

## Kapitel 12

# Objektrelationales Mapping (ORM) mit dem ADO.NET Entity Framework 4.0

### In diesem Kapitel:

Einführung	596
Objektrelationale Abbildung: Grundlagen und Anforderungen	597
Beispiel für dieses Kapitel	606
Grundbegriffe und Grundkonzepte des ADO.NET Entity Framework	607
Architekturmodelle	612
Erstellen eines EDM-Modells	614
Daten abfragen	628
Navigation und Ladestrategien	641
Objektzuweisungen	646
Änderungsverfolgung und Persistierung	650
Steuerung der Codegenerierung durch austauschbare T4-Vorlagen	658
Persistence Ignorance mit Plain Old CLR Objects (POCO)	664
Änderungsverfolgung in verteilten Systemen (Self-Tracking Entities)	670
Unterstützung für Stored Procedures	681
Forward Engineering (Model-First/Domain First)	688
Anpassungen und Erweiterbarkeit	699
Direktes Programmieren mit Entity SQL (eSQL)	702
Leistungsüberlegungen	709
Weitere Funktionen	715
Verbliebene Schwächen	715
Vergleich mit anderen ORM-Werkzeugen für .NET	716
Fazit	718

# Einführung

Während in der Java-Welt das Objektrelationale Mapping (ORM) schon sehr lange zu den etablierten Techniken gehört, hat Microsoft diesen Trend lange verschlafen bzw. es nicht vermocht, ein geeignetes Produkt zur Marktreife zu führen. Bis zum .NET Framework 3.5 gab es daher kein marktreifes ORM-Werkzeug von Microsoft. Das klassische ADO.NET (Command, DataReader, DataSet) beschränkt sich auf den direkten, tabellenorientierten Datenzugriff und die Abbildung zwischen XML-Dokumenten und dem relationalen Modell. Mehr als 20 Werkzeuge aus dem kommerziellen und nicht-kommerziellen Umfeld teilten sich daher bisher den Markt der ORM-Werkzeuge für .NET. Bekannte Drittanbieterprodukte sind NHibernate, Telerik Open Access (früher: Vanatec Open Access), .NET Data Objects (NDO) und Genome, wobei die letzten drei im deutschsprachigen Raum deswegen besonders bekannt sind, weil die Anbieter ihren Sitz in Deutschland bzw. Österreich haben.

Einzig Microsoft hat der Drang zur Veröffentlichung eines ORM-Werkzeugs lange nicht voll erfasst. Die Geschichte in Kurzfassung:

- Schon im Jahr 2003 (kurz nach Erscheinen von .NET 1.1) gab es eine Alpha-Version eines OR-Mappers unter dem Codenamen *Object Spaces*
- Diese war auch in .NET 2.0 Beta 1 enthalten, hat es aber leider nicht zur Marktreife geschafft
- Ein Grund dafür war, dass eine andere Abteilung bei Microsoft parallel an einem anderen ähnlichen Werkzeug entwickelt hat: In dem SQL-Server-basierten Dateisystem *WinFS*, das einst mit Windows Vista erscheinen sollte, aber dann beerdigt wurde, gab es auch Objektrelationale Mapping-Funktionen. Zwei ähnliche Produkte machten damals für die Redmonder Chefetage keinen Sinn.

Umso kurioser ist es nun, dass es seit dem Jahr 2008 dann noch wieder zwei OR-Mapper von Microsoft für .NET gibt: LINQ to SQL und die ADO.NET Entity Framework Object Services. LINQ to SQL ist im November 2007 als fester Bestandteil des .NET Framework 3.5 erschienen. Das ADO.NET Entity Framework (EF) und die zugehörigen Object Services lieferte Microsoft erst mit .NET 3.5 Service Pack 1 aus.

Inzwischen ist diese verwirrende Situation wieder etwas bereinigt:

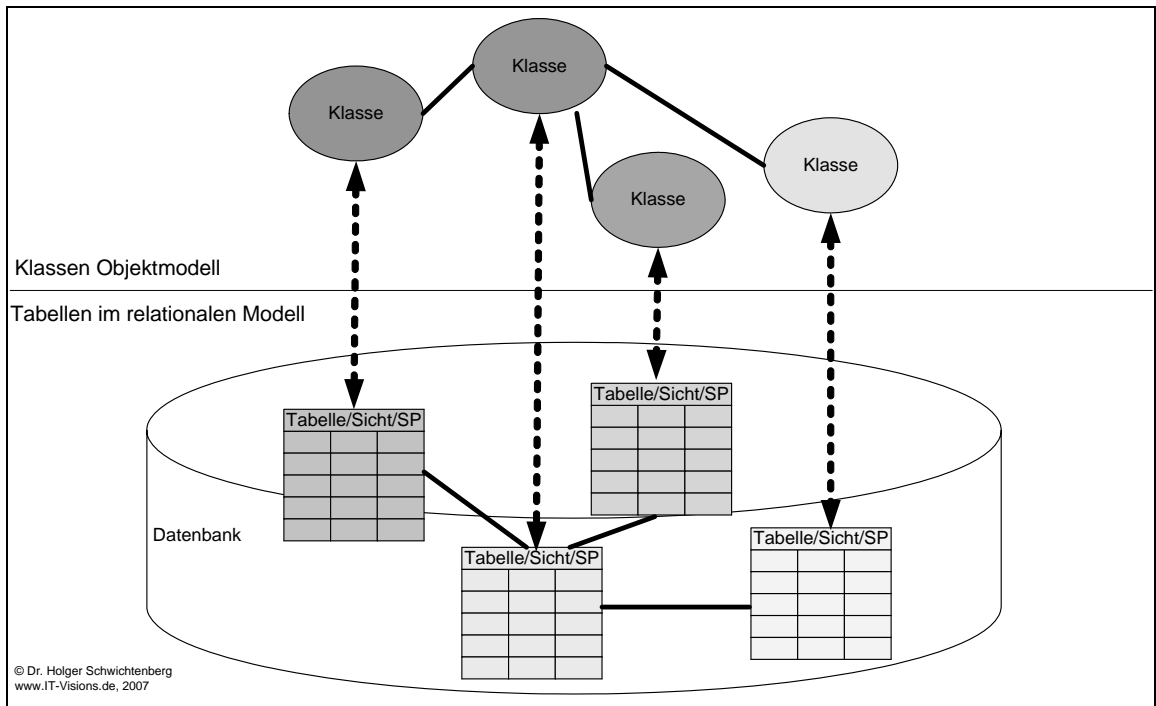
- Microsoft hat angekündigt, dass man auf das ADO.NET Entity Framework (EF) setzen wird.
- LINQ to SQL wird vorerst nicht aus .NET gelöscht, aber nicht mehr weiterentwickelt.<sup>1</sup> Dementsprechend ist das Entwicklungsteam von LINQ to SQL aufgelöst
- In .NET 4.0 gibt es zahlreiche neue Funktionen für das ADO.NET Entity Framework, darunter auch solche, in denen LINQ to SQL in .NET 3.5 SP1 noch besser war
- Für LINQ to SQL gibt es in .NET 4.0 einige kleinere Fehlerbehebungen, aber keine neuen Funktionen.

---

<sup>1</sup> Es gibt noch keine konkreten Pläne, nur Indizien, dass LINQ to SQL weniger lang in .NET existieren wird als ADO.NET Entity Framework.

# Objektrelationale Abbildung: Grundlagen und Anforderungen

In seinen Schulungen und Beratungsgesprächen trifft der Autor dieses Buchs immer wieder auf .NET-Entwickler, die mit dem Begriff *Objektrelationale Abbildung* und der zugehörigen Abkürzung *ORM* noch gar nichts anfangen können. Für diese Zielgruppe sei hier zunächst ORM definiert: ORM ist die Abbildung von Geschäftsobjekten im Hauptspeicher auf Datensätze in relationalen Datenbanktabellen zum Zwecke der Objektpersistenz (siehe Abbildung 12.1). Neben der Darstellung der Herausforderungen des ORM ist es Ziel dieses Abschnitts, Anforderungen und Bewertungskriterien für ORM-Werkzeuge festzulegen.



**Abbildung 12.1** ORM ist die Abbildung zwischen Objekten und Datensätzen bzw. Klassen und Tabellen

## Impedance Mismatch

Kern des Objektorientierten Programmierens (OOP) ist die Arbeit mit Objekten als Instanzen von Klassen im Hauptspeicher. Die meisten Anwendungen haben dabei auch die Anforderung, in Objekten gespeicherte Daten dauerhaft zu speichern, insbesondere in Datenbanken. Grundsätzlich existieren Objektorientierte Datenbanken (OODB), die direkt in der Lage sind, Objekte zu speichern. Aber Objektorientierte Datenbanken haben bisher nur eine sehr geringe Verbreitung. Der vorherrschende Typus von Datenbanken sind relationale Datenbanken, die jedoch Datenstrukturen anders abbilden als Objektmodelle.

Zwei besonders hervorstechende Unterschiede zwischen Objektmodell und Relationenmodell sind N:M-Beziehungen und Vererbung. Während man in einem Objektmodell eine N:M-Beziehung zwischen Objekten durch eine wechselseitige Objektmenge abbilden kann, benötigt man in der relationalen Datenbank eine Zwischentabelle. Vererbung kennen relationale Datenbanken gar nicht. Hier gibt es verschiedene Möglichkeiten der Nachbildung, doch dazu später mehr. Die unterschiedliche Art der Datenspeicherung zwischen Objektmodell und relationalem Modell wird in der Fachwelt als *Impedance Mismatch* oder *Semantic Gap* bezeichnet.

Wenn ein .NET-Entwickler via ADO.NET mit einem Datareader oder DataSet Daten aus einer Datenbank einliest, dann betreibt er noch kein ORM. Datareader und DataSet sind zwar .NET-Objekte, aber diese verwalten nur Tabellenstrukturen. Datareader und DataSet sind aus der Sicht eines Objektmodells untypisierte, unspezifische Container. Erst wenn ein Entwickler spezifische Klassen für die in den Tabellen gespeicherten Strukturen definiert und die Inhalte aus DataSet oder Datareader in diese spezifischen Datenstrukturen umkopiert, betreibt er ORM.

Dies ist allein schon für den Lesezugriff (gerade bei sehr breiten Tabellen) eine sehr aufwändige, mühselige und eintönige Programmierarbeit. Will man dann auch noch Änderungen in den Objekten wieder speichern, wird die Arbeit allerdings zur intellektuellen Herausforderung, denn man muss erkennen können, welche Objekte verändert wurden, da man sonst ständig alle Daten wieder speichert, was in Mehrbenutzerumgebungen ein Unding ist.

Viele .NET-Entwickler haben sich in den letzten Jahren daran gesetzt, diese Arbeit zu vereinfachen mit Hilfsbibliotheken und Werkzeugen. Dies war die Geburtsstunde vieler ORM-Werkzeuge, die in der Entwicklerumgangssprache in der Regel einfach als *OR-Mapper* bezeichnet werden. Dabei scheint es so, dass in dem geflügelten Wort, dass ein Mann in seinem Leben einen Baum gepflanzt, ein Kind gezeugt und ein Haus gebaut haben sollte, viele .NET-Entwickler einen der drei Punkte gegen »einen OR-Mapper geschrieben« austauschen wollten. Anders ist die Vielfalt der ähnlichen Lösungen kaum erklärbar. Neben den öffentlich bekannten ORM-Werkzeugen für .NET findet man in den Softwareschmieden zahlreiche haus eigene Lösungen.

Die große Vielfalt an Produkten führte bisher dazu, dass keiner der Drittanbieter eine überragende Marktposition einnehmen konnte. Der Autor dieses Buchs kann sich hier selbst nicht aus der Affäre ziehen, denn auch er hat in der Vergangenheit einen eigenen OR-Mapper geschrieben, den er aber nun zugunsten des ADO.NET Entity Frameworks aufgeben will. Neben den aktiven Entwicklern von ORM-Werkzeugen für .NET und den passiven Nutzern gibt es eine noch größere Fraktion von Entwicklern, die ORM bisher nicht einsetzen. Meist herrscht Unwissenheit, die auch nicht aufgearbeitet wird, denn es herrscht das Motto »Wenn Microsoft es nicht macht, ist es auch nicht wichtig!«.

## Unterstützte Datenbanken

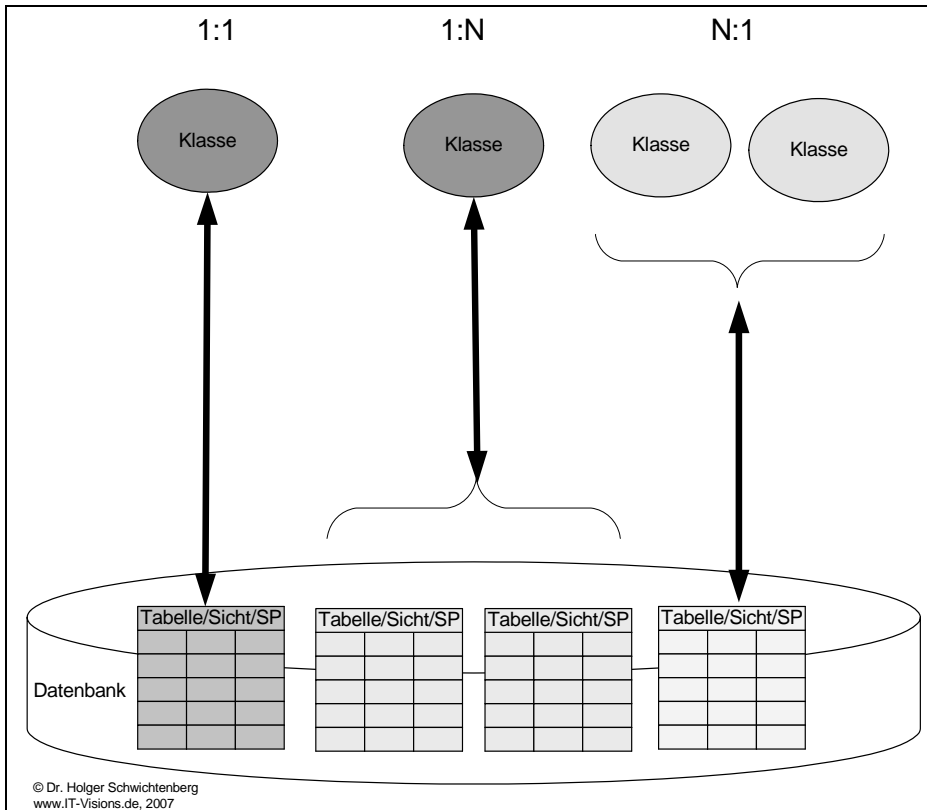
Da SQL und auch Datentypen datenbankspezifisch sind, benötigt ein ORM-Werkzeug für jedes Datenbankmanagementsystem einen speziellen Treiber. Hier scheiden sich zwischen den verfügbaren Werkzeugen stark die Geister, welche Datenbanken unterstützt werden. So unterstützt LINQ to SQL nur Microsoft SQL Server, während das ADO.NET Entity Framework zahlreiche Datenbanken unterstützen soll.

## Abbildungsszenarien

Mapping-Szenarien ist das zentrale Stichwort, dass LINQ to SQL vom EF unterscheidet. Mit einer Ausnahme beherrscht LINQ to SQL nur die Abbildung einer Tabelle auf genau eine Klasse, also alle Zeilen dieser Tabelle werden zu Instanzen dieser Klasse. Lediglich die zu verwendenden Spalten kann man hier einschränken.

Die oben genannte Ausnahme betrifft die Vererbung. LINQ to SQL kann alle Instanzen aller Klassen einer Vererbungshierarchie in einer einzigen Tabelle speichern. Nur mithilfe einer Unterscheidungsspalte (Diskriminator) kann LINQ to SQL auseinanderhalten, welche Zeile zu welcher Klasse gehört. Dieses Verfahren wird in Fachkreisen *Filtered Mapping*, *Shared Mapping*, *Flat Polymorphism* oder *Table per Hierarchy* genannt. Der wesentliche Nachteil des Verfahrens liegt auf der Hand: Diese eine Tabelle muss so breit sein wie die Menge aller Attribute aller Klassen in der Vererbungshierarchie, wodurch es viele immer leere Zellen gibt und somit Speicherplatz vergeudet wird. LINQ to SQL beherrscht weder andere Abbildungsmöglichkeiten für Vererbung, noch die Auflösung der N:M-Zwischentabellen oder die Abbildung einer Klasse auf mehrere Tabellen.

Dabei können die Anforderungen an die Abbildung in der Praxis vielfältig sein. Die nächste Abbildung zeigt die Abbildungsmöglichkeiten, die es allein für den »flachen« Fall gibt. Neben der Abbildung einer Klasse auf eine Tabelle kann man sich wünschen, eine Klasse auf mehrere Tabellen oder eine Tabelle auf mehrere Klassen abzubilden. Dabei hat man immer im Hinterkopf, dass bei der Datenbank Gesichtspunkte wie Effizienz im Zugriff, Vermeidung von Redundanzen und Speicherplatz im Vordergrund stehen, während man bei den Objekten mehr an eine natürliche, intuitive Sicht auf die Aufgabe und den Komfort des Softwareentwicklers denkt. So kann man zum Beispiel die Spalten einer Tabelle in mehrere Unterobjekte (*Composite Types*) gruppieren. Auch ist es nicht absurd, in dem Objektmodell den aktuellen Flug eines Passagiers als Attribute in die Passagier-Klasse zu integrieren, während man in der Datenbank diesen Flug zusammen mit den weiteren vergangenen Flügen korrekt normalisiert in einer Detailtabelle pflegt.



**Abbildung 12.2** Abbildungsszenarien zwischen Klassen und Tabellen beim ORM

Das zweite wichtige Gebiet neben der Abbildung auf Entitätsebene ist die Abbildung von Beziehungen. Relationale Datenbanken verwenden Joins für 0/1:1 und 0/1:n-Beziehungen. N:M-Beziehungen können Sie nicht direkt abbilden. Objektmodelle hingegen verwenden Objektverweise. Eine 1:1-Beziehung ist hier ein Attribut des Typs der anderen Klasse. Mengenbeziehungen werden durch Objektmengen (typisiert z. B. List oder untypisiert z. B. ArrayList) ausgedrückt. Wenn die Klasse Flug eine Liste von Passagier-Objekten und die Klasse Passagier eine Liste von Flug-Objekten aufnehmen kann, dann existiert zwischen beiden eine M:N-Beziehung. Ein gutes ORM-Werkzeug sollte solche wechselseitigen Mengen und deren Abbildung auf die im relationalen Modell notwendigen Zwischentabellen unterstützen.

Auch bei der Vererbung gibt es weitere Wünsche neben dem *Filtered Mapping*. Beim *Vertical Mapping* (alias *Joined Mapping*, *Inheritance Table per Class* oder *Vertical Polymorphism*) gibt es für jede Klasse in der Vererbungshierarchie genau eine Tabelle, also sowohl für die Unterklassen als auch die Basisklassen, selbst dann, wenn diese abstrakt sind. Objekte entstehen hier also immer erst durch einen Join zwischen einer oder mehreren Tabellen. Dieses Verfahren verwendet keinen Speicherplatz, kostet aber Zeit durch die Joins.

Beim *Horizontal Mapping* (alias *Table per Subclass*, *Horizontal Polymorphism*) gibt es Tabellen nur für konkrete Klassen. In diesen Tabellen sind alle Attribute der Klasse (auch die geerbten Attribute) als Spalten enthalten. Wenn man alle Instanzen der Klasse *x* erhalten möchte, muss man nur eine Klasse abfragen. Wenn man hingegen alle Instanzen der Oberklasse *x* und ihrer Unterklassen *y* und *z* benötigt, muss man die Vereinigungsmenge der zugehörigen Tabelle *x*, *y* und *z* bilden. Dies ist jedoch akzeptabel, da dieser Fall seltener ist.

Die Abbildungsinformationen werden in der Regel entweder in XML-Dateien abgelegt oder durch Annotationen im Quellcode gesteuert.

## Abfragesprachen

Bevor es Datenbankeinträge auf Objekte abzubilden gibt, muss der Softwareentwickler zunächst einmal festlegen, welche Daten er in den Speicher laden möchte. In relationalen Datenbanken geschieht dies mit SQL und auch die ORM-Werkzeuge unterstützen SQL. Aber SQL ist nicht die erste Wahl für die Abfrage beim OR-Mapping, denn SQL bezieht sich immer auf Tabellenstrukturen und die dort definierten Tabellen- und Spaltennamen. Wenn man *Flug*-Objekte laden will, dann will man auch mit diesem Namen arbeiten und nicht mit dem Namen der zugrunde liegenden Tabelle *FL\_Fluege*.

Die Object Query Language (OQL) der Object Data Management Group (ODMG) ist hier eine Alternative, die einige ORM-Werkzeuge (z.B. Telerik Open Access und Genome) einsetzt. OQL erkennt man an dem Wort *Extend*, das allen Klassennamen nachzustellen ist, wenn man die Menge der Instanzen der Klasse meint: `Select * from FlugExtend as Flug where Flug.FLAbflugort = 'Rom'`. NHibernate hat seine eigene *Hibernate Query Language* (HQL). NDO hat ebenfalls eine eigene objektorientierte Abfragesprache, für die der Hersteller aber keinen Namen benannte.

Der Trend geht jedoch zur Language Integrated Query (LINQ) mit der eleganten Integration in die Sprachsyntax von C# und Visual Basic (seit den 2008er Versionen). Die Verwendung von LINQ ist durch die dokumentierten Schnittstellen nicht alleine Microsoft vorbehalten und so gibt es schon zahlreiche LINQ-Provider von Drittanbietern (siehe Liste der Provider im Kapitel 10 »Language Integrated Query (LINQ)«), darunter auch einige ORM-Werkzeuganbieter. Das Entity SQL (eSQL) im ADO.NET Entity Framework ist vergleichbar mit HQL und OQL.

---

### HINWEIS

In die Sprachsyntax integriertes LINQ ist immer eine statische Abfrage. Statisch bedeutet, dass es in der Abfrage immer die gleichen Konstrukte gibt, man kann z. B. nicht zwei oder vier Bedingungen in WHERE angeben, sondern man hat immer genau vier. Dynamische LINQ-Befehle erreicht man durch die Erstellung von Ausdrucksbäumen (Expression Trees) per Programmcode.

---

Interessant ist die Frage, ob ein ORM-Werkzeug neben einer eleganten und datenbankneutralen Objektabfragesprache auch noch die direkte Verwendung von datenbankspezifischem SQL anbietet. Sinn kann datenbankspezifisches SQL machen, wenn man optimieren will. Bei einer Objektabfragesprache werden die SQL-Befehle generiert und der Entwickler ist diesem Generator »ausgeliefert«. Eine Objektabfragesprache ist also nichts für »SQL-Kontroll-Fetischisten«.

## Ladestrategien

Die Definition der Abfrage selbst ist aber noch nicht alles, was über die tatsächlich zu ladenden Objekte entscheidet. Eine entscheidende Frage für die Leistungsfähigkeit von ORM ist die Frage, wann verbundene Objekte zu laden sind. Wenn man zu einem Flug sofort alle Buchungen und die dazugehörigen Passagier- und Personendaten lädt, lädt man unter Umständen mehr als wirklich gebraucht wird. Holt man die Daten jedoch nicht, müssen sie beim Zugriff auf eine Objektreferenz nachgeladen werden. Das Nachladen bei Bedarf nennt man *Lazy Loading*, *Deferred Loading* oder *Delayed Loading* im Gegensatz zum *Eager Loading* (alias *Immediate Loading*). Lazy Loading ist der Standard in allen ORM-Werkzeugen, Eager Loading kann man auf Ebene einer Datenbankverbindung (alias Datenkontext oder Data Scope) oder einer einzelnen Anfrage steuern.

Lazy Loading beinhaltet eine besondere Herausforderung, denn das ORM-Werkzeug muss jeglichen Zugriff auf alle Objektreferenzen »abfangen«, um hier bei Bedarf die verbundenen Objekte nachladen zu können. Dieses Abfangen erfolgt durch die Verwendung bestimmter Klassen für Einzelreferenzen und Mengenklassen. Der Unterschied zwischen den ORM-Werkzeugen liegt hier darin, ob der Entwickler diese Klassen explizit im Code verwenden muss oder ob das ORM-Werkzeug diese beim Kompilieren oder zur Laufzeit austauscht.

**TIPP**

Die Wahl zwischen Lazy Loading und Eager Loading ist oft auch die Wahl zwischen Pest und Cholera: Wenn man nicht genau weiß, ob die zusätzlichen Daten benötigt werden oder nicht, dann kann es ungünstig sein, sie direkt zu laden. Es kann aber auch ungünstig sein, sie später nachladen zu müssen. Wichtige Entscheidungskriterien sind:

- Wie wahrscheinlich ist, dass die Daten benötigt werden?
- Wie groß ist die Datenmenge?
- Kann ich überhaupt später die Daten einfach nachladen? (Wenn die Daten zwischenzeitlich serialisiert wurden, ist ein automatisches Nachladen in der Regel nicht mehr möglich!)

Allen ORM-Werkzeugen ist gemein, dass die Mengenreferenzen nicht völlig beliebig sind, sondern nur ein bestimmtes Spektrum von Objektmengenklassen (meist festgemacht an bestimmten Schnittstellen) unterstützt werden (Details siehe Vergleichstabelle). Der Grund dafür liegt in der Unterstützung von Lazy Loading.

**ACHTUNG**

Bitte bedenken Sie, dass ORM nicht für alle Datenzugriffsszenarien die beste Lösung ist. Wenn Sie Massendatenänderungen (*Batch-Verarbeitung*) durchführen wollen, z. B. alle Preise um 3% erhöhen, dann ist es besser, wenn Sie einen entsprechenden SQL-UPDATE-Befehl direkt an die Datenbank senden. Mit ORM müssten Sie erst alle Produkte laden und dann einzeln (!) speichern, weil bisher kein ORM intelligent genug ist, eine solche Massendatenänderung auf einen Befehl zu optimieren. Der Zeitaufwand ist dann viel höher!

## Forward Engineering vs. Reverse Engineering

Ein wesentliches Unterscheidungskriterium beim Objektrelationalen Mapping ist die Henne- oder Ei-Frage. Übertragen auf die Welt der OR-Mapper geht es darum, ob der Entwickler zuerst die Datenbank erstellt und daraus dann Geschäftsobjekte generieren lässt, oder erst die Geschäftsobjekte erstellt und daraus dann die Datenbank generieren lässt. Oder er macht beides manuell (d.h. er erstellt Geschäftsobjekte und Datenbank unabhängig voneinander) und definiert dann eine (komplexe) Abbildung zwischen den Welten.



Vielen Datenbankadministratoren sträuben sich sicherlich die Haare, wenn man ihnen erzählt, dass ein Entwicklungswerkzeug Datenbanken generieren will. Doch in der Java-Welt ist dies seit Langem vorhanden (Container Managed Persistence, CMP), wenn auch nicht ohne Kritik [DEVX01]. Dabei legt die Wortwahl der ORM-Werkzeuganbieter nahe, dass gerade dies der üblichere Weg ist. Wenn man erst Geschäftsobjekte erstellt und die Datenbank generieren lässt, sprechen sie von *Forward Engineering*. Den etwas negativ vorbelegten Begriff *Reverse Engineering* benutzen sie für Ansätze, in denen es die Datenbank zuerst gibt. Etwas neutraler werden die Begriffe, wenn man *Mapping* statt *Engineering* verwendet oder von *Modell First* oder *Code First* oder *Domain First* im einen bzw. *Database First* im anderen spricht.

Reverse Mapping können alle Werkzeuge, Forward Mapping hingegen nicht. Die anderen Anbieter unterscheiden sich dadurch, ob die Generierung der Datenbank zur Entwicklungszeit oder zur Laufzeit erfolgt. Eine besondere Herausforderung besteht bei Veränderung des Datenbankschemas. Ein gutes ORM-Werkzeug kann bei solchen Schemaänderungen die Abbildungsinformationen aktualisieren.

## Geschäftsobjektklassen

Interessant ist, welche Anforderungen es an die Geschäftsobjektklasse seitens des ORM gibt. Im Idealfall kann man jede beliebige einfache .NET-Klasse auf eine Datenbanktabelle abbilden, ohne dass die Klasse irgendeine Voraussetzung erfüllen muss. Die Fachwelt spricht dann von *Persistence Ignorance* bzw. von *Plain Old CLR Objects* (POCOs). Nicht alle Anbieter unterstützen POCO. Einige Werkzeuge haben mehr oder weniger einschränkende Anforderungen, z.B.:

- Eine Annotation durch ein .NET-Attribut
- Eine bestimmte Basisklasse für die Geschäftsobjektklassen
- Die Anforderung, die Geschäftsobjektklasse als abstrakte Basisklasse zu definieren

Wenn die Geschäftsobjektklasse im Rahmen des Reverse Mapping generiert wird, dann ist es wichtig zu sehen, welche Unterstützungen und Erweiterungsmöglichkeiten diese Klasse mitbringt. Eine gute Unterstützung für Datenbindung in Windows Forms-Anwendungen erfordert die Schnittstellen *INotifyPropertyChanging* und *INotifyPropertyChanged*. Einige Anbieter verfolgen hier aber eigene Strategien (z.B. *ObjectView* bei Telerik Open Access).

In der Welt von ASP.NET erfolgt die Datenbindung am einfachsten über Datenquellensteuerelemente. Microsoft bietet hier eine *LinqDataSource* (enthalten ab ASP.NET 3.5) und *EntityDataSource* (enthalten ab ASP.NET 3.5 SP1). In anderen ORM-Werkzeugen müssen Sie derzeit selbst eine passende Manager-Klasse schreiben, um die seit ASP.NET 2.0 vorhandene *ObjectDataSource* zur Datenbindung zu nutzen. Einige Anbieter haben wieder eigene Lösungen (z.B. *GenomeDataSource* bei Genome).

## Mapping-Interna (Objekt-Materialisierung)

Interessant ist, wie die Werkzeuge intern arbeiten. Nicht alle Werkzeuge verwenden im Untergrund ADO.NET, einige aus der Java-Welt portierte Werkzeuge basieren intern auf Java Database Connectivity (JDBC). Wenn ADO.NET zum Einsatz kommt, dann zum Datenlesen natürlich kein *DataSet* mit seinem großen Overhead, sondern ein schlanker *Datareader*. Das Zurückschreiben von Änderungen erledigen die ORM-Werkzeuge – aus Gründen der Leistung und Sicherheit – am besten mit parametrisierten SQL-DML-Befehlen.

Um das eigentliche Mapping zwischen Datenbankzugriffsobjekt (z.B. `DataReader`) und Geschäftsobjekt zur Laufzeit auszuführen, braucht ein ORM-Werkzeug Programmcode, der die Geschäftsobjekte instanziiert und die einzelnen Informationseinheiten aus der Tabelle entnimmt und in das Geschäftsobjekt überträgt. Dies bezeichnet man als Objekt-Materialisierung. Grundsätzlich könnte man dies per .NET Reflection allgemein lösen, aber Reflection ist viel zu langsam. Microsoft LINQ to SQL, die Entity Framework Object Services und NHibernate nutzen daher Reflection in Kombination mit der Codegenerierung zur Laufzeit, d.h. Programmcode in Intermediate Language (IL) zur Abbildung zwischen Tabelle und Objekt wird zur Laufzeit erzeugt. Einige Werkzeuge hingegen erzeugen den zusätzlichen Programmcode zur Entwicklungszeit durch zusätzliche Kompilierungsschritte in Visual Studio. Dabei werden entweder die von Visual Studio erzeugten Assemblys mit weiteren IL-Befehlen angereichert (*Enhancing*) oder eine eigenständige Assembly generiert.

Viele ORM-Werkzeuge bilden entweder die Spalten auf einfache Attribute (*Fields*) ab oder Properties (oder beides). Wichtig ist auch die Unterstützung für wertelose Wertetypen (*Nullables*). Das heißt, dass die ORM-Werkzeuge im Gegensatz zu ADO.NET eine leere Integer-Zelle tatsächlich als `int?` (bzw. `Integer?` oder `System.Nullable<Int32>`) und nicht als `DBNull` signalisieren.

## Änderungsverfolgung und Persistierung

Nachdem ein Objekt in den Hauptspeicher geladen wurde, überwachen die ORM-Werkzeuge die Objekte auf Änderungen (Change Tracking), sodass beim Speichern nur die tatsächlich geänderten Objekte übertragen werden müssen. Es wäre nicht praktikabel, alle Objekte erneut zu speichern. Wünschenswert ist zudem, dass der ORM auf Attributebene die Änderungen bemerkt und daher nur die Spalten aktualisiert, deren Werte geändert werden müssen und nicht alle Spalten der zu dem Objekt gehörenden Tabellenzeile.

Beim Speichern von Objekten muss ein ORM-Werkzeug Datenbanktransaktionen unterstützen, auch in der eleganten Form über den schon in .NET 2.0 eingeführten Namensraum `System.Transactions`. Einige Werkzeuge wollen bei jeder einzelnen Änderung eine explizite Transaktion. Das ist inhaltlich nicht schlimm, schont aber nicht die Fingerkuppen der Entwickler. Alle ORM-Werkzeuge unterstützen von der Datenbank automatisch vergebene Werte (z.B. automatisch hochzählende Primärschlüssel oder Zeitstempel), das heißt nach dem Persistieren eines neu angelegten Objekts bzw. dem Ändern eines Objekts mit Zeitstempel fragen sie bei der Datenbank noch einmal mit einem `SELECT` nach den aktuellen Werten.

Für den gleichzeitigen Zugriff unterstützen ORM-Werkzeuge das aus ADO.NET bekannte optimistische Sperren, in der Regel werden Konflikte über Originalwertvergleich, Zeitstempel oder auch Versionsnummer erkannt. Pessimistisch Sperren kann der Entwickler bei ORM-Werkzeugen auch nur über Transaktionen, sofern das Werkzeug nicht einen eigenen Mechanismus implementiert.

## Objektcontainer

Für die Änderungsverfolgung ist ein Container notwendig, in dem die Objekte im RAM liegen. Diese Objektcontainer haben in allen Werkzeugen verschiedene, aber zum Teil ähnliche Namen: `DataContext`, `ObjectContext`, `ObjectScope`, `Session`, `PersistenceManager` oder `DataDomain`. Ihnen gemeinsam ist, dass der Entwickler den Container vor der ersten Abfrage erzeugen und die Abfrage dann dort aufrufen muss, sodass die Container »mitbekommen«, welche Objekte mit welchem Ausgangszustand geladen werden.

Die Objektcontainer öffnen die Datenbankverbindung meist selbst. Einige Werkzeuge erlauben auch, dass der Benutzer eine ADO.NET-Datenbankverbindung selbst erstellt und den Objektcontainer dann damit arbeiten lässt.

## Zwischenspeicherung (Caching)

Alle Objektcontainer speichern die geladenen Objekte zwischen (Caching) und verhindern so, dass ein Objekt unnötig mehrfach geladen werden muss. Auch Inkonsistenzen verhindern die Container, wenn der Entwickler ein bereits im RAM verändertes Objekt erneut durch eine andere Abfrage lädt. Alle Drittanbieter mieten darüber hinaus einen *Zwischenspeicher auf zweiter Ebene (Second Level Cache)*, der containerübergreifend ist. NHibernate bietet hier mit der Auswahl zwischen der Zwischenspeicherung im Hauptspeicher, im ASP.NET-Zwischenspeicher, in einer anderen Datenbank oder im Dateisystem die bei Weitem größte Auswahl. Genome erlaubt genau wie NHibernate, selbst einen Second Level Cache zu schreiben.

Sehr interessant ist das Konzept der verteilten Zwischenspeicherung: Verschiedene Anwendungsserver mit einem Zwischenspeicher können Änderungen synchronisieren (z.B. über MSMQ-Nachrichten).

## Serialisierung

Wenn ein Geschäftsobjekt die Prozessgrenze verlassen muss (z.B. im Rahmen eines Webservices), dann ist seine Serialisierbarkeit wichtig. Dafür sollte ein Werkzeug sowohl die Auszeichnung mit der schon in .NET 1.0 eingeführten Annotation `[System.Serializable]` als auch mit dem in .NET 3.0 eingeführten `[System.Runtime.Serialization.DataContract]` bieten. Im Idealfall wäre eine Serialisierung ganz ohne eine Annotation möglich.

Bei der Serialisierung muss man unterscheiden, ob nur einzelne Objekte oder auch damit verbundene (assoziierte) Objekte serialisiert werden können (Graph Serialization).

## Unterstützung für verteilte Systeme

Serialisierbarkeit ist aber nicht alles bei ORM. Solange ein Objekt im gleichen Prozess wie der Kontext ist, überwacht der Kontext alle Änderungen an dem Objekt (Change Tracking). Die Änderungsverfolgung wird zu einem Problem, wenn die Geschäftsobjekte länger leben als der ORM-Container, zum Beispiel in Webanwendungen oder anderen zustandslosen Anwendungsservern. Hier spricht man von Mehrschichtunterstützung, *Disconnected Objects* oder *Detached Objects*. Bei fast allen ORM-Werkzeugen ist es möglich, Geschäftsobjekte aus dem Container zu lösen (entweder durch einfaches Schließen des Containers oder eine explizite Operation). Große Unterschiede gibt es aber dann, wenn diese Objekte später wieder einem Container angefügt werden sollen. Dies ist insbesondere dann notwendig, wenn der Entwickler Änderungen in den Objekten persistieren will. Der neue Container muss aber wissen, ob es sich bei dem angefügten Objekt um ein geändertes, unverändertes oder neues Objekt handelt. Man braucht also einen serialisierbaren Objektcontainer, der wie ein DataSet in ADO.NET die Prozessgrenzen überschreiten kann und die in ihm lebenden Objekte beobachtet. Einige Anbieter übertragen die Änderungsverfolgung bei losgelösten Objekten aber dem Entwickler, was zu ein wenig aufwändiger Programmierung im Client führt.

## Plattformen

Ein wesentlicher Unterschied zwischen den ORM-Werkzeugen sind die unterstützten Plattformen. Während Microsofts Lösungen auf jeden Fall .NET 3.5 voraussetzen, sind viele Drittanbieterprodukte ab .NET 2.0 oder sogar mit .NET 1.x lauffähig. Mono und das .NET Compact Framework unterstützen nur sehr wenige Produkte.

## Werkzeuge

Erhebliche Unterschiede gibt es auch bei der Werkzeugunterstützung. Ideal ist ein in Visual Studio integrierter, grafischer Designer für ORM, bei dem man die Abbildung zwischen Geschäftsobjektklasse und Tabelle grafisch definieren kann. Alternativ möglich, aber weniger flexibel, sind Assistenten-gesteuerte oder Dialog-gesteuerte IDE-Werkzeuge für das ORM. Eine Unterstützung durch Kommandozeilenwerkzeuge ist sehr wünschenswert, sollte aber nicht die einzige Möglichkeit sein.

## Überwachung

Im Bereich der Überwachung sollte der ORM die Möglichkeit bieten, die erzeugten SQL-Befehle zu protokollieren. Schön wären auch Windows-Leistungsindikatoren (*Performance Counter*) zur Überwachung der Leistung des ORM.

## Erweiterbarkeit

Wie so oft bei der Bewertung von Software geht es auch bei den ORM-Werkzeugen um die Frage der Erweiterbarkeit, also die Möglichkeiten für fortgeschrittene Anwender, die Funktionen des Produkts zu erweitern und so bestehende Restriktionen aufzuheben. Alle Anbieter ermöglichen es, über eigenen Programmcode den Lebenszyklus eines Geschäftsobjekts und des Objektcontainers zu überwachen und zu beeinflussen. Einige Anbieter sprechen hier von *Interception*, andere von *Lifecycle Events* (*Lebenszyklusereignis*) oder *Hooks*.

Interessant ist die Frage, ob man selbst andere Datenbankmanagementsysteme anbinden kann, für die der Anbieter noch selbst keine Treiber liefert. Open Source-Projekte wie NHibernate sind hier naturgemäß im Vorteil.

## Beispiel für dieses Kapitel

Bitte beachten Sie, dass in diesem Kapitel bewusst eine Variante des WorldWideWings-Datenmodells verwendet wird, die das Mapping mit dem ADO.NET Entity Framework optimal unterstützt. Das bedeutet nicht, dass das ältere Schema das Mapping nicht unterstützt. Das in diesem Kapitel verwendete Schema ist so aufgebaut, das möglichst wenige manuelle Anpassungen im Modell notwendig sind, um »schöne« Klassennamen zu erhalten.

In der Schema-Version 6.3 des Datenmodells gibt es insbesondere folgende Änderungen gegenüber dem im ADO.NET-Kapitel verwendeten Schema:

- Alle Tabellennamen sind im Singular
- Auf Vorsilben (Prefixe) wird komplett verzichtet
- Die Buchungen sind nicht in der Tabelle *GF\_GebuchteFluege*, sondern in *Flug\_Passagier* abgelegt worden. Der Unterschied zwischen beiden Tabellen ist: *Flug\_Passagier* besteht nur aus den Fremdschlüsseln von *Passagier\_PersonID* und *Flug\_FlugNr* und ist daher eine reine Zwischentabelle, während *GF\_GebuchteFluege* weitere Informationen (z. B. Buchungscode) enthält. *GF\_GebuchteFluege* könnte durch das EF nicht »wegoptimiert« werden und bietet sich daher hier als Beispiel nicht an.
- Um Vererbung zu ermöglichen, verwenden die Tabellen *Person*, *Passagier*, *Mitarbeiter* und *Pilot* alle *PersonID* als Primärschlüssel

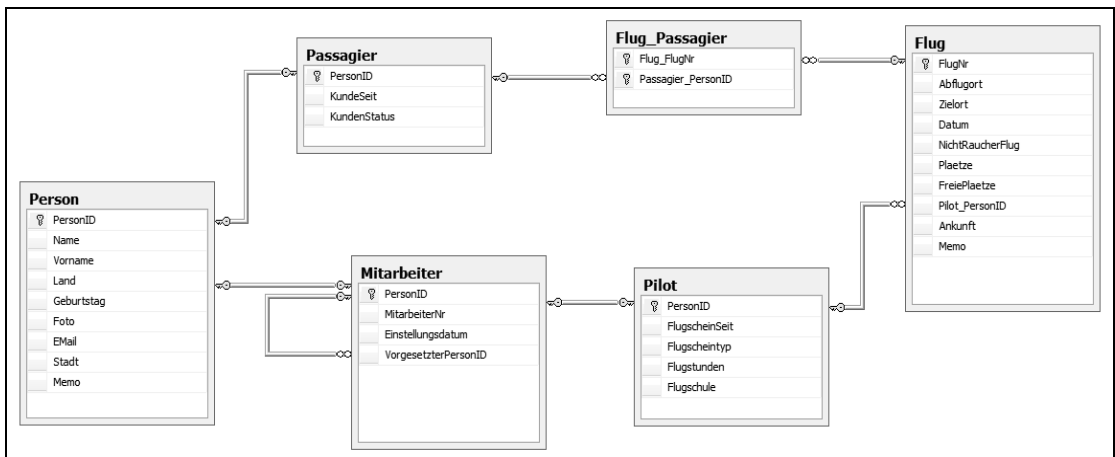


Abbildung 12.3 Eine leicht modifizierte Version der WorldWideWings-Datenbank

## Grundbegriffe und Grundkonzepte des ADO.NET Entity Framework

Die erste Version des ADO.NET Entity Framework, die im Rahmen von .NET Framework 3.5 Service Pack 1 und Visual Studio 2008 Service Pack 1 im August 2008 erschien war, konnte nicht in allen Szenarien überzeugen. Die zweite Version im Rahmen des .NET Framework 4.0 und Visual Studio 2010 nennt sich nicht ADO.NET Entity Framework 2.0, sondern ADO.NET Entity Framework 4.0. ADO.NET Entity Framework 4.0 (kurz: EF4) bietet mehr und verbesserte Funktionen sowie in einigen Fällen einen deutlichen Geschwindigkeitszuwachs.

Dieser Abschnitt erläutert zunächst einige Grundbegriffe und Grundkonzepte des ADO.NET Entity Framework.

