C++Builder
Datenbankprogrammierung

PEARSON Education

ADDISON-WESLEY

Michael Ebner

C++Builder Datenbankprogrammierung

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Bonn • Reading, Massachusetts • Menlo Park, California • New York • Harlow, England
Don Mills, Ontario • Sydney • Mexico City • Madrid Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme
C++Builder-Datenbankprogrammierung / *Ebner, Michael*
Bonn; Reading, Mass. [u.a.]: Addison-Wesley-Longman, 1997
ISBN 3-8273-1295-7

Lektorat: Tomas Wehren, Judith Stevens, Bonn
Korrektorat: Johannes Gerritsen, Emmerich-Elten
Satz: Michael Ebner, Berlin – gesetzt aus der Palatino 10 Punkt
Produktion: Claudia Lucht, Bonn
Belichtung, Druck und Bindung: Bercker Graphischer Betrieb, Kevelaer
Umschlaggestaltung: Hommer DesignProduction, Haar bei München

Das verwendete Papier ist aus chlorfrei gebleichten Rohstoffen hergestellt und alterungsbeständig. Die Produktion erfolgt mit Hilfe umweltschonender Technologien und unter strengsten Auflagen in einem geschlossenen Wasserkreislauf unter Wiederverwendung unbedruckter, zurückgeführter Papiere. Text, Abbildungen und Programme wurden mit größter Sorgfalt erarbeitet. Verlag, Übersetzer und Autoren können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Inhaltsverzeichnis

Vorwort

Zu den herausragendsten Eigenschaften von C++Builder gehört sicherlich die gelungene Datenbankanbindung, insbesondere der einfache Zugriff auf Client-Server-Systeme.

Der Inhalt dieses Buches gliedert sich in vier Teile: Der erste Teil umfaßt die Kapitel 1 bis 7 und gibt eine Einführung in die Thematik. Hier werden die Datenzugriffs- und die Datensteuerungskomponenten behandelt, die *QuickReport*-Komponenten vorgestellt, außerdem die Besonderheiten bei der Installation und der Erstellung der Online-Hilfe erklärt.

Im zweiten Teil – er umfaßt die Kapitel 8 bis 12 – werden zwei größere Beispielprogramme für Desktop-Datenbanken vorgestellt. Der dritte Teil – Kapitel 13 bis 15 – befaßt sich mit InterBase und Client-Server-Systemen. Im vierten und letzten Teil, in den Kapitel 16 und 17, wird das Datenmodell für die Mitgliederverwaltung einer fiktiven Partei entwickelt.

Wenn Sie sich bislang noch nicht oder nur wenig mit Datenbankprogrammierung befaßt haben, dann sollten Sie das Buch von vorne bis hinten durcharbeiten; sollten Client-Server-Anwendungen für Sie nicht relevant sein, dann könnten Sie nach Kapitel 12 aufhören. Die Kapitel selbst bilden in sich abgeschlossene Einheiten, so daß Leser, die schon weitgehend mit der Datenbankprogrammierung vertraut sind, in beliebiger Reihenfolge lesen können.

Sollten Sie Fehler in diesem Buch entdeckt haben, dann schicken Sie mir bitte eine EMail – die Korrekturen werden dann auf meinen Web-Pages abrufbar sein. Die Adresse meine Homepage lautet http://members.aol.com/MEbner1969/home.htm.

Berlin, Oktober 1997

Michael Ebner

MEbner1969@aol.com

1 Datenbanken

In diesem ersten Kapitel soll eine allgemeine Einführung zu Datenbanken gegeben werden. Wir wollen Fragen behandeln wie: *Was versteht man unter relationalen Datenbanken? Worin besteht der Unterschied zwischen Desktop- und Client-Server-Datenbanken? Wie erstellt man ein zweckmäßiges Datenmodell?*

1.1 Was sind Datenbanken?

Es gehört wohl zu den universellen Gesetzmäßigkeiten, daß (nicht nur in der Philosopie) die einfachsten Begriffe am schwersten zu definieren sind. So beginnt ein (unbestreitbar ernstzunehmendes) Buch über Datenbanken mit der Definition: *Eine Datenbank ist eine Sammlung von nicht-redundanten Daten, die von mehreren Applikationen benutzt werden.*

Auf meiner Festplatte sind im Verzeichnis *C:\TIFFS* viele Bild-Dateien. Weil diese die Festplatte schon genug belegen, sind sie jeweils nur einmal vorhanden, also nicht-redundant. Zugreifen kann ich darauf mit dem *PhotoStyler*, mit *Corel PhotoPaint* und dem Layout-Programm *PageMaker*. Also eine Datenbank? Auf der anderen Seite gibt es die Mitglieder-»Datenbank« einer bundesdeutschen Partei (ich will keine Namen nennen ...), die ist (»leider, leider ...«) alles andere als *nicht-redundant*, und zugegriffen wird darauf nur mit der eigens dafür erstellten Anwendung, multi-user-fähig ist das System »zur Freude aller Beteiligten« ohnehin nicht. Also keine Datenbank?

Was soll man von einem Programm halten, das sich *Datenbank-Compiler* nennt? Und wenn Sie in ein Computergeschäft gehen, auf eine Paradox-, Access- oder dBase-Schachtel zeigen und fragen, was das denn für ein Programm sei, dann wird man Ihnen wohl sagen: *Dies ist eine Datenbank*.

Und was lehrt uns dieses? *Datenbank* ist ein Begriff, der ziemlich beliebig auf Datenbanksysteme, Datenbank-Management-Systeme und Datenbestände bezogen wird. Überlassen wir also die Definition den Wissenschaftlern, verwenden wir den Begriff ebenso beliebig, wie dies von der Mehrheit getan wird, und vermeiden wir ihn, wenn wir uns präzise ausdrücken wollen.

1.1.1 Historisches

Die ersten beiden Generationen von Datenbanken (wenn man diese schon so nennen möchte) waren sogenannte File-Systeme (die erste Generation auf Band, die zweite auf Platte). In diesen File-Systemen wurden die Datensätze nacheinander abgespeichert.

Damit konnte man beispielsweise Adressen speichern und auch wieder zurückerhalten, aber bei allem, was darüber hinausging, fingen die Probleme an. Wenn man einen bestimmten Datensatz suchen wollte, dann konnte man nur alle Datensätze auslesen und vergleichen, ob der jeweilige Datensatz den gestellten Bedingungen entsprach.

Bei den Systemen der ersten Generation war dabei noch nicht einmal ein sogenannter wahlfreier Zugriff möglich: Wollte man den 365. Datensatz auslesen, dann wurde das Band bis zur Dateianfangsmarke (BOF, *begin of file*) zurückgespult und dann Datensatz für Datensatz ausgelesen, bis man den 365. hatte. Bei den Systemen der zweiten Generation hatte man dann wenigstens Festplatten, auf den gewünschten Datensatz konnte man hier (mehr oder minder) direkt zugreifen. Bei der Suche nach bestimmten Kriterien war man dann aber immer noch auf die sequentielle Suche angewiesen (dies ist man häufig auch heute noch).

Redundanz, Inkonsistenz und Integrität

Bei diesen Systemen machten unter anderem Redundanz und Inkonsistenz sowie Integritätsprobleme Sorgen. Nehmen wir als Beispiel die Auftragsverwaltung eines Versandhauses, welche wir zu diesem Zweck sehr grob vereinfachen wollen, und zwar zu einer Kunden- und einer Auftragsdatei.

Zur Auftragsdatei gehören lauter Datensätze über laufende oder abgeschlossene Aufträge; ein Datensatz enthälthäufig Angaben über Bestelldatum, Anzahl, Bestellnummer, Bezeichnung, Einzel- und Gesamtpreis der gelieferten Waren und natürlich über den Kunden. Hierfür gibt es prinzipiell zwei Möglichkeiten:

- Die eine ist, daß sämtliche Kundendaten aus der Kundendatei in die Auftragsdatei kopiert werden. Ein und dieselbe Adresse ist also doppelt vorhanden, man spricht hier von *Redundanz*. So etwas vermehrt nicht nur den Bedarf an Speicherplatz, es führt auch zur *Inkonsistenz*, wenn an nur einem Datensatz Änderungen durchgeführt werden. Nehmen wir einmal an, der Kunde zieht um, meldet dies der Firma, und diese ändert entsprechend die Kundendatei.

  Nun hat der Kunde aber in der Umzugshektik vergessen, die Rechnung aus der letzten Lieferung zu begleichen. Die Buchhaltung untersucht alle Rechnungen, ob die denn auch beglichen seien, findet den Vorgang und schickt an die alte Adresse eine Mahnung (welche natürlich zurückkommt, weil der Kunde an den Nachsendeauftrag auch nicht gedacht hat). Die Buchhaltung

ist nun auch nicht »blöde« und schaut in der Kundendatei beispielsweise unter *Stefan Meier* nach, den es vielleicht siebenmal gibt. Ohne Kundennummer hat man nun ein Problem.

Variation der Geschichte: Zusammen mit der neuen Adresse wurde eine neue Kundennummer vergeben, weil man daraus beispielsweise die Filiale erkennen soll, welche den Kunden zu betreuen hat.

- Die andere Möglichkeit ist, daß man in der Auftragsdatei nur die Kundennummer speichert und sich der Rechner bei Bedarf einfach die nötigen Adressdaten aus der Kundendatei holt. Redundanz wird somit (in diesem Punkt) vermieden, bei den heutigen relationalen Datenbanken macht man das im Prinzip auch nicht anders.

  Nun bittet beispielsweise *Stefan Meier*, in Zukunft keinen Katalog mehr zu erhalten, die Adresse wird nun aus der Kundendatei gelöscht. Wenn die Buchhaltung nun eine Mahnung adressieren möchte, dann hat sie nur die Kundennummer – und somit auch ein Problem.

Prinzipiell wäre es möglich, die Anwendungsprogramme so zu erstellen, daß diese Probleme erkannt und vermieden werden. Nun ist es allerdings häufiger der Fall, daß für ein und denselben Datenbestand immer wieder neue Anwendungsprogramme verwendet werden. In diese jedesmal von neuem die erforderlichen »Sicherungen« einzufügen ist unökonomisch (und sowieso fehleranfällig). Es hat sich deshalb durchgesetzt, daß die Anwendungsprogramme nicht direkt auf den Datenbestand zugreifen, sondern über ein spezielles, für den Anwender »unsichtbares« Programm, welches (unter anderem) diese Sicherheitsmaßnahmen durchführt. Dieses Programm nennt man *Datenbank-Management-System (DBMS)*.

Hierarchische und Netzwerk-Datenbanken

Durch diese Trennung von Anwendungs- und Datenverwaltungsprogramm entstanden die Datenbanken der dritten Generation (von manchen werden sie auch die »ersten echten« Datenbanken genannt). Vertreter dieser Spezies sind beispielsweise die hierarchischen Datenbanken oder die Netzwerk-Datenbanken. Solche Datenbanken sind stellenweise noch auf Großrechnern im Einsatz, werden aber heutzutage bei Neuprogrammierungen nicht mehr verwendet. Für unsere Zwecke können wir diese Datenbanken ohnehin vergessen – mit dem C++Builder werden Sie nicht darauf zugreifen können. Eine Beschreibung dieser Datenbanken soll aus diesem Grund hier unterbleiben.

1.1.2 Desktop- und Client-Server-Datenbanken

Der gerade verwendete Begriff *Netzwerk-Datenbank* bezieht sich nicht darauf, daß die Datenbank über ein Netzwerk von mehreren Anwendern gleichzeitig genutzt werden kann – Großrechnersysteme haben immer eine Client-Server-Architektur. Lassen Sie uns auch diese Begriffe klären.

Stand-Alone-Datenbank

Am wenigsten Kopfzerbrechen macht eine *Stand-Alone*-Datenbank, welche zu den *Desktop-Datenbanken* gezählt wird. Die Daten befinden sich auf einem Arbeitsplatzrechner, auf die Daten kann immer nur ein Anwender mit immer nur einer Anwendung zugreifen. Es ist zwar möglich, daß über ein Netzwerk auch Anwender B auf die Daten zugreift, aber nur dann, wenn Anwender A seine Applikation geschlossen hat. Probleme, die dadurch entstehen, daß zwei Anwender zur selben Zeit am selben Datensatz etwas ändern wollen, können schon prinzipiell nicht auftreten; bei jeder größeren Datenbank wird aber der eine Arbeitsplatz zum Nadelöhr.

File-Share-Datenbank

Moderne Netzwerke bieten die Möglichkeit, daß mehrere Anwender auf ein und dieselbe Datei zugreifen. Auf diese Weise ist es auch möglich, daß mit zwei Datenbankanwendungen auf dieselbe Datenbankdatei zugegriffen wird. Diese Version der *Desktop*-Datenbank nennt man *File-Share*-Datenbank, und damit ist schon echter Multi-User-Betrieb möglich.

Das ganze hat jedoch (unter anderem) einen entscheidenden Nachteil: Die Datenverarbeitung erfolgt auf den Arbeitsplatzrechnern; für Abfragen muß jeweils der ganze Datenbestand (der jeweiligen Tabellen) zum Arbeitsplatzrechner transferiert werden, dementsprechend hoch ist die Belastung (und somit niedrig die Performance) des Netzwerks.

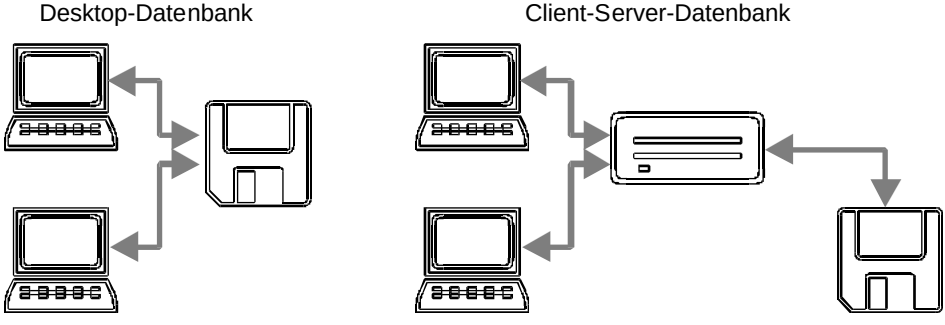


Bild 1.1: Unterschied zwischen Desktop- und Client-Server-Datenbank

Client-Server-Datenbank

Einen anderen Ansatz verfolgen *Client-Server*-Datenbanken: Zugriff auf die Dateien des Datenbestandes hat dort nur der Datenbank-Server (nicht zu verwechseln mit dem File-Server eines Netzwerkes!), der die Arbeitsplatzrechner bedient. Anfragen werden also nicht auf dem Arbeitsplatzrechner bearbeitet, sondern auf

dem Datenbank-Server (der hardwaremäßig entsprechend ausgerüstet sein sollte), es werden dann nur die Ergebnisse an die Arbeitsplatzrechner geschickt.

Ein Beispiel soll den Unterschied zur *File-Share*-Datenbank erläutern: Nehmen wir an, in einem großen Versandhaus werden Mahnungen geschrieben. Um Redundanzen zu vermeiden, sind in der Tabelle *Rechnungen* nur die Kundennummern gespeichert, beim Erstellen der vielleicht hundert Mahnungen müßten ebenso viele Kundenadressen in die Standardtexte (»sicher haben Sie übersehen ...«) eingefügt werden. Eine entsprechende SQL-Anweisung könnte lauten:

```
SELECT a.vornamen || " " || a.nachnamen AS namen,
    a.straße, a.plz || " " || a.ort AS wohnort,
    r.datum, r.betrag, r.betrag + 5 AS mahnsumme
  FROM adressen a, rechnungen r
  WHERE (r.kunde = a.nummer)
    AND (r.datum < :Mahngrenze) AND (r.bezahlt IS NULL)
```

(Es macht nichts, wenn Sie diese Anweisung noch nicht ganz verstehen, das lernen Sie in Kapitel 3.) Bei einer *File-Share*-Datenbank würden nun (um einmal Größenordnungen zu schätzen) 300 000 Rechnungsdatensätze und 100 000 Kundendatensätze zum Arbeitsplatzrechner transferiert; das können gut und gerne 20 MByte an Daten sein.

Bei einem *Client-Server*-System würde der Server die Anfrage selbst bearbeiten und dann rund 10 kByte zum Arbeitsplatzrechner übertragen. Dies würde einer Beschleunigung um den Faktor 2000 entsprechen, und bei manchen Abfragen sind die Verhältnisse noch viel krasser.

Hinzu kommt, daß *Client-Server*-Systeme meist viel besser auf die Erfordernisse des Betriebs mit großen Datenmengen ausgerichtet sind. Dazu gehören dezidierte Zugangskontrollen und Zugriffsrechte oder – so banal sich das auch anhören mag – die Fähigkeit, bei laufendem Betrieb ein Backup zu ziehen. (Stellen Sie sich vor, Sie gehen an einen Bankautomaten nachts um 2.07 Uhr, und das Display meldet *Zwischen 2.00 Uhr und 2.13 Uhr keine Auszahlung, von unserem Server wird ein Backup gezogen.)*

Fazit der ganzen Problematik: Wenn Sie mit wirklich großen Datenmengen zu tun haben, dann scheuen Sie nicht den Mehraufwand (und die Mehrkosten) für eine Client-Server-Datenbank, letztlich lohnt sich das immer. Bei angenommen drei Jahren Systemlaufzeit (sehr vorsichtig geschätzt) und zehn daran beschäftigten Mitarbeitern fallen allein rund zwei Millionen DM an Lohn- und Lohnnebenkosten an. Daran gemessen sind die Mehrkosten für C-S-Systeme wirklich »Peanuts«.

1.2 Relationale Datenbanken

Der Begriff *relationale Datenbanken* geht zurück auf einen Artikel von E.F. Codd: *A Relational Model of Data for Large Shared Data Banks*, der 1970 veröffentlich wurde. Inzwischen sind von Codd 333 Kriterien erstellt worden, die ein Datenbank-Management-System erfüllen muß, damit es sich relational nennen »darf«.

Nach der Ansicht von Experten erfüllt derzeit kein einziges System alle 333 Kriterien. »In der Praxis« wird ein DBMS *relational* genannt, wenn es der »Philosophie« dieser Kriterien gerecht wird und die wesentlichsten Bedingungen erfüllt.

1.2.1 Begriffe

Man kann nicht über relationale Datenbanken sprechen, ohne vorher einige Begriffe zu klären.

Relation

Eine *Relation* ist eine Tabelle. Relationale Datenbanken könnte man als »auf Tabellen basierende Datenbanken« bezeichnen. Sämtliche Daten werden in *Relationen*, also in Tabellen gespeichert.

Eine *Relation* (Tabelle) ist eine logische Verbindung von einer festen Anzahl von *Attributes* (Spalten) und einer variablen Anzahl von *Tuples* (Zeilen, Reihen). *Relationen* werden wir später noch ausführlicher behandeln.

Domain

Eine *Domain* ist ein Wertebereich, ähnlich dem, was in C ein Typ ist. Bei relationalen Datenbanken sind die Domains allerdings *atomar*, sie lassen sich also nicht weiter zerteilen (zumindest nicht sinnvoll).

Beispielsweise ist der *Name* eines Menschen nicht atomar, weil er sich in *Vorname* und *Nachname* (und ggf. akademische Grade) zerlegen läßt. *Vorname* und *Nachname* sind dann allerdings atomare Werte, also *Domains*.

Bei einer Datenbank sind stets einige *Domains* vordefiniert, meist hält man sich dabei an Bereiche, welche das binäre Zahlenmodell vorgibt (Integerzahlen). Es ist aber auch möglich, eigene *Domains* zu definieren. Hierzu zwei Beispiele (als SQL-Befehle, wir werden dies später behandeln):

```
CREATE DOMAIN dnachnamen AS VARCHAR(20);

CREATE DOMAIN dabteilungsnummer AS CHAR(3)
  CHECK  (VALUE = "000" OR
    (VALUE > "0" AND VALUE <= "999")
    OR VALUE IS NULL)
```

Beim ersten Beispiel fragt man sich, warum hierfür eigens eine eigene Domain definiert wird, schließlich könnte man die Angabe der Länge auch bei der Spaltendefinition machen. Im Prinzip ist das richtig. Nun gibt es allerdings Referenzen, also den Verweis auf andere Tabellenspalten. Dafür ist es zwingend erforderlich, daß beide *Attributes* (Spalten) derselben *Domain* unterliegen. Auch dies ist nun weiter kein Problem, es werden alle Nachnamen-*Attributes* als VARCHAR(20) definiert.

Nun wird später eine Änderung nötig; beispielsweise kommen Doppelnamen in Mode, und die Nachnamen lauten nun beispielsweise *Leuthäuser-Schnarrenberger* oder *Freiherr Marschall von Biberstein* (den Namen gibt's wirklich). Prinzipiell ist das auch noch kein Problem: Es müssen halt alle *Attributes* (Spalten) entsprechend umdefiniert werden. Sollte aber dabei auch nur eine Spalte vergessen werden, dann arbeitet die Datenbank nicht mehr korrekt. Deshalb ist es sicherer, auf eigens definierte *Domains* zu verweisen, denn dann muß die Änderung nur an einer einzigen Stelle vorgenommen werden.

Bei der Definition von *Domains* können auch etwas kompliziertere Wertebereiche vorgegeben werden, wie das zweite Beispiel zeigt. Die Definiton von *Domains* werden wir uns in Kapitel 14.1 (es handelt vom *Local InterBase Server*) noch näher ansehen.

NULL

Ein Ausdruck, der bei Einsteigern immer wieder für Verwirrung sorgt, ist der Wert NULL. NULL ist eben nicht gleich *Null*. Vielleicht hätte man eine andere Bezeichnung dafür finden sollen (beispielweise UNK für *unknown, unbekannt*), jedenfalls heißt der Wert nun NULL und steht für unbekannte oder nicht zutreffende Werte.

Während unbekannte Werte leicht vorstellbar sind, mag man sich zunächst fragen, was man sich unter nicht zutreffenden Werten eigentlich vorzustellen hat. Dazu ein Beispiel: Nehmen wir einmal an, in einer *Relation* (Tabelle) seien die Plätze von Konferenzräumen aufgelistet, dazu die Teilnehmernummer der Person, für welche der Platz reserviert ist. Nun hat aber der Konferenzraum mehr Plätze als Teilnehmer, und so gibt es Plätze, zu deren *Attribut* (Spalte) *Teilnehmer* es keinen zutreffenden Wert gibt. Auf diesen Plätzen sitzt auch nicht der Teilnehmer *0* oder der Teilnehmer *x*, sondern für diese Spalte gibt es einfach keinen zutreffenden Wert.

Dieser Wert NULL macht übrigens aus einer zweiwertigen Logik eine dreiwertige, was einigermaßen verwirrend sein kann.

Degree und Kardinalität

Die Zahl der *Attributes* (Spalten) einer *Relation* (Tabelle) nennt man *Degree* (Ausdehnungsgrad). Relationen mit nur einer Spalte nennt man *unär*. So etwas kann durchaus sinnvoll sein, beispielsweise als Liste der gesetzlichen Feiertage – in der Regel wird (den Rechner) nur interessieren, an welchem Datum der Feiertag ist, und nicht, wie dieser genannt wird. Eine *Relation* mit einem *Degree* von *zwei* nennt man dementsprechend *binär*.

Die Zahl der *Tuples* (Datensätze) wird *Kardinalität* genannt. Während ein *Degree* von Null keinen Sinn hat, kann es durchaus Relationen mit null Tuples, also leere Tabellen geben. Als Beispiel sei die Tabelle der momentan offenen Posten genannt – wenn alle Kunden ihre Rechnungen bezahlt haben.

1.2.2 Keys (Schlüssel)

Im Zuge der sogenannten »Normalisierung« (wir werden das später klären) werden viele Tabellen entstehen. Um diese Tabellen wieder richtig zusammensetzen zu können, aber auch, um die gewünschten Datensätze überhaupt finden zu können, werden sog. *Keys*, also Schlüssel, verwendet.

Candidate Key

Prinzipiell ist es ein sehr triviales Problem, Daten irgendwo zu speichern – interessant wird es erst dann, wenn man versucht, an diese Daten wieder heranzukommen, insbesondere dann, wenn man nicht an allen *Tuples* (Datensätzen) interessiert ist, sondern an einem ganz bestimmten.

Die Tabellen einer relationalen Datenbank sind nicht sequentiell organisiert (intern manchmal schon, aber nicht aus Sicht des Benutzers), es gibt also nicht den ersten, den letzten, den 385. Datensatz. Um einen Datensatz eindeutig zu bestimmen (und ihn damit suchen, besser gesagt, finden zu können), ist ein *Candidate Key*, ein »eindeutiger Schlüssel«, nötig.

Unter einem *Candidate Key* versteht man eine *Attribut*-Menge (also eine Menge von einer oder mehreren Spalten), die *unique* (eindeutig) ist. So ein *Candidate Key* ist im einfachsten Fall eine fortlaufende Nummer, beispielsweise eine Kundennummer oder eine Bestellnummer. Zwingend notwendig ist das (zumindest theoretisch) nicht. Eine Person könnte man auch durch die Kombination mehrerer Personendaten eindeutig identifizieren – es würden allerdings viele Personendaten sein.

Die Kombination von Vor- und Nachnamen reicht da bei weitem nicht: Wir hatten schon in der Schule (bei einer Klassengröße von vielleicht 30 Schülern) zwei *Peter Glöckler*, bei größeren Datenmengen treten diese Probleme dann verstärkt auf. Die Adresse eignet sich kaum, weil die sich schnell ändern kann. Besser ist

schon eine Kombination von Vor- und Nachnamen, Geburtsdatum und Geburtsort – die Übereinstimmung zweier Personen in allen vier Spalten ist recht unwahrscheinlich. Andererseits: Wenn eine Übereinstimmung auftritt, dann wird sich die Datenbank weigern, den zweiten Datensatz anzunehmen, da können Sie sich »auf den Kopf stellen und mit den Füßen wackeln«.

Außerdem gibt es bei zusammengesetzten Schlüsseln ein zweites Problem: Solche Schlüssel werden zur Herstellung einer Referenz verwendet. Im Normalfall wird also in die Rechnungsdatei die Kundennummer aufgenommen, um die Kundenadresse mit den Rechnungsdaten zu verbinden. Die Alternative wäre, Vor- und Zunamen, Geburtsdatum und Geburtsort in die Rechnungsdatei aufzunehmen, der Speicherplatzbedarf für diesen Schlüssel wäre rund zehnmal so groß.

Außerdem dürfen solche *Candidate Keys* nicht geändert werden. Wenn die betreffende Person mit dem Familienstand auch ihren Nachnamen wechselt, dann kann dies in der Kundendatei nicht einfach geändert werden, weil dann die Rechnungen nicht mehr zugeordnet werden könnten – ein brauchbares RDBMS würde in einem solchen Fall von sich aus die Veränderung der Daten verhindern.

Candidate Keys sollten sich also aus *Attributes* (Spalten) zusammensetzen, deren Werte sich garantiert nie ändern. Also nicht *höchstwahrscheinlich nicht*, sondern *garantiert nicht.* Und hier gibt es eindeutig mehr unbrauchbare als brauchbare Eigenschaften. Geschlecht von Personen? Transsexuelle nach der Umwandlung. Namen von Nationalstaaten? Ein Blick in die jüngere Geschichte. Namen von Städten? Die Gemeindereform läßt grüßen. Einzig bei unären Relationen (Tabellen mit einer Spalte, im Prinzip Listen) mag man auf durchlaufende Nummern verzichten können, solange alle Werte wirklich nur einmal aufgenommen werden. (In einer Liste der gesetzlichen Feiertage wird der 3. Oktober nur einmal benötigt.)

An dieser Stelle möchte ich nochmals darauf hinweisen, daß Datenbanken – im Gegensatz zu Karteikarten – extrem unflexible Gebilde sind. Methoden wie »das notieren wir auf der Rückseite und machen vorne links unten ein dickes Kreuz, damit man auch nachsieht« lassen sich hier nicht anwenden.

Von daher gilt: Alle Tabellen haben eine durchlaufende oder von anderen Spalten unabhängige Nummer als *Candidate Key*, Ausnahmen bestätigen die Regel!

Primary Key

Jede Relation besitzt einen *Primary Key* (Primärschlüssel, Primärindex), um einen Datensatz eindeutig zu identifizieren, der *Primary Key* muß also ein *Candidate Key* sein. Es ist aber durchaus möglich, daß es in der Relation einen zweiten (dritten, vierten ...) *Candidate Key* gibt, der nicht Primärschlüssel ist.

Denkbar wäre beispielsweise, daß nach einer Firmen-Fusion in einer Tabelle die Kundennummern von Firma A den Kundennummern der Firma B zugeordnet

werden. Jeder Kunde läßt sich sowohl anhand der einen als auch der anderen Kundennummer zweifelsfrei identifizieren, aber *Primary Key* kann nur eine sein.

Es ist auch zu unterscheiden zwischen Schlüsseln, die momentan (noch) *Candidate Key* sind (Vor- und Nachnamen, Geburtsdatum und Geburtsort), und solchen, die garantiert immer *Candidate Key* sein werden (also den durchlaufenden Nummern beispielsweise). Als *Primary Key* eignen sich nur letztere.

Alternate Key

Zusätzlich zu den Primärschlüsseln lassen sich auch *Alternate Keys*, auch *Secondary Keys* (Sekundärschlüssel, Sekundärindizes) definieren. Dafür gibt es zwei Gründe: Zum einen kann dadurch die Performance steigen. Die Datensätze werden intern nicht sequentiell verwaltet, sondern baumartig. Nehmen wir an, das RDBMS bekommt folgende Anweisung:

```
SELECT * FROM testadr
  WHERE nummer = 12345
```

Es soll der Datensatz mit der Kundennummer (Primärschlüssel) *12345* ausgegeben werden. Würde hier eine sequentielle Suche gestartet, dann würde das RDBMS den ersten Datensatz suchen, vergleichen, ob die Nummer paßt, dann den nächsten, wieder vergleichen, und so weiter.

Man darf sich die Tabelle nicht als ein großes Array vorstellen, bei dem die 12345. Zeile auch dem Datensatz mit der Kundennummer 12345 entspricht. Es ist ohne weiteres möglich, daß die Hälfte der Datensätze schon wieder gelöscht ist. Außerdem muß die Kundennummer ja nicht eine fortlaufende Nummer sein; das kann von Karteikarten übertragen und daher völlig durcheinandergeraten sein. Es bleibt zunächst nichts anderes als die sequentielle Suche.

Nun wird aber anhand des Primärschlüssels eine Art Suchbaum aufgebaut (das ist eine extra Tabelle, bei Desktop-Datenbanken eine separate Datei). Hier müssen dann nicht 12345 Datensätze gelesen und verglichen werden, sondern die Suche geht in etwa folgendermaßen vor sich (bei der höchsten Kundennummer von 100 000):

- Ist die Kundennummer größer oder kleiner als 50 000? Sie ist kleiner.
- Ist die Kundennummer größer oder kleiner als 25 000? Sie ist kleiner.
- Ist die Kundennummer größer oder kleiner als 12 500? Sie ist kleiner.
- Ist die Kundennummer größer oder kleiner als 6250? Sie ist größer.
- Ist die Kundennummer größer oder kleiner als 9375? Sie ist größer.

Und so weiter und so fort. Nach noch ein paar solchen Vergleichen (die ich Ihnen jetzt ersparen möchte) hat man die physikalische Adresse des Datensatzen 12345, sucht anhand der Adresse diesen auf der Platte und gibt ihn aus. Dies alles erle-

digt das RDBMS selbstständig, für den Anwender spielt sich der Vorgang also »unsichtbar« ab. Ob eine Suche sequentiell oder nach einem Schlüssel erfolgt, läßt sich nur daraus erahnen, wie schnell das Ergebnis bereit steht.

Um nach anderen Attribut-Mengen (also Spalten oder Zusammenfassungen von Spalten) beschleunigt zu suchen, lassen sich auch für diese Attribut-Mengen solche Suchbäume erstellen. Diese werden *Alternate Key* (Sekundärschlüssel) genannt, belegen zusätzlichen Platz auf der Festplatte und beschleunigen (hoffentlich) das Zustandekommen des Abfrageergebnisses. (Ich habe versuchsweise die Bearbeitungsdauer von SQL-Abfragen bei Paradox-Dateien mit und ohne Sekundärschlüssel auf dem betreffenden Feld untersucht. Unterschiede waren dabei nicht feststellbar, weder bei der 16- noch bei der 32-Bit-BDE. Es ist aber nicht auszuschließen, daß das Beispiel ungünstig gewählt war.)

Es gibt aber noch einen zweiten Grund für die Verwendung von Sekundärindizes: Während SQL-Abfragen rein mengenorientiert sind und von daher auch alle Spalten als Suchkriterium kennen, kann die C++Builder-Komponente *TTable* nur nach Datensätzen im jeweilig gültigen Schlüssel suchen; dies ist in der Regel der Primärschlüssel, kann aber auch ein Sekundärschlüssel sein. Auch bei der Einschränkung der Datenmenge auf entsprechende Bereiche werden Schlüssel benötigt. (Ich sehe das allerdings weniger als ein Argument für Sekundärindizes, sondern eher dafür, *TQuery*, und damit SQL-Abfragen, statt *TTable* zu verwenden.)

Foreign Key

Gerade bei relationalen Datenbanken werden sehr viele Master-Detail-Verknüpfungen erstellt. Ein Beispiel dafür wäre die Rechnungstabelle und die Kundentabelle einer Firma (Bild 1.2): In der Rechnungstabelle finden sich Rechnungsnummer, Datum und Betrag, außerdem interessiert die Adresse des Kunden. Es wird jedoch nicht die komplette Adresse des Kunden in der Rechnungstabelle gespeichert, sondern nur die Kundennummer – mit dieser kann dann in der Kundentabelle die Adresse ermittelt werden.

Die Kundennummer, also die Spalte *Kunde*, wäre hier ein *Foreign Key*, also ein Fremdschlüssel. Dieser verweist auf den Primärschlüssel einer anderen Tabelle und stellt über ihn die Verbindung zu dieser her. Der SQL-Befehl für die Tabelle *Rechnungen* würde folgendermaßen lauten:

```
CREATE TABLE rechnungen
     (nummer INTEGER NOT NULL,
     datum DATE NOT NULL,
     kunde INTEGER,
     betrag MONEY NOT NULL,
     PRIMARY KEY (nummer),
     FOREIGN KEY (kunde) REFERENCES kunden (nummer))
```

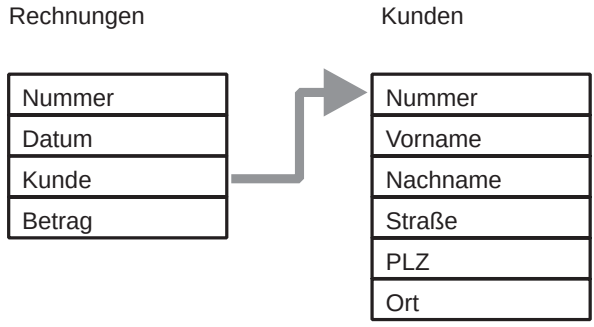


Bild 1.2: Einsatz von Fremdschlüsseln bei Master-Detail-Verknüpfungen

Beachten Sie bitte, daß der *Foreign Key* und der dazugehörende *Primary Key* der gleichen *Domain* unterliegen müssen.

Referentielle Integrität

Eine wesentliche Aufgabe der RDBMS ist die Sicherstellung der *referentiellen Integrität*. Darunter versteht man, daß für jeden Wert einer Spalte, die als *Foreign Key* definiert ist, ein entsprechender Wert in der damit verbundenen Tabelle vorhanden sein muß. Es darf also nicht vorkommen, daß in der Tabelle *Rechnungen* ein Kunde *2345* angegeben ist, der in der Tabelle *Kunden* nicht existiert. Dazu sind zwei Sicherheitsmechanismen erforderlich:

- Wird in die Tabelle *Rechnungen* ein neuer Datensatz eingefügt (oder ein bestehender geändert), dann muß sichergestellt werden, ob für den Wert in *Kunde* ein Datensatz in *Kunden* vorhanden ist.
- Wird in der Tabelle *Kunden* ein Datensatz gelöscht (oder entsprechend geändert), dann muß das RDBMS prüfen, ob es in der Tabelle *Rechnungen*, Spalte *Kunde* Einträge gibt, welche auf diesen Datensatz verweisen. Ist dies der Fall, dann ist eine der drei folgenden Maßnahmen zu treffen:
  - Die Aktion (löschen oder ändern) ist zu verweigern. Dies ist das naheliegendste, kann aber zu einem Konflikt mit dem Datenschutzgesetz führen. Wie man sich daran halbwegs vorbeimogelt, werden wir uns gleich ansehen.
  - Alle Werte der Spalte *Kunde* (in der Tabelle *Rechnungen*) werden auf NULL gesetzt. Dies geht allerdings nur, wenn bei der Erstellung der Tabelle nicht die Vorgabe NOT NULL getätigt worden ist (in diesem Fall: neue Adresse erstellen »Andi Anonymus, Aus-Datenschutzgründen-gelöscht-Straße 17, 00 000 Unbekannthausen« und alle Rechnungen auf diesen »Kunden« abändern).

In jedem Fall bleibt das Problem, daß der Kunde nicht mehr zu ermitteln ist. Das kann unter Umständen böse Folgen haben: Nehmen wir an, ein Kunde tätigt eine Bestellung und »bezahlt« per Einzugsermächtigung; kurz darauf weist er die Firma an, seine Adresse aus der EDV zu löschen, und diese kommt dem Wunsch nach. Nun geht dieser Kunde zu seiner Bank und widerspricht dem Einzug, die Firma erhält eine Rücklastschrift und möchte eine Mahnung schreiben, aber es fehlt die Adresse ... (mögliche Rettung: Backup-Bänder).

– Als dritte Möglichkeit bleibt, gleich auch noch alle damit verbundenen Rechnungen zu löschen; die referentielle Integrität wäre damit sichergestellt, aber im gerade erwähnten Beispiel könnte man dann noch nicht einmal nachvollziehen, welche Rechnung da nicht bezahlt wurde. (Vor Gericht könnte es dann so aussehen, als ob man ungerechtfertigt willkürliche Beträge einzöge.)

Views

Etwas aus dem Zusammenhang gerissen, aber als Antwort auf die gerade aufgeworfene Problemstellung: Bei allen (vernünftigen) Client-Server-Datenbanken lassen sich *Views*, also Ansichten definieren. Dabei handelt es sich um eine eingeschränkte oder zusammenfassende Sicht auf Tabellen.

Meistens wird man in Tabellen bestimmte Spalten sperren, welche Daten erhalten, die nicht für alle Anwender sichtbar sein sollen. Nehmen wir einmal an, in der Personaltabelle eines Unternehmens befänden sich die interne Durchwahl und das monatliche Gehalt (es findet sich noch viel mehr, aber das lassen wir aus Gründen der Übersichtlichkeit weg):

```
CREATE TABLE mitarbeiter
  (nummer AUTOINC,
  namen VARCHAR(20),
  durchwahl SMALLINT,
  gehalt MONEY,
  PRIMARY KEY (nummer));
```

Nun soll sich jeder Mitarbeiter zwar die Durchwahl seiner Kollegen beschaffen dürfen, aber das Gehalt geht ihn nun einmal nichts an. Deswegen erstellt man eine *View* auf diese Tabelle:

```
CREATE VIEW durchwahlen AS
  SELECT nummer, namen, durchwahl FROM mitarbeiter.
```

Mit Hilfe von Zugriffsberechtigungen erhält jeder den (Lese-)Zugriff auf die View *durchwahlen*, aber nur die Buchhaltung und die Personalabteilung erhalten den Zugriff auf die Tabelle *Mitarbeiter*.

Nun zurück zum Problem des Datenschutzes: Man kann das Problem halbwegs umgehen, wenn man neben der Kundentabelle auch noch eine *View* auf dieselbe erstellt:

```
CREATE TABLE tab_kunden
  (nummer AUTOINC,
  vornamen VARCHAR(20),
  ...
  anzeigen SMALLINT,
  PRIMARY KEY (nummer);

CREATE VIEW kunden AS
  SELECT nummer, vornamen,... FROM tab_kunden
  WHERE anzeigen = 1;
```

Der eigentlichen Datenbankapplikation gewährt man nur Zugriff auf die View *Kunden*. Wird dort ein Datensatz gelöscht, dann wird das Feld *anzeigen* auf NULL gesetzt, und somit scheint der Datensatz gelöscht – in Wahrheit befindet er sich immer noch in der Tabelle.

Bei der Referenz gibt es nun zwei Möglichkeiten:

- Es wird eine Referenz auf die Tabelle *Kunden* erstellt, die Adresse ist so weiterhin verfügbar; auf diese Weise ist aber auch schnell ersichtlich, daß man es mit den Datenschutzbestimmungen nicht so genau nimmt.
- Es wird eine Referenz auf die View *kunden* erstellt und das Feld *Kunde* beim Löschen dann auf NULL gesetzt; die Kundennummer muß in einem anderen Feld gesichert werden. Im Falle eines Falles meldet sich die Buchhaltung dann beim System-Administrator, der schaut sich die Tabelle *tab_kunden* an und erinnert sich an die Adresse.

1.2.2 Normalisierung

Unter *Normalisierung* versteht man die schrittweise Optimierung der Datenbank durch Veränderung der Tabellendefinition. Dazu werden mehrere *Normalformen* mit jeweils zusätzlichen Bedingungen definiert.

Eine Datenbank befindet sich beispielsweise in der dritten Normalform, wenn sie den Bedingungen der ersten, zweiten und dritten, nicht aber der vierten Normalform genügt. Würde sie beispielsweise den Bedingungen der ersten, dritten, vierten und fünften, nicht aber der zweiten Normalform genügen, dann befände sie sich in der ersten Normalform.

- Bedingungen der ersten Normalform:
  - Keine doppelten Datensätze.
  - Die Tupelreihenfolge und Attributreihenfolge muß beliebig sein dürfen. Die Funktionialität der Datenbank darf also nicht davon abhängen, daß auf eine Zeile oder eine Spalte eine bestimmte nächste Zeile bzw. Spalte folgt. Die Zeilen werden ausschließlich über den Primärschlüssel, die Spalten ausschließlich über den Spaltennamen angesprochen.
  - Alle Attributwerte unterliegen einer Domäne und sind somit atomar.
- (Zusätzliche) Bedingung der zweiten Normalform ist, daß jedes Schlüssel-Attribut funktional abhängig ist vom Gesamtschlüssel, nicht aber von Teilen desselben.
- Bedingung der dritten Normalform ist, daß es keine funktionalen Abhängigkeiten zwischen Attributen gibt, die nicht als Schlüssel definiert sind.

Es gibt noch weitere Normalformen, die wir allerdings hier ignorieren wollen (ab der dritten Normalform ist eine Datenbank schon sehr brauchbar). Wer sich näher für die Normalisierung interesstiert, dem sei das Buch *Relationale Datenbanken* von *Hermann Sauer* empfohlen (Addison-Wesley, ISBN 3-89319-821-0).

Wenn Sie bislang nur Bahnhof verstanden haben, dann ist dies nicht weiter tragisch, denn wir werden die Problematik zum besseren Verständnis an einem Beispiel erläutern. Betrachten Sie zunächst Bild 1.3 auf der nächsten Seite. Der Datensatzaufbau könnte der Mitarbeiter-Datei einer Firma entnommen worden sein: Diese Tabelle entspricht noch nicht einmal der ersten Normalform, da hier zwei Spalten Werte haben, die nicht atomar sind (*P-Nummer, P-Namen*). Auch wenn man mit einigen Tricks solche Daten in eine Tabelle zwängt (bei der Nummer eine *VarChar-Domain*), handelt man sich dabei nichts als Ärger ein; beispielsweise stehen Sie dann vor einem Problem, wenn Sie über solche Spalten eine Referenz bilden möchten.

Es gibt noch zwei weitere Möglichkeiten, die erste Normalform nicht zu erfüllen: Zum einen wären das Konstruktionen, die eine bestimmte Anordnung der Datensätze (oder Spalten) erfordern. So könnte man zum Beispiel auf die Idee kommen, daß der Datensatz, welcher als erster in der Tabelle steht, die Abteilung darstellt, welche für den Mitarbeiter »disziplinarisch« zuständig ist (also dort, wo er zuerst erfaßt wurde). Es gibt allerdings bei relationalen Datenbanken keine Reihenfolge der Datensätze (und der Attribute), so daß solche Konstruktionen gewagt sind (auch wenn man es meistens hinbekommt, daß die Sache in der Regel funktioniert).

Zum anderen gibt es das Problem der doppelten Datensätze. Sie kennen das vielleicht auch, daß man von einer Firma stets zwei Anschreiben, zwei Kataloge, zwei Weihnachtsgrüße bekommt. Liegen dabei unterschiedliche Kundennum-

Nummer	Name	Durchwahl	A-Nummer	A-Name	A-Chef	P-Nummer	P-Name
123	Müller	56	7	Konstruktion	Maier	8, 9	Zündmaschinen, Multiplexer

Bild 1.3: Relation genügt nicht der ersten Normalform

mern vor, dann sind es strenggenommen verschiedene Datensätze (und somit liegt kein Verstoß gegen die erste Normalform vor).

Erst recht problematisch wird es, wenn keine Kundennummer definiert ist und die Tabelle dadurch keinen *Candidate Key* besitzt. Nehmen wir einmal an, *Michael Mustermann* bekäme alles doppelt und würde der Firma schreiben, daß es zukünftig auch einmal reicht. Dort überprüft man das zunächst. Die entsprechende SQL-Anweisung lautet (auf Straße, PLZ und Wohnort verzichten wir hier der Kürze wegen):

```
SELECT * FROM kundenadressen
  WHERE (vornamen = "Michael") AND (nachnamen = "Mustermann")
```

Tatsächlich werden zwei Datensätze ausgegeben. Die Möglichkeit, einen von beiden zu löschen, gibt es allerdings nicht. Denn mit der SQL-Anweisung

```
DELETE FROM kundenadressen
  WHERE (vornamen = "Michael") AND (nachnamen = "Mustermann")
```

löscht man nun einmal beide Datensätze. Wenn der Anwender dieses Problem nicht kennt, dann gibt es für Herrn Mustermann zukünftig keinen Katalog mehr. Wenn er es kennt, dann wird er auf einem Blatt Papier die Mustermannschen Daten notieren, beide Datensätze löschen und den Datensatz erneut eingeben. (Gemäß Murphys Gesetz hat das Programm ein Feature, das Neukunden automatisch den Katalog zukommen läßt. Auf seine Bitte, künftig nur noch einen Katalog zu bekommen, wird ihm somit kommentarlos ein dritter zugeschickt). Wieder bestätigt sich die Regel, daß alle Tabellen als Primärindex eine (durchlaufende) Nummer bekommen sollten.

Die erste Normalform

Die folgende Tabelle genügt der ersten Normalform:

Nummer	Name	Durchwahl	A-Nummer	A-Name	A-Chef	P-Nummer	P-Name
123	Müller	56	7	Konstruktion	Maier	8	Zündmaschinen
123	Müller	56	7	Konstruktion	Maier	9	Multiplexer

Bild 1.4: Relation genügt der ersten Normalform

Gegen die zweite Normalform kann nur verstoßen werden, wenn der Primärschlüssel aus mehreren Spalten zusammengesetzt ist, was hier nicht der Fall ist (zumindest nicht, solange (sinnvollerweise) nur *Nummer* Primärschlüssel ist).

Sinn der Sache ist es, keine unnötig zusammengesetzten Primärschlüssel zu erhalten. Wäre beispielsweise der Primärschlüssel aus *Nummer* und *Name* zusammengesetzt, dann wären nicht nur alle übrigen Spalten vom Gesamtschlüssel abhängig, sondern auch vom Feld *Nummer* (*Müller* könnte es mehrere geben).

Wenn Sie sich an die Regel halten, alle Datensätze mit einer (durchlaufenden) Nummer als Primärschlüssel zu versehen, dann werden Sie auch nie gegen die Regeln der zweiten Normalform verstoßen.

Nach den Regeln der dritten Normalform sind keine funktionalen Abhängigkeiten zwischen Spalten erlaubt, die nicht als Primärschlüssel definiert sind.

In Bild 1.4 finden sich solche Abhängigkeiten:

- Abteilungsnummer, Abteilungsname und Abteilungsleiter sind voneinander funktional abhängig. Abteilung *sieben* ist immer *Konstruktion* und hat immer den Chef *Maier* (umgekehrt ist das nicht zwingend; es könnte eine weitere Konstruktionsabteilung geben, welche dann auch eine andere Nummer hätte).
- Projektnummer und Projektnamen sind auch voneinander abhängig.

Was stören diese Abhängigkeiten: Normalerweise sind in so einer Tabelle mehrere, beispielsweise 80 Datensätze aus der Abteilung *sieben*, und jedesmal wird wiederholt, daß die Abteilung *Konstruktion* und der Chef *Maier* heißt. Das ist nicht nur ein großzügiger Umgang mit Speicherplatz, wenn der Abteilungsleiter wechselt, dann sind 80 Datensätze anstatt eines Datensatzes zu ändern.

Schwieriger wird das Problem noch, wenn mit dem Chef auch die Sekretärin wechselt und der Vorgang von zwei verschiedenen Personen bearbeitet wird. Zunächst werden alle 80 Datensätze auf *Lehmann* umgestellt, danach wird die Sekretärin in die Abteilung versetzt. Der zuständige Anwender muß erstens (unnötigerweise) Abteilungsnamen und Abteilungsleiter von Hand eingeben, zweitens ist ihm noch nicht gesagt worden, daß der neue Chef *Lehmann* heißt, und so gibt er *Maier* ein; schon ist wieder eine Falschinformation mehr im System.

Die dritte Normalform

Um diese funktionalen Abhängigkeiten zu reduzieren, wird die ursprüngliche Tabelle einfach in mehrere kleinere Tabellen aufgeteilt, wie dies Bild 1.5 zeigt.

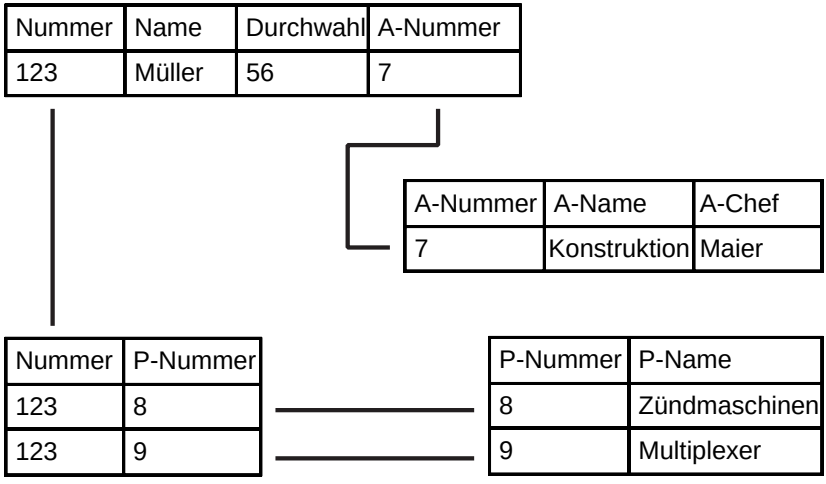


Bild 1.5: Aufteilung in mehrere Tabellen

1.2.3 Das erweiterte Entity-Relationship-Modell

Mit den Normalisierungsregeln läßt sich ein System vorhandener Tabellen in ein System optimierter Tabellen umwandeln. Bisweilen muß aber eine sehr komplexe und vielschichtige »Realität« in Tabellen gepreßt werden, ohne daß dabei Eigenschaften und Zusammenhänge verloren gehen.

Um das Erstellen der Tabellen zu erleichtern und zu systematisieren, wurde das Entity-Relationship-Modell geschaffen. Dadurch, daß hier nur drei Grade der Entity-Beziehung und die obligatorische Mitgliedschaft als zusätzliches Kriterium definiert werden, ist dieses Modell komplizierter als eigentlich erforderlich. Wir werden deshalb die Zahl der Entity-Grade etwas erweitern und so die Notwendigkeit beseitigen, die obligatorische Mitgliedschaft als zusätzliches Kriterium zu sehen.

Das Entity-Relationship-Modell

Das Entity-Relationship-Modell gliedert die »Realität« in *Entities* (Objekte, Hauptgruppen, also beispielsweise Kunden, Aufträge, Rechnungen...) und *Relationships* (Beziehungen zwischen den Objekten).

Das ER-Modell kennt dabei drei Grade von Beziehungen:

- 1:1 die Eins-zu-eins-Beziehung (Beispiel: Eine Haupstadt in einem Land)
- 1:N die Eins-zu-viele-Beziehung (Beispiel: Ein Land, in dem es viele Städte gibt)

- N:M die Viele-zu-viele-Beziehung (Beispiel: Flüsse, die durch mehrere Städte fließen und Städte, durch die mehrere Flüsse fließen)

Als zusätzliches Kriterium gibt es die *obligatorische Mitgliedschaft*. Darunter versteht man, daß der entsprechende Spaltenwert nicht NULL sein darf. Beim gerade erwähnten Beispiel der 1:1-Beziehung besteht bei beiden Attributen obligatorische Mitgliedschaft. Es gibt weder ein Land ohne Hauptstadt, noch gibt es eine Haupstadt ohne Land.

Es sind durchaus auch *Entities* vorstellbar, bei denen die Mitgliedschaft nur bei einem Attribut obligatorisch (NOT NULL) ist, genauso wie es *Entities* geben kann, bei denen die Mitgliedschaft bei keinem der Attribute obligatorisch ist. (Beispiel: Mitarbeiter arbeiten an Projekten, manche Mitarbeiter – z.B. die Reinigungskräfte – arbeiten an keinem Projekt, an manchen Projekten arbeitet kein Mitarbeiter, weil sie derzeit ruhen.)

Das erweiterte ER-Modell

Ob eine obligatorische Mitgliedschaft besteht, wurde nun beim herkömmlichen ER-Modell in Blockdiagrammen dargestellt, siehe Bild 1.6. Es ist nun aber anschaulicher, diese Information in die Bezeichnung der Beziehung aufzunehmen, so daß aus einer *1:1*-Beziehung je nach obligatorischer Mitgliedschaft *0/1 : 0/1*-, *1 : 0/1*-, *0/1 : 1*- oder *1 : 1*-Beziehungen entstehen. Hier besteht nun nicht mehr die Möglichkeit, daß man die Darstellungsformen von obligatorischer und nicht obligatorischer Mitgliedschaft durcheinanderbringt, außerdem läßt sich diese Information auch außerhalb von Blockdiagrammen darstellen.

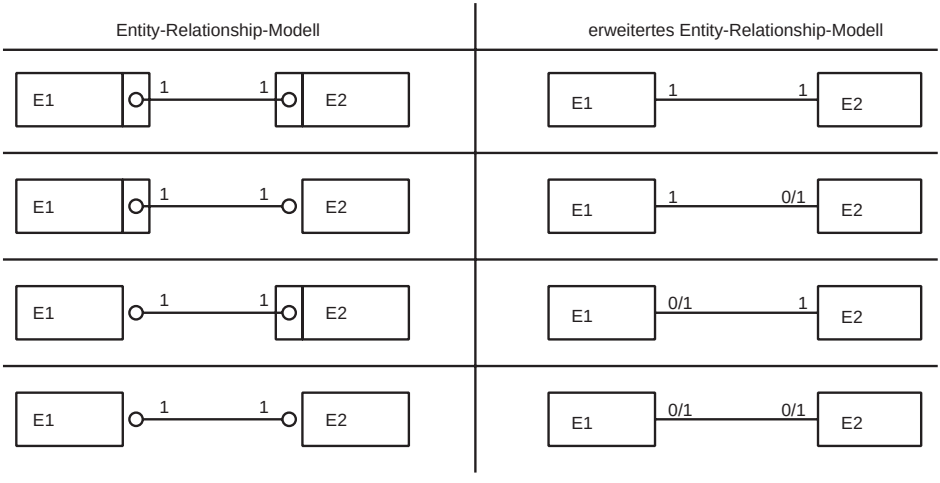


Bild 1.6: Konventionelles und erweitertes ER-Modell

Aus *1:N*- und *N:M*-Beziehungen werden dann beispielsweise *0/1 : 0/1/N*- oder auch *1/N : 0/1/M*-Beziehungen. Theoretisch denkbar wären hier beispielsweise auch 1 : N-Beziehungen, wenn zu einem Entity mehrere andere gehören (Gemarkung : Grundstücke). Um hier das EER-Modell vom ER-Modell zu unterscheiden, könnte man die jeweiligen Werte unterstreichen (1 : 1, 1 : N).

Transformation von Relationships in Attribute oder Relationen

Die Verknüpfungen (*Relationships*) der einzelnen Objekte (*Entities*) geschehen durch die Einfügung zusätzlicher Attribute (Spalten) oder durch Erstellung neuer Relationen (Tabellen).

- Bei der 1 : 1-Beziehung besteht grundsätzlich kein Grund, verschiedene Tabellen zu verwenden, Haupstädte und Länder sind einfach verschiedene Spalten derselben Tabelle. Ausnahmen sind eigentlich nur bei Desktop-Datenbanken sinnvoll:
  - wenn die Tabelle sonst zu groß wird, um noch auf eine Diskette zu passen
  - wenn zur Realisierung unterschiedlicher Zugriffsrechte nicht auf VIEWS zurückgegriffen werden kann

  Die 1 : 1-Verknüpfung wird dann dadurch hergestellt, daß die »zweite« Tabelle auf den Primärschlüssel der ersten referenziert wird.
- Bei der 0 / 1 : 1- (oder 1 : 0 / 1-) Beziehung hängt die Vorgehensweise davon ab, wie das Verhältnis der Zahl der *null*-Werte zur Zahl der *eins*-Werte ist:
  - Ist der *eins*-Wert die Regel (die meisten Lehrer sind Klassenlehrer irgendeiner Klasse), dann wird die Verknüpfung über eine weitere Spalte hergestellt, die ggf. NULL-Werte enthält.
  - Ist der *null*-Wert die Regel (die wenigsten Mitarbeiter fahren einen Dienstwagen), dann wird eine zusätzliche Tabelle erstellt.

  Bild 1.7 zeigt die beiden Möglichkeiten.
- Die beiden vorhin genannten Möglichkeiten gibt es auch bei der 0 / 1 : 0 / 1-Beziehung. Nun kann es allerdings auch vorkommen, daß bei beiden *Entities* die Zahl der *null*-Werte überwiegt. Als Beispiel seien die Professoren einer Uni und die Preisträger eines Wissenschaftspreises genannt. Nur sehr wenige Dozenten haben den Preis erhalten, und auch nur wenige Preisträger stammen von dieser Uni.

  Hier wird man dann eine zusätzliche Tabelle zur Herstellung der Verknüpfung erstellen, wie dies Bild 1.8 (nächste Seite) zeigt (die Namen sind hier frei erfunden). In diesem Fall ist es nicht unbedingt erforderlich, als Primärschlüssel für diese Tabelle eine durchlaufende Nummer zu verwenden – eine Kombination aus den beiden Spalten reicht hier vollständig aus.

Nummer	Name
1	Maier
2	Müller
3	Schulze
4	Lehmann
5	Schütz
6	Jansen
7	Goedecke
8	Anton

L-Nummer	L-Name	L-Klasse
1	Schneider	NULL
2	Poncelet	9
3	Will	8
4	Kowalski	10
5	Beck	5a

Die meisten Lehrer sind Klassenlehrer

Die wenigsten Mitarbeiter fahren einen Dienstwagen

Nummer	Fahrer	Fahrzeug	Kennzeichen
1	7	VW-Bus	B-UM 123
2	8	Opel Corsa	B-AB 456

Bild 1.7: Die Herstellung der 0/1:1-Beziehung

An diesem Beispiel wird auch noch einmal deutlich, welchen Vorteil die Verwendung von (durchlaufenden) Nummern als Primärindex hat: Prinzipiell hätte man für die Preisträger-Tabelle auch Jahreszahl und Fachbereich verwenden können (solange der Preis immer nur an eine Person vergeben wird), damit wäre der Datensatz eindeutig zu identifizieren gewesen. Bei der Verknüpfungstabelle hätte man dann aber statt einer einzelnen Nummer Jahreszahl und Fachbereich als Fremdschlüssel benötigt.

- Bleiben wir bei der Uni: Die Dozenten schreiben auch Fachbücher, die (neben Werken anderer Autoren) in der Bibliothek stehen. Hier haben wir dann die *0/1 : 0/1/N*-Beziehung: Die einzelnen Dozenten haben keins, eins oder mehrere Werke geschrieben, aber längst nicht alle Bücher der Bibliothek sind von einem Dozenten dieser Uni geschrieben worden. Hier wird dann die Verknüpfung über eine eigene Relation hergestellt, wie dies in Bild 1.9 gezeigt wird.

  Auf dieselbe Weise werden auch *0/1 : 1/N*- und *0/1 : N*-Beziehungen aufgebaut. Eine Ausnahme machen die *Relationships*, bei denen kaum *null*-Werte auf der *0/1*-Seite vorhanden sind; diese können auch nach dem Muster der 1 : 0 / 1 / N-Beziehung aufgebaut werden:

Nummer	Name	Fachbereich
1	Maier	Physik
2	Müller	Physik
3	Schulze	Physik
4	Lehmann	Physik
5	Schütz	Maschinenbau
6	Jansen	Maschinenbau
7	Goedecke	Chemie
8	Anton	Chemie

Nummer	Jahr	Name	Fachbereich
1	2011	Lehn	Physik
2	2011	Kolb	Mathematik
3	2011	Goedecke	Chemie
4	2011	Smith	Literatur
5	2012	Schulze	Physik

0 / 1

0 / 1

Dozent	Preis
7	3
3	5

Bild 1.8: Dritte Möglichkeit einer 0/1 : 0/1-Verknüpfung

Nummer	Name	Fachbereich
1	Maier	Physik
2	Müller	Physik
3	Schulze	Physik
4	Lehmann	Physik
5	Schütz	Maschinenbau
6	Jansen	Maschinenbau
7	Goedecke	Chemie
8	Anton	Chemie

Nummer	Autor	Titel
1	Lichtenstern	Witz und Aphorismus
2	Ebner	Programmieren in Delphi
3	Müller	Einführung in die Optik
4	Müller	Wellen- und Strahlenoptik
5	Schulze	Quantenmechanik

Dozent	Buch
2	3
2	4
3	5
5	27

Bild 1.9: Die 0/1 _ 0/1/N-Verknüpfung

- Als nächstes Beispiel sei die Auftrags-Kunden-Beziehung einer Firma genannt, wobei es sich strenggenommen um eine *1 : 1/N*-Beziehung handelt: Ein Auftrag kann nur von einem Kunden stammen (von wem sonst), und Kunde ist, wer mindestens einen Auftrag erteilt hat. Nun wird die Kundentabelle meist aber auch als Grundlage für den Katalogversand verwendet, so daß sich darin auch potentielle Kunden befinden, also Personen, die um einen Katalog gebeten, aber noch keinen Auftrag erteilt haben. Hier würde es sich dann um eine *1 : 0/1/N*-Beziehung handeln, was aber bei der Ausgestaltung keinen Unterschied macht: Da so oder so jeder Auftrag einen Kunden hat, wird in die Auftragstabelle eine Spalte eingefügt, die auf den Kunden verweist (siehe Bild 1.10).
- Recht übersichtlich wird es dann wieder bei der *(0)/(1)/N : (0)/(1)/M*-Beziehung. Als Beispiel sollen hier – wie bereits erwähnt – die Städte und Flüsse dienen: Durch manche Städte fließen mehrere Flüsse und die meisten Flüsse fließen durch mehrere Städte. Hier wird man wohl stets auf eine extra Beziehungs-Relation zurückgreifen (also auf eine Tabelle, deren eine Spalte auf die erste Tabelle und deren zweite Spalte auf die zweite Tabelle verweist). Auf ein Bild wurde hier verzichtet, inzwischen werden Sie das selbst hinbekommen.

  Eine Ausnahme könnte dann sinnvoll sein, wenn auf einer Seite die *null*- oder *eins*-Werte sehr deutlich überwiegen, der *n*-Wert dagegen höchst selten ist; ein passendes Beispiel fällt mir aber dazu nicht ein.

Nummer	Kunde	Datum	Betrag
1	3	1. 1. 96	100,00
2	3	2. 3. 96	200,00
3	3	4. 5. 96	123,45
4	4	2. 2. 96	234,56
5	4	1. 3. 98	345,67

Nummer	Name	Ort
1	Lichtenstern	Bad Buchau
2	Ebner	Berlin
3	Müller	Hamburg
4	Maier	Hamburg
5	Schulze	München

Bild 1.10: Die 1 : (0)/1/N-Beziehung

1.3 Borland Database Engine

Die *Borland Database Engine (BDE)* ist die zentrale Schnittstelle zwischen Datenbanken (also Tabellen, welche Daten enthalten) und den Datenbankanwendungen. Bild 1.11 beschreibt ein wenig vereinfacht die Zusammenhänge.

Betrachten Sie zunächst das Kästchen mit dem dicken Rahmen in der Mitte: Das ist die *Borland Database Engine*, kurz BDE. Sie nimmt im Prinzip eine Doppelfunktion wahr: Zum einen ist sie ein Datenbank-Management-System für *Paradox*, *dBase*- und *ASCII*-Datenbanken, zum anderen ist sie Schnittstelle für weitere Treiber, mit denen man auf ODBC- oder Client-Server-Datenbanken zugreifen kann. Zum Zugriff auf den *Local InterBase Server* ist noch nicht einmal ein zusätzlicher Treiber erforderlich. Eine Vielzahl von Programmen arbeiten mit der BDE zusammen. Wir wollen versuchen, hier für ein bißchen Klarheit zu sorgen.

Beginnen wir zunächst links oben beim Kästchen *Paradox, dBase, Delphi IDE, Delphi-Applikationen*. Auch diese Programme nutzen die BDE, was Auswirkungen auf die Weitergabe der entsprechenden Dateien hat: Prinzipiell ist die Weitergabe aller BDE-Dateien unentgeltlich, jedoch schreiben die Lizenz-Bedingungen vor, dem Anwender stets alle (!) dazugehörende Dateien inklusive des Setup-Programms zur Verfügung zu stellen. Damit soll sichergestellt werden, daß diese Borland-Produkte (und damit erstellte Anwendungen) auch nach der Installation einer C++Builder-Anwendung noch einwandfrei funktionieren (oder sich zumindest wieder zum Laufen bringen lassen).

Es gibt hier noch eine zusätzliche Einschränkung: Sie müssen die Installation mit Hilfe eines zertifizierten Installationsprogrammes durchführen lassen. Das ist jedoch weiter kein Hindernis, da das Programm *InstallShield Express* ohnehin zum Lieferumfang von C++Builder (Professional und Client-Server-Suite) gehört. Das Thema Installation werden wir in Kapitel 7 noch ausführlicher behandeln.

Außerdem greift natürlich eine mit C++Builder entwickelte (Datenbank-)Anwendung auf die BDE zu. Nützlich, manchmal aber auch problematisch ist es, daß die C++Builder-IDE genauso auf die BDE zugreifen kann. Dies hat den Vorteil, daß man schon bei der Entwicklung mit sogenannten Live-Daten arbeiten kann. Nehmen Sie einmal an, Sie würden eine Eingabemaske für eine Adressen-Tabelle erstellen: Schon beim Plazieren der einzelnen *TDBEdit*-Komponenten zeigen diese (normalerweise) den ersten Datensatz dieser Tabelle an. Das senkt die Wahrscheinlichkeit, daß Sie Straße und Postleitzahl vertauschen.

Hier kann es aber bisweilen zu Zugriffskonflikten kommen: Manche Methoden von *TTable* setzen voraus, daß die Datenbank *exclusive* geöffnet ist. Diese Eigenschaft läßt sich selbstverständlich nur setzen, wenn in diesem Moment kein anderes Programm auf diese Tabelle zugreift – was die C++Builder-IDE aber tut. Hier müssen Sie im Objekt-Inspektor die Eigenschaft *Active* auf *false* setzen und

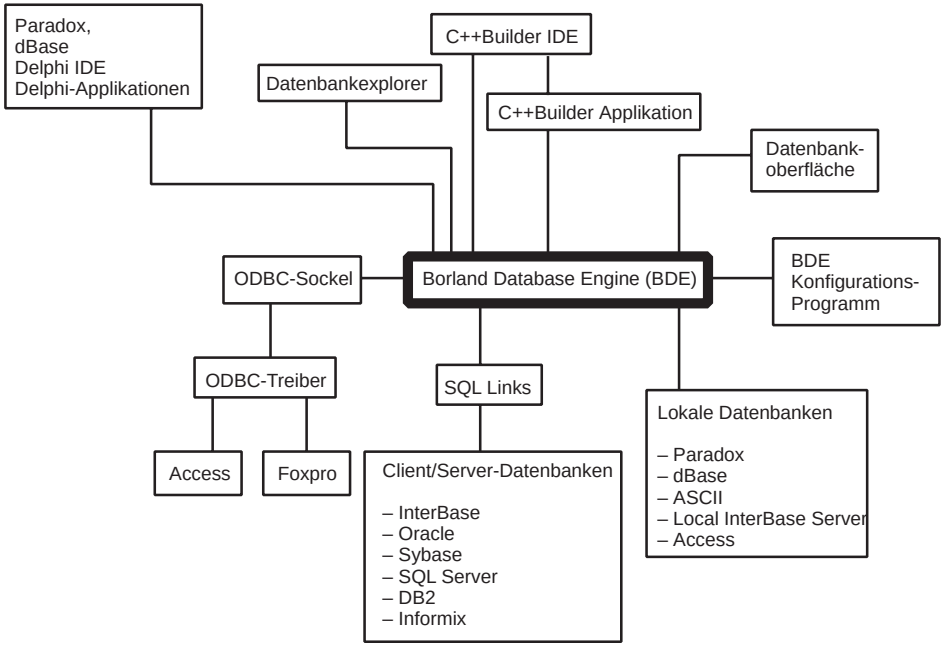


Bild 1.11: Die Funktion der BDE

die *TTable*-Komponente erst zur Laufzeit öffnen. (Vorsicht: Wenn Sie im Objektinspektor die *TTable*-Eigenschaften *Active* und *Exclusive* auf *true* setzen, können Sie die Anwendung nicht einmal mehr korrekt starten.)

Ein weiteres Programm, das auf die BDE zugreift, ist die *Datenbankoberfläche*. Dabei handelt es sich um eine universelle Datenbankanwendung, mit der sie Tabellen erstellen und löschen können, Datensätze darstellen und editieren, Abfragen erstellen können, usw. Für gewöhnlich werden Sie Tabellen nicht zur Laufzeit von der C++Builder-Applikation erstellen lassen, sondern diese mit der Datenbankoberfläche erzeugen und die Dateien dann beilegen. Auch wenn beim Entwikkeln der Anwendung Fehler auftreten und Sie dann wissen wollen, welche Daten nun tatsächlich in der Tabelle stehen, ist die Datenbankoberfläche sehr hilfreich. Näheres dazu in Kapitel 1.4.

Mit dem BDE-Konfigurationsprogramm kann man Aliase erstellen oder die Form der Datumsanzeige vorgeben. Wir werden uns dieses Programm gleich ansehen. Eine Art Mischung zwischen dem BDE-Konfigurationsprogramm und der Datenbankoberfläche ist der *Datenbankexplorer*, den wir in Kapitel 1.5 behandeln werden.

Nun zu den Datenbanken, auf welche die BDE zugreift: Zunächst hat die BDE von Haus aus die Fähigkeit, auf *Paradox*-, *dBase*-, *ASCII*- und neuerdings auch *Access*-Tabellen zuzugreifen. Bei diesen Datenbanken nimmt die BDE die Funktion des DBMSs wahr.

Des weiteren besteht die Möglichkeit, direkt auf den *Local InterBase Server* zuzugreifen; dabei handelt es sich um eine Einplatzversion des Datenbankservers *InterBase*. Mit Hilfe des LIBSs können Desktop-Datenbanken erstellt werden, die sich später sehr einfach auf Client-Server-Datenbanken aufrüsten (upsizen) lassen. Auch stehen hier viele Funktionen eines RDBMSs zur Verfügung (Views, Zugriffsberechtigungen...).

Um auf »echte« Client-Server-Datenbanken zuzugreifen, wird das Programm *SQL Links* benötigt, das der Client-Server-Suite von C++Builder beiliegt. Der Zugriff erfolgt – in Bild 1.11 wurde dies unterschlagen – über das Protokoll des entsprechenden Netzwerkes.

Für den Zugriff auf weitere Desktop-Datenbanken lassen sich ODBC-Treiber installieren. Damit kann dann auf Datenbanken wie beispielsweise *Access* oder *Foxpro* zugegriffen werden. Dies ist insbesondere dann notwendig, wenn die Applikation auf einen schon existierenden Datenbestand dieser Datenbanken zugreifen soll.

1.3.1 Das BDE-Konfigurationsprogramm

Mit dem BDE-Konfigurationsprogramm können Aliase erstellt, Treiber eingebunden und die Darstellungen für Zeit, Datum und numerische Werte festgelegt werden. Das Konfigurationsprogramm ist als *Tabbed Notebook* aufgebaut, die einzelnen Seiten können über die Reiter am unteren Rand ausgewählt werden.

Es soll nun eine kurze Einführung in das BDE-Konfigurationsprogramm gegeben werden. Ich habe hier auf die Vollständigkeit verzichtet, um das Kapitel nicht mit weniger wichtigen Details zu überfrachten. Sie können nämlich fast immer davon ausgehen, daß die vorhandenen Einstellungen sinnvoll sind und nicht geändert werden müssen; im Zweifelsfall hilft ein Blick in die Online-Hilfe.

Die Seite Treiber

Auf der Seite *Treiber* können neue Treiber eingebunden und auch wieder gelöscht werden. Des weiteren können hier bei den Treibern einige Modifikationen vorgenommen werden, was in der Regel nicht erforderlich sein sollte.

Beachtenswert ist die Option *StrictIntegrity* des Paradox-Treibers: Für gewöhnlich ist diese *true*. Es kann aber sein, daß Sie (oder Ihr Kunde) die Tabellen mit einer frühen Paradox-Version öffnen möchten, welche die referentielle Integrität noch nicht unterstützt. Bei dieser Einstellung wäre dies nicht möglich. Deshalb

kann *StrictIntegrity* auch auf *false* gestellt werden – dann ist es jedoch nicht mehr gewährleistet, daß die Integritätsregeln eingehalten werden.

Die Seite Aliase

Die Seite *Aliase* wird vor allem dazu verwendet, um neue Aliase zu erstellen.

Ein Alias ist eine Art »Abkürzung« für ein Verzeichnis. Dessen Verwendung hat zwei Vorteile: Zum einen erspart man sich die Eingabe endlos langer Verzeichnisnamen (statt *c:\Programme\Borland\Datenbankoberfläche* nur *Arbeit*). Zum anderen müssen beim Anwender die Datenbankdateien nicht notwendigerweise im gleichen Verzeichnis stehen wie beim Programmierer.

Um einen Alias zu erstellen, wird zunächst der Button *Neuer Alias* betätigt, worauf sich ein Dialogfenster öffnet. Geben Sie hier einen Alias-Namen ein und wählen Sie einen Alias-Typ aus. Für *Paradox-*, *dBase-* und *ASCII*-Datenbanken wählen Sie *Standard*, für Datenbanken auf dem *Local InterBase Server* wählen Sie *IntrBase*. Haben Sie die Client-Server-Suite von C++Builder, dann können Sie auch Aliase für einen Datenbank-Server einrichten.

Nachdem der Alias definiert worden ist, muß noch das Verzeichnis (*Path*) angegeben werden; in diesem Verzeichnis werden die Datenbankdateien gespeichert.

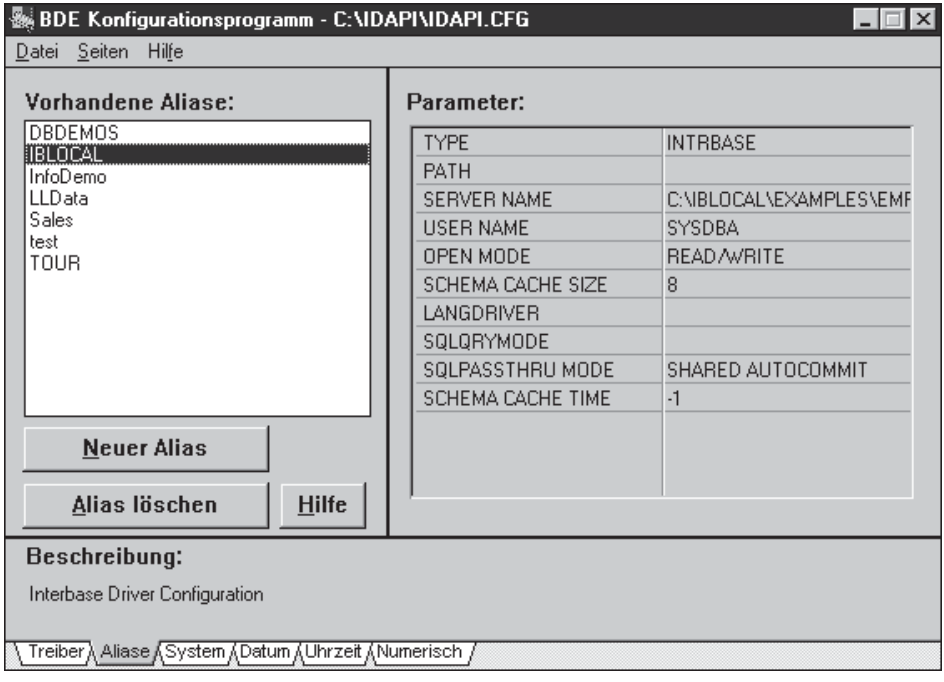


Bild 1.12: Das BDE-Konfigurationsprogramm

Die Seite System

Auf der Seite System können Werte für den gewünschten Sprachtreiber oder die zweckmäßige Blockgröße auf der Festplatte festgelegt werden. Interessant für den Fall, daß auch Nicht-BDE-Anwendungen auf eine Tabelle zugreifen, ist die Option *LocalShare* (Voreinstellung *false*). Soll gleichzeitig von der BDE und einer Nicht-BDE-Anwendung auf dieselbe Datei zugegriffen werden können, muß *LocalShare* auf *true* gestellt werden.

Die Seite Datum

Hier wird eingestellt, wie der Wert eines Datum-Feldes in einen String gewandelt wird und umgekehrt. Des weiteren kann gewählt werden, welches Trennzeichen (*Separator*) verwendet wird (*1.1.97* oder *1/1/97*), in welcher Reihenfolge Tag, Monat und Jahr erscheinen (*Mode*), ob die Jahreszahl zwei- oder vierstellig angezeigt wird und so weiter. Details entnehmen Sie bitte der Online-Hilfe

Die Seite Uhrzeit

Hier kann sowohl zwischen 12- und 24-Stunden-Anzeige gewählt werden als auch vorgegeben werden, ob Sekunden (oder sogar Millisekunden) angezeigt werden sollen.

Die Seite Numerisch

Auf dieser Seite werden Dezimal- und Tausendertrennzeichen eingestellt, die Anzahl der Nachkommastellen gewählt, und es kann entschieden werden, ob Vorkomma-Nullen angezeigt werden (*0.67, LeadingZeroOn := true*) oder nicht (*.67*).

1.4 Die Datenbankoberfläche

Die Datenbankoberfläche (DBO) ist eine Universal-Applikation zum Zugriff auf die BDE. Mit ihr lassen sich Tabellen anzeigen, editieren, erzeugen, verändern und löschen sowie noch viele andere Dinge tun. In der Regel wird man die Tabellen einer Datenbankapplikation zur Entwurfszeit mit der Datenbankoberfläche erstellen (oder auf vorhandene Dateien zugreifen) und lediglich die »Benutzeroberfläche« in C++Builder programmieren.

Wir wollen nun zunächst besprechen, wie man mit der Datenbankoberfläche Tabellen erstellt und mit Daten füllt.

1.4.1 Erstellen von Tabellen mit der DBO

Um eine Tabelle zu erstellen, wählen Sie DATEI | NEU | TABELLE und im dann erscheinenden Fenster, welchen Tabellentyp Sie verwenden wollen. Wenn Sie die Tabellen »rücksichtslos« erstellen können, also nicht darauf angewiesen sind, daß sie von bestimmten Datenbankprogrammen gelesen werden können, dann sollten sie *Paradox 7.0*-Tabellen erstellen.

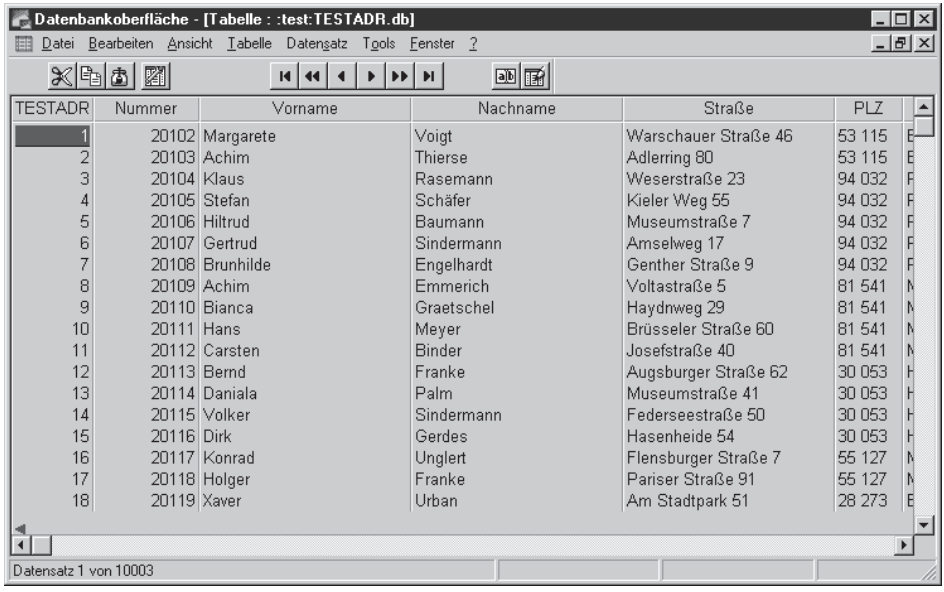
Nachdem der Tabellentyp gewählt und der OK-Button betätigt worden ist, wird folgendes Fenster geöffnet:

Hier geben Sie zunächst die Namen der einzelnen Spalten ein, welcher Typ dafür verwendet wird, gegebenenfalls die Länge derselben und ob sie zum Primärschlüssel gehört. Verwenden Sie in den Feldnamen auf keinen Fall das Punkt-Zeichen – hier wird dies akzeptiert, aber es wird keine SQL-Abfrage mehr brauchbar mit dieser Spalte funktionieren.

Die Feldtypen

Mit der rechten Maustaste oder der Leertaste rufen Sie dann die Liste der verfügbaren Feldtypen auf. Bei *Paradox 7.0*-Tabellen stehen folgende Feldtypen zur Verfügung:

- *A (Alpha):* String-Feld für bis zu 255 (druckbare) ASCII-Zeichen. Entspricht dem Pascal-Typ *String*. Die Länge muß angegeben werden.
- *N (Zahl, numerisch):* Zahlenfeld für Zahlen zwischen 10^{-307} und 10^{308} bei einer Genauigkeit der Gleitkomma-Mantisse von 15 Stellen. Achtung: Für Zahlen wie Postleitzahlen und Telefonnummern sollten Sie aus folgenden Gründen Alpha-Felder verwenden:
  - numerische Felder eliminieren alle anführenden Nullen (in Ostdeutschland gibt es auch Postleitzahlen mit führender Null)
  - in numerischen Feldern dürfen keine Leerstellen oder Trennzeichen verwendet werden
- *DM (oder $, Währung):* Diesem Zahlenfeld wird automatisch ein String mit der jeweiligen Währung hinzugefügt (in Deutschland DM). Für gewöhnlich werden zwei Nachkommastellen angezeigt, intern wird mit sechs Nachkommastellen gerechnet.
- *S (Short):* Hierbei handelt es sich um eine 16-Bit-Integer-Zahl (also eine Ganzzahl, ohne Nachkommastellen. Der Wertebereich liegt zwischen + / - 32 767.
- *I (Integer):* Eine 32-Bit-Integerzahl mit dem Wertebreich von + / - 2 147 483 647.
- *# (BCD):* Enthalten Zahlen im BCD-Format. Dieser Feldtyp ist vor allem der Kompatibilität mit anderen Systemen wegen vorhanden und sollte normalerweise nicht verwendet werden.



TESTADR	Nummer	Vorname	Nachname	Straße	PLZ
1	20102	Margarete	Voigt	Warschauer Straße 46	53 115
2	20103	Achim	Thierse	Adlerring 80	53 115
3	20104	Klaus	Rasemann	Weserstraße 23	94 032
4	20105	Stefan	Schäfer	Kieler Weg 55	94 032
5	20106	Hiltrud	Baumann	Museumstraße 7	94 032
6	20107	Gertrud	Sindermann	Amselweg 17	94 032
7	20108	Brunhilde	Engelhardt	Genther Straße 9	94 032
8	20109	Achim	Emmerich	Voltastraße 5	81 541
9	20110	Bianca	Graetschel	Haydnweg 29	81 541
10	20111	Hans	Meyer	Brüsseler Straße 60	81 541
11	20112	Carsten	Binder	Josefstraße 40	81 541
12	20113	Bernd	Franke	Augsburger Straße 62	30 053
13	20114	Daniala	Palm	Museumstraße 41	30 053
14	20115	Volker	Sindermann	Federseestraße 50	30 053
15	20116	Dirk	Gerdes	Hasenheide 54	30 053
16	20117	Konrad	Unglert	Flensburger Straße 7	55 127
17	20118	Holger	Franke	Pariser Straße 91	55 127
18	20119	Xaver	Urban	Am Stadtpark 51	28 273

Bild 1.14: Anzeige einer Tabelle mit der Datenbankoberfläche

- *D (Datum):* Datumsfelder in Paradox-Tabellen dürfen alle gültigen Daten vom 1. Januar 9999 v. Chr. bis 31. Dezember 9999 enthalten. Die Datenbank berücksichtigt die Schaltjahre und -jahrhunderte korrekt und prüft die Daten auf ihre Gültigkeit. Das Datumsformat wird mit dem BDE-Konfigurationsprogramm eingestellt.
- *T (Zeit):* Intern wird die Zeit gespeichert als Zahl der Millisekunden, welche seit 0.00 Uhr vergangen sind. Diese Zahl wird vor der Anzeige in das gebräuchliche Zeitformat gewandelt, wobei im BDE-Konfigurationsprogramm eingestellt werden kann, wie die Zeit genau angezeigt werden soll (z.B. mit oder ohne Sekunden oder Millisekunden).
- *@ (Zeitstempel, also Datum und Zeit):* Kombiniert in einem Feld Datum und Zeit.
- *M (Memo):* Wird für Strings verwendet, die länger als 255 Zeichen sind oder mit Zeilenumbrüchen formatiert werden sollen. Memos werden vollständig in einer separaten Datei gespeichert. Des weiteren wird eine anzugebende Zahl von Zeichen in der Tabelle selbst gespeichert; auf diese Zeichen kann schneller zugegriffen werden, was das Durchscrollen einer Tabelle beschleunigt. Eine Eingabe von Memos ist mit der Datenbankoberfläche nicht möglich; verwenden Sie dazu die C++Builder-Komponente *TDBMemo*.

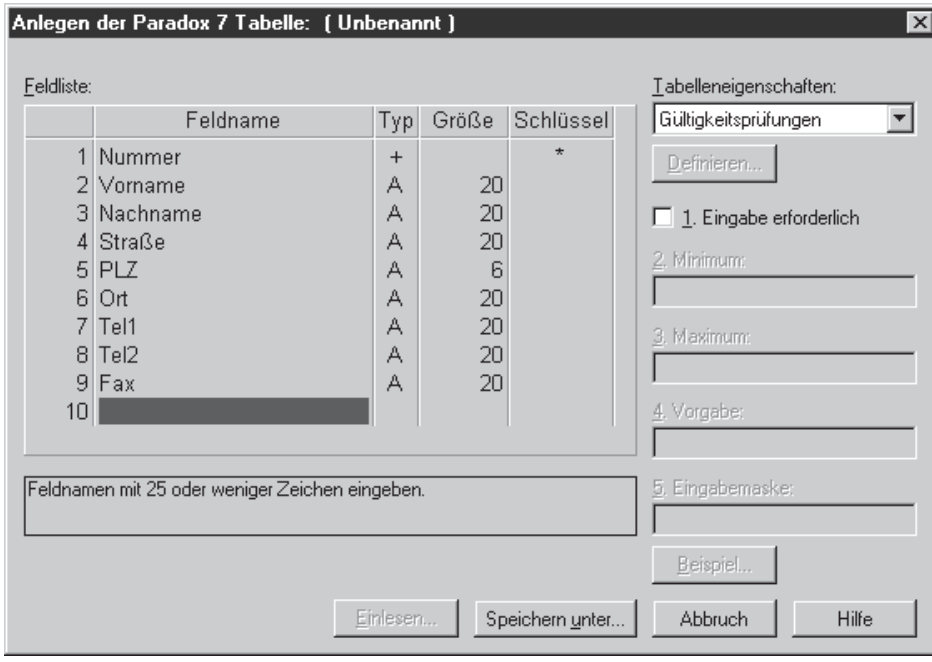


Bild 1.15: Erstellen einer Tabelle

- *F (formatiertes Memo):* Hier können auch Informationen zur Schrift u.ä. aufgenommen werden. Es gibt allerdings (noch) keine C++Builder-Komponente, welche darauf zugreifen kann.
- *G (Graphik):* In diese Felder können Bilder gespeichert werden. Eine Eingabe von Bildern ist mit der Datenbankoberfläche nicht möglich; verwenden Sie dazu die C++Builder-Komponente *TDBImage.*
- *O (OLE):* Dient zur Speicherung von OLE-Feldern.
- *L (Logik):* In logischen Feldern können nur die Werte *wahr* und *falsch* gespeichert werden. Dieser Spaltentyp entspricht dem Pascal-Typ *boolean.*
- *+ (selbstinkrementierend):* Ganzzahl, welche bei jedem neu eingegebenen Datensatz um eins erhöht wird. Sie ist somit ein unverwechselbares Kennzeichen dieses Datensatzes und eignet sich sehr gut als Primärschlüssel. Nummern von Datensätzen, welche gelöscht werden, werden dabei nicht neu vergeben.
- *B (Binär):* Dient der Speicherung binärer Daten (Klänge, Videos u.ä.). Es gibt keine C++Builder-Komponente, die darauf zugreift.
- *Y (Bytes):* Auch hier handelt es sich um binäre Daten, welche aber in der Tabelle gespeichert werden.

Primärschlüssel

Das Feld (oder – bei zusammengesetzten Schlüsseln – die Felder), das als Primärschlüssel dient, erhält einen * in der entsprechenden Spalte. Beachten Sie bitte, daß nur das erste Feld (oder die ersten Felder) als Primärschlüssel verwendet werden können, und verändern Sie gegebenenfalls die Reihenfolge der Felder.

Sie können sich sehr viel Ärger ersparen, wenn Sie als Primärschlüssel eine selbstinkrementierende Zahl verwenden. Zwar können auch andere Felder als Primärschlüssel definiert werden, aber diese sind oft nicht *unique* (einzigartig, damit eindeutig). Hier wäre es dann nötig, eine Kombination von mehreren Feldern als Primärschlüssel zu verwenden. Sobald eine Referenz auf diese Tabelle gebildet wird, müßten dann alle diese Spalten dort wieder verwendet werden, um eindeutig auf den Primärschlüssel zu verweisen – die Referenztabelle würde unnötig lang werden.

Bei Relationen, die aus einer Beziehung erstellt werden und somit lediglich zwei Spalten mit den entsprechenden Referenzen auf die jeweiligen Tabellen haben, macht ein selbstinkrementierender Primärschlüssel wenig Sinn, auch dann nicht, wenn er als zusätzliche Spalte eingefügt wird. Hier wird der Primärschlüssel als Kombination aus den beiden Spalten gebildet, die (wenn die anderen Tabellen als Primärindex eine selbstinkrementierende Integerzahl haben) jeweils vom Typ *Integer* sind.

Gültigkeitsprüfungen

Sie können die Datenbank jede Eingabe daraufhin überprüfen lassen, ob sie bestimmten Kriterien entspricht; falls dies nicht der Fall ist, wird eine Fehlermeldung ausgegeben.

- *Eingabe erforderlich:* Dies entspricht der SQL-Anweisung NOT NULL. In vielen Fällen macht ein freigelassenes Feld keinen Sinn (Adresse ohne Nachname). Bedenken Sie aber stets, daß ein Datensatz ohne diesen Wert nicht eingegeben werden kann. (Oder man mogelt sich darum herum, indem man beispielsweise *Unbekannt* eingibt.)
- In den Feldern *Minimum* und *Maximum* können Werte eingeben werden, die den Eingabebereich begrenzen. Der eingegebene Werte muß größer gleich dem *Minimum* und kleiner gleich dem *Maximum* sein.
- Wird häufig derselbe Werte eingegeben (z.B. der Mehrwertsteuersatz), dann kann mit *Vorgabe* ein Wert eingegeben werden, der automatisch in das betreffende Feld eingefügt wird. Wenn der Anwender damit nicht einverstanden ist, kann er ihn immer noch ändern.
- *Eingabemaske:* Bisweilen ist es sinnvoll, daß die eingegebenen Werte einer bestimmten Formatierung unterliegen. So könnte beispielsweise gefordert sein, daß Postleitzahlen in der Form *12 345* eingegeben werden. Dies kann durch eine entsprechende Eingabemaske sichergestellt werden.

Nachschlagetabelle

Die Nachschlagetabelle gewährleistet die »halbe referentielle Integrität«. Hier wird sichergestellt, daß in die Tabelle A nur Werte eingefügt werden können, die auch in der Tabelle B vorhanden sind. Die Nachschlagespalte von Tabelle B muß dabei Primärindex sein.

Hier gibt es dann zwei Möglichkeiten: Entweder erlaubt die Tabelle A nur Eingaben aus Tabelle B, stellt aber keine Hilfe zur Eingabe bereit (*Einlesen ohne Hilfe*, kann den Anwender bisweilen zur Verzweiflung bringen), oder sie stellt dem Anwender in einem Fenster eine Nachschlagetabelle zur Verfügung, aus welcher er den entsprechenden Wert auswählt (*Hilfe und einlesen*, die Hilfe wird von C++Builder leider nicht unterstützt).

Im Gegensatz zu Fremdschlüsseln (referentiellen Integrität) dürfen Werte aus Tabelle B gelöscht werden, auch wenn sie in Tabelle A eingefügt wurden.

Sekundärindizes

Viele Methoden der C++Builder-Komponente *TTable* arbeiten nur mit Indizes, für gewöhnlich wird also über den Primärindex gesucht, sortiert oder gefiltert. Nun ist es oft wünschenswert, daß diese Methoden auch mit anderen Spalten oder Spaltenkombinationen arbeiten – dazu können dann sogenannte Sekundärindizes erstellt werden.

Zum Erstellen eines Sekundärindexes werden die Spalten ausgewählt, welche den Index bilden sollen, diese gegebenenfalls in die richtige Reihenfolge gebracht, und dem Sekundärindex wird ein Name zugewiesen.

