HANSER

Leseprobe

zu

Visual C# 2017
Grundlagen, Profiwissen und Rezepte

Walter Doberenz
Thomas Gewinnus
Jürgen Kotz
Walter Saumweber

ISBN (Buch): 978-3-446-45359-3
ISBN (E-Book): 978-3-446-45370-8
Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/
sowie im Buchhandel

Inhalt

Vorwort

C# ist die momentan von Microsoft propagierte Sprache, sie bietet die Möglichkeiten und Flexibilität von C++ und erlaubt trotzdem eine schnelle und unkomplizierte Programmierpraxis wie einst bei Visual Basic. C# ist (fast) genauso mächtig wie C++, wurde aber komplett neu auf objektorientierter Basis geschrieben.

Damit ist C# das ideale Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework, beginnend bei Windows Forms- über WPF-, ASP.NET-, und mobilen-Anwendungen (mittlerweile auch für Android und iOS) bis hin zu systemnahen Applikationen.

Das vorliegende Buch ist ein Angebot für künftige wie auch für fortgeschrittene C#-Programmierer. Seine Philosophie knüpft an die vielen anderen Titel an, die in dieser Reihe in den vergangenen zwanzig Jahren zu verschiedenen Programmiersprachen erschienen sind:

- Programmieren lernt man nicht durch lineares Durcharbeiten eines Lehrbuchs, sondern nur durch unermüdliches Ausprobieren von Beispielen, verbunden mit ständigem Nachschlagen in der Referenz.
- Der Umfang einer modernen Sprache wie C# in Verbindung mit Visual Studio ist so gewaltig, dass ein seriöses Programmierbuch das Prinzip der Vollständigkeit aufgeben muss und nach dem Prinzip „so viel wie nötig“ sich lediglich eine „Initialisierungsfunktion“ auf die Fahnen schreiben kann.

Gegenüber anderen Büchern zur gleichen oder ähnlichen Thematik nimmt unser Titel für sich in Anspruch, gleichzeitig Lehr- und Übungsbuch zu sein.

Zum Buchinhalt

Wie Sie bereits dem Buchtitel entnehmen können, wagt das vorliegende Werk den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in Visual C# 2017 zu liefern oder all die Informationen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation (MSDN) ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die F1-Taste trennt.

- Für den *Einsteiger* wollen wir den einzig vernünftigen und gangbaren Weg beschreiten, nämlich nach dem Prinzip „so viel wie nötig“ eine schmale Schneise durch den Urwald der .NET-Programmierung mit Visual C# 2017 schlagen, bis er eine Lichtung erreicht hat, die ihm erste Erfolgserlebnisse vermittelt.

- Für den *Profi* wollen wir in diesem Buch eine Vielzahl von Informationen und Know-how bereitstellen, wonach er bisher in den mitgelieferten Dokumentationen, im Internet bzw. in anderen Büchern vergeblich gesucht hat.

Die Kapitel des Buchs und die Bonuskapitel haben wir in vier Themenkomplexen gruppiert:

1. Grundlagen der Programmierung mit C# (Buch)
2. Technologien (Buch)
3. Windows Forms-Anwendungen (online)
4. WPF-Anwendungen (online)

Die Kapitel innerhalb eines Teils bilden einerseits eine logische Aufeinanderfolge, können andererseits aber auch quergelesen werden. Im Praxisteil eines jeden Kapitels werden anhand realer Problemstellungen die behandelten Programmiertechniken im Zusammenhang demonstriert.

Zu den Codebeispielen

Um den Einstieg in C# so einfach wie möglich zu gestalten, haben wir uns entschlossen die Beispiele mit WindowsForms zu erstellen.

Alle Beispieldaten dieses Buchs und die Bonuskapitel können Sie sich unter der folgenden Adresse herunterladen:

downloads.hanser.de

Beim Nachvollziehen der Buchbeispiele beachten Sie bitte Folgendes:

- Kopieren Sie die Buchbeispiele auf die Festplatte. Wenn Sie auf die Projektmappendatei (**.sln*) klicken, wird Visual Studio in der Regel automatisch geöffnet und das jeweilige Beispiel wird in die Entwicklungsumgebung geladen, wo Sie es z. B. mittels F5-Taste kompilieren und starten können.
- Einige wenige Datenbankprojekte verwenden absolute Pfadnamen, die Sie vor dem Kompilieren des Beispiels erst noch anpassen müssen.
- Für einige Beispiele sind ein installierter Microsoft SQL Server Express LocalDB erforderlich.
- Beachten Sie die zu einigen Beispielen beigefügten *Liesmich.txt*-Dateien, die Sie auf besondere Probleme hinweisen.

Nobody is perfect

Sie werden in diesem Buch nicht alles finden, was Visual C# 2017 bzw. das .NET Framework 4.7 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel noch besser oder ausführlicher beschrieben. Aber Sie halten mit unserem Buch einen überschaubaren Breitband-Mix in den Händen, der sowohl vertikal vom Einsteiger bis zum Profi als auch horizontal von den einfachen Sprachelementen bis hin zu komplexen Anwendungen jedem etwas bietet, ohne dabei den Blick auf das Wesentliche im .NET-Dschungel zu verlieren.

Wenn Sie Vorschläge oder Fragen zum Buch haben, können Sie uns gern unter

juergen.kotz@primetime-software.de

kontaktieren.

Wir hoffen, dass wir Ihnen mit diesem Buch einen nützlichen Begleiter bei der .NET-Programmierung zur Seite gestellt haben, der es verdient, seinen Platz nicht im Regal, sondern griffbereit neben dem Computer einzunehmen.

Jürgen Kotz, Walter Saumweber, Walter Doberenz und Thomas Gewinnus

München, im Dezember 2017

14 Einführung in ADO.NET und Entity Framework

Nach wie vor ist ADO.NET die Basis-Datenzugriffstechnologie des .NET-Frameworks. Auch wenn in vielen neuen Projekten mittlerweile das *Entity Framework* zum Einsatz kommt, darf nicht vergessen werden, dass die Basis des Entity Frameworks auch weiterhin ADO.NET ist. Der Inhalt des vorliegenden Kapitels beschränkt sich auf eine knappe Einführung in die ADO.NET-Technologie sowie des Entity Frameworks, wobei der Schwerpunkt auf dem ersten Teil des Objektmodells, den Datenprovidern, liegt.

Erst das Kapitel 14 widmet sich ausführlich einem der Kernobjekte von ADO.NET, dem *DataSet*. Auf die speziellen Probleme der Datenbindung, d. h., auf die Interaktion der Benutzerschnittstelle mit den ADO.NET-Objekten, gehen wir in den speziellen Kapiteln 26 und 33 in Bezug auf den jeweiligen Anwendungstyp näher ein.

HINWEIS: Ausgeklammert werden musste in diesem Buch die weiterführende Technologie LINQ to Entities, einen kleinen Vorgeschmack bietet Ihnen die Kurzeinführung zu Entity Framework.

14.1 Eine kleine Übersicht

Die umfangreichen Klassenbibliotheken von ADO.NET verlangen vom Einsteiger eine ziemlich steile „Lernkurve“. Er tut gut daran, sich zunächst einen Gesamtüberblick zu verschaffen.

14.1.1 Die ADO.NET-Klassenhierarchie

ADO.NET setzt sich aus einer ziemlich komplexen Hierarchie vieler Klassen zusammen. Die daraus erzeugten Objekte lassen sich zunächst in zwei Gruppen aufteilen:

- Datenprovider
- Datenkonsument

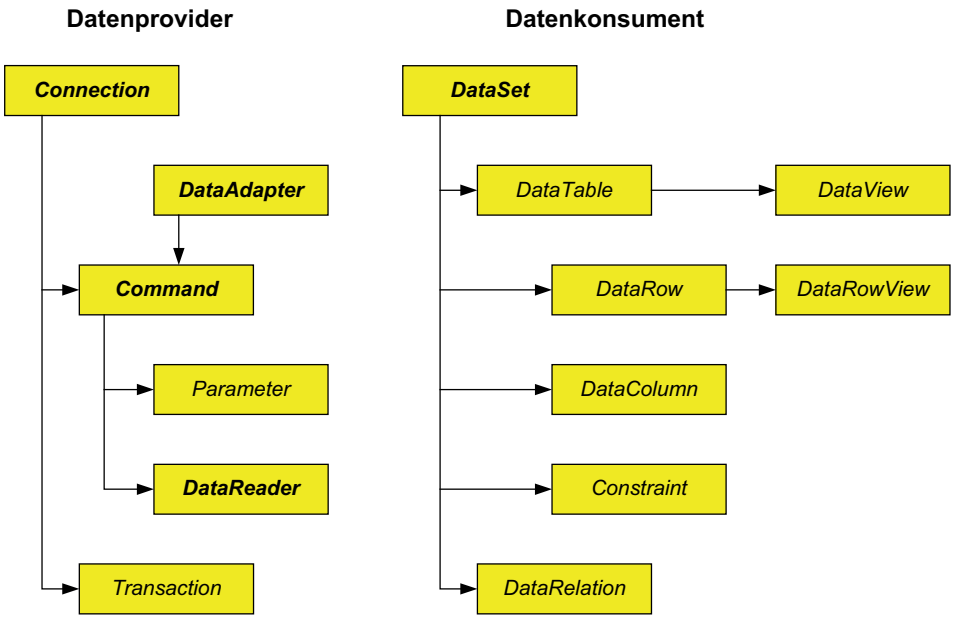
Während der *Datenprovider* die Daten zur Verfügung stellt, ist der *Datenkonsument* der Teil einer Applikation, welcher die Dienste eines Datenproviders nutzt, um auf beliebige Daten zuzugreifen, sie zu lesen, zu speichern und zu ändern.

Die Objekte *Connection*, *Command*, *DataReader* und *DataAdapter* sind die Hauptelemente des .NET-Datenprovider-Modells.

Man bezeichnet die Datenprovider auch als *Verbundene Objekte*, da sie immer in Beziehung zu einer bestimmten Datenquelle stehen. Die Datenkonsumenten hingegen sind *Unverbundene Objekte*, weil sie – ganz im Sinne der ADO.NET-Philosophie – unabhängig von einer Datenquelle ihr völlig autarkes Dasein führen.

Der allen übergeordnete Datenkonsument ist das *DataSet*. Es ist gewissermaßen das Kernobjekt von ADO.NET und vergleichbar mit den vom alten ADO her bekannten *Recordset*-Objekten. Allerdings ist es weitaus komplexer, da es z. B. mehrere *DataTable*-Objekte und die Beziehungen (Relationen) zwischen ihnen kapseln kann. Ein *DataSet* kann (unter Verwendung eines *DataAdapters*) direkt von der Datenquelle geladen werden. Es kann aber auch – ähnlich einem Array – völlig unabhängig von einer Datenbank mit Werten gefüllt werden. Der *DataAdapter* ist quasi das Verbindungsglied zwischen Datenprovider (Datenbank) und Datenkonsument (unverbundene Objekte).

HINWEIS: Um ein erstes praktisches Feeling für die ADO.Net-Klassen zu entwickeln, sollte der Einsteiger bereits jetzt ein einfaches Beispiel ausprobieren, z. B. das Praxisbeispiel in Abschnitt 14.9.1, Wichtige ADO.NET-Objekte im Einsatz.



14.1.2 Die Klassen der Datenprovider

Im Einklang mit dem ADO.NET-Objektmodell sind Datenprovider stets in mehrfacher Ausfertigung vorhanden. Die Präfixe charakterisieren die Zugehörigkeit zu einem bestimmten *.NET-Datenprovider,* z. B.:

- *OleDb...*
  Diese Klassen (z. B. *OleDbConnection*) dienen dem OLE-Db-Zugriff auf unterschiedlichste Datenbanktypen, für die ein Treiber installiert ist.
- *Sql...*
  Diese Klassen (z. B. *SqlConnection*) dienen dem schnelleren Direktzugriff auf den Microsoft SQL Server.

Der *Datenprovider* im .NET Framework kapselt die Datenbank und ermöglicht den Zugriff über eine einheitliche Schnittstelle. Er fungiert quasi als Brücke zwischen einer Anwendung und einer Datenbank und wird zum Abrufen von Daten aus einer Datenbank und zum Abgleichen von Änderungen an diesen Daten mit der Datenbank verwendet.

Die Datenquelle selbst kann eine beliebige Struktur haben und sich an einem beliebigen Ort befinden, z. B. eine lokale Access-Datenbank, ein SQL-Server oder aber auch eine Oracle-Datenbank.

.NET-Datenprovider

In der folgenden Tabelle sind wichtige Klassen der *OleDb-* und *SqlServer*-Provider paarweise aufgelistet:

System.Data.OleDb	System.Data.SqlClient	Bedeutung
OleDbConnection	*SqlConnection*	Stellt die Verbindung zur Datenquelle her
OleDbCommand	*SqlCommand*	Führt eine SQL-Abfrage aus
OleDbDataReader	*SqlDataReader*	Ermöglicht einen sequenziellen Nur-Lese-Zugriff auf die Datenquelle
OleDbDataAdapter	*SqlDataAdapter*	Ermöglicht das Füllen eines *DataSet*s mit den Ergebnissen einer SQL-Abfrage
OleDbCommandBuilder	*SqlCommandBuilder*	Erstellt automatisch *Command*-Objekte für die Übernahme der in einem *DataSet* vorgenommenen Änderungen in die Datenbank
OleDbTransaction	*SqlTransaction*	Organisiert die Anwendung von Transaktionen

Weitere Datenprovider

Die Liste der .NET-Datenprovider ist keinesfalls nur auf die nach einer Standardinstallation vorhandenen Provider beschränkt. Neben *OleDb* und *SqlClient* sind in ADO.NET u. a. auch die folgenden Provider enthalten:

- *System.Data.Odbc*
- *System.Data.SqlServerCe*
- *System.Data.OracleClient*

Anzeige der installierten Datenprovider

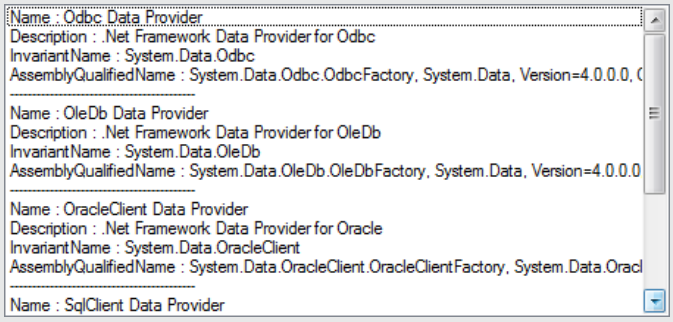
Einen Überblick über alle auf Ihrem System installierten ADO.NET-Datenprovider können Sie mit der Methode *GetFactoryClasses* der *DbProviderFactories*-Auflistung aus dem *System.Data.Common*-Namespace gewinnen.

