Kapitel 1

Das kleine Einmaleins der Visual-Basic-Datenbanken

Wer sich mit dem Thema Datenbanken beschäftigt, muß sich mit einer Menge Begriffe, Definitionen und Abkürzungen herumschlagen, die meistens nur indirekt etwas mit der Datenbank und ihren Daten, um die es sich letztendlich immer dreht, zu tun haben. Da gibt es Tabellen und Felder als Grundelemente einer jeden Datenbank und Begriffe wie Relationen, Recordsets, Primärschlüssel, referentielle Integrität (ein wahres Wortmonstrum, mit einer zum Glück recht einfachen Bedeutung) und eine schier unübersehbare Vielfalt an Abkürzungen, wie ADO, DAO, ODBC, OLE DB, RDO, RDS, SQL usw. (dies wurde in der Einleitung bereits angedeutet). Als hoffnungsfroher Visual-Basic-Programmierer, der eigentlich nur eines will, Daten in einer (Access-) Datenbank zu speichern und diese, wenn es erlaubt ist, auch wieder abzurufen, ist man natürlich zunächst geplättet und meistens alles andere als erfreut. Es gibt aber keinen Grund zur Frustration, denn zum einen ist das meiste doch recht simpel, zum anderen darf man nicht erwarten, daß sich ein Gebiet, das in den letzten drei Jahrzehnten immerhin gut 80% der klassischen Datenverarbeitung ausgemacht hat und sicherlich auch 80% der heutigen Informationstechnik ausmacht, in wenigen Sätzen umfassend beschreiben und an einem Wochenende vollständig erschließen läßt. Und auch, wenn es zu diesem Zeitpunkt eher ein wenig zynisch klingen mag: Als Visual-Basic-Programmierer darf man froh sein, daß das Ganze noch relativ überschaubar geblieben ist. Denn wenn es SQL nicht gäbe und Microsoft mit ODBC und jetzt mit OLE-DB/ADO keine verbindlichen Standards geschaffen hätte, sehe die Angelegenheit vielleicht noch sehr viel unübersichtlicher aus (spätestens am Ende dieses Kapitels wissen Sie, was es mit den drei Abkürzungen auf sich hat).

In diesem Kapitel lernen Sie das kleine Einmaleins der Datenbanken kennen, bezogen auf Visual Basic und die »Microsoft-Welt«[1]. Eine gewisse Redundanz läßt sich leider nicht vermeiden. Sie werden einige der in diesem Kapitel vorgestellten Begriffsdefinitionen im weiteren Verlauf des Buches noch einmal lesen. Das ist hoffentlich im Sinne der meisten Leser, denn einige Dinge kann man nicht oft genug wiederholen[2].

Sie lernen in diesem Kapitel etwas über die folgenden Themen:

- ✘ Was ist eigentlich eine Datenbank?
- ✘ Visual Basic und die Datenbanken
- ✘ Die Rolle der Jet-Engine
- ✘ Der Unterschied zwischen der Jet-Engine und Microsoft Access
- ✘ ADO und DAO
- ✘ Die wunderbare »Genialität« von SQL
- ✘ Die Rolle von ODBC
- ✘ Lokale Datenbanken und Remote-Datenbanken
- ✘ Das Client/Server-Prinzip

1.1 Was ist eigentlich eine Datenbank?

Eine Datenbank ist ein Ort, an dem Daten gespeichert werden. Woher der Begriff Datenbank stammt, läßt sich offenbar nicht genau feststellen. Vermutlich ist es lediglich eine Übersetzung des Originalbegriffs »Database«, wobei dazu gesagt werden muß, daß, anders als viele andere Dinge, Datenbanken auch bei uns in Deutschland eine Tradition besitzen, die noch auf die sechziger Jahre zurückgeht. (Es ist also nicht so, daß Datenbanken bei uns erst mit Microsoft Access oder dBase – einer PC-Datenbank der frühen achtziger Jahre – populär wurden.) Wann die erste Datenbank ins Leben gerufen wurde, läßt sich vermutlich nicht genau bestimmen. Ein markantes Datum in der Geschichte der Datenbanken dürfte jene 11. Volkszählung aus

---

1 Dieser Hinweis ist nicht unwichtig, da es natürlich nicht nur Microsoft gibt und andere Datenbankfirmen, wie etwa Oracle, Informix, IBM oder Inprise (früher Borland), leistungsfähige Alternativen anbieten und dabei teilweise auch andere Namen und Abkürzungen verwenden.

2 Was natürlich kein »Freischein« des Autors für endlose Wiederholungen und andere Platitüden sein soll.

dem Jahre 1890 in den USA gewesen sein (die sogenannte »Inventur der Amerikaner«), bei der zum ersten Mal die Daten aller Einwohner auf Lochkarten erfaßt wurden. Die Idee stammte von dem Amerikaner Herman Hollerith (1860–1929), der 1884 ein Patent unter dem Namen »Art of Compiling Statistics« für sein Lochkartensystem anmeldete, das 1889 auch erteilt wurde. Bereits 1896 gründete der geschäftstüchtige Ingenieur die *Tabulating Machine Company*, aus der 1924 die *International Business Machines Corporation* hervorging, die den meisten eher unter dem Kürzel IBM bekannt sein dürfte. Der deutsche Ableger mit dem Namen *Deutsche Hollerith Maschinengesellschaft* (Dehomag) ging 1949 in der IBM auf[1].

Der Begriff Daten *steht universell für alles, das sich erfassen, kategorisieren und damit auch (irgendwo) speichern läßt. Ein Name, die Temperatur des Badewassers, der Kurs einer Aktie, die Lautstärke einer Grille oder der Füllstand eines Tanks sind Daten, die man erfassen (also irgendwo aufschreiben), einordnen (d.h. einer bestimmten Kategorie zuordnen) und abspeichern kann. Werden die Daten systematisch erfaßt, entsteht eine Datenbank. Werden die Daten nach verschiedenen Gesichtspunkten neu zusammengestellt, spricht man von Datenverarbeitung (DV).*

So innovativ und umwälzend Herman Holleriths Idee am Ende des 19. Jahrhunderts auch war, die erfaßten Daten mußten damals mechanisch erfaßt und auf Papier gespeichert werden, was den Zugriff auf die Daten deutlich einschränkte. Eine elektronische Erfassung wurde erst mit dem Aufkommen der Rechenanlagen in den fünfziger Jahren möglich, wobei der kommerzielle Einsatz der Großrechner erst in den sechziger Jahren begann.

Ein weiteres markantes Datum in der Geschichte der Datenbanken sind die Arbeiten der amerikanischen Mathematiker *C. Date* und *E.F. Codd*[2], die Anfang der 70er Jahre zum relationalen Modell der Datenbanken und zur Datenbanksprache SQL führten. Wenngleich dies auch schon wieder 30 Jahre zurückliegt, sind die damals entworfenen Grundregeln für die Organisation einer Datenbank gemäß dem relationalen Modell und die Abfrage und Manipulation per SQL auch heute noch topaktuell. Sie werden im weiteren Verlauf des Buches im Zusammenhang mit einer kleinen Access-Datenbank angewendet.

---

1 Alles wunderschön nachzulesen in dem Buch »Informationen, Daten und Signale« von Rolf Oberliesen (rororo Sachbuch Nr. 7709), das es unter Umständen noch im »Souvenierladen« des Deutschen Museums in München zu kaufen gibt.

2 Wer hat gesagt, daß die Dinge einfach sein müssen?

1.1.1 Die Zukunft der Datenbanken: OLAP(?)

Auch wenn die theoretischen Grundlagen für das, was Sie während der nächsten 300 Seiten beschäftigen wird, bereits etwas älter sind, hat es natürlich auch in den achtziger und neunziger Jahren wichtige Fortschritte im Bereich der Datenbanktechnologien gegeben (Stichwort: »Verteilte und objektorientierte Datenbanken« oder der 1992 verabschiedete ANSI 92-SQL-Standard). Dennoch hat keine dieser Entwicklungen den Datenbankbereich so grundlegend prägen können, wie es mit dem relationalen Modell und SQL der Fall war. Das könnte sich allerdings in den nächsten Jahren (endlich) wieder ändern. Unter den Stichworten *Knowledge Management*, *Data Warehousing* und vor allem On-Line Analytical Processing, kurz OLAP, werden neue Verfahren beschrieben, nach denen verknüpfte Tabellen multidimensional in Form von (unsichtbaren, also nur logisch existierenden) Würfeln dargestellt werden. Auf diese Weise lassen sich die Daten unter Gesichtspunkten auswerten, die mit dem relationalen Modell alleine nicht oder nur aufwendig möglich wären. OLAP, dem eine große Zukunft vorausgesagt wird, ist bereits Realität, da unter anderem der Microsoft SQL Server 7.0 in Form seiner OLAP Services und in Verbindung mit ADO, das einen speziellen OLE DB Provider für OLAP zur Verfügung stellt, eine Möglichkeit bietet, Datenbankdaten in Form eines multidimensionalen Würfels zu analysieren.

Der Begriff Information *steht für Daten, die sich in einem bestimmten Kontext befinden. Die Zahl 23 kann alles bedeuten, wird sie dagegen im Kontext einer Temperaturangabe genannt, erhält sie nicht nur eine bestimmte Bedeutung, sondern auch eine Reihe kontextabhängiger Implikationen. Steht sie im Zusammenhang mit der Lufttemperatur, impliziert sie z.B. den Zustand »angenehm«, steht sie dagegen im Zusammenhang mit dem Inhalt einer Badewanne, impliziert sie den Zustand »eindeutig zu kalt«. In den letzten Jahren fand ein Übergang von der reinen Datenverarbeitung (DV) zur Informationsverarbeitung (IT) statt, bei der es nicht mehr alleine um das Erfassen, Abspeichern und Auswerten von Daten, sondern in zunehmenden Maße auch um die Interpretation dieser Daten geht.*

Anwendungen für OLAP gibt es mehr, als Sie wahrscheinlich zunächst vermuten würden. Allerdings lohnt sich der Aufwand nur dann, wenn große Datenmengen im Spiel sind, aus denen in kürzester Zeit neue Erkenntnisse gewonnen werden sollen. Ein Beispiel von vielen sind jene Einzelhändler, bei denen ein großer Teil der Kundschaft bereits mit der EC-Karte bezahlt und so (oft ohne es zu wissen) mit jeder Transaktion neben dem Verkaufsda-

tum auch Anschriftendaten hinterläßt. Zusammen mit mikrodemoskopischen Daten (etwa dem Kaufkraftquotienten für eine bestimmte Region) sitzt der Einzelhändler auf einer wahren »Daten-Schatztruhe«, die er nur noch mit dem passenden Schlüssel öffnen muß. »Wieviel Prozent der über die letzte Zeitungsbeilage angesprochenen Kunden waren innerhalb der darauffolgenden Tage in meinem Laden und haben für mehr als 100 DM etwas gekauft?« »Wie anziehend ist ein Sonderangebot auf eine bestimmte Region?« »Wo werden die meisten Shorts im Winter verkauft?« Fragen über Fragen, die sich mit einfachem SQL nicht oder nur mit großem Aufwand beantworten lassen. OLAP, das, sowohl als Endanwenderwerkzeug als auch in Form programmierbarer Objekte, fester Bestandteil von Excel 2000 ist, könnte hier eine Antwort bieten.

Das ist für Sie als angehender Datenbankprogrammierer allerdings im Moment nichts mehr als ein Ausblick in eine ferne Zukunft. Am Ende des Buches sollen Sie in erster Linie Access-Datenbanken ansprechen, Tabellen öffnen, mit *Recordset*-Objekten umgehen und einfache SQL-Abfragen formulieren können, nicht aber die Kaufkraft der Region auf zwei Stellen nach dem Komma analysieren. In diesem Kapitel geht es um sehr viel elementarere Fragen. Die wichtigste Frage, nämlich was ist eine Datenbank, wurde noch nicht vollständig beantwortet.

1.1.2 Datenbank = Tabelle + Datensätze + Felder

Sie wissen bereits, daß eine Datenbank ein Ort ist, an dem Daten aufbewahrt werden (wie dieser Ort physikalisch aussieht, wird in Kapitel 2 erklärt). Allerdings sind diese Daten nicht bunt durcheinander gewürfelt, sondern auf verschiedene »Behälter« verteilt, die als Tabellen bezeichnet werden. Jeder dieser Behälter besitzt eine feste Anzahl an Unterteilungen, die man sich als kleine Schubladen vorstellen kann. Eine solche Schublade wird auch als Feld bezeichnet. In jede Schublade kann genau ein »Datum«[1], etwa ein Name oder eine Zahl, abgelegt werden. Möchte man etwa den Namen, das Geburtsdatum und das Lieblingsgetränk einer Person speichern, werden dafür drei Schubladen (Felder) benötigt. Eine Schublade für den Namen, eine weitere Schublade für das Geburtsdatum und schließlich eine dritte Schublade für den Namen des Getränks. Alle Schubladen (Felder) zusammen bilden einen Datensatz. In jedem Datensatz läßt sich (bezogen auf das Beispiel) genau ein Name, ein Geburtsdatum und ein Getränkename speichern. Möchte man weitere Personendaten speichern, so wird einfach eine neue Reihe bestehend aus drei Schubladen, also ein Datensatz, hinzugefügt.

1 Die Einzahl von Daten.

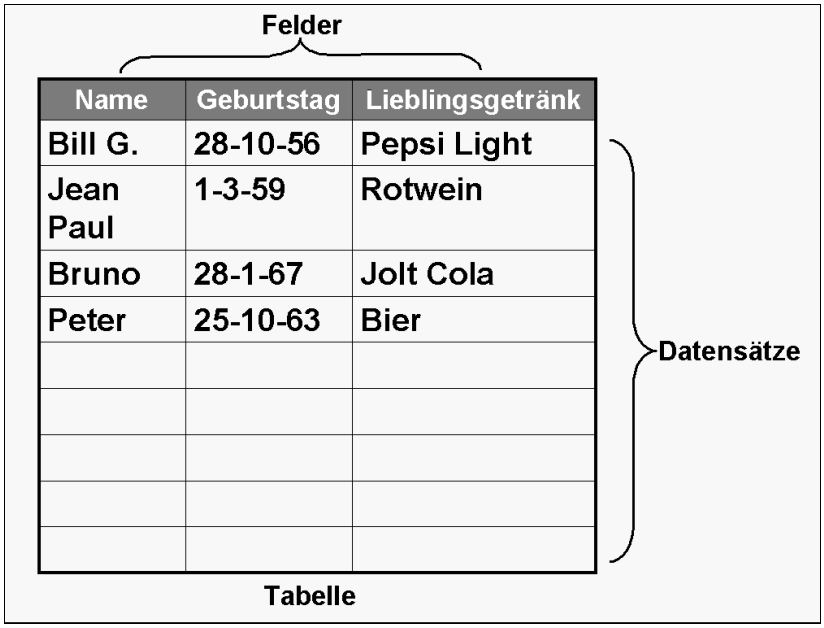


Bild 1.1: Eine Datenbank besteht aus Tabellen, Datensätzen und Feldern

Eine Datenbank besteht aus einer oder mehreren Tabellen. Jede Tabelle enthält keinen, einen oder mehrere Datensätze. Jeder Datensatz besteht aus einer festen Anzahl von Feldern. Alle Datensätze der gleichen Tabelle weisen die gleiche Anordnung an Feldern auf. Jedes Feld enthält genau ein Datenelement.

Gemäß dieser Definition ist eine Datenbank mit einem Karteikasten vergleichbar, in dem auf den einzelnen Kärtchen die Namen und Adressen von Personen aufgeführt sind. Der gesamte Karteikasten ist die Datenbank. Eine Unterteilung (etwa in private und geschäftliche Adressen) entspricht einer Tabelle. Jede Tabelle enthält ein oder mehrere Kärtchen, die als Datensätze bezeichnet werden. Anders als im richtigen Leben muß es auf jeder Karteikarte einer Tabelle die gleiche Unterteilung geben. Jemand, der seine Aufgabe sehr genau nimmt, würde daher, fein säuberlich mit Bleistift und Lineal, auf jeder Karte die gleiche Anzahl an Unterteilungen eintragen, wobei jede Unterteilung, die in diesem Zusammenhang als Feld bezeichnet wird, den gleichen Typ von Daten enthält. Weder für die Anzahl der Unterteilungen noch für die Anzahl der Karteikarten gibt es eine Grenze. Wer einen schnell wachsenden Freundeskreis oder aus geschäftlichen Gründen einen schnell wachsenden Kundenkreis besitzt (erfahrungsgemäß geht im richtigen Leben manchmal das eine mit dem anderen einher), schafft einfach neue

Karteikarten oder neue Kästen an. Dem Wachstum der »Datenbank« sind rein physikalisch keine Grenzen gesetzt.

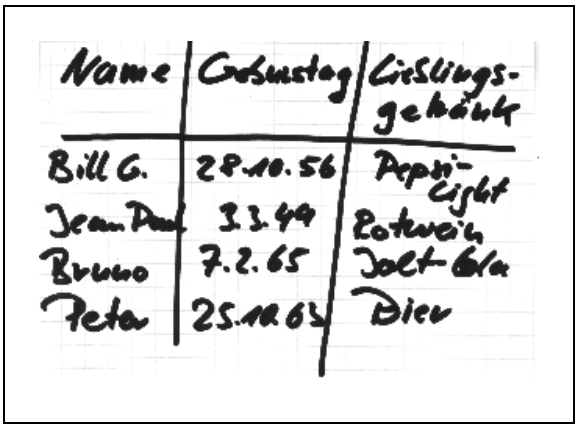


Bild 1.2: Die gute alte Karteikarte entspricht einem einzelnen Datensatz, wo jeder Eintrag in einer Spalte ein Feld darstellt

1.1.3 Wie wird aus Daten eine Datenbank?

Eine Antwort auf diese wichtige Frage kann im Moment nur angedeutet werden, da Sie die wichtigsten Regeln zur Umsetzung einer Datenbank erst in Kapitel 2 und 3 kennenlernen werden. Sie haben die Daten, wie wird daraus eine Datenbank? Der erste Schritt ist das Eingeben der Daten in elektronischer Form, so daß sie auf einem Datenträger, wie einer Festplatte, gehalten werden können. Das ist einfacher, als Sie es vielleicht vermuten. Windows verfügt, wie jedes Betriebssystem, über das denkbar einfachste »Datenerfassungsprogramm«, den Editor. Starten Sie also Notepad (den Windows-Editor), und geben Sie Ihre Daten Datensatz für Datensatz ein. Die einzige Bedingung ist, daß Sie die Daten in Form von Datensätzen eingeben. Das bedeutet konkret, daß die erste Zeile die Namen der einzelnen Felder enthält. Also zum Beispiel:

Name, Geburtstag, Lieblingsgetränk

Nun folgen Zeile für Zeile die einzelnen Datensätze, wobei Sie stets die gleiche Reihenfolge einhalten und die einzelnen Felder durch ein Trennzeichen, z.B. ein Semikolon, trennen müssen. Also zum Beispiel:

Jean Paul, 3.3.49, Rotwein
Bill G., 28.10.56, Pespi Light
Bruno, 7.2.65, Jolt-Cola
Peter, 25.10.63, Bier

Speichern Sie das Ganze nun in einer Textdatei ab – fertig ist die »Datenbank«. Daß es sich dabei tatsächlich um eine (Pseudo-)Datenbank handelt, wird deutlich, wenn Sie die Schrittfolge in Kapitel 10 (»Datenbankzugriff auf eine Textdatei«) durchführen, durch die Sie jede (tabellarisch aufgebaute) Textdatei wie eine Datenbank ansprechen können. Natürlich ist dies noch keine echte Datenbank, da wichtige Merkmale fehlen, doch erfüllt auch eine Textdatei die eingangs getroffene Definition. Mit der wichtigen Ausnahme, daß jede Textdatei für eine Tabelle steht. Soll die Datenbank aus mehreren Tabellen bestehen, müssen alle Textdateien in einem Verzeichnis untergebracht werden. In diesem Fall spielt das Verzeichnis die Rolle der Datenbank, die aus mehreren Tabellen besteht.

Bild 1.3: Der Inhalt einer »Datenbank« einmal vom Editor ...

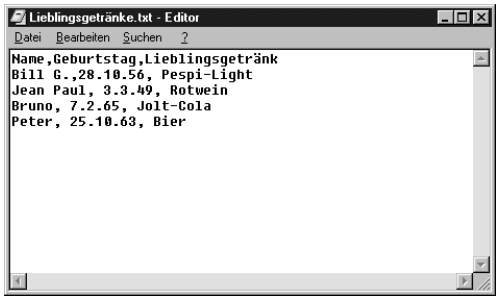
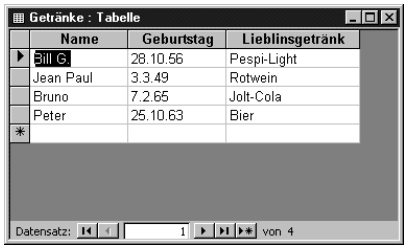


Bild 1.4: ... und einmal durch Microsoft Access dargestellt



Na wunderbar, warum sind wir dann nicht einfach beim Karteikartensystem geblieben? Statt dessen müssen wir viel Geld (demnächst in Euro) ausgeben, um genau das zu haben, was unsere »Vorfahren« scheinbar mit etwas Kleingeld und sehr viel weniger Streß schon hatten.

Nun, dieser Vergleich ist nur scheinbar so attraktiv. Ein Karteikasten ist zwar als »Datenhalde« prinzipiell geeignet, versagt jedoch, wenn es um das Abfragen der Daten geht. Stellen Sie sich nur vor, wie lange es dauern würde, in einer Kartei mit 1000 Freunden nur alle jene herauszusuchen, die seit mehr als fünf Jahren nichts mehr von sich hören lassen haben. Was nach dem »guten alten« System einen Nachmittag in Anspruch nehmen würde,

läßt sich mit einer elektronischen Datenbank im Bruchteil einer Sekunde erledigen. Wenn Sie jedoch noch feststellen möchten, ob zwischen Ihrem Versenden von Weihnachtskarten (über das Sie ebenfalls Buch geführt haben) und der Kontaktfreudigkeit Ihrer Freunde ein Zusammenhang besteht, so läßt sich dieser (etwa mit dem neuen OLAP) in wenigen Sekunden erkennen, während die traditionelle Methode vermutlich im Ansatz scheitern würde.

Das soll nicht heißen, daß Karteikarten (oder Aktenordner) als Datenmedium prinzipiell untauglich wären. Sie bieten einfach nicht mehr die erforderliche Flexibilität[1]. Ein wenig anders sieht es aus, wenn Sie Ihre Karteikarten mit dem unter Windows 3.1 damals populären Karteimanager verwalten. Da dieser seine Daten in tabellarischer Textform speichert, ist es kein großer Aufwand, in Visual Basic einen OLE DB-Provider zu programmieren, so daß die »Cardfile-Datenbank«, etwa per SQL, wie eine richtige Datenbank angesprochen werden kann. Möglich ist dank der Flexibilität von OLE DB und ADO praktisch alles. Voraussetzung ist natürlich, daß die Daten in elektronischer Form vorliegen.

Halten wir fest: Eine Datenbank ist ein Ort, an dem Daten in tabellarischer Form gespeichert werden. Jede Tabelle enthält Datensätze, ein Datensatz besteht aus einer bestimmten Anzahl an Feldern. Ein Datensatz ist damit eine Informationseinheit innerhalb einer Datenbank. Vergleicht man eine Datenbank mit einem Karteikasten, so entspricht ein Datensatz einer Karteikarte. Die Informationen in einem Datensatz sind nochmals in sogenannte Felder unterteilt. Bei einer Adreßdatenbank mit einer Tabelle können in einem Datensatz z.B. die Felder Nachname, Vorname, Straße, Hausnummer, Postleitzahl, Ort und Telefonnummer existieren.