Referentielle Integrität

Was man unter dem Begriff *referentielle Integrität* versteht, wurde in Kapitel 1.2.2 erläutert. Um eine solche für Paradox-Tabellen zu erstellen, muß man sowohl das Feld, das auf eine andere Tabelle verweist, als auch die Tabelle, auf die verwiesen werden soll, auswählen. Das Referenz-Feld und der Primärschlüssel der zugeordneten Tabelle müssen der gleichen *Domain* unterliegen.

Es können pro Tabelle mehrere Referenzen definiert werden; um diese zu unterscheiden, muß die Referenz benannt, also mit einem Namen versehen werden.

Des weiteren haben Sie die Möglichkeit, zwischen den Aktualisierungsregeln *weitergeben* oder *verhindern* zu wählen. Nehmen wir an, Sie hätten in einer Kundentabelle den Datensatz mit der Kundennummer *12345* und wollten diese – aus was für Gründen auch immer – auf *54321* ändern. Damit die *Aktualisierungsregel* relevant wird, muß es einen Datensatz in einer anderen Tabelle geben (Aufträge, Rechnungen ...), welcher über eine Referenz mit diesem Kundendatensatz verbunden ist.

Ist die Aktualisierungsregel *verhindern* gewählt, dann läßt sich die Kundennummer nicht ändern. Ist *weitergeben* gewählt, dann werden alle Fremdschlüssel-Felder von *12345* auf *54321* geändert – die Integrität der Datenbank bleibt gewährleistet.

Paßwortschutz

In Datenbanken werden oft Daten abegelegt, die nicht für jedermanns Zugriff bestimmt sind, sei es, daß Datenschutzbestimmungen dies einschränken, sei es, daß Betriebsgeheimnisse o.ä. dort gespeichert sind. Ein wesentliches Kriterium eines DBMSs ist es daher, daß es die Tabellen mit Paßwörtern vor fremdem Zugriff schützt.

Nun wäre ein solcher Paßwortschutz wenig wert, wenn man die entsprechende Tabelle zwar nicht mehr mit der Datenbankoberfläche oder anderen BDE-Programmen öffnen könnte, aber ein simpler Text-Editor reichen würde, um die Daten lesen und auch manipulieren zu können.

Bild 1.16 zeigt, wie eine sehr einfache, kleine Datenbank im Texteditor aussieht. (Nebenbei bemerkt, ist die Auswirkung großzügig definierter Felder auf den Speicherplatzbedarf deutlich zu erkennen. Gerade bei Datenbanktabellen können deshalb Packer-Programme wahre Wunder vollbringen.)

Bild 1.17 zeigt, wie dieselbe Tabelle mit Paßwortschutz aussieht. Wie Sie sehen, sehen Sie nichts – zumindest nichts, was Sie ohne weiteres lesen könnten. Im übrigen vermittelt Bild 1.17 den falschen Eindruck, daß verschlüsselte Datentabellen deutlich mehr Speicherplatz belegen als unverschlüsselte: Bei großen Tabellen ist so gut wie kein Unterschied feststellbar.

Um ein Paßwort vorzugeben, wählen Sie die entsprechende Option, geben das entsprechende Paßwort ein, bestätigen es zur Sicherheit und schließen die Aktion mit OK ab. Sobald auf diese Weise ein Hauptpaßwort erstellt worden ist, können auch Hilfspaßwörter erstellt werden: Während Benutzer des Hauptpaßwortes ein uneingeschränktes Zugriffsrecht auf die Tabelle haben, kann dieses für die Benutzer der Hilfspaßwörter eingeschränkt werden:

Zum einen besteht die Möglichkeit, für jede Spalte zu definieren, ob der Inhaber des Hilfspaßwortes Schreib- und Lese-Recht, Nur-Lese-Recht oder gar kein Zugriffsrecht hat. Hat der Benutzer beispielsweise nur ein Lese-Recht in einer bestimmten Spalte, dann werden in einer C++Builder-Applikation die Versuche, den Feldinhalt zu ändern, mit einer Exception bestraft.

Des weiteren besteht die Möglichkeit, die Zugriffsrechte für die gesamte Tabelle einzuschränken. Hier stehen folgende Möglichkeiten zur Verfügung:

- *Alle:* Gewährt einem Benutzer alle Rechte für alle Tabellenfunktionen, einschließlich der Möglichkeit, eine Tabelle umzustrukturieren oder zu löschen. Die einzige verbotene Operation ist die Änderung des Master-Paßworts. Wird

Bild 1.16: Editoransicht einer Datenbank ohne Paßwortschutz

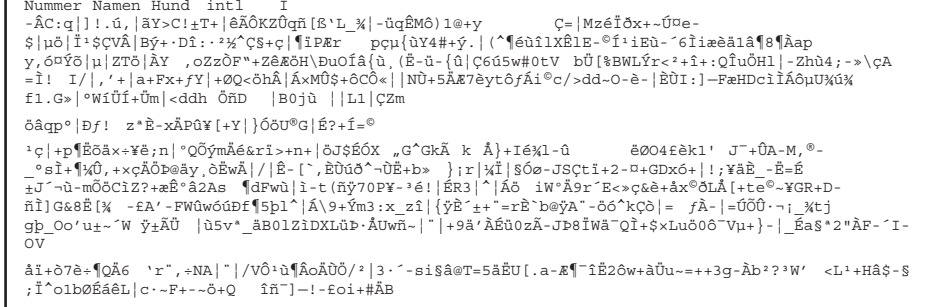


Bild 1.17: Editoransicht derselben Datenbank mit Paßwortschutz

das Zugriffsrecht um auch nur eine einzige Spalte eingeschränkt, dann wird das Tabellen-Zugriffsrecht auf *Aktualisieren* geändert.

- *Einfügen & Löschen:* Gewährt einem Benutzer das Recht, Datensätze einzufügen oder zu löschen, nicht aber die Tabelle zu löschen.
- *Dateneingabe:* Der Benutzer hat lediglich das Recht, Daten in die Tabelle einzugeben oder bestehende Datensätze zu ändern.
- *Aktualisieren:* Gewährt einem Benutzer das Recht, Tabellen zu betrachten und Nicht-Schlüsselfelder zu ändern, nicht aber Datensätze einzufügen oder zu löschen oder Schlüsselfelder zu verändern.
- *Nur Lesen:* Der Benutzer kann die Daten lesen, aber keine Datensätze einfügen, ändern oder löschen.

Sonstiges

Unter der Rubrik *Sprachtreiber* kann eingestellt werden, welche Sprache beispielsweise bei der Sortierung von Datensätzen verwendet werden soll. Mit der Option *abhängige Tabellen* kann man feststellen, ob andere Tabellen diese Tabelle als Fremdschlüssel verwenden (hier klärt sich dann beispielsweise, warum bestimmte Aktionen verweigert werden).

dBase-Tabellen

Mit der Datenbankoberfläche können auch Tabellen im dBase-Format angelegt und bearbeitet werden. Bisweilen kann dies sinnvoll (oder gar notwendig) sein, um beispielsweise die Kompatibilität mit anderen Programmen zu wahren.

Sie handeln sich jedoch damit eine ganze Reihe von Nachteilen ein:

- Es stehen weniger Feldtypen zur Verfügung, insbesondere ist das Fehlen von selbstinkrementierenden Feldern ärgerlich (Das Fehlen von Grafik-Feldtypen ist dagegen unerheblich, erfüllt doch auch der Feldtyp *binär* diesen Zweck).
- Es lassen sich keine Nachschlagetabellen und keine Referenzen definieren. Während man Nachschlagetabellen auch mit C++Builder basteln kann, ist das Fehlen von Referenzen sehr ärgerlich: Das Sicherstellen der referentiellen Integrität ist Aufgabe des DBMSs und nicht der Anwendung.
- Es lassen sich keine Paßwörter vergeben.

InterBase-Tabellen

Bei der Erstellung von InterBase-Tabellen haben Sie die Wahl zwischen der Datenbankoberfläche und *Windows ISQL*. Die Datenbankoberfläche bietet den Vorteil, daß die Bedienung etwas einfacher ist – wesentliche Merkmale wie beispielsweise Fremdschlüssel (referentielle Integrität) sind damit allerdings nicht machbar. Es bleibt also nichts weiter übrig, als *Windows ISQL* zu verwenden und die entsprechenden SQL-Befehle zu lernen, was zumindest nicht schaden kann.

1.4.2 Bearbeiten von Tabellen

Mit der Datenbankoberfläche können nicht nur Tabellen erstellt werden, es können damit auch Daten eingefügt werden. Dafür gibt es prinzipiell zwei Möglichkeiten: Die Eingabe von Hand oder die Übernahme aus anderen Tabellen.

Eingeben von Daten

Sie können mit der Datenbankoberfläche auch Daten in die Tabelle eingeben; wählen Sie dazu aus dem Hauptmenü ANSICHT | DATEN EDITIEREN oder kurz F9. Beachten Sie, daß sie keine BLOB-Daten eingeben können (BLOB = *binary large Object*, also Graphiken, Memos, usw.).

Stapel-Operationen

Mit der Datenbankoberfläche sind auch Stapel-Operationen möglich, also Aktionen, die in C++Builder sowohl mit der Komponente als auch mit der Methode *BatchMove* getätigt werden. Dabei werden Änderungen in einer Tabelle auf der Grundlage einer anderen Tabelle durchgeführt. Die entsprechenden Anweisungen finden Sie unter dem Menü-Titel *Tabellenoperationen.*

Auf diese Weise können Sie beispielsweise Kopien in einem anderen Format anlegen. Ein anderes Anwendungsbeispiel ist, daß beispielsweise ein Außendienstmitarbeiter in der Zentrale eine Diskette mit einer Tabelle neuer Kunden abgibt, die mittels einer Stapel-Operation »in einem Rutsch« in die Kundentabelle eingefügt werden.

1.5 Der Datenbankexplorer

Bislang, also vor allem in den Delphi-Versionen 1 und 2, waren drei Tools zum Verwalten von Datenbanken vorgesehen:

- Die *Datenbankoberfläche* zum Erstellen von Tabellen für Desktop-Datenbanken sowie zur Eingabe Daten und zum Generieren von Abfragen.
- Das *BDE-Konfigurationsprogramm* zum Einrichten von Datenbank-Aliasen.

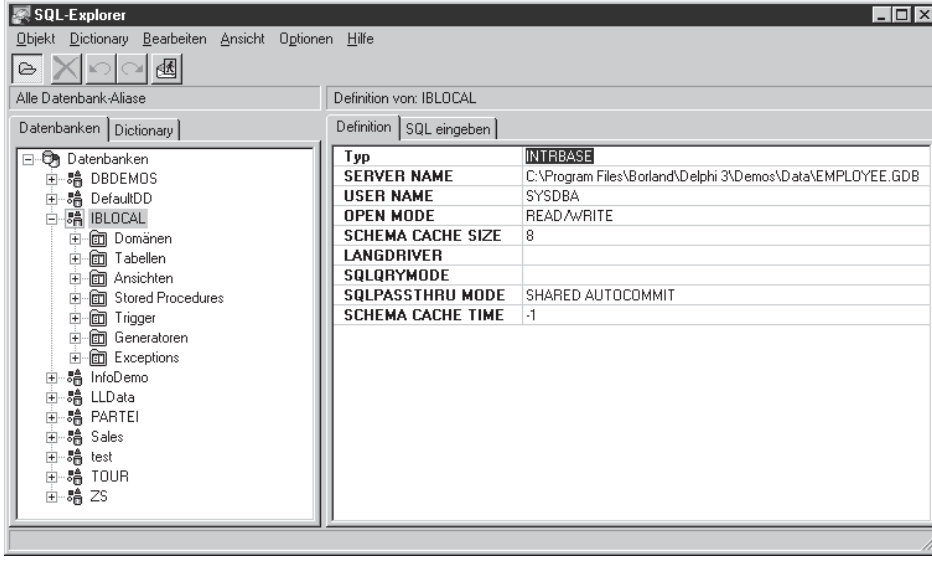


Bild 1.18: Der Datenbankexplorer

- InterBase *Interactive SQL* zum Definieren von Metadaten für den Local InterBase Server und den InterBase Server for Workgroups.

Diese drei Programme können zwar nach wie vor verwendet werden, ihre Funktionalität wurde jedoch im *Datenbankexplorer* (bei der Client-Server-Suite im *SQL-Explorer*) zusammengefaßt. Auf den ersten Blick ist die Bedienung ein wenig ungewohnt, sobald man jedoch dieses Programm beherrscht, möchte man es nicht mehr missen.

Aliase einrichten und ändern

Wie Bild 1.18 zeigt, zeigt der Datenbankexplorer seine Informationen in einer baumartigen Struktur an. In der ersten Verzweigung werden alle eingerichteten Datenbank-Aliase angezeigt. An dieser Stelle lassen sich auch neue Aliase einrichten oder die Daten bestehender Aliase ändern. Soll beispielsweise bei einer InterBase-Datenbank eine andere Datenbankdatei verwendet werden, dann wird in der Zeile *Server Name* ein anderer Pfad angegeben (die Datenbank muß verständlicherweise vorher geschlossen werden).

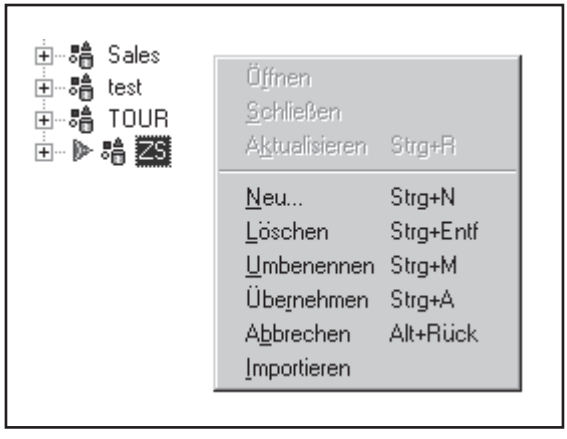


Bild 1.19: Geänderte Konfiguration

Datenbanken, bei denen auf diese Weise die Konfiguration geändert wurde, werden mit einem roten Dreieck gekennzeichnet, siehe Bild 1.19. Wird im Kontextmenü der Menüpunkt ÜBERNEHMEN aufgerufen, dann werden diese Weise die geänderten Konfigurationsdaten gespeichert. Mit Hilfe des Kontextmenüs (oder des entsprechenden Tastenkürzels) lassen sich auf neue Datenbank-Aliase definieren, sowie bestehende Aliase umbenennen beziehungsweise löschen.

Metadaten ermitteln

Im Strukturbaum der Datenbank werden alle Metadaten der jeweiligen Datenbank angezeigt. Bild 1.20 zeigt beispielsweise die Struktur der Tabelle *Employee* in der InterBase-Beispieldatenbank *Employee.gdb*. Die Informationen beschränken sich dabei nicht auf die Spaltennamen und deren Domänen, sondern es werden beispielsweise auch alle Primär- und Fremschlüssel angezeigt.

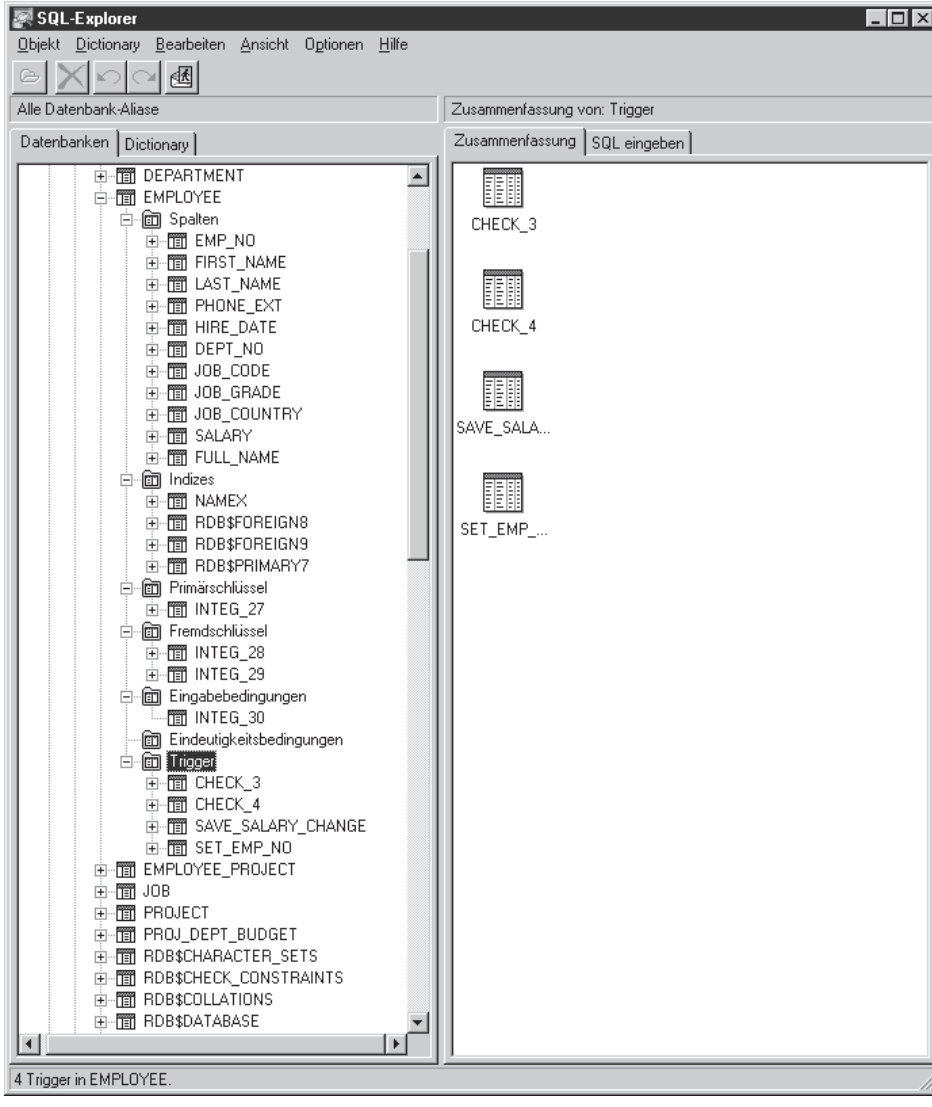


Bild 1.20: Struktur einer Tabelle

Systemtabellen anzeigen

Nicht ganz ungefährlich ist die Möglichkeit, sich alle Systemtabellen von Client-Server-Datenbanken anzeigen zu lassen – und diese auch gleich zu editieren. Bevor Sie diese Option nutzen, sollten Sie lieber ein Backup ziehen.

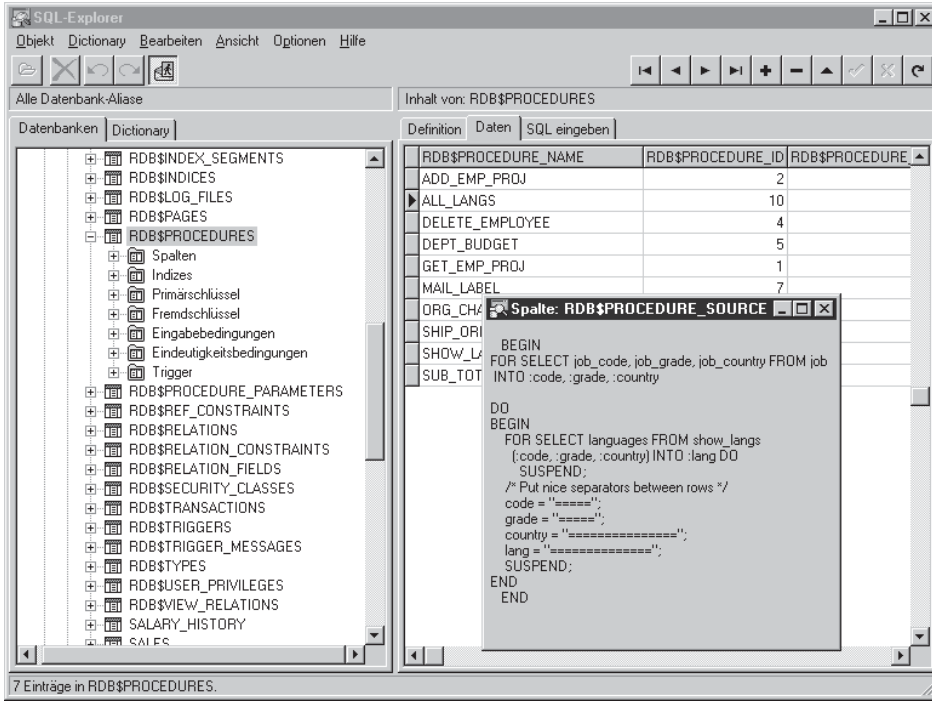


Bild 1.21: Anzeige einer Systemtabelle

Werden bei Ihnen keine Systemtabellen angezeigt, so können Sie dies mit ANSICHT | SYSTEMDATEN ändern (anschließend im Kontextmenü AKTUALISIEREN aufrufen). In jedem Fall können Sie alle Datentabellen ansehen und editieren.

Metadaten erstellen

Mit Hilfe der Seite *SQL eingeben* lassen sich auch alle Definitionen der Metadaten vornehmen. So lassen sich beispielsweise Tabellen, VIEWs und STORED PROCEDUREs erstellen, ändern und löschen.

Allerdings sind alle diese Definitionen (noch) als SQL-Anweisungen einzugeben – ich gehe davon aus, daß in der nächsten Version dieses Programms sich wenigstens Tabellen und VIEWs dialogorientiert erstellen und per Menübefehl löschen lassen.

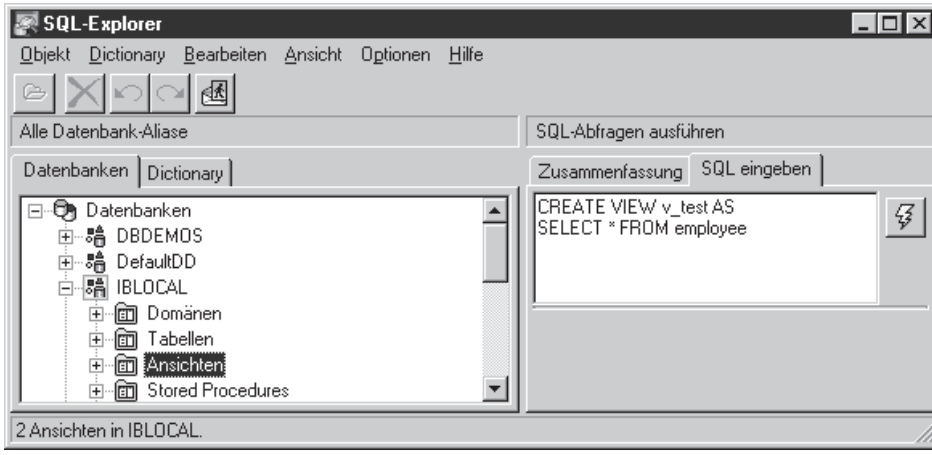


Bild 1.22: Erstellung einer VIEW

1.6 Datenbankapplikationen mit C++Builder

Das Programmieren von Datenbankanwendungen unterscheidet sich deutlich vom Programmieren anderer Anwendungen. Während herkömmliche Programme, insbesondere kleine Projekte, mehr oder minder »geradeaus« programmiert werden können, ist bei Datenbanken ein planmäßiges Vorgehen zwingend notwendig. C++Builder ist ein RAD (Rapid Application Development)-Werkzeug, in zehn Minuten können sie eine einfache Datenbankanwendung erstellen, die auf den ersten Blick ganz nett aussieht – allerdings nur auf den ersten Blick. Wir werden uns in Teil B und D noch sehr ausführlich mit dem planmäßigen Erstellen von Datenbankanwendungen befassen.

Datenbankapplikationen können Sie sowohl mit eigentlichen Programmierwerkzeugen (C++Builder, Delphi, Visual Basic) als auch mit den Makrosprachen von Datenbankprogrammen (dBase, Paradox) erstellen. Der Vorteil von Programmen wie C++Builder ist der, daß Sie eine verbreitete Hochsprache verwenden und meist auch flexibler sind – der Vorteil zeigt sich dann, wenn Sie Probleme zu lösen haben, die abseits der gewöhnlichen Datenbankproblematik liegen.

C++Builder oder Delphi

Im Prinzip sind C++Builder und Delphi nahezu identische Programmierwerkzeuge – mit dem wohl einzig gravierenden Unterschied, daß hier C und dort Pascal als Programmiersprache verwendet wird. Es gibt kaum einen plausiblen Grund, sich als C-Programmierer mit Delphi zu beschäftigen, umgekehrt sollte sich auch kein Pascal-Programmierer mit dem C++Builder abmühen (es sei denn, er ist Fachbuchautor und »übersetzt« sein erfolgreiches Delphi-Buch nach C++Builder).

Der einzige Grund, der C++Builder-Programmierer zu Delphi wechseln lassen könnte, wäre der Wunsch, 16-Bit-Anwendungen zu erstellen. Bei allen höheren Delphi-Versionen liegt nämlich Delphi 1.02 bei, mit dem sich 16-Bit-Anwendungen erstellen lassen.

2 Die Komponenten TTable und TDatasource

Für all diejenigen, die das erste Kapitel übersprungen haben, sei hier wiederholt, daß das Erstellen einer Datenbankapplikation sich in zwei Schritte gliedert:

- Das Erstellen der eigentlichen Datenbank, also das Definieren der Tabellen mit den jeweiligen Spalten. Dies geschieht mit der Datenbank-Oberfläche oder – bei Client-Server-Datenbanken – mit dem entsprechenden Werkzeug, beispielsweise *Windows-ISQL*.
- Das Programmieren der eigentlichen Applikation, vereinfacht gesagt, der Oberfläche, mit der auf diese Datenbank zugegriffen werden soll. Dies geschieht mit dem C++Builder. (Gegebenenfalls muß noch ein Report-Generator integriert werden.)

Für solche Datenbankapplikationen gibt es beim C++Builder zwei Gruppen von Komponenten:

- Eine Gruppe mit datensensitiven Dialog-Elementen, welche (für meinen Geschmack etwas unglücklich) *Datensteuerung* genannt wurde. Gesteuert wird lediglich mit der Komponente *DBNavigator*, ansonsten befinden sich in dieser Gruppe Komponenten wie Edit-Felder (*DBEdit*), Label (*DBText*), Tabellen (*DBGrid*) usw., welche für das Anzeigen von Informationen aus einer Datenbank vorbereitet sind.
- In der Gruppe *Datenzugriff* befinden sich nicht-visuelle Komponenten, welche die Verbindung zwischen den datensensitiven Dialog-Elementen und der Datenbank herstellen.

In diesem Kapitel sollen die Komponenten *TTable* und *TDatasource* behandelt werden, die zur Gruppe Datenzugriff gehören, im nächsten Kapitel geht es um die ebenfalls zu dieser Gruppe gehörende Komponente *TQuery*. Der Umgang mit datensensitiven Dialog-Elementen wird in Kapitel 4 erläutert.

Das Zusammenwirken der einzelnen Komponenten

Am Anfang mag die Vielzahl der benötigten Komponenten etwas verwirrend sein, doch im Prinzip ist alles ganz simpel, Bild 2.1 schafft hier Klarheit.

- Ausgangspunkt ist die Datenbank, die aus mehreren Tabellen manchmal auch nur aus einer Tabelle besteht.

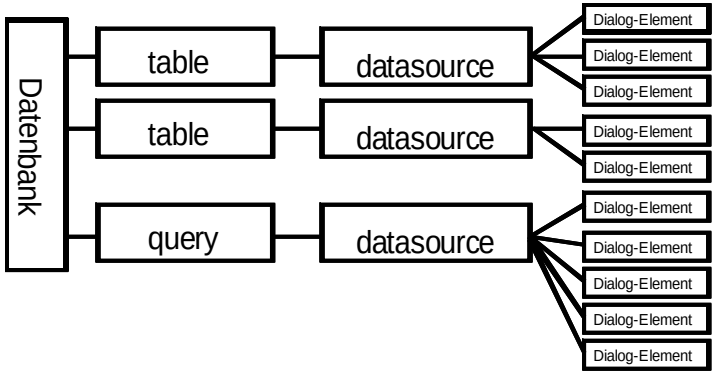


Bild 2.1: Zusammenwirken der einzelnen Komponenten

- Auf die Tabellen dieser Datenbank greifen die Komponenten *TTable* und *TQuery* zu. Der Unterschied zwischen den beiden Komponenten ist (grob vereinfacht) folgender:
  - Die Komponente *TTable* greift auf eine ganze Tabelle zu, um Datensätze zu lesen, hinzuzufügen, zu ändern oder zu löschen.
  - Die Komponente *TQuery* greift auf eine Teilansicht der Tabelle zu, um Datensätze zu lesen. Dabei wird die Standard-Abfragesprache SQL verwendet. Eine solche Anweisung könnte beispielsweise anordnen, alle (Adressen-)Datensätze auszugeben, bei denen der Ort den Wert *Berlin* hat.

    Auch bei mit *TQuery* ist es möglich, Datensätze hinzuzufügen, zu ändern oder zu löschen.

  In einem Formular können mehrere Komponenten vom Typ *TTable* und / oder *TQuery* vorhanden sein, welche auf dieselbe oder auf verschiedene Datenbanken zugreifen.
- Die gewonnene Datenmenge (bei *TTable* meist eine ganze Tabelle, bei *TQuery* eine Teilmenge) wird nun an die Komponente *TDatasource* weitergegeben, welche die Daten an die entsprechenden Dialog-Elemente weiterleitet.
- Die Dialog-Elemente schließlich geben die Information auf dem Bildschirm aus und nehmen Eingaben entgegen.

Man könnte sich nun fragen, warum man die Datenzugriffs-Komponenten benötigt und nicht gleich die Dialog-Elemente direkt mit der Datenbank verbindet. Dazu eine (unter vielen möglichen) Begründung: Man möchte beispielsweise die Adressen aller Personen anzeigen, die in Berlin in der Dolgenseestraße wohnen. Dazu würde man folgenden SQL-Text formulieren:

```
SELECT * FROM adressen WHERE
(ort = "Berlin") AND (straße = "Dolgenseestraße")
```

Würde man Dialog-Komponenten einsetzen, welche direkt auf eine Datenbank zugreifen, dann müßte man jeder dieser Komponenten diesen (hier wirklich noch sehr übersichtlichen) SQL-Text zuweisen. Die Komponente würde dann bei der Datenbank anfragen, die entsprechenden Datensätze erhalten und daraus das Feld herausfiltern, das sie anzuzeigen hätte.

Nun besteht eine Adresse ja aus mehreren Elementen (Vorname, Nachname, Straße, PLZ, Ort usw.), so daß in der Regel auch mehrere Dialog-Komponenten eingesetzt werden müßten; wenn diese nun alle ihre Anfragen an die Datenbank richten würden, wäre diese ganz schön beschäftigt. Des weiteren müßte man dann relativ aufwendig koordinieren, damit alle Komponenten denselben Datensatz anzeigen.

Auch für die Trennung von den Komponenten *TTable* und *TQuery* einerseits und *TDatasource* andererseits gibt es durchaus plausible Gründe, auch hier ein Beispiel: In ein Formular sollen Adressen eingegeben werden, die entsprechenden Dialog-Elemente sind deshalb mit *TTable1* verbunden. Nun soll auf diesem Formular vorübergehend eine Abfrage ausgegeben werden. Beim derzeitigen Konzept leitet man einfach die Eigenschaft *dataset* der Komponente *datasource1* auf *TQuery1* um:

```
datasource1->dataset = query1;
```

Wären die einzelnen Dialog-Elemente direkt verbunden, müßte man formulieren:

```
DBText1->dataset = query1;
DBEdit2->dataset = query1;
DBEdit3->dataset = query1;
DBEdit4->dataset = query1;
DBEdit5->dataset = query1;
DBEdit6->dataset = query1;
...
```

Auch dann, wenn wieder auf *table1* zurückgestellt werden muß, wäre der Aufwand entsprechend hoch.

Bevor wir uns nun en detail mit den Eigenschaften, Ereignissen und Methoden von *TTable* und *TDatasource* beschäftigen, wollen wir eine kleine Anwendung programmieren.

2.1 Erstellen von Testdatensätzen

Um eine Datenbank wirklich testen und optimieren zu können, ist eine Datenmenge erforderlich, welche der Größenordnung der später verwendeten Datenmenge in etwa entspricht. Ein Suchvorgang beispielsweise, der bei 20 Datensätzen noch durchaus schnell ist, kann den Anwender bei 10 000 Datensätzen leicht zur Verzweiflung treiben.

Gerade wenn man für Firmen Applikationen erstellt, sieht man sich oft mit dem Problem konfrontiert, daß diese nur höchst ungern ihre Daten herausgeben (aus verständlichen Gründen). Andererseits haben die meisten Programmierer allenfalls eine mittelgroße Adreßdatei, aber höchstens in Ausnahmefällen eine – sagen wir einmal – Artikeldatenbank mit 70 000 Einträgen (und das wäre noch eher harmlos). Der Versuch, solch eine Datenbank selbst einzugeben, dürfte allein aus Zeitgründen unrealistisch sein.

Nun besteht aber auch die Möglichkeit, sich eine solche Datenmenge vom Rechner generieren zu lassen. Ein entsprechendes Programm wollen wir nun erstellen. In diesem Fall soll eine simple Adressen-Datenbank generiert werden. Das Programm kann aber leicht so abgeändert werden, daß beliebige Datenmengen anderer Art erzeugt werden können.

2.1.1 Erstellen der Tabellen

Zunächst benötigen wir drei Tabellen, welche die folgenden Eigenschaften aufweisen:

Feldname	Typ	Größe	Schlüssel
Nummer	+		*
Vorname	A	20	
Nachname	A	20	
Straße	A	20	

Tabelle 2.1: Namen.DB

In die Tabelle *Namen.DB* geben Sie jeweils rund 100 Vornamen, Nachnamen und Straßennamen (ohne Hausnummern) ein. Die Einträge in den verschiedenen Spalten eines Datensatzes benötigen keinen Zusammenhang, sie werden später ohnehin zufällig durcheinandergemischt.

Feldname	Typ	Größe	Schlüssel
Nummer	+		*
PLZ	A	6	
Ort	A	20	
Vorwahl	A	5	

Tabelle 2.2: Orte.DB

In die Tabelle *Ort.DB* geben Sie rund 30 Orte mit den dazugehörigen (!) Postleitzahlen und Telefon-Vorwahlen ein.

Feldname	Typ	Größe	Schlüssel
Nummer	+		*
Vorname	A	20	
Nachname	A	20	
Straße	A	25	
PLZ	A	6	
Ort	A	20	
Tel1	A	20	
Tel2	A	20	
Fax	A	20	

Tabelle 2.3: Testadr.DB

Die Tabelle *Testadr.DB* benötigt keine Einträge, denn diese wollen wir ja nun generieren. Zu diesem Zweck wird ein Formular nach Bild 2.2 erstellt und mit Komponenten bestückt.

Dabei werden die Komponenten in *GroupBox1* (links oben) mit *Namen.DB* verbunden, die Komponenten in *GroupBox2* (rechts oben) mit *Ort.DB*, die Komponenten in *GroupBox3* (unten) mit *Testadr.DB*. Wenn hierbei Schwierigkeiten auftreten sollten, dann betrachten Sie die entsprechenden Dateien auf der beiliegenden CD.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  randomize();
}
```

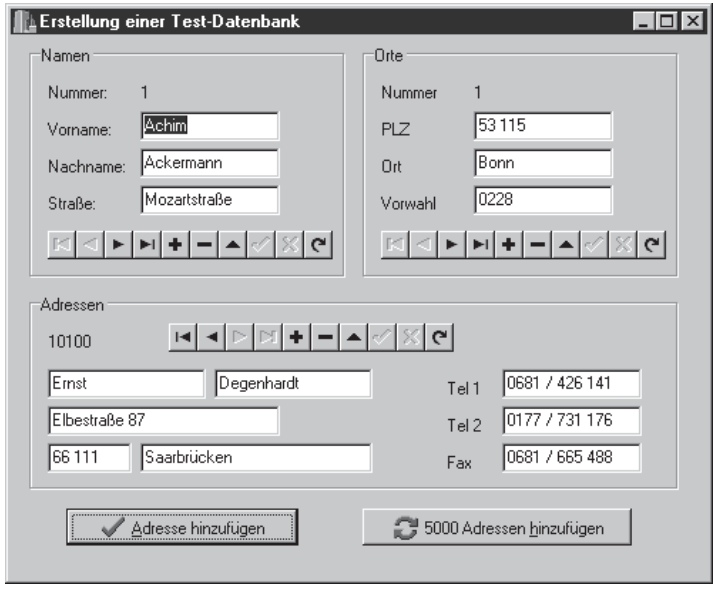


Bild 2.2: Das Formular des Testdaten-Generators

Zunächst eine sehr simple Prozedur: Das Mischen der Ausgangsdaten soll mit der Funktion *random* vorgenommen werden. Bevor der interne Zufallsgenerator eingesetzt werden darf, muß er mit dem Befehl *randomize* initialisiert werden, was automatisch beim Erstellen des Formulars geschieht.

Die Prozedur *BitBtn1Click* erstellt einen einzelnen neuen Datensatz.

```
                TForm1::BitBtn1Click(TObject *Sender)
{
  int i, j, l;
  AnsiString vorwahl, nummer, handy;
  Table3->Append();
```

Mit dem Befehl *Append* wird die Datenmenge geöffnet, es können nun weitere Datensätze angehängt werden.

```
// Vorname, Nachname, Straße, Hausnummer
  Table1->FindKey(new TVarRec(random(94) + 1), vtInteger);
  Table3->FieldByName("Nachname")->AsString
    = Table1->FieldByName("Nachname")->AsString;
  Table1->FindKey(new TVarRec(random(94) + 1), vtInteger);
  Table3->FieldByName("Vorname")->AsString
```

```
    = Table1->FieldByName("Vorname")->AsString;
  Table1->FindKey(new TVarRec(random(94) + 1), vtInteger);
  Table3->FieldByName("Straße")->AsString
    = Table1->FieldByName("Straße")->AsString + " " + (random(94) +
1);
```

Mit dem Befehl *FindKey* wird ein Datensatz anhand seines Primärschlüssels gesucht, das ist in diesem Fall die fortlaufende *Nummer*. Der Datensatz, welcher eingefügt wird, wird mit dem Befehl *random* zufällig ausgewählt; die Addition von eins stellt sicher, daß nicht nach dem Datensatz Null gesucht wird, den es hier nicht gibt.

Daß hier der Befehl *FieldByName("Vorname")* statt des Befehls *Fields[1]* verwendet wurde, hat folgenden Grund: Es wäre möglich, daß später einmal in die Tabelle *Namen.DB* vor *Vorname* eine Spalte für den akademischen Grad eingefügt wird, welche dann den Index eins hätte (die Zählung beginnt bei Null). Beim Befehl *Fields[1]* würde dann diese Spalte eingefügt, während beim Befehl *FieldByName("Vorname")* das Programm nach wie vor korrekt arbeiten würde, denn schließlich hätte sich die Bezeichnung der Spalte nicht geändert.

An den Straßennamen wird noch eine zufällige Hausnummer angehängt.

```
// PLZ, Ort und Vorwahl
  Table2->FindKey(new TVarRec(random(28) + 1), vtInteger);
  Table3->FieldByName("PLZ")->AsString
    = Table2->FieldByName("PLZ")->AsString;
  Table3->FieldByName("Ort")->AsString
    = Table2->FieldByName("Ort")->AsString;
  vorwahl = Table2->FieldByName("Vorwahl")->AsString;
```

Bei der Auswahl des Ortes muß nur eine Zufallszahl generiert werden, schließlich gehören Postleitzahl, Ort und Vorwahl zusammen. Die Vorwahl wird der gleichnamigen Variablen zugewiesen, die später den Telefonnummern vorangestellt wird.

```
// Telefonnummern
  l = strlen(&vorwahl[1]);
  for(j=1; j<=3; j++)
  {
    nummer = "";
    switch(l)
    {
      case 3: for(i=1; i<=9; i++)
        {
```

```
        if((i==4)+(i==7))
          nummer = nummer + " ";
        else
          nummer = nummer + (random(9) + 1);
      }
      break;
    case 4: for(i=1; i<=7; i++)
      {
        if(i==4)
          nummer = nummer + " ";
        else
          nummer = nummer + (random(9) + 1);
      }
      break;
    case 5: for(i=1; i<=6; i++)
      {
        if(i==3)
          nummer = nummer + " ";
        else
          nummer = nummer + (random(9) + 1);
      }
      break;
  } //switch(l)
  switch(j)
  {
    case 1: Table3->FieldByName("Tel1")->AsString
      = vorwahl + " / " + nummer;
      break;
    case 2:
      i = random(9);
      switch(i)
      {
        case 1: handy = "0171";
          break;
        case 2: handy = "0172";
          break;
        case 6: handy = "0161";
          break;
        case 7: handy = "0177";
          break;
        default: handy = "k";
```

```
        } //switch i
        if (handy == "k")
          Table3->FieldByName("Tel2")->AsString = "";
        else Table3->FieldByName("Tel2")->AsString
          = handy + " / " + nummer;
        break;
      case 3: Table3->FieldByName("Fax")->AsString
        = vorwahl + " / " + nummer;
        break;
    } //switch(j)
  } //for(j=1; j<=3; j++)
  Table3->Post();
} // void __fastcall TForm1::BitBtn1Click(TObject *Sender)
```

Die erste Routine zur Erstellung der Telefonnummern hat drei Durchläufe: einen für die Nummer *Tel1*, einen für die Handy-Nummer *Tel2*, und einen dritten für die Fax-Nummer. Es wird hier einfach unterstellt, daß alle unsere Personen einen Fax-Anschluß haben, ob eine Person ein Handy hat, entscheidet wieder eine *random*-Funktion.

Zunächst wird festgestellt, wie viele Stellen die Vorwahl hat. Bei drei Stellen wird eine siebenstellige Nummer vergeben, bei vier Stellen eine sechsstellige, und bei einer fünfstelligen Vorwahl hat auch die Durchwahl fünf Stellen; die Existenz sechsstelliger Vorwahlen werden wir hier ignorieren.

Von der Vorstellung, daß Telefonnummern ganz normale Zahlen seien (und auch so gespeichert werden könnten), muß man sich ohnehin verabschieden, die Datenbank würde aus 0305134505 ganz gnadenlos 305134505 machen. Außerdem sind endlos lange Zahlenreihen nicht gerade leicht und damit auch nicht fehlerfrei zu erfassen. Deshalb wird man diese Telefonnummer normalerweise in etwa so formatieren: 030 / 513 45 05.

Was das für die Suchen-Funktion bedeutet, insbesondere dann, wenn die Leerzeichen nicht einheitlich eingefügt werden, besprechen wir im nächsten Kapitel. Hier interessiert uns nur, daß bei siebenstelligen Zahlen nach der dritten und fünften Ziffer ein Leerzeichen eingefügt wird (bei sechsstelligen Zahlen nach der dritten Ziffer, bei fünfstelligen Zahlen nach der zweiten Ziffer). Die einzelnen Ziffern werden zufällig erzeugt und zu einem *AnsiString* zusammengefügt. Dieser wird dann zusammen mit der Vorwahl und dem Schrägstrich dem Datenfeld *tel1* bzw. *fax* zugewiesen.

Bei der Handy-Nummer *tel2* wird zunächst zufällig ein Wert für die Variable *i* generiert. Je nach dem Inhalt dieser Variable hat die betreffende Person C-, D1-, D2- oder E-Netz – oder aber überhaupt kein Funktelefon. Dementsprechend wird auch die Variable *handy* gesetzt, in welcher die Vorwahl des Funktelefonnetzes

gespeichert wird. Als Durchwahl wird dann wieder eine siebenstellige Zahl mit den dazugehörenden Formatierungsleerzeichen erzeugt.

Abschließend werden die Einträge mit *Table3->Post* in die Datenbank geschrieben. Das Generieren eines Datensatzes geht nun zwar deutlich schneller, als wenn man ihn von Hand eingeben würde, der Gedanke an 10 000 Klicks mit der linken Maustaste ist aber immer noch keine besonders erfreuliche Perspektive. Wir wollen deshalb eine Prozedur erstellen, welche gleich 5000 Datensätze hintereinander erzeugt.

```
void __fastcall TForm1::BitBtn2Click(TObject *Sender)
{
  int i;
  for(i=1; i<=5000; i++)
  {
    BitBtn1Click(Sender);
    Application->ProcessMessages();
  }
} // void __fastcall TForm1::BitBtn2CliTObject *Sender)
```

Die Prozedur ist im Prinzip nichts anderes als eine Schleife, die 5000 mal durchlaufen wird und dabei die Prozedur *BitBtn1Click* aufruft. Damit dabei stets die Anzeige aktualisiert wird, wird die Anweisung *Application->ProcessMessages* eingefügt. Da Graphikausgabe immer besonders lange braucht, ist diese Prozedur auch ziemlich langsam – bei mir auf dem Rechner benötigte sie 5 Minuten und 10 Sekunden.

```
void __fastcall TForm1::BitBtn2Click(TObject *Sender)
{
  int i;
  DataSource1->Enabled = false;
  DataSource2->Enabled = false;
  DataSource3->Enabled = false;
  Screen->Cursor = crHourGlass;
  for(i=1; i<=5000; i++)
  {
    BitBtn1Click(Sender);
    Application->ProcessMessages();
  }
  Screen->Cursor = crDefault;
  DataSource1->Enabled = true;
  DataSource2->Enabled = true;
  DataSource3->Enabled = true;
} // void __fastcall TForm1::BitBtn2Click(TObject *Sender)
```

Für die zweiten 5000 Datensätze wollen wir die Prozedur nun so ergänzen, daß alle *TDataSource*-Komponenten von den Datenbanken abgekoppelt werden, daß die Änderungen somit zwar durchgeführt, aber nicht mehr angezeigt werden. Damit nicht der Eindruck entsteht, daß das Programm abgestürzt ist, wird der Cursor während der Ausführung in eine Sanduhr umgewandelt. Die Ausführungszeit für 5000 Datensätze beträgt nun nur noch 22 Sekunden.

Nur unwesentlich länger (24 Sekunden) benötigt die Ausführung, wenn lediglich alle 100 Datensätze die Anzeige der Zieldatenbank aktualisiert wird. Da aber dann etwa alle halbe Sekunde ein neuer Datensatz angezeigt wird, sieht der Anwender, daß die Schleife noch korrekt arbeitet.

```
void __fastcall TForm1::BitBtn2Click(TObject *Sender)
{
  int i, j;
  div_t x;
    ...
  for(i=1; i<=5000; i++)
  {
    BitBtn1Click(Sender);
    x = div(i, 100);
    j = x.rem;
    if(j==0)
    {
      DataSource3->Enabled = true;
      Application->ProcessMessages();
      DataSource3->Enabled = false;
    }
  }
    ...
} // void __fastcall TForm1::BitBtn2Click(TObject *Sender)
```

2.2 Master-Detail-Verknüpfung

Nehmen wir einmal an, daß die gerade erstellte Tabelle *testadr.db* die Kundenliste einer fiktiven Firma sei. Diese Personen hätten bei dieser Firma Waren bestellt – und nicht immer gleich bezahlt. Deshalb gibt es die Tabelle *offenpo.db*, eine Liste der Forderungen, welche die Firma noch an diese Kunden hat.

Nun soll ein Formular erstellt werden, das die Kundenadressen gleichzeitig mit etwaigen Forderungen an diese Kunden anzeigt, so daß beispielsweise bei der Annahme einer Bestellung gesehen werden kann, wie es um die Zahlungsmoral

des Kunden bestellt ist (im einen oder anderen Fall wird man dann auf Vorkasse bestehen).

Prinzipiell handelt es sich um ein Formular mit zwei *TTable/TDataSource*-Kombinationen, von denen eine die Tabelle *testadr.db* und die andere die Tabelle *offenpo.db* anzeigt:

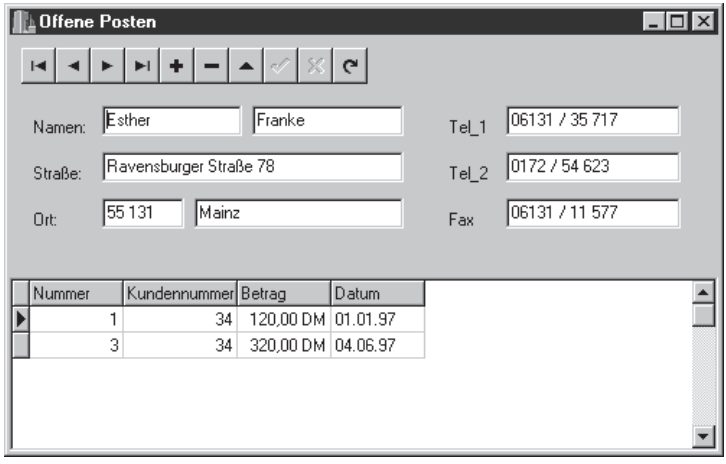


Bild 2.3: Ein Master-Detail-Formular

Nun sollen aber im *DBGrid* nicht alle Forderungen angezeigt werden, sondern nur die Forderungen gegenüber dem Kunden, der gerade angezeigt wird. Dazu muß eine Master-Detail-Verknüpfung hergestellt werden.

Zunächst wird der Eigenschaft *MasterSource* von *TTable2* der Wert *DataSource1* zugewiesen. Zum Setzen der Eigenschaft *MasterFields* startet man den Feldverbindungs-Designer (siehe Bild 2.4 auf der nächsten Seite). Hier wird nun eingestellt, nach welchem Feld sich die Master-Detail-Verbindung richten soll.

Momentan gibt es seitens der Tabelle *offenpo.db* nur die Möglichkeit, das Feld *nummer* zu wählen. Würde man dieses mit dem Feld *nummer* von *testadr.db* verbinden, dann würde zum ersten Kunden die erste Forderung (an eine ganz andere Person), zum zweiten Kunden die zweite Forderung usw. angezeigt. So geht es also nicht.

Wir stehen hier vor dem Problem, daß zur Verbindung mit einer anderen Tabelle ein indiziertes Feld benötigt wird. Momentan gibt es allerdings nur den Primärindex auf das Feld *nummer*; in der ComboBox *verfügbare Indizes* gibt es folglich nur den Eintrag *Primary*. Für die Tabelle *offenpo.db* muß somit ein Sekundärindex auf das Feld *kunde* eingerichtet werden (dies geschieht mit der Datenbankoberfläche, Näheres siehe Kapitel 1).

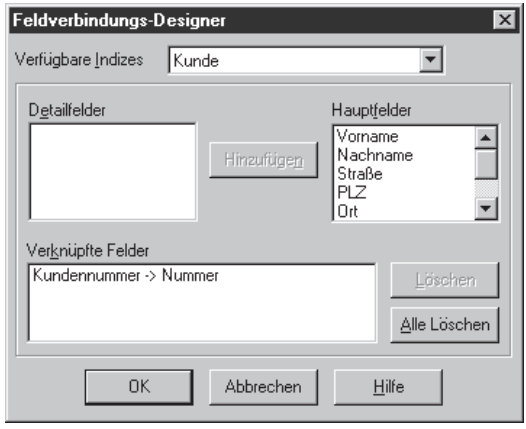


Bild 2.4: Erstellung einer Master-Detail-Verbindung

Nun kann unter den *verfügbaren Indizes* auch der Sekundärindex *Kundennummer* ausgewählt werden (oder wie man den Sekundärindex auch immer nennen möchte) und mit dem Feld *nummer* von *testadr.db* verbunden werden.

Es gibt auch die Möglichkeit, eine Master-Detail-Verknüpfung über mehrere Felder herzustellen. Nehmen wir einmal an, beide Tabellen würden ohne Kundennummer arbeiten, dafür gäbe es in der Tabelle *offenpo.db* die Felder *vorname* und *nachname* (mit den Sekundärindizes *S_vorname* und *S_nachname*). (Gründe gegen diese Vorgehensweise habe ich in Kapitel 1 bereits ausführlich dargelegt.)

Nun würde man das Feld *vorname* von *offenpo.db* mit dem Feld *vorname* von *testadr.db* verbinden, und ebenso die Felder *nachname* beider Tabellen. Zu einem Master-Datensatz werden nun alle Detail-Datensätze angezeigt, bei denen die beiden Feldverbindungs-Bedingungen zutreffen.

Bisweilen ist es schwierig bis unmöglich, für eine Tabelle einen Sekundärindex einzurichten. Beispielsweise wäre es denkbar, daß man laufend aktualisierte Tabellen ohne Sekundärindex von der Buchhaltung bekommt. Dem Anwender ist es in der Regel jedoch nicht zuzumuten, einen solchen Index von Hand einzufügen. Hier bietet sich die Komponente *TQuery* an, welche Master-Detail-Verknüpfungen ohne Index erlaubt.

2.3 Erstellen von Tabellen zur Laufzeit

Für gewöhnlich wird man die Tabellen mit der Datenbankoberfläche erstellen und vom Installationsprogramm kopieren lassen. Bisweilen kann es aber auch notwendig werden, Tabellen zur Laufzeit von *TTable* (oder *TQuery*) erstellen oder entfernen zu lassen.

Wir wollen dies an einem kleinen Beispielprogramm demonstrieren. Benötigt werden ein Formular mit einem Panel, darauf ein *TDBNavigator*, des weiteren eine *TTable/TDataSource*-Kombination sowie ein *DBGrid*. Die Datenbankkomponenten werden wie gewohnt miteinander verbunden.

Des weiteren benötigen wir ein Menü mit den Menüpunkten nach Bild 2.5

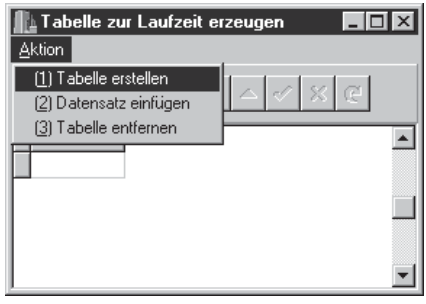


Bild 2.5: Die Menü-Punkte für das Beispielprogramm

Erstellen von Tabellen

Zum Erstellen der Tabelle sind folgende Schritte nötig:

```
void __fastcall TForm1::N1Tabelleerstellen1Click(TObject *Sender)
{
  TIndexOptions IndexOptions;
  Table1->Active = false;
  Table1->TableName = "Test_1.db";
  Table1->TableType = ttParadox;
```

Bevor Tabellennamen und Alias vorgegeben werden, muß die Eigenschaft *Active* *false* sein, ansonsten würde sich die Eigenschafte *TableName* nicht verändern lassen. Die Eigenschaft *TableType* ist hier optional, die BDE kann auch aus der Dateiendung erkennen, um welchen Tabellentyp es sich handelt.

```
//Spalten definieren
  Table1->FieldDefs->Clear();
  Table1->FieldDefs->Add("Nummer", ftAutoInc, 0, false);
  Table1->FieldDefs->Add("Vorname", ftString, 20, false);
  Table1->FieldDefs->Add("Nachname", ftString, 20, false);
  Table1->FieldDefs->Add("Straße", ftString, 20, false);
  Table1->FieldDefs->Add("PLZ", ftString, 6, false);
  Table1->FieldDefs->Add("Ort", ftString, 20, false);
  Table1->FieldDefs->Add("Tel1", ftString, 20, false);
```

```
  Table1->FieldDefs->Add("Tel2", ftString, 20, false);
  Table1->FieldDefs->Add("Fax", ftString, 20, false);
```

Im nächsten Schritt werden die einzelnen Felder definiert. Bei der Definition von Feldtypen, welche keine Längenangabe benötigen (Zahlen, Datum, Zeit...) muß eine Länge von Null angegeben werden, bei Feldtypen, welche der Längenangabe bedürfen (z.B. *ftString*), muß die Längenangabe erfolgen.

Als Feldtypen sind verfügbar: *(ftUnknown), ftString, ftSmallint, ftInteger, ftWord, ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime, ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, ftTypedBinary*

```
// Index definieren
  Table1->IndexDefs->Clear();
  IndexOptions << ixPrimary << ixUnique;
  Table1->IndexDefs->Add("Primary", "Nummer", IndexOptions);

// Tabelle erzeugen und öffnen
  Table1->CreateTable();
  Table1->Open();
} // TForm1::N1Tabelleerstellen1Click
```

Zuletzt wird der Index definiert, hier lediglich ein Primärindex auf das Feld *nummer*. Dann wird die Tabelle erstellt und geöffnet. In *DBGrid1* wird daraufhin eine Tabelle ohne Einträge angezeigt.

Datensatz einfügen

In die Tabelle soll nun per Prozedur ein Datensatz eingefügt werden:

```
void __fastcall TForm1::N2Datensatzeinfgen1Click(TObject *Sender)
{
  Table1->Active = false;
  Table1->TableName = "Test_1.db";
  Table1->Active = true;
  Table1->Append();
  Table1->FieldByName("Vorname")->AsString = "Ludwig";
  Table1->FieldByName("Nachname")->AsString = "Meier";
  Table1->FieldByName("Straße")->AsString = "Mozartstraße 32";
  Table1->FieldByName("PLZ")->AsString = "10 319";
  Table1->FieldByName("Ort")->AsString = "Berlin";
  Table1->FieldByName("Tel1")->AsString = "030 / 513 12 34";
  Table1->FieldByName("Tel2")->AsString = "";
```

```
  Table1->FieldByName("Fax")->AsString = "030 / 513 12 35";
  Table1->Post();
} // TForm1::N2Datensatzeinfgen1Click
```

Wir müssen zunächst berücksichtigen, daß diese Prozedur an zwei verschiedenen Stellen aufgerufen werden kann: entweder nach dem Erstellen der Tabelle oder nach einem Neustart des Programms. Im ersten Fall wären *DatabaseNamen* und *Table Name* bereits korrekt gesetzt, im zweiten Fall müßte dies noch geschehen. Jedes Programm sollte auf jede Handhabung seitens des Anwenders vorbereitet sein – so unerwartet sie auch sein mag.

Prinzipiell könnte man dann den Befehl *Edit* aufrufen, dann könnte aber immer nur der aktuelle Datensatz geändert werden – es würde also nie mehr als ein Datensatz eingefügt (wenigstens würde in eine leere Datenbank automatisch der erste Datensatz eingefügt).

Sinnvoller sind hier die Befehle *Insert* (fügt den Datensatz an der aktuellen Cursor-Position ein) oder – wie hier verwendet – *Append*, welcher neue Datensätze anhängt. Anschließend werden die einzelnen Daten der Eigenschaft *Fields* zugewiesen, abschließend wird mit *Post* bestätigt.

Da das Feld *Nummer* ein selbstinkrementierendes Feld ist, braucht ihm kein Wert zugewiesen zu werden. Deswegen ist es auch möglich, hier mehrmals einen Datensatz einzufügen. Wäre das Feld *Nummer* ein Integer-Feld, dann müßte ihm ein Wert zugewiesen werden – beim Versuch, den Menüpunkt *Datensatz einfügen* ein zweites Mal aufzurufen, würde dann ein Indexfehler auftreten.

Tabelle entfernen

Das Entfernen der Tabelle ist dann vergleichsweise simpel:

```
void __fastcall TForm1::N3Tabelleentfernen1Click(TObject *Sender)
{
  Table1->Active = false;
  Table1->TableName = "Test_1.db";
  Table1->DeleteTable();
}
```

2.4 Suchen und Filtern

Von den Möglichkeiten, die bei *TTable* zur Verfügung stehen, um einen bestimmten Datensatz zu finden oder die Menge der anzuzeigenden Datensätze einzuschränken, sollen nun die wichtigsten vorgestellt werden.

2.4.1 Einen Bereich setzen

Um die anzuzeigenden Datensätze einzuschränken, kann ein Bereich vorgegeben werden, dessen Kriterien die Datensätze erfüllen müssen, um angezeigt zu werden. Prinzipiell gibt es dafür zwei Vorgehensweisen, denen allerdings gemeinsam ist, daß die Felder, für welche Bereichsanfang und Bereichsende definiert werden, indiziert sein müssen. Die etwas umständlichere Variante funktioniert folgendermaßen:

```
void __fastcall TForm1::BereichKeyexclusive1Click(TObject *Sender)
{
  Table1->SetRangeStart();
  Table1->FieldByName("Nummer")->AsInteger = 100;
  Table1->SetRangeEnd();
  Table1->FieldByName("Nummer")->AsInteger = 200;
  Table1->ApplyRange();
}
```

Nach den Methoden *SetRangeStart* bzw. *SetRangeEnd* werden dem jeweiligen Indexfeld (oder – wenn mehrere Felder gemeinsam einen Index bilden – den Indexfeldern) die Werte für den Bereichsbeginn und das Bereichsende vorgegeben. Die Bereichsgrenzen gehören normalerweise auch noch zum Bereich, die Datensätze *100* und *200* sind also auch in der Datenmenge enthalten.

Es besteht jedoch die Möglichkeit, die Eigenschaft *KeyExlusive* auf *true* zu setzen, so daß die Bereichsgrenzen nicht mehr zum Bereich gehören. Wird in Integer-Feldern gesucht, dann ist diese Funktion nahezu überflüssig, man würde den Bereich einfach um je eins kleiner machen (hier also *101* und *199*). Nun ist es aber auch denkbar, daß mit String-Feldern gearbeitet wird, z.B. daß alle Städte nach *München* angezeigt werden sollen. Was ist nun der auf *München* folgende Städtename? (*Münchena*, aber dies muß einem ja nicht sofort einfallen.) Die folgende Prozedur liefert nun den Bereich zwischen *101* und *199*.

```
void __fastcall TForm1::BereichKeyexclusive1Click(TObject *Sender)
{
  Table1->SetRangeStart();
  Table1->FieldByName("Nummer")->AsInteger = 100;
  Table1->KeyExclusive = true;
  Table1->SetRangeEnd();
  Table1->FieldByName("Nummer")->AsInteger = 200;
  Table1->KeyExclusive = true;
  Table1->ApplyRange();
}
```

Die Methode SetRange

Etwas einfacher zu handhaben ist die Methode *SetRange*, welcher die Bereichsgrenzen gleich als Parameter übergeben werden; dabei entfällt auch die Methode *ApplyRange*. Die Methode erwartet als Parameter *TVarRec*-Arrays, welche hier mit dem Macro *OPENARRAY* erstellt werden.

```
void __fastcall TForm1::Bereich1Click(TObject *Sender)
{
  Table1->SetRange(OPENARRAY (TVarRec, (100)),
    OPENARRAY (TVarRec, (200)));
}
```

Bereich aufheben

Um wieder alle Datensätze anzuzeigen, muß lediglich die Methode *CancelRange* aufgerufen werden.

2.4.2 Datensätze suchen

Wenn in einer Tabelle von mehreren tausend Einträgen ein bestimmter gesucht werden soll, dann möchte man dies nicht unbedingt mit dem *DBNavigator* tun. Deshalb sollte man die Möglichkeit implementieren, nach bestimmten Datensätzen zu suchen. Das einfachste (und schnellste) ist es, wenn nach Werten in indizierten Feldern gesucht wird.

```
void __fastcall TForm1::Schlsselwert1Click(TObject *Sender)
{
  AnsiString s = InputBox
    ("Datensatz suchen", "nach Nummer",  5000");
  Table1->FindKey(OPENARRAY(TVarRec, (s)));
}
```

Hierzu muß lediglich die Methode *FindKey* verwendet werden, der als Parameter der zu suchende String übergeben wird. Statt der Zuweisung eines konstanten Werts wird hier die Funktion *InputBox* aufgerufen, die es dem Anwender ermöglicht, eigene Werte einzugeben.

Inkrementale Suche

In Spalten, die nicht indiziert sind, kann (mit *TTable*) nur gesucht werden, wenn ein Datensatz nach dem anderen daraufhin überprüft wird, ob er den Kriterien entspricht. Es ist wohl einleuchtend, daß dies seine Zeit in Anspruch nimmt. Die folgende Prozedur benötigt für die Suche dann auch ganze 62 Sekunden – wir werden gleich sehen, wie man das beschleunigen kann.

```
void __fastcall TForm1::inkrementa11Click(TObject *Sender)
{
  Screen->Cursor = crHourGlass;
  AnsiString k, s = InputBox
    ("inkrementale Suche", "nach Nummer:", "100");
  Table1->First();
// DataSource1->Enabled = false;
  for (int j = 0; /*j <= Table1->RecordCount*/ 200; j++)
    {
      k = Table1->FieldByName("Nummer")->AsString;
      if (k == s)
        break;
      Table1->Next();
    } // for (j = 0, j <= Table1->FieldCount, j++)
// DataSource1->Enabled = true;
  Screen->Cursor = crDefault;
} // TForm1::inkrementa11Click
```

Zunächst wird zum ersten Datensatz gesprungen, danach wird ein Datensatz nach dem anderen darauf geprüft, ob er der Such-Bedingung entspricht. Ist dies der Fall, dann wird die Schleife verlassen, anderenfalls wird der nächste Datensatz aufgerufen. Entsprechen mehrere Datensätze dem Suchkriterium (was hier im Beispiel nicht vorkommen kann), dann wird auf diese Weise nur der erste gefunden.

Nun zur Beschleunigung: Die meiste Zeit nimmt nicht das Suchen in Anspruch, sondern die laufende Aktualisierung der Bildschirmanzeige. Wird die Eigenschaft *DataSource1->Enabled* vor der Schleife auf *false* gesetzt (die erforderlichen Anweisungen sind im eben abgedruckten Listing als Kommentar vorhanden), dann wird die Ausführungszeit auf rund 690 msec gesenkt.

Soll in Spalten gesucht werden, in welchen derselbe Wert mehmals vorkommen kann, dann wird auf die Anweisung *First* verzichtet. Es wäre auch möglich, rückwärts zu suchen, man würde statt *Next* dann *Prior* verwenden.

2.4.3 Datensätze filtern

Soll eine Datenmenge gefiltert werden, dann bietet sich die Verwendung einer *TQuery*-Komponente an. Jedoch gibt es auch bei *TTable* (und auch bei anderen *TDataSet*-Komponenten, also auch bei *TQuery*) die Möglichkeit, den Datenbestand zu filtern. Zu diesem Zweck gibt es die Eigenschaft *Filter* sowie das Ereignis *OnFilterRecord*.

Die Eigenschaft Filter

Um eine Datenmenge zu filtern, wird der Eigenschaft *Filter* eine Anweisung zugewiesen, welche stark an die SQL-Syntax angelehnt ist, jedoch bei weitem nicht deren Funktionsumfang bietet.

```
void __fastcall TForm1::EigenschaftFilter1Click(TObject *Sender)
{
  EigenschaftFilter1->Checked = ! EigenschaftFilter1->Checked;
  Table1->Filter = "(Nachname = 'Müller') OR (Nachname = 'Borst')";
  // Table1->Filter = "(Nachname = 'Müller') AND (Vorname =
'Lisa')";
  //  Table1->Filter = "(Nummer < 10)";
  Table1->Filtered = EigenschaftFilter1->Checked;
}
```

Der Eigenschaft *Filter* wird die Anweisung als *AnsiString* zugwiesen. Dabei sind nicht nur die Vergleichs-Operatoren <, <=, 0, >, >= und <> erlaubt, sondern auch die logischen Operatoren AND, OR und NOT, welche das Verknüpfen der einzelnen Anweisungen erlauben. Um die Filter-Anweisung anzuwenden, wird die Eigenschaft *Filtered* auf *true* gesetzt.

Die als Kommentar gesetzten Anweisungen sind weitere Beispiele für korrekte Filter-Anweisungen. Bedauerlicherweise ist die Verwendung des LIKE-Operators nicht möglich.

Das Ereignis OnFilterRecord

Ist die Eigenschaft *Filtered* auf *true* gesetzt, dann wird bei jedem neu einzufügenden Datensatz das Ereignis *OnFilterRecord* aufgerufen. Der dazugehörenden Routine wird der Variablen-Parameter *Accept* übergeben; wird dieser auf *false* gesetzt, dann wird der betreffende Datensatz nicht aufgenommen.

Hier im Beispiel werden alle Datensätze angezeigt, welche mit *A* beginnen. Um jedoch auf diese Weise verschiedene Filter-Kriterien zu verwenden, womöglich auch noch mit frei wählbaren Parametern, sind einige Verrenkungen nötig – hier sollte man dann doch die Komponente *TQuery* einsetzen.

```
void __fastcall TForm1::Table1FilterRecord(TDataSet *DataSet, bool
&Accept)
{
  AnsiString s;
  if (OnFilterRecord1->Checked)
  {
    s = DataSet->FieldByName("Nachname")->AsString[1];
    if (s == "A")
```

```
      Accept = true;
    else
      Accept = false;
  } // if (OnFilterRecord1->Checked)
} // TForm1::Table1FilterRecord
```

Der Variablen *s* wird zunächst das erste Zeichen des jeweiligen Datensatzes zugewiesen. Anschließend wird geprüft, ob dieses gleich *A* ist.

2.5 Die Komponente TDataSet

Die Komponente *TTable*, aber auch *TQuery* und *TStoredProcedure* sind von *TDataSet* und *TDBDataSet* abgeleitet. Bild 2.6 zeigt diesen Sachverhalt in der *Symbolanzeige*, einem Tool, das bei C++Builder noch nicht vorhanden ist.

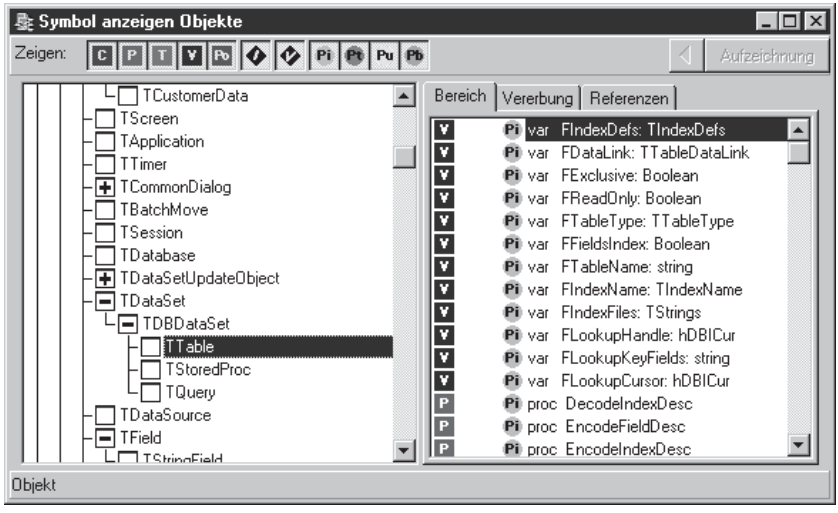


Bild 2.6: Der Stammbaum der Datenzugriffskomponenten

Im Gegensatz zu *TTable*, *TQuery* und *TStoredProcedure* sind die Komponenten *TDataSet* und *TDBDataSet* nicht zum direkten Gebrauch gedacht. Vielmehr implementieren sie diejenigen Eigenschaften, Methoden und Ereignisse, welche diesen drei Datenzugriffskomponenten gemeinsam sind.

Im folgenden werden die Methoden ignoriert, welche von *TObject*, *TPersistent* und *TComponent* implementiert werden.

2.5.1 Die veröffentlichten Eigenschaften von TDataSet

- Active (TDataSet)

```
__property bool Active;
```

  Mit der Eigenschaft *Active* läßt sich die Datenmenge vom Objektinspektor aus öffnen und schließen. Zur Laufzeit können dafür die Methoden *Open* und *Close* verwendet werden.

- AutoCalcFields (TDataSet)

```
__property bool AutoCalcFields;
```

  Die Eigenschaft *AutoCalcFields* bestimmt, wann das Ereignis *OnCalcFields* ausgelöst wird. Hat AutoCalcFields den Wert *true*, wird *OnCalcFields* in den folgenden Situationen ausgelöst:

  – Beim Öffnen einer Datenmenge.

  – Beim Wechseln des Fokus von einem visuellen Steuerelement zu einem anderen bzw. von einer Spalte einer datensensitiven Gitterkomponente zu einer anderen.

  – Beim Abrufen eines Datensatzes aus der Datenbank.

  Hat *AutoCalcFields* den Wert, dann wird *OnCalcFields* nur bei einem Aufruf der Methode *Post* ausgelöst.

- ChachedUpdates (TDataSet)

```
__property bool CachedUpdates;
```

  Die Eigenschaft *CachedUpdates* legt fest, ob zwischengespeicherte Aktualisierungen verwendet werden. Zwischengespeicherte Aktualisierungen vermindern bei Client-Server-Datenbanken die Netzwerkbelastung und die Belastung des Servers, bergen aber die Gefahr, daß Änderungen nicht ausgeführt werden können, weil andere Benutzer zwischenzeitlich denselben Datensatz bearbeitet haben.

  Bei der Verwendung von *TUpdateObject* muß die Eigenschaft *Cached Updates* auf *true* gesetzt werden.

- Filtered (TDataSet)

```
__property bool Filtered;
```

  Die Eigenschaft *Filtered* gibt an, ob eine Datenmenge gefiltert wird.

- FilterOptions (TDataSet)

```
__property TFilterOptions FilterOptions;
```

  Mit der Eigenschaft *FilterOptions* soll sich festlegen lassen, ob beim Filtern der Datenmenge die Groß- und Kleinschreibung beachtet wird (*foCaseInsensitive*), und ob Teilstrings berücksichtigt werden (*foNoPartialCompare*). Beide Optionen arbeiten in der aktuellen Version offensichtlich noch nicht korrekt.

- Filter (TDataSet)

```
__property System::AnsiString Filter;
```

  Mit Hilfe der Eigenschaft *Filter* läßt sich die Datenmenge filtern. Näheres ist in Kapitel 2.4.3 beschrieben

- Name (TComponent)
- Tag (TComponent)
- UpdateObject (TDataSet)

```
__property TDataSetUpdateObject* UpdateObject;
```

  Zur Komponente *TUpdateObject* siehe Kapitel 6.4.

2.5.2 Die öffentlichen Eigenschaften von TDataSet

- Bof (*TDataSet*, nur Lesen)

```
__property bool Bof;
```

  Die Eigenschaft *Bof* gibt an, daß sich der Cursor auf dem ersten Datensatz einer Datenmenge befindet. Dies ist beispielsweise nach Aufruf der Methode *First* der Fall. Wechselt man mit der Methode *Prior* vom zweiten auf den ersten Datensatz, dann ist die Eigenschaft *Bof = false* – erst nach einem weiteren *Prior*-Aufruf wird die Eigenschaft *Bof = true*.

  Des weiteren ist die Eigenschaft *Bof* bei neu geöffneten oder leeren Datenmengen gleich *true*.

  Im Gegensatz zu manchen Stellen in Handbuch und Online-Hilfe ist die Schreibweise *Bof* und nicht *BOF*.

- Bookmark (*TDataSet*)

```
__property System::AnsiString Bookmark;
```

  Zum Thema Lesezeichen mehr bei den Methoden von *TDataSet*.

- CanModify (*TDataSet*, nur Lesen)

```
__property bool CanModify;
```

  Mit Hilfe der Eigenschaft *CanModify* läßt sich feststellen, ob eine Datenmenge geändert werden kann. Dies ist beispielsweise nicht der Fall, wenn bei *TQuery* die Eigenschaft *RequestLive = false* ist, oder wenn sich die Abfrage nicht ändern läßt, was beispielsweise bei JOINs der Fall ist.

- DataSource (*TDataSet*, nur Lesen)

```
__property TDataSource* DataSource;
```

  Siehe Methode *IsLinkedTo*.

- DefaultFields (*TDataSet*, nur Lesen)

```
__property bool DefaultFields;
```

  Ist gleich *true*, wenn keine persistenten Feldkomponenten definiert wurden und somit beim Öffnen der Datenmenge dynamische Feldkomponenten automatisch generiert worden sind.

- Designer (*TDataSet*, nur Lesen)

```
__property TDataSetDesigner* Designer;
```

- Eof (*TDataSet*, nur Lesen)

```
__property bool Eof;
```

  Die Eigenschaft *Eof* gibt an, daß sich der Cursor auf dem letzten Datensatz einer Datenmenge befindet. Dies ist beispielsweise nach Aufruf der Methode *Last* der Fall. Wechselt man mit der Methode *Next* vom zweiten auf den ersten Datensatz, dann ist die Eigenschaft *Eof = false* – erst nach einem weiteren *Next*-Aufruf wird die Eigenschaft *Bof = true*.

  Des weiteren ist die Eigenschaft *Eof* bei leeren Datenmengen gleich *true* – hier ist dann auch die Eigenschaft *Bof* gleich *true*.

  Im Gegensatz zu manchen Stellen in Handbuch und Online-Hilfe ist die Schreibweise *Eof* und nicht *EOF*.

  Die Eigenschaften *Bof* und *Eof* werden insbesondere bei *while*-Schleifen verwendet, wenn sichergestellt werden soll, daß die Schleife beendet wird, sobald alle Datensätze der Datenbank durchlaufen sind.

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  FAbbruch = false;
  while((Table1->Eof == false) * (FAbbruch == false))
```

```
    {
      Table1->Next();
      Application->ProcessMessages();
    }
  } // TForm1::Button2Click
```

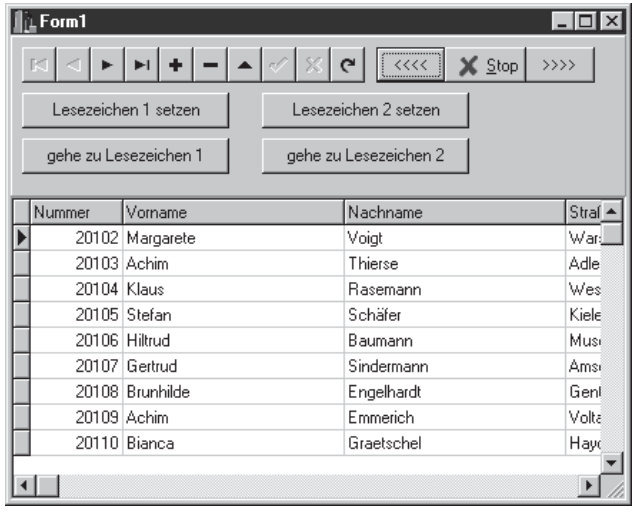


Bild 2.7: Die »Schnell-Browse-Buttons«

Die Funktion *Button2Click* durchläuft beispielsweise eine Datenmenge nach hinten, bis der Anwender das Feld *FAbbruch* auf *true* setzt. Auf diese Weise kann der Anwender schnell durch umfangreiche Datenmengen navigieren.

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
  FAbbruch = true;
}
```

- ExpIndex (*TDataSet*, nur Lesen)

```
__property bool ExpIndex;
```

Ist *true*, wenn die Datenmenge einen berechneten Index verwendet.

- FieldCount (*TDataSet*, nur Lesen)

```
__property int FieldCount;
```

Anzahl der Spalten in einer Datenmenge. Wird beispielsweise dann benötigt, wenn alle Spalten in einer Schleife durchlaufen werden sollen.

```
for(int i = 0; i < Table1->FieldCount; i++)
  Memo1->Lines->Add(Table1->Fields[i]->AsString);
```

- FieldDefs (*TDataSet*)

```
__property TFieldDefs* FieldDefs;
```

Zeigt auf die Liste der Felddefinitionen. Wird beispielsweise zum Erstellen einer Tabelle benötigt, siehe Kapitel 2.3.

- Fields (*TDataSet*)

```
__property TField* Fields[int Index]
```

Über die Eigenschaft *Fields* kann beispielsweise auf die Daten des aktuellen Datensatzes zugegriffen werden. Sinnvoller ist hier jedoch die Eigenschaft *FieldValues* oder die Methode *FieldByName*, weil diese auch dann auf dieselbe Spalte zugreift, wenn die zugrundeliegende Tabelle oder Abfrage umdefiniert worden ist.

```
Label1->Caption = Table1->Field[3]->AsString;
```

- FieldValues (*TDataSet*)

```
__property System::Variant FieldValues[System::AnsiString
FieldName];
```

Erlaubt den Lese- und Schreibzugriff auf einzelne Felder der Datenmenge, die über den Spaltennamen indiziert werden.

```
Table1->FieldValues["Nummer"] = "27 030";
```

- Found (*TDataSet*, nur Lesen)

```
__property bool Found;
```

Ist gleich *true*, wenn der letzte Aufruf einer der Methoden *FindFirst*, *FindLast*, *FindNext*, *FindPrior*, *First*, *Last*, *Next* oder *Prior* erfolgreich verlaufen ist.

- Handle (*TDataSet*, nur Lesen)

```
__property Bde::hDBICur Handle;
```

Gibt das BDE-Handle für diese Datenmenge an. Wird nur benötigt, wenn Funktionen der BDE-API aufgerufen werden.

- KeySize (*TDataSet*, nur Lesen)

```
__property unsigned short KeySize;
```

Mit *KeySize* können Sie die Größe des aktuellen Schlüssels der Datenmenge ermitteln. *KeySize* entspricht der Größe der Schlüsselfelder im Primärindex der Datenmenge. Bei dBASE-Tabellen mit berechnetem Index ist *KeySize* mit der Größe dieses Indexes identisch.

- Locale (*TDataSet*, nur Lesen)

```
__property void * Locale;
```

Mit *Locale* können Sie den BDE-Sprachtreiber einer Datenmenge bestimmen. Wird in einer Anwendung die BDE-API direkt aufgerufen, müssen gegebenenfalls Sprachtreiberinformationen als Funktionsparameter übergeben werden.

- Modified (*TDataSet*, nur Lesen)

```
__property bool Modified;
```

Ist *true*, wenn der Datensatz geändert, aber noch nicht *Post* oder *Cancel* aufgerufen worden ist.

- RecordNo (*TDataSet*, nur Lesen)

```
__property long RecNo;
```

Nummer des aktuellen Datensatzes. Die folgende Funktion zeigt die Nummer des aktuellen Datensatzes sowie die Gesamtzahl aller Datensätze an.

```
void __fastcall TForm1::DataSource1DataChange(TObject *Sender,
  TField *Field)
{
  Label1->Caption = "Datensatz " + IntToStr(Table1->RecNo) + "
von "
     + IntToStr(Table1->RecordCount) + " Datensätzen";
}
```

- RecordCount (*TDataSet*, nur Lesen)

```
__property long RecordCount;
```

Die Eigenschaft *RecordCount* gibt die Gesamtzahl der Datensätze in einer Datenmenge an.

- RecordSize (*TDataSet*, nur Lesen)

```
__property unsigned short RecordSize;
```

Größe (in *Byte*) des Puffers, der einem Datensatz der Datenmenge zugewiesen wird.

- State (*TDataSet*, nur Lesen)

```
__property TDataSetState State;
```

Die Eigenschaft *State* kann folgende Werte annehmen:

- *dsInactive:* Die Datenmenge ist geschlossen. Auf ihre Daten kann nicht zugegriffen werden.
- *dsBrowse:* Die Daten können angezeigt, jedoch nicht geändert werden. Dies ist der Standardmodus einer geöffneten Datenmenge.
- *dsEdit:* Der aktuelle Datensatz kann geändert werden.
- *dsInsert:* Ein neuer Datensatz kann eingefügt werden.
- *dsSetKey:* Nur für *TTable*. Die Datensatzsuche ist aktiviert oder eine *SetRange*-Operation wird durchgeführt. Eine eingeschränkte Datenmenge kann angezeigt, jedoch können keine Datensätze geändert oder eingefügt werden.
- *dsCalcFields:* Ein *OnCalcFields*-Ereignis wurde ausgelöst. Die nicht berechneten Felder können bearbeitet und neue Datensätze eingefügt werden.
- *dsUpdateNew:* Eine zwischengespeicherte Aktualisierung wird durchgeführt. Daten können weder bearbeitet noch eingefügt werden.
- *dsUpdateOld:* Eine zwischengespeicherte Aktualisierung wird durchgeführt. Daten können weder bearbeitet noch eingefügt werden.
- *dsFilter:* Ein *OnFilterRecord*-Ereignis wurde ausgelöst. Eine eingeschränkte Datenmenge kann angezeigt, jedoch können keine Datensätze geändert oder eingefügt werden.

- UpdateRecordTypes (*TDataSet*)

```
__property TUpdateRecordTypes UpdateRecordTypes;
```

Die Eigenschaft *UpdateRecordTypes* gibt den Datensatztyp an, der bei aktivierter Zwischenspeicherung in einer Datenmenge sichtbar ist. Standardmäßig wird eine Datenmenge mit der Menge *UpdateRecordTypes [rtModified, rtInserted, rtUnmodified]* erstellt. Es sind dann alle vorhandenen, geänderten und eingefügten Datensätze für den Benutzer sichtbar.

- UpdatesPending (*TDataSet*, nur Lesen)

```
__property bool UpdatesPending;
```

Die Eigenschaft *UpdatesPending* gibt an, ob der Puffer noch nicht eingetragene zwischengespeicherte Aktualisierungen enthält. Diese müssen gegebenenfalls mit der Methode *ApplyUpdates* in die Datenbank geschrieben werden.

2.5.3 Die Methoden von TDataSet

- ~TDataSet (*TDataSet*)

```
__fastcall virtual ~TDataSet(void);
```

Der Destruktor *~TDataSet* gibt den Speicher frei, der für die *TDataSet*-Instanz reserviert war.

- ActiveBuffer (*TDataSet*)

```
char * __fastcall ActiveBuffer(void);
```

Gibt einen Zeiger auf den Puffer des aktuellen Datensatzes zurück. Wird von vielen Datenmengenmethoden intern verwendet, muß normalerweise nie vom Benutzer aufgerufen werden.

- Append (*TDataSet*)

```
void __fastcall Append(void);
```

Die Methode *Append* hängt einen neuen leeren Datensatz am Ende der Datenmenge an. (Die Methode *Insert* fügt dagegen einen neuen Datensatz an der aktuellen Position des Datenzeigers ein. Da die Datenmenge normalerweise ohnehin nach einem Schlüssel sortiert wird, ist es egal, ob *Append* oder *Insert* verwendet wird.)

```
Table1->Append();
Table1->FieldByName("Nummer")->AsInteger = 30105;
Table1->FieldByName("Text")->AsString = "Test-Eintrag";
Table1->Post();
```

- AppendRecord (*TDataSet*)

```
void __fastcall AppendRecord(const System::TVarRec *Values,
  const int Values_Size);
```

Die Methode *AppendRecord* hängt einen neuen Datensatz an, schreibt die übergebenen Feldwerte hinein und bestätigt mit *Post*. Das bei der Methode *Append* angeführte Beispiel könnte man auch wie folgt formulieren:

```
Table1->AppendRecord(OPENARRAY (TVarRec, (30105, "Test-String")));
```

- ApplyUpdates (*TDataSet*)

```
void __fastcall ApplyUpdates(void);
```

Die Methode *ApplyUpdates* schreibt die anstehenden zwischengespeicherten Aktualisierungen in die Datenbank. Wird benötigt, wenn die Eigenschaft *CachedUpdates* = *true* ist.

- Cancel (*TDataSet*)

```
void __fastcall Cancel(void);
```

  Verwirft die vorgenommenen Änderungen an einem Datensatz (die Methode *Post* würde sie bestätigen).

- CancelUpdates (*TDataSet*)

```
void __fastcall CancelUpdates(void);
```

  Verwirft die anstehenden zwischengespeicherten Aktualisierungen und stellt den ursprünglichen Zustand wieder her.

- CheckBrowseMode (*TDataSet*)

```
void __fastcall CheckBrowseMode(void);
```

  Wird vor allem intern verwendet und stellt sicher, daß beim Wechsel des Datensatzes die Eigenschaft *State* den Wert *dsBrowse* hat. Näheres siehe in der Online-Hilfe.

- ClearFields (*TDataSet*)

```
void __fastcall ClearFields(void);
```

  Die Methode *ClearFields* löscht den Inhalt aller Felder des aktuellen Datensatzes.

- Close (*TDataSet*)

```
void __fastcall Close(void);
```

  Schließt die Datenmenge, die Eigenschaft *State* ist dann *dsInactive*. Zur Entwurfszeit kann dazu die Eigenschaft *Active* verwendet werden.

- CommitUpdates (*TDataSet*)

```
void __fastcall CommitUpdates(void);
```

  Die Methode *CommitUpdates* löscht den Puffer für zwischengespeicherte Aktualisierungen.

- ControlsDisabled (*TDataSet*)

```
bool __fastcall ControlsDisabled(void);
```

  Mit Hilfe von *ControlsDisabled* läßt sich feststellen, ob die mit der Datenmenge verbundenen Datensteuerungskomponenten aktiviert sind oder nicht.

- CursorPosChanged (*TDataSet*)

```
void __fastcall CursorPosChanged(void);
```

Die Methode *CursorPosChanged* setzt die Position des BDE-Cursors der Datenmenge auf -1. Dieser Wert entspricht keiner gültigen Position. *CurPosChanged* wird von den Methoden *Locate* und *Lookup* aufgerufen, damit sie bei Erfolg den Cursor auf den ersten übereinstimmenden Datensatz setzen.

- Delete (*TDataSet*)

```
void __fastcall Delete(void);
```

Löscht den aktuellen Datensatz. Was dabei sonst noch für Aktionen ausgeführt werden, kann in der Online-Hilfe nachgeschlagen werden.

- DisableControls (*TDataSet*)

```
void __fastcall DisableControls(void);
```

Deaktiviert die Anzeige in den datensensitiven Steuerelementen, die mit der Datenmenge verbunden sind. Dies bringt eine höhere Verarbeitungsgeschwindigkeit beispielsweise bei der inkrementalen Suche.

- Edit (*TDataSet*)

```
void __fastcall Edit(void);
```

Die Methode *Edit* ermöglicht das Bearbeiten der Daten in einer Datenmenge; siehe auch Online-Hilfe.

- EnbleControls (*TDataSet*)

```
void __fastcall EnableControls(void);
```

Aktiviert die Anzeige in den Datensteuerungskomponenten, macht die Methode *DisabledControls* wieder rückgängig.

- FetchAll (*TDataSet*)

```
void __fastcall FetchAll(void);
```

Normalerweise werden aus einer Datenbank nur diejenigen Datensätze abgefragt, welche auch angezeigt werden. Mit *FetchAll* werden alle Datensätze bis zum Ende der Datenmenge abgefragt und lokal gespeichert. Somit wird die Abfrage an einer Stelle zusammengefaßt, was beispielsweise bei Client-Server-Systemen die Netzwerkbelastung reduziert.

- FieldByName (*TDataSet*)

```
TField* __fastcall FieldByName(const System::AnsiString FieldName);
```

Ermöglicht den Zugriff auf die einzelnen Felder über den Spaltennamen und ergibt – im Gegensatz zur Eigenschaft *Fields* – auch dann korrekte Ergebnisse, wenn die Anordnung der Spalten geändert wurde.

- FindField (*TDataSet*)

```
TField* __fastcall FindField(const System::AnsiString FieldName);
```

Ermöglicht auch den Zugriff über den Spaltennamen. Während jedoch bei *FieldByName* eine Exception ausgelöst wird, wenn es den angegebenen Spaltennamen nicht gibt, liefert *FindField* den Wert NULL zurück.

- FindFirst (*TDataSet*)

```
bool __fastcall FindFirst(void);
```

Setzt den Datencursor auf den ersten Datensatz in einer gefilterten Datenmenge. Aktualisiert – im Gegensatz zur Methode *First* – die Filterung, zwischenzeitlich in die Datenmenge aufgenommene Datensätze werden also berücksichtigt. Dies gilt analog auch für *FindLast*, *FindNext* und *FindPrior*.

- FindLast (*TDataSet*)

```
bool __fastcall FindLast(void);
```

Setzt den Datencursor auf den letzten Datensatz in einer gefilterten Datenmenge. (Siehe auch *FindFirst*.)

- FindNext (*TDataSet*)

```
bool __fastcall FindNext(void);
```

Setzt den Datencursor auf den nächsten Datensatz in einer gefilterten Datenmenge. (Siehe auch *FindFirst*.)

- FindPrior (*TDataSet*)

```
bool __fastcall FindPrior(void);
```

Setzt den Datencursor auf den vorigen Datensatz in einer gefilterten Datenmenge. (Siehe auch *FindFirst*.)

- First (*TDataSet*)

```
void __fastcall First(void);
```

Die *Methode First* setzt den Datenzeiger auf den ersten Datensatz.

- FreeBookmark (*TDataSet*)

```
void __fastcall FreeBookmark(void * Bookmark);
```

Gibt den einem bestimmten Lesezeichen zugewiesenen Speicher frei.

- GetBookmark (*TDataSet*)

```
void * __fastcall GetBookmark(void);
```

Die Methode *GetBookmark* weist der aktuellen Cursorposition in einer Datenmenge ein Lesezeichen zu.

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  FLesezeichen1 = Table1->GetBookmark();
}
```

Vor der Verwendung muß das Lesezeichen in der Header-Datei deklariert werden.

```
class TForm1 : public TForm
{
  __published: // IDE-verwaltete Komponenten
    ...
  private: // Benutzer-Deklarationen
    void* FLesezeichen1;
    void* FLesezeichen2;
  public:  // Benutzer-Deklarationen
    __fastcall TForm1(TComponent* Owner);
};
```

- GetCurrentRecord (*TDataSet*)

```
bool __fastcall GetCurrentRecord(char * Buffer);
```

Liest den aktuellen Datensatz in einen Puffer ein.

- GetFieldList (*TDataSet*)

```
void __fastcall GetFieldList(Classes::TList* List,
  const System::AnsiString FieldNames);
```

Normalerweise ist es sinnvoller, direkt auf die einzelnen Feldwerte zuzugreifen. Wird jedoch eine Kopie der Daten benötigt, so kann diese mit *GetFieldList* erstellt werden.

- GetFieldNames (*TDataSet*)

```
void __fastcall GetFieldNames(Classes::TStrings* List);
```

Ruft eine Liste mit den Namen der Felder ab.

- GotoBookmark (*TDataSet*)

```
void __fastcall GotoBookmark(void * Bookmark);
```

Die Methode *GotoBookmark* setzt den Cursor auf den Datensatz, auf den ein bestimmtes Lesezeichen zeigt.

- Insert (*TDataSet*)

```
void __fastcall Insert(void);
```

Fügt einen neuen leeren Datensatz in eine Datenmenge ein. Siehe auch die Methode *Append*.

- InsertRecord (*TDataSet*)

```
void __fastcall InsertRecord(const System::TVarRec *Values,
  const int Values_Size);
```

Die Methode *InsertRecord* fügt einen neuen Datensatz ein, schreibt die übergebenen Feldwerte hinein und bestätigt mit *Post*. (Beispiel siehe bei der Methode *AppendRecord*.)

- IsLinkedTo (*TDataSet*)

```
bool __fastcall IsLinkedTo(TDataSource* DataSource);
```

Die Methode *IsLinkedTo* prüft, ob eine Datenmenge mit einer bestimmten Datenquelle verbunden ist.

- Last (*TDataSet*)

```
void __fastcall Last(void);
```

Setzt den Datenzeiger auf den letzten Datensatz einer Datenmenge.

