9 Datenzugriff mit ADO.NET 2.0

157 Einführung
158 Neuerungen im Überblick
159 Die ADO.NET-Architektur
162 Der Weg der Daten von der Datenquelle bis zum Verbraucher
165 Daten lesen mit einem Datareader
169 Datenverarbeitung mit einem Dataset
179 Umwandlung zwischen Dataset und Datareader
181 Befehlsausführung mit Command-Objekten
183 Datenproviderunabhängiger Datenzugriff durch Providerfabriken
184 Asynchrone Befehlsausführung
186 Benachrichtigungen über Datenänderungen
189 Massenkopieren (Bulkcopy/Bulkimport)
192 Weitere neue Funktionen
197 ADO.NET 2.0 Feature-Matrix

Einführung

ADO.NET ist die zentrale Datenzugriffsschnittstelle für .NET-Anwendungen und Nachfolger der COM-basierten ActiveX Data Objects (ADO). Die Schreibweise *ActiveX Data Object .NET* wird jedoch selten verwendet; in der Regel findet man nur die Abkürzung. ADO.NET ist Teil der .NET-Klassenbibliothek (Namensraum `System.Data`).

Die Gemeinsamkeiten zwischen ADO und ADO.NET sind nicht sehr groß. Microsoft hat zentrale Teile des Datenzugriffs auf dem Weg von COM nach .NET stark verändert:

- Die Trennung in unterschiedliche Schnittstellen für verschiedene Zielgruppen (OLE DB und ADO) wurde aufgehoben; ADO.NET ist eine einheitliche Schnittstelle für alle Sprachen. ADO.NET Managed Data Provider ersetzen die bisherigen OLE DB-Provider.

- Das primäre Datenzugriffsmodell ist ein verbindungsloses Modell, bei dem die Daten nach dem Einlesen in ein so genanntes Dataset keine Verbindung mehr zu der Datenquelle haben. Ein Dataset ist eine Art In-Memory-Datenbank, die zu einem späteren Zeitpunkt mit der ursprünglichen Datenquelle oder einer anderen Datenquelle synchronisiert werden kann. Ein DataSet kann mehr als eine Tabelle enthalten; die Tabellen können hierarchische Beziehungen untereinander besitzen.
- Der Cursor-basierte Zugriff auf Datenbanken ist nur lesend möglich mit der Klasse `DataReader`. Für .NET 2.0 war zunächst geplant, ein schreibfähiges Pendant einzurichten. Diese Funktion hat Microsoft aber nach der Beta 1-Version wieder verworfen.
- Ein Dataset besitzt eine enge Verbindung zur Extensible Markup Language (XML): XML-Daten können in ein Dataset importiert und aus einem DataSet exportiert werden. Das DataSet kann sich in XML-Form serialisieren. Außerdem kann ein DataSet über das XML Document Object Model (DOM) bearbeitet werden.
- ADO.NET führt automatisch ein Verbindungs-Pooling durch, um bestehende Datenbankverbindungen wiederzuverwenden.

Neuerungen im Überblick

Die Datenbankzugriffsschnittstelle ADO.NET gehörte zu den Teilen des .NET Framework 1.0/1.1, die zahlreichen, an das klassische ADO gewöhnten Entwicklern nicht unerhebliche Umstellungsprobleme bereiteten, da es galt, sich auf das neue, verbindungslose Datenkonzept sowie auf das Fehlen einiger Funktionen einzustellen. Mit ADO.NET 2.0 legt Microsoft nun einige der vermissten Funktionen nach:

- Asynchrone Befehlsausführung
- Massenkopieren (Bulkcopy/Bulkimport)
- Mehrere aktive Datareader auf einer Verbindung (Multiple Active Result Sets – MARS)
- Benachrichtigungen über Datenänderungen
- Setzen der Anzahl der gleichzeitig zu übermittelnden Änderungen (Batch-Größe) für Datenadapter
- Umwandlung zwischen Dataset und Datareader
- Optional binäre (und damit schnellere) Serialisierung für Datasets
- Serialisierung einzelner `DataTable`-Objekte
- Datenproviderunabhängiges Programmieren durch Providerfabriken
- Ermittlung der auf einem System installierten Datenprovider
- Ermittlung der verfügbaren SQL Server-Installation in einer Domäne
- Auslesen des Datenbankschemas
- Statistiken über die Nutzung einer Datenbankverbindung
- Zusammensetzen von Verbindungszeichenfolgen mit dem `ConnectionStringBuilder`
- Verbesserungen beim Verbindungs-Pooling

- Ändern von SQL Server-Datenbankbenutzerkennwörtern
- Unterstützung für benutzerdefinierte SQL Server 2005-Datentypen
- Unterstützung für Snapshot Isolation im SQL Server 2005
- Unterstützung für Datenbankspiegelung (Client Failover) im SQL Server 2005
- Zugriff auf Datenbankschemata
- Verzicht auf MDAC für den ADO.NET Provider für Microsoft SQL Server

WICHTIG: Nicht alle der vorgenannten Funktionen sind für alle ADO.NET-Datenprovider verfügbar. Auskunft über die Verfügbarkeit gibt das Unterkapitel »ADO.NET 2.0 Feature-Matrix« am Ende des Kapitels 8. Zahlreiche Funktionen stehen leider nur in Zusammenhang mit dem Microsoft SQL Server 2005 (Codename »Yukon«) zur Verfügung.

HINWEIS: Der Datenzugriff in .NET 2.0 wurde auch durch die Erweiterung der datengebundenen Steuerelemente in Windows Forms und Web Forms sowie durch neue Funktionen in Visual Studio 2005 verbessert. Diese Funktionen werden in den entsprechenden Kapiteln erläutert.

Verlorene Funktionen

Alle Funktionen aus ADO.NET 1.0 und 1.1 sind auch in ADO.NET 2.0 noch enthalten. Jedoch werden einige Funktionen, die zwischenzeitlich geplant und in den Alpha-Versionen von .NET 2.0 auch zum Teil schon implementiert waren, laut derzeitigem Stand nicht in die finale Version Einzug halten und sie sind zum Teil auch schon in der Beta-Version entfallen oder als obsolet gekennzeichnet:

- Paging-Unterstützung im DataReader (Methode `ExecutePageReader()` im `Command`-Objekt).
- Serverseitige Cursor als schreibendes Pendant zum DataReader (Klasse `SqlResultSet`).
- Asynchroner Aufbau von Datenbankverbindungen.
- Klasse `SqlDataTable` zur Vereinfachung der Arbeit mit DataSets.
- SQL Command Sets zur Ausführung von mehreren Befehlen in einem Block.

Es wird spekuliert, ob diese Funktionen in einer späteren Aktualisierung (vergleichbar mit dem ODBC-Dataprovider unter .NET 1.0) als ADO.NET 2.1 oder erst mit der kommenden Windows-Version »Longhorn« wieder in das .NET Framework integriert werden.

HINWEIS: Auch der ursprünglich für .NET 2.0 angekündigte und in der auf der PDC 2003 verteilten Alpha-Version enthaltene Objekt-Relationale Mapper (ORM) mit Namen *ObjectSpaces* wurde von Microsoft im Frühjahr 2004 auf das Longhorn-Zeitalter verschoben, da sich Überschneidungen mit der in Longhorn enthaltenen Datenbanktechnologie WinFS ergeben haben.

Die ADO.NET-Architektur

Im Veröffentlichen von Datenbankschnittstellen ist Microsoft seit vielen Jahren Weltmeister: ODBC, OLE DB, RDO, DAO, ADO und ADO.NET. Mit ADO.NET hat Microsoft auf seine Universal Data Access (UDA)-Strategie im wahrsten Sinne des Wortes noch eins »draufgesetzt« (siehe folgende Abbildung). Die Remote Data Objects (RDO) und die Data Access Ob-

jects (DAO) sowie die Open Database Connectivity (ODBC) gelten dabei schon länger als veraltet. Außerdem nicht ausdrücklich in der Grafik berücksichtigt wurde die Möglichkeit, von Managed Code aus via COM-Interoperabilität und P/Invoke die alten Schnittstellen zu nutzen.

Genauso wie ODBC und OLE DB verwendet ADO.NET auch datenquellenspezifische Treiber, die *ADO.NET Data Provider*, *.NET Data Provider* oder *Managed Provider* genannt werden. Data Provider für OLE DB und ODBC stellen dabei die Abwärtskompatibilität von ADO.NET für Datenquellen her, für die (noch) keine spezifischen ADO.NET-Datenprovider existieren.

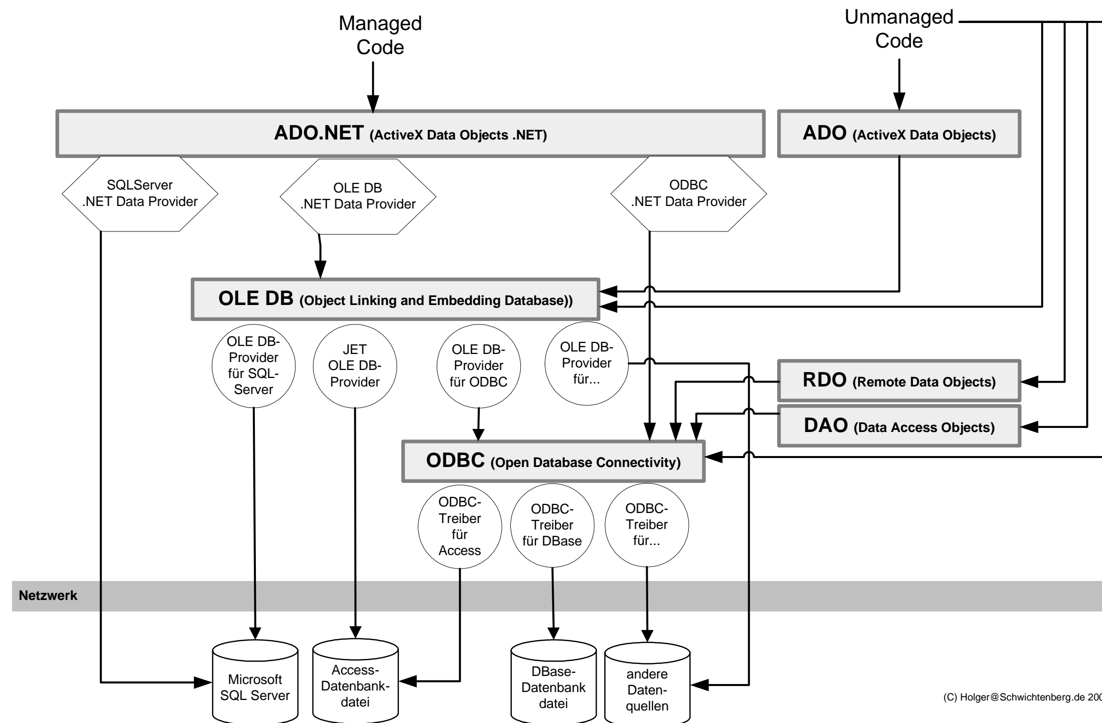

***Abbildung 9.1**: ADO.NET im Zusammenspiel mit anderen Datenzugriffsschnittstellen*

Provider von Microsoft

ADO.NET 2.0 wird mit folgenden Datenprovidern (alias .NET Data Provider oder Managed Data Provider) ausgeliefert:

- `System.Data.SqlClient` (spezieller Treiber für Microsoft SQL Server 7.0/2000 und 2005)
- `System.Data.SqlServer` (In-Proc-Treiber für MS SQL 2005, neu in .NET 2.0)
- `System.Data.SqlServerCe` (spezieller Treiber für Microsoft SQL Server CE)
- `System.Data.OracleClient` (spezieller Treiber für Oracle-Datenbanken)
- `System.Data.OleDb` (Brücke zu OLE DB-Providern)
- `System.Data.Odbc` (Brücke zu ODBC-Treibern)

HINWEIS: Eine interne Änderung in ADO.NET 2.0, die sich nicht direkt in den Klassen widerspiegelt, besteht darin, dass der SQL Server-Datenprovider (`System.Data.SqlClient`) nicht mehr auf den Microsoft Data Access Components (MDAC) basiert, sondern in seiner Implementierung völlig unabhängig davon ist.

Provider von anderen Herstellern

Weitere Provider werden von anderen Herstellern geliefert, z. B.

- Datenprovider für MySQL (MySQLDirect .NET Data Provider) [ADONET01]
- Datenprovider für Oracle, DB2, Sybase, Microsoft SQL Server [ADONET02]
- Open Source-Datenprovider für Firebird [ADONET03]
- Open Source ADO.NET Provider für MySql und PostgresSql [ADONET04]
- Datenprovider für MySql, Informix, DB2, Oracle, Ingres, Sybase und Microsoft SQL Server [ADONET05]

TIPP: Weitere ADO.NET-Datenprovider finden Sie in der Werkzeug- und Komponentenreferenz des Autors: [DOTNET02]

Ermittlung der installierten Datenprovider

Die auf einem System vorhandenen ADO.NET-Datenprovider können über die Methode `System.Data.Common.DbProviderFactories.GetFactoryClasses()` aufgelistet werden. Diese Funktion ist neu in ADO.NET 2.0. Ein Fernzugriff auf die Provider eines anderen Systems ist jedoch leider nicht möglich.

