

## Kapitel 11

# Datenzugriff mit ADO.NET

### In diesem Kapitel:

Einführung	532
ADO versus ADO.NET	532
Basisfunktionsumfang von ADO.NET 1.0/1.1	533
Neuerungen in ADO.NET 2.0 im Überblick	533
Neuerungen in ADO.NET 3.x im Überblick	535
Neuerungen in ADO.NET 4.0 im Überblick	535
Die ADO.NET-Architektur	535
Datenbankverbindungen (Connection)	541
Verbindungszeichenfolgen zusammensetzen mit demConnectionStringBuilder	542
Befehlsausführung mit Befehlsobjekten	544
Daten lesen mit einem Datareader	550
Daten lesen und verändern mit einem DataSet	555
LINQ-to-DataSet	573
Datenproviderunabhängiger Datenzugriff durch Provider-Fabriken	577
Benachrichtigungen über Datenänderungen (Query Notifications)	578
Massenkopieren (Bulkcopy/Bulkimport)	582
Providerstatistiken	585
Datenbankschema auslesen	587
Zusatzdienste für ADO.NET	588
ADO.NET Data Services	588
ADO.NET Synchronization Services	589
Positionierung von ADO.NET und Ausblick	593

# Einführung

ADO.NET ist die Datenbankzugriffsschnittstelle für .NET-Anwendungen und Nachfolger der COM-basierten ActiveX Data Objects (ADO). Die Schreibweise *ActiveX Data Objects .NET* wird jedoch selten verwendet; in der Regel findet man nur die Abkürzung. ADO.NET ist eine der wichtigen Teilbibliotheken der .NET-Klassenbibliothek (Namensraum System.Data).

**HINWEIS** Bis .NET 3.5 war ADO.NET die einzige Datenbankzugriffsschnittstelle. Seit .NET 3.5 sind höherwertige Konzepte wie LINQ to SQL und die ADO.NET Entity Framework Object Services hinzugekommen (siehe Kapitel »Objektrelationales Mapping (ORM) mit .NET«). ADO.NET behält aber weiterhin seine Bedeutung als unterste und direkte Datenbankzugriffsschnittstelle, bei der der Entwickler die volle Kontrolle über die ausgeführten Befehle hat.

Das vorliegende Kapitel betrachtet im Schwerpunkt den Zugriff auf Microsoft SQL Server 2005/2008 (inkl. R2). Andere Datenbanksysteme werden aus Platzgründen nur am Rande betrachtet. Microsoft SQL Server 2000 wird zwar noch von ADO.NET unterstützt, aber nicht mehr von den Datenbankwerkzeugen in Visual Studio 2010.

## ADO versus ADO.NET

Die Gemeinsamkeiten zwischen dem klassischen, COM-basierten *ADO* und dem .NET-basierten *ADO.NET* sind nicht sehr groß. Microsoft hat zentrale Teile des Datenzugriffs auf dem Weg von COM zu .NET stark verändert:

- Die Trennung in unterschiedliche Schnittstellen für verschiedene Zielgruppen (OLEDB und ADO) wurde aufgehoben; ADO.NET ist eine einheitliche Schnittstelle für alle Sprachen. Die ADO.NET Managed Data Provider ersetzen die bisherigen OLEDB-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. Das Auslesen der Daten erfolgt mit einem Client Cursor, d. h., der Client durchläuft nach dem anfänglichen Einlesen die Daten ohne Rückgriff auf die Datenquelle. Ein *DataSet* ist eine Art In-Memory-Datenbank, die zu einem späteren Zeitpunkt mit der ursprünglichen oder einer anderen Datenquelle synchronisiert werden kann. Ein *DataSet* kann mehr als eine Tabelle enthalten; die Tabellen können hierarchische Beziehungen untereinander besitzen.
- Der Zugriff auf Datenbanken mit einem Server Cursor ist nur lesend möglich durch Einsatz 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 und ist auch in .NET 4.0 nicht enthalten.
- Ein *DataSet* besitzt einen XML-Relationalen Mapper (XRM): 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 Verbindungspooling durch, um bestehende Datenbankverbindungen wieder zu verwenden.

**HINWEIS** Die Datenbankzugriffsschnittstelle ADO.NET gehört zu den Teilen des .NET Framework, 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.

## Basisfunktionsumfang von ADO.NET 1.0 / 1.1

In ADO.NET 1.0/1.1 waren folgende Funktionen enthalten:

- Treiber für Microsoft SQL Server- und Oracle-Datenbanken sowie Kompatibilitätstreiber für vorhandene OLEDB- und ODBC-Datenbanken
- Verbindungsaufbau und -abbau mit Verbindungsobjekten (Connection)
- Ausführung von SQL-Befehlen und gespeicherten Prozeduren mit Befehlsobjekten (Command)
- Daten lesen mit speziellen Datenlese-Objekten (mehrere Klassen, die das Wort `DataReader` im Namen tragen)
- Daten lesen und ändern mit `DataSet`-Objekten (Klasse `DataSet` sowie dazugehörend: `DataTable`, `DataRow`,  `DataColumn`, `DataRelation`, `DataView`, `DataAdapter` und `CommandBuilder`)
- Serialisierung von `DataSet`-Objekten in XML-Form
- Unterstützung für Datenbanktransaktionen (Transaction)

## Neuerungen in ADO.NET 2.0 im Überblick

In ADO.NET 2.0 (November 2005) hat Microsoft folgende Funktionen neu eingeführt:

- Asynchrone Befehlsausführung
- Massenkopieren (Bulkcopy / Bulkimport)
- Mehrere gleichzeitige Aktionen 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 aus einem `DataSet`
- Datenproviderunabhängiges Programmieren durch Provider-Fabriken
- 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`
- Etwas mehr Kontrolle über das Verbindungspooling

- Ändern von SQL Server-Datenbankbenutzerkennwörtern
- Unterstützung für benutzerdefinierte SQL Server-Datentypen (ab 2005)
- Unterstützung für Snapshot Isolation im SQL Server (ab 2005)
- Unterstützung für Datenbankspiegelung (Client Failover) im SQL Server (ab 2005)
- Verzicht auf MDAC (Microsoft Data Access Components) 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 die unten folgende »ADO.NET-Funktionsmatrix«. Zahlreiche Funktionen stehen leider nur in Zusammenhang mit dem Microsoft SQL Server 2005 / 2008 (inkl. R2) zur Verfügung.

### ADO.NET-Funktionsmatrix

Die folgende Tabelle stellt zusammenfassend dar, welche Funktionen von ADO.NET für alle Datenbanken und welche nur für den Microsoft SQL Server verfügbar sind. Die leergelassenen Zelle in bedeuten, dass dies providerabhängig ist. Der ADO.NET-Treiber von Oracle (ODP.NET) unterstützt inzwischen z.B. auch Bulk Copy und Benachrichtigungen über Datenänderungen.

	Alle ADO.NET-Datenprovider	Microsoft SQL Server 7.0 / 2000	Microsoft SQL Server 2005 / 2008 (inkl. R2)
Provider-Fabriken	X	X	X
Auflisten der verfügbaren Datenbankserver	X	X	X
Klasse <code>ConnectionStringBuilder</code>	X	X	X
Schema-Zugriff	X	X	X
Batch-Größe für Datenadapter	X	X	X
Leeren des Verbindungspools		X	X
Multiple Active Results Sets (MARS)			X
Benachrichtigungen über Datenänderungen			X
Unterstützung des Isolationsebene <i>Snapshot</i>			X
Asynchrone Befehlsausführung		X	X
Client Failover			X
Bulkcopy		X	X
Kennwortänderung			X
Statistiken		X	X

**Tabelle 11.1** ADO.NET-Funktionsmatrix

# Neuerungen in ADO.NET 3.x im Überblick

Die Neuerungen in .NET 3.0 und .NET 3.5 sind geringer im Vergleich zu den Neuerungen in ADO.NET 2.0.

**HINWEIS**

Microsoft spricht nicht offiziell von *ADO.NET 3.5*, sondern nur von *neuen Funktionen für ADO.NET in .NET 3.5*.

## ADO.NET im .NET Framework 3.0

Das .NET Framework 3.0 enthält gegenüber dem .NET Framework 2.0 keinerlei Neuerungen für ADO.NET.

## ADO.NET im .NET Framework 3.5

Das .NET Framework 3.5 enthält folgende Neuerungen:

- LIQN to DataSet (hier in diesem Kapitel beschrieben)
- Verbesserung des Spaltenzugriffs für untypisierte DataSets (hier in diesem Kapitel beschrieben)
- Mehrschichtunterstützung für typisierte DataSets (hier in diesem Kapitel beschrieben)
- ADO.NET Sync Services (hier in diesem Kapitel beschrieben unter »Zusatzdienste für ADO.NET«)
- ADO.NET Entity Framework (ab Service Pack 1, siehe Kapitel zu »Objektrelationales Mapping (ORM) mit ADO.NET Entity Framework «)
- ADO.NET Data Services (ab Service Pack 1, siehe Kapitel 14 zu »Windows Communication Foundation (WCF) 4.0«)

# Neuerungen in ADO.NET 4.0 im Überblick

Für das klassische ADO.NET gibt es keine Neuerungen. Es gibt aber umfangreiche Neuerungen in:

- ADO.NET Entity Framework. Die zweite Version trägt die Nummer »4.0« und hat wesentliche Verbesserungen (siehe Kapitel 12 zu »Objektrelationales Mapping (ORM) mit ADO.NET Entity Framework 4.0«)
- Die ADO.NET Data Services heißen nun WCF Data Services und sind erweitert (siehe Kapitel zu »Windows Communication Foundation (WCF)«)

## Die ADO.NET-Architektur

Im Veröffentlichen von Datenbankschnittstellen ist Microsoft seit vielen Jahren Weltmeister: ODBC, OLEDB, 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 Objects (DAO) sowie die Open Database Connectivity (ODBC) gelten dabei schon länger als veraltet. In der Grafik durch eine gestrichelte Linie berücksichtigt wurde die Möglichkeit, vom Managed Code aus via COM-Interoperabilität und P/Invoke die alten Schnittstellen zu nutzen, was aber nicht empfehlenswert ist.

## Providermodell

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

Die Grafik zeigt auch das Verhältnis von ADO.NET zu den Objektrelationalen Mappern LIQN to SQL und ADO.NET Entity Framework: Beide setzen auf ADO.NET auf. Bei dem ADO.NET Entity Framework gibt es noch die Besonderheit, dass es auch einen ADO.NET Data Provider *EntityClient* gibt, mit dem man über ADO.NET auf das im ADO.NET Entity Framework hinterlegte Konzeptuelle Datenmodell zugreifen kann. Dies wird auch im Kapitel zu »Objektrelationales Mapping (ORM) mit .NET« behandelt.

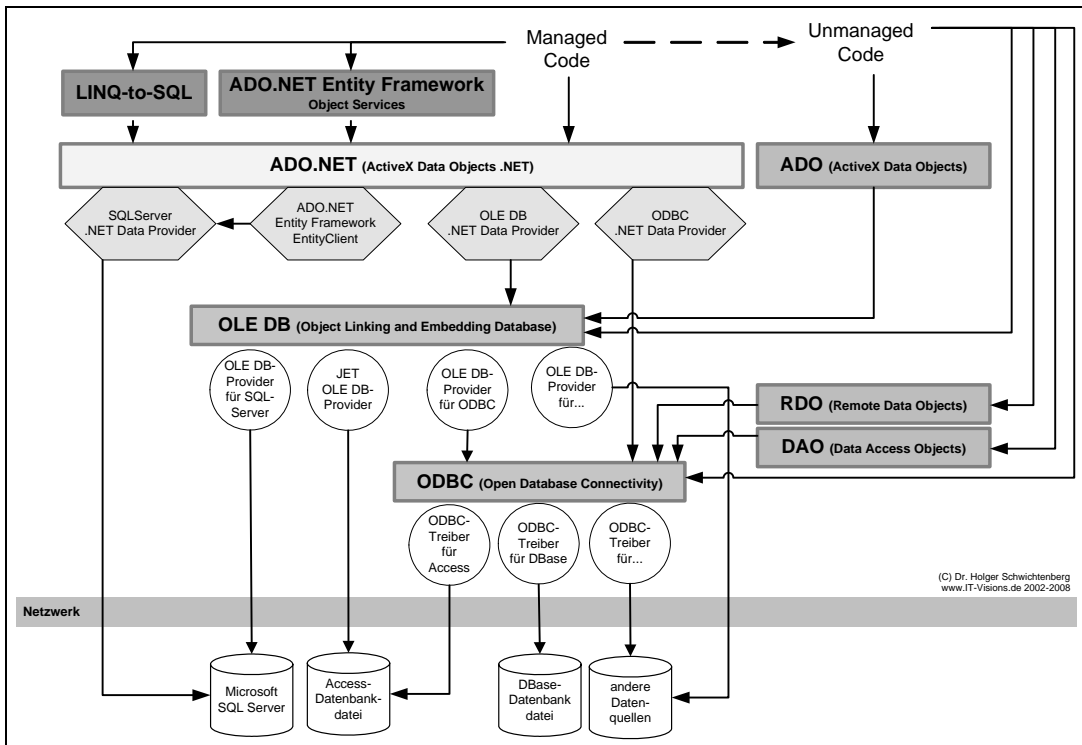


Abbildung 11.1 ADO.NET-Providerarchitektur und Zusammenspiel mit anderen Datenzugriffsschnittstellen

## Datenprovider von Microsoft

ADO.NET wird in .NET 2.0/3.0/3.5/4.0 mit folgenden Daten Providern (alias .NET Data Provider oder Managed Data Provider) ausgeliefert:

- **System.Data.SqlClient** (spezieller Treiber für Microsoft SQL Server 7.0/2000 und 2005/2008 (inkl. R2); dieser Treiber wird auch innerhalb des SQL Server 2005/2008 (inkl. R2) für Managed-Code-Anwendungen benutzt)

- `System.Data.SqlClient` (spezieller Treiber für Microsoft SQL Server CE)
- `System.Data.OracleClient` (spezieller Treiber für Oracle-Datenbanken)
- `System.Data.OleDb` (Brücke zu OLEDB-Providern)
- `System.Data.Odbc` (Brücke zu ODBC-Treibern)

**HINWEIS** Eine interne Änderung in ADO.NET ab Version 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. Dadurch wird die Leistung verbessert.

**WICHTIG** In .NET 4.0 hat Microsoft den Treiber `System.Data.OracleClient` in den Status »deprecated« gesetzt was bedeutet, dass Microsoft ihn in zukünftigen .NET-Versionen nicht mehr unterstützen wird. Sie sollten den offiziellen Treiber von Oracle (ODP.NET), siehe [ORACLE01]) oder einen Drittanbietertreiber (z.B. von DevArt dotConnect for Oracle [DEVART01]) verwenden.

## Datenprovider von anderen Herstellern

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

- **DevArt** Datenprovider für MySQL, Oracle, Postgres, SQLite [ADONET01]
- **DataDirect** Datenprovider für Oracle, DB2, Sybase, Microsoft SQL Server [ADONET02]
- **FireBird** Open Source-Datenprovider für Firebird [ADONET03]
- **Open Link** Datenprovider für MySQL, Informix, DB2, Oracle, Ingres, Sybase und Microsoft SQL Server [ADONET05]
- **Open Source** ADO.NET Provider für MySQL und PostgreSQL [ADONET04]

**TIPP** Weitere ADO.NET-Datenprovider finden Sie in der Werkzeug- und Komponentenreferenz des Autors unter [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 in ADO.NET seit Version 2.0 enthalten. Ein Fernzugriff auf die Provider eines anderen Systems ist jedoch leider nicht möglich.

Die installierten Provider sind nicht in der Registrierungsdatenbank, sondern – wie es sich für eine .NET-Anwendung gehört – in der *machine.config* abgelegt (Sektion `<system.data>` `<DbProviderFactories>`).

