

HANSER

Leseprobe

Fabrice Marguerie, Steve Eichert, Jim Wooley

LINQ im Einsatz

Übersetzt aus dem Englischen von Walter Doberenz

ISBN: 978-3-446-41429-7

Weitere Informationen oder Bestellungen unter

<http://www.hanser.de/978-3-446-41429-7>

sowie im Buchhandel.

Erweiterte LINQ to SQL-Features

In den beiden Vorgängerkapiteln haben wir Ihnen die wichtigsten Komponenten für das Arbeiten mit relationalen Daten unter LINQ to SQL vorgestellt.

In diesem Kapitel werden wir die Basiskonzepte weiter ausbauen und einige der fortgeschrittenen Features von LINQ to SQL kennen lernen. Wir beginnen mit der Fortsetzung unserer Diskussion über den Lebenszyklus von Objekten, wobei wir uns auf das Konkurrenzverhalten und auf Transaktionsprobleme konzentrieren werden. Danach untersuchen wir, wie wir noch direkter mit der Datenbank arbeiten können und nutzen dazu die speziellen Funktionalitäten des SQL Servers. Dann verlassen wir die Datenzugriffsschicht und schauen nach den Optionen, die uns LINQ to SQL zur Anpassung der Geschäftsschicht bereitstellt, das Vorkompilieren von Abfrageausdrücken inbegriffen. Wir verwenden partielle Klassen und Polymorphie, die immer mit Vererbung verbunden ist. Schließlich werfen wir noch einen Blick auf das kommende Entity Framework als eine Alternative zu LINQ to SQL beim Zugriff auf relationale Daten.

8.1 Umgang mit simultanen Änderungen

Beim Entwurf von Systemen für einzelne User muss sich der Entwickler keine Gedanken darüber machen, wie die von einer Person vorgenommenen Änderungen die Änderungen durch andere Personen beeinflussen können. Für ein Produktionssystem ist es allerdings ungewöhnlich, wenn es nur von einem User benutzt werden darf. Wenn das System wächst und mehrere Nutzer ermöglicht, müssen wir mit der Situation rechnen, dass zwei Nutzer zur gleichen Zeit denselben Datensatz verändern. Allgemein gibt es dafür zwei Strategien: pessimistische Konkurrenz, welche für einen zweiten User solange das Ändern von Datensätzen sperrt, bis der erste User die Sperre wieder aufhebt, und optimistische Konkurrenz, welche es zwei Usern erlaubt, Änderungen vorzunehmen. Im Fall optimistischer Konkurrenz muss der Anwendungsentwickler entscheiden, ob die Werte des ersten Users, die Werte des zweiten Users oder die des letzten Updates erhalten werden, oder ob die Werte beider User auf irgendeine Weise gemischt werden. Jede dieser Strategien bietet verschiedene Vor- und Nachteile.

8.1.1 Pessimistische Konkurrenz

In den Zeiten vor .NET unterstützten viele Applikationen offene Verbindungen zur Datenbank. Für diese Systeme schrieben die Entwickler häufig Applikationen, bei denen die in Bearbeitung befindlichen Datensätze für andere Benutzer gesperrt wurden. Diesen Sperrmechanismus nennt man pessimistische Konkurrenz. Kleinere Windows-basierte Applikationen, die dieses Prinzip verwenden, arbeiteten in der Regel problemlos. Als man diese Systeme aber auf immer mehr User erweiterte, führte dieser Sperrmechanismus zum Blockieren des Systems.

Zur gleichen Zeit, als diese Skalierungsprobleme immer gravierender hervortraten, begann man mit der Entwicklung zustandsloser, web-basierter Architekturen, die die herkömmlichen Client-Server Architekturen ablösen und das Verteilen von Anwendungen erleichtern sollten. Die Anforderungen zustandsloser Web Applikationen erzwangen allerdings den Verzicht auf langandauernde pessimistische Sperren.

