D B P R O G

Datenbank-Programmierung

© Herbert Paukert

ENTWICKLUNG VON DATENBANKEN

Mit dem Programm "***prodat***" soll eine einfache Datenbank mit nur einer Datenbankdatei (Tabelle) erzeugt und verwaltet werden. Die Tabelle hat den Dateinamen "***dat.db***" und enthält die acht Datenfelder **Nummer**, **Zuname, Vorname, Geburt, Strasse, PLZ, Ort** und **Telefon**.

[01] Grundlagen der Datenbank-Programmierung

In dem Entwicklungssystem Delphi ist ein sehr mächtiges Datenbank-Werkzeug integriert, und zwar die so genannte *Borland Database Engine (BDE)*. Die *BDE* befindet sich standardmäßig im Verzeichnis *C:\Programme\Gemeinsame Dateien\Borland Shared\BDE* und besteht aus mehreren Hilfsprogrammen und DLLs. Das Kernprogramm heißt *bdeadmin.exe*. Die *BDE* wirkt bei Delphi-Datenbankanwendungen als Schnittstelle zwischen den Windows-Komponenten auf der einen Seite und den eigentlichen Datenbanken auf der anderen Seite. Für die Datenbank-Formate Paradox, dBase, MS-Access und ASCII stellt die *BDE* mit den entsprechenden Treiberdateien eine standardmäßige Unterstützung zur Verfügung.

Der große Vorteil der *BDE* für den Entwickler von Software ist, dass es ihn nicht mehr zu interessieren braucht, auf welche Art von Datenbanken zugegriffen wird. Durch die *BDE* wird es nahezu belanglos, in welchem Format die Daten gespeichert sind, da diese sich mit Hilfe der *BDE* gleichartig bearbeiten lassen. Es ist auch nicht von Bedeutung, ob sich die Datenbank lokal auf dem Rechner befindet oder ob der Rechner in einer Client/Server-Umgebung eingebettet ist und die Datenbank sich auf einem entfernten Server befindet. Auf jeden Fall stellt die *BDE* die Daten in einer solchen Form zur Verfügung, dass sie von Delphi gelesen und verarbeitet werden können.

In unserem Beispiel werden wir das Gerüst unserer Datenbank mit der zu Delphi mitgelieferten Datenbankoberfläche *Borland Database Desktop* (*BDD*) entwerfen. Das Hilfsprogramm *dbd32.exe* befindet sich mit Zusatzdateien im Verzeichnis *C:\Programme\Borland\Datenbankoberfläche* und dient der komfortablen Erzeugung von Datenbanken entsprechend den Wünschen des Anwenders. Es wird von Delphi über das Menü *Tools/Datenbankoberfläche* aufgerufen.

- **Der Datenbank-Entwurf**

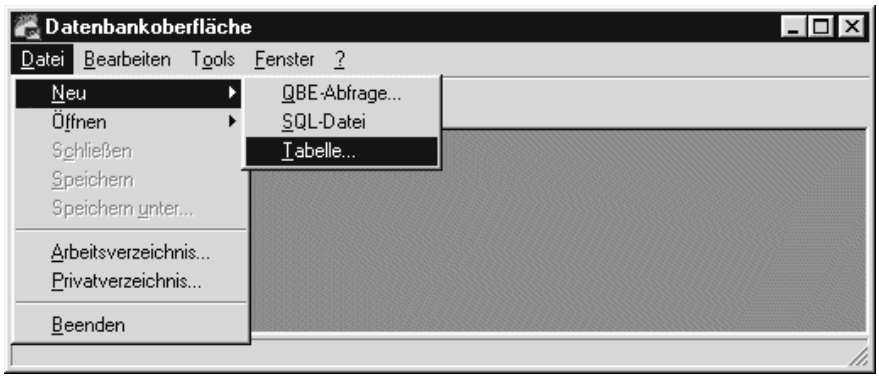
Normalerweise beginnt die Entwicklung einer Datenbankanwendung mit einer Problemanalyse und dem Entwurf der Datenbankstruktur. Da wir es in unserem Beispiel nur mit einer einzigen Tabelle zu tun haben, brauchen wir uns lediglich überlegen, welche Felder pro Person abgespeichert werden sollen und von welchem Datentyp und von welcher Feldgröße diese sind. Unsere Tabelle soll *"dat.db"* heißen und folgenden Aufbau haben:

Feldname	***Feldtyp***	***Feldgröße***
Nummer	Zähler	4
Zuname	Alpha	25
Vorname	Alpha	25
Geburt	Datum	6
Strasse	Alpha	25
PLZ	Alpha	5
Ort	Alpha	25
Telefon	Alpha	15

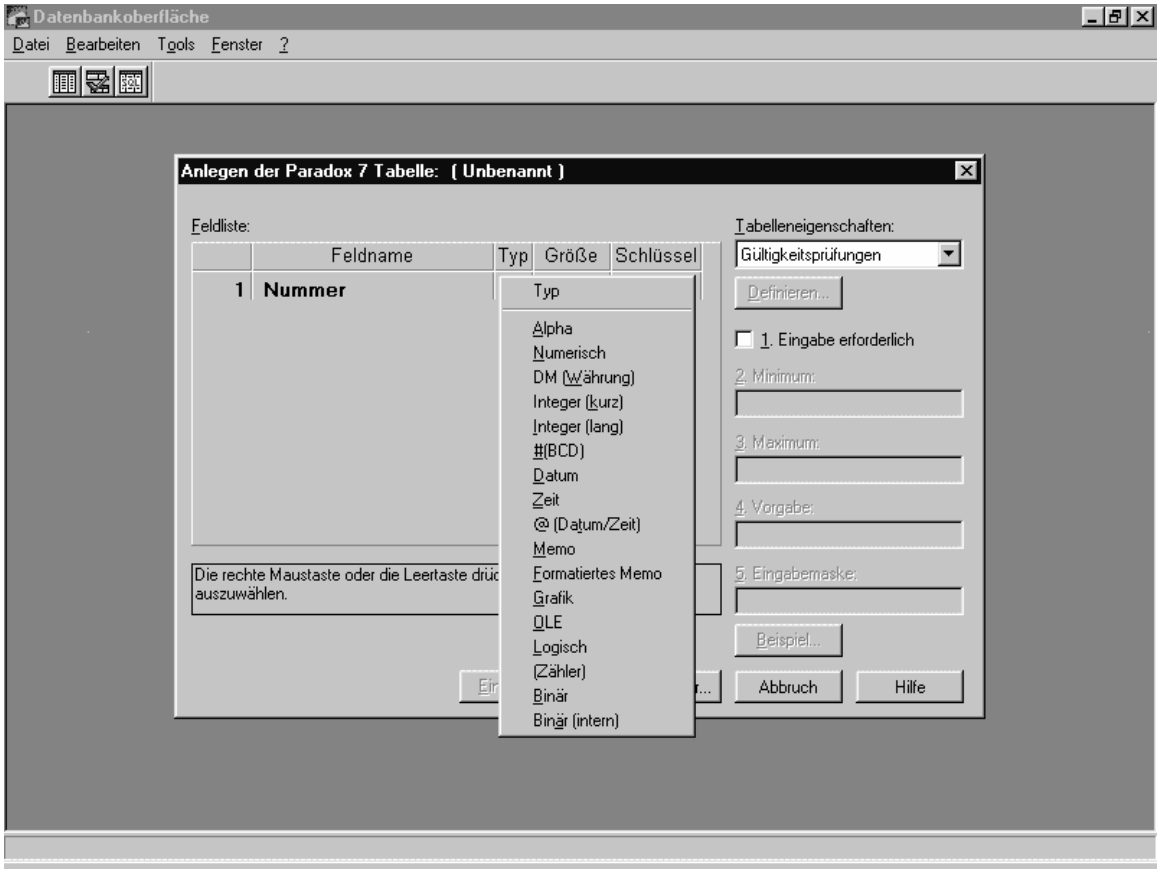
Wie ersichtlich, besteht ein Datensatz der Tabelle aus acht Feldern mit insgesamt 130 Bytes.

Bei den meisten Feldern handelt es sich um Texte (Datentyp *Alpha*). Hier ist es nur erforderlich, eine passende Feldgröße zu wählen (d.h. die maximale Anzahl von Zeichen). Da innerhalb einer *Telefonnummer* Trennstriche auftreten können und der *Postleitzahl* ein Buchstabe zur Länderkennung vorangestellt sein kann, ist es sinnvoll, auch für diese beiden Felder keinen numerischen Datentyp zu verwenden. Das Feld *Geburt* ist vom Typ *Datum*, was einem achtstelligen Datumsformat "TT.MM.JJJJ" entspricht. Das erste Feld *Nummer* wird als *Zähler* definiert.

Wir erzeugen unsere Tabelle in der zu Delphi mitgelieferten Datenbankoberfläche, welche ein universelles Werkzeug zur Erstellung und Verwaltung von Datenbanken darstellt. Zuerst wählen wir das Menü *Tools/Datenbankoberfläche* und dann *Datei/Neu/Tabelle*:



In dem sich dann öffnenden Dialogfenster entscheiden wir uns für das moderne Datenbankformat von *Paradox-7* und bestätigen dies durch einen Mausklick auf die <OK>-Schaltfläche. Nun sind wir schon im Entwurfsfenster und beginnen mit der Eingabe des ersten Feldnamens ***Nummer*** und der Zuweisung des entsprechenden Datentyps *Zähler*. Das Auswahl-Popup-Menü für den Datentyp öffnet sich nach Klick mit der rechten Maustaste auf die Typ-Spalte.



Das erste Datenfeld ist in unserem Fall ein numerisches Feld vom Typ *Zähler*. Bei jeder Eingabe eines neuen Datensatzes wird dieses Zählerfeld automatisch um Eins erhöht (Autoinkrement). Das bringt den großen Vorteil, dass jeder Datensatz dadurch eine eindeutige Nummer erhält. Der kleine Nachteil ist, dass beim Löschen von Datensätzen deren Nummern entfernt werden und somit Lücken in der Nummernfolge auftreten.

Unser erstes Feld *Nummer* kann auch als so genanntes primäres Schlüsselfeld definiert werden. Ein solcher ***primärer Index*** bewirkt eine schnelle, automatische Sortierung der Tabelle. Zu beachten ist dabei, dass im primären Schlüsselfeld nur eindeutige Datenwerte abgelegt werden können, d.h., der gleiche Feldwert kann nicht in verschiedenen Datensätzen vorkommen. Offenkundig ist diese Bedingung in unserem *Zähler*-Feld erfüllt.

Wir fügen nun die anderen Datenfelder *Zuname, Vorname* usw. hinzu (mit <Enter> abschließen).

Das Festlegen der Tabellenstruktur schließt eine spätere Umstrukturierung, z.B. Typänderungen oder das Hinzufügen oder das Löschen von Datenfeldern, nicht aus. Dazu muss nur das Menü *Datei/Öffnen/Tabelle* und *Tabelle/Umstrukturieren* aufgerufen werden.