Beispiel

Im folgenden Beispiel werden alle auf dem lokalen System vorhandenen Datenprovider an der Konsole ausgegeben.

```
public static void GetAllProviders()
{
 Console.WriteLine("=== DEMO Providerliste");
 // --- Ermittlung der Provider
 DataTable providers = System.Data.Common.DbProviderFactories.GetFactoryClasses();
 // --- Ausgabe
 foreach (DataRow provider in providers.Rows)
 {
  foreach (DataColumn c in providers.Columns)
   Console.WriteLine(c.ColumnName + ":" + provider[c]);
  Console.WriteLine("--");
 }
```

***Listing 9.1**: Auflistung der vorhandenen ADO.NET-Datenprovider [Enumerationen.cs]*

Der Weg der Daten von der Datenquelle bis zum Verbraucher

Die nachstehende Abbildung zeigt die möglichen Datenwege in ADO.NET 2.0 von einer Datenquelle zu einem Datenverbraucher. Alle Zugriffe auf eine Datenquelle laufen auf jeden Fall über ein `Command`-Objekt, das datenproviderspezifisch ist. Zum Auslesen von Daten bietet das Modell zwei Wege: Daten können über einen providerspezifischen `DataReader` oder ein providerunabhängiges `DataSet` zum Datenverbraucher gelangen. Das `DataSet` benötigt zur Beschaffung der Daten ein `DataAdapter`-Objekt, das wiederum in jedem Datenprovider separat zu implementieren ist. Ab .NET 2.0 existieren Möglichkeiten, nachträglich noch von einem in das andere Zugriffsmodell zu wechseln. Datenänderungen erfolgen, indem der Datenverbraucher direkt Befehle an ein `Command`-Objekt sendet. Ab .NET 2.0 stellt .NET so genannte Datenquellensteuerelemente bereit, die dem Entwickler die Bindung von Daten an ein Steuerelement erleichtern. Dabei unterscheidet sich die Architektur in Windows Forms und Web Forms: Während Windows Forms mit einer allgemeinen Klasse `BindingSource` auf Basis von typisierten Datasets arbeiten, verwendet ASP.NET providerspezifische Klassen (z. B. `SqlDataSource`, `AccessDataSource`).

ACHTUNG: Für einige Anwendungen ist es wichtig, sich nicht auf einen speziellen Datenprovider festzulegen. Die Möglichkeiten, providerunabhängig zu programmieren, wurden in .NET 2.0 verbessert. Mehr dazu werden Sie im Unterkapitel »Providerunabhängige Datenzugriff durch Providerfabriken« erfahren.

HINWEIS: Die Tatsache, dass einige Klassen in providerspezifischen Ausprägungen (mit zum Teil sehr unterschiedlichen Namen) vorliegen, macht die schriftliche Darstellung einiger Sachverhalte schwierig. Dieses Buch folgt dem üblichen Entwicklersprachgebrauch, die Begriffe Datareader, Datenadapter, Command und Connection als Technologie-Oberbegriffe für die jeweiligen providerspezifischen Klassen zu nutzen – auch wenn diese Klassennamen in der Klassenbibliothek gar nicht existieren. Um den Text zu straffen, wird daher an verschiedenen Stellen synonym zu »Das `DataReader`-Objekt…« (mit Auszeichnung des Objektnamens in der Schriftart Courier) einfach »Der Datareader…« (ohne andere Schriftart) verwendet.

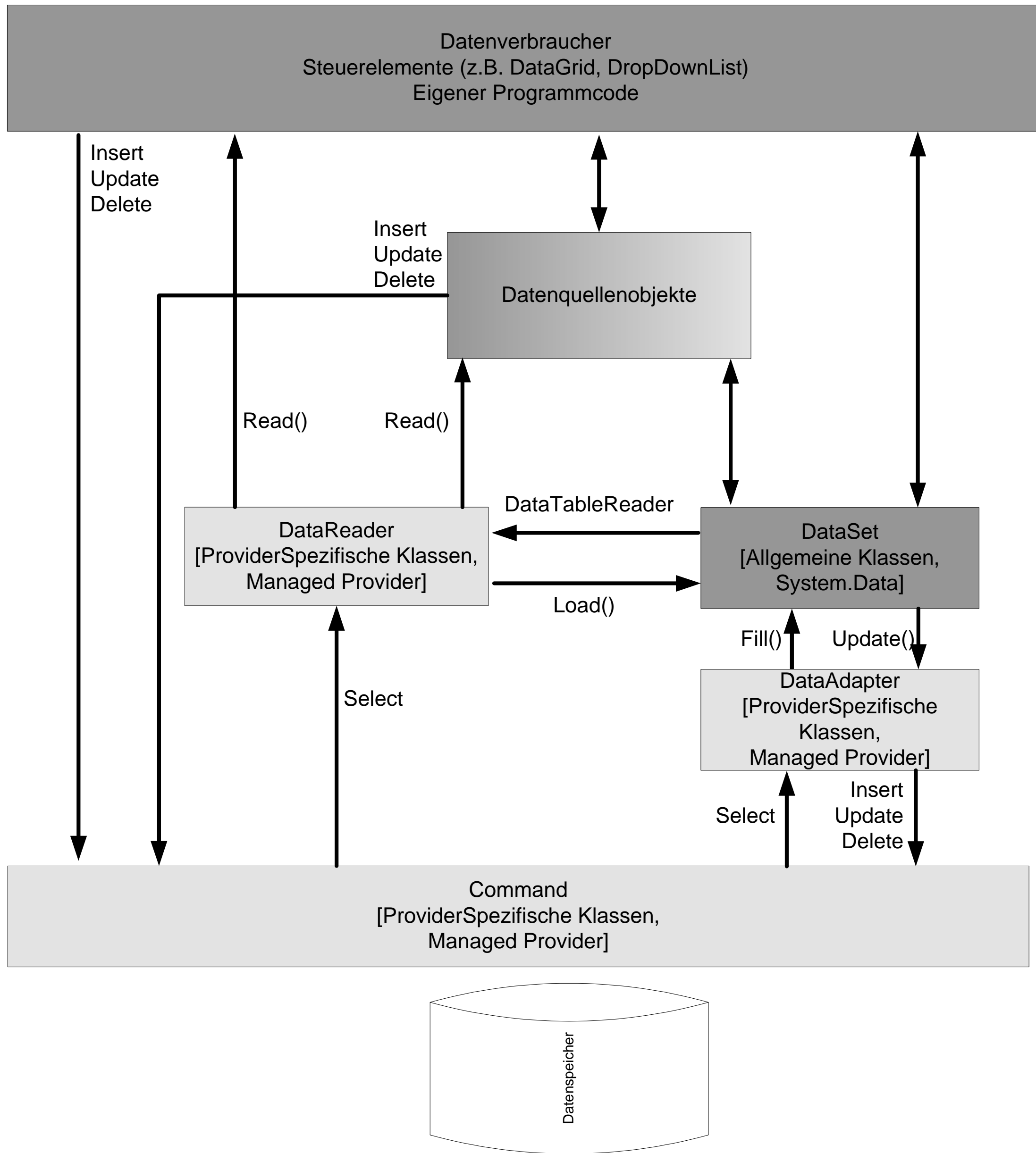

***Abbildung 9.2**: Datenwege in ADO.NET 2.0*

	Datareader	Dataset
Implementiert in	Jedem Datenprovider	`System.Data`

Basisklassen	`DbDataReader`
	`MarshalByRefObject`
	`Object`
		`MarshalByValueComponent`
		`Object`

▶

	Datareader	Dataset
Schnittstellen	`IDataReader`, `IDisposable`, `IDataRecord`, `IEnumerable`	`IListSource`, `IXmlSerializable`, `ISupportInitialize`, `ISerializable`
Daten lesen	Ja	Ja
Daten vorwärts lesen	Ja	Ja
Daten rückwärts lesen	Nein	Ja
Direktzugriff auf beliebigen Datensatz	Nein	Ja
Direktzugriff auf beliebige Spalte in Datensatz	Ja	Ja
Daten verändern	Nein, nur über separate *Command*-Objekte	Ja (über Datenadapter)
Befehlserzeugung für Datenänderung	Komplett manuell	Tlw. automatisch (`CommandBuilder`)

***Tabelle 9.1**: Datareader vs. Dataset*

Typisierte Datasets

So genannte typisierte Datasets (engl. *Typed Dataset*) sind eine weitere Abstraktionsform des Datenzugriffs, die von Visual Studio angeboten wird. Ein typisiertes Dataset ist im Kern eine Klasse, die einen Wrapper um die ADO.NET-`DataSet`-Klasse bildet. Die Wrapper-Klasse stellt die Spalten der enthaltenen Tabelle als Attribute in Tabellen-Objekten zur Verfügung und bietet Methoden zum Holen, Ändern, Hinzufügen und Löschen von Daten.

Neben der Wrapper-Klasse gehören zu einem typisierten Dataset auch eine XML-Schema-Beschreibung (*.xsd*-Datei) und die Beschreibung der Anordnung der Daten in der grafischen Ansicht (*.xss*- und *.xcs*-Dateien).

Visual Studio 2005 besitzt für Datasets eine eigene grafische Entwurfsoberfläche, die per Drag&Drop aus dem Server Explorer befüllt werden und auch Verknüpfungen zwischen `DataTable`-Objekten in einem `DataSet`-Objekt modellieren kann. Das typisierte Dataset kann zur Drag&Drop-Datenbindung in Windows Forms-Fenstern oder per Programmcode verwendet werden.

Typisierte Datasets existieren bereits seit Visual Studio .NET 2002, wurden aber in Visual Studio 2005 stark verändert. Ihr Nachteil ist die große Menge generierten Codes (ca. 1600 Zeilen für zwei verknüpfte Tabellen).

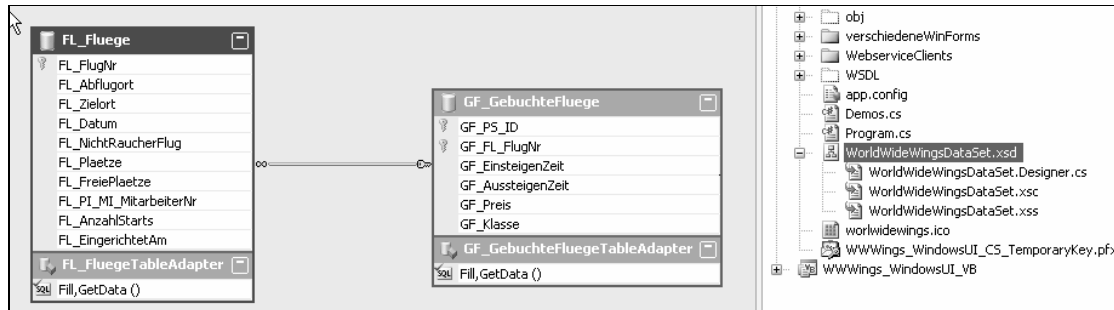

***Abbildung 9.3**: Verknüpfung zweier Tabellen im Designer für typisierte DataSet in Visual Studio 2005*

Daten lesen mit einem Datareader

Bei einem `DataReader`-Objekt handelt es sich um einen serverseitigen Cursor, welcher unidirektionalen Lesezugriff (nur vorwärts) auf das Ergebnis einer SELECT-Anwendung (Resultset) erlaubt. Eine Veränderung der Daten ist nicht möglich. Im Gegensatz zum `DataSet`, unterstützt der `DataReader` nur eine flache Darstellung der Daten. Die Datenrückgabe erfolgt immer zeilenweise, deshalb muss über die Ergebnismenge iteriert werden. Verglichen mit dem klassischen ADO entspricht ein ADO.NET-DataReader einem »read-only« und »forward-only« Recordset (»Vorwärtscursor«).

Jeder ADO.NET Datenprovider implementiert seine eigene `DataReader`-Implementierung, sodass es zahlreiche verschiedene `DataReader`-Klassen im .NET Framework gibt (z. B. `SqlDataReader` und `OleDbDataReader`). Die `DataReader`-Klassen sind abgeleitet von `System.Data.ProviderBase.DbDataReaderBase` und implementieren `System.Data.IDataReader`.

Ein `DataReader` benötigt zur Beschaffung der Daten immer ein `Command`-Objekt, das ebenso providerspezifisch ist (z. B. `SqlCommand` und `OleDbCommand`). Für die Verbindung zur Datenbank selbst wird ein providerspezifisches `Connection`-Objekt (z. B. `SqlConnection` oder `OleDbConnection`) benötigt. Die nachstehenden Abbildungen zeigen den Zusammenhang dieser Objekte am Beispiel der Datenprovider für OLE DB und SQL Server. Bei dem Provider für SQL Server (`SqlClient`) existiert ab .NET 2.0 eine zusätzliche Klasse `SqlRecord`, die einen einzigen Datensatz als Ergebnis eines Befehls repräsentiert.

Abbildung 9.4: Objektmodell für den Datenprovider für OLE DB

Abbildung 9.5: Objektmodell für den Datenprovider für Microsoft SQL Server

HINWEIS: Das Connection-Objekt muss explizit mit `Open()` geöffnet werden, bevor Sie `ExecuteReader()` ausführen.

Im Gegensatz zum `DataSet`-Objekt dürfen Sie das `DataReader`-Objekt und das `Connection`-Objekt erst durch den Aufruf der Methode `Close()` schließen, wenn Sie alle Daten gelesen haben. Danach sollten Sie aber auch die Verbindung schließen – selbst wenn Sie in Kürze weitere Daten lesen wollen. ADO.NET besitzt einen integrierten Verbindungs-Pooling-Mechanismus, der die effiziente Wiederverwendung von Datenbankverbindungen sicherstellt.

Ablauf

Das Lesen von Daten mit einem `DataReader`-Objekt geht in folgenden Schritten vor sich:

- Aufbau einer Verbindung zu der Datenbank mit einem `Connection`-Objekt. Bei der Instanziierung dieses Objekts kann die Verbindungszeichenfolge übergeben werden.
- Instanziierung der Klasse `Command` und Bindung dieses Objekts an das `Connection`-Objekt über die Eigenschaft `Connection`.
- Festlegung eines SQL-Befehls, der Daten liefert (also z. B. SELECT oder eine Stored Procedure), im `Command`-Objekt in der Eigenschaft `CommandText`.
- Die Ausführung der Methode `ExecuteReader()` in der `Command`-Klasse liefert als Ergebnis ein `DataReader`-Objekt.

Danach kann der `DataReader` entweder an ein datenkonsumierendes Steuerelement (z. B. `DataGrid` – siehe Kapitel über ASP.NET und Windows Forms) gebunden oder per Programmcode durchlaufen werden. Zum Durchlauf per Programmcode stellt das `DataReader`-Objekt die Methode `Read()` bereit, die jeweils den nächsten Datensatz liest. Anders als beim klassischen ADO steht der Cursor zu Beginn nicht auf dem ersten Datensatz, sondern vor diesem. Wie beim Auslesen von Dateien muss man den Cursor so lange vorwärts setzen, bis `Read()` als Ergebnis *false* liefert. `Item("Spaltenname")` oder `Item[SpaltenIndex]` liefert dann für die jeweils aktuelle Zeile den Inhalt einer Spalte als `System.Object`. Einen spezifischen Datentyp erhält man durch Mitglieder wie beispielsweise `GetString()`, `GetInt32()`, `GetFloat()` oder `GetGuid()`. Diese Methoden unterstützen aber leider nur den indexbasierten Zugriff. Der Index beginnt immer bei `0`.