1.1.4 Datenbanken – eine technische Definition

Mit dieser hoffentlich recht einfachen Definition kann man es für den Anfang bewenden lassen. Wer mehr über das Wesen der Datenbanken verstehen will, muß ein wenig tiefer einsteigen. Ein kurzer Abstecher in die Informatikvorlesung 1. Semester eröffnet tiefergehende Einblicke. Zu den wichtigsten Merkmalen einer Datenbank gehören:

- ✘ Strukturierung der Daten
- ✘ Trennung von Daten und Anwendung

1 Auf meinen Schreibtisch steht nach wie vor ein File-O-Flex mit kleinen Kärtchen. Ein »Personal Organizer« hat sich noch nicht durchsetzen können.

Der erste Punkt ist schnell erklärt. Eine Datenbank dient der Strukturierung von Daten (wie Sie bereits wissen in Tabellen, Datensätze und Felder). Auch der zweite Punkt ist im Prinzip selbstverständlich. Die Daten werden in einer Datenbank und nicht im Programmtext gespeichert – Anwendung und Daten sind voneinander unabhängig. Damit ergeben sich eine Reihe weiterer Merkmale:

- ✘ Datenunabhängigkeit
- ✘ Integrität, d.h. Konsistenz, zum Beispiel eine automatische Validierung (Kontrolle) eingegebener Daten
- ✘ Datensicherung
- ✘ Datenschutz
- ✘ Zeitliche Permanenz (Daten werden durch die Datenbank gespeichert)
- ✘ Spezifische Benutzersichten (der Benutzer kann festlegen, auf welche Weise die Daten präsentiert werden sollen)

Die ersten beiden Punkte erscheinen als selbstverständlich, denn der Aufbau einer Datenbank ist unabhängig von den Daten, die darin gespeichert werden, da sich prinzipiell sämtliche Daten speichern lassen sollten. Datenbanken sollten für ihre eigene Konsistenz sorgen, das heißt verhindern, daß ein Anwender unsinnige Daten eingeben oder einen Datensatz in der Stammdatentabelle löschen kann, auf den in anderen Datensätzen verwiesen wird. Datensicherung und Datenschutz sind ebenfalls wichtige Bereiche, wobei zumindest Microsoft Access den ersten Punkt dem Anwender überläßt und beim zweiten Punkt nicht gerade brilliert. Zeitliche Permanenz bedeutet nichts anderes, als daß sich die Datenbank um das Abspeichern der eingegebenen Daten kümmert. Auch die spezifischen Benutzersichten sind jedem Access-Anwender bestens vertraut. Es bedeutet, daß sich der Anwender verschiedene Ansichten entwerfen kann, die festlegen, auf welche Weise der Inhalt einer oder mehrerer Tabellen zurückgegeben (also angezeigt) wird. Möchte ein Anwender von den Aufträgen in der Auftragstabelle nur die Beträge sehen oder gleich die Namen und den Kontostand (bezogen auf die bisher getätigten Bestellungen) der Auftraggeber? Diese Information mag nicht direkt in der Datenbank enthalten sein, doch durch Zusammenstellen einer Ansicht tut die Datenbank so, als stammen alle Daten aus einer Tabelle.

1.2 Visual Basic und die Datenbanken

Von der Welt der abstrakten Begriffe und Definitionen geht es nun in jene Welt zurück, der wir gerne einen Teil unserer kostbaren Frei- und Arbeitszeit widmen, der Visual-Basic-Programmierung. Der Zugriff auf Datenbanken spielt bei der Visual-Basic-Programmierung eine sehr wichtige Rolle. Doch Visual Basic, genauer gesagt VBA (die Programmiersprache von Visual Basic), enthält keinen einzigen »Datenzugriffsbefehl«[1] und ist daher von Haus alleine nicht in der Lage, auf eine Datenbank zuzugreifen. Für den Datenzugriff benutzt Visual Basic vielmehr ein allgemeines Prinzip, das grundsätzlich immer dann zur Anwendung kommt, wenn ein Visual-Basic-Programm auf Funktionen zugreifen soll, die sich in anderen Programmen, DLLs oder Systemdateien mit einer sogenannten *COM-Schnittstelle* befinden: Die Einbindung einer Objektbibliothek, die die Namen von Objekten und deren Eigenschaften, Methoden und Ereignisse enthält. Über den Menübefehl PROJEKT/VERWEISE wird durch die Auswahl der entsprechenden Objektbibliothek die Verbindung zwischen dem Visual-Basic-Programm und einer Datenbankschnittstelle hergestellt. Letztere enthält eine Reihe allgemeiner (Datenbank-)Objekte, mit denen z.B. Datenbankabfragen durchgeführt werden.

VBA enthält keine eingebauten »Datenbankbefehle«. Der Zugriff auf eine Datenbank (bzw. eine Datenquelle) wird über eine Objektbibliothek hergestellt, die über den Menübefehl PROJEKT/VERWEISE in ein Projekt eingebunden wird.

Grundlage für den Datenbankzugriff ist das Component Object Modell *(COM), das als eine Systemerweiterung von Windows unter anderem dafür sorgt, daß sich zwei Objekte in einer Anwendung über Eigenschaften, Methoden und Namen ansprechen können.*

1 Von den netten »Oldtimer-Befehlen« Open, Close, Get und Put einmal abgesehen – mehr dazu am Ende des Kapitels.

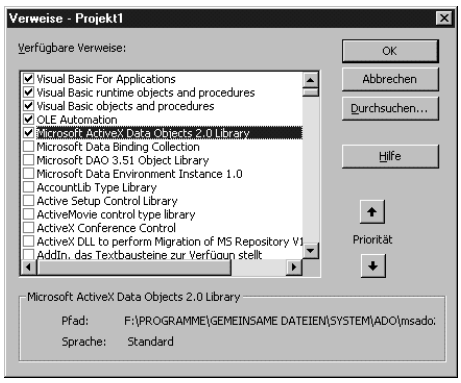


Bild 1.5: Erst durch Einbinden einer Referenz auf eine Objektbibliothek wird Visual Basic »datenbankfähig«

Um ein VBA-Projekt »datenbankfähig« zu machen, gehen Sie stets wie folgt vor. Sie öffnen das PROJEKT-Menü, wählen den Eintrag VERWEISE und wählen jene Objektbibliothek aus, über deren Objekte der Zugriff auf die Datenbank durchgeführt werden soll. Wie Sie im weiteren Verlauf dieses Kapitels noch erfahren werden, gibt es dafür zwei Alternativen:

- ✘ Die Microsoft ActiveX Data Objects Library 2.0
- ✘ Die Microsoft DAO Object Library 3.51

Was es damit auf sich hat und welche Unterschiede zwischen beiden Objektbibliotheken bestehen, wird noch ausführlich erörtert. Zunächst muß die Rolle des Datenbank-Management-Systems geklärt werden.

1.3 Das Datenbank-Management-System

Eine Datenbank ist letztendlich nichts anderes als eine Ansammlung von Daten in einer einzelnen Datei oder einer eigens dafür vorgesehenen Struktur auf der Festplatte. Eine Programmiersprache, wie VBA, greift auf diese Daten über Objekte zu. Doch diese Objekte führen nicht den eigentlichen Datenbankzugriff durch. Sie sind lediglich Vermittler zwischen dem Programm und dem Datenbank-Management-System, dessen Aufgabe die Verwaltung der Datenbank ist. Der Begriff Datenbank-Management-System (kurz DBMS, wenngleich diese Abkürzung in der Welt der PC-Datenbanken nur selten verwendet wird) darf nicht zuviel implizieren. Es kann eine gigantisch große (und entsprechend teure) Software sein, es kann aber auch ein Miniprogramm sein, das auf eine einzelne HD-Diskette paßt. Allen DBMS ist gemeinsam, daß sie folgende Grundtätigkeiten ausführen können:

- ✘ Das Erstellen von Datenbanken und das Hinzufügen von Tabellen und Feldern
- ✘ Das Erstellen von Abfragen mit einer Abfragesprache (in der Regel per SQL)
- ✘ Administrative Aufgaben, wie das Komprimieren der Datenbank oder das Exportieren bzw. Importieren von Daten

Visual Basic ist (anders als z.B. Microsoft Access) kein vollständiges DBMS. Es ist lediglich in der Lage, mit Hilfe von Objektbibliotheken über ein (im Prinzip beliebiges) DBMS auf Datenbanken zuzugreifen und die oben beschriebenen Tätigkeiten mit Hilfe des DBMS auszuführen. Da die Datenbankprogrammierung bei Visual Basic einen hohen Stellenwert besitzt, wird Visual Basic seit der Version 3.0 mit einem DBMS ausgeliefert. Es ist die Jet-Engine, die auch von Microsoft Access verwendet wird und im nächsten Abschnitt besprochen wird. Darüber hinaus bietet seit der Version 5.0 die Enterprise Edition von Visual Basic mit dem Microsoft SQL-Server ein weiteres, überaus leistungsstarkes DBMS an.

Gehorcht ein DBMS den allgemeinen Regeln relationaler Datenbanken (mehr dazu in Kapitel 2), wird es auch als RDBMS (Relationales DBMS) bezeichnet. Das ist allerdings nur eine allgemeine Klassifizierung und kein Qualitätsmerkmal. Microsoft Access unterstützt zwar das relationale Datenbankmodell, ist aber dennoch ein »typisches« RDBMS, da es z.B. Datenbanken in Dateien speichert.

Sie wissen nun, was eine Datenbank ist, welche Rolle das DBMS übernimmt und wie man prinzipiell von einem Visual-Basic-Programm über Objektschnittstellen auf ein DBMS und damit auf eine Datenbank zugreift. Im nächsten Abschnitt wird mit der Jet-Engine das wichtigste DBMS für Visual-Basic-Programmierer vorgestellt.

1.4 Die Rolle der Jet-Engine

Der Begriff »Jet-Engine« ist erfahrungsgemäß nicht allen Visual-Basic-Programmierern geläufig. Häufig werden die Begriffe Jet-Engine, Microsoft Access und Access-Datenbank auch synonym verwendet, was streng genommen nicht korrekt ist und daher bei der Frage, welche Form der Datenbankunterstützung Visual Basic bietet, nicht selten Verwirrung stiftet. Microsoft Access ist eine (Büro-)Anwendung, genau wie Visual Basic und

Microsoft Word. Was ist allen drei Anwendungen gemeinsam? Sie werden mit ein und demselben Datenbank-Management-System (DBMS) ausgeliefert. Dieses DBMS heißt *Jet-Engine*. Die Jet-Engine ist kein ausführbares Programm, sondern liegt in Gestalt mehrerer Systemdateien vor (Sie können sie daher nicht aufrufen). Diese Systemdateien (DLLs) können über ihre Objektschnittstelle DAO, über ODBC und jüngst über OLE DB und ADO von vielen Programmen aus angesprochen werden – von Visual Basic (VBA), von Java wie von VBScript. Microsoft Access ist lediglich der überaus komfortable Rahmen, in dem sich Jet bedienen läßt und mit dem sich Anwendungen auf der Basis von Jet aufbauen lassen. Da Microsoft Access und die Jet-Engine eng miteinander verflochten sind, ist es für einen Anwender praktisch unmöglich, eine Trennlinie zu ziehen (aus der Sicht eines Anwenders ist dies auch gar nicht erforderlich). Auch für Programmierer ergibt sich diese erst bei näherer Betrachtung. Deutlicher wird der Unterschied bei Microsoft Access 2000, wo die Anwender auswählen können, ob eine neue anlegte Datenbank auf der Jet-Engine (die in der Version 4.0 vorliegt) oder der neuen Microsoft Desktop Engine (MDE), der Desktop-Version des Microsoft SQL-Servers 7.0, basieren soll.

Die Jet-Engine ist ein kleines, aber sehr leistungsfähiges DBMS. Allerdings ist es keine eigenständige Anwendung. Um auf die Jet-Engine zuzugreifen, werden z.B. Microsoft Access, der Visual Data Manager von Visual Basic (siehe Kapitel 3) oder ein Visual-Basic-Programm benötigt.

Die Jet-Engine unterstützt gleiche mehrere Datenbankformate, wobei das Datenbankformat die Art und Weise beschreibt, wie die Daten in der zugrundeliegenden Datei organisiert werden:

- ✘ Das eigene Format (auch Access- oder Mdb-Format genannt)
- ✘ Verschiedene ISAM-Formate (z.B. dBase, Paradox, FoxPro usw.)
- ✘ ODBC-Datenbankformate

Das mit Abstand wichtigste Format ist das »Access-Format«. Hier liegt die Datenbank im gleichen Format vor (zu erkennen an der typischen Erweiterung *.Mdb*), das auch von Microsoft Access verwendet wird[1].

1 Wenn Sie über die Frage grübeln, was es eher gab: Die Jet-Engine wurde für Microsoft Access 1.0 entwickelt, gleichzeitig aber so konzipiert, daß sie auch von anderen Produkten genutzt werden kann. Ein »Henne-und-Ei«-Problem gibt es hier nicht.

1.5 Allgemeines zu den »Objekten«

Sie werden in diesem Buch eine Menge über Objekte lesen. Als erfahrener Visual-Basic-Programmierer ist Ihnen das alles natürlich vertraut, denn bei Visual Basic sind von Anfang an Objekte in Form von Steuerelementen, Formularen, dem *Clipboard*-Objekt und natürlich in Form von Klassen im Spiel. Ein Objekt ist, das noch einmal zur Wiederholung, ein Programmname, der über Eigenschaften und Methoden angesprochen wird und in einigen Fällen auch auf Ereignisse reagieren kann.

Beim Datenbankzugriff stehen die Objekte im Mittelpunkt. Microsoft hat sich dafür entschieden, sämtliche Datenbankzugriffe ausschließlich über Objekte regeln zu lassen. VBA besitzt, wie Sie bereits erfahren haben, keinen einzigen »Datenbankbefehl«. Die Datenbankobjekte werden in der Entwicklungsumgebung über PROJEKT/VERWEISE einem Projekt hinzugefügt und stehen anschließend zur Verfügung, als wären sie schon immer Teil von VBA gewesen. Allerdings heißen die Datenbankobjekte nicht etwa »VBA Data Objects«, sondern *Active Data Objects* (ADO), *Remote Data Objects* (RDO) und *Data Access Objects* (DAO). Weitere Objektschnittstellen gibt es übrigens nicht, so daß Sie nicht befürchten müssen, mit weiteren Kürzeln konfrontiert zu werden (zumindest nicht was die Datenbankobjekte angeht).

In diesem Buch finden Sie an vielen Stellen den Begriff ADO-Objekte (eigentlich müßte es ja »AD-Objekte« heißen, weil das »O« natürlich für »Objects« steht, doch hat sich diese Kurzschreibweise nicht eingebürgert).

Wie in der Einleitung schon dargestellt wurde, bespricht dieses Buch nicht die DAOs (oder die RDOs). Im Mittelpunkt stehen die ADOs. Dahinter verbergen sich sieben Objekte (in der Version 2.1 – in Zukunft können es mehr werden), die für den Zugriff auf eine Datenquelle benutzt werden müssen. Welche Objekte dies sind, wird in Kapitel 5 ausführlich erläutert. Sie sollten sich den Begriff ADO-Objekte aber jetzt schon merken und wann immer Sie ihn lesen oder hören an diese sieben Objekte und ihre spezifischen Eigenschaften, Methoden und auch Ereignisse denken. Das ist der Kernpunkt bei der Datenbankprogrammierung mit Visual Basic, Microsoft Office usw.

Entsprechend steht auch der Begriff DAO-Objekte für verschiedene Objekte, die für den Datenbankzugriff verwendet werden können. Sie tragen allerdings andere Namen und besitzen eine andere Bedeutung. Eine kurze Gegenüberstellung ADO-DAO finden Sie im Anhang C.

Erschrecken Sie bitte nicht angesichts von Begriffen wie ADO-Objekte, Objektmodell, Objektschnittstelle usw. Hinter all diesen Begriffen steht ein simples Prinzip. Da die Begriffe absolut essentiell für die Datenbankprogrammierung sind[1], sollten Sie sich mit ihnen möglichst schnell anfreunden.

1.6 ADO oder DAO?

Keine Sorge, in die Zwischenüberschrift hat sich kein tückischer Buchstabendreher eingeschlichen, es erwartet Sie auch kein Ratespielchen mit zweifelhaftem Hauptgewinn. Es geht vielmehr um die Rolle der zwei wichtigsten Datenbankschnittstellen für den Zugriff auf die Jet-Engine im speziellen und alle übrigen Datenbanken im allgemeinen. Wie es in der Einleitung schon beschrieben wurde, gab es bis Visual Basic 5.0 nur DAO. Mit Visual Basic 6.0 wurde als Alternative ADO eingeführt (wenngleich ADO unabhängig von Visual Basic ist). Für jemanden, der gerade erst mit der Datenbankprogrammierung beginnt, ist es wie mit den meisten jungen Leuten, die nach der Wiedervereinigung geboren wurden. Für sie ist das Kürzel DDR nur ein mehr oder weniger umfangreicher Teil der Geschichtsbücher, zu dem sie vermutlich keinerlei persönlichen Bezug entwickeln werden. Ähnlich ist es mit DAO, dem in ein paar Jahren vermutlich nur eine freundliche Fußnote in den Datenbankprogrammierbüchern zuteil werden wird. ADO ist die Zukunft, und das ist (bis auf einige Ausnahmen) auch gut so. Anders sieht es für die Visual-Basic-Programmierer aus, die bereits mehr oder weniger umfangreiche Programme auf der Basis von DAO entwickelt haben. Sie werden ihre Programme zwar auch in Zukunft ausführen und pflegen können, müssen aber mittelfristig auf ADO »umlernen«. Allerdings, und das werden Sie auch noch öfter lesen, ist es kein Problem, ADO und DAO in einem Programm gemeinsam zu benutzen. Es ist also denkbar, ein Programm schrittweise auf ADO umzustellen.

1.7 Die wunderbare »Genialität« von SQL

Es gibt Erfindungen, die sind so genial, daß man sich kaum noch vorstellen, wie das Leben ohne sie aussehen würde. Das Telefon fällt in diese Kategorie, der Verbrennungsmotor, das Fernsehen sicher auch und mit Penicillin und dem Pizzaservice ließe sich die Liste noch viel weiter fortführen. In der Welt der DV ist die *Structured Query Language* (zu deutsch »strukturierte

1 Ich hoffe, dies ist die korrekte Steigerungsform.

Abfragesprache«, meistens aber ausgesprochen wie »ciehquell« oder kurz SQL) eine solche geniale Erfindung. Genial deswegen, weil SQL genau das richtige Werkzeug für den sehr wichtigen und praktisch allgegenwärtigen Zweck der Daten(bank)abfrage ist, weil es zum richtigen Zeitpunkt kam (nämlich so früh, daß es nicht in die »Machtspielchen« der großen Software-Firmen geraten konnte) und weil es von jedem vorbehaltlos akzeptiert wird (was nicht heißen soll, daß es nicht verbesserungswürdig ist)[1].

SQL ist keine Programmiersprache, sondern eine Beschreibungssprache. Anstatt genau festzulegen, wie eine Datenbank geöffnet und ihr Inhalt ausgewertet werden soll, sagt man per SQL nur, welche Daten man erhalten oder welche Daten auf welche Weise verändert werden sollen. Um das genaue Wie und Wo kümmert sich das DBMS. SQL besteht aus Kommandos, Funktionen und Operatoren. Das vermutlich einfachste SQL-Kommando sieht wie folgt aus:

```
SELECT * FROM Authors
```

Dieses SQL-Kommando gibt alle Datensätze der Tabelle *Authors* zurück. Möchte man lediglich die Namen aller Autoren sehen, die mit dem Buchstaben S beginnen, muß das *SELECT*-Kommando ein wenig erweitert werden:

```
SELECT Author FROM Authors Where Author Like 'S*'
```

Auf diese Weise läßt sich sehr genau angeben, nach welchen Kriterien die Datenbank durchsucht werden soll, wobei es natürlich auch möglich ist, verschiedene Tabellen oder gar verschiedene Datenbanken in die Suche einzubeziehen. Machen Sie sich im Moment noch keine Gedanken, auf welche Weise SQL und VBA zusammenspielen und wie z.B. Datensätze an ein Programm »zurückgegeben« werden. Spätestens in Kapitel 7 wird diese Angelegenheit ausführlich besprochen.

SQL besitzt nicht nur Kommandos zur Datenmanipulation (diese Kommandos bilden die *Data Manipulation Language*, kurz DML), sondern auch Kommandos zur Datendefinition (die *Data Definition Language*, kurz DDL), mit denen sich z.B. neue Tabellen oder ganze Datenbanken anlegen lassen.

Wie universell SQL (dank ADO) unter Windows ist, soll das folgende kleine Experiment deutlich machen. Es setzt allerdings voraus, daß Sie mit Win-

1 Wer sich über den aktuellen Stand von SQL (Stichwort: SQL3) informieren möchte, findet unter *http://www.jcc.com/sql_stnd.html* eine Übersicht. Doch Vorsicht, SQL spielt sich nach wie vor zum größten Teil in der Groß-DV ab. Begriffe wie Windows, Visual Basic oder ADO finden sich nicht im SQL-Standarddokument.

dows 98 arbeiten und/oder der *Windows Scripting Host* (eine kleine Windows-Erweiterung, die man bei Windows 95 und Windows NT 4.0 kostenlos nachinstallieren kann – Sie finden ihn im Internet z.B. unter *http://msdn.microsoft.com/scripting/windowshost/download/default.htm)* – installiert wurde. Außerdem müssen bereits die ADO-Objekte installiert sein, was z.B. durch die Installation von Visual Basic 6.0 automatisch geschieht. Haben Sie diese allgemeinen Voraussetzungen erfüllt, führen Sie folgende Schritte aus:

Schritt 1:

Starten Sie Notepad (den Windows-Editor), und geben Sie folgende (VBScript-) Befehle ein:

```
Option Explicit
Dim Rs, Cn
Set Cn = CreateObject("ADODB.Connection")
Set Rs = CreateObject("ADODB.Recordset")
Cn.Provider = "Microsoft.Jet.OLEDB.3.51"
Cn.ConnectionString = _
"C:\Programme\Microsoft Visual Studio\VB98\Biblio.mdb"
Cn.Open
Set Rs.ActiveConnection = Cn
Rs.Source = _
"Select * From Authors Where Author = 'Tischer, Michael'"
Rs.Open
MsgBox Rs.Fields("Au_ID").Value
Cn.Close
```

Schritt 2:

Überprüfen Sie bitte den Datenbankpfad in der Befehlszeile:

```
Cn.ConnectionString = _
"C:\Programme\Microsoft Visual Studio\VB98\Biblio.mdb"
```

Hier muß unbedingt der korrekte Pfad der Datei *Biblio.mdb* auf Ihrem PC eingetragen werden. Bei *Biblio.mdb* handelt es sich um die zweite Demodatenbank (neben *Northwind.mdb*), die von Visual Basic 6.0 automatisch installiert wird. Sie finden sie im Visual-Basic-Verzeichnis.

Schritt 3:

Speichern Sie die Textdatei unter dem Namen *SQLAbfrage.vbs*. Während der Dateiname keine Rolle spielt, ist die Erweiterung *.vbs* wichtig, da die Datei als Scriptdatei erkannt werden soll.

Schritt 4:

Öffnen Sie die Scriptdatei (z.B. per Doppelklick). Wenn Sie alles richtig gemacht haben (und der Windows Scripting Host und die ADOs auf Ihrem PC installiert sind), sollte nach einer kurzen »Bedenkzeit« die Zahl 1155 angezeigt werden. Dabei handelt es sich allerdings nicht um die Antwort auf alle Fragen des Universums, sondern lediglich um den Inhalt des Feldes *Au_Id* in der Tabelle *Authors* der Datenbank *Biblio.mdb* des Autors »Michael Tischer«, dessen Datensatz über eine SQL-Anweisung lokalisiert wurde.

