Tabelle 5.6: Die wichtigsten Mitglieder des Recordset-*Objekts (Fortsetzung)*

Mitglied	Bedeutung
MoveFirst-Methode	Bewegt den Datensatzzeiger auf den ersten Datensatz in der Datensatzgruppe.
MoveLast-Methode	Bewegt den Datensatzzeiger auf den letzten Datensatz in der Datensatzgruppe.
Move-Methode	Bewegt den Datensatzzeiger um eine bestimmte Anzahl Datensätze vor oder zurück.
MoveFirst-Methode	Bewegt den Datensatzzeiger auf den ersten Datensatz in der Datensatzgruppe.
Find-Methode	Bewegt den Datensatzzeiger auf den ersten bzw. nächsten Datensatz, der einem angegebenen Kriterium entspricht.
AbsolutePosition-Eigenschaft	Steuert einen Datensatz aufgrund seiner Position in der Datensatzgruppe an oder gibt die aktuelle Position des Datensatzzeigers zurück. Insbesondere die Abfrage wird nicht von jedem OLE DB-Provider unterstützt.
ActiveCommand-Eigenschaft	Steht für das *Command*-Objekt, sofern eines im Spiel ist, das das *Recordset*-Objekt erstellt hat.
ActiveConnection-Eigenschaft	Steht für die Verbindung (*Connection*-Objekt), über die der Datenbankzugriff durchgeführt wird.
Bookmark-Eigenschaft	Gibt die aktuelle Position des Datensatzzeigers (Datentyp *Double*) an oder setzt diesen auf eine gewünschte Position.
CacheSize-Eigenschaft	Legt die Anzahl Datensätze fest oder gibt sie zurück, die im Arbeitsspeicher zwischengespeichert werden. Aus Performance-Gründen kann es ratsam sein, diesen Wert (z.B. auf 100) zu begrenzen. Die Vorgabe beträgt 1, was bedeutet, daß alle Datensätze zurückgegeben werden.
AddNew-Methode	Fügt einen neuen Datensatz an die Datensatzgruppe an. Wo dieser erscheint, hängt von der aktuellen Sortierung ab. Übernommen wird der neue Datensatz über die *Update*-Methode. Wenn der gewählte Cursortyp keine dynamischen Mitgliedergruppen unterstützt, muß die *Requery*-Methode ausgeführt werden.
CancelUpdate-Methode	Bricht eine begonnene Aktualisierung der Datensatzgruppe, z.B. durch Aufruf der *AddNew*-Methode, wieder ab.
Clone-Methode	Erstellt eine Kopie des angegebenen *Recordset*-Objekts, wobei dieses den Nur-Lese-Modus erhalten kann. Der Vorteil ist, daß mehrere unabhängige Lesemarken verwaltet werden können.

Tabelle 5.6: Die wichtigsten Mitglieder des Recordset*-Objekts (Fortsetzung)*

Mitglied	Bedeutung
Close-Methode	Schließt das *Recordset*-Objekt.
CursorType-Methode	Gibt den Cursortyp an oder setzt ihn.
Delete-Methode	Löscht den aktuellen Datensatz.
Filter-Eigenschaft	Ermöglicht es, daß nur bestimmte Datensätze aufgrund einer oder mehrerer Bedingungen (WHERE-Klausel) in der Datensatzgruppe »sichtbar« sind.
GetRows-Eigenschaft	Liest eine vorgegebene Anzahl an Datensätzen in eine Feldvariable ein.
GetString-Eigenschaft	Gibt die komplette Datensatzgruppe in Gestalt einer Zeichenkette zurück.
LockType-Eigenschaft	Legt die Art der Sperre beim Mehrfachzugriff auf einen Datensatz an oder gibt die Art der aktuellen Sperre zurück.
MaxRecords-Eigenschaft	Legt die maximale Anzahl an Datensätzen fest, die eine Abfrage zurückgeben soll.
Open-Methode	Öffnet ein *Recordset*-Objekt, das zuvor geschlossen gewesen sein muß.
RecordCount-Eigenschaft	Steht für die Anzahl der Datensätze in der Datensatzgruppe. Bei einigen Cursortypen bzw. OLE DB-Providern ist der Wert stets 1 oder –1.
Requery-Methode	Liest alle Mitglieder der Datensatzgruppe erneut in das *Recordset*-Objekt ein.
Save-Methode	Speichert die Datensatzgruppe in einer lokalen Datei ab (wahlweise binär oder im XML-Format – nur ADO 2.1).
Sort-Eigenschaft	Sortiert die Datensätze der Datensatzgruppe nach einem bestimmten Kriterium (*SORT BY-Klausel*).
State-Eigenschaft	Gibt den aktuellen Zustand des *Recordset*-Objekts an (adStateClosed=0, adStateOpen=1).
Update-Methode	Aktualisiert die Datensatzgruppe, d.h. Daten, die sich im Zwischenspeicher befinden (etwa weil einem Feld ein neuer Wert zugewiesen wurde), werden in die Datenbank übertragen.

5.6 Das Field-Objekt

Ein *Recordset*-Objekt steht für eine Datensatzgruppe und einen Cursor, der sich um alles kümmert. Der Zugriff auf die Daten erfolgt über das *Fields*-Objekt, das für jedes Feld eines Datensatzes ein *Field*-Objekt bietet.

5.6.1 Die wichtigsten Mitglieder des Field-Objekts

Da das *Field*-Objekt einem »Datenbehälter« entspricht, haben die meisten Eigenschaften des *Field*-Objekts auch etwas mit den Daten zu tun. Mit *AppendChunk* und *GetChunk* gibt es auch zwei Methoden. Sie sind für das Speichern und Lesen von binären Daten zuständig, bei denen es keine feste Feldgröße gibt und daher ein Zugriff über die *Value*-Eigenschaft nicht möglich ist.

Vielleicht fragen Sie sich, warum das *Field*-Objekt in den Mittelpunkt gestellt wird, nicht aber das *Fields*-Objekt? Ganz einfach, weil eine Auflistung im allgemeinen keine eigenen Eigenschaften besitzt, sondern nur einige Standardmethoden, wie *Add* und *Remove*, sowie die Standardeigenschaft *Count* enthält.

Mitglied	Bedeutung
ActualSize-Eigenschaft	Gibt die tatsächliche Größe des Feldes in Byte an. Diese Anzahl läßt sich nicht immer bestimmen.
DefinedSize-Eigenschaft	Gibt die maximale, bei der Definition angegebene Größe des Feldes an.
Name-Eigenschaft	Enthält den Namen des Feldes.
OriginalValue-Eigenschaft	Gibt den Wert eines Feldes an, bevor Änderungen, z.B. über die *Update*-Methode, an dem Datensatz durchgeführt wurden.
Precision-Eigenschaft	Gibt die Genauigkeit des Feldes in Bezug auf die maximale Anzahl an Ziffern in einer Zahl oder Zeichen in einem Textfeld an. Dieser Wert hängt vom Datentyp des Feldes ab (*Type*-Eigenschaft). Welche Genauigkeiten die einzelnen Datentypen besitzen, wird in der MSDN-Hilfe aufgelistet.
Type-Eigenschaft	Gibt den Datentyp des Feldes an.
Type-Eigenschaft	Gibt den Datentyp des Feldes in Form einer Konstanten an (die z.B. vom Objektkatalog in der Kategorie *DataTypeEnum* aufgelistet wird).

Tabelle 5.7: Die wichtigsten Eigenschaften eines Field*-Objekts*

Tabelle 5.7: Die wichtigsten Eigenschaften eines Field-Objekts

Mitglied	Bedeutung
UnderlyingValue-Eigenschaft	Gibt den aktuellen Wert eines Feldes in der Datenbank an, der vom dem Wert von *OriginalValue* und *Value* abweichen kann. Diesen Wert verwendet die *Resync*-Methode, um den Wert von *Value* zu aktualisieren.
Value-Eigenschaft	Enthält den Inhalt des Feldes.

5.7 Das Command-Objekt

Mit *Connection*, *Recordset* und *Field* haben Sie die drei wichtigsten ADO-Objekte kennengelernt. Damit können Sie bereits die wichtigsten Datenbankoperationen und z.B. auch SQL-Abfragen durchführen. Und damit könnten wir es auch zunächst belassen. Doch zum einen sollen Sie in diesem Kapitel alle (sieben) ADO-Objekte kennenlernen, zum anderen werden Sie auch für das *Command*-Objekt schnell überaus nützliche Anwendungen finden. Das *Command*-Objekt steht über einer beliebigen Datenbankabfrage. Wenn Sie wissen möchten, wie viele Müller es in Ihrer Adreßdatenbank gibt, die seit mehr als einem Jahr nichts von sich haben hören lassen, dann legen Sie ein neues *Command*-Objekt an, weisen der *CommandText*-Eigenschaft das entsprechende SQL-Kommando zu und können anschließend durch Öffnen des *Command*-Objekts die Datenbankabfrage ausführen. Das geht mit einem *Recordset*-Objekt allerdings auch. Die eigentliche Daseinsberechtigung für *Command*-Objekte liegt in einem anderen Bereich. Zum einen kann die Quelle für ein *Command*-Objekt, neben einer Tabelle, eine sogenannte gespeicherte Abfrage (engl. »stored procedure«) sein, die es aber nur beim Microsoft SQL Server und nicht bei der Jet-Engine gibt (denken Sie stets daran, daß die ADOs nicht alleine für die Jet-Engine gemacht wurden). Zum anderen, und davon profitieren auch Programmierer, die mit der Jet-Engine arbeiten, lassen sich nur über das *Command*-Objekt SQL-Abfragen durchführen, die mit Parametern arbeiten (mehr dazu in Kürze).

Die folgende Befehlsfolge zeigt, wie sich über ein *Command*-Objekt ein einfaches *UPDATE*-Kommando durchführen läßt (mehr zu den SQL-Kommandos in Kapitel 7). Ein Ergebnis gibt es bei dieser Form der Abfrage nicht, allerdings teilt uns ADO auf Wunsch mit, wie viele Datensätze von der Änderung betroffen wurden.

```
Dim Cmd As ADODB.Command
Set Cmd = New ADODB.Command
Set Cn = New ADODB.Connection
```

```
With Cn
    .Provider = "Microsoft.Jet.OLEDB.3.51"
    .ConnectionString = _
       "Data Source=C:\Eigene Dateien\Testdb.mdb"
    .Open
End With

With Cmd
    .CommandText = _
        "UPDATE Mitarbeiter SET Gehalt = Gehalt * 1.2 WHERE _
         Eintrittsdatum < #1/1/1990#"
    .CommandType = adCmdText
    Set .ActiveConnection = Cn
End With
Cmd.Execute
Cn.Close
```

Wenn Sie der Meinung sind, daß die Beispiele in diesem Buch immer umfangreicher werden, haben Sie natürlich recht. Einen Teil ihres Umfangs verdanken sie allerdings dem Umstand, daß jedesmal ein *Connection*-Objekt angelegt wird, was für die Erklärung des Sachverhalts streng genommen nicht notwendig wäre. Dafür können Sie das Beispiel, z.B. innerhalb von *Form_Load*, direkt ausführen und müssen nicht jedesmal den allgemeinen Ablauf zurechtlegen. Richten Sie im obigen Beispiel Ihr Augenmerk auf das *Command*-Objekt, das über die Objektvariable *Cmd* angelegt wird. Drei Dinge benötigt das *Command*-Objekt, damit es aktiv werden kann:

1. Den Befehlstext.
2. Eine Angabe darüber, wie der Befehlstext zu interpretieren ist (z.B. als SQL-Kommando).
3. Eine aktive Verbindung.

Sind alle drei Angaben gemacht, wird das *Command*-Objekt über seine *Execute*-Methode zur Ausführung gebracht.

Auch wenn es nicht zwingend notwendig ist, sollten Sie über die *CommandType*-Eigenschaft den Typ des *Command*-Objektes festlegen. Sie ersparen es ADO damit, daß es versucht, den Typ selber herauszufinden, was unnötig Zeit kostet.

Wäre es nicht nett, wenn wir erfahren könnten, wie viele Datensätze von der Ausführung des Kommandos betroffen wurden? Bezogen auf das obige Beispiel wurde ein *UPDATE*-Kommando ausgeführt, das aufgrund der Einschränkungsklausel nicht alle Datensätze betrifft. Um herauszubekommen, wie viele Datensätze tatsächlich geändert wurden, muß man für das Argument *RecordsAffected* der *Execute*-Methode den Namen einer Variablen übergeben, in der diese Anzahl abgelegt wird.

In diesem Beispiel wird von dem Argument *RecordsAffected* Gebrauch gemacht, um die Anzahl der veränderten Datensätze zu erfahren.

```
Dim Anzahl As Long
' Irgendwelche Befehle
Cmd.Execute RecordsAffected:=Anzahl
MsgBox Prompt:= "Es sind " & Anzahl & " Datensätze betroffen"
```

Tabelle 5.8: Die wichtigsten Mitglieder des Command-*Objekts*

Mitglied	Bedeutung
ActiveConnection-Eigenschaft	Gibt das *Connection*-Objekt an, über das das *Command*-Objekt ausgeführt wird.
CommandText-Eigenschaft	Enthält den Kommandotext, z.B. das SQL-Kommando oder den Namen der Abfrage.
CommandText-Eigenschaft	Enthält den Text des auszuführenden Kommandos.
CommandType-Eigenschaft	Gibt an, welche Sorte von *Command*-Objekt ausgeführt wird (siehe Tabelle 5.9)
CreateParameter-Methode	Legt für Abfragen mit Parametern ein neues *Parameter*-Objekt an.
Execute-Methode	Führt das *Command*-Objekt aus und gibt gegebenenfalls ein *Recordset*-Objekt vom Typ »Forward Only» zurück.
Parameters-Eigenschaft	Steht für alle *Parameter*-Objekte der Abfrage.

Tabelle 5.9: Die verschiedenen Typen des Command-*Objekts*

Konstante	Das Command-Objekt basiert auf ...
adCmdFile	... einem Wert für die *CommandText*-Eigenschaft, der aus einer Datei stammt.
adCmdStoredProc	... einer Stored Procedure (gibt es nicht bei der Jet-Engine)
adCmdTable	... direkt auf einer Tabelle.
adCmdText	... auf einem SQL-Kommando oder einer anderen textuellen Beschreibung (etwa einer Abfrage).
adCmdUnknown	... einem beliebigen Objekt bzw. ist nicht bekannt.

5.8 Das Parameter-Objekt – Abfragen mit Parametern

Mit dem Erlernen von SQL ist es wie mit dem Erlernen einer Programmiersprache. Zunächst ist man froh, das Prinzip des *SELECT*-Kommandos verstanden zu haben. Doch sehr schnell möchte man mehr. So stellt es sich bei normalen Abfragen schnell heraus, daß diese ein wenig unflexibel sind. Schauen Sie sich dazu folgende SQL-Abfrage an:

```
SELECT * FROM Fahrzeugdaten WHERE Preis > 30000
```

Diese Abfrage gibt alle Datensätze zurück, bei denen das Feld *Preis* einen Wert größer als 30000 besitzt. Doch was ist, wenn der Preis in der Abfrage variabel gehalten werden soll? Nun, auch das ist kein Problem. Wie Sie es in Kapitel 7 lernen werden, muß dazu lediglich die Zahl durch eine Variable ersetzt werden:

```
"SELECT * FROM Fahrzeugdaten WHERE Preis > " & Preislimit
```

Um das SQL-Kommando von der Variablen zu trennen, deren Wert eingesetzt werden soll, wurde das Kommando in Anführungsstriche gesetzt. Doch ist diese Lösung optimal? Nicht ganz, denn dieses Kommando muß bei jeder Ausführung von der Datenquelle (bzw. dem OLE DB-Provider) neu analysiert und übersetzt werden, was entsprechend Ausführungszeit kostet. In einer Produktionsdatenbank möchte man, daß häufig auszuführende Abfragen möglichst schnell ausgeführt werden. Und genau dafür sind jene Abfragen da, die bereits in der Datenbank enthalten sind. Eine solche Abfrage wird einmal erstellt und liegt anschließend in übersetzter, optimierter Form (und nicht mehr als SQL-Text) in der Datenbank vor. Der Microsoft SQL Server (und die MSDE von Microsoft Access 2000 übrigens auch) bietet dazu die sehr leistungsfähigen *Stored Procedures*, die es bei der Jet-Engine leider nicht gibt. Hier besteht dagegen die Möglichkeit, Abfragen in der Datenbank zu speichern und in die Abfrage eine oder mehrere Variablen einzubauen, die allerdings nicht als Variablen, sondern als Parameter bezeichnet werden. Geöffnet wird die Abfrage wie eine Tabelle über ein *Recordset*-Objekt (oder über die *Execute*-Methode eines *Command*-Objekts). Nun gilt es aber noch, das »Problem« der Parameterübergabe zu lösen, denn wenn die Abfrage, wie im obigen Beispiel, mit einem (oder mehreren) variablem Wert durchgeführt wird, muß es auch eine Möglichkeit geben, diesen variablen Wert vor dem Öffnen der Abfrage zu übergeben. Und genau diese Möglichkeit bietet das *Parameter*-Objekt, das über die *Parameters*-Eigenschaft eines *Command*-Objekts zur Verfügung gestellt wird.

Die folgenden Befehlszeilen öffnen die Datenbank *Fuhrpark.mdb* und führen eine dort (mit Hilfe von Microsoft Access) eingetragene Abfrage aus. Diese Abfrage mit dem Namen *AbfrageMitParameter* erwartet als einzigen Parameter eine Zahl, die für den Preis steht. Anschließend gibt sie jene Datensätze zurück, bei denen das Feld *Preis* unter diesem Wert liegt.

```
Option Explicit
Private Cn As ADODB.Connection
Private Rs As ADODB.Recordset
Private Cmd As ADODB.Command
Private P As ADODB.Parameter
Const DBPfad = "C:\Eigene Dateien\Fuhrpark.mdb"

Private Sub Form_Load()
    Set Cn = New ADODB.Connection
    With Cn
     .Provider = "Microsoft.Jet.OLEDB.3.51"
     .ConnectionString = "Data Source=" & DBPfad
     .Open
    End With
    Set Cmd = New ADODB.Command
    Set P = New ADODB.Parameter
    With P
    .Name = "Preislimit"
    .Value = InputBox("Preislimit:")
    .Type = adCurrency
    End With
    With Cmd
     .Parameters.Append P
     .CommandText = "AbfrageMitParametern"
     .CommandType = adCmdUnknown
     .ActiveConnection = Cn
    End With
    Set Rs = Cmd.Execute
    Do While Not Rs.EOF = True
       Debug.Print Rs.Fields("Modellname").Value
       Rs.MoveNext
    Loop
    Rs.Close
    Cn.Close
End Sub
```

Nachdem die Abfrage über die *Execute*-Methode ausgeführt wurde, wird die Ergebnisdatensatzmenge der Variablen *Rs* zugewiesen. Die *RecordsAffected*-Eigenschaft läßt sich in diesem Fall leider nicht verwenden, um die Anzahl der zurückgegebenen Datensätze abzufragen. Auch die *RecordCount*-Eigenschaft hilft hier nichts, da das geöffnete *Recordset* vom Typ »ForwardOnly« ist.

5.9 Das Errors-Objekt für die Fehlerkontrolle

Ein eher unscheinbares Objekt wird bei der Umsetzung Ihrer ersten Anwendung auf der Basis von ADO schnell zu einem unverzichtbaren Helfer werden: das *Errors*-Objekt. Tritt während einer Datenbankoperation nämlich ein Fehler auf, stehen alle Fehlermeldungen und deren Fehlernummern über die *Errors*-Auflistung zur Verfügung. Zum Abfangen von Laufzeitfehlern sind auch bei den ADOs der *On Error Goto*-Befehl und das *Err*-Objekt zuständig, dessen *Description*-Eigenschaft auch in diesem Fall den Fehlertext angibt. Kommt es allerdings vor, daß durch eine Operation mehrere Fehler ausgelöst wurden, reicht das *Err*-Objekt nicht mehr aus. Möchte man nicht nur den letzten Fehler, sondern alle aufgetretenen Fehler sehen, muß man dazu die *Errors*-Auflistung bemühen. Diese steht über das *Connection*-Objekt zur Verfügung.

Die folgende Befehlsfolge gibt die Fehlermeldungen aller aufgetretenen Fehler aus:


```
Dim E As Error
For Each E In Cn.Errors
    Debug.Print E.Description
Next
```

Über die *Clear*-Methode des *Errors*-Objekt werden alle anstehenden Fehlermeldungen gelöscht.

Mitglied	Bedeutung
Description-Eigenschaft	Enthält den Fehlertext.
NativeError-Eigenschaft	Gibt die interne Fehlernummer des OLE DB-Providers zurück.
Number-Eigenschaft	Enthält die Fehlernummer.
Source-Eigenschaft	Gibt den Namen der Quelle bzw. des Objekts an, die den Fehler verursacht hat.

Tabelle 5.10: Die wichtigsten Mitglieder des Error-*Objekts*

5.10 Das Property-Objekt

Zum Schluß soll Ihnen mit dem *Property*-Objekt auch das letzte ADO-Objekt nicht vorenthalten werden, wenngleich es recht spezieller Natur ist und am Anfang nur sehr selten eine Rolle spielen dürfte. Natürlich können die sieben ADO-Objekte nicht sämtliche Eigenschaften einer Datenquelle beschreiben. Viele dieser Eigenschaften sind so speziell auf eine Datenquelle zugeschnitten, daß es nicht sinnvoll wäre, dafür eigenständige Eigenschaften einzuführen, die vielleicht nur bei einem Bruchteil der Datenquellen eine Bedeutung hätten. Alle diese Spezialeigenschaften sowie die Standardeigenschaften werden über die *Properties*-Eigenschaft der ADO-Objekte *Connection*, *Command*, *Parameter* und *Recordset* zur Verfügung gestellt. Die *Properties*-Eigenschaft ist eine Auflistung, die für eines oder mehrere *Property*-Objekte steht. Jedes einzelne *Property*-Objekt steht für eine Eigenschaft eines ADO-Objekts und verfügt u.a. über eine *Name*- und eine *Value*-Eigenschaft. Die einzelnen *Property*-Objekte werden entweder über ihren Index oder direkt über ihren Namen angesprochen:

```
?Cn.Properties("Provider Version").Value
03.52.1527
```

Welche *Property*-Eigenschaften ein ADO-Objekt zur Verfügung stellt, hängt vom OLE DB-Provider (und unter Umständen auch von dessen Versionsnummer) ab. So wird es bei einer SQL-Server-Datenbank andere Properties geben als bei einer simplen Textdatei. Da sich die über *Property*-Objekte zur Verfügung gestellten Eigenschaften im Prinzip beliebig ergänzen lassen, werden sie auch als dynamische Eigenschaften bezeichnet.

Die meisten Eigenschaften sind Nur-Lese-Eigenschaften. Eine Auflistung aller dynamischen Eigenschaften eines OLE DB-Providers gibt es (anscheinend) nicht. Die meisten Eigenschaften sind aufgrund ihrer »sprechenden« Namen aber selbsterklärend.

Die Namen der Tabellen und andere »Schema«-Daten erhält man nicht über die *Properties*-Eigenschaft. Dafür gibt es beim *Connection*-Objekt die *OpenSchema*-Methode, die diese Informationen zurückliefert. Der folgende Befehl überträgt alle Tabelleninformationen in ein *Recordset*-Objekt:

```
Set Rs = Cn.OpenSchema (Schema:=adSchemaTables)
```

Bei Microsoft Access 2000 gibt es mit der bereits erwähnten *ADOX*-Erweiterung eine sehr viel elegantere und weitreichendere Möglichkeit, auf die Schema-Daten zuzugreifen.

5.11 Die Ereignisse der ADO-Objekte

Die ADO-Objekte *Connection* und *Recordset* können Ereignisse auslösen. Das bedeutet konkret, daß sich ein *Connection*-Objekt »meldet«, nachdem eine Verbindung hergestellt wurde, oder ein *Recordset*-Objekt ein Ereignis auslöst, nachdem ein neuer Datensatz angesteuert wurde. Allerdings sind ADO-Objekte keine Steuerelemente, bei denen die Ereignisverarbeitung bereits »fest eingebaut« ist. Die Ereignisweiterleitung muß vielmehr erst mit dem Visual-Basic-Programm »verbunden« werden. Das geschieht, indem die Objektvariable, über die der Zugriff auf das ereignisauslösende Objekt erfolgt, mit dem Zusatz *WithEvents* deklariert wird. In der Regel geschieht dies in zwei Schritten:

1. Deklaration einer ADO-Variablen im Allgemein-Teil des Formulars mit *Private* und *WithEvents*:

   ```
   Private WithEvents RsFahrzeugbestand As ADODB.Recordset
   ```

   Durch das Schlüsselwort *WithEvents* werden die Ereignisse des *Recordset*-Objekts an die Variable *RsFahrzeugbestand* »umgeleitet«.

2. Instanzieren des Objekts innerhalb von *Form_Load*:

   ```
   Set RsFahrzeugbestand = New ADODB.Recordset
   ```

Auch wenn dies (wie immer) nicht der einzige Weg ist, wird es in allen Beispielprogrammen des Kapitels so gemacht.

Anschließend können Sie, wie es beim Einbeziehen von Ereignissen, die nicht sichtbare COM-Objekte (also alle COM-Objekte, bei denen es sich nicht um ActiveX-Steuerelemente handelt) auslösen, üblich ist, in der Objektliste im Programmcodefenster den Eintrag »RsFahrzeugbestand« und in der rechten Auswahlliste eines der zur Verfügung stehenden Ereignisse auswählen.

5.11.1 Allgemeines zu den ADO-Ereignissen

Die ADO-Ereignisse dürften auch auf erfahrene Visual-Basic-Programmierer am Anfang ein wenig »merkwürdig« wirken. Da wäre zum einen der Umstand, daß einige Ereignisnamen scheinbar »doppelt« vorkommen und sich lediglich durch die Vorsilbe »Will« und den Anhang »Completed« unterschei-

den. Und da wären die zahlreichen Argumente, die beim Aufruf einer Ereignisprozedur übergeben werden, und die am Anfang reichlich undurchsichtig wirken. Zum Glück ist alles dann doch relativ einfach[1]. Man muß sich nur ein wenig intensiver mit dem Prinzip der ADO-Ereignisse beschäftigen.

