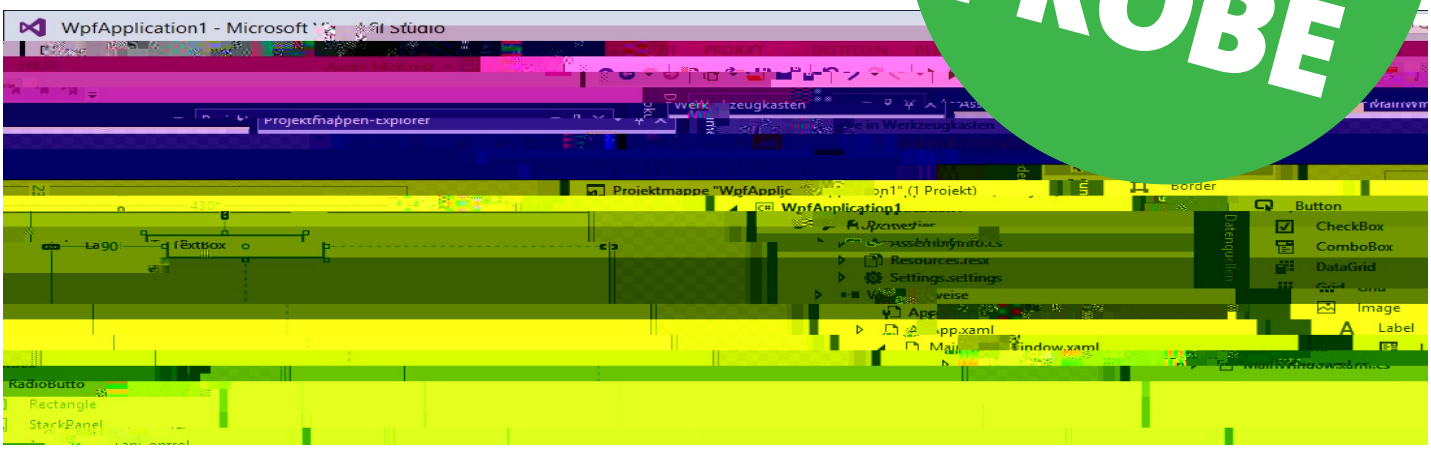
AUSGABE 03/2016

DATENBANK ENTWICKLER

Gratis LESE-PROBE

MAGAZIN FÜR DIE DATENBANKENTWIC
VISUAL STUDIO FÜR DESKTOP, WEB U



TOP-THEMEN:

C#-GRUNDLAGEN	Von VBA zu C#: Fehlerbehandlung	SEITE 3
PROGRAMMIEREN	Objektorientierung: Interfaces	SEITE 19
INTERAKTIV	C#-DLL für COM/VBA erstellen	SEITE 24
WPF-BASICS	Datenbindung an einfache Objekte	SEITE 36
DATENZUGRIFF	Entity-Framework-Grundlagen	SEITE 51

André Minhorst Verlag

C#-GRUNDLAGEN	Von VBA zu C#: Fehlerbehandlung	3
C#-PROGRAMMIERTECHNIK	Objektorientierte Programmierung: Interfaces	19
INTERAKTIV	C#-DLL für COM/VBA erstellen	24
WPF-GRUNDLAGEN	WPF-Datenbindung: Einfache Objekte	36
	WPF-Datenbindung: Listen-Objekte	42
DATENZUGRIFFSTECHNIK	Einführung in das Entity Framework	51
TIPPS UND TRICKS	Tipps und Tricks zu Fenstern und Steuerelementen	65
SERVICE	Impressum	2
DOWNLOAD	Die Downloads zu dieser Ausgabe finden Sie unter folgendem Link: **http://www.amvshop.de** Klicken Sie dort auf **Mein Konto**, loggen Sie sich ein und wählen dann **Meine Sofortdownloads**.	

Impressum

DATENBANKENTWICKLER

Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

Von VBA zu C#: Fehlerbehandlung

Nur wenige Access-Programmierer statten ihre Anwendungen mit einer ordentlichen Fehlerbehandlung aus. Für viele ist dies ein leidiges Thema. Die Fehlerbehandlung sorgt im Optimalfall sowohl unter VBA als auch unter C# dafür, dass eine Anwendung stabiler wird und nach dem Auftreten von Laufzeitfehlern nicht unerwartet reagiert oder sogar abstürzt. Während es unter VBA nur wenige Konstrukte gibt, um eine Fehlerbehandlung zu implementieren, bietet C# schon eine Menge mehr. Dieser Artikel liefert eine Einführung und zeigt, wie Sie die von VBA gewohnten Techniken unter C# einsetzen.

Die minimale Fehlerbehandlung unter VBA sieht so aus, dass Sie vor fehlerträchtigen Anweisungen die eingebaute Fehlerbehandlung mit der folgenden Programmzeile deaktivieren:

```
On Error Resume Next
```

Alle folgenden Fehler in der aktuellen Routine und in solchen, die von dieser Routine aufgerufen werden, werden nicht behandelt. Dadurch gibt es zwar immerhin keine Fehlermeldung und es werden keine Variablen geleert, was beim Auftreten unbehandelter Fehler auftreten kann. Allerdings erledigen fehlerhafte Zeilen ihre Aufgabe nicht, was zu Folgefehlern (auch logischen Fehlern) in den folgenden Anweisungen führen kann.

Spätestens mit dem Ende der Routine endet die Deaktivierung der eingebauten Fehlerbehandlung. Vorher erhalten Sie dies mit der folgenden Anweisung:

```
On Error Goto 0
```

Dadurch reagiert VBA wieder wie gewohnt mit entsprechenden Fehlermeldungen auf Fehler.

Dazwischen haben Sie Gelegenheit, benutzerdefiniert auf die auftretenden Fehler zu reagieren – beispielsweise, indem Sie den Wert der Eigenschaft **Number** des **Err**-Objekts auslesen und für die interessanten Werte entsprechende Fehlerbehandlungen hinzufügen.

Dies kann beispielsweise wie folgt aussehen:

```
On Error Resume Next
Debug.Print 1/0
Select Case Err.Number
    Case 11
        MsgBox "Fehler: Teilen durch 0."
End Select
On Error Goto 0
```

Unter C# gibt es zur Fehlerbehandlung andere Möglichkeiten, die Sie in den folgenden Abschnitten kennen lernen.

Warum überhaupt eine Fehlerbehandlung?

Um auf Nummer sicher zu gehen, wollen wir zuvor noch einmal kurz auf die Gründe für die Programmierung einer benutzerdefinierten Fehlerbehandlung eingehen. Wenn Sie

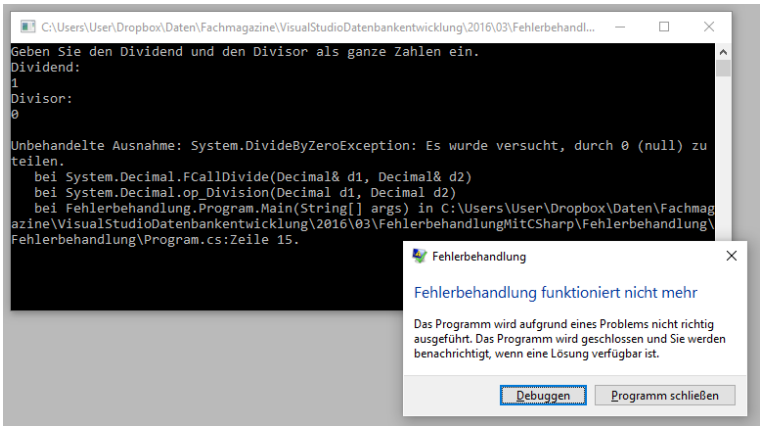


Bild 1: Unbehandelte Ausnahme bei einer Konsolenanwendung

Code programmieren, der fehlerhafte Eingaben zulässt und keine entsprechende benutzerdefinierte Fehlerbehandlung aufweist, erhalten Sie beispielsweise bei einer Konsolenanwendung eine Meldung plus Fehlerbehandlungsfenster wie in Bild 1.

Wenn Sie einen solchen Fehler in einer WPF-Anwendung auslösen, erhalten Sie noch nicht einmal einen kleinen Hinweis auf den Fehler im Code, der zu dieser Ausnahme führte (siehe Bild 2).

Noch übler wird es, wenn nicht Sie selbst die Software bedienen, sondern ein Kunde: Erstens macht dies nie einen guten Eindruck, zweitens kann der Kunde auf Basis der hier gelieferten Fehlerinformationen kaum helfen, den Fehler zu finden.

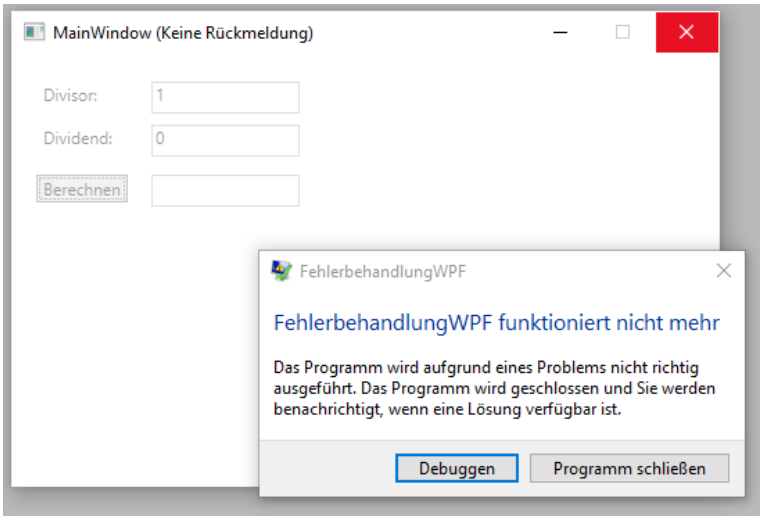


Bild 2: Unbehandelte Ausnahme bei einer Windows-Anwendung

Also sollten Sie dafür sorgen, dass der Kunde einen aussagekräftigen Text als Fehlermeldung erhält, der ihn beispielsweise bei einem Eingabefehler auf die Ursache des Fehlers hinweist oder aber ihm die Informationen liefert, die er zur Behebung des Problems an den Entwickler der Software weitergeben kann.

Der wichtigste Punkt beim Implementieren einer benutzerdefinierten Fehlerbehandlung etwa unter VBA ist jedoch die stabile Fortsetzung der Anwendung: Wenn ein Fehler aufgetreten ist, beispielsweise durch eine Fehleingabe oder einen fehlerhaften Zugriff auf ein Element, dann sollte die Anwendung nach der Ausgabe der Fehlermeldung fortgesetzt werden können und nicht einfach abbrechen. Dabei ist außerdem sicherzustellen, dass die Funktion und der stabile Zustand der Anwendung durch den Fehler nicht beeinträchtigt werden. Bei VBA war das mitunter ganz einfach deshalb nicht der Fall, weil durch unbehandelte Fehler Variableninhalte gelöscht oder Objekte zerstört wurden. Wurden die Zeilen, die den Fehler auslösten, hingegen per **On Error Resume Next** einer benutzerdefinierten Fehlerbehandlung zugeführt, statt einfach unbehandelte Fehler auszulösen, war zumindest schon einmal der Inhalt und der Zustand der Variablen sichergestellt.

Fehlerhafter Code

Schauen wir uns das Beispiel an, das zum Auslösen der Ausnahme der Konsolenanwendung führte. Den Code finden Sie in Listing 1. Die Zeile, die das Ergebnis der Division der Werte der Variablen **Dividend** und **Divisor** ermitteln soll, löst die Ausnahme aus, wenn der Divisor zuvor den Wert **0** zugewiesen bekommen hat – der Fehler tritt also durch eine Division durch **0** auf.

```
static void Main(string[] args) {
    Console.WriteLine("Geben Sie den Dividend und den Divisor als ganze Zahlen ein.");
    Console.WriteLine("Dividend:");
    decimal Dividend = Convert.ToDecimal(Console.ReadLine());
    Console.WriteLine("Divisor:");
    decimal Divisor = Convert.ToDecimal (Console.ReadLine());
    decimal Quotient = Dividend / Divisor;
    Console.WriteLine("Der Quotient lautet: {0}", Quotient);
    Console.ReadLine();
}
```

Listing 1: Methode, die einen Fehler auslöst, wenn als Divisor der Wert **0** angegeben wird

Wenn Sie die Anwendung debuggen, also diese von Visual Studio aus etwa mit

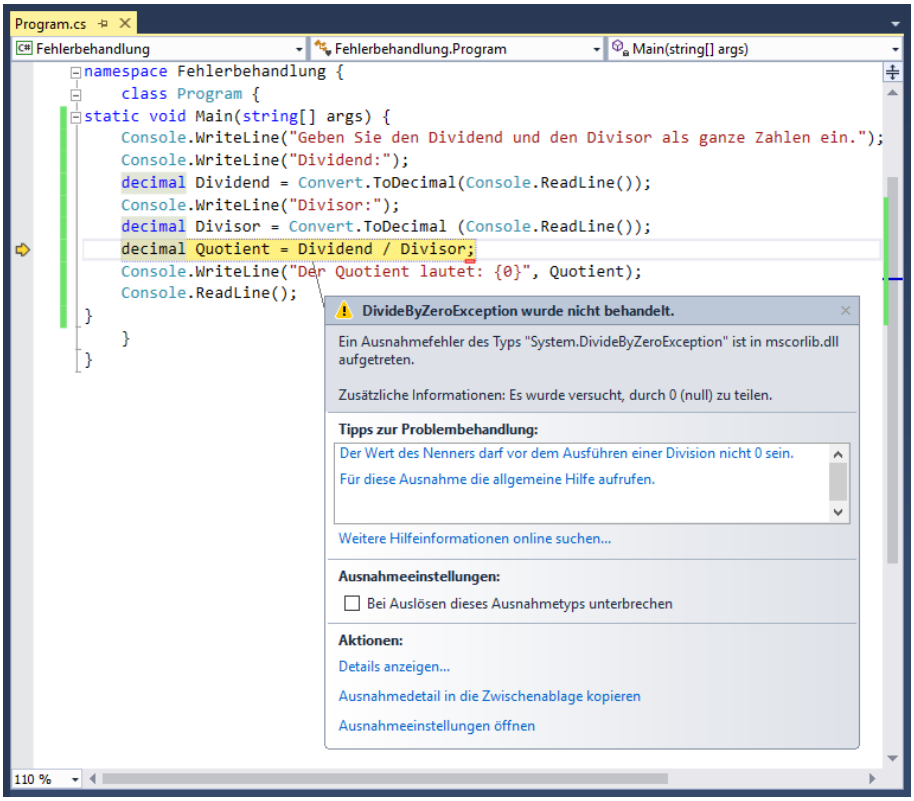


Bild 3: Auslösen einer Ausnahme unter C#

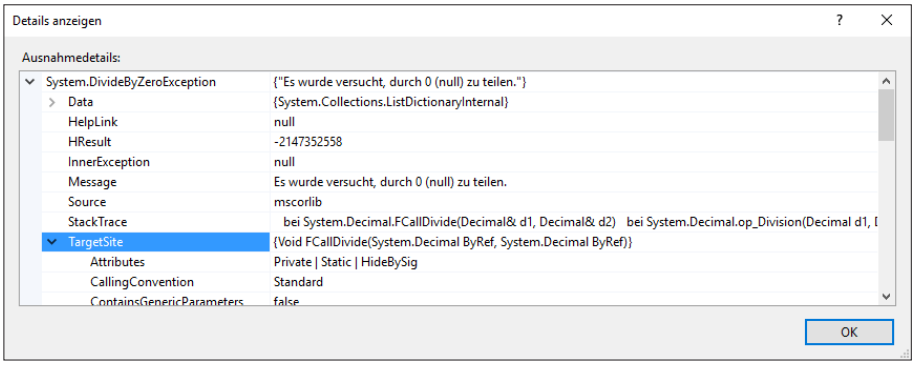


Bild 4: Details zu einer Ausnahme beim Debuggen in Visual Studio

der Taste **F5** starten, erhalten Sie einige Fehlerinformationen mehr. Außerdem markiert Visual Studio gleich die fehlerhafte Zeile (siehe Bild 3). Es gibt sogar noch Tipps zur Problembehandlung.

Wenn Sie möchten, erhalten Sie auch noch weitere Details, und zwar durch einen Mausklick auf den Link **Details anzeigen ...**, der den Dialog aus Bild 4 öffnet.

Für Fehler wie diese gibt es einige Beispiele – die Eingabe unzulässiger Werte wie in diesem Beispiel, Zugriff auf nicht vorhandene Dateien et cetera.

try...catch statt On Error Resume Next

Unter C# können Sie Laufzeitfehler in sogenannten **try...catch**-Blöcken behandeln. Das sieht dann so aus, dass Sie den Code, der einen Fehler auslösen könnte, in den **try**-Block packen und den Code, der ausgelöst werden soll, wenn innerhalb des **try**-Blocks ein Fehler auftritt, in den **catch**-Block.

In unserem Fall wollen wir die Zeile, welche die Division durchführt, in den **try**-Block überführen. Gleichzeitig fügen wir dort auch die Zeile ein, welche das Ergebnis aus der Variablen **Quotient** in der Konsole ausgibt – sonst ergibt dies einen Syntaxfehler, weil **Quotient** nicht in jedem Falle deklariert und initialisiert wird.

In den **catch**-Block schreiben wir eine Anweisung, welche einen Hinweis auf das Auftreten eines Fehlers in der Konsole ausgibt (siehe Listing 2). Wie Sie hier erkennen, wird die folgende Anweisung, die ja das Ergebnis der Berechnung ausgeben sollte, ignoriert – dies gilt grundsätzlich für alle Anweisungen, die der fehlerhaften Anweisung folgen.

Nun erhält der Benutzer zwar auch keine wesentlich aussagekräftigere Meldung, aber dafür wird das Programm auch nicht einfach abgebrochen. Außerdem haben wir ja auch

```
try {
    decimal Quotient = Dividend / Divisor;
    Console.WriteLine("Der Quotient lautet: {0}", Quotient);
}
catch {
    Console.WriteLine("Ups! Es ist ein Fehler beim Rechnen aufgetreten!");
}
```

Listing 2: Abfangen eines Fehlers per **try...catch**-Block

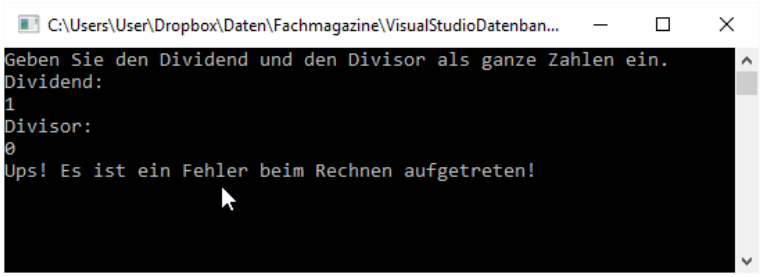


Bild 5: Benutzerdefinierte Fehlermeldung beim Auftreten der Ausnahme

noch gar nicht geprüft, um was für einen Fehler es sich handelt – dies prüfen wir in der folgenden Version.