### Beispiel

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

```
public static void GetAllProviders()
{
    Demo.Print("=== 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)
            Demo.Print(c.ColumnName + ":" + provider[c]);
        Demo.Print("---");
    }
}
```

**Listing 11.1** Auflistung der vorhandenen ADO.NET-Datenprovider [VerschiedeneDemos/ADONET/ProviderEnumerationen]

## Der Weg der Daten von der Datenquelle bis zum Verbraucher

Die nachstehende Abbildung zeigt die möglichen Datenwege in ADO.NET 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 ein providerspezifisches DataReader-Objekt oder über ein providerunabhängiges DataSet-Objekt zum Datenverbraucher gelangen. Das DataSet-Objekt benötigt zur Beschaffung der Daten ein DataAdapter-Objekt, das wiederum in jedem Datenprovider separat zu implementieren ist. Man kann in Visual Studio eine Wrapper-Klasse für ein DataSet für eine oder mehrere Tabellen generieren lassen. Dieses so genannte *Typisierte DataSet* (*Typed DataSets*) bietet dem Entwickler mehr Komfort im Datenzugriff.

Seit .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. Seit .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 Webforms: Während Windows Forms mit einer allgemeinen Klasse BindingSource auf Basis von typisierten DataSets arbeitet, verwendet ASP.NET providerspezifische Klassen (z. B. SqlDataSource, AccessDataSource). WPF besitzt ein sehr eng in den Kern von WPF integriertes Datenbindungskonzept (vgl. Kapitel 15 zu »Windows Presentation Foundation (WPF) 4.0«).

Ab .NET 3.5 besteht die Möglichkeit, die Abfrage von Daten und das Verknüpfen von Daten in einem DataSet (sowohl einem normalen, untypisierten als auch einem typisierten DataSet) durch Einsatz von LINQ to DataSet wesentlich mächtiger und eleganter zu gestalten.

---

**ACHTUNG** Für einige Anwendungen ist es wichtig, sich nicht auf einen speziellen Datenprovider festzulegen. Die Möglichkeiten für providerunabhängiges Programmieren wurden schon in .NET 2.0 verbessert. Mehr dazu werden Sie im Abschnitt »Datenproviderunabhängiger Datenzugriff durch Provider-Fabriken« erfahren.

---

**HINWEIS** Datengebundene Steuerelemente und Datenbindung werden in den jeweiligen Kapiteln zu Oberflächentechnologien (also Windows-Oberflächen mit Windows Forms, Windows Presentation Foundation (WPF) und ASP.NET) erläutert.

---



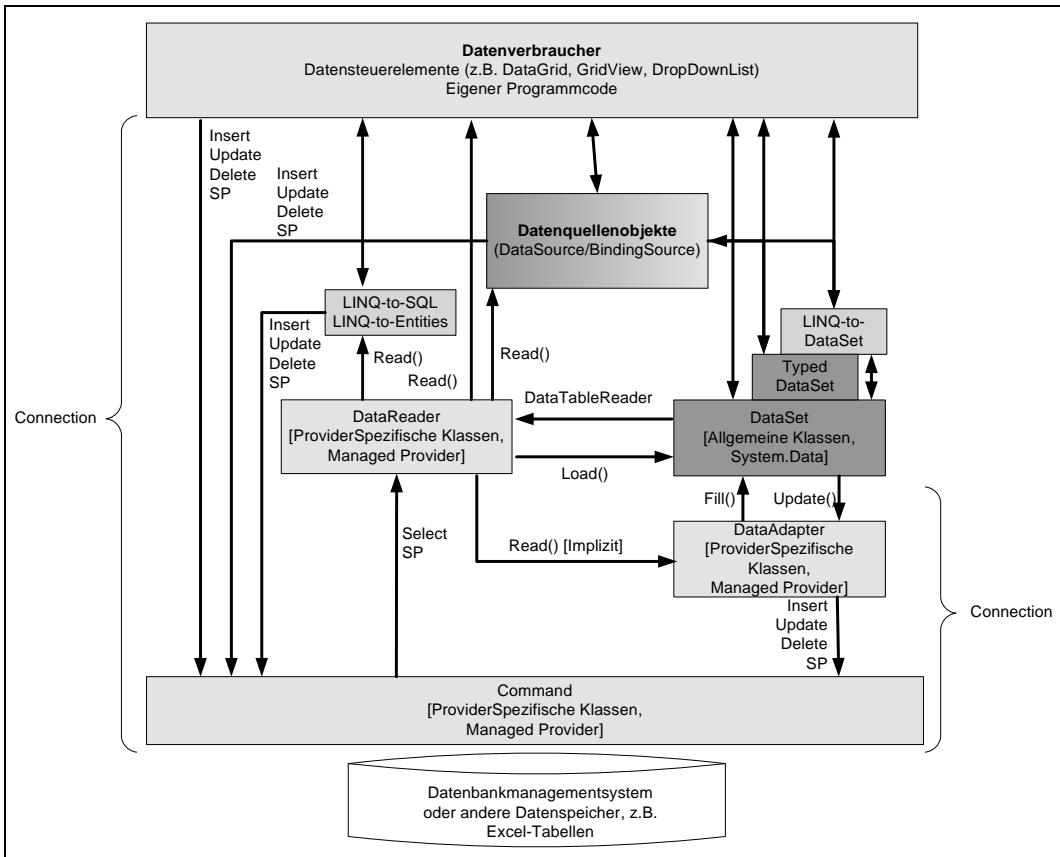


Abbildung 11.2 Datenwege in ADO.NET

## Datareader versus DataSet

In der Einführung wurden bereits die Datenzugriffsmodelle Datareader und DataSet angesprochen. Die folgende Tabelle und die Grafik vergleichen die beiden Zugriffsverfahren im Detail.

	Datareader	DataSet
Modell	Server Cursor	Client Cursor
Implementiert in	Jedem Datenprovider	System.Data
Basisklassen	DbDataReader MarshalByRefObject Object	MarshalByValueComponent Object
Schnittstellen	IDataReader, IDisposable, IDataRecord, IEnumerable	ICollection, IXmlSerializable, ISupportInitialize, ISerializable
Daten lesen	Ja	Ja

	Datareader	DataSet
Daten vorwärts lesen	Ja	Ja
Daten rückwärts lesen	Nein	Ja
Direktzugriff auf beliebigen Datensatz	Nein	Ja
Direktzugriff auf beliebige Spalte im Datensatz	Ja	Ja
Daten verändern	Nein, nur über separate Command-Objekte	Ja (über Datenadapter)
Befehlszeugung für Datenänderung	Komplett manuell	tlw. automatisch (CommandBuilder)
Zwischenspeicher für Daten	Nein	Ja
Änderungshistorie	Nein	Ja
Speicherverbrauch	Niedrig	Hoch
Geeignet für Datentransport zwischen Schichten	Nein	Ja

Tabelle 11.2 Datareader vs. DataSet

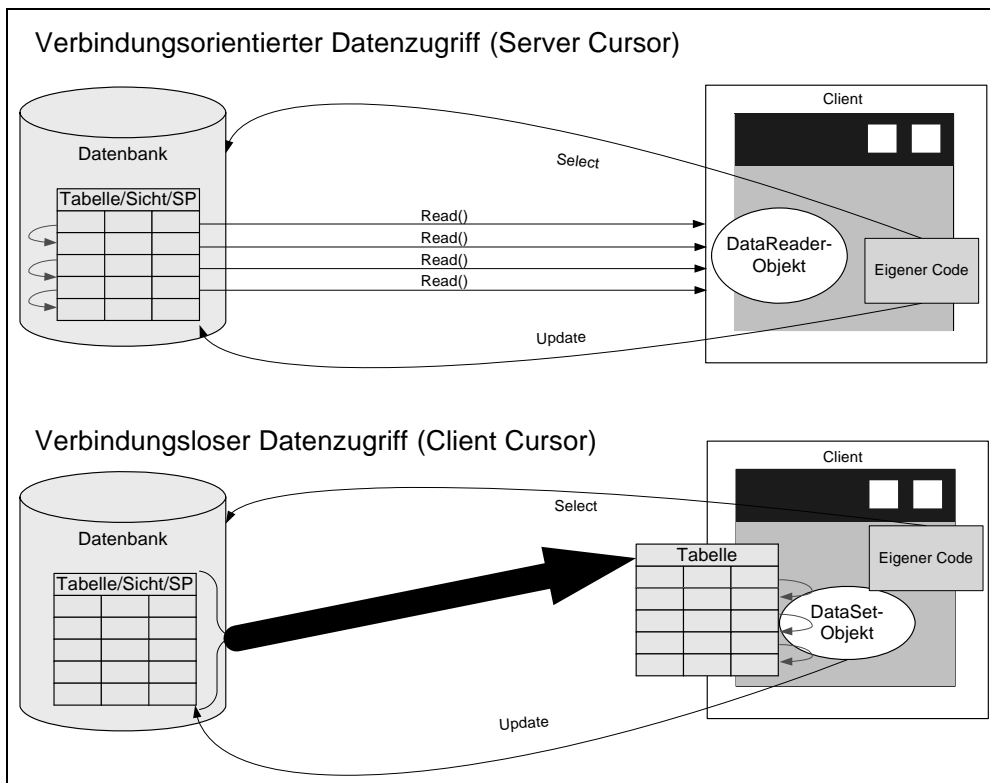


Abbildung 11.3 Vergleich der Zugriffsmodelle

**WICHTIG** Es gibt im .NET Framework keinen schreibenden Cursor. Eine solche Implementierung war ursprünglich für .NET 2.0 vorgesehen, wurde von Microsoft dann aber im endgültigen Produkt nicht ausgeliefert.

## Datenbankverbindungen (Connection)

Egal welche Datenzugriffsform gewählt wird und egal welche Aktion ausgeführt werden soll: Für die Kommunikation mit dem Datenbankmanagementsystem ist immer eine Verbindung notwendig.

### Verbindungen aufbauen und schließen

Jeder Datenprovider hat eine eigene Implementierung für die Verbindungsklasse: `SqlConnection`, `OracleConnection`, `OleDbConnection`, usw. Bei der Instanziierung dieser Objekte kann die Verbindungszeichenfolge übergeben werden. Danach erfolgt der Aufruf von `Open()`. Eine Verbindung muss geschlossen werden durch `Close()`.

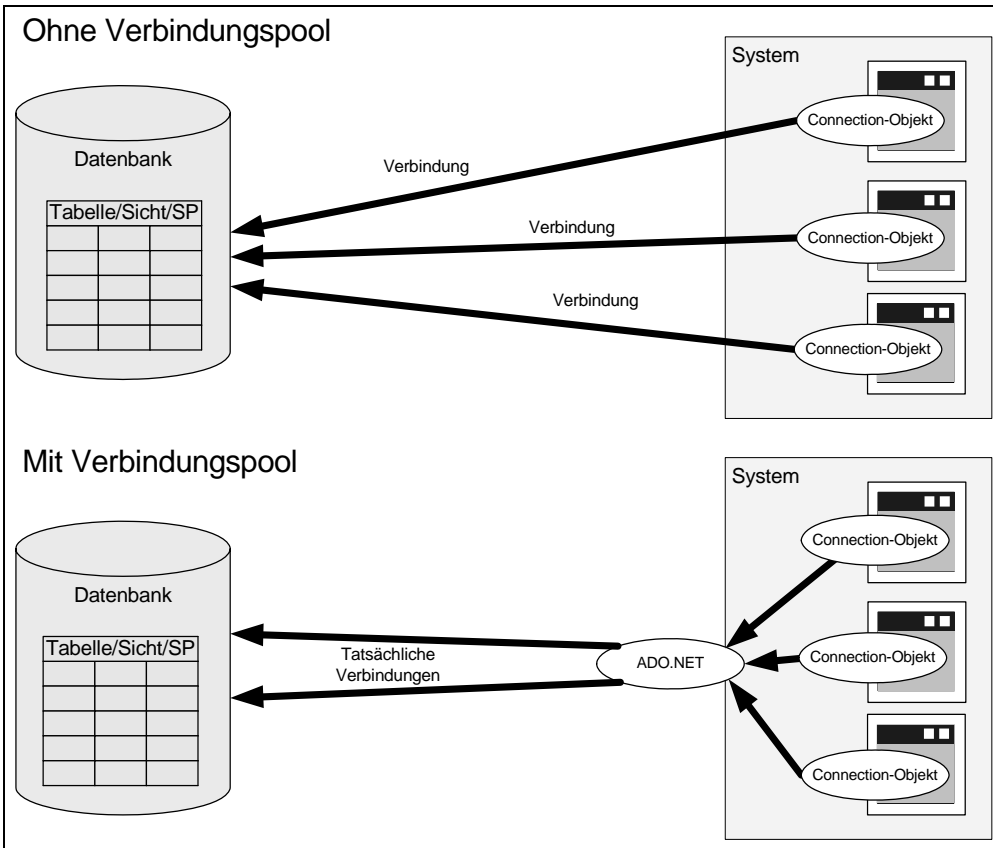
```
// === Daten lesen mit einem DataReader
public void DataReader_Demo()
{
    const string CONNSTRING = @"Integrated Security=SSPI;Persist Security Info=False;Initial
                                Catalog=WorldWideWings;Data Source=Server01\sqlexpress";
    // Verbindung aufbauen
    SqlConnection sqlConn = new SqlConnection(CONNSTRING);
    sqlConn.Open();
    ...
    // Verbindung schließen
    sqlConn.Close();
}
```

**Listing 11.2** Verbindung aufbauen und schließen

### Verbindungspooling

ADO.NET-Datenprovider implementieren (auch schon in ADO.NET 1.x) eigene Verbindungs-Pooling-mechanismen (*ADO.NET Connection Pooling*). ADO.NET schließt eine Datenbankverbindung nicht, wenn eine Anwendung sie schließt, sondern hält die Verbindung für die spätere Wiederverwendung (auch durch andere Anwendungen) – für eine bestimmte Zeit – offen. Die Verbindungen werden in einem so genannten Verbindungspool zur späteren Wiederverwendung vorgehalten.

Ein ADO.NET-Entwickler muss sich daher nicht um die optimale Dauer einer Datenbankverbindung kümmern; er sollte die Verbindung immer so schnell wie möglich schließen. Die Steuerung des Verbindungspoolings erfolgt über Parameter in der Verbindungszeichenfolge.



**Abbildung 11.4** Verbindungspooling in ADO.NET

In ADO.NET 1.x hatte der Entwickler aber auch keinen Einfluss darauf, wann die Verbindung tatsächlich abgebaut wurde. Die Datenprovider für Microsoft SQL Server und Oracle ermöglichen es dem Entwickler ab ADO.NET 2.0, den Zeitpunkt zum Leeren der Verbindungspools selbst zu bestimmen. Die folgenden Methoden werden als statische Methoden auf den entsprechenden `Connection`-Objekten angeboten:

- Die Methode `ClearPool(Conn)` entfernt die angegebene Verbindung aus dem Verbindungspool
- Die Methode `ClearAllPools()` löscht alle Verbindungen aus dem Verbindungspool

## Verbindungszeichenfolgen zusammensetzen mit dem `ConnectionStringBuilder`

ADO.NET (ab Version 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 Datenprovider-spezifische Attribute, die die Verwendung vereinfachen.

## Beispiel

Das Codefragment 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 11.3** Zusammensetzen einer Verbindungszeichenfolge [/VerschiedeneDemos/ADONET/StringBuilder]

## 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 im Kapitel 9 »*.NET-Klassenbibliothek 4.0*« unter dem Stichwort *System.Configuration* behandelt.