Sie sehen an diesem kleinen Beispiel: Um Datenbankzugriffe per SQL durchführen zu können, benötigen Sie weder Visual Basic noch Microsoft Access. Dank der *Active Data Objects* (ADOs) und einem kleinen Scriptprogramm (in diesem Beispiel ist es ein VBScript-Programm), das auf diese Objekte zugreifen kann, ist SQL ganz allgemein unter Windows einsetzbar. Natürlich können Sie das Beispiel 1:1 unter Visual Basic (bzw. allgemein VBA) ausführen, doch ich denke, daß diese kleine Demonstration etwas eindrucksvoller ist.

1.8 Beziehungen zwischen Tabellen – das relationale Modell

Sie wissen inzwischen, daß eine Datenbank aus Tabellen, eine Tabelle aus Datensätzen und ein Datensatz aus Feldern besteht. Dieses einfache Modell ist aber noch nicht jenes Modell, das die moderne Datenbankentwicklung kennzeichnet. Das moderne Modell basiert auf Relationen zwischen Tabellen und wird daher als *relationales Datenbankmodell* bezeichnet. Es geht zwar bereits auf die 70er Jahre zurück, ist aber auch fast dreißig Jahre später das dominierende Modell im Bereich der PC-Datenbanken.

Haben Sie das allgemeine Modell der Datenbanken verstanden, also die Rolle von Tabellen, Datensätzen und Feldern, ist es zum relationalen Modell nicht mehr weit. Es basiert auf diesem einfachen Modell, packt es aber in einen mathematischen Kontext (der in diesem Buch aber nicht vertieft wird) und verallgemeinert es dabei.

Gemäß dem relationalen Modell besteht eine Datenbank aus Tabellen, die als relationale Tabellen oder einfach nur Relationen bezeichnet werden. Jede Tabelle enthält einen oder mehrere Datensätze, die in diesem Zusammenhang allerdings als *Reihen* (engl. »rows«) bezeichnet werden. Wo sich diese Tabellen physikalisch aufhalten, spielt genauso wenig eine Rolle wie die Anzahl der Tabellen und Datensätze sowie die Reihenfolge der einzelnen

Reihen in einer Tabelle (es gibt also anders als z.B. in einer Excel-Tabelle oder einem Visual-Basic-Grid keine festgelegte Sortierreihenfolge).

Jede Reihe besteht aus einem oder mehreren Feldern, die in diesem Zusammenhang auch als Spalten (engl. »columns«) bezeichnet werden. Eine Tabelle setzt sich demnach aus Reihen und Spalten zusammen. Nach dem relationalen Modell ist eine Reihe nicht zwingend an eine Tabelle gebunden, denn durch eine Datenbankabfrage können mehrere Spalten zu einer neuen (virtuellen) Tabelle kombiniert werden, die für den Anwender (und damit auch für den Programmierer) als eine Tabelle mit eigenen Zeilen und Reihen erscheint.

Halten wir fest: Eine Reihe in einer Tabelle entspricht einem Datensatz, eine Spalte einem Feld des Datensatzes. Eine Datenbank kann eine beliebige Anzahl an Tabellen besitzen.

Bislang war alles noch einfach, und das wird auch so bleiben. Das Attribut »relational« leitet sich von dem Umstand ab, daß zwischen den Feldern verschiedener Tabellen Beziehungen bestehen können, die in diesem Fall ebenfalls Relationen heißen. Relationen werden am besten an einem konkreten Beispiel deutlich. Stellen Sie sich eine Datenbank eines kleinen PC-Händlers vor, der sowohl die Adressen seiner Kunden als auch die Aufträge dieser Kunden in einer Datenbank speichern möchte. Der erste Ansatz könnte darin bestehen, alle Daten in einer Tabelle abzulegen:

Name	Adresse	Bestell-datum	Artikel-Name1	Artikel1-Preis	Artikel2-Name	Artikel2-Preis
Bernd	Hohe Straße 1, 22333 Hamburg	2.3.99	Celeron 333-CPU	189.00		
Dieter	Westanlage 52, 35398 Gießen	4.5.95	T8-Transputer	999.00	256 Mbyte RAM	159.00

usw.

Bereits diese aus zwei Datensätzen bestehende Tabelle macht deutlich, daß dieser Ansatz scheitern muß. Es wäre sehr ungeschickt, zu jedem Auftrag sämtliche Kundendaten zu speichern. Nicht nur, daß die Tabelle dadurch relativ umfangreich werden würde, das Ausmaß an identischen Daten (Redundanz) wäre enorm. Für diesen Fall bietet das relationale Modell eine Lösung an: das Verteilen der Daten auf mehrere Tabellen und das Herstellen von Beziehungen zwischen den Feldern einzelner Tabellen. Bezogen auf unseren PC-Händler müßte dieser als erstes eine Tabelle für seine Kundendaten einführen:

KundenNr	Name	Straße	Ort	PLZ	Telefon	eMail
1001	Dieter	Westanlage 52	Gießen	35398	–	–
1002	Reinhold	Henselstraße 4	Gießen	35396	–	–

usw.

Diese Tabelle (die natürlich rein fiktive Daten enthält) wird auch als *Stammdatentabelle* bezeichnet, da die darin enthaltenen Daten den Kern (Stamm) des Datenbestands bilden und, anders als z.B. die Auftragsdaten, relativ selten geändert werden.

Auch für die Produkte wird eine eigene Tabelle angelegt:

ProduktNr	Produkt	Typ	Distributor	EK-Preis	AnzahlLager
5001	Celeron-CPU 333	1	ABC	179,00	33
5001	T8-Transputer	3	DEF	555,00	2

usw.

Auch hier sind die Inhalte, wie z.B. die Produktnummern, fiktiv. In der Praxis werden sich diese Daten meistens an bereits vorhandenen Gegebenheiten, wie einen Produktkatalog oder eine Preisliste, orientieren. Bleiben nur noch die Auftragsdaten, denen ebenfalls eine Tabelle spendiert wird.

AuftragsNr	KundenNr	Datum	ProduktNr	Anzahl	Rabatt	Zahlart
9001	1001	1.4.1999	5001	2	20	Kredit
9002	1002	2.4.1999	5001	12	40	Bar

usw.

Anstatt alle Daten in eine Tabelle zu packen, wurden die Daten auf drei Tabellen verteilt. Ein kleines Problem gilt es aber noch zu lösen. Die Tabelle mit den Auftragsdaten enthält keine Kundennamen mehr. Wie kann ein Datenbankprogramm herausfinden, welche Aufträge etwa der Kunde »Dieter« bereits erteilt hat? Ganz einfach, die Tabelle mit den Auftragsdaten enthält ein Feld mit dem Namen *KundenNr*, über dessen Inhalt ein Datensatz in der Tabelle *Kundendaten* ausgewählt wird. Das Feld *KundenNr* der Auftragstabelle verweist somit auf das Feld *KundenNr* der Kundenstammdatentabelle (die Namensübereinstimmung ist rein zufällig). Durch diese Relation wird eine Beziehung zwischen beiden Tabellen hergestellt. Das ist die Idee relationaler Datenbanken.

1.8.1 Die Rolle der Schlüssel

Die Felder, die für das Herstellen einer Relation zuständig sind, werden als *Schlüssel* (engl. »keys«) bezeichnet. Das Feld mit der Kundennummer in der Kundenstammdatentabelle wird als *Primärschlüssel* bezeichnet (engl. »primary key«), da es das wichtigste Feld der Datenbank ist, weil es jeden Datensatz eindeutig kennzeichnet. Primärschlüssel, die auch aus mehreren Feldern bestehen können, besitzen die sehr wichtige Eigenheit, daß ihr Inhalt pro Tabelle nur einmal vorkommen darf, was bei einer Kundennummer der Fall ist, nicht aber z.B. bei einem Nachnamen. Das Feld mit der Kundennummer in der Auftragsdatenbank wird dagegen als *Fremdschlüssel* (engl. »foreign key«) bezeichnet. Zwischen dem Primärschlüssel in der Stammdatentabelle und dem Fremdschlüssel in der Auftragstabelle besteht eine 1:n-Beziehung. Ein Kundennummerfeld in der Stammdatentabelle kann auch beliebig viele (also n) Datensätze in der Auftragstabelle anzeigen, denn ein Kunde kann beliebig viele Aufträge erteilen.

In einer relationalen Datenbank existieren zwischen den einzelnen Tabellen (in diesem Zusammenhang auch Relationen genannt werden) Beziehungen. Diese Beziehungen werden über Felder der Tabellen hergestellt[1].

Die Beziehung stellt eine logische Verbindung zwischen zwei Tabellen her, wobei eine Tabelle auch mehrere Beziehungen unterhalten kann, die aber stets unabhängig voneinander sind. In dem relationalen Modell werde drei »Beziehungstypen« unterschieden:

- ✘ Die 1:n-Beziehung
- ✘ Die 1:1-Beziehung
- ✘ Die n:m-Beziehung

Die 1:n-Beziehung haben Sie bereits kennengelernt. Sie besteht zwischen einem Primärschlüssel in einer Tabelle und mehreren Fremdschlüsseln in einer anderen Tabelle. Damit ein Feld als Primärschlüssel in Frage kommt, muß eine wichtige Voraussetzung erfüllt sein: Der Feldinhalt darf nur einmal

1 Die doppelte Bedeutung des Begriffs »Relation«, nämlich einmal als ein Name für eine Tabelle gemäß dem relationalen Modell und ein weiteres Mal als eine andere Bezeichnung für die Beziehung zwischen zwei Tabellen, wird in diesem Buch nicht sauber getrennt, da das relationale Modell aufgrund seines Umfangs und seiner »theoretischen Schwere« nur erwähnt, nicht aber formell eingeführt und beschrieben wird.

vorkommen. Dies wäre bei der Kundennummer erfüllt, da diese (im allgemeinen) pro Kunde nur einmal vergeben wird. Das Namensfeld kann dagegen nicht als Primärschlüssel verwendet werden, da Namen doppelt vorkommen können. Wird das Namensfeld dagegen mit dem Kundennummerfeld kombiniert, ergibt sich wieder eine eindeutige Kombination, so daß diese als Primärschlüssel geeignet ist.

Der Fremdschlüssel ist eine Kopie des Primärschlüssels in einer anderen Tabelle. Beide Datensätze stehen in einer Beziehung, da zwei ihrer Felder den gleichen Inhalt aufweisen.

Neben der am häufigsten vorkommenden 1:n-Beziehung gibt es noch die 1:1- und die n:m-Beziehungen. Bei einer 1:1-Beziehung verweist ein Datensatz in einer Tabelle auf genau einen anderen Datensatz in einer anderen Tabelle. Enthält die Kundenstammdatentabelle nur die Namen der Kunden und sind die Adressen in einer zweiten Tabelle abgelegt, liegt eine 1:1-Beziehung vor, da ein Datensatz in der Kundentabelle stets auf genau einen Datensatz in der Adreßtabelle verweist. (Es sein denn, ein Kunde hätte mehrere Adressen, z.B. eine Privat- und eine Geschäftsadresse. In diesem Fall liegt wieder eine 1:n-Beziehung vor.) Bei einer n:m-Beziehung verweisen mehrere Datensätze in einer Tabelle auf mehrere Datensätze in einer anderen Tabelle. Bezogen auf das Beispiel mit dem PC-Händler liegt eine solche Beziehung bei einer Artikeldatentabelle vor, da es zu jedem Artikel mehrere Lieferanten und Distributoren geben kann. Ein Distributor wird wiederum auf mehrere Artikel in der Artikeldatentabelle verweisen.

Das Herstellen solcher Relationen ist (genau wie die Aufteilung der zu speichernden Daten auf Tabellen) ein Vorgang, der zur Implementation einer (relationalen) Datenbank gehört. Es existieren eine Reihe von Regeln, die eingehalten werden müssen, damit eine Datenbank den Forderungen des relationalen Modells gerecht wird. Das stufenweise Umsetzen dieser Regel wird als *Normalisierung* der Datenbank bezeichnet. Auf das Prinzip dieser Regeln wird in Kapitel 2 eingegangen.

Es sei allerdings darauf hingewiesen, daß das relationale Datenbankmodell Anfang der siebziger Jahre von Mathematikern und nicht von Visual-Basic-Programmierern entworfen wurde. Einige der Begriffe, wie z.B. Relation oder Schlüssel, mögen daher auf Anhieb etwas merkwürdig klingen, besitzen aber den unschätzbaren Vorteil, daß sie das Modell hinreichend definieren und eine enorme Flexibilität im Umgang mit den Daten gewährleisten.

Trotz allem Komfort, den Visual Basic seit Version 6.0 auch für die Implementierung einer Datenbank zu bieten hat, eine Hilfestellung beim Entwurf der Datenbank (d.h. beim Aufteilen der zu speichernden Daten auf Tabellen

und Felder) und der Normalisierung seiner Tabellen geben Ihnen weder Visual Basic noch Microsoft Access. Sie sollten sich mit diesen Dingen sehr ausführlich beschäftigen, bevor Sie Ihre erste »große« Datenbank implementieren. Ein wichtiger Grundsatz darf nicht vergessen werden: Das »Feintuning« einer im täglichen Einsatz befindlichen Datenbank hat wenig Sinn, wenn das zugrundeliegende »Design« der Datenbank nicht stimmt. Und das bedeutet konkret: Tabellen, die den Normalisierungsregeln gehorchen, die sinnvolle Vergabe von Indices, Primär- und Fremdschlüsseln und die Verwendung von SQL-Anweisungen, die die Datenbank-Engine nicht dazu zwingen, unnötige Verarbeitungsschritte einzulegen.

1.8.2 Datenintegrität

Ein wichtiger Aspekt, der sich durch das Herstellen von Relationen ergibt, ist die Datenintegrität. Datenintegrität bedeutet, daß (per SQL) keine Operationen erlaubt sind, durch die etwa Beziehungen zwischen zwei Tabellen gestört werden. Das beste Beispiel ist sicherlich (wieder einmal) die Kundenstammdatentabelle, die auf mehrere Datensätze in der Auftragstabelle verweisen kann. Würde man einen Datensatz in der Kundenstammdatentabelle löschen, würden auf einmal alle Datensätze in der Auftragstabelle mit den Aufträgen dieses Kunden auf einen nicht mehr existierenden Datensatz in der Kundentabelle verweisen. Das darf nicht passieren, da dadurch die Integrität der Datenbank gefährdet wird.

Bei der Jet-Engine kann eingestellt werden, daß ein Löschen eines Datensatzes mit einem Primärschlüssel zu einem Laufzeitfehler führt. Weiterhin läßt sich einstellen, daß sich automatisch alle Fremdschlüssel anpassen, wenn sich der Primärschlüssel ändert. Diese automatische Anpassung, die als *referentielle Integrität* bezeichnet wird, spart dem Programmierer eine Menge Arbeit und sorgt dafür, daß die Integrität des Datenbestandes erhalten bleibt. Unterstützt ein DBMS keine referentielle Integrität, muß dieses »zu Fuß« programmiert werden.

1.8.3 Die Rolle der Indices

Ein Index ist eine sortierte Durchnumerierung aller Datensätze einer Tabelle, der außerhalb der Tabelle aber in der Regel in der Datenbank gespeichert wird. Als Sortierkriterium kann (im Prinzip) jedes beliebige Feld der Tabelle herangezogen werden. Stellen Sie sich noch einmal die Auftragstabelle vor, die unser PC-Händler einmal nach Kundennummern, ein anderes Mal nach den Auftragsdaten und ein drittes Mal nach der Höhe der Aufträge sortieren möchte. Müßte die Datenbank nach jeder Änderung der Reihenfolge jeden

Datensatz neu positionieren, wäre dies mit einem erheblichen Zeitaufwand verbunden. Indizes sind dafür eine Lösung. Jedes Feld, auf das ein Index gelegt wird, legt eine Sortierreihenfolge der einzelnen Datensätze fest, die beim Festlegen des Index getroffen wird. Damit sortiert das DBMS einmal die Tabelle nach dem Kriterium durch, das durch die Auswahl eines Index getroffen wird. Damit diese Reihenfolge bei der späteren Auswahl des Index wiederhergestellt werden kann, merkt sich das DBMS (vereinfacht ausgedrückt) die Nummern der Datensätze in der bestimmten Reihenfolge. Dadurch müssen bei Auswahl eines Index die Datensätze nicht neu sortiert werden. Statt dessen werden die Datensätze lediglich anhand der gespeicherten Reihenfolge neu aufgelistet.

Indizes haben daher in erster Linie etwas mit Performance beim Zugriff auf eine Datenbank zu tun und können bei großen Tabellen eine entscheidende Rolle spielen. Außerdem sollen sie sicherstellen, daß die Felder einer Spalte in einer Tabelle keine identischen Werte enthalten. Bei der Jet-Engine werden Indizes und Schlüssel kombiniert, d.h., beim Anlegen eines Index kann angegeben werden, ob dieser auch die Rolle eines Primärschlüssels spielen soll.

1.9 Ein Blick zurück – so war es früher

Programmierer müssen nicht erst seitdem es Windows oder Microsoft Access gibt mit Daten umgehen. Das trifft auch für Visual-Basic-Programmierer zu, denn unter Visual Basic 1.0 und seinem populären Vorgänger QuickBasic (damals noch für DOS) gab es noch keine Form der »eingebauten« Datenbankunterstützung[1]. Was tut ein Programmierer, der Daten speichern möchte, dem aber kein Datenbank-Management-System zur Verfügung steht? Er (oder sie) speichert die Daten einfach in einer Datei. Wenn es einfache, unstrukturierte Daten sind, genügt eine simple Textdatei; besitzen die Daten bereits den Aufbau eines Datensatzes, ist eine Random-Datei sinnvoller (wenngleich keine Voraussetzung). Random-Dateien sind Dateien, die eine feste Aufteilung besitzen, die durch eine Variable mit einem benutzerdefinierten Datentyp (Type-Variable) vorgegeben wird. Sie werden mit dem *Open*-Befehl geöffnet und mit dem *Close*-Befehl wieder geschlossen. Der Zugriff auf einzelne Datensätze erfolgt über die Befehle

1 Lediglich Microsoft Basic PDS 7.0 bot eine einfache ISAM-Unterstützung, die bedingt durch die Einführung von Visual Basic aber keine lange Lebensdauer hatte.

Get und *Put*. Random-Dateien gibt es schon »ewig«, vermutlich so lange, wie es Microsoft Basic gibt[1].

Zum Abschluß dieses Kapitels soll Ihnen ein kleines Visual-Basic-Programm andeutungsweise zeigen, wie sich auf der Grundlage einer Random-Datei, den »Datenbankbefehlen« *Get* und *Put* sowie ein wenig VBA-Programmcode eine einfache »Datenbankverwaltung« realisieren läßt. In dieser Datenbank, die auf einer simplen Textdatei basiert, können Sie bei vorhandenen Datensätzen deren Daten anzeigen und ändern, neue Datensätze hinzufügen und Datensätze löschen. Das gesamte Programm läuft auf jedem Windows-PC (und liefe auch unter DOS, Linux und allen anderen Betriebssystemen, für die es einen »kompatiblen« Basic-Interpreter gibt), kommt ohne Datenbankschnittstelle, Objekte und Treiber aus und belegt als kompilierte Exe-Datei nur wenige Kbyte. Bitte betrachten Sie das folgende Beispiel lediglich unter dem Aspekt, daß es »Datenbankprogrammierung« bei (Visual) Basic schon immer gegeben hat. Doch erst mit der Jet-Engine (bzw. anderen Datenbank-Engines) steht der Komfort eines richtigen DBMS zur Verfügung.

Ganz überflüssig sind die guten alten Dateizugriffe übrigens nicht. So kann es bei ADO-*Recordset*-Objekten notwendig sein, diese lokal, d.h. außerhalb der Datenbank, zu speichern. Und genau dafür können die hier vorgestellten Dateizugriffsbefehle wieder herangezogen werden (wenngleich es mit der *Save*-Methode eines *Recordset*-Objekts seit ADO 2.1 eine Alternative gibt).

Listing 1.1: Das Programmlisting der »Mikrodatenbank«

```
' ***********************************************
' Eine kleine "Mikrodatenbank" auf der Grundlage
' von Random-Dateien
' Peter Monadjemi, Markt&Technik, 1999
' ***********************************************

Option Explicit

' Hier wird ein Datensatz definiert
Private Type typAutodaten
    ModellName As String * 50
    Preis As Currency
    Geschwindigkeit As Single
    Farbe As String * 12
End Type
```

1 Für die jüngeren Leser: Microsofts erstes Produkt war ein Basic-Interpreter. Microsoft Basic war in den achtziger Jahren auf praktisch jedem Heimcomputer zu finden und wurde von IBM, zusammen mit MS-DOS, für die ersten PCs lizenziert und war dort unter den Namen BasicA und GW-Basic bekannt.

```
' Die Variable AutoDatenFeld nimmt die Datensätze auf
Private AutoDatenFeld(0 To 100) As typAutodaten
Private MaxDatenSatzNr As Integer
Private DatensatzNr As Integer
Private Updatemodus As Boolean

' Wird aufgerufen, wenn ein Hinzufügen abgebrochen
' werden soll
Private Sub cmdAbbrechen_Click()
    DatensatzNrAnzeigen
    cmdDatensatzHinzufügen.Caption = "&Hinzufügen"
    cmdAbbrechen.Enabled = False
    cmdDatensatzLöschen.Enabled = True
    Updatemodus = False
End Sub

' Wird aufgerufen, wenn ein Datensatz hinzugefügt wird
Private Sub cmdDatensatzHinzufügen_Click()
' Wenn noch kein Updatemodus aktiv, dann
' auf Updatemodus umschalten
    If Updatemodus = False Then
       lblDatensatzNr.Caption = MaxDatenSatzNr + 1
       cmdDatensatzHinzufügen.Caption = "&Aktualisieren"
       cmdAbbrechen.Enabled = True
       cmdDatensatzLöschen.Enabled = False
       EingabefelderLöschen
       Updatemodus = True
    Else
 ' Updatemodus aktiv, Datensatz an das Ende hängen
       MaxDatenSatzNr = MaxDatenSatzNr + 1
       DatensatzNr = MaxDatenSatzNr
       Datenaktualisieren
       DatensatzNrAnzeigen
       cmdDatensatzHinzufügen.Caption = "&Hinzufügen"
       cmdAbbrechen.Enabled = False
       cmdDatensatzLöschen.Enabled = True
       Updatemodus = False
    End If
End Sub

' Alle Eingabefelder löschen
Sub EingabefelderLöschen()
    txtGeschwindigkeit.Text = ""
    txtModellName.Text = ""
    txtPreis.Text = ""
    txtFarbe.Text = ""
    txtModellName.SetFocus
End Sub
```

```
' Wird aufgerufen, wenn ein Datensatz gelöscht werden soll
Private Sub cmdDatensatzLöschen_Click()
    Dim n As Integer
    If MaxDatenSatzNr > 0 Then
       For n = DatensatzNr To MaxDatenSatzNr - 1
           With AutoDatenFeld(n)
            .Geschwindigkeit = AutoDatenFeld(n + 1).Geschwindigkeit
            .ModellName = AutoDatenFeld(n + 1).ModellName
            .Preis = AutoDatenFeld(n + 1).Preis
            End With
       Next
       MaxDatenSatzNr = MaxDatenSatzNr - 1
       MovePrevious
    End If
End Sub

' Datensatzzeiger eine Position vor bewegen
Private Sub cmdVor_Click()
    Datenaktualisieren
    MoveNext
End Sub

' Datensatzzeiger eine Position zurückbewegen
Private Sub cmdZurück_Click()
    Datenaktualisieren
    MovePrevious
End Sub

' Lädt die Datenbank oder legt eine neue an
Private Sub Form_Load()
    Dim DateiNr As Integer
' Wenn noch keine Autodaten.dat-Datei vorhanden,
' 3 Datensätze initialisieren
    If Dir("Autodaten.dat") = "" Then
       With AutoDatenFeld(0)
        .ModellName = "Audio Avant"
        .Geschwindigkeit = "188"
        .Preis = "32600"
        .Farbe = "Rot"
       End With
       With AutoDatenFeld(1)
        .ModellName = "VW Golf TS"
        .Geschwindigkeit = "178"
        .Preis = "22600"
        .Farbe = "Blau"
       End With
       With AutoDatenFeld(2)
        .ModellName = "BMW 316i"
        .Geschwindigkeit = "192"
```

```
        .Preis = "28400"
        .Farbe = "Grün"
       End With
       MaxDatenSatzNr = 2
    Else
' Wenn Autodaten.dat vorhanden, dann Inhalt einlesen
       DateiNr = FreeFile
       Open "Autodaten.dat" For Random As DateiNr
            Do
               Get #DateiNr, , AutoDatenFeld(MaxDatenSatzNr)
               If EOF(DateiNr) = True Then
                  Exit Do
               End If
               MaxDatenSatzNr = MaxDatenSatzNr + 1
            Loop
       Close
    End If
    DatensatzNr = 0
    DatensatzAnzeigen DatensatzNr
    cmdAbbrechen.Enabled = False
End Sub

' Inhalt des aktuellen Datensatzes anzeigen
Sub DatensatzAnzeigen(DatensatzNr As Integer)
    With AutoDatenFeld(DatensatzNr)
      txtGeschwindigkeit.Text = .Geschwindigkeit
      txtModellName.Text = .ModellName
      txtPreis.Text = .Preis
      txtFarbe.Text = .Farbe
    End With
    DatensatzNrAnzeigen
End Sub

' Aktuelle Datensatznummer anzeigen
Sub DatensatzNrAnzeigen()
    lblDatensatzNr.Caption = DatensatzNr
End Sub
' Beim Verlassen des Formulars Datensätze in
' Autodaten.dat zurückschreiben
Private Sub Form_Unload(Cancel As Integer)
    Dim DateiNr As Integer, n As Integer
    DateiNr = FreeFile
    Datenaktualisieren
    Open "Autodaten.dat" For Random As DateiNr
       For n = 0 To MaxDatenSatzNr
          Put #DateiNr, , AutoDatenFeld(n)
       Next
    Close
End Sub
```

```
' Datensatzzeiger auf den nächsten Datensatz bewegen
Private Sub MoveNext()
    DatensatzNr = DatensatzNr + 1
    If DatensatzNr > MaxDatenSatzNr Then
       DatensatzNr = MaxDatenSatzNr
    End If
    DatensatzAnzeigen DatensatzNr
End Sub

' Datensatzzeiger auf den vorherigen Datensatz bewegen
Private Sub MovePrevious()
    DatensatzNr = DatensatzNr - 1
    If DatensatzNr < 0 Then
       DatensatzNr = 0
    End If
    DatensatzAnzeigen DatensatzNr
End Sub

' Feststellen, ob Datensatz geändert wurde und
' in Datenbank aktualisieren
Sub Datenaktualisieren()
    Dim DirtyFlag As Boolean
    DirtyFlag = txtGeschwindigkeit.DataChanged _
     Or txtModellName.DataChanged _
     Or txtPreis.DataChanged _
     Or txtFarbe.DataChanged
    If DirtyFlag = True Then
       With AutoDatenFeld(DatensatzNr)
        .Geschwindigkeit = txtGeschwindigkeit.Text
        .ModellName = txtModellName.Text
        .Preis = txtPreis.Text
        .Farbe = txtFarbe.Text
       End With
       txtGeschwindigkeit.DataChanged = False
       txtModellName.DataChanged = False
       txtPreis.DataChanged = False
       txtFarbe.DataChanged = False
    End If
End Sub

' *** Ende des Programms ***
```

