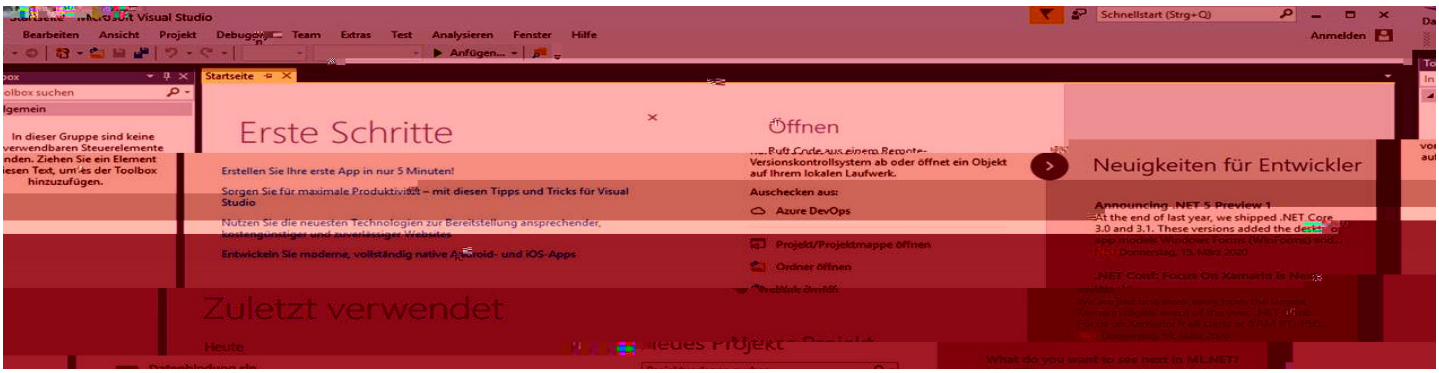
AUSGABE 04-05/2020

DATENBANK ENTWICKLER

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



TOP-THEMEN:

André Minhorst Verlag

Entity Framework: Aktionsabfragen mit LINQ

Wer lange mit Access gearbeitet hat, schreibt Aktionsabfragen entweder mit der Entwurfsansicht für Abfragen oder erstellt diese per SQL-Quellcode. Vielleicht kombinieren Sie auch beide Varianten und passen per Entwurf erstellte Abfragen in der SQL-Ansicht an. Unter VBA verwendet man außerdem oft DAO, um Datensatzgruppen zu öffnen und Daten anzulegen, zu bearbeiten oder zu löschen. In einem Entity Data Model können Sie auch SQL-Abfragen absetzen, sowohl für Auswahl- als auch für Aktionsabfragen. Allerdings gibt es auch LINQ – die Abfragesprache für das Entity Framework. Dieser Artikel zeigt, wie Sie gängige SQL-Aktionsabfragen oder DAO-Aktionen in LINQ übersetzen.

Beispieldateien

Im Download zu diesem Artikel finden Sie einen Ordner mit den Laufzeitdateien der als Beispiel verwendeten Rechnungsverwaltung. Außerdem finden Sie in der Datei **VonSQLZuLINQ.linq** alle in diesem Artikel vorgestellten Beispiele.

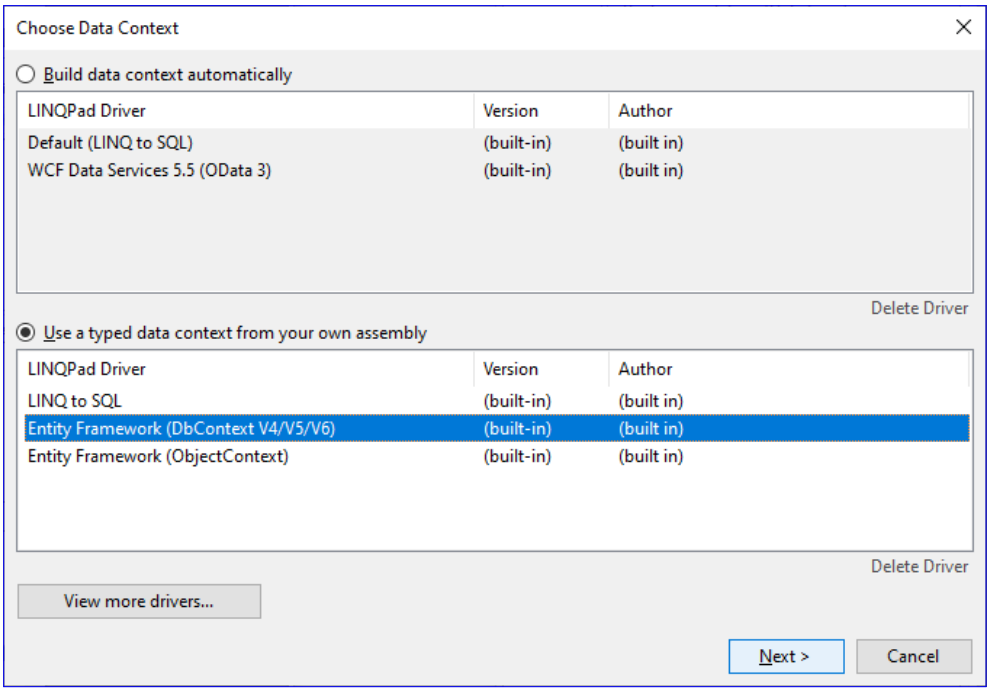


Bild 1: Einrichten einer Datenquelle mit LINQ

Was ist LINQ?

LINQ heißt **Language Integrated Query** und ist die Abfragesprache für Entity Data Models. Im Gegensatz zu VBA, wo Sie die Abfragen immer in SQL formulieren und als Parameter von Methoden wie **OpenRecordset** (bei Auswahlabfragen) oder **Execute** (bei Aktionsabfragen) angegeben haben, bietet LINQ die direkte Integration in die Sprache Visual Basic (und auch C#).

Im Artikel **EF: Daten abfragen mit VB und LINQ** (**www.datenbankentwickler.net/166**) haben wir bereits gezeigt, wie Sie Auswahlabfragen mit LINQ und VB realisieren. Nun schauen wir uns an, wie wir Aktionsabfragen unter LINQ abbilden können.

IntelliSense mit LINQ nutzen

LINQ bietet gegenüber SQL-Abfragen den Vorteil, IntelliSense nutzen zu können. Am einfachsten testen können wir LINQ-Ausdrücke nach wie vor mit dem Tool **LINQPad**. Wir arbeiten hier mit der Version 5. Um auf die Daten einer SQL Server-Datenbank zugreifen zu können, richten wir mit folgenden Schritten eine Verbindung ein.

Die einzige Voraussetzung ist, dass es eine **.exe** für das Projekt gibt und dass Sie wissen, wo die Konfigurationsdatei gespeichert ist – was nach dem Debuggen eines Desktop-Projekts mit Visual Studio aber beides vorhanden ist:

- Starten Sie LINQPad 5.
- Klicken Sie im linken Bereich auf **Add connection**.
- Wechseln Sie im Dialog **Choose Datacontext** aus Bild 1 zum unteren Bereich und wählen Sie dort den Eintrag **Entity Framework (DbContext V4/V5/V6)** aus und klicken Sie auf **Next**.
- Wählen Sie im Dialog **Entity Framework dbContext Connection** die Dateien für die Eigenschaften **Path to Custom Assembly** und **Path to application config file** aus. Hier wählen Sie die **.exe**-Datei aus, beispielsweise aus dem Ordner **Rechnungsverwaltung** aus dem Download zu diesem Artikel. Nach der Auswahl der ersten Datei werden Sie in einem weiteren Dialog aufgefordert, die Datenbank-Context-Klasse auszuwählen.
- Anschließend sieht der Dialog **Entity Framework DbContext Connection** wie in Bild 2 aus. Hier können Sie mit der Schaltfläche **Test** noch prüfen, ob die Verbindung hergestellt werden kann. Mit dem Aktivieren der Option **Remember this connection** stellen Sie sicher, dass die Verbindung auch beim nächsten Start von **LINQPad** noch vorhanden ist.

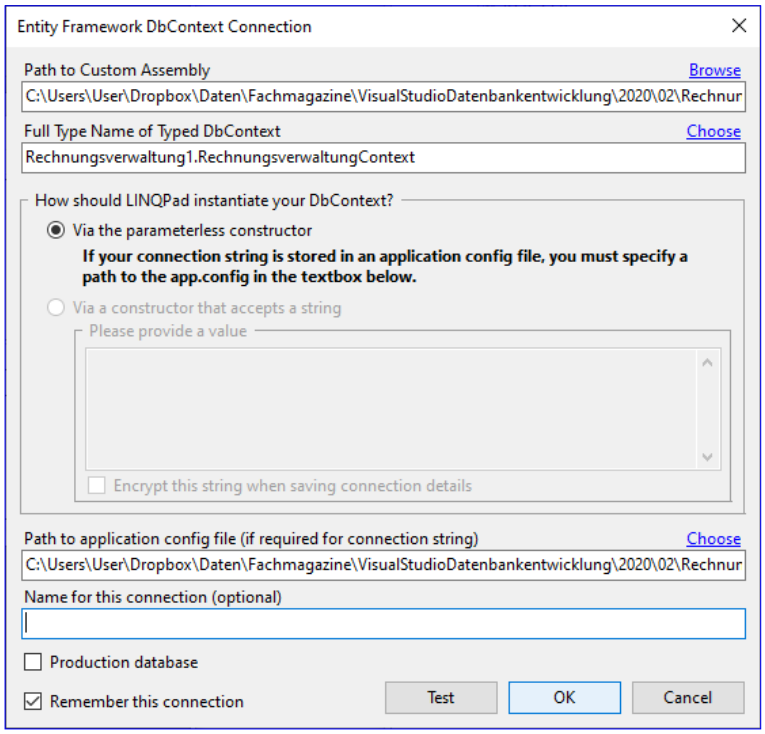


Bild 2: Einrichten der Verbindungsinformationen

Der linke Bereich zeigt nun die Entitäten und Felder der Verbindung an (siehe Bild 3). Mit einem Klick der rechten Maustaste auf eine der Entitäten öffnen Sie ein Kontextmenü, das bereits einige mögliche Befehle für diese Entität anzeigt. Klicken Sie einen dieser Einträge an, wird die passende LINQ-Abfrage im Query-Fenster eingefügt.

Standardmäßig zeigt dies nun eine C#-Abfrage an. Wenn Sie möchten, dass automatisch VB-Abfragen erstellt werden, öffnen Sie

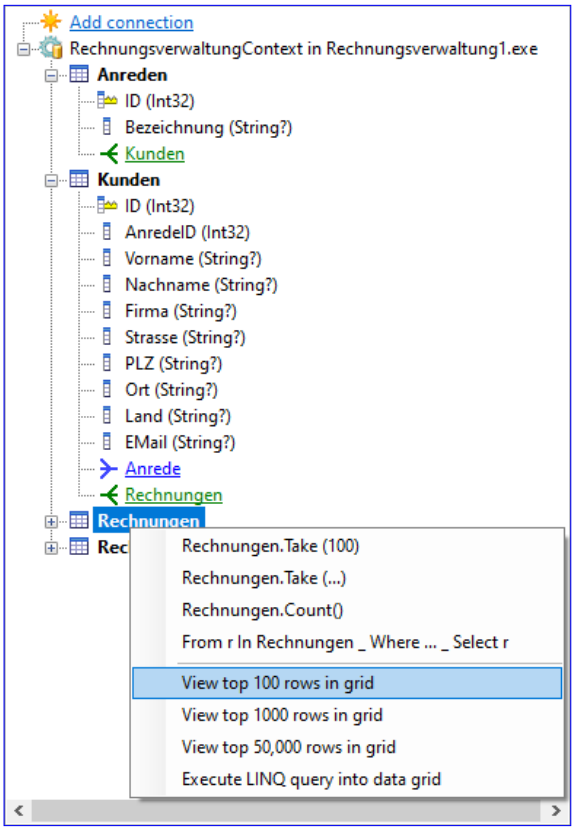


Bild 3: Die Entitäten der neuen Verbindung

mit dem Menüeintrag **Edit|Preferences** den Optionen-Dialog und stellen Sie auf der Seite **Query** die Option **Default Query Language** auf **VBA Expression** oder **VB Statement(s)** ein.

Damit sind Sie nun auch bereit, eigene LINQ-Abfragen auf Basis der Entitäten des Entity Data Models abzusetzen. Diese geben Sie einfach auf einer Seite mit dem Wert **VB Expression** im Auswahlfeld **Language** ein. Sie können übrigens in LINQPad den Verweis auf die Context-Klasse des Entity Data Models weglassen.

Ein Zugriff auf die Entitätslisten kann auch ohne Angabe dieser Klasse erfolgen. Allerdings ist das in Visual Studio nicht möglich, sodass wir in den folgenden Beispielen jeweils auf das Objekt **dbContext** zugreifen. Die Deklaration und Initialisierung wollen wir jedoch nur einmal anführen:

```
Dim dbContext As RechnungsverwaltungContext
dbContext = New RechnungsverwaltungContext
```

Einen Datensatz hinzufügen

Um einen Datensatz hinzuzufügen, benötigen wir noch nicht einmal einen LINQ-Ausdruck. Wir erstellen einfach ein neues Objekt des gewünschten Typs, hier **Kunde**, und weisen diesem die Eigenschaftswerte zu. Dann fügen wir das neue **Kunde**-Objekt mit der **Add**-Methode zur Auflistung **Kunden** hinzu und übernehmen die Änderungen durch einen Aufruf der **SaveChanges**-Methode in die Datenbank. Schließlich geben wir als Nachweis des erfolgreichen Anlegens noch den Wert des Primärschlüsselfeldes des neuen Datensatzes aus:

```
'###Beispiel 1
'Einen Kunden hinzufügen
Dim dbContext As RechnungsverwaltungContext
dbContext = New RechnungsverwaltungContext
Dim NeuerKunde As Kunde
NeuerKunde = New Kunde
With NeuerKunde
    .AnredeID = 1
    .Vorname = "André"
    .Nachname = "Minhorst"
End With
dbContext.Kunden.Add(NeuerKunde)
dbContext.SaveChanges
Debug.Print(NeuerKunde.ID)
```

Platzsparende Schreibweise für das Anlegen eines neuen Elements

Wir können ein paar Zeilen Code sparen, wenn wir das Anlegen wie folgt programmieren:

```
'###Beispiel 2
'Kunde hinzufügen, Kurzfassung
```

```
Dim NeuerKunde As Kunde
NeuerKunde = New Kunde With {.AnredeID = 1, .Vorname = "André", .Nachname = "Minhorst"}
dbContext.Kunden.Add(NeuerKunde)
dbContext.SaveChanges
Debug.Print(NeuerKunde.ID)
```

Die **NeuerKunde**-Variable könnten wir auch noch einsparen, wenn wir später nicht die **ID** des neuen Kunden ausgeben wollten:

```
dbContext.Kunden.Add(New Kunde With {.AnredeID = 1, .Vorname = "André", .Nachname = "Minhorst"})
dbContext.SaveChanges
```

Mehrere Datensätze hinzufügen

Wenn Sie mehrere Datensätze hinzufügen möchten, können Sie dies natürlich erledigen, indem Sie einfach immer einen neuen Kunden erstellen, seine Eigenschaften füllen und diesen mit der **Add**-Methode dem jeweiligen **DbSet**-Element zuweisen.

Dazu erstellen wir zuerst eine Liste mit Elementen des Typs **Kunde** und fügen dieser dann mit der **Add**-Methode die neuen **Kunde**-Elemente hinzu – mit der kurzen Schreibweise, die wir zuvor vorgestellt haben:

```
'### Beispiel 3
'Mehrere Kunden hinzufügen
Dim dbContext As RechnungsverwaltungContext
dbContext = New RechnungsverwaltungContext
Dim NeueKunden As List(Of Kunde)
NeueKunden = New List(Of Kunde)
NeueKunden.Add(New Kunde With {.AnredeID = 1, .Vorname = "André", .Nachname = "Minhorst"})
NeueKunden.Add(New Kunde With {.AnredeID = 2, .Vorname = "Claudia", .Nachname = "Müller"})
```

Danach geben wir die aktuelle Anzahl von Kunden in der Datenbank aus und rufen dann die **AddRange**-Methode auf, der wir die Liste der Kunden namens **NeueKunden** als Parameter übergeben. Nach dem Speichern der Änderungen mit der **SaveChanges**-Methode geben wir die neue Anzahl Kunden aus:

```
Debug.Print("Vorher: " & dbContext.Kunden.Count.ToString)
dbContext.Kunden.AddRange(NeueKunden)
dbContext.SaveChanges
Debug.Print("Nachher: " & dbContext.Kunden.Count.ToString)
```

Verknüpfte Daten hinzufügen

Wenn Sie beispielsweise einen **Rechnung**-Datensatz anlegen und diesen gleich um einen oder mehrere Rechnungsposition-Datensätze ergänzen wollen, gehen Sie wie folgt vor:

```
'### Beispiel 4
```

Debugging in Visual Studio

Wenn eine Anwendung sich nicht so verhält wie gewünscht oder diese sogar Ausnahmen auslöst, wollen wir herausfinden, warum das so ist. Wurde ein Wert nicht oder falsch gesetzt, konnte ein Objekt nicht gefüllt werden oder was ist die Ursache für das Problem? Hier kommen die Debugging-Techniken von Visual Studio ins Spiel. Der vorliegende Artikel stellt die gängigsten Techniken zum Debuggen von Code unter Visual Studio vor.

Das Wort **Debuggen** kommt der Sage nach vom Wort Bug. Früher wurden zum Durchführen von Rechenoperationen in Computern Relais verwendet, die durch Insekten blockiert werden konnten. Das Entfernen dieser Tierchen wurde schließlich **Debuggen** genannt – ein Begriff, der sich bis heute gehalten hat. Nur, dass Bug heute nicht mehr sprichwörtlich Käfer bedeutet, sondern schlicht Programmierfehler.

Anwendung zum Debuggen starten

Grundsätzlich gibt es zwei Modi zum Ausführen von .NET-Anwendungen: Den Debug-Modus und den Release-Modus. Der Release-Modus erzeugt eine leistungsoptimierte Version der Anwendung – diese entspricht auch der Version der Anwendung, die Sie mit dem Erstellen eines Projekts erhalten. Der Debug-Modus hingegen erlaubt es, bei Fehlern den Quellcode anzuzeigen, der für den Fehler verantwortlich ist. Außerdem können Sie in diesem Modus Informationen über den Zustand von Objekten und Variablen erhalten, die gerade im Gültigkeitsbereich liegen und Meldungen ausgeben sowie weitere Funktionen nutzen, die wir Ihnen im Laufe dieses Artikels vorstellen werden. Standardmäßig ist für das Starten einer Anwendung von Visual Studio aus der Debug-Modus eingestellt. Diese Einstellung können Sie leicht über das Menü von Visual Studio ändern (siehe Bild 1).

Um die Anwendung nun im Debugging-Modus zu starten, betätigen Sie entweder die Taste **F5**, klicken auf die Schaltfläche mit dem grünen Pfeil und dem Text **Starten** oder wählen den Menü-Eintrag **Debuggen|Debuggen starten**.

Zum Debuggen anhalten

Um eine Anwendung zu debuggen, muss der laufende Code angehalten werden. Dazu gibt es verschiedene Möglichkeiten:

- Es tritt eine Ausnahme auf, also ein Fehler im Code, der eine weitere Ausführung des Codes verhindert. Ein Beispiel sehen Sie in Bild 2.

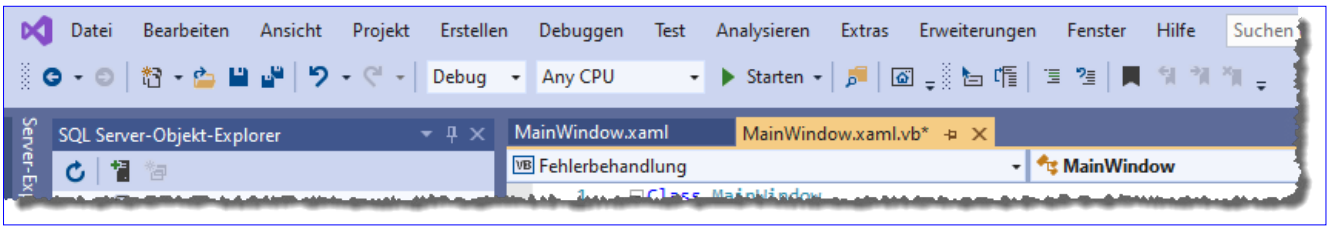


Bild 1: Einstellen des Ausführungsmodus, hier **Debug**

- Sie legen einen Haltepunkt im Code fest, damit der Code auch ohne einen Laufzeitfehler an der gewünschten Stelle angehalten und untersucht werden kann.

- Sie fügen der Anwendung an der gewünschten Stelle die Methode **Break** der Klasse **Debugger** hinzu. Dies entspricht der **Stop**-Anweisung, die Sie vielleicht von Access/VBA kennen. Der Code wird genau an dieser Stelle angehalten.

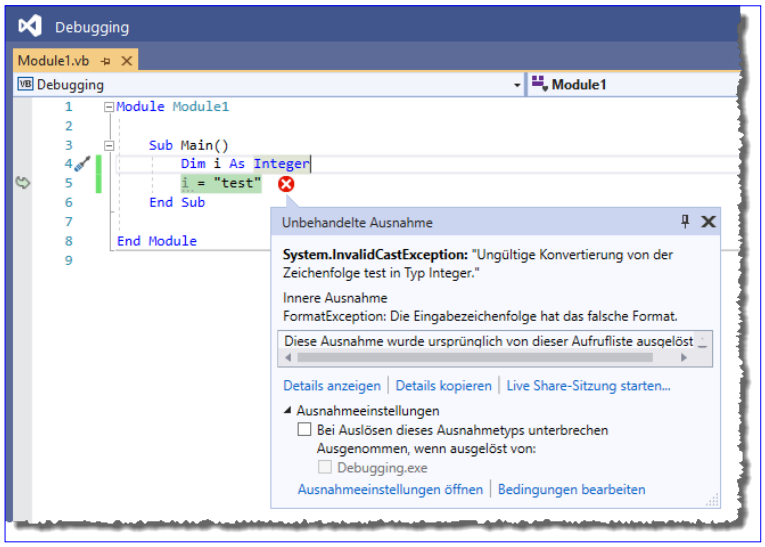


Bild 2: Debuggen wird durch Laufzeitfehler ermöglicht

Haltepunkt setzen

Einen Haltepunkt im Code setzen Sie, indem Sie im Codefenster in der Zeile, in der die Ausführung angehalten werden soll, in den linken, grauen Bereich klicken, sodass dort ein roter Punkt erscheint (siehe Bild 3).

