AUSGABE 05/2018

DATENBANK ENTWICKLER

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

TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER

Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

Von Access zum Entity Framework: Basics

In den vorherigen Ausgaben haben wir bereits verschiedene Techniken erläutert und unter anderem kleine Desktop- und Webanwendungen programmiert. Dabei sind wir ein wenig vom eigentlichen Ziel des Magazins abgewichen – Access-Entwicklern die Möglichkeiten von Visual Studio und den dortigen Technologien für die Migration von Access-Anwendungen in Desktop- oder Webanwendungen aufzuzeigen. Mit diesem Artikel kehren wir dorthin zurück und erklären, wie Sie den Umzug einer Anwendung von Access zu .NET einleiten und welche Techniken wir dazu in Zukunft nutzen wollen.

Es gibt zahllose Möglichkeiten, wie Sie Desktop-Anwendungen ähnlich denen, die man mit Access erstellt, unter .NET zu programmieren. Das beginnt mit der Wahl der Technik für die Gestaltung der Benutzeroberfläche, geht über die Entscheidung, welches Datenbanksystem man einsetzt bis hin zur Technik für den Datenzugriff.

Benutzeroberfläche

Was die Gestaltung der Benutzeroberfläche angeht, haben wir von der ersten Ausgabe an gezeigt, wie Sie die Benutzeroberfläche mit WPF umsetzen. In der Zwischenzeit haben wir auch einige Artikel darauf verwendet, die Erstellung von Web-Anwendungen auf Basis von HTML zu erläutern.

Backend

Bezüglich des verwendeten Backends haben wir verschiedene Phasen durchlaufen: Zu Beginn haben wir gezeigt, wie Sie über ADO.NET auf die Tabellen einer Access-Datenbank zugreifen können. Access-Datenbanken sind aber aus verschiedenen Gründen nicht die erste Wahl für das Backend: Erstens ist die Dateigröße auf zwei Gigabyte beschränkt, zweitens gibt es kein Berechtigungssystem und drittens kann es nur eine stark begrenzte Anzahl gleichzeitiger Zugriffe verarbeiten. Hier sind andere Lösungen gefragt, weshalb wir auf den SQL Server respektive dessen abgespeckter Version namens LocalDb umgestiegen sind. Für solche Zwecke, wo kein SQL Server vorhanden ist, haben wir den Einsatz von SQLite erläutert.

Datenzugriffstechnik

Um den Umstieg zu erleichtern, haben wir zu Beginn noch auf Basis von Access-Datenbanken gearbeitet und auf diese über ADO.NET zugegriffen. Aus den oben genannten Gründen haben wir uns dann auf SQL Server und SQLite konzentriert. Das hat noch einen anderen Grund: Wer die Vorteile der objektorientierten Entwicklung stärker nutzen möchte und gleichzeitig weniger Zeit in die Programmierung des Datenzugriffs stecken möchte, kann einen sogenannten O/R-Mapper nutzen, einen Mapper, der ein Bindeglied zwischen der objektorientierten und der relationalen Welt bildet. In diesem Fall wollen wir den Zugriff auf die Daten über eine Reihe geeigneter Objekte realisieren, indem wir zum Beispiel auf die in Tabellen gespeicherten Entitäten in Form von Objekten zugreifen – und auf die Gesamtheit dieser Entitäten über entsprechende Auflistungsklassen.

Unsere Wahl fiel dabei auf das von Microsoft entwickelte Entity Framework. Hier haben wir zu Beginn erläutert, wie Sie ein Entity Data Model, also im Prinzip die Abbildung des relationalen Datenmodells, auf Basis einer bestehenden Tabelle erstellen. Dazu ist, wenn Sie von Access aus kommen, zuerst die Migration der Datenbank in eine Datenbank eines geeigneten Datenbanksystems erforderlich. Das Entity Framework bietet Treiber für verschiedene Datenbanksysteme an, zum Beispiel SQLite oder den

SQL Server, aber nicht für Access – daher können wir hier nicht bei Access als Backendsystem verharren. Der Einsatz des Entity Frameworks bedingt also die komplette Abkehr von Access.

Wann Access nutzen und wann nicht

Nebenbei bemerkt: Es ist für viele Szenarien sinnvoll, mit Access zu arbeiten – also etwa dann, wenn nicht sehr viele Zugriffe auf die Datenbank erfolgen und kein Berechtigungssystem erforderlich ist. Die Geschwindigkeit, mit der Sie eine Anwendung auf Basis von Access erstellen, dürfte mit keinem anderen System möglich sein, auch nicht mit Visual Studio.

Es ist auch dann noch sinnvoll, mit Access zu arbeiten, wenn es nur das Frontend für den Zugriff auf eine Backend-Datenbank etwa auf dem SQL Server stellt. Auch dann profitieren Sie vom einfachen Entwurf der Benutzeroberfläche. Wenn es jedoch darum geht, ein Backend zu einem alternativen Frontend zu Access zu wählen, also etwa für eine WPF- oder Webanwendung, dann sollten Sie nicht auf den Tabellen in einem Access-Backend verharren, sondern eine andere Lösung wählen.

Von Database First zu Model First

In den ersten Artikeln zum Thema Entity Framework haben wir dann auch noch die Tabellen in einer SQL Server- oder SQLite-Datenbank erstellt und dann ein Entity Data Model auf Basis dieser Tabellen erstellen lassen. Dazu haben wir dann meist die Vorlage EF Designer aus Datenbank verwendet. Dieser Ansatz bot uns auch noch die Sicherheit, dass wir das so generierte Modell in einer grafischen Benutzeroberfläche anpassen konnten.

Hinter dieser grafischen Benutzeroberfläche steckte dabei das Entity Data Model, das aus den Entitäten beziehungsweise Entitätsklassen besteht sowie einer Auflistungsklasse für jede Entität. In den Entitäten sind dann die Beziehungen zwischen den Entitäten durch entsprechende Eigenschaften definiert.

In den vorherigen Ausgaben sind wir dann einen Schritt weitergegangen und haben begonnen, uns für eine Weile von dem Vorhandensein einer fertigen Datenbank als Ausgangspunkt zu lösen. Dort haben wir zuerst die Entitätsklassen erstellt und die Beziehungen zwischen den Klassen über die dazu vorgesehenen Eigenschaften festgelegt. Daraus haben wir dann auf umgekehrtem Wege wie zuvor die Datenbank erstellt. Dieser Ansatz heißt Code First.

Von Access über Model First zur Datenbank

In dieser Ausgabe von **DATENBANKENTWICKLER** gehen wir schließlich noch einen Schritt weiter und wollen ein Entity Data Modell aus einer Access-Datenbank entwickeln. Dazu haben wir im Artikel **Von Access zu Entity Framework: Datenmodell** beschrieben, wie Sie mit einer VBA-Routine aus Access heraus den Code generieren können, aus dem das Entity Data Model besteht – also aus den Entitätsklassen und den Anweisungen zur Deklaration von **DbSet**-Objekten für den Zugriff auf die Auflistungen. Ein Objekt auf Basis einer Entitätsklasse entspricht dabei einem Datensatz einer Tabelle und ein **DbSet**-Objekt ist die Grundlage, Auflistungsklassen auf Basis der Daten einer Tabelle zu erzeugen, also quasi mit Recordsets vergleichbar ist.

Das so erstellte Entity Data Model können Sie dann nutzen, um mit wenigen Anweisungen eine Datenbank auf Basis der Definition der Klassen zu erzeugen.

Im Artikel **Von Access zu Entity Framework: Daten** berichten wir dann darüber, wie Sie Anweisungen zum Füllen der zuvor auf Basis des Entity Data Models erzeugten Klassen zusammenstellen. Dabei durchlaufen wir die Tabellen, deren Definition

wir zuvor schon in Entitätsklassen umgewandelt haben, und legen für jeden Datensatz eine eigene Anweisung an, die diesen Datensatz dann über das Entity Framework zu der neu erzeugten SQL Server-Datenbank hinzufügt.

Und weiter?

Mit diesem Handwerkzeug ausgestattet können Sie bereits einfache Datenmodelle aus Access-Anwendungen in ein Entity Data Model umwandeln, eine Datenbank daraus erzeugen und die Tabellen mit Daten füllen. Damit brauchen Sie sich nicht mehr um die Datenbank und die enthaltenen Tabellen zu kümmern, denn Sie greifen nur noch über das Entity Data Model auf die Daten in der Datenbank zu.

Hier setzen wir in weiteren Artikeln an und zeigen Ihnen, wie Sie die Daten so ähnlich wie möglich wie in der gewohnten Access-Benutzeroberfläche abbilden – und das komplett auf Basis des Entity Data Models, das Sie direkt aus Ihrer Access-Anwendung erstellt haben. Hier schauen wir uns unter anderem die folgenden Techniken an:

- **Datenblatt**: Mit einem **DataGrid**-Steuerelement wollen wir die Funktionen eines Datenblatts abbilden. Das heißt, dass wir direkt im **DataGrid**-Steuerelement vorhandene Datensätze bearbeiten und löschen, neue Datensätze hinzufügen und in den Datensätzen navigieren wollen.

- **Detailansichten**: Ausgehend vom oben beschriebenen DataGrid wollen wir die Möglichkeit schaffen, einzelne Datensätze in einem neuen Formular in der Detailansicht anzuzeigen.

- **Anzeige und Bearbeitung von Daten in 1:n-Beziehungen**: Wir wollen die Daten aus 1:n-Beziehungen in eigenen Fenstern darstellen, wobei der Datensatz mit dem Primärschlüsselfeld der Beziehung als einzelner Datensatz angezeigt wird und die über das Fremdschlüsselfeld damit verknüpften Datensätze in einem DataGrid/Datenblatt. Beispiel: Kunden und Projekte.

- **Anzeige und Bearbeitung von Daten in m:n-Beziehungen**: Auch die Daten in m:n-Beziehungen wollen wir auf diese Art anzeigen, wobei es sich hier ja normalerweise um eine verkappte 1:n-Beziehung handelt – Beispiel: Bestellung und Bestelldetails/Artikel

Die hier dargestellten Szenarien kommen Ihnen vermutlich größtenteils bekannt vor, da wir diese schon in der einen oder anderen Weise beschrieben haben. Wir wollen diesmal allerdings auch die Möglichkeit bieten, diese Ansichten automatisch auf Basis der dazu gewählten Tabellen und Felder per Code zu generieren, damit Sie sich die vielen manuellen Handgriffe sparen.

Damit werden wir in der nächsten Ausgabe von **DATENBANKENTWICKLER** beginnen, in dieser Ausgabe zeigen wir Ihnen ja bereits, wie Sie das Entity Data Model auf Basis einer Access-Datenbank erstellen und mit den entsprechenden Daten füllen. Und früher oder später werden wir uns dann auch endlich dem Thema Reporting widmen, das wir bisher etwas vernachlässigt haben. Das liegt aber auch daran, dass es keine für jeden Zweck passende Technik gibt wie etwa unter Access – hier konnte man mit dem Objekttyp Bericht quasi alles von der Rechnung über Kataloge abdecken. Unter .NET gibt es keine solch einfache Lösung.

Entity Framework: Datenbankinitialisierung

Wenn Sie mit Code First arbeiten, also Ihre Datenbank auf Basis eines Entity Data Models erstellt wird, können Sie verschiedene Strategien auswählen, um die Datenbank zu erstellen oder anzupassen, wenn Sie die Anwendung an einen anderen Benutzer weitergeben. Dieser Artikel zeigt, welche Möglichkeiten es gibt und wie diese funktionieren.

In den letzten Ausgaben von **DATENBANKENTWICKLER** haben wir vermehrt mit Anwendungen gearbeitet, in denen wir ein Entity Data Model definiert haben, also verschiedene Klassen für die Objekte und entsprechende Auflistungen. Daraus erstellt das Entity Framework dann bei Bedarf eine Datenbank, welche die Klassen und Auflistungen berücksichtigt. Dabei gibt es verschiedene Möglichkeiten, die Datenbank zu erstellen, wenn diese noch nicht vorhanden ist oder diese anzupassen, wenn Sie eine neue Version der Datenbank an den Benutzer weitergeben.

Als Voraussetzung wollen wir ein einfaches Modell heranziehen, dass nur aus den beiden Tabellen **Articles** und **Categories** besteht. Diese haben wir in einem neuen Projekt namens **InitializationSample** als Visual Basic-WPF-App angelegt, dem wir ein neues Objekt des Typs **ADO.NET Entity Data Model** für ein **Leeres Code First-Modell** namens **ArticleContext** hinzugefügt haben.