Im Bestreben, die Entwickler von den mit Skalierung und Datensatzsperrern verbundenen Problemen des pessimistischen Konkurrenzmodells fernzuhalten, wurde das .NET Framework konsequent auf die verbindungslose Natur web-basierter Applikationen ausgerichtet. ADO.NET, die Datenzugriffs-API von .NET, wurde deshalb nicht mit Cursor-Fähigkeiten ausgestattet. Damit hatte sich auch automatisch die Option einer pessimistischen Konkurrenz erledigt. Applikationen können aber weiterhin damit entwickelt werden, wenn die Datensätze ein "checked out"-Flag erhalten. Dieses wird geprüft, wenn man nachfolgend versucht, auf dieselben Datensätze zuzugreifen. Allerdings werden diese CheckedOut Flags häufig nicht zurückgesetzt, sodass es schwierig ist zu ermitteln, wann der User den Datensatz nicht mehr verwendet. Wegen dieser Probleme hat das pessimistische Konkurrenzmodell in einer verbindungslosen Umgebung weitgehend seine Bedeutung verloren.

8.1.2 Optimistische Konkurrenz

Wegen der in einer verbindungslosen Umgebung auftretenden Probleme verwendet man typischerweise eine alternative Strategie, die optimistische Konkurrenz. Dieses Modell erlaubt es jedem User, Änderungen an seinen Datensatzkopien vorzunehmen. Sollen die Werte dann gespeichert werden, prüft das Programm die vorhergehenden Werte um festzustellen, ob sie geändert wurden. Falls das nicht zutrifft, wird der Datensatz als nicht gesperrt betrachtet und gespeichert. Anderenfalls tritt ein Konflikt auf und das Programm muss wissen, ob es die Änderungen automatisch überschreiben, die neuen Änderungen wegwerfen oder beides irgendwie zusammenfassen soll.

Die erste Hälfte zur Bestimmung der optimistischen Konkurrenz ist relativ einfach. Ohne eine Konkurrenzüberprüfung würde das gegen die Datenbank abgesetzte SQL-Statement etwa die folgende Syntax haben: `UPDATE TABLE SET [field = value] WHERE [Id = value]`. Um optimistische Konkurrenz hinzuzufügen, muss die WHERE-Klausel so erweitert werden, dass darin nicht nur der Wert der ID-Spalte enthalten ist, sondern dass dort auch die Originalwerte jeder Tabellenspalte verglichen werden. Listing 8.1 demonstriert ein SQL-Statement, welches die *Book*-Tabelle unseres durchgängigen Beispiels auf optimistische Konkurrenz überprüft.

Listing 8.1

SQL Update-Statement für optimistische Konkurrenz der Book-Tabelle

C#

```
UPDATE dbo.Book
SET Title = @NewTitle,
    Subject = @NewSubject,
    Publisher = @NewPublisher,
    PubDate = @NewPubDate,
    Price = @NewPrice,
    PageCount = @NewPageCount,
    Isbn = @NewIsbn,
    Summary = @NewSummary,
    Notes = @NewNotes
WHERE ID = @ID AND Title = @OldTitle AND
    Subject = @OldSubject AND
    Publisher = @OldPublisher AND
    PubDate = @PubDate AND
    Price = @Price AND
    PageCount = @PageCount AND
    Isbn = @OldIsbn AND
    Summary = @OldSummary AND
    Notes = @OldNotes
RETURN @@RowCount
```

Um uns vom Erfolg der Update-Operation zu überzeugen, wollen wir mit Listing 8.1 einen Datensatz aktualisieren und den *RowCount* überprüfen. Falls 1 zurückgegeben wird wissen wir, dass die Originalwerte unverändert sind und dass das Update funktionierte. Bei Rückgabe von 0 wissen wir, dass irgendjemand zumindest einen der Werte geändert hat, seitdem das letzte Mal darauf zugegriffen wurde, weil wir keinen Datensatz finden können, der noch den gleichen Wert wie beim erstmaligen Laden hat. In diesem Fall wurde der Datensatz nicht aktualisiert. An diesem Punkt können wir den User darüber informieren, dass es einen Konflikt gab und dass die Konkurrenzverletzung entsprechend zu behandeln ist. Eine Behandlung von Konkurrenzproblemen ist in LINQ to SQL eingebaut.