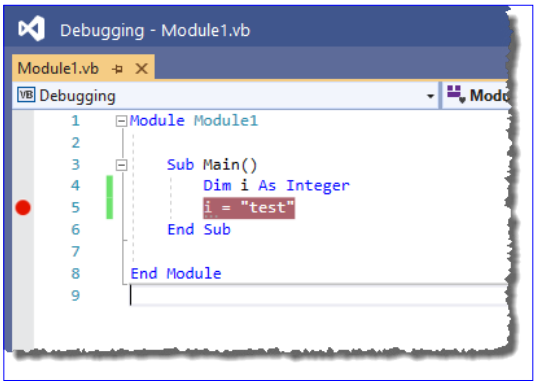


Bild 3: Haltepunkt

Eine alternative Möglichkeit zum Hinzufügen von Haltepunkten ist die Taste **F9**. Bewegen Sie die Einfügemarke in die gewünschte Zeile und betätigen Sie diese Taste, um einen Haltepunkt hinzuzufügen oder diesen wieder zu entfernen.

Sie können alle ausführbaren Zeilen mit einem Haltepunkt versehen, was alle Zeilen mit Ausnahme der Deklarationszeilen umfasst.

Wenn die Ausführung an der markierten Zeile stoppt, bedeutet das übrigens, dass die markierte Zeile noch nicht ausgeführt wurde. Wenn Sie also Variablen in einer Zeile nach der Ausführung (etwa der Zuweisung eines Wertes an eine Variable) untersuchen möchten, können Sie den Haltepunkt auch gleich auf die folgende Zeile verschieben.

Haltepunkte per Code

Eine weitere Möglichkeit, einen Haltepunkt zu setzen, bietet die Klasse Debugger. Diese stellt die Methode **Break** zur Verfügung, die beim Erreichen den Code unterbricht (siehe Bild 4).

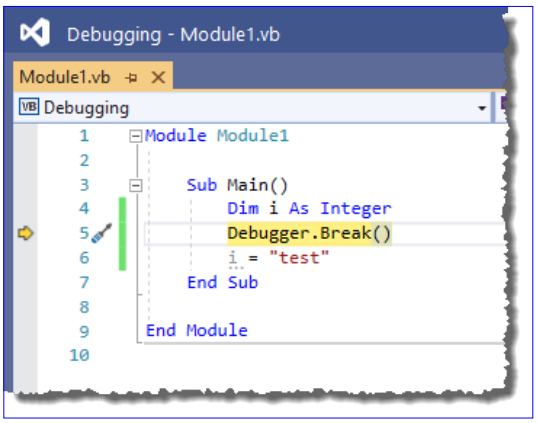


Bild 4: Programmierter Stop

Warum sollte man diese Methode nutzen, statt einfach einen Haltepunkt zu setzen? Unter Access und dem VBA-Editor wurden Haltepunkte gelöscht, wenn man die Anwendung geschlossen und erneut geöffnet hat. Das war unter Access der Grund, die **Stop**-Anweisung zu verwenden – wenn die Anwendung abgestürzt ist, war diese nach dem Neustart wieder vorhanden, die Haltepunkte aber nicht. Unter Visual Studio sind die Haltepunkte allerdings nach dem

Schließen und erneuten Öffnen des Projekts noch vorhanden – eigentlich braucht man **Debugger.Break** also nicht.

Beim Erreichen eines Haltepunktes

Wenn Sie beim Debuggen einen Haltepunkt erreichen, können Sie verschiedene Aktionen durchführen. Die erste ist, Informationen über die aktuelle Situation zu ermitteln. Dazu können Sie beispielsweise über den Namen einer Variablen fahren, um ihren aktuellen Wert zu ermitteln (siehe Bild 5).

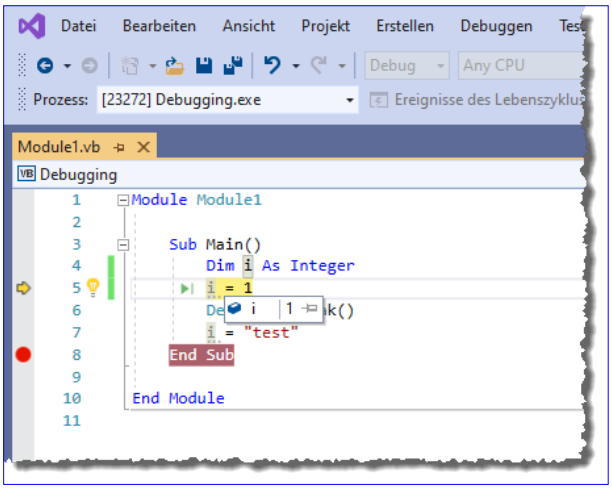


Bild 5: Auslesen von Variableninhalten

Weiter im Code

Wenn Sie einen Haltepunkt erreicht haben und das Debuggen nun fortsetzen wollen, gibt es dazu folgende Möglichkeiten:

- Sie betätigen erneut die Taste **F5** und der Code läuft weiter bis zum Schluss oder bis er durch einen weiteren Haltepunkt oder einen Fehler angehalten wird.

- Sie betätigen die Taste **F11**, um jeweils eine Zeile auszuführen.

Letztere Möglichkeit ist interessant, wenn Sie im Detail die einzelnen Zeilen des Codes durchlaufen wollen.

Laufzeitfehler untersuchen

Wenn Sie beim Debuggen auf einen Laufzeitfehler stoßen, der nicht behandelt wird, können Sie damit einige Informationen erhalten, die für eine Behandlung des Fehlers im Code nützlich sind.

Wie die Fehlermeldung einer unbehandelten Ausnahme aussieht, haben wir bereits weiter oben betrachtet. Hier finden wir noch einige weiterführende Möglichkeiten. Die wichtigste ist der Link **Details anzeigen**, mit dem Sie ein Fenster wie in Bild 6 öffnen können.

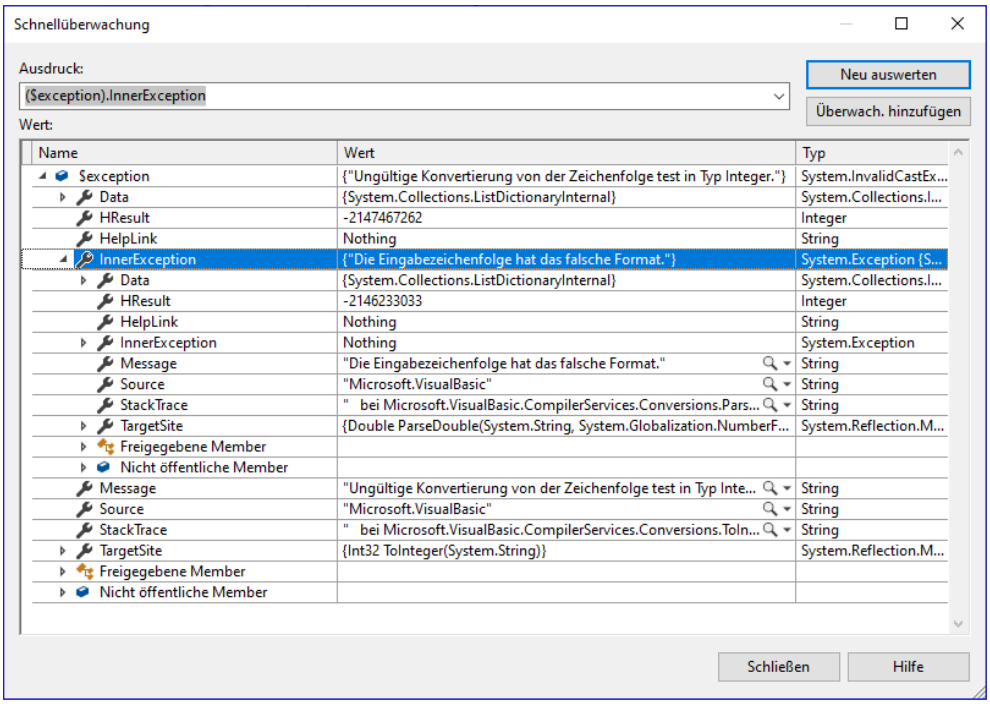


Bild 6: Fehlerdetails beim Auftreten einer Ausnahme

Hier finden Sie detaillierte Informationen zu der Ausnahme. In vielen Fällen können Sie unter **InnerException** noch weitere Informationen anzeigen, wie es in der Abbildung der Fall ist.

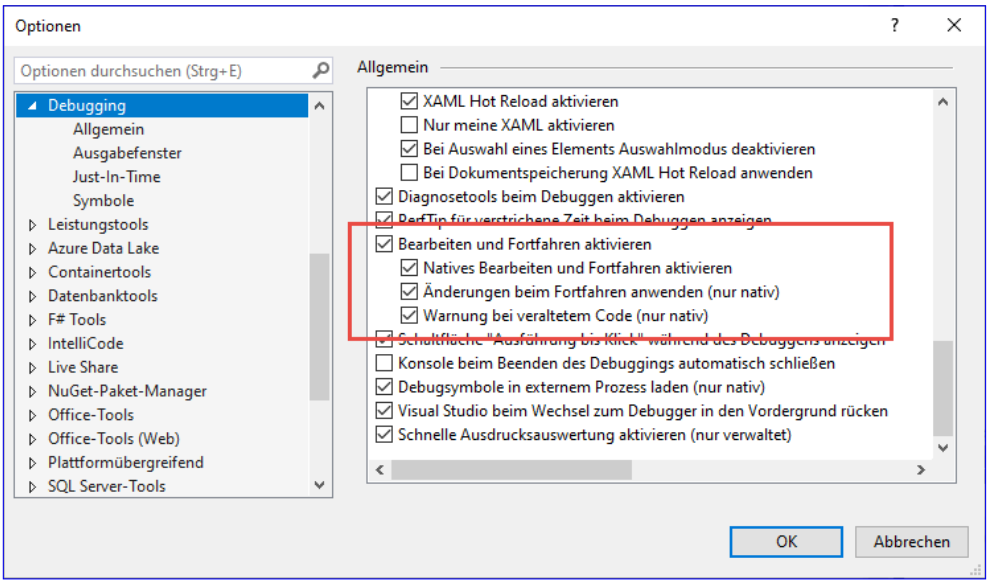


Bild 8: Aktivieren des Bearbeitens von Code während des Debuggens

Code während des Debugging bearbeiten

Eine wichtige Funktion ist das Bearbeiten des Codes während des Debuggings. Damit dies funktioniert, aktivieren Sie die Optionen aus Bild 8 im Bereich **Debugging** des Optionen-Dialogs, den Sie mit dem Menübefehl **Extras|Optionen** öffnen – und die normalerweise bereits aktiviert sind..

Es werden allerdings nicht alle Änderungen unterstützt. Unter folgendem Link finden Sie eine Liste der unterstützten und nicht unterstützten Änderungen:

`https://github.com/dotnet/roslyn/blob/master/docs/wiki/EnC-Supported-Edits.md`

Debugging steuern

Neben dem Starten und Fortsetzen des Debuggings mit **F5** und dem Durchführen von Einzelschritten mit **F11** gibt es noch weitere Optionen, die Sie auch im Menüpunkt **Debuggen** finden (siehe Bild 7). Wenn Sie einmal die Tastenkombination für einen der Befehle suchen, können Sie diese auch in diesem Menü einsehen. Die Befehle im Einzelnen:

- **Weiter** (**F5**): Setzt den Code bis zum nächsten Haltepunkt oder Fehler beziehungsweise bis zum Ende des Codes durch.
- **Alle unterbrechen** (**Strg + Alt + Pause**): Unterbricht an der aktuell ausgeführten Codezeile.
- **Debuggen beenden** (**Umschalt + F5**): Beendet das Debuggen.

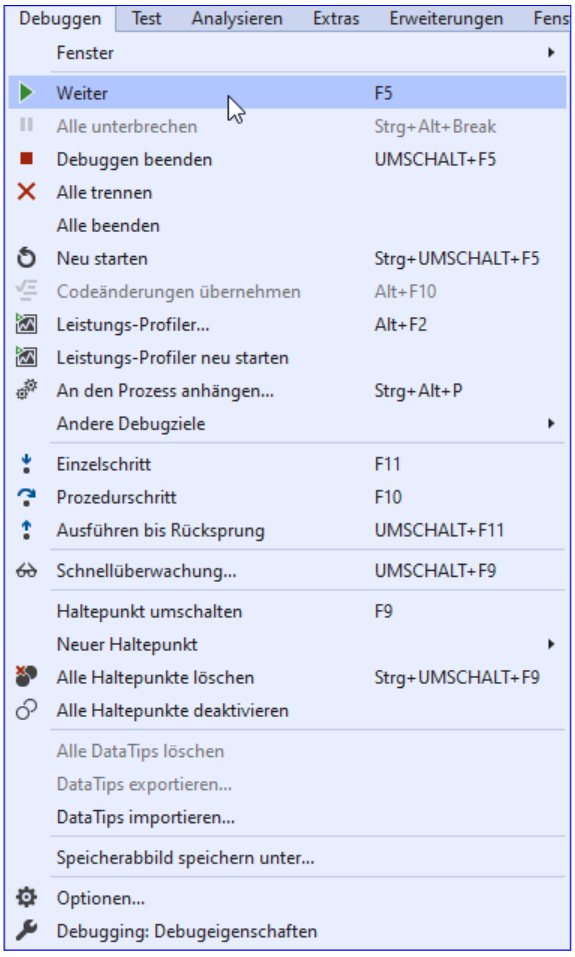


Bild 7: Kontextmenü-Befehle zum Debuggen

- **Neu starten** (**Strg + Umschalt + F5**): Beendet das Debugging und startet es erneut.

- **Prozedurschritt** (**F10**): Führt die Anweisungen der aktuellen Methode im Einzelschrittmodus aus, aber springt nicht in andere, von dieser Methode aus aufgerufene Methoden.

- **Ausführen bis Rücksprung** (**Umschalt + F11**): Springt von der aktuellen Methode direkt zur aufrufenden Methode, wenn überhaupt ein Aufruf von einer anderen Methode erfolgt ist.

Damit erhalten wir recht exakt die gleichen Funktionen und Tastenkombinationen wie im VBA-Editor von Access – mit dem Unterschied, dass statt der Taste **F8** die Taste **F11** zum Einsatz kommt.

Es gibt noch weitere Befehle, die allerdings erst bei laufendem Debugging durch einen Mausklick mit der rechten Maustaste in das Codefenster auftauchen (siehe Bild 9). Interessant sind hier die folgenden Befehle:

- **HaltepunktlHaltepunkt einfügen**: Fügt einen Haltepunkt in der aktuellen Zeile ein.

- **Nächste Anweisung anzeigen** (**Alt + Num ***): Durch das Betätigen dieses Kontextmenübefehls beziehungsweise der Tastenkombination aus der **Alt**-Taste und der ***-Taste des Nummernblocks wird die Einfügemarke auf die nächste auszuführende Anweisung verschoben. Dieser Befehl ist sinnvoll, wenn Sie während des Debuggens andere Stellen im Code betrachten und dabei die aktuell auszuführende Anweisung aus dem sichtbaren Bereich verschwindet – Sie gelangen dann schnell wieder zur aktiven Stelle zurückl

- **Ausführen bis Cursor** (**F10**): Mit diesem Befehl können Sie die Einfügemarke an eine beliebige Stelle im Code platzieren und den Code dann bis zu dieser

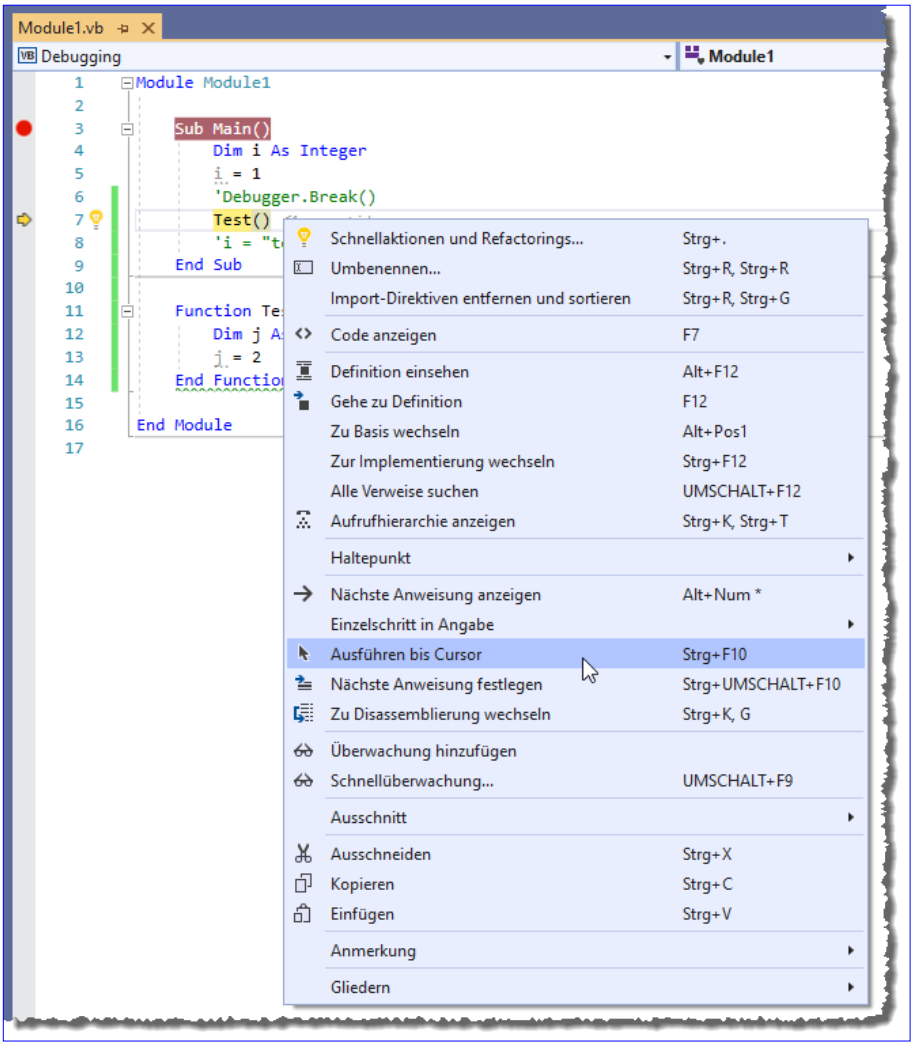


Bild 9: Weitere Debugging-Befehle

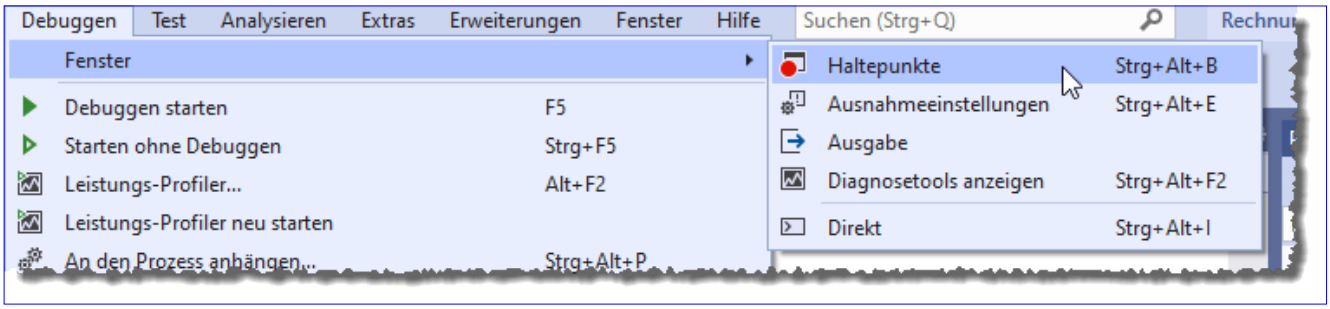


Bild 10: Aktivieren des Tool Windows zur Verwaltung der Haltepunkte

Stelle durchlaufen lassen. Das ist die Kurzversion für das Setzen eines Haltepunkts, der direkt wieder entfernt werden soll.

- **Nächste Anweisung festlegen** (**Strg + Umschalt + F10**): Wenn Sie aktuell schrittweise durch den Code laufen oder dieser anderweitig unterbrochen ist (die aktuelle Zeile ist gelb hinterlegt), können Sie direkt zu einer anderen Zeile springen – und zwar zu der, in der sich bei Betätigen des Menübefehls beziehungsweise der Tastenkombination die Einfügemarke befindet. Dies gelingt jedoch nur für Anweisungen innerhalb der aktuellen Methode.

Tool Windows mit Debugging-Informationen

Visual Studio bietet eine ganze Reihe von Tool Windows, die verschiedene Informationen anzeigen – darunter auch Debugging-Informationen.

Haltepunkte verwalten

In längeren Debugging-Sessions mit Fehlersuche kommen gern einige Haltepunkte zusammen. Diese müssen Sie nicht mühsam von Hand aus den betroffenen Modulen entfernen – Sie können ein Tool Window verwenden, das die Haltepunkte übersichtlich anzeigt und die Möglichkeit bietet, die Haltepunkte zu verwalten. Dieses Tool Window zeigen Sie entweder über den Menüeintrag **Debuggen|Fenster|Haltepunkte** (siehe Bild 10) oder über die Tastenkombination **Strg + Alt + B** an.

Dies aktiviert das Tool Window namens **Haltepunkte** (siehe Bild 11). Die Liste zeigt alle aktuellen Haltepunkte des Projekts an. Der Bereich bietet verschiedene Möglichkeiten. Um alle markierten Haltepunkte schnell loszuwerden, klicken Sie einfach auf die **Löschen**-Schaltfläche. Nicht zu entfernende Haltepunkte können Sie vorher abwählen.