- Locate (*TDataSet*)

```
bool __fastcall Locate(const System::AnsiString KeyFields,
  const System::Variant &KeyValues,  TLocateOptions  Options);
```

Mit der Methode *Locate* lassen sich Datensätze finden. Im Gegensatz zur *TTable*-Methode *FindKey* ist die Suche dabei nicht auf Primärschlüssel- und Indexfelder beschränkt.

```
void __fastcall TForm1::Locate1Click(TObject *Sender)
{
  TLocateOptions SearchOptions;
  SearchOptions << loPartialKey;
  Table1->Locate("Nachname", "Borst", SearchOptions);
}
```

- Lookup (*TDataSet*)

```
System::Variant __fastcall Lookup(const System::AnsiString
  KeyFields, const System::Variant &KeyValues,
  const System::AnsiString ResultFields);
```

Die Methode *Lookup* gibt die Feldwerte eines Datensatzes zurück, der mit den angegebenen Suchwerten übereinstimmt.

- MoveBy (*TDataSet*)

```
int __fastcall MoveBy(int Distance);
```

Die Methode *MoveBy* positioniert den Cursor in einer Datenmenge um eine bestimmte Anzahl Datensätze vor oder zurück. Die tatsächliche Anzahl der Datensätze, um die der Cursor versetzt wurde, wird zurückgegeben. In den meisten Fällen entspricht der Rückgabewert dem absoluten Wert von *Distance*. Wird jedoch vor Erreichen von *Distance* der Anfang oder das Ende der Datei erreicht, ist *Result* kleiner als *Distance*.

- Next (*TDataSet*)

```
void __fastcall Next(void);
```

Setzt den Cursor auf den nächsten Datensatz.

- Open (*TDataSet*)

```
void __fastcall Open(void);
```

Öffnet eine Datenmenge. Zur Entwurfszeit kann dasselbe Ziel erreicht werden, indem die Eigenschaft *Active* auf *true* gesetzt wird.

- Post (*TDataSet*)

```
void __fastcall Post(void);
```

Die Methode *Post* schreibt einen geänderten Datensatz in die Datenbank oder in den Aktualisierungspuffer.

- Prior (*TDataSet*)

```
void __fastcall Prior(void);
```

Setzt den Datenzeiger auf den vorhergehenden Datensatz.

- Refresh (*TDataSet*)

```
void __fastcall Refresh(void);
```

Die Methode *Refresh* aktualisiert eine Datenmenge durch erneutes Abrufen der Daten aus der Datenbank. Als Methode von *TQuery* aktualisiert *Refresh* die Abfrage nicht. Damit die aktuellsten Daten angezeigt werden, muß die Abfrage geschlossen und wieder geöffnet werden.

```
Query1->Close();
Query1->Open();
```

- Resync (*TDataSet*)

```
void __fastcall Resync( TResyncMode Mode);
```

Die Methode *Resync* ruft den aktuellen Datensatz sowie die anderen für die Anzeige benötigten Datensätze ab.

- RevertRecord (*TDataSet*)

```
void __fastcall RevertRecord(void);
```

Die Methode *RevertRecord* macht die Änderungen des aktuellen Datensatzes rückgängig. Sie funktioniert nur, wenn mit zwischengespeicherten Aktualisierungen gearbeitet wird.

- SetFields (*TDataSet*)

```
void __fastcall SetFields(const System::TVarRec *Values,
  const int Values_Size);
```

Weist den Feldern in einem Datensatz die angegebenen Werte zu. Hat die Datenmenge mehr Spalten als *Values* Einträge, werden die restlichen Spalten mit NULL-Werten aufgefüllt.

- UpdateCursorPos (*TDataSet*)

```
void __fastcall UpdateCursorPos(void);
```

Die Methode *UpdateCursorPos* setzt den Cursor in den aktuellen Datensatz. Sie wird in der Regel nur intern verwendet und braucht vom Benutzer nicht aufgerufen zu werden.

- UpdateRecord (*TDataSet*)

```
void __fastcall UpdateRecord(void);
```

Die Methode *UpdateRecord* löst bei einer Datensatzaktualisierung das entsprechende Datenereignis aus. Sie wird in der Regel nur intern verwendet und braucht vom Benutzer nicht aufgerufen zu werden.

- UpdateStatus (*TDataSet*)

```
TUpdateStatus __fastcall UpdateStatus(void);
```

Gibt den Aktualisierungsstatus einer Datenmenge zurück. Dieser kann folgende Werte annehmen:

– *usUnmodified:* Der Datensatzinhalt wurde nicht verändert.

– *usModified:* Der Datensatzinhalt wurde verändert.

– *usInserted:* Der Datensatz wurde eingefügt.

– *usDeleted:* Der Datensatz wurde gelöscht.

2.5.4 Die Ereignisse von TDataSet

Im Gegensatz zur Referenz der Eigenschaften und Methoden wird hier nicht die Deklaration, sondern die Parameterliste der dazugehörenden Ereignisbehandlungsroutine aufgeführt. Bei den *Afterxxx*- und *Beforexxx*-Ereignissen wurde auf eine Beschreibung verzichtet, da sich diese aus der Bezeichnung eindeutig ergibt.

- OnAfterCancel (*TDataSet*)

```
OnAfterCancel(TDataSet *DataSet)
```

- OnAfterClose (*TDataSet*)

```
OnAfterClose(TDataSet *DataSet)
```

- OnAfterDelete (*TDataSet*)

```
OnAfterDelete(TDataSet *DataSet)
```

- OnAfterEdit (*TDataSet*)

```
OnAfterEdit(TDataSet *DataSet)
```

- OnAfterInsert (*TDataSet*)

```
OnAfterInsert(TDataSet *DataSet)
```

- OnAfterOpen (*TDataSet*)

```
OnAfterOpen(TDataSet *DataSet)
```

- OnAfterPost (*TDataSet*)

```
OnAfterCancel(TDataSet *DataSet)
```

  Das Ereignis *OnAfterPost* kann dazu verwendet werden, um nach einer Änderung in der Datenmenge die damit verbundenen Abfragen zu aktualisieren:

```
void __fastcall TForm1::Table1AfterPost(TDataSet *DataSet)
{
  Query1->Close();
  Query1->Open();
}
```

- OnBeforeCancel (*TDataSet*)

```
OnBeforeCancel(TDataSet *DataSet)
```

- OnBeforeClose (*TDataSet*)

```
OnBeforeClose(TDataSet *DataSet)
```

- OnBeforeDelete (*TDataSet*)

```
OnBeforeDelete(TDataSet *DataSet)
```

Mit Hilfe des Ereignisses *OnBeforeDelete* kann das Löschen eines Datensatzes wie folgt verhindert werden:

```
void __fastcall TForm1::Table1BeforeDelete(TDataSet *DataSet)
{
  if(MessageBox(0, "Datensatz löschen?", "Bestätigen", MB_YESNO)
        != IDYES)
    Abort();
}
```

Solange das Programm unter dem C++Builder-Debugger läuft, wird dabei der Programmablauf unterbrochen, sobald das Programm jedoch eigenständig läuft, arbeitet diese Routine ohne Fehlermeldung.

- OnBeforeEdit (*TDataSet*)

```
OnBeforeEdit(TDataSet *DataSet)
```

- OnBeforeInsert (*TDataSet*)

```
OnBeforeInsert(TDataSet *DataSet)
```

- OnBeforeOpen (*TDataSet*)

```
OnBeforeOpen(TDataSet *DataSet)
```

- OnBeforePost (*TDataSet*)

```
OnBeforePost(TDataSet *DataSet)
```

- OnCalcFields (*TDataSet*)

```
OnCalcFields(TDataSet *DataSet)
```

Werden in einer Datenmenge berechnete Felder verwendet, dann wird einmal pro Datensatz, der angezeigt werden soll, das Ereignis *OnCalcFields* ausgelöst – unabhängig davon, ob der Datensatz nun tatsächlich angezeigt werden soll oder nicht. Dieses Ereignis eignet sich insbesondere dazu, um berechneten Feldern ihre Werte zuzuweisen.

```
void __fastcall TForm1::Table1CalcFields(TDataSet *DataSet)
{
  Table1->FieldByName("Gesamtpreis")->AsCurrency
```

```
    = Table1->FieldByName("Stückzahl")->AsCurrency
    * Table1->FieldByName("Einzelpreis")->AsCurrency;
}
```

- OnDeleteError (*TDataSet*)

```
DeleteError(TDataSet *DataSet, EDatabaseError *E,
  TDataAction &Action)
```

Das Ereignis *OnDeleteError* wird ausgelöst, wenn beim Löschen eines Datensatzes eine Exception auftritt. Mit Hilfe des Parameters *Action* kann der Benutzer vorgeben, wie auf den Fehler reagiert werden soll:

– *daFail:* Die Löschoperation wird abgebrochen und eine Fehlermeldung angezeigt (Voreinstellung).

– *daAbort:* Die Löschoperation wird ohne Ausgabe einer Fehlermeldung abgebrochen.

– *daRetry:* Die Löschoperation wird erneut durchgeführt. (Berichtigen Sie vorher den Fehler!)

- OnEditError (*TDataSet*)

```
OnEditError(TDataSet *DataSet, EDatabaseError *E,
  TDataAction &Action)
```

Das Ereignis *OnEditError* wird ausgelöst, wenn beim Ändern oder Einfügen eines Datensatzes eine Exception auftritt. Zum Parameter *Action* siehe *OnDeleteError*.

- OnFilterRecord (*TDataSet*)

```
OnFilterRecord(TDataSet *DataSet, bool &Accept)
```

Das Ereignis *OnFilterRecord* wird jedesmal dann ausgelöst, wenn eine andere Datenzeile in der Datenmenge zum aktuellen Datensatz wird und die Eigenschaft *Filtered* = *true* ist. Damit der jeweilige Datensatz akzeptiert wird, muß der Parameter *Accept* auf *true* gesetzt werden. Ein Beispiel dazu sehen Sie in Kapitel 2.4.3.

Beachten Sie mögliche Interaktionen zwischen der Ereignisbehandlungsroutine von *OnFilterRecord* und der Eigenschaft *Filter*.

- OnNewRecord (*TDataSet*)

```
OnNewRecord(TDataSet *DataSet)
```

Das Ereignis *OnNewRecord* wird beim Ein- oder Anfügen (*Insert* oder *Append*) eines neuen Datensatzes in der Datenmenge ausgelöst.

- OnPostError (*TDataSet*)

```
OnPostError(TDataSet *DataSet, EDatabaseError *E,
  TDataAction &Action)
```

Das Ereignis *OnPostError* wird ausgelöst, wenn beim Eintragen eines Datensatzes eine Exception auftritt. Dies kann beispielsweise bei einer Schlüsselverletzung der Fall sein. Zum Parameter *Action* siehe *OnDeleteError*.

- OnServerYield (*TDataSet*)

```
OnServerYield(TDataSet *DataSet, bool &AbortQuery)
```

Das Ereignis *OnServerYield* wird ausgelöst, wenn ein Remote-Datenbank-Server während der Bearbeitung einer Abfrage ein Callback an die BDE sendet. Mit Hilfe des Parameters kann die Abfrage dann gegebenenfalls abgebrochen werden.

- OnUpdateError (*TDataSet*)

```
OnUpdateError(TDataSet *DataSet, EDatabaseError *E,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
```

Das Ereignis *OnUpdateError* wird ausgelöst, wenn beim Schreiben von zwischengespeicherten Aktualisierungen (*Cached Updates*) in die Datenbank eine Exception auftritt. Mit Hilfe des Parameters *TUpdateAction* kann festgestellt werden, bei welcher Art von Aktion der Fehler aufgetreten ist.

- *ukModify:* Fehler beim Bearbeiten eines vorhandenen Datensatzes
- *ukInsert:* Fehler beim Ein- oder Anfügen eines neuen Datensatzes
- *ukDelete:* Fehler beim Löschen eines Datensatzes.

Der Parameter *UpdateAction* gibt die Aktion an, die durchgeführt wird, wenn die Ereignisbehandlungsroutine für *OnUpdateError* aufgerufen wird.

- *uaAbort:* Die Aktualisierung wird ohne Ausgabe einer Fehlermeldung abgebrochen.
- *uaFail:* Die Aktualisierung wird abgebrochen und eine Fehlermeldung angezeigt (Voreinstellung).
- *uaRetry:* Die Aktualisierung, die zu der Fehlerbedingung geführt hat, wird wiederholt – vorher sollte der Fehler behoben werden.
- *uaSkip:* Die Aktualisierung des jeweiligen Datensatzes wird übersprungen. Alle nicht eingetragenen Änderungen bleiben im Zwischenspeicher.

- OnUpdateRecord (*TDataSet*)

```
UpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
```

Das Ereignis *OnUpdateRecord* wird ausgelöst, wenn zwischengespeicherte Aktualisierungen in einen Datensatz geschrieben werden. Auf diese Weise können Aktualisierungen verarbeitet werden, die nicht von einer einzelnen Aktualisierungskomponente durchgeführt werden können. Dazu gehört beispielsweise die Implementierung von nachfolgenden Aktualisierungs-, Einfüge- und Löschoperationen in den entsprechenden Datenbanktabellen.

Die möglichen Zustände des Parameters *UpdateKind* sind bei *OnUpdateError* beschrieben. Die möglichen Zustände des Parameters *UpdateAction* sind bei *OnUpdateError* beschrieben, hinzu kommt:

– *uaApplied:* Die Aktualisierung wird durchgeführt und der betreffende Datensatz aus dem Zwischenspeicher freigegeben.

2.6 Die Komponente TDBDataSet

Die Komponente *TDBDataSet* implementiert diejenigen Eigenschaften und Ereignisse, die zur Anbindung einer Datenmenge an eine Datenbank erforderlich sind.

Im folgenden sollen diejenigen Eigenschaften und Ereignisse, die TDBDataSet von anderen Komponenten und Objekten – insbesondere von *TDataSet* – geerbt hat, nicht erwähnt werden.

Die Komponente *TDBDataSet* implementiert keine eigenen Ereignisse.

2.6.1 Die Eigenschaften von TDBDataSet

- Database (*TDBDataSet*, nur lesen)

  ```
  __property TDatabase* Database;
  ```

  Mit *Database* kann man auf die Eigenschaften, Ereignisse und Methoden der *TDatabase*-Komponente zugreifen, die mit der Datenmenge verknüpft ist. Diese wird entweder explizit vom Benutzer eingefügt, andernfalls wird sie automatisch von C++Builder erzeugt.

- DatabaseName (*TDBDataSet*, veröffentlicht)

  ```
  __property System::AnsiString DatabaseName;
  ```

  Die Eigenschaft *DatabaseName* gibt den Namen der Datenbank an, die dieser Datenmenge zugeordnet werden soll. Es kann hier folgendes vorgegeben werden:

- ein definierter BDE-Alias
- ein Verzeichnis für Desktop-Datenbankdateien
- ein Verzeichnis und einen Dateiname für eine *Local InterBase Server*-Datenbank
- ein anwendungsspezifischer Alias, der über eine Komponente *TDatabase* definiert wurde

- DBHandle (*TDBDataSet*, nur lesen)

```
__property Bde::hDBIDb DBHandle;
```

Gibt das Datenbank-Handle der Borland Database Engine an. Wird benötigt, wenn Funktionen der BDE-API direkt aufgerufen werden sollen.

- DBLocale (*TDBDataSet*, nur lesen)

```
__property void * DBLocale;
```

Die Eigenschaft *DBLocale* gibt den Sprachtreiber der Borland Database Engine für die Datenmenge an.

- DBSession (*TDBDataSet*, nur lesen)

```
__property System::AnsiString SessionName;
```

Die Eigenschaft *SessionName* gibt den Namen der Sitzung an, mit der die Datenmenge verknüpft ist. Solange der Eigenschaft *SessionName* nicht der Name einer *TSession*-Instanz zugewiesen wurde, hat *DBSession* den Wert der Standard-Sitzung *Session*.

- SessionName (*TDBDataSet*, veröffentlicht)

```
__property System::AnsiString SessionName;
```

Mit der Eigenschaft *SessionName* können Sie ermitteln beziehungsweise festlegen, mit welcher Sitzung die Datenmenge verknüpft ist.

- UpdateMode (*TDBDataSet*, veröffentlicht)

```
__property TUpdateMode UpdateMode;
```

Die Eigenschaft *UpdateMode* legt fest, wie Delphi Datensätze finden wird, welche auf einem Remote-Datenbank-Server aktualisiert werden sollen. Solche Datenbanken sind ja nicht satzorientiert, sondern mengenorientiert. Insbesondere bei Mehrbenutzerdatenbanken kann es vorkommen, daß ein Datensatz, der vom Benutzer A editiert wird, zwischenzeitlich vom Benutzer B verändert worden ist.

Nehmen wir an, der Datensatz *12345 (Index)* | *Karl* | *Müller* | *Am Ufer 3* | *10 319 Berlin* wird vom Benutzer *A* abgeändert in *12345* | *Karl* | *Müller-Maier* | *Am Ufer 3* | *10 319 Berlin*, dann wird folgende SQL-Anweisung zum Server geschickt:

```
UPDATE tabelle SET nummer = 12345, vorname = "Karl",
     nachname = "Müller-Maier", straße = "Am Ufer 3",
     ort = "10 319 Berlin"
   WHERE (nummer  = 12345) AND (vorname  = "Karl") AND
     (nachname = "Müller") AND (straße = "Am Ufer 3") AND
     (ort = "10 319 Berlin")
```

Zwischen der Abfrage der Daten und dem Update-Befehl hat nun Benutzer B die Umzugsmeldung von Herrn Müller(-Maier) bearbeitet und diesen in die *Parkaue 23, 10 387 Berlin* versetzt. Folge der ganzen Geschichte: Das RDMS gibt eine Fehlermeldung zurück, weil es den Datensatz nicht findet. Diese Verhaltensweise käme zustande, wenn die Eigenschaft *UpdateMode* auf *Where All* steht (dies ist die Voreinstellung).

Ändert der Programmierer die Eigenschaft auf *WhereKeyOnly*, dann wird lediglich nach dem Schlüssel gesucht:

```
UPDATE tabelle SET nummer = 12345, vorname = "Karl",
     nachname = "Müller-Maier", straße = "Am Ufer 3",
     ort = "10 319 Berlin"
   WHERE (nummer  = 12345)
```

Dies hat zur Folge, daß die Änderungen von Anwender B überschrieben werden. Eine dritte Variante ist der Wert *WhereChanged*. Dabei werden die Indexspalte und die geänderte Spalte für die Suche verwendet:

```
UPDATE tabelle SET nummer = 12345, vorname = "Karl",
     nachname = "Müller-Maier", straße = "Am Ufer 3",
     ort = "10 319 Berlin"
   WHERE (nummer  = 12345) AND (nachname = "Müller")
```

2.6.1 Die Methoden von TDBDataSet

- ~TDBDataSet (*TDBDataSet*)

```
__fastcall virtual ~TDBDataSet(void);
```

Der Destruktor *~TDBDataSet* beseitigt die Instanz dieser Komponente. Rufen Sie den Destruktor nicht direkt auf, sondern verwenden Sie die Methode *Free*.

- CheckOpen (*TDBDataSet*)

```
bool __fastcall CheckOpen(unsigned short Status);
```

  Mit *CheckOpen* können Sie ermitteln, ob die BDE bei einem Zugriffsversuch auf eine Tabelle einen Fehler zurückgibt. Status ist der Rückgabewert eines vorhergehenden BDE-Aufrufs. Wenn *CheckOpen* den Wert *false* zurückgibt, bedeutet dies, daß auf die Datenmenge nicht zugegriffen werden konnte.

- TDBDataSet (*TDBDataSet*)

```
__fastcall virtual TDBDataSet(Classes::TComponent* Aowner);
```

  Der Konstruktor *TDBDataSet* erzeugt eine neue Instanz dieser Komponente.

2.7 Die Komponente TTable

In der folgenden Referenz von *TTable* werden diejenigen Methoden ignoriert, welche von *TObject*, *TPersistent* und *TComponent* geerbt wurden.

2.7.1 Die veröffentlichten Eigenschaften von TTable

- Active (*TDataSet*)
- AutoCalcFields (*TDataSet*)
- CachedUpdates (*TDataSet*)
- DatabaseName (*TDBDataSet*)
- Exclusive (*TTable*)

```
__property bool Exclusive;
```

  Die Eigenschaft *Exclusive* ermöglicht es einer Anwendung, exklusiven Zugriff auf eine Tabelle einer Desktop-Datenbank zu erhalten. Ist die Eigenschaft *Exclusive* = *true*, dann kann die Tabelle von keiner anderen Anwendung geöffnet werden. Wird versucht, eine Tabelle exklusiv zu öffnen, die schon von einer anderen Anwendung verwendet wird, dann wird eine Exception ausgelöst.

  Werden bei einer Tabelle zu Entwurfszeit die Eigenschaften *Active* und *Exclusive* auf *true* gesetzt, dann kann diese Anwendung nicht gestartet werden.

- Filter (*TDataSet*)
- Filtered (*TDataSet*)

- FilterOptions (*TDataSet*)
- IndexFieldNames (*TTable*)

```
__property System::AnsiString IndexFieldNames;
```

Die Eigenschaft *IndexFieldNames* gibt die Spalten an, die als Index für eine Tabelle verwendet werden. Statt einen bereits bei der Definition der Metadaten erstellten Index zu verwenden, können die Spalten, die den Index bilden sollen, auch zu Entwurfs- oder Laufzeit bestimmt werden. Bei Desktop-Datenbanken müssen die verwendeten Spalten bereits indiziert sein, bei Client-Server-Systemen ist dies nicht erforderlich.

Die Eigenschaften *IndexFieldNames* und *IndexName* schließen sich gegenseitig aus. Das Setzen einer dieser Eigenschaften löscht die andere.

- IndexFiles (*TTable*)

```
__property Classes::TStrings* IndexFiles;
```

Die Eigenschaft *IndexFiles* gibt eine oder mehrere dBASE-Indexdateien an, die von einer dBASE-Tabelle mit nicht-gewarteten Indizes verwendet werden.

- IndexName (*TTable*)

```
__property System::AnsiString IndexName;
```

Wählt einen Sekundärindex zur Sortierung der Daten aus. Wird der Eigenschaft ein leerer String zugewiesen (Voreinstellung), dann wird der Primärindex verwendet.

Die Eigenschaften *IndexFieldNames* und *IndexName* schließen sich gegenseitig aus. Das Setzen einer dieser Eigenschaften löscht die andere.

- MasterFields (*TTable*)

```
__property System::AnsiString MasterFields;
```

Die Eigenschaft *MasterFields* wird verwendet, um eine Master-Detail-Verknüpfung zwischen zwei Tabellen zu erstellen. Der Eigenschaft *MasterFields* werden der oder die Spaltennamen der Hauptabelle zugewiesen, über welche die Verknüpfung zur Detailtabelle hergestellt werden. Werden mehrere Spaltennamen angegeben, dann werden diese mit Semikola getrennt. Näheres zum Thema Master-Detail-Verknüpfung in Kapitel 2.2.

- MasterSource (*TTable*)

```
__property Db::TDataSource* MasterSource;
```

Mit der Komponenteneigenschaft *MasterSource* wird die Datenquelle für die Master-Detail-Verknüpfung angegeben. Näheres zum Thema Master-Detail-Verknüpfung in Kapitel 2.2.

- ReadOnly (*TTable*)

  ```
  __property bool ReadOnly;
  ```

  Wird die Eigenschaft *ReadOnly* auf *true* gesetzt, dann kann die Tabelle von dieser *TTable*-Komponente nicht geändert werden.

- SessionName (*TDBDataSet*)
- TableName (*TTable*)

  ```
  __property System::AnsiString TableName;
  ```

  Gibt den Tabellennamen der verwendeten Tabelle an. Wenn Die Eigenschaft *DatabaseName* bereits gesetzt ist, kann zur Entwurfszeit ein gültiger Tabellenname in der *TableName*-Dropdown-Liste im Objektinspektor gewählt werden. Damit *TableName* ein Wert zugewiesen werden kann, muß die Eigenschaft *Active = false* sein.

- TableType (*TTable*)

  ```
  __property TTableType TableType;
  ```

  Mit *TableType* kann festgelegt werden, ob eine *dBASE*-, *Paradox*- oder *ASCII*-Tabelle verwendet wird. *TableType* kann jeder der folgenden Werte zugewiesen werden:t

  – *tDefault*: Der Tabellentyp wird gemäß der Dateierweiterung bestimmt (Vorgabe).

  – *ttParadox*: Paradox-Tabelle.

  – *ttDBase*: dBASE-Tabelle.

  – *ttASCII*: Die Tabelle ist eine Comma-Delimited-Textdatei mit Strings in Anführungszeichen für jedes Feld.

  TableType wird für Tabellen auf Remote-SQL-Servern nicht verwendet.

- UpdateMode (*TDBDataSet*)
- UpdateObject (*TDataSet*)

2.7.2 Die öffentlichen Eigenschaften von TTable

- BOF (*TDataSet*, nur Lesen)
- Bookmark (*TDataSet*)
- CanModify (*TDataSet*, nur Lesen)
- Database (*TDataSet*, nur Lesen)

- DataSource (*TDataSet*, nur Lesen)
- DBHandle (*TDBDataSet*, nur Lesen)
- DBLocale (*TDBDataSet*, nur Lesen)
- DBSession (*TDBDataSet*, nur Lesen)
- DefaultFields (*TDataSet*, nur Lesen)
- Designer (*TDataSet*, nur Lesen)
- EOF (*TDataSet*, nur Lesen)
- ExpIndex (*TDataSet*, nur Lesen)
- FieldCount (*TDataSet*, nur Lesen)
- FieldDefs (*TDataSet*)
- Fields (*TDataSet*)
- FieldValues (*TDataSet*)
- Found (*TDataSet*, nur Lesen)
- Handle (*TDataSet*, nur Lesen)
- IndexDefs (*TTable*)

```
__property TIndexDefs* IndexDefs;
```

Die Array-Eigenschaft *IndexDefs* enthält eine Liste aller für die jeweilige Tabelle verfügbaren Indizes. Diese Eigenschaft wird beispielsweise dann benötigt, wenn zur Laufzeit eine Tabelle erstellt wird, welche Indizes enthalten soll. Ein Beispiel dazu siehe Kapitel 2.3.

Die einzelnen Listenelemente sind Klassen vom Typ *TIndexDef* (ohne *s* am Ende) und implementieren unter anderem folgende Eigenschaften

– *Fields:* Der *AnsiString* beinhaltet die Namen aller Spalten, welche den Index bilden. Mehrere Spalten werden durch Semikola getrennt.

– *Name:* Der *AnsiString* benennt den Index. Bei Paradox-Tabellen erhält der Primärschlüssel einen leeren String.

– *Options:* Die Mengeneigenschaft faßt weitere Informationen über den Index zusammen (*ixPrimary*, *ixUnique*, *ixDescending*, *ixExpression*, *ixCaseInsensitiv*)

Das folgende Beispiel definiert einen absteigenden, eindeutigen Sekundärindex namens *ZweitIndex* für das Feld *Telefon*. Ein weiteres Beispiel ist in Kapitel 2.3 zu finden.

```
Table1->IndexDefs->Clear();
IndexOptions << ixDescending << ixUnique;
Table1->IndexDefs->Add("ZweitIndex", "Telefon", IndexOptions);
```

- IndexFieldCount (*TTable*, nur Lesen)

  ```
  __property int IndexFieldCount;
  ```

  Gibt die Anzahl der Felder an, die den aktuellen Schlüssel bilden.

- IndexFields (TTable)

  ```
  __property Db::TField* IndexFields[int Index];
  ```

  Die Array-Eigenschaft *IndexFields* beinhaltet diejenigen Felder, die den aktuellen Index bilden.

- KeyExclusive (TTable)

  ```
  __property bool KeyExclusive;
  ```

  Die Eigenschaft *KeyExclusive* bestimmt, ob der jeweilige Grenzwert noch zum Bereich gehört oder nicht. Ein Beispiel dazu finden Sie in Kapitel 2.4.1.

- KeyFieldCount (TTable)

  ```
  __property int KeyFieldCount;
  ```

  Die Eigenschaft *KeyFieldCount* bezeichnet die Anzahl der Felder, die zur Suche von Teilübereinstimmungen verwendet werden.

- KeySize (*TDataSet*, nur Lesen)
- Locale (*TDataSet*, nur Lesen)
- Modified (*TDataSet*, nur Lesen)
- RecordNo (*TDataSet*, nur Lesen)
- RecordCount (*TDataSet*, nur Lesen)
- RecordSize (*TDataSet*, nur Lesen)
- State (*TDataSet*, nur Lesen)
- TableLevel (*TTable*)

  Die Eigenschaft *TableLevel* ist in der Online-Hilfe erwähnt, jedoch nicht in der Header-Datei aufgeführt.

- UpdateRecordTypes (*TDataSet*)
- UpdatesPending (*TDataSet*, nur Lesen)

2.7.3 Die Methoden von TTable

- ~TTTable (*TTable*)

  ```
  __fastcall virtual ~TTable(void);
  ```

  Der Destruktor *~TTable* gibt den Speicher frei, der für die *TTable*-Instanz reserviert gewesen ist. *~TTable* sollte nicht direkt aufgerufen werden, statt dessen ist die Methode *Free* zu verwenden.

- ActiveBuffer (*TDataSet*)
- AddIndex (*TTable*)

  ```
  void __fastcall AddIndex(const System::AnsiString Name,
    const System::AnsiString Fields, TIndexOptions Options);
  ```

  Mit *AddIndex* kann ein neuer Index für die mit der *TTable*-Instanz verbundenen Tabelle erzeugt werden. *Name* bezeichnet den Namen des neuen Indexes. *Fields* ist eine Liste der Felder, die in den Index aufgenommen werden sollen, die einzelnen Feldnamen werden dabei durch Semikola getrennt. Zu den möglichen Werten der Mengenvariable *Options* siehe die Eigenschaft *IndexDefs*.

- Append (*TDataSet*)
- ApplyRange (*TTable*)

  ```
  void __fastcall ApplyRange(void);
  ```

  Die Methode *ApplyRange* wendet einen Bereich auf die Tabelle an. Ein Beispiel dazu ist in Kapitel 2.4.1 zu finden.

- AppendRecord (*TDataSet*)
- ApplyUpdates (*TDataSet*)
- BatchMove (*TTable*)

  ```
  long __fastcall BatchMove(Db::TDataSet* ASource,
    TBatchMode AMode);
  ```

  Die Methode *BatchMove* dient zum Ausführen einer Stapeloperation. Der Parameter *ASource* gibt die Datenquelle-Komponente der Quelltabelle (oder der Quellabfrage) an. Für den Parameter *BatchMode* (*TBatchMode*) gibt es folgende Möglichkeiten. (Im folgenden wird die im Parameter übergebene Tabelle Quelltabelle genannt, sie wird nicht geändert; geändert wird nur die Zieltabelle.)

  – *batCopy* legt eine exakte Kopie der im Parameter angegebenen Tabelle an.

  – *batDelete* löscht alle Datensätze, welche in der Quelltabelle vorhanden sind aus der Zieltabelle; dazu muß der verwendete Index übereinstimmen.

- *batAppend* hängt alle Datensätze der Quelltabelle an die Zieltabelle an.
- *batUpdate* ändert alle in der Quelltabelle vorkommenden Datensätze auch in der Zieltabelle; die Auswahl erfolgt über den Index. (Beispielsweise können so in der Filiale durchgeführte Adressenänderungen auf den Hauptrechner gespielt werden.)
- *batAppendUpdate* kombiniert *batUpdate* und *batAppend*. Datensätze, welche (über den Index gesucht) schon in der Zieltabelle vorhanden sind, werden geändert, alle anderen angehängt.

Stapeloperationen können auch mit der Komponente *TBatchMove* vorgenommen werden.

- Cancel (*TDataSet*)
- CancelRange (*TTable*)

```
void __fastcall CancelRange(void);
```

Die Methode *CancelRange* löscht die momentan aktiven Bereiche. Näheres zum Thema *Bereiche* ist in Kapitel 2.4.1 zu finden.

- CancelUpdates (*TDataSet*)
- CheckBrowseMode (*TDataSet*)
- CheckOpen (*TDBDataSet*)
- ClearFields (*TDataSet*)
- Close (*TDataSet*)
- CloseIndexFile (*TTable*)

```
void __fastcall CloseIndexFile
  (const System::AnsiString IndexFileName);
```

Die Methode *CloseIndexFile* schließt die angegebene dBASE-Indexdatei mit einem nicht gewarteten Index.

- CommitUpdates (*TDataSet*)
- ControlsDisabled (*TDataSet*)
- CreateTable (*TTable*)

```
void __fastcall CreateTable(void);
```

Mit der Methode *CreateTable* wird eine neue Tabelle erstellt. Ein Beispiel dazu ist in Kapitel 2.3 zu finden.

- CursorPosChanged (*TDataSet*)
- Delete (*TDataSet*)
- DeleteIndex (*TTable*)

```
void __fastcall DeleteIndex(const System::AnsiString Name);
```

Die Methode *DeleteIndex* löscht den angegebenen Sekundärindex der Tabelle.

- DeleteTable (*TTable*)

```
void __fastcall DeleteTable(void);
```

Mit *DeleteTable* wird die in der Eigenschaft *TableName* angegebene Tabelle gelöscht. Alle Daten und Metadaten gehen dabei verloren. Mit dieser Methode sollte daher vorsichtig verfahren werden.

Vor dem Einsatz der Methode *DeleteTable* muß die Tabelle geschlossen werden. Ein Beispiel zu dieser Methode ist in Kapitel 2.3 zu finden.

- DisableControls (*TDataSet*)
- Edit (*TDataSet*)
- EditKey (*TTable*)

```
void __fastcall EditKey(void);
```

Zum Setzen und Ändern von Suchschlüsseln wird für gewöhnlich die Methode *SetKey* verwendet. Setzt sich der Schlüssel jedoch aus mehreren Spalten zusammen, von denen nur ein einzelner Wert geändert werden soll, dann kann dies mit *EditKey* bewerkstelligt werden.

- EditRangeEnd (*TTable*)

```
void __fastcall EditRangeEnd(void);
```

Zum Setzen und Ändern der oberen Bereichsgrenze wird für gewöhnlich die Methode *SetRangeEnd* verwendet. Setzt sich der Schlüssel jedoch aus mehreren Spalten zusammen, von denen nur ein einzelner Wert geändert werden soll, dann kann dies mit *EditRangeEnd* geschehen.

- EditRangeStart (*TTable*)

```
void __fastcall EditRangeStart(void);
```

Zum Setzen und Ändern der unteren Bereichsgrenze wird für gewöhnlich die Methode *SetRangeStart* verwendet. Setzt sich der Schlüssel jedoch aus mehreren Spalten zusammen, von denen nur ein einzelner Wert geändert werden soll, dann kann dies mit *EditRangeStart* realisiert werden.

- EmptyTable (*TTable*)

```
void __fastcall EmptyTable(void);
```

  Mit *EmptyTable* werden alle Datensätze aus einer Tabelle gelöscht. Im Gegensatz zu *DeleteTable* bleiben jedoch die Metadaten erhalten. Auch diese Methode sollte sehr vorsichtig eingesetzt werden.

- EnbleControls (*TDataSet*)
- FetchAll (*TDataSet*)
- FieldByName (*TDataSet*)
- FindField (*TDataSet*)
- FindFirst (*TDataSet*)
- FindKey (*TTable*)

```
bool __fastcall FindKey(const System::TVarRec *KeyValues,
  const int KeyValues_Size);
```

  Die Methode *FindKey* sucht einen Datensatz mit dem entsprechenden Wert im aktuellen Index. Bei Erfolg wird der Datenzeiger auf diesen Datensatz gesetzt und der Wert *true* zurückgegeben.

  Bei Desktop-Datenbanken muß der Schlüssel immer ein Index sein, der in der Eigenschaft *IndexName* angegeben wird. Wenn für *IndexName* kein Wert angegeben wurde, wird der Primärindex der Tabelle verwendet.

  Bei Cient-Server-Systemen kann der Schlüssel einem in *IndexName* angegebenen Index oder einer Liste von Feldnamen in der Eigenschaft *IndexFieldNames* entsprechen.

  Im folgenden Beispiel wird nach einem Datensatz gesucht, dessen Wert im Primärschlüssel vom Benutzer eingegeben wird.

```
void __fastcall TForm1::Schlsselwert1Click(TObject *Sender)
{
  AnsiString s = InputBox
    ("Datensatz suchen", "nach Nummer",  5000");
  Table1->FindKey(OPENARRAY(TVarRec, (s)));
}
```

- FindLast (*TDataSet*)

- FindNearest (*TTable*)

```
void __fastcall FindNearest(const System::TVarRec *KeyValues,
  const int KeyValues_Size);
```

Sucht – ebenso wie *FindKey* – nach einem Datensatz mit dem angegebenen Wert im aktuellen Index. Während jedoch *FindKey* bei erfolgloser Suche den Wert *false* zurückgibt und den Datenzeiger nicht verändert, setzt die Methode *FindNearest* den Datenzeiger stets auf den Datensatz mit der größten Übereinstimmung.

- FindNext (*TDataSet*)
- FindPrior (*TDataSet*)
- First (*TDataSet*)
- FreeBookmark (*TDataSet*)
- GetBookmark (*TDataSet*)
- GetCurrentRecord (*TDataSet*)
- GetFieldList (*TDataSet*)
- GetFieldNames (*TDataSet*)
- GetIndexNames (*TTable*)

```
void __fastcall GetIndexNames(Classes::TStrings* List);
```

Die Methode *GetIndexNames* ermittelt die Namen aller verfügbaren Indizes und speichert sie in der im Parameter angegebenen Liste.

```
Table1->GetIndexNames(ComboBox1->Items);
```

- GotoBookmark (*TDataSet*)
- GotoCurrent (*TTable*)

```
void __fastcall GotoCurrent(TTable* Table);
```

Die Methode *GotoCurrent* setzt den Datenzeiger auf die gleiche Position wie den Datenzeiger der als Parameter übergebenen Komponente *Table*. Damit der Datenzeiger von *Table2* stets auf derselben Position steht wie der Datenzeiger von *Table1*, würde man folgende Anweisung formulieren:

```
void __fastcall TForm1::DataSource1DataChange
  (TObject *Sender, TField *Field)
{
  if((Table1->State == dsBrowse)*(Table2->State == dsBrowse))
  Table2->GotoCurrent(Table1);
}
```

- GotoKey (*TTable*)

```
bool __fastcall GotoKey(void);
```

Mit *GotoKey* wird der Datensatz gesucht, dessen Werte im aktuellen Schlüssel denjenigen Werten entsprechen, die mit *SetKey* beziehungsweise *EditKey* vorgegeben wurden.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Table1->SetKey();
  Table1->FieldByName("Nummer")->AsInteger = 5000;
  Table1->GotoKey();
}
```

- GotoNearest (*TTable*)

```
void __fastcall GotoNearest(void);
```

Sucht – ebenso wie *GotoKey* – nach einem Datensatz im aktuellen Index. Während jedoch *GotoKey* bei erfolgloser Suche der Wert *false* zurückgibt und den Datenzeiger nicht verändert, setzt die Methode *GotoNearest* den Datenzeiger stets auf den Datensatz mit der größten Übereinstimmung.

- Insert (*TDataSet*)
- InsertRecord (*TDataSet*)
- IsLinkedTo (*TDataSet*)
- Last (*TDataSet*)
- Locate (*TDataSet*)
- LockTable (*TTable*)

```
void __fastcall LockTable(TLockType LockType);
```

Die Methode *LockTable* sperrt die betreffende Tabelle. Dies ist jedoch nur bei Desktop-Datenbanken möglich. Der Parameter *LockType* kann folgende Werte annehmen:

– *ltReadLock:* Die Tabelle kann von anderen Prozessen nicht gelesen werden.

– *ltWriteLock:* In die Tabelle kann von anderen Prozessen nicht geschrieben werden.

Sollen beide Sperren gemeinsam gesetzt werden, dann ist *LockTable* zweimal (mit jeweils anderem Parameter) aufzurufen.

- Lookup (*TDataSet*)

- MoveBy (*TDataSet*)
- Next (*TDataSet*)
- Open (*TDataSet*)
- OpenIndexFile (*TTable*)

```
void __fastcall OpenIndexFile(const System::AnsiString IndexName);
```

Die Methode *OpenIndexFile* öffnet die angegebene dBASE-Indexdatei mit einem nicht gewarteten Index.

- Post (*TDataSet*)
- Prior (*TDataSet*)
- Refresh (*TDataSet*)
- RenameTable (*TTable*)

```
void __fastcall RenameTable
  (const System::AnsiString NewTableName);
```

Die Methode *RenameTable* benennt die zugrundeliegende Tabelle um; sie funktioniert nur bei Desktop-Datenbanken.

- Resync (*TDataSet*)
- RevertRecord (*TDataSet*)
- SetFields (*TDataSet*)
- SetKey (*TTable*)

```
void __fastcall SetKey(void);
```

Mit *SetKey* werden Schlüsselwerte zum Suchen in einer Tabelle gesetzt.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Table1->SetKey();
  Table1->FieldByName("Nummer")->AsInteger = 5000;
  Table1->GotoKey();
}
```

- SetRange (*TTable*)

```
void __fastcall SetRange
  (const System::TVarRec *StartValues, const int StartValues_Size,
  const System::TVarRec *EndValues, const int EndValues_Size);
```

Die Methode *SetRange* erlaubt es, einen Bereich mit einer einzelnen Anweisung zu setzen. Das folgende Beispiel setzt einen Bereich zwischen 100 und 200 (für das Primärschlüsselfeld *Nummer*).

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Table1->SetRange(OPENARRAY (TVarRec, (100)),
    OPENARRAY (TVarRec, (200)));
}
```

- SetRangeEnd (*TTable*)

  ```
  void __fastcall SetRangeEnd(void);
  ```

  Die Methode *SetRangeEnd* startet das Setzen der oberen Bereichsgrenze. Ein Beispiel dazu ist in Kapitel 2.4.1 zu finden.

- SetRangeStart (*TTable*)

  ```
  void __fastcall SetRangeStart(void);
  ```

  Die Methode *SetRangeStart* startet das Setzen der unteren Bereichsgrenze. Ein Beispiel dazu ist in Kapitel 2.4.1 zu finden.

- TTable (*TTable*)

  ```
  __fastcall virtual TTable(Classes::TComponent* AOwner);
  ```

  Der Konstruktor *TTable* erzeugt eine neue Instanz von *TTable.*

- UnlockTable (*TTable*)

  ```
  void __fastcall UnlockTable(TLockType LockType);
  ```

  Die Methode *UnlockTable* entfernt eine mit *LockTable* gesetzte Sperre der Tabelle. Die möglichen Werte des Parameters *LockType* sind bei der Methode *LockTable* aufgeführt.

- UpdateCursorPos (*TDataSet*)
- UpdateRecord (*TDataSet*)
- UpdateStatus (*TDataSet*)

2.7.4 Die Ereignisse von TTable

- AfterCancel (*TDataSet*)
- AfterClose (*TDataSet*)
- AfterDelete (*TDataSet*)

- AfterEdit (*TDataSet*)
- AfterInsert (*TDataSet*)
- AfterOpen (*TDataSet*)
- AfterPost (*TDataSet*)
- AfterScroll (*TDataSet*)
- BeforeCancel (*TDataSet*)
- BeforeClose (*TDataSet*)
- BeforeDelete (*TDataSet*)
- BeforeEdit (*TDataSet*)
- BeforeInsert (*TDataSet*)
- BeforeOpen (*TDataSet*)
- BeforePost (*TDataSet*)
- BeforeScroll (*TDataSet*)
- OnCalcFields (*TDataSet*)
- OnDeleteError (*TDataSet*)
- OnEditError (*TDataSet*)
- OnFilterRecord (*TDataSet*)
- OnNewRecord (*TDataSet*)
- OnPostError (*TDataSet*)
- OnServerYield (*TDataSet*)
- OnUpdateError (*TDataSet*)
- OnUpdateRecord (*TDataSet*)

2.8 Die Komponente TDataSource

Relativ unspektakulär ist die Komponente *TDataSource*, welche die *TDataSet*-Komponenten *TTable*, *TQuery* und *TStoredProcedure* mit den entsprechenden datensensitiven Dialogelementen verbindet.

TDataSource ist direkt von *TComponent* abgeleitet. Im folgenden sollen jedoch nur diejenigen Eigenschaften, Methoden und Ereignisse erwähnt werden, die direkt von *TDataSource* implementiert werden.

2.8.1 Die Eigenschaften von TDataSource

- AutoEdit (*TDataSource*, veröffentlicht)

```
__property bool AutoEdit;
```

  Ist die Eigenschaft *AutoEdit* gleich *true* (Voreinstellung), wechselt die Datenquelle und die damit verbundene Datenmenge immer dann automatisch in den Modus *dsEdit*, wenn der Anwender versucht, in einem datensensitiven Dialogelement eine Änderung durchzuführen. Ist die Eigenschaft *AutoEdit* gleich *false*, dann muß vorher die Methode *Edit* aufgerufen werden.

- DataSet (*TDataSource*, veröffentlicht)

```
__property TDataSet* DataSet;
```

  Mit Hilfe der Komponenten-Eigenschaft *DataSet* wird festgelegt, mit welcher *TDataSet*-Komponente die Datenquelle verbunden ist. Diese Eigenschaft kann auch zur Laufzeit verändert werden.

- Enabled (*TDataSource*, veröffentlicht)

```
__property bool Enabled;
```

  Die Eigenschaft *Enabled* legt fest, ob der jeweilige Datensatz in den Dialogelementen angezeigt wird. Für gewöhnlich ist *Enabled* gleich *true*, bei der inkrementalen Suche kann es den Suchvorgang aber sehr beschleunigen, wenn man vorübergehend die Dialogelemente abkoppelt. Ein Beispiel dazu ist in Kapitel 2.4.2 zu finden.

- State (*TDataSource*, nur Lesen)

```
__property TDataSetState State;
```

  Die Eigenschaft *State* gleicht der gleichnamigen Eigenschaft der verbundenen Datenmenge, es sei denn, die Eigenschaft *Enabled* ist gleich *false* – dann ist *State* gleich *dsInactive*. Zu den Werten, die *State* annehmen kann, siehe Kapitel 2.5.2.

2.8.2 Die Methoden von TDataSource

- ~TDataSource (*TDataSource*)

```
__fastcall virtual ~TDataSource(void);
```

  Der Destruktor *~TDataSource* gibt den Speicher frei, der für die *TDataSource*-Instanz reserviert gewesen ist. *~TDataSource* sollte nicht direkt aufgerufen werden, statt dessen ist die Methode *Free* zu verwenden.

- Edit (*TDataSource*)

```
void __fastcall Edit(void);
```

  Die Methode *Edit* ruft die gleichnamige Methode der mit der *TDataSource*-Instanz verbundenen Datenmenge auf.

- IsLinkedTo (*TDataSource*)

```
bool __fastcall IsLinkedTo(TDataSet* DataSet);
```

  Die Methode *IsLinkedTo* prüft, ob die *TDataSource*-Instanz mit einer bestimmten Datenmenge verbunden ist.

- TDataSource (*TDataSource*)

```
__fastcall virtual TDataSource(Classes::TComponent* AOwner);
```

  Der Konstruktor *TDataSource* erzeugt eine Instanz dieser Komponente.

2.8.3 Die Ereignisse von TDataSource

Etwas spannender wird es dann bei den drei Ereignissen von *TDataSource*. Auch hier sollen wieder nicht die Deklarationen, sondern die damit erstellten Parameterlisten abgedruckt werden.

- OnDataChange (*TDataSource*)

```
OnDataChange(TObject *Sender, TField *Field)
```

  Das Ereignis *OnDataChange* wird immer dann ausgelöst, wenn sich die anzuzeigenden Daten (im weitesten Sinne) ändern; in der Regel wird dies der Fall sein, wenn ein anderer Datensatz aufgerufen wird, also auch, wenn sich die Eigenschaft *State* von *dsInactive* auf *dsBrowse* ändert.

  Das Ereignis eignet sich also dazu, zwei *DataSet*-Komponenten zu synchronisieren. Hier im Beispiel ist *DataSource2* mit *TQuery1* und einem *DBGrid* verbunden. Immer wenn in diesem *DBGrid* auf einen anderen Datensatz gewechselt wird, soll dieser in der aus *TEdit*-Komponenten zusammengesetzten Eingabemaske angezeigt werden:

```
void __fastcall TForm1::DataSource1DataChange
   (TObject *Sender, TField *Field)
{
  if((Query1->State == dsBrowse)*(Table1->State == dsBrowse))
  Table1->FindKey(OPENARRAY(TVarRec,
     (Query1->FieldByName("Nummer")->AsInteger)));
}
```

Da es sich hier um eine *TQuery*- und nicht um eine *TTable*-Komponente handelt, kann die *TTable*-Methode *GotoCurrent* nicht verwendet werden.

- OnStateChange (*TDataSource*)

```
OnStateChange(TObject *Sender)
```

Das Ereignis *OnStateChange* tritt auf, wenn sich die Eigenschaft *State* ändert. Dieses Ereignis wird beispielsweise dann gebraucht, wenn eine Datenmenge nicht mit dem *DBNavigator*, sondern über Menüpunkte, Bit- oder Speed-Buttons gesteuert wird. Soll sich ein Programm ausschließlich mit Hilfe der Tastatur bedienen lassen, werden Sie bei der Realisierung dieses Vorhabens mit dem *DBNavigator* schnell in Schwierigkeiten geraten – hier müssen Sie dann die entsprechenden *DataSet*-Methoden über Buttons oder Menüpunkte aufrufen.

Sollen – analog zum *DBNavigator* – nur die Menü-Punkte oder Bit-Buttons die Eigenschaft *Enabled = true* erhalten, deren Verwendung auch Sinn macht (im Zustand *dsBrowse* benötigen Sie beispielsweise weder *Post* noch *Cancel*), dann können Sie mit dem Ereignis *OnStateChange* eine Prozedur auslösen, welche die Eigenschaft *State* abfragt und die *Enabled*-Eigenschaft dementsprechend setzt.

- OnUpdateData (*TDataSource*)

```
OnUpdateData(TObject *Sender)
```

Das Ereignis *OnUpdateDate* wird ausgelöst, wenn bei der verbundenen *TDataSet*-Komponente die Methode *Post* (direkt oder indirekt) oder die Methode *UpdateData* aufgerufen wird.

2.9 Die Komponente TField

Die Komponente *TField* werden Sie nicht in der Komponentenpalette finden, werden entsprechende Instanzen doch automatisch von C++Builder erzeugt.

- Standardmäßig wird für jede Spalte einer Datenmenge (*TTable*, *TQuery* oder *TStoredProc*) automatisch eine *TField*-Instanz erzeugt.
- Mit Hilfe des Feldeditors lassen sich statische *TField*-Instanzen erstellen. Diese sind insbesondere für berechnete Felder erforderlich. Sind für eine Datenmenge statische *TField*-Instanzen erstellt worden, dann werden keine weiteren *TField*-Instanzen automatisch erzeugt. Es ist somit erforderlich, daß für jede Spalte, die in der Datenmenge enthalten sein soll, eine statische *TField*-Instanz erzeugt wird.

Mit Hilfe statischer *TField*-Instanzen lassen sich einzelne Spalten gezielt aus einer Datenmenge ausschließen. Soll jedoch lediglich die Anzeige von Spalten in einer *TDBGrid*-Komponente verhindert werden, dann läßt sich dies einfacher mit dem Spalteneditor von *TDBGrid* erreichen.

2.9.1 Erzeugen statischer TField-Instanzen

Um statische Feldkomponenten zu erzeugen, wird ein Doppelklick auf die Datenmengenkomponente ausgeführt. C++Builder öffnet dann den Feldeditor (siehe Bild 2.8 rechts). Hier sind zunächst keine Einträge zu sehen: Solange nichts anderes definiert wird, erstellt C++Builder automatisch entsprechende Feldkomponenten.

Nun wird aus dem Kontextmenü (rechte Maustaste) der Menüpunkt FELDER HINZUFÜGEN gewählt. Den auf diese Weise geöffneten Dialog kann man in der Regel mit OK bestätigen – überflüssige Feldkomponenten, das heißt, Felder, die nicht verwendet werden sollen, lassen sich später immer noch mit der ENTF-Taste ausschließen.

Für jede Spalte der Datenmenge wird nun eine Feldkomponente erstellt, welche sich mit einem Mausklick auf den jeweiligen Eintrag auswählen läßt. Die dazu-

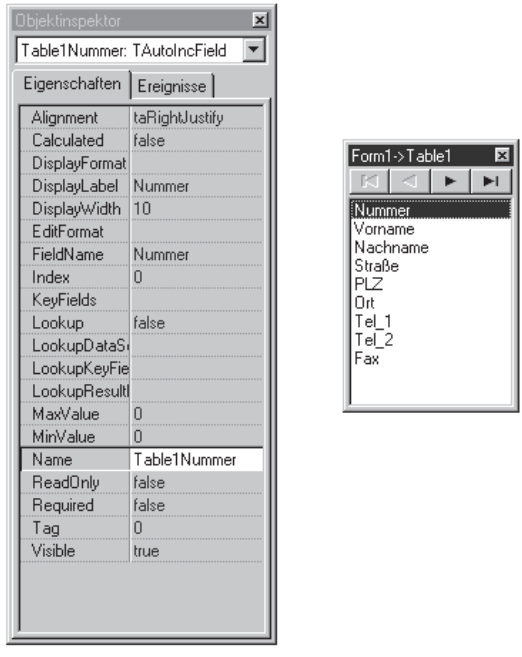


Bild 2.8: Eine statische TField-Komponente im Objektinspektor

gehörenden Eigenschaften lassen sich dann im Objektinspektor bearbeiten (Bild 2.8 rechts).

Mit Hilfe der Browse-Buttons im Feldeditor kann man schon zur Entwurfszeit in der Datenmenge navigieren. Dies ist insbesondere dann hilfreich, wenn andere Datensätze als der erste Datensatz besonders lange Einträge in einzelnen Spalten aufweisen. Das Navigieren in der Datenmenge vereinfacht hier das Design ausreichend bemessener Formulare.

Berechnete Felder

Um Redundanzen oder gar widersprüchliche Informationen innerhalb eines Datensatzes zu vermeiden, werden Spalten, deren Inhalte sich durch andere Spalten berechnen lassen würden, nicht in die Tabelle aufgenommen. So würde man beispielsweise bei einer Rechnungspostentabelle eine Spalte *Gesamtpreis* nicht vorsehen, weil sich der Inhalt einer solchen Spalte stets durch eine Multiplikation der Felder *Stückzahl* und *Einzelpreis* berechnen ließe.

Nun sollte man aber dem Anwender nicht zumuten, sich diese Werte dann mit Hilfe der eigenen Rechenkünste oder eines Taschenrechners zu beschaffen: Um einer Datenmenge Felder hinzuzufügen, welche in der zugrundeliegenden Tabelle oder Abfrage nicht vorhanden sind, werden *berechnete Felder* verwendet.

Um ein solches berechnetes Feld zu erstellen, wählt man im Kontextmenü des Feldeditors den Eintrag NEUES FELD und öffnet auf diese Weise einen Dialog nach Bild 2.9. Aus dem einzugebendes Spaltennamen – im Beispiel *Test* – wird automatisch ein geeignetes – sprich eindeutiger – Komponentenname generiert. Des weiteren muß angegeben werden, welcher Datentyp dem berechneten Feld zugrunde liegen soll – hier im Beispiel soll ein String-Feld erzeugt werden.

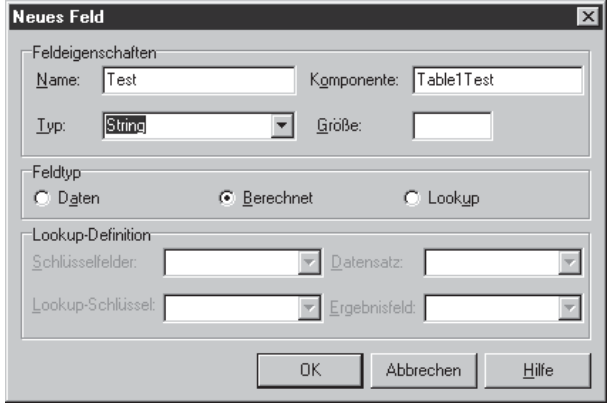


Bild 2.9: Erzeugen eines berechneten Feldes

Informationen darüber, auf welche Weise denn das Feld berechnet werden soll, können nicht eingegeben werden – eine entsprechende Anweisung obliegt einzig und allein der Ereignisbehandlungsroutine des *TDataSet*-Ereignisses *OnCalcFields*.

```
void __fastcall TForm1::Table1CalcFields(TDataSet *DataSet)
{
  Table1->FieldByName("Test")->AsString =
    Table1->FieldByName("Vorname")->AsString + " "
    + Table1->FieldByName("Nachname")->AsString;
}
```

Hier im Beispiel wird dem Inhalt des berechneten Feldes eine Zusammenfügung aus Vor- und Nachname zugewiesen. Dies ist beispielsweise dann sinnvoll, wenn Adreßetiketten gedruckt werden sollen. Bei der Verwendung der Komponente *TQuery* kann man – solange die Standard-SQL-Operatoren dafür ausreichen – auch per SQL-Anweisung eine berechnete Spalte erstellen:

```
SELECT vorname, nachname,
    vorname || " " || nachname AS test
    ...
  FROM ...
```

Lookup-Felder

Eine Abwandlung der berechneten Felder sind die Lookup-Felder. Hier wird der Spaltenwert nicht per *OnCalcFields*-Ereignisbehandlungsroutine zugewiesen, sondern aus einer anderen Tabelle bezogen. Dazu gleich ein Beispiel:

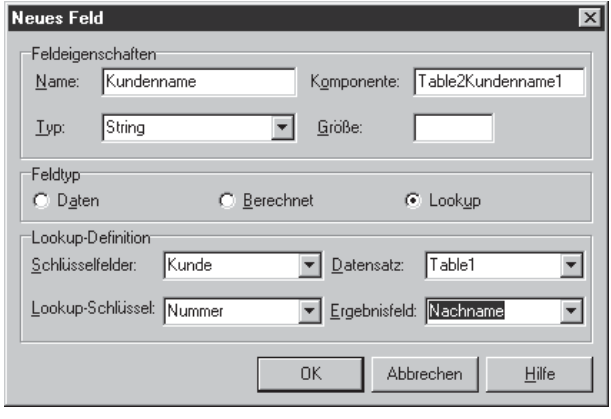


Bild 2.10: Definition eines Lookup-Feldes

Die Tabelle *Offenpo.DB* beinhaltet die Außenstände einer Firma. In dieser Tabelle werden jedoch nicht die Kundennamen oder gar die Kundenandressen gespeichert, sondern lediglich die Kundennummern. Wird nun diese Tabelle in einem *DBGrid* angezeigt, dann erscheint die für den Anwender weniger aussagekräftige Kundennummer. Mit Hilfe eines Lookup-Feldes könnte man nun auch den Kundennamen in die Datenmenge mit aufnehmen.

Wie bei der Definition eines berechneten Feldes müssen auch hier Spaltenname und Spaltentyp eingegeben werden, des weiteren wird der Radiobutton *Lookup* gewählt. Die Definition der Lookup-Beziehung erfolgt über das *Schlüsselfeld* und den *Lookup-Schlüssel* der zusätzlichen Datenmengenkomponente (*Datensatz*). Das Feld, welches in die Datenmenge aufgenommen werden soll, wird in der Combobox *Ergebnisfeld* ausgewählt.

Auch eine solche Lookup-Beziehung könnte man mit Hilfe einer SQL-Anweisung definieren:

```
SELECT o.nummer, o.kunde, t.nachname, o.betrag, o.datum
  FROM offenpo o, testadr t
  WHERE o.kunde = t.nummer
```

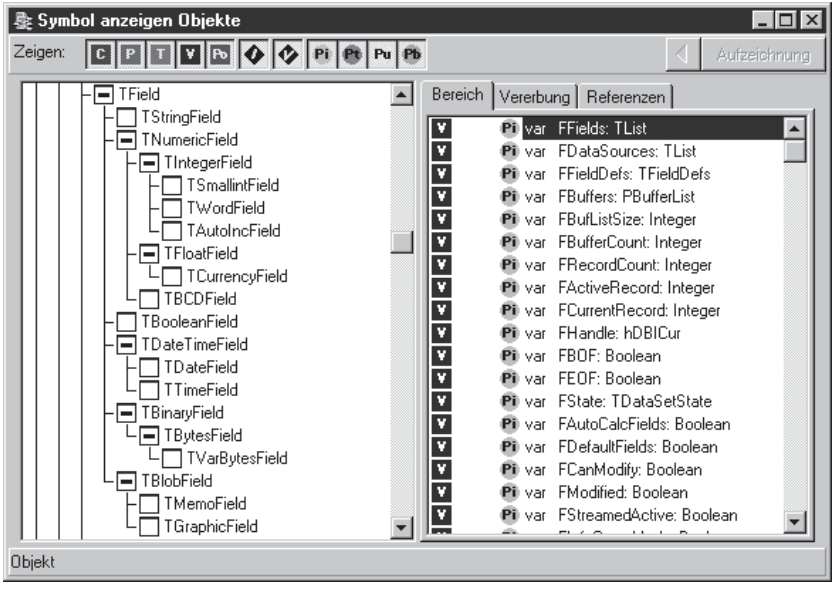


Bild 2.11: Die TField-Nachkommen

2.9.2 Die veröffentlichten TField-Eigenschaften

In C++Builder wird jedes Feld einer Datenbank durch ein Objekt repräsentiert, das von *TField* abgeleitet worden ist (*TStringField*, *TCurrencyField* usw); dieses Objekt hat folglich sowohl öffentliche als auch veröffentlichte Eigenschaften, Methoden und Ereignisse, wobei letztere auch über den Objektinspektor mit einer entsprechenden Prozedur verbunden werden können. Bild 2.10 zeigt die Hierarchie der verschiedenen *TField*-Nachkommen.

- Alignment (*TField*)

  ```
  __property Classes::TAlignment Alignment;
  ```

  Alignment spezifiziert die Ausrichtung des Strings (oder was auch immer) im jeweiligen Dialogelement (mögliche Werte: *taLeftJustify*, *taRightJustify* und *taCenter*). Die Voreinstellung bei String-Feldern ist linksbündig, bei Zahlenfeldern rechtsbündig.

- BlobType (*TBlobField*)

  ```
  __property TBlobType BlobType;
  ```

  Die Eigenschaft *BlobType* kann folgende Werte annehmen:

  – *ftBlob*: BLOB-Feld
  – *ftMemo*: Memo-Feld
  – *ftGraphic*: Bitmap-Feld
  – *ftFmtMemo*: Formatiertes Memo-Feld
  – *ftParadoxOle*: Paradox-OLE-Feld
  – *ftDBaseOle*: dBase-OLE-Feld
  – *ftTypedBinary*: Typisiertes binäres Feld

- Calculated (*TField*)

  ```
  __property bool Calculated;
  ```

  Mit der Eigenschaft *Calculated* wird entschieden, ob es sich um ein berechnetes Feld handelt. Sie können auch bei Feldern, die in der Datenmenge vorhanden sind, die Eigenschaft *Calculated* auf *true* setzen und dann beispielsweise irgendwelche Berechnungen mit dem Feldinhalt durchführen. Bei Lookup-Feldern ist die Eigenschaft *Calculated* = *false*;

- Currency (*TFloatField, TBCDField*)

```
__property bool Currency;
```

  Ist die Eigenschaft *Currency = true*, dann wird die Zahl als Geldbetrag formatiert.

- DisplayFormat (*TNumericField, TDateTimeField*)

```
__property System::AnsiString DisplayFormat;
```

  Mit Hilfe der Eigenschaft *DisplayFormat* kann die Anzeige in den Anzeige- und Dialogelementen formatiert werden. Damit beispielsweise Felder, die nicht vom Typ *TCurrencyField* sind, Zahlen als Geldbeträge darstellen, kann man der Eigenschaft *DisplayFormat* den Wert *0.00 DM* zuweisen. Näheres dazu in der Online-Hilfe.

- DisplayLabel (*TField*)

```
__property System::AnsiString DisplayLabel;
```

  Normalerweise wird als Titel einer *DBGrid*-Spalte der Name des Feldes angegeben. Soll ein anderer Titel angegeben werden, kann dieser mit der Eigenschaft *DisplayLabel* vorgegeben werden. Der Titel kann jedoch auch im Spaltenditor des Datengitters geändert werden.

- DisplayValues (*TBooleanField*)

```
__property System::AnsiString DisplayValues;
```

  Soll die Anzeige von booleschen Feldern in Datengittern und anderen textbasierten Komponenten von *Wahr* und *Falsch* auf beispielsweise *Ja* und *Nein* umgestellt werden, so ist die Eigenschaft *DisplayValues* dafür auf *Ja;Nein* zu setzen.

- DisplayWidth (*TField*)

```
__property int DisplayWidth;
```

  Die Breite einer *DBGrid*-Spalte richtet sich nach der Länge des dazugehörenden Feldes; hier kann mit der Eigenschaft *DisplayWidth* gekürzt oder erweitert werden.

- EditFormat (*TNumericField*)

```
__property System::AnsiString EditFormat;
```

  Mit Hilfe der Eigenschaft *EditFormat* kann die Formatierung eines Feldes gesteuert werden, wenn in einem datensensitiven Dialogelement der Wert bearbeitet wird.

- EditMask (*TStringField*)

```
__property System::AnsiString EditMask;
```

  Bisweilen kann es sinnvoll sein, eine Eingabe in ein bestimmtes Schema zu zwingen (beispielsweise eine Postleitzahl in der Form *12 345*), oft dient dies auch der Überprüfung der Eingabe. Diesem Zweck dient die Eigenschaft *EditMask*.

- FieldName (*TField*)

```
__property System::AnsiString FieldName;
```

  Die Eigenschaft *FieldName* enthält den Spaltennamen. Dieser wird in der Regel von der Tabelle oder der Abfrage übernommen, kann aber auch geändert werden.

- Index (*TField*)

```
__property int Index;
```

  Die Eigenschaft *Index* beinhaltet die Indexnummer in der *Fields*-Eigenschaft der jeweiligen Datenmenge.

- KeyFields (*TField*)

```
__property System::AnsiString KeyFields;
```

  Die Eigenschaft *KeyFields* braucht nur bei Lookup-Feldern gesetzt werden (Felder, bei denen *FieldKind* den Wert *fkLookup* und *Lookup* den Wert *true* hat). *KeyFields* legen das Feld oder die Felder fest, für die bei Auftreten eines Lookup eine Übereinstimmung erreicht werden muß. Sollen mehrere Felder berücksichtigt werden, trennen Sie die Feldnamen durch Semikola.

- Lookup (*TField*)

```
__property bool Lookup;
```

  Die Eigenschaft *Lookup* ist *true*, wenn es sich um ein Lookup-Feld handelt.

- LookupDataSet (*TField*)

```
__property TDataSet* LookupDataSet;
```

  Die Eigenschaft *LookupDataSet* bestimmt die Datenmenge, in der die Feldwerte gesucht werden.

- LookupKeyFields (*TField*)

```
__property System::AnsiString LookupKeyFields;
```

In der Eigenschaft *LookupKeyFields* werden das oder diejenigen Felder angegeben, über welche die Verknüpfung hergestellt wird. Die betreffenden Felder müssen indiziert sein und mit den Feldern der Eigenschaft *KeyFields* übereinstimmen.

- LookupResultField (*TField*)

```
__property System::AnsiString LookupResultField;
```

In der Eigenschaft *LookupResultField* wird das Datenfeld der Lookup-Datenquelle angegeben, welches dann angezeigt wird.

- MaxValue (*TIntegerField, TFloatField, TBCDField*)

```
__property long MaxValue;
```

Die Eigenschaft *MaxValue* bestimmt den maximalen Wert eines Integer-Feldes.

- MinValue (*TIntegerField, TFloatField, TBCDField*)

```
__property long MinValue;
```

Die Eigenschaft *MinValue* legt den minimalen Wert eines Integer-Feldes fest.

- Precision (*TFloatField*)

```
__property int Precision;
```

Legt die Genauigkeit fest, mit der Gleitkommazahlen angezeigt werden.

- ReadOnly (*TField*)

```
__property bool ReadOnly;
```

Wird *ReadOnly* auf *true* gesetzt, dann läßt sich dieses Feld nicht mehr verändern.

- Required (*TField*)

```
__property bool Required;
```

Wird *Required* auf *true* gesetzt, dann muß für das Feld ein Wert eingegeben werden, andernfalls wird eine Exception ausgelöst.

- Size (*TStringField, TBlobField*)

```
__property unsigned short Size;
```

Die Eigenschaft *Size* gibt die Größe des Datenfeldes in Bytes an.

- Transliterate (*TStringField, TMemoField*)

```
__property bool Transliterate;
```

Setzen Sie die Eigenschaft *Transliterate* auf *true*, wenn die durch die Datenmenge bezeichnete physikalische Datenbanktabelle keinen ANSI-Sprachtreiber verwendet und die Daten ASCII-Sonderzeichen (deutsche Umlaute beispielsweise) enthalten.

- Visible (*TField*)

```
__property bool Visible;
```

Wird *Visible* auf *false* gesetzt, dann wird das betreffende Feld im Datengitter nicht angezeigt. Dasselbe kann auch mit dem Spalteneditor der *TDBGrid*-Komponente erreicht werden.

2.9.3 Die öffentlichen TField-Eigenschaften

Bei den öffentlichen Eigenschaften von *TField* tritt eine Besonderheit auf: Normalerweise können alle öffentlichen und veröffentlichten Eigenschaften einer Variablen auch verwendet werden. Eigenschaften, die sich nicht verwenden lassen, sondern nur für zu erstellende Nachkommen implementiert werden, werden als *protected* deklariert.

Bei *TField* gibt es jedoch die *As*...-Eigenschaften, welche beim Lese- oder Schreibzugriff eine Exception auslösen. Diese Eigenschaften können erst bei den dafür geeigneten Nachkommen verwendet werden – *AsBoolean* beispielsweise bei *TBooleanField*.

- AsBoolean (*TField*)

```
__property bool AsBoolean;
```

- AsCurrency (*TField*)

```
__property System::Currency AsCurrency;
```

- AsDateTime (*TField*)

```
__property System::TDateTime AsDateTime;
```

- AsFloat (*TField*)

```
__property double AsFloat;
```

- AsInteger (*TField*)

```
__property long AsInteger;
```

- AsString (*TField*)

```
__property System::AnsiString AsString;
```

- AsVariant (*TField*)

```
__property System::Variant AsVariant;
```

- AttributeSet (*TField*)

```
__property System::AnsiString AttributeSet;
```

  Die Eigenschaft *AttributeSet* ist der Name der Attributmenge im Data-Dictionary, die auf diese Feldkomponente angewendet wird.

- BDECalcField (*TField*, nur Lesen)

```
__property bool BDECalcField;;
```

  Die Eigenschaft *BDECalcField* ist gleich *true*, wenn ein Feld im Rahmen einer SQL-Abfrage von der BDE berechnet wurde. Im folgenden Beispiel wäre *gesamtpreis* ein solches berechnetes Feld:

```
SELECT anzahl, artikelnummer, einzelpreis,
  anzahl * einzelpreis AS gesamtpreis
  FROM ...
```

- CanModify (*TField*, nur Lesen)

```
__property bool CanModify;
```

  Die Eigenschaft *CanModify* ist gleich *true*, wenn sich das betreffende Feld ändern läßt. Nicht ändern lassen sich beispielsweise berechnete Felder oder Felder in »nicht lebenden« Abfragen.

- DataSet (*TField*)

```
__property TDataSet* DataSet;
```

  Die Komponenteneigenschaft *DataSet* verweist auf die Datenmengenkomponente, aus der das Feld stammt.

- DataSize (*TField*, nur Lesen)

```
__property unsigned short DataSize;
```

  Grüße des Speichers in Byte, der für das betreffende Feld benötigt wird. Liefert für BLOB-Felder und Nachfahren den Wert null zurück – näheres siehe Online-Hilfe.

- DataType (*TField*, nur Lesen)

```
__property TFieldType DataType;
```

  Die Eigenschaft *DataType* gibt den Datentyp des Feldes an. String-Felder haben beispielsweise den Typ *ftString*.

- DisplayName (*TField*, nur Lesen)

```
__property System::AnsiString DisplayName;
```

  Die Eigenschaft *DisplayName* liefert den aktuellen Feldnamen zu Anzeigezwecken. Ist *DisplayLabel* definiert, dann ist *DisplayName* gleich *DisplayLabel*, andernfalls ist *DisplayName* gleich *FieldName*.

- DisplayText (*TField*, nur Lesen)

```
__property System::AnsiString DisplayText;
```

  Der Inhalt von *DisplayText* entspricht normalterweise dem von *AsString*. Mit Hilfe der *OnGetText*-Ereignisbehandlungsroutine kann davon Abweichendes programmiert werden.

- FieldKind (*TField*)

```
__property TFieldKind FieldKind;
```

  Die Eigenschaft *FieldKind* kann folgende Werte annehmen:

  – *fkData*: Feld aus einer Datenbank
  – *fkCalculated*: berechnetes Feld
  – *fkLookup*: Lookup-Feld.

- FieldNo (*TField*, nur Lesen)

```
__property int FieldNo;
```

  Feldnummer der eigentlichen Datenbanktabelle. Wird nur für Funktionen der BDE-API benötigt.

- IsIndexField (*TField*, nur Lesen)

```
__property bool IsIndexField;
```

  Die Eigenschaft *IsIndexField* ist gleich *true*, wenn das betreffende Feld indiziert ist.

- IsNull (*TField*, nur Lesen)

```
__property bool IsNull;
```

  Bei einem leeren Datenfeld ist die Eigenschaft *IsNull* gleich *true*.

- NewValue (*TField*)

```
__property System::Variant NewValue;
```

Die Werte von *NewValue* und *Value* sind identisch, nur während des Eintragens von zwischengespeicherten Aktualisierungen nicht unbedingt. Wird *NewValue* in einer Ereignisbehandlungsroutine für *OnUpdateError* oder für *OnUpdateRecord* gesetzt, unterscheiden sich die Werte *NewValue* und *Value*, bis die zwischengespeicherten Aktualisierungen vollständig in der zugrundeliegenden Datenbanktabelle eingetragen sind.

- OldValue (*TField*, nur Lesen)

```
__property System::Variant OldValue;
```

Durch *OldValue* kann der ursprüngliche Wert des Feldes ermittelt werden, den es vor dem Eintragen zwischengespeicherter Aktualisierungen hatte.

- Size (*TField*)

```
__property unsigned short Size;
```

Größe des Feldes in Zeichen (*TStringField*, *TMemoField*) oder Bytes (*TBlobField* und Nachkommen).

- Text (*TField*)

```
__property System::AnsiString Text;
```

Für gewöhnlich entspricht der Inhalt der Eigenschaft *Text* dem der Eigenschaft *AsString*. Abweichungen können mit Hilfe der *OnGetText*-Ereignisbehandlungsroutine implementiert werden.

- Value (*TField*)

```
__property System::Variant Value;
```

Die Eigenschaft *Value* enthält den Inhalt des Feldes. Die Nachfahren von *TField* implementieren diese Eigenschaft mit verschiedenen Datentypen.

2.9.4 Die Methoden von TField

In der folgenden Referenz der Methoden von *TField* und Nachfahren sollen die Konstruktoren und Destruktoren nicht erwähnt werden.

- Assign (*TField*)

```
virtual void __fastcall Assign(Classes::TPersistent* Source);
```

Mit Hilfe von *Assign* kann einem Feld der Inhalt eines anderen Feldes oder eines anderen *TPersistent*-Nachfahren zugewiesen werden. Bei den Nachfah-

ren von *TBlobField* ist dies in der Regel der einzige Weg, um auf den Feldinhalt zugreifen zu können.

```
Table1->FieldByName("Graphik")->Assign(Image1->Picture);
```

- AssignValue (*TField*)

```
void __fastcall AssignValue(const System::TVarRec &Value);
```

Die Methode *AssignValue* weist dem Feld mit Hilfe der Eigenschaft *AsString*, *AsInteger*, *AsBoolean* oder *AsFloat* den Wert *Value* zu. In der Regel ist es sinnvoller, den Wert der entsprechenden Eigenschaft direkt zuzuweisen.

- Clear (*TField*)

```
virtual void __fastcall Clear(void);
```

Die Methode *Clear* löscht den Feldinhalt und weist somit dem Feld den Wert NULL zu.

- FocusControl (*TField*)

```
void __fastcall FocusControl(void);
```

Die Methode *FocusControl* setzt den Eingabefokus auf ein mit dieser Komponente verbundenes Steuerelement. Diese Methode sollte beispielsweise dann aufgerufen werden, wenn bei der Validierung ein Fehler festgestellt worden ist, so daß der Anwender diesen unverzüglich beheben kann.

- GetData (*TField*)

```
bool __fastcall GetData(void * Buffer);
```

Enthält das Feld Daten, dann werden diese unformatiert in den übergebenen Puffer geschrieben, andernfalls wird *false* zurückgegeben.

- IsValidChar (*TField*)

```
virtual bool __fastcall IsValidChar(char InputChar);
```

Gibt den Wert *true* zurück, wenn die Eingabe des Zeichens *InputChar* in das betreffende Feld erlaubt ist.

- LoadFromFile (*TBlobField*)

```
void __fastcall LoadFromFile(const System::AnsiString FileName);
```

Die Methode *LoadFromFile* lädt den Wert aus einer Datei in das Feld.

- LoadFromStream (*TBlobField*)

```
void __fastcall LoadFromStream(Classes::TStream* Stream);
```

Die Methode *LoadFromStream* lädt den Wert aus einem Stream in das Feld.

- SaveToFile (*TBlobField*)

```
void __fastcall SaveToFile(const System::AnsiString FileName);
```

  Die Methode *SaveToFile* speichert den Feldinhalt in eine Datei.

- SaveToStream (*TBlobField*)

```
void __fastcall SaveToStream(Classes::TStream* Stream);
```

  Die Methode *SaveToStream* speichert den Feldinhalt in einen Stream.

- SetData (*TField*)

```
void __fastcall SetData(void * Buffer);
```

  Die Methode *SetData* weist dem Feld die unformatierten Daten im Puffer zu.

- SetFieldType (*TField*)

```
virtual void __fastcall SetFieldType(TFieldType Value);
```

  Die Methode *SetFieldType* ist lediglich eine abstrakte Methode – damit sie verwendet werden darf, muß sie erst entsprechend implementiert werden. Für *TBlobField* und Nachfahren ist diese Methode implementiert und erlaubt folgende Parameterwerte:

  – *ftBlob:* BLOB-Feld
  – *ftMemo:* Memo-Feld
  – *ftGraphic:* Bitmap-Feld
  – *ftFmtMemo:* formatiertes Memo-Feld
  – *ftParadoxOle:* Paradox-OLE-Feld
  – *ftDBaseOle:* dBase-OLE-Feld
  – *ftTypedBinary:* typisiertes binäres Feld

2.9.5 Die Ereignisse von TField

- OnChangeField (*TField*)

```
OnChange(TField *Sender)
```

  Das Ereignis *OnChange* tritt nach dem Speichern der Felddaten auf.

- OnGetText (*TField*)

```
OnGetText(TField *Sender, AnsiString &Text, bool DisplayText)
```

Das Ereignis *OnGetText* wird ausgelöst, wenn die Eigenschaften *DisplayText* oder *Text* gelesen werden. Der Parameter *Text* nimmt den formatierten Text auf, der von den Eigenschaften *DisplayText* oder *Text* verwendet wird. Der Parameter *DisplayText* zeigt an, ob der Text nur zur Anzeige oder auch bei der Bearbeitung verwendet werden soll.

- OnSetText (*TField*)

```
OnSetText(TField *Sender, const AnsiString Text)
```

Das Ereignis *OnSetText* wird ausgelöst, wenn der Eigenschaft *Text* ein Wert zugewiesen wird.

- OnValidate (*TField*)

```
OnValidate(TField *Sender)
```

Vor dem Speichern der Daten wird das Ereignis *OnValidate* ausgelöst. Dadurch ist eine Überprüfung der Eingabe möglich.

3 Abfragen mit TQuery

Die Verbindung zu einer Datenbank kann nicht nur über die Komponente *TTable* erfolgen, sondern auch über die Komponente *TQuery*. Dabei handelt es sich um eine Komponente, die sich hauptsächlich für Abfragen eignet und dabei den Industrie-Standard SQL als Abfragesprache verwendet.

Wir wollen zunächst wieder einer kleines Beispiel programmieren, um uns danach ausführlich mit den Eigenschaften, Methoden und Ereignissen von *TQuery* zu beschäftigen.

3.1 Suche nach Telefonnummern

Ein Thema, das immer wieder für viel »Freude« bei Entwicklern und Anwendern sorgt, ist das Thema »Datenbanken und Telefonnummern«. Ganz Unbedarfte werden sich nichts dabei denken und dafür einfach ein numerisches Feld verwenden. Der Anwender gibt dann dort beispielsweise *0305134505* ein, und die Datenbank macht *305134505* daraus. Wenn der Anwender geduldig und tolerant ist, dann wird er sich daran gewöhnen, spätestens aber dann, wenn ausländische und einheimische Nummern durcheinander gehen, dürfte der Geduldsfaden reißen.

Man wird also nicht umhin kommen, für Telefonnummern ein alphanumerisches Feld zu verwenden und dies so groß zu dimensionieren, daß man zur Not auch noch eine Auslandstelefonnummer darin unterbringen kann.

Diesen Platz wird der Anwender sofort nutzen, um die Telefonnummer nach seiner »Hausnorm« zu formatieren. So wird aus der gerade erwähnten Nummer *030/5134505, 030 / 5134505, 030 / 513 45 05, 030 / 51 34 505, 030 513 45 05, 030 - 5135 45 05* oder was es da noch an mehr oder weniger sinnvollen Kombinationen gibt. So weit ist das alles kein Problem, die Datenbank akzeptiert das alles problemlos.

Wenn aber nach dieser Telefonnummer gesucht werden soll, wird es ziemlich schwierig, weil bei einer Stringsuche die exakte Schreibweise bekannt sein sollte. Nun könnte man einwenden, daß der Anwender, wenn er schon seine Telefonnummern formatiert, dies wenigstens so einheitlich tun sollte, daß er seine Einträge wiederfindet. Ich bin allerdings der Meinung, daß Programme, die sich professionell nennen, auch dann möglichst fehlerfrei arbeiten, wenn der Anwender ziemlich schlampig damit umgeht.

Zudem – und das ist wohl das gewichtigere Argument – muß man bei einer Datenbank stets davon ausgehen, daß mehrere Anwender damit arbeiten, und es ist reichlich naiv zu glauben, daß sich auch nur zwei Personen auf eine einheitliche Schreibweise von Telefonnummern einigen könnten. Zur Lösung dieses Problems gibt es nun mehrere Möglichkeiten:

- Durch Vorgabe einer Maske wird die Formatierung durch den Anwender verhindert, das Programm akzeptiert nur Ziffern. Dies ist die »Holzhammer-Methode«; sie funktioniert, aber eine Ziffernfolge wie *0305134505* ist alles andere als leicht (= fehlerfrei) zu erfassen.
- Die Eingabemaske zwingt den Anwender zur Einhaltung einer Norm. Dies ist gerade bei Telefonnummern mit deren Trennung in Vor- und Durchwahl, gegebenenfalls zuzüglich Anschlußnummer oder Auslandsvorwahl, und alles in variabler Länge, ein reichlich schwieriges Unterfangen. Ganz abgesehen davon, daß ich mir als Anwender nicht vom Rechner vorschreiben lassen möchte, wie ich meine Nummern zu formatieren habe.
- Die Telefonnummer wird zweimal gespeichert: Einmal als vom Anwender formatierter String und zum anderen als reine Ziffernfolge, welche das Programm aus der Anwendereingabe generiert. Angezeigt wird dabei nur die Anwendereingabe, gesucht nur in der Ziffernfolge. Das ist nun eine uneingeschränkt saubere Lösung, die lediglich den Nachteil hat, daß mehr Speicherplatz dafür benötigt wird.
- Man kann auch die Abfrage so formulieren, daß die Schreibweise keine Rolle spielt; wie das funktioniert, werden wir uns gleich ansehen. Auch diese Version hat einen Nachteil: Die Suche dauert etwas länger. Da die Abfrage selbst jedoch sehr schnell geht (zum Beispiel 911 msec bei direkter Suche, 982 msec bei der schreibweisentoleranten Suche), halte ich das für ein untergeordnetes Problem. Bei wirklich großen Datenmengen (ab 100 000 Adressen aufwärts) kann man dann zwischen dieser und der vorhergehenden Lösung abwägen.

3.1.1 Ein Programm zur Telefonnummernsuche

Das Beispielprogramm soll Datensätze über die Telefonnummer suchen. Dabei soll die Testdatenbank verwendet werden, welche in Kapitel 2 erstellt wurde. Die Zeit, welche für die Abfrage benötigt wird, soll vom Programm gemessen werden. Um das Programm zu verstehen, ist eine kurze Einweisung in den SQL-Befehl SELECT nötig. Im weiteren Verlauf des Buches werden wir die Abfragesprache SQL noch ausführlich behandeln.

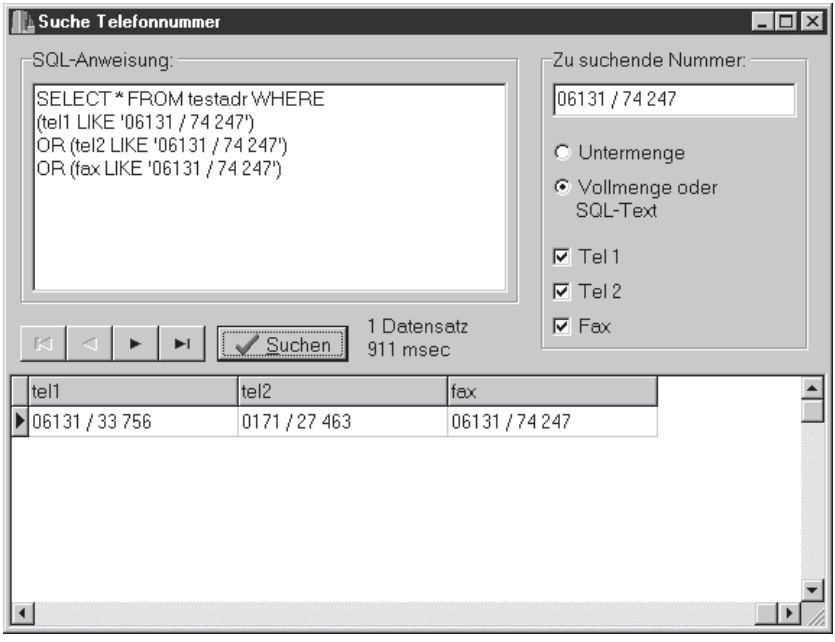


Bild 3.1: Suche nach Telefonnummern mit SQL

Kurz-Einführung in den SQL-Befehl SELECT

Der Befehl SELECT dient der Abfrage von Daten aus einer Tabelle. Seine einfachste Form lautet:

```
SELECT * FROM adressen
```

Dies würde heißen, daß als Datenmenge alle Spalten übergeben werden, welche in der Tabelle *adressen* vorhanden sind. Laut Konvention werden alle SQL-Schlüsselwörter mit Großbuchstaben und alle Variablen mit Kleinbuchstaben geschrieben.

Solange keine Beschränkung genannt wird, werden alle Datensätze der Tabelle übergeben, genau wie dies im Regelfall die Komponene *TTable* tun würde. Nun soll hier allerdings eine Beschränkung erfolgen, da ja nicht alle Adressen angezeigt werden sollen, sondern nur diejenige, nach der über die Telefonnummer gesucht wird. Dazu dient die WHERE-Klausel:

```
SELECT * FROM adressen
WHERE tel1 = "030 / 513 45 05"
```

Hier werdem dann nur noch die Datensätze übergeben, deren Feld *tel1* den Wert *030 / 513 45 05* aufweist. Hier ist eine exakte Schreibweise zwingend erforderlich.

Übergebene Strings müssen übrigens auch bei SQL in Anführungszeichen gesetzt werden, wobei hier sowohl einfache (') als auch doppelte (") Anführungszeichen verwendet werden können.

Um Abfragen zu generieren, welche nach Ähnlichkeiten suchen, gibt es den LIKE-Operator, der zwei Joker-Zeichen erlaubt: Der Unterstrich (_) ersetzt dabei ein einzelnes Zeichen, während das Prozentzeichen (%) kein, ein oder mehrere Zeichen ersetzt.

```
SELECT * FROM adressen
WHERE tel1 LIKE "030%"
```

Diese Abfrage würde nun Datensätze zurückgeben, die mit *030* beginnen, also alle Berliner Telefonnummern. Die WHERE-Klauseln erlauben auch logische Verknüpfungen:

```
SELECT * FROM adressen
WHERE (tel1 LIKE "030%")
  OR (tel1 LIKE "040%")
```

Diese Anwendung würde nun alle Datensätze zurückgeben, die eine Berliner oder Hamburger Vorwahl haben.

```
SELECT * FROM adressen
WHERE (tel1 LIKE "030%")
  AND (tel2 LIKE "0177%")
```

Diese Anweisung sucht nun alle Berliner mit E-Netz-Handy. Zurück zu unserem eigentlichen Problem: Gesucht werden soll die Nummer 030 / 513 45 05, unabhängig von der Schreibweise. Die Lösung ist folgende SQL-Anweisung:

```
SELECT * FROM adressen
WHERE (tel1 LIKE "0%3%0%5%1%3%4%5%0%5%")
```

Erstellt wird nun ein Programm, das aus einer übergebenen Telefonnummer eine solche SQL-Anweisung erstellt.

Das Beispielprogramm

Zunächst wird ein Formular gemäß Bild 3.1 mit Komponenten bestückt. Bis auf das *DBGrid* und den *DBNavigator* handelt es sich um normale, also um nicht datensensitive Komponenten. Der *DBNavigator* wird übrigens auf die Schaltflächen reduziert, die auch tatsächlich benötigt werden, Näheres siehe in Kapitel 4.4.4. Des weiteren werden noch eine *TQuery*- und eine *TTable*-Komponente benötigt.

Der Button wird nun mit folgender *OnClick*-Ereignisbehandlungsprozedur verknüpft:

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
  int i, t1, t2;
  AnsiString s, t = "";
  char u;
  char v[10];
  bool erster = true;
// Aufbereiten der Zeichenfolge
  s = Edit1->Text;
  for(i = 1; i<= strlen(&s[1]); i++)
  {
    u = s[i];
    if(('0' <= u) && (u <= '9')) t = t + u + "%";
  }
```

Zunächst wird die eingegebene Nummer entsprechend aufgearbeitet. Die Telefonnummer, welche der Anwender in *Edit1* eingegeben hat, wird in einzelne Zeichen zerlegt, die dann daraufhin untersucht werden, ob es sich um eine Ziffer handelt. Ist dies der Fall, dann wird diese gefolgt von einem %-Zeichen dem String *t* hinzugefügt.

```
// SQL-Anweisung generieren
  Memo1->Clear();
  Memo1->Lines->Add("SELECT * FROM testadr WHERE");
  if(CheckBox1->Checked == true)
  {
    if(RadioButton1->Checked == true)
      Memo1->Lines->Add("(tel1 LIKE '" + t + "')");
    else
      Memo1->Lines->Add("(tel1 LIKE '" + Edit1->Text + "')");
    erster = false;
  }
```

Bei der Generierung der SQL-Anweisung wird zunächst die alte Anweisung gelöscht und der Beginn der Anweisung eingefügt. Ist *CheckBox1.Checked* gleich *true*, dann soll im ersten Feld nach der Telefonnummer gesucht werden. Vom Zustand des *RadioButton1* hängt es ab, ob es sich um eine streng kongruente Suche handelt oder ob die Suche verschiedene Formatierungen toleriert. Wird eine strenge Suche gewählt, dann lassen sich auch direkt SQL-Sequenzen eingeben; so könnte man mit der Eingabe *030%* direkt nach Berliner Vorwahlen suchen, während bei der toleranten Suche nach *030* (*0%3%0*) auch Telefonnummern wie ***0****40 / 2****3****7 ****0****9 68* gefunden werden könnten. (Im Beispielprojekt *Tourplaner* werden wir ein Suchfenster entwickeln, welches auch explizit nach Vorwahlen sucht.) Zur Variablen *erster* kommen wir gleich.

```
  if(CheckBox2->Checked == true)
  {
    if(erster == true)s = ""; else s = "OR ";
    if(RadioButton1->Checked == true)
      s = s + "(tel2 LIKE '" + t + "')";
    else
      s = s + "(tel2 LIKE '" + Edit1->Text + "')";
    Memo1->Lines->Add(s);
    erster = false;
  }
```

Im Prinzip handelt es sich bei dieser Sequenz für das Feld *tel2* um die gleichen Anweisungen. Zusätzlich muß jedoch geprüft werden, ob die generierten SQL-Anweisungen die ersten sind (dazu dient die Variable *erster*) oder nicht. Ist letzteres der Fall, dann muß ein OR-Operator eingefügt werden.

Die Sequenz bezüglich der *CheckBox3* und des Fax-Feldes ist ähnlich aufgebaut und braucht hier nicht aufgeführt zu werden.

```
  if(erster == true)
    Memo1->Text = "SELECT * FROM testadr";
// SQL-Abfrage durchführen
  Screen->Cursor = crHourGlass;
  Query1->SQL = Memo1->Lines;
  t1 = GetCurrentTime();
  Query1->Open();
  t2 = GetCurrentTime();
  s = ("%n", i = Query1->RecordCount);
  if(Query1->RecordCount == 1)
    Label2->Caption = "1 Datensatz";
  else
    Label2->Caption = s + " Datensätze";
  s = ("%n", t2 - t1);
  Label3->Caption = s + " msec";
  Screen->Cursor = crDefault;
} // TForm1::BitBtn1Click
```

Zunächst wird nun daraufhin geprüft, ob die Suche für wenigstens ein Telefonnummernfeld aktiviert wurde, ansonsten würde nämlich die SQL-Anweisung *SELECT * FROM testadr WHERE* lauten, was keinen Sinn ergibt. In diesem Fall würde die SQL-Anweisung so formuliert, daß alle Datensätze angezeigt werden.

Während der Abfrage wird der Cursor in eine Sanduhr verwandelt. Die Anweisunungen werden nun von *Memo1* ist die Eigenschaft *SQL* von *Query1*

kopiert, danach wird die Ausführung mit *Query1->Open* gestartet. Da wir die Zeit der Ausführung messen wollen, wird direkt vor dem Start und nach Beendigung der Anweisung die Systemzeit in den Variablen *t1* und *t2* festgehalten. Anschließend werden die Datensätze gezählt, die benötigte Zeit wird berechnet, und beide Ergebnisse werden angezeigt.

3.2 Die Datenbanksprache SQL

Die Datenbanksprache SQL hat sich insbesondere im Bereich der Client-Server-Datenbanken als Standard durchgesetzt. Sie verbindet die Vorteile der weiten Verbreitung mit der leichten Erlernbarkeit und den mächtigen Funktionen.

Die Komponente *TQuery* setzt SQL als Abfragesprache ein, weshalb hier zunächst eine Einführung in SQL erfolgen soll, bevor wir uns mit den Eigenschaften, Methoden und Ereignissen von *TQuery* befassen. Abschließend sollen die Vor- und Nachteile von *TQuery* und *TTable* besprochen werden.