Die Konfiguration von Klassen für die Unterstützung der optimistischen Konkurrenz ist extrem einfach. Tatsächlich haben wir bereits beim Erstellen der Mappings für Tabellen und Spalten die entsprechenden Einstellungen für optimistische Konkurrenz getroffen. Beim Aufruf von *SubmitChanges* implementiert der Datenkontext automatisch das optimistische Konkurrenzmodell. Um den SQL-Code für ein einfaches Update zu zeigen, wollen wir ein Beispiel betrachten, in welchem wir die teuersten Bücher um 10% billiger machen (siehe Listing 8.2).

Listing 8.2

Standardmäßige Implementierung von Konkurrenz mit LINQ to SQL

C#

```
Ch8DataContext context = new Ch8DataContext()
Book mostExpensiveBook = (from book in context.Books
    orderby book.Price descending
    select book).First();
decimal discount = .1M;
mostExpensiveBook.Price -= mostExpensiveBook.Price * discount;
context.SubmitChanges();
```

Das produziert den SQL Code zum Selektieren der Bücher, genauso wie das folgende SQL zum Aktualisieren:

```
UPDATE [dbo].[Book]
SET [Price] = @p8
FROM [dbo].[Book]
WHERE ([Title] = @p0) AND ([Subject] = @p1) AND ([Publisher] = @p2)
      AND ([PubDate] = @p3) AND ([Price] = @p4) AND ([PageCount] = @p5)
      AND ([Isbn] = @p6) AND ([Summary] IS NULL) AND ([Notes] IS NULL)
      AND ([ID] = @p7)
```

Beim Aufruf von *SubmitChanges* auf dem *DataContext* wird das *Update* Statement generiert und an den Server abgesetzt. Wird kein passender Datensatz gefunden, basierend auf den Vorgängerwerten in der WHERE-Klausel, so stellt der Kontext fest, dass keine Datensätze durch dieses Statement beeinflusst werden und löst eine *ChangeConflictException* aus.

Situationsabhängig kann die Anzahl von Parametern, wie sie für das Implementieren der optimistischen Konkurrenz benötigt werden, Performance-Probleme nach sich ziehen. In diesen Fällen sollten wir unser Mapping verfeinern und nur die Felder ausweisen die absichern, dass sich die Werte nicht geändert haben. Wir können dies durch Setzen des *UpdateCheck*-Attributs tun. Standardmäßig ist *UpdateCheck* auf *Always* gesetzt, d.h., LINQ to SQL wird diese Spalte immer auf optimistische Konkurrenz überprüfen. Alternativ können wir nur prüfen lassen, wenn sich der Wert geändert hat (*WhenChanged*) oder gänzlich darauf verzichten (*Never*).

Wollen Sie wirklich die Leistungsfähigkeit des *UpdateCheck*-Attributs ausnutzen und haben Sie die Möglichkeit zum Verändern des Tabellenschemas, so könnten Sie eine *RowVersion* oder *TimeStamp*-Spalte zu jeder Tabelle hinzufügen. Die Datenbank wird den Wert von *RowVersion* jedes Mal automatisch aktualisieren, wenn sich der Datensatz geändert hat. Konkurrenzüberprüfungen werden nur über der Kombination der Versions- und ID-Spalten ausgeführt. Alle anderen Spalten werden auf *UpdateCheck=Never* gesetzt. Wir haben das bereits bei der *Author*-Klasse in Kapitel 7 angewendet. Listing 8.3 illustriert die überarbeitete *Author*-Klasse, wobei dieselben Änderungen wie im Vorgängerbeispiel vorgenommen wurden. Die Verwendung der *TimeStamp*-Spalte erlaubt eine optimierte WHERE-Klausel im *Update*-Statement.