## Ermittlung der verfügbaren Microsoft SQL Server

Mithilfe der Klasse `SqlDataSourceEnumerator` kann man ab ADO.NET Version 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`

## 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
public static void GetAllSQLServers2()
{
    DataTable servers = SqlDataSourceEnumerator.Instance.GetDataSources();
    foreach (DataRow src in servers.Rows)
        PrintTable(servers);
}
```

**Listing 11.4** Ausgabe der erreichbaren SQL Server-Installationen [/VerschiedeneDemos/ADONET/ProviderEnumerationen]

## 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 (ab 2.0) bietet im Zusammenhang mit dem SQL Server 2005/2008 (inkl. R2) die Möglichkeit, ein Kennwort auch eleganter über die statische Methode `ChangePassword()` in der Klasse `SqlConnection` zu ändern. Microsoft hat diese eine Methode *Password Change API* genannt.

## 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 11.5** Kennwortänderung für das sa-Benutzerkonto [/VerschiedeneDemos/ADONET/BulkImport/MSSQL\_KennwortAenderung]

## Befehlsausführung mit Befehlsobjekten

Sie können jegliche Form von SQL-Befehlen, gespeicherten Prozeduren (Stored Procedures) oder anderen für das jeweilige Datenbankmanagementsystem verständlichen Befehlszeichenketten direkt auf der Datenbank aufrufen, indem Sie eine providerspezifische Befehlsklasse nutzen.

**ACHTUNG** Weder Visual Studio noch das .NET Framework prüfen oder beeinflussen die über ein Befehlsobjekt abgesendeten Befehle. Diese sind aus der Sicht der Entwicklungsumgebung und der Laufzeitumgebung nur Zeichenketten, für deren Korrektheit allein der Entwickler selbst bürgen muss.

## Methoden der Befehlsklassen

Neben dem bereits verwendeten `ExecuteReader()` stellen providerspezifische Befehlsklassen (enthält `Command` im Klassennamen, z. B. `SqlCommand`, `OracleCommand`, `OleDbCommand`, usw.) 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 ab .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 Transaktionsobjekt (Basisklasse `DbTransaction`, konkrete Klassen z. B. `SqlTransaction` und `OracleTransaction`), das `Commit()` und `Rollback()` als Methoden anbietet. Dieses eigenständige Transaktionsobjekt ist den einzelnen Befehlen explizit zuzuweisen (`sqlCmd.Transaction = t;`). Über das Attribut `IsolationLevel` kann die Art der Transaktion beeinflusst werden. Standardwert ist `ReadCommitted`.

### HINWEIS

Weitere Möglichkeiten zur Definition von Transaktionen finden Sie im Kapitel »Enterprise Services und Transaktionen« (als PDF).

## Beispiel zur Ausführung von Befehlen

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 funktioniert, 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, 8/1/2008, 500, 'F')";
    const string SQL2 = @"INSERT INTO GF_GebuchteFluege ([GF_PS_ID], [GF_FL_FlugNr], [GF_FlugDatum], " +
        "[GF_Preis], [GF_Klasse]) " + "VALUES (1, 203, 8/1/2008, 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: " + sqlCommand.ExecuteScalar().ToString());
SqlTransaction t = null;
try
{
    t = sqlConn.BeginTransaction();
    sqlCommand.CommandText = SQL1;
    sqlCommand.Transaction = t;
    Demo.Print(SQL1);
    Demo.Print("Betroffene Zeilen: " + sqlCommand.ExecuteNonQuery());
    sqlCommand.CommandText = SQL2;
    sqlCommand.Transaction = t;
    Demo.Print(SQL2);
    Demo.Print("Betroffene Zeilen: " + sqlCommand.ExecuteNonQuery());
    t.Commit();
    Demo.Print("Transaktion erfolgreich ausgeführt!");
}
catch (Exception ex)
{
    t.Rollback();
    Demo.Print("Transaktion fehlgeschlagen: " + ex.Message);
}
finally
{
    sqlCommand.CommandText = SQL3;
    Demo.Print("Anzahl Datensätze nachher: " + sqlCommand.ExecuteScalar().ToString());
    sqlConn.Close();
}
}

```

**Listing 11.6** Befehle in einer Transaktion ausführen [/VerschiedeneDemos/ADONET/Command]

## Parameter für Befehle

Meistens sind die SQL-Befehle nicht statisch, sondern setzen sich aus Parametern zusammen. Grundsätzlich ist es dabei möglich, die SQL-Befehlszeichenkette aus einzelnen Teilzeichenketten zusammenzubauen:

```

string SQL2 = @"INSERT INTO GF_GebuchteFluege ([GF_PS_ID], [GF_FL_FlugNr], [GF_Preis], [GF_Klasse]) " +
"VALUES (" + PS_ID + ", " + FL_ID + ", 0, 'F')";

```

In Szenarien, in denen die Parameter aus Eingaben des Endbenutzers stammen, besteht dabei aber die Gefahr der so genannten *SQL Injection*, einer Angriffsmöglichkeit, bei der ein unberechtigter Benutzer den gesamten Datenbestand lesen oder verändern kann. Außerdem können viele Datenbankmanagementsysteme parametrisierte Befehle zwischenspeichern für eine spätere Verwendung, während zusammengesetzte Befehlszeichenketten immer wieder neu geprüft und übersetzt werden müssen.

Daher ist es deutlich besser, die Befehle mithilfe der Parameters-Menge eines Command-Objekts zusammenzusetzen. Dabei enthält die SQL-Befehlszeichenkette benannte Platzhalter (eingeleitet durch »@« für Microsoft SQL Server oder »:« für Oracle-Datenbanken), die durch die Parameters-Menge gefüllt werden. Empfohlen seit ADO.NET 2.0 ist, dass die Parameters-Menge mit `AddWithValue()` statt mit `Add()` befüllt werden sollte. Die Methode `Add()` ist als veraltet gekennzeichnet.



In dem nachfolgenden Beispiel wird das Command-Objekt wieder verwendet. In diesem Fall muss man entweder die Parameters-Objektmenge mit `Clear()` vor der Wiederverwendung löschen oder direkt auf die bestehenden Parameter in der Objektmenge zugreifen, um sie neu zu befüllen. Dabei müssen nur die gegenüber der ersten Verwendung geänderten Werte wieder gesetzt werden.

```
// === SQL-Befehle in einer Transaktion ausführen unter Verwendung von Parametern
public void ADONET_Command_Parameters()
{
    Demo.PrintHeader("Command-Transaction-Demo mit Parametern");
    string CONNSTRING = DemoConfig.CONNSTRING;
    const string SQL1 = @"INSERT INTO GF_GebuchteFluege ([GF_PS_ID], [GF_FL_FlugNr], [GF_Preis],
[GF_Klasse]) " +
        "VALUES (@PS, @FL, @Preis, @Klasse)";
    const string SQL2 = @"INSERT INTO GF_GebuchteFluege ([GF_PS_ID], [GF_FL_FlugNr], [GF_Preis],
[GF_Klasse]) " +
        "VALUES (@PS, @FL, @Preis, @Klasse)";
    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;
        sqlCmd.Parameters.AddWithValue("@PS", 1);
        sqlCmd.Parameters.AddWithValue("@FL", 100);
        sqlCmd.Parameters.AddWithValue("@Preis", 599);
        sqlCmd.Parameters.AddWithValue("@Klasse", "F");

        Demo.Print(SQL1);
        Demo.Print("Betroffene Zeilen: " + sqlCmd.ExecuteNonQuery());
        sqlCmd.CommandText = SQL2;
        sqlCmd.Transaction = t;
        sqlCmd.Parameters["@FL"].Value = 200;
        sqlCmd.Parameters["@Preis"].Value = 699;
        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

```

```
{
    sqlCommand.CommandText = SQL3;
    Demo.Print("Anzahl Datensätze nachher: " + sqlCommand.ExecuteScalar().ToString());
    sqlCommand.Close();
}
```

**Listing 11.7** Befehle mit Parametern ausführen [/VerschiedeneDemos/ADONET/Command]

## Asynchrone Befehlsausführung

ADO.NET (ab Version 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 Rückruffroutine, die ein Objekt mit der Schnittstelle `IAsyncResult` empfängt

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

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

Nur die Klasse `SqlCommand`, also der Provider für Microsoft SQL Server, verfügt über diese Methoden, obwohl ursprünglich auch andere Provider davon profitieren sollten.

---

**WICHTIG** Wichtig ist, dass die Absicht, asynchrone Aufrufe auszuführen, schon in der Verbindungszeichenfolge mit `Asynchronous Processing=true` (oder kurz: `Async=true`) angezeigt wird.

Die ursprünglich für ADO.NET 2.0 angekündigte Funktion des asynchronen Verbindungsaufbaus ist zunächst aus dem Funktionsumfang von .NET 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()`.

## 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=Server01\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 11.8** Asynchrone Ausführung einer SELECT-Anweisung mit dem Polling-Modell [/VerschiedeneDemos/ADONET/AsynchroneBefehle.vb]

## 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=Server01\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 11.9** Asynchrone Ausführung einer SELECT-Anweisung mit dem Warte-Modell [/VerschiedeneDemos/ADONET/AsynchroneBefehle.vb]

## 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=Server01\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 11.10** Asynchrone Ausführung einer SELECT-Anweisung mit dem Callback-Modell [/VerschiedeneDemos/ADONET/AsynchroneBefehle.vb]

## 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 mit dem DataReader nicht möglich (wie der Name schon sagt). Zum Daten ändern kann man als komplementäres Konzept dann nur die Command-Objekte einsetzen, wobei der Entwickler für die Konstruktion der Änderungsbefehle (in SQL DML oder durch Aufruf von gespeicherten Prozeduren) selbst verantwortlich ist. 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/forward-only recordset* (zu Deutsch: *Vorwärtsdatensatzzeiger*).

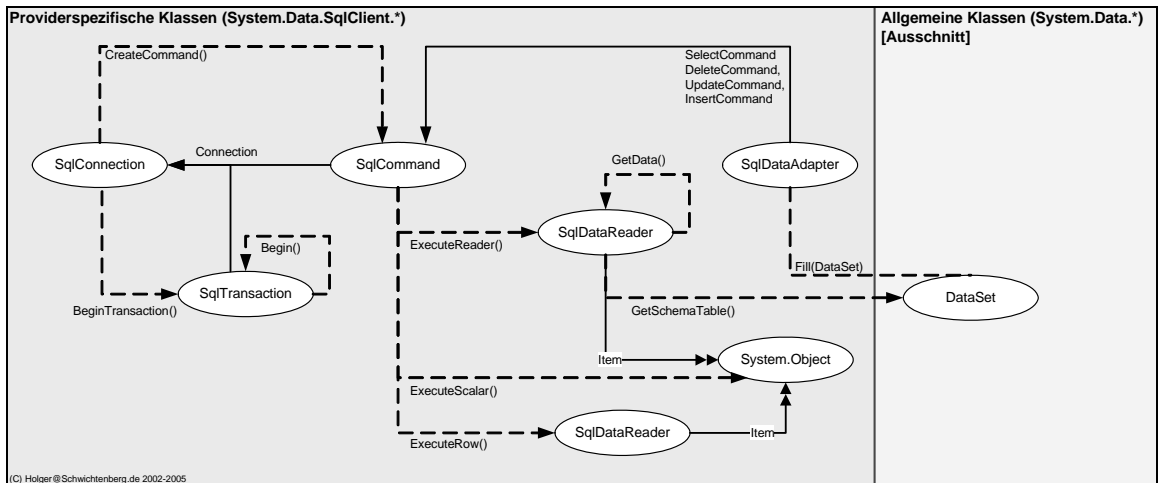
Jeder ADO.NET-Datenprovider enthält seine eigene DataReader-Implementierung, so dass es zahlreiche verschiedene DataReader-Klassen im .NET Framework gibt (z. B. SqlDataReader und OleDbDataReader). Die

`DataReader`-Klassen sind abgeleitet von `System.Data.ProviderBase.DbDataReaderBase` und implementieren `System.Data.IDataReader`.

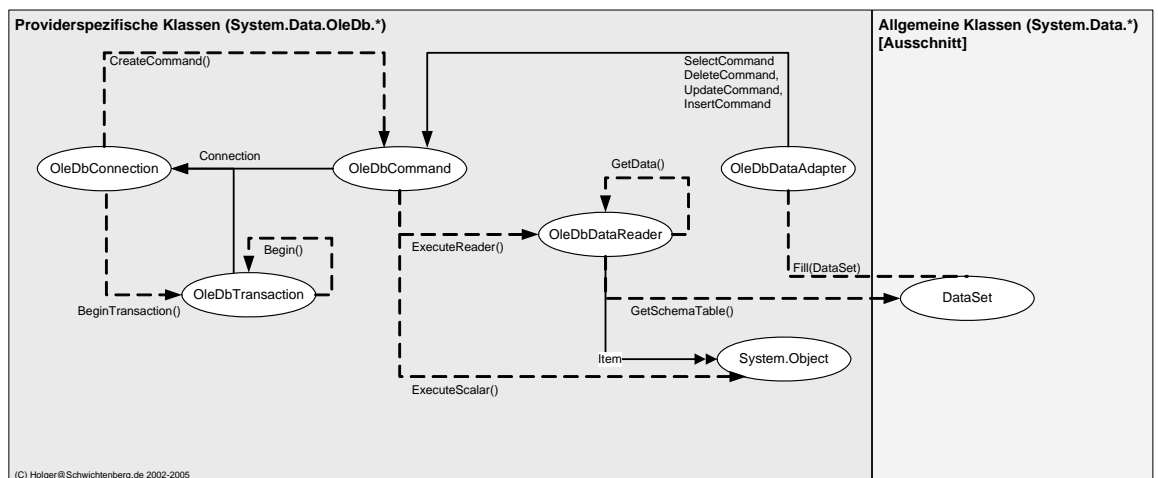
Ein `DataReader` benötigt zur Beschaffung der Daten ein Befehlsobjekt (z.B. `SqlCommand`, `OleDbCommand`), das ebenso providerspezifisch ist (z.B. `SqlCommand` und `OleDbCommand`). Für die Verbindung zur Datenbank selbst wird ein providerspezifisches Verbindungsobjekt (z.B. `SqlConnection` oder `OleDbConnection`) benötigt. Die nachstehenden Abbildungen zeigen den Zusammenhang dieser Objekte am Beispiel der Datenprovider für OLEDB 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.

**TIPP**

Später in diesem Buch werden Sie noch die providerunabhängige Nutzung eines `DataReader`s kennenlernen.



**Abbildung 11.5** Objektmodell für den Datenprovider für Microsoft SQL Server (`System.Data.SqlClient`)



**Abbildung 11.6** Objektmodell für den Datenprovider für OLEDB (`System.Data.OleDb`)

**HINWEIS** Das Connection-Objekt muss explizit mit `Open()` geöffnet werden, bevor man `ExecuteReader()` ausführt.

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 Verbindungspooling-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 Verbindungsobjekt. Bei der Instanziierung dieses Objekts kann die Verbindungszeichenfolge übergeben werden.
- Instanziierung der Klasse `Command` und Bindung dieses Objekts an das Verbindungsobjekt über die Eigenschaft `Connection`
- Festlegung eines SQL-Befehls, der Daten liefert (also beispielsweise `SELECT` oder eine Stored Procedure), in einem Befehlsobjekt in der Eigenschaft `CommandText`
- Die Ausführung der Methode `ExecuteReader()` in dem Befehlsobjekt liefert als Ergebnis ein `Datareader`-Objekt

Danach kann der `Datareader` entweder an ein datenkonsumierendes Steuerelement (z.B. `DataGrid`) 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, weil 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=Server01\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 11.11** Daten lesen mit einem DataReader [VerschiedeneDemos/ADONET/Datareader]

## 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++)
            Demo.Print("Spalte: " + reader.GetName(i) + "\t = " + reader.GetValue(i));
}
```

**Listing 11.12** Hilfsroutine `PrintReader()`

## Behandlung von Null-Werten

Leere Zellen in der Datenbank (*null*-Werte) werden im `DataReader`-Objekt dadurch signalisiert, dass der Zugriff auf die betreffende Spalte ein Objekt vom Typ `System.DBNull` liefert. Dies muss explizit geprüft werden (`IsDBNull()`). Leider ist bisher für den `Datareader` keine Integration zwischen *null*-Werten in der Datenbank und den in .NET 2.0 eingeführten wertelosen Wertetypen vorhanden. Der `Datareader` liefert eine Instanz von `DBNull`, was sich nicht automatisch in den Wert `null` bzw. `nothing` von .NET umwandeln lässt. Daher muss man hier manuell prüfen. Diese sehr lästige Einschränkung für das `DataReader`-Objekt gilt auch noch für .NET 4.0. Besser haben es seit .NET 3.5 die `DataSets` (siehe in dem dortigen Kapitel die Ausführung zu *null*-Werten).