## Bausteine des ADO.NET Entity Framework

Das .NET Entity Framework umfasst folgende Bausteine und Grundbegriffe:

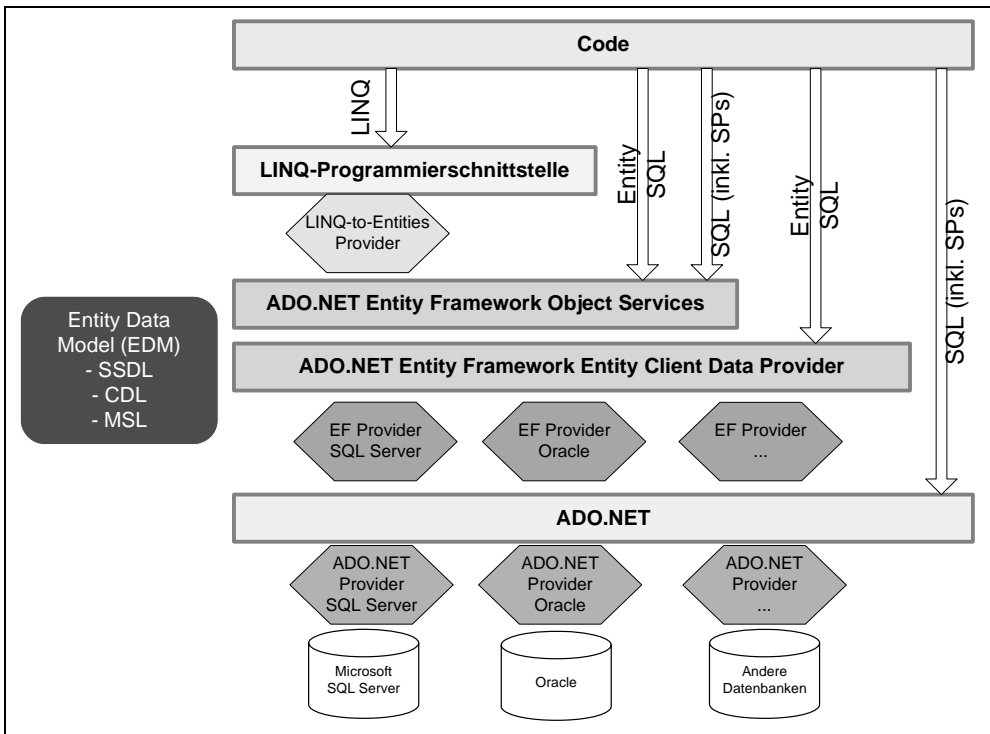
- Das Kernkonzept des EF ist die Abbildung physikalischer Datenstrukturen (Tabellen in Datenbanken) auf konzeptionelle Datenstrukturen (Entitäten, wie man sie aus dem Entity Relationship-Modell kennt) durch das *Entity Data Model (EDM)*. Das geschieht erstmal unabhängig von dem Objektrelationalen Mapping auf Tabellenebene. Es handelt sich also um ein Relational-zu-Relational-Mapping. Eine N:M-Zwischentabelle wird hier also schon auf Tabellenebene beseitigt.
- Zur Erstellung eines Modells liefert Microsoft einen grafischen *Designer* in Visual Studio und das *Kommandozeilenwerkzeug EdmGen.exe*
- Abfragen auf einem EDM-Modell erfolgen nicht mit SQL, sondern per *Entity SQL (eSQL)*, dem datenbankneutralen SQL-Dialekt des Entity Framework
- *LINQ to Entities* ist die LINQ-basierte Abfragesprache für die Object Services, die alternativ zu Entity SQL oder klassischem SQL eingesetzt werden kann, um auf Ebene der Geschäftsobjekte mithilfe der LINQ-Syntax abzufragen. LINQ to Entities arbeitet auf den Entitäten und Entitäten werden auf Tabellen abgebildet.
- Der ADO.NET-Datenbanktreiber *EntityClient* erlaubt die Ausführung von eSQL-Abfragen
- Der EntityClient verwendet einen so genannten *Entity Framework Provider (EF Provider)*, um auf ADO.NET zuzugreifen und ADO.NET verwendet dann einen ADO.NET Data Provider für den eigentlichen Datenzugriff
- Die ADO.NET Entity Framework *Object Services* vollbringen das Objektrelationale Mapping. Hier werden Entitäten auf Objekte abgebildet

### HINWEIS

Das EF zielt nicht nur auf das ORM in .NET, sondern ist weitreichender durch die Möglichkeit, die Tabellenabbildungen auch unabhängig vom Objektrelationalen Mapping zu verwenden. Microsoft plant, EF-Abbildungen zukünftig auch für SQL Server-Dienste wie Reporting (SSRS), Analysis (SSAS) und Integration (SSIS) bereitzustellen.

## Architektur des ADO.NET Entity Framework

Die folgende Abbildung zeigt die Schichtenarchitektur des Entity Framework: Programmcode greift über LINQ, Entity SQL oder klassisches SQL auf die Object Services zu. Darunter liegt der Entity Client Data Provider, der auch direkt per Entity SQL angesprochen werden kann. Der Entity Client Data Provider verwendet datenbankspezifische Entity Framework-Provider (für die Umsetzung in SQL-Befehle, Umsetzung der Datentypen etc.). Darunter liegt das klassische ADO.NET mit den ADO.NET Providern. Der Pfeil SQL (inkl. SPs) zeigt an, dass Programmcode auch weiterhin parallel direkt auf ADO.NET zugreifen kann. Entity Framework und klassisches ADO.NET können sich sogar eine Datenbankverbindung teilen. An allen Stellen, wo SQL eingesetzt werden kann, kann man alternativ auch mit Gespeicherten Prozeduren (Stored Procedures, SP) arbeiten.



**Abbildung 12.4** Architektur des ADO.NET EF (Quelle: MSDN)

## Konzeptionelle Datenmodelle

Unter dem *konzeptionellen Datenmodell* wird in der Literatur (vgl. z.B. [JAR01, Seite 24ff.]) eine von der physikalischen Speicherform unabhängige Beschreibung von Datenstrukturen bezeichnet. Das konzeptionelle Modell legt Objekttypen, die Eigenschaften der Objekttypen, die Identifizierungsmerkmale der Instanzen des Objekttyps und die Zusammenhänge (Beziehungen) zwischen Objekttypen fest. Üblicherweise wird das konzeptionelle Modell heute durch das Entity Relationship Model (ERM) oder die Unified Modeling Language (UML) ausgedrückt.

Der Datenzugriff erfolgte in .NET bisher mit ADO.NET 1.x und 2.0 jedoch auf der Ebene des logischen Datenbankschemas, d.h. man verwendet Tabellen und Sichten (Views) direkt. Der Unterschied zwischen dem konzeptionellen Datenmodell und dem logischen Datenbankschema wird besonders deutlich an einer N-zu-M-Beziehung: In der WorldWideWings-Datenbank gibt es Flüge und Passagiere: Jeder Passagier kann beliebig viele Flüge buchen und auf einem einzelnen Flug ist kein Passagier alleine an Bord. Zur Modellierung dieser Beziehung braucht man auf der konzeptionellen Ebene zwei Objekttypen (*Flug* und *Passagier*), im logischen Datenbankschema einer relationalen Datenbank sind aber drei Tabellen (*Flug*, *Passagier* und die Zwischentabelle *Flug\_Passagier*) notwendig.

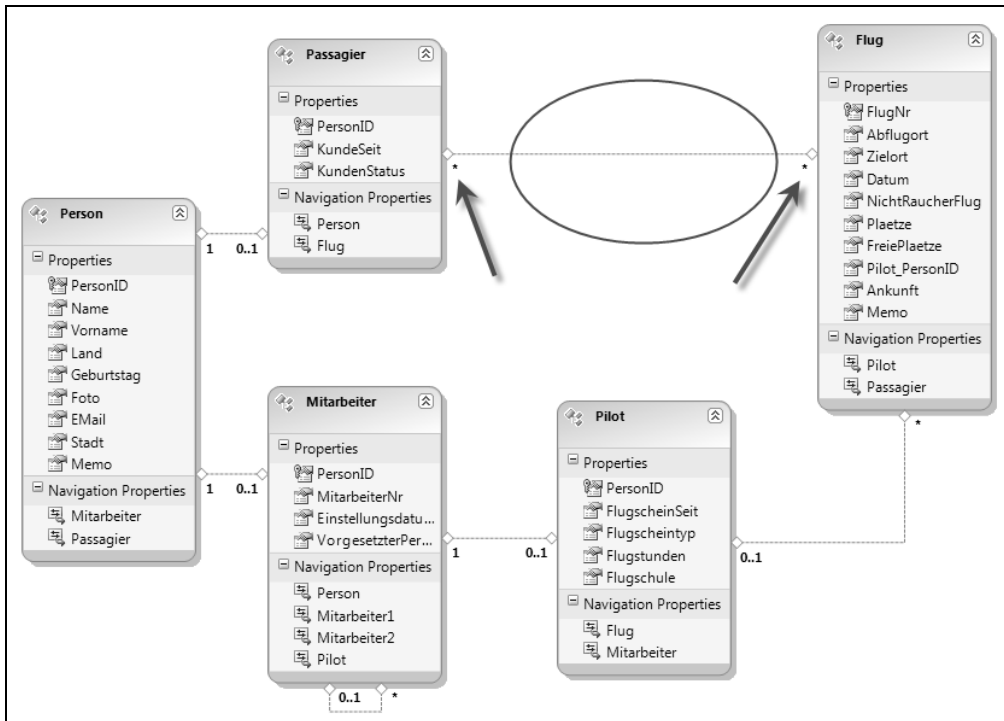


Abbildung 12.5 Ein konzeptuelles Modell für das Datenmodell aus Abbildung 12.3

## Entity Data Model (EDM)

Das Entity Data Model (EDM) ist eine XML-Sprache zur konzeptionellen Beschreibung von Datenstrukturen aller Art (relationale Daten, XML-Daten, .NET-Objekte). EDM unterstützt komplexe Datentypen, Vererbung und Beziehungen (Assoziationen). EDM-Beschreibungen können auf anderen EDM-Beschreibungen oder Datenspeichern abgebildet werden.

### EDM-Bestandteile

EDM besteht aus drei Teilen:

- **SSDL** Store Schema Definition Language (*.ssdl-Datei*) mit der Beschreibung des Schemas der Datenbank
- **CSDL** Conceptual Schema Definition Language (*.csdl-Datei*) mit der Beschreibung des konzeptuellen Modells und zugehöriger Quellcodedatei (*.cs* oder *.vb*)
- **MSL** Mapping Specification Language (*.msl-Datei*) zur Abbildung von Datenbankschemata auf EDM-Schema. MSL ist eine XML-Sprache zur Beschreibung der Abbildung eines konzeptionellen Datenmodells in der Conceptual Schema Definition Language (CSDL) auf ein Datenspeicherschema, das durch die Store Schema Definition Language (SSDL) beschrieben wird.



## EDM-Format

Alle EDM-Dateien sind XML-Dateien und könnten daher mit einem beliebigen Editor erstellt werden, was aber sehr mühsam ist. Besser ist die Erstellung eines EDM-Modells über das *Kommandozeilenwerkzeug* *EdmGen.exe* und noch einfacher mit dem EDM-Designer in Visual Studio.

---

**HINWEIS** Eine *.edmx*-Datei ist die Zusammenfassung der drei o.g. Dateitypen zu einer Datei. Der grafische EDM-Designer in Visual Studio erstellt *.edmx*-Dateien. Der Compiler trennt diese Datei dann auf.

---

## Abbildungsszenarien

Das Entity Framework unterstützt (fast) beliebige Abbildungen. Insbesondere kann das ADO.NET Entity Framework reine Zwischentabellen in N:M-Beziehungen eliminieren und unterstützt bei der Vererbung auch *Vertical Mapping* (alias *Joined Mapping*, *Inheritance Table per Class* oder *Vertical Polymorphism*) sowie *Horizontal Mapping* (alias *Table per Subclass*, *Horizontal Polymorphism*). LINQ to SQL arbeitet direkt auf dem Datenbankschema, nicht auf dem konzeptuellen Modell, das bei der Entity-Relationship-Modellierung (ERM) verwendet wird. Das ADO.NET Entity Framework unterstützt Mapping auch auf Tabellenebene (d.h. auch ohne Objektrelationales Mapping). LINQ to SQL bietet das nicht.

---

**ACHTUNG** Eine der Abbildungseinschränkungen des Entity Framework gibt es bei Vererbung: Die »erbenden« Tabellen müssen den gleichen Primärschlüssel wie die Elterntabellen besitzen. Mit Fremdschlüsseln darf man hier nicht arbeiten.

---

## Assemblies und Namensräume

Das Entity Framework ist implementiert in den Assemblies `System.Data.Entity.dll` und `System.Data.Entity.Design.dll`. Die wichtigsten Namensräume sind `System.Data.Objects` und `System.Data.Objects.DataClasses`. Weitere Namensräume sind `System.Data.Objects.SqlClient`, `System.Data.Metadata.Edm`, `System.Data.Mapping`, `System.Data.Entity.Design` und `System.Data.EntityClient`. Außerdem wird der Namensraum `System.Linq` aus der `System.Core.dll` gebraucht.

## Unterstützte Datenbanken

Für das Entity Framework gilt das Gleiche, was inzwischen auch für das klassische ADO.NET gilt: Microsoft unterstützt nur den eigenen Microsoft SQL Server (der Oracle-Treiber, den es gab für das klassische ADO.NET bietet Microsoft in neueren .NET-Versionen nicht mehr an).

Microsoft unterstützt bei Entity Framework aber genau wie beim klassischen ADO.NET Drittanbieter darin, Treiber zu entwickeln.

Folgende Anbieter bieten Entity Framework-Treiber für andere Datenbanken an (Quelle: [MSDN33]):

- **Devart** Oracle, MySQL, PostgreSQL
- **Phoenix** SQLite
- **Npqsql** PostgreSQL
- **Sybase** SQL Anywhere

- **IBM** Informfix, DB2, U2
- **Firebird** Firebird
- **Synergex** Synergy
- **DataDirect** Oracle
- **OpenLink** Virtuoso, ODBC, JDBC, Oracle, SQL Server, DB2, Sybase Informix, Ingres, Progress, MySQL, PostgreSQL, Firebird

---

**HINWEIS** Oracle selbst hat schon länger einen eigenen Entity Framework-Provider, der ist aber zum Redaktionsschluss dieses Buchs noch nicht verfügbar. Der Autor dieses Buchs arbeitet beim Zugriff auf Oracle-Datenbanken mit dem Devart-Treiber.

---

---

**HINWEIS** Aus Platzgründen ist es nicht möglich, verschiedene Datenbankmanagementsysteme in diesem Buch zu berücksichtigen. Daher arbeiten alle Beispiele in diesem Buch mit Microsoft SQL Server. Sie können wahlweise die Version 2005 oder 2008 oder 2008 R2 einsetzen sowie SQL Server Express oder eine kommerzielle Version. Bitte beachten Sie, dass das Entity Framework 4.0 den SQL Server 2000 nicht mehr unterstützt.

---

## Überblick über die Object Services

Die ADO.NET Entity Framework Object Services sind der eigentliche ORM des Entity Framework. Die Object Services bestehen aus Klassen im Namensraum `System.Data.Objects.DataClasses` sowie den vom EDM-Designer generierten Kontext- und Entitätsklassen.

Die ADO.NET EF Object Services unterstützen u.a.:

- Abfragen mit eSQL, SQL und LINQ to Entities
- Navigationsbeziehungen zwischen Entitätsklassen
- Verzögertes Laden (Lazy Loading)
- Änderungsverfolgung
- Speichern von neuen, geänderten und gelöschten Objekten (auch mit Transaktionen)
- Optimistisches Sperren
- Serialisierung von Objektbäumen

## Architekturmodelle

Viele Entwickler fragen sich, welche Rolle das Entity Framework in einer mehrschichtigen Softwarearchitektur (Benutzerschnittstellensteuerung, Geschäftslogik, Daten-/Ressourcenzugriff) einnehmen kann. Im Folgendem sind drei typische Architekturmodelle grafisch skizziert:

- In Modell 1 übernehmen die Entitätsklassen die Rollen der Datenobjekte (alias Geschäftsobjekte), der Entity Framework-Objektkontext die Rolle der Datenzugriffsschicht. Dies bedeutet, die Steuerbefehle des Kontextes (Abfragen/Speicherbefehle) müssen aus der Geschäftslogik ausgeführt werden. Dies könnte man als eine Vermischung von Logik und Datenzugriff ablehnen. In vielen Projekten gibt es aber wenig Logik im engeren Sinne, sodass eine weitere Trennung hier nur unnötige Arbeit machen würde. Den Objektkontext des Entity Framework könnte man als hinreichend abstrakt (als Steuereinheit einer generischen Datenzugriffsschicht) ansehen. Das Modell ist sehr »schlank«, weil man wenig Code (im Vergleich zu den anderen Modellen) schreiben muss.
- Im Modell 2 gibt es oberhalb des Entity Framework-Kontextes eine eigene Datenzugriffsschicht.
- In Modell 3 werden die Entitätsklassen des Entity Framework in der Datenzugriffsschicht zusätzlich auf eigene Klassen abgebildet. Hier ist der Implementierungsaufwand sehr groß. Der Vorteil liegt aber darin, dass die Logik und die Benutzeroberflächensteuerung keinerlei Abhängigkeiten vom Entity Framework haben.

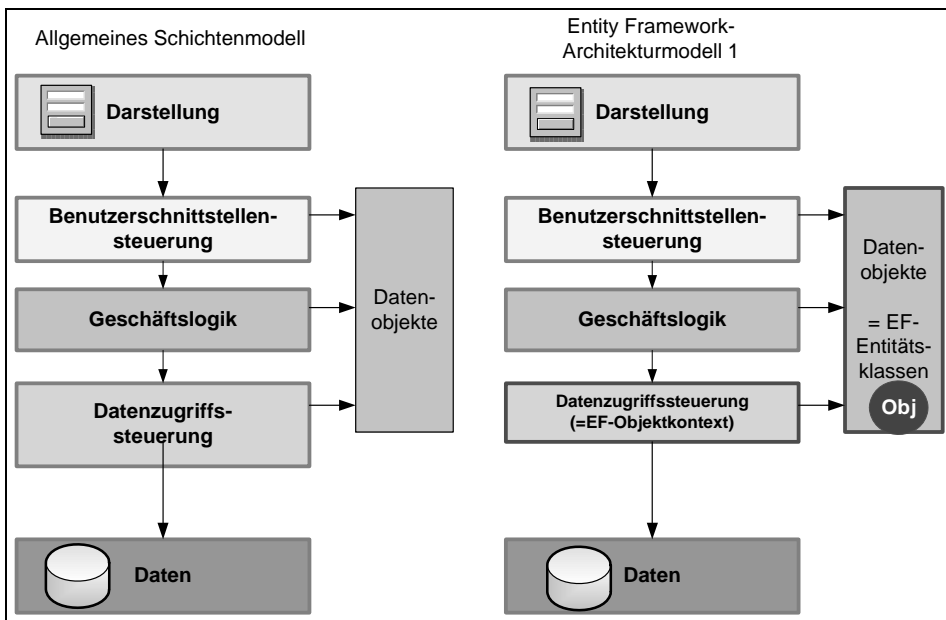


Abbildung 12.6 Allgemeines Schichtenmodell und Entity Framework-Architekturmodell 1

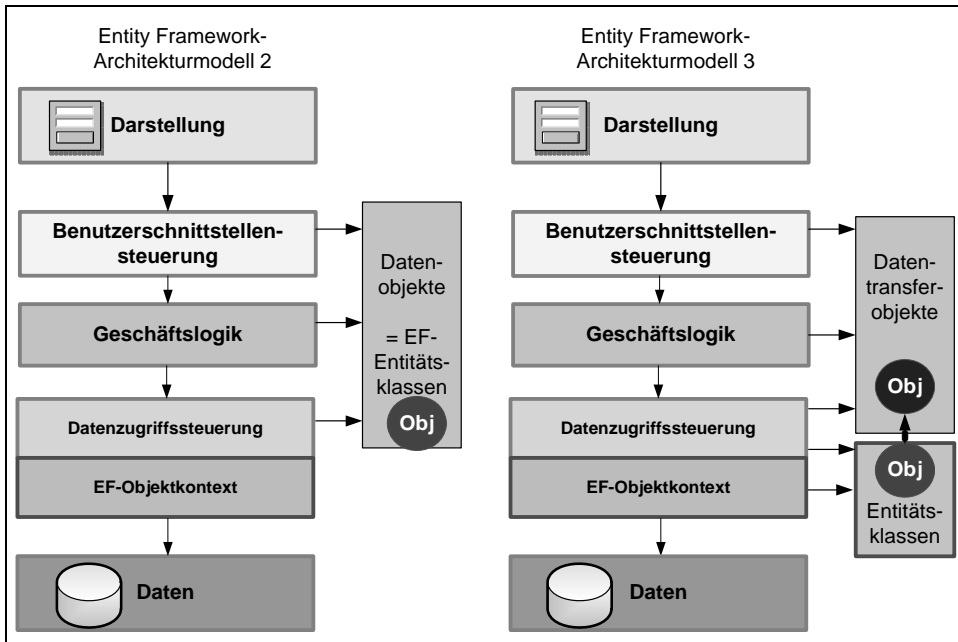


Abbildung 12.7 Entity Framework-Architekturmodell 2 und 3

**TIPP**

Welches Modell ist das beste? Das kann man ganz allgemein beantworten: Es kommt auf Ihre Anforderungen & Entwicklungsphilosophie an! Der Autor dieses Buchs verwendet in verschiedenen Projekten alle drei Modelle. Das World Wide Wings-Beispiel in diesem Buch verwendet ab Version 0.6 das Modell 1. In Version 0.5 kam noch Modell 2 zum Einsatz, das wurde aber von vielen Lesern dieses Buchs als zu komplex empfunden.

## Erstellen eines EDM-Modells

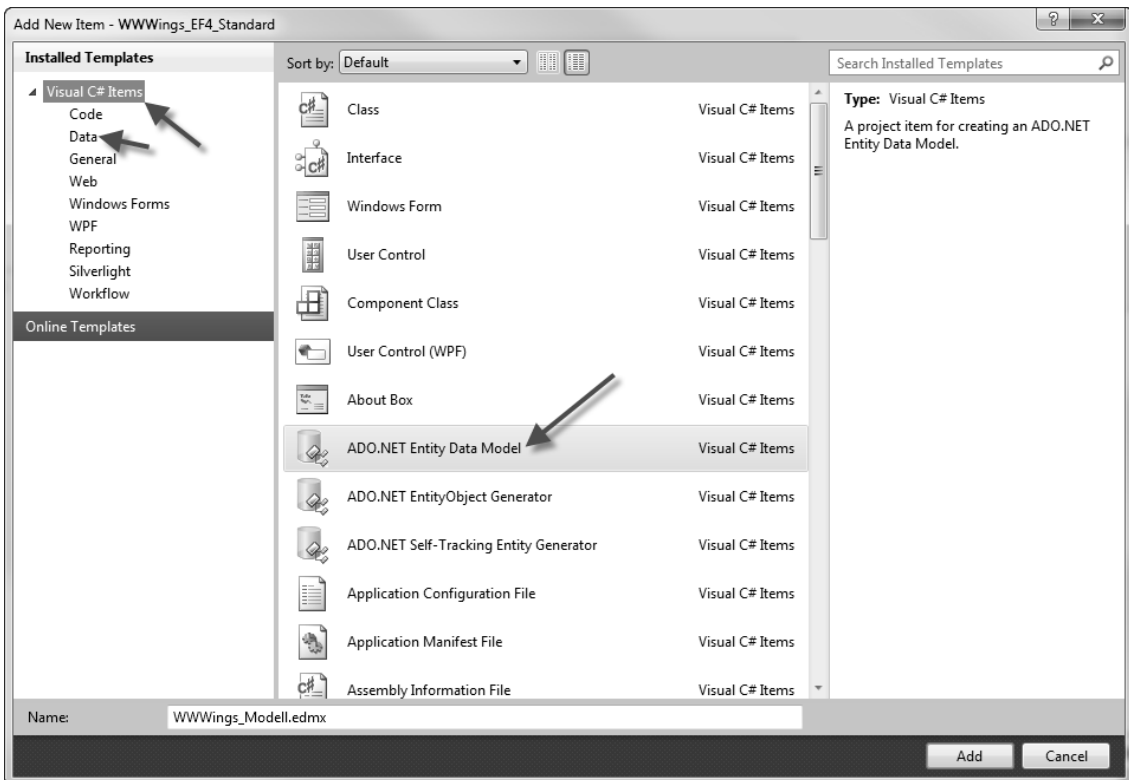
Dieser Abschnitt beschreibt das Anlegen eines EDM-Modells mithilfe der Entity Framework-Werkzeuge in Visual Studio 2010. Diese Werkzeuge bieten einige wesentliche Verbesserungen gegenüber dem Vorgänger in Visual Studio 2008 SP1, sind aber bei Weitem noch nicht so komplett, wie man es sich wünschen würde.

## Elementvorlage

Der Entwickler legt ein Modell über die Elementvorlage *ADO.NET Entity Data Model* an. Dadurch startet ein Assistent (*Entity Data Model Wizard*), mit dem man die Datenbank und die Tabelle auswählen kann. Der Assistent erstellt dann eine Designer-Ansicht (*Entity Data Model Designer*), wobei Entitätstypen für Tabellen entstehen. Anders als bei LINQ to SQL wird das Erstellen eines Modells per Drag & Drop aus dem Server Explorer im Entity Framework-Designer nicht unterstützt.

**HINWEIS** In der Sektion *Data* erscheint nur diese Vorlage zum Suchbegriff *ADO.NET Entity Framework*, wenn man aber alle Elemente betrachtet (siehe nachstehende Abbildung), findet man hier auch weitere Einträge mit *ADO.NET* und *Entity* im Namen. Dies sind Vorlagen zur Steuerung der Codegenerierung des Designers, die man in der Regel erst einbindet, wenn man bereits ein Modell erstellt hat.

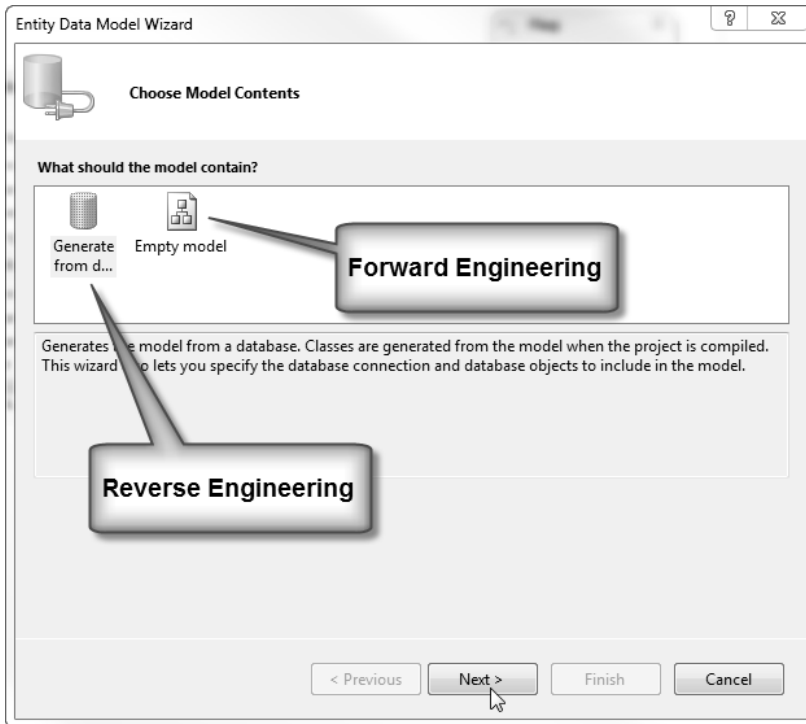
**TIPP** Grundsätzlich kann man ein Entity Framework-Modell in fast jedem Projekttyp anlegen (ausgenommen sind zum Beispiel Silverlight-Projekte!). Darüber hinaus kann man ein Entity Framework-Modell direkt in einer ausführbaren Datei (z. B. Konsolenanwendung, Windows Forms-Anwendung, WPF-Anwendung) oder auch einem Webprojekt anlegen. Es bietet sich jedoch an, das Entity Framework-Modell im Sinne der Schichtentrennung und in Hinblick auf Wiederverwendbarkeit und Wartbarkeit, in einem getrennten Klassenbibliotheksprojekt anzulegen.



**Abbildung 12.8** Anlegen eines ADO.NET Entity Framework Modells

## Forward Engineering oder Reverse Engineering

Der Designer fragt im ersten Schritt nach der gewünschten Vorgehensweise. Entweder es wird eine Datenbank als Vorlage genommen (Database First alias Reverse Engineering) oder man startet mit einem Modell (Modell First alias Domain First alias Forward Engineering). Während in Visual Studio 2008 SP1 die zweite Option fast ohne Wert war, kann man nun tatsächlich auf Basis eines Modells eine Datenbank erzeugen lassen. Dies wird im Abschnitt »Forward Engineering« besprochen.

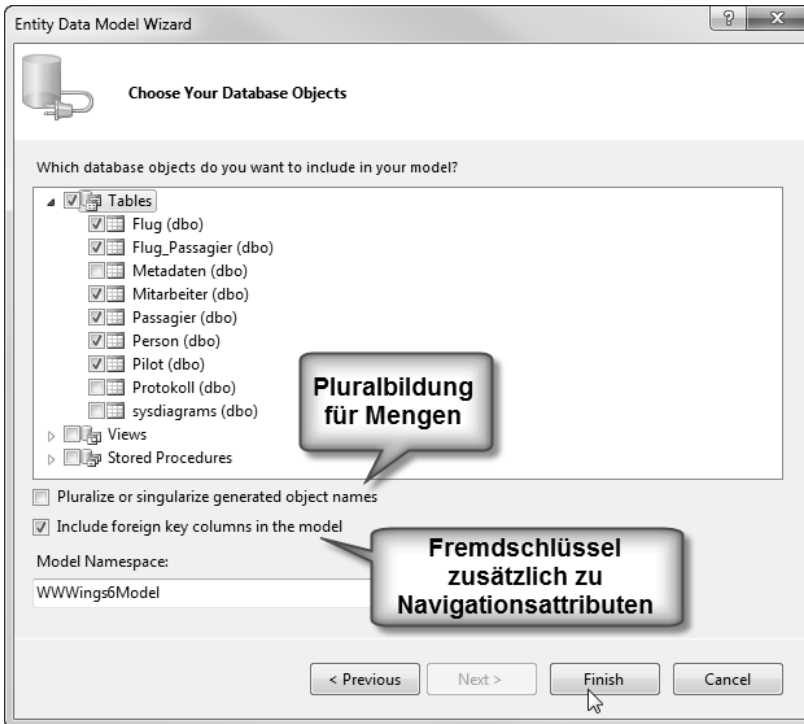


**Abbildung 12.9** Wahl zwischen Reverse und Forward Engineering

Bei Reverse Engineering folgt nun die Auswahl der Datenbankverbindung. Hieran hat sich gegenüber Visual Studio 2008 SP1 nichts geändert.

## Auswahl der Datenbankelemente, Pluralisierung und Fremdschlüssel

Im dritten Schritt, der Auswahl der Tabelle, Sichten und gespeicherten Prozeduren gibt es nun gegenüber Visual Studio 2008 SP1 zwei neue Optionen: Zum einen kann man Einfluss darauf nehmen, wie der Name von Objektmengen (Entity Sets) gebildet wird (siehe unten) und zum anderen kann man auch Fremdschlüssel zusätzlich zu den Navigationsproperties erzeugen lassen (siehe Abschnitt »Objektzuweisungen«).



**Abbildung 12.10** Zwei neue Optionen im Assistenten

Der Assistent zum Anlegen des Modells aus einer Datenbank bietet die Optionen, die Entitätsmengenklassen in den Plural zu setzen. Standard ist die Pluralbildung mit dem Anhängen eines »s«.

**HINWEIS** Der Assistent zeigt im Tooltip an, dass die Pluralisierung nur für die englische Sprache existiert. Stolz ist Microsoft, dass man einige Ausnahmen definiert hat. So wird aus einer Entität »Mouse« die Menge »Mice« und aus »Person« wird »People«. Dabei hätte Microsoft diese Zeit besser investiert, die Pluralisierung anpassbar zu machen. Ein Entwickler hätte sich sicherlich gerne frei für ein Suffix, z.B. »Set« oder »Menge« entscheiden wollen. Grundsätzlich kann man einen eigenen Entity Framework-Pluralisationsdienst schreiben, indem man von der abstrakten Basisklasse `System.Data.Entity.Design.PluralizationServices.PluralizationService` ableitet; diese eigene Implementierung ist aber leider nicht in Visual Studio einbindbar. Man kann dann den Pluralisierungsdienst nur für die codebasierte Generierung mit der Klasse `EntityModelSchemaGenerator` nutzen, indem man den selbst erstellten `PluralizationService` als zweiten Parameter im Konstruktor übergibt.

## Entity Framework-Designer

Danach öffnet sich der Designer. Der Designer zeigt auf den ersten Blick, dass durch den Assistenten reine N:M-Zwischentabellen in \*-\*-Beziehungen aufgelöst wurden. Leider bietet der Assistent keine Möglichkeit, diese Auflösung zu deaktivieren. Auch bietet der Assistent nicht an, die Beziehung zwischen Person, Passagier, Mitarbeiter und Pilot als Vererbung zu modellieren.

Der Entity Framework-Designer besteht aus vier Fenstern:

- *Designer-Oberfläche* mit der grafischen Darstellung der Entitäten (nahtloses Zoomen möglich)
- *Eigenschaftenfenster* für Festlegungen der Eigenschaften der Entitäten und ihrer Attribute
- *Modell-Browser* mit einer hierarchischen Darstellung von SSDL und CSDL
- *Mapping-Details* mit einer Darstellung der Abbildung zwischen SSDL und CSDL

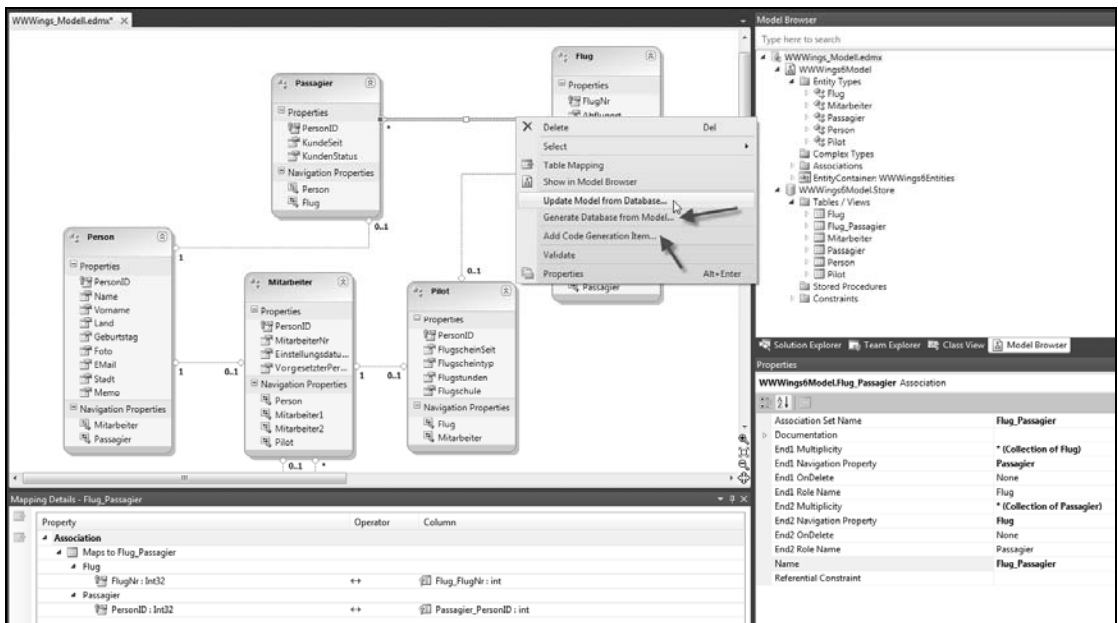


Abbildung 12.11 Der Entity Framework-Designer in Visual Studio 2010

Im Kontextmenü der Designer-Oberfläche findet man einige Aktionen, von denen nur einige hier exemplarisch erklärt werden können:

- *Update Modell from Database* zum Aktualisieren des Modells aus der Datenbank (siehe Tipp unten!)
- *Generate Database from Model* erzeugt eine Datenbank aus dem Modell (siehe Abschnitt »Forward Engineering (Model-First/Domain First)«)
- *Add Code Generation Item* erlaubt den Austausch der Code-Generierung (siehe Abschnitt »Steuerung der Codegenerierung durch austauschbare T4-Vorlagen«)



**HINWEIS** In Visual Studio 2010 behoben ist ein sehr lästiger Bug aus dem EF-Designer in Visual Studio 2008: Nach dem Löschen einer Entität aus dem Modell konnte der Entwickler später diese Entität nicht wieder in das Modell aufnehmen, weil Visual Studio zwar im Conceptual Model und im Mapping die Entität löscht, nicht aber die Beschreibung der Tabelle im Store Model. Dies musste der Entwickler händisch in XML erledigen. Visual Studio 2010 ist hier netter und fragt den Entwickler beim Löschen, ob das Store Model auch bereinigt werden soll.

**TIPP** Der Assistent *Update Modell from Database* ergänzte schon in VS 2008 SP1 problemlos Entitäten und neue Attribute im Modell, wenn es neue Tabellen bzw. neue Spalten für bestehende Modelle gab. Der Assistent aktualisiert in anderen Fällen (z.B. Spalte gelöscht oder anderer Datentyp) weiterhin nur das Store Model und man muss das Conceptual Model selbst anpassen.

Wenn man im Modell nichts händisch umbenannt oder umkonfiguriert hat, kann es sich beim Reverse Engineering lohnen, das ganze Modell zu löschen und neu anzulegen. Dabei löscht man entweder alle Entitäten und verwendet dann *Update Modell from Database*, um diese neu aufzunehmen. Oder man löscht die ganze *.edmx*-Datei. Dann muss man aber auch die zugehörige Verbindungszeichenfolge aus der Konfigurationsdatei (*app.config/web.config*) in dem gleichen Projekt entfernen, bevor man den Assistenten neu startet. Sonst wird der Assistent die vorhandene Verbindungszeichenfolge bemerken und für das neue Modell eine Verbindungszeichenfolge mit neuem Namen anlegen. Dann muss man aber diese neue Verbindungszeichenfolge in alle Anwendungsprojekte kopieren, die Zugriff auf das Modell haben. Wenn Sie die Verbindungszeichenfolge vorher entfernen (oder auskommentieren), bleibt der Name gleich und Sie haben weniger Arbeit!

In Summe empfiehlt der Autor dieses Buchs auf Basis seiner Projekterfahrung mit dem Entity Framework: Wenn irgendwie möglich, erstellen Sie das Datenbankmodell so, dass Sie mit dem vom Assistenten generierten Modell halbwegs zufrieden sein können. Wenn irgendwie möglich, nehmen Sie keinen händischen Änderungen vor.

**ACHTUNG** Bitte beachten Sie, dass der EF-Designer nicht das Ziehen & Fallenlassen von Elementen aus den Datenverbindungen (im Server Explorer) wie im DataSet- oder LINQ to SQL-Designer unterstützt. Elemente aus einer Datenbank hinzufügen kann man nur durch erneutes Aufrufen des Assistenten (Funktion *Update Model from Database* im Kontextmenü).

**HINWEIS** Wenn man als Target Framework *.NET Framework 3.5* wählt, bekommt man in Visual Studio 2010 ebenfalls den verbesserten Assistenten auch für ältere Projekte. Allerdings sind nicht alle neuen Funktionen verfügbar. Das Anlegen von Fremdschlüsselattributen kann man im Assistenten nicht aktivieren, wohl aber die Namenspluralisierung. Das Generieren einer Datenbank aus einem Modell funktioniert. *Add Code Generation Item* ist verfügbar. Die verfügbaren Vorlagen funktionieren aber nicht: *Self-Tracking Entities* wird gar nicht erst angezeigt; die Online-Vorlage *POCO* wird angezeigt und ausgeführt, der erzeugte Programmcode ist aber unter *.NET Framework 3.5 Service Pack 1* nicht kompilierbar.

**TIPP** Einen alternativen Designer mit einigen nützlichen Zusatzfunktionen (z.B. Einfluss auf Namensgebung, Drag & Drop von Tabellen auf Designeroberfläche, Datenvorschau und Dateneingabegitter) bietet der Entity Developer der Firma DevArt (kostenpflichtig, siehe <http://www.devart.com/entitydeveloper>).

## Generierte Klassen

Wenn man eine *.edmx-Datei* anlegt (z.B. *Modell.edmx*), dann entsteht dabei auch eine zweite Datei mit Programmcode, die *Modell.edmx.Designer.cs*- bzw. *Modell.edmx.Designer.vb*-Datei. Diese Datei sieht man nur, wenn man im Projektmappen-Explorer *Alle Dateien anzeigen* wählt. Die »Designer«-Datei enthält automatisch generierte Klassen.

- Für jede Entität wird eine Klasse erzeugt (*Entitätsklasse*)
- Zusätzlich entsteht eine so genannte *Kontextklasse* (alias *Objektkontext*), die die Datenbank als Ganzes darstellt und Einstiegspunkt für alle Datenbankoperationen ist

In dem vorliegenden Beispiel gibt es also sechs Klassen (fünf Entitätsklassen und eine Kontextklasse). Auch diesen Programmcode kann man betrachten, man sollte ihn aber nicht ändern, da bei jeglicher Änderung am Modell der komplette Programmcode neu generiert wird und damit alle Änderungen verworfen werden.

### Kontextklasse (Objektkontext)

Die Kontextklasse hat folgende Aufgaben:

- Verwalten der ADO.NET-Datenbankverbindung (im Standard automatisches Öffnen und Schließen bei Bedarf)
- Einstiegspunkt für Abfragen (LINQ to Entities, eSQL, SQL). Dafür bietet die Kontextklasse für jede Entitätsklasse eine Menge vom Typ `ObjectSet<T>` an. In Entity Framework war dies `ObjectQuery<T>`. `Object<T>` bietet mehr Möglichkeiten.
- Der Kontext ist Zwischenspeicher (Cache) für Objekte
- Der Kontext sorgt für die Änderungsverfolgung (Change Tracking) der geladenen Objekte

---

#### HINWEIS      Bitte beachten Sie folgende Hinweise:

- Nach der Verwendung des Datenkontextes sollte man diesen mit `Dispose()` explizit vernichten. Es bietet sich an, einen `using`-Block zu verwenden.
  - Bei der ersten Instanziierung innerhalb eines Prozesses erzeugt die Kontextklasse den Mapping-Code. Dies kann bei umfangreicheren Modellen mehrere Sekunden dauern. Die Folgeinstanziierung innerhalb des gleichen Prozesses ist aber fast ohne Zeitaufwand (weniger als eine Millisekunde) möglich.
  - Wenn die Erstinstanziierungszeit des Kontextes zu lange dauert, hilft nur die Partitionierung des Kontextes, d.h. die Aufteilung des großen Modells auf mehrere kleine Modelle. Damit kann aber verbunden sein, dass einzelne Entitäten doppelt existieren. Dabei muss man im Hinterkopf haben, dass man Entitäten eines Kontextes nicht an einen anderen übergeben kann.
  - Der Kontext ist nicht »thread-safe«. Auf keinen Fall dürfen mehrere Threads den gleichen Kontext benutzen.
  - Der Kontext merkt sich alle Änderungen seit dem Laden oder dem letzten Speichern. Beim Speichern werden alle diese Änderungen gespeichert. Daher dürfen auf keinen Fall in einer Anwendung mehrere Benutzer die gleiche Instanz des Kontextes verwenden.
  - Sie können Objekte nicht ohne Weiteres zwischen Kontexten austauschen. Die Anweisung `flug.Pilot = meier` schlägt mit einer `InvalidOperationException` fehl, wenn `flug` und `meier` durch zwei verschiedene Kontexte geladen wurden. Für einen Kontextwechsel muss man entweder die Zuweisung durch Fremdschlüssel machen (geht aber nur bei 1:1-/1:N-, nicht N:M-Beziehungen), oder durch Neuladen des Objekts in den zweiten Kontext oder durch Loslösen des Objekts aus einem Kontext (`Detach()`-Methode) und Anfügen an den anderen Kontext (`Attach()`-Methode).
-

Die Kontextklasse leitet immer von `System.Data.Objects.ObjectContext` ab. Die Kontextklasse erhält kurioserweise den Namen, den man im Entity Framework-Assistenten für die Verbindungszeichenfolge vergeben hat. Man kann den Namen der Kontextklasse aber ändern, indem man in den Eigenschaften des Designers (Klick auf den leeren Hintergrund!) den *Entity Container Name* auswechselt.

Die Kontextklasse besitzt mehrere Konstruktoren, bei denen man entweder den Namen einer Verbindungszeichenfolge aus der Konfigurationsdatei übergeben kann oder eine Instanz der Klasse `EntityConnection`. Diese Verbindung kann wahlweise schon geöffnet sein oder vom Kontext bei Bedarf geöffnet werden. Wenn man dem Konstruktor gar nichts übergibt, verwendet der Kontext die Verbindungszeichenfolge aus der Konfigurationsdatei, die beim Anlegen des Kontextes dort abgelegt wurde.

```
public partial class WWWings6Entities : ObjectContext
{
    #region Constructors

    /// <summary>
    /// Initializes a new WWWings6Entities object using the connection string found in the 'WWWings6
Entities' section of the application configuration file.
    /// </summary>
    public WWWings6Entities() : base("name=WWWings6Entities", "WWWings6Entities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }

    /// <summary>
    /// Initialize a new WWWings6Entities object.
    /// </summary>
    public WWWings6Entities(string connectionString) : base(connectionString, "WWWings6Entities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }

    /// <summary>
    /// Initialize a new WWWings6Entities object.
    /// </summary>
    public WWWings6Entities(EntityConnection connection) : base(connection, "WWWings6Entities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }

    ...

    /// <summary>
    /// No Metadata Documentation available.
    /// </summary>
    public ObjectSet<Flug> Flug
    {
        get
        {
            if ((_Flug == null))
            {
                _Flug = base.CreateObjectSet<Flug>("Flug");
            }
        }
    }
}
```

```

        return _Flug;
    }
}
private ObjectSet<Flug> _Flug;
...

```

**Listing 12.1** Ausschnitt aus einer generierten Kontextklasse

## Entitätsklassen

Im Standard sind die generierten Entitätsklassen von `System.Data.Objects.DataClasses.EntityObject` abgeleitet. Es gibt aber alternative Codegenerierungsvorlagen (vgl. Abschnitt »Persistence Ignorance mit Plain Old CLR Objects (POCO)«), die diese Basisklasse nicht erfordern. Dies ist eine in .NET 4.0 neu eingeführte Option.