Dem ***Zuname***-Feld weisen wir im Nachhinein einen ***Sekundärschlüssel*** zu. Nach Anwahl des Menüpunktes *Tabelle/Umstrukturieren* wird in der Rollbox "Tabelleneigenschaften" der Eintrag "Sekundärindizes" und anschließend die Schaltfläche "Definieren" angeklickt. Es öffnet sich nun ein Dialogfenster, in dessen linker Liste das Datenfeld *Zuname* markiert und danach in den rechten Bereich der indizierten Felder kopiert wird (Doppelklick). Alle anderen Standardeinstellungen werden beibehalten. Durch einen Klick auf <OK> gelangt man schließlich in ein Dialogfenster, in welchem ein symbolischer Name für den Index vergeben wird, in unserem Falle soll er *IxFeld* heißen.

Eine Datenbankdatei erhält in ***Paradox-7*** automatisch die Namenserweiterung *.DB*, eine primäre Indexdatei *.PX* und die sekundären Indexdateien *.XGn* bzw. *.YGn*, wobei *n* eine fortlaufende Nummernbezeichnung für den Index ist. Sinn und Zweck solcher, in Indexdateien abgespeicherter Schlüsselfelder ist der schnelle Datenzugriff beim Sortieren und Suchen. In so genannten Memodateien (*.MB*) können zu jedem Datensatz mehrzeilige Texte abgespeichert werden. Eventuelle Eingabeoptionen für die einzelnen Datenfelder sind in *.VAL*-Dateien niedergelegt.

Achtung: Wird kein Primärindex definiert, so können angelegte Sekundärindizes nicht automatisch gewartet werden, d.h. an neue Dateneingaben angepasst werden. Bei der Aktivierung eines nicht gewarteten Sekundärindex erfolgt daher eine temporäre Sperre der Tabelle, sodass die Daten zwar gelesen und ausgedruckt, aber nicht editiert werden können. Wird ein solcher Sekundärindex inaktiviert, dann sind wieder Schreibzugriffe auf die Tabelle möglich.

Sind alle Felder definiert, so klickt man auf die Schaltfläche *Speichern unter*. Dort geben wir als Dateinamen den Namen der Tabelle *dat.db* ein und wählen ganz oben das Datenbankverzeichnis, in welchem sich die Tabelle *dat.db* befindet. Zu erwähnen ist noch die Möglichkeit, lange Verzeichnis- und Dateinamen mit einem kurzen Namen zu bezeichnen (**Alias**). Zur Vergabe von solchen Aliasnamen gibt es einen eigenen Alias-Manager. Dadurch kann der Speicherort der Datenbank, d.h. das Datenbankverzeichnis, bei späteren Anwendungen flexibel gewählt werden. In unserem Projekt verwenden wir keine Aliasnamen. Das Programm "***prodat.exe***" und die Datenbank "***dat.db***" und ihre Indexdateien sollen sich immer im aktuellen Verzeichnis befinden.

- **Der Datenbank-Zugriff**

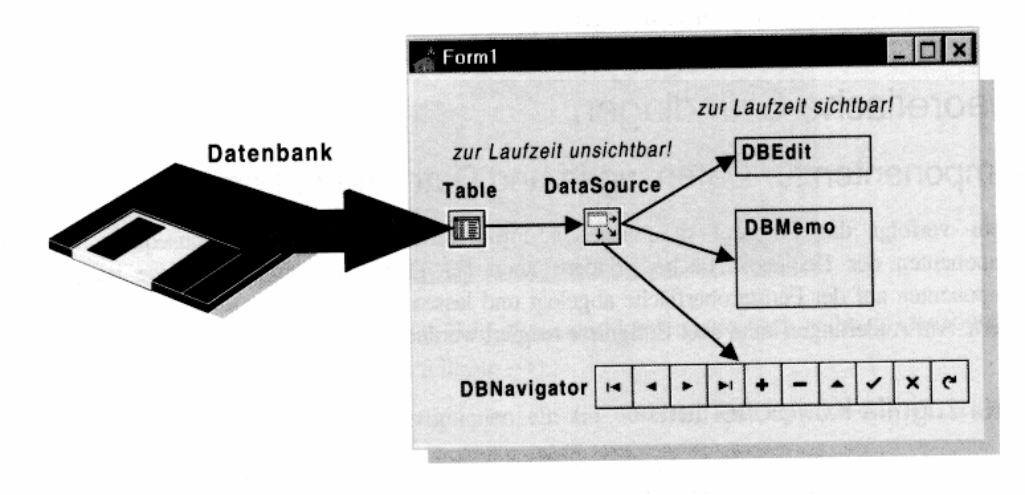
Delphi verfolgt das Konzept der visuellen Programmierung nicht nur konsequent bei den Komponenten der Bedienungsoberfläche, sondern auch bei Datenbank-Objekten. Diese werden wie Komponenten auf dem Formularfenster abgelegt und lassen sich über Eigenschaften und Methoden steuern. Für Datenbankanwendungen in Delphi benötigen wir eine Verbindungsschiene (Pipeline) aus drei Komponenten, welche nachfolgend genau beschrieben werden.

(1) Eine Komponente vom Typ ***TTable***, die von der abstrakten Basisklasse *TDataSet* abgeleitet ist. Diese Komponente stellt die eigentliche Verbindung zwischen einer externen Datenbank und dem Delphi-Programm her. *TTable* repräsentiert die gesamte Datentabelle. Die zusätzliche Komponente ***TQuery*** hingegen greift auf die Daten der Tabelle mit Hilfe von speziellen Abfragebefehlen zu. (SQL = Standard Query Language).

(2) Eine Komponente vom Typ ***TDataSource***, welche die Daten von *TTable* oder *TQuery* übernimmt und an eine visuelle Bedienungskomponente weiterleitet. Sowohl die Komponenten *TTable* und *TQuery* als auch die Komponente *TDataSource* findet man auf der Registerseite *Datenzugriff* (Data Access) der Komponentenpalette.

(3) Eine ***visuelle Komponente***, welche die Daten, die sie von *TDataSource* empfängt, im Formular anzeigt und bearbeiten lässt. Diese visuellen Komponenten sind größtenteils Spezialversionen von Standardkomponenten, die wir schon in anderen Programmen verwendet haben. Ein Beispiel hierfür ist die Komponente *DBText*, welche ähnlich einem *Label* den Inhalt eines Datenbankfeldes anzeigt, jedoch keine Edition zulässt. *DBEdit* dient der Eingabe und Ausgabe von Datenbankfeldern und mit *DBMemo* werden mehrzeilige Textdaten editiert. Eine sehr mächtige Komponente ist *DBNavigator*. Mit seiner Hilfe wird der interne **Datensatzzeiger**, der immer die Adresse des aktuellen Datensatzes enthält, nach Wunsch eingestellt. Dadurch kann sich der Anwender bequem innerhalb der Datenbank vorwärts und rückwärts bewegen (d.h. navigieren).

Die visuellen Komponenten zur Datenbearbeitung findet man in der Komponentenpalette auf der Registerseite *Datensteuerung* (Data Control). Diese sind natürlich zur Laufzeit sichtbar, während die Datenzugriffs-Komponenten zur Laufzeit unsichtbar sind. Die Grafik zeigt den Zusammenhang zwischen Datenzugriffs- und Datensteuerungs-Komponenten.



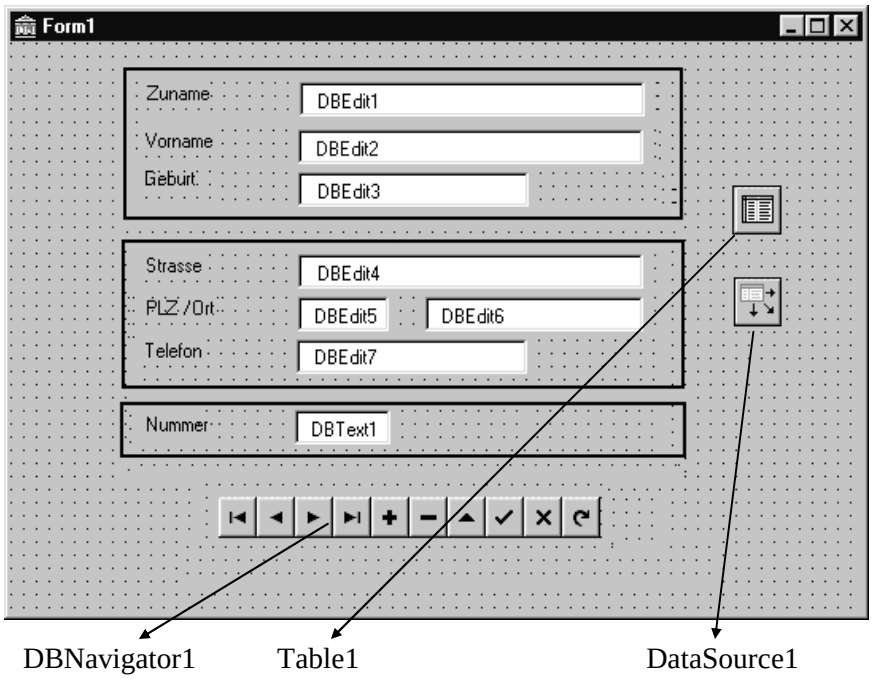
[02] Zugriffs- und Steuerungs-Komponenten

Wir fangen wieder mit dem Platzieren der benötigten Komponenten auf einem Formular an und bestücken dieses Formular mit folgenden Objekten:

(1) **Die Datensteuerungs-Komponenten** *DBText1*, *DBEdit1*, *DBEdit2*, ..., *DBEdit7* (für die acht Datenfelder) und *DBNavigator1* (für die Steuerung des Datensatzzeigers). Sie dienen zum Editieren der Daten und zur Bewegung innerhalb der Datenbank.

(2) **Die Datenzugriffs-Komponenten** *Table1* und *DataSource1* (für die Verbindung der Steuerelemente mit der eigentlichen Datenbank). Später wird noch *TQuery1* zur Verwendung spezieller Datenabfragebefehle (SQL, Standard Query Language) hinzukommen.

(3) **Einige Labels** zur Beschriftung.



Die Anbindung an die Datenbank ist mit wenigen Mausklicks im Objektinspektor erledigt. Wir beginnen mit der ***Table1***-Komponente. Diese Komponente stellt die Verbindung zur Datenbank her. Von der Vielzahl der Eigenschaften sind vorerst nur vier interessant (*Active, DatabaseName, TableName, IndexName*), die übrigen belassen wir auf den Standardwerten. Als Erstes wollen wir die *Active*-Property auf *False* setzen. Dadurch ist am Anfang die Verbindung zur Datenbank getrennt. Nun ist die Zuweisung der *DataBaseName*-Eigenschaft vorzunehmen. Hier geben wir den vollen Namen des Verzeichnisses ein, in welchem die Tabelle *dat.db* abgespeichert wurde. Statt des Datenbankverzeichnisses kann hier auch ein Aliasname eingegeben werden.