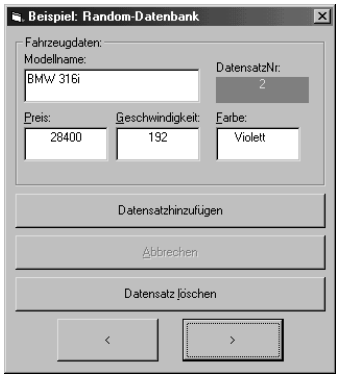


Bild 1.6: Die »Mikrodatenbank« in Aktion

Wenn Sie diese »Mikrodatenbank« einmal ausprobieren möchten, müssen Sie folgende Schritte zur Umsetzung durchführen:

1. Legen Sie ein Visual-Basic-Standard-Exe-Projekt an.
2. Ordnen Sie die Steuerelemente auf einem Formular an, wie es Bild 1.6 zu sehen ist. Die Namen der Steuerelemente ergeben sich aus ihrem Titel.
3. Fügen Sie den Programmcode aus Listing 1.1 hinzu.
4. Starten Sie das Programm.

Auch wenn die Lösung sehr einfach ist, ist sie natürlich alles andere als ideal. Vergleichen Sie einmal den Aufwand, den Umfang des Programmcodes und die fehlende Funktionalität mit einer Alternative, die auf der Jet-Engine und den ADO-Objekten basiert. Die »Mikrodatenbank« auf der Basis der Random-Dateien kann in ihrer jetzigen Form nur eine einzige Datensatzgruppe (mit vier Feldern) verwalten, bietet keine Suchfunktion (diese ließe sich mit wenig Aufwand implementieren), keine referentielle Integrität, keine automatische Bindung an Steuerelemente und keinerlei Möglichkeiten, Daten aus verschiedenen Tabellen zu verknüpfen. Zwar ließen sich alle diese Leistungsmerkmale mit purem VBA implementieren, doch der Aufwand wäre enorm. Es bleibt also alles, wie es ist. Auch wenn eine einfache Lösung, gerade für erfahrene Programmierer, attraktiver erscheinen mag, ist dies ein Trugschluß. Der Umgang mit den ADO-Objekten, OLE DB-Providern und anderen Dingen mag am Anfang unnötig kompliziert erscheinen. Doch der Lernaufwand lohnt sich, denn sobald eine Datenbankanwendung eine gewisse Größe überschreitet (und das geht sehr schnell), wird eine Komplexität erreicht, die sich mit herkömmlicher Programmierung kaum in den Griff kriegen läßt.

1.10 Wichtige Begriffe aus der Datenbankwelt

Zum Abschluß dieses Kapitels sollen Sie eine Reihe wichtiger Begriffe aus der (Windows-)Datenbankwelt kennenlernen, die aber am Anfang für Sie vermutlich nur eine geringe Bedeutung besitzen werden, da sie beim Zugriff auf eine Access-Datenbank keine oder nur indirekt eine Rolle spielen. Dennoch sind sie wichtig, da sie in Artikeln und Fachbüchern sehr häufig auftreten.

1.10.1 Die Rolle von ODBC

Der erste Begriff, den jeder Datenbankprogrammierer kennen sollte, ist ODBC, die Abkürzung für *Open Database Connectivity*. Microsoft erkannte bereits Anfang der 90er Jahre, daß zwar viele Programmierer Windows als Plattform akzeptieren, dennoch aber ihre Daten lieber auf einem Großrechner, einem Unix-Rechner oder auf einem Windows-PC mit einer bestimmten Datenbank halten. Damit Visual Basic oder ein anderes Endanwenderwerkzeug wie Excel, in dem der Datenbankzugriff ebenfalls eine wichtige Rolle spielt (Excel bot mit Ms Query schon seit frühen Versionen eine komfortable Möglichkeit der Datenbankabfrage), nicht für jeden Datenbanktyp eigene Befehle anbieten muß, hat Microsoft, in Zusammenarbeit mit anderen Datenbankherstellern, Anfang der 90er Jahre ODCB ins Leben gerufen. ODBC basiert auf einem »Baukastenprinzip«, bei dem vereinfacht formuliert der untere Teil des Baukastens »weiß«, wie eine bestimmte Datenbank angesprochen wird. Der obere Teil des Baukastens dagegen weiß nichts über die Datenbank, sondern ist lediglich in der Lage, ein Kommando (über eine Mittelschicht) an den unteren Teil zwecks Bearbeitung weiterzureichen. Visual-Basic-Programmierer sprechen über die ODBC-API-Funktionen nur den oberen Teil an und sind idealerweise in der Lage, alle Datenbanken ansprechen zu können, ohne Änderungen am Programm vornehmen zu müssen. Voraussetzung ist, daß für die Datenbank ein ODBC-Treiber existiert.

OBDC steht für Open Database Connectivity *und soll dafür sorgen, daß ein Visual-Basic-Programm (per SQL) auf eine Datenbank zugreifen kann, ohne deren spezifische Besonderheiten zu kennen. Das setzt voraus, daß für die Datenbank ein ODBC-Treiber existiert. Allerdings ist ODBC in erster Linie für relationale Datenbanken gemacht und an SQL als Abfragesprache gebunden. OLE DB besitzt als »Nachfolger« von ODBC diese Einschränkung nicht.*

Für einen Visual-Basic-Programmierer stehen alle Funktionen des ODBC-Treibers in Gestalt von API-Funktionen zur Verfügung. Da diese Programmierung aber ein wenig umständlich ist (und bis Visual Basic 4.0 auch keine asynchrone Abarbeitung von SQL-Abfragen zuließ, bei der die Datenbank nach Beendigung der Abfrage eine Callback-Funktion hätte aufrufen müssen, was wiederum ohne *AddressOf*-Operator nicht möglich ist), wurden mit Visual Basic 4.0 Enterprise Edition die *Remote Data Objects* (RDOs) eingeführt. Die RDOs bieten den Komfort der DAOs kombiniert mit der Performance eines direkten ODBC-Zugriffs.

Interessant ist das Prinzip, wie mit ODBC eine Datenbank ausgewählt wird, da dies auch in Grundzügen von OLE DB/ADO übernommen wurde. Anstelle des Namens einer Datenbank möchte Visual Basic bei ODBC eine Verbindungszeichenfolge (engl. »connect string«) wissen. Hier ein kleines Beispiel für den Zugriff auf eine Datenbank über ODBC und die *OpenDatabase*-Methode des *DBEngine*-Objekts bei ADO:

```
Dim Db As Database
Set Db = DBEngine.OpenDatabase("Biblio")
```

Anstelle eines konkreten Datenbanknamens mit genauer Pfadangabe wird bei diesem Aufruf lediglich ein sogenannter *Data Source Name*, kurz DSN, übergeben. Visual Basic erkennt beim Aufruf, daß es sich bei Biblio nicht um einen Datenbanknamen, sondern um einen DSN handelt. Der formell etwas korrektere Aufruf:

```
Set Db = DBEngine.OpenDatabase(Name:="Biblio", _
Options:=dbDriverNoPrompt, Connect:="ODBC;")
```

macht die Angelegenheit zwar etwas klarer, ist aber bei Visual Basic nicht notwendig. Doch was muß man sich unter einem DSN vorstellen? Ein DSN ist nichts anderes als eine Zusammenfassung aller Angaben, die für den Zugriff auf eine Datenbank benötigt werden. Zu diesen Angaben gehört der Name oder Verzeichnispfad der Datenbank, aber auch das für den Zugriff benötigte Kennwort. Alle diese Angaben werden unter einem Namen zusammengefaßt, dem DSN. Ein DSN wird entweder innerhalb des Programms oder vom ODBC-Manager in der Windows-Systemsteuerung angelegt. Dieser speichert die angegebenen Informationen in der Registry (früher unter Windows 3.1 noch in der *ODBC.INI*-Datei). Der DSN muß mindestens eine Angabe enthalten, den Namen der Datenbank. Bezogen auf das obige Beispiel wäre es die Access-Datenbankdatei *Biblio.mdb*. Und da wir gerade dabei sind. Am letzten Beispiel haben Sie das Öffnen einer ODBC-Datenbank mit DAO kennengelernt, so sieht es dagegen unter ADO aus:

```
Dim Cn As ADODB.Connection
Set Cn = New ADODB.Connection
Cn.Open ConnectionString:="DSN=Biblio"
```

Diese Befehlsfolge öffnet eine sogenannte Verbindung zu der Datenquelle, die über den DSN *Biblio* definiert wird. Sie sehen, der Unterschied zwischen DAO und ADO ist gar nicht so groß, nur daß bei ADO ein *Connection*-Objekt resultiert, bei DAO ein *Database*-Objekt.

OLE DB und ADO verwenden DSNs nur aus Kompatibilitätsgründen (nämlich wenn eine Datenbank über einen ODBC-Treiber angesprochen werden soll). Bei OLE DB/ADO wird der DSN durch ein sogenanntes *Data Link* ersetzt. Hierbei handelt es sich entweder um eine Zeichenfolge oder um eine Datei mit der Erweiterung *.Udl*, die im wesentlichen jene Informationen enthalten, die bei ODBC in einem DSN gespeichert werden. Anstelle eines DSN wird einfach die Verbindungszeichenfolge oder der Name einer UDL-Datei angegeben:

```
Cn.Open ConnectionString:="FILE NAME=C:\Biblio.udl"
```

Ein Nachteil dieses Verfahrens soll allerdings nicht verschwiegen werden. Wird die Datenbank in einem anderen Verzeichnis abgelegt, muß auch die UDL-Datei angepaßt werden. Sonst kann Visual Basic die Datenquelle nicht finden.

Bei Visual Basic 6.0 spielen die Data Links übrigens keine so wichtige Rolle, da es bequemer ist, eine Datenumgebung anzulegen. Wie Sie ein Data Link anlegen und dieses für den Zugriff auf eine Access-Datenbank benutzen, wird in Kapitel 3.9 besprochen.

In der Einleitung wurde es bereits angedeutet: ODBC wird für Sie vermutlich keine Rolle mehr spielen. Wenn Sie sich für die Jet-Engine als DBMS entscheiden, was am Anfang naheliegend ist, benötigen Sie kein ODBC (wenngleich man per ODBC auch auf die Jet-Engine zugreifen kann, etwa um den Datenbankzugriff einheitlich zu gestalten). Wenn Sie dagegen auf einen Microsoft SQL-Server oder auf einen Oracle-Server zugreifen möchten, gibt es mit OLE DB eine in den meisten Situationen bessere, weil flexiblere Alternative[1]. ODBC und ADO sind daher in erster Linie für die Visual-Basic-Programmierer interessant, die bereits Projekte mit ODBC

1 Bezogen auf Oracle muß dies aber mit Einschränkungen gesagt werden, da der Microsoft OLE DB-Provider nicht alle Merkmale von Oracle anbietet. Hier kann ein ODBC-Treiber zur Zeit noch eine bessere Funktionalität bieten.

angefangen haben, oder für deren Datenbank es keine OLE DB-Treiber gibt[1].

1.10.2 Die Rolle von OLE DB

Die Zukunft der Microsoft-Datenbankstrategie heißt OLE DB. Moment, heißt die Zukunft nicht ADO oder UDA? Bringt jetzt etwa auch der Autor alles durcheinander? Zum Glück nicht, denn einer muß halbwegs die Übersicht behalten. Hier ist die Auflösung: Die Microsoft-Datenbankstrategie wird unter dem Namen *Universal Data Access* (also UDA) zusammengefaßt und basiert vollständig auf OLE DB. Um aber von VBA auf OLE DB zugreifen zu können, wird ADO benötigt, da dies bei VBA (anders als etwa bei C++ oder Java) nicht direkt geht.

Bliebe noch zu klären, was OLE DB ist, bei dem es sich ausnahmsweise nicht um eine Abkürzung handelt[2]. OLE DB verspricht, wie ODBC, einen weitestgehend universellen Datenbankzugriff. Konkret, ein Visual-Basic-Programmierer ist dank der ADOs immer unabhängig von einer Datenquelle. Arbeitet ein Programm mit einer Access-Datenbank als Datenquelle und sollen zusätzlich Daten angesprochen werden, die sich auf einem Oracle Server oder in einem HTML-Dokument befinden, so muß lediglich ein anderer OLE DB-Provider und eine andere Datenquelle ausgewählt werden, das Programm muß dagegen (im Idealfall) nicht angepaßt werden. OLE DB erweitert das Konzept von ODBC, indem es von SQL als Abfragesprache und von relationalen Datenbanken als Datenquelle unabhängig ist. OLE DB setzt weder eine Abfragesprache, noch eine bestimmte Anordnung der Daten (etwa eine Tabellenstruktur) voraus. Alles, was Daten enthält, soll per OLE DB und ADO in einheitlicher Weise angesprochen werden können. Wie das am Beispiel einer Access-Datenbank in der Praxis aussieht, wird ab Kapitel 4 ausführlich gezeigt.

1.10.3 Lokale Datenbanken und Remote-Datenbanken

Für angehende Visual-Basic-Datenbankprrogrammierer ist es selbstverständlich, daß sich die Datenbank auf dem gleichen PC befindet. Diese soge-

---

1 Was in Anbetracht der Tatsache, daß die Firma Intersolv (die 1999 mit der Firma MicroFocus unter dem neuen Namen Merant fusionierte) praktisch für alle gängigen Datenbanken OLE DB-Treiber anbieten wird bzw. will, relativ unwahrscheinlich ist. Infos zu OLE DB und DataDirect Connect 2.0 von Intersolv/Merant gibt es unter *www.merant.com/datadirect/products/OLE DB/Connect/overview.asp*.

2 Ganz richtig ist das natürlich nicht, doch die Hintergründe und die Frage, warum OLE OLE heißt, sind relativ uninteressant. Nur soviel: DB steht natürlich für Database.

nannten lokalen Datenbanken sind jedoch als typisches PC-Produkt in der »richtigen« Datenbankwelt eine echte Ausnahme. Hier liegt die Datenbank auf einem speziell dafür vorgesehenen Computer, bei dem es sich häufig um einen PC handeln wird, genauso oft aber auch um einen Unix-Rechner oder um einen Großrechner. Da sich diese Datenbanken in einer gewissen »Entfernung« von dem PC befinden, der auf die Datenbank zugreifen möchte, werden sie als *Remote-Datenbanken* bezeichnet. Da Remote-Datenbanken nicht als Dateien vorliegen, deren Netzwerkpfad man vor dem Öffnen der Datenbank angeben könnte, sondern lediglich als Namen existieren, muß es für den Zugriff auf Remote-Datenbanken auch bestimmte Mechanismen geben. Diese Mechanismen gibt es bei OLE DB mit den Datalink-Dateien (mehr dazu in Kapitel 3.9) und bei ODBC mit den *Data Source Names* und dem ODBC32-Administrator in der Systemsteuerung.

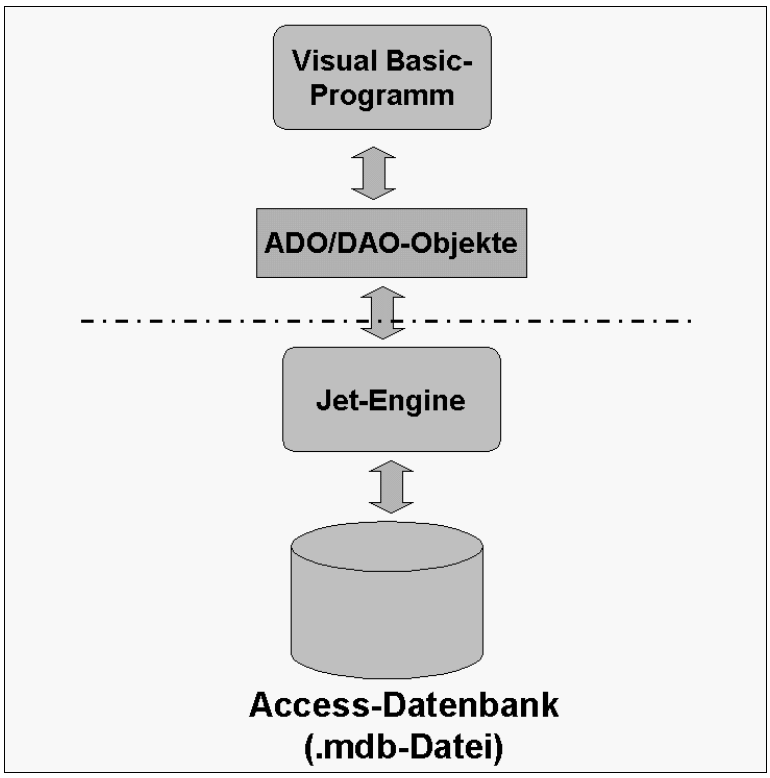


Bild 1.7: Das Zugriffsprinzip auf eine (lokale) Access-Datenbank

1.10.4 Das Client/Server-Prinzip

Einen wichtigen Begriff müssen Sie noch kennenlernen, dann haben Sie es geschafft (zumindest was die Begrifflichkeiten angeht). Die Rede ist von dem *Client/Server-Prinzip*, das in der modernen DV-Welt eine zentrale Rolle spielt. Der Begriff Client/Server ist ein allgemeiner Begriff, der auf eine Vielzahl von Situationen angewendet werden kann. Wann immer eine Komponente A mit einer Komponente B, die nicht im gleichen Adreßraum des Prozessors läuft, kommuniziert, liegt das Client/Server-Prinzip vor. Die mit Abstand wichtigste Bedeutung hat der Begriff in der Datenbankwelt. Er beschreibt jenes allgemeine Szenario, in dem ein DBMS mit anderen Anwendungen, etwa einem Visual-Basic-Programm, in einem Netzwerk (das ist allerdings keine Bedingung) zusammenarbeitet. Das DBMS ist der Server, weil es anderen etwas zur Verfügung stellt (es »serviert« die Daten und damit verbunden bestimmte Datenbankdienste), die Anwendung ist dagegen der Client, weil sie die Dienste des Servers in Anspruch nimmt. Die Attribute Server und Client sind nicht feststehend. Wenn etwa das DBMS die Dienste eines anderen Programms, etwa eines Backup-Programms, in Anspruch nimmt, wird das DBMS zum Client, und das Backup-Programm spielt den Server.

Der Begriff Client/Server beschreibt ein Szenario, in dem ein DBMS (Server) einer beliebigen Anwendung (Client) Daten und Dienste zur Verfügung stellt.

Für die Datenbankprogrammierung mit Visual Basic ist das Client/Server-Prinzip insofern von (großer) Bedeutung, als sich mit der Enterprise Edition von Visual Basic sehr gut Client/Server-Anwendungen erstellen lassen. Der im Paket enthaltene Microsoft SQL Server übernimmt die Rolle des Servers, das Visual-Basic-Programm die Rolle des Clients. Handelt es sich um eine sehr moderne Anwendung, wird die Mittelschicht durch ActiveX-DLLs, die unter Umständen vom Microsoft Transaction Server verwaltet werden, gebildet. Visual Basic ist spätestens seit der Version 6.0 ein echtes Client/Server-Entwicklungswerkzeug.

Microsoft SQL-Server ist der Name des »großen« Datenbank-Management-Systems von Microsoft. Die aktuelle Version lautet 7.0. Visual Basic enthält in der Version 6.0 die Version 6.5 des SQL-Servers, so daß sich echte Client/Server-Anwendungen erstellen lassen.

In diesem Buch spielt das Client/Server-Prinzip dagegen keine Rolle. Die in Kapitel 3 vorgestellte Access-Datenbank wird lokal gehalten, die kleinen Visual-Basic-Beispielprogramme greifen direkt auf diese Datenbank zu. Alles spielt sich daher auf ein und demselben PC ab (wenngleich das Netzwerk zwar keine zwingende Voraussetzung für Client/Server ist, in der Praxis jedoch ist). Doch da der Zugriff über ADO erfolgt, das genauso gut mit einer Remote-Datenbank, wie dem SQL-Server, klar kommt, würde sich bei einer Client/Server-Anwendung am Prinzip des Datenbankzugriffs nicht viel ändern. Indem Sie ADO lernen, legen Sie damit den Grundstein für künftige Client/Server-Anwendungen.