Die Geschäftsobjektclassen besitzen in der Standardeinstellung sowohl eine Auszeichnung mit `[Serializable]` als auch `[DataContract]` und `[DataMember]`. Durch den Zusatz `IsReference = true` werden Objektassoziationen angezeigt, was Grundlage dafür ist, dass das EF ganze Objektbäume serialisieren kann (Graph Serialization).

```

[global::System.Data.Objects.DataClasses.EdmEntityTypeAttribute(NamespaceName = "WWWingsModel", Name =
"FL_Fluege")]
[global::System.Runtime.Serialization.DataContractAttribute(IsReference = true)]
[global::System.Serializable()]
public partial class FL_Fluege : global::System.Data.Objects.DataClasses.EntityObject
{
    ...
}

```

**Listing 12.2** Kopf der generierten Geschäftsobjektclass

1:N-Assoziationen sind Instanzen der generischen Klasse `System.Data.Objects.DataClasses.EntityCollection`. 1:1-Assoziationen werden durch ein Paar von zwei Attributen dargestellt: Einerseits gibt es ein einfaches Attribut (vom Typ der assoziierten Klasse) und andererseits ein Referenzattribut (vom generischen Typ `System.Data.Objects.DataClasses.EntityReference`). Da auch die Assoziationen als `[DataMember]` gekennzeichnet sind, ist im Gegensatz zu LINQ to SQL eine Serialisierung kompletter Objektbäume möglich.

```

[global::System.Data.Objects.DataClasses.EdmRelationshipNavigationPropertyAttribute
("WWWingsModel", "FK_GF_GebuchteFluege_FL_Fluege", "FL_Fluege")]
[global::System.Xml.Serialization.XmlIgnoreAttribute()]
[global::System.Xml.Serialization.SoapIgnoreAttribute()]
[global::System.Runtime.Serialization.DataMemberAttribute()]
[global::System.ComponentModel.BrowsableAttribute(false)]
public FL_Fluege FL_Fluege
{
    get
    {
        return ((global::System.Data.Objects.DataClasses.IEntityWithRelationships)(this))
.RelationshipManager.GetRelatedReference<FL_Fluege>("WWWingsModel.FK_GF_GebuchteFluege_FL_Fluege",
"FL_Fluege").Value;
    }
    set

```

```

{
    ((global::System.Data.Objects.DataClasses.IEntityWithRelationships)
        (this)).RelationshipManager.GetRelatedReference<FL_Fluege>(
        "WWWingsModel.FK_GF_GebuchteFluege_FL_Fluege", "FL_Fluege").Value = value;
}
}

```

**Listing 12.3** Beispiel für eine 1:N-Assoziation im EF

```

[global::System.Data.Objects.DataClasses.
    EdmRelationshipNavigationPropertyAttribute("WWWingsModel",
    "FK_PS_Passagier_PE_Person", "PE_Person")]
[global::System.Xml.Serialization.XmlIgnoreAttribute()]
[global::System.Xml.Serialization.SoapIgnoreAttribute()]
[global::System.Runtime.Serialization.DataMemberAttribute()]
[global::System.ComponentModel.BrowsableAttribute(false)]
public PE_Person PE_Person
{
    get
    {
        return ((global::System.Data.Objects.DataClasses.IEntityWithRelationships)
            (this)).RelationshipManager.GetRelatedReference<PE_Person>(
            "WWWingsModel.FK_PS_Passagier_PE_Person", "PE_Person").Value;
    }
    set
    {
        ((global::System.Data.Objects.DataClasses.IEntityWithRelationships)(this))
            .RelationshipManager.GetRelatedReference<PE_Person>(
            "WWWingsModel.FK_PS_Passagier_PE_Person", "PE_Person").Value = value;
    }
}

[global::System.Runtime.Serialization.DataMemberAttribute()]
[global::System.ComponentModel.BrowsableAttribute(false)]
public global::System.Data.Objects.DataClasses.EntityReference<PE_Person> PE_PersonReference
{
    get
    {
        return ((global::System.Data.Objects.DataClasses.IEntityWithRelationships)(this))
            .RelationshipManager.GetRelatedReference<PE_Person>(
            "WWWingsModel.FK_PS_Passagier_PE_Person", "PE_Person");
    }
    set
    {
        if ((value != null))
        {
            ((global::System.Data.Objects.DataClasses.IEntityWithRelationships)(this))
                .RelationshipManager.InitializeRelatedReference<PE_Person>(
                "WWWingsModel.FK_PS_Passagier_PE_Person", "PE_Person", value);
        }
    }
}

```

**Listing 12.4** Beispiel für eine 1:1-Assoziation im EF

## Umsetzung der Datentypen

Die folgende Tabelle zeigt, wie die Microsoft SQL Server-Datentypen in .NET-Datentypen umgesetzt werden. Entsprechende Tabellen für andere Datenbanksysteme erhalten Sie von den Anbietern der jeweiligen Entity Framework-Treiber.

SQL Server-Datentyp	EF-Datentyp / .NET-Datentyp (sofern abweichend von EF-Datentyp)
bigint	Int64
binary	Binary / Byte[]
bit	Boolean
char	String
date (2008)	DateTime
datetime	DateTime
datetime2 (2008)	DateTime
datetimeoffset	DateTimeOffset
decimal	Decimal
float	Double
geography (2008)	<i>Nicht ohne Tricks möglich</i>
geometry (2008)	<i>Nicht ohne Tricks möglich</i>
hierarchyid (2008)	<i>Nicht ohne Tricks möglich</i>
image	Binary / Byte[]
int	Int32
money	Decimal
nchar	String
ntext	String
numeric	Decimal
nvarchar	String
nvarcharmax	String
real	Single
smalldatetime	DateTime
smallint	Int16
smallmoney	Decimal
sqlvariant (2008)	<i>Nicht ohne Tricks möglich</i>
text	String
time (2008)	Time / Nullable<Timespan>
timestamp/rowversion	Binary / Byte[]
tinyint	Byte 

SQL Server-Datentyp	EF-Datentyp / .NET-Datentyp (sofern abweichend von EF-Datentyp)
uniqueidentifier	Guid
varbinary	Binary / Byte[]
varbinarymax	Binary / Byte[]
varchar	String
varcharmax	String
xml	String

**Tabelle 12.1** Microsoft SQL Server-Datentypen und .NET-Datentypen

**TIPP** Zu den Einträgen »Nicht ohne Tricks möglich« siehe [FOLLWAS01].

## Entity Framework-Verbindungszeichenfolgen

Eine Verbindungszeichenfolge für das Entity Framework ist etwas komplexer als eine normale ADO.NET-Verbindungszeichenfolge.

Daher besteht die Verbindungszeichenfolge aus drei Sektionen:

- Festlegung der EDM-Beschreibung (CSDL, SSDL und MSL), siehe Metadata= in dem folgenden Beispiel (die normalerweise als Ressourcen in die Assembly hineinkompiliert werden, dafür die Angabe »res:« und dann eingebettet die normale ADO.NET-Verbindungszeichenfolge)
- Festlegung des darunterliegenden ADO.NET-Datenbanktreibers, siehe Provider= in dem folgenden Beispiel
- Festlegung der Datenbank mit einer klassischen Verbindungszeichenfolge, siehe Connection String= in dem folgenden Beispiel

Wie man in dem Beispiel sieht, enthält also eine Verbindungszeichenfolge für den Entity Client eine klassische Verbindungszeichenfolge. " steht dabei für das Ausführungszeichen für die innere Verbindungszeichenfolge innerhalb der Verbindungszeichenfolge, die ja schon in Anführungszeichen steht.

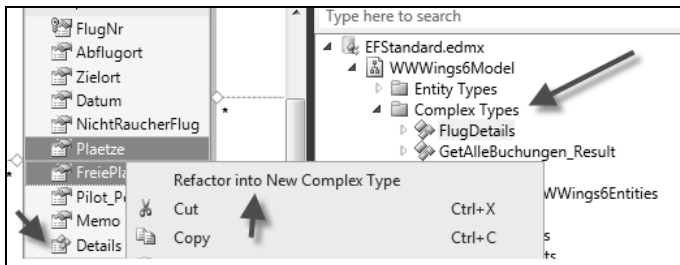
```
<connectionStrings>
<add name="WWWings6Entities" connectionString=
"metadata=res:/*/*EF.EF_Model1.csd1|res:/*/*EF.EF_Model1.ssd1|res:/*/*EF.EF_Model1.msl;
provider=System.Data.SqlClient;provider connection string="
Data Source=server\sqlexpress;Initial Catalog=WWWings6;
Integrated Security=True;MultipleActiveResultSets=True";
providerName="System.Data.EntityClient" />
</connectionStrings>
```

**HINWEIS** Wenn Sie – wie in diesem Buch empfohlen – das Modell in einer getrennten Klassenbibliothek anlegen, dann dürfen Sie nicht vergessen, die Verbindungszeichenfolge in die Konfigurationsdatei des jeweiligen Startprojekts zu kopieren, denn .NET ignoriert beim Start die Konfigurationsdateien zu einer DLL. Leider legt Visual Studio immer noch alle automatisch generierten Konfigurationsdaten nur in das Projekt, in dem sie erzeugt wurden und nicht automatisch auch in das Startprojekt, wo sie benötigt werden.

## Unterstützung für komplexe Typen im Designer

So genannte *komplexe Typen* (Complex Types) gab es auch schon in der ersten Version des Entity Framework. Komplexe Typen sind Klassen, die keinen eigenen Entitätstyp darstellen, sondern aus der Sicht von .NET die Entitätsobjekte in Unterobjekte gliedern. Komplexe Typen erlauben, eine Menge von Spalten aus einer Tabelle in ein Unterobjekt des Entitätsobjekts auszulagern. Komplexe Typen sind eigene Klassen, aber ohne eigenes Schlüsselattribut. Am besten ist dies an einem Beispiel erklärt: Die Tabelle Mitarbeiter mit MitarbeiterID, VorgesetzterID, Strasse, Postleitzahl und Ort kann im EF-Modell leicht in zwei Klassen *Mitarbeiter* und *MitarbeiterAdresse* aufgespalten werden, wobei dann jede Instanz von Mitarbeiter eine Instanz von *MitarbeiterAdresse* besitzt.

Komplexe Typen konnte man noch in Visual Studio 2008 nur erstellen, indem man das XML-Mapping (.edmx-Datei bzw. .csdl, .ssdl und .msl-Datei) manuell bearbeitet hat. Komplexe Typen kann man in Visual Studio 2010 einfach erstellen, indem man ein oder mehrere Attribute in einer Entitätsklasse markiert und dann *Create Complex Type* im Kontextmenü wählt.



**Abbildung 12.12** Extrahieren eines komplexen Typs aus einer Entitätsklasse

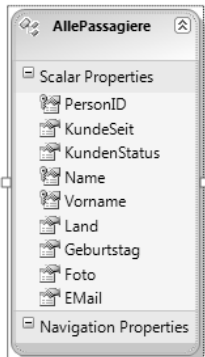
**ACHTUNG** Es gibt Schwierigkeiten, wenn Sie Entitäts-Klassen, die komplexe Typen enthalten, durch eine Stored Procedure laden wollen. Wenn Sie Stored Procedures für das Laden von Entitäten verwenden wollen, sollten Sie auf komplexe Typen verzichten.

Aber: Komplexe Typen kommen automatisch zum Einsatz, wenn eine Stored Procedure weder einen primitiven Typ noch einen Entitätstyp liefert, siehe Abschnitt über die verbesserte Stored Procedure-Unterstützung.

## Verwendung von Views

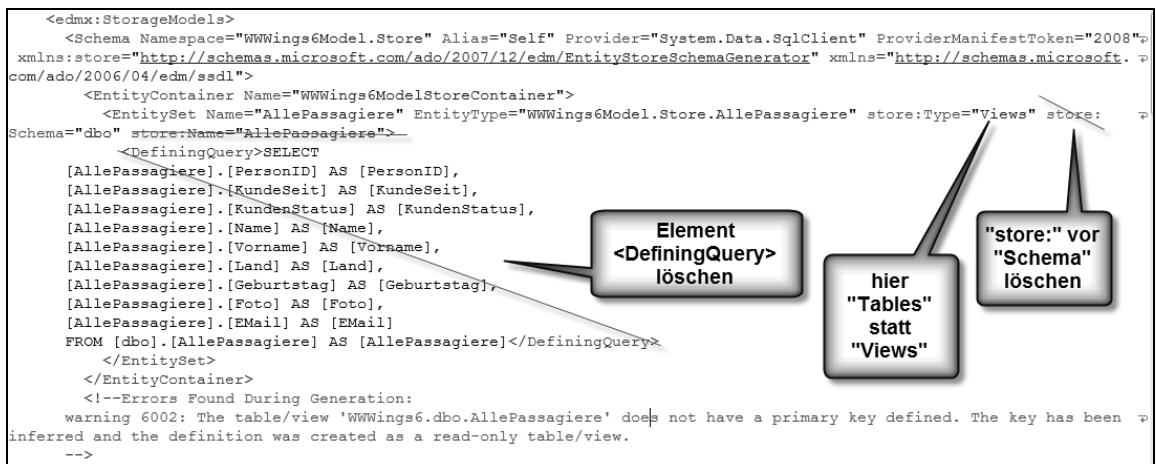
Entity Framework unterstützt Datenbank-Sichten (Views), aber die Entity Framework-Werkzeuge in Visual Studio stehen damit leider auf dem Kriegsfuß. Nimmt man mithilfe des Assistenten eine View in das Modell auf, hat dieser plötzlich sehr viele Primärschlüssel, wie man in Abbildung 12.13 anhand des Views *AllePassagiere* aus der Beispieldatenbank *World Wide Wings* sieht. Alle Spalten, die nicht *null* sind, werden zum Primärschlüssel. Microsoft lieferte dazu unter [MSFOR02] eine wenig ermutigende Aussage: »We have spent a tremendous amount of energy trying to solve this, but it doesn't look like we'll be able to do a lot for this in the next release.« Leider ist hier als nächste Version schon Visual Studio 2010 gemeint.





**Abbildung 12.13** Ein View mit unerklärlich vielen Primärschlüsseln im EF-Designer in Visual Studio

Dennoch kann man sich behelfen, wenn man bereit ist, die von dem Assistenten erzeugte EDMX-Datei in roher XML-Form nachzubearbeiten. Die drei folgenden Abbildungen zeigen genau, was in den Sektionen *<StorageModels>* und *<ConceptualModels>* in der *.edmx*-Datei zu ändern ist, um *AllePassagiere* von dem Entity Framework genauso wie eine Tabelle behandeln zu lassen. In der Folge dieser Änderungen könnte man dann sogar schreibend auf den View zugreifen, sofern die Sicht auch wirklich aktualisierbar ist.



**Abbildung 12.14** Änderungen an der SSDL-Sektion in der EDMX-Datei (Teil 1)

```

<EntityType Name="AllePassagiere">
  <Key>
    <PropertyRef Name="PersonID" />
    <PropertyRef Name="Name" />
    <PropertyRef Name="Vorname" />
  </Key>
  <Property Name="PersonID" Type="int" Nullable="false" />
  <Property Name="KundeSeit" Type="datetime" />
  <Property Name="KundenStatus" Type="nchar" MaxLength="1" />
  <Property Name="Name" Type="nvarchar" Nullable="false" MaxLength="50" />
  <Property Name="Vorname" Type="nvarchar" Nullable="false" MaxLength="50" />
  <Property Name="Land" Type="nvarchar" MaxLength="2" />
  <Property Name="Geburstag" Type="datetime" />
  <Property Name="Foto" Type="varbinary(max)" />
  <Property Name="EMail" Type="nvarchar" MaxLength="50" />
</EntityType>
</Schema>
</edmx:StorageModels>

```

Zwei Zeilen löschen

Abbildung 12.15 Änderungen an der SSDL-Sektion in der EDMX-Datei (Teil 2)

```

<edmx:ConceptualModels>
  <Schema Namespace="WWWings6Model" Alias="Self" xmlns="http://schemas.microsoft.com/ado/2006/04/edm">
    <EntityContainer Name="CS_WWWings">
      <EntitySet Name="AllePassagiere" EntityType="WWWings6Model.AllePassagiere" />
    </EntityContainer>
    <EntityType Name="AllePassagiere">
      <Key>
        <PropertyRef Name="PersonID" />
        <PropertyRef Name="Name" />
        <PropertyRef Name="Vorname" />
      </Key>
      <Property Name="PersonID" Type="Int32" Nullable="false" />
      <Property Name="KundeSeit" Type="DateTime" />
      <Property Name="KundenStatus" Type="String" MaxLength="1" Unicode="true" FixedLength="true" />
      <Property Name="Name" Type="String" Nullable="false" MaxLength="50" Unicode="true" FixedLength="false" />
      <Property Name="Vorname" Type="String" Nullable="false" MaxLength="50" Unicode="true" FixedLength="false" />
      <Property Name="Land" Type="String" MaxLength="2" Unicode="true" FixedLength="false" />
      <Property Name="Geburstag" Type="DateTime" />
      <Property Name="Foto" Type="Binary" MaxLength="Max" FixedLength="false" />
      <Property Name="EMail" Type="String" MaxLength="50" Unicode="true" FixedLength="false" />
    </EntityType>
  </Schema>
</edmx:ConceptualModels>

```

Zwei Zeilen löschen

Abbildung 12.16 Änderungen an der CSDL-Sektion in der EDMX-Datei

## Daten abfragen

Dieser Abschnitt dokumentiert Möglichkeiten zum Abfragen von Datenbanken mit dem ADO.NET Entity Framework.

## Instanzieren der Kontextklasse

Vor der ersten Aktion mit den EF Object Services benötigt man eine Instanz des Objektkontextes, der die gleiche Rolle spielt wie der Datenkontext bei LINQ to SQL. Eine Instanz erzeugt man auf folgende Weise:

- Instanzieren der Basisklasse `ObjectContext` unter Angabe des Namens einer EF-Verbindungszeichenfolge (aus der Konfigurationsdatei) oder eines geöffneten `EntityConnection`-Objekts, z.B. `ObjectContext db = new ObjectContext(CS).`

- Instanziiieren einer typisierten Kontextklasse, die durch den EDM-Designer erstellt wurde, z.B. `WorldWideWingsModel.WorldWideWings db = new WorldWideWingsModel.WorldWideWings()`. Hier kann man als Parameter wahlweise einen Namen einer EF-Verbindungszeichenfolge aus der Konfigurationsdatei, ein `EntityConnection`-Objekt oder gar nichts angeben. Im letzten Fall wird die in der Konfigurationsdatei hinterlegte Verbindungszeichenfolge verwendet, die beim Erstellen des Modells dort angelegt wurde.

**HINWEIS** Als Zeichenkette ist tatsächlich nicht die Verbindungszeichenfolge selbst, sondern der Name einer Verbindungszeichenfolge anzugeben, der in der Konfigurationsdatei hinterlegt ist, z.B.

```
WorldWideWings db = new WorldWideWings("Name=WWings6Entities");
```

Möchte man eine Verbindungszeichenfolge komplett im Code angeben, muss man vorher eine Instanz der Klasse `EntityConnection` erzeugen:

```
string ConnString =  
    (@"metadata=res:/*/*WWings6_MF.csd1|res:/*/*WWings6_MF.ssd1|res:/*/*WWings6_MF.msl';  
    provider=System.Data.SqlClient;provider connection string='Data Source=.;  
    Initial Catalog=WWings6Entities;Integrated Security=True;  
    MultipleActiveResultSets=True'");  
Console.WriteLine(ConnString);  
EntityConnection econn = new EntityConnection(ConnString);  
WorldWideWings db = new WorldWideWings(econn);
```

Über `EntityConnectionStringBuilder` kann man eine EF-Verbindungszeichenfolge auch aus Einzelteilen zusammenbauen.

## Abfragesprachen

Das Entity Framework unterstützt vier verschiedene Abfragesprachen:

- **Entity SQL (eSQL)** eine zeichenkettenbasierte Sprache, die SQL ähnlich ist (die einzige Syntaxform, in der »alles« geht, was EF kann)
- **LINQ to Entities** LINQ-Variante für Entity Framework, integriert in die Syntax von C# und Visual Basic
- **Query Builder-Methoden** Einzelne Methoden wie `Where()` und `OrderBy()`, die Zeichenkette als Parameter akzeptieren – gut geeignet, um LINQ to Entities-Abfragen dynamisch zu machen
- **Direkte SQL-Ausführung** (Ab Entity Framework 4.0).

## Abfragen mit LINQ to Entities

Die Abfrage mit LINQ to Entities basiert komplett auf der Syntax von LINQ (`from`, `where`, `select`, `orderby`, `Skip()`, `SingleOrDefault()`, `ToList()` etc.). LINQ to Entities-Abfragen führt man auf den als `ObjectSet<T>` deklarierten Mitgliedern der Kontextklasse aus, die jeweils die Gesamtheit aller Instanzen einer Entitätsklasse repräsentieren.

## Einfache Abfrage mit Bedingung

Das folgende Beispiel zeigt eine LINQ to Entities-Abfrage aller Flüge von einem Abflugort, die mindestens einen Passagier haben und auf denen mindestens ein Passagier mit einem bestimmten Nachnamen gebucht ist. Für diese Abfrage werden also schon die Beziehungen zwischen den Entitäten ausgenutzt. Das Entity Framework erzeugt automatisch einen SQL-Befehl mit den notwendigen Joins über die vier Tabellen Flug, Flug\_Passagier, Passagier und Person.

```
// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWings6Entities modell = new WWings6Entities())
{
    string ort = "Rom";
    string name = "Müller";

    // Abfrage definieren
    var fluege = from f in modell.Flug
                where f.Abflugort == ort && f.Passagier.Count > 0 && f.Passagier.Any(p => p.Person.Name
== name)
                select f;
    // Ergebnis ausgeben
    foreach (WWings.G0.EF.Flug f in fluege)
    {
        Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!", f.FlugNr, f.Abflugort, f.Zi
elort, f.FreiePlaetze);
    }
} // Ende using-Block -> Dispose() wird aufgerufen
```

**Listing 12.5** Beispiel für eine Mengenabfrage mit LINQ to Entities

## Abfrage mit Paging

Besonders eindrucksvoll wird die Prägnanz von LINQ to Entities gegenüber SQL beim Einsatz von Paging. Paging bedeutet, dass aus einer Ergebnismenge nur ein bestimmter Teilbereich geliefert werden soll, z.B. Datensätze 20 bis 29. Dies realisiert man in LINQ to Entities wie in LINQ to Objects mit den Methoden Skip() und Take() (bzw. den Sprachelementen Skip und Take in Visual Basic .NET). Verglichen mit dem umfangreichen SQL-Code, den man normalerweise dafür schreiben muss, ist LINQ to Entities hier eine Revolution!

```
// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWings6Entities modell = new WWings6Entities())
{
    string ort1 = "Frankfurt";
    string ort2 = "München";
    string ort3 = "Berlin";
    int von = 20;
    int anzahl = 10;

    // Abfrage definieren mit Paging -> orderby erforderlich!
    var fluege = (from f in modell.Flug
                where f.Abflugort == ort1 || f.Abflugort == ort2 || f.Abflugort == ort3
                orderby f.FlugNr
                select f).Skip(von).Take(anzahl);

    // Ergebnis ausgeben
    foreach (WWings.G0.EF.Flug f in fluege)
```

```
{
    Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
        f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);
}
} // Ende using-Block -> Dispose() wird aufgerufen
```

**Listing 12.6** Beispiel für eine Mengenabfrage mit LINQ to Entities inklusive Paging

**ACHTUNG** In Abfragen mit Paging muss man bei LINQ to Entities immer eine Sortierbedingung explizit mit angeben (hier *orderby f.FlugNr*), sonst kommt es zu einem Laufzeitfehler.

## Abfrage nach Einzelobjekten

Die sprachintegrierte Suchsprache LINQ (Language Integrated Query) bietet schon seit der ersten Version in .NET 3.5 vier Operationen, um das erste bzw. einzige Element einer Menge zu wählen:

- **First()** Das erste Element einer Menge. Wenn mehrere Elemente in der Menge sind, werden alle anderen bis auf das erste verworfen. Wenn es kein Element gibt, tritt ein Laufzeitfehler auf.
- **FirstOrDefault()** Das erste Element einer Menge oder ein Standardwert (bei Referenztypen null bzw. Nothing), wenn die Menge leer ist. Wenn mehrere Elemente in der Menge sind, werden alle anderen bis auf das erste verworfen.
- **Single()** Das einzige Element einer Menge. Wenn es kein Element gibt oder wenn mehrere Elemente in der Menge sind, tritt ein Laufzeitfehler auf.
- **SingleOrDefault()** Das einzige Element einer Menge. Wenn es kein Element gibt, wird der Standardwert (bei Referenztypen null oder Nothing) geliefert. Wenn mehrere Elemente in der Menge sind, tritt ein Laufzeitfehler auf.

Während in LINQ to SQL alle vier Operationen von Anfang an unterstützt wurden, waren in LINQ to Entities, der LINQ-Variante in ADO.NET Entity Framework, in der Version 1.0 die Operatoren `Single()` und `SingleOrDefault()` nicht unterstützt. In den meisten Situationen (z.B. Zugriff auf ein Element über den Primärschlüssel, wobei es höchstens ein Element in der Ergebnismenge geben kann) konnte man zwar `First()` bzw. `FirstOrDefault()` als Ersatz verwenden. Aber es gab einen extrem ärgerlichen Umstand: Wenn man `Single()` oder `SingleOrDefault()` im Programmcode für den Zugriff auf ein EF-Modell verwendet hat, gab es keine Fehler von dem Compiler, sondern erst zur Laufzeit eine Ausnahme der Klasse `NotSupportedException`.

In EF 4.0 hat Microsoft dann doch die Operationen `Single()` und `SingleOrDefault()` implementiert.

**HINWEIS** `First()` und `FirstOrDefault()` beschränken die Ausgabemenge datenbankseitig mit dem SQL-Operator *TOP(1)*. `Single()` und `SingleOrDefault()` verwenden ein *Top(2)* und stellen damit fest, ob es mehr als ein Element gibt, was dann zum Laufzeitfehler führt.

```
/// <summary>
/// Abfrage nach Einzelobjekten
/// </summary>
public static void LTE_Beispiel3_Einzelobjekt()
```

```

{
    // Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
    using (WWings6Entities modell = new WWings6Entities())
    {
        int flugNr = 110;

        // Abfrage definieren mit Einschränkung auf ein Objekt
        var flug = (from f in modell.Flug
                    where f.FlugNr == flugNr
                    select f).SingleOrDefault();

        Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
                          flug.FlugNr, flug.Abflugort, flug.Zielort, flug.FreiePlaetze);
    } // Ende using-Block -> Dispose() wird aufgerufen
}

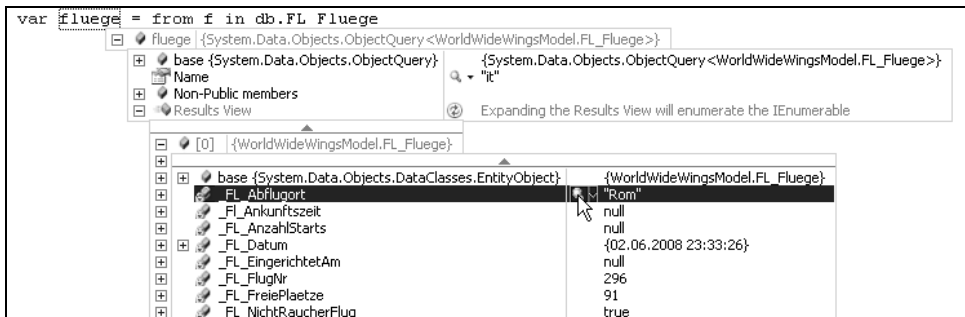
```

**Listing 12.7** Beispiel für eine Einzelobjektabfrage über den Primärschlüssel mit LINQ to Entities

## Protokollierung und Tracing

DasObjectContext-Objekt des EF bietet leider keine einfache Protokollierung wie das DataContext-Objekt in LINQ to SQL über das Log-Attribut. Auch kann man den hinter einer definierten LINQ-Abfrage liegenden Befehl nicht einfach mit ToString() ausgegeben. Im EF muss man erst das Abfrageobjekt in ein typisiertes ObjectQuery-Objekt umwandeln, um dort die Methode ToTraceString() aufzurufen.

```
(ObjectQuery<WWingsModelKompakt.FL_Fluege>)fluege).ToTraceString())
```



**Abbildung 12.17** Der Debugger zeigt auf Anforderung die Ergebnismenge, nicht aber den SQL-Befehl, wie dies bei LINQ to SQL erfolgt

### TIPP

Ein interessantes (aber kostenpflichtiges!) Zusatzwerkzeug ist der Entity Framework Profiler, mit dem man alle vom Entity Framework erzeugten SQL-Befehle betrachten kann (siehe <http://efprof.com/>).

## Konvertierungsfunktionen

Die LINQ-Konvertierungen mit ToArray(), ToLookup(), ToList() und ToDictionary() sind verfügbar. Anders als bei LINQ to SQL bewirken sie aber keine Loslösung vom Kontext. Genau wie bei LINQ to SQL wird die Abfrage durch obige Methoden aber sofort ausgeführt.

```
// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWWings6Entities modell = new WWWings6Entities())
{
    var fluege = from f in modell.Flug
                 where f.Abflugort == "Rom"
                 select f;

    WWWings.GO.EF.Flug flug = fluege.ToArray()[0];
    Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
        flug.FlugNr, flug.Abflugort, flug.Zielort, flug.FreiePlaetze);

    flug.FreiePlaetze--;
    Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
        flug.FlugNr, flug.Abflugort, flug.Zielort, flug.FreiePlaetze);

    modell.SaveChanges();

    Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
        flug.FlugNr, flug.Abflugort, flug.Zielort, flug.FreiePlaetze);
}
```

**Listing 12.8** Dieses Beispiel beweist, dass beim EF die Objekte durch die Konvertierungsmethoden nicht vom Kontext gelöst werden

## Vorkompilierte Abfragen

Sowohl LINQ to SQL als auch das ADO.NET Entity Framework unterstützen die Vorkompilierung von LINQ-Abfragen. Das Übersetzen von LINQ in SQL benötigt einige Zeit. Durch das Vorkompilieren gleichartiger Abfragen kann man die Leistung erheblich steigern, wenn man diese Abfrage mehrfach ausführt (vgl. Abschnitt »Ladegeschwindigkeit«).

```
List<int> FlugListe = new List<int>() { 105, 110, 116, 118, 117, 119 };
// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWWings6Entities modell = new WWWings6Entities())
{
    // Abfrage definieren und vorkompilieren
    Func<WWWings6Entities, int,
    IQueryable<WWWings.GO.EF.Flug>>
    compiledQuery = System.Data.Objects.CompiledQuery.Compile(
        (WWWings6Entities m, int FlugNr) =>
        (
            from f in m.Flug
            where f.FlugNr == FlugNr
            select f));

    foreach (int FlugNr in FlugListe)
    {
        // Abfrage ausführen
        var Ergebnis = compiledQuery(modell, FlugNr);
        // Ergebnis ausgeben
        foreach (WWWings.GO.EF.Flug f in Ergebnis)
        {
            Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
                f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);
        }
    }
} // Ende using-Block -> Dispose() wird aufgerufen
```

**Listing 12.9** Verwendung einer vorkompilierten Abfrage mit LINQ to Entities

## Dynamische Abfragen

Die bisherigen Abfragen waren (bis auf die Parameter) starr. Es gibt aber auch viele Situationen, in denen zum Beispiel die Anzahl der Bedingungen variabel ist. LINQ to Entities bietet hier folgende Möglichkeiten:

- Schrittweises Zusammensetzen von LINQ to Entities-Abfragen
- Verwendung der so genannten Query Builder-Methoden
- Einsatz von eSQL oder SQL (siehe weiter unten)
- Verwendung von Expression Trees (komplexes Thema, in diesem Buch leider aus Platzgründen nicht möglich)

### Schrittweises Zusammensetzen von LINQ to Entities-Abfragen

Das folgende Beispiel zeigt, wie man abhängig von Variablen eine Grundabfrage um verschiedene Bedingungen erweitern kann. Dabei macht man sich zunutze, dass LINQ to Entities genau wie LINQ to Objects mit verzögerter Ausführung arbeitet. Das heißt: Die Datenbank wird erst gefragt, wenn die Objekte tatsächlich angefordert werden, also in der Ausgabeschleife. Solange kann man die Abfrage noch modifizieren.

```
// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWings6Entities modell = new WWings6Entities())
{
    string abflugort = "Berlin";
    string zielort = "Alle";
    int freiePlaetze = 0;

    // Grundabfrage
    var fluege = (from f in modell.Flug
                  where f.Datum > DateTime.Now
                  select f);

    // Optional Bedingungen anfügen
    if (abflugort != "Alle") fluege = from f in fluege where f.Abflugort == abflugort select f;
    if (zielort != "Alle") fluege = from f in fluege where f.Zielort == zielort select f;
    if (freiePlaetze > 0) fluege = from f in fluege where f.FreiePlaetze == freiePlaetze select f;

    // Ergebnis ausgeben (erst jetzt wird Abfrage ausgeführt!)
    foreach (WWings.GO.EF.Flug f in fluege)
    {
        Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
                          f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);
    }
} // Ende using-Block -> Dispose() wird aufgerufen
```

**Listing 12.10** Beispiel für eine dynamische Abfrage mit LINQ to Entities

### Verwendung der so genannten Query Builder-Methoden

Alternativ kann man die Methoden der Klasse `ObjectQuery` (aus dem Namensraum `System.Data.Objects`) verwenden, um die Bedingungen als Zeichenketten anzugeben. Auf den ersten Blick sieht man: Das ist weniger elegant.

---

**HINWEIS** Die bei den Query Builder-Methoden anzugebende Abfragesyntax ist eSQL! »it« ist ein feststehender Bezeichner für das aktuelle Objekt. Es gibt weitere Query Builder-Methoden wie `Top()`, `GroupBy()`, etc.

---



```

/// <summary>
/// Dynamische Abfrage mit LTE und QueryBuilder-Methode
/// </summary>
public static void LTE_Beispiel5_DynamischeAbfrage()
{
    // Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
    using (WWings6Entities modell = new WWings6Entities())
    {
        string abflugort = "Rom";
        string zielort = "Alle";
        int freiePlaetze = 0;

        // Grundabfrage
        string bedingungen = "";

        // Optional Bedingungen anfügen
        if (abflugort != "Alle") bedingungen += (bedingungen != "" ? " and " : "") +
            "it.Abflugort = '" + abflugort + "'";
        if (zielort != "Alle") bedingungen += (bedingungen != "" ? " and " : "") +
            "it.Zielort = '" + zielort + "'";
        if (freiePlaetze > 0) bedingungen += (bedingungen != "" ? " and " : "") +
            "it.FreiePlaetze > " + freiePlaetze + "";

        Console.WriteLine(bedingungen);
        // Bedingungen anwenden
        ObjectQuery<WWings.GO.EF.Flug> fluege = modell.Flug.Where(bedingungen);

        // Ergebnis ausgeben (erst jetzt wird Abfrage ausgeführt!)
        foreach (WWings.GO.EF.Flug f in fluege)
        {
            Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
                f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);
        }
    } // Ende using-Block -> Dispose() wird aufgerufen
}

```

**Listing 12.11** Beispiel für eine dynamische Abfrage mit Query Builder Methods

## Abfragen mit Entity SQL (eSQL)

Eine Abfrage mit eSQL erzeugt man mit der Methode `CreateQuery()` auf einem Objektcontext, siehe Beispiel.

```

// ESQL-Befehl
string ESQL = @"SELECT value f1 " +
    "FROM WWings6Entities.Flug AS f1 " +
    "WHERE f1.Abflugort = @Ort";

// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWings6Entities modell = new WWings6Entities())
{
    // ESQL-Abfrageobjekt erstellen
    ObjectQuery<WWings.GO.EF.Flug> fluege = modell.CreateQuery<WWings.GO.EF.Flug>(
        ESQL, new ObjectParameter("Ort", "Rom"));

    // Ausführen & Ausgabe
    foreach (var f in fluege)
    {
        Console.WriteLine(f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);
    }
}

```

```
{  
    Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",  
        f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);  
}  
} // Ende using-Block -> Dispose() wird aufgerufen
```

**Listing 12.12** Abfrage mit eSQL

## Direkte SQL-Ausführung

Interessant ist die Frage, ob ein ORM-Werkzeug neben einer eleganten und datenbankneutralen Objektabfragesprache auch noch die direkte Verwendung von datenbankspezifischem SQL anbietet. Sinn kann datenbankspezifisches SQL machen, wenn man optimieren will. Bei einer Objektabfragesprache werden die SQL-Befehle generiert und der Entwickler ist diesem Generator »ausgeliefert«. Eine Objektabfragesprache ist also nichts für »SQL-Kontroll-Fetischisten«. Der Autor dieses Buchs hat selbst erlebt, wie es für das EF 1.0 das absolute KO-Kriterium in zwei großen Projekten war, dass die Angabe von individuellem SQL nicht möglich war.

### Methoden für die direkte SQL-Ausführung

Dieses große Manko hat Microsoft in EF 4.0 (größtenteils) beseitigt. EF 4.0 unterstützt in der Klasse `ObjectContext` folgende Funktionen:

- **ExecuteStoreCommand()** Die Methode erlaubt die Angabe eines SQL-Befehls und optionaler Parameter. Rückgabetyt ist eine Zahl (int), die die Anzahl der geänderten Datensätze angibt. Mit `ExecuteStoreCommand()` führt man also SQL-Befehle aus, die keine Ergebnismenge liefern (z.B. Insert, Update, Delete).
- **ExecuteStoreQuery<Typ>()** Unter Angabe eines SQL-Befehls und optionaler Parameter liefert die Methode `ObjectResult<Typ>` zurück. Diese Methode umfasst das Abholen der Daten (per `DataReader` und die Objektmaterialisierung).
- **Translate<Typ>()** Unter Angabe eines geöffneten `DataReader`-Objekts liefert die Methode `ObjectResult<Typ>` zurück. Diese Methode beschränkt sich also auf die Objektmaterialisierung.

### Typen für die Materialisierung

Bei `ExecuteStoreQuery<Typ>()` und `Translate<Typ>()` kann `Typ` sowohl ein primitiver Typ (z.B. Int, String, DateTime), ein Entity-Typ oder eine beliebige .NET-Klasse sein. Voraussetzungen für eine Klasse sind:

- Die Klasse darf nicht abstrakt sein
- Die Klasse muss einen Standardkonstruktor haben
- Jedes Property braucht einen Setter und muss einem CSDL-Typ entsprechen

Die Klasse muss kein Entitätstyp sein. Die Klasse kann auch außerhalb eines Entity Framework-Modells definiert werden. Allerdings kann man bei Nicht-Entitätsklassen keine Änderungen an den geladenen Objekten zurückspeichern.

## Parametrisierung der SQL-Abfrage

Bei den o.g. Befehlen kann man beliebige SQL-Befehle angeben, auch solche, die man aus Einzelteilen zusammengesetzt hat (Stichwort Parametrisierung). Genau wie bei der direkten Verwendung von ADO.NET sollte man für die Parametrisierung zur Vermeidung von SQL-Injektionsangriffen jedoch besser mit den Instrumenten des .NET Framework arbeiten. Die o.g. Befehle bieten zwei sichere Alternativen für die Parametrisierung:

- Angabe im Stil von ADO.NET durch Instanzen von SqlParameter:

```
var Fluege = db.ExecuteStoreQuery<WWings_EF_GO_POCO.Flug>(
    "Select * from Flug where Abflugort = @A0rt and Zielort = @Z0rt",
    new object[] { new SqlParameter("@A0rt", A0rt), new SqlParameter("@Z0rt", Z0rt) });
```

- Angabe im Stil von Console.WriteLine() und String.Format() durch eine einfache Parameterliste ohne Bezug zum Namensraum System.Data.SqlClient:

```
var Fluege = db.ExecuteStoreQuery<WWings_EF_GO_POCO.Flug>(
    "Select * from Flug where Abflugort = {0} and Zielort = {1}", A0rt, Z0rt );
```

## Änderungsverfolgung

Im Standard materialisieren die o.g. Methoden die Objekte ohne Änderungsverfolgung, d.h. Änderungen an den Objekten können zwar vorgenommen werden, speichern sich jedoch nicht bei Aufruf von SaveChanges(). Dies lässt sich auch nicht nachträglich durch einen Befehl wie

```
ctx.ObjectStateManager.ChangeObjectState(obj, EntityState.Modified);
```

gerade rücken.

Wenn man Änderungsverfolgung möchte, muss man schon bei der Ausführung des SQL-Befehls zwei weitere Parameter angeben:

- Name der Entitätsklasse als Zeichenkette
- Einen der Werte aus der Wertaufzählung System.Data.Objects.MergeOption, wobei NoTracking nicht wirkt.

## Beispiel

Das folgende Beispiel führt eine parametrisierte SQL-Abfrage aus und ändert Werte in den geladenen Flug-Objekten. Flug ist eine Entitätsklasse.

```
/// <summary>
/// Beispiel zur direkten SQL-Ausführung (mit Entitätsklasse)
/// </summary>
public static void Demo_DirektSQL()
{
    string A0rt = "Berlin";
    string Z0rt = "Paris";

    WWings_EF4_Standard.WWings6Entities ctx = new WWings_EF4_Standard.WWings6Entities();
    ctx.ObjectMaterialized +=
        new System.Data.Objects.ObjectMaterializedEventHandler(db_ObjectMaterialized);
```

```
// Beispiel 1: ohne Parameter
// var Fluege = db.ExecuteStoreQuery<WWings_EF_GO_POCO.Flug>("Select * from Flug", null);

// Beispiel 2: mit Parameter und ohne Änderungsverfolgung/ Syntaxform 1
// var Fluege = db.ExecuteStoreQuery<WWings_EF_GO_POCO.Flug>(
    "Select * from Flug where Abflugort = @A0rt and Zielort = @Z0rt",
    new object[] { new SqlParameter("@A0rt", A0rt), new SqlParameter("@Z0rt", Z0rt) });

// Beispiel 3: mit Parameter und ohne Änderungsverfolgung / Syntaxform 2
// var Fluege = ctx.ExecuteStoreQuery<WWings_EF_GO_POCO.Flug>(
    "Select * from Flug where Abflugort = {0} and Zielort = {1}", A0rt, Z0rt );

// Beispiel 4: mit Parameter und mit Änderungsverfolgung / Syntaxform 2
var Fluege = ctx.ExecuteStoreQuery<WWings_EF4_Standard.Flug>(
    "Select * from Flug where Abflugort = {0} and Zielort = {1}", "Flug",
    System.Data.Objects.MergeOption.AppendOnly, A0rt, Z0rt);

// Auflisten und ändern
foreach (var f6 in Fluege)
{
    f6.FreiePlaetze--;
    Console.WriteLine(f6.FlugNr + " = " + f6.FreiePlaetze);
    ctx.ObjectStateManager.ChangeObjectState(f6, EntityState.Modified);
}
// Änderungen speichern
Console.WriteLine("Geänderte Datensätze: " +
    ctx.ObjectStateManager.GetObjectStateEntries(System.Data.EntityState.Modified).Count());
ctx.SaveChanges();
}
```

**Listing 12.13** Beispiel zur direkten SQL-Ausführung (mit Entitätsklasse)

Das zweite Beispiel befüllt eine eigene Klasse, die nicht Entitätsklasse ist.

```
/// <summary>
/// Eigene Klasse für die Befüllung mit SQL-Befehl
/// </summary>
class FlugCustom
{
    public int FlugNr { get; set; }
    public Int16 FreiePlaetze { get; set; }
    public string AbflugOrt { get; set; }
    public string ZielOrt { get; set; }
    public string Zusatz { get; set; }
}

/// <summary>
/// Direktes SQL zum Mapping auf eigene Klasse
/// Änderungsverfolgung/Speichern nicht möglich!
/// </summary>
private static void Demo_DirektSQL2()
{
    WWings_EF4_Standard.WWings6Entities ctx = new WWings_EF4_Standard.WWings6Entities();

    string Ort = "Rom";
    var Fluege = ctx.ExecuteStoreQuery<FlugCustom>("Select FlugNr, AbflugOrt, ZielOrt, FreiePlaetze
from Flug where Abflugort = {0}", "Flug", System.Data.Objects.MergeOption.NoTracking, Ort);

    foreach (var f6 in Fluege)
```

```

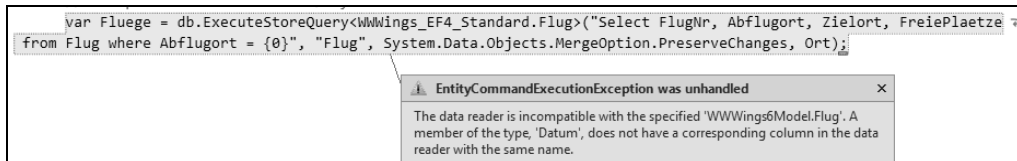
{
    Console.WriteLine(f6.FlugNr + " = " + f6.FreiePlaetze);
}
}

```

**Listing 12.14** Direktes SQL zum Mapping auf eigene Klasse

## Hinweis auf Fallstricke

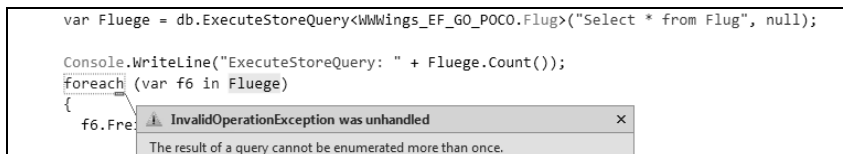
Eine zentrale Einschränkung bei der direkten SQL-Ausführung ist, dass man Entitätsobjekte leider nur komplett befüllen kann, d.h. in dem SQL-Befehl müssen alle Spalten entweder explizit oder durch den Stern-Operator angegeben werden. Die nachstehende Bildschirmabbildung zeigt die Fehlermeldung, wenn man dies missachtet. Das partielle Befüllen von Nicht-Entitätsklassen ist aber möglich.