Einige der insgesamt 21 ADO-Ereignisse treten paarweise auf. Das bedeutet konkret, daß vor der Ausführung der betreffenden Aktion, wie etwa der Wechsel des aktuellen Datensatzes auf einen anderen Datensatz, ein *Will*-Ereignis ausgelöst wird. Dadurch teilt ADO dem (Visual-Basic-)Programm mit, daß die Aktion durchgeführt werden soll, es zu diesem Zeitpunkt aber die Möglichkeit gibt, sie durch Setzen des *adStatus*-Parameters auf *adCancel* abzubrechen. Geschieht dies nicht und treten keine anderen Umstände ein, die die Ausführung der Aktion verhindern, wird die Aktion durchgeführt und anschließend ein *Completed*-Ereignis ausgelöst. Hier ein einfaches Beispiel. Zeigt der Datensatzzeiger auf den ersten Datensatz eines *Recordset*-Objekts und wird eine *MoveNext*-Methode ausgeführt, treten beim *Recordset*-Objekt nacheinander die folgenden Ereignisse auf:

- *WillMove*
- *MoveComplete*

Über den *adReason*-Parameter teilt ADO allen, die es interessiert, mit, daß der Grund für die beiden Ereignisse die *MoveNext*-Methode ist. Soll der Datensatzzeiger aus irgendeinem Grund nicht bewegt werden, muß der *adStatusParameter* in der *WillMove*-Prozedur den Wert *adStatusCancel* erhalten. Dies führt zum Abbruch der Operation und zu einem Laufzeitfehler, der entsprechend abgefangen werden muß. Wird *adStatus* dagegen innerhalb von *WillMove* auf den Wert *adStatusUnwantedEvent* gesetzt, wird lediglich die Ereignismeldung (beim nächsten Mal) unterdrückt, die Operation wird dagegen ausgeführt. Außerdem wird ein *MoveComplete*-Ereignis ausgelöst.

Die zweite Besonderheit der ADO-Ereignisse, die am Anfang das Verständnis ein wenig erschwert, ist die Verschachtelung der Ereignisse. Auch hier ein konkretes Beispiel. Wird ein Feld geändert, treten nacheinander die Ereignisse *WillChangeRecord*, *WillChangeField*, *FieldChangeCompleted* und *RecordchangeCompleted* auf. In jedem der beiden *Will*-Ereignisse besteht wieder die Möglichkeit, die jeweilige Aktion abzubrechen. Stellt man fest, z.B. nach einer Überprüfung der eingegebenen Daten, daß zwar der Datensatz, nicht aber ein einzelnes Feld geändert werden soll, muß der *adStatus*-Parameter im *WillChangeField*-Ereignis den Wert *adCancel* erhalten.

---

1 Wenngleich es bei den ADO-Ereignissen offenbar ein paar Ungereimtheiten zu geben scheint.

Soll dagegen der komplette Datensatz unverändert bleiben, wird statt dessen der *adStatus*-Parameter im *WillChangeRecordset*-Ereignis gesetzt.

Ereignis	**Wann tritt es ein?**
EndOfRecordset	Wenn der Datensatzzeiger entweder über den Beginn oder das Ende der Datensatzgruppe bewegt wurde. Welcher Fall eingetreten ist, kann über die Eigenschaften BOF und EOF abgefragt werden.
FetchComplete	Wenn alle Datensätze in einer asynchronen Abfrage geholt wurden und die Abfrage damit beendet ist.
FetchProgress	Um einen Zwischenstand bezüglich der Anzahl der bereits zurückgegebenen Datensätze in einer asynchronen Abfrage zu melden.
WillChangeField	Wenn eine Operation im Begriff ist, den Inhalt eines oder mehrerer Felder zu ändern.
FieldChangeComplete	Wenn der Inhalt eines oder mehrerer Felder geändert wurde.
WillChangeRecord	Wenn eine Operation im Begriff ist, den Inhalt des aktuellen Datensatzes zu ändern.
RecordChangeComplete	Wenn der Inhalt des aktuellen Datensatzes geändert wurde.
WillMove	Wenn eine Operation im Begriff ist, den Datensatzzeiger auf einen anderen Datensatz zu bewegen.
MoveCompleted	Wenn der Datensatzzeiger auf einen anderen Datensatz bewegt wurde
EndOfRecordset	Wenn der Datensatzzeiger über das Ende oder den Anfang der Datensatzgruppe hinaus bewegt wurde.
WillChangeRecordset	Wenn eine Operation im Begriff ist, die aktuelle Datensatzgruppe zu ändern.
RecordsetChangeComplete	Wenn die aktuelle Datensatzgruppe geändert wurde.

Tabelle 5.11: Die Ereignisse eines Recordset-*Objekts*

Ereignis	**Wann tritt es ein?**
BeginTransComplete	Nachdem eine Transaktion über die *BeginTrans*-Methode gestartet wurde.
CommitTransComplete	Nachdem eine Transaktion über die *CommitTrans*-Methode bestätigt wurde.
ConnectComplete	Wenn die Verbindung zu einer Datenquelle hergestellt wurde.

Tabelle 5.12: Die Ereignisse eines Connection-*Objekts*

Tabelle 5.12: Die Ereignisse eines Connection*-Objekts (Fortsetzung)*

Ereignis	Wann tritt es ein?
Disconnect	Wenn die Verbindung zu einer Datenquelle wieder beendet wurde.
ExecuteComplete	Wenn ein über die *Execute*-Methode ausgeführtes SQL-Kommando beendet wurde.
InfoMessage	Wenn sich der OLE DB-Provider beim Herstellen einer Verbindung mit einer Mitteilung bemerkbar machen möchte.
RollbackTransComplete	Nachdem eine Transaktion über die *Rollback*-Methode abgebrochen und zurückgesetzt wurde.
WillConnect	Wenn die Verbindung zu einer Datenquelle hergestellt werden soll.
WillExecute	Wenn ein SQL-Kommando über die *Execute*-Methode ausgeführt werden soll.

5.11.2 Die Parameter der ADO-Ereignisse

Die Parameter der verschiedenen ADO-Ereignisse können einem schon ein wenig den (virtuellen) Angstschweiß in die Stirn treiben. Sie geben sich beim ersten Kennenlernen kryptisch und alles andere als einladend. Der Prozedurkopf der *WillChangeField*-Ereignisprozedur sollte als Beleg genügen:

```
Sub Rs_WillChangeField (ByVal cFields As Long, _
    ByVal Fields As Variant, _
    adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

Dennoch sind sie der Schlüssel zum Verständnis der ADO-Ereignisse, da sie dem Programmierer die Möglichkeit geben, das scheinbar Unabwendbare, wie das Zurückschreiben von geänderten Daten in die Datenbank, doch noch zu verhindern. Es hilft also (wieder einmal) nichts, man sollte so ungefähr über die Bedeutung der einzelnen Parameter Bescheid wissen. Am besten ist es daher, sich zuerst einmal eine Übersicht zu verschaffen. Damit diese Übersicht auch richtig übersichtlich wird, werden die Parameter der *Will*- und der *Completed*-Ereignisse getrennt behandelt, auch wenn es zwischen beiden Gruppen natürlich mehr Übereinstimmungen als Unterschiede gibt.

Alle weiteren Referenzen, etwa eine Auflistung der für die Parameter *adStatus* und *adReason* in Frage kommenden Werte, finden Sie im Anhang B. Hier noch ein kleiner Tip. Um etwa herauszufinden, welche Bedeutung

der Wert 11 des *adReason*-Parameters hat, muß man nicht in der Hilfe nachschlagen, sondern man erfährt dies auch über den Objektkatalog.

Parameter	Datentyp	Was hat es zu bedeuten?
CFields	*Long*	Anzahl der Felder, die von der Operation betroffen sind.
Fields	*Variant*	Die Felder als eine Auflistung von *Field*-Objekten, die von der Änderung betroffen sind.
AdStatus	*EventStatusEnum*	Gibt den aktuellen Status der Operation (*OK*, *Fehler* oder *kann nicht abgebrochen werden*) an.
PRecordset	*Recordset*	Referenz auf ein *Recordset*-Objekt, über das alle Datensätze der Datensatzgruppe zur Verfügung stehen.
AdReason	*EventReasonEnum*	Enthält den Grund der angeforderten Änderung in Gestalt einer Konstanten. *adReason* besitzt den Wert *adRsnFirstChange,* wenn der Datensatz zum ersten Mal geändert wurde.

Tabelle 5.13: Diese Parameter werden den Will-*Ereignissen übergeben*

Parameter	Datentyp	Was hat es zu bedeuten?
pError	*Error*	Enthält die Fehlerinformationen, falls die Operation abgebrochen wurde. In diesem Fall besitzt *adStatus* den Wert *adStatusErrorsOccurred*.
adStatus	*EventStatusEnum*	Gibt an, ob die Operation ausgeführt werden konnte oder nicht.
pRecordset	*Recordset*	Referenz auf ein *Recordset*-Objekt, über das alle Datensätze der Datensatzgruppe zur Verfügung stehen.
adReason	*EventReasonEnum*	Enthält den Grund der angeforderten Änderung in Gestalt einer Konstanten.
cFields	*Long*	Anzahl der Felder, die von der Operation betroffen sind.
Fields	*Variant*	Die Felder als eine Auflistung von *Field*-Objekten, die von der Änderung betroffen sind.

Tabelle 5.14: Diese Parameter werden den Completed-*Ereignissen übergeben*

5.11.3 Verhindern, daß ein geändertes Feld in die Datenbank übernommen wird

Das Zusammenspiel der verschiedenen ADO-Ereignisse wird am besten an einem kleinen Beispiel deutlich. Wer es zum ersten Mal geschafft hat, daß der Inhalt einer Tabelle in gebundenen Steuerelementen angezeigt wird, wird vermutlich noch nicht darauf achten. Geht es dagegen an die »richtige« Datenbankprogrammierung, wird der Umstand sofort offensichtlich. Wird nämlich in einem gebundenen Steuerelement eine Änderung des angezeigten Wertes durchgeführt und wird der Datensatzzeiger, etwa über ein Datensteuerelement, auf einen anderen Datensatz bewegt, wird die Änderung in die Datenbank übernommen. Und zwar ohne eine Bestätigung vom Anwender zu verlangen. Da dieses Verhalten aber nur selten erwünscht ist, muß es eine Möglichkeit geben, dies zu verhindern. Eine solche Möglichkeit gibt es auch. Sie hat, wie Sie es sich vermutlich schon gedacht haben, mit den Ereignissen eines *Recordset*-Objekts zu tun. Wann immer der Inhalt eines Feldes geändert wird, treten folgende Ereignisse der Reihe nach ein:

- *WillChangeRecord*
- *WillChangeField*
- *FillChangeComplete*
- *RecordChangeComplete*

Geschah das Auslösen der Ereigniskette in der Absicht, den aktuellen Datensatzzeiger auf den nächsten Datensatz zu bewegen, treten noch vier weitere Ereignisse ein:

- *WillMove*
- *WillChangeRecord*
- *RecordChangeComplete*
- *MoveComplete*

Richten Sie Ihr Augenmerk auf die ersten vier Ereignisse. Damit die geänderten Feldinhalte nicht in die Datenbank übernommen werden, müssen Sie in einem der ersten beiden *Will*-Ereignisse eingreifen und dort den Prozedurparameter *adStatus* auf den Wert *adCancel* setzen. Soll nur ein bestimmtes Feld nicht übernommen werden, entscheiden Sie sich über das *WillChangeField*-Ereignis. Soll der komplette Datensatz nicht übernommen werden, entsprechend für das *WillChangeRecord*-Ereignis. Der eigentliche Abbruch der Aktion geschieht durch die folgende Anweisung, die z.B. innerhalb von *WillChangeRecord* ausgeführt wird:

```
Sub Rs_WillChangeRecord _
    (ByVal adReason As ADODB.EventReasonEnum, _
     ByVal cRecords As Long, _
     adStatus As ADODB.EventStatusEnum, _
     ByVal pRecordset As ADODB.Recordset)
    If Abbruch =True Then
       adStatus = adStatusCancel
    End If
End Sub
```

Die Operation wird dadurch nicht durchgeführt. Zwar wird auch in diesem Fall ein *FieldChangeCompleted*-Ereignis ausgelöst, diesmal besitzt *adStatus* aber den Wert *adStatusErrorsOccurred*. Zusätzlich zeigt der übergebene *Error*-Parameter den Fehler an, wobei die Fehlernummer entweder von den ADO-Objekten oder direkt vom OLE DB-Provider stammt. Außerdem wird ein Laufzeitfehler in jener Prozedur ausgelöst, in der die fehlerverursachende Aktion gestartet wurde (sofern dies durch Programmcode geschah).

Im Grunde ist also alles ganz einfach. Doch wenn Sie das Ganze in der Praxis ausprobieren, werden Sie feststellen, daß auf einmal ein Laufzeitfehler mit dem merkwürdig anmutenden Fehlertext »Die Änderung wurde während der Benachrichtigung abgebrochen; es wurden keine Spalten geändert« erscheint und der Datensatzzeiger nicht bewegt wurde. Ja mehr noch, wenn Sie erneut probieren, zu einem anderen Datensatz zu wechseln, erscheint auch die Fehlermeldung erneut. Ursache für dieses Verhalten ist der Umstand, daß durch den Abbruch der Operation nicht die *DataChanged*-Eigenschaft der betroffenen gebundenen Steuerelemente auf *False* gesetzt wurde. Bleibt diese Eigenschaft gesetzt, geht ADO fälschlicherweise davon aus, daß erneut ein Wert geändert wurde. Um den Laufzeitfehler zu vermeiden, muß innerhalb des entsprechenden *Complete*-Ereignisses sichergestellt werden, daß die *DataChanged*-Eigenschaft bei jedem gebundenen Steuerelement den Wert *False* besitzt. Und damit dies nur dann geschieht, wenn ein Abbruch angefordert wurde, muß beim Parameter *adStatus* vorher geprüft werden, ob dieser den Wert *adStatusErrorsOccurred* besitzt:

```
Sub Rs_RecordChangeComplete _
    (ByVal adReason As ADODB.EventReasonEnum, _
     ByVal cRecords As Long, _
     ByVal pError As ADODB.Error, _
     adStatus As ADODB.EventStatusEnum, _
     ByVal pRecordset As ADODB.Recordset)
    If adStatus = adStatusErrorsOccurred Then
       DataChangedZurücksetzen
```