Wir wollen hier jedoch nicht den gesamten Sprachumfang von SQL besprechen; viele SQL-Befehle werden bei der Verwendung von *TQuery* wohl kaum eingesetzt, manche sind beim Zugriff auf die BDE auch nicht verwendbar, sondern können nur an einen SQL-Server weitergeleitet werden. Eine Besprechung dieser Befehle erfolgt dann bei Behandlung des *Local InterBase Servers*.

Den Befehlssatz von SQL kann man in zwei Gruppen einteilen:

- Die Daten-Definitionsanweisungen (data definition language, DDL), mit denen Tabellen, Ansichten, usw. definiert, geändert oder gelöscht werden.
- Die Daten-Manipulationsanweisungen (data manipulation language, DML), zu denen die Befehle zum Einfügen, Ändern und Löschen von Daten sowie der Abfragebefehl SELECT gehören.

Hier sollen zunächst nur der Befehl SELECT (DML) und der Befehl CREATE TABLE (DDL) besprochen werden.

3.2.1 Der Befehl SELECT

Die vollständige abstrakte SELECT-Befehl lautet

```
SELECT DISTINCT [columns]
  FROM [tables]
  WHERE [search_conditions]
  GROUP BY [columns] HAVING [search_conditions]
  ORDER BY [sort_orders]
```

Nicht alle Befehlsoptionen müssen dabei benutzt oder aufgeführt werden. Im einfachsten Fall lautet der SELECT-Befehl:

```
SELECT * FROM testadr
```

Dies würde bedeuten, daß alle Spalten (und alle Datensätze) der Tabelle *adressen* zurückgegeben werden.

Zum Test der folgenden Beispiele gibt es auf der beiliegenden CD (unter Kapitel 3) einen SQL-Editor, bei dem die Beispiel-Anweisungen sich über das Menü einfügen lassen; selbstverständlich ist eine anschließende Modifikation möglich. Im Text werden Hinweise auf diese Menüpunkte geklammert angeführt, beispielsweise: (Text1.1).

Spezifizieren von Spalten und Tabellen, JOINs

Nicht immer ist es wünschenswert, sich alle Spalten einer Tabelle anzeigen zu lassen. Nach dem Befehl SELECT können die Spalten angegeben werden, welche zurückgegeben werden sollen, oder es kann – wie wir das bislang getan haben – mit dem Joker-Zeichen * die Rückgabe aller Spalten veranlaßt werden:

```
SELECT vorname, nachname FROM testadr
```

Diese Anweisung (Text 1.1) würde nun lediglich die Spalten *vorname* und *nachname* zurückgeben.

Des weiteren besteht die Möglichkeit, mehrere Spalten zusammenzufassen und den Feldern vorgegebene Zeichen (-folgen) hinzuzufügen. Die nächste Beispielanweisung (Text 1.2) entnehmen Sie bitte Bild 3.2.

Die verschiedenen Spalten werden mit Hilfe des ||-Zeichens (zweimal Alt Gr + <) zusammengefügt. In der Regel wird es dabei notwendig sein, ein Leerzeichen einzufügen. Die Rückgabe von vorgegebenen Strings ist nicht auf die Verbindung mit Spaltenwerten beschränkt, Sie können auch alle Felder mit demselben String füllen. Sinvoll könnte dies beispielsweise dann sein, wenn mit *TBatchMove* Daten in eine andere Datenbank eingefügt werden sollen. Beachten Sie auch die Spaltenüberschrift im *DBGrid*.

Noch ein Hinweis: Wenn hier dauernd von *Spalten* gesprochen wird, dann ist damit das gemeint, was man im C++Builder *Felder* nennt.

Bisweilen möchte man Daten aus mehreren Tabellen kombinieren. Nehmen wir an, wir hätten in einer Tabelle die offenen Posten (Forderungen) einer Firma. Neben Datum und Betrag sind dort auch die Kunden gespeichert, die diesen Betrag schulden. Ganz im Sinne der Theorie relationaler Datenbanken wird dort aber nicht die volle Kundenadresse gespeichert, sondern nur die Kundennummer.

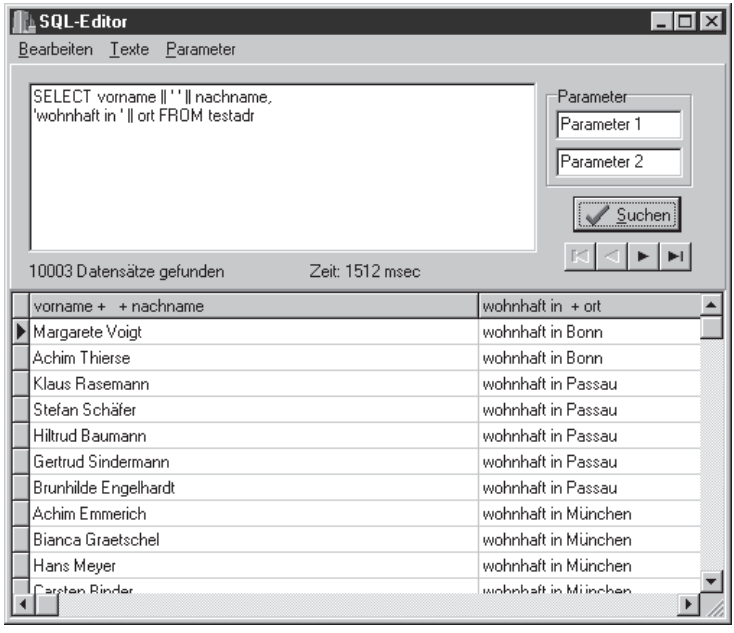


Bild 3.2: Zusammenfassung von Feldern

Wird nun eine Abfrage nach den offenen Posten gestartet, dann bekäme man aber gerne nicht nur die Kundennummern (wer kennt die schon alle auswendig), sondern auch die vollständigen Adressen angezeigt. Dazu ist eine Verbindung von mehreren Tabellen, ein sogenannter »JOIN« nötig. Die entsprechende Anweisung (Text 1.3) lautet:

```
SELECT offenpo.nummer, testadr.vorname || " " || testadr.nachname,
    testadr.straße, testadr.ort, offenpo.betrag, offenpo.datum
  FROM offenpo, testadr
  WHERE offenpo.kunde = testadr.nummer
```

Nach dem FROM-Befehl werden die Tabellen aufgelistet, aus denen bei dieser Abfrage Daten bezogen werden. Nach dem SELECT-Befehl folgen wie immer die einzelnen Spalten; neu ist hier, daß vor jedem Spaltennamen der Tabellenname steht. Die Notwendigkeit dieser Angabe ist leicht einzusehen: Sowohl die Tabelle *offenpo* als auch die Tabelle *testadr* haben ein Feld *nummer*; würde die Tabelle nicht spezifiziert, dann wüßte die Abfrage nicht, welche Spalte nun gemeint ist.

Tabellenname und Spaltenname werden durch einen Punkt getrennt. Dies ist der Grund, warum Sie bei Spaltennamen (Feldnamen) keine Punkte verwenden sollten, auch wenn die Datenbankoberfläche dies akzeptiert: Bei der ersten SQL-Anweisung würden sie Probleme bekommen.

Neben den Spalten und den Tabellen müssen Sie auch noch eine Verknüpfungsanweisung eingeben, wozu die Anweisung nach dem Schlüsselwort WHERE dient. In diesem Fall werden alle Datensätze angezeigt, bei denen die Felder *offenpo.kunde* und *testadr.nummer* übereinstimmen. Dies bezeichnet man als INNER JOIN oder als EQUI JOIN (engl. *equal* gleich). Andere JOINS werden wir uns gleich ansehen.

Zunächst aber zu einem anderen Problem: Das dauernde Wiederholen des Tabellennamens macht das Schreiben der SQL-Anweisung kompliziert und das Ergebnis unübersichtlich. Deshalb erlaubt es SQL, *Synonyme* (manchmal auch Tabellen-Alias genannt) zu verwenden (Text 1.4):

```
SELECT o.nummer, t.vorname || " " || t.nachname,
    t.straße, t.ort, o.betrag, o.datum
  FROM offenpo o, testadr t
  WHERE o.kunde = t.nummer
```

Dazu wird hinter den Tabellennamen (hinter dem Schlüsselwort FROM), getrennt durch ein Leerzeichen, jeweils das Synonym angeführt (hier *o* und *p*). An allen anderen Stellen kann dann statt des Tabellennamens das Synonym angegeben werden. Das Synonym besteht hier nur aus einem einzelnen Buchstaben, es sind aber auch längere Synonyme erlaubt.

Nun zurück zu den JOINs und zu unserer *Offene-Posten*-Tabelle. Für gewöhnlich wird man über eine Referenz sicherstellen, daß für alle *o.kunde*-Werte ein gültiger Datensatz in der Tabelle *testadr* vorliegt. Nun wollen wir einmal annehmen, daß dies nicht geschehen sei, und nun kommt es wie es kommen muß: Für eine oder mehrere offene Posten sind Kundennummern angegeben, für die kein Adressen-Datensatz existiert.

Nun benötigt die Buchhaltung eine Übersicht über die offenen Posten, damit die Bilanz erstellt werden kann. Bei einer Abfrage mit oben angegebener Anweisung würde eine Liste der offenen Posten erstellt, bei denen die Adresse des Kunden in der Adressentabelle vorhanden ist. Wird auf dieser Grundlage nun die Bilanz erstellt, dann kann das Ergebnis nicht stimmen, weil die offenen Posten mit »unbekannter« Adresse unterschlagen werden. (Im Gegensatz zu den Naturwissenschaften sind bei der Buchhaltung Näherungen grundsätzlich unzulässig.)

Wir benötigen also eine Anweisung, die anordnet: *Suche alle offenen Posten. Wo die Adresse des Kunden bekannt ist, gebe diese an*. Diese Anweisung wird mit einem OUTER JOIN realisiert (Text 1.5):

```
SELECT o.nummer,
    t.vorname || " " || t.nachname,
    t.straße, t.ort, o.betrag, o.datum
  FROM offenpo o LEFT OUTER JOIN testadr t
  ON o.kunde = t.nummer
```

Bei einem LEFT OUTER JOIN werden alle Datensätze der linken Seite (also der Tabelle, die vor dem LEFT OUTER JOIN steht) angezeigt. Wo sich Werte der rechten Seite zuordnen lassen, wird dies getan, wo dies nicht möglich ist, werden leere Zellen zurückgegeben.

Des weiteren gibt es auch den RIGHT OUTER JOIN (Text 1.6), der in diesem Fall alle Kundenadressen zurückgeben würde und, sofern bei diesem Kunden offene Posten bestehen, Datum und Betrag.

Zu klären ist noch, was bei Personen geschieht, bei denen mehrere Rechnungen offen stehen. Bei einem LEFT OUTER JOIN werden – wie nicht anders zu erwarten – alle Datensätze der Tabelle *offenpo* angezeigt, bisweilen tritt dieselbe Adresse mehrmals auf. Interessanter wird es beim RIGHT OUTER JOIN: Dort wird der Adressendatensatz zweimal zurückgegeben, einmal kombiniert mit dem ersten offenen Posten, das zweite mal mit dem zweiten; Bild 3.3 zeigt diese Verhaltensweise.

Des weiteren gibt es den FULL OUTER JOIN (Text 1.7): Dieser führt die Datensätze beider Tabellen auf und verbindet diese, wo dies möglich ist. Es würden also Kunden mit unbezahlten Rechnungen, Kunden ohne unbezahlte Rechnungen und unbezahlte Rechnungen ohne Kunden zurückgegeben. In unserer Beispielanwendung startet die Rückgabe immer mit dem ersten Datensatz aus

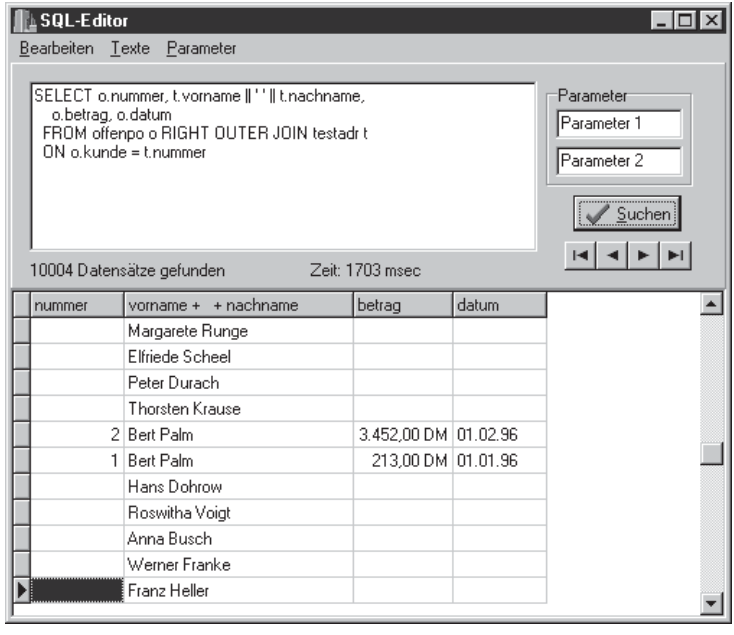


Bild 3.3: Doppelte Einträge beim OUTER JOIN

der Tabelle *testadr* (zumindest auf meinem Rechner), auch wenn man die Angabe der beiden Tabellen vertauscht hat.

Ein Kuriosum ist der *Self-Join*, bei dem derselben Tabelle zwei unterschiedliche Synonyme zugewiesen werden (Text 1.8):

```
SELECT * FROM testadr t, testadr a
  WHERE (t.tel1 = a.tel1)
    AND (t.nummer <> a.nummer)
```

Hier werden alle Datensätze zurückgegeben, welche dieselbe Telefonnummer *tel1* haben, aber eine andere Datensatznummer; gesucht werden also verschiedene Personen mit der gleichen Telefonnummer. Hätten zwei Datensätze dieselbe Telefonnummer, die aber jeweils anders geschrieben wäre, würden diese allerdings nicht ausgegeben. In meiner Testadressen-Tabelle befanden sich immerhin 42 Adressen mit 21 verschiedenen Übereinstimmungen.

Bild 3.4 zeigt, wie man mit einem solchen *Self-Join* die Datenbank auf doppelte Einträge – in diesem Fall nur Vor- und Nachnamen – hin untersucht (Text 1.9). Die Abfrage kombiniert hier in allen möglichen Variationen, so daß aus drei gleichen Namen sechs Datensatzeinträge werden. Die Anweisung DISTINCT sorgt dafür, daß nur Datensätze mit verschiedenen Daten zurückgegeben werden. Würde man auf die Anzeige der Nummern verzichten, dann würden erheblich

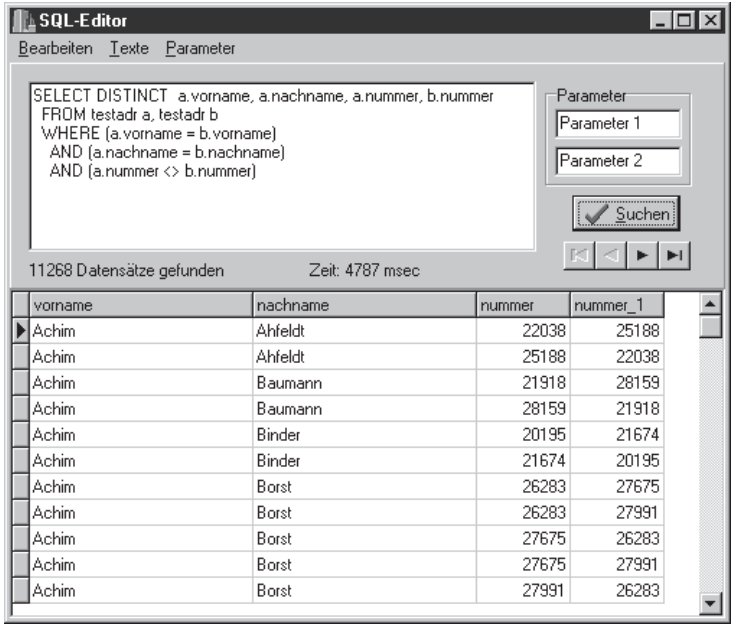


Bild 3.4: Anzeige doppelter Vor- und Nachnamen

weniger Datensätze zurückgegeben (bei meiner Testadressen-Datenbank 2866 statt 11 698).

Selektieren mit der WHERE-Anweisung

Die WHERE-Anweisung, die wir schon ein wenig verwendet haben, beschränkt die Rückgabe auf diejenigen Werte, welche den angegebenen Anforderungen genügen. Beispielsweise lassen sich alle Berliner Adressen aussortieren (Text 2.1):

```
SELECT * FROM testadr
  WHERE ort = "Berlin"
```

Die SELECT-Anweisung erlaubt sowohl AND(*und*)- als auch OR(*oder*)-Verknüpfungen. Soll nach einem Teilstring gesucht werden, verwendet man die LIKE-Anweisung (Text 2.2):

```
SELECT * FROM testadr
  WHERE (ort = "Berlin")
    AND (tel2 LIKE "0161%")
```

Diese Anweisung würde alle Berliner Adressen heraussuchen, die ein C-Netz-Telefon haben, deren *tel2* also mit *0161* beginnt. Das *%*-Zeichen ist ein sogenanntes »Joker-Zeichen« und ersetzt entweder kein, ein oder mehrere andere Zeichen. Es gibt auch das _-Zeichen, welches exakt ein anderes beliebiges Zeichen ersetzt (Text 2.3):

```
SELECT * FROM testadr
  WHERE ((ort = "Berlin") OR (ort = "Hamburg"))
    AND (tel2 LIKE "0161 / _5%")
```

Diese Anweisung würde alle Berliner und Hamburger zurückgeben, deren C-Netz-Durchwahl als zweites Zeichen eine *5* hat (suchen Sie mal lieber keinen Sinn in dieser Anweisung). Die zusätzliche Einklammerung der Sequenz *(ort = "Berlin") OR (ort = "Hamburg")* ist zwingend erforderlich, weil die Abfrage sonst folgendermaßen interpretiert würde:

```
SELECT * FROM testadr
  WHERE (ort = "Berlin") OR ((ort = "Hamburg")
    AND (tel2 LIKE "0161 / _5%"))
```

Eine Abfrage nach leeren Datensätzen läßt sich über das Schlüsselwort NULL erzielen. Hier wird allerdings nicht mit dem Gleichheitszeichen gearbeitet, sondern mit dem Schlüsselwort IS, bei der Suche nach Feldern mit Inhalt lautet die Sequenz dann IS NOT NULL. Das folgende Beispiel sucht nach Berliner Adressen mit Funktelefon (Text 2.4):

```
SELECT * FROM testadr
  WHERE (ort = "Berlin")
    AND (tel2 IS NOT NULL )
```

SQL erlaubt auch die Operatoren größer (>), kleiner (<), größer / gleich (>=) und kleiner / gleich (<=). Die folgende Abfrage sucht alle Adressen, die dem Postleitzahlenbreich *2* zugehören (Text 2.5):

```
SELECT * FROM testadr
  WHERE ((plz >= "20 000") AND (plz < "30 000"))
```

Statt zwei Vergleiche mit einer UND-Bedingung zu verknüpfen, kann man auch den BETWEEN-Operator verwenden (Text 2.6).

```
SELECT * FROM testadr
  WHERE plz BETWEEN "20 000" AND "29 999"
```

Die nächste Anweisung sucht nach den Adressen in den Postleitzahlgebieten 2 und 8. Hier wird statt mit größer / kleiner-Operatoren mit LIKE und einem String-Vergleich gearbeitet (Text 2.7):

```
SELECT * FROM testadr
  WHERE (plz LIKE "2%")
    OR (plz LIKE "8%")
```

Im Vergleich dazu die Abfrage, welche die gleiche Ergebnismenge zurückgibt, aber den BETWEEN-Operator verwendet (Text 2.8).

```
SELECT * FROM testadr
  WHERE (plz BETWEEN "20 000" AND "29 999")
    OR (plz BETWEEN "80 000" AND "89 999")
```

Werden viele Gleichheitsbedingungen mit einer ODER-Verknüpfung verbunden, so kann man dies auch übersichtlicher mit dem IN-Operator gestalten. In diesem Fall wird eine Person namens *Claudia* gesucht, die in einer deutschen Millionenstadt lebt (Text 2.9):

```
SELECT * FROM testadr
  WHERE (vorname = "Claudia")
    AND (ort IN ("Berlin", "Hamburg", "München"))
```

Die Aggregat-Funktionen

Zurück zu der Tabelle der offenen Posten. Nehmen wir einmal an, die Buchhaltung möchte wissen, wie hoch die Summe aller Außenstände ist. Nun ist es aufwendig und fehleranfällig, sich die Liste aller Außenstände anzeigen zu lassen und diese dann mit dem Taschenrechner zu addieren. Einfacher ist es, sich die Summe gleich von der Abfrage anzeigen zu lassen (Text 3.1):

```
SELECT SUM(betrag) FROM offenpo
```

Hier wird eine Tabelle mit nur einem Wert (und der Spaltenüberschrift) zurückgegeben, nämlich die Summe der Außenstände. Neben der Summe gibt es noch weitere Aggregatfunktionen (Text 3.2):

```
SELECT SUM(betrag), COUNT(betrag),
  MIN(betrag), MAX(betrag),
  AVG(betrag) FROM offenpo
```

Die Funktion COUNT zählt die Datensätze, MIN und MAX ermitteln den jeweils kleinsten und größten Wert, die Funktion AVG bildet den Durchschnitt. Mit einer WHERE-Klausel läßt sich die Datenmenge, über welche die Funktion gebildet wird, weiter einschränken. Im folgenden Beispiel (Bild 3.5) werden die Summe und die anderen Funktionen aller Außenstände mit Berliner Adressen ermittelt (Text 3.3).

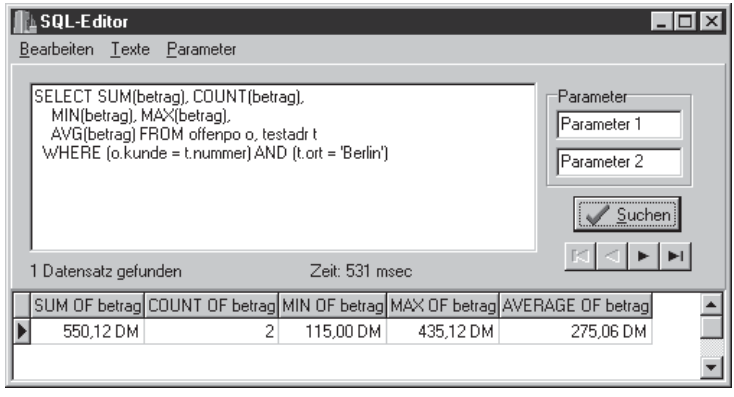


Bild 3.5: Aggregat-Funktion mit Equi-Join

Will man nun auf diese Weise die Außenstände nach Städten (oder wonach auch immer) sortieren, dann ist man ziemlich beschäftigt. Deshalb gibt es auch die Möglichkeit, die Datenmenge zu gruppieren und die Aggregatfunktionen über die jeweilige Gruppe zu bilden (Text 3.4):

```
SELECT t.ort, SUM(o.betrag), COUNT(o.betrag),
    MIN(o.betrag), MAX(o.betrag),
    AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.ort
```

Hier wird die Datenmenge nach Orten gruppiert und von den Außenständen der Kunden, die im jeweiligen Ort wohnen, werden die Summe und die übrigen Aggregatfunktionen gebildet. Da es sich um einen Equi-Join handelt, werden nur diejenigen Orte angezeigt, in denen säumige Kunden wohnen. Alle anderen Orte werden nicht angezeigt, genausowenig ein »leerer« Ort für die offenen Forderungen, deren Kundennummern sich nicht zuordnen lassen.

Um alle Orte anzuzeigen, auch wenn die Zahl der offenen Posten dort Null beträgt, wird ein RIGHT oder FULL OUTER JOIN benötigt. Bei letzterem wird dann auch ein Datensatz ohne Ort-Namen angezeigt, welcher die Außenstände aufsummiert, deren Kundennummern sich nicht zuordnen lassen.

```
SELECT t.ort, SUM(o.betrag), COUNT(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o FULL OUTER JOIN testadr t
  ON (o.kunde = t.nummer)
  GROUP BY t.ort
```

Für gewöhnlich werden alle Datensätze nach dem Wert in der ersten Spalte sortiert. Es wäre aber denkbar, daß eine andere Sortierreihenfolge gewünscht wird. Beispielsweise soll nach der Höhe der Außenstände sortiert werden (Text 3.5).

```
SELECT t.ort, SUM(o.betrag), COUNT(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.ort
  ORDER BY SUM(o.betrag)

/* funktioniert nicht */
```

Das Sortieren von Datensätzen geschieht mit der Anweisung ORDER BY, welche wir später noch ausführlicher behandeln werden. Das Problem in diesem Fall ist, daß weder die 16- noch die 32-Bit-BDE die Anweisung *ORDER BY SUM(o.betrag)* akzeptieren. (Die 16-Bit-BDE, die beispielsweise bei Delphi 1.0 zum Einsatz kommt, akzeptiert die Anweisung *ORDER BY o.betrag*.) Mit dem »Trick« einer Spaltenumbenennung kommt man hier jedoch problemlos ans Ziel (Text 3.6):

```
SELECT t.ort, COUNT(o.betrag) AS gesamtsumme, SUM(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.ort
  ORDER BY gesamtsumme
```

In vielen Firmen werden Forderungen, welche unter einer bestimmten Grenze liegen, nicht weiter verfolgt – der Aufwand wäre zu groß. Nehmen wir einmal an, diese Grenze läge bei 100,- DM, und nun soll eine Statistik der nach Orten gruppierten Außenstände erstellt werden, bei denen die Bagatellbeträge ignoriert werden. Auch diese Möglichkeit besteht, und zwar mit der HAVING-Klausel (Text 3.7):

```
SELECT t.ort, COUNT(o.betrag), SUM(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.ort HAVING SUM(o.betrag) >= 100
```

Bei der HAVING-Klausel wird die Gruppierung eingeschränkt wie mit der WHERE-Klausel die Abfrage; auch die Syntax ist dieselbe. Auf ähnliche Weise kann man feststellen, welche Kunden mehrere Rechnungen nicht bezahlt haben (Text 3.8):

```
SELECT t.nachname, t.vorname, COUNT(o.betrag), SUM(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.nachname, t.vorname HAVING COUNT(o.betrag) >= 2
```

Beachten Sie dabei, daß in die GROUP-Anweisung sowohl *t.nachname* als auch *t.vorname* aufgeführt werden müssen, sonst generiert die BDE eine Fehlermeldung. In der nächsten Anweisung wird eine kleine Statistik aus der Adressendatenbank erstellt. Zum einen wird nach Städten aufgelistet, wieviel Adressen es gibt und wie viele dort eine Funktelefonnummer haben. Als weitere Spalte wird dann der Prozentsatz der Funktelefone pro Adressen berechnet (Text 3.9):

```
SELECT t.ort, COUNT(t.nummer), COUNT(t.tel2),
     (COUNT(t.tel2) * 100) / COUNT(t.nummer)
  FROM testadr t
  GROUP BY t.ort
```

Sortieren der Einträge

Solange nichts anderes angegeben wird, werden die Datensätze in alphabetischer bzw. numerischer Reihenfolge aufsteigend bezüglich der ersten Spalte sortiert. Diese Reihenfolge läßt sich mit der ORDER BY-Anweisung verändern (Text 4.1):

```
SELECT * FROM testadr t
  ORDER BY t.nachname, t.vorname, t.ort
```

Hier werden die Einträge in aufsteigender alphabetischer Reihenfolge zunächst nach *t.nachname* sortiert. Bei gleichen Einträgen in *t.nachname* werden diese nach *t.vorname* sortiert, zuletzt nach *t.ort*. Eine absteigende Sortierung wird mit dem Schlüsselwort DESC erreicht (Text 4.2):

```
SELECT * FROM testadr t
  ORDER BY t.nachname, t.vorname DESC, t.ort
```

Hier wird bei gleichen *t.nachname* (welche aufsteigend sortiert werden) nach *t.vorname* absteigend sortiert; die Liste beginnt mit *Xaver Ackermann* statt mit *Achim Ackermann*. Für jede Spalte, nach der sortiert wird, kann bestimmt werden, ob sie aufsteigend (ASC, kann entfallen) oder absteigend (DESC) sortiert wird.

```
SELECT t.nummer, t.nachname || ", " || t.vorname AS Name
    FROM testadr t
  ORDER BY name
```

Hier werden die Spalten *t.nachname* und *t.vorname* zu einem Feld zusammengefaßt (Text 4.3). Diesem wird die Bezeichnung *name* zugewiesen, der gesamte String ist Grundlage für die Sortierung. Werden Felder, Rechenergebnisse oder zusammengefügte Strings mit AS umbenannt, dann erscheint diese Bezeichnung auch als Spaltenüberschrift im *DBGrid* (aus diesem Grund beginnt *Name* auch mit einem Großbuchstaben).

Die Möglichkeit der Umbenennung der Spaltenüberschriften ist im nächsten Beispiel besonders sinnvoll (Versuchen Sie es einmal ohne...) (Text 4.4):

```
SELECT o.datum, EXTRACT (DAY FROM o.datum) AS tag,
    EXTRACT (MONTH FROM o.datum) AS monat,
    EXTRACT (YEAR FROM o.datum) AS jahr
  FROM offenpo o
```

Hier werden der Tag, der Monat und das Jahr mit EXTRACT als Einzelwerte aus dem Datum ermittelt. Zuletzt wollen wir noch ein wenig rechnen, und zwar wieder mit der Tabelle der offenen Posten. Zunächst soll eine Mahngebühr von 5,- DM eingeführt werden, dann soll nicht nur das Datum der Rechnungsstellung angezeigt werden, sondern auch das Datum drei Wochen später, denn dann soll die Mahnung geschrieben werden (Text 4.5):

```
SELECT t.vorname, t.nachname, o.betrag AS rechnungssumme,
    o.betrag + 5 AS mahnsumme, o.datum, o.datum + 21 AS mahndatum
  FROM offenpo o, testadr t
  WHERE o.kunde = t.nummer
```

Nun wollen wir aus der Rechnungssumme auch noch den Nettobetrag berechnen (MWSt 15 %, wobei ich nicht weiß, ob das während der »Laufzeit« des Buches aktuell bleibt ...) (Text 4.6):

```
SELECT t.vorname, t.nachname, o.betrag AS rechnungssumme,
    o.betrag/1.15 AS nettosumme, o.datum, o.datum + 21 AS mahndatum
  FROM offenpo o LEFT OUTER JOIN testadr t
  ON o.kunde = t.nummer
```

Hier werden wiederum alle offenen Posten angezeigt (OUTER JOIN).

Unterabfragen

Bisweilen kommt es vor, daß man für eine Abfrage einen Wert benötigt, den man erst aus einer anderen Abfrage erhält. Hier ist es dann sinnvoll, eine Unterabfrage zu verwenden. Als Beispiel sollen hier die Kunden ermittelt werden, deren (einzelne) Außenstände über dem Durchschnitt liegen (Text 4.7):

```
SELECT t.vorname, t.nachname, o.betrag AS rechnungssumme
  FROM offenpo o LEFT OUTER JOIN testadr t
    ON o.kunde = t.nummer
  WHERE o.betrag >
    (SELECT AVG(o.betrag) FROM offenpo o)
```

Der Mittelwert der einzelnen Außenstände wird hier durch die Unterabfrage ermittelt; bei der Unterabfrage handelt es sich wieder um einen SELECT-Befehl mit der bekannten Syntax.

3.2.2 Erstellen von Tabellen

Für gewöhnlich wird man Tabellen, Indizes und ähnliches mit der Datenbankoberfläche erstellen. Bisweilen kann es aber sinnvoll sein, eine Tabelle aus den Komponenten *TTable* oder *TQuery* heraus anzulegen. Leider sind dabei einige wichtige Merkmale bei Desktop-Datenbanken nicht verfügbar, so daß diese Vorgehensweise schnell an ihre Grenzen stößt. Wir werden deshalb auf die *DDL* (*data definition language*) erst bei der Behandlung des *Local InterBase Servers* näher eingehen. Die Erstellung einer Tabelle erfolgt mit der SQL-Anweisung CREATE TABLE (Text 5.1):

```
CREATE TABLE "test_1.db"
  (nummer AUTOINC,
  vorname CHAR(20),
  nachname CHAR(20),
  straße CHAR(20),
  plz CHAR(6),
  ort CHAR(20),
  tel1 CHAR(20),
  tel2 CHAR(20),
```

```
  fax CHAR(20),
  PRIMARY KEY(nummer))

  /* Bearbeiten | SQL-Anweisung ausführen
```

Zunächst beachten Sie bitte, daß die SQL-Anweisung nicht mit dem Button *Suchen* gestartet werden darf, sondern es muß dafür der Menü-Punkt BEARBEITUNG | SQL-ANWEISUNG AUSFÜHREN verwendet werden. Im Listing sehen Sie an dieser Stelle auch, wie man Kommentare in SQL-Anweisungen einfügt.

Durch die Dateiendung *.db* wird die BDE angewiesen, eine Paradox-Tabelle zu erstellen. Die Nummer soll selbstinkrementierend automatisch vergeben werden, dafür wird der Spaltentyp AUTOINC verwendet. Alphanumerische Felder haben hier den Spaltentyp CHAR, die Länge muß dabei vorgegeben werden. Weitere häufig benötigte Feldtypen sind:

- *SMALLINT* (16 Bit) und *INTEGER* (32 Bit)
- *NUMERIC* (Gleitkommazahl)
- *DATE* (Datum)
- *MONEY* (Währung)

Näheres zu den Typen in der Online-Hilfe oder in den Kapiteln über den *Local InterBase Server*. Die Tabelle soll nun mit einem Datensatz versehen werden (Text 5.2):

```
INSERT INTO "test_1.db" (vorname, nachname, straße,
  plz, ort, tel1, tel2, fax) VALUES ("Ludwig", "Meier",
  "Mozartstraße 32", "10 001", "Berlin",
  "030 / 123 45 67", "", "030 / 123 45 68");

  /* Bearbeiten | SQL-Anweisung ausführen
```

Die Werte werden in der Reihenfolge in die Tabelle eingefügt, in der die Spalten vor dem Schlüsselwort VALUES aufgeführt sind. Vergessen Sie nicht, die Ausführung mit BEARBEITEN | SQL-ANWEISUNG AUSFÜHREN zu starten. Mit der bekannten Anweisung schaut man sich nun die Tabelle an (Text 5.3):

```
SELECT * FROM test_1
```

Zum Schluß soll die Tabelle wieder entfernt werden. Dazu verwendet man den Befehl DROP TABLE (Text 5.4):

```
DROP TABLE test_1

  /* Bearbeiten | SQL-Anweisung ausführen
```

3.3 Parameter

Die Verwendung von SQL-Anweisungen kann mit Hilfe von Parametern weiter vereinfacht werden. Nehmen wir einmal an, daß alle Datensätze gesucht werden sollten, deren Spalte *vorname* gleich dem ist, was der Anwender in *Edit1* eingibt, und deren Spalte *nachname* gleich dem ist, was in *Edit2* angegeben wurde. Die entsprechende Anweisung könnte man folgendermaßen konstruieren:

```
void __fastcall TForm1::ohneParameter1Click(TObject *Sender)
{
  Memo1->Clear();
  Memo1->Lines->Add("SELECT * FROM testadr");
  Memo1->Lines->Add("  WHERE (vorname = '" + Edit1->Text + "')");
  Memo1->Lines->Add("    AND(nachname = '" + Edit2->Text + "')");
}
```

Man kann das Problem aber auch mittels der Verwendung von Parametern lösen:

```
void __fastcall TForm1::mitParameter1Click(TObject *Sender)
{
  Memo1->Clear();
  Memo1->Lines->Add("SELECT * FROM testadr");
  Memo1->Lines->Add("  WHERE (vorname = :para1)");
  Memo1->Lines->Add("    AND(nachname = :para2)");
}

void __fastcall TForm1::Suchen1Click(TObject *Sender)
{
  AnsiString s;
  int t1, t2;
  try
  {
    Screen->Cursor = crHourGlass;
    Query1->SQL->Clear();
    Query1->SQL = Memo1->Lines;
    Query1->ParamByName("para1")->AsString = Edit1->Text;
    Query1->ParamByName("para2")->AsString = Edit2->Text;
    t1 = GetCurrentTime();
    Query1->Open();
    t2 = GetCurrentTime();
    s = ("%n", Query1->RowsAffected);
```

```
    Label1->Caption = s + " Zeilen bearbeitet";
    s = ("%n", t2 - t1);
    Label2->Caption = "Zeit: " + s + " msec";
    Screen->Cursor = crDefault;
  } //try
  catch(...)
  {
    Screen->Cursor = crDefault;
    MessageDlg("SQL-Anweisung fehlerhaft", mtWarning,
      TMsgDlgButtons() << mbOK,0);
  } //catch
} // TForm1::Suchen1Click
```

In der SQL-Anweisung werden die Parameter *para1* und *para2* durch Voranstellung eines Doppelpunktes definiert, diesen werden dann über die Eigenschaft *Params* oder die Methode *ParamByName* die Texte von *Edit1* bzw. *Edit2* zugewiesen. In der Eigenschaft *Params* müssen die Parameter in derselben Reihenfolge aufgeführt sein wie sie in der SQL-Anweisung definiert sind, etwas weniger fehleranfällig ist hier die Methode *ParamByName*, die mit den Parameternamen arbeitet. Bevor die Parameter zugewiesen werden, muß der Komponente *TQuery* die SQL-Anweisung zugewiesen werden (woran sonst sollte sie erkennen, welche Art von Parametern sie entgegennehmen soll?).

Über Parameter läßt sich auch eine Master-Detail-Beziehung zu einer anderen Tabelle herstellen. Nehmen wir einmal an, daß die *Table1/DataSource1*-Kombination uns die Personen der *testadr*-Tabelle anzeigt. Nun wüßten wir bei der Anzeige der Personen (etwa bei einem Neuauftrag) auch gerne, wie viele Rechnungen von diesen Personen noch nicht bezahlt worden sind. Es soll also in einem *DBGrid* angezeigt werden, welche Außenstände der betreffende Kunde bei der Firma hat. Bild 3.6 zeigt ein solches Fenster. (Es würde sich besser machen, wenn man die Kundenadresse über einzelne *DBEdit*-Felder und nicht über ein *DBGrid* anzeigen würde, aber hier kommt es momentan nicht auf Schönheit an.)

Das zweite *DBGrid* mit den offenen Posten wird über *Query1/DataSource2* mit Daten versorgt. Die Eigenschaft *SQL* von *Query1* lautet nun folgendermaßen:

```
SELECT * FROM offenpo
  WHERE kunde = :nummer
```

Nun wird der Parameter *nummer* nicht über die Eigenschaft *Params* zugewiesen (die bleibt in diesem Fall leer), sondern die Eigenschaft *DataSource* wird auf *DataSource1* gesetzt. Jedesmal, wenn in der Kundentabelle ein neuer Datensatz aktuell wird, wird *Query1* die Kundennummer neu zugewiesen, und *Query1* sucht sich anhand der Kundennummer dann die Außenstände dieser Person heraus. Der Parameter muß dabei den Namen der Spalte haben, aus der er den Verknüpfungswert – hier die Kundennummer – beziehen soll.

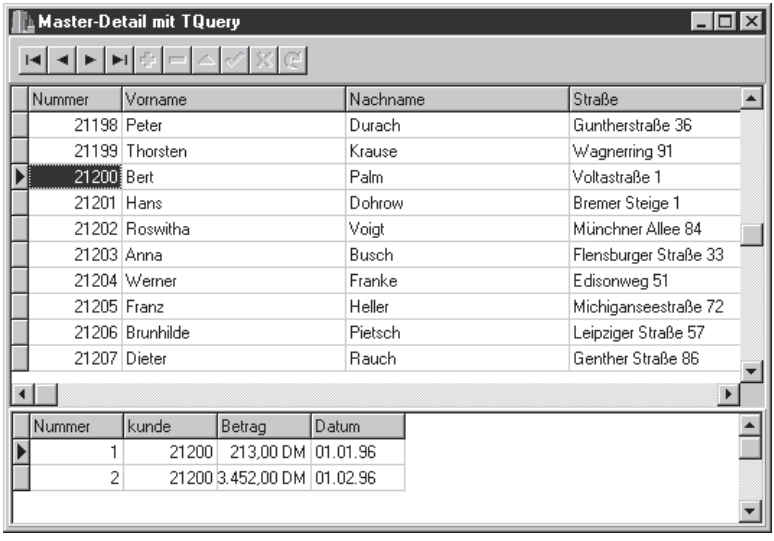


Bild 3.6: Master-Slave-Beziehung mit TQuery

Wenn zusätzlich noch die Summe der Außenstände des betreffenden Kunden angezeigt werden soll, dann wird eine zweite Abfrage benötigt, welche auf dieselbe Weise mit *DataSource1* verknüpft wird. Die SQL-Anweisung muß hier dann folgendermaßen lauten:

```
SELECT SUM(betrag) FROM offenpo
  WHERE kunde = :nummer
```

3.4 Referenz TQuery

Hier sollen nun die Eigenschaften, Methoden und Ereignisse von *TQuery* besprochen werden. Die Komponente *TQuery* ist – ebenso wie *TTable* – von *TDBDataSet* abgeleitet. Die Mehrzahl der Eigenschaften, Methoden und Ereignisse übernimmt sie daher von *TDataSet* und *TDBDataSet*. Die Referenz dieser beiden Komponenten erfolgte bereits in Kapitel 2 und soll hier nicht widerholt werden. Bei den geerbten Eigenschaften, Methoden und Ereignissen wird deshalb lediglich auf die jeweilige Klasse verwiesen.

Alle Eigenschaften und Methoden, die in *TComponent*, *TPersistent* oder *TObject* implementiert wurden, werden im folgenden nicht erwähnt.

3.4.1 Die veröffentlichten Eigenschaften von TQuery

- Active (*TDataSet*)
- AutoCalcFields (*TDataSet*)
- CachedUpdates (*TDataSet*)
- Constrained (*TQuery*)

  ```
  __property bool Constrained;
  ```

  Ist *Constrained* gleich *true*, dann dürfen bei Paradox- und dBase-Tabellen keine Datensätze eingefügt werden, welche nicht die Bedingungen der aktuellen SQL-Anweisung erfüllen. Angenommen, die SQL-Anweisung einer *TQuery*-Komponente würde folgenermaßen lauten:

  ```
  SELECT * FROM testadr
    WHERE ort = "Berlin"
  ```

  Die folgenden Anweisungen werden dann nicht ausgeführt, wenn *Constrained* gleich *true* ist:

  ```
  INSERT INTO testadr (nachname, ort)
    VALUES ('Meyer', 'Hamburg')

  UPDATE testadr
    SET ort = 'Frankfurt'
    WHERE nummer = 12345
  ```

- DatabaseName (*TDBDataSet*)
- DataSource (*TQuery*)

  ```
  __property Db::TDataSource DataSource;
  ```

  Gibt an, aus welcher Datenmenge die Parameter entnommen werden, die anderweitig nicht definiert sind. Die Parameternamen und die Spaltennamen müssen dabei übereinstimmen. Mit Hilfe der Eigenschaft *DataSource* kann eine aus *TQuery*-Komponenten bestehende Master-Detail-Verknüpfung erstellt werden.

- Filter (*TDataSet*)
- Filtered (*TDataSet*)
- FilterOptions (*TDataSet*)

- ParamCheck (*TQuery*)

  ```
  __property bool ParamCheck;
  ```

  Ist *ParamCheck* gleich *true*, dann werden alle Parameter gelöscht, wenn eine neue SQL-Anweisung zugewiesen wird. Per Voreinstellung ist *ParamCheck* gleich *true* und sollte in der Regel auf diesem Wert bleiben, weil ansonsten nicht sichergestellt ist, daß die Parameter zur SQL-Anweisung passen.

- Params (*TQuery*)

  ```
  __property TParams* Params
  ```

  Die Array-Eigenschaft *Params* enthält alle Parameter einer SQL-Anweisung. Die Indexzählung beginnt bei Null für den ersten in der SQL-Anweisung erwähnten Parameter. Gegeben sei folgende SQL-Anweisung:

  ```
  SELECT * FROM testadr
    WHERE (vorname = :vor)
      AND (nachname = :nach)
  ```

  Mit den folgenden Anweisungen kann man nun die Parameter setzen:

  ```
  Query1->Params[0]->AsString = "Yvonne";
  Query1->Params[1]->AsString = "Bach";
  ```

  Die Verwendung der Eigenschaft *Params* führt leicht zu Fehlern, wenn die SQL-Anweisung modifiziert und deshalb die Reihenfolge der Parameter verändert wird. Sicherer ist hier die Verwendung der Methode *ParamByName*.

- RequestLive (*TQuery*)

  ```
  __property bool RequestLive;
  ```

  Ist *RequestLive* – gemäß der Voreinstellung – gleich *false*, dann liefert die Komponente eine Ereignismenge, welche nicht editiert werden kann. Sollen jedoch Datensätze eingefügt, geändert oder gelöscht werden, muß die Eigenschaft *RequestLive* auf *true* gesetzt werden.

- SessionName (*TDBDataSet*)

- SQL (*TQuery*)

  ```
  __property Classes::TStrings* SQL;
  ```

  Die Eigenschaft *SQL* beinhaltet die SQL-Anweisung von *TQuery*. Es sind dabei nicht nur Abfragen erlaubt, sondern auch Datenänderungsbefehle (INSERT, UPDATE, DELETE) und Datendefinitionsbefehle (CREATE TABLE, DELETE TABLE, usw.).

  Wird die Eigenschaft *SQL* modifiziert, dann wird die Datenmenge automatisch geschlossen. Bei Abfragen muß sie anschließend mit *Open* geöffnet werden, alle anderen SQL-Anweisungen werden mit *ExecSQL* ausgeführt.

- UniDirectional (*TQuery*)

  ```
  __property bool UniDirectional;
  ```

  Legt fest, ob der BDE-Datencursor in beide Richtungen bewegt werden kann oder nicht; per Voreinstellung *false* und in der Regel ohne Bedeutung.
- UpdateMode (*TDBDataSet*)
- UpdateObject (*TDataSet*)

3.4.2 Die öffentlichen Eigenschaften von TQuery

- BOF (*TDataSet*, nur Lesen)
- Bookmark (*TDataSet*)
- CanModify (*TDataSet*, nur Lesen)
- Database (*TDataSet*, nur Lesen)
- DBHandle (*TDBDataSet*, nur Lesen)
- DBLocale (*TDBDataSet*, nur Lesen)
- DBSession (*TDBDataSet*, nur Lesen)
- DefaultFields (*TDataSet*, nur Lesen)
- Designer (*TDataSet*, nur Lesen)
- EOF (*TDataSet*, nur Lesen)
- ExpIndex (*TDataSet*, nur Lesen)
- FieldCount (*TDataSet*, nur Lesen)
- FieldDefs (*TDataSet*)
- Fields (*TDataSet*)
- FieldValues (*TDataSet*)
- Found (*TDataSet*, nur Lesen)
- Handle (*TDataSet*, nur Lesen)
- KeySize (*TDataSet*, nur Lesen)
- Local (*TQuery*, nur Lesen)

  ```
  __property bool Local;
  ```

  Ist *true*, wenn sich die Abfrage nur auf *Paradox*- oder *dBase*-Tabellen bezieht. Richtet sich die Abfrage an einen SQL-Server, ist *Local* gleich *false*.
- Locale (*TDataSet*, nur Lesen)

- Modified (*TDataSet*, nur Lesen)
- ParamCount (*TQuery*)

  ```
  __property unsigned short ParamCount;
  ```

  Die Eigenschaft *ParamCount* gibt die Anzahl der Parameter in der aktuellen Abfrage an.
- Prepared (*TQuery*)

  ```
  __property bool Prepared;
  ```

  Mit der Eigenschaft *Prepared* läßt sich feststellen, ob eine Abfrage vorbereitet ist. Um eine Abfrage vorzubereiten, kann die Eigenschaft *Prepared* auf *true* gesetzt werden, in der Regel wird man jedoch zu diesem Zweck die Methode *Prepare* aufrufen.
- RecordNo (*TDataSet*, nur Lesen)
- RecordCount (*TDataSet*, nur Lesen)
- RecordSize (*TDataSet*, nur Lesen)
- RowsAffected (*TQuery*, nur Lesen)

  ```
  __property int RowsAffected;
  ```

  Mit der Eigenschaft *RowsAffected* kann festgestellt werden, wie viele Datensätze bei der letzen SQL-Anweisung verarbeitet (eingefügt, aktualisiert oder gelöscht) wurden. Durch die Anzeige dieser Zahl kann der Anwender sehen, ob die von ihm formulierte SQL-Anweisung den gewünschten Erfolg hatte.

  ```
  AnsiString s;
  s = ("%n", Query1->RowsAffected);
  Label1->Caption = s + " Zeilen bearbeitet";
  ```

- SQLBinary (*TQuery*)

  ```
  __property char* SQLBinary;
  ```

  Die Eigenschaft *SQLBinary* zeigt auf einen binären Daten-Stream, mit dessen Hilfe *TQuery* mit der BDE kommuniziert. Diese Eigenschaft sollte nicht verwendet werden.
- State (*TDataSet*, nur Lesen)
- StmtHandle (*TQuery*, nur Lesen)

  ```
  __property Bde::hDBIStmt StmtHandle;
  ```

  Die Eigenschaft *StmtHandle* bezeichnet das Anweisungshandle der BDE. Diese Eigenschaft wird nur benötigt, wenn Funktionen der BDE-API direkt aufgerufen werden.

- Text (*TQuery*, nur Lesen)

  ```
  __property System::AnsiString Text;
  ```

  Die Eigenschaft *Text* enthält die vollständige SQL-Anweisung, so wie sie der BDE übergeben wird.

- UpdateRecordTypes (*TDataSet*)
- UpdatesPending (*TDataSet*, nur Lesen)

3.4.3 Die Methoden von TQuery

- ~TQuery (*TQuery*)

  ```
  __fastcall virtual ~TQuery(void);
  ```

  Der Destruktor *~TQuery* gibt den Speicher frei, der für die *TQuery*-Instanz reserviert gewesen ist. *~TQuery* sollte nicht direkt aufgerufen werden, statt dessen ist die Methode *Free* zu verwenden.

- ActiveBuffer (*TDataSet*)
- Append (*TDataSet*)
- AppendRecord (*TDataSet*)
- ApplyUpdates (*TDataSet*)
- Cancel (*TDataSet*)
- CancelUpdates (*TDataSet*)
- CheckBrowseMode (*TDataSet*)
- CheckOpen (*TDBDataSet*)
- ClearFields (*TDataSet*)
- Close (*TDataSet*)
- CommitUpdates (*TDataSet*)
- ControlsDisabled (*TDataSet*)
- CursorPosChanged (*TDataSet*)
- Delete (*TDataSet*)
- DisableControls (*TDataSet*)
- Edit (*TDataSet*)
- EnbleControls (*TDataSet*)

- ExecSQL (*TQuery*)

  ```
  void __fastcall ExecSQL(void);
  ```

  Mit der Methode *ExecSQL* werden SQL-Anweisungen zur Datenänderung (INSERT, UPDATE, DELETE) und zur Datendefinition (CREATE TABLE, DELETE TABLE, usw.) ausgeführt. Zum Öffnen von Datenmengen mit SELECT-Anweisungen muß die Methode *Open* verwendet werden (alternativ kann die Eigenschaft *Active* auf *true* gesetzt werden).

- FetchAll (*TDataSet*)
- FieldByName (*TDataSet*)
- FindField (*TDataSet*)
- FindFirst (*TDataSet*)
- FindLast (*TDataSet*)
- FindNext (*TDataSet*)
- FindPrior (*TDataSet*)
- First (*TDataSet*)
- FreeBookmark (*TDataSet*)
- GetBookmark (*TDataSet*)
- GetCurrentRecord (*TDataSet*)
- GetFieldList (*TDataSet*)
- GetFieldNames (*TDataSet*)
- GotoBookmark (*TDataSet*)
- Insert (*TDataSet*)
- InsertRecord (*TDataSet*)
- IsLinkedTo (*TDataSet*)
- Last (*TDataSet*)
- Locate (*TDataSet*)
- Lookup (*TDataSet*)
- MoveBy (*TDataSet*)
- Next (*TDataSet*)
- Open (*TDataSet*)
- ParamByName (*TQuery*)

```
TParam* __fastcall ParamByName(const System::AnsiString Value);
```

Auf die Parameterliste kann nicht nur über die Array-Eigenschaft *Params*, sondern auch mit der Methode *ParamByName* zugegriffen werden. Letztere verwendet nicht den Index, sondern den Namen des Parameters, und ist somit unanfällig für Umstellungen der SQL-Anweisung.

```
SELECT * FROM testadr
  WHERE (vorname = :vor)
    AND (nachname = :nach)
```

Für die oben aufgeführte SQL-Anweisung würde man die Parameter folgendermaßen setzen:

```
Query1->ParamByName("vor")->AsString = "Yvonne";
Query1->ParamByName("nach")->AsString = "Bach";
```

- Post (*TDataSet*)
- Prepare (*TQuery*)

  ```
  void __fastcall Prepare(void);
  ```

  Mit der Methode *Prepare* kann eine Abfrage an die BDE oder an den Server übermittelt werden, damit diese vor der Ausführung optimiert wird.
- Prior (*TDataSet*)
- Refresh (*TDataSet*)

  Anmerkung: Als Methode von *TQuery* aktualisiert *Refresh* die Abfrage nicht. Damit die aktuellsten Daten angezeigt werden, muß die Abfrage geschlossen und wieder geöffnet werden.

  ```
  Query1->Close();
  Query1->Open();
  ```
- Resync (*TDataSet*)
- RevertRecord (*TDataSet*)
- SetFields (*TDataSet*)
- TQuery (*TQuery*)

  ```
  __fastcall virtual TQuery(Classes::TComponent* AOwner);
  ```

  Der Konstruktor *TQuery* erzeugt eine neue *TQuery*-Instanz.
- UnPrepare (*TQuery*)

  ```
  void __fastcall UnPrepare(void);
  ```

  Hebt die Vorbereitung einer Abfrage auf.

- UpdateCursorPos (*TDataSet*)
- UpdateRecord (*TDataSet*)
- UpdateStatus (*TDataSet*)

3.4.4 Die Ereignisse von TQuery

- AfterCancel (*TDataSet*)
- AfterClose (*TDataSet*)
- AfterDelete (*TDataSet*)
- AfterEdit (*TDataSet*)
- AfterInsert (*TDataSet*)
- AfterOpen (*TDataSet*)
- AfterPost (*TDataSet*)
- AfterScroll (*TDataSet*)
- BeforeCancel (*TDataSet*)
- BeforeClose (*TDataSet*)
- BeforeDelete (*TDataSet*)
- BeforeEdit (*TDataSet*)
- BeforeInsert (*TDataSet*)
- BeforeOpen (*TDataSet*)
- BeforePost (*TDataSet*)
- BeforeScroll (*TDataSet*)
- OnCalcFields (*TDataSet*)
- OnDeleteError (*TDataSet*)
- OnEditError (*TDataSet*)
- OnFilterRecord (*TDataSet*)
- OnNewRecord (*TDataSet*)
- OnPostError (*TDataSet*)
- OnServerYield (*TDataSet*)
- OnUpdateError (*TDataSet*)
- OnUpdateRecord (*TDataSet*)

3.5 TTable oder TQuery?

Ob nun *TTable* oder *TQuery* eingesetzt werden soll, hängt vor allem vom beabsichtigten Einsatzzweck ab:

- Soll eine Verbindung mit allen Datensätzen einer einzelnen Tabelle hergestellt werden, dann kann sowohl *TTable* als auch *TQuery* (*SELECT * FROM testadr, RequestLive := true*) verwendet werden; bei der Ausführungszeit habe ich auch keine Unterschiede gemessen, solange die Eigenschaft *RequestLive* gleich *true* ist.
- Sollen die verbundenen Datensätze beschränkt werden, dann ist *TQuery* flexibler.
- Abfragen über mehrere Tabellen sind ohnehin nur mit *TQuery* möglich.
- Bei der Verwendung von Aggregatfunktionen ist *TQuery* vorzuziehen, die meisten Funktionen sind mit *TTable* gar nicht möglich.
- Soll es dem Anwender ermöglicht werden, eigene Abfragen zu erstellen, dann kann mit *TQuery* auf eine standardisierte und verbreitete Abfragesprache zurückgegriffen werden.

4 Datensteuerungskomponenten

Der Name *Datensteuerung* für diese Komponentengruppe ist nicht sonderlich gut gewählt, denn mit den dort aufgeführten Komponenten wird nicht gesteuert, sondern angezeigt und editiert – vom *TDBNavigator* einmal abgesehen. Da aber C++Builder diesen Begriff so verwendet, soll er hier beibehalten werden.

Nachfolgend sollen die Komponenten dieser Gruppe mit ihren wichtigsten Eigenschaften, Methoden und Ereignissen vorgestellt werden. Eine vollständige Referenz der Komponenten soll hier allerdings aus Platzgründen unterbleiben.

DataSource und DataField

Vorneweg zwei Eigenschaften, welche in (nahezu) allen Datensteuerungs-Komponenten anzutreffen sind:

Mit der Eigenschaft *DataSource* legt man fest, von welcher *DataSource*-Komponente die Daten bezogen werden sollen. Diese Eigenschaft kann zur Entwurfs- und zur Laufzeit festgelegt bzw. verändert werden. Bei der Komponente *TDBNavigator* wird dadurch bestimmt, welche *DataSource* damit gesteuert werden soll. (Es ist möglich, eine *DataSource* von zwei *DBNavigator*-Komponenten steuern zu lassen, macht aber in der Regel keinen Sinn. Der umgekehrte Weg (ein Navigator steuert zwei Datenquellen) ist nicht (direkt) möglich, eine Synchronisation ist aber immerhin mit entsprechenden Anweisungen realisierbar, siehe Beispielprogramme.

Die Eigenschaft *DataField* bestimmt, aus welcher Tabellenspalte die Komponente ihre Daten beziehen soll. Es ist darauf zu achten, daß die Spalte einen Typ verwendet, welcher von der betreffenden Komponente auch angezeigt werden kann. Bei den Tabellen-Komponenten und beim Navigator gibt es diese Eigenschaft nicht, sie würde hier auch keinen Sinn machen.

4.1 Anzeige von Text

In diesem Kapitel sollen die Komponenten behandelt werden, welche zur Anzeige von Text bestimmt sind. Ausgenommen davon sind die Tabellen-Komponenten und die List- und Combo-Boxen, welchen eigene Kapitel gewidmet sind.

4.1.1 Die Komponente TDBText

Bei der Komponente *TDBText* handelt es sich um ein *Label*, welches die Eigenschaft besitzt, seinen Text entsprechend einer *TDataSource*-Komponente zu ändern.

Wenn es in der Online-Hilfe heißt, *die Komponente TDBText ist ein datensensitives Steuerelement*, dann ist dies nicht ganz korrekt, weil diese Komponente zwar datensensitiv ist, aber nichts steuert. Es besteht noch nicht einmal die Möglichkeit, den Text direkt zu ändern. Anweisungen wie

```
DBEdit1->Text = "Test";
DBEdit1->Caption = "Test";
```

scheitern schon allein deswegen, weil es weder die Eigenschaft *Text* noch die Eigenschaft *Caption* überhaupt gibt. Es ist somit auch nicht möglich, den Text der Komponente *TDBText* auszulesen, was meist auch überhaupt nicht notwendig ist, weil man direkt auf das Feld der entsprechenden Tabelle zugreifen kann:

```
AnsiString s = Table1->FieldByName("Namen")->AsString;
```

Nun kann es allerdings vorkommen, daß man die Eigenschaft *DataSource* zur Laufzeit ändert, beispielsweise, weil man nur einen Teil der Datensätze anzeigen möchte und deshalb als *DataSource* vorübergehend eine *TQuery*-Komponente verwendet.

In diesem Fall muß man dafür sorgen, daß immer das Feld der richtigen *DataSet*-Komponente verwendet wird:

```
s = DBText1->DataSource->DataSet->FieldByName("Namen")->AsString;
```

Des weiteren ist darauf zu achten, daß der benötigte Feld-Name (in diesem Fall *Namen*) auch in der neuen *DataSet* enthalten ist (und den gewünschten Inhalt hat). Wird dabei lediglich einmal mit *TTable* und einmal mit *TQuery* auf dieselbe Tabelle zugegriffen, dann treten hier sicher keine Probleme auf. Wird dagegen auf andere Tabellen zugegriffen, dann empfiehlt es sich, an dieser Stelle besonders aufmerksam zu sein.

Wie beim Label kann auch hier die Eigenschaft *Transparent* auf *true* gesetzt werden, um darunterliegende Elemente (Bitmaps beispielsweise) nicht durch die Hintergrundfarbe (Eigenschaft: *Color*, Farbe der Schrift: *Font.Color*) zu verdekken. Mit der Eigenschaft *WordWrap* kann ein Zeilenumbruch veranlaßt werden. Wenn die Eigenschaft *Autosize* auf *false* gestellt wird, dann wird immer umgebrochen.

Die Komponente *TDBText* kann ohne weitere Umwandlungen Text, Zahlen und Daten (Datums-Angaben) anzeigen. Bei booleschen Feldern werden die Wörter *wahr* oder *falsch* ausgegeben.

4.1.2 Die Komponente TDBEdit

Mit der Komponente *TDBEdit* können nicht nur Daten angezeigt, sondern auch vom Anwender Eingaben gemacht werden, die dann in die Datenbank übernommen werden (können). Es ist auch möglich, daß das Programm auf *TDBEdit* zugreift, beispielsweise um Ergebnisse von Berechnungen anzuzeigen und diese dann in die Datenbank zu übernehmen. Folgende Anweisungen sind also möglich:

```
s = DBEdit1->Text;
DBEdit1->Text = s;
```

Die Eigenschaft *Text* ist allerdings nur zur Laufzeit verfügbar. Die Komponente *TDBEdit* kann ohne weitere Umwandlungen Text, Zahlen und Daten (Datums-Angaben) anzeigen. Bei booleschen Feldern werden die Wörter *wahr* oder *falsch* ausgegeben.

Sollen (vorübergehend) nur Daten ausgegeben werden, dann ist die Eigenschaft *ReadOnly* auf *true* zu setzen. Sollen Ein- und Ausgaben nicht auf dem Bildschirm erscheinen, dann kann *PasswordChar* ein Zeichen zugewiesen werden, welches dann statt der ein- bzw. ausgegebenen Zeichen auf dem Bildschirm erscheint.

Wird die Eigenschaft *AutoSelect* auf *true* gestellt, dann wird jedesmal, wenn per *Tab*-Taste zu diesem Feld gewechselt wird, der gesamte Text markiert. Die Methode *SelectAll* zeitigt dasselbe Ergebnis. Die Methode *CutToClipboard* entfernt den markierten Teil des Textes und gibt ihn in die Zwischenablage, *CopyToClipboard* kopiert ihn dorthin. Mit *PasteFromClipboard* kann der Inhalt der Zwischenablage in den Text eingefügt werden. Der markierte Text – das muß nicht der ganze Text des *DBEdit*s sein – ist in der Eigenschaft *SelText* enthalten.

Bisweilen muß vermieden werden, daß durch uneinheitliche Groß- und Kleinschreibung Fehler entstehen. Die Eigenschaft *CharCase* erlaubt es, optional nur Großbuchstaben (*ecUpperCase*) oder nur Kleinbuchstaben (*ecLowerCase*) anzuzeigen.

4.1.3 Die Komponente TDBMemo

Im Gegensatz zu *TDBEdit* können mit *TDBMemo* auch Memo-Felder angezeigt werden, deren Text über mehrere Zeilen umgebrochen wird. Diese Komponente eignet sich auch zur Anzeige der üblichen Zeichen-, Zahlen- und Datumfelder, boolesche Felder werden wie gewohnt behandelt (*wahr* oder *falsch*). Auf den Inhalt kann – wie bei *TDBEdit* – mit der Eigenschaft *Text* zugegriffen werden.

Ein Zugriff auf die Daten ist aber auch mit der Eigenschaft *Lines* vom Typ *TStrings* möglich. Der Schreibzugriff sieht dabei wie folgt aus:

```
DBMemo1->Clear();
DBMemo1->Lines>Strings[0] = "Zeile0";
DBMemo1->Lines->Add("Zeile1");
DBMemo1->Lines->Add("Zeile2");
```

Ein Lesezugriff gestaltet sich folgendermaßen:

```
s1 = DBMemo1->Lines->Strings[0];
s2 = DBMemo1->Lines->Strings[1];
s3 = DBMemo1->Lines->Strings[2];
```

Bisweilen werden in einem Memo größere Texte untergebracht, weshalb das Anzeigen des Memos relativ viel Zeit benötigt, was sich insbesondere beim Scrollen des Textes auf langsamen Rechnern bemerkbar macht. Diesem Effekt kann man mit dem Setzen der Eigenschaft *AutoDisplay* auf *false* begegnen, welche dafür sorgt, daß für gewöhnlich ein leeres Memo-Feld (Mit dem Text *(Memo)*) angezeigt wird. Sobald aber der Anwender einen Doppelklick auf das Memofeld ausführt oder die Methode *LoadMemo* ausgeführt wird, erscheint der aktuelle Memo-Text auf dem Bildschirm, solange bis der Datensatz gewechselt wird.

Für gewöhnlich wird die *Tab*-Taste bei Windows dazu verwendet, um zwischen den einzelnen Dialog-Elementen zu wechseln, bei Textverarbeitungen dient sie jedoch zur Formatierung des Textes. Deshalb existiert bei *TDBMemo* die Eigenschaft *WantTabs*; ist sie *false*, dann wird die *Tab*-Taste auch in *TDBMemo* zum Wechseln zum nächsten Dialog-Element verwendet. Ist sie dagegen *true*, dann wird die *Tab*-Taste zum Formatieren verwendet, und man kann damit zwar ein Memo anspringen, es aber mit dieser Taste nicht mehr verlassen.

Mit den Methoden *CutToClipboard*, *CopyToClipboard* und *PasteFromClipboard* kann die Komponente auf die Zwischenablage zugreifen.

4.1.5 Die Komponente TDBMaskEdit

Die Komponente *TDBMaskEdit* gibt es in C++Builder nicht. Dies ist auch gar nicht notwendig, da die entsprechende Eingabemaske dem einzelnen Tabellenfeld zugewiesen wird. Ein entsprechender Aufruf kann zur Laufzeit erfolgen:

```
Table1->FieldByName("Datum")->EditMask "= "!90/90/00;1;_";
```

Bei der Vorgehensweise nach oben gezeigtem Listing wird die Eigenschaft *EditMask* des entsprechenden Feldobjekts der Tabelle geändert. Wird dieses Feld auch noch an anderer Stelle angezeigt, beispielsweise in einem *DBGrid*, dann wird auch hier die gesetzte Eingabemaske verwendet.

4.2 Gitterelemente

Zur tabellenartigen Anzeige von Daten gibt es die Komponenten *TDBGrid* und *TDBCtrlGrid*. Des weiteren werden wir im Laufe dieses Kapitels eine eigene Komponente namens *TDBBGrid* erstellen.

4.2.1 Die Komponente TDBGrid

Zur Anzeige von Daten in einer herkömmlichen Tabelle dient die Komponente *TDBGrid*. Mit ihr können Zahlen-, String-, Datums- und Währungsfelder direkt angezeigt werden. Bei booleschen Feldern wird *wahr* oder *falsch* ausgegeben.

Wie Bild 4.1 zeigt, ist die Anzeige von Memos und (erwartungsgemäß) von Bildern nicht möglich, was wir später ändern werden.

Zunächst wollen wir die entsprechenden Spalten lediglich aus der Tabelle entfernen, stören sie doch das Erscheinungsbild der Tabelle erheblich. Mit einem Doppelklick auf die Komponente öffnen wir den Spalteneditor, siehe Bild 4.2. Hier können dann nicht nur einzelne Spalten entfernt werden, es ist auch möglich, Eigenschaften wie Breite, (Hintergrund-) Farbe oder Schrift für einzelne Spalten oder einzelne Überschriften einzustellen.

Wenn Ihnen die Überschriften der Spalten nicht gefallen sollten, dann können Sie dies unter TITEL-EIGENSCHAFTEN | BESCHRIFTUNG ändern. Des weiteren ist es möglich, eine Auswahlliste zu definieren, damit die jeweils ausgewählte Zelle dieser Spalte wie eine ComboBox reagiert. Die Einstellung *Schalterstil* sollte man dabei

Form1

Species No	Category	Common_Name	Species Name	Length (cm)	Length_In	Notes	Graphic
90020	Triggerfish	Clown Triggerfish	Ballistoides conspicillum	50	0393700787	(Memo)	(Graphic)
90030	Snapper	Red Emperor	Lutjanus sebae	60	0472440945	(Memo)	(Graphic)
90050	Wrasse	Giant Maori Wrasse	Cheilinus undulatus	229	4803149606	(Memo)	(Graphic)
90070	Angelfish	Blue Angelfish	Pomacanthus nauarchus	30	0236220472	(Memo)	(Graphic)
90080	Cod	Lunartail Rockcod	Variola louti	80	6062992126	(Memo)	(Graphic)
90090	Scorpionfish	Firefish	Pterois volitans	38	6239212598	(Memo)	(Graphic)
90100	Butterflyfish	Ornate Butterflyfish	Chaetodon Ornatissimus	19	1496062992	(Memo)	(Graphic)
90110	Shark	Swell Shark	Cephaloscyllium ventriosum	102	4803149606	(Memo)	(Graphic)
90120	Ray	Bat Ray	Myliobatis californica	56	2440944882	(Memo)	(Graphic)
90130	Eel	California Moray	Gymnothorax mordax	150	1181102362	(Memo)	(Graphic)
90140	Cod	Lingcod	Ophiodon elongatus	150	1181102362	(Memo)	(Graphic)
90150	Sculpin	Cabezon	Scorpaenichthys marmoratus	99	3779527559	(Memo)	(Graphic)
90160	Spadefish	Atlantic Spadefish	Chaetodiperus faber	90	0708661417	(Memo)	(Graphic)
90170	Shark	Nurse Shark	Ginglymostoma cirratum	400	8031496063	(Memo)	(Graphic)
90180	Ray	Spotted Eagle Ray	Aetobatus narinari	200	0157480315	(Memo)	(Graphic)
90190	Snapper	Yellowtail Snapper	Ocyurus chrysurus	75	5590551181	(Memo)	(Graphic)

Bild 4.1: Memos und Bilder können nicht mit TDBGrid angezeigt werden

auf *cbsAuto* lassen, mit der Eigenschaft *Zeilenanzahl* wird festgelegt, wie viele Zeilen in der heruntergeklappten Liste zu sehen sind, in Bild 4.3 sind es drei.

Sind dabei mehr Einträge in der Auswahlliste als Zeilen vorgegeben wurden, dann wird der Liste ein vertikaler Scroll-Balken hinzugefügt. Der Spalten-Editor des *TDBGrid* erlaubt nette Spielereien; man muß bisweilen jedoch alle Spalten löschen, wenn man die dazugehörige Tabelle wechselt.

Diese Einstellungen kann man auch zur Laufzeit vornehmen, dafür steht die Eigenschaft *Columns* zur Verfügung:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  DBGrid1->Columns->Items[0]->Color = clLime;
  DBGrid1->Columns->Items[0]->Font->Color = clRed;
  DBGrid1->Columns->Items[0]->PickList->Clear();
  DBGrid1->Columns->Items[0]->PickList->Add("Eintrag 1");
  DBGrid1->Columns->Items[0]->PickList->Add("Eintrag 2");
  DBGrid1->Columns->Items[0]->PickList->Add("Eintrag 3");
  DBGrid1->Columns->Items[0]->PickList->Add("Eintrag 4");
  DBGrid1->Columns->Items[0]->DropDownRows = 4;
  DBGrid1->Columns->Items[0]->Title->Caption = "Neuer Titel";
  DBGrid1->Columns->Items[0]->Title->Color = clRed;
  DBGrid1->Columns->Items[0]->Title->Font->Size = 24;
}
```

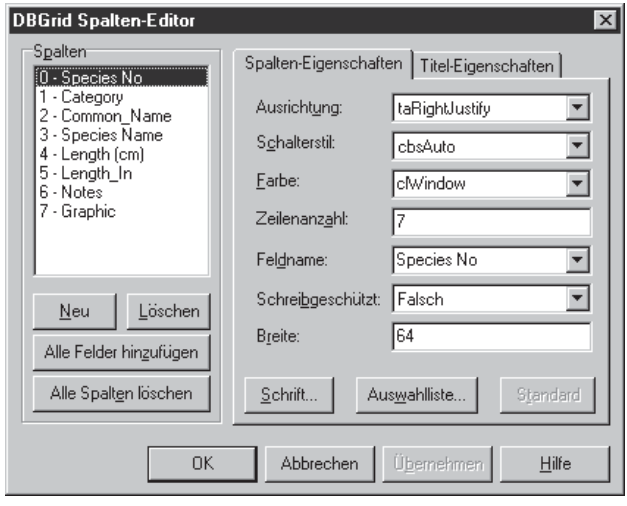


Bild 4.2: Der Spalten-Editor

Das Listing zeigt, wie man einige wesentliche Eigenschaften der ersten Spalte (Items, die Zählung beginnt bei Null) von *DBGrid1* ändert. Die Eigenschaft *Color* setzt die Hintergrundfarbe der Spalte, mit *Font* wird die Schriftart bestimmt, wobei hier nur die Farbe geändert wird. Selbstverständlich können auch die anderen Eigenschaften von *TFont* geändert werden.

Die Auswahlliste befindet sich in der Eigenschaft *PickList*, welche vom Typ *TStrings* ist. Hier werden in der Liste zunächst alte Einträge gelöscht, danach werden vier Einträge hinzugefügt. Die Zahl der Reihen dieser *DropDown*-Liste wird auf vier gesetzt.

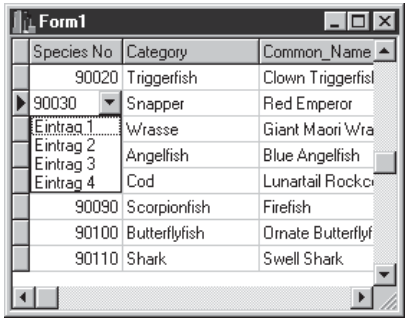


Bild 4.3: Eine Auswahlliste in TDBGrid

Festlegungen bezüglich der Spaltenüberschrift werden in der Eigenschaft *Title* angegeben. Hier interessieren insbesondere die Eigenschaften *Caption* (damit wird der Text der Spaltenüberschrift geändert), *Color* (Hintergrundfarbe) und *Font* (Schrift).

Noch ein Wort zur Eigenschaft *ButtonStyle* (im Spalten-Editor *Schalter-Stil*): Diese läßt sich auch auf den Wert *cbsEllipsis* setzen, dann erhält man die Schaltflächen mit den drei schwarzen Punkten, wie man sie beispielsweise aus dem Objektinspektor kennt. Wird diese Schaltfläche dann angeklickt, dann wird keine Auswahlliste angezeigt, sondern das Ereignis *OnEditButtonClick* ausgelöst; dazu soll nun ein kleines Demo-Programm erstellt werden:

Aus einer Tabelle Bitmaps anzeigen

Wie bereits erwähnt wurde, können in *TDBGrid* keine Bitmaps angezeigt werden. Wir wollen nun mit Hilfe der gerade erwähnten Schaltfläche ein dafür vorgesehenes Fenster starten, welches die Bitmaps nicht nur anzeigen, sondern auch in die Zwischenablage ausschneiden und kopieren sowie von dort aus einfügen kann. Zudem besteht die Möglichkeit, sie als Datei zu laden und zu speichern.

Zunächst wird das Hauptformular mit einem *Panel*, einem *DBNavigator* und einem *DBGrid* erstellt. Die Komponenten *TTable* und *TDataSource* werden in einem Daten-Modul untergebracht. Als Tabelle soll hier *Biolife.db* (Alias *dbdemos*) verwendet werden.

Im Spalten-Editor wird für die Spalte *Graphic* der Eigenschaft *Schalter-Stil* der Wert *cbs.Ellipsis* zugewiesen. Die Ereignisbehandlungsprozedur von *OnEditButtonClick* startet dann ein weiteres Formular namens *Form2*.

Das Formular *Form2* enthält ein Menü (die Menüpunkte entnehmen Sie bitte dem nachfolgenden Listing) und die Komponente *TDBImage*, deren Eigenschaften *DataSource* auf *Form1->DataSource1* und *DataField* auf *Graphic* gesetzt werden. *DBImage1* zeigt nun das entsprechende Bild an.

Zum Öffnen und Speichern sollen die Standard-Dialoge verwendet werden, es werden also je ein *Open*- und ein *SaveDialog* eingefügt und für Bitmap-Dateien (**.bmp*) eingerichtet. Für die Datei- und Zwischenablage-Operationen müssen folgende Prozeduren erstellt werden:

```
void __fastcall TForm2::Speichern1Click(TObject *Sender)
{
  if(SaveDialog1->Execute())
  {
    if(FileExists(SaveDialog1->FileName))
    {
      int i = Application->MessageBox
        ("Datei existiert bereits", "Warnung", MB_YESNOCANCEL);
      if(i == 6)
        DBImage1->Picture->SaveToFile(SaveDialog1->FileName);
    } // if(FileExists(SaveDialog1->FileName))
    else
      DBImage1->Picture->SaveToFile(SaveDialog1->FileName);
  } // if(SaveDialog1->Execute())
} // TForm2::Speichern1Click

void __fastcall TForm2::ffnen1Click(TObject *Sender)
{
  if(OpenDialog1->Execute())
    DBImage1->Picture->LoadFromFile(OpenDialog1->FileName);
}
```

Beim Speichern wie beim Öffnen wird zunächst der jeweilige Standard-Dialog ausgeführt, damit der Anwender Dateinamen und Pfad auswählen kann. Beim Speichern wird dann sicherheitshalber noch geprüft, ob es den Dateinamen schon

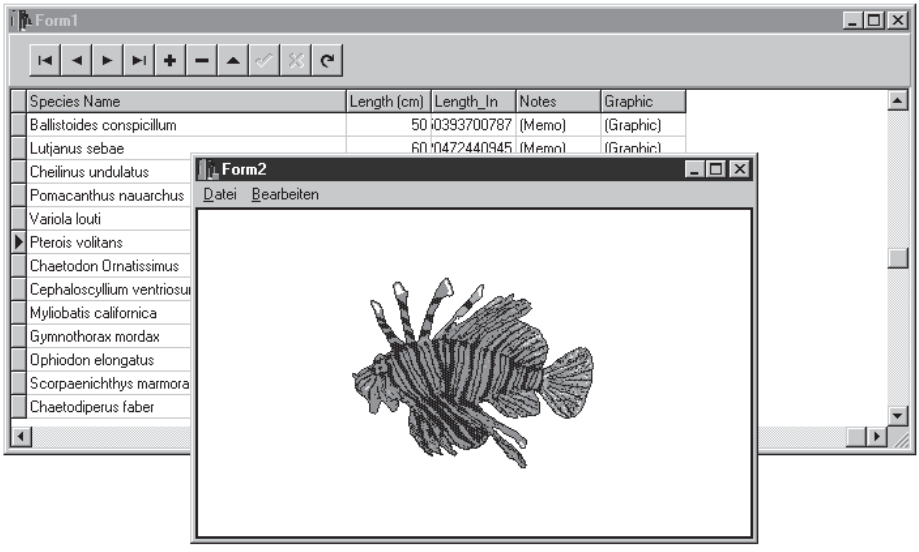


Bild 4.4: Starten eines weiteren Fensters mit der Schaltfläche

gibt, dem Anwender wird in diesem Fall die Möglichkeit gegeben, den Vorgang abzubrechen und so das Überschreiben dieser Datei zu vermeiden. Zum Speichern und Laden werden dann die *Picture*-Methoden *SaveToFile* und *LoadFromFile* verwendet.

```
void __fastcall TForm2::Ausschneiden1Click(TObject *Sender)
{
  DBImage1->CutToClipboard();
}

void __fastcall TForm2::Kopieren1Click(TObject *Sender)
{
  DBImage1->CopyToClipboard();
}

void __fastcall TForm2::Einfgen1Click(TObject *Sender)
{
  DBImage1->PasteFromClipboard();
}
```

Die Prozeduren für den Zugriff auf die Zwischenablage verwenden direkt die entsprechenden Methoden von *TDBImage*.

4.2.2 Die Komponente TDBBGrid

Bei *TDBBGrid* (*DataBaseBetterGrid*) handelt es sich um die gewohnte *TDBGrid*-Komponente, die aber vom Autor ein wenig verbessert worden ist. Wie schon vorhin festgestellt, lassen sich Memos und Bilder nicht im Datengrid anzeigen.

Dieses Verhalten wollen wir der Komponente nun ganz schnell abgewöhnen. Dazu wird mit DATEI | NEUE KOMPONENTE das Dialogfenster zum Erstellen einer neuen Komponente aufgerufen. Als *Klassennamen* geben Sie *TDBBGrid* und als *Vorfahrtyp TDBGrid* ein. Die Komponente soll dann auf der Seite *Datensteuerung* installiert werden.

C++Builder erstellt dann ein ein Grundgerüst, welches sich zu diesem Zeitpunkt schon fehlerfrei kompilieren lassen würde, aber noch keine über *TDBGrid* hinausgehende Funktionen besitzt. Zunächst wird nun die Header-Datei folgendermaßen ergänzt:

```
class TDBBGrid : public TDBGrid
{
private:
  bool FAutoDisp_Image;
  bool FAutoDisp_Memo;
  TPicture* FPicture;
  TStringList* FStringList;
protected:
public:
  __fastcall virtual TDBBGrid(TComponent* Owner);
  __fastcall virtual ~TDBBGrid(void);
  virtual void __fastcall DrawDataCell(const TRect &Rect,
      TField *Field, TGridDrawState State);
__published:
  __property bool AutoDisp_Image = {read=FAutoDisp_Image,
      write=FAutoDisp_Image, default = true};
  __property bool AutoDisp_Memo = {read=FAutoDisp_Memo,
      write=FAutoDisp_Memo, default = true};
};
```

Zunächst werden zwei Felder benötigt, mit deren Hilfe man wählen kann, ob Memos und Graphiken angezeigt werden sollen oder nicht; da die Anzeige zeitaufwendig ist, kann es durchaus Gründe geben, dies nicht zu tun.

Gemäß den Grundsätzen der objektorientierten Programmierung greifen veröffentlichte Eigenschaften (*property*) auf private Felder zu. Zudem benötigt man einen Konstruktor für die *default*-Einstellungen und zum Erstellen der Objekte.

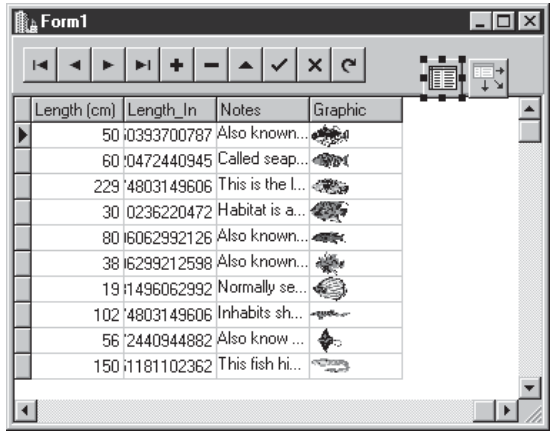


Bild 4.5: Grid mit Bildern und Memos

Die eigentliche Anzeige wird an das Ereignis *DrawDataCell* geknüpft, welches immer dann aufgerufen wird, wenn eine Zelle gezeichnet wird.

Nun zur eigentlichen Quelltextdatei: Im Konstruktor werden die Felder *FAutoDisp_Image* und *FAutoDisp_Memo* auf *true* gesetzt, zudem müssen die Objekte *FPicture* und *FStringList* erzeugt werden – diese sind dann im Destruktor zu beseitigen.

```
__fastcall TDBBGrid::TDBBGrid(TComponent* Owner): TDBGrid(Owner)
{
  FAutoDisp_Image = true;
  FAutoDisp_Memo = true;
  FPicture = new TPicture();
  FStringList = new TStringList();
}

__fastcall TDBBGrid::~TDBBGrid(void)
{
  delete FPicture;
  delete FStringList;
}
```

Die Anzeige von Memos und Bildern obliegt der Funktion *DrawDataCell*.

```
void __fastcall TDBBGrid::DrawDataCell(const TRect &Rect,
  TField *Field, TGridDrawState State)
{
  int i;
```

```
int weite;
TRect NRect;
AnsiString s;
TDBGrid::DrawDataCell(Rect, Field, State);

if(Field->ClassNameIs("TGraphicField"))
{
  if(FAutoDisp_Image == true)
  {
    FPicture->Assign(Field);
    i = int((Rect.Bottom - Rect.Top) *
      (FPicture->Width / (FPicture->Height + 0.001)));
    NRect.Left = Rect.Left;
    NRect.Top = Rect.Top;
    NRect.Right = Rect.Left + i;
    NRect.Bottom = Rect.Bottom;
    Canvas->FillRect(Rect);
    Canvas->StretchDraw(NRect, FPicture->Bitmap);
  } // if(FAutoDisp_Image == true)
} // if(Field->ClassNameIs("TGraphicField"))
```

Es wird zunächst untersucht, ob es sich bei dem zu zeichnenden Feld um ein Graphik-Feld handelt. Ist dies der Fall, dann wird entschieden, ob die Graphik gezeichnet oder ob aus Geschwindigkeitsgründen lieber darauf verzichtet werden soll.

Soll die Graphik gezeichnet werden, dann wird der Variablen *FPicture* der Inhalt dieses Feldes zugewiesen. Prinzipiell könnte man das Bild mit der *Canvas*-Methode *StretchDraw* nun in das Gitter zeichnen.

Für gewöhnlich wird aber die Tabellenzelle im Vergleich zur Höhe sehr lang sein, das Bild würde dementsprechend verzerrt. Deshalb werden zunächst neue Grenzen berechnet, welche sicherstellen, daß die Bildproportionen beibehalten werden. Die Anweisung *FPicture->height + 0.001* vermeidet einfach aber wirkungsvoll eine Division durch Null. Dies kommt dann vor, wenn bei einem leeren Datensatz die Höhe des Bildes Null ist. Vor dem Zeichnen mit der Methode *StretchDraw* werden frühere Einträge mit *FillRect* aus der Zelle entfernt.