Beispiel

Das Beispiel listet die Spalten *FL_FlugNr* und *FL_Abflugort* der Tabelle *FL_Zielort* auf. Während *FL_FlugNr* über den Spaltennamen adressiert wird, erfolgt die Nutzung der anderen beiden Spalten über den Spaltenindex. Der Spaltenindex für die zweite Spalte ist 1, da die Zählung bei 0 beginnt.

```
// === Daten lesen mit einem DataReader
  public void DataReader_Demo()
  {
   Demo.PrintHeader("Liste der Flüge (Datareader-Demo)");

   const string CONNSTRING = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=marl\sqlexpress";
   const string SQL = "Select * from FL_Fluege";

   // Verbindung aufbauen
   SqlConnection sqlConn = new SqlConnection(CONNSTRING);
   sqlConn.Open();
   // Befehl ausführen
   SqlCommand sqlCmd = sqlConn.CreateCommand();
   sqlCmd.CommandText = SQL;
   // Datareader erzeugen
   SqlDataReader dr = sqlCmd.ExecuteReader();

   while (dr.Read())
   {
    Demo.Print("Flug-ID: " + dr["FL_FlugNr"] + " von " + dr.GetString(1) + " nach " + dr.GetString(2));
   }
```

```
    // Schließen
    dr.Close();
    sqlConn.Close();
   }
  }
```

***Listing 9.2**: Daten lesen mit einem DataReader*

```
Datareader Demo
Website-ID: 1 URL: http://www.dotnetframework.de
Website-ID: 2 URL: http://www.windows-scripting.de
Website-ID: 3 URL: http://www.aspnetdev.de
Website-ID: 4 URL: http://www.IT-Visions.de
```

***Listing 9.3**: Ausgabe des Beispiels*

Hilfsroutine PrintReader

Die nachstehende Hilfsroutine `PrintReader()` gibt ein beliebiges Objekt mit der `IDataReader`-Schnittstelle komplett aus (alle Zeilen, alle Spalten). `PrintReader()` kommt in verschiedenen nachfolgenden Beispielen zum Einsatz, um den Quellcode auf das Wesentliche fokussieren zu können.

```
public static void PrintReader(IDataReader reader)
  {
   while (reader.Read())
    for (int i = 0; i < reader.FieldCount; i++)
     Console.WriteLine("Spalte: " + reader.GetName(i) + "\t = " + reader.GetValue(i));
  }
```

***Listing 9.4**: Hilfsroutine PrintReader()*

Multiple Active Results Sets (MARS)

In ADO.NET 1.x konnte zu einem Zeitpunkt pro Verbindung nur ein Datareader aktiv sein; es war also zum Beispiel nicht möglich, zwei Datareader gleichzeitig auf einer Verbindung zu durchlaufen. Wenn schon ein Datareader geöffnet ist, führt das Öffnen eines zweiten zu der Fehlermeldung »There is already an open Datareader associated with this Command which must be closed first.« (*Es gibt bereits einen geöffneten Datareader, der mit diesem Command verbunden ist und zunächst geschlossen werden muss.*) Diese Architektur kann als **Single Active Results Sets (SARS)** bezeichnet werden.

ADO.NET 2.0 unterstützt hingegen zusätzlich **Multiple Active Results Sets (MARS)**, also die Mehrfachverwendung einer Verbindung. Durch MARS können sowohl Abfragen als auch SQL DML-Befehle (***D**ata **M**anipulation **L**anguage*, Datenmanipulationssprache – dazu gehören INSERT, UPDATE, DELETE) gleichzeitig auf einer Verbindung ausgeführt werden. MARS ist jedoch zunächst nur für den SQL Server 2005 verfügbar und dort standardmäßig aktiviert.

Beispiel

Das folgende Beispiel zeigt, wie innerhalb der Schleife über eine Menge von Flügen mit einem Datareader auf der gleichen Datenbankverbindung zunächst ein weiteres SELECT und dann ein DELETE ausgeführt wird, um alle mit den ausgewählten Flügen in Beziehung stehenden Buchungen zu löschen.

```
public void Mars_Demo()
  {
   Demo.PrintHeader("Buchungen für bestimmte Flüge löschen (MARS, nur SQL Server 2005!)");

   const string CONNSTRING = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=Marl\sqlexpress";
   const string SQL1 = "Select * from FL_Fluege where Fl_Abflugort = 'Frankfurt'";

   // Verbindung aufbauen
   SqlConnection sqlConn = new SqlConnection(CONNSTRING);
   sqlConn.Open();
   // Befehl ausführen
   SqlCommand sqlCmd = sqlConn.CreateCommand();
   sqlCmd.CommandText = SQL1;
   // Datareader erzeugen
   SqlDataReader reader1 = sqlCmd.ExecuteReader();

   // --- SChleife über alle relevanten Flüge
   while (reader1.Read())
   {
    // Befehl ausführen
    Demo.Print("Buchungen für Flug: " + reader1["FL_FlugNr"]);
    // --- Buchungen auflisten
    string SQL2 = @"Select GF_PS_ID from GF_GebuchteFluege where GF_FL_FlugNr = " + reader1["FL_FlugNr"];
    SqlCommand sqlCmd2 = sqlConn.CreateCommand();
    sqlCmd2.CommandText = SQL2;
    SqlDataReader reader2 = sqlCmd2.ExecuteReader();
    Demo.PrintReader(reader2);
    reader2.Close();
    Demo.Print("Lösche diese Buchungen...");
    // --- Buchungen löschen
    string SQL3 = @"Delete from GF_GebuchteFluege where GF_FL_FlugNr = " + reader1["FL_FlugNr"];
    SqlCommand sqlCmd3 = sqlConn.CreateCommand();
    sqlCmd3.CommandText = SQL3;
    int anz = sqlCmd3.ExecuteNonQuery();
    Demo.Print("Befehl ausgeführt: " + anz + " Buchungen gelöscht!");
   }
   reader1.Close();
   sqlConn.Close();
  }
```

***Listing 9.5**: Beispiel für MARS bei Verbindungen zum SQL Server 2005*

Datenverarbeitung mit einem Dataset

Ein `DataSet` enthält eine Sammlung von Datentabellen, welche durch einzelne `DataTable`-Objekte dargestellt werden. Die `DataTable`-Objekte können aus beliebigen Datenquellen gefüllt werden, ohne dass eine Beziehung zwischen dem Objekt und der Datenquelle existiert; das `DataTable`-Objekt weiß nicht, woher die Daten kommen. Die `DataTable`-Objekte können auch ohne Programmcode zeilenweise mit Daten befüllt werden; eine Datenbank ist nicht notwendig.

Ein `DataSet` bietet – im Gegensatz zum `DataReader` – alle Zugriffsarten, also auch das Hinzufügen, Löschen und Ändern von Datensätzen. Auch lassen sich hierarchische Beziehungen zwischen

einzelnen Tabellen darstellen und im `DataSet` speichern. Dadurch ist eine Verarbeitung hierarchischer Datenmengen möglich.

Ein Dataset ist ein clientseitiger Daten-Cache, der die Änderung mitprotokolliert. Das Konzept eines serverseitigen Cursors ist in ADO.NET nur durch die `DataReader`-Klasse realisiert.

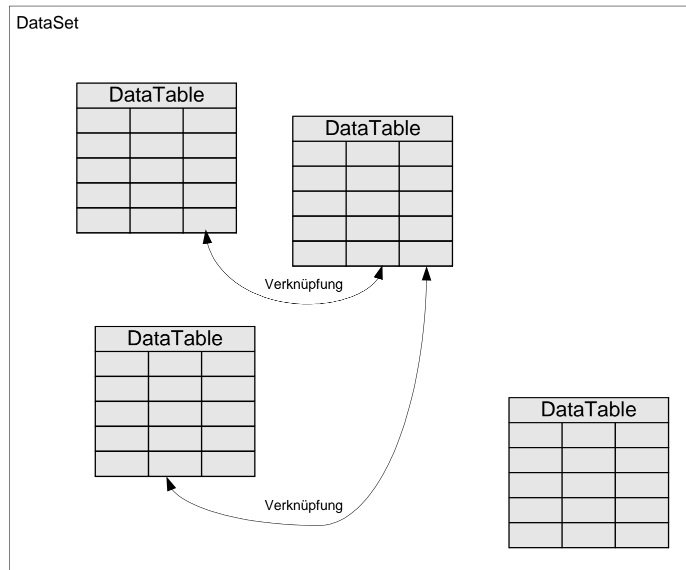

Abbildung 9.6: *Aufbau eines DataSet*

Das Objektmodell

Im Gegensatz zum Recordset im klassischen ADO ist ein ADO.NET-Dataset konsequent objektorientiert. Ein `DataSet`-Objekt besteht aus einer Menge von `DataTable`-Objekten (`DataTableCollection`). Jedes `DataTable`-Objekt besitzt über das Attribut `DataSet` einen Verweis auf das Dataset, zu dem es gehört. Das `DataTable`-Objekt besitzt eine `DataColumnCollection` mit `DataColumn`-Objekten für jede einzelne Spalte in der Tabelle und eine `DataRowCollection` mit `DataRow`-Objekten für jede Zeile. Innerhalb eines `DataRow`-Objekts kann man die Inhalte der Zellen durch das indizierte Attribut `Item` abrufen. `Item` erwartet alternativ den Spaltennamen, den Spaltenindex oder ein `DataColumn`-Objekt.

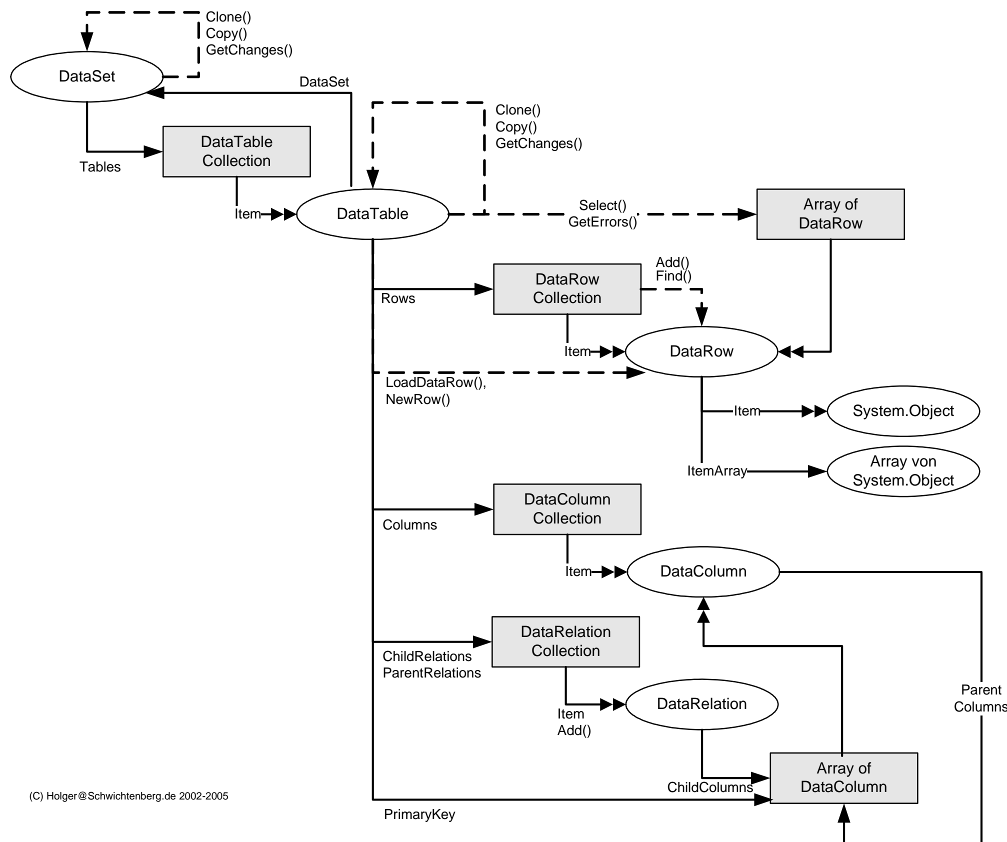

***Abbildung 9.7**: Objektmodell des DataSet*

Daten lesen mit DataSets

Ein Dataset benötigt zum Einlesen von Daten einen providerspezifischen Datenadapter. Das Lesen von Daten mit einem Dataset erfolgt in folgenden Schritten:

- Aufbau einer Verbindung zu der Datenbank mit einem `Connection`-Objekt. Bei der Instanziierung dieses Objekts kann die Verbindungszeichenfolge übergeben werden.
- Instanziierung der Klasse `Command` und Bindung dieses Objekts an das `Connection`-Objekt über die Eigenschaft `Connection`.
- Festlegung eines SQL-Befehls, der Daten liefert (also z. B. SELECT oder eine Stored Procedure), im `OleDbCommand`-Objekt in der Eigenschaft `CommandText`.
- Instanziieren des Datenadapters auf Basis des `Command`-Objekts.
- Instanziierung des `DataSet`-Objekts (ohne Parameter).
- Die Ausführung der Methode `Fill()` in dem `DataSet`-Objekt kopiert die kompletten Daten in Form eines `DataTable`-Objekts in das Dataset. Als zweiter Parameter kann bei `Fill()` der Aliasname für das `DataTable`-Objekt innerhalb des `DataSet` angegeben werden. Ohne diese Angabe erhält das `DataTable`-Objekt den Namen *Table.*

- Optional können weitere Tabellen eingelesen und im Dataset miteinander verknüpft werden.
- Danach kann die Verbindung sofort geschlossen werden.

TIPP: Die Datenbankverbindung muss vor dem Aufruf von `Fill()` nicht explizit geöffnet werden; der Datenadapter erledigt dies, wenn die angegebene Datenbankverbindung noch nicht geöffnet ist. Nach dem Abholen der Daten muss keine aktive Verbindung zur Datenquelle mehr bestehen. Daher kann innerhalb des `DataSet`-Objekts navigiert und geändert werden, ohne eine Verbindung zur Datenbank offen zu halten, weil die Daten auf dem Client vorliegen. Das Dataset ist ein Cache für Daten.

