



Objektrelationales Mapping
mit Microsofts Entity Framework 6.0

Klar abbilden

Holger Schwichtenberg

Nachdem Microsoft den Markt der objektrelationalen Mapper lange anderen überlassen hat, konnte sich das kürzlich in der Version 6.0 erschienene Entity Framework mittlerweile als führendes ORM-Werkzeug in der .NET-Welt etablieren. Tipps zum Einsatz des neuen Release.

In den letzten zwei Jahrzehnten hat Microsoft etliche Datenzugriffstechniken auf den Markt gebracht: RDO, DAO, ADO, ADO.NET, LINQ-to-SQL und Entity Framework. Lange Zeit arbeiteten die Redmonder auf der Ebene von Tabellen, Zeilen und Spalten, während die Konkurrenz, etwa Java, längst relationale Konzepte auf Objekte abbildete. Das Geschäft mit objektrelationalen Mappern (ORM) überließ man auch beim Erscheinen des .NET-Frameworks im Jahr 2001 zunächst Drittanbietern.

Ansätze wie WinFS und ObjectSpaces ließ Microsoft 2006 in der Alpha-Phase krepieren. Das 2007 erschienene LINQ-to-SQL lebte nur neun Monate und musste im August 2008 dem Entity Framework weichen. Die erste Version hieß noch ADO.NET Entity Framework und wurde im Rahmen des .NET Framework 3.5 Service Pack 1 ausgeliefert. Mittlerweile gibt es die Version 6.0 (wobei der Softwareriese die Versionen zwischen 1.0 und 4.0 aus politischen Gründen ausgelassen hat), die sich von dem Vorwort

„ADO.NET“ sowie aus dem .NET Framework im engeren Sinne verabschiedet hat. Heute ist das Entity Framework ein unabhängiges Open-Source-Projekt (alle relevanten Webinformationen zu diesem Text findet man unter „Alle Links“). Microsoft bleibt jedoch weiterhin treibende Kraft dahinter und bietet Support über seine kommerziellen und nichtkommerziellen Kanäle an.

Auf dem Komponentenportal nuget.org rangiert das Entity Framework unter den drei beliebtesten Softwareprodukten – weit vor dem älteren und immer noch mächtigeren NHibernate, einer Portierung aus der Java-Welt, die jedoch in letzter Zeit Unterstützer verliert. Microsoft empfiehlt seiner Entwicklergemeinschaft, neue Projekte mit Entity Framework zu starten, betont jedoch zugleich, dass im .NET Framework verankerte tabellenorientierte ADO.NET-Konzepte wie *DataReader* und *DataSet* erhalten bleiben zur Pflege alter Anwendungen sowie für einige Fälle, die ein ORM nicht abdecken kann.

LINQ: Datenbankneutral und hilfsbereit

Ein wesentlicher Erfolgsfaktor für das Entity Framework ist die Language Integrated Query (LINQ). Die mit .NET 3.5 eingeführte SQL-ähnliche Sprache kann auf beliebige Datenspeicher zugreifen und fügt sich in die Programmiersprachen C# und Visual Basic .NET ein. Der Editor schlägt bei der Eingabe Statements vor, und der Compiler moniert viele Fehler, die beim Einbetten von typischen SQL-Zeichenketten in den Programmcode oder der Auslagerung in externe Ressourcendateien in der Regel erst zur Laufzeit auffallen.

LINQ-to-Entities, wie der Einsatz von LINQ bei Entity Framework heißt, bietet eine datenbankneutrale Abfragesprache, die in vielen Fällen mit Prägnanz gegenüber dem datenbankspezifischen SQL hervorsteht. Im Untergrund von Entity Framework arbeiten Klassen aus ADO.NET: *DbConnection*, *DbCommand* und *DbDataReader*. Voraussetzung für das Nutzen des Frameworks ist neben dem passenden ADO.NET-Datenbanktreiber ein zusätzlicher Treiber für das jeweilige Datenbanksystem, der LINQ-to-Entities zur Laufzeit in datenbankspezifisches SQL übersetzt. Microsoft liefert entsprechende Treiber nur für den hauseigenen SQL Server (einschließlich des kostenfreien SQL Server Express) sowie die lokale Variante SQL Server Compact Edition. Treiber für andere Datenbanken wie Oracle, MySQL und Firebird gibt es entweder

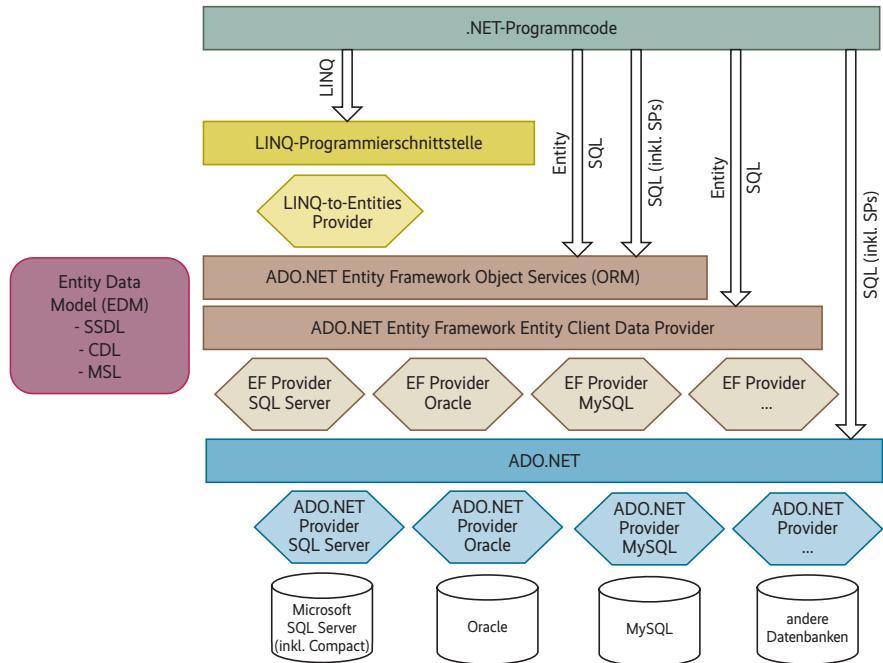
von den Herstellern oder von Drittanbietern wie Devart und DataDirect.

Wie Abbildung 1 zeigt, existieren neben LINQ-to-Entities weitere Abfragesprachen für Entity Framework: Entity SQL ist zwar ebenfalls datenbankneutral, hat jedoch eine nicht vom Editor beziehungsweise Compiler prüfbare Textform. Wer der Übersetzung von LINQ-to-Entities und Entity SQL in SQL nicht traut, kann mit SQL selbst oder dem Aufruf von Stored Procedures mit der Datenbank interagieren, verliert dann aber häufig die Unabhängigkeit der Zugriffsschicht vom Datenbankmanagementsystem. Der Standard zur Arbeit mit Entity Framework ist LINQ, das es mittlerweile auch für den Konkurrenten NHibernate gibt.

Das Entity Data Model (EDM) ist das Herzstück des Frameworks. EDM beschreibt das Datenbankschema, die Struktur der .NET-Klassen sowie das Mapping zwischen diesen beiden Ebenen. Entity Framework unterstützt drei Vorgehensmodelle für das OR-Mapping (Abbildung 2). Beim Reverse Engineering erzeugt der Entwickler ein EDM auf Basis einer bestehenden relationalen Datenbank. Komfortable Unterstützung bieten dabei ein Assistent sowie ein grafischer Designer (Abbildung 3), die Microsoft in der Entwicklungsumgebung Visual Studio ausliefert. Weitere Funktionen stellt der Entity Developer von Devart bereit.

Mehr Freiheit für den Entwickler

Aus dem EDM erstellt ein Codegenerator .NET-Klassen. Diesen Vorgang kann der Entwickler via Text Template Transformation Toolkit (T4) steuern. Es entstehen sogenannte Entitätsklassen, die Daten aus der Datenbank aufnehmen, sowie eine sogenannte Kontextklasse, die eine Programmierschnittstelle für das Laden und Verändern von Instanzen der Entitätsklassen anbietet. Während in der ersten Framework-Version die Vererbung der Entitäts-



Architektur des Entity Framework: Neben dem Standard LINQ-to-Entities existieren weitere Abfragesprachen (Abb. 1).

klassen von der Basisklasse *EntityObject* noch Pflicht war, ist der Entwickler inzwischen beim Aufbau der Klassen frei und kann sogenannte Plain Old CLR Objects (POCOs) schaffen, die nicht vom Framework abhängig sind. Zu den Kontextklassen existieren zwei Alternativen: *ObjectContext* und *DbContext*. Letztere offeriert als neuere Variante einige Zusatzfunktionen, agiert jedoch leider manchmal etwas langsamer, denn *DbContext* ist ein Wrapper um *ObjectContext*. Von der unglücklichen Namensgebung darf man sich nicht verwirren lassen: Beide Kontextklassen arbeiten auf der gleichen Ebene und erfüllen vergleichbare Funktionen.

Alternativ zum Reverse Engineering gibt es zwei Varianten des Forward Engineering, bei denen das Objektmodell im Mittelpunkt steht und ein Generator das Datenbankschema erzeugt. Bei Model First erstellt der Entwickler das Objektmodell im grafischen Designer, bei Code First (alias Code Only) tippt er den Quell-

code für das Objektmodell ein. Microsoft favorisiert mittlerweile klar das Code-First-Vorgehen und hat in den letzten Versionen für Model First die Funktionen leider reduziert statt ausgebaut.

Die Listings 1 bis 3 zeigen Klassen für den Code-First-Ansatz am Beispiel einer Fluggesellschaft. Der komplette Programmcode liegt auf dem FTP-Server der iX (siehe „Alle Links“). Listing 1 erzeugt eine POCO-Klasse *Flug*, Listing 2 entsprechende Klassen für *Person* und davon abgeleitet *Pilot* und *Passagier*. Die Klasse in Listing 3 erstellt den benötigten Entity-Framework-Kontext, der bei Code First immer von *DbContext* abgeleitet werden muss. Bei Code First gilt das Prinzip „Konvention vor Konfiguration“: Microsoft hat zahlreiche Konventionen definiert, die in den meisten Fällen zu ei-

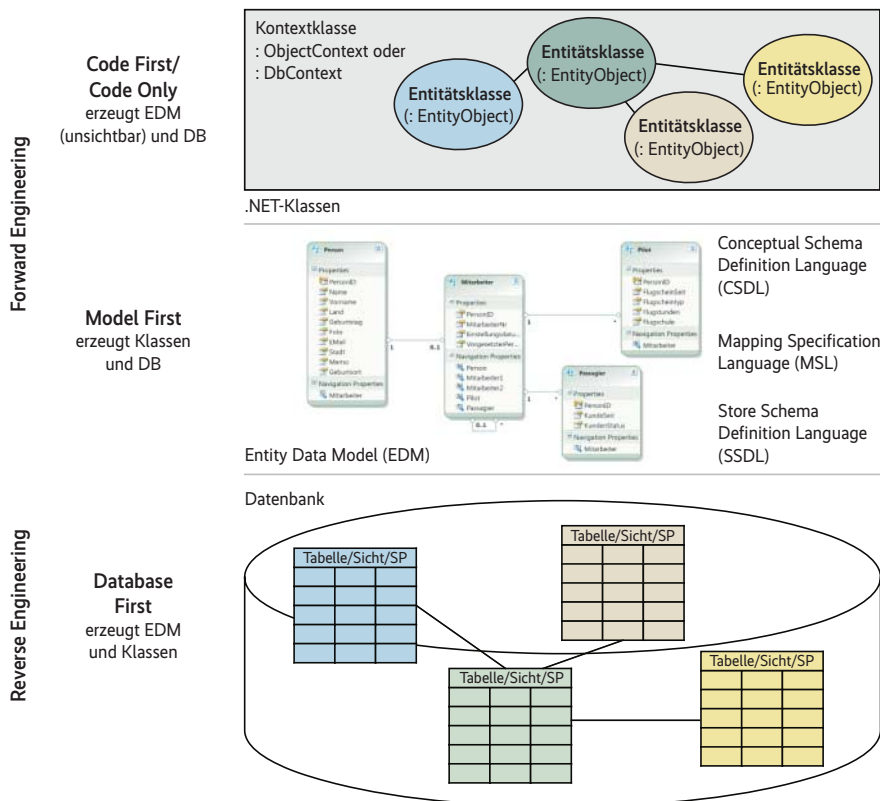
Listing 1: Entitätsklasse Flug

```
public partial class Flug
{
    // --- Primärschlüssel
    [Key]
    public int FlugNr { get; set; }
    // --- Weitere Eigenschaften
    [StringLength(50), MinLength(3)]
    public string Abflugort { get; set; }
    [StringLength(50), MinLength(3)]
    public string Zielort { get; set; }
    public System.DateTime Datum { get; set; }
    [Range(100, 250)]
    // --- Beziehungen zu anderen Entitäten
    public virtual ICollection<Passagier>
        Passagiere { get; set; }

    [ForeignKey("PilotId")]
    public virtual Pilot Pilot { get; set; }
    [ForeignKey("CopilotId")]
    public virtual Pilot Copilot { get; set; }
    public int PilotId { get; set; }
    public int CopilotId { get; set; }
}
```



- Microsofts Entity Framework bietet drei Vorgehensmodelle für objektrelationales Mapping. Das Werkzeug hat sich als führender OR-Mapper in der .NET-Welt festgesetzt.
- Beim Code-First-Ansatz beeinflusst der Entwickler durch Konventionen und Konfiguration die Datenbankschema-Generierung.
- LINQ-Befehle sind datenbankunabhängig, stabiler und in der Regel erheblich prägnanter als SQL.



Die drei Vorgehensmodelle beim Entity Framework: Microsoft favorisiert inzwischen das Code-First-Vorgehen (Abb. 2)

nem funktionierenden und akzeptablen Datenbankschema führen.

Eine Konvention besagt, dass automatisch alle Spalten, die *ID* oder *KlassennameID* heißen, zu Primärschlüsseln mit Autowerten umgewandelt werden. Die Tabellen nennen sich wie die Klassen, nur im Plural. Eine Konvention bildet Navigationsbeziehungen zwischen Klassen via Fremdschlüssel oder im *n:m*-Fall über Zwischentabellen ab. Die Fremdschlüsselnamen ergeben sich aus der Navigationsbeziehung mit angehängtem *ID*. Für alle Fremdschlüssel generiert Entity Framework einen Index. Jeder .NET-Datentyp besitzt einen korrespondierenden Standardtyp in der Datenbank. Für *string* beispielsweise wäre es im SQL Server *nvarchar(max)*. Die generische Klasse

Nullable<T> steuert, ob ein Datenbankfeld den Wert *null* annehmen darf. Vererbungsbeziehungen bildet Code First im Regelfall durch einen Table per Hierarchy-Strategie ab. Das heißt, für alle Klassen in einer Vererbungshierarchie gibt es nur eine Tabelle mit einer *Discriminator*-Spalte. Die Deklaration aller Navigationseigenschaften als *virtual* ist zwingend notwendig für das Funktionieren von automatischen Nachladefunktionen (Lazy Loading).

Das Standardverhalten lässt sich auf vier Arten ändern:

1. Konventionen lassen sich abschalten. Dies sieht man in Listing 3 in der Methode *OnModelCreating()*, die die Konventionen zur Pluralbildung der Tabellennamen ausschaltet. Microsoft hat nur an den Heimat-

markt gedacht, und so funktioniert die automatische Pluralbildung nur für Englisch. Ob das Ganze überhaupt sinnvoll ist, sei mal dahingestellt. So entstehen beim Standardverhalten die Tabellen *Flugs* und *People* (englischer Plural von *Person*). Bei abgeschalteter Konvention heißen die Tabellen wie die Klassen.

2. Seit Entity Framework Version 6.0 kann jeder Entwickler eigene Konventionen schreiben. Eine einfache Konvention sorgt dafür, dass statt *Table* per Hierarchy-Strategie *Table* per Type-Strategie zum Einsatz kommt, indem sie durchsetzt, dass alle Entitätstypen auf eine gleichnamige Tabelle abzubilden sind (Listing 3). Solche Konventionen kann man auch in wiederverwendbare eigene Klassen auslagern.

3. Die dritte Alternative sind Datenannotations auf Klassen und Klasseigenschaften. Listing 1 und 2 zeigen einige Beispiele: *[Key]* macht die nicht der Konvention entsprechende Eigenschaft *FlugNr* zum Primärschlüssel, *[StringLength]* fügt den Zeichenspalten in der Datenbank eine Längenbegrenzung hinzu, und *[InverseProperty]* legt fest, wie die doppelte Navigationsbeziehung zwischen *Flug* und *Pilot* modelliert werden soll.

4. Die letzte Variante ist das Konfigurieren über das Fluent API. In *OnModelCreating()* (Listing 3) führt der Aufruf von *MapToStoredProcedures()* dazu, dass für alle *INSERT*-, *UPDATE*- und *DELETE*-Aktionen Stored Procedures entstehen, die Entity Framework statt direkter SQL-Anweisungen verwendet.

Ein Datenbankschema entsteht aus den erfassten Klassen mithilfe der Schema-Migrationswerkzeuge von Entity Framework. Diese hilfreichen Tools stehen als PowerShell-Commandlets in der Package-Manager-Konsole von Visual Studio zur Verfügung. Sie können alternativ auch beim Start der Anwendung zum Einsatz kommen. Abbildung 4 zeigt die Befehlskette für das Erzeugen des Datenbankschemas, das entweder die erzeugten Befehle wahlweise direkt zur Datenbank

Listing 2: Entitätsklassen Person, Pilot und Passagier

```
public partial class Person
{
    // --- Primärschlüssel
    public int PersonID { get; set; }
    // --- Weitere Eigenschaften
    [StringLength(50)]
    public string Name { get; set; }
    [StringLength(25)]
    public string Vorname { get; set; }
    public Nullable<System.DateTime> Geburtstag { get; set; }
}

public partial class Passagier : Person
{
    // Primärschlüssel wird geerbt!
    [StringLength(1), MinLength(1), RegularExpression("[ABC]")]

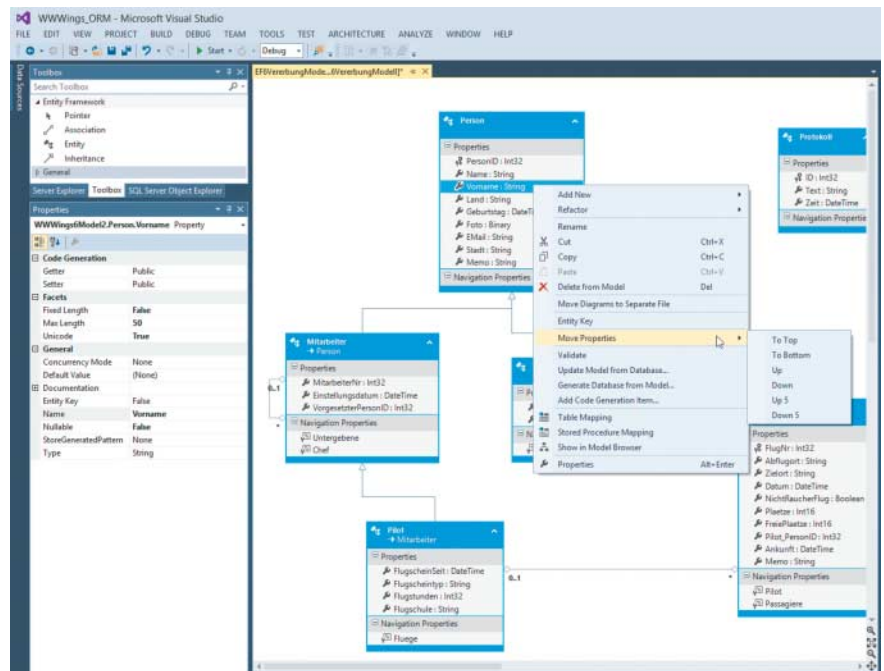
    public virtual string KundenStatus { get; set; }
    public virtual ICollection<Flug> Flug { get; set; }
}

public partial class Pilot : Person
{
    // Primärschlüssel wird geerbt!
    // --- Weitere Eigenschaften
    public virtual System.DateTime FlugscheinSeit { get; set; }
    public virtual Nullable<int> Flugstunden { get; set; }
    // --- Beziehungen zu anderen Entitäten
    [InverseProperty("Pilot")]
    public virtual ICollection<Flug> FluegeAlsPilot { get; set; }
    [InverseProperty("Copilot")]
    public virtual ICollection<Flug> FluegeAlsCopilot { get; set; }
}
```


sendet oder als getrennt startbares SQL-Skript auswirft. Abbildung 5 gibt Einblicke in das Datenbankschema, das via Listing 1 bis 3 entsteht.

Interessant ist hier die Tabelle *MigrationHistory*. Gemeinsam mit der durch das Add-Migration-Commandlet generierten Migrationsklasse sorgt sie dafür, dass der Entwickler beim Programmieren der Entitätsklassen Änderungsskripte erhält, die in vielen Fällen den aktuellen Datenbestand bewahren. Falls die Änderungsaktion zu komplex erscheint für den Automatismus (etwa beim Aufteilen von Entitäten), kann er die Migrationsklasse oder das SQL-Skript manuell anpassen. Die *MigrationHistory* versetzt ihn in die Lage, später zu einem früheren Stand des Datenbankschemas zurückzukehren.

Mithilfe der ebenfalls kostenfreien Entity Framework Power Tools für Visual Studio 2010, 2012 und 2013 lässt sich mit dem Code-First-Ansatz auch dann arbeiten, wenn schon eine Datenbank existiert. Mit diesen Werkzeugen erzeugt der Entwickler die Entitätsklassen sowie eine Kontextklasse für die Datenbank. Die Entitätsklassen lassen sich später ändern und als Schema-Migration in die Datenbank übernehmen. Ebenfalls ist es möglich, ein Diagramm für die Entitätsklassen bei der



Der Designer für das Reverse Engineering kommt mit Visual Studio (Abb. 3).

Code-First-Vorgehensweise zu erstellen. In diesem Fall dient das Diagramm allerdings nur Ansichtszwecken, grafisch modellieren geht nicht.

Listing 4 zeigt das Befüllen der erzeugten Datenbank mit exemplarischen Daten. Die Instanziierung des Framework-Kontexts ist Voraussetzung für die Interaktion mit der Datenbank. Entitäts-

objekte können jedoch zunächst ohne Beteiligung des Kontexts im RAM angelegt und verbunden werden. Es reicht, ein Objekt aus einem Objektraum dem Kontext über die *Add()*-Methode zu melden. Der Aufruf von *SaveChanges()* speichert alle neuen Objekte in einer Transaktion. Die vorherige Umlenkung der Log-Eigenschaft des Kontexts auf die Konsole sorgt

Listing 3: Kontextklasse

```
public class FluggesellschaftContext : DbContext
{
    // Konstruktoren mit Angabe des Connection String
    public FluggesellschaftContext() : base("DB_Fluggesellschaft") { }
    public FluggesellschaftContext(string Connstring) : base(Connstring)
    { }

    // Einsprungpunkte für Zugriff auf Entitätsklassen
    public DbSet<Flug> Fluege { get; set; }
    public DbSet<Pilot> Piloten { get; set; }
    public DbSet<Person> Personen { get; set; }
    public DbSet<Passagier> Passagiere { get; set; }

    /// <summary>
    /// Methode zur Aktivierung/Deaktivierung von Konventionen
    /// sowie zur manuellen Konfiguration per Fluent-API
    /// </summary>
}
```

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    // Konventionen ausschalten: Keine Pluralisierung
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

    modelBuilder.Conventions.Remove<PluralizingEntitySetNameConvention>();

    // Eigene Konventionen: Table per Type statt Table per Hierarchie
    modelBuilder.Types().Configure(c => c.ToTable(c.ClrType.Name));

    // Konfiguration durch Fluent API: INSERT, UPDATE, DELETE über Stored Procedures
    modelBuilder.Entity<Flug>().MapToStoredProcedures();
    modelBuilder.Entity<Passagier>().MapToStoredProcedures();
    modelBuilder.Entity<Person>().MapToStoredProcedures();
    modelBuilder.Entity<Pilot>().MapToStoredProcedures();
}
```

Listing 4: Anlegen und Speichern von Objekten

```
// Pilot anlegen
Pilot pl = new Pilot();
pl.Name = "Mustermann";
pl.Vorname = "Max";
pl.FlugscheinSeit = new DateTime(1970, 1, 1);

// Kopilot anlegen
Pilot cpl = new Pilot();
cpl.Name = "Musterfrau";
cpl.Vorname = "Martina";
cpl.FlugscheinSeit = new DateTime(1968, 1, 1);

// Flug anlegen
Flug f = new Flug();
f.Abflugort = "Essen/Mülheim";
f.Zielort = "Hannover";
f.Plaetze = 150;
f.Datum = new DateTime(2014, 8, 1);
f.Pilot = cpl;
f.Copilot = cpl;
f.Passagiere = new List<Passagier>();

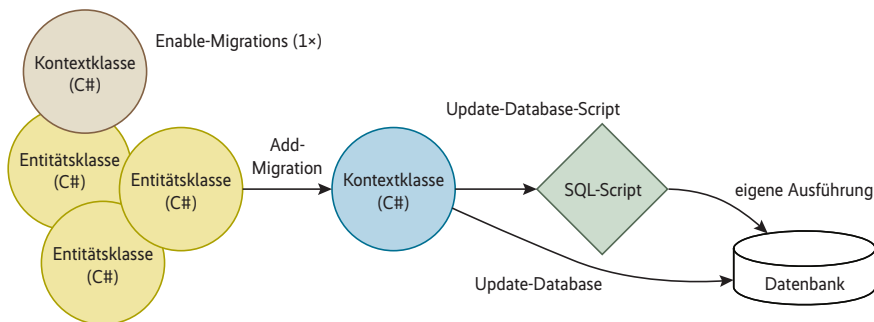
// Zehn Passagiere anlegen
```

```
for (int j = 1; j <= 10; j++)
{
    Passagier p = new Passagier();
    p.Name = "Passagier #" + j;
    p.Vorname = "";
    p.KundenStatus = "C";
    f.Passagiere.Add(p);
    Console.WriteLine(p.Name);
}

// Kontext erzeugen und mit Datenbank interagieren
using (FluggesellschaftContext ctx = new FluggesellschaftContext())
{
    // Protokollierung auf die Konsole
    ctx.Database.Log = Console.WriteLine;

    // Das Hauptobjekt muss mit dem Kontext verbunden werden
    ctx.Fluege.Add(f);

    // Alle Objekte in einer Transaktion speichern
    var anz = ctx.SaveChanges();
    Console.WriteLine("Anzahl gespeicherter Datensätze: " + anz);
}
```



Aus den Entitätsklassen entsteht über den Zwischenschritt einer Migrationsklasse das Datenbankschema (Abb. 4).

dafür, dass man die zur Datenbank gesendeten Befehle sieht. Das Protokoll kann der Entwickler an jede Methode senden, die eine Zeichenkette entgegennimmt. Detaillierte Steuerungsoptionen bietet die *LogFormatter*-Klasse. Noch genauere Informationen liefert der Entity Framework Profiler von Hibernating Rhinos, der allerdings 389 Dollar kostet.

Ein Beispiel für das Nutzen von Daten aus dieser Datenbank zeigt Listing 5. Zunächst definiert der Programmcode eine komplexe LINQ-to-Entities-Abfrage: Sie selektiert alle Flüge vom Abflugort Essen/Mülheim, denen bereits Pilot und Copilot zugewiesen wurde, die noch mindestens einen nicht belegten Platz besitzen und die mindestens einen Passagier mit Kundenstatus „C“ befördern. Die Abfrage sortiert die Ergebnismenge absteigend nach Flugdatum. Über Paging mit *Skip(10).Take(5)* gibt die Datenbank nur den elften bis fünften Datensatz aus. Und die drei *Include()*-Anweisungen erweitern die Ergebnismenge im Sinne eines Eager Loading (direktes Mitladen abhängiger Datensätze) um die Details zu Pilot, Copilot und allen Passagieren.

Anders als bei tabellenorientierten Datenzugriffsschnittstellen erhält der Entwickler die Objekte nicht „flachgeklopft“, sondern als Objektbaum. Ohne die *Include()*-Anweisungen würden zwar auch alle Datensätze in der späteren Schleife ausgegeben, jeder Zugriff auf ein verbundenes Objekt hätte aber ein explizites Nachladen (Lazy Loading) jedes einzelnen Detaildatensatzes zur Folge. Das ist meist deutlich langsamer in Szenarien, die viele Detaildatensätze benötigen. Die überschaubare LINQ-to-Entities-Abfrage erzeugt einen SQL-Befehl, der aus 16 312 Zeichen besteht. Ursache dafür ist neben dem Eager Loading, das mehrere Joins erstellt, vor allem das Paging, das verschachtelte Select-Anweisungen erfordert.

Der Entwickler hat die Wahl zwischen Lazy Loading und Eager Loading sowie eine Reihe weiterer Einflussmöglichkeiten auf die Performance. So kann er sogenannte No-Tracking-Abfragen absetzen, deren materialisierte Objekte der Entity-Framework-Kontext nicht auf Änderungen überwacht. Dies beschleunigt das Materialisieren der Objekte

deutlich, erlaubt es aber dennoch, später Objekte für eine Änderungsverfolgung beim Framework-Kontext mit der Methode *Attach()* anzumelden. Dieses Verfahren bietet sich an, wenn viele Datensätze zu laden sind, von denen der Benutzer typischerweise nur wenige modifiziert. Seit Entity Framework 6.0 sind asynchrone, das heißt nicht-blockierende Datenbankoperationen möglich unter Verwendung der in .NET 4.5 eingeführten Schlüsselwörter *async* und *await*.

Viele Optionen für mehr Performance

Grundsätzlich erzielt man durch Entity Framework eine mit dem *DataSet* aus ADO.NET vergleichbare Geschwindigkeit beim Datenzugriff. Wer extreme Anforderungen an die Reaktionszeiten hat, sollte weiterhin die *DataReader*-Klasse benutzen, die Entity Framework niemals überbieten kann, weil es intern selbst den *DataReader* verwendet, darauf aufbauend aber wesentlich mehr leistet.

Nicht alle Annotationen in Listing 1 und 2 schlagen auf die Datenbank durch. So werden *[MinLength]* und *[Range]* genauso wie *[RegularExpression]* nur zur Prüfung innerhalb des Framework-Kontexts verwendet, bevor *SaveChanges()* versucht, die Objekte zu speichern. Alle Regelverletzungen quittiert *SaveChanges()* mit einer *ValidationException*. Alternativ kann der Entwickler vor dem Speicherversuch die Methode *GetValidationErrors()* aufrufen, die ihm eine detaillierte Liste aller fehlerhaften Werte ausgibt. Diese kann er elegant mit den Steuerelementen der Bedienoberfläche verbinden und so dem Benutzer Eingabefehler anzeigen.

Einige Steuerelemente in .NET reagieren automatisch auf solche Annotationen. *GetValidationErrors()* findet jedoch keine Fehler, die sich erst aus dem Datenbankzustand ergeben. Doppelte Datensätze oder fehlende Masterdatensätze beispielsweise fallen erst beim tatsächlichen Speicherversuch auf. Genau wie die Basiskomponente ADO.NET sperrt Entity Framework standardmäßig keine Datensätze. Konkurrierende Änderungen an Daten fallen im Sinne eines „optimistischen Sperrens“ erst auf, wenn ein bereits geänderter Datensatz nochmals modifiziert werden soll.

Für das Feststellen der Änderungskonflikte gibt es zwei Verfahren: Zum einen kann der Entwickler eine als Byte-Array zu deklarierende und mit *[TimeStamp]* zu annotierende Zeitstempel-Ei-

Listing 5: LINQ-Abfrage definieren und ausführen

```
// Kontextinstanz erzeugen
using (FluggesellschaftContext ctx = new FluggesellschaftContext())
{
    // Abfrage definieren
    var flugAbfrage = (from f in ctx.Fluege.Include(f => f.Passagiere).Include(f => f.Pilot).Include(f => f.Copilot)
        where f.Abflugort == "Essen/Mülheim" // &&
            f.Plaetze > f.Passagiere.Count &&
            f.Passagiere.Any(p=>p.KundenStatus == "A") &&
            f.Pilot != null && f.Copilot != null
        orderby f.Datum descending
        select f).Skip(10).Take(5);

    // Abfrage ausführen
    var flugListe = flugAbfrage.ToList();

    // Schleife über alle Flüge
    foreach (var f in flugListe)
    {
        // Flug mit Pilot und Copilot ausgeben
        Console.WriteLine("Flug #" + f.FlugNr + " wird geflogen von " + f.Pilot.Name + " und " +
            f.Copilot.Name);

        // Alle Passagiere ausgeben
        foreach (var p in f.Passagiere)
        {
            Console.WriteLine("- " + p.Name);
        }
    }
}
```

genschaft schaffen. Zum anderen ist Entity Framework in der Lage, die aktuellen Spaltenwerte mit den Ursprungswerten zu vergleichen. Dazu sind die entsprechenden Eigenschaften mit *[ConcurrencyCheck]* zu annotieren. Alternativ lassen sich explizite Transaktionen um Framework-Aktionen legen, die abhängig vom gewählten Isolations-Level Datensätze sperren.

Zum Feststellen von Modifikationen an Objekten besitzt das Framework drei Mechanismen (Basisklasse *EntityObject*, *Runtime Proxy* und *Snapshot Tracking*), die je nach Szenario und Änderungsmenge Vor- und Nachteile bieten. Änderungen überwacht der Framework-Kontext aber nur im eigenen Prozess. Für Objekte, die den Prozess verlassen (zum Beispiel bei der Beförderung per Webservice), müssen entweder sogenannte Self-Tracking-Entities (die sich selbst merken, was an ihnen geändert wurde) oder ergänzende Bibliotheken wie WCF Data Services [1] zum Einsatz kommen.

Noch bleiben viele Wünsche offen

Obwohl Microsoft seinem Entity Framework seit Version 4.0 zahlreiche neue Funktionen spendiert hat, bleiben weiterhin viele Wünsche offen. So fehlt ein Second Level Cache, der Objekte über mehrere Kontextinstanzen hinweg speichert. Derzeit sind die geladenen Objekte an genau eine Kontextinstanz gebunden. Und es ist aus vielerlei Gründen nicht praktikabel (etwa weil der Kontext nicht Multi-Threading-fähig ist), nur mit einer Kontextinstanz in einer Anwendung zu arbeiten. Entity Framework unterstützt zwar seit Version 5.0 auch Geo-Datentypen, XML-Spalten und beliebige selbst definierte Spaltentypen (User Defined Types), die sich jedoch nicht auf Objekte abbilden lassen. Stored Procedures zum Datenlesen und Tabellenwertfunktionen sind bisher nur beim Reverse Engineering verwendbar. Beim Letzteren kann man Vererbungsbeziehungen im Objektmodell nur unter eng gesteckten Voraussetzungen realisieren; NHibernate und andere ORM-Produkte sind da flexibler.

Entity Framework sendet mehrere Änderungen bisher einzeln statt in einem Batch zur Datenbank. Auch Massenoperationen, für die das objektweise Verarbeiten keine Option darstellt, kann Microsofts ORM nicht durchführen. Bei großen Datenbankschemata mit mehreren

Hundert Tabellen entstehen Leistungsprobleme sowohl im Designer als auch bei der Erstinstanziierung des Framework-Kontextes, während der eigentliche Mapping-Code generiert wird.

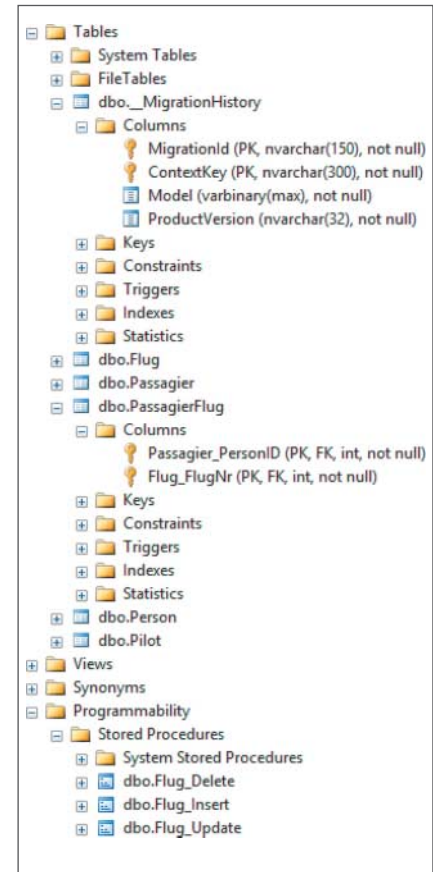
Bisher gibt es keine Möglichkeit, im Objektmodell Unique Constraints oder beliebige Indizes für die Datenbank festzulegen. Zudem geht das Framework immer davon aus, dass der Entwickler alle Spalten einer Tabelle zugleich laden möchte. Fallweises Laden beliebiger Spaltenkombinationen und das Nachladen weiterer Spalten funktioniert nur äußerst umständlich über eine Erweiterung. Zudem lässt sich eine Typkonvertierung beim Mapping nicht durchführen, um zum Beispiel ein „Ja“ oder „Nein“ in einem Objekt auf eine Bit-Spalte in der Tabelle abzubilden. Immerhin ermöglicht Microsoft den Nutzern, Verbesserungsvorschläge einzubringen und darüber abzustimmen. Zudem legt das Entwicklungsteam Pläne für kommende Versionen früh und detailliert offen.

Neben der Hoffnung, dass Microsoft diese Lücken bald schließt, haben Nutzer die Gelegenheit, sich selbst zu betätigen, denn die Open-Source-Lizenz gestattet das Weiterentwickeln des Entity Framework. In Version 6.0 hat der Softwarekonzern das Produkt zudem stärker als bisher modularisiert, sodass man Teile einfacher austauschen kann, ohne den Quellcode des Kernsystems zu verändern. Beispielsweise ließe sich die bisher nur englischsprachige Pluralisierung von Namen durch eine eigene Implementierung ersetzen.

Fazit

Nach einigen Anlaufschwierigkeiten hat Microsoft sein Entity Framework als objektrelationalen Mapper für .NET etabliert. Die Prägnanz, die Eingabeunterstützung sowie die Compiler-Prüfung für LINQ-to-Entities-Abfragen erhöhen den Komfort für Softwareentwickler in der Version 6.0 enorm. Anwendungen, die LINQ-to-Entities verwenden, sind per se robuster als SQL-Zeichenketten.

Da die erzeugten SQL-Befehle nicht immer die optimale Lösung darstellen, lässt sich LINQ in solchen Situationen umgehen. Es hat sich in vielen Projekten als praktikabel erwiesen, Entity Framework zur Standard-Datenzugriffstechnik zu erklären, denn es deckt erfahrungsgemäß 80 bis 90 Prozent der Fälle zufriedenstellend ab. Im Rahmen eines späteren Refactoring muss der Entwickler nur an den Stellen, wo die Leistung nicht



Das generierte Datenbankschema enthält auch eine Klasse `__MigrationHistory` mit allen Zwischenzuständen, die das Datenbankschema hatte (Abb. 5).

ausreicht, die Datenzugriffe optimieren oder das Framework zurückbauen und durch klassische Zugriffstechniken ersetzen. Framework und tabellenbasierte ADO.NET-Zugriffstechniken können in einer Anwendung koexistieren.

Der Code-First-Ansatz besitzt Charme und viele Anpassungsmöglichkeiten, ist jedoch ungeeignet für Entwickler, die das Datenbankschema nicht selbst verändern dürfen. Aber auch für das Reverse Engineering bestehender Datenbanken bietet Microsoft brauchbare Werkzeuge. (jd)

Dr. Holger Schwichtenberg

leitet das Expertennetzwerk www.IT-Visions.de, das Beratung, Schulungen und Softwareentwicklung im .NET-Umfeld anbietet. Er hält Vorträge auf Fachkonferenzen und ist Autor zahlreicher Fachbücher.

Literatur

- [1] Holger Schwichtenberg; Webdienste; Datenpumpe; Datenbasierte Webservices mit dem Open Data Protocol; iX 10/2012, S. 90