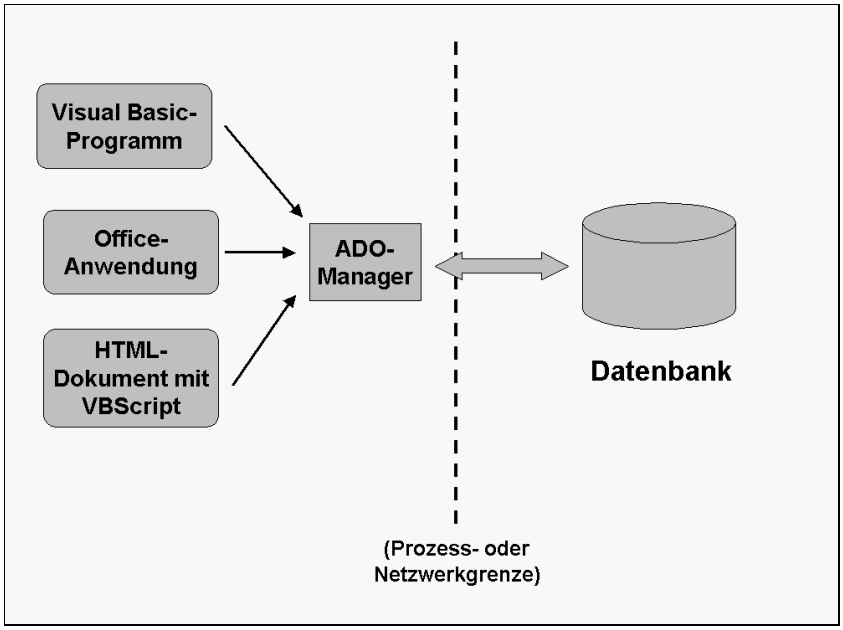


Bild 1.8: Beim Client/Server-Prinzip arbeitet ein Datenbankserver mit verschiedenen (Visual-Basic-) Clients zusammen

1.11 Zusammenfassung

Alle in einer Datenbank gespeicherten Daten sind in Form von *Tabellen* organisiert. Eine *Tabelle* besitzt einen sehr einfachen Aufbau. Sie besteht aus einer Reihe von Zeilen, die in Spalten unterteilt sind. Jede Zeile besitzt eine feste Anzahl von Spalten, wobei jede Spalte eine feste Größe besitzt (die durch den Datentyp ihres Inhalts bestimmt wird).

Für den Zugriff auf eine Datenbank bietet Visual Basic mit ADO, DAO und RDO gleich drei Objektschnittstellen an, von denen Sie ADO sehr ausführlich und DAO in einer Zusammenfassung in Anhang C kennenlernen werden.

1.12 Wie geht es weiter?

In diesem Kapitel haben Sie das Einmaleins der Datenbankprogrammierung unter Visual Basic und vor allem jene Begriffe kennengelernt, über die Sie im weiteren Verlauf des Buches noch manches Mal »stolpern« werden. Falls Sie in Zukunft wissen möchten, was man unter einer Tabelle, ADO, ODBC oder einer Client/Server-Anwendung versteht, schlagen Sie entweder in diesem Kapitel oder im Anhang A nach. In Kapitel 2 geht es um den allgemeinen Aufbau einer Datenbank. Am Beispiel der Fuhrpark-Datenbank, die sich wie ein roter Faden durch dieses Buch zieht, werden die wichtigsten Grundsätze, wie die Aufteilung in Tabellen, das Herstellen von Relationen und das Anlegen von Indizes ausführlicher erläutert. In Kapitel 3 ist die Praxis an der Reihe. Hier wird die Fuhrpark-Datenbank in Gestalt einer Access-Datenbank Schritt für Schritt umgesetzt.

1.13 Fragen

Frage 1:

Definieren Sie den Begriff Datenbank in einem Satz.

Frage 2:

Nennen Sie die Abkürzungen für die drei Datenbankobjektschnittstellen, die unter Visual Basic zur Verfügung stehen.

Frage 3:

Was ist der wichtigste Unterschied zwischen ADO und DAO?

Frage 4:

Auf welche Weise erfährt ein Visual-Basic-Programm, das über OLE DB auf eine Datenbank zugreift, wo sich diese Datenbank befindet?

Frage 5:

Welche Vorteile bringt das Client/Server-Prinzip beim Datenbankzugriff?

Frage 6:

Nennen Sie ein (beliebiges) Client/Server-Datenbank-Management-System.

Die Auflösungen zu den Übungsaufgaben finden Sie in Anhang D.

Kapitel 2

Datenbankdesign für Visual-Basic-Programmierer

Eine Datenbank besteht, das wissen Sie aus Kapitel 1, aus Tabellen, Datensätzen und Feldern. Neben diesen »Rohdaten« werden in einer Datenbank auch spezielle Elemente, wie eingebaute Abfragen, Sicherheits- und Zugriffsinformationen und vieles mehr gespeichert. Bei einer Access-Datenbank gehören dazu auch Formulare, Makros und VBA-Programmcode. Die Daten, auf die es in erster Linie ankommt, sind auf eine oder mehrere Tabellen verteilt, wobei zwischen einzelnen Feldern verschiedener Tabellen Beziehungen (Relationen) bestehen können (aber nicht müssen). Dadurch ergibt sich die Struktur der (relationalen) Datenbank. Allerdings entstehen Datenbanken nicht von alleine und schon gar nicht »per Knopfdruck«. Der Aufbau der Datenbankstruktur ist ein Vorgang, der Erfahrung und gewisse Grundkenntnisse über die Theorie relationaler Datenbanken erfordert. Das gilt auch für Access-Datenbanken. In diesem Kapitel lernen Sie am Beispiel der Fuhrpark-Datenbank, der Beispieldatenbank des Buches, die in Kapitel 3 umgesetzt wird, nicht nur den allgemeinen Aufbau einer Datenbank an einem konkreten Beispiel kennen, sondern auch jene Grundüberlegungen, die zum Aufteilen der Daten auf verschiedene Tabellen geführt haben.

Sie lernen in diesem Kapitel etwas zu folgenden Themen:

- ✘ Der Entwurf einer Datenbank
- ✘ Die Rolle von Tabellen und Feldern
- ✘ Die Umsetzung einer Datenbank
- ✘ Die Normalisierung einer Datenbank
- ✘ Die Aufteilung der Daten auf verschiedene Tabellen

- ✘ Der Datentyp eines Feldes
- ✘ Die Rolle der Schlüssel
- ✘ Die Organisation einer Access-Datenbank

2.1 Allgemeine Überlegungen zum Datenbankdesign

Anwenderprogramme, wie Microsoft Access, bei der auch unerfahrene Anwender unter Mithilfe komfortabler Assistenten in »wenigen Minuten« zu ihrer ersten Datenbank kommen, verdecken den Umstand, daß der Implementation einer Datenbank eine ausführliche Planungsphase vorausgehen sollte und in der Praxis auch vorausgehen muß. Insbesondere dann, wenn es sich nicht um eine kleine Adreßverwaltung handelt, sondern um eine Datenbank, die im Praxisbetrieb mehrere tausend oder hunderttausend (was im Gesamtvergleich noch in die Kategorie »kleine Datenbank« fällt) Datensätze umfassen soll, und mit der täglich mehrere (oder gar viele) Anwender gleichzeitig arbeiten sollen. Ein schlechtes Datenbankdesign kann nicht nur zu einer schlechten Performance (etwa bei der Zugriffsgeschwindigkeit bei Abfragen) führen, sondern auch zu einem erhöhten Programmieraufwand und im ungünstigsten Fall zu Datenverlusten, etwa wenn es einem Anwender gestattet wird, Datensätze in einer Tabelle zu löschen, auf die sich Datensätze in anderen Tabellen beziehen. Und das nicht nur einmal, sondern im gesamten »Lebenslauf« der Datenbankanwendung. Doch was bedeutet gutes Datenbankdesign konkret, und wie läßt es sich erlernen? Der Begriff *Datenbankdesign* steht allgemein für jene Überlegungen, die der Implementation (also der konkreten Umsetzung) einer Datenbank vorausgehen. Hier ein Beispiel: Sie möchten einen kleinen (Internet-)Shop betreiben und müssen sich überlegen, auf welche Weise die Produktdaten, die Preislisten, die Daten der Kunden, Bestellungen, Rechnungen, Lieferantendaten und vieles mehr in die Datenbank eingebaut werden. Für das Datenbankdesign existieren eine Reihe von allgemeinen Regeln, die im nächsten Abschnitt erläutert werden. Da es in der realen Welt viele unterschiedliche »Datenbankszenarios« gibt und keine Regel sämtliche Fälle abdecken kann, werden die »Lehrbuchregeln« durch Erfahrungswerte ergänzt. Und damit wären wir gleich beim nächsten Punkt: Datenbankdesign setzt sowohl theoretische als auch praktische Fähigkeiten voraus. Konkret, es gibt viele Lehrbücher, in denen das relationale Datenbankmodell, das die Grundlage für die überwiegende Mehrheit der Datenbanken darstellt, ausführlich und in allen theoretischen Variationen beschrieben wird. Doch ohne Praxiserfahrung lassen sich diese Regeln nicht effektiv umsetzen. Ja mehr noch, nicht alle Regeln sollten

unter bestimmten Situationen angewendet werden. Das reine Lehrbuchwissen, so wichtig es ist, stellt im »harten Datenbankalltag« nur die sprichwörtliche halbe Miete dar. Ein guter Datenbankdesigner kennt die Theorie und besitzt idealerweise eine mehrjährige Erfahrung bei der Implementation von Datenbanken. Dabei spielt es gar keine so große Rolle, ob es sich um »kleine« Access-Datenbanken oder die großen SQL-Server-Datenbanken handelt. Die Grundregeln des Datenbankdesigns gelten sowohl im Kleinen als auch im Großen[1].

Der Umsetzung einer Datenbank sollte eine möglichst ausführliche Planungsphase vorausgehen. Die einzelnen Schritte werden unter dem Begriff Datenbankdesign *zusammengefaßt. Es ist meistens keine gute Idee, Planung und Umsetzung einer Datenbank in einem Schritt durchzuführen, auch wenn Microsoft Access dazu geradezu einlädt.*

Leider gibt es auch gleich eine schlechte Nachricht: Hilfsmittel, die nach bestimmten Vorgaben im Stile eines Assistenten das Design einer Datenbank übernehmen, bieten weder Microsoft Access noch die meisten der übrigen Datenbank-Management-Systeme im PC-Bereich. Dieses Know-how muß der angehende Datenbankprogrammierer entweder bereits mitbringen oder sich durch »learning by doing« und mehr oder weniger regelmäßige Ausflüge in die Datenbanktheorie aneignen. Es gibt allerdings auf dem Markt eine Reihe von Hilfsmitteln, mit denen sich der Entwurf eines Datenbankmodells unabhängig von einer bestimmten Datenbank und damit verbunden auch die Implementierung der Datenbank vereinfacht. Diese Hilfsmittel werden als Modellierungswerkzeuge oder CASE-Tools (»Computer Aided Software Engineering«) bezeichnet und sind in der Regel relativ teuer. Ein populäres Datenmodellierungswerkzeug ist *ERwin* von der Firma *LogicWorks*, das praktisch alle gängigen Datenbanken, darunter auch Microsoft Access, unterstützt[2]. Mit ERwin erstellt man ein Datenbankmodell mit Hilfe von Symbolen für Tabellen, Beziehungen usw. Ist das Modell fertig, macht das Programm daraus auf Wunsch auch eine Datenbank, indem es Tabellen anlegt, Felder einfügt und Relationen herstellt. Leider sind Erwin&Co nicht ganz billig, so daß diese nützlichen Helfer nur für Entwicklungsteams in größeren Unternehmen in Frage kommen dürften.

1 Wobei jene Anwender, die 400-Mbyte-Datenbanken mit Microsoft Access verwalten, nicht unbedingt typisch sind.

2 Der Programmname steht nicht etwa für den Namen des Chefprogrammierers, sondern vermutlich für »Entity Relationship unter Windows«. Infos gibt es im Internet unter: www.platinum.com.

Die Datenbankfirma Sybase bietet ihren PowerDesigner 6.1 im Internet unter der Adresse *www.sybase.com/products/powerdesigner* im Rahmen einer 30-Tage-Lizenz zum kostenlosen Download an. Damit lassen sich u.a. auch Datenbankmodelle aus bereits vorhandenen Access-Datenbanken erstellen. Wer ein Datenmodellierungswerkzeug unverbindlich kennenlernen möchte, sollte den PowerDesigner einmal ausprobieren.

2.2 Regeln für den Entwurf einer Datenbank

In diesem Abschnitt lernen Sie einige der grundlegenden Regeln kennen, die beim Entwurf einer Datenbank eine Rolle spielen. Es geht dabei vor allem um die sogenannte *Normalisierung* einer Datenbank. Aus Platzgründen und weil in diesem Buch die Datenbankprogrammierung mit Visual Basic im Vordergrund steht, werden lediglich die wichtigsten Grundregeln vorgestellt[1]. Jeder angehende Datenbankprogrammierer, der vor der Umsetzung einer größeren Datenbank steht, sollte sich mit diesem Thema aus den erwähnten Gründen möglichst ausführlich beschäftigen.

Unter der Normalisierung einer Tabelle versteht man das Anwenden bestimmter Regeln, so daß die Tabelle bezüglich ihrer Felder den allgemeinen Anforderungen relationaler Datenbanken genügt. Ziel der Normalisierung ist es, redundante Daten zu entfernen und zu einem optimalen Aufbau der einzelnen Tabellen zu kommen.

2.2.1 Von der Idee zum Datenbankdesign

In diesem Buches wird eine kleine Access-97-Datenbank mit dem Namen »Fuhrpark« vorgestellt, die in Gestalt der Datei *Fuhrpark.mdb* vorliegt. Diese Datenbank enthält eine Reihe von Tabellen, die wiederum aus Feldern bestehen. Ähnliche Datenbanken (allerdings mit mehr Tabellen und Abfragen und vor allem größeren Datenbeständen) finden Sie übrigens im Visual-Basic-Verzeichnis in Gestalt der Datenbanken *Biblio.mdb* und *Northwind.mdb*. Allen diesen Datenbanken ist gemeinsam, daß ihre Struktur nicht zufällig entstanden ist, sondern auf gewissen Überlegungen basiert.

1 In den Literaturhinweisen in Anhang C finden Sie weitere Informationen.

Ihnen liegt ein Datenbankdesign zugrunde, das mehr oder wenig gründlich durchdacht wurde. Wie aus einer vagen Idee oder Vorgabe ein Datenbankdesign wird, soll im folgenden erläutert werden.

2.2.2 Aller Anfang ist (nicht) schwer

Wie es in Kapitel 3 noch ausführlicher erläutert wird, soll die Fuhrpark-Datenbank den Wagenpark eines mittelgroßen Unternehmens verwalten, der den Mitarbeitern für die verschiedenen Aktivitäten zur Verfügung steht. Insgesamt enthält der Fuhrpark mehrere Dutzend Fahrzeuge, die regelmäßig entliehen werden. Die Datenbank soll daher nicht nur den aktuellen Fahrzeugbestand, sondern auch die Entleihaktivitäten der Mitarbeiter erfassen. Soviel zur Ausgangssituation. Wie bei der Lösung eines Programmierproblems geht es auch bei der Lösung eines Datenbankdesignproblems darum, den ersten Schritt zu machen. Dieser besteht in der Regel darin, die anfallenden Daten auf ein Blatt Papier zu schreiben und sie dabei bereits in Gruppen einzuteilen, d.h. eine Vorstrukturierung vorzunehmen. Folgende Daten fallen bei der Fuhrpark-Verwaltung an:

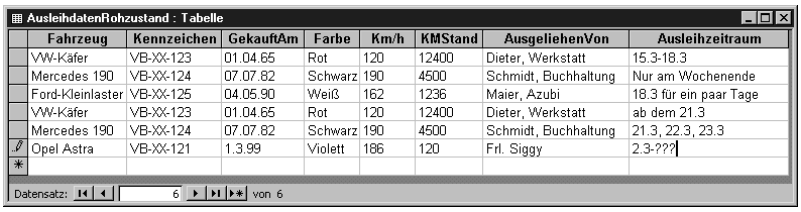
- ✘ Die Daten eines im Fuhrpark vorhandenen Fahrzeugs (z.B. Preis, Anschaffungsdatum, Kfz-Kennzeichen, Farbe, Inspektionstermine usw.)
- ✘ Die technischen Daten der Fahrzeugmodelle (z.B. Modellname, Leistung, Hubraum, Geschwindigkeit, Verbrauch usw.)
- ✘ Die Daten eines Entleihvorgangs (z.B. Nummer des ausgeliehenen Fahrzeugs, Name des Mitarbeiters, Entleihdatum, Rückgabedatum usw.)

Weitergehende Daten, wie z.B. die Daten eines einzelnen Mitarbeiters (etwa Abteilung, Dienstgrad oder Eintrittsdatum in das Unternehmen) oder finanzielle Aspekte (etwa Leasingraten und Abschreibung) werden in der Datenbank nicht berücksichtigt. Auch werden aus Gründen der Übersichtlichkeit nicht sämtliche Daten erfaßt. Wer die Datenbank zur Verwaltung eines realen Fuhrparks einsetzen möchte, muß also noch »ein paar« Felder und gegebenenfalls auch die eine oder andere Tabelle hinzufügen.

Die im folgenden beschriebene Umwandlung eines »ersten Entwurfs« der Fuhrpark-Datenbank in eine normalisierte Form wird aus Gründen der Übersichtlichkeit und besseren Anschaulichkeit nicht 100% konsistent durchgeführt. Das bedeutet konkret, daß in einigen Beispielen Felder eingeführt werden, die in der nächsten Ausbaustufe nicht mehr dabei sind. Die »finale« Fuhrpark-Datenbank mit einem verbindlichen Aufbau wird in Kapitel 3 vorgestellt.

2.2.3 Ein erster Entwurf für eine Tabelle

Daß beim Datenbankentwurf der erste Ansatz nicht immer der beste sein muß, soll der folgende Versuch deutlich machen. Anstatt die zu verwaltenden Daten auf verschiedene Tabellen zu verteilen, werden sie zunächst alle in eine Tabelle gepackt. Das Ergebnis ist eine Tabelle mit dem Namen *Ausleihdaten,* wie sie in Bild 2.1 zu sehen ist. Das Ziel der Tabelle ist es, die mit einem entliehenen Fahrzeug anfallenden Daten (möglichst vollständig) zu erfassen und nicht etwa den Fahrzeugbestand oder die Daten der Mitarbeiter, die einen Wagen entliehen haben. Diese Vorgabe ist sehr wichtig, denn daraus resultiert automatisch eine bestimmte Aufteilung der Tabelle.


AusleihdatenRohzustand : Tabelle

Fahrzeug	Kennzeichen	GekauftAm	Farbe	Km/h	KMStand	AusgeliehenVon	Ausleihzeitraum
VW-Käfer	VB-XX-123	01.04.65	Rot	120	12400	Dieter, Werkstatt	15.3-18.3
Mercedes 190	VB-XX-124	07.07.82	Schwarz	190	4500	Schmidt, Buchhaltung	Nur am Wochenende
Ford-Kleinlaster	VB-XX-125	04.05.90	Weiß	162	1236	Maier, Azubi	18.3 für ein paar Tage
VW-Käfer	VB-XX-123	01.04.65	Rot	120	12400	Dieter, Werkstatt	ab dem 21.3
Mercedes 190	VB-XX-124	07.07.82	Schwarz	190	4500	Schmidt, Buchhaltung	21.3, 22.3, 23.3
Opel Astra	VB-XX-121	1.3.99	Violett	186	120	Frl. Siggy	2.3-???

Datensatz: 6 von 6

Bild 2.1: Der erste Datenbankentwurf ist nicht unbedingt als gelungen zu bezeichnen

Die in Bild 2.1 dargestellte Tabellenstruktur ist natürlich bewußt »mißlungen«. So sollte eine Tabelle nicht strukturiert sein, auch wenn Microsoft Access nichts dagegen hat und ein solcher Ansatz in der Praxis funktionieren mag. Folgende »Designfehler« sind in der Tabelle enthalten:

- ✘ Die Tabelle weist eine unnötige Redundanz auf. Für jedes Fahrzeug werden die stets gleichbleibenden Fahrzeugdaten (in diesem Fall das Kennzeichen, das Kaufdatum, die Farbe und die Geschwindigkeit) erneut gespeichert. Dieses Problem wird beim Überführen der Tabelle in die 2. Normalform behoben.
- ✘ Das Feld *Ausleihzeitraum* enthält keine einheitlichen Werte.
- ✘ Das Feld *AusgeliehenVon* ist nicht »atomar«. Dieses Problem wird beim Überführen der Tabelle in die 1. Normalform behoben.
- ✘ Die Tabelle besitzt keinen Schlüssel, über die ein Datensatz eindeutig identifiziert werden kann (es kann im Fahrzeugbestand z.B. mehrere »VW-Käfer« geben).

Auch wenn der erste Tabellenentwurf aus didaktischen Gründen alles andere als optimal gewählt wurde, läßt sich aus diesem Ansatz eine wichtige Lehre ziehen: Eine spontane Idee ist nur selten eine Grundlage für ein gutes Datenbankdesign. Insbesondere dann nicht, wenn die Erfahrung fehlt. Aus

diesem Grund gibt es beim Datenbankdesign eine Reihe von Regeln, die im folgenden vorgestellt werden sollen.

2.2.4 Felder müssen einen einheitlichen Inhalt besitzen

Diese Regel ist im Grunde selbstverständlich. Kein Programmierer würde auf die Idee kommen, in ein und derselben Variablen einmal einen Vornamen und drei Zeilen später eine Telefonnummer abzulegen (auch wenn es programmtechnisch bei Verwendung von *Variant*-Datentypen möglich ist und es in Ausnahmefällen dafür auch Gründe geben mag). Genau wie eine Variable nur einen bestimmten Typ von Werten erhält, sollte (bzw. muß) auch ein Datenbankfeld stets einen einheitlichen Inhalt besitzen. Die Tabelle in Bild 2.1 verstößt gegen diese Regel, indem das Feld *Ausleihzeitraum* einmal ein Datumsbereich, ein anderes Mal nur ein Datum (wenn die Rückgabe noch nicht erfolgte) und ein drittes Mal lediglich eine vage Angabe enthält. Angaben wie »Nur am Wochenende« haben in einer Datenbank im allgemeinen nichts zu suchen und eignen sich höchstens als Anmerkungen für ein Memo-Feld.

2.2.5 Felder sollten einen Inhalt besitzen

Wird ein Datenbankfeld nicht mit einem Inhalt gefüllt, erhält es vom DBMS automatisch den Spezialwert *NULL* (siehe Kapitel 2.3.2). *NULL*-Werte sind zwar nicht verboten, sollten aber nach Möglichkeit vermieden werden. Die Jet-Engine bietet die Möglichkeit, bei jedem einzelnen Feld eine Option einzustellen, die verhindert, daß das Feld einen *NULL*-Wert erhalten kann. Wird der Datensatz später aktualisiert und erhält das Feld keinen Wert, ist ein Laufzeitfehler die Folge, und der Datensatz kann zunächst nicht aktualisiert werden. Gleichzeitig kann ein Standardwert festgelegt werden, der immer dann verwendet wird, wenn dem Feld kein Wert zugewiesen wurde. Diese Lösung ist nicht immer optimal, da in einigen Fällen der Anwender entscheiden soll, welchen Standardwert ein Feld erhält. Bei Datumsangaben ist es üblich, das Datum »1.1.9999« (oder ein ähnliches Datum) zu verwenden, das im Praxisbetrieb »nie« eintreten kann[1]. Tritt dieses Datum bei einer Abfrage auf, erkennt das Programm, daß hier ein Feld nicht belegt wurde. Bei der Tabelle mit den Ausleihdaten kommt dieser Fall immer dann vor, wenn kein Rückgabedatum festgelegt wurde.

---

1 Man soll zwar nie nie sagen, doch in diesem Fall sind wir mit an Sicherheit grenzender Wahrscheinlichkeit für die nächsten Jahre auf der sicheren Seite.