```
int? x;
if (dataReader.IsDBNull(index)) { x = null; }
else { x = dataReader.GetInt32(index); }
```

**Listing 11.13** Prüfung auf *null*-Werte im `DataReader`-Objekt und Zuweisung an einen wertelosen Wertetyp

## Multiple Active Results Sets (MARS)

In ADO.NET 1.x konnte zu einem Zeitpunkt pro Verbindung nur ein Datareader oder ein Befehl aktiv sein; es war also zum Beispiel nicht möglich, zwei Datareader oder einen Datareader und einen Befehl gleichzeitig auf einer Verbindung zu durchlaufen. Wenn schon ein Datareader geöffnet ist, führt das Öffnen eines zweiten (oder eines Befehls) 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 ab Version 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 (*Data Manipulation Language*, Datenmanipulationssprache – dazu gehören INSERT, UPDATE, DELETE) gleichzeitig auf einer Verbindung ausgeführt werden.

---

**ACHTUNG** MARS ist jedoch bisher nur für den SQL Server 2005/2008 (inkl. R2) verfügbar. Die Nutzung von MARS muss dem SQL Server beim Aufbau der Verbindung in der Verbindungszeichenfolge angezeigt werden mit `MultipleActiveResultSets=true`

---

### 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 / 2008!)");

    const string CONNSTRING = @"Integrated Security=SSPI;Persist Security Info=False;" +
        "Initial Catalog=WorldWideWings;Data Source=Server01\squlexpress; MultipleActiveResultSets=true";
    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();
}
```

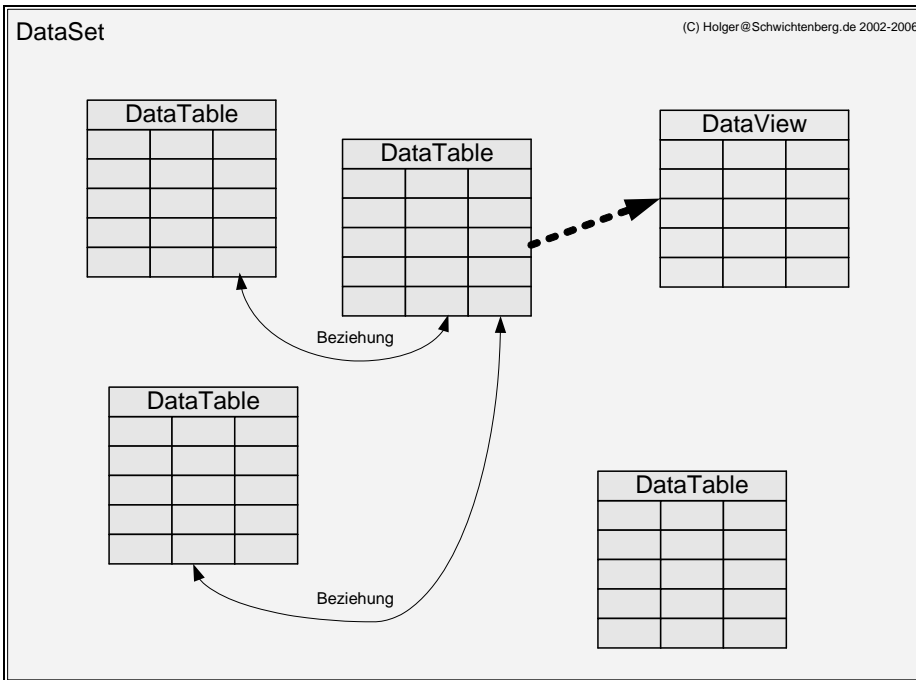
**Listing 11.14** Beispiel für MARS bei Verbindungen zum SQL Server [/VerschiedeneDemos/ADONET/MSSQL\_MARS.vb]

## Daten lesen und verändern 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. Ebenfalls lassen sich hierarchische Beziehungen zwischen einzelnen Tabellen darstellen und im DataSet speichern. Dadurch ist eine Verarbeitung hierarchischer Datenmengen möglich. Im Untergrund verwendet ein DataSet übrigens einen DataReader zum Einlesen der Daten.

Ein DataSet ist ein clientseitiger Datenzwischenspeicher, der die Änderung mitprotokolliert. Das DataSet nimmt keine Sperrung von Datensätzen auf der Datenquelle vor, sondern verwendet immer das so genannte *Optimistische Sperren*, d.h., Änderungskonflikte treten erst auf, wenn man versucht, die Daten zurückzuschreiben. Das Konzept eines serverseitigen Cursors ist in ADO.NET nur durch die DataReader-Klasse realisiert. Einen serverseitigen Cursor mit Schreibfunktion und pessimistischem (»echtem«) Sperren gibt es in .NET auch in Version 4.0 nicht.



**Abbildung 11.7** Aufbau eines DataSet

### WICHTIG

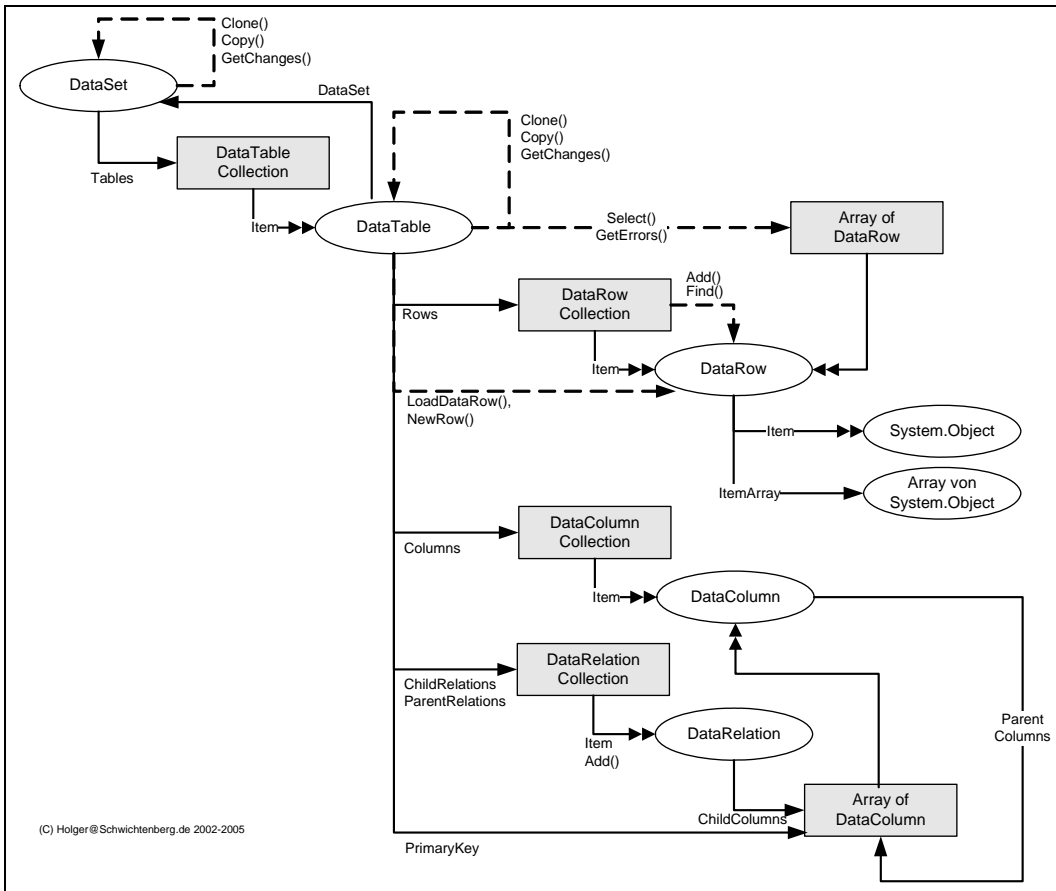
Warnung: Das DataSet verbraucht sehr viel mehr Speicher als eine selbst definierte Datenstruktur. Das Abholen von Daten mit einem Datareader, das Speichern in einer selbst definierten Datenstruktur und das Speichern von Änderungen mit direkten SQL-Befehlen macht zwar mehr Arbeit bei der Entwicklung, ist aber wesentlich effizienter bei der Ausführung. Dies ist insbesondere bei serverbasierten Anwendungen wichtig.

## 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.

Während die DataTable-Objekte in ADO.NET 1.x dem DataSet-Objekt noch völlig untergeordnet waren, besitzt die DataTable-Klasse in ADO.NET seit Version 2.0 viele der Import- und Exportmöglichkeiten, über die auch die DataSet-Klasse verfügt.

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 11.8** Objektmodell des DataSet

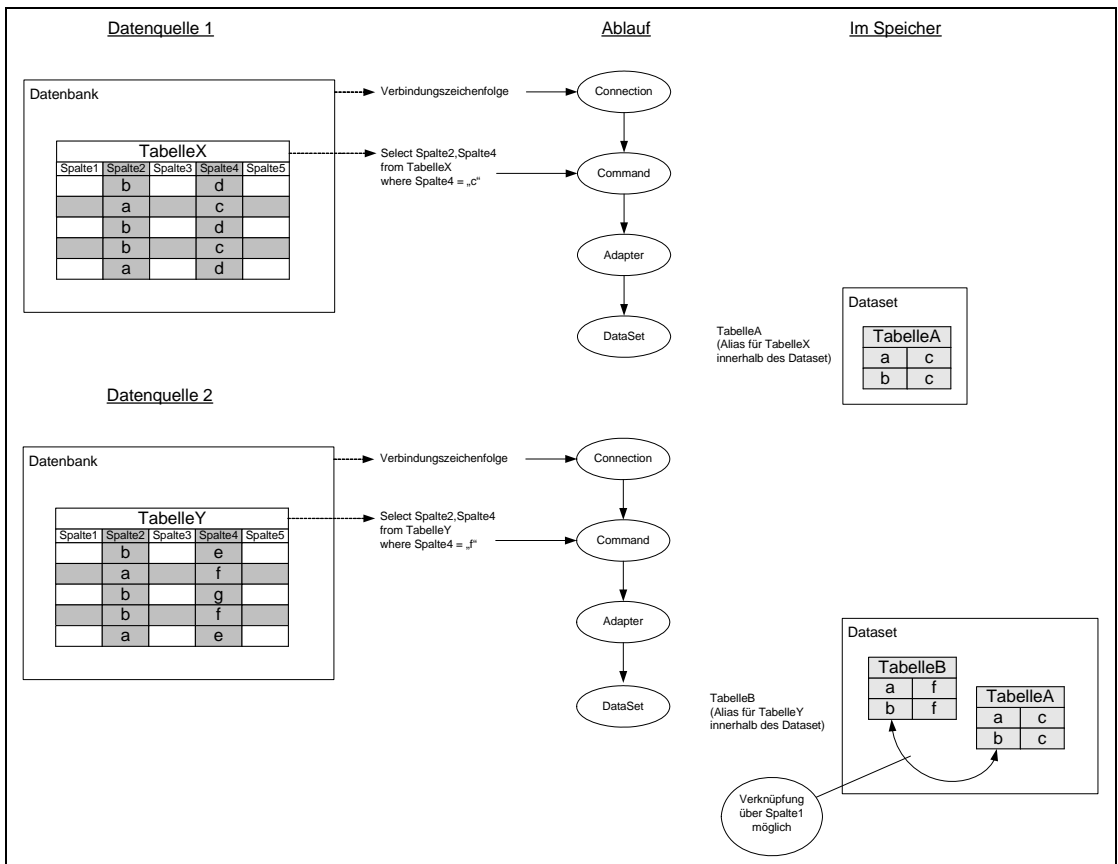
## Daten lesen mit DataSets

Ein DataSet benötigt zum Einlesen von Daten einen providerspezifischen Datenadapter. Das Lesen von Daten mit einem DataSet läuft in folgenden Schritten ab:

- 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 Befehls, der Daten liefert (also z.B. SELECT oder eine gespeicherte Prozedur), im SqlCommand-Objekt in der Eigenschaft CommandText
- Instanziierung des Datenadapters auf Basis des Befehlsobjekts
- 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`. Sofern der Befehl im Datenadapter mehrere Ergebnismengen liefert, entstehen mehrere Tabellen mit fortlaufenden Nummern `Table0`, `Table1`, `Table` usw.
- 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 Zwischenspeicher (Cache) für Daten.



**Abbildung 11.9** Einlesen von mehreren Datenquellen in ein DataSet

## Beispiel: Auslesen von flachen 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=Server01\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 11.15** Ausgabe der Tabelle FL\_Fluege mit einem DataSet [/VerschiedeneDemos/ADONET/DataSet\_Lesen]

**TIPP** Ab .NET 3.5 gibt es eine alternative Möglichkeit zum Zugriff auf die Zellen eines DataRow-Objekts über die generische Erweiterungsmethode `Field<Typ>()`. So kann man schreiben:

- `dr.Field<string>("FL_Abflugort ")` statt `dr["FL_Abflugort"].ToString()`
- `dr.Field<Int32>("FL_FreiePlaetze")` statt `Convert.ToInt32(dr["FL_FreiePlaetze "])`
- usw.

Das Beschreiben erfolgt dann über `SetField<Typ>()`, z. B.

- `dr.SetField<int>("FL_FreiePlaetze", 89)`
- `dr.SetField<DateTime>("FL_Datum", DateTime.Now)`
- usw.

## Behandlung von null-Werten

Bei der Behandlung von *null*-Werten sind grundsätzlich die gleichen Fallunterscheidungen notwendig wie bei *DataReader*-Objekten.

```
public virtual int Insert(int FL_FlugNr, string FL_Abflugort, string FL_Zielort,
    System.Nullable<System.DateTime> FL_Datum, System.Nullable<bool> FL_NichtRaucherFlug,
    System.Nullable<short> FL_Plaetze, System.Nullable<int> FL_FreiePlaetze, System.Nullable<int>
    FL_PI_MI_MitarbeiterNr, System.Nullable<int> FL_AnzahlStarts, System.Nullable<System.DateTime>
    FL_EingerichtetAm) {
    this.Adapter.InsertCommand.Parameters[0].Value = ((int)(FL_FlugNr));
    if ((FL_Abflugort == null)) {
        this.Adapter.InsertCommand.Parameters[1].Value = System.DBNull.Value;
    }
    else {
        this.Adapter.InsertCommand.Parameters[1].Value = ((string)(FL_Abflugort));
    }
}
```

**Listing 11.16** Ausschnitt aus dem generierten Code eines typisierten DataSet

**TIPP** Ab .NET 3.5 gibt es eine elegantere Lösung für untypisierte DataSets über die Erweiterungsmethode `Field<Typ>()`. Mit dieser Methode lässt sich jede Zelle in einen wertelosen Wertetyp konvertieren, so dass man `DBNull`-Werte auf elegante Weise in `null`-Werte umwandeln kann.

```
int? FL_FreiePlaetze = dr.Field<int?>("FL_FreiePlaetze");
if (!FL_FreiePlaetze.HasValue)
{
    Console.WriteLine("Kein StartDatum gesetzt!");
    dr.SetField<int?>("FL_FreiePlaetze", dr.Field<Int32?>("FL_Plaetze"));
};
```

## Beispiel: Arbeit mit Tabellenverknüpfungen

Das folgende Beispiel zeigt, dass man mithilfe mehrerer Datenadapter mehrere Datenmengen in ein DataSet-Objekt laden und diese dort optional mittels eines *DataRelation*-Objekts hierarchisch verknüpfen kann. Bei der Ausgabe können dann zu jedem Datensatz der Eltern-Tabelle die entsprechenden Detaildatensätze angezeigt werden (bzw. umgekehrt). In dem nachfolgenden Beispiel wird die Verbindung zwischen der Sicht *AllePassagiere* (Eltern-Tabelle) und der Tabelle *GF\_GebuchteFluege* (Kind-Tabelle) hergestellt.

Das *DataRelation*-Objekt erwartet bei der Instanziierung einen Namen für die Verknüpfung (hier: *Passagiere\_GebuchteFluege*) 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, weil die erstellte Beziehung sonst 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.

Sie können immer nur zwei Tabellen verknüpfen. Echte relationale Verknüpfungen (Joins) kann man im DataSet ab .NET 3.5 mit LIQN to DataSet erzeugen (siehe dazu eigenen Abschnitt).