**Abbildung 12.18** Fehler beim Versuch, ein Entitätsobjekt partiell zu befüllen

Die Einschränkung, Entitätsklassen nicht partiell befüllen zu können, ist ärgerlich. In LINQ to SQL ist dies möglich.

Das Ergebnis einer direkten SQL-Ausführung kann man nur ein einziges Mal verwenden, siehe folgende Abbildung.



**Abbildung 12.19** Fehler beim Versuch, ein Ergebnis mehrmals zu verwenden

Eine Wiederverwendung ist nur im RAM möglich, indem man das Ergebnis in einer Liste zwischenspeichert, dann kann man sowohl Count() als auch den Enumerator aufrufen. Das Zählen erfolgt dann im RAM mit LINQ to Objects.

```

var FluegeList = Fluege.ToList();
Console.WriteLine("ExecuteStoreQuery: " + FluegeList.Count());
foreach (var f6 in FluegeList)
{ ... }

```

## MergeOptions

Die Auflistung System.Data.Objects.MergeOption wird verwendet, um beim Laden von Objekten aus der Datenbank festzulegen, wie das Entity Framework mit den geladenen Objekten umgeht, insbesondere in Bezug auf bereits bestehende Objekte im RAM. System.Data.Objects.MergeOption kann man angeben bei

- dem Entity Framework-Kontext für komplette Mengen:

```
db.Flug.MergeOption = MergeOption.NoTracking;
```

- LINQ-Abfragen durch Typkonvertierung auf `ObjectQuery`:

```
var FlugQuery = (from x in db.Flug select x).OrderBy(f4 => f4.FlugNr).Skip(10);  
(FlugQuery as ObjectQuery).MergeOption = MergeOption.PreserveChanges;
```

- SQL-Abfragen als Parameter bei `ExecuteStoreQuery()` oder `Translate()` – dies ist neu ab Entity Framework 4.0:

```
var Fluege = ctx.ExecuteStoreQuery<WWings_EF4_Standard.Flug>("Select * from Flug where Abflugort = {0} and Zielort = {1}", "Flug", System.Data.Objects.MergeOption.AppendOnly, A0rt, Z0rt);
```

`System.Data.Objects.MergeOption` bietet vier Optionen:

- **NoTracking** Objekte werden im Zustand *losgelöst* (Detached) geladen und bieten daher keine Änderungsverfolgung an
- **AppendOnly** Objekte, die im Objektkontext nicht vorhanden sind, werden an den Kontext angefügt. Wenn ein nun zu ladendes Objekt bereits im Kontext vorhanden ist, wird es in seinem aktuellen Zustand belassen
- **OverwriteChanges** Objekte, die im Objektkontext nicht vorhanden sind, werden an den Kontext angefügt. Wenn ein nun zu ladendes Objekt bereits im Kontext vorhanden ist und dort andere Werte besitzt, werden diese anderen Werte überschrieben (das Objekt wird also aus der Datenquelle aktualisiert).
- **PreserveChanges** Objekte, die im Objektkontext nicht vorhanden sind, werden an den Kontext angefügt. Objekte im Objektkontext werden aktualisiert, wenn Sie den Zustand *unverändert* (Unchanged) haben. Objekte im Objektkontext im Zustand *modified* werden Attribut für Attribut verglichen. Abgeänderte Werte solcher Objekte werden im Kontext beibehalten. Unveränderte Werte werden überschrieben, wenn es neue Werte in der Datenbank gibt.

---

**ACHTUNG** Bei `PreserveChanges` gibt es eine Änderung in Entity Framework 4.0. Entity Framework 1.0 hat bei Objekten im Objektkontext im Zustand *modified* einzelne unmodifizierte Attribute einfach mit Werten aus der Datenbank überschrieben. In Entity Framework 4.0 neu ist, dass dabei diese Attribute als *modifiziert* gekennzeichnet werden.

Um in Entity Framework 4.0 das alte Verhalten beizubehalten, kann man im Objektkontext das Attribut `ContextOptions.UseLegacyPreserveChangesBehavior` auf `true` setzen.

---

---

**TIPP** Die Verwendung von `NoTracking` macht Abfragen wesentlich schneller. Nutzen Sie `NoTracking`, wenn Sie Daten nicht ändern wollen!

---

## Serialisierung

Der Visual Studio-EDM-Designer erzeugt für alle Geschäftsobjektklassen die Annotationen `[Serializable]` und `[DataContract]`. Unterstützt werden somit alle Serialisierer des .NET Framework. Assoziationen mit `EntityCollection` und `EntityReference` sind auch serialisierbar, sodass ein ganzer Objektbaum serialisiert

werden kann (Graph Serialization). Laut der Dokumentation kann lediglich der XML-Serialisierer die Beziehungen nicht serialisieren (»XML serialization does not serialize related objects.«). Auf jeden Fall werden nur die Geschäftsobjekte selbst, nicht aber auch der Objektkontext serialisiert.

**HINWEIS** Im Standard werden nur die Attribute eines Entitätsobjekts einschließlich aller Beziehungen serialisiert, nicht aber der Zustand der Objekte. Die Self-Tracking-Entities (siehe späterer Abschnitt) können auch den Zustand serialisieren.

## Navigation und Ladestrategien

Der Entity Framework-Assistent erstellt beim Reverse Engineering automatisch Assoziationen zwischen Entitäten anhand der in der Datenbank hinterlegten Fremdschlüsselbeziehungen. Diese Assoziationen wurden schon in einer Abfrage (Zur Erinnerung: ... where f.Abflugort == ort && f.Passagier.Count > 0 && f.Passagier.Any(p => p.Person.Name == name) ...) verwendet. Man kann die Assoziationen aber auch zur Navigation in der Ergebnismenge verwenden.

### Automatisches Nachladen

Das folgende Beispiel zeigt, wie man innerhalb der Flugmenge zu den zugeordneten Passagieren und deren Personendaten navigieren kann.

Dieses Beispiel ist aber extrem ineffizient, denn das Programm sendet eine Vielzahl von SQL-Befehlen zur Datenbank:

- Zunächst wird die Liste der Flüge geladen
- Dann wird für jeden Flug die Passagierliste geladen
- Dann werden für jeden Passagier einzeln noch die Personendaten geladen

Grund dafür ist die Verwendung des transparenten Lazy Loading.

```
// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWWings6Entities modell = new WWWings6Entities())
{

    string ort = "Rom";
    string name = "Müller";

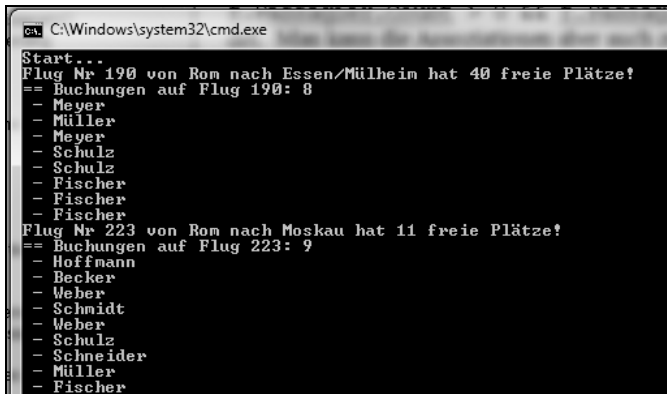
    // Abfrage definieren
    var fluege = from f in modell.Flug
        where f.Abflugort == ort && f.Passagier.Count > 0 &&
            f.Passagier.Any(p => p.Person.Name == name)
        select f;

    // Ergebnis ausgeben
    foreach (WWWings6Entities.Flug f in fluege)
    {
        Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
            f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);
    }
}
```

```
// Navigation zu verbundenen Objekten
Console.WriteLine("== Buchungen auf Flug " + f.FlugNr + ": " + f.Passagier.Count());

foreach (WWings.GO.EF.Passagier ps in f.Passagier)
{
    Console.WriteLine(" - " + ps.Person.Name);
}
}
}
} // Ende using-Block -> Dispose() wird aufgerufen
```

**Listing 12.15** Modifizierte Version der Flugausgabe



```
Start...
Flug Nr 190 von Rom nach Essen/Mülheim hat 40 freie Plätze!
== Buchungen auf Flug 190: 8
- Meyer
- Müller
- Meyer
- Schulz
- Schulz
- Fischer
- Fischer
- Fischer
Flug Nr 223 von Rom nach Moskau hat 11 freie Plätze!
== Buchungen auf Flug 223: 9
- Hoffmann
- Becker
- Weber
- Schmidt
- Weber
- Schulz
- Schneider
- Müller
- Fischer
```

**Abbildung 12.20** Ausgabe des obigen Beispiels

Der obige »Fehler« hätte in Entity Framework 1.0 gar nicht passieren können, denn dort gab es noch kein automatisches Nachladen. EF 1 unterstützte zwar grundsätzlich Lazy Loading, aber nur ein explizites *Lazy Loading*, bei dem der Entwickler seinen Bedarf stets mit dem Ausführen der Methode `Load()` dem Entity Framework explizit kundtun musste. Microsofts Entwickler hatten es dabei durchaus gut gemeint, weil sie wollten, dass ihre Kunden sich bewusst machen, wann weitere Datenbankzugriffe erfolgen. Aber in der Praxis war das ständige Ausführen von `Load()` nicht nur lästig, sondern auch eine Fehlerquelle.

---

**HINWEIS** Transparentes Lazy Loading wie in LINQ to SQL unterstützt das Entity Framework erst in Version 4.0. Indem man `ObjectContext.ContextOptions.LazyLoadingEnabled` im Kontext auf `true` setzt (dies ist die neue Standardeinstellung), aktiviert man transparentes Lazy Loading für alle folgenden Zugriffe, d.h. ein Nachladen erfolgt automatisch auch ohne Ausführen von `Load()`.

---



---

**TIPP** Ob das transparente Lazy Loading aktiv ist, kann man über `model.ContextOptions.LazyLoadingEnabled` abfragen.

---

## Explizites Lazy Loading

Durch Setzen von `ObjectContext.ContextOptions.LazyLoadingEnabled=false` kann man das Verhalten wie in EF 1 erreichen. Dafür stellen die Klassen `EntityCollection` und `EntityReference` die Methode `Load()` bereit. Vorher kann man mit `IsLoaded` prüfen, ob das Laden schon erfolgt ist. In diesem Kapitel wurde ja schon



erläutert, das es für eine 1:1-Beziehung zwei Attribute gibt. Für das Nachladen ist das Attribut zu verwenden, das auf Reference endet (und vom Typ `EntityReference` ist) und nicht das einfache Attribut, das vom Typ der Zielklasse ist und daher kein `Load()` bietet.

Neu in Entity Framework 4.0 ist die Schreibweise `ctx.LoadProperty(flug, f => f.Passagier)` statt `flug.Passagier.Load()`. Diese neue Schreibweise ist normalerweise eine Option. Dies ist aber die einzige Möglichkeit für das Nachladen im Fall von POCO-Klassen (vgl. Abschnitt »POCO-Vorlage«), denn dort werden Beziehungen nicht durch die Klassen `EntityCollection` und `EntityReference` ausgedrückt, die `Load()` bereitstellen.

```
// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWWings6Entities modell = new WWWings6Entities())
{
    // Explizites Nachladen erzwingen!
    modell.ContextOptions.LazyLoadingEnabled = false;

    string ort = "Rom";
    string name = "Müller";

    // Abfrage definieren
    var fluege = from f in modell.Flug
        where f.Abflugort == ort && f.Passagier.Count > 0 &&
            f.Passagier.Any(p => p.Person.Name == name)
        select f;

    // Ergebnis ausgeben
    foreach (WWWings.G0.EF.Flug f in fluege)
    {
        Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!", f.FlugNr,
            f.Abflugort, f.Zielort, f.FreiePlaetze);

        // Explizites Laden
        // f.Passagier.Load();
        // Alternative. Für POCO-Klassen einzige Alternative!
        modell.LoadProperty(f, x => x.Passagier);

        // Navigation zu verbundenen Objekten
        Console.WriteLine("== Buchungen auf Flug " + f.FlugNr + ": " + f.Passagier.Count());

        foreach (WWWings.G0.EF.Passagier ps in f.Passagier)
        {
            // Nochmals explizites Laden notwendig!
            // ps.PersonReference.Load();
            // Alternative. Für POCO-Klassen einzige Alternative!
            modell.LoadProperty(ps, p => p.Person);

            Console.WriteLine(" - " + ps.Person.Name);
        }
    }
}
// Ende using-Block -> Dispose() wird aufgerufen
```

**Listing 12.16** Beispiel für Nicht-Transparentes (explizites) Lazy Loading

## Eager Loading

In diesem Beispiel ist weder das transparente Lazy Loading noch das explizite Lazy Loading die richtige Wahl, denn es steht ja schon zu Beginn fest, dass die verbundenen Objekte gebraucht werden. Hier sollte man also Eager Loading einsetzen.

Anstelle des `DataLoadOptions`-Objekts in LINQ to SQL tritt in EF die Methode `Include()` auf, bei der man einen Ladepfad (leider weiterhin nur als Zeichenkette – in diesem Punkt ist LINQ to SQL immer noch besser) angeben kann. Der folgende LINQ-Befehl lädt die Flüge sowie die zugehörigen Passagiere und deren Personendaten. Dabei ist die Zwischentabelle `Flug_Passagier` nicht explizit anzugeben, weil diese Tabelle durch das EDM bereits eliminiert wurde. Der Ladepfad bezieht sich also auf die Entitäten im Modell, nicht auf die Tabelle. Wenn man die Entitäten im Modell umbenannt hat, muss man also auch diese anderen Namen hier verwenden.

```
string ort = "Rom";
string name = "Müller";

// Abfrage definieren
var fluege = from f in modell.Flug.Include("Passagier.Person")
             where f.Abflugort == ort && f.Passagier.Count > 0 &&
                   f.Passagier.Any(p => p.Person.Name == name)
             select f;

// Ergebnis ausgeben
foreach (WWings.GO.EF.Flug f in fluege)
{
    Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
                      f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);

    // Navigation zu verbundenen Objekten
    Console.WriteLine("== Buchungen auf Flug " + f.FlugNr + ": " + f.Passagier.Count());

    foreach (WWings.GO.EF.Passagier ps in f.Passagier)
    {
        Console.WriteLine(" - " + ps.Person.Name);
    }
}
} // Ende using-Block -> Dispose() wird aufgerufen
```

**Listing 12.17** Beispiel für Eager Loading

## Diskussion der Ladestrategien

Alle drei obigen Listings liefern genau die gleiche Ausgabe. Der wesentliche Unterschied sind der Programmierstil, Risiken und Anzahl der ausgeführten SQL-Befehle.

	Transparentes Lazy Loading	Nicht-Transparentes (explizites) Lazy Loading	Eager Loading
Anzahl der ausgeführten SQL-Befehle	1 für Flug + 1 für Passagierliste + Anzahl der Passagier	1 für Flug + 1 für Passagierliste + Anzahl der Passagier	Genau 1!
Risiken	Entwickler übersieht, dass er so viele SQL-Befehl ausführt und wundert sich über schlechte Leistung	Entwickler vergisst ein Load() und bekommt keine Ergebnisse bzw. eine <code>NullReferenceException</code>	Man lädt Daten, die man vielleicht gar nicht braucht; der SQL-Befehl umfasst viele Joins und wird dadurch langsam
Programmierstil	Einfach	Komplex	Mittel

**Tabelle 12.2** Vergleich der Ladeoptionen in Entity Framework 4.0

## Drei Warnungen vor bösen Überraschungen

Das transparente Lazy Loading kann Ihnen bei der Serialisierung zum Verhängnis werden! Wenn Sie das transparente Lazy Loading aktiviert haben, kann beim Serialisieren eines einzelnen Objekts ein sehr großer Objektgraph erzeugt werden, denn die Serialisierer greifen auf alle Attribute (auch die Navigationsattribute) zu und sorgen damit für ein Nachladen. Im Extremfall kann so die ganze Datenbank serialisiert werden. Gegebenenfalls sollte man vorher das transparente Lazy Loading abschalten.

Ein Nachladen nach der Deserialisierung kann aber dann nicht stattfinden, denn das serialisierte Objekt ist losgelöst vom Kontext. Erst nach dem erneuten Anfügen mit `Attach()` ist ein Nachladen wieder möglich.

Es gibt einen Serialisierer, bei dem sich die Frage nicht stellt: Der `System.Xml.Serialization.XmlSerializer` serialisiert die Navigationsattribute nicht. Sehr gut betrachten kann man die Serialisierung aber mit dem `System.Runtime.Serialization.NetDataContractSerializer`. Der `System.Runtime.Serialization.Formatters.Soap.SoapFormatter` eignet sich nicht, weil er mit `Nullable<T>` nicht klarkommt.

Das Entity Framework optimiert auch keine Nachladeoperationen. Wenn Sie einen Flug laden und in `flug101` im RAM ablegen und dann `flug101.Pilot.Mitarbeiter.Person.Name` aufrufen, führt dies zu drei unabhängigen Nachladevorgängen. Das `Pilot`-Objekt, das `Mitarbeiter`-Objekt und das `Person`-Objekt werden jeweils unabhängig voneinander mit drei SQL-Befehlen geladen.

Das Einschalten des Eager Loading mit `Include()` bedeutet nicht, dass es zu keinem Lazy Loading mehr kommen kann. Wenn Sie zum Beispiel `Flug.Include("Pilot.Mitarbeiter")` im LINQ-Befehl festlegen, dann aber für ein Objekt `flug101` die Anweisung `flug101.Pilot.Mitarbeiter.Person` aufrufen, wird das `Person`-Objekt nachgeladen.

Ein Nachladen kann auch erfolgen, wenn Sie `Flug.Include("Pilot.Mitarbeiter.Person")` festgelegt haben im Fall eines Wechsels eines Objekts. Wenn dem Flug 101 vorher der Pilot 140 zugeordnet ist und Sie dann festlegen:

```
flug101.Pilot_PersonID = 155;
```

dann führt

```
flug101.Pilot.Mitarbeiter.Person.Name
```

doch wieder zu drei unabhängigen Nachladenvorgängen, denn ursprünglich wurden ja die Objekte für den Pilot 140 geladen und nicht für 155.

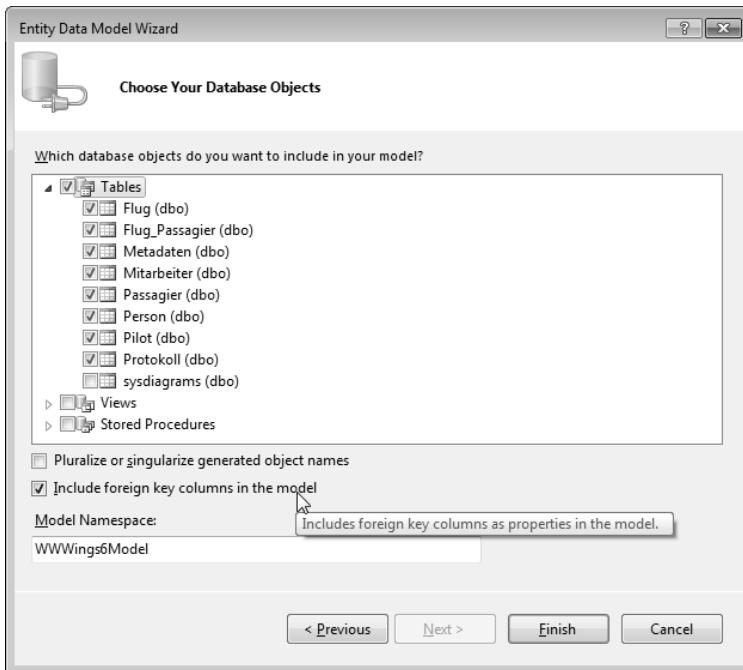
Dies kann man nur optimieren, indem man erneut die Abfrage komplett ausführt:

```
flug101 = (from f in modell.Flug.Include("Pilot.Mitarbeiter.Person")
          where f.FlugNr == 101 select f).SingleOrDefault();
```

Dies ist dann nur ein SQL-Befehl statt drei!

## Objektzuweisungen

Bereits bei der Vorstellung des Assistenten wurde die neue Fremdschlüsselloption erwähnt. Der alte Designer in Visual Studio 2008 erzeugte für zwei in Beziehung stehende (»assozierte«) Entitätsklassen ein Navigationsattribut (z.B. `Flug.Pilot`), aber nicht das dieser Assoziation auf Datenbankebene zugrunde liegende Fremdschlüsselattribut (z.B. `Flug.Pilot_PersonID`). Das bedeutete in der Praxis nicht nur unnötigen Overhead beim Zuweisen (weil dafür immer der Vorgesetzte geladen werden muss), sondern war auch unhandlich bei der Gestaltung von Oberflächen, denn Steuerelemente wie `DropDownList` konnten mit Navigationsattributen nichts anfangen. In Entity Framework 4.0 erhält der Entwickler auf Wunsch auch die Fremdschlüsselattribute, wenn er im Assistenten das entsprechende Häkchen setzt.



**Abbildung 12.21** Aktivieren der Fremdschlüsselbeziehungen

## Beziehungen herstellen

Das folgende Listing zeigt drei Varianten, wie man einem Flug einen Piloten zuweisen kann (Hinweis: Bei der Fluggesellschaft gibt es aus Rationalisierungsgründen nur noch einen Piloten für jeden Flug). Die erste Variante `flug.Pilot = neuerPilot` ist der objektorientierte Ansatz, der aber (meist) sehr ineffizient ist, weil

das Pilot-Objekt erst geladen werden muss, auch wenn es ansonsten gar nicht gebraucht würde. In Entity Framework 1.0 konnte man sich hier nur mit einer Instanz der Klasse EntityKey (aus dem Namensraum System.Data) behelfen (Variante 2). Da hier der Klassenname und der Name des/der Primärschlüsselattribut(e) als Zeichenkette angegeben werden mussten, bestand die Gefahr von Eingabefehlern. Zudem ist dieser Befehl sehr umständlich. Entity Framework 4.0 erlaubt nun auch die direkte Zuweisung über das Attribut Pilot.PersonID, das in der Klasse Flug zusätzlich zu dem Navigationsobjekt Pilot erzeugt wird, wenn man die o.g. Option während der Modellerzeugung wählt.

```
// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWWings6Entities modell = new WWWings6Entities())
{
    int flugNr = 110;
    int pilotNr = 171;

    // Flug laden
    WWWings.GO.EF.Flug flug =
        (from f in modell.Flug.Include("Pilot") where f.FlugNr == flugNr select f).FirstOrDefault();
    Console.WriteLine("Pilot vorher: " + flug.Pilot.PersonID);

    // Variante 1: EF1 und EF4: Pilot laden und zuweisen
    WWWings.GO.EF.Pilot neuerPilot = modell.Pilot.Where(p => p.PersonID == pilotNr).FirstOrDefault();
    flug.Pilot = neuerPilot;

    // Variante 2: EF1 und EF4: Entity Key erzeugen und zuweisen
    flug.PilotReference.EntityKey = new EntityKey("WWWings6Entities.Pilot", "PersonID", pilotNr);

    // Variante 3: nur EF4: Direkter Zugriff
    flug.Pilot.PersonID = pilotNr;

    // Speichern
    Console.WriteLine("Gespeicherte Änderungen: " + modell.SaveChanges());

    // Kontrolle
    flug =
        (from f in modell.Flug.Include("Pilot") where f.FlugNr == flugNr select f).FirstOrDefault();
    Console.WriteLine("Pilot nachher: " + flug.Pilot.PersonID);
} // Ende using-Block -> Dispose() wird aufgerufen
```

**Listing 12.18** Zuweisung von Objekten in drei Varianten

## Relationship Fix Up

Als Teil des »Relationship Fix Up« sorgt das ADO.NET Entity Framework dafür, dass sowohl Navigationsattribut als auch Fremdschlüsselattribut synchron sind.

- Weist man dem Navigationsattribut ein Objekt zu, setzt EF direkt das Fremdschlüsselattribut
- Weist man dem Fremdschlüsselattribut einen neuen Wert zu, lädt das EF das Objekt nach. Allerdings variiert der Zeitpunkt dieses Nachladens von der Vorlage: Bei der Standardcodegenerierung erfolgt das Nachladen sofort bei der Zuweisung. Es werden auch alle abhängigen Objekte mit geladen, die vorher schon geladen waren. Bei der POCO-Vorlage löscht ein Ändern des Fremdschlüssels das Navigationsattribut und lädt dann beim nächsten das passende Objekt aus der Datenbank nach.

**ACHTUNG** Wenn der Fremdschlüssel ungültig ist, kommt es normalerweise beim Speichern zu einem Fehler. Erfolgt zuvor jedoch ein Zugriff auf das Navigationsattribut, dann kommt es bereits in diesem Moment zum Fehler.

```
public virtual Nullable<int> Pilot_PersonID
{
    get { return _pilot_PersonID; }
    set
    {
        try
        {
            settingFK = true;
            if (_pilot_PersonID != value)
            {
                if (Pilot != null && Pilot.PersonID != value)
                {
                    Pilot = null;
                }
                _pilot_PersonID = value;
            }
        }
        finally
        {
            _settingFK = false;
        }
    }
}
```

**Listing 12.19** Programmcode, den die POCO-Vorlage beim Setzen des Fremdschlüsselattributs ausführt

```
public virtual Pilot Pilot
{
    get { return _pilot; }
    set
    {
        if (!ReferenceEquals(_pilot, value))
        {
            var previousValue = _pilot;
            _pilot = value;
            FixupPilot(previousValue);
        }
    }
}

private void FixupPilot(Pilot previousValue)
{
    if (previousValue != null && previousValue.Flug.Contains(this))
    {
        previousValue.Flug.Remove(this);
    }

    if (Pilot != null)
    {
        if (!Pilot.Flug.Contains(this))
        {
            Pilot.Flug.Add(this);
        }
        if (Pilot_PersonID != Pilot.PersonID)
        {
            Pilot_PersonID = Pilot.PersonID;
        }
    }
}
```

```

    }
    else if (!_settingFK)
    {
        Pilot_PersonID = null;
    }
}

```

**Listing 12.20** Programmcode, den die POCO-Vorlage beim Setzen des Navigationsattributs ausführt

## Mit Fremdschlüssel abfragen

Die Fremdschlüssel vereinfachen auch die Abfrage (siehe Listing). Bitte beachten Sie, dass der Unterschied nur in dem ».« bzw. »\_« zwischen Pilot und PersonID liegt.

```

// Variante 1: in EF1 nicht anders möglich!
var FluegeFuerEinenPiloten1 =
    (from f in modell.Flug where f.Pilot.PersonID == 180 select f);

// Variante 2: ab EF4 möglich!
var FluegeFuerEinenPiloten2 =
    (from f in modell.Flug where f.Pilot_PersonID == 180 select f);

```

**HINWEIS** Dieser kleine Unterschied kann aber erhebliche Geschwindigkeitsverbesserungen mit sich bringen. Der Autor dieses Kapitels hat es selbst in Projekten erlebt.

Zunächst einmal gibt es keinen Geschwindigkeitsvorteil der beiden o.g. Syntaxformen. Das Entity Framework erzeugt in beiden Fällen den gleichen SQL-Befehl. Wenn man sich aber entscheidet, das Ergebnis dieser Abfrage im RAM zwischenspeichern und später in der zwischengespeicherten Menge die Suche mit LINQ to Objects verfeinert, dann ist die Variante 2 erheblich schneller, denn LINQ to Objects muss im ersten Fall jedes Mal über den internen RelationshipManager der Klasse Flug für jede Instanz im Zwischenspeicher gehen. Das kostet Zeit. Viel Zeit! Im zweiten Fall ist der RelationshipManager nicht beteiligt. Die LINQ to Objects-Abfrage ist dann viel effizienter.

Der Wunsch, das Abfrageergebnis zwischenspeichern, ist gerade dann durchaus üblich, wenn es eine Vielzahl von Zugriffen auf eine sich selten verändernde Menge gibt.

```

[XmlIgnoreAttribute()]
[SoapIgnoreAttribute()]
[DataMemberAttribute()]
[EdmRelationshipNavigationPropertyAttribute("WWWings6Model", "FK_FL_Flug_PI_Pilot", "Pilot")]
public Pilot Pilot
{
    get
    {
        return ((IEntityWithRelationships)this).RelationshipManager.GetRelatedReference<Pilot>(
            "WWWings6Model.FK_FL_Flug_PI_Pilot", "Pilot").Value;
    }
    set
    {
        ((IEntityWithRelationships)this).RelationshipManager.GetRelatedReference<Pilot>(
            "WWWings6Model.FK_FL_Flug_PI_Pilot", "Pilot").Value = value;
    }
}

```

**Listing 12.21** Implementierung des Attributs *Pilot* für die Entität *Flug* im generierten Code (Standardvorlage)

# Änderungsverfolgung und Persistierung

Bei der Änderungsverfolgung und Persistierung gibt es erhebliche Unterschiede zwischen den verschiedenen Codegenerierungsvorlagen im Entity Framework. Die folgenden Ausführungen beziehen sich zunächst auf die Standardvorlage. Unterschiede bei anderen Codegenerierungsvorlagen entnehmen Sie bitte den jeweiligen Abschnitten zu den Codegenerierungsvorlagen.

## Änderungen an geladenen Objekten

Der Entwickler kann jederzeit schreibend auf die aus der Datenbank geladenen Entitätsobjekte zugreifen. Er muss dabei weder vor der Schreiboperation diese »ankündigen« noch nachher diese »anmelden«. Die Kontextklasse verfolgt alle Änderungen an den Objekten mit (Change Tracking). Im Fall der Entity Framework-Standardvorlage sorgt dafür die Basisklasse `EntityObject`. Andere Entity Framework-Codegenerierungsvorlagen benutzen aber andere Mechanismen (z.B. Runtime Proxies).

## Objekte hinzufügen

Zum Anfügen eines neuen Objekts kann man bei der Entity Framework-Standardvorlage das Objekt ganz normal mit dem `New`-Operator instanziiieren. Anschließend muss man das neue Objekt der Kontextklasse aber anmelden. Dazu gibt es drei Alternativen:

- `modell.AddToKLASSE(obj);`
- `modell.KLASSE.AddObject(obj);`
- `modell.AddObject("KLASSE", obj);`

## Objekte löschen

Um ein Objekt zu löschen, muss man `modell.DeleteObject(obj)` aufrufen. Die Methode liefert nichts zurück. Ein verbundenes Objekt löscht man mit `Remove()` auf der Menge, z.B. `Flug.Passagier.Remove(Passagier);`

**TIPP**

`DeleteObject()` setzt voraus, dass man das Objekt vorher schon geladen hat. Das kann extrem ineffizient sein, wenn man ein Objekt löschen will, dessen Primärschlüssel man kennt, aber den Rest des Objekts gar nicht gebraucht hat. In dieser Situation gibt es einen Trick: Man konstruiert im RAM ein neues Objekt nur mit dem Primärschlüssel, fügt es dem Kontext mit `Attach()` hinzu und ruft dann Löschen auf. Der Trick funktioniert, weil das Entity Framework zum Löschen im Standard nur den Primärschlüssel kennen muss. Wenn das Modell aber so eingestellt ist, dass beim Speichern der Wert anderer Spalten verglichen werden soll, müssen auch diese befüllt werden.

```
// Löschen ohne Laden
f = new WWings.GO.EF.Flug() { FlugNr = flugNr };
modell.Flug.Attach(f);
modell.DeleteObject(f);
```



**ACHTUNG** ORM eignet sich nicht für Massenlöschen im Sinne von `Delete from Tabelle where x=y`. In diesem Fall sollten Sie immer auf klassische Techniken (SQL oder Stored Procedures) zurückgreifen, da der Einsatz von ORM hier ein Vielfaches langsamer wäre.

## Änderungen speichern

Für das Speichern aller Änderungen im aktuellen Kontext gibt es `SaveChanges()` (statt `SubmitChanges()` wie bei LINQ to SQL). Die Methode liefert als Zahl die Anzahl der durchgeführten Änderungen zurück.

**HINWEIS** `SaveChanges()` speichert alle Änderungen seit dem Laden bzw. dem letzten `SaveChanges()`. Es ist im Entity Framework leider nicht möglich, nur einzelne Änderungen von mehreren erfolgten abzuspeichern.

`SaveChanges()` speichert nur die geänderten Objekte und für die geänderten Objekte auch nur die geänderten Attribute. Man kann aber durch die erneute Zuweisung des eigenen Werts oder durch den Aufruf `SetModifiedProperty()` im `ObjectStateManager` erreichen, dass auch andere Attribute erneut geschrieben werden.

Das folgende Listing zeigt ein Beispiel für Löschen, Anfügen und Speichern.

```
int flugNr = 99;

using (WWings.GO.EF.WWings6Entities modell = new WWings.GO.EF.WWings6Entities())
{
    WWings.GO.EF.Flug f;

    /// Löschen mit Laden
    //f = modell.Flug.Where(x => x.FlugNr == flugNr).SingleOrDefault();

    /// Löschen, wenn schon vorhanden
    //if (f != null)
    //{
    //    //modell.DeleteObject(f);
    //}

    // besser: Löschen ohne Laden
    f = new WWings.GO.EF.Flug() { FlugNr = flugNr };
    modell.Flug.Attach(f);
    modell.DeleteObject(f);

    // Speichern
    modell.SaveChanges();
    Console.WriteLine("Flug gelöscht!");

    // Flug erzeugen
    f = new WWings.GO.EF.Flug();
    f.FlugNr = 99;
    f.Abflugort = "Essen/Mühlheim";
    f.Zielort = "Rom";
    f.Datum = new DateTime(2011, 1, 10);
}
```

```
f.Plaetze = 200;
f.FreiePlaetze = 200;

// Neues Objekt beim Kontext anmelden
modell.AddToFlug(f);
// Alternativ: modell.Flug.AddObject(f);
// Alternativ: modell.AddObject("Flug", f);

// Speichern
modell.SaveChanges();
Console.WriteLine("Flug angefügt!");
}
```

**Listing 12.22** Löschen und Anfügen von Objekten

## Informationen über nicht gespeicherte Änderungen

Per Programmcode kann man jederzeit ein einzelnes Objekt nach seinem Zustand fragen, denn die Geschäftsobjektklassen erben von der Basisklasse `EntityObject` u.a. das Attribut `EntityState`. Diese Aussage gilt aber natürlich nicht für den Fall von POCO-Objekten, die nicht von `EntityObject` erben.

Details über alle erfolgten Änderungen (Liste aller geänderten Objekte, jeweils mit altem Zustand und neuem Zustand) erfährt man von dem `ObjectStateManager` des Kontextes (auch abhängig von der gewählten Codegenerierungsvorlage!).

Objekte anfügen an Mengen oder löschen aus Mengen kann man mit `Add()` oder `Remove()`. Ein einzelnes Objekt kann man auch mit der Methode `DeleteObject()` der Kontextklasse löschen.

## Transaktionen

Die Speicherung der Änderungen bei `SaveChanges()` erfolgt als eine Datenbanktransaktion, d.h. es werden alle oder keine der Änderungen in der Datenbank persistiert.

Man kann auch Datenbanktransaktionen über mehrere Ausführungen von `SaveChanges()` hinweg und sogar über mehrere verschiedene Kontextinstanzen hinweg ausführen, indem man selbst einen Transaktionsbereich definiert.

Die Kontextklasse hat im EF kein explizites `Transaction`-Attribut für die Zuweisung einer Transaktion. Transaktionsunterstützung kann man aber über einen `TransactionScope`-Block (siehe dazu auch das PDF-Kapitel »Enterprise Services und Transaktionen«) oder über das Unterobjekt `Connection` der Kontextklasse aktivieren.

## Beispiel

Das folgende größere Beispiel zeigt das Erzeugen einer neuen Buchung in einer Transaktion:

- Zunächst wird in dem `Flug`-Objekt die Platzanzahl reduziert
- Dann wird ein bestehendes `Passagier`-Objekt der Liste der Passagiere des Fluges angefügt. Dadurch entsteht ein neuer Eintrag in der Zwischentabelle `Flug_Passagier`.
- Es wird jeweils eine Änderungsstatistik mithilfe des `ObjectStateManager` ausgegeben

**HINWEIS** Der Zustand (EntityState) des Flug-Objekts ändert sich durch das Anfügen des Passagiers nicht und die Liste Passagier besitzt kein Attribut zur Zustandssignalisierung. Immerhin listet der ObjectStateManager ein hinzugefügtes Objekt auf und die Änderung wird auch gespeichert.

```

/// <summary>
/// Buchungstransaktion
/// </summary>
public static void Beispiel1_Save()
{
    using (WWWings.GO.EF.WWWings6Entities db = new WWWings.GO.EF.WWWings6Entities())
    {
        int flugNr = 120;

        WWWings.GO.EF.Flug Flug;

        using (System.Transactions.TransactionScope t = new System.Transactions.TransactionScope())
        {
            Flug = (from f in db.Flug.Include("Passagier.Person")
                    where f.FlugNr == flugNr
                    select f).FirstOrDefault();
            Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze und {4} Buchungen!",
                             Flug.FlugNr, Flug.Abflugort, Flug.Zielort, Flug.FreiePlaetze, Flug.Passagier.Count);
            Console.WriteLine("Zustand des Objekts: " + Flug.EntityState);

            // ==> Zunächst wird in dem Flug-Objekt die Platzanzahl reduziert.
            Console.WriteLine("--- Reduzieren der Platzanzahl....");
            Flug.FreiePlaetze--;
            Console.WriteLine("Zustand des Objekts: " + Flug.EntityState);

            GetStatistik(db);

            // StateManager auslesen für ein Objekt
            Console.WriteLine("Anzahl geänderter Attribute: " +
                             db.ObjectStateManager.GetObjectStateEntry(Flug).GetModifiedProperties().Count());
            Console.WriteLine("Neuer Wert: " + db.ObjectStateManager.GetObjectStateEntry(Flug).CurrentValues["FreiePlaetze"]);
            Console.WriteLine("Alter Wert: " + db.ObjectStateManager.GetObjectStateEntry(Flug).OriginalValues["FreiePlaetze"]);

            // Änderungen speichern
            db.SaveChanges();

            // ==> Dann wird ein bestehendes Passagier-Objekt der Liste der Passagiere des Flugs angefügt
            Console.WriteLine("Zustand des Objekts: " + Flug.EntityState);
            Console.WriteLine("--- Anfügen eines Passagiers....");

            WWWings.GO.EF.Passagier Passagier = (from p in db.Passagier.Include("Person")
                                                where p.PersonID == 7
                                                select p).FirstOrDefault();

            Console.WriteLine("Zustand des Objekts: " + Flug.EntityState);
            Flug.Passagier.Add(Passagier);
            GetStatistik(db);
        }
    }
}

```

```

    db.SaveChanges();
    Console.WriteLine("Zustand des Objekts: " + Flug.EntityState);

    t.Complete();
    Console.WriteLine("Buchungstransaktion erfolgreich!");
}

Flug = (from f in db.Flug.Include("Passagier.Person")
        where f.FlugNr == flugNr
        select f).FirstOrDefault();
Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze und {4} Buchungen!",
    Flug.FlugNr, Flug.Abflugort, Flug.Zielort, Flug.FreiePlaetze, Flug.Passagier.Count);
}
}

/// <summary>
///StateManager auslesen: Statistik über alle Objekte
/// </summary>
/// <param name="db">Bestehender Kontext</param>
private static void GetStatistik(WWWings.GO.EF.WWWings6Entities db)
{
    Console.ForegroundColor = ConsoleColor.Yellow;

    Console.WriteLine("# Anzahl geänderter Objekte: " +
        db.ObjectStateManager.GetObjectStateEntries( EntityState.Modified ).Count());
    Console.WriteLine("# Anzahl hinzugefügter Objekte: " +
        db.ObjectStateManager.GetObjectStateEntries( EntityState.Added ).Count());
    Console.WriteLine("# Anzahl gelöschter Objekte: " +
        db.ObjectStateManager.GetObjectStateEntries( EntityState.Deleted ).Count());
    Console.WriteLine("# Anzahl unveränderter Objekte: " +
        db.ObjectStateManager.GetObjectStateEntries( EntityState.Unchanged ).Count());
    Console.ForegroundColor = ConsoleColor.Gray;
}

```

**Listing 12.23** Eine Buchungstransaktion mit den EF Object Services

```

file:///h:/WWW/_TestProjekte/Console_LINQ/bin/Debug/WWW.Console.EXE
=== Start
Flug Nr 101 von Berlin nach Frankfurt hat 94 freie Plätze und 1 Buchungen!
Zustand des Objekts: Unchanged
--- Reduzieren der Platzanzahl...
Zustand des Objekts: Modified
# Anzahl geänderter Objekte: 1
# Anzahl hinzugefügter Objekte: 0
# Anzahl gelöschter Objekte: 0
# Anzahl unveränderter Objekte: 4
Anzahl geänderter Attribute: 1
Neuer Wert: 93
Alter Wert: 94
Zustand des Objekts: Unchanged
--- Anfügen eines Passagiers...
Zustand des Objekts: Unchanged
# Anzahl geänderter Objekte: 0
# Anzahl hinzugefügter Objekte: 1
# Anzahl gelöschter Objekte: 0
# Anzahl unveränderter Objekte: 8
Zustand des Objekts: Unchanged
Buchungstransaktion erfolgreich!
Flug Nr 101 von Berlin nach Frankfurt hat 93 freie Plätze und 2 Buchungen!
=== ENDE

```

**Abbildung 12.22** Ausgabe des Transaktionsbeispiels

## Konflikte erkennen

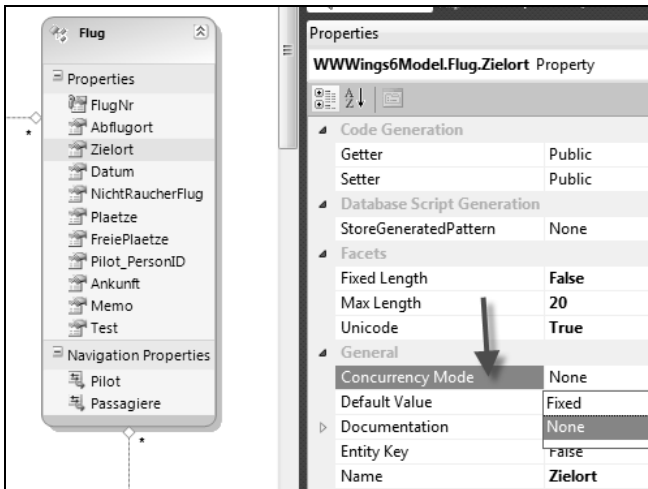
Im Standardfall überschreibt das Entity Framework zwischenzeitliche Änderungen von anderen Vorgängen an Datensätzen, denn bei Speichervorgängen zieht das EF nur die Primärschlüssel heran:

```
exec sp_executesql N'update [dbo].[Flug]
set [FreiePlaetze] = @0
where ([FlugNr] = @1)
',N'@0 smallint,@1 int',@0=9,@1=101
```

**Listing 12.24** Änderung mit Vergleich nur des Primärschlüssels

Der einzige Änderungskonflikt, der so auffallen könnte, wäre ein Löschen des Datensatzes durch einen anderen Vorgang.

Im Modell kann man aber durch `Concurrency Mode=Fixed` festlegen, welche Attribut bei Änderungen (oder beim Löschen) in der Where-Bedingung mit den Ursprungswerten verglichen werden sollen.



**Abbildung 12.23** Festlegung, dass die Spalte bei Änderungen vorher verglichen werden soll

```
exec sp_executesql N'update [dbo].[Flug]
set [FreiePlaetze] = @0
where (((((((([FlugNr] = @1) and ([Abflugort] = @2)) and ([Datum] = @3)) and
([NichtRaucherFlug] = @4)) and ([Plaetze] = @5)) and ([FreiePlaetze] = @6)) and
([Pilot_PersonID] = @7)) and ([Ankunft] = @8)) and [Memo] is null)
',N'@0 smallint,@1 int,@2 nvarchar(20),@3 datetime2(7),@4 bit,@5 smallint,@6 smallint,@7 int,@8
datetime2(7)',@0=6,@1=101,@2=N'Graz-2',@3='2010-01-18
05:39:56.5500000',@4=1,@5=250,@6=7,@7=155,@8='2010-01-18 07:39:56.5500000'
```

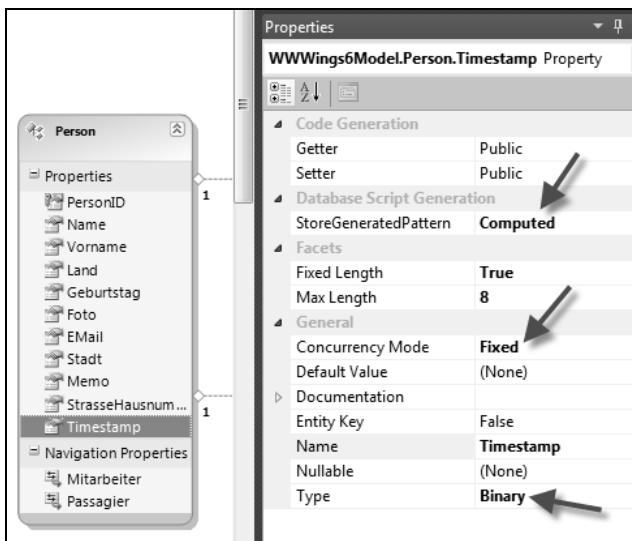
**Listing 12.25** Änderung an einem Flug mit Vergleich aller Spalten mit den Ursprungswerten

Wenn eine Tabelle eine Versionierungsspalte (bei Microsoft SQL Server ist dies der Typ `timestamp`) besitzt, dann kann das EF diese zur Feststellung von Änderungskonflikten heranziehen. Das EF verwendet die Versionierungsspalte als Kriterium bei `Update`- und `Delete`-Befehlen und fragt den von der Datenbank dann automatisch neu gesetzten Wert der Versionierungsspalte nach jeder Änderungsoperation auch wieder ab.

```
exec sp_executesql N'update [dbo].[Person]
set [Stadt] = @0
where ((([PersonID] = @1) and ([Timestamp] = @2))
select [Timestamp]
from [dbo].[Person]
where @@ROWCOUNT > 0 and [PersonID] = @1',N'@0 nvarchar(30),@1 int,
@2 binary(8)',@0=N'Essen',@1=4,@2=0x00000000000000C447
```

**Listing 12.26** Änderung an einer Person mit Versionierungsspalte

Die nachstehende Abbildung zeigt die notwendigen Einstellungen für diese Form der Konfliktfeststellung.