2.2.6 Felder müssen »thematisch« zusammengehören

Auch diese Forderung entspricht mehr dem »gesunden Programmierempfinden« als einer Lehrbuchregel. In einer Tabelle mit Verkaufsdaten hat die Anzahl der Einwohnerzahl einer x-beliebigen Stadt nichts verloren. Alle Felder müssen einen thematischen Bezug besitzen, damit sie gemäß dem relationalen Datenbankmodell eine sogenannte *Relation* (dies entspricht bei einem DBMS einer Tabelle) darstellen. Eine Relation definiert sich dadurch, daß alle ihre Felder vom Primärschlüssel funktional abhängen (mehr dazu später). Mit anderen Worten, wählt man in der Relation (Tabelle) über den Primärschlüssel einen Datensatz aus, ergeben sich dadurch bestimmte Werte für die übrigen Felder. Bei der Auswahl einer Rechnungsnummer kann sich nicht die Einwohnerzahl einer x-beliebigen Stadt ergeben, da beide Daten thematisch nicht zusammengehören. Was thematisch zusammenpaßt, ergibt sich einzig und allein durch den Kontext der Anwendung (dem DBMS sind die Namen und Inhalte der Felder salopp gesprochen egal).

2.2.7 Felder müssen »atomar« sein – die 1. Normalform

Daß der Ausleihzeitraum im letzten Beispiel so nicht angegeben werden kann, liegt auf der Hand, denn in der jetzigen Form wäre es z.B. nur schwer möglich, nach Fahrzeugen zu suchen, die etwa vor dem 1.4.1999 ausgeliehen, oder die noch nicht zurückgegeben wurden. Die Tabelle in Bild 2.1 verstößt noch gegen eine weitere Regel, die allerdings nicht so offensichtlich ist. Das Feld *AusgeliehenVon* enthält streng genommen nicht einen Wert, sondern zwei: den Namen des Mitarbeiters und die Abteilung, in der dieser arbeitet. Dieser Umstand verhindert z.B., daß sich die Datensätze nach Abteilungen sortieren lassen (etwa um herauszufinden, welche Abteilungen die meisten Wagen ausgeliehen hat). Wechselt ein Mitarbeiter die Abteilung, muß das komplette Namensfeld editiert werden, was sicherlich nicht sehr aufwendig, aber unnötig ist. Sehr viel besser in Hinblick auf spätere Abfragen und die Pflege des Datenbestandes ist es, beide Angaben auf zwei Felder zu verteilen. Ein Feld, das nur einen einzigen Wert enthält, wird als *atomar* (also »unteilbar«) bezeichnet. Trifft dies auf alle Felder einer Tabelle zu, befindet sich diese in der 1. Normalform. Und damit wären wir auch schon bei der 1. von insgesamt drei Normalisierungsregeln.

Eine Tabelle befindet sich in der 1. Normalform (1NF), wenn alle Felder atomare Werte besitzen.

Um die Tabelle in die 1. Normalform zu bringen, wird in unserem Beispiel einfach ein weiteres Feld mit dem Namen *Abteilung* eingeführt. Außerdem werden bei dieser Gelegenheit (auch wenn es keine Forderung der 1. Normalform ist) die Angabe des Ausleihzeitraums und die des Fahrzeugnamens vereinheitlicht, indem das erste Feld durch die Felder *AusgeliehenAm* und *RückgabeAm* (beide enthalten lediglich ein Datum) ersetzt und beim Fahrzeugnamen der Hersteller ein eigenes Feld erhält. Aus dem »Opel Astra« wird daher ein konkreter Modellname (z.B. Astra 1.6 D).

Nicht immer reicht es aus, weitere Felder einzuführen, um die 1. Normalform zu erreichen. Für den Fall, daß beim Ausleihen eines Wagens die Namen aller »fahrberechtigten« Mitarbeiter anzugeben sind, müßte man entweder für jeden Mitarbeiter ein eigenes Feld einführen oder aber, und das wäre meistens die bessere Lösung, eine neue Tabelle mit dem Namen »Fahrberechtigte Mitarbeiter« einführen. Das Umwandeln einer Tabelle in die 1. Normalform kann daher auch das Aufteilen der Tabelle auf zwei (oder mehr) Tabellen nach sich ziehen. In diesem konkreten Beispiel ist es jedoch nicht erforderlich. An der Anzahl der Tabellen ändert sich im Moment nichts.

Die 1. Normalform ist eine sehr simple und im Grunde auch naheliegende Regel für Tabellen. Kein Visual-Basic-Programmierer würde vermutlich auf die Idee kommen, mehrere Werte (etwa Nachname, Vorname und Wohnort) in ein Feld zu packen, sondern dafür stets mehrere Felder verwenden. Allerdings, wann ein Feld atomar ist, hängt vom Kontext ab. So ist es durchaus legitim, bei einer Adreßangabe die Hausnummer zusammen mit dem Straßennamen in einem Feld abzuspeichern. Der Feldinhalt »Kropbacher Weg 35« ist daher in diesem Kontext atomar, auch wenn es möglich ist, die Hausnummer in einem separaten Feld unterzubringen. Das gleiche gilt für den Modellnamen. Gegen die Angabe »Opel Astra 1.6D« ist grundsätzlich nichts einzuwenden. Streng genommen ist sie nicht atomar, da sie Herstellernamen und Modellnamen kombiniert. In einer solchen Tabelle gäbe es keine Möglichkeit, Modelle nach Herstellernamen zu sortieren. Wird dies nicht benötigt, ist dies kein Problem. Ansonsten muß die Tabelle normalisiert werden.

Ausleihdaten1NF : Tabelle

Fahrzeug	Kennzeichen	GekauftAm	Farbe	Km/h	KMStand	MitarbeiterName	Abteilung	AusgeliehenAm	RückgabeAm
VW-Käfer	VB-XX-123	01.04.65	Rot	120	12400	Dieter	Werkstatt	15.3.99	18.3.99
Mercedes 190	VB-XX-124	07.07.82	Schwarz	190	4500	Schmidt	Buchhaltung	1.3.99	3.3.99
Ford-Kleinlaster	VB-XX-125	04.05.90	Weiß	162	1236	Maier	Azubi	18.3.99	25.3.99
VW-Käfer	VB-XX-123	01.04.65	Rot	120	12400	Dieter	Werkstatt	21.3.99	1.1.9999
Mercedes 190	VB-XX-124	07.07.82	Schwarz	190	4500	Schmidt	Buchhaltung	21.3.99	23.3.99
Opel Astra	VB-XX-121	1.3.99	Violett	186	120	Frl. Siggy	Rechtsabt.	2.3.99	1.1.9999

Datensatz: 6 von 6

Bild 2.2: Die Tabelle mit den Ausleihdaten befindet sich in der 1. Normalform – Mitarbeitername und Abteilung wurden getrennt

Eine Tabelle in der 1. Normalform bedeutet zwar, daß jedes Feld atomar ist, doch es gibt Situationen, in denen dies zu neuen Problemen führt. Was ist, wenn in einer Datenbank mit Videofilmen alle Hauptdarsteller eines Films zusammen mit dem Filmtitel in einer Tabelle gespeichert werden sollen? Da das gemeinsame Unterbringen der Schauspielernamen in einem Feld gegen die 1. Normalform verstoßen würde, müßte man für jeden Schauspieler ein eigenes Feld einführen. Doch wie viele Felder müßten es sein? 2, 4 oder gar 10? Nimmt man zuwenig, können u.U. bei vielen Filmen nicht alle Hauptdarsteller untergebracht werden. Nimmt man zu viele Felder, bleiben bei einigen Filmen etliche Felder leer. Dieses Problem der »Mehrfachfelder« wird in einem der folgenden Abschnitte durch die Aufteilung der Tabelle in eine Eltern- und in eine Tochtertabelle gelöst. Zuvor müssen wir uns aber noch um ein dringenderes Problem kümmern.

2.2.8 Das Problem der Redundanz

Die 1. Normalform ist ein Zustand, der zwar notwendig, aber noch alles andere als befriedigend ist. Was ist das größte Problem mit der Tabelle in Bild 2.2? Es ist der Umstand, daß viele Daten mehrfach vorkommen. Dieses Phänomen wird auch als *Redundanz* bezeichnet. So werden mit jedem entliehenen Wagen Angaben wie der Tag der Erstzulassung und das Kfz-Kennzeichen aufgeführt, obwohl diese Angaben stets gleich sind (bzw. sich nur selten ändern). Redundanz ist aus folgenden Gründen schlecht:

- ✘ Die Datenbank wird unnötig umfangreich (in unserem Beispiel wurden aus Gründen der Übersichtlichkeit Daten wie Geschwindigkeit, Kaufpreis und viele andere noch gar nicht aufgeführt – in der »echten« Datenbank sind sie natürlich enthalten).
- ✘ Beim Löschen eines Datensatzes gehen Informationen verloren. Wird in der obigen Tabelle ein Fahrzeug entfernt, gehen dadurch auch Informationen über den Mitarbeiter und das Modell verloren.
- ✘ Änderungen am Datenbestand müssen an mehreren Stellen gemacht werden.
- ✘ Abfragen benötigen sehr viel länger, da sich das DBMS durch viele Daten »hindurchwühlen« muß.

Oberstes Ziel eines durchdachten und effektiven Datenbankdesigns muß es daher sein, die Redundanz auf ein Minimum zu reduzieren. Gar keine Redundanz ist, auch wenn sie theoretisch erreichbar ist, aber auch nicht optimal, da die Datenbank dann aus zu vielen kleinen Tabellen besteht, was sich wiederum negativ auf die Performance auswirkt und sogar zu »Informationsverlusten« führen kann. Redundanz wird durch das Verteilen der Daten auf mehrere Tabellen abgebaut. Möchten Sie konkret vermeiden, daß in jedem Datensatz eines entliehenen Fahrzeugs dessen (stets gleichbleibenden) technische Daten enthalten sind, werden diese in einer separaten Tabelle untergebracht.

Wichtigstes Merkmal eines guten Datenbankdesigns ist ein Minimum an identischen Feldern (Stichwort: Redundanz).

2.2.9 Die Rolle der Schlüssel

Bevor wir uns an die Arbeit machen und die Tabelle in Bild 2.2 auf mehrere Tabellen verteilen, muß die Rolle der Schlüssel erklärt werden. In großen Datenbanken kommt es darauf an, einzelne Datensätze in einer Tabelle schnell lokalisieren zu können. Möchte ich z.B. gezielt die Daten aller Wagen eines bestimmten Modells mit der Ausleihhäufigkeit in Beziehung setzen, so wäre ein Durchlaufen der Modelltabelle und ein Vergleich des Modellnamens mit einem Suchbegriff viel zu langsam. Sehr viel effektiver ist es, wenn es in einer Tabelle ein oder mehrere Felder gibt, durch die ein Datensatz (eindeutig) identifiziert wird (und diese Felder gleichzeitig mit einem Index belegt werden, um den Zugriff zu beschleunigen). Diese Felder werden als *Schlüsselfelder* oder kurz Schlüssel (engl. »keys«) bezeichnet und stellen das Fundament relationaler Datenbanken dar. Ein Schlüsselfeld besitzt folgende Merkmale:

- ✘ Es kann sich aus einem oder mehreren Feldern zusammensetzen. In letzterem Fall liegt ein zusammengesetzter Schlüssel vor (engl. »composite key«).
- ✘ Besteht ein Schlüssel aus mehreren Feldern, müssen diese nicht nebeneinanderliegen.
- ✘ Ein Schlüssel muß nicht eindeutig sein, wenngleich dies in den meisten Fällen wünschenswert ist.
- ✘ Ist ein Schlüssel eindeutig, wird er als *Primärschlüssel* bezeichnet. Auch ein Primärschlüssel kann sich aus mehreren Feldern zusammensetzen

(ist ein zusammengesetztes Feld ein Primärschlüssel, spricht man von einem »super key«). Primärschlüssel dürfen keine *NULL*-Werte besitzen.

Unter einem Schlüssel versteht man ein oder mehrere Felder, über die ein Datensatz in einer Tabelle ausgewählt wird.

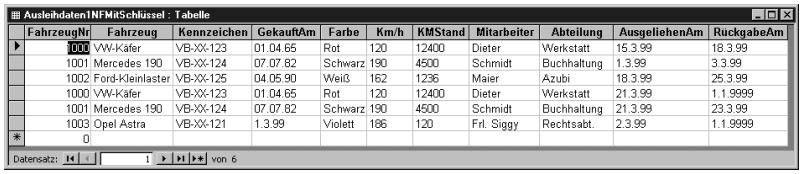
Die Tabelle in Bild 2.2 besitzt noch keinen Schlüssel. Doch welches Feld käme als Schlüssel in Frage? Das Feld *Fahrzeug*, das den allgemeinen Modellnamen eines Fahrzeugs enthält, alleine würde einen Datensatz nicht eindeutig identifizieren, da ein bestimmtes Fahrzeug mehrfach nacheinander ausgeliehen werden kann (zwar ist die Eindeutigkeit eines Schlüssels kein absolutes Kriterium für seine Festlegung, bei kleinen Tabellen ist sie jedoch wünschenswert). Außerdem sind Namensfelder als Schlüsselfelder nur selten gut geeignet. Ein wenig besser ist das Feld *Kennzeichen*, auch wenn es ebenfalls nicht eindeutig ist. Erst die Kombination des Kennzeichens mit dem Ausleihtag ergibt einen eindeutigen Schlüssel, vorausgesetzt, daß ein Wagen am Tag nicht mehrfach ausgeliehen werden kann. Um die Dinge ein wenig zu vereinfachen, wird ein neues Feld mit dem Namen *FahrzeugNr* eingeführt, das zusammen mit dem Ausleihdatum die Rolle des *Primärschlüssels* übernimmt. Dieses Feld wird auch als »künstlicher Schlüssel« bezeichnet, da es zur Beschreibung der Daten nicht notwendig ist.

Nicht immer finden sich in einer Tabelle Felder, die einen Datensatz eindeutig identifizieren. In diesem Fall bietet es sich an, einen künstlichen Schlüssel in Gestalt eines weiteren Feldes einzuführen, das jeden Datensatz numeriert. Bezogen auf die Tabelle mit den Ausleihdaten könnte dieses Feld *AusleihNr* heißen. Über das Feld *FahrzeugNr* wird nun jeder Datensatz eindeutig identifiziert.

Das Verwenden künstlicher Schlüssel ist allerdings nicht immer eine Option. Um in jenen Situationen, in denen das Hinzufügen eines künstlichen Schlüssels nicht praktikabel ist, dennoch einen eindeutigen Schlüssel zu erhalten, kann es wie gezeigt erforderlich sein, mehrere Felder zusammenzufassen. Es ist wichtig, zu verstehen, daß sich die Eindeutigkeit eines Schlüssels nur auf den Kontext der Tabelle beziehen muß[1]. Wird dieser Kontext erweitert, etwa durch Einführen neuer Regeln (bezogen auf dieses Beispiel wäre dies eine Regel, die das mehrfache Ausleihen an einem Tag ermöglicht), müssen auch die Eindeutigkeitsregeln erneut geprüft werden, denn

1 Und nicht etwa auf das gesamte Universum, wie es z.B. bei den GUIDs der Fall ist.

ansonsten kann es passieren, daß der Primärschlüssel seine Eindeutigkeit verliert (die Jet-Engine würde in diesem Fall einen Laufzeitfehler melden). Von Primärschlüsseln wird im weiteren Verlauf dieses Kapitels noch mehrfach die Rede sein. Bevor es an das »Reduzieren der Redundanz« und eine Umwandlung der Tabelle in die 2. Normalform (2NF) geht, muß der Begriff der funktionalen Abhängigkeit erklärt werden.



Ausleihdaten1NFMitSchlüssel : Tabelle

FahrzeugNr	Fahrzeug	Kennzeichen	GekauftAm	Farbe	Km/h	KMStand	Mitarbeiter	Abteilung	AusgeliehenAm	RuckgabeAm
1000	VW-Käfer	VB-XX-123	01.04.65	Rot	120	12400	Dieter	Werkstatt	15.3.99	18.3.99
1001	Mercedes 190	VB-XX-124	07.07.82	Schwarz	190	4500	Schmidt	Buchhaltung	1.3.99	3.3.99
1002	Ford-Kleinlaster	VB-XX-125	04.05.90	Weiß	162	1236	Maier	Azubi	18.3.99	25.3.99
1000	VW-Käfer	VB-XX-123	01.04.65	Rot	120	12400	Dieter	Werkstatt	21.3.99	1.1.9999
1001	Mercedes 190	VB-XX-124	07.07.82	Schwarz	190	4500	Schmidt	Buchhaltung	21.3.99	23.3.99
1003	Opel Astra	VB-XX-121	1.3.99	Violett	186	120	Frl. Siggy	Rechtsabt.	2.3.99	1.1.9999
0										

Datensatz: 1 von 6

Bild 2.3: Die Tabelle Ausleihdaten *wurde um ein Feld ergänzt, das, zusammen mit dem Ausleihdatum, die Rolle des Schlüssels spielen kann*

2.2.10 Die Bedeutung der funktionalen Abhängigkeit

Ein wichtiger Begriff im Zusammenhang mit der Normalisierung einer Tabelle ist die *funktionale Abhängigkeit*, die zwischen zwei oder mehr Feldern existieren kann. Zum Glück ist der Zusammenhang sehr einfach. Ein Feld A ist von einem Feld B funktional abhängig, wenn der Inhalt des Feldes B den Inhalt des Feldes A bestimmt. Die formelle Schreibweise dafür lautet wie folgt:

FeldB ----> FeldA

Dazu ein konkretes Beispiel: Das Feld *FahrzeugNr* bestimmt in der Tabelle *Ausleihdaten* den Inhalt des Feldes *Kennzeichen*. Ändert sich die Fahrzeugnummer, ergibt sich automatisch ein anderes Kennzeichen, denn keine zwei Fahrzeuge können das gleiche Kennzeichen besitzen. Mit anderen Worten, das Feld *Kennzeichen* ist funktional vom Feld *FahrzeugNr* abhängig:

FahrzeugNr ----> Kennzeichen

Funktionale Abhängigkeiten sind weder gut noch schlecht, sondern eine natürliche Eigenschaft von Relationen (Tabellen). Im Zusammenhang mit der Normalisierung einer Tabelle sind sie ein Kriterium, um entscheiden zu können, ob sich eine Tabelle in der 2. oder 3. Normalform befindet. In diesem Zusammenhang soll noch einmal daran erinnert werden, daß ein Schlüssel aus mehreren Feldern bestehen kann und in der Praxis auch besteht. Bezogen auf die Tabelle mit den Ausleihdaten ergibt sich eine funktionale Abhängigkeit der meisten Felder vom Schlüssel. So ist das Feld *Mitarbeiter* vom Schlüssel *FahrzeugNr+AusgeliehenAm* abhängig:

FahrzeugNr, AusgeliehenAm----> Mitarbeiter

Durch die Kombination beider Schlüsselfelder ergibt sich stets ein bestimmter Mitarbeitername.

Teilweise und vollständige Abhängigkeiten

Eine besondere Situation stellt in diesem Zusammenhang der Primärschlüssel dar, denn in einer (echten) Relation sind alle übrigen Felder vom Primärschlüssel funktional abhängig. Ist in einer Tabelle mit den Feldern

FahrzeugNr, AusleihDatum, ModellNr, MitarbeiterNr

das Feld *FahrzeugNr* der Primärschlüssel, sind die übrigen Felder vom Inhalt dieses Feldes funktional abhängig. Doch was ist, wenn der Primärschlüssel aus mehreren Feldern besteht – etwa aus den Feldern *FahrzeugNr* und *AusleihDatum*? In diesem Fall kann es eine Teilabhängigkeit eines Feldes vom Schlüssel geben, die immer dann vorliegt, wenn ein Nicht-Schlüsselfeld (also ein Feld, das nicht zu dem Schlüssel gehört) nur von einem oder mehreren Feldern des Schlüssels, nicht aber von allen Feldern des Schlüssels funktional abhängig ist. Man muß bei der funktionalen Abhängigkeit daher zwischen einer vollständigen und einer teilweisen Abhängigkeit (engl. »partial key dependency«) unterscheiden:

- ✘ Eine vollständige Abhängigkeit liegt vor, wenn alle Nicht-Schlüsselfelder vom gesamten Schlüssel abhängen.
- ✘ Eine teilweise Abhängigkeit liegt vor, wenn ein Nicht-Schlüsselfeld nur von einem Teil des Schlüssels abhängt.

Teilweise Abhängigkeiten sind im allgemeinen nicht erwünscht, da sie bedeuten, daß zwei thematisch unterschiedliche Gruppen von Feldern in einer Tabelle zusammengefaßt wurden.

2.2.11 Die 2. Normalform – keine teilweisen Abhängigkeiten

Wir kommen nun zu einem entscheidenden Punkt, der den Aufbau der Tabelle mit den Ausleihdaten in ihrem jetzigen Zustand grundlegend verändern wird. Dazu müssen Sie sich noch einmal die Bedeutung eines Schlüssels in Erinnerung rufen. In der Tabelle *Ausleihdaten* wird jeder Datensatz über die Felder *FahrzeugNr* und *AusgeliehenAm* eindeutig identifiziert. In der vorliegenden Form der Tabelle *Ausleihdaten* sind allerdings einige Felder vom Teilschlüssel *FahrzeugNr* (funktional) unabhängig. Das ist aufgrund der 2. Normalisierungsregel aber nicht erlaubt, da eine solche Tabelle unnötige

Redundanzen aufweisen würde. Beispiele für nicht abhängige Felder sind die Felder *MitarbeiterName* oder *Geschwindigkeit*, da sich ersteres auf den Ausleihvorgang, letzteres auf ein bestimmtes Modell und beide nicht auf den Ausleihvorgang beziehen. Der Wert dieser Felder wird nicht durch das (Teil-)Schlüsselfeld *FahrzeugNr* bestimmt. Es liegt eine teilweise Abhängigkeit vor.

Funktional abhängig sind dagegen die Felder *Kennzeichen*, *GekauftAm, Farbe*, *KmStand*, *RückgabeAm*. Da die Tabelle *Ausleihdaten* einen »Ausleihvorgang« (und nicht ein konkretes Fahrzeug) beschreibt, ist dies auch ohne mathematischen Beweis nachvollziehbar. Felder ohne Abhängigkeit vom (Gesamt-)Schlüssel führen zu unnötigen Redundanzen und verstoßen gegen die 2. Normalisierungsregel. Eine Tabelle befindet sich erst dann in der 2. Normalform, wenn sämtliche Nicht-Schlüsselfelder vom (Gesamt-)Schlüssel abhängen.

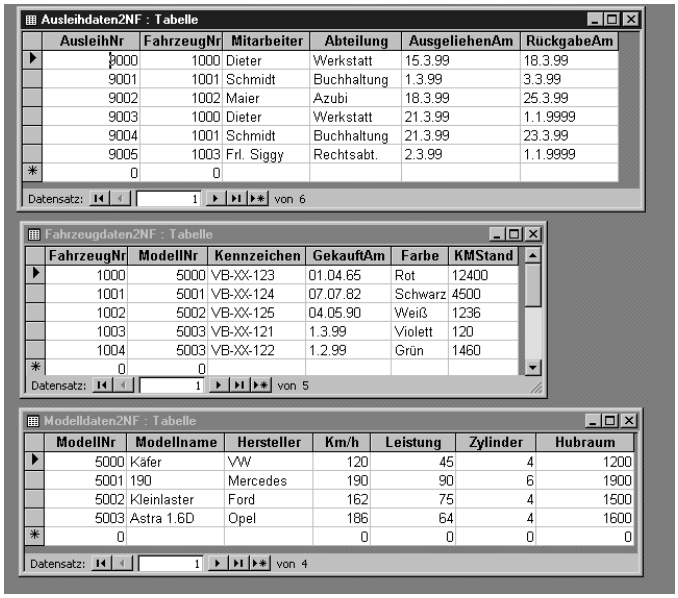
Eine Tabelle befindet sich in der 2. Normalform (2NF), wenn sie sich in der 1. Normalform befindet und es kein Nicht-Schlüsselfeld gibt, das nicht vom Schlüssel oder nur von einem Teil des Schlüssels abhängt.