```
    End If
End Sub
```

Bei *DataChangedZurücksetzen* handelt es sich um eine kleine Prozedur, die das Zurücksetzen der *DataChanged*-Eigenschaft aller beteiligten Steuerelemente durchführt. Ein wenig umständlich, doch anscheinend nicht anders zu lösen.

5.11.4 Abschließende Anmerkungen zu den ADO-Ereignissen

Folgende Dinge müssen im Zusammenhang mit den ADO-Ereignissen angemerkt werden:

- ✘ Auch wenn einige Ereignisse paarweise auftreten, müssen sie nicht paarweise eingesetzt werden. Es ist also durchaus möglich, in einem *Will*-Ereignis eine Aktion durchzuführen und das dazugehörige *Complete*-Ereignis zu ignorieren.
- ✘ Das Abbrechen einer Operation mit *adStatus=adCancel* führt zu einem Laufzeitfehler, wenn die Ursache für die *WillChangeField*- bzw. *WillChangeRecords*-Ereignisse nicht wieder beseitigt wurde. Bei gebundenen Steuerelementen geschieht dies durch Setzen von *DataChanged* auf den Wert *False*.
- ✘ Ist ein *Recordset*-Objekt an ein Steuerelement gebunden, tritt das erste *Will*-Ereignis bereits ein, sobald das Steuerelement den Fokus verliert. Das ist anders als bei den beiden Ereignissen *Reposition* und *Validate* des alten Datensteuerelements[1], die explizit über die Schaltflächen des Datensteuerelements ausgelöst wurden.

5.12 ADO und Microsoft Access 2000

Mit der Einführung von Microsoft Access 2000, dem Nachfolger von Microsoft Access 97, gibt es bei den ADOs wichtige Neuerungen. Doch keine Sorge, am Inhalt dieses Kapitels ändert sich nichts[2]. Mit Microsoft Access 2000 steht unter dem Namen »Microsoft ADO Ext. 2.1 for DDL and Security« eine zusätzliche ADO-Bibliothek für die Jet-Engine zur Verfügung, mit der z.B. ein Zugriff auf die Sicherheitsfunktionen oder auf die Datenbank- und Tabellenstruktur (die DDL-Funktionalität bezogen auf SQL –

1 Wie schön einfach hier doch alles war.

2 Es wurde noch mit Schwerpunkt auf Microsoft Access 97 geschrieben.

siehe Kapitel 7; DAO-Programmierer werden hier z.B. das Pendant zu TableDefs & Co. finden) möglich ist. Es ist wichtig zu verstehen, daß diese neue ADOX-Bibliothek als Ergänzung und nicht als Ersatz der ADODB-Bibliothek fungiert.

5.13 Zusammenfassung

Die Active(X) Data Objects (ADO) gehören zweifelsohne zu den größten Herausforderungen, denen sich angehende Visual-Basic-Datenbankprogrammierer stellen müssen. Alles fing in diesem Buch so einfach an. Tabellen, Datensätze, Felder und sicherlich auch Schlüssel sind Konzepte, die sich direkt an dem orientieren, um was es bei der Datenbankprogrammierung im Kern geht. Und nun tauchen mit den ADOs auf einmal neue Begriffe, Programmiertechniken und eine am Anfang verwirrende Auswahl an Möglichkeiten auf, die alles scheinbar unnötig kompliziert machen. Doch es hilft alles nichts. Nur wer die ADO-Objekte wirklich versteht, kann alle Möglichkeiten der Datenbankprogrammierung nutzen, die Visual Basic ab Version 6.0 zu bieten hat. Microsoft hat die Latte bewußt sehr hoch gesetzt, denn jede (unnötige) Vereinfachung in der Datenbankschnittstelle würde eine Einschränkung bezüglich der Flexibilität bedeuten, die die meisten Programmierer nicht in Kauf nehmen möchten. Eines darf nicht vergessen werden, die ADO-Objekte richten sich weniger an den typischen Access-Programmierer, der mit seiner »kleinen« Datenbank zufrieden ist und nicht beabsichtigt, daran etwas in absehbarer Zeit zu ändern. Die ADO-Objekte richten sich an jene Programmierer, die unterschiedliche Datenquellen ansprechen müssen, wobei es sich bei diesen Datenquellen vornehmlich um sehr große Datenbanken handelt, und daher auf ein flexibles Modell angewiesen sind. Im Grunde müßte es speziell für die Access-Programmierer ein eigenes, sehr viel enger mit der Jet-Engine verbundenes und damit einfacheres Objektmodell (eine Art »ADO Light«) geben. Nun, zum einen gibt es ein solches Objektmodell mit den DAOs (der Nachteil ist, daß diese Objekte von den modernen Datenbankwerkzeugen von Visual Basic 6.0 nicht unterstützt werden). Zum anderen wäre es kein Problem, ein solches Objektmodell (das ginge auch in Visual Basic, wenngleich C++ dafür besser geeignet wäre) auf der Basis von OLE DB zu programmieren. ADO ist lediglich eine Variante von theoretisch unendlich vielen.

5.14 Wie geht es weiter?

In diesem Kapitel haben Sie mit den ADOs das Fundament der Datenprogrammierung unter Visual Basic kennengelernt. Haben Sie die ADOs im Prinzip verstanden, haben Sie die größte Hürde auf dem Weg zum Visual-Basic-Datenbankprogrammierer genommen. Die folgenden Kapitel führen »nur noch« einige praktische Anwendungen vor, die allesamt auf den ADOs basieren. In Kapitel 6 wird der Datenumgebungs-Designer vorgestellt, der den Umgang mit den ADOs deutlich vereinfachen kann. In Kapitel 7 lernen Sie mit SQL eine universelle Sprache für Datenbankabfragen kennen. SQL und ADO sind keine Alternativen, sondern ergänzen sich wunderbar. Indem etwa beim Aufruf der *Open*-Methode eines *Recordset*-Objekts anstelle eines Tabellennamens ein SQL-Kommando übergeben wird, werden nur jene Datensätze zurückgegeben, die durch das SQL-Kommando ausgewählt werden. SQL ist daher keine neue Variante bei den Datenbankzugriffsmethoden, sondern eine universelle »Datenbanksprache«, die bei allen Datenbankzugriffsmethoden im Mittelpunkt steht.

5.15 Fragen

Frage 1:

Welche allgemeine Aufgabe besitzen die ADOs?

Frage 2:

Wie hängen ADO und Visual Basic zusammen?

a) Die ADOs wurden speziell für Visual Basic 6.0 entwickelt.

b) ADO ist lediglich eine Datenbankschnittstelle, die nach dem Start von Visual Basic ausgewählt werden muß.

c) ADO kann von Visual Basic aufgrund inkompatibler Schnittstellen nicht direkt aufgerufen werden. Dazu wird OLE DB benötigt.

Frage 3:

Warum ist ohne ein *Connection*-Objekt kein Datenbankzugriff möglich?

Frage 4:

Welche beiden Angaben muß die Verbindungszeichenfolge mindestens besitzen?

Frage 5:

Wo wird die Verbindungszeichenfolge angegeben – beim Öffnen des *Connection*- oder beim Öffnen des *Recordset*-Objekts?

Frage 6:

Eine Programmiererin möchte über ein SQL-Kommando den Inhalt einer Tabelle aktualisieren. Benötigt sie dazu ein *Command*- oder ein *Recordset*-Objekt?

Die Antworten zu den Fragen finden Sie in Anhang D.

Kapitel 6

Der Umgang mit Datenumgebungen

Der Umgang mit den ADO-Objekten hat (mindestens) einen kleinen Nachteil. Der Name der Datenquelle, der zu verwendende OLE DB-Provider, die anzusprechenden Tabellen und Abfragen oder SQL-Kommandos müssen ein wenig umständlich im Programmtext festgelegt werden. Soll ein neues Projekt entstehen, das auf der gleichen Datenquelle aufbaut, müssen die Befehle entweder noch einmal neu eingegeben oder aus dem alten Projekt kopiert und gegebenenfalls modifiziert werden. Wäre es nicht praktisch, wenn diese Formalitäten nur einmal erledigt werden müßten und sich bei jedem künftigen Projekt über einen Namen abrufen ließen? Genau diesen Komfort bietet Datenumgebungen. Eine Datenumgebung ist ein spezielles Objekt, das alle für einen Datenbankzugriff erforderlichen Einstellungen, vor allem aber die beteiligten *Connection*-, *Command*- und *Recordset*-Objekte umfaßt. Erstellt werden Datenumgebungsobjekte mit dem Datenumgebungs-Designer (den es übrigens auch beim Ablaufmodell von Visual Basic 6.0 gibt). Auch wenn es sich nach fortgeschrittener Programmierung anhören mag, der Umgang mit dem Datenumgebungs-Designer ist ganz einfach. Je eher Sie sich mit dem Datenumgebungs-Designer anfreunden, um so leichter haben Sie es bei der Programmierung von Datenbankanwendungen auf der Basis von ADO.

Dieses Kapitel ist für das gesamte Buch noch in einer anderen Hinsicht von grundlegender Bedeutung. Sie legen in diesem Kapitel eine Datenumgebung für die kleine Fuhrpark-Datenbank aus Kapitel 3 an, die in allen kommenden Beispielen, die etwas mit dieser Datenbank zu tun haben, vorausgesetzt wird. Doch was ist eine Datenumgebung, warum ist sie wichtig, und warum kann ich nicht einfach direkt auf eine Datenbank zugreifen (daß es

einem Microsoft-Leute auch immer so kompliziert machen müssen)? Fragen über Fragen, auf die es in diesem Kapitel natürlich Antworten gibt.

Während sich die ADO-Objekte, auf denen der Datenumgebungs-Designer basiert, auch mit Visual Basic 5.0 benutzen lassen, steht der Designer selber erst ab Version 6.0 zur Verfügung.

Sie erfahren in diesem Kapitel etwas zu folgenden Themen:

- ✘ Was ist eine Datenumgebung?
- ✘ Warum sind Datenumgebungen so praktisch?
- ✘ Der Umgang mit dem Datenumgebungs-Designer
- ✘ Eine Datenumgebung für die Fuhrpark-Datenbank

6.1 Was ist eine Datenumgebung?

Sie wissen bereits, daß Sie für den Zugriff auf eine Datenbank die ADO-Objekte *Connection* und *Recordset* verwenden müssen und das simple »Öffnen« einer Datenbank eine Reihe von Teilschritten umfaßt: das Anlegen eines *Connection*-Objekts, die Auswahl eines OLE DB-Providers, das Festlegen der Datenquelle, das Öffnen des *Connection*-Objekts und schließlich das Öffnen des *Recordset*-Objekts, wobei beim Öffnen (oder auch vorher) z.B. der Tabellenname angegeben werden muß. Da sich diese Vorgänge stets aufs neue wiederholen, liegt es natürlich nahe, diese unter einem Namen, am besten gleich unter einem Objekt, zusammenzufassen. Diese Möglichkeit besteht bei Visual Basic 6.0 in Gestalt der Datenumgebungen. Eine Datenumgebung (engl. »data environment«) ist ein Objekt, das alle für den Zugriff auf eine Datenbank (bzw. allgemein eine Datenquelle, denn die Zugriffe erfolgen natürlich über OLE DB) erforderlichen ADO-Objekte mit für den Zugriff benötigten Einstellungen unter einem Namen zusammenfaßt. Hier ein konkretes Beispiel. Wenn Sie in verschiedenen Programmen stets auf die Fuhrpark-Datenbank zugreifen und dabei stets auf die gleichen Tabellen oder SQL-Abfragen zugreifen möchten, legen Sie sich einmal eine Datenumgebung an und speichern diese in einer Datei ab. Wenn Sie im nächsten Programm auf die gleiche Datenbank zugreifen möchten, laden Sie einfach die Designerdatei in das Projekt und greifen auf alle Tabellen, Abfragen usw. zu, als wären sie schon immer Teil des Projekts gewesen. Das heißt aber nicht, daß sich der Einsatz von Datenumgebungen erst dann lohnt, wenn mehrere Programme auf ein und dieselbe Datenquelle zugreifen. Vielmehr ist es so, daß der Vorteil von Datenumgebungen dann besonders gut zum Tragen kommt. Und auch das ist wichtig: Eine Datenumge-

bung erleichtert den Datenbankzugriff, da Sie nicht jedesmal alle ADO-Objekte neu anlegen und deren Eigenschaften Werte zuweisen müssen. Voraussetzung für den Datenbankzugriff ist sie jedoch nicht.

Wir könnten also auf die Datenumgebung vollständig verzichten und dennoch alles mit den ADO-Objekten erledigen. Allerdings, und das werden Sie schnell feststellen, der Mehraufwand ist enorm. Mit Hilfe der Datenumgebung für die Fuhrpark-Datenbank, die am Ende dieses Kapitels angelegt wird, fügen Sie in künftige Projekte einfach die Designerdatei *Fuhrpark.dsr* hinzu und verfügen nicht nur über alle für den Zugriff auf die einzelnen Tabellen benötigten *Recordset*-Objekte, sondern können die einzelnen Felder direkt an Steuerelemente auf einem Formular binden. Damit Sie Datenumgebungen wirklich effektiv einsetzen können, müssen Sie aber zwei kleine Hürden nehmen:

- ✘ Sie müssen sich mit den ADO-Objekten auskennen.
- ✘ Sie müssen den Umgang mit dem Datenumgebungs-Designer lernen.

Beides sind natürlich keine unüberwindbaren Hindernisse, können aber dazu beitragen, daß Datenbankneulinge versucht sind, den Designer zunächst links liegen zu lassen.

6.2 Die Idee der Designer

Designer wurden zwar schon mit Visual Basic 5.0 eingeführt, sie dürften den meisten Visual-Basic-Programmierern aber noch relativ unbekannt sein. Dabei arbeiten Sie als Visual-Basic-Programmierer von der ersten Minute an mit einem Designer, nur daß dieser nicht so genannt wird. Wissen Sie, von was die Rede ist? Natürlich von den Formularen und dem »Formulardesigner«. Sobald Sie nämlich ein Formular im Entwurfsmodus öffnen, öffnen Sie damit den Formulardesigner. Seine Aufgabe ist es, daß Anordnen von Steuerelementen auf dem Formular und das Einstellen von Eigenschaften zu ermöglichen und eine (Formular-)Klasse zu erstellen. Starten Sie das Visual-Basic-Programm, wird der Designer geschlossen und das mit Hilfe des Designers erstellte Formular angezeigt. Niemand würde auf die Idee kommen, beim nächsten Laden des Projekts das Formular neu zu erstellen. Statt dessen wird das Formular als Textdatei mit der Erweiterung *.frm* gesichert und beim Erstellen der Exe-Datei in dieselbe eingebaut. Genau nach diesem Prinzip arbeiten die »richtigen« Designer, nur daß diese keine Formulare, sondern etwas anderes »designen«. Beim Datenumgebungs-Designer ist es eine Datenumgebung, bei dem mit Visual Basic 6.0 eingeführten DHTML-Designer z.B. eine Dynamic HTML-Seite. Allen

Designern ist gemeinsam, daß sie zur Entwurfszeit Klassen erzeugen, aus denen zur Ausführungszeit automatisch Objekte werden. Dies gilt auch für den Formulardesigner, denn selbstverständlich ist ein Formular ein Objekt, und hinter jedem Formular steht eine gleichnamige Klasse. Allen Designern ist weiterhin gemeinsam, daß sie eigenständige Module in einem Projekt darstellen und in Dateien mit der Erweiterung *.dsr* gespeichert werden.

Ein Designer ist ein Fenster innerhalb der Entwicklungsumgebung, in dem Klassenmodule mit einer bestimmten Aufgabe und meistens auch einer bestimmten Oberfläche erstellt werden.

In Tabelle 6.1 finden Sie eine Übersicht über die bei Visual Basic zur Verfügung stehenden Designer und ihre Aufgaben. Auch wenn es möglich ist, Designer in Visual C++ zu programmieren, hat von dieser vielversprechenden Möglichkeit, die Entwicklungsumgebung zu erweitern, außer Microsoft offenbar noch niemand Gebrauch gemacht[1].

Tabelle 6.1: Übersicht über die in Visual Basic verfügbaren Designer

Designer	Aufgabe
Datenumgebungs-Designer	Erstellt Datenumgebungsobjekte.
UserConnection-Designer	Wurde mit Visual Basic 5.0 Enterprise eingeführt und erstellt Connection-Objekte für einen ODBC-Datenbankzugriff. Kann als Vorfahre des Datenumgebungs-Designers angesehen werden.
Microsoft Forms 2.0	Erlaubt das Erstellen von MsForms-Formularen, die in Office ab Version 97 eingesetzt werden. Wurde bereits mit Visual Basic 5.0 eingeführt.
DHTML-Designer	Erstellt DHTML-Seiten, die vom Internet Explorer angezeigt werden.
WebClass-Designer	Erstellt im Rahmen von IIS-Applikationen-Projekten WebClass-Komponenten. Eine WebClass-Komponente ist eine ActiveX-DLL, die im Rahmen einer Active-Server-Page-HTML-Seite von einem Webserver ausgeführt wird.

1 Zumindest sind mir keine verfügbaren Designer bekannt.

6.3 Warum sind Datenumgebungen so praktisch?

Datenumgebungen sind deswegen so praktisch, weil sie dem ADO-Programmierer eine Menge Arbeit ersparen. Anstatt bei jedem neuen Projekt, das auf ein und dieselbe Datenquelle zugreift, jedes *Connection*- und *Recordset*-Objekt einzeln anzulegen, wird die Aufgabe einmal mit Hilfe des Datenumgebungs-Designers erledigt und als Designerdatei gespeichert. Bei jedem weiteren Projekt muß lediglich die Designerdatei hinzugefügt werden, und fertig ist die Datenumgebung. Das ist jedoch noch nicht alles:

- ✘ Mit Hilfe eines SQL-Abfragegenerators lassen sich SQL-Abfragen durch Auswahl der Tabellen, Felder und Verknüpfungen mit der Maus erstellen. So wie es bei Microsoft Access schon immer ging.
- ✘ Im Zusammenhang mit dem Microsoft und Oracle SQL Server läßt sich in der Datenumgebung auch die Datenbank selbst bearbeiten, und es lassen sich z.B. neue Tabellen anlegen (beim Zugriff auf Microsoft Access geht das allerdings nicht).
- ✘ Einzelne Felder eines *Command*-Objekts, aber auch das komplette *Command*-Objekt, können mit der Maus auf ein Formular gezogen werden, was zur Folge hat, daß für jedes Feld ein passendes Steuerelement mit einer Bindung an das jeweilige Feld gesetzt wird.
- ✘ Der Datenumgebungs-Designer erlaubt nicht nur das Anlegen einfacher *Command*-Objekte. Indem zu einem *Command*-Objekt ein weiteres *Command*-Objekt als Unterobjekt eingefügt wird, lassen sich hierarchische Datensatzgruppen erstellen (das Verfahren basiert auf dem »neuen« SQL-Kommando *SHAPE*). Ein einfaches Beispiel ist das klassische »Kundenauftragsdaten«-Problem. In einer Auftragstabelle wird üblicherweise nur die Kundennummer gespeichert. Möchte man zu jedem Auftrag auch den Namen des Kunden oder seine Adresse sehen, muß das *Command*-Objekt, welches für die Auftragstabelle steht, ein *Command*-Objekt, das für die Kundenstammdaten steht, als Unterobjekt enthalten. Die Beziehung wird über ein gemeinsames Feld, in diesem Fall die Kundennummer, hergestellt. Wird jetzt das *Command*-Objekt *Auftragstabelle* auf ein Formular gezogen, stellt Visual Basic die hierarchische Beziehung automatisch her, indem es passende Felder für die Kundenstammdaten anordnet, so daß beim Scrollen durch die Auftragsdaten

auch die Kundenstammdaten angezeigt werden. Auf das Thema hierarchische Datensatzgruppen wird in Kapitel 6.6.3 kurz eingegangen[1].

6.4 Der Umgang mit dem Datenumgebungs-Designer

Datenumgebungen werden mit dem Datenumgebungs-Designer erstellt und bearbeitet. Der Datenumgebungs-Designer, im folgenden mit *DUD* abgekürzt, wird über den Eintrag DATA ENVIRONMENT HINZUFÜGEN im PROJEKT-Menü gestartet. Sollte der Eintrag dort nicht angeboten werden, müssen Sie zunächst den Eintrag KOMPONENTEN wählen, die Registerkarte *Designer* aktivieren und anschließend den DUD (Eintrag »Data Environment«) auswählen.

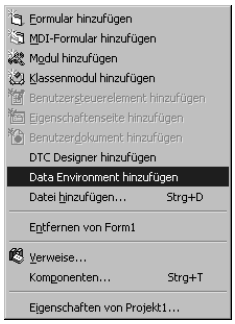

Bild 6.1: Eine Datenumgebung wird über das PROJEKT-Menü zu einem Projekt hinzugefügt

Der »Vorfahre« des Datenumgebungs-Designers ist der UserConnection-Designer aus Visual Basic 5.0 Enterprise Edition. Er ist ausschließlich für das Erstellen von ODBC-Verbindungen auf der Basis der RDOs zuständig und wurde von den Programmierern offenbar kaum beachtet. Mit den ADO-Objekten hat er nichts zu tun.

Der Datenumgebungs-Designer besitzt ein eigenes Objektmodell (das »Data Environment Object Model«). Nach Einbinden einer Referenz auf die Objektbibliothek »Microsoft Data Environment Extensibility Objects 1.0« steht ein Objektmodell mit dem *DEExtDesigner*-Objekt an oberster Stelle zur Verfügung, mit dem eine komplette Datenumgebung auch programmgesteuert aufgebaut werden kann.

1 Ansonsten gehört dieses Thema nämlich bereits in die Kategorie fortgeschrittene Datenbankprogrammierung.

6.5 Eine Datenumgebung für die Fuhrpark-Datenbank

In diesem Abschnitt wird der Datenumgebungs-Designer für ein wichtiges Projekt eingesetzt. Im folgenden wird Schritt für Schritt eine Datenumgebung für die Fuhrpark-Datenbank angelegt. Damit lernen Sie nicht nur den Umgang mit dem DUD, sondern erhalten gleichzeitig eine Datenumgebung, die für alle noch folgenden Beispielprogramme benutzt wird.

Aus Gründen der besseren Übersichtlichkeit wird in diesem Abschnitt aber noch nicht die komplette Umgebung für den Zugriff auf alle Tabellen vorgestellt (zumal die Datenbank im Laufe der nächsten Kapitel noch ein wenig wachsen soll), sondern lediglich der Zugriff auf die Tabelle *Modelldaten*. Nach einer kurzen Einarbeitungsphase werden Sie mit dem DUD virtuos umgehen, und es sollte überhaupt kein Problem sein, weitere Tabellen und Abfragen hinzuzufügen.

Schritt 1: Hinzufügen einer Datenumgebung zum Projekt

Starten Sie Visual Basic, und legen Sie ein Standard-Exe-Projekt an. Fügen Sie über das PROJEKT-Menü eine Datenumgebung hinzu.

Auch wenn es grundsätzlich keine Rolle spielt, in welchem Projekt eine Datenumgebung angelegt wird, können Sie sich ein wenig Arbeit sparen, indem Sie anstelle eines Standard-Exe-Projekts ein Datenprojekt anlegen. Dies ist aber lediglich eine Projektvorlage aus dem Unterverzeichnis *\Templates*, die automatisch mit einer Datenumgebung und einem Datenreport (den Sie ruhig entfernen können, falls Sie keinen Report benötigen) angelegt wird.

Schritt 2: Die Datenumgebung erhält einen Namen

Sie verfügen damit über eine Datenumgebung, in der bereits ein ADO-*Connection*-Objekt angelegt wurde. Geben Sie der Datenumgebung den Namen »envFuhrpark«, indem Sie das *DataEnvironment*-Objekt im Designer selektieren und die F4-Taste drücken, um das Eigenschaftsfenster zu öffnen.

Schritt 3: Das Connection-Objekt erhält einen Namen

Auch das *Connection*-Objekt muß umbenannt werden. Selektieren Sie es im Designer, drücken Sie die F4-Taste, und geben Sie der *Name*-Eigenschaft im Eigenschaftsfenster den Wert »conFuhrpark«.

Schritt 4: Die Verbindung zu »Fuhrpark.mdb« wird hergestellt

Auch wenn es nicht zwingend notwendig ist, wird die Verbindung zur Datenquelle bereits zur Entwurfszeit hergestellt. Unser *Connection*-Objekt soll mit der Access-Datenbank *Fuhrpark.mdb* verbunden werden, die in Kapitel 3 erstellt wurde. Klicken Sie dazu das *Connection*-Objekt mit der rechten Maustaste an, und wählen Sie den Eintrag EIGENSCHAFTEN. Sie könnten alle Einstellungen auch im Eigenschaftenfenster vornehmen, doch geht es im Eigenschaftendialogfeld (man achte auf den kleinen Unterschied) sehr viel einfacher. Das Eigenschaftendialogfeld mit Namen *Datenlinkeigenschaften* begrüßt Sie mit der Registerkarte *Provider*. Hier wird der OLE DB-Provider ausgewählt. Da es sich bei *Fuhrpark.mdb* um eine Access-97-Datenbank handelt, wählen Sie den Eintrag »Microsoft Jet 3.51 OLE DB Provider«. Bevor Sie auf *OK* klicken, müssen Sie die Datendatenbank auswählen. Das geschieht wie immer auf der Registerkarte *Verbindung*, auf die Sie entweder über die *Weiter>>*-Schaltfläche (unten rechts) oder über direkte Auswahl gelangen. Stellen Sie hier den Dateinamen ein, und klicken Sie auf *OK*, um das Dialogfeld wieder zu schließen.

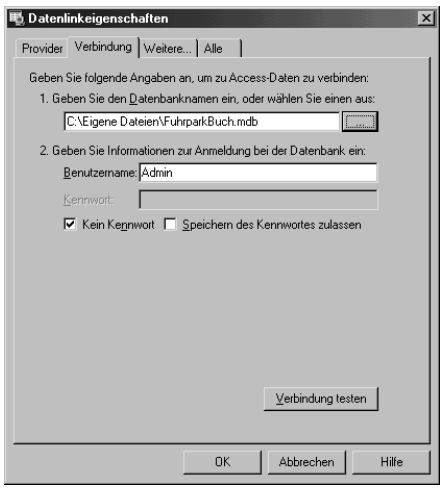

Bild 6.2: In diesem Dialogfeld wird die Datenquelle ausgewählt

Schritt 5: Hinzufügen eines Command-Objekts für die Tabelle »Modelldaten«

Das *Connection*-Objekt ist nun mit der Datenquelle verbunden und damit fertig. Fügen Sie nun ein *Command*-Objekt hinzu, um auf die Tabelle *Modelldaten* zugreifen zu können. Selektieren Sie dazu das *Connection*-Objekt (eine Datenumgebung kann mehrere *Connection*-Objekte umfassen), und klicken Sie auf das »Befehl hinzufügen«-Symbol in der Symbol-

leiste. Sie werden feststellen, daß das *Connection*-Objekt einen Untereintrag mit dem Namen »Command1« enthält. Ein *Command*-Objekt ist aber keine 1:1-Entsprechung einer Tabelle (das wäre ein *Recordset*-Objekt). Es ist vielmehr ein Kommando, mit dem sich z.B. der Inhalt einer Tabelle einem *Recordset*-Objekt zuweisen läßt. Sie müssen dem *Command*-Objekt noch mitteilen, was es tun soll. Das geschieht (wie üblich), indem Sie das *Command*-Objekt mit der rechten Maustaste anklicken und den Eintrag EIGENSCHAFTEN wählen.

Bild 6.3: Im Eigenschaftendialogfeld eines Command-*Objekts wird eingestellt, was das Objekt tun soll*

Geben Sie dem *Command*-Objekt als erstes den Namen »cmdModelldaten«. Wählen Sie in der Auswahlliste *Datenbank-Objekt* die Kategorie »Tabelle«. Das bedeutet, daß das *Command*-Objekt für eine Tabelle stehen soll. Ging alles gut, erscheinen in der darunterliegenden Listenauswahl *Objektname* die Tabellennamen der Datenbank *Fuhrpark.mdb*. Das ist keine Zauberei, sondern liegt daran, daß sich das *Command*-Objekt auf das *Connection*-Objekt *conTest* bezieht (eine Zuordnung, die sich jederzeit ändern läßt). Wählen Sie die Tabelle *Modelldaten* aus. Auch wenn ein *Command*-Objekt eine Vielzahl an Einstellungen ermöglicht, sind für dieses Beispiel damit alle Einstellungen getroffen. Schließen Sie das Dialogfeld über *OK*.

Schritt 6: Testen des Command-Objekts

Nach Schließen des Dialogfeldes werden Sie feststellen, daß vor dem *Command*-Objekt im Designer ein »+«-Zeichen erscheint. Das bedeutet, daß das *Command*-Objekt für eine Datensatzgruppe steht, deren Felder Sie sich bereits ansehen können. Die Inhalte stehen zu diesem Zeitpunkt aber noch nicht zur Verfügung.

Bild 6.4: Das »+«-Zeichen vor dem Command-*Objekt deutet an, daß es aktiv ist und für eine bestimmte Anzahl an Feldern steht*

Dieser Umstand wird am Anfang schnell übersehen: Genau wie beim direkten Öffnen eines *Recordset*-Objekts wird auch über das *Command*-Objekt ein statisches, schreibgeschütztes *Recordset*-Objekt angelegt. Mit anderen Worten, Änderungen an der Datensatzgruppe sind nicht möglich. Ist das nicht gewünscht, muß dies in der Registerkarte *Weitere* geändert werden (stellen Sie bei der Auswahl *Cursorposition* »2 – Server Side-Cursor verwenden« ein).

Schritt 7: Hinzufügen eines Command-Objekts für die Tabelle »Fahrzeugdaten«

Auch für die Tabelle *Fahrzeugdaten* soll es ein *Command*-Objekt geben. Selektieren Sie dazu im Designer erneut das *Connection*-Objekt, und klikken Sie auf das »Befehl hinzufügen«-Symbol in der Symbolleiste. Es wird ein weiteres *Command*-Objekt hinzugefügt. Geben Sie diesem den Namen »cmdFahrzeugdaten«, und verbinden Sie es diesmal mit der Tabelle *Fahrzeugdaten*.

Schritt 8: Die Datenumgebung wird gespeichert

Wäre die Datenquelle umfangreicher, könnten Sie nach Herzenslust weitere *Command*-Objekte hinzufügen. Ein *Command*-Objekt muß nicht einer Tabelle entsprechen, sondern kann für ein beliebiges SQL-Kommando stehen (mehr dazu später). Da die Fuhrpark-Datenbank zur Zeit aber nur zwei Tabellen umfaßt, ist die Datenumgebung damit erst einmal fertig. Damit Sie sie auch in anderen Projekten verwenden können, speichern Sie sie jetzt über den Menübefehl PROJEKT/SPEICHERN VON ENVTEST UNTER unter dem Namen »Fuhrpark.dsr« ab.

Schritt 9: Die Datenumgebung tritt in Aktion

Auch wenn die Datenumgebung zu diesem Zeitpunkt noch nicht benötigt wird, sollen Sie sie natürlich auch einmal in Aktion erleben. Ordnen Sie dazu ein Textfeld auf dem Formular an, und geben Sie diesem den Namen »txtModellName«. Ordnen Sie als nächstes eine Schaltfläche an, die den Namen »cmdVorwärts« und die Überschrift »Vorwärts« erhalten soll. Fügen Sie nun in *Form_Load* folgende Befehle ein:

```
Private Sub Form_Load()
    With txtModellname
     Set .DataSource = envFuhrpark
     .DataMember = "cmdModelldaten"
     .DataField = "ModellName"
    End With
End Sub
```

Diese Befehle stellen eine Verbindung zwischen dem Textfeld und dem Datenfeld *ModellName* her. Zum Schluß soll auch die Schaltfläche etwas zu tun bekommen. Fügen Sie in deren *Click*-Prozedur den folgenden Befehl ein:

```
Private Sub cmdVorwärts_Click()
    envFuhrpark.rscmdModelldaten.MoveNext
End Sub
```

Dieser Befehl führt die *MoveNext*-Methode beim *Recordset*-Objekt *rscmdModelldaten* aus. Dieses Objekt haben Sie nicht anlegen müssen, da es uns die Datenumgebung freundlicherweise zur Verfügung stellt.

Wenn Sie das kleine Programm jetzt starten, werden Sie feststellen, daß der Inhalt des Feldes *Modellname* im Textfeld angezeigt wird und Sie über die Schaltfläche zum nächsten Datensatz wechseln können.

Wenn Sie im Codefenster den Namen der Datenumgebung, in diesem Fall »envFuhrpark«, eintippen und einen Punkt folgen lassen, werden alle Unterobjekte der Datenumgebung angezeigt. Dazu gehören neben dem angelegten *Connection*-Objekt und den beiden angelegten *Command*-Objekten, die als Methoden angeboten werden, auch zwei Eigenschaften namens *rscmdModelldaten* und *rscmdFahrzeugdaten*. Über diese beiden Eigenschaften ist der direkte Zugriff auf jene *Recordset*-Objekte möglich, die von den *Command*-Objekten automatisch angelegt werden (ein besonderer Service der Datenumgebung). Die *Command*-Objekte selber werden als Methoden angeboten, da sie aufgerufen werden müssen, damit etwas passiert. Und was genau passiert beim Aufruf eines *Command*-Objekts? Das hängt

natürlich vom Inhalt des Kommandos (*CommandText*-Eigenschaft) ab. In diesem konkreten Fall wird eine Tabelle geöffnet. Anders als man es vermuten könnte, besitzen diese Methoden keinen Rückgabewert, so daß der folgende Befehl nicht erlaubt ist:

```
Dim Rs As ADODB.Recordset
' Weitere Befehle
Set Rs = envFuhrpark.cmdModelldaten
```

Möchte man das *Recordset*-Objekt einer Variablen zuweisen, muß dafür die entsprechende Eigenschaft verwendet werden, denn dazu ist diese da:

```
Set Rs = envFuhrpark.rscmdModelldaten
```

Möchte man direkt auf ein *Command*-Objekt und seine Eigenschaften zugreifen, muß dies über die *Commands*-Eigenschaft der Datenumgebung erfolgen:

```
?envFuhrpark.Commands("cmdFahrzeugdaten").CommandText
```

Dieser Befehl gibt den Text des Kommandos *cmdFahrzeugdaten* im Direktfenster aus. Der große Vorteil einer Datenumgebung besteht nicht darin, daß diese Dinge überhaupt möglich werden (das geht auch ohne Datenumgebung). Er besteht vielmehr darin, daß diese Dinge sehr viel einfacher möglich werden. Wenn Sie beim nächsten Projekt auf die Fuhrpark-Datenbank zugreifen möchten, laden Sie einfach die Datenumgebung in das Projekt. Sie erhalten damit automatisch alle angelegten *Connection*- und *Recordset*-Objekte.

6.6 Fortgeschrittene Eigenschaften der Datenumgebung

Eine Datenumgebung ist mit dem bloßen Herstellen einer Verbindung oder dem Bereitstellen einfacher *Command*- und *Recordset*-Objekte ein wenig unterfordert, wenngleich am Anfang meistens keine weiteren Aufgaben anfallen. Doch, Sie werden es sich schon gedacht haben, eine Datenumgebung kann noch mehr. Dazu gehört unter anderem:

- ✘ Der direkte Zugriff auf die *Recordset*-Objekte
- ✘ Das Erstellen von SQL-Kommandos
- ✘ Der Aufbau von Befehlshierarchien

- ✘ Die Nutzung von SQL-Aggregatfunktionen
- ✘ Das Gruppieren von Datensatzgruppen

6.6.1 Der direkte Zugriff auf die Recordset-Objekte

Wie Sie direkt auf die *Recordset*-Objekte einer Datenumgebung, die automatisch von den *Command*-Objekten erzeugt werden, zugreifen, haben Sie bereits erfahren. Sie geben den Namen der Datenumgebung ein und können nach Eingabe des Punkts ein *Recordset*-Objekt als Eigenschaft des Datenumgebungsobjekts auswählen.

Der folgende Befehl weist das *Recordset*-Objekt *rscmdModelldaten* der Variablen *Rs* zu:


```
Set Rs = envFuhrpark.rscmdModelldaten
```

6.6.2 Das Erstellen von SQL-Kommandos

Der Datenumgebungs-Designer ist Ihnen auf Wunsch auch beim Erstellen von SQL-Kommandos für eine Abfrage behilflich. Kleine Kostprobe gefällig? Dann gehen Sie wie folgt vor:

1. Legen Sie ein neues *Command*-Objekt an.
2. Wählen Sie für das *Command*-Objekt kein bereits vorhandenes Objekt aus, sondern wählen Sie die Option *SQL-Anweisung*.
3. Sie können nun in das Eingabefeld ein SQL-Kommando eingeben (eine direkte Syntaxüberprüfung gibt es aber nicht). Sie können aber auch auf die Schaltfläche *SQL-Generator* klicken. Es öffnet sich ein weiteres Fenster, in dem Sie das SQL-Kommando wie in Microsoft Access durch Auswahl einzelner Tabellen, Felder und Vergleichsoperatoren zusammenbauen.
4. Nebenbei lernen Sie auch etwas, denn wenn Sie den SQL-Generator schließen, wird das generierte SQL-Kommando im Eingabefeld angezeigt, wo Sie es bei Bedarf modifizieren können (um beispielsweise einen Alias einzufügen).

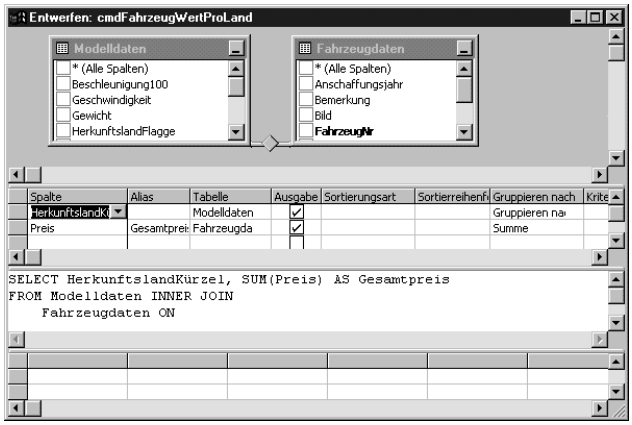

Bild 6.5: Der eingebaute SQL-Generator ist beim Zusammenbauen eines SQL-Kommandos behilflich

6.6.3 Der Aufbau von Befehlshierarchien

Einer der dicken Pluspunkte der Datenumgebung ist die Möglichkeit, Hierarchien zwischen *Command*-Objekten aufbauen zu können Das klassische Beispiel ist jener Fall, wo ein Anwender aus einer Kundenstammdatentabelle einen Datensatz auswählen und damit gleichzeitig auch alle Aufträge aus der Auftragstabelle sehen möchte, die dieser Kunde getätigt hat. Hier liegt eine typische Eltern/Kind-Hierarchie zwischen der Kundentabelle und der Auftragstabelle vor. Diese Hierarchie wird in einer Datenumgebung dadurch verwirklicht, daß ein *Command*-Objekt ein weiteres *Command*-Objekt als Unterobjekt enthält. Indem in dem oberen *Command*-Objekt die Kundentabelle, in dem unteren *Command*-Objekt die Auftragstabelle und beide *Command*-Objekte über ein gemeinsames Feld verbunden werden, in diesem Fall die Kundennummer, läßt sich eine Hierarchie auf einfache Weise nachbauen. Gegenüber einer SQL-JOIN-Operation, mit der sich ein ähnliches Resultat erzielen läßt, besteht der Vorteil, daß, wann immer über das obere *Command*-Objekt ein Datensatz ausgewählt wird, das untere *Command*-Objekt automatisch die passenden Datensätze zur Verfügung stellt. Technisch realisiert wird das Ganze durch den Umstand, daß das letzte Feld des oberen *Command*-Objekts die vom unteren *Command*-Objekt zur Verfügung gestellte Datensatzgruppe enthält und damit einen direkten Zugriff auf diese Datensatzgruppe erlaubt. Und es kommt noch besser. Wenn Sie das obere *Command*-Objekt auf ein Formular ziehen, wird automatisch ein HFlexGrid-Steuerelement (siehe Kapitel 4.5) angeordnet, in dem sich diese Hierarchie anzeigen läßt.

Die Hierarchie von Datensatzgruppen basiert auf dem neuen SQL-Kommando *SHAPE*, bei dem es sich aber nicht um ein Standard-SQL-Kommando handelt. Sie können sich das *SHAPE*-Kommando jederzeit anschauen, indem Sie ein hierarchisches *Command*-Objekt im Designer mit der rechten Maustaste anklicken und den Eintrag STRUKTURINFO wählen.

Auch die Fuhrpark-Datenbank erlaubt in ihrer jetzigen Ausbaustufe bereits eine solche einfache Hierarchie. Stellen Sie sich vor, Sie wählen aus der Tabelle *Modelldaten* ein Modell aus und möchten nun alle Fahrzeuge dieses Modells sehen, die in der Tabelle *Fahrzeugdaten* enthalten sind. Wofür früher ein umfangreicheres SQL-Kommando erforderlich war, genügt es dank der Datenumgebung, das zuständige *Command*-Objekt auf ein Formular zu ziehen. Ein *Command*-Objekt, das genau das kann, soll im folgenden angelegt werden. Für die folgenden Schritte wird davon ausgegangen, daß die Datenumgebung *Fuhrpark.dsr* bereits existiert.

Schritt 1: Anlegen eines neuen Command-Objekts

Fügen Sie im Designer ein weiteres *Command*-Objekt hinzu, geben Sie diesem den Namen »cmdAlleFahrzeugeEinesModells«, und verbinden Sie es mit der Tabelle *Modelldaten*.

Schritt 2: Anlegen eines Unterobjekts

Selektieren Sie das neu angelegte *Command*-Objekt, und klicken Sie in der Symbolleiste auf die Schaltfläche *Untergeordneten Befehl hinzufügen*. Sie werden feststellen, daß in der Feldliste des *Command*-Objekts ein Eintrag mit dem Namen »Command1« angelegt wurde.

Schritt 3: Das Unter-Command-Objekt erhält Eigenschaften

Klicken Sie das neue *Command*-Objekt mit der rechten Maustaste an, wählen Sie EIGENSCHAFTEN, und geben Sie ihm den Namen »cmdAlleFahrzeuge«. Wählen Sie in der Auswahlliste *Datenbank-Objekt* den Eintrag »Tabelle« und in der Auswahlliste *Objektname* den Tabellenamen »Fahrzeugdaten« aus. Klicken Sie noch nicht auf *OK*, sondern schalten Sie auf die Registerkarte *Beziehung* um, denn hier wird die Beziehung zu dem übergeordneten *Command*-Objekt bearbeitet.

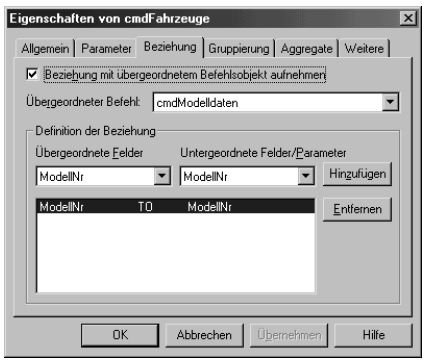

Bild 6.6: Auf der Registerkarte Beziehung *wird die Beziehung zwischen zwei* Command-*Objekten festgelegt*

Sie werden feststellen, daß der übergeordnete Befehl bereits richtig angezeigt wird. Auch das Feld *ModellNr*, über das die Beziehung hergestellt wird, wurde richtig erkannt. Sie müssen es allerdings noch hinzufügen, was über die Schaltfläche *Hinzufügen* geschieht.

Schließen Sie das Dialogfeld über *OK*. Im Designerfenster sollte der Feldeintrag »cmdAlleFahrzeuge« mit einem »+«-Zeichen erscheinen, was bedeutet, daß dieses Feld für eine ganze Datensatzgruppe steht.

Fertig, damit wurde ein Eltern-Kind-Beziehung zwischen zwei Befehlen (*Command*-Objekten) etabliert. Damit Sie diese Beziehung auch in Aktion erleben können, gibt es zwei Möglichkeiten:

- ✘ Sie ziehen das obere *Command*-Objekt auf ein Formular.
- ✘ Sie ordnen auf einem Formular das (hierarchische) HFlexGrid-Steuerelement an, verbinden die *DataSource*-Eigenschaft mit der Datenumgebung und wählen über die *DataMember*-Eigenschaft das neu angelegte (Ober-)*Command*-Objekt *cmdAlleFahrzeugeEinesTyps* aus.

6.6.4 Die Nutzung von SQL-Aggregatfunktionen

Mit SQL läßt sich eine Menge machen, davon wird in Kapitel 7 ausführlich die Rede sein. Stellen Sie sich vor, Sie möchten den Gesamtanschaffungspreis aller Fahrzeuge in der Tabelle *Fahrzeugdaten* wissen. Nun, mit dem, was Sie in Kapitel 5 über das *Recordset*-Objekt und in diesem Kapitel über Datenumgebungen gelernt haben, könnte ein Lösung wie folgt aussehen:

```
Set Rs = envFuhrpark.rscmdFahrzeugdaten
Do While Not Rs.EOF
    Summe = Summe + Rs.Fields("Preis").Value
    Rs.MoveNext
Loop
```

Indem die *MoveNext*-Methode der Reihe nach alle Datensätze »besucht«, muß bei jedem Durchlauf lediglich der Inhalt des Preisfeldes aufaddiert werden. Diese Lösung ist nicht schlecht, sie ist aber auch nicht besonders elegant. Stellen Sie sich vor, wie lange es dauern würde, wenn die Tabelle nicht 30, sondern vielleicht 3000 Datensätze umfaßt. Das »Besuchen« aller Datensätze kommt für Datenbankprofis nur in absoluten Ausnahmefällen in Frage, da es viel zu langsam ist. Zum Glück gibt es eine sehr einfache Lösung, die SQL-Funktion *SUM*, die in diesem Zusammenhang auch als Aggregatfunktion bezeichnet wird, da sie den Inhalt mehrerer Felder zusammenfaßt. Das folgende SQL-Kommando bildet die Summe und gibt das Ergebnis in Gestalt eines einzigen Datensatzes mit einem einzigen Feld, das *Gesamtpreis* heißt, zurück:

```
SELECT SUM (Preis) AS Gesamtpreis FROM Fahrzeugdaten
```

Sie können dieses SQL-Kommando direkt dem Kommandotext eines *Command*-Objekts zuweisen. Sie können die Aggregatfunktion aber auch in der Registerkarte *Aggregate* im Eigenschaftsdialog eines *Command*-Objekts auswählen. Dieses Verfahren kommt normalerweise nicht bei so einfachen Anwendungen in Frage, sondern wird in der Praxis dazu benutzt, die Hierarchie zwischen *Command*-Objekten zu verfeinern. Das Ergebnis ist nämlich nicht ein einfaches SQL-Kommando, sondern ebenfalls ein *SHAPE*-Kommando. Auch wenn einem *SHAPE*-Kommando eine einfache Überlegung zugrunde liegt, ist es nichts für SQL-Einsteiger.

Führen Sie die folgende Schrittfolge aus, um ein *Command*-Objekt zu erhalten, das für den Gesamtwert der Fahrzeuge in der Tabelle *Fahrzeugdaten* steht.

Schritt 1: Hinzufügen eines Command-Objekts

Fügen Sie zur bestehenden Datenumgebung ein neues *Command*-Objekt hinzu (achten Sie darauf, daß es auf der Ebene der ersten Verbindung angeordnet wird), und geben Sie ihm den Namen »cmdGesamtwert«. Alternativ können Sie auch eine neue Datenumgebung anlegen.

Schritt 2: Festlegen der CommandText-Eigenschaft

Wählen Sie auf der Registerkarte *Allgemein* des *Command*-Objekts die Option *SQL-Anweisung,* und geben Sie das obige SQL-Kommando in das Textfeld ein.

Schritt 3: Hinzufügen der SUM-Funktion

Schalten Sie auf die Registerkarte *Aggregate* um, und klicken Sie auf *Hinzufügen* (wird die Schaltfläche nicht angeboten, stimmt etwas mit dem

SQL-Kommando nicht). In diese Liste der Aggregatfunktionen wird ein neuer Eintrag mit dem Namen »Aggregat1« eingefügt.

Bild 6.7: Auf der Registerkarte Aggregate *werden alle* Aggregate*-Befehlsobjekte aufgelistet*

Schritt 4: Ändern der Voreinstellungen

Sie sollten eine Reihe von Namen ändern. Das Aggregat erhält den Namen »Gesamtsumme«, und der Gesamtsummenbefehl soll den Namen »cmdGesamtsumme« erhalten. Der Feldname *Gesamtpreis* wird dagegen durch das SQL-Kommando vorgegeben.

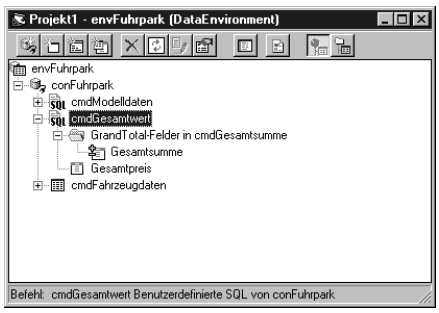

Bild 6.8: Die Datenumgebung wurde um ein Command*-Objekt erweitert*

Schritt 5: Das Aggregate-Befehlsobjekt im Einsatz

Das neu angelegte *Command*-Objekt, das auf einem *SHAPE*-Kommando basiert, kann auf vielfältige Weise benutzt werden. Ziehen Sie das *cmdGesamtwert*-Objekt mit der rechten Maustaste auf ein Formular. Es erscheint ein Menü, aus dem Sie auswählen können, auf welche Weise die Felder des *Command*-Objekts dargestellt werden sollen. Wählen Sie den Eintrag GEBUNDENE STEUERELEMENTE, wird die Summe nach dem Programmstart in einem Textfeld angezeigt.

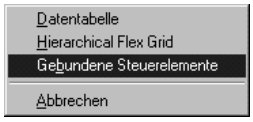

Bild 6.9: Aus diesem Kontextmenü wird ausgewählt, auf welche Weise ein Command-*Objekt auf dem Formular dargestellt wird*

6.6.5 Gruppieren des Fahrzeugbestands nach Fahrzeugtyp

Zum Abschluß dieses Kapitels soll gezeigt werden, wie sich mit dem Datenumgebungs-Designer einfache Gruppierungen vornehmen lassen. Eine Gruppierung basiert auf SQL und bedeutet, daß mehrere Datensätze unter Verwendung eines gemeinsamen Feldes zu einer Gruppe zusammengefaßt werden. Möchten Sie z.B. die Anschaffungssummen aller Fahrzeuge aus dem gleichen Herkunftsland zusammengefaßt sehen, ist dies ein Fall für eine Gruppierung. Das SQL-Kommando, das diese Gruppierung durchführt, ist diesmal relativ kompliziert:

```
SELECT HerkunftslandKürzel, SUM(Preis) AS Gesamtpreis FROM
Modelldaten INNER JOIN Fahrzeugdaten ON Modelldaten.ModellNr
= Fahrzeugdaten.ModellNr GROUP BY Herkunftslandkürzel
```

Gruppierungen werden bei einem *Command*-Objekt in der Registerkarte *Gruppierung* vorgenommen. Auch hier gilt, daß es nicht die Aufgabe dieser Registerkarte ist, einfache *GROUP BY*-SQL-Kommandos zu erzeugen, sondern vielmehr eine Verfeinerung im Rahmen einer *Command*-Hierarchie vorzunehmen. Das Ergebnis ist daher auch kein *SELECT*-, sondern ein *SHAPE*-Kommando.

Führen Sie die folgende Schrittfolge aus, um ein *Command*-Objekt zu erhalten, das die Gesamtwerte aller Fahrzeuge in der Tabelle *Fahrzeugdaten* nach Herkunftsländern gruppiert.

Schritt 1: Hinzufügen eines Command-Objekts

Fügen Sie zur bestehenden Datenumgebung ein neues *Command*-Objekt hinzu, und geben Sie ihm den Namen »cmdFahrzeugWertProLand«.

Schritt 2: Festlegen der CommandText-Eigenschaft

Wählen Sie auf der Registerkarte *Allgemein* des *Command*-Objekts die Option *SQL-Anweisung,* und geben Sie das obige SQL-Kommando in das Textfeld ein.

Schritt 3: Hinzufügen der Gruppierung

Schalten Sie zur Registerkarte *Gruppierung* um, wählen Sie die Option *Command-Objekt gruppieren*, und fügen Sie die beiden in der Auswahlliste *Felder in Befehl* aufgeführten Feldnamen in die Liste *Felder zum Gruppieren* ein.

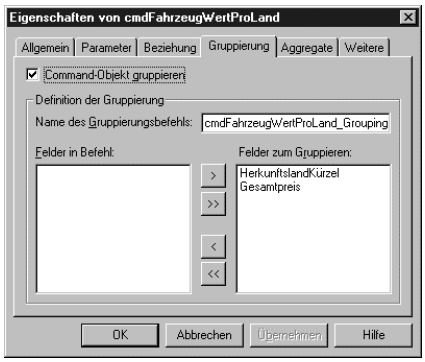

Bild 6.10: Auf dieser Registerkarte des Command-Objekts werden die Felder für die Gruppierung ausgewählt

Schritt 4: Arbeiten mit dem neuen Command-Objekt

Das Ergebnis ist ein neues *Command*-Objekt (mit einem recht langen Namen), das Sie wie alle *Command*-Objekte auf ein Formular ziehen können (wählen Sie diesmal den Darstellungsmodus »Datentabelle«).

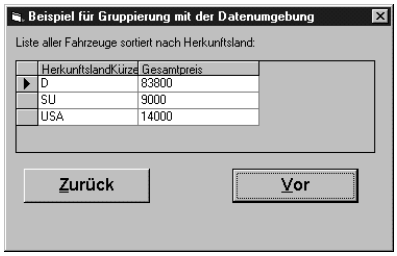

Bild 6.11: Dank des neuen Command-Objekts werden die Fahrzeugwerte nach Ländern gruppiert

Möchten Sie nur den Gesamtwert eines einzelnen Landes erhalten, muß an das SQL-Kommando eine *HAVING*-Klausel angehängt werden (z.B. *HAVING HerkunftslandKürzel = 'D'*).

6.7 Zusammenfassung

Datenumgebungen sind eine der wichtigsten mit Visual Basic 6.0 eingeführten Neuerungen. Sie sind zwar nur eine Option, doch erleichtern sie die Datenbankprogrammierung so deutlich, daß wohl niemand (nach einer kurzen Phase der Skepsis, die völlig normal ist) auf sie verzichten möchte. Eine Datenumgebung ist eine weitere Ebene oberhalb der ADO-Objekte. Sie besteht aus einem *DataEnvironment*-Objekt, von dem sich die damit verbundenen *Connection*-, *Recordset*- und *Command*-Objekte ableiten. Alles, was eine Datenumgebung zur Laufzeit bietet, läßt sich auch mit den ADO-Objekten alleine erreichen. Der große Pluspunkt einer Datenumgebung ist der Komfort sowohl zur Entwurfs- als auch zur Ausführungszeit. So bietet die Datenumgebung für jedes angelegte *Command*-Objekt automatisch ein *Recordset*-Objekt an, über das die vom *Command*-Objekt geholten Datensätze in gewohnter Manier zur Verfügung stehen. Besonders leistungsfähig wird die Datenumgebung, wenn man zusätzliche Möglichkeiten, wie hierarchische Recordset-Objekte, Gruppierungen oder Aggregatfunktionen, nutzt. Viele der fortgeschrittenen Möglichkeiten werden Sie erst dann entdecken, wenn Sie die Datenbanksprache SQL besser kennengelernt haben.

6.8 Wie geht es weiter?

Mit den ADO-Objekten, dem ADO-Datensteuerelement und dem Umgebungsdesigner haben Sie die wichtigsten Werkzeuge kennengelernt, die Visual Basic für den Datenbankzugriff zur Verfügung stellt. Sie können damit (über OLE DB) die Verbindung zu einer Access-Datenbank herstellen, auf den Inhalt der Datenbank zugreifen, um z.B. den Inhalt einer Tabelle oder in der Datenbank gespeicherte Abfragen in ein *Recordset*-Objekt zu übernehmen, und die Verbindung am Ende wieder schließen. Es wurde bereits mehrfach angesprochen: Um das Potential von ADO im allgemeinen und der Datenumgebung im speziellen wirklich nutzen zu können, müssen Sie die Grundlagen von SQL kennen. Und genau darum soll es im nächsten Kapitel, dem letzten Grundkapitel des Buches übrigens, gehen.

6.9 Fragen

Frage 1:

Beschreiben Sie die Aufgabe eines Designers in einem Satz.

Frage 2:

Wie unterscheidet sich ein Designer von einem Add-In?

Frage 3:

Was ist eine Datenumgebung?

Frage 4:

Ist eine Datenumgebung eine Alternative oder eine Ergänzung zu den ADO-Objekten?

Frage 5:

Profitieren auch Visual-Basic-Programmierer, die »nur« auf eine Access-97-Datenbank zugreifen möchten, vom Datenumgebungs-Designer?

Frage 6:

Ein Projekt basiert auf einer Datenumgebung für die Anbindung an die Datenbank. Unter mysteriösen Umständen geht die Designerdatei verloren und kann nicht mehr hergestellt werden. Welche Möglichkeit gibt es, auch ohne Datenumgebung den Zugriff auf die Datenbank durchführen zu können.

Die Antworten zu den Fragen finden Sie im Anhang D.

Kapitel 7

Das ABC der Datenbank-sprache SQL

In diesem Kapitel lernen Sie die Datenbank(abfrage)sprache SQL kennen. SQL ist, wie bereits in Kapitel 1 erwähnt (und versprochen) wurde, ein universelles Mittel, um Daten aus einer Datenbank herauszuholen, um Daten in einer Datenbank »abzulegen«, aber auch, um den Aufbau von Datenbanken und deren Tabellen und Feldern festzulegen. Auch wenn Microsoft Access, Visual Basic 6.0 im Rahmen des Datenumgebungs-Designers und andere Entwicklungswerkzeuge komfortable SQL-Generatoren offerieren, welche die Eingabe einfacher SQL-Kommandos überflüssig machen sollen, kommen angehende Datenbankprogrammierer nicht ohne Grundkenntnisse in SQL aus. Es ist erstaunlich, welches Potential in SQL steckt und wie ein gut durchdachtes SQL-Kommando den Programmaufbau vereinfachen kann. Es lohnt sich daher, Zeit in ein Thema zu investieren, das bei oberflächlicher Betrachtung vielleicht den Eindruck erweckt, es wäre nicht so wichtig.

Sie erfahren in diesem Kapitel etwas zu den folgenden Themen:

- ✘ Ein kurzer Rückblick auf die Entstehung von SQL
- ✘ Eine Übersicht über die wichtigsten SQL-Kommandos
- ✘ Die Datendefinitionssprache (DDL) und die Datenmanipulationssprache (DML)
- ✘ Das SQL-Kommando *SELECT*
- ✘ Beispiele für einfache Datenbankabfragen
- ✘ Die SQL-Kommandos *UPDATE*, *INSERT* und *DELETE*

- ✘ SQL-Aggregatfunktionen
- ✘ Verknüpfungen mehrerer Tabellen

Am Ende des Kapitels besitzen Sie einen Überblick über die Möglichkeiten von SQL, kennen wichtige SQL-Kommandos und können einfache Datenbankabfragen mit SQL in einem VBA-Programm oder mit Hilfe anderer Werkzeuge ausführen.

7.1 Ein kurzer Rückblick zum Thema SQL

Dieser Abschnitt soll selbstverständlich niemanden über Gebühr langweilen, doch ein kurzer Rückblick muß sein. SQL ist nämlich keine Erfindung von Microsoft oder einer der zahlreichen Software-Schmieden aus dem Silicon Valley, es stammt vielmehr aus den 70er Jahren, als es noch keine PCs gab und Programmierer ernsthafte Menschen mit einer geregelten Arbeitszeit waren, die mit weißen Kitteln bekleidet in klimatisierten Räumen ihrer Tätigkeit nachgingen. Auch wenn SQL zunächst nur für Großrechner gedacht war, konnte es sich Mitte der achtziger Jahre auch in der PC-Welt etablieren. Heute ist es völlig selbstverständlich, daß jedes ernstzunehmende DBMS SQL für Datenbankoperationen anbietet. Da es seit längerem einen weltweit anerkannten ANSI-Standard für SQL gibt (SQL 92) und sich die verschiedenen »SQL-Dialekte«, die trotz Standard existieren, sehr ähnlich sind, ist das für Datenbankprogrammierer eine wunderbare Angelegenheit. Das wäre in etwa so, als würde man für jedes Betriebssystem der Welt Scriptprogramme in Basic (oder noch besser in VBA) programmieren können.

Auch wenn SQL ursprünglich für »Structured Query Language«, also strukturierte Abfragesprache, stand (dieser inzwischen nicht mehr offiziell gültige Begriff stammt aus den 70er Jahren, als das Attribut »strukturiert« einen besonderen Klang bei den Programmierern hatte und soviel wie »fortschrittlich« bedeutete)[1], lassen sich mit SQL-Kommandos auch ganze Datenbanken anlegen, Tabellen hinzufügen oder löschen. SQL ist jedoch keine richtige Programmiersprache, es gibt z.B. keine Programmschleifen, Entscheidungen oder Variablen (diese werden von den verschiedenen SQL-Dialekten, wie z.B. T-SQL beim Microsoft SQL-Server hinzugefügt). SQL umfaßt relativ wenige Kommandos, eine sehr viel größere Anzahl an Schlüsselwör-

1 Würde SQL heute erfunden, hieße es womöglich Active Database Basic, womit die, völlig fiktive, Abkürzung ADB jedem Datenbankprogrammierer geläufig wäre – oder zumindest geläufig sein sollte.

tern und eine Reihe einfacher Funktionen. SQL ist rein kommandoorientiert, d.h., es gibt keine SQL-Programme[1]. SQL-Kommandos werden einzeln ausgeführt und sind stets unabhängig von eventuell vorausgegangenen oder folgenden SQL-Kommandos. Den besten Eindruck erhalten Sie, wenn Sie sich das kleine SQL-Beispiel im übernächsten Abschnitt in Ruhe zu Gemüte führen.

7.1.1 AccessSQL

Microsoft verwendet für ihr Desktop-DBMS Access einen Dialekt, der zwar in vielen wichtigen Bereichen mit ANSI SQL 92 übereinstimmt, aber auch eine Reihe von Unterschieden aufweist. Dieser SQL-Dialekt heißt *AccessSQL* und wird in diesem Buch vorgestellt (mit Microsoft Access 2000 und der Jet-Engine 4.0 geht AccessSQL in T-SQL, dem zweiten SQL-Dialekt von Microsoft über, das sehr nahe an SQL 92 angelehnt ist).

7.1.2 SQL und VBA

SQL ist kein Bestandteil von VBA. Es ist einzig und allein Sache des DBMS, SQL zu unterstützen. Da dies bei der Jet-Engine mit AccessSQL der Fall ist und diese wiederum über die ADO- oder DAO-Objekte von VBA aus zugänglich ist, können Sie in VBA per SQL Datenbankabfragen durchführen. Die SQL-Kommandos werden dabei allerdings nicht direkt in eine Befehlszeile geschrieben, sondern als Zeichenketten (in diesem Fall werden sie von VBA nicht interpretiert) an die Jet-Engine geschickt. Eine SendSQL- oder SQLExecute-Methode gibt es allerdings nicht, das SQL-Kommando wird vielmehr als Zeichenfolge einzelnen Eigenschaften zugewiesen oder einer Methode übergeben. Sobald z.B. ein *Recordset*-Objekt geöffnet wird, kommt das SQL-Kommando zur Ausführung und verrichtet in der Datenbank seine Arbeit. VBA spielt also nur den »Mittelmann«, der SQL-Kommandos an die Jet-Engine schickt. Dort werden sie analysiert, gegebenenfalls optimiert und ausgeführt.

Damit die VBA-Befehle von den SQL-Befehlen besser unterscheidbar sind, werden letztere in diesem Buch als *SQL-Kommandos* bezeichnet. Außerdem werden in diesem Buch Großbuchstaben verwendet, obwohl die Groß-/Kleinschreibung bei SQL-Kommandos keine Rolle spielt.

1 Von den verschiedenen herstellerspezifischen SQL-Sprachen, wie T-SQL (Microsoft SQL Server) oder PL/SQL (Oracle), die eine Reihe von Erweiterungen einführen, abgesehen.

Damit Sie sich diesen sehr wichtigen Aspekt besser vorstellen können, zeigt das folgende Codebeispiel, wie eine Datenbankabfrage per SQL in VBA durchgeführt wird.