Beispiel 14.1: Alle verfügbaren Datenprovider in einer *ListBox* anzeigen

C#

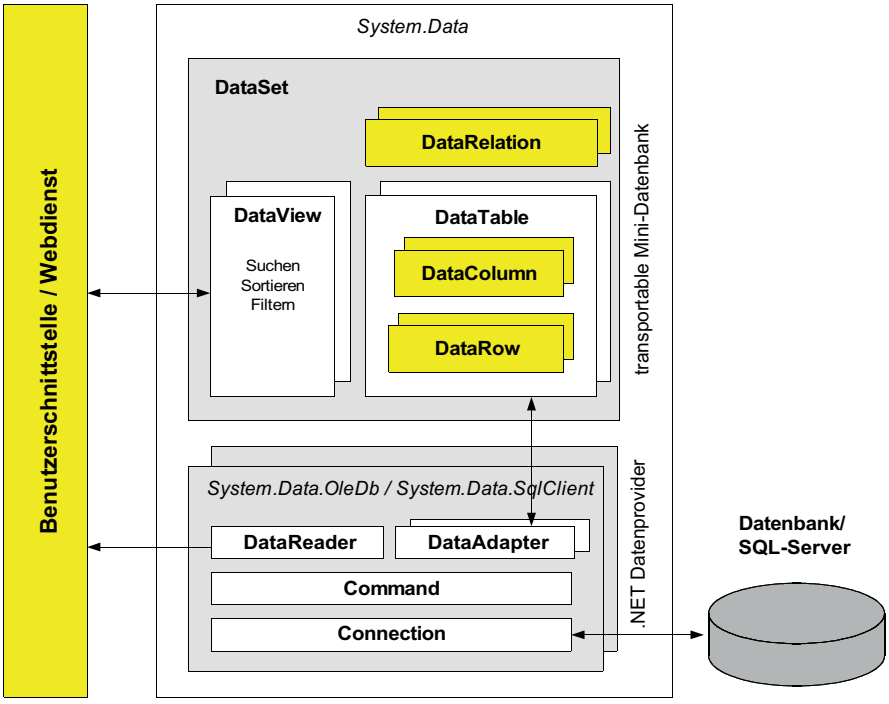
```
DataTable providers = System.Data.Common.DbProviderFactories.GetFactoryClasses();
foreach (DataRow provider in providers.Rows)
{
   foreach (DataColumn col in providers.Columns)
          listBox1.Items.Add(col.ColumnName + " : " + provider[col]);
   listBox1.Items.Add("-----------------------------------------");
}
```

Ergebnis



14.1.3 Das Zusammenspiel der ADO.NET-Klassen

In der Abbildung wird versucht, den grundlegenden Zusammenhang zwischen den ADO.NET-Klassen in vereinfachter Form zu verdeutlichen:



Die in der obigen Abbildung angegebenen Namespaces (Namensräume) für die ADO.NET-Klassen sind:

- *System.Data*
- *System.Data.OleDb*
- *System.Data.SQLClient*

Das *DataSet* ist vollständig von der Datenbank entkoppelt, denn dazwischen hat sich ein *.NET-Datenprovider* geschoben, der im Bedarfsfall den Datentransport (über die OLE-DB- bzw. die direkte SQL-Server-Schnittstelle) übernimmt.

HINWEIS: Eine einfache und schnellere Möglichkeit für den Zugriff auf die Datenquelle ist der *DataReader*, da mit ihm auf direktem Weg – also ohne *DataAdapter* und *DataSet* – Daten in die Benutzeroberfläche eingelesen werden können. Das Entity Framework benutzt ausschließlich den DataReader, um die Objektstrukturen zu füllen.

14.2 Das Connection-Objekt

Um überhaupt auf eine Datenbank zugreifen zu können, muss als Erstes eine Verbindung zu ihr hergestellt werden. Dazu führt in der Regel kein Weg am *Connection*-Objekt vorbei.

14.2.1 Allgemeiner Aufbau

Der am häufigsten zum Erzeugen und Initialisieren eines *Connection*-Objekts benutzte Konstruktor nimmt einen *ConnectionString* als Parameter entgegen:

Syntax:

```
Connection conn = new Connection(string ConnectionString);
```

Der *ConnectionString* – die gleichzeitig auch wichtigste Eigenschaft des *Connection*-Objekts – kapselt alle erforderlichen Verbindungsparameter.

Durch Aufruf der (parameterlosen) *Open*-Methode erhält das *Connection*-Objekt eine offene Verbindung aus dem Verbindungspool, falls diese verfügbar ist. Andernfalls wird eine neue Verbindung mit der Datenquelle erstellt.

Nach einer Standardinstallation von Visual Studio stehen – je nach Auswahl des .NET-Providers – verschiedene *Connection*-Objekte zur Verfügung, z. B.:

- *OleDbConnection*-Objekt
  ... gewährleistet den Zugriff auf eine Vielzahl von Datenquellen, angefangen mit einfachen Textdateien über Tabellen bis hin zu kompletten Datenbanken.
- *SqlConnection*-Objekt
  ... ist speziell für die Verwendung mit dem SQL Server optimiert, indem die OLE-DB-Schicht umgangen wird.

14.2.2 SqlConnection

Parameter für Sql-Zugriff

Die Tabelle zeigt die wichtigsten Angaben für den Sql-Server-Zugriff:

Parameter	Bedeutung
Data Source	Name der Datenquelle (Name der SQL-Server-Instanz, z. B. *localhost* \ *SqlExpress*)
Initial Catalog	Name der Datenbank
Integrated Security	Kennzeichen, ob über die Windows-Authentifizierung auf die Datenbank zugegriffen werden soll (alternativ *UserName* und *Password*)

Sql-Provider für Microsoft-SQL-Server-Datenbank

Beispiel 14.2: Öffnen einer SQL-Verbindung zur Northwind-Datenbank mittels Windows-Authentifizierung.

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection(
       "Data Source=.\\SqlExpress;Initial Catalog=Northwind;
                        Integrated Security=true");
conn.Open();
```

Im obigen Beispiel wurde der *ConnectionString* dem *new*-Konstruktor übergeben. Man kann ihn aber auch separat zuweisen, wie das folgende Beispiel zeigt.

Beispiel 14.3: Eine zum Vorgängerbeispiel äquivalente Variante

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection();
conn.ConnectionString = "Data Source=.\\SqlExpress;
        Initial Catalog=Northwind;Integrated Security=true");
conn.Open();
```

Wie Sie erkennen, besteht ein *ConnectionString* aus einer Zeichenfolge mit Attribut/Wert-Paaren für Informationen, die zum Anmelden an einen Datenbankserver und Zugriff auf eine bestimmte Datenbank erforderlich sind.

HINWEIS: Die Reihenfolge der Parameter im *ConnectionString* ist ohne Bedeutung!

14.2.3 Schließen einer Verbindung

Nachdem die Daten übertragen worden sind, sollte die Verbindung mithilfe der *Close*-Methode wieder geschlossen werden. Ansonsten bleibt die Connection weiter geöffnet, auch nachdem die Connection-Instanz selbst terminiert wurde!

Um die Netzbelastung gering zu halten, sollte man – ganz im Sinne der ADO.NET-Philosophie – das Öffnen und Schließen einer Verbindung möglichst innerhalb einer einzigen Befehlskette durchführen.

Beispiel 14.4: Es wird kurzzeitig eine Verbindung zur SqlServer-Datenbank *Northwind* geöffnet, um die Kundentabelle in ein DataSet zu übertragen. Danach wird die Verbindung sofort wieder geschlossen.

C#

```
using System.Data.SqlClientOleDb;
...

SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
```

Verbindung öffnen:

```
       conn.Open();
```

Daten übertragen:

```
       DataSet ds = new DataSet();
       SqlDataAdapter da = new SqlDataAdapter(
                           "SELECT * FROM Customers", conn);
       da.Fill(ds, "Kunden");
```

Verbindung schließen:

```
       conn.Close();
```

HINWEIS: Obiger Code ist in vielen Fällen nicht ganz unproblematisch, z. B. wenn die Datenbank nicht vorhanden ist. Die saubere Lösung wird im Abschnitt 14.3.5diskutiert. Vielleicht sollten Sie hier auch einen using-Block verwenden.

14.2.4 Eigenschaften des Connection-Objekts

Da wir im Verlauf dieses Abschnitts bereits viele Eigenschaften des *Connection*-Objekts en passant besprochen haben, soll diese knappe Zusammenfassung den Überblick erleichtern und gleichzeitig einige zusätzliche Informationen liefern.

ConnectionString-Eigenschaft

Diese zweifelsohne wichtigste Eigenschaft des *Connection*-Objekts kapselt sämtliche Verbindungsinformationen zur Datenbank. Außerdem ist es die einzige Eigenschaft, die nicht schreibgeschützt ist (wenn keine Verbindung zur Datenquelle besteht).

Database- und DataSource-Eigenschaft

Was ist der Unterschied zwischen beiden Eigenschaften? Die *DataSource*-Eigenschaft des *Connection*-Objekts entspricht dem *Data Source*-Attribut innerhalb des *ConnectionString*s und enthält den Speicherort der Datenquelle.

Für eine serverbasierte Datenquelle (Microsoft SQL Server, Oracle) bedeutet der Speicherort den Namen des Computers, auf dem der Server installiert ist. Bei dateibasierten Datenbanken, wie z. B. Access, verweist diese Eigenschaft auf den Datenbankpfad (z. B. *c:\Beispiele\Nordwind.mdb*).

Die *Database*-Eigenschaft ist hingegen für Datenquellen, wie z. B. den SQL Server, gedacht, die mehrere Datenbanken unterstützen, und entspricht dem Attribut *Initial Catalog* im *ConnectionString*. Beim *SQL-Server-OleDb-Provider* können wir aber alternativ beide Attributbezeichner verwenden.

Beispiel 14.5: Zwei gleichwertige Möglichkeiten

C#

```
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=.\\SQLEXPRESS; " +
                        "Initial Catalog=Northwind;Integrated Security=SSPI";
oder
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=.\\SQLEXPRESS; " +
                        "Database=Northwind;Integrated Security=SSPI";
label1.Text = conn.DataSource;                // liefert ".\SQLEXPRESS"
label2.Text = conn.Database;                  // liefert "Northwind"
```

Provider-Eigenschaft

Es klingt möglicherweise etwas verwirrend: Während wir unter dem Begriff *.NET-Datenprovider* eine Klassenbibliothek für den Datenzugriff verstehen, ist *Provider* auch eine Eigenschaft des *OleDbConnection*-Objekts.

Die *Provider*-Eigenschaft bezeichnet hier die OLE-DB-Schnittstelle, welche die Datenquelle des jeweiligen Herstellers kapselt. Die Tabelle erklärt einige häufig benutzte OLE-DB-Schnittstellen:

Datenquelle	Provider-Eigenschaft
Microsoft Access	Microsoft.Jet.OLEDB.4.0
Microsoft SQL Server	SQLOLEDB.1
Microsoft Indexing Service	MSIDXS.1
Oracle	MSDAORA.1

ServerVersion-Eigenschaft

Diese Eigenschaft liefert eine Zeichenfolge zurück, die die Version der Datenbank enthält. Durch Abprüfen von *ServerVersion* können Sie z. B. gewährleisten, dass keine Abfragen an den Server geschickt werden, die von diesem nicht unterstützt werden (z. B. Abfrageergebnisse als XML liefern).

ConnectionTimeout-Eigenschaft

Obwohl diese Eigenschaft schreibgeschützt ist, haben Sie trotzdem die Möglichkeit, innerhalb des *ConnectionString* anzugeben, wie viel Sekunden der OleDb-Provider versuchen soll, die Verbindung zur Datenbank herzustellen.

Beispiel 14.6: Zeit bis zum Timeout der Verbindungsaufnahme auf zehn Sekunden festlegen

C#

```
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=.\\SQLEXPRESS;...;
                         Connect Timeout=10; ... " ;
```

State-Eigenschaft

Diese Eigenschaft liefert den aktuellen Verbindungsstatus. Die möglichen Werte sind Mitglieder der *ConnectionState*-Enumeration.

Konstante	Verbindungszustand
Open	Geöffnet
Closed	Geschlossen
Connecting	Verbindung wird aufgebaut.
Executing	Eine Abfrage wird ausgeführt.
Fetching	Daten werden abgerufen.
Broken	Unterbrochen

14.2.5 Methoden des Connection-Objekts

Open- und Close-Methode

Wenn Sie die *Open*-Methode auf einem bereits geöffneten *Connection*-Objekt ausführen, wird ein Fehler ausgelöst. Hingegen verursacht der Aufruf von *Close* über einer bereits geschlossenen Verbindung keinen Fehler.

HINWEIS: Da Sie standardmäßig mit Verbindungspooling arbeiten, wird die Verbindung nicht wirklich geschlossen, sondern nur zurück an den Pool gesendet.

Es ist keine Vergesslichkeit der Autoren, wenn in manchen Beispielen das *Connection*-Objekt weder mit *Open* geöffnet noch mit *Close* geschlossen wird. Gewissermaßen im Hintergrund können *Fill*- und *Update*-Methode eines *DataAdapter*-Objekts automatisch die Verbindung öffnen (wenn sie nicht schon geöffnet ist) und sie auch wieder schließen, wenn die Operation beendet ist.

ChangeDatabase-Methode

Viele Server, wie z. B. auch der SQL Server, unterstützen mehrere Datenbanken. Mit der *ChangeDatabase*-Methode können Sie die Datenbank zur Laufzeit wechseln, ohne den USE-Befehl verwenden zu müssen.

Beispiel 14.7: Zwei äquivalente Varianten zum Wechseln der Datenbank

C#

```
conn.ChangeDatabase("Northwind");
```

oder

```
OleDbCommand cmd = conn.CreateCommand();
cmd.CommandText = "USE Northwind";
cmd.ExecuteNonQuery();
```

CreateCommand-Methode

Mit dieser Methode können Sie ein neues *Command*-Objekt erzeugen und damit etwas Schreibarbeit einsparen (siehe obiges Beispiel).

14.2.6 Der ConnectionStringBuilder

Um das Zusammenbauen eines *ConnectionString*s etwas übersichtlicher zu gestalten, gibt es die providerspezifische *ConnectionStringBuilder*-Klasse.

Beispiel 14.8: Vergleich von zwei Möglichkeiten für das Erstellen einer Verbindungszeichenfolge zur *Northwind*-Datenbank des SQL Servers

C#

```
using System.Data.SqlClient;
```

Ohne *ConnectionStringBuilder*:

```
string connStr = "Data Source = .\\SQLEXPRESS; Initial Catalog=Northwind;
                  Integrated Security=true";
conn = new SqlConnection(connStr);
```

Mit *ConnectionStringBuilder*:

```
SqlConnectionStringBuilder csb = new SqlConnectionStringBuilder();
csb.DataSource = ".\\SQLEXPRESS";
csb.IntegratedSecurity = true;
csb.InitialCatalog = "Northwind";
```

14.3 Das Command-Objekt

An Abfragen aller Art (SQL-Queries, Stored Procedures) führt beim Programmieren von Datenbankanwendungen kein Weg vorbei. Unter ADO.NET werden für alle Datenbankabfragen *Command*-Objekte benutzt, die zentraler Bestandteil der jeweiligen .NET-Datenprovider sind.

14.3.1 Erzeugen und Anwenden eines Command-Objekts

Wie bei fast allen anderen ADO.NET-Objekten, bieten sich auch zum Erzeugen eines *Command*-Objekts verschiedene Konstruktoren an. Eine übliche Vorgehensweise ist es, die gewünschte Abfrage neben dem zuvor angelegten *Connection*-Objekt an den Konstruktor der Klasse zu übergeben:

Syntax:

```
Command cmd = new Command(string sqlCommand, Connection conn);
```

Das so erzeugte und initialisierte *Command*-Objekt kann dann z. B. an den Konstruktor der *DataAdapter*-Klasse weitergereicht werden, um letztendlich ein *DataSet* zu füllen.