Anschließend weisen wir die *TableName*-Property zu. Wir klicken dazu auf das leere Feld in der rechten Spalte des Objektinspektors und öffnen die kleine Rollbox. Es werden nun alle im Datenbankverzeichnis befindlichen Tabellen-Namen angezeigt. In unserem Fall wählen wir *dat.db*. Die Eigenschaft *IndexName* bestimmt den jeweiligen Namen für einen gesetzten Sekundärindex. Er kann auch durchaus leer sein.

Später soll zur Laufzeit am Programmanfang dem *DataBaseName* der aktuelle Verzeichnisname zugewiesen werden, z.B. mit der Anweisung *Table1.DataBaseName := GetCurrentDir*. Dadurch ist gewährleistet, dass auch auf anderen Computern die Datenbank-Tabelle gefunden wird.

Als Nächstes müssen wir die Komponente ***DataSource1*** an *Table1* ankoppeln. Das erfolgt mit Hilfe der *DataSet*-Property. Wir weisen dieser *DataSet*-Property über eine kleine Rollbox die *Table1*-Komponente zu.

Nun ist es noch notwendig, die Datenfelder der Tabelle den gewünschten Datensteuerungs-Komponenten im Formular (***DBEdit***-Feldern) zuzuweisen. *DataSource* übernimmt dabei eine Art "Brückenfunktion". Wir setzen die *DataSource*-Properties aller Datensteuerungs-Komponenten auf *DataSource1* und weisen anschließend die entsprechende *DataField*-Eigenschaft zu, d.h. die Namen der verschiedenen Datenfelder. Wir dürfen auch nicht vergessen, die *DataSource*-Eigenschaft von ***DBNavigator1*** auf die gleiche Art und Weise über *DataSource1* an *Table1* anzubinden.

Setzen wir jetzt die ***Active***-Property von *Table1* wieder auf *True*. Obwohl das Programm noch nicht gestartet ist, sollte der erste Datensatz bereits angezeigt werden. Die *Active*-Property erlaubt es bereits zur Entwurfszeit, die Verbindung zur Datenbank herzustellen. Ist diese Eigenschaft *False*, so ist diese Verbindung inaktiv, andernfalls aktiv (*True*).

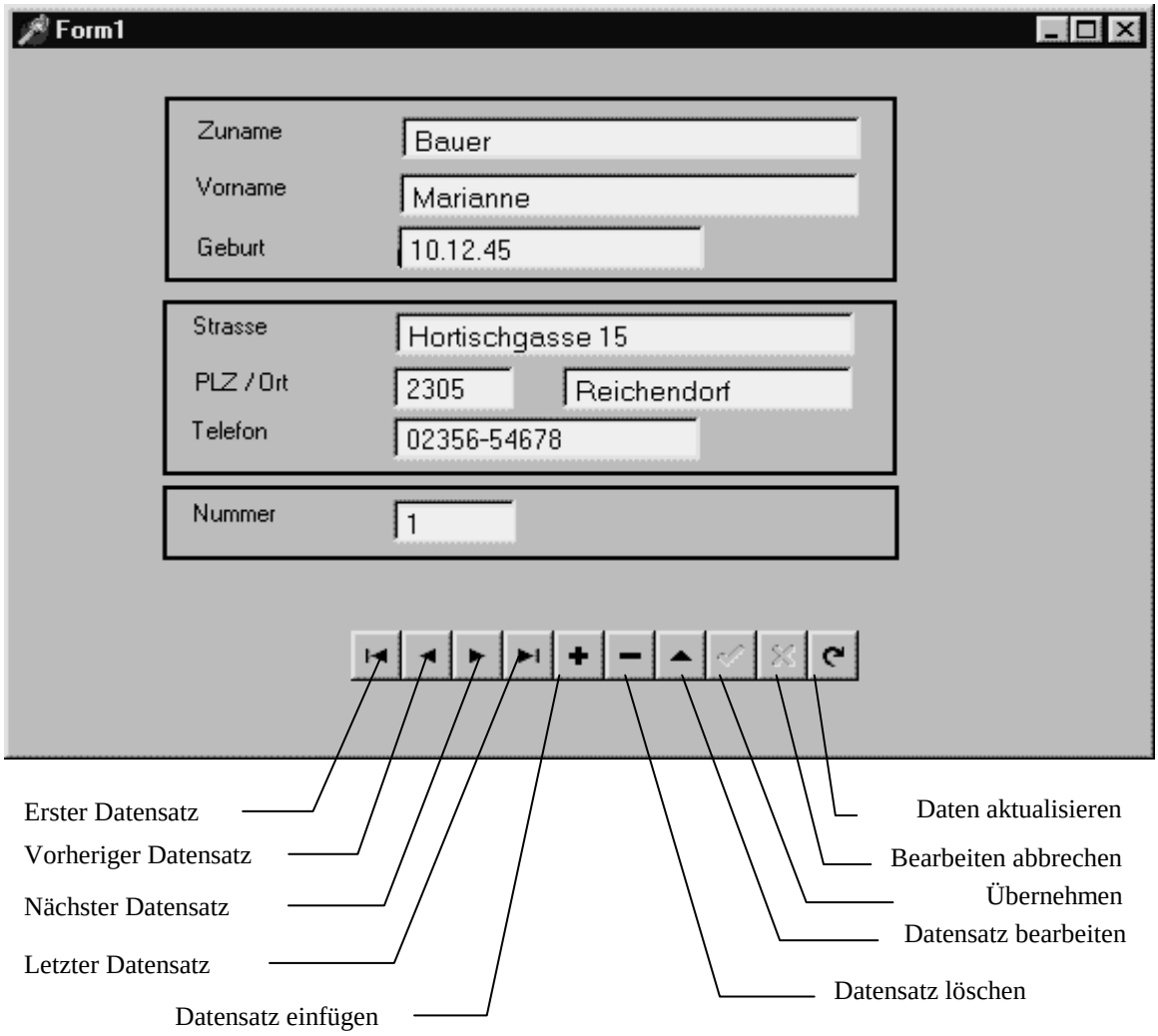
Immer wenn man später *DataBaseName*, *TableName* oder *IndexName* ändern will, sollte man vorher *Active* auf *False* setzen, da sonst eine Fehlermeldung erscheint. Ein Bewegen durch die Datenbank mittels Navigatorschaltflächen ist allerdings erst zur Laufzeit des Programmes möglich. (Anmerkung: Die Funktionen der Navigatorschaltflächen entsprechen Programmroutinen, welche von Delphi automatisch codiert sind).

Hinweis:

Die Erzeugung der Datenbank bzw. Datentabelle erfolgte in unserem Beispiel mit Hilfe der von DELPHI mitgelieferten Datenbankoberfläche BDD (*borland database desktop*). Statt dieses Werkzeug zu benutzen, kann auch der in der *Table*-Komponente integrierte Feld-Designer verwendet werden. Dazu muss im Formular auf das *Table*-Symbol ein Doppelklick mit der Maus ausgeführt werden. Im aufspringenden Feld-Designer wird mit der <*Einfg*>-Taste ein neues Feld mit all seinen gewünschten Eigenschaften (Name, Typ, Größe) angelegt. Schließlich können mit dem Objektinspektor über die Eigenschaft *IndexDefs* von *Table* auch Indizes definiert werden.

Das gesamte Programmprojekt soll unter dem Namen ***prodat.exe*** und die zugehörige Unit unter ***prodat_u.pas*** abgespeichert werden.

Testen wir zum Abschluss das compilierte Programm. Mit dem Datensatznavigator können wir uns nun durch die Datenbank bewegen. Es lassen sich die Datensätze editieren (bearbeiten), löschen oder neu hinzufügen.



Die oben abgebildete *DBNavigator*-Komponente umfasst mehrere mächtige Methoden, welche es ermöglichen, Datensätze einer Tabelle zu editieren und sich in der Tabelle von einem Datensatz zum anderen zu bewegen. Im Folgenden sollen diese Funktionen besprochen werden:

- **Positionieren, Editieren, Einfügen und Löschen**

Auf einen Datensatz wird durch den internen Datensatzzeiger zugegriffen. Dieser Zeiger kann durch die *DatatSet*-Methoden der *Table*-Komponente verändert werden, sodass ein anderer Datensatz aktualisiert wird.

First	zum ersten Datensatz der Tabelle
Prior	zum vorangehenden Datensatz
Next	zum nachfolgenden Datensatz
Last	zum letzten Datensatz

Eng mit obigen Methoden verknüpft sind die booleschen Properties *BOF* und *EOF*, welche den Beginn und das Ende der Datenbank anzeigen. Die zusätzlichen Methoden *DisableControls* und *EnableControls* bewirken die Trennung bzw. die Verbindung der *Table*-Komponente mit den Steuerkomponenten (*DBEdit* usw.). Damit kann die Zeit des physischen Zugriffs auf die Datensätze bei einem sequentiellen Lauf durch die Tabelle optimiert werden.

Die Steuerkomponente *DBNavigator* enthält für alle diese Operationen entsprechende Schalter, die eine gezielte Bewegung (Navigation) innerhalb der Tabelle ermöglichen.

Weitere wichtige Methoden der *Table*-Komponente sind:

Edit	versetzt die Tabelle in den Editiermodus
Post	schreibt Änderungen in die Tabelle
Cancel	macht etwaige Änderungen rückgängig
Append	fügt einen neuen Datensatz am Tabellenende an
Insert	fügt einen neuen Datensatz an der aktuellen Position ein
Delete	löscht den aktuellen Datensatz

Auch für diese Operationen stellt *DBNavigator* entsprechende Schalter zur Verfügung.

[03] Die SQL-Sprache mit *TQuery*

Unter **SQL** (Structured Query Language) versteht man im engeren Sinne eine genormte Abfrage-Sprache für Datenbanken. Sehr oft werden die SQL-Befehle mit Hilfe eines eigenen Übersetzungsprogrammes (Präcompiler) in die Originalsprache eines Datenbanksystems umgeformt. Ein SQL-Befehl entspricht dabei einem Unterprogramm-Aufruf aus einer eigenen Funktions-Bibliothek. Dadurch können mehrere Seiten Quellcode durch einen einzigen Befehl ersetzt werden, was die Kommunikation zwischen Benutzer und Datenbanksystem wesentlich vereinfacht. Man kann den Befehlsvorrat in drei Bereiche einteilen:

DDL	Database Definition Language (Befehle zur Datenbankerzeugung)
DML	Database Manipulation Language (Befehle zur Datenbankbearbeitung)
DSL	Database Security Language (Befehle zum Datenbankschutz)

Im Folgenden werden einige wichtige SQL-Befehle kurz beschrieben.

---
(1) Erzeugung einer neuen Tabelle

Eine Tabelle besteht aus gleich langen Datensätzen (records). Ein Datensatz besteht aus Datenfeldern (fields), welche die Tabellenstruktur bilden.

```
CREATE TABLE "Tabelle"

 ( Feld1 AUTOINC,
   Feld2 CHAR(10),
   Feld3 CHAR(20),
   Feld4 INTEGER,
   Feld5 FLOAT(8,2),
   Feld6 DATE,
   Feld7 BLOB(10,1),
   PRIMARY KEY(Feld1) )
```

"Tabelle" ist der Tabellenname, *FeldXX* sind die einzelnen Feldnamen gefolgt von ihren Feldtypen mit ihren Feldlängen. Wichtige Feldtypen sind:

CHAR(n)	Text-Feld mit der Länge n
INTEGER	Ganzahlen-Feld
FLOAT(n,d)	Gleitkommazahlen-Feld (mit d Dezimalstellen)
MONEY	Währungszahlen-Feld
BOOLEAN	Logik-Feld (True="Wahr",False="Falsch")
DATE	Datums-Feld (TT.MM.JJJJ)
BLOB(n,1)	Memo-Feld (z.B. mit dem Rahmenparameter n=10)
AUTOINC	Satznummern-Feld (wird bei jedem neuen Datensatz automatisch erhöht und kann nur gelesen werden. Mit der zusätzlichen Anweisung *PRIMARY KEY* sollte es immer als Primär-Schlüssel verwendet werden)

---
(2) Bestehende Tabelle laden

```
SELECT Feld1,Feld2,..... ( = Feldliste)
FROM "Tabelle"

SELECT * FROM "Tabelle"   ( = alle Felder)
```

---
(3) Bestehende Tabelle löschen

```
DROP TABLE "Tabelle"
```

---
(4) Daten-Selektion und Daten-Update

Unter Daten-Selektion versteht man die Auswahl bestimmter Datensätze entsprechend einem bestimmten Auswahl-Kriterium (Bedingung).
Unter Daten-Update versteht man die nachträgliche Modifikation der Inhalte bestimmter Datenfelder.

(4.1) Einfache Daten-Selektion

```
SELECT * FROM "Tabelle"
WHERE Selektions-Bedingung
```

Beispiele für Selektions-Bedingungen (Vergleiche):

```
(Feld4 > 160) AND (Feld4 < 180)
Upper(Feld2) <= "P"
Feld2 = "Herbert"
Feld2 LIKE "P%"        (Der Joker % steht für Beliebiges)
Feld3 IN ("Wien","Graz","Linz")
Feld3 IS (NOT) NULL    (NULL steht für unbelegte Felder)
```

(4.2) Datensätze löschen

```
DELETE FROM "Tabelle"
WHERE Trim(Upper(Feld2)) > "P"
```

Hier werden alle Datensätze mit Einträgen in Feld2,
die im Alphabet hinter *"P"* liegen, gelöscht. *Trim*
entfernt alle führenden und nachfolgenden Blanks,
Upper setzt auf Großschrift. Ohne *WHERE* werden
alle Datensätze in der Tabelle gelöscht.

(4.3) Textfelder manipulieren

```
UPDATE "Tabelle"
SET Feld2 = Feld2 + ' ' + Upper(Substring(Feld3 FROM 1 FOR 5))
WHERE ........
```

Hier werden für bestimmte Sätze an Feld2 die ersten
fünf Buchstaben von Feld3 in Großschrift angehängt.

(4.4) Zahlenfelder berechnen (mit +,-,*,/)

```
UPDATE "Tabelle"
SET Feld5 = Feld1 + 2*Feld4
WHERE ........
```

Hier wird für bestimmte Sätze in Feld5 die Summe von
Feld1 und von dem verdoppelten Feld4 abgespeichert.

(4.5) Aggregat-Funktionen

```
SELECT COUNT(FeldXX) AS Anzahl FROM "Tabelle" WHERE ....
SELECT SUM(FeldXX) AS Summe FROM "Tabelle" WHERE ....
SELECT AVG(FeldXX) AS Mittel FROM "Tabelle" WHERE ....
SELECT MIN(FeldXX) AS Minimum FROM "Tabelle" WHERE ....
SELECT MAX(FeldXX) AS Maximum FROM "Tabelle" WHERE ....
```

Diese Funktionen erlauben statistische Auswertungen
von bestimmten Datenfeldern mit optionaler Selektion.
Das Ergebnis steht im virtuellen Anzeigefeld hinter AS.
Virtuelle Speicher-Felder können in reale Daten-Felder
mit dem INSERT-Befehl (siehe 4.8) kopiert werden.

(4.6) Gruppierungen

```
SELECT FeldXX, Count(FeldXX) AS Anzahl
FROM "Tabelle"
GROUP BY FeldXX
```

Hier werden gleichwertige Einträge von *FeldXX* zu Gruppen zusammengefasst und gezählt.

(4.7) Sortierung der Tabelle

```
SELECT * FROM "Tabelle" ORDER BY FeldXX
```

Die Sortierung versetzt die Tabelle in den Anzeigemodus, sodass keine Datenedition möglich ist (read only). Sortieren mit *ORDER BY* läuft schneller, wenn vorher eine sekundäre Indexdatei erstellt wird, wobei beispielsweise der Name *IxField* als interner Bezeichner verwendet wird.

```
CREATE INDEX IxField ON "Tabelle" (FeldXX)
```

Anstelle eines einzelnen Sortierfeldes *FeldXX* kann auch eine Feldliste *Feld1,Feld2,Feld3,....* stehen.

(4.8) Daten an externe Tabelle anfügen

Als Quelle dient eine bestehende Tabelle und als Ziel auch eine bestehende Tabelle, wobei die Reihenfolge der Zielfelder (FeldYn) und Quellfelder (FeldXn) wichtig ist.

```
INSERT INTO "Zieltabelle" (FeldY1, FeldY2, ........)
SELECT FeldX1, FeldX2, ........
FROM "Quelltabelle"
WHERE ........
```

(4.9) Feld in Tabellenstruktur anfügen oder löschen

```
ALTER TABLE "Tabelle" ADD FeldXX CHAR(20)
ALTER TABLE "Tabelle" DROP FeldXX
```

(4.10) Die Verwendung von JOINS

Es können mehrere verschiedene Tabellen verbunden werden (join), wenn sie ein gleichnamiges Schlüsselfeld enthalten. Mit dem Schlüsselfeld wird zwischen den verschiedenen Tabellen eine Beziehung (relation) erzeugt. Zu beachten ist, dass jede Tabelle durch ein Kürzel ersetzt werden kann (z.B *"Tabelle t"*). Ein Feld wird dann mit *"Kürzel.Feldname"* bezeichnet (z.B. *"t.FeldXX"*).

```
SELECT r.ReNr, k.KuNr, k.KuName
FROM rechnung r, kunden k
WHERE r.KuNr = k.KuNr
```

Hier gibt es zwei Tabellen *"kunden"* und *"rechnung"*, die über das Schlüsselfeld *"KuNr"* relationiert sind.

Das Ergebnis von SQL-Abfragen, die Joins, Groups oder Aggregate enthalten, ist ein virtuelles Speicherbild, welches nur angezeigt, aber nicht editiert werden kann.

In DELPHI ermöglicht die Komponente ***TQuery*** den Zugriff auf eine Datenbank mit Hilfe von SQL-Befehlen. Dazu müssen zuerst die Eigenschaften *TQuery.Database* und *TQuery.TableName* entsprechend gesetzt werden. Dann wird mit der Zuweisung *TDataSource.DataSet := TQuery* die Komponente ***TQuery*** mit den Datensteuerungselementen *TDBEdit* und *TDBGrid* verknüpft. Nach diesen Vorarbeiten werden folgende Anweisungen durchgeführt:

```
(1)   S := ' ... ';       //  Zuweisung des SQL-Befehls an String-Variable S
(2)   Query1.Close;       //  Schließen von TQuery
(3)   Query1.SQL.Clear;   //  Liste der SQL-Befehle löschen
(4)   Query1.SQL.Add(S);  //  SQL-Befehl in die Liste einfügen
(5)   Query1.ExecSQL;     //  SQL.Befehl ausführen
(6)   Query1.Close;       //  Schließen von TQuery
```

Wie ersichtlich ist das zentrale Merkmal von ***TQuery*** die Eigenschaft *TQuery.SQL*. Dabei handelt es sich um eine Stringliste, die zur Aufnahme der einschlägigen SQL-Befehle dient. Die Befehlsausführung efolgt mit *TQuery.ExecSQL*. Wird ein "*Select*"-Befehl verwendet, wo der interne Datenbankzeiger (Cursor) positioniert wird, dann sollte die Befehlsausführung mit *TQuery.Open* erfolgen. Zur bequemen Eingabe der Befehle kann eine *TMemo*-Komponente eingerichtet werden, in welcher die Befehle editiert werden. Die Anweisung *TQuery.SQL.Assign(TMemo.Lines)* kopiert dann die SQL-Befehle aus *TMemo* in die SQL-Eigenschaft von ***TQuery***. Auf diese Art und Weise kann in DELPHI ein SQL-Editor programmiert werden.

[04] Direkter Datenzugriff mit *TFields*

Die im System vordefinierte Variable *Fields* ist eine Instanz des Objekttyps *TFields*, der eigentlich aus einem Array von Objekten besteht, welche die einzelnen Datenfelder einer Tabelle beschreiben. *TFields* ist ein Nachfahre von *TDataSet* und ist über Vererbung in *TTable* und *TQuery* vorhanden. So gibt beispielsweise *Table1.FieldCount* die Anzahl der Datenfelder der geöffneten Tabelle an, wobei die Feldindizes bei Null beginnen. Also entspricht *Table1.Fields[0]* genau dem ersten Datenfeld. Die *Fields*-Komponente ermöglicht den direkten Zugriff auf die Datenfelder einer Tabelle. Die wichtigsten *Fields*-Properties sind:

Fields[i].Value	liefert den Inhalt des i-ten Datenfeldes
Fields[i].AsString	konvertiert den Feldinhalt in einen String
Fields[i].FieldName	liefert den Namen des Feldes
Fields[i].DataType	liefert den Datentyp des Feldes
Fields[i].DataSize	gibt die Feldgröße in Bytes an
Fields[i].Alignment	bestimmt die Ausrichtung des Feldes (links-, rechtsbündig, zentriert)
Fields[i].ReadOnly	bestimmt den Lese- oder Schreibzugriff auf das Feld

Ein nützliches Merkmal von *TTable* ist auch *FieldByName*, mit welcher ein Feld der Datenbank nicht über seinen Index, sondern über seinen Namen angesprochen werden kann. Der Datenzugriff erfolgt wie beim Merkmal *Fields*. Beispielsweise liefert *Table1.FieldByName(´Geburt´).AsString* den Inhalt des Datumsfeldes *Geburt* des jeweils aktuellen Datensatzes als String. Die Länge des Feldes *Zuname* in der Datenbank wird mit *Table1.FieldByName(´Zuname´).DataSize* ermittelt.

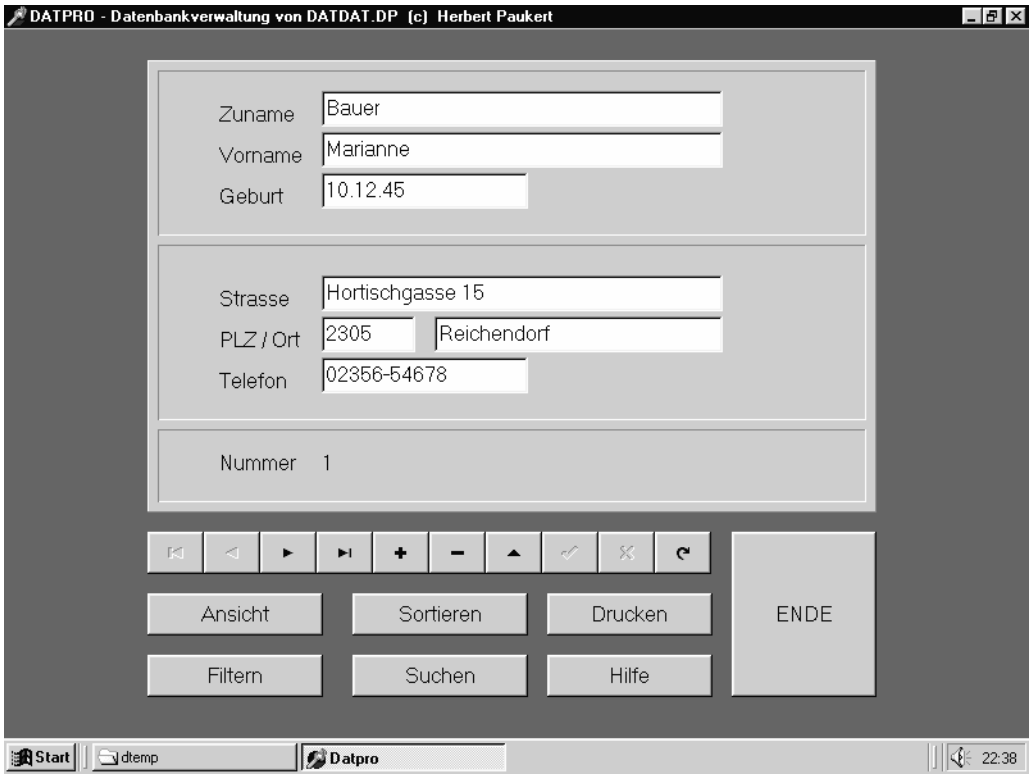
[05] Filtern, Sortieren, Suchen, Drucken

Unser Programmprojekt ***prodat***, welches der Verwaltung der Datentabelle ***dat.db*** dient, soll nun durch zusätzliche Möglichkeiten erweitert werden:

- Ausgabe der Daten in Tabellenform (Komponente *DBGrid*)
- Filtern, Sortieren und Suchen von Daten
- Ausgabe der Daten am Drucker (in Masken- oder Listenform)

Diese Erweiterungen werden durch Verwendung von insgesamt sieben Schaltknöpfen aufgerufen, und zwar *Button1* (Ansicht), *Button2* (Filtern), *Button3* (Sortieren), *Button4* (Suchen), *Button5* (Drucken), *Button6* (Hilfe) und *Button7* (Beenden). Die Schaltknöpfe können im abgebildeten Formular unterhalb des Navigators platziert werden. Im Gegensatz zu den Navigatorfunktionen muss der Programmierer den Code der entsprechenden Routinen selbst schreiben.

Die beiden Abbildungen zeigen die Verwaltungsmöglichkeiten, zunächst in Maskenform (mittels *DBEdit*) und dann in Tabellenform (mittels *DBGrid*).



Im Folgenden sollen die einzelnen Verwaltungsroutinen, die zumeist als Eventhandler der Buttons programmiert sind, ausführlich besprochen werden.

- **Die Routinen <InitData> und <CreateData>**

Wichtig ist die Tatsache, dass beim Erzeugen des Formulares zu Beginn der Anwendung in der Prozedur *FormCreate* das Datenbankverzeichnis immer auf das aktuelle Verzeichnis gesetzt wird (*DatabaseName := GetCurrentDir*). So müssen sich die Datentabelle ***dat.db*** und ihr Verwaltungsprogramm ***prodat*** immer im selben Verzeichnis befinden. Die eigentliche Prozedur, in welcher die Zuweisungen von Anfangswerten (Initialisierungen) erfolgen, heißt *InitData*. Zusätzlich besteht noch die Möglichkeit, unter einem neuen Namen eine neue Tabelle zu erzeugen. Dies geschieht in der Prozedur *CreateData*.

```
var DirName, FilName, IndName: String;

procedure CreateData(QUE: TQuery; FN: String);
{ Neue Tabelle "FN" erzeugen }
var S: String;
begin
  ShowMessage(' Neue Tabelle ' + FN + ' wird erzeugt!');
  S := 'CREATE TABLE "' + FN + '" ' +
       ' (Nummer AUTOINC, ' +
       '  Zuname CHAR(25), ' +
       '  Vorname CHAR(25), ' +
       '  Geburt DATE, ' +
       '  Strasse CHAR(25), ' +
       '  PLZ CHAR(5), ' +
       '  Ort CHAR(25), ' +
       '  Telefon CHAR(15), ' +
       '  PRIMARY KEY(Nummer)) ';
  try
    QUE.Close;
    QUE.SQL.Clear;
    QUE.SQL.Add(S);
    QUE.ExecSQL;
    QUE.Close;
  Except
    on EDBEngineError do Raise;      //  führt alle Fehlerroutinen aus
    on EDatabaseError do Abort;      //  und beendet alle laufenden Prozesse
  end;
end;

procedure InitData;
{ Initialisierungen }
begin
  with Form1 do begin
    DirName := GetCurrentDir;        //  Datenbank-Name
    FilName := 'dat.db';             //  Tabellen-Name
    IndName := 'IxFeld';             //  Sekundär-Index

    DataSource1.DataSet := Query1;
    Query1.Close;
    Query1.DataSource   := NIL;      //  Derzeit noch keine Anbindung!
    Query1.DatabaseName := DirName;

    FilName := InputBox('Name der Datenbank','',FilName);
    if NOT FileExists(DirName + '\' + Filname) then
       CreateData(Form1.Query1,FilName);

    DataSource1.DataSet := Table1;
    Table1.Close;
    Table1.DatabaseName := DirName;
    Table1.TableName    := FilName;
    Table1.Open;
    DBGrid1.Columns.Items[0].ReadOnly := True;
  end;
end;
```

- **Die Routine <Ansicht>**

Der entsprechende Schalter ermöglicht abwechselnd das Umschalten von der Edition in Maskenform auf die Edition in Tabellenform. Im ersten Fall werden entsprechende *DBEdit*-Felder verwendet, im zweiten Fall ein so genanntes Tabellengitter (*DBGrid*). In diesem können zur Laufzeit Daten editiert, aber auch Spalten verschoben und in ihrer Breite verändert werden. Das Objekt *TDBGrid* ist eine mächtige Datensteuerungs-Komponente, welche über *TDataSource* mit *TTable* verknüpft ist.

```
procedure TForm1.Button1Click(Sender: TObject);
{ Ansicht wechseln zwischen Masken- und Tabellenformat }
begin
  DBGrid1.Visible := NOT DBGrid1.Visible;
end;
```

- **Die Routine <Filtern>**

Dadurch wird aus der Grundmenge aller Datensätze eine bestimmte Teilmenge selektioniert. Diese kann dann editiert werden. Beispiel für ein Filterkriterium: *(Zuname < ´P´) and (Ort = ´Wien´).* Als Filteroptionen stehen *foNoPartialCompare* (exakt vergleichend) und *foCaseInsensitiv* (Großschreibung nicht beachtend) zur Verfügung. Ein derart gesetzter Filter wird durch die Eingabe eines Leerstrings als Filterkriterium wieder aufgehoben. Die boolesche *Table*-Eigenschaft *Filtered* gibt an, ob ein Filter aktiv ist oder nicht. Im nachfolgenden Listing ist zwischen Kommentarklammern als Alternative die Erstellung von Filtern mit einer *TQuery*-Komponente programmiert.

```
procedure TForm1.Button2Click(Sender: TObject);
{ Selektion mittels Filter (mit TABLE oder QUERY) }
var S : String;
begin
  S := 'Zuname > ''K''';
  if InputQuery(' Fiterkriterium oder Leer','',S) then begin
     Table1.Filter   := '';
     Table1.Filtered := False;
     Label7.Caption  := '';
     if S = '' then Exit;
{
 // alternative Filter-Erstellung mit QUERY
     Try
       DataSource1.DataSet := Query1;
       S := 'SELECT * FROM "' + FilName + '" WHERE ' + S;
       Query1.Close;
       Query1.SQL.Clear;
       Query1.SQL.Add(S);
       Query1.Open;
     Except
       on EDBEngineError do Raise;      //  führt alle Fehlerroutinen aus
       on EDatabaseError do Abort;      //  und beendet alle laufenden Prozesse
     end;
}
     Try
       Table1.Filter   := S;
       Table1.Filtered := True;
       Table1.First;
       Label7.Caption  := 'FILT';
     Except
       ShowMessage(' Filterfehler! ');
     end;
   end;
end;
```

- **Die Routine <Sortieren>**

Hier wird die *TTable*-Komponente verwendet, um eine sekundäre Indexdatei zum schnellen Sortieren der Tabelle zur Laufzeit anzulegen. Das leistet die Methode *TTable.AddIndex*.

Es ist zu beachten, dass durch die Existenz eines Primärindex (*Nummer*) die Sekundärindizes vom System gewartet werden. Somit kann auch nach Aktivierung eines Sekundärindex in der Tabelle editiert werden. Die Anpassungen an neue Dateneingaben erfolgen dann immer automatisch.

Im nachfolgenden Programmlisting sind *FilName* und *IndName* die globalen Stringvariablen, welche den Dateinamen (*dat.db*) und den Indexnamen (*IxFeld*) enthalten. Erwähnenswert ist noch, dass nicht nur ein einfacher Index wie *Zuname*, sondern auch ein kombinierter Index wie beispielsweise *Zuname; Vorname* gesetzt werden kann. Arbeitet man statt mit *TTable* mit *TQuery*, dann muss das Trennzeichen zwischen den Feldnamen ein Komma sein. Das Programm ist so gestaltet, dass zwar verschiedene Schlüsselfelder ausgewählt werden können – diese aber immer nur unter demselben Indexnamen (*IxFeld*) abgespeichert werden. Dadurch ist immer nur eine sekundäre Indexdatei im Datenbankverzeichnis vorhanden.

Im nachfolgenden Listing ist zwischen Kommentarklammern als Alternative die Erstellung von sekundären Indizes mit einer *TQuery*-Komponente programmiert.

```
procedure TForm1.Button3Click(Sender: TObject);
{ Sortieren mittels Sekundär-Index (mit TABLE oder QUERY) }
var S : String;
begin
  S := 'Zuname; Vorname';
  if InputQuery(' Feldliste zum Sortieren oder Leer','',S) then begin
     Table1.IndexName := '';
     Label8.Caption:= '';
     if S = '' then Exit;
     Try
       Table1.Close;
       Table1.AddIndex(IndName,S,[ixCaseInsensitive]);
       Table1.Open;
{
       // alternative Index-Erstellung mittels QUERY
       DataSource1.DataSet := Query1;
       S := 'CREATE INDEX ' + IndName + ' ON "' + FilName + '" ('+ S + ')';
       Query1.Close;
       Query1.SQL.Clear;
       Query1.SQL.Add(S);
       Query1.ExecSQL;
       Query1.Close;
       DataSource1.DataSet := Table1;
}
       Table1.IndexName := IndName;
       Table1.First;
       Label8.Caption:= 'SORT';
     Except
       ShowMessage(' Indexfehler! ');
     end;
   end;
end;
```

- **Die Routine <Suchen>**

Beim Suchen wird zunächst das Datenfeld und dann der Suchbegriff eingegeben. Hierauf werden alle Datensätze der Tabelle durchlaufen und im Datenfeld der Suchbegriff gesucht. Bei Erfolg wird der Datensatzzeiger automatisch auf den gefundenen Datensatz gestellt.

Als Suchoptionen stehen wie beim Filtern wieder *loPartialKey* und *loCaseInsensitive* zur Verfügung. Im nachfolgenden Listing wird zum Suchen die *Table*-Methode ***Locate*** verwendet. Dieser werden Feldname, Suchbegriff und eventuelle Optionen als Parameter übergeben und sie liefert nach ihrer Ausführung *True*, wenn die Suche erfolgreich war, andernfalls liefert sie *False*. Im nachfolgenden Listing ist zwischen Kommentarklammern als Alternative die Suche mit Hilfe einer sekundären Indexdatei programmiert.

```
procedure TForm1.Button4Click(Sender: TObject);
{ Suchen mittels Locate (TABLE) }
var S,S1,S2: String;
    P: Integer;
begin
  S := 'Zuname, Zen';
  if InputQuery(' Feldname, Suchbegriff oder Leer','',S) then begin
     if S = '' then Exit;
     P  := Pos(',',S);
     if Not (P>0) then begin
        ShowMessage(' Suchfehler! ');
        Exit;
     end;
     S1 := Trim(Copy(S,1,P-1));
     S2 := Trim(Copy(S,P+1,Length(S)));
     try
       Table1.First;
       if Table1.Locate(S1,S2,[loCaseInsensitive,loPartialKey]) then
          ShowMessage('Suchbegriff gefunden!')
       else
          ShowMessage(' Suchbegriff NICHT gefunden! ');
     except
     end;
{
     // alternative Suche mittels vorher erstelltem Sekundär-Index
     try
       Table1.IndexName := IndName;
       Table1.FindNearest([S2]);
     except
     end;
}
  end;
end;
```

- **Die Routine <Drucken>**

Wenn die Ansicht im Maskenformat vorliegt, so wird der aktuelle Datensatz zur Gänze gedruckt. Liegt Tabellenformat vor, muss zuerst ein Datenfilter gesetzt und eine gewünschte Feldliste eingegeben werden. Dann werden alle Sätze der Datenbank durchlaufen und nur die Felder der gefilterten Sätze in Tabellenform ausgedruckt. Die Routinen heißen *PrintRecord* und *PrintTable*.

In der Routine ***PrintRecord*** wird zunächst der Drucker zur Textdatei gemacht und zum Schreiben geöffnet. In einer Zeile wird links der Feldname und rechts daneben der Feldinhalt ausgedruckt, dazwischen ein Tabulatorsprung. Am Ende muss der Drucker als Textdatei geschlossen werden.

```
procedure PrintRecord;
{ Ausdruck des aktuellen Datensatzes in Maskenform }
var PText: TextFile;
    S    : String;
    N,P  : Integer;
begin
  AssignPrn(PText);
  Rewrite(PText);
  with Form1.Table1 do begin
    N := FieldCount;
    Writeln(PText);
    Writeln(PText,'  ---------------------------------------');
    For P := 0 to N-1 do begin
        S := Fields[P].FieldName;
        S := '  ' + S + ':  ' + #9;
        Write(PText,S);
        S := Fields[P].AsString;
        Writeln(PText,S);
    end;
    Writeln(PText,'  ---------------------------------------');
  end;
  CloseFile(PText);
end;
```

Die Routine ***PrintTable*** setzt einen Datenfilter voraus. Um alle Datensätze zu drucken, kann ein Filterkriterium der Art *Zuname > ´A´* verwendet werden. Außerdem müssen die auszudruckenden Felder in Form einer Liste eingegeben werden – beispielsweise *Zuname, Vorname, Telefon*. Der Kern der Routine besteht in einer Schleife, welche die gesamte Tabelle durchläuft und wo von jedem, im Filter befindlichen Datensatz die gewählten Felder ausgedruckt werden. Den genauen Programmcode findet man im nachfolgenden Listing.

```
procedure PrintTable;
{ Ausdruck der gefilterten Records in Listenform }
var PText: TextFile;
    A,D,S,T: String;
    P: Integer;
begin
  With Form1 do begin
    if Table1.Filter = '' then begin
       ShowMessage(' KEIN Datenfilter gesetzt! ');
       Exit;
    end;
    S := 'Zuname, Vorname, Telefon';
    if InputQuery(' Liste der Feldnamen oder Leer ','',S) then begin
      if S = '' then  Exit;
      A := Trim(UpperCase(S));
      AssignPrn(PText);
      Rewrite(PText);
      Writeln(PText);
      Writeln(PText,' ----------------------------------------');
      With Table1 do begin
        DisableControls;
        First;
        While not EOF do begin
          S := A;
          P := Pos(',',S);
          while P > 0 do begin
            T := Trim(Copy(S,1,P-1));
            S := Trim(Copy(S,P+1,Length(S)));
            D := ' ' + Trim(FieldByName(T).AsString);
            Write(PText,D,',');
            P := Pos(',',S);
          end;
          D := ' ' + Trim(FieldByName(S).AsString);
          Writeln(PText,D);
          Next;
        end;
        EnableControls;
      end;
      Writeln(PText,' ----------------------------------------');
      Writeln(PText);
      CloseFile(PText);
    end;
  end;
end;
```

[06] Komplettes Listing von "*prodat*"

unit prodat_u;

```
{ Unit zur Datenbankverwaltung "prodat" }
{ der Datenbank "dat.db" (c) H. Paukert }

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls, DBCtrls, DBTables, Db, Mask, Grids, DBGrids, Printers;

type
  TForm1 = class(TForm)
    Panel1: TPanel;           // Container für Datenedition
    Bevel1: TBevel;
    Bevel2: TBevel;
    Bevel3: TBevel;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;

    DBText1: TDBText;         // Nummer (Zähler als Primärindex)
    DBEdit1: TDBEdit;         // Zuname
    DBEdit2: TDBEdit;         // Vorname
    DBEdit3: TDBEdit;         // Geburt
    DBEdit4: TDBEdit;         // Strasse
    DBEdit5: TDBEdit;         // PLZ
    DBEdit6: TDBEdit;         // Ort
    DBEdit7: TDBEdit;         // Telefon

    DBGrid1: TDBGrid;
    DBNavigator1: TDBNavigator;

    DataSource1: TDataSource;
    Table1: TTable;
    Query1: TQuery;

    Button1: TButton;         // Ansicht (Maske/Tabelle}
    Button2: TButton;         // Filtern
    Button3: TButton;         // Sortieren
    Button4: TButton;         // Suchen
    Button5: TButton;         // Drucken
    Button6: TButton;         // Hilfe
    Button7: TButton;         // Programm beenden

    Memo1: TMemo;             // MemoBox für Hilfe

    procedure FormActivate(Sender: TObject);
    procedure FormKeyPress(Sender: TObject; var Key: Char);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    procedure Button7Click(Sender: TObject);

  private { Private declarations }
  public  { Public declarations }
end;

var Form1: TForm1;
```

```
implementation
{$R *.DFM}

var  DirName : String;           // Datenbank-Verzeichnis
     FilName : String;           // Datenbank-Tabelle
     IndName : String;           // Sekundär-Index

procedure FitForm(F :TForm);
// Anpassung des Formulares an die Monitorauflösung
const SW: Integer = 1024;
      SH: Integer = 768;
      FS: Integer = 96;
      FL: Integer = 120;
var   X,Y,K: Integer;
      V0,V : Real;
begin
  with F do begin
    Scaled := True;
    X  := Screen.Width;
    Y  := Screen.Height;
    K  := Font.PixelsPerInch;
    V0 := SH / SW;
    V  := Y / X;
    if V < V0 then ScaleBy(Y,SH)
              else ScaleBy(X,SW);
    if (K <> FL) then ScaleBy(FL,K);
    WindowState := wsMaximized;
  end;
end;

procedure CreateData(Q: TQuery; FN: String);
// Neue Tabelle "FN" erzeugen
var S : String;
begin
  ShowMessage(' Neue Tabelle ' + FN + ' wird erzeugt !');
  S := ' CREATE TABLE "' + FN + '" ' +
       ' (Nummer AUTOINC, ' +
       '  Zuname CHAR(25), ' +
       '  Vorname CHAR(25), ' +
       '  Geburt DATE, ' +
       '  Strasse CHAR(25), ' +
       '  PLZ CHAR(5), ' +
       '  Ort CHAR(25), ' +
       '  Telefon CHAR(15), ' +
       '  PRIMARY KEY(Nummer)) ';
  with Form1 do begin
    try
      Q.Close;
      Q.SQL.Clear;
      Q.SQL.Add(S);
      Q.ExecSQL;
      Q.Close;
    except
      on EDBEngineError do Raise;      //  beendet alle Fehlerroutinen
      on EDatabaseError do Abort;      //  und bereinigt alle Prozeduren
    end;
  end;
end;

procedure InitData;
// Initialisierungen
begin
  with Form1 do begin
    DirName := GetCurrentDir;
    FilName := 'dat.db';
    IndName := 'IxFeld';

    DataSource1.DataSet := Query1;
    Query1.Close;
    Query1.DataSource   := NIL;
    Query1.DatabaseName := DirName;
```

```
    FilName := InputBox('Name der Datentabelle','',FilName);
    if NOT FileExists(DirName + '\' + Filname) then
       CreateData(Form1.Query1,FilName);

    DataSource1.DataSet := Table1;
    Table1.Close;
    Table1.DatabaseName := DirName;
    Table1.TableName    := FilName;
    Table1.Open;
    DBGrid1.Columns.Items[0].ReadOnly := True;
  end;
end;

procedure TForm1.FormActivate(Sender: TObject);
// Formular und Komponenten initialisieren
begin
  Session.NetFileDir := ExtractFilePath(Application.EXEName);
  FitForm(Form1);
  InitData;
  KeyPreview := True;
  Printer.Canvas.Font.Name := 'Courier New';
  Printer.Canvas.Font.Size := 12;
end;

procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
// Ermöglicht den Eingabeabschluss mit <ENTER>
begin
   if (key = #13) and (Sender = DBEdit1) then DBEdit2.SetFocus;
   if (key = #13) and (Sender = DBEdit2) then DBEdit3.SetFocus;
   if (key = #13) and (Sender = DBEdit3) then DBEdit4.SetFocus;
   if (key = #13) and (Sender = DBEdit4) then DBEdit5.SetFocus;
   if (key = #13) and (Sender = DBEdit5) then DBEdit6.SetFocus;
   if (key = #13) and (Sender = DBEdit6) then DBEdit7.SetFocus;
   if (key = #13) and (Sender = DBEdit7) then DBEdit1.SetFocus;
end;

procedure TForm1.Button1Click(Sender: TObject);
{ Ansicht wechseln zwischen Masken- und Tabellenformat }
begin
  DBGrid1.Visible := NOT DBGrid1.Visible;
end;

procedure TForm1.Button2Click(Sender: TObject);
// Selektion mittels Filter (mit TABLE oder QUERY)
var S : String;
begin
  S := 'Zuname > ''K''';
  if InputQuery(' Fiterkriterium oder Leer','',S) then begin
     Table1.Filter   := '';
     Table1.Filtered := False;
     Label7.Caption  := '';
     if S = '' then Exit;
{
     // alternative Filter-Erstellung mit QUERY
     Try
       DataSource1.DataSet := Query1;
       S := 'SELECT * FROM "' + FilName + '" WHERE ' + S;
       Query1.Close;
       Query1.SQL.Clear;
       Query1.SQL.Add(S);
       Query1.Open;
     Except
       on EDBEngineError do Raise;      //  beendet alle Fehlerroutinen
       on EDaradaseError do Abort;      //  und bereinigt alle Prozeduren
     end;
}
     Try
       Table1.Filter   := S;
       Table1.Filtered := True;
       Table1.First;
       Label7.Caption  := 'FILT';
```

```
     Except
       ShowMessage(' Filterfehler ! ');
     end;
   end;
end;


procedure TForm1.Button3Click(Sender: TObject);
// Sortieren mittels CreateIndex (mit TABLE oder QUERY)
var S : String;
begin
  S := 'Zuname; Vorname';
  if InputQuery(' Feldliste zum Sortieren oder Leer','',S) then begin
     Table1.IndexName := '';
     Label8.Caption:= '';
     if S = '' then Exit;
     Try
       Table1.Close;
       Table1.AddIndex(IndName,S,[ixCaseInsensitive]);
       Table1.Open;
{
     // alternative Index-Erstellung mittels QUERY
     // ACHTUNG: Mehrere Felder müssen hier durch Kommas getrennt werden !

     DataSource1.DataSet := Query1;
     S := 'CREATE INDEX ' + IndName + ' ON "' + FilName + '" ('+ S + ')';
     Query1.Close;
     Query1.SQL.Clear;
     Query1.SQL.Add(S);
     Try
       Query1.ExecSQL;
       Query1.Close;
       DataSource1.DataSet := Table1;
}
       Table1.IndexName := IndName;
       Table1.First;
       Label8.Caption:= 'SORT';
     Except
       ShowMessage(' Indexfehler ! ');
     end;
   end;
end;


procedure TForm1.Button4Click(Sender: TObject);
// Suchen mittels Locate (TABLE)
var S,S1,S2: String;
    P: Integer;
begin
  S := '';
  if InputQuery(' Feldname, Suchbegriff oder Leer','',S) then begin
     if S = '' then Exit;
     P  := Pos(',',S);
     if Not (P>0) then begin
        ShowMessage(' Suchfehler ! '); Exit;
     end;
     S1 := Trim(Copy(S,1,P-1));
     S2 := Trim(Copy(S,P+1,Length(S)));
     Try
       Table1.First;
       if Table1.Locate(S1,S2,[loCaseInsensitive,loPartialKey]) then
          ShowMessage(' Suchbegriff gefunden ! ')
       else
          ShowMessage(' Suchbegriff nicht gefunden ! ')
     Except
     end;
  end;
end;
```

```
procedure PrintRecord;
// Ausdruck des aktuellen Datensatzes in Maskenform
var PText: TextFile;
    S    : String;
    N,P  : Integer;
begin
    AssignPrn(PText);
    Rewrite(PText);
    with Form1.Table1 do begin
       N := FieldCount;
       Writeln(PText);
       Writeln(PText,'  ----------------------------------------');
       For P := 0 to N-1 do begin
           S := Fields[P].FieldName;
           S := '  ' + S + ':  ' + #9;
           Write(PText,S);
           S := Fields[P].AsString;
           Writeln(PText,S);
       end;
       Writeln(PText,'  ----------------------------------------');
    end;
    CloseFile(PText);
end;

procedure PrintTable;
{ Ausdruck der gefilterten Records in Tabellenform }
var PText  : TextFile;
    A,D,S,T: String;
    P      : Integer;
begin
  With Form1 do begin
    if Table1.Filter = '' then begin
       ShowMessage(' KEIN Datenfilter gesetzt ! ');
       Exit;
    end;
    S := 'Zuname, Vorname, Telefon';
    if InputQuery(' Liste der Feldnamen oder Leer ','',S) then begin
      if S = '' then  Exit;
      A := Trim(UpperCase(S));
      AssignPrn(PText);
      Rewrite(PText);
      Writeln(PText);
      Writeln(PText,' ----------------------------------------');
      With Table1 do begin
        DisableControls;
        First;
        While not EOF do begin
          S := A;
          P := Pos(',',S);
          while P > 0 do begin
            T := Trim(Copy(S,1,P-1));
            S := Trim(Copy(S,P+1,Length(S)));
            D := ' ' + Trim(FieldByName(T).AsString);
            Write(PText,D,',');
            P := Pos(',',S);
          end;
          D := ' ' + Trim(FieldByName(S).AsString);
          Writeln(PText,D);
          Next;
        end;
        EnableControls;
      end;
      Writeln(PText,' ----------------------------------------');
      Writeln(PText);
      CloseFile(PText);
    end;
  end;
end;
```

```
procedure TForm1.Button5Click(Sender: TObject);
// Drucken von Records in Listen- oder Masken-Format
begin
  if DBGrid1.Visible then PrintTable
                     else PrintRecord;
end;

procedure TForm1.Button6Click(Sender: TObject);
// Hilfsinformationen ein-/ausblenden
begin
  Memo1.Visible := Not Memo1.Visible;
end;

procedure TForm1.Button7Click(Sender: TObject);
// Programm beenden
begin
   Application.Terminate;
end;

end.
```

Wichtiger Hinweis:

Der Befehl ***"Session.NetFileDir := ExtractFilePath(Application.EXEName);"*** in der Prozedur ***"TForm1.FormActivate"*** bewirkt, dass die von der BDE automatisch angelegte Hilfsdatei PDOXUSRS.NET immer im aktuellen Anwenderverzeichnis abgespeichert wird. Diese Datei enthält nur einfache Informationen über Netzwerk-Benutzer der Paradox-Datenbank.

[07] Zusätzliche Programm-Erweiterungen

Der interessierte Leser kann auf diesem Grundgerüst eines Verwaltungsprogrammes für Datenbanken noch weitere Routinen erstellen. Vier Erweiterungsmöglichkeiten seien vorgeschlagen: ***Erstens,*** die Darstellung von Grafikdateien, welche beispielsweise eingescannte Fotos enthalten. ***Zweitens,*** der Einbau von Memofeldern, in denen mehrzeilige Texte für jeden Datensatz editiert werden können. ***Drittens***, die Erstellung spezieller Druckreports. ***Viertens***, die Synchronisation (Relation) verschiedener Datentabellen.

Um eine Grafikdatei im Formular darzustellen, ist zunächst beim Formularentwurf eine *Scrollbox* einzurichten. In diese ist dann eine ***Image***-Komponente mit Ausrichtung als *Client* einzupassen.

Beide sollen nur dann sichtbar werden, wenn ein entsprechender Button gedrückt wird. Dann wird aus einem eigenen Datenfeld, wo der *Name* der gewünschten Grafikdatei eingegeben werden muss, dieser *Name* gelesen und mit *Image.Picture.LoadFromFile(Name)* die Grafik am Bildschirm dargestellt. Ist der Button als Wechselschalter programmiert, dann bewirkt eine neuerliche Betätigung des Buttons, dass die Sichtbarkeit von *Scrollbox* und *Image* auf *False* gesetzt wird und die normale Formularoberfläche wieder zur Verfügung steht.

Die Einrichtung eines ***DBMemo***-Feldes bewirkt die Anlage einer Memodatei für mehrzeilige Texte pro Datensatz. Mit Hilfe eines entsprechenden Buttons kann das Memofeld ein- oder ausgeblendet werden. Im Memofeld wird der Text beliebig editiert. In der *Lines*-Property steht er dann zeilenweise zur Verfügung und kann beispielsweise ausgedruckt oder in die Zwischenablage kopiert werden.

Zur Erstellung spezieller Druckreports stellt Delphi leistungsstarke Werkzeuge zur Verfügung, so genannte ***QuickReport***-Komponenten, die hier aber nicht näher erläutert werden.

Zwei Datentabellen ***Table1*** (Master-Tabelle) und ***Table2*** (Detail-Tabelle) enthalten jeweils ein gleichnamiges Schlüsselfeld. Mit dessen Hilfe können sie so verknüpft werden, dass bei einem Wechsel des Datensatzes in der ersten Tabelle automatisch in der zweiten Tabelle auf jene Datensätze zugegriffen wird, welche im Schlüsselfeld jenen Wert aufweisen, welcher auch im Schlüsselfeld des Datensatzes der ersten Tabelle enthalten ist. Um eine solche Synchronisation (Relation) zu erreichen, muss erstens die Eigenschaft *MasterSource* von *Table2* auf den Tabellennamen von *Table1* gesetzt werden. Zweitens muss die Eigenschaft *Masterfields* von *Table2* auf das gemeinsame Schlüsselfeld der beiden Tabellen gesetzt werden. Dadurch sind die Tabellen in gewünschter Weise miteinander verknüpft. Im Programm-Formular müssen beide Tabellen mit ihren Zugriffs- und Steuerungs-Komponenten eingerichtet werden.

Zum Abschluss sei noch erwähnt, dass mit der Codierung und Testung eines Programmes bei komplexeren Delphi-Applikationen die Arbeit noch nicht erledigt ist. Zur Verwendung von einfachen Programmen auf Computern, wo Delphi nicht installiert ist, genügt durchaus die bloße Weitergabe des entsprechend compilierten EXE-Files. Bei Datenbankprogrammen hingegen müssen neben dem EXE-File auch die eigentliche Datenbank mit ihren Indexdateien und vor allem die *BDE* (Borland Database Engine) mitgeliefert werden. Ohne das Management der *BDE* sind Datenbankzugriffe nicht möglich. Um das Programm lauffähig auf einem Fremdcomputer zu verwenden, muss eine so genannte Installations-Diskette bzw. Installations-CD erstellt werden. Mit dieser kann dann das Programm und die *BDE* installiert werden. Eine solche Diskette oder CD wird mit Hilfe des *InstallShield*-Programmes *ISX.EXE* erzeugt. Dieses befindet sich auf der Delphi-CD im Verzeichnis *\isxpress*. Leider ist die Hilfestellung von *InstallShield* nicht übersichtlich: Es müssen insgesamt 16 Etappen durchlaufen werden, welche nur wenig erklärt sind.

[08] Dynamische Erzeugung von Datenbanken

In den bisherigen Beispielen wurde eine Datentabelle mit fester Struktur vor der eigentlichen Programmierung mit Hilfe der BDD (*borland database desktop*) entworfen. Jetzt wollen wir eine Datentabelle mit frei bestimmbarer Struktur erst zur Laufzeit des Programmes erzeugen, was man als dynamisch bezeichnet. Dazu müssen im Formular die beiden Zugriffskomponenten *TTable* und *TQuery* und die beiden Steuerungskomponenten *TDBGrid* und *TDBNavigator* definiert werden. *TDataSource* dient der Verbindung. Der Datenbankname *DirName* soll so wie bisher das aktuelle Verzeichnis sein, der Tabellenname *FilName* soll am Programmanfang leer sein. Die vier Schalter *<Tabelle laden>*, *<Tabelle erzeugen>*, *<Tabelle/Maske wechseln>* und *<Programm beenden>* dienen der bequemen Handhabung. Die beiden Prozeduren ***CreateData*** und ***CreateField*** stellen dabei die Kernroutinen dar.

In der ersten Routine ***CreateData*** wird zunächst ein Tabellenname und dann in einem Memofeld der SQL-Befehl zur Erzeugung einer Tabelle mit der gewünschten Datenstruktur eingegeben.

Der SQL-Befehl wird mit Hilfe der *TQuery*-Komponente ausgeführt. Damit ist die dynamische Erzeugung der Tabelle abgeschlossen und es können jetzt mit Hilfe von *TDBNavigator* neue Daten in *TDBGrid* in Tabellenform eingegeben werden. Das Grundgerüst einer solchen Routine findet der Leser im vorangehenden Abschnitt.

In der zweiten Routine ***CreateField*** soll dynamisch zur Laufzeit anstelle des Tabellengitters eine einfache Editiermaske erzeugt werden. Dazu werden in einer *Scrollbox* untereinander *Labels* und *Editfelder* für die einzelnen Datenfelder der Tabelle positioniert. Die *Scrollbox* ist dazu notwendig, weil ja die Anzahl der Datenfelder für den Bildschirm zu groß sein könnte. Im nachfolgenden Listing wird das Grundgerüst einer solchen Routine vorgestellt. Als Parameter wird die Anzahl der Datenfelder *FMax* übergeben. Diese wird durch *FMax := Table1.FieldCount* ermittelt. In der letzten Routine ***DeleteField*** werden die dynamisch erzeugten Komponenten wieder entfernt.

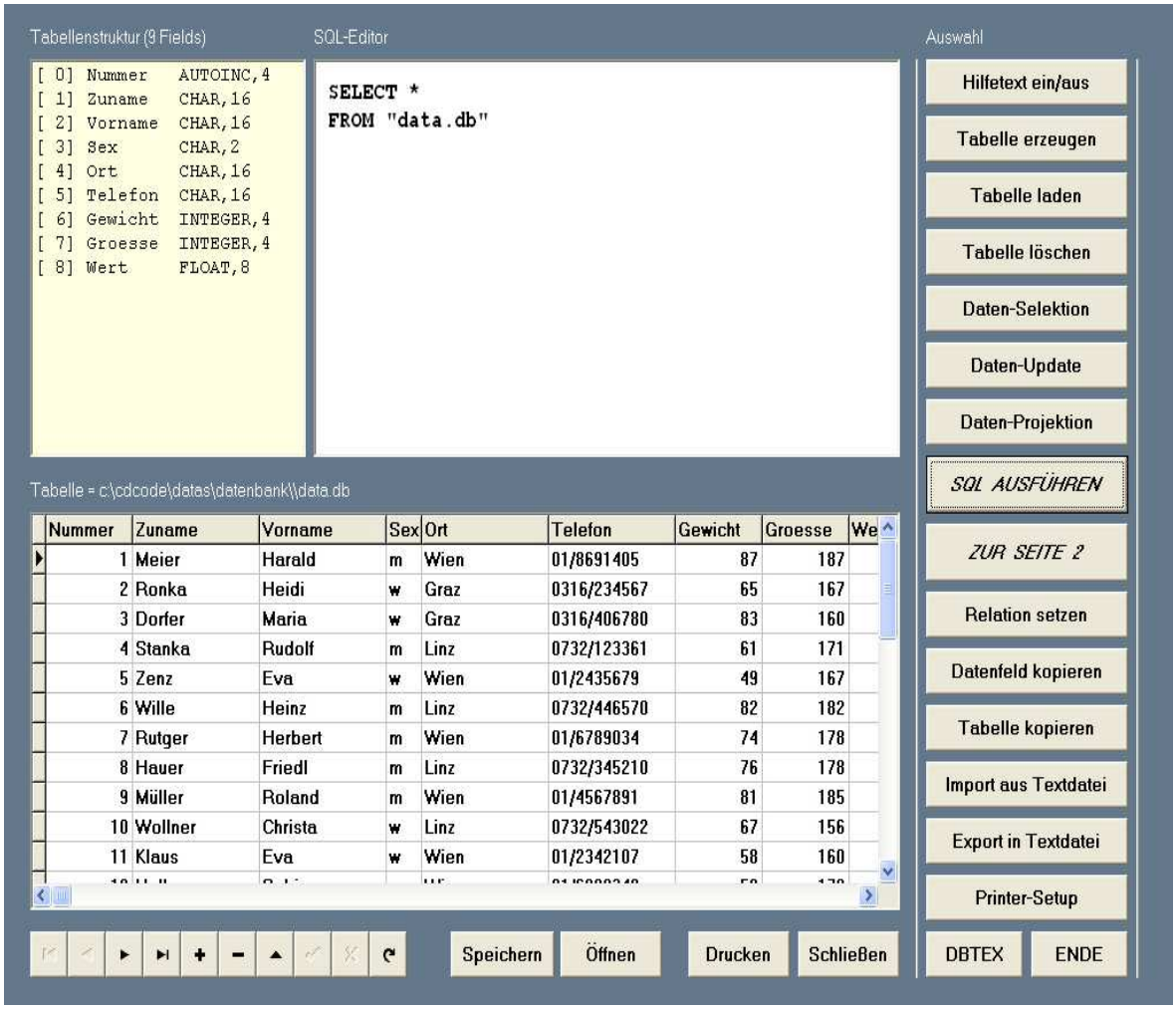
```
var EdLabel: TLabel;
    EdField: TDBEdit;
    FMax: Integer;


procedure CreateField(FMax: Integer);
{ Dynamische Erzeugung einer Editiermaske für Datenfelder }
var links,oben,breit,hoch,Len,L,I: Integer;
    S: String;
begin
  with Form1 do begin
    L := 1;
    for I := 0 to FMax-1 do begin
      Len := Length(Table1.Fields[I].FieldName);
      if Len > L then L := Len;
    end;
    S := 'AA';
    for I := 1 to L do S := S + 'A';
    L := Canvas.TextWidth(S);
    links := 100; oben := 100;
    breit := 400; hoch := 40;
    for I := 0 to FMax-1 do begin
      if TLabel(FindComponent('Label'+IntToStr(I))) = NIL then begin
        EdLabel := TLabel.Create(Form1);
        EdLabel.Parent := ScrollBox1;
        EdLabel.Name := 'Label'+IntToStr(I);
        EdLabel.Caption := '[' + IntToStr(I) + '] ' + Table1.Fields[I].FieldName;
        EdLabel.SetBounds(links,oben+(I-1)*Hoch,breit,hoch);
      end;
      if TEdit(FindComponent('DBEdit'+IntToStr(I))) = NIL then begin
        EdField := TDBEdit.Create(Form1);
        EdField.Parent := ScrollBox1;
        EdField.Name := 'DBEdit'+IntToStr(I);
        EdField.DataSource := DataSource1;
        EdField.DataField  := Table1.Fields[I].FieldName;
        EdField.SetBounds(L+links,oben+(I-1)*Hoch,breit,hoch);
      end;
    end;
  end;
end;

procedure DeleteField(FMax: Integer);
{ Dynamisches Entfernen einer Editiermaske für Datenfelder }
var I : Integer;
begin
  with Form1 do begin
    for I := 0 to FMax-1 do begin
      EdLabel := TLabel(FindComponent('Label'+IntToStr(I)));
      EdLabel.Free;
      EdField := TDBEdit(FindComponent('DBEdit'+IntToStr(I)));
      EdField.Free;
    end;
  end;
end;
```

[09] Ein universeller SQL-Editor (*dbsql*)

Das Programm "***dbsql***" stellt einen universellen SQL-Editor dar. Mit seiner Hilfe können in einer *RichEdit*-Komponente alle einschlägigen SQL-Befehle eingegeben und dann ausgeführt werden. Eine komfortable Menüführung ermöglicht es dem Anwender, viele häufig gebrauchte Routinen der Datenbank-Verwaltung aufzurufen. Filtern, Sortieren, Suchen und Updaten von Daten werden mit der SQL-Sprache durchgeführt.



Das Programm "***dbsql***" verwirklicht die meisten der auf den vorangehenden Seiten beschriebenen Techniken, beispielsweise können Memo-Felder editiert und Grafikdateien für jeden Datensatz angezeigt werden. Zuätzlich wird eine Editiermaske dynamisch erzeugt. Auch besteht u.a. die Möglichkeit, die Struktur einer bestehenden Tabelle in eine leere Tabelle zu kopieren (copy structure).

Damit mit den Programmen "***prodat***" und "***dbsql***" arbeiten kann, muss vorher ein Installationsprogramm "***dbsetup***" aufgerufen werden. So werden alle wichtigen Dateien der Borland Database Engine (BDE) und alle notwendigen Paradox-Treiber auf der Festplatte installiert.

Eine ausführliche Beschreibung des Programms "***dbsql***" findet man in der Datei "***dbhelp.pdf***", und außerdem können mit "***demodat.db***" und "***mediadat.db***" Demo-Datenbanken verwendet werden.

Hinweis: Das Programm "***dbsql***" legt automatisch die Hilfsdatei PDOXUSRS.NET im aktuellen Verzeichnis an.