Das Überführen in die 2. Normalform zieht einen erheblichen Umbau der Tabelle nach sich. Allgemein gesprochen geht es darum, die Tabelle mit den Ausleihdaten in »thematisch verwandte« Tabellen aufzuteilen. Bei näherer Betrachtung lassen sich drei Themenbereiche erkennen:

- ✘ Die Daten eines Entleihvorgangs: *FahrzeugNr*, *MitarbeiterName*, *Abteilung*, *AusgeliehenAm*, *RückgabeAm* (Tabelle *Ausleihdaten*)
- ✘ Die Daten eines entliehenen Fahrzeugs: *Modellname*, *Kennzeichen*, *GekauftAm*, *Farbe*, *KMStand* (Tabelle *Fahrzeugdaten*)
- ✘ Die technischen Daten eines Fahrzeugmodells: *Hersteller*, *Km/h* (Tabelle *Modelldaten*)

Damit ergeben sich drei Tabellen, auf die die vorhandenen Felder verteilt werden (in diesem Zusammenhang wird auch endlich der Fahrzeugname auf die Felder *Modellname* und *Hersteller* aufgeschlüsselt). Damit sich das Feld *km/h* nicht so einsam fühlen muß, wird es um weitere Modelldaten, wie Leistung, Hubraum und Zylinder, ergänzt. Das Ergebnis der Umwandlung ist in Bild 2.3 zu sehen. Eine zentrale Rolle kommt dabei den neu eingeführten (künstlichen) Schlüsselfeldern zu. So enthält die Tabelle *Ausleihdaten* keine Informationen über das entliehene Fahrzeug oder den Entleiher, da diese in separaten Tabellen gespeichert sind. Die Verbindung zu diesen Tabellen

wird über gemeinsame Felder, die Schlüsselfelder, hergestellt. Über das Feld *FahrzeugNr* in der Tabelle *Ausleihdaten* wird der Bezug zu einem Datensatz der Tabelle *Fahrzeugdaten* hergestellt. Diese Tabelle stellt über das Feld *ModellNr* eine Beziehung zur Tabelle *Modelldaten* her. Diese Beziehungen sind das wichtigste Merkmal relationaler Datenbanken. Um die Mitarbeiterdaten, die ebenfalls in eine eigene Tabelle gehören, kümmern wir uns im nächsten Schritt.



Ausleihdaten2NF : Tabelle

AusleihNr	FahrzeugNr	Mitarbeiter	Abteilung	AusgeliehenAm	RückgabeAm
9000	1000	Dieter	Werkstatt	15.3.99	18.3.99
9001	1001	Schmidt	Buchhaltung	1.3.99	3.3.99
9002	1002	Maier	Azubi	18.3.99	25.3.99
9003	1000	Dieter	Werkstatt	21.3.99	1.1.9999
9004	1001	Schmidt	Buchhaltung	21.3.99	23.3.99
9005	1003	Frl. Siggy	Rechtsabt.	2.3.99	1.1.9999
0	0				

Datensatz: 1 von 6

Fahrzeugdaten2NF : Tabelle

FahrzeugNr	ModellNr	Kennzeichen	GekauftAm	Farbe	KMStand
1000	5000	VB-XX-123	01.04.65	Rot	12400
1001	5001	VB-XX-124	07.07.82	Schwarz	4500
1002	5002	VB-XX-125	04.05.90	Weiß	1236
1003	5003	VB-XX-121	1.3.99	Violett	120
1004	5003	VB-XX-122	1.2.99	Grün	1460
0	0				

Datensatz: 1 von 5

Modelldaten2NF : Tabelle

ModellNr	Modellname	Hersteller	Km/h	Leistung	Zylinder	Hubraum
5000	Käfer	VW	120	45	4	1200
5001	190	Mercedes	190	90	6	1900
5002	Kleinlaster	Ford	162	75	4	1500
5003	Astra 1.6D	Opel	186	64	4	1600
0			0	0	0	0

Datensatz: 1 von 4

Bild 2.4: Das Aufteilen der Ausleihdaten auf die Tabellen Ausleihdaten, Fahrzeugdaten und Modelldaten führt zur 2. Normalform

Bemerkenswert an der jetzigen Aufteilung auf drei Tabellen ist der Umstand, wie mehrfach vorkommende Fahrzeuge in der Datenbank gespeichert werden. So ist es natürlich denkbar, daß ein Opel Astra 1.6D mehrfach vorkommt. Die Tabelle *Fahrzeugdaten* enthält aber kein Feld á la AnzahlFahrzeuge, da es (in diesem Kontext) keine identischen Fahrzeuge geben kann. Statt dessen wird für jeden Astra 1.6D ein neuer Datensatz angelegt, der die gleiche Modellnummer, aber eine unterschiedliche Fahrzeugnummer besitzt, denn die »Astras« besitzen die gleichen technischen Daten, unterscheiden sich aber vermutlich bei der Farbe, beim Anschaffungsdatum, beim Kilometerstand und in jedem Fall beim Kfz-Kennzeichen.

2.2.12 Die 3. Normalform – keine transitiven Abhängigkeiten

Die 2. Normalform ist bereits ein wichtiger Zwischenschritt. Wir könnten die Datenbank in ihrem jetzigen Zustand belassen, ohne daß beim Arbeiten spürbare Nachteile auftreten würden. Dennoch ist eine Verbesserung möglich, denn die Redundanz in der Tabelle *Ausleihdaten* ist noch nicht auf einem optimalen Niveau.

Um zu verstehen, was die 3. Normalform auszeichnet, müssen zwei Voraussetzungen geschaffen werden:

- ✘ Die Tabelle besitzt einen (aus mehreren Feldern) zusammengesetzten Schlüssel, denn ansonsten findet die 3. Normalform keine Anwendung. Tabellen, die sich in der 2. Normalform befinden und nur einen einzelnen Schlüssel besitzen, sind automatisch in der 3. Normalform (sowie in allen weiteren).
- ✘ Der Begriff der *transitiven Abhängigkeiten* muß eingeführt werden.

Ob die erste Voraussetzung erfüllt ist, hängt von der Beschaffenheit der Tabelle ab. Läßt sich bei einer Tabelle kein zusammengesetzter Schlüssel finden (z.B. weil bereits ein künstlicher Schlüssel vorhanden ist, der alle Datensätze eindeutig kennzeichnet und von dem alle Felder funktional abhängig sind), und befindet sich die Tabelle in der 2. Normalform, spielen die 3. Normalform sowie alle folgenden Normalformen für die Tabelle keine Rolle. Nur wenn ein zusammengesetzter Schlüssel vorliegt, kann es eine transitive Abhängigkeit geben. Diese liegt immer dann vor, wenn in einer Tabelle ein Nicht-Schlüsselfeld A von einer Kombination aus einem Teil-Schlüsselfeld B und einem weiteren Nicht-Schlüsselfeld C funktional abhängig ist. Oder anders herum, jedes Nicht-Schlüsselfeld eines Datensatzes darf in der 3. Normalform nur vom kompletten Schlüssel abhängig sein und von nichts anderem.

Eine Tabelle befindet sich in der 3. Normalform, wenn sich die Tabelle in der 2. Normalform befindet und jedes Nicht-Schlüsselfeld vom vollständigen Schlüssel funktional abhängig ist. Mit anderen Worten, es darf kein Nicht-Schlüsselfeld geben, das von einer Kombination aus Teil-Schlüsselfeldern und einem weiteren Nicht-Schlüsselfeld abhängig ist (dies wäre eine transitive Abhängigkeit).

Das klassische Beispiel einer transitiven Abhängigkeit sieht wie folgt aus. Die Tabelle *MitarbeiterDaten* enthält folgende Felder:

MitarbeiterNr, MitarbeiterName, PLZ, Strasse, Ort, Bundesland

Der Schlüssel wird durch die Felder *MitarbeiterNr* und *MitarbeiterName* gebildet. *PLZ*, *Strasse* und *Ort* sind funktional vom gesamten Schlüssel abhängig. Das Feld *Bundesland* im Prinzip auch, allerdings ist es zusätzlich von den Feldern *PLZ* und *Ort* abhängig. Da beide aber nicht Teil des Schlüssels sind, befindet sich die Tabelle nicht in der 3. Normalform. Die transitive Abhängigkeit ergibt sich aus dem Umstand, daß der Wert des Feldes *Bundesland* einmal direkt über den Wert von *MitarbeiterNr* (gegebenenfalls ergänzt um *MitarbeiterName*), ein anderes Mal über den Umweg von *PLZ* (oder *Ort*) erreicht werden kann. Auch ohne Kenntnis der Normalisierungsregeln würde erfahrenen Programmierern die ursprüngliche Form der Tabelle »suspekt« vorkommen. Warum wird das Bundesland in die Tabelle aufgenommen, da es unnötigerweise mehrfach abgespeichert wird? Lagern wir die Adreßangaben in eine eigene Tabelle aus, wird diese Redundanz vermieden.

Wie sieht es bei den Fuhrpark-Tabellen bezüglich der 3. Normalisierung aus? Von den drei Tabellen bedarf lediglich die Tabelle *Ausleihdaten* einer näheren Betrachtung. Es ist offensichtlich, daß das Nicht-Schlüsselfeld *Abteilung* vom Feld *Mitarbeiter* abhängt, das ebenfalls ein Nicht-Schlüsselfeld ist. Es liegt eine transitive Abhängigkeit und damit eine (unnötige) Redundanz vor. Abhilfe schafft eine erneute Tabellenteilung. Von der Tabelle *Ausleihdaten* spaltet sich eine Tabelle *Mitarbeiterdaten* ab, die aus den Feldern *MitarbeiterNr*, *Mitarbeitername* und *Abteilung* besteht. Die Beziehung zwischen den beiden Tabellen wird über das gemeinsame Feld *MitarbeiterNr* hergestellt. Wenn Sie der Meinung sind, daß diese Aufteilung schon etwas eher hätte erfolgen sollen, so haben Sie selbstverständlich Recht. Es kommt in der Praxis häufig vor, daß sich durch die Umwandlung von der 1. in die 2. Normalform dadurch eine Tabelle ergibt, die in der 3. Normalform vorliegt. Überhaupt darf bei diesen Maßnahmen nicht der Eindruck entstehen, es ginge darum, bestimmte mehr oder weniger realitätsnahe »Lehrbuchforderungen« zu erfüllen. Das oberste Ziel ist es, Redundanz zu vermeiden und eine für spätere Abfragen optimale, thematisch orientierte Aufteilung der Daten auf verschiedene Tabellen zu erzielen. Wenn man dieses Ziel konsequent verfolgt, ergeben sich normalisierte Tabellen praktisch von selbst.

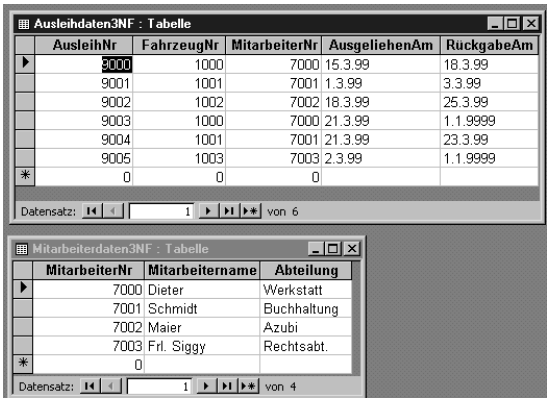


Bild 2.5: Durch Auslagern der Mitarbeiterdaten in eine eigene Tabelle wird die Tabelle Ausleihdaten in die 3. Normalform überführt

2.2.13 Weitere Normalformen

Die ersten drei Normalisierungsregeln wurden von E. F. Codd 1970 im Rahmen eines Fachzeitschriftenartikels vorgeschlagen[1]. Im Laufe der Jahre stellte sich heraus, daß die ersten drei Regeln nicht ausreichen, um Redundanzen in Tabellen mit zusammengesetzten Schlüsseln vollständig zu eliminieren, so daß in den späten 70er Jahren eine 4. und 5. sowie mit der *Boyce-Codd-Normalform* (BCNF) eine »optimierte« Alternative zur 3. Normalform hinzukamen. Letztere stellt eine (notwendige) Verbesserung der klassischen 3. Normalform dar, indem sie fordert, daß jedes Feld, von dem ein anderes abhängt (ein solches Feld wird in diesem Zusammenhang auch als *Determinante* bezeichnet), ein Schlüssel sein muß. Damit werden auch jene Redundanzen eliminiert, die von der normalen 3. Normalform »übersehen« werden würden.

Da die 4. und die 5. Normalform aber weder bei der Implementation der Fuhrpark-Datenbank noch typischer PC-Datenbanken eine Bedeutung haben, werden sie in diesem Buch nicht weiter erwähnt. Dennoch sind auch diese Regeln wichtig. Wer große Datenbanken entwerfen will, sollte sie sich daher zu Gemüte führen.

2.2.14 Vorsicht vor Übernormalisierung

In diesem Zusammenhang darf der Hinweis nicht fehlen, daß eine »Übernormalisierung« einer Datenbank, d.h. eine allzu strenge Aufteilung in Untertabellen sich nicht nur negativ auf die Performance auswirken, son-

1 Der Artikel erschien in der Ausgabe 13/1970 der Zeitschrift »Communications of the ACM« (*www.acm.org*) unter dem Titel » A Relational Model of Data for Large Shared Data Banks«.

dern im ungünstigsten Fall auch Relationen auseinanderreißen und damit zu Informationsverlusten führen kann. Es kommt, wie so oft, auf den goldenen Mittelweg an.

2.2.15 Über den Sinn der Normalisierung

Es ist wichtig, zu verstehen, daß die Anwendung der drei Normalisierungsregeln kein »Ritual« ist, das Datenbankdesigner blind befolgen. Liegt eine Tabelle in der 3. Normalform vor, bietet das handfeste Vorteile beim Umfang der Datenbank:

- ✘ Es können Datensätze aus der Tabelle gelöscht werden, ohne daß dies Auswirkungen auf jene Tabellen hat, die über Schlüsselfelder verbunden sind. Konkret: Sie können Datensätze in der Tabelle *Ausleihdaten* löschen, ohne daß dies eine Auswirkung auf die Tabelle *Modelldaten* hat.
- ✘ Es können Datensätze zur Tabelle hinzugefügt werden, ohne daß dazu die übrigen Tabellen aktualisiert werden müssen. Konkret: Die Tabelle *Modelldaten* kann beliebig viele Datensätze enthalten. Die Tabelle *Fahrzeugdaten* enthält dagegen nur Verweise auf jene Datensätze, für die auch ein Fahrzeug existiert.
- ✘ Da es keine Redundanzen gibt, müssen Änderungen auch nur an einer Stelle durchgeführt werden. Ändert ein Mitarbeiter (etwa durch Heirat) seinen Namen, betrifft dies nur die Tabelle mit den Mitarbeiterdaten. Es wäre geradezu grotesk, müßten diesen Änderungen etwa in den Ausleihdaten durchgeführt werden.

Bereits diese einzelnen Gründe machen deutlich, daß Normalisierung (unabhängig davon, ob sie durch Anwenden der Regeln oder »intuitiv« erreicht wurde) eine wichtige Forderung beim Datendesign ist.

2.2.16 Das Problem der »Mehrfachfelder«

Dieser Punkt wurde bereits im Zusammenhang mit der 2. Normalform angesprochen, doch da er so wichtig ist, erfolgt hier noch einmal eine kurze Wiederholung. Häufig tritt das Problem auf, daß bei einer Tabelle ein Feld mehrfach vorkommen muß. Ein typisches Beispiel ist eine Auftragstabelle, die neben der Kundennummer für jeden bestellten Artikel ein Feld enthält:

AuftragsNr, AuftragsDatum, KundenNr, Artikel1, Artikel2, Artikel3 usw.

Das Problem liegt auf der Hand. Da man nicht wissen kann, wie viele Artikel ein Kunde pro Auftrag bestellt, ist es auch nicht möglich, eine optimale Anzahl an Feldern einzurichten (Arrays sind als Feldtypen bei der Jet-

Engine wie bei den meisten DBMS nicht vorgesehen). Zum Glück sind Sie inzwischen in der Lage, eine Lösung zu finden: die Aufteilung der Tabelle in zwei Tabellen. Eine Tabelle enthält die Auftragsdaten:

AuftragsNr, AuftragsDatum, KundenNr

und die andere die Artikeldaten:

AuftragsNr, ArtikelNr

Die erste Tabelle enthält pro Auftrag einen Datensatz mit einer bestimmten Auftragsnummer, z.B. 9001. Die zweite Tabelle enthält alle bei den einzelnen Aufträgen bestellten Artikel. Indem ein Datensatz der zweiten Tabelle in seinem Feld *AuftragsNr* z.B. den Wert 9001 enthält, wird eine Beziehung zu einem bestimmten Auftrag und damit auch zu einem Kunden hergestellt. Das ist das Prinzip der Normalisierung. Es ist wichtig, zu verstehen, daß die Aufteilung nicht zwingend vorgeschrieben ist. Wenn man voraussetzen kann, daß pro Auftrag z.B. nicht mehr als zehn Artikel bestellt werden können, ist es eine Alternative, in der Auftragstabelle zehn Felder vorzusehen. Das bietet den Vorteil, daß bei späteren Abfragen nicht zwei Tabellen kombiniert werden müssen. Sie sehen an diesem Beispiel ein weiteres Mal, daß die Normalisierungsregeln lediglich Empfehlungen darstellen. In der Praxis muß von Fall zu Fall entschieden werden, wie wichtig und wie vorteilhaft eine Normalisierungsregel ist.

2.2.17 Designüberlegungen bei der Fuhrpark-Datenbank

In den letzten Abschnitten ging es weniger um das konkrete Design der Fuhrpark-Datenbank, sondern mehr um die Veranschaulichung der wichtigsten Datenbankdesign- und Normalisierungsregeln am Beispiel einer konkreten Datenbank. Sie werden beim Festlegen eines Datenbankentwurfs die Erfahrung machen, daß es zwar wichtig ist, diese Grundregeln zu kennen, daß die eigene Erfahrung und das intuitive Gefühl aber im Vordergrund stehen sollten. Mit anderen Worten, Sie sollten sich bei dem Entwurf einer Datenbank nicht zu sehr von den Normalisierungsregeln, sondern stärker von jenen »Vernunftsregeln« leiten lassen, die Sie auch beim Programmentwurf zugrunde legen würden. Die Normalisierungsregeln, die in den 70er Jahren entstanden, waren als allgemeine Regeln gedacht, die kein spezielles (Programmier-)Wissen voraussetzen. Wäre es damals darum gegangen, angehenden Datenbankprogrammierern mit Programmiererfahrung in einer leistungsfähigen Programmiersprache allgemeine Empfehlungen an die Hand zu geben, wären sie vermutlich weit weniger formell ausgefallen. Auf

der anderen Seite liegt gerade in dem strengen Formalismus eine gewisse Stärke, die die Regeln universell anwendbar macht.

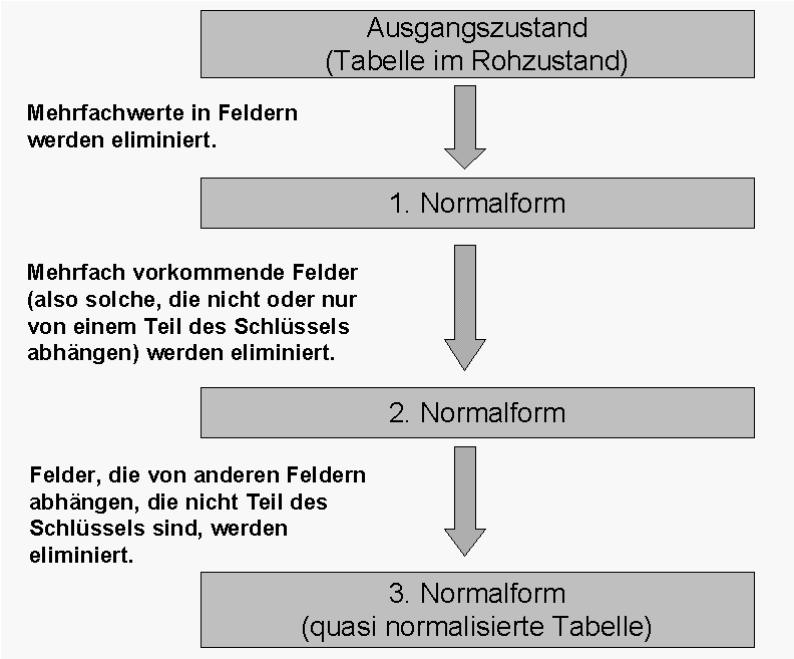


Bild 2.6: Die verschiedenen Normalisierungsstufen in der Übersicht

Beim Entwurf der Fuhrpark-Datenbank, wie er in Kapitel 3 vorgegeben wird und wie Sie ihn in der Datenbank *Fuhrpark.mdb* vorfinden, haben die Normalisierungsregeln nur indirekt eine Rolle gespielt. Im Vordergrund stand der Wunsch, eine Datenbank zu schaffen, die sich mit den Möglichkeiten von Visual Basic, einfachen (d.h. nicht zu komplexen) SQL-Abfragen und den Möglichkeiten der *Active Data Objects* (ADO) möglichst optimal benutzen läßt. Daß eine Aufteilung in verschiedene Tabellen erforderlich ist, war dabei von Anfang an klar und hat sich nicht erst durch strenges Anwenden der Normalisierungsregeln ergeben.

Ausgangspunkt war dabei die Überlegung, daß folgende Daten möglichst sinnvoll auf verschiedene Tabellen verteilt werden mußten:

- ✘ Die technischen Daten eines Fahrzeugmodells (etwa Geschwindigkeit, Leistung, Hubraum usw.)

- ✘ Die spezifischen Daten eines im Fuhrpark vorhandenen Fahrzeugs (etwa Kfz-Kennzeichen, Baujahr, Farbe, Anschaffungspreis, Termin der nächsten Inspektion usw. – für die Inspektionstermine wäre es aufgrund des »Mehrfachfeldproblems« sinnvoller, eine eigene Tabelle anzulegen)
- ✘ Die Daten eines Ausleihvorgangs (etwa Fahrzeugnummer, Name des Mitarbeiters, Ausleihdatum usw.)
- ✘ Die Daten eines Mitarbeiters (Name, Dauer der Firmenzugehörigkeit usw.)

Durch Anwenden allgemeiner Regeln und damit indirekt auch der Normalisierungsregeln ergaben sich in der Fuhrpark-Datenbank folgende Tabellen:

- ✘ Die Tabelle *Modelldaten* mit den Modelldaten aller Fahrzeuge
- ✘ Die Tabelle *Fahrzeugdaten* mit den Daten der tatsächlich vorhandenen Fahrzeuge
- ✘ Die Tabelle *Entleihdaten* mit den Daten der entliehenen Fahrzeuge
- ✘ Die Tabelle der *Mitarbeiterdaten*, wenngleich diese Daten in der Praxis in einer anderen Datenbank (etwa der Personaldatenbank) enthalten sind

Das ist lediglich die Grobstruktur, zumal längst nicht alle relevanten Daten berücksichtigt wurden (u.U. möchte man die laufenden Kosten oder die vorhandenen Ersatzteile für ein Fahrzeug erfassen – diese Möglichkeit ist in der Beispieldatenbank nicht vorgesehen, denn dann würde es richtig »kompliziert« werden). Bei der Implementation der Datenbank würden Sie feststellen, daß in einigen Fällen sogar eine weitere Unterteilung sinnvoll ist. Allzu granular sollte die Datenbank aber auch nicht werden, denn zu viele kleine Tabellen können sich ebenfalls negativ auf die Performance auswirken. Es kommt, wie bereits erwähnt, auf die goldene Mitte an.