Aber es geht auch ohne *DataAdapter* und *DataSet*, denn um SQL-Anweisungen direkt gegen die Datenquelle zu fahren, kann eine der *Execute*-Methoden (*ExecuteNonQuery*, *ExecuteReader*, *ExecuteScalar*) aufgerufen werden.

Beispiel 14.9: Erzeugen und Anwenden eines *Command*-Objekts

C#

Es werden zwei *SqlCommand*-Objekte erstellt. Mit dem ersten werden die Namen der Firmen aller Pariser Kunden aus der Northwind-Datenbank geändert, mit dem zweiten wird ein *SqlDataAdapter* erstellt, der zum Befüllen eines *DataSet*-Objekts mit den Kundendatensätzen dient.

```
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand updCmd = new SqlCommand("UPDATE Custmers
          SET CompanyName = 'Pariser Kunde' WHERE City = 'Paris'", conn);
SqlCommand selCmd = new SqlCommand("SELECT CompanyName, ContaktName, City
                    FROM Customers WHERE City = 'Paris'", conn);
SqlDataAdapter da = new SqlDataAdapter(selCmd);
DataSet ds = new DataSet();
conn.Open();
```

UPDATE-Befehl wird gegen die Datenbank gefahren:

```
updCmd.ExecuteNonQuery();
```

DataSet erhält neue Tabelle („PariserKunden“) mit Datensätzen gemäß SELECT-Befehl:

```
da.Fill(ds, "PariserKunden");
conn.Close();
```

HINWEIS: Den vollständigen Code finden Sie im Praxisbeispiel in Abschnitt 14.9.2, Eine Aktionsabfrage ausführen.

14.3.2 Erzeugen mittels CreateCommand-Methode

Auch mithilfe der *CreateCommand*-Methode eines *Connection*-Objekts können Sie ein *Command*-Objekt erzeugen. Damit ersparen Sie sich etwas Schreibarbeit.

Beispiel 14.10: Zwei äquivalente Varianten, wenn ein gültiges *Connection*-Objekt vorliegt

C#

Variante A:

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
```

Variante B:

```
SqlCommand cmd = conn.CreateCommand();
```

14.3.3 Eigenschaften des Command-Objekts

Wir werden uns auch hier auf eine knappe Darstellung der wichtigsten Eigenschaften beschränken.

Connection- und CommandText-Eigenschaft

Beide Eigenschaften werden üblicherweise bereits im Konstruktor übergeben (siehe oben), man kann sie aber auch separat zuweisen.

Beispiel 14.11: Zwei Varianten zum Erzeugen und Initialisieren eines *SqlCommand*-Objekts

C#

```
SqlCommand cmd = new SqlCommand("UPDATE Customers
        SET CompanyName = 'Pariser Firma' WHERE City = 'Paris'", conn);
```

... ist äquivalent zu:

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = "UPDATE CompanyName SET City = 'Pariser Firma'
                           WHERE City = 'Paris'";
```

CommandTimeout-Eigenschaft

Um festzulegen, wie lange die Ausführung einer Abfrage maximal dauern darf, können Sie der *CommandTimeout*-Eigenschaft einen Wert in Sekunden zuweisen (Standardwert = 30 Sekunden).

Beispiel 14.12: Ein *DataSet* wird mit der Kundenliste der SQL-Datenbank *Northwind* gefüllt, wofür maximal 30 Sekunden zur Verfügung stehen.

C#

```
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand cmd = new SqlCommand("SELECT CustomerID, CompanyName
                 FROM Customers", conn);
//Die Ausführung der Abfrage darf maximal 30 sek dauern:
cmd.CommandTimeout = 30;
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = cmd;
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "Kunden");
conn.Close();
```

HINWEIS: Sollte eine Abfrage dennoch zu lange dauern, so können Sie diese innerhalb einer asynchronen Umgebung mithilfe der *Cancel*-Methode abbrechen.

Zum Ausführen einfacher Datenbankabfragen (wie im obigen Beispiel) können Sie auf das explizite Erzeugen eines *Command*-Objekts verzichten, denn man kann den SQL-String auch direkt dem *DataAdapter* als Parameter übergeben.

Beispiel 14.13: Eine gleichwertige Realisierung des Vorgängerbeispiels

C#

```
SqlDataAdapter da = new SqlDataAdapter("SELECT CustomerID, CompanyName
            FROM Customers", conn);
da.SelectCommand.CommandTimeout = 30;
```

CommandType-Eigenschaft

Mit der *CommandType*-Eigenschaft definieren Sie die auszuführende Operation. Mittels der gleichnamigen Enumeration stehen drei Möglichkeiten zur Verfügung:

- *Text* (Standardwert)
  Hier können Sie eine frei definierbare SQL-Abfrage übergeben.
- *StoredProcedure*
  Hier soll eine in der Datenbank gespeicherte Prozedur bzw. Auswahlabfrage aufgerufen werden.
- *TableDirect*
  Hier wird direkt der Name einer Tabelle angegeben (entspricht *SELECT * FROM <Tabellenname>*).

Beispiel 14.14: Aufruf der Stored Procedure „Sales by Years“ in der Datenbank *Northwind*

C#

```
SqlCommand cmd = new SqlCommand("Sales by Year", conn);
cmd.CommandType = CommandType.StoredProcedure;
SqlParameter parm1 = new SqlParameter("@Beginning_Date", SqlDbType.DateTime);
```

Definition als Input-Parameter:

```
parm1.Direction = ParameterDirection.Input;
```

Das *Beginn*-Datum wird der ersten *TextBox* entnommen:

```
parm1.Value = Convert.ToDateTime(textBox1.Text);
```

Parameter hinzufügen:

```
cmd.Parameters.Add(parm1);
```

14.3.4 Methoden des Command-Objekts

ExecuteNonQuery-Methode

Diese Methode setzen Sie vor allem ein, um Aktionsbefehle auf Basis von UPDATE, INSERT oder DELETE direkt gegen die Datenbank auszuführen (also ohne Verwendung von *DataAdapter* und *DataSet*). Rückgabewert ist hier die Anzahl der betroffenen Datensätze (sonst –1).

Beispiel 14.15: *ExecuteNonQuery*-Methode

C#

Ein *SqlCommand*-Objekt wird erzeugt und eine UPDATE-Anweisung gegen die Datenbank ausgeführt. Die Anzahl betroffener Datensätze wird angezeigt (ein gültiges *SqlConnection*-Objekt *conn* wird vorausgesetzt).

```
string updStr =
   "UPDATE Customer SET CompanyName = 'Pariser Firma' WHERE City = 'Paris'";
SqlCommand updCmd = new SqlCommand(updStr, conn);
conn.Open();
//SQL-Anweisung ausführen und Anzahl betroffener Datensätze anzeigen:
label1.Text = cmd.ExecuteNonQuery.ToString();
```

HINWEIS: Das ausführliche Beispiel befindet sich im Praxisbeispiel in Abschnitt 14.9.2, Eine Aktionsabfrage ausführen.

Weitere Möglichkeiten für Aktionsbefehle sind die Abfrage der Struktur einer Datenbank oder das Erstellen von Datenbankobjekten, wie z. B. Tabellen.

ExecuteReader-Methode

Auf Basis eines SELECT-Befehls erzeugt diese Methode ein *DataReader*-Objekt. Ein Instanziieren des *DataReader*s mittels *new*-Konstruktor entfällt deshalb.

Beispiel 14.16: Auf Basis eines *SqlConnection*-Objekts und eines SELECT-Befehls wird ein *SqlCommand*-Objekt erstellt und zum Erzeugen eines *SqlDataReader*-Objekts verwendet.

C#

```
string selStr = "SELECT CompanyName, ContactName, City
                FROM Customers WHERE City = 'Paris'";
SqlCommand selCmd = new SqlCommand(selStr, conn);
conn.Open();
SqlDataReader dr = selCmd.ExecuteReader(CommandBehavior.CloseConnection);
```

ExecuteScalar-Methode

Rückgabewert dieser Methode ist das Objekt der ersten Spalte der ersten Zeile aus der Menge der zurückgegebenen Datensätze. Generell eignet sich die *ExecuteScalar*-Methode des *Command*-Objekts für alle Abfragen, bei denen man nur an der Rückgabe eines einzigen Werts interessiert ist.

Beispiel 14.17: Abfrage des Namens der Firma eines bestimmten Kunden

C#

```
SqlCommand cmd = new SqlCommand("SELECT Companyname
                 FROM Customers WHERE CustomerId = 'ALFKI'", conn);
conn.Open();
label1.Text = Convert.ToString(cmd.ExecuteScalar);
```

Besonders vorteilhaft kann man *ExecuteScalar* zur Ausführung von Aggregatfunktionen verwenden, was weniger Aufwand erfordert als die Anwendung der *ExecuteReader*-Methode.

Beispiel 14.18: Aus der Datenbank *Northwind* wird die Anzahl der in Paris wohnhaften Kunden abgefragt und angezeigt.

C#

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = "SELECT COUNT(*) AS Anzahl FROM Customers
                   WHERE City = 'Paris'";
cmd.Connection.Open();
// oder auch: conn.Open();
label1.Text = cmd.ExecuteScalar().ToString();
cmd.Connection.Close()
```

14.3.5 Freigabe von Connection- und Command-Objekten

In einfachen Codebeispielen stellt man häufig fest, dass der Aufruf von *Dispose()* auf *SqlConnection*- und *SqlCommand*-Objekten fehlt.[1] Auch wurde der Datenzugriffscode nicht in *try-finally*-Blöcke eingerahmt.

Beispiel 14.19: Die (leider nicht ganz saubere) Programmierung einer Datenbankverbindung

C#

```
SqlConnection conn = new SqlConnection(connString);
SqlCommand cmd = new SqlCommand(cmdString, conn);
conn.Open();
cmd.ExecuteNonQuery();
conn.Close();
```

Das Problem ist, dass *SqlConnection* und *SqlCommand* die Schnittstelle *IDisposable* implementieren, d. h., es können auch Ressourcen aus nicht verwaltetem (unmanaged) Code zu bereinigen sein. Als Programmierer müssen Sie dann unter allen Umständen absichern, dass *Dispose()* auf diesen Objekten aufgerufen wird, nachdem die Arbeit mit ihnen beendet ist. Weil bei Nichtverfügbarkeit der Datenbank immer ein Fehler auftreten kann, sollten Sie den Aufruf von *Dispose()* auch für diesen Fall gewährleisten.

Das Problem lässt sich elegant mittels des *using*-Schlüsselworts lösen, welches Ihnen lästige Schreibarbeiten abnimmt, denn intern wird automatisch ein *try-finally*-Block um das entsprechende Objekt generiert und beim Beenden wird für das Objekt *Dispose()* aufgerufen.

Beispiel 14.20: Saubere Programmierung des Vorgängerbeispiels

C#

```
using (SqlConnection conn = new SqlConnection(connString))
{
    using (SqlCommand cmd = new SqlCommand(cmdString, conn))
    {
        conn.Open();
        cmd.ExecuteNonQuery();
    }
}
```

Der intern generierte Code für obige Zeilen dürfte etwa folgendermaßen aussehen:

```
SqlConnection conn =  null;
SqlCommand cmd = null;
try
{
    conn = new SqlConnection(connString);
    cmd = new SqlCommand(cmdString, conn);
    conn.Open();
    cmd.ExecuteNonQuery();
}
finally
```

[1] Auch die Codebeispiele dieses Buchs bilden da (aus Platzgründen!) keine Ausnahme.

```
{
    if (null != cmd) cmd.Dispose();
    if (null != conn) conn.Dispose();
}
```

HINWEIS: Falls Sie, wie im obigen Beispiel, den Aufruf von *Close()* für die *SqlConnection* vermissen, seien Sie trotzdem unbesorgt: Intern überprüft *Dispose()* den Status der Verbindung und schließt diese für Sie.

14.4 Parameter-Objekte

In vielen Fällen enthält ein *Command*-Objekt Parameter bzw. Parameter-Auflistungen, mit denen parametrisierte Abfragen durchführbar sind.

14.4.1 Erzeugen und Anwenden eines Parameter-Objekts

Einer der möglichen Konstruktoren:

Syntax:

```
Parameter prm = new Parameter(string pName, DbType pType);
```

Nach dem Zuweisen weiterer Eigenschaften erfolgt das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
cmd.Parameters.Add(Parameter prm );
```

Beispiel 14.21: Ein *SqlParameter*-Objekt *p1* wird zur *Parameters*-Collection eines vorhandenen *SqlCommand*-Objekts hinzugefügt.

C#

Im Konstruktor Namen und Datentyp übergeben:

```
SqlParameter p1 = new SqlParameter("@Geburtstag", SqlDbType.DateTime);
```

Datumswert aus einer *TextBox* zuweisen ...

```
p1.Value = Convert.ToDateTime(textBox1.Text);
```

... und zum *SqlCommand*-Objekt hinzufügen:

```
cmd.Parameters.Add(p1);
```

Die so erzeugten Parameter werden zur Laufzeit in die *CommandText*-Eigenschaft des *Command*-Objekts „eingebaut“.

Beispiel 14.22: Der definierte Parameter *@Geburtstag* wird in einer SQL-Abfrage eingefügt.

C#

```
cmd.CommandText = "SELECT * FROM Employees WHERE (BirthDate > @Geburtstag)";
```

14.4.2 Eigenschaften des Parameter-Objekts

ParameterName- und Value-Eigenschaft

Beide Eigenschaften dürften selbsterklärend sein.

Beispiel 14.23: Eine alternative Zuweisung für das obige erste Beispiel wäre:

C#

```
p1.ParameterName = "@Geburtstag";
p1.Value = Convert.ToDateTime(textBox1.Text);
```

DbType, OleDbType und SqlDbType-Eigenschaft

Durch das Spezifizieren des Datentyps wird der Wert des Parameters dem Datentyp des .NET-Datenproviders angepasst, bevor er an die Datenquelle weitergereicht wird. Fehlt die Typangabe, so leitet ihn ADO.NET von der *Value*-Eigenschaft des *Parameter*-Objekts ab.

Alternativ zur *OleDbType*- bzw. *SqlDbType*-Eigenschaft kann der Datentyp eines Parameters auch allgemein (generisch) aus *System.Data.DbType* abgeleitet werden.

Beispiel 14.24: Ein *Byte*-Parameter wird erzeugt, initialisiert und zur *Parameters*-Collection eines *SqlCommand*-Objekts hinzugefügt.

C#

```
SqlParameter prm = cmd.Parameters.Add("@p2", SqlDbType.TinyInt);
```

Direction-Eigenschaft

Die Eigenschaft bestimmt die Richtung des Parameters relativ zum *DataSet*. Die *ParameterDirection*-Enumeration enthält die in der Tabelle aufgeführten Werte:

ParameterDirection	Beschreibung
Input	Es handelt sich um einen Eingabeparameter (Standard).
InputOutput	Der Parameter unterstützt sowohl Ein- als auch Ausgabe.
Output	Es handelt sich um einen Ausgabeparameter.
ReturnValue	Der Parameter ist ein Rückgabewert aus einer Operation.

Beispiel 14.25: Ein *OleDbParameter* wird erstellt und seine *Direction*-Eigenschaft festgelegt.