Listing 8.3

Optimistische Konkurrenz von Authors mit TimeStamp-Spalte**C#**

```
Ch8DataContext context = new Ch8DataContext();
Author authorToChange = (context.Authors).First();

authorToChange.FirstName = "Jim";
authorToChange.LastName = "Woolley";

context.SubmitChanges();
```

Das ergibt den folgenden SQL-Code:

```
UPDATE [dbo].[Author]
SET [LastName] = @p2, [FirstName] = @p3
FROM [dbo].[Author]
WHERE ([ID] = @p0) AND ([TimeStamp] = @p1)
```

```
SELECT [t1].[TimeStamp]
FROM [dbo].[Author] AS [t1]
WHERE ((@ROWCOUNT) > 0) AND ([t1].[ID] = @p4)
```

In Ergänzung zur standardmäßigen optimistischen Konkurrenz sind noch einige andere Konkurrenzmodelle verfügbar. Die erste Option besteht einfach im Ignorieren aller Konkurrenzzugriffe und im bedingungslosen Aktualisieren der Datensätze, wobei immer das letzte Update akzeptiert wird. In diesem Fall ist *UpdateCheck* für alle *Properties* auf *Never* zu setzen. Empfehlenswert ist diese Lösung nicht, es sei denn Sie können garantieren, dass Konkurrenz in Ihrer Applikation kein Problem sein wird. In den meisten Fällen ist es das Beste, den User über den Konflikt zu informieren und Optionen für das Bereinigen der Situation bereitzustellen.

In manchen Fällen ist es empfehlenswert, wenn Sie es zwei Usern erlauben, in derselben Tabelle Änderungen an verschiedenen Spalten vorzunehmen. In diesem Fall setzen Sie das *UpdateCheck*-Attribut auf *WhenChanged* anstatt auf *Always*.

Natürlich ist das nicht immer empfehlenswert, speziell dann nicht, wenn mehrere Felder dazu beitragen einen Endwert auszurechnen. So verfügt zum Beispiel eine typische *OrderDetail*-Tabelle über Spalten für Anzahl, Preis und Gesamtpreis. Wenn eine Änderung entweder bei der Anzahl oder beim Preis vorgenommen wurde, muss auch der Gesamtpreis angepasst werden. Wenn nun ein User die Anzahl ändert und ein anderer den Preis, wird sich auch der Gesamtpreis entsprechend ändern. Auch diese Form der automatischen Konkurrenzverwaltung hat ihren Platz. Letztendlich entscheiden die konkreten Geschäftsanforderungen über die Anwendbarkeit.

Mit LINQ to SQL kann das Überprüfen der Konkurrenz auf Basis des Feldniveaus realisiert werden. Das Framework wurde so entwickelt, dass es eine große Flexibilität für die verschiedensten benutzerspezifischen Implementierungen bietet. Das standardmäßige Verhalten basiert auf der vollen Unterstützung des optimistischen Konkurrenzmodells.

Bis jetzt wissen wir lediglich, wie man einen Konflikt erkennen kann und wo er auftritt. Im Folgenden wollen wir darüber diskutieren, was man mit diesem Wissen alles anfangen kann.

8.1.3 Behandlung von Konkurrenz-Ausnahmen

Beim Verwenden der *Always*- oder *WhenChanged*-Optionen für *UpdateCheck* ist es unvermeidbar, dass zwei User dieselben Werte verändern wollen und dadurch Konflikte verursachen. In diesen Fällen wird der *DataContext* eine *ChangeConflictException* auslösen, sobald der zweite User eine *SubmitChanges*-Anforderung absetzt. Wegen der großen Wahrscheinlichkeit, in eine solche Ausnahme hineinzulaufen, müssen wir den Aktualisierungsvorgang in einen Fehlerbehandlungsblock einschließen.

Wurde die Ausnahme ausgelöst, gibt es verschiedene Optionen zu ihrer Behandlung. Der *DataContext* hilft nicht nur beim Aufspüren der konfliktverursachenden Objekte, sondern auch mit Eigenschaften, die zwischen Originalwert, geändertem Wert und aktuellem Wert in der Datenbank unterscheiden. Um diese Informationen bereitzustellen, können wir den *RefreshMode* spezifizieren, um zu festzulegen, ob der kollidierende Datensatz zunächst wieder von der Datenbank aufgefrischt werden soll, um die aktuellen Werte zu bestimmen. Haben wir die aufgefrischten Werte, können wir uns entscheiden, ob wir die Originalwerte, die aktuellen Datenbankwerte oder unsere neuen

Werte behalten wollen. Nehmen wir letztere Option, so behandeln wir den Änderungskonflikt auf dem Kontextobjekt, wobei wir festlegen, dass wir die Änderungen behalten wollen. Listing 8.4 illustriert einen solchen typischen *try-catch*-Block.