```
Dim Rs As ADODB.Recordset
Dim Cn As ADODB.Connection
Dim SQLKommando As String
Set Rs = New ADODB.Recordset
Set Cn = New ADODB.Connection
SQLKommando = "SELECT * FROM Modelldaten WHERE " & _
   "Geschwindigkeit = (SELECT MAX(Geschwindigkeit) " & _
   "FROM Modelldaten)"

With Cn
 .Provider = "Microsoft.Jet.OLEDB.3.51;"
 .ConnectionString = "C:\Eigene Dateien\Fuhrpark.mdb"
 .Open
End With

With Rs
 .Source = SQLKommando
 .ActiveConnection = Cn
 .Open
End With

MsgBox Prompt:="Der schnellste Wagen ist: " &
Rs.Fields("ModellName").Value & " mit " & _
Rs.Fields("Geschwindigkeit").Value & " km/h"

Rs.Close
Cn.Close
```

Sie können diese Befehle z.B. in die *Form_Load*-Prozedur eintippen und das Programm zur Ausführung bringen. Vorausgesetzt, die Active Data Objects wurden zuvor als Referenz eingebunden (siehe Kapitel 5) und im Verzeichnis *C:\Eigene Dateien* befindet sich die Datenbank *Fuhrpark.mdb* mit einer Tabelle *Modelldaten*. Dann wird das Fahrzeug mit der höchsten Geschwindigkeit angezeigt.

Haben Sie erkannt, wo SQL im Spiel ist? Die Variable *SQLKommando* enthält den Kommandotext. Doch da es sich um eine Stringvariable handelt, ist für VBA der Inhalt unerheblich. Erst durch die Übergabe der Stringvariablen beim Aufruf der *Open*-Methode des *Recordset*-Objekts wird das

SQL-Kommando an die Jet-Engine übermittelt und dort ausgeführt. Das Ergebnis der Abfrage wird der Variablen *Rs* in Gestalt einer Datensatzgruppe zugewiesen. Nach diesem Prinzip werden alle SQL-Abfragen ausgeführt (für jene SQL-Kommandos, die keine Datensätze zurückgeben, etwa der Befehl *DELETE* zum Löschen eines Datensatzes, gibt es z.B. die *Excute*-Methode des *Connection*-Objekts).

SQL-Kommandos sind nicht von der Groß-/Kleinschreibung (oder einem Zeilenumbruch) abhängig. Es spielt daher keine Rolle, ob Sie SELECT oder select eingeben.

7.2 Ein Überblick über SQL

SQL besteht, ähnlich einer Programmiersprache, aus Kommandos, Funktionen, Schlüsselwörtern und Operatoren. Die SQL-Kommandos lassen sich in zwei Gruppen unterteilen:

- ✘ Die Kommandos der *Data Definition Language* (DDL).
- ✘ Die Kommandos der *Data Manipulation Language* (DML).

Mit den Kommandos der Datendefinitionssprache (DDL), wie z.B. *CREATE TABLE*, erstellen Sie Tabellen, Indizes oder fügen Felder einer Tabelle hinzu. Mit den Kommandos der Datenbearbeitungssprache (DML) führen Sie Abfragen durch. Ein Beispiel ist das *SELECT*-Kommando, das in ca. 80% aller Datenbankabfragen verwendet werden dürfte. In diesem Buch stehen die DML-Kommandos im Vordergrund, da sie am häufigsten benötigt werden. Auch wenn es alles andere als kompliziert ist, werden einzelne Tabellen oder gar eine ganze Datenbank in der Praxis mit einem entsprechenden Werkzeug, wie etwa Microsoft Access, und nicht mit den DDL-Kommandos erstellt. Diese bieten allerdings den Vorteil, daß sie jederzeit 1:1 reproduzierbar sind, was man bei einer mit Microsoft Access erstellten Datenbank nur selten sagen kann.

7.2.1 Eine Übersicht über die wichtigsten SQL-Schlüsselwörter

Wenn von SQL-Schlüsselwörtern die Rede ist, sind damit alle Kommandos, Funktionen und sonstige »Befehlswörter« gemeint, die in einem SQL-Kommando auftauchen können. Das SQL-Schlüsselwort *FROM* ist beispiels-

weise kein Kommando, da es nur im Zusammenhang mit dem *SELECT*-Kommando zum Einsatz kommen kann. Das gleiche gilt für das Schlüsselwort *ORDER BY* oder die Aggregatfunktion *MAX*.

Tabelle 7.1: Die Access-SQL-Kommandos in der Übersicht

SQL-Kommando	Bedeutung	Syntax
ALTER TABLE	Ändert den Entwurf einer Tabelle, nachdem dieser mit CREATE TABLE erstellt wurde.	*ALTER TABLE Tabelle {ADD {COLUMN Feld Typ[(Größe)] [NOT NULL] [CONSTRAINT Index]* | *CONSTRAINT Mehrfelderindex}* | *DROP {COLUMN Feld I CONSTRAINT Indexname } }*
CREATE INDEX	Legt einen neuen Index für eine bereits vorhandene Tabelle an.	*CREATE [UNIQUE] INDEX Index ON Tabelle (Feld [ASC | DESC][, Feld [ASC | DESC], ...]) [WITH { PRIMARY* | *DISALLOW NULL* | *IGNORE NULL }]*
CREATE TABLE	Legt eine neue Tabelle an.	*CREATE TABLE Tabelle (Feld1 Typ [(Größe)] [NOT NULL] [Index1] [, Feld2 Typ [(Größe)] [NOT NULL] [Index2] [, ...]] [, CONSTRAINT Mehrfelderindex [, ...]])*
DELETE	Löscht einen oder mehrere Datensätze (aber nicht die gesamte Tabelle, dafür kann DROP TABLE verwendet werden).	*DELETE [Tabelle.*] FROM Tabelle WHERE Kriterien*
DROP	Löscht eine Tabelle oder einen Index.	*DROP {TABLE Tabelle* | *INDEX Index ON Tabelle}*
INSERT INTO	Fügt einer Tabelle einen oder mehrere Datensätze hinzu (Anfügeabfrage).	*INSERT INTO Ziel [(Feld1[, Feld2[, ...]])] VALUES (Wert1[, Wert2[, ...])*

Tabelle 7.1: Die Access-SQL-Kommandos in der Übersicht (Fortsetzung)

SQL-Kommando	Bedeutung	Syntax
SELECT	Wählt einen oder mehrere Feldnamen aus. Der * wählt alle Feldnamen aus.	*SELECT [Prädikat] { * | Tabelle.* | [Tabelle.]Feld1 [AS Alias1] [, [Tabelle.]Feld2 [AS Alias2] [, ...]]}* *FROM Tabellenausdruck [, ...] [IN ExterneDatenbank]* *[WHERE...]* *[GROUP BY...]* *[HAVING...]* *[ORDER BY...]* *[WITH OWNERACCESS OPTION]*
TRANSFORM	Erstellt eine Kreuztabellenabfrage (Pivot-Tabelle).	*TRANSFORM AggFunktion* *Auswahlanweisung* *PIVOT Pivot-Feld [IN (Wert1[, Wert2[, ...]])]*
UPDATE	Erstellt eine sogenannte Aktualisierungsabfrage, die Werte in Feldern einer angegebenen Tabelle aufgrund angegebener Kriterien ändert.	*UPDATE Tabelle* *SET NeuerWert* *WHERE Kriterien;*
SHAPE	Erstellt einen hierarchischen Recordset, wo ein Feld des obersten Recordset-Objekts auf einen anderen Recordset verweist (Microsoft- und OLE DB-spezifisch).	Siehe MSDN-Hilfe.

Tabelle 7.2: Die wichtigsten SQL-Schlüsselwörter

SQL-Schlüsselwort	Bedeutung
AS	Wird in einer *SELECT*-Abfrage verwendet, um einem Feld einen Aliasnamen zu geben.
ASC	Legt bei einer Sortierung über *ORDER BY* die absteigende Sortierreihenfolge fest.
BETWEEN	SQL-Operator, der bestimmt, ob ein Wert in einem Wertebereich liegt.
CONSTRAINT	Wird bei *ALTER TABLE* und *CREATE TABLE* verwendet, um gewisse Einschränkungen, wie z.B. ein Index, der nicht mehrfach vorkommen oder *NULL* sein darf, einzugeben.
DESC	Legt bei einer Sortierung über *ORDER BY* die aufsteigende Sortierreihenfolge fest.
DISTINCT	Wählt in einer *SELECT*-Abfrage nur jene Datensätze aus, die in den ausgewählten Feldern keine mehrfach vorkommenden Daten enthalten.
FROM	Legt fest, auf welche Tabelle(n) sich eine *SELECT*-Abfrage bezieht.
GROUP	Faßt Datensätze in einer *SELECT*-Abfrage, die in den angegebenen Felder die gleichen Werte enthalten, zusammen. Wird in der Regel zusammen mit einer SQL-Aggregatfunktion eingesetzt.
HAVING BY	Führt in einer *SELECT*-Abfrage mit *GROUP BY*-Gruppierung eine Selektion durch.
INNER JOIN	Gibt bei einer *SELECT*-Abfrage eine *INNER JOIN*-Verknüpfung an.
ORDER BY	Sortiert die Datensätze in einer Datensatzgruppe.
TOP	Wird in der Form *TOP n [PERCENT]* in einer *SELECT*-Abfrage eingesetzt und gibt nur die obersten n% der Datensätze zurück.
UNION	Erstellt in einer *SELECT*-Abfrage eine sogenannte Union-Abfrage, in der die Ergebnisse zweier oder mehrerer unabhängiger Abfragen oder Tabellen mit gleicher Anzahl an Spalten kombiniert werden. *UNION* ist kein Teil von SQL-92.
WHERE	Legt eine Bedingung fest, die bestimmt, welche Zeilen der beteiligten Tabellen in die Ergebnismenge übernommen werden.

7.3 Der Visual Data Manager als »SQL-Trainer«

Das kleine VBA-Beispiel in Kapitel 7.1 hat gezeigt, wie SQL-Kommandos in einem VBA-Programm abgeschickt werden. Wäre es nicht toll, wenn es bereits ein fertiges Programm gäbe, bei dem man SQL-Kommandos eintippen und das Ergebnis sehen kann, ohne dies umständlich im Quelltext festlegen zu müssen? Das wäre ungemein praktisch, denn damit ließen sich die verschiedenen SQL-Variationen nach Herzenslust ausprobieren. Zum Glück gibt es solche Programme und zwar in einer erstaunlichen Vielzahl. Da wäre z.B. Microsoft Access, bei dem Sie seit der ersten Version Abfragen durch Auswahl der verschiedenen Tabellen und Felder mit der Maus zusammenstellen und das resultierende SQL-Kommando in einem separaten Fenster betrachten können. Da wäre das inzwischen ein wenig »altbackene« MS-Query, das in Office 97 mit von der Partie ist. Einen ähnlichen Komfort bietet inzwischen auch Visual Basic im Rahmen des SQL-Abfragegenerators des Datenumgebung-Designers (siehe Kapitel 6), der aber leider nicht Bestandteil der Ablaufmodell/Einsteiger-Version von Visual Basic 6.0 ist. Ein Hilfsprogramm, über das alle Visual-Basic-Versionen verfügen, ist der Visual Data Manager (VDM), der in Kapitel 3 vorgestellt wird, und der sich wunderbar als einfacher SQL-Trainer einsetzen läßt.

Um mit dem VDM ein SQL-Kommando austesten zu können, gehen Sie stets wie folgt vor:

1. Starten Sie den Visual Data Manager, und laden Sie die Datenbank (z.B. *Fuhrpark.mdb*).
2. Öffnen Sie gegebenenfalls über das FENSTER-Menü das Fenster *SQL-Anweisung*.
3. Tippen Sie das SQL-Kommando in das Fenster ein, und klicken Sie wahlweise auf die *Ausführen*-Schaltfläche, oder drücken Sie die ↵-Taste.
4. Bestätigen Sie stets die erfolgende Anfrage, ob es sich um eine SQL-Pass-Through-Anfrage handelt, mit »Nein« (die Option »Ja« spielt nur dann eine Rolle, wenn Sie sich mit einer ODBC-Datenbank verbunden haben – in diesem Fall werden die SQL-Kommandos an der Jet-Engine vorbeigeleitet, was einen deutlichen Geschwindigkeitsgewinn bewirken kann).
5. Das Ergebnis der Abfrage wird daraufhin in einem eigenen Fenster angezeigt.

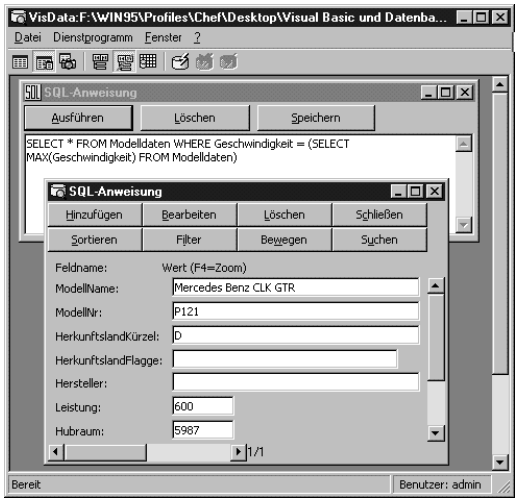

Bild 7.1: Der Visual Data Manager zeigt das Ergebnis einer SQL-Abfrage in einem eigenen Fenster an

Ganz so toll ist der Komfort des VDM natürlich nicht. So erhalten Sie die Ergebnismenge in einem Fenster zurück, das nur einen Datensatz auf einmal anzeigt und daher das Blättern in der Ergebnismenge erforderlich macht. Übersichtlicher wäre manches Mal eine tabellarische Darstellung, wie sie auch in vielen SQL-Einführungsbüchern verwendet wird. Doch man kann bekanntlich nicht alles haben.

Leider kann das SQL-Anweisungsfenster immer nur eine SQL-Anweisung enthalten. Gerade beim Kennenlernen ist es jedoch praktisch, eine komplexere SQL-Anweisung nicht jedesmal komplett neu eintippen zu müssen. Sie sollten es sich daher zur Angewohnheit machen, vor der Eingabe eines neuen SQL-Kommandos das alte Kommando, zusammen mit einem kurzen Kommentar, der das Kommando erklärt, in ein Notepad-Fenster zu kopieren und den gesamten Inhalt als Textdatei zu speichern. Auf diese Weise können Sie sich häufig benutzte SQL-Kommandos aus einem Fenster auswählen und müssen diese nicht jedesmal neu eingeben. Diese Lösung ist sicherlich nicht der Weisheit letzter Schluß, aber immerhin eine Hilfe.

Sie können zwar im SQL-Anweisungsfenster des VDM nur jeweils ein SQL-Kommando ausführen, aber mehrere Ereignismengen parallel betrachten.

Die *Speichern*-Schaltfläche speichert, anders als man es vielleicht vermuten könnte, nicht das aktuelle SQL-Kommando in einer Textdatei ab. Es wandelt das SQL-Kommando vielmehr in eine gespeicherte Abfrage um und macht diese zu einem Bestandteil der Datenbank. Diese Option sollten Sie immer dann nutzen, wenn Sie eine Abfrage ausgetestet haben und diese von nun an verwenden möchten. Gegenüber einem SQL-Kommando, das lediglich in Textform an die Datenbank geschickt wird, bieten gespeicherte Abfragen einen Geschwindigkeitsvorteil, da sie nicht jedesmal neu in ihre Bestandteile zerlegt werden müssen.

7.3.1 SQL-Abfragen mit Komfort

Der Visual Data Manager bietet für das Anfertigen von SQL-Kommandos einen Komfort, den man dem Programm gar nicht zutraut. Anstatt ein SQL-Kommando in das Textfenster einzutippen, können Sie sich die Abfrage mit Hilfe des Dienstprogramms Abfragegenerator, ähnlich wie in Microsoft Access, auch zusammenstellen. Der Abfragegenerator bietet zwar bei weitem nicht alle Möglichkeiten, so werden keine Aggregatfunktionen angeboten, doch für den Anfang reicht es aus.

Die folgende Schrittfolge zeigt, wie die SQL-Abfrage

```
SELECT ModellName, Geschwindigkeit FROM Modelldaten WHERE
Geschwindigkeit>160
```

mit Hilfe des Abfragegenerators generiert und ausgeführt wird:

1. Starten Sie den Visual Data Manager, und öffnen Sie die Datenbank *Fuhrpark.mdb*.
2. Starten Sie über das Menü DIENSTPROGRAMM den Abfragegenerator.
3. Selektieren Sie in der Auswahlliste *Tabellen* den Eintrag »Modelldaten«.
4. Wählen Sie aus der Auswahlliste *Feldname* den Feldnamen *Modelldaten.Geschwindigkeit* aus (der Abfragegenerator stellt den Tabellennamen voraus, auch wenn dies nicht erforderlich wäre).
5. Wählen Sie aus der Auswahlliste *Operator* den Vergleichsoperator, in diesem Fall ist es der Eintrag »>«.
6. Tragen Sie in das Eingabefeld *Wert* den Vergleichswert ein, in diesem Fall ist es 160.

7. Klicken Sie auf die Schaltfläche *And in Kriterien*, damit diese Bedingung in die Abfrage eingebaut wird.
8. Wählen Sie in der Auswahlliste *Anzuzeigende Felder* jene Felder aus, die in die Ergebnismenge übernommen werden sollen. Durch Drücken der ⇧-Taste werden mehrere Felder selektiert. Wählen Sie die Felder *Modelldaten.Modellname* und *Modelldaten.Geschwindigkeit*.
9. Klicken Sie auf die *Anzeigen*-Schaltfläche, um das erzeugte SQL-Kommando zu kontrollieren.
10. Klicken Sie auf die *Kopieren*-Schaltfläche, um das erzeugte SQL-Kommando in die Zwischenablage zu übernehmen. Sie können es auf diese Weise bequem in ein VBA-Programm einfügen.
11. Klicken Sie auf die *Ausführen*-Schaltfläche, um das erzeugte SQL-Kommando auszuführen (beantworten Sie die Frage »Handelt es sich um eine SQLPassThrough-Abfrage« mit »Nein«). Die Ergebnismenge wird daraufhin in einem eigenen Fenster angezeigt.

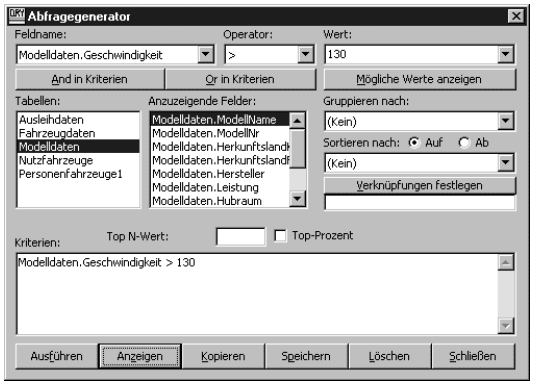

Bild 7.2: Eine SQL-Abfrage wurde mit dem SQL-Abfragegenerator generiert

Möchten Sie eine (funktionierende) Abfrage in der Datenbank »verewigen«, klicken Sie auf die *Speichern*-Schaltfläche. Dadurch wird in der Datenbank eine neue Abfrage angelegt, die wie eine Tabelle geöffnet wird.

In den folgenden Abschnitten lernen Sie die wichtigsten SQL-Kommandos und -Funktionen kennen. Auf das Austesten dieser Kommandos im Visual Data Manager wird nicht mehr eingegangen. Nehmen Sie sich aber die Zeit, und probieren Sie alles in Ruhe aus. Sie wissen nun, wie es geht.

Wo erfahre ich denn »alles« über SQL? Wo gibt es z.B. eine komplette Befehlsreferenz? Nun, ein eigenes SQL-Handbuch wäre natürlich eine feine Sache, doch leider bietet auch die Enterprise Edition von Visual Basic 6.0 diesen »Luxus« nicht. Eine SQL-Referenz ist in der MSDN-Hilfe »tief vergraben« und leider nicht einfach zu finden (in der Rubrik des Microsoft SQL Servers finden Sie noch die ausführlichste Beschreibung zu SQL und eine Beschreibung von T-SQL, einem SQL-Dialekt, der mit jenem, den die Jet-Engine verwendet, in den wichtigsten Punkten übereinstimmt[1]. Wer alles über SQL wissen will, muß sich eines der zahlreichen Fachbücher zu diesem Thema zulegen.

7.4 Allgemeine Regeln zum Umgang mit SQL

Bevor es losgeht, sollen Sie ein paar allgemeine Regeln kennenlernen, so daß Sie auch, z.B. mit dem Visual Data Manager, eigene »Experimente« mit SQL anstellen können, ohne das komplette Kapitel lesen zu müssen:

- ✘ Die Groß-/Kleinschreibung spielt bei SQL-Kommandos keine Rolle (das wurde ja schon erwähnt).
- ✘ Bei »SQL-Interpretern«, wie Microsoft Access oder dem Visual Data Manager, spielt es keine Rolle, ob das SQL-Kommando in einer Zeile eingegeben oder auf mehrere Zeilen verteilt wird.
- ✘ Wie bei VBA muß auf jedes SQL-Kommando mindestens ein Leerzeichen folgen.
- ✘ Zeichenketten werden in einfache Apostrophe gesetzt.
- ✘ Enthält ein Feldname ein Leerzeichen oder ein anderes Zeichen, das bei SQL eine spezielle Bedeutung besitzt, muß er in eckige Klammern gesetzt werden.
- ✘ Möchten Sie sich auf ein Feld in einer bestimmten Tabelle beziehen, muß dem Feldnamen getrennt durch einen Punkt der Tabellenname vorausgehen (z.B. *Modelldaten.ModellName*). Das ist aber kein Objek-

1 Es soll an dieser Stelle festgehalten werden, daß selbst im Hause Microsoft zwei SQL-Versionen verwendet werden, die sich mehr oder weniger geringfügig unterscheiden. Mit Access 2000 wird der SQL-Dialekt der Jet-Engine aber weitestgehend an T-SQL und damit auch an ANSI92-SQL angeglichen. So ist als Wildcard beim Textvergleich neben dem * auch das %-Zeichen erlaubt.

tausdruck, d.h., erwarten Sie nicht, daß sich nach der Eingabe des Punktes eine Auswahlliste öffnet)[1].

- ✘ Datumsangaben werden bei der Jet-Engine in »#«-Zeichen gesetzt (z.B. #25/10/1961#).
- ✘ Offiziell wird ein SQL-Kommando durch ein Semikolon abgeschlossen, das bei der Jet-Engine aber entfallen kann. Das Anhängen eines Semikolons erleichtert die Portierung auf andere Systeme.

Der Visual Data Manager hat offenbar kleinere Probleme bei der Datumsdarstellung. Bei der Eingabe stehen im Datensatzfenster nur vier Zeichen zur Verfügung, bei Datumsvergleichen muß das Datum im Format »#Monat/Tag/Jahr#« angegeben werden.

Wenn Sie sich an diese allgemeinen Regeln halten, sollte am Anfang nicht viel schief gehen. Enthält ein SQL-Kommando einen Fehler, bekommen Sie meist eine relativ aussagekräftige Fehlermeldung zurück. Und, abstürzen kann ein SQL-Programm nicht.

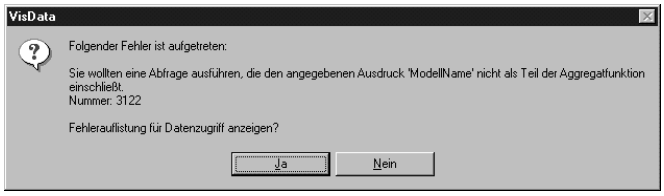

Bild 7.3: Der Visual Data Manager zeigt Syntaxfehler in einem SQL-Kommando an

7.5 Das SELECT-Kommando

In diesem Abschnitt lernen Sie mit dem *SELECT*-Kommando das wichtigste und am häufigsten eingesetzte SQL-Kommando kennen. Das *SELECT*-Kommando ist dazu da, eines oder mehrere Felder in einer oder mehreren Tabellen auszuwählen. Die Vielseitigkeit des *SELECT*-Kommandos wird am besten an der Syntaxbeschreibung deutlich. Erschrecken Sie bitte nicht angesichts des Umfangs. In einer Syntaxbeschreibung werden stets alle Möglichkeiten zusammengefaßt. Dennoch ist die Anwendung des *SELECT*-Kommandos meist sehr einfach, wie auch die folgenden Beispiele zeigen. Aus der Syntaxbeschreibung können Sie z.B. entnehmen, daß auf das

1 Das ist natürlich nicht ganz ernst gemeint, doch bei dem Komfort, den Visual Basic inzwischen zu bieten hat, vergißt man schnell, daß nicht alles aus Objekten besteht.

SELECT-Kommando mindestens ein Feldname (oder der Stern), das Schlüsselwort *FROM* sowie mindestens ein Tabellenname folgen müssen. Alles andere ist optional.


```
SELECT [Prädikate] Feldnamen FROM Tabellen
[WHERE Bedingung]
[GROUP BY Feldnamen]
[HAVING Gruppenbedingung]
[ORDER BY Sortierkriterium]
```

Tabelle 7.3: Die Elemente der SELECT-Syntaxbeschreibung

Syntaxelement	Bedeutung
SELECT	Schlüsselwort. Muß stets als erstes aufgeführt werden.
Prädikate	Legen fest, welche Datensätze zurückgegeben werden. Zur Auswahl stehen ALL, DISTINCT, DISTINCTROW und TOP.
Feldnamen	Stehen für die Feldnamen in den zurückgegebenen Datensätzen. Diese können auch aus einem Ausdruck hervorgehen. Ein »*« steht für alle Felder. Ist ein Feldname mehrdeutig, muß ihm der Tabellenname, getrennt durch einen Punkt, vorausgehen.
FROM	Schlüsselwort. Muß stets auf SELECT folgen.
Tabellen	Name(n) der Tabelle(n), aus denen die Datensätze stammen.
WHERE Bedingung	Schlüsselwort. Legt eine optionale Bedingung fest, die die Gruppe der zurückgegebenen Datensätze einschränkt.
GROUP BY Feldnamen	Schlüsselwort. Legt eine optionale Bedingung fest, die Datensätze unter Verwendung einer Aggregatfunktion zusammenfaßt.
HAVING Gruppenkriterium	Schlüsselwort. Legt ein Kriterium fest, das über GROUP BY zusammengefaßte Datensätze erfüllen müssen.
ORDER BY Sortierkriterium	Schlüsselwort. Legt ein Kriterium fest, nach dem die zurückgegebenen Datensätze sortiert werden.

```
SELECT * FROM Modelldaten
```

Dieses Kommando wählt alle Datensätze und alle Felder der Tabelle *Modelldaten* aus.

Möchte man, z.B. aus Gründen der Effektivität, nicht alle Felder zurückerhalten, müssen die Feldnamen einzeln aufgeführt werden:

```
SELECT ModellName, Geschwindigkeit FROM Modelldaten
```

Dieses Kommando gibt zwar alle Datensätze, aber nur mit den Feldern *ModellName* und *Geschwindigkeit* zurück. Die Ergebnismenge umfaßt daher nur zwei Spalten.

Rein formell und immer dann, wenn mehrere Tabellen beteiligt sind, die identische Feldnamen enthalten, kann bzw. muß dem Feldnamen der Tabellenname vorausgehen:

```
SELECT Modelldaten.ModellName, Modelldaten.Geschwindigkeit
FROM Modelldaten
```

Der Fall, daß alle Datensätze zurückgegeben werden sollen, ist in der Datenbankpraxis eher die Ausnahme. Es ist ja gerade die Stärke von SQL, Entscheidungen auf der Grundlage von Vergleichen durchführen zu können. Für die Festlegung, welche Datensätze zur Ergebnismenge gehören sollen, kommt das Schlüsselwort *WHERE* zum Einsatz:

```
SELECT * FROM Modelldaten WHERE Geschwindigkeit > 160
```

Diese Abfrage gibt alle Datensätze zurück, bei denen das Feld *Geschwindigkeit* einen Wert größer 160 besitzt. Auch wenn diese Ähnlichkeit vermutlich eher zufälliger Natur ist, verwendet SQL für Vergleiche ähnliche Operatoren wie VBA (siehe Tabelle 7.3).

Für die *WHERE*-Klausel stehen nicht nur die SQL-Operatoren, sondern auch beliebige VBA-Funktionen zur Verfügung.

```
SELECT * FROM Modelldaten
WHERE Instr(ModellName, "Coupe") <> 0;
```

Diese Abfrage gibt alle Datensätze zurück, in denen beim Feld *ModellName* das Wort »Coupe« vorkommt.

Operator	Bedeutung
=, <, >, <=, >=, <>	Die von VBA bekannten Vergleichsoperatoren.
LIKE	Prüft, ob ein Feld einen Teilausdruck enthält (z.B. WHERE Nachname LIKE 'S*'). Wird in der Regel zusammen mit einem Platzhalter eingesetzt.
BETWEEN	Prüft, ob ein Feld in einem durch zwei Grenzwerte festgelegten Bereich liegt (z.B. WHERE AnschaffungsDatum BETWEEN #04/1/95# AND #07/1/95#).
IN	Prüft, ob ein Ausdruck mit einem Wert aus einer Liste übereinstimmt (z.B. WHERE Stadt IN ('Bonn', 'Gießen', 'München', 'Erkrath').

Tabelle 7.4: SQL-Operatoren

7.5.1 Doppelte Datensätze ausfiltern mit DISTINCT

Um zu verhindern, daß eine *SELECT*-Abfrage Datensätze mit identischen Feldern zurückgibt, muß das Schlüsselwort (auch Prädikat genannt) *DISTINCT* verwendet werden.

```
SELECT DISTINCT HerkunftslandKürzel FROM Modelldaten
```

Dieses Kommando gibt nur jene Datensätze zurück, die unterschiedliche Werte im Feld *HerkunftslandKürzel* enthalten. Es ist wichtig zu verstehen, daß die Kombination der auf *DISTINCT* folgenden Felder eindeutig sein muß. Die folgende Abfrage hat keinen Filtereffekt, da die Kombination der Felder *HerkunftslandKürzel*, *ModellNr*, *Geschwindigkeit* bei jedem Datensatz unterschiedlich ist.

```
SELECT DISTINCT HerkunftslandKürzel, ModellNr,
Geschwindigkeit FROM Modelldaten
```

Datensatzgruppen, die über ein *SELECT DISTINCT*-Kommando zurückgegeben werden, sind nicht updatefähig.

Neben dem Prädikat *DISTINCT* kennt SQL noch das Prädikat *DISTINCTROW*. Es gibt im Rahmen einer *JOIN*-Abfrage, also einer Verknüpfung zweier Tabellen, keine mehrfach auftretenden Datensätze (und nicht mehrfach auftretende Felder wie bei *DISTINCT*) zurück.

Die folgende *JOIN*-Abfrage gibt die Modellnamen aller in der Tabelle *Fahrzeugdaten* enthaltenen Fahrzeuge zurück.

```
SELECT ModellName FROM Modelldaten INNER JOIN Fahrzeugdaten
ON Modelldaten.ModellNr=Modelldaten.ModellNr
```

Um dagegen alle in der Tabelle *Fahrzeugdaten* mehrfach vorkommenden Modellnamen auszufiltern, muß das Prädikat *DISTINCTROW* eingebaut werden:

```
SELECT DISTINCTROW ModellName FROM Modelldaten INNER JOIN
Fahrzeugdaten ON Modelldaten.ModellNr=Modelldaten.ModellNr
```

7.5.2 Das Prädikat TOP

Eine interessante Filtermöglichkeit bietet das *TOP*-Prädikat, denn es erlaubt es, die obersten n Prozent einer Tabelle zurückzugeben.

```
SELECT TOP 10 Preis, FahrzeugNr FROM Fahrzeugdaten
ORDER BY Preis DESC
```

Diese Abfrage gibt die Felder *Preis* und *FahrzeugNr* der zehn teuersten Fahrzeuge der Tabelle *Fahrzeugdaten* zurück, wobei die Datensätze nach dem Preis in absteigender Reihenfolge sortiert werden.

7.5.3 Vergleiche mit Namen

Der Vergleich mit Feldern, die Zahlen enthalten, bedarf keiner weiteren Erklärungen, denn zu VBA gibt es hier keine Unterschiede. Ein wenig anders sieht es aus, wenn in einem SQL-Kommando Felder verglichen werden, die Namen enthalten (also den Datentyp *Text*) besitzen. Hier gelten folgende Besonderheiten:

- ✘ Namenkonstanten werden in einfache Apostrophe gesetzt und nicht in Anführungsstriche. Das ist immer dann von Bedeutung, wenn ein SQL-Kommando in einem VBA-Programm als Stringkonstante eingebaut wird. Der Visual Data Manager (und viele andere »Einrichtungen«) akzeptieren aber auch die Anführungsstriche.
- ✘ SQL kennt offiziell keine Stringfunktionen, wie z.B. *Left*. Allerdings können bei der Jet-Engine VBA-Funktionen in SQL-Kommandos eingebaut werden. Das ist enorm praktisch, macht die SQL-Kommandos aber nicht portierbar.

- ✘ Für die Suche nach ähnlichen Namen gibt es den *LIKE*-Operator, dem als Platzhalter ein *-Zeichen (Zeichenkette) oder ein ?-Zeichen (einzelnes Zeichen) übergeben wird.

```
SELECT * FROM Modelldaten WHERE Modellname = 'VW'
```

Sollen dagegen alle Datensätze zurückgegeben werden, in denen die Silbe »VW« auftaucht, muß der *LIKE*-Operator zum Einsatz kommen:

```
SELECT * FROM Modelldaten WHERE Modellname LIKE'VW*'
```

Dieses SQL-Kommando gibt alle Datensätze zurück, bei denen im Feld *ModellName* die Silbe »VW« am Anfang steht (die Groß-/Kleinschreibung spielt hier keine Rolle). Sollen dagegen alle Datensätze gefunden werden, die »VW« an beliebigen Stellen enthalten, muß ein weiterer Platzhalter zum Einsatz kommen:

```
SELECT * FROM Modelldaten WHERE Modellname LIKE'*VW*'
```

Dieses Kommando gibt den Datensatz zurück, dessen Feld *ModellName* den Inhalt »VW« besitzt.

Möchte man lediglich einzelne Zeichen offen lassen, muß als Platzhalter das ? zur Anwendung kommen.

```
SELECT * FROM Modelldaten WHERE HerkunftslandKürzel LIKE 'S?'
```

Dieses Kommando gibt alle Datensätze zurück, deren Feld *Herkunftsland-Kürzel* als erstes Zeichen ein »S« und als zweites Zeichen ein beliebiges Zeichen enthält.

7.5.4 Ein Wort zu den Platzhaltern

AccessSQL (der SQL-Dialekt der Jet-Engine bis zur Version 3.51) verwendet den Stern als Platzhalter für eine Zeichenkette und das Fragezeichen für ein einzelnes Zeichen. Bei der Ausführung eines SQL-Kommandos innerhalb von VBA muß dagegen, wenn die Datenbankverbindung auf ADO basiert, das Prozentzeichen anstelle des Sterns und der Unterstrich anstelle des Fragezeichens verwendet werden:

```
SQLKommando = "SELECT * FROM Modelldaten WHERE ModellName
Like 'VW%'"
```

Da in diesem Buch sowohl SQL-Beispiele für den Visual Data Manager als auch für den VBA-Quellcode vorgestellt werden, kommen je nach Beispiel verschiedene Platzhalter zum Einsatz.

Der ANSI-SQL-Standard legt als Platzhalter das Prozentzeichen und den Unterstrich fest.

7.5.5 Vergleiche mit Stringvariablen

Wir kommen jetzt zu einem etwas »heikleren« Punkt, der angehenden Datenbankprogrammierern erfahrungsgemäß gewisse Verständnisschwierigkeiten bereitet (auch wenn das Ganze wirklich simpel ist). Wie baue ich denn eine Stringvariable in ein SQL-Kommando ein? Konkret, der Suchbegriff soll nicht im SQL-Kommando stehen, sondern durch eine Variable repräsentiert werden:

```
Dim Suchname As String
Suchname = "VW"
SQLKommando = "SELECT * FROM Modelldaten WHERE ModellName
Like ''" & Suchname "%'"
```

Das »Problem« dabei ist die Aufeinanderfolge von einfachen Apostrophen und doppelten Anführungszeichen, die in Listings nur schwer zu erkennen ist. Das SQL-Kommando erwartet stets, daß sich der Suchbegriff in einfachen Apostrophen befindet. Damit VBA jedoch nicht den Variablennamen, sondern den Inhalt der Variablen in den SQL-String einfügt, muß die Variable außerhalb der doppelten Anführungsstriche stehen, was eine nicht immer auf Anhieb überschaubare Stringverknüpfung erforderlich macht (insbesondere dann, wenn bei einer Abfrage mehrere Bedingungen verknüpft werden sollen).

Damit das Kennenlernen von SQL nicht an so einfachen Dingen scheitert, gibt es ein einfaches Hilfsmittel: Stellen Sie die einfachen Apostrophe durch ein *Chr(39)*-Zeichen dar, dadurch wird klarer erkennbar, an welcher Stelle sich ein einfaches Apostroph befindet:

```
Suchbegriff = "VW%"
SQLKommando = "SELECT * FROM Modelldaten WHERE ModellName
Like " & Chr(39) & Suchbegriff & Chr(39)
```

Jetzt wirkt der String schon gleich ein wenig übersichtlicher. Denken Sie daran, daß es der Jet-Engine (bzw. allgemein dem SQL-Preprocessor einer Datenbank) »egal« ist, auf welche Weise das SQL-Kommando im VBA-Programm zusammengesetzt wurde. Wichtig ist nur, daß die SQL-Syntax eingehalten wird.

7.5.6 Verknüpfte Bedingungen

Es dürfte Sie nicht überraschen, daß sich die Bedingungen in einer *FROM*-Klausel über die Operatoren *AND* und *OR* verknüpfen lassen. Was in SQL-Einführungsbüchern daher oft mehrere Seiten umfaßt, läßt sich für erfahrene VBA-Programmierer mit einem Beispiel abhandeln:

```
SELECT * FROM Modelldaten WHERE Modellname LIKE 'VW*' AND
Geschwindigkeit > 130
```

Dieses SQL-Kommando gibt alle Datensätze zurück, die zwei Bedingungen erfüllen:

1. Das Feld *Modellname* muß mit der Silbe »VW« beginnen.
2. Das Feld *Geschwindigkeit* muß einen Wert größer 130 enthalten.

7.5.7 Der Operator AND

Dieser Operator verknüpft zwei Bedingungen nach der logischen *UND*-Regel. Die Gesamtbedingung ist nur dann erfüllt, wenn beide Teilbedingungen erfüllt sind.

7.5.8 Der Operator OR

Dieser Operator verknüpft zwei Bedingungen nach der logischen *ODER*-Regel. Die Gesamtbedingung ist erfüllt, wenn eine der Teilbedingungen erfüllt ist. Das folgende SQL-Kommando gibt alle Datensätze zurück, in denen das Feld *HerkunftslandKürzel* entweder den Wert »S« oder den Wert »D« besitzt:

```
SELECT * FROM Modelldaten WHERE HerkunftslandKürzel = 'S' OR
HerkunftslandKürzel = 'D'
```

7.5.9 Der Operator NOT

Wie in VBA negiert auch der SQL-Operator *NOT* eine Bedingung.

```
SELECT* FROM Modelldaten WHERE NOT HerkunftslandKürzel = 'D'
```

Dieses SQL-Kommando gibt alle Datensätze zurück, deren Feld *HerkunftslandKürzel* nicht den Buchstaben »D« enthält.

7.5.10 Rechenoperationen in SQL-Kommandos

Innerhalb von SQL-Kommandos lassen sich Rechenoperationen durchführen, was sich beim ersten Lesen sicherlich ein wenig ungewöhnlich anhört. Daher gleich ein Beispiel.

```
SELECT FahrzeugNr, Preis - Preis / 10 FROM Fahrzeugdaten
```

Dieses Kommando gibt alle Datensätze zurück, zeigt aber nur zwei Spalten an. Während Spalte Nr. 1 den Inhalt des Feldes *FahrzeugNr* enthält, gibt Spalte Nr. 2 den um 10% verringerten Wert des Feldes *Preis* an.

Doch keine Sorge, der Inhalt der Datenbank wurde nicht verändert (diesen Job übernimmt das *UPDATE*-Kommando). Es wurde lediglich eine »virtuelle« Spalte in die Ergebnismenge eingefügt, wie sich durch eine erneute Abfrage schnell überprüfen läßt:

```
SELECT FahrzeugNr, Preis FROM Fahrzeugdaten
```

Wie das Ergebnisfenster bei der ersten Abfrage zeigt, erhält die virtuelle Spalte einen Standardnamen (in diesem Fall »Expr1001«). Sollte dies nicht gewünscht sein, muß die Spalte einen sogenannten Aliasnamen erhalten:

7.5.11 Felder können einen Aliasnamen besitzen

In Situationen, in denen ein SQL-Kommando einen Default-Namen, wie z.B. Expr1000, zurückgibt, oder in denen Feldnamen relativ lang sind, empfiehlt es sich, dem Feld einen neuen Namen zu geben. Dieser Name wird *Aliasname* genannt und über das Schlüsselwort *AS* festgelegt.

```
SELECT SUM(Preis) AS Gesamtpreis FROM Fahrzeugdaten
```

Dieses SQL-Kommando berechnet die Summe aller Felder der Spalte *Preis* und gibt dem Ergebnisfeld den Namen *Gesamtpreis*.

7.5.12 VBA-Funktionen

SQL-Kommandos, die in einem VBA-Programm ausgeführt werden, können VBA-Funktionen enthalten, was die Möglichkeit von SQL deutlich erweitert.

```
SELECT ModellName, Geschwindigkeit FROM Modelldaten WHERE
Left(Modellname,1)='S'
```

In diesem Beispiel wird das Filterkriterium für die *WHERE*-Klausel durch eine VBA-Funktion gebildet.

Das Microsoft-Jet-Datenbankmodul verwendet den VBA-Ausdruck-Interpreter, um einfache arithmetische Operationen und Funktionen auszuwerten. Alle Operatoren, die in Microsoft-Jet-SQL-Ausdrücken verwendet werden (außer *BETWEEN*, *IN* und *LIKE*), werden durch den Ausdruck-Interpreter von VBA definiert. Außerdem stellt der Ausdruck-Interpreter über 100 VBA-Funktionen zur Verfügung, die in SQL-Ausdrücken verwendet werden können.

7.6 Gruppieren von Datensätzen mit GROUP BY

Die Aggregatfunktionen eröffnen Möglichkeiten, die von einem *SELECT*-Kommando alleine nicht vollständig genutzt werden können. Angenommen, ein *SELECT*-Kommando gibt über die *AVG*-Funktion die Durchschnittsgeschwindigkeit aller Fahrzeuge in der Tabelle *Modelldaten* zurück:

```
SELECT AVG(Geschwindigkeit) AS Durchschnitt FROM Modelldaten
```

Diese Abfrage gibt lediglich einen Datensatz mit einem Feld zurück, das die Durchschnittsgeschwindigkeit enthält. Im »richtigen« Leben möchte man jedoch genauere Aussagen. Zum Beispiel die Durchschnittsgeschwindigkeit bezogen auf die verschiedenen Modelle. Dies nennt man eine Gruppierung, da verschiedene Datensätze, die alle eine Gemeinsamkeit aufweisen, in diesem Fall die gleiche Modellnummer, mit einer Aggregatfunktion zusammengefaßt werden. Für das Zusammenfassen ist das Schlüsselwort *GROUP BY* zuständig.

Das folgende SQL-Kommando gibt eine Datensatzgruppe zurück, in der jeder Datensatz die Durchschnittsgeschwindigkeit einer der vorhandenen »Zylinderklassen« enthält:

```
SELECT AVG(Geschwindigkeit) AS Durchschnitt FROM Modelldaten
GROUP BY Zylinder
```

7.6.1 Die HAVING-Klausel

Was für das einfache *SELECT*-Kommando die *WHERE*-Klausel ist, für das *GROUP BY*-Schlüsselwort die *HAVING*-Klausel. Sie bietet eine Möglichkeit, die Gruppe der Datensätze einzugrenzen.

Die folgende Abfrage gibt nur Datensätze mit einer Zylinderzahl größer oder gleich 6 zurück.

```
SELECT AVG(Geschwindigkeit) FROM Modelldaten GROUP BY
Zylinder HAVING Zylinder>=6
```

7.6.2 Gruppierung und JOIN-Operation

Das Gruppieren von Datensätzen läßt sich mit einer JOIN-Operation (mehr dazu in Kapitel 7.11) kombinieren.

Die folgende Abfrage gruppiert die Durchschnittspreise der in der Tabelle *Fahrzeugdaten* vorhandenen Fahrzeuge nach den Modellnamen. Dazu ist eine tabellenübergreifende Verknüpfung (JOIN-Operation) erforderlich, da sich der Modellname in der Tabelle *Modelldaten*, der Preis in der Tabelle *Fahrzeugdaten* befindet. Beide Tabellen besitzen *ModellNr* als gemeinsames Feld.

```
SELECT ModellName, AVG(Preis) FROM Modelldaten INNER JOIN
Fahrzeugdaten ON Modelldaten.ModellNr =
Fahrzeugdaten.ModellNr GROUP BY Modelldaten.ModellName
```

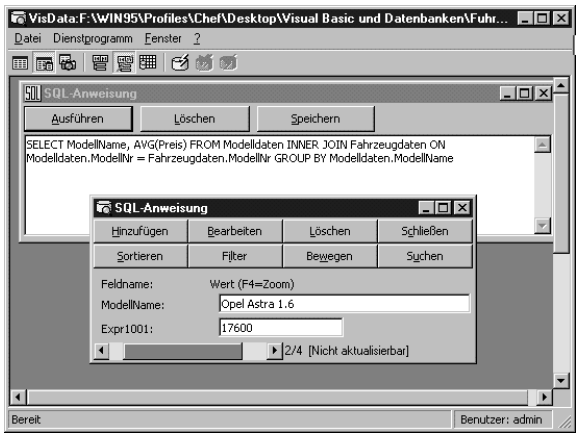

Bild 7.4: Durch GROUP BY und HAVING wird die Durchschnittsgeschwindigkeit der Datensätze nach Modellnamen sortiert

7.7 Das UPDATE-Kommando

Die Aufgabe des *UPDATE*-Kommando ist es, den Inhalt der Datenbank zu aktualisieren. Anstatt alle Datensätze einzulesen, ihren Inhalt zu ändern und sie danach wieder in die Datenbank zu übertragen, führt man ein einziges *UPDATE*-Kommando aus. Da Sie sich nun bereits ein wenig heimisch mit

SQL fühlen sollten, soll Ihnen die offizielle Syntax des Kommandos nicht vorenthalten werden:


```
UPDATE Tabellenname SET Spaltenname1 = Wert1 [, Spaltenname2 =
Wert2] WHERE Suchbedingung
```

Das Kommando untersucht zuerst die *WHERE*-Klausel nach einer Suchbedingung. Diese legt fest, bei welchen Datensätzen die angegebenen Spalten mit dem korrespondierenden Wert aktualisiert werden. Das ist zugegebenermaßen eine etwas umständliche Erklärung (das ist leider bei allgemeinen Beschreibungen so). Sehr viel aussagekräftiger ist das nächste Beispiel, das in der Tabelle *Fahrzeugdaten* die Spalte *Preis* bei allen Fahrzeugen um 10% erhöht, deren Anschaffungsjahr nach 1990 ist.


```
UPDATE Fahrzeugdaten SET Preis = Preis + Preis / 10 WHERE
Anschaffungsjahr > 1990
```

Eine »Undo-Funktion« gibt es hier übrigens nicht, d.h., alle Änderungen werden in der Datenbank durchgeführt. Läßt man die *WHERE*-Klausel weg, bezieht sich die Aktualisierung auf alle Datensätze in der angegebenen Tabelle:

```
UPDATE Fahrzeugdaten SET Preis = Preis + Preis / 10
```

Dieses Kommando bedeutet eine generelle Preiserhöhung.

Das *UPDATE*-Kommando gehört zu jenen SQL-Kommandos, die keine Datensätze zurückgeben. Aus diesem Grund öffnet sich nach der Ausführung des Kommandos im Visual Data Manager auch kein Datensatzfenster. Ein dickes Minus ist, daß der Visual Data Manager nicht anzeigt, wie viele Datensätze von der Aktualisierung betroffen wurden.

7.8 Das DELETE-Kommando

Wir nähern uns einer kritischen Zone. Während das *SELECT*-Kommando völlig harmlos ist, da es lediglich Daten aus der Datenbank abfragt, ist das *UPDATE*-Kommando dazu da, den Inhalt der Datenbank zu verändern. Das *DELETE*-Kommando geht einen entscheidenden Schritt weiter. Mit seiner Hilfe werden Datensätze (unwiederbringlich) gelöscht. Sie sollten es also nur mit Bedacht einsetzen und es zuvor an »harmlosen« Daten ausführlich testen.

Sollten Sie die Beispieldatenbank *Fuhrpark.mdb* inzwischen angelegt haben, fertigen Sie mit dem Windows-Explorer ein oder zwei Kopien an (nennen Sie diese *Fuhrpark1.mdb* und *Fuhrpark2.mdb*). Sollte beim Experimentieren mit dem *DELETE*-Kommando etwas schief gehen, greifen Sie auf die »Sicherungskopien« zurück. Ansonsten müssen Sie die Datensätze noch einmal eintippen.

```
DELETE FROM Tabellenname WHERE Bedingung
```

Die Syntaxbeschreibung macht es bereits deutlich, das *DELETE*-Kommando ist einfach in seiner Anwendung (aber unter Umständen »gefährlich« in seiner Ausführung, setzen Sie es daher bitte mit Bedacht ein).

Zum Kennenlernen des *DELETE*-Kommandos empfiehlt es sich, dieses zu Testzwecken durch das *SELECT*-Kommando zu ersetzen. Sie erhalten dadurch alle Datensätze zurück, die das *DELETE*-Kommando gelöscht hätte.

Das folgende SQL-Kommando löscht aus der (fiktiven) Stammdatentabelle *Kunden* alle Einträge, die seit dem 1.1.1879 keine Aufträge mehr getätigt haben.

```
DELETE FROM Kunden WHERE DatumLetzterAuftrag < #01/01/1879#
```

Das folgende SQL-Kommando löscht in der Tabelle *Fahrzeugdaten* alle Datensätze, in denen es im Feld *ModellNr* keinen Wert gibt (diese Felder besitzen den Spezialwert *NULL* – mehr dazu in Kapitel 2.3).

```
DELETE FROM Fahrzeugdaten WHERE ModellNr = NULL
```

Beim *DELETE*-Kommando gilt es folgende Besonderheiten zu beachten:

- ✘ Das *DELETE*-Kommando löscht nur komplette Datensätze aus einer Tabelle. Einzelne Felder lassen sich nicht löschen (verwenden Sie dafür *UPDATE*).
- ✘ Das *DELETE*-Kommando löscht nur Datensätze und nicht die Tabelle selbst. Verwenden Sie für das Löschen einer ganzen Tabelle das SQL-Kommando *DROP TABLE*.
- ✘ Wie bei *INSERT* und *UPDATE* kann das Löschen von Datensätzen über *DELETE* bei aktivierter referentieller Integrität weitere Löschaktionen in anderen Tabellen nach sich ziehen. Das bedeutet konkret, daß

das Löschen eines Datensatzes mit einem Primärschlüssel bei der Jet-Engine zum Löschen sämtlicher Daten mit einem Fremdschlüssel in anderen Tabellen führen kann, wenn dies zuvor in der Datenbank so festgelegt wurde. Halten Sie sich diesen möglichen Problembereich immer vor Augen, wenn Sie Daten in einer Datenbank modifizieren (die in diesem Kapitel vorgestellten Beispiele berücksichtigen diesen Aspekt nicht).

✘ Beim *DELETE*-Kommando gibt es weder eine vorherige Bestätigung noch eine Undo-Funktion. Das Kommando *DELETE * FROM* Kunden löscht alle Datensätze in dieser Tabelle – unwiderruflich. In der Praxis wird man von Datenbanken Sicherungskopien anlegen, und im Rahmen einer Transaktion wird eine *DELETE*-Operation durchgeführt, so daß durch Ablehnen der Transaktionsbestätigung die gesamte Aktion nicht in der Datenbank durchgeführt wird.

7.9 Das INSERT-Kommando

Das *INSERT*-Kommando fügt neue Datensätze in eine Tabelle ein. Es ist eines jener SQL-Kommandos, die im Zusammenhang mit VBA verzichtbar wären, da hier das Einfügen von Datensätzen über die *AddNew*-Methode eines ADO-*Recordset*-Objekts erledigt wird. Dennoch soll auch das *INSERT INTO*-Kommando kurz vorgestellt werden.


```
INSERT INTO Tabellenname (Spalte1, Spalte2...) VALUES (Wert1,
Wert2...)
```


```
INSERT INTO Fahrzeugdaten (FahrzeugNr, Farbe) VALUES (9999,
"Lila")
```

Dieses Kommando fügt in die Tabelle *Fahrzeugdaten* einen neuen Datensatz ein, in dem aber nur die Felder *FahrzeugNr* und *Farbe* mit Werten vorbelegt werden. Die übrigen Felder erhalten NULL-Werte.


```
INSERT INTO Fahrzeugdaten (FahrzeugNr, ModellNr, Farbe)
SELECT FahrzeugNr, ModellNr, Farbe FROM Fahrzeugdaten WHERE
ModellNr = 3
```

Dieses Kommando fügt in die Tabelle *Fahrzeugdaten* die Felder *FahrzeugNr*, *ModellNr* und *Farbe* in neue Datensätze ein, wobei die Daten für diese Datensätze aus jenen Datensätzen stammen, bei denen die Bedingung »ModellNr=3« zutrifft.

7.10 Aggregatfunktionen

SQL-Abfragen liefern als Ergebnis stets einen oder mehrere Datensätze zurück. Diese Gruppe von Datensätzen, die als Resultat einer Abfrage an ein Programm übergeben werden, heißt Datensatzgruppe, Resultset, Rowset oder Recordset. Bislang wurden die zurückgegebenen Datensätze über ein Kriterium ausgewählt, das mit einem oder mehreren Feldinhalten verglichen wurde. SQL kennt aber noch eine weitere Variante. Hier besteht das Auswahlkriterium aus einer allgemeinen Funktion, wie Maximalwert, Durchschnitt oder die größten zehn Werte. Die Funktionen, die für diese Abfrage eingesetzt werden, werden als Aggregatfunktionen bezeichnet, da sie den Inhalt einer Tabelle (bzw. allgemein einer Datensatzgruppe) zusammenfassen. Eine Übersicht über die wichtigsten SQL-Aggregatfunktionen gibt Tabelle 7.5.

Es versteht sich von selbst, daß Funktionen, wie *SUM* oder *AVG,* nur auf Spalten mit numerischen Werten angewendet werden können. Enthält ein Feld in der Spalte einen nicht numerischen Wert, resultiert daraus eine Fehlermeldung. Bei den Funktionen *MIN* und *MAX* ist dagegen auch ein Stringvergleich möglich.

Beachten Sie, daß auch die Aggregatfunktionen einen Datensatz zurückgeben. Er besteht in der Regel nur aus einem Feld, das den Default-Namen »Expr1001« besitzt (sofern kein Alias vergeben wurde – siehe Abschnitt 7.5.11).

Tabelle 7.5: SQL-Aggregatfunktionen

Aggregatfunktion	Bedeutung	Beispiel
AVG	Berechnet das arithmetische Mittel aller Feldwerte einer Datensatzgruppe.	*SELECT AVG (Preis) FROM Fahrzeugdaten*
COUNT	Zählt die Datensätze, z.B. jene, die eine bestimmte Bedingung erfüllen.	*SELECT COUNT (*) FROM Modelldaten WHERE Geschwindigkeit > 130*
MAX	Ermittelt den größten Feldwert in einer Datensatzgruppe.	*SELECT MAX(Preis) FROM Fahrzeugdaten*
MIN	Ermittelt den kleinsten Feldwert in einer Datensatzgruppe.	*SELECT MIN(Preis) FROM Fahrzeugdaten*

Aggregatfunktion	Bedeutung	Beispiel
StDev	Statistische Funktion (mehr dazu in der MSDN-Hilfe).	–
StDevP	Statistische Funktion (mehr dazu in der MSDN-Hilfe).	–
SUM	Bildet die Summe über eine Reihe von Feldern.	*SELECT SUM (Preis) FROM Fahrzeugdaten*

Tabelle 7.5: SQL-Aggregatfunktionen (Fortsetzung)

7.10.1 Aggregatfunktionen und Unterabfragen

Bei Funktionen wie *MIN* oder *MAX* möchte man in der Regel nicht den Wert, sondern den dazugehörigen Namen oder andere Feldnamen sehen. Wie müßte die SQL-Abfrage lauten, die den Namen des schnellsten Fahrzeugs in der Tabelle *Modelldaten* ermittelt? Vielleicht so:

```
SELECT NAME FROM Modelldaten WHERE Geschwindigkeit =
MAX(Geschwindigkeit)?
```

Leider nicht, denn Aggregatfunktionen sind in der *WHERE*-Klausel (bei AccessSQL) nicht erlaubt.

Hier muß eine sogenannte Unterabfrage zum Einsatz kommen, die bereits in die Kategorie fortgeschrittenes SQL fällt. Sehen Sie selbst:

```
SELECT ModellName FROM Modelldaten WHERE Geschwindigkeit =
(SELECT MAX(Geschwindigkeit) FROM Modelldaten)
```

Unterabfragen lassen sich mehrfach verschachteln. Es versteht sich von selbst, daß sich SQL-Experten auf diese Weise sehr mächtige Abfragen zusammenstellen können.

Falls Sie beim Austesten von SQL-Kommandos im Visual Data Manager permanent nach einem Parameternamen gefragt werden, kann dies an einem falsch geschriebenen oder nicht vorhandenen Feldnamen liegen.

7.11 Verknüpfungen mehrerer Tabellen

Bislang wurden alle Datenbankzugriffe mit einer einzigen Tabelle durchgeführt. Die eigentliche Stärke von SQL liegt jedoch in dem Umstand, theoretisch beliebig viele Tabellen in einer Abfrage verknüpfen zu können. Damit lassen sich enorm leistungsfähige Operationen durchführen. Diese Operationen werden als *Join-Operationen* bezeichnet, da Felder aus mehreren Tabellen zu einer neuen, virtuellen Tabelle »vereinigt« wurden.

Die folgenden Übungen basieren auf der Datenbank *Fuhrpark.mdb*, wobei aber diesmal zwei Tabellen im Spiel sind: die Tabelle *Modelldaten* mit allen Fahrzeugmodelldaten und die Tabelle *Fahrzeugdaten* mit den Daten des Fuhrparkbestands. Damit Sie die Beispiele direkt nachvollziehen können, sollten Sie sich noch einmal den Aufbau und den Inhalt der beiden Tabellen in Erinnerung rufen:

- ✘ Die Tabelle *Fahrzeugdaten* enthält die Inventarliste des gesamten Fahrzeugparks. Das Feld *FahrzeugNr* gibt die Nummer des Fahrzeugs an. Das Feld *ModellNr* verweist auf ein Fahrzeug in der Tabelle *Modelldaten*.
- ✘ Die Tabelle *Modelldaten* enthält die Modelldaten der einzelnen Fahrzeuge. Das Feld *ModellNr* ist eine Zahl, durch die ein Modelltyp eindeutig identifiziert wird (es ist der Primärschlüssel der Tabelle). Dieses Feld ist von zentraler Bedeutung, denn über die Modellnummer läßt sich eine Verbindung zwischen einem Fahrzeug in der Tabelle *Fahrzeugdaten* und seinen Daten in der Tabelle *Modelldaten* herstellen.

Das folgende SQL-Kommando holt alle Modellnamen aus der Tabelle *Modelldaten* und ordnet jedem Namen über das gemeinsame Feld *ModellNr* aus der Tabelle *Fahrzeugdaten* den Preis zu.

```
SELECT ModellName, Preis FROM Modelldaten INNER JOIN
Fahrzeugdaten ON Modelldaten.ModellNr =
Fahrzeugdaten.ModellNr
```

Die Verwendung des Schlüsselwortes *INNER JOIN* stellt die formelle Variante des *JOIN*-Kommandos dar. Es geht bei Jet-SQL auch ein wenig »formloser«, wie das folgende Beispiel zeigt:

```
SELECT ModellName, Preis FROM Modelldaten, Fahrzeugdaten
WHERE Modelldaten.ModellNr = Fahrzeugdaten.ModellNr
```

7.12 Zusammenfassung

SQL ist eine universelle und vor allem von einer bestimmten Datenbank weitestgehend unabhängige Datenbanksprache, die sich über die ADO-Objekte wunderbar mit VBA ergänzt. Wird ein *Recordset*-Objekt geöffnet, kann anstelle eines Tabellennamens über die *Source*-Eigenschaft oder direkt beim Aufruf der *Open*-Methode anstelle eines Tabellennamens auch ein SQL-Kommando übergeben werden. Dieses sorgt z.B. dafür, daß nur jene Datensätze zurückgegeben werden, die bestimmte Kriterien erfüllen. Handelt es sich um eine Aktionsabfrage, die Veränderungen an der Datenbank durchführt, aber keine Datensätze zurückgibt, wird diese über die *Execute*-Methode des *Connection*-Objekts ausführt.

7.13 Wie geht es weiter?

Mit SQL und den ADO-Objekten aus Kapitel 5 haben Sie die beiden Fundamente der Datenbankprogrammierung von Visual Basic und VBA kennengelernt. In den folgenden Kapiteln lernen Sie keine weiteren Grundlagen kennen, sondern praktische Beispiele der Datenbankprogrammierung mit ADO (und natürlich SQL). In Kapitel 8 werden die verschiedenen Einsatzmöglichkeiten eines *Recordset*-Objekts vorgestellt. In Kapitel 9 geht es um Formulare, mit denen sich wichtige Datenbankoperationen, wie das Erfassen neuer Datensätze, umsetzen lassen.

7.14 Fragen

Frage 1:

Wo liegt die wichtigste Stärke von SQL?

Frage 2:

Wie wird ein SQL-Kommando in einem VBA-Programm ausgeführt?

Frage 3:

Wie muß ein SQL-Kommando lauten, das aus der Tabelle *Fahrzeugdaten* nur jene Fahrzeuge zurückgibt, die vor höchstens einem Jahr angeschafft wurden?

Frage 4:

Wie muß ein SQL-Kommando lauten, das das Durchschnittsalter der Fahrzeuge in der Tabelle *Fahrzeugdaten* zurückgibt?

Frage 5:

Im Rahmen einer neuen Abschreibungsregel soll der Preis aller Fahrzeuge in der Tabelle *Fahrzeugdaten* um 10% reduziert werden. Wie muß das SQL-Kommando lauten?

Frage 6:

Fügen Sie für einen Geschwindigkeitsvergleich über ein kleines VBA-Programm und mit dem *INSERT INTO*-Kommando 10000 Datensätze mit beliebigen Daten in die Tabelle *Fahrzeugdaten* ein, wobei das Feld *FahrzeugNr* den Wert 9999 erhalten soll, um diese Datensätze später wieder löschen zu können. Erhöhen Sie den Wert des Feldes *Preis* mit dem *UPDATE*-Kommando um 10%, und messen Sie die Zeit mit der *Timer*-Funktion. Führen Sie die gleiche Aktion in einer *For Next*-Schleife aus, die über das *Recordset*-Objekt auf die Datensätze zugreift. Messen Sie hier ebenfalls die Ausführungszeit. Welche Variante ist schneller?

Die Antworten zu den Fragen finden Sie in Anhang D.

Kapitel 8

Der Umgang mit Datensatzgruppen

Datensatzgruppen, in der ADO-Terminologie, *Recordsets* bzw. *Rowsets* genannt, sind der Dreh- und Angelpunkt beim Datenbankzugriff. Alle Daten, die Sie von einer Datenquelle möchten, werden in Gestalt von Recordsets geliefert. Auch wenn dieses Kapitel nicht explizit dem *Recordset*-Objekt gewidmet ist, drehen sich ca. 90% der Beispiele um dieses ADO-Objekt. Es ist erstaunlich vielseitig und bietet eine Reihe von »Raffinessen«, wie z.B. die Fähigkeit, sich selbst lokal, d.h. unabhängig von einer Datenbank, abspeichern zu können. Einige der wirklich leistungsfähigen Eigenschaften, wie z.B. die Möglichkeit, einen kompletten Recordset über eine Internet-/Intranet-Verbindung an einen entfernen PC zu übermitteln, so daß dieser dort bearbeitet und später in die Datenbank »rückgeführt« werden kann (Stichwort: Batch-Updates), werden in diesem Buch allerdings nicht behandelt, da dies zu den fortgeschritteneren Programmiertechniken zählt, für die auch eine bestimmte »Infrastruktur« in Gestalt eines Intranets vorhanden sein muß. Sie werden in der MSDN-Hilfe und in zahlreichen Fachartikeln ausführlich beschrieben.

Sie erfahren in diesem Kapitel etwas zu folgenden Themen:

- ✘ Das Öffnen und Schließen von Datensatzgruppen
- ✘ Die Rolle des Datensatzzeigers
- ✘ Das Bewegen in einer Datensatzgruppe
- ✘ Das Sortieren von Datensätzen in einer Datensatzgruppe
- ✘ Suchen und Finden von Datensätzen

- ✘ Das Hinzufügen und Löschen von Datensätzen
- ✘ Spezielle Operationen mit Datensatzgruppen

Dieses Kapitel versteht sich in erster Linie als Helfer für die Praxis. Die meisten Beispiele sind so gehalten, daß Sie sie im Prinzip 1:1 in Ihre Programme übernehmen können. Gewisse Überschneidungen zu den Beispielen aus Kapitel 5, in denen es mit dem Überblick über die ADO-Objekte natürlich auch um das *Recordset*-Objekt geht, lassen sich daher nicht ganz vermeiden.

8.1 Datensatzgruppen öffnen und schließen

Der Zugriff auf eine Datensatzgruppe erfolgt bei ADO (wie auch bei DAO) stets über ein *Recordset*-Objekt. Unmittelbar nach seiner Deklaration erhält es (natürlich) noch keine Datensätze. Das Auffüllen mit Datensätzen geschieht beim Öffnen des *Recordset*-Objekts. Dafür gibt es drei Alternativen:

- ✘ Über die *Open*-Methode eines *Recordset*-Objekts
- ✘ Über die *Execute*-Methode eines *Connection*-Objekts
- ✘ Über die *Execute*-Methode eines *Command*-Objekts

Welche Alternative wann verwendet wird, hängt vom Anwendungskontext ab. In der Regel wird man sich für die *Open*-Methode entscheiden, während die *Execute*-Methode Aktionsabfragen und der Ausführung von Abfragen mit Parametern und von Stored Procedures beim Microsoft SQL-Server und der neuen MSDE bei Microsoft Access 2000 vorbehalten ist.

8.1.1 Das Öffnen einer Verbindung

Bevor ein *Recordset*-Objekt geöffnet wird, muß (im allgemeinen) eine Verbindung existieren, d.h. ein *Connection*-Objekt geöffnet worden sein. Beim Öffnen einer Verbindung wird ein OLE DB-Provider und, sofern es sich um eine Access-Datenbank handelt, der Pfad der Datenbank angegeben. Die folgende Befehlsfolge öffnet ein *Connection*-Objekt für den Zugriff auf die Access-Datenbank *C:\Eigene Dateien\Fuhrpark.mdb*.

```
Private Cn As ADODB.Connection
Const DBPfad = "C:\Eigene Dateien\Fuhrpark.mdb"
```

```
Sub Verbindung_Öffnen()
    Set Cn = New ADODB.Connection
    With Cn
        .Provider = "Microsoft.Jet.OLEDB.3.51"
        .ConnectionString = "Data Source=" & DBPfad
        .Open
     End With
End Sub
```

8.1.2 Direktes Öffnen eines Recordset-Objekts

Das direkte Öffnen eines *Recordset*-Objekts geschieht üblicherweise in drei Schritten:

1. Deklaration einer Variablen vom Typ ADODB.Recordset
2. Instanzierung der Variablen über den Set-Befehl
3. Öffnen des Recordset-Objekts über seine Open-Methode

Die Aufteilung der Deklaration in zwei Teilschritte bringt den Vorteil, daß der Zeitpunkt der Instanzierung definiert ist und nicht davon abhängt, wann die Variable zum ersten Mal angesprochen wird.

```
Recordset.Open [Source, [ActiveConnection, _
[CursorType, [LockType, [Options]]]]]
```

Die *Open*-Methode erwartet eine Reihe von Angaben, wobei alle diese Angaben optional sind. Das bedeutet nicht, daß sie nicht benötigt werden, sondern daß sie alternativ zuvor über Eigenschaften angegeben werden können:

- ✘ Der Parameter *Source* gibt an, woher die Datensätze kommen sollen. Anstelle eines Tabellennamens kann hier auch ein *Command*-Objekt angegeben werden.
- ✘ Der Parameter *ActiveConnection* gibt an, über welche Verbindung die Datensätze kommen (d.h. aus welcher Datenbank). Hier kann auch eine Verbindungszeichenfolge (etwa ein DSN) aufgeführt werden.
- ✘ Die Parameter *CursorType* und *LockType* legen den Typ der Datensatzgruppe (Cursor) fest.

Für den *Source*-Parameter wird entweder ein Tabellenname, eine Abfrage, eine Stored Procedure (nicht bei der Jet-Engine), ein SQL-Kommando oder

ein *Command*-Objekt angegeben (Sie haben also eine große Auswahl). Das Kombinieren mehrerer Datenbanken in einer Datensatzgruppe ist bei ADO zur Zeit nicht möglich.

Öffnen einer Datensatzgruppe auf der Grundlage einer Tabelle

Die folgende Befehlsfolge öffnet eine Datensatzgruppe auf der Grundlage der Tabelle *Modelldaten*. Es wird ein geöffnetes *Connection*-Objekt in der Variablen *Cn* vorausgesetzt.

```
Sub Datensatzgruppe_Öffnen()
    Set Rs = New ADODB.Recordset
    With Rs
        .ActiveConnection = Cn
        .CursorType = adOpenKeyset
        .LockType = adLockOptimistic
        .Source = "Modelldaten"
        .Open
    End With
End Sub
```

Die Auswahl eines Cursortyps (*CursorType*-Eigenschaft) hängt genau wie die Festlegung des Sperrtyps (*LockType*-Eigenschaft) von der geplanten Anwendung ab. Soll die Datensatzgruppe sowohl vollständig »scrollfähig« als auch beschreibbar sein, empfiehlt sich die Einstellung:

```
CursorType = adOpenKeyset
LockType = adLockOptimistic
```

Diese Einstellung wird, sofern nichts dagegen spricht, für alle Beispiele in diesem Kapitel verwendet.

Die einzelnen Angaben für das *Recordset*-Objekt werden in der Regel nicht erst mit der *Open*-Methode übergeben, sondern über die dafür zuständigen Eigenschaften bereits vorher eingestellt.

Öffnen einer Datensatzgruppe auf der Grundlage einer SQL-Abfrage

Die folgende Befehlsfolge öffnet eine Datensatzgruppe auf der Grundlage einer SQL-Abfrage, die alle Datensätze der Tabelle *Modelldaten* zurückgibt, bei denen das Feld *Geschwindigkeit* einen Wert größer 130 besitzt. Es wird ein geöffnetes *Connection*-Objekt in der Variablen *Cn* vorausgesetzt.

```
Sub Datensatzgruppe_ÖffnenSQL()
    Set Rs = New ADODB.Recordset
    With Rs
        .ActiveConnection = Cn
        .CursorType = adOpenKeyset
        .LockType = adLockOptimistic
        .Source = "SELECT * FROM Modelldaten " & _
                  "WHERE Geschwindigkeit > 130"
        .Open
    End With
End Sub
```

Verbindungsloses Öffnen eines Recordset-Objekts

Die folgende Befehlsfolge öffnet eine Datensatzgruppe auf der Grundlage einer SQL-Abfrage, ohne daß zuvor eine Verbindung geöffnet wurde.

```
Sub Datensatzgruppe_VerbindungslosÖffnen()
    Set Rs = New ADODB.Recordset
    With Rs
     .Source = "Select * From Modelldaten"
     .ActiveConnection = "DSN=Fuhrpark"
     .CursorType = adOpenKeyset
     .LockType = adLockOptimistic
    .Open
    End With
    MsgBox Prompt:=Rs.RecordCount & " Datensätze"
End Sub
```

Das *Connection*-Objekt steht, auch wenn es nicht explizit geöffnet wurde, über die *ActiveConnection*-Eigenschaft des *Recordset*-Objekts zur Verfügung. Das abgesetzte SQL-Kommando wird als Zeichenkette übergeben.

Welcher Cursor für welchen Zweck

Die Wahl des richtigen Cursors (Vorwärts, Keyset, Statisch oder Dynamisch) kann einen deutlichen Einfluß auf die Performance bei umfangreicheren Datenbankabfragen haben. Da das Thema bereits in Kapitel 5.5 zur Sprache kam, enthält Tabelle 8.1 nur eine kurze Zusammenfassung.

Tabelle 8.1: Die »Wann nehme ich welchen Cursor«-Beratung für Unentschlossene

Cursortyp	Wann optimal?
Dynamisch	Bieten den meisten »Komfort« in Form von möglichen Operationen. Bei kleinen Datensatzgruppen gut, bei sehr großen vermutlich zu langsam.
Statisch	Gut geeignet, wenn die Wahrscheinlichkeit gering ist, daß zeitlich andere Anwender Datensätze aus der Datensatzgruppe entfernen oder hinzufügen, oder wenn dies keine Rolle spielt.
Vorwärts	Wenn die Daten nicht geändert, sondern in erster Linie »nachgeschaut« werden sollen.
Keyset	In der Regel eine gute Wahl. Fügen andere Anwender Datensätze hinzu oder entfernen sie sie, ist ein Requery erforderlich.

8.1.3 Feststellen, ob Datensätze vorhanden sind

Um festzustellen, ob eine Datensatzgruppe nach dem Öffnen überhaupt Datensätze enthält, müssen die Eigenschaften *BOF* und *EOF* gemeinsam abgefragt werden (eine Empty-Eigenschaft gibt es nicht). Sind beide Eigenschaften *True*, enthält die Datensatzgruppe keine Datensätze.

8.1.4 Die Anzahl der Datensätze feststellen

Die *RecordCount*-Eigenschaft gibt die Anzahl der Datensätze in einer Datensatzgruppe an. Allerdings enthält diese Eigenschaft nicht immer einen Wert, so daß sie nur unter bestimmten Bedingungen verwendet werden kann.

Liefert die *RecordCount*-Eigenschaft den Wert –1, wird dieses Merkmal vom OLE DB-Provider nicht unterstützt.

8.1.5 Feststellen, was ein Recordset-Objekt so alles kann

Die »Fähigkeiten« eines *Recordset*-Objekts hängen vom OLE DB-Provider und vom gewählten Cursortyp ab. So unterstützt ein Recordset keine *Delete*-Methode, wenn es sich um einen Vorwärts-Cursor handelt. Um Laufzeitfehler und andere Überraschungen zu vermeiden, empfiehlt es sich, über die *Supports*-Methode eines *Recordset*-Objekts bestimmte Fähigkeiten abzufragen:

```
If Rs.Supports(adDelete) = True Then
    Rs.Delete
End If
```

In diesem Beispiel wird ein Datensatz nur dann gelöscht, wenn dies vom OLE DB-Provider unterstützt wird[1].

8.2 Ausführen von SQL-Kommandos

Bei ADO gibt es drei Möglichkeiten, SQL-Kommandos auszuführen:

1. Beim Öffnen eines *Recordset*-Objekts.
2. Über die *Execute*-Methode eines *Command*-Objekts.
3. Über die *Execute*-Methode eines *Connection*-Objekts.

SQL-Abfragen beim Öffnen eines Recordset-Objekts

Diese Variante kommt nur dann in Frage, wenn das SQL-Kommando Datensätze zurückgibt, wie es bei *SELECT*-Kommandos der Fall ist:

```
With Rs
  .Open Source:= "Select * From Modelldaten", _
        ActiveConnection:=Cn
End With
```

Diesmal werden die Verbindung und die Quelle beim Aufruf von *Open* übergeben. Bitte beachten Sie, daß ein *SELECT * FROM* zwar beliebt bei Buchautoren, aber gefürchtet bei Datenbankprogrammierern ist, denn es bedeutet im ungünstigsten Fall, daß sämtliche Datensätze vom Datenbankserver (meistens über ein Netzwerk) an das Programm übermittelt werden. Mit allen Nachteilen für die Performance und die Netzwerkbelastung.

Ausführen von SQL-Kommandos, die keine Datensätze zurückgeben

In diese Kategorie fallen z.B. *DELETE*-, *INSERT*- und *UPDATE*-Kommandos. Für diese »Aktionsabfragen« ist die *Execute*-Methode zuständig, die es sowohl beim *Command*- als auch beim *Connection*-Objekt gibt. Letzteres bringt den Vorteil, daß kein *Recordset*-Objekt angelegt werden muß, das ohnehin nicht benötigt wird. Die *Execute*-Methode des *Command*-Objekts kommt immer dann zum Einsatz, wenn die Aktionsabfrage einen einzelnen Wert zurückgibt, wie es z.B. bei Aggregatfunktionen (etwa bei der *SUM*-Funktion) der Fall ist. Da SQL-Kommandos keinen »Rückgabewert« besit-

1 Der von der Support-Methode zurückgegebene True/False-Wert sagt nur etwas über die grundsätzlichen Fähigkeiten des OLE DB-Providers aus und gibt nicht an, ob die betreffende Operation im vorliegenden Programmkontext durchführbar ist.

zen, geschieht die Rückgabe in einem neutralen Feld einer Datensatzgruppe, die nur aus einem Datensatz besteht.

Die *Execute*-Methode des *Command*-Objekts liefert in jedem Fall einen Vorwärts-Cursor zurück, der nicht aktualisiert werden kann.

Die folgende Befehlsfolge führt ein *UPDATE*-Kommando über die *Execute*-Methode eines *Connection*-Objekts aus:

```
Sub ExecuteSQLConnection()
    Dim SQLString As String
    Dim AnzahlDatensätze As Long
    SQLString = "UPDATE Fahrzeugdaten SET Kennzeichen " & _
                "= 'D-OOO' WHERE Kennzeichen=NULL"
    With Cn
     Set Rs = .Execute(CommandText:=SQLString, _
              RecordsAffected:=AnzahlDatensätze)
    End With
    MsgBox "Betroffene Datensätze: " & AnzahlDatensätze
End Sub
```

Auch wenn die *Execute*-Methode ein *Recordset*-Objekt zurückgeben kann, wird diese Möglichkeit nicht genutzt, da es bei einem *UPDATE*-Kommando keine Datensatzgruppe gibt und ein *Recordset*-Objekt daher nicht existiert.

Ausführen von SQL-Kommandos über ein Command-Objekt

Ein *Command*-Objekt kann sowohl SQL-Kommandos, die Datensatzgruppen zurückgeben, als auch Aktionsabfragen ausführen. Seine Hauptaufgabe besteht jedoch in der Ausführung von Abfragen (also in der Datenbank gespeicherten SQL-Kommandos oder Stored Procedures (SQL-Server oder MSDE), denen Parameter übergeben werden sollen.

Die folgende Befehlsfolge verdoppelt den Wert im Feld *Preis* in der Tabelle *Fahrzeugdaten*.

```
Sub UpdateSQL()
    Dim SQLText As String
    Dim AnzahlDatensätze As Long
    SQLText = "UPDATE Fahrzeugdaten SET Preis = Preis * 2"
    Set Cmd = New ADODB.Command
    With Cmd
    .ActiveConnection = Cn
    .CommandType = adCmdText
```

```
    .CommandText = SQLText
    .Execute RecordsAffected:=AnzahlDatensätze
    End With
    MsgBox Prompt:=AnzahlDatensätze & " Datensätze"
End Sub
```

Über die Variable *AnzahlDatensätze* erfährt man, wie viele Datensätze von dem *UPDATE*-Kommando betroffen waren.

8.3 Schließen von Datensatzgruppen

Das Schließen einer Datensatzgruppe geschieht kurz und schmerzlos über die *Close*-Methode. In größeren Programmen (wo man schon einmal die Übersicht verlieren kann, wann welches *Recordset* geschlossen wurde) empfiehlt es sich, vor dem Schließen abzufragen, ob das *Recordset*-Objekt bereits geschlossen ist:

```
If Rs.State = adStateOpen Then
    Rs.Close
End If
```

Wird ein *Connection*-Objekt geschlossen, werden damit auch alle *Recordset*-Objekte geschlossen, die mit dem *Connection*-Objekt verbunden sind.

8.4 Bewegen in der Datensatzgruppe

Ist die Datensatzgruppe erst einmal da, muß mit ihr gearbeitet werden. Eine Form des Arbeitens besteht darin, einen bestimmten Datensatz »anzusteuern«, um ihn zum aktuellen Datensatz zu machen. Dazu muß man wissen, daß der aktuelle Datensatz durch die aktuelle Position des Datensatzzeigers bestimmt wird, die sich wiederum über die *Move*-Methode verändern läßt.

8.4.1 Move around!

Der Datensatzzeiger wird über verschiedene Methoden verschoben (siehe Tabelle 8.2). Welche Methoden erlaubt sind, ist auch eine Frage des Cursortyps, denn ein Vorwärts-Cursor erlaubt z.B. keine Rückwärtsbewegung.

Die folgenden Befehlsfolge »besucht« jeden einzelnen Datensatz in einer Datensatzgruppe und ändert dabei den Wert eines bestimmten Feldes. Das Ende der Datensatzgruppe wird über die *EOF*-Eigenschaft festgestellt.

```
Sub AlleBesuchenUndÄndern()
    Dim dMerker As Double
    dMerker = Rs.Bookmark
    With Rs
     Do Until .EOF = True
     .Fields("Baujahr").Value = .Fields("Baujahr").Value + 1
     .Update
     .MoveNext
     Loop
    .Bookmark = dMerker
    End With
End Sub
```

Erst die *Update*-Methode sorgt dafür, daß die Änderung in die Datenbank übernommen wird.

Können Sie sich vorstellen, daß die gesamte Schleife durch eine einzige Anweisung ersetzt werden kann? Nun, das SQL-Kommando *UPDATE* macht es möglich. Den Beweis finden Sie (indirekt) in Kapitel 8.2. Bei großen Datensatzgruppen kommt eine Schleife mit *MoveNext*-Methode daher nicht in Frage.

Tabelle 8.2: Methoden des Recordset-Objekts zum Navigieren in einer Datensatzgruppe

Methode	Bedeutung
MoveNext	Bewegt den Datensatzzeiger auf den nächsten Datensatz in der Datensatzgruppe.
MovePrevious	Bewegt den Datensatzzeiger auf den vorhergehenden Datensatz in der Datensatzgruppe.
MoveFirst	Bewegt den Datensatzzeiger auf den ersten Datensatz in der Datensatzgruppe.
MoveLast	Bewegt den Datensatzzeiger auf den letzten Datensatz in der Datensatzgruppe.
Move	Bewegt den Datensatzzeiger um eine bestimmte Anzahl an Positionen vor oder zurück.

8.4.2 Das direkte Ansteuern eines Datensatzes

Über die *Move*-Methode oder die *AbsolutePosition*-Eigenschaft ist ein direktes Ansteuern eines Datensatzes möglich. Die folgende Befehlsfolge zeigt, wie sich dies bewerkstelligen läßt.

```
Sub GeheZu()
    Dim DatensatzNr As Long
    Set Rs = New ADODB.Recordset

    With Rs
     .Source = "Modelldaten"
     .ActiveConnection = Cn
     .CursorType = adOpenStatic
     .LockType = adLockOptimistic
     .Open
     DatensatzNr = InputBox(Prompt:="Datensatznummer:")
     .MoveFirst
     .Move NumRecords:=DatensatzNr - 1
    End With

    MsgBox Prompt:="Das Modell heißt: " &
Rs.Fields("Modellname").Value
' Alternativ über AbsolutePosition-Eigenschaft

    With Rs
     .MoveFirst
     .AbsolutePosition = DatensatzNr
     MsgBox Prompt:="Das Modell heißt: " &
.Fields("Modellname").Value
    End With

End Sub
```

Möchte man nach dem Ansteuern des Datensatzes die alte Position wiederherstellen, muß die *Bookmark*-Eigenschaft zuvor in einer Variablen gesichert und anschließend der Wert der Eigenschaft wieder zugewiesen werden. Zwischen *Move* und *AbsolutePosition* gibt es einen kleinen Unterschied. Der Wert, der der *Move*-Methode übergeben wird, muß um eins kleiner sein als jener, der bei *AbsolutePosition* angegeben wird, da es bei *Move* heißt »um n Datensätze nach vorne/zurück bewegen«, bei *AbsolutePosition* dagegen »Gehe zu Datensatz n«.

8.4.3 Ist das Ende erreicht?

Ein wichtiger Aspekt beim Bewegen in einer Datensatzgruppe ist die Frage, ob das Ende der Datensatzgruppe erreicht wurde. Wie die berühmte Wurst hat auch die Datensatzgruppe zwei »Enden«: einen Anfang und ein Ende.

Befindet sich der Datensatzzeiger am Anfang der Datensatzgruppe, wird dies durch die *BOF*-Eigenschaft angezeigt, die in diesem Fall den Wert *True* besitzt. Wird der Datensatzzeiger über den letzten Datensatz hinausbewegt, erhält die *EOF*-Eigenschaft den Wert *True*. Es ist also wichtig zu verstehen, daß, wenn der Datensatzzeiger lediglich auf den ersten oder letzten Datensatz zeigt, noch nichts passiert. Erst wenn der Datensatzzeiger sozusagen den Schritt über den »Abgrund« macht, gehen die Warnlampen *BOF* oder *EOF* an. Übrigens können auch beide Warnlampen gleichzeitig angehen: In diesem Fall besitzt die Datensatzgruppe keinen Datensatz. Die Abfrage:

```
If Rs.BOF = True And Rs.EOF = True Then
```

prüft daher, ob die Datensatzgruppe überhaupt einen Inhalt besitzt, was z.B. immer dann nicht der Fall sein kann, wenn eine SQL-Abfrage zu keinem Ergebnis geführt hat (was z.B. zutrifft, wenn Sie alle Datensätze in der Tabelle *Modelldaten* sehen möchten, bei denen das Feld *Geschwindigkeit* einen Wert größer 400 oder einen negativen Wert besitzt – da es keinen Datensatz gibt, ist die Ergebnismenge leer[1]). Auch wenn es denkbar wäre: Einen »Umlauf«, der bewirken würde, daß der Datensatzzeiger von der letzten Position bei einem *MoveNext* wieder auf die erste Position wandert, gibt es nicht.

Um zu vermeiden, daß ein *MovePrevious* oder *MoveNext* zu einem Laufzeitfehler führt, wird eine entsprechende Abfrage nachgeschaltet:

```
Rs.MovePrevious
If Rs.BOF = True Then
    Rs.MoveFirst
End If
```

(Lese-)Zeichen setzen mit der Bookmark-Eigenschaft

Möchte man sich eine bestimmte Position des Datensatzzeigers merken, muß die *Bookmark*-Eigenschaft in einer Variablen (vom Typ *Double*) gespeichert werden. Soll dieser Datensatz später wieder zum aktuellen Datensatz werden, muß die Variable der *Bookmark*-Eigenschaft wieder zugewiesen werden. Diese Methode ist zuverlässiger, als die Datensatzzeigerposition zu »berechnen«.

---

1 In der Grundschule nannte man das früher eine leere Menge. Aber nur dann, wenn Mengenlehre auf dem Unterrichtsplan stand. Doch ohne Ironie: Zwischen der Mengenlehre und der relationalen Algebra gibt es naturgemäß Parallelen.

Die aktuelle Position des Datensatzzeigers steht über die Bookmark*-Eigenschaft zur Verfügung.*

8.4.4 Die Position des Datensatzzeigers abfragen

Die aktuelle Position des Datensatzzeigers steht auch über die *AbsolutePosition*-Eigenschaft zur Verfügung, welche die Nummer des aktuellen Datensatzes enthält. Ob *AbsolutePosition* einen Wert enthält, hängt allerdings von zwei äußeren Faktoren ab: vom OLE DB-Provider und vom Cursortyp. Gibt es keinen aktuellen Datensatz oder läßt sich dieser nicht ermitteln, erhält die Eigenschaft den Wert -1. Über die Abfrage

```
If Rs.Supports(adApproxPosition) = True Then
```

läßt sich feststellen, ob eine Datensatzgruppe Positionsangaben unterstützt.

8.5 Sortieren von Datensatzgruppen

Beim Öffnen einer Datensatzgruppe (aus einer Access-Datenbank) liegen die Datensätze in jener Reihenfolge vor, in der sie in die Tabelle eingefügt wurden. Über die *Sort*-Eigenschaft kann diese Reihenfolge geändert werden, indem der Eigenschaft der Name eines Feldes zugewiesen wird. Die folgende Befehlsfolge listet zunächst alle Feldnamen einer Tabelle in einem Listenfeld auf und verwendet in dessen *Click*-Ereignis den ausgewählten Namen als Sortierkriterium:

```
Dim Fi As ADODB.Field
For Each Fi In Rs.Fields
    lstFelder.AddItem Item:=Fi.Name
Next
```

Im nächsten Schritt wird das *Recordset*-Objekt, das für alle Datensätze der Tabelle *Modelldaten* steht, einem DataGrid zugewiesen:

```
Set dbgrdModelldaten.DataSource = Rs
```

Schließlich muß das Listenfeld den ausgewählten Namen der *Sort*-Eigenschaft zuweisen.

```
Private Sub lstFelder_Click()
    Rs.Sort = lstFelder.Text
End Sub
```

Da das DataGrid gebunden ist, wird die neue Reihenfolge automatisch dargestellt. Ein Refresh ist nicht erforderlich.

Die *Sort*-Eigenschaft kann beim Zugriff auf Access-Datenbanken nur verwendet werden, wenn über die *CursorLocation*-Eigenschaft ein clientseitiger Cursor (*CursorLocation=adUseClient*) eingestellt wurde.

8.6 Filtern von Datensatzgruppen

Das Filtern einer Datensatzgruppe bedeutet, daß Datensätze, auf die ein vorgegebenes Kriterium zutrifft, ausgeblendet werden. Über die *Filter*-Eigenschaft können eine oder mehrere jener Kriterien aufgeführt werden, die in einem SQL-*SELECT*-Kommando auf die *WHERE*-Klausel folgen würden. Die folgende Befehlsfolge filtert alle Datensätze aus, bei denen das Feld *Geschwindigkeit* einen Wert unterschreitet:

```
Sub FilterTest()
    MsgBox Prompt:="Anzahl Datensätze=" & Rs.RecordCount
    Rs.Filter = "Geschwindigkeit < 200"
    MsgBox Prompt:="Anzahl Datensätze=" & Rs.RecordCount
End Sub
```

Durch Setzen der *Filter*-Eigenschaft auf einen Leerstring wird der Filter wieder aufgehoben, und alle Datensätze sind wieder da.

8.7 »Suchen« und Finden von Datensätzen

Eine echte Suchoperation gibt es bei Datensatzgruppen nicht, denn schließlich ist eine Datensatzgruppe in der Regel das Ergebnis einer Suchoperation. Und sollte diese nicht den gewünschten Datensatz enthalten, muß die Suchoperation modifiziert und erneut ausgeführt werden. Dennoch bietet das *Recordset*-Objekt sowohl eine *Find*-Methode als auch eine Seek-Methode, die beide allerdings der Lokalisierung eines bereits vorhandenen Datensatzes dienen. Die *Seek*-Methode basiert auf den Indizes einer Tabelle und ist damit meistens schneller, kann dafür aber nur bei Recordsets vom Typ *adCmdTableDirect* angewendet werden. Konnte der Datensatz nicht lokalisiert werden, wird in beiden Fällen die *BOF*- bzw. *EOF*-Eigenschaft (je nach Suchrichtung) auf *True* gesetzt. Es ist daher in der Regel notwendig, die Position des Datensatzzeigers zuvor zu sichern.

8.7.1 Die Find-Methode in der Übersicht

Die Tabelle 8.3 zeigt die verschiedenen Varianten der *Find*-Methode und stellt sie ihren Pendants bei den Data Access Objects (DAOs) gegenüber. DAO-Programmierer werden feststellen, daß sich zwar ein paar Kleinigkeiten geändert haben (so gibt es keine *NotMatch*-Eigenschaft), aber keine Funktionalität fehlt.

Wird es nicht anders festgelegt, startet die *Find*-Methode bei der aktuellen Datensatzzeigerposition.

Aktion	DAO-Methode	ADO-Find mit SkipRows und SearchDirection
Ersten Datensatz finden	*FindFirst*	0 und *adSearchForward*
Nächsten Datensatz finden	*FindNext*	1 und *adSearchForward*
Zurückliegenden Datensatz finden	*FindPrevious*	1 und *adSearchBackward*
Letzten Datensatz finden	*FindLast*	0 und *adSearchBackward*

Tabelle 8.3: Die verschiedenen Find-Varianten im Überblick und im Vergleich zu DAO

Das erfolglose Ergebnis einer Suche wird sowohl bei *Find* als auch bei *Seek* durch Setzen der *BOF*- bzw. *EOF*-Eigenschaft auf *True* angezeigt. Der Datensatzzeiger verliert also seine ursprüngliche Position.

8.7.2 Die Find-Methode im Einsatz

Die folgende Befehlsfolge positioniert den Datensatzzeiger auf den ersten Datensatz, der einem Kriterium entspricht:

```
Sub FindTest()
    Dim AltePosition As Double
    AltePosition = Rs.Bookmark
    Rs.Find Criteria:="Geschwindigkeit > 200"
    If Rs.EOF = True Then
       Rs.Bookmark = AltePosition
       MsgBox Prompt:="Leider nichts gefunden!"
    Else
       MsgBox Prompt:="Das Modell heißt " & _
        Rs.Fields("Modellname").Value & _
```

```
            " mit einer Geschwindigkeit von " & _
            Rs.Fields("Geschwindigkeit").Value
        End If
End Sub
```

Auf die *Find*-Methode kann nur ein Kriterium folgen. Es ist nicht möglich, mehrere Kriterien zu kombinieren. Für diesen Zweck muß die *Filter*-Eigenschaft verwendet werden.

ADO und DAO unterscheiden sich in der Art und Weise, wie ein Vergleich mit *NULL* durchgeführt wird. Bei DAO wird der *Is*-Operator verwendet (gegebenenfalls zusammen mit Not), bei ADO sind es dagegen die Operationen = oder <>.

8.7.3 Die Seek-Methode im Einsatz

Als Alternative zur *Find*-Methode gibt es bei einem *Recordset*-Objekt die *Seek*-Methode, die auf einer ausgeklügelten indexbasierenden Suchmethode basiert und daher im allgemeinen sehr viel schneller ist[1]. Natürlich gibt es auch eine Einschränkung. Diese Art der Suche funktioniert nur mit Recordsets, die direkt auf einer Tabelle basieren. Datensatzgruppen, die z.B. auf einem SQL-Kommando oder einer Abfrage basieren, lassen sich nicht mit *Seek* durchsuchen. Und es gibt weitere Einschränkungen zu beachten:

- ✘ Bei *CursorLocation* darf nicht *adUseClient* eingestellt sein.
- ✘ Das ist die größte Einschränkung: Die Suche mit *Seek* funktioniert nur ab ADO 2.1 mit Access-2000-Datenbanken, d.h., die Datenbank muß im Format der Jet-Engine 4.0 vorliegen, und es muß der OLE DB-Provider für Jet 4.0 (»Microsoft.Jet.OLEDB.4.0«) gewählt sein, der durch Microsoft Access 2000 installiert wird. Bei Access-97-Datenbanken läßt sich die *Seek*-Methode (anscheinend) nicht anwenden, denn die *Supports*-Methode liefert für das Argument *adSeek* den Wert *Falsch,* und man erhält beim Aufruf von *Seek* die Meldung, daß dies vom OLE DB-Provider nicht unterstützt wird.

1 Offiziell wurde diese Methode erst mit ADO 2.1 eingeführt. Es kann daher sein, daß sie bei Ihnen noch nicht zur Verfügung steht.

Wird vor der Ausführung der *Seek*-Methode über die *Index*-Eigenschaft kein Index angegeben, wird der Primärschlüssel als Index verwendet.

Die folgende Befehlsfolge zeigt ein Beispiel für die *Seek*-Methode im Zusammenhang mit einer Access-2000-Datenbank. Als erstes muß das *Recordset*-Objekt geöffnet werden:

```
Set Rs = New ADODB.Recordset
With Rs
  .ActiveConnection = Cn
  .CursorType = adOpenKeyset
  .CursorLocation = adUseServer
  .LockType = adLockOptimistic
  .Source = "Modelldaten"
  .Open Options:=adCmdTableDirect
End With
```

Anschließend kann die *Seek*-Methode benutzt werden, um einen Datensatz zu lokalisieren:

```
Rs.Index = "PrimaryKey"
Rs.Seek keyValues:=txtSuchbegriff.Text, _
  SeekOption:=adSeekFirstEQ
```

Bei *PrimaryKey* handelt es sich um den Namen des Primärschlüssels. Die verschiedenen Konstanten für *SeekOption* werden in der MSDN-Hilfe beschrieben.

Sollen mehrere Werte als Suchbegriff übergeben werden, müssen diese (laut MSDN-Hilfe) dem Argument *KeyValues* über die *Array*-Funktion zugewiesen werden.

Das Anlegen von Indizes ist mit den ADODB-Objekten nicht möglich. Dazu werden die ADOX-Objekte benötigt, die u.a. von Microsoft Access 2000 zur Verfügung gestellt werden.

8.8 Hinzufügen von Datensätzen

Für das Hinzufügen eines Datensatzes ist die *AddNew*-Methode des *Recordset*-Objekts zuständig. Anschließend müssen die einzelnen Felder mit Werten belegt werden. Ansonsten erhalten diese einen *NULL*-Wert, was im allgemeinen vermieden werden sollte. Aktualisiert wird der neue Datensatz über die *Update*-Methode. Damit sichergestellt wird, daß bei einem Abbruch während des Zuweisens von Werten an die einzelnen Felder die Datenbank in einem definierten Zustand verbleibt, sollte das Hinzufügen von Datensätzen in eine Transaktion eingebunden werden.

Die folgende Befehlsfolge fügt der Tabelle *Modelldaten* einen Datensatz hinzu und bettet das Ganze in eine Transaktion ein:

```
Sub AddNewTest()
    Dim Antwort As Integer
    Cn.BeginTrans
    With Rs
     .AddNew
     .Fields("ModellNr").Value = "9999"
     .Fields("Modellname").Value = "Nur ein Testwagen"
     .Fields("Geschwindigkeit").Value = "0"
     .Update
    End With
    Antwort = MsgBox(Prompt:="Änderung durchführen?", _
     Buttons:=vbYesNo + vbExclamation)
    If Antwort = vbYes Then
        Cn.CommitTrans
    Else
        Cn.RollbackTrans
    End If
    Cn.Close
End Sub
```

8.9 Entfernen von Datensätzen

Das Entfernen eines Datensatzes ist eine sehr simple Operation. Der Aufruf der *Delete*-Methode genügt, um den aktuellen Datensatz zu löschen. Da das Entfernen ohne Bestätigung geschieht, sollte dies vom Programm eingefügt werden. Die folgende Befehlsfolge löscht den aktuellen Datensatz:

```
Rs.Delete
```

Über den optionalen Parameter *RecordsAffected* läßt sich durch Übergabe einer Variablen feststellen, wie viele Datensätze von der Löschoperation betroffen wurden.

8.10 Spezielle Operationen mit Datensatzgruppen

Neben den »Basisoperationen«, wie Öffnen, Schließen und Verschieben des Datensatzzeigers, erlauben *Recordset*-Objekte eine Reihe von »Spezialoperationen«, um die es in diesem Abschnitt geht.

8.10.1 Einlesen einer kompletten Datensatzgruppe

Über die *GetRows*-Methode ist es möglich, eine komplette (oder teilweise) Datensatzgruppe einer Feldvariablen zuzuweisen. Die folgende Befehlsfolge liest die komplette Tabelle *Modelldaten* in eine Feldvariable:

```
Sub GetRowsTest()
    Dim DatensatzFeld As Variant
    DatensatzFeld = Rs.GetRows(Rows:=100)
    MsgBox Prompt:="Anzahl Felder im Feld = " &
UBound(DatensatzFeld, 2) - LBound(DatensatzFeld, 2)
End Sub
```

Anschließend befinden sich alle Datensätze (maximal jedoch 100) in der zweidimensionalen Feldvariablen *DatensatzFeld:*

```
?Datenfeld (0, 1)
```

Diese Anweisung gibt das erste Feld des zweiten Datensatzes aus, denn die erste Dimension (die X-Koordinate, also die Spalten) adressiert die Felder, die zweite dagegen (Y-Koordinate) die Datensätze. Über die *UBound*-Methode erhält man u.a. die Anzahl der Felder:

```
?Ubound(Datensatzfeld,1)
```

bzw. die Anzahl der Datensätze:

```
?Ubound(Datensatzfeld,2)
```

8.10.2 Umwandeln einer Datensatzgruppe in eine Zeichenkette

Über die *GetString*-Methode kann der Inhalt einer Datensatzgruppe entweder vollständig oder teilweise in eine Zeichenkette umgewandelt werden:

```
AlleDatensätze = Rs.GetString
```

Das genaue Format des Strings wird über Parameter der *GetString*-Methode eingestellt.

8.11 Zusammenfassung

Das wichtigste ADO-Objekt ist das *Recordset*-Objekt, denn es ermöglicht den Zugriff auf die von einer Datenquelle gelesenen Daten. Jedes *Recordset*-Objekt steht für keinen, einen oder mehrere Datensätze. Je nach Typ des *Recordset*-Objekts können die eingelesenen Daten bearbeitet werden, wobei die Änderungen sofort (oder später) von der Datenquelle übernommen werden. Das *Recordset*-Objekt bietet eine reichhaltige Auswahl von Eigenschaften und Methoden, mit deren Hilfe sich praktisch »alles« erledigen läßt. Das reicht von elementaren Operationen, wie dem Bewegen des Datensatzzeigers auf einen anderen Datensatz, bis hin zum Abspeichern einer kompletten Datensatzgruppe in einem Feld. Allerdings sind nicht alle Operationen unter allen Umständen möglich. So läßt sich die schnelle *Seek*-Methode über OLE DB mit Access-Datenbanken nur dann ausführen, wenn die Datenbank im Access-2000-Format vorliegt. Es kann also nach wie vor Situationen geben, in denen ein Ausweichen auf die Data Access Objects (DAO – mehr dazu in Anhang C) eine Alternative sein kann.

Die Leichtigkeit im Umgang mit dem *Recordset*-Objekt überdeckt schnell eine potentielle »Performance«-Falle und damit eine Art »Designschwäche«. Es ist für weniger erfahrene Programmierer kein Problem, schlecht performante Datenbankzugriffe zu programmieren. Eine ungünstig formulierte Schleife oder ein unpassend ausgewählter Cursortyp fällt bei einer kleinen Access-Datenbank nicht auf. Wird diese Datenbank jedoch skaliert oder steigt die Nutzlast, treten auf einmal Engpässe auf, wo vorher keine vermutet wurden. Die Leichtigkeit bei der Programmierung erspart daher keine grundsätzlichen Überlegungen und macht langjährige Erfahrung in diesem Bereich nicht überflüssig.

8.12 Wie geht es weiter?

Wir nähern uns langsam, aber sicher dem Ende des Buches. Auch in den nächsten Kapiteln geht es ausschließlich um die Praxis. Während Kapitel 9 anhand häufig benötigter Dialogfelder die praktische Anwendung der in Kapitel 5 und in diesem Kapitel vorgestellten ADO-Objekte zeigt, geht es in Kapitel 10 bereits um die etwas fortgeschrittenere ADO-Programmierung.

8.13 Fragen

Frage 1:

Sie haben ein *Recordset*-Objekt geöffnet und es der Variablen *Rs* zugewiesen. Wie läßt sich später im Programm feststellen, auf welche Weise das *Recordset*-Objekt ursprünglich geöffnet wurde?

Frage 2:

Warum kann die folgende Befehlsfolge so nicht funktionieren:

```
Set Rs = New ADODB.Recordset
With Rs
  .CursorType = adOpenKeyset
  .LockType = adLockOptimistic
  .Source = "Modelldaten"
  .Open
End With
```

Frage 3:

Warum kommt es bei der folgenden Zuweisung an die *Sort*-Eigenschaft zur Fehlermeldung »Die angeforderte Operation wird vom Provider nicht unterstützt«?

```
Set Rs = New ADODB.Recordset
With Rs
  .ActiveConnection = Cn
  .CursorType = adOpenKeyset
  .LockType = adLockOptimistic
  .Source = "Getränke"
  .Open
  .Sort = "Name"
End With
```

Was hat der Programmierer übersehen?

Frage 4:

Warum ergibt der folgende Aufruf der *Find*-Methode kein Resultat (auch wenn die Tabelle *Modelldaten* ein Feld enthält), das das Suchkriterium erfüllt?

```
Rs.Find Criteria:= "Geschwindigkeit > 200", SkipRecords:=1
```

Frage 5:

Wie unterscheidet sich die *Find*-Methode eines *Recordset*-Objekts von der *Seek*-Methode und welche Einschränkungen gilt es bei *Seek* zu beachten? Wann ist die *Seek*-Methode zu empfehlen und wann nicht?

Frage 6:

Eine Programmiererin möchte den Inhalt der Tabelle *Modelldaten* mit minimalem Aufwand in ein Word-Dokument übernehmen. Können Sie ihr einen Vorschlag machen?

Die Antworten zu den Fragen finden Sie in Anhang D.

Kapitel 9

Datenbank-oberflächen

Die bisherigen Beispiele zu ADO, zum Datenumgebungs-Designer und zu SQL waren hoffentlich lehrreich und interessant, doch für die Praxis waren sie noch nicht unbedingt zu gebrauchen. Keinem Anwender kann und möchte man es zumuten, auf diese Weise auf Datenbanken zuzugreifen – von der stets drohenden Gefahr einmal abgesehen, daß der (»böse«) Anwender das sorgsam aufgebaute Datenbankmodell mit ein paar gezielten Fehleingaben völlig durcheinander bringt. Eine Benutzeroberfläche ist allerdings nicht nur dazu da, dem Anwender möglichst viel Komfort zur Verfügung zu stellen, sondern auch die Zugriffe auf die Datenbank in geordnete Bahnen zu lenken (wenngleich alle modernen Datenbank-Management-Systeme mit Sicherheitsstufen arbeiten, so daß auch bei direktem Zugriff, etwa über den Visual Data Manager, längst nicht jeder Anwender alles machen darf). Zu den Aufgaben des vielbeschäftigten Datenbankprogrammierers gehören daher nicht nur der virtuose Umgang mit SQL, Recordsets und Cursortypen, sondern auch die Gestaltung der »perfekten« Datenbankoberfläche. Doch Visual Basic ist (leider oder zum Glück) nicht Microsoft Access. Es gibt keine Assistenten, die »per Knopfdruck« eine Datenbankanwendung auf den Bildschirm zaubern (was sich dahinter verbirgt, haben Sie in den letzten acht Kapiteln kennengelernt), und ein Formular verfügt zunächst einmal über keinerlei »Datenbankfähigkeiten«. Fast alles muß von Grund auf programmiert werden, wobei Datenumgebungen die ADO-Programmierung deutlich vereinfachen und kein Visual-Basic-Programmierer davon abgehalten wird, die aus Microsoft Access bekannten Assistenten »nachzuprogrammieren«. In diesem Kapitel geht es um die Oberfläche einer Datenbankanwendung. Sie lernen ein Formular kennen, mit dem sich der Datenbestand der Fuhrpark-Datenbank (am Beispiel einer Tabelle) pflegen

und erweitern läßt, ohne daß dazu ein Visual Data Manager oder SQL-Kommandos bemüht werden müssen.

Sie lernen in diesem Kapitel etwas über:

- Die Validierung der Eingabe
- Die formatierte Anzeige von Datenbankfeldern und das *Format*-Objekt
- Das *DataCombo*-Steuerelement zum Darstellen einer n:1-Beziehung
- Das *HFlexGrid*-Steuerelement für hierarchische Beziehungen
- Ein Formular zum Bearbeiten der *Modelldaten*-Tabelle
- Eine Anmerkung zum Jahr-2000-Problem

9.1 Die Validierung der Eingabe

Die Eingabevalidierung ist ein sehr wichtiger Aspekt bei einer Datenbankanwendung, denn warum sich die Mühe machen, um mit Hilfe von Validierungsregeln, referentieller Integrität oder Triggern (die es bei der Jet-Engine allerdings nicht gibt) die Integrität der Datenbank sicherzustellen, wenn es doch sehr viel einfacher ist, die »unerwünschten« Daten gar nicht erst in die Datenbank hineinzulassen. Konkrekt: Anstatt durch Validierungsregeln auf der Ebene der Jet-Engine sicherzustellen, daß eine Altersangabe nicht negativ sein kann, sorgt man dafür, daß der Benutzer negative Werte gar nicht erst eingeben kann. Die Eingabevalidierung muß in einem Visual-Basic-Formular in der Regel für jedes Steuerelement einzeln durchgeführt werden. Es liegt daher nahe, Eingabefelder des gleichen Typs zu einem Steuerelementefeld zusammenzufassen. Dies bringt den Vorteil, daß sich alle Steuerelemente ein und dieselbe Ereignisprozedur teilen, in der eine Validierung vorgenommen werden kann (Steuerelementefelder sind allerdings nicht nach Microsoft Office portierbar, da *MsForms*-Formulare keine Steuerelementefelder kennen). Doch ob Steuerelementefeld oder einzelnes Steuerelement, die grundsätzliche Problematik ist dieselbe: das Feststellen, ob eine in einem Textfeld getätigte Eingabe gültig ist (bei Listen- und Optionsfeldern ist eine Eingabevalidierung in der Regel überflüssig, da ohnehin keine ungültigen Einträge bzw. Optionen angeboten werden). Die entscheidende Frage ist daher, wo findet die Validierung statt: Unmittelbar nach der Eingabe, an einer zentralen Stelle, z.B. beim Bestätigen des Dialogfeldes, oder wenn der Benutzer im Begriff ist, ein Eingabefeld zu verlassen? Ohne eine Statistik zitieren zu können, dürfte die dritte Variante jene sein, die in der Praxis am häufigsten zum Einsatz kommt.