**Abbildung 12.24** Verwendung einer Timestamp-Spalte zur Feststellung von Änderungskonflikten

## Konflikte lösen

Die folgende Bildschirmabbildung dokumentiert einen typischen Änderungskonflikt der sich ergibt, wenn zwei Prozesse den gleichen Wert im RAM ändern. Beim Speichern fällt dann dem zweiten Prozess auf, dass der ursprüngliche Wert nicht mehr vorhanden ist.

Das Entity Framework bietet zwei Möglichkeiten der Reaktion:

- Übernehmen des bereits geänderten Werts
- Überschreiben des bereits geänderten Werts

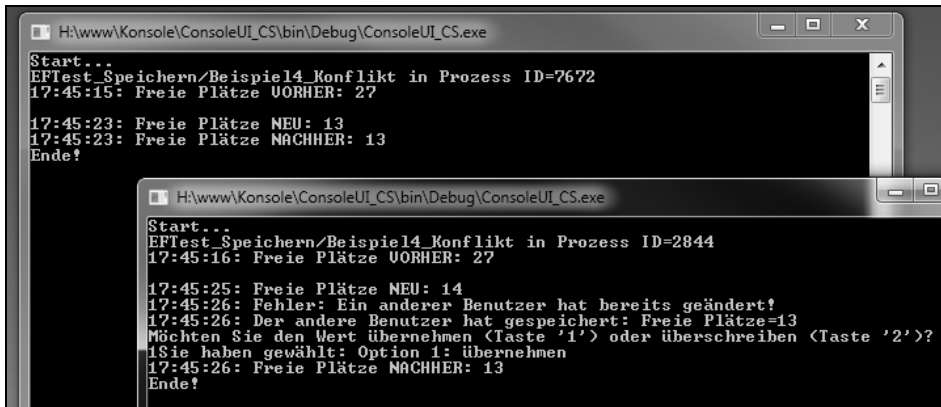


Abbildung 12.25 Ein Änderungskonflikt

Damit der obige Fall auffällt und der als zweites speichernde Prozess nicht abstürzt, sondern dem Benutzer die Wahl zwischen Übernehmen und Überschreiben gegeben werden kann, sind folgende Voraussetzungen notwendig:

- Für das Attribut »Freie Plätze« ist im Modell gesetzt: Concurrency Mode=Fixed
- Bei `SaveChanges()` wird `System.Data.OptimisticConcurrencyException` abgefangen

Das folgende Beispiel zeigt den zugehörigen Programmcode. Wenn man diesen Code zweimal parallel startet, kann man den Änderungskonflikt simulieren.

Zum Lösen des Konflikts stehen folgende Instrumente zur Verfügung:

- `modell.Refresh(RefreshMode.StoreWins, obj)` aktualisiert ein Objekt im RAM mit den aktuellen Werten aus der Datenbank
- `modell.Refresh(RefreshMode.ClientWins, f)` markiert ein Objekt im RAM so, dass beim nächsten Speicherversuch ein Änderungskonflikt ignoriert wird

```

// Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
using (WWWings6Entities modell = new WWWings6Entities())
{

    WWWings.GO.EF.Flug f = modell.Flug.Where(x => x.FlugNr == flugNr).SingleOrDefault();

    Console.WriteLine(DateTime.Now.ToLongTimeString() + ": Freie Plätze VORHER: " + f.FreiePlaetze);
    Console.ReadLine(); // Warten (zum Starten eines zweiten Prozesses!)

    // Änderung
    f.FreiePlaetze -= (short)(new System.Random(DateTime.Now.Millisecond).Next(20));
    Console.WriteLine(DateTime.Now.ToLongTimeString() + ": Freie Plätze NEU: " + f.FreiePlaetze);
    try
    {
        // Speicherversuch
        modell.SaveChanges();
    }
}

```

```

catch (System.Data.OptimisticConcurrencyException ex)
{
    // Zeige aktuellen Wert aus DB
    Console.WriteLine(DateTime.Now.ToLongTimeString() +
        ": Fehler: Ein anderer Benutzer hat bereits geändert!");
    WWings.GO.EF.Flug f2 = modell.Flug.Where(x => x.FlugNr == flugNr).SingleOrDefault();
    modell.Refresh(RefreshMode.StoreWins, f2); // WICHTIG !!!
    Console.WriteLine(DateTime.Now.ToLongTimeString() +
        ": Der andere Benutzer hat gespeichert: Freie Plätze=" + f2.FreiePlaetze);

    // Frage nach
    Console.WriteLine("Möchten Sie den Wert übernehmen (Taste '1') oder überschreiben (Taste '2')?");
    ConsoleKeyInfo key = Console.ReadKey();
    if (key.Key == ConsoleKey.D1)
    {
        Console.WriteLine("Sie haben gewählt: Option 1: übernehmen");
        modell.Refresh(RefreshMode.StoreWins, f);
    }
    else
    {
        Console.WriteLine("Sie haben gewählt: Option 2: überschreiben");
        modell.Refresh(RefreshMode.ClientWins, f);
        f.FreiePlaetze--;
        modell.SaveChanges();
    }
}

Console.WriteLine(DateTime.Now.ToLongTimeString() + ": Freie Plätze NACHHER: " + f.FreiePlaetze);
} // Ende using-Block -> Dispose() wird aufgerufen

```

**Listing 12.27** Beispiel für Änderungskonfliktlösung

**HINWEIS** In dem Code mag verwundern, wieso man schon beim Befüllen des Objekts `f2` mit der Neubabfrage der Datenbank `modell.Refresh(RefreshMode.StoreWins, f2)` angeben muss. Würde man das nicht machen, würde Entity Framework immer den Wert für `f2` verwenden, der schon in `f` liegt, denn Entity Framework erkennt, dass genau dieses Objekt schon im Zwischenspeicher des Kontextes vorliegt. Mit dem `Refresh()` muss man diesen Mechanismus aufheben.

## Steuerung der Codegenerierung durch austauschbare T4-Vorlagen

Mit dem durch den EF-Designer generierten Code waren viele Entwickler in Visual Studio 2008 nicht einverstanden, insbesondere nicht wegen der Ableitung aller Entitätsklassen von der Basisklasse `System.Data.Objects.DataClasses.EntityObject`. Zwar konnte man auch in Visual Studio 2008 die Generierung anpassen, indem man einen eigenen Codegenerator schreibt und diesen als *Custom Tool* in Visual Studio anstelle des *EntityModelCodeGenerator* einsetzt, aber diese Vorgehensweise war zum einen aufwändig und zum anderen konnte man damit auch nicht alle Restriktionen ausbessern, insbesondere nicht die Basisklassenanforderungen.



In EF 4 sind die Entitätsklassen nun wirklich beliebig. Es reichen also auch POCOs, Plain-Old-CLR-Objects, die keine Basisklasse oder Schnittstelle besitzen und aus ganz einfachen Attributen aufgebaut sind.

## T4-Vorlagen

Die Code-Generierung lässt sich in Visual Studio 2010 viel leichter durch so genannte T4-Vorlagen anpassen. T4 steht für *Text Template Transformation Toolkit* und ist eine Beschreibungssprache für die Umwandlung von einem Format in ein anderes. Sie ähnelt etwas der Syntax der früheren Active Server Pages (ASP) bzw. Inline-Code in ASP.NET: Es gibt statische Elemente und darin eingebettet in `<%...%>`-Platzhaltern Codeelemente, die die Ausgabe auf die jeweilige Quelle anpassen (siehe Abbildung 12.26). T4 gab es zwar schon in Visual Studio 2008, konnte da aber noch nicht ohne weiteres für EF genutzt werden.

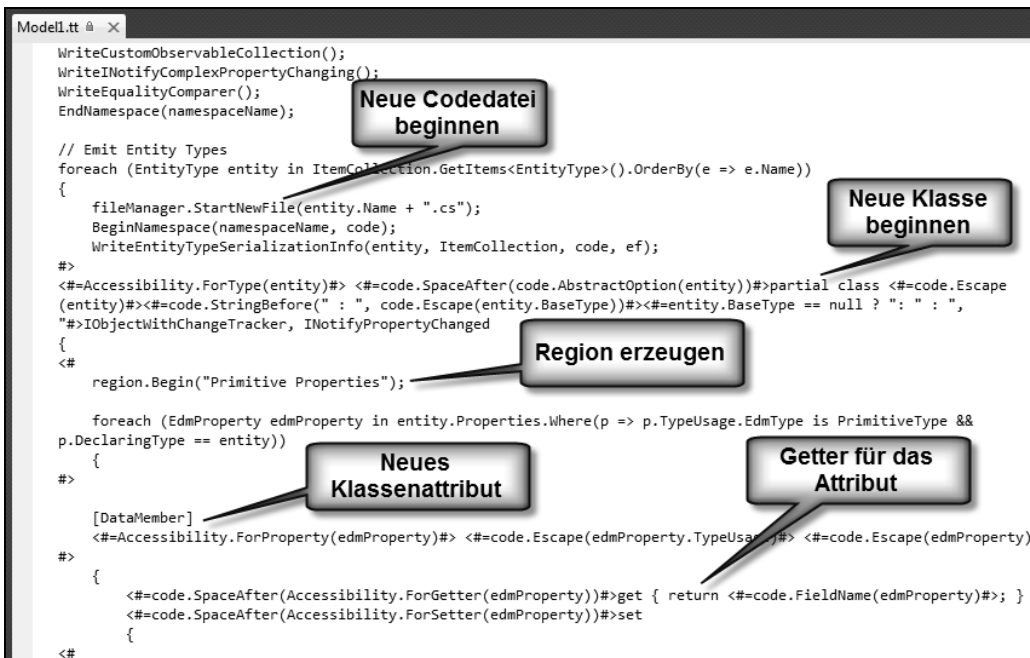


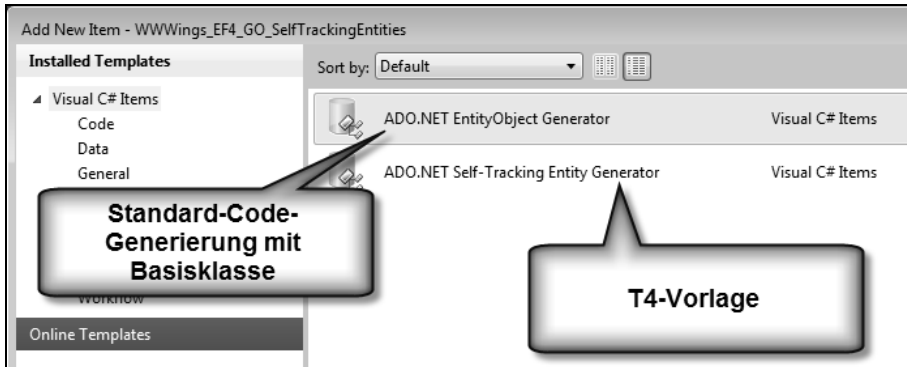
Abbildung 12.26 Ausschnitt aus einer T4-Vorlage für die Codegenerierung in Visual Studio 2010

**HINWEIS** Visual Studio 2010 bietet leider keinerlei Eingabeunterstützung (kein IntelliSense und noch nicht einmal Farbher-vorgebung) für T4-Vorlagen. Hierzu benötigt man ein Drittanbieterwerkzeug von Tangible Solutions (kostenfrei, <http://t4-editor.tangible-engineering.com>) oder Clarius (kostenpflichtig, <http://www.visualt4.com>).

## Auswahl der T4-Vorlage

Eine T4-Vorlage wählt man über die Funktion *Add New Artifact Generation Item* direkt im Kontextmenü des Designers aus. Zusammen mit Visual Studio 2010 liefert Microsoft neben der Standardcodegenerierung wie in Visual Studio 2008 nur eine einzige T4-Vorlage aus: *ADO.NET Self-Tracking Entity Generator* (Details

dazu siehe späterer Abschnitt). Zum Redaktionsschluss des Buchs findet man unter den zusätzlich herunterladbaren *Online Templates* auch noch einen *ADO.NET POCO Entity Generator* (siehe auch dazu eigener Abschnitt). Microsoft will in Zukunft weitere Vorlagen verbreiten.

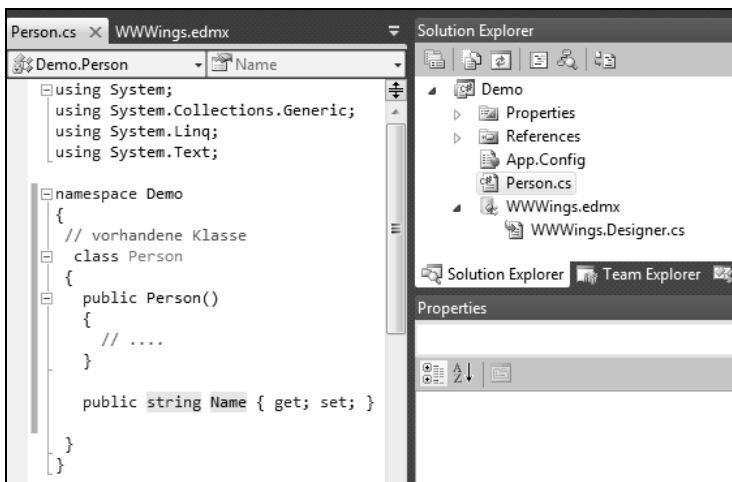


**Abbildung 12.27** In Visual Studio 2010 mitgelieferte T4-Vorlagen

#### **ACHTUNG** Visual Studio 2010 überschreibt ohne Warnung Codedateien

Vor einem schweren »Bug« in Visual Studio 2010 sei an dieser Stelle dringend gewarnt. Die T4-Vorlagengenerierung überschreibt eventuell schon vorhandene gleichnamige Dateien ohne Warnung!

Wenn eine Entität (z.B. Person) genauso heißt, wie eine bereits in dem gleichen Projekt auf der gleichen Orderebene vorhandene Codedatei (z.B. Person.cs), dann überschreibt die T4-Vorlagenauswertung die vorhandene *Person.cs*, ohne den Benutzer zu warnen. Auch ein Rückgängigmachen ist dann nicht mehr möglich!



**Abbildung 12.28** Situation vor dem Anwenden der T4-Vorlage

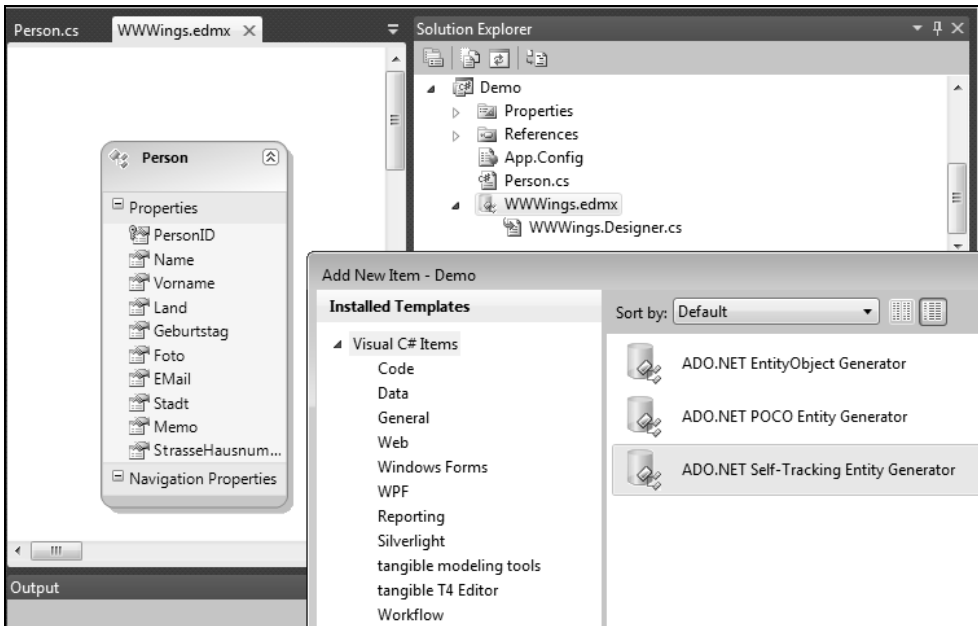


Abbildung 12.29 Auswahl der T4-Vorlage für das Modell mit der Entität *Person*

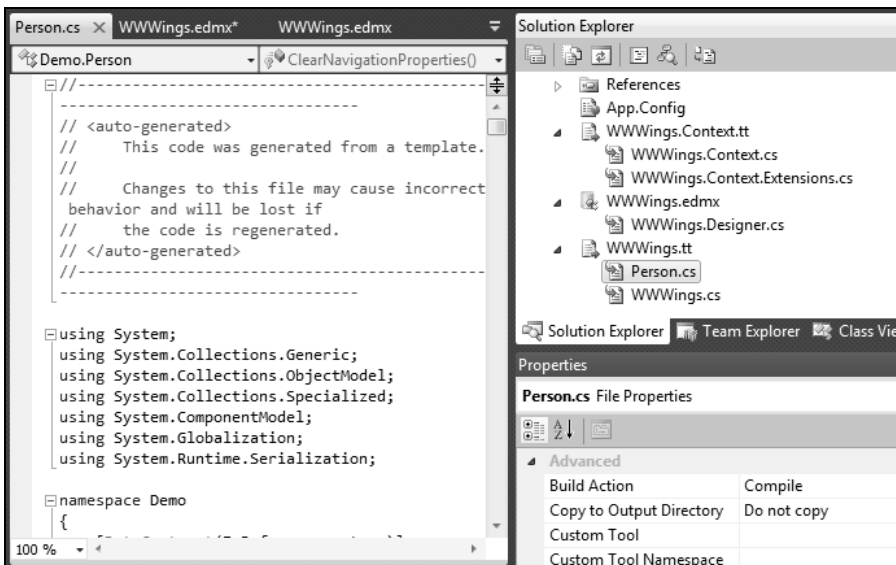


Abbildung 12.30 Die T4-Vorlage hat ohne Warnung die *Person.cs*-Datei mit der generierten gleichnamigen Datei überschrieben!

## Reaktivierung der Standardgenerierung

Nach dem Aktivieren einer T4-Vorlage enthält die direkt hinter der *.edmx*-Datei liegende *.Designer.cs*-Datei keinen Programmcode mehr, sondern nur noch einen Hinweis darauf, wie man die Standardcodegenerierung reaktiviert. Dazu muss man in der Designeroberfläche in den Eigenschaften des Designers die *Code Generation Strategy* wieder auf *Default* zurückstellen.

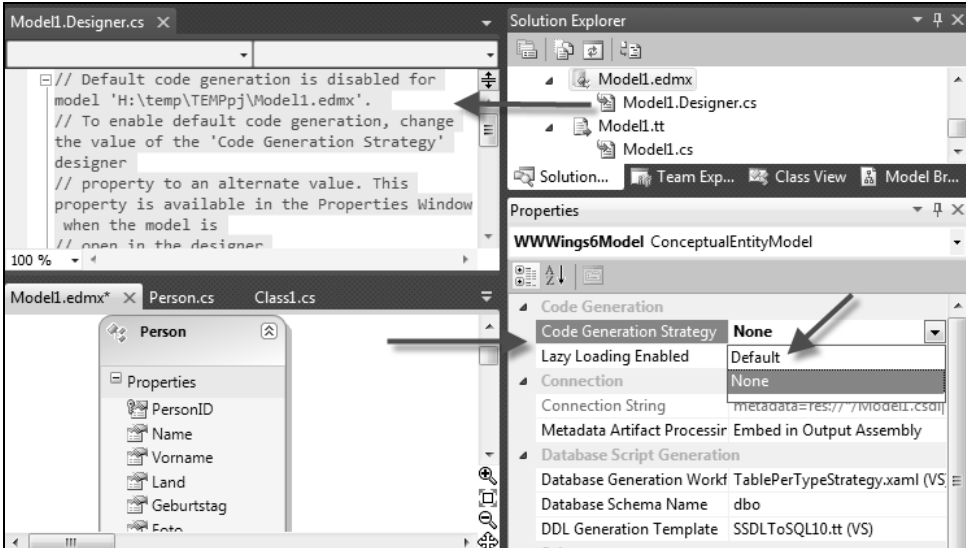


Abbildung 12.31 Reaktivierung der Standardcodegenerierung für Entity Framework-Modelle

### TIPP Anpassen der Standardgenerierung

Die Standardgenerierung erfolgt ohne eine sichtbare T4-Vorlage. Auf den ersten Blick scheint es so, als wenn man diesen Standardcode also nicht leicht anpassen könnte, sondern man hier wie in EF 1.0 auf die Anpassung des *EntityModelCodeGenerator* setzen müsste. Aber es gibt einen Trick: Microsoft liefert auch eine T4-Vorlage mit Visual Studio mit, die die gleiche Funktion erfüllt. Dazu muss man *Add Code Generation Item* und dann *ADO.NET Entity Object Generator* wählen. Durch wird dem Projekt die T4-Vorlage hinzugefügt, die die Standardcodegenerierung erledigt.

## Trennung von Kontext und Entitätsklassen

Der Einsatz von T4-Vorlagen erlaubt auch die Trennung der generierten Kontextklasse von den Entitätsklassen, denn eine T4-Vorlage kann beliebig viele Dateien erzeugen. Diese bisher in Visual Studio 2008 nicht mögliche Trennung ist sinnvoll, weil in mehrschichtigen Anwendungen die Clients die Entitätsklasse, nicht aber die Kontextklasse kennen müssen und sollten.

Anders als bei Typisierten DataSets kann man die Trennung leider in Visual Studio nicht vollautomatisch vollziehen. Nachstehend sind die Schritte für folgende Ausgangssituation beschrieben:

- Es gibt ein DLL-Projekt *BOT\_DAL* mit dem EF-Modell *Modell.edmx*
- Auf *Modell.edmx* wurde eine T4-Codegenerierungsvorlage angewendet, die aus zwei getrennten T4-Dateien (z.B. *Modell.tt* und *Modell.Context.tt*) besteht, eine für den Kontext und eine für die Geschäftsobjektclassen. Dies gilt z.B. für die Vorlagen *ADO.NET C# POCO Entity Generator* und *ADO.NET Self-Tracking Entity Generator*.

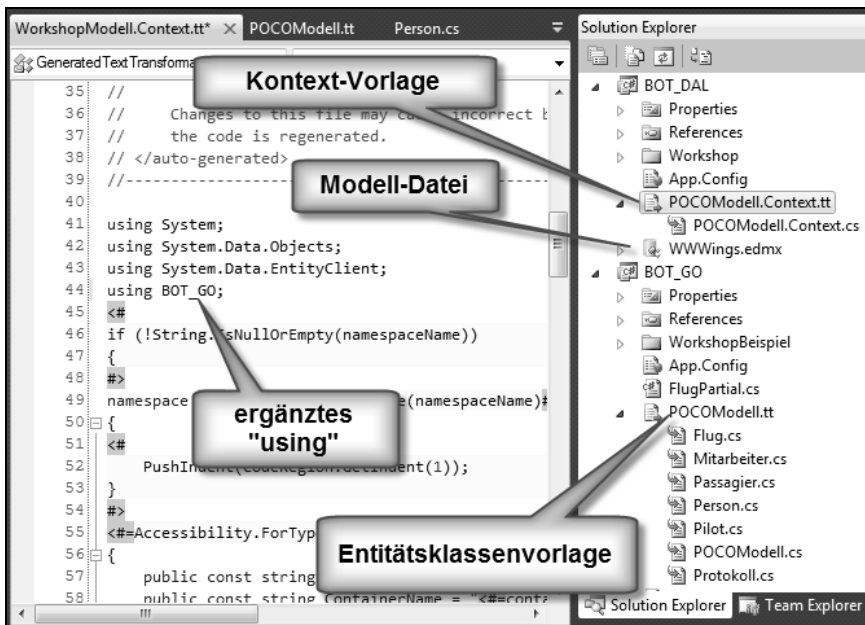
Um nun die Klassen zu trennen, sind folgende Schritte notwendig:

1. Anlegen eines weiteren DLL-Projekts *BOT\_BO*.
2. Referenzieren des Projekts *BOT\_BO* im Projekt *BOT\_DAL*.
3. Referenzieren der Assembly *System.Data.Entity.dll* im Projekt *BO*.
4. Verschieben der *Modell.tt* in das Projekt *BOT\_BO*.

Anpassen des Pfads zur *.edmx*-Datei in *Modell.tt*. Ein relativer Pfad ist ratsam!

z.B. `string inputFile = @"..\BOT_DAL\WWings.edmx";`

5. Ergänzen eines `using`-Befehls für den Namensraum, in dem sich die generierten Geschäftsobjektclassen befinden, hier also: `using BOT_GO;`



**Abbildung 12.32** Darstellung der Trennung von Objektcontext und Entitätsklassen in verschiedene Projekte

## Persistence Ignorance mit Plain Old CLR Objects (POCO)

Ein zentraler Kritikpunkt am Entity Framework 1.0 in .NET 3.5 SP1 war die Tatsache, dass die Entitätsklassen immer eine Basisklasse `System.Data.Objects.DataClasses.EntityObject` benötigten. Damit war die Anwendbarkeit des Entity Framework in einigen Szenarien (z.B. Transport der Objekte per Webservice an einen Nicht-.NET-Client, z.B. JavaScript) unmöglich und es mussten eigene Datentransferobjekte geschaffen werden.

In Entity Framework 4.0 werden nun auch Entitätsklassen unterstützt, die weder eine Basisklasse besitzen noch eine bestimmte Schnittstelle implementieren. Man spricht von POCOs, Plain-Old-CLR-Objects.

### POCO-Vorlage

Die grundsätzliche Fähigkeit für POCOs ist in Entity Framework 4.0 enthalten. Eine entsprechende T4-Vorlage für den Visual Studio 2010-Entity Framework-Designer liefert Microsoft aber kurioserweise nicht in Visual Studio 2010 mit, sondern separat über die Add-In-Download-Seite. Um die T4-Vorlage zu laden, rufen Sie *Add New Item/Online Templates/Database* auf. Dort finden Sie die Vorlage *ADO.NET C# POCO Entity Generator*. Eine entsprechende Variante für Visual Basic ist ebenfalls verfügbar.

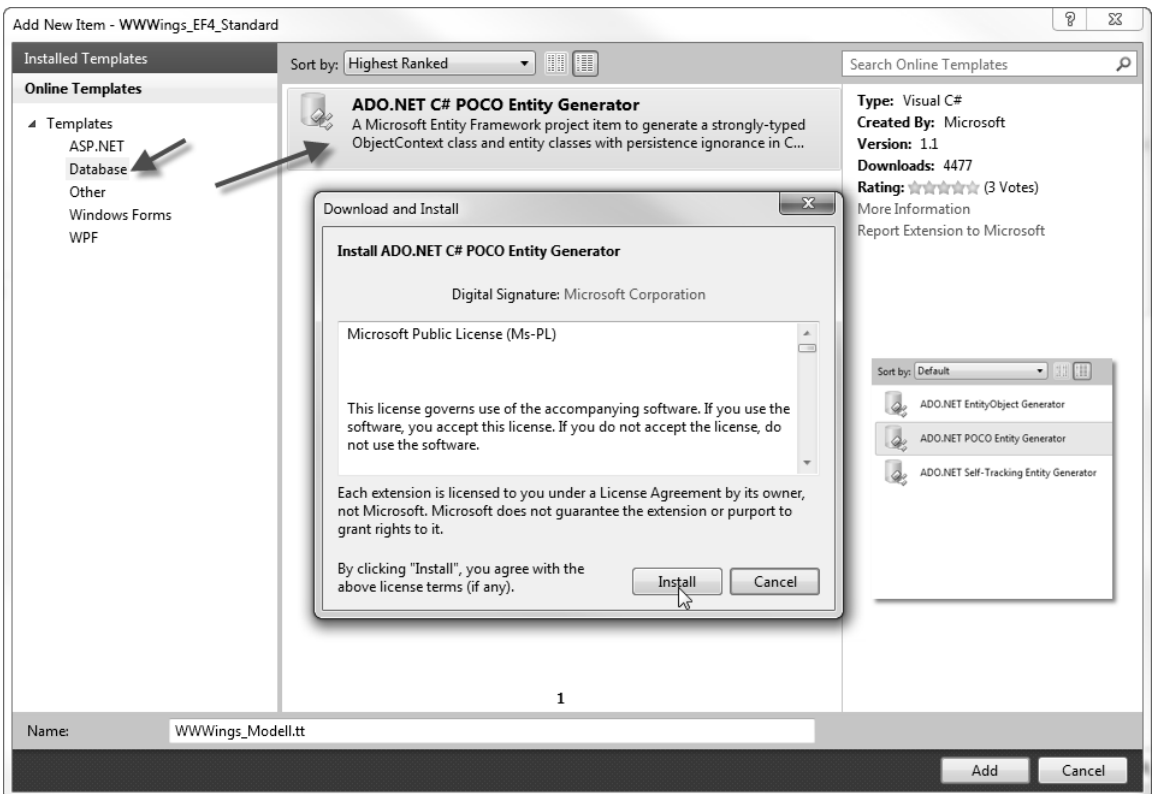


Abbildung 12.33 Nachträgliches Installieren der POCO-Vorlage

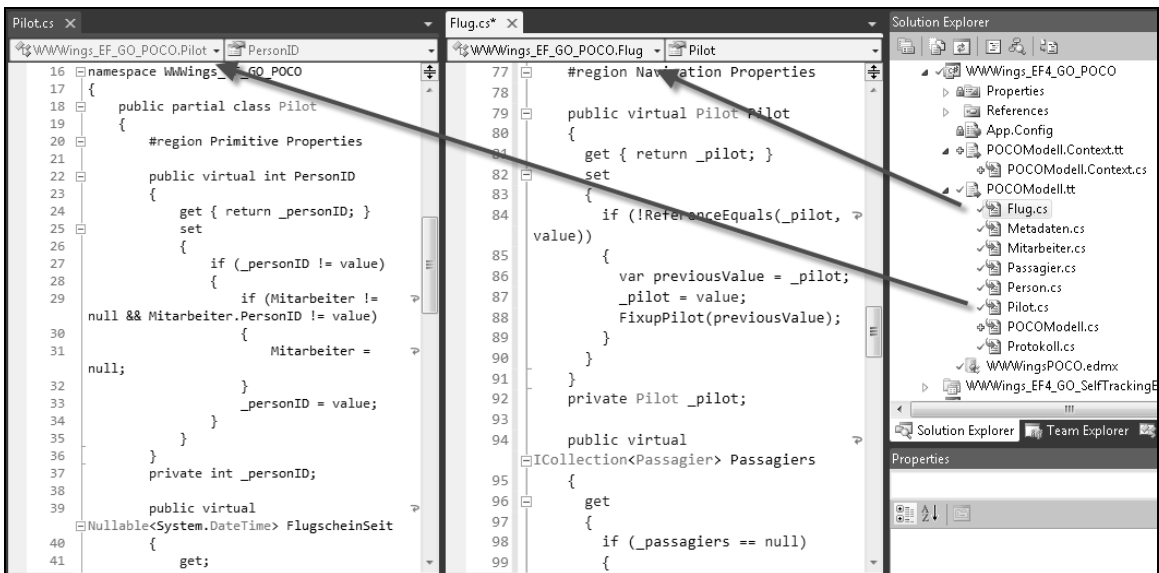
**ACHTUNG** Sie können nicht mehrere EF-Modelle mit der POCO-Vorlage im gleichen Namensraum haben, denn es wird jeweils die Klasse `FixupCollection` erzeugt, die dann zweimal im gleichen Namensraum existieren würde und dies führt zu einem Kompilierungsfehler!

## Anforderungen an die Entitätsklassen

Für die Grundfunktionalität des Mappings gibt es keine Anforderungen bezüglich der POCO-Entitätsklasse. Wenn man aber Funktionen wie Lazy Loading und die Änderungsverfolgung nutzen möchte, dann muss die Entitätsklasse doch einige Punkte beachten:

- Die Klasse muss `public` sein
- Die Klasse darf nicht `sealed/NotInheritable` oder `abstract/ MustInherit` sein
- Die Klasse muss einen parameterlosen Konstruktor haben, der `public` oder `protected` ist
- Die Navigationseigenschaften müssen `public` und `virtual/Overridable` sein
- Alle Klassenattribute müssen einen Getter haben, der `public` und `virtual/Overridable` ist (gilt nur für Lazy Loading)
- Navigationseigenschaften, die eine Menge repräsentieren, müssen von einem Mengentyp sein, der `ICollection<T>` implementiert (gilt nur für Change Tracking)

Alle diese Anforderungen werden mit der T4-Vorlage *ADO.NET C# POCO Entity Generator* (bzw. dem entsprechenden Visual Basic-Pendant) erfüllt. Um diese Vorlage auf ein Modell anzuwenden, wählt man *Add Code Generation Item* im Kontextmenü des Designer-Hintergrundes. Dadurch entstehen einzelne Code-Dateien für die Entitätsklassen und den Kontext. Die folgende Abbildung zeigt den Anfang der Klassendefinition der Klasse `Pilot` und den Bezug von `Flug` zu `Pilot` durch das Navigationsattribut `Pilot`.



**Abbildung 12.34** Ausschnitte aus dem erzeugten Code nach Anwenden der T4-POCO-Vorlage

## Beziehungsaktualisierung

Neben den o.g. Anforderungen ist durch einige Codezeilen ein weiterer Mechanismus implementiert, den Microsoft *Relationship Fixup* (Beziehungsaktualisierung) nennt. Für eine Beziehung zwischen zwei Entitätsklassen gibt es immer zwei Attribute in jeder der beiden in Beziehung stehenden Klassen. Wenn sich auf einer Seite etwas ändert, muss auch die andere Seite aktualisiert werden. Das folgende Listing zeigt einen Ausschnitt aus dem generierten Programmcode am Beispiel der 1:N-Beziehung zwischen Pilot und Flugzeug. Wenn für einen Flug ein Pilot geändert wird, dann muss der Flug aus der Liste der Flüge des alten Piloten entfernt werden und der Liste der Flüge des neuen Piloten zugewiesen werden.

```
public partial class Flug
{
...
    public virtual Pilot Pilot
    {
        get { return _pilot; }
        set
        {
            if (!ReferenceEquals(_pilot, value))
            {
                var previousValue = _pilot;
                _pilot = value;
                FixupPilot(previousValue);
            }
        }
    }
...
    private void FixupPilot(Pilot previousValue)
    {
        if (previousValue != null && previousValue.Flugs.Contains(this))
        {
            previousValue.Flugs.Remove(this);
        }

        if (Pilot != null)
        {
            if (!Pilot.Flugs.Contains(this))
            {
                Pilot.Flugs.Add(this);
            }
        }
    }
}
```

**Listing 12.28** Beziehungsaktualisierung in der POCO-Vorlage

Noch eine Implementierung, die die T4-POCO-Vorlage generiert, dient der Bereitstellung von Fremdschlüsseln zusätzlich zu Navigationsbeziehungen, sofern dies im Assistenten beim Anlegen des Modells gewählt wurde.



```

public virtual Nullable<int> Pilot_PersonID
{
    get { return _pilot_PersonID; }
    set
    {
        try
        {
            _settingFK = true;
            if (_pilot_PersonID != value)
            {
                if (Pilot != null && Pilot.PersonID != value)
                {
                    Pilot = null;
                }
                _pilot_PersonID = value;
            }
        }
        finally
        {
            _settingFK = false;
        }
    }
}

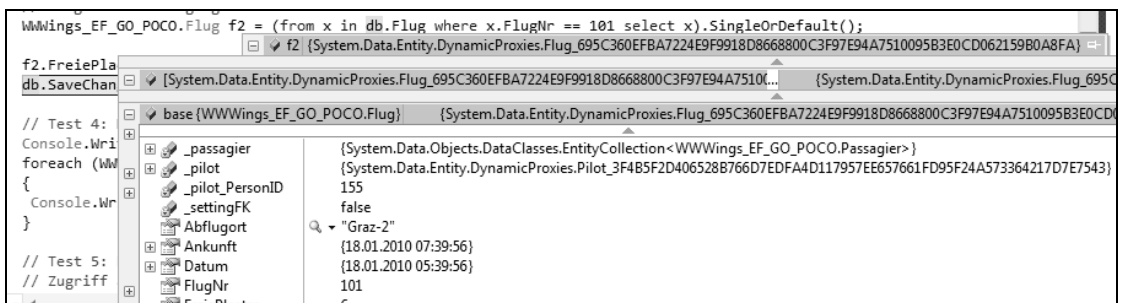
```

**Listing 12.29** Beziehungen über Fremdschlüssel in der POCO-Vorlage

## Änderungen gegenüber den Nicht-POCO-Entitätsklassen

Bei Verwendung einer POCO-Entitätsklasse muss Folgendes beachtet werden:

- Der Objektkontext muss `ObjectContextOptions.ProxyCreationEnabled = true` besitzen
- Neue Instanzen der Klasse sind nicht mit dem `New`-Operator, sondern mit der Methode `CreateObject<T>()` im Objektkontext zu erzeugen (notwendig für Änderungsverfolgung, damit .NET einen so genannten *Runtime Proxy* um das Objekt bauen kann, der Änderungen an den Objektkontext meldet)



**Abbildung 12.35** Man sieht im Debugger, dass ein Objekt einen *Runtime Proxy* besitzt

Listing 12.30 zeigt die Anwendung zahlreicher Entity Framework-Instrumente auf POCO-Klassen, insbesondere:

- Mengen lesen
- Einzelobjekte lesen
- Navigation mit Lazy Loading
- Objekte verändern
- Objekte löschen
- Objekte anlegen

#### HINWEIS

In Summe sieht man: Beim Umgang mit POCO-Entity Framework-Klassen aus der T4-POCO-Vorlage muss man nur Eines beachten: Objekte müssen mit `CreateObject()` erzeugt werden. Wenn der `New`-Operator verwendet wird, dann muss man vor `SaveChanges()` erst noch `DetectChanges()` aufrufen.

```

/// <summary>
/// Lese- und Schreiboperationen auf Entitätsklassen mit der POCO-Vorlage
/// </summary>
private static void POCO_Write()
{
    WWings_EF_GO_POCO.WWings6Entities db = new WWings_EF_GO_POCO.WWings6Entities();

    // Liste
    var ff = (from x in db.Flug where x.Abflugort == "Berlin"
              orderby x.Abflugort select x).Skip(5).Take(5);

    // TRACE
    Console.WriteLine(((System.Data.Objects.ObjectQuery)ff).ToTraceString());

    Console.WriteLine("Anzahl Flüge: " + ff.Count());
    foreach (Flug f in ff)
    {
        Console.WriteLine(f.FlugNr + ": " + f.Abflugort + " -> " + f.Zielort);
    }

    // Einzelobjekt // oder: .Include("Passagier.Person")
    Flug f2 = (from x in db.Flug
              where x.FlugNr == 101 select x).SingleOrDefault(); // ging noch nicht in EF1
    Console.WriteLine("Buchungen auf Flug " + f2.FlugNr + ": " + f2.Passagier.Count());
    foreach (Passagier ps in f2.Passagier) // Lazy Loading
    {
        Console.WriteLine(" - " + ps.Person.Name);
    }

    // Zugriff auf den ersten Passagier
    Passagier p = (from px in db.Passagier select px).First();
    Console.WriteLine("IsProxy(p):" + IsProxy(p));
    if (f2.Passagier.Contains(p)) // Löschen, wenn in dem Flug vorhanden
    {
        f2.Passagier.Remove(p);
        PrintChangeInfo(db);
    }
}

```

```

        db.SaveChanges();
        Console.WriteLine("Passagier entfernt!");
        Console.WriteLine("Buchungen auf " + f2.FlugNr + ": " + f2.Passagier.Count());
    }
    f2.Passagier.Add(p); // dem Flug wieder hinzufügen
    f2.FreiePlaetze--; // Freie Plätze verringern
    Console.WriteLine("Passagier hinzugefügt!");

    Console.WriteLine("Buchungen auf " + f2.FlugNr + ": " + f2.Passagier.Count());

    PrintChangeInfo(db);
    db.SaveChanges();

    f2 = (from x in db.Flug where x.FlugNr == 101 select x).SingleOrDefault();
    Console.WriteLine("Kontrolle: Buchungen auf " + f2.FlugNr + ": " + f2.Passagier.Count());

    // Löschen, wenn schon da durch vorherigen Testlauf!
    if (db.Flug.Where(x => x.FlugNr == 81).SingleOrDefault() != null)
    { db.DeleteObject(db.Flug.Where(x => x.FlugNr == 81).SingleOrDefault()); db.SaveChanges(); }

    // Neuen Flug anlegen
    Flug fneu = db.CreateObject<Flug>(); // statt new Flug() !!!
    fneu.FlugNr = 81;
    fneu.Abflugort = "Essen/Mülheim";
    fneu.Zielort = "Graz";
    fneu.Datum = DateTime.Now;
    db.Flug.AddObject(fneu);
    PrintChangeInfo(db);
    Console.WriteLine("IsProxy(fneu):" + IsProxy(fneu));
    db.SaveChanges();
    fneu.FreiePlaetze = 10;
    // db.DetectChanges(); // nur notwendig, wenn man new Flug() verwendet!
    PrintChangeInfo(db);
    Console.WriteLine("IsProxy(fneu):" + IsProxy(fneu));
    db.SaveChanges();
}

```

**Listing 12.30** Nutzung von POCO-Klassen

**ACHTUNG** Hinsichtlich der Ladeoptionen unterstützen POCO-Klassen Eager Loading und transparentes Lazy Loading. Da jedoch für Beziehungen nicht die Klassen `EntityCollection` und `EntityReference` verwendet werden, steht die `Load()`-Methode für explizites Nachladen nicht zur Verfügung. Hier muss man dann die neue Methode `LoadProperty()` der `ObjectContext`-Klasse verwenden.

# Änderungsverfolgung in verteilten Systemen (Self-Tracking Entities)

Die Unterstützung für verteilte Systeme (physikalische Schichtenarchitektur, alias »N-Tier-Szenarien«) im Entity Framework 1.0 war schwach. Zwar konnte man Entitätsobjekte auf vielfältige Weise serialisieren und somit über einen (Web)Service versenden, aber es gab zwei große Schwächen:

- Auf der Clientseite gab es für die vom Objektkontext losgelösten Objekte keine Änderungsverfolgung. Der Entwickler musste selbst im Code aufzeichnen, welche Objekte sich geändert haben, welche hinzugefügt und welche gelöscht wurden
- Auf der Serverseite war relativ viel Programmieraufwand notwendig um die Änderungen der losgelösten Objekte auf den Objektkontext zu übertragen

Im Entity Framework 4.0 gibt es zum einen einige Verbesserungen bei den Basismechanismen für die Serverseite. Zum anderen liefert Microsoft über eine eigene T4-Vorlage mit den Namen Self-Tracking Entities (STE) eine Lösung für Änderungsverfolgung (Change Tracking) bei Objekten, die über den Kontext geladen wurden, aber nun vom Kontext losgelöst sind, weil sie die Prozessgrenze verlassen haben.

---

**ACHTUNG** Vorweg sollen einige Einschränkungen benannt sein, damit niemand falsche Erwartungen bekommt:

- Lazy Loading wird bei STE nicht unterstützt, weder transparent noch explizit. Eager Loading mit `Include()` ist aber möglich.
  - STE hat einen eigenen Change Tracking Mechanismus. Die Informationen, die der `ObjectStateManager` liefert, sind daher nicht aussagekräftig.
  - Die Methode `AcceptAllChanges()` wirkt nicht bei STE
  - Die Entitätsklassen im STE-Modell sind mit `[DataContract]`, aber nicht mit `[Serializable]` ausgezeichnet, daher ist die Serialisierung mit den alten Serialisierern wie `BinaryFormatter` und `SoapFormatter` zunächst nicht möglich. Dies könnte man in der T4-Vorlage aber selbst nachbessern.
- 

## Anwendung der T4-Vorlage

Zur Anwendung der T4-Vorlage für die Self-Tracking Entities wählt man auf dem Hintergrund des Entity Framework-Designers in Visual Studio im Kontextmenü *Add Code Generation Item* und dann *ADO.NET Self-Tracking Entity Generator*.

Wie bei der POCO-Vorlage auch entstehen zwei Blöcke von Codedateien:

- `xy.tt` erzeugt für jede Entitätsklasse eine Klassendatei sowie Hilfsklassen für die Änderungsverfolgung (ca. 400 Zeilen Code). Dieser Programmcode wird auch in einem Client benötigt, der serialisierte/deserialisierte Objekte verwendet. Der Client muss also die Assembly, die diesen Programmcode enthält, referenzieren.
- `xy.Context.tt` erzeugt die von `ObjectContext` abgeleitete Klasse und zahlreiche Hilfsfunktionen in einer getrennten Klassendatei (ca. 1200 Zeilen Code). Dieser Programmcode wird im Client nicht benötigt. In der architektonisch besten Lösung würde man diesen Programmcode daher in eine andere Assembly verschieben, die nicht vom Client referenziert wird.