Listing 8.4

Auflösen von Änderungskonflikten mit KeepChanges**C#**

```
try
{
    context.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException)
{
    context.ChangeConflicts.ResolveAll(RefreshMode.KeepChanges);

    context.SubmitChanges();
}
```

Wenn wir die *KeepChanges*-Option verwenden, brauchen wir uns die geänderten Werte nicht anzusehen. Wir behaupten einfach, dass unsere Werte korrekt sind und zwingen sie in den entsprechenden Datensatz hinein. Diese *last-in-wins*-Methodik ist aber potenziell gefährlich. Auch Spalten, die wir gar nicht aktualisieren wollten, werden so mit dem aktuellen Datenbankwert aufgefrischt.

Falls es die Geschäftsanforderungen verlangen, könnten wir die Änderungen mit den neuen Werten aus der Datenbank kombinieren, indem wir einfach *RefreshMode* in *KeepCurrentValues* ändern. Auf diese Weise vereinigen wir die Änderungen von anderen Usern in unserem Datensatz und fügen unsere eigenen Änderungen hinzu. Wenn jedoch beide User denselben Wert geändert haben, wird der neue Wert den Wert überschreiben, den der erste User aktualisiert hat.

Um sicher zu gehen, können wir die Werte überschreiben, die der zweite User versucht hat zu ändern. In diesem Fall ist *RefreshMode.OverwriteCurrentValues* zu verwenden. An diesem Punkt wäre es wenig sinnvoll, die Änderungen zurück an die Datenbank zu schicken, weil kein Unterschied zwischen dem aktuellen Objekt und den Werten in der Datenbank besteht. Wir würden den aufgefrischten Datensatz dem User präsentieren, damit er die entsprechenden Änderungen erneut vornehmen kann.

Abhängig von der Anzahl der vom User vorgenommenen Änderungen mag es mehr oder weniger lästig sein, alle Daten nochmals einzugeben. Da *SubmitChanges* mehrere Datensätze im Stapel aktualisieren kann, könnte die Anzahl der Änderungen signifikant sein. Um hier hilfreich zu sein, nimmt die *SubmitChanges* Methode einen überladenen Wert entgegen der festlegt, wie wir nach einem Datensatzkonflikt weitermachen wollen. Wir können entweder die Auswertung weiterer Datensätze stoppen, oder aber eine Liste aller betroffenen Objekte zusammenstellen. Die *ConflictMode*-Enumeration spezifiziert zwei Optionen: *FailOnFirstConflict* und *ContinueOnConflict*.

Mit der *ContinueOnConflict*-Option müssen wir über die konfliktverursachenden Optionen iterieren und dafür den entsprechenden *RefreshMode* verwenden. Listing 8.5 illustriert, wie man alle konfliktfreien Datensätze an die Datenbank sendet und die konfliktverursachenden Datensätze mit den aktuellen Werten aus der Datenbank überschreibt.

Listing 8.5

Ersetzen der Werte des Users durch die Werte aus der Datenbank

C#

```
try
{
    context.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException)
{
    context.ChangeConflicts.ResolveAll(RefreshMode.OverwriteCurrentValues);
}
```

Mit dieser Methode können wir mindestens einige Werte an die Datenbank senden und dann den User auffordern, seine Informationen erneut in die konfliktverursachenden Elemente einzugeben. Dies kann allerdings ziemlich lästig sein, da der User sich erneut alle Änderungen anschauen müsste um festzustellen, welche Datensätze zu ändern sind.

Eine bessere Lösung wäre es, dem User alle geänderten Datensätze und Felder zu präsentieren. LINQ to SQL erlaubt nicht nur den Zugriff auf diese Informationen, sondern liefert für das konfliktverursachende Objekt auch den aktuellen Wert, den Originalwert und den Wert in der Datenbank. Listing 8.6 demonstriert die Verwendung der *ChangeConflicts*-Collection des *DataContext*, um die Details für jeden Konflikt zusammenzustellen.