Innerhalb der Schleife über alle Zeilen in der Sicht *AllePassagiere* kann dann für jede DataRow mithilfe der Methode `GetChildRows()` die Menge der zugehörigen Buchungsdatensätze abgerufen werden. `GetChildRows()` erwartet als Parameter ein `DataRelation`-Objekt oder den Namen der Beziehung. Über die umgekehrte Navigationsrichtung von der Detail- zur Master-Tabelle existiert die Methode `GetParentRow()`.

```
// === 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=Server01\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 dal = new SqlDataAdapter(SQL1, CONNSTRING);
    SqlDataAdapter da2 = new SqlDataAdapter(SQL2, CONNSTRING);
    // --- Daten abholen
    dal.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 11.17** Verknüpfen der Sicht *AllePassagiere* und der Tabelle *GF\_GebuchteFluege* in einem DataSet [/VerschiedeneDemos/ADONET/DataSet\_Lesen]

```
Name: Schröder Vorname: Gerhard
  Flug: 102 Datum: 01.03.2008 00:00:00
  Flug: 203 Datum: 02.03.2008 00:00:00
Name: Merkel Vorname: Angela
  Flug: 200 Datum: 04.03.2008 00:00:00
```

**Listing 11.18** 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 *Table*, *Table1*, *Table2* usw. heißen. Mithilfe der TableMappings-Menge im Datenadapter kann man aber auch sinnvollere Namen vergeben. Diese Nutzung mehrerer SELECT-Anweisungen im Datenadapter wird nicht von allen ADO.NET-Datenprovidern unterstützt.

## Datensichten (Dataviews)

Innerhalb eines DataSets kann der Entwickler mit der Klasse `DataGridView` dynamische Sichten auf einzelne Datentabellen definieren, wobei das Filtern und Sortieren von Datensätzen möglich ist.

```
// Datensicht definieren
Demo.PrintHeader("Selektierte Daten:");
DataGridView dv = new DataGridView(dt);
dv.RowFilter = "FL_Abflugort = 'Paris'";
dv.Sort = "FL_Zielort desc";
// --- Iteration über Daten
foreach (DataGridViewRow drv in dv)
{
    DataRow dr = drv.Row;
    Demo.Print("Flug-ID: " + dr["FL_FlugNr"] + " von " + dr[1] + " nach " + dr[2]);
}
```

**Listing 11.19** Definition und Ausgabe einer Datensicht auf Basis eines DataTable-Objekts [/VerschiedeneDemos/ADONET/DataSet\_Lesen]

## Daten ändern mit DataSets

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

- Jede Zelle kann direkt jederzeit beschrieben werden. Einen expliziten »Änderungsmodus« gibt es nicht.
- 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 der Datenadapter die notwendigen SQL-DML-Befehle nicht ganz automatisch. Vorgesehen ist, dass der Entwickler die Befehle manuell in dem Datenadapter ablegt. Von dieser lästigen Arbeit kann er sich aber etwas entlasten durch die so genannten Befehlserzeugerklassen. Befehlserzeugerklassen sind providerspezifische Klassen (z.B. `SqlCommandBuilder`, `OleDbCommandBuilder`), die auf Basis eines `SELECT`-Befehls passende SQL-DML-Befehle zur Laufzeit erzeugen. Eine Befehlserzeugerklasse erwartet bei der Instanziierung ein `DataAdapter`-Objekt – mehr ist nicht zu tun für den Entwickler. Leider funktionieren die `CommandBuilder` lediglich 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.

**WICHTIG** Die Befehlserzeugerklassen funktionieren nur, wenn die Tabelle in der Datenbank einen Primärschlüssel besitzt und es keinen `JOIN` in dem SQL-Befehl gibt.

### Batch-Größe für Datenadapter

Durch das seit ADO.NET Version 2.0 neu eingeführte Attribut `UpdateBatchSize` kann die Anzahl der durch den Datenadapter gleichzeitig zu übermittelnden Änderungen beliebig gesetzt werden, z.B.:

```
da.UpdateBatchSize = 50
```

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

### Konfliktoptionen

Die Befehlserzeugerklassen erzeugen wahlweise SQL-Befehle, die optimistisches Sperren oder kein Sperren berücksichtigen. Pessimistisches Sperren ist – wie bereits an anderen Stellen erwähnt – in ADO.NET nicht verfügbar.

*Kein Sperren* (`OverwriteChanges`) bedeutet, dass die Änderungen ausgeführt werden, ohne Berücksichtigung, ob andere Benutzer die betreffenden Zeilen inzwischen geändert haben. In der Bedingung des SQL-Befehls kommen nur die Primärschlüssel vor.

Beim *Optimistischen Sperren* (`CompareAllSearchableValues`) werden alle ursprünglichen Spaltenwerte in die Bedingung einbezogen, so dass auffällt, wenn ein Benutzer / Prozess zwischenzeitlich den Datensatz geändert hat. Gab es eine Änderung, wird eine Ausnahme (`DbConcurrencyException`) ausgelöst.

Die o.g. Optionen sind durch `CommandBuilder.ConflictOption` zu setzen. Alternativ ist eine Steuerung auch in den Datenquellensteuerelementen von ADO.NET durch das dortige Attribut `ConflictDetection` möglich.

## Beispiel: Daten ändern

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 um eins erhöht. 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 Florenz 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=Server01\squlexpress";
    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("Löschke: " + 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"] = "Florenz";
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 11.20** Verschiedene Möglichkeiten zur Datenänderung in einem DataSet [/VerschiedeneDemos/ADONET/DataSet\_Lesen]

```
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 11.21** Ausgabe des Beispiels

## Typisierte DataSets (Typed DataSets)

So genannte *typisierte DataSets* (engl. *Typed DataSets*) sind eine Abstraktionsform des Datenzugriffs, die von Visual Studio – nicht nur für Microsoft SQL Server, sondern für viele durch ADO.NET-Treiber unterstützte Datenbanktypen – angeboten wird. Ein typisiertes DataSet ist im Kern eine von Visual Studio generierte Klasse, die eine Verpackung (engl. *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.

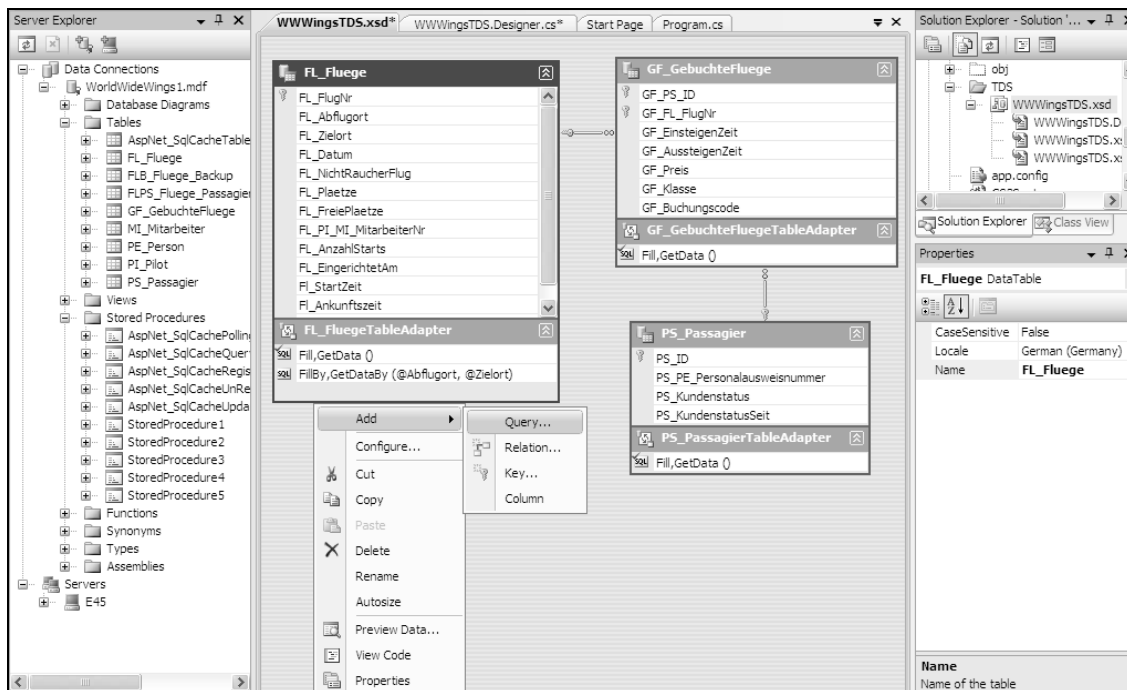
### Hintergründe

Das typisierte DataSet ist eine generierte .NET-Klasse, die von System.Data.DataSet erbt. Das typisierte DataSet enthält typisierte Tabellenklassen, die wiederum typisierte Zeilenklassen enthalten. Außerdem generiert Visual Studio eine so genannte Tabellenadapterklasse, die dazu dient, die Daten zu laden und das DataSet zu befüllen. Sie erbt aber nicht – wie man vermuten könnte – von einer der Datenadapterklassen aus ADO.NET. Die Tabellenadapterklasse des typisierten DataSet kapselt die Funktionen der Verbindungs-klasse, der Befehlsklasse und der Tabellenadapterklasse.

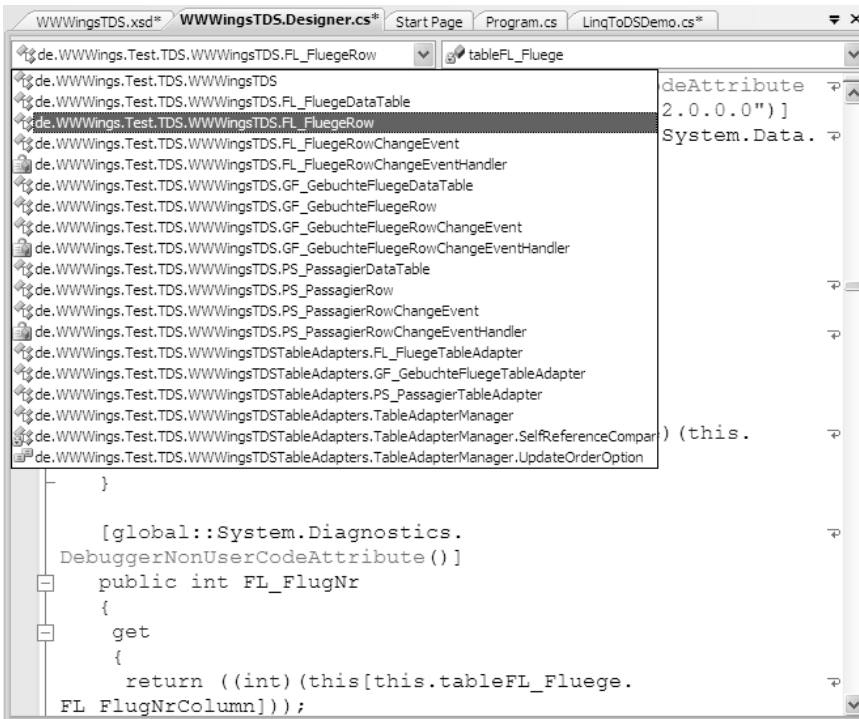
Neben diesen generierten Klassen gehören zu einem typisierten DataSet innerhalb der Entwicklungsumgebung auch eine XML-Schema-Beschreibung (.xsd-Datei) und die Beschreibung der Anordnung der Daten in der grafischen Ansicht (.xss- und .xcs-Dateien).

Visual Studio besitzt für DataSets eine eigene grafische Entwurfsoberfläche, die per Drag & Drop aus dem Ast *Datenverbindungen* im *Server Explorer* befüllt werden und auch Verknüpfungen zwischen DataTable-Objekten in einem DataSet-Objekt modellieren kann. Die Tabellenadapterklasse stellt eine `Fill()`-Methode zum Befüllen des typisierten DataSets bereit. Über einen Assistenten kann man weitere, parametrisierte Überladungen dieser Methoden bereitstellen, wobei die Parameter auf Parameter in einem SELECT-Befehl oder einer gespeicherten Prozedur abgebildet werden können. Man hat in dem Assistenten sogar die Wahl, für einen SELECT-Befehl automatisch eine gespeicherte Prozedur in der Datenbank anlegen zu lassen.

Typisierte DataSets existieren bereits seit Visual Studio .NET 2002, wurden aber in Visual Studio 2005 stark verändert und in Visual Studio 2008 nochmals etwas erweitert. In Visual Studio 2010 gibt es hier keine Neuerungen mehr.



**Abbildung 11.10** Verknüpfung dreier Tabellen im Designer für typisierte DataSets in Visual Studio



**Abbildung 11.11** Einblick in die Vielzahl der generierten Klassen und in den generierten Programmcode für eine typisierte Spaltenklasse (FL\_Fluege\_Row)

## Beispiel

Das folgende Beispiel zeigt das Befüllen und Ausgeben des in der obigen Abbildung dargestellten typisierten DataSets.

**ACHTUNG** Bitte beachten Sie, dass die in dem Beispiel verwendete Methode `Select()` zum Filtern lokal, d. h. im Hauptspeicher ausgeführt wird. Größere Datenmengen sollten Sie immer datenbankseitig filtern. Dies erreichen Sie über das Anlegen einer neuen Abfrage (*Query*) im Designer (siehe Bildschirmabbildung des Designers).

```
public void DataSet_TDS()
{
    Console.WriteLine("Flüge von ROM (TDS mit lokalem Filter)");

    // Tabellenadapter instanziiieren
    TDS.WWWingsTDS_TableAdapters.FL_FluegeTableAdapter ta =
        new de.WWWings.Test.TDS.WWWingsTDS_TableAdapters.FL_FluegeTableAdapter();
    // Typisiertes DataSet instanziiieren
    TDS.WWWingsTDS.FL_FluegeDataTable AlleFluege =
        new de.WWWings.Test.TDS.WWWingsTDS.FL_FluegeDataTable();
    // Tabelle laden
    ta.Fill(AlleFluege);
}
```

```
// Filter anwenden
DataRow[] FluegeVonRom = AlleFluege.Select("FL_Abflugort = 'Rom' and FL_Datum > '1/1/2008'");

// --- Iteration über Daten
foreach (TDS.WWingsTDS.FL_FluegeRow dr in FluegeVonRom)
{
    Console.WriteLine("Flugnummer: " + dr.FL_FlugNr + ": " + dr.FL_Abflugort + "->" + dr.FL_Zielort +
        ". Freie Plätze: " + dr.FL_FreiePlaetze);

    if (!dr.IsFL_StartZeitNull()) Console.WriteLine("Kein StartDatum gesetzt!");
}
}
```

**Listing 11.22** Beispiel für den Einsatz eines typisierten DataSets

## Mehrschichtunterstützung für typisierte DataSets

Seit Visual Studio 2008 kann man das typisierte DataSet im engeren Sinne von dem ebenfalls generierten Tabellenadapter trennen, d.h. diese können in zwei verschiedenen Projekten liegen. Ziel dabei ist es, eine Bibliothek zu erhalten, in der die Datenstrukturbeschreibung (in Form der von DataSet abgeleiteten Klassen) liegt und ein anderes Projekt, in dem diese Datenstruktur durch den Tabellenadapter befüllt wird. Damit hat man erstmals die Möglichkeit, auch mit typisierten DataSets eine saubere Schichtentrennung zu realisieren.

Diese Trennung vollzieht man durch die Einstellung *DataSet Project* im Eigenschaftsfenster des Designers eines typisierten DataSets. Hier kann man ein anderes Projekt der gleichen Projektmappe auswählen. Dadurch entsteht in dem gewählten Zielprojekt eine Datei *DataSetName.DataSet.Designer.cs* bzw. *.vb* mit dem Quellcode für die von DataSet abgeleitete Klasse.

---

**HINWEIS** Microsoft spricht in diesem Zusammenhang auch von *Multi-Tier-DataSets*.

---

## Diskussion typisierte DataSets

Typisierte DataSets sind ein Politikum, d.h. es gibt es unterschiedliche Meinungen dazu in der Entwickler- und Expertenszene. Auf der »Habenseite« stehen auf jeden Fall:

- Starke Vereinfachung des Einlesens von Daten in ein DataSet
- Typisierter Zugriff auf die Spaltennamen
- Vorteile des DataSet (insbesondere Serialisierbarkeit und Mehrschichteinsetzbarkeit)
- Das typisierte DataSet kann zur Drag & Drop-Datenbindung in Windows Forms-Fenstern oder per Programmcode verwendet werden.

Typisierte DataSets haben aber auch Nachteile:

- Es wird sehr viel Programmcode generiert (in dem obigen Beispiel sind es 3868 Zeilen!)
- Der generierte Programmcode ist nicht immer fehlerfrei. Eine Ausbesserung des Programmcodes ist aufwändig und nicht von Dauer, da jede kleine Änderung in dem Programmcode zur Neugenerierung des Codes führt.
- Ein typisiertes DataSet ist kein echtes Geschäftsobjekt, sondern nur eine von DataSet abgeleitete Klasse. Daher ist ein typisiertes DataSet kein Objektrelationaler Mapper im engeren Sinne.
- Das typisierte DataSet verwendet für Spalten, die DBNull sein können, in dem zugehörigen Attribut der typisierten DataRow-Klasse als Datentypen keinen wertelosen Wertetyp (Nullable Value Type), sondern bietet ein eigenes Verfahren über eine Methode `IsSpaltennameNull()` an. Dies integriert sich leider nicht reibungslos in andere Funktion von .NET und Visual Studio.
- Der Speicherrahmen des DataSets ist auch hier vorhanden

## Umwandlung zwischen DataSet und XML

ADO.NET enthält einen XML-Relationalen Mapper (XRM). Die Klassen DataSet und DataTable besitzen jeweils vier 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 DataSets, 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.