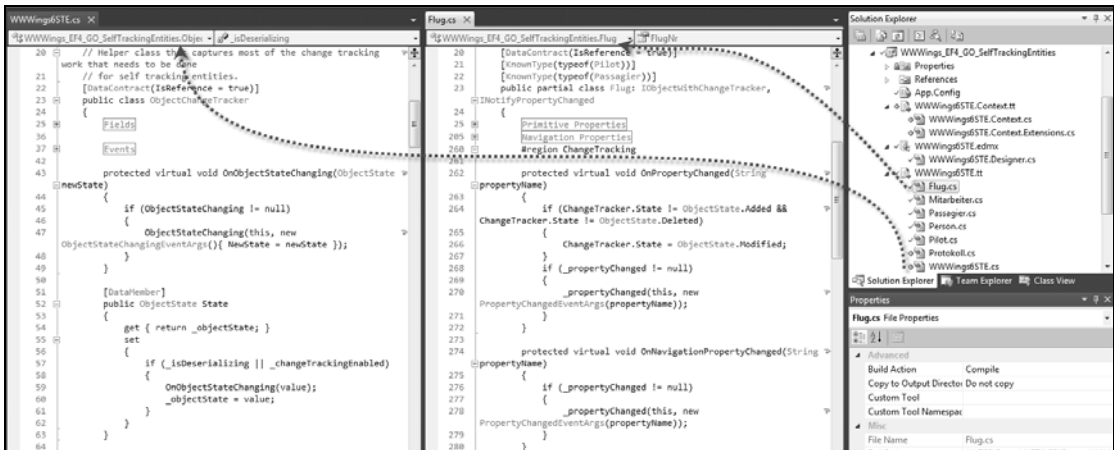


Abbildung 12.36 Kleine Ausschnitte aus den generierten Klassen

## Mechanismen der Self-Tracking Entities

Dieser Abschnitt gibt einen Überblick über die wesentlichen Mechanismen der Self-Tracking Entities.

Jede Entitätsklasse besitzt ein Attribut `ChangeTracker` vom Typ `ObjectChangeTracker`.

Die Klasse `ObjectChangeTracker` stammt nicht aus dem Entity Framework selbst, sondern wird durch den generierten Programmcode erzeugt.

Die Entitätsklasse realisiert die Schnittstelle `INotifyPropertyChanged`. Jedes einzelne Attribut der Klasse löst im Setter `OnPropertyChanging()` aus. Die Implementierung von `OnPropertyChanging()` setzt das Attribut `State` im Unterobjekt `ChangeTracker` voraus.

Über Erweiterungsmethoden (realisiert in der ebenfalls generierten Klasse `ObjectWithChangeTrackerExtensions`) erhält jede Entitätsklasse weitere Methoden:

- `StartTracking()` setzt `ChangeTrackingEnabled` des Objekts `ChangeTracker` auf `true`
- `StopTracking()` bewirkt das Gegenteil
- Markieren eines Objekts für einen bestimmten Zustand: `MarkAsDeleted()`, `MarkAsAdded()`, `MarkAsModified()` und `MarkAsUnchanged()`. Dadurch wird jeweils der `State` des `ChangeTracker` gesetzt. Außerdem wird `ChangeTrackingEnabled` des Objekts `ChangeTracker` auf `true` gesetzt.
- `AcceptChanges()` setzt die Liste der geänderten Attribute des Objekts zurück und setzt den Zustand des Objekts auf *unmodified*
- Über die Erweiterungsmethoden der Klasse `SelfTrackingEntitiesContextExtensions` werden zudem die Klassen `ObjectContext` und `ObjectSet<T>` erweitert um die Methode `ApplyChanges()`. Diese recht komplexe Methode informiert den Objektkontext über alle Änderungen in dem jeweils übergebenen Entitätsobjekt. Intern verwendet die Methode u.a. `ObjectStateManager.ChangeObjectState()` sowie `ChangeRelationshipState()` sowie `GetUpdatableOriginalValues()`. Dies sind im Entity Framework hinterlegte Verbesserungen für N-Tier-Anwendungen.

**HINWEIS** `ChangeObjectState()` und `ChangeRelationshipState()` erlauben den Zustand, den der `ObjectStateManager` von einem Entitätsobjekt verwaltet, zu beeinflussen (z.B. dem `ObjectStateManager` mitteilen, dass ein Objekt als »modified« gelten soll). In Entity Framework 1 war dies nicht möglich. `GetUpdatableOriginalValues()` liefert eine Liste der Originalwerte mit der Möglichkeit, diese zu ändern.

Es gibt weitere neue Basismechanismen im Entity Framework für die Handhabung von losgelösten Objekten, z.B. `ApplyCurrentValues()` und `ApplyOriginalValues()`. Um Änderungen in einem vom Kontext losgelösten Objekt auf das gleiche Objekt im Kontext zu übertragen, musste man bisher `db2.ApplyPropertyChanges("Flug", FlugNeu)` verwenden. Jetzt geht es etwas komfortabler und typsicherer mit `db2.Flug.ApplyCurrentValues(FlugNeu)`. Alternativ kann man in einem Szenario, in dem der Kontext die neuen Werte enthält und die früheren Werte separat vorliegen, `ApplyOriginalValues()` anwenden, damit der Kontext weiß, was er speichern muss. Diese Basismechanismen muss man aber bei den Self-Tracking Entities nicht anwenden. Sie sind hier nur der Vollständigkeit halber erwähnt.

## Beispielanwendung »WPF-DataGrid mit Self-Tracking Entities«

Zur Veranschaulichung des Praxiseinsatzes der Self-Tracking Entities soll ein Datengitter (DataGrid) in einer Windows Presentation Foundation (WPF)-Anwendung dienen. Die Anwendung, die ein Teil des World Wide Wings-Beispiels ist (siehe Ordner */Desktop/WPF*), zeigt Flüge, wobei ein Filtern nach Abflugort möglich ist. Der Benutzer kann in dem Datengitter die Daten verändern, neue Datensätze anfügen und bestehende Datensätze löschen.

Dabei werden bewusst drei verschiedene Architekturmodelle (und deren Verbindung in einer einzigen Anwendung) gezeigt:

- **DBDirekt** Direkter Zugriff aus der Anwendung auf die Datenbank (2-Layer/2-Tier)
- **BL** Zugriff über die dedizierte Geschäftslogikschicht auf die Datenbank (3-Layer/2-Tier)
- **Service** Zugriff über einen Webservice auf einen Application Server, der auf die Datenbank zugreift (n-Layer/3-Tier)

Die drei verschiedenen Zugriffswege wählt man über ein Auswahlmennü.

**HINWEIS** Die »Service«-Variante arbeitet dabei mit so genannten *Shared Contracts*, d.h. der Client besitzt eine Komponentenreferenz auf die Bibliothek, in der die Entitätsklassen realisiert sind. Dies ist sowieso für die ersten beiden Szenarien notwendig. Es werden also keine Proxyklassen für die Entitätsklassen benötigt.

Der Einsatz von *Shared Contracts* vereinfacht die Umschaltung zwischen den drei Architekturmodellen im Client erheblich. Mit generierten Proxyklassen wäre mehr Aufwand notwendig. Generierte Proxyklassen wären nur dann wirklich zwingend notwendig, wenn der Client nicht .NET wäre. Dann müsste man aber auch in der anderen Programmiersprache den ChangeTracking-Mechanismus selbst realisieren, was ein umfangreiches Unterfangen ist. Dies soll in diesem Buch aber aus Platzgründen ausgeklammert sein.

**HINWEIS** Das Ergebnis sei vorweggenommen: Leider arbeitet das Entity Framework noch nicht optimal mit der Benutzeroberfläche zusammen. Lästige Aufgaben für den Entwickler gibt es insbesondere mit dem Anfügen und dem Löschen von Objekten.

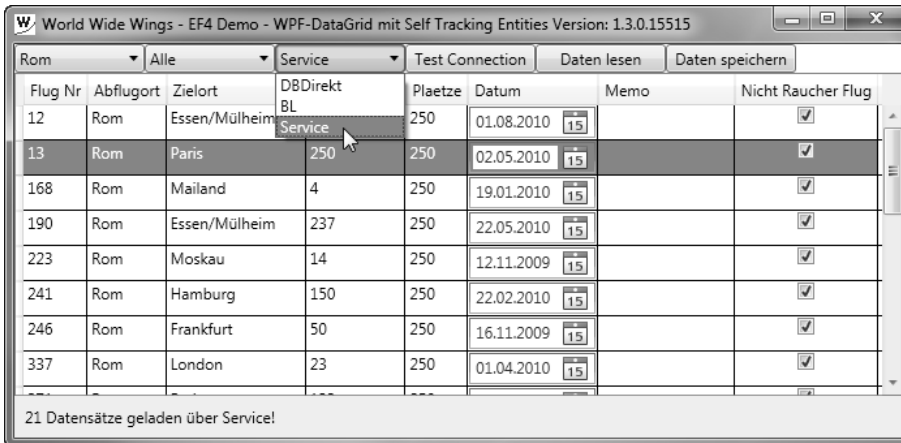


Abbildung 12.37 Auswahl des Ladeweges

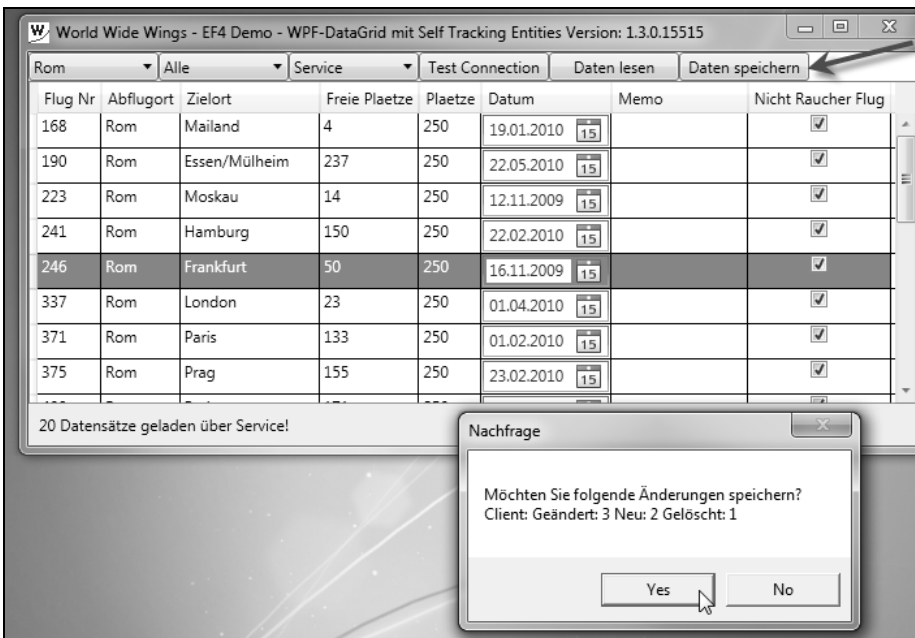


Abbildung 12.38 Speichern der Änderungen

## Implementierung der Geschäftslogik

Das folgende Listing zeigt die Implementierung der Geschäftslogik. Der Kontext wird für jede einzelne Operation neu erzeugt. Im Architekturszenario *BL* könnte die Geschäftslogik natürlich alternativ eine Instanz des Kontextes behalten. Dann wäre diese Geschäftslogik aber nicht mehr für den zustandslosen Webservice nutzbar. Daher ist die Geschäftslogik ebenfalls zustandslos realisiert.

Interessant (und wichtig!) ist nur der Aufruf von `ApplyChanges()` für jedes einzelne Flug-Objekt, das beim Speichern übergeben wird. `ApplyChanges()` wertet die Change Tracker-Informationen in jedem Objekt aus und übergibt sie dem Objektkontext, sodass dieser bei `SaveChanges()` weiß, was er speichern muss.

```

/// <summary>
/// Geschäftslogik für Beispiel "WPF-DataGrid mit Self-Tracking Entities"
/// </summary>
public class BLManager
{
    /// <summary>
    /// Flüge laden
    /// </summary>
    public List<WWings_EF4_GO_SelfTrackingEntities.Flug> GetFluege(string Ort)
    {
        // Laden und als Liste zurückgeben
        using (WWings_EF4_GO_SelfTrackingEntities.WWings6STE db =
            new WWings_EF4_GO_SelfTrackingEntities.WWings6STE())
        {
            return (from f in db.Flug where f.Abflugort == Ort select f).ToList();
        }
    }

    /// <summary>
    /// Flüge speichern
    /// </summary>
    public string SaveFluege(List<WWings_EF4_GO_SelfTrackingEntities.Flug> Fluege)
    {
        using (WWings_EF4_GO_SelfTrackingEntities.WWings6STE db =
            new WWings_EF4_GO_SelfTrackingEntities.WWings6STE())
        {
            // Änderungen für jeden einzelnen Flug übernehmen
            foreach (Flug f in Fluege)
            {
                db.Flug.ApplyChanges(f);

                // Konsistenzprüfung :-)
                if (f.Datum < new DateTime(1900, 1, 1)) f.Datum = DateTime.Now;
            }

            // Statistik der Änderungen zusammenstellen
            string Rueckgabe = "";
            Rueckgabe += "Geändert: " +
                db.ObjectStateManager.GetObjectStateEntries(System.Data.EntityState.Modified).Count();
            Rueckgabe += " Neu: " +
                db.ObjectStateManager.GetObjectStateEntries(System.Data.EntityState.Added).Count();
            Rueckgabe += " Gelöscht: " +
                db.ObjectStateManager.GetObjectStateEntries(System.Data.EntityState.Deleted).Count();

            // Änderungen speichern
            db.SaveChanges();
        }
    }
}

```



```
        // Statistik der Änderungen zurückgeben
        return Rueckgabe;
    }
}
```

**Listing 12.31** Geschäftslogik

## Implementierung des Webservices

Der Webservice ist nur eine Fassade für die oben beschriebene Geschäftslogik. Der Webservice ist mit der Windows Communication Foundation (WCF) realisiert.

**HINWEIS** Auf die Darstellung der übrigen Bestandteile der WCF-Webservice-Realisierung (Schnittstelle, Konfigurationsdatei, Client-Proxy) wird hier aus Platzgründen verzichtet, da diese einerseits nicht neu in .NET 4.0 sind und andererseits mit dem Datenzugriff selbst nichts zu tun haben.

```
/// <summary>
/// Service ist nur Fassade für Geschäftslogik
/// </summary>
public class FlugService : IFlugService
{
    public List<WWings_EF4_GO_SelfTrackingEntities.Flug> GetFluege(string Ort)
    {
        return new WWings_EF4_BL.BLManager().GetFluege(Ort);
    }

    public string SaveFluege(List<WWings_EF4_GO_SelfTrackingEntities.Flug> Fluege)
    {
        return new WWings_EF4_BL.BLManager().SaveFluege(Fluege);
    }

    public string Hello(string name)
    {
        return "Hallo " + name + "!";
    }
}
```

**Listing 12.32** Webservice-Implementierung

## Variablen

In der Coderegion *Variablen* werden zunächst für jeden der drei Zugriffswege die entsprechenden Zugangsobjekte instanziiert. Dies ist im Fall *DBDirekt* der Entity Framework-Kontext, im Fall *BL* die Geschäftslogikklasse und bei *Service* der Client-Proxy des Webservice.

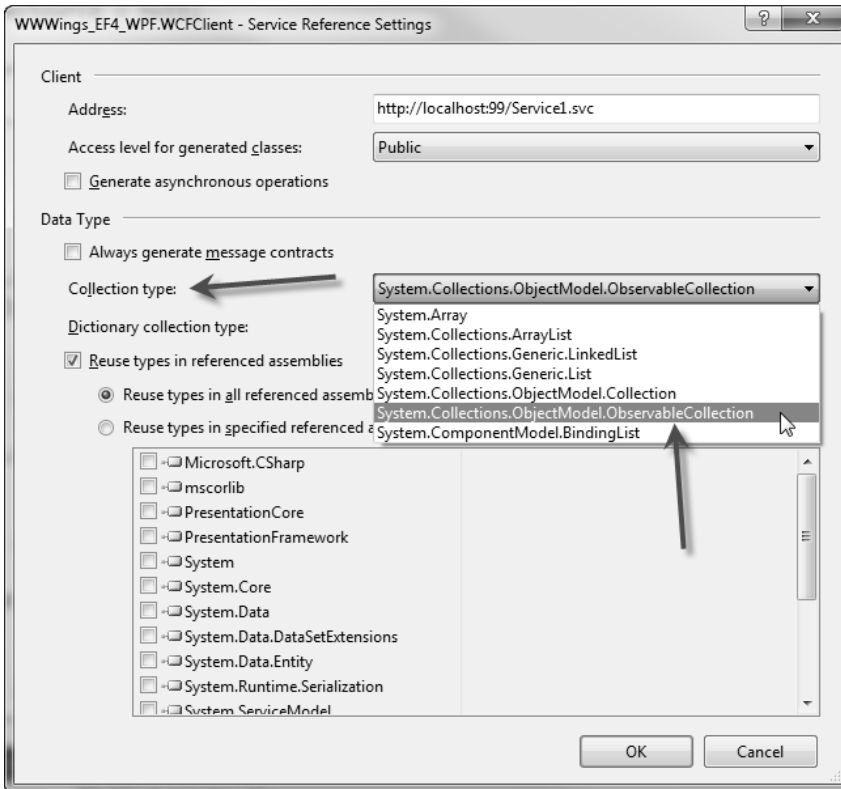
Für die Speicherung der geladenen Flüge gibt es zwei – statt wie man erwarten könnte, nur eine Liste. Dies ist notwendig für das Löschen von Objekten. Wenn eine Liste von Entity Framework-Objekten losgelöst ist vom Kontext, muss man später dem Kontext bei der Zurückführung der Objekte auch diejenigen Objekte erneut übermitteln, die gelöscht wurden. Sie sind als gelöscht zu markieren und daher weiß der Kontext, was er löschen soll. Das Datengitter arbeitet aber nach einem anderen Prinzip: Es markiert die Objekte nicht als gelöscht, sondern löscht sie wirklich aus der gebundenen Liste. Dann kann man später dieses Objekt nicht zum Kontext zurückführen. Daher muss die Anwendung zusätzlich zu der gebundenen Liste die Originalliste verwalten.

```
#region Variablen
// DBDirekt
WWings6STE DB = new WWings6STE();
// BL
WWings_EF4_BL.BLManager BL = new WWings_EF4_BL.BLManager();
// Service
WCFCClient. FlugServiceClient
Dienst = new WWings_EF4_WPF.WCFCClient. FlugServiceClient
();

// Liste der Flüge - an das Datengitter gebunden
ObservableCollection<Flug> fluegeAktuelleAnsicht;
// Kopie der Flüge für Feststellung der gelöschten Flüge
ObservableCollection<Flug> fluegeKomplett;
#endregion
```

## Laden der Objekte

Das folgende Listing zeigt das Laden der Flüge. Die Menge der Flugobjekte wird in allen drei Fällen als `ObservableCollection<Flug>` an das Datengitter gebunden, denn dadurch ist es möglich, im Fall des Anfügens und Löschens von Objekten ein Ereignis zu erhalten. Im Fall *DBDirekt* und *BL* muss die gelieferte `List<Flug>` in `ObservableCollection<Flug>` überführt werden. Der Client-Proxy für den Webservice wurde bereits beim Anlegen unter *Advanced Settings* so konfiguriert, dass er die Menge als `ObservableCollection<Flug>` liefert.



**Abbildung 12.39** Konfigurieren des Typs für die Objektmengen im Client-Proxy für den WCF-Service

```

/// <summary>
/// Laden der Flüge
/// </summary>
private void C_Lesen_Click(object sender, RoutedEventArgs e)
{
    if (this.C_Modus.Text == "") { SetStatus("Keine Datenzugriffsweg gewählt!"); return; }
    // Anzeige löschen
    this.flugDataGrid.ItemsSource = null;
    // Ausgewählter Ort
    string Ort = this.C_Orte.Text.ToString();
    // Statusanzeige
    SetStatus("Lade über " + this.C_Modus.Text + "...");

    // DBDirekt
    if (this.C_Modus.Text == "DBDirekt")
    {
        fluegeAktuelleAnsicht = new System.Collections.ObjectModel.ObservableCollection<Flug>(
            from f in DB.Flug where f.Abflugort == Ort select f);
    }
    // BL
    if (this.C_Modus.Text == "BL")
    {

```

```

        fluegeAktuelleAnsicht =
            new System.Collections.ObjectModel.ObservableCollection<Flug>(BL.GetFluege(Ort));
    }
    // Service
    if (this.C_Modus.Text == "Service")
    {
        fluegeAktuelleAnsicht = Dienst.GetFluege(Ort);
    }

    // Registrieren auf Änderungsereignis der Liste (für Anfügen und Löschen von Objekten!)
    fluegeAktuelleAnsicht.CollectionChanged +=
        new System.Collections.Specialized.NotifyCollectionChangedEventHandler(
            fluegeAktuelleAnsicht_CollectionChanged);

    // Aktivieren der Änderungsverfolgung (für DBDirekt und BL wichtig!)
    foreach (Flug f in fluegeAktuelleAnsicht)
    { f.StartTracking(); }

    // Kopie anlegen für Merken der gelöschten Objekte
    fluegeKomplett = new ObservableCollection<Flug>(fluegeAktuelleAnsicht);

    // Datenbindung an Datengitter
    this.flugDataGrid.ItemsSource = fluegeAktuelleAnsicht;

    // Status setzen
    SetStatus((this.flugDataGrid.ItemsSource as IEnumerable<Flug>).Count() +
        " Datensätze geladen über " + this.C_Modus.Text + "!");
}

```

**Listing 12.33** Laden der Flüge mit Fallunterscheidung für die drei Architekturszenarien

## Sonderbehandlung für angefügte und gelöschte Objekte

Lästige Aufgaben gibt es insbesondere mit dem Anfügen und dem Löschen von Objekten:

- Wenn ein Objekt angefügt werden soll, reicht es nicht, das Objekt der Liste hinzuzufügen. Man muss es zusätzlich auch dem Kontext mit der `Add()`-Methode als neu anmelden.
- Wenn ein Objekt gelöscht wurde, dann erwartet das Entity Framework, dass man das Objekt auch mit `DeleteObject()` explizit aus dem Kontext löscht

Das WPF-Datengitter kennt diese Mechanismen nicht, für andere Datengitter gilt dies auch. Microsoft hätte es den Entwicklern einfach machen können mit der Implementierung einer speziellen Objektmengenklasse, die `INotifyCollectionChanged` implementiert im Sinne der o.g. Anforderungen des Entity Framework. Leider hat Microsoft diese Unterstützung auf eine kommende Version verschoben.

Der Entwickler muss daher selbst für die Benachrichtigung des Kontextes sorgen. Dabei ist noch zu unterscheiden, ob die Objekte an den Kontext angefügt sind (Attached) oder losgelöst sind (Detached). Angefügt sind die Objekte nur im Szenario *DirektDB*. In diesem Fall muss im Ereignis `CollectionChanged` der `ObservableCollection` `DB.DeleteObject(f)` bzw. `DB.Flug.Add(f)` aufgerufen werden.

In den anderen Fällen ist `MarkAsDeleted()` für die gelöschten Objekte aufzurufen. Dieser Aufruf markiert das zu löschende Objekt. Es wird dann durch das Datengitter aus der Menge `fluegeAktuelleAnsicht` tatsächlich gelöscht, bleibt aber in `fluegeKomplett` mit dieser Markierung bestehen (Hinweis: Beide Listen enthalten jeweils nur Verweise auf dieselben Objekte). Das Pendant `MarkAsAdded()` muss man nicht explizit aufrufen. Das Entity Framework erkennt selbst, dass es sich um ein neues Objekt handelt. Da in dem Beispiel aber die Implementierung mit der Originalliste `fluegeKomplett` gewählt wurde, sollte man diese auch um das Objekt ergänzen. Das vereinfacht nachher die Speicherung, denn man muss sich um die Menge `fluegeAktuelleAnsicht` gar nicht mehr kümmern, sondern kann die Menge `fluegeKomplett` heranziehen.

**HINWEIS** Die Verwaltung der Originalliste ist nicht die einzige Implementierungsalternative. Man könnte zum Beispiel auch nur eine Liste der gelöschten Elemente separat verwalten, an die ein Objekt immer erst beim Löschen angefügt wird. Beim Zurückspeichern könnte man diese Liste der gelöschten Objekte dann entweder separat zum Kontext übermitteln oder – wenn man den zusätzlichen Rundgang vermeiden will – die Liste der gelöschten Objekte in einer neuen Liste mit der Hauptliste verbinden. Weniger eigener Codierungsaufwand ist das aber nicht.

```
/// <summary>
/// Sonderbehandlung für angefügte und gelöschte Objekte
/// </summary>
void fluegeAktuelleAnsicht_CollectionChanged(object sender,
                                             System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
    if (e.Action == System.Collections.Specialized.NotifyCollectionChangedAction.Remove)
    {
        foreach (Flug f in e.OldItems)
        { // Als zu löschen markieren (wirkt auf beide Listen)
            f.MarkAsDeleted();
            // Für direkten DB-Zugriff: Löschanweisung an Kontext senden
            if (this.C_Modus.Text == "DBDirekt") DB.DeleteObject(f);
        }
    }

    if (e.Action == System.Collections.Specialized.NotifyCollectionChangedAction.Add)
    {
        foreach (Flug f in e.NewItems)
        { // Auch der Originalliste anfügen, damit diese alle Änderungen enthält
            fluegeKomplett.Add(f);
            // Für direkten DB-Zugriff: An Kontext anfügen
            if (this.C_Modus.Text == "DBDirekt") DB.Flug.AddObject(f);
        }
    }
}
```

**Listing 12.34** Implementierung der notwendigen Sonderbehandlung für Löschen und Anfügen

## Speichern

Vor dem Speichern zeigt die Anwendung dem Benutzer an, wie viele Änderungen verzeichnet wurden. Dies kann sehr einfach für alle drei Fälle auf Basis der Menge `fluegeKomplett` erfolgen, denn diese Liste besitzt alle Elemente, sowohl die geänderten als auch die gelöschten und hinzugefügten. Eine Methode `GetChanges()` wie im `DataSet` gibt es nicht. Den `ObjectStateManager` des Entity Framework-Objektcontextes kann man

nicht verwenden, denn auf diesen hat man nur im Architekturmodell *DBDirekt* Zugang. Die Self-Tracking Entities besitzen aber jeweils ein Unterobjekt *ChangeTracker*, das den Zustand verwaltet. Durch LINQ to Objects-Abfragen kann man einfach die Objektmenge durchsuchen und diejenigen Objekte herausfiltern, die in *ChangeTracker.State* einen bestimmten Zustand haben.

Danach folgt die Fallunterscheidung für die drei Architekturmodelle:

- Bei *DBDirekt* kann man direkt *SaveChanges()* auf dem Kontext aufrufen
- Bei *BL* ruft man die *SaveFluege()*-Methode unter Angabe der *fluegeKomplett*-Menge auf
- Bei *Service* ruft man die *SaveFluege()*-Operation des Webservice auf. Hier empfiehlt es sich, diejenigen Objekte vorher auszufiltern, die sich nicht verändert haben, denn damit reduziert man die Netzwerklast erheblich. Auch dazu kann man wieder LINQ to Objects einsetzen, denn ein *GetChanges()* wie im Data-Set gibt es ja nicht. Natürlich kann man diesen Mechanismus auch im Fall *BL* anwenden; dort hat er aber kaum einen Effekt, denn die *SaveFluege()*-Methode befindet sich ja im gleichen Adressraum.

Wichtig sind dann zum Abschluss noch zwei Schritte:

- Man muss auf jedem Objekt *AcceptChanges()* aufrufen, um den Zustand des Objekts auf *unmodified* zurückzusetzen
- Man muss die Liste *fluegeKomplett* auf den Zustand der *fluegeAktuelleAnsicht* setzen, damit die gelöschten Objekte aus *fluegeKomplett* verschwinden und nicht beim nächsten Speichern versucht würde, sie erneut zu löschen

```

/// <summary>
/// Speichern der geänderten Flüge
/// </summary>
private void C_Speichern_Click(object sender, RoutedEventArgs e)
{
    if (fluegeKomplett == null) return; // keine Daten geladen!

    // Änderungen anzeigen und nachfragen
    if (MessageBox.Show("Möchten Sie folgende Änderungen speichern?\n" +
        String.Format("Client: Geändert: {0} Neu: {1} Gelöscht: {2}",
            (from f in fluege where f.ChangeTracker.State == ObjectState.Modified
            select f).Count(),
            (from f in fluegeKomplett where f.ChangeTracker.State == ObjectState.Added
            select f).Count(),
            (from f in fluegeKomplett where f.ChangeTracker.State == ObjectState.Deleted
            select f).Count()),
        "Nachfrage", MessageBoxButton.YesNo) == MessageBoxResult.No) return;

    string Ergebnis = "";
    // DB DIREKT
    if (this.C_Modus.Text == "DBDirekt")
    {
        //DB.DetectChanges(); nur, wenn nicht StartTracking()!
        Ergebnis = DB.SaveChanges().ToString();
    }
    // BL
    if (this.C_Modus.Text == "BL")
        Ergebnis = BL.SaveFluege(fluegeKomplett.ToList()).ToString();
}

```

```
// APP SERVER
if (this.C_Modus.Text == "Service")
{
    // nur geänderte Objekte an den Server senden (vgl. GetChanges() im DataSet!)
    ObservableCollection<Flug> fluegeNurGeaenderte = new ObservableCollection<Flug>(
        (from f in fluegeKomplett where f.ChangeTracker.State != ObjectState.Unchanged
         select f).ToList());
    // Webservice aufrufen
    Ergebnis = Dienst.SaveFluege(fluegeNurGeaenderte);
}

// AcceptChanges muss man für jedes Objekt nun einzeln aufrufen !!!
foreach (Flug f in fluege)
{
    f.AcceptChanges();
}

// Originalliste aktualisieren, damit die gelöschten Objekte dort verschwinden
fluegeKomplett = new ObservableCollection<Flug>(fluegeAktuelleAnsicht);

// Status anzeigen
SetStatus("Gespeichert in der Datenbank: " + Ergebnis);
}
```

**Listing 12.35** Speichern der geänderten Flüge mit Fallunterscheidung für die drei Architekturszenarien

## Unterstützung für Stored Procedures

Das EF unterstützt gespeicherte Prozeduren sowohl zum Lesen als auch zum Ändern (INSERT, UPDATE, DELETE) von Daten. Um gespeicherte Prozeduren zu nutzen, muss man diese mit dem Visual Studio-Assistenten importieren. Anders als für Tabellen und Sichten legt der Entity Framework-Designer für gespeicherte Prozeduren nicht automatisch Programmcode an. Damit man eine gespeicherte Prozedur dann tatsächlich im Programmcode aufrufen kann, muss man einen so genannten »Funktionsimport« erstellen.

Bereits in EF1 gab es die Möglichkeit, Stored Procedures zu nutzen. Stored Procedures konnte man auf Methoden des Objektkontextes abbilden. Stored Procedures, die bestimmte Parameter besaßen, konnte man darüber hinaus für die Insert-, Update- und Delete-Operationen von Entitäten verwenden.

Leider gab es in EF 1 ein Problem mit Stored Procedures, die elementare Datentypen zurückliefern. Microsofts Implementierung war unvollständig und fehlerhaft. Die Implementierung in EF 4.0 ist besser.

## Beispiele

Am Beispiel von vier Stored Procedures soll die Nutzung von Stored Procedures im Entity Framework 4.0 gezeigt werden (siehe Tabelle 12.3).

Stored Procedures	Bedeutung	Implementierung
GetFlug	Liefert einen Flug anhand der Flugnummer. Ergebnis ist eine Tabelle (mit einem Eintrag), die der Entität Flug entspricht.	Select * from Flug where (FlugNr = @FlugNr)
GetAlleBuchungen	Liefert das Ergebnis des Joins aus Passagier und Person. Ergebnis ist eine Tabelle, die keiner Entität entspricht.	SELECT dbo.Passagier.PersonID, dbo.Passagier.KundeSeit, dbo.Passagier.KundenStatus, dbo.Person.Name, dbo.Person.Vorname, dbo.Person.Land, dbo.Person.Geburtstag, dbo.Person.Foto, dbo.Person.EMail FROM dbo.Passagier INNER JOIN dbo.Person ON dbo.Passagier.PersonID = dbo.Person.PersonID
CountFlugByOrt	Liefert die Anzahl der Flüge von einem Abflugort. Ergebnis ist eine Zahl.	SELECT COUNT(FlugNr) AS Expr1, Abflugort FROM Flug GROUP BY Abflugort HAVING (Abflugort = @Ort)
GetAbflugOrte	Liefert eine Liste aller Abflugorte. Ergebnis ist eine Liste von Zeichenketten.	SELECT DISTINCT Abflugort FROM Flug AS t0 ORDER BY Abflugort

**Tabelle 12.3** Vier Stored Procedures mit unterschiedlichen Rückgabetypen als Testszenario

## Entity Framework-Assistent

Im ersten Schritt ist der Entity Framework-Assistent (entweder beim Neuanlegen eines Modells oder nachträglich mit *Update Modell From Database*) zu verwenden, um die Stored Procedures, die genutzt werden sollen, in das Modell einzubinden.



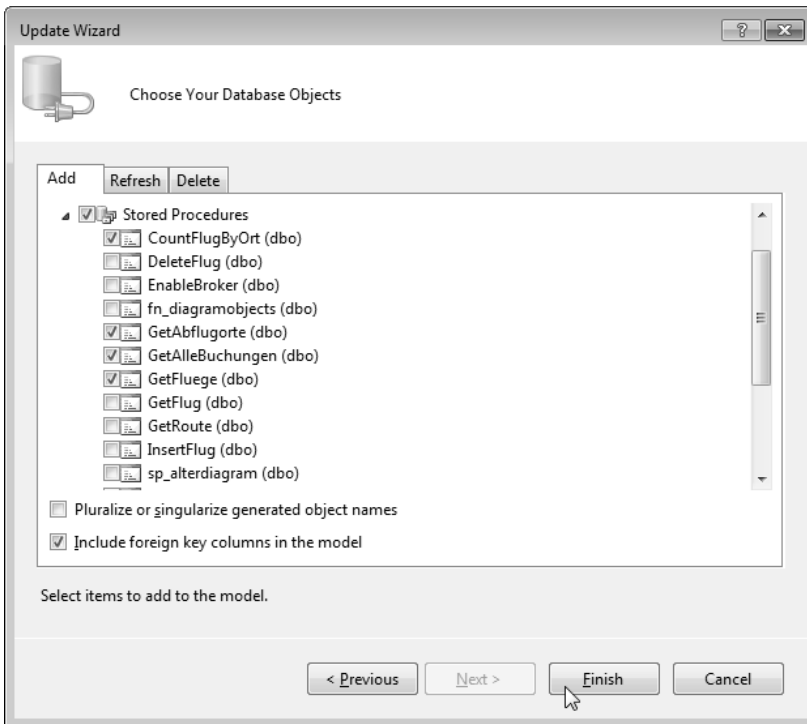


Abbildung 12.40 Auswahl der Stored Procedures

## Verwendung der Stored Procedures

Die derart eingebundenen Stored Procedures kann man nun für Insert-/Update-/Delete-Operationen von Entitätsklassen verwenden oder zum direkten Funktionsaufruf. An der ersten Möglichkeit, die es auch schon in Visual Studio 2008 gab, hat sich nichts geändert. Sie ist weiterhin über *Mapping Details* aufzurufen. Die zweite Funktion über den *Model Browser* gab es zwar grundsätzlich auch schon, sie wurde aber erheblich erweitert.

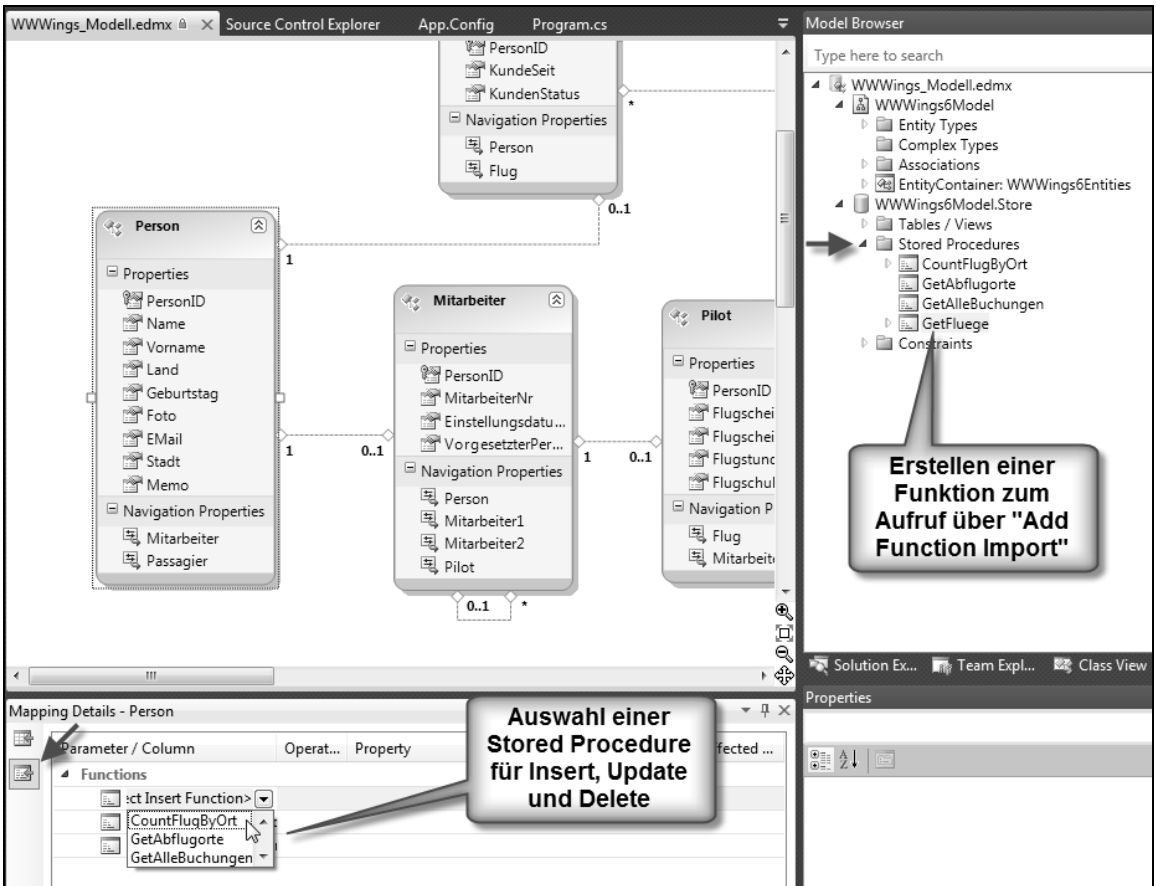
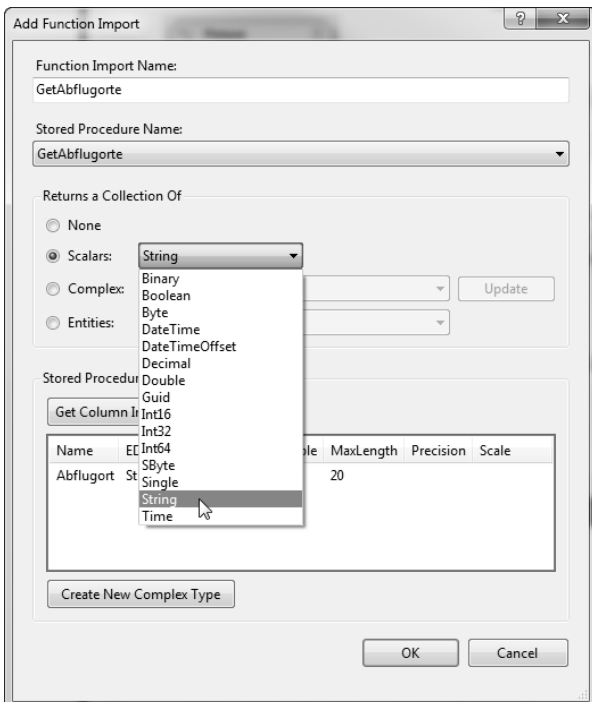


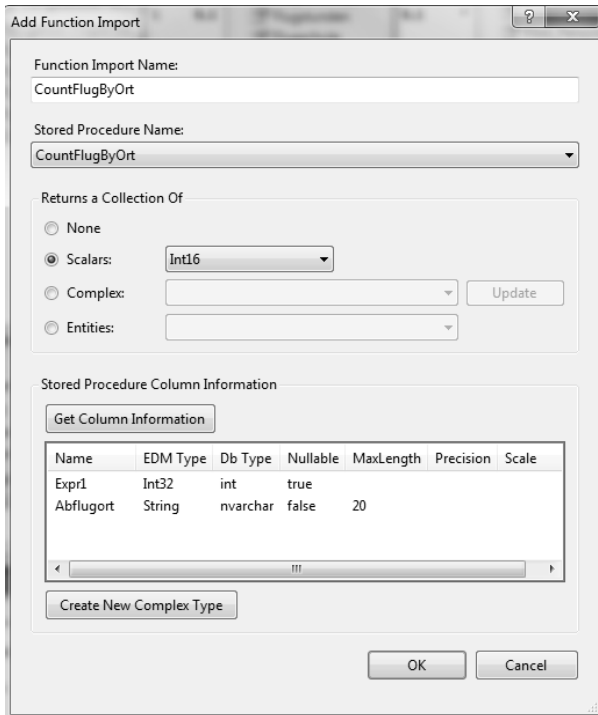
Abbildung 12.41 Verwendung einer Stored Procedure im Entity Framework-Designer

## Add Function Import

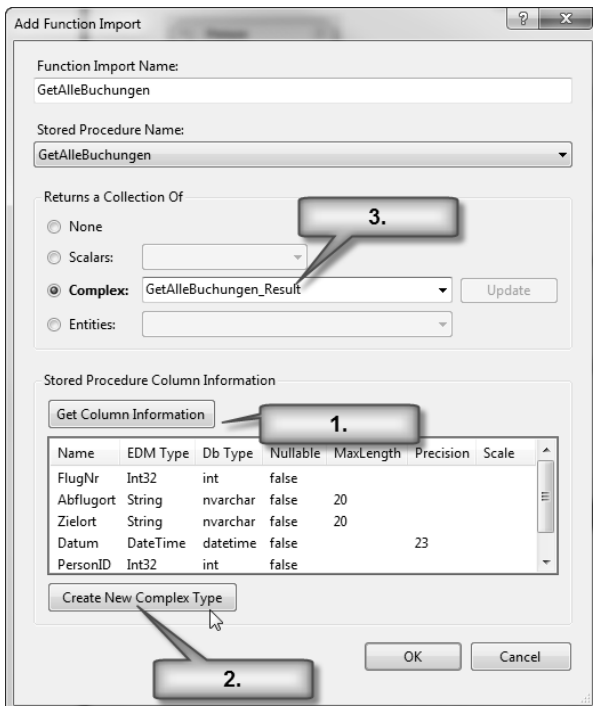
Nach dem Aufruf der Funktion *Add Function Import* im Kontextmenü einer Stored Procedure im Modell Browser erscheint ein überarbeitetes Dialogfeld. Hier kann man sich über *Get Column Information* anzeigen lassen, was die Stored Procedure zurückliefert. Auf dieser Basis muss der Entwickler die Art des Rückgabewerts festlegen:

- **None** Kein Rückgabewert
- **Scalar** Die Stored Procedure liefert einen elementaren Datentyp
- **Entities** Die Stored Procedure liefert eine der im Modell vorhandenen Entitäten
- **Complex** Die Stored Procedure liefert einen zusammengesetzten (»komplexen«) Datentyp. Diese Option ist für alle strukturierten Typen gedacht, die nicht einer Entitätsklasse entsprechen, z.B. das Ergebnis eines Views. Gegebenenfalls ist vorher der komplexe Typ zu erzeugen mit *Create New Complex Type*.