Listing 8.6

Anzeige von Konfliktdetails

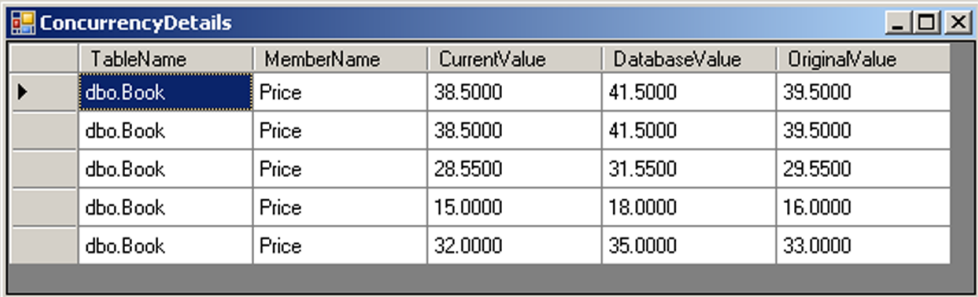
C#

```
try
{
    context.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException)
{
    var exceptionDetail =
        from conflict in context.ChangeConflicts
        from member in conflict.MemberConflicts
        select new
        {
            TableName = context.GetTableName(conflict.Object),
            MemberName = member.Member.Name,
            CurrentValue = member.CurrentValue.ToString(),
            DatabaseValue = member.DatabaseValue.ToString(),
            OriginalValue = member.OriginalValue.ToString()
        };
    ObjectDumper.Write(exceptionDetail);
}
```

Jedes Element in der *ChangeConflicts*-Collection enthält sowohl das konfliktverursachende Objekt als auch eine *MemberConflicts*-Collection. Letztere liefert Informationen über die Mitglieder *CurrentValue*, *DatabaseValue* und *OriginalValue*. Haben wir diese Informationen, so können wir sie für den User auf beliebige Weise zur Verfügung stellen.

Mit obigem Code können wir Einzelheiten zu den Konkurrenzverletzungen durch den User anzeigen (siehe Abbildung 8.1). Betrachten Sie die Möglichkeit, wenn zwei User den Preis eines

Buchs zur gleichen Zeit ändern wollen. Was passiert, wenn der eine den Preis um 2 Dollar erhöhen, der andere ihn aber um einen Dollar senken möchte? Der erste User hätte keine Probleme beim Abspeichern seiner Änderungen. Wenn aber der zweite User seine Änderungen abspeichern möchte, wird eine *ChangeConflictException* ausgelöst.



	TableName	MemberName	CurrentValue	DatabaseValue	OriginalValue
▶	dbo.Book	Price	38.5000	41.5000	39.5000
	dbo.Book	Price	38.5000	41.5000	39.5000
	dbo.Book	Price	28.5500	31.5500	29.5500
	dbo.Book	Price	15.0000	18.0000	16.0000
	dbo.Book	Price	32.0000	35.0000	33.0000

Abbildung 8.1 Anzeige der originalen, aktuellen und in der Datenbank gespeicherten Werte zwecks Auflösung von Konkurrenzverletzungen

Wird er mit den Konfliktdetails konfrontiert, so kann der zweite User für jeden Datensatz individuell entscheiden, wie er den jeweiligen Konflikt auflösen will. Den Schlüssel dazu liefert der *DataContext*, der weit mehr als nur ein *Connection*-Objekt bietet, sondern standardmäßig auch die Änderungen an Datensätzen verfolgt und das Konkurrenzmanagement übernimmt. Wir müssen selbst Hand anlegen, um die Optionen zur optimistischen Konkurrenz auszuschalten.

Beim Entwurf von Systemen mit Multiuserzugriff müssen wir uns tiefgründige Gedanken über Konkurrenzprobleme machen. In den meisten Fällen ist dies nicht allein mit dem Auslösen einer *ChangeConflictException* getan, sondern auch eine Frage des Zeitpunkts. Beim Behandeln der Ausnahme können wir diese entweder mit einem der Auflösungsmodi oder durch Zurückrollen der gesamten Transaktion behandeln. Im nächsten Abschnitt betrachten wir die Optionen für das Transaktionsmanagement innerhalb von LINQ to SQL.