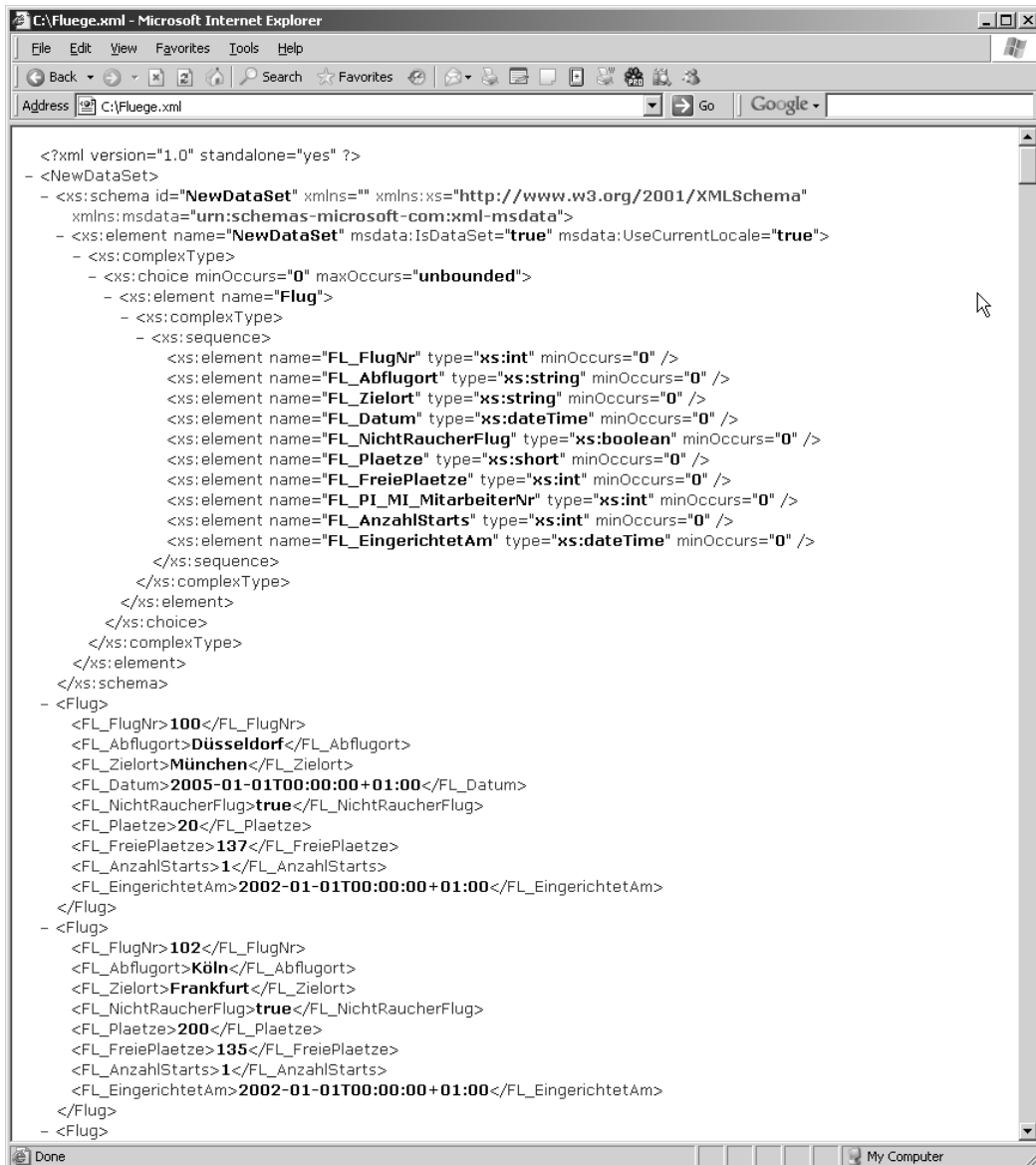


Abbildung 11.12 Darstellung eines DataSets inklusive Schema in XML-Form

Neben den Import- und Exportmöglichkeiten des DataSets 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

Seit 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.

```
DataSet ds = new DataSet();
ds.Load(reader, LoadOption.OverwriteChanges, new String[] { "FL_Fluege" });
Demo.Out(ds.GetXml());
```

**Listing 11.23** Befüllen eines DataSets mit einem Datareader

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

```
DataTableReader dtr = new DataTableReader(ds.Tables["FL_Fluege"]);
Demo.PrintReader(dtr);
```

**Listing 11.24** Auslesen eines DataSets mit einem Datareader

## 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). Seit Version 2.0 unterstützt ADO.NET 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 11.25** Aktivierung der binären Serialisierung für ein DataSet [/VerschiedeneDemos/ADONET/DataSetBinaerSerialisierung]

**TIPP** In ADO.NET 1.x konnten nur komplette DataSet-Objekte, nicht jedoch einzelne DataTable-Objekte serialisiert werden. Seit 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 (DataRow-Objekte eines DataTable-Objekts) 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« vor. 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.

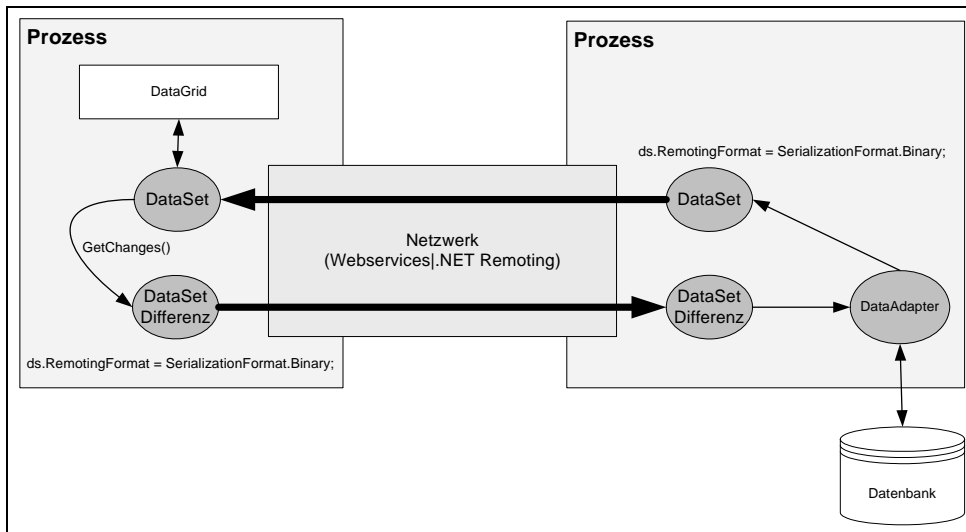


Abbildung 11.13 Effizientes Remoting von DataSets

## LIQN to DataSet

LIQN to DataSet ermöglicht die Abfrage von Daten in einem DataSet. Die vor .NET 3.5 in einem DataSet vorhandenen Möglichkeiten zum Filtern mit der Methode Select() und der Klasse DataView sowie dem Verknüpfen mit der DataRelation-Klasse waren sehr bescheiden. Mit LIQN to DataSet sind nun vielfältige Suchabfragen und Verknüpfungen über beliebige Spalten (Joins) möglich. Außerdem liefert LIQN to DataSet Verbesserungen hinsichtlich der Typisierung bei untypisierten DataSets.

**HINWEIS** Es können sowohl typisierte als auch untypisierte DataSets verwendet werden. LIQN to DataSet ist eine Schicht oberhalb des DataSets. Unterhalb (Datenprovider, Datenverbindungen, Befehlsobjekt) ändert sich durch den Einsatz von LIQN to DataSet nichts.

**ACHTUNG** Vergessen Sie niemals, dass LINQ to DataSet im RAM arbeitet. Bei großen Datenmengen bzw. wenn die Menge der zu übertragenden Daten wichtig ist, sollten Sie selektieren und verknüpfen immer im Datenbankmanagementsystem, nicht im RAM!

## Voraussetzung

Voraussetzung für die Nutzung von LINQ to DataSet ist die Referenzierung der Assembly *System.Data.DataSetExtensions.dll*, die Erweiterungen für die Klasse *DataTable* (*DataTableExtensions*) und *DataRow* (*DataRowExtensions*) bereitstellt.

**HINWEIS** Für allgemeine Erläuterungen zur Syntax von LINQ lesen Sie bitte das Kapitel 10 »Language Integrated Query (LINQ)« in diesem Buch.

## Abfragen über eine Tabelle

Um eine LINQ-Abfrage über eine Tabelle in Form eines *DataTable*-Objekts ausführen zu können, muss das *DataTable*-Objekt in ein Objekt des Typs *System.Data.EnumerableRowCollection<System.Data.DataRow>* umgewandelt werden. Dies geschieht mithilfe der Erweiterungsmethode *AsEnumerable()*.

```
// --- Tabelle aus einem vorher befüllten DataSet
DataTable AlleFluege = ds.Tables["Fluege"];

// --- LINQ Abfrage
IEnumerable<DataRow> FluegeVonRom =
    from Flug in AlleFluege.AsEnumerable()
    where Convert.ToDateTime(Flug["FL_Datum"]) >
        new DateTime(2008, 1, 1) &&
        Flug["FL_AbflugOrt"].ToString() == "Rom"
    select Flug;
```

**Listing 11.26** Eine LINQ-Abfrage über ein zuvor befülltes *DataTable*-Objekt

Zur Formulierung der Abfrage kann alternativ auch die durch die Klasse *DataRowExtensions* bereitgestellte generische Erweiterungsmethode *Field<Typ>()* verwendet werden, um die Spalten anzusprechen. In typisierten *DataSets* können die Spalten direkt über die Punktnotation angesprochen werden.

```
// --- Tabelle aus einem vorher befüllten DataSet
DataTable AlleFluege = ds.Tables["Fluege"];

// --- LINQ Abfrage
IEnumerable<DataRow> FluegeVonRom =
    from Flug in AlleFluege.AsEnumerable()
    where Flug.Field<DateTime>("FL_Datum") >
        new DateTime(2008, 1, 1) &&
        Flug.Field<string>("FL_AbflugOrt") == "Rom"
    select Flug;
```

**Listing 11.27** Eine LINQ-Abfrage über ein zuvor befülltes *DataTable*-Objekt (Alternative)

**TIPP** Das Ergebnis der Abfrage ist – sofern keine Projektion festgelegt wird, wieder eine Menge des Typs `System.Data.EnumerableRowCollection<System.Data.DataRow>`. Meist verwendet man als Typ aber allgemein die Schnittstelle `IEnumerable<DataRow>`.

Ein `DataTable`-Objekt kann man durch Aufruf der Methode `CopyToDataTable()` erhalten. Die Methode `AsDataView()` liefert ein `DataView`-Objekt.

## Abfragen über typisierte DataSets

Das folgende Beispiel zeigt die Abfrage eines typisierten DataSets mit LIQN to DataSet.

```
public void DataSet_TDS_LINQ()
{
    Console.WriteLine("Flüge von ROM (TDS mit lokalem Filter über LIQN to DataSet)");

    // Tabellenadapter instanziiieren
    TDS.WWwingsTDSTableAdapters.FL_FluegeTableAdapter ta = new
de.WWwings.Test.TDS.WWwingsTDSTableAdapters.FL_FluegeTableAdapter();
    // Typisiertes DataSet instanziiieren
    TDS.WWwingsTDS.FL_FluegeDataTable AlleFluege = new
de.WWwings.Test.TDS.WWwingsTDS.FL_FluegeDataTable();
    // Tabelle laden
    ta.Fill(AlleFluege);

    var FluegeVonRom =
        from Flug in AlleFluege
        where Flug.FL_Datum > new DateTime(2008, 1, 1)
        && Flug.FL_Abflugort == "Rom"
        select Flug;

    // --- Iteration über Daten
    foreach (TDS.WWwingsTDS.FL_FluegeRow dr in FluegeVonRom)
    {
        Console.WriteLine("Flugnummer: " + dr.FL_FlugNr + ": " + dr.FL_Abflugort + "->" + dr.FL_Zielort +
            ". Freie Plätze: " + dr.FL_FreiePlaetze);

        if (!dr.IsFL_StartZeitNull()) Console.WriteLine("Kein StartDatum gesetzt!");
    }
}
```

**Listing 11.28** LIQN to DataSet mit typisiertem DataSet

## Abfragen über mehrere Tabellen (Joins)

Mit der Klasse `DataRelation` (seit .NET 1.0 vorhanden) kann man nur hierarchische Eltern-Kind-Beziehungen erzeugen. Echte relationale Beziehungen (Joins) kann man mit dem `join`-Operator in LIQN to DataSet erzeugen, siehe nachfolgendes Beispiel.

```

public void DataSet_Beziehungen()
{
    Console.WriteLine("Passagiere mit ihren Flügen");

    // --- Parameter
    string CONNSTRING = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=H:\WWW\Datenbanken\WorldWideWings1.mdf;Integrated
Security=True;Connect Timeout=30;User Instance=True";
    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();

    // --- Zeiger auf Tabellen
    DataTable AllePassagiere = ds.Tables["AllePassagiere"];
    DataTable GebuchteFluege = ds.Tables["GebuchteFluege"];

    // --- Join definieren
    var Abfrage =
        from p in AllePassagiere.AsEnumerable()
        join b in GebuchteFluege.AsEnumerable()
        on p.Field<int>("PS_ID") equals
            b.Field<int>("GF_PS_ID")
        where p.Field<int>("AnzahlFluege") > 0
        select new
        {
            FlugNr =
                b.Field<int>("GF_FL_FlugNr"),
            Vorname =
                p.Field<string>("PE_Vorname"),
            Name =
                p.Field<string>("PE_Name"),
            PassagierID =
                p.Field<int>("PS_ID")
        };

    // --- Ergebnis anzeigen
    foreach (var Ergebnis in Abfrage)
    {
        Console.WriteLine("Passagier: " + Ergebnis.PassagierID + " Name: " + Ergebnis.Name + " Vorname: " +
            Ergebnis.Vorname + " gebucht auf Flug " + Ergebnis.FlugNr);
    }
}

```

**Listing 11.29** Join zwischen zwei Tabellen in einem DataSet

# Datenproviderunabhängiger Datenzugriff durch Provider-Fabriken

In den bisherigen Beispielen kamen verschiedene Klassen vor in Abhängigkeit davon, welcher Datenbank-provider (Microsoft Access oder Microsoft SQL Server) verwendet wurde. Dies ist unschön, wenn man auf verschiedene Datenbanken zugreifen muss oder die Datenbank später einmal wechseln möchte. ADO.NET unterstützt auch den providerunabhängigen Datenzugriff.

Stark vereinfacht wurde in ADO.NET seit Version 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 Namensraum `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 Daten Providern einen geeigneten Provider auswählt.