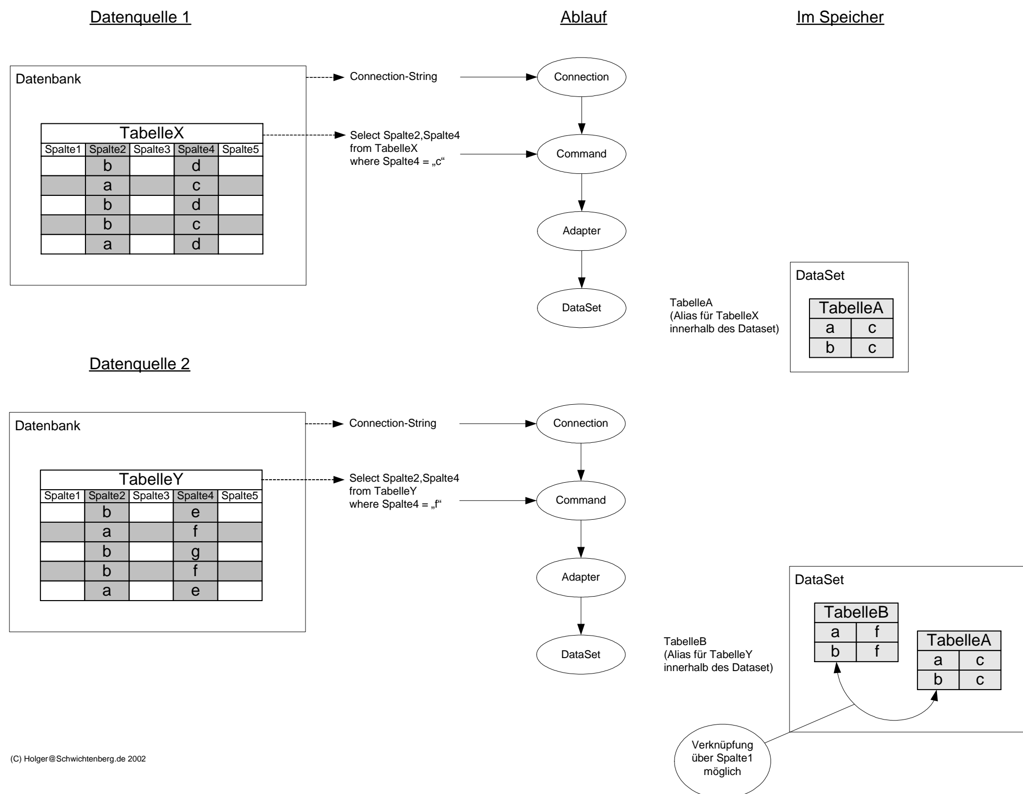

***Abbildung 9.8**: Einlesen von mehreren Datenquellen in ein DataSet*

Beispiel 1: Flache Daten

Analog zum Beispiel für den Einsatz des Datareader wird nun gezeigt, wie die gleiche Ausgabe mit einem Dataset erzeugt werden kann.

```
// === Daten lesen mit einem DataSet
  public void DataSet_Lesen()
  {
   Demo.PrintHeader("Datareader Demo");

   const string CONNSTRING = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=MARL\sqlexpress";
```

```
  const string SQL = "Select * from FL_Fluege";

  // --- Verbindung aufbauen
  SqlConnection conn = new SqlConnection(CONNSTRING);
  conn.Open();
  // --- Befehl ausführen
  SqlCommand cmd = new SqlCommand(SQL, conn);
  // --- Datenadapter erzeugen
  SqlDataAdapter da = new SqlDataAdapter(cmd);
  // --- DataSet erzeugen
  DataSet ds = new DataSet();
  // --- Daten abholen
  da.Fill(ds);
  // --- Verbindung jetzt schon schließen!
  conn.Close();
  // --- Zugriff auf die einzige Tabelle
  DataTable dt = ds.Tables[0];
  Demo.Print("Name der Tabelle: " + dt.TableName);
  // --- Iteration über Daten
  foreach (DataRow dr in dt.Rows)
  {
   Demo.Print("Flug-ID: " + dr["FL_FlugNr"] + " von " + dr[1] + " nach " + dr[2]);
  }
 }
```

***Listing 9.6**: Ausgabe DerTabelle FL_Fluege mit einem DataSet*

Beispiel 2: Tabellenverknüpfungen

Das folgende Beispiel zeigt, dass man mit Hilfe mehrerer Datenadapter mehrere Datenmengen in ein `DataSet`-Objekt laden und diese dort dann optional mit Hilfe eines `DataRelation`-Objekts hierarchisch verknüpfen kann. Bei der Ausgabe können dann zu jedem Datensatz der Master-Tabelle die entsprechenden Detaildatensätze angezeigt werden (bzw. umgekehrt). In dem nachfolgenden Beispiel wird die Verbindung zwischen *FL_Fluege* (Master-Tabelle) und *Bücher* (Detailtabelle) hergestellt.

Das `DataRelation`-Objekt erwartet bei der Instanziierung einen Namen für die Verknüpfung (hier: *FL_Fluege_Books*) sowie zwei `DataColumn`-Objekte mit der Spalte aus der Master-Tabelle (Primärschlüssel) und der Spalte aus der Detailtabelle (Fremdschlüssel), über die die Verknüpfung erstellt werden soll. Wenn mehrere Spalten für die Verknüpfung notwendig sind, kann auch jeweils ein Array mit `DataColumn`-Objekten übergeben werden. Sie müssen das neue `DataRelation`-Objekt explizit an die `Relations`-Menge des `DataSet` anfügen, da sonst die erstellte Beziehung nicht wirkt.

HINWEIS: ADO.NET stellt Beziehungen im Dataset nicht automatisch aus vorhandenen Beziehungen in der Datenquelle her. Sie müssen die Beziehungen im `DataSet`-Objekt immer explizit definieren.

Innerhalb der Schleife über alle Zeilen in der Tabelle *FL_Fluege* kann dann für jede `DataRow` mit Hilfe der Methode `GetChildRows()` die Menge der zugehörigen Buchungs-Datensätze abgerufen werden. `GetChildRows()` erwartet als Parameter ein `DataRelation`-Objekt oder den Namen der Beziehung. Über die umgekehrte Navigationsrichtung von Detail- zu Mastertabelle existieren die Methoden `GetChildRow()` und `GetChildRows()`.

```
// === Daten verknüpfen in einem DataSet
  public void Dataset_Beziehungen()
  {
   Demo.PrintHeader("Passagiere mit ihren Flügen (Dataset-Beziehungen-Demo)");

   const string CONNSTRING = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=MARL\sqlexpress";
   const string SQL1 = "Select * from AllePassagiere";
   const string SQL2 = "Select * from GF_GebuchteFluege";

   // --- Verbindung aufbauen
   SqlConnection conn = new SqlConnection(CONNSTRING);
   conn.Open();
   // --- Leeres DataSet erzeugen
   DataSet ds = new DataSet();
   // --- Datenadapter erzeugen
   SqlDataAdapter da1 = new SqlDataAdapter(SQL1, CONNSTRING);
   SqlDataAdapter da2 = new SqlDataAdapter(SQL2, CONNSTRING);
   // --- Daten abholen
   da1.Fill(ds, "AllePassagiere");
   da2.Fill(ds, "GebuchteFluege");
   // --- Verbindung jetzt schon schließen!
   conn.Close();
   // --- Verknüpfung herstellen
   DataColumn dc1 = ds.Tables["AllePassagiere"].Columns["PS_ID"];
   DataColumn dc2 = ds.Tables["GebuchteFluege"].Columns["GF_PS_ID"];
   DataRelation drel = new DataRelation("Passagiere_GebuchteFluege", dc1, dc2);
   ds.Relations.Add(drel);

   // --- Zugriff auf Tabelle
   DataTable dt = ds.Tables["AllePassagiere"];
   // --- Iteration über Daten
   foreach (DataRow dr in dt.Rows)
   {
    Demo.Print("Name: " + dr["PE_Name"] + " Vorname: " + dr["PE_Vorname"]);
    foreach (DataRow dr2 in dr.GetChildRows(drel))
    {
     Demo.Print("  Flug: " + dr2["GF_FL_FlugNr"] + " Datum: " + dr2["GF_Flugdatum"]);
    }
   }
  }
```

***Listing 9.7**: Verknüpfen von FL_Fluege und Büchern in einem DataSet*

```
Name: Schröder Vorname: Gerhard
  Flug: 102 Datum: 01.03.2005 00:00:00
  Flug: 203 Datum: 02.03.2005 00:00:00
Name: Merkel Vorname: Angela
  Flug: 200 Datum: 04.03.2005 00:00:00
```

***Listing 9.8**: Ausgabe des Beispiels*

TIPP: Ein `DataAdapter`-Objekt kann mit mehreren durch Semikola getrennten SELECT-Befehlen initialisiert werden. Der Datenadapter erzeugt automatisch für jedes einzelne SELECT eine eigene Tabelle, die dann automatisch »Table«, »Table1«, »Table2« usw. heißen. Mit Hilfe der `TableMappings`-Menge im Datenadapter kann man aber auch sinnvollere Namen vergeben. Die-

se Nutzung mehrerer SELECTs im Datenadapter wird nicht von allen ADO.NET-Datenprovidern unterstützt.

Daten ändern mit Datasets

Die Manipulation der Daten in einem `DataSet`-Objekt ist sehr einfach:

- Jede Zelle kann direkt beschrieben werden. Ein Editiermodus muss nicht aktiviert werden.
- Zum Löschen eines Datensatzes ruft man auf dem entsprechenden `DataRow`-Objekt die Methode `Delete()` auf.
- Zum Anfügen eines Datensatzes muss man mit der Fabrik-Methode `NewRow()` der entsprechenden Tabelle ein `DataRow`-Objekt erzeugen und dieses anschließend der `Rows`-Auflistung des `DataTable`-Objekts hinzufügen.

Anders als im klassischen ADO müssen geänderte Zeilen nicht einzeln bestätigt werden. Das `DataSet` speichert während der Datenmanipulation immer die geänderten und die originalen Werte. Die `AcceptChanges()`-Methode des `DataSet`-Objekts überführt die geänderten Werte in die Originalwerte, während die `RejectChanges()`-Methode die Änderungen verwirft.

HINWEIS: Neben der Methode `Delete()` im `DataRow`-Objekt besitzt auch die `DataRowCollection` ein `Remove()` zum Löschen einer Zeile. Der letztgenannte Befehl führt im Gegensatz zu dem ersten automatisch ein `AcceptChanges()` aus.

Um die Änderungen über den Datenadapter an die Datenquelle zurückzugeben, ist im `DataAdapter`-Objekt die Methode `Update()` mit dem `DataSet`-Objekt als Parameter aufzurufen. Der Datenadapter sendet dann die Änderungen in Form von SQL-DML-Befehlen (INSERT, UPDATE, DELETE) an die Datenquelle.

Leider erzeugt auch in .NET 2.0 der Datenadapter die notwendigen SQL-DML-Befehle nicht selbst. Vorgesehen ist, dass der Entwickler die Befehle selbst in dem Datenadapter ablegt. Von dieser lästigen Arbeit kann er sich aber etwas entlasten durch die so genannten `CommandBuilder`-Klassen. `CommandBuilder` sind providerspezifische Klassen, die auf Basis eines SELECT-Befehls passende SQL-DML-Befehle zur Laufzeit erzeugen. Die `CommandBuilder`-Klasse erwartet bei der Instanziierung ein `DataAdapter`-Objekt – mehr ist nicht zu tun für den Entwickler. Leider funktionieren die `CommandBuilder` nur in dem Fall, dass sich SELECT die Daten nur aus einer einzigen Tabelle/Abfrage holt und die Datenmenge einen Primärschlüssel besitzt. In allen anderen Fällen muss der Entwickler die Attribute `InsertCommand`, `UpdateCommand` und `DeleteCommand` in dem `DataAdapter`-Objekt selbst füllen.

TIPP: Batch-Größe für Datenadapter

Durch das in ADO.NET 2.0 neu eingeführte Attribut `UpdateBatchSize` kann die Anzahl der gleichzeitig zu übermittelnden Änderungen beliebig gesetzt werden, z. B.

```
da.UpdateBatchSize = 50;
```

Die Größe `0` bedeutet, dass alle Änderungen in einem Vorgang übermittelt werden. Die Übermittlung in einem Vorgang bei einer größeren Anzahl von Änderungen wird von Microsoft nicht empfohlen, da dies die Performanz negativ beeinflussen kann.