Abbildung 12.42 Festlegung des Rückgabetypes für *GetFlug*Abbildung 12.43 Festlegung des Rückgabetypes für *GetAbflugorte*



**Abbildung 12.44** Festlegung des Rückgabetypes für *CountFlugByOrt*



**Abbildung 12.45** Festlegung des Rückgabetypes für *GetAlleBuchungen*: 1. Abholen der Spalteninformationen, 2. Erzeugen eines passenden .NET-Typs, 3. Anzeige des generierten Klassennamens

## Verwendung der Stored Procedures

Das folgende Listing zeigt sowohl den Quellcode zum Aufruf der vier Stored Procedures als auch den Inhalt der Rückgabe mithilfe der Debugger Data Tipps von Visual Studio 2010:

- `GetFlug()` liefert unter Angabe einer Flugnummer eine Liste mit genau einem Flug-Objekt (oder eine leere Liste, wenn es den Flug nicht gibt!)
- `CountFlugByOrt()` liefert unter Angabe eines Ortes eine Liste mit genau einer Zahl (oder eine leere Liste, wenn es den Ort nicht gibt!)
- `GetAbflugorte()` liefert eine Liste von Zeichenketten
- `GetAlleBuchungen()` liefert eine Liste des neu im *Add Function Import*-Dialogfeld erstellten Typs `GetAlleBuchungen_Result`

```
private static void Demo_SPs()
{
    WWWings_EF4_Standard.WWWings6Entities ctx = new WWWings_EF4_Standard.WWWings6Entities();

    // Aufruf der SP "GetFluege"
    ObjectResult<WWWings_EF4_Standard.Flug> Fluege = ctx.GetFlug(101);
    List<WWWings_EF4_Standard.Flug> Fluege2 = Fluege.ToList();
    if (Fluege2.Count > 0) Console.WriteLine("Flug gefunden!");

    // Aufruf der SP "CountFlugByOrt"
    ObjectResult<int?> FluegeVonRom = ctx.CountFlugByOrt("Rom");
    List<int?> FluegeVonRom2 = FluegeVonRom.ToList();
    if (FluegeVonRom2.Count > 0) Console.WriteLine("Anzahl Flüge: " + FluegeVonRom2[0]);

    // Aufruf der SP "GetAbflugorte"
    ObjectResult<string> AbflugOrte = ctx.GetAbflugorte();
    List<string> AbflugOrte2 = AbflugOrte.ToList();
    Console.WriteLine("Anzahl Abflugorte: " + AbflugOrte2.Count);

    // Aufruf der SP "GetAlleBuchungen"
    ObjectResult<WWWings_EF4_Standard.GetAlleBuchungen_Result> Buchungen = ctx.GetAlleBuchungen();
    List<WWWings_EF4_Standard.GetAlleBuchungen_Result> Buchungen2 = Buchungen.ToList();
    Console.WriteLine("Anzahl Buchungen: " + Buchungen2.Count);
}
```

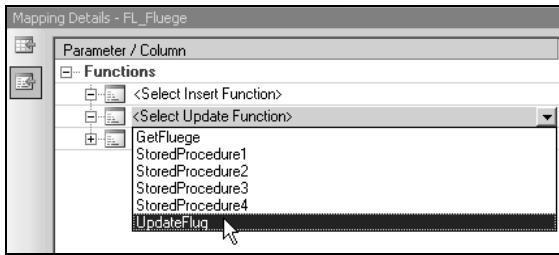
Abbildung 12.46 zeigt die Visual Studio 2010 Debugger Data View mit den folgenden Werten:

- `Fluege2`: Count = 1
- `FluegeVonRom2`: Count = 1, Value = 32
- `AbflugOrte`: (System.Data.Objects.ObjectResult<string>)
- `Buchungen2`: Count = 4281

Abbildung 12.46 Aufruf und Rückgabewerte der Stored Procures

## Gespeicherte Prozeduren zur Verwendung für INSERT, UPDATE und DELETE

Die Verwendung von gespeicherten Prozeduren zur Anwendung von INSERT, UPDATE und DELETE legt man im Fenster *Mapping Details* fest. Dort muss man zunächst die Sicht *Map Entity to Functions* (zweite Schaltfläche von oben an der linken Seite) und dann die gewünschte gespeicherte Prozedur auswählen. Alternativ kommt man dorthin, indem man im Designer auf einer Entität *Stored Procedure Mapping* auswählt.



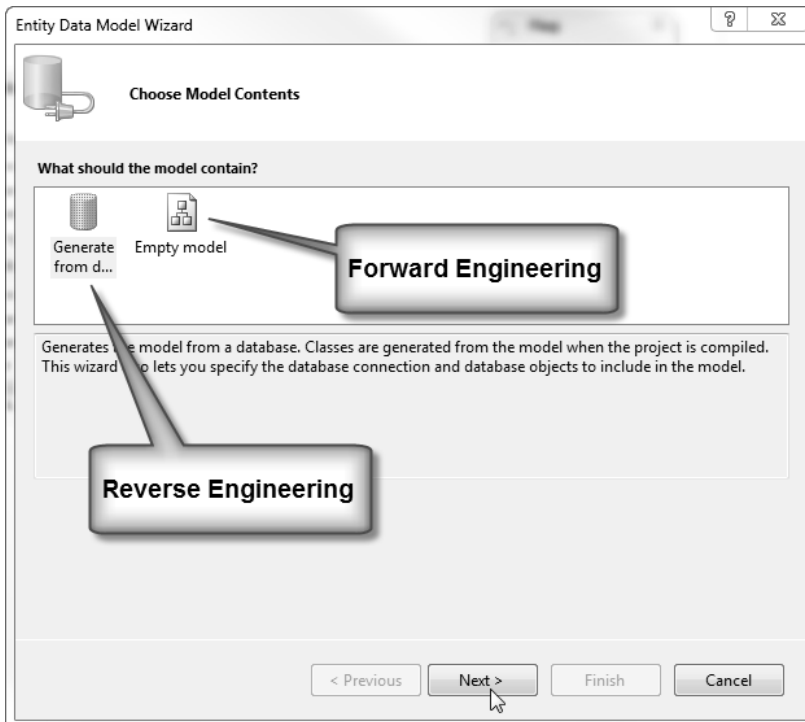
**Abbildung 12.47** Festlegen einer gespeicherten Prozedur für die Aktualisierung der Tabelle *FL\_Fluege*

## Forward Engineering (Model-First / Domain First)

Die erste Version des EF konnte nur Reverse Engineering. EF 4 beschert den .NET-Entwicklern nun auch Forward Engineering.

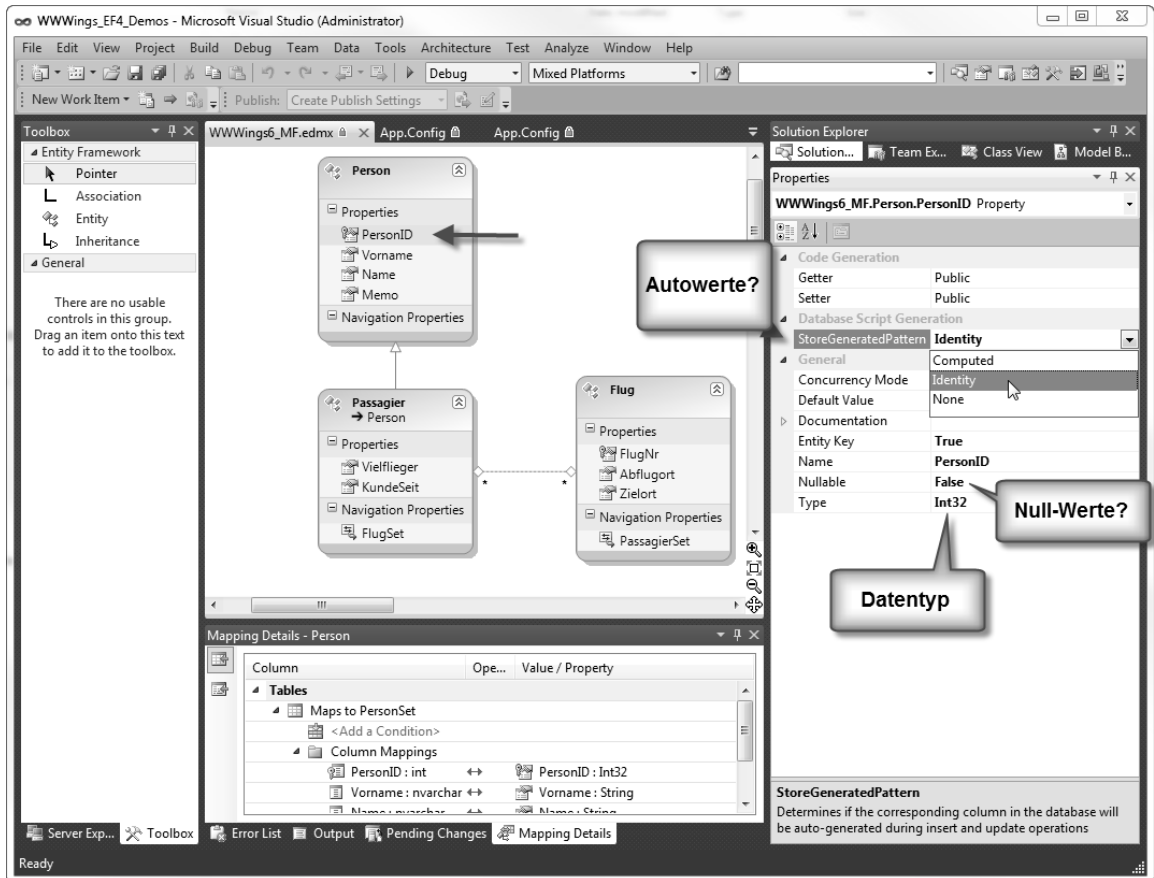
### Anlegen eines Modells für Forward Engineering

Für Forward Engineering startet man den Entity Framework-Designer (Vorlage *ADO.NET Entity Framework Data Modell*) in Visual Studio 2010 mit einem leeren EF-Modell (Auswahl *Empty Model*).



**Abbildung 12.48** Forward Engineering ist nun eine echte Option in Visual Studio 2010

Die leere Zeichenfläche füllt man nun aus der Werkzeugleiste (links) mit Entitäten (Entity) sowie zugehörigen Assoziationen (Association) und Vererbungsbeziehungen (Inheritance). Die folgenden drei Abbildungen zeigen, wie man wichtige Eigenschaften (Datentyp, Null-Werte erlauben, Autowerte, Eigenschaften der Zeichenketten und Kardinalität für Assoziationen) durch das Eigenschaftsfenster festlegen kann.



**Abbildung 12.49** Festlegung, ob Primärschlüssel per Autowert erzeugt werden soll

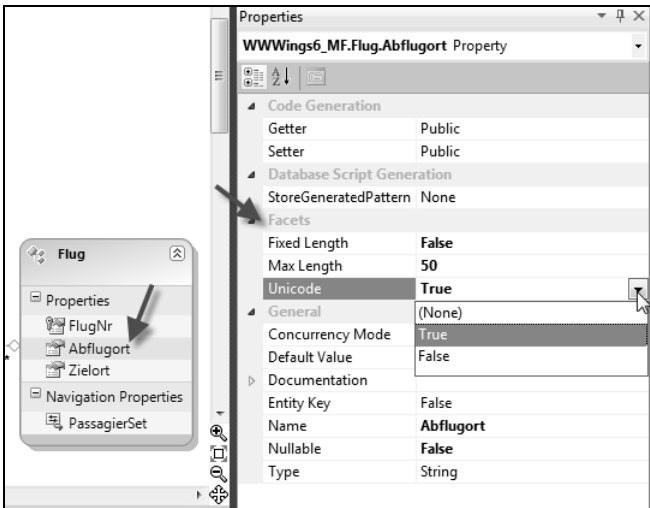


Abbildung 12.50 Festlegung, wie eine Zeichenkettenspalte erzeugt werden soll

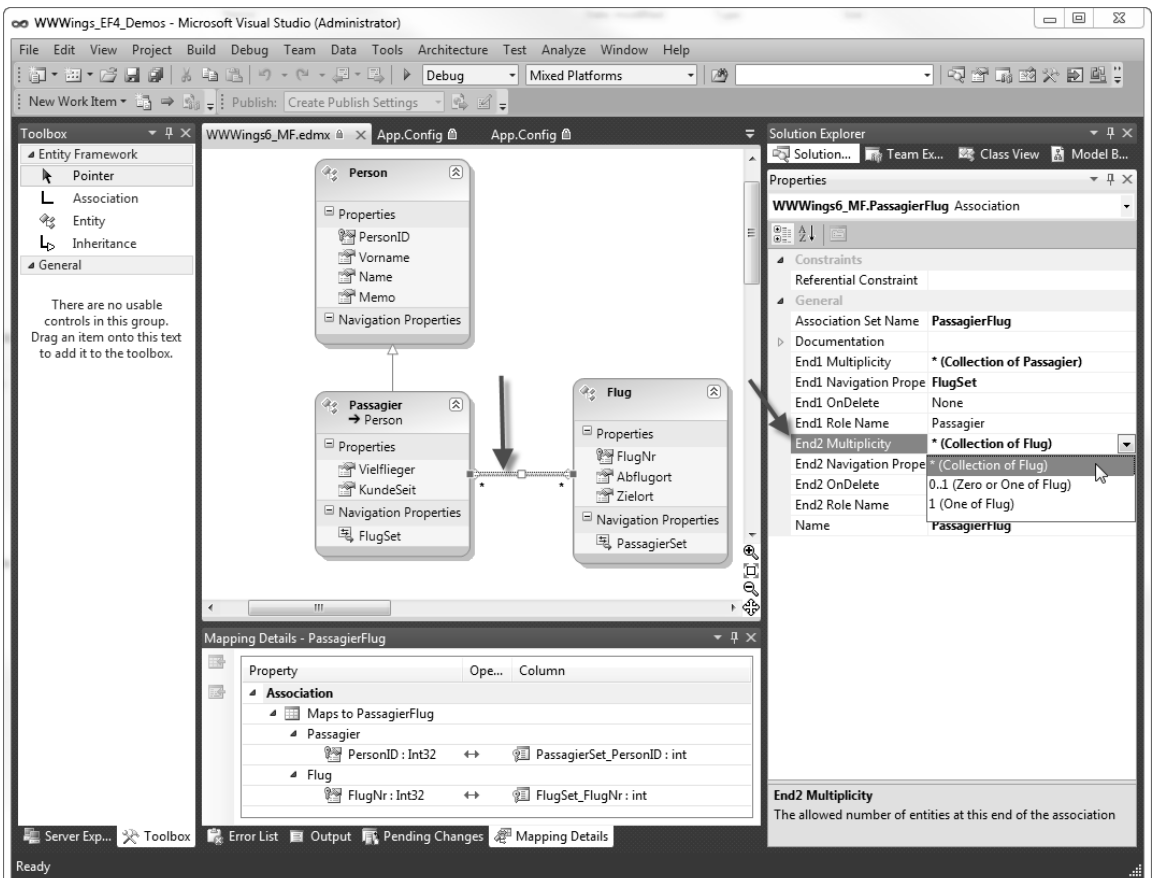
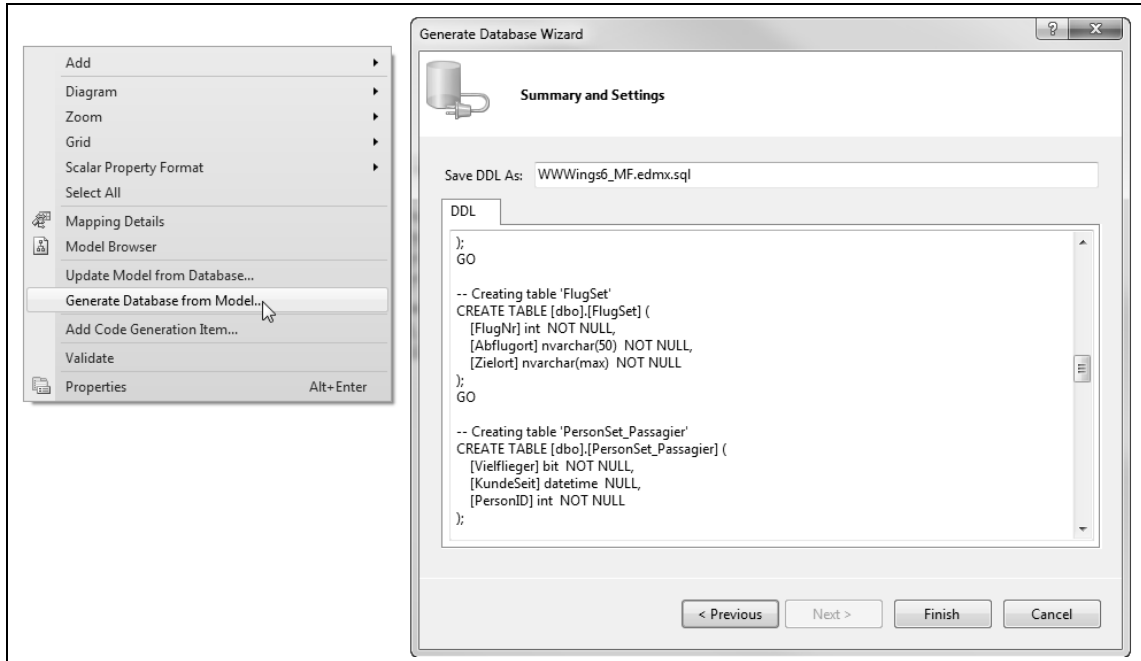


Abbildung 12.51 Festlegung der Kardinalitäten für eine Assoziation



## Erzeugen der Datenbank

Nach dem Anlegen des Modells stößt der Entwickler dann über die neue Funktion *Generate Database Script from Model* im Kontextmenü des Designers (siehe Abbildung 12.52) das Erstellen eines SQL-DDL-Skripts (DDL = Data Definition Language) an mit entsprechenden *Create Table*-Befehlen.



**Abbildung 12.52** Anstoßen der SQL-Generierung

Das Skript wird in Visual Studio angezeigt und als Teil des Projekts in einer *.sql*-Datei gespeichert. Über das Symbol *Execute SQL* (`Strg` + `⇧` + `E`) kann man es dann direkt ausführen.

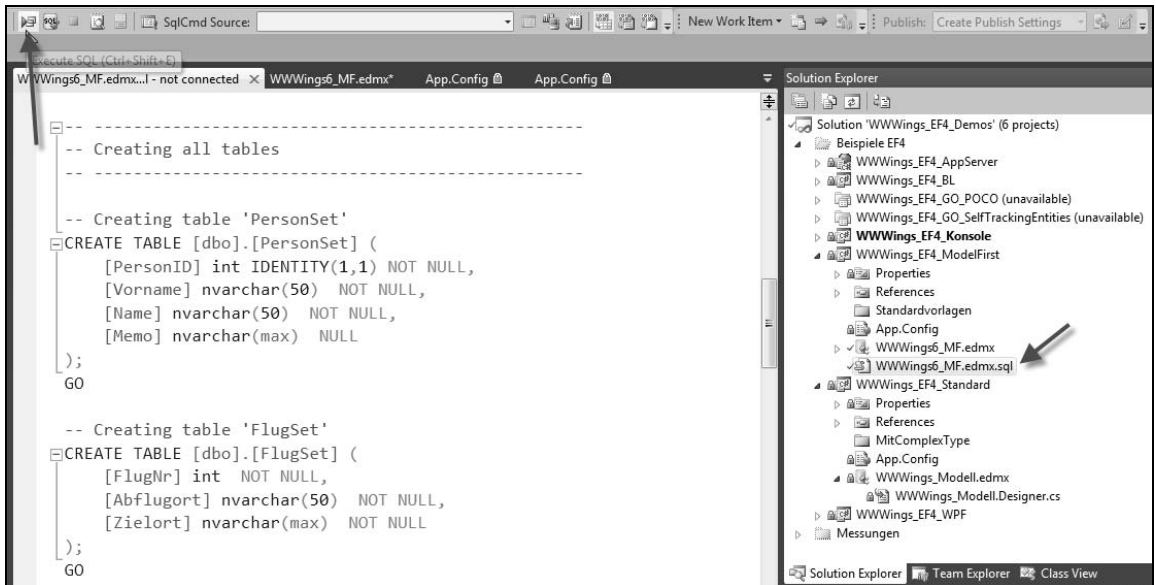


Abbildung 12.53 Das generierte SQL-Skript im Projekt und der Befehl *Execute SQL*

#### HINWEIS

Das SQL-Skript enthält keinen Create Database-Befehl, sondern geht davon aus, dass man die Datenbank vorher manuell angelegt hat.

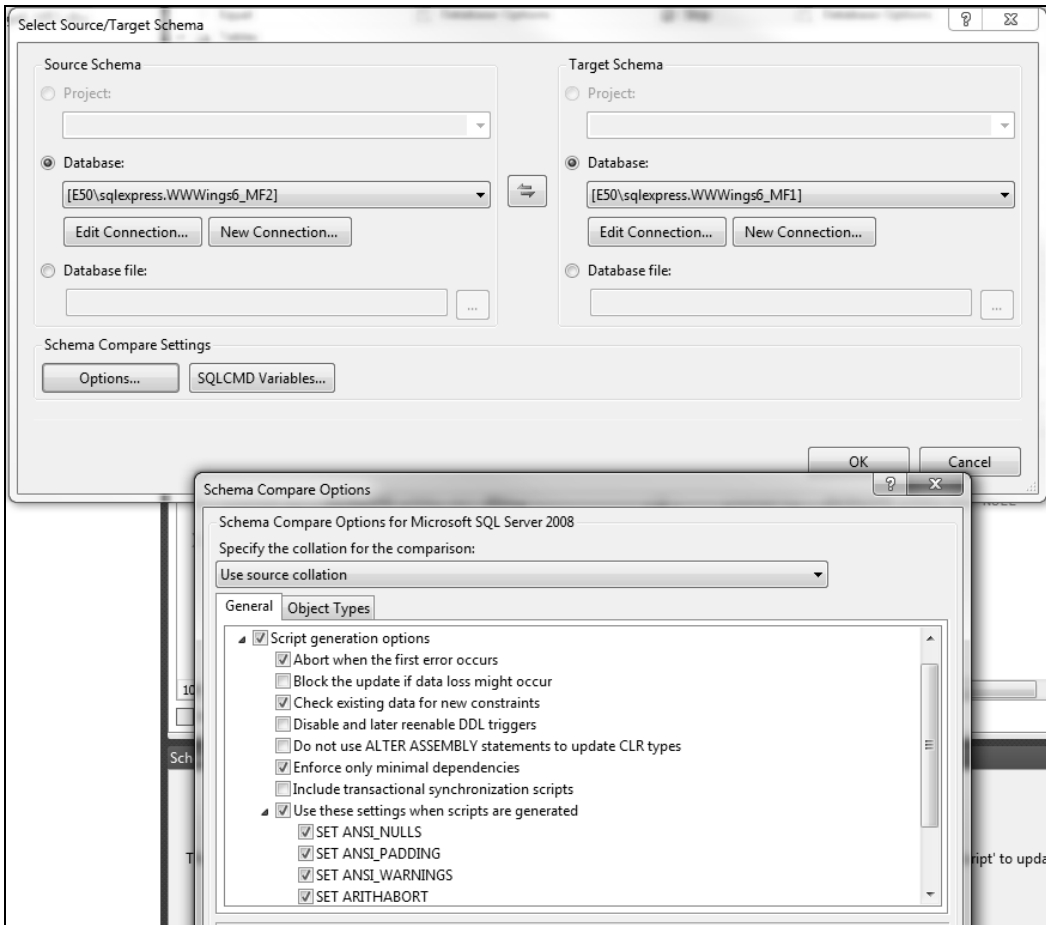
## Modell- und Schemaversionierung

Eine besondere Herausforderung besteht bei Veränderung des Modells im Zuge neuer Versionen. Ein gutes ORM-Werkzeug kann bei solchen Modelländerungen ein Änderungsskript für die Datenbank erzeugen, das die Datenbank auf das neue Schema aktualisiert. Leider vermag der Visual Studio 2010-Entity Framework Designer direkt noch keine Änderungsskripte für das Datenmodell zu erstellen, wenn sich an dem Objektmodell nachträglich Änderungen ergeben.

Man kann sich aber behelfen, indem man eine zweite Datenbank für die neue Version von dem Entity Framework-Designer erstellen lässt und dann mithilfe der Datenbankwerkzeuge in Visual Studio 2010 (Menü *Data/Schema Compare*) einen Schemavergleich startet, der ein Änderungsskript erstellt.

#### HINWEIS

Dies Funktion *Schema Compare* ist nur in Visual Studio 2010 Premium und Visual Studio 2010 Ultimate verfügbar, nicht aber in der Express- oder Professional-Version.



**Abbildung 12.54** Anlegen eines Schema-Vergleichs zwischen zwei verschiedenen Versionen einer Datenbank

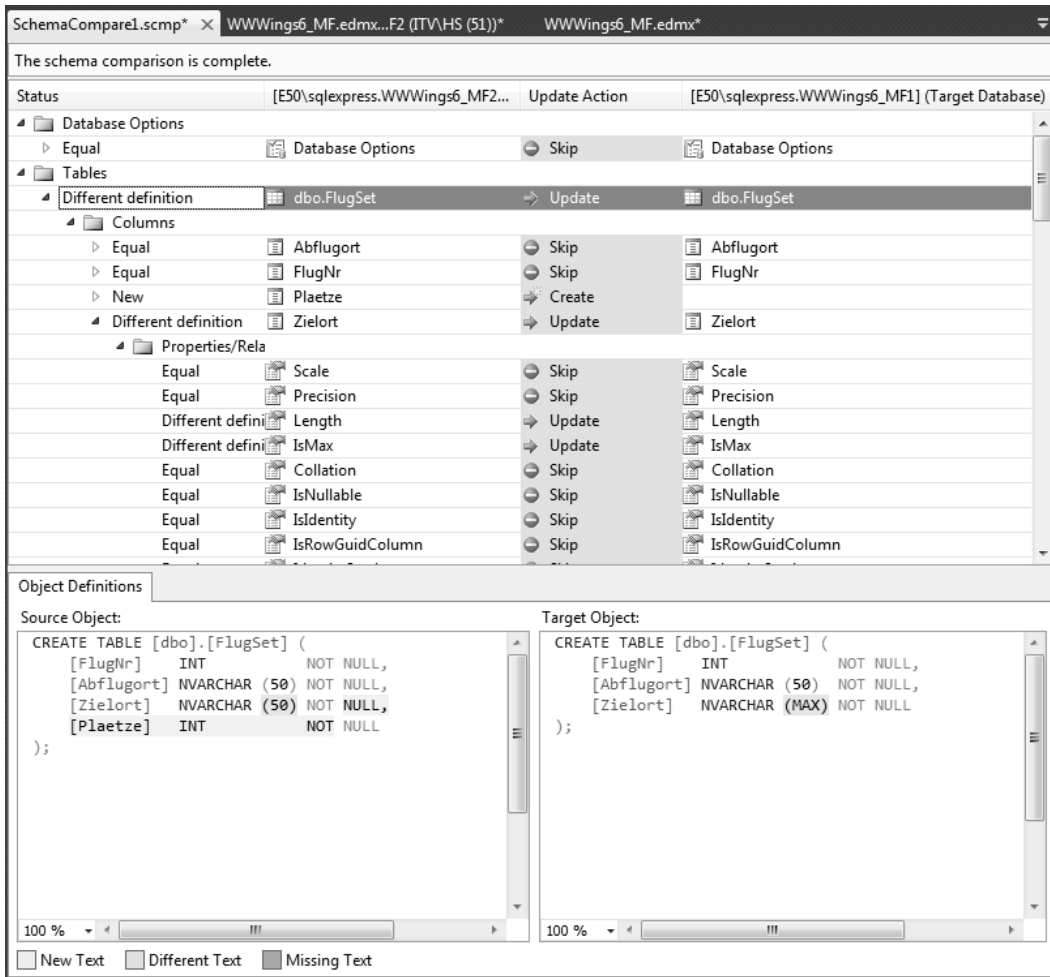
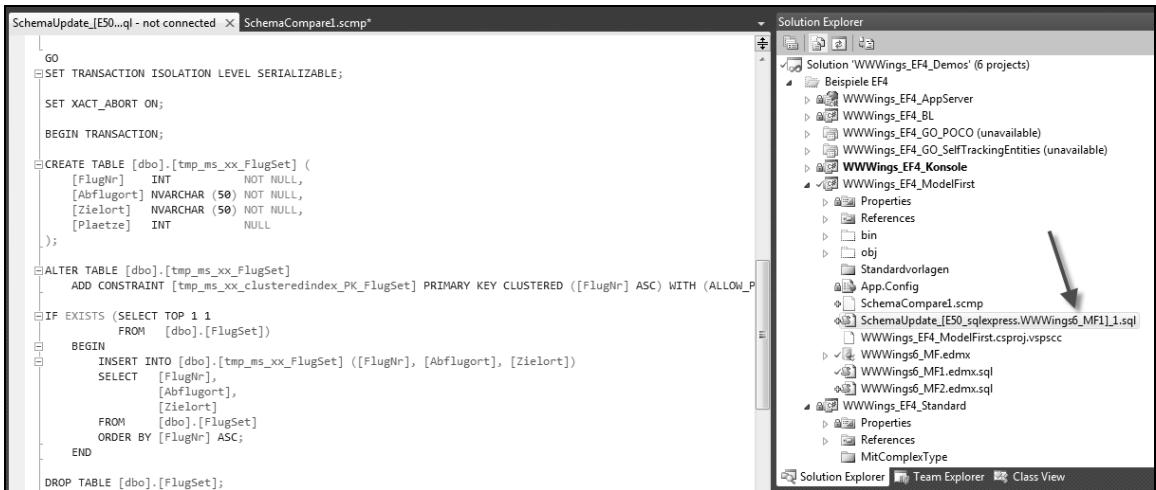


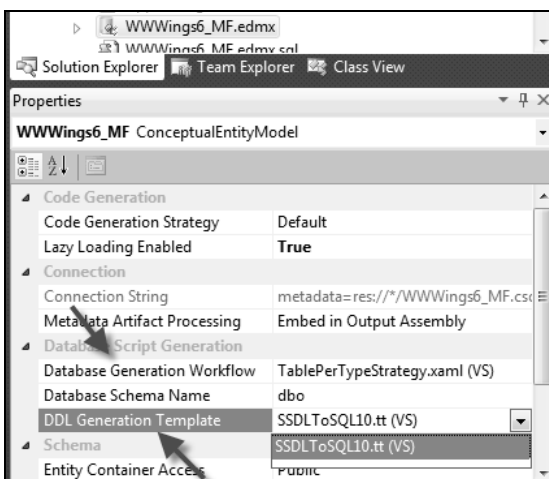
Abbildung 12.55 Visual Studio zeigt die Unterschiede der beiden Schemata



**Abbildung 12.56** Das erzeugte Änderungsskript, das aus der Datenbank Version 1 eine Datenbank der Version 2 macht – unter Beibehaltung der Daten

## Anpassung der Datenbankgenerierung

Fortgeschrittene Benutzer können sowohl das generierte SQL als auch den Vorgang der SQL-Generierung anpassen. Das generierte SQL ergibt sich aus einer T4-Vorlage (SSDLToSQL10.tt) und der Ablauf aus einem Workflow (TablePerTypeStrategy.xaml – mit Windows Workflow Foundation 4.0). Beide Vorlagen findet man unter *C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\Extensions\Microsoft\Entity Framework Tools\DBGen*. Man kann die Vorlagen in den Eigenschaften des Modells unter *Database Script Generation* (siehe Abbildung 12.57) ändern. Man kann weitere Vorlagen auch ablegen unter *%localappdata%\microsoft\visualstudio\10.0\extensions\microsoft\entity framework tools\dbgen*.



**Abbildung 12.57** Änderung der Datenbankgenerierung

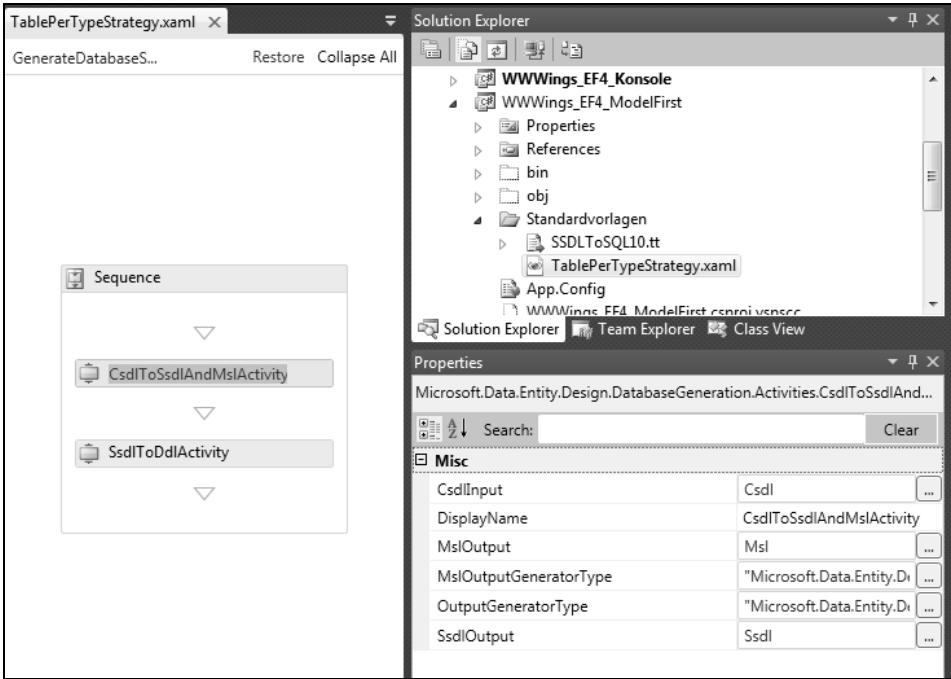


Abbildung 12.58 Standardworkflow für die Datenbankgenerierung

## Entity Designer Database Generation Power Pack

Microsoft arbeitet an weiteren Verbesserungen insbesondere hinsichtlich der Erzeugung von Änderungsskripten und anderen Strategien zur Umsetzung der Vererbung in der Datenbank (z.B. Table-per-Type-Strategie oder Table-per-Hierarchy-Strategie). Diese Verbesserungen liegen zum Zeitpunkt der Erstellung dieses Buchkapitels unter dem Titel *Entity Designer Database Generation Power Pack* als optionales Add-On vor [VSGa1101] – allerdings nur für Visual Studio 2010 *Release Candidate* (RC). Eine Testinstallation auf der RTM-Version war zwar erfolgreich, man sollte solche Vorabversionen aber mit großer Vorsicht verwenden.

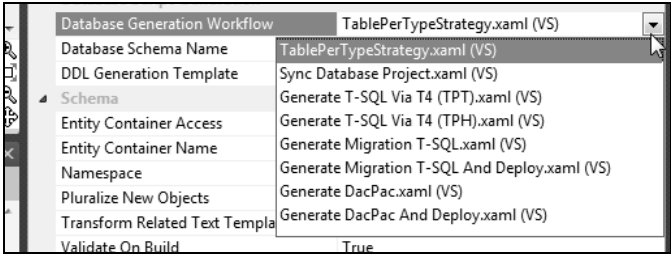
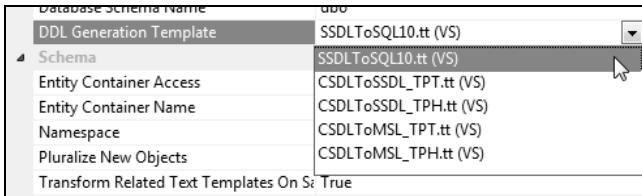


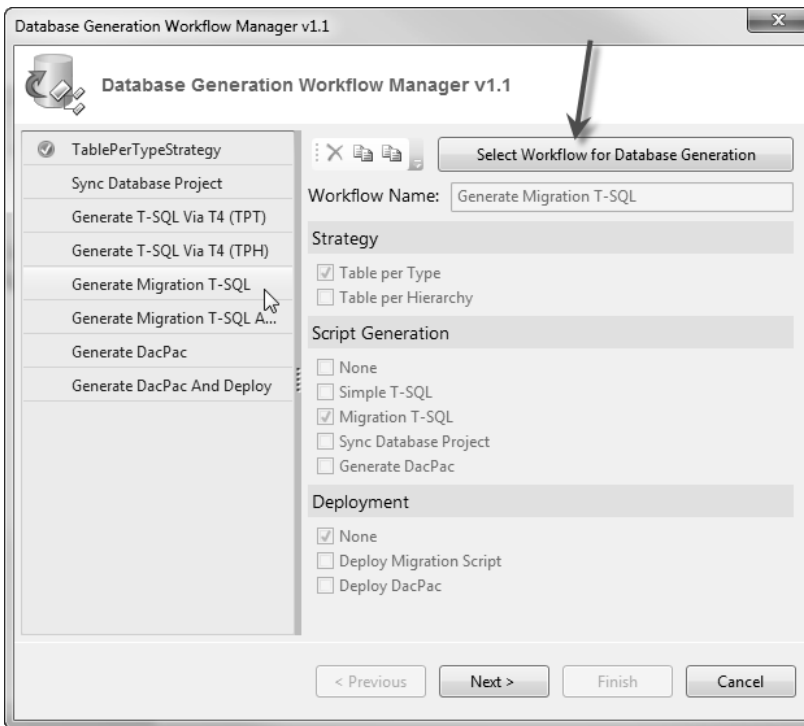
Abbildung 12.59 Liste des Datenbankerstellungsworkflows nach dem Installieren des Entity Designer Database Generation Power Pack



**Abbildung 12.60** Liste der T4-Vorlagen nach dem Installieren des Entity Designer Database Generation Power Pack

Nach dem Installieren des PowerPacks zeigt der Aufruf von *Generate Database Script from Model* ein ganz neues Dialogfeld, in dem man die verfügbaren Workflows auswählen kann.

**ACHTUNG** Zur Wahl eines alternativen Workflows reicht es nicht, diesen in der Spalte links mit der Maus auszuwählen. Man muss zusätzlich auch *Select Workflow for Database Generation* anklicken, sodass der grüne Haken neben dem gewünschten Workflow steht.



**Abbildung 12.61** Auswahl der Workflows zu Beginn der Generierung

Die Auswahl *Generate Migration T-SQL* erzeugt dann tatsächlich ein Änderungsskript (siehe folgende Abbildung).

**HINWEIS** Bei neu hinzugefügten Spalten zu bestehenden mit Daten gefüllten Tabellen muss man diese neuen Spalten mit `Nullable=True` im Modell erstellen, da die Migration sonst an fehlenden Werten scheitern wird.

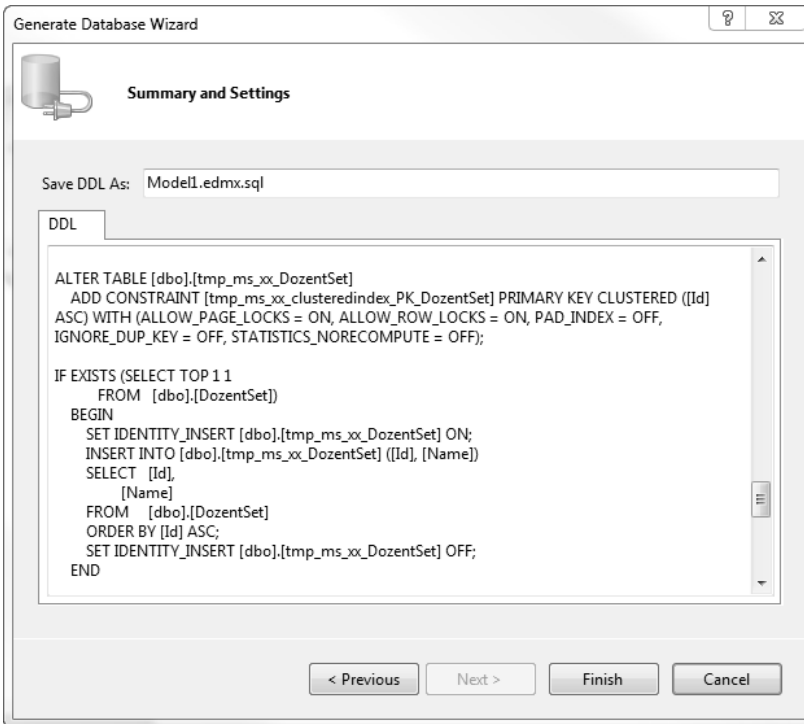


Abbildung 12.62 Änderung einer bestehenden Tabelle unter Beibehaltung der Daten

## Datenbankgenerierung zur Laufzeit

Interessant ist, dass man nun in EF wie in LINQ to SQL auch, die Generierungsfunktion auf einfache Weise zur Laufzeit starten kann. DieObjectContext-Klasse in EF 4 bietet dazu die neuen Methoden `CreateDatabaseScript()`, `CreateDatabase()`, `DatabaseExists()` und `DeleteDatabase()`.

```
/// <summary>
/// Erzeugung der Datenbank aus dem Modell zur Laufzeit und Befüllen mit Daten
/// </summary>
private static void ModelFirstDemo()
{
    Console.WriteLine("Model First Demo mit Datenbankerzeugung...");

    string ConnString =
    (@"metadata=res://*/WWWings6_MF.csdl|res://*/WWWings6_MF.ssdl|res://*/WWWings6_MF.msl";
    provider=System.Data.SqlClient;provider connection string='Data Source=.;
    Initial Catalog=WWWings6_MF;Integrated Security=True;MultipleActiveResultSets=True');
    Console.WriteLine(ConnString);
    EntityConnection econn = new EntityConnection(ConnString);
    WWWings_EF4_ModelFirst.WWWings6_MFContainer mf = new
    WWWings_EF4_ModelFirst.WWWings6_MFContainer(econn);
```



```
// ----- DB löschen, wenn sie schon vorhanden ist
if (mf.DatabaseExists())
{
    Console.WriteLine("Lösche vorhandene DB...");
    mf.DeleteDatabase();
}
// ----- DB erzeugen
Console.WriteLine("Erzeuge DB...");
mf.CreateDatabase();
Console.WriteLine("DB erzeugt!");

// ----- Befüllen
WWings_EF4_ModelFirst.Passagier p = new WWings_EF4_ModelFirst.Passagier();
//p.PersonID = 1; // Nur wenn PersonID nicht AutoWert!
p.Vorname = "Holger";
p.Name = "Schwichtenberg";
p.Vielflieger = false;
p.KundeSeit = DateTime.Now;
mf.PersonSet.AddObject(p);
mf.SaveChanges();
Console.WriteLine("Person: " + p.PersonID);
WWings_EF4_ModelFirst.Flug f = new WWings_EF4_ModelFirst.Flug();
f.FlugNr = p.PersonID + 100;
f.Abflugort = "Essen";
f.Zielort = "Düsseldorf";
mf.FlugSet.AddObject(f);
Console.WriteLine("Flug: " + f.FlugNr);
f.PassagierSet.Add(p);
mf.SaveChanges();
Console.WriteLine("Model First Demo: Daten gespeichert!");
}
```

**Listing 12.36** Erzeugung der Datenbank aus dem Modell zur Laufzeit und Befüllen mit Daten

## Anpassungen und Erweiterbarkeit

Die Themen Anpassungen und Erweiterbarkeit umfassen in Entity Framework folgende Möglichkeiten:

- Austauschen der Codegenerierungsvorlage (dazu gab es schon einen Abschnitt)
- Erweitern der generierten Klassen durch partielle Klassen (siehe unten)
- Ereignisse der Klassen abfangen
- Erstellen eines eigenen Entity Framework-Providers (hier nicht behandelt, vgl. als Muster [MSDN34]).

**ACHTUNG** Es macht keinen Sinn, direkt Änderungen in den generierten Codedateien (*designer*-Dateien) vorzunehmen, da der Entity Framework-Designer bei jeder Änderung am Modell die Codegenerierung neu anstößt. Entweder muss man die Codegenerierungsvorlage anpassen (wenn es um eine systematische Änderung geht) oder für einzelne Klassen eine partielle Klasse schreiben (siehe unten).

## Erweitern der generierten Klassen durch partielle Klassen

Alle generierten Klassen (sowohl die Entitätsklassen als auch die Kontextklasse) sind partielle Klassen, die man in eigenen Codedateien (im gleichen Projekt!) erweitern kann. In diesen partiellen Klassen kann man:

- Im Konstruktor (komplexere) Standardwerte definieren
- Berechnete Attribute ergänzen
- Die partiellen Methoden und Ereignisse abfangen, die die generierten Klassen bereitstellen

---

**TIPP** Bei den Entitätsklassen gibt es für jedes Attribut jeweils zwei partielle Methoden, die vor und nach der Änderung eines Werts aufgerufen werden.

---

```

/// <summary>
/// Erweiterungen der generierten Entitätsklasse
/// </summary>
partial class Person
{
    public Person()
    {
        // Standardwerte setzen im Konstruktor
        if (DateTime.Now.Year >= 1990)
            this.Land = "DE";
        else
            this.Land = "BRD";
    }

    /// <summary>
    /// Berechnetes Attribut
    /// </summary>
    public string GanzerName
    {
        get
        { return this.Vorname + " " + this.Name; }
    }

    /// <summary>
    /// Validierung
    /// </summary>
    partial void OnGeburtstagChanging(DateTime? value)
    {
        if (this.Geburtstag > DateTime.Now)
        {
            Console.WriteLine("Geburtstag darf nicht in Zukunft liegen!");
            value = DateTime.Now;
        }
    }
}

```

**Listing 12.37** Erweitern der generierten Entitätsklasse *Person*

## Ereignisse der Kontextklasse

Die Kontextklasse bietet im Standard zwei Ereignisse:

- Das `ObjectMaterialized()`-Ereignis (neu ab Entity Framework 4.0!) wird jedes Mal ausgelöst, wenn ein Entitätsobjekt materialisiert wird, d.h. wenn eine Instanz einer Entitätsklasse erzeugt und mit Daten aus der Datenbank befüllt wird. Dabei wird das Ereignis auch dann ausgelöst, wenn die Objekte nicht direkt aus der Datenbank stammen, sondern aus einem `DataReader` materialisiert wurden (vgl. Abschnitt »Direkte SQL-Ausführung«). Das Ereignis erhält im Parameter `sender` einen Verweis auf den Objektkontext und als `ObjectMaterializedEventArgs` ein Objekt mit einem einzigen Attribut `Entity`, das auf das erzeugte Entitätsobjekt verweist.
- `SavingChanges()` wird ausgeführt, wenn Entity Framework speichert. Hier kann man den `ObjectStateManager` abfragen. Der Parameter `sender` enthält dafür einen Verweis auf die Kontextklasse. Der zweite Parameter ist leer.

---

**HINWEIS** Das Ereignis `ObjectMaterialized()` wird nicht erzeugt, wenn man eine neue Instanz einer Entitätsklasse im Programmcode anlegt, sondern nur beim Laden bestehender Objekte.