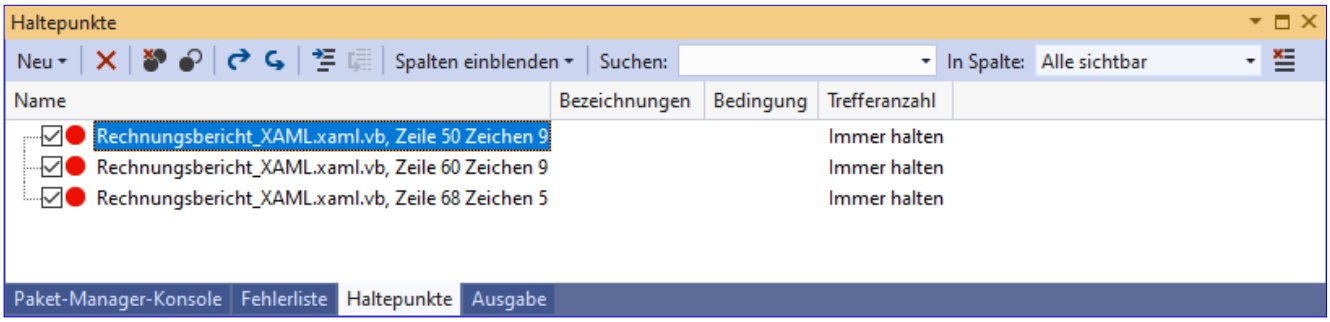


Bild 11: Anzeige der Haltepunkte einer Anwendung

Fehlerbehandlung unter VB.NET

Was wäre eine Anwendung ohne Fehlerbehandlung? Auch wenn wir in den bisherigen Beispielanwendungen meist gar keine Fehlerbehandlung genutzt haben, weil der Code so übersichtlicher ist. Wenn Sie aber eine Anwendung an Kunden weitergeben, sollte diese keine unbehandelten Ausnahmen liefern. Deshalb beschreiben wir im vorliegenden Artikel, wie die Fehlerbehandlung unter VB.NET funktioniert und welche Unterschiede sich zur Fehlerbehandlung unter Access/VBA ergeben.

Beispiele ausprobieren

Wir haben die Beispiele im Tool LINQPad eingegeben und ausprobiert. Das Tool ist in der Basisversion kostenlos und erlaubt das Ausführen von VB.NET-Prozeduren, ohne das Sie immer erst ein Projekt zum Debuggen starten müssen. Den Download des Tools finden Sie unter **www.linqpad.net**.

Von VBA zu VB.NET

Unter VBA war die Fehlerbehandlung recht einfach: Wir haben mit **On Error Resume Next** festgelegt, dass die Fehlerbehandlung für die folgenden Zeilen ausgesetzt wird und haben selbst mit **Err.Number** geprüft, ob ein Fehler aufgetreten ist und entsprechend darauf reagiert. Damit haben wir dann gezielt Fehler behandelt. Oder man hat zu Beginn einer Prozedur mit einer Anweisung wie **On Error Goto Fehler** festgelegt, dass die Prozedur im Falle eines Fehlers zu einer Marke namens **Fehler** springen sollte, wo wir dann eine allgemeine Fehlerbehandlung eingebaut haben, die beispielsweise eine E-Mail mit der Fehlermeldung und weiteren Informationen an den Hersteller der Software geschickt hat.

Wenn Sie eine Anwendung von Access nach VB.NET migrieren und dabei möglichst viel Quellcode übernehmen wollen, gibt es allerdings eine gute Nachricht: Die unter VBA verwendeten Befehle für die Fehlerbehandlung funktionieren auch noch unter VB.NET.

Das folgende klassische Beispiel liefert allerdings nicht den Beweis, dass die Fehlerbehandlung wie unter VBA funktioniert. Dafür benötigen Sie allerdings einen Verweis auf den Namespace **Microsoft.VisualBasic**:

```
Sub DivisionByZero
    On Error Resume Next
    Debug.Print(1 / 0)
    If Not (Err.Number = 0) Then
        Debug.Print("Fehler: " + Err.Number + " " + Err.Description)
    End If
End Sub
```

Das Ergebnis der Division von **1** und **0** wird nämlich einfach wie folgt ausgegeben:

∞

Also lösen wir den Fehler mit der entsprechenden .NET-Anweisung aus, nämlich **ThrowException**:

```
On Error Resume Next
Throw New DivideByZeroException()
'Debug.Print(1 / 0)
If Not (Err.Number = 0) Then
    Debug.Print("Fehler: " & Err.Number & " " & Err.Description)
End If
```

Damit haben wir auch gleich die erste Anweisung der Ausnahmebehandlung unter VB.NET kennengelernt – die Thow-Anweisung dient dazu, Ausnahmen auszulösen. Mehr dazu weiter unten! Übrigens sollten Sie, auch wenn es möglich ist, nicht mehr die von VBA gewohnten Möglichkeiten zur Fehlerbehandlung nutzen.

Fehlerbehandlung unter VB.NET

.NET bietet wesentlich umfassendere Möglichkeiten für die Fehlerbehandlung oder, wie man hier sagt, Ausnahmebehandlung. Die grobe Richtschnur für eine Fehlerbehandlung unter VB.NET ist ein Block ähnlich einer **If...Then...Else**-Bedingung mit den drei Anweisungen **Try**, **Catch** und **Finally**. Die Mindestanforderung sind die beiden Anweisungen **Try**, **Catch** und die **End Try**-Anweisung:

```
Try
    'auszuführender Code, der eine Ausnahme auslösen könnte
Catch ex As Exception
    'Code, der beim Auftreten der Ausnahme ausgeführt werden soll
End Try
```

Das Pendant zu der Fehlerbehandlung von oben mit der Division durch 0 würde also wie folgt aussehen:

```
Sub DivisionByZeroVBNET
    Try
        Throw New DivideByZeroException()
    Catch ex As Exception
        Debug.Print("Fehler: " & ex.Message)
    End Try
End Sub
```

Der Aufbau ist also prinzipiell gar nicht so viel anders als unter VBA. Der Teil zwischen **Catch...** und **End Try** wird nur ausgeführt, wenn eine Ausnahme ausgelöst wird. Ein wichtiger Unterschied: Es gibt unter VB.NET kein **Err**-Objekt (zumindest nicht, wenn Sie **Try...Catch** verwenden). Dafür legen Sie in der **Catch**-Anweisung jedoch ein Objekt des Typs **Exception** fest, hier mit der Bezeichnung **ex**. **Exception** liefert uns wie **Err** zwar eine Meldung, aber keine Fehlernummer.

Wie sollen wir das nun vernünftig auswerten und je nach Fehler entsprechend reagieren?

Ganz einfach: Es gibt für jede Ausnahme eine spezielle Exception. Für diesen Fall ist diese leicht zu finden, denn die Exception wird durch das »Werfen« der Ausnahme **DivideByZeroException** ausgelöst. Dementsprechend können wir die **Catch**-Anweisung direkt etwas genauer definieren, sodass diese nur auf **DivideByZeroException**-Ausnahmen reagiert:

```
Sub DivisionByZeroVBNET_DivideByZeroException
    Try
        Throw New DivideByZeroException()
    Catch ex As DivideByZeroException
        Debug.Print("Fehler: " & ex.Message & " ")
    End Try
End Sub
```

Da Sie wissen, welcher Fehler hier behandelt wird, können Sie die Meldung auch anpassen, ohne auf **ex.Message** zuzugreifen:

```
Debug.Print ("Division durch Null.")
```

Tritt nach der **Try**-Anweisung eine andere Ausnahme als **DivideByZeroException** auf, führt dies allerdings zu einer unbehandelten Ausnahme – zum Beispiel wenn Sie mit folgender Anweisung einen Dateifehler auslösen:

```
Throw New FileNotFoundException
```

Wir sollten also auch mit der allgemeinen Ausnahme **Exception** alle möglichen Ausnahmen abfangen. Dazu fügen wir einfach einen weiteren **Catch**-Zweig zu dem Konstrukt hinzu:

```
Try
    Throw New FileNotFoundException
    Throw New DivideByZeroException()
Catch ex As DivideByZeroException
    Debug.Print("Die Datei konnte nicht gefunden werden.")
Catch ex As Exception
    Debug.Print("Fehler: " & ex.Message)
End Try
```

Restarbeiten mit und ohne Ausnahme

Unter VBA haben wir für Restarbeiten, die in jedem Fall, also auch beim Auftreten eines Fehlers zu erledigen sind, wie folgt untergebracht:

```
    On Error Resume Fehler
    'Fehlerauslösende Zeile
Ende:
    'Restarbeiten
```

```
    Exit Sub
Fehler:
    'Fehlerbehandlung
    Goto Ende
```

Wir haben also unter der Marke **Ende** Befehle untergebracht, die auch beim Ausbleiben eines Fehlers ausgelöst werden. Damit die **Fehler**-Marke nur erreicht wird, wenn tatsächlich ein Fehler auftritt, haben wir dieser die **Exit Sub**- beziehungsweise **Exit Function**-Anweisung vorangestellt. Damit die Aufräumarbeiten aber auch nach Auftreten eines Fehlers noch ausgeführt werden, haben wir am Ende der Fehlerbehandlung noch einen Verweis zur Marke **Ende** hinzugefügt.

Unter VB.NET benötigen wir dazu nur einen weiteren Zweig im **Try...End Try**-Konstukt. Dieser Zweig heißt **Finally** und wird immer am Ende eingebaut:

```
Sub TryCatchFinally
    Try
        Throw New FileNotFoundException
        Throw New DivideByZeroException()
    Catch ex As DivideByZeroException
        Debug.Print("Fehler: " & ex.Message)
    Catch ex As Exception
        Debug.Print("Fehler: " & ex.Message)
    Finally
        Debug.Print("Restarbeiten")
    End Try
End Sub
```

Die Anweisungen im **Finally**-Zweig werden auf jeden Fall ausgeführt – unabhängig davon, ob eine Ausnahme ausgelöst wird oder nicht.

Ausnahmetyp ermitteln

Wenn Sie programmieren, werden Sie vermutlich zunächst mit dem allgemeinen Ausnahmetyp **Exception** arbeiten. Um gezielt Ausnahmen zu behandeln, müssen Sie dann wissen, welche Ausnahme genau ausgelöst wird.

Wir wir oben schon angedeutet haben, gibt es eine Hierarchie der Ausnahme-Objekte. Das oberste Objekt ist **Exception**, darunter folgen dann weitere wie zum Beispiel **DivideByZeroException**. Es gibt auch mehrere Ebenen – zum Beispiel gibt es unter der **Exception**-Ausnahme eine **IOException**-Ausnahme und darunter wieder die **FileNotFoundException**-Ausnahme. In den **Catch**-Zweigen des **Try...End Try**-Blocks arbeiten Sie die Ausnahmen wie oben gezeigt in der umgekehrten Reihenfolge ab.

Doch zurück zum Ermitteln der spezifischen Ausnahme für einen Fehler – diesen können Sie einfach über die Eigenschaft **GetType** ermitteln:

```
Sub AusnahmeHerausfinden
    Try
        Throw New DivideByZeroException()
    Catch ex As Exception
        Debug.Print("Fehler: " & ex.GetType.ToString())
    End Try
End Sub
```

In diesem Beispiel liefert das die folgende Ausgabe:

```
Fehler: System.DivideByZeroException
```

Variablen, die in allen Zweigen gültig sein sollen, außerhalb deklarieren

Wenn Sie im **Try**-Block Variablen verwenden wollen, auf die Sie nachher im **Finally**-Block noch zugreifen wollen, müssen Sie diese außerhalb des **Try...End Try**-Konstrukts deklarieren. Im folgenden Fall haben wir die Deklaration im **Try**-Zweig vorgenommen, was zu einem Fehler im **Finally**-Zweig führt:

```
Sub Variablengueltigkeit
    Try
        Dim strTest As String
        strTest = "Beispieltext"
        Throw New DivideByZeroException()
    Catch ex As DivideByZeroException
        Debug.Print("Fehler: " & ex.Message)
    Catch ex As Exception
        Debug.Print("Fehler: " & ex.Message)
    Finally
        Debug.Print("Restarbeiten: " + strTest)
    End Try
End Sub
```

Die Deklaration müssen wir also wie folgt vorziehen:

```
Sub Variablengueltigkeit
    Dim strTest As String
    Try
        strTest = "Beispieltext"
    ...
```

Eigenschaften von Exception-Klassen

Die **Exception**-Klassen stellen einige interessante Eigenschaften bereit, die beim Umgang mit Ausnahmen hilfreich sein können:

XAML-Eigenschaften beim Debuggen testen

Visual Studio 2019 und höher

XAML bietet sehr viele Möglichkeiten, das Design von Steuerelementen in einer Anwendung anzupassen. Viele der Eigenschaften sind von den Standardsteuerelementen vorgegeben, andere passen Sie selbst durch Definition des XAML-Codes an. Wenn dann beim Testen der Anwendung Elemente nicht wie erwartet aussehen, stellt sich die Frage, woran das liegt. Dann lautet die Aufgabe, herauszufinden, woher die jeweilige Einstellung stammt und diese so anzupassen, dass das Design wie gewünscht erscheint. Für solche Zwecke gibt es in Visual Studio eingebaute Tools, die wir uns in diesem Artikel ansehen. Mit Visual Studio 2019 gibt es viele Verbesserungen, die ein Update lohnenswert machen.

Logical Tree vs. Visual Tree

Es gibt zwei verschiedene Hierarchien bei der Darstellung von XAML-Benutzeroberflächen. Die erste kennen Sie sehr gut – dabei handelt es sich um den Logical Tree. Der Logical Tree besteht aus den Elementen, die Sie im XAML-Editor definieren – also Elemente wie **Window**, **Grid**, **StackPanel**, **TextBox**, **Button** und so weiter.

Der Logical Tree definiert dabei nicht nur, welche Elemente in der XAML-Definition enthalten sind, sondern auch deren hierarchische Anordnung.

Die zweite Hierarchie ist der Visual Tree. Warum benötigen wir eine zweite Hierarchie? Weil die Elemente des Logical Tree auf Vorlagen basieren, die nicht nur aus einem einzigen Element bestehen, sondern aus mehreren. Der Logical Tree ist also eine vereinfachte Form des Visual Tree. Wenn der Logical Tree gerendert wird, also wenn die Darstellung der Benutzeroberfläche daraus generiert wird, entstehen daraus meist sehr viel mehr Elemente. Ein **ListView**-Steuerelement etwa besteht aus Rändern, Bildlaufleisten, Spaltenköpfen, Zeilen, Spalten und weiteren Elementen.

Warum müssen wir das an dieser Stelle wissen? Weil wir zur Laufzeit nicht auf die Eigenschaften der Elemente des Logical Trees zugreifen können, aber auf die Eigenschaften des Visual Trees. Wie das gelingt, zeigen wir in den folgenden Abschnitten.

Debugging-Funktionen aktivieren

Damit wir die nachfolgend vorgestellten Funktionen nutzen können, müssen wir diese zunächst in den Optionen von Visual Studio aktivieren. Dazu öffnen Sie den Optionen-Dialog mit dem Menübefehl **Extras|Optionen**. Im Optionen-Dialog wechseln Sie dann zum Bereich **Debugging|Allgemein**. Hier finden Sie die Option **UI-Debugtools für XAML aktivieren** (siehe Bild 1).

Die einzelnen Einstellungen haben die folgende Bedeutung:

- **UI-Debugtools für XAML aktivieren**: Mit dieser Option legen Sie fest, ob die Debugtools überhaupt aktiviert sein sollen. Die untergeordneten Optionen steuern dann, wie die Debug-Tools bereitgestellt werden.

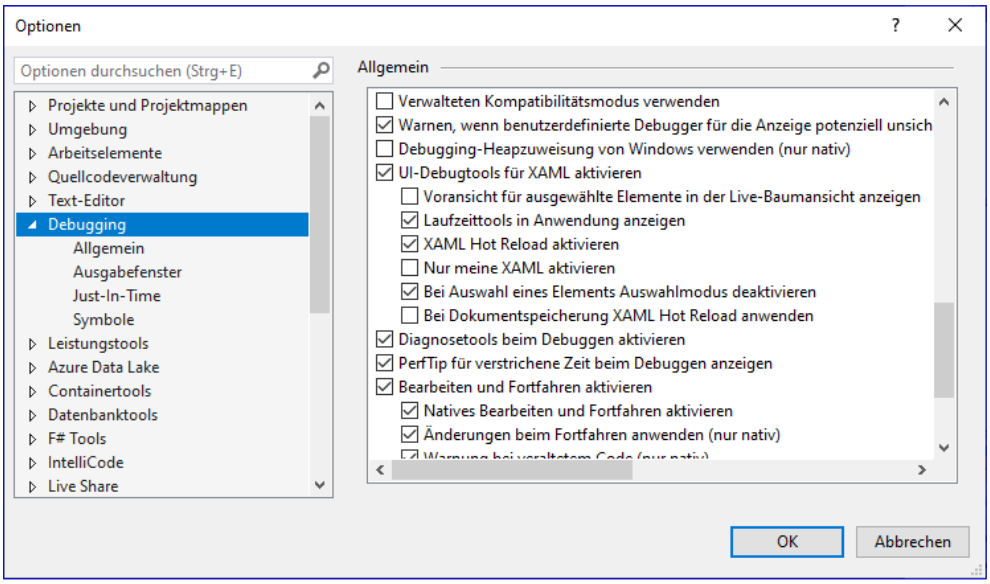


Bild 1: Einstellungen für das Debuggen von XAML-Code

- **Voransicht für ausgewählte Elemente in der Live-Baumansicht anzeigen**: Das aktuell ausgewählte XAML-Element wird im Bereich **Visueller Livebaum** angezeigt.

- **Laufzeittools in Anwendung anzeigen**: Blendet eine Menüleiste oben im Fenster der zu testenden Anwendung ein, über welche die Debugging-Funktionen schnell aufgerufen werden können (siehe Bild 2). Die Laufzeittools können Sie übrigens auch zur Laufzeit durch Aktivieren der Option einblenden.

- **XAML Hot Reload aktivieren**: Aktiviert die Funktion XAML Hot Reload, die dazu dient, zur Laufzeit Änderungen am XAML-Code vorzunehmen und diese direkt in Echtzeit zu testen.

- **Nur meine XAML aktivieren**: Der visuelle Livebaum zeigt ab Visual Studio 2019 (Version 16.4) nur benutzerdefinierte Elemente an, wenn diese Option aktiviert ist. Wenn Sie alle Elemente sehen wollen, deaktivieren Sie diese Option.

- **Bei Auswahl eines Elements Auswahlmodus deaktivieren**: Wenn Sie bei aktivierter Option in der Menüleiste für die Debugging-Tools in der Anwendung auf die Schaltfläche **Element auswählen** klicken und dann über die Elemente der Anwendung fahren, wird das jeweils überfahrene Element mit einem gestrichelten, roten Rahmen versehen. Klicken Sie es an,

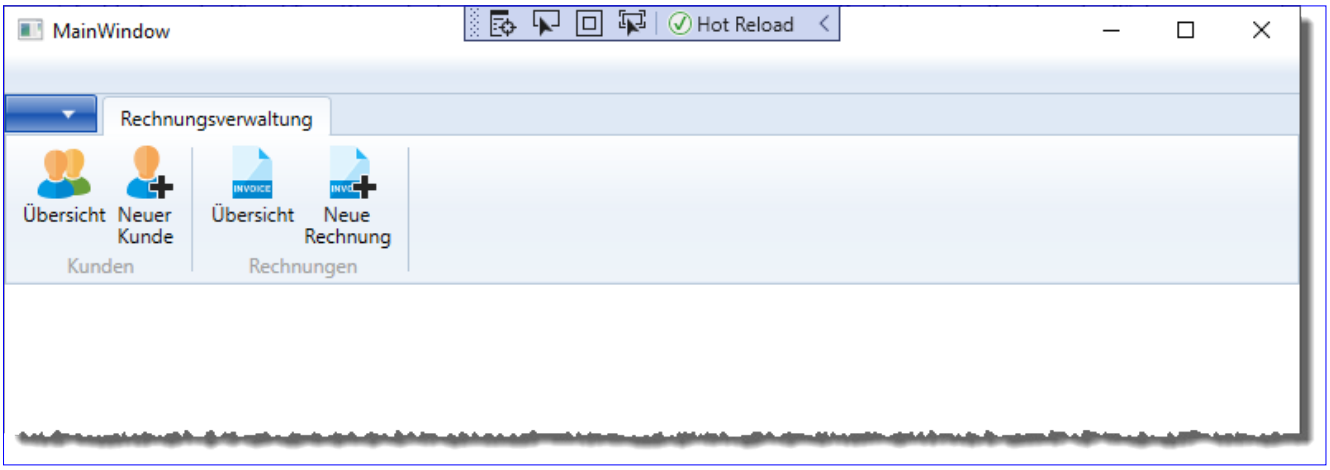


Bild 2: Steuerelemente für das Debuggen von XAML-Eigenschaften

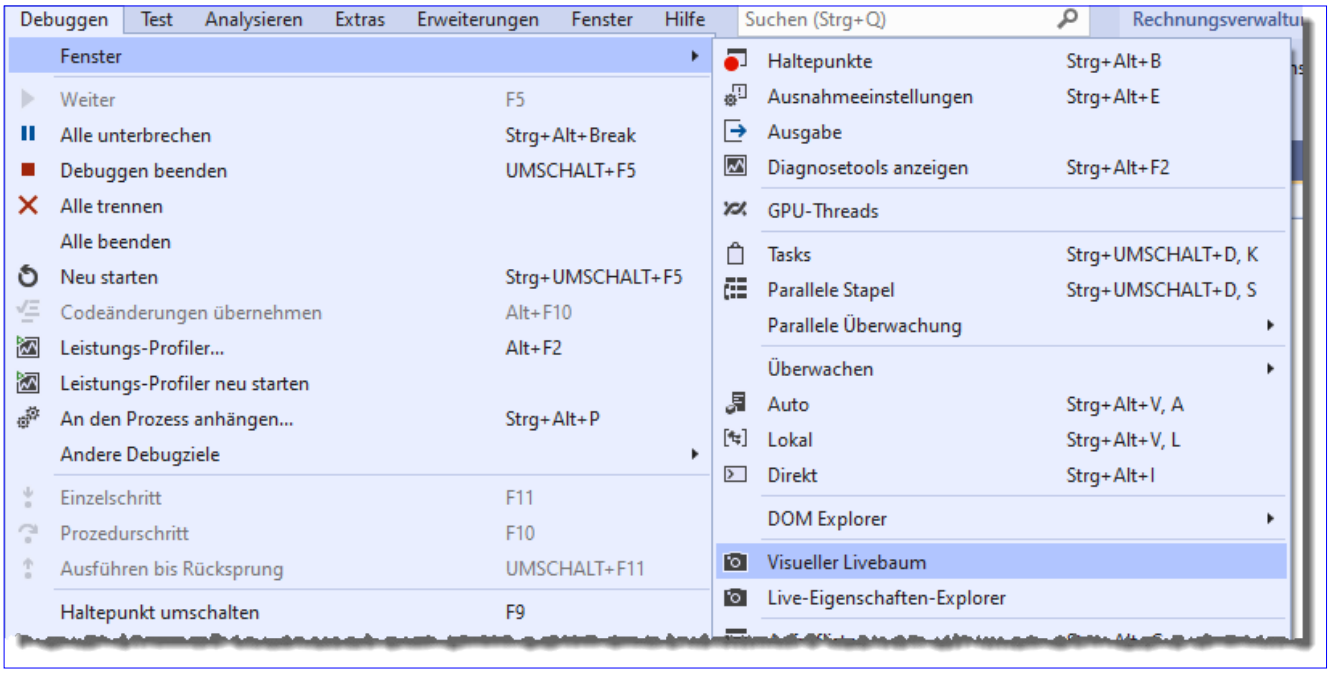


Bild 3: Visuellen Livebaum und Live-Eigenschaften-Explorer per Menü aktivieren

wird das jeweilige Element im visuellen Livebaum hervorgehoben. Danach wird die Schaltfläche Element auswählen wieder deaktiviert und Sie müssen diese für die erneute Auswahl eines Elements erneut anklicken. Ist die Option deaktiviert, können Sie nach dem Anklicken der Schaltfläche **Element auswählen** immer wieder neue Elemente im visuellen Livebaum anzeigen lassen. Die Schaltfläche **Element auswählen** bleibt dann aktiviert.