Beispiel

Das folgende Listing zeigt sehr kompakt viele Möglichkeiten der Datenänderung in einem Dataset am Beispiel der Tabelle *FL_Fluege* auf:

- Nach dem Einlesen der Daten werden zunächst in einer Schleife alle Zeilen gelöscht, bei denen in der Spalte *FL_Abflugort* der Wert *Essen/Mülheim* steht.
- In einer zweiten Schleife werden alle Startzähler (»`FL_AnzahlStarts`«) für die verbliebenen Flüge auf 0 gesetzt. Dabei ist zu beachten, dass diejenigen Zeilen ausgenommen werden müssen, die bereits gelöscht wurden. Sie erkennen dies an dem Wert `DataRowState.Deleted` in dem Attribut `RowState`.
- Im dritten Teil wird ein neuer Datensatz für den Flug 123 von Essen/Mülheim nach Eichstätt erzeugt.
- Danach wird ausgegeben, wie viele Datensätze geändert, gelöscht und hinzugefügt wurden.
- Zur Aktualisierung der Datenquelle wird der `SqlCommandBuilder` auf den Datenadapter angewendet. Bevor die Daten mit `Update()` zurückgeschrieben werden, werden zu Kontrollzwecken die generierten SQL-DML-Befehle ausgegeben. Die Batch-Größe wird dabei auf fünf festgelegt.

```
// === Daten schreiben mit einem DataSet
  public void DataSet_Schreiben()
  {
   Demo.PrintHeader("DataSet: Daten ändern");

   const string CONNSTRING = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=MARL\sqlexpress";
   const string SQL = "Select * from FL_Fluege";

   // --- Verbindung aufbauen
   SqlConnection conn = new SqlConnection(CONNSTRING);
   conn.Open();
   // --- Befehl ausführen
   SqlCommand cmd = new SqlCommand(SQL, conn);
   // --- Datenadapter erzeugen
   SqlDataAdapter da = new SqlDataAdapter(cmd);
   // --- DataSet erzeugen
   DataSet ds = new DataSet();
   // --- Daten abholen
   da.Fill(ds);
   // --- Verbindung jetzt schon schließen!
   conn.Close();

   DataTable dt = ds.Tables[0];

   // --- Datensätze löschen
   foreach (DataRow dr2 in dt.Rows)
   {
    if (dr2["FL_AbflugOrt"].ToString() == "Essen/Mülheim")
    {
     Demo.Print("Lösche: " + dr2["FL_FlugNr"]);
     dr2.Delete();
    }
   }
```

```
// --- Datensätze ändern
foreach (DataRow dr2 in dt.Rows)
{
 if (dr2.RowState != DataRowState.Deleted)
 {
  dr2["FL_AnzahlStarts"] = Convert.ToInt32(dr2["FL_AnzahlStarts"]) + 1;
  Demo.Print("Counter erhöht für: " + dr2["FL_FlugNr"]);
 }
}

// --- Datensätze anfügen
DataRow dr = dt.NewRow();
dr["FL_FlugNr"] = "123";
dr["FL_AnzahlStarts"] = 0;
dr["FL_EingerichtetAm"] = DateTime.Now;
dr["FL_Abflugort"] = "Essen/Mülheim";
dr["FL_Zielort"] = "Eichstätt";
dt.Rows.Add(dr);

// --- Statistik
if (ds.HasChanges(DataRowState.Added))
 Demo.Print("Anzahl der hinzugefügten Datensätze: " +
  dt.GetChanges(DataRowState.Added).Rows.Count);
if (ds.HasChanges(DataRowState.Modified))
 Demo.Print("Anzahl der geänderten Datensätze: " +
  dt.GetChanges(DataRowState.Modified).Rows.Count);
if (ds.HasChanges(DataRowState.Deleted))
 Demo.Print("Anzahl der gelöschten Datensätze: " +
  dt.GetChanges(DataRowState.Deleted).Rows.Count);

 // --- Befehle für Datenadapter erzeugen
SqlCommandBuilder cb = new SqlCommandBuilder(da);
// --- Kontrollausgabe
Demo.Print("Erzeugte SQL-DML-Befehle:");
Demo.Print("UPDATE: " + cb.GetUpdateCommand().CommandText);
Demo.Print("DELETE: " + cb.GetDeleteCommand().CommandText);
Demo.Print("INSERT: " + cb.GetInsertCommand().CommandText);

// --- Aktualisieren
da.Update(ds.Tables[0]);
Demo.Print("Daten wurden aktualisiert!");

}
```

***Listing 9.9**: Verschiedene Möglichkeiten zur Datenänderung in einem DataSet*

DataSet: Daten ändern

```
Lösche: 123
Counter erhöht für: 101
Counter erhöht für: 102
Counter erhöht für: 200
Counter erhöht für: 201
Counter erhöht für: 202
Counter erhöht für: 203
```

```
Anzahl der hinzugefügten Datensätze: 1
Anzahl der geänderten Datensätze: 6
Anzahl der gelöschten Datensätze: 1
Erzeugte SQL-DML-Befehle:
UPDATE: UPDATE [FL_Fluege] SET [FL_FlugNr] = @p1, [FL_Abflugort] = @p2, [FL_ZielOrt] = @p3, [FL_NichtRaucherFlug]
= @p4, [FL_Plaetze] = @p5, [FL_PI_MI_MitarbeiterNr] = @p6, [FL_AnzahlStarts] = @p7, [FL_EingerichtetAm] = @p8
WHERE (([FL_FlugNr] = @p9) AND ((@p10 = 1 AND [FL_Abflugort] IS NULL) OR ([FL_Abflugort] = @p11)) AND ((@p12 = 1
AND [FL_ZielOrt] IS NULL) OR ([FL_ZielOrt] = @p13)) AND ((@p14 = 1 AND [FL_NichtRaucherFlug] IS NULL) OR
([FL_NichtRaucherFlug] = @p15)) AND ((@p16 = 1 AND [FL_Plaetze] IS NULL) OR ([FL_Plaetze] = @p17)) AND ((@p18 = 1
AND [FL_PI_MI_MitarbeiterNr] IS NULL) OR ([FL_PI_MI_MitarbeiterNr] = @p19)) AND ((@p20 = 1 AND [FL_AnzahlStarts]
IS NULL) OR ([FL_AnzahlStarts] = @p21)) AND ((@p22 = 1 AND [FL_EingerichtetAm] IS NULL) OR ([FL_EingerichtetAm] =
@p23)))
DELETE: DELETE FROM [FL_Fluege] WHERE (([FL_FlugNr] = @p1) AND ((@p2 = 1 AND [FL_Abflugort] IS NULL) OR
([FL_Abflugort] = @p3)) AND ((@p4 = 1 AND [FL_ZielOrt] IS NULL) OR ([FL_ZielOrt] = @p5)) AND ((@p6 = 1 AND
[FL_NichtRaucherFlug] IS NULL) OR ([FL_NichtRaucherFlug] = @p7)) AND ((@p8 = 1 AND [FL_Plaetze] IS NULL) OR
([FL_Plaetze] = @p9)) AND ((@p10 = 1 AND [FL_PI_MI_MitarbeiterNr] IS NULL) OR ([FL_PI_MI_MitarbeiterNr] = @p11))
AND ((@p12 = 1 AND [FL_AnzahlStarts] IS NULL) OR ([FL_AnzahlStarts] = @p13)) AND ((@p14 = 1 AND
[FL_EingerichtetAm] IS NULL) OR ([FL_EingerichtetAm] = @p15)))
INSERT: INSERT INTO [FL_Fluege] ([FL_FlugNr], [FL_Abflugort], [FL_ZielOrt], [FL_NichtRaucherFlug], [FL_Plaetze],
[FL_PI_MI_MitarbeiterNr], [FL_AnzahlStarts], [FL_EingerichtetAm]) VALUES (@p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8)
Daten wurden aktualisiert!
```

***Listing 9.10**: Ausgabe des Beispiels*

Umwandlung zwischen DataSet und XML

Die `DataSet`-Klasse besitzt zwei Attribute und zwei Methoden zum Austausch mit XML:

- `GetXmlSchema()` liefert eine Zeichenkette mit der Struktur der Daten im `DataSet` in Form eines XSD-Schemas.
- `GetXml()` liefert eine Zeichenkette mit dem Inhalt des `DataSet`-Objekts in Form eines XML-Dokuments.
- `WriteXml()` schreibt die XML-Daten und – optional – das zugehörige XSD-Schema in eine Datei, ein `System.IO.Stream`-Objekt, ein `System.IO.TextWriter`-Objekt oder ein `System.IO.XmlWriter`-Objekt. Mit dem optionalen zweiten Parameter `XmlWriteMode.DiffGram` wird ein XML-Dokument im *DiffGram*-Format erzeugt. Ein Diffgram dokumentiert nicht nur den aktuellen Zustand eines `DataSet`, sondern auch alle ausgeführten Änderungen.
- `ReadXml()` liest XML-Daten in ein `DataSet`-Objekt ein. Mögliche Eingabequellen sind eine Datei (spezifiziert durch einen URL), ein `System.IO.Stream`-Objekt, ein `System.IO.TextReader`-Objekt oder ein `XmlReader`-Objekt. Die XML-Daten können ein Schema enthalten; das Schema kann aber auch abgeleitet werden, indem als Parameter `XmlReadMode.InferSchema` angegeben wird.

***Abbildung 9.9**: Darstellung eines DataSet inklusive Schema in XML-Form*

Neben den Import- und Exportmöglichkeiten des DataSet existiert noch eine Integration über das XML Document Object Model (DOM). Ein `XmlDataDocument`-Objekt ermöglicht die Bearbeitung des Inhalts eines `DataSet`-Objekts über das XML DOM. Die Klasse `System.Xml.XmlDataDocument` ist abgeleitet von der Klasse `System.Xml.XmlDocument`.

Umwandlung zwischen Dataset und Datareader

In ADO.NET 2.0 ist ein fliegender Wechsel zwischen Dataset und Datareader möglich. Für die Übernahme der Daten aus einem Datareader in ein `DataSet`-Objekt stellt die Klasse `DataSet` die neue Methode `Load()` bereit.

```
// Konvertierung DataReader->DataSet
DataSet ds = new DataSet();
ds.Load(reader, LoadOption.OverwriteRow, new String[] { "FL_Fluege" });
Demo.Out(ds.GetXml());
```

Für die Umwandlung eines Dataset in einen Datareader existiert die neue Klasse `DataTableReader`, die eine `IDataReader`-Schnittstelle implementiert.

```
// Konvertierung DataSet->DataReader
DataTableReader dtr = new DataTableReader(ds.Tables["FL_Fluege"]);
Demo.PrintReader(dtr);
```

Serialisierung und Remoting für Datasets

In ADO.NET 1.x hat die »geschwätzige« Serialisierung von Datasets im XML-Format Kritik hervorgerufen (in ADO.NET 1.x wurden Datasets immer in XML-Form serialisiert). ADO.NET 2.0 unterstützt daher optional die binäre Serialisierung. Dazu ist das neue Attribut `RemotingFormat` in einer Instanz der Klasse `DataSet` auf den Wert `SerializationFormat.Binary` zu setzen. Standard ist `SerializationFormat.Xml`.

```
public static void run()
{
 Demo.Out("=== DEMO Binäres Serialisieren eines DataSet")

 …
 SqlDataAdapter ada = new SqlDataAdapter(cmd);

 // --- DataSet erzeugen
 DataSet ds = new DataSet();
 ada.Fill(ds);

 // --- Serialisierungsformat festlegen
ds.RemotingFormat = SerializationFormat.Binary;
}
```

***Listing 9.11**: Aktivierung der binären Serialisierung für ein Dataset [DataSet_Binaer.cs]*

TIPP: In ADO.NET 1.x konnten nur komplette `DataSet`-Objekte, nicht jedoch einzelne `DataTable`-Objekte serialisiert werden. In ADO.NET 2.0 ist nun auch die Klasse `System.Data.DataTable` serialisierbar. Auch hier kann über die Eigenschaft `RemotingFormat` zwischen binärer und XML-Serialisierung gewählt werden.

Remoting von Datasets

Die ADO.NET-Datenadapter sind so »intelligent«, dass sie Änderungsbefehle zur Datenbank nur für die Zeilen eines Dataset senden, die geändert, hinzugefügt oder gelöscht wurden. Die unveränderten Zeilen verursachen kaum Overhead. Beim Einsatz von `DataSet`-Objekten bei Fernaufrufen (via .NET Remoting oder XML-Webservice) liegt der Datenadapter aber »entfernt«. Sie können viel Netzwerklast einsparen, wenn Sie nur die Änderungen zum Server übertragen. Durch den Befehl

```
DataSet ds_dif = ds.GetChanges();
```

reduzieren Sie das `DataSet`-Objekt auf die Änderungen. Nur `ds_dif` muss dann zum Server übertragen werden. Der Datenadapter kann dies handhaben.

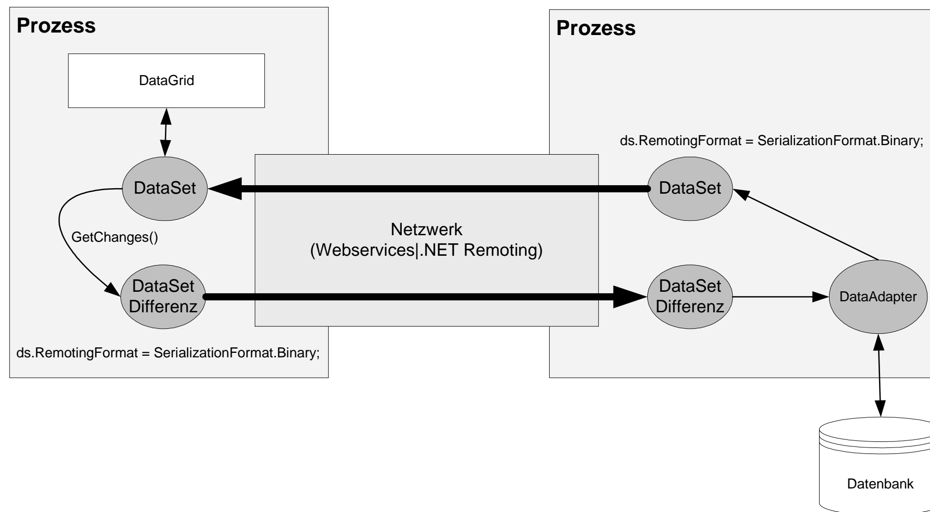

***Abbildung 9.10**: Effizientes Remoting von Datasets*

Befehlsausführung mit Command-Objekten

Sie können jegliche Form von SQL-Befehlen und Stored Procedures direkt auf der Datenbank aufrufen, indem Sie ein providerspezifisches `Command`-Objekt direkt nutzen. Neben dem bereits verwendeten `ExecuteReader()` stellen die `Command`-Klassen noch folgende Methoden bereit:

- `ExecuteNonQuery()` zur Ausführung von DML- und DDL-Befehlen, die keine Datenmenge zurückliefern. Sofern die Befehle die Anzahl der betroffenen Zeilen zurückliefern, steht diese Zahl im Rückgabewert der Methode. Sonst ist der Wert -1.
- `ExecuteRow()` liefert die erste Zeile der Ergebnismenge in Form eines `SqlRecord`-Objekts (nur SQL Server). Diese Funktion ist neu in .NET 2.0.
- `ExecuteScalar()` liefert nur die erste Spalte der ersten Zeile der Ergebnismenge.

Transaktionen

Befehle können auf einfache Weise in eine Transaktion verpackt werden. Der Beginn einer Transaktion ist mit `BeginTransaction()` zu markieren. `BeginTransaction()` liefert ein providerspezifisches `Transaction`-Objekt, das `Commit()` und `Rollback()` als Methoden anbietet. Dieses eigenständige `Transaction`-Objekt ist den einzelnen Befehlen explizit zuzuweisen (`sqlCmd.Transaction = t;`). Über das Attribut `IsolationLevel` kann die Art der Transaktion beeinflusst werden. Standardwert ist `ReadCommitted`.