Die Anwendung wird dann mit den Anweisungen fortgesetzt, die nach dem Ende des **catch**-Blocks folgen. Dies erkennen Sie in diesem Beispiel daran, dass das Betätigen der Eingabetaste das Programm beendet, weil die letzte **Console.ReadLine()**-Methode ausgeführt wird.

Exception statt Err

Unter C# heißt das Objekt, das die per Code auswertbaren Fehlerinformationen enthält, **Exception**. Es entspricht etwa dem Objekt **Err** unter VBA, das ja mit seinen Eigenschaften **Description** oder **Number** oft hilfreiche Informationen liefert.

Damit Sie dieses Objekt nutzen können, müssen Sie es für den **catch**-Zweig als Parameter hinzufügen, also wie in Bild 5 mit **catch (Exception e)**.

Die folgenden drei Zeilen des Beispiels geben verschiedene Informationen auf die Konsole aus, die Sie auch in Bild 6 sehen. Die erste ist der Inhalt der Eigenschaft **Message**. Die Eigenschaft **GetBaseException** liefert den Text, den auch ein unbehandelter Fehler auf die Konsole zaubert. Schließlich gibt uns die Eigenschaft **GetType** hilfreiche Informationen, wenn es darum geht, den Typ der Ausnahme zu ermitteln. In diesem Fall lautet der Typ **Divi-**

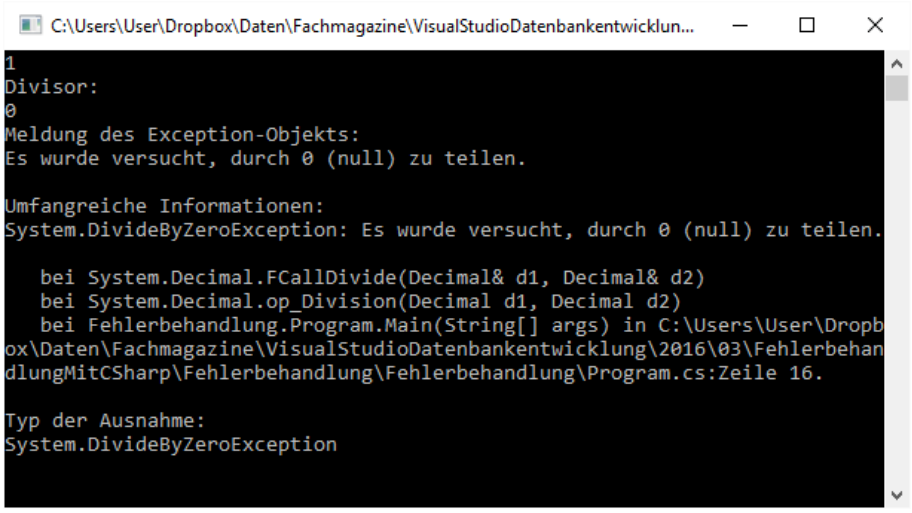


Bild 6: Erweiterte Fehlermeldung des **Exception**-Objekts

```
catch (Exception e) {
    Console.WriteLine("Meldung des Exception-Objekts:\n{0}\n", e.Message);
    Console.WriteLine("Umfangreiche Informationen:\n{0}\n", e.GetBaseException().ToString());
    Console.WriteLine("Typ der Ausnahme:\n{0}", e.GetType().ToString());
}
```

Listing 3: Ausstatten des **catch**-Block mit einigen weiteren Informationen

deByZeroException. Damit können wir später gezielt auf bestimmte Fehler reagieren.

Allgemeine Exception

Was wir im vorherigen Beispiel getan haben, war die Behandlung einer allgemeinen Exception. Die Klasse **Exception** ist quasi die Mutter aller Fehlerklassen. Es gibt eine ganze Reihe von Fehlerklassen, die von dieser Klasse abgeleitet sind.

Wenn Sie im **catch**-Zweig einer Ausnahmebehandlung in Klammern ein Objekt des Typs **Exception** übergeben, fangen Sie damit alle möglichen Ausnahmen ab. Dies entspricht etwa dem **Case Else**-Zweig in einer VBA-Fehlerbehandlung wie der folgenden:

```
Select Case Err.Number
    Case 11 'Geteilt durch Null
        'Fehler behandeln
    Case 0 'kein Fehler, nichts ist zu tun
    Case Else
        'Alle übrigen Fehlernummern behandeln
End Select
```

Nun wollen wir noch wissen, wie wir einen speziellen Fehler abfangen, in diesem Fall die **DivideByZeroException**. Wenn Sie nur diesen einen Fehler abfangen möchten, reicht die folgende Variante aus:

```
try {
    decimal Quotient = Dividend / Divisor;
    Console.WriteLine("Der Quotient lautet: {0}", Quotient);
}
catch (DivideByZeroException e) {
    Console.WriteLine("Der Dividend darf nicht 0 sein.");
}
```

Der **catch**-Zweig wird in diesem Fall nur angesteuert, wenn eine **DivideByZero**-Ausnahme ausgelöst wird. Im Falle eines jeden anderen Fehlers tritt eine unbehandelte Ausnahme auf.

Um einen weiteren, andersartigen Fehler auszulösen und behandeln zu können, ziehen wir die übrigen Anweisungen der Methode wie in Listing 4 ebenfalls in den **try**-Block hinein. Außerdem fügen wir neben dem **catch**-Block, der sich um den Fehler kümmert, der beim Teilen durch **0** auftritt, einen

```
public static void Andere_Exception() {
    try {
        Console.WriteLine("Geben Sie den Dividend und den Divisor als ganze Zahlen ein.");
        Console.WriteLine("Dividend:");
        decimal Dividend = Convert.ToDecimal(Console.ReadLine());
        Console.WriteLine("Divisor:");
        decimal Divisor = Convert.ToDecimal(Console.ReadLine());
        decimal Quotient = Dividend / Divisor;
        Console.WriteLine("Der Quotient lautet: {0}", Quotient);
    }
    catch (DivideByZeroException e) {
        Console.WriteLine("Der Dividend darf nicht 0 sein.");
    }
    catch (Exception e) {
        Console.WriteLine("Es ist ein Fehler aufgetreten: {0}", e.GetType().ToString());
    }
    Console.ReadLine();
}
```

Listing 4: Abfangen eines speziellen und eines allgemeinen Fehlers per **try...catch**-Block

weiteren **catch**-Block hinzu. Dieser soll wieder alle übrigen Fehler abfangen.

Wenn wir nun beispielsweise für die erste **Console.ReadLine**-Anweisung keine Zahl, sondern eine Zeichenkette eingeben, erhalten wir einen weiteren Fehler. Dieser wird ausgelöst, da wir das Ergebnis der Eingabe direkt in einen Wert des Datentyps **decimal** umwandeln wollen. Das gelingt mit einer Zeichenkette natürlich nicht, also liefert dies eine weitere Ausnahme – wie Bild 7 zeigt.

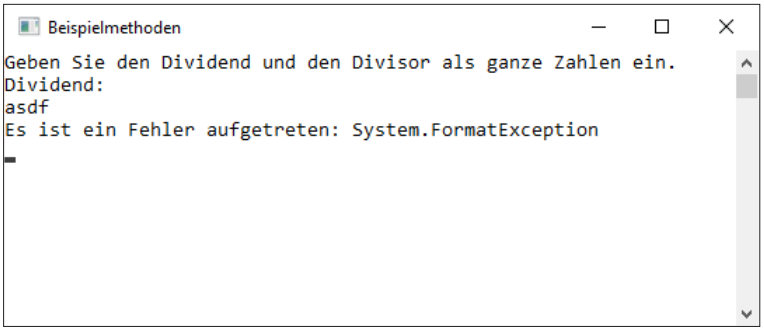


Bild 7: Fehler durch die Eingabe von Daten im falschen Format

Nachdem wir wissen, dass dies die Ausnahme **FormatException** auslöst, fügen wir auch diese als weiteren **catch**-Block zur Fehlerbehandlung hinzu:

```
try {
    ...
}
catch (DivideByZeroException e) {
    Console.WriteLine("Der Dividend darf nicht 0 sein.");
}
catch (FormatException e) {
    Console.WriteLine("Bitte geben Sie eine ganze Zahl ein.");
}
catch (Exception e) {
    Console.WriteLine("Es ist ein Fehler aufgetreten:
        {0}", e.GetType().ToString());
}
```

Auf ähnliche Weise legen wir für alle Fehler, die auftreten können, eine Ausnahmebehandlung an (zumindest für die, die wir bereits erkennen können – der Benutzer wird sicher noch weitere Schwachstellen aufdecken ...).

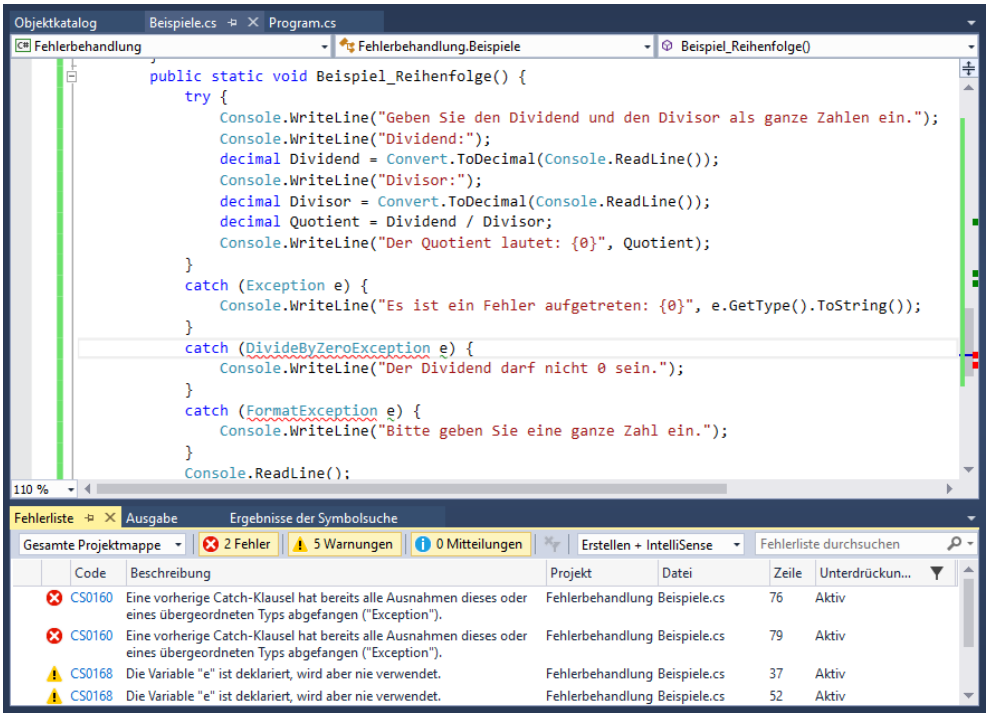


Bild 8: Exceptions müssen in der Hierarchie von unten nach oben abgearbeitet werden.

Die Reihenfolge der **catch**-Blöcke spielt natürlich eine übergeordnete Rolle, gerade wenn Sie eine allgemeine Ausnahmebehandlung (mit **catch (Exception e)**) und speziellere Ausnahmebehandlungen (wie mit **catch (DivideByZeroException)**) implementieren. Die catch-Blöcke werden nämlich immer in der angegebenen Reihenfolge abgearbeitet. Wenn Sie also direkt in der ersten

catch-Anweisung auf die allgemeine Ausnahme **Exception** prüfen, wird diese bei jeder denkbaren Ausnahme angesteuert, da ja alle anderen Ausnahmen von **Exception** erben. Aber Visual Studio beugt dem direkt vor, denn untergeordnete Ausnahmen müssen immer vor übergeordneten Ausnahmen abgearbeitet werden. Wenn wir versuchen, die allgemeine **Exception**-Ausnahme vor der **DivideByZeroException**-Ausnahme zu behandeln, erhalten wir einen Kompilierfehler (siehe Bild 8).

Hierarchie der Exceptions

Damit kommen wir zurück zur Mutter aller Fehlerklassen und den untergeordneten Elementen. Woher erfahren wir, in welcher Hierarchie-Ebene sich eine **...Exception**-Klasse befindet und welche die übergeordneten Klassen sind?

Dazu klicken Sie etwa im Code-Fenster mit der rechten Maustaste auf den Text **DivideByZeroException** und wählen dort den Eintrag **Definition einsehen** aus. Dies öffnet einen gelb hinterlegten Bereich, der die öffentliche Klasse beschreibt, die von der Klasse **ArithmeticException** abgeleitet ist (siehe Bild 9). Diese Abhängigkeit erkennen Sie an dem Doppelpunkt zwischen den beiden Klassennamen (**DivideByZeroException : ArithmeticException**).

Die Klasse **ArithmeticException** wird also vermutlich sämtliche Ausnahmen behandeln, die durch die Verwendung von Rechenoperatoren unter C# ausgelöst werden, wobei **DivideByZeroException** nur ein Spezialfall ist.

Es geht aber noch weiter: Wenn Sie in der Definition mit der rechten Maustaste auf den Eintrag **ArithmeticException** klicken und dann wiederum den Kontextmenü-Eintrag **Definition einsehen** auswählen, erhalten Sie die Definition der Klasse **ArithmeticException** und mit **SystemException** die übergeordnete Ausnahmeklasse. Dass die Hierarchie einigermaßen flach ist, erkennen Sie, wenn Sie auch noch die übergeordnete Klasse von **SystemException** ermitteln – hier landen Sie dann nämlich bei der Klasse **Exception**, die keine weitere übergeordnete Ausnahmeklasse besitzt.

Exception mit oder ohne Variable?

In allen bisherigen Beispielen haben wir das **Exception**-Objekt in einer Variablen namens **e** gespeichert, um innerhalb des **catch**-Blocks auf die Eigenschaften des Ausnahme-Objekts zugreifen zu können. Diese Variable können Sie auch weglassen, wenn Sie gar nicht auf die Eigenschaften zugreifen wollen. Meist ist dies nicht notwendig – im Falle der **DivideByZeroException** etwa gibt es in der betroffenen Methode ja nur eine Zeile, die diesen Fehler auslösen kann, und der Grund dafür ist dann auch offensichtlich. Sie können e also auch einfach weglassen:

```
...
catch (DivideByZeroException) {
    Console.WriteLine("Der Dividend darf nicht 0 sein.");
}
...
```

In diesem Fall erscheinen dann auch keine Warnungen wie **Die Variable "e" ist deklariert, wird aber nie verwendet.** mehr in der Fehlerliste von Visual Studio.

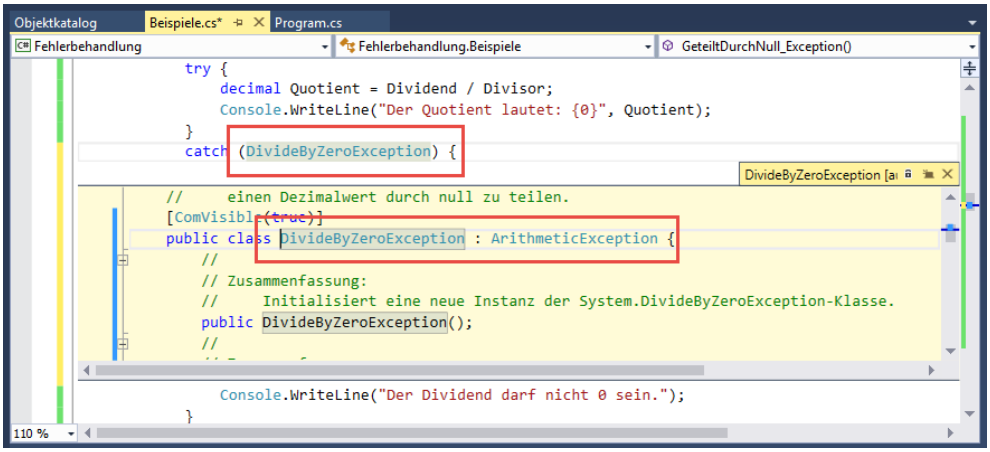


Bild 9: Die Definition einer Klasse liefert Hinweise auf die übergeordnete Klasse

Objektorientierte Programmierung: Interfaces

Interfaces beziehungsweise Schnittstellen bezeichnen in der Computerwelt eine Menge verschiedener Dinge. Es gibt eine Benutzerschnittstelle, Softwareschnittstellen, Maschinenschnittstellen und mehr. Uns interessiert das Interface, wie es in der objektorientierten Programmierung genutzt wird. Schnittstellen dienen dort als zuverlässige Definition der verfügbaren Methoden, Eigenschaften oder Ereignisse für eine oder mehrere Klassen, welche diese tatsächlich definieren.

Auf diese Weise kann man sich beim Programmieren »gegen« eine Schnittstelle darauf verlassen, dass die Klasse, welche die Schnittstelle implementiert, auch die dort definierten Methoden, Eigenschaften, Ereignisse und Indexer enthält (Indexer haben Sie im Rahmen dieses Magazins noch nicht kennen gelernt, daher gehen wir in diesem Artikel nicht auf diese Elemente ein).

Wozu aber sollte man überhaupt verschiedene Klassen programmieren, die alle die gleichen Methoden, Eigenschaften oder Ereignisse enthalten und somit die Schnittstelle implementieren? Ein Grund ist, dass Sie nur einmal lernen möchten, wie etwas funktioniert (die in der Schnittstelle definierten Elemente), die Implementierung jedoch je nach Anwendungsfall variieren kann.

Aus dem Leben gegriffen

Stellen Sie sich das Anziehen oder Lösen einer Schraube vor: Sie verwenden den für die Schraube passenden Schraubendreher, zum Beispiel einen Kreuzschlitz- oder einen Längsschlitz-Schraubendreher. Nun haben beide Schraubendreher vielleicht unterschiedliche Griffe, wodurch Sie jeweils anders greifen müssen. Nun stellen Sie sich einen Schraubendreher mit austauschbaren Bits vor: Die (Benutzer-)Schnittstelle ist der Griff, mit dem Sie den Schraubendreher nach links oder rechts drehen können, aber je nach Schraube verwenden Sie ein anderes Bit, um diese anzuziehen oder zu lösen. Sie kennen also die beiden Methoden der Schnittstelle, also nach links und nach rechts drehen, und können verschiedene Implementierungen nutzen – also etwa ein Bit für Kreuzschlitz- und ein anderes für Längsschlitz-Schrauben. Genau genommen ist auch die Schnittstelle zwischen dem Schraubendreher und dem Bit ein Interface – wir haben also gleich ein doppeltes Beispiel für ein Interface.

Beispiel: E-Mail versenden

Genau das Gleiche kann in der Software-Welt geschehen: Sie möchten beispielsweise eine E-Mail verschicken und dazu einem Objekt den Empfänger, den Betreff und den Inhalt übergeben (die Eigenschaften) und die Mail dann mit der **Senden**-Methode abschicken. Welcher Mechanismus dahinter steckt, soll für den Aufruf uninteressant sein – es könnte sich um die Programmierung von Outlook handeln, um die Mail mit diesem Programm zu versenden, aber vielleicht verwendet eine andere Implementierung nicht Outlook, sondern eine speziell für den Versand von E-Mails vorgesehene SMTP-Schnittstelle.