Hier legen wir im Unterordner **Data** die erste Klasse namens **Articles.vb** an:

```
Public Class Article
    Public Property ID As Integer
    Public Property Name As String
    Public Property CategoryID As Integer
    Public Property Category As Category
    Public Property Price As Decimal
End Class
```

Die zweite Klasse nimmt die Kategorien auf und heißt **Categories.cs**:

```
Public Class Category
    Public Property ID As Integer
    Public Property Name As String
    Public Property Articles As ICollection(Of Article)
End Class
```

Dazu kommen zwei Zeilen, welche die Auflistungen der Elemente dieser beiden Klassen repräsentieren. Diese fügen wir in der Klasse ein, die beim Hinzufügen des Entity Data Models angelegt wurde und den Namen **ArticleContext.vb** trägt. Diese sieht dann so aus:

```
Imports System.Data.Entity
```

```
Public Class ArticleContext
    Inherits DbContext
    Public Sub New()
        MyBase.New("name=ArticleContext")
    End Sub
    Public Property Categories() As DbSet(Of Category)
    Public Property Articles() As DbSet(Of Article)
End Class
```

Datenbank beim ersten Zugriff erstellen

Entity Framework hat einen Mechanismus, der dafür sorgt, dass eine Datenbank automatisch erstellt wird, wenn Sie das erste Mal auf diese zugreifen, aber noch keine Datenbank vorhanden ist. Um dies auszuprobieren, fügen Sie der Code behind-Klasse des Fensters **MainWindow.xaml** zunächst den Konstruktor hinzu und füllen diesen dann mit den folgenden Anweisungen:

```
Class MainWindow
    Public Sub New()
        Using dbContext As ArticleContext = New ArticleContext
            Dim MyCategory As Category
            MyCategory = New Category()
            MyCategory.Name = "Getränke"
            dbContext.Categories.Add(MyCategory)
            dbContext.SaveChanges()
        End Using
    End Sub
End Class
```

Die Methode **New** erstellt einen neuen Kontext auf Basis der Klasse **ArticleContext** und speichert diese in der Variablen **dbContext**. Dann erstellen wir ein neues Objekt des Typs **Category**, weisen der Eigenschaft **Name** den Wert **Getränke** zu und fügen das neue Objekt zur Auflistung **Categories** des Objekts **dbContext** hinzu. Die Änderungen speichern wir dann mit der Methode **SaveChanges**.

Nun brauchen Sie die Anwendung nur noch zu starten. Es dauert dann ein wenig länger bis zum Erscheinen des Anwendungsfensters, aber dafür wird in der Zwischenzeit auch gleich die Datenbank erstellt.

Diese finden Sie dann nach dem Beenden der Anwendung unter Visual Studio im SQL Server-Objekt-Explorer, den Sie mit dem Menübefehl **Ansicht|SQL Server-Objekt-Explorer**

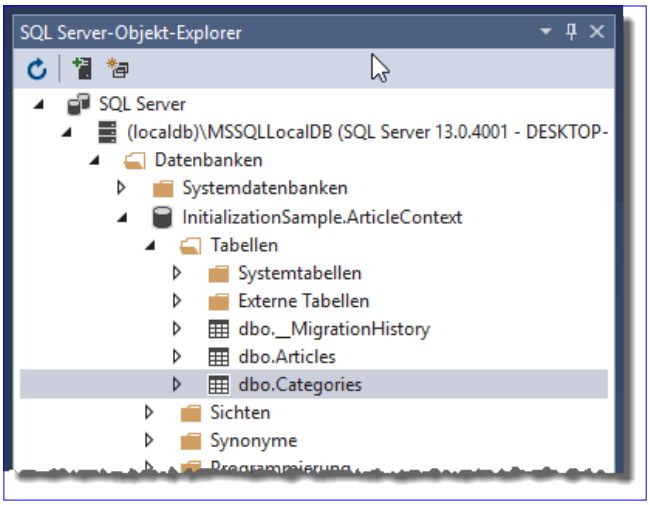

Bild 1: Die neue Datenbank im SQL Server-Objekt-Explorer

einblenden können. Hier klappen Sie den Eintrag **Datenbanken** auf und finden die Datenbank **InitializationSample.ArticleContext** vor (siehe Bild 1).

Wenn Sie nun noch das Kontextmenü des Eintrags **dbo.Categories** öffnen und dort den Befehl **Daten anzeigen** auswählen, erscheinen die Daten der Tabelle **Categories** – in diesem Fall der eine in der Konstruktor-Methode angelegte Datensatz mit dem Wert **Getränke** im Feld **Name**.

Ergebnisse beim Anlegen der Datenbank

Schauen wir uns an, was im Detail geschehen ist. Als Erstes ist festzuhalten: Es wurde eine Datenbank angelegt, und zwar auf dem Server **(LocalDb)\MSSQLLocalDB** und unter dem Namen **InitializationSample.ArticleContext**.

Wenn Sie diese Einstellung ändern wollen, können Sie das in der Datei **App.config** (für Desktop-Anwendungen) oder **Web.config** (für Web-Anwendungen) erledigen, wo Sie den folgenden Abschnitt anpassen müssen:

```
<connectionStrings>
    <add name="ArticleContext" connectionString="data source=(LocalDb)\MSSQLLocalDB;
        initial catalog=InitializationSample.ArticleContext;integrated security=True;MultipleActiveResultSets=True;
        App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Woher weiß Entity Framework, dass es genau diese Verbindungszeichenfolge nutzen soll? Wir sehen hier, dass diese den Namen **ArticleContext** aufweist. Dieser wird in der Konstruktor-Methode der Klasse **ArticleContext.vb**, die mit dem Entity Data Model erstellt wurde, referenziert:

```
Public Sub New()
    MyBase.New("name=ArticleContext")
End Sub
```

Wenn Sie also eine andere Verbindungszeichenfolge nutzen wollen, stellen Sie diese hier ein.

Alternativer Datenbankname

Während wir dem **New**-Konstruktor von **MyBase** soeben den Namen der Verbindungszeichenfolge übergeben haben, können Sie hier auch den Namen der zu erstellenden Datenbank angeben. Dazu lassen Sie einfach den Namen des Parameters weg und geben direkt den gewünschten Datenbanknamen an, zum Beispiel **InitializationSample**:

```
Public Sub New()
    MyBase.New("InitializationSample")
End Sub
```

Sie können aber auch einfach den Namen der Datenbank in der Verbindungszeichenfolge anpassen. Diesen finden Sie unter dem Parameternamen **initial catalog**.

Entity Framework: Datenbankmigration

Unter Access hatten Sie ein Problem, wenn Sie eine neue Version einer Datenbank ausliefern wollten, deren Datenmodell sich geändert hat. Dann war Handarbeit angesagt! Das Entity Framework bietet für das Übertragen von Änderungen am Datenmodell die sogenannten Migrationen an. Dieser Artikel zeigt, was es sich damit auf sich hat und wann Sie Migrationen gegenüber Datenbankinitialisierern nutzen sollten.

Wenn Sie eine Datenbank weiterentwickeln, die schon im produktiven Betrieb genutzt wird, gibt es verschiedene Änderungen, die unterschiedlich gehandhabt werden. Wenn Sie lediglich Änderungen an der Benutzeroberfläche oder an der Anwendungslogik vornehmen, haben Sie leichtes Spiel: Sie brauchen dann einfach nur die neue Version der Anwendung zu verteilen, die dann auf die vorhandene Datenbank zugreift.

Interessant wird es, wenn Sie nicht nur Änderungen an der Benutzeroberfläche oder der Anwendungslogik vornehmen, sondern wenn Sie das Entity Data Model manipulieren. Dann müssen nämlich auch die Tabellen in der zugrunde liegenden Datenbank angepasst werden, da sonst das Modell nicht mehr zur Datenbank passt. Hier wird es interessant, denn Entity Framework bietet die sogenannten Migrations, um Änderungen an den Entities zuverlässig in die Datenbank zu übertragen.

Während der Entwicklung der Anwendung, also bevor diese erstmals auf dem Rechner des Benutzers landet, ist der Umgang mit Aktualisierungen im Entity Data Model einfach: Man konnte dann einfach die komplette vorhandene Datenbank löschen und diese auf Basis des Entity Data Models neu aufbauen. Gegebenenfalls hat man dann noch mit der **Seed**-Methode einige Daten zu den frisch erstellten Tabellen hinzugefügt. Wie das im Detail funktioniert, erfahren Sie im Artikel **Entity Framework: Datenbank-Initialisierer**.

Sobald die Anwendung aber beim Kunden im Einsatz ist und dieser damit begonnen hat, Daten in die Tabellen zu schreiben oder diese zu ändern, kann man nicht mehr so einfach die komplette Datenbank löschen und neu erstellen – dann wäre die ganze Arbeit des Benutzers verloren. Alternativen wären das manuelle Schreiben von SQL-Anweisungen, mit denen die Änderungen im Entity Data Model feingranular in die Datenbank übertragen werde, ohne diese löschen und neu erstellen zu müssen. Oder man erstellt die Datenbank neu und erzeugt ein Skript, mit dem die Daten aus der Datenbank in der alten Version in die neue Version übertragen werden. Auch das ist mit viel Arbeit verbunden. Aber schauen wir uns doch an, was Entity Framework zu diesem Thema zu bieten hat.

Beispielprojekt

Das Beispielprojekt **DatabaseMigrations** auf Basis der Vorlage **Visual BasicIWPF-App** haben wir wieder mit einem neuen Element des Typs **ADO.NET Entity Data Model** namens **ArticleContext** ausgestattet, für das wir als Modellinhalt **Leeres Code First-Modell** gewählt haben. Dem Projekt haben wir dann einen neuen Ordner namens **Data** mit den beiden Klassen **Article.cs** und **Category.cs** hinzugefügt. Außerdem haben wir der durch Entity Framework erstellten Klasse **ArticleContext.vb** noch die beiden **DbSet**-Auflistungen **Articles** und **Categories** hinzugefügt (siehe Beispielprojekt).

Migrations

Dieses bietet nämlich eine Technik namens **Migrations** an. Dabei gibt es zwei verschiedene Funktionen:

- Code-basierte Migration: Muss durch den Entwickler ausgelöst werden.
- Automatische Migration: Erfolgt automatisch, hat aber Einschränkungen.

In den folgenden Abschnitten schauen wir uns die verschiedenen Funktionen an.

Code-basierte Migration

Die code-basierte Migration erfordert den Aufruf verschiedener Befehle über die Paket-Manager-Konsole. Diese starten Sie über den Menübefehl **Extras|NuGet-Paket-Manager|Paket-Manager-Konsole**. Hier können Sie die folgenden Anweisungen eingeben:

- **Enable-Migrations**: Aktiviert die Migration für das aktuelle Projekt und erstellt einen neuen Ordner namens **Migrations** mit einer neuen Klasse namens **Configuration**.
- **Add-Migration**: Erfasst die Unterschiede zwischen dem Entity Data Model im Vergleich zur Datenbank und trägt diese in jeweils eine neue Klasse in zwei Methoden namens **Up** und **Down** ein. Die Methoden enthalten die Anweisungen zum Erstellen, Löschen oder Ändern der Datenbankobjekte entsprechend den Änderungen im Entity Data Model.
- **Update-Database**: Diese Methode überträgt alle noch nicht übertragenen Änderungen aus den mit **Add-Migration** erstellten Klassen in die Tabellen der Datenbank.

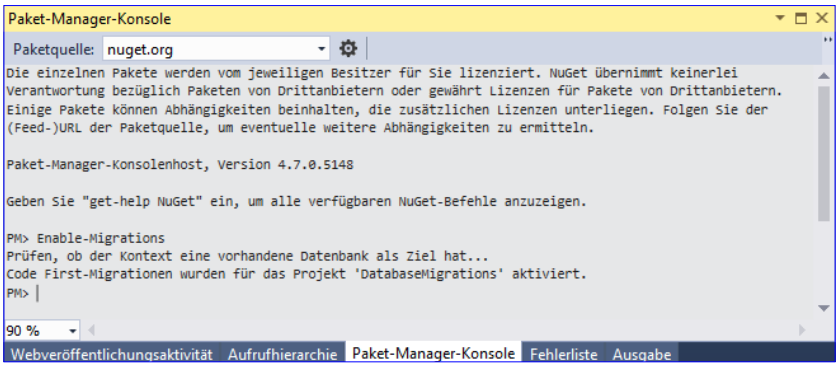

Bild 1: Aktivieren der Datenbank-Migrationen

Um die Migrations-Funktionen zu nutzen, müssen sie diese mit der ersten Anweisung **Enable-Migrations** aktivieren. Diese Anweisung setzen Sie in der Paket-Manager-Konsole ab (siehe Bild 1):

```
Enable-Migrations
```

Dies quittiert die Paket-Manager-Konsole damit, dass Code First-Migrationen für das Projekt aktiviert sind. Was hat sich dadurch verändert? Im Projektmappen-Explorer finden wir nun einen neuen Ordner namens **Migrations** mit der Klasse **Configurations.vb** (siehe Bild 2).

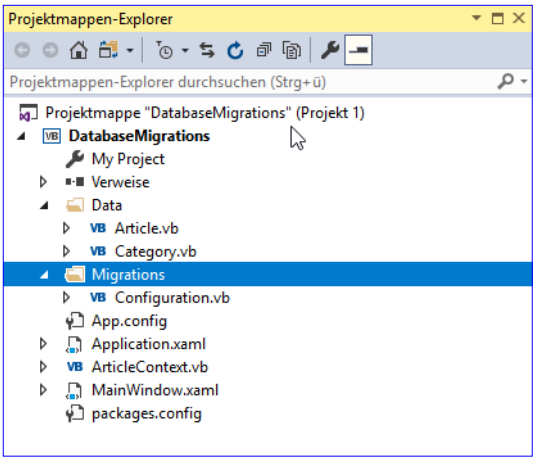

Bild 2: Neuer Ordner **Migrations**

Die Klasse stellt zwei Methoden bereit – die Konstruktor-Methode **New** sowie eine Methode namens **Seed**:

```
Imports System.Data.Entity.Migrations
```

```
Namespace Migrations
    Friend NotInheritable Class Configuration
        Inherits DbMigrationsConfiguration(Of ArticleContext)
        Public Sub New()
            AutomaticMigrationsEnabled = False
        End Sub
        Protected Overrides Sub Seed(context As ArticleContext)
            '  This method will be called after migrating to the latest version.