Beispiel

Das Beispiel zeigt die Ausführung von zwei INSERT-Befehlen in einer Transaktion: Die Flugbuchung soll nur erfolgen, wenn der Passagier auf beiden Teilstrecken gebucht werden kann. Mit dem `Try-Catch`-Block wird dafür gesorgt, dass `Rollback()` aufgerufen wird, falls es bei der Ausführung eines der beiden Befehle zu einem Fehler kommt. Ob der Rollback tatsächlich funkti-

oniert, können Sie testen, indem Sie in einem der beiden SQL-Befehle ungültige Werte übergeben.

```
// === SQL-Befehle in einer Transaktion ausführen
  public void ADONET_Transaction_Demo()
  {
   const string CONNSTRING = @"…";
   const string SQL1 = @"INSERT INTO GF_GebuchteFluege ([GF_PS_ID], [GF_FL_FlugNr], [GF_FlugDatum], [GF_Preis],
[GF_Klasse]) " +
    "VALUES (1, 101, 8/1/2005, 500, 'F')";
   const string SQL2 = @"INSERT INTO GF_GebuchteFluege ([GF_PS_ID], [GF_FL_FlugNr], [GF_FlugDatum], [GF_Preis],
[GF_Klasse]) " +
    "VALUES (1, 203, 8/1/2005, 500, 'F')"; // Erfolgreicher Befehl
   const string SQL3 = "Select count(*) from GF_GebuchteFluege";
   // Verbindung aufbauen
   SqlConnection sqlConn = new SqlConnection(CONNSTRING);
   sqlConn.Open();

   // Befehlsobjekt erzeugen
   SqlCommand sqlCmd = sqlConn.CreateCommand();
   sqlCmd.CommandText = SQL3;
   Demo.Print("Anzahl Datensätze vorher: " + sqlCmd.ExecuteScalar().ToString());
   SqlTransaction t = null;
   try
   {
    t = sqlConn.BeginTransaction();
    sqlCmd.CommandText = SQL1;
    sqlCmd.Transaction = t;
    Demo.Print(SQL1);
    Demo.Print("Betroffene Zeilen: " + sqlCmd.ExecuteNonQuery());
    sqlCmd.CommandText = SQL2;
    sqlCmd.Transaction = t;
    Demo.Print(SQL2);
    Demo.Print("Betroffene Zeilen: " + sqlCmd.ExecuteNonQuery());
    t.Commit();
    Demo.Print("Transaktion erfolgreich ausgeführt!");
   }
   catch (Exception ex)
   {
    t.Rollback();
    Demo.Print("Transaktion fehlgeschlagen: " + ex.Message);
   }
   finally
   {
    sqlCmd.CommandText = SQL3;
    Demo.Print("Anzahl Datensätze nachher: " + sqlCmd.ExecuteScalar().ToString());
    sqlConn.Close();
   }
  }
```

***Listing 9.12**: Befehle in einer Transaktion ausführen*

Datenproviderunabhängiger Datenzugriff durch Providerfabriken

Stark vereinfacht wurde in ADO.NET 2.0 die Möglichkeit, unabhängig von einer konkreten Datenbank zu programmieren. Durch die neuen Basisklassen `DbProviderFactory`, `DbConnection`, `DbCommand`, `DbDataReader` sowie die bereits vorher vorhandene `DbDataAdapter`-Klasse kann man nun die Informationen zum Datenprovider in einer zur Laufzeit austauschbaren Zeichenkette halten. Die Klassen befinden sich im Namespace `System.Data.Common`.

TIPP: Durch die neue Funktion zur Ermittlung der installierten ADO.NET-Datenprovider (siehe Anfang dieses Kapitels) wird es möglich, dass eine Anwendung zur Laufzeit aus den verfügbaren Datenprovidern einen geeigneten auswählt.

Es findet aber keine Übersetzung von SQL-Befehlen statt. Wenn Sie Datenbankmanagementsystem-spezifische Befehle nutzen, verlieren Sie die Providerunabhängigkeit.

Beispiel

Das folgende Beispiel zeigt das Lesen von Daten mit einem Datareader und einem Dataset mit Hilfe des providerunabhängigen Programmiermodells. Dabei wird bei der Instanziierung der Klasse `DbProviderFactory` der Datenprovider (hier: `System.Data.SqlClient`) festgelegt. Durch die Instanz der Klasse `DbProviderFactory` können dann spezifische Verbindungsobjekte (`provider.CreateConnection()`), Befehlsobjekte (`provider.CreateCommand()`) und Datenadapter (`provider.CreateDataAdapter()`) erzeugt werden.

```
public static void run()
{
 Console.WriteLine("=== DEMO Provider Factory");
 const string PROVIDER = "System.Data.SqlClient";
 const string CONNSTRING = "Integrated Security=SSPI;Persist Security Info=False;Initial Catalog=itvisions;
                           Data Source=Wetter\SQLEXPRESS";
 const string SQL1 = "Select * from FL_Fluege";
 const string SQL2 = "Select * from FLB_Fluege_Backup";

 // --- Fabrik erzeugen
 DbProviderFactory provider = DbProviderFactories.GetFactory(PROVIDER);
 // --- Verbindung aufbauen
 DbConnection conn = provider.CreateConnection();
 conn.ConnectionString = CONNSTRING;
 conn.Open();

 // --- Teil 1: DataReader
 // Befehl erzeugen
 DbCommand cmd = provider.CreateCommand();
 cmd.CommandText = SQL1;
 cmd.Connection = conn;
 // Befehl ausführen
 DbDataReader reader = cmd.ExecuteReader();
 // Daten ausgeben
 Demo.PrintReader(reader);

 // --- Teil 2: DataSet
 // Befehl erzeugen
 DbCommand command = provider.CreateCommand();
```

```
  command.CommandText = SQL2;
  command.Connection = conn;
  // DataAdapter erzeugen
  DbDataAdapter adapter = provider.CreateDataAdapter();
  adapter.SelectCommand = command;
  // DataSet erzeugen
  DataSet ds = new DataSet();
  // DataSet befüllen
  adapter.Fill(ds);
  // Daten ausgeben
  DataTable t = ds.Tables[0];
  Console.WriteLine("Anzahl Spalten: " + t.Columns.Count);
  Console.WriteLine("Anzahl Zeilen: " + t.Rows.Count);
 }
```

***Listing 9.13**: Datenbankunabhängige Programmierung mit der DbProviderFactory [ProviderFactory.cs]*

Asynchrone Befehlsausführung

ADO.NET 2.0 erlaubt die asynchrone Ausführung von Datenbankbefehlen, der Aufrufer ist also während der Abarbeitung des Befehls nicht blockiert und kann weiterarbeiten. Dabei verwendet ADO.NET die im .NET Framework üblichen Entwurfsmuster (Patterns) für asynchrone Aufrufe mit der Schnittstelle `IAsyncResult`, wahlweise

- durch Polling auf das Attribut `IsCompleted` in der `IAsyncResult`-Schnittstelle,
- durch Warten mit einem `WaitHandle`-Objekt,
- mit einer Callback-Routine, die ein Objekt mit der Schnittstelle `IAsyncResult` empfängt.

In ADO.NET 2.0 stellen die `Command`-Klassen Methoden für das Starten und Beenden asynchroner Aufrufe bereit:

- `BeginExecuteNonQuery()` und `EndExecuteNonQuery()`
- `BeginExecuteReader()` und `EndExecuteReader()`
- `BeginExecuteXmlReader()` und `EndExecuteXmlReader()`

In der Beta-Version verfügt nur die Klasse `SqlCommand` über diese Methoden. Laut der ADO.NET 2.0 Feature-Liste [MSDN09] soll der asynchrone Aufruf später auch für andere Datenprovider zur Verfügung stehen.

WICHTIG: Wichtig ist, dass die Absicht, asynchrone Aufrufe auszuführen, schon in der Verbindungszeichenfolge mit `Asynchronous Processing=true` deklariert wird. In der Beta-Version funktioniert alternativ dazu auch die Kurzform `Async=true`. Ob dies in der Endfassung so bleiben wird, ist noch unklar.

Die asynchrone Befehlsausführung wird nur in NT-basierten Windows-Versionen unterstützt.

Die ursprünglich angekündigte Funktion des asynchronen Verbindungsaufbaus ist zunächst aus dem Funktionsumfang von .NET 2.0 entfernt worden.

TIPP: Man sollte eine Verbindung nur dann als asynchron öffnen, wenn sie tatsächlich für asynchrone Operationen verwendet werden soll. Die Ausführung synchroner Befehle über

eine für asynchrone Befehle geöffnete Verbindung beeinträchtigt die Performanz der synchronen Befehle.

Der Aufruf einer der *End*-Methoden vor dem Ende der Operation bewirkt, dass die Anwendung auf die Fertigstellung des Befehls wartet, also so lange blockiert.

Die `Command`-Klassen stellen auch eine Methode bereit, mit der ein Entwickler eine asynchrone Ausführung abbrechen kann: `sqlCmd.Cancel()`. Allerdings ist diese Methode in der Beta 1-Version zwar vorhanden, liefert aber nur ein »not supported« zurück.

Beispiel für das Polling-Modell

Beim Polling empfängt man ein Objekt mit der `IAsyncResult`-Schnittstelle von der Methode `BeginExecuteReader()`.

```
  // Asynchrone Befehlsausführung via Polling
  public static void run_Polling()
  {
   Demo.Out("=== DEMO Asynchrone Ausführung - Polling");
   const string CONNSTRING = "Integrated Security=SSPI;Persist Security Info=False;Asynchronous
Processing=true;Initial Catalog=itvisions;Data Source=Wetter\SQLEXPRESS";
   const string SQL = "Select * from FL_Fluege";
   // Verbindung aufbauen
   SqlConnection sqlConn = new SqlConnection(CONNSTRING);
   sqlConn.Open();
   // Befehl definieren
   SqlCommand sqlCmd = sqlConn.CreateCommand();
   sqlCmd.CommandText = SQL;
   // Befehl starten
   IAsyncResult result = sqlCmd.BeginExecuteReader(CommandBehavior.CloseConnection);
   // Warten...
   while (!result.IsCompleted) { Demo.Out("Warte..."); }
   // Ergebnis auswerten
   SqlDataReader reader = sqlCmd.EndExecuteReader(result);
   Demo.PrintReader(reader);
  }
```

***Listing 9.14**: Asynchrone Ausführung einer SELECT-Anweisung mit dem Polling-Modell [MSSQL_MARS.cs]*

Beispiel für das Warte-Modell

Das Warte-Modell ist dem Polling-Modell ähnlich. Allerdings wird hier das Warten durch ein `WaitHandle`-Objekt realisiert.

```
// Asynchrone Befehlsausführung via WaitHandle
  public static void run_Warten()
  {
   Demo.Out("=== DEMO Asynchrone Ausführung - Warte-Modell");

   const string CONNSTRING = "Integrated Security=SSPI;Persist Security Info=False;Asynchronous
Processing=true;Initial Catalog=itvisions;Data Source=Wetter\SQLEXPRESS";
   const string SQL = "Select * from FL_Fluege";

   // Verbindung aufbauen
   SqlConnection sqlConn = new SqlConnection(CONNSTRING);
   sqlConn.Open();
```

```
  // Befehl definieren
  SqlCommand sqlCmd = sqlConn.CreateCommand();
  sqlCmd.CommandText = SQL;
  // Befehl starten
  IAsyncResult result = sqlCmd.BeginExecuteReader(CommandBehavior.CloseConnection);
  // Warten...
  result.AsyncWaitHandle.WaitOne();
  // Ergebnis auswerten
  SqlDataReader reader = sqlCmd.EndExecuteReader(result);
  Demo.PrintReader(reader);
 }
```

***Listing 9.15**: Asynchrone Ausführung einer SELECT-Anweisung mit dem Warte-Modell [MSSQL_MARS.cs]*

Beispiel für das Callback-Modell

Beim Callback-Modell ist ein `AsyncCallback`-Objekt zu erzeugen, das als Parameter an `BeginExecuteReader()` zu übergeben ist. Die `CallbackHandler()`-Funktion, die nach Beendigung des Befehls aufgerufen wird, empfängt dann die `IAsyncResult`-Schnittstelle.

```
public class AsyncCommand
 {
  public static void run()
  {
   Demo.Out("=== DEMO Asynchrone Ausführung - Callback-Modell");
   const string CONNSTRING = "Integrated Security=SSPI;Persist Security Info=False;Asynchronous
Processing=true;Initial Catalog=itvisions;Data Source=Wetter\SQLEXPRESS";
   const string SQL = "Select * from FL_Fluege";
   SqlConnection sqlConn = new SqlConnection(CONNSTRING);
   sqlConn.Open();
   SqlCommand sqlCmd = sqlConn.CreateCommand();
   sqlCmd.CommandText = SQL;
   AsyncCallback callback = new AsyncCallback(CallbackHandler);
   sqlCmd.BeginExecuteReader(callback, sqlCmd, CommandBehavior.CloseConnection);
  }
  private static void CallbackHandler(IAsyncResult result)
  {
   Demo.Out("Callback von asynchronem Reader-Aufruf...");
   SqlCommand command = (SqlCommand)result.AsyncState;
   SqlDataReader reader = command.EndExecuteReader(result);
   Demo.Out("Hat der Befehl Zeilen geliefert? " + reader.HasRows);
// Ausgabe der Ergebnisse
   while (reader.Read())
    for (int i = 0; i < reader.FieldCount; i++)
     Demo.Out("Column: " + reader.GetName(i) + "  Value: " + reader.GetValue(i));
  }
```

***Listing 9.16**: Asynchrone Ausführung einer SELECT-Anweisung mit dem Callback-Modell [MSSQL_MARS.cs]*

Benachrichtigungen über Datenänderungen

Eine sehr interessante Option in ADO.NET 2.0 sind Benachrichtigungen über Datenänderungen (Query Notifications). Dabei meldet der Client gegenüber dem Datenbankmanagementsys-

tem sein Interesse an einer bestimmten Datenmenge an. Jedes Mal, wenn eine Veränderung in dieser Datenmenge eintritt, wird eine Ereignisbehandlungsroutine aufgerufen, sodass der Client weiß, dass er die Daten erneut abrufen sollte. Durch Notifications entfällt das ständige Polling der Datenbank.

