(lstAusgewaehlteKunden, lstAusgewaehlteKunden.SelectedItem, DragDropEffects.Copy);
        }
    }
    catch (System.Exception ex) {
        MessageBox.Show(ex.Message);
        throw;
    }
}
```

Dies liefert dann die Meldung aus Bild 3. Offensichtlich versuchen wir auf irgendeine Weise, ein Element des Typs **Null** zu ziehen. Welches das ist, erfahren wir per Debugging.

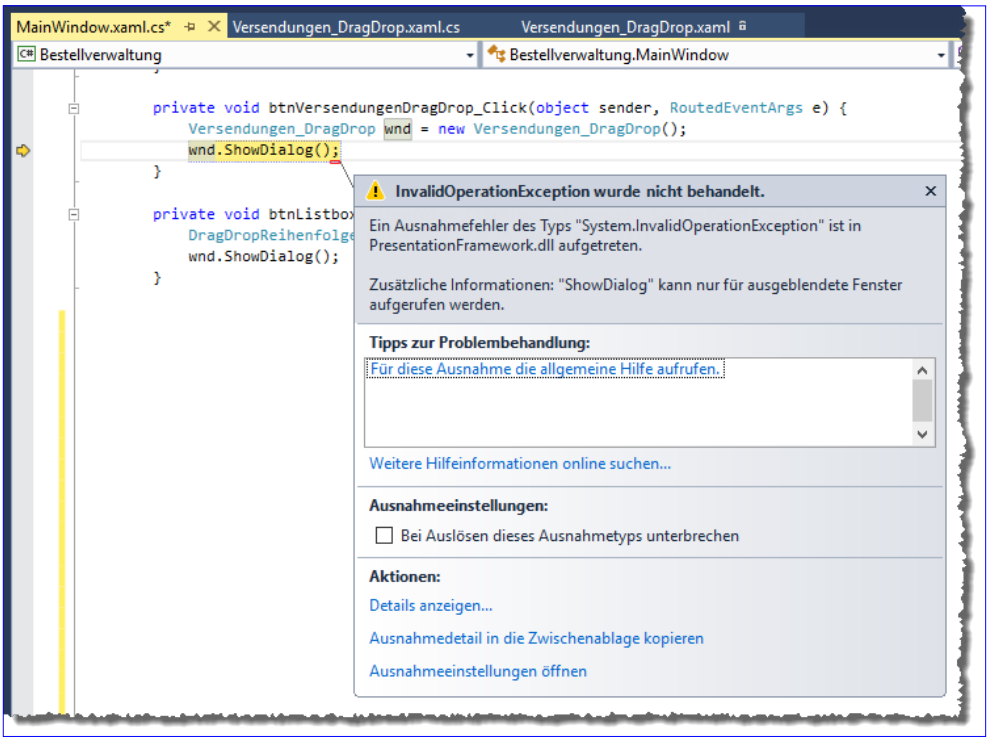


Bild 2: Fehler beim Versuch, ein Element aus der linken ListBox zu ziehen

Dazu setzen wir einen Haltepunkt in der ersten Zeile der Methode **lstAusgewaehlteKunden_PreviewMouseDown** und schauen uns den Inhalt der Parameter der Methode **DragDrop.DoDragDrop** an. Hier erfahren wir, dass die **SelectedItem**-Eigenschaft des **ListBox**-Elements den Wert **null** liefert (siehe Bild 4).

Wir müssen also offensichtlich einen ganz anderen Weg wählen, um beim Herunterdrücken der Maustaste Zugriff auf das dabei überfahrene Element zu bekommen. Dabei nutzen wir eine kleine Hilfsklasse, um das zu verschiebende Element zwischenzuspeichern:

```
public class DragDropData {
    public object ActualData { get; set; }
}
```

Dieses soll in der Eigenschaft **ActualData** das zu verschiebende Element speichern. Die Ereignismethode **lstAusgewaehlteKunden_PreviewMouseDown** ändern wir wie folgt ab:

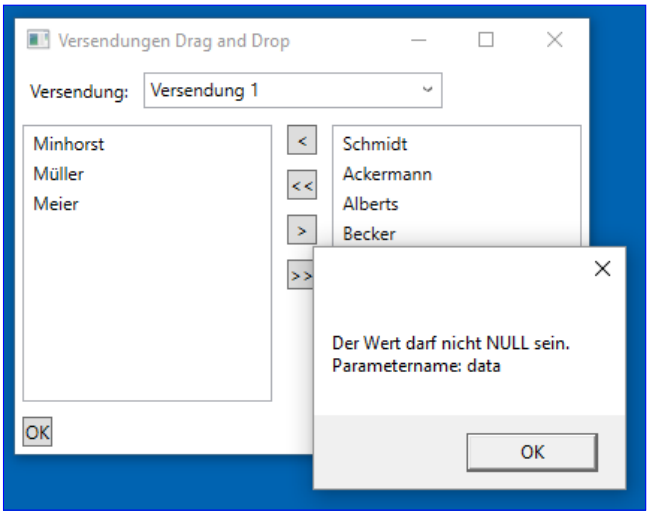


Bild 3: Der gleiche Fehler, diesmal mit Fehlerbehandlung

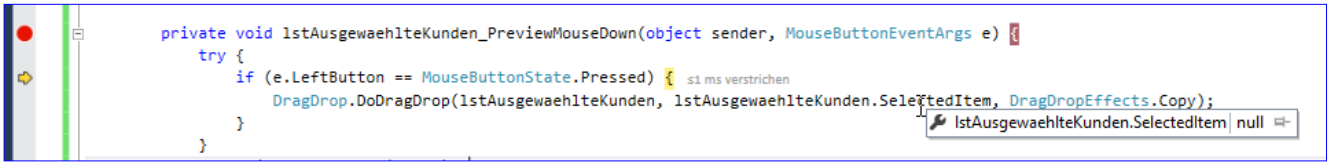


Bild 4: Der gleiche Fehler, diesmal mit Fehlerbehandlung

```
private void lstAusgewaehlteKunden_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    try {
        if (e.LeftButton == MouseButtonState.Pressed) {
            var item = (ListBoxItem)ItemsControl.ContainerFromElement(lst, (DependencyObject)e.OriginalSource);
            Kunde kunde = (Kunde)item.Content;
            DragDropData dragDropData = new DragDropData
            {
                ActualData = kunde,
            };
            DragDrop.DoDragDrop(lst, dragDropData, DragDropEffects.Move);
        }
    }
    catch (System.Exception ex) {
        MessageBox.Show(ex.Message);
        throw;
    }
}
```

Die Methode prüft zunächst, ob der Benutzer die linke Maustaste heruntergedrückt hat. In diesem Fall erstellt er ein neues Element des Typs **var** mit dem Namen **item**. Dies füllen wir mit dem **ListBoxItem**-Element, welches der Benutzer gerade in das andere Listenfeld ziehen möchte. Die Vorgehensweise, um an dieses Element zu kommen, ist nicht gerade intuitiv – zumindest nicht, wenn man noch nicht allzu tief im Objektmodell steckt: Wir ermitteln mit **e.OriginalSource** das Element, das der Benutzer mit der Maus ziehen möchte. In diesem Fall handelt es sich dabei etwa um ein **TextBlock**-Element, welches den Eintrag anzeigt. Dieses konvertieren wir in ein **DependendyObject**.

Dann ermitteln wir mit der **ContainerFromElement**-Methode der **ItemControl**-Klasse das Element der ListBox aus **lst**, welches das **TextBlock**-Element enthält – in diesem Fall das **ListBoxItem**-Element. Dieses konvertieren wir dann in ein **ListBoxItem**-Element. Der Inhalt des **ListBoxItem**-Elements sollte in unserem Fall ein Element des Typs **Kunde** sein, welches wir dann über die **Content**-Eigenschaft ermitteln, in **Kunde** konvertieren und in der Variablen **kunde** speichern. Dann erstellen wir eine neue Instanz unserer weiter oben vorgestellten Hilfsklasse **DragDropData** und weisen unser **Kunde**-Objekt **kunde** der einzigen Eigenschaft der Klasse zu. Damit können wir schließlich die **DoDragDrop**-Methode aufrufen und neben dem Verweis auf das **ListBox**-Element aus **lst** die Instanz der Klasse **DragDropData** übermitteln.

Wenn der Kunde nun das gezogene Element in der Ziel-ListBox abwirft, löst dies die **Drop**-Ereignismethode dieser ListBox aus. Diese sieht wie folgt aus:

```
private void lstAbgewaehlteKunden_Drop(object sender, DragEventArgs e) {
    DragDropData dragDropData = (DragDropData)e.Data.GetData(typeof(DragDropData));
    Kunde kunde = (Kunde)dragDropData.ActualData;
    KundeEntfernen(kunde);
}
```

Die Methode ist aufgrund unserer Vorarbeiten im Beispiel der beiden **ListBox**-Elemente recht kurz ausgefallen. Der Hauptgrund dafür ist, dass wir die Funktionen zum Entfernen eines Kunden aus der einen und zum Einfügen in die andere ListBox bereits programmiert haben und in Form einer Methode aufrufen können. Doch eins nach dem anderen.

Als Erstes füllen wir hier eine Variable namens **dragDropData** mit dem Inhalt der Eigenschaft **Data** des **DragEventArgs**-Parameter **e**, den wir über die **GetData**-Methode mit dem Typ **DragDropData** als Parameter erhalten. Den Inhalt von **dragDropData** finden wir dann in der Eigenschaft **ActualData**. Diesen lesen wir aus und speichern ihn in der Variablen **kunde** des Typs **Kunde**. Da wir nun wissen, welches **Kunde**-Element der Benutzer aus der einen ListBox entfernen und zur anderen ListBox hinzufügen möchte, können wir einfach die Methode **KundeEntfernen** auf und übergeben den Verweis auf das **Kunde**-Objekt als Parameter. Die im Artikel **m:n-Beziehung mit Listenfeld** vorgestellte Methode übernimmt dann die nötigen Schritte – das Verschieben des Elements vom einen in das andere Listenfeld sowie das Ändern der zugrunde liegenden Daten in der Datenbank.

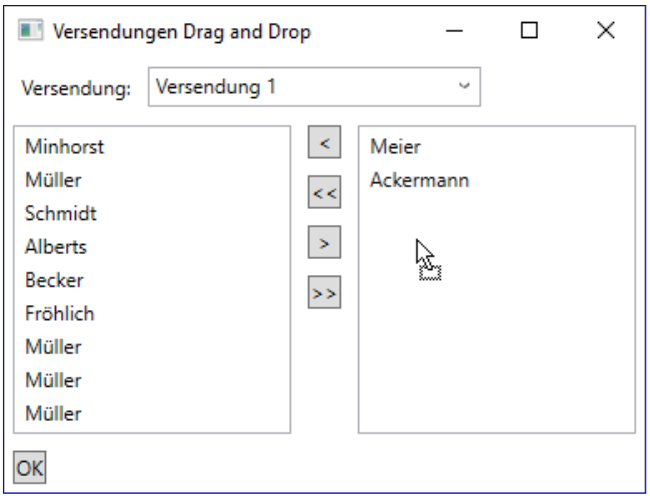


Bild 5: Drag and Drop mit dem richtigen Icon

Nun wollen wir noch dafür sorgen, dass die Ziel-Listbox nur ein Symbol anzeigt, das andeutet, dass hier ein Drop eines Elements möglich ist, wenn es sich bei dem auf das **ListBox**-Element bewegten Element tatsächlich um ein **Kunde**-Objekt handelt (siehe Bild 5). Diese Methode sieht wie folgt aus und prüft, ob **GetDataPresent** ein Element des Typs **DragDropData** enthält:

```
private void lstAbgewaehlteKunden_DragOver(object sender, DragEventArgs e) {
    if(e.Data.GetDataPresent(typeof(DragDropData))) {
        e.Effects = DragDropEffects.Move;
    }
    else {
        e.Effects = DragDropEffects.None;
    }
}
```

Drag and Drop in die andere Richtung

Nun implementieren wir die gleiche Prozedur noch in die andere Richtung, also um Elemente vom rechten in das linke **ListBox**-Element zu ziehen. Die ListBox **lstAbgewaehlteKunden**, also die rechte ListBox, erweitern wir wie folgt um das Attribut **PreviewMouseDown**:

```
<ListBox x:Name="lstAbgewaehlteKunden" ... AllowDrop="True" Drop="lstAbgewaehlteKunden_Drop"
DragOver="lstAbgewaehlteKunden_DragOver" PreviewMouseDown="lstAbgewaehlteKunden_PreviewMouseDown"/>
```

Excel-Export mit Spreadsheet Light

Das Thema Reporting haben wir im DATENBANKENTWICKLER noch gar nicht behandelt. Das liegt auch daran, dass es unter C#/WPF keine so einfach einsetzbare Reporting-Funktion wie etwa die Berichte und Access gibt. Also wollen wir uns einmal verschiedene Möglichkeiten ansehen, um die Daten einer Datenbank aus einer C#-Anwendung zu exportieren, um diese etwa in einer anderen Anwendung zu öffnen und auszudrucken. Den Start machen wir mit Microsoft Excel.

Voraussetzungen

In diesem Artikel wollen wir uns nicht, wie etwas unter Access/VBA möglich, per Automation an Excel heranwagen und die Excel-Dokumente per Excel erstellen. Stattdessen nutzen wir eine Bibliothek namens **Spreadsheet Light**, welche kostenlos verfügbar ist und Excel-Dateien erstellen kann, ohne dass Excel überhaupt auf dem Rechner existiert. Das ist natürlich gerade für Webanwendungen interessant, da so keine Office-Installation benötigt wird, aber auch für Desktop-Anwendungen.

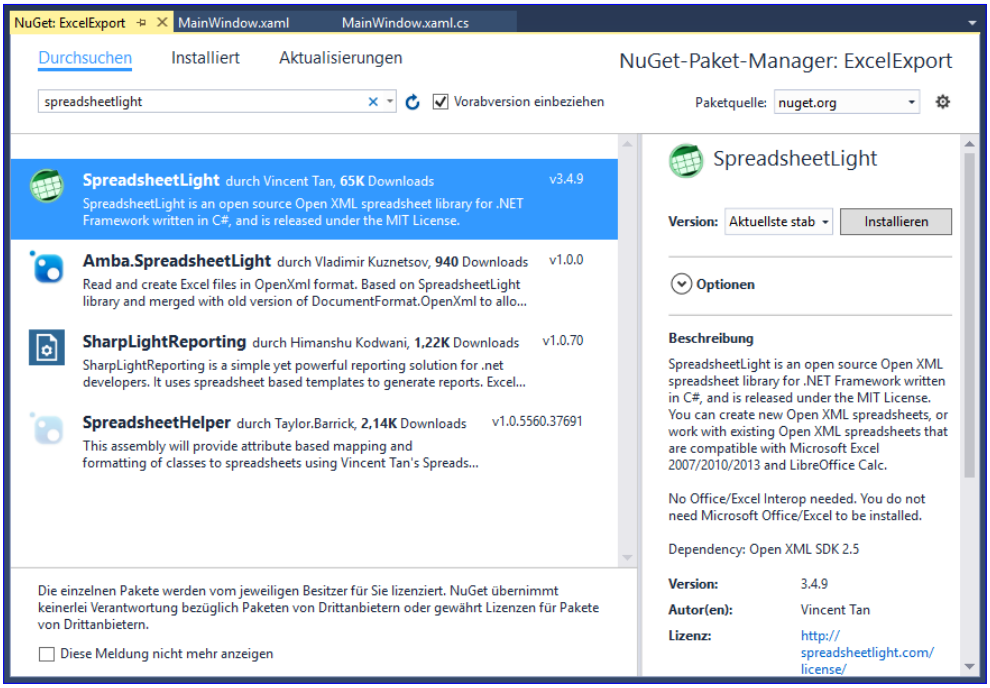


Bild 1: Hinzufügen von SpreadsheetLight über den Nuget-Paket-Manager

Spreadsheet light installieren

Spreadsheet light ist eine Open Source-Anwendung, die Sie kostenlos in Ihren Projekten dürfen. In den Beispielen verwenden wir ein C#/WPF-Projekt, dem Sie noch das entsprechende Paket plus Verweis hinzufügen müssen. Sie finden den Eintrag **SpreadsheetLight** im Nuget-Paket-Manager, den Sie über den Kontextmenü-Eintrag **Nuget-Pakete verwalten** des Projekts im Projektmappen-Explorer starten (siehe Bild 1).

Erste leere Excel-Datei erzeugen

Damit haben wir alle Voraussetzungen geschaffen. Ein erstes Beispiel wollen wir durch einen Mausklick auf eine neue Schaltfläche namens **btnDateiErstellen** ausführen. Diese fügen wir dem Fenster **MainWindow.xaml** hinzu und tragen für die Ereignisprozedur **btnDateiErstellen_Click** die folgenden beiden Befehle ein:

```
private void btnDateiErstellen_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    xls.SaveAs("LeereDatei.xlsx");
}
```

Der hier verwendete Typ **SLDocument** dürfte nun noch als fehlerhaft markiert werden. Das ist kein Problem – nachdem Sie den Verweis auf den Namespace **SpreadsheetLight** hinzugefügt haben, verschwindet die Markierung:

```
using SpreadsheetLight;
```

Wenn Sie das Projekt nun starten und die Schaltfläche betätigen, erstellt die Anwendung im Hintergrund ein erstes, leeres Excel-Dokument (siehe Bild 2). Das neue Dokument wurde übrigens im Verzeichnis der erstellte **.exe**-Datei erstellt, also beim Testen von Visual Studio aus etwa im Verzeichnis **bin\Debug** des Projektverzeichnisses.

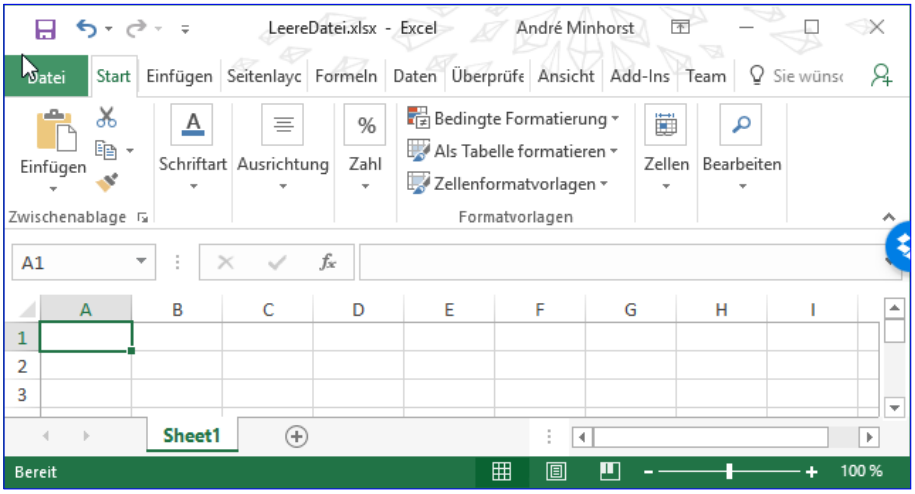


Bild 2: Die erste per SpreadsheetLight erstellt Excel-Datei

Wenn Sie Excel schon einmal von Access/VBA aus automatisiert haben, wissen Sie, dass Sie sonst einige Befehle mehr benötigen. Schauen wir uns nun an, wie wir die Excel-Datei füllen und formatieren können.

Daten zur Excel-Tabelle hinzufügen

Im zweiten Beispiel wollen wir eine Zelle der Tabelle mit einem Wert füllen, in diesem Fall einer Zeichenkette. Dazu verwenden wir die Methode **SetCellValue**, die als ersten Parameter die Angabe des Zellbezugs erwartet (hier **"A1"**) und als zweiten Parameter den einzufügenden Wert (**"BeispielText"**):

```
private void btnWertSchreiben_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    xls.SetCellValue("A1", "Beispieltext");
    xls.SaveAs("LeereDatei.xlsx");
}
```

Dabei markiert Visual Studio nun die Anweisung mit der Methode **SetCellValue** und liefert eine Fehlermeldung, nach der ein Verweis auf die Bibliothek **DocumentFormat.OpenXML** fehlt. Diese fügen Sie durch Auswahl des Menüeintrags **Projekt|Verweis hinzufügen...** und die Auswahl des Eintrags **DocumentFormat.OpenXML** im nun erscheinenden Dialog

Verweis-Manager hinzu (siehe Bild 3). Anschließend verschwindet auch diese Fehlermarkierung und Sie können das Einfügen des Zellinhalts nach dem Start des Debuggens und einem Klick auf die Schaltfläche **cmdWertSchreiben** ausprobieren.

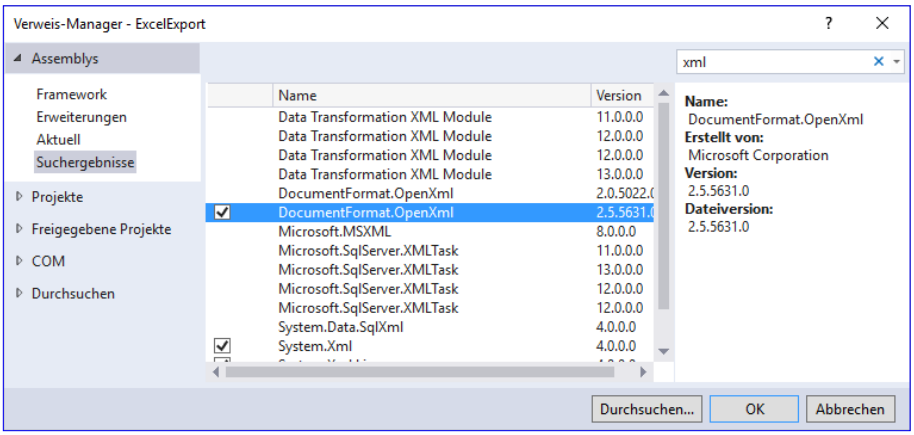


Bild 3: Verweis auf die Bibliothek **DocumentFormat.OpenXML**

Verschiedene Zellbezüge

Im ersten Beispiel haben wir als Zellbezug mit **A1** die klassische Variante gewählt. Excel zeigt ja auch die Zeilenüberschriften als Buchstaben und die Spaltenüberschriften als Zahlen an. Für die Programmierung ist dies indes nicht unbedingt immer hilfreich, vor allem, wenn Sie Daten innerhalb einer Schleife in die Excel-Tabelle schreiben wollen. Doch das ist unter **Spreadsheet Light** kein Problem: Die Methode **SetCellValue** bietet insgesamt 36 Überladungen. Warum so viele? Erstens, weil die Methode verschiedene Datentypen für den einzufügenden Wert entgegennehmen können soll, in diesem Fall **bool**, **byte**, **DateTime**, **decimal**, **InlineString**, **double**, **float**, **int**, **long**, **short**, **SLRstType**, **string, uint**, **ulong**, **ushort**, **DateTime** (mit **bool** für 1904-Epoche), **DateTime** (als **string**), **DateTime** (mit **bool** für 1904-Epoche als String). Das sind bereits 18 Varianten für den zweiten Parameter. Diese gibt es jeweils in Kombination mit dem **string**-Parameter **CellReference** (also beispielsweise **"A1"**) und mit den beiden **int**-Parametern **RowIndex** und **ColumnIndex** zur Angabe der zu füllenden Zelle als 1-basierten Index.

Wenn Sie also eine Matrix von 10x10 Zellen jeweils mit dem Produkt des Zellindexes füllen wollen, verwenden Sie die folgende Methode:

```
private void btnZehnMalZehn_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    for (int i = 1; i < 11; i++) {
        for (int j = 0; j < 11; j++) {
            xls.SetCellValue(i, j, i*j);
        }
    }
    xls.SaveAs("ZehnMalZehn.xlsx");
}
```

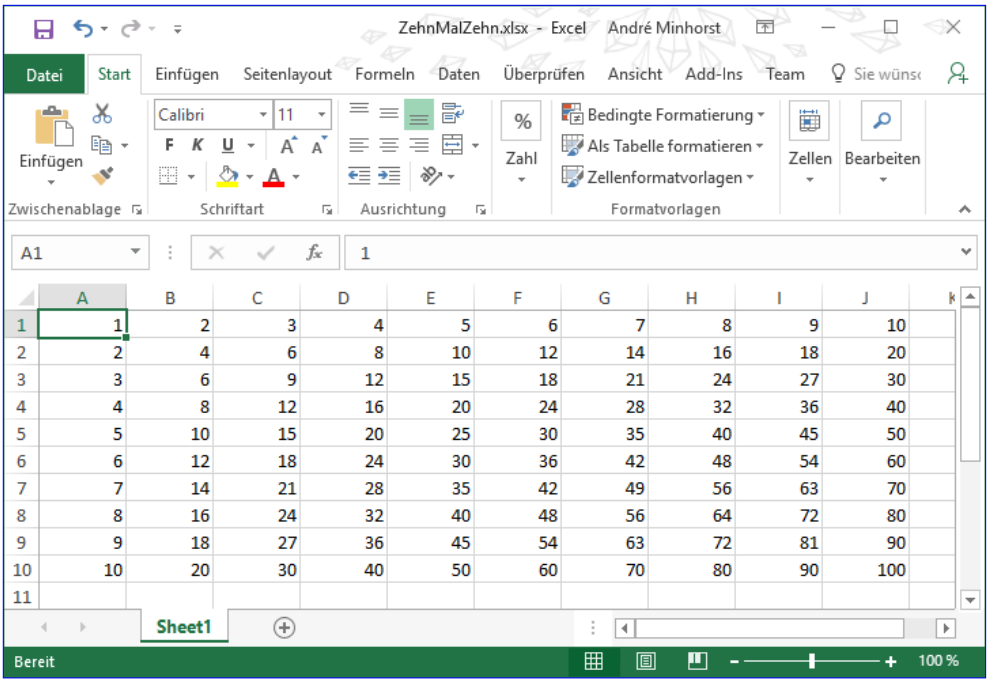
Diese Methode durchläuft zwei verschachtelte **for**-Schleifen mit dem Laufvariablen **i** und **j** und trägt jeweils das Produkt aus **i** und **j** in die Zelle mit dem entsprechenden Zeilen- und Spaltenindex ein (siehe Bild 4).

Zellen formatieren

Wollen wir nun ein Datum wie mit der folgenden Methode schreiben, erhalten wir nicht das erwartete Ergebnis:

```
private void btnDatumreihe_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    xls.SetCellValue(1, 1, new DateTime(2017,5,29));
    xls.SaveAs("DatumMitFormeln.xlsx");
}
```

Stattdessen erscheint in der Zelle der Wert **42884**, was der Anzahl der Tage seit dem 31.12.1899 entspricht. Im Gegensatz zu den bisher übergebenen Werten ist hier also eine Formatierung des Zellinhalts notwendig. Dies erledigen wir mit einer weiteren Methode namens **SetCellFormat**. Diese erwartet als ersten Parameter beziehungsweise für die ersten beiden Parameter die zu formatierende Zelle und als zweiten beziehungsweise dritten Parameter ein Objekt des Typs **SLStyle**. Dieses Objekt deklarieren und initialisieren wir zuerst und weisen seiner Eigenschaft **FormatCode** die Vorlage für das Format zu (in diesem Fall **"dd.mm.yyyy"**). Unsere Methode sieht nun wie folgt aus:



	A	B	C	D	E	F	G	H	I	J
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100
11										

Bild 4: Excel-Datei, die per Zeilen- und Spaltenindex gefüllt wurde

```
private void btnDatumreihe_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    xls.SetCellValue(1, 1, new DateTime(2017,5,29));
    SLStyle style = new SLStyle();
    style.FormatCode = "dd.mm.yyyy";
    xls.SetCellStyle (1,1, style);
    xls.SaveAs("DatumMitFormeln.xlsx");
}
```

Spaltenbreiten und Zeilenhöhen

Damit erhalten wir nun auch eine entsprechend formatierte Zelle, die das Datum korrekt anzeigt. Allerdings ist die Spalte nun zu schmal, um das Datum wie gewünscht anzuzeigen. Die Spaltenbreite stellen Sie für jede Spalte individuell ein, und zwar mit der Methode **SetColumnWidth**:

Bestellverwaltung á la Visual Basic

In einer Umfrage haben uns viele Leser bescheinigt, dass Sie viel besser mit Visual Basic-Code arbeiten würden anstatt den großen Schritt von VBA auf C# zu wagen. Also wollen wir uns in diesem Artikel einmal ansehen, wie es aussieht, wenn wir unsere in den bisherigen Ausgaben entwickelte Beispielanwendung auf VB umstellen. Eines vorweg: Wir müssen nicht den kompletten Code anfassen, denn die Benutzeroberfläche haben wir ja vollständig mit XAML beschrieben. Aber auch der Rest liefert noch eine Menge Arbeit, wie die folgenden Seiten zeigen werden ...

Die erste Frage, die sich stellt, ist die nach der richtigen Vorgehensweise. Das C#-Projekt, das wir umwandeln möchten, besteht aus Code behind-Klassen mit C#-Code, die zu den **Window**- und **Page**-Objekten gehören, Klassendateien auf Basis von C# (**.cs**) und Konfigurationsdateien, die mit Sicherheit an der einen oder anderen Stelle festlegen, in welcher Sprache das Projekt programmiert wurde. So ist die **AssemblyInfo.cs**-Datei beispielsweise auch in der jeweiligen Sprache programmiert. Beginnen wir bei der **.sln**-Datei, welche die Lösung zusammensetzt – in unserem Fall ja nur aus dem reinen Anwendungsprojekt. Die dort referenzierte Projektdatei hat die Dateiendung **.csproj**, womit die C#-Affinität auch schon ausgeschöpft ist:

```
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Bestellverwaltung", "Bestellverwaltung\Bestellverwaltung.csproj",
"{FCD792BB-5FC8-4BC1-A66B-4165B450B126}"
```

In der **.csproj**-Datei finden sich beispielsweise die Referenzen, wozu auch eine auf die C#-Bibliothek gehört:

```
<Reference Include="Microsoft.CSharp" />
```

Danach folgen die Einträge für die einzelnen Klassen (**.xaml.cs**) und die dazugehörigen **.xaml**-Dateien sowie die übrigen **.cs**-Dateien, die zum Beispiel wie folgt aussehen:

```
<Compile Include="Kundenuebersicht.xaml.cs">
    <DependentUpon>Kundenuebersicht.xaml</DependentUpon>
</Compile>
```

Können wir nun also die entsprechenden Stellen in diesen Dateien ändern, die **.cs**-Dateien durch **.vb**-Dateien ersetzen und die Anwendung starten? So einfach ist es leider nicht. Wir finden beispielsweise keine Möglichkeit, etwa eine **.vb**-Klasse zum C#-Projekt hinzuzufügen. Selbst wenn wir eine neue VB-Klasse außerhalb des Projekts erstellen und es dann per Drag and Drop in den Projektmappen-Explorer zum Projekt hinzufügen, wird es zwar automatisch in der **.csproj**-Datei referenziert. Die darin enthaltene Klasse kann jedoch von den anderen C#-Klassen aus nicht referenziert werden.

Aus C# mach VB

Also legen wir ein neues VB-Projekt namens **Bestellverwaltung_VB** an. Dazu wählen Sie im Dialog **Neues Projekt** die Vorlage **Visual Basic\Windows\WPF-Anwendung** aus (siehe Bild 1).

Das VB-Projekt ist ähnlich aufgebaut wie ein C#-Projekt auf Basis der Vorlage WPF-Anwendung. Wir bekommen gleich die Klasse **MainWindow** zu Gesicht – allerdings hat die Code behind-Klasse diesmal mit der Dateiendung **.xaml.vb**.

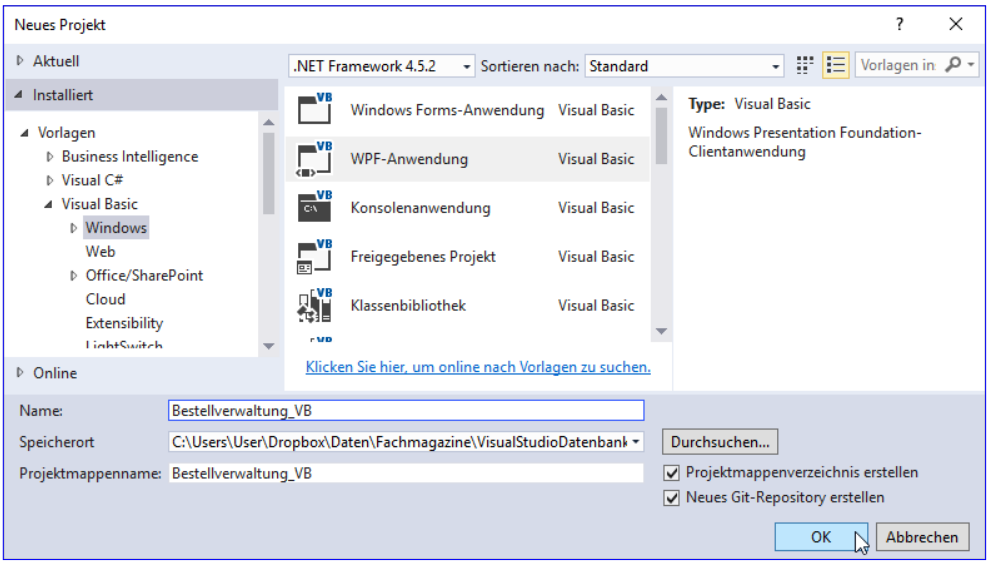


Bild 1: Anlegen des VB-Projekts

Konverter

Hier setzen wir direkt an und wollen die Code behind-Klasse des Fensters **MainWindow** der C#-Anwendung, also **MainWindow.xaml.cs**, in die Datei **MainWindow.xaml.vb** übertragen. Dazu verwenden wir einen automatischen Code-Konverter wie den auf der Webseite **converter.telerik.com**. Hier wählen Sie die gewünschte Übersetzung aus, hier also etwa C# to VB, und kopieren den zu übersetzenden Code in das linke Fenster. Nach einem Klick auf die Schaltfläche **Convert Code** wird dieser in das rechte Fenster konvertiert (siehe Bild 2).

Wie dies gelungen ist, wird sich zeigen, wenn wir nach und nach alle Elemente übersetzt haben. Dazu kopieren wir zunächst den Inhalt des rechten Fensters in die Klasse **MainWindow.xaml.vb**. Das liefert einige Fehlermeldungen, die zum größten Teil daraus resultieren, dass die dort referenzierten Klassen nicht vorhanden sind.

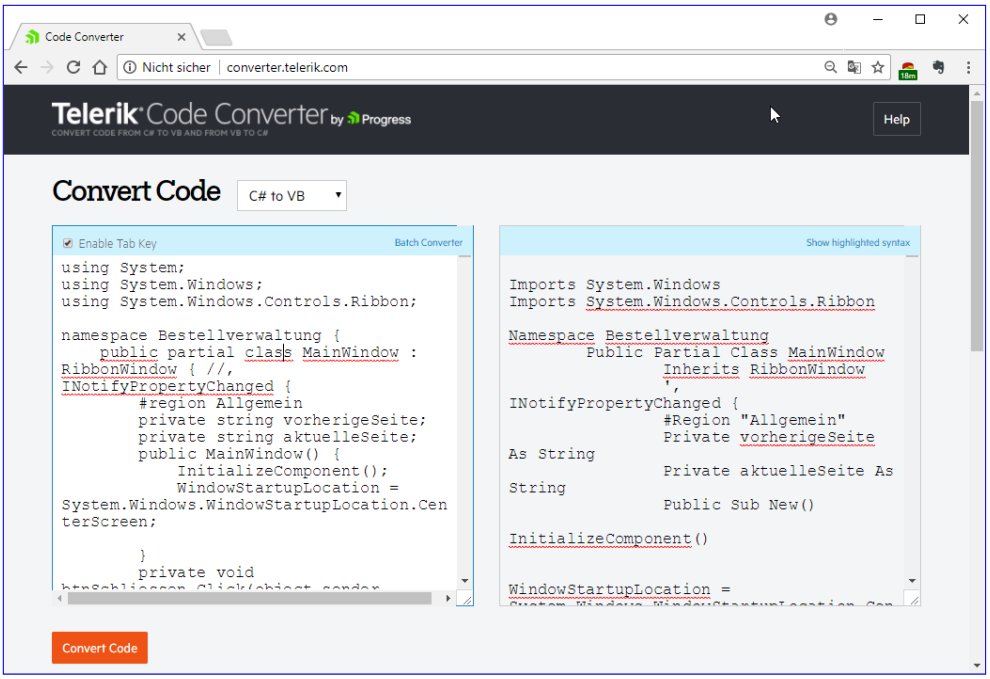


Bild 2: Automatisches Übersetzen des VB-Codes

Die erste können wir bereits beheben, indem wir einen Verweis auf die Klasse **System.Windows.Controls.Ribbon** hinzufügen, und zwar über den Dialog **Verweis-Manager**, den wir über den Menüeintrag **Projekt|Verweise** öffnen (siehe Bild 4).

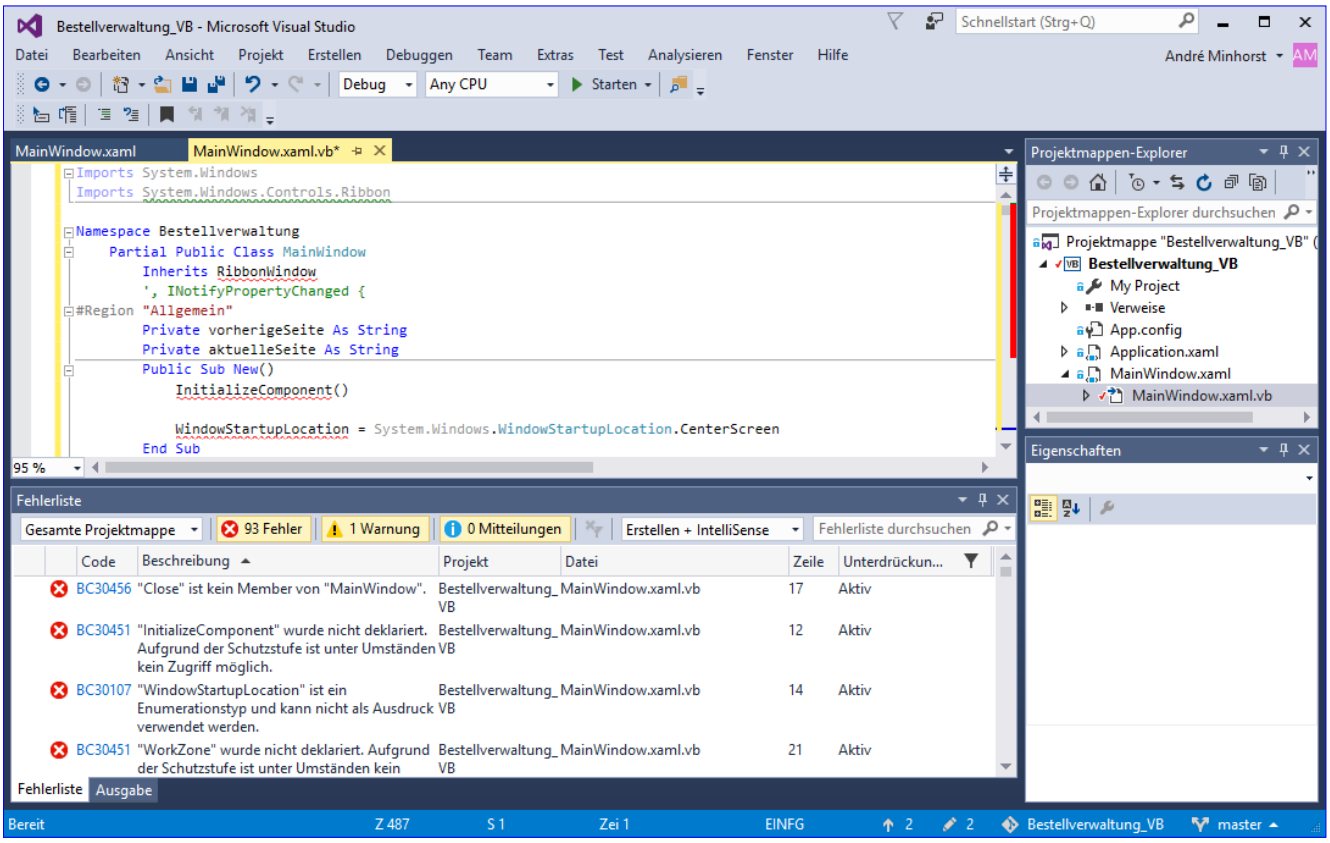


Bild 3: Beim Einfügen erscheinen einige Fehlermeldungen ...

Code temporär auskommentieren

Anschließend kommentieren wir fast den kompletten Code erst einmal aus. Dazu markieren Sie die auszukommentierenden Teile und betätigen die Tastenkombination **Strg + K, C**. Wir lassen nur die folgenden Zeilen übrig, um von dort aus Stück für Stück die entsprechenden Teile wieder in die Klasse einzubinden:

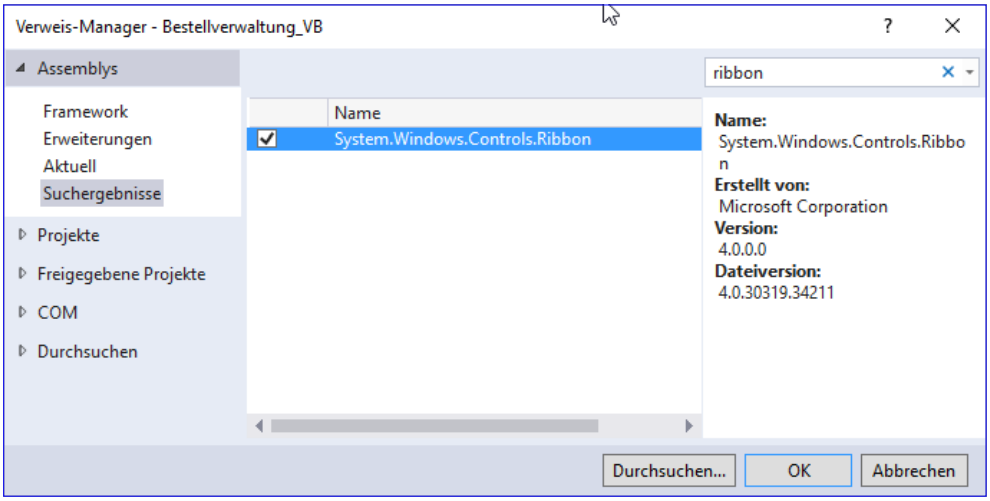


Bild 4: Verweis auf die Ribbon-Klasse hinzufügen