```
if(Field->ClassNameIs("TMemoField"))
{
  if(FAutoDisp_Memo == true)
  {
    FStringList->Assign(Field);
    weite = Rect.Right - Rect.Left - Canvas->TextWidth("...");
```

```
      if(FStringList->Count > 0)
        s = FStringList->Strings[0];
      else
        s = "";
      while(Canvas->TextWidth(s) > weite)
        s.Delete(s.Length(), 1);
      Canvas->FillRect(Rect);
      Canvas->TextRect(Rect, Rect.Left, Rect.Top, s + "...");
    } // if(FAutoDisp_Memo == true)
  } // if(Field->ClassNameIs("TMemoField"))
} // DBBGrid1DrawDataCell
```

Zum Zeichnen von Memo-Feldern in die Tabelle wird zunächst der Inhalt des Feldes der Variablen *Zeilen* zugewiesen. Die erste Zeile – so diese denn existiert – wird in die Variable *s* eingelesen.

Damit der Anwender über den Inhalt des Feldes nicht getäuscht wird, sollen diesem drei Punkte angehängt werden (zum Zeichen, daß der Inhalt des Feldes noch über die Zellengrenze hinausgehen würde). Deshalb wird zunächst berechnet, wieviel Platz für den eigentlichen Text zur Verfügung steht. Danach wird die erste Zeile der String-Liste so lange gekürzt, bis der gewonnene String zusammen mit den Punkten in die Zelle paßt.

Die Zelle selbst wird dann wieder von früheren Einträgen befreit, danach wird der Textanfang aus dem Memo-Feld in die Zelle geschrieben.

Leicht problematisch ist die hier verwendete Vorgehensweise, wenn in der ersten Zeile extrem wenig Text oder jedesmal derselbe Text steht. Es wäre denkbar (wenn auch nicht sinnvoll), einen Memo-Text stets mit der Zeile *Beschreibung:* beginnen zu lassen. In der Tabelle würde dann stets immer nur dieses Wort angezeigt.

4.2.3 Die Komponente TDBCtrlGrid

Die Anzeige von Daten in *TDBGrid* ist relativ unflexibel. Ein Alternative hierzu ist die Komponente *TDBCtrlGrid*.

Auf das erste Panel des *DBCtrlGrids* werden während des Entwurfs die gewünschten Datensteuerungskomponenten abgelegt. Zur Laufzeit werden später entsprechend viele Panels angezeigt, die alle nach dem Muster des ersten Panels aufgebaut sind, aber jeweils verschiedene Datensätze anzeigen.

Im Prinzip lassen sich mit dieser Komponente sehr individuelle Eingabemasken erstellen. Das Problem dabei ist, daß die für uns wirklich interessanten Datensteuerungskomponenten sich nicht in das *DBCtrlGrid* einfügen lassen. Es handelt sich dabei um:

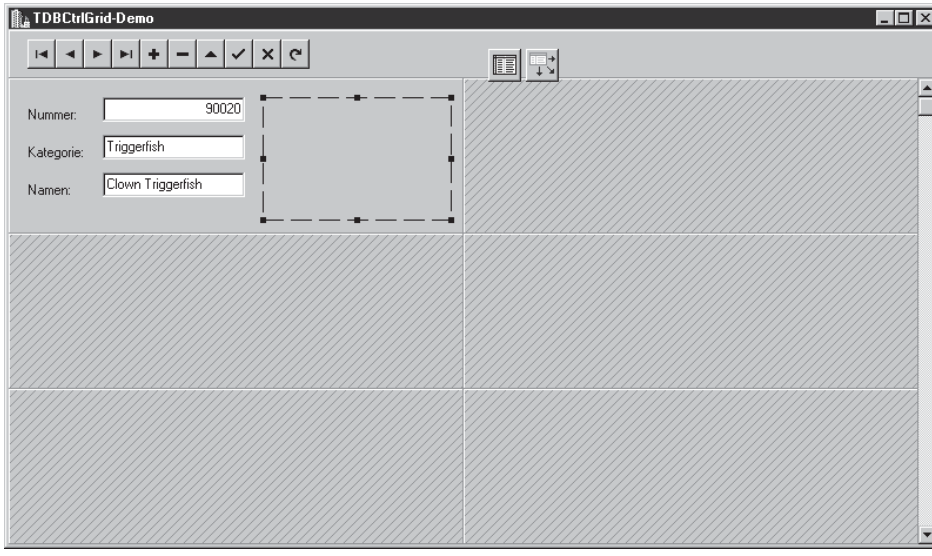


Bild 4.6: TDBCtrlGrid im Entwurf

- *DBMemo*
- *DBImage*
- *DBListBox* und *DBLookUpListBox*
- *DBRadioGroup*
- *DBGrid*, *DBNavigator* und *DBCtrlGrid*

Das Auftauchen von *DBImage* in dieser Liste mag den Leser zunächst ein Versehen des Autors vermuten lassen, sind doch in Bild 4.7 reichlich Bilder zu sehen. Zur Erstellung dieses Bildes wurde jedoch ein normales *Image* eingefügt, dessen Datenzugriff über den entsprechenden Quelltext sichergestellt wurde:

```
void __fastcall TForm1::DBBCtrlGrid1PaintPanel
  (TDBCtrlGrid *DBCtrlGrid,int Index)
{
  TRect NRect;
  FPicture->Assign(Table1->FieldByName("Graphic"));
  NRect.Left = 0;
  NRect.Top = 0;
  NRect.Right = Image1->Width;
  NRect.Bottom = Image1->Height;
  Image1->Canvas->StretchDraw(NRect, FPicture->Bitmap);
}
```

Das Ereignis *OnDBCtrlGrid1PaintPanel* wird immer dann ausgelöst, wenn eines der Panels neu dargestellt werden muß. Die Abmessungen des Images machen es überflüssig, daß zunächst aufwendig auf die Einhaltung der Bildproportionen geachtet werden muß.

Neben der Komponente *Image* können auch noch die Komponenten *Label, Panel, GroupBox, Shape, Bevel* und *PaintBox* in das *DBCtrlGrid* eingefügt werden. Soll der Inhalt eines Memos dargestellt werden, dann muß dies über *Canvas*-Methoden geschehen.

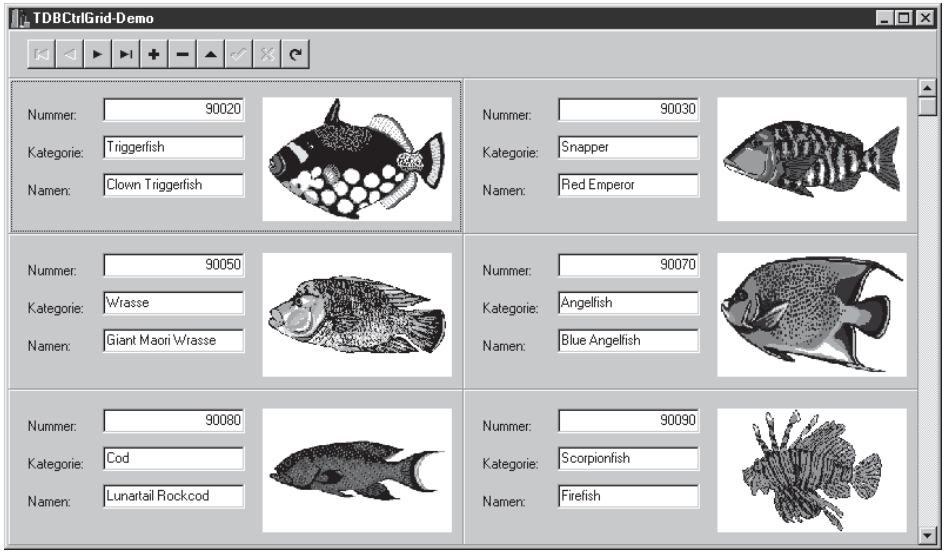


Bild 4.7: TDBCtrlGrid zur Laufzeit

4.2.4 Die Komponente TDBBCtrlGrid

Borland hat bei der Komponente *TDBCtrlGrid* die Veröffentlichung der Eigenschaft *Align* vergessen. Wer hier nicht auf ein Bug-Fix warten möchte, kann dies leicht selbst nachholen. Dazu wird eine Komponente von *TDBCtrlGrid* abgeleitet und die Header-Datei wie folgt erweitert:

```
class TDBBCtrlGrid : public TDBCtrlGrid
{
    ...
__published:
  __property Align;
};
```

4.3 Listen- und Kombinationsfelder

Auch die Komponenten *ListBox* und *ComboBox* haben ihre datensensitiven Varianten, und davon gibt es gleich sechs Stück:

4.3.1 Die Komponente TDBListBox

Eine einfache datensensitive *ListBox* bietet die Komponente *DBListBox*. Ihre Einträge werden in der Eigenschaft *Items* festgelegt. Liegt in dem verbundenen Datenfeld ein Eintrag vor, welcher in dieser Stringliste *Items* (Typ *TStrings*) vorhanden ist, dann wieder dieser Eintrag markiert angezeigt, ist der Eintrag nicht vorhanden (weil er beispielsweise auch über ein Edit-Feld eingegeben werden kann), dann wird kein Eintrag markiert.

Wird zur Laufzeit ein Eintrag angeklickt (und läßt sich die Datenmenge editieren), dann ist der String der Eigenschaft *Items* mit dem entsprechenden Index der neue Inhalt des Feldes. *Items* erlaubt übrigens auch leere Strings.

Die Eigenschaft *Style* hat die Voreinstellung *lbStandard*, in diesem Fall werden ausschließlich die Strings der Eigenschaft *Items* angezeigt. Wird der Eigenschaft *Style* der Wert *lbOwnerDrawFixed* zugewiesen, dann läßt sich die Höhe der Einträge mit der Eigenschaft *ItemHeight* einstellen. Des weiteren wird dann beim Zeichnen eines jeden Eintrags das Ereignis *OnDrawItem* ausgelöst und diesem der Zeichenbereich (*Rect: TRect*) als Parameter übergeben.

Somit hat der Programmierer die Möglichkeit, jeweils selbst auf *DBListBox.Canvas* zuzugreifen, beispielsweise um erläuternde Bilder hinzuzufügen oder die Einträge mit in verschiedenen Farben oder mit verschiedenen Schrifttypen durchzuführen.

Wird die Eigenschaft *Style* auf *lbOwnerDrawVariable* gesetzt, dann wird vor dem Zeichnen das Ereignis *OnMeasureItem* ausgelöst. Bei der entsprechenden Ereignisbehandlungsroutine

```
void __fastcall TForm1::DBListBox1MeasureItem(TWinControl *Control,
  int Index, int &Height)
```

wird die Höhe des jeweiligen Eintrags als Variablen-Parameter übergeben und kann somit vom Programmierer individuell eingestellt werden.

Wenn Sie die Eigenschaft *Sorted* auf *true* stellen, dann werden die Einträge in alphabetischer Reihenfolge sortiert. In diesem Zusammenhang wird dann auch eine Verhaltensweise von *DBListBox* richtig nützlich: Die Suche nach dem richtigen Eintrag in einer langen Liste – beispielsweise einer Liste von Städtenamen – läßt sich nämlich dadurch etwas abkürzen, daß man zunächst die *ListBox* an-

klickt, und dann die ersten Buchstaben des gewünschten Strings eingibt. Beginnt man beispielsweise mit *S*, dann wird zunächst *Saarbrücken* markiert, erweitert man den eingegebenen String auf *Stut*, dann erhält man *Stuttgart*.

4.3.2 Die Komponente TDBComboBox

Bei Platzmangel auf dem Formular empfiehlt es sich, statt einer Listbox eine ComboBox einzusetzen, da diese im Normalfall nicht mehr Platz benötigt als ein Edit-Feld. Zum Auswählen können die Einträge dann aufgeklappt werden.

Mit der Eigenschaft *Style* kann festgelegt werden, ob der Anwender wie bei einem *DBEdit* auch listenfremde Einträge per Tastatur eingeben kann (*csDropDown*), oder ob ausschließlich Einträge, die in der Liste enthalten sind, ausgewählt werden können (*cs-DropDownList*). Des weiteren besteht die Möglichkeit, wie bei der *DBListBox* beim Zeichnen der Einträge ein Ereignis auszulösen sowie (vorübergehend) die Drop-Down-Liste zu deaktivieren – die Komponente verhält sich dann wie ein *DBEdit*.

Einträge, die der Anwender über die Tastatur eingibt, werden nicht in die Liste aufgenommen, es sei denn, daß man dies per Quelltext selbst tut:

```
void __fastcall TForm1::DBComboBox1Exit(TObject *Sender)
{
  if(DBComboBox1->Items->IndexOf(DBComboBox1->Text) == -1)
    DBComboBox1->Items->Add(DBComboBox1->Text);
}
```

In diesem Fall sollte man dem Anwender auch die Chance geben, die hinzugefügten Einträge auch wieder zu entfernen:

```
void __fastcall TForm1::DBComboBox1KeyDown(TObject *Sender, WORD
&Key,
  TShiftState Shift)
{
  // ShowMessage(IntToStr(Key));
  if(Key == 114)
  {
    int i = Application->MessageBox
      ("Eintrag löschen", "Bestätigen", MB_YESNOCANCEL);
    if(i == 6)
      DBComboBox1->Items->Delete
        (DBComboBox1->Items->IndexOf(DBComboBox1->Text));
  } // if(Key == 114)
} // TForm1::DBComboBox1KeyDown
```

Betätigt der Anwender die Taste *F3*, dann löscht die Anweisung nach einer Sicherheitsabfrage den aktuellen Eintrag. Die Anweisung in der Kommentarklammer zeigt, wie man herausbekommt, hinter welcher Taste welcher Wert steht.

4.3.3 Die Komponenten TDBLookUpListBox und TDBLookUpComboBox

Bei den Komponenten *TDBLookUpListBox* und *TDBLookUpComboBox* wird statt einer Eigenschaft *Items* ein Feld einer weiteren Datenbanktabelle verwendet, aus denen die Komponenten ihre Einträge beziehen.

Bild 4.8 zeigt, wie die Komponente *TDBLookUpListBox* eingesetzt wird: Zwei Tabellen sind über eine 1:0 / 1 / n-Entität logisch miteinander verbunden. Zu jedem Kunden (Adreßdateien) gibt es keine, eine oder mehrere Rechnungen. In der Rechnung wird nicht die komplette Adresse des Kunden gespeichert, sondern nur die Kundennummer.

Für gewöhnlich wird man diese Tabellen über eine Referenz miteinander verbinden, für das Funktionieren der Komponenten ist dies jedoch nicht nötig. Durch die Verwendung einer *LookUp*-Komponente wird sichergestellt, daß keine Rechnungen in die Rechnungstabelle kommen, deren Kundennummern nicht in der Kundentabelle vorhanden sind (zumindest beim Erstellen der Rechnung – das spätere Löschen eines Kundendatensatzes ist bei dieser Vorgehensweise möglich, deshalb sollten solche Referenzen immer in der Datenbank definiert werden).

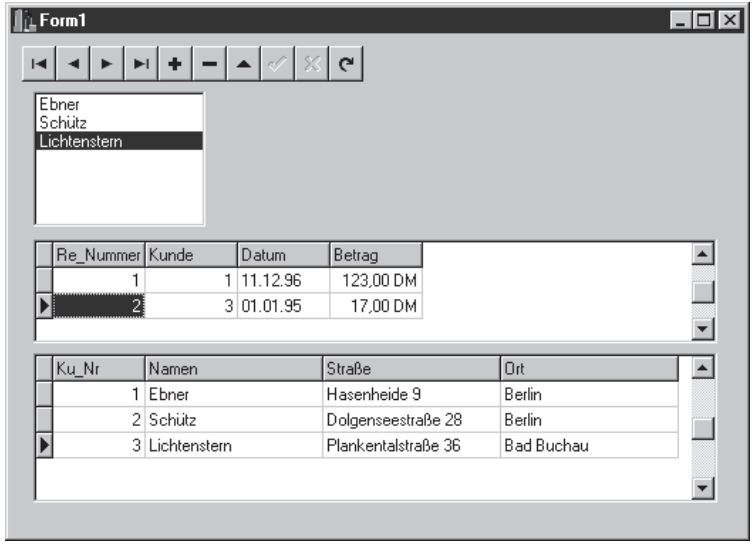


Bild 4.8: Verwendung einer LookUp-Liste

Für die *LookUp*-Einträge muß die Komponente mit einer zweiten Datenbank-Tabelle verbunden werden. Über die Eigenschaft *ListSource* wird die Komponente mit einer anderen(!) Datenquelle (als der mit der Eigenschaft *DataSource* spezifizierten Datenquelle) verbunden. Die Eigenschaft *KeyField* wird auf das Feld gesetzt, über welches die Referenz hergestellt wird. In vielen Fällen wird dieses Feld – wie hier – eine Nummer enthalten, und diese ist für die Darstellung in der Liste nicht besonders anschaulich.

Deshalb gibt es die Möglichkeit, mit der Eigenschaft *ListField* die Felder zu wählen, welche in der Liste angezeigt werden sollen. Soll lediglich ein einzelnes Feld angezeigt werden, dann kann dieses über die DropDown-Liste im Objektinspektor ausgewählt werden. Sollen hingegen mehrere Felder angezeigt werden (wie dies in Bild 4.9 gezeigt wird), dann werden diese durch Semikola getrennt der Eigenschaft *ListField* zugewiesen. Dies ist auch zur Laufzeit möglich:

```
DBLookUpList1->LookUpDisplay = "Namen;Straße;Ort";
```

Es ist darauf zu achten, daß dabei keine zusätzlichen (Leer-)Zeichen eingefügt werden.

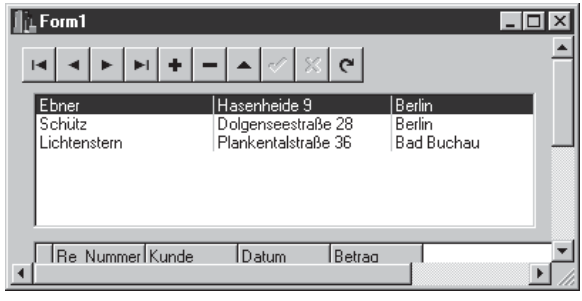


Bild 4.9: Mehrere Spalten in einer LookUp-Liste

Deutlich platzsparender kann die Komponente *TDBLookUpCombo* sein, wie Bild 4.10 beweist. Hier kann die Eigenschaft *DropDownWidth* so gewählt werden, daß für die aufgeklappte Liste (dann werden – so gewünscht – mehrere Felder angezeigt) mehr Platz verwendet wird als für das dazugehörende Edit-Feld.

Noch einen Hinweis zu den ScreenShots: Normalerweise sollte man die *LookUp*-Komponenten mit anderen einzelnen Datenbank-Dialogkomponenten kombinieren; damit hier aber gleichzeitig mehrere Datensätze angezeigt werden können, wurde hier die Kombination mit einem *DBGrid* gewählt (was in einer Anwendung nichts als Ärger bringt).

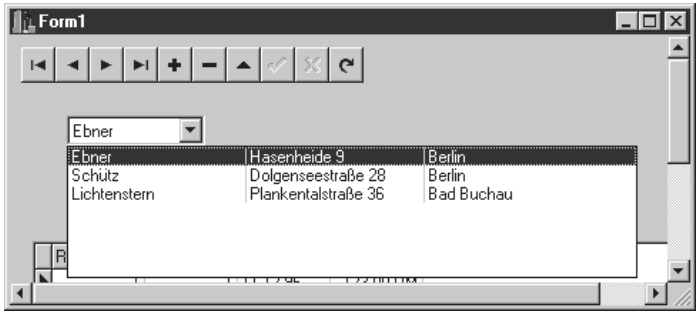


Bild 4.10: Die Komponente TDBLookUpCombo

4.3.4 Die Komponenten TDBLookUpList und TDBLookUpCombo

Bei C++Builder sind die Komponenten *TDBLookUpList* und *TDBLookUpCombo* auf der Seite *Win3.1* zu finden und hier auch nur der Abwärtskompatibilität wegen enthalten; sie werden durch die neuen Komponenten *TDBLookUpListBox* und *TDBLookUp-ComboBox* ersetzt. Da alle Delphi 2.0-Komponenten auch in C++Builder vorhanden sind, gibt es hier auch diese beiden Komponenten – aber keine Vorgängerversion, mit der sie kompatibel sein könnten. Wir wollen uns deshalb nicht näher damit beschäftigen.

4.4 Sonstige

Zudem gibt es an datensensitiven Dialogelementen die Komponenten *TDBCheck-Box*, *TDBRadioGroup* und *TDBImage*.

4.4.1 Die Komponente TDBCheckBox

Die Komponente *TDBCheckBox* ist insbesondere für den Zugriff auf boolesche Felder geeignet. Im Gegensatz zur Komponente *TCheckBox* ist die Eigenschaft *State* nur zur Laufzeit verfügbar. Standardmäßig ist deren Wert *cbChecked* mit dem Feldwert *true* verbunden, *cbUnchecked* mit *false* und *cbGrayed* mit dem Wert NULL (bei noch leeren Datensätzen beispielsweise).

Normalerweise erlaubt die Komponente *TDBCheckBox* lediglich die Eingabe der Zustände *cbChecked* und *cbUnchecked* – der Zustand *cbGrayed* wird nur bei noch leeren Datensätzen angezeigt. Wird jedoch die Eigenschaft *AllowGrayed* auf *true* gesetzt, dann können auch leere Feldwerte eingegeben werden; dies kann beispielsweise dann notwendig sein, wenn über einen Status keine Klarheit besteht.

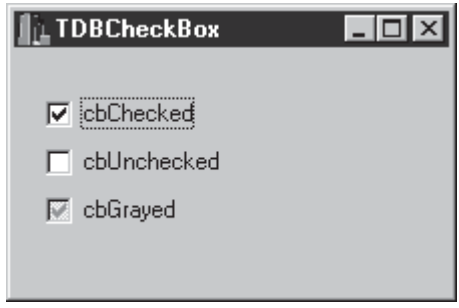


Bild 4.11: Die Status-Möglichkeiten einer TDBCheckBox

Die Komponente *TDBCheckBox* kann auch zusammen mit anderen als booleschen Feldern verwendet werden. Dazu dienen die beiden Eigenschaften *ValueChecked* und *ValueUnchecked*, deren Voreinstellung (in der deutschen Version) die Strings *Wahr* und *Falsch* sind. Wird die Komponente mit einem String-Feld verbunden, dann wird einer der beiden Strings in das Feld eingetragen, je nach Zustand der *CheckBox*.

Umgekehrt wird beim Blättern durch die Datenbank die Komponenente *Checked* angezeigt, wenn der Feldwert dem Wert der Eigenschaft *ValueChecked* entspricht, entsprechend verhält es sich mit *ValueUnchecked*. Des weiteren erlaubt es C++Builder, diesen beiden Eigenschaften mehrere Werte zuzuweisen, diese müssen dabei durch Semikola getrennt werden:

```
CheckBox1->ValueChecked = "Kreuz;Pik;Herz;Karo";
```

In diesem Fall würde die Komponente immer *Checked* anzeigen, wenn der Feldwert *Kreuz*, *Pik*, *Herz* oder *Karo* beträgt. Wird jedoch ein Datensatz editiert und dabei die Komponente auf *Checked* gesetzt, dann wird immer der erste Eintrag, in diesem Fall *Kreuz* eingefügt. Wird ein Datensatz editiert, dessen betreffender Feldwert *Herz* ist, dann bleibt dieser so, bis an der *CheckBox* etwas geändert wird. Wird jedoch die *CheckBox* zunächst auf *Unchecked* und dann wieder auf *Checked* gestellt, dann lautet der Feldinhalt *Kreuz*.

4.4.2 Die Komponente TDBRadioGroup

Die Komponente *TDBRadioGroup* ähnelt in ihrer Funktion stark der Komponente *TDBListBox*. Auch hier werden in der Eigenschaft *Items* Werte vorgegeben, welche dann die Beschriftung des davorstehenden *DBRadioButton* bilden. Im Gegensatz zur *DBListBox* läßt sich hier nichts scrollen, so daß die Anzahl der Einträge sehr limitiert ist, es sei denn, man verwendet eine *ScrollBox*, in welche eine entsprechend große *DBRadioGroup* gesteckt wird.

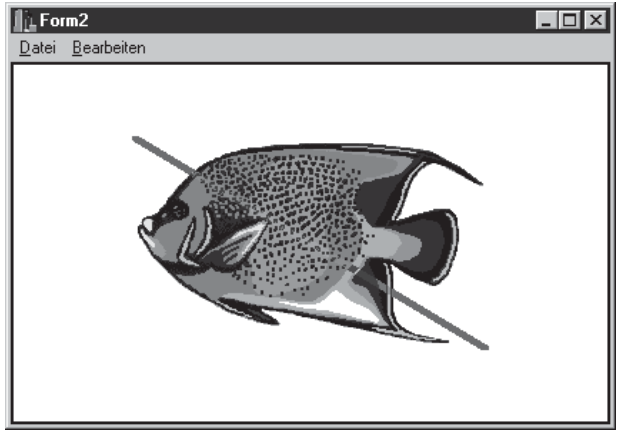
Für den Fall, daß die Beschriftung des RadioButtons nicht zum Inhalt des Datenbankfeldes werden soll, kann zusätzlich die Eigenschaft *Values* (*TStrings*) gesetzt werden; in diesem Fall wird der entsprechende Eintrag von *Items* als Beschriftung angezeigt, während der Eintrag von *Values* in die Datenbank übernommen wird.

Beim Durchblättern der Datenbank wird der entsprechende *DBRadioButton* markiert, dessen *Values*- oder *Items*-Wert gerade im entsprechenden Feld erscheint. Ist der Feldeintrag nicht als String in diesen Eigenschaften enthalten, dann wird kein *DBRadioButton* markiert.

4.4.3 Die Komponente TDBImage

Die Komponente *TDBImage* wird dazu verwendet, um Bilder einer Datenbank anzuzeigen. Es ist auch möglich, diese Bilder in die Zwischenablage zu verschieben oder zu kopieren sowie neue Bilder über die Zwischenablage einzufügen. In Kapitel 4.2.1 sind die entsprechenden Anweisungen verwendet worden, dort wird dann auch gezeigt, wie sich die Bilder in eine Datei schreiben und von dort aus laden lassen.

Das Durchblättern einer Datenbank kann sehr zeitaufwendig werden, wenn dabei Bilder angezeigt werden sollen. Deshalb besteht die Möglichkeit, die Eigenschaft *AutoDisplay* auf *false* zu setzen. In der Regel wird dann der Feldname als Text in Klammern angezeigt. Führt der Anwender nun einen Doppelklick auf das *DBImage* aus oder wird die Methode *LoadPicture* aufgerufen, dann wird das Bild angezeigt.



4.12: Zeichen auf dem Canvas von TDBImage

Wird die Eigenschaft *Stretch* auf *true* gesetzt, dann wird das dargestellte Bild der Größe der Komponente *TDBImage* angepaßt, auf ein Beibehalten der Bildproportionen wird dabei allerdings kein Wert gelegt (die Ergebnisse können somit auch ein wenig seltsam ausfallen). Sinnvoll wäre es, wenn man hier die Eigenschaft *Autosize* auf *true* setzen könnte; diese Eigenschaft gibt es allerdings bei *TDBImage* nicht (sie läßt sich auch nicht einfach veröffentlichen).

Vorhanden ist die Eigenschaft *Picture* (*TPicture*), die einen vernünftigen Zugriff auf das enthaltene Bild ermöglicht, beispielsweise das Laden von und das Speichern in Dateien (in Kapitel 4.2.1 wurde davon Gebrauch gemacht).

Des weiteren läßt sich so auf ein *Canvas*-Objekt der Komponente zugreifen. Wir wollen dies kurz ausprobieren und dazu das Projekt aus Kapitel 4.2.1 erweitern. Wie Sie sich vielleicht noch erinnern werden, kann man dort aus einem *DBGrid* ein Formular öffnen, welches das jeweilige Bild in einer *TImage*-Komponente anzeigt; zudem wurden Möglichkeiten zum Speichern und Laden sowie für Zwischenablage-Aktionen geschaffen.

Nun wollen wir auf die Tatsache reagieren, daß auf dieser Erde jeden Tag Tierarten aussterben (ob die gezeigten Fische gefährdet sind, entzieht sich allerdings meiner Kenntnis). In diesem Fall soll der Datensatz nicht gelöscht, sondern das Bild rot durchgestrichen werden. Der entsprechende Menüpunkt startet folgende Prozedur:

```
void __fastcall TForm2::Durchstreichen1Click(TObject *Sender)
{
  TPen* OldPen;
  OldPen = DBImage1->Picture->Bitmap->Canvas->Pen;
  DBImage1->Picture->Bitmap->Canvas->Pen->Width = 5;
  DBImage1->Picture->Bitmap->Canvas->Pen->Color = clTeal;
  DBImage1->Picture->Bitmap->Canvas->Pen->Mode = pmXor;
  DBImage1->Picture->Bitmap->Canvas->MoveTo(0,0);
  DBImage1->Picture->Bitmap->Canvas->LineTo
    (DBImage1->Picture->Width, DBImage1->Picture->Height);
  DBImage1->Picture->Bitmap->Canvas->Pen = OldPen;
}
```

Zum Durchstreichen soll eine dicke, rote Linie verwendet werden, die Eigenschaft *Pen* muß also entsprechend geändert werden. Damit die Einstellungen später wiederhergestellt werden können, sollen die aktuellen Werte der Variablen *OldPen* zugewiesen werden. Um ein versehentliches Durchstreichen wieder rückgängig machen zu können, wird als *Pen->Mode = pmXor* verwendet. Anschließend wird die Linie gezeichnet, und danach werden die alten Stifteinstellungen wieder aktiviert.

4.4.4 Die Komponente TDBNavigator

Mit dem *DBNavigator* steuert man komfortabel durch die verbundene Datenmenge. Im einzelnen sind folgende Funktionen verfügbar (von links nach rechts):

- erster Datensatz (*First*)
- vorheriger Datensatz (*Prior*)
- nächster Datensatz (*Next*)
- letzter Datensatz (*Last*)
- Datensatz einfügen (*Insert*)
- Datensatz löschen (*Delete*)
- Datensatz bearbeiten (*Edit*)
- übernehmen (*Post*)
- Bearbeitung abbrechen (*Cancel*)
- Datensatz aktualisieren (*Refresh*)

Die Methodennamen in Klammern geben an, welche *DataSet*-Methode dabei aufgerufen wird. Machen Aufrufe keinen Sinn, dann werden die entsprechenden Schaltflächen deaktiviert dargestellt. In Bild 4.13 sind dies die Schaltflächen für den ersten und vorherigen Datensatz (folglich steht der Datenzeiger auf dem ersten Datensatz), des weiteren die Schaltflächen für Übernehmen und Bearbeitung abbrechen (folglich wurde an diesem Datensatz noch nichts geändert).

Mit der Eigenschaft *VisibleButtons* kann festgelegt werden, welche der zehn Schaltflächen angezeigt werden sollen. Wird beispielsweise mit dem *DBNavigator* eine *TQuery*-Komponente gesteuert, deren Eigenschaft *RequestLive* auf *false* steht, dann brauchen nur die ersten vier Schaltflächen angezeigt zu werden, weil die restlichen ohnehin nie aktiviert werden würden.

Wird die Eigenschaft *ShowHint* auf *true* gesetzt, dann wird zu jeder Schaltfläche des Navigators ein kleiner Hilfetext angezeigt (siehe Aufzählung oben), sobald der Cursor längere Zeit darauf verweilt. Die vordefinierten Hilfetexte können überschrieben werden, indem die Eigenschaft *Hints* gesetzt wird. Die Eigenschaft *ConfirmDelete* sorgt dafür, daß eine Sicherheitsabfrage durchgeführt wird, bevor über *DBNavigator* ein Datensatz gelöscht wird.

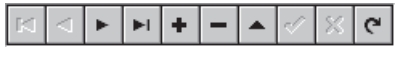


Bild 4.13: Die Komponente TDBNavigator

4.5.4 Die Komponente TDBNavigatorSpec

Bei der Komponente *TDBNavigatorSpec* handelt es sich um eine FreeWare-Komponente, welche von mir leicht überarbeitet wurde. Den Pascal-Quelltext finden Sie auf der beiliegenden CD.

TDBNavigatorSpec wurde um vier Buttons erweitert. Zum einen lassen sich damit Sprünge um viele Datensätze vornehmen. Soll mit dem normalen *DBNavigator* ein Datensatz in der Mitte einer großen Tabelle gefunden werden, dann ist dies recht zeitaufwendig. Mit *TDBNavigatorSpec* läßt sich die Datenmenge in Sprüngen von beispielsweise 50 Datensätzen durchscrollen; die Anzahl der Datensätze läßt sich mit der Eigenschaft *Jump* einstellen. Die beiden *Hint*-Texte werden dementsprechend automatisch gesetzt.

Des weiteren läßt sich ein Lesezeichen (engl. *bookmark*) definieren, zu dem dann per Button-Klick gesprungen werden kann. Noch ein Hinweis zu den *Hints*: Auch bei dieser Komponente lassen sich keine zwei *Hints* auf einmal anzeigen, Bild 4.14 ist mit Hilfe eines Bildbearbeitungsprogramms entstanden.

Bild 4.14: Die Komponente TDBNavigatorSpec

5 QuickReport-Komponenten

Für gewöhnlich möchte man sich seine Daten und Abfrageergebnisse nicht nur auf dem Bildschirm ausgeben lassen, sondern sie auch zu Papier bringen. Dafür gibt es beim C++Builder prinzipiell zwei Möglichkeiten:

- Mit den *QuickReport*-Komponenten lassen sich schnell einfache Reports erstellen, welche sich relativ nahtlos in die C++Builder-Programme einfügen.
- Daten lassen sich auch mit den *Printer*-Methoden von C++Builder ausdrukken. Dies ist aber längst nicht so komfortabel wie mit den *QuickReport*-Komponenten, dafür aber noch flexibler. Zum Ausdruck mit den *Printer*-Methoden mehr in Kapitel 6.

5.1 Eine Adressenliste

Zunächst wollen wir als Einstieg eine einfache Adressenliste ausdrucken, wie sie in Bild 5.1 auf der nächsten Seite gezeigt wird. Da der Umgang mit den *QuickReport*-Komponenten etwas gewöhnungsbedürftig und die OnlineHilfe erstens dürftig und zweitens nicht gerade leicht verständlich ist, soll zunächst sehr ausführlich auf die nötigen Schritte eingegangen werden:

Zunächst wird ein weiteres Formular eingefügt, dessen Eigenschaft *Visible* auf *false* gestellt wird – die Anzeige übernimmt später die *QuickReport*-Komponente. Die Unit des neuen Formulars wird dann in die Unit des Hauptformulars eingebunden.

Des weiteren wird das Hauptmenü des Hauptformulars um einen Menüoberpunkt *Reports* erweitert, unter den die Menüpunkte *(1) Adressen* und schon gleich auch *(2) Etiketten* eingefügt werden. Der Menüpunkt Adressen wird dann mit folgener Ereignisbehandlungsroutine verknüpft:

```
void __fastcall TForm1::Adressenliste1Click(TObject *Sender)
{
  Form2->QuickReport1->Preview();
}
```

Nun muß noch der eigentliche Report eingerichtet werden. Dazu wird zunächst ein neues Formular erstellt. Dort werden die Komponenten *TDataSource* und

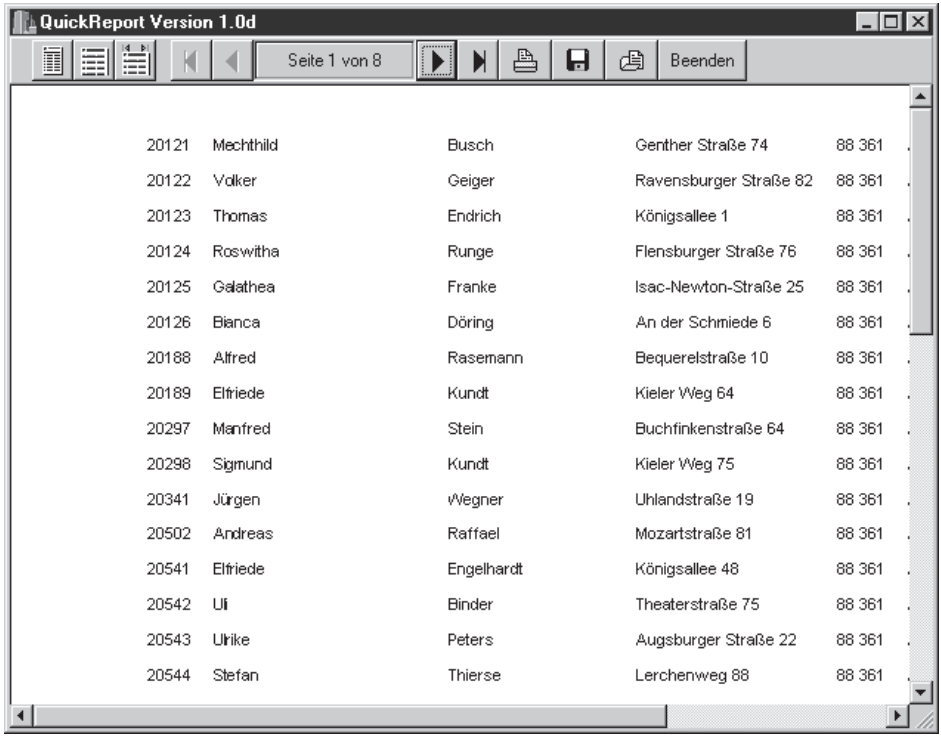


Bild 5.1: Eine Adressenliste mit QuickReport

TQuery eingefügt und mit der entsprechenden Tabelle verknüpft. Da dies hier eine Demo und kein Geduldsspiel werden soll, sollen nicht alle Adressen aufgelistet werden; als *SQL*-Anweisung wird deshalb folgendes eingegeben:

```
SELECT * FROM adressen
  WHERE ort = "Altshausen"
```

Danach fügt man in *Form2* die Komponente *TQuickReport* ein. Diese Komponente enthält die Steuerung und ändert das Formular in einen Report. Zu den Eigenschaften und Methoden kommen wir später, momentan können alle Eigenschaften ihre Standardwerte beibehalten, lediglich die Eigenschaft *DataSource* wird auf *DataSource1* gestellt.

Als nächstes wird eine *TQRBand*-Komponente eingefügt. Deren Eigenschaft *BandType* wird auf *rbDetail* gesetzt, in solchen Bändern werden die eigentlichen Daten ausgegeben. In *QRBand1* werden nun *TQRDBText*-Komponenten eingefügt, und zwar für jedes anzuzeigende Feld eine. Die Eigenschaft *DataSource* wird auf *DataSource1* geändert, der Eigenschaft *DataField* weist man das anzuzeigende Feld zu.

Die Anwendung kann dann gestartet werden. Nach dem Aufruf des Menüs sieht man eine Fortschrittsanzeige, danach wird der Report wie in Bild 5.1 angezeigt.

Der Report kann auch durch einen Doppelklick auf das Icon von *TReport* gestartet werden – derjenige Teil der Funktionalität jedoch, der durch den C++Builder-Quelltext implementiert wird, funktioniert jedoch dabei nicht.

Mit den Buttons unter der Titelleiste läßt sich der Report steuern. Die ersten drei Buttons ändern die Größe der Darstellung: Es läßt sich wahlweise eine ganze Seite darstellen, eine Anzeige in Originalgröße vornehmen oder die Seite auf Bildschirmbreite zoomen. Mit den nächsten vier Buttons kann man durch die Seiten des Reports navigieren. Des weiteren läßt sich der Report ausdrucken, speichern und ein anderer Report laden.

5.2 Ausdruck von Etiketten

Eine der wohl häufigsten Aufgaben von Reportgeneratoren ist der Ausdruck von Etiketten; dies geht mit QuickReport-Komponenten besonders einfach. Die ersten Schritte sind gleich wie beim ersten Report, der Unit-Namen ist entsprechend abzuändern.

Die erste Änderung betrifft die SQL-Anweisung von *Form4->Query1*:

```
SELECT vornamen || " " || nachnamen AS namen,
    straße, plz || " " || ort AS wohnort
  FROM testadr
  WHERE (ort = "Altshausen") AND (tel2 IS NOT NULL)
```

Mit der Erweiterung dieser Anweisung werden zwei Ziele verfolgt: Zum einen soll aus Geschwindigkeitsgründen die Datenmenge weiter eingeschränkt werden, deshalb werden nur Datensätze mit Funktelefonnummer aufgenommen. A propos Geschwindigkeit: Es ist keinesfalls so, daß die Erstellung von *QuickReports* besonders langsam wäre. Dieses Beispiel benötigt knapp eine Sekunde für die Darstellung von 136 Datensätzen auf sechs Seiten, im Gegensatz zu manch anderen Report-Generatoren ist das ziemlich schnell; trotzdem möchte ich Ihnen bei einem Demo keine unnötige Warterei zumuten.

Die andere Neuerung betrifft die Zusammenfassung von Vor- und Nachnamen sowie von Postleitzahl und Ort. Dadurch wird jeweils nur ein *QRDBText* erforderlich, und die beiden Wörter schließen direkt aneinander an. (Bei zwei Feldern müßten Sie zwei *QRDBText*-Komponenten verwenden. Da die Vornamen unterschiedlich lang sind, würde zwischen Vor- und Nachnamen eine unterschiedlich lange Lücke entstehen.)

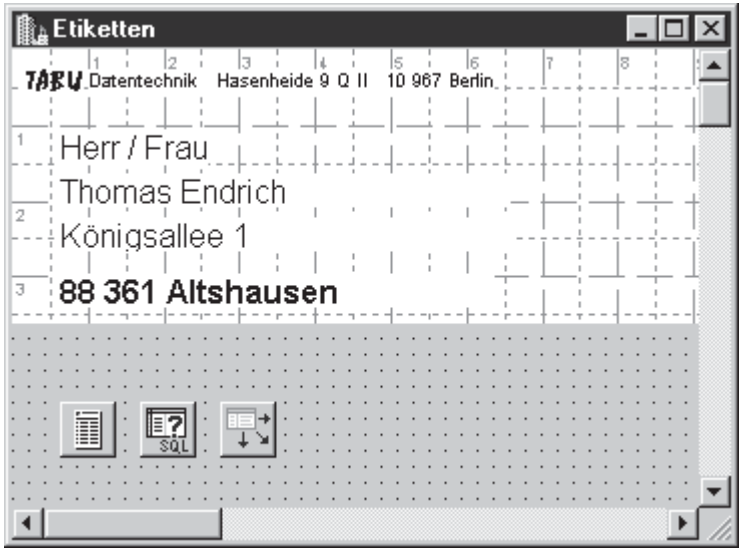


Bild 5.2: Entwickeln von Adreßetiketten

Die Eigenschaft *Columns* von *Form3->QuickReport1* ist auf *3* zu setzen; die Ausgabe wird dann in drei Spalten formatiert (so einfach geht das mit den Etiketten). Das *QRBand1* muß nun in der Höhe den Etiketten angepaßt werden. Dazu setzt man die Eigenschaft *Ruler* auf *qrrCmHV* und erhält damit eine horizontale wie vertikale Zentimeterteilung, wie sie in Bild 5.2 zu sehen ist. Mit der Größe muß dann ein wenig experimentiert werden.

Auf jedes Etikett soll eine Absenderangabe stehen, wozu die Komponente *TQRLabel* verwendet wird. Auf dem Etikett können verschiedene Schriftarten, -größen und -stile verwendet werden. Mit der Verwendung von TrueType-Schriften kann man an dieser Stelle einige Komplikationen vermeiden.

5.2.1 Eigenes Vorschauformular

Wie vorhin festgestellt wurde, ist das Vorschauformular von *TQuickReport* nicht optimal; es kann allerdings leicht durch ein eigenes ersetzt werden. Dazu wird ein neues Formular erstellt, welches zunächst mit einem Menü, einem Panel und der Komponente *TQRPreview* versehen wird.

Gegenüber dem Standard-Vorschaufenster sind einige weitere Optionen implementiert worden:

- Der Zoom-Faktor der Ansicht läßt sich von 10% bis 1000 % in 10%-Schritten einstellen. Die Möglichkeit, die ganze Seite darzustellen oder auf Seitenbreite zu zoomen, sind weiterhin enthalten.

Bild 5.3: Das eigene Vorschau-Formular

- Beim Ausdruck können die auszudruckenden Seiten und die Anzahl der Kopien gewählt werden.
- Es können Übersichtsseiten erstellt werden, d.h., auf einer Druckseite können eine, vier, neun oder sechzehn Reportseiten ausgedruckt werden.

Der Aufruf des eigenen Vorschauformulars ist ein wenig aufwendiger als der Aufruf Standard-Formulars. Zunächst muß *QRPrinter->OnPreview* zugewiesen werden, welche Prozedur auszuführen ist, um das eigene Vorschauformular zu starten. Soll das Standard-Vorschauformular verwendet werden, dann muß *QRPrinter.OnPreview* der Wert *NULL* zugewiesen werden.

```
void __fastcall TForm1::Etiketten1Click(TObject *Sender)
{
  QRPrinter->OnPreview = ShowLabels;
  Form3->QuickReport1->Preview();
}
```

Die Prozedur *ShowLabels* (»zeige Etiketten«) besteht lediglich aus dem *ShowModal*-Aufruf des Formulars. Es ist jedoch weder möglich, diese Anweisung direkt *QRPrinter->OnPreview* zuzuweisen, noch die Methode *Show* statt *ShowModal* zu verwenden.

```
void __fastcall TForm1::ShowLabels(void)
{
  Form4->ShowModal();
}
```

Wenn das Formular gestartet wird, dann wird zunächst der Zoom-Faktor auf 100 % gestellt. Die Variable *PageCount* ist eine globale Formular-Variable (deklarieren nicht vergessen !), welche die Zahl der Report-Seiten beinhaltet. Sobald der Wert zugewiesen worden ist, wird der Wert im *Seiten*-Panel mit der Prozedur *ShowPageNumber* angezeigt.

```
void __fastcall TForm4::FormShow(TObject *Sender)
{
  UpDown1->Position = 100;
  ComboBox1->Text = "";
  FPageCount = Form3->QuickReport1->PageCount;
  ShowPageNumber(Sender);
}
```

Zur Ausgabe der aktuellen Seite und der Gesamtseitenzahl wird *Label2* verwendet.

```
void __fastcall TForm4::ShowPageNumber(TObject *Sender)
{
  Label2->Caption = "Seite " + IntToStr(QRPreview1->PageNumber)
    + " von " + IntToStr(FPageCount);
}
```

Zum Navigieren durch die Seiten werden vier Buttons verwendet, welche die Eigenschaft *QRPreview->PageNumber* entsprechend setzen. Es ist dabei darauf zu achten, daß der Sprung auf nicht vorhandene Seiten ausgeschlossen wird:

```
void __fastcall TForm4::Button1Click(TObject *Sender)
{
  QRPreview1->PageNumber = 1;
  ShowPageNumber(Sender);
}

void __fastcall TForm4::Button4Click(TObject *Sender)
{
  QRPreview1->PageNumber = FPageCount;
  ShowPageNumber(Sender);
}
```

```
void __fastcall TForm4::Button2Click(TObject *Sender)
{
  if(QRPreview1->PageNumber > 1)
    QRPreview1->PageNumber = QRPreview1->PageNumber - 1;
  ShowPageNumber(Sender);
}

void __fastcall TForm4::Button3Click(TObject *Sender)
{
  if(QRPreview1->PageNumber < FPageCount)
    QRPreview1->PageNumber = QRPreview1->PageNumber + 1;
  ShowPageNumber(Sender);
}
```

Nun soll die Zoom-Funktion implementiert werden. Dazu werden in *ComboBox1* die Einträge *100%*, *Ganze Seite* und *Seitenbreite* hinzugefügt; *ComboBox1* wird mit folgender *OnChange*-Ereignisbehandlungsroutine verknüpft.

```
void __fastcall TForm4::ComboBox1Change(TObject *Sender)
{
  if(ComboBox1->ItemIndex == 0)
  // 100 %
  {
    QRPreview1->Zoom = 100;
    UpDown1->Position = 100;
  }
  if(ComboBox1->ItemIndex == 1)
  // Ganze Seite
  {
    QRPreview1->ZoomToFit();
    UpDown1->Position = QRPreview1->Zoom;
  }
  if(ComboBox1->ItemIndex == 2)
  // Seitenbreite
  {
    QRPreview1->ZoomToWidth();
    UpDown1->Position = QRPreview1->Zoom;
  }
} // TForm4::ComboBox1Change
```

Um weitere Vergrößerungsfaktoren einstellen zu können, wird eine *TUpDown*-Komponente eingefügt, deren Eigenschaften folgendermaßen zugewiesen werden: *Increment := 10, MaxValue := 1000, MinValue := 10.* Die *OnChange*-Routine ist

dann sehr einfach zu programmieren; vergessen Sie nicht, *ComboBox1* zu löschen, damit keine widersprüchlichen Werte angezeigt werden.

```
void __fastcall TForm4::UpDown1Click(TObject *Sender, TUDBtnType Button)
{
  QRPreview1->Zoom = UpDown1->Position;
  ComboBox1->Text = "";
}
```

Statt mit *ComboBox1* können die drei Standardvergrößerungen auch über das Menü eingestellt werden; da die Anweisungen identisch sind, sollen sie hier nicht wiederholt werden.

Bisweilen möchte man einen Report zunächst abspeichern, um ihn später wieder zu laden und dann auszudrucken (beispielsweise wenn man die Reports unterwegs auf dem Notebook erstellt und daheim dann ausdruckt). Deshalb wollen wir nun Methoden zum Speichern und Laden der Reports bereitstellen.

```
void __fastcall TForm4::Berichtspeichern1Click(TObject *Sender)
{
  if(SaveDialog1->Execute())
    QRPrinter->Save(SaveDialog1->FileName);
}

void __fastcall TForm4::Berichtffnen1Click(TObject *Sender)
{
  if(OpenDialog1->Execute())
    QRPrinter->Load(OpenDialog1->FileName);
  QRPreview1->Zoom = QRPreview1->Zoom + 1;
  UpDown1->Position = 100;
  Application->ProcessMessages();
  ShowPageNumber(Sender);
}
```

In *Save-* und *OpenDialog1* wird **.qrp* als Quick-Report-Filter eingefügt. Es mag verwundern, daß beim Öffnen eines Reports der Zoom-Faktor leicht erhöht wird – dies ist erforderlich, damit die Anzeige aktualisiert wird.

Etwas aufwendiger wird der Ausdruck des Reports, zumindest dann, wenn nicht einfach der komplette Report unverändert gedruckt werden soll.

```
void __fastcall TForm4::Drucken1Click(TObject *Sender)
{
  PrintDialog1->MinPage = 1;
  PrintDialog1->FromPage = 1;
```

```
  PrintDialog1->MaxPage = FPageCount;
  PrintDialog1->ToPage = FPageCount;
  if(PrintDialog1->Execute())
  {
    QRPrinter->Title = "QuickReport Etiketten";
    QRPrinter->FromPage = PrintDialog1->FromPage;
    QRPrinter->ToPage = PrintDialog1->ToPage;
    QRPrinter->Orientation = Form3->QuickReport1->Orientation;
    QRPrinter->Print();
  }
} // TForm4::Drucken1Click
```

Da die Zahl der Report-Seiten zur Entwurfszeit noch nicht feststeht, werden die *PrintDialog1*-Eigenschaften *MinPage*, *FromPage*, *MaxPage* und *ToPage* erst zur Laufzeit gesetzt. Die Lage der Druckseite wird der des Berichts angepaßt. Mit der Methode *Print* wird der Ausdruck gestartet.

Beim Ausdruck von Etiketten eher eine Spielerei, bei anderen Reports dagegen manchmal sehr nützlich, ist der Ausdruck von Übersichtsseiten. Dazu werden vier, neun oder sechzehn Reportseiten auf einer Druckseite zusammengefaßt (bei entsprechender Verkleinerung). Die Verwendung von Übersichtsseiten wird über das Menü gesteuert, die Zahl der Übersichtsseiten wurde dabei auf höchstens sechzehn begrenzt. Um weniger Quelltext schreiben zu müssen, wurde eine gemeinsame *OnClick*-Routine für alle vier Menüpunkte erstellt:

```
void __fastcall TForm4::Keine1Click(TObject *Sender)
// Zusammenfassungsseiten
{
  Keine1->Checked = false;
  N41->Checked = false;
  N91->Checked = false;
  N161->Checked = false;
  if(Sender == Keine1)
  {
    Keine1->Checked = true;
    QRPrinter->Thumbs = 1;
  }
  if(Sender == N41)
  {
    N41->Checked = true;
    QRPrinter->Thumbs = 2;
  }
```

```
  if(Sender == N91)
  {
    N91->Checked = true;
    QRPrinter->Thumbs = 3;
  }
  if(Sender == N161)
  {
    N161->Checked = true;
    QRPrinter->Thumbs = 4;
  }
} // TForm4::Keine1Click
```

Das eigene Vorschauformular ist nun fertig und kann auch für andere Berichte und in anderen Projekten verwendet werden.

5.3 Reports mit Bildern und Memos

Mit den *QuickReport*-Komponenten können auch Reports erstellt werden, die Bilder und Memos anzeigen, dazu werden lediglich entsprechende *TImage*- und *TDBImage*-Komponenten auf die Bänder gesetzt. Die Fähigkeit zur Anzeige von Memos ist gleich in den *TQRDBText*-Komponenten enthalten.

Bild 5.5 zeigt einen solchen Report zur Entwurfszeit. Hier sind nun drei verschiedene Bänder vorgesehen worden: Beim obersten Band ist die Eigenschaft *BandType* gleich *rbTitle*, dieses Band soll also zu Beginn des Reports einmalig auf der ersten Seite gedruckt werden. In dieses Band ist eine *TImage*-Komponente eingefügt. (Die Graphik mit den verschiedenen Fischen wurde per Bildbearbeitungsprogramm erstellt.)

Das nächste Band ist das wohlbekannte *rbDetail*-Band, welches einmal pro (Haupt-) Datensatz verwendet wird. Hier wird eine *TDBImage*-Komponente eingefügt; mehr braucht für die Anzeige von Bildern nicht getan zu werden.

Für das Memo wird die Komponente *TQRDBText* verwendet. *QuickReport* fügt automatisch weitere Zeilen an, wenn sich mehr Text im Memo-Feld befindet. Es ist nicht nötig, diese Komponente auf den Bereich zu vergrößern, in welchem das Memo erscheinen soll (damit würde man nur den Zeilenabstand auf einen unsinnig großen Wert setzen).

Als nächstes wird hier eine Fußzeile für alle Seiten erstellt: Zu diesem Zweck fügt man ein weiteres Band ein und setzt dessen Eigenschaft *BandType* auf *rbPageFooter*. Hier sollen nun der Report-Titel, Datum und Zeitpunkt der Reporterstellung sowie die Seitenzahl erscheinen. Für die Darstellung solcher Informationen gibt es

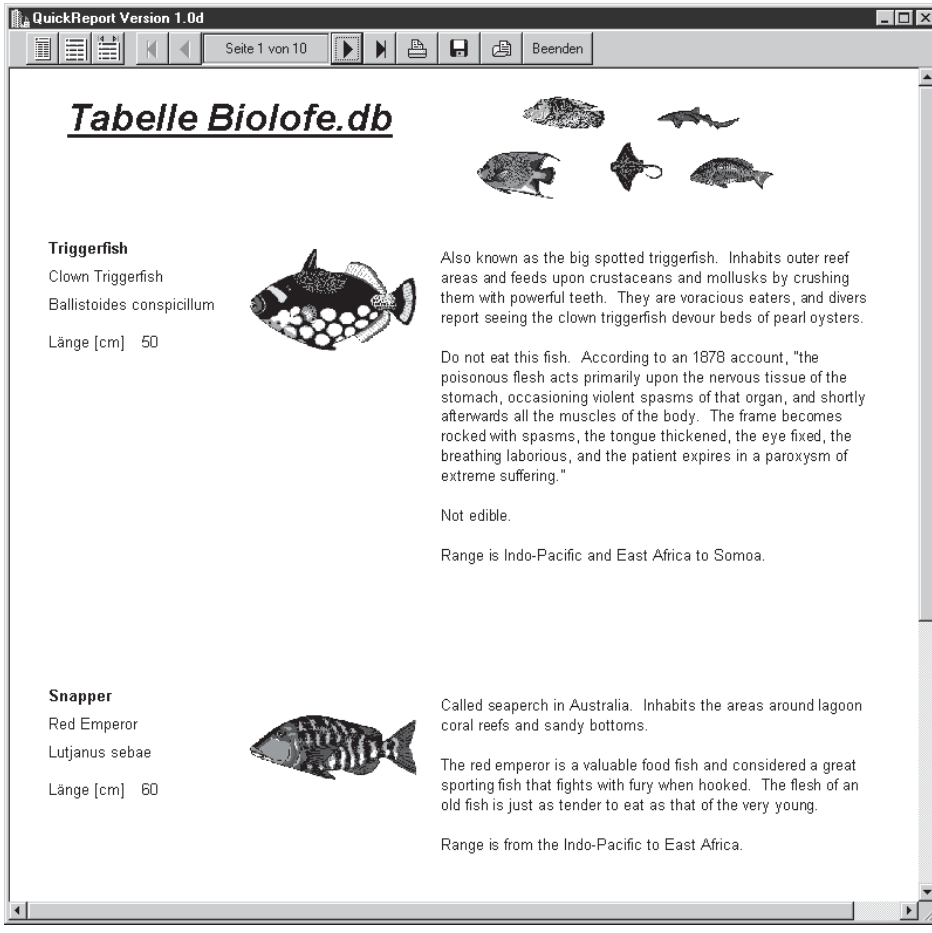


Bild 5.4: Reports mit Bildern und Memos

die Komponente *TQRSysData*. Deren Eigenschaft *Data* wird auf *qrsReportTitle*, *qrsDateTime* bzw. *qrsPageNumber* gestellt. Soll ein Text vorangestellt werden (*Seite 3* statt *3*), dann muß dieser der Eigenschaft *Text* zugewiesen werden. Der Titel des Reports wird übrigens der Eigenschaft *ReportTitle* von *TQuickReport* zugewiesen.

Sie werden sich vielleicht wundern, daß bislang überhaupt kein Quelltext geschrieben wurde; dies ist auch für die Definition eines Reports überhaupt nicht erforderlich.

5.4 Master-Detail-Reports

Mit den *QuickReport*-Komponenten ist es auch möglich, Master-Detail-Reports zu generieren. Wir wollen nun eine Liste der säumigen Kunden erstellen, welche mit den dazugehörenden Außenständen aufgelistet werden. Zum Schluß soll dann noch eine kleine Übersicht über die Gesamtaußenstände unserer fiktiven Firma gebildet werden. Master-Detail-Reports sind wegen der Vielzahl der beteiligten Komponenten ein wenig aufwendiger und fehleranfälliger als einfache Reports; halten Sie sich deshalb genau an die vorgegebenen Anweisungen und deren Reihenfolge.

Noch ein Tip: Um sich einen Bericht anzusehen, muß man nicht die ganze Anwendung neu starten. Ein Doppelklick auf das Icon der *TQuickReport*-Komponente genügt, um den Report (im Standardfenster) anzuzeigen. Nun zu unserem Master-Detail-Report:

Zunächst wird ein neues Formular erstellt. In das Menü von *Form1* wird ein neuer Menüpunkt eingefügt, dessen *OnClick*-Routine vom vorhergehenden Menüpunkt kopiert und dann entsprechend geändert wird. Die Eigenschaft *Font* von *Form6* wird auf einen TrueType-Font geändert, beispielsweise auf *Arial*.

In *Form6* werden eine *TQuery*- und eine *TDataSource*-Komponente eingefügt. Es wird folgende SQL-Anweisung verwendet:

```
SELECT DISTINCT t.nummer,
    t.vornamen || " " || t.nachnamen AS namen,
    t.straße, t.plz || " " || t.ort AS wohnort
  FROM testadr t, offenpo o
  WHERE t.nummer = o.kunde
```

In *Form6* wird eine *TQuickReport*-Komponente eingefügt. Setzen Sie hier die Eigenschaft *DataSource* auf *DataSource1*.

Das erste Band wird eingefügt, die Eigenschaft *BandType* bleibt auf *rbTitle*. In das Band wird eine *TQRLabel*-Komponente eingefügt, deren Eigenschaft *Caption* den Wert *Offene Posten* erhält; die Schrift darf ruhig etwas größer gewählt (24 Punkt) und fett gesetzt werden.

Das zweite Band wird eingefügt, hier wird die Eigenschaft *BandType* auf *rbDetail* abgeändert. Die Namen und Adressen der säumigen Kunden sollen weiß auf blauem Hintergrund geschrieben werden, deshalb wird die Eigenschaft *Color* auf *clBlue* geändert. Um dies später nicht bei allen *TQRDBText*-Komponenten einzeln ändern zu müssen, wird der Eigenschaft *Font.Color clwhite* zugewiesen.

Nun werden vier *TQRDBText*-Komponenten eingefügt, die mit *DataSource1* verbunden werden und jeweils eins der vier Felder anzeigen.

Der »Master« ist nun soweit fertig und kann mit einem Doppelklick auf das Icon von *TQuickReport* gestartet werden. Dabei werden – wie erwartet – die entsprechenden Kunden angezeigt, jedoch noch nicht deren Außenstände.

Die Details

Der Report soll nun um das Detail-Band erweitert werden:

Zunächst werden wieder eine *TQuery*- und eine *TDataSource*-Komponente eingefügt. Um eine Master-Detail-Beziehung zu *Query1* herzustellen, wird die Eigenschaft *DataSource* von *Query2* auf *DataSource1* gesetzt. Die *SQL*-Anweisung von *Query2* lautet dann folgendermaßen:

```
SELECT nummer, datum, betrag
  FROM offenpo
  WHERE kunde = :nummer
```

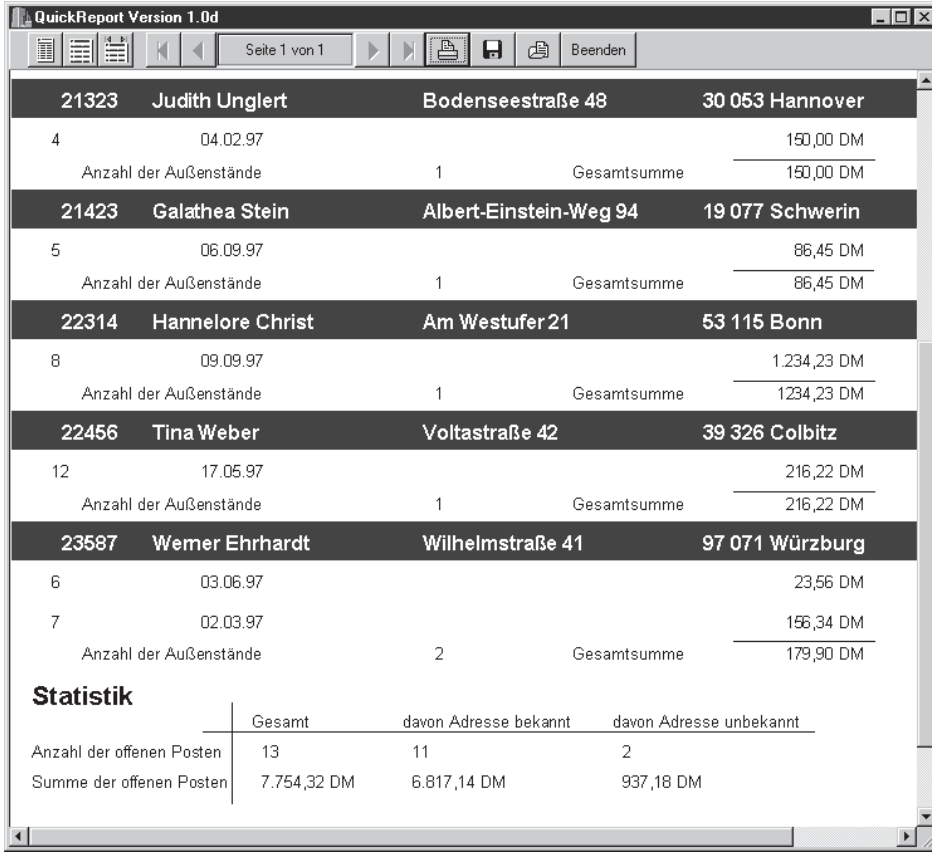


Bild 5.5: Master-Detail-Report

Ein weiteres Band mit der Eigenschaft *BandType* gleich *rbSubDetail* wird eingefügt, auf dieses werden die drei *TQRDBText*-Komponenten zur Anzeige von *Nummer, Datum* und *Betrag* der offenen Posten plaziert.

Ein viertes Band erhält die Eigenschaft *BandType* gleich *rbGroupFooter*; die Eigenschaft *LinkBand* muß auf *QRBand3* gesetzt werden. Neben zwei *TQRLabel*-Komponenten mit den Aufschriften *Anzahl der Außenstände* und *Gesamtsumme* werden zwei *TQRDBCalc*-Komponenten eingefügt. Diese Komponenten werden eingesetzt, um Rechenoperationen mit Datenfeldern durchzuführen und das Ergebnis dann auszugeben.

Zunächst soll die Anzahl der Außenstände (pro Kunde) gezählt werden. Dazu setzt man die Eigenschaften *DataSource* auf *DataSource2*, *DataField* auf eins der Felder (beispielsweise *Nummer*, das ist hier aber wirklich egal) und *Operation* auf *qrcCOUNT*. *QRDBCalc1* zählt nun die Anzahl der Nummern, was der Anzahl der offenen Rechnungen entspricht.

Nun soll die *QRDBCalc1* bei jedem Kunden neu anfangen zu zählen, sonst macht diese Anzeige nicht viel Sinn. Die Eigenschaft *ResetBand* wird deshalb auf *QRBand2* gesetzt, welches das Master-Band ist. *QRDBCalc1* fängt also jedesmal neu an zu zählen, wenn eine neue Kundenadresse ausgegeben wird.

Zur Anzeige des Gesamtbetrags wird *QRDBCalc1* ähnlich konfiguriert; die Eigenschaft *DataField* muß auf *Betrag* geändert werden, und statt der Anzahl soll die Gesamtsumme der Rechnungsbeträge ausgegeben werden Die Eigenschaft *Operation* ist deshalb auf *qrcSUM* zu setzen. Um hier für jeden Kunden ein Summe bilden zu können, wird auch hier die Eigenschaft *ResetBand* auf *QRBand2* gesetzt.

Um die gleiche Formatierung zu erhalten wie bei den Rechnungsbeträgen, wird die Eigenschaft *PrintMask* auf *0.00 DM* gesetzt.

Zuletzt soll noch ein Summenstrich zwischen den Rechnungsbeträgen und der Gesamtsumme eingefügt werden. Dazu wird eine *QRShape*-Komponente eingefügt, deren Eigenschaft *Shape* auf *qrsHorLine* gesetzt wird.

Zuletzt müssen die Detail-Bänder mit dem Master-Band verbunden werden. Dazu dient die Komponente *TQRDetailLink*. Die Eigenschaft *DataSource* wird hier auf *DataSource2* gesetzt, die Eigenschaften *DetailBand* auf *QRBand3* und *FooterBand* auf *QRBand4*; ein *HeaderBand* wäre möglich, wird hier aber nicht benötigt. Die Eigenschaft *Master* wird auf *QuickReport1* gesetzt; es wäre auch möglich (hier nicht, aber generell), als *Master* ein *TQRDetailLink* zu verwenden, um Details zu einem Detail anzuzeigen.

Die Statistik

Zum Schluß wollen wir noch eine Statistik der offenen Posten erstellen. Diese soll die Anzahl und die Gesamtsumme aller Außenstände ermitteln. Dabei soll die Statistik noch zwischen denen mit bekannten und denen mit unbekannten Adressen unterscheiden.

Zunächst wird wieder ein Band eingefügt und dessen Eigenschaft *BandType* auf *rbSummary* geändert; damit erscheint es auf der letzten Seite nach allen Master- und Detail-Bändern, aber noch vor den *rbPageFooter*-Bändern. Es ist auch möglich, dafür eine extra Seite zu verwenden (dies geht bei allen Bändern), indem die Eigenschaft *ForceNewPage* auf *true* gesetzt wird.

- Das Einfügen der Beschriftungen und der Linien werden Sie wohl inzwischen beherrschen.
- Des weiteren werden drei *TQuery-/ TDataSource*-Kombinationen eingefügt; die SQL-Anweisungen lauten folgendermaßen:

```
SELECT COUNT(Betrag), SUM(Betrag)
  FROM offenpo

SELECT COUNT(Betrag), SUM(Betrag)
  FROM offenpo
  WHERE kunde IS NOT NULL

SELECT COUNT(Betrag), SUM(Betrag)
  FROM offenpo
  WHERE kunde IS NULL
```

- Für diese Datenquellen werden nun jeweils zwei *TQRDBText*-Komponenten eingefügt.

5.5 Referenz der QuickReport-Komponenten

In diesem Unterkapitel soll eine kurze Referenz der *QuickReport*-Komponenten gegeben werden.

5.5.1 Die Komponente TQuickReport

Die Komponente *TQuickReport* macht aus einem C++Builder-Formular ein Report-Formular. Sie ist für jedes Report-Formular zwingend erforderlich, jedoch nicht für eigene Vorschauformulare.

TQuickReport wird als Icon in das Formular eingefügt; durch einen Doppelklick auf dieses Icon kann der Report probeweise erstellt werden, ohne daß die eigentlich Anwendung gestartet werden muß.

Die Eigenschaften von TQuickReport

- Mit der Eigenschaft *DataSource* wird festgelegt, aus welcher Datenquelle der Report (oder der Master) seine Daten bezieht. Für gewöhnlich wird der Report mit dem ersten Datensatz begonnen. Soll aber ab dem aktuellen Datensatz begonnen werden, so wird die Eigenschaft *RestartData* auf *false* gesetzt. *QuickReport* verwendet intern die Zahl *RecordCount*, welche sich auch zur Laufzeit abfragen läßt. Wenn eine Client-Server-Datenbank diese Funktion nicht unterstützt und die BDE somit eine Fehlermeldung ausgibt, dann stellen Sie *SQLCompatible* auf *true*.

- Ob ein Report im Hochformat (*poPortrait*) oder im Querformat (*poLandscape*) ausgegeben wird, hängt von der Eigenschaft *Orientation* ab; beachten Sie bitte, daß diese beiden Konstanten in der Datei *Printers.h* definiert werden, die deshalb in die Header-Deklaration aufgenommen werden muß.

  Die Bänder beginnen am linken Druckbereich, also an der Blattkante zuzüglich eines Stegs von meist rund 5 mm, welcher vom Drucker nicht bedruckt werden kann. Größere Ränder wird man meist beim Design der Bänder vorsehen. Es ist jedoch auch möglich, zusätzlich einen linken Rand zu definieren, beispielsweise dann, wenn alle Bänder nach rechts geschoben werden sollen, ohne daß man das ganze Design über den Haufen werfen möchte. Die Breite des zusätzlichen linken Randes kann sowohl in Millimetern (*LeftMarginMM*) als auch in zehntel Inch (*LeftMarginInches*) angegeben werden.

  Für gewöhnlich wird auf der ersten Seite zuerst das Band *PageHeader* und danach das Band *Title* ausgegeben (sofern beide erstellt wurden). Möchte man diese Reihenfolge ändern, dann kann man die Eigenschaft *TitleBeforeHeader* auf *true* setzen.

- Um mehrspaltige Berichte auszudrucken, wird die Zahl der Spalten in der Eigenschaft *Columns* angegeben. Die Stegbreite zwischen den Spalten wird mit *ColumnMarginInches* bzw. *ColumnMarginMM* angegeben.

- Beim Erstellen eines Reports wird eine Fortschrittsanzeige ausgegeben, was auch sehr sinnvoll ist, da so der Anwender weiß, daß die Anwendung nicht abgestürzt ist. Wenn man dies – aus was für Gründen auch immer – vermeiden möchte, dann wird die Eigenschaft *ShowProgress* auf *false* geändert.

- Auf die Eigenschaft *DisplayPrintDialog* kann man bei der Erstellung eigener Vorschaufenster zurückgreifen. Der Eigenschaft *ReportTitle* kann ein Titel des Reports zugewiesen werden, um diesen beispielsweise im Druckmanager auszugeben.

- Mit der Eigenschaft *PaperSize* wird die Seitengröße eingestellt – die Voreinstellung *qrpDefault* kann in der Regel belassen werden. Wird *qrpCustom* gewählt, dann kann mit *PaperSize* und *PaperWidth* die Papiergröße eingestellt werden. Mit Hilfe der Eigenschaft *PageFrame* läßt sich ein Rahmen um die Seite ziehen.
- Des weiteren gibt es zur Laufzeit die Eigenschaften *PageCount*, *PageNumber*, *PageWidth*, *PageHeight*, *RecordCount*, *RecordNumber* sowie *RecordType*, welche die Werte enthalten, die man angesichts ihres Namens von ihnen erwartet.

Die Methoden von TQuickReport

- Mit der Methode *Preview* startet man die Anzeige des Berichts im QuickReport- oder im selbsterstellten Vorschaufenster. Mit *Print* wird der Ausdruck gestartet.
- Mit *NewPage* wird ein Seitenumbruch erzwungen.
- Die Methode *Prepare* wird aufgerufen, wenn man einen Bericht erstellen möchte, ohne ihn automatisch zu drucken oder als Vorschau auszugeben. *Prepare* erstellt den Bericht und sendet ihn an das Objekt *QRPrinter*; er läßt sich dann mit *Preview* oder *Print* aufrufen, wobei die Bearbeitung dann weniger Zeit in Anspruch nimmt. Sie müssen die Methode *QRPrinter->CleanUp* aufrufen, wenn Sie den Bericht wieder aus dem Speicher entfernen wollen.

  Über die Multi-Tasking-Möglichkeiten von C++Builder 2.0 kann man einen Report auf diese Weise im Hintergrund erstellen, während die Anwendung darauf wartet, daß der Benutzer irgendwelche Eingaben macht (bei solchen Arbeiten ist der Rechner meist deutlich unterbeschäftigt). In diesem Fall macht es dann auch Sinn, die Eigenschaft *ShowProgress* auf *false* zu stellen.

Die Ereignisse von TQuickReport

- Die Ereignisse *OnBeforeDetail* oder *OnAfterDetail* werden vor bzw. nach der Erstellung eines Detail-Bandes in einem Report aufgerufen.
- Das Ereignis *OnBeforePrint* wird aufgerufen, wenn ein Bericht zum Druck vorbereitet wird. Die Ereignisbehandlungsroutine enthält als Variablen-Parameter *PrintReport*; wird dieser auf *false* gesetzt, wird der Druck des Berichts abgebrochen. Das Ereignis *OnAfterPrint* wird aufgerufen, nachdem der gesamte Bericht an das Objekt *TQRPrinter* gesandt worden ist.
- Das Ereignis *OnStartPage* wird aufgerufen, wenn eine neue Seite begonnen, das Ereignis *OnEndPage*, wenn eine Seite beendet wird.
- Stammen die Daten nicht aus einer Datenquelle, dann bleibt die Eigenschaft *DataSource* leer, stattdessen wird eine Ereignisbehandlungsprozedur für das *OnNeedData*-Ereignis erstellt.

5.5.2 Die Komponente TQRBand

Die einzelnen Elemente werden bei *QuickReport* auf Bändern, also auf *TQRBand*-Komponenten plaziert.

- Je nach Wert der Eigenschaft *BandType* entscheidet sich, ob die Elemente bei jedem Datensatz oder beispielsweise nur einmal auf der Seite im Fußtext angezeigt werden. Hier gibt es folgende Möglichkeiten:
  - Das Band *rbTitle* wird einmal zu Beginn des Berichts gedruckt. Die Eigenschaft *QuickReport->TitleBeforeHeader* bestimmt, ob zuerst das Band *PageHeader* (Voreinstellung) oder *Title* erscheint. Analog dazu erscheint das Band *rbSummary* am Ende eines Berichts.
  - Auf jeder Seite erscheinen automatisch die *rbPageHeader*(Kopfzeilen)-Bänder, von denen mehrere definiert werden können. Entsprechendes gilt für die *rbPageFooter*(Fußzeilen)-Bänder.
  - Das Band *rbDetail* wird einmal für jeden Datensatz der Haupttabelle ausgedruckt. Hier werden die eigentlichen Report-Daten plaziert.
  - Wenn ein Master-Detail-Report mit Hilfe der Komponente *QRDetailLink* entworfen wird, so stellt das Band *rbSubDetail* das Detail-Band für Ihre Detailtabelle(n) dar.
  - Für *QRGroup*- und *QRDetailLink*-Komponenten werden die Bänder *rbGroupHeader* bzw. *rbGroupFooter* als Gruppenkopf- bzw. Fußzeilen verwendet. Sie müssen mit derartigen Komponenten verbunden sein, um gedruckt zu werden.
  - Bei mehrspaltigen Berichten wird das Band *rbColumnHeader* über jeder Spalte gedruckt.
  - Das Band *rbOverlay* wird einmal pro Seite gedruckt und überlagert den übrigen Text sowie übrige Graphiken der Seite. Auf diese Weise kann man Hinweise einfügen, die beim Photokopieren nicht so einfach vergessen werden können.
- Die Eigenschaft *Enabled* aktiviert oder deaktiviert den Druck des jeweiligen Bandes. Ist diese Eigenschaft *false*, so werden auch die Ereignisse *OnBeforePrint* und *OnAfterPrint* nicht ausgelöst. Man kann diese Eigenschaft dazu verwenden, um die zu erstellenden Bänder erst zur Programmlaufzeit oder sogar ereignisgesteuert beim Erstellen des Berichts festzulegen.
- Wird die Eigenschaft *ForceNewPage* auf *true* gestellt, so wird für dieses Band immer eine neue Seite begonnen.
- Mit der Eigenschaft *Frame* läßt sich ein Rahmen um das Band ziehen. Zur Gestaltung dienen des weiteren die Eigenschaften *Color* und *Font*.

- Die Eigenschaft *LinkBand* stellt sicher, daß zwei oder mehrere Bänder stets auf derselben Seite gedruckt werden.
- Mit der Eigenschaft *Ruler* können zur Entwurfszeit Gitter in Zentimeter- oder Inch-Teilung auf dem Band angezeigt werden. Auf diese Weise kann man die Elemente präziser plazieren, außerdem weiß man so, wieviel Platz man zur Verfügung hat.
- Die Ereignisse *OnBeforePrint* und *OnAfterPrint* werden erwartungsgemäß vor bzw. nach dem Druck des jeweiligen Bandes ausgelöst.

5.5.3 Die Komponente TQRLabel

Um einzeilige unveränderliche Texte auf die Bänder zu bringen, wird die Komponente *TQRLabel* verwendet. Die Komponente gleicht weitgehend der Komponente *TLabel*. Der auszugebende Text wird der Eigenschaft *Caption* zugewiesen. Interessant ist außerdem die Eigenschaft *AlignToBand*; wird sie auf *true* gesetzt, dann bezieht sich die Ausrichtungsangabe der Eigenschaft *Alignment* nicht auf die Komponente selbst, sondern auf das Band. Auf diese Weise können *TQRLabel* beispielsweise am rechten Blattrand plaziert werden; diese Ausrichtung bleibt auch dann erhalten, wenn die Eigenschaft *Orientation* geändert wird. Als Ereignis gibt es hier lediglich *OnPrint*.

5.5.4 Die Komponente TQRMemo

Um mehrzeilige unveränderliche Texte auf die Bänder zu bringen, wird die Komponente *TQRMemo* verwendet, welche an *TMemo* angelehnt ist, jedoch nicht die Möglichkeit bietet, zur Laufzeit Eingaben zu machen. Der Text wird der Eigenschaft *Lines* zugewiesen, Ereignisse gibt es nicht.

5.5.5 Die Komponente TQRDBText

Um den Inhalt von Datenbankfeldern in den Report zu bringen, wird die Komponente *TQRDBText* verwendet, welche große Ähnlichkeiten mit *TQRLabel* hat. Statt der Eigenschaft *Caption* gibt es hier die Eigenschaften *DataSource* und *DataField*. Mit *TQRText* können auch Memos dargestellt werden. Es werden dabei – so benötigt – automatisch weitere Zeilen hinzugefügt.

5.5.6 Die Komponente TQRShape

Die Komponente *TQRShape* dient zur Aufnahme geometrischer Formen in den Report. In der Regel werden dies horizontale oder vertikale Linien sein, es gibt aber auch andere Möglichkeiten. Mit der Eigenschaft *Shape* wird festgelegt, welche Form die Komponente haben soll, mit *Pen* werden Stärke und Farbe der (Umriß-) Linie vorgegeben. Bei gefüllten Objekten (*qrsCircle, qrsRectangle*) kann mit der Eigenschaft *Brush* die Art der Füllung spezifiziert werden.

5.5.7 Die Komponente TQRDBCalc

Bisweilen müssen in einem Report einfache Berechnungen durchgeführt werden, beispielsweise die Summe aller Abteilungsergebnisse. Prinzipiell könnte man dazu eine neue *TQuery/TDataSource*-Kombination verwenden und die Rechenanweisung als SQL-Befehl formulieren. Mit der Komponente *TQRDBCalc* geht es aber in der Regel einfacher. Hier sind folgende Eigenschaften relevant:

- Den Eigenschaften *DataSource* und *DataField* wird zugewiesen, welche Tabellenspalte für die Berechnung verwendet werden soll.
- Mit der Eigenschaft *Operation* spezifiziert man die Rechenoperation, welche anzuwenden ist. Hier gibt es folgende Möglichkeiten:
  - SUM bildet die Summe der Feldwerte.
  - AVG bildet den Mittelwert der Feldwerte.
  - MIN und MAX ermittelt das Minimum bzw. Maximum der Feldwerte.
  - COUNT zählt die Anzahl der Felder mit einem Eintrag.
- Um den Zahlenwert formatiert darzustellen, wird die Eigenschaft *PrintMask* verwendet. Wird hier beispielsweise der String *0.00 DM* eingegeben, dann wird die Zahl mit zwei Nachkommastellen ausgegeben, zusätzlich werden ein Leerzeichen und der String *DM* angehängt.
- Nicht immer sollen die Rechenoperationen über den gesamten Bericht gebildet werden; bei Master-Detail-Reports beispielsweise wird man oft Rechenoperationen nur über die Daten der jeweiligen Detail-Datengruppe bilden wollen. Hier gibt es die Möglichkeit, der Eigenschaft *ResetBand* einen Wert zuzuweisen; jedesmal beim Einfügen eines neuen Bandes dieses Typs werden alle Rechenoperationen abgeschlossen und neu initialisiert. Beim erwähnten Master-Detail-Report wird man als *ResetBand* das jeweilige Master-Band verwenden.
- Nur während der Laufzeit und nur zum Lesen stehen die Eigenschaften *AsInteger* und *AsReal* zur Verfügung, welche den ausgegebenen String in eine Integer- oder Gleitkommazahl gewandelt zur Verfügung stellen.

- Außerdem steht das Ereignis *OnPrint* zur Verfügung. Wir wollen den Master-Detail-Report aus Kapitel 5.4 so erweitern, daß alle Summen rot dargestellt werden, die den Betrag von 500,- DM übersteigen. Dazu muß lediglich folgende Anweisung eingefügt werden:

```
void __fastcall TForm6::QRDBCalc2Print(TObject *sender,
  AnsiString &Value)
{
  if(QRDBCalc2->AsReal > 500)
    QRDBCalc2->Font->Color = clRed;
  else
    QRDBCalc2->Font->Color = clBlack;
}
```

5.5.8 Die Komponente TQRSysData

Die Komponente *TQRSysData* dient zur Ausgabe von *QuickReport*-internen Daten. Hier sind lediglich zwei Eigenschaften relevant, ansonsten gleicht diese Komponente *TQRLabel*:

- Die Eigenschaft *Data* spezifiziert, welcher Wert ausgegeben werden soll. Hier gibt es die folgenden Möglichkeiten:
  - *qrsTime* gibt die aktuelle Zeit aus.
  - *qrsDate* gibt das aktuelle Datum aus.
  - *qrsDateTime* gibt Zeit und Datum aus.
  - *qrsPageNumber* liefert die aktuelle Seitenzahl.
  - *qrsReportTitle* liefert die Berichtsüberschrift, wie sie in der Eigenschaft *TQuickReport->ReportTitle* vorgegeben wird.
  - *qrsDetailCount* beinhaltet die Anzahl der im Bericht vorhandenen Datensätze.
  - *qrsDetailNo* liefert die Nummer des aktuellen Datensatzes.
- Die Eigenschaft *Text* ermöglicht es, dem eigentlichen Wert noch einen (erläuternden) Text voranzustellen; die Ausgabe lautet dann beispielsweise nicht *7*, sondern *Seite 7*.

5.5.9 Die Komponente TQRDetailLink

Um Master-Detail-Reports zu erstellen, müssen nicht nur die entsprechenden *DataSet*- und *DataSource*-Komponenten eingefügt werden, sondern es muß auch zwischen den *rbDetail*- und den *rbSubDetail*-Bändern eine Verbindung hergestellt werden. Dazu dient die Komponente *TQRDetailLink*. Hier sind folgende Eigenschaften und Ereignisse erwähnenswert:

- Die Eigenschaft *DataSource* verweist auf die Datenquelle, aus der das *QRDetailLink*-Band seine Daten bezieht. Achten Sie darauf, daß Sie eine funktionierende Master-Detail-Verbindung der *DataSet*-Komponenten selbst erstellen müssen.
- Die Eigenschaft *DetailBand* verweist auf das Band, in welchem die Detail-Daten ausgegeben werden; es wird jeweils ein Band pro Detail-Datensatz verwendet, dessen Eigenschaft *BandType* wird auf *rbSubDetail* gesetzt.
- Die Detail-Bänder können um ein Kopf- und ein Fuß-Band ergänzt werden; diese werden in der Eigenschaft *HeaderBand* bzw. *FooterBand* spezifiziert, die Eigenschaft *BandType* des Kopf- bzw. Fußbandes wird auf *rbGroupHeader* bzw. *rbGroupFooter* gesetzt.
- Die Eigenschaft *Master* wird auf diejenige Komponente gesetzt, welche der jeweiligen Detail-Tabelle übergeordnet ist. Bei einfachen Master-Detail-Reports wird dies die Komponente *TQuickReport* sein, werden mehrere Detail-Ebenen verwendet, so kann dies auch eine (andere) *TQRDetailLink*-Komponente sein.
- Bevor das Detail-Band in den Report eingefügt wird, wird das Ereignis *OnFilter* ausgelöst. Dies ermöglicht es, den übergebenen Variablen-Parameter *PrintRecord* auf *false* zu setzen, damit das entsprechende Band nicht in den Report eingefügt wird.
- Des weiteren gibt es hier das Ereignis *OnNeedData*; mit der entsprechenden Routine können vom Programm Daten eingefügt werden, welche nicht aus *DataSource* stammen.

5.5.10 Die Komponente TQRGroup

Die Komponente *TQRGroup* dient – wie der Name bereits erahnen läßt – zur Gruppierung der Daten. Wir wollen die Adressenliste unseres Beispielprogramms nun mit Hilfe dieser Komponente den Nachnamen nach gruppieren. Zunächst wird die SQL-Anweisung von *Query1* so ergänzt, daß die Datensätze den Nachnamen nach sortiert angezeigt werden.

```
SELECT * FROM testadr
  WHERE ort = 'Altshausen'
  ORDER BY nachname
```

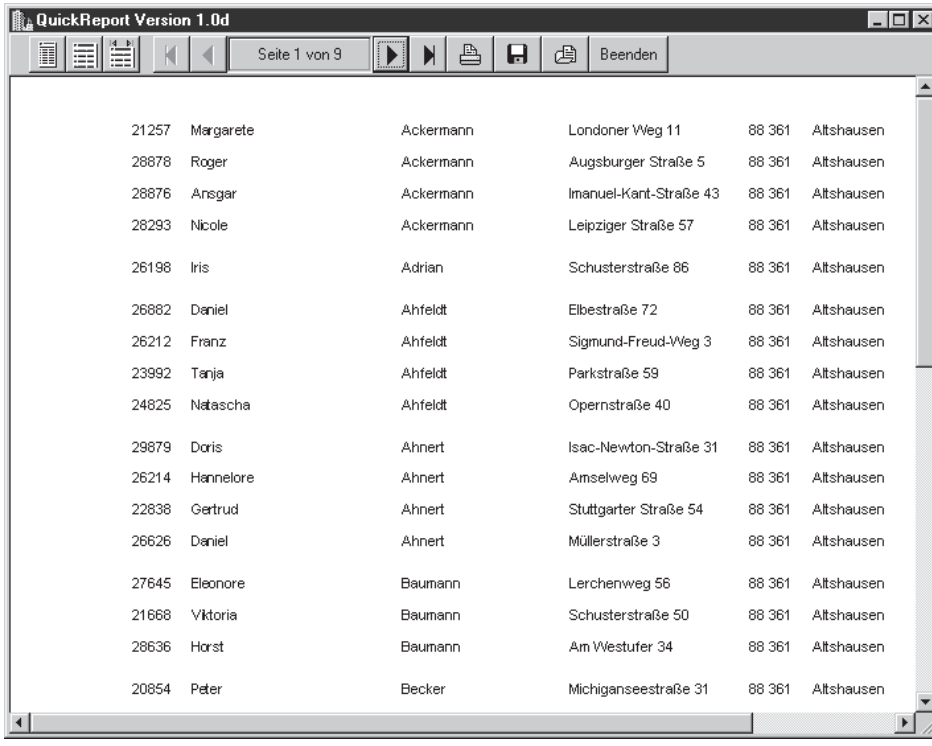


Bild 5.6: Gruppieren nach dem Nachnamen

Nun werden in den Report die Komponenten *QRGoup1* sowie *QRBand3* eingefügt. Die Eigenschaften von *QRGroup1* werden wie folgt gesetzt:

- Die Eigenschaft *FooterBand* wird auf *QRBand3* gesetzt. Im Anschluß an jede Gruppe wird somit dieses Band ausgegeben.
- Die Eigenschaft *DataSource* wird auf *DataSource1* und die Eigenschaft *DataField* auf *Nachname* gesetzt. Somit wird immer dann, bevor ein neuer Nachname in der Liste erscheint, *QRBand3* ausgegeben.

Gruppierung anhand der Initiale

Als nächstes sollen die Datensätze nicht nach Nachnamen, sondern nach den Anfangsbuchstaben der Nachnamen gruppiert werden. Zu diesem Zweck werden wir zunächst das Ereignis *OnNeedData* einsetzen. Zunächst werden die Inhalte der Eigenschaften *DataSource* und *DataField* gelöscht, dann wird die folgende Ereignisbehandlungsroutine erstellt.

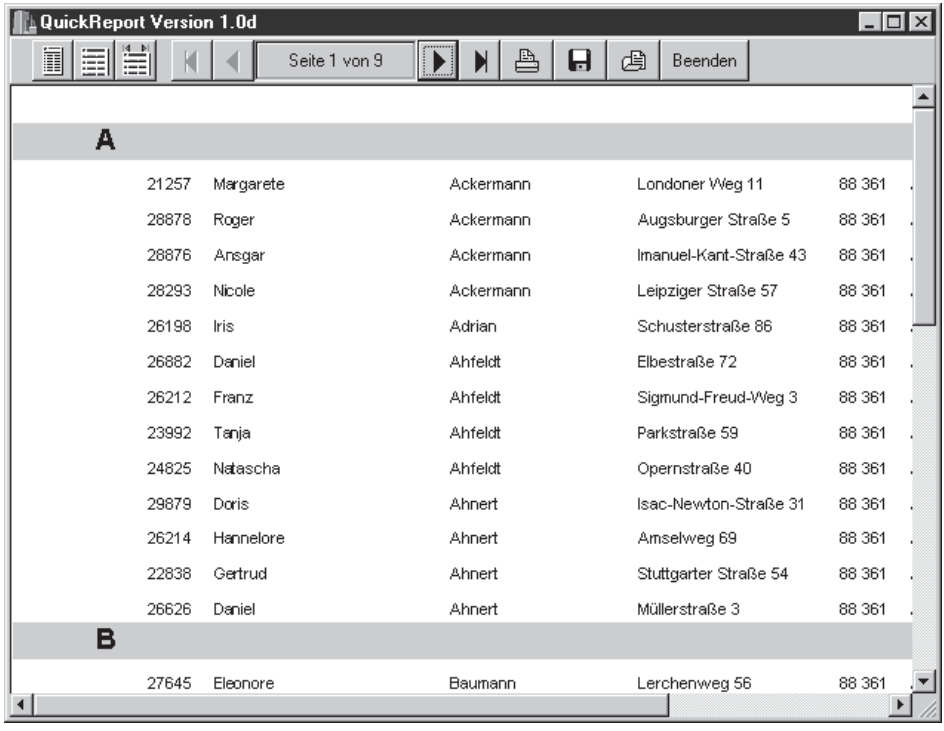


Bild 5.7: Gruppieren nach der Initiale

```
void __fastcall TForm2::QRGroup1NeedData(AnsiString &Value)
{
  Value = Query1->FieldByName("Nachname")->AsString[1];
}
```

Hier wird dem Parameter *Value* der erste Buchstabe des Feldes *Nachname* zugewiesen. Nun sollen, wie in Bild 5.7 zu sehen, die jeweiligen Initialen auf einem Header-Band angezeigt werden. Hier muß dann etwas anders vorgegangen werden. Zunächst wird die *OnNeedData*-Routine wieder gelöscht, dafür wird ein berechnetes Feld mit dem Namen *Initiale* erzeugt.

```
void __fastcall TForm2::Query1CalcFields(TDataSet *DataSet)
{
  Query1->FieldByName("Initiale")->AsString
    = Query1->FieldByName("Nachname")->AsString;
}
```

Nun werden die entsprechenden *QRGroup1*-Eigenschaften auf *DataSource1* und *Initiale* gesetzt, zudem wird *QRBand3* nun als Header- statt als Footer-Band ver-

wendet. Mit Hilfe der Komponente *QRDBText7* wird die Initiale auf dem Header-Band angezeigt.

5.5.11 Die Komponente TQRPreview

Um eigene Vorschau-Formulare erstellen zu können, wurde die Komponente *TQRPreview* geschaffen. Wie man damit ein eigenes Vorschau-Formular erstellt, wurde schon in Unterkapitel 5.2.1 ausführlich beschrieben. Hier sind folgende Eigenschaften, Methoden und Ereignisse vorhanden:

- Der Eigenschaft *PageNumber* kann zugewiesen werden, welche Seite angezeigt werden soll.
- Der Eigenschaft *Zoom* wird der Vergrößerungsfaktor in Prozent zugewiesen. Zur Darstellung in Originalgröße wird *Zoom* auf *100* gesetzt.
- Mit der Methode *ZoomToWidth* wird der Vergrößerungsfaktor so gesetzt, daß Seitenbreite und Formularbreite übereinstimmen, während mit *ZoomToFit* die ganze Seite dargestellt wird.
- Das Ereignis *OnShow* tritt nur in der Online-Hilfe auf, dafür gibt es aber dieselben Ereignisse wie bei *TScrollBox*.

5.5.12 Das Objekt TQRPrinter

Das Objekt *TQRPrinter* kapselt eine Menge von Eigenschaften und Methoden, welche zum Ausdrucken des Berichts erforderlich sind. Die meisten davon werden Sie wahrscheinlich nie benutzen, gegebenenfalls können Sie in der Online-Hilfe nachsehen.

Eine Instanz von *TQRPrinter* wird erstellt, wenn die Anwendung gestartet wird; es ist also nicht nötig, selbst eine Variable von *TQRPrinter* zu instantisieren. Folgende Eigenschaften, Methoden und Ereignisse sind relevant:

- Die Eigenschaft *FromPage* gibt an, ab welcher Seite gedruckt werden soll, die Eigenschaft *ToPage*, bis zu welcher Seite. In Zusammenarbeit mit einem *PrintDialog* bietet sich folgende Zuweisung an:

```
QRPrinter->FromPage := PrintDialog1->FromPage;
QRPrinter->ToPage := PrintDialog1->ToPage;
```

- Die Eigenschaft *Orientation* legt fest, ob der Report im Hoch- oder im Querformat gedruckt wird. Es bietet sich an, dies der *QuickReport->Orientation* anzupassen, auch wenn die jeweils andere Ausrichtung möglich wäre (was aber doch etwas seltsam aussieht):

```
QRPrinter->Orientation := Form4->QuickReport1->Orientation;
```

- Der Eigenschaft *Title* kann ein String zugewiesen werden, der dann für diesen Druck-Job im Druck-Manager angezeigt wird.
- Mit der Eigenschaft *Thumbs* können mehrere Report-Seiten auf einer Druck-Seite zusammengefaßt werden. Wird *Thumbs* auf *zwei* gesetzt, dann werden vier Report-Seiten auf einer Druckseite gedruckt, bei *drei* neun usw.
- Anhand der Ereignisbehandlungsroutine von *OnPreview* wird festgelegt, ob die Voransicht des Reports im Standardformular oder in einem selbstdefinierten Formular erfolgt. Soll nach einer Änderung wieder das Standardformular verwendet werden, dann lautet die Anweisung:

  ```
  QRPrinter->OnPreview := NULL;
  ```

  Soll ein selbsterstelltes Formular verwendet werden, dann muß eine separate Aufrufprozedur erstellt werden, welche das entsprechende Formular modal aufruft:

  ```
  QRPrinter->OnPreview = ShowLabels;


  void __fastcall TForm1::ShowLabels(void)
  {
    Form4->ShowModal();
  }
  ```

- Mit der Methode *Save* kann ein Report gespeichert werden (um ihn beispielsweise zu einem späteren Zeitpunkt auszudrucken), mit *Load* kann er geladen werden.
- Die Methode *Print* druckt den Report aus. Mit der Methode *Cancel* kann der Ausdruck abgebrochen werden.

5.5.13 Das Objekt TQRCustomControl

Das Objekt *TQRCustomControl* ist eine Basisklasse für alle druckbaren *QuickReport*-Komponenten. Wenn Sie eigene *QuickReport*-Komponenten entwickeln wollen, müssen Sie diese von *TQRCustomControl* ableiten. Näheres dazu erfahren Sie in der Online-Hilfe.

6 Weitere Objekte

In diesem Kapitel sollen einige weniger bedeutende Datenzugriffskomponenten behandelt werden, nämlich *TSession*, *TUpdateObject* und *TBatchMove*. Zunächst wollen wir aber besprechen, wie man mit Hilfe des Objektes *TPrinter* Daten ausdrucken kann.

6.1 Daten drucken

Zum Drucken der Daten verwendet man für gewöhnlich die *QuickReport*-Komponenten. Hin und wieder treten auch Anforderungen auf, die sich mit diesen beiden Report-Generatoren nicht oder nur mit großem Aufwand erfüllen lassen. Der Ausdruck über das C++Builder-Objekt *TPrinter* ist zwar ebenfalls recht aufwendig, aber auch sehr flexibel. In unserem Beispielprojekt *Tourplaner* wird deshalb der Ausdruck mit Hilfe von *TPrinter* durchgeführt.

6.1.2 Drucken mit den TCanvas-Methoden

Die *TCanvas*-Methoden erlauben die exakte Positionierung der Textstellen. Auch können hier Linien und andere geometrische Formen gedruckt werden. Mit den *TCanvas*-Methoden kann man im übrigen nicht nur auf den Drucker, sondern auch auf den Bildschirm oder auf Graphik-Dateien zugreifen. Die folgende Prozedur zeigt die erforderlichen Anweisungen:

```
void __fastcall TChild::Drucken1Click(TObject *Sender)
{
  TFontStyles Style;
  Printer()->Orientation = poPortrait;
  Printer()->Title = "Druckertest";
  Printer()->BeginDoc();
  Printer()->Canvas->MoveTo(1,1);
  SetMapMode(Handle, MM_LOMETRIC);
  Printer()->Canvas->Font->Name = "Arial";
  Printer()->Canvas->Font->Size = 50;
  Style << fsBold;
```

```
  Printer()->Canvas->Font->Style = Style;
  Printer()->Canvas->TextOut(300, 400, "Test fett");
  Style << fsItalic;
  Printer()->Canvas->Font->Style = Style;
  Printer()->Canvas->TextOut(300, 800, "Test kursiv und fett");
  Style >> fsItalic;
  Style >> fsBold;
  Printer()->Canvas->Font->Style = Style;
  Printer()->Canvas->TextOut(300, 1200, "Test normal");
  Printer()->EndDoc();
} // TChild::Drucken1Click
```

Mit der *Printer*-Eigenschaft *Orientation* legen Sie fest, ob Sie das Blatt hoch (*poPortrait*) oder quer (*poLandscape*) bedrucken möchten. Der Text, den sie *Printer()->Title* zuweisen, wird im Druck-Manager sowie bei einigen Druckern im Display angezeigt. Mit der Methode *BeginDoc* wird der Ausdruck gestartet.

Alle Canvas-Methoden verwenden normalerweise die Maßeinheit Pixel, was bei der Ausgabe auf dem Bildschirm durchaus sinnvoll ist. Nun möchte man aber, daß der Ausdruck auf allen Druckern in etwa gleich aussieht, egal, ob die Auflösung des Druckers 300 dpi oder 1200 dpi beträgt. Erfreulicherweise besteht die Möglichkeit, mit der Anweisung *SetMapMode* eine andere Maßeinheit einzustellen, in diesem Fall 1/10 Millimeter (*MM_loMetric*). Damit von *Printer()->Canvas* schon ein *Handle* existiert, muß zuvor irgendeine *TCanvas*-Methode aufgerufen werden, in unserem Fall *MoveTo*.

Als Schriftgröße wird hier 50 Punkt verwendet, des weiteren werden die Texte mit unterschiedlichen Formatierungen gedruckt. Der *TCanvas*-Methode *TextOut* übergeben Sie die Koordinaten, an denen gedruckt werden soll, sowie den zu druckenden Text als AnsiString. Abschließend muß die Methode *EndDoc* aufgerufen werden.

6.2 Die Komponente TBatchMove

Es gibt Situationen, in denen große Datenmengen von einer Tabelle in eine andere kopiert werden müssen.

- Wird von einer Desktop-Datenbank auf ein Client-Server-System umgestellt (der Fachausdruck dafür lautet *Upsizing*), dann müssen die Daten auf den Server übertragen werden.
- Umgekehrt gibt es auch Situationen, in denen eine Tabelle von einem Server aus in eine Desktop-Datenbank kopiert werden soll, beispielsweise, weil ein Außendienstmitarbeiter die Daten auf seinem Notebook benötigt.

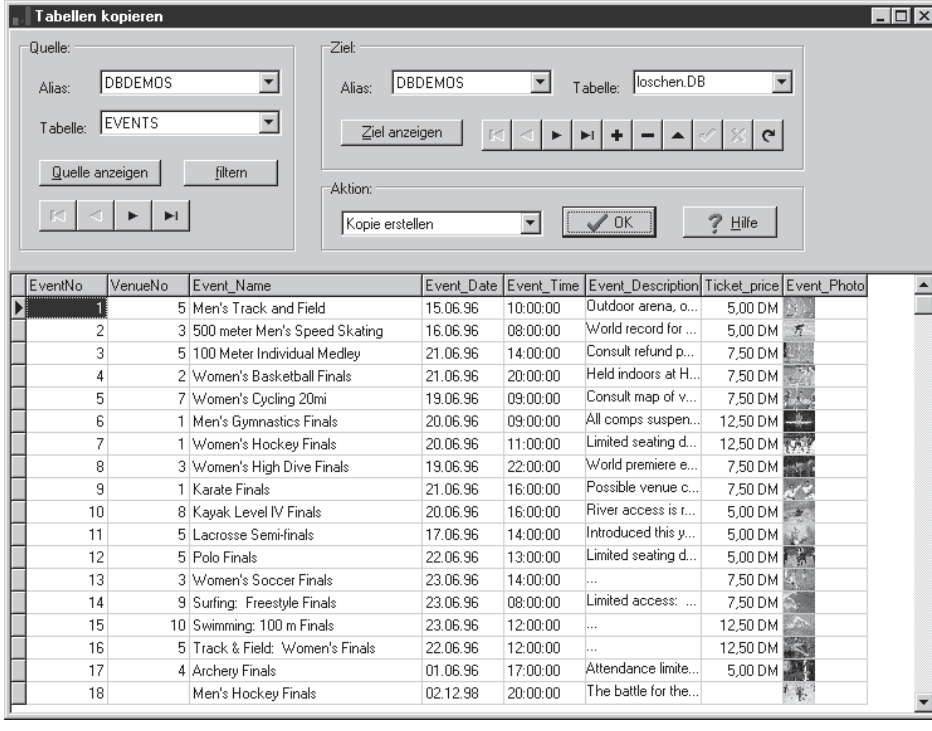
- Manchmal sollen nur Teile eines Datenbestandes weitergegeben werden, beispielsweise alle bayrischen Adressen an die dortige Filiale.
- Umgekehrt müssen auch Daten von Außendienstmitarbeitern und Filialen auf den Server transferiert werden.

Für diese Operationen gibt es die Komponente *TBatchMove*, mit deren Hilfe Datenmengen in Tabellen geändert, gelöscht oder auch ergänzt werden können.

6.2.1 Ein Datentransfer-Programm

Mit der Komponente *TBatchMove* soll nun eine kleine Beispielanwendung programmiert werden. Die Bedienung ist im Prinzip ganz simpel: Man wählt die Quelle und das Ziel, legt die gewünschte Batch-Operation fest und betätigt den Button *OK*.

Damit man nicht die Übersicht verliert, in welcher Tabelle sich nun welche Daten befinden, kann mit *DBGrid1* wahlweise die Quell- oder die Zieltabelle angezeigt werden.



EventNo	VenueNo	Event_Name	Event_Date	Event_Time	Event_Description	Ticket_price	Event_Photo
1	5	Men's Track and Field	15.06.96	10:00:00	Outdoor arena, o...	5,00 DM	
2	3	500 meter Men's Speed Skating	16.06.96	08:00:00	World record for ...	5,00 DM	
3	5	100 Meter Individual Medley	21.06.96	14:00:00	Consult refund p...	7,50 DM	
4	2	Women's Basketball Finals	21.06.96	20:00:00	Held indoors at H...	7,50 DM	
5	7	Women's Cycling 20mi	19.06.96	09:00:00	Consult map of v...	7,50 DM	
6	1	Men's Gymnastics Finals	20.06.96	09:00:00	All comps suspen...	12,50 DM	
7	1	Women's Hockey Finals	20.06.96	11:00:00	Limited seating d...	12,50 DM	
8	3	Women's High Dive Finals	19.06.96	22:00:00	World premiere e...	7,50 DM	
9	1	Karate Finals	21.06.96	16:00:00	Possible venue c...	7,50 DM	
10	8	Kayak Level IV Finals	20.06.96	16:00:00	River access is r...	5,00 DM	
11	5	Lacrosse Semi-finals	17.06.96	14:00:00	Introduced this y...	5,00 DM	
12	5	Polo Finals	22.06.96	13:00:00	Limited seating d...	5,00 DM	
13	3	Women's Soccer Finals	23.06.96	14:00:00	...	7,50 DM	
14	9	Surfing: Freestyle Finals	23.06.96	08:00:00	Limited access: ...	7,50 DM	
15	10	Swimming: 100 m Finals	23.06.96	12:00:00	...	12,50 DM	
16	5	Track & Field: Women's Finals	22.06.96	12:00:00	...	12,50 DM	
17	4	Archery Finals	01.06.96	17:00:00	Attendance limite...	5,00 DM	
18		Men's Hockey Finals	02.12.98	20:00:00	The battle for the...		

Bild 6.1: Ein Datentransfer-Programm mit TBatchMove

Es besteht zudem die Möglichkeit, die Quelldatenbank mit Hilfe einer SQL-Anweisung zu filtern. Das SQL-Eingabefenster wurde dabei aus dem Beispielprojekt *Tourplaner* kopiert. Es ist in Kapitel 10 beschrieben und soll hier nicht vorweggenommen werden.

Auswählen der Datenbanken und der Tabellen

Mit Hilfe von vier *TComboBox*-Komponenten sollen die Datenbank und die Tabelle ausgewählt werden. Bei der *Quelle* wird damit die SQL-Anweisung *SELECT * FROM (Tabellennamen)* erstellt. Soll hier der Datenbestand gefiltert werden oder soll ein JOIN über mehrere Tabellen erstellt werden, dann muß eine entsprechende SQL-Anweisung manuell eingegeben werden.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Session->GetAliasNames(ComboBox1->Items);
  Session->GetAliasNames(ComboBox3->Items);
}
```

Beim Starten des Programms werden alle vorhandenen Datenbank-Aliase den entsprechenden *ComboBoxen* als Einträge zugewiesen.

```
void __fastcall TForm1::ComboBox1Change(TObject *Sender)
{
  Query1->Close();
  Query1->DatabaseName = ComboBox1->Text;
  Session->GetTableNames(ComboBox1->Text,
    "", false, false, ComboBox2->Items);
  ComboBox2->Text = "";
}
```

Wird ein Alias ausgewählt, dann werden die Tabellennamen dieser Datenbank als Einträge der jeweiligen *ComboBox* und der *DataSet*-Komponente zugewiesen. Die Prozedur *ComboBox3Change* muß dabei entsprechend angepaßt werden.

```
void __fastcall TForm1::ComboBox2Change(TObject *Sender)
{
  Query1->Close();
  Query1->SQL->Clear();
  Query1->SQL->Add("SELECT * FROM " + ComboBox2->Text);
  Query1->Open();
}
```

Wird eine Quelltabelle ausgewählt, dann wird eine entsprechende SQL-Anweisung generiert und die Abfrage geöffnet. Bei der Prozedur *ComboBox4Change* muß

der ausgewählte Tabellenname der Eigenschaft *TableName* von *Table1* zugewiesen werden.

```
void __fastcall TForm1::ComboBox5Change(TObject *Sender)
{
  if(ComboBox5->ItemIndex == 0)
    BatchMove1->Mode = batAppend;
  if(ComboBox5->ItemIndex == 1)
    BatchMove1->Mode = batUpdate;
  if(ComboBox5->ItemIndex == 2)
    BatchMove1->Mode = batAppendUpdate;
  if(ComboBox5->ItemIndex == 3)
    BatchMove1->Mode = batDelete;
  if(ComboBox5->ItemIndex == 4)
    BatchMove1->Mode = batCopy;
}
```

Mit *ComboBox5* kann man auswählen, welche Operation die Komponente *BatchMove1* durchführen soll, die Eigenschaft *Mode* wird entsprechend gesetzt.