HINWEIS: In der Beta-Version sind Notifications nur für den SQL Server 2005 vorgesehen. Der SQL Server 2005 hat besitzt eine echte Notification-Architektur auf Basis des integrierten Service Brokers. Die Netzwerklast ist minimal, weil es wirklich nur für den Fall von Datenänderungen zu Benachrichtigungen kommt. Geplant ist, eine ähnliche Funktionalität (allerdings auf Basis von internem Polling) auch für die Vorgängermodelle 7.0 und 2000 anzubieten. Dabei ist eine kleine konstante Belastung des Netzwerks durch das Polling zu erwarten.

Realisierung

ADO.NET 2.0 stellt für Notifications die Klassen `SqlNotificationRequest` und `SqlDependency` bereit, wobei die letztgenannte Klasse eine höhere Abstraktion und damit mehr Entwicklungskomfort bietet. Die `SqlDependency`-Klasse ist zu instanziieren mit einem `SqlCommand`-Objekt, da es einen SQL-SELECT-Befehl mit der zu überwachenden Datenmenge repräsentiert.

Leider unterliegt der SELECT-Befehl starken Einschränkungen. Die wichtigsten Einschränkungen auf einer langen Liste sind, dass er folgende Konstrukte nicht enthalten darf:

- Stern-Operator zur Spaltenauswahl
- Aggregatfunktionen COUNT, AVG, MAX, MIN
- Schlüsselwörter UNION, TOP, INTO, FOR BROWSE
- Outer Joins

Außerdem muss der Tabellenname mit dem führenden »dbo.« genannt werden. Die komplette Liste der Voraussetzungen finden Sie als Kommentar in der Codedatei des nachfolgenden Beispiels.

Datenänderungen werden dem Client durch das Ereignis `OnChange()` gemeldet. Das dabei übermittelte Objekt vom Typ `SqlNotificationEventArgs` liefert Informationen über die Art der Datenänderung (Hinzufügen, Löschen, Ändern), nicht aber über die geänderten Zeilen.

ACHTUNG: Wenn Sie ein Ereignis mit dem Parameter `SqlNotificationEventArgs.Type` mit Wert `SqlNotificationType.Subscribe` erhalten, ist ein Fehler beim Einrichten der Notification aufgetreten. In der Regel haben Sie dann eine der Bedingungen für den SELECT-Befehl nicht beachtet.

Beispiel

In dem folgenden Beispiel wartet eine Anwendung nach erfolgreichem, einmaligem Auslesen der Datenmenge auf Benachrichtigungen über Datenänderungen in der Datenquelle. Das `SqlDependency`-Objekt bezieht sich auf das `SqlCommand`-Objekt mit dem Befehl »Select FL_FlugNr, FL_Zielort, Fl_Plaetze from dbo.FL_Fluege where FL_Abflugort = 'Frankfurt'«. Der SQL Server wird ADO.NET unterrichten, sobald sich Daten in dieser Tabelle ändern. Im Programmcode wird dann die Ereignisbehandlungsroutine `dep_OnChanged()` aufgerufen.

Die Benachrichtigungen sind nicht an eine spezielle Form des Datenlesens gebunden; diese können daher auch mit einem Dataset verwendet werden.

```
public class Notfications_Demos
 {
  private static SqlDependency dep = null;

  static bool DataChanged = false;

  // === Warten auf Datenänderungen in der Flug-Tabelle
  public void run()
  {
   Demo.PrintHeader("Überwachung der Flugliste [Query Notifications]");

   string CONNSTRING2 = @"Integrated Security=SSPI;Database=WorldWideWings;Persist Security Info=True;Data
Source=Marl\sqlexpress";
   const string SQL = "Select FL_FlugNr, FL_Zielort, Fl_Plaetze from dbo.FL_Fluege where FL_Abflugort =
'Frankfurt'";

   // Verbindung aufbauen
   SqlConnection conn = new SqlConnection(CONNSTRING2);
   conn.Open();

   while (true)
   {
    DataChanged = false;
    // Befehl definieren
    SqlCommand cmd = new SqlCommand(SQL, conn);

    // Überwachung anlegen
    dep = new SqlDependency(cmd);

    // Ereignisbehandlung binden
    dep.OnChange += new OnChangeEventHandler(dep_OnChange);



    while (!DataChanged)
    { System.Threading.Thread.Sleep(1000); Console.Write("."); }
   }
  }

  // Ereignisbehandlung für geänderte Daten
  static void dep_OnChange(object sender, SqlNotificationEventArgs e)
  {
   Console.BackgroundColor = ConsoleColor.Cyan;
   Demo.Print("\nDaten haben sich geändert!");
   Console.BackgroundColor = ConsoleColor.Black;
   Demo.Print("Typ: " + e.Type.ToString());
   Demo.Print("Quelle: " + e.Source.ToString());
   Demo.Print("Info: " + e.Info.ToString());
   DataChanged = true;
  }
 }
```

***Listing 9.17**: Warten auf Benachrichtigung über Datenänderungen [MSSQL_Notifikationen.cs]*

Massenkopieren (Bulkcopy/Bulkimport)

Wenn eine große Datenmenge von einer Datenquelle zu einer anderen bewegt werden soll, ist es unzweckmäßig, die Daten zeilenweise zu übertragen. Der Microsoft SQL Server besitzt für den Massendatenimport das Werkzeug *bcp.exe*. Eine ähnliche Funktionalität ist jetzt auch innerhalb von ADO.NET 2.0 verfügbar durch die Klasse `SqlBulkCopy`.

HINWEIS: Die Massenkopierfunktion ist auch als **Bulkcopy** oder **Bulkimport** bekannt. In der Beta-Version ist diese Funktion nur für den Microsoft SQL Server verfügbar. Laut der ADO.NET Feature-Matrix [MSDN09] ist jedoch noch eine Implementierung für andere Datenprovider zu erwarten.

Während das Ziel ein Microsoft SQL Server sein muss, ist `SqlBulkCopy` hinsichtlich der Eingabedaten flexibel und akzeptiert folgende Eingabedatenformen:

- `DataTable`
- ein Array von `DataRow`-Objekten
- einen Datareader (Objekt, das `IDataReader` implementiert).

Dabei ist die Datenherkunft (z. B. Datenbank, Datei) beliebig.

Vorgehensweise

Bei der Instanziierung der Klasse `SqlBulkCopy` ist die Verbindung zum Ziel anzugeben. Danach muss der Entwickler die Zieltabelle festlegen. Die Operation wird mit `WriteToServer()` gestartet.

Wenn sich die Tabellenstrukturen unterscheiden, versucht `SqlBulkCopy` eine Abbildung anhand der Position der Spalten. Eine Abbildung kann explizit durch die `ColumnMappings`-Collection definiert werden, z. B.

```
BulkCopy.ColumnMappings.Add("FL_Abflugort", "FL_Zielort");
BulkCopy.ColumnMappings.Add("FL_Zielort", "FL_Abflugort");
```

HINWEIS: Zumindest in der vorliegenden Beta-Version unterscheidet die `ColumnMappings`-Klasse zwischen Groß- und Kleinschreibung.

Das Ereignis `SqlRowsCopied()` informiert den Aufrufer nach jeweils n kopierten Zeilen, um ihm eine Fortschrittsanzeige zu ermöglichen. Die Zahl *n* wird durch das Attribut `NotifyAfter` festgelegt.

Beispiel 1

Im ersten Beispiel wird eine Sicherungskopie der Tabelle *FL_Fluege* innerhalb des SQL Servers erstellt, also eine Microsoft SQL Server-Datenmenge in eine andere Microsoft SQL Server-Tabelle (*FLB_Fluege_Backup*) kopiert – ohne explizite Festlegung der Abbildung und ohne Ereignisbehandlung.

```
public void run_SQLtoSQL_Einfach()
  {

   Demo.PrintHeader("Bulk Copy - SQL Server->SQL Server - Einfach");
   const string CS_Quelle = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=Marl\sqlexpress";
   const string CS_Ziel = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=Marl\sqlexpress";
```

```
    const string SQL_Quelle = "select * from FL_Fluege";
    const string ZIELTABELLE = "FLB_Fluege_Backup";


    // Verbindung zur Quelle
    Demo.Print("Verbindung zur Quelle öffnen...");
    SqlConnection C_Quelle = new SqlConnection(CS_Quelle);
    C_Quelle.Open();
    // Verbindung zum Ziel
    Demo.Print("Verbindung zum Ziel öffnen...");
    SqlConnection C_Ziel = new SqlConnection(CS_Ziel);
    C_Ziel.Open();

    // Daten holen
    Demo.Print("Daten aus Quelle einlesen...");
    SqlCommand CMD_Quelle = new SqlCommand(SQL_Quelle, C_Quelle);
    SqlDataReader Reader = CMD_Quelle.ExecuteReader();

    // Kopiervorgang
    Demo.Print("Daten in Zieltabelle schreiben...");
    SqlBulkCopy BulkCopy = new SqlBulkCopy(C_Ziel);
    BulkCopy.DestinationTableName = ZIELTABELLE;
    BulkCopy.WriteToServer(Reader);

    // Ende
    Reader.Close();
    C_Quelle.Close();
    C_Ziel.Close();
  }
```

***Listing 9.18**: Einfaches Beispiel für eine Massenkopie zwischen zwei Microsoft SQL Servern [BulkImport.cs]*

Beispiel 2

Im zweiten Beispiel wird die Tabelle *FL_Fluege* aus einer Microsoft Access-Datenbank *WorldWideWings.mdb* in die SQL Server-Tabelle *FLB_Fluege_Backup* kopiert – ohne explizite Festlegung der Abbildung und ohne Ereignisbehandlung.

```
// Massenkopie von Access-Datenbank in SQL Server-Datenbank
  public void run_AccessToSQL()
  {

    Demo.PrintHeader("Bulk Copy - Access->SQL Server - Einfach");
    const string CS_Quelle = @"Provider='Microsoft.Jet.OLE DB.4.0';Data
Source='E:\N2C\N2C_ersteAnwendungen\N2C_JS_WebAnwendung\App_Data\WorldWideWings.mdb'";
    const string CS_Ziel = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=Marl\sqlexpress";
    const string SQL_Quelle = "select * from FL_Fluege";
    const string ZIELTABELLE = "FLB_Fluege_Backup";

    // Verbindung zur Quelle
    Demo.Print("Verbindung zur Quelle öffnen...");
    OleDbConnection C_Quelle = new OleDbConnection(CS_Quelle);
    C_Quelle.Open();
    // Verbindung zum Ziel
```

```
    Demo.Print("Verbindung zum Ziel öffnen...");
    SqlConnection C_Ziel = new SqlConnection(CS_Ziel);
    C_Ziel.Open();

    // Daten holen
    Demo.Print("Daten aus Quelle einlesen...");
    OleDbCommand CMD_Quelle = new OleDbCommand(SQL_Quelle, C_Quelle);
    OleDbDataReader Reader = CMD_Quelle.ExecuteReader();

    // Kopiervorgang
    Demo.Print("Daten in Zieltabelle schreiben...");
    SqlBulkCopy BulkCopy = new SqlBulkCopy(C_Ziel);
    BulkCopy.DestinationTableName = ZIELTABELLE;
    BulkCopy.WriteToServer(Reader);

    // Ende
    Reader.Close();
    C_Quelle.Close();
    C_Ziel.Close();
   }
```

***Listing 9.19**: Beispiel für eine Massenkopie zwischen einer Access-Datenbank und einem Microsoft SQL Server [BulkImport.cs]*

Beispiel 3

Im dritten Beispiel wird eine Datenmenge aus einer Microsoft SQL Server-Datenbank in eine andere SQL Server-Tabelle kopiert, wobei die Spaltenabbildung explizit definiert (Vertauschen der Spalten *FL_Abflugort* und *FL_Zielort*) und eine Ereignisbehandlung für das `SqlRowsCopied()`-Ereignis implementiert werden. In der Ereignisbehandlungsroutine wird für jede Zeile eine Ausgabe an der Konsole erzeugt.

```
// Massenkopie zwischen SQL Server-Datenbanken mit Optionen
  public void SQLToSQL_Advanced()
  {

    Demo.PrintHeader("Bulk Copy - SQL Server->SQL Server - Erweitert");

    const string CS_Quelle = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=Marl\sqlexpress";
    const string CS_Ziel = @"Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=Marl\sqlexpress";
    const string SQL_Quelle = "select * from FL_Fluege";
    const string ZIELTABELLE = "FLB_Fluege_Backup";

    // Verbindung zur Quelle
    Demo.Print("Verbindung zur Quelle öffnen...");
    SqlConnection C_Quelle = new SqlConnection(CS_Quelle);
    C_Quelle.Open();
    // Verbindung zum Ziel
    Demo.Print("Verbindung zum Ziel öffnen...");
    SqlConnection C_Ziel = new SqlConnection(CS_Ziel);
    C_Ziel.Open();

    // Daten holen
    Demo.Print("Daten aus Quelle einlesen...");
```

```
    SqlCommand CMD_Quelle = new SqlCommand(SQL_Quelle, C_Quelle);
    SqlDataReader Reader = CMD_Quelle.ExecuteReader();

    // Kopiervorgang
    Demo.Print("Daten in Zieltabelle schreiben...");
    SqlBulkCopy BulkCopy = new SqlBulkCopy(C_Ziel);
    BulkCopy.DestinationTableName = ZIELTABELLE;

    BulkCopy.ColumnMappings.Add("FL_Abflugort", "FL_ZielOrt");
    BulkCopy.ColumnMappings.Add("FL_ZielOrt", "FL_Abflugort");
    BulkCopy.NotifyAfter = 2;
    BulkCopy.SqlRowsCopied += new SqlRowsCopiedEventHandler(BulkCopy_SqlRowsCopied);
    BulkCopy.WriteToServer(Reader);

    // Ende
    Reader.Close();
    C_Quelle.Close();
    C_Ziel.Close();
   }

  // Ereignisbehandlung
  static void BulkCopy_SqlRowsCopied(object sender, SqlRowsCopiedEventArgs e)
  {
   Demo.Print("Zeile kopiert: " + e.RowsCopied);
  }
```

***Listing 9.20**: Beispiel für eine Massenkopie mit Fortschrittsanzeige zwischen zwei Microsoft SQL Servern [BulkImport.cs]*

Weitere neue Funktionen

Dieses Kapitel stellt einige weitere Funktionen zusammen, die in ADO.NET 2.0 ergänzt wurden.