8.1.4 Auflösen von Konflikten mittels Transaktionen

Als wir über die Konkurrenzoptionen sprachen, hatten wir bemerkt, dass die Anwendung von *SubmitChanges* einen einzelnen Datensatz oder auch mehrere Datensätze (sogar über mehrere Tabellen hinweg) aktualisieren kann. Wenn ein Konflikt auftritt, können wir entscheiden, wie wir ihn behandeln wollen. Allerdings haben wir noch nicht darauf hingewiesen, welche Anstrengungen erforderlich sind, um die Änderungen rückgängig zu machen. Wenn dadurch einige Datensätze gespeichert werden und andere nicht, könnte das die Datenbank in einen instabilen Zustand versetzen.

Warum ist es schlecht, einige Datensätze zu speichern und andere nicht? Stellen Sie sich vor, Sie gehen in einen Computerladen, um dort die Komponenten für einen neuen Computer zu kaufen. Sie nehmen ein Motherboard, ein Gehäuse, ein Netzteil, eine Festplatte, eine Grafikkarte und gehen damit zur Kasse. Der aufmerksame Verkäufer bemerkt aber, dass Arbeitsspeicher und Prozessor noch fehlen. Er sucht in den Regalen und im Lagerraum und stellt fest, dass kein kompati-

bler Prozessor vorrätig ist. An diesem Punkt stehen Sie vor der Entscheidung, entweder den Kauf trotzdem abzuschließen und zu hoffen, in einem anderen Geschäft einen passenden Prozessor zu finden oder das Motherboard auszutauschen oder aber den gesamten Kauf rückgängig zu machen.

Nehmen Sie nun an, dass der Computer Ihre Datenbank ist und dass die Komponenten Datensätze sind, die aktualisiert werden müssen und dass der Verkäufer die Rolle des *DataContext* spielt. Der Verkäufer, der das Problem bemerkt hat, ist vergleichbar mit einem *DataContext*, der eine *ChangeConflictException* auslöst. Wenn Sie die erste Option wählen (Sie kaufen das, was Sie bis jetzt haben), könnten Sie *ConflictMode.ContinueOnConflict* verwenden und die aufgetretenen Konflikte ignorieren. Natürlich muss der *DataContext* über die Art der Konfliktbehandlung bereits informiert werden bevor die Konflikte auftreten. Wählen Sie die dritte Option (Sie geben auf und gehen nach Hause), müssten alle Änderungen rückgängig gemacht werden (Sie erhalten Ihr Geld zurück). Wenn Sie die mittlere Option wählen, müssten Sie die Änderungen in der Datenbank rückgängig machen und dann entscheiden, welche Änderungen Sie durchführen wollen. Haben Sie diese Änderungen durchgeführt, können Sie erneut versuchen, diese in der Datenbank abzuspeichern.

LINQ to SQL offeriert drei Hauptmechanismen um Transaktionen zu verwalten. In der ersten Option, die standardmäßig eingestellt ist, erzeugt der *DataContext* beim Aufruf von *SubmitChanges* eine Transaktion. Diese wird, in Abhängigkeit von der gewählten *ConflictMode*-Option, automatisch zurückgerollt.

Wollen wir die Transaktion manuell begleiten, offeriert der *DataContext* die Fähigkeit, die bereits für die Datenbankverbindung vorhandene Transaktion zu verwenden. In diesem Fall rufen wir *BeginTransaction* auf dem *DataContext.Connection* auf, bevor wir versuchen, die Änderungen abzuspeichern. Danach können wir versuchen, entweder die Änderungen zu bestätigen oder sie zurückzurollen. Listing 8.7 demonstriert diese Alternative.