- **Bei Dokumentspeicherung XML Hot Reload anwenden**: XML Hot Reload wird angewendet, wenn Sie das Dokument speichern.

Visuellen Livebaum und Eigenschaften-Explorer aktivieren

Wenn Sie die Symbole zum Aufrufen der Debugging-Funktionen im Anwendungsfenster nicht sehen wollen, können Sie die wichtigsten Funktionen auch über das Menü von Visual Studio aufrufen. Dort finden Sie die Einträge **Debuggen|Fenster|Visueller Livebaum** und **Debuggen|Fenster|Live-Eigenschaften-Explorer** (siehe Bild 3).

Den visuellen Livebaum nutzen

Den visuellen Livebaum haben wir nun schon mehrfach erwähnt, aber wir haben ihn uns noch nicht angesehen. Den logischen Baum kennen wir, also schauen wir uns nun den visuellen Livebaum an. Diesen aktivieren Sie beispielsweise über den oben genannten Menübfehl **Debuggen|Fenster|Visueller Livebaum** oder, wenn das Laufzeittools-Menü im Anwendungsfenster aktiviert ist, über die Schaltfläche **Zur visuellen Echtzeitstruktur wechseln** (linke Schaltfläche). Wir haben ein einfaches Fenster mit einer Schaltfläche erstellt, deren Code im logischen Baum vereinfacht wie folgt aussieht:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Window.Resources>
        <Style TargetType="Button">
```

```
            <Setter Property="Margin" Value="5"></Setter>
        </Style>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
        <Button x:Name="btnTest">Beispielschaltfläche</Button>
    </Grid>
</Window>
```

Aktivieren wir nun den visuellen Livebaum mit einer der beiden genannten Methoden, sieht dieser zunächst wie in Bild 4 aus.

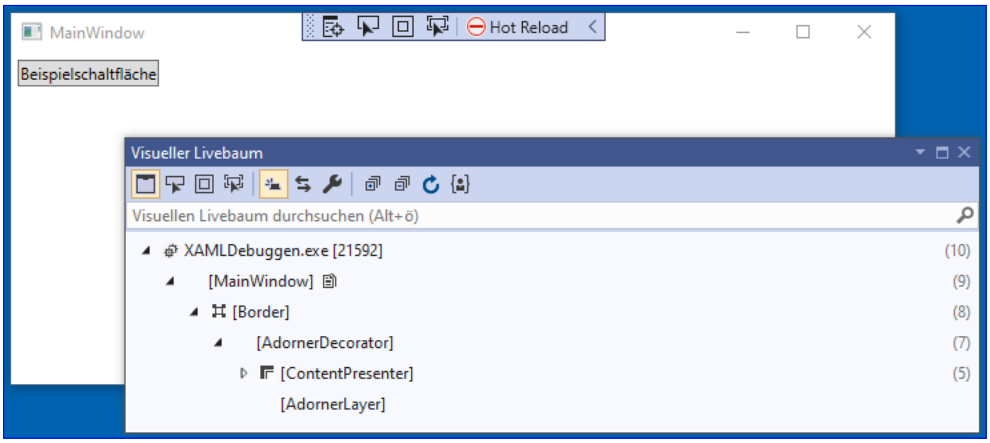


Bild 4: Visueller Livebaum für ein Fenster mit einer Schaltfläche

Nun wollen wir ein Element auswählen und klicken dazu zunächst die Schaltfläche **Element auswählen** an. Fahren wir nun über die Schaltfläche, wird diese mit einem roten, gestrichelten Rand versehen (siehe Bild 5).

Hier ist ein wenig Fingerspitzengefühl gefragt, denn zum Markieren der Schaltfläche müssen Sie genau den Bereich zwischen dem Rand und der Beschriftung der Schaltfläche erwischen. Und das ist bereits ein Beispiel für den Unterschied zwischen den Elementen

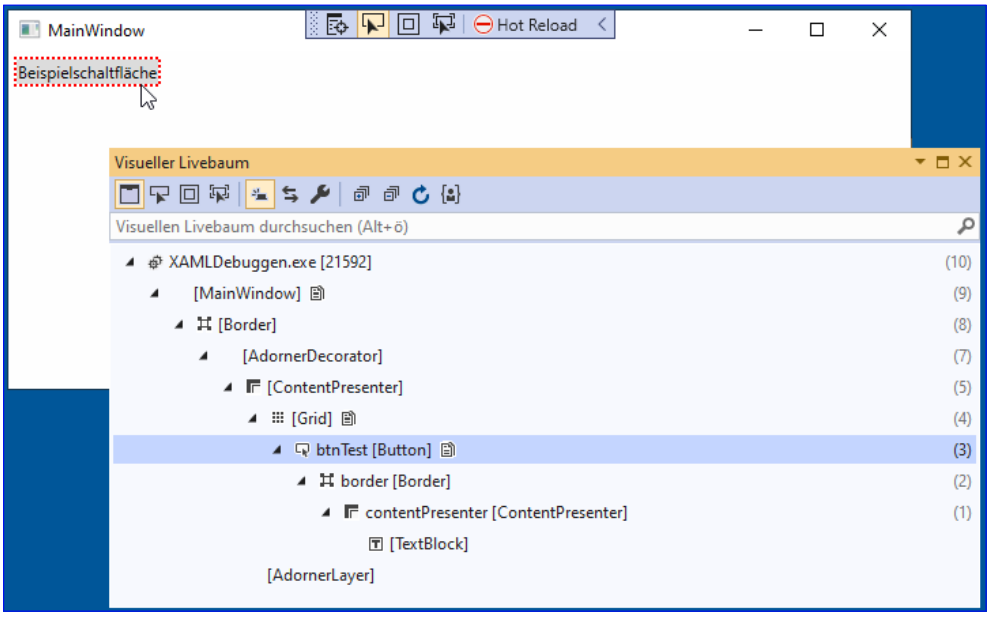


Bild 5: Visueller Livebaum nach dem Überfahren der Schaltfläche

FlowDocumente mit VB zusammenstellen

FlowDocument-Element werden per RichTextBox-Feld eingegeben, per XAML definiert oder auch per VB zusammengestellt. Letzteres ist vor allem dann interessant, wenn Sie das Dokument mit den Daten aus einer externen Quelle wie einer Datenbank füllen wollen. So können Sie beispielsweise Rechnungen erstellen, Auflistungen von Produkten, Angebote und beliebige andere Dokumente. Diese können Sie dann als PDF verschicken oder ausdrucken. Dieser Artikel zeigt die Grundlagen zum Erstellen von FlowDocument-Objekten und zum Füllen mit den gewünschten Elementen und Inhalten.

In anderen Artikeln wie **FlowDocument-Elemente mit XAML** (**www.datenbankentwickler.net/203**) oder **Das RichTextBox-Steuerelement** (**www.datenbankentwickler.net/204**) haben wir die Grundlagen zu **FlowDocument**-Elementen geliefert und zu den Steuerelementen, in denen sie angezeigt und bearbeitet werden können.

Nun wollen wir zeigen, wie Sie Elemente, die Sie in den zuvor genannten Beiträgen mit XAML-Definitionen erstellt oder über ein **RichTextBox**-Steuerelement eingegeben haben, per VB-Code erstellen können.

Vorbereitung

Die Basis unseres Beispiels definieren wir wie folgt. Wir haben ein Grid, in dessen erster Zeile ein **StackPanel**-Element liegt, das wir im Laufe des Artikels mit Schaltflächen zum Ausführen unserer Beispiele füllen werden. In der zweiten **Grid**-Zeile haben wir ein **FlowDocumentPageViewer**-Element zur Anzeige der erzeugten **FlowDocument**-Elemente definiert:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal">
        <Button x:Name="btnEinfacherText" Click="BtnEinfacherText_Click">Einfacher Text</Button>
    </StackPanel>
    <FlowDocumentPageViewer x:Name="fdp" Margin="3" Grid.Row="1"></FlowDocumentPageViewer>
</Grid>
```

Die Beispielmethoden, die durch die Schaltflächen ausgelöst werden, legen wir im Code behind-Modul des Fensters an. Für den ersten Beispieltext hinterlegen wir die folgende Prozedur in der Klasse **FlowDocumentPerVB**:

```
Public Class FlowDocumentPerVB
    Private Sub BtnEinfacherText_Click(sender As Object, e As RoutedEventArgs)
        Dim objFlowDocument As FlowDocument
```

```
        Dim objParagraph As Paragraph
        objFlowDocument = New FlowDocument
        objParagraph = New Paragraph()
        With objParagraph
            .Inlines.Add("Dies ist ein erster Absatz.")
        End With
        objFlowDocument.Blocks.Add(objParagraph)
        fdp.Document = objFlowDocument
    End Sub
End Class
```

Klicken wir nach dem Starten des Projekts auf die Schaltfläche, erhalten wir das Ergebnis aus Bild 1. Hier haben wir noch keine Absatzformate definiert. Die Zeilen, die dem Paragraph-Objekt den Text hinzufügen, können wir auch noch abkürzen durch folgenden Ausdruck:

```
objParagraph = New Paragraph(New Run("Dies ist ein erster Absatz."))
```

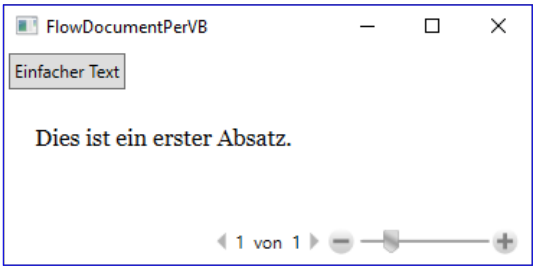


Bild 1: Erster Beispieltext

Absatz mit Formatierungen

Im zweiten Beispiel wollen wir einen Absatz mit Formatierungen versehen – der Text soll in der Schriftart **Calibri** in der Größe **24** mit fetter Auszeichnung ausgegeben werden.

Dazu verwenden wir die folgende Methode, die durch die Schaltfläche mit der Beschriftung **Text mit Format** ausgelöst wird:

```
Private Sub BtnTextMitFormat_Click(sender As Object, e As RoutedEventArgs)
    Dim objFlowDocument As FlowDocument
    Dim objParagraph As Paragraph
    objFlowDocument = New FlowDocument
    objParagraph = New Paragraph(New Run("Dies ist ein erster Absatz."))
    With objParagraph
        .FontFamily = New FontFamily("Calibri")
        .FontSize = 24
        .FontWeight = FontWeights.Bold
    End With
    objFlowDocument.Blocks.Add(objParagraph)
    fdp.Document = objFlowDocument
End Sub
```

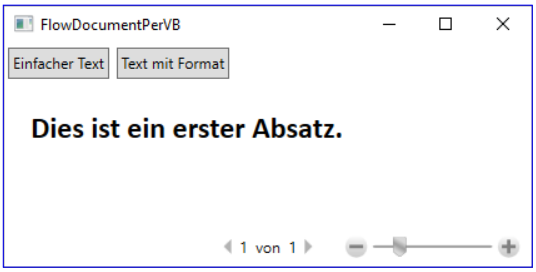


Bild 2: Text mit Formatierungen

Das Ergebnis sehen Sie in Bild 2. Die Formatierungen können wir einfach über die Eigenschaften des **Paragraph**-Objekts zuweisen. Gegenüber Programmiersprachen wie VBA lässt sich hier jedoch nicht in jedem

Fall einfach der Name der Schriftart zuweisen, sondern wir müssen dazu neue Objekte erstellen wie **FontFamily** oder Auflistungen aus Klassen verwenden wie **FontWeights.Bold**.

Tabelle mit Elementen erstellen

Das Erstellen einer einfachen Tabelle ist schon etwas aufwendiger. Schauen wir uns ein einfaches Beispiel an.

Dazu wollen wir die Tabelle aus Bild 3 erstellen – also eine Tabelle mit drei Spalten und vier Zeilen sowie Inhalt und Rahmenlinien.

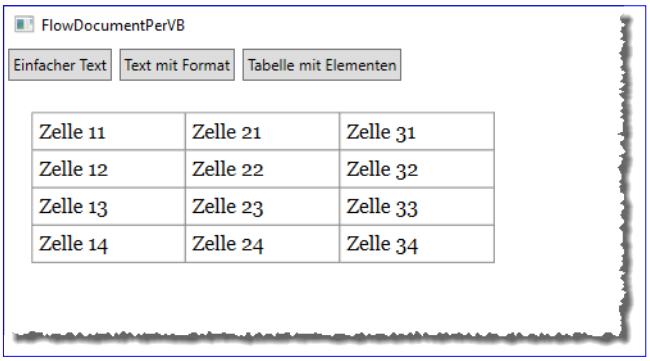


Bild 3: Erstellen einer einfachen Tabelle

Wir deklarieren zunächst einige Variablen, die wir für die Erstellung der Tabelle benötigen sowie die zwei Laufvariablen **i** und **j**, mit denen wir die Zeilen und Spalten zum Erstellen der Zellen durchlaufen wollen:

```
Private Sub BtnTabelle_Click(sender As Object, e As RoutedEventArgs)
    Dim objFlowDocument As FlowDocument
    Dim objTable As Table
    Dim objTableColumn As TableColumn
    Dim objTableRowGroup As TableRowGroup
    Dim objTableRow As TableRow
    Dim objTableCell As TableCell
    Dim objParagraph As Paragraph
    Dim i As Integer
    Dim j As Integer
```

Dann erstellen wir ein neues **FlowDocument**- und ein neues **Table**-Element. Das **Table**-Element fügen wir über die **Add**-Methode der **Blocks**-Auflistung zum **FlowDocument** hinzu:

```
    objFlowDocument = New FlowDocument
    objTable = New Table
    objFlowDocument.Blocks.Add(objTable)
```

Für das **Table**-Objekt können wir nun bereits allgemeine Einstellungen vornehmen, die sich auch auf die darunter liegenden Elemente vererben können wie hier die Schriftart und die Schriftgröße. Eine weitere Einstellung namens **CellSpacing** gibt den Abstand zwischen den Zellen an. Diesen müssen wir auf **0** einstellen, wenn wir durchgehende Rahmenlinien für die einzelnen Zeilen und Spalten erhalten wollen:

```
    With objTable
        .FontFamily = New FontFamily("Calibri")
        .FontSize = 12
```

```
        .CellSpacing = 0
    End With
```

Bevor wir Zellen anlegen können, müssen wir der Tabelle Spalten und Zeilen hinzufügen. Bei den Spalten gelingt das etwas einfacher, denn hier brauchen wir nur neue **TableColumn**-Elemente zu erstellen und diese dann über die **Add**-Methode der **Columns**-Auflistung zur Tabelle aus **objTable** hinzuzufügen. Hier fügen wir in einer **For...Next**-Schleife drei Spalten zur Tabelle hinzu:

```
    For i = 1 To 3
        objTableColumn = New TableColumn
        objTable.Columns.Add(objTableColumn)
    Next
```

Um Zeilen hinzuzufügen, benötigen wir zunächst ein **TableRow**-Objekt, das wir erstellen und der **RowGroups**-Auflistung hinzufügen:

```
    objTableRowGroup = New TableRowGroup
    objTable.RowGroups.Add(objTableRowGroup)
```

Die Tabellenzellen erstellen wir in einem Zuge mit den Tabellenzeilen. Dazu durchlaufen wir eine **For...Next**-Schleife mit der Variablen **j** für die Zeilen. Hier erstellen wir ein neues **TableRow**-Objekt und weisen es der **Rows**-Auflistung der **TableRowGroup** zu:

```
    For j = 1 To 4
        objTableRow = New TableRow
        objTableRowGroup.Rows.Add(objTableRow)
```

Damit können wir in dieser Zeile die Tabellenzellen anlegen und mit Inhalt füllen. Dabei nehmen wir direkt ein paar Formatierungen vor. Wir legen mit **Padding** den Abstand zum enthaltenen Text fest. Mit **BorderBrush** definieren wir die Rahmenfarbe. Danach folgen einige Bedingungen, mit denen wir ermitteln, in welcher Zeile und Spalte wir uns gerade befinden.

In der obersten Zeile soll der Rahmen oben und unten und in der ersten Spalte links und rechts gesetzt werden, in den folgenden Zeilen soll der Rahmen nur unten und in den folgenden Spalten nur rechts gesetzt werden. Anderenfalls erscheint der Rahmen dicker:

```
        For i = 1 To 3
            objTableCell = New TableCell()
            objTableCell.Padding = New Thickness(5)
            objTableCell.BorderBrush = Brushes.Gray
            If j = 1 Then
                If i = 1 Then
```

XML mit .NET

XML ist eines der bekanntesten Datenaustauschformate. XML bedeutet Extensible Markup Language und erlaubt das Speichern von Daten in Textdateien in einem hierarchisch strukturierten Format. Dies wird zum Beispiel in den Konfigurationsdateien eines Visual Studio-Projekts genutzt – und auch XAML, die Sprache zum Definieren von WPF-Benutzeroberflächen, ist auf XML aufgebaut. Wenn Daten zwischen zwei nicht kompatiblen Systemen ausgetauscht werden sollen, kommt oft XML zum Einsatz. Auch wenn Sie Daten von einem Webservice abrufen wollen, kommen die Daten in vielen Fällen im XML-Format. Je nach Webservice können Sie diesem auch Daten im XML-Format schicken, die dann etwa durch Speichern in einer Datenbank weiterverarbeitet werden. Grund genug, sich XML einmal genauer anzusehen und zu schauen, welche Möglichkeiten .NET für das Lesen, Bearbeiten und Erstellen von XML-Dokumenten bietet.

Grundlagen zu XML

XML-Dokumente enthalten strukturierte Daten im ASCII-Format – so können Anwendungen von verschiedenen Plattformen den Inhalt verstehen. Dementsprechend können Sie XML-Dokumente auch in einem einfachen Texteditor öffnen und bearbeiten. XML-Dokumente, die als Datei gespeichert werden, erhalten die Dateiendung **.xml**, also beispielsweise **Kunden.xml**.

Ein XML-Dokument in Visual Studio anlegen

Auch wenn XML-Dokumente keinen speziellen Editor benötigen, wollen wir die Fähigkeiten von Visual Studio nutzen. Also öffnen Sie Visual Studio und legen mit **Datei|Neu|Datei...** eine neue Datei an. Wählen Sie im nun erscheinenden Dialog **Neue Datei** den Eintag **XML-Datei** aus. Dies fügt eine neue, leere XML-Datei namens **XMLFile1.xml** hinzu, die im Verzeichnis **C:\Users\<Benutzername>\source\repos** gespeichert wird. Erfreulicherweise werden Fehler im XML-Dokument von Visual Studio im Bereich **Fehlerliste** angezeigt (siehe Bild 1). Hier sehen Sie direkt, dass es durchaus Anforderungen an XML-Dokumente gibt – in diesem Fall wird das Fehlen eines Stammelements bemängelt.

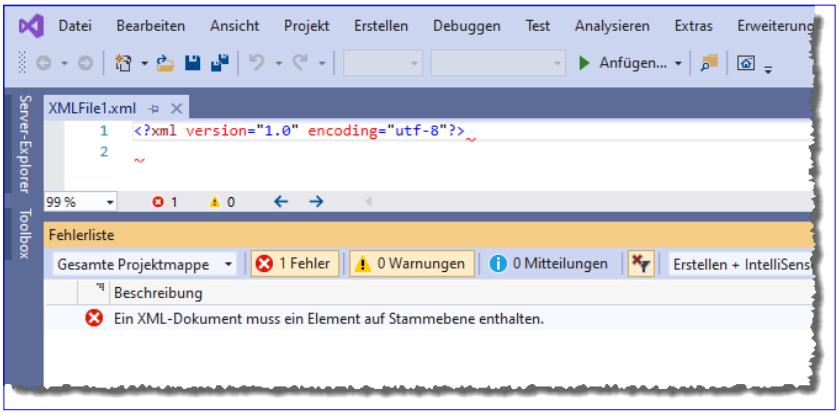


Bild 1: Ein neues, leeres XML-Dokument

XML-Dokument als solches definieren

Damit ein XML-Dokument als solches erkannt werden kann, geben wir in der ersten Zeile einen sogenannten Prolog an. Dieser hat eine andere Syntax als die übrigen XML-Zeilen, der Ihnen von der Programmiersprache PHP bekannt vorkommen könnte:

```
<?xml version="1.0" ?>
```

Hier können Sie zusätzlich noch die Kodierung des Dokuments angeben, hier zum Beispiel **utf-8**:

```
<?xml version="1.0" encoding="utf-8"?>
```

Elemente eines XML-Dokuments

Wie im Screenshot oben zu sehen, enthält ein XML-Dokument neben der Zeile mit der XML-Version und der Kodierung also mindestens ein Element auf der obersten Ebene. Ein solches Element hat entweder ein öffnendes und ein schließendes Tag-Element (Starttag und Endtag) oder ein abgeschlossenes Tag-Element (Leertag). Zwischen dem öffnenden und dem schließenden Tag-Element befinden sich untergeordnete Elemente.