            '  You can use the DbSet(Of T).AddOrUpdate() helper extension method
            '  to avoid creating duplicate seed data.
        End Sub
    End Class
End Namespace
```

Die Konstruktor-Methode stellt den Wert der Variablen **AutomaticMigrationsEnabled** auf den Wert **False** ein. Was bedeutet das? Sie setzen **AutomaticMigrationsEnabled** auf **True**, wenn Sie keine Versionierung in dem Sinne planen, dass Sie nur Upgrades verwenden und keine Downgrades und keine strikte Versionierung planen. Das schauen wir uns weiter unten unter **Automatische Migration aktivieren** an. Wenn Sie die Variable auf dem Wert **False** belassen, müssen Sie die Migrationsschritte selbst definieren. Das schauen wir uns gleich im Anschluss an.

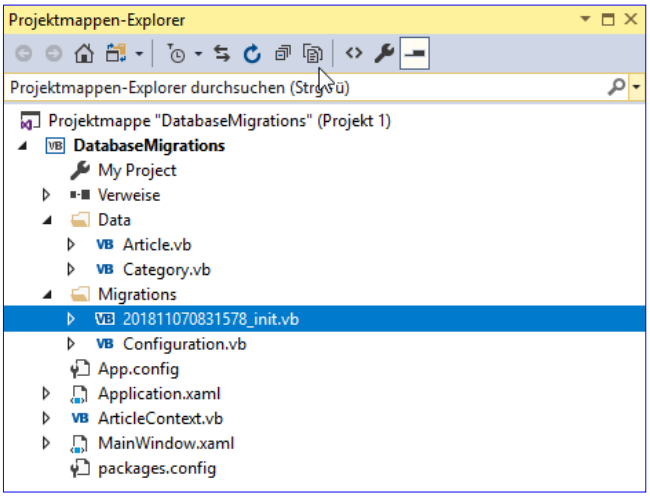

Bild 3: Neue Klasse, die auf **init.vb** endet

Änderungen in Klasse notieren

Danach können Sie erstmals die Anweisung **Add-Migration** aufrufen und dieser einen Parameter mitgeben, der später als Teil des Namens der durch diese Anweisung erstellten Klasse verwendet wird (der Rest besteht aus Datum und Zeit). Die erste Migration soll beispielsweise den Text **Init** im Namen der zu erstellenden Klasse tragen. Die folgende Anweisung setzen Sie ebenfalls in der Paket-Manager-Konsole ab. Die Ausführung kann einige Sekunden dauern:

```
Add-Migration Init
```

Die so erstellte Klasse heißt **201811070831578_init.vb** und taucht im Ordner **Migrations** des Projektmappen-Explorers auf (siehe Bild 3). Sie sieht mit den beiden Methoden **Up** und **Down** in gekürzter Form wie folgt aus:

```
Imports System.Data.Entity.Migrations
```

```
Namespace Migrations
    Public Partial Class init
        Inherits DbMigration
        Public Overrides Sub Up()
            CreateTable("dbo.Articles",
                Function(c) New With
                    {
                        .ID = c.Int(nullable := False, identity := True),
                        .Name = c.String(),
                        .CategoryID = c.Int(nullable := False),
                        .Price = c.Decimal(nullable := False, precision := 18, scale := 2),
                        .Description = c.String()
                    }) _
                .PrimaryKey(Function(t) t.ID) _
                .ForeignKey("dbo.Categories", Function(t) t.CategoryID, cascadeDelete := True) _
                .Index(Function(t) t.CategoryID)
            CreateTable(
                "dbo.Categories",
                Function(c) New With
                    {
                        .ID = c.Int(nullable := False, identity := True),
                        .Name = c.String()
                    }) _
                .PrimaryKey(Function(t) t.ID)
        End Sub

        Public Overrides Sub Down()
            DropForeignKey("dbo.Articles", "CategoryID", "dbo.Categories")
            DropIndex("dbo.Articles", New String() { "CategoryID" })
            DropTable("dbo.Categories")
            DropTable("dbo.Articles")
        End Sub
    End Class
End Namespace
```

Die meisten der Methoden, die hier zum Einsatz kommen, stammen aus dem Namespace **System.Data.Entity.Migrations**. Die Methode **Up** enthält zwei **CreateTable**-Anweisungen, mit denen die Tabellen **Articles** und **Categories** erstellt werden sollen. Wir haben also hier eine Möglichkeit, Tabellen und ihre Felder sowie die Restriktionen objektorientiert anzulegen statt mit T-SQL-Anweisungen. Letztendlich werden die Befehle natürlich auch wieder in T-SQL übersetzt, aber für manchen Entwickler, der sich vielleicht nicht mit T-SQL anfreunden möchte, bietet sich hier eine Alternative. Die Methode **Down** enthält vier Anweisungen, mit denen nacheinander die Restriktionen und die Tabellen gelöscht werden.

Änderungen in die Datenbank übertragen

Schließlich wollen Sie die Änderungen im Entity Data Model, die wir durch die **Add-Migration**-Anweisung in die Klasse mit den Methoden **Up** und **Down** geschrieben haben, noch in das Datenmodell der Datenbank übertragen. Dazu rufen wir die folgende Anweisung auf:

```
Update-Database
```

Nun werden die **Up**- und **Down**-Methoden aller seit dem letzten Aufruf von **Update-Database** angelegten Migrationsklassen ausgeführt. In unserem Fall wird die Datenbank nun überhaupt erst angelegt, denn bisher haben wir dies ja noch nicht erledigt. Auch **Update-Database** dauert einige Sekunden und schließlich erhalten Sie das Ergebnis des Aufrufs in der Paket-Manager-Konsole:

```
PM> update-database
Geben Sie das Flag '-Verbose' an, um die auf die Zieldatenbank angewendeten SQL-Anweisungen anzuzeigen.
Explizite Migrationen werden angewendet: [201811070831578_init].
Die explizite Migration wird angewendet: 201811070831578_init.
Die Seed-Methode wird ausgeführt.
```

Was ist nun geschehen? Dazu werfen wir einen Blick in den SQL Server-Objekt-Explorer (siehe Bild 4). Hier finden wir unter **Datenbanken** die neue Datenbank namens **DatabaseMigrations.ArticleContext**, die überraschenderweise nicht nur die zwei Tabellen **Categories** und **Articles** enthält, sondern noch eine weitere Tabelle namens **__MigrationHistory**. Das hört sich logisch an: Hier werden vermutlich Informationen über den aktuellen Stand der Datenbank gespeichert.

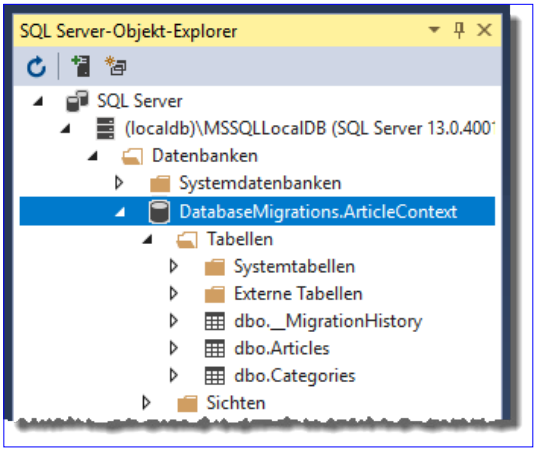

Bild 4: Die neue Datenbank im SQL Server-Objekt-Explorer

Schauen wir uns den Inhalt dieser Tabelle an (siehe Bild 5), finden wir genau einen Datensatz, der den Namen der bei **Add**-Migration erstellten Datei ohne Dateiendung enthält (**201811070831578_init**) sowie ein Feld namens **ContextKey**, der die **Configuration**-Klasse im Projekt angibt (**DatabaseMigrations.Migrations.Configuration**). Diese Tabelle bildet also die Grundlage, wenn es später darum geht, zu prüfen, welche der im Ordner **Migrations** angelegten Klassen bereits auf die Datenbank angewendet wurden und welche nicht. Die **Add-Migration**-Anweisung fügt die neue Klasse mit den **Up**- und **Down**-Methoden

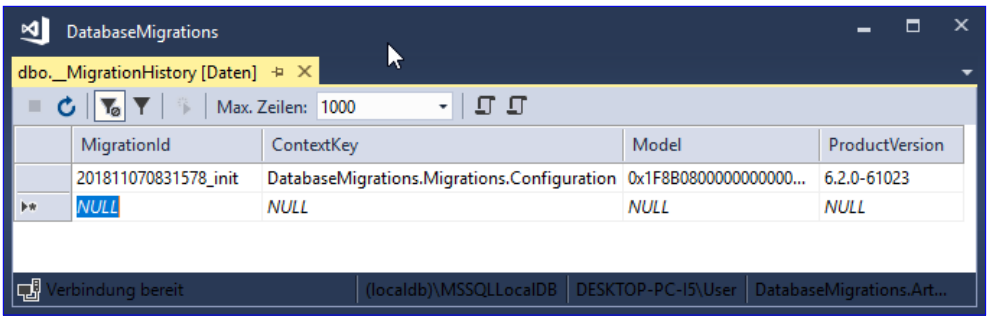

Bild 5: Datensatz für die erste Migration in der Tabelle **__MigrationHistory**

Von Access zu Entity Framework: Datenmodell

Viele Leser dieses Magazins programmieren auch mit Access. Der eine oder andere hat vielleicht sogar eigene Anwendungen oder Anwendungen von Kunden auf Access-Basis, die er gern in Form eines WPF- oder ASP.NET-Projekts umsetzen würde. Das Problem: Der Zugriff auf die Access-Datenbank ist unter .NET nur begrenzt möglich, die tolle Datenzugriffstechnologie Entity Framework beispielsweise unterstützt Access-Datenbanken nicht. Dafür unterstützt es allerdings SQL Server-Datenbanken. Wie gehen wir also vor? Wir migrieren die Access-Datenbanken zum SQL Server und bauen dann ein Entity Data Model auf Basis dieser Datenbank. Es geht allerdings auch anders: Sie könnten auch ein paar Routinen in VBA schreiben, die ein Entity Data Model direkt aus Access heraus auf Basis des gewünschten Datenmodells erzeugen. Dieser Artikel zeigt, wie letztere Möglichkeit funktioniert.

Als Beispiel habe ich eine kleine Bestellverwaltung zusammengestellt, welche die wesentlichen Tabellen enthält (siehe Bild 1).

Ziel ist es, aus diesem Datenmodell und den enthaltenen Daten ein Entity Data Model mit je einer Klasse für jede Tabelle und einer Liste der benötigten **DbSet**-Auflistungen zu erstellen.

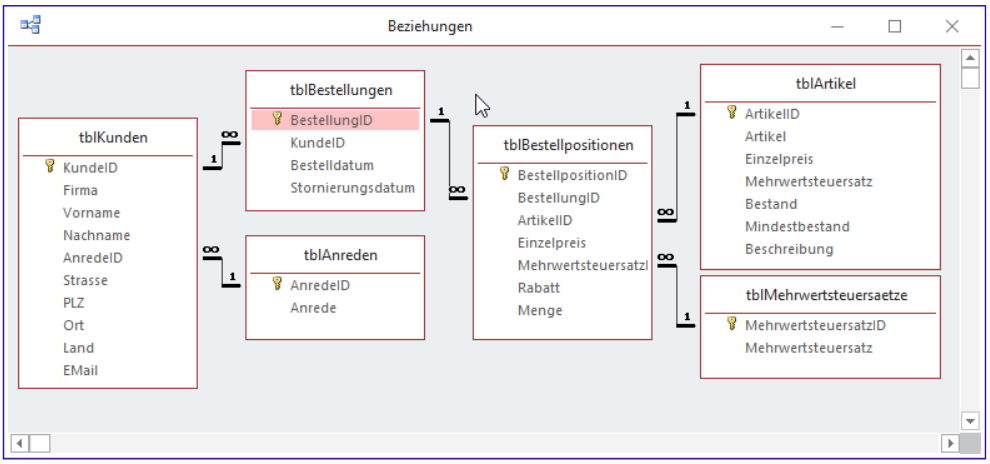

Bild 1: Beispiel für ein zu migrierendes Datenmodell

Außerdem wollen wir die Befehle für eine **Seed**-Methode zusammenstellen, die notwendig ist, um die Daten aus den Access-Tabellen dann beim Initialisieren des Entity Data Models in die zu erstellende Datenbank zu schreiben. Die Grundlagen zur Initialisierung einer Datenbank finden Sie im Artikel **Entity Framework: Datenbankinitialisierung**.

Was wollen wir also genau tun? Wir möchten beispielsweise für die Tabelle **tblAnreden**, deren Entwurf Sie in Bild 2 sehen, zwei Dinge erzeugen:

- die Definition einer Klasse und
- die Definition eines **DbSet**-Objekts.

Die Klassendefinition soll beispielsweise wie folgt aussehen:

```
Public Class Anrede
    Public Property AnredeID As System.Int32
    Public Property Anrede As System.String
End Class
```

Die Definition des **DbSet**-Objekts lautet so:

```
Public Overridable Property Anreden() As
DbSet(Of Anrede)
```

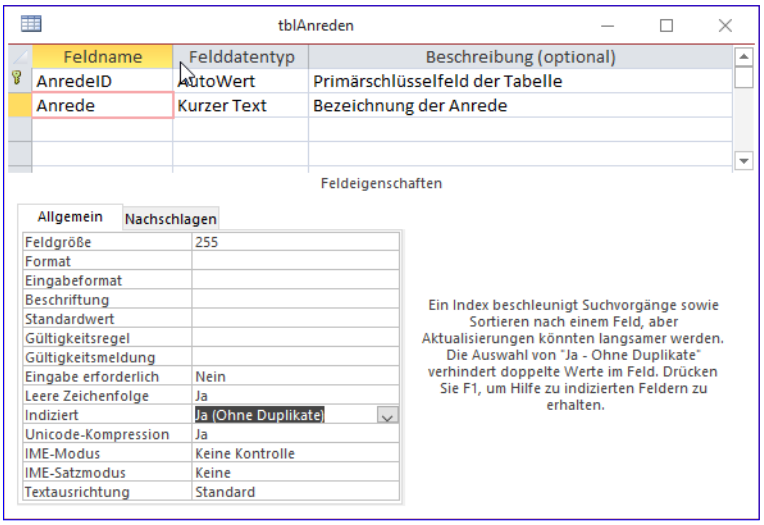

Bild 2: Zu migrierende Tabelle

Hier würden wir von einer automatischen Erfassung des Tabellennamens, der Felder und des Primärschlüssels ausgehen und von ein paar Anpassungen, um die Migration einigermaßen tauglich zu gestalten und beispielsweise nicht das Präfix **tbl** mitzuschleppen. Wir sehen hier, dass die Klasse beispielsweise **Anrede** heißt. Diese Bezeichnung können wir nur schwer aus dem Tabellennamen extrahieren, also holen wir ihn aus dem Namen des Primärschlüsselfeldes (hier **AnredeID**), von dem wir lediglich den Zusatz **ID** entfernen. Die beiden Felder belassen wir im ersten Schritt bei **AnredeID** und **Anrede**. Optimaler wäre beispielsweise **ID** und **Name**. Ersteres, weil wir in der objektorientierten Programmierung etwa über **<Objektname>.<Feldname>** auf die Eigenschaften einer Entität zugreifen und **Anrede.AnredeID** schlicht redundante Daten enthält – **Anrede.ID** wäre viel schöner. Und **Anrede.Anrede** ist ebenfalls optimierungsfähig. Für die Deklaration des **DbSet**-Objekts verwenden wir aktuell den Namen der Tabelle ohne das Präfix **tbl** und für den Typ der im **DbSet** enthaltenen Daten verwenden wir wieder den aus dem Primärschlüsselfeld ermittelten Namen, also **Anrede**. Gerade bei den Bezeichnern für die DbSets und die Entitäten muss man darauf achten, ob sich Plural und Singular unterscheiden. Das ist beispielsweise bei **Artikel** schwierig (ein Artikel/zwei Artikel), weshalb man, wenn man schon weiß, dass man mal objektorientiert mit dem Datenmodell seiner Datenbank arbeiten möchte, besser gleich eine Bezeichnung wie **Produkt**/**Produkte** verwendet. In unserer Beispieldatenbank verwenden wir aber weiter **tblArtikel**, gerade weil wir auch zeigen wollen, wie Sie solche Probleme durch Umbenennungen lösen können.

Beispieldatenbank

Die Beispieldatenbank enthält die im Beziehungen-Fenster von Access abgebildeten Tabellen und Beziehungen. Diese wollen wir nun, möglichst ohne manuellen Eingriff, in die entsprechenden Entitäten und DbSets umwandeln. Dazu können wir zwei Wege gehen: Entweder wir greifen von einem Visual Studio-Projekt aus auf die Datenbank zu oder wir bauen unsere Routinen direkt in der Access-Datenbank. Mir selbst geht VBA beim Zugriff auf Tabellen, Felder, Beziehungen und so weiter immer noch viel schneller von der Hand als mit VB oder C#. Außerdem wäre der Zugriff, wie er für das Auslesen der Access-Tabellen nötig wäre, eine Technik, die wir prinzipiell unter .NET nicht mehr benötigen – hier wollen wir Datenbanksysteme wie SQL Server oder SQLite nutzen, auf die wir mit dem Entity Framework zugreifen können.

Außerdem dürfte das Programmieren von Routinen, die uns aus dem Access-Datenmodell ein Entity Data Model macht, mit einer Menge Testen und Ändern verbunden sein. Das geht unter Access wesentlich schneller und komfortabler als in Visual Studio. In Access schreiben wir einfach die Prozeduren und klicken auf **F5**, um diese auszuführen, in Visual Studio müssten wir

immer das Projekt neu starten, die Ausführung prüfen, das Projekt wieder beenden, den Code anpassen und das Ganze immer wieder von vorn beginnen. Also entscheiden wir uns in diesem Fall für die ältere, aber pragmatischere Vorgehensweise.

.NET-Projekt zum Testen vorbereiten

Damit wir die gleich unter Access generierten Klassen im .NET-Projekt auf ihre Tauglichkeit prüfen können, legen wir gleich ein neues Projekt des Typs **VB\WPF-App** mit dem Namen **AccessZuEF** an. Diesem fügen wir ein neues Objekt des Typs **ADO.NET Entity Data Model** namens **BestellverwaltungContext** mit dem Typ **Leeres Code First-Modell** hinzu. Dies legt automatisch die Klasse **BestellverwaltungContext.vb** für uns an, der wir dann gleich unsere Liste der **DbSet**-Elemente hinzufügen können. Der Einfachheit halber packen wir unsere **Class**-Elemente zum Testen auch erst einmal hier herein.

Zugriff auf die Tabellen, Felder und Beziehungen

Den Zugriff auf die für uns interessanten Elemente gestalten wir mit der DAO-Bibliothek und VBA. Wir arbeiten uns nun Schritt für Schritt an die Umwandlung des Datenmodells und der enthaltenen Daten an die entsprechenden .NET-Klassen heran. Zuerst sehen wir uns ein paar Hilfsfunktionen an, die uns auf dem Weg behilflich sind.

Datentypen umwandeln

Unter dem Link **https://support.microsoft.com/de-de/help/320435/info-oledbtype-enumeration-vs-microsoft-access-data-types** finden wir eine Liste der Access-Datentypen und der entsprechenden Datentypen unter .NET. Daraus bauen wir uns eine kleine Funktion, die den Access-Datentyp als Parameter erwartet und den .NET-Datentyp als Ergebnis zurückliefert. Diese Funktion sieht wie folgt aus:

```
Public Function DataTypeNET(intDatatype As DataTypeEnum, strDatatype As String, bolArray As Boolean) As Boolean
    Dim bolDatatype As Boolean
    bolDatatype = True
    bolArray = False
    Select Case intDatatype
        Case dbText: strDatatype = "System.String"
        Case dbMemo: strDatatype = "System.String"
        Case dbByte: strDatatype = "System.Byte"
        Case dbBoolean: strDatatype = "System.Boolean"
        Case dbDate: strDatatype = "System.DateTime"
        Case dbCurrency: strDatatype = "System.Decimal"
        Case dbDecimal: strDatatype = "System.Decimal"
        Case dbDouble: strDatatype = "System.Double"
        Case dbGUID: strDatatype = "System.Guid"
        Case dbLong: strDatatype = "System.Int32"
        Case dbLongBinary
            strDatatype = "System.Byte"
            bolArray = True
        Case dbSingle: strDatatype = "System.Single"
        Case dbInteger: strDatatype = "System.Int16"
```

```
        Case dbBinary
            strDatatype = "System.Byte"
            bolArray = True
        Case Else
            bolDatatype = False
    End Select
    DataTypeNET = bolDatatype
End Function
```

Die Funktion erwartet drei Parameter, von denen nur der erste Parameter Daten an die Funktion schickt – die übrigen beiden liefern die Rückgabewerte. **intDataType** erwartet die Konstante beziehungsweise den entsprechenden Zahlenwert, den wir aus der Eigenschaft **Type** eines **Field**-Elements erhalten. **strDatatype** liefert den .NET-Datentyp als String zurück. Manche Datentypen werden in .NET als Byte-Array gestaltet. Wenn der Datentyp ein Array erfordert, markieren wir den dritten Parameter **bolArray** als **True**. Es kann auch sein, dass der Datentyp nicht vorhanden ist, was schlicht daran liegt, dass er in dieser Funktion nicht berücksichtigt wird – in diesem Fall kommt der eigentliche Rückgabewert der Funktion zum Einsatz, der den Datentyp **Boolean** hat.

Die Funktion stellt die Variable **bolDatatype**, die festlegt, ob die Funktion einen passenden Datentyp in .NET gefunden hat, standardmäßig auf **True** ein. Diese wird erst auf **False** eingestellt, wenn der Datentyp in der folgenden **Select Case**-Bedingung nicht gefunden wurde. Die Variable **bolArray** wird zuerst auf **False** eingestellt und erhält erst den Wert **True**, wenn der Datentyp ein Array benötigt. Das ist in diesem Fall nötig, da **bolArray** ja auch in der aufrufenden Prozedur verwendet wird und gegebenenfalls vom vorherigen Aufruf noch den Wert **True** enthält. Davon ab vergleicht die **Select Case**-Bedingung den mit **intDatatype** übergebenen Wert mit Konstanten wie **dbText**, **dbMemo**, **dbByte** und so weiter und trägt, wenn sie einen passenen Eintrag gefunden hat, den entsprechenden .NET-Datentyp in die Variable **strDatatype** ein. Am Ende wird schließlich der Wert von **bolDatatype** noch als Rückgabewert der Funktion festgelegt.

Primärschlüsselfeld ermitteln

Wir gehen davon aus, dass die Primärschlüsselfelder der Tabellen mit einer Bezeichnung versehen sind, die aus dem Singular der Bezeichnung besteht, die sich im Tabellennamen befindet, und der angehängten Zeichenkette **ID** – bei **tblAnreden** also etwa **AnredeID** (das ist zumindest die Konvention, die ich seit 20 Jahren in Magazin **Access im Unternehmen** und in all meinen anderen Veröffentlichungen über Access verwende). Um den Namen herauszubekommen, müssen wir nun allerdings noch zuverlässig bestimmen können, welches Feld das Primärschlüsselfeld ist. Wir gehen an dieser Stelle davon aus, dass jede Tabelle auch nur einen aus einem Feld bestehenden Primärschlüssel verwendet.

Die folgende Funktion namens **GetPrimaryKeys** ermittelt dennoch die Namen aller Primärschlüsselfelder einer Tabelle. Dazu erhält sie mit **strTable** den Tabellennamen als Parameter sowie **strPKs** als Rückgabeparameter. Die Funktion lädt ein **TableDef**-Objekt auf Basis des Namens der Tabelle und durchläuft dann alle **Index**-Elemente dieser Tabelle in einer **For Each**-Schleife. Wenn es sich bei dem Index um einen Primärschlüssel handelt (**Primary = True**), durchläuft die Funktion alle Felder des Indexes und fügt die Namen der Felder in der Variablen **strPKTemp** durch Semikola getrennt aneinander. Das Ergebnis wird mit dem Parameter **strPKs** zurückgeschickt. Der Rückgabewert der Funktion **GetPrimaryKeys** wird mit dem Wert **True** gefüllt, wenn es mindestens ein Primärschlüsselfeld gibt.