---

**ACHTUNG** Bei dem providerunabhängigen Datenzugriff findet 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` mithilfe des provider-unabhängigen Programmiermodells. Dabei wird bei der Instanziierung der Klasse `DbProviderFactory` der Daten-provider (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()
{
    Demo.Print("=== DEMO Provider Factory");
    const string PROVIDER = "System.Data.SqlClient";
    const string CONNSTRING = "Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=itvisions;Data Source=Server01\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];
Demo.Print("Anzahl Spalten: " + t.Columns.Count);
Demo.Print("Anzahl Zeilen: " + t.Rows.Count);
}
```

**Listing 11.30** Datenbankunabhängige Programmierung mit der DbProviderFactory [/VerschiedeneDemos/ADONET/ProviderFactory.vb]

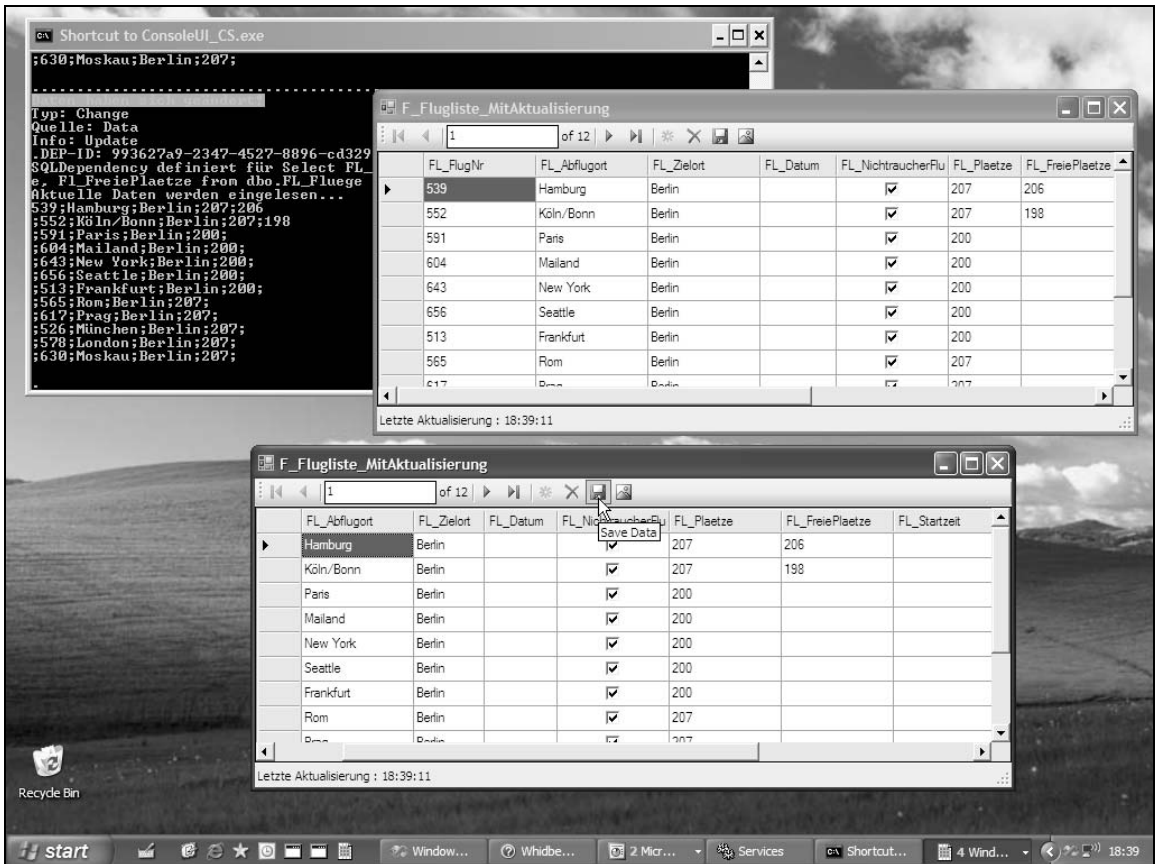
## Benachrichtigungen über Datenänderungen (Query Notifications)

Eine sehr interessante Option in ADO.NET seit Version 2.0 sind Benachrichtigungen über Datenänderungen (Query Notifications) im Push-Verfahren. Dabei meldet der Client gegenüber dem Datenbankmanagementsystem sein Interesse an einer bestimmten Datenmenge in Form eines Abonnements (Subscription) an. Jedes Mal, wenn eine Veränderung in dieser Datenmenge eintritt, wird eine Ereignisbehandlungsroutine aufgerufen, so dass der Client weiß, dass er die Daten erneut abrufen sollte. Durch Query Notifications entfällt das ständige Abfragen (Polling) der Datenbank.

**HINWEIS** Laut der Dokumentation sind Query Notifications nur für den Microsoft SQL Server (ab Version 2005) vorgesehen. Der Microsoft SQL Server besitzt eine echte Benachrichtigungsarchitektur auf Basis des integrierten Service Brokers. Die Netzwerklast ist minimal, weil es wirklich nur für den Fall von Datenänderungen zu Benachrichtigungen kommt. Ursprünglich geplant war, eine ähnliche Funktionalität (allerdings auf Basis von internem Polling) auch für die Vorgängermodelle 7.0 und 2000 anzubieten.

Query Notifications haben übrigens nichts mit den Notification Services des Microsoft SQL Server zu tun. Notification Services senden Benachrichtigungen an Anwender, z. B. in Form von E-Mails.





**Abbildung 11.14** Beispiel für den Einsatz von Datenänderungenbenachrichtigungen: Sobald in einer Tabelle Datenänderungen gespeichert werden, werden die anderen Anwendungen informiert

## Realisierung

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

Leider unterliegt der SELECT-Befehl starken Einschränkungen. Die wichtigsten der vielen Einschränkungen 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 Tabellename 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. Alternativ kann der Client über `HasChanges()` abfragen, ob es Änderungen gibt.

---

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

---

Datenänderungsbenachrichtigungen können zusammen mit ASP.NET-Zwischenspeicherung eingesetzt werden.

## Ablauf

Zum erfolgreichen Einrichten einer Benachrichtigung mit der Klasse `SqlDependency` sind folgende Schritte notwendig:

- Der Client muss die Benachrichtigungen für eine Verbindung aktivieren, indem er den Befehl `SqlDependency.Start(CONNSTRING)` aufruft
- Der Client öffnet die Datenbankverbindung
- Der Client erzeugt ein Befehlsobjekt für die zu überwachende Datenmenge
- Der Client erzeugt ein `SqlDependency`-Objekt für den Befehl: `dep = new SqlDependency(cmd)`
- Wichtig ist, dass nach dem Einrichten der `SqlDependency` der zugehörige SQL-Befehl genau einmal an den SQL Server gesendet wird. Erst dadurch wird die Query Notification aktiv. Die Ergebnisse müssen dabei nicht abgerufen werden.
- Der Client bindet eine Ereignisbehandlungsroutine an das Ereignis `OnChange()` im `SqlDependency`-Objekt

## 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 den Client 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`-Objekt verwendet werden.

```

public class Notifications_Demos
{
    private static SqlDependency dep = null, dep2 = null;
    static bool DataChanged = false;
    // == Warten auf Datenänderungen in der Flug-Tabelle
    public void run()
    {
        Demo.Print("Überwachung der Flugliste [Query Notifications]");
        const string CONNSTRING =
        @"Data Source=.sqlexpress;Initial Catalog=WorldWideWings3;Integrated Security=True;Pooling=False";
        const string SQL =
        "Select FL_FlugNr, FL_Abflugort, FL_Zielort, FL_Plaetze, FL_FreiePlaetze from dbo.FL_Fluege" +
        " where FL_Zielort = 'Berlin'";
        // Verbindung aufbauen
        while (true)
        {
            DataChanged = false;
            // Befehl definieren
            SqlCommand cmd = GetDep(CONNSTRING, SQL);
            Demo.Print("SQLDependency definiert für " + SQL);
            // Ereignisbehandlung binden
            dep.OnChange += new OnChangeEventHandler(dep_OnChange);

            // Befehl ausführen
            Demo.Print("Aktuelle Daten werden eingelesen...");
            SqlDataReader r = cmd.ExecuteReader();
            Demo.PrintReader(r);
            cmd = null;
            // Warten!!
            while (!DataChanged)
            { System.Threading.Thread.Sleep(1000); Console.Write("."); }
        }
    }
    private static SqlCommand GetDep(string CONNSTRING, string SQL)
    {
        SqlDependency.Start(CONNSTRING);
        SqlConnection conn = new SqlConnection(CONNSTRING);
        conn.Open();
        SqlCommand cmd = new SqlCommand(SQL, conn);
        dep = new SqlDependency(cmd);
        Demo.Print("DEP-ID: " + dep.Id);
        return cmd;
    }
    // Ereignisbehandlung für geänderte Daten
    static void dep_OnChange(object sender, SqlNotificationEventArgs e)
    {
        if (e.Type == SqlNotificationType.Subscribe)
        {
            Demo.Print("Fehler: " + e.Info.ToString());
        }
        else
        {
            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 11.31** Warten auf Benachrichtigung über Datenänderungen [/VerschiedeneDemos/ADONET/MSSQL\_Notifikationen.vb]

Das obige Beispiel zeigt nur die Implementierung der Konsolenanwendung aus Abbildung 11.14. Aus Platzgründen ist die mehrschichtige Lösung mit einem Windows Forms-GridView-Steuerelement hier nicht abgedruckt. Sie finden diese in den herunterladbaren Beispielen zu diesem Buch [*WWWings\_WindowsUI\_CS/Fenster/F\_Flugliste\_MitAktualisierung.vb*].

**TIPP** Zu beachten ist, dass Datenänderungsbenachrichtigungen zu einem erhöhten Verarbeitungsaufwand auf dem SQL Server führen. Jede Änderung an einer Tabelle wird aufwendiger, weil zunächst alle Abonnements auf dieser Tabelle geprüft werden und ggf. Benachrichtigungen ausgelöst werden müssen. Datenänderungsbenachrichtigungen sollten daher nur für Szenarien genutzt werden, in denen Daten häufig gelesen, aber selten geändert werden. Außerdem kann der Aufwand für Datenänderungsbenachrichtigungen auf dem Server reduziert werden, indem man die SQL-Befehle für die Benachrichtigungen auf einer Tabelle möglichst gleich aufbaut (Abfrage über gleiche Spalten).

## Massenkopieren (Bulkcopy/Bulkimport)

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

**HINWEIS** Die Massenkopierfunktion ist auch als *Bulkcopy* oder *Bulkimport* bekannt.

**TIPP** Das Einfügen von Datensätzen ist mit Bulkimport um ein Vielfaches schneller als das Absenden von einzelnen INSERT-Befehlen mit dem Command-Objekt bzw. als alle darauf aufbauenden Mechanismen im DataSet und ADO.NET Entity Framework.

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

- *DataTable*
- ein Array von *DataRow*-Objekten
- einen *DataReader*
- Ein Objekt, das *IDataReader* implementiert

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

### Vorgehensweise

Bei der Instanziierung der Klasse *SqBulkCopy* 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 *SqBulkCopy* eine Abbildung anhand der Position der Spalten. Eine Abbildung kann explizit durch die *ColumnMappings*-Objektmenge definiert werden, beispielsweise

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

**HINWEIS** Die ColumnMappings-Klasse unterscheidet 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 von SQL Server 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=Server01\sqlexpress";
    const string CS_Ziel = @"Integrated Security=SSPI;Persist Security Info=False;" +
        "Initial Catalog=WorldWideWings;Data Source=Server01\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 11.32** Einfaches Beispiel für eine Massenkopie zwischen zwei Microsoft SQL Servern [/VerschiedeneDemos/ADONET/BulkImport]

### 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.OLEDB.4.0';Data Source='D:\WorldWideWings.mdb'";
    const string CS_Ziel = @"Integrated Security=SSPI;Persist Security Info=False;" +
        "Initial Catalog=WorldWideWings;Data Source=Server01\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 11.33** Beispiel für eine Massenkopie zwischen einer Access-Datenbank und einem Microsoft SQL Server [/VerschiedeneDemos/ADONET/BulkImport]

### 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 erzeugt.

```
// Massenkopie zwischen SQL Server-Datenbanken mit Optionen
public void SQLToSQL_Erweitert()
{
    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=Server01\sqlexpress";
    const string CS_Ziel = @"Integrated Security=SSPI;Persist Security Info=False;Initial
        Catalog=WorldWideWings;Data Source=Server01\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 11.34** Beispiel für eine Massenkopie mit Fortschrittsanzeige zwischen zwei Microsoft SQL Servern [/VerschiedeneDemos/ADONET/BulkImport]

## Providerstatistiken

ADO.NET (seit Version 2.0) unterstützt für Datenprovider eine Statistikfunktion 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.

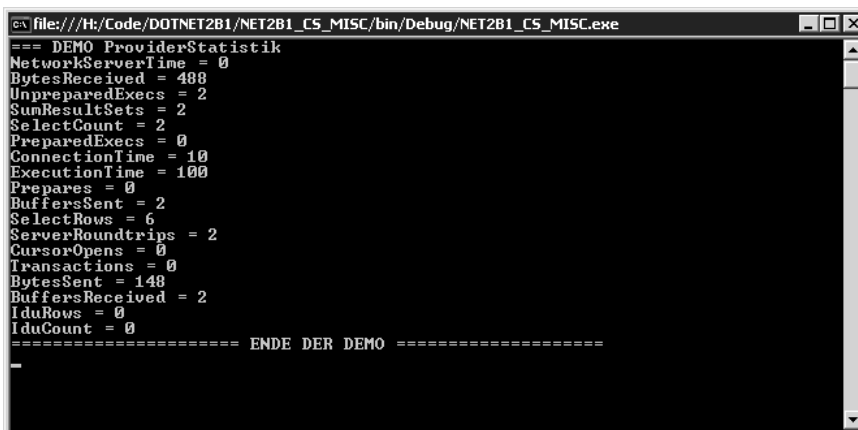
---

## Beispiel

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

```
public static void run()
{
    Demo.Out("=== DEMO Provider-Statistik");
    const string CONNSTRING = "Integrated Security=SSPI;Persist Security Info=False;Asynchronous
Processing=true;Initial Catalog=itvisions;Data Source=Server01\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 11.35** Ausgabe der Nutzungsstatistik [/VerschiedeneDemos/ADONET/Statistik]



**Abbildung 11.15** Beispiel für eine Statistikausgabe nach der Ausführung von zwei SELECT-Befehlen



# Datenbankschema auslesen

Das Schema-API von ADO.NET (seit Version 2.0) besteht aus einer einzigen Methode: `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-Arrays erlaubt die Angabe eines Filters.

Dabei gibt es fünf allgemeine Auflistungen, die durch die Aufzählung `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 unterstützte 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 11.36** Nutzung des ADO.NET Schema API [/VerschiedeneDemos/ADONET/SchemaAPI]

## 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 11.37**   Eingeschränkte Suche

## Zusatzdienste für ADO.NET

In diesem Kapitel sind noch zwei Zusatzdienste erwähnt, die im Namen auch *ADO.NET* tragen, aber nicht zum Kern des .NET Framework gehören.

## ADO.NET Data Services

ADO.NET Data Services heißen inzwischen WCF Data Services und werden im Kapitel 14 zu »Windows Communication Foundation (WCF) 4.0« beschrieben.

# ADO.NET Synchronization Services

Die ADO.NET Synchronization Services (alias *Microsoft Synchronization Services for ADO.NET*) sind eine Programmierschnittstelle zur Synchronisierung von Daten, insbesondere für Systeme, die nicht ständig, sondern nur gelegentlich miteinander verbunden sind. Client ist Microsoft SQL Server Compact. Server ist eine beliebige Datenbank oder ein Dienst.

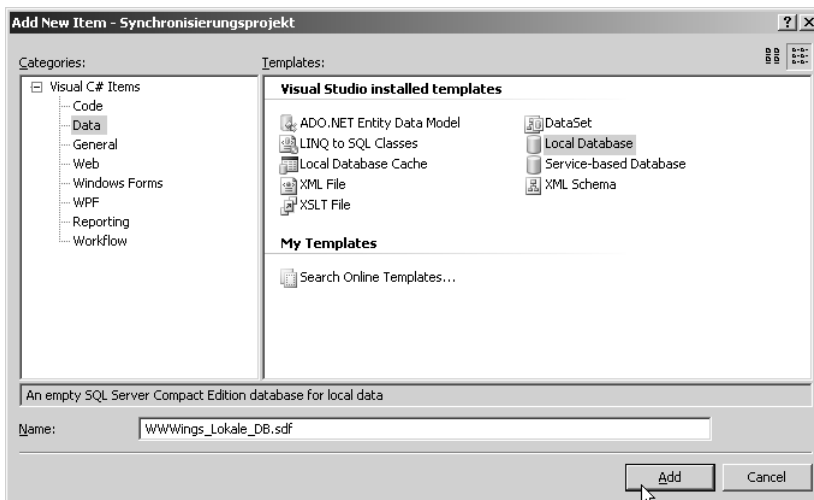
Die ADO.NET Synchronization Services wurden mit Microsoft SQL Server Compact 3.5 eingeführt (sie sind also nicht offizieller Bestandteil des .NET Framework) und sind realisiert im Namensraum `Microsoft.Synchronization.Data` (Assemblys *Microsoft.Synchronization.Data.dll*, *Microsoft.Synchronization.Data.Client.dll* und *Microsoft.Synchronization.Data.Server.dll*).

## HINWEIS

Microsoft SQL Server Compact ist eine stark funktionsreduzierte Version des Microsoft SQL Server zum Aufbau eines lokalen Datenspeichers. Der Zugriff auf eine Compact-Datenbank erfolgt über Programmierschnittstellen direkt auf eine Datenbankdatei (.sdf). SDF-Dateien sind auf 256 gleichzeitige Verbindungen und eine Maximalgröße von vier Gigabyte begrenzt. Die Funktionsreduzierung gilt sowohl für T-SQL als auch für den Zugriff via ADO.NET. Außerdem fehlt die Multi-User-Fähigkeit. Früherer Name war *SQL Server Everywhere*. SQL Server Compact ist kostenlos und wird zusammen mit Visual Studio ausgeliefert.

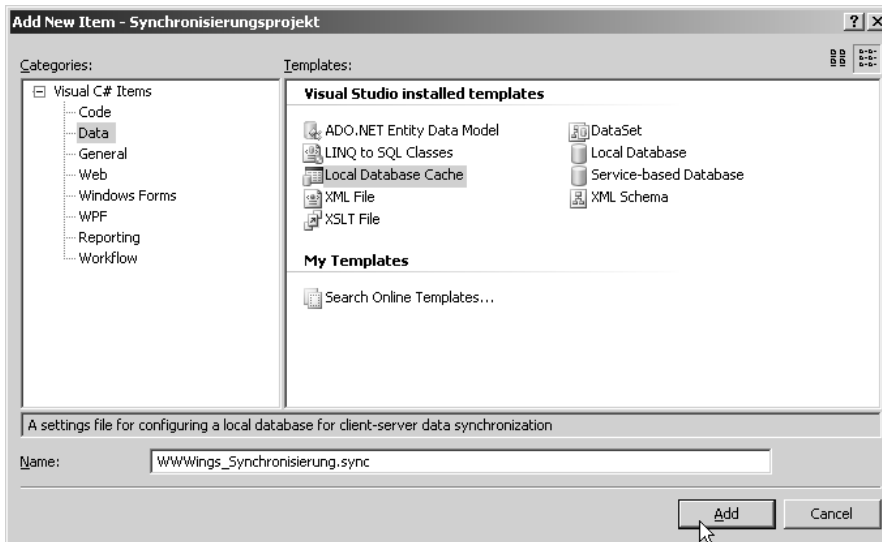
## Erste Schritte

Fügen Sie einem bestehenden Visual Studio-Projekt ein Element vom Typ *Local Database* hinzu. Dadurch wird dem Projekt eine Datenbankdatei im Format von Microsoft SQL Server Compact (Dateinamenserweiterung .sdf) hinzugefügt. Außerdem wird ein (leeres) typisiertes DataSet erzeugt.



**Abbildung 11.16** Anlegen einer SQL Server Compact-Datenbank

Fügen Sie einem bestehenden Visual Studio-Projekt ein Element vom Typ *Local Database Cache* hinzu.



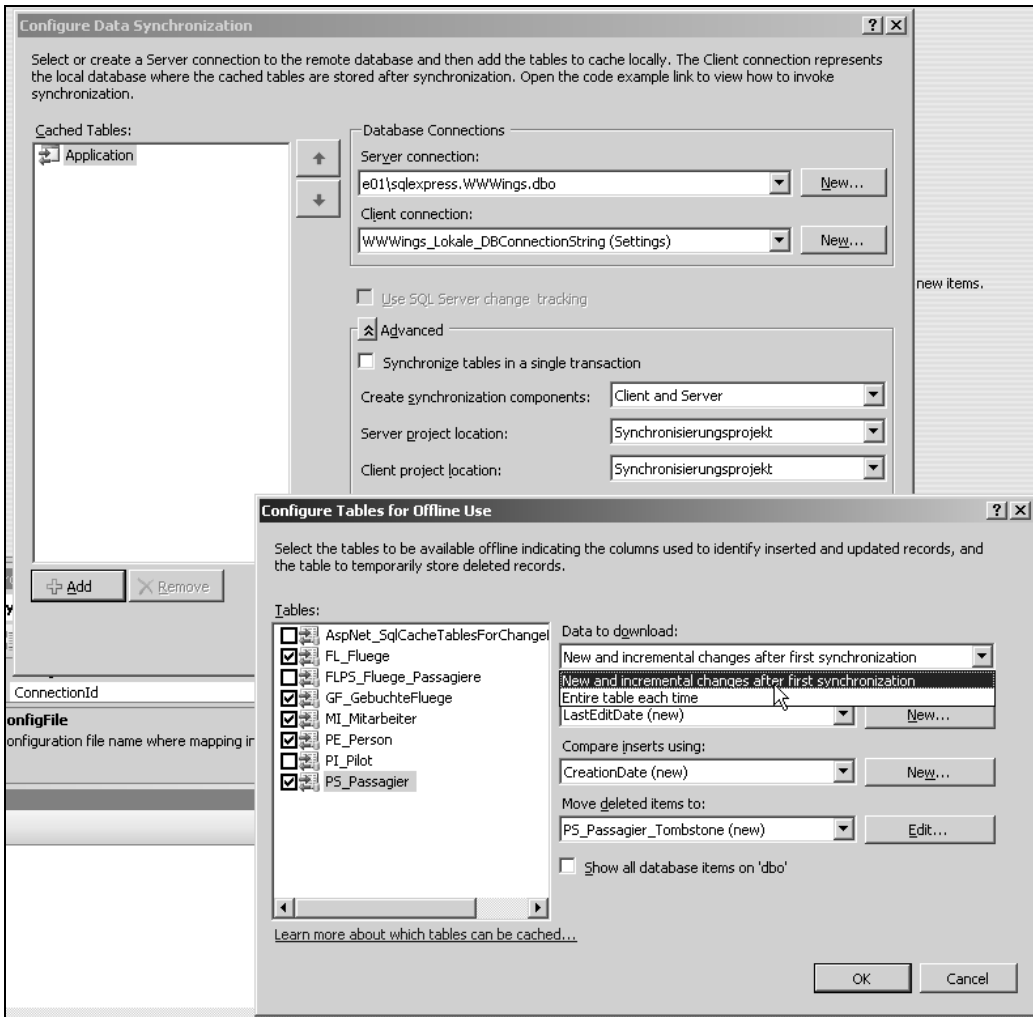
**Abbildung 11.17** Anlegen einer .sync-Datei

Danach kann man in einem Assistenten (*Configure Data Synchronization*) die Synchronisierungspartner sowie die zu synchronisierenden Tabellen und die Synchronisierungsart wählen.

---

**ACHTUNG** Man kann nur Tabellen synchronisieren, die einen Primärschlüssel haben und ausschließlich Datentypen, die es auch bei SQL Server Compact gibt. Außerdem dürfen Namen maximal 118 Zeichen lang sein.

---



**Abbildung 11.18** Konfiguration der Synchronisierungspartner, der zu synchronisierenden Tabellen und der Synchronisierungsart

Der Synchronisierungsassistent richtet die beiden beteiligten Datenbanken für die Synchronisierung ein. Je nach gewählten Optionen werden dabei auch neue Spalten in die bestehenden Tabellen eingefügt (z.B. CreationDate). Außerdem werden Trigger ergänzt sowie – optional – so genannte *Grabstein-Tabellen* (»Tombstone«) zum Merken gelöschter Zeilen. Weiterhin generiert der Assistent einen Synchronisierungsagenten. Dies ist eine Klasse, die von Microsoft.Synchronization.SyncAgent erbt und alle notwendigen SQL-Befehle für die Synchronisierung sowie deren Ausführung per ADO.NET kapselt.

Die Synchronisierung kann dann sehr einfach über die Instanziierung dieser Klasse und den anschließenden Aufruf der Methode Synchronize() durchgeführt werden. Auf Wunsch kann man sich über die Ereignisse SessionProgress() und StateChanged() über den Fortgang informieren lassen. Die Methode Synchronize() arbeitet synchron (d.h. nicht in einem eigenen Thread) und liefert als Rückgabe ein Objekt des Typs SyncStatistics, das Informationen über die synchronisierten Daten liefert.

## Beispiel

Das folgende Beispiel zeigt das Starten und Überwachen der Synchronisierung in einer Konsolenanwendung.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Synchronisierung\n");

        // Vorher
        Console.WriteLine("Lokaler Datenbankinhalt vorher:");
        GetFluege();

        Console.WriteLine("Starte Synchronisierung...");
        // Einrichten
        WWWings_SynchronisierungSyncAgent syncAgent = new WWWings_SynchronisierungSyncAgent();

        syncAgent.SessionProgress += new
        EventHandler<Microsoft.Synchronization.SessionProgressEventArgs>(syncAgent_SessionProgress);

        syncAgent.StateChanged += new
        EventHandler<Microsoft.Synchronization.SessionStateChangedEventArgs>(syncAgent_StateChanged);

        // Starten
        Microsoft.Synchronization.Data.SyncStatistics syncStats = syncAgent.Synchronize();

        // Ergebnis
        Console.WriteLine("Synchronisierung beendet:");
        Console.WriteLine("Datensätze Heruntergeladen OK: " + syncStats.DownloadChangesApplied);
        Console.WriteLine("Datensätze Heruntergeladen Fehler: " + syncStats.DownloadChangesFailed);
        Console.WriteLine("Datensätze Heraufgeladen OK: " + syncStats.UploadChangesApplied);
        Console.WriteLine("Datensätze Heraufgeladen Fehler: " + syncStats.UploadChangesFailed);

        Console.WriteLine("Datensätze Dauer in Millisekunden: " + (syncStats.SyncCompleteTime -
        syncStats.SyncStartTime).TotalMilliseconds);

        // Nachher
        GetFluege();

        Console.ReadLine();
    }

    /// <summary>
    /// Ereignis: Zustandsänderung bei der Synchronisierung
    /// </summary>
    static void syncAgent_StateChanged(object sender,
    Microsoft.Synchronization.SessionStateChangedEventArgs e)
    {
        Console.WriteLine("Zustand: " + e.SessionState.ToString());
    }

    /// <summary>
    /// Ereignis: Fortschrittsanzeige
    /// </summary>
}
```

```
static void syncAgent_SessionProgress(object sender,
Microsoft.Synchronization.SessionProgressEventArgs e)
{
    Console.WriteLine("Fortschritt: " + e.PercentCompleted);
}

/// <summary>
/// Ausgabe des lokalen Datenbestandes
/// </summary>
private static void GetFluege()
{
    WWings_Lokale_DBDataSet ds = new WWings_Lokale_DBDataSet();
    WWings_Lokale_DBDataSetTableAdapters.FL_FluegeTableAdapter ta = new
Synchronisierungsprojekt.WWings_Lokale_DBDataSetTableAdapters.FL_FluegeTableAdapter();
    ta.Fill(ds.FL_Fluege);

    Console.WriteLine("Anzahl lokaler Datensätze: " + ds.FL_Fluege.Rows.Count);
}
}
```

**Listing 11.38** Durchführen einer Synchronisierung

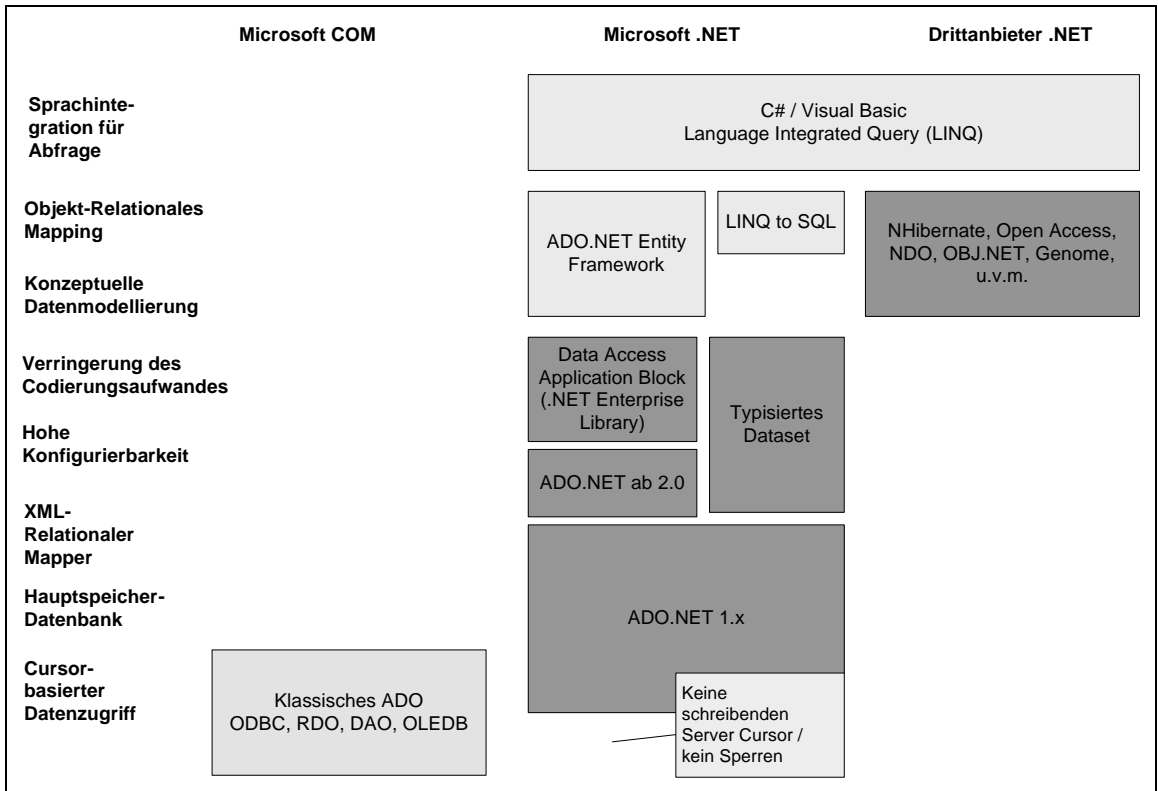
**HINWEIS** Eine weitere Betrachtung dieses Themas ist hier in diesem Buch aus Platzgründen nicht möglich. Eine empfehlenswerte Website zu diesem Thema ist [SYNC01].

## Positionierung von ADO.NET und Ausblick

Die früheren Datenbankzugriffsschnittstellen (ODBC, OLEDB, RDO, DAO, ADO) boten einen rein cursor-basierten Datenzugriff. ADO.NET verfügt durch die In-Memory-Datenbank (DataSets) und die typisierten DataSets über ein höheres Abstraktionsniveau, verzichtet aber auf den in den klassischen Datenbankzugriffsschnittstellen vorhandenen Schreibzugriff mit einem serverseitigen Cursor.

Einige Mechanismen (z. B. die Handhabung der Datenadapter) sind in ADO.NET umständlich und veranlassen jeden Entwickler schnell dazu, sich eigene Hilfsroutinen für wiederkehrende Codezeilen zu schreiben. Microsoft selbst bietet inzwischen im Rahmen der .NET Enterprise Library eine Vereinfachung zu ADO.NET in Form des so genannten *Data Access Application Block (DAAB)* an. Eine andere Abstraktion ist das typisierte DataSet (hierbei erzeugt Visual Studio eine Wrapper-Klasse, die von DataSet abgeleitet ist).

Ein echtes Objektrelationales Mapping offeriert das typisierte DataSet nicht. Dieses Feld überließ Microsoft bis .NET 3.5 den Drittanbietern. ORM-Werkzeuge für .NET sind inzwischen sehr zahlreich sowohl im kommerziellen Bereich als auch im kostenlosen Umfeld vorzufinden. Viele Implementierungen sind Portierungen aus dem Java-Umfeld, z. B. Vanatec Open Access (VOA) und das Open-Source-Projekt NHibernate. Eine Liste der .NET-ORM-Werkzeuge findet der geneigte Leser unter [DOTNET02]. Seit .NET 3.5 gibt es LIQN to SQL und das ADO.NET Entity Framework (siehe Kapitel »Objektrelationales Mapping (ORM) mit ADO.NET Entity Framework«)



**Abbildung 11.19** Positionierung von ADO.NET im Vergleich zu anderen heutigen und zukünftigen Datenbankschnittstellen