C#

```
public void CreateOleDbParameter()
{
    OleDbParameter p1 = new OleDbParameter("Description", OleDbType.VarChar, 50);
    p1.IsNullable = true;
    p1.Direction = ParameterDirection.Output;
}
```

14.5 Das CommandBuilder-Objekt

Das manuelle Zuweisen der *Insert-*, *Update-* und *DeleteCommand*-Eigenschaften des *DataAdapter*s ist mitunter eine ziemlich aufwendige Angelegenheit. Falls Ihr *DataAdapter* nur auf einer einzigen Datenbanktabelle aufsetzt, können Sie vorteilhaft den *CommandBuilder* zum automatischen Generieren der *Command*-Objekte verwenden.

14.5.1 Erzeugen

Voraussetzung für den Einsatz eines *CommandBuilder*-Objekts ist, dass dem *DataAdapter* vorher die *SelectCommand*-Eigenschaft zugewiesen wurde. Eine einzige Anweisung reicht dann aus, um einen *CommandBuilder* mit einem *DataAdapter* zu verkoppeln:

Syntax:

```
CommandBuilder cmdBuilder = new CommandBuilder(DataAdapter da);
```

Der *CommandBuilder* verfolgt nun argwöhnisch alle am *DataSet* vorgenommenen Änderungen und generiert die erforderlichen Queries bzw. *Command*-Objekte selbstständig im Hintergrund.

14.5.2 Anwenden

Die *Update*-Methode des *DataAdapter*s würde im folgenden Beispiel ohne *OleDbCommandBuilder* fehlschlagen.

Beispiel 14.26: Aktualisieren der Kunden-Tabelle aus der *Northwind*-Datenbank

C#

Beim Instanziieren erhält der *DataAdapter* automatisch auch seine *SelectCommand*-Eigenschaft, sodass diese nicht explizit zugewiesen werden muss:

```
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Customers", conn);
```

Ein *SqlCommandBuilder* wird mit dem *SqlDataAdapter* verbunden:

```
SqlCommandBuilder cmdB = new SqlCommandBuilder(da);
```

Ein *DataSet* wird mit den Daten gefüllt:

```
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "Kunden");
...
```

Nachdem die Daten geändert wurden, werden die Änderungen in die Datenbank zurückgeschrieben:

```
da.Update(ds, "Kunden");
```

Beim Aufruf von *Dispose* wird die Zuordnung von *CommandBuilder* zu *DataAdapter* aufgehoben und die generierten Befehle werden nicht mehr verwendet.

HINWEIS: Ein Beispiel für den Einsatz des *CommandBuilder*s finden Sie im Praxisbeispiel in Abschnitt 14.9.4, Die Datenbank aktualisieren.

14.6 Das DataReader-Objekt

Häufig genügt ein Lesezugriff auf die Datensätze. Dabei müssen im Frontend meist nur einige für die Listendarstellung benötigte Komponenten gefüllt bzw. aktualisiert werden (*ListBox*, *ComboBox*, *ListView*, *TreeView*, *DataGridView* usw.).

Im .NET-Framework gibt es für diesen Zweck den *DataReader*. Diese Klasse ist für einen einmaligen ReadOnly-Hochgeschwindigkeitszugriff auf eine Datensatzgruppe optimiert und ähnelt anderen *Reader*-Objekten wie *TextReader*, *StreamReader* und *XmlReader*. In Abhängigkeit vom verwendeten .NET-Datenprovider gibt es auch hier unterschiedliche Typen (*SqlDataReader*, *OleDbDataReader*).

14.6.1 DataReader erzeugen

Einen *DataReader* erzeugt man in der Regel nicht mit dem *new*-Konstruktor, sondern mit der *ExecuteReader*-Methode des zugrunde liegenden *Command*-Objekts:

Syntax:

```
DataReader dr = cmd.ExecuteReader();
```

Mitunter wird auch dem *Execute*-Konstruktor als Argument der Wert *CloseConnection* (aus der *CommandBehavior*-Enumeration) übergeben. Damit ist gewährleistet, dass die Verbindung automatisch nach dem Durchlauf des *DataReader*s geschlossen wird.

Beispiel 14.27: Ein *DataReader* wird instanziiert.

C#

```
DataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
```

14.6.2 Daten lesen

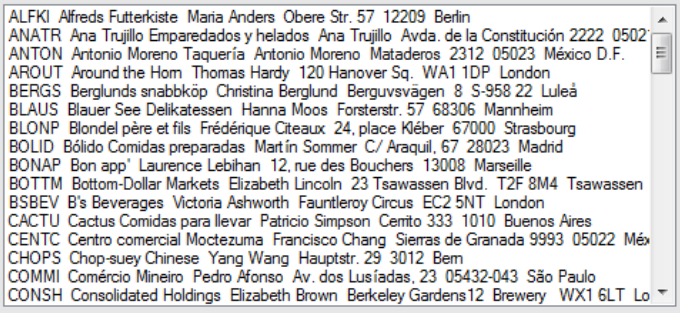
Das Auslesen der Informationen innerhalb einer Schleife ist typisch für die Arbeit mit dem *DataReader*.

Beispiel 14.28: Die Kundentabelle aus der *Northwind*-Datenbank wird zeilenweise ausgelesen.

C#

```
const string SQL = "SELECT * FROM Customers ORDER BY CustomerId";
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand cmd = new SqlCommand(SQL, conn);
SqlDataReader dr;
conn.Open();
dr = cmd.ExecuteReader();
string str;
string tab = "  ";
while (dr.Read())
{
  str = dr["CustomerId"] + tab;
  str += dr["CompanyName"] + tab;
  str += dr["ContaktName"] + tab;
  str += dr["Street"] + tab;
  str += dr["PostalCode"] + tab;
  str += dr["City"];
  listBox1.Items.Add(str);
}
dr.Close();
conn.Close();
```

Ergebnis



HINWEIS: Es ist wichtig, dass Sie den *DataReader* so schnell wie möglich nach dem Auslesen der Daten wieder schließen, da sonst das *Connection*-Objekt blockiert ist!

14.6.3 Eigenschaften des DataReaders

Item-Eigenschaft

Diese Eigenschaft ermöglicht den Zugriff auf die aktuelle Spalte, der Rückgabewert ist vom *Object*-Datentyp (ähnlich der *Item*-Eigenschaft des *DataRow*-Objekts). Falls der Datentyp vorher bekannt ist, sollte man eine der *Get*-Methoden (siehe unten) für den Zugriff verwenden.

FieldCount-Eigenschaft

Diese Eigenschaft liefert die Gesamtanzahl der Datensätze.

IsClosed-Eigenschaft

Der Wert ist *true*, falls der *DataReader* geschlossen ist.

14.6.4 Methoden des DataReaders

Read-Methode

Damit wird das automatische Weiterbewegen zum nächsten Datensatz innerhalb der *while*-Schleife ermöglicht (Rückgabewert *true/false*).

GetValue- und GetValues-Methode

Während *GetValue* – ähnlich der *Item*-Eigenschaft – den Wert einer Spalte (basierend auf dem Spaltenindex) zurückgibt, nimmt *GetValues* ein Array entgegen, in welchem der *DataReader* den Inhalt der aktuellen Zeile ablegt. Mit *GetValues* wird beste Performance erreicht.

GetOrdinal- und ähnliche Methoden

Eine Vielzahl von *Get...*-Methoden ermöglichen ein Konvertieren der gelesenen Werte in fast jeden Datentyp.

Beispiel 14.29: Ein Datumswert aus der *Employee*-Tabelle wird ausgelesen.

C#

```
DateTime aDate;
aDate = dr.GetDateTime(dr.GetOrdinal("BirthDate"));
```

Bemerkungen

Nach dem Laden der Daten mit dem *DataReader* kopiert man in der Regel die Datensätze zeilenweise in Objekte um (Entity Framework erledigt das automatisch für Sie).

Beispiel 14.30: Einlesen von Kundendaten

C#

```
Kunde kd = new Kunde();
kd.ID = Convert.ToInt64(dr["CustomerId"]);
k.Ort = Convert.ToString(dr["City"]);
```

Das ist insbesondere bei vielen Tabellenspalten sehr arbeitsaufwendig. Außerdem liegt die meiste Arbeit noch vor Ihnen, denn in der Regel wollen Sie die Daten nicht nur lesen, sondern Sie wollen auch Änderungen speichern.

14.7 Das DataAdapter-Objekt

Datenadapter werden in einer Art „Brückenfunktion“ dazu genutzt, Daten mittels SQL-Anweisungen aus Datenquellen in *DataSet*s zu transportieren bzw. um Datenquellen mit den geänderten Inhalten von *DataSet*s zu aktualisieren. Das *DataAdapter*-Objekt verwendet das *Connection*-Objekt des jeweiligen .NET-Datenproviders, um eine Verbindung zu einer Datenquelle herzustellen, und ist außerdem auf verschiedene *Command*-Objekte angewiesen.

Hin- und Rücktransport der Daten zwischen Datenquelle und *DataSet* werden mit der *Fill*- und *Update*-Methode des *DataAdapter*s realisiert. Beide lösen die entsprechenden SQL-Anweisungen aufgrund der dem *DataAdapter* übergebenen *Command*-Objekte aus.

14.7.1 DataAdapter erzeugen

Mehrere überladene Konstruktoren stellen den Newcomer vor die Qual der Wahl.

Konstruktor mit SELECT-String und Connection-Objekt

Im einfachsten Fall kommt man sogar ohne *Command*-Objekt aus, es genügt, dem Konstruktor des *DataAdapter*-Objekts eine SELECT-Anweisung und das *Connection*-Objekt als Parameter zu übergeben:

Syntax:

```
DataAdapter da = new DataAdapter(string selectStr, Connection conn);
```

Beispiel 14.31: Ein *DataAdapter* füllt ein *DataSet* mit Datensätzen aus der *Northwind*-Datenbank.

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Customers
                    WHERE City = 'Paris'", conn);
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "PariserKunden");
conn.Close();
```

Konstruktor mit SelectCommand-Objekt

Eine weitere Möglichkeit ist die Verwendung eines Konstruktors, dem ein *Command*-Objekt (SELECT-Befehl) zu übergeben ist:

Syntax:

```
DataAdapter da = new DataAdapter(Command selectCommand);
```

Beispiel 14.32: Das Vorgängerbeispiel wird mit einem *Command*-Objekt realisiert.

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand cmd = new SqlCommand("SELECT CompanyName FROM Customers
                                 WHERE City = 'Paris'");
cmd.Connection = conn;
SqlDataAdapter da = new SqlDataAdapter(cmd);
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "PariserKunden");
conn.Close();
```

14.7.2 Command-Eigenschaften

Ein *DataAdapter* benötigt für die komplette Zusammenarbeit mit der Datenquelle vier verschiedene *Command*-Objekte, die als Eigenschaften zugewiesen werden:

- *SelectCommand* zur Abfrage
- *UpdateCommand* zur Aktualisierung
- *InsertCommand* zum Einfügen
- *DeleteCommand* zum Löschen

Beispiel 14.33: Realisierung der Vorgängerbeispiele mittels *SelectCommand*-Eigenschaft

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand cmd = new SqlCommand("SELECT CompanyName FROM Customers
                                 WHERE City = 'Paris'");
cmd.Connection = conn;
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = cmd;
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "PariserKunden");
conn.Close();
```

Die *SelectCommand*-Eigenschaft muss gesetzt werden, **bevor** die *Fill*-Methode des *DataAdapter*s aufgerufen wird.

14.7.3 Fill-Methode

Die relativ unkomplizierte *Fill*-Methode des *DataAdapter*s hatten Sie bereits in zahlreichen Beispielen kennengelernt. Hier noch einmal die am häufigsten benutzte Aufrufvariante:

Syntax:

```
DataAdapter da.Fill(DataSet ds, string tblName)
```

Beispiel 14.34: Ein *DataSet* wird mit der Kundentabelle aus der *Northwind*-Datenbank gefüllt. Im *DataSet* sollen die Namen aller Firmen aus Paris geändert werden in „Pariser Firma“.

C#

```
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Customers", conn);
DataSet ds = new DataSet();
da.Fill(ds, "Kunden");
```

Das Arbeiten mit den Daten im *DataSet*:

```
DataTable dt = ds.Tables["Kunden"];
foreach (DataRow cRow in dt.Rows)
// alle Zeilen der DataTable durchlaufen
{
    if (cRow["City"].ToString() == "Paris")
    {
       cRow["City"] = "Pariser Firma";
    }
}
...
```

Das Beispiel wird im folgenden Abschnitt fortgesetzt!

Begrenzen der Datenmenge

Geht es nur um die Übertragung kleinerer Datenmengen, so ist die bislang praktizierte Vorgehensweise problemlos, nicht aber wenn es sich um Hunderte von Datensätzen handelt.

Abhilfe schafft eine (überladene) Version der *Fill*-Methode, die die Anzahl der zu transportierenden Datensätze begrenzt:

Syntax:

```
int z  = DataAdapter da.Fill(DataSet ds, int start, int anzahl, string tblName);
```

start = Nummer der Startzeile

anzahl = Anzahl der abzurufenden Datensätze

z = Anzahl der tatsächlich zurückgegebenen Datensätze

Beispiel 14.35: Ab Zeile 100 werden 50 Zeilen aus der Datenbank abgerufen und in die „Kunden"-Tabelle gefüllt.

C#

```
int z = da.Fill(ds, 100, 50, "Kunden");
```

14.7.4 Update-Methode

Irgendwann einmal müssen die im *DataSet* vorgenommenen Änderungen in die Datenquelle zurückgeschrieben werden. Zu diesem Zweck wird (kurzzeitig) eine Verbindung zur Datenbank aufgebaut. Genauso wie beim Füllen spielt auch hier ein *DataAdapter*-Objekt die Vermittlerrolle, wobei dessen *Update*-Methode gewissermaßen das Pendant zur *Fill*-Methode ist und zum Zurückschreiben der im *DataSet* vorgenommenen Änderungen in die Datenquelle dient.

Genauso wie die *Fill*-Methode benötigt die *Update*-Methode als Parameter die Instanz eines *DataSet*s und (optional) den Namen der *DataTable*.

Syntax:

```
DataAdapter da.Update(DataSet ds, string tblName);
```

Bei der *Update*-Methode läuft es nicht ganz so einfach ab wie bei der *Fill*-Methode, da ein *DataSet* völlig autark existiert und nur gelegentlich mit der Datenbank verbunden wird.

Beispiel 14.36: (*Fortsetzung*)

C#

Ziel ist das Zurückschreiben der in der Spalte „CompanyName" (und nur dort!) vorgenommenen Änderungen in die Datenquelle. Grundlage ist eine UPDATE-Anweisung mit zwei Parametern (die @ sind die Platzhalter):

```
...
SqlCommand cmd = new SqlCommand("UPDATE Customers
         SET CompanyName = @name WHERE CustomerId = @id", conn);
```

Der *Add*-Methode werden Parametername, Datentyp und Spaltenbreite übergeben:

```
cmd.Parameters.Add("@name", SqlDbType.NVarChar, 30);
```

Für die Schlüsselspalte:

```
SqlParameter prm = cmd.SqlParameters.Add("@id", SqlDbType.NVarChar);
prm.SourceColumn = "CustomerId";
```

Der ursprüngliche Wert (beim Füllen des *DataSet*s) ist maßgebend:

```
da.UpdateCommand = cmd;
da.Update(ds, "Kunden");
```

HINWEIS: Der Kern der Aktualisierungslogik liegt in der WHERE-Bedingung der UPDATE-Anweisung. Der Datensatz wird nur dann aktualisiert, wenn der Wert der Schlüsselspalte, mit dem er geladen wurde, noch vorhanden ist.

- Durch Einsatz eines *CommandBuilder*-Objekts kann das manuelle Erstellen der *UpdateCommand*-, *InsertCommand*- und *DeleteCommand*-Eigenschaften automatisiert werden.
- Ein komplettes Beispiel finden Sie im Praxisbeispiel in Abschnitt 14.9.4, Die Datenbank aktualisieren.

14.8 Entity Framework

14.8.1 Überblick

Das Entity Framework ist Microsofts ORM (Objektrelationaler Mapper), der den Zugriff auf die Datenbank über ein Objektmodell gewährleistet. Für jede Tabelle der Datenbank, auf die der Zugriff gewährt werden soll, wird im Objektmodell eine Klasse definiert (eine sogenannte Entität). Jede Spalte wird dabei durch eine Property dargestellt. Methoden wird man in den Entitätsklassen nicht finden.

Da wir heutzutage zumeist mit relationalen Datenbanken arbeiten, ist auch noch interessant, wie Relationen im Objektmodell dargestellt werden, da es ja keine Fremdschlüsselbeziehungen bei Objekten gibt.

Dies wird in EF dadurch realisiert, dass das übergeordnete Objekt eine zusätzliche Property als Liste des untergeordneten Objekts enthält, und auch das untergeordnete Objekt enthält als zusätzliche Property einen Verweis auf das übergeordnete Objekt. Diese Properties werden auch als *NavigationProperties* bezeichnet.

Diese NavigationProperties werden jedoch erst geladen, wenn ein Zugriff auf diese Eigenschaft erfolgt. Dieser Vorgang nennt sich *LazyLoading*. Würden die abhängigen Objekte auch sofort geladen, würde sehr schnell die Gefahr bestehen, dass man beim Lesen eines Datensatzes sich die halbe Datenbank in den Speicher zieht.

HINWEIS: Das Entity Framework (aktuell in der Version 6) ist nicht Bestandteil des .NET Frameworks. Das Entity Framework muss über den Nuget-Paketmanager zu Ihrem Projekt hinzugefügt werden.

Der gesamte Datenzugriff wird dann über eine Klasse, die von *DbContext* erbt, erledigt. In Ihrem abgeleiteten Context definieren Sie lediglich, welche Objekte bearbeitet werden und auf welche Tabellen somit zugegriffen wird. Die gesamte Basisfunktionalität zum lesenden und schreibenden Zugriff steckt bereits in der Basisklasse. Sie müssen somit keine SQL-Befehle mehr in Ihrer Anwendung definieren und laden und speichern Ihre Daten nur mit bereits bestehenden Methoden der von *DbContext* abgeleiteten Klasse.

Zum Erzeugen des Objektmodells gibt es prinzipiell zwei unterschiedliche Technologien, die wir uns in diesem Einführungskapitel nun ansehen wollen.

Zum einen können Sie von einer bestehenden Datenbank das Objektmodell automatisch generieren lassen (*DatabaseFirst*). Bei Änderungen am Datenbankmodell können Sie dann Ihr Modell immer wieder aktualisieren. Sollten Tabellenspalten gelöscht oder umbenannt oder ganze Tabellen gelöscht worden sein, dann erkennen Sie den jetzt fehlerhaften Zugriff+ sofort durch Compilerfehler. Ohne einen Objektrelationalen Mapper ist das ein nicht zu unterschätzender Aufwand, da ja sämtliche SQL-Anweisungen in Zeichenketten stehen, die der Compiler natürlich nicht analysieren kann.

DatabaseFirst gab es seit den ersten Versionen von Entity Framework, jedoch wird momentan die zweite Zugriffstechnologie *CodeFirst* empfohlen. Zum einen ist *CodeFirst* wesentlich wartbarer und zum anderen wird *DatabaseFirst* in zukünftigen Versionen des Entity Framework Cores nicht mehr angeboten.

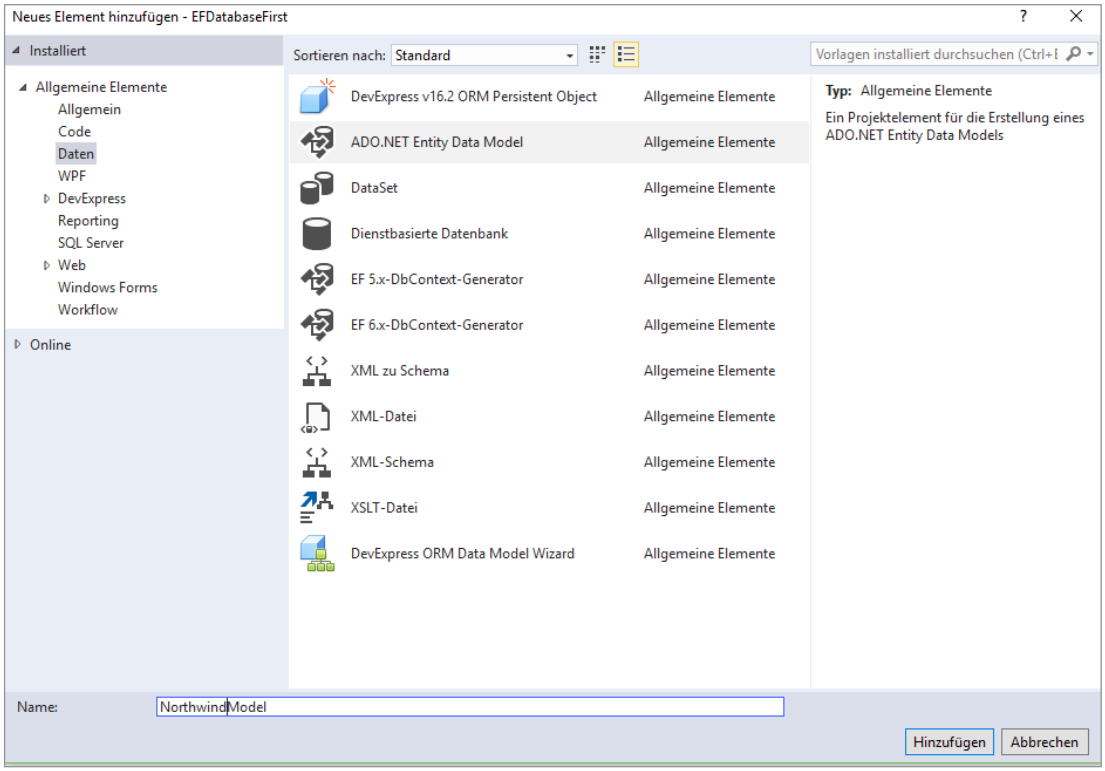
Bei *CodeFirst* definieren Sie das Objektmodell und aus diesen Klassendefinitionen können Sie dann automatisch die Datenbank erzeugen. Durch Migrationen können Sie außerdem definieren, dass Änderungen an Ihrem Objektmodell automatisch zu einer Anpassung der Datenstruktur führen.

HINWEIS: Es gibt auch die Möglichkeit, aus einer bestehenden Datenbank ein CodeFirst-Modell zu erzeugen. Sie erhalten dann eine 1:1-Abbildung des Modells. Sämtliche weiteren Änderungen der Struktur erfolgen dann aber über Ihre Entitätsklassen und durch Migrationen wird die Datenbankstruktur dann aktualisiert.

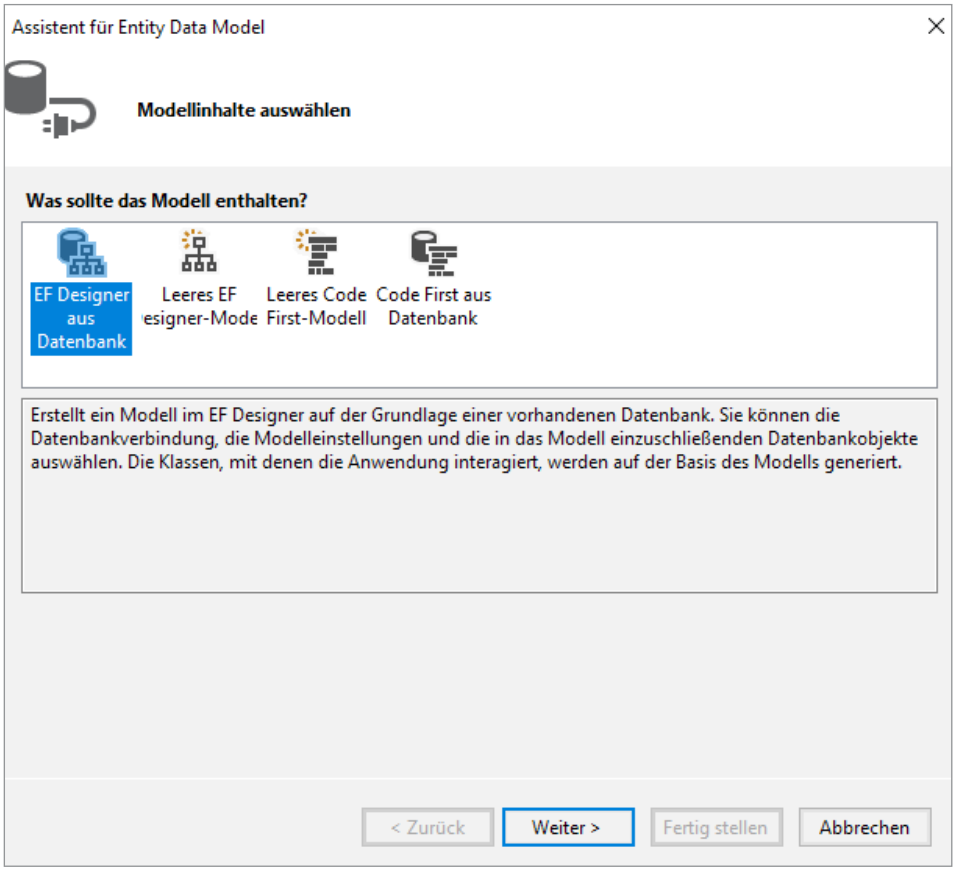
So, nun aber genug der Theorie. Schauen wir uns doch beide Sachen einfach mal praktisch an. Als Beispiel dient wieder unsere bereits bekannte *Northwind*-Datenbank.

14.8.2 DatabaseFirst

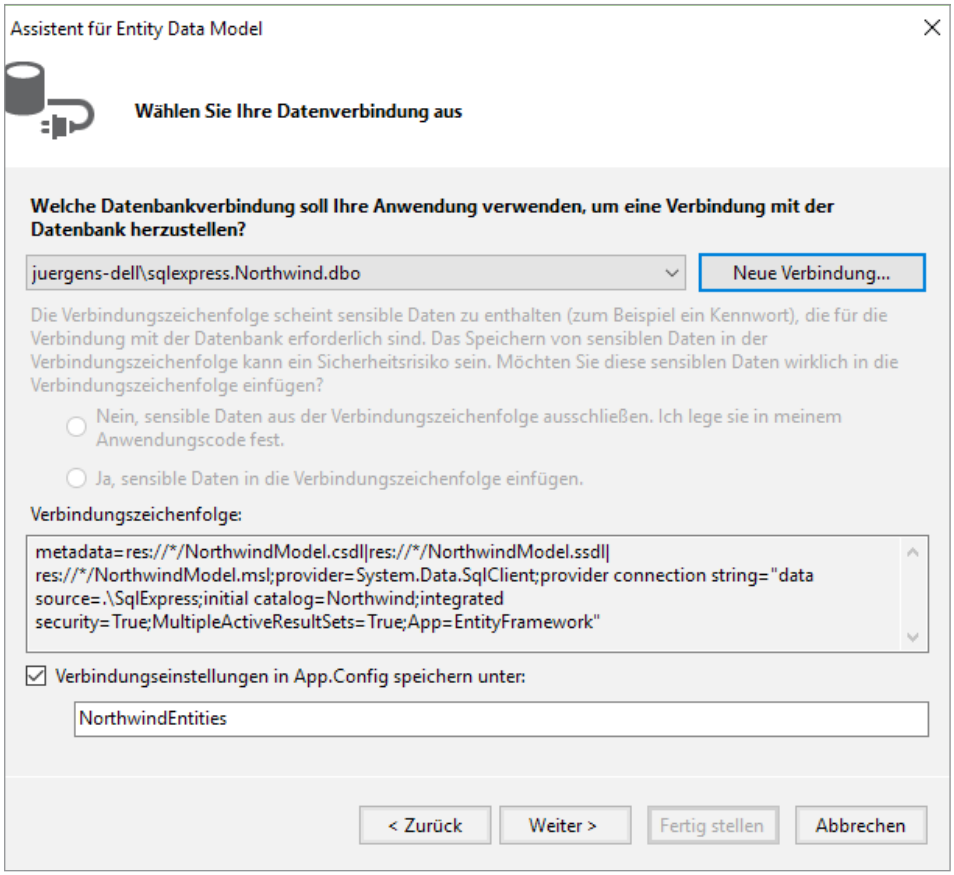
Zuerst müssen wir einem beliebigen Projekttyp (ich werde eine WindowsFormsApp als Beispiel definieren) ein neues Element vom Typ *ADO.NET Entity Data Model* zum Projekt hinzufügen und geben diesem den Namen *NorthwindModel*.



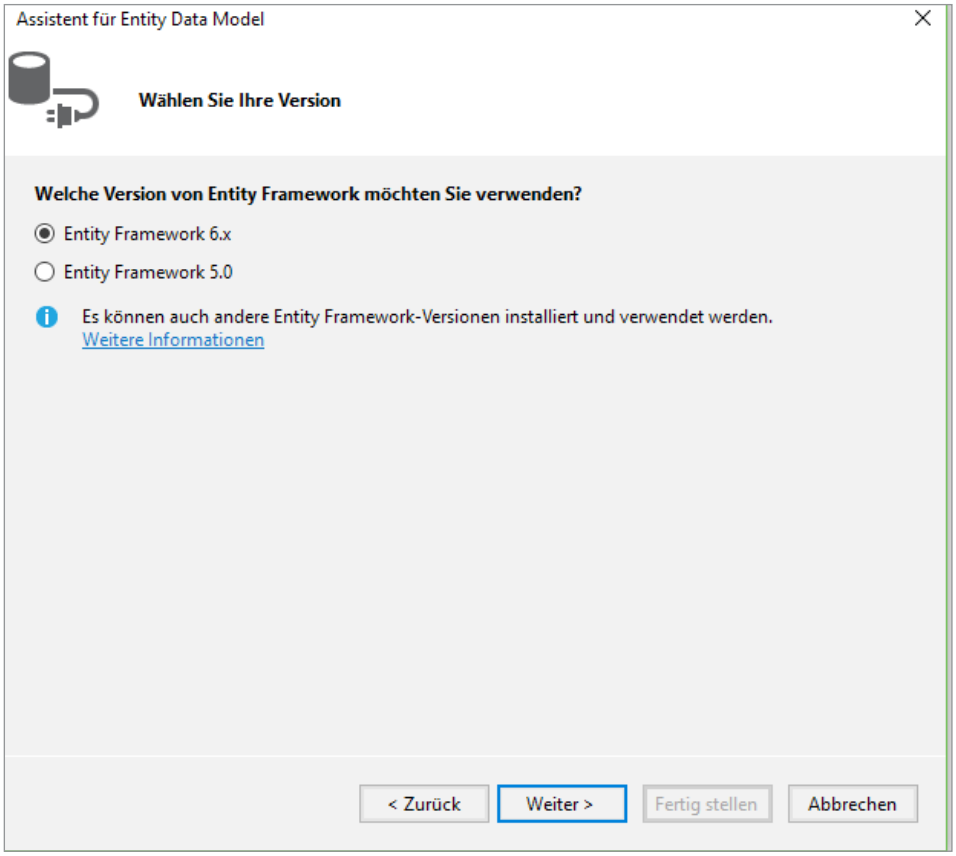
Im nächsten Schritt wählen Sie aus, ob Sie *DatabaseFirst* oder *CodeFirst* verwenden wollen:



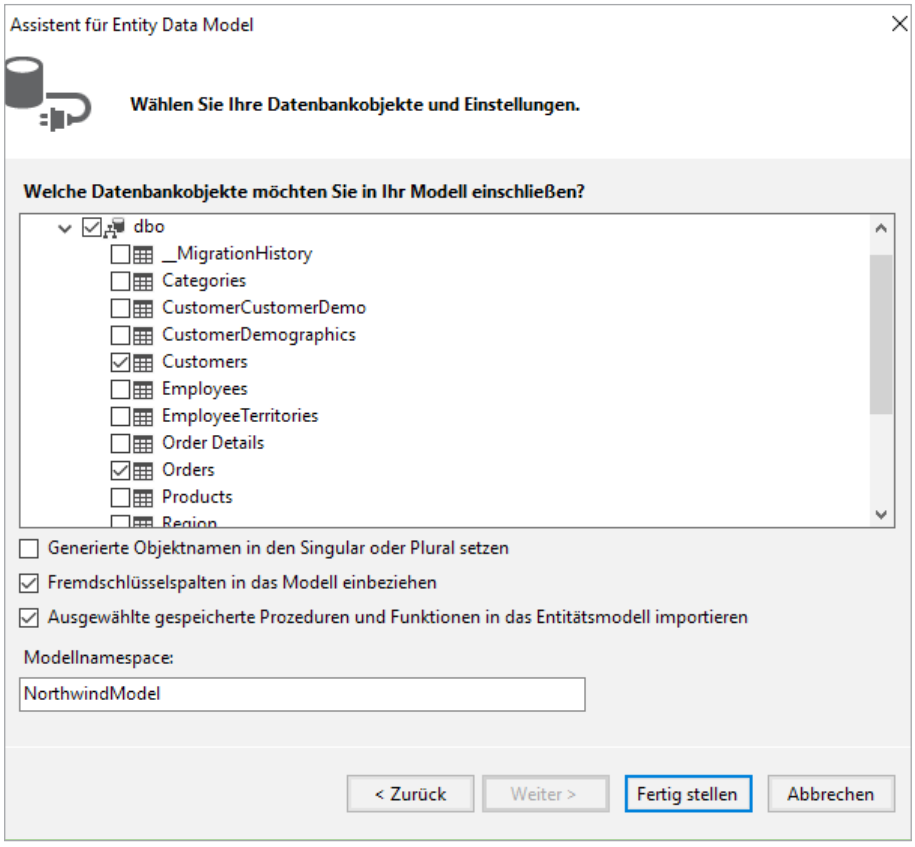
Im folgenden Dialog wird dann die Datenbank ausgewählt, aus deren Struktur das Modell erzeugt wird:



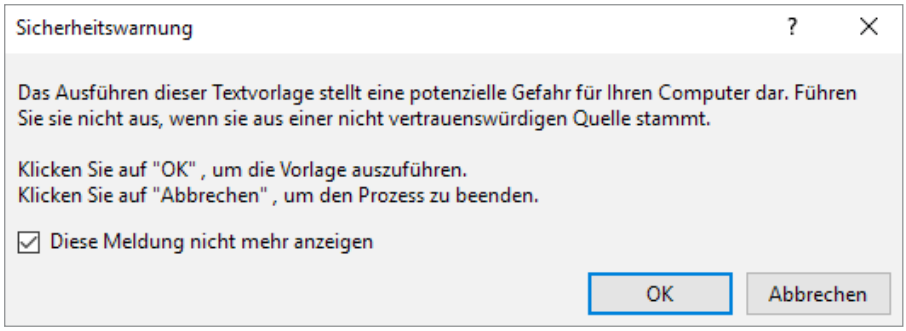
Sollten Sie das Entity Framework noch nicht per Nuget-Paket zu Ihrem Projekt hinzugefügt haben, erledigt das der Assistent für Sie. Wählen Sie einfach aus, welche Version von Entity Framework verwendet werden soll (ich würde immer die aktuellste empfehlen):



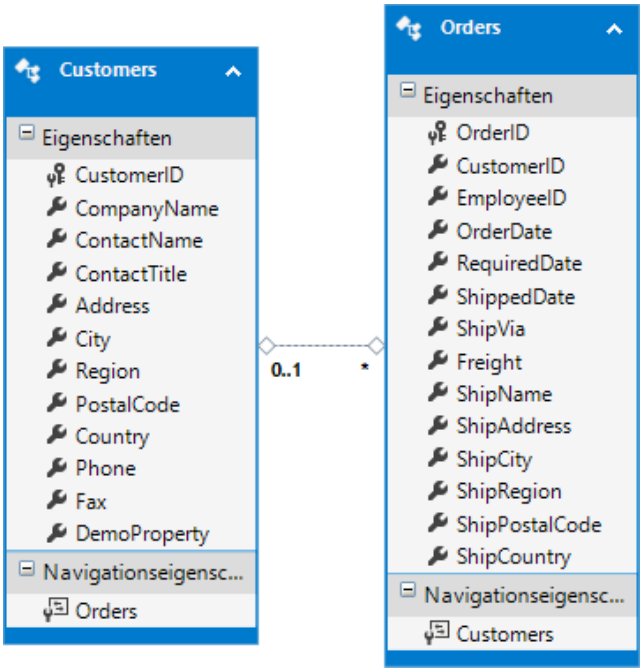
Im abschließenden Schritt definieren Sie nur noch, welche Datenbankobjekte in Ihr Objektmodell integriert werden sollen. Der Übersichlichkeit halber wähle ich jetzt nur die beiden Tabellen *Customers* und *Orders* aus:



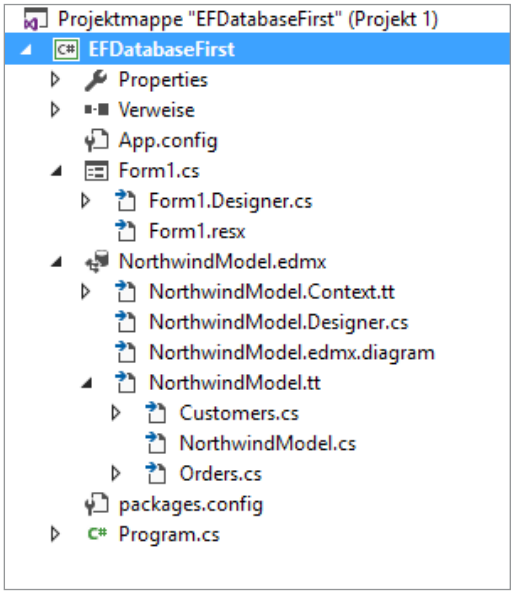
Durch einen Klick auf den Button *Fertig stellen* wird das Modell nun komplett erzeugt. Da jetzt im Hintergrund Code erzeugt wird, können Sie die aufpoppende Sicherheitswarnung mit *OK* bestätigen und auch die weitere Anzeige der Meldung unterdrücken:



Das Ergebnis wird dann in einer relationalen Ansicht dargestellt:



Tatsächlich wurde jedoch eine sogenannte *Northwind.edmx*-Datei erzeugt und wenn man diese im Projektmappenexplorer aufklappt, kann man auch die zugehörigen erzeugten Klassen sehen:



Schauen wir uns das Ergebnis mal etwas genauer an.

Die übergeordneten Elemente, die Sie sehen, sind tt-Dateien. tt (in der Literatur liest man auch öfter T4) sind sogenannte TextTransformationToolkitTemplate-Dateien. Darin ist die Definition enthalten, wie die Codedateien zu generieren sind. Sie können diese Dateien, natürlich mit ein bisschen Hintergrundwissen über T4) auch editieren, wenn Sie ein paar Sachen anders machen möchten.

Ein direktes Manipulieren der Codedateien ist leider nicht möglich (möglich schon, aber nicht sinnvoll), da beim nächsten Generieren alle Ihre Änderungen überschrieben werden.

So betrachten wir uns nun die erzeugten Codedateien.

Beispiel 14.37: Generierte Codedatei NorthwindModel.Context.cs

C#

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;

public partial class NorthwindEntities : DbContext
{
    public NorthwindEntities()
        : base("name=NorthwindEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public virtual DbSet<Customers> Customers { get; set; }
    public virtual DbSet<Orders> Orders { get; set; }
}
```