Ein abgeschlossenes Tag-Element enthält keine weiteren untergeordneten Elemente. Ein Element mit öffnendem und schließendem Tag sieht wie folgt aus:

```
<Beispielelement></Beispielelement>
```

Der Name des Elements muss im öffnenden wie im schließenden Tag genau gleich geschrieben sein – das gilt auch für Groß- und Kleinschreibung. Das geschlossene Element sieht so so:

```
<Beispielelement />
```

Das Element wird durch den Schrägstrich vor dem Größer-Zeichen als abgeschlossen markiert. Beide Varianten reichen Visual Studio schon aus, um die Fehlermeldung verschwinden zu lassen. Jedes offene Element kann ein oder mehrere Unterelemente enthalten:

```
<Hauptelement>
    <Unterelement />
    <Unterelement></Unterelement>
</Hauptelement>
```

Hauptelement und **Unterelement** sind die Namen der Elemente. Die eigentlichen Daten werden an zwei Stellen gespeichert: in Attributen oder als Werte von Elementen. Attribute werden innerhalb des öffnenden Tags hinter dem Elementnamen angegeben, und zwar durch Attributname und Attributwert. Jeder Attributname darf nur einmal je Element vorkommen, der Attributwert muss in Hochkommata oder Anführungszeichen angegeben werden:

```
<Hauptelement Beispielattribut="Beispielwert"></Hauptelement>
```

Bei geschlossenen Elementen landet das Attribut ebenfalls hinter dem Elementnamen:

```
<Hauptelement Beispielattribut="Beispielwert" />
```

Außerdem können Sie für jedes Element alternativ zu weiteren untergeordneten Elementen auch eine Zeichenkette als Wert des Elements angeben:

```
<Elementname>Elementinhalt</Elementname>
```

Ein Element mit einer Zeichenkette als Inhalt kann keine weiteren Elemente enthalten.

Elementnamen verwenden

Für die Elementnamen gelten bestimmte Regeln:

- Der Elementname folgt unmittelbar auf die öffnende spitze Klammer beziehungsweise beim schließenden Tag hinter dem Schrägstrich.
- Der Elementname darf keine Leerzeichen enthalten.
- Der Elementname darf Buchstaben, Ziffern, Unterstrich, Bindestrich, Punkt und Doppelpunkt enthalten. Doppelpunkte dürfen allerdings nur hinter Namespaces angegeben werden – mehr dazu weiter unten. Als erstes Zeichen dürfen nur Buchstaben zum Einsatz kommen.
- Der Elementname darf nicht mit einer Ziffer oder mit der Zeichenfolge **xml** beginnen (letzteres wird von Visual Studion nicht bemängelt).

Wohlgeformtheit von XML-Dokumenten

Die wichtigsten Kriterien für die Wohlgeformtheit eines XML-Dokuments lauten also:

- Das Dokument muss eine Verarbeitungsanweisung enthalten (**<?xml version="1.0"?>**).
- Das Dokument besitzt in der ersten Ebene genau ein Element.
- Jedes Element muss geschlossen werden (**<Beispielelement></Beispielelement>** oder **<Beispielelement />**)
- Start- und Endtags müssen den gleichen Namen haben, auch bezüglich der Groß- und Kleinschreibung.
- Die Start- und Endtags werden paarweise verschachtelt, das heißt, alle innerhalb eines übergeordneten Elements angelegten Unterelemente müssen erst wieder geschlossen werden, bevor das übergeordnete Element geschlossen wird.

Unterschied zwischen Attributen und Elementen

Wenn Sie als Wert eine einfache Zeichenkette verwenden wollen, können Sie diese gleichwertig für ein Attribut oder für ein Unterelement angeben. Ein **Kunde**-Element mit **Vorname** und **Nachname** würde mit Attributen so aussehen:

```
<Kunde Vorname="Andre" Nachname="Minhorst"></Kunde>
```

Sie können **Vorname** und **Nachname** jedoch auch in untergeordneten Elementen darstellen:

```
<Kunde>
  <Vorname>Andre</Vorname>
  <Nachname>Minhorst</Nachname>
</Kunde>
```

Wenn ein Element allerdings einen Text enthalten soll, können Sie diesem keine weiteren Unterelemente hinzufügen – dann müssen Sie zusätzliche Informationen als Attribut angeben:

```
<Artikel Artikelnummer="123">Access im Unternehmen</Artikel>
```

Kommentare

Wenn Sie Kommentare zu einem XML-Dokument hinzufügen wollen, beginnen Sie diesen mit **<!--** und beenden ihn mit **-->**, also genau wie unter HTML:

```
<!--Dies ist ein Kommentar.-->
```

Texte in Elementen

Wenn ein Element oder ein Attribut eines Elements einen Text enthält, können Sie dafür alle möglichen Zeichen verwenden – mit wenigen Ausnahmen, nämlich dem Größer- und dem Kleiner-Zeichen, dem Und-Zeichen, dem Hochkomma und dem Anführungszeichen. Diese müssen Sie durch bestimmte andere Zeichenkombinationen ersetzen:

- Und-Zeichen (**&**): Ersetzen durch **&**
- Kleiner-Zeichen (**<**): Ersetzen durch **<**
- Größer-Zeichen (**>**): Ersetzen durch **>**
- Hochkomma (**'**): Ersetzen durch **'**
- Anführungszeichen (**"**): Ersetzen durch **"**

Sonderzeichen trotzdem verwenden mit CData

Wenn Sie Sonderzeichen wie die oben beschriebenen dennoch verwenden wollen, gibt es eine Alternative. Dazu verwenden Sie einen **CData**-Abschnitt als Inhalt eines Elements. Der **CData**-Abschnitt beginnt mit **<![CDATA[** und endet mit **]]>**.

Dazwischen können Sie Texte mit beliebigen Zeichen platzieren, also auch solche wie die oben genannten Sonderzeichen:

```
<ElementMitSonderzeichen><![CDATA[Die Sonderzeichen <, >, &, ', " sind hier erlaubt.]]></ElementMitSonderzeichen>
```

Die einzige Ausnahme ist die Zeichenfolge **]]>**, denn diese zeigt das Ende des **CData**-Abschnitts an. Der Rest des eigentlichen **CData**-Abschnitts würde dann als Fehler interpretiert werden. Die Verwendung von **CData**-Abschnitten eignet sich beispielsweise, um ein XML-Dokument innerhalb des Elements eines XML-Dokuments einzufügen.

Namespaces für eindeutige Zuordnung verwenden

Namespaces verwendet man, um Verwechslungen gleichnamiger Elemente zu verhindern. Wenn Sie beispielsweise in einem XML-Dokument sowohl Kunden als auch Lieferanten verwalten und beide enthalten Elemente namens Firma, dann könnte es beim Zugriff auf dieses Element zu Verwechslungen kommen. Es kann auch sein, dass Sie in Ihren eigenen XML-Dokumenten sehr genau darauf achten, dass alle Elemente eine eindeutige Bezeichnung haben. Vielleicht fügt aber ein Nutzer Ihres XML-Dokuments dieses mit einem anderen XML-Dokument zusammen und erhält dann doppelte Bezeichnungen für Elemente mit unterschiedlichen Bedeutungen. In folgendem Beispiel wird das deutlich:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
  <Kunden>
    <Kunde>
      <Firma>André Minhorst Verlag</Firma>
    </Kunde>
    ...
  </Kunden>
  <Lieferanten>
    <Lieferant>
      <Firma>Beispiellieferant GmbH</Firma>
    </Lieferant>
    ...
  </Lieferanten>
</Bestellverwaltung>
```

Suchen Sie hier nun nach allen Elementen mit dem Namen **Firma**, erhalten Sie sowohl die Firmenangaben für die Kunden als auch für die Lieferanten. Um das zu ändern, definieren wir Namespaces. Das können wir direkt im Stammelement **Bestellverwaltung** erledigen. Wenn wir das tun, können die Namespaces im gesamten XML-Dokument verwendet werden. Die Namespaces werden in Form von Attributen definiert, für die bestimmte Regeln gelten. Die beiden Namespaces für die Kunden und die Lieferanten würden wir wie folgt definieren:

```
<Bestellverwaltung xmlns:k="http://minhorst.com/namespaces/kunde"
                   xmlns:l="http://minhorst.com/namespaces/lieferant">
```

Der Name des Attributs zum Festlegen eines Namespaces beginnt immer mit **xmlns**. Wenn Sie einen Namespace definieren wollen, der für alle Elemente unter dem Element mit der Namespace-Definition gilt, geben Sie als Attributname nur **xmlns** an. Mit der folgenden Angabe fügen Sie alle Elemente des XML-Dokuments, die nicht anderes zugeordnet sind, dem Standardnamespace **http://minhorst.com/namespaces/bestellverwaltung** zu:

```
<Bestellverwaltung xmlns="http://minhorst.com/namespace/bestellveraltung" ...>
```

Der Wert des Attributs für einen Namespace ist eine URI. Diese muss es nicht geben, denn sie wird niemals abgefragt. Wichtig ist nur, dass diese für alle vorgesehenen Kontexte eindeutig ist.

Alle Elemente unterhalb eines Elements, für das mit **xmlns="..."** ein Standardnamespace definiert wurde, werden automatisch diesem Namespace zugeordnet. Wollen Sie ein Element einem anderen Namespace außer dem Standardnamespace zuordnen, stellen sie dem Elementnamen den Buchstaben voran, den Sie im Namen des Namespace-Attributs hinter dem Doppelpunkt angegeben haben – das sogenannte Namespace-Präfix. Für **xmlns:k="http://minhorst.com/namespaces/kunde"** wäre das also etwa der Buchstabe **k**. Diesen fügen Sie durch einen Doppelpunkt getrennt vor dem Elementnamen ein – beim Starttag und auch beim Endtag:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung xmlns="http://minhorst.com/namespaces/bestellverwaltung"
                   xmlns:k="http://minhorst.com/namespaces/kunde"
                   xmlns:l="http://minhorst.com/namespaces/lieferant">
  <k:Kunden>
    <k:Kunde>
      <k:Firma>André Minhorst Verlag</k:Firma>
    </k:Kunde>
  </k:Kunden>
  <l:Lieferanten>
    <l:Lieferant>
      <l:Firma>Beispiellieferant GmbH</l:Firma>
    </l:Lieferant>
  </l:Lieferanten>
</Bestellverwaltung>
```

Wie Sie hier sehen, haben wir den Buchstaben **k** für das **Kunden**-Element und alle Unterelemente angegeben. Hätten wir den Namespace wie folgt nur für das **Kunden**-Element angegeben, würden die untergeordneten Elemente nicht zum **k**-Namespace gehören:

```
<k:Kunden>
  <Kunde>
    <Firma>André Minhorst Verlag</Firma>
  </Kunde>
</k:Kunden>
```

Anders sieht das beim Standardnamespace aus, also bei dem Namespace, für den Sie kein Namespace-Präfix angeben. Alle Elemente, die dem Element mit der Angabe des Standardnamespaces untergeordnet sind, gehören automatisch zum Standard-namespace – außer, Sie werden explizit einem anderen Namespace zugeordnet.

XML-Dokumente schnell lesen mit XmlReader

Unter .NET gibt es verschiedene Methoden und Klassen, mit denen Sie auf die Daten in XML-Dokumenten zugreifen können. Diese nutzen Sie je nach Einsatzzweck. Wenn Sie sehr große Dateien einlesen wollen, bietet sich die XmlReader-Klasse an. Mit dieser durchlaufen Sie das XML-Dokument sequenziell, das heißt, Element für Element. Dieser Artikel zeigt, wie Sie mit den Methoden der XmlReader-Klasse auf die Elemente eines XML-Dokuments zugreifen. Außerdem schauen wir uns an, wie Sie die eingelesen Inhalte in Objekte schreiben können, über die Sie die Inhalte dann in die Tabellen einer Datenbank eintragen.

Wer unter Access/VBA auf XML-Dokumente zugegriffen hat, hat dazu in der Regel das Document Object Model verwendet und dabei die Elemente mit den gewünschten Daten gezielt angesteuert. Unter VBA gibt es noch eine weitere Technik namens SAX. Diese ist recht kompliziert und wird sehr selten verwendet. Dieser Technik kommt der hier verwendete **XmlReader** jedoch recht nahe. Seine Handhabung ist allerdings einfacher als bei SAX.

Dokument öffnen

Um ein XML-Dokument zu öffnen, benötigen wir ein Objekt auf Basis der **XMLReader**-Klasse. Diese ist Teil des Namespaces **System.XML**, den wir mit der folgenden Anweisung etwa in der Code behind-Klasse eines Fensters einer neuen Desktop-Anwendung auf Basis von XAML und VB referenzieren:

```
Imports System.Xml
```

Danach können wir auf die **XmlReader**-Klasse und ihre Elemente zugreifen. Bei der Klasse handelt es sich um eine abstrakte Klasse, das heißt, Sie können diese nicht instanzieren. Wir können aber dennoch eine Objektvariable nutzen, um auf das zu lesende XML-Dokument zuzugreifen. Um ein XML-Dokument zu öffnen, das auf der Festplatte liegt, verwenden Sie die **Create**-Methode. Dieser übergeben wir den Pfad zu der einzulesenden XML-Datei:

```
Dim objReader As XmlReader
Dim strPfad As String
strPfad = "c:\...\Kunden.xml"
objReader = XmlReader.Create(strPfad)
```

Danach befindet sich das XML-Dokument im Zugriff. Wir schauen uns die Funktionsweise der Klasse **XmlReader** anhand des folgenden XML-Dokuments an:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
  <!--Auflistung der Kunden-->
```

```
  <Kunden>
    <Kunde ID="1">
      <Firma>André Minhorst Verlag</Firma>
      <Vorname>André</Vorname>
      <Nachname>Minhorst</Nachname>
      <Beschreibung><![CDATA[Beschreibung mit CDATA]]></Beschreibung>
    </Kunde>
    <Kunde ID="2">
      <Firma>Müller GmbH</Firma>
      <Vorname>Klaus</Vorname>
      <Nachname>Müller</Nachname>
    </Kunde>
  </Kunden>
</Bestellverwaltung>
```

XML-Dokument sequenziell durchlaufen mit Read

Im Gegensatz zum Durchlaufen einer Textdatei durchlaufen wir hier nicht Zeile für Zeile, sondern Element für Element. Ein Element ist dabei nicht etwa eine Einheit wie **<Vorname>André</Nachname>**. In diesem Fall sind **<Vorname>**, **André** und **</Vorname>** jeweils ein Element. Auch der Prolog des XML-Dokuments ist ein eigenes Element. Auf diese Elemente greifen wir mit der **Read**-Methode zu. Nachdem wir diese erstmals aufgerufen haben, referenziert **objReader** das erste Element, in diesem Fall den Prolog. Danach geht es weiter mit den folgenden Elementen – also den Starttags, den enthaltenen Texten, den Endtags und auch den leeren Elementen dazwischen, den sogenannten **Whitespace**-Elementen. Wie erkennen wir nun, welchen Elementtyp **objReader** gerade geladen hat? Dazu können wir die Eigenschaft NodeType verwenden. Diesen lassen wir uns in einer ersten Schleife ausgeben:

```
Do While objReader.Read
    Debug.Print(objReader.NodeType)
Loop
```

Wir durchlaufen also alle Elemente in einer **Do While**-Schleife, deren Abbruch **objReader.Read** lautet. Die **Read**-Methode liefert den Wert **True** zurück, falls noch ein neues Element eingelesen werden konnte und den Wert **False**, wenn das Ende des Dokuments erreicht wurde. Das Ergebnis der Ausgabe ist wenig aussagekräftig, denn **NodeType** liefert einen Zahlenwert (siehe Bild 1).

Welcher Wert entspricht nun welchem Elementtyp? Das können wir uns im Objektkatalog ansehen, wo wir nach der Eigenschaft **NodeType** suchen und den Eintrag **System.Xml.XmlNodeType** vorfinden. Beim Anklicken der Elemente zeigt der Dialog auch die Zahlenwerte an, die wir mit denen unserer Ausgabe abgleichen können (siehe Bild 2).

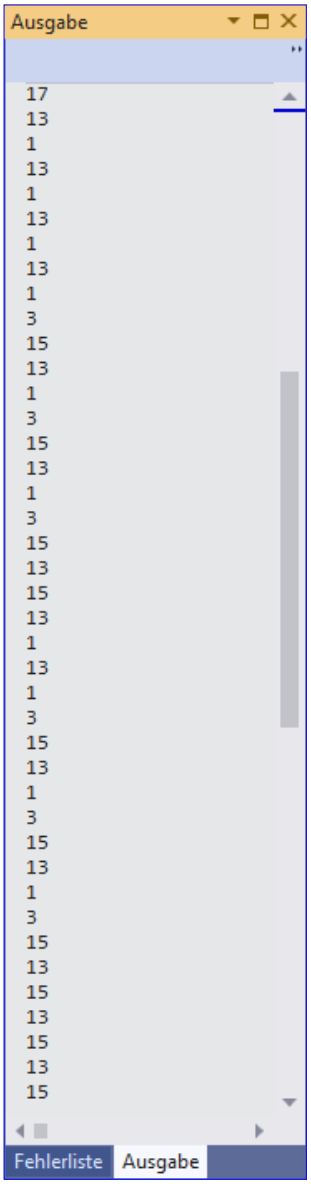


Bild 1: Elementtypen

Also reichern wir unsere **Do While**-Schleife um eine **Select Case**-Bedingung an, welche den Typ ermittelt und diesen im Direktbereich ausgibt. Für uns sind die folgenden Elementtypen interessant:

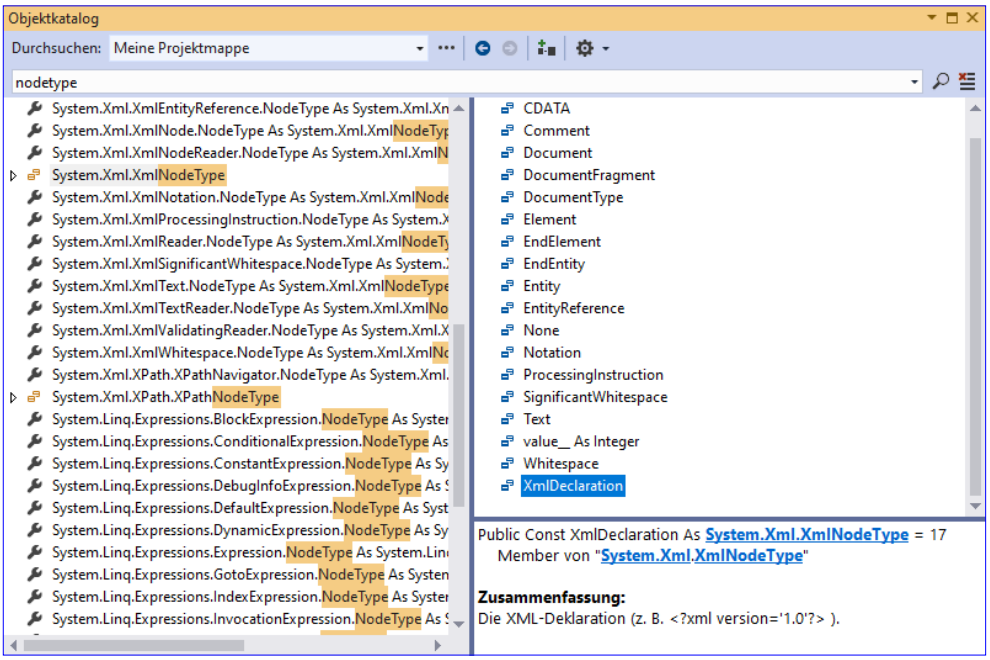


Bild 2: Elementtypen im Objektkatalog

- **Attribute**: Attribut
- **CDATA**: Block mit einem CDATA-Abschnitt
- **Comment**: Kommentar, zum Beispiel **<!--Kommentar-->**
- **Element**: Starttag, zum Beispiel **<Kunde>**
- **EndElement**: Endtag, zum Beispiel **</Kunde>**
- **Text**: Text in einem Element
- **Whitespace**: Leerraum zwischen Elementen
- **XmlDeclaration**: XML-Deklarationszeile, zum Beispiel **<?xml version='1.0'>**

In der folgenden Schleife geben wir die Typen der Elemente unseres Dokuments aus:

```
Do While objReader.Read
    Select Case objReader.NodeType
        Case XmlNodeType.XmlDeclaration
            Debug.Print("XmlDeclaration")
        Case XmlNodeType.Element
            Debug.Print("Element")
        Case XmlNodeType.EndElement
            Debug.Print("EndElement")
        Case XmlNodeType.Whitespace
            Debug.Print("Whitespace")
```

```
        Case XmlNodeType.Text
            Debug.Print("Text")
        Case Else
            Debug.Print(objReader.NodeType)
    End Select
Loop
```

Nun wollen wir für die unterschiedlichen Elementtypen entsprechende Ausgaben bewirken. Diese erhalten wir über die Eigenschaften des aktuellen Elements. **NodeType** war die erste Eigenschaft, die Sie kennengelernt haben. Es gibt jedoch noch weitere Eigenschaften, die für uns interessant sein können:

- **Depth**: Tiefe des aktuellen Elements in der Hierarchie
- **IsStartElement**: Gibt an, ob das aktuelle Element ein Starttag ist. Der folgende Code gibt beispielsweise alle Namen von Starttags aus:

```
Do While objReader.Read
    If (objReader.IsStartElement) Then
        Debug.Print(objReader.Name)
    End If
Loop
```

- **Name**: Name des Elements. Vor allem interessant bei den Elementtypen **Element** und **EndElement**.
- **NodeType**: Typ des Elements
- **Value**: Wert des Elements, vor allem interessant bei den Elementtypen **Text** oder **Comment**.
- **IsEmptyElement**: Gibt an, ob das Element einen Wert enthält.

Attribute erkennen mit HasAttributes und AttributeCount

Möglicherweise ist Ihnen aufgefallen, das wir zwar die Inhalte der Elemente ausgegeben haben, jedoch noch nicht die der Attribute. Diese erhalten wir auf einem anderen Weg. Grundsätzlich finden wir Attribute nur bei Elementen des Typs **Element**. Hier können wir mit der Eigenschaft **HasAttributes** prüfen, ob überhaupt Attribute vorliegen. Ist das der Fall, lässt sich mit der Eigenschaft **AttributeCount** die Anzahl der Attribute ermitteln. Wir können diese aber auch direkt in einer **Do While**-Schleife durchlaufen.

Dazu verwenden wir wieder eine Abbruchbedingung, die gleichzeitig das nächste Attribut einliest. Diese lautet **MoveToNextAttribut**. Die Methode liefert den Wert **True**, wenn sie ein weiteres Attribut gefunden hat und **False**, wenn kein weiteres Attribut vorliegt. Den Namen des Attributs ermitteln Sie dann wiederum mit der Eigenschaft **Name**. Den Wert des Attributs liefert die Eigenschaft **Value**.

Mit diesem Wissen können wir die **Do While**-Schleife zum Durchlaufen der Elemente nun weiter verfeinern und diese zur Ausgabe der relevanten Informationen nutzen. Dabei ermitteln wir nun den Elementtyp und geben davon abhängig die gewünschten Informationen aus. Wir beginnen mit der Deklarationszeile:

```
Do While objReader.Read
    Select Case objReader.NodeType
        Case XmlNodeType.XmlDeclaration
            Debug.Print("XmlDeclaration: " + objReader.Value)
```

Im Falle eines Elements des Typs **Element** geben wir den Namen des Elements aus und untersuchen, ob das Element noch Attribute enthält. Dazu prüfen wir den Wert der Eigenschaft **HasAttributes**. Hat diese den Wert **True**, durchlaufen wir die Attribute in einer **Do While**-Schleife solange, bis die Methode **MoveToNextAttribute** den Wert **False** zurückgibt. In der Schleife geben wir den Namen und den Wert des Attributs aus:

```
        Case XmlNodeType.Element
            Debug.Print("Element: " + objReader.Name)
            If objReader.HasAttributes Then
                Do While objReader.MoveToNextAttribute
                    Debug.Print(objReader.Name + ": " + objReader.Value)
                Loop
            End If
```

Für den Typ **EndElement** geben wir nur den Namen des Elements aus:

```
        Case XmlNodeType.EndElement
            Debug.Print("EndElement:" + objReader.Name)
```

Auch den Text eines Elements geben wir im Direktbereich aus – genauso wie den Inhalt eines **CDATA**-Abschnitts:

```
        Case XmlNodeType.Text
            Debug.Print("Text: " + objReader.Value)
        Case XmlNodeType.CDATA
            Debug.Print("CDATA: " + objReader.Value)
```

Leerräume zwischen Elementen wollen wir schlicht übergehen:

```
        Case XmlNodeType.Whitespace
```

Schließlich geben wir auch den Inhalt von Kommentaren aus:

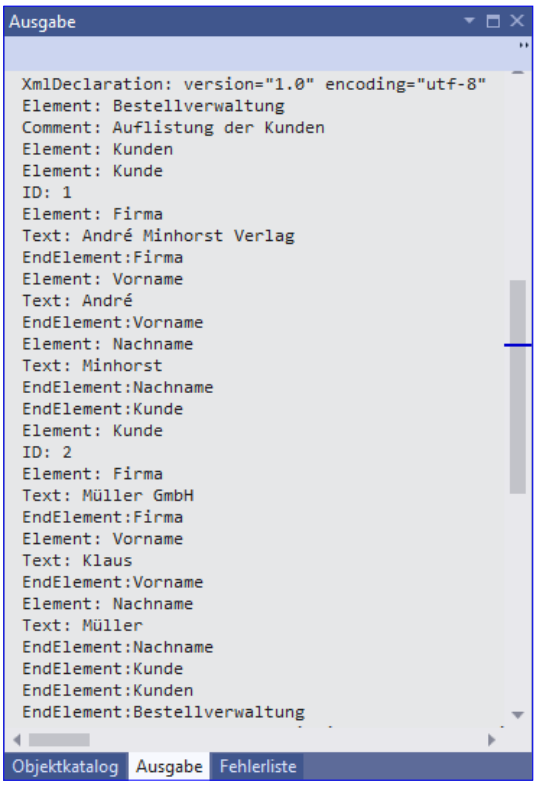


Bild 3: Ausgabe der wichtigsten Informationen

Bestellungen per XmlReader einlesen

Mit dem XmlReader durchlaufen Sie ein XML-Dokument Element für Element. Da dies recht unflexibel ist, müssen wir beim Einlesen verschachtelter XML-Dokumente umso flexibler programmieren. Dieser Artikel zeigt, wie Sie die Daten eines XML-Dokuments mit Kunden, Bestellungen, Bestellpositionen und Artikeln per XmlReader zuerst in ein Entity Data Model einlesen und die Daten von dort in den zugrunde liegenden Tabellen speichern.

Zu importierendes XML-Dokument

Das XML-Dokument, dessen Daten wir importieren wollen, hat ein Stammelement namens Bestellverwaltung. Darunter finden wir ein Auflistungselement namens **Kunden**, das alle **Kunde**-Elemente enthält. Das **Kunde**-Element hat ein Attribut zum Speichern der **ID** sowie einige untergeordnete Felder mit Firma, Vorname, Nachname und Adresse des Kunden:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
  <Kunden>
    <Kunde ID="1">
      <Firma>André Minhorst Verlag</Firma>
      <Vorname>André</Vorname>
      <Nachname>Minhorst</Nachname>
      <Strasse>Borkhofer Str. 17</Strasse>
      <PLZ>47137</PLZ>
      <Ort>Duisburg</Ort>
```

Unterhalb des **Kunde**-Elements finden wir außerdem für jeden Kunden ein Auflistungselement namens **Bestellungen**. Dieses nimmt ein oder mehrere **Bestellung**-Elemente auf. Diese enthalten ebenfalls in einem Attribut namens **ID** einen eindeutigen Identifizierer. Darunter finden wir ein Element namens **Bestelldatum** mit dem Datum der Bestellung sowie ein weiteres Auflistungselement namens **Bestellpositionen**. Dieses enthält für jede Position ein Element des Typs **Bestellposition** mit dem Attribut **ID**.

Jede Bestellposition weist die vier Elemente **Artikel**, **Einzelpreis**, **Menge** und **Mehrwertsteuersatz** auf:

```
      <Bestellungen>
        <Bestellung ID="1">
          <Bestelldatum>2020-11-17T12:00:00+00:00</Bestelldatum>
          <Bestellpositionen>
            <Bestellposition ID="1">
              <Artikel>Access im Unternehmen</Artikel>
              <Einzelpreis>124</Einzelpreis>
              <Menge>1</Menge>
```

```
            <Mehrwertsteuersatz>7</Mehrwertsteuersatz>
          </Bestellposition>
```

Von den meisten dieser Elemente gibt es noch weitere Exemplare, die wir hier aus Platzgründen nicht explizit abbilden:

```
          ...
        </Bestellpositionen>
      </Bestellung>
      ...
    </Bestellungen>
  </Kunde>
  ...
</Kunden>
</Bestellverwaltung>
```

Entity Framework vorbereiten

Nach dem Erstellen des WPF-Projekts mit Visual Basic fügen wir das Entity Framework hinzu. Dazu klicken Sie mit der rechten Maustaste auf das Projekt-Element im Projektmappen-Explorer und wählen aus dem Kontextmenü den Eintrag **Hinzufügen|Neues Element...** aus. Im Dialog **Neues Element hinzufügen** selektieren Sie **ADO.NET Entity Data Model** und nennen dieses **BestellverwaltungContext**.

Im folgenden Dialog **Assistent für Entity Data Model** wählen Sie die Option **Leeres Code First-Modell** aus. Dies legt die Elemente für das Entity Framework an und legt in der Datei **App.config** einen Connectionstring für eine Datenbank namens **Bestellverwaltung.BestellverwaltungContext** fest, wobei standardmäßig der Server **(LocalDb)\MSSQLLocalDB** verwendet wird.

Die Klasse **BestellverwaltungContext.vb** ergänzen wir wie folgt und definieren so die drei DbSet-Elemente, welche die Auflistungen für Kunden, Bestellungen und Bestellpositionen enthalten:

```
Public Class BestellverwaltungContext
    Inherits DbContext

    Public Sub New()
        MyBase.New("name=BestellverwaltungContext")
    End Sub

    Public Overridable Property Bestellpositionen() As DbSet(Of Bestellposition)
    Public Overridable Property Bestellungen() As DbSet(Of Bestellung)
    Public Overridable Property Kunden() As DbSet(Of Kunde)

End Class
```

Die Entitätsklasse für die Kunden hinterlegen wir wie folgt in der Klassendatei **Kunde.vb** in einem neuen Ordner namens **DataModel**. Die Klassen enthalten alle Informationen, die für das Erstellen in Form der Tabellen einer Datenbank notwendig sind. Dazu sind auch jeweils die **DataAnnotations**-Namespaces erforderlich:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

<Table("Kunden")>
Public Partial Class Kunde
    Public Property ID As System.Int32
    <StringLength(255)>
    Public Property Firma As System.String
    <StringLength(255)>
    Public Property Vorname As System.String
    <StringLength(255)>
    Public Property Nachname As System.String
    <StringLength(255)>
    Public Property Strasse As System.String
    <StringLength(255)>
    Public Property PLZ As System.String
    <StringLength(255)>
    Public Property Ort As System.String
    Public Overridable Property Bestellungen As ICollection(Of Bestellung)
End Class
```

Die Klasse für die Bestellpositionen heißt **Bestellung.vb** und enthält die folgende Beschreibung der Felder:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

<Table("Bestellungen")>
Public Partial Class Bestellung
    Public Property ID As System.Int32
    Public Property KundeID As System.Int32
    <Column(TypeName:="datetime2")>
    Public Property Bestelldatum As System.DateTime
    Public Overridable Property Bestellpositionen As ICollection(Of Bestellposition)
    Public Overridable Property Kunde As Kunde
End Class
```

Fehlt noch die Klasse für die Bestellpositionen, die wir **Bestellposition.vb** nennen:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

<Table("Bestellpositionen")>
Public Partial Class Bestellposition
    Public Property ID As System.Int32
    <StringLength(255)>
    Public Property Artikel As System.String
    Public Property Einzelpreis As System.Decimal
    <StringLength(255)>
    Public Property Menge As System.String
    Public Property Mehrwertsteuersatz As System.Decimal
    Public Property BestellungID As System.Int32
    Public Overridable Property Bestellung As Bestellung
End Class
```

Datenbank anlegen

Zeigen Sie nun die Paket-Manager-Konsole an (Menüeintrag **Ansicht|Weitere Fenster|Paket-Manager-Konsole**) und geben Sie dort nacheinander die drei folgenden Befehle ein:

```
enable-migrations
add-migration init
update-database
```

Damit erstellen Sie die Datenbank auf Basis der Klassen und DbSets des Entity Data Models in der Datenbank, die in der Verbindungszeichenfolge angegeben ist.

System.XML-Namespace verfügbar machen

In der XAML-Datei **MainWindow.xaml** wollen wir das Einlesen mit einer Schaltfläche starten. Diese deklarieren wir wie folgt innerhalb des **Grid**-Elements:

```
<Button x:Name="btnKundenUndBestellungenInKlassen" Click="btnKundenUndBestellungenInKlassen_Click">Kunden und Bestellungen in Klassen</Button>
```

In der Code behind-Klasse **MainWindow.xaml.vb** hinterlegen wir einen Verweis auf den Namespace **System.Xml**:

```
Imports System.Xml
```

Klasse initialisieren per Konstruktor

Der Klasse fügen wir eine Variable namens **dbContext** hinzu, mit der wir auf den Datenbankkontext **BestellverwaltungContext** zugreifen:

```
Class MainWindow

    Dim dbContext As BestellverwaltungContext
```

In der Konstruktormethode **New** initialisieren wir den Datenbankkontext und sorgen mit **InitializeComponent** dafür, dass die Elemente der XAML-Datei umgesetzt werden:

```
    Public Sub New()
        dbContext = New BestellverwaltungContext
        InitializeComponent()
    End Sub
```

Kunden mit XmlReader einlesen

Die durch die Schaltfläche **cmdKundenUndBestellungenInKlassen** ausgelöste Methode sieht wie folgt aus. Hier deklarieren wir eine Variable für die **XmlReader**-Klasse und legen den Pfad zur einzulesenden XML-Datei fest:

```
    Private Sub btnKundenUndBestellungenInKlassen_Click(sender As Object, e As RoutedEventArgs)
        Dim objReader As XmlReader
        Dim strPfad As String
        Dim objKunde As Kunde
        strPfad = "C:\Users\User\Dropbox\Daten\Fachmagazine\VisualStudioDatenbankentwicklung\2020\04\" _
            & "BestellungenPerXMLReader\Bestellverwaltung.xml"
```

Dann öffnen wir die XML-Datei mit der **Create**-Methode der **XmlReader**-Klasse:

```
        objReader = XmlReader.Create(strPfad)
```

Anschließend starten wir direkt mit dem Einlesen, indem wir in einer **Do While**-Schleife so oft die Methode **Read** aufrufen, bis diese kein neues Element mehr findet. Bis dahin jedoch werten wir den Elementyp über die Eigenschaft **NodeType** in einer ersten **Select Case**-Bedingung aus.

Finden wir hier ein **Kunde**-Element, heißt das, dass die Definition eines neuen Kunden begonnen hat. Die Daten dieses Kunde-Elements befinden sich in den folgenden, untergeordneten Elementen. Also legen wir ein neues Objekt der Klasse **Kunde** an und referenzieren es mit **objKunde**:

```
        Do While objReader.Read
            Select Case objReader.NodeType
                Case XmlNodeType.Element
                    Select Case objReader.Name
                        Case "Kunde"
                            objKunde = New Kunde
```

XML-Dokumente erstellen mit XmlWriter

Im Artikel »XML-Dokumente schnell lesen mit XmlReader« haben Sie die Klasse XmlReader kennengelernt, mit der Sie schnell durch XML-Dokumente navigieren können. Damit können Sie zum Beispiel Exporte aus anderen Anwendungen in Ihre .NET-Anwendung importieren. Aber was, wenn Sie selbst einmal einen Export mit Ihren Daten im XML-Format bereitstellen wollen? Dazu gibt es eine passende Klasse namens XmlWriter. Wie Sie damit XML-Dokumente erstellen und mit den gewünschten Daten füllen, erfahren Sie im vorliegenden Artikel.

Namespace System.Xml referenzieren

Genau wie die Klasse **XmlReader** ist auch **XmlWriter** Bestandteil des Namespaces **System.Xml**. Diesen referenzieren wir deshalb wie folgt im Kopf der Code behind-Klasse unseres Beispielfensters **MainWindow.xaml**:

```
Imports System.Xml
```

Leeres XML-Dokument erstellen mit Create

Um ein XML-Dokument als Datei zu erstellen, rufen Sie die Methode **Create** des **XmlWriter**-Objekts auf und übergeben dieser den Pfad der zu erstellenden XML-Datei. Wenn Sie einfach nur den Dateinamen angeben, wird die Datei im gleichen Verzeichnis erstellt wie die **.exe**-Datei des Projekts:

```
Dim objWriter As XmlWriter
objWriter = XmlWriter.Create("XmlWriter.xml")
objWriter.Close()
```

Dies legt allerdings nur ein leeres Dokument an.

XML-Dokument mit Element erstellen

Um ein Dokument mit dem Prolog und einem ersten Element zu erstellen, bedarf es einiger weiterer Zeilen. Die Methoden **WriteStartDocument** und **WriteEndDokument** starten den Schreibprozess und beenden diesen wieder. Die Methode **WriteStartElement** fügt ein Starttag namens **Bestellverwaltung** hinzu und **WriteEndElement** schließt das Element mit dem Endtag ab. Dabei müssen Sie **WriteEndElement** keinen Parameter mehr übergeben – die Methode bezieht sich immer auf das zuletzt geöffnete Starttag:

```
Private Sub btnDokumentMitElementErstellen_Click(sender As Object, e As RoutedEventArgs)
    Dim objWriter As XmlWriter
    objWriter = XmlWriter.Create("XmlWriter.xml", objSettings)
    objWriter.WriteStartDocument()
    objWriter.WriteStartElement("Bestellverwaltung")
    objWriter.WriteEndElement()
```

```
    objWriter.WriteEndDocument()
    objWriter.Close()
End Sub
```

Das Ergebnis ist das folgende XML-Dokument:

```
<?xml version="1.0" encoding="utf-8"?><Bestellverwaltung />
```

Hier finden wir keinen Zeilenumbruch, was wir als Erstes ändern möchten

Eigenschaften für das Erstellen festlegen

Die **Create**-Methode erwartet in einer Überladung einen zweiten Parameter des Typs **XmlWriterSettings**. Wir erstellen also zuvor ein solches Objekt und übergeben es dann – allerdings nicht, ohne einige Eigenschaften dieser Klasse zu nutzen:

- **Encoding**: Gibt an, ob das Dokument mit UTF8, Unicode oder einer anderen Kodierung erstellt werden soll. Die verschiedenen Werte stellt **System.Text.Encoding** zur Verfügung, was Sie auch per IntelliSense nutzen können. Standardwert ist **UTF8**.
- **Indent**: Boolean-Wert, der angibt, ob Einrückungen hinzugefügt werden sollen.
- **IndentChar**: String-Wert, der die für einen Einrückungsschritt zu verwendende Zeichenkette enthält. Wenn Sie das Tabulatorzeichen verwenden wollen, geben Sie etwa **ControlChars.Tab** an, wenn Sie eine bestimmte Anzahl Leerzeichen verwenden wollen, nutzen Sie einfach die Zeichenkette **" "**.
- **NewLineChars**: Gibt an, welches Zeichen für Zeilenumbrüche verwendet werden soll (Standard: **Chr(13) + Chr(10)**).
- **NewLineOnAttributes**: Gibt an, ob Attribute in einer eigenen Zeile ausgegeben werden sollen. Dient bei vielen Attributen gegebenenfalls der Übersichtlichkeit beim manuellen Lesen.
- **OmitXmlDeclaration**: Gibt mit **True** an, ob der Prolog weggelassen werden soll.

Im folgenden Beispiel nutzen wir die beiden Eigenschaften **Indent** und **IndentChar**, um festzulegen, dass jeder Einzug durch vier Leerzeichen realisiert werden soll. Außerdem haben wir ein paar weitere Elemente unterhalb des **Bestellverwaltung**-Elements hinzufügt, damit die Einzüge im erstellten Dokument zur Geltung kommen:

```
Private Sub btnDokumentMitEinstellungen_Click(sender As Object, e As RoutedEventArgs)
    Dim objWriter As XmlWriter
    Dim objSettings As XmlWriterSettings
    objSettings = New XmlWriterSettings
    With objSettings
        .Indent = True
        .IndentChars = "    "
```

```
    End With
    objWriter = XmlWriter.Create("XmlWriter.xml", objSettings)
    objWriter.WriteStartDocument()
    objWriter.WriteStartElement("Bestellverwaltung")
    objWriter.WriteStartElement("Kunden")
    objWriter.WriteEndElement()
    objWriter.WriteEndElement()
    objWriter.WriteEndDocument()
    objWriter.Close()
End Sub
```

Das Ergebnis sieht nun wie folgt aus:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
    <Kunden>
        <Kunde />
    </Kunden>
</Bestellverwaltung>
```

Mehrere Elemente auf einer Ebene

Bisher haben wir mit **WriteStartElement**/**WriteEndElement** immer Elemente erstellt, die unter einem anderen Element angeordnet werden. Wenn Sie beispielsweise zwei **Kunde**-Elemente im Element **Kunden** anlegen wollen, müssen Sie einfach das erste **Kunde**-Element abschließen und dann das nächste **Kunde**-Element beginnen:

```
---
objWriter.WriteStartElement("Kunden")
objWriter.WriteStartElement("Kunde")
objWriter.WriteEndElement()
objWriter.WriteStartElement("Kunde")
objWriter.WriteEndElement()
objWriter.WriteStartElement("Kunde")
objWriter.WriteEndElement()
objWriter.WriteEndElement()
---
```

Elemente mit Inhalt erstellen mit WriteElementString

Wenn Sie dem **Kunde**-Element nun Elemente wie **Firma**, **Vorname** und **Nachname** mit den entsprechenden Werten hinzufügen wollen, verwenden Sie nicht die **WriteStartElement**-Methode, sondern die **WriteElementString**-Methode. Diese erwartet als ersten Parameter den Namen des zu erstellenden Elements und als zweiten Parameter den Wert. Wenn wir dies nach dem Erstellen des **Kunde**-Starttags erledigen, werden **Firma**, **Vorname** und **Nachname** wie folgt dem Element **Kunde** untergeordnet:

Bestellungen per XmlWriter in XML exportieren

Im Artikel »XML-Dokumente erstellen mit XmlWriter« haben wir die grundlegenden Techniken beschrieben, mit denen Sie XML-Dokumente mit der XmlWriter-Klasse erzeugen und mit den gewünschten Daten füllen. Im vorliegenden Artikel liefern wir ein Praxisbeispiel dazu und wollen die Daten eines Entity Data Models, das aus einer SQL Server-Datenbank befüllt wird, in ein XML-Dokument schreiben. Dieses können Sie dann mit der Lösung aus dem Artikel »Bestellungen per XmlReader einlesen« wieder einlesen und somit Daten zwischen zwei Anwendungen per XML-Dokument übertragen.

Im Artikel **Bestellungen per XmlReader einlesen** (**www.datenbankentwickler.net/234**) haben wir gezeigt, wie Sie Daten aus einem XML-Dokument in ein Entity Data Model einlesen und diese dann in der damit verknüpften SQL Server-Datenbank speichern. Im vorliegenden Artikel wollen wir nun den umgekehrten Weg gehen. Dazu nutzen wir das im oben genannten Artikel erstellte Projekt. Dort haben wir schon beschrieben, wie das Entity Data Model, bestehend aus den drei Entitäten **Kunde**, **Bestellung** und **Bestelldetail** aufgebaut ist. Darüber wollen wir nun die Daten der dahinter stehenden Tabellen in das Entity Data Model einlesen und diese dann in ein neues XML-Dokument exportieren.

Damit lernen Sie nicht nur die Möglichkeiten der **XmlWriter**-Klasse besser kennen, sondern es ist auch eine schöne Fingerübung für den Umgang mit den Daten eines Entity Data Models.