Beispiel: Daten abrufen

Oder, was in diesem Magazin auch bereits das eine oder andere Mal beschrieben wurde: Sie möchten ein Objekt mit Daten füllen, also etwa ein Kunden-Objekt mit den Eigenschaften **KundeID**, **Vorname** und **Nachname**. Wenn Sie mit der Programmierung beginnen, sollen die Daten aus einer Access-Datenbank kommen. Aber vielleicht wechseln Sie demnächst zu einer SQL Server-Datenbank? Oder lesen die Daten aus einer XML-Datei ein? Dann benötigen Sie in jedem Fall unterschiedliche Implementierungen der Methoden, da der Zugriff auf die Daten ja nicht auf die gleiche Art erfolgt.

Verschiedene Methoden oder verschiedene Implementierungen?

Natürlich könnten Sie, um beim Beispiel des Abrufs von Daten aus verschiedenartigen Datenquellen zu bleiben, für

jeden Zugriff eine andere Methode implementieren – also beispielsweise **HoleKundeAccess**, **HoleKundeSQLServer** oder **HoleKundeXML**. Dann müssten Sie allerdings auch irgendwo eine Bedingung unterbringen, die prüft, welche der Methoden zum Abrufen eines Kunden-Objekts aus einer der verschiedenen Datenquellen zum Einsatz kommen soll.

In der Objektorientierung soll dies jedoch anders ablaufen. Hier definieren Sie jeweils eine Klasse mit Methoden für den Zugriff auf die Daten der verschiedenen Datenquellen, also etwa Access, SQL Server oder XML. Dort sollen natürlich für jede Datenquellenart noch weitere Eigenschaften, Methoden oder auch Ereignisse definiert werden. Würden Sie diese alle in einer Klasse unterbringen, hätten Sie erstens eine recht unübersichtliche Anhäufung von Elementen. Außerdem gäbe es keine Prüfung, ob tatsächlich alle Methoden, Eigenschaften und Ereignisse für die verschiedenen Datenquellen definiert wurden.

Wenn Sie hingegen eine Schnittstelle beziehungsweise ein Interface definieren und für jede Datenquellenart eine eigene Klasse erstellen, welche die in der Interface-Klasse definierten Methoden, Eigenschaften und Ereignisse implementiert, erhalten Sie automatisch eine Prüfung, ob alle benötigten Elemente vorliegen. Dazu müssen Sie beim Programmieren der Klasse nur angeben, welche Schnittstelle diese implementiert.

Interfaces zur Einschränkung

Es gibt noch einen weiteren Grund, eine Schnittstelle für eine Klasse zu programmieren. Die Grundlagen dazu finden Sie im Artikel **C#-DLL für COM/VBA erstellen**. Wenn Sie eine DLL mit Visual Studio erstellen, wollen Sie nur genau die dafür vorgesehenen Eigenschaften, Methoden und Ereignisse bereitstellen. Ohne eine Schnittstelle ist dies jedoch nicht so einfach möglich, denn die Standardelemente eines Objekts werden ebenfalls angeboten. Mithilfe eines Interfaces können Sie die zu veröffentlichenden Elemente gezielt einschränken.

Einfaches Beispiel für ein Interface

Für ein Beispiel einer Schnittstelle wollen wir verschiedene Methoden und Eigenschaften nutzen. Diese definieren wir in der Interface-Klasse, die sich von einer normalen Klasse vor allem in drei Punkten unterscheidet:

- Sie verwendet das Schlüsselwort **Interface** statt **Class**.
- Sie definiert nur die Member der Klasse, aber enthält keinerlei Code.
- Ihr Name sollte mit einem großen **i** beginnen (**I**).

Interfaces können Sie als **internal** (Standard) oder **public** deklarieren. Ein mit **internal** deklariertes Interface ist innerhalb der Anwendung nutzbar, ein als **public** deklariertes auch von außen – also beispielsweise, wenn ein COM-Client auf eine entsprechend ausgestattete C#-DLL zugreift.

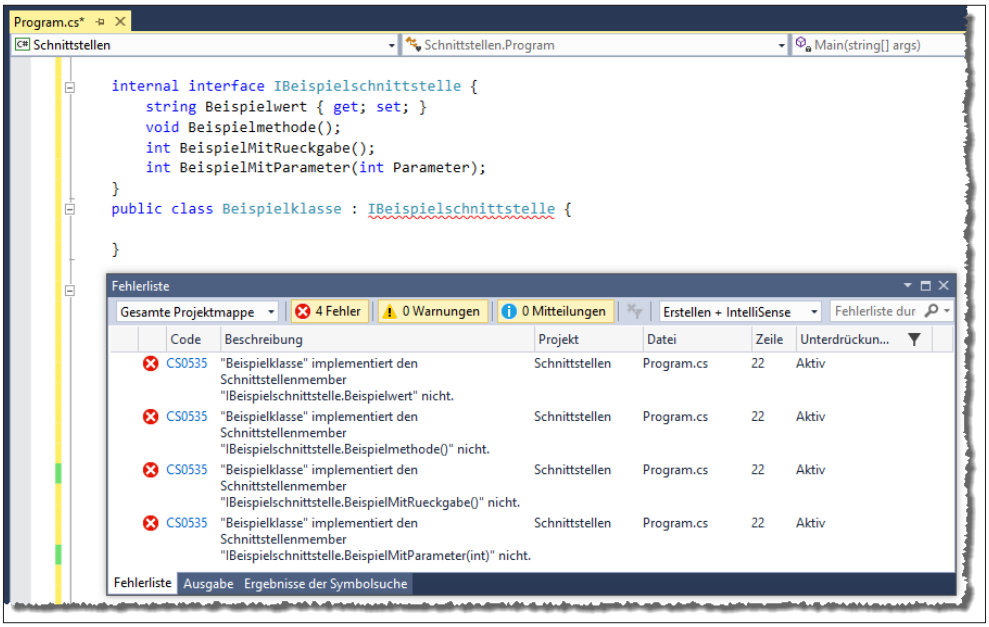


Bild 1: Fehlermeldungen für nicht implementierte Schnittstellenmember

Unser Beispielinterface sieht wie folgt aus:

```
public interface IBeispielschnittstelle {
    string Beispielwert { get; set; }
    void Beispielmethode();
    int BeispielMitRueckgabe();
    int BeispielMitParameter(int Parameter);
}
```

Damit deklarieren wir eine Eigenschaft, eine Methode, eine Methode mit Rückgabewert und eine Methode mit Parameter und Rückgabewert. Auffällig ist, dass keine der Definitionen einen Zugriffsmodifizierer wie **public** oder **private** enthält. Aber wozu auch? Es ist eine Schnittstelle, da wollen Sie ja ohnehin auf alle Elemente zugreifen. Diese sind somit standardmäßig öffentlich deklariert.

Schnittstelle implementieren

Um die Schnittstelle in Form einer Klasse zu implementieren, erstellen Sie die erste Zeile der Klasse und fügen hinter dem Klassenbezeichner einen Doppelpunkt gefolgt vom Namen der Schnittstelle ein:

```
public class Beispielklasse :
        IBeispielschnittstelle {
}
```

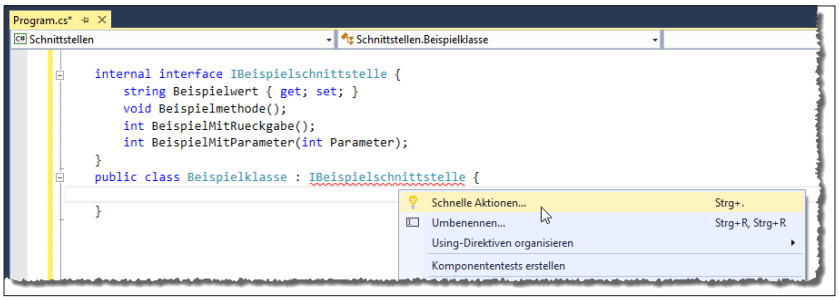


Bild 2: Aufrufen des Kontextmenü-Befehls **Schnelle Aktionen...**

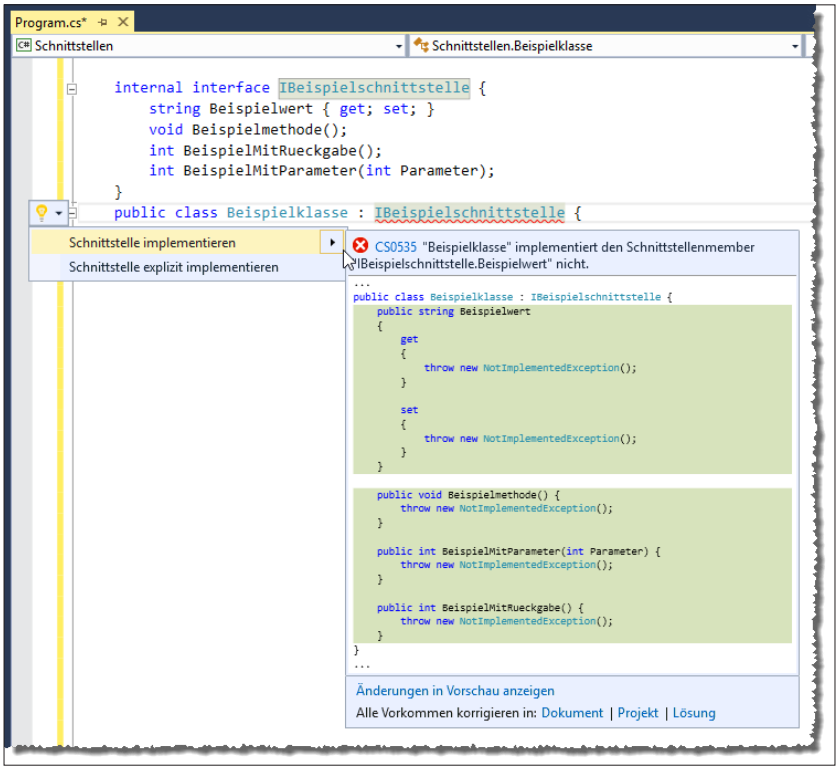


Bild 3: Vorschau der zu erstellenden Elemente

Dies sieht also genauso aus, als wenn Sie von einer Klasse erben – nur, dass es sich nicht um eine echte Klasse, sondern um ein Interface handelt. Sobald Sie das Interface angelegt haben, erscheinen einige Fehlermeldungen im Bereich **Fehlerliste** (siehe Bild 1). Das liegt im Wesen der Implementation einer Schnittstelle: Wenn man dies tut, muss man auch alle in der Schnittstelle definierten Elemente implementieren. Aber wie machen wir das?

Sie könnten die notwendigen Methoden natürlich von Hand anlegen. Aber die Arbeit können Sie sich sparen – zumindest den fehleranfälligen Teil davon, nämlich das Eingeben der Methoden auf Basis der Schnittstellenelemente.

Dazu wählen Sie einfach den Eintrag **Schnelle Aktionen...** aus dem Kontextmenü des Namens der Schnittstelle aus der Kopfzeile der Klasse aus (siehe Bild 2).

Danach erscheint ein Popup-Menü, das zum Beispiel den Befehl **Schnittstelle implementieren** anbietet. Während dieser Eintrag aktiviert ist, zeigt ein weiteres Popup-Element eine Vorschau des zu erstellenden Codes an (siehe Bild 3).

C#-DLL für COM/VBA erstellen

Das .NET-Framework bietet im Vergleich zu VBA eine schier unendliche Menge nutzbarer Bibliotheken mit Objekten, Methoden und Eigenschaften für die verschiedensten Anwendungsfälle. Wer noch nicht komplett auf eine .NET-Anwendung umsteigen möchte, mag sich aber vielleicht die Möglichkeiten des Frameworks unter VBA erschließen. Dazu programmieren Sie eine DLL-Bibliothek, welche die benötigen .NET-Elemente enthält und für externe Anwendungen wie etwa eine Access-Datenbank bereithält. Der vorliegende Artikel erklärt, wie dies funktioniert.

Grundlagen

Wenn es darum geht, von einem VBA-Projekt etwa auf die Methoden einer mit C# programmierten DLL zuzugreifen, benötigen Sie eine Technologie namens COM (Component Object Model). Diese wurde von Microsoft bereits vor einer ganzen Weile eingeführt, nämlich in den frühen neunziger Jahren. Das Ziel war, dass Komponenten, die mit verschiedenen Programmiersprachen programmiert wurden, miteinander kommunizieren können.

COM liefert die Vorgaben für die Schnittstellen, die diese Interaktion ermöglichen. Damit nun auch noch Komponenten, die auf .NET basieren, mit COM-Objekten zusammenarbeiten, gibt es eine weitere Technologie, die sich COM Interop nennt. Diese erlaubt es beispielsweise, die in einer .NET-DLL definierten Objekte, Methoden und Eigenschaften auch für eine COM-Komponente zugänglich zu machen.

Eine .NET-Komponente ist beispielsweise eine DLL-Datei. Damit Sie von COM-Objekten auf diese DLL-Datei zugreifen können, müssen Sie zunächst eine COM-Type Library (**.tlb**) erstellen, welche die relevanten Informationen bereithält. Dies erledigen Sie mit einem Befehlszeilentool namens **REGASM.EXE** oder **TLBEXP.EXE**.

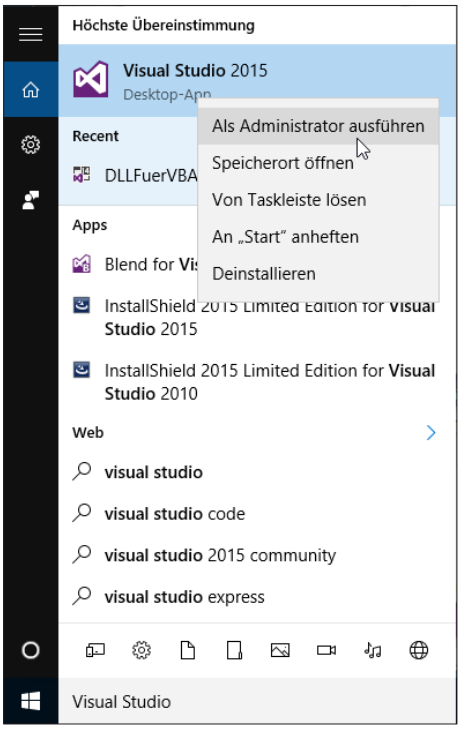


Bild 1: Visual Studio als Administrator starten

Wichtig: Admin-Modus!

Wenn Sie eine DLL erstellen und registrieren möchten, müssen Sie Visual Studio als Administrator ausführen. Dazu klicken Sie mit der rechten Maustaste auf den Visual Studio-Eintrag, der beispielsweise erscheint, wenn Sie im Suchfeld den Suchbegriff **Visual Studio** eingeben. Im folgenden Kontextmenü wählen Sie den Eintrag **Als Administrator ausführen** aus (siehe Bild 1). In diesem Modus können die notwendigen Registry-Einträge gesetzt werden.

Stolperfalle: DLL in Verwendung

Sollten Sie die DLL bereits einmal erstellt haben, wollen Sie diese natürlich im VBA-Projekt einer Access-Anwendung testen. Wenn Sie die DLL etwa per Verweis referenzieren und die Datenbank geöffnet ist, kann die referenzierte DLL nicht überschrieben werden, da diese schreibgeschützt ist. Sollten Sie das C#-Projekt mit der DLL zu diesem Zeitpunkt also in Visual Studio editieren und neu erstellen wollen, gelingt dies nicht, da die DLL ja nicht überschrieben werden kann. Dies erfahren Sie dann auch im Bereich **Fehlerliste** (siehe Bild 2).

Also, auch wenn es mit der Zeit nervt: Nicht vergessen, die Anwendung, mit der Sie die DLL testen, vor dem Neuerstellen der DLL zu schließen.

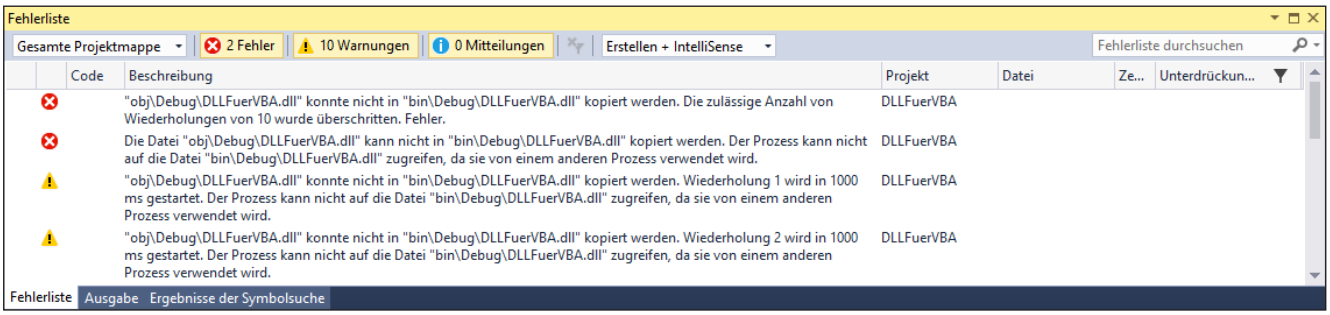


Bild 2: Fehler beim Versuch, eine aktuell in Verwendung befindliche DLL neu zu erstellen

Aber: Es gibt eine einfachere Methode, die wir weiter unten nach der Erstellung des ersten Beispiels erläutern.

DLL erstellen

Um eine DLL zu erstellen, deren Funktionen Sie später von einem VBA-Projekt nützen können, legen Sie in Visual Studio ein Projekt auf Basis der Vorlage **Klassenbibliothek** an (siehe Bild 3). Ändern Sie den Namen der automatisch hinzugefügten Klassendatei namens **Class1** in **LateBinding** um und erledigen Sie dies auch für den Klassennamen selbst.

Danach fügen Sie dem Klassenmodul den folgenden einfachen Code hinzu. Die Klasse deklariert zwei Variablen namens **zahl1** und **zahl2**, die über die Eigenschaften **Zahl1** und **Zahl2** gefüllt werden sollen. Die Methode **Produkt** soll dann das Produkt der beiden Zahlen zurückliefern:

```
namespace DLLFuerVBA {
    public class NoIntelliSense {
        int zahl1;
        int zahl2;
        public int Zahl1 {
            set { zahl1 = value; }
        }
        public int Zahl2 {
            set { zahl2 = value; }
        }
        public int Produkt() {
            return zahl1 * zahl2;
        }
    }
}
```

Dass wir die Klasse **NoIntelliSense** nennen, hat seinen Grund – mehr dazu weiter unten.

Der Clou ist nun, dass diese Klasse nicht innerhalb des C#-Projekts genutzt werden soll, sondern von einem VBA-Projekt in einer Access-Anwendung. Dazu sind zwei bereits durchgeführte und noch zwei weitere Schritte notwendig:

- Der erste bereits erledigte Schritt ist, dass die Klassendefinition als **Public** deklariert wird.
- Der zweite Schritt war, dass auch die Member der Klasse, die von außen erreicht werden sollen, das Schlüsselwort **Public** enthalten.
- Der dritte Schritt ist, dass Sie die Option **Für COM-Interop registrieren** aktivieren (siehe weiter unten).