Verbesserungen beim Verbindungs-Pooling

ADO.NET-Datenprovider implementieren (auch schon in ADO.NET 1.x) eigene Verbindungs-Pool-Mechanismen (ADO.NET Connection Pooling). Der Entwickler muss sich daher nicht um die optimale Dauer einer Datenbankverbindung kümmern; er sollte die Verbindung immer so schnell wie möglich schließen.

In ADO.NET 1.x hatte der Entwickler aber auch keinen Einfluss darauf, wann die Verbindung tatsächlich abgebaut wird. Die Datenprovider für Microsoft SQL Server und Oracle ermöglichen es dem Entwickler in ADO.NET 2.0 nun, den Zeitpunkt zum Leeren der Verbindungs-Pools selbst zu bestimmen.

- Der Befehl `SqlConnection.ClearPool(Conn)` entfernt die angegebene Verbindung aus dem Verbindungs-Pool.
- `SqlConnection.ClearAllPools()` löscht alle Verbindungs-Pools.

Provider-Statistiken

ADO.NET 2.0 unterstützt für Datenprovider eine Statistik-Funktion über alle Aktivitäten innerhalb einer Datenbankverbindung, z. B. Anzahl der ausgeführten Befehle, Anzahl der übermittelten Datensätze, Anzahl der übermittelten Bytes. Diese statistischen Informationen können jederzeit von einem Verbindungsobjekt abgerufen werden.

Folgende Hinweise sind jedoch zu beachten:

- Die Statistik muss durch `StatisticsEnabled = True` für ein Verbindungsobjekt aktiviert werden.
- Die Methode `RetrieveStatistics()` liefert ein Objekt mit einer `IDictionary`-Schnittstelle mit Attribut-Wert-Paaren.
- Die Werte besitzen alle den Datentyp `System.Int64`.

TIPP: Alle statistischen Zähler können jederzeit mit `ResetStatistics()` auf den Wert 0 zurückgesetzt werden.

Beispiel

In dem folgenden Beispiel werden zwei Datenmengen per `SqlDataReader` durchlaufen. Danach wird die Provider-Statistik ausgegeben.

```
  public static void run()
  {
   Demo.Out("=== DEMO ProviderStatistik");

   const string CONNSTRING = "Integrated Security=SSPI;Persist Security Info=False;Asynchronous
Processing=true;Initial Catalog=itvisions;Data Source=Wetter\SQLEXPRESS";
   const string SQL = "Select * from FL_Fluege";
   const string SQL2 = "Select * from AllePassagiere";

   SqlConnection sqlConn = new SqlConnection(CONNSTRING);
   // Statistik aktivieren
   sqlConn.StatisticsEnabled = true;
   // Verbindung aufbauen
   sqlConn.Open();
   // Befehl ausführen
   SqlCommand sqlCmd = sqlConn.CreateCommand();
   sqlCmd.CommandText = SQL;
   Demo.ReadWithoutPrinting(sqlCmd.ExecuteReader());
   // Noch einen Befehl ausführen
   SqlCommand sqlCmd2 = sqlConn.CreateCommand();
   sqlCmd2.CommandText = SQL2;
   Demo.ReadWithoutPrinting(sqlCmd2.ExecuteReader());


   // --- Hole Statistik
   System.Collections.IDictionary Stat = sqlConn.RetrieveStatistics();
   // --- Schleife über alle Einträge‘
   foreach (System.Collections.DictionaryEntry de in Stat)
   {
    Demo.Out(de.Key + " = " + de.Value);
   }
```

***Listing 9.21**: Ausgabe der Nutzungsstatistik [Statistik.cs]*

```
file:///H:/Code/DOTNET2B1/NET2B1_CS_MISC/bin/Debug/NET2B1_CS_MISC.exe
=== DEMO ProviderStatistik
NetworkServerTime = 0
BytesReceived = 488
UnpreparedExecs = 2
SumResultSets = 2
SelectCount = 2
PreparedExecs = 0
ConnectionTime = 10
ExecutionTime = 100
Prepares = 0
BuffersSent = 2
SelectRows = 6
ServerRoundtrips = 2
CursorOpens = 0
Transactions = 0
BytesSent = 148
BuffersReceived = 2
IduRows = 0
IduCount = 0
===================== ENDE DER DEMO ===================
_
```

***Abbildung 9.11**: Beispiel für eine Statistik-Ausgabe nach der Ausführung von zwei SELECT-Befehlen*

Verbindungszeichenfolgen zusammensetzen mit dem ConnectionStringBuilder

ADO.NET 2.0 bietet Hilfsklassen zum Zusammensetzen von Verbindungszeichenfolgen. Diese Hilfsklassen nehmen Parameterwerte entgegen und liefern als Ausgabe eine Verbindungszeichenfolge als Zeichenkette.

Während die allgemeine Klasse `System.Data.Common.DbConnectionStringBuilder` mit untypisierten Attribut-Wert-Paaren arbeitet, besitzen die davon abgeleiteten Klassen

- `System.Data.Odbc.OdbcConnectionStringBuilder`
- `System.Data.OleDb.OleDbConnectionStringBuilder`
- `System.Data.OracleClient.OracleConnectionStringBuilder`
- `System.Data.SqlClient.SqlConnectionStringBuilder`

jeweils datenproviderspezifische Attribute, die die Verwendung vereinfachen.

Beispiel

Das Code-Fragment zeigt das Zusammensetzen einer Verbindungszeichenfolge für einen Microsoft SQL Server.

```
public static void run()
{
 System.Data.SqlClient.SqlConnectionStringBuilder csb = new SqlConnectionStringBuilder();
 // --- Eingabedaten
 csb.UserID = "HS";
 csb.Password = "geheim";
 csb.DataSource = "Essen";
 csb.InitialCatalog = "Demo";
 csb.PersistSecurityInfo = true;
 csb.IntegratedSecurity = false;
```

```
  // --- Zusammengesetzte Verbindungszeichenfolge
  Demo.Out(csb.ConnectionString);
 }
```

***Listing 9.22**: Zusammensetzen einer Verbindungszeichenfolge [ConnStringBuilder.cs]*

Verbindungszeichenfolgen aus der Konfigurationsdatei auslesen

Die XML-Anwendungskonfigurationsdateien bieten ab .NET 2.0 eine separate Sektion für die Ablage von (verschlüsselten) Verbindungszeichenfolgen. Dieses Thema wurde bereits in Kapitel 11 unter dem Stichwort »System.Configuration« behandelt.

Ermittlung der verfügbaren Microsoft SQL Server

Mit Hilfe der Klasse `SqlDataSourceEnumerator` kann man ab ADO.NET 2.0 die in der Windows-Domäne verfügbaren Microsoft SQL Server auflisten. Die Methode `GetDataSources()` liefert ein `DataTable`-Objekt mit folgenden Feldern:

- `ServerName`
- `InstanceName`
- `IsClustered`
- `Version`

HINWEIS: In der aktuellen Beta-Version sind die drei letztgenannten Attribute leider immer leer.

Beispiel

Die folgende Methode gibt alle erreichbaren SQL Server-Installationen aus. Die Methode `PrintTable()`, die hier aus Platzgründen nicht abgedruckt ist, gibt alle Zeilen und Spalten eines `DataTable`-Objekts aus.

```
  // Auflistung aller MS SQL-Server in der Domäne
 [Test]
 public static void GetAllSQLServers2()
 {
  DataTable servers = SqlDataSourceEnumerator.Instance.GetDataSources();
  foreach (DataRow src in servers.Rows)
   PrintTable(servers);
 }
```

***Listing 9.23**: Ausgabe der erreichbaren SQL Server-Installationen [Enumerationen.cs]*

Datenbankbenutzerkennwörter ändern

Der Microsoft SQL Server unterstützt neben der Windows-integrierten Authentifizierung auch eine eigene Benutzerdatenbank. Nach einer Standardinstallation existiert dort nur das Administratorkonto »sa«. Um die Kennwörter für die SQL-Benutzerkonten zu ändern, konnte man bisher nur den Transact SQL-Befehl `ALTER LOGIN` verwenden.

ADO.NET 2.0 bietet im Zusammenhang mit dem SQL Server 2005 die Möglichkeit, ein Kennwort auch eleganter über die statische Methode `ChangePassword()` in der Klasse `SqlConnection` zu ändern.

Beispiel

In dem folgenden Beispiel wird das Kennwort für den Benutzer »sa« geändert.

```
// Eingabedaten für Demo
const string CONNSTRING = "Server=essen;User ID=sa;Password=test123$123;Database=Test;Persist Security
                           Info=True";
// Kennwort ändern
SqlConnection.ChangePassword(CONNSTRING, "demo123$123");
// Ausgabe
Demo.Out("Kennwort für das Benutzerkonto 'sa' wurde geändert!"
```

***Listing 9.24**: Kennwortänderung für das sa-Benutzerkonto [MSSQL_KennwortAenderung.cs]*

Schema-API

Das neue Schema-API von ADO.NET 2.0 besteht aus einem einzigen Befehl. `GetSchema()` ruft Schema-Informationen in Form eines `DataTable`-Objekts von einer Datenbank ab. `GetSchema()` erwartet eine Zeichenkette, die die Menge der zu übermittelnden Informationen angibt. Ein zweiter Parameter in Form eines Zeichenketten-Array erlaubt die Angabe eines Filters.

Dabei gibt es fünf allgemeine Auflistungen, die durch die Konstantenliste `System.Data.Common.DbMetaDataCollectionNames` festgelegt sind:

- `MetaDataCollections`: eine Liste der verfügbaren Mengen (z. B. »Tables«, »Views«, »Users«, etc.)
- `Restrictions`: Eine Liste der verfügbaren Filter
- `DataSourceInformation`: Informationen zur Datenbankinstanz, auf die der Datenprovider verweist
- `DataTypes`: Informationen über von der Datenbank unterstützten Datentypen
- `ReservedWords`: Liste aller reservierten Wörter der Datenbanksprache

```
SqlConnection sqlConn = new SqlConnection(CONNSTRING);
   sqlConn.Open();
   Demo.PrintHeader("MetaDataCollections:");
    dt = sqlConn.GetSchema(System.Data.Common.DbMetaDataCollectionNames.MetaDataCollections);
   Demo.PrintTable(dt);
   Demo.PrintHeader("Liste der Tabellen:");
   DataTable dt = sqlConn.GetSchema("Tables");
   Demo.PrintTable(dt);
   Demo.PrintHeader("Liste der Views:");
    dt = sqlConn.GetSchema("Tables");
   Demo.PrintTable(dt);
   Demo.PrintHeader("Unterstützte Datentypen:");
    dt = sqlConn.GetSchema(System.Data.Common.DbMetaDataCollectionNames.DataTypes);
   Demo.PrintTable(dt);
   Demo.PrintHeader("Unterstützte Einschränkungen:");
    dt = sqlConn.GetSchema(System.Data.Common.DbMetaDataCollectionNames.Restrictions);
   Demo.PrintTable(dt);
   Demo.PrintHeader("Informationen über die Datenbank:");
```

```
  dt = sqlConn.GetSchema(System.Data.Common.DbMetaDataCollectionNames.DataSourceInformation);
  Demo.PrintTable(dt);
  Demo.PrintHeader("Reservierte Wörter:");
  dt = sqlConn.GetSchema(System.Data.Common.DbMetaDataCollectionNames.ReservedWords);
 Demo.PrintTable(dt);
```

***Listing 9.25**: Nutzung des ADO.NET 2.0 Schema API*

Einschränkungen angeben

Die Nutzung der Filter ist wenig komfortabel, entspricht aber der Anforderung, eine universelle Programmierschnittstelle für die unterschiedlichen Datenbankmanagementsysteme zu schaffen. `DbMetaDataCollectionNames.Restrictions` liefert für den Metadatentyp `Columns` vier Filter: `Catalog`, `Owner`, `Table` und `Column`. Dies bedeutet, dass im zweiten Parameter bei `GetSchema()` vier Werte anzugeben sind. Filter, die nicht gesetzt werden sollen, sind mit Null zu belegen. Das folgende Beispiel zeigt, wie Sie eine Liste der Spalten der Tabelle *FL_Fluege* erhalten.

```
Demo.PrintHeader("Liste der Spalten der Tabelle FL_Fluege:");
  string[] e = new String[] { null,null,"Fl_Fluege",null };
  dt = sqlConn.GetSchema("Columns",  e);
  Demo.PrintTable(dt);
```

***Listing 9.26**: Eingeschränkte Suche*

ADO.NET 2.0 Feature-Matrix

Die folgende Tabelle stellt zusammenfassend dar, welche Funktionen von ADO.NET 2.0 für alle Datenbanken und welche nur für den Microsoft SQL Server verfügbar sein werden.

	Alle Datenprovider	Microsoft SQL Server 7.0/2000	Microsoft SQL Server 2005
Providerfabriken	X	X	X
Partial Trust	X	X	X
Auflisten der Microsoft SQL Server	X	X	X
Connection String Builder	X	X	X
Schema-Zugriff	X	X	X
Batch-Größe für Datenadapter	X	X	X
Tracing Support	X	X	X
Leeren des Verbindungs-Pools	SqlClient, OracleClient, OleDb	X	X
Multiple Active Results Sets (MARS)			X
Benachrichtigungen über Daten-änderungen			X
Unterstützung des Isolationlevel »Snapshot«			X
Asynchrone Befehlsausführung	X	X	X

▶

	Alle Datenprovider	Microsoft SQL Server 7.0/2000	Microsoft SQL Server 2005
Client Failover			X
Bulkcopy	X	X	X
Kennwortänderung			X
Statistiken	X	X	X
Neue Datentypen			X

***Tabelle 9.2**: ADO.NET 2.0 Feature-Matrix [MSDN09]*