von ralf westphal

ABKEHR VOM ÜBLICHEN: DATENBANKPROGRAMMIERUNG MIT DEM „.NET FRAMEWORK" (TEIL 2)

Das .NET Framework als sprachübergreifende Standardbibliothek enthält auch eine komplette Programmierschnittstelle (API) für die Datenbankprogrammierung: „ADO.NET". Deren Klassen sind jedoch nicht einfach nur ein neuer Aufguss bisheriger Datenbank-APIs. Vielmehr verkörpern sie die Abkehr vom Üblichen des cursor-basierten Zugriffs auf Datenbanken. ADO.NET setzt auf eine strikte Trennung von Datencontainer und Datenquelle.

der autor

Ralf Westphal (E-Mail: ralfw@ralfw.de) ist freier Softwaretechnologievermittler und arbeitet als Fachautor, Coach, Softwareentwickler und Sprecher auf Konferenzen im In- und Ausland. Darüber hinaus ist er einer der deutschen Microsoft MSDN Regional Directors.

Im ersten Teil dieser Artikelserie ([Wes02-a]) wurde der Datencontainer der ADO.NET-Klassenbibliothek vorgestellt. Die Klasse DataSet (zusammen mit weiteren Klassen, wie DataTable, DataColumn, DataRow) stellt eine leistungsfähige *in-memory* Datenbank dar und dient als Cache für Daten aus unterschiedlichen Quellen.

Mit dem DataSet bietet Microsoft ein generisches Objektmodell insbesondere für tabellarisch strukturierte Informationen. Abgesehen von seiner Allgemeinheit und Unabhängigkeit von der Datenherkunft bietet es komfortable Dienste, die über „selbst gebaute" Objektmodelle hinausgehen:

- Datenversionierung,
- Sichten (DataView, XmlDocument),
- Such- und Sortierfunktionen sowie
- Serialisierung.

Schon für sich allein genommen ist also ein DataSet eine sehr wertvolle Struktur für die Datenhaltung und den Transport (z.B. in verteilten Anwendungen). Darüber hinaus bietet es sich an, wo immer Sie persistente Informationen in Objektmodellen halten, darüber nachzudenken, ob solche Objektmodelle nicht davon profitieren, wenn sie auf einem DataSet basieren (durch Aggregation oder Vererbung, vgl. typisierte DataSet-Objekte in [Wes02-a]).

Im Rahmen der Datenbankprogrammierung löst das DataSet mehrere Probleme: In ihm können Sie Daten sammeln, sie verändern und bidirektional traversieren. Traditionelle Datenbank-APIs bieten dafür Cursor. Cursor sind jedoch insofern hybride Konzepte, als dass sie zum einen die Verwaltung von Daten in dieser Weise ermöglichen, zum anderen aber auch für die Datenbeschaffung zuständig sind. Cursor sind im doppelten Sinn immer eng mit einer Datenquelle verbunden: Sie sind für bestimmte Arten von Datenquellen ausgelegt (z.B. ODBC- oder OLE-DB-Datenquellen) und mit einer konkreten Datenquelle während der Arbeit an den ausgewählten Daten ständig in Kontakt.

Dem DataSet fehlt diese Verbundenheit mit Datenquellen komplett. Es ist auf Datenverwaltungsaufgaben hin optimiert und kennt weder die Herkunft seiner Daten, noch ist es auf bestimmte Arten von Datenquellen zugeschnitten. Für die Datenbankprogrammierung mit ADO.NET steht also die Frage im Raum, wie Sie auf konkrete Datenquellen zugreifen, um deren Inhalte dann unter anderem in einem DataSet zu verwalten.

ADO.NET-Architektur: Anbindung an Datenquellen

Managed Provider

Ein wichtiger Aspekt der Abkehr vom Üblichen bisheriger Datenbank-APIs ist die Kapselung der reinen Kommunikation mit konkreten Datenquellen in einer begrenzten Zahl individueller ADO.NET-Klassen. Nur was beim Umgang mit Datenquellen verschiedenster Art immer wieder anders sein kann, ist in den so genannten *Managed Providern* zusammengefasst (alle anderen Aufgaben erledigt das DataSet):

- Verbindungsaufbau, z.B. via „OLE DB Provider" oder ODBC-Treiber oder eine datenquellen-spezifische Programmierschnittstelle,
- Kommandoausführung, z. B. in Form von SQL-Anweisungen oder einer anderen Kommandosprache,
- Transaktionen,
- cursor-basierter Zugriff,
- Transfer von Daten zwischen DataSet und Datenquelle.

Das .NET Framework wird bisher standardmäßig mit zwei *Managed Providern* ausgeliefert:

- *OLE DB Managed Provider* (Namespace System.Data.OleDb) für den Zugriff auf alle Datenquellen, für die ein OLE-DB-Provider existiert.
- *SQL Managed Provider* (Namespace System.Data.SqlClient) für den Zugriff auf SQL-Server-Datenbanken; der *Managed Provider* kommuniziert direkt mit der nativen SQL-Server-Programmierschnittstelle via TCP/IP-Sockets bzw. *Named Pipes*.