Die Klasse *NorthwindEntities* erbt dabei von der Klasse *DbContext*. Über den Konstruktor wird der *ConnectionString* aus der Konfigurationsdatei ausgelesen (der Assistent hat für uns einen ConnectionString mit dem Namen *NorthwindEntities* in die *app.config* eingetragen).

Zu guter Letzt wurden noch die zwei Properties *Customers* und *Orders* als generische *DbSet*s definiert, über die wir auf die Tabellen zugreifen können.

Beispiel 14.38: Generierte Codedatei Customers.cs.

C#

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

    public partial class Customers
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
                  "CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Customers()
        {
            this.Orders = new HashSet<Orders>();
```

```
        }

        public string CustomerID { get; set; }
        public string CompanyName { get; set; }
        public string ContactName { get; set; }
        public string ContactTitle { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Region { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
        public string Phone { get; set; }
        public string Fax { get; set; }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
                            "CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Orders> Orders { get; set; }
    }
```

In der Klasse *Customers* sind nur noch Properties definiert. Die Namen und Datentypen passen dabei zu den Spalten der Tabelle *Customers* in der *Northwind*-Datenbank.

Einzig die Property *Orders* ist als eine zusätzliche Liste definiert (und bildet somit die Fremdschlüsselbeziehung zwischen *Customers* und *Orders* ab). Durch die Definition als *virtual* wird übrigens für diese Liste *LazyLoading* aktiviert.

Jetzt stellt sich noch die Frage, wie man mit diesen Objekten arbeitet. Dazu wollen wir die Daten in einem *DataGrid* anzeigen und auch editieren lassen (auch wenn ich kein Freund davon bin, Daten in einem Grid zu editieren). Mit einem Speichern-Button sollen die Änderungen dann in die Datenbank zurückgeschrieben werden.

Beispiel 14.39: Lesen und Speichern der Daten

C#

```
public partial class Form1 : Form
{
    private NorthwindEntities context;
    public Form1()
    {
       InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
       context = new NorthwindEntities();
       dataGridView1.DataSource = context.Customers.ToList();
    }

    private void btnSpeichern_Click(object sender, EventArgs e)
    {
        context.SaveChanges();
    }
}
```

Innerhalb des Formulars definieren wir eine Variable für unseren Context. Dieser wird im Konstruktor instanziiert und mit der Anweisung *context.Customers.ToList()* werden die Daten aus der Datenbank geladen. Die geladenen Daten werden als Datenquelle der *DataGridView* zugewiesen.

Unter dem Click-Event des Speicherbuttons folgt nur eine Anweisung: *context.SaveChanges()*. Das Entity Framework protokolliert alle Ihre Änderungen und schickt die entsprechenden Anweisungen als SQL-Befehle innerhalb einer Transaktion an die Datenbank.

Sie müssen natürlich nicht alle Daten aus der Tabelle laden, mit einer *Linq*-Abfrage können Sie die Daten selbstverständlich auch filtern.

Wenn Sie zum Beispiel nur Kunden aus Deutschland haben wollen, dann führen Sie folgende Abfrage aus:

Beispiel 14.40: Laden von ausschließlich deutschen Kunden

C#

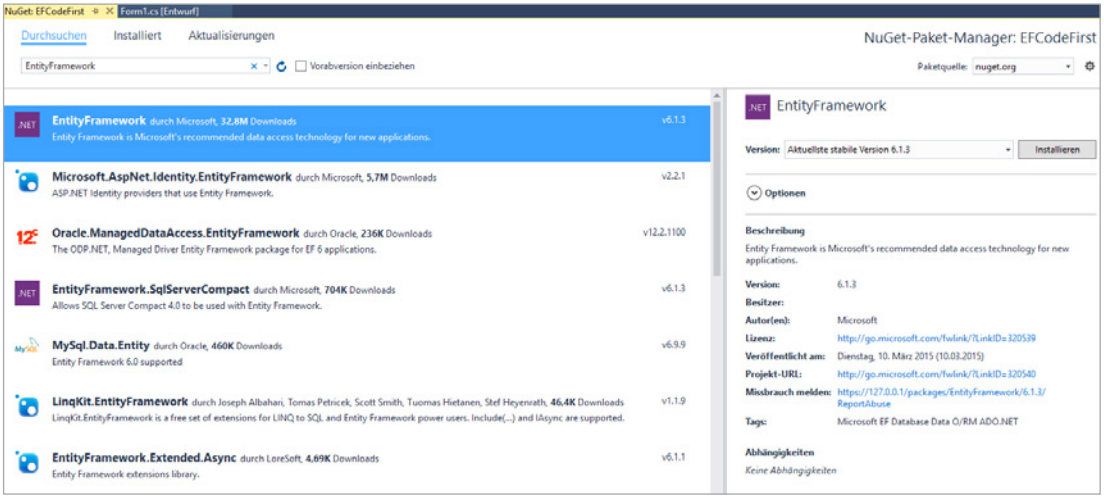
```
dataGridView1.DataSource =
    context.Customers.Where(c=>c.Country=="Germany").ToList();
```

14.8.3 CodeFirst

Bei *CodeFirst* ist der einzige Unterschied die Erstellung des Modells. Der Einsatz der Klassen und der Datenzugriff sind danach anlog wie bei *DatabaseFirst*.

Bei *CodeFirst* definieren Sie einfach Ihre Entitätsklassen und die Contextklasse, fertig. Ein *edmx*-Modell gibt es bei *CodeFirst* nicht, was die Wartbarkeit erheblich erhöht.

Zuerst müssen wir aber unserem Projekt das Entity Framework NuGet-Paket hinzufügen.



Im zweiten Schritt definieren wir (der Einfachheit halber) nur eine Entitätsklasse mit einigen Properties.

Beispiel 14.41: Entitätsklasse Person.cs

C#