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
  if(ComboBox5->ItemIndex == 4)
  {
    if(ComboBox4->Items->IndexOf(ComboBox4->Text) > -1)
    {
      int i = Application->MessageBox(
        "Tabelle existiert bereits! Überschreiben?",
        "Warnung", MB_YESNOCANCEL);
      if(i==6)
        BatchMove1->Execute();
    }
    else // if(ComboBox4->Items->IndexOf(ComboBox4->Text) > -1)
      BatchMove1->Execute();
  } // if(ComboBox5->ItemIndex == 4)
  else
    BatchMove1->Execute();
  Query1->Close();
  Query1->DatabaseName = ComboBox1->Text;
  Session->GetTableNames(ComboBox1->Text,
    "", false, false, ComboBox2->Items);
  ComboBox2->Text = "";
```

```
  Table1->Close();
  Table1->DatabaseName = ComboBox3->Text;
  Session->GetTableNames(ComboBox3->Text,
    "", true, false, ComboBox4->Items);
  ComboBox4->Text = "";
} // TForm1::BitBtn1Click
```

Im Prinzip braucht zum Starten der Stapeloperation nur die Methode *Execute* aufgerufen zu werden, lediglich bei der Operation *batCopy* gibt es ein kleines Problem: Als einzige Operation erlaubt *batCopy* die Verwendung eines noch nicht existierenden Tabellennamens – die Tabelle wird dann von der Stapeloperation angelegt. Als einzige der fünf *TComboBox*-Komponenten hat *ComboBox4* den Wert *csDrowDown* und erlaubt somit die Eingabe von Tabellennamen, die noch nicht vorhanden sind.

Nun könnte es allerdings vorkommen, daß der Anwender zufälligerweise den Namen einer bereits existierenden Tabelle eingibt, welche dann überschrieben werden würde. Für diesen Fall ist eine Sicherheitsabfrage vorgesehen.

Für den Fall, daß eine neue Tabelle erstellt wird, werden die Tabellennamen neu in die Combo-Boxen geschrieben.

6.2.2 Die Eigenschaften von TBatchMove

Im Folgenden soll eine kurze Übersicht über die wesentlichen Eigenschaften von *TBatchMove* gegeben werden:

- Die Eigenschaft *Source* spezifiziert die *DataSet*-Komponente (*TTable* oder *TQuery*), von welcher aus die Daten übernommen werden.
- In *Destination* wird die *TTable*-Komponente angegeben, in deren Tabelle die Daten eingefügt, geändert oder gelöscht werden. Hier kann keine *TQuery*-Komponente verwendet werden. Die Zieltabelle muß (bei *Mode* = *batCopy* kann) dabei schon existieren.
- Welche Stapeloperation ausgeführt wird, hängt von der Eigenschaft *Mode* ab. Hier sind folgende Einträge möglich:
  - *batAppend* hängt die Datensätze an die Zieltabelle an.
  - *batUpdate* ändert die Datensätze in der Zieltabelle gemäß den Einträgen in der Quelltabelle. Die Zieltabelle muß dabei über einen Index verfügen, damit die Datensätze zugeordnet werden können.
  - Bei *batAppendUpdate* werden die Datensätze, deren Indexwert in der Zieltabelle schon vorhanden sind, geändert, alle anderen Datensätze werden angehängt.

- *batCopy* erstellt eine exakte Kopie der Quelltabelle. Existiert die Zieltabelle schon, dann wird sie vollständig überschrieben.
- Bei *batDelete* werden alle Datensätze aus der Zieltabelle gelöscht, die mit Datensätzen der Quelltabelle übereinstimmen. Die Zieltabelle muß dabei über einen Index verfügen.

- Für gewöhnlich müssen Quell- und Zieltabelle in der Struktur übereinstimmen – der Wert von Feld 1 der Quelltabelle wird in Feld 1 der Zieltabelle eingefügt, der Wert von Feld 2 in Feld 2 usw. Hin und wieder wird es allerdings vorkommen, daß man eine Stapeloperation mit zwei Tabellen durchführen muß, deren Strukturen nicht identisch sind. In solchen Fällen kann dann eine Zuordnungsliste erstellt werden, die der Eigenschaft *Mappings* zugewiesen wird.

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
    BatchMove1->Mappings->Clear();
    BatchMove1->Mappings->Add("Nummer=Ku_Nr");
    BatchMove1->Mappings->Add("Namen=Namen_1");
}
```

  Stimmen die Feldtypen nicht überein, dann versucht *TBatchMove*, so weit wie möglich zu konvertieren. Felder, die nicht in die Zuordnungsliste aufgenommen werden, setzt *TBatchMove* auf NULL.

- Bei einer Stapeloperation können alle möglichen Fehler auftreten. Mit den Eigenschaften *AbortOnKeyViol* und *AbortOnProblem* – beide vom Typ *boolean* – legen Sie fest, ob bei Schlüsselverletzungen oder anderen Problemen die Stapeloperation abgebrochen oder fortgeführt werden soll. Für den Fall, daß sie fortgeführt werden soll, können Sie den Eigenschaften *KeyViolTableName* und *ProblemTableName* jeweils eine *TTable*-Komponente zuweisen; dort werden dann alle aufgetretenen Problemfälle gespeichert, damit sie später manuell nachgearbeitet werden können.

Die Komponente *TBatchMove* verfügt über einige weitere Eigenschaften, die ich jedoch für nicht wesentlich halte. Wenn Sie eine Eigenschaft benötigen, welche hier nicht aufgeführt ist, dann ist es zu empfehlen, die Online-Hilfe zu konsultieren.

6.2.3 Die Methoden von TBatchMove

Neben den *TObject*-Methoden *Create*, *Destroy* und *Free* gibt es bei *TBatchMove* nur die Methode *Execute*, welche die Stapeloperation startet. Ereignisse gibt es bei *TBatchMove* überhaupt keine.

6.3 Die Komponente TSession

Sobald das Formular um eine Datenzugriffskomponente ergänzt wird, wird automatisch die Komponente *TSession* eingefügt. Die dabei von *TSession* abgeleitete Instanz heißt *Session*. Wenn Sie eine Multi-Thread-Anwendung entwickeln, bei der mehrere Threads gleichzeitig auf Datenbanken zugreifen, dann müssen Sie weitere Instanzen von *TSession* einfügen, von daher ist die Komponente *TSession* in der Komponentenpalette enthalten. Wenn Sie keine Multi-Thread-Anwendung programmieren, dann sollten Sie auch nie eine *TSession*-Komponente in ein Formular oder ein Datenmodul einfügen.

Einen vollständigen Überblick über die Eigenschaften und Methoden der Komponente *TSession* erhalten Sie über die Online-Hilfe. Hier sollen nur die zwei wichtigsten Themen angesprochen werden.

Paßwort

Tabellen mit sensiblen Daten wird man in der Regel mit einem Paßwort sichern, welches beim Start der Anwendung dann eingegeben werden muß. Nun kann eine Datenbank – insbesondere dann, wenn sie normalisiert ist – ohne weiteres über 20 Tabellen enthalten. Sind diese alle mit verschiedenen Paßwörtern gesichert, dann muß der Anwender nach dem Start der Anwendung erst einmal entsprechend viele Paßwörter eingeben.

Erfreulicherweise besteht hier die Möglichkeit, alle Tabellen mit dem gleichen Paßwort zu versehen, so daß ein einmaliges Eingeben genügt – der Anwender hat dann aber Zugriff auf alle Tabellen. Oft möchte man auch vermeiden, daß der Anwender direkt auf die Tabellen zugreifen kann – beispielsweise mit der Datenbankoberfläche – und auf diese Weise Sicherungssysteme der Datenbankanwendung umgeht. Hier besteht nun die Möglichkeit, daß für die Anwendung ein anderes Paßwort vergeben wird, mit dem der Anwender zwar auf das Programm, nicht aber direkt auf die Tabellen zugreifen kann.

Es ist auch nicht gerade angenehm, bei der Entwicklung von Datenbankanwendungen bei jedem Programmstart erneut das Paßwort eingeben zu müssen. Wie Sie sehen, gibt es viele Gründe dafür, daß die Anwendung ohne Paßworteingabe auf die Tabellen zugreifen kann. Für Client-Server-Systeme verwendet man dazu die Komponente *TDatabase*, siehe Kapitel 16, bei Paradox-Tabellen kann man mit der *TSession*-Methode *AddPassword* der aktuellen *Session* ein Paßwort hinzufügen. Versucht die *Session* dann, auf eine Tabelle zuzugreifen, deren Paßwort schon eimal eingegeben oder mit *AddPassword* bekanntgegeben wurde, dann kann diese Tabelle ohne Paßworteingabe geöffnet werden.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Session->AddPassword("chriss");
  Table1->Open();
}
```

Damit die Paßwortabgfrage unterbleibt, muß erst mit *AddPassword* das Paßwort bekanntgegeben werden. Erst dann darf die Tabelle geöffnet werden. Die Eigenschaft *Active* im Objektinspektor muß somit auf *false* gesetzt werden. Mit *RemovePassword* kann ein Paßwort wieder aus der Liste entfernt werden, mit *RemoveAllPasswords* werden alle Einträge in der Paßwortliste gelöscht.

Alias- und Tabellennamen

Die Komponente *TSession* enthält einige Methoden, mit deren Hilfe man Informationen über die Datenbank erhalten kann.

- Mit *GetAliasNames* erhält man eine Liste aller verfügbaren BDE-Aliase.
- Mit *GetTableNames* erhält man eine Liste der Tabellen- und View-Namen der Datenbank, deren Namen man als einen der Parameter angeben muß.
- Mit *GetStoredProcNames* erhält man die Namen der STORED PROCEDUREs, welche in der jeweiligen Datenbank vorhanden sind; es muß sich dabei um eine Client-Server-Datenbank handeln, weil es bei Desktop-Datenbanken keine STORED PROCEDUREs gibt.

In der Online-Hilfe erfahren Sie, wie die Syntax dieser Methoden lautet, und welche Eigenschaften und Methoden bei *TSession* neben den bereits geschilderten noch vorhanden sind.

6.4 Die Komponente TUpdateSQL

Bei normalisierten Datenbanken muß man in der Regel auf JOINs zurückgreifen, um aussagekräftige Daten zurückzuerhalten. Als Beispiel sollen hier die beiden Tabellen *Angestellter* und *Chef* aus Bild 6.2 dienen. Um hier im *DBGrid* links oben aussagekräftige Daten zu erhalten, wurde die folgende SQL-Anweisung formuliert.

```
SELECT a.nummer, a.namen, c.chef, c.nummer
  FROM uo_ang a, uo_chef c
  WHERE a.chef = c.nummer
```

Nun wird allerdings eine solche Abfrage keine Live-Daten zur Verfügung stellen, es ist also nicht möglich, hier Daten einzufügen, zu ändern oder zu löschen.

In der Beispielanwendung *Tourplaner* werden in solchen Fällen eine *TQuery*- und eine *TTable*-Komponente gekoppelt, wobei die *TQuery*-Komponente das Anzeigen der Daten und die *TTable*-Komponente das Einfügen, Ändern und Löschen derselben übernehmen wird. Diese Vorgehensweise ist in Kapitel 12 beschrieben.

Eine andere Möglichkeit ist die Verwendung der Komponente *TUpdateSQL*, die nun vorgestellt werden soll. Zunächst einmal wird eine Instanz der Komponente *TUpdateSQL* in das Projekt eingefügt, danach werden zwei Eigenschaften der dazugehörenden *TQuery*-Komponente geändert: Die Eigenschaft *ChachedUpdates* wird auf *true* und die Eigenschaft *UpdateObject* auf *UpdateSQL1* (oder wie die Instanz auch immer genannt werden mag) gesetzt.

Mit einem Doppelklick auf das Symbol von *TUpdateSQL* startet man ein Formular zur Definition der drei Eigenschaften *InsertSQL*, *ModifySQL* und *DeleteSQL*. Dabei handelt es sich um die SQL-Anweisungen, welche im Falle des Einfügens, Änderns oder Löschens eines Datensatzes ausgeführt werden. Mit dem Button *SQL generieren* kann man sich die dafür erforderlichen SQL-Anweisungen generieren lassen, allerdings wird man in den meisten Fällen nicht umhin können, diese automatisch generierten Anweisungen abzuändern.

Betrachten wir zunächst die Anweisungen, die hier automatisch erstellt werden würden.

```
insert into uo_ang
   (Nummer, Namen, Chef)
values
   (:Nummer, :Namen, :Chef)

update uo_ang
set
   Nummer = :Nummer,
   Namen = :Namen,
   Chef = :Chef
where
   Nummer = :OLD_Nummer and
   Namen = :OLD_Namen and
   Chef = :OLD_Chef

delete from uo_ang
where
   Nummer = :OLD_Nummer and
   Namen = :OLD_Namen and
   Chef = :OLD_Chef
```

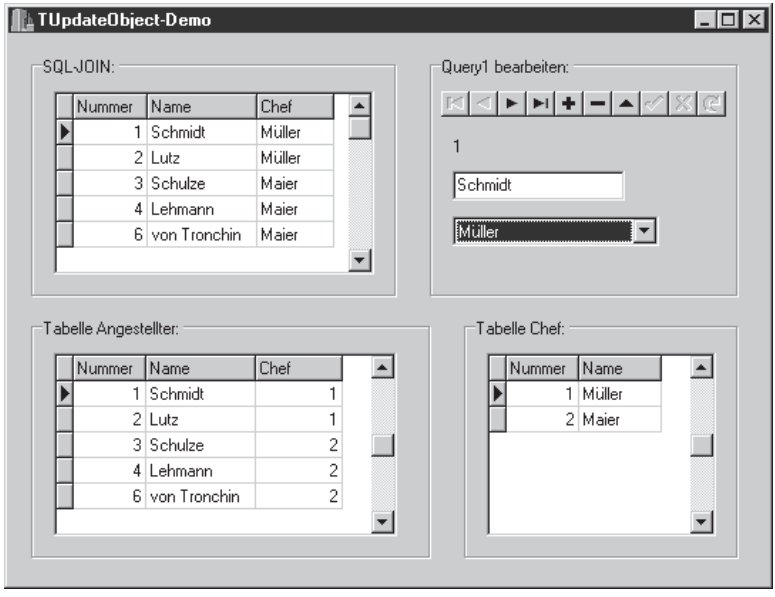


Bild 6.2: Beispiel für die Verwendung von TUpdateObject

Diese Anweisungen enthalten nun mehrere Stellen, die zu einer *Exception* führen würden:

- Für *uo_ang.nummer* wurde ein selbstinkrementierendes Feld gewählt. Dieses wird von der Datenbank selbst verwaltet, und jeder Versuch, dieses zu setzen oder zu verändern, würde zu einer Fehlermeldung führen.
- Das Feld *uo_ang.chef* ist ein *Integer*-Feld, während es sich bei *OLD_chef* um ein *String*-Feld handelt.

Die Anweisungen müßten also folgendesmaßen umformuliert werden:

```
INSERT INTO uo_ang
  (Namen, Chef)
VALUES
  (:Namen, :nummer_1)

UPDATE uo_ang
SET
  Namen = :Namen,
  Chef = :nummer_1
WHERE
  Nummer = :OLD_Nummer
```

```
DELETE FROM uo_ang
WHERE
  Nummer = :OLD_Nummer
```

Die zu ändernden oder zu löschenden Datensätze werden lediglich anhand des Primärschlüssels gesucht. Die Befehle zum Setzen oder Ändern des Feldes *uo_ang.Nummer* werden ersatzlos gestrichen, und dem Feld *uo_angCchef* wird der Abfrage-Parameter *:nummer_1* zugewiesen.

```
void __fastcall TForm1::Query1AfterDelete(TDataSet *DataSet)
{
  Query1->ApplyUpdates();
  Query1->Close();
  Query1->Open();
  Table1->Refresh();
}
```

Das einzige, was für dieses Programm an Quelltext benötigt wird, ist die Funktion *Query1AfterPost*. Da bei der Verwendung von *TUpdateSQL* zwingend die Eigenschaft *CachedUpdates* auf *true* gesetzt werden muß, ist ein Aufruf der Methode *ApplyUpdates* nötig, damit die Änderungen auch übernommen werden. Des weiteren müssen die Abfrage und die *Table2* aktualisiert werden. Diese Funktion muß auch dem Ereignis *OnAfterDelete* zugewiesen werden.

7 Erstellen professioneller Anwendungen

Wir wollen in diesem Kapitel besprechen, was zu beachten ist, wenn man professionelle Anwendungen erstellen möchte. Zunächst gilt es zu klären, was denn unter einem professionellen Programm eigentlich zu verstehen ist. Nach meiner Definition sind dies Programme, über die sich der Anwender nicht ärgert.

Wie ein Blick in das momentane Software-Angebot bestätigt, gibt es kaum eine Möglichkeit, den Anwender zu ärgern, die nicht irgendwo genutzt würde. Sei es, daß die Programme alles andere als stabil laufen, die Benutzeroberfläche den Anwender zur Verzweiflung bringen kann, Dokumentation und Online-Hilfe jeder Beschreibung spotten oder was einem die Software-Industrie sonst noch an Knüppeln zwischen die Beine werfen kann.

In diesem Kapitel sollen einige Fehler aufgezeigt werden, die es zu vermeiden gilt, will man seine Anwender möglichst zufriedenstellen. Im ersten Teil geht es um die Arbeitsorganisation. Diese muß zwar nicht unbedingt Auswirkungen auf das Endprodukt haben, doch die Erfahrung zeigt, daß es die Qualität der Programme nicht gerade fördert, wenn sich selbst der Programmierer nicht mehr im Quelltext zurechtfindet. Danach sollen einige Aspekte der Gestaltung der Benutzeroberfläche besprochen werden. Schließlich soll noch darauf eingegangen werden, wie die Software zu installieren ist und wie man eine zweckmäßige Dokumentation beziehungsweise Online-Hilfe verfaßt.

7.1 Arbeitsorganisation

Wer zum ersten Mal ein größeres Projekt erstellt, neigt oft dazu, die Wichtigkeit der Arbeitsorganisation völlig zu übersehen. Damit macht man sich nicht nur selbst die Arbeit unnötig schwer, man beeinflußt meist auch die Produktqualität. So erzählte mit vor einige Jahren der Hersteller eines rechnergestützten Meßsystems, warum es für die DOS-Version keine Updates mehr geben werde – der Programmierer habe nämlich vollständig die Übersicht über sein Programm verloren.

7.1.1 Strukturierung des Quelltextes

Ein C-Compiler ist ein recht anspruchsloser Gesell – der Quelltext, den Sie ihm vorsetzen, muß von der Syntax her richtig sein, Übersichtlichkeit ist nicht gefragt. Das folgende Listing ist noch nicht einmal auf Unübersichtlichkeit hin optimiert (mehrere Anweisungen in einer Zeile, Zeilenwechsel innerhalb der Anweisung, usw.), es wurden lediglich die einzelnen Anweisungen »geradeaus« eingegeben. Trotzdem ist der Quelltext – und vor allem dessen Struktur –alles andere als leicht zu erfassen.

```
void __fastcall TChild::Table1AfterInsert(TDataSet *DataSet)
{
if(FSchreiben == false){
ShowMessage
("Sie haben keine Schreibberechtigung für dieses Projekt");
Table1->Cancel();}else{
Table1->FieldByName("Show")->AsInteger
= Query1->FieldByName("Nummer")->AsInteger;
if(Query2->RecordCount > 0){
Query2->Last();
Table1->FieldByName("Tag")->AsString
= Query2-> FieldByName("Tag")->AsString;}
DBGrid1->Font->Color = clSilver;}
}
```

Beim folgenden Quelltext, der inhaltlich exakt dieselben Anweisungen aufweist, wurden lediglich drei simple Formatierungsregeln beachtet, und schon sieht das Ergebnis deutlich besser aus.

```
void __fastcall TChild::Table1AfterInsert(TDataSet *DataSet)
{
  if(FSchreiben == false)
  {
    ShowMessage
      ("Sie haben keine Schreibberechtigung für dieses Projekt");
    Table1->Cancel();
  }
  else
  {
    Table1->FieldByName("Show")->AsInteger
      = Query1->FieldByName("Nummer")->AsInteger;
    if(Query2->RecordCount > 0)
```

```
    {
      Query2->Last();
      Table1->FieldByName("Tag")->AsString
        = Query2-> FieldByName("Tag")->AsString;
    }
    DBGrid1->Font->Color = clSilver;
  } // else(FSchreiben == false)
} // TChild::Table1AfterInsert
```

- Nach jedem *{* wird der Quelltext um zwei Leerzeichen eingerückt und dementsprechend bei jedem *}* wieder um zwei Zeichen ausgerückt. Auf diese Weise wird schnell deutlich, in welchen Schleifen oder in welchen Verzweigungen die einzelne Anweisung steht. Um ganze Blöcke ein- bzw. auszurücken, markieren Sie den Block und betätigen Strg + Umsch + I bzw. Strg + Umsch + U.

- Hinter jedem *}* steht ein Kommentar, der etwas darüber aussagt, zu welchem *{* dieses *}* gehört. Im Quelltext-Fenster des C++Builders können Sie – je nach dessen Größe – manchmal nur rund zwölf Zeilen Quelltext gleichzeitig betrachten, einzelne Prozeduren können aber ohne weiteres einige hundert Zeilen lang werden. Auch besteht die Möglichkeit, daß – wie hier – bei einem Ausdruck eine Prozedur von einem Seitenwechsel unterbrochen wird.

  In diesem Fall können Sie auch anhand des Einrückens nicht mehr erkennen, wo nun ein Block endet. Wenn Sie nun versuchen, Quelltext zu kopieren oder einzufügen, eine Schleife um eine *else*-Klausel zu ergänzen oder Sicherheitsabfragen nachträglich einzufügen, dann steigt die Wahrscheinlichkeit eines Fehlers um das Quadrat der Unübersichtlichkeit.

  Bei *if-else*-Verzweigungen sollte hinter beiden *}*-Anweisungen ein entsprechender Kommentar stehen.

```
if(i ==)
{
  ...
} // if i = 3
else
{
  ...
} // else i = 3
```

7.1.2 Kommentare

Zwischen /* */-Zeichen können Sie Kommentare einfügen, also Zeichen, die vom Compiler ignoriert werden. Auch zwei Schrägstrichen (//) folgt ein Kommentar, der mit dem Ende der Zeile endet.

Kommentare dienen dazu, den Quelltext lesbarer zu machen, man sollte Sie deshalb sparsam einsetzen. In folgenden Situationen könnte dies sinnvoll sein:

- Bei umfangreicheren Prozeduren, Schleifen und Verzweigungen sollten Sie zu jedem *}* als Kommentar die Anweisung anhängen, zu welcher Anweisung das jeweilige *{* gehört.
- Während bei Menüpunkten die Beschriftung schon aus dem Namen des Menüpunktes ersichtlich ist, kann man bei Buttons nur ahnen, was sie denn auslösen sollen und wie sie beschriftet sind. Hier könnte man einen entsprechenden Kommentar einfügen.

```
void __fastcall TChild::BitBtn1Click(TObject *Sender)
// Adresse ohne Bemerkung einfügen
```

- Lange Prozeduren sollten in einzelne Abschnitte gegliedert werden. Hier sollte ein Kommentar kurz erläutern, was der kommende Abschnitt tut.

In den meisten Fällen werden Kommentare nur dazu dienen, daß Sie sich selbst im Quelltext zurechtfinden. Hier reichen in der Regel sehr kurze Kommentare. Wenn Sie dagegen den Quelltext veröffentlichen, und dabei nicht – wie hier im Buch – einen Textteil zum Kommentieren haben, sollten Sie ruhig ein wenig großzügiger verfahren.

7.1.3 Bezeichnernamen

Für Personen, die sich schwer entscheiden können, bietet C++Builder eine großartige Hilfe: Die meisten Bezeichnernamen werden automatisch vergeben. So werden die Quelltextdateien mit *unit1, unit2, unit3,* usw. durchnumeriert, alle Komponenten erhalten beim Einfügen einen Namen, die *OnClick*-Ereignisbehandlungsroutine von *BitBtn1* lautet *procedure TForm1::BitBtn1Click*.

Die von C++Builder vergebenen Bezeichner haben den Vorteil, daß sie einmalig sind (darauf legt der Compiler großen Wert), und daß man nicht groß nachdenken muß. Auf der anderen Seite wird man sich in den seltensten Fällen merken können, was *Button3* von *BitBtn7* unterscheidet. C++Builder erlaubt bei allen Komponenten, die Eigenschaft *Name* zu ändern und damit die Komponente anders zu benennen.

Ob man diese Möglichkeit nutzt, hängt vor allem vom eigenen Geschmack ab. Ich persönlich arbeite insgesamt am effektivsten, wenn ich die Benennungen so belasse, wie sie von C++Builder generiert werden. Bei kleineren Projekten mache ich mir noch nicht einmal die Arbeit, das Projekt und die Units zu benennen – dies zwingt allerdings dazu, jedes Projekt in ein anderes Verzeichnis zu stecken (was ohnehinzu empfehlen ist).

Was Sie selbst benennen müssen, sind die selbst deklarierten Variablen – und davon gibt es gerade bei Datenbank-Anwendungen herzlich wenig. In den meisten Fällen werden Sie eine Laufvariable für eine *for*-Schleife benötigen, dafür verwende ich stets *i*, gegebenenfalls *j* und *k*. Des weiteren werden noch häufig Strings benötigt, beispielsweise um Texte zusammenzufassen, bevor sie ausgegeben werden, dafür verwende ich *s* (*t*, *u*, *v*, *w*). Wie Sie sehen, bevorzuge ich vor allem kurze Variablen – das erhöht die Übersichtlichkeit, und warum sollte man sich unnötig Arbeit machen.

7.1.2 Datenmodule

Häufig wird es vorkommen, daß man von mehreren Formularen auf ein und dieselbe *TDataSet*-Komponente (*TTable, TQuery, TStoredProcedure*) zugreifen möchte. In manchen Fällen könnte man dies umgehen, indem man einfach in jeweils alle entsprechenden Formulare diese *TDataSet*-Komponenten einfügt, doch damit handelt man sich meist nur Ärger ein.

So müssen beispielsweise alle *TQuery*-Komponenten geschlossen und wieder geöffnet werden, um eine aktuelle Sicht der Dinge zu bekommen, wenn eine andere Komponente die betreffende Tabelle geändert hat. Greifen alle Programmteile auf dieselbe *TDataSet*-Komponente zu, dann kann man sich eine entsprechende Aktualisierung sparen.

Um auf eine *TDataSource*-Komponente eines anderen Formulars zuzugreifen, wird einfach die Eigenschaft *DataSource* der sogenannten Datensteuerungskomponenten entsprechend gesetzt.

```
DBEdit1.DataSource := Form2.DataSource1;
```

Ein Datenmodul dagegen ist ein »Formular«, das ausschließlich nichtvisuelle Komponenten aufnimmt, in dem Sie also alle Datenzugriffskomponenten sammeln können. Dies hat den Vorteil, daß sich alle Datenzugriffskomponenten an einem bestimmten Platz befinden und sich somit leicht auffinden lassen.

7.2 Installation

Bei Programmen ohne Datenbankzugriff ist die Installation meist sehr simpel: Das Programm besteht meist nur aus einer *Exe*- und einer *Hlp*-Datei, die in der Regel zusammen auf eine Diskette passen. Diese muß der Anwender dann in ein Verzeichnis kopieren und sich das Programm im Programm-Manager einrichten. Ein eigenes Installationsprogramm ist hier in der Regel nicht erforderlich, könnte aber ohne weiteres selbst geschrieben werden.

Die Installation von Datenbankanwendungen ist um einige Größenordnungen schwieriger: Neben der eigentlichen Anwendung muß auch noch die Datenbank installiert werden – bei Desktop-Datenbanken können das schnell mehr als 20 Dateien sein. Des weiteren will die BDE installiert und dort ein entsprechender Alias eingerichtet werden, der dann auch tatsächlich für das Verzeichnis eingerichtet ist, in welchem sich die Datenbankdateien befinden. Außerdem könnte es sein, daß *SQLLinks* benötigt wird.

Wie Sie sehen, ist die Installation von Datenbankanwendungen alles andere als trivial. Aus diesem Grund ist C++Builder das Programm *InstallShield Express* beigelegt, welches dieses Problem wesentlich vereinfacht.

Der Installationsvorgang wird bei *InstallShield Express* in Dialogfeldern zusammengestellt, die Bedienung des Programms ist recht einfach zu erlernen. Aus diesem Grund soll hier nicht näher darauf eingegangen werden. Lediglich die Weitergabe des Datenbank-Tabellen möchte ich kurz erläutern, da dieser Vorgang ein wenig komplizierter ist.

Zunächst werden die Datenbank-Tabellen in eine Gruppe kopiert, beispielsweise in die Gruppe *Beispieldateien*, dazu wird mit KOMPONENTEN UND DATEIEN FESTLEGEN | GRUPPEN UND DATEIEN das dazugehörende Dialogfenster aufgerufen. Um Dateien in eine Gruppe zu kopieren, werden sie einfach mit *Drag&Drop* aus dem Dateimanager oder dem Explorer herübergezogen. Das Verzeichnis der Beispieldateien lautet *<Installdir>\Samples*, sie befinden sich also im Unterverzeichnis *Samples* des Verzeichnisses, in dem laut Benutzervorgabe beim Installationsvorgang die Anwendung installiert wird.

Als nächstes wird der BDE-Alias definiert. Wählen Sie dazu INSTALLSHIELD-OBJEKTE FÜR C++BUILDER WÄHLEN | ALLGEMEINE OPTIONEN und klicken Sie danach die CheckBox *BDE(Borland Database Engine)* an. Es werden danach in einigen Dialogfenster-Eingaben von Ihnen entgegengenommen. Zunächst können Sie wählen, ob Sie die BDE vollständig oder nur teilweise installieren wollen – in der Regel sollten Sie *Vollständige BDE-Installtion* auswählen.

Danach werden Sie nach den Alias-Namen für die BDE-Aliase gefragt, die Sie installieren möchten. Hier im Beispiel soll der Alias *TOUR* erstellt werden. Im nächsten Fenster werden Sie nach einem Pfad und einem Treiber für diesen Alias

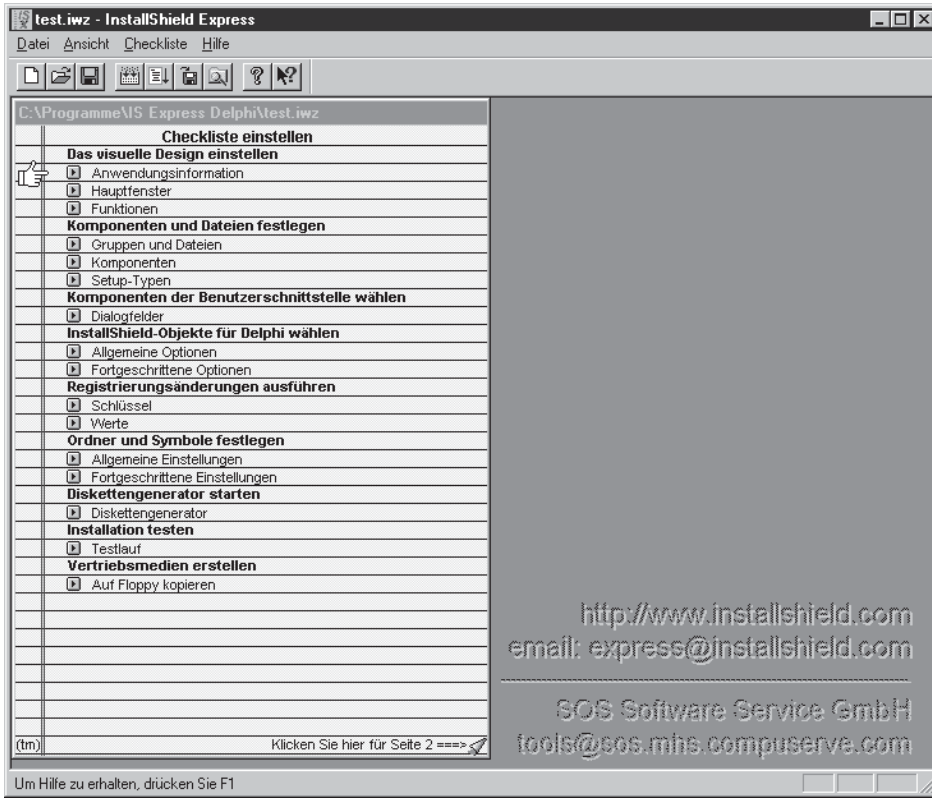


Bild 7.1: Das Programm InstallShield Express

gefragt. Als Pfad geben Sie hier *<Installdir> \ Samples* ein, als Treiber (*Typ*) müßte in diesem Fall der Paradox-Treiber gewählt werden. Damit würde nach der Installation ein entsprechener BDE-Alias existieren und auf das angegebene Verzeichnis zugreifen.

7.2.1 Was sonst bei der Installation zu beachten ist

Auch wenn die Installation soweit problemlos über die Bühne geht, gibt es doch noch einige Punkte zu beachten.

- Wird die Datenbank mit einem Paßwort geschützt, dann sollte wenigstens ein Benutzer angemeldet und dessen Benutzername sowie dessen Paßwort im Installationshandbuch (oder wenigstens in der Datei *readme.txt*) angegeben worden sein.
- Die mitgelieferten Datenbank-Tabellen sollten grundsätzlich keine Einträge enthalten. Der Kunde würde es bestimmt nicht lustig finden, wenn beim er-

sten Mailing einige hundert Sendungen zurückkommen, weil die Adressen von Ihnen frei erfunden worden sind. Eine Ausnahme bilden hier die Tabellen, welche allgemeine Informationen beinhalten, beispielsweise Nachschlagetabellen für die Anrede.

Des weiteren sollte man versuchen, diese leeren Tabellen versuchsweise mit Daten zu füllen. Es wäre denkbar, daß man zwei oder mehrere Tabellen mit Referenzen so miteinander verbunden hat, daß die Eingabe eines Datensatzes nur dann möglich ist, wenn in der jeweils anderen Tabelle schon mindestens ein Datensatz vorhanden ist.

- Was passiert, wenn der Anwender Datensätze aus einer anderen Datenbank einfügen möchte? Es ist nicht unwahrscheinlich, daß der Anwender beispielsweise schon eine Adressen-Datenbank auf dem Rechner hat und in die nun installierte Datenbank kopieren möchte.

7.3 Dokumentation und Online-Hilfe

Bei vielen Programmen werden die Dokumentation und die Online-Hilfe etwas stiefmütterlich behandelt. Dies kann die Brauchbarkeit des Programms stark einschränken, denn letztlich ist es egal, ob in einem Programm ein Feature nicht implementiert ist oder ob es der Anwender mangels brauchbarer Dokumentation nicht findet.

Dokumentation und Online-Hilfe müssen nicht lang sein. Schon oft hab ich mir gewünscht, daß sich die Autoren kürzer gefaßt hätten. Für eine schlecht gemachte Dokumentation gibt es allerdings keine Entschuldigung.

7.3.1 Die Dokumentation

Leider wird bei vielen Programmen gänzlich auf eine schriftliche Dokumentation verzichtet, die einzige verfügbare Information bleibt dann die Online-Hilfe. Gerade bei CD-Samplern ist dies oft nicht anders machbar; was dabei zu beachten ist, lesen sie in Kapitel 7.3.2.

Sollte es jedoch irgendwie machbar sein, dann sollte dem Programm wenigstens eine kurze Einführung mitgegeben werden. Ein paar Seiten im nächsten Copy-Shop vervielfältigt reichen oft aus. Hier wird zunächst die Installation besprochen. Bei der Installation ist die Online-Hilfe noch nicht verfügbar, es erübrigt sich also, dort ein Kapitel mit Installationshinweisen aufzunehmen. Des weiteren sollten die ersten Schritte mit dem Programm in Form eines Tutorials erklärt werden. Gerade Anwender, die mit niedriger Bildschirmauflösung arbeiten, können das Programm und die Online-Hilfe nicht gleichzeitig auf dem Bildschirm darstellen.

Sollte Ihr Programm eine Auftragsarbeit sein, dann werden Sie vermutlich auch die Anwender in der Bedienung des Programms schulen. Auch hier sollten Sie schriftliche Unterlagen bereitstellen, damit sich neue Mitarbeiter in das Programm einarbeiten können.

7.4.2 Die Online-Hilfe

Die Online-Hilfe wird immer dann angezeigt, wenn der Anwender die Funkionstaste *F1* betätigt oder einen entsprechenden Menüpunkt aufruft. Sie können bei den meisten C++Builder-Komponenten die Eigenschaft *HelpContext* setzen, deren Wert mit der dazugehörenden Seite der Online-Hilfe korrespondieren sollte. Betätigt der Anwender dann die Funktionstaste *F1*, dann erhält er Informationen zur gerade aktuellen Sitution.

Die Menüpunkte des Hilfe-Menüs verbinden Sie mit den folgenden Anweisungen:

```
void __fastcall TForm1::Contents1Click(TObject *Sender)
{
  Application->HelpContext(1);
}

void __fastcall TForm1::SearchforHelpOn1Click(TObject *Sender)
{
  AnsiString s = "";
  Application->HelpCommand(HELP_PARTIALKEY, int(&s));
}

void __fastcall TForm1::HowtoUseHelp1Click(TObject *Sender)
{
  Application->HelpCommand(HELP_HELPONHELP, 0);
}
```

Mit der Anweisung *Application->HelpContext* springen Sie auf die angegebene Seite der Online-Hilfe, in diesem Fall auf die Seite 1, welche die Inhaltsseite der Online-Hilfe sein sollte. Mit der Methode *HelpCommand* können Sie verschiedene Anweisungen an die Online-Hilfe geben. Mit der Anweisung *HELP_PARTIALKEY* rufen Sie das Suchen-Fenster auf. Dort wird gleich der Text ausgewählt, auf den der Zeiger zeigt, welchen Sie als Parameter übergeben. In unserem Fall wird ein Zeiger auf einen leeren String übergeben, weshalb im Suchen-Fenster noch kein Eintrag ausgewählt ist. Informationen über die Verwendung der Online-Hilfe rufen Sie mit dem Anweisung *HELP_HELPONHELP* auf.

Erstellen der Online-Hilfe

Für das Erstellen der Online-Hilfe müssen die Texte im RTF-Format vorliegen. Eine Projektdatei muß angelegt und das Ganze dann mit dem Microsoft-Help-Compiler auf DOS-Kommandozeilenebene kompliliert werden. Der ganze Vorgang ist dermaßen aufwendig, daß ich dringend zu einem speziellen Hilfe-Editor raten möchte; ich selbst verwende den *Help Magician*, es gibt jedoch auch andere Programme, die sich ähnlich gut für diesen Zweck eignen.

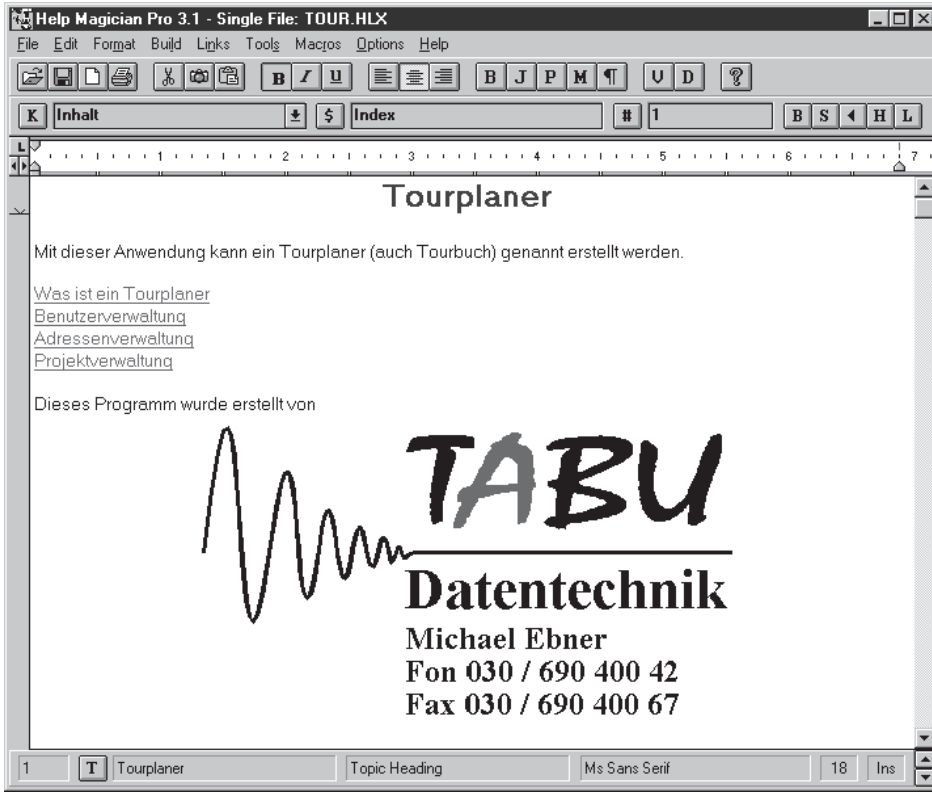


Bild 7.2: Erstellung der Online-Hilfe

Der Text der Online-Hilfe wird seitenweise aufgebaut. Dabei sollte man durchaus so großzügig sein, jedem Thema mindestens eine Seite zu spendieren, auch wenn die Zusammenfassung von mehreren Themen auf einer Seite sich technisch durchaus verwirklichen ließe. Auf der anderen Seite sollte man versuchen, sich kurz zu fassen.Es ist auch nicht unbedingt günstig, auf einer Seite mehr Text unterzubringen als auf eine Bildschirmseite paßt (bei Standard-VGA-Auflösung).

Es gibt zwar durchaus die Möglichkeit, durch wesentlich längere Texte zu scrollen, doch geht dies schnell zu Lasten der Übersichtlichkeit. Gibt es zu einem Thema viel zu schreiben, dann sollte man Jumps und Pop-Ups verwenden. Es ist auch nicht immer sinnvoll, einen Screenshot des Formulars zu zeigen, von dem aus der Anwender das Hilfe-Thema aufruft, denn dieses dürfte ihm in der Regel sowieso vorliegen. Außerdem schlagen sich viele Bitmaps in erhöhtem Speicherbedarf und geringerer Ausführungsgeschwindigkeit nieder.

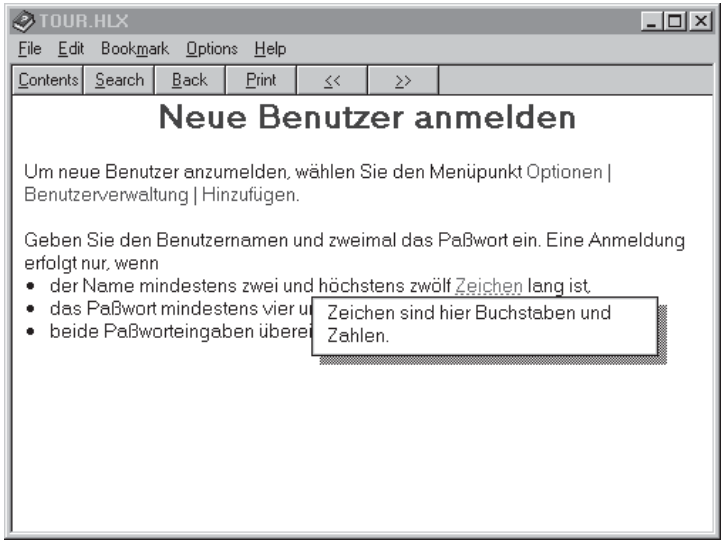


Bild 7.3: Beispiel für ein Pop-Up

Jumps, Pop-Ups und Browse-Folgen

Mit Jumps (zu deutsch Sprünge) lassen sich Querverweise recht einfach erstellen. Die Wörter, für welche diese Sprünge definiert werden, werden mit einer durchgezogenen Linie grün unterstrichen. Klickt der Anwender solch ein Wort an, dann wechselt die Online-Hilfe auf die entsprechende Seite. Mit der Schaltfläche *Zurück* wird die jeweils vorangehende Seite aufgerufen.

Pop-Ups sind kleine Unterfenster, die für kurze Erklärungen oder ähnliches direkt in das Hilfe-Fenster eingeblendet werden; Bild 7.3 zeigt ein solch geöffnetes Pop-Up. Die Wörter, für die Pop-Ups definiert sind, werden mit einer gestrichelten Linie grün unterstrichen.

In Bild 7.3 sehen Sie auch die Button-Leiste unterhalb der Menüzeile. Auf der rechten Seite sind dort zwei Pfeil-Buttons zu finden, mit denen man weiterblättern und zurückblättern kann. Sie haben nämlich die Möglichkeit, so etwas wie

Kapitel, nämlich *Browse*-Folgen einzurichten; darunter versteht man eine Auflistung von mehreren Seiten, die mit Hilfe eben dieser Pfeiltasten durchblättert werden können.

8 Der Tourplaner

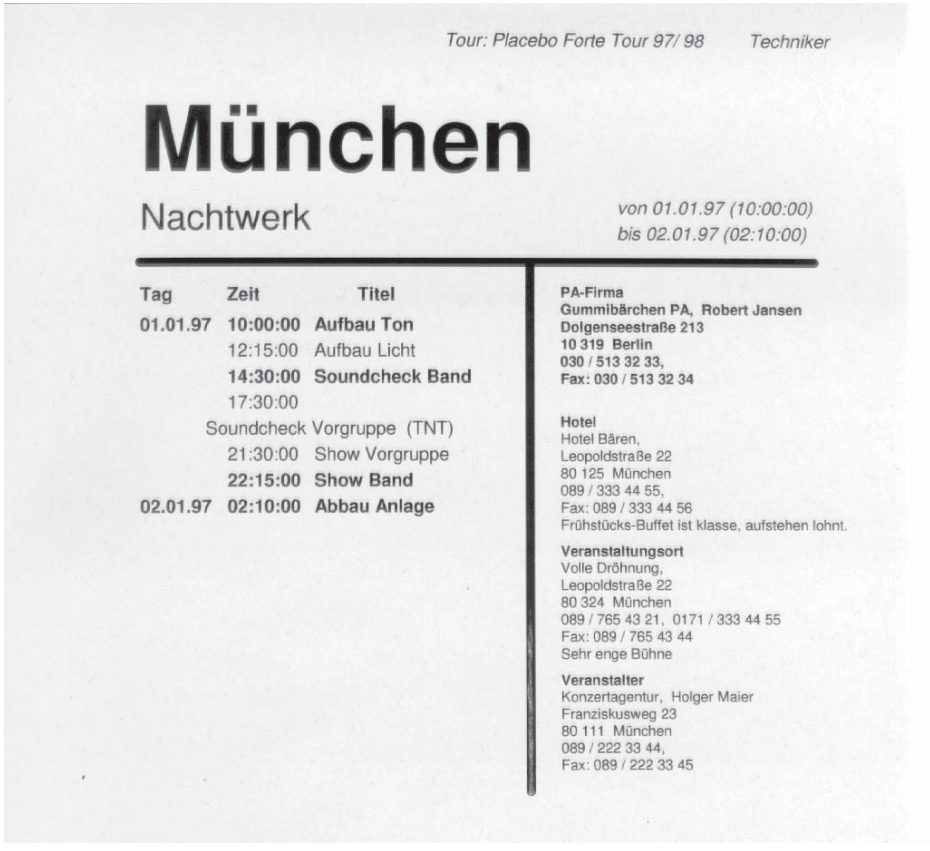
Im zweiten Teil dieses Buches wollen wir eine etwas aufwendigere Beispielanwendung entwickeln. Nachdem einfache Adreßdatenbanken nun schon auf fast jeder Shareware-CD zu finden sind, habe ich mich zur Programmierung eines Programmes entschlossen, mit dem man relativ komfortabel Tourplaner erstellen kann.

Es ist nun einmal das Schicksal von Fachbuchautoren, daß 95% der Leser mit dem jeweils gewählten Beispielprojekt herzlich wenig anfangen können. Mit jedem anderen Beispielprojekt wäre es mir ähnlich gegangen. Jedoch lernen Sie in diesem Beispielprogramm einige grundlegende Dinge kennen, die Sie in gleicher oder ähnlicher Form sicher oft gebrauchen können, wenn Sie anspruchsvollere Datenbank-Anwendungen erstellen wollen:

- Wie erstellt man bei normalisierten Datenbanken aussagekräftige Tabellen (JOINs über zwei, drei oder noch mehr Tabellen) und wie fügt man in diese dann Daten ein?
- Wie erstellt man eine MDI-Applikation für Datenbank-Anwendungen?
- Wie erstellt man eine einfache Benutzerverwaltung?
- Wie programmiert man eine Adreßdatenbank mit komfortablen Funktionen zum Suchen einzelner Datensätze oder zum Herausfiltern eines Teiles des Datenbestandes?

Auch wenn Sie nie im Leben einen Tourplaner erstellen werden, gibt es gute Gründe, sich mit dieser Beispiel-Anwendung zu befassen. Noch eine Warnung: Wenn Sie eine Anwendung programmieren, dann sollten Sie die Handhabung dieser Anwendung so einheitlich wie möglich gestalten. Im folgenden Beispielprogramm wird von dieser Regel hin und wieder etwas abgewichen, damit für ein und dasselbe Grundproblem mehrere Lösungswege aufgezeigt werden können.

Des weiteren möchte ich Sie gleich darauf hinweisen, daß diese Applikation ziemlich umfangreich wird: Über dreißig *DataSet*-Komponenten (*TTable* und *TQuery*) greifen auf 13 Tabellen zu. In den Units von siebzehn Formularen stehen einige tausend Zeilen Quelltext, was insbesondere für Datenbankanwendungen ziemlich viel ist.



Tour: Placebo Forte Tour 97/ 98 Techniker

München

Nachtwerk

von 01.01.97 (10:00:00)
bis 02.01.97 (02:10:00)

Tag	Zeit	Titel
01.01.97	10:00:00	**Aufbau Ton**
	12:15:00	Aufbau Licht
	14:30:00	**Soundcheck Band**
	17:30:00	
		Soundcheck Vorgruppe (TNT)
	21:30:00	Show Vorgruppe
	22:15:00	**Show Band**
02.01.97	**02:10:00**	**Abbau Anlage**

PA-Firma
Gummibärchen PA, Robert Jansen
Dolgenseestraße 213
10 319 Berlin
030 / 513 32 33,
Fax: 030 / 513 32 34

Hotel
Hotel Bären,
Leopoldstraße 22
80 125 München
089 / 333 44 55,
Fax: 089 / 333 44 56
Frühstücks-Buffet ist klasse, aufstehen lohnt.

Veranstaltungsort
Volle Dröhnung,
Leopoldstraße 22
80 324 München
089 / 765 43 21, 0171 / 333 44 55
Fax: 089 / 765 43 44
Sehr enge Bühne

Veranstalter
Konzertagentur, Holger Maier
Franziskusweg 23
80 111 München
089 / 222 33 44,
Fax: 089 / 222 33 45

Bild 8.1: Tagesseite eines Tourplaners

8.1 Was ist ein Tourplaner?

Leser, die nicht mit der Veranstaltungstechnik vertraut sind, werden sich unter einem Tourplaner herzlich wenig vorstellen können, deshalb zunächst einige erklärende Worte:

Wenn eine Truppe von Künstlern (Musiker, Schauspieler) eine Tournee (kurz *Tour*) durch mehrere Städte macht, dann kann dies mit einem relativ hohen organisatorischen Aufwand verbunden sein. Die Beteiligten müssen darüber informiert werden, wo sich in den einzelnen Städten die Veranstaltungsstätten befinden, in welchem Hotel sie untergebracht sind, wie der Terminplan aussieht, usw.

Tourneen mit 50 (oder auch weit mehr) Spielorten sind keine Seltenheit. Würde man diese Informationen den Beteiligten unorganisiert in die Hand geben, dann wäre ein erhebliches Chaos schon vorprogrammiert. Deshalb werden die rele-

vanten Daten in einem Tourplaner (oder Tourbuch) zusammengefaßt. Dort erhält jeder Tag oder jeder Ort eine extra Seite, auf der die benötigten Adressen sowie der Ablaufplan stehen. Die einzelnen Seiten werden dann irgendwie gebunden, vorneweg fügt man die Adressen aller Beteiligten ein und vielleicht kommen noch einige Worte zur aktuellen Produktion hinzu (damit auch die Techniker erfahren, was den Künstlern vorschwebt).

Für gewöhnlich werden solche Tourplaner mit der Textverarbeitung erstellt. Nun ist es allerdings relativ frustrierend, für jede Produktion wieder Adressen einzugeben, welche zur Hälfte ohnehin im Rechner sind. Was liegt hier näher, als eine Datenbank zu verwenden, aus welcher die entsprechenden Adressen bloß noch ausgelesen werden müssen. Ein solches Programm soll hier erstellt werden.

Bild 8.1 zeigt die Tagesseite eines solchen Tourplaners. In der Kopfzeile wird angegeben, zu welcher Tour diese Seite gehört – der Band-Name *Placebo forte* ist frei erfunden wie auch die angegebenen Adressen. Des weiteren findet man in der Kopfzeile die Kategoriebezeichnung *Techniker* – was es damit auf sich hat, soll später geklärt werden.

Als Titel steht hier die Stadt, in der gespielt wird, als Untertitel der Veranstaltungsort. Titel und Untertitel sind jeweils frei wählbar, sie werden nicht aus anderen Angaben übernommen. Rechts daneben sind Beginn und Ende des jeweiligen Arbeitstages durch Datum und Uhrzeit spezifiziert. Diese Angaben werden automatisch aus der Terminliste übernommen. Das Programm verwendet nicht zwangsläufig für jeden Tag eine eigene Seite. Es ist durchaus möglich, auf einer Seite mehrere Tage zusammenzufassen – beispielsweise dann, wenn mehrere Veranstaltungen am selben Ort stattfinden – oder auch einen Tag auf mehrere Seiten zu verteilen – was dann sinnvoll sein kann, wenn sehr viele Termine am selben Tag zu erledigen sind.

Auf der linken Seite werden dann die einzelnen Termine mit Datum und Uhrzeit aufgelistet. Hierbei fällt auf, daß einige der Termine fett gedruckt sind, andere jedoch nicht. Dies rührt daher, daß nicht für alle Beteiligten alle Termine relevant sind. So sind beispielsweise für die Techniker Auf- und Abbau der Tonanlage relevant, die Musiker interessieren sich dagegen nicht für diese Informationen. Hier bietet das Programm die Möglichkeit, drei Kategorien von Beteiligten zu definieren (beispielsweise Musiker, Techniker, Vorgruppe); bei allen Terminen kann dann angegeben werden, für welche Kategorie der jeweilige Termin hervorgehoben werden soll und für welche nicht.

Wie Kenner der Branche wissen, sind bei jeder Seite weitgehend dieselben Termine aufgelistet. Das Datum ändert sich dabei immer, die Uhrzeiten ändern sich öfter, die Bezeichnungen der Termine so gut wie nie. Hier soll das Programm dann zwei Optionen bieten, die Alternativen zur manuellen Eingabe aller Daten darstellen:

- Zum einen soll eine Tabelle aller Termintexte angelegt werden, welche dann jeweils über eine *DBComboBox* ausgewählt werden können. Da bei den verschiedenen Projekten manchmal sehr unterschiedliche Texte verwendet werden, mit steigender Zahl der Einträge in der *DBComboBox* aber die Übersichtlichkeit sinkt, sollen die Texte dem jeweiligen Projekt zugeordnet werden. Es ist daher die Möglichkeit vorzusehen, diese Texte aus einem anderen Projekt heraus zu kopieren.
- Zum anderen soll es möglich sein, die Termine von einer anderen Seite zu übernehmen. Angaben zu Tag, Uhrzeit und Titel sind in diesem Zusammenhang relevant.

  Es ändert sich allerdings – wie bereits erwähnt – bei den verschiedenen Seiten immer das Datum. Im Gegensatz zum Ausdruck in Bild 8.1 wird in der Tabelle zu jedem Termin das Datum gespeichert und auch im *DBGrid* so angegeben. Es wäre jedoch ärgerlich, wenn nach dem Kopieren dann bei allen Terminen das Datum manuell abgeändert werden muß.

  Hier besteht dann mit dem Menüpunkt *Übernehmen* die Möglichkeit, beim Einfügen gleich automatisch das Datum entsprechend abzuändern. Sollen beispielsweise die Termine der Seite in Bild 8.1 auf die nächste Seite übernommen werden, dann wird vor dem Einfügen ein Datum abgefragt. Wird hier beispielsweise *2.1.97* eingegeben, dann werden alle Termine vom *1.1.97* auf den *2.1.97* und die Termine vom *2.1.97* auf den *3.1.97* abgeändert. Will man dann die Termine auch noch für alle weiteren Tourneetage verwenden, dann muß man nur jeweils den Menüpunkt *Übernehmen* aufrufen und das passende Datum eingeben. Mit dieser Funktion ist das Programm jeder Textverarbeitung überlegen, bei welcher die Daten mit *Suchen & Ersetzen* geändert werden müßten.

Auf der rechten Seite stehen dann die für den Tag relevanten Adressen. Auch hier besteht wieder die Möglichkeit, gewisse Adressen durch fett-Formatierung hervorzuheben. Dies ist beispielsweise dann sinnvoll, wenn die Beteiligten in verschiedenen Hotels untergebracht sind oder wenn Musiker vor der Show noch ein Interview geben müssen. Die Überschriften über den Adressen sind jeweils frei wählbar, schließlich könnten bei anderen Veranstaltungen wieder ganz andere Adressen relevant sein. Die Überschriften können jedoch gespeichert und dann über eine *DBComboBox* abgerufen werden.

8.1.1 Weitere Funktionen

Mit diesem Programm sollen sich auch noch zwei andere Seiten ausdrucken lassen.

- Bei größeren Projekten ist es ganz sinnvoll, eine Übersicht über alle Tage zu erstellen. Auf diesen sollen dann jeweils der Start- und Ende-Termin sowie

der Projekttitel und der Untertitel aufgelistet werden. Diese Daten kann das Programm aus den Tagesdaten ermitteln, so daß dafür keine weiteren Eingaben erforderlich sind.

- Des weiteren soll eine Liste der Teilnehmer erstellt werden können. Dies könnte man prinzipiell genausogut mit der Textverarbeitung machen. Wenn in der Adreßdatenbank jedoch auch die Adressen der Musiker und Techniker vorhanden sind, dann ist es sinnvoller, die Teilnehmerliste mit diesem Programm zu erstellen. Als zusätzliche Information soll bei jedem Beteiligten die Funktion angegeben werden, außerdem sollen die Beteiligten in Gruppen zusammengefaßt werden, nach denen die Adressen dann beim Ausdruck gruppiert werden.

8.1.2 Weitere Anforderungen an das Programm

Bevor man das Datenmodell erstellt, sollte man sich über den Funktionsumfang der Anwendung noch gründlich Gedanken machen. Ansonsten könnte sich während des Programmierens herausstellt, daß man die Tabellen besser ganz anders definiert hätte. Müßte man doch dann nicht nur die Tabellen umdefinieren, vielfach würde auch schon ein erheblicher Teil der bis dahin programmierten Applikation hinfällig.

Benutzerverwaltung

- Das Programm soll nur von Personen gestartet werden können, welche als Benutzer angemeldet sind und sich mit ihrem Paßwort legitimieren.
- Auf die einzelnen Projekte sollen die Benutzer nur dann Zugriff haben, wenn
  - sie selber das Projekt erstellt haben,
  - das Projekt für öffentlich erklärt wurde oder wenn
  - dem Benutzer der Zugriff explizit gewährt wurde.

  Im letzten Fall kann dann festgelegt werden, ob der Benutzer das Projekt nur lesen darf oder ob ihm auch Datenänderungen (Einfügen, Ändern Löschen) gestattet sind.
- Es kann festgelegt werden, ob ein Logfile erstellt werden soll. Dabei handelt es sich um eine Tabelle, in der festgehalten wird, wann und von wem das Projekt geöffnet wurde und ob dabei Datenänderungen vorgenommen worden sind. Das Logfile kann nur vom Besitzer (dem Benutzer, der das Projekt angelegt hat) sowie von anderen Benutzern, denen der Zugriff auf das Logfile explizit gestattet wurde, eingesehen werden.

Oberfläche

- Das Programm soll als MDI-Applikation erstellt werden, es sollen sich also mehrere Fenster gleichzeitig öffnen und bearbeiten lassen.
- Der Anwender soll so weit wie möglich von Texteingaben entlastet werden. Ziel soll es sein, daß ein und derselbe Text nur einmal eingegeben werden muß und dann immer wieder abgerufen werden kann. Auch Datums- und Uhrzeitangaben sollen mit der Maus wählbar sein.
- Informationen auf dem Bildschirm sollen weitgehend in Klartext dargestellt werden. Bei einer Teilnehmerliste sollen beispielsweise nicht die Adreßnummern der Teilnehmer mit aufgelistet werden, sondern deren Namen, obwohl die zugrunde liegende Tabelle mit Adreßnummern arbeitet. Diese Vorgehensweise erfordert es beispielsweise, eine *TQuery*- und eine *TTable*-Komponente zu einer Kombination zu verknüpfen.

8.2 Vorentscheidungen

Nachdem die Anforderungen an das Programm zumindest in groben Zügen geklärt sind, müssen noch einige Entscheidungen bezüglich der Programmiersprache oder der verwendeten Datenbanken getroffen werden. Im folgenden soll dargelegt werden, welche Gründe für die getroffenen Entscheidungen sprechen.

C++Builder

Für die Programmierung der Anwendung soll der C++Builder verwendet werden. Daß dieses Programm verwendet werden soll, ergibt sich schon aus der Thematik dieses Buches. Darüberhinaus gibt es jedoch viele weitere Gründe, um hier gerade mit dem C++Builder zu arbeiten. So sind viele Funktionen nicht oder nur mit erheblichem Aufwand mit der Makrosprache von Datenbankprogrammen wie dBase oder Paradox zu verwirklichen. Bei allem, was nicht zu den Standard-Datenbankfunktionen gehört, macht sich auch der Unterschied zwischen einer kompilierten EXE-Datei und einer zu interpretierenden Datei deutlich bemerkbar.

Selbstverständlich wäre es auch möglich, mit Delphi zu arbeiten – der Tourplaner wurde ursprünglich auch mit Delphi 1.0 programmiert und erst später auf den C++Builder portiert.

Paradox-Tabellen

Für die Daten sollen Paradox-Tabellen (Paradox 7.0 für Windows) verwendet werden. Zunächst war hierbei zu entscheiden, ob eine Desktop-Datenbank oder

ein Client-Server-System verwendet werden soll. Im Gegensatz zu einer Auftrags- oder einer Mitgliederverwaltung handelt es sich hierbei um eine Applikation, welche vom Anwender vergleichsweise selten eingesetzt werden wird. Die Wahrscheinlichkeit, daß mit mehreren Rechnern zugleich auf denselben Datenbestand zugegriffen werden soll, kann als vernachlässigbar gelten. Deshalb wäre ein Client-Server-System hier ein ungerechtfertigter Mehraufwand.

Es bleibt die Entscheidung zwischen Paradox und dBase. Wie bereits in Kapitel 1 festgestellt wurde, bieten Paradox-Datenbanken eine Reihe von Vorteilen gegenüber dBase-Datenbanken, so daß hier die Entscheidung nicht schwergefallen ist.

Ausdruck mit dem Objekt Printer

Die Ausdrucke der einzelnen Seiten sollen über das Objekt *Printer* vorgenommen werden. Die Alternativen wären hier die *QuickReport*-Komponenten gewesen. Hier wäre es jedoch sehr schwierig gewesen, die Termine und die Adressen nebeneinander darzustellen, so daß der Ausdruck mit *Printer* die einfachste Lösung ist.

MDI

Die Anwendung soll als MDI-Appliaktion erstellt werden, also als Anwendung, bei der mehrere Fenster und somit Projekte gleichzeitig geöffnet und bearbeitet werden können. Auf diese Weise kann der Anwender relevante Daten ohne größeren Aufwand in alten Projekten nachschlagen, auch lassen sich manche Daten zwischen den einzelnen Projekten austauschen.

Zudem erhalte ich auf diese Weise Gelegenheit, Ihnen die notwendigen Grundkenntnisse zu vermitteln, die Sie für die Programmierung einer MDI-Anwendung benötigen. (Bei manchen Problemen werden Sie von den Handbüchern und der Online-Hilfe ziemlich allein gelassen.)

8.3 Das Datenmodell des Grundgerüstes

Man sollte seine Anwendungen stets so programmieren, daß zunächst diejenigen Teile erstellt werden, die sich vielleicht auch bei anderen Projekten verwenden lassen. Sobald das Projekt in diesem Rahmen einwandfrei läuft, werden die betreffenden Dateien in ein anderes Verzeichnis kopiert. Bei späteren Projekten kann man dann auf dieses Grundgerüst zurückgreifen und sich viel Arbeit ersparen. Darüber hinaus trägt diese Vorgehensweise dazu bei, daß sich die einzelnen Programme ähnlich bedienen lassen, was der Anwender in der Regel mit Freude zur Kenntnis nimmt.

Wir wollen deshalb zunächst ein Grundgerüst für eine MDI-Datenanwendung erstellen, in welchem die Benutzerverwaltung und die Anweisungen zur Führung des Logfiles bereits enthalten sind. Analog zu dieser Vorgehensweise soll zunächst auch das Datenmodell des Grundgerüstes erstellt werden.

8.3.1 Anforderungen an das Datenmodell

Bild 8.2 zeigt die Verknüpfung der Objekte *User* und *Projekt* nach dem erweiterten Entity-Relationship-Modell (siehe Kapitel 1.2.3). Zwischen den Objekten *User* und *Projekt* gibt es zwei Verknüpfungen, *Zugriff* und *Logfile*. Bei der Beziehung *Zugriff* besteht für das Entity *User* die obligatorische Mitgliedschaft: Beim Anlegen des Projektes erhält der Besitzer das Zugriffsrecht (und kann sich dieses Zugriffsrecht auch nicht entziehen). Bei der Tabelle *Zugriff* ändert sich dies jedoch: Da der Besitzer eines Projektes in der Tabelle *Projekt* gespeichert wird, kann es tatsächlich Projekte geben, in denen keine (weitere) Zugriffsberechtigung erteilt worden ist.

Die Führung des Logfiles ist optional, somit gibt es auch Projekte, die auf diese Weise mit keinem User verbunden sind, weil eben kein Logfile geführt wird. Umgekehrt betrachtet gibt es selbstverständlich angemeldete Benutzer, welche weder auf ein Projekt zugegriffen noch auf ein Projekt eine Zugriffsberechtigung haben.

8.3.2 Die Tabellen des Grundgerüstes

Aus diesem Entity-Relationship-Modell soll nun das Datenmodell erstellt werden. Wie bei *n:n*-Verbindungen üblich, werden für die Beziehungen zwischen den Entities eigene Tabellen erstellt.

Die Tabelle *User.DB* besteht aus den Spalten *Nummer*, *Namen* und *Paßwort*. Als *Nummer* wird eine selbstinkrementierende Zahl verwendet, welche auch den Primärschlüssel bildet. Für das Feld *Namen* sind zwölf alphanumerische Stellen vorgesehen – da die Benutzernamen nicht aus dem vollen Vor- und Zunamen bestehen müssen, ist dies hier ausreichend.

Bei Paradox-Tabellen kann das Paßwort unverschlüsselt in die Tabelle geschrieben werden, da die Tabelle selbst verschlüsselt wird, wenn sie mit einem Paßwortschutz versehen wird. Auf dBase- und InterBase-Tabellen trifft dies allerdings nicht zu, hier müßte die Verschlüsselung von der Anwendung durchgeführt werden. Zu Übungszwecken soll dies hier auch bei einer Paradox-Tabelle geschehen, das dafür verwendete Verfahren verdreifacht die Zahl der Stellen – für ein zehnstelliges Paßwort sind deshalb 30 alphanumerische Stellen vorzusehen.

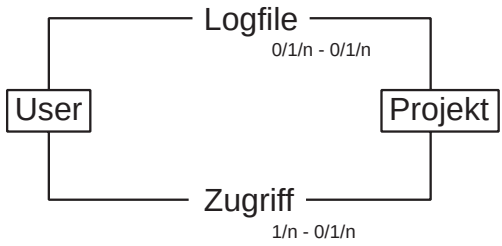


Bild 8.2: Die Objekte des Grundgerüstes nach dem erweiterten ER-Modell

Noch ein Wort zur Bezeichnung dieser Tabelle: Man sollte es vermeiden, für Spalten- oder Tabellennamen SQL-Schlüsselwörter zu verwenden. Es ist jedoch nicht auszuschließen, daß Sie später auf Tabellen zugreifen müssen, bei denen diese Regel nicht beachtet worden ist. Damit ich Ihnen zeigen kann, wie Sie die dabei auftretenden Probleme umgehen, soll hier bewußt dieser Fehler gemacht werden.

Bei umfangreicheren SQL-Anweisungen werden Sie später Tabellen-Synonyme verwenden, welche – der Übersichtlichkeit wegen – nur aus einem Buchstaben bestehen sollten. Wenn Sie bereits beim Entwurf des Datenmodells festlegen, welches Synonym Sie jeweils verwenden wollen, werden Sie später nicht so leicht durcheinander kommen.

Etwas umfangreicher ist die Tabelle *Projekt.DB*. Zunächst einmal finden sich dort die Nummer und der Name des Projektes. Das Feld *Besitzer* beinhaltet die Benutzernummer des Anwenders, der das Projekt angelegt hat. Der Besitzer eines Projektes hat besondere Rechte auf dieses Projekt, beispielsweise darf er weiteren Benutzern den Zugriff gestatten. Deshalb wird diese Zugriffsberechtigung auf das Projekt nicht in der Tabelle *Zugriff.DB* gespeichert, weil dies zum dort einen eine weitere Spalte erfordern würde und weil zum anderen einige Funktionen aufwendiger zu programmieren wären.

Das boolesche Feld *Oeffentlich* (vermeiden Sie Umlaute) legt fest, ob jedem Benutzer der Zugriff auf dieses Projekt gestattet ist, mit dem booleschen Feld *Logfile* legt man fest, ob ein Logfile geführt werden soll oder nicht. Das Feld *Datum* beinhaltet den Datums- und Zeitwert des letzten Zugriffs. Zum Öffnen der Projekte wird eine Tabelle erstellt werden, welche die vorhandenen Projekte auflistet – anhand des Datums wird die Tabelle so sortiert, daß die zuletzt geöffneten Tabellen zuerst in der Liste erscheinen. Dafür wird das Feld *Datum* benötigt.

Die Spalten *A*, *B* und *C* beinhalten Strings, mit denen später die drei Kategorien (zur Hervorhebung von Terminen und Adressen) beschriftet werden. Diese drei Spalten sind für unser Grundgerüst unerheblich. Wenn Sie eigene Projekte erstellen, können Sie diese Spalten bei der Definition der Tabellen ignorieren.

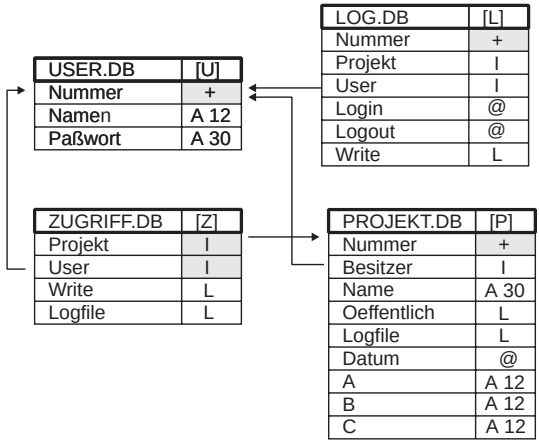


Bild 8.3: Das Datenmodell für das Grundgerüst

Die Tabelle *Zugriff.DB* beinhaltet die Zugriffsrechte weiterer Benutzer für das Projekt. Da ein Benutzer für ein und dasselbe Projekt nur eine Zugriffsberechtigung benötigt, reicht als Primärschlüssel die Kombination von Projekt- und Benutzernummer aus. Sobald eine solche Kombination in der Tabelle existiert, hat der angegebene Benutzer für das jeweilige Projekt das Leserecht. Um ihm zusätzlich das Schreibrecht einzuräumen, wird das Feld *Write* auf *true* gesetzt. Soll ihm die Einsicht in das Logfile gestattet werden, dann wird das Feld *Logfile* auf *true* gesetzt.

Ist für das jeweilige Projekt ein Logfile vorgesehen, dann wird der Zeitpunkt des Öffnens und des Schließens in der Tabelle *Log.DB* vermerkt. Da ein Benutzer mehr als einmal auf ein und dasselbe Projekt zugreifen kann, wäre die Kombination aus Projekt- und Benutzernummer nicht mehr eindeutig und somit als Primärschlüssel nicht zu gebrauchen.

Prinzipiell könnte man nun eine weitere Spalte einführen, in der vermerkt wird, der wievielte Zugriff des jeweiligen Benutzers auf dieses Projekt vorliegt, doch ist es ein wenig aufwendig, solche Werte zu generieren. Außerdem sollte ein Primärschlüssel nur wenige Spalten umfassen, auch wenn – zumindest in unserem Beispiel – keine Referenz auf diese Tabelle gebildet wird. Eine Alternative wäre, eine der beiden Datums- und Zeitkombinationen als Primärschlüssel zu verwenden, dies wird jedoch nicht von allen Datenbanken unterstützt.

In solchen Fällen ist es das einfachste, eine fortlaufende Nummer zu vergeben und ausschließlich diese als Primärschlüssel zu verwenden. In der Spalte *Write* wird festgehalten, ob der Anwender nur das Projekt gelesen hat (*false*) oder ob von ihm auch Daten geändert worden sind, genauer gesagt, ob er mindestens einmal die Methode *Post* aufgerufen hat (*true*).

8.4 Das Datenmodell des Tourplaners

Bild 8.4 zeigt die Objekte des restlichen Programms im erweiterten Entity-Relationship-Modell, wobei der Übersichtlichkeit halber einige Objekte nicht dargestellt worden sind.

Ausgangspunkt sind hier die Projekte. An diesem Projekt sind Personen als Mitwirkende beteiligt, über die Beziehung *Beteiligte* werden deren Adressen mit dem Projekt verknüpft. Dabei handelt es sich um eine 0/1/n - 0/1/n-Beziehung. (Eine Adresse ist keinem, einem oder mehreren Projekten zugeordnet, ein Projekt ist mit keiner, einer oder mehreren Adressen verbunden.)

Des weiteren sind die Projekte mit dem Objekt *Show* verbunden. Dabei handelt es sich im Prinzip um eine Seite des späteren Tourplaners, egal ob die sich – wie meistens – über einen Tag erstreckt oder ob hier andere Zeiträume zugrunde liegen. Zwischen den Objekten *Projekt* und *Show* besteht eine 1 - 0/1/n-Beziehung, das heißt, zu einem Projekt können mehrere Shows angelegt werden, eine Show ist aber stets mit exakt einem Projekt verbunden.

Dies könnte man genaugenommen auch anders handhaben: Bei großen Konzertagenturen besteht die Möglichkeit, daß sie zwei Gruppen betreuen, die zufälligerweise an einem Termin gemeinsam spielen – hier wäre dann eine Show mit mehreren Projekten verknüpft. Allerdings ist das Auftreten dieses Falls sehr selten, so daß sich der hierbei deutlich höhere Aufwand nicht lohnen würde, die Beziehung als 1/n - 0/1/n-Beziehung zu implementieren. Dadurch entsteht allerdings die Möglichkeit einer Integritätsverletzung – den beiden Projekten könnten für diesen Tag unterschiedliche, ja gar widersprechende Terminpläne zugewiesen werden. Gemildert wird diese Fehlerquelle dadurch, daß man in diesem Fall die Termine vom einen zum anderen Projekt kopieren kann.

Mit der Show sind die Objekte *Termin* und *Adresse* verbunden. Beim Objekt *Termin* handelt es sich wieder um 1 - 0/1/n-Beziehungen. Ein und derselbe Terminplan kann niemals mehreren Shows gleichzeitig zugewiesen werden. Selbstverständlich gibt es dann auch hier wieder die Möglickeit, die Daten von einer Show zur anderen zu kopieren.

Mit den Adressen werden die Shows über die 0/1/n - 0/1/n-Beziehung *Orte* verknüpft. Auch hier wird man die Beziehung über eine eigene Tabelle herstellen müssen.

8.4.1 Die Tabellen des Tourplaners

Nun kann man daran gehen, die restlichen Tabellen des Tourplaners zu definieren, so wie sie in Bild 8.4 dargestellt sind.

Betrachten wir zunächst die Tabelle *Shows.DB*. Wie bei Entities so üblich, werden alle Einträge mit einer fortlaufenden Nummer versehen, die Spalte *Projekt* bildet einen Fremdschlüssel auf die Tabelle *Projekt.DB* und stellt auf diese Weise eine 1 - 0/1/n-Beziehung her. Als Titel und Untertitel für die einzelne Seite dienen die Spalten *Titel1* und *Titel2*.

Auf dieselbe Weise sind die Einträge der Tabelle *Termin.DB* mit der Tabelle *Shows.DB* verbunden. Hier finden sich als weitere Spalten *Tag* und *Zeit*, die Bezeichnung des Termins sowie die Felder *A*, *B* und *C*, die Informationen zur Hervorhebung einzelner Termine aufnehmen. Um auf die Datums- und Zeitangaben leichter einzeln zugreifen zu können, wurde hier keine gemeinsame Datums- und Zeitspalte gewählt.

Die gepunktete Linie zur Tabelle *Tertex.DB* stellt dar, daß es sich hierbei nicht um eine Referenz handelt, sondern daß für die Spalte *Titel* eine Nachschlagetabelle angelegt wird. Da jedoch auch Werte erlaubt sein sollen, welche nicht in der Tabelle *Tertex.DB* stehen, darf hier keine Referenz gebildet werden. (Eine Referenz würde man außerdem nicht über den Text, sondern über die Nummer bilden.)

Neben der Liste der Termine gibt es auf jeder Seite des Tourplaners auch noch eine Adreßliste, bei der es sich um die Relationship-Tabelle *Orte.DB* handelt. Eigentlich würde bei einer solchen Tabelle der Primärschlüssel aus den beiden Spalten gebildet werden, welche die anderen beiden Tabellen referenzieren, also in diesem Fall die Spalten *Show* und *Adresse*. Wir werden jedoch später bei dieser Tabelle eine *TTable*- und eine *TQuery*-Tabelle koppeln müssen, und da erleichtert es ein wenig die Arbeit, wenn der Primärschlüssel nur mit einer Spalte gebildet wird.

Als weiteres Feld gibt es hier das Memo-Feld *Bemerkung*. Dieses kann wahlweise aus der Tabelle *Adresse.DB* übernommen werden oder mit einem anderen Text gefüllt werden, weshalb hier der Verweis auf die Tabelle *Adresse.DB* nicht ausreicht. Jede der Adressen erhält zudem noch eine Überschrift (damit man schneller erkennt, ob es sich um das Hotel oder das Büro des Veranstalters handelt), auch dieser Text kann wiederum einer Nachschlagetabelle entnommen werden. Mit den Spalten *A*, *B* und *C* wird dann wieder festgelegt, welche der Adressen für welche Kategorie von Beteiligten hervorgehoben werden soll.

Der obere Teil der Tabelle *Adresse.DB* bietet weiter nichts Interessantes. Zum Schluß sind dann sieben boolesche Felder angehängt, mit denen die Branche der Adresse bestimmt werden kann. Im Laufe der Zeit werden sich in dieser Tabelle sehr viele Adressen ansammeln. Wenn so eine Adresse wiedergefunden werden soll, dann muß ein wenig gefiltert werden. Da es nicht auszuschließen ist, daß ein und dieselbe Person in mehreren Branchen tätig ist, müssen hier entsprechend viele boolesche Felder verwendet werden. Damit diese Felder im Objektinspektor alle am Ende der DropDown-Liste aufgeführt werden, wurde allen ein *Z_* vorangestellt.

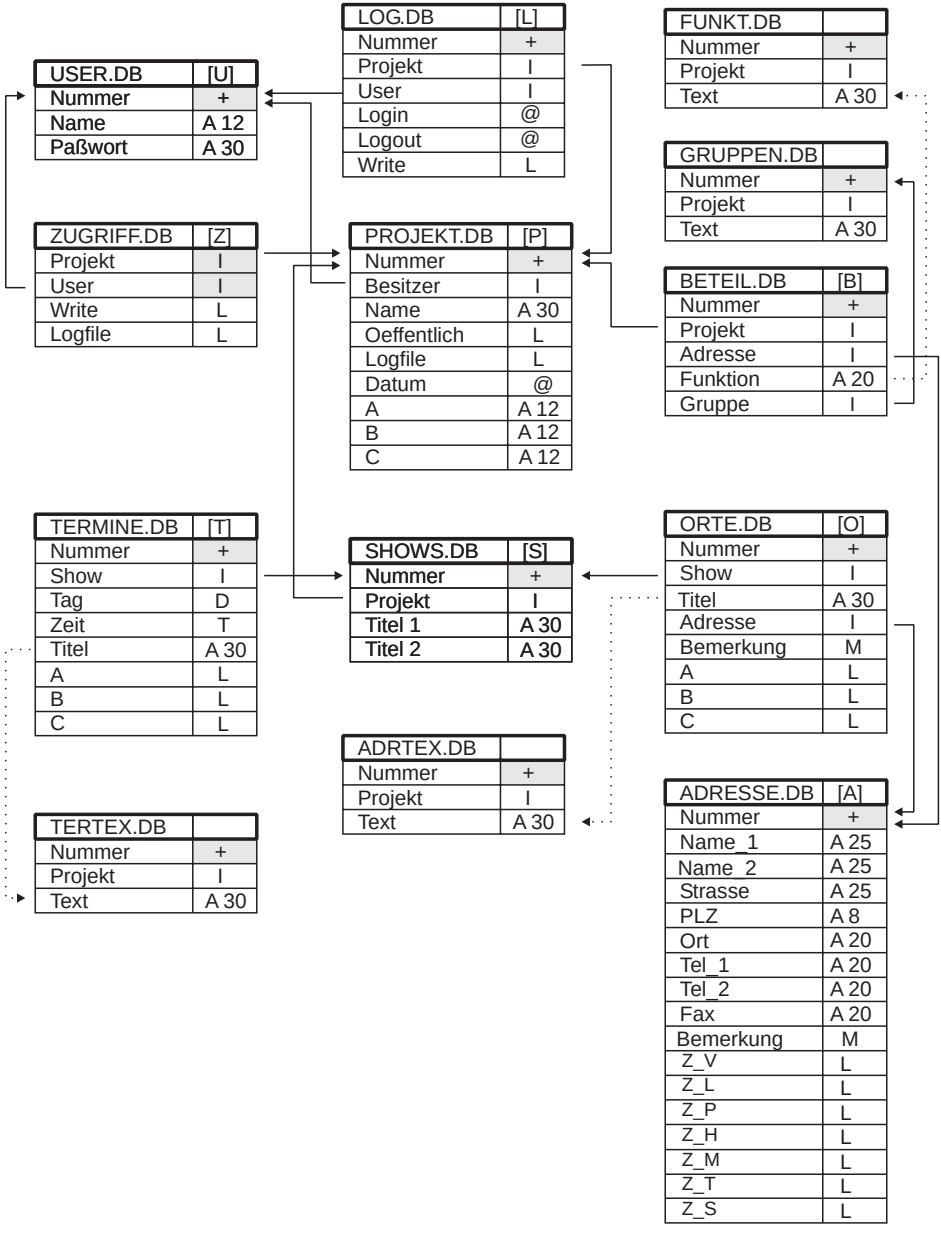


Bild 8.4: Die Tabellen der kompletten Anwendung

Schließlich müssen auch noch die Adressen der Beteiligten mit dem Projekt verbunden werden, dazu dient die Tabelle *Beteil.DB*. Auch hier hat es durchaus seinen Grund, daß der Primärschlüssel wiederum aus einer fortlaufenden Nummer und nicht als Kombination der beiden Fremdschlüsselspalten gebildet wird: Es kommt gar nicht so selten vor, daß ein und dieselbe Person bei einer Tour mehrere Funktionen wahrnimmt, beispielsweise könnte der Schlagzeuger der Vorgruppe gleichzeitig der Monitormischer der Hauptgruppe sein. Weil alle Beteiligten in Gruppen eingeteilt werden, kann es möglich sein, daß dieselbe Projekt-Adressen-Kombination zweimal auftritt, und sich somit die Kombination nicht als Primärschlüssel eignet.

Eine Liste der Beteiligten ist nur bedingt aussagekräftig, wenn damit nicht die Information verbunden wird, welche Funktionen diese Personen ausführen. Zur Aufnahme dieser Information dient die Spalte *Funktion*, der ebenfalls eine Nachschlagetabelle beigegeben ist, um die Tipparbeit für den Anwender zu minimieren. Des weiteren ist jeder Beteiligte Mitglied einer Gruppe (beispielsweise Musiker, Techniker, Vorgruppe). Da hier eine Abweichung von den vorgegebenen Werten nicht erlaubt werden soll, wird eine Referenz auf die Tabelle *Gruppen.DB* gebildet.

9 Erstellung eines Grundgerüstes

Man kann sich auf Dauer viel Arbeit ersparen, wenn man seine Programmteile so aufbaut, daß sie sich an anderer Stelle wiederverwenden lassen. Deshalb soll hier zunächst eine Art »Grundgerüst« erstellt werden, mit dem die Benutzeranmeldung und die Paßwortüberprüfung vorgenommen wird.

Im Prinzip benötigt man dafür eine Datenbank, in welcher die Namen der Benutzer und die entsprechenden Paßwörter stehen. Beim Starten des Programms werden Benutzernamen und Paßwort abgefragt. Wenn die Angaben stimmen, kommt man zum Hauptformular. Des weiteren benötigt man noch die Möglichkeit, neue Benutzer anzumelden, vorhandene Benutzer zu löschen und die Paßwörter zu ändern.

Diese Funktionalität kann man entweder nach dem hierarchischen oder nach dem egalitären Ansatz implementieren:

- Beim hierarchischen Ansatz haben die Anwender unterschiedliche Rechte. Beispielsweise hat der *SysOp* sämtliche Rechte, *Hauptbenutzer* dürfen zwar neue Anwender anmelden, aber nur solche wieder löschen, die sie selbst angemeldet haben, einfache *Benutzer* dürfen zwar mit den Daten arbeiten, aber keine anderen Anwender anmelden oder löschen, und *Gäste* dürfen die Daten nur lesen, nicht aber ändern.

  Ein solcher Ansatz ist mit nicht unerheblichem Aufwand verbunden, gerade bei Programmen, die mit eher unsensiblen Daten arbeiten, bedeutet das, mit Kanonen auf Spatzen zu schießen.

- Beim egalitären Ansatz haben alle Benutzer die gleichen Rechte. Sobald ein Benutzer angemeldet ist, kann er selbst beliebig viele Benutzer anmelden. Löschen kann man einen Benutzer nur mit Kenntnis des entsprechenden Paßwortes. Wenn man also davon ausgeht, daß das Paßwort nur dem Benutzer selbst bekannt ist, dann kann er auch nur sich selbst als Benutzer löschen.

  Da die Sicherheit auf der Anwenderseite hier minimal ist, muß die Sicherheit auf der Datenseite geschaffen werden. Auf ein Projekt hat zunächst nur der Benutzer Zugriff, der dieses Projekt eingerichtet hat. Schreib- und Leserechte können an andere Benutzer weitergegeben werden, darüber hinaus besteht die Möglichkeit, ein Projekt als öffentliches zu deklarieren, so daß alle Benutzer darauf zugreifen können.

Es liegt in der Natur der Sache, daß sich dieser Ansatz für Datenbanken, welche ausschließlich große Mengen von Einzeldatensätzen verwalten (z.B. Adreßdatenbanken), nicht eignet; man müßte sonst schließlich jeder Adresse noch die Information beifügen, wer darauf Lese- oder Schreibzugriff hat. Bei projektorientierten Datenbanken wie in diesem Fall ist solch ein Ansatz durchaus vorteilhaft. Das Grundgerüst für den hierarchischen Ansatz wollen wir beim nächsten Beispielprojekt erstellen.

9.1 Verschlüsselung des Paßwortes

Prinzipiell könnte man die Paßwörter auch unverschlüsselt in die Datenbank schreiben, doch es würde dann genügen, die entsprechende Datei mit einem Editor zu betrachten, um an diese Information zu gelangen. Paradox-Tabellen werden zwar automatisch verschlüsselt, wenn für sie ein Paßwort vergeben wird, bei dBase- und InterBase-Tabellen ist dies aber nicht der Fall.

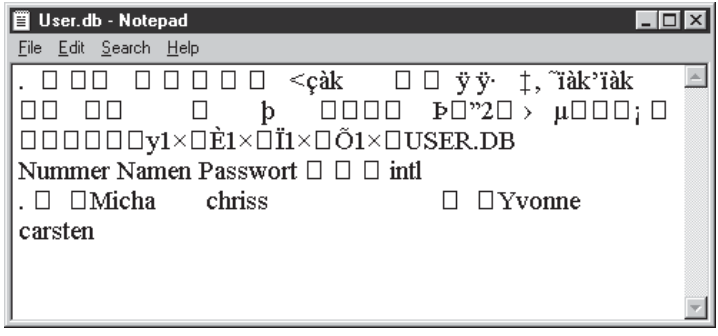


Bild 9.1: Unverschlüsselte Paßwortdatei

Wie Bild 9.1 zeigt, kann man alle Informationen im Klartext lesen: Es gibt zwei Benutzer (*Micha, Yvonne*), deren Paßwörter *Chriss* und *Carsten* lauten. Hier könnte man sich fast die Paßwörter sparen. Der einzige Schutz bestünde darin, daß kaum jemand auf die Idee käme, die Paßwörter seien unverschlüsselt abgespeichert, um daraufhin die Daten mit einem Editor zu betrachten.

Vielleicht würde dieser Schutz in diesem Fall schon ausreichen; da hier aber ein Grundgerüst auch für andere Programme erstellt werden soll, sollten wir hier schon ein paar Hürden mehr einbauen. Für diese Verschlüsselung dient die Funktion *key*, welche aus einem Klartext-String einen verschlüsselten String erstellt.