Darüber hinaus hat Microsoft inzwischen zwei weitere *Managed Provider* implementiert, die Teil des nächsten Release des .NET Framework sein werden:

- *ODBC Managed Provider* (Namespace System.Data.Odbc, vgl. [ODBC]) für den Zugriff auf alle Datenquellen, für die ein ODBC-Treiber existiert.
- *Oracle Managed Provider* (Namespace System.Data.OracleClient, vgl. ▸

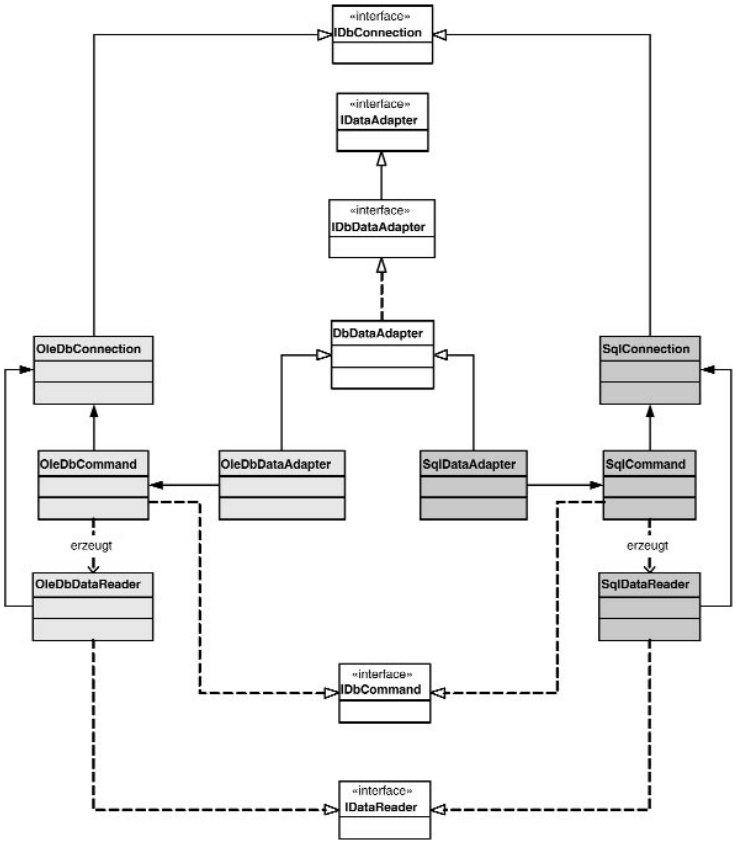

Abb. 1: Auszugsweises Klassendiagramm der Managed Provider für OLE DB und SQL Server, die im Standardumfang des .NET-Frameworks enthalten sind

[Ora]) für den Zugriff auf Oracle-Datenbanken mittels des nativen Oracle-Programmierschnittstelle.

So unterschiedlich und datenquellen-spezifisch die Implementierungen der *Managed Provider* intern aber auch sein mögen, sie bestehen doch alle aus Klassen gleicher Art. Alle enthalten beispielsweise eine konkrete, durch einen providerabhängigen Präfix gekennzeichnete Connection, Command-, DataReader- und DataAdapter-Klasse. Allerdings sind diese nicht von allgemeineren Basisklassen abgeleitet, sondern implementieren lediglich dieselben aufgabenspezifischen Interfaces (alle Connection-Klassen z. B. IDbConnection, **siehe Abb. 1**).

Trotz potenziell großer Unterschiedlichkeit von Datenquellen lässt sich somit prinzipiell Code datenquellen-unabhängig formulieren, solange er nur mit den allgemeinen Interfaces arbeitet. Für viele Bereiche der Datenbankprogrammierung funktioniert das auch gut – 100% generischer Code lässt sich jedoch nicht erreichen, weil es keine *Factory*-Methoden für Connection- und DataAdapter-Objekte gibt und die *Managed Provider* im Fehlerfall spezifische Ausnahmen werfen (z.B. SqlException beim *SQL Managed Provider*).

Die Bezeichnung der Datenquellen-Provider als *managed* ist dem Umstand geschuldet, dass sie komplett in *Managed Code*, d. h. in der *Intermediate Language (IL)*, vorliegen. Sie fügen sich damit nahtlos in das Klassengefüge der .NET-Framework-Standardbibliothek ein und unterliegen der Aufsicht der *Common Language Runtime (CLR)*. Auch für den Datenbankzugriff gilt somit z. B. das Sicherheitsrahmenwerk *Code Access Security*, d.h. Code kann aufgrund seiner Herkunft (beispielsweise von einer bestimmten Netzwerk-Share, einer Domain im Internet oder einem bestimmten Hersteller) der Umgang mit Datenquellen zugestanden oder verwehrt werden.

Verbindungen

Die größte Ähnlichkeit zwischen üblichen Datenbank-APIs und ADO.NET besteht sicherlich in den Klassen der *Managed Provider*. Grundlage für die Kommunikation mit einer Datenquelle ist immer noch eine Verbindung. Alle *Managed Provider* besitzen daher ein Connection-Objekt:

```
using System.Data.SqlClient;
...
SqlConnection conn =
new SqlConnection("server=(local);...");
conn.Open();
...
conn.Close();
```

Jede Interaktion mit einer Datenquelle erfordert eine offene Verbindung. Der *Connection-String* (hier der Konstruktor-Parameter) ist dabei natürlich abhängig von *Managed Provider* (hier: *SQL Server Managed Provider*) und Datenquelle.

Um Verbindungen (und andere Ressourcen) nach Gebrauch garantiert wieder freizugeben, empfiehlt es sich, sie in einem Try-Catch-Block zu öffnen und zu schließen oder in C# eine using-Anweisung zu benutzen:

VB.NET:
```
Dim conn as New SqlConnection("...")
Try
  conn.Open()
  ...
Finally
  conn.Close()
End Try
```
C#:
```
using(SqlConnection conn =
new SqlConnection("..."))
{
  conn.Open();
  ...
}
```

Für ein *Connection Pooling* sorgt gegebenenfalls der *Managed Provider* selbst oder eine darunter liegende Treiberschicht (z.B. *OLE DB Provider*). Im Sinne eines skalierbaren Datenzugriffs gilt also und gerade auch bei ADO.NET, dass Verbindungen möglichst spät geöffnet und möglichst schnell wieder geschlossen werden sollten.

Kommandos

Sobald eine Verbindung aufgebaut ist, können Sie auf dieser in üblicher Weise Kommandos an die Datenquelle absetzen.

```
SqlCommand cmd = new SqlCommand(_
  "select count(*) from customers", conn);
int n = (int)cmd.ExecuteScalar();
Console.WriteLine(n);
```

Die Syntax der Kommandos ist abhängig vom *Managed Provider* bzw. der darunter liegenden Treiberschicht. In den meisten Fällen geht es aber natürlich um einen SQL-Dialekt.

Kommandos bieten mehrere Ausführungsmodi:

- ExecuteScalar(): führt ein Kommando aus, von dem erwartet wird, dass es genau einen Wert zurückliefert (s.o.);
- ExecuteNonQuery(): führt ein Kommando aus, das einen Effekt auf der Datenquelle hat, aber selbst kein Ergebnis zurückliefert, z. B. ein SQL-DML-Kommando (wie insert) oder ein DDL-Kommando (wie create table);

- ExecuteReader(): führt ein Kommando aus, das einen DataReader, also einen Cursor zurückliefert (siehe unten), z.B. ein SQL select.

Darüber hinaus können einzelne *Managed Provider* weitere Funktionalität implementieren. So kann das Ergebnis einer select-Abfrage via SqlCommand an eine SQL-Server-Datenbank auch als XmlReader zurückgeliefert werden (Methode ExecuteXmlReader()).

Welche Arten von Kommandos ein Command-Objekt akzeptiert, hängt vom *Managed Provider* ab. Gewöhnlich sind es SQL-Anweisungen, ein *Stored Procedure*-Name oder ein Tabellenname, wie zum Beispiel:

```
SqlCommand cmd = new SqlCommand(
  "CustOrderHist", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add("@CustomerID",
  SqlDbType.VarChar).Value = "ALFKI";
```

Außerdem können Sie selbstverständlich, Parameter an Kommandos übergeben und auch Werte zurückerhalten. Jeder Parameter ist durch einen Namen identifiziert, hat einen vom *Managed Provider* abhängigen Typ (z.B. aus der Menge SqlDbType) und eine Richtung (z. B. Input, ReturnValue).

Inwiefern ein Parameter jedoch innerhalb eines Kommandos mit seinem Namen identifizierbar ist, hängt vom *Managed Provider* bzw. einem darunter liegenden Treiber ab. Der *SQL Managed Provider* erlaubt die Referenzierung mittels Name:

```
SqlCommand cmd = new SqlCommand(
  "select customerid from customers " &
  where customerid=@CustomerID",
  conn);
cmd.Parameters.Add("@CustomerID",
  SqlDbType.VarChar).Value = "ALFKI";
```

```
conn.Open();
using(SqlTransaction tx = conn.BeginTransaction())
{
  SqlCommand cmd = new SqlCommand("insert
    into suppliers (supplierid, companyname)
    values (@id, @name)", conn);
  cmd.Transaction = tx;
  cmd.Parameters.Add("@id",
SqlDbType.VarChar).Value = ...;
  cmd.Parameters.Add("@name",
SqlDbType.VarChar).Value = ...;
  cmd.ExecuteNonQuery();
  tx.Commit();
}
```

Listing 1: Transaktionen müssen Kommandos explizit zugeordnet werden

Der *OLE DB Managed Provider* versteht beim Zugriff auf eine SQL-Server-Datenbank hingegen nur „?" als Platzhalter:

```
OleDbCommand cmd = new OleDbCommand(
  "select customerid from customers "
  where customerid=?",
   conn);
cmd.Parameters.Add("@CustomerID",
OleDbType.VarChar).Value = "ALFKI";
```

Transaktionen

Um mehrere Datenbankkommandos als Einheit zusammenzufassen, bietet ADO.NET natürlich Transaktionen. Anders als bei bisherigen Datenbank-APIs laufen Kommandos jedoch nicht implizit in Transaktionen ab, sondern müssen ihnen immer ausdrücklich zugeordnet werden (vgl. Listing 1). Falls umgekehrt zu einem Zeitpunkt, an dem eine Transaktion offen ist, Kommandos ohne zugewiesene Transaktion ausgeführt werden sollen, wird ein Fehler geworfen.

```
using(SqlTransaction tx = conn.BeginTransaction())
{
  SqlCommand cmd = new SqlCommand("insert ... ", conn);
  cmd.Transaction = tx;
  ...
  cmd.ExecuteNonQuery();
  tx.Save("A");
  try
  {
    cmd.Parameters["@id"].Value = ...;
    cmd.Parameters["@name"].Value = ...;
    cmd.ExecuteNonQuery();
  }
  catch(Exception e)
  {
    tx.Rollback("A");
  }
  tx.Commit();
}
```

Listing 2: Zurückrollen einer Transaktion bis zu einem Sicherungspunkt

Transaktionen werden durch die *Factory*-Methode BeginTransaction() der Connection-Objekte erzeugt und geöffnet. Für eine positive Beendigung oder einen Abbruch rufen Sie allerdings Commit() bzw. Rollback() auf dem Transaktionsobjekt aus. Das sprachübergreifende Muster dafür ist:

```
tx = conn.BeginTransaction()
Try
  ...
  tx.Commit()
Catch
  tx.Rollback()
  ...
End Try
```

In Listing 1 kann das Rollback() jedoch fehlen, weil es implizit am Ende der using-Anweisung ausgeführt würde, falls vor oder während Commit() ein Fehler aufträte.

Funktionalität von Transaktionen, die darüber hinausgeht, ist von den einzelnen *Managed Providern* bzw. den darunter liegenden Treibern abhängig. So unterstützt der *SQL Managed Provider* beispielsweise Sicherungspunkte (vgl. Listing 2) und der *OLE DB Managed Provider* zusammen mit dem OLE-DB-Treiber für MS-Access-Datenbanken sogar geschachtelte Transaktionen.

Cursor

Verbindungen, Kommandos und auch Transaktionen entsprechen bei ADO.NET noch dem üblichen Programmiermodell für Datenbanken. Bei Cursorn weicht ADO.NET jedoch vom Üblichen ab. *ADO.NET Managed Provider* bieten keine Cursor-Vielfalt mit der Entscheidungsnot, wie ein Cursor Veränderungen an den unterliegenden Daten behandeln oder wie er Daten sperren soll usw.

ADO.NET bietet lediglich einen einzigen Cursor: einen *read-only, forward-only* Cursor in Form eines DataReader-Objekts. Einen DataReader können Sie nur über die *Factory*-Methode ExecuteReader() der Command-Objekte erzeugen und öffnen.

```
SqlCommand cmd = new SqlCommand(
  "select customerid from customers ...",
  conn);
using(SqlDataReader r = cmd.ExecuteReader())
{
  while(r.Read())
  ...
}
```

Ein DataReader hält Ressourcen und muss nach Gebrauch explizit mit Close() geschlossen werden. Alternativ können Sie das in C# aber wieder einer using-Anweisung überlassen, die am Ende ein Dispose() aufruft.

Ein wesentlicher Unterschied zu bisherigen Cursorn besteht darin, dass Sie – während ein DataReader auf einer Verbindung offen ist – keine (!) anderen Kommandos über diese Verbindung ausführen können. Das unterstreicht die Rolle, die Cursor für Microsoft in ADO.NET spielen: Sie dienen lediglich der Sichtung von Daten und vor allem der performanten Übertragung von Abfrageresultaten aus einer Datenquelle in eine *in-memory* Datenstruktur, z.B. ein DataSet (siehe unten) oder ein anderes Objektmodell (vgl. Listing 3).

Datenmanipulationen via Cursor mit pessimistischer Sperrung sind in diesem ▶

```
class Supplier
{
  public string id, name;
  public Supplier(string id, string name)
  {
    this.id = id;
    this.name = name;
  }
}
...
ArrayList alSuppliers;
using(SqlConnection conn = new SqlConnection("..."))
{
  conn.Open();
  SqlCommand cmd = new SqlCommand("select supplierid,
    companyname from suppliers", conn);
  using(SqlDataReader dr = cmd.ExecuteReader())
  {
    alSuppliers = new ArrayList();
    while(dr.Read())
      alSuppliers.Add(new Supplier
      (dr["supplierid"].ToString(), dr["companyname"].ToString()));
  }
}
```

Listing 3: Füllen eines eigenen Objektmodells mit einem DataReader

Bild nicht enthalten. Microsoft spricht sich eindeutig dagegen aus und setzt auf das besser skalierbare Programmiermodell, in dessen Mittelpunkt ein verbindungsloser Daten-Cache steht. Damit zielt ADO.NET vor allem auf die Erfordernisse in serverseitigem Code von Internet-Anwendungen.

Wenn Sie für Anwendungen in einem lokalen Netzwerk die Datenmanipulation via verbindungsbehaftetem Cursor auch in .NET-Framework-Programmen benötigen, müssen Sie also weiterhin ADO benutzen – und sind damit auch an die Nachteile von COM gebunden.

DataAdapter 1: DataSet füllen

Da eine satzorientierte Manipulation und bidirektionale Traversierung von Daten nicht mit den Klassen der *Managed Provider* möglich ist, sondern nur in einem DataSet, stellt sich die Frage, wie Daten aus einer Datenquelle möglichst einfach in ein DataSet geladen und später wieder gespeichert werden?

Da dies eine wiederkehrende, zentrale Anforderung der meisten datenbankorientierten Anwendungen ist, enthalten *Managed Provider* eine DataAdapter-Klasse. Sie ist dafür verantwortlich, Abfrageresultate in eine DataSet-Tabelle zu übertragen und umgekehrt, veränderte Sätze aus DataTable-Objekten zurück in eine Datenquelle zu schreiben.

```
SqlConnection conn = new SqlConnection("...");
DataSet ds = new DataSet();
SqlDataAdapter adap = new SqlDataAdapter(
  "select customerid from customers", conn);
adap.Fill(ds, "kunden");
```

Jeder DataAdapter repräsentiert ein Abfragekommando (Eigenschaft SelectCommand), das beim Aufruf von Fill() gestartet und dessen Ergebnis in eine Tabelle mit dem angegebenen Namen (hier „kunden") übertragen wird. Existiert die Tabelle im DataSet (hier ds) noch nicht, erzeugt sie der DataAdapter.

Die Unabhängigkeit des DataSet von der Datenquelle unterstreicht, dass der *in-memory* Tabellenname („kunden") nicht dem in der Datenquelle („customers") entsprechen muss. Im Falle eines SQL-*Join* gäbe es ja ohnehin keinen eindeutigen Quell-Tabellennamen.

Intern verwendet ein DataAdapter ein Command-Objekt für das Abfragekommando und erzeugt damit einen DataReader. Über ihn werden gegebenenfalls Metainformationen zum Aufbau der Zieltabellenstruktur ermittelt und schließlich die Resultatsmenge traversiert, um die Datensätze einzeln in DataRow-Objekte zu übertragen und an die Tabelle anzuhängen. Dieses Vorgehen entspricht dem in [Wes02-a] gezeigten für das Füllen von Tabellen. Sobald die Daten im DataSet angekommen sind, sind die *Managed-Provider*-Klassen also aus dem Spiel, was die Datenverwaltung angeht.

DataAdapter 2: DataSet speichern

DataAdapter kommen jedoch wieder ins Spiel, wenn es darum geht, veränderte Datensätze aus dem DataSet in eine Datenquelle zu übertragen. Dabei ist es unerheblich, ob es dieselbe ist wie die, aus der die Daten geladen wurden. Die Entkopplung von Datencontainer und Datenquelle macht es also auch möglich, im selben DataSet Daten verschiedener Datenquellen zusammenzuführen und mit Relationen zu verbinden. Diese Möglichkeit gibt es mit Cursorn nicht.

Änderungen an DataSet-Inhalten übertragen DataAdapter Tabellenzeile für Tabellenzeile durch Ausführung von Kommandos (**siehe Listing 4**). Für jede mögliche Veränderung (Neuanlage, Manipulation, Löschung) enthält ein DataAdapter ein Kommando (Eigenschaften InsertCommand, UpdateCommand, DeleteCommand). Welche Kommandos dann genau bei welcher Veränderung ausgeführt werden sollen, können Sie durch Zuweisung vor Aufruf der Methode Update() festlegen. In **Listing 4**, das der Einfachheit halber nur den Fall eines neuen Datensatzes behandelt, ist dies das SQL-Kommando insert.

Die Methode Update() sorgt für die Speicherung der Veränderungen an einer wie bei Fill() spezifizierten DataSet-Tabelle.

```
DataTable tb = ds.Tables["kunden"];
DataRow r = tb.NewRow();
r["supplierid"] = ...;
r["companyname"] = ...;
tb.Rows.Add(r);
SqlCommand cmdIns = new SqlCommand("insert into suppliers (supplierid,
  companyname) values (@id, @name)", conn);
cmdIns.Parameters.Add("@id", SqlDbType.VarChar,0, "supplierid");
cmdIns.Parameters.Add("@name", SqlDbType.VarChar, 0, "companyname");
adap.InsertCommand = cmdIns;
adap.Update(ds, "kunden");
```

Listing 4: Speicherung von Änderungen an einem DataSet mit einem DataAdapter

Dafür durchläuft sie alle Zeilen der Tabelle, prüft, ob und wie sie verändert wurden (**Listing 5**, Eigenschaft RowState) und ruft das passende Kommando auf.

Die Übergabe der Tabellenspaltenwerte an das Kommando erfolgt über Parameter, die Sie festlegen. Die Zuordnung eines Parameternamens zu einer Spalte in der Datenquelle ist dann implizit durch dessen Position in der SQL-Anweisung festgelegt.

```
foreach(DataRow r in ds.Tables["kunden"].Rows)
{
  switch(r.RowState)
  {
    case DataRowState.Added:
      this.InsertCommand.Parameters["@id"].Value = r["supplierid"];
      this.InsertCommand.Parameters["@name"].Value = r["companyname"];
      this.InsertCommand.ExecuteNonQuery();
    case DataRowState.Deleted: ...;
    case DataRowState.Modified: ...;
  }
  r.AcceptChanges();
}
```

Listing 5: Blick in die Update()-Methode eines DataAdapter anhand eines fest verdrahteten Beispiels (vgl. Listing 4)

Aber die Zuordnung einer Tabellenspalte zu einem Parameter müssen Sie explizit bei dessen Definition vornehmen:

```
cmdIns.Parameters.Add("@id", SqlDbType.VarChar, 0,
"supplierid");
```

Änderungen, die erfolgreich persistiert wurden, werden anschließend im DataSet akzeptiert (Methode AcceptChanges()). Das heißt, veränderte Zeilen sind nicht mehr als verändert markiert und Spaltenversionsinformationen werden verworfen.

Auch wenn Update() alle Änderungen an einer Tabelle speichert, läuft die Methode doch nicht transaktional! Wenn Sie sicherstellen wollen, dass entweder alle oder keine Änderungen – falls beispielsweise auch nur ein Kommando fehlschlägt – in der Datenquelle vorgenommen werden, müssen Sie selbst um Update() eine Transaktion legen und sie dem SelectCommand des DataAdapter zuweisen.

Die Funktionsweise der DataAdapter ist im Grunde trivial. Sie dienen vor allem Ihrer Bequemlichkeit, indem Sie Schleifen für das Lesen bzw. Schreiben implementieren. Die entscheidendende Kommunikation mit der Datenquelle geschieht durch DataReader bzw. Command-Objekte.

Bequem ist es allerdings nicht, wenn Sie immer alle Speicherkommandos selbst formulieren und zuweisen müssen, bevor Sie Update() aufrufen können. Es mag der Performance dienen, weil Sie frei in ihrer Definition sind und also auch *Stored Procedures* einsetzen können. Aber der Aufwand im Code ist hoch.

Und noch einen zweiten Nachteil hat die ausdrückliche und einmalige Zuweisung von Kommandos: Für jeden veränderten Datensatz wird – abhängig von der Veränderungsart - dasselbe Kommando aufgerufen. Das kann vor allem für veränderte Datensätze nachteilig sein, denn UpdateCommand überträgt immer alle durch Parameter spezifizierten Spalten – unabhängig davon, ob auch an allen Veränderungen vorgenommen wurden. Es kann also zu einem Datentransport-Overhead kommen.

Abhilfe schafft hier die Klasse CommandBuilder. Sie erzeugt selbstständig alle nötigen Kommandos für einen DataAdapter und tut dies auch noch dynamisch:

```
SqlCommandBuilder cb = new SqlCommandBuilder(adap);
adap.Update(ds, "kunden");
```

Sobald ein CommandBuilder an einen DataAdapter gebunden ist, lauscht er auf das Event RowUpdating des DataAdapter. Dieses wird für jede veränderte Zeile vor (!) der Speicherung gefeuert und gibt dem CommandBuilder Gelegenheit, sie zu inspizieren. Die existierenden CommandBuilder erzeugen dann aufgrund des Änderungsstatus sowie der tatsächlich veränderten Spalten für jede Zeile ein spezielles Speicherkommando (soweit keines durch Sie vorgegeben wurde) und geben dieses zurück an den DataAdapter, der es ausführt.

Diese Kommandos sind aber nicht nur im Hinblick auf die konkret vorliegenden Veränderungen optimiert, sondern implementieren auch optimistisches Sperren. D.h. sie schreiben Änderungen in eine Datenquelle nur dann zurück, wenn der Quelldatensatz unverändert ist, seitdem er ins DataSet geladen wurde. Die Prüfung auf Unverändertheit findet immer und durch Vergleich von Spaltenwerten statt, nicht über einen *Timestamp*. Andere Strategien müssen Sie selbst realisieren.

Auch die Behandlung von DataSet-Tabellen, die das Ergebnis eines SQL-*Join* sind, müssen Sie selber realisieren. CommandBuilder können nicht mit Tabellen umgehen, deren Spalten aus verschiedenen Basistabellen stammen.

Ein neues Programmiermodell für Datenbanken

Bis hierher haben Sie einen Überblick über die beiden Seiten von ADO.NET bekommen: den datenquellen-unabhängigen, für sich allein stehenden Datencontainer DataSet und die datenquellen-spezifischen *Managed Provider*. Auch wenn auf viele Details nicht eingegangen werden konnte, so hat die Darstellung sicherlich schon jetzt deutlich gemacht, inwieweit ADO.NET eine Abkehr vom Üblichen der bisherigen Datenbank-APIs ist.

Eine Einführung in ADO.NET ist aber nicht vollständig, wenn sie nicht über die Beschreibung der Funktionsweise der Klassen hinausweist und darauf eingeht, was das alles für Sie *bedeutet*. ADO.NET steht für nichts weniger als für ein neues Programmiermodell für den Umgang mit Datenbanken.

Das ist unerheblich, solange Sie über *Managed Provider* nur direkt Kommandos an Datenquellen absetzen wollen. Verbindungen, Kommandos und Nur-Lese-Cursor funktionieren so, wie Sie es von anderen Datenbank-APIs her gewohnt sind. Der Anteil einer nicht-trivialen Anwendung, der sich auf diese Funktionalität beschränkt, ist jedoch gewöhnlich klein.

Sobald Sie aber ein DataSet ins Spiel bringen, verändert sich das Programmiermodell so drastisch, dass es für die meisten von Ihnen einer Revolution gleich kommt. Der Verlust veränderbarer, ständig verbundener Cursor und die Organisation des DataSet haben massiven Einfluss darauf, wie Sie Ihre Daten laden, speichern und ihre Strukturen entwerfen.

In Bezug auf die im Folgenden dargestellten fünf wichtige Punkte sollten bzw. müssen Sie daher Ihr bisheriges Vorgehen bei der Datenbankprogrammierung neu überdenken.

Datensatzidentifikation

Bei veränderbaren Cursorn konnten Sie einen Datensatz, sobald er im Cursor anlag, einfach ändern und mussten sich nicht darum kümmern, wie die Änderungen in die darunter liegende Tabelle gelangten. Sie mussten dafür keinen Primärschlüssel besitzen. Bei ADO.NET ist das komplett anders. Da Datensätze in einem DataSet nicht mehr mit ihrer ▶

Quelle verbunden sind, müssen Sie immer (!) durch einen Primärschlüssel eindeutig identifizierbar sein, sonst können die DataAdapter-Kommandos Veränderungen nicht in die Datenquelle zurückübertragen.

Primärschlüsselvergabe

Wenn Sie neue Datensätze in einem DataSet anlegen, sollten Primärschlüsselwerte nicht durch die Datenquelle vergeben werden. Verzichten Sie auf Auto-Inkrement-Felder oder *Trigger* zur Primärschlüsselerzeugung. Diese Verfahren erzwingen eine performance-raubende Synchronisation des DataSet nach der Satzspeicherung, um die endgültigen Primärschlüsselwerte dorthin zurückzuübertragen. Primärschlüssel sollten dort vergeben werden, wo der neue Datensatz in das DataSet gehängt wird (vgl. hierzu auch [Wes02-b]).

SQL-Joins

SQL-*Joins* können Sie weiter wie gewohnt erzeugen und auch in ein DataSet laden. Falls Sie an den Daten Veränderungen vornehmen, müssen Sie deren Speicherung jedoch komplett selbst implementieren; die Bequemlichkeit der CommandBuilder ist dann keine Option mehr.

Unabhängig davon sollten Sie überdenken, wo SQL-*Joins* in Zukunft Sinn machen, wenn Sie Daten in ein DataSet laden. Die üblichen unhandlichen Datendopplungen, die darin durch die Verbindung von Tabellen in *1:n*-Beziehungen entstehen, weil die natürliche Datenhierarchie nicht erhalten bleiben kann, lassen sich jetzt vermeiden. Wenn Sie bisherige SQL-*Join*-Abfrage aufspalten und daraus mehrere, über Relationen verbundene DataSet-Tabellen erzeugen, können Sie vorhandene Baumstrukturen bewahren und leicht traversieren sowie Änderungen wesentlich einfacher speichern.

Datensatzsperrung

Eine pessimistische Datensatzsperrung mittels eines Cursor und einer Lock()-Funktion für den anliegenden Datensatz oder ähnliches ist mit ADO.NET nicht möglich. Unabhängig von der Diskussion über den prinzipiellen Nutzen und die Nachteile eines solchen Vorgehens, müssen Sie also überlegen, ob und wie Sie konkurrierende Datenveränderungen verhindern. ADO.NET setzt von sich aus auf optimistisches Sperren, d. h. Veränderungen werden nur an unveränderten Datensätzen ausgeführt. Wer also „zu spät kommt", den „bestraft" ADO.NET. Pessimistische Sperren, die womöglich schon beim Laden eines Datensatzes geprüft werden könnten, müssen Sie selbst implementieren. Sie sollten also in jedem Fall überdenken, inwiefern Ihre bisherige Praktik der Satzsperrung wirklich notwendig war, um zukünftig Aufwand zu vermeiden.

Datenmengen

Das vielfach übliche Vorgehen, auf Benutzeranfragen an eine Datenquelle (z.B. „Suche alle Kunden, die ein Kriterium erfüllen.") eine beliebig große Zahl von Ergebnissen zu akzeptieren und in einer Liste anzuzeigen, die dann am Bildschirm durchblättert werden kann, ist mit ADO.NET in den allermeisten Fällen inakzeptabel. Unabhängig von der immer schon fragwürdigen Sinnhaftigkeit der Anzeige mehrerer Tausend Datensätze steht ihr entgegen, dass diese Datensätze jetzt zur Anzeige erst komplett in einen Cache (DataSet oder eine andere bindbare Datenstruktur) geladen werden müssen. Cursor-gebundene Listensteuerelemente, die während des Scrollens bei Bedarf Daten nachgeladen haben, gibt es nicht mehr. Sie sollten für ADO.NET also Ihre Benutzerschnittstellen auf solche möglichen Performance- bzw. Speicher-Fresser überprüfen und eine andere Selektions- bzw. Ladestrategie wählen.

Zusammenfassung

Der Umstieg auf ADO.NET von ADO, DAO oder auch JDBC ist nur oberflächlich leicht. Die „Verhältnisse" haben sich geändert und es gibt auch Lücken (z.B. fehlt eine direkte Unterstützung für die Metadatenverwaltung, d. h. den Zugriff auf Datenquellenschemata, wie sie ADOX bietet). Um sie zu schließen, können Sie aber natürlich jederzeit auf vorhandene Programmierschnittstellen (ADO, ADOX, auch DAO) zurückgreifen, die per „COM-Interop" aus jeder .NET-Framework-Anwendung immer noch nutzbar sind – allerdings unter Rückgriff auf COM als Komponententechnologie mit all ihren Nachteilen.

Gerade die verbindungslose Datenhaltung von ADO.NET hat unter der Oberfläche entscheidende Konsequenzen für Sie. Microsoft hat sich dafür entschieden, um die Skalierbarkeit von Datenbankanwendungen zu verbessern und das Programmiermodell objektorientierter zu machen. Diese Vorteile können Sie aber nicht ohne Aufwand einstreichen, wie die im vorhergehenden Abschnitt angerissenen Punkte zeigen (vgl. dazu auch [Wes02-b]).

Ihnen diesen Aufwand aufzubürden, hat Microsoft sich allerdings nicht leicht gemacht. Vielmehr hat man die Alternativen gegeneinander abgewogen – und sich zugunsten der Anforderungen des Web-Programmierparadigmas entschieden, das zukünftig durch Web-Services die Entwicklung von Geschäftslogik und damit auch die Datenbankprogrammierung noch mehr bestimmen wird.

In den beiden Teilen dieser Artikelserie konnten natürlich nur die Grundlagen von ADO.NET angerissen werden. Es ging darum, Sie für die Chancen und Anforderungen sensibel zu machen, die die Abkehr des ADO.NET-Programmiermodells vom Üblichen mit sich bringt.

Ob das gelungen ist und inwiefern Ihre Anwendungen von ADO.NET profitieren können, müssen Sie nun selbst anhand prototypischer Umsetzungen prüfen. Ohne solche Experimente und ohne, dass Sie sich auf das neue Programmiermodell einlassen, geht es nicht. Aber der Aufwand lohnt sich – allemal im Hinblick eines Reviews „alt hergebrachter" Datenbankschemata und eingefahrener Programmierwege. ■

Literatur & Links

[ODBC] ODBC Managed Provider Download, siehe: http://msdn.microsoft.com/downloads/sample.asp?url=/MSDN-FILES/027/001/668/msdncompositedoc.xml

[Ora] Oracle Managed Provider Download, siehe: http://msdn.microsoft.com/downloads/sample.asp?url=/MSDN-FILES/027/001/940/msdncompositedoc.xml

[Wes02-a] R. Westphal, Abkehr vom Üblichen: Datenbankprogrammierung mit dem „.NET Framework" (Teil 1), in: OBJEKTspektrum 6/02

[Wes02-b] R. Westphal, ADO.NET Datenbankprogrammierung, Addison-Wesley 2002