```
Imports System.Windows.Controls.Ribbon

Namespace Bestellverwaltung
```

```
    Partial Public Class MainWindow
        Inherits RibbonWindow
#Region "Allgemein"
        Private vorherigeSeite As String
        Private aktuelleSeite As String
        Public Sub New()
            InitializeComponent()
            WindowStartupLocation = System.Windows.WindowStartupLocation.CenterScreen
        End Sub
#End Region

    End Class
End Namespace
```

An dieser Stelle erhalten wir nur noch eine Fehlermeldung, und zwar für die Anweisung **InitializeComponent()**. Diese entfernen wir zunächst einfach.

XAML-Code kopieren

Der XAML-Code sollte keine Referenz auf die als Code behind-Datei verwendete **.cs**-Datei haben, sondern nur auf die Klasse beziehungsweise den Namespace. Also kopieren Sie einfach den kompletten Inhalt der Datei **MainWindow.xaml** des C#-Projekts in die gleichnamige Datei des neu erstellten VB-Projekts. Das Ergebnis sieht schon recht gut aus. Wir müssen allerdings noch die Bilddateien nachreichen, denn diese liegen ja noch nicht im Zielprojekt vor (siehe Bild 5). Um dies zu erledigen, würden wir nun am lieben einfach den Ordner **images** aus dem Projektmappen-Explorer des C#-Projekts in den des VB-Projekts ziehen. Das gelingt allerdings nicht.

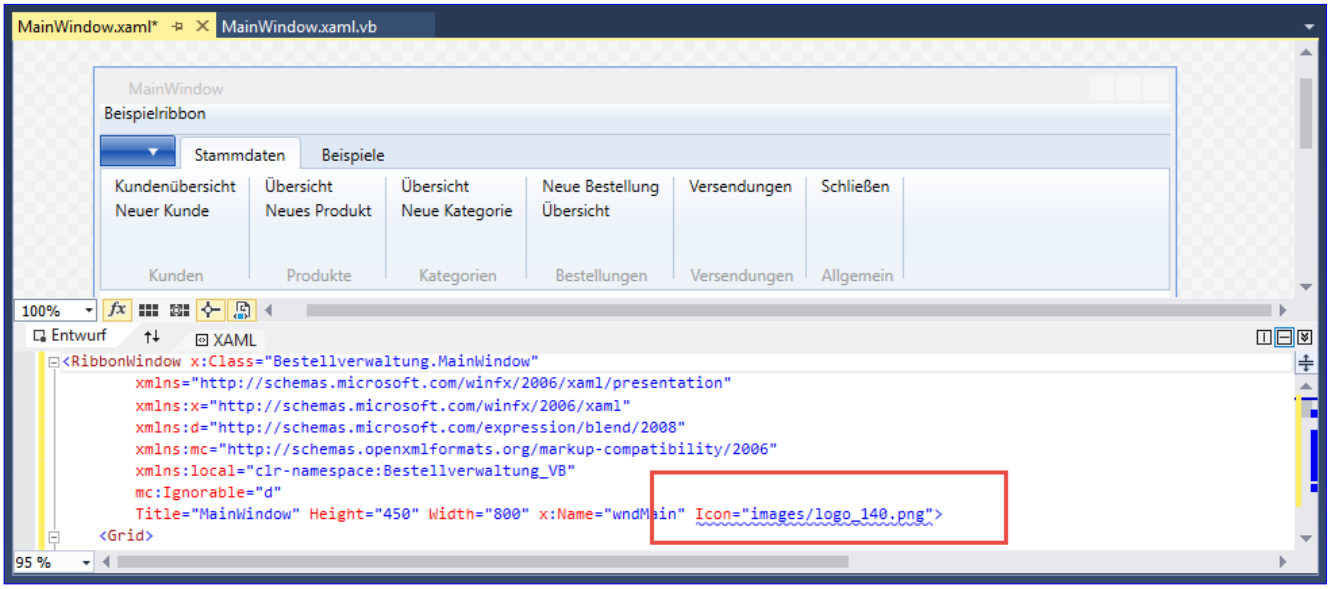


Bild 5: Die Bilddateien fehlen noch.

Bilder hinzufügen

Also ziehen wir einfach den Ordner **images** aus dem Windows Explorer in den Projektmappen-Explorer des VB-Projekts. Das gelingt, allerdings erscheinen die enthaltenen Bilddateien dort nicht! Allerdings wurde das Verzeichnis **images** im Projektordner im Dateisystem mit den enthaltenen Dateien angelegt. Deshalb ziehen wir die Bilder noch aus dem **images**-Verzeichnisses im Projektordner des VB-Projekts in den Ordner **images** im Projektmappen-Explorer. Danach erscheinen dort nicht nur die Bilder, sondern auch im Ribbon (siehe Bild 6).

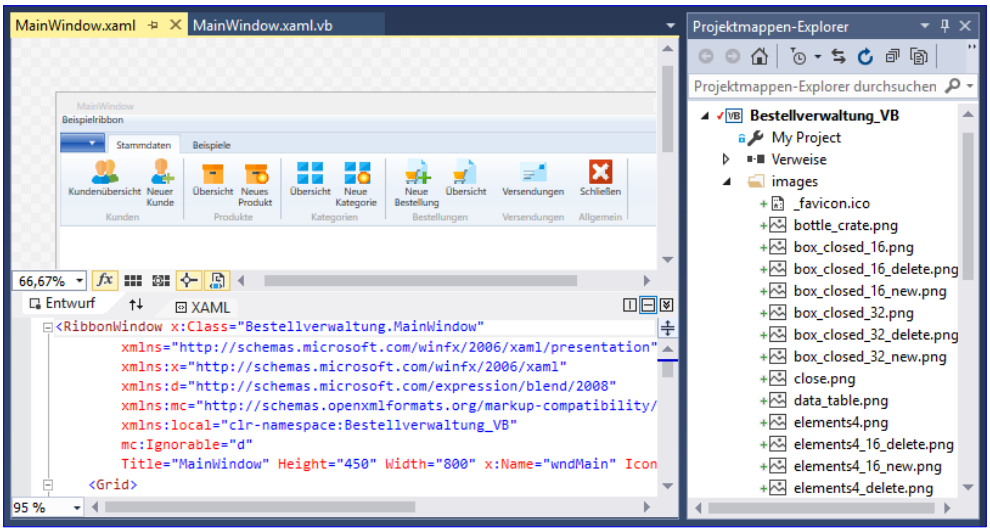


Bild 6: Anlegen der fehlenden Bilddateien

Versendungen-Fenster

Als Erstes nehmen wir nun das Fenster hinein, welches durch einen Klick auf die Ribbon-Schaltfläche **btnVersendungen** angezeigt wird. Dazu nehmen wir die Kommentarzeichen für die folgenden Zeilen in der Klasse **MainWindow.xaml.vb** wieder weg:

```
Private Sub btnVersendungen_Click(sender As Object, e As RoutedEventArgs)
    Dim wnd As New Versendungen()
    wnd.ShowDialog()
End Sub
```

Hier wird die Klasse **Versendungen** rot markiert, da diese ja noch gar nicht im Projekt enthalten ist. Also holen wird dies nach, indem wir eine neue **Window**-Klasse namens **Versendungen** anlegen. Dann kopieren wir den kompletten XAML-Code aus der Klasse **Versendungen.xaml** des C#-Projekts und fügen diesen in die neu erstellte Klasse im VB-Projekt ein. Sie brauchen hier nur den Namespace für das Attribut **xmlns:local** des **Window**-Elements auf **Bestellverwaltung_VB** zu ändern.

Dann kopieren wir auch hier einfach den Inhalt der Klasse **Versendungen.xaml.cs** in das linke Fenster der Webseite **http://converter.telerik.com/** und erzeugen mit einem Klick auf **Convert Code** den entsprechenden VB-Code. Daraus resultieren wiederum einige rote Markierungen und entsprechende Einträge in der Fehlerliste von Visual Studio (siehe Bild 7). Die meisten davon beziehen sich auf fehlende Objekte wie **Versendung**, **Kunde** oder **KundeVersendung**. Spätestens, wenn wir das nicht definierte Objekt **BestellverwaltungEntities** entdecken, wird dann klar, dass wir ja noch gar nicht das Entity Data Model zur Datenbank hinzugefügt haben.

Entity Data Model hinzufügen

Die zum Erstellen des Entity Data Model notwendigen Schritte in aller Kürze:

Drag and Drop-Grundlagen

Unter Access fehlen einige Features, die in anderen Programmiersprachen und Entwicklungsumgebungen zum guten Ton gehören. Eines davon ist die Drag and Drop-Funktionalität, die sich nur aufwendig abbilden ließ – und auch nur mit bestimmten ActiveX-Steuerelementen. Die eingebauten Steuerelemente wie Textfelder oder Listenfelder ließen leider kein natives Drag and Drop zu. Unter WPF und den .NET-Programmiersprachen sieht das ganz anders aus. Dieser Artikel liefert Grundlagen zu Drag and Drop.

Wozu braucht man aber Drag and Drop? In der Tat lassen sich die meisten Tätigkeiten, die Sie damit ausführen können, auch auf andere Weise realisieren. Allerdings ist Drag and Drop und somit das einfache Ziehen von Elementen der Benutzeroberfläche mit der Maus doch ein sehr intuitiver Weg, um Aktionen durchzuführen. Ein Beispiel sind die beiden Listenfelder aus dem Artikel **m:n-Beziehung mit Listenfeld** (siehe Bild 1). Im Artikel haben wir Schaltflächen bereitgestellt, mit denen ein oder alle Artikel von einem Listenfeld zum anderen übertragen werden können. Nun wollen wir uns weiter vortasten und Drag and Drop-Funktionalität zu diesem Fenster hinzufügen. Bevor wir uns das im Artikel **Drag And Drog mit ListBox**-Elementen ansehen, schauen wir uns einige grundlegende Beispielen an.

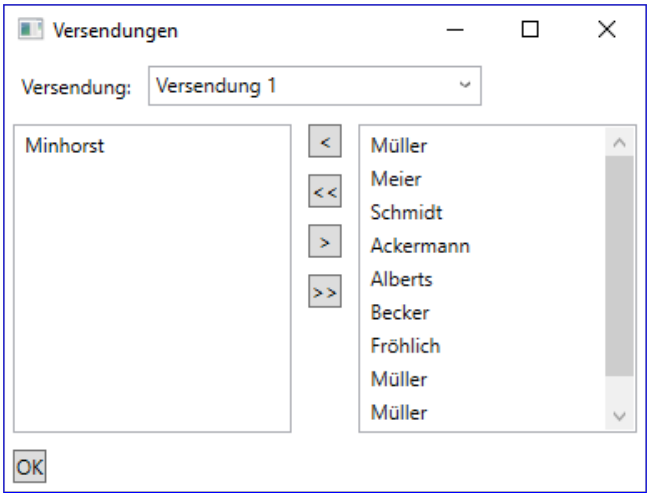


Bild 1: Für das Hin- und Herziehen von Einträgen zweier Listenfelder wäre Drag and Drop eine interessante Alternative.

Was ist Drag and Drop?

Drag and Drop nennen wir den Vorgang, bei dem der Benutzer ein Element der Benutzeroberfläche mit der linken Maustaste anklickt und diese dabei gedrückt hält. Dann wird die Maus mit dem Element zum Zielelement bewegt und das zu bewegende Element dort durch Loslassen der linken Maustaste an das Zielelement übergeben.

Dabei müssen Start- und Zielelement gar nicht unbedingt zwei verschiedene Elemente sein wie beim Beispiel mit den Listenfeldern. Es kann auch nur ein einziges Listenfeld sein, dessen Einträge Sie per Drag and Drop umsortieren wollen. Oder Sie bewegen die Elemente in einem **TreeView**-Steuerelement von einem übergeordneten Element zum nächsten.

Dazu benötigen wir ein paar Informationen. Zunächst müssen wir für das abgebende Element festlegen, dass es das Ziehen der enthaltenen Elemente mit der Maus überhaupt erlaubt. Dann wollen wir uns irgendwie merken, welches Element wir mit der Maus angepackt haben, um es zu verschieben (oder auch zu kopieren). Schließlich soll das aufnehmende Element mit der Möglichkeit zum Aufnehmen des zu übergebenden Elements ausgestattet werden, also als Drag and Drop-Ziel verfügbar gemacht werden. Da man unter Windows eine ganze Menge Dinge durch die Gegend ziehen kann – zum Beispiel von Windows Explorer auf verschiedene andere Ziele – müssen Sie dem Zielelement auch noch mitteilen, auf welche zu übergebenden Elemente es überhaupt reagieren soll, indem es das Symbol als mögliches Zielelement einblendet. Schließlich benötigen wir verschiedene

Ereignisse, für die wir passende Methoden implementieren, um das Verhalten von Drag and Drop-Quelle und -Ziel festzulegen.

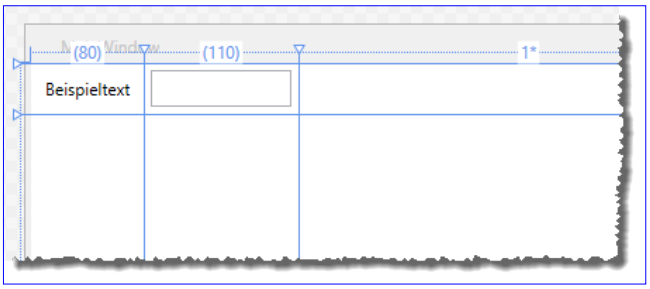


Bild 2: Label und TextBox als erstes Drag and Drop-Beispiel

Drag and Drop ist auch eine Alternative zum Kopieren/Ausschneiden und Einfügen. Der wesentliche Unterschied neben den dazu notwendigen Techniken (Ziehen mit der Maus bei Drag and Drop auf der einen, Tastenkombinationen wie **Strg + X**, **V** oder **C** und Kontextmenü-Einträge auf der anderen Seite) ist der Ort, an dem wir uns merken, welches Element von A nach B bewegt werden sollen. Beim Kopieren/Ausschneiden und Einfügen ist dies die Zwischenablage. Bei Drag and Drop benötigen wir eine andere Möglichkeit, in diesem Fall ein Element des Typs **DataObject**.

Einfaches Beispiel: Bezeichnungsfeld in Textfeld ziehen

Wir schauen uns zu Beginn ein einfaches Beispiel an, bei dem wir die Beschriftung eines Bezeichnungsfeldes einfach in ein Textfeld ziehen wollen. Dazu legen wir entsprechende Steuerelemente in einem neuen Fenster namens **MainWindow.xaml** eines neuen Projekts an (s. Bild 2). Die beiden Steuerelemente definieren wir wie wie folgt:

```
<TextBox x:Name="txtDrop" Grid.Column="1" Grid.Row="1" Margin="5" Drop="txtDrop_Drop" Width="100" />
<Label x:Name="lblDrag" Content="Beispieltext" Grid.Column="0" Grid.Row="1" Margin="5" MouseDown="lblDrag_MouseDown" />
```

Unter Visual Basic sieht die Ereignismethode für das Ereignis **MouseDown** des Bezeichnungsfeldes wie folgt aus:

```
'Visual Basic
Private Sub lblDrag_MouseDown(sender As Object, e As MouseButtonEventArgs)
    DragDrop.DoDragDrop(lblDrag, lblDrag.Content, DragDropEffects.Copy)
End Sub
```

Wir benötigen also nur eine einzige Zeile, um den Vorgang zu starten! Dabei rufen wir die Methode **DoDragDrop** der Klasse **DragDrop** auf. Diese erwartet drei Parameter:

- **dragSource**: Quelle des Drag and Drop-Vorgangs, also beispielsweise ein Steuerelement wie in diesem Fall das Bezeichnungsfeld
- **data**: Die per Drag and Drop bewegten Daten werden mit dem Parameter **data** in einen Zwischenspeicher ähnlich der Zwischenablage gegeben und können dort beim Drop wieder abgerufen werden.
- **allowedEffects**: Effekte sind verschiedene Darstellungsarten des Mauszeigers.

Im vorliegenden Fall übergeben wir für den ersten Parameter einen Verweis auf das Bezeichnungsfeld **lblLabel**. Der zweite Parameter nimmt den Inhalt der Eigenschaft **Content** des Bezeichnungsfeldes auf, hier eine einfache Beschriftung. Da es sich

hierbei um ein reines Kopieren von Daten handelt, nämlich das Kopieren der Beschriftung des Bezeichnungsfeldes in das Textfeld, soll nur das Symbol für das Kopieren mit der Maus angezeigt werden (siehe Bild 3).

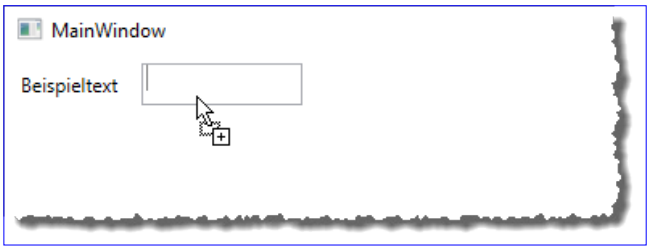


Bild 3: Veränderter Mauszeiger beim Drag and Drop

Damit kommen wir Fallenlassen des zu kopierenden Inhalts über dem Zielobjekt. Hier verwenden wir nun das Ereignis Drop des Textfeldes, das wir wie folgt implementieren:

```
'Visual Basic
Private Sub txtDrop_Drop(sender As Object, e As DragEventArgs)
    Dim str As String
    str = e.Data.GetData(DataFormats.StringFormat)
    txtDrop.Text = str
End Sub
```

Die Methode liefert die gewohnten beiden Parameter, wobei der zweite den Typ **DragEventArgs** hat (mehr dazu weiter unten). In diesem Fall nutzen wir das Objekt **data** der mit dem Parameter e übergebenen **DragEventArgs**-Klasse, dessen Methode **GetData** mit dem Parameter **DataFormats.StringFormat** den gespeicherten Text zurückgibt. Diesen speichern wir in der String-Variablen **str** und weisen diese dann der **Text**-Eigenschaft des Textfeldes zu. Das Ergebnis sieht schließlich wie in Bild 4 aus.

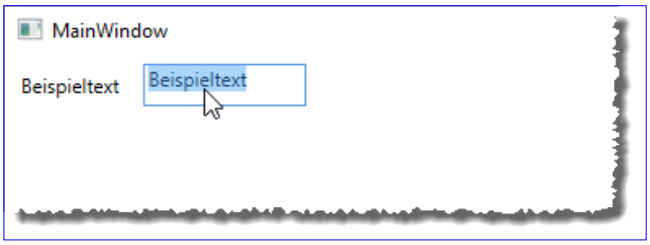


Bild 4: Erfolgreicher Drag and Drop-Vorgang

Zusammengefasst haben wir für den Start von Drag and Drop das Ereignis **MouseDown** genutzt, dort den Drag and Drop-Vorgang gestartet und dadurch auch das Ereignis **Drop** aktiviert. Ohne den Aufruf von **DoDragDrop** können Sie nämlich mit der Maus ziehen und fallenlassen, was sie möchten – das **Drop**-Ereignis wird nicht ausgelöst.

Verfeinerung: Maustaste prüfen

Im aktuellen Zustand können Sie auch mit der rechten Maustaste den Drag and Drop-Vorgang ausführen. Das liegt daran, dass wir in der Methode **lblDrag_MouseOver** nicht geprüft haben, welche Maustaste gerade gedrückt wurde. Dies holen wir nun nach und passen die Methode wie folgt an:

```
Private Sub lblDrag_MouseDown(sender As Object, e As MouseButtonEventArgs)
    If e.LeftButton = MouseButtonState.Pressed Then
        DragDrop.DoDragDrop(lblDrag, lblDrag.Content, DragDropEffects.Copy)
    End If
End Sub
```

Hier prüfen wir vorab, ob die linke Maustaste gedrückt ist. Damit wird das Drag and Drop mit gedrückter rechter Maustaste unterbunden.

Drag and Drop aus anderen Anwendungen

Wenn Sie einen Text aus einer anderen Anwendung in ein Textfeld einer WPF-Anwendung ziehen wollen, klappt das mit der gegebenen Konfiguration ebenfalls. Sie können nun beispielsweise Text aus einem Text-Editor oder aus Word markieren und diesen dann in das Textfeld ziehen. Aber: Dazu wären die bisherigen Schritte gar nicht nötig gewesen – Sie können auch einfach so den Text aus einer externen Anwendung in das Textfeld ziehen. Dazu müssen Sie noch nicht einmal ein Attribut des Textfeldes anpassen. Interessant ist noch die Frage: Feuert denn das Drop-Ereignis des Textfeldes, wenn Text aus einer externen Anwendung abgelegt wird? Die Antwort lautet nein.

Drag and Drop in andere Anwendungen

Bei Texten gelingt das Drag and Drop aus einem Textfeld in eine andere Anwendung, welche als Drop-Ziel ausgelegt ist wie etwa Microsoft Word, ohne weiteres Zutun.

Auch wenn man einen Drag and Drop-Vorgang in der WPF-Anwendung so beginnt, dass der zu verschiebende Inhalt in die Drag and Drop-Zwischenablage übertragen wird, können Sie den Inhalt etwa in eine Textverarbeitung übertragen. Auf diese Weise können Sie etwa den Text der Schaltfläche aus dem ersten Beispiel in eine Textverarbeitung statt in das Textfeld ziehen.

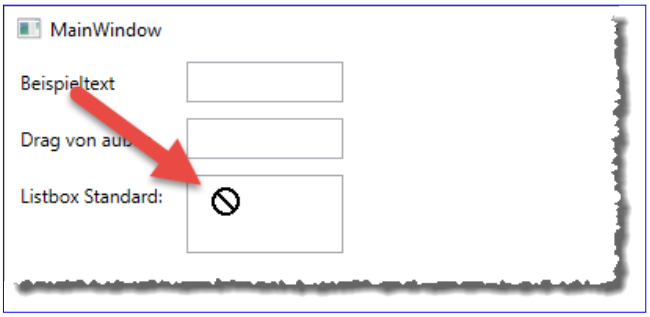


Bild 5: Drag and Drop auf eine Standard-ListBox

Drag and Drop in eine ListBox

Textfelder sind also immer für Drag and Drop empfänglich, wenn es sich um Daten im passenden Typ handelt, also um Texte. Was aber, wenn wir einen Text auf ein anderes Steuerelement droppen – beispielsweise ein **ListBox**-Steuerelement? Dazu fügen wir unserem Beispielformular eine ListBox hinzu und nennen diese **lstDrop**:

```
<ListBox x:Name="lstDropStandard" Grid.Column="1" Grid.Row="3" Margin="5" Width="100" Height="50" />
```

Wenn wir den Text des Bezeichnungsfeldes **lblDrag** auf das **ListBox**-Element ziehen, wird der Mauszeiger in ein Symbol umgewandelt, dass deutlich macht, dass das **ListBox**-Element kein valides Ziel für die Drag and Drop-Anweisung ist (siehe Bild 5). Also stellen wir nun das Attribut **AllowDrop** auf **True** ein:

```
<ListBox x:Name="lstDropStandard" ... AllowDrop="True" />
```

Damit erscheint schon einmal das richtige Symbol, wenn Sie das **ListBox**-Element beim Drag and Drop mit der Maus überfahren. Beim Fallenlassen geschieht jedoch nichts. Kein Wunder: Ein **ListBox**-Element nimmt seine Inhalte ja auch über ganz andere Wege entgegen – zum Beispiel über die **Add**-Methode der **Items**-Auflistung. Diese implementieren wir nun für die **Drop**-Methode des **ListBox**-Element, das wir dazu als neues Element wie folgt erweitern: