25 ADO.NET-Klassen (SqlClient)

Dieses und das folgende Kapitel beschreiben die wichtigsten ADO.NET-Klassen und geben Beispiele für ihre Anwendung und ihr Zusammenspiel. Die Kapitel sollen Ihnen dabei helfen, sich in der ADO.NET-Klassenhierarchie besser zurechtzufinden und selbst Code zu entwickeln, um typische Datenbankaufgaben durchzuführen. Dabei gilt das Motto *code only*, d.h., Sie werden hier keine Informationen zum Einsatz von Steuerelementen oder der Datenbankwerkzeuge der Entwicklungsumgebung finden.

Widerstehen Sie der Versuchung, gleich zu Kapitel 28 weiterzublättern! Ich weiß, Grundlagenkapitel sind immer ein wenig trocken; aber ohne diese Grundlagen wird es Ihnen nicht gelingen, Datenbanksteuerelemente oder die von den Visual Data Tools erzeugten Klassen erfolgreich einsetzen!

> > > HINWEIS

Über ADO.NET ließen sich Tausende von Seiten füllen. So viel Platz ist hier aber nicht. Deswegen beschränke ich mich in diesem Buch auf die am häufigsten eingesetzten ADO.NET-Klassen und auf den SQL Server als Datenbanksystem.

25.1 ADO.NET-Klassenübersicht

Wenn in diesem Buch von ADO.NET die Rede ist, ist eigentlich die Bibliothek *System.Data* gemeint. Damit Sie ADO.NET-Anwendungen entwickeln können, muss Ihr Projekt Referenzen auf diese Bibliothek sowie auf *System.Xml* enthalten, was standardmäßig bei den meisten Projekttypen der Fall ist. (*System.Data* verwendet intern *System.Xml*-Klassen. Deswegen ist auch die *System.Xml*-Bibliothek erforderlich.)

Grundsätzlich sieht ADO.NET eine Trennung zwischen dem sogenannten Datenprovider und den Klassen zur Verarbeitung der Daten vor (siehe auch Abbildung 25.1):

- **Kommunikation mit der Datenbank (Provider):** Der Datenprovider ist die eigentliche Schnittstelle zur Datenbank. Mit den Klassen des Providers stellen Sie die Verbindung zur Datenbank her, führen SQL-Kommandos aus und steuern Transaktionen. Die Objekte des Datenproviders stehen in direkter Verbindung zur Datenbank, deswegen spricht man oft vom *connected layer* oder von *connected objects*.

  Es gibt verschiedene Implementierungen für den Provider. Die ADO.NET-Bibliothek *System.Data* enthält bereits vier:

Sql	für den SQL Server, Namensraum *System.Data.SqlClient*
OleDb	für Jet und andere Datenbanken mit OLE-DB-Treiber
Odbc	für alle Datenbanken mit ODBC-Treiber
OracleClient	von Microsoft entwickelter Treiber für Oracle

  Neu im .NET-Framework 3.5 ist die Bibliothek *System.Data.SqlServerCe*, die den Provider für die SQL Server Compact Edition enthält:

  SqlCe für die SQL Server Compact Edition, Namensraum *System.Data.SqlServerCe*

  Daneben stellen manche Datenbankhersteller eigene .NET-Provider für Ihre Datenbanksysteme zur Verfügung, z.B. *MySql* (für MySQL) oder *Oracle* (firmeneigener Oracle-Treiber). Je nachdem, welchen Provider Sie einsetzen, heißt die *Connection*-Klasse *SqlConnection*, *SqlCeConnection*, *OdbcConnection*, *MySqlConnection*, *OracleConnection* etc. Analog gilt das für alle Klassen des Datenproviders.

  Dieses Kapitel konzentriert sich auf die wichtigsten *SqlXxx*-Klassen aus dem *SqlClient*-Namensraum.

- **Datenverarbeitung (Konsument):** Wurden die Daten einmal aus der Datenbank gelesen, können sie innerhalb eines *DataSet*-Objekts lokal bearbeitet werden. Ein *DataSet* stellt zusammen mit zahlreichen weiteren Klassen (*DataTable*, *DataRow* etc.) gewissermaßen ein komplettes, im RAM befindliches Datenbanksystem dar. Die *DataSet*-Objekte stehen nicht in direkter Verbindung zur Datenbank (daher die Bezeichnung *disconnected layer/objects*). Der Transfer von Daten zwischen Datenbank und *DataSet* erfolgt durch *DataAdapter*-Objekte (wobei für jede Tabelle bzw. für jedes *SELECT*-Ergebnis ein eigener *DataAdapter* erforderlich ist).

  Die wichtigsten Klassen rund um *DataSets* und *DataTables* stehen im Mittelpunkt von Kapitel 26. Dort erfahren Sie, wie Sie lokale Daten unabhängig vom zugrunde liegenden Datenbanksystem verarbeiten.

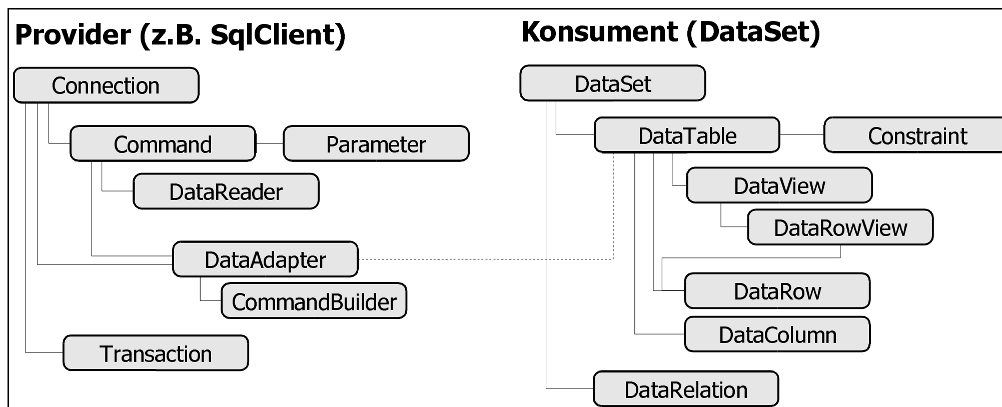

Abbildung 25.1: Die wichtigsten ADO.NET-Klassen

Theoretisch wären wenige Provider-Objekte ausreichend, um jede beliebige Datenbankoperation durchzuführen. Sie können damit SQL-Kommandos ausführen (*Command*) sowie die Ergebnisse eines *SELECT*-Kommandos auslesen (*DataReader*).

Dennoch spielen die *DataSet*-Objekte in der Praxis meist eine größere Rolle als die Provider-Objekte. Der Grund: Sie vereinfachen viele Bearbeitungsschritte und stellen die Schnittstelle zu den Steuerelementen dar. Wenn Sie beispielsweise ein *SELECT*-Ergebnis in einem *DataGridView*-Steuerelement darstellen möchten, müssen die Ergebnisse als *DataTable* innerhalb eines *DataSets* zur Verfügung stehen.

Die Klassen der Bibliothek *System.Data* sind über mehrere Namensräume verteilt (siehe die folgende Tabelle). Aus diesem Grund ist es zumeist zweckmäßig, am Beginn des Programmcodes die Zeilen *Imports System.Data* und *Imports System.Data.SqlClient* einzufügen, um den Tippaufwand für allgemeine Klassen sowie für die Klassen des eingesetzten Providers zu minimieren. Dieses Kapitel beschreibt die wichtigsten *SqlClient*-Provider. In Kapitel 26 folgen *DataSet*-Klassen und in Kapitel 33 die *SqlCe*-Klassen für die SQL Server Compact Edition.

Namensraum	**Inhalt**
System.Data	allgemeine Klassen zur Datenverarbeitung (*DataSet* etc.)
System.Data.Odbc	Klassen des Providers für ODBC-Datenquellen
System.Data.OleDb	Klassen des Providers für OLEDB-Datenquellen
System.Data.OracleClient	Klassen des Providers für Oracle
System.Data.SqlClient	Klassen des Providers für den SQL Server

ADO.NET versus ADO

ADO.NET hat so gut wie nichts mit dem alten ADO zu tun, mit dem viele VB6-Programmierer vertraut sind. ADO.NET ist eine vollkommen neue Datenbankbibliothek mit anderen Klassen und Konzepten. Insbesondere fehlt das vertraute *RecordSet*. Stattdessen können Sie *SELECT*-Ergebnisse nun mit einem *DataReader* auslesen (*read only, forward only*) oder als *DataTable* umfassend bearbeiten (*read-write, random access*).

Ein fundamentaler Unterschied zwischen ADO und ADO.NET besteht darin, dass ADO.NET keinen *server side cursor* kennt. Wenn Sie Daten bearbeiten bzw. verändern, erfolgt dies immer *offline* in einem *DataSet* bzw. in einer *DataTable* ohne direkte Verbindung zur Datenbank. Änderungen werden erst durch das explizite Ausführen einer *Update*-Methode ausgeführt. Dieses Konzept (Schlagwort *disconnected datasets*) bietet manche Vorteile, ist aber auch mit Einschränkungen verbunden und verlangt ein vollständiges Umdenken bei der Entwicklung von Datenbankanwendungen.

ADO stand ursprünglich für *ActiveX Data Objects*. ADO.NET basiert aber auf dem .NET-Framework und verwendet weder ActiveX noch dessen zugrunde liegende Komponentenarchitektur. Insofern ist die Abkürzung ADO.NET eigentlich irreführend.

25.2 SqlConnection (Verbindung herstellen)

Viele Wege führen nach Rom, und beinahe ebenso viele Wege gibt es, um eine Verbindung zu einer Datenbank herzustellen. An dieser Stelle beschränke ich mich auf den SQL Server als Datenbanksystem. Selbst in diesem Fall bestehen drei verschiedene Möglichkeiten, je nachdem, welchen Provider Sie einsetzen. Vorweg ein erster Überblick:

- **Verbindung mit dem SqlClient-Provider**

```
' Beispiel ADO.NET-classes\connection-variants
```

```
Imports System.Data.SqlClient
Dim sqlConn As SqlConnection
sqlConn = New SqlConnection("Server=localhost;Database=mylibrary;Integrated Security=True")
sqlConn.Open()
```

- **Verbindung mit dem OleDb-Provider**

```
Imports System.Data.OleDb
Dim oleConn As OleDbConnection
oleConn = New OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;" + _
  "Initial Catalog=mylibrary;Integrated Security=SSPI")
oleConn.Open()
```

Verbindung mit dem Odbc-Provider

```
Imports System.Data.Odbc
Dim odbcConn As OdbcConnection
odbcConn = New OdbcConnection( _
  "Driver={SQL Server};Server=localhost;Database=mylibrary;Trusted_connection=yes")
odbcConn.Open()
```

* * * TIPP

Je nach Netzwerk- und SQL-Server-Konfiguration müssen Sie bei den Varianten 2 und 3 statt localhost *den tatsächlichen Namen des Rechners angeben. Bei meinen Tests hat* localhost *nur funktioniert, wenn der SQL Server so konfiguriert wurde, dass Remote-Verbindungen via TCP/IP erlaubt waren.*

Der Verbindungsaufbau mit einer SqlConnection *erfolgt wesentlich schneller als bei den anderen zwei Varianten. Allein aus diesem Grund sollten Sie den* SqlClient*-Provider nach Möglichkeit vorziehen.*

Sie sehen, der Code sieht eigentlich immer gleich aus. Der einzige Unterschied ist der Aufbau der Zeichenkette, die die Verbindungsdaten enthält, also den Namen des Rechners, auf dem der SQL Server läuft, den Namen der gewünschten Datenbank etc. Jeder Provider sieht für diese Zeichenkette eine eigene Syntax vor, die in der Hilfe bei der Eigenschaft *ConnectionString* dokumentiert ist.

Welchen dieser drei Provider sollten Sie nun einsetzen? Die Antwort hängt davon ab, was Ihnen wichtiger ist: Geschwindigkeit und Funktionsreichtum oder Kompatibilität zu möglichst vielen Datenbanksystemen. Im ersten Fall sollten Sie sich für den *SqlClient*-Provider entscheiden, im zweiten Fall für die *OleDb*- oder *Odbc*-Varianten.

Diese *OleDb*- oder *Odbc*-Varianten haben den Vorteil, dass es relativ leicht ist, das gesamte Programm später auf ein anderes Datenbanksystem umzustellen. Im Idealfall reicht es aus, einfach die Zeichenkette für den Verbindungsaufbau auszutauschen. Allerdings tritt dieser Idealfall in der Praxis selten ein: Verschiedene Datenbanksysteme unterscheiden sich in der Regel durch diverse Zusatzfunktionen, eigene Erweiterungen an der SQL-Syntax, Abweichungen bei der Unterstützung von *Stored Procedures* etc. Aus diesem Grund ist die Umstellung auf ein anderes Datenbanksystem zumeist mit riesigem Aufwand verbunden, wobei die Anpassungen im Client-Code nur ein Aspekt sind.

Mit anderen Worten: Der Vorteil der höheren Kompatibilität reicht nicht sehr weit. Daher empfehle ich Ihnen, den für Ihr Datenbanksystem am besten geeigneten Provider einzusetzen – also *SqlClient* für den SQL Server. Alle weiteren Beispiele verwenden diesen Provider.

Aufbau der Zeichenkette mit den Verbindungsdaten für den SqlClient-Provider

An den Konstruktor der ***SqlConnection***-Klasse übergeben Sie eine Verbindungszeichenkette. Sobald das Objekt existiert, kann die Zeichenkette aus der ***ConnectionString***-Eigenschaft gelesen werden. Die Teile der Zeichenkette werden durch Strichpunkte getrennt. Die wichtigsten Schlüsselwörter sind in der folgenden Tabelle zusammengefasst.

ConnectionString-Schlüsselwörter	Bedeutung
Data Source oder *Server* oder *Address*	gibt den Namen des Rechners an, auf dem der SQL Server läuft. Wenn der SQL Server auf demselben Rechner wie das VB-Programm ausgeführt wird, lautet die korrekte Einstellung *localhost* oder *(local)* oder ist einfach ein Punkt. Wenn der SQL Server nicht als Standardinstanz installiert ist (z.B. bei Express-Installationen), muss auch der Instanzname angegeben werden. Die Schreibweise lautet *hostname\instanzname*. Lokale Instanzen können in der Kurzschreibweise *.\instanzname* angesprochen werden. Der Instanzname für Express-Installationen lautet standardmäßig *sqlexpress*. Um also eine Verbindung zur lokalen SQL Server Express Edition herzustellen, geben Sie *.\sqlexpress* an.
Initial Catalog oder *Database*	gibt den Namen der Datenbank an.
Connect Timeout oder *Connection Timeout*	gibt an, wie lange der Verbindungsaufbau dauern darf, bevor ein Fehler ausgelöst wird (standardmäßig 15 Sekunden).
Integrated Security oder *Trusted_Connection*	gibt an, ob die Authentifizierung beim SQL Server durch das Windows-interne Authentifizierungssystem erfolgt (Einstellungen *yes* oder *true* oder *sspi*) oder durch die explizite Angabe von Loginname und Passwort (*no* oder *false*). Mehr Details zur Authentifizierung finden Sie in Abschnitt 24.4. Achtung, die Standardeinstellung für diese Option lautet *no*!
User ID	gibt den Loginnamen für die Authentifizierung beim SQL Server an (nur für *Integrated Security=no*). Der Name kann in einfache oder doppelte Anführungszeichen gestellt werden, also z.B. User ID="name".
Password oder *Pwd*	gibt das Passwort an (nur für *Integrated Security=no*). Auch das Passwort kann in Anführungszeichen gestellt werden.
Persist Security Info	gibt an, ob Benutzername und Passwort nach der Authentifizierung weiterhin in der *ConnectionString*-Eigenschaft gespeichert werden sollen. Da dies ein Sicherheitsrisiko ist, lautet die Standardeinstellung *no* und sollte nicht verändert werden. Die Option ist nur für *Integrated Security=no* relevant.
MultipleActiveResultSets	gibt an, ob die Verbindung mehrere offene SQL-Kommandos erlaubt (normalerweise *False*). Die Einstellung *True* setzt zumindest Version 2005 des SQL Servers voraus. Erst diese Version lässt es zu, gleichzeitig mehrere Ergebnisse zu verarbeiten (*MARS = Multiple Active Result Sets*).

ConnectionString-Schlüsselwörter	Bedeutung
AttachDBFilename oder *Initial File Name*	gibt den Dateinamen einer Datenbankdatei an. Das kann sinnvoll sein, wenn Sie eine SQL-Server-Datenbankdatei zusammen mit einer SQL-Server-Express-Installation weitergeben. Dieses Anwendungsszenario wird in Abschnitt 25.3 beschrieben.
User Instance	gibt an, ob die Datenbank in einer eigenen Benutzerinstanz des SQL Servers geöffnet werden soll. Die Einstellung *True* ist nur bei der SQL Server 2005 Express Edition zulässig und nur in Kombination mit *AttachDBFilename* zweckmäßig. Weitere Details zu *User Instance* folgen ebenfalls in Abschnitt 25.3.
Asynchronous Processing oder *Async*	gibt an, ob die Verbindung die asynchrone Verarbeitung von SQL-Kommandos unterstützt (*True*/*False*). Standardmäßig gilt *False*. Der Parameter sollte nur auf *True* gestellt werden, wenn Sie tatsächlich asynchrone Kommandos ausführen wollen, wie dies in Abschnitt 27.10 beschrieben wird.
Net	gibt an, welche Netzwerkbibliothek zur Kommunikation mit dem SQL Server verwendet werden soll. Zur Auswahl stehen unter anderem *dbmssocn* (TCP/IP), *dbmslpcn* (*shared memory*) und dbnmpntw (*names pipes*). Normalerweise entscheidet sich ADO.NET selbst für eine geeignete Bibliothek, sodass dieser Parameter nicht angegeben werden muss.

ConnectionString mit dem SqlConnectionStringBuilder zusammenfügen

Sie können die Verbindungszeichenkette wahlweise selbst zusammensetzen oder sich dabei vom ***SqlConnectionStringBuilder*** helfen lassen. Diese Klasse stellt für alle *ConnectionString*-Schlüsselwörter Eigenschaften zur Verfügung, von denen Sie nur die einstellen, die Sie benötigen. Anschließend liefert die Eigenschaft *ConnectionString* die erforderliche Verbindungszeichenkette.

```
Dim csb As New SqlConnectionStringBuilder()
csb.DataSource = "."
csb.InitialCatalog = "mylibrary"
csb.IntegratedSecurity = False
csb.UserID = "sa"
csb.Password = "xxxx"
conn = New SqlConnection(csb.ConnectionString)
```

Der wesentliche Pluspunkt des *SqlConnectionStringBuilders* besteht darin, dass er mit Sonderzeichen in *User Id* und *Password* sehr vorsichtig umgeht. Wenn das Passwort beispielsweise die Zeichen ; oder ' oder " oder = enthält, wird es automatisch in Anführungszeichen gestellt. Das ist besonders dann wichtig, wenn der Benutzername und das Passwort aus einem Logindialog stammen. Hier besteht nämlich die Gefahr, dass Hacker im Login- oder Passwortfeld zusätzliche Parameter angeben.

Beispielsweise könnte ein Anwender im Loginfeld den Text *loginname;IntegratedSecurity= False* eingeben. Wird diese Eingabe direkt in den *ConnectionString* eingefügt, würde dadurch der Parameter *IntegratedSecurity* verändert. Das kann zu Sicherheitsproblemen führen. Der *SqlConnectionStringBuilder* vermeidet dieses Problem.

Logindialog mit Fehlerabsicherung

Beim Versuch, eine Verbindung zum SQL Server herzustellen, kann es aus den verschiedensten Gründen zu Fehlern kommen: Die Authentifizierung schlägt fehl oder der SQL Server läuft nicht, es gibt Netzwerkprobleme (falls Client und Server auf unterschiedlichen Rechnern laufen) etc. Es ist daher unbedingt erforderlich, dass Sie den Authentifizierungscode absichern und gegebenenfalls eine verständliche Fehlermeldung anzeigen und dem Anwender die Möglichkeit geben, einen neuen Authentifizierungsversuch durchzuführen.

Damit Sie das Rad nicht immer wieder neu erfinden müssen, enthält das Beispielprogramm ADO.NET-classes\connection-login Formular *Login-Form* (siehe Abbildung 25.2), das Sie in eigenen Projekten einsetzen können.

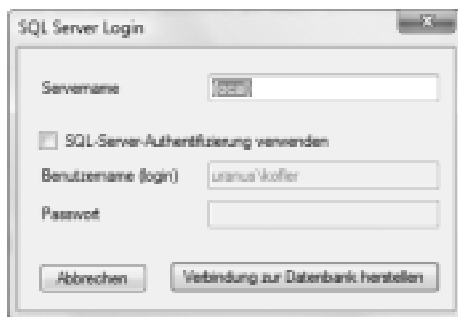

Abbildung 25.2: Loginformular

Die Anwendung des Formulars ist denkbar einfach: Sie stellen den gewünschten Datenbanknamen (Eigenschaft *databasename*) ein und rufen *ShowDialog()* auf. Wenn diese Methode *OK* zurückgibt, können Sie das *SqlConnection*-Objekt aus der Eigenschaft *connection* lesen.

```
' Beispiel ADO.NET-classes\connection-login\Form1.vb
Dim myconn As SqlConnection
Dim result As Windows.Forms.DialogResult
Dim loginform As Login_Form = My.Forms.Login_Form
loginform.databasename = "mylibrary"
result = loginform.ShowDialog()
```

```
If result = Windows.Forms.DialogResult.OK Then
  myconn = loginform.connection
Else
  ... keine Verbindung
End If
```

Die interessanteste Prozedur in Login-Form.vb ist die der Verbindungs-Buttons. Dort wird je nachdem, welches Authentifizierungsverfahren gewünscht ist, die Verbindungszeichenkette zusammengesetzt und versucht, die Verbindung herzustellen. Wenn das gelingt, wird der Dialog geschlossen, andernfalls wird eine Fehlermeldung angezeigt.

```
' Beispiel ADO.NET-classes\connection-login\Login-Form.vb

Private Sub ConnectButton_Click(...) Handles ConnectButton.Click
  Dim csb As New SqlConnectionStringBuilder()
  Try
    csb.DataSource = ServerTextBox.Text
    csb.InitialCatalog = databasename

    If SQLAuthCheckBox.Checked Then
      'SQL-Server-Authentifizierung
      csb.IntegratedSecurity = False
      csb.UserID = LoginTextBox.Text
      csb.Password = PasswordTextBox.Text
    Else
      'Windows-Authentifizierung
      csb.IntegratedSecurity = True
    End If

    connection = New SqlConnection(csb.ConnectionString)
    'während Verbindungsaufbau Sanduhr-Mauszeiger
    Cursor = Cursors.WaitCursor
    connection.Open()
    Cursor = Cursors.Default
    DialogResult = Windows.Forms.DialogResult.OK
    Close()

  Catch ex As Exception
    Cursor = Cursors.Default
    MsgBox("Fehler beim Verbindungsaufbau: " + ex.Message, MsgBoxStyle.Exclamation)
  End Try
End Sub
```

Wenn Sie möchten bzw. wenn es für Ihre Anwendung zweckmäßig ist, können Sie im Logindialog das Textfeld *ServerTextBox* durch ein Kombinationslistenfeld ersetzen, das Einträge für alle im Netz bzw. auf dem lokalen Rechner gefundenen Server-Instanzen enthält. Möglichkeiten zur Ermittlung dieser Liste zeigt der nächste Abschnitt auf.

Passwort für die SQL-Server-Authentifizierung ändern

Mit *SqlConnection.**ChangePassword*** kann das Passwort für die SQL-Server-Authentifizierung verändert werden. An die Methode muss ein *ConnectionString* sowie das neue Passwort übergeben werden. Der *ConnectionString* muss zumindest die Schlüsselwörter *User Id* und *Password* enthalten, um eine Authentifizierung des Benutzers zu ermöglichen. Es ist nicht erforderlich, dass bereits eine offene Verbindung zum Server existiert:

```
SqlConnection.ChangePassword("User Id=name;Pwd=altespasswort", "neuespasswort")
```

ChangePassword setzt zumindest Version 2005 des SQL Servers voraus.

Schema-Informationen des SQL Servers ermitteln

Normalerweise wissen Sie im Voraus, zu welcher Datenbank Sie eine Verbindung herstellen möchten, welche Tabellen diese Datenbank enthält etc. Wenn Sie aber allgemeine Administrationswerkzeuge entwickeln möchten, benötigen Sie Methoden, um die vom SQL Server verwalteten Objekte zu ermitteln. Genau diese Aufgabe erfüllt die Methode ***GetSchema*** der *SqlConnection*-Klasse.

Ohne Parameter liefert *GetSchema* ein *DataTable*-Objekt, das Auskunft über die zur Auswahl stehenden Aufzählungen z.B. *DataBases*, *Tables* (für die aktuelle Datenbank) oder *Procedures* gibt. Wenn Sie den Namen einer dieser Aufzählungen als Zeichenkettenparameter an *GetSchema* übergeben, liefert die Methode genauere Informationen zu dieser Aufzählung. Beispielsweise liefert *conn.GetSchema("databases")* eine Liste aller Datenbanken auf dem SQL Server.

Noch genauer können Sie den Informationsfluss durch den dritten Parameter steuern. Dieser erwartet ein *String*-Feld mit Filterausdrücken (*restrictions*). Welche Filter es gibt, hängt vom Typ der Aufzählung ab. Eine Tabelle mit allen Filtern für alle Aufzählungen liefert *GetSchema("restrictions")*. Beispielsweise können für die Aufzählung *Procedures* vier Filterausdrücke angegeben werden: *Catalog*, *Owner*, *Name* und *Type*. Für Filter, die Sie nicht nutzen möchten, übergeben Sie *Nothing*. Um die Liste aller Funktionen der aktuellen Datenbank zu ermitteln, gehen Sie so vor:

```
Dim dt As DataTable = conn.GetSchema("procedures", _
  New String() {Nothing, Nothing, Nothing, "Function"})
```

> > > HINWEIS

GetSchema ist nicht ausreichend, um wirklich alle Schemadaten einer Datenbank zu ermitteln. Noch mehr Informationenen liefern die direkte Auswertung der INFORMATION_SCHEMA*-Tabellen sowie die T-SQL-Funktion* COLUMNPROPERTY*. Die Online-Hilfe zum SQL-Server verrät weitere Details.*

Der in Abschnitt 33.4 vorgestellte Datenbankkonverter vom MDF- in das SDF-Format macht intensiven Gebrauch von der GetSchema*-Methode und setzt auch die oben erwähnten* INFORMATION_SCHEMA*-Tabellen und die* COLUMNPROPERTY*-Funktion ein.*

Verbindungen schließen

Eine Grundregel für den Umgang mit Verbindungen lautet: Sobald die Datenbankverbindung nicht mehr benötigt wird, sollte sie durch ***Close*** oder noch besser durch ***Dispose*** geschlossen werden. Verbindungen beanspruchen Ressourcen auf dem Server und sollten sparsam eingesetzt werden.

> > > HINWEIS

Es gibt im Internet lange Diskussionen darüber, ob Close *oder* Dispose *besser geeignet ist, um ein* SqlConnection*-Objekt endgültig zu schließen. Das Ergebnis:* Dispose *ruft intern auf jeden Fall* Close *auf, führt aber zusätzliche Arbeiten aus, um das Objekt endgültig aus dem Speicher zu entfernen. Wenn Sie viele Verbindungen öffnen und schließen, sind Sie mit* Dispose *auf der sicheren Seite. (Aber auch mit* Close *machen Sie nichts grundlegend falsch.)*

http://www.eggheadcafe.com/forumarchives/NETFrameworkADONET/Mar2006/post25804729.asp = http://tinyurl.com/2jtxlw

Wenn Sie ein Connection*-Objekt schließen und später wieder öffnen möchten (auch das ist möglich!), kommt nur* Close *in Frage.*

Wie soll sich nun aber ein Windows-Programm verhalten, das möglicherweise stundenlang läuft und in dieser Zeit immer wieder Daten abruft bzw. ändert? Es gibt zwei mögliche Strategien:

- **Verbindung offen lassen:** Sie öffnen zu Programmbeginn eine Verbindung, speichern diese in einer im gesamten Programm zugänglichen Variable und nutzen sie bis zum Programmende. Damit beansprucht das Programm während der gesamten Laufzeit eine Verbindung.
- **Verbindung für Einzelkommandos öffnen und wieder schließen:** Jedes Mal, wenn Ihr Programm eine Datenbankoperation durchführen möchte, öffnen Sie ein vorhandenes Verbindungsobjekt und schließen es anschließend wieder.

  Ein typisches interaktives Programm wartet ohnedies die meiste Zeit auf Benutzereingaben. Die Zeit für die Ausführung von Datenbankkommandos ist gering im Vergleich zur Gesamtlaufzeit, und nur während dieser Momente wird eine Verbindung beansprucht. Diesem Vorteil steht der Nachteil gegenüber, dass der wiederholte Aufbau einer Verbindung trotz *connection pooling* (siehe unten) ein wenig Zeit kostet.

  Diese Vorgehensweise wird durch einige Methoden der *SqlClient*-Klassen unterstützt, die mit einem *geschlossenen SqlConnection*-Objekt verbunden werden können (beispielsweise die Methode *SqlDataAdapter.Fill*): Bei der Ausführung solcher Methoden wird die geschlossene Verbindung kurzzeitig geöffnet und dann sofort wieder geschlossen.

Aus alter Gewohnheit bin ich eher ein Anhänger der ersten Strategie. Wenn der sparsame Umgang mit Ressourcen höchste Priorität hat, ist die zweite Variante aber eindeutig besser. Das gilt umso mehr, als oft gar kein Zusatzaufwand im Code erforderlich ist! Entfernen Sie einfach *conn.**Open*** aus Ihrem Programm, und testen Sie es dann. In vielen Fällen werden Sie feststellen, dass das Programm unverändert läuft, weil die diversen von Ihnen eingesetzten Methoden der *SqlClient*-Klassen die Verbindung nun automatisch öffnen und schließen.

ADO.NET-Verbindungsmanagement (connection pooling)

Der Aufbau einer neuen Verbindung kostet Zeit. Viele Programme (insbesondere Web-Applikationen) erstellen laufend neue Verbindungen, nutzen diese kurz und schließen sie dann wieder. Um solche Anwendungen so effizient wie möglich auszuführen, führt ADO.NET ein automatisches *connection pooling* durch: Das bedeutet, dass nicht mehr benötigte Datenbankverbindungen noch eine Weile offengehalten werden. Fordert nun ein Programm eine neue Verbindung mit exakt demselben *ConnectionString* an, wird die bestehende Verbindung neuerlich genutzt.

Normalerweise brauchen Sie sich um das *connection pooling* nicht zu kümmern. Es erfolgt transparent und automatisch. Für den Fall, dass Sie doch Einfluss auf die Speicherung der Verbindungen nehmen möchten, bietet ADO.NET zwei Möglichkeiten, den Pool explizit zu leeren: *SqlConnection.**ClearPool**(conn)* löscht alle Verbindungen aus dem Pool, die dieselbe Verbindungszeichenkette wie das Objekt *conn* haben. *SqlConnection.**ClearAllPools*** löscht sämtliche Verbindungen aus dem Pool. Weiters können Sie Interna des *connection poolings* durch Zusatzparameter in der Verbindungszeichenkette beeinflussen (siehe die Online-Hilfe zu *ConnectionString)*.

25.3 Verbindung zur SQL Server Express Edition

Grundsätzlich ist es für den Verbindungsaufbau gleichgültig, ob eine Datenbank von einer Vollversion oder von der Express Edition des SQL Servers verwaltet wird. Entscheidend ist nur, dass Sie in der *ConnectionString*-Zeichenkette den richtigen Namen für den Server angeben. Bei der Express Edition lautet dieser in der Regel *.\sqlexpress* bzw. *(local)\sqlexpress*. Dabei ist . bzw. *(local)* der Rechnername und *sqlexpress* der Name der SQL-Server-Instanz.

```
Dim sqlConn As SqlConnection
sqlConn = New SqlConnection( _
  "Server=.\sqlexpress;Database=mylibrary;Integrated Security=True")
sqlConn.Open()
```

Diese Art des Verbindungsaufbaus setzt allerdings voraus, dass die Datenbank dem SQL Server bereits bekannt ist (dass die Datenbank also mit dem Server verbunden bzw. *attached* ist). Während der Programmentwicklung ist diese Voraussetzung immer erfüllt.

Anders sieht es aus, wenn eine Anwendung, die aus einem Programm und einer Datenbank besteht, auf einem anderen Rechner installiert wird: Bevor die Datenbank auf dem Kundenrechner genutzt werden kann, muss ein Datenbankadministrator die neue Datenbankdatei mit der SQL Server Express Edition verbinden. Dieser Schritt setzt nicht nur einige Systemkenntnis voraus, sondern auch Administrationswerkzeuge (z.B. SQL Server Management Studio Express), die mit der SQL Server Express Edition nicht mitgeliefert werden.

Verbindung direkt zur Datenbankdatei herstellen (AttachDBFilename)

Da Datenbankanwendungen auf Basis der SQL Server Express Edition oft direkt vom Kunden (nicht von irgendwelchen Administratoren) installiert und genutzt werden, muss es einen einfacheren Weg geben. Die Lösung ist der zusätzliche Parameter ***AttachDBFilename***, der in die *ConnectionString*-Zeichenfolge eingebaut wird. Die folgenden Zeilen zeigen die übliche Anwendung dieses Parameters. (Statt *AttachDBFilename* können Sie den Dateinamen auch durch *Initial File Name* angeben.)

```
conn.ConnectionString = _
  "Server=.\sqlexpress;AttachDBFilename=|DataDirectory|\mylibrary.mdf;" + _
  "Integrated Security=True"
conn.Open()
```

An *AttachDBFilename* übergeben Sie den Dateinamen der *.mdf-Datenbankdatei. |*DataDirectory*| ist dabei eine Abkürzung für das Verzeichnis, in dem sich die *.exe-Datei befindet. Sie können auf |*DataDirectory*| auch verzichten, müssen dann aber den Verzeichnisnamen selbst angeben:

```
conn.ConnectionString = _
  "Server=.\sqlexpress;AttachDBFilename=" + CurDir() + "\mylibrary.mdf;" + _
  "Integrated Security=True"
conn.Open()
```

Im selben Verzeichnis wie die *.mdf-Datei muss sich auch die Logging-Datei befinden (*dbname*_-log.ldf). Der SQL Server benötigt diese Datei zur internen Protokollierung von Datenbanktransaktionen. Wenn die *.ldf-Datei beim ersten Verbindungsaufbau fehlt, ist das kein Problem: Der SQL Server erzeugt einfach eine neue Logging-Datei. Schlimmer ist es, wenn das Programm zum wiederholten Mal ausgeführt wird: Der SQL Server kennt die Datenbank jetzt schon und weiß, dass es eine Logging-Datei geben müsste. Ist diese verschwunden, wird der Verbindungsaufbau mit diversen Fehlermeldungen abgebrochen, wobei aus den Fehlermeldungen die wahre Ursache des Problems nicht hervorgeht.

Den ***Database***-Parameter in der Verbindungszeichenfolge können Sie entweder ganz weglassen oder mit einer leeren Zeichenkette einstellen. Explizit nicht zulässig ist die Nennung des Datenbanknamens (also *Database=mylibrary*). Das funktioniert zwar manchmal, führt aber zu Problemen, wenn es bereits eine Datenbank mit diesem Namen, aber einer anderen Datenbankdatei gibt. Dieser Fall tritt recht oft ein, wenn auf der Festplatte unterschiedliche Versionen der Datenbankdateien herumliegen.

Jetzt bleibt noch eine Frage zu beantworten: Woher nehmen Sie die *.mdf/*.ldf-Dateien? Normalerweise existiert die Datenbank aus der Entwicklungsphase des Programms ja schon und wurde bisher wie alle anderen Datenbanken vom SQL Server verwaltet.

Die naheliegendste Vorgehensweise bestünde darin, die Dateien einfach aus dem Verzeichnis C:\Programme\Microsoft SQL Server\MSSQL.*n*\MSSQL\Data zu kopieren. Das ist aber gefährlich! Solange die Datenbank genutzt wird und der SQL Server auf die Dateien zugreift, dürfen Sie die Dateien nicht einfach kopieren. Sie müssen zuerst im Management Studio die Datenbank vom SQL Server lösen

(Kontextmenükommando TASKS|TRENNEN). Anschließend kopieren Sie die Dateien in Ihr Visual-Basic-Projektverzeichnis oder an einen anderen Ort.

Falls Sie die Datenbank weiterhin auch im SQL Server nutzen möchten, können Sie die Datenbank anschließend wieder mit den ursprünglichen Datenbankdateien verbinden (Kontextmenükommando ANFÜGEN). Das führt aber dazu, dass es nun zwei Versionen Ihrer Datenbank gibt: eine, auf die Ihr VB-Programm direkt zugreift, und eine zweite, die vom SQL Server verwaltet wird. Das stiftet oft Verwirrung.

* * * TIPP

*Nach meinen Erfahrungen ist der direkte Zugriff auf Datenbankdateien oft mit Komplikationen verbunden. Vermeiden Sie diesen Fall so lange wie möglich, und stellen Sie Ihr VB-Programm erst dann auf einen direkten Dateizugriff um, wenn Sie den **AttachedDBFilename**-Mechanismus ausprobieren möchten.*

Erstellen Sie eine Sicherheitskopie Ihrer Datenbankdateien! Die Dateien dürfen zu diesem Zeitpunkt nicht in Verwendung sein, müssen also vom SQL Server getrennt werden.

Benutzerinstanzen für Anwender ohne Administrationsrechte (User Instance)

Der Verbindungsaufbau zur Datenbankdatei gelingt nur, wenn Sie bzw. Ihrer Kunde mit Administratorrechten arbeiten. Um auch unter Windows ein wenig mehr Sicherheit zu erreichen, zwingen sicherheitsbewusste Firmen Ihre Mitarbeiter dazu, einen Login ohne Administratorrechte zu verwenden. Das ist unbequem, weil nun alle möglichen Programme nicht mehr funktionieren. Ihr gerade entwickeltes VB-Datenbankprogramm gehört leider dazu. Der Grund: Der SQL Server erlaubt nur zuvor eingerichteten Datenbankbenutzern den Zugriff auf eine Datei (siehe auch Abschnitt 24.4). Ausgenommen von dieser Regel sind Benutzer mit Administratorrechten.

Damit Windows-Nutzer ohne Administratorrechte und auch ohne eine komplizierte Neukonfiguration der Datenbankbenutzerverwaltung auf die SQL Server Express Edition zugreifen können, hat sich Microsoft in Version 2005 ein neues Verfahren ausgedacht: sogenannte Benutzerinstanzen:

Wenn die Verbindungszeichenkette den Parameter ***User Instance=True*** enthält, wird beim Verbindungsaufbau eine neue Instanz der SQL Server Express Edition gestartet. Während die Standardinstanz des SQL Servers mit Administratorrechten ausgeführt wird, hat die Benutzerinstanz nur die Rechte des Nutzers, der die Verbindung initiiert hat. Der wesentliche Vorteil besteht darin, dass der Benutzer nun (aus Sicht der Benutzerinstanz des SQL Servers) uneingeschränkte Rechte über seine eigene Datenbankdatei hat, also gewissermaßen sein eigener Datenbankadministrator ist. Da die Benutzerinstanz des SQL Servers nur auf die Dateien des Benutzers (aber nicht auf andere Dateien des Systems) zugreifen darf, ergeben sich daraus keine zusätzlichen Sicherheitsrisiken.

Damit Benutzerinstanzen funktionieren, müssen zwei Voraussetzungen erfüllt sein:

- Die SQL Server Express Edition muss bereits als Systemdienst laufen. Das ist bei einer Standardinstallation der SQL Server Express Edition immer der Fall. Dieser Dienst kümmert sich um den Start der Benutzerinstanz.
- Der Benutzer muss Lese- und Schreibrechte für die Datenbankdatei haben. Das bedeutet in der Regel, dass sich die Datenbankdatei in einem persönlichen Verzeichnis des Benutzers befinden muss, nicht im Verzeichnis des Programms.

Die folgenden Zeilen gehen davon aus, dass sich die Datenbankdateien im Verzeichnis Dokumente und Einstellungen*benutzername*\\Anwendungsdaten befinden.

```
Dim dbname As String = _
  Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\mylibrary.mdf"
conn.ConnectionString = "Server=.\sqlexpress;AttachDBFilename=" + dbName + ";" + _
  "Integrated Security=True;User Instance=True;Connection Timeout=90"
conn.Open()
```

Das Konzept der Benutzerinstanzen ist auch mit Nachteilen verbunden:

- **Lange Startzeit:** Beim ersten Verbindungsaufbau muss eine neue Instanz des SQL Servers gestartet werden. Das dauert auf älteren Rechnern ziemlich lange. Aus diesem Grund ist es empfehlenswert, im *ConnectionString* einen höheren Timeout-Wert einzustellen.
- **Hoher Ressourcenbedarf (insbesondere RAM):** Die SQL Server Express Edition läuft doppelt, einmal als Systemdienst und einmal als Benutzerinstanz. Nach dem Schließen aller Verbindungen wird die Benutzerinstanz automatisch beendet – allerdings nicht sofort, sondern erst nach geraumer Zeit.
- **Inkompatibilität zu anderen SQL-Server-Versionen:** Nur die Express Edition unterstützt Benutzerinstanzen. Die Vorgehensweise ist damit inkompatibel zu den Vollversionen des SQL Servers.
- **Kein Datenaustausch:** Nur ein einziger Benutzer hat Zugriff auf die Datenbank. Ein gleichzeitiger Zugriff durch mehrere Benutzer ist nicht möglich. Damit besteht keine sinnvolle Möglichkeit zum Datenaustausch zwischen mehreren Benutzern. Benutzerinstanzen sind somit nur für *single-user*-Programme zweckmäßig.

* * * TIPP

Angesichts der letzten Einschränkung ist es eigentlich absurd, ein vollwertiges relationales Client-Server-System einzusetzen. Eine bessere Alternative ist zumeist die Verwendung der SQL Server Compact Edition, die ich Ihnen in Kapitel 33 näher vorstelle.

Probleme

Beim Testen der Parameter *AttachDBFilename* und *User Instance* sind immer wieder Probleme aufgetreten. Die Fehlermeldungen halfen zumeist nicht weiter. Angeblich wurde der Zugriff auf die Datei verweigert, der SQL Server konnte die Standarddatenbank des Benutzers nicht öffnen, die Logging-Datei nicht finden etc. Hier einige Tipps, wenn es Probleme gibt:

- Starten Sie die SQL Server Express Edition (und alle anderen auf dem Rechner laufenden SQL Server) neu. Zum Neustart können Sie unter Windows Vista wahlweise SYSTEMSTEUERUNG | SYSTEM UND WARTUNG | VERWALTUNG | DIENSTE oder das Programm SQL SERVER 2005 | KONFIGURATIONSTOOLS | OBERFLÄCHENKONFIGURATION einsetzen.

- Löschen Sie im Management Studio Datenbanken, deren Name durch einen Dateinamen angegeben wird, die sich dort aber nicht öffnen lassen. (Die Datenbank wird also im Management Studio angezeigt. Ein Doppelklick auf die Datenbank bleibt aber ohne Wirkung.)

  Vorsicht: Die Datenbankdateien werden dadurch tatsächlich gelöscht! Ich gehe davon aus, dass Sie eine Sicherheitskopie besitzen und ausgehend von dieser Kopie Ihre weiteren Versuche durchführen.

- Stellen Sie sicher, dass die von Ihnen angegebene Datenbankdatei nicht in Verwendung ist. Sie können keine Verbindung zu einer Datenbankdatei herstellen, die irgendeine Version des SQL Servers als normale (verbundene) Datenbank nutzt.

- Führen Sie im Direktfenster von Visual Studio *System.Data.SqlClient.SqlConnection.ClearAllPools()* aus, beenden Sie alle Programme, die möglicherweise auf die Datenbank zugreifen (inklusive Visual Studio selbst), und gehen Sie einen Kaffee trinken. Wenn Sie Glück haben, sind Ihre Probleme danach auf wunderbare Weise verschwunden. Mit Wundern hat das natürlich nichts zu tun, sondern mit dem automatischen *connection pooling* des SQL Servers. Dieser Mechanismus hält Datenbankdateien eine Weile auch dann noch geöffnet, wenn Sie die Verbindung eigentlich schon geschlossen haben. Mehr Informationen zum *connection pooling* finden Sie am Ende von Abschnitt 25.2.

Warum derartige Probleme so häufig auftreten, ist mir nicht ganz klar geworden. Wahrscheinlich haben die Probleme mit der in Visual Studio üblichen Vorgehensweise zu tun, zum Projekt gehörige Datenbankdateien bei jedem Programmstart neuerlich in das Bin\Debug-Verzeichnis zu kopieren. Für den SQL Server sieht das so aus, als würde jedes Mal eine neue Datenbankdatei verbunden. Offensichtlich kommt der Server unter gewissen Umständen mit der Verwaltung der verbundenen Datenbankdateien durcheinander. Insgesamt wirkt der direkte Zugriff auf Datenbankdateien extrem fehleranfällig und noch ziemlich unausgereift.

25.4 Liste aller SQL-Server-Instanzen ermitteln

Es ist zulässig, mehrere SQL Server auf einem Rechner zu installieren. Microsoft hat diese Möglichkeit insbesondere für die SQL Server Express Edition geschaffen, die im Rahmen eines Programms oder einer Demo auf dem Rechner installiert werden kann, ohne bereits existierenden SQL-Server-Installationen ins Gehege zu kommen. Bei derartigen Mehrfachinstallationen wird jeder Server als eigene Instanz bezeichnet.

Wenn auf dem Rechner eine Developer- oder Vollversion des SQL Servers installiert ist, gilt diese Instanz üblicherweise als Standardinstanz. Sie wird beim Verbindungsaufbau einfach in der Form *Server=rechnername* angesprochen. Wenn der SQL Server auf dem lokalen Rechner läuft, sind auch die Schreibweisen *Server=localhost*, *Server=(local)* und *Server=.* zulässig.

Alle anderen Instanzen bekommen bei der Installation einen Namen. Die erste Instanz der SQL Server Express Edition heißt standardmäßig *sqlexpress*. Beim Verbindungsaufbau geben Sie diesen Namen in der Form *Server=rechnername\instanzname* an. Für Instanzen auf dem lokalen Rechner ist auch die Kurzschreibweise *Server=.\instanzname* zulässig.

Es bestehen zahlreiche Möglichkeiten, Listen der im Netzwerk erreichbaren bzw. auf dem lokalen Rechner installierten Server-Instanzen zu ermitteln:

- *Sql.SqlDataSourceEnumerator*
- Registrierdatenbank
- ODBC- oder OLEDB-API-Funktionen
- DMO-Bibliothek (ActiveX)
- Stored Procedures (setzt den Kontakt zu einem laufenden SQL-Server voraus)

Leider liefert keine dieser Methoden zuverlässige Ergebnisse. Generell ist es ein schwieriges Unterfangen, aktive SQL Server zu entdecken. Das Resultat ist von der Konfiguration der SQL Server (sind die Netzwerkfunktionen aktiviert?), von der Netzwerkkonfiguration des Systems, von Firewalls etc. abhängig. Damit Sie die im lokalen Netzwerk laufenden SQL Server finden, sollte auf dem lokalen Rechner der Windows-Dienst *SQL Browser* laufen. Außerdem müssen die SQL Server so konfiguriert sein, dass Sie via TCP/IP kommunizieren, was standardmäßig aus Sicherheitsgründen nicht der Fall ist.

Beispielprogramm

Das Beispielprogramm ADO.NET-classes\enumerate-sql-instances enthält Code für die neue Klasse *SqlServerHelper*, deren Methoden *NetInstances, OdbcInstances* und *RegistryInstances* jeweils ein *String*-Array mit den gefundenen Server-Instanzen zurückliefern.

NetInstances greift auf die Klasse *Sql.SqlDataSourceEnumerator* zurück, *OdbcInstances* ruft ODBC-API-Funktionen auf, und *RegistryInstances* wertet die Registrierdatenbank aus. Beachten Sie, dass sowohl *NetInstances* als auch *OdbcInstances* ebenso langsam wie unzuverlässig sind. Sie müssen sich in der Regel mehrere Minuten gedulden, bis Sie Ergebnisse erhalten.

Zum Zeitpunkt des Testlaufs waren insgesamt vier Server-Instanzen verfügbar: je eine Developer und Express Edition auf dem lokalen Rechner *uranus* sowie im Netzwerk auf dem Rechner *merkur*. Bei allen Instanzen waren TCP/IP-Verbindungen aktiviert. Auf beiden Rechnern lief der SQL Server Browser.

Abbildung 25.3 zeigt das triste Ergebnis: *NetInstances* findet keine einzige Instanz, *OdbcInstances* die beiden Instanzen auf dem lokalen Rechner (wobei bei *(local)* eine Klammer fehlt), und *Registry-Instances* findet ebenfalls nur die beiden Instanzen des lokalen Rechners.

Nach meinen Erfahrungen gibt es momentan keine Methode, um SQL-Server-Instanzen im Netzwerk auch nur einigermaßen zuverlässig aufzuspüren. *RegistryInstances* findet die lokalen Instanzen sehr schnell, gibt aber keine Informationen darüber, welche der lokalen Instanzen tatsächlich läuft! (Insofern ist das Ergebnis von *RegistryInstances* nicht besonders praxistauglich: Es ist nicht möglich, eine Datenbankverbindung zu einer zwar installierten, aber nicht gestarteten Instanz herzustellen.)

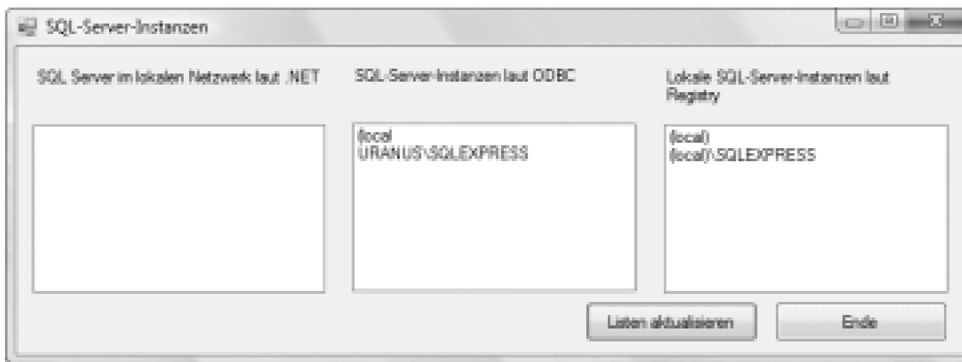

Abbildung 25.3: SQL-Server-Instanzen

SQL-Server-Instanzen mit der Sql.SqlDataSourceEnumerator-Klasse ermitteln

Die Klasse ***Sql.SqlDataSourceEnumerator*** hat keinen Konstruktor. Der einzige Weg zu einem *Sql-DataSourceEnumerator*-Objekt führt über die Eigenschaft *Instance*. Für das so ermittelte Objekt führen Sie nun die Methode ***GetDataSources*** aus. Als Ergebnis erhalten Sie ein *DataTable*-Objekt. (Allgemeine Informationen zur *DataTable*-Klasse gibt Abschnitt 26.2.) Jede Zeile dieser virtuellen Tabelle beschreibt in vier Spalten eine SQL-Server-Instanz:

- *ServerName* liefert den Hostnamen des Rechners, auf dem der Server läuft.
- *InstanceName* soll den Instanznamen des Servers angeben. Allerdings enthielt diese Spalte bei meinen Tests immer *DBNull.Value*.
- *IsClustered* gibt an, ob es sich um einen Cluster-Server handelt.
- *Version* gibt die Versionsnummer des Servers an.

Die folgenden Zeilen zeigen, wie aus den Informationen der *DataTable* ein *String*-Feld zusammengesetzt wird, dessen Einträge *ServerName* bzw. *ServerName/InstanceName* lauten. Der Name des lokalen Rechners wird dabei durch *(local)* ersetzt.

```
' Beispiel ADO.NET-classes\enumerate-sql-instances\Class1.vb
Imports System.Data

Public Class SqlServerHelper
  Public Shared Function NetInstances() As String()
    Dim i As Integer
    Dim srv, inst As String
    Dim result As String() = {}
    Dim dt As DataTable

    dt = Sql.SqlDataSourceEnumerator.Instance.GetDataSources()
    If dt.Rows.Count > 0 Then
      ReDim result(dt.Rows.Count - 1)
      For i = 0 To dt.Rows.Count - 1
        srv = dt.Rows(i)!ServerName
        If srv = My.Computer.Name Then srv = "(local)"
        inst = IIf(dt.Rows(i)!InstanceName Is DBNull.Value,
                   "", "\" + dt.Rows(i)!InstanceName)
        result(i) = srv + inst
      Next
    End If
    Return result
  End Function
End Class
```

Lokale Server-Instanzen aus der Registrierdatenbank lesen

Die einzig verlässliche Methode, um die auf dem lokalen Rechner installierten SQL-Server-Instanzen zu ermitteln, ist offensichtlich die Registrierdatenbank. In den Verzeichnissen

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Microsoft SQL Server\Instance Names\SQL
```

und

```
HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\Microsoft SQL Server\Instance Names\SQL
```

befindet sich je eine Liste mit den Namen aller 64-Bit- bzw. 32-Bit-Instanzen, wobei die Standardinstanz den Namen *MSSQLSERVER* hat. Die im Folgenden abgedruckte Funktion stellt den Instanznamen noch *(local)* voran.

```
Public Shared Function RegistryInstances() As String()
  Dim i As Integer
  Dim result As String() = {}
  Dim rk As RegistryKey

  rk = Registry.LocalMachine.OpenSubKey( _
    "Software\Microsoft\Microsoft SQL Server\Instance Names\SQL")
  If Not (rk Is Nothing) Then
    result = rk.GetValueNames()
    For i = 0 To result.GetUpperBound(0)
      If result(i) = "MSSQLSERVER" Then
        result(i) = "(local)"
      Else
        result(i) = "(local)\" + result(i)
      End If
    Next
    rk.Close()
  End If

  rk = Registry.LocalMachine.OpenSubKey( _
    "Software\Wow6432Node\Microsoft\Microsoft SQL Server\Instance Names\SQL")
  ... Auswertung wie oben

  Return result
End Function
```

25.5 SqlCommand und SqlParameter (SQL-Kommandos ausführen)

SqlCommand-Objekte benötigen Sie, um SQL-Kommandos auszuführen. Die folgenden Zeilen zeigen die prinzipielle Anwendung der Klasse. *conn* ist ein *SqlConnection*-Objekt, das eine aktive Verbindung zum SQL Server enthält. *n* enthält nach der Ausführung des Kommandos die Anzahl der veränderten Datensätze (in diesem Beispiel also 1).

```
Dim n As Integer
Dim com As New SqlCommand("INSERT INTO titles (title) VALUES('test')", conn)
n = com.ExecuteNonQuery()
```

Je nachdem, welche Art von SQL-Kommando Sie ausführen und wie Sie die Ergebnisse verarbeiten möchten, stehen vier *Execute*-Methoden oder die Verwendung einer *SqlDataAdapter*-Klasse zur Wahl:

- ***ExecuteNonQuery* für Kommandos ohne Ergebnisse:** *INSERT-, UPDATE-, DELETE-* und andere SQL-Kommandos, die keine unmittelbaren Ergebnisse liefern, führen Sie mit *ExecuteNonQuery* aus. Die Methode liefert als Ergebnis die Anzahl der geänderten Datensätze.

```
' Beispiel ADO.NET-classes\command-intro\Form1.vb
Dim n As Integer
Dim com As New SqlCommand("INSERT INTO titles (title) VALUES('test')", conn)
n = com.ExecuteNonQuery()
```

- ***ExecuteScalar* für *SELECT*-Kommandos mit einem Einzelergebnis:** *SELECT*-Kommandos, die einen einzelnen Wert als Ergebnis liefern, führen Sie am effizientesten mit *ExecuteScalar* aus. Die Methode liefert das Ergebnis als *Object* zurück. Sie eignet sich für Kommandos wie *SELECT COUNT(*) FROM table* oder *SELECT column FROM table WHERE id=1234*.

```
Dim n As Integer
Dim com As New SqlCommand("SELECT COUNT(*) FROM titles", conn)
n = com.ExecuteScalar()
```

Falls Sie für Ihr Projekt *Option Strict* verwenden, muss das *ExecuteScalar*-Ergebnis von *Object* in den gewünschten Datentyp umgewandelt werden:

```
n = CInt(com.ExecuteScalar())
```

Sonderfälle: Wenn das *SELECT*-Kommando kein Ergebnis liefert, gibt *ExecuteScalar* den Wert *Nothing* zurück. Wenn das *SELECT*-Kommando mehrere Datensätze bzw. mehrere Spalten liefert, gibt *ExecuteScalar* den Wert der ersten Spalte des ersten Datensatzes zurück. In beiden Fällen tritt kein Fehler auf. Im zweiten Fall besteht allerdings keine Möglichkeit festzustellen, ob das Ergebnis der *SELECT*-Abfrage eindeutig war oder ob es mehrere Ergebnisse gab. Verwenden Sie gegebenenfalls *ExecuteReader*!

- ***ExecuteReader* für gewöhnliche *SELECT*-Kommandos:** Ein gewöhnliches *SELECT*-Kommando ergibt eine im Voraus oft unbekannte Anzahl von Datensätzen. Die Methode *ExecuteReader* liefert deswegen als Ergebnis ein *DataReader*-Objekt zurück, mit dem Sie anschließend alle Datensätze auslesen können (siehe Abschnitt 25.6).

```
Dim com As New SqlCommand("SELECT title FROM titles ORDER BY title", conn)
Dim reader As SqlDataReader
reader = com.ExecuteReader()
While reader.Read()
  'Zugriff auf einzelne Spalten durch reader!spaltenname
End While
```

- ***SqlDataAdapter* für gewöhnliche *SELECT*-Kommandos:** Mehr Komfort bei der Weiterverarbeitung von *SELECT*-Ergebnissen bieten *DataSets*. In diesem Fall müssen Sie das *SqlCommand*-Objekt zur Initialisierung eines *SqlDataAdapters* verwenden, der die Schnittstelle zum *DataSet* darstellt (siehe Abschnitt 25.7).

- ***ExecuteXmlReader* für XML-*SELECT*-Kommandos:** Der SQL Server kann *SELECT*-Ergebnisse als XML-Baum liefern. Dazu muss das *SELECT*-Kommando um die Schlüsselwörter *FOR XML RAW|AUTO|EXPLICIT* erweitert werden. (Diese Erweiterung der *SELECT*-Syntax entspricht allerdings keinem gängigen SQL-Standard.)

  Wenn Sie derartige Kommandos ausführen möchten, müssen Sie die Methode *ExecuteXmlReader* einsetzen. Als Ergebnis erhalten Sie ein *System.Xml.XmlReader*-Objekt. Die folgenden Zeilen zeigen, wie das *XmlReader*-Objekt durch eine Schleife mit wiederholten *ReadOuterXml*-Aufrufen in eine Zeichenkette umgewandelt wird.

```
Dim s As String
Dim com As New SqlCommand("SELECT * FROM titles ORDER BY title FOR XML AUTO", conn)
Dim reader As System.Xml.XmlReader
Dim sb As New System.Text.StringBuilder()
reader = com.ExecuteXmlReader()
reader.MoveToContent()
s = reader.ReadOuterXml()
While (s <> "")
  sb.Append(s + vbCrLf)
  s = reader.ReadOuterXml()
End While
'nun liefert sb.ToString() das gesamte XML-Ergebnis als Zeichenkette
```

Das Beispielprogramm command-intro illustriert die verschiedenen Verfahren und zeigt die Ergebnisse in einer Textbox an (siehe Abbildung 25.4). Durch den DELETE-Button werden übrigens nur jene Bücher aus der *titles*-Tabelle gelöscht, deren Titel mit *test* beginnt.

Abbildung 25.4: SqlCommand-Varianten ausprobieren

Bei der Ausführung von SQL-Kommandos kann es zu diversen Fehlern kommen (Syntaxfehler im SQL-Code, Verletzung von Integritätsregeln etc.). Daher sollten Sie die *Execute*-Methode in der Regel durch *Try/Catch* absichern.

Die Eigenschaft ***CommandTimeout*** bestimmt, wie lange maximal auf die Ausführung des Kommandos gewartet wird (normalerweise 30 Sekunden). Wenn der SQL Server innerhalb dieser Zeit keine Rückmeldung gibt, wird ein Fehler ausgelöst.

Wert der ID-Spalte nach INSERT ermitteln

Bei Tabellen mit einer *ID*-Spalte ist es nach *INSERT*-Kommandos oft erforderlich, den von der Datenbank automatisch erzeugten ID-Wert zu ermitteln. Bei herkömmlichen ID-Spalten (*Integer*-Datentyp und *IsIdentity=True*) führen Sie dazu einfach eine weitere Abfrage mit *SELECT @@IDENTITY* aus. Der Rückgabewert ist der gesuchte ID-Wert.

Am einfachsten ist es, gleich nach dem Verbindungsaufbau ein im gesamten Programm verfügbares *SqlCommand*-Objekt zu erzeugen, das nur diesem Zweck dient:

```
' Beispiel ADO.NET-classes\command-intro\Form1.vb

Dim identityComm As SqlCommand
identityComm = New SqlCommand("SELECT @@IDENTITY", conn)
```

Später führen Sie dann, wann immer Sie einen *ID*-Wert benötigen, die Methode *ExecuteScalar* aus:

```
identityComm.ExecuteScalar()
```

> > > HINWEIS

SELECT @@IDENTITY muss für dieselbe Datenbankverbindung ausgeführt werden, für die auch das INSERT-Kommando ausgeführt wurde. Der SQL-Server liefert auch dann den korrekten Wert, wenn in die betreffende Tabelle mittlerweile über eine andere Verbindung weitere Datensätze eingefügt wurden.

Falls Sie mit Transaktionen arbeiten, muss SELECT @@IDENTITY innerhalb der laufenden Transaktion ausgeführt werden. Wenn Ihre Datenbank Trigger einsetzt oder wenn Sie mit Stored Procedures arbeiten, müssen Sie statt @@IDENTITY die Funktionen SCOPE_IDENTITY oder IDENT_CURRENT auswerten (siehe Abschnitt 24.5).

Tipps zum Einfügen verknüpfter Datensätze in eine Datenbank finden Sie in den Abschnitten 27.1 (für gewöhnliche ID-Spalten) und 27.3 (für GUID-Spalten).

SqlXmlCommand

Die Methode *ExecuteXmlReader* unterstützt nur einen Teil der XML-Funktionen des SQL Servers und hat außerdem einige bekannte Mängel. Einer besteht darin, dass dem resultierenden *XmlReader*-Objekt ein *root*-Tag fehlt, weswegen *ReadOuterXml* immer nur einen Datensatz liefert (siehe auch den oben abgedruckten Code zur Auswertung der Daten).

Mehr Flexibilität bei der Nutzung der XML-Funktionen bieten die Klassen der Bibliothek *Microsoft.Data.Xml*. Diese Bibliothek ist nicht Teil des .NET-Frameworks! Vielmehr wird die Bibliothek zusammen mit dem SQL Server installiert (üblicherweise in das Verzeichnis C:\Programme\SQLXML 4.0). Sie müssen eine Referenz auf die Bibliothek einrichten.

Die zusätzlichen Funktionen erkaufen Sie sich durch den Nachteil, dass die Klassen dieser Bibliothek unbegreiflicherweise nicht in der Lage sind, eine vorhandene Verbindung zur Datenbank zu nutzen. Vielmehr müssen Sie beim Erzeugen neuer Objekte eine Verbindungszeichenkette in der OLEDB-Syntax angeben. Es fehlt hier der Platz, detailliert auf die *SqlXml*-Klassen einzugehen. Stattdessen geben die folgenden Zeilen ein Beispiel für die Anwendung von ***SqlXmlCommand***. Mit der Eigenschaft *com.**RootTag*** stellen Sie ein, welchen Namen das *root*-Tag des XML-Ergebnisses haben soll.

```
' Beispiel ADO.NET-classes\command-intro\Form1.vb

Dim com As New SqlXmlCommand( _
  "Provider=SQLOLEDB;Server=(local);database=mylibrary;Integrated Security=SSPI")
Dim reader As System.Xml.XmlReader
com.CommandText = "SELECT * FROM titles ORDER BY title FOR XML AUTO"
com.RootTag = "titles"
reader = com.ExecuteXmlReader()
reader.MoveToContent()
' reader.ReadOuterXml() liefert eine XML-Zeichenkette mit allen Ergebnisdatensätzen
```

Mehrere Kommandos auf einmal ausführen

Die SQL-Zeichenkette darf mehrere durch Strichpunkte getrennte Kommandos enthalten. In diesem Fall führt der SQL Server alle Kommandos der Reihe nach aus. Beachten Sie aber, dass nicht alle ADO.NET-Provider diese Möglichkeit bieten. Insbesondere ist dies in *SqlCeCommands* für die SQL Server Compact Edition nicht zulässig!

SQL-Kommandos mit Parametern ausführen

Alle bisherigen Beispiele wurden ohne Parameter ausgeführt, d.h., der im *SqlCommand*-Objekt angegebene SQL-Code war bereits komplett. Parameter sind dann zweckmäßig, wenn Sie ein Kommando mehrfach hintereinander ausführen möchten und dabei nur einen Parameter verändern möchten oder wenn die Übergabe der Daten syntaktisch kompliziert ist: Wissen Sie auswendig, in welcher Syntax Sie Daten und Zeiten an den SQL Server übergeben müssen? Wie Sie binäre Daten angeben? Wie Sie mit Zeichenketten umgehen, die selbst die Zeichen ' oder " enthalten? Wie Sie *NULL* übergeben?

Sie ersparen sich die Suche im SQL-Handbuch, wenn Sie das SQL-Kommando mit einem Parameter formulieren und den betreffenden Wert in Form eines ***SqlParameter***-Objekts angeben. Parameter werden innerhalb des SQL-Kommandos durch @*name* gekennzeichnet. (Beachten Sie, dass die Parametersyntax innerhalb des SQL-Codes vom .NET-Datenbankprovider abhängig ist! Beim ODBC-Treiber müssen Sie statt @*name* das Zeichen ? verwenden.)

Anschließend müssen Sie für jeden Parameter ein *SqlParameter*-Objekt erzeugen und dessen *Value*-Eigenschaft einstellen. Bei Kommandos, die nur ein einziges Mal ausgeführt werden sollen, sieht die einfachste Vorgehensweise so aus:

```
Dim com As New SqlCommand( _
  "INSERT INTO titles (title, publdate) VALUES (@title, @date)", conn)
com.Parameters.Add("@title", Data.SqlDbType.NVarChar).Value = "test"
com.Parameters.Add("@date", Data.SqlDbType.DateTime).Value = Now
n = com.ExecuteNonQuery()
```

Wenn das Kommando mehrere Male ausgeführt werden soll, müssen Sie die *SqlParameter*-Objekte in Variablen speichern:

```
Dim i As Integer
Dim com As New SqlCommand( _
  "INSERT INTO titles (title, publdate) VALUES (@title, @date)", conn)
Dim titlepara, datepara As SqlParameter
titlepara = com.Parameters.Add("@title", Data.SqlDbType.NVarChar)
datepara = com.Parameters.Add("@date", Data.SqlDbType.DateTime)

For i = 1 To 3
  titlepara.Value = "test " + i.ToString
  datepara.Value = Now
  com.ExecuteNonQuery()
Next
```

Selbstverständlich können Sie jede Art von SQL-Kommando mit Parametern ausführen, also auch *DELETE*- oder *SELECT*-Kommandos. Bevor Sie das Kommando zum ersten Mal durch *ExecuteXxx* ausführen, können Sie *com.**Prepare**()* ausführen. Der SQL Server bereitet das Kommando dann für die weitere Ausführung vor, was bei einem mehrfachen Aufruf von *ExecuteXxx* einen kleinen Geschwindigkeitsvorteil mit sich bringen kann. *Prepare* setzt allerdings voraus, dass Sie bei allen Parametern mit variabler Länge (z.B. beim Datentyp *NVarChar*) die maximale Länge angeben:

```
... wie bisher
titlepara = com.Parameters.Add("@title", Data.SqlDbType.NVarChar, 100)
datepara = com.Parameters.Add("@date", Data.SqlDbType.DateTime)
com.Prepare()
```

Parameter und NULL

Damit Parameter *NULL* verarbeiten können, muss die Eigenschaft ***IsNullable*** auf *True* gesetzt werden. (Diese Eigenschaft hat nichts mit *Nullable(Of werttyp)* zu tun!)

```
datepara.IsNullable = True
```

Wenn Sie an den Parameter nun tatsächlich den Wert *NULL* übergeben möchten, weisen Sie der *Value*-Eigenschaft *DBNull.Value* zu (nicht *Nothing*!):

```
datepara.Value = DBNull.Value
```

25.6 SqlDataReader (SELECT-Ergebnisse lesen)

Mit einem ***SqlDataReader***-Objekt können Sie die Ergebnisse einer *SELECT*-Abfrage einmal der Reihe nach auslesen. Es ist nicht möglich, in beliebiger Reihenfolge oder wiederholt auf die Ergebnisdatensätze zuzugreifen. Ebenso wenig können Sie die Daten ändern. Zu guter Letzt fehlt eine *Count*-Eigenschaft; um die Anzahl der Ergebnisdatensätze zu ermitteln, müssen Sie beim Auslesen mitlesen. Wenn Sie mehr Komfort wünschen, müssen Sie mit *DataSets* arbeiten. Falls Sie früher mit ADO gearbeitet haben: Der *SqlDataReader* hat große Ähnlichkeiten mit einem *Recordset*-Objekt mit den Eigenschaften *forward-only, read-only* und *server side cursor*.

Die Ergebnisdatensätze verbleiben auf dem Datenbankserver und werden erst bei Bedarf zu Ihrem Programm übertragen. Der *SqlDataReader* beansprucht auch bei riesigen *SELECT*-Ergebnissen nur wenig Speicherplatz; allerdings muss der SQL Server die Daten vorrätig halten, bis Sie das *SqlDataReader*-Objekt explizit schließen oder es automatisch durch eine *garbage collection* aus dem Speicher entfernt wird. Führen Sie *Close* aus, sobald Sie mit dem Auslesen der Daten fertig sind!

Um einen *SqlDataReader* zu erzeugen, führen Sie die Methode *ExecuteReader* für ein *SqlCommand*-Objekt aus. Anschließend lesen Sie die Datensätze zeilenweise durch *Read* aus. Diese Methode liefert *True* oder *False*, je nachdem, ob noch ein Datensatz gefunden wurde oder ob bereits das Ende der Datensatzliste erreicht ist.

Der Zugriff auf die Spalten des gerade aktuellen Datensatzes erfolgt durch *reader!spaltenname*. (Das ist die Kurzschreibweise für *reader.Items("spaltenname")*.) Als Ergebnis erhalten Sie *Object*-Daten. Falls Sie in Ihrem Projekt *Option Strict* verwenden, müssen Sie die Ergebnisse mit *CStr*, *CInt* etc. in das gewünschte Datenformat umwandeln. Alternativ können Sie den aktuellen Datensatz auch mit *reader.GetByte(n)*, *-.GetDateTime(n)*, *-.GetInt32(n)* etc. lesen. Dabei ist *n* die Spaltennummer (0 für die erste Spalte).

Bei Zuweisungen von *SELECT*-Ergebnissen (z.B. *stringvar = CStr(reader!spaltenname)*) müssen Sie immer auch den Fall berücksichtigen, dass eine Spalte *NULL* enthält. *reader!spaltenname* liefert dann den Wert *DBNull.Value*. Visual Basic ist nicht in der Lage, diesen Wert einer Variablen zuzuweisen (egal ob *Nullable* oder nicht, siehe auch Abschnitt 27.8). Es bestehen verschiedene Möglichkeiten, *NULL* festzustellen:

```
If Convert.IsDBNull(reader!spaltenname) Then ...
If reader!spaltenname Is DBNull.Value Then ...
If reader.IsDBNull(n) Then ...
```

Die folgenden Zeilen lesen aus der *titles*-Tabelle alle Buchtitel sowie deren Veröffentlichungsdatum. Die Ergebnisse werden zeilenweise ausgelesen, mit einem *StringWriter*-Objekt zu einer Zeichenkette zusammengesetzt und schließlich angezeigt.

```
' Beispiel ADO.NET-classes\command-intro\Form1.vb
Dim result As New System.IO.StringWriter
Dim com As New SqlCommand("SELECT title, publDate FROM titles ORDER BY title", conn)
Dim reader As SqlDataReader
reader = com.ExecuteReader()
While reader.Read()
  If reader!publDate Is DBNull.Value Then
    result.WriteLine("{0}", reader!title)
  Else
    result.WriteLine("{0} ({1})", reader!title, reader!publDate)
  End If
End While
reader.Close()
MsgBox("Die Tabelle titles enthält die folgenden Titel: " + _
       vbCrLf + vbCrLf + result.ToString())
```

Mehrere SqlDataReader parallel benutzen (MARS)

ADO.NET in Kombination mit dem SQL Server 2005 unterstützt *Multiple Active Result Sets*, kurz *MARS*. Das bedeutet, dass Sie mehrere *SqlDataReader* parallel benutzen und während des Auslesens auch andere SQL-Kommandos ausführen können. Um *MARS* zu nutzen, müssen Sie in der Verbindungszeichenkette *MultipleActiveResultSets = True* einbauen.

Bei ADO.NET 1.0/1.1, bzw. wenn Sie mit älteren SQL-Server-Versionen arbeiten (2000 und früher), kann pro Verbindung (*Connection*) nur ein *SqlDataReader* aktiv sein. Erst wenn alle Ergebnisse des *SqlDataReaders* ausgelesen sind, kann ein neuer *SqlDataReader* benutzt oder ein anderes SQL-Kommando ausgeführt werden! Diese Einschränkung können Sie umgehen, indem Sie eine zweite Verbindung zum SQL Server einrichten.

Metadaten zum SELECT-Ergebnis

Mit den *SqlDataReader*-Methoden ***GetName****(n)* und ***GetDataTypeName****(n)* können Sie zu jeder Spalte den Spaltennamen und den Datentyp ermitteln. Wenn Sie wissen möchten, ob die Abfrage überhaupt Ergebnisse geliefert hat, werten Sie die Eigenschaft ***HasRows*** aus. ***FieldCount*** liefert die Anzahl der Spalten des Ergebnisses. Wie ich bereits erwähnt habe, gibt es aber keine Eigenschaft oder Methode, um die Anzahl der Ergebnisdatensätze festzustellen.

Wenn Sie genauere Informationen zu den einzelnen Spalten benötigen, führen Sie ***ExecuteReader*** mit dem optionalen Parameter ***KeyInfo*** aus (damit die Metadaten gesammelt werden) und erzeugen dann mit ***GetSchemaTable*** eine *DataTable*, die alle erdenklichen Daten zu allen Spalten enthält. Jede Zeile der *DataTable* beschreibt eine Spalte des *SELECT*-Ergebnisses. Jede Spalte der *DataTable* ist vordefiniert (siehe die Online-Hilfe zu *GetSchemaTable*). Beispielsweise liefert *row!ColumnName* den Namen der Spalte des *SELECT*-Ergebnisses, *row!ColumnSize* die maximale Größe etc. Wenn Sie alle Daten ermitteln möchten, führen Sie einfach wie im folgenden Beispiel eine Schleife über alle *DataTable*-Spalten aus (*For Each col ...*). Abbildung 25.5 zeigt das Ergebnis dieser Schleife.

Abbildung 25.5: Metadaten zur Abfrage SELECT * FROM titles

```
' Beispiel ADO.NET-classes\command-intro\Form1.vb

Dim result As New System.IO.StringWriter
Dim com As New SqlCommand("SELECT * FROM titles ORDER BY title", conn)
Dim reader As SqlDataReader
Dim schema As DataTable
Dim col As DataColumn
Dim row As DataRow

reader = com.ExecuteReader(Data.CommandBehavior.KeyInfo)
schema = reader.GetSchemaTable()
```

```
For Each row In schema.Rows
  result.WriteLine("{0}", row!ColumnName)
  For Each col In schema.Columns
    result.WriteLine("  {0}: {1}", col.ColumnName, row(col.ColumnName))
  Next
  result.WriteLine()
Next
reader.Close()
ResultTextBox.Text = _
  "Die Tabelle titles enthält die folgenden Spalten: " + _
  vbCrLf + vbCrLf + result.ToString()
```

Mehrere SELECT-Ergebnisse

Mit einem *SqlCommand* können Sie mehrere durch Strichpunkte getrennte SQL-Kommandos auf einmal übergeben. Das hat zur Folge, dass ein einziges *SqlCommand* mehrere *SELECT*-Ergebnisse liefern kann. Auch *Stored Procedures* können mehrere *SELECT*-Ergebnisse liefern (siehe Abschnitt 27.5).

Der *SqlDataReader* gibt standardmäßig nur Zugriff auf das erste Ergebnis. Die weiteren Ergebnisse aktivieren Sie mit der Methode ***NextResult***. Sie liefert *False* zurück, wenn es keine weiteren Ergebnisse mehr gibt. Die folgenden Zeilen bilden eine Schleife über alle Ergebnisse (*Do*), über alle Datensätze jedes Ergebnisses (*While*) und über alle Spalten jedes Datensatzes (*For*) und geben die Daten als Zeichenkette aus.

```
' Beispiel ADO.NET-classes\command-intro\Form1.vb
Dim i As Integer
Dim result As New System.IO.StringWriter
Dim com As New SqlCommand( _
  "SELECT * FROM publishers ORDER BY publName;" & _
  "SELECT * FROM languages;SELECT * FROM titles", conn)
Dim reader As SqlDataReader = com.ExecuteReader()
Do
  result.WriteLine("------------")
  While reader.Read()
    For i = 0 To reader.FieldCount - 1
      If reader.IsDBNull(i) Then
        result.Write("NULL  ")
      Else
        result.Write("{0}  ", reader.GetValue(i))
      End If
    Next
    result.WriteLine()
  End While
Loop While reader.NextResult()
reader.Close()
MsgBox(result.ToString())
```

25.7 SqlDataAdapter und SqlCommandBuilder (Verbindung zum DataSet)

SqlDataAdapter

Der *SqlDataAdapter* stellt die Verbindung zwischen Datenbank und *DataSet* her bzw. konkreter gesagt zwischen SQL-Abfragen (als Zeichenkette oder als *SqlCommand*-Objekt) und den provider-unabhängigen *DataTable*-Objekten, die meist innerhalb eines *DataSets* verwaltet werden. Diese Verbindung gilt für beide Richtungen:

- Wenn Sie die Ergebnisse eines *SELECT*-Kommandos als *DataTable* verarbeiten möchten, benötigen Sie zum Datentransfer in die *DataTable* einen *SqlDataAdapter*. Das *DataTable*-Objekt ist zumeist ein Bestandteil innerhalb eines *DataSets*, es kann aber auch als selbstständiges Objekt eingesetzt werden.
- Wenn Sie in der *DataTable* Änderungen an den Daten vornehmen und diese bleibend in der Datenbank speichern möchten, ist auch hierfür der *SqlDataAdapter* verantwortlich. Das Objekt generiert passende *INSERT-*, *UPDATE-* und *DELETE*-Kommandos selbst.

  Beachten Sie aber, dass dabei einige Einschränkungen gelten: Datenänderungen können nur dann mit der Datenbank synchronisiert werden, wenn die zugrunde liegende *SELECT*-Abfrage das Primärschlüsselfeld der Tabelle enthält und wenn die Abfrage nur Daten einer Tabelle umfasst (keine *JOINs*). Sind diese Voraussetzungen nicht erfüllt, können Sie die *INSERT-*, *UPDATE-* und *DELETE*-Kommandos, die zur Synchronisierung verwendet werden sollen, selbst definieren. Dabei können Sie sogar auf *Stored Procedures* zurückgreifen. Der *SqlDataAdapter* bietet also eine Menge Flexibilität!

Die folgenden Zeilen zeigen eine Anwendung eines *SqlDataAdapters* (der Einfachheit halber ohne *DataSet*). Der *SqlDataAdapter* wird für das Kommando *SELECT * FROM titles ORDER BY title* und eine bereits bestehende Datenbankverbindung erzeugt. Mit ***Fill*** werden die Abfrageergebnisse in eine *DataTable* übertragen. Die Methode liefert die Anzahl der übertragenen Datensätze zurück. In vielen Programmen wird aber auf die Auswertung dieser Information verzichtet. (*dt.Rows.Count* liefert dieselbe Information.)

```
' Beispiel ADO.NET-classes\datatable-intro\Form1.vb

Dim da As New SqlDataAdapter("SELECT * FROM titles ORDER BY title", conn)
Dim dt As New DataTable

' DataTable füllen und Inhalt anzeigen
da.Fill(dt)
MsgBox(DataTableContent(dt))
```

Die Methode *DataTableConent* liest alle Zeilen und Spalten der *DataTable* und gibt eine Zeichenkette zurück.

```
Private Function DataTableContent(ByVal dt As DataTable) As String
  Dim sw As New IO.StringWriter
  Dim row As DataRow
  Dim n As Integer

  For Each row In dt.Rows
    For n = 0 To dt.Columns.Count - 1
      If row.Item(n) Is DBNull.Value Then
        sw.Write("NULL ")
      Else
        sw.Write(row.Item(n).ToString + " ")
      End If
    Next
    sw.WriteLine()
  Next
  Return sw.ToString()
End Function
```

Mehr Details zu den Eigenschaften und Methoden des *SqlDataAdapters* sowie weitere Beispiele folgen in Abschnitt 26.2. Werfen Sie auch einen Blick in Kapitel 29, das den von der Entwicklungsumgebung (*Visual Data Tools*) generierten Code für *DataAdapter* und typisierte *DataSets* beschreibt.

SqlCommandBuilder

Der folgende Code zum Einfügen einer neuen Zeile in die *DataTable* greift auf verschiedene *DataTable*-Methoden zurück. (Details zur *DataTable*-Klasse folgen in Abschnitt 26.2.)

```
'neuen Datensatz hinzufügen
Dim newrow As DataRow
newrow = dt.NewRow()
newrow!title = "test"
dt.Rows.Add(newrow)
```

Anschließend geht es darum, diese Änderungen in der zugrunde liegenden Datenbank zu speichern. Jetzt kommt wieder der *SqlDataAdapter* ins Spiel. Wenn Sie die für das Update erforderlichen SQL-Kommandos nicht selbst angeben möchten, nehmen Sie einen sogenannten ***SqlCommandBuilder*** zu Hilfe. Dieses Objekt erzeugt bei Bedarf die für das Update erforderlichen SQL-Kommandos. Das gelingt allerdings nur, wenn die zugrunde liegenden Daten relativ einfach strukturiert sind und die Primärschlüsselspalte enthalten. *da.Update(dt)* führt das Update schließlich durch.

```
'Änderungen speichern
Dim cb As New SqlCommandBuilder(da)
da.Update(dt)
```

Weitere Informationen zum Ändern von Daten in *DataTables* und zum Speichern dieser Änderungen in der Datenbank gibt Abschnitt 26.5. Eine Erklärung, wie die vom *SqlCommandBuilder* erzeugten Kommandos funktionieren und wie Sie bei Bedarf statt der automatisch generierten Kommandos eigene verwenden können, finden Sie in Abschnitt 27.4.