Dokument erstellen und Schreiben der Daten initialisieren

Die Schaltfläche **btnKundenUndBestellungenInXMLDokument** löst die folgende Methode aus. Diese deklariert neben einer Variablen für die Zieldatei noch ein Listenelement für die Kunden sowie ein **XmlWriter**- und ein **XmlWriterSettings**-Objekt. Letzteres erstellt die Methode zuerst und legt fest, dass wir Einrückungen mit vier Leerzeichen verwenden wollen:

```
Private Sub btnKundenUndBestellungenInXMLDokument_Click(sender As Object, e As RoutedEventArgs)
    Dim strXmlDatei As String
    Dim Kunden As List(Of Kunde)
    Dim objWriter As XmlWriter
    Dim objSettings As XmlWriterSettings
    objSettings = New XmlWriterSettings
    With objSettings
        .Indent = True
        .IndentChars = "    "
    End With
```

Danach ermitteln wir mit der Funktion **ZieldateiErmitteln** den Pfad zu der zu erstellenden Datei und erstellen unter Angabe dieses Pfades mit der **Create**-Methode das **XmlWriter**-Objekt:

```
    strXmlDatei = ZieldateiErmitteln()
```

```
    objWriter = XmlWriter.Create(strXmlDatei, objSettings)
```

Nach dem notwendigen Aufruf der **WriteStartDocument**-Methode füllen wir alle Kunden der Tabelle **Kunden** über das **DbSet** namens **Kunden** des Entity Data Models in die Liste **Kunden**. Damit und mit **objWriter** als Parameter rufen wir eine weitere Methode namens **KundenInXMLSchreiben** auf, welche die Kunden zum XML-Dokument hinzufügen soll:

```
    objWriter.WriteStartDocument()
    Kunden = New List(Of Kunde)(dbContext.Kunden)
    KundenInXMLSchreiben(objWriter, Kunden)
    objWriter.WriteEndElement()
    objWriter.WriteEndDocument()
    objWriter.Close()
End Sub
```

Zielpfad mit SaveFileDialog auswählen

Die oben verwendete Funktion **ZieldateiErmitteln** verwendet die **SaveFileDialog**-Klasse, um den Zielpfad zu ermitteln. Dazu stellen wir als Standard-Dateiendung den Wert **.xml**, als Filter ebenfalls **.xml** und als Startverzeichnis das Verzeichnis der **.exe**-Datei der Anwendung ein. Die **ShowDialog**-Methode öffnet den Dialog und der Code läuft erst weiter, wenn dieser geschlossen wird. Liefert **ShowDialog** den Wert **True**, wurde eine Datei ausgewählt. Diese speichert die Methdoe in **strZieldatei** gespeichert und gibt den enthalten Wert dann mit **Return** an die aufrufende Routine zurück:

```
Private Function ZieldateiErmitteln()
    Dim objFileDialog As SaveFileDialog
    Dim strZieldatei As String
    objFileDialog = New SaveFileDialog
    With objFileDialog
        .DefaultExt = ".xml"
        .Filter = "XML-Dokumente (.xml)|*.xml"
        .InitialDirectory = System.AppDomain.CurrentDomain.BaseDirectory
    End With
    If (objFileDialog.ShowDialog) = True Then
        strZieldatei = objFileDialog.FileName
    End If
    Return strZieldatei
End Function
```

Kunden in die XML-Datei schreiben

Die von der Hauptmethode aufgerufene Routine KundenInXMLSchreiben erwartet zwei Parameter: das **XmlWriter**-Objekt aus **objWriter** sowie die dort ermittelte Liste der Kunden aus der Datenbank. Wichtig ist, dass Sie den Parameter **objWriter** mit dem Schlüsselwort **ByRef** auszeichnen. So wirken sich die Änderungen an **objWriter** aus dieser Routine auch auf die Variable aus, die von der aufrufenden Methode aus übergeben wurde.

Die Routine **KundenInXMLSchreiben** erstellt en neues **Kunden**-Element in **objWriter** und durchläuft dann in einer **For Each**-Schleife alle Elemente der Auflistung Kunden. Dabei legt sie unter dem **Kunden**-Element jeweils ein neues **Kunde**-Element an. Das **Kunde**-Element erhält dann ein Attribut mit dem Namen **ID** und dem Primärschlüsselwert des Kunden als Wert. Dann fügt die Routine weitere Elemente mit den Inhalten der Eigenschaften **Firma**, **Vorname**, **Nachname**, **Strasse**, **PLZ** und **Ort** hinzu. Schließlich liest sie alle **Bestellung**-Elemente des Kunden in die Liste **Bestellungen** ein. Diese übergibt sie neben **objWriter** an eine weitere Methode, die für jeden Kunden einmal aufgerufen wird und die **BestellungenInXMLSchreiben** heißt:

```
Private Sub KundenInXMLSchreiben(ByVal objWriter As XmlWriter, Kunden As List(Of Kunde))
    Dim objKunde As Kunde
    Dim Bestellungen As List(Of Bestellung)
    objWriter.WriteStartElement("Kunden")
    For Each objKunde In Kunden
        objWriter.WriteStartElement("Kunde")
        objWriter.WriteAttributeString("ID", objKunde.ID)
        objWriter.WriteElementString("Firma", objKunde.Firma)
        objWriter.WriteElementString("Vorname", objKunde.Vorname)
        objWriter.WriteElementString("Nachname", objKunde.Nachname)
        objWriter.WriteElementString("Strasse", objKunde.Strasse)
        objWriter.WriteElementString("PLZ", objKunde.PLZ)
        objWriter.WriteElementString("Ort", objKunde.Ort)
        Bestellungen = New List(Of Bestellung)(From b In dbContext.Bestellungen
                                               Where b.Kunde.ID = objKunde.ID)
        BestellungenInXMLSchreiben(objWriter, Bestellungen)
        objWriter.WriteEndElement()
    Next
    objWriter.WriteEndElement()
End Sub
```

Mit dieser ersten Methode haben wir bereits die Grundstruktur des XML-Dokuments hergestellt. Was nun noch fehlt, sind die Bestellungen der Kunden:

```
<?xml version="1.0" encoding="utf-8"?>
<Kunden>
    <Kunde ID="29">
        <Firma>André Minhorst Verlag</Firma>
        <Vorname>André</Vorname>
        <Nachname>Minhorst</Nachname>
        <Strasse>Borkhofer Str. 17</Strasse>
        <PLZ>47137</PLZ>
        <Ort>Duisburg</Ort>
        ... Bestellungen
```

XML lesen mit dem Document Object Model

Mit der Klasse XmlReader, die wir im Artikel »XML-Dokumente schnell lesen mit XmlReader« vorgestellt haben, können Sie XML-Dokumente schnell sequenziell durchlaufen. Dies eignet sich vor allem für den Zugriff auf sehr große XML-Dokumente. Wenn Sie selektiver auf die Inhalte des XML-Dokuments zugreifen wollen, können Sie das Document Object Model nutzen. Implementierungen dieser Schnittstelle, die kurz DOM genannt wird, gibt es für fast alle Programmiersprachen – so auch für die von .NET. Dieser Artikel zeigt, wie Sie die Klassen, Eigenschaften und Methoden des Document Object Models für den Zugriff auf XML-Dokumente nutzen.

DOM im Gegensatz zu XmlReader und XmlWriter

Einer der wichtigsten Unterschiede des Document Object Model im Vergleich zur **XmlReader**-Klasse ist, dass Sie damit nicht nur lesend, sondern auch schreibend auf die XML-Dokumente zugreifen können. Das heißt, wenn Sie ein Element gefunden haben, dessen Inhalt Sie ändern möchten, dann können Sie das direkt mit den Methoden des Document Object Models erledigen. Sie können damit nicht nur Elemente bearbeiten, sondern auch neue Elemente hinzufügen oder bestehende Elemente löschen. Und natürlich können Sie damit auch komplett neue XML-Dokumente erstellen und mit Daten füllen. Dazu steht allerdings auch die Klasse **XmlWriter** zur Verfügung, die wir im Artikel **XML-Dokumente erstellen mit XmlWriter** vorgestellt haben.

Mit dem Document Object Model laden Sie das komplette XML-Dokument in den Speicher, weshalb einzelne Elemente gezielt angesteuert werden können. Während beim XmlReader für alle Elemente die gleichen Eigenschaften abgefragt werden können und je nach Elementtyp mal die eine, mal die andere Eigenschaft Werte zurückliefert, liest das Document Object Model die einzelnen Elemente des XML-Dokuments mit einem entsprechenden Objekttyp ein.

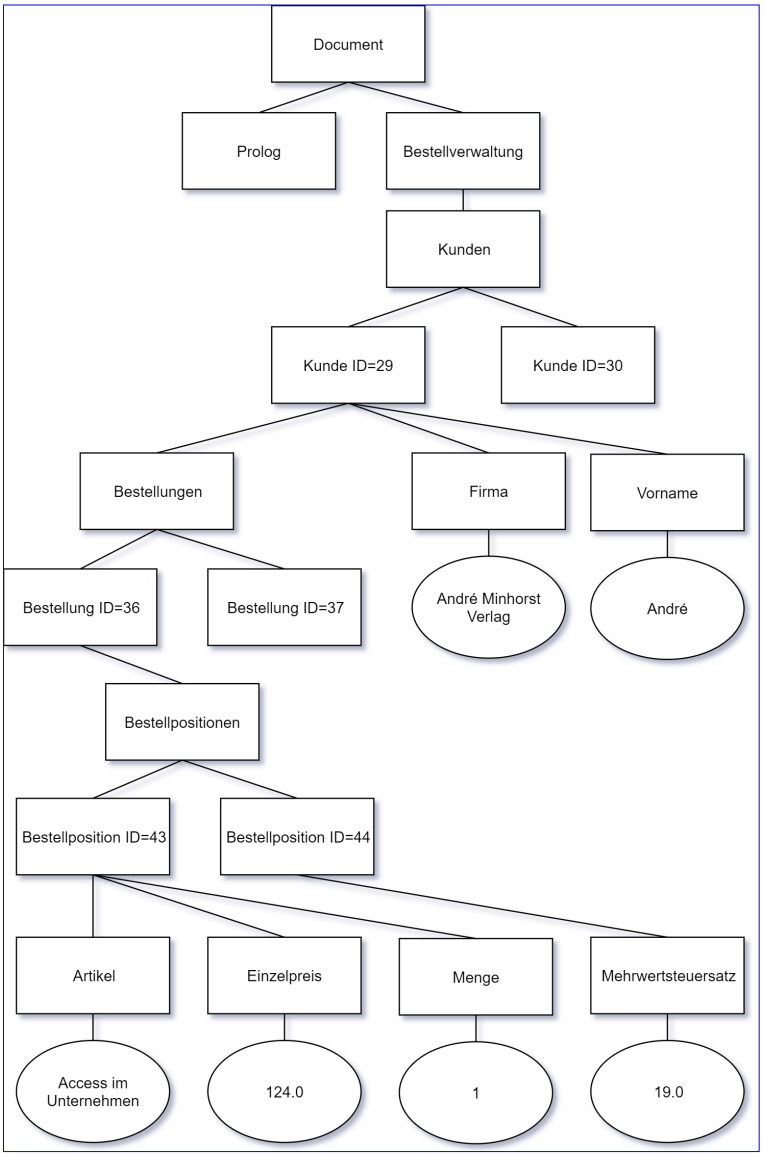


Bild 1: Baum eines XML-Dokuments

XML-Baum

Intern bildet DOM das XML-Dokument als Baumstruktur ab. Das kann man sich so vorstellen, dass wir ein Stammelement haben, das an der Spitze sitzt. Darunter gibt es dann ein oder mehrere untergeordnete Elemente, die wiederum untergeordnete Elemente haben, bis dann irgendwann Werte in den Elementen stehen – diese landen immer am anderen Ende des Baums. Einen Baum für unser Beispiel-XML-Dokument finden Sie in Bild 1.

Dieser bildet die aus Platzgründen gekürzte Version des XML-Dokuments aus Bild 2 ab.

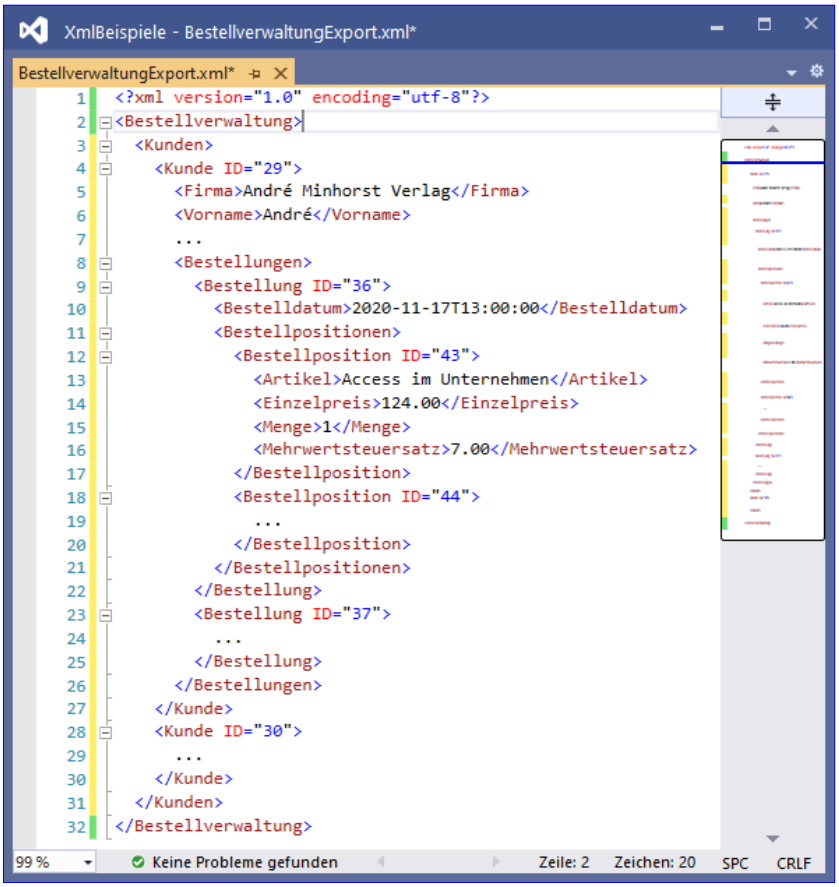


Bild 2: Ausschnitt des Dokuments entsprechend dem Baum aus der ersten Abbildung

Verweis auf den Namespace System.Xml

Bevor wir ein Objekt auf Basis der Klasse **XmlDocument** nutzen können, müssen wir einen Verweis auf den folgenden Namespace zur Klasse hinzufügen:

```
Imports System.Xml
```

Deklarieren und Initialisieren von XmlDocument

Mit den folgenden beiden Anweisungen deklarieren und initialisieren wir ein Objekt namens **objXML** mit dem Typ **XmlDocument**:

```
Dim objXML As XmlDocument
objXML = New XmlDocument
```

Laden des XML-Dokuments

Für das Laden eines XML-Dokuments bietet die **XmlDocument**-Klasse zwei Methoden an:

- **Load**: Erwartet den Pfad des zu ladenden XML-Dokuments als Parameter, falls es sich um eine Datei handelt. Sie können auch eine URL angeben. Überladungen erlauben auch das Laden aus einem **Stream**-, einem **TextReader**- oder einem **XmlReader**-Objekt.

- **LoadXml**: Erwartet das XML-Dokument als String-Parameter.

Die folgenden Anweisungen erzeugen ein **XmlDocument**-Objekt, laden das Dokument **Bestellverwaltung.xml** hinein und geben den Inhalt über die Eigenschaft **OuterXml** aus:

```
Dim objXML As XmlDocument
objXML = New XmlDocument
objXML.Load("C:\...\Bestellverwaltung.xml")
Debug.Print(objXML.OuterXml)
```

Die Ausgabe mit **InnerXml** liefert für das **XmlDocument** selbst übrigens das gleiche Ergebnis wie **OuterXml**.

Fehler beim Einlesen abfangen

Im Gegensatz zur **Load**-Methode von DOM unter VBA, die einen **Boolean**-Wert zurückliefert, falls beim Einlesen ein Fehler auftritt, hat die **Load**-Methode unter VB.NET keinen Rückgabewert. Wenn ein nicht gültiges XML-Dokument geladen wird, köst die Methode schlicht und einfach einen Fehler aus. Diesen können wir jedoch per **Try...Catch** wie folgt abfangen:

```
Try
    objXML.Load("C:\...\Bestellverwaltung.xml")
Catch ex As XmlException
    MessageBox.Show("Fehler beim Einlesen:" + vbNewLine + vbNewLine + ex.Message)
End Try
```

Sie können einen solchen Fehler leicht auslösen, indem Sie einen der Start- oder Endtags aus der Beispieldatei **Bestellverwaltung.xml** entfernen und dann versuchen, diese einzulesen.

Klassen für die Untersuchung von XML-Dokumenten mit DOM

Die Basisklasse für die Arbeit mit XML-Dokumenten heißt **XmlNode**. Es gibt einige abgeleitete Klassen, zu denen auch **XmlDocument** gehört. Das heißt, das **XmlNode** die grundlegenden Methoden und Eigenschaften bereitstellt und die abgeleiteten Klassen je nach Klasse weitere Elemente. **XmlDocument** bietet beispielsweise die bereits beschriebenen Methoden **Load** und **LoadXml** an. Weitere von **XmlNode** abgeleitete Klassen sind **XmlLinkedNode** und **XmlAttribute**. Von **XmlLinkedNode** sind weiterhin beispielsweise **XmlProcessingInstruction**, **XmlDeclaration** oder **XmlElement** abgeleitet – sowie die Klasse **XmlCharacterData**, die wiederum die Basis für Klassen wie **XmlComment** oder **XmlCDATASection** bildet.

Das heißt, dass Sie alle Elemente eines XML-Dokuments mit der Klasse **XmlNode** referenzieren können, aber wenn Sie die spezifischen Methoden und Eigenschaften nutzen wollen, finden Sie den genauen Typ heraus und nutzen diesen.

Nodes durchlaufen mit der ChildNodes-Auflistung

Mit **objXML** haben wir nun einen Verweis auf das **XmlDocument**-Objekt selbst. Wie kommen wir nun an die untergeordneten Elemente heran? Dafür bietet die **XmlNode**-Klasse und somit auch die **XmlDocument**-Klasse die Auflistung **ChildNodes** an. Wir können zunächst herausfinden, ob **XmlDocument** überhaupt untergeordnete Elemente hat. Das erledigen wir mit der Eigenschaft **HasChildNodes**. Sind untergeordnete Elemente vorhanden, können wir über die **ChildNodes**-Auflistung auf diese zugreifen. Diese bietet mit der Eigenschaft **Count** die Möglichkeit, die Anzahl zu ermitteln. Im folgenden Beispiel geben wir die

Werte von **HasChildNodes** und **ChildNodes.Count** aus und durchlaufen dann eine Schleife über alle **ChildNode**-Elemente. Diese referenzieren wir mit einer Variablen des Typs **XmlNode**. Das ist wichtig, weil sich in der Auflistung **ChildNodes** verschiedene Elemente mit von **XmlNode** abgeleiteten Klassen befinden. Wenn wir eine Variable des Typs **XmlElement** nutzen, erhalten wir einen Fehler, wenn eines der **ChildNode**-Elemente etwa den Typ **XmlDeclaration** hat.