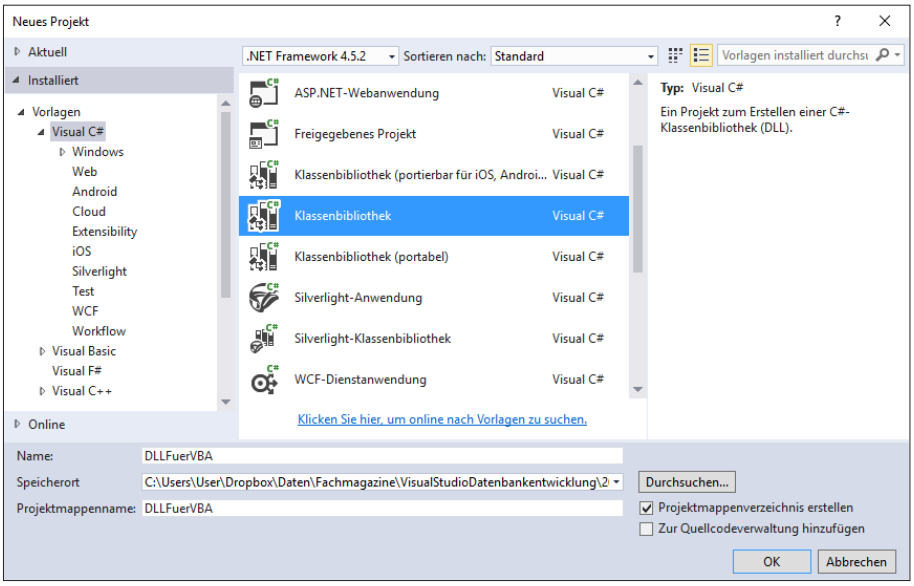


Bild 3: Hinzufügen eines Projekts, das wir als DLL verwenden können

- Der vierte Schritt ist das Aktivieren der Option **Assembly COM-sichtbar machen** (siehe ebenfalls weiter unten).

Für COM-Interop registrieren

Die Option **Für COM-Interop registrieren** finden Sie, wenn Sie im Projektmappen-Explorer mit der rechten Maustaste auf den Projektnamen klicken, den Eintrag **Eigenschaften** anklicken und im dann erscheinenden Fenster zum Bereich **Erstellen** wechseln. Unten unter **Ausgabe** finden Sie die gesuchte Option, die Sie per Klick auf das Kontrollkästchen aktivieren (siehe Bild 4). Wenn Sie diese Option nicht aktivieren, erstellt Visual Studio beim Erstellen des Projekts im Unterordner **\bin\debug** nur die übliche **.dll**-Datei plus einer **.pdb**-Datei. Damit Sie die DLL von einem VBA-Projekt aus nutzen können, müssen Sie diese jedoch entsprechend registrieren. Das können Sie auf zwei Arten erledigen – entweder mit einem Kommandozeilentool oder durch die Aktivierung der obigen Option vor dem Erstellen des Projekts. Dabei werden zwei Schritte durchgeführt: Visual Studio erstellt eine weitere Datei mit der Endung **.tlb**, welche die Definition der Schnittstelle enthält, die Sie später vom VBA-Projekt aus nutzen können. Außerdem legt Visual Studio auch gleich einige Einträge in der Registry an, damit Sie die Bibliothek über den Verweise-Dialog des VBA-Editors auswählen können.

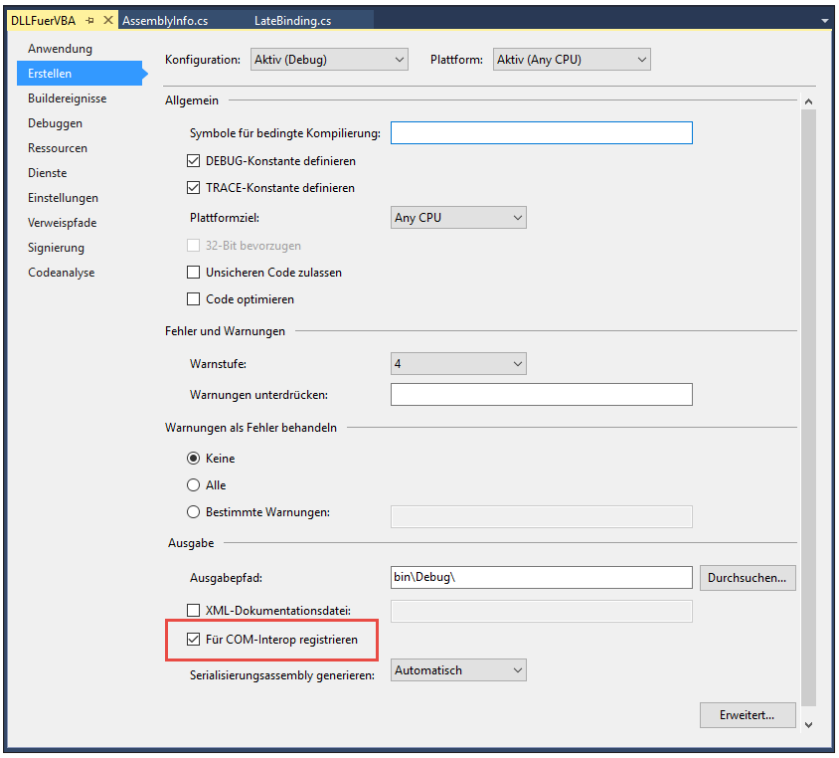


Bild 4: Einstellen der Option **Für COM-Interop registrieren**

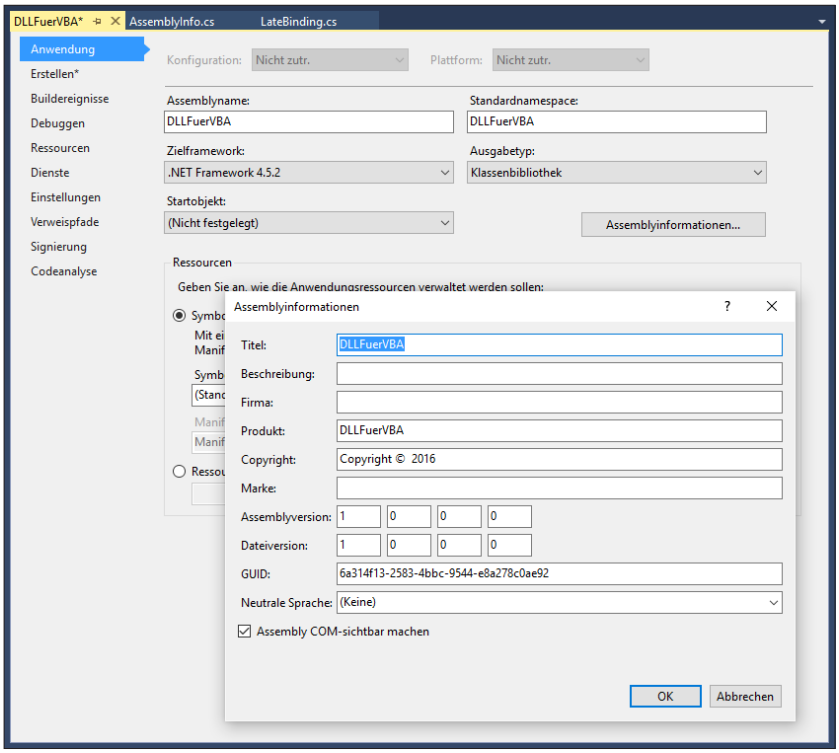


Bild 5: Aktivieren der Option **Assembly COM-sichtbar machen**

Assembly COM-sichtbar machen

Der zweite notwendige Schritt ist wieder mit dem Setzen einer einfachen Option erledigt. Diese ist jedoch etwas besser versteckt. Wechseln Sie in den Projekteigenschaften zum Bereich **Anwendungen** und klicken Sie dort auf die Schaltfläche **Assemblyinformationen**. Dies öffnet den Dialog **Assemblyinformationen**, der wie in Bild 5 aussieht. Hier aktivieren Sie

im unteren Bereich die Option **Assembly COM-sichtbar machen** und schließen den Dialog wieder.

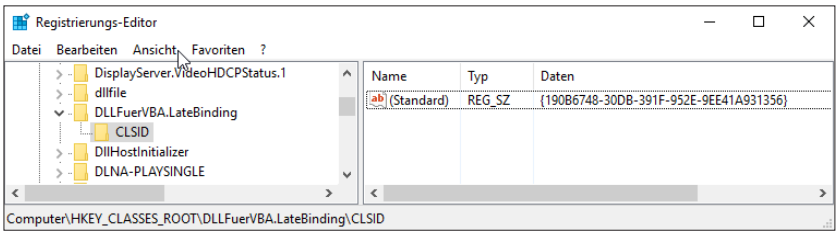


Bild 6: Erster Eintrag in der Registry

Einfaches Projekt erstellen

Wenn Sie nun das Projekt mit dem obigen Code erstellen, legt Visual Studio einige Einträge in der Registry des aktuellen Rechners an. Der erste Eintrag, der für uns wichtig ist, befindet sich in der Registry, die Sie mit dem Befehl **RegEdit** öffnen, unter dem Pfad **HKEY_CLASSES_ROOT\DLLFuerVBA.LateBinding**. Die wichtige Information dieses Pfades befindet sich im Schlüssel CLSID. Dort finden Sie nämlich eine GUID, die auf den Schlüssel mit weiteren Informationen hinweist (siehe Bild 6).

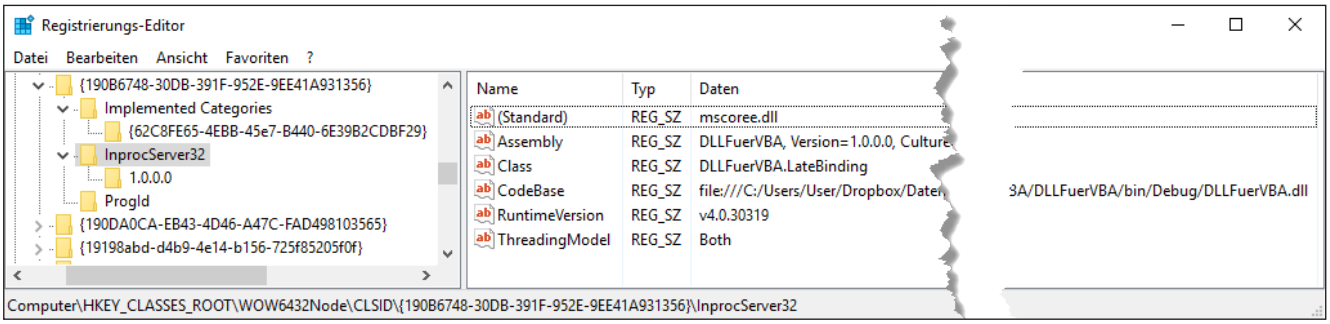


Bild 7: Details zur Registrierung der DLL

Diesen finden Sie in unserem Beispiel unter folgendem Pfad (die GUID kann variieren): **HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{190B6748-30DB-391F-952E-9EE41A931356}**. Unterhalb dieses Schlüssels finden Sie den Schlüssel **InprocServer32**, der einige Informationen liefert (siehe Bild 7).

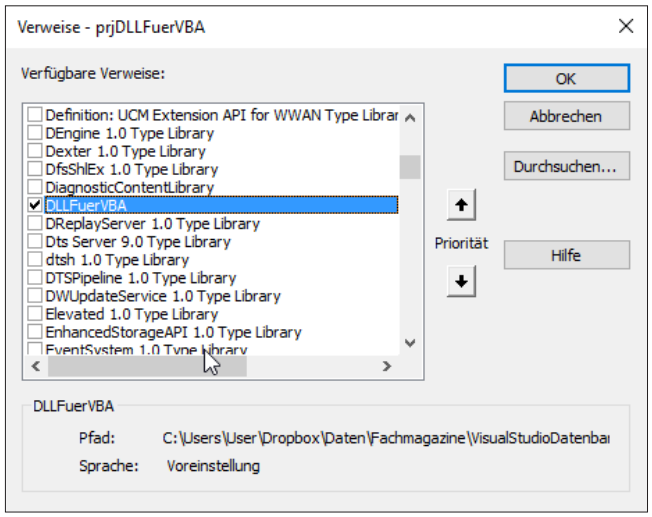


Bild 8: Der Verweis auf die Objektbibliothek lässt sich bereits setzen.

Objektbibliothek verfügbar

Zu diesem Zeitpunkt können Sie auch bereits einen Verweis vom VBA-Projekt einer Access-Datenbank auf die DLL beziehungsweise die erstellte **.tlb**-Datei setzen – dazu öffnen Sie einfach den **Verweise**-Dialog (VBA-Editor, Menüeintrag **Extras|Verweise**) und suchen den Eintrag **DLLFuerVBA** (siehe Bild 8).

Wenn wir uns das Objekt nun allerdings im Objektkatalog ansehen, finden wir dort noch keine Eigenschaften, Methoden oder Ereignisse (siehe Bild 9).

Nun erfahren Sie, warum wir unsere erste Klasse **NoIntelliSense** genannt haben: Wie zu erwarten, finden sich nicht nur keine Elemente im Objektkatalog, sondern auch der Versuch, per IntelliSense auf die Eigenschaften eines Objekts auf Basis der Klasse **DLLFuerVBA.NoIntelliSense** zuzugreifen, schlägt fehl. Immerhin können wir die Eigenschaften **Zahl1** und **Zahl2** sowie die Methode **Produkt** überhaupt nutzen:

```
Public Sub Test_EarlyBinding()
    Dim obj As DLLFuerVBA.NoIntelliSense
    Set obj = New DLLFuerVBA.NoIntelliSense
    With obj
        .Zahl1 = 2
        .Zahl2 = 3
        Debug.Print .Produkt
    End With
End Sub
```

Diese VBA-Prozedur gibt den erwarteten Wert **6** im Direktfenster aus. Nun fehlt uns allerdings noch die IntelliSense-Unterstützung und auch die Anzeige der Elemente des Objekts im Objektkatalog. Darum kümmern wir uns nach einem kleinen Ausflug in die Grundlagen für die Beziehung zwischen .NET und COM.

Das Bindeglied zwischen .NET und COM-Clients

Wenn Sie eine .NET-Anwendung erstellen möchten, auf deren Objekte Sie von einem COM-Client wie einer Access-Anwendung zugreifen wollen, benötigen Sie ein Bindeglied zwischen dem Managed Code von .NET und dem Unmanaged COM -Client.

Dieses heißt **COM Callable Wrapper** (**CCW**) und kann als eine Art Proxy angesehen werden. Dieser soll dafür sorgen, dass Sie mit einem COM-Client auf eine .NET-Klasse (oder auch eine **.exe**-Datei) wie auf ein anderes COM-Objekt zugreifen können. Dabei kümmert es sich um die korrekte Zuordnung der Datentypen, die sich ja auf beiden Seiten unterscheiden können, und stellt eine COM-Schnittstelle zur Verfügung. Außerdem organisiert es im Hintergrund die Mechanismen, die dafür sorgen, dass ein .NET-Objekt nach erfolgtem Zugriff gegebenenfalls beendet und der Speicher freigegeben wird.

Schnittstelle zwischen COM und .NET

Damit unser COM-Client, also beispielsweise das VBA-Projekt, mit der .NET-DLL kommunizieren kann, benötigen wir eine Schnittstelle. Diese kommt in Form der **.tlb**-Datei beziehungsweise als Type Library. Die **.tlb**-Datei wird beim

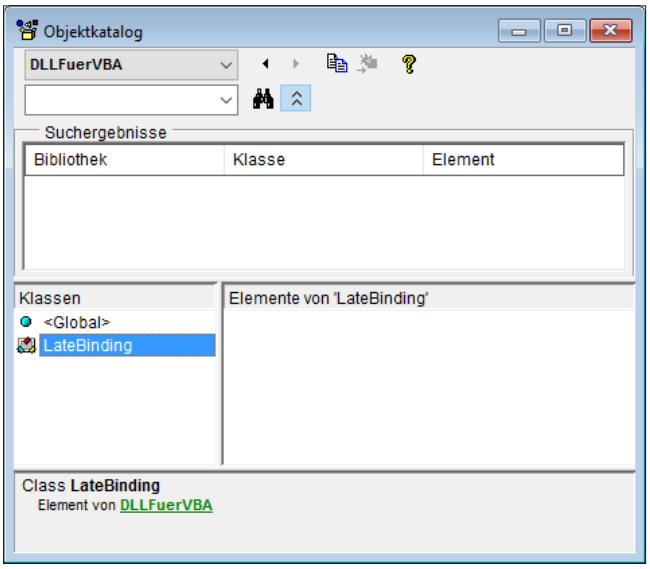


Bild 9: Der Objektkatalog offenbart keinerlei Eigenschaften, Methoden oder Ereignisse.

Erstellen des Projekts in Visual Studio automatisch generiert, wenn Sie die obigen Optionen **Für COM-Interop registrieren** und **Assembly COM-sichtbar machen** aktiviert haben. Alternativ können Sie die Erstellung dieser **.tlb**-Datei sowie die Registrierung mit einem Befehlszeilentool wie **tlbexp.exe** oder **regasm.exe** durchführen. Die Erstellung der Type Library basiert dabei auf dem Inhalt des Interface Definition Language-Codes in der von Visual Studio erstellten DLL.

IDL-Code ansehen

Wenn Sie genau wissen möchten, wie sich die Optionen und auch die weiter unten vorgestellten verschiedenen Deklarationen von Klassen und Membern auswirken, können Sie sich den Inhalt einer COM-DLL ansehen. Dazu verwenden Sie das Tool **OleView.exe**, das Sie etwa unter folgendem Link bei Microsoft herunterladen können: **http://download.microsoft.com/download/win2000platform/oleview/1.00.0.1/NT5/EN-US/oleview_setup.exe**

Nach der Installation finden Sie das Tool im Verzeichnis **C:\Program Files (x86)\Resource Kit** unter **oleview.exe**. Gegebenenfalls müssen Sie noch eine weitere DLL namens **iviewers.dll** herunterladen und im gleichen Verzeichnis speichern. Diese DLL findet sich an verschiedenen Stellen im Internet, hier empfiehlt sich eine Google-Suche.

Starten Sie **OleView.exe**, erhalten Sie das Fenster aus Bild 10.