```
AnsiString TForm2::key(AnsiString eingabe)
{
  AnsiString a, s;
```

```
  int i, k = 1;
  while (eingabe.Length() < 8)
  {
    eingabe = eingabe + char(55 + k);
    k = k + 7;
  }
  for(i=1; i<9; i++)
  {
    k = int(eingabe[i]);
    k = k ^ ((i-1) * 3 + 5);
    s = IntToStr(k);
    a = a + s;
  }
  return(a);
}
```

Das Ergebnis der Verschlüsselung zeigt Bild 9.2. Zunächst muß der Eingabe-String auf eine einheitliche Länge von acht Zeichen gebracht werden, dazu werden zusätzliche Zeichen eingefügt. Prinzipiell könnte man stets dasselbe Zeichen einfügen, dies würde aber einen Ansatz zum Entschlüsseln geben, da dann alle aufgefüllten verschlüsselten Paßwörter auf dieselben Zeichen enden würden. Deshalb wird bei jedem Durchlauf ein anderes Zeichen hinzugefügt.

Als nächstes wird jeder Buchstabe des Passworts in seinen ASCII-Wert gewandelt. Mit dieser Integer-Zahl wird dann eine XOR-Operation (exklusives Oder) durchgeführt. Das Ergebnis wird in einen AnsiString gewandelt und der Variablen *a* hinzugefügt. Am Ende der Funktion wird diese Variable *a* dann als Ergebnis zurückgegeben.

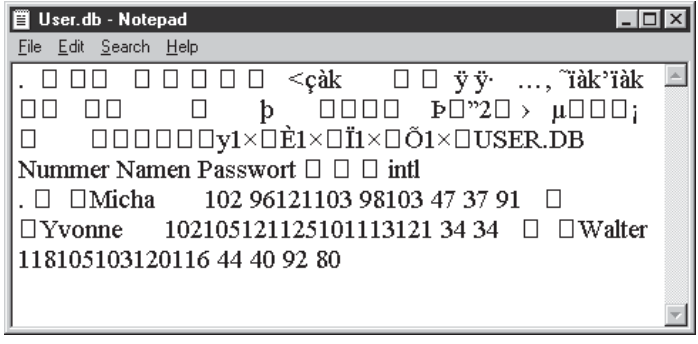


Bild 9.2: Verschlüsselte Paßwortdatei

9.1.1 Verschlüsselungs-Algorithmen

Verschlüsselungs-Algorithmen dienen dazu, das Entschlüsseln der Paßwort-Datei zu erschweren. Unmöglich gemacht werden kann es nicht, insbesondere dann nicht, wenn die Person, welche das Paßwort zu ermitteln sucht, selbst Zugang zur Datenbank hat und so über die Änderung des eigenen Paßworts und den jeweiligen Verschlüsselungsergebnissen darauf schließen kann, nach welchen Regeln die Verschlüsselung erfolgt. Hier gibt es vier Schwierigkeitsstufen.

Stufe 1

Die einfachste Form der Verschlüsselung (welche auch am einfachsten zu knakken ist) besteht darin, jede Zahl nach demselben Muster umzurechnen. Beispielsweise wäre es möglich, zu jeder Zahl 2 hinzuzuaddieren oder 7 zu subtrahieren, auch Multiplikation und Division sind möglich.

Schon etwas kniffliger ist eine XOR-Funktion (exklusives Oder), welche bitweise die einzelnen Buchstaben bearbeitet. Wie dies funktioniert, ist in der folgenden Tabelle dargestellt. Hier soll der Buchstabe *a* (ASCII-Zeichen 97) mit der Zahl 100 durch eine XOR-Funktion verschlüsselt werden. Das Ergebnis wäre hier fünf:

	1	2	4	8	16	32	64	128
'a' (97)	1	0	0	0	0	1	1	0
100	0	0	1	0	0	1	1	0
97 xor 100	1	0	1	0	0	0	0	0

Tabelle 9.1: Verschlüsselung mit der XOR-Funktion

Die XOR-Funktion hat den Vorteil, daß man die Verschlüsselung mit derselben Funktion wieder rückgängig machen kann. (5 ^ 100 ergibt wieder 97.)

Zum Knacken eines Algorithmus der Stufe 1 vergleicht man die ASCII-Zahlen seines eigenen Paßwortes mit den Zahlen des verschlüsselten Paßwortes und ermittelt so, nach welchen Kriterien die Umrechnung erfolgt. Alternativ stellt man eine Tabelle für jedes Zeichen auf.

Stufe 2

Erhöhten Schutz erhält man, wenn man jedes Zeichen anders verschlüsselt, so wie das in der Funktion *key* geschieht:

```
for(i=1; i<9; i++)
{
  k = int(eingabe[i]);
  k = k ^ ((i-1) * 3 + 5);
  s = IntToStr(k);
  a = a + s;
}
```

Hier wird aus dem Platz in der Reihenfolge der Buchstaben eine Zahl berechnet, mit der dann eine XOR-Verknüpfung vorgenommen wird.

Um hier den Code zu knacken, müßte man für jeden Platz in der Reihenfolge der Buchstaben eine Tabelle aufstellen, welche dann die Rückermittlung des zu dieser Zahl gehörenden Buchstabens erlauben würde. Man würde also rund 60 mal ein Paßwort eingeben, welches aus lauter gleichen Zeichen besteht, und dann mit dem Editor nachschauen, welche Zahlen das Programm daraus gemacht hat. Das Knacken des Codes ist hier also vor allem etwas aufwendiger, aber nicht weiter schwer.

Stufe 3

Deutlich besser wäre es deshalb, nicht bei jedem Zeichen, sondern bei jedem Datensatz einen anderen Schlüssel zu verwenden. Man könnte beispielsweise die Zahl, mit der die XOR-Verknüpfung durchgeführt wird, aus der Nummer des Datensatzes ableiten.

```
k := StrToInt(Table1->Fields[0]->AsString);
for(i=1; i<9; i++)
{
  k = int(eingabe[i]);
  k = k ^ (k * 3 + 5);
  ...
}
```

Wenn die Datenbank angewiesen wurde, für das erste Feld (Index Null) fortlaufende Nummern zu vergeben, dann wird diese Nummer auch nur ein einziges Mal vergeben, selbst dann, wenn Datensätze gelöscht wurden. Hier kann man dann nicht mehr mit Tabellen arbeiten, sondern muß wirklich die Formel finden, nach der *k* in den XOR-Operanden umgerechnet wird.

Stufe 4

Das Verfahren nach Stufe 3 kann man noch ausbauen, indem man zusätzlich noch jedes Zeichen anders verschlüsselt. Außerdem könnte man die Verschlüsselung nicht nur von der Nummer des Datensatzes, sondern auch noch von einem

oder mehreren Buchstaben des Namens und / oder des Paßworts abhängig machen.

Wer Programme erstellt, die mit wirklich sensiblen Daten arbeiten, kann auf diese Weise noch ein paar Erschwernisse einbauen. In unserem Grundgerüst soll ein Schutz nach Stufe 2 ausreichen.

9.2 Die Paßwort-Datenbank

Zunächst soll eine Datenbank-Applikation zur Verwaltung der Benutzer und der Paßwörter erstellt werden. Zu diesem Zweck wird mit der *Datenbankoberfläche* zunächst eine Datenbank nach Bild 9.3 erstellt. Nun beginnen Sie ein neues Projekt, fügen diesem ein *TabbedNotebook* hinzu und bestücken die erste Seite folgendermaßen mit Komponenten (normale *TEdit*-Komponenten!):

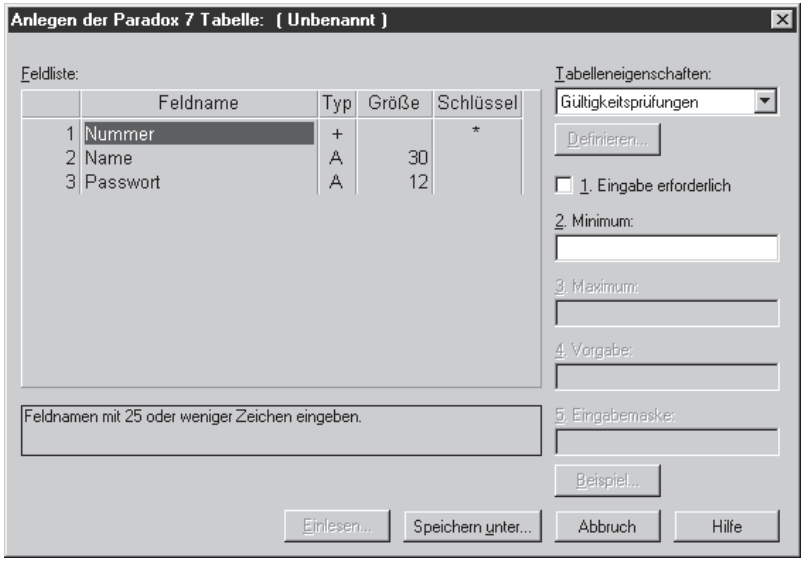


Bild 9.3: Erstellung der Datenbank mit der Datenbankoberfläche

Wie Sie sehen, werden in den Edit-Feldern nicht die Buchstaben des Paßwortes angezeigt, sondern *-Zeichen. Dies können Sie mit der Eigenschaft *PasswordChar* im Objektinspektor einstellen. Da in diesem Fall für den Anwender nicht mehr zu erkennen ist, was er denn nun eigentlich geschrieben hat, muß das Paßwort doppelt eingegeben werden. Nur wenn beide Eingaben übereinstimmen, wird der Benutzer unter diesem Paßwort angemeldet, ansonsten erfolgt eine Fehlermeldung.

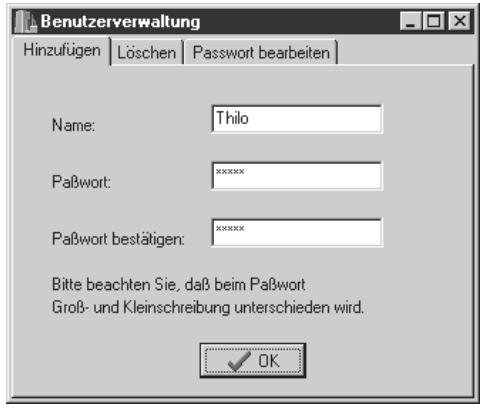


Bild 9.4: Seite zum Hinzufügen neuer Benutzer

Des weiteren müssen die Komponenten *table1* und *datasource1* hinzugefügt werden. Die Komponente *table1* wird mit der gerade erstellten Paßwort-Datenbank verbunden, *datasource1* mit *table1*. (Das Einfügen von *datasource1* wäre zu diesem Zeitpunkt noch nicht erforderlich, da noch keine datensensitiven Dialogelemente verwendet werden, dies erfolgt erst auf den nächsten *Notebook*-Seiten.)

Für *BitBtn3* kann nun folgende *OnClick*-Ereignisbehandlungsroutine erstellt werden:

```
void __fastcall TForm3::BitBtn3Click(TObject *Sender)
// Neuen Benutzer anmelden
{
  if (Edit6->Text == Edit7->Text)
  {
    if (Edit6->Text.Length() < 4)
      ShowMessage
        ("Fehler: Passwort muß mindestens 4 Buchstaben umfassen.");
    else
      if (Edit6->Text.Length() > 8)
        ShowMessage
          ("Fehler: Passwort darf höchstens 8 Buchstaben umfassen.");
      else
        if (Edit5->Text.Length() < 4)
          ShowMessage
            ("Fehler: Name muß mindestens 4 Buchstaben umfassen.");
        else // Name und Passwort weisen die korrekte Länge auf
        {
          Table1->Append();
```

```
            Table1->FieldByName("Namen")->AsString = (Edit5->Text);
            Table1->FieldByName("Passwort")->AsString
              = key(Edit6->Text);
            Table1->Post();
            ShowMessage("Neuer Benutzer wurde angemeldet.");
          } // else (Edit5->Text.Length() < 4)
  } // if (Edit6->Text == Edit7->Text)
  else
  {
    ShowMessage("Fehler: Passwörter stimmen nicht überein.");
    Edit6->Text = "";
    Edit7->Text = "";
  }
} // TForm3::BitBtn3Click
```

Die Prozedur gliedert sich in drei Sicherheitsabfragen und die Anweisungen zum Einrichten des Benutzers in der Datenbank. Zunächst wird sichergestellt, daß:

- der Benutzername aus mindestens zwei Buchstaben besteht
- das Paßwort mindestens vier und höchstens acht Buchstaben umfaßt
- beide Paßworteingaben übereinstimmen

Zum Speichern in der Datenbank muß diese zunächst geöffnet werden; dies geschieht mit dem Befehl *Append*, welcher besagt, daß ein neuer Datensatz hinzugefügt werden soll. Den entsprechenden Feldern werden die Eingaben aus den Edit-Feldern zugewiesen. Dem Feld *Nummer* braucht (und kann) kein Wert zugewiesen werden, da die Datenbank ja fortlaufende Nummern vergibt. Mit dem Befehl *Post* wird das Schreiben des Datensatzes abgeschlossen. Außerdem wird eine Meldung ausgegeben, daß der neue Benutzer angemeldet ist.

Benutzer löschen

Als nächstes soll die Möglichkeit geschaffen werden, die eingerichteten Benutzer auch wieder zu entfernen; die entsprechende *Notebook*-Seite wird gemäß Bild 9.5 mit Komponenten bestückt.

Hier wird nun für die Eingabe des Benutzernamens die Komponente *DBLookupCombo2* verwendet, welche die schnelle Auswahl der angemeldeten Benutzer erlaubt. Dabei bleiben die Felder *DataField* und *DataSource* leer, das Feld *LookupDisplay* erhält den Wert *Namen*, das Feld *LookupSource* den Wert *DataSource1*. Des weiteren muß die Eigenschaft *LookupField* auf *Nummer* gesetzt werden.

Für *BitBtn2* wird nun folgende *OnClick*-Ereignisbehandlungsroutine erstellt:

Bild 9.5: Seite zum Löschen von Benutzern

```
void __fastcall TForm3::BitBtn2Click(TObject *Sender)
// Benutzer löschen
{
  bool b = false;
  Table1->First();
  while (Table1->Eof != true)
  {
    if (DBLookupComboBox2->Text
       == Table1->FieldByName("Namen")->AsString)
    {
      b = true;
      if (key(Edit4->Text)
        == Table1->FieldByName("Passwort")->AsString)
      {
        Table1->Delete();
        ShowMessage("Benutzer wurde gelöscht.");
      }
      else ShowMessage("Fehler: Passwort ist falsch.");
    } //  if DBLookupComboBox1->Text = Table1->FieldByName("Namen...
    Table1->Next();
  } // while (Table1->EOF != true)
  if (b == false) ShowMessage("Namen nicht gefunden");
} // TForm3::BitBtn2Click
```

Die Prozedur sucht zunächst nach einem Datensatz, bei dem der Benutzername mit der Eingabe übereinstimmt. Da das Feld *Namen* weder ein Primär- noch ein Sekundär-Index ist, müssen alle Datensätze einzeln daraufhin überprüft werden

(man könnte auch eine SQL-Anweisung verwenden). Anschließend wird geprüft, ob das eingegebene Paßwort stimmt, und gegebenenfalls der Datensatz gelöscht.

Liegen mehrere Datensätze mit demselben Benutzernamen vor, dann wird davon ausgegangen, daß der zuletzt eingegebene Datensatz bestehen bleiben soll, alle anderen jedoch gelöscht werden sollen; dementsprechend ist es nur möglich, den jeweils ersten Datensatz mit diesem Benutzernamen zu löschen.

(Normalerweise erfolgt eine Anmeldung unter einem bestehenden Benutzernamen nur dann, wenn der Benutzer sein Paßwort vergessen hat. Wenn es ihm dann wieder einfällt, kann der ursprüngliche Eintrag gelöscht werden. Fällt ihm das ursprüngliche Paßwort nicht mehr ein, dann gibt es mit diesem Benutzernamen zwei Einträge, wenn diese Person sehr vergesslich ist, noch weitere. Da es bei einer egalitären Verwaltungsordnung keinen *SysOp* gibt, der diese Einträge wieder löschen kann, bleibt unter Umständen viel Überflüsiges in der Benutzerliste. Andererseits würde es bei einer hierarchischen Verwaltungsordnung in ein mittelschweres Chaos münden, wenn der *Sysop* sein Paßwort vergißt; in unserem Fall braucht er lediglich wieder von einem anderen Benutzer angemeldet zu werden.)

Paßwort ändern

Zuletzt soll auch noch die Möglichkeit vorgesehen werden, das Paßwort zu ändern; die entsprechende *Notebook*-Seite ist folgendermaßen zu bestücken:

Für das Namensfeld wird wieder die Komponente *DBLookupCombo* verwendet, ansonsten kommen gewöhnliche Edit-Felder zum Einsatz. *BitBtn1* löst die hier folgende Prozedur aus:

```
void __fastcall TForm3::BitBtn1Click(TObject *Sender)
// Passwort ändern
{
  bool b = false;
  Table1->First();
  while (Table1->Eof != true)
  {
    if (DBLookupComboBox1->Text
      == Table1->FieldByName("Namen")->AsString)
    {
      b = true;
      if (key(Edit1->Text)
        == Table1->FieldByName("Passwort")->AsString)
      {
        if (Edit2->Text == Edit3->Text)
```

Bild 9.5: Seite zum Löschen von Benutzern

```
        {
          if (Edit2->Text.Length() < 4)
            ShowMessage("Fehler: Passwort muß mindestens 4
              Buchstaben umfassen");
          else
            if (Edit2->Text.Length() > 8)
              ShowMessage("Fehler: Passwort darf höchstens 8
                Buchstaben umfassen");
            else
            {
              Table1->Edit();
              Table1->FieldByName("Passwort")->AsString
                = key(Edit2->Text);
              Table1->Post();
              ShowMessage("Passwort wurde geändert.");
            }
          } // if (Edit2->Text == Edit3->Text)
          else ShowMessage
            ("Fehler: Neue Passwörter stimmen nicht überein.");
        } // if (key(Edit1->Text) == Table1->FieldByName("Passw...
        else ShowMessage("Fehler: Altes Passwort ist falsch.");
      } //  if DBLookupComboBox1->Text = Table1->FieldByName("Nam...
      Table1->Next();
    } // while (Table1->EOF != true)
    if (b == false) ShowMessage("Fehler: Namen nicht gefunden.");
} // TForm3::BitBtn1Click
```

Zunächst wird wieder der entsprechende Datensatz gesucht und daraufhin überprüft, ob das alte Paßwort korrekt eingegeben wurde. Beim neuen Paßwort wird dann daraufhin überprüft, ob es mindestens vier und höchstens acht Buchstaben aufweist und ob die beiden Eingaben identisch sind; ist dies der Fall, wird der Datensatz geöffnet, die Anweisung *Edit* zeigt an, daß der Datensatz dabei geändert wird. Nun wird das neue Paßwort geschrieben und die Operation mit *Post* abgeschlossen.

9.3 Der Paßwort-Dialog

Als nächstes soll das Formular erstellt werden, das beim Starten des Programms nach dem Benutzernamen und dem Paßwort fragt. Zu diesem Zweck werden dem Projekt zwei neue Formulare hinzugefügt, ein Formular für den Paßwort-Dialog und ein Formular, das als Rahmenformular für die spätere Anwendung dient und dann geöffnet wird, wenn das Paßwort korrekt eingegeben wurde.

Das Rahmenformular – in meinem Projekt *Form1* genannt – ist demnach das erste Formular, welches erstellt wird. Dies muß unter OPTIONEN | PROJEKT | FORMULARE | HAUPTFORMULAR entsprechend eingestellt werden. (Prinzipiell könnte man auch das Paßwort-Formular als Hauptformular erstellen, aber das würde später eine ganze Reihe von Problemen mit sich bringen.) Sobald das Programm gestartet wird, muß das Paßwort-Formular aktiviert werden. Dazu dient die folgende *OnShow*-Ereignisbehandlungsroutine von *Form1*.

```
void __fastcall TForm1::FormShow(TObject *Sender)
// Passwortdialog aufrufen
{
  if (Form2->ShowModal() == mrAbort) Form1->Close();
}
```

Wie der Paßwort-Dialog zu bestücken ist, zeigt das Bild 9.7. Für *BitBtn1* wird folgende *OnClick*-Ereignisbehandlungsprozedur erstellt:

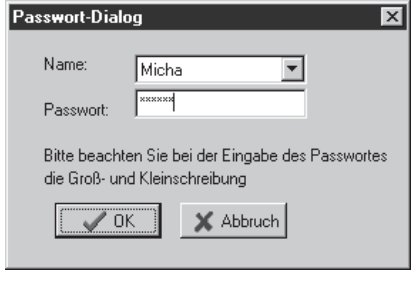


Bild 9.7: Der Paßwort-Dialog

```
void __fastcall TForm2::BitBtn1Click(TObject *Sender)
// OK-Button
{
  bool b = false;
  Table1->First();
  while (Table1->Eof != true)
  {
    if (DBLookupComboBox1->Text == Table1->FieldByName("Namen")-
>AsString)
    {
      if (key(Edit1->Text) == Table1->FieldByName("Passwort")-
>AsString)
      {
        Form2->Close();
        b = true;
      }
      else
      {
        ShowMessage("Passwort ist falsch, Anwendung wird beendet");
        Application->Terminate();
      } // else key(Edit1->Text = Table1->FieldByName("Passwort")-
>AsString)
      break;
    } //  if DBLookupComboBox1->Text = Table1->FieldByName("Namen")-
>AsString
    Table1->Next();
  } // while (Table1->EOF != true)
  if (b == false)
  {
    ShowMessage("Namen nicht gefunden");
    Application->Terminate();
  }
} // TForm2::BitBtn1Click
```

Viel Neues gibt es hier nicht, es wird auch in dieser Prozedur nach dem Benutzernamen gesucht und das Paßwort auf Gleichheit hin überprüft. Ist alles korrekt, dann wird der Paßwort-Dialog geschlossen und die Prozedur mit *break* abgebrochen. Wird ein falscher Benutzername oder ein dazu unpassendes Paßwort eingegeben, dann muß die Anwendung mit *Application.Terminate* geschlossen werden. Dasselbe gilt, wenn *BitBtn2* betätigt wird.

Des weiteren muß in das Rahmenformular (*Form1*) ein Menü eingebunden werden, um die Benutzerverwaltung aufrufen zu können. Ich vermute, daß Sie das

selbstständig auf die Reihe bringen, andernfalls können Sie die Dateien auf der beiliegenden CD-ROM zu Rate ziehen.

Das nun erstellte Programm kann als Ausgangspunkt für andere Programme dienen. Sie sollten allerdings nicht vergessen, den Algorithmus für die Paßwort-Verschlüsselung ein wenig zu modifizieren.

9.3.1 Position des Formulars speichern

Unser Grundgerüst ist nun so weit, daß es sich schon starten und als Grundlage für weitere Projekte verwenden läßt. Wir wollen allerding noch die Projektverwaltung integrieren und einige kleinere Features einfügen.

Wenn Ihr Rechner nicht gerade über einen 21"-Monitor verfügt, dann werden Sie sich beim Programmieren sicher schon oft darüber geärgert haben, daß die erstellten Formulare in der Größe und an der Position auf dem Bildschirm erscheinen, wie Sie diese vorher entworfen haben. Entweder machen Sie Ihre Formulare so groß, daß sie dann den Objektinspektor verdecken, oder sie sind nachher so klein, daß man nicht vernünftig damit arbeiten kann. Gerade beim Programmieren, wenn die Anwendung häufig gestartet wird, ist es äußerst lästig, das Formular immer von Hand auf die passende Größe zu bringen. Aus diesem Grunde möchte ich empfehlen, die folgenden Anweisungen schon sehr früh in das Projekt aufzunehmen – es schont die Nerven ungemein.

Um Arbeitsplatzeinstellungen bis zum nächsten Programmstart zu speichern, wird bei 32-Bit-Programmen die Registry verwendet. Um auf diese zuzugreifen, kann man das Objekt *TRegistry* verwenden. Etwas einfacher zu verwenden – vor allem, wenn ein Programm als 16- und als 32-Bit Programm zugleich entwickelt werden soll (was mit dem C++Builder derzeit nicht möglich ist, aber beispielsweise bei *Delphi* sehr nützlich) – ist das Objekt *TRegIniFile*. Dieses Objekt verwendet man, wie man es von *Ini*-Dateien her gewöhnt ist – es greift aber auf die Registry zu.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  TRegIniFile& Ini = *new TRegIniFile("Software\\TABU\\Tourplaner");
  Left = Ini.ReadInteger("Position","Links",10);
  Top = Ini.ReadInteger("Position","Oben",10);
  Width = Ini.ReadInteger("Position","Breite",620);
  Height = Ini.ReadInteger("Position","Höhe",460);
  delete &Ini;
}
```

Bevor eine *TRegIniFile*-Instanz verwendet wird, muß sie mit *new* erzeugt werden; als Parameter wird der Registry-Pfad übergeben (vergessen Sie nicht, daß

\-Zeichen in C jeweils doppelt auszuführen sind). Die einzelnen Werte für Position und Größe des Formulars werden dann aus der Registry ausgelesen. Als letzter Parameter werden den *Read*-Methoden *default*-Werte übergeben, welche den jeweiligen Variablen zugewiesen werden, wenn ein entsprechender Eintrag in der Registry (noch) nicht existiert oder die gesuchten Daten nicht gelesen werden können. Abschließend wird die *TRegIniFile*-Instanz mit der Methode *free* entfernt. Vergessen Sie dabei nicht, *#include <vcl\Registry.hpp>* in die Header-Datei mit aufzunehmen.

Beim Verlassen des Programms werden dann die jeweiligen Formular-Eigenschaften von *Form1* in die Registry geschrieben.

```
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  TRegIniFile& Ini = *new TRegIniFile("Software\\TABU\\Tourplaner");
  Ini.WriteInteger("Position","Links",Left);
  Ini.WriteInteger("Position","Oben",Top);
  Ini.WriteInteger("Position","Breite",Width);
  Ini.WriteInteger("Position","Höhe",Height);
  delete &Ini;
}
```

9.4 Anlegen neuer Projekte

Als nächstes sollen die Funktionen zum Anlegen neuer Projekte implementiert werden. Wie vorher bereits erwähnt wurde, soll dieses Programm als MDI-Anwendung erstellt werden. Zunächst müssen also die Voraussetzung dafür geschaffen werden, MDI-Kindfenster erstellen zu können.

9.4.1 MDI-Rahmenformular und Kindfenster

Wir wollen zunächst *Form1* in ein MDI-Rahmenformular abändern. Dazu muß eigentlich nur die Eigenschaft *FormStyle* in *fsMDIForm* geändert werden. Um die Kindfenster entsprechend verwalten zu können, werden einige Menüpunkte unter dem Menütitel *Fenster* erstellt und mit den entsprechenden *OnClick*-Ereignisbehandlungsroutinen verknüpft.

```
void __fastcall TForm1::Nebeneinander1Click(TObject *Sender)
{
  TileMode = tbVertical;
  Tile();
}
```

```
void __fastcall TForm1::Untereinander1Click(TObject *Sender)
{
  TileMode = tbHorizontal;
  Tile();
}

void __fastcall TForm1::berlappend1Click(TObject *Sender)
{
  Cascade();
}

void __fastcall TForm1::AlleAnordnen1Click(TObject *Sender)
{
  ArrangeIcons();
}
```

Des weiteren ist es üblich, dem Fenster-Menü die Namen der jeweils geöffneten Fenster als Menüpunkte anzuhängen, um auch auf diese Weise zwischen den einzelnen Fenstern wechseln zu können. Versuchen Sie hier bitte keine großartigen Konstruktionen, es ist lediglich erforderlich, die Eigenschaft *WindowMenu* von *Form1* auf den Wert *Fenster1* zu setzen, alles andere wird dann automatisch erledigt.

```
void __fastcall TForm1::Neu1Click(TObject *Sender)
{
  CreateMDI(Sender);
}

void __fastcall TForm1::CreateMDI(TObject *Sender)
{
  if (Form4->ShowModal() == mrOk)
  {
    Application->CreateForm(__classid(TChild), &Child);
    Child->Caption = ("Neues Projekt");
  }
}
```

Die Trennung der Funktionen in *Neu1Click* und *CreateMDI* erlaubt es uns, später auch von anderen Stellen aus Kindfenster zu erzeugen, ohne alle Anweisungen kopieren zu müssen (wenn später dann etwas geändert wird, dann werden gerade diese Kopien gerne vergessen, was zu entsprechenden Fehlfunktionen führen kann).

Bei *TChild* handelt es sich um das Kindfenster. Erzeugen Sie dafür eine neues Formular und weisen Sie der Eigenschaft *Name* den String *Child* zu. Des weiteren muß die Eigenschaft *FormStyle* auf *fsMDIChild* gesetzt werden. Damit Sie die Kindfenster dann auch wieder schließen können, ist die folgende Anweisung erforderlich.

```
void __fastcall TChild::FormClose(TObject *Sender, TCloseAction
&Action)
{
  Action = caFree;
}
```

Normalerweise werden alle Formulare beim Programmstart erstellt – dies soll bei einem Kindfenster jedoch nicht geschehen. Wählen Sie deshalb OPTIONEN | PROJEKT | FORMULARE und das Formular *Child*. Mit dem >-Button wird es von der Liste der automatisch erstellten Formulare in die Liste der verfügbaren Formulare transferiert.

Es ist möglich, daß die kommenden Schritte für Ihre eigenen Projekte nicht mehr verwendbar sind. Deshalb sollten Sie das bisherige Projekt in einem eigenen Verzeichnis abspeichern, um später an dieser Stelle weitermachen zu können.

9.4.2 Formular zum Erstellen von Projekten

Bei dieser Beispiel-Anwendung werden die Tourplaner-Projekte fest mit jeweils einem MDI-Kindfenster verbunden. Soll ein anderes Projekt geöffnet werden, dann wird auch ein neues Kindfenster erstellt. Die Kindfenster werden so aufgebaut sein, daß die Tabelle *Shows.DB* mit den Tabellen *Termine.DB* und *Orte.DB* mittels einer Master-Detail-Verknüpfung verbunden werden.

Es ist jedoch nicht möglich, Daten wie den Projekttitel oder die Tabelle *Beteil.DB* im Kindfenster zu editieren. Es wäre zwar möglich gewesen, auch diese Funktionen im Kindfenster zu implementieren, doch dann wäre dieses Formular noch unübersichtlicher geworden. Zum Editieren dieser Daten werden deshalb eigene Fenster verwendet. Zum Neuanlegen und zum Öffnen von Projekten werden separate Fenster benutzt. Wird ein neues Projekt angelegt, dann wird zunächst das Formular *Form4* modal geöffnet. Wird dieses mit *OK* geschlossen, dann wird ein neues Kindfenster erzeugt.

Bild 9.8 zeigt das Formular *Form4*. Beachten Sie bitte, daß es sich bei den Dialogelementen nicht um die datensensitiven Varianten handelt. Wenn Sie eigene Projekte auf dieses Grundgerüst aufbauen, dann werden Sie vermutlich die Edit-Felder zur Betitelung der drei Hervorhebungs-Kategorien nicht benötigen, hingegen werden Sie vielleicht andere Daten in der Projekt-Tabelle speichern wollen. Mit dem *OK*-Button ist die folgende Prozedur verbunden.

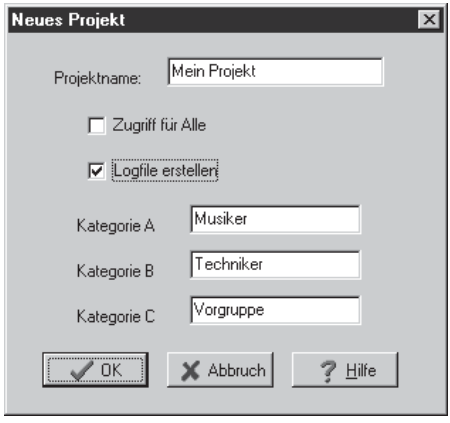


Bild 9.8: Dialogfenster zum Anlegen neuer Projekte

```
void __fastcall TForm4::BitBtn1Click(TObject *Sender)
// OK-Button
{
  if (Edit1->Text == "")
    ModalResult = mrAbort;
  else
  {
    Form1->Table1->Append();
    Form1->Table1->FieldByName("Besitzer")->AsInteger
      = Form2->Table1->FieldByName("Nummer")->AsInteger;
    Form1->Table1->FieldByName("Namen")->AsString = Edit1->Text;
    Form1->Table1->FieldByName("Oeffentlich")->AsBoolean =
      CheckBox1->Checked;
    Form1->Table1->FieldByName("Logfile")->AsBoolean =
      CheckBox2->Checked;
    Form1->Table1->FieldByName("A")->AsString = Edit2->Text;
    Form1->Table1->FieldByName("B")->AsString = Edit3->Text;
    Form1->Table1->FieldByName("C")->AsString = Edit4->Text;
    Form1->Table1->FieldByName("Datum")->AsDateTime = Now();
    Form1->Table1->Post();
```

Wenn ein neues Projekt geöffnet wird, dann werden zunächst einige Daten in die Tabelle *Projekt.DB* geschrieben. Zunächst wird ermittelt, welche Nummer der Anwender hat, der sich momentan bei diesem Programm angemeldet hat – diese Nummer wird in das Feld *Besitzer* geschrieben. Die weiteren Daten werden gemäß den Angaben in den Dialog-Elementen in die Tabelle geschrieben. In das Feld *Datum* werden das aktuelle Datum und die aktuelle Uhrzeit geschrieben.

```
  if (CheckBox2->Checked == true)
  {
    Form1->Table2->Open();
    Form1->Table2->Append();
    Form1->Table2->FieldByName("Projekt")->AsInteger =
      Form1->Table1->FieldByName("Nummer")->AsInteger;
    Form1->Table2->FieldByName("User")->AsInteger =
      Form2->Table1->FieldByName("Nummer")->AsInteger;
    Form1->Table2->FieldByName("Login")->AsDateTime = Now();
    Form1->Table2->FieldByName("Write")->AsBoolean = true;
    Form1->Table2->Post();
    Form1->Table2->Close();
  }
} // TForm4::BitBtn1Click
```

Für den Fall, daß ein Logfile erstellt werden soll, werden die Angaben Projektnummer, Benutzernummer und die aktuelle Zeit in die Tabelle *Log.DB* geschrieben. Da das Erstellen eines Projektes immer ein Schreibvorgang ist, wird das Feld *Write* entsprechend gesetzt.

Es wird davon ausgegangen, daß in den meisten Fällen kein Logfile geführt wird. Um das Starten des Programms zu beschleunigen und Ressourcen zu sparen, ist deshalb *Form1->Table2* normalerweise geschlossen und wird nur bei Bedarf geöffnet.

9.4.3 Formular zum Öffnen von Projekten

Beim Öffnen eines Projektes wird *Form5* modal aufgerufen, anschließend wird – sofern *Form5* mit *OK* geschlossen wurde – die Funktion *CreateMDI* aufgerufen.

```
void __fastcall TForm1::ffnen1Click(TObject *Sender)
// Projekt öffnen
{
  if (Form5->ShowModal() == mrOk)
    CreateMDI(Sender);
}
```

Bild 9.9 zeigt das Formular zum Öffnen von Projekten. In der Tabelle werden die vorhandenen Projekte aufgelistet, wobei die zuletzt geöffneten Projekte zuoberst angezeigt werden. Das *DBGrid* ist mit eine *TQuery*-Komponente in *Form1* verbunden, deren SQL-Anweisung wie folgt lautet:

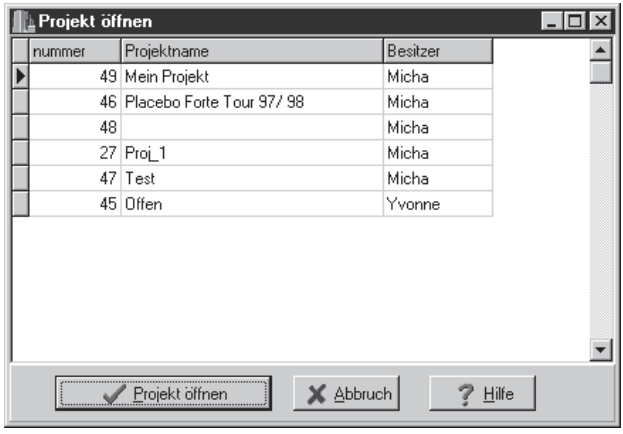


Bild 9.9: Formular zum Öffnen von Projekten

```
SELECT p.nummer, p.namen, u.namen, p.datum
  FROM projekt p, "user" u
  WHERE u.nummer = p.besitzer
  ORDER BY datum DESC
```

Das Sortieren der Einträge erfolgt mit der Anweisung ORDER BY, mit DESC wird festgelegt, daß die höchsten – in diesem Fall die spätesten – Einträge zuerst angezeigt werden.

Zum Öffnen des Formulars muß zunächst sichergestellt werden, daß der Anwender überhaupt eine Zugriffsberechtigung auf dieses Projekt besitzt. Ist dies der Fall, dann wird der Eigenschaft *ModalResult* der Wert *mrOK* zugewiesen, so daß ein Kindfenster geöffnet wird. Gegebenenfalls wird ein Eintrag in das Logfile geschrieben.

```
void __fastcall TForm5::BitBtn1Click(TObject *Sender)
{
  ModalResult = mrNone;
  bool Zugriff = false;
  int user;
  TZugriff z;
  TVarRec q(int(Form1->Query1->FieldByName("Nummer")->AsInteger));
  Form1->Table1->FindKey(&q, 0);
```

Zunächst wird der Datenzeiger der Projekt-Tabelle *Form1.Table1* auf das zu öffnende Projekt gesetzt. Außerdem sind einige Variablen zu deklarieren.

```
  if (Form1->Table1->FieldByName("Besitzer")->AsInteger == user)
```

```
    Zugriff = true;
  if (Form1->Table1->FieldByName("Oeffentlich")->AsBoolean == true)
    Zugriff = true;
  z = Form1->Zugriff(3, user);
  if(Zugriff == true)
    z.lesen = false;
  if (z.lesen == true)
  {
    Zugriff = true;
    FSchreiben = z.schreiben;
    FLogfile = z.logfile;
  }
  else
  {
    FSchreiben = true;
    FLogfile = true;
  }
```

Im nächsten Abschnitt wird geprüft, ob der Anwender eine Zugriffsberechtigung für dieses Projekt hat – wenn dies der Fall ist, wird die boolesche Variable *Zugriff* auf *true* gesetzt. Eine Zugriffsberechtigung besteht zum einen, wenn der Anwender dieses Projekt angelegt hat – wenn also das Feld *Besitzer* gleich der Variablen *user* ist – oder wenn das Projekt als öffentliches deklariert wurde. Drittens besteht die Möglichkeit, daß dem Anwender explizit das Zugriffsrecht gewährt worden ist. Dies wird von der *Form1*-Methode *Zugriff* überprüft, welche ein Ergebnis vom Typ *TZugriff* zurückgibt. Aus diesem Ergebnis kann dann auch ersehen werden, ob der Anwender Schreibberechtigung hat und ob er das Logfile einsehen darf, die Felder *FSchreiben* und *FLogfile* werden entsprechend gesetzt.

Der Anwender, der das Projekt angelegt hat, hat immer Schreibberechtigung und darf auch stets das Logfile einsehen. Das gleiche gilt für alle Anwender, wenn das Projekt öffentlich ist – dementsprechend werden die Felder *FSchreiben* und *FLogfile* in diesen beiden Fällen stets auf *true* gesetzt. Die Funktion *Zugriff* werden wir im Anschluß an diese Funktion besprechen.

```
  if (Zugriff == true)
  {
    Form1->Table1->Edit();
    Form1->Table1->FieldByName("Datum")->AsDateTime = Now();
    Form1->Table1->Post();
    Form1->Query1->Close();
    Form1->Query1->Open();
    ModalResult = mrOk;
```

Damit die Projekte immer beginnend mit dem letzten Zugriff angezeigt werden, werden dem Feld *Datum* der Tabelle *Projekt.DB* das aktuelle Datum und die aktuelle Uhrzeit zugewiesen, anschließend muß *Form1.Query1* aktualisiert werden.

```
    if (Form1->Query1->FieldByName("Logfile")->AsBoolean == true)
    {
      Form1->Table2->Insert();
      Form1->Table2->FieldByName("Projekt")->AsInteger
        = Form1->Query1->FieldByName("Nummer")->AsInteger;
      Form1->Table2->FieldByName("User")->AsInteger = user;
      Form1->Table2->FieldByName("Login")->AsDateTime = Now();
      Form1->Table2->Post();
    };
  } // if (Zugriff == true)
  else ShowMessage("Keine Zugriffsberechtigung für das Projekt");
} // TForm5::BitBtn1Click
```

Wird ein Logfile geführt, dann werden in dieses nun Projekt- und Benutzernummer sowie das aktuelle Datum und die aktuelle Uhrzeit geschrieben. Besteht keine Zugriffsberechtigung für das Projekt, dann soll eine entsprechende Fehlermeldung ausgegeben werden.

Das Öffnen des Projektes soll nicht nur mit *BitBtn1*, sondern auch mit einem Doppelklick auf das *DBGrid* ausgelöst werden können. Dazu weist man dessen Ereignis *OnDblClick* im Objektinspektor die Prozedur *BitBtn1Click* zu. Alternativ könnte man auch folgendermaßen formulieren:

```
void __fastcall TForm5::DBGrid1DblClick(TObject *Sender)
{
  BitBtn1Click(Sender);
}
```

Die Methode Zugriff

Mit der *TForm1*-Methode *Zugriff* wird ermittelt, ob ein Benutzer für ein Projekt das Leserecht hat, ob er auch Schreibzugriff hat und ob er das Logfile einsehen darf. Da Funktionen keine Arrays zurückgeben können, muß hier eine eigene Struktur definiert werden, die drei boolesche Felder beinhaltet.

```
struct TZugriff
{
  bool lesen, schreiben, logfile;
};
```

Der Funktion *Zugriff* werden zwei Parameter übrgeben: zum einen die Projektnummer und zum anderen die Benutzernummer desjenigen Anwenders, der auf das Projekt zugreifen möchte.

```
TZugriff __fastcall TForm1::Zugriff(int projekt, int user)
{
  TZugriff result;
  Query2->Close();
  Query2->ParamByName("Nummer")->AsInteger = user;
  Query2->Open();
  Query2->First();
  while(!Query2->Eof)
  {
    if(Query2->FieldByName("User")->AsInteger == user)
    {
      result.lesen = true;
      result.schreiben = Query2->FieldByName("Write")->AsBoolean;
      result.logfile = Query2->FieldByName("Logfile")->AsBoolean;
      break;
    }
    Query2->Next();
  } // while(! Query2->Eof == true)
  return result;
} // TForm1::Zugriff
```

Der Eigenschaft *SQL* von *Query2* wird folgende Anweisung zugewiesen:

```
SELECT * FROM zugriff
  WHERE projekt = :nummer
```

Dementsprechend wird zunächst dem Parameter *nummer* der Inhalt des Funktions-Parameters *Projekt* zugewiesen. Da in der Regel nicht sehr viele Zugriffsberechtigungen auf ein und dasselbe Projekt vergeben werden, kann hier ohne weiteres inkremental nach der Benutzernummer gesucht werden. Ist diese vorhanden, wird das Feld *lesen* des zurückzugebenden Ergebnisses (*result*) auf *true* gesetzt. Die Felder *schreiben* und *logfile* werden je nach Inhalt der Felder *Write* und *Logfile* gesetzt. Die Prozedur kann dann mit der Anweisung *break* abgebrochen werden.

9.5 Die Projektverwaltung

Wie vorhin bereits festgestellt wurde, kann von den Kindfenstern aus nicht auf die Tabelle *Projekt.DB* zugegriffen werden. Wir wollen deshalb ein neues Formular entwickeln, mit dem dieser Zugriff möglich ist. Dabei soll gleichzeitig eine Möglichkeit für den Zugriff auf die Tabelle *User.DB* geschaffen werden – von diesem Formular aus soll es also möglich sein, weiteren Benutzern den Zugriff auf die Projekte zu gestatten.

Bild 9.10 zeigt, wie dieses Formular gestaltet wird. In unserem Beispielprojekt wird in diesem Formular auch die Teilnehmerliste bearbeitet, eine entsprechende *TabbedNotebook*-Seite ist in Bild 9.10 bereits zu sehen.

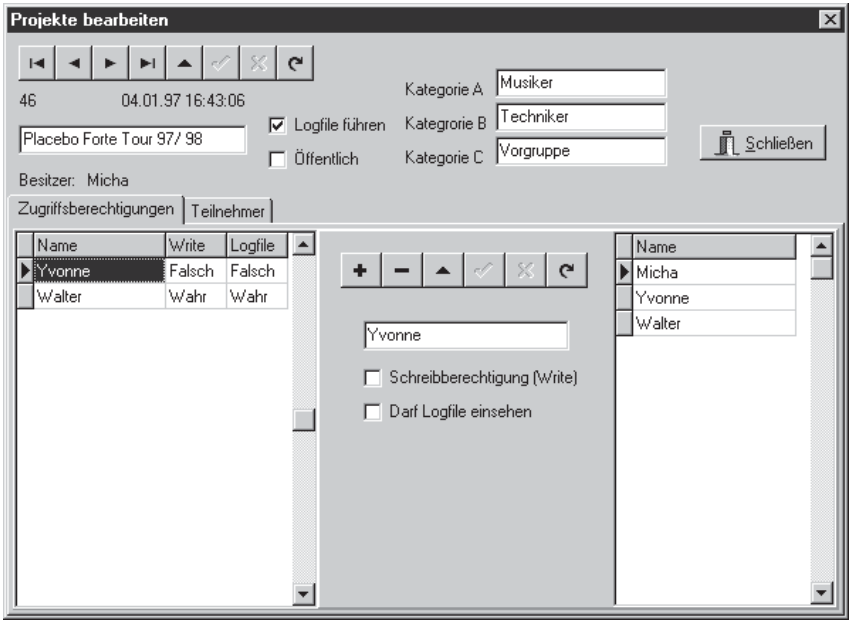


Bild 9.10: Das Formular zur Projektverwaltung

Zur Bearbeitung der Tabelle *Projekt.DB* werden eine *TTable*- und eine *TDataSource*-Komponente, des weiteren die entsprechenden datensensitiven Dialogelemente sowie ein *DBNavigator* benötigt. Das Bearbeiten der Tabelle *Projekt.DB* wird von diesen Komponenten ohne zusätzlichen Quelltext ermöglicht. Es muß lediglich noch sichergestellt werden, daß nur der Benutzer auf die Projektdaten zugreift, der dieses Projekt auch angelegt hat.

```
void __fastcall TForm10::DataSource2DataChange(TObject *Sender,
TField *Field)
{
  if(Form2->Table1->FieldByName("Nummer")->AsInteger
    == Table1->FieldByName("Besitzer")->AsInteger)
    Panel2->Visible = true;
  else
    Panel2->Visible = false;
} // TForm10::DataSource2DataChange
```

Zu diesem Zweck sind die Komponenten auf der *TabbedNotebook*-Seite *Zugriffsberechtigungen* auf *Panel2* zusammengefaßt. Bei jedem *DataChange*-Ereignis von *DataSource2* wird nun geprüft, ob der angemeldete Anwendender nun auch Besitzer des jeweiligen Projektes ist. Ist dies nicht der Fall, dann wird die Eigenschaft *Visible* von *Panel2* auf *false* gesetzt, die darauf enthaltenen Komponenten sind somit auch nicht sichtbar.

```
void __fastcall TForm10::Table1AfterEdit(TDataSet *DataSet)
{
  if(Panel2->Visible == false)
  {
    ShowMessage
      ("Sie haben keine Zugriffsberechtigung für dieses Projekt");
    Table1->Cancel();
  }
} // TForm10::Table1AfterEdit
```

Versucht der Anwender, einen Datensatz der Tabelle *Projekt.DB* zu editieren, dann wird geprüft, ob die Eigenschaft *Visible* von *Panel2* gleich *true* ist, ob also der Anwender derjenige Benutzer ist, der das Projekt angelegt hat. Ist dies nicht der Fall, dann wird eine entsprechende Fehlermeldung ausgegeben und die Bearbeitung von *Table1* mit *Cancel* abgebrochen. Mit der Eigenschaft *VisibleButtons* von *TDBNavigator* kann festgelegt werden, welche Buttons angezeigt werden und welche nicht. In diesem Fall werden die Buttons *nbInsert* und *nbDelete* ausgeschlossen, denn das Einfügen und das Löschen von Projekten ist an dieser Stelle nicht vorgesehen.

```
void __fastcall TForm10::Table1AfterPost(TDataSet *DataSet)
{
  Form1->Query1->Close();
  Form1->Query1->Open();
}
```

Damit die Veränderungen an der Tabelle *Projekt.DB* überhaupt von *Form1.Query1* wahrgenommen werden, muß diese Datenzugriffskomponente geschlossen und wieder geöffnet werden.

Wenn Sie Bild 9.10 aufmerksam betrachtet haben, dann haben Sie festgestellt, daß der Benutzername in Klartext in einer *TDBText*-Komponente enthalten ist, obwohl in der Tabelle *Projekt.DB* nur die Benutzernummer gespeichert ist. Um das zu erreichen ist es erforderlich, eine neue *TQuery*-Komponente einzufügen, welche mit *Table1* über eine Master-Detail-Beziehung verbunden ist – die Eigenschaft *DataSource* von *Query1* muß dann auf *DataSource1* gesetzt werden. Die SQL-Anweisung wird dann folgendermaßen formuliert:

```
SELECT u.namen
  FROM "user" u, projekt p
  WHERE (u.nummer = p.besitzer)
    AND (p.nummer = :nummer)
```

Bei der Erstellung des Datenmodells wurde darauf hingewiesen, daß es nicht günstig ist, SQL-Schlüsselwörter als Spalten- und Tabellennamen zu verwenden. In dieser Anweisung ist beispielsweise das Wort *User* ein SQL-Schlüsselwort und muß deshalb in Anführungszeichen gesetzt werden.

9.5.1 Die Zugriffsverwaltung

Wenn Sie die *TabbedNotebook*-Seite für die Zugriffsberechtigungen betrachten, dann sehen Sie links eine Liste der bislang vergebenen Zugriffsberechtigungen für dieses Projekt. Auch in dieser Liste stehen die Benutzernamen in Klartext, so daß auch hier ein JOIN von zwei Tabellen erforderlich ist. Hier lautet dann die SQL-Anweisung wie folgt:

```
SELECT z.projekt, z."user", u.namen, z."write", z."logfile"
  FROM zugriff z, "user" u
  WHERE (z.projekt = :nummer)
    AND (u.nummer = z."user")
```

Da hier nur die Spalten *u.namen, z.write* und *z.logfile* im *DBGrid* angezeigt werden sollen, müssen die anderen Spalten mit dem Spalteneditor (Doppelklick auf *DBGrid1*) ausgeschlossen werden. Wird bei der SQL-Anweisung einer *TQuery*-Komponente ein JOIN verwendet, dann eignet sich diese *TQuery*-Komponente nicht dafür, Datensätze in die Tabelle einzufügen, zu editieren oder zu löschen. Aus diesem Grund muß zusätzlich eine *TTable*-Komponente eingefügt werden, welche auf die Tabelle *Zugriff.DB* zugreift.

Zunächst muß hierbei sichergestellt werden, daß die *TQuery*- und die *TTable*-Komponenten den gleichen Datensatz anzeigen. Zu diesem Zweck werden die But-

tons *First*, *Prior*, *Next* und *Last* von *DBNavigator2* entfernt, so daß auf dieser Seite keine Änderung des Datensatzes erfolgen kann. Um einen Datensatz anzuwählen, wird auf die entsprechende Zeile im *DBGrid* ein Mausklick ausgeführt – die Prozedur *DataSource3DataChange* sorgt dann dafür, daß bei *Table2* der korrekte Datensatz ausgewählt wird.

```
void __fastcall TForm10::DataSource3DataChange(TObject *Sender,
  TField *Field)
// Table2 und Query2 koppeln
{
  if(Table2->State == dsBrowse)
  {
    Table2->FindKey(OPENARRAY(TVarRec,
      (int(Query2->FieldByName("Projekt")->AsInteger),
      int(Query2->FieldByName("User")->AsInteger))));
    Edit1->Text = Query2->FieldByName("Namen")->AsString;
  }
} // DataSource3DataChange
```

Da in der Tabelle *Zugriff.DB* nur die Benutzernummer und nicht der Benutzername steht, kann letzterer nicht einfach über ein datensensitives Dialogelement angezeigt werden. Bei der Tabelle *Projekt.DB* wurde hier eine zusätzliche *TQuery*-Komponente eingefügt. Hier soll nun ein anderer Weg gegangen werden: Da der Spalteninhalt von der SQL-Abfrage bereits erfaßt wird, soll hier eine normale *TEdit*-Komponente verwendet werden, welcher dann der Inhalt direkt zugewiesen wird.

Um nun bei der Komponente *Table2* den Datenzeiger auf den richtigen Datensatz zu setzen, wird die Methode *FindKey* verwendet. Da bei der Tabelle *Zugriff.DB* ein zusammengesetzter Primärschlüssel verwendet wird, müssen hier Projekt- und Usernummer als Parameter übergeben werden. Die Verwendung der Methode *FindKey* würde zu einer Exception führen, wenn die Datenmenge inaktiv ist – beispielsweise beim Programmstart. Deshalb wird die Eigenschaft *State* von *Table2* daraufhin überprüft, ob sie den Wert *dsBrowse* aufweist.

```
void __fastcall TForm10::Table2AfterInsert(TDataSet *DataSet)
{
  Table2->FieldByName("Projekt")->AsInteger
    = Table1->FieldByName("Nummer")->AsInteger;
  Table2->FieldByName("User")->AsInteger
    = Table3->FieldByName("Nummer")->AsInteger;
  Edit1->Text = Table3->FieldByName("Namen")->AsString;
}
```

Soll nun ein neuer Datensatz eingefügt werden – also eine neue Zugriffsberechtigung erteilt werden –, dann müssen Projekt- und Benutzernummer gesetzt werden. Während sich die Projektnummer aus dem aktuellen Projekt ergibt, wird die Benutzernummer aus dem *DBGrid* übernommen, welches die Namen der Tabelle *User.DB* auflistet. Der jeweilige Benutzername wird in *Edit1* geschrieben.

```
void __fastcall TForm10::Table2AfterPost(TDataSet *DataSet)
{
  Query2->Close();
  Query2->Open();
}
```

Damit das *DBGrid*, welches die vorhandenen Zugriffsberechtigungen anzeigt, das Einfügen, Ändern oder Löschen eines Datensatzes berücksichtigt, muß *Query2* geschlossen und wieder geöffnet werden. Die entsprechenden Anweisungen werden mit den Ereignissen *OnPost* und *OnDelete* verknüpft.

Wenn während des Einfügens oder Änderns eines Datensatzes in der Tabelle *Zugriff.DB* ein anderer Benutzername ausgewählt wird, dann müssen dessen Nummer der Spalte *User* und dessen Name *Edit1* zugewiesen werden.

```
void __fastcall TForm10::DataSource5DataChange(TObject *Sender,
  TField *Field)
// Neuer Benutzername ausgewählt
{
  if((Table2->State == dsInsert) + (Table2->State == dsEdit))
  {
    Table2->FieldByName("User")->AsInteger
      = Table3->FieldByName("Nummer")->AsInteger;
    Edit1->Text = Table3->FieldByName("Namen")->AsString;
  }
}
```

9.6 Das Logfile anzeigen

Zuletzt soll noch ein Formular erstellt werden, mit dem das Logfile angezeigt werden kann. Dazu werden lediglich eine *TQuery*- und eine *TDataSource*-Komponente sowie ein *DBGrid* und ein *DBNavigator* benötigt. Der Komponente *TQuery* wird die folgende SQL-Anweisung zugewiesen, mit dem Spalteneditor werden dann die nicht benötigten Spalten ausgeblendet.

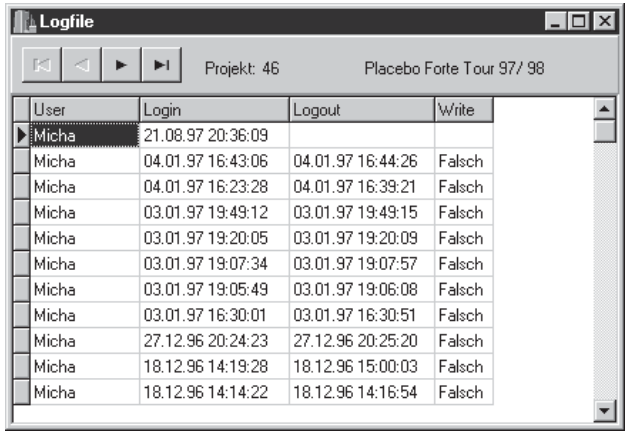


Bild 9.11: Anzeigen des Logfiles

```
SELECT l.projekt, p.namen, u.namen, l.login, l.logout, l."write"
    FROM log l, "user" u, projekt p
  WHERE (projekt = :pn) AND (p.nummer = l.projekt)
    AND (u.nummer = l."user")
  ORDER BY l.login DESC
```

Das Formular *Form12* wird vom jeweiligen Kindfenster aus aufgerufen, mit dem entsprechenden Menüpunkt wird die folgende Prozedur verbunden:

```
void __fastcall TChild::Logfileanzeigen1Click(TObject *Sender)
{
  if(FLogfile == true)
  {
    Form11->Query1->Close();
    Form11->Query1->ParamByName("pn")->AsInteger = FProjekt;
    Form11->Query1->Open();
    Form11->Show();
  }
  else
    ShowMessage("Keine Berechtigung zum Einsehen des Logfiles");
}
```

Zunächst wird ermittelt, ob das Feld *FLogfile* gleich *true* ist, andernfalls hat der momentane Benutzer keine Zugriffsberechtigung für das Logfile und sein Ansinnen wird mit einer entsprechenden Fehlermeldung abgewiesen. Darf das Logfile eingesehen werden, dann wird dem Parameter *pn* die Projektnummer zugewiesen und *Query1* durch das Schließen und anschließende Öffnen aktualisiert.

10 Die Adressen-Datenbank

Bevor mit der eigentlichen Tourplaner-Anwendung begonnen wird, soll zunächst noch die Adressen-Datenbank implementiert werden. Diese verwaltet die Adressen, welche später den einzelnen Shows zugeordnet werden. Es ist daher nötig, erst einige Adressen einzugeben, bevor diese dann eingefügt werden können. Auch ist es nicht ganz auszuschließen, daß Sie auf den Stand von Grundgerüst plus Adressen-Verwaltung bei einem späteren Projekt werden aufbauen können.

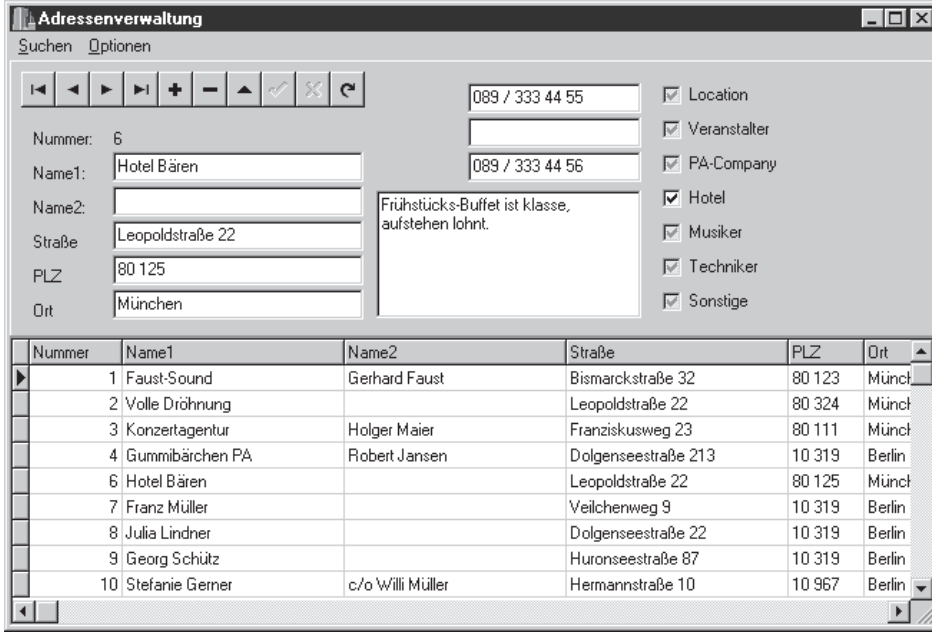


Bild 10.1: Formular für die Adressen-Datenbank

Bestücken Sie ein Formular gemäß Bild 10.1. Die Komponenten im oberen Panel werden dabei mit *Datasource1 / Table1* verbunden, das *DBGrid1* dagegen mit *Datasource2 / Query1*. Diese Vorgehensweise hat folgenden Grund: Mit *DBGrid1* sollen Abfrageergebnisse angezeigt werden, mit der oberen Datenbankmaske dagegen Eingaben vorgenommen werden. Wie man bewirkt, daß der aktuelle Da-

tensatz von *DBGrid1* auch in der Datenbankmaske angezeigt wird, werden wir gleich behandeln. Vorläufig sollen in *DBGrid1* alle Datensätze angezeigt werden. Der Eigenschaft *SQL* von *Query1* wird demnach (über den Objektinspektor) folgende Anweisung zugewiesen:

```
SELECT * FROM adressen
```

Wenn Sie mit SQL-Anweisungen noch nicht vertraut sind, dann sollten Sie zunächst Kapitel 3 lesen.

10.1 TTable und TQuery koppeln

Um den aktuellen Datensatz von *DBGrid1* in der Eingabemaske anzuzeigen, sollen zwei Möglichkeiten vorgesehen werden: Zum einen soll mit dem Menüpunkt *EditMaskeaktualisieren1* der aktuelle Datensatz von *DBGrid1* in die Maske übernommen werden, zum anderen soll mit dem Menupunkt *EditTabellekoppeln1* die Option gewählt werden können, die bewirkt, daß die Maske automatisch den aktuellen Datensatz von *DBGrid1* anzeigt.

```
void __fastcall TForm6::Editmaskeaktualisieren1Click(TObject
*Sender)
{
  TVarRec q(int(Query1->FieldByName("Nummer")->AsInteger));
  Table1->FindKey(&q, 0);
}
```

Mit der *TTable*-Methode *FindKey* kann nach einem Wert im aktuellen Index – normalerweise also im Primärindex – gesucht werden. Primärindex ist hier das Feld *Nummer*.

```
void __fastcall TForm6::Edittabellekoppeln1Click(TObject *Sender)
{
  Edittabellekoppeln1->Checked = !Edittabellekoppeln1->Checked;
}
```

Wenn *Query1* und *Table1* fest gekoppelt werden sollen, dann wird die Eigenschaft *EditTabellekoppeln1->checked* auf *true* gesetzt. Die Aktualisierung muß genau dann erfolgen, wenn ein anderer Datensatz ausgewählt wird. Das *TDataSource*-Ereignis *OnDataChange* wird unter anderem dann ausgelöst, wenn ein neuer Datensatz ausgewählt wird, die entsprechenden Anweisungen werden deshalb mit diesem Ereignis verknüpft.

```
void __fastcall TForm6::DataSource2DataChange(TObject *Sender,
  TField *Field)
{
  if(Edittabellekoppeln1->Checked == true)
    Editmaskeaktualisieren1Click(Sender);
}
```

10.2 Suchen

Um nach einem oder mehreren Datensätzen zu suchen, muß lediglich der Eigenschaft *SQL* der Komponente *Query1* eine SQL-Anweisung zugewiesen werden. Da es möglich wäre, daß die Anweisung in mehreren Schritten geschrieben wird, wird bei einer Änderung der Eigenschaft *SQL* die Eigenschaft *Active* automatisch auf *false* gesetzt. Nach der Zuweisung der SQL-Anweisung muß deshalb die Eigenschaft *Active* auf *true* gesetzt werden.

Alle Datensätze anzeigen

Zunächst soll der Menü-Punkt implementiert werden, der eine Anzeige aller Datensätze bewirkt:

```
void __fastcall TForm6::AlleDatenstzeanzeigen1Click(TObject *Sender)
{
  Query1->SQL->Clear();
  Query1->SQL->Add("SELECT * FROM adresse");
  Query1->Open();
}
```

Zunächst werden alle bisherigen SQL-Zeilen gelöscht, danach erfolgt die Anweisung, alle Datensätze der Tabelle *adresse* anzuzeigen, und schließlich wird die Datenmenge geöffnet. Für all diejenigen, welche Kapitel 3 übersprungen haben: Groß- und Kleinschreibung ist bei SQL-Anweisungen beliebig, es ist aber Konvention, alle SQL-Schlüsselwörter mit Großbuchstaben und alles andere mit Kleinbuchstaben zu schreiben.

10.2.1 Das SQL-Eingabeformular

In der Regel werden Sie *Suchen*-Formulare erstellen, welche beim Anwender keine SQL-Kenntnisse voraussetzen; die entsprechenden Optionen werden dann über Edit-Felder, CheckBoxen und RadioButtons einstellbar sein.

Nun werden aber unter Garantie einige Anwender nach Kriterien oder vor allem Kombinationen von Kriterien suchen wollen, die Sie bei der Erstellung der Formulare nicht vorgesehen haben. Hier gibt es dann drei Möglichkeiten:

- Eine entsprechende Abfrage ist nicht möglich und der Anwender ärgert sich über das Programm.
- Sie entwickeln eine eigene Abfragesprache. Das ist nicht nur eine hervorragende Arbeitsbeschaffungsmaßnahme, es garantiert auch, daß der Anwender noch nie damit zu tun hatte und es auch keine Literatur auf dem Markt gibt, mit deren Hilfe man sich kundig machen könnte, wenn Sie bei der Beschreibung dieser Abfragesprache eine weniger glückliche Hand hatten.
- Sie ermöglichen die direkte Eingabe von SQL-Anweisungen; das ist so simpel, daß man es in fünf Minuten erledigen kann. (Wenn das Eingabeformular allerdings so komfortabel wie hier ausfallen soll, dann dauert es schon etwas länger.)

Für einen einfachen SQL-Editor benötigen Sie lediglich ein Formular, eine *TRichEdit*-Komponente sowie einen Button, mit dem dann die Ausführung der Anweisung gestartet wird. Hier in diesem Beispiel soll noch die Möglichkeit implementiert werden, die SQL-Anweisungen zu speichern und zu laden (von der Festplatte oder aus der Zwischenablage). Außerdem sollen über das Menü Beispiel-SQL-Anweisungen abrufbar sein, in welche der Anwender dann nur noch die Suchbegriffe seinen Bedürfnissen entsprechend abändern muß; für so etwas sind insbesondere Einsteiger sehr dankbar.

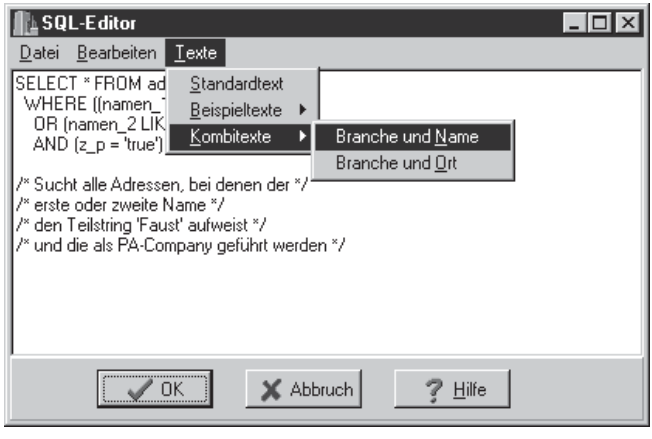


Bild 10.2: Das SQL-Eingabe-Formular

```
void __fastcall TForm6::SQLFenster1Click(TObject *Sender)
{
  if (Form7->ShowModal() == mrOk)
  {
    Query1->SQL = Form7->RichEdit1->Lines;
    Query1->Open();
  }
} // TForm6::SQLFenster1Click
```

Mit dem Menüpunkt *SQLFenster* wird *Form7* modal geöffnet. Wird das SQL-Fenster mit dem *OK*-Button geschlossen, dann wird der Eigenschaft *SQL* von *Query1* der Inhalt von *RichEdit1* des SQL-Fensters zugewiesen und die Abfrage geöffnet. Mehr ist für ein funktionierendes SQL-Fenster nicht erforderlich, alles andere ist eigentlich Luxus.

Das Datei-Menü des SQL-Editors

SQL-Anweisungen können bisweilen recht umfangreich und kompliziert werden. Da die Wahrscheinlichkeit hoch ist, daß der Anwender dieselben oder ähnliche SQL-Anweisungen mehrmals verwenden möchte, soll zunächst die Möglichkeit geschaffen werden, den Text von *RichEdit1* zu speichern, zu laden und zu drucken.

```
void __fastcall TForm7::Neu1Click(TObject *Sender)
{
  RichEdit1->Clear();
}
```

Mit dem Befehl *Neu* wird eine neue SQL-Anweisung begonnen, in *RichEdit1* werden dann alle vorherigen Einträge gelöscht. Auf eine Sicherheitsabfrage wurde hier verzichtet, schließlich kann mit dem Menüpunkt BEARBEITEN | RÜCKGÄNGIG die SQL-Anweisung wiederhergestellt werden.

```
void __fastcall TForm7::ffnen1Click(TObject *Sender)
{
  if(OpenDialog1->Execute() == true)
  {
    RichEdit1->Lines->LoadFromFile(OpenDialog1->FileName);
    AltDatAkt(OpenDialog1->FileName);
  }
} // TForm7::ffnen1Click
```

Mit dem Befehl *Öffnen* wird eine gespeicherte Datei geöffnet, auf den Befehl *AltDatAkt* kommen wir später zurück.

```
void __fastcall TForm7::Speichernunter1Click(TObject *Sender)
{
  if(SaveDialog1->Execute() == true)
  {
    RichEdit1->Lines->SaveToFile(SaveDialog1->FileName);
    AltDatAkt(SaveDialog1->FileName);
  }
} // TForm7::Speichernunter1Click
```

Ähnlich funktioniert das Speichern in einer Datei; die Methoden von *TRichEdit* machen obige Operationen recht einfach.

```
void __fastcall TForm7::Druckereinrichtung1Click(TObject *Sender)
{
  PrintDialog1->Execute();
}
```

Noch einfacher ist der Aufruf des Dialogs *PrinterSetup*.

```
void __fastcall TForm7::Drucken1Click(TObject *Sender)
{
  if(PrintDialog1->Execute() == true)
  {
    RichEdit1->Print("SQL-Text");
  }
} // TForm7::Drucken1Click
```

Um den SQL-Text auszudrucken – dies kann vor allem bei der Fehlersuche recht nützlich sein – wird zunächst *PrintDialog1* aufgerufen. Durch die Methode *Print* wird das Eigentliche Drucken sehr einfach – dies ist auch der Grund, aus dem *TRichEdit* anstatt *TMemo* verwendet wurde.

Die Liste der alten Dateien

Gerade bei diesem SQL-Editor wird man wohl mit eher wenigen speziellen SQL-Anweisungen auskommen, die dann aber entsprechend häufiger verwendet werden. Um hier das Öffnen dieser Dateien ein wenig zu beschleunigen, soll dem Datei-Menü eine Liste der vier zuletzt verwendeten Datei-Namen angehängt werden, die mit einem Mausklick auf den dazugehörenden Eintrag geöffnet werden können.

Wie diese Liste aussieht, zeigt Bild 10.3. Um diese Option zu implementieren, benötigen wir zunächst eine StringListe, in der die entsprechenden Dateinamen gespeichert werden.

```
TStringList& AlteDateien = *new TStringList();
```

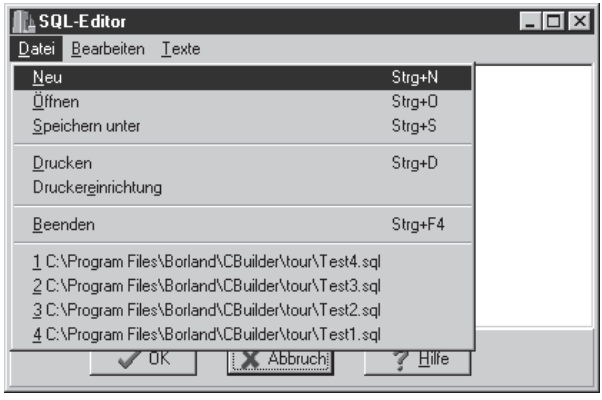


Bild 10.3: Die Menü-Liste der alten Dateien

Die folgende Prozedur aktualisiert diese Liste bei jedem Speichern und Öffnen:

```
void __fastcall TForm7::AltDatAkt(AnsiString FileName)
{
  if(AlteDateien.IndexOf(FileName) > -1)
    AlteDateien.Delete(AlteDateien.IndexOf(FileName));
  AlteDateien.Insert(0, FileName);
  if(AlteDateien.Count > 4)
    AlteDateien.Delete(4);
  while(Datei1->Count > 9)
    Datei1->Delete(9);
  int i;
  for(i=0; i < AlteDateien.Count; i++)
  {
    TMenuItem& NewItem = *new TMenuItem(Form7);
    NewItem.Caption = "&" + IntToStr(i+1)+ ' ' +
      AlteDateien.Strings[i];
    NewItem.OnClick = AltLadenClick;
    Datei1->Add(&NewItem);
  }
} // TForm7::AltDatAkt
```

Zunächst wird die StringListe *AlteDateien* daraufhin geprüft, ob der als Parameter übergebene Dateiname bereits darin enthalten ist – ist dies der Fall, dann wird er gelöscht. Anschließend wird er an der ersten Stelle (Index 0) neu eingefügt. Wird dadurch die Länge der Liste größer vier, dann wird das fünfte Element (Index 4) gelöscht. Auf diese Weise sind nie mehr als vier Einträge in der Liste enthalten.

Danach werden die alten Einträge im Menü *Datei1* entfernt. Da nicht feststeht, wie viele Einträge bislang an das Menü angehängt worden sind, wird hier eine *while*-Schleife verwendet. Nun werden die (maximal) vier Menüpunkte dem Menü *Datei1* angehängt; dazu wird zunächst ein Menü-Eintrag erstellt. Diesem wird der Dateiname als Titel zugewiesen, vorangestellt wird eine unterstrichene (&) Indexnummer. Als Ereignisbehandlungsroutine wird *AltLadenClick* zugewiesen, danach wird der Menüpunkt dem Menü *Datei1* hinzugefügt.

```
void __fastcall TForm7::AltLadenClick(TObject *Sender)
// Mausklick auf die Liste der alten Dateien im Datei-Menü
{
  TFileName Datei;
  Datei = dynamic_cast<TMenuItem*>(Sender)->Caption;
  Datei.Delete(1,3);
  RichEdit1->Lines->LoadFromFile(Datei);
  AltDatAkt(Datei);
}
```

Für die Prozedur *AltLadenClick* muß man ein wenig in die Trickkiste der objektorientierten Programmierung greifen; im Prinzip funktioniert die Prozedur folgendermaßen: Anhand des Parameters *Sender* wird festgestellt, welcher Menüpunkt das Ereignis abgesetzt hat; dessen Eigenschaft *Caption* enthält den benötigten Dateinamen, wegen der vorangestellten, unterstrichenen Ziffer aber erst ab der vierten Stelle. Mit diesem Dateinamen wird die Datei geladen und abschließend *AltDatAkt* aufgerufen.

Nun ist aber der Parameter *Sender* vom Typ *TObject*, und dieser enthält keine Eigenschaft *Caption*. Somit muß der Parameter *Sender* erst einer Typenumwandlung unterzogen werden, zu diesem Zweck wird die Anweisung *dynamic_cast* verwendet.

Das ganze funktioniert schon recht brauchbar, leider geht der Inhalt des Arrays *AlteDateien* bei jedem Programmende verloren. Um diesen zu speichern, verwendet man bei 32-Bit-Programmen die Registry.

```
void __fastcall TForm7::FormDestroy(TObject *Sender)
{
  TRegIniFile& Ini = *new TRegIniFile("Software\\TABU\\Tourplaner");
  int i;
  for(i=0; i < AlteDateien.Count; i++)
    Ini.WriteString("Alte Dateien", "Datei " + IntToStr(i+1),
      AlteDateien.Strings[i]);
  delete &Ini;
  delete &AlteDateien;
}
```

Das Ereignis *FormDestroy* wird dann ausgelöst, wenn das Formular aus dem Speicher entfernt wird – beim diesem SQL-Editor ist dies bei Beenden der Anwendung der Fall. Zum Zugriff auf die Registry wird hier das Objekt *TRegIniFile* verwendet, welches vor der Verwendung erzeugt und hinterher wieder gelöscht werden muß.

```
void __fastcall TForm7::FormCreate(TObject *Sender)
{
  TRegIniFile& Ini = *new TRegIniFile("Software\\TABU\\Tourplaner");
  int i;
  AnsiString s;
  for(i=1; i<5; i++)
  {
    s = Ini.ReadString("Alte Dateien", "Datei " + IntToStr(i), "");
    if(s != "")
      AlteDateien.Add(s);
  }
  if(AlteDateien.Count > 0)
    AltDatAkt(AlteDateien.Strings[0]);
  delete &Ini;
}
```

Wenn das Formular erzeugt wird, dann werden dementsprechend die Einträge aus der Registry in die Stringliste *AlteDateien* geschrieben. Die Funktion *FormCreate* stellt hier nicht fest, wie viele Einträge es in der Registry gibt, sondern versucht einfach, vier Dateinamen auszulesen. Wird dabei ein leerer String zurückgegeben, dann existiert ein entsprechender Eintrag nicht und kann dann auch nicht der Stringliste hinzugefügt werden. Zum Anhängen der Einträge an das Menü wird die Funktion *AltDatAkt* verwendet.

Das Menü Bearbeiten

```
void __fastcall TForm7::Ausschneiden1Click(TObject *Sender)
{
  RichEdit1->CutToClipboard();
}

void __fastcall TForm7::Kopieren1Click(TObject *Sender)
{
  RichEdit1->CopyToClipboard();
}
```

```
void __fastcall TForm7::Einfgen1Click(TObject *Sender)
{
  sperren = true;
  RichEdit1->PasteFromClipboard();
  Application->ProcessMessages();
  sperren = false;
  RichEdit1Change(Sender);
}

void __fastcall TForm7::AllesAuswhlen1Click(TObject *Sender)
{
  RichEdit1->SelectAll();
}
```

Die Komponente *TRichEdit* verfügt erfreulicherweise schon über alle Methoden, die zum Handling der Zwischenablage benötigt werden. Die zusätzlichen Anweisungen bei *Einfgen1Click* ergeben sich aus den Anforderungen der Storno-Funktion. Für diese gibt es leider keine *TRichEdit*-Methode, ihre Realisierung gestaltet sich somit auch ein wenig aufwendiger.

```
void __fastcall TForm7::RichEdit1Change(TObject *Sender)
{
  if (sperren == false)
  {
    LinesAlt.Assign(&LinesNeu);
    LinesNeu.Assign(RichEdit1->Lines);
  }
}
```

Eine Storno-Funktion stellt nach einer unerwünschten Aktion (beispielsweise nach versehentlichem Überschreiben eines markierten Blocks) den alten Zustand wieder her; zu diesem Zweck ist es erforderlich, den alten Zustand zu speichern. Prinzipiell müßte die Prozedur *RichEdit1Change* – sie wird immer dann aufgerufen, wenn in *RichEdit1* etwas geändert wurde – jeweils nach jeder Änderung die alte Fassung in die Stringliste *LinesAlt* schreiben; nun kann aber diese Prozedur nicht feststellen, wie diese alte Fassung ausgesehen hat.

Von daher ist es notwendig, eine zweite Stringliste zu führen (*LinesNeu*). Bei einer Änderung wird zunächst *LinesNeu* (das ist sozusagen die alte Fassung) nach *LinesAlt* kopiert, danach werden die Zeilen von *RichEdit1* nach *LinesNeu* kopiert.

Da es sich bei den Typen *TStrings* und *TStringList* – ersterer kann nur innerhalb von Komponenten verwendet werden – um Objekte handelt, wird die Zuweisung nicht mit =, sondern mit *Assign* durchgeführt.

Nun wird auch beim Ausführen der Storno-Funktion *RichEdit1* geändert; würden in diesem Fall auch die beiden Stringlisten aktualisiert werden, wäre ein Chaos das Ergebnis. Deshalb kann man das Aktualisieren der Stringlisten vermeiden, indem man die globale Variable *sperren* auf *true* setzt.

```
void __fastcall TForm7::Rckgngig1Click(TObject *Sender)
{
  sperren = true;
  RichEdit1->Lines->Assign(&LinesAlt);
  sperren = false;
  RichEdit1Change(Sender);
}
```

Soll die letzte Änderung storniert werden, dann müssen einfach nur die Zeilen von *LinesAlt* zurück nach *Memo1* geschrieben werden, vorher jedoch muß – wie gerade erwähnt – die Aktualisierung der Stringlisten während dieser Aktion verhindert werden. Nach Abschluß des Kopierens wird die Variable *sperren* wieder auf *false* gesetzt. Um auch die Stornierung stornieren zu können, wird nun die Prozedur *RichEdit1Change* aufgerufen, die beiden Stringlisten können nun aktualisiert werden.

```
void __fastcall TForm7::FormShow(TObject *Sender)
{
  LinesNeu.Assign(RichEdit1->Lines);
}
```

Damit die Storno-Funktion von Anfang an funktioniert, muß der Inhalt von *RichEdit1* der Stringliste *LinesNeu* zugewiesen werden.

Das Menü Texte

Um die Arbeit mit dem SQL-Editor zu vereinfachen, insbesondere, um dem Anfänger den Einstieg zu erleichtern, sollen einige SQL-Standard-Anweisungen zu einem Menü zusammengefaßt werden. Dessen *OnClick*-Ereignisbehandlungsprozedur sieht dann beispielsweise folgendermaßen aus:

```
void __fastcall TForm7::Standardtext1Click(TObject *Sender)
{
  sperren = true;
  RichEdit1->Lines->Clear();
  RichEdit1->Lines->Add("SELECT * FROM adresse");
  sperren = false;
  RichEdit1Change(Sender);
}
```

Weitere Zeilen der SQL-Anweisung werden dann mit der folgenden Anweisung hinzugefügt:

```
RichEdit1->Lines->Add(" {Anweisung} ");
```

Nun könnte ein solcher SQL-Text versehentlich ausgewählt werden. Würde der Anwender nun versuchen, mit der Funktion *Rückgängig* die vorige SQL-Anweisung wiederherzustellen, würde er nur die letzte Zeile der jetzigen Anweisung löschen – denn die Funktion *Rückgängig* storniert die jeweils letzte Anweisung, und das wäre in diesem Fall eine *RichEdit1->Lines->Add*-Anweisung gewesen.

Um während des Einfügens der vordefinierten SQL-Texte die Aktualisierung der beiden Stringlisten zu vermeiden, wird die Variable *sperren* vorher auf *true* und hinterher auf *false* gesetzt. Um nach dem Einfügen des SQL-Textes die Stringlisten zu aktualisieren, wird dann die Prozedur *RichEdit1Change* aufgerufen.

Im folgenden sind nun die einzelnen SQL-Anweisungen aufgeführt, auf die Prozedur-Rahmen wurde dabei jeweils verzichtet.

```
SELECT * FROM ADRESSEN
WHERE (name1 LIKE "%Faust%")
OR (name2 LIKE "%Faust%")

/* Sucht alle Adressen, bei denen der */
/* erste oder zweite Name den */
/* Teilstring »Faust« aufweist */
```

Kommentare werden bei SQL-Anweisungen zwischen /* und */ gesetzt.

```
SELECT * FROM ADRESSEN
WHERE ort = "Berlin"
/* Sucht alle Berliner Adressen */

SELECT * FROM ADRESSEN
WHERE (tel1 LIKE "030%513%")
OR (tel2 LIKE "030%513%")
/* Sucht alle Berliner Adressen */
/* (Vorwahl 030), deren erste oder zweite */
/* Telefonnummer die Ziffernfolge 513 enthält */

SELECT * FROM ADRESSEN
WHERE fax LIKE "030%513%"
/* Sucht alle Berliner Adressen */
/* (Vorwahl 030), deren Faxnummer*/
/* die Ziffernfolge 513 enthält */
```

```
SELECT * FROM ADRESSEN
WHERE z_p = "true"
/* Sucht alle PA-Companys */

SELECT * FROM ADRESSEN
WHERE ((name1 LIKE "%Faust%")
OR (name2 LIKE "%Faust%"))
AND (z_p = "true")
/* Sucht alle Adressen, bei denen der */
/* erste oder zweite Name den */
/* Teilstring »Faust« aufweist */
/* und als »PA-Company« geführt werden */

SELECT * FROM ADRESSEN
WHERE (z_p = "true")
AND (Ort = "Berlin")
/* Sucht alle PA-Companys */
/* aus Berlin*/
```

10.2.2 Das Suchen-Formular

Auch wenn mit einem SQL-Editor wohl alle auftretenden Abfragen gelöst werden können, sollte dieser nicht die einzige Möglichkeit zur Suche sein. Schließlich ist das Erstellen von SQL-Anweisungen relativ aufwendig und fehleranfällig (eine vergessene Klammer, und das Programm reagiert mit einer Fehlermeldung).

Von daher sollte man ein normales Suchen-Formular vorsehen, welches zum eingegebenen Suchbegriff die korrekte SQL-Anweisung erstellt. Wie das hier aussieht, zeigt Bild 10.4. Es kann hier ausgewählt werden, ob der Suchstring als Vollstring, Anfang oder Teilstring zu interpretieren ist. Wird beispielsweise nach dem String *Licht* gesucht, dann dürfte bei der Suche nach einem Vollstring der Feldinhalt nur aus diesem Wort bestehen (selbst ein nachgestelltes Leerzeichen wäre hier schon problematisch, ebenso eine abweichende Groß- und Kleinschreibung).

Da hier der Suchstring direkt in eine SQL-Anweisung umgesetzt wird, kann man auch SQL-Zeichen eingeben. Beispielsweise könnte man *%üller* eingeben, um sowohl *Müller* als auch *müller* sowie alle weiteren Strings zu erhalten, welche mit *müller* enden. Eine Eingabe *(%üller) or (%aier)* würde dagegen nicht zum gewünschten Ergebnis führen, denn die daraus erstellte SQL-Anweisung würde lauten:

```
SELECT * FROM adressen WHERE namen1 LIKE '(%üller) or (%aier)'
```

Wird nach einem Teilstring gesucht, dann muß der eingegebene String irgendwo (zusammenhängend) in dem entsprechenden Feld vorkommen, ob am Anfang, am Ende oder irgendwo dazwischen, spielt dabei keine Rolle. Wird nach einem Anfang gesucht, dann können noch weitere Zeichen folgen. Dies ist besonders dann hilfreich, wenn nicht feststeht, inwieweit beispielsweise ein Firmenname ausgeschrieben ist (Tabu, Tabu Licht- und Tontechnik, Tabu GbR usw).

Des weiteren läßt sich die Suche auf bestimmte Felder beschränken. Wenn eine Firma *Maier* gesucht wird, dann möchte man keine zehn Firmen, die in einer Oskar-Maier-, Fridolin-Maier- oder Otto-Maier-Straße zu finden sind. Des weiteren wird der aus den Vorgaben erzeugte SQL-String angezeigt; dies kann insbesondere dann hilfreich sein, wenn das Ergebnis nicht ganz wie gewünscht ausfällt. Anhand der SQL-Anweisung wird meist sehr schnell deutlich, wo der Hund begraben liegt.

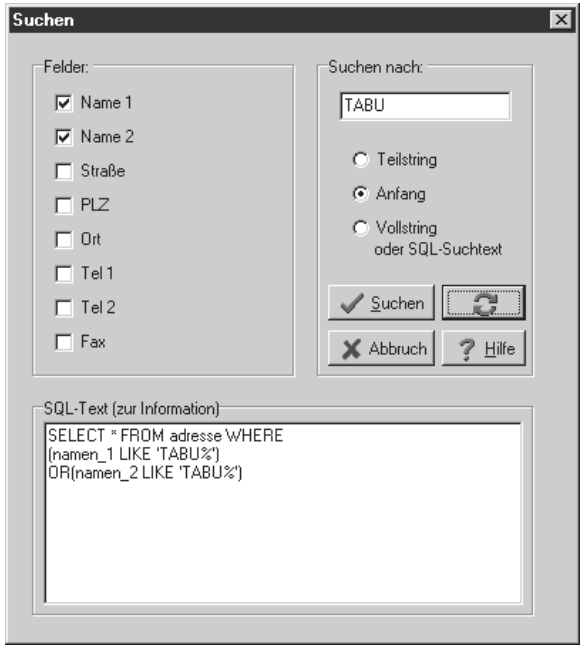


Bild 10.4: Das Suchen-Formular

```
void __fastcall TForm8::BitBtn2Click(TObject *Sender)
{
  AnsiString s;
```

```
  erster = true;
  Memo1->Clear();
  Memo1->Lines->Add("SELECT * FROM adresse WHERE");

  // Name 1
  if(CheckBox1->Checked == true)
  {
    if(RadioButton1->Checked)
      s = "(namen_1 LIKE '%" + Edit1->Text + "%')";
    if(RadioButton2->Checked)
      s = "(namen_1 LIKE '" + Edit1->Text + "%')";
    if(RadioButton3->Checked)
      s = "(namen_1 LIKE '" + Edit1->Text + "')";
    if(erster == false)
      Memo1->Lines->Add("OR" + s);
    else
      Memo1->Lines->Add(s);
    erster = false;
  } // if(CheckBox1->Checked == true)

  // Name 2
    ...
  // Straße
    ...
  // PLZ
    ...
  // Ort
    ...
  // Tel 1
    ...
  // ´Tel 2
    ...
  // Fax
    ...
} // TForm8::BitBtn2Click
```

Mit der Prozedur *BitBtn2Click* werden die Vorgaben in eine SQL-Anweisung umgesetzt; dieser Vorgang ist einzeln durchführbar, damit vor der Suche geprüft werden kann, ob das Ergebnis den Erwartungen entspricht.

Die Prozedur ist nicht weiter interessant, es muß lediglich noch geprüft werden, ob das jeweilige Feld das erste Feld ist, das angewählte wurde, denn ab da muß ein OR vor die jeweilige Zeile gesetzt werden.

```
void __fastcall TForm8::BitBtn1Click(TObject *Sender)
{
  BitBtn2Click(Sender);
}
```

Das Suchen-Formular wird wieder modal geöffnet. Damit beim Betätigen des *OK*-Buttons die SQL-Anweisung generiert wird, muß hier explizit *BitBtn2Click* aufgerufen werden.

10.2.3 Suchen nach Telefonnummern

Wie aus Kapitel 3 bekannt ist, gestaltet sich die (fehlerfreie) Suche nach Telefonnummern etwas aufwendiger. Zu diesem Zweck soll ein eigenes Suchen-Formular erstellt werden.

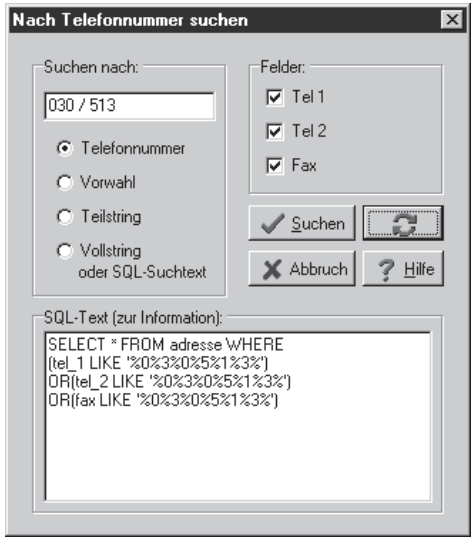


Bild 10.5: Suche nach Telefonnummern

Im wesentlichen gleichen sich die Anweisungen denen des vorigen Suchen-Formulars, lediglich die Suche nach einer *Telefonnummer* ist neu hinzugekommen:

```
if(RadioButton1->Checked)
// Suche nach Telefonnummer
{
  t = Edit1->Text;
  s = "";
```

```
      for(i=0;i<t.Length(); i++)
      {
        char c = t[i+1];
        if(('0' <= c) * (c <= '9'))
          s = s + c + "%";
      };
      s = "(tel_1 LIKE '%" + s + "')";
    }; // if(RadioButton1->Checked)
```

Der Suchstring *Edit1->Text* wird in einzelne Buchstaben zerlegt. Handelt es sich dabei um eine Ziffer, dann wird sie dem String *s* hinzugefügt; zusätzlich wird ein *%*-Zeichen eingefügt. Auf diese Weise ist es unerheblich, ob eine Telefonnummer in der Form *030 / 513 45 05* oder in der Form *0305134505* eingegeben wird. Näheres siehe Kapitel 3.

10.2.4 Nach Branche filtern

Zuletzt soll noch ein Untermenü erstellt werden, welches das Filtern nach den einzelnen Branchen erlaubt. Die entsprechende Prozedur bedarf wohl keiner Erläuterung mehr.

```
void __fastcall TForm6::Hotel1Click(TObject *Sender)
{
  Query1->SQL->Clear();
  Query1->SQL->Add("SELECT * FROM adresse");
  Query1->SQL->Add("  WHERE z_h = 'true'");
  Query1->Open();
}
```

11 Weitere Funktionen

Erst zu diesem Zeitpunkt beginnen wir mit der eigentlichen Anwendung. Bevor wir das Kindfenster vervollständigen, wollen wir zunächst die Methode *CreateMDI* des Rahmenformulars erweitern. Bislang hat diese Funktion lediglich ein Kindfenster erzeugt und diesem den Titel *Neues Projekt* zugewiesen.

Noch nicht alle Anweisungen der Methode *CreateMDI* sind zu diesem Zeitpunkt schon möglich. So ist beispielsweise eine Komponente *DBCheckBox1* noch nicht vorhanden, folglich kann ihr auch keine Beschriftung zugewiesen werden. Um *CreateMDI* im Laufe dieses Kapitels nicht mehrmals behandeln zu müssen, soll sie hier vollständig abgedruckt werden. Soll das Projekt gestartet werden, dann müssen diese Anweisungen durch Kommentarzeichen stillgelegt werden.