---

```
/// <summary>
/// Kontext-Ereignisse
/// </summary>
public static void Beispiel3_Events()
{
    // Kontext instanziiieren unter Verwendung der Verbindungszeichenfolge aus Konfigurationsdatei
    using (WWings6Entities modell = new WWings6Entities())
    {
        modell.SavingChanges += new EventHandler(modell_SavingChanges);
        modell.ObjectMaterialized += new ObjectMaterializedEventHandler(modell_ObjectMaterialized);

        string ort = "Rom";
        string name = "Müller";

        // Abfrage definieren
        var fluege = from f in modell.Flug
                     where f.Abflugort == ort && f.Passagier.Count > 0 &&
                          f.Passagier.Any(p => p.Person.Name == name)
                     select f;

        // Ergebnis ausgeben
        foreach (WWings.G0.EF.Flug f in fluege)
        {
            f.FreiePlaetze++;
            Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
                              f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);
        }

        modell.SaveChanges();
    } // Ende using-Block -> Dispose() wird aufgerufen
}
```

```
static void modell_SavingChanges(object sender, EventArgs e)
{
    WWWings6Entities modell = sender as WWWings6Entities;
    Console.WriteLine("== Speichern:");
    Console.WriteLine("# Anzahl geänderter Objekte: " +
        modell.ObjectStateManager.GetObjectStateEntries(EntityState.Modified).Count());
    Console.WriteLine("# Anzahl hinzugefügter Objekte: " +
        modell.ObjectStateManager.GetObjectStateEntries(EntityState.Added).Count());
    Console.WriteLine("# Anzahl gelöschter Objekte: " +
        modell.ObjectStateManager.GetObjectStateEntries(EntityState.Deleted).Count());
    Console.WriteLine("# Anzahl unveränderter Objekte: " +
        modell.ObjectStateManager.GetObjectStateEntries(EntityState.Unchanged).Count());
}

static void modell_ObjectMaterialized(object sender, ObjectMaterializedEventArgs e)
{
    Console.WriteLine("  Materialisierung: " + e.Entity.ToString());
}
```

**Listing 12.38** Verwendung des Kontextklassen-Ereignisses

## Direktes Programmieren mit Entity SQL (eSQL)

Dieser Abschnitt beschäftigt sich mit der direkten Programmierung von Entity SQL, ohne Einsatz der Object Services.

### Entity SQL (eSQL)

Zur Abfrage von Informationen aus Datenquellen, die durch EDM beschrieben werden, verwendet Microsoft eine neue Erweiterung von SQL mit Namen *Entity SQL*. Die Redmonder kürzen dies mit eSQL ab, obwohl eSQL oft für einige SQL-Varianten (Embedded SQL, Extended SQL und Eiffel SQL) verwendet wird. eSQL ist syntaktisch dem klassischen SQL sehr ähnlich, bietet aber einige wesentliche Vorteile:

- eSQL arbeitet auf dem konzeptuellen Datenmodell (das mit EDM beschrieben wurde), nicht auf dem physikalischen Datenmodell
- eSQL ist DBMS-unabhängig und wird durch den ADO.NET EF Provider in DBMS-spezifisches SQL übersetzt
- eSQL erlaubt Unterabfragen an allen Stellen

**WICHTIG** eSQL bezieht sich also auf die Entitäten im Modell, nicht auf die Tabellen. Wenn man die Entitäten im Modell umbenannt hat, muss man also auch diese anderen Namen hier verwenden.

eSQL kennt nur eine SELECT-Anweisung. Andere SQL-Konzepte werden nicht unterstützt, also auch keine Datenänderungen!

## Nutzung von eSQL

Zur Nutzung von eSQL per Programmcode gibt es drei Alternativen:

- Ausführung über den ADO.NET-Datenprovider *Entity Client Data Provider* (siehe unten)
- Erstellen von Modellfunktionen (siehe unten)
- Ausführung über einen Objektcontext der Entity Framework Object Services (schon behandelt in diesem Kapitel)
- Indirekte Ausführung über LINQ to Entities. Hier wird die in die Sprachsyntax integrierte LINQ-Syntax automatisch in eSQL übersetzt (schon behandelt in diesem Kapitel)

## Entity Client Data Provider

Der *ADO.NET Entity Client Data Provider* (früher: *Map Provider*) ist ein ADO.NET-Datenbanktreiber vergleichbar mit *SQL Client* und *Oracle Client*. Der Entity Client (Namensraum `System.Data.EntityClient`) greift aber nicht direkt auf eine Datenbank zu, sondern auf ein Entity Data Model. Der Entity Client versteht nur eSQL. Das folgende Beispiel zeigt die Anwendung des Entity Clients zum Zugriff auf eine Tabelle.

```
string CS = @"metadata=res:/*/*EF_Model1.csd|res:/*/*EF_Model1.ssd|res:/*/*EF_Model1.msl;
provider=System.Data.SqlClient;provider connection string='Data Source=.\\sqlexpress;
Initial Catalog=WWings6;Integrated Security=True;MultipleActiveResultSets=True'";
// eSQL-Befehl
string ESQ = @"SELECT value f1 " +
"FROM WWings6Entities.Flug AS f1 " +
"WHERE f1.Abflugort = @Ort";
// Verbindung aufbauen
using (EntityConnection conn = new EntityConnection(CS))
{
    conn.Open();
    // Befehl ausführen
    using (EntityCommand cmd = conn.CreateCommand())
    {
        cmd.CommandText = ESQ;
        cmd.Parameters.AddWithValue("Ort", "Rom");
        using (EntityDataReader f = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
        {
            // Schleife über alle Datensätze
            while (f.Read())
            {
                Console.WriteLine("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
                    f["FlugNr"], f["Abflugort"], f["Zielort"], f["FreiePlaetze"]);
            }
        }
    }
} // Ende using
```

**Listing 12.39** Anwendung des Entity Clients zum Zugriff auf eine Tabelle

**ACHTUNG** Über den Entity Client Provider kann man nur Daten lesen. Änderungen, Ergänzungen und Löschen sind nicht möglich!

## Werkzeug eSQL Blast

*eSQL Blast* ist ein Werkzeug zum Testen von eSQL-Befehlen. Es ist weder Bestandteil des .NET Framework noch von Visual Studio, sondern ein Zusatzwerkzeug, das man bei Microsoft (siehe Website [MSDN26]) bekommt.

Zunächst muss man das Werkzeug auf ein EDM konfigurieren. Dazu benötigt die erste Registerkarte die Angabe der Verbindungszeichenfolge sowie der CSDL-, SSDL- und MSL-Datei. Die (kleine) Herausforderung besteht darin, diese Dateien einzeln zu erhalten, denn der Visual Studio-EDM-Designer legt eine *.edmx*-Datei an, in der alle drei XML-Beschreibungen zusammen enthalten sind. Der Compiler trennt zwar diese Datei auf, bündelt die Einzeldateien dann aber als Ressourcen in die Assembly ein, sodass sie von eSQL Blast dort nicht nutzbar sind.

### TIPP

Der Trick zum Erstellen getrennter Dateien für eSQL Blast besteht darin, in den Eigenschaften des Visual Studio-EDM-Designers die Eigenschaft *Metadata Artifact Processing* auf *Copy to Output Directory* zu stellen. Dann erzeugt Visual Studio im Ausgabeverzeichnis des Projekts aus der *.edmx*-Datei diese drei einzelnen Dateien.



**Abbildung 12.63** Konfiguration von eSQL Blast

Wenn das Werkzeug richtig konfiguriert ist, kann man unter der Registerkarte *Query* Befehle eingeben und das Ergebnis (inklusive des tatsächlich ausgeführten SQL-Befehls) unter *Results* ansehen.

The screenshot shows the eSqlBlast application window with the following content:

**Entity Command**

```
SELECT value fl FROM WWWingsEntitiesKompakt.FL_Fluege AS fl WHERE fl.FL_Abflugort = 'rom'
```

-- Powered by eSqlBlast;

**Store Command**

```
SELECT
[Extent1].[FL_FlugNr] AS [FL_FlugNr],
[Extent1].[FL_Abflugort] AS [FL_Abflugort],
[Extent1].[FL_Zielort] AS [FL_Zielort],
[Extent1].[FL_Datum] AS [FL_Datum],
[Extent1].[FL_NichtRaucherFlug] AS [FL_NichtRaucherFlug],
[Extent1].[FL_Plaetze] AS [FL_Plaetze],
[Extent1].[FL_FreiePlaetze] AS [FL_FreiePlaetze],
[Extent1].[FL_AnzahlStarts] AS [FL_AnzahlStarts],
[Extent1].[FL_EingerichtetAm] AS [FL_EingerichtetAm],
[Extent1].[Fl_StartZeit] AS [Fl_StartZeit],
[Extent1].[Fl_Ankunftszeit] AS [Fl_Ankunftszeit],
[Extent1].[Timestamp] AS [Timestamp]
FROM [dbo].[FL_Fluege] AS [Extent1]
WHERE [Extent1].[FL_Abflugort] = 'rom'
```

**Record Count**

54

FL_FlugNr	FL_Abflugort	FL_Zielort	FL_Datum	FL_NichtRaucherFlug	FL_Plaetze	FL_FreiePlaetze	FL_AnzahlStarts
296	Rom	Berlin	02.06.2008 23:33:26	True	250	91	NULL
297	Rom	Frankfurt	11.10.2008 14:43:26	True	250	221	NULL
298	Rom	Frankfurt	14.06.2008 00:50:26	True	250	102	NULL
299	Rom	Frankfurt	22.10.2008 16:00:26	True	250	232	NULL
300	Rom	Frankfurt	25.06.2008 02:07:26	True	250	113	NULL
301	Rom	Frankfurt	03.11.2008 17:24:26	True	250	244	NULL
302	Rom	München	06.07.2008 03:24:26	True	250	124	NULL

Connected | To execute more Entity SQL queries, go back to the Query tab.

Abbildung 12.64 Testen von eSQL-Abfragen in eSQL Blast

## Beispiele

Eine syntaktische Beschreibung von eSQL würde den Rahmen dieses Buch sprengen. Im Folgenden sind einige Beispiele mit dem zugehörigen Ergebnis wiedergegeben.

Das erste Beispiel zeigt eine einfache eSQL-Anweisung über eine einzige Entität mit Bedingungen.

```
SELECT fl.FL_FlugNr, fl.FL_AbflugOrt, fl.FL_Zielort
FROM WWWingsEntitiesKompakt.FL_Fluege AS fl
WHERE fl.FL_FlugNr < 105 AND fl.FL_FreiePlaetze > 0
```

Listing 12.40 Einfache eSQL-Abfrage

Entity Command
<pre>SELECT fl.Fl_FlugNr, fl.Fl_AbflugOrt, fl.Fl_Zielort FROM WWWIngsEntitiesKompakt.Fl_Fluege AS fl WHERE fl.Fl_FlugNr &lt; 105 AND fl.Fl_FreiePlaetze &gt; 0 ;</pre>
Store Command
<pre>SELECT 1 AS [C1], [Extent1].[FL_FlugNr] AS [FL_FlugNr], [Extent1].[FL_Abflugort] AS [FL_Abflugort], [Extent1].[FL_Zielort] AS [FL_Zielort] FROM [dbo].[Fl_Fluege] AS [Extent1] WHERE ([Extent1].[FL_FlugNr] &lt; 105) AND ([Extent1].[FL_FreiePlaetze] &gt; 0)</pre>
Record Count
5
FL_FlugNr FL_AbflugOrt FL_Zielort
100 Berlin Frankfurt
101 Berlin Frankfurt
102 Berlin Frankfurt
103 Berlin Frankfurt
104 Berlin München

**Abbildung 12.65** Ergebnis der obigen eSQL-Abfrage in eSQL Blast

Die Besonderheit von eSQL ist, dass im Gegensatz zu SQL, die Ergebnismenge auch Unterobjekte enthalten kann. Die folgende Abbildung zeigt das Ergebnis einer eSQL-Abfrage, in der in der SELECT-Anweisung Bezug auf die verbundene Entität *PS\_Passagier* genommen wird.

```
SELECT fl.Fl_FlugNr, fl.Fl_AbflugOrt, fl.Fl_Zielort, fl.PS_Passagier
FROM WWWIngsEntitiesKompakt.Fl_Fluege AS fl
where fl.Fl_FlugNr < 105 and fl.Fl_FreiePlaetze > 0;
```

**Listing 12.41** eSQL-Abfrage mit einem hierarchischen Ergebnis

FL_FlugNr	FL_AbflugOrt	FL_Zielort	PS_Passagier		
			PS_ID	PS_Kundenstatus	PS_KundenstatusSeit
100	Berlin	Frankfurt	1	1	01.06.2008 00:00:00
			2	3	01.07.2008 00:00:00
			3	4	01.01.2005 00:00:00
101	Berlin	Frankfurt	PS_ID	PS_Kundenstatus	PS_KundenstatusSeit
			1	1	01.06.2008 00:00:00
102	Berlin	Frankfurt			
103	Berlin	Frankfurt			
104	Berlin	München			

**Abbildung 12.66** Ergebnis der obigen eSQL-Abfrage mit einem hierarchischen Ergebnis

In der in eSQL eingebauten Funktion `Flatten()` kann man untergeordnete Objekte »flachklopfen«, sodass wieder eine »normale« Tabelle entsteht.



```
Flatten(SELECT value f1.PS_Passagier
FROM WWIngsEntitiesKompakt.FL_Fluege AS f1
where f1.FL_FlugNr < 105 and f1.FL_FreiePlaetze > 0 )
```

## Modellfunktionen

Neu in EF 4.0 ist die Möglichkeit, im konzeptuellen Modell Funktionen in eSQL-Syntax zu hinterlegen, die anschließend in eSQL- und LINQ to Entities-Anfragen verwendet werden können. Microsoft nennt diese Ergänzungen:

- Model Defined Functions
- Database Functions in LINQ-Abfragen
- Conceptual Model Functions und
- User Defined Functions.

**HINWEIS** Modellfunktionen sind abzugrenzen von Erweiterungen der partiellen Entitätsklasse. Erweiterungen der partiellen Entitätsklassen können im normalen Programmcode, aber nicht in LINQ to Entities- oder eSQL-Abfragen verwendet werden. Modellfunktionen hingegen sind auf die Verwendung in diesen Abfragen ausgerichtet, können optional auch im »normalen« Programmcode genutzt werden.

Modellfunktionen werden nicht im RAM ausgeführt, sondern in Datenbankfunktionen umgesetzt.

Das folgende Listing zeigt eine Modellfunktion zur Anwendung auf die Entität *Person*. Die Funktion fügt Vor- und Nachnamen zusammen.

```
<Function Name="GanzerName" ReturnType="Edm.String">
  <Parameter Name="p" Type="WWIngs6Model.Person">
  </Parameter>
  <DefiningExpression>
    Trim(p.Vorname) + " " + Trim(p.Name)
  </DefiningExpression>
</Function>
```

### Listing 12.42 Eine Modellfunktion

Die Modellfunktion kann man nicht über den Designer festlegen. Vielmehr muss man die Modellfunktion manuell in der XML-Datei eintragen. Die folgende Bildschirmabbildung zeigt die Platzierung der Modellfunktion in der XML-Datei (innerhalb des CSDL, aber nicht in einem <EntityType>-Element).

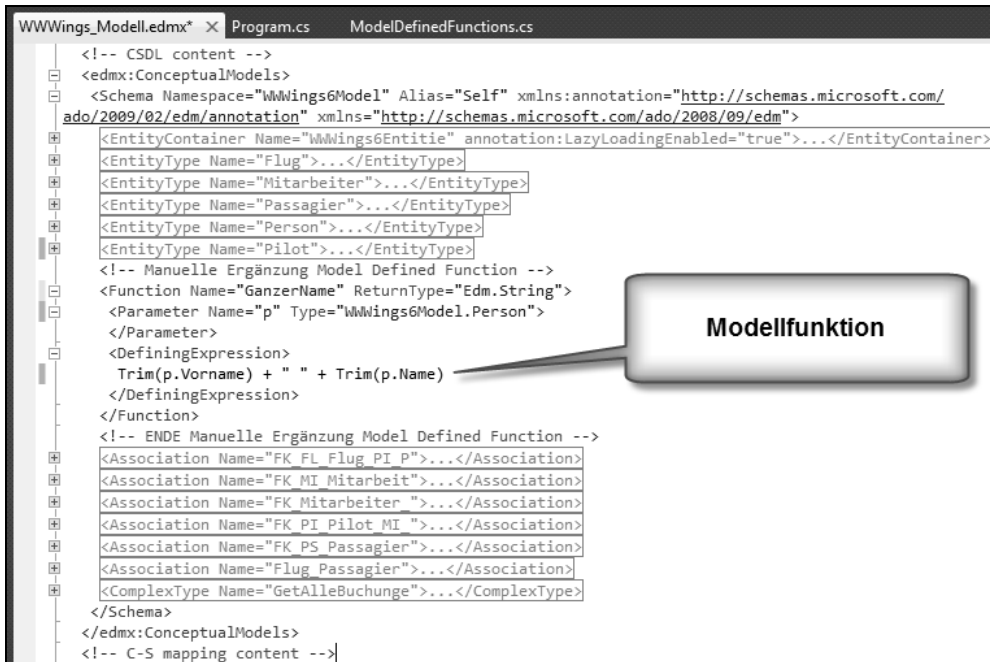


Abbildung 12.67 Platzierung der Modellfunktion in XML

Damit die Modellfunktion genutzt werden kann, muss auch noch Programmcode geschrieben werden. Es gibt leider noch keinen Generator für diesen Programmcode in Visual Studio. Dabei ist der Programmcode eigentlich immer gleich: Man schreibt eine Erweiterungsmethode für die Entitätsklasse, die genauso heißt wie die Modellfunktion. Die Methode ist mit [EdmFunction] auszuzeichnen, um den Verweis zu der Modellfunktion herzustellen. Eine Implementierung der Methode muss man nicht liefern; im Standard würde man in der Methode nur einen Fehler erzeugen, wenn sie direkt aufgerufen wird. Bei der Verwendung in Abfragen sorgt das Entity Framework für die Umleitung auf die Modellfunktion.

**TIPP** Es spricht nichts dagegen, die Methode dennoch auszuimplementieren, sodass sie auch außerhalb von Abfragen verwendet werden kann. Dann muss man aber die Logik nochmals hinterlegen (siehe folgendes Listing).

```

using System.Data.Objects.DataClasses;

namespace WWings_EF4_Standard
{
    public static class ModelDefinedFunctions
    {
        [EdmFunction("WWings6Model", "GanzerName")]
        public static string GanzerName(this Person p)
        {
            // Nochmalige Implementierung nur notwendig für Aufruf aus "normalem" Code
            return p.Vorname.Trim() + " " + p.Name.Trim();
            // Man könnte auch schreiben:
            // throw new NotSupportedException
            // ("Diese Funktion kann nur in Abfragen verwendet werden!");
        }
    }
}

```

```

    }
}
}

```

**Listing 12.43** Implementierung der Modellfunktion als Erweiterungsmethode für die Entitätsklasse

Das nachstehende Listing zeigt die Verwendung der Modellfunktion in einer LINQ to Entities-Abfrage und – da sie auch im Programmcode ausimplementiert ist – in einer foreach-Schleife.

```

/// <summary>
/// Einsatz einer Modellfunktion
/// </summary>
public static void MDFDemo()
{
    WWings_EF4_Standard.WWings6Entities modell = new WWings_EF4_Standard.WWings6Entities();
    var Personen = from p in modell.Person where p.GanzerName().Length > 15 select p;
    foreach (var pers in Personen)
    {
        Console.WriteLine(pers.GanzerName());
    }
    modell.Dispose();
}

```

**Listing 12.44** Nutzung der Modellfunktion

Das letzte Listing in diesem Abschnitt zeigt die Umsetzung der Abfrage in SQL.

```

SELECT
[Extent1].[PersonID] AS [PersonID],
[Extent1].[Name] AS [Name],
[Extent1].[Vorname] AS [Vorname],
[Extent1].[Land] AS [Land],
[Extent1].[Geburtstag] AS [Geburtstag],
[Extent1].[Foto] AS [Foto],
[Extent1].[EMail] AS [EMail],
[Extent1].[Stadt] AS [Stadt],
[Extent1].[Memo] AS [Memo]
FROM [dbo].[Person] AS [Extent1]
WHERE ( CAST(LEN(LTRIM(RTRIM([Extent1].[Vorname])) + ' ' + LTRIM(RTRIM([Extent1].[Name])))) AS int)) > 15

```

**Listing 12.45** SQL-Repräsentation der Abfrage mit Modellfunktion GanzerName()

## Leistungsüberlegungen

Dieses Kapitel widmet sich Leistungsfragen, die sicherlich in vielen Situationen von Bedeutung sind bei der Entscheidung, klassisches ADO oder ADO.NET Entity Framework zu verwenden.

### TIPP

Weitere Leistungsmessungen zu verschiedenen ORM-Werkzeugen finden Sie unter [ORMBATTLE01].



## Ladegeschwindigkeit

Die Messung der Geschwindigkeit einer Datenzugriffsbibliothek ist ein komplexes Unterfangen. Die folgenden Messungen mit ausgewählten Szenarien können Anhaltspunkte geben, aber keineswegs eigene Messungen am konkreten Fall ersetzen.

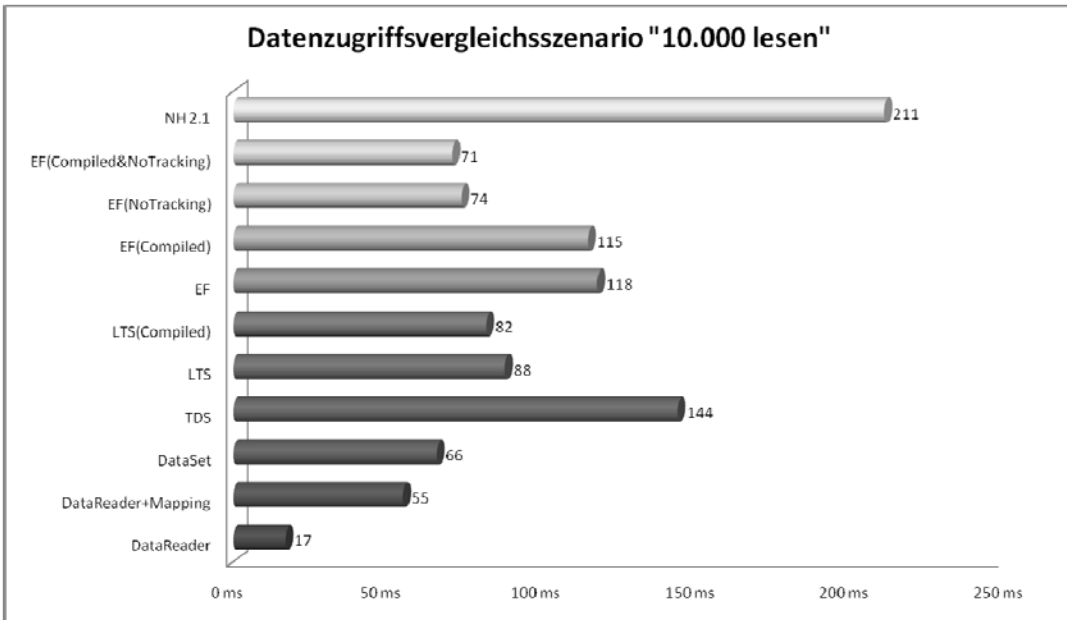
Für die Beurteilung der Lesegeschwindigkeit wird die Tabelle *Flug* in der Datenbank World Wide Wings in der Version 6.3 verwendet.

- Die Flugtabelle der WorldWideWings-Datenbank wird vorab mit 10.000 Datensätzen befüllt
- Eine Ausgabe oder Weiterverarbeitung findet nicht statt, da jegliche weitere Aktion die Aussage verfälschen würde
- Vor der ersten Messung findet ein unbewerteter Zugriff statt, um den SQL Server aus dem Dornröschenschlaf zu erwecken
- Das Laden und Durchlaufen wird zehn Mal nacheinander wiederholt
- Das Messergebnis ist der Durchschnitt über alle zehn Messungen
- Der Client ist ein Konsolenclient (Windows 7, .NET 4.0)
- Der Server läuft auf einem anderen Rechner (Windows Server 2008 R2) im lokalen Netzwerk (Gigabit-Ethernet, Abendstunden ohne andere nennenswerte Last)
- Verglichen werden jeweils `DataReader` mit SQL-Abfrage, `DataReader` mit SQL-Abfrage und im Code hinterlegten Mapping auf Objekte, `DataSet` mit SQL-Abfrage, Typisiertes `DataSet` (TDS) mit SQL-Abfrage, LINQ to SQL (LTS) als kompilierte und nicht-kompilierte LINQ-Abfrage, Entity Framework (EF) als kompilierte und nicht-kompilierte LINQ-Abfrage und NHibernate (Version 2.1) mit HQL-Abfrage

Im ersten Szenario werden alle 10.000 Flüge in einer Abfrage geladen und in einer Schleife durchlaufen.

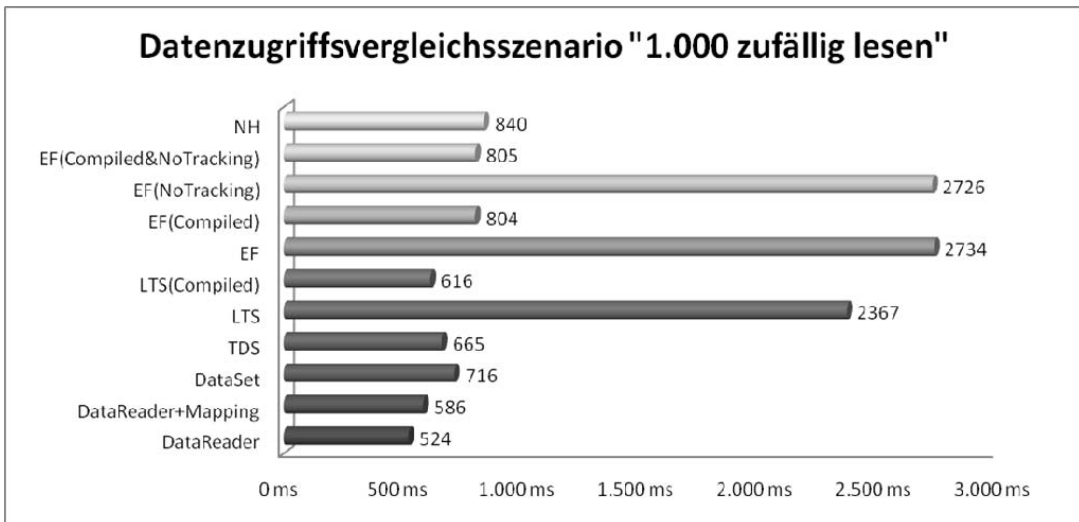
Erwartungsgemäß ist der `DataReader` das bei Weitem schnellste Instrument zum Lesen von Daten und alle abstrakteren Instrumente sind langsamer, denn sie basieren intern alle auf dem `DataReader`. Man darf beim Betrachten der Messergebnisse aber nicht vergessen: Die Dauer ist in Millisekunden angegeben. Bedeutet also: Das Entity Framework liest 10.000 Objekte über ein Gigabit-Netzwerk in rund 150-200 Millisekunden.

Bei diesem Szenario lässt sich eine Entity Framework-Abfrage vor allem durch den Verzicht auf die Änderungsverfolgung (Change Tracking) optimieren.



**Abbildung 12.70** Leistungsmessung: Lesezenario 1 (Angaben in Millisekunden)

Im zweiten Szenario liegt die Optimierung hingegen in der Vorkompilierung der Abfrage. In diesem Szenario wird 1.000-mal ein gleichartiger Befehl an die Datenbank gesendet. Wenn man die Abfrage in LINQ oder Entity SQL direkt im Programmcode hinterlegt, erfolgt 1.000-mal eine Umwandlung in datenbankspezifisches SQL. Diese Zeit kann man sich sparen durch das »Vorkompilieren« der Abfrage, was in diesem Fall der Umwandlung in SQL entspricht.



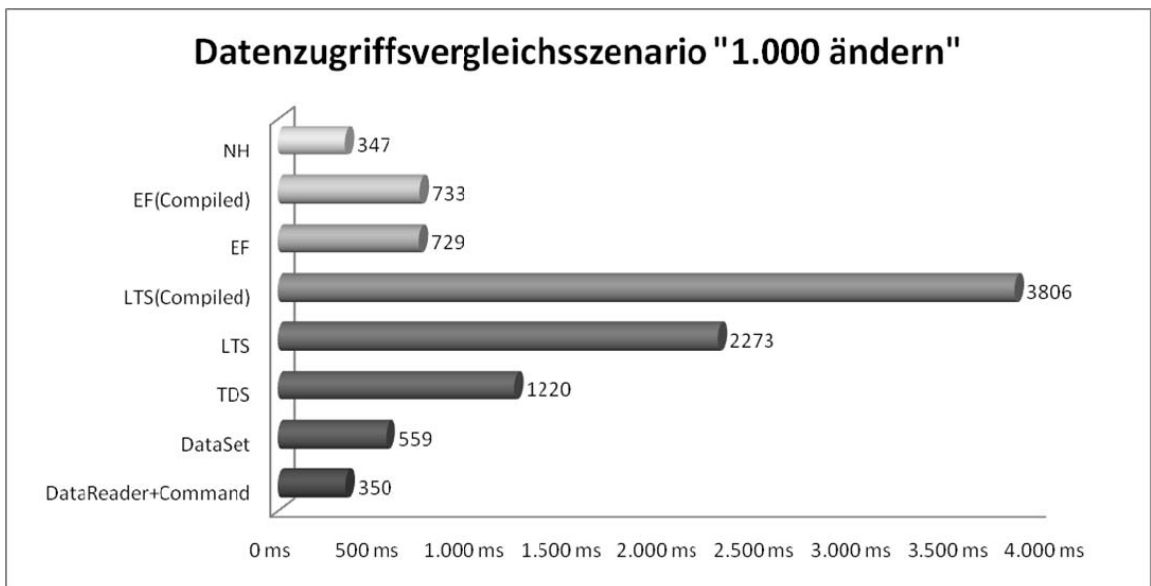
**Abbildung 12.71** Leistungsmessung: Lesezenario 2 (Angaben in Millisekunden)

## Speichergeschwindigkeit

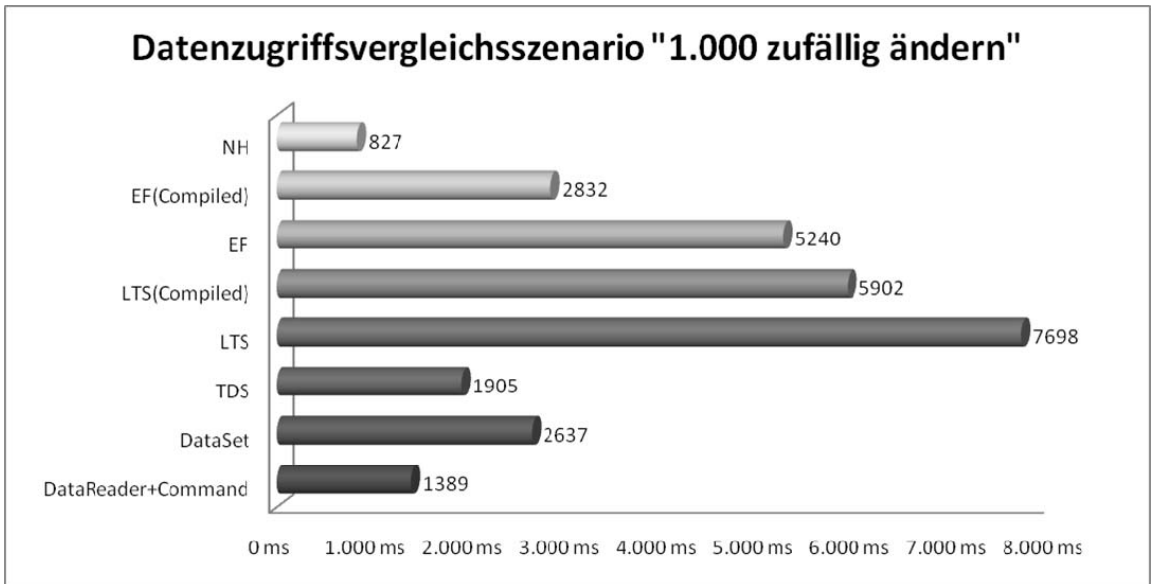
Bei den Messungen der Speichergeschwindigkeit wurde die gleiche Tabelle wie beim Lesen verwendet.

Beim Szenario »1.000 ändern« lädt man mit einem SQL-Befehl 1.000 Datensätze und speichert Änderungen an allen geladenen Datensätzen, was zu 1.000 Update-Befehlen führt. Beim Szenario »1.000 zufällig ändern« lädt man wieder 1000 einzelne Datensätze zufällig und speichert jeweils eine Änderung (also ebenfalls 1.000 Update-Befehle).

Im Fall des DataReader muss natürlich ein selbstbefülltes Command-Objekt zur Aktualisierung erhalten. Die Erkenntnisse sind interessant: Der relative Leistungsverlust von Entity Framework gegenüber dem DataReader/Command sind geringer als beim reinen Lesen. Und: LINQ to SQL versagt beim Speichern in Hinblick auf die Leistung völlig. Im zweiten Szenario wirkt sich die Vorkompilierung von LINQ-Befehlen natürlich wieder positiv aus.

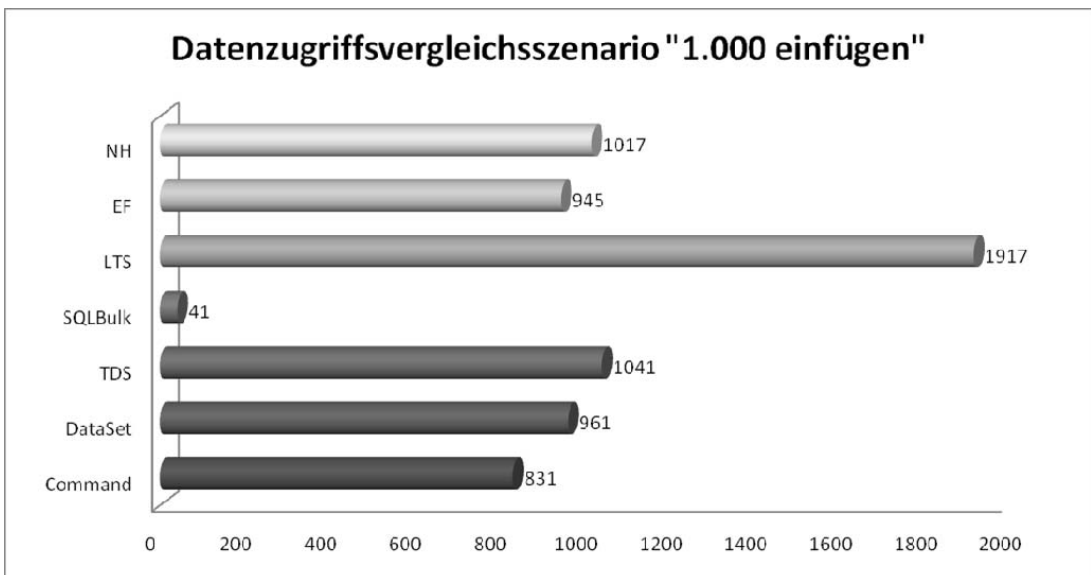


**Abbildung 12.72** Leistungsmessung: Ändern-Szenario 1



**Abbildung 12.73** Leistungsmessung: Ändern-Szenario 2

Das letzte Szenario (mehr Platz ist leider hier im Buch nicht) ist der Fall des Einfügens von 1.000 Datensätzen. Hier ist ein Command-Objekt nicht die schnellste Lösung, sondern der Bulkimport-Mechanismus des Microsoft SQL Server (vgl. Kapitel zu »ADO.NET«).



**Abbildung 12.74** Ergebnis der Geschwindigkeitsmessungen



## Weitere Funktionen

EF bietet einige weitere Funktionen, die hier aus Platzgründen nicht mehr genauer besprochen werden:

- Optimierung der Objektkontextinstanzierungsdauer durch so genannte View Generation mit *EdmGen.exe*
- Unterstützung für Vererbung mit Filtered Mapping, Vertical Mapping und Horizontal Mapping
- Generierung eines EF-Kontextes und der Entitätsklassen an der Kommandozeile mit *EdmGen.exe*

## Verbliebene Schwächen

Man muss aber auch erwähnen, dass nicht alle Probleme aus EF 1 im EF4 gelöst sind. Nur einige der verbliebenen Schwächen seien hier erwähnt:

- Das EF unterstützt kein Mapping allein durch Annotationen, sondern nur XML-basiertes Mapping
- Der EDM-Designer unterstützt kein Ziehen & Fallenlassen von Tabellen aus dem Server Explorer
- Die Generierung von Entitätsklassen für Datenbanksichten (Views) erfordert weiterhin manuelle Nachbearbeitung, denn der Assistent denkt sich sehr merkwürdige Primärschlüssel für Views aus. Alle Views sind ohne manuelle Nachbearbeitung im XML nicht aktualisierbar.
- Leider wird im Designer die gewählte Pluralisierungsoption nicht beachtet, wenn man nachträglich mit *Update Modell* Tabellen hinzufügt. Hier wird der Name immer 1:1 übernommen.
- In der Datenbank hinterlegte Standardwerte für Spalten (Default Values) werden weiterhin nicht in das Modell übernommen. Man muss diese manuell im Modell nachpflegen.
- Es gibt keine Hilfe im Designer für Modelle, die mit mehreren verschiedenen Datenbankmanagementsystemen zusammenarbeiten können
- Im Entity Framework darf es keine Spalte geben, die so heißt wie eine Tabelle
- Die Klasse `EntityCollection<T>`, die 1:N-Beziehungen zwischen Entitäten (im Fall der Standardcodegenerierung) erzeugt, implementiert nicht `INotifyCollectionChanged`. Microsoft hat diese Funktion für EF 4.0 nicht mehr geschafft (»Implementing `INotifyCollectionChanged` on `EntityCollection<T>` is something that has been on our priority list along with many other improvements to the control data binding experience using Entity Framework classes. Unfortunately this support is not going to make it into Visual Studio 2010 and .NET 4.0, but we plan to revisit these requests for a future release.»). [CONNECT01]
- Das partielle Befüllen von Entitätsobjekten und automatisches Nachladen einzelner Attribute ist nicht möglich
- Es gibt keine partiellen Methoden oder Ereignisse, mit denen man sich über den Lebenszyklus einer Instanz eines Entitätsobjekts informieren lassen kann (Lebenszyklusereignisse). In LINQ to SQL gibt es hier zumindest die Methoden `OnLoaded()`, `OnValidate()` und `OnCreated()`.
- Unterstützung für *Table Valued Functions* (TVF) ist nicht vorhanden
- Das Speichern einzelner Änderungen unabhängig von anderen Änderungen (mehr Optionen für die Zwischenspeicherung) ist nicht möglich

- Vertikales und horizontales Vererbungsmapping ist nicht möglich, wenn die Primärschlüsselspalten nicht gleich sind
- Rein codebasiertes Mapping ist nicht enthalten (eine entsprechende Funktion war unter dem Namen *Code Only* in Arbeit, ist dann aber nicht mit Entity Framework 4 erschienen. Es gibt dazu zum Redaktionsschluss eine Alpha-Version [MSDN35].)

## Vergleich mit anderen ORM-Werkzeugen für .NET

Die folgende Tabelle zeigt den kompakten Vergleich von ADO.NET Entity Framework mit LINQ to SQL.

	LINQ to SQL	ADO.NET Entity Framework
<b>.NET-Version</b>	3.5/4.0 (keine neue Funkt.)	3.5 mit Service Pack 1/4.0 (viele neue Funktionen)
<b>Datenbanken</b>	Microsoft SQL Server	Viele Datenbanksysteme
<b>Mapping</b>	1:1-Mapping und Filtered Mapping für Vererbung	1:1, 1:N; M:N sowie Vererbung durch Filtered Mapping, Vertical Mapping und Horizontal Mapping
<b>Abfragesprachen</b>	LINQ, SQL	LINQ to Entities, eSQL, SQL
<b>Reverse Engineering</b>	Ja	Ja
<b>Forward Engineering</b>	Ja	Ja
<b>Serialisierung</b>	Einfach	Komplette Objektbäume
<b>Nachladen von Objekten</b>	transparent	explizit oder transparent
<b>POCOs</b>	tlw.	Ja
<b>Flexibilität</b>	0	+
<b>Unterstützung für verteilte Anwendungen</b>	-	+ Self Tracking Entities
<b>Leistung Lesen</b>	++	+
<b>Leistung Schreiben</b>	0	++
<b>Zukunftssicherheit</b>	0	++

**Tabelle 12.4** LINQ to SQL versus ADO.NET Entity Framework

Der Autor dieses Buch hat auch einen sehr detaillierten Vergleich von einigen ORM-Werkzeugen für .NET erstellt:

- LINQ to SQL
- ADO.NET Entity Framework
- NHibernate

- Telerik Open Access
- Genome
- .NET Data Objects (NDO)

Aus Platzgründen kann diese Tabelle (elf DIN-A4-Seiten) hier leider nicht vollständig abgedruckt werden. Sie erhalten die vollständige Tabelle von der Leser-Website (Zugang siehe Vorwort).

<b>Comparison of Object Relational Mapping Tools for the .NET Framework</b> <small>Author: Dr. Holger Schwichtenberg, <a href="http://www.IT-Visions.de">www.IT-Visions.de</a>  Version 2.0 BETA, 25. September 2009</small>							
	Typed DataSet (TDS)	LINQ-to-SQL (formerly "DLINQ")	ADO.NET Entity Framework Object Services	Telerik OpenAccess (VOA)	NHibernate	.NET Data Objects (NDO)	Genome
0/1:1 BO Associations	Y	Y	Y	Y	Y	Y	Y
0/1:N BO Associations	N	N	Y	Y	Y	Y	Y
0/N:M BO Associations	N	N	Y	Y	Y	Y	Y
Requirements for 0/1:1 Associations	Y, DataRow	Y, EntityRef	N / EntityReference	N (POCO)	N (POCO)	N	N
Support collection types	DataTable	EntitySet	EntityCollection	ArrayList, IList, ICollection, IDictionary, TrackedList<T>, TrackedBindingList<T>	IList, ISet, IDictionary, ICollection	IList	IEnumerable<T>, ICollection<T>
Code-based Mapping (based on Annotations)	N	Y	N	N	Y (optional, with NHibernate Mapping Attributes)	Y (optional – creates mapping file)	N
XML-based Mapping	Y	Y	Y	Y	Y	Y	Y
Mapping File Generation	Y	Y	Y	Y	Y	Y	Y
Compile time processing and error checking of mapping	Y	N	N	Y (during schema update)	N	N	Y
Filtered Mapping, Inheritance	N	Y	Y	Y	Y	N	Y

**Abbildung 12.75** Nur ein kleiner Ausschnitt aus der umfangreichen Vergleichstabelle

**WICHTIG** Bitte beachten Sie folgenden wichtigen Hinweis zum Ursprung der Daten in der Tabelle: Die Angaben zu LINQ to SQL, ADO.NET Entity Framework und NHibernate wurden vom Autor des Beitrags erhoben. Die Angaben zu Telerik Open Access, NDO und Genome wurden von den Herstellern geliefert. Der Autor hat nur die Angaben von Telerik Open Access zum Teil an eigenen Projekten verifiziert. Der Autor hat NDO und Genome nicht im Einsatz und kann die Angaben der Hersteller hier nicht überprüfen.

## Fazit

In den Jahren 2008 und 2009 war die Entscheidung für die Datenzugriffsstrategie noch schwer. Bleibt man bei klassischem ADO.NET? Oder nutzt man die Vorteile von ORM und LINQ beim Datenbankzugriff? LINQ to SQL oder ADO.NET Entity Framework?

Inzwischen hat es sich doch weitestgehend geklärt: Microsoft hat eine klare Aussage zugunsten von ADO.NET Entity Framework getroffen. Das ADO.NET Entity Framework 4.0 ist wesentlich reifer. Es hat viele (wenn auch leider nicht wirklich alle) Funktionen bekommen, die LINQ to SQL bisher exklusiv vorbehalten waren. LINQ to SQL ist für viele daher kein Thema mehr. Auch der Umstieg von LINQ to SQL auf ADO.NET Entity Framework 4.0 ist machbar: Ein neues Modell und eine *Suchen-und-Ersetzen*-Migration sind mit überschaubarem Aufwand machbar.

Das ADO.NET Entity Framework ist in Version 4.0 deutlich reifer geworden. Es bleibt für viele die Frage, ob es reif genug ist.

Das ist eine Frage, die der Autor dieses Buchs nicht pauschal beantworten kann, denn es hängt von vielen Faktoren ab (Anwendungsszenario, verwendete Datenbanksysteme, Datenmengen, Art der Abfrage, Leistungserwartungen, Programmierstil, u.v.m.).

Der Autor dieses Buchs hält es in den Projekten in seiner Firma so: Im Standard wird alles mit Entity Framework gemacht, denn die Produktivität von Entity Framework im Vergleich zu klassischem ADO.NET ist überragend.

Dort, wo die Geschwindigkeit des Entity Framework nicht ausreicht, werden klassische Techniken (DataReader und SQL Server Bulkimport) eingesetzt. Nach diesem Vorgehensmodell konnten schon einige Projekte gestemmt werden, auch ein Projekt mit über einer Millionen neuen Datensätzen pro Tag!

Warum nicht NHibernate? Weil es dort lange Zeit an der LINQ-Unterstützung fehlte und es dort immer noch keinen grafischen Designer gibt. NHibernate bietet zwar mehr Funktionen als Entity Framework, aber die Produktivität ist geringer.

Für kommende Projekte in Frage kommt noch Telerik Open Access, die seit 2010 neben der LINQ-Unterstützung auch einen Designer für Visual Studio haben, der sehr ähnlich dem Entity Framework-Designer ist.

Andere ORM außer den genannten werden es im Markt schwer haben, dauerhaft zu überleben.