```
Public Function GetPrimaryKeys(strTable As String, strPKs As String) As Boolean
     Dim db As DAO.Database
     Dim tdf As DAO.TableDef
     Dim idx As DAO.Index
     Dim strPKTemp As String
     Dim i As Integer
     Set db = CurrentDb
     Set tdf = db.TableDefs(strTable)
     For Each idx In tdf.Indexes
         If idx.Primary Then
             For i = 0 To idx.Fields.Count - 1
                 strPKTemp = strPKTemp & idx.Fields(0).Name & ";"
             Next i
         End If
     Next idx
     If Len(strPKTemp) = 0 Then
        Exit Function
     End If
     strPKs = Left(strPKTemp, Len(strPKTemp) - 1)
     GetPrimaryKeys = True
End Function
```

Die folgende Funktion **GetPrimaryKey** unterscheidet sich im Namen nur durch den Singular gegenüber dem Plural von der Funktion **GetPrimaryKey**. Was macht diese Funktion? Sie ruft in erster Linie die zuvor beschriebene Funktion **GetPrimaryKeys** auf und prüft, ob die mit dem Parameter **strTable** übergebene Tabelle nur um einen oder mehrere Primärschlüssel enthält. Falls es mehrere sind, wird mit **strPK** eine leere Zeichenkette und mit dem Rückgabewert der Wert **False** zurückgegeben. Falls es sich um ein einziges Primärschlüsselfeld handelt, wird mit **strPK** der Name des Primärschlüsselfeldes sowie mit dem Rückgabewert der Wert **True** zurückgeliefert:

```
Public Function GetPrimaryKey(strTable As String, strPK As String) As Boolean
    Dim strPKs As String
    If GetPrimaryKeys(strTable, strPKs) = True Then
        If InStr(1, strPKs, ";") Then
            Exit Function
        Else
            strPK = strPKs
            GetPrimaryKey = True
        End If
    Else
        Exit Function
    End If
End Function
```

In der aufrufenden Prozedur, die wir uns gleich im Anschluss ansehen, können wir dann aufgrund der Rückgabewerte entweder die Definition der Klasse und der **DbSet**-Anweisung zusammenstellen oder eine Meldung ausgeben, dass die Bedingungen für das Erstellen der Elemente wegen keinem Primärschlüsselfeld oder mehr als einem Primärschlüsselfeld nicht gegeben sind.

Klassen und DbSet-Deklarationen zusammenstellen

Die folgende Prozedur erledigt den eigentlichen Teil der Arbeit: Sie nutzt die Ergebnisse der zuvor beschriebenen Funktionen und weitere Daten, um die Klassendefinitionen und die **DbSet**-Deklarationen zusammenzustellen. Im ersten Schritt wollen wir diese noch mit **Debug.Print** im Direktbereich ausgeben. Später werden wir direkt entsprechende Dateien daraus erzeugen, die Sie nur noch aus dem Windows-Explorer in den Projektmappen-Explorer des Ziel-Projekts in Visual Studio ziehen. Die Prozedur heißt **EDMErstellen** und durchläuft nach dem Deklarationsteil zunächst einmal alle **TableDef**-Elemente, also alle Tabellendefinitionen, der mit **CurrentDb** referenzierten Datenbank – also der aktuell geöffneten Datenbank:

```
Public Sub EDMErstellen()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field
    Dim strDbSets As String
    Dim strEntity As String
    Dim strPK As String
    Dim strDatatype As String
    Dim bolArray As Boolean
    Dim strEntities As String
    Dim bolDatatypeExists As Boolean
    Dim strArray As String
    Set db = CurrentDb
    For Each tdf In db.TableDefs
```

Hier prüft die Prozedur dann, ob der Name der Tabelle nicht mit **MSys** beginnt, denn das sind die Systemtabellen, die wir nicht berücksichtigen wollen. Beginnt die Tabelle nicht mit **MSys**, prüfen wir, ob die Tabelle genau ein Primärschlüsselfeld enthält. Falls nicht, geben wir im Direktbereich einen entsprechenden Hinweis aus, den wir als Kommentar kennzeichnen:

```
        If Not Left(tdf.Name, 4) = "MSys" Then
            strPK = ""
            If Not GetPrimaryKey(tdf.Name, strPK) Then
                Debug.Print "    '****"
                Debug.Print "    'Mehrere PKs in " & tdf.Name & ". Die Klasse wird nicht erstellt."
```

Anderenfalls, also wenn genau ein Primärschlüsselfeld vorhanden ist, ermitteln wir den Namen der Entität aus dem Namen des Primärschlüsselfelds, von dem wir zuvor die Zeichenkette **ID** entfernen (also etwa **Anrede** statt **AnredeID**). Den Namen für die **DbSet**-Auflistung entnehmen wir dem Tabellennamen und entfernen dort das Präfix **tbl** (also **Anreden** statt **tblAnreden**). Die Zeile für das **DbSet** können wir damit schon zusammenstellen. Wir sammeln diese Zeilen in der Variablen **strDbSets**:

```
            Else
                strEntity = Replace(strPK, "ID", "")
                strEntities = Replace(tdf.Name, "tbl", "")
                strDbSets = strDbSets & "Public Overridable Property " & tdf.Name & "() As DbSet(Of " & strEntity & ")" _
                    & vbCrLf
```

Danach starten wir mit der Ausgabe der Klassendefinition und beginnen mit Public Class und dem Namen der Entität:

```
            Debug.Print "Public Class " & strEntity
```

Danach durchlaufen wir in einer **For Each**-Schleife alle Felder der aktuell in **tdf** befindlichen Tabelle und ermitteln mit der Funktion **DataTypeNET** den Datentyp für .NET für das aktuelle Feld. Wenn **bolArray** den Wert **True** hat, tragen wir in die Variable **strArray** die Zeichenkette **()** ein, sonst eine leere Zeichenkette. Dann prüfen wir, ob **bolDataTypeExists** den Wert **True** hat und geben die Zeile für die Deklaration der Eigenschaft dieses Feldes im Direktbereich aus.

Falls nicht, tragen wir hier einen Text ein, der darauf hinweist, dass kein Datentyp für dieses Feld gefunden werden konnte:

```
            For Each fld In tdf.Fields
                bolDatatypeExists = DataTypeNET(fld.Type, strDatatype, bolArray)
                strArray = IIf(bolArray, "()", "")
                If bolDatatypeExists Then
                    Debug.Print "    Public Property " & fld.Name & strArray & " As " & strDatatype
                Else
                    Debug.Print "    'Kein Datentyp für " & fld.Name & " gefunden."
                End If
            Next fld
```

Nachdem wir alle Felder der aktuellen Tabelle durchlaufen haben, schließen wir die Definition der Klasse mit **End Class** im Direktbereich ab und fahren mit der nächsten Tabelle fort. Anschließend geben wir noch die in **strDbSets** gesammelten Deklarationen der DbSets aus:

```
            Debug.Print "End Class"
        End If
    End If
Next tdf
Debug.Print strDbSets
End Sub
```

Wenn wir die Prozedur nun aufrufen, erhalten wir im Direktbereich schon recht ansehnliche Ergebnisse, die wir so auch einmal in die Kontext-Klasse in unserem dafür vorbereiteten .NET-Projekt einfügen können – allein um zu prüfen, ob die enthaltenen Informationen schon einmal die korrekte Syntax aufweisen.

Das Ergebnis sieht dann beispielsweise wie in Bild 3 aus. Zumindest werden keine Fehlermarkierungen angezeigt – das ist schon einmal positiv!

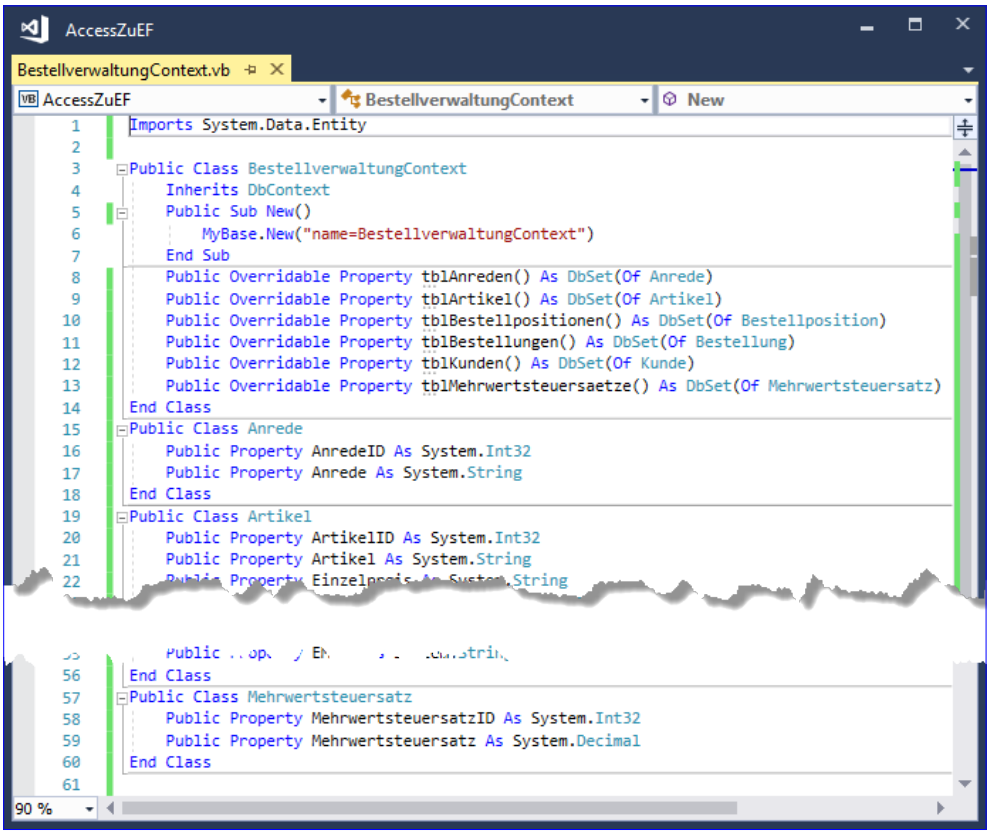

Bild 3: Erster Test mit den Definitionen im .NET-Projekt

Code First-Datenbank intialisieren

An dieser Stelle können wir, auch wenn natürlich noch einige Informationen fehlen, ja auch durchaus schon einmal probieren, die Code First-Datenbank zu initialisieren. Wie das im Detail gelingt, erfahren Sie im Artikel **Entity Framework: Datenbankinitialisierung**. Wir machen die Kurzfassung: Sie blenden die Paket-Manager-Konsole in Visual Studio ein und geben zunächst diesen Befehl ein:

```
Enable-Migrations
```

Das Ergebnis ist etwas ernüchternd und liefert uns noch einige Zusatzarbeit:

Während der Modellgenerierung wurde mindestens ein Überprüfungsfehler erkannt:

```
Anrede: Name: Der Name 'Anrede' kann nicht in Typ 'AccessZuEF.Anrede' verwendet werden. Elementnamen dürfen nicht mit dem einschließenden Typ übereinstimmen.
Artikel: Name: Der Name 'Artikel' kann nicht in Typ 'AccessZuEF.Artikel' verwendet werden. Elementnamen dürfen nicht mit dem einschließenden Typ übereinstimmen.
Mehrwertsteuersatz: Name: Der Name 'Mehrwertsteuersatz' kann nicht in Typ 'AccessZuEF.Mehrwertsteuersatz' verwendet werden. Elementnamen dürfen nicht mit dem einschließenden Typ übereinstimmen.
```

Wir können also keine Eigenschaftsnamen verwenden, die genauso lauten wie das Element selbst. Wenn also die Entität **Anrede** heißt, darf die Entität keine Eigenschaft namens **Anrede** enthalten. Wie können wir das im Code berücksichtigen? Wir können hier schon eine der weiter oben als Optimierung betrachtete Änderung vornehmen. Genau genommen hat die Meldung uns auch einen Hinweis geliefert, wie wir herausfinden, welches der Felder wir mit der Bezeichnung **Name** versehen können

Von Access zu Entity Framework: Daten

Viele Leser dieses Magazins programmieren auch mit Access. Daher haben wir im Artikel »Von Access zu Entity Framework: Datenmodell« bereits gezeigt, wie Sie die meisten Elemente eines Datenmodells in Klassen für ein Entity Data Model überführen, die Sie dann wiederum zum Erstellen einer SQL Server-Datenbank per Code First nutzen können. Was fehlt, sind allerdings noch die Daten in diesen Tabellen. Wie Sie den Code erstellen, um auch die Daten über eine entsprechend Seed-Methode in die Datenbank zu schreiben, erfahren Sie in diesem Artikel.

Im Artikel **Von Access zu Entity Framework: Datenmodell** haben wir uns darum gekümmert, das Datenmodell unserer kleinen Access-Beispielanwendung aus Bild 1 zuerst in die Klassen eines Entity Data Models zu überführen. Dann haben wir mit den Methoden **Add-Migration** und **Update-Database** in der Paket-Manager-Konsole dafür gesorgt, dass diese Klassen zusammen mit den passenden **DbSet**-Auflistungen in eine neue SQL Server-Datenbank überführt wurden. Die dort vorgestellten Routinen sind sicher noch nicht perfekt, aber sie sind eine gute Grundlage, die Sie selbst erweitern können – oder Sie teilen uns einfach mit, bei welchen Eigenarten von Tabellendefinitionen oder Datenmodellen die Erstellung des Entity Data Models oder der SQL Server-Datenbank noch nicht funktioniert.

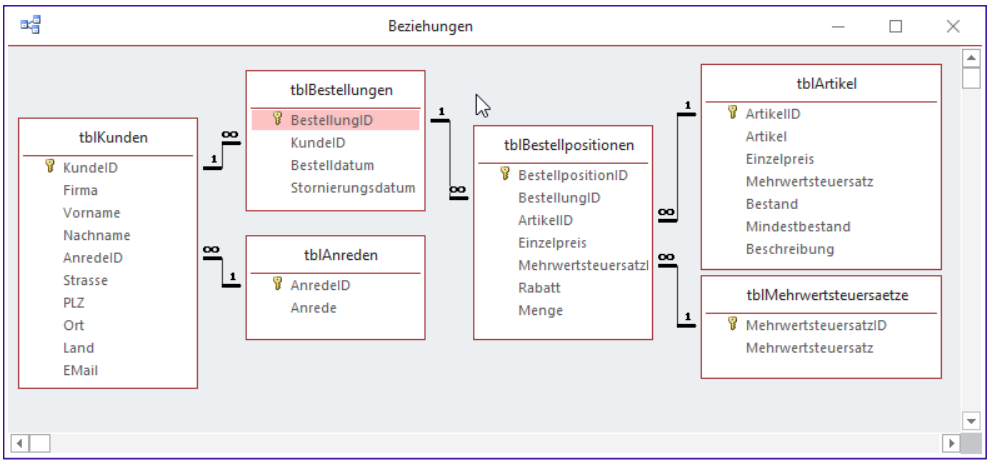

Bild 1: Beispiel für ein zu migrierendes Datenmodell

Im vorliegenden Artikel nun wollen wir die Daten in diesen Tabellen in eine Form bringen, in der wie diese beim Erstellen der Datenbank von Visual Studio aus mit der **Seed**-Methode direkt in die frisch angelegten Tabellen schreiben können. Die Grundlagen dafür erhalten Sie im Artikel **Entity Framework: Tabellen füllen** sowie in den beiden Artikeln **Entity Framework: Datenbankinitialisierung** und **Entity Framework: Datenbankmigration**.

Anlegen der Daten für eine einzelne Tabelle ohne Fremdschlüsselfelder

Wir wollen uns langsam an die Lösung der Aufgabe heranbewegen und uns zuerst einmal eine einfache Tabelle heraussuchen, deren Daten wir dann über den Weg der **Seed**-Methode etwa der **Configuration**-Klasse im **Migrations**-Ordner nach dem Erstellen des Datenmodells in die Datenbank schreiben (wie Sie diesen Ordner erstellen, erfahren Sie im Artikel **Entity Framework: Datenbankmigration**).

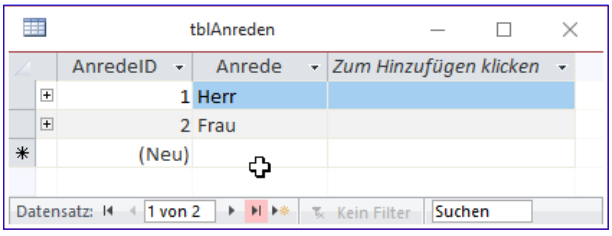

Bild 2: Die Tabelle **tblAnreden** der Ausgangsdatenbank

Die Tabelle **tblAnreden** unserer Ausgangsdatenbank enthält nur zwei Datensätze (siehe Bild 2). Diese wollen wir nun in einem ersten Schritt in Anweisungen schreiben, die wir in die **Seed**-Methode der .NET-Anwendung schreiben können. Wir schauen uns erst einmal an, wie die fertige **Seed**-Methode anschließend aussehen soll:

```
Namespace Migrations
    Friend NotInheritable Class Configuration
        Inherits DbMigrationsConfiguration(Of BestellverwaltungContext)
        Public Sub New()
            AutomaticMigrationsEnabled = False
        End Sub
        Protected Overrides Sub Seed(context As BestellverwaltungContext)
            context.Anreden.AddOrUpdate(Function(x) x.ID,
                New Anrede() With {.Name = "Herr"},
                New Anrede() With {.Name = "Frau"}
            )
        End Sub
    End Class
End Namespace
```

Wir sollten uns überlegen, ob wir per VBA immer die komplette Klasse namens **Configuration** wie oben abgebildet erstellen sollen oder nur die **Seed**-Methode in dieser Klasse. Praktischer ist es vermutlich, gleich die komplette Klasse zu erstellen, damit man entweder – im ersten Schritt – die Ausgabe dieser Klasse aus dem Direktbereich des VBA-Editors von Access komplett kopieren und als Ganzes in diese Klasse einsetzen kann. Im zweiten Schritt könnten wir den Code direkt in die entsprechende Datei schreiben, die dann von Visual Studio aktualisiert eingelesen wird.

Die **Seed**-Methode enthält hier eine Anweisung, bei der wir die **AddOrUpdate**-Methode der Auflistung **Anreden** aufrufen und dieser eine Funktion übergeben, welche jeweils ein neues Objekt auf Basis der Klasse **Anrede** enthält. Für diese legen wir dann jeweils die Anrede fest.

Wir bauen zunächst eine Prozedur, welche den Rahmen im Direktfenster ausgibt, also die **Imports**-Anweisungen, den Namespace, die Klasse und die enthaltenen Methoden – mit Ausnahme der **Seed**-Methode:

```
Public Sub SeedErstellen()
    Dim strSeed As String
    strSeed = strSeed & "Imports System" & vbCrLf
    strSeed = strSeed & "Imports System.Data.Entity"
    strSeed = strSeed & "Imports System.Data.Entity.Migrations"
    strSeed = strSeed & "Imports System.Linq"
    strSeed = strSeed & ""
    strSeed = strSeed & "Namespace Migrations"
    strSeed = strSeed & "    Friend NotInheritable Class Configuration"
```

```
    strSeed = strSeed & "        Inherits DbMigrationsConfiguration(Of BestellverwaltungContext)"
    strSeed = strSeed & "        Public Sub New()"
    strSeed = strSeed & "            AutomaticMigrationsEnabled = False"
    strSeed = strSeed & "        End Sub"
    strSeed = strSeed & "        Protected Overrides Sub Seed(context As BestellverwaltungContext)"
    strSeed = strSeed & SeedData
    strSeed = strSeed & "        End Sub"
    strSeed = strSeed & "    End Class"
    strSeed = strSeed & "End Namespace"
End Sub
```

Das ist noch kein Hexenwerk. Interessant ist die hier eingebettete Prozedur **SeedData**. Diese stellt die Anweisungen zusammen, die wir zur Methode **Seed** hinzufügen wollen. Statisch sieht das für unsere Tabelle **tblAnreden** zunächst wie folgt aus:

```
Public Sub SeedData()
    Dim strSeedData As String
    strSeedData = strSeedData & "            context.Anreden.AddOrUpdate(Function(x) x.ID,"
    strSeedData = strSeedData & "                New Anrede() With {.Name = ""Herr""},"
    strSeedData = strSeedData & "                New Anrede() With {.Name = ""Frau""}"
    strSeedData = strSeedData & "            )"
    SeedData = strSeedData
End Sub
```

Um dies aus VBA heraus dynamisch für die Tabelle **tblAnreden** zu erzeugen, müssen wir dies allerdings mit einigen erst zur Laufzeit ermittelten Daten füllen. Das erledigen wir in den folgenden Schritten.

Seed für mehrere Tabellen

Die Prozedur **SeedData** wandeln wir in eine Funktion um, die das Ergebnis an die aufrufende Prozedur zurückgeben soll. Sie enthält dafür für jede Tabelle der Datenbank einen Aufruf der Tabelle **SeedData** und fügt das Ergebnis jeweils an die Zeichenkette **strSeed** an. Die Aufrufe der Tabellen erledigen wir in einer **For Each**-Schleife über alle **TableDef**-Elemente der mit **CurrentDb** referenzierten aktuellen Datenbank. In der Schleife prüfen wir zunächst, ob es sich bei der aktuellen Tabelle nicht um eine Systemtabelle von Access handelt. Nur wenn das nicht der Fall ist, rufen wir die Funktion **SeedData** mit dem Namen der Tabelle auf. Den Namen der Tabelle ermitteln wir dabei mit der Eigenschaft **Name** des **TableDef**-Objekts.

```
Public Sub SeedErstellen()
    ...
    strSeed = strSeed & "        Protected Overrides Sub Seed(context As BestellverwaltungContext)" & vbCrLf
    For Each tdf In db.TableDefs
        If Not Left(tdf.Name, 4) = "MSYS" Then
            strSeed = strSeed & SeedData(tdf.Name) & vbCrLf
        End If
```

```
    Next tdf    strSeed = strSeed & "        End Sub" & vbCrLf
    strSeed = strSeed & "    End Class" & vbCrLf
    strSeed = strSeed & "End Namespace"
    Inzwischenablage strSeed
End Sub
```

Am Ende der Prozedur nutzen wir die Prozedur **InZwischenablage**, um den Inhalt der Variablen **strSeed**, in der wir die Anweisungen für die **Configuration.vb**-Klasse gesammelt haben, in die Zwischenablage zu kopieren. Das ist eine bessere Lösung als die Ausgabe im Direktbereich, weil dieser nur eine begrenzte Anzahl Zeichen aufnehmen kann und bei vielen Daten der obere Teil der Ausgabe dann nicht mehr verfügbar ist. Die Prozedur **InZwischenablage** finden Sie im Modul **mdlZwischenablage**.

Die Funktion SeedData zum Zusammenstellen der AddOrUpdate-Aufrufe

Die Funktion **SeedData** bohren wir gegenüber der vorherigen Version erheblich auf. Als Erstes fällt der Parameter **strTabelle** auf, welcher die Information enthält, für die Datensätze welcher Tabelle die **AddOrUpdate**-Methoden zusammengestellt werden sollen.

```
Public Function SeedData(strTabelle As String) As String
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strSeedData As String
    Dim strPK As String
    Dim strFelder As String
    Dim fld As DAO.Field
    Dim strEntity As String
    Dim strEntities As String
    Dim strProperty As String
```

Nach der Deklaration der Variablen speichern wir einen Verweis auf die aktuelle Datenbank in der Variablen **db**. Dann prüfen wir mit der Funktion **GetPrimaryKey**, die wir bereits im Artikel **Von Access zu Entity Framework: Datenmodell** erläutert haben, ob die gerade untersuchte Tabelle genau ein Primärschlüsselfeld bereitstellt. Ist das der Fall, wird eine entsprechende Meldung im Direktbereich ausgeben und die Funktion wird mit **Exit Function** verlassen:

```
    Set db = CurrentDb
    If Not GetPrimaryKey(strTabelle, strPK) Then
        Debug.Print "    'Mehrere PKs in " & strTabelle & ". Die Klasse wird nicht erstellt."
        Exit Function
```

Kann jedoch genau ein Primärschlüsselfeld gefunden werden, landet dies in der Variablen **strPK**. Daraus ermitteln wir zunächst den Namen der Entität, indem wir hinten die Zeichenkette **ID** entfernen. Außerdem ermittelten wir den Namen der Tabelle, indem wir vom Tabellenname vorn **tbl** abtrennen und das Ergebnis in der Variablen **strEntitaeten** speichern:

```
Else
    strEntity = Replace(strPK, "ID", "")
    strEntities = Replace(strTabelle, "tbl", "")
```

Damit starten wir in die Ermittlung der benötigten Daten, indem wir ein Recordset auf Basis der mit **strTabelle** übergebenen Tabelle erstellen. Außerdem tragen wir die erste Zeile in die Variable **strSeedData** ein, welche zum Beispiel bei der Tabelle **tblAnreden** nun **context.Anreden.AddOrUpdate(Function(x) x.ID,** lautet:

```
Set rst = db.OpenRecordset("SELECT * FROM " & strTabelle, dbOpenDynaset)
strSeedData = strSeedData & "            context." & strEntities & ".AddOrUpdate(Function(x) x.ID," & vbCrLf
```

Danach durchlaufen wir in einer **Do While**-Schleife alle Datensätze der aktuell bearbeiteten Tabelle. Im ersten Schritt legen wir die Zeile **New Anrede() With {** an:

```
Do While Not rst.EOF
    strSeedData = strSeedData & "                New " & strEntity & "() With {"
```

Dann leeren wir die Variable **strFelder**, die wir anschließend mit den Zuweisungen für die einzelnen Felder mit den Werten des Recordsets füllen:

```
    strFelder = ""
```

In einer **For Each**-Schleife über alle Felder des Recordsets durchlaufen wir dann ein Feld nach dem anderen. Dabei prüfen wir zuerst, ob es sich beim aktuellen Feld nicht um das Primärschlüsselfeld handelt. Falls nicht, prüfen wir dann, ob der Feldname dem Wert aus **strEntity** entspricht. Das ist zum Beispiel beim Feld **Anrede** der Tabelle **tblAnreden** der Fall. Das würde bedeuten, dass das Feld genauso heißen würde wie die Entität, also **Anrede**. Wie wir spätestens im Artikel **Von Access zu Entity Framework: Datenmodell** gelernt haben, ist das allerdings per Definition nicht erlaubt. Für diesen Fall haben wir in dem Artikel vorgesehen, dass wir das Feld dann einfach **Name** nennen (man hätte auch **Bezeichnung** nehmen können). Um dem gerecht zu werden, verwenden wir hier die Variable **strProperty**, die wir dann entweder mit dem tatsächlichen Feldnamen füllen oder, wenn der Feldname dem Entitätsnamen entspricht, einfach mit **Name**:

```
    For Each fld In rst.Fields
        If Not fld.Name = strPK Then
            If Not fld.Name = strEntity Then
                strProperty = fld.Name
            Else
                strProperty = "Name"
            End If
```

Danach machen unterscheiden wir in einer **Select Case**-Bedingung nach dem Datentyp der Felder. Hier benötigen wir beispielsweise einen **Case**-Zweig für Zahlen-Datentypen, einen für Text-Datentypen, einen für das Datum und einen für Währun-

Von Access zu EF: Step by step

Im Artikel »Von Access zu Entity Framework: Datenmodell« zeigen wir, wie Sie von Access aus auf Basis des aktuellen Datenmodells Klassen für ein Entity Data Model erstellen. In »Von Access zu Entity Framework: Daten« zeigen wir, wie Sie noch eine Seed-Anweisung hinzufügen, welche die Daten der Access-Datenbank in die auf Basis des Entity Data Models erstellte Datenbank füllt. Im vorliegenden Artikel nun liefern wir nochmal eine Kurzanleitung, wie Sie die in den Artikeln vorgestellten Techniken Schritt für Schritt nutzen, um ein .NET-Projekt mit Entity Data Model samt Daten aus der Access-Datenbank zu erstellen.

Um aus einer Access-Datenbank wie in Bild 1 ein Entity Data Model zu machen, daraus eine SQL Server-Datenbank zu generieren und dies auch noch mit den Daten der Access-Datenbank zu füllen ist gar nicht so schwer, wenn Sie die Lösungen aus den oben genannten Artikeln verwenden.

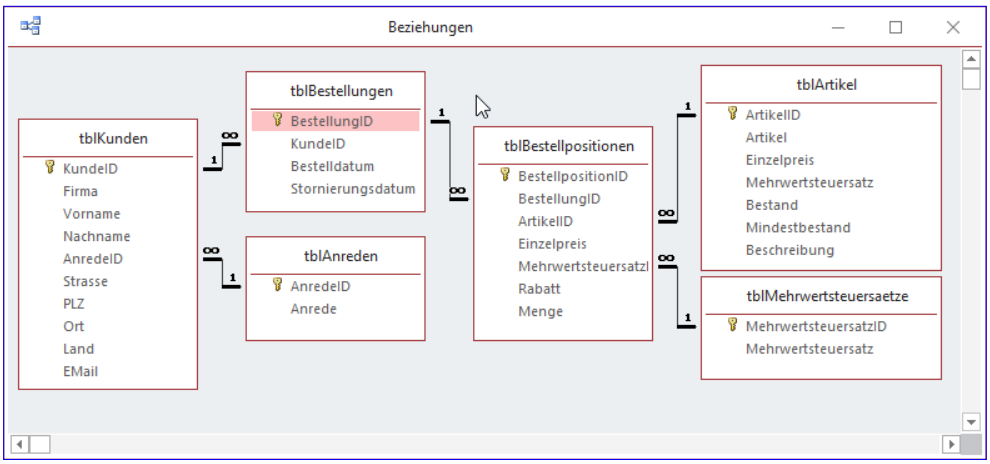

Bild 1: Zu migrierende Access-Datenbank

Projekt erstellen

Als Erstes erstellen wir ein Projekt auf Basis der Vorlage **Visual Basic|Windows Desktop|WPF-App** und nennen es beispielsweise **AccessZuEF**. Dann fügen wir diesem über den Kontextmenü-Eintrag **Hinzufügen|Neues Element...** des Projekt-Elements im Projektmappen-Explorer ein neues Element des Typs **ADO.NET Entity Data Model** hinzu, etwa namens **Bestellverwaltung**Context. Im **Assistent für Entity Data Model** wählen wir für das Modell den Typ **Leeres Code First-Modell** aus. Damit sind die Arbeiten am Projekt auf .NET-Ebene erst einmal erledigt.

Entity Data Model-Code erstellen

In der Access-Datenbank, die Sie mit dem Code aus der Beispieldatenbank ausstatten, rufen wir nun den Befehl **EDMErstellen** aus dem Modul **mdlEDM** auf. Die mittlerweile optimierte Version dieser Prozedur speichert das Ergebnis direkt in der Zwischenablage. Sie brauchen nach dem Aufruf also nur noch zu Visual Studio zu wechseln, dort die Klasse **BestellverwaltungContext.vb** zu öffnen und dort ganz unten unter der Klasse **BestellverwaltungContext** den Inhalt der Zwischenablage einzufügen.

Die unteren paar Zeilen enthalten die **DbSet**-Deklarationen, die Sie ausschneiden und in die Klasse **BestellverwaltungContext** einfügen. Darunter finden Sie die benötigten Namespace-Verweise, die Sie ausschneiden und ganz oben über der Klassendefinition ergänzen können.

Seed-Methode erstellen

Danach wechseln Sie zum Modul **mdlEDMData** im VBA-Editor der Access-Anwendung. Hier rufen Sie die Prozedur **SeedErstellen** auf. Diese kopiert ihr Ergebnis ebenfalls in die Zwischenablage. Um diese im .NET-Projekt nutzen zu können, müssen Sie zuerst die Migrationen aktivieren. Dazu zeigen Sie die Paket-Manager-Konsole in Visual Studio an und setzen dort die folgende Anweisung ab:

```
Enable-Migrations
```

Dadurch wird ein neuer Ordner namens **Migrations** angelegt, der eine Klasse namens **Configuration** enthält. Diese Klasse enthält dann eine Methode namens **Seed**. Diese markieren Sie und ersetzen sie komplett durch den Inhalt der Zwischenablage.

Migration definieren

Damit sind Sie bereits soweit. Sie können nun mit der folgenden Anweisung die Änderungen definieren, die durch das Übertragen der Entity Data Model-Klassen in die Datenbank nötig werden:

```
Add-Migration Init
```

Sollte dies nicht gelingen, finden Sie in der Regel Fehlermeldungen im Bereich **Fehlerliste**. Anderenfalls fügt dieser Befehl eine neue Datei zum Ordner **Migrations** hinzu. Diese enthält alle Anweisungen für das Erstellen der Elemente der Datenbank. Die Klasse **Configuration.vb** enthält mit der Seed-Methode alles, was nach dem Erstellen der Tabellen zum Eintragen der Daten in die Tabellen notwendig ist.

Migration durchführen

Sie können also nun mit dem folgenden Befehl die Migration durchführen und so die Tabellen in einer neuen Datenbank anlegen. Die **Seed**-Methode wird dabei automatisch aufgerufen und füllt die neuen Tabellen mit den Daten:

```
Update-Database
```

Wenn nun alles funktioniert hat, sind alle Schritte erledigt. Sie haben nun auf Basis der Tabellen einer Access-Datenbank sowohl ein Entity Data Model samt den **DbSet**-Definitionen angelegt als auch eine neue Datenbank mit den Tabellen und Daten aus der Access-Datenbank erstellt.

Zusammenfassung und Ausblick

Die hier beschriebene Vorgehensweise kann Ihnen eine Menge Arbeit sparen, wenn Sie schnell ein Entity Data Model auf Basis einer Access-Datenbank erstellen und die Datenbank mit Tabellen und Daten einrichten wollen. Sie können nach dem Aufruf zweier VBA-Prozeduren sowie dem Einfügen der Inhalte der durch diese in die Zwischenablage geschriebenen Code-Zeilen mit drei Befehlen die fehlenden Schritte zum Erstellen der Datenbank durchführen. Anschließend können Sie die Benutzeroberfläche auf Basis des so bereitgestellten Entity Data Models samt zugrunde liegender Datenbank aufbauen.

EDM: DataGrid als Datenblatt

Wer von Access kommt und Datenbankanwendungen mit Visual Studio programmieren möchte, vermisst vermutlich das einfache Datenblatt, das man unter Access mit wenigen Mausklicks zusammenstellen konnte. Dieses zeigt nicht nur die Datensätze der als Datenherkunft verwendeten Tabelle oder Abfrage an, sondern bietet auch die Möglichkeit, die enthaltenen Daten zu ändern, zu löschen oder durch neue Datensätze zu ergänzen. Dies wollen wir in diesem Artikel durch den Einsatz eines entsprechend programmierten DataGrid-Steuerelements nachbauen.

Beispieldaten

In den Artikeln **Von Access zu Entity Framework: Datenmodell** und **Von Access zu Entity Framework: Daten** haben wir gezeigt, wie Sie ein Entity Data Modell auf Basis eines Access-Datenmodells erstellen und die Daten für die daraus zu erzeugenden Tabellen in eine **Seed**-Methode schreiben. Im Beispielprojekt dieses Artikels haben wir das Entity Data Model und die Klasse mit der **Seed**-Methode bereits angelegt, sodass Sie nach dem Kopieren des Projektordners auf Ihren Rechner einfach nur den die Anweisung Update-Database in der Paket-Manager-Konsole ausführen müssen. Dadurch sollte automatisch eine Datenbank für den in der Verbindungszeichenfolge angegebenen SQL Server, hier **LocalDb**, angelegt und mit den angegebenen Tabellen und Daten gefüllt werden.

Fenster mit DataGrid anlegen

Wir wollen gleich ein eigenes Fenster für dieses Beispiel anlegen, das Sie dann über eine Schaltfläche vom Fenster **MainWindow** aus aufrufen können. Dazu legen wir die folgende Schaltfläche in diesem Fenster an:

```
<Grid>
    <Button x:Name="btnProdukteDataGrid" Click="btnProdukteDataGrid_Click">DataGrid mit Produkten</Button>
</Grid>
```

Die Schaltfläche soll die folgende Ereignismethode aufrufen:

```
Private Sub btnProdukteDataGrid_Click(sender As Object, e As RoutedEventArgs)
    Dim wnd As Window
    wnd = New ProdukteDataGrid
    wnd.ShowDialog()
End Sub
```

Das neue Fenster nennen wir **ProdukteDatagrid.xaml**. Zunächst legen wir ein paar Elemente in der Code behind-Klasse an, der wir zunächst die Konstruktor-Methode hinzufügen:

```
Public Class ProdukteDataGrid
    Public Sub New()
```

```
    End Sub
End Class
```

Wir benötigen einen Verweis auf den Namespace **System.Collections.ObjectModel**:

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

In der Klasse definieren wir eine lokale Variable für die Kontextklasse sowie die Liste der Produkte:

```
Private dbContext As BestellverwaltungContext
Private m_Produkte As ObservableCollection(Of Produkt)
```

Die Liste der Produkte wollen wir über eine öffentliche Eigenschaft namens **Produkte** verfügbar machen:

```
Public Property Produkte() As ObservableCollection(Of Produkt)
    Get
        Return m_Produkte
    End Get
    Set(ByVal value As ObservableCollection(Of Produkt))
        m_Produkte = value
    End Set
End Property
```

Damit das DataGrid beim Öffnen direkt mit den Daten gefüllt wird, erweitern wir die Konstruktormethode mit diesen Anweisungen:

```
Public Sub New()
    InitializeComponent()
    dbContext = New BestellverwaltungContext
    Produkte = New ObservableCollection(Of Produkt)(dbContext.Produkte)
    DataContext = Me
End Sub
```

Darin füllen wir die Kontext-Variable und die **List**-Variable **Produkte** mit der Liste der Produkte. Schließlich stellen wir den **DataContext** auf die Code behind-Klasse ein. Wenn wir nun das Projekt starten und im Fenster **MainWindow** auf die Schaltfläche klicken, erscheint das DataGrid wie in Bild 1.

Kombinationsfelder anzeigen

Die Daten der Felder **KategorieID** und **MehrwertsteuersatzID** möchten wir gern aus den jeweils verknüpften Auflistungen **Kategorien** und **Mehrwertsteuersaetze** füllen. Außerdem wollen wir noch die Spaltenüberschriften anpassen. Das erledigen wir mit den folgenden Erweiterungen, wobei wir erst die Daten für die Kombinationsfelder in der Code behind-Klasse verfügbar machen. Dazu fügen wir zwei private Variablen für die Listen der Kategorien und der Mehrwertsteuersätze hinzu:

```
Private m_Kategorien As List(Of Kategorie)
Private m_Mehrwertsteuersaetze As List(Of Mehrwertsteuersatz)
```

Für beide richten wir entsprechende öffentliche Eigenschaften ein:

```
Public Property Kategorien As List(Of Kategorie)
    Get
        Return m_Kategorien
    End Get
    Set(ByVal value As List(Of Kategorie))
        m_Kategorien = value
    End Set
End Property

Public Property Mehrwertsteuersaetze As List(Of Mehrwertsteuersatz)
    Get
        Return m_Mehrwertsteuersaetze
    End Get
    Set(ByVal value As List(Of Mehrwertsteuersatz))
        m_Mehrwertsteuersaetze = value
    End Set
End Property
```

Schließlich füllen wir diese in der Konstruktor-Methode aus den entsprechenden DbSets:

```
Public Sub New()
    ...
    Kategorien = New List(Of Kategorie)(dbContext.Kategorien)
    Mehrwertsteuersaetze = New List(Of Mehrwertsteuersatz)(dbContext.Mehrwertsteuersaetze)
    DataContext = Me
End Sub
```

Damit stehen die Daten der Kategorien und der Mehrwertsteuersätze über die öffentlichen Variablen **Kategorien** und **Mehrwertsteuersaetze** für den Zugriff aus der XAML-Seite bereit. Diese passen wir nun wie folgt an. Zunächst stellen wir **AutoGenerateColumns** auf **False** ein, da wir die Spalten nun von Hand definieren:

```
<DataGrid x:Name="dgProdukte" ItemsSource="{Binding
Produkte}" AutoGenerateColumns="False">
```

ProdukteDataGrid

ID	Name	KategorieID	Einzelpreis	MehrwertsteuersatzID	Bestand	Mindestbestand
1	Artikel 1	1	414	1	696	180
2	Artikel 2	2	544	1	815	541
3	Artikel 3	3	510	1	229	620
4	Artikel 4	4	681	2	887	371
5	Artikel 5	5	294	2	151	530
6	Artikel 6	6	585	1	364	876
7	Artikel 7	1	191	2	684	748
8	Artikel 8	2	782	2	163	808
9	Artikel 9	3	957	1	67	62
10	Artikel 10	4	380	2	464	120
11	Artikel 11	5	175	1	49	715
12	Artikel 12	6	561	2	218	469
13	Artikel 13	1	753	1	400	903

Bild 1: DataGrid mit den Daten der Tabelle **Produkte**

Dann folgen die einzelnen Spaltendefinitionen im **DataGrid.Columns**-Element:

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

Das Kombinationsfeld realisieren wir als **DataGridComboBocColumn**-Element, für das wir den Header festlegen sowie das gebundene Feld der Datenquelle des DataGrids beziehungsweise des Fensters (**SelectedValueBinding**), den Namen des anzuzeigenden Feldes (**DisplayMemberPath**) und den Namen des gebundenen Feldes (**SelectedValuePath**):

```
    <DataGridComboBoxColumn Header="Kategorie" SelectedValueBinding="{Binding KategorieID}"
        DisplayMemberPath="Name" SelectedValuePath="ID">
```

Die Datenquelle für das Kombinationsfeld müssen wir über einen Umweg festlegen, indem wir die Eigenschaft **ItemsSource** manuell definieren und füllen. Die Daten beziehen wir dabei aus der entsprechenden Eigenschaft des übergeordneten **Window**-Elements. Dies legen wir sowohl für die Eigenschaft **ElementStyle** als auch für die Eigenschaft **EditingElementStyle** fest. Dabei liefert **ElementStyle** die Formatierung für den angezeigten Text des Kombinationsfeldes und **EditingElementStyle** die Formatierung für die Elemente in der Liste nach dem Aufklappen:

```
        <DataGridComboBoxColumn.ElementStyle>
            <Style TargetType="{x:Type ComboBox}">
            <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Kategorien,
                RelativeSource={RelativeSource AncestorType={x:Type Window}}}" />
        </Style>
        </DataGridComboBoxColumn.ElementStyle>
        <DataGridComboBoxColumn.EditingElementStyle>
            <Style TargetType="{x:Type ComboBox}">
                <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Kategorien,
                    RelativeSource={RelativeSource AncestorType={x:Type Window}}}" />
            </Style>
        </DataGridComboBoxColumn.EditingElementStyle>
    </DataGridComboBoxColumn>
```

Beim Kombinationsfeld für die Mehrwertsteuersätze ist noch eine weitere Information in Form eines zusätzlichen Setters erforderlich, und zwar die Angabe der Formatierung. Während die Mehrwertsteuersätze in Form der Werte **0,07** und **0,19** gespeichert sind, sollen diese als **7,00%** und **19,00%** im Kombinationsfeld erscheinen. Daher definieren wir hier zusätzlich den Setter für die Eigenschaft **ItemStringFormat**:

```
    <DataGridComboBoxColumn Header="MwSt.-Satz" SelectedValueBinding="{Binding MehrwertsteuersatzID}"
        DisplayMemberPath="Name" SelectedValuePath="ID">
        <DataGridComboBoxColumn.ElementStyle>
            <Style TargetType="{x:Type ComboBox}">
```

```
                    <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Mehrwertsteuersaetze,
                        RelativeSource={RelativeSource AncestorType={x:Type Window}}}" />
                    <Setter Property="ItemStringFormat" Value="P"></Setter>
                </Style>
            </DataGridComboBoxColumn.ElementStyle>
            <DataGridComboBoxColumn.EditingElementStyle>
                <Style TargetType="{x:Type ComboBox}">
                    <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Mehrwertsteuersaetze,
                        RelativeSource={RelativeSource AncestorType={x:Type Window}}}" />
                    <Setter Property="ItemStringFormat" Value="P"></Setter>
                </Style>
            </DataGridComboBoxColumn.EditingElementStyle>
        </DataGridComboBoxColumn>
```

Schließlich folgen noch die Spaltendefinitionen für die übrigen Felder:

```
        <DataGridTextColumn Binding="{Binding Path=Einzelpreis}" Header="Einzelpreis"></DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Path=Bestand}" Header="Bestand"></DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Path=Mindestbestand}" Header="Mindestbestand"></DataGridTextColumn>
        <DataGridCheckBoxColumn Binding="{Binding Path=Auslaufprodukt}" Header="Auslaufprodukt"></DataGridCheckBoxColumn>
    </DataGrid.Columns>
</DataGrid>
```

Nach einem Neustart sieht das **DataGrid**-Steuerelement nun wie in Bild 2 aus.

Bearbeiten vorhandener Datensätze

Nun schauen wir uns an, wie der Benutzer die vorhandenen Datensätze bearbeiten und die Änderungen an den bearbeiteten Datensätzen speichern kann. Dazu bedarf es keiner großartigen Änderungen am Design des **DataGrid**-Steuerelements. Die Änderungen sind jetzt schon aktiviert – Sie können Änderungen an den Feldern durchführen, die auch nach dem Verlassen der Zeile beibehalten werden. Allerdings werden diese natürlich nicht in der zugrunde liegenden Tabelle gespeichert.

Geänderten Datensatz speichern

Dazu müssen wir noch einen entsprechenden Mechanismus einrichten. Als Erstes benötigen wir ein Ereignis, dass beim Verlassen der aktuell markierten Zeile ausgelöst wird. Dieses finden wir in der Ereignisprozedur **RowEditEnding**, welches wir dem **DataGrid**-Element wie folgt hinzufügen:

```
<DataGrid x:Name="dgProdukte" ItemsSource="{Binding Produkte}" AutoGenerateColumns="False" RowEditEnding="dgProdukte_Ro-
wEditEnding">
```

Dieses Ereignis hinterlegen wir in der Code behind-Datei von **ProdukteDataGrid.xaml** wie folgt und füllen es mit einer einzigen Anweisung, nämlich der folgenden:

```
Private Sub dgProdukte_RowEditEnding(sender As Object, e As DataGridRowEditEndingEventArgs)
    dbContext.SaveChanges()
End Sub
```

Diese speichert nach dem Auslösen die für den Datenbankkontext durchgeführten Änderungen in die Datenbank. Wenn Sie nun direkt die Anwendung starten und die Speichern-Funktion testen wollen, werden Sie keinen Erfolg haben: Noch wird dieses Ereignis nämlich nicht ausgelöst. Dazu müssen Sie erst alle Bindungen der zu ändernden Steuerelemente mit ein paar zusätzlichen Attributen versehen. Für die Textfelder sieht das etwa wie folgt aus:

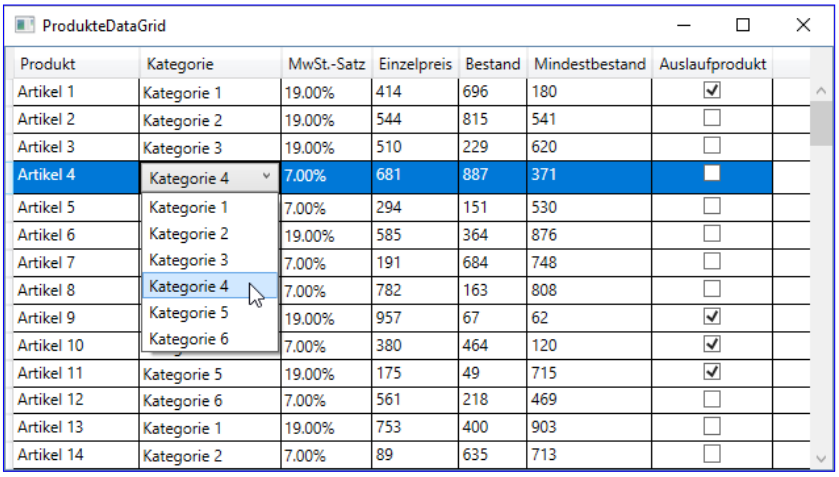

Bild 2: DataGrid mit Kombinationsfeldern

```
<DataGridTextColumn Binding="{Binding Path=Name, NotifyOnTargetUpdated=True, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" Header="Produkt"></DataGridTextColumn>
```

Und dies ist die Änderung, die Sie für ein Kombinationsfeld durchführen müssen:

```
<DataGridComboBoxColumn Header="Kategorie" SelectedValueBinding="{Binding KategorieID, NotifyOnTargetUpdated=True, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" DisplayMemberPath="Name" SelectedValuePath="ID">
```

Wir fügen also der **Binding**-Eigenschaft der Textspalten und der **SelectedValueBinding**-Eigenschaft der Kombinationsfeldspalte die drei Attribute **NotifyOnTargetUpdate**, **Mode** und **UpdateSourceTrigger** mit den angegebenen Werten hinzu. Anschließend können wir den Inhalt einer der Spalten für einen Datensatz ändern und die Änderungen werden auch in die Datenbank übertragen. Die **Binding**-Eigenschaften müssen Sie natürlich für alle Steuerelemente, die eine Spalte im **DataGrid**-Steuerelement repräsentieren, auf diese Weise anpassen.

Neuen Datensatz anlegen

Um einen neuen Datensatz anzulegen, brauchen Sie erstmal nichts zu tun – das gelingt genau wie das Ändern vorhandener Datensätze automatisch. Allerdings werden auch hier die Änderungen nicht ohne weiteres Zutun gespeichert. Also nutzen wir auch hier eine Ereigniseigenschaft des **DataGrid**-Elements, in diesem Fall **AddingNewItem**:

```
<DataGrid x:Name="dgProdukte" ItemsSource="{Binding Produkte}" AutoGenerateColumns="False" RowEditEnding="dgProdukte_RowEditEnding" AddingNewItem="dgProdukte_AddingNewItem">
```

In diesem Fall deklarieren wir zunächst eine neue **Boolean**-Variable in der Code behind-Klasse **ProdukteDataGrid.xaml.vb**:

```
Private bolEinfuegen As Boolean
```