```
[Table("Personen")]
public class Person
{
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    public DateTime Geburtstag { get; set; }
    [StringLength(20)]
    public string Telefon { get; set; }
    public int? Groesse { get; set; }
}
```

In dieser Klasse wird eine Property *Id* vom Datentyp *int* definiert. Es ist eine Konvention, dass eine Property *Id* automatisch der Primärschlüssel dieser Tabelle wird. Da es sich beim Datentyp um einen Integer handelt, bedeutet dies auch, dass es eine *Identity*-Spalte ist.

Die Property *Name* wird mit dem Attribut *Required* dekoriert. Dies bedeutet, dass es sich um eine *NotNull*-Spalte handelt. Über Attribute wird einiges gesteuert (mehr dazu in der nachfolgenden Tabelle). Die Property *Telefon* erhält das Attribut *StringLength*, um die maximale Länge des Strings in der Datenbank zu definieren. Die Eigenschaft *Groesse* letztendlich ist als *int?* definiert, somit wird diese Spalte in der Datenbank als *null*-Spalte angelegt.

Attribute aus System.ComponentModel.DataAnnotations

In der folgenden Tabelle sind die wichtigsten Attribute zur Definition bei CodeFirst aufgeführt:

Attribut	Bedeutung
Key	Spalte wird als Primärschlüssel definiert
Table	Name der Tabelle, wenn nicht angegeben identisch mit Klassennamen
Column	Name der Spalte, wenn nicht angegeben identisch mit Propertyname; damit auch Angabe von alternativen Datentypen möglich
Index	Definition eines Index, auch mehrspaltig möglich
Required	Bei Strings *NotNull*-Spalte
DatabaseGenerated	Angabe wie der Primärschlüssel definiert wird (*Identity*, *Computed* oder *None*)
NotMapped	Property wird nicht der Datenbanktabelle hinzugefügt

In dieser Art und Weise können Sie jetzt beliebige weitere Entitätsklassen zu Ihrem Projekt hinzufügen.

Dann fehlt uns nur noch die Definition unserer Contextklasse.

Beispiel 14.42: Contextklasse PersonContext.cs

C#

```
public class PersonContext : DbContext
{
    public PersonContext():base("name=PersonContext")
    {
    }
    public virtual DbSet<Person> Personen { get; set; }
}
```

In dieser Klasse definieren wir, welche Klassen dem Context hinzugefügt werden (wir haben ja nur eine). *Personen* wird dabei als generisches *DbSet* vom Typ *Person* definiert.

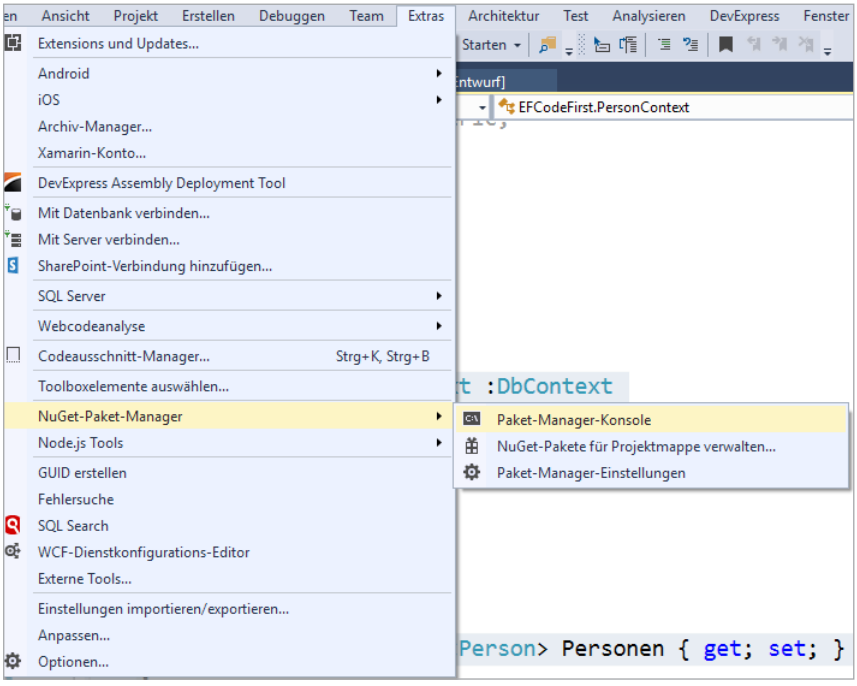
Über den Konstruktor definieren wir, dass der *Connectionstring* in der Konfigurationsdatei steht und *PersonContext* heißt.

Fügen Sie also der Datei *app.config* folgenden Abschnitt hinzu:

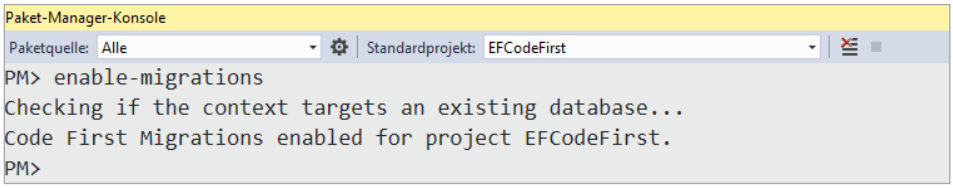
```
<connectionStrings>
    <add name="PersonContext" connectionString="data source=.\SqlExpress;  initial
catalog=PersonenDb;integrated security=True; MultipleActiveResultSets=True;
App=Entity Framework" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

Somit ist unser Modell fertig. Nun müssen wir nur noch mittels Migrationen die Datenbank anlegen und später auch aktualisieren.

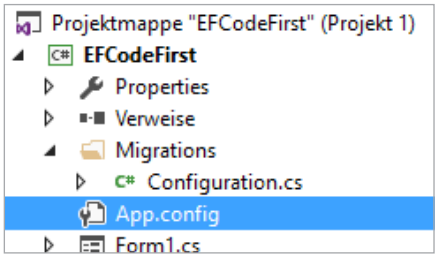
Öffnen Sie dafür die Konsole des Paketmanagers mittels dem Menübefehl *Extras/NuGet-Paket-Manager/Paket-Manager-Konsole*.



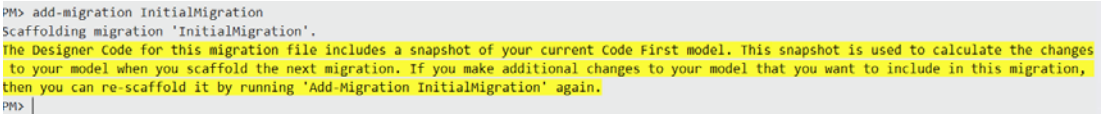
Geben Sie anschließend in der Konsole den Befehl *enable-migrations* ein.



Durch die Ausführung dieses Befehls wird im Projekt ein Ordner *Migrations* mit einer Klasse *Configuration.cs* angelegt. Innerhalb dieser Klasse gibt es eine *Seed*-Methode, in der Sie zum Beispiel Testdaten erzeugen können, die nach jeder Migration in die Datenbank geschrieben werden.



Um nun die erste Migration anzulegen, geben Sie in der Konsole den Befehl *add-migration InitialMigration* ein.



Dadurch wird automatisch eine neue Migrationsklasse angelegt. Der Name der Klasse hat noch einen führenden Zeitstempel.

Beispiel 14.43: Migrationsklasse InitialMigration.cs.

C#

```
using System.Data.Entity.Migrations;

public partial class InitialMigration : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Personen",
            c => new
                {
                    Id = c.Int(nullable: false, identity: true),
                    Name = c.String(nullable: false),
```

```
                    Geburtstag = c.DateTime(nullable: false),
                    Telefon = c.String(maxLength: 20),
                    Groesse = c.Int(),
                })
            .PrimaryKey(t => t.Id);
    }
    public override void Down()
    {
        DropTable("dbo.Personen");
    }
}
```

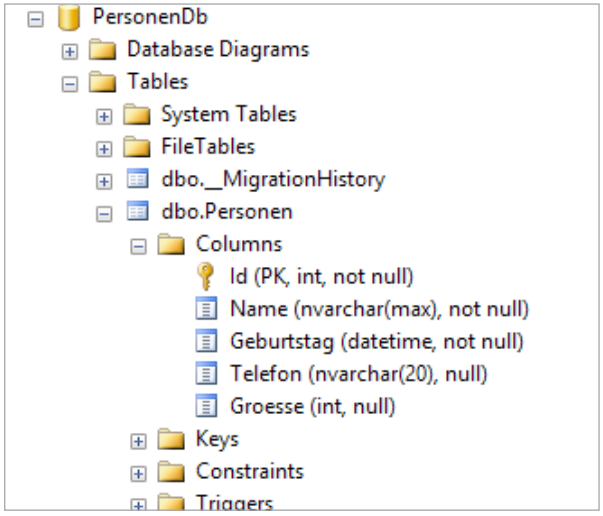
Die Routine *Up()* enthält den Code, der auf der Datenbank ausgeführt werden soll. *Down()* wird für einen Downgrade benötigt.

Um die Migration jetzt durchzuführen, geben Sie in der Konsole den Befehl *update-database* an.

```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying explicit migrations: [201708152118030_InitialMigration].
Applying explicit migration: 201708152118030_InitialMigration.
Running Seed method.
```

Sie können dies natürlich auch per Programmcode machen, sodass zum Beispiel bei jedem Programmstart die Datenbank aktualisiert wird. Sehr vorteilhaft bei lokalen Datenbanken, bei Serveranwendungen vielleicht nicht so optimal.

Nachdem der Befehl erfolgreich ausgeführt wurde, haben Sie eine neue Datenbank angelegt.



Die Tabelle *Personen* ist genauso angelegt worden, wie wir es definiert haben. Zusätzlich gibt es noch eine Tabelle *__MigrationHistory*. Über diese Tabelle weiß die Datenbank, welche Migrationen schon ausgeführt wurden, und führt diese somit nicht doppelt aus.

Alle weiteren Änderungen am Objektmodell werden analog mit zusätzlichen Migrationen erstellt.

Auch der Zugriff funktioniert analog, wie im DatabaseFirst-Beispiel gezeigt.

Dieses Kapitel soll nur einen ersten Überblick über das Entity Framework geben. Für detaillierte Informationen empfehlen wir die Lektüre von Büchern über das Entity Framework.

14.9 Praxisbeispiele

14.9.1 Wichtige ADO.NET-Objekte im Einsatz

Wer sich nicht nur blind auf die Hilfe von Assistenten verlassen möchte, sollte sich in der ADO.NET-Objekthierarchie ein wenig auskennen, damit er die Objekte bei Bedarf selbst per Code programmieren kann.

Die *Columns*- und *Rows*-Auflistungen zählen zu den wichtigsten Eigenschaften der *DataTable*-Klasse, weil sie den Zugriff auf sämtliche Spalten und Zeilen der Tabelle ermöglichen. Das vorliegende Beispiel soll das Zugriffsprinzip verdeutlichen, indem es uns den Inhalt der *Products*-Tabelle der *Northwind*-Datenbank anzeigt.

Oberfläche

Sie brauchen lediglich eine *ListBox* und einen *Button* zum Beenden (siehe Laufzeitansicht).

Quellcode

```
using System.Data.SqlClient;
...
```

Alles beginnt mit der Festlegung der Verbindungszeichenfolge (*ConnectionString*) zur *Northwind*-Datenbank.

```
    private void Form1_Load(object sender, System.EventArgs e)
    {
        SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                    Initial Catalog=Northwind;Integrated Security=true");
        string cmdStr =
           "SELECT ProductId  As ArtikelNr, ProductName as Artikelname,
            QuantityPerUnit As Liefereinheit, UnitPrice As Einzelpreis,
            ReorderLevel As Mindestbestand FROM Products";
        SqlCommand cmd = new SqlCommand(cmdStr, conn);
```

HINWEIS: Im obigen Select (sowie auch in allen folgenden) wurden Alias-Namen für die Spalten vergeben, um bei den Beschriftungen deutschsprachige Spaltennamen auszugeben.

Nun geht es um das Füllen des *DataSets* mithilfe des *DataAdapter*:

```
        SqlDataAdapter da = new SqlDataAdapter(cmd);
        DataSet ds = new DataSet();
        conn.Open();
        da.Fill(ds, "ArtikelListe");
        conn.Close();
```

Die Datenbankverbindung ist ab jetzt wieder getrennt und der Benutzer arbeitet mit dem abgekoppelten *DataSet* quasi wie mit einer Minidatenbank:

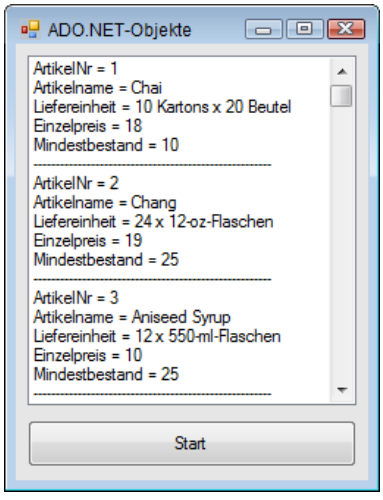
```
        DataTable dt = ds.Tables[«ArtikelListe»];
```

Nachdem je eine Zeilen- und Spaltenvariable definiert sind, sorgen zwei ineinander verschachtelte *foreach*-Schleifen für den Durchlauf der Auflistungen:

```
        foreach (DataRow cRow in dt.Rows)
        {
            foreach (DataColumn cCol in dt.Columns)
                listBox1.Items.Add($"{cCol.ColumnName} =  {cRow[cCol.Ordinal]}");
           listBox1.Items.Add("-----------------------------------------------");
        }
    }
```

Test

Sofort nach Programmstart erscheint der Inhalt der *Artikel*-Tabelle in der *ListBox*.



Bemerkungen

- Am Quellcode können Sie den typischen Ablauf eines Datenbankzugriffs studieren: Verbindung öffnen, Daten übertragen, Verbindung schließen.
- Beim Durchlaufen der Datensätze werden Sie die vom altvertrauten ADO-*Recordset*-Objekt her bekannten Methoden wie *MoveFirst*, *MoveNext* etc. vergeblich suchen. Dafür besteht unter ADO.NET keinerlei Notwendigkeit mehr, da alle Datensätze im *DataSet* quasi wie in einem Array gespeichert sind und ein sofortiger (indizierter) Zugriff möglich ist, ohne dass man sich erst mühsam „hinbewegen“ muss.

14.9.2 Eine Aktionsabfrage ausführen

Wir wollen an die *Northwind*-Beispieldatenbank folgendes SQL-Statement absetzen:

```
UPDATE Customers SET CompanyName = 'Londoner Firma' WHERE City = 'London'
```

Das vorliegende Beispiel zeigt, wie Sie dazu die *ExecuteNonQuery*-Methode des *Command*-Objekts verwenden können.

Oberfläche

Sie brauchen ein *DataGridView*, zwei *TextBox*en, zwei *Buttons* und einige *Labels* (siehe Laufzeitansicht). Beide *TextBox*en sollen dazu dienen, dass Sie die Einträge für den Firmennamen und den Ort zur Laufzeit verändern können.