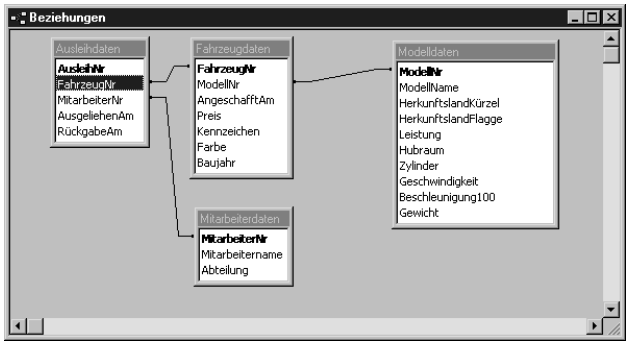


Bild 2.7: Der Aufbau der Datenbank Fuhrpark.mdb in ihrer ersten Ausbaustufe – die Verbindungslinien geben die Relationen an

2.2.18 Die Rolle der Schlüssel

Die letzten Abschnitte haben eines deutlich gemacht: Die Aufteilung der Felder auf mehrere Tabellen ist in der Datenbankpraxis ein Muß. Allerdings geht durch die Aufteilung zunächst Information verloren. Wenn in einer Tabelle nur die technischen Daten der im Fuhrpark befindlichen Daten enthalten sind, gibt es keine Information darüber, wer welches Auto ausgeliehen hat, denn diese Information befindet sich in einer anderen Tabelle. Das Verbindungsstück stellen die Beziehungen (Relationen) her. Diese werden, wie es in Kapitel 1 bereits angedeutet und in den vergangenen Abschnitten auch angewendet wurde, über Schlüssel (Schlüsselfelder) hergestellt. Eine Tabelle besitzt in der Regel einen Primärschlüssel, auf den mehrere Fremdschlüssel in einer anderen Tabelle verweisen. Was sich ein wenig abstrakt anhören mag, ist in der Praxis ganz einfach. Ein Beispiel aus der Fuhrpark-Datenbank soll dies veranschaulichen. Gemäß der im letzten Abschnitt aufgestellten Unterteilung werden die Modelldaten in einer eigenen Tabelle mit dem Namen *Modelldaten* gespeichert, die u.a. folgende Felder umfaßt:

- ModellNr
- ModellName
- Hersteller
- Leistung
- Geschwindigkeit
- Hubraum
- Zylinder

usw.

Daten wie Preis, Farbe oder das Kfz-Kennzeichen sind hier nicht enthalten, denn diese können bei ein und demselben Modell variieren. Die Tabelle enthält nur jene Daten, die bei ein und demselben Modell identisch sind. Eine besondere Bedeutung kommt dem Feld *ModellNr* zu. Es ist kein technisches Merkmal, wie etwa der Hubraum, sondern lediglich eine Zahl, durch die sich jedes Modell unterscheidet und die z.B. von 1 beginnend mit jedem neuen Modell um eins erhöht wird. Die Werte des *ModellNr*-Feldes lauten 1, 2, 3 usw. (oder 9001, 9002, 9003, um eine in der Datenbankwelt vorteilhaftere Mehrstelligkeit eines Schlüssels von Anfang an zu gewährleisten – Rechnungsnummern beginnen deswegen auch nicht bei 1, sondern bei einem Fixwert). Nun kommt der springende Punkt. Da die Modellnummer bei jedem Modell verschieden ist, spielt dieses Feld die Rolle des Primärschlüs-

sels, denn über seine Modellnummer wird der Datensatz eindeutig identifiziert.

Festgelegt wird ein Primärschlüssel bei Microsoft Access durch Auswahl des Feldes in der Entwurfsansicht mit der rechten Maustaste und Auswahl des Menüeintrags PRIMÄRSCHLÜSSEL. Dadurch wird gleichzeitig ein Index definiert. Beim Visual Data Manager wird der Primärschlüssel dagegen durch Hinzufügen eines Index definiert, der das Attribut »unique« (also einzigartig) besitzt. Ansonsten ist das Primärschlüsselfeld ein Feld wie jedes andere auch.

Die Struktur der Tabelle mit den Daten der Modelldaten steht. Nun ist die Tabelle *Fahrzeugdaten* mit den vorhandenen Fahrzeugen an der Reihe. Sie enthält u.a. folgende Felder:

- ✘ FahrzeugNr
- ✘ Anschaffungsdatum
- ✘ Baujahr
- ✘ Farbe
- ✘ Preis
- ✘ Kennzeichen
- ✘ NächsteInspektion

usw.

Fällt Ihnen bei dieser Tabelle etwas auf? Richtig, es gibt noch keinen Bezug auf die technischen Daten eines Fahrzeugs, die sich in der Tabelle *Modelldaten* befinden. Wie läßt sich ein solcher Bezug herstellen? Ganz einfach, indem die Tabelle ein weiteres Feld enthält, dessen Inhalt eine Modellnummer aus der Tabelle *Modelldaten* (oder einer anderen Tabelle mit Fahrzeugdaten) ist:

- ✘ ModellNr

Über dieses Feld wird die Beziehung zu dem Primärschlüssel *ModellNr* in der Tabelle *Modelldaten* hergestellt. Es wird daher auch als *Fremdschlüssel* bezeichnet. Zwischen dem Primärschlüssel *ModellNr* in der Tabelle *Modelldaten* und dem Fremdschlüssel *ModellNr* in der Tabelle *Fahrzeugdaten* (die Namensübereinstimmung ist nicht zwingend notwendig) besteht eine 1:n-Beziehung, da ein bestimmtes Modell mehrfach im Fahrzeugbestand vorkommen kann.

Es ist wichtig, zu verstehen, daß eine Beziehung zwischen einem Primärschlüssel und einem Fremdschlüssel nicht über einen bestimmten Befehl in dem jeweiligen Datenbankprogramm hergestellt werden muß, da sie sich lediglich aus der sinnvollen Aufteilung der Felder auf die verschiedenen Tabellen ergibt. Bei der Jet-Engine sowie bei den meisten modernen DBMS kann sie dagegen auch »physikalisch« implementiert werden. Bei der Jet-Engine wird durch das explizite Festlegen einer Relation z.B. die referentielle Integrität implementiert. Das bringt u.a. den Vorteil, daß das versehentliche Löschen eines Datensatzes mit einem Primärschlüssel verhindert wird, wenn in anderen Tabellen Datensätze existieren, die sich auf diesen Primärschlüssel beziehen. In diesem Fall wird ein Laufzeitfehler ausgelöst, auf den das (VBA-)Programm entsprechend reagieren sollte. Bezogen auf die Fuhrpark-Datenbank läßt sich so mit relativ wenig Aufwand verhindern, daß der Anwender direkt oder indirekt einen Datensatz aus der Tabelle *Modelldaten* löschen kann, wenn es in der Tabelle *Fahrzeugdaten* einen Wagen dieses Modells gibt. Würde der Datensatz nämlich gelöscht werden, enthielten ein oder mehrere Datensätze in der Tabelle *Fahrzeugdaten* Modellnummern, die es nicht mehr gibt. Dies würde beim Aufruf des Datensatzes und dem Abrufen der Modelldaten über den Fremdschlüssel von einem VBA-Programm aus nicht nur zu einem Laufzeitfehler führen, sondern auch die Integrität der Datenbank gefährden. Bei einer Datenbank, deren Integrität nicht mehr gewährleistet ist, lassen sich elementare Datenbankoperationen nicht mehr korrekt durchführen (da sich diese z.B. auf Felder beziehen, die nicht mehr existieren), was nicht nur die Performance herabsetzt, sondern auch zu Datenverlust führen kann.

Soviel zur Theorie. Wie aus den in diesem Abschnitt vorgestellten Tabellen eine richtige Acccess-Datenbank wird und wie z.B. Schlüssel festgelegt werden, erfahren Sie in Kapitel 3.

Bild 2.8: In diesem Dialogfeld werden bei Microsoft Access Beziehungen zwischen zwei Feldern zweier Tabellen hergestellt

2.2.19 Zusammenfassung zum Thema Datenbankdesign

Der Entwurf einer Datenbank beschäftigt sich im wesentlichen mit der Frage, wie die zu speichernden Daten auf mehrere Tabellen verteilt werden, wobei die Normalisierungsregeln eine wichtige Grundlage darstellen. Datenbankdesign ist eine sehr wichtige Angelegenheit, denn eine »schlechte« Aufteilung kann die Geschwindigkeit beim Datenzugriff entscheidend bremsen. Insbesondere dann, wenn die Datenbank mehr als die üblichen zehn Datensätze der meisten Beispielprogramme enthält[1]. Endanwender-Datenbanken, wie Microsoft Access, verbergen es dank zahlreicher Assistenten und Vorlagen geschickt, daß der Datenbankentwurf eigentlich auf einem Blatt Papier und mit einem Bleistift[2] stattfinden und nach folgendem simplen Schema ablaufen sollte:

- ✘ Aufschreiben aller zu speichernden Daten und Festlegen von Feldnamen für diese Daten.
- ✘ Vorläufiges Verteilen der Felder auf verschiedene Tabellen. In der Regel werden die Felder »thematisch« gruppiert (also z.B. Rechnungsdaten in eine, die Artikeldaten in eine andere Tabelle, denn das Mischen dieser Themengruppen führt zu Redundanz).
- ✘ Zuordnen von Beziehungen zwischen einem Primärschlüssel in einer und einem Fremdschlüssel in einer anderen Tabelle.
- ✘ Durchspielen der aufgebauten Beziehungen anhand einfacher (SQL-) Abfragen.
- ✘ Normalisieren der Tabellen in der 1. und 2. und gegebenenfalls auch 3. Normalform.

Die hier vorgestellten Schritte sind alles andere als kompliziert, sondern beschreiben lediglich das »gesunde« Empfinden, das die meisten Programmierer ohnehin einer Entscheidung bezüglich der Aufteilung der Daten auf Tabellen zugrunde legen würden. Sie sind nichtsdestotrotz wichtig, denn ein schlechtes Datenbankdesign läßt sich später, wenn das Programm unter Umständen bereits beim Anwender im Einsatz ist, nur noch mit großem Aufwand (und Ärger) korrigieren.

---

1 Die Visual-Basic-Beispieldatenbank *Biblio.mdb* ist in diesem Punkt eine löbliche Ausnahme, denn ihre *Authors*-Tabelle enthält weit über 8000 Datensätze.

2 Oder, wenn Sie 10000 DM entbehren können, mit einem CASE-Werkzeug (CASE=Computer Aided Software Engineering, eine Methode, die vor allem für den Entwurf von Datenbanken verwendet wird).

2.3 Die Organisation einer Access-Datenbank

Eine Datenbank enthält nach der reinen Lehre relationaler Datenbanken Tabellen, Datensätze, Felder und Relationen, die durch Beziehungen zwischen zwei Feldern oder (bei n:m-Beziehungen) durch Einfügen von »Zwischentabellen«[1] definiert sind. In der Praxis muß eine Datenbank sehr viel mehr speichern als nur die reinen Daten. In diesem Abschnitt geht es um den physikalischen Aufbau einer Access-Datenbank. Im Vergleich zu älteren PC-Datenbankprogrammen (etwa dBase, das in den achtziger Jahren sehr populär war) werden die Tabellen bei Microsoft Access nicht als einzelne Dateien abgelegt (bei dBase wurde jede Tabelle in einer eigenen Datei mit der Erweiterung *.Dbf* abgelegt). Vielmehr werden alle Tabellen in einer Datei mit der Erweiterung *.Mdb* zusammengefaßt.

Die »dateibasierende Datenhaltung« ist bei Datenbank-Management-Systemen eher untypisch. Große DBMS, etwa der Microsoft SQL-Server oder der Oracle SQL-Server, legen die Daten in einer speziellen Festplattenstruktur ab, die von außen nicht als einzelne Datei in Erscheinung tritt und damit von außen nicht zugänglich ist. Die einfache Formel »1 Datenbank = 1 Datei« gilt hier nicht, zumal sich die Datenbank an einem beliebigen Ort innerhalb des PC-Netzwerks befinden kann[2]. Hier ist es nicht möglich, eben mal schnell mit dem Explorer eine Datenbank zu kopieren. Für alle Zugriffe auf die Datenbanken werden Administrationsprogramme, beim Microsoft SQL-Server z.B. der SQL-Enterprise-Manager, benötigt.

Bei Microsoft Access hat man sich anfangs bewußt für eine einfachere Lösung entschieden, die sich stärker an den Gewohnheiten der meisten PC-Anwender orientiert. Möchten Sie die komplette Datenbank auf einen anderen PC transferieren, kopieren Sie die *Mdb*-Datei einfach auf eine Diskette oder einen anderen Datenträger. Wo sich die *Mdb*-Datei physikalisch befindet, befindet sich auch die Datenbank. Diese Einfachheit birgt natürlich auch Nachteile in sich:

- ✘ Die Daten in einer *Mdb*-Datenbank sind grundsätzlich ungeschützt, da jeder, der über Microsoft Access, Visual Basic oder ein anderes Programm verfügt, auf die Datenbank zugreifen kann (allerdings besteht die

1 Das ist für Sie inzwischen keine Neuigkeit mehr, doch bestimmte Dinge kann man am Anfang nicht oft genug wiederholen.

2 Früher war dies sogar der Normalzustand. Der Zugriff auf eine Datenbank erfolgte über schlichte Terminals, während sich die Datenbank in einem abgeschlossenen, klimatisierten Raum befand, zu dem nur die Systemoperatoren Zutritt hatten.

Möglichkeit, den Inhalt der *Mdb*-Datenbank zu verschlüsseln und mit einem Kennwort zu sichern).

- ✘ Es ist zunächst einmal nicht möglich, den Mehrfachzugriff auf eine *Mdb*-Datenbank zu limitieren, da jeder die Datei öffnen kann. Möchte man eine Zugriffsbegrenzung einführen (z.B. um die Anzahl der gleichzeitigen Benutzer auf 10 zu limitieren), muß dies programmtechnisch geschehen. Beim Microsoft SQL-Server sind diese Dinge in die Datenbank eingebaut.

Da diese »Nachteile« aufgrund der stetig wachsenden Anforderungen auf Dauer die Konkurrenzfähigkeit eines DBMS einschränken, wird beginnend mit Microsoft Access 2000 eine Alternative zur Jet-Engine angeboten: die *Microsoft Desktop Engine* (MSDE), die der Desktop-Version des Microsoft SQL-Servers 7.0 entspricht und daher die gleiche Form der Datenbankverwaltung besitzt.

Neben den Tabellen enthält eine *Mdb*-Datenbank noch weitere »Familienmitglieder«:

- ✘ Die Systemtabellen, zu erkennen an der Vorsilbe »MSys«. Hier werden Daten gespeichert, die Microsoft Access zur internen Verwaltung benötigt (z.B. Informationen über den Aufbau der Tabellen, über Relationen, über die Replikation von Tabellen oder über Zugriffsbeschränkungen). Auf diese Tabellen greift man im allgemeinen nicht direkt zu (wenngleich es möglich ist).

Abfragen (auch »Queries« genannt). Dies sind SQL-Abfragen, die »vorverarbeitet« in der *Mdb*-Datenbank gespeichert sind und damit nicht bei jedem Aufruf neu interpretiert werden müssen. Sie werden von außen wie Tabellen angesprochen.

- ✘ Formulare, Reports, Makros und VBA-Module. Dies sind Elemente, die nur von Microsoft Access in die Datenbank eingefügt und nur von Microsoft Access benutzt werden können. Zwar ist es möglich, von Visual Basic aus einen in einer *Mdb*-Datei enthaltenen Report zu drucken oder ein Makro zu starten, doch muß dazu vom Visual-Basic-Programm (über Automation) erst Microsoft Access gestartet werden, was dessen physikalische Installation voraussetzt. Ein direkter Zugriff auf diese Elemente über die Jet-Engine ist mit den ADO/DAOs alleine nicht möglich.

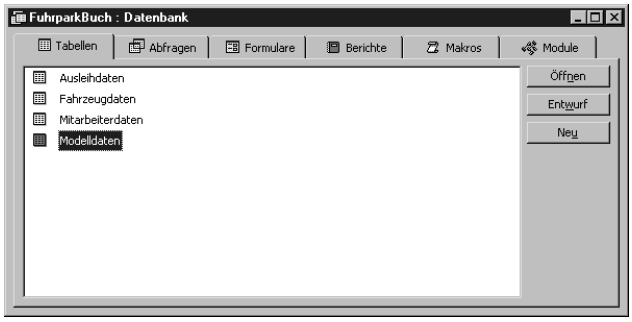


Bild 2.9: Im Datenbankfenster von Microsoft Access 97 werden alle Elemente einer Mdb-Datei aufgelistet

2.3.1 Die Rolle von Tabellen und Feldern

Im letzten Abschnitt wurde deutlich, daß eine *Mdb*-Datenbank viele verschiedene Dinge enthalten kann, wobei alle Informationen, sowohl die Daten als auch zusätzliche Informationen, in Tabellen gehalten werden. Eine Tabelle gibt aber nur die Struktur, d.h. die Anordnung und Datentypen der einzelnen Felder in jedem Datensatz vor. Die eigentlichen Daten befinden sich daher in den Feldern der Datenbank, die zugleich die kleinste, atomare Einheit in der Datenbank darstellen (d.h., die Unterteilung eines Feldes in »Unterfelder« ist bei Access-Datenbanken nicht vorgesehen).

Für die Größe einer Tabelle und eines Datensatzes gibt es keine (echten) Limits (wenngleich dies natürlich stets relativ ist). Eine Tabelle enthält mindestens ein Feld, sonst wäre sie keine Tabelle. Sie kann dagegen null oder mehr Datensätze enthalten, wobei es pro Datensatz bei der Jet-Engine nicht mehr als 256 Felder geben kann. Dies ist eine durchaus vernünftige Einschränkung, denn aus praktischen Erwägungen sollte eine Tabelle nicht zu viele Felder enthalten. Auch für die maximale Größe einer Datenbank gibt es bei Microsoft Access ein natürliches Limit: Es liegt bei der Version 97 bei 2 Gbyte. Möchten Sie also die Adressen aller Ihrer Freunde und Bekannten verwalten und geht man davon aus, daß ein einzelner Datensatz 1024 Byte belegt, können es maximal ca. 1 Million Datensätze werden. Sollte das nicht ausreichen, müßte der Datenbestand entweder auf mehrere Datenbanken verteilt werden (etwa nach guten Freunden und weniger guten Freunden unterteilt), oder Sie müßten auf ein leistungsfähigeres DBMS umsteigen.

2.3.2 Der Datentyp eines Feldes

Halten wir fest, die eigentlichen Daten einer Datenbank befinden sich in den Feldern eines Datensatzes. Aus der Sicht eines Programmierers entsprechen Felder Variablen, nur daß deren Inhalt automatisch dauerhaft gespeichert wird (genau dazu ist die Datenbank da). Jedes Datenbankfeld erhält beim Einrichten

der Tabelle einen Datentyp zugewiesen. Welche Datentypen zur Auswahl stehen, hängt vom Datenbank-Management-System ab. Genau wie die Programmiersprache Java andere Datentypen besitzt als die Programmiersprache VBA, unterscheidet sich auch die Jet-Engine in diesem Punkt von ihren »Mitbewerbern«, wobei es selbstverständlich, genau wie bei den Programmiersprachen, Übereinstimmungen bei den elementaren Datentypen gibt. Die Datentypen der Jet-Engine werden in Kapitel 3 zusammengefaßt.

Die Rolle von NULL

Ein Sonderfall tritt bei der Datenbankprogrammierung immer dann auf, wenn ein Datenbankfeld keinen Wert beinhaltet. Es ist dann nicht einfach 0 oder leer, in der Datenbankfachsprache besitzt dieses Feld den Wert *NULL* (ausgesprochen wie »nall«). *NULL* ist ein feststehender Begriff, der konkret bedeutet: »Das Feld besitzt keinen Inhalt.« *NULL*-Werte treten z.B. auf, wenn in einer Tabelle ein neuer Datensatz angelegt wird, aber nicht alle Felder mit Werten belegt wurden. Da der Begriff *NULL* sehr wichtig ist, existiert er bei VBA auch als eigener Datentyp, allerdings als Unterdatentyp des Datentyps *Variant*. Möchten Sie feststellen, ob eine Variable oder ein Datenbankfeld den Wert *NULL* besitzt, verwenden Sie die *IsNull*-Funktion.

Der folgende VBA-Befehl greift auf ein Datenbankfeld (einer zuvor geöffneten Tabelle) zu und prüft, ob es sich um einen *NULL*-Wert handelt:

```
Dim TestWert As Variant
TestWert = Rs.Fields("Modellname").Value
If IsNull(TestWert) Then
```

Der Variablen *TestWert* wird der Wert eines Datenbankfeldes zugewiesen. Anschließend wird geprüft, ob dieser Wert den Wert *NULL* besitzt. Beachten Sie dabei, daß Sie einen *NULL*-Wert einer Variablen nur dann zuweisen können, wenn diese vom Typ *Variant* ist. Besitzt die Variable z.B. den Typ *Long*, wäre durch die Zuweisung ein Laufzeitfehler die Folge.

Um beim Zuweisen eines Datenbankfeldes an eine Variable, die nicht vom Typ *Variant* ist, einen Laufzeitfehler auszuschließen, kann man einen Leerstring an den zugewiesenen Wert hängen. Das stellt sicher, daß in jedem Fall etwas übergeben wird:

```
TestWert = Rs.Fields("Modellname").Value & ""
```

Dies ist jedoch keine Ideallösung, da der Umstand, daß ein Feld einen NULL-Wert besitzt, nicht einfach übergangen, sondern gesondert behandelt werden sollte. Welche Rolle *NULL*-Werte spielen, ist ein Thema, über das man sich am Anfang noch keine Gedanken machen sollte. Die Abfrage auf

NULL muß jedoch ein fester Bestandteil einer Datenbankanwendung sein, um Laufzeitfehler und andere unangenehme Begleiterscheinungen zu vermeiden[1].

Die Rolle von Memo-Feldern

Nicht alle Felder, die in einer Tabelle gespeichert werden, spielen beim Zugriff eine »aktive« Rolle. Stellen Sie sich vor, daß Sie in der Fuhrpark-Datenbank zu jedem Wagen auch einen Kommentar mit allgemeinen Anmerkungen speichern möchten. Da das zuständige Feld einen rein passiven Charakter besitzt und z.B. bei Datenbankabfragen nicht als Suchkritierium verwendet wird, wäre es aus Performance-Gründen zu aufwendig, dafür ein reguläres Feld zu verwenden. Für diesen Zweck bietet die Jet-Engine sogenannte Memo-Felder, die man sich als »neutrale« Zusatzfelder mit einem beliebigen (Text-)Inhalt vorstellen kann. Besitzt ein Feld den Datentyp *Memo,* können darin im Prinzip beliebig viele Zeichen gespeichert werden (die Größe wird durch die mögliche maximale Größe der Datenbank limitiert). Die maximale Größe einer Access '97-Datenbank beträgt 1 Gbyte.

2.3.3 Bilder, Hyperlinks und andere »binäre« Dinge

Damit ein DBMS möglichst flexibel eingesetzt werden kann, muß es in der Lage sein, nicht nur die Standarddatentypen Zahlen und Texte zu speichern. Die Jet-Engine geht da mit gutem (wenngleich nicht perfektem) Beispiel voran, indem sie neben den Standarddatentypen, wie *Boolean*, *Byte*, *Integer* und *Text*, die Datentypen *Binary* und *Hyperlink* unterstützt. Ist ein Feld vom Typ *Binary* (bei Microsoft Access heißt es *OLE Object*), kann es beliebige binäre Daten enthalten. Dies sind Datenelemente, die sich aus einer beliebigen Folge von Bytes zusammensetzen. Am häufigsten werden *Binary*-Felder für die Speicherung von Bitmaps (siehe Kapitel 10) verwendet. In Feldern vom Typ *Hyperlink* werden dagegen sogenannte URLs gespeichert. Die Aufgabe eines *Unified Resource Locators*, wie z.B. *http://www.activetraining.de*, ist es, eine beliebige Ressource, etwa eine Computeradresse, ein HTML-Dokument oder eine Bitmap, im Internet oder einem Intranet zu adressieren. Indem die Jet-Engine den Datentyp *Hyperlink* direkt unterstützt, ist es sehr einfach, URLs in einer Datenbank zu speichern (man müßte sie ansonsten als Textelemente speichern, wo-

1 Wenn Sie sich etwas ausführlicher mit der Demodatenbank *Biblio.mdb* beschäftigen, werden Sie feststellen, daß hier aus »didaktischen« Gründen eine Reihe von Feldern NULL-Werte besitzen.