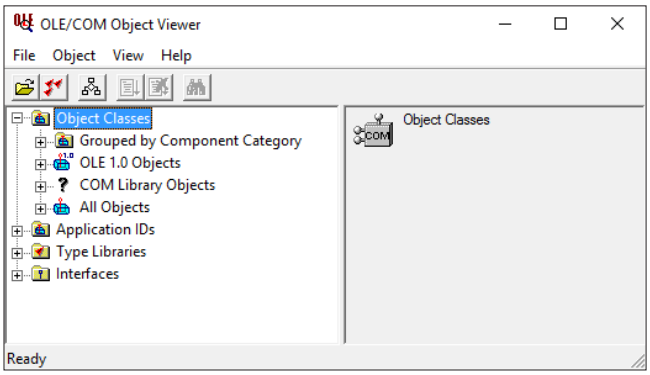


Bild 10: Das Tool **OleView.exe**

Der für uns interessante Bereich heißt **All Objects**. Erweitern Sie diesen, klicken auf das erste Objekt und geben dann die ersten Zeichen des Projektnamens ein, landen Sie schnell beim gesuchten Eintrag (siehe Bild 11). Was Sie hier sehen, sind die verschiedenen Interfaces für das Objekt. Auch wenn wir im C#-Projekt selbst kein Interface, also keine Schnittstelle definiert haben, wird eine Standard-Schnittstelle festgelegt – in diesem Fall **_NoIntelliSense**. Nun wollen wir uns den IDL-Code für diese Schnittstelle ansehen. Dazu klicken Sie mit der rechten Maustaste auf den Eintrag **_NoIntelliSense** und wählen aus dem Kontextmenü den Eintrag **View** aus. Im folgenden Dialog klicken Sie auf **View Type Info**.

Nun öffnet sich das Fenster **ITypeInfo Viewer** (siehe Bild 12) und zeigt die IDL-Definition für die Standard-Schnittstelle unserer Klasse an. Der IDL-Code sieht überschaubar aus: Hier findet sich kein Hinweis auf unsere beiden Eigenschaften **Zahl1** und **Zahl2** und auch nicht auf die Methode **Produkt**.

Keine Schnittstelle?

Warum ist dies der Fall und warum werden unsere Methoden und Eigenschaften nicht unter VBA angezeigt? Der Grund ist die Wahl der Schnittstelle – beziehungsweise die Wahl keiner Schnittstelle. Sie könnten die Klasse mit einer Klassenschnittstelle versehen, aber im aktuellen Zustand ist dies nicht der Fall. Um dies zu ändern, erstellen wir eine neue Klasse, diesmal namens **EarlyBinding**, aber mit dem gleichen Inhalt. Wir fügen der Klasse nur zwei Zeilen hinzu:

```
namespace DLLFuerVBA {
    using System.Runtime.InteropServices;
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class EarlyBinding {
        int zahl1;
        int zahl2;
        public int Zahl1 {
            set { zahl1 = value; }
        }
    ...
    }
}
```

Die Zeile mit der **using**-Direktive fügt den Namespace **System.Runtime.InteropServices** zum Namespace **DLLFu-**

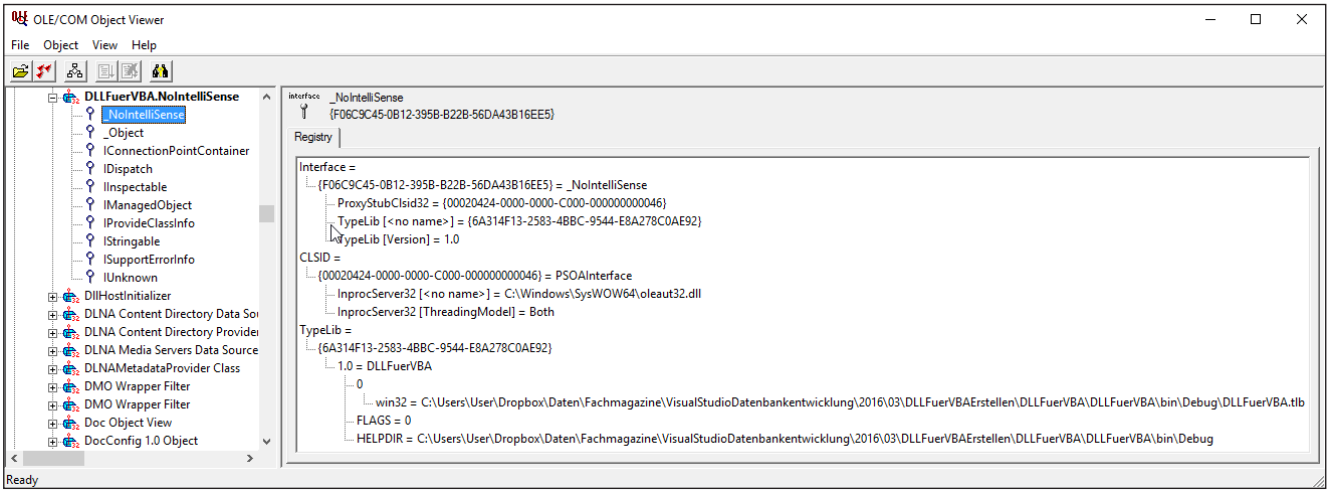


Bild 11: Die Standard-Schnittstelle unserer DLL

WPF-Datenbindung: Einfache Objekte

Wenn Sie WPF-Anwendungen erstellen wollen, wollen Sie die Daten der Anwendung in entsprechenden Fenstern darstellen. Ein Weg zu diesem Ziel ist das Bereitstellen der Daten in Form von Objekten auf Basis von Klassen, welche die Eigenschaften des jeweiligen Objekts beschreiben. Dieser Artikel zeigt, welche Schritte nötig sind, um ein Objekt auf Basis einer einfachen Klasse zu erstellen und seine Daten in den Steuerelementen eines WPF-Fensters anzuzeigen.

WPF-Anwendung erstellen

Da wir nicht nur ein Objekt erstellen, sondern auch seine Daten in einem WPF-Fenster abbilden möchten, benötigen wir ein Projekt auf Basis der Vorlage **Visual C#|WPF-Anwendung**, die wir im Falle des Beispielprojekts **WPFDatenbindung** benannt haben.

Beispielklasse für einen Kunden erstellen

Gleich danach erstellen wir die Beispielklasse, welche die Definition für die Kunden-Objekte enthält. Diese fügen Sie beispielsweise über den Kontextmenü-Eintrag **Hinzufügen|Klasse...** des Projektnamens im Projektmappen-Explorer hinzu. Legen Sie den Namen **Kunde** für die Klasse fest.

Die neu erstellte Klasse ergänzen Sie wie in Listing 1 angegeben. Dort finden Sie drei private Eigenschaften namens **vorname**, **nachname** und **geburtsdatum**. Für alle drei haben wir Eigenschaftsmethoden angelegt (**Vorname**, **Nachname** und **Geburtsdatum**), mit denen die Werte für die Variablen an das Objekt übergeben und auch ausgelesen werden können.

```
using System;

namespace WPFDatenbindung {
    class Kunde {
        private string vorname;
        private string nachname;
        private DateTime geburtsdatum;
        public string Vorname {
            get { return vorname; }
            set { vorname = value; }
        }
        public string Nachname {
            get { return nachname; }
            set { nachname = value; }
        }
        public DateTime Geburtsdatum {
            get { return geburtsdatum; }
            set { geburtsdatum = value; }
        }
    }
}
```

Listing 1: Eine einfache Kunden-Klasse

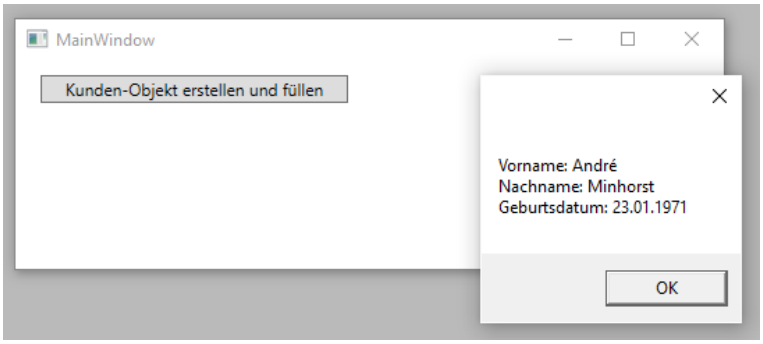


Bild 1: Erstellen eines Objekts und anzeigen seiner Werte

Kunden-Klasse testen

Um diese Klasse wie in Bild 1 zu testen, fügen wir dem Hauptfenster der Anwendung namens **MainWindow** die Schaltfläche **btnObjektErstellen** hinzu.

Für diese legen wir das Attribut **Click** mit dem Wert **btnObjektErstellen_Click** fest und erstellen gleichzeitig die gleichnamige Methode aus Listing 2 (dazu das Attribut **Click** gefolgt von Gleichheitszeichen und Anführungszeichen für das **button**-

```
private void btnObjektErstellen_Click(object sender, RoutedEventArgs e) {
    Kunde kunde = new Kunde();
    kunde.Vorname = "André";
    kunde.Nachname = "Minhorst";
    kunde.Geburtsdatum = new DateTime(1971, 1, 23);
    MessageBox.Show("Vorname: " + kunde.Vorname + "\nNachname: " + kunde.Nachname + "\nGeburtsdatum: "
        + kunde.Geburtsdatum.ToShortDateString());
}
```

Listing 2: Testen der Klasse **Kunde.cs**

Element eingeben und dann einfach mit der Tabulatortaste vervollständigen – die Methode **btnObjektErstellen_Click** finden Sie dann in der Code behind-Klasse namens **MainWindows.xaml.cs**).

Dies erstellt ein neues Objekt auf Basis der Klasse **Kunde** und füllt die drei Eigenschaften mit den entsprechenden Werten. Dabei übergeben wir das Geburtsdatum als neues Objekt des Typs **DateTime** mit dem Jahr, dem Monat und dem Tag als Parameter. Bei der Ausgabe im Meldungsfenster greifen wir über die Eigenschaft **ToShortDateString** auf das reine Datum ohne Angabe der Uhrzeit zu.

Kunden-Objekt im Fenster anzeigen

Nun wollen wir ein WPF-Fenster erstellen und zunächst die Daten eines gefüllten **Kunde**-Objekts dort abbilden. Dazu erstellen Sie ein neues WPF-Fenster namens **EinfachesObjektBinden**, wodurch auch eine Datei **EinfachesObjektBinden.xaml** plus die Code behind-Datei **EinfachesObjektBinden.xaml.cs** erzeugt werden.

Dem Fenster fügen Sie über den XAML-Code ein Grid hinzu, das aus zwei Spalten und vier Zeilen besteht (siehe Listing 3). Die drei **Label**-Steuerelemente weisen Sie über die **Grid.Column**-Eigenschaft der ersten Spalte und über **Grid.Row**

```
<Window x:Class="WPFDatenbindung.EinfachesObjektBinden" ... Title="EinfachesObjektBinden" Height="200" Width="417">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="55*"/>
            <ColumnDefinition Width="91*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="40"/>
            <RowDefinition Height="40"/>
            <RowDefinition Height="40"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Label x:Name="lblVorname" Content="Vorname:" Margin="5,5,5,5" Grid.Column="0" Grid.Row="0"/>
        <Label x:Name="lblNachname" Content="Nachname:" Margin="5,5,5,5" Grid.Column="0" Grid.Row="1"/>
        <Label x:Name="lblGeburtsdatum" Content="Geburtsdatum:" Margin="5,5,5,5" Grid.Column="0" Grid.Row="2"/>
        <TextBox x:Name="txtVorname" Grid.Column="1" Margin="5,5,5,5" Text="TextBox" />
        <TextBox x:Name="txtNachname" Grid.Column="1" Grid.Row="1" Margin="5,5,5,5" Text="TextBox" />
        <TextBox x:Name="txtGeburtsdatum" Grid.Column="1" Grid.Row="2" Margin="5,5,5,5" Text="TextBox" />
    </Grid>
</Window>
```

Listing 3: Definition des WPF-Fensters zur Anzeige der Kundendaten

den jeweiligen Zeilen zu, Gleiches gilt für die drei **TextBox**-Steuerelemente, die allerdings in der zweiten Spalte landen sollen. Um dieses Fenster zu initialisieren und anzuzeigen, fügen Sie dem Hauptfenster der Anwendung eine weitere Schaltfläche namens **btnObjektBinden** hinzu. Diese soll die folgende einfache Methode auslösen und somit das Fenster **EinfachesObjektBinden** anzeigen:

```
private void btnObjektBinden_Click(object sender,
        RoutedEventArgs e) {
    EinfachesObjektBinden wnd = new EinfachesObjektBinden();
    wnd.Show();
}
```

Steuerelemente mit Kundendaten füllen

Nun sollen die Steuerelemente die Daten eines bereits erstellten und mit Eigenschaften gefüllten Objekts auf Basis der Klasse **Kunde** anzeigen. Unter Access/VBA hätten wir dies durch die Zuweisung der entsprechenden Objekteigenschaften zu den Steuerelementen per VBA erledigt. WPF bietet jedoch für seine Fenster und Steuerelemente Eigenschaften an, mit denen Sie die Daten von Objekten per Bindung zuweisen können.

Damit die Textfelder die Daten der drei Eigenschaften Vorname, Nachname und Geburtsdatum eines **Kunde**-Objekts anzeigen, müssen wir dieses zunächst einmal initialisieren und füllen. Dies erledigen wir in der Code behind-Klasse des Fensters **EinfachesObjektBinden**, indem wir ein neues **Kunde**-Objekt mit dem **new**-Schlüsselwort erzeugen und seinen Eigenschaften direkt beim Initialisieren übergeben:

```
namespace WPFDatenbindung {
    public partial class EinfachesObjektBinden : Window {
        Kunde kunde = new Kunde {
            Vorname = "André",
            Nachname = "Minhorst",
            Geburtsdatum = new DateTime(1971, 1, 23)
        };
        public EinfachesObjektBinden() {
            InitializeComponent();
            this.DataContext = kunde;
        }
    }
}
```

Die Konstruktor-Methode **EinfachesObjektBinden**, die beim Initialisieren des Fensters automatisch ausgelöst wird, erweitern Sie wie oben zu erkennen um eine einzige Anweisung:

```
this.DataContext = kunde;
```

Damit stellen Sie die Datenherkunft für das Fenster auf das zuvor erstellte Objekt namens **kunde** ein. Die beiden Methoden werden in der abgebildeten Reihenfolge aufgerufen, die Klasse **Kunde** also bereits vor dem Zuweisen in Form des Objekts **kunde** initialisiert (davon können Sie sich beispielsweise durch das Setzen zweier Haltepunkte beim Testen der Anwendung überzeugen).

Textfelder binden

Nun fehlt noch die Anzeige der Inhalte des Objekts **kunde** in den drei Textfeldern **txtVorname**, **txtNachname** und **txtGeburtsdatum**. Dies erledigen wir, indem wir die Eigenschaft **Text** der drei Steuerelemente wie folgt füllen:

```
<TextBox x:Name="txtVorname" ...
    Text="{Binding Path=Vorname}" />
<TextBox x:Name="txtNachname" ...
    Text="{Binding Path=Nachname}" />
<TextBox x:Name="txtGeburtsdatum" ...
    Text="{Binding Path=Geburtsdatum}" />
```

Wir legen also jeweils eine Bindung zu einer der Eigenschaften des mit **DataContext** festgelegten Objekts an, wobei wir für das **Path**-Attribut den Namen der jeweiligen Objekt-Eigenschaft hinterlegen. Wenn Sie das Fenster nun testen, zeigen die drei Textfelder fast die gewünschten Daten an (siehe Bild 2). Es wäre noch praktisch, wenn auch hier das Datum im kurzen Datumsformat erschiene, also ohne Angabe der Uhrzeit. Dazu ändern Sie noch den Inhalt des **Text**-Attributs wie folgt:

WPF-Datenbindung: Listen-Objekte

Wenn Sie WPF-Anwendungen erstellen wollen, wollen Sie die Daten der Anwendung in entsprechenden Fenstern darstellen. Ein Weg zu diesem Ziel ist das Bereitstellen der Daten in Listenform, also etwa als Collection oder Dictionary. Dieser Artikel zeigt, welche Schritte nötig sind, um eine Liste von Objekten auf Basis einer einfachen Klasse zu erstellen und seine Daten in geeigneten Listen-Steuerelementen eines WPF-Fensters anzuzeigen.

Im Beitrag **WPF-Datenbindung: Einfache Objekte** haben wir uns angesehen, wie Sie ein einfaches Objekt wie einen Kunden mit Eigenschaften wie **Vorname**, **Nachname** und **Geburtsdatum** in den Textfeldern eines WPF-Fensters anzeigen. Außerdem haben Sie dort erfahren, wie Sie die Daten in beiden Richtungen synchron halten können.

Nun wollen wir ein Listenobjekt wie beispielsweise eine **Collection** mit mehreren **Kunde**-Objekten füllen und diese samt ausgewählter Eigenschaften in verschiedenen Listen-Steuerelementen von WPF anzeigen – zum Beispiel im **DataGrid**, in der **ComboBox**, in der **ListBox** oder in der **ListView**. Welche Steuerelemente für die Darstellung einer Auflistung infrage kommen, hängt übrigens allein davon ab, ob diese von der Klasse **ItemsControl** abgeleitet sind. Was die Listen-Objekte angeht, deren Inhalt wir im Listen-Steuerelement abbilden wollen, so müssen diese lediglich die **IEnumerable**-Schnittstelle implementieren.

Ob ein Steuerelement die Klasse **ItemsControl** als Basistyp verwendet, können Sie beispielsweise im Objektkatalog nachlesen. Suchen Sie dort einfach nach dem gewünschten Steuerelement und schauen sich im linken Bereich die Basistypen des Steuerelements an. In Bild 1 finden Sie beispielsweise **ItemsControl** als Basistyp für das **ComboBox**-Steuerelement vor.

Der Objektkatalog hilft auch dabei, die als Datenquelle für ein Listen-Steuerelement infrage kommenden Klassen zu identifizieren. In Bild 2 etwa erkennen wir, dass das **Collection**-Objekt die Schnittstelle **IEnumerable** implementiert und somit als Datenquelle taugt.