Quellcode

Für das Ausführen des Beispiels wären eigentlich ein *Connection*- und ein *Command*-Objekt völlig ausreichend. Da wir uns aber auch von der Wirkung des UPDATE-Befehls überzeugen wollen, müssen wir einigen zusätzlichen Aufwand für die Anzeige betreiben: Das *DataGridView* benötigt ein *DataSet* als Datenquelle, welches wiederum von einem *DataAdapter* gefüllt wird.

```
using System.Data.SqlClient;
public partial class Form1 : Form
{
    SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                    Initial Catalog=Northwind;Integrated Security=true");
    DataSet ds = new DataSet();
    SqlCommand cmd = new SqlCommand();
```

Aktionsabfrage starten:

```
    private void button1_Click(object sender, System.EventArgs e)
    {
        SqlDataAdapter da = new SqlDataAdapter("SELECT CompanyName As Firma,
                            ContactName As Kontaktperson, City As Ort
                            FROM Customers ORDER BY CompanyName", conn);
        ds.Clear();
        cmd.Connection = conn;
```

Das Zusammenbasteln des UPDATE-Strings verlangt etwas Fingerspitzengefühl, darf man doch auch die Apostrophe ('), die die Feldbezeichner einschließen, nicht vergessen:

```
        cmd.CommandText = $"UPDATE Customers SET CompanyName = '{textBox1.Text}'
                           WHERE City = '{textBox2.Text}'";
```

Sicherheitshalber haben wir diesmal den kritischen Programmteil in eine Fehlerbehandlungsroutine eingebaut:

```
        try
        {
             conn.Open();
```

Die folgende Anweisung führt den UPDATE-Befehl aus und zeigt gleichzeitig die Anzahl der in der Datenbank geänderten Datensätze an:

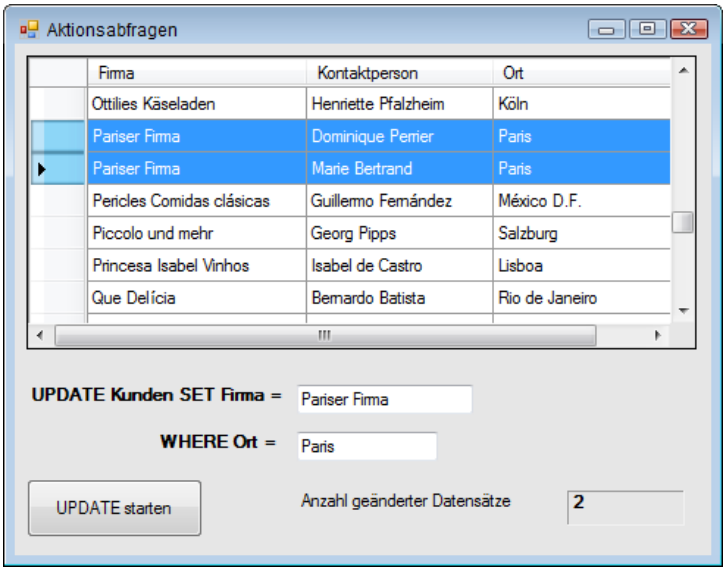
```
             label1.Text = cmd.ExecuteNonQuery().ToString();
        }
        catch (Exception ex)
        {
             MessageBox.Show(ex.Message);
        }
        da.Fill(ds, "Kunden");
        conn.Close();
```

Das *DataGridView* an das *DataSet* anklemmen:

```
        dataGridView1.DataSource = ds;
        dataGridView1.DataMember = "Kunden";
    }
}
```

Test

Stimmt die Verbindungszeichenfolge des *Connection*-Objekts, dürfte es keine Probleme beim Ausprobieren unterschiedlicher Updates geben.

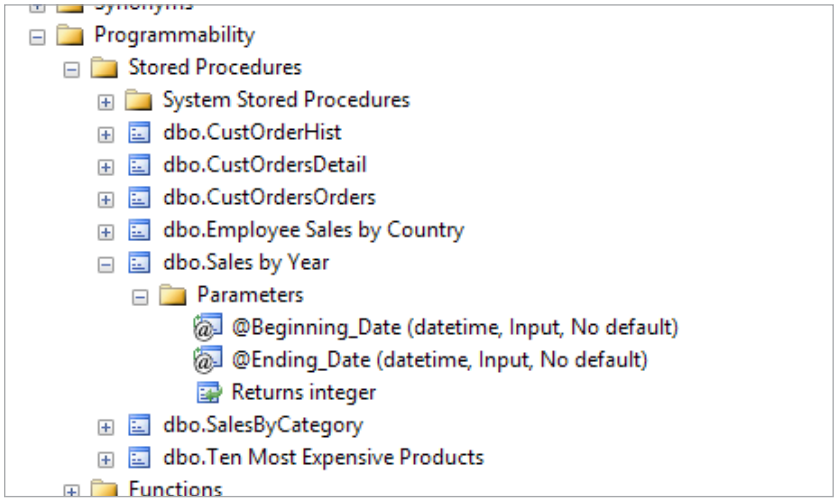


Bemerkungen

- Bei SQL-Aktionsabfragen werden keine Datensätze gelesen bzw. zurückgeliefert, sondern es geht lediglich um direkte Änderungen in der Datenquelle per SQL-Befehl (UPDATE, INSERT, DELETE). Ein *DataSet* ist dabei nicht beteiligt!
- Wie Sie die Änderungen zuerst in einer *DataTable* vornehmen und erst danach in die Datenbank zurückschreiben, erfahren Sie im Praxisbeispiel in Abschnitt 14.9.4, Die Datenbank aktualisieren.

14.9.3 Eine StoredProcedure aufrufen

Öffnen Sie im SqlServer Managment Studio innerhalb der *Northwind*-Datenbank den Knoten *Programmability* und Sie sehen alle vordefinierten StoredProcedures der Datenbank, die Sie natürlich auch selbst um weitere ergänzen können:



Hinter jeder Auswahlabfrage verbirgt sich in der Regel eine parametrisierte SQL-SELECT-Anweisung, die Sie sich im Management-Studio anzeigen lassen können. Dabei finden Sie auch die zu übergebenden Parameter und deren Datentypen leicht heraus:

Oberfläche

Ein *DataGridView*, zwei *TextBox*en und ein *Button* sollen für unseren Test genügen (siehe Laufzeitansicht am Schluss).

Quellcode

```
using System.Data.SqlClient;
public partial class Form1 : Form
{
...
```

```
private void button1_Click(object sender, System.EventArgs e)
{
    SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                  Initial Catalog=Northwind;Integrated Security=true");

    SqlCommand cmd = new SqlCommand("[Sales By Year]", conn);
    cmd.CommandType = CommandType.StoredProcedure;
```

Die Definition der beiden Parameter und das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
    SqlParameter parm1 = new SqlParameter("@Beginning_Date", SqlDbType.Date);
    parm1.Direction = ParameterDirection.Input;
    parm1.Value = Convert.ToDateTime(textBox1.Text);
    cmd.Parameters.Add(parm1);

    SqlParameter parm2 = new SqlParameter("@Ending_Date", SqlDbType.Date);
    parm2.Direction = ParameterDirection.Input;
    parm2.Value = Convert.ToDateTime(textBox2.Text);
    cmd.Parameters.Add(parm2);
```

Das *Command*-Objekt wird dem Konstruktor des *DataAdapter*s übergeben. Nach dem Öffnen der *Connection* wird die Abfrage ausgeführt. Die zurückgegebenen Datensätze werden in einer im *DataSet* neu angelegten Tabelle mit einem von uns frei bestimmten Namen *Jahresumsätze* gespeichert:

```
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    DataSet ds = new DataSet();
    try
    {
        conn.Open();
        da.Fill(ds, "Jahresumsätze");
        conn.Close();
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
```

Die Anzeige:

```
    dataGridView1.DataSource = ds;
    dataGridView1.DataMember = "Jahresumsätze";
```

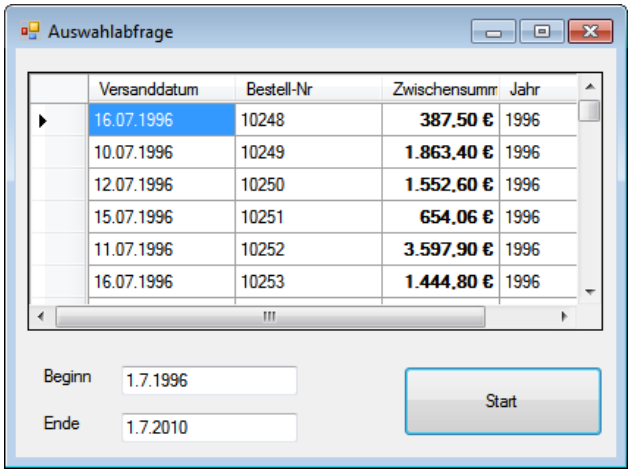
Wenigstens die Währungsspalte sollte eine ordentliche Formatierung erhalten (bei den übrigen Spalten belassen wir es bei den Standardeinstellungen):

```
  dataGridView1.Columns.Remove("Subtotal");
  DataGridViewTextBoxColumn tbc = new DataGridViewTextBoxColumn();
  tbc.DataPropertyName = "Subtotal";
  tbc.HeaderText = "Zwischensumme";
  tbc.Width = 80;
  tbc.DefaultCellStyle.Format = "c";
  tbc.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
  tbc.DefaultCellStyle.Font = new Font(dataGridView1.Font, FontStyle.Bold);
  tbc.DisplayIndex = 2;
```

```
        dataGridView1.Columns.Add(tbc);
    }
}
```

Test

Nach Eingabe sinnvoller Datumswerte dürfte sich Ihnen der folgende Anblick bieten:



14.9.4 Die Datenbank aktualisieren

Aktualisieren (UPDATE), Hinzufügen (INSERT) und Löschen (DELETE) von Datensätzen zählen zu den kritischen Datenbankoperationen, die auch unter ADO.NET weitaus mehr Aufmerksamkeit erfordern als eine einfache SELECT-Abfrage.

Ganz im Einklang mit der ADO.NET-Philosophie müssen wir dabei in drei Etappen vorgehen:

- Das *DataSet* mit der Datenbank verbinden, um bestimmte Datensätze von dort abzuholen (hierzu wird das *SelectCommand*-Objekt des *DataAdapter*s eingesetzt).
- Bei abgekoppelter Datenbank die Änderungen direkt im *DataSet* vornehmen (hierzu ist eine SQL-Anweisung leider untauglich, da das *DataSet* kein SQL kennt).
- Das *DataSet* irgendwann wieder mit der Datenbank verbinden, um die Inhalte zu aktualisieren (hierzu werden *UpdateCommand*-, *InsertCommand*- und *DeleteCommand*-Objekt des *DataAdapter*s gebraucht).

Wir wollen das am Beispiel der *Products*-Tabelle aus der Datenbank *Northwind.mdb* demonstrieren.

Oberfläche

Neben zwei *Button*s zum Anzeigen und Aktualisieren brauchen wir noch eine *DataGridView*-Komponente (siehe Laufzeitabbildung).

Quellcode (Command-Objekte selbst programmiert)

```
using System.Data.SqlClient;

public partial class Form1 : Form
{
    //Die wichtigsten Objekte sollten global verfügbar sein:
    private SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                    Initial Catalog=Northwind;Integrated Security=true");
    private SqlDataAdapter da = null;
    private DataSet ds = null;
    ...
```

Die folgende Methode *getArtikel* liefert ein gefülltes *DataSet* zurück:

```
    public DataSet getArtikel()
    {
```

SelectCommand-Objekt für *DataAdapter* erstellen (geschieht automatisch beim Instanziieren):

```
        string selStr = "SELECT ProductId As ArtikelNr, ProductName As Artikelname,
                         UnitPrice As Einzelpreis, ReorderLevel As Mindestbestand
                         FROM Products ORDER BY Productname";
        da = new SqlDataAdapter(selStr, conn);
```

Die folgende Anweisung sorgt dafür, dass neu hinzugefügte Datensätze sofort einen Primärschlüssel erhalten.

```
        da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
        conn.Open();
        DataSet ds = new DataSet();
        da.Fill(ds, "Artikel");
        conn.Close();
        return ds;
    }
```

Der Methode *setArtikel* wird ein gefülltes *DataSet* per Referenz übergeben. Mithilfe eines *CommandBuilder*s werden im Hintergrund für den *DataAdapter* die *UpdateCommand*-, *InsertCommand*- und *DeleteCommand*-Objekte erstellt, die für das Zurückschreiben der im *DataSet* vorgenommenen Änderungen in die Datenbank verantwortlich zeichnen.

```
    public void setArtikel(ref DataSet ds)
    {
        SqlCommandBuilder cb = new SqlCommandBuilder(da);

        conn.Open();
        da.Update(ds, "Artikel");
        conn.Close();
    }
```

Anzeigen:

```
    private void button1_Click(object sender, EventArgs e)
    {
```

```
        dataGridView1.DataSource = null;
        ds = getArtikel();
        dataGridView1.DataSource = ds;
        // DataGridView mit DataSet verbinden
        dataGridView1.DataMember = "Artikel";
        formatDataGridView(dataGridView1);
    }
```

Aktualisieren:

```
    private void button2_Click(object sender, EventArgs e)
    {
```

Nur die Änderungen in die Datenbank zurückschreiben:

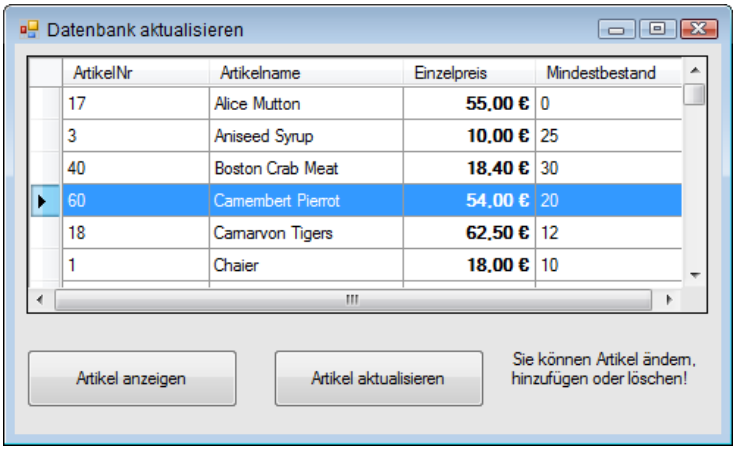
```
        DataSet ds1 = ds.GetChanges();
        if (ds1 != null)
        {
            try
            {
                setArtikel(ref ds1);
```

Die per Referenz zurückgegebenen Datensätze werden mit dem Original-*DataSet* zusammengeführt:

```
                ds.Merge(ds1);
                ds.AcceptChanges();
                MessageBox.Show("Datenbank wurde aktualisiert!", "Erfolg");
            }
            catch (Exception ex)
            {
                ds.RejectChanges();
                MessageBox.Show(ex.Message, "Fehler");
            }
        }
    }
}
```

Test

Klicken Sie auf die „Artikel anzeigen“-Schaltfläche, um das *DataSet* anzuzeigen. Nehmen Sie dann einige Änderungen direkt im *DataGridView* vor, fügen Sie Datensätze hinzu (dazu scrollen Sie an das Ende des *DataGridView*s) oder löschen Sie Datensätze (mit der *Entf*-Taste, nachdem Sie die komplette Zeile markiert haben). Klicken Sie auf „Artikel aktualisieren“, um die Änderungen in die Datenbank zu übertragen. Lassen Sie sich danach erneut die Artikel anzeigen, um sich davon zu überzeugen, dass alle Änderungen tatsächlich in der Datenbank angekommen sind.



Bemerkungen

- Der Code zur Formatierung der *Einzelpreis*-Spalte des *DataGridView*s wurde hier nicht mit abgedruckt (siehe Beispieldaten).
- Es ist auch möglich, mehrere Datensätze hintereinander zu ändern, hinzuzufügen bzw. zu löschen, bevor der Abgleich mit der Datenbank erfolgt.
- In der Regel werden Sie nur die von Ihnen selbst hinzugefügten Datensätze löschen können, da die originalen Datensätze in Relationen zu anderen Tabellen eingebunden sind.

Index

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

Q

R

S

T

U

V

W

X

Y

Z