Wir können aber innerhalb der Schleife genau herausfinden, um welchen Typ es sich handelt. Dazu nutzen wir die Eigenschaft **NodeType**. Außerdem geben wir mit folgenden Zeilen den Namen der untergeordneten Elemente aus:

```
Dim objNode As XmlNode
Debug.Print (objXML.HasChildNodes)
Debug.Print (objXML.ChildNodes.Count)
For Each objNode In objXML.ChildNodes
    Debug.Print (objNode.Name + " " + objNode.NodeType.ToString)
Next objNode
```

Das Ergebnis für dieses Beispiel lautet:

```
xml XmlDeclaration
Bestellverwaltung Element
```

Damit haben wir also die beiden Elemente der obersten Ebene erfasst, wobei das erste den Typ **XmlDeclaration** und das zweite den Typ **Element** hat:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
```

Nun wollen wir auch die übrigen Elemente erkunden. Dazu erstellen wir eine rekursiv definierte Methode, mit der wir uns durch die Hierarchie des XML-Dokuments arbeiten.

Nodes rekursiv durchlaufen

Die rekursive Methode rufen wir in der Schleife der Hauptmethode wie folgt auf:

```
For Each objNode In objXML.ChildNodes
    Debug.Print(objNode.Name + " " + objNode.NodeType.ToString)
    Nodes_Rekursiv(objNode, "  ")
Next objNode
```

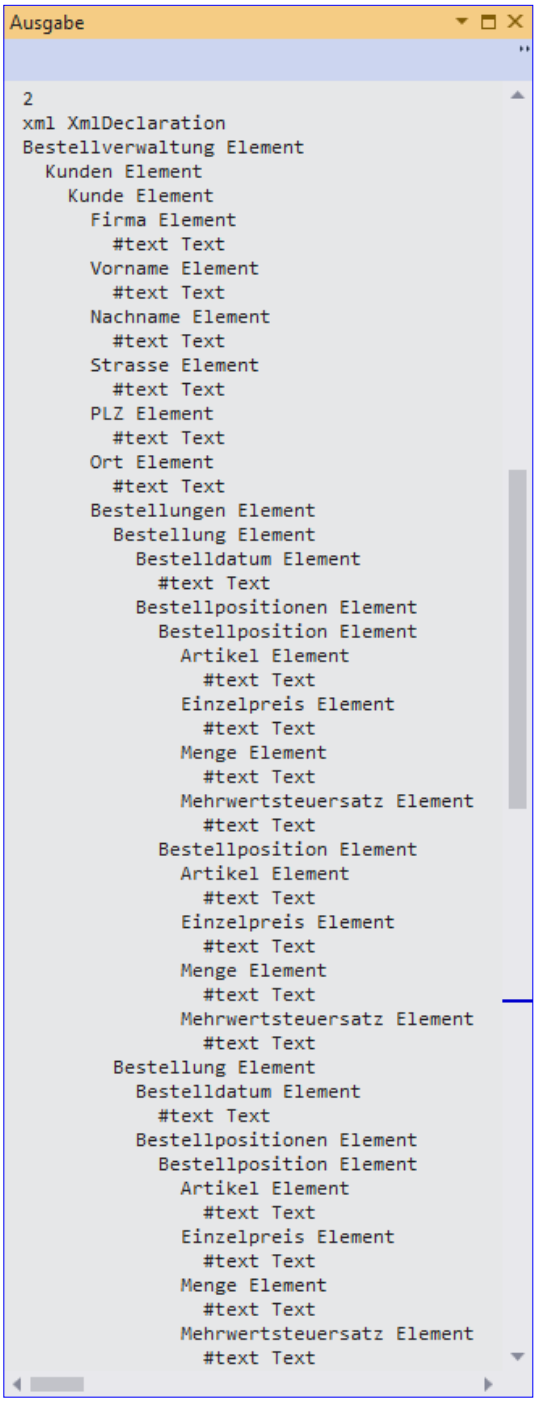


Bild 3: Name und Typen von Elementen

Onlinebanking mit DDBAC: Benutzeroberfläche

Nachdem wir die Funktionen zum Abrufen von Kontoständen und Umsätzen sowie für das Überweisen programmiert haben, können wir uns an den Entwurf einer Benutzeroberfläche für diese Funktionen begeben. Diese soll kompakte Formulare für die wichtigsten bereits behandelten Onlinebanking-Funktionen enthalten. Dazu gehört die Auswahl von Bankaccount und Konto sowie die Anzeige von Umsätzen und Kontostand. Dies erledigen wir im Hauptfenster der Anwendung. Ein weiteres Fenster erlaubt dann das Ausführen einer Überweisung für das aktuell im Hauptfenster ausgewählte Konto. Außerdem benötigen wir noch eine Tabelle, mit der wir die abgefragten Umsatzdaten speichern können. Diese fügen wir per Entity Framework hinzu.

Voraussetzungen

Voraussetzung für das Umsetzen der Lösung dieses Artikels ist das Vorhandensein einer DDBAC-Lizenz. Die damit nutzbaren Komponenten sind leider nicht mehr kostenlos verfügbar, sondern müssen lizensiert werden, bevor man diese in seine Produkte einbauen kann. Lizenzen finden Sie im Onlineshop unter **https://shop.minhorst.com/access-tools/295/ddbac-jahreslizenz?c=78**.

Vorbereitungen

Als Projekt legen wir ein neues Projekt des Typs **Visual Basic|Windows Desktop|WPF-App** an.

Dann fügen wir ein neues Element des Typs **ADO.NET Entity Data Model** hinzu. Dieses dient dazu, ein Entity Data Model zu erstellen und über dieses automatisch eine Datenbank mit der benötigten Tabelle zu generieren. Nachdem Sie mit dem Menübefehl **Projekt|Neues Element hinzufügen...** den Dialog **Neues Element hinzufügen** geöffnet haben, tragen Sie den Namen des Entity Data Models ein (hier **OnlinebankingContext**) und klicken auf **Hinzufügen**. Im nun erscheinenden **Assistent für Entity Data Model** wählen Sie den Typ **Leeres Code First-Modell** aus. Wenige Sekunden später hat der Assistent die notwendigen Elemente hinzugefügt.

Durch diesen Schritt haben wir außerdem direkt Änderungen an der Datei **App.config** vorgenommen. Dort steht nun im Element **connectionStrings** die Verbindungszeichenfolge, mit der wir später die Datenbank zum Speichern der Umsätze erstellen und verwenden.

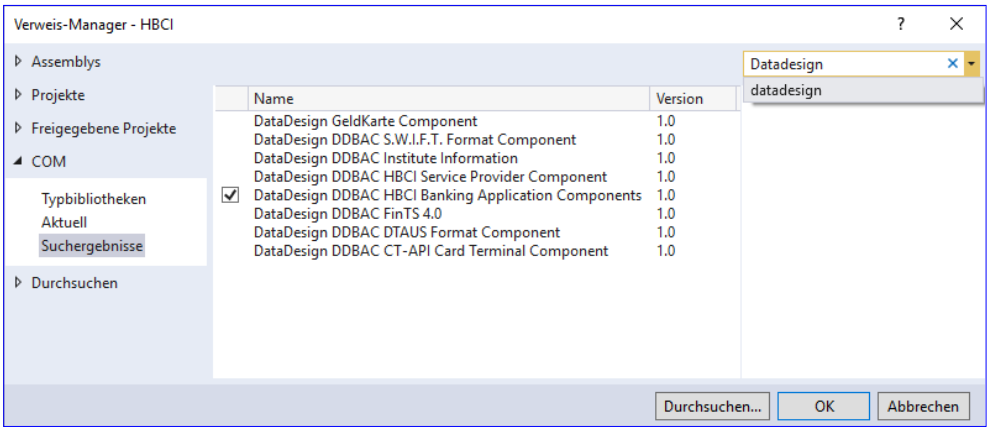


Bild 1: Verweis auf die DDBAC-Bibliothek

Außerdem benötigen wir noch eine Referenz auf die

DDBAC-Komponenten, die den Zugriff auf Ihre Konten überhaupt erst ermöglichen. Dazu fügen Sie dem Projekt den Verweis aus Bild 1 hinzu.

Kontostand anzeigen

Wir beginnen mit der Anzeige des Kontostands. Hier wollen wir in einem Fenster die Auswahl des Kontakts und anschließend des gewünschten Kontos erlauben. Per Klick auf eine Schaltfläche soll dann der Kontostand eingelesen und in einem Textfeld angezeigt werden. Wie Sie bereits in den vorherigen Artikeln erfahren haben, legen wir die Kontakte und Konten über die Systemsteuerung an, und zwar über die App **Homebanking Administrator (32-Bit)**. Auf die dort angelegten und synchronisierten Elemente können wir dann per Code zugreifen.

Ein neues WPF-Fenster soll nun zunächst ein Kombinationsfeld zur Auswahl des Kontakts anbieten. Nachdem der Benutzer dort einen Eintrag ausgewählt hat, sollen die zum gewählten Kontakt gehörenden Konten in einem zweiten Kombinationsfeld angezeigt werden. Das passende Steuerelement für diesen Zweck ist jeweils das **ComboBox**-Element. Also fügen wir zunächst zwei solche Elemente im Hauptfenster **MainWindow.xaml** an. Damit die zusätzlich angelegten Bezeichnungsfelder optisch ansprechend dargestellt werden, erstellen wir zuvor einige Spalten und Zeilen im bereits vorhandenen **Grid**-Element.

Klasse für die Kontakte anlegen

Die über die Systemsteuerung hinzufügten Onlinebanking-Kontakte lesen wir mithilfe der im Artikel **Onlinebanking mit DDBAC: Saldo und Umsätze** beschriebenen Funktion ein, die wir jedoch leicht abwandeln. Die eingelesenen Daten sollen je Kontakt in einer neuen Instanz der nachfolgend beschriebenen Klasse landen:

```
Public Class Contact
    Public BankCode As String
    Public CountryCode As String
    Public UserID As String
End Class
```

Diese Klasse legen Sie als neues Element im Projekt an und speichern es unter **Contact.vb**. Wir benötigen allerdings die **INotifyPropertyChanged**-Schnittstelle in dieser Klasse, daher ändern wir diese wie folgt.

Wir fügen außerdem noch eine Eigenschaft namens **BACContact** hinzu, mit der wir einen Verweis auf das **BACContact**-Objekt speichern, aus dem wir die Eigenschaften der Klasse ausgelesen haben:

```
Imports System.ComponentModel

Public Class Contact
    Implements INotifyPropertyChanged
    Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged
    Protected Overridable Sub OnPropertyChanged(propname As String)
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propname))
    End Sub
```

```
Private _bankCode As String
Private _countryCode As String
Private _userID As String
Private _bACContact As BACContact

Public Property BankCode As String
    Get
        Return _bankCode
    End Get
    Set(value As String)
        _bankCode = value
        OnPropertyChanged("BankCode")
    End Set
End Property

Public Property CountryCode As String
    Get
        Return _countryCode
    End Get
    Set(value As String)
        _countryCode = value
        OnPropertyChanged(CountryCode)
    End Set
End Property

Public Property UserID As String
    Get
        Return _userID
    End Get
    Set(value As String)
        _userID = value
        OnPropertyChanged("UserID")
    End Set
End Property

Public Property BACContact As BACContact
    Get
        Return _bACContact
    End Get
    Set(value As BACContact)
        _bACContact = value
```

```
        End Set
    End Property
End Class
```

Klasse für die Konten anlegen

Die Klasse für die Konten sieht ähnlich aus wie die für die Kontakte. Auch hier implementieren wir die **INotifyPropertyChanges**-Schnittstelle. Die Klasse bietet die Eigenschaften **AccountNumber**, **AcctName**, **BIC** und **IBAN** an. Darüberhinaus wollen wir das **BACAccount**-Objekt, über das wir die Klasse füllen, mit den Instanzen der Klasse speichern. Die Klassendefinition sieht wie folgt aus:

```
Imports System.ComponentModel
Imports BankingApplicationComponents

Public Class Account
    Implements INotifyPropertyChanged

    Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged

    Protected Overridable Sub OnPropertyChanged(propname As String)
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propname))
    End Sub

    Private _accountNumber As String
    Private _acctName As String
    Private _bIC As String
    Private _iBAN As String
    Private _bACAccount As BACAccount

    Public Property AccountNumber As String
        Get
            Return _accountNumber
        End Get
        Set(value As String)
            _accountNumber = value
            OnPropertyChanged("AccountNumber")
        End Set
    End Property
    ... weitere Property-Eigenschaften

    Public Property BACAccount As BACAccount
        Get
```

```
            Return _bACAccount
        End Get
        Set(value As BACAccount)
            _bACAccount = value
        End Set
    End Property
End Class
```

Auflistung mit Kontakt-Elementen füllen

Wenn die Anwendung gestartet und somit das Fenster **MainWindow.xaml** geöffnet wird, sollen die Kontakte aus der Systemsteuerung eingelesen und in jeweils eine neue Klasse des Typs **Contact** geschrieben werden. Die neuen Instanzen wollen wir außerdem direkt zu einer Liste hinzufügen, die wir anschließend dem ersten **ComboBox**-Element zuweisen können. Wir nutzen dazu eine Funktion namens **GetBACContacts**, die eine Liste von Elementen des Typs **BACContact** liefert. Diese fügen wir einem Modul namens **DDBACObjects.vb** hinzu, dem wir außerdem noch per **Imports**-Anweisung die Elemente des Namespace **BankingApplicationComponents** hinzufügen.

Die folgende privat deklarierte Variable speichert die **Contacts**-Auflistung zwischen, damit diese nicht bei jedem Zugriff erneut ausgelesen werden müssen:

```
    Private m_Contacts As BACContacts
```

Die Funktion **GetBACContacts** liest die Kontakte mit den Methoden der DDBAC-Bibliothek aus der Systemsteuerung in die Variable **m_BACContacts** ein. Diese wird dann anschließend auch als Ergebnis der Funktion zurückgegeben. Wenn **m_BACContacts** bereits vorhanden ist, wird die Auflistung nicht erneut gefüllt. Einzige Ausnahme ist, wenn der Aufruf den Wert **True** für den Parameter **bolReset** enthält. In diesem Fall werden die Kontakte auf jeden Fall erneut eingelesen:

```
Public Function GetBACContacts(Optional bolReset As Boolean = False) As BACContacts
    If m_BACContacts Is Nothing Then
        m_BACContacts = New BACContacts
        m_BACContacts.Populate("")
    Else
        If bolReset = True Then
            m_BACContacts.Populate("")
        End If
    End If
    Return m_BACContacts
End Function
```

Damit haben wir mit **GetBACContacts** eine Liste der **BACContact**-Elemente eingelesen. Diese müssen wir nun noch in eine **ObservableCollection** mit Elementen des oben definierten Typs **Contact** umfüllen. Diese Liste wollen wir dann als Datenquelle des ersten **ComboBox**-Elements in unserem Fenster verwenden. Diese Aufgabe übernimmt die Funktion **GetContacts**. Sie

erstellt eine neue **ObservableCollection** in der Variablen **objContacts**. Dann durchläuft sie eine **For Each**-Schleife über alle **BACContact**-Elemente, welche die oben beschriebene Funktion **GetContacts** liefert. Innerhalb der Schleife legt sie ein neues **Contact**-Element an (nicht zu verwechseln mit **BACContact**), und schreibt die entsprechenden Werte in die Eigenschaften **BankCode**, **CountryCode** und **UserID** sowie einen Verweis auf das aktuelle **BACContact**-Objekt in die Eigenschaft **BACContact**. Anschließend fügt sie das neu erstellte und gefüllte Objekt zur ObservableCollection hinzu:

```
Public Function GetContacts() As ObservableCollection(Of Contact)
    Dim objContacts As ObservableCollection(Of Contact)
    Dim objContact As Contact
    Dim objBACContact As BACContact
    objContacts = New ObservableCollection(Of Contact)
    For Each objBACContact In GetBACContacts()
        objContact = New Contact
        With objContact
            .BankCode = objBACContact.BankCode
            .CountryCode = objBACContact.CountryCode
            .UserID = objBACContact.UserID
            .BACContact = objBACContact
            objContacts.Add(objContact)
        End With
    Next
    Return objContacts
End Function
```

Auflistung für die Konten füllen

Wenn der Benutzer in der gleich noch zu erstellenden ComboBox einen Kontakt ausgewählt hat, soll die zweite ComboBox die zu diesem Kontakt gehörenden Konten zur Auswahl anbieten. Deshalb benötigen wir analog zu den oben beschriebenen Funktionen **GetBACContacts** und **GetContacts** eine weitere Funktion.

Die Funktion **GetBACAccounts** erwartet als Parameter ein **BACContact**-Objekt, zu dem sie die **BACAccount**-Objekte einlesen soll. Sie verwendet dann schlicht die **Accounts**-Auflistung des **BACContact**-Objekts, um die Liste in der Variablen **m_Accounts** des Typs **BACAccounts** zu speichern:

```
Public Function GetBACAccounts(objBACContact As BACContact) As BACAccounts
    Return objBACContact.Accounts
End Function
```

Wenn der Benutzer im ersten **ComboBox**-Feld mit den Kontakten einen Eintrag ausgewählt hat, holen wir mit der folgenden Funktion die Daten zum Füllen des zweiten **ComboBox**-Feldes zur Anzeige der Konten dieses Kontakts. Dabei verwenden wir auch hier als Parameter das **BACContact**-Element des zu untersuchenden Kontaktes. Die Funktion erstellt als Erstes wieder eine neue ObservableCollection.

In einer **For Each**-Schleife durchlaufen wir diesmal die mit der Funktion **GetBACAccounts** geholten Konten. Innerhalb der **For Each**-Schleife legen wir ein neues Objekt des Typs **Account** an und weisen diesem die Werte für die Eigenschaften **AccountNumber**, **AcctName**, **IBAN** und **BIC** zu sowie einen Verweis auf das **BACAccount**-Objekt dieses Kontos. Anschließend fügen wir es der ObservableCollection namens **objAccounts** zu, die wir als Funktionsergebnis zurückliefern:

```
Public Function GetAccounts(objBACContact As BACContact) As ObservableCollection(Of Account)
    Dim objAccounts As ObservableCollection(Of Account)
    Dim objAccount As Account
    Dim objBACAccount As BACAccount
    objAccounts = New ObservableCollection(Of Account)
    For Each objBACAccount In GetBACAccounts(objBACContact)
        objAccount = New Account
        With objAccount
            .AccountNumber = objBACAccount.AccountNumber
            .AcctName = objBACAccount.AcctName
            .BIC = objBACAccount.BIC
            .IBAN = objBACAccount.IBAN
            .BACAccount = objBACAccount
        End With
        objAccounts.Add(objAccount)
    Next
    Return objAccounts
End Function
```

Fenster zur Anzeige der Auswahlfelder für Kontakte und Konten

Das Fenster soll im oberen Bereich zwei **ComboBox**-Elemente zur Auswahl von Kontakten und Konten anzeigen. Im Kontakt-Kombinationsfeld wollen wir die **UserID** und die **Bankleitzahl** anzeigen. Das Konto-Auswahlfeld soll nur die IBAN anzeigen. Die Definition dieser beiden Steuerelemente sieht wie folgt aus. Als Erstes das **ComboBox**-Element **cboContacts**, das an die Auflistung **Contacts** gebunden ist, die wir gleich noch im Code behind-Modul definieren.

Das **ComboBox**-Element hat außerdem ein Attribut für das Ereignis **SelectionChanged**. Da wir zwei Felder im **ComboBox**-Element anzeigen wollen, stellen wir diese in einem **ItemTemplate** und den entsprechenden untergeordneten Elementen wie folgt dar:

```
<Label Grid.Column="0" Content="Kontakt:"/>
<ComboBox x:Name="cboContacts" Grid.Row="0" Grid.Column="1" Width="200"
            ItemsSource="{Binding Contacts}"
            SelectedValuePath="UserID"
            SelectionChanged="CboContacts_SelectionChanged">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock>
```

```
                <TextBlock.Text>
                    <MultiBinding StringFormat="{}{0} {1}">
                        <Binding Path="UserID" />
                        <Binding Path="BankCode" />
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

Das zweite **ComboBox**-Element für die Konten ist an die Liste **Accounts** gebunden. Sie soll nur das Feld **IBAN** anzeigen und löst ebenfalls das Ereignis **SelectionChanged** aus:

```
<Label Grid.Column="0" Grid.Row="1" Content="Konto:"/>
<ComboBox x:Name="cboAccounts" Grid.Row="1" Grid.Column="1" Width="200"
            ItemsSource="{Binding Accounts}"
            SelectedValuePath="IBAN"
            SelectionChanged="CboAccounts_SelectionChanged"
            DisplayMemberPath="IBAN">
</ComboBox>
```

Code behind-Klasse des Fensters

Die Code behind-Klasse verwendet die folgenden beiden Namespaces:

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

Außerdem implementiert sie die **INotifyPropertyChanged**-Schnittstelle:

```
Class MainWindow
    Implements INotifyPropertyChanged

    Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged

    Protected Overridable Sub OnPropertyChanged(propname As String)
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propname))
    End Sub
```

Wir verwenden zwei Eigenschaften für die beiden Auflistungen der Kontakte und der Konten. Die privaten Variablen dafür sehen wie folgt aus:

```
Private _contacts As ObservableCollection(Of Contact)
Private _accounts As ObservableCollection(Of Account)
```

Passend dazu verwenden wir die folgenden **Property**-Eigenschaften:

```
Public Property Contacts As ObservableCollection(Of Contact)
    Get
        Return _contacts
    End Get
    Set(value As ObservableCollection(Of Contact))
        _contacts = value
        OnPropertyChanged("Contacts")
    End Set
End Property

Public Property Accounts As ObservableCollection(Of Account)
    Get
        Return _accounts
    End Get
    Set(value As ObservableCollection(Of Account))
        _accounts = value
        OnPropertyChanged("Accounts")
    End Set
End Property
```

In der Konstruktor-Methode, die beim Initialisieren ausgelöst wird, initialisieren wir zunächst das per XAML definierte Fenster mit **InitializeComponent**. Dann weisen wir der Eigenschaft **Contacts** das Ergebnis der oben beschriebenen Funktion **GetContacts** zu. Mit dem Wert **True** für die Eigenschaft **IsDropDownOpen** stellen wir ein, dass das Kombinationsfeld direkt aufgeklappt wird. Außerdem stellen wir die Eigenschaft **DataContext** auf **Me** ein, was dafür sorgt, dass die im Code behind-Modul enthaltenen öffentlichen Eigenschaften als Datenquelle für die Kombinationsfelder verwendet werden können:

```
Public Sub New()
    InitializeComponent()
    Contacts = GetContacts
    cboContacts.IsDropDownOpen = True
    DataContext = Me
End Sub
```

Sollte der Benutzer einen der Einträge des ersten Kombinationsfeldes auswählen, löst er damit auch das **SelectionChanged**-Ereignis aus, für das wir die folgende Methode hinterlegt haben. Diese Methode ermittelt das im auslösenden **ComboBox**-Element enthaltene Element und speichert es in der Variablen **objContact**.

Dann liest es mit der Funktion **GetAccounts** die Konten für diesen Kontakt ein und schreibt sie in die Eigenschaft **Accounts**:

```
Private Sub CboContacts_SelectionChanged(sender As Object, e As SelectionChangedEventArgs)
    Dim cbo As ComboBox
    Dim objContact As Contact
    cbo = sender
    objContact = cbo.SelectedItem
    Accounts = GetAccounts(objContact.BACContact)
    cboAccounts.IsDropDownOpen = True
End Sub
```

Die letzte Eigenschaft sorgt dafür, dass das **ComboBox**-Element **cboAccounts** nach der Auswahl eines Kontakts aufgeklappt wird. Da die Eigenschaft **Accounts** mit der **OnPropertyChanged**-Methode ausgestattet ist, wurde das daran gebundene **ComboBox**-Element **cboAccounts** automatisch aktualisiert.

Kontostand ermitteln

Wenn der Benutzer einen Kontakt und ein Konto ausgewählt hat, können wir den Kontostand ermitteln. Die Frage ist noch, ob das immer automatisch geschehen soll oder erst nach einer Aktion wie dem Anklicken einer Schaltfläche. Wir erledigen das hier manuell mit einer entsprechenden Schaltfläche.

Kontostand im Textfeld anzeigen

Das **Label**-, das **TextBox** und das **Button**-Element für das Einlesen und Anzeigen des Kontostands fügen wir wie folgt zum Fenster hinzu:

```
<Label Grid.Row="2">Kontostand:</Label>
<TextBox x:Name="txtKontostand" Grid.Row="2" Grid.Column="1" Width="100"></TextBox>
<Button x:Name="btnKontostandAbrufen" Grid.Row="2" Grid.Column="2"
    Click="BtnKontostandAbrufen_Click">Kontostand abrufen</Button>
```

Die Schaltfläche **btnKontostandAbrufen** löst die folgende Methode aus. Diese übergibt den Index des Kombintionsfeldes **cboContacts** und des Kombinationsfeldes **cboAccounts** an die Funktion **KontostandAbfragen**, die den Kontostand zurückliefert. Dieser landet schließlich im Textfeld **txtKontostand**:

```
Private Sub BtnKontostandAbrufen_Click(sender As Object, e As RoutedEventArgs)
    Dim decKontostand As Decimal
    decKontostand = KontostandAbfragen(cboContacts.SelectedIndex, cboAccounts.SelectedIndex)
    txtKontostand.Text = decKontostand
End Sub
```

Die hier aufgerufene Funktion **KontostandAbfragen** ist eine neue Version der gleichnamigen Funktion aus dem Artikel **Onlinebanking mit DDBAC: Saldo und Umsätze** (**www.datenbankentwickler.net/192**). Geändert wurden der Teil, der den Dialog