Listing 8.7

Transaktionsverwaltung durch den DataContext**C#**

```
try
{
    context.Connection.Open();
    context.Transaction = context.Connection.BeginTransaction();
    context.SubmitChanges(ConflictMode.ContinueOnConflict);
    context.Transaction.Commit();
}
catch (ChangeConflictException)
{
    context.Transaction.Rollback();
}
```

Die Kehrseite einer Transaktionsverwaltung direkt durch den *DataContext* ist die, dass damit keine Mehrfachverbindungen oder mehrfache *DataContext*-Objekte erfasst werden können. Als eine dritte Option kann das mit dem .NET 2.0 Framework eingeführte *System.Transactions.TransactionScope*-Objekt verwendet werden. Dazu ist ein Verweis auf die *System.Transactions*-Library einzurichten.

Dieses Objekt skaliert automatisch die Transaktionen der dazugehörigen Objekte. Falls es sich nur um einen einfachen Datenbankaufruf handelt, wird nur eine einfache Datenbanktransaktion ver-

wendet. Bei mehreren Klassen mit Mehrfachbindungen wird automatisch auf eine Enterprise Transaktion skaliert. Außerdem müssen wir nicht explizit die Transaktion beginnen oder sie zurückrollen. Nur um den Abschluss müssen wir uns kümmern. Listing 8.8 zeigt die Verwendung von *TransactionScope* unter LINQ to SQL.

Listing 8.8

Transaktionsverwaltung mit dem TransactionScope-Objekt**C#**

```
using (System.Transactions.TransactionScope scope =
    new System.Transactions.TransactionScope())
{
    context.SubmitChanges(ConflictMode.ContinueOnConflict);
    scope.Complete();
}
```

Im Unterschied zu den anderen Transaktionsmechanismen brauchen wir den Code nicht in einen *try-catch*-Block einzubetten um die Transaktion zurückzurollen. Mit dem *TransactionScope* wird die Transaktion automatisch zurückgerollt, es sei denn, wir rufen die *Complete*-Methode auf. Wird eine Ausnahme beim Aufruf von *SubmitChanges* geworfen, so wird *SubmitChanges* die *Complete*-Methode umgehen. Wir brauchen die Transaktion nicht explizit zurückzurollen. Zwar muss diese noch explizit in einen Fehlerbehandlungsblock eingebunden werden, aber die Behandlung kann näher am User Interface erfolgen.

Das wirkliche Highlight am *TransactionScope*-Objekt ist, dass es sich automatisch auf Basis des vorhandenen Kontexts skaliert. Es funktioniert gleichermaßen mit lokalen Transaktionen und mit heterogenen Quellen. Wegen seiner Flexibilität und Stabilität ist der Einsatz des *TransactionScope*-Objekts die bevorzugte Methode unter LINQ to SQL.

Das Verwalten von Transaktionen und die Behandlung von Konkurrenz gehören in den meisten Applikationen mit zu den wichtigsten Aufgaben. Auch wenn LINQ to SQL grundlegende Implementierungen dieser wichtigen Konzepte bereitstellt, hat der Programmierer dennoch die Möglichkeit, spezifische Anpassungen vorzunehmen. Diese sind allerdings nicht nur auf Transaktionen und Konkurrenz beschränkt, sondern sie beziehen sich auch auf einige datenbankspezifische Fähigkeiten, mit denen wir arbeiten können. Im folgenden Abschnitt wollen wir einen Blick darauf werfen.

8.2 Fortgeschrittene Datenbankfeatures

In vielen Fällen ist das standardmäßige Mapping zwischen Tabellen und Objekten für einfache CRUD¹ Operationen völlig ausreichend. Manchmal aber genügt eine solche direkte Beziehung nicht mehr. In diesem Abschnitt wollen wir einige der zusätzlichen Optionen diskutieren, die LINQ to SQL für die Anpassung des Datenzugriffs bereitstellt. In jedem Fall reduziert das Programmiermodell den Anteil handgeschriebenen Codes beträchtlich. Wir beginnen mit dem Untersuchen von Statements, die wir direkt gegen die Datenbank absetzen. Dann betrachten wir die Programmieroptionen für den SQL Server, wie gespeicherte Prozeduren und benutzerdefinierte Funktionen.

¹ Create, Read, Update, Delete