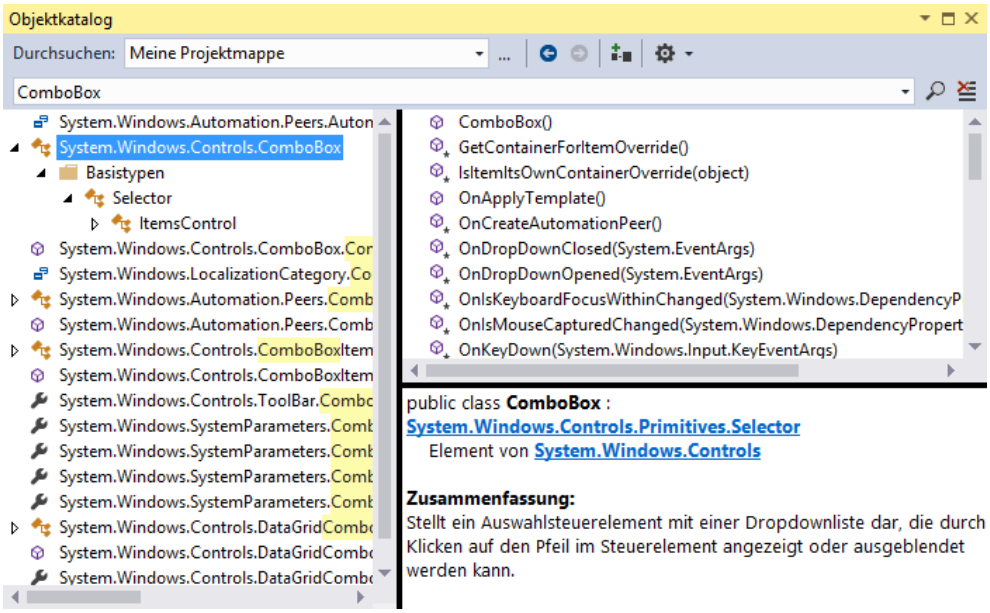


Bild 1: Prüfen, ob ein Steuerelement zur Darstellung von Listen geeignet ist

Liste mit Kunden füllen

Schauen wir uns zunächst an, wie wir ein Listen-Objekt mit einigen verschiedenen **Kunde**-Objekten füllen. Da wir in diesem Fall wissen, dass wir immer den gleichen Objekttyp zum Listen-Objekt hinzufügen möchten, verwenden wir nicht die üblicherweise verwendeten Listen-Typen wie **Collection**

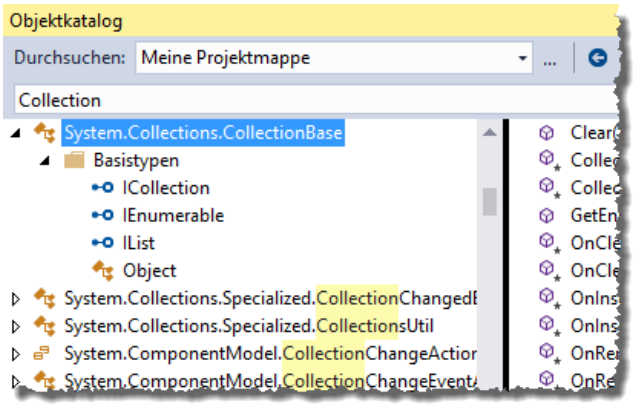


Bild 2: Untersuchen einer Klasse auf Listen-Fähigkeit

oder **ArrayList**. Diese haben nämlich den Nachteil, dass sie beliebige Objekte aufnehmen können und dadurch beim Zugriff immer eine Typkonvertierung nötig ist.

Im Gegensatz dazu gibt es die typsicheren generischen Auflistungsklassen, bei denen man direkt bei der Deklaration angibt, welchen Typ die aufzunehmenden Elemente haben. Dieser wird dabei in spitzen Klammern hinter dem Schlüsselwort für die zu verwendende Auflistungsklasse angegeben. In unserem Beispiel wollen wir eine Methode namens **GetKunden** verwenden, um eine generische Auflistungsklasse des Typs **List** mit Elementen des Typs **Kunde** zu füllen und zurückzugeben.

Um die generische Klasse **List** zu verwenden, benötigen wir die Klasse **System.Collections.Generic**, die wir mit folgender Anweisung bereitstellen:

```
using System.Collections.Generic;
```

Die Methode **GetKunden** finden Sie in Listing 1. Nach dem Deklarieren und Initialisieren des **List**-Objekts **kunden** fügt die Methode mit der **Add**-Methode nacheinander drei neue Objekte des Typs **Kunde** hinzu. Die Definition der Klasse **Kunde** entnehmen Sie dem Beispielprojekt (siehe **Kunde.cs**), eine Beschreibung dieser Klasse liefert der Artikel **WPF-Datenbindung: Einfache Objekte**.

Die Methode **GetKunden** testen wir mit einer kleinen Methode, die durch die Schaltfläche **btnListenelementeAusgeben** unseres Beispielfensters **ListeBinden.xaml** ausgelöst wird (Code siehe Listing 2). Die Methode durchläuft die **Kunde**-Objekte der mit **GetKunden** gelieferten generischen Liste in einer **foreach**-Schleife und stellt damit eine Zeichenkette zusammen, welche Vorname, Nachname und Geburtsdatum der Kunden enthält.

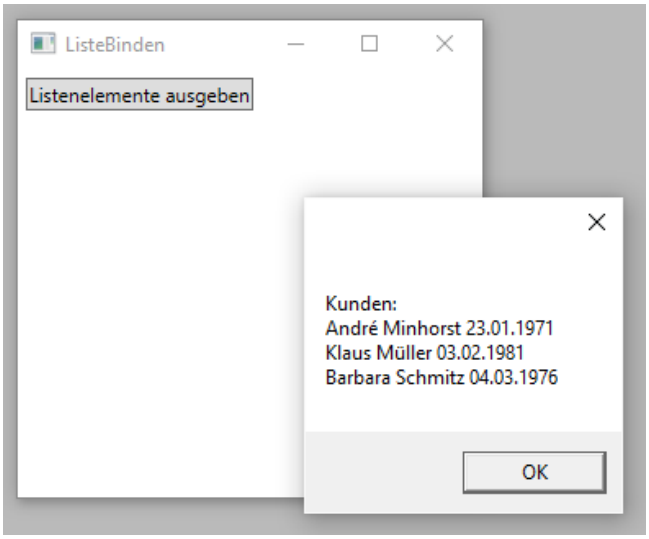


Bild 3: Ausgabe der Daten einer Klasse

```
private List<Kunde> GetKunden() {
    List<Kunde> kunden = new List<Kunde>();
    kunden.Add(new Kunde { Vorname = "André", Nachname = "Minhorst", Geburtsdatum = new DateTime(1971, 1, 23) });
    kunden.Add(new Kunde { Vorname = "Klaus", Nachname = "Müller", Geburtsdatum = new DateTime(1981, 2, 3) });
    kunden.Add(new Kunde { Vorname = "Barbara", Nachname = "Schmitz", Geburtsdatum = new DateTime(1976, 3, 4) });
    return kunden;
}
```

Listing 1: Füllen einer generischen Liste mit **Kunde**-Objekten

```
private void btnListenelementeAusgeben_Click(object sender, RoutedEventArgs e) {
    List<Kunde> kunden = GetKunden();
    string strKunden = "";
    foreach(Kunde kunde in kunden) {
        strKunden += "\n" + kunde.Vorname + " " + kunde.Nachname + " " + kunde.Geburtsdatum.ToShortDateString();
    }
    MessageBox.Show("Kunden:" + strKunden);
}
```

Listing 2: Testen der Methode **GetKunden**

Die Zeichenkette gibt die Methode dann per **MessageBox** auf der Benutzeroberfläche aus (siehe Bild 3).

Liste an ListBox binden

Wenn wir einmal eine Auflistung mit den gewünschten Objekten gefüllt haben, ist die Anzeige in einem Listen-Steuerelement nicht mehr besonders kompliziert. Wir wollen es zunächst mit einem **ListBox**-Steuerelement probieren.

Dazu fügen Sie zum Grid des Fensters wie in Listing 3 ein **Label**- und ein **ListBox**-Element hinzu.

Für das **ListBox**-Steuerelement legen wir vorerst keine weiteren Eigenschaften fest. Das Füllen erledigen wir mit zwei Anweisungen, die beim Öffnen des Fensters ausgeführt werden und daher in der Konstruktor-Methode der Klasse **ListeBinden.xaml.cs** landen.

Die erste Anweisung stellt die Eigenschaft **ItemsSource** auf die Methode **GetKunden** ein, die zweite erklärt das Feld **Nachname** zu dem Feld, das im Listenfeld angezeigt werden soll:

```
public ListeBinden() {
    InitializeComponent();
    lstKunden.ItemsSource = GetKunden();
    lstKunden.DisplayMemberPath = "Nachname";
}
```

Das Ergebnis sieht schon einmal den Erwartungen entsprechend aus, wie Bild 4 zeigt.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="60" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="30" />
        <RowDefinition Height="130" />
        <RowDefinition Height="30" />
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2">
        <Button Content="Listenelemente ausgeben" Name="btnListenelementeAusgeben" Margin="5,5,5,5"
            Click="btnListenelementeAusgeben_Click"/>
    </StackPanel>
    <Label Content="Kunden:" Grid.Row="1" Grid.Column="0" />
    <ListBox x:Name="lstKunden" Margin="5,5,5,5" VerticalAlignment="Top" Grid.Row="1" Grid.Column="1" />
</Grid>
```

Listing 3: Definition des Grids mit dem **ListBox**-Steuerelement

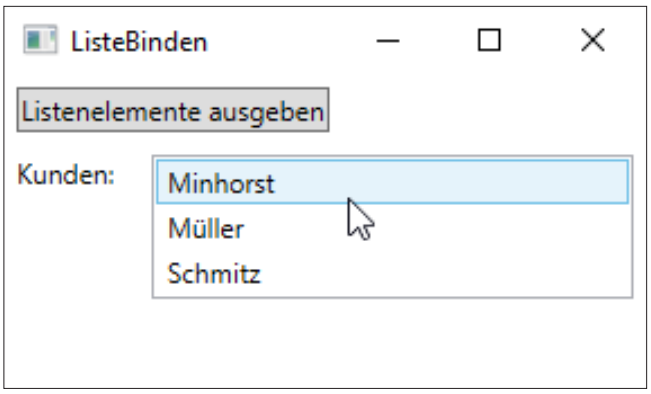


Bild 4: Anzeigen von Kunden in einer ListBox

Gewähltes Objekt anzeigen

Zunächst wollen wir uns ansehen, wie wir die Inhalte des aktuell gewählten Eintrags des **ListBox**-Steuerelements im Fenster anzeigen können. Dazu fügen wir zunächst drei Textfelder zu jeweils einer neuen Zeile im Grid hinzu.

Diese Textfelder binden wir gleich über einen speziellen Wert für das Attribut **Text** an ein Feld des aktuell in der **ListBox** ausgewählten Objekts, für das Textfeld **txtVorname** beispielsweise wie folgt:

```
<TextBox x:Name="txtVorname"
    Text="{Binding Path=Vorname}" ... />
```

Das allein hilft allerdings noch nicht, da das aktuell ausgewählte Objekt im **ListBox**-Steuerelement noch als **DataContext** angegeben werden muss. Wenn Sie nur ein einziges Textfeld an das Objekt im **ListBox**-Steuerelement binden wollten, könnten Sie das Attribut **DataContext** gleich dem entsprechenden **TextBox**-Steuerelement hinzufügen:

```
<TextBox DataContext="{Binding ElementName=lstKunden,
    Path=SelectedItem}" x:Name="txtVorname"
    Text="{Binding Path=Vorname}" ... />
```

Binding gibt an, dass eine Bindungsdefinition folgt. **ElementName** gibt das Element an, welches als Quelle dient, und **Path** die Eigenschaft, welche das Quellobjekt liefert. Hier binden wir das Attribut **Text** des **TextBox**-Steuerelements also an das Steuerelement **lstKunden**, genau genommen an die Eigenschaft **SelectedItem**. Dies liefert eine Referenz auf das aktuell markierte Element zurück, genau genommen auf ein Element des Typs **Kunde**. Auf diese Weise zeigt das Textfeld **Vorname** bereits den Vornamen des aktuell markierten Kunden im Listenfeld an.

Allerdings wollen wir ja gleich drei Textfelder mit den Werten verschiedener Eigenschaften des Objekts **Kunde** aus dem **ListBox**-Steuerelement füllen. Wir müssen aber nun nicht alle drei Textfelder mit der gleichen **DataContext**-Eigenschaft versehen, sondern können diese auch in einem übergeordneten Element festlegen – in diesem Fall etwa im **Grid**-Element:

```
<Grid DataContext="{Binding ElementName=lstKunden,
Path=SelectedItem}">
```

```
<Grid DataContext="{Binding ElementName=lstKunden, Path=SelectedItem}">
    ...
    <Label Content="Kunden:" Grid.Row="1" Grid.Column="0" />
    <ListBox x:Name="lstKunden" Margin="5,5,5,5" VerticalAlignment="Top" Grid.Row="1" Grid.Column="1" />
    <Label Content="Vorname:" Grid.Row="2" Grid.Column="0" />
    <TextBox x:Name="txtVorname" Text="{Binding Path=Vorname}" Grid.Row="2" Grid.Column="1" Margin="5,5,5,5" />
    <Label Content="Nachname:" Grid.Row="3" Grid.Column="0" />
    <TextBox x:Name="txtNachname" Text="{Binding Path=Nachname}" Grid.Row="3" Grid.Column="1" Margin="5,5,5,5"/>
    <Label Content="Geburtsdatum:" Grid.Row="4" Grid.Column="0" />
    <TextBox x:Name="txtGeburtsdatum" Text="{Binding Path=Geburtsdatum}" Grid.Row="4" Grid.Column="1" Margin="5,5,5,5"/>
</Grid>
```

Listing 4: Einstellungen für Textfelder und Grid für die Datenbindung an das aktuelle Objekt im Listenfeld

Einführung in das Entity Framework

In vorherigen Beiträgen haben wir gezeigt, wie Sie Fenster und Steuerelemente wie das DataGrid mit den Daten aus einem ADO.NET-DataSet füllen können. Wenn Sie echte mehrschichtige Anwendungen bauen möchten, gehen Sie einen Schritt weiter und nutzen einen objektrelationalen Mapper, um die Daten aus den Tabellen in Objekten abzulegen, bevor Sie diese als Datenquelle für die Benutzeroberfläche nutzen. Der Clou eines solchen Mappers ist, dass es sich um ein Framework handelt - das heißt, dass Sie nur einen geringen Teil des notwendigen Codes selbst schreiben müssen. Dieser Artikel gibt eine kleine Einführung in das Entity Framework.

Datenzugriff gestern und heute

Unter Access waren Sie es gewohnt, direkt auf die Tabellen und Abfragen der Datenbank zuzugreifen und diese beispielsweise als Datenherkunft eines Formulars oder als Datensatzherkunft von Kombinations- und Listenfeldern anzugeben. Unter VBA konnten Sie leicht mit den Methoden des **Database**-Objekts der DAO-Bibliothek wie **OpenRecordset**, **Execute** et cetera per Code auf die Daten zugreifen.

In einigen Artikeln des vorliegenden Magazins haben wir gezeigt, wie Sie per ADO.NET auf die Daten einer Datenbank zugreifen und wie Sie die dort verwendeten DataSet- und DataTable-Objekte nutzen, um die enthaltenen Daten in Fenstern und ihren Steuerelementen anzuzeigen.

Damit haben wir allerdings immer noch eine recht direkte Bindung zu den Daten der Datenbank hergestellt. Das Ziel moderner Programmiersprachen ist es jedoch, die Daten in Form von Objekten zur Verfügung zu stellen, um diese in Benutzeroberflächen etwa auf Basis von WPF (Windows Presentation Foundation) anzuzeigen und bearbeiten zu können.

Mit der Kenntnis der ADO.NET-Technologie und der Grundlagen der Objektorientierung kann man dann eine Menge Code produzieren, die dafür sorgt, dass der Inhalt etwa eines Datensatzes einer Tabelle namens **tblKunden** in ein Objekt auf Basis einer Klasse namens **clsKunde** landet. Diese stellt dann über Eigenschaften wie **Vorname**, **Nachname**, **Strasse** und so weiter alle Informationen zur Verfügung, die im entsprechenden Datensatz der Tabelle **tblKunden** gespeichert sind. Die Eigenschaften dieses Objekts können Sie nun den Steuerelementen eines Fensters zuweisen, damit der Benutzer diese ansehen und bearbeiten kann. Hat der Benutzer Änderungen an diesem Datensatz durchgeführt, sollen diese natürlich in die zugrunde liegende Datenbank übertragen werden. Dazu müssen Sie den Klassen, auf denen die Objekte basieren, entsprechende Methoden hinzufügen, die sich um das Speichern der Änderungen in die Tabellen der Datenbank kümmern.

Der benötigte Code ist kein Hexenwerk, allerdings wird es schnell recht umfangreich. Je mehr Tabellen die Datenbank enthält, desto mehr Klassen sind zu definieren, und da es meist auch noch Beziehungen zwischen den einzelnen Tabellen gibt, sind diese natürlich auch entsprechend abzubilden.

Damit Sie diesen Aufwand nicht stemmen müssen, gibt es objektrelationale Mapper wie beispielsweise das Entity Framework. Damit nimmt Ihnen Microsoft zwei wichtige Aufgaben ab: Erstens erlaubt es, auf Basis einer vorhandenen Datenbank die benötigten Klassen zu erstellen, zweitens stellt es auch noch den Code zur Verfügung, mit dem die Daten von den Tabellen in die Objekte und umgekehrt übertragen werden.

Schichtweise

Im Entity Framework spricht man von verschiedenen Schichten:

- **Konzeptionelle Schicht**: Diese Schicht entspricht den einzelnen Geschäftsobjekten der Anwendung, die in Form etwa von einzelnen Klassen/Entitäten und ihren Beziehungen untereinander erscheint. Entitäten und Beziehungen werden in der sogenannten **Conceptual Schema Definition Language** (**CSDL**) beschrieben.

- **Logische Schicht**: Die logische Schicht entspricht dem Datenmodell der Datenbank. Allerdings gibt es dafür eine eigene Sprache namens **Store Schema Definition Language** (**SSDL**). Die SSDL-Dateien enthalten die Beschreibung der Tabellen, Felder, Schlüssel, Abfragen, gespeicherten Prozeduren et cetera, allerdings in einem vom Datenbanksystem unabhängigen Format.

- **Zuordnungsschicht**: Damit fehlt noch das Bindeglied zwischen den beiden, nämlich die Zuordnungsschicht. Diese wird in der Sprache **Mapping Schema Language** (**MSL**) verfasst. Diese Schicht legt die Zuordnung zwischen der konzeptionellen Schicht und der logischen Schicht fest, und zwar wieder im XML-Format.

Von A nach B und wieder zurück

Genau genommen bietet das Entity Framework mehrere Möglichkeiten, um die Daten zu den drei Schichten zu erstellen.

Dabei kommt es darauf an, welche Voraussetzungen vorliegen:

- **Database First**: Es liegt eine Datenbank vor, auf deren Basis die drei Schichten erzeugt werden sollen. Dann besteht eine weitere Voraussetzung darin, dass es einen entsprechenden Datenprovider gibt. Dies ist beispielsweise für SQL Server-Datenbanken der Fall, für Access-Datenbanken nicht.

- **Model First**: Hier verwenden Sie den Entity Data Model-Designer von Visual Studio, um die Entitäten und ihre Beziehungen untereinander zu definieren. Anschließend erzeugen Sie auf dieser Basis die Datenbank beziehungsweise ein entsprechendes Skript.

- **Code First**: Hier erstellen Sie einfach die Klassen für die einzelnen Entitäten und legen dort die Eigenschaften und Beziehungen fest. Außerdem benötigen Sie eine weitere Klasse, mit der Sie ein paar zusätzliche Informationen angeben. Auf dieser Basis erzeugt Visual Studio dann die verschiedenen Schichten des Entity Data Models und erstellt sogar noch die Datenbank.

Für uns ist zunächst die erste Variante interessant, bei der wir das Entity Data Model auf Basis einer bestehenden Datenbank erstellen. Dazu schauen wir uns auf den folgenden Seiten ein einführendes Beispiel an.

Voraussetzungen

Voraussetzung für die hier verwendete Vorgehensweise ist Visual Studio 2015 mit Update 2. Ohne Update 2 kam es bei unseren Experimenten teilweise zu Abstürzen beim Hinzufügen einer Datenquelle auf Basis eines Objekts.

Erstellen der Anwendung

Für das Beispiel verwenden wir eine Anwendung auf Basis der Vorlage **Visual C#\WPF-Anwendung**, die Sie mit dem Dialog erstellen können, der beim Anklicken des Menüpunkts **Datei\Neu** erscheint. Unser Projekt soll **EntityBeispiel** heißen.

Entity Framework-Unterstützung hinzufügen

Das Entity Framework besteht aus einer Reihe von zusätzlichen Elementen, die Sie Ihrer Anwendung hinzufügen müssen. Um dies zu erledigen, klicken Sie mit der rechten Maustaste auf den Namen des Projekts im Projektmappen-Explorer und wählen dort den Eintrag **NuGet-Pakete verwalten...** aus. Es erscheint der **NuGet-Paket-Manager**, in dem Sie zunächst zum Bereich **Durchsuchen** wechseln. Dort sollte gleich als erster Eintrag das **EntityFramework** auftauchen, das Sie auswählen und mit einem Mausklick auf die Schaltfläche **Installieren** auf der rechten Seite installieren (siehe Bild 1).

Es erscheint ein weiterer Dialog, der den folgenden Vorgang einleitet und mit **OK** bestätigt werden kann (siehe Bild 2).

Nachdem Sie auch noch die Lizenzbedingungen bestätigt haben, installiert Visual Studio das Paket im aktuellen Projekt.

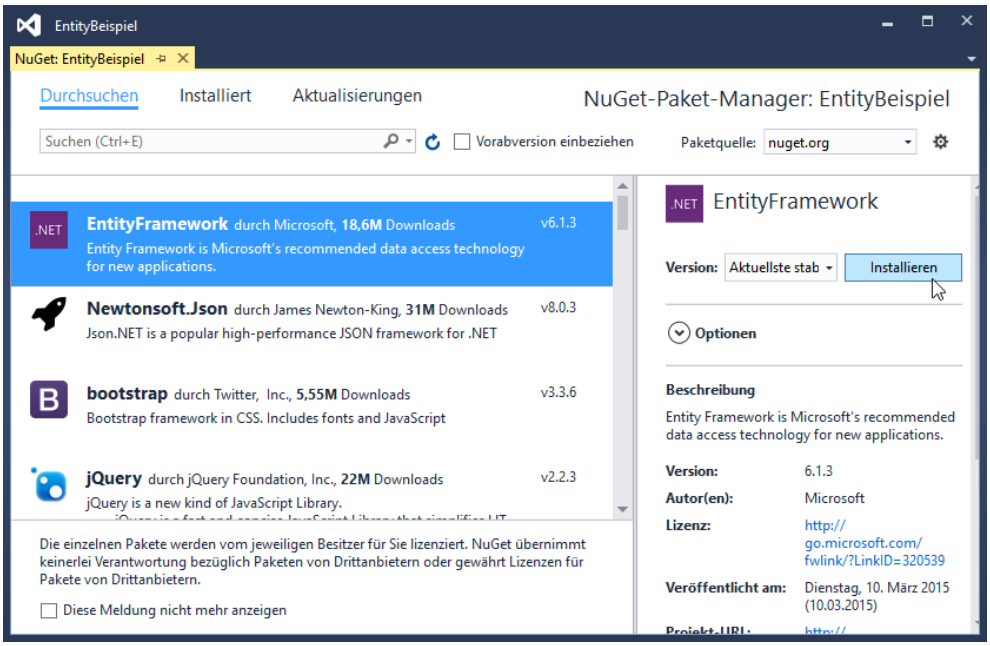


Bild 1: Hinzufügen des Entity Frameworks

Datenbank erstellen

Für den Fall, dass Sie aktuell keine SQL Server-Datenbank parat haben, die Sie als Quelle für das Beispiel nutzen können, erstellen wir schnell eine – unabhängig davon, ob Sie überhaupt SQL Server installiert haben oder nicht.

Visual Studio kommt seit der Version 2012 mit einer kleinen, für Entwickler-Zwecke geeigneten und weitgehend mit dem SQL Server kompatiblen Datenbank-Engine namens **LocalDB**. Um diese zu nutzen, aktivieren Sie mit **Ansicht|Server-Explorer** den Server-Explorer, klicken mit der rechten Maustaste auf den Eintrag **Datenverbindungen** und wählen den Kontextmenü-Eintrag **Verbindung hinzufügen...** aus (siehe Bild 3).

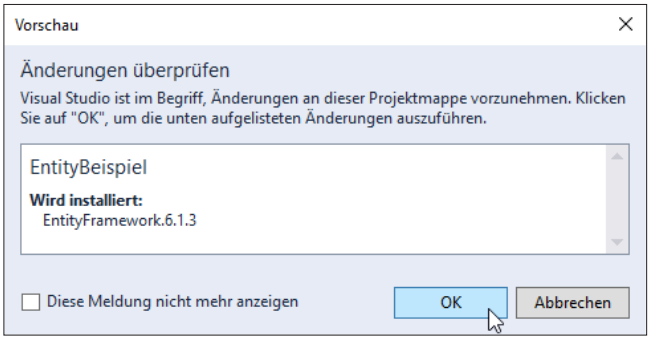


Bild 2: Installation des Entity Frameworks

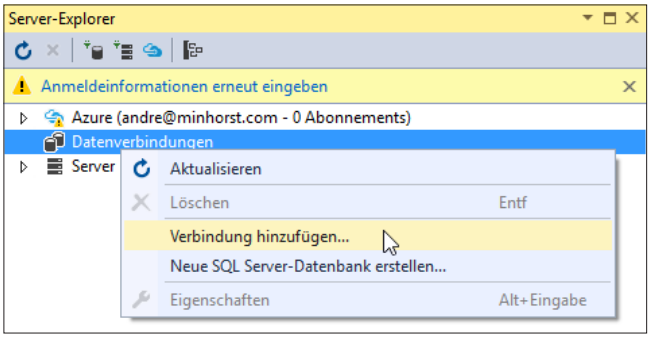


Bild 3: Hinzufügen einer neuen Verbindung

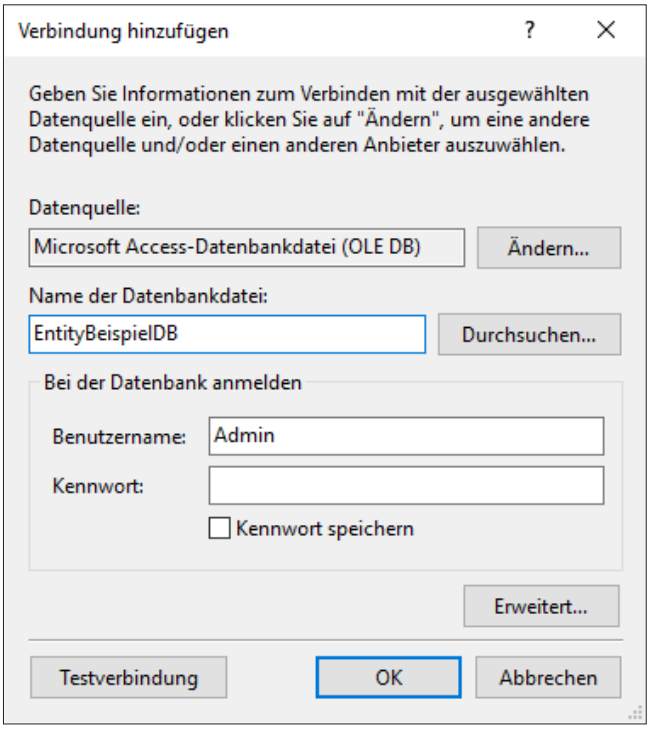


Bild 4: Hinzufügen einer neuen Datenbankverbindung

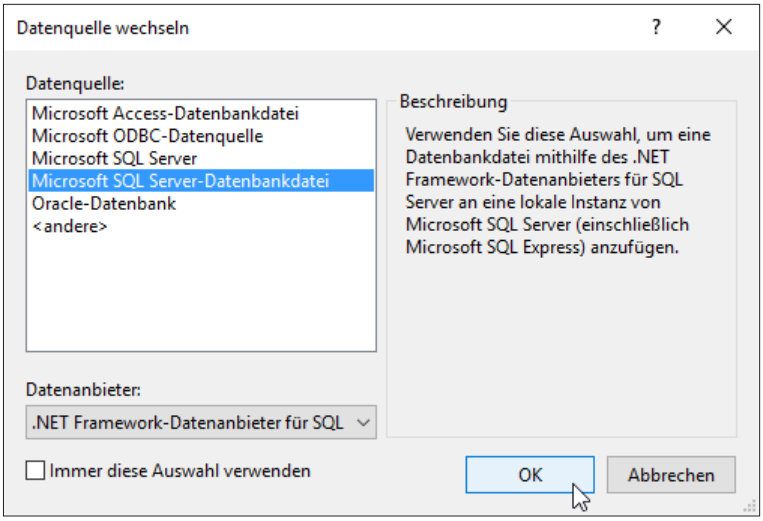


Bild 5: Auswahl des Datenquellen-Typs

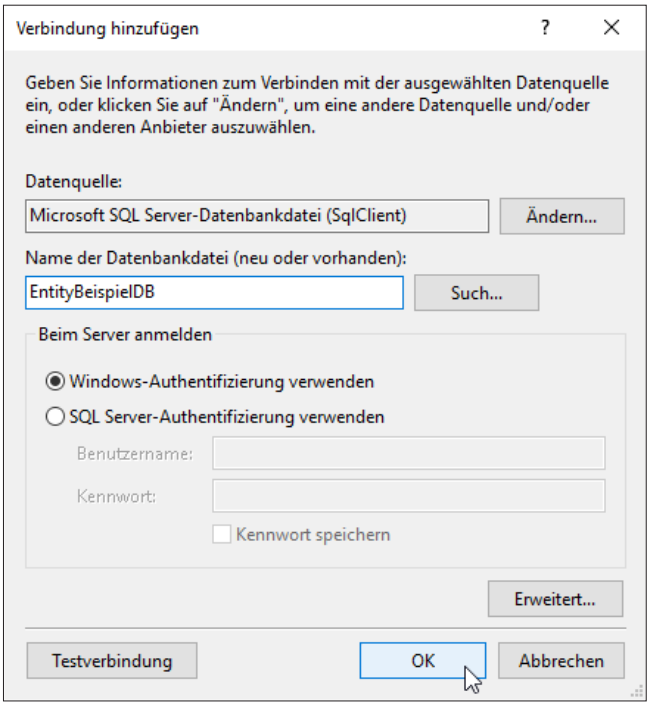


Bild 6: Angabe des Namens der neuen Datenbank

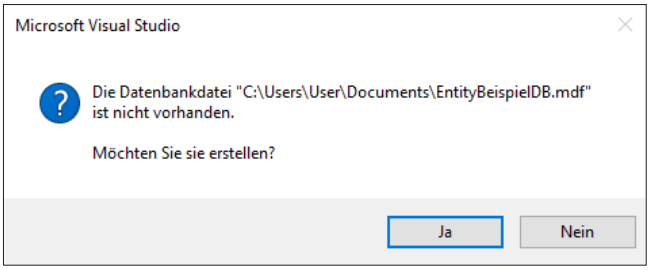


Bild 7: Visual Studio fragt, ob Sie eine neue Datenbankdatei anlegen möchten.

Im nun erscheinenden Dialog **Verbindung hinzufügen** (siehe Bild 4) klicken Sie zunächst auf die Schaltfläche **Ändern**.

Der Dialog **Datenquelle wechseln** bietet alle verfügbaren Datenquellen an, wobei Sie hier **Microsoft SQL Server Datenbankdatei** selektieren und die Auswahl mit einem Klick auf die Schaltfläche **OK** bestätigen (siehe Bild 5).

Damit kehren Sie zurück zum nun etwas anders gestalteten Dialog **Verbindung hinzufügen** (siehe Bild 6). Hier geben Sie nun den Namen der zu erstellenden Datenbankdatei ein, zum Beispiel **EntityBeispielDB**. Klicken Sie dann auf die Schaltfläche **OK**.

Es erscheint noch eine Rückfrage, ob Sie die noch nicht vorhandene Datenbankdatei anlegen möchten. Bestätigen Sie diese mit **Ja** (siehe Bild 7).

Damit erscheint die neue Datenbank nun im Server-Explorer im Bereich **Datenverbindungen** (siehe Bild 8). Wie Sie sehen, gibt es allerdings noch keine Tabellen. Aber keine Sorge – das ändern wir gleich im nächsten Schritt.

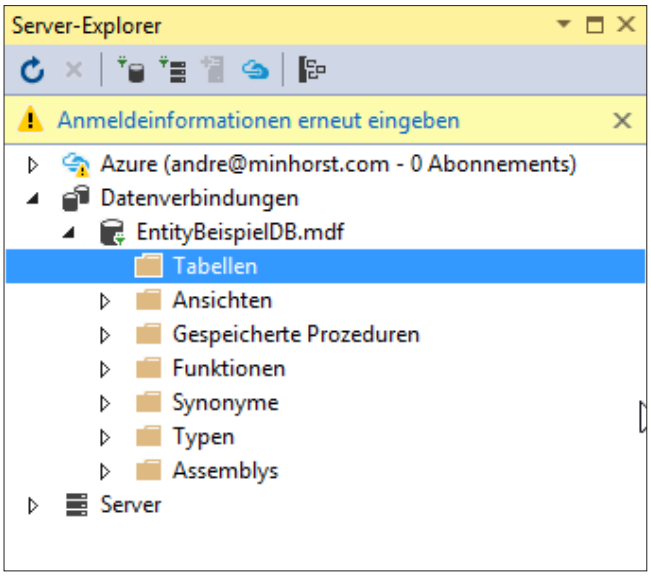


Bild 8: Die neue von **localDB** verwaltete Datenbank, hier noch ohne Tabellen

Tabellen zur Datenbank hinzufügen

Um Tabellen zur Datenbank hinzuzufügen, klicken Sie mit der rechten Maustaste auf den Eintrag **Tabellen** im Server-Explorer. Wählen Sie hier den Eintrag **Neue Abfrage** aus, um ein Fenster zur Eingabe von SQL-Code zu öffnen. Hier tragen Sie zunächst den Code aus Listing 1 ein.

```
CREATE TABLE [dbo].[tblKategorien] (
    [KategorieID] [int] NOT NULL IDENTITY,
    [Kategoriename] [nvarchar](max),
    CONSTRAINT [PK_dbo.tblKategorien] PRIMARY KEY ([KategorieID])
)
CREATE TABLE [dbo].[tblArtikel] (
    [ArtikelID] [int] NOT NULL IDENTITY,
    [Artikelname] [nvarchar](max),
    [KategorieID] [int] NOT NULL,
    CONSTRAINT [PK_dbo.tblArtikel] PRIMARY KEY ([ArtikelID]) )
CREATE INDEX [IX_KategorieID] ON [dbo].[tblArtikel]([KategorieID])
ALTER TABLE [dbo].[tblArtikel]
    ADD CONSTRAINT [FK_dbo.tblArtikel_dbo.tblKategorien_KategorieID]
    FOREIGN KEY ([KategorieID])
    REFERENCES [dbo].[tblKategorien] ([KategorieID]) ON DELETE CASCADE
```

Listing 1: Skript zum Erstellen zweier Tabellen für die Datenbank

Danach klicken Sie oben links auf das Menü mit dem Play-Symbol. Wählen Sie den Eintrag **Ausführen** aus, werden alle aktuell im Fenster angezeigten SQL-Anweisungen ausgeführt (siehe Bild 9). Alternativ können Sie sich auch die Tastenkombination **Strg + Umschalt + E** für diesen Befehl merken. Wer gelegentlich mit dem **SQL Server Management Studio** arbeitet, hat sich vielleicht an die Taste **F5** für das Starten von SQL-Code gewöhnt. Diese Taste ist in Visual Studio aber leider schon für das Starten eines Projekts zum Debuggen reserviert.

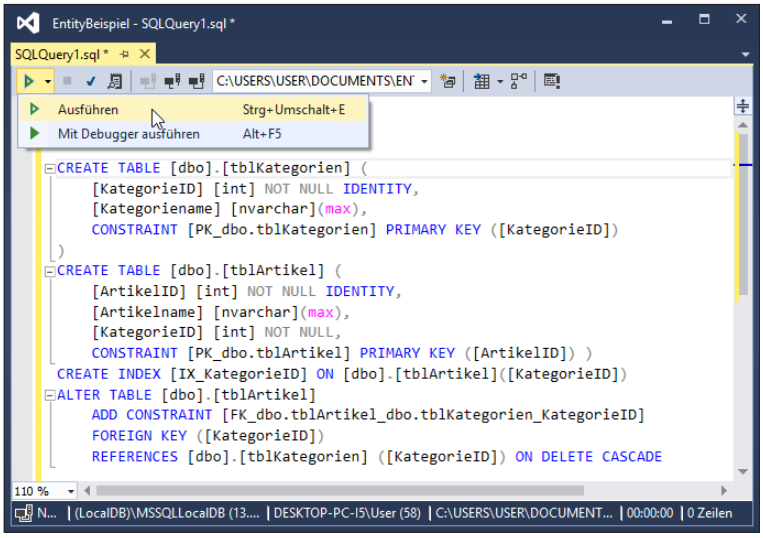


Bild 9: Ausführen des eingegebenen SQL-Codes

Die SQL-Anweisungen sollten zwei Tabellen zur Datenbank hinzufügen – eine namens **tblKategorien** mit den Feldern **KategorieID** (als Primärschlüsselfeld) und **Kategoriename** und eine namens **tblArtikel** mit den Feldern **ArtikelID** (ebenfalls als Primärschlüsselfeld), **Artikelname** und **KategorieID**. Letztere dient als Fremdschlüsselfeld zur Verknüpfung mit den Datensätzen der Tabelle **tblKategorien**.

Die erste **CREATE TABLE**-Anweisung erstellt die Tabelle **tblKategorien**, die zweite die Tabelle **tblArtikel**. Die dritte Anweisung erstellt einen Index für das Feld **KategorieID** der Tabelle **tblArtikel**, die vierte fügt die Definition des

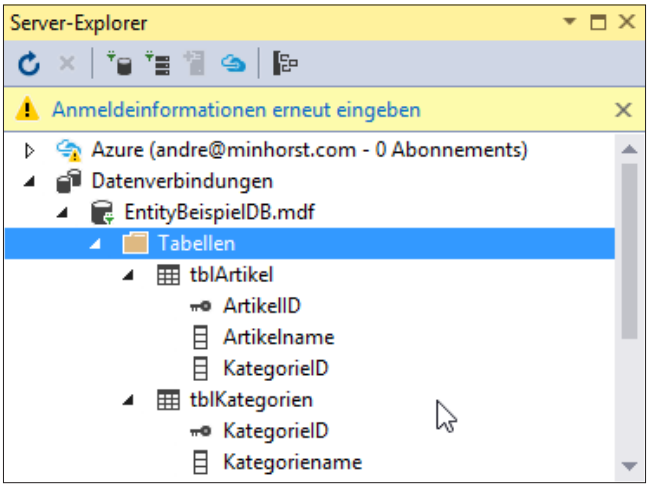


Bild 10: Der Server Explorer mit den beiden neuen Tabellen

Feldes **KategorieID** der Tabelle **tblArtikel** als Fremdschlüsselfeld zur Verknüpfung mit dem Primärschlüsselfeld der Tabelle **tblKategorien** ein. Die dadurch entstehende **FOREIGN KEY**-Beziehung wird mit **ON DELETE CASCADE** als »mit Löschweitergabe« definiert.

Die beiden Tabellen erscheinen nun wie in Bild 10 im Server Explorer. Nun fehlen noch ein paar Daten als Testmaterial. Diese können Sie ebenfalls per SQL-Anweisung hinzufügen – beispielsweise so:

```
INSERT INTO tblKategorien(Kategoriename)
VALUES('Getränke');
INSERT INTO tblKategorien(Kategoriename)
VALUES('Fastfood');
INSERT INTO tblArtikel(Artikelname, KategorieID)
VALUES('Coca Cola', 1);
INSERT INTO tblArtikel(Artikelname, KategorieID)
VALUES('König Pilsener', 1);
```

Wenn Sie diese Anweisungen unter die vorhandenen Anweisungen im SQL-Editor schreiben und nur diese Anweisungen ausführen wollen, markieren Sie diese und betätigen dann die Tastenkombination **Strg + Umschalt + E**. Alternativ können Sie die Daten auch direkt über die Benutzeroberfläche eingeben. Dazu klicken Sie mit der rechten Maustaste auf die jeweilige Tabelle und wählen aus dem Kontextmenü den Eintrag **Tabellendaten anzeigen** aus. Hier können Sie dann wie in Bild 11 eigene Datensätze eingeben.

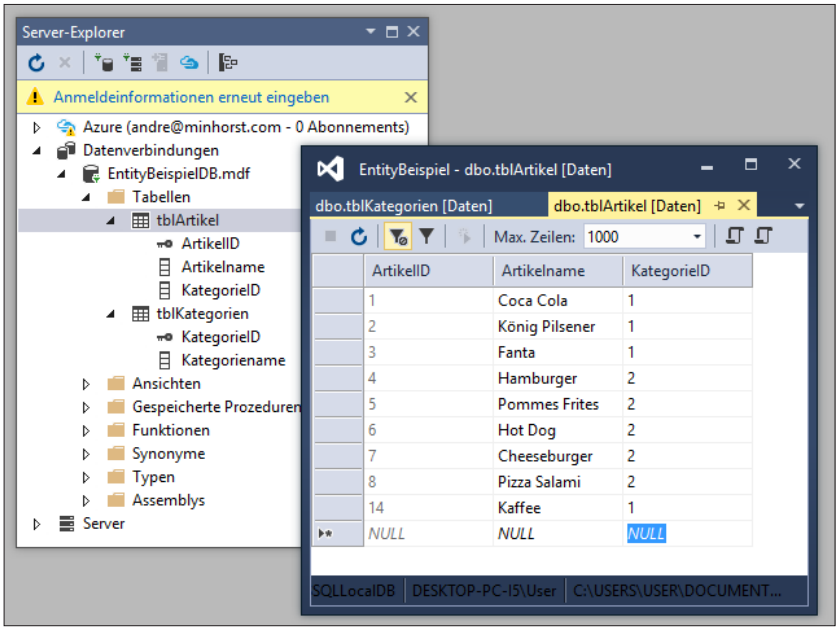


Bild 11: Eingabe von Daten in die Tabelle **tblArtikel**

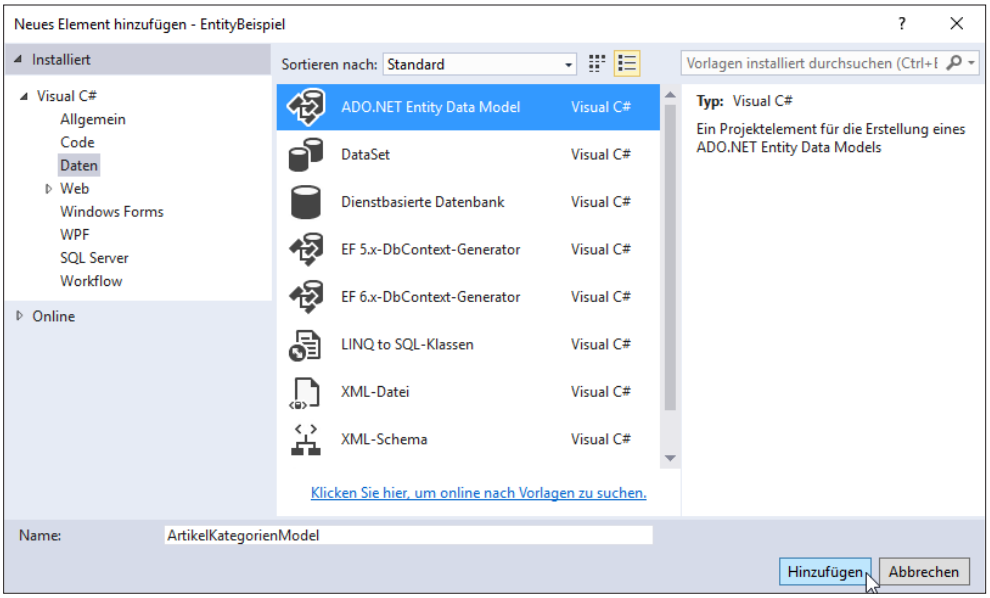


Bild 12: Hinzufügen eines Entity Data Models

Auftritt Entity Framework

Nun wird es interessant: Wir wollen mithilfe des Entity Frameworks ein **Entity Data Model** (**EDM**) erstellen. Dazu fügen wir ein neues Element zum Projekt hinzu, und zwar über den Eintrag **Hinzufügen|Neues Element** des Kontextmenüs des Projektnamens im Projektmappen-Explorer. Dies öffnet den Dialog **Element hinzufügen**, wo Sie links auf **Visual C#|Daten** klicken,

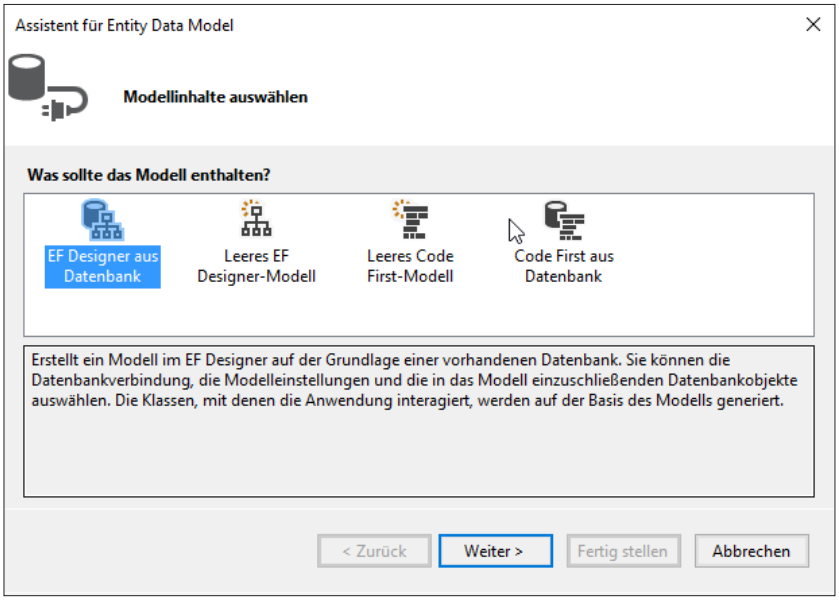


Bild 13: Festlegen des Modell-Typs

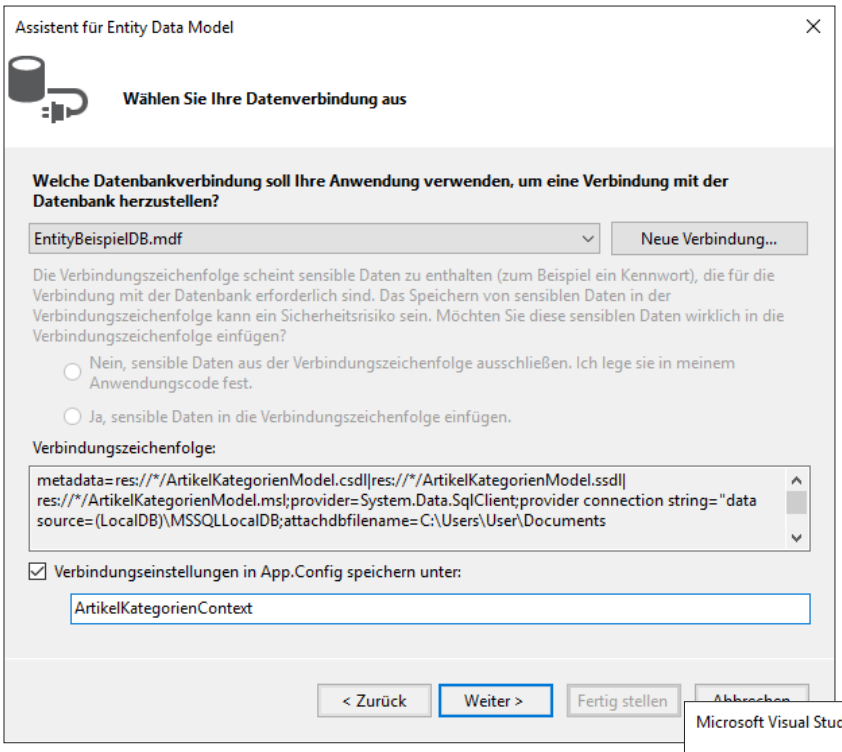


Bild 14: Angabe der Datenbankverbindung sowie des Namens für die Verbindungseigenschaften

dann in der Mitte den Eintrag **ADO.NET Entity Data Model** auswählen, den Namen **ArtikelKategorienModel** angeben und auf **Hinzufügen** klicken (siehe Bild 12).

Nun erscheint ein weiterer Dialog namens **Assistent für Entity Data Model**, wo Sie die Auswahl zwischen vier Optionen haben. Wählen Sie die erste mit der Bezeichnung **EF Designer aus Datenbank** (siehe Bild 13).

Der nächste Schritt im Assistenten erfordert die Auswahl der zu verwendenden Verbindung. Da wir diese zuvor bereits auf Basis der neu angelegten Datenbank erstellt haben, wird die Verbindung **EntityBeispielDB.mdf** voreingestellt – diesen Wert können wir beibehalten. Außerdem geben wir als Name für die Verbindungseinstellungen in der Anwendungskonfiguration **ArtikelKategorienKontext** an (siehe Bild 14).

Im nächsten Dialog können Sie angeben, dass die Datenbankdatei mit dem Projekt gespeichert und die Verbindungseigenschaft entsprechend geändert werden soll (siehe Bild 15).

Nun müssen Sie noch die Tabellen beziehungsweise Datenbankobjekte auswählen, die bei der Erstellung des Modells berücksichtigt werden sollen. In diesem Fall enthält die Datenbank ja nur zwei Tabellen, die wir beide markieren (siehe Bild 16). Im gleichen Dialog legen Sie als Name für das Modell den Wert **ArtikelKategorienModel** fest. Außerdem ist hier die Option

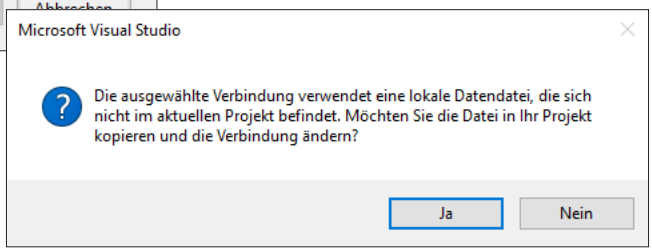


Bild 15: Datenbankdatei übernehmen?

Tipps und Tricks

Kleine Tipps und Tricks rund um die Programmierung mit Visual Studio, die keinen eigenen Artikel benötigen, landen regelmäßig in dieser Kategorie.

Kommentieren in Visual Studio/C#

Die üblichen Kommentarzeichen kennen Sie ja bereits:

- der doppelte Schrägstrich markiert den Rest der Zeile als Kommentar und
- die Kombination Schrägstrich/Sternchen und Sternchen/Schrägstrich markiert alle dazwischen liegenden Zeichen als Kommentar.

Mit dem doppelten Schrägstrich markieren Sie also eine komplette Zeile als Kommentar:

```
//Dies ist eine Kommentarzeile.
```

Oder nur der hintere Teil soll als Kommentar interpretiert werden:

```
Console.WriteLine(); //Leere Zeile
```

Mehrere Zeilen schließen Sie entsprechend ein:

```
/*Dies ist ein
mehrzeiliger Kommentar.*/
```

Sie können allerdings auch einen Kommentar mitten in einer Codezeile unterbringen:

```
int /*Mittendrin*/ Test;
```

Ob und wo dies sinnvoll ist, entscheiden Sie selbst. Vielleicht kann man so Stellen im Code markieren, an denen noch etwas geändert werden muss.

Tastenkombinationen für Kommentare

Wenn Sie eine komplette Zeile oder mehrere Zeilen schnell als Kommentar markieren möchten, betätigen Sie einfach die Tastenkombination **(Strg + K) (Strg + C)**. Um als Kommentarzeilen markierte Zeilen wieder auszukommentieren, verwenden Sie die Tastenkombination **(Strg + K) (Strg + U)**. Zum Auskommentieren einer kompletten Zeile reicht es, wenn sich vor dem Betätigen der Tastenkombination die Einfügemarke in der Zeile befindet. Für mehrere Zeilen muss sich die Markierung über die betroffenen Zeilen erstrecken, wobei die erste und letzte Zeile nicht komplett markiert sein müssen.

Wenn Sie nur ein paar Zeichen innerhalb einer Zeile markieren möchten, also beispielsweise einige Zeichen mitten in der Zeile oder die hinteren Zeichen der Zeile, markieren Sie den Bereich und betätigen dann wiederum **(Strg + K) (Strg + C)**. Die Zeichen werden dann in **/*** und ***/** eingefasst – auch wenn sich die Markierung am Ende der Zeile befindet und theoretisch das Einleiten des auskommentierten Bereichs mit **//** ausgereicht hätte.

Die Auskommentierung mehrerer Zeilen mit **/*** und ***/** erreichen Sie übrigens nur, wenn Sie dies manuell eingeben – das Markieren mehrerer Zeilen und das Betätigen der Tastenkombination **(Strg + K) (Strg + C)** versieht mehrere markierte Zeilen immer mit führendem **//**.

Wer Tastenkombinationen nicht so gern nutzt, kann den markierten Bereich auch mit den beiden Schaltflächen der Symbolleiste Text-Editor ein- und auskommentieren (siehe Bild 1).

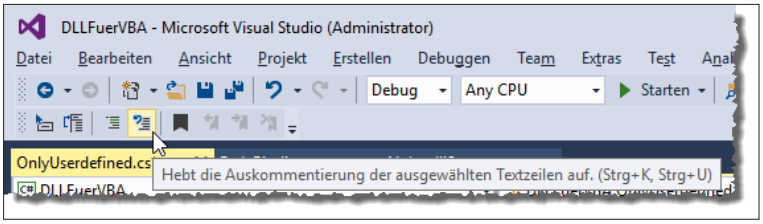


Bild 1: Ein- und Auskommentieren per Menüleiste

Einer Anwendung beim Start Parameter übergeben

Wenn Sie eine **.exe**-Datei erstellen, beispielsweise als Konsolenanwendung, möchten Sie dem Benutzer vielleicht die Übergabe von Parametern ermöglichen. Dazu ergänzen Sie die Methode **Main** um den Code aus Listing 1.

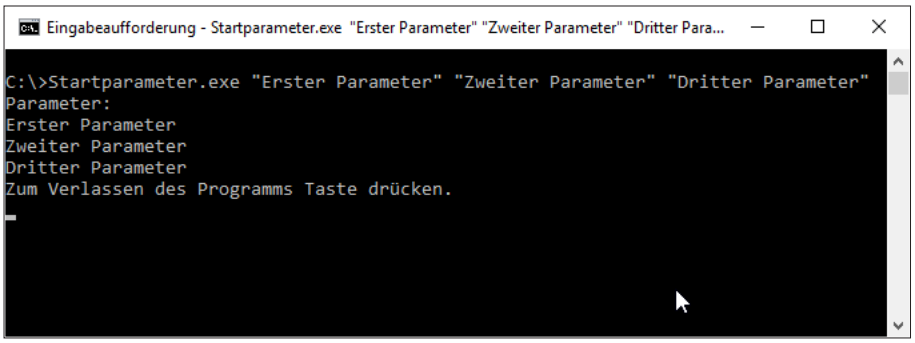


Bild 2: Ausprobieren der Parameterübergabe

Die Methode erwartet mit dem Parameter **args** eine Liste von **String**-Argumenten. Die **if**-Bedingung prüft mit dem Vergleich der Eigenschaft **args.Length** mit dem Wert **0**, ob **args** mehr als **0** Elemente enthält. Ist dies der Fall, durchläuft eine **for**-Schleife die enthaltenen Elemente. Dazu verwendet sie als Startwert für die Laufvariable **i** den Wert **0** und erhöht diesen mit jedem Durchlauf um den Wert **1**

Solange **i** kleiner als die Anzahl der enthaltenen Elemente ist, gibt die Methode den Inhalt des Arrays **args** für den jeweiligen Indexwert aus der Variablen **i** in der Konsole aus. Bei drei Elementen durchläuft die Schleife also die Werte **0**, **1** und **2**. Der Wert **3** ist nicht mehr kleiner als die Anzahl der Elemente, also wird die Schleife nicht erneut durchlaufen.

Um dies selbst auszuprobieren, legen Sie ein neues Projekt auf Basis der Vorlage **Visual C#|Konsolenanwendung** an und ergänzen den Code der Methode **Main** wie hier abgedruckt. Betätigen Sie dann den Menübefehl **Erstellen|Startparameter erstellen** (vorausgesetzt Sie haben das Projekt ebenfalls **Startparameter** genannt).

Sie können die dabei erzeugte **.exe**-Datei **Startparameter.exe** dann aus dem Projektverzeichnis **C:\Users\<Benutzername>\...\Startparameter\bin\Debug** etwa nach **c:** kopieren und dann mit folgendem Befehl aufrufen:

```
Startparameter.exe "Erster Parameter" "Zweiter Parameter"
"Dritter Parameter"
```

```
class Program {
    static void Main(string[] args) {
        if (args.Length > 0 ) {
            Console.WriteLine("Parameter:");
            for(int i=0;i < args.Length;i++) {
                Console.WriteLine(args[i]);
            }
        }
        else {
            Console.WriteLine("Keine Parameter übergeben.");
        }
        Console.WriteLine("Zum Verlassen des Programms Taste drücken.");
        Console.ReadLine();
    }
}
```

Listing 1: Programm, das die Auswertung von Parametern zeigt