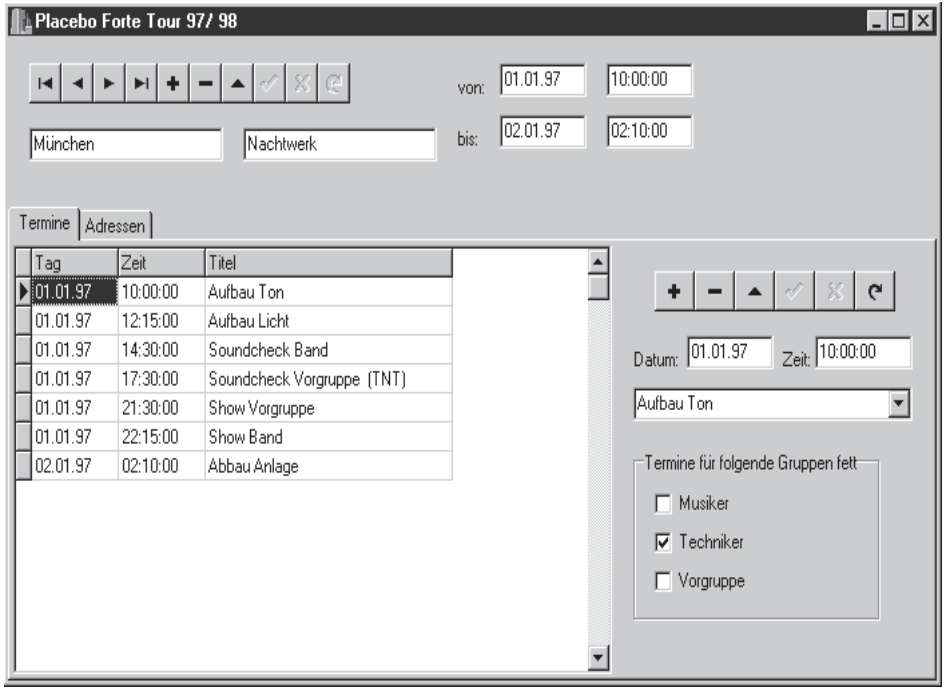


Bild 11.1: Das Kindfenster

```
void __fastcall TForm1::CreateMDI(TObject *Sender)
// Neues Kindfenster öffnen
{
  AnsiString s;
  Application->CreateForm(__classid(TChild), &Child);
  Child->Caption = Table1->FieldByName("Namen")->AsString;
  Child->FProjekt = Table1->FieldByName("Nummer")->AsInteger;
  if(Sender == Neu1)
  {
    Child->FSchreiben = true;
    Child->FLogfile = Table1->FieldByName("Logfile")->AsBoolean;
  }
  else
  {
    Child->FSchreiben = Form5->FSchreiben;
    Child->FLogfile = Form5->FLogfile;
  }
  if(Child->FLogfile == true)
    Child->FLogNum = Table2->FieldByName("Nummer")->AsInteger;
```

Als Titel wird nun der Projektname verwendet. Des weiteren müssen die Felder *FProjekt* (Projektnummer), *FSchreiben* (Schreibberechtigung) und *FLogfile* (Logfile führen) gesetzt werden. Diese Felder sind in der Header-Datei als öffentliche Felder zu deklarieren.

Die Vorgehensweise beim Setzen der Felder unterscheidet sich beim Öffnen von Projekten vom Neuanlegen von Projekten. Letzteres geschieht definitionsgemäß vom Besitzer des Projektes, der somit auch immer die Schreibberechtigung hat. Bei Öffnen dagegen wird die Schreibberechtigung dagegen von der Methode *Zugriff* festgestellt und im Feld *FSchreiben* von *Form5* zwischengespeichert.

Wird ein Logfile geführt, dann muß beim Verlassen des Projektes dorthin geschrieben werden, wann der Anwender das Projekt verläßt, und ob er Änderungen vorgenommen hat. Da hier eine MDI-Anwendung erstellt wird, können zwischenzeitlich andere Projekte geöffnet worden sein, was auch zu anderen Einträgen in der Tabelle *LOG.DB* geführt haben kann. Da man sich nicht darauf verlassen kann, daß der Datenzeiger noch auf dem Datensatz steht, auf dem er beim Einloggen gestanden hat, muß man dafür sorgen, daß der Datenzeiger auf diesen Datensatz explizit gesetzt wird.

Prinzipiell könnte man diesen Datensatz anhand der Projektnummer und des Log-In-Datums finden, einfacher geht es jedoch, wenn man beim Einloggen die Datensatznummer zwischenspeichert. Auch hier bestätigt sich wieder die Regel, daß man sich viel Arbeit spart, wenn man eine fortlaufende Nummer als Primärschlüssel vergibt.

```
  Child->Query1->Close();
  Child->Query1->ParamByName("Projekt")->AsInteger
    = Child->FProjekt;
  Child->Query1->Open();
  s = Table1->FieldByName("A")->AsString;
  if(s != "")
  {
    Child->DBCheckBox1->Caption = s;
    Child->DBCheckBox4->Caption = s;
  }
  s = Table1->FieldByName("B")->AsString;
  if(s != "")
  {
    Child->DBCheckBox2->Caption = s;
    Child->DBCheckBox5->Caption = s;
  }
  s = Table1->FieldByName("C")->AsString;
  if(s != "")
  {
    Child->DBCheckBox3->Caption = s;
    Child->DBCheckBox6->Caption = s;
  }
  Child->FormActivate(Sender);
} // TForm1::CreateMDI
```

Die Projektnummer wird *Query1* des Kindfensters als Parameter zugewiesen, anschließend wird die Abfrage neu geöffnet. Des weiteren werden – sofern die Kategorien *A*, *B* und *C* definiert sind – diese Strings den Komponenten *DBCheckBox1* bis *DBCheckBox6* zugewiesen. Zur Methode *FormActivate* kommen wir gleich.

Zunächst soll nun ermöglicht werden, der Tabelle *Shows.DB* weitere Zeilen hinzuzufügen. Die dazu benötigten Komponenten werden auf einem *Panel* plaziert, dessen Eigenschaft *Align* auf *alTop* geändert wird. Nun werden zwei *TDBEdit*-, eine *TDBNavigator*-, zwei *TLabel*- und vier *TEdit*-Komponenten darauf plaziert (siehe Bild 11.1).

Die Datensteuerungskomponenten werden über eine *TDataSource*-Komponente mit einer *TQuery* verbunden, deren SQL-Anweisung wie folgt formuliert wird:

```
SELECT * FROM shows
  WHERE projekt = :projekt
```

Dem Parameter *:projekt* wurde die jeweilige Projektnummer bereits von der Funktion *CreateMDI* zugewiesen.

```
void __fastcall TChild::Query1AfterEdit(TDataSet *DataSet)
{
  if(FSchreiben == false)
  {
    ShowMessage
      ("Sie haben keine Schreibberechtigung für dieses Projekt");
    Query1->Cancel();
  }
}
```

Soll ein Datensatz verändert werden, dann wird zunächst geprüft, ob der Anwender überhaupt die Schreibberechtigung für dieses Projekt besitzt. Andernfalls wird eine Fehlermeldung ausgegeben und die Datenmenge mit der Methode *Cancel* wieder in den Status *dsBrowse* zurückversetzt. Diese Anweisung wird bei mehreren Datenzugriffskomponenten im Kindfenster notwendig sein und soll nicht jedesmal angedruckt werden.

```
void __fastcall TChild::Query1AfterInsert(TDataSet *DataSet)
{
  if(FSchreiben == false)
  {
    ShowMessage
      ("Sie haben keine Schreibberechtigung für dieses Projekt");
    Query1->Cancel();
  }
  else
  {
    Query1->FieldByName("Projekt")->AsInteger = FProjekt;
    DBEdit1->SetFocus();
  } // else(FSchreiben == false)
} // TChild::Query1AfterInsert
```

Auch beim Versuch, Datensätze einzufügen, muß die Schreibberechtigung überprüft werden. Des weiteren muß dem Feld *Projekt* die jeweilige Projektnummer zugewiesen werden. Anschließend wird der Komponente *DBEdit1* der Fokus übertragen, dies erspart dem Anwender einen Mausklick.

```
void __fastcall TChild::Query1AfterPost(TDataSet *DataSet)
{
  FGeschrieben = true;
}
```

Um einen entsprechenden Eintrag in das Logfile machen zu können, muß festgestellt werden, ob der Anwender am Projekt etwas verändert hat. Zu diesem Zweck dient das Feld *FGeschrieben*.

Anfangs- und Endtermin

Auf dem Panel befinden sich noch vier *TEdit*-Komponenten, welche den Anfangs- und Endtermin (Datum und Uhrzeit) der jeweiligen Show anzeigen. Diese beziehen ihre Daten aus der (momentan noch nicht vorhandenen) *DataSet*-Komponente *Query2*, welche mit der Tabelle *Termine.DB* verbunden ist.

```
void __fastcall TChild::DataSource1DataChange(TObject *Sender,
  TField *Field)
// Start- und Endtermin anzeigen
{
  if((Query1->State == dsBrowse)*(Query2->State == dsBrowse))
  {
    Query2->Last();
    Edit3->Text = Query2->FieldByName("Tag")->AsString;
    Edit4->Text = Query2->FieldByName("Zeit")->AsString;
    Query2->First();
    Edit1->Text = Query2->FieldByName("Tag")->AsString;
    Edit2->Text = Query2->FieldByName("Zeit")->AsString;
  } // if
} // TChild::DataSource1DataChange
```

Schließen des Kindfensters

Folgende Anweisungen sind beim Schließen des Kindfensters erforderlich:

```
void __fastcall TChild::FormClose(TObject *Sender,
  TCloseAction &Action)
{
  if(FLogfile == true)
  {
    TVarRec q(FLogNum);
    Form1->Table2->FindKey(&q, 0);
    Form1->Table2->Edit();
    Form1->Table2->FieldByName("Logout")->AsDateTime = Now();
    Form1->Table2->FieldByName("Write")->AsDateTime = FGeschrieben;
    Form1->Table2->Post();
  }
  Action = caFree;
}
```

Da das Kindfenster zur Laufzeit erzeugt wird, muß es auch explizit wieder entfernt werden, was mit der Anweisung *Action = caFree* geschieht. Wird ein Logfile geführt, dann werden der Logout-Zeitpunkt sowie die Information, ob Daten geändert worden sind dort festgehalten.

11.1 Editieren der Termine

Wie Bild 11.1 zeigt, besteht die *TabbedNotebook*-Seite *Termine* aus einem *DBGrid*, welches die Termine anzeigt, und aus *DBEdit*-Feldern, mit denen sich die Tabelle *Termine.DB* editieren läßt. Das *DBGrid* ist mit der Komponente *Query2* verbunden, deren SQL-Text wie folgt lautet:

```
SELECT * FROM termine
  WHERE show = :nummer
  ORDER BY tag, zeit, nummer
```

Die nicht benötigten Felder werden dann mit Hilfe des Spalteneditors von *DBGrid1* ausgeschlossen. Nun erlaubt diese SQL-Anweisung keine Änderung der zugrundeliegenden Tabelle. (Sie können das feststellen, indem Sie die *TQuery*-Eigenschaft *RequestLive* auf *true* setzen und dann die Laufzeit-Eigenschaft *CanModify* abfragen.)

Um die Tabelle ändern zu können, wird zusätzlich eine *TTable*-Komponente eingefügt, die mit der Tabelle *Termine.DB* verbunden wird. An diese Komponente werden dann über eine *TDataSource* die *DBEdit*-Felder angeschlossen. Mit der folgenden Anweisung wird dafür gesorgt, daß in den *DBEdit*-Feldern stets der Datensatz angezeigt wird, der auch im *DBGrid* ausgewählt ist.

```
void __fastcall TChild::DataSource2DataChange(TObject *Sender,
  TField *Field)
// TTable folgt TQuery
{
  if(Table1->State == dsBrowse)
  {
    TVarRec q(int(Query2->FieldByName("Nummer")->AsInteger));
    Table1->FindKey(&q, 0);
  }
}
```

Soll ein neuer Datensatz eingefügt werden, dann muß zunächst festgestellt werden, ob der Anwender überhaupt die Schreibberechtigung für dieses Projekt hat – die Information darüber wurde dem Feld *FSchreiben* zugewiesen. Wenn der Benutzer keine Zugriffsberechtigung hat, dann wird eine entsprechende Meldung ausgegeben und das Einfügen in die Datenmenge mit *Cancel* abgebrochen.

Des weiteren muß die aktuelle *Show*-Nummer dem entsprechenen Feld zugewiesen werden; diese kann aus *Query1* gelesen werden.

```
void __fastcall TChild::Table1AfterInsert(TDataSet *DataSet)
{
  if(FSchreiben == false)
  {
    ShowMessage
      ("Sie haben keine Schreibberechtigung für dieses Projekt");
    Table1->Cancel();
  }
  else
  {
    Table1->FieldByName("Show")->AsInteger
      = Query1->FieldByName("Nummer")->AsInteger;
```

Nun noch ein bißchen Luxus: In der Regel wird bei den meisten Terminen der Tag mit dem Tag des vorigen Termins übereinstimmen. Von daher soll dieser Wert gleich dem entsprechenden Feld zugewiesen werden. Außerdem sollen die Einträge in *DBGrid1* grau dargestellt werden, damit der Anwender gleich sieht, daß Daten eingefügt oder geändert werden können.

```
    if(Query2->RecordCount > 0)
    {
      Query2->Last();
      Table1->FieldByName("Tag")->AsString
        = Query2-> FieldByName("Tag")->AsString;
    }
    DBGrid1->Font->Color = clSilver;
  } // else(FSchreiben == false)
} // TChild::Table1AfterInsert
```

Die Prozedur *TChild::Table1AfterEdit* ist teilweise identisch *TChild::Table1After-Insert*: Auch hier wird auf die Schreibberechtigung hin geprüft und die Darstellung von *DBGrid1* geändert. Die Zuweisung der *Show*-Nummer und des Datums kann dagegen ersatzlos entfallen.

Wird das Einfügen oder Ändern eines Datensatzes mit *Post* bestätigt, dann muß zunächst *Query2* aktualisiert werden. Des weiteren wird die Darstellung von *DBGrid1* wieder in den ursprünglichen Zustand versetzt. Außerdem wird das Feld *FGeschrieben* auf *true* gesetzt – wird ein Logfile geführt, dann wird dort eingetragen, daß sich der Datensatz geändert hat.

```
void __fastcall TChild::Table1AfterPost(TDataSet *DataSet)
{
  Query2->Close();
  Query2->Open();
```

```
  DBGrid1->Font->Color = clBlack;
  FGeschrieben = true;
}
```

Die Prozedur *TChild::Table1AfterCancel* benötigt nur die Anweisung *DBGrid1->Font.Color = clBlack*. In *TChild::Table1AfterDelete* muß dagegen lediglich *Query2* aktualisiert werden.

11.1.1 Eingabehilfe für das Datum

Bei der Verwendung von Windows-Programmen ist es relativ ärgerlich und auch wenig effektiv, wenn laufend zwischen Maus und Tastatur gewechselt werden muß. Man sollte deshalb die Programme so erstellen, daß man möglichst vieles auch mit der Maus eingeben kann. (Auf der anderen Seite gibt es Situationen, in denen keine Maus vorhanden ist oder diese nicht richtig funktioniert – in solchen Fällen sollte sich das Programm vollständig mit der Tastatur bedienen lassen.)

Wir wollen nun eine Vorrichtung programmieren, die es erlaubt, das Datum per Mausklick aus einem Kalender auszuwählen. Dazu wird auf der rechten Seite der *TabbedNotebook*-Seite *Termine* (dort wo die *DBEdit*-Felder untergebracht sind) eine *TNotebook*-Komponente mit den Seiten *Termine*, *Kalender* und *Uhr* plaziert. Wird *Label3* angeklickt, dann soll auf die *Notebook*-Seite *Kalender* gewechselt werden, welche gemäß Bild 11.2 bestückt wird.

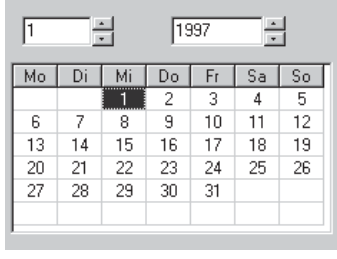


Bild 11.2: Der Kalender

Die Komponente *TCalendar* befindet sich auf der Palettenseite *Beispiele*. Ist diese Seite oder diese Komponente nicht vorhanden, dann muß die *Datei $(BCB)\Examples\Controls\Sampreg.cpp* in die Komponentendateiliste aufgenommen werden (KOMPONENTE | INSTALLIEREN).

Wird auf *Label3* ein Mausklick ausgeführt, dann wird zunächst ermittelt, ob sich *Table1* überhaupt im Modus *dsEdit* oder *dsInsert* befindet. Ist dies der Fall, dann wird zunächst die *Notebook*-Seite gewechselt. Ist bereits ein Datum vorhanden, dann soll dies im Kalender eingestellt werden. Dazu wird das Datum in die Wer-

te Tag, Monat und Jahr zerlegt, und diese Teilstrings werden den entsprechenden *TCalendar*-Eigenschaften zugewiesen. Des weiteren müssen auch noch die *TEdit*-Felder entsprechend gesetzt werden. Die Komponente *TCalendar* ist auf der Komponenten-Seite *Beispiele* zu finden.

```
void __fastcall TChild::Label3Click(TObject *Sender)
// "Eingabehilfe Datum" aufrufen
{
  if((Table1->State == dsEdit)+(Table1->State == dsInsert))
  {
    Notebook1->ActivePage = "Datum";
    if(DBEdit7->Text != "")
    {
      unsigned short year, month, day;
      DecodeDate(StrToDate(DBEdit7->Text), year, month, day);
      Calendar1->Year = year;
      Calendar1->Month = Month;
      Calendar1->Day = day;
      Edit5->Text = IntToStr(Calendar1->Month);
      Edit6->Text = IntToStr(Calendar1->Year);
    } // if(DBEdit3->Text != "")
  } // if((Table1->State == dsEdit)+(Tab
} // TChild::Label3Click
```

Nach einem Doppelklick auf den Kalender soll wieder die *Notebook*-Seite *Termine* angezeigt werden. Des weiteren wird das Kalenderdatum in einen String gewandelt und *DBEdit7* zugewiesen.

```
void __fastcall TChild::Calendar1DblClick(TObject *Sender)
// Verlassen der "Eingabehilfe Datum"
{
  AnsiString s;
  s = IntToStr(Calendar1->Day) + "."
    + IntToStr(Calendar1->Month) + "."
    + IntToStr(Calendar1->Year);
  Notebook1->ActivePage = "Termin";
  DBEdit7->Text = s;
}
```

Mit den beiden *TUpDown*-Komponenten lassen sich Monats- und Jahreszahl vergrößern oder verkleinern. Das *OnClick*-Ereignis beider Komponenten wird mit folgender Ereignisbehandlungsroutine verbunden:

```
void __fastcall TChild::UpDown1Click(TObject *Sender, TUDBtnType
Button)
// Monat oder Jahr in der "Eingabehilfe Datum" ändern
{
  Calendar1->Month = StrToInt(Edit5->Text);
  Calendar1->Year = StrToInt(Edit6->Text);
}
```

Die Eigenschaft *Associate* der beiden *TUpDown*-Komponenten muß auf *Edit5* beziehungsweise *Edit6* gesetzt werden. Des weiteren muß die Eigenschaft *Thousands* auf *false* gesetzt werden, damit sich die Funktion *StrToInt* nicht über Tausender-Trennzeichen beschwert. Außerdem müssen bei der *TUpDown*-Komponente für den Monat die Eigenschaften *Min* und *Max* auf *eins* beziehungsweise auf *zwölf* gesetzt werden.

11.1.2 Eingabehilfe für die Uhrzeit

Auch für die Uhrzeit soll eine Eingabehilfe programmiert werden. Dafür kann man aber leider nicht mehr auf eine fertige Komponente zurückgreifen werden. Wie Bild 11.3 zeigt, werden dafür zwei *TListBox*-Komponenten verwendet: Die eine zeigt alle Stunden an, die andere die Minuten im Fünf-Minuten-Takt – genauere Eingaben machen in der Regel ohnehin keinen Sinn.

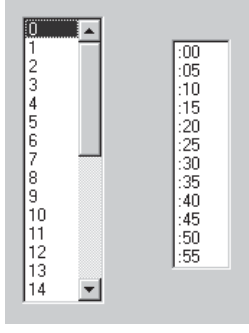


Bild 11.3: Die Uhr

Die Stunden-Zeiten reichen von 0:00 Uhr über 23:00 Uhr bis 6:00 Uhr. Mit einem Klick auf *Label4* wird die *Notebook*-Seite *Uhr* angewählt und die bislang vorhandene Uhrzeit in den beiden ListBoxen eingestellt (so genau dies eben möglich ist).

```
void __fastcall TChild::Label4Click(TObject *Sender)
// "Eingabehilfe Zeit" aufrufen
{
  if((Table1->State == dsEdit) + (Table1->State == dsInsert))
  {
    Notebook1->ActivePage = "Zeit";
    if(DBEdit8->Text == "")
      ListBox1->ItemIndex = 10;
    else
    {
      unsigned short hour, min, sec, msec;
      DecodeTime(StrToTime(DBEdit8->Text), hour, min, sec, msec);
      if(hour < 25)
        ListBox1->ItemIndex = hour;
      if(min < 60)
        ListBox1->ItemIndex = int(min/5);
    } //  else(DBEdit8->Text == "")
  } //  if((Table1->State = dsEdit) + (Tabl
} // TChild::Label4Click
```

Mit einem Doppelklick auf die Minuten-ListBox wechselt man zurück zur *Notebook*-Seite *Termin*. Außerdem wird die mit den ListBoxen eingestellte Uhrzeit in einen String gewandelt und *DBEdit8* zugewiesen.

```
void __fastcall TChild::ListBox2DblClick(TObject *Sender)
// "Eingabehilfe Zeit" schließen
{
  AnsiString s;
  s = ListBox1->Items->Strings[ListBox1->ItemIndex]
    + ListBox2->Items->Strings[ListBox2->ItemIndex];
  Notebook1->ActivePage = "Termin";
  DBEdit8->Text = s;
}
```

11.1.3 Die Nachschlageliste für die Termine

Wie in Bild 11.1 zu sehen ist, gibt es eine *DBComboBox*, mit welcher die einzelnen Tätigkeiten – im Beispiel *Aufbau Ton* – ausgewählt werden können. Selbstverständlich ist es auch möglich, andere Texte einzugeben. Wie schon bei der Erstellung des Datenmodells erwähnt wurde, sind die Texte jeweils einem Projekt zugeordnet.

Bezieht eine *ComboBox* ihre Daten aus einer Datenbank, dann wird für gewöhnlich eine *TDBLookUpCombo* verwendet, welche hier jedoch aus noch nicht geklärter Ursache nicht korrekt gearbeitet hat. Deshalb muß hier eine *TDBComboBox* verwendet werden, welche die Einträge dann jeweils aufnimmt.

Die SQL-Anweisung von *Query3* ähnelt der von *Query2*.

```
SELECT text FROM texte
  WHERE projekt = :projekt
```

Die folgenden Anweisungen werden beim Ereignis *OnActivate* ausgeführt, so daß die Liste aktualisiert wird, wenn die Tabelle *Texte.DB* geändert wird.

```
// Termintexte
Query3->Close();
Query3->ParamByName("Projekt")->AsInteger = FProjekt;
Query3->Open();
DBComboBox1->Items->Clear();
while(!Query3->Eof)
{
  DBComboBox1->Items->Add(Query3->FieldByName("Text")->AsString);
  Query3->Next();
}
```

11.2 Das Fenster für die Einträge

Bevor wir mit der *TabbedNotebook*-Seite *Adressen* fortfahren, soll das Fenster erstellt werden, mit dem die Tabelle *Texte.DB* bearbeitet werden kann. Identisch aufgebaute Fenster werden auch für die Bearbeitung anderer Tabellen benötigt, beispielsweise für *Adrtex.DB*. Hier liegt es nahe, nur ein einziges Formular zu verwenden, und die darin enthaltene SQL-Anweisung jeweils entsprechend abzuändern.

Da die Texte stets einem Projekt zugeordnet sind, muß die Möglichkeit geschaffen werden, eins der Projekte auszuwählen. Dazu dient eine *TTable / TDataSource*-Kombination, welche mit der Tabelle *Projekte.DB* verbunden ist. Da an dieser Stelle keine Projekte neu angelegt oder gelöscht werden sollen, reichen bei *DBNavigator1* die Buttons zum Bewegen des Datenzeigers. Des weiteren wird eine *TDBEdit*-Komponente eingefügt, die den Projektnamen anzeigt.

Auf diese Tabelle wird eine Master-Detail-Verknüpfung mit *Query1* hergestellt. Die SQL-Anweisung wird beim Öffnen des Fensers zugewiesen.

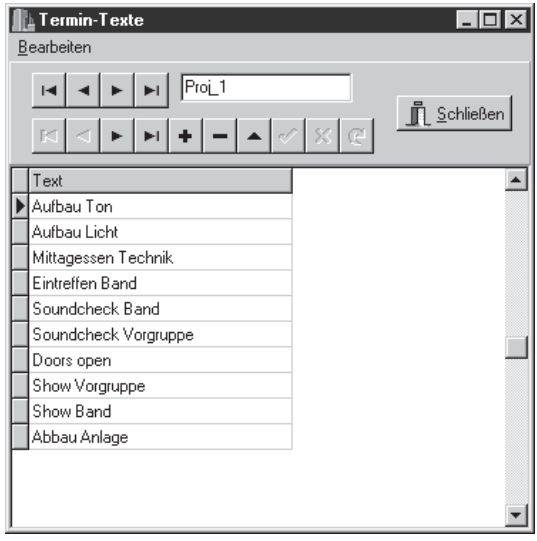


Bild 11.4: Das Fenster zum Bearbeiten der Tabelle Texte.DB

```
void __fastcall TChild::Termintexteanzeigen1Click(TObject *Sender)
{
  Form12->Show();
  Form12->Caption = "Termin-Texte";
  Form12->Query1->SQL->Clear();
  Form12->Query1->SQL->Add("SELECT * FROM texte");
  Form12->Query1->SQL->Add("WHERE projekt = :nummer");
  Form12->Query1->Open();
}
```

Hier muß kein Parameter übergeben werden, es reicht, die *Query1*-Eigenschaft *DataSource* auf *DataSource1* zu setzen. Nun müssen über *DataSource2* noch die Komponenten *DBNavigator2* und *DBGrid1* verbunden werden. Des weiteren muß dem neu eingefügten Datensatz die aktuelle Projektnummer zugewiesen werden.

```
void __fastcall TForm12::Query1AfterInsert(TDataSet *DataSet)
// Neuen Einträgen eine Projektnummer zuweisen
{
  Query1->FieldByName("Projekt")->AsInteger
    = Table1->FieldByName("Nummer")->AsInteger;
}
```

Zudem muß nach jedem neuen Eintrag die Abfrage aktualisiert werden, wozu sie geschlossen und wieder geöffnet werden muß. Außerdem wird der Datenzeiger auf den letzten Datensatz gesetzt – das erspart beim Eingeben das erneute Scrollen, wenn mehr Einträge vorhanden sind, als in der Tabelle dargestellt werden können. Diese Funktion wird auch mit dem *OnAfterDelete*-Ereignis verbunden.

```
void __fastcall TForm12::Query1AfterPost(TDataSet *DataSet)
// Anzeige aktualisieren
{
  Query1->Close();
  Query1->Open();
  Query1->Last();
}
```

11.2.1 Kopieren, Einfügen und Löschen von Einträgen

Häufig werden dieselben Einträge in mehreren Projekten benötigt. Damit diese nicht laufend neu eingegeben werden müssen, wollen wir Prozeduren schreiben, die diese Einträge von einem Projekt zum anderen kopieren können.

Mit dem Menübefehl *Kopieren* werden alle Einträge in die String-Liste *Kopie* kopiert, dabei wird in *DBGrid1* das Verschieben des Datenzeigers angezeigt. Bei großen Tabellen – hier im Beispiel gilt das weniger – würde dies den Programmablauf deutlich bremsen. Deshalb würde man dann die Eigenschaft *Enabled* der entsprechenden *TDataSource*-Komponente vor dem Kopieren auf *false* und hinterher auf *true* setzen.

```
void __fastcall TForm12::Copy1Click(TObject *Sender)
// Einträge kopieren
{
  Query1->First();
  while(!Query1->Eof)
  {
    Kopie.Add(Query1->FieldByName("Text")->AsString);
    Query1->Next();
  }
}
```

Um die Einträge einzufügen, werden sie einfach von der String-Liste wieder nach *Query1* kopiert.

```
void __fastcall TForm12::Paste1Click(TObject *Sender)
// Kopie einfügen
{
```

```
  int i;
  for(i=0; i<Kopie.Count; i++)
  {
    Query1->Append();
    Query1->FieldByName("Text")->AsString = Kopie.Strings[i];
    Query1->Post();
  }
}
```

Bisweilen mag auch der Wunsch aufkommen, die Einträge allesamt zu löschen. Mit dem *DBNavigator* ist das ein wenig umständlich, deshalb erstellen wir auch dafür auch eine Prozedur. Zunächst wird eine Sicherheitsabfrage durchgeführt, danach werden alle Einträge gelöscht.

```
void __fastcall TForm12::Entfernen1Click(TObject *Sender)
// Alle Einträge löschen
{
  if(Application->MessageBox
    ("Alle Einträge löschen?", "Bestätigung", 0) == 1)
  {
    Query1->First();
    while(Query1->Eof != true)
    {
      Query1->Delete();
      Query1->First();
    }
  } // if(Application->MessageBox
} // TForm12::Entfernen1Click
```

11.3 Editieren der Adressenliste

Die *TabbedNotebook*-Seite zum Editieren der Adressenliste gleicht in weiten Teilen der Seite zum Editieren der Terminliste: Auch hier müssen eine *TQuery*- und eine *TTable*-Komponente zusammenarbeiten. Die dafür erforderlichen Anweisungen wurden bereits in Kapitel 11.1 besprochen und sollen hier nicht mehr wiederholt werden. Außerdem ist der Quelltext des Projektes auf der beiliegenden CD-ROM zu finden und kann bei Bedarf zu Rate gezogen werden.

Mit einem Mausklick auf *Label5* (*Adresse*) wird die *Notebook*-Seite nach Bild 11.6 gezeigt, die das Auswählen einer Adresse erlaubt. Hier gibt es zwei kleine Unterschiede zwischen den Eingabehilfen für Datum und Uhrzeit auf der einen und der Adresse auf der anderen Seite: Zum einen können Datum und Uhrzeit auch

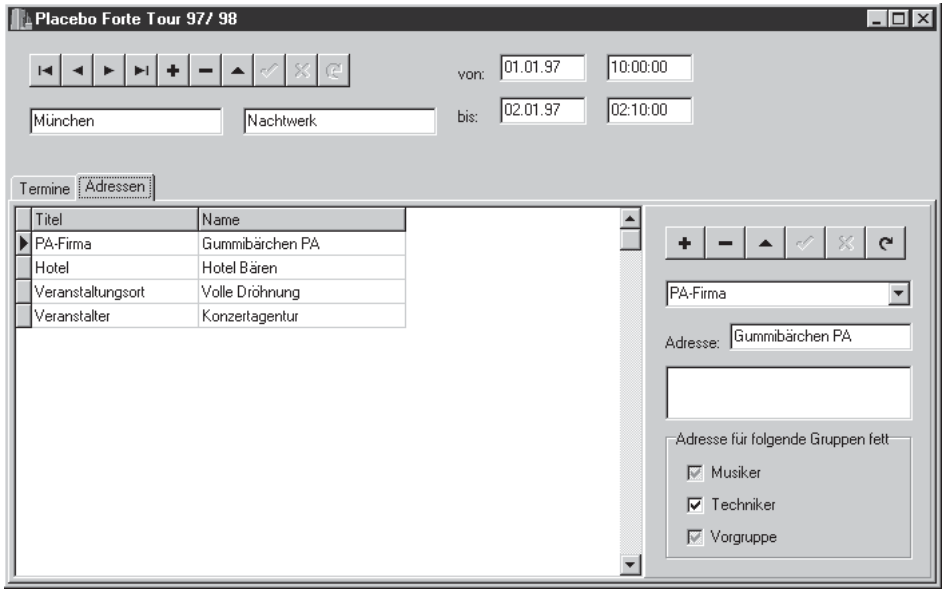


Bild 11.5: Editieren der Adressenliste

mit der Tastatur eingegeben werden, während die Adresse nur aus den gespeicherten Adressen ausgewählt werden kann.

Zum anderen können die *Notebook*-Seiten für das Datum und die Adresse nur aufgerufen werden, wenn sich die Datenmenge entweder im Status *dsInsert* oder im Status *dsEdit* befindet, wohingegen die *Notebook*-Seite für die Adresse immer aufgerufen werden kann. Adressen können allerdings nur eingefügt werden, wenn sich die Datenmenge in einem der beiden Stati befindet. Welche von beiden Lösungen die zweckmäßigere ist, halte ich für eine Frage des Geschmacks und habe Ihnen deshalb beide Möglichkeiten vorgestellt.

Damit man erkennen kann, ob mit dem Mausklick auf einen der beiden Bit-Buttons nun eine Adresse eingefügt wird oder ob lediglich die Seite von *Notebook2* gewechselt wird, wird die Beschriftung beziehungsweise die Eigenschaft *Enabled* der Buttons geändert.

```
void __fastcall TChild::Label5Click(TObject *Sender)
{
  Notebook2->ActivePage = "Adressen";
  if((Table2->State == dsEdit) +  (Table2->State == dsInsert))
  {
    BitBtn1->Caption = "Adresse übernehmen";
    BitBtn2->Enabled = true;
  }
```

```
  else
  {
    BitBtn1->Caption = "Schließen";
    BitBtn2->Enabled = false;
  } // else((Table2->State == dsEdit) +  (Table...
} // TChild::Label5Click
```

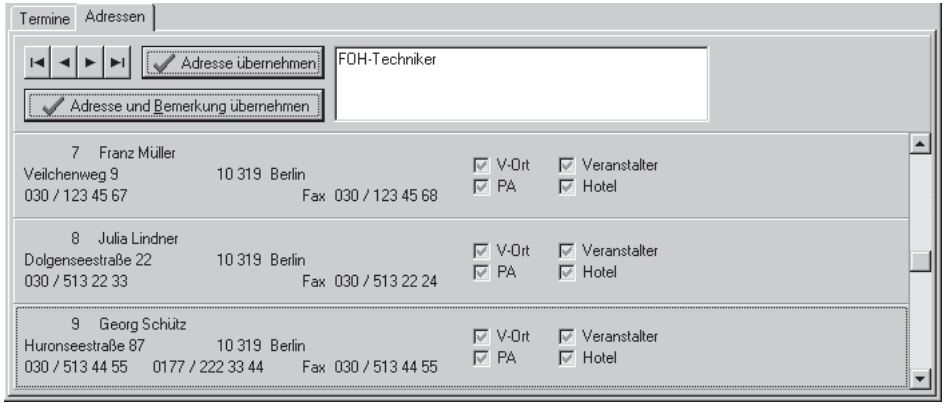


Bild 11.6: Auswählen einer Adresse

Da in der Tabelle *Orte.DB* nur die Adressen-Nummern gespeichert werden, ist auch die Prozedur *TChild::BitBtn1Click* recht übersichtlich. Die Adresse soll nicht nur mit einem Mausklick auf *BitBtn1*, sondern auch mit einem Doppelklick auf *DBGrid1* eingefügt werden können. Die folgende Prozedur wird deshalb auch mit dem Ereignis *OnDblClick* von *DBGrid1* verbunden.

```
void __fastcall TChild::BitBtn1Click(TObject *Sender)
// Adresse einfügen
{
  Notebook2->ActivePage = "Orte";
  if((Table2->State == dsEdit) +  (Table2->State == dsInsert))
  {
    Table2->FieldByName("Adresse")->AsInteger
      = Query6->FieldByName("Nummer")->AsInteger;
//  Table2->FieldByName("Bemerkung")->Assign
//    (Query6->FieldByName("Bemerkung"));
    Edit7->Text = Query6->FieldByName("Namen_1")->AsString;
  } //  if((Table2->State = dsEdit) +  (Tabl...
} // TChild::BitBtn2Click
```

In der Tabelle *Orte.DB* gibt es das Feld *Bemerkung*. Der Inhalt dieses Feldes kann entweder in *DBMemo1* eingegeben oder aus der Tabelle *Adresse.DB* übernommen werden. Dazu dient die Prozedur *TChild::BitBtn2Click*, welche der Prozedur *TChild::BitBtn1Click* gleicht, lediglich die als Kommentar dargestellte Anweisung kommt hier hinzu. Beachten Sie, daß Variablen vom Typ *TField* Objekte sind und die Zuweisung deshalb mit *Assign* erfolgen muß.

11.3.1 Suchen und Filtern

Sobald einige Projekte erstellt worden sind, wird der Adressenbestand auf eine Größe angewachsen sein, bei der die Übersichtlichkeit stark nachläßt. Deshalb sollen einige Menüpunkte eingefügt werden, die das Finden der Adressen erleichtern.

```
void __fastcall TChild::Veranstalter1Click(TObject *Sender)
{
  Query6->SQL->Clear();
  Query6->SQL->Add("SELECT * FROM adresse");
  Query6->SQL->Add("  WHERE z_v = 'true'");
  Query6->Open();
}
```

Diese Anweisungen beschränken die Anzeige auf alle Veranstalter, also auf alle Adressen, deren Spalte *z_v* den Wert *true* hat. Ähnlich aufgebaut sind die Prozeduren, die nach Veranstaltungsstätten, Hotels und PA-Firmen filtern.

```
void __fastcall TChild::Namen1Click(TObject *Sender)
{
  AnsiString s;
  if(InputQuery("Adresse suchen", "Nach folgendem Namen suchen", s))
  {
    Query6->SQL->Clear();
    Query6->SQL->Add("SELECT * FROM adresse");
    Query6->SQL->Add("  WHERE (namen_1 LIKE '%" + s + "%')");
    Query6->SQL->Add("    OR (namen_2 LIKE '%" + s + "%')");
    Query6->Open();
  }
} // TChild::Namen1Click
```

Die Funktion *InputQuery* öffnet ein kleines Dialogfenster, in das ein String eingegeben werden kann. Dieser String wird als Ergebnis der Funktion zurückgegeben. Weitere Prozeduren zum Suchen von Orten und Postleitzahlen sind ähnlich aufgebaut, so daß sie hier nicht aufgeführt werden sollen.

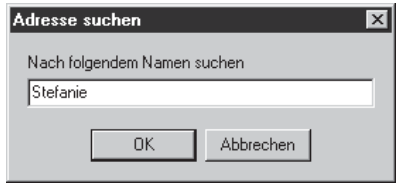


Bild 11.7: Eine InputBox

11.4 Kopieren

Häufig wird es vorkommen, daß die Termin- oder Adressenliste von mehreren Tagen gleich oder zumindest ähnlich aufgebaut ist. Hier kann man dem Anwender die Arbeit erleichtern, indem man Funktionen bereitstellt, die das Kopieren dieser Informationen erlauben.

Für gewöhnlich arbeiten die Funktionen zum Kopieren und Einfügen mit der Windows-Zwischenablage. Dies wäre auch hier möglich, aber nicht gerade einfach zu programmieren. Außerdem hätte die Verwendung der Windows-Zwischenablage lediglich den Vorteil, daß auch andere Programme auf diese Daten zugreifen können – dies jedoch dürfte hier wohl nicht benötigt werden. Wird dagegen beim Kopieren die Windows-Zwischenablage nicht verwendet, dann bleiben die kopierten Listen »im Speicher«, auch wenn zwischenzeitlich andere Programme auf die Zwischenablage zugegriffen haben sollten.

Zum Kopieren der Daten gibt es hier prinzipiell zwei Möglichkeiten: Zum einen können die Daten in geeignete Variablen – in diesem Fall String-Listen – geschrieben werden, zum anderen könnte man lediglich die Datensatznummer speichern, und beim Einfügen die Daten direkt aus der Tabelle transferieren. Wir wollen hier beide Möglichkeiten einsetzen.

```
void __fastcall TChild::Kopieren1Click(TObject *Sender)
// Termine oder Orte kopieren
{
  if(TabbedNotebook1->ActivePage == "Termine")
  {
    k_Tag.Clear();
    k_Zeit.Clear();
    k_Titel.Clear();
    k_A.Clear();
    k_B.Clear();
    k_C.Clear();
    Query2->First();
```

```
    while(!Query2->Eof)
    {
      k_Tag.Add(Query2->FieldByName("Tag")->AsString);
      k_Zeit.Add(Query2->FieldByName("Zeit")->AsString);
      k_Titel.Add(Query2->FieldByName("Titel")->AsString);
      k_A.Add(Query2->FieldByName("A")->AsString);
      k_B.Add(Query2->FieldByName("B")->AsString);
      k_C.Add(Query2->FieldByName("C")->AsString);
      Query2->Next();
    }
  } // if(TabbedNotebook1->ActivePage == "Termine")
  else
  {
    k_Nummer = Query4->ParamByName("Nummer")->AsInteger;
  } // if(TabbedNotebook1->ActivePage == "Termine")
} // TChild::Kopieren1Click
```

Bei den Terminen sollen die Daten in Stringlisten gespeichert werden. Beim Kopieren werden nun lediglich die betreffenden Spalten der Tabelle *Termine.DB* in die String-Listen kopiert. Bei den booleschen Feldern wird die *TField*-Eigenschaft *AsString* verwendet, welche die booleschen Felder in Strings konvertiert, so daß sie problemlos in String-Listen gespeichert werden können. Bevor der Tabelleninhalt in die String-Listen geschrieben wird, müssen jedoch die alten Einträge entfernt werden.

Wesentlich einfacher ist das Kopieren der Adressen-Liste: Hier muß lediglich die Datensatznummer in die Variable *k_Nummer* geschrieben werden.

Einfügen

Wir wollen nun die Prozedur erstellen, welche die Daten wieder in die entsprechenden Tabellen einfügt.

```
void __fastcall TChild::Einfgen1Click(TObject *Sender)
// Termine oder Orte einfügen
{
  if(TabbedNotebook1->ActivePage == "Termine")
  {
    int i;
    for(i=0; i<k_Tag.Count; i++)
    {
      Table1->Append();
      Table1->FieldByName("Tag")->AsString = k_Tag.Strings[i];
      Table1->FieldByName("Zeit")->AsString = k_Zeit.Strings[i];
```

```
      Table1->FieldByName("Titel")->AsString = k_Titel.Strings[i];
      Table1->FieldByName("A")->AsString = k_A.Strings[i];
      Table1->FieldByName("B")->AsString = k_B.Strings[i];
      Table1->FieldByName("C")->AsString = k_C.Strings[i];
      Table1->Post();
    }
  } // if(TabbedNotebook1->ActivePage == "Termine")
  else
  {
    Query7->Close();
    Query7->ParamByName("Nummer")->AsInteger = k_Nummer;
    Query7->Open();
    Query7->First();
    while(!Query7->Eof)
    {
      Table2->Append();
      Table2->FieldByName("Titel")->AsString
        = Query7->FieldByName("Titel")->AsString;
      Table2->FieldByName("Adresse")->AsString
        = Query7->FieldByName("Adresse")->AsString;
      Table2->FieldByName("Bemerkung")->AsString
        = Query7->FieldByName("Bemerkung")->AsString;
      Table2->FieldByName("A")->AsString
        = Query7->FieldByName("A")->AsString;
      Table2->FieldByName("B")->AsString
        = Query7->FieldByName("B")->AsString;
      Table2->FieldByName("C")->AsString
        = Query7->FieldByName("C")->AsString;
      Table2->Post();
      Query7->Next();
    } // while(!Query7->Eof)
  } // else (TabbedNotebook1->ActivePage == "Termine")
} // TChild::Einfgen1Click
```

Um die Daten in die Termin-Liste einzufügen, wird einfach der Inhalt der Stringlisten zurück in die Tabelle *Termine.DB* geschrieben. Etwas aufwendiger gestaltet sich dies bei der Adressen-Liste. Dazu muß zunächst eine *TQuery*-Komponente eingefügt werden, deren SQL-Anweisung wie folgt lautet:

```
SELECT * FROM orte
  WHERE show = :nummer
```

Dem Parameter *Nummer* wird mit dem Inhalt der Variablen *k_Nummer* belegt, also mit der Nummer derjenigen Termin-Liste, welche kopiert werden soll. Nun werden nacheinander alle Datensätze ausgelesen und nach *Table2* geschrieben.

Übernehmen

Bei einer Tour werden sich an vielen Tagen die Termine gleichen, so daß man sich mit der Kopier-Funktion viel Arbeit ersparen kann. Leider wird man in eigentlich allen Fällen das Datum manuell ändern müssen – und dies gleich bei allen Zeilen der Termin-Liste. Deshalb soll der Menüpunkt *Übernehmen* implementiert werden; die dazugehörige Prozedur ändert beim Einfügen der Daten gleich das Datum nach Vorgabe ab.

```
void __fastcall TChild::bernehmen1Click(TObject *Sender)
// Einfügen und Datum anpassen
{
  if(TabbedNotebook1->ActivePage == "Termine")
  {
    AnsiString dvs, s;
    TDateTime dv, dn, v;
    dvs = k_Tag.Strings[0];
    dv = StrToDate(dvs);
    if(InputQuery("Datumsverschiebung",
      "Beginnen mit folgendem Datum", dvs) == true)
    {
      dn = StrToDate(dvs);
      v = dn - dv;
      int i;
      for(i=0; i<k_Tag.Count; i++)
      {
        Table1->Append();
        dv = StrToDate(k_Tag.Strings[i]);
        dn = dv + v;
        Table1->FieldByName("Tag")->AsString = DateToStr(dn);
        Table1->FieldByName("Zeit")->AsString = k_Zeit.Strings[i];
        Table1->FieldByName("Titel")->AsString = k_Titel.Strings[i];
        Table1->FieldByName("A")->AsString = k_A.Strings[i];
          ...
        Table1->Post();
      } // for(i=0; i<k_Tag.Count; i++)
    } // if(InputBox("Datumsverschiebung",
  } // if(TabbedNotebook1->ActivePage == "Termine")
} // TChild::bernehmen1Click
```

Mit einer *InputQuery* wird abgefragt, in welches Datum die eingefügten Termine geändert werden sollen. Nun werden sich in vielen Fällen die Termine über Mitternacht hinweg erstrecken, so daß diese entsprechend anders abgeändert werden müssen. Das mit der *InputQuery* abgefragte Datum ist demnach das Datum, auf welches der erste Termin der Liste geändert werden soll – alle anderen Termine werden relativ dazu geändert.

Zunächst wird die Variable *v* berechnet, welche angibt, um welchen Wert jedes Datum geändert werden soll. Dazu werden die Strings aus der *InputQuery* und aus dem ersten Eintrag der Termin-Liste in Datums-Werte gewandelt, daraus wird dann die Differenz gebildet. Beim Einfügen der Daten wird bei jedem Datums-Feld das dort gespeicherte Datum in einen Datums-Wert gewandelt, zur Variable *v* addiert, zurück in einen String gewandelt und in die Tabelle eingefügt.

Da die Umrechnung erst beim Einfügen erfolgt, kann ein und dieselbe Termin-Liste schnell und unkompliziert auf alle Tage kopiert werden.

Löschen

Insbesondere dann, wenn man bei Einfügen oder Übernehmen einen Fehler gemacht hat, ist man über eine Funktion erfreut, welche den ganzen Unsinn wieder löscht. Dazu werden drei Menüpunkte implementiert, von denen der eine alle Termine, der andere alle Adressen löscht. Der dritte Menüpunkt, auf dessen Besprechung wir uns hier beschränken wollen, löscht alle Termine und Adressen.

```
void __fastcall TChild::TermineundAdressen1Click(TObject *Sender)
// Alle Termine und Adressen eines Tages löschen
{
  if(Application->MessageBox("Alle Termine und Adressen löschen?",
    "Sicherheitsabfrage", 0) == 1)
  {
    Screen->Cursor = crHourGlass ;
    Query4->Last();
    while(Query4->RecordCount > 0)
    {
      Table2->Delete();
      Query4->Close();
      Query4->Open();
    }
    Query2->Last();
    while(Query2->RecordCount > 0)
    {
      Table1->Delete();
      Query2->Close();
```

```
      Query2->Open();
    }
    Screen->Cursor = crDefault;
  } // if(Applikation->MessageBox("Alle Termine...
} // TChild::TermineundAdressen1Click
```

Nach einer Sicherheitsabfrage wird ein Datensatz nach dem anderen aus den Tabellen *Termine.DB* beziehungsweise *Adresse.DB* gelöscht. Dazu wird der Datenzeiger immer auf den letzten Datensatz von *Query4/Query2* gesetzt. Der gleiche Datensatz wird automatisch in *Table1/Table2* ausgewählt und dann gelöscht. Damit *Query4/Query2* immer aktualisiert wird, muß sie geschlossen und anschließend wieder geöffnet werden. Dies beschleunigt nicht gerade den Programmablauf, bei einer überschaubaren Anzahl von Terminen ist dies aber durchaus akzeptabel.

11.5 Drucken

Wie schon in Kapitel 9 erklärt wurde, soll das Drucken der Daten mit dem Objekt *Printer* geschehen. Das Drucken wird mit dem Menüpunkt *Drucken* gestartet, der sich im Menü des MDI-Rahmenformulars und nicht – wie bei MDI-Anwendungen oft der Fall – im Menü des Kindfensters befindet. Diese Vorgehensweise ist in diesem Fall auch die zweckmäßigere: Wie Bild 11.8 zeigt, weicht der benötigte Drucken-Dialog stark von C++Builders *TPrintDialog* ab, weshalb hier ein eigenes Formular entwickelt werden muß. Dieses Formular wird beim Mausklick auf den Menüpunkt *Drucken* aufgerufen – und zwar unabhängig davon, welches Kindfenster gerade aktiv ist; deshalb kann hier dieser Menüpunkt in das Menü des MDI-Rahmenformulars eingefügt werden.

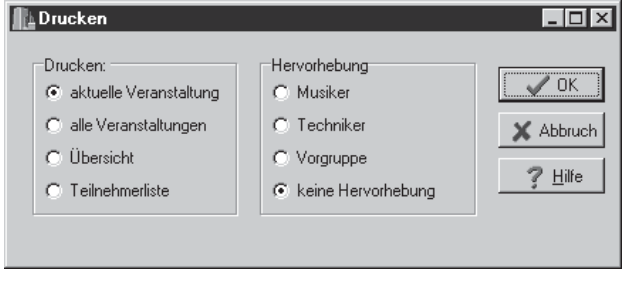


Bild 11.8: Der Dialog zum Drucken des Tourplaners

```
void __fastcall TForm1::Drucken1Click(TObject *Sender)
{
  if(MDIChildCount > 0)
    Form13->ShowModal();
  else
    ShowMessage("Kein Projekt geöffnet");
}
```

Es ist wohl logisch, daß nur dann ein Projekt gedruckt werden kann, wenn mindestens ein Projekt geöffnet ist. Ist dies nicht der Fall, dann wird eine Fehlermeldung ausgegeben.

```
void __fastcall TForm13::FormActivate(TObject *Sender)
{
  BitBtn1->SetFocus();
  AnsiString s;
  TChild* Child = dynamic_cast<TChild*>(Form1->ActiveMDIChild);
  s = Child->DBCheckBox1->Caption;
  if(s != "")
    RadioGroup2->Items->Strings[0] = s;
  else
    RadioGroup2->Items->Strings[0] = "A";
  s = Child->DBCheckBox2->Caption;
  if(s != "")
    RadioGroup2->Items->Strings[1] = s;
  else
    RadioGroup2->Items->Strings[1] = "B";
  s = Child->DBCheckBox3->Caption;
  if(s != "")
    RadioGroup2->Items->Strings[2] = s;
  else
    RadioGroup2->Items->Strings[2] = "C";
} // TForm13::FormActivate
```

Beim Aufrufen des Fensters sollen diejenigen Radio-Buttons, welche die Hervorhebung der Einträge steuern, gemäß der Vorgabe beschriftet werden. Die betreffenden Strings befinden sich in der Tabelle *Projekt.DB*. Sind keine Kategorie-Bezeichnungen definiert, dann sollen die Radio-Buttons mit *A*, *B* und *C* beschriftet werden.

```
void __fastcall TForm13::BitBtn1Click(TObject *Sender)
{
  TChild* Child = dynamic_cast<TChild*>(Form1->ActiveMDIChild);
  Child->Drucken();
}
```

Beim Betätigen des *OK*-Buttons wird die Methode *Drucken* des Kindfensters aufgerufen und das Dialog-Formular geschlossen. Bei der *OnClick*-Prozedur des *Abbruch*-Buttons wird nur die Methode *Close* aufgerufen.

```
void __fastcall TForm13::FormKeyDown(TObject *Sender, WORD &Key,
  TShiftState Shift)
{
  if(Key == 27)
    Close();
}
```

Noch ein kleines Feature am Rande: Da das Programm auch mit der Tastatur zu bedienen sein muß, soll man das Formular auch mit der *ESC*-Taste verlassen können. Die Prozedur *TForm13::FormKeyDown* muß dazu mit den *OnKeyDown*-Ereignissen aller Komponenten in diesem Formular verknüpft werden.

Die Methoden des Kindfensters zum Drucken

Der *Drucken*-Dialog (*Form13*) ruft die Methode *Drucken* des gerade aktiven Kindfensters auf.

```
void __fastcall TChild::Drucken()
{
  if(Form13->RadioGroup1->ItemIndex == 0)
    // aktuelle Veranstaltung drucken
    ShowDrucken();

  if(Form13->RadioGroup1->ItemIndex == 1)
    // alle Veranstaltungen drucken
  {
    Query1->First();
    while(!Query1->Eof)
    {
      ShowDrucken();
      Query1->Next();
    }
  } // if(Form15->RadioGroup1->ItemIndex == 1)
} // TChild::Drucken
```

Soll nur die aktuelle Seite des Tourplaners ausgedruckt werden, dann wird lediglich die Prozedur *ShowDrucken* aufgerufen. Sollen alle Seiten des Tourplaners gedruckt werden, dann werden in *Query1* alle Seiten aufgerufen und dann einzeln mit *ShowDrucken* ausgedruckt.

Anweisungen zum Drucken von Daten sind in der Regel lang und nicht besonders interessant. Deshalb wurden hier die Prozedurteile zum Ausdrucken der Übersicht und der Teilnehmerliste nicht wiedergegeben – die dazu erforderlichen Anweisungen werden ohnehin alle in der Prozedur *ShowDrucken* besprochen.

```
void __fastcall TChild::ShowDrucken()
{
  double ff;
  AnsiString r, s, sv, t;
  int i, f;
  f = Form13->RadioGroup2->ItemIndex;
  s = Query1->FieldByName("Titel_1")->AsString;
  Printer()->Orientation = poPortrait;
  Printer()->Title = "Tourplaner - Show " + s;
  Printer()->BeginDoc();
  Printer()->Canvas->MoveTo(1,1);
  SetMapMode(Handle, MM_LOMETRIC);
  Printer()->Canvas->Font->Name = "Arial";
  Printer()->Canvas->Font->Size = 50;
  ff = 949 / Printer()->Canvas->TextWidth("TestString");
```

Im ersten Teil werden zunächst alle Vorarbeiten erledigt: Die Variable *f* beinhaltet, ob eine, und wenn ja, welche Kategorie von Terminen und Adressen hervorgehoben werden soll. Da auf diese Information häufig zugegriffen werden muß, empfiehlt sich die Verwendung einer lokalen Variablen.

Die Seite soll hochkant bedruckt werden. Der String, welcher *Printer->Title* zugewiesen wird, erscheint als Titel im Druckmanager (und auch im Display mancher Drucker). Mit *Printer->BeginDoc* wird der Ausdruck gestartet. Damit der Ausdruck auch bei der Verwendung anderer Drucker gleich (oder zumindest ähnlich) aussieht, soll die Maßeinheit auf zehntel Millimeter gestellt werden. Damit das Handle von *Printer->Canvas* vorliegt, muß vorher irgendeine *Canvas*-Methode aufgerufen werden, in diesem Fall *MoveTo*.

Wie bereits in Kapitel 7 erwähnt, kann ein und dieselbe Schriftgröße bei unterschiedlichen Druckern zu höchst unterschiedlichen Ergebnissen führen. Deshalb wird die Variable *ff* (*FontFaktor*) verwendet, mit welcher später alle Schriftgrößen-Angaben multipliziert werden. Näheres dazu in Kapitel 7.

```
// Projektkopf drucken ============================================
  Printer()->Canvas->Font->Size = int(50 * ff);
  TFontStyles Style;
  Style << fsBold;
  Printer()->Canvas->Font->Style = Style;
```

```
while(Printer()->Canvas->TextWidth(s) > 1550)
  Printer()->Canvas->Font->Size = Printer()->Canvas->Font->Size - 1;
Printer()->Canvas->TextOut(300, 150, s);
s = Query1->FieldByName("Titel_2")->AsString;
Printer()->Canvas->Font->Size = int(20 * ff);
Style >> fsBold;
Printer()->Canvas->Font->Style = Style;
while(Printer()->Canvas->TextWidth(s) > 1000)
  Printer()->Canvas->Font->Size = Printer()->Canvas->Font->Size - 1;
Printer()->Canvas->TextOut(300, 400, s);
Printer()->Canvas->Font->Size = int(10 * ff);
Style << fsItalic;
Printer()->Canvas->Font->Style = Style;
s = Edit1->Text + " (" + Edit2->Text + ")";
Printer()->Canvas->TextOut(1400, 400, "von " + s);
s = Edit3->Text + " (" + Edit4->Text + ")";
Printer()->Canvas->TextOut(1400, 460, "bis " + s);
Printer()->Canvas->TextOut(1000, 10, "Tour: " + Caption);
switch(f)
{
  case 0:
    s = DBCheckBox1->Caption;
  break;
  case 1:
    s = DBCheckBox2->Caption;
  break;
  case 2:
    s = DBCheckBox3->Caption;
  break;
  default:
    s = "";
}
Printer()->Canvas->TextOut(1700, 10, s);
Printer()->Canvas->Pen->Width = 20;
Printer()->Canvas->MoveTo(300, 550);
Printer()->Canvas->LineTo(1850, 550);
```

Zuerst werden die Felder *Titel_1* und *Titel_2* der Tabelle *Shows.DB* gedruckt. Diese Felder können bis zu 30 Zeichen lang sein. Würde man eine Schriftgröße wählen, mit welcher auch lange Titel noch vollständig in die Zeile passen, dann wäre diese Schriftgröße in den meisten Fällen unnötig klein. Deshalb wählt man eine

angemessene Schriftgröße, prüft, ob die Titel noch in die vorgesehenen Stellen passen, und verkleinert andernfalls schrittweise die Schriftgröße.

Des weiteren werden Angaben darüber gedruckt, von wann bis wann auf dieser Seite Termine zu finden sind. In der Kopfzeile findet man nicht nur den Titel des Projekts, sondern auch die Kategorie, für die auf dieser Seite die Hervorhebungen gesetzt worden sind – der entsprechende String wird in der *Case*-Verzweigung zugewiesen. Zum Schluß wird mit *LineTo* eine horizontale Linie gezeichnet.

```
// Termine drucken ================================================
  Query2->First();
  i = 0;
  Printer()->Canvas->Font->Size = int(10 * ff);
  Style >> fsItalic;
  Style << fsBold;
  Printer()->Canvas->Font->Style = Style;
  Printer()->Canvas->TextOut(300, 600, "Tag");
  Printer()->Canvas->TextOut(500, 600, "Zeit");
  Printer()->Canvas->TextOut(800, 600, "Titel");
  while(!Query2->Eof)
  {
    Style >> fsBold;
    switch(f)
    {
      case 0:
      {
        if(Query2->FieldByName("A")->AsBoolean == true)
          Style << fsBold;
      }
      case 1:
      {
        if(Query2->FieldByName("B")->AsBoolean == true)
          Style << fsBold;
      }
      case 2:
      {
        if(Query2->FieldByName("C")->AsBoolean == true)
          Style << fsBold;
      }
    } // switch(f)
    Printer()->Canvas->Font->Style = Style;
    r = Query2->FieldByName("Tag")->AsString;
    s = Query2->FieldByName("Zeit")->AsString;
```

```
    t = Query2->FieldByName("Titel")->AsString;
    if(r != sv)
      Printer()->Canvas->TextOut(300, (i * 60) + 670, r);
    sv = r;
    Printer()->Canvas->TextOut(500, (i * 60) + 670, s);
    if(Printer()->Canvas->TextWidth(t) > 450)
    {
      i++;
      Printer()->Canvas->TextOut(450, (i * 60) + 670, t);
    }
    else
      Printer()->Canvas->TextOut(700, (i * 60) + 670, t);
    Query2->Next();
    i++;
  } // while (!Query2->EOF)
```

Zunächst werden die Titel der Termintabelle gedruckt, danach werden alle zu dieser Show gehörenden Datensätze der Tabelle *Termine.DB* ausgedruckt. Mit Hilfe der *Case*-Verzweigung wird festgestellt, ob der jeweilige Datensatz für die Kategorie, welche gerade hervorgehoben gedruckt werden soll, ausgewählt worden ist. In diesem Fall wird *fsBold* in die Eigenschaft *Printer->Canvas->Font->Style* aufgenommen.

In der Termintabelle soll nicht bei jedem Eintrag ein neues Datum gedruckt werden, sondern nur am Anfang der Tabelle und nur dann, wenn sich das Datum ändert. Deshalb wird der Datums-String mit dem String verglichen, welcher beim letzten Datensatz gedruckt worden ist. Nur wenn die beiden Strings voneinander abweichen, wird der String und somit das Datum mit *TextOut* ausgegeben.

Die Länge der Terminbezeichnung kann den dafür vorgesehenen Platz überschreiten – in diesem Fall wird eine neue Zeile begonnen und der Termin dort ausgegeben. Damit die Termine untereinander ausgegeben werden, muß nach jedem Datensatz die Variable *i* um den Wert eins erhöht werden.

```
// Orte drucken ========================================================
  Query4->First();
  i = 0;
  Printer()->Canvas->Font->Size = int(8 * ff);
  while(!Query4->Eof)
  {
    Style << fsBold;
    Printer()->Canvas->Font->Style = Style;
    s = Query4->FieldByName("Titel")->AsString;
    Printer()->Canvas->TextOut(1270, (i * 300) + 600, s);
```

```
    Style >> fsBold;
    switch(f)
    {
      case 0:
      {
        if(Query4->FieldByName("A")->AsBoolean == true)
          Style << fsBold;
      }
      case 1:
      {
        if(Query4->FieldByName("B")->AsBoolean == true)
          Style << fsBold;
      }
      case 2:
      {
        if(Query4->FieldByName("C")->AsBoolean == true)
          Style << fsBold;
      }
    } // switch(f)
    Printer()->Canvas->Font->Style = Style;
    s = Query4->FieldByName("Namen_1")->AsString;
    t = Query4->FieldByName("Namen_2")->AsString;
    Printer()->Canvas->TextOut(1270, (i * 300) + 640, s + ",  " + t);
      ...
    s = DBMemo1->Lines->Strings[0];
    if(DBMemo1->Lines->Count > 1)
      s = s + DBMemo1->Lines->Strings[1];
    Printer()->Canvas->TextOut(1270, (i * 300) + 840, s);
    Query4->Next();
    i++;
  } // while(!Query4.EOF)
  Printer()->Canvas->MoveTo(1200, 550);
  Printer()->Canvas->LineTo(1200, ((i + 1) * 300) + 880);
  Printer()->EndDoc();
} // TChild::ShowDrucken
```

Der Ausdruck der Adressenliste bietet dann nicht viel Neues. Aus diesem Grund sind auch viele Anweisungen zum Ausdruck der einzelnen Datenbankspalten hier nicht abgedruckt. Zum Schluß wird die vertikale Linie ausgegeben und das Drucken mit *Printer->EndDoc* abgeschlossen.

11.6 Die Teilnehmerliste

Mit dem Menüpunkt OPTIONEN | PROJEKTE BEARBEITEN läßt sich ein Fenster öffnen, in dem nicht nur – wie bereits besprochen – weiteren Benutzern Zugriff auf das Projekt gewährt werden kann, hier wird auch die Teilnehmerliste erstellt. Bild 11.9 zeigt die entsprechende *TabbedNotebook*-Seite.

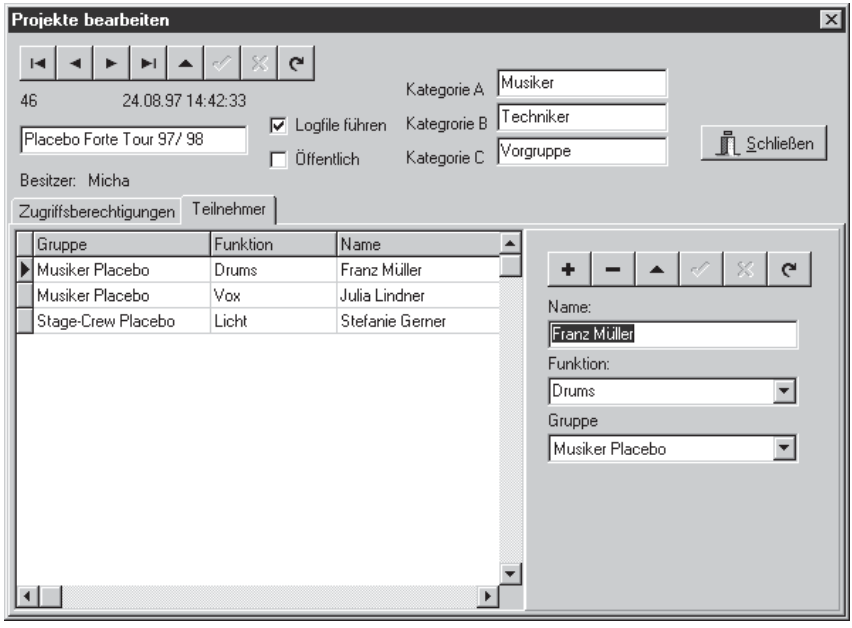


Bild 11.9: Die TabbedNotebook-Seite für die Teilnehmerliste

Wie man beim Anblick des *DBGrid*s und der Eingabemaske ahnt, wird auch hier die Arbeit zwischen einer *TQuery*- und einer *TTable*-Komponente aufgeteilt. Die dafür notwendigen Prozeduren wurden schon in diesem Kapitel besprochen und sollen deshalb hier nicht wiederholt werden. Auch die *Notebook*-Seite zum Auswählen der Adresse, die mit einem Mausklick auf *Label2* aufgerufen wird, ist nichts Neues. Deshalb wollen wir uns bei der Beschreibung dieser Funktion auf die zwei Details beschränken, die hier neu sind.

Wie ein Blick auf Bild 8.4 zeigt, können in die Spalte *Funktion* der Tabelle *Beteil.DB* Strings eingegeben werden. Deshalb können hier die Einträge von *DBComboBox1* direkt übernommen werden. Die Spalte *Gruppe* ist dagegen vom Typ *Integer*. Hier wird eine Referenz auf die Tabelle *Gruppen.DB* gebildet. Deshalb muß für den ausgewählten Eintrag von *ComboBox2* erst dessen Datensatznummer ermittelt werden. Die dafür nötigen Anweisungen sollen mit dem *TComboBox*-Ereignis *OnChange* verknüpft werden.

```
void __fastcall TForm10::ComboBox1Change(TObject *Sender)
{
  Query4->First();
  while((Query4->FieldByName("Text")->AsString != ComboBox1->Text)
       *(! Query4->Eof))
    Query4->Next();
  if(Table4->State == dsBrowse)
    Table4->Edit();
  if((Table4->State == dsEdit)+(Table4->State == dsInsert))
  Table4->FieldByName("Gruppe")->AsInteger
    = Query4->FieldByName("Nummer")->AsInteger;
}
```

Da hier die Komponente *TQuery* verwendet wird, kann man sich entweder eine entsprechende SELECT-Anweisung erstellen oder nach dem Eintrag inkremental suchen. Da die Gruppennamen an die Projekte gekoppelt sind, werden von *Query6* meist nur sehr wenige Gruppennamen ermittelt (in der Praxis kann man mit drei bis sechs Gruppennamen rechnen) – somit gibt es keine triftigen Gründe gegen eine inkrementale Suche.

Während des Programmierens von *while*- oder *repeat*-Schleifen muß man davon ausgehen, daß durch einen Fehler die Abbruchbedingung nie erreicht wird. Hier sollte man eine entsprechende Sicherung erstellen, welche dafür sorgt, daß das Programm nicht in einer Endlosschleife landet. Diesem Zweck dient der Teil *(! Query4->EOF)* in der *while*-Bedingung.

Nun zu einem weiteren Problem: Die Zuweisung der Gruppennummer an die Spalte *Gruppe* erfolgt nur, wenn sich der Einrag in der *ComboBox* ändert. Nun ist es aber möglich, daß die *ComboBox* schon den richtigen Eintrag anzeigt und vom Anwender nicht geändert wird. Für das *DBGrid* werden jedoch die Daten mit einer EQUI-JOIN-Anweisung ermittelt, und ohne eine Gruppennummer würde der Datensatz überhaupt nicht angezeigt.

```
SELECT b.nummer, g.namen AS gruppe, b.funktion,
    a.namen_1 AS namen
  FROM beteil b, adresse a, gruppen g
  WHERE (b.projekt = :nummer)
    AND (b.adresse = a.nummer)
    AND (b.gruppe = g.nummer)
  ORDER BY g.namen, b.funktion
```

Die Zuweisung muß also noch mit einem anderen Ereignis verknüpft werden, beispielsweise mit dem TTable-Ereignis *OnAfterInsert*. Anstatt alle Anweisungen zu übernehmen, wird hier lediglich die Prozedur *ComboBox1Change* aufgerufen.

```
void __fastcall TForm10::Table4AfterInsert(TDataSet *DataSet)
{
  Table4->FieldByName("Projekt")->AsInteger
    = Table1->FieldByName("Nummer")->AsInteger;
  Edit2->Text = "";
  ComboBox1Change(Table4);
}
```

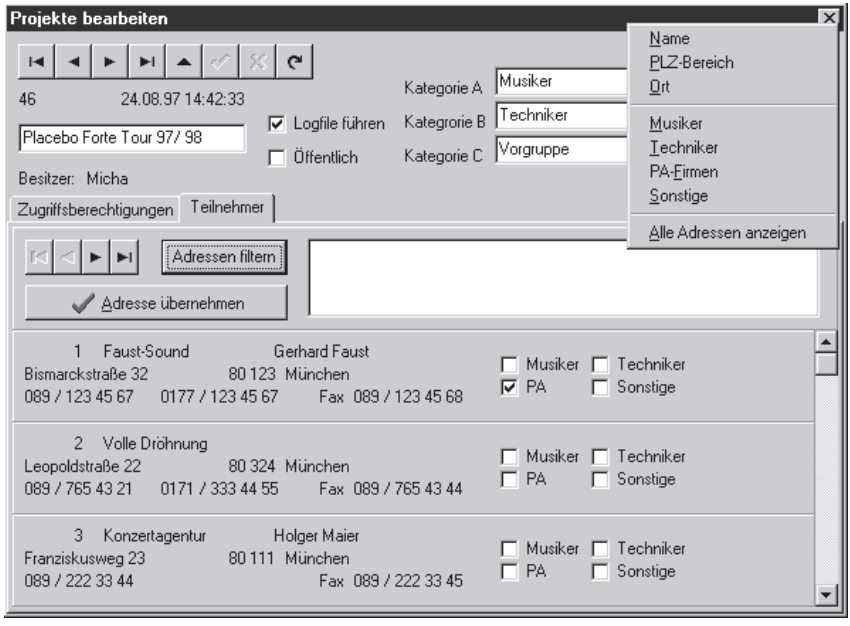


Bild 11.10: Adressenliste mit Popup-Menü

Das Popup-Menü

Auch auf der Adreß-Auswahl-Seite soll die Möglichkeit bestehen, den Adressen-Bestand zu filtern. Dafür soll nicht eigens ein Formular-Menü eingefügt werden. Wir wollen diesmal ein *Popup*-Menü verwenden. In Ihren Programmen sollten Sie solche Abwechslungen eher vermeiden, weil sie zur Verwirrung der Anwender führen. In diesem Beispielprogramm wollen wir uns jedoch die Chance nicht entgehen lassen, auch noch ein *Popup*-Menü zu verwenden.

Bild 11.10 zeigt, welche Menüpunkte im *Popup*-Menü zu finden sind. Die entsprechenden Prozeduren sind direkt aus dem Kindfenster übernommen – lediglich die Branchen wurden hier ein wenig modifiziert.

Der Eigenschaft *PopupMenu* des Formulars wird der Wert *PopupMenu1* zugewiesen, alle in diesem Formular vorhandenen Komponenten rufen mit der rechten Maustaste auch dieses Menü auf, wenn deren Eigenschaft *PopupMenu* nicht (oder nicht anders) gesetzt ist. Wenn nur ein einziges Formular der Anwendung ein *Popup*-Menü verwendet, dann wird ein solches vom Anwender nicht erwartet – die Online-Hilfe wird in der Regel eher lückenhaft gelesen. Deswegen sollte sich das *Popup*-Menü auch mit einem Button aufrufen lassen.

```
void __fastcall TForm10::Button1Click(TObject *Sender)
{
  PopupMenu1->Popup(Left + 440, Top + 15);
}
```

Der *TPopup*-Methode *Popup* werden als Parameter die Koordinaten übergeben, die bestimmen, an welcher Stelle das Popup-Fenster auf dem Bildschirm erscheinen soll. Da hier absolute Koordinaten verwendet werden, müssen die Koordinaten auf die absolute Position des Fensters bezogen werden, wenn das *Popup*-Menü stets an der gleichen Stelle im Fenster angezeigt werden soll.

12 Adressensuche

Die Suche nach Adressen ist nicht nur einer der interessantesten Aspekte der Datenbankprogrammierung, er ist auch einer der wichtigsten.

- Um Redundanzen zu vermeiden, möchte man jede Adresse möglichst nur einmal im Rechner haben. Hier sollte dann möglichst bei der Eingabe schon geprüft werden, ob die Adresse nicht vielleicht schon vorhanden ist.
- Oft ist es nötig, einen Datensatz anhand der Adresse zu suchen – die wenigsten Personen geben bei der Korrespondenz stets die Kunden- oder Mitgliedsnummer an.
- Manchmal sollen verschiedene Datenbestände abgeglichen werden.

Im Prinzip wäre die Suche nach einer Adresse nicht weiter problematisch, würden hier nicht unterschiedliche Schreibweisen ganz erhebliche Schwierigkeiten bereiten. Nehmen wir einmal an, eine neu eingegebene Adresse würde mit der folgenden SQL-Anweisung gesucht.

```
SELECT * FROM adressen
  WHERE (vorname = "Michael")
    AND (nachname = "Mustermann")
    AND (strasse = "Karl-Marx-Straße 9 Q II")
    AND (plz = "10967")
    AND (ort = "Berlin-Neukölln")
```

Diese Anweisung ist so formuliert, daß die Adresse exakt so in der Datenbank stehen müßte, um gefunden zu werden. Die Wahrscheinlichkeit dafür, daß sie exakt so in der Datenbank steht, schätze ich auf rund eins zu zehn. Sie ist stark davon abhängig, wie viele verschiedene Personen Adressen in die Datenbank eingeben und wie sorgfältig sie dabei vorgehen.

Was könnte hier nun alles danebengehen?

Vornamen

- Beim Eingeben der Adresse wurde der Name abgekürzt – in der Regel deshalb, weil diese Person ihn bei der Korrespondenz auch abgekürzt hat.
- Die Person nennt sich normalerweise *Micha*, hat aber einen Stempel mit *Michael*.

- Der vollständige Name lautet *Michael Ludwig*, und es werden folgende Varianten verwendet:
  - *Michael*
  - *Michael L.*
  - *M.L.*, *M.-L.*
  - *Michael-Ludwig*
  - *M. Ludwig*

  Man sollte nicht glauben, daß immer derselbe Vorname verwendet wird. Im Laufe der Zeit modifizieren viele Personen die Schreibweise ihres Namens. Wenn aus der einen Periode das Briefpapier, aus der anderen die Visitenkarte, aus der dritten die Adreß-Etiketten, aus der vierten der Stempel stammt, dann können auch bei kurz hintereinander eintreffenden Schreiben die Vornamen differieren.
- Manche Personen schreiben auch nicht besonders deutlich. Meinen Vornamen gibt es in den Datenbanken diverser Firmen auch in höchst unterschiedlichen Schreibweisen (*Micha, Michael, Michaela, Mirco, Mieko* und *Mischa* fallen mir spontan ein).

Nachnamen

Auch die Nachnamen sind leider nicht so eindeutig, wie das aus Sicht eines Datenbankbetreibers wünschenswert wäre.

- Personen heiraten und nehmen nach der Trennung wieder den alten Namen an.
- Die vielfältigen Möglichkeiten zur Verwendung eines Doppelnamens werden genutzt, natürlich unter Verwendung aller möglichen Schreibweisen.
- Manche EDV-Systeme beherrschen noch nicht die deutschen Umlaute – aus *Müller* wird so *Mueller*.
- Akademische Grade und Adelstitel werden mal mit dem Vornamen, mal mit dem Nachnamen, mal überhaupt nicht verwendet. Manche Namen (wie *Freiherr Marschall von Biberstein* – den Namen gibt's wirklich) sind so lang, daß sie in fast kein Formular passen. Hier wird je nach des zur Verfügung stehenden Platzes immer eine andere Kombination gewählt.
- Bei der mündlichen Übermittlung sind verschiedene Schreibweisen nicht zu unterscheiden – beispielsweise, ob es *Mayer, Meyer, Maier, Meier* oder *Meyr* heißen muß.

Straßen

Ein besonders beliebtes Feld für kreative Gestaltungen sind Straßennamen und Hausnummern.

- Vielfach ist nicht klar, ob mehrere Wörter zusammengeschrieben werden müssen, also ob es beispielsweise *Berliner Straße* oder *Berlinerstraße* heißen muß. Hier im Beispiel könnte man *Karl Marx Straße*, *Karlmarxstraße*, *Karl Marx-Straße* oder *Karl-Marx-Straße* schreiben.
- Das Wort *Straße* kann man auch mit *Str.* abkürzen oder – die Rechtschreibreform läßt grüßen – auch *Strasse* schreiben.
- Die Hausnummer kann mit und ohne Leerzeichen angehängt werden. Manche Grundstücke umfassen mehrere Hausnummern und heißen dann beispielsweise *Wachsbleiche 7 - 12*. Hier könnten dann auch folgende Varianten gebildet werden:
  - *Wachsbleiche 7-12*
  - *Wachsbleiche 7, Wachsbleiche 12*
  - *Wachsbleiche 12 - 7*
  - *Wachsbleiche 7 bis 12*
  - *Wachsbleiche 7/12*

  Viele Systeme erlauben es bei solchen Kombinationen nicht, die höhere Zahl zuerst einzugeben. Dies soll für Einheitlichkeit sorgen, macht aber auf der anderen Seite wieder Probleme. Meine Adresse im Studentenwohnheim lautete *Dolgenseestraße 28 / 24* (Haus Nummer 28, Wohnung Nummer 24), woraus einige Firmen dann *24 / 28* oder gleich nur *24* gemacht haben. Entsprechend adressierte Schreiben wurden von der Post nur sehr selten korrekt zugestellt.
- Des weiteren müssen bisweilen zusätzliche Informationen angehängt werden. Das *Q II* steht für Quergebäude, zweiter Stock, und dies läßt sich auch durchaus ganz anders formulieren.
  - Hasenheide 9 Q 2
  - Hasenheide 9Q2, Hasenheide 9QII
  - Hasenheide 9 Q-Geb. II, 9 Q-Geb 2,
  - Hasenheide 9 Q-Geb II. St, 9 Q-Geb 2. St,
  - Hasenheide 9 Quergebäude 2. Stock

Man sollte nicht vergessen, daß man auch mehrere Möglichkeiten kombinieren kann. Allein bei *Karl-Marx-Straße 9 Q II* könnte man sicher über hundert verschiedene Varianten finden.

Postleitzahl und Ort

Bei der Postleitzahl ist es dann wieder relativ einfach, da gibt es prinzipiell nur drei Möglichkeiten: *10967, 10 967, 109 67*. Man sollte hier nicht der Versuchung erliegen, die Schreibweise von Postleitzahlen per Hausnorm festzulegen. Schließlich werden sich doch einige Mitarbeiter nicht daran halten. Entweder regelt man dies mit einer Eingabemaske oder man gestaltet die Abfrage so, daß man mit allen Schreibweisen zurechtkommt.

Auch bei den Orten muß man sich nicht allzu große Gedanken machen. Es gibt zwar Personen, die *Wuppertal* mit *W-Tal* und *Ravensburg* mit *R'burg* abkürzen, aber solche Abkürzungen werden wohl selten in EDV-Systeme eingegeben. (Vorsicht: Man sollte dem Anwender prinzipiell nie vernünftiges Vorgehen unterstellen. Vielmehr zeichnen sich gute Programme dadurch aus, daß sie auch bei der Eingabe allerlei Unsinns noch verläßlich funktionieren.)

Schwierigkeiten gibt es gegebenenfalls dann noch dadurch, daß Teilortsnamen mal geschrieben werden, mal nicht. Auf diese Weise könnte aus *Berlin-Neukölln* dann *Berlin* (selten *Neukölln*) werden. Weitere Beispiele ließen sich durchaus noch finden.

Einige Leser werden sich nun fragen, weswegen man noch nach Ortsnamen suchen soll, wenn auch eine Suche nach Postleitzahlen möglich ist. Prinzipiell ist dieser Einwand richtig, er unterstellt jedoch, daß die Postleitzahlen korrekt eingegeben worden sind. Dies ist jedoch nicht immer gewährleistet, insbesondere die Umstellung auf fünfstellige Postleitzahlen hat sich als fehlerträchtig erwiesen. Die zusätzliche Prüfung über den Ortsnamen erhöht die Chance, Übereinstimmungen zu erkennen.

12.1 Generierung von Suchstrings

Um bei der Suche von Adressen eine hohe Trefferquote zu erzielen, muß die SELECT-Anweisung entsprechend gestaltet werden:

- Es wird nie eine Übereinstimmung in allen Feldern gefordert, sondern es reicht aus, wenn Übereinstimmungen oder hohe Ähnlichkeiten in einigen Feldern erreicht wird.
- Man arbeitet immer mit dem LIKE-Operator und nie mit dem =.
- Die Strings, nach denen gesucht wird, werden entsprechend aufgearbeitet. So wird beispielsweise bei Straßennamen immer nur nach dem Teilstring *str* gesucht, so daß es gleichgültig ist, ob das Wort *Straße* abgekürzt wurde oder nicht.

Letztlich wird man es nicht vermeiden können, daß die Ergebnisse der Suche manuell nachgearbeitet werden müssen. Das Programm sucht alle Datensätze heraus, bei denen eine Übereinstimmung möglich ist, und der Anwender entscheidet dann, ob tatsächlich eine Übereinstimmung vorliegt.

Bild 12.1: Generierung von Suchstrings

Wir wollen uns nun einige Möglichkeiten ansehen, aus einer Adresse geeignete Suchstrings zu generieren. Im nächsten Unterkapitel soll dann ein Programm erstellt werden, das eine Tabelle auf doppelte Einträge hin untersucht.

Bild 12.1 zeigt, wie die einzelnen Funktionen den eingegebenen String umwandeln. Die erste Funktion wandelt alle Zeichen, die keine Buchstaben sind, in %-Zeichen um, so daß bei der Verwendung des LIKE-Operators an dieser Stelle entweder kein, ein oder mehrere Zeichen stehen dürfen. Auch die deutschen Umlaute und das *ß* werden entfernt, so daß es egal ist, ob man Müller oder Mueller schreibt (...*WHERE nachname LIKE "M%ller"*).

Die nächste Funktion tut fast das gleiche, läßt aber zusätzlich noch Zahlen zu. Diese beiden Funktionen sind universell für viele Felder geeignet. Für besondere Probleme wie beispielsweise Straßennamen werden wir jedoch speziellere Funktionen benötigen.

```
AnsiString __fastcall likeqa(AnsiString rein)
// Wandelt Strings in SQL-Suchstrings um
{
  int i;
  for(i=0; i < rein.Length(); i++)
  {
    char c = rein[i+1];
    if((('a' <= c) * (c <= 'z'))+(('A' <= c) * (c <= 'Z')))
      rein[i+1] = c;
    else
      rein[i+1] = '%';
  } // for(int i=0; i<rein.Length(); i++)
  rein = "%" + rein + "%";
  while(rein.Pos("%%") > 0)
    rein.Delete(rein.Pos("%%"), 1);
  return rein;
} // likeqa
```

Zunächst werden alle Stellen des Strings einzeln daraufhin überprüft, ob es sich um einen Buchstaben handelt. Ist dies nicht der Fall, dann werden diese Zeichen durch ein %-Zeichen ersetzt. Nun wird an den Anfang und an das Ende des Strings ein %-Zeichen gesetzt. Abschließend werden alle doppelten %-Zeichen eliminiert.

```
AnsiString __fastcall likeqz(AnsiString rein)
// Wandelt Strings mit Zahlen in SQL-Suchstrings um
{
    ...
  if((('a' <= c) * (c <= 'z'))
      +(('A' <= c) * (c <= 'Z'))
      +(('0' <= c) * (c <= '9')))
    rein[i+1] = c;
  else
    rein[i+1] = '%';
    ...
} // likeqz
```

Die Funktion *likeqz* ähnelt der Funktion *likeqa*, es werden jedoch auch Ziffern akzeptiert.

12.1.1 Suche nach Straßennamen

Mit der vorherigen Funktion können Straßennamen nicht optimal aufgearbeitet werden, deshalb soll hier eine neue Funktion erstellt werden.

```
AnsiString __fastcall likeqs(AnsiString rein)
// Wandelt Straßennamen in SQL-Suchstrings um
{
  // Erseten von "Straße" (in allen Schreibweisen) durch "Str%%%"
  int i;
  i = rein.Pos("straße");
  if(i == 0)
    i = rein.Pos("Straße");
  if(i != 0)
  {
    rein[i+3] = '%';
    rein[i+4] = '%';
    rein[i+5] = '%';
  }
  i = rein.Pos("strasse");
  if(i == 0)
    i = rein.Pos("Strasse");
  if(i != 0)
  {
    rein[i+3] = '%';
    rein[i+4] = '%';
    rein[i+5] = '%';
  }
```

Zunächst soll das Wort *Straße* durch *Str%%%* ersetzt werden. Da die Funktion *Pos*, welche die Position eines Teilstrings innerhalb eines Strings sucht, auf Groß- und Kleinschreibung achtet, muß sowohl nach *Straße* als auch nach *straße* gesucht werden. Zudem wird nach der Schreibweise mit *ss* gesucht.

```
  // Ersetzen aller deutschen Umlaute und Sonderzeichen durch "%"
  // Zwischen Zahlen und Buchstaben "%" einsetzen
  bool zahln = false;
  bool zahla = false;
  int l = rein.Length();
  AnsiString s;
```

```
for(i=0; i < l; i++)
{
  char c = rein[i+1];
  if((('a' <= c) * (c <= 'z'))
    +(('A' <= c) * (c <= 'Z'))
    +(('0' <= c) * (c <= '9')))
  {
    zahln = (('0' <= c) * (c <= '9'));
    if(zahln != zahla)
    {
      s = s + "%";
      s = s + c;
    }
    else
      s = s + c;
    zahla = zahln;
  } // if((('a' <= c) * (c <= '
  else
    s = s + "%";
} // for(int i=0; i<rein.Length(); i++)
```

In dieser Schleife werden alle Zeichen daraufhin untersucht, ob es sich um gewöhnliche Buchstaben oder um Zahlen handelt. Die deutschen Umlaute und die Sonderzeichen werden durch ein %-Zeichen ersetzt. Des weiteren werden zwischen Zahlen und Buchstaben %-Zeichen eingefügt, so daß es keine Rolle spielt, ob hier ein Leerzeichen eingesetzt worden ist oder nicht.

```
// Römische Zahlenzeichen am Ende des Strings durch "%" ersetzen
for(i=int(0.75*(l+k)-3); i<l+k; i++)
{
  if((s[i+1] == 'I') + (s[i+1] == 'V'))
    s[i+1] = '%';
}
```

Für die Angabe von Ergänzungen zur Hausnummer werden oft lateinische Zahlen verwendet (zum Beispiel *IV Stock*). Damit keine Probleme dadurch entstehen, daß mal lateinische, mal arabische Zahlen verwendet werden, sollen die Zeichen *V* und *I* auch durch %-Zeichen ersetzt werden. Dies soll jedoch nur gegen Ende des Strings geschehen.

```
// Doppelte "%"-Zeichen eliminieren und Funktion beenden
s = "%" + s + "%";
while(s.Pos("%%") > 0)
  s.Delete(s.Pos("%%"), 1);
```

```
  return s;
} // likeqs
```

Schließlich werden noch %-Zeichen am Anfang und am Ende eingefügt sowie alle doppelt vorkommenden %-Zeichen entfernt.

12.1.2 Hausnummern ohne Zusätze

Vielfach werden Zusätze zu den Hausnummern mal eingegeben, mal ignoriert. Die folgende Funktion ähnelt der vorhergehenden. Nach dem Auftreten einer Ziffernfolge – in der Regel also der Hausnummer – werden alle weiteren Zeichen durch %-Zeichen ersetzt. So wird eine Adresse auch dann gefunden, wenn in ihr die Zusätze nicht erwähnt wurden. Die Funktion *likeqh* gleicht am Anfang und am Ende der eben besprochenen Funktion *likeqs*, lediglich die Zeichenüberprüfung sieht etwas anders aus:

```
  // Ersetzen aller deutschen Umlaute und Sonderzeichen durch "%"
  // Zwischen Zahlen und Buchstaben "%" einsetzen
  // Nach der ersten Zahl alle Zeichen durch "%" ersetzen
  bool zahln = false;
  bool zahla = false;
  bool ende = false;
  int l = rein.Length();
  AnsiString s;
  for(i=0; i < l; i++)
  {
    char c = rein[i+1];
    zahln = (('0' <= c) * (c <= '9'));
    if((zahln == false)*(zahla == true))
      ende = true;
    if(((('a' <= c) * (c <= 'z'))
      +(('A' <= c) * (c <= 'Z'))
      +(('0' <= c) * (c <= '9')))*(ende == false))
    {
      if(zahln != zahla)
      {
        s = s + "%";
        s = s + c;
      }
      else
        s = s + c;
```

```
      zahla = zahln;
    } // if((('a' <= c) * (c <= '
    else
      s = s + "%";
  } // for(int i=0; i<rein.Length(); i++)
```

Entscheidend ist hier die zweite Anweisung in der *for*-Schleife. Tritt hier eine Ziffer auf, dann wird die Variable *zahln* auf *true* gesetzt, die Anweisung *zahla = zahln* setzt somit auch die Variable *zahla* auf *true*. Sobald dann wieder ein Buchstabe auftritt, ist *zahln* gleich *false* und *zahla* gleich *true*, somit wird die Variable *ende* auf *true* gesetzt, und alle weiteren Zeichen werden durch das %-Zeichen ersetzt.

12.1.3 Ortsnamen

Die meisten Ortsnamen stellen bei der Suche kein Problem dar – vorausgesetzt, sie werden korrekt geschrieben. Lediglich bei zusammengesetzen Ortsnamen wie *Berlin-Hohenschönhausen* oder *VS-Schwenningen* könnten Probleme auftreten. Diese sollen hier dadurch behoben werden, daß der eingegebene Ortsname in den ersten und zweiten Teilstring aufgeteilt wird.

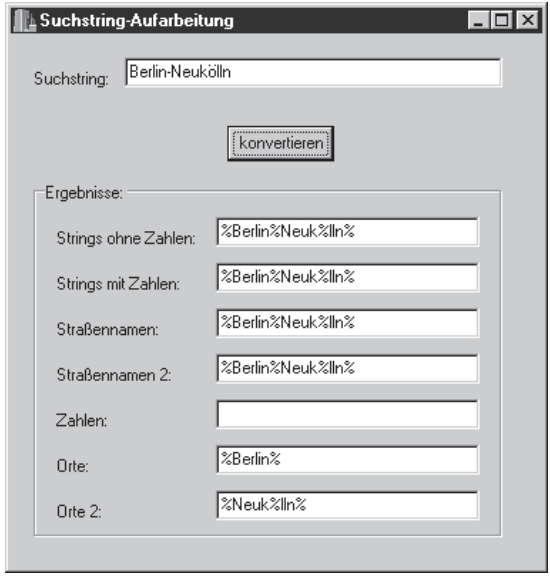


Bild 12.2: Aufteilung der Ortsnamen

```
AnsiString __fastcall likeqo(AnsiString rein)
// Wandelt Ortsnamen in SQL-Suchstrings um
// eliminiert nach dem ersten Teilstring alle Zeichen
{
  int i;
  bool ende = false;
  for(i=0; i<rein.Length(); i++)
  {
    char c = rein[i+1];
    if((('a' <= c) * (c <= 'z'))
      +(('A' <= c) * (c <= 'Z'))
      +(c == 'ä') +(c == 'ö') +(c == 'ü') +(c == 'ß')
      +(c == 'Ü') +(c == 'Ö') +(c == 'Ü')) * (ende == false))
    {
      if((('a' <= c) * (c <= 'z'))
          +(('A' <= c) * (c <= 'Z')))
        rein[i+1] = c;
      else
        rein[i+1] = '%';
    } //  if (außen)
    else
    {
      rein[i+1] = '%';
      ende = true;
    } //  else (außen)
  } // for(i=0; i<rein.Length(); i++)
  rein = "%" + rein + "%";
  while(rein.Pos("%%") > 0)
    rein.Delete(rein.Pos("%%"), 1);
  if(rein.Length() == 1)
    rein = "";
  return rein;
} // likeqo
```

Diese Funktion geht davon aus, daß beim Auftreten des ersten Zeichens, welches kein Buchstabe ist, der erste Teilstring zu Ende ist. Dann wird die Variable *ende* auf *true* gesetzt, und folglich werden alle folgenden Zeichen durch ein *%*-Zeichen ersetzt. Damit aus *Köln-Porz* nicht der Suchstring *%K%*, sondern *%K%ln%* generiert wird, müssen hier auch die deutschen Umlaute in die *if*-Anweisung aufgenommen werden.

Die Funktion *likeqp* generiert aus einem Ortsnamen die zweite Hälfte als Suchstring, wie das Bild 12.2 zeigt. Diese Funktion ist nahezu identisch mit *likeqo*, so daß sie hier nicht abgedruckt werden soll. Die letzte Zeile ist bei *likeqp* besonders wichtig: Aus Ortsnamen, welche aus nur einem Wort bestehen, würde diese Funktion andernfalls den Suchstring % generieren, somit würden bei der Suche alle Datensätze zurückgegeben, was wenig sinnvoll wäre.

12.1.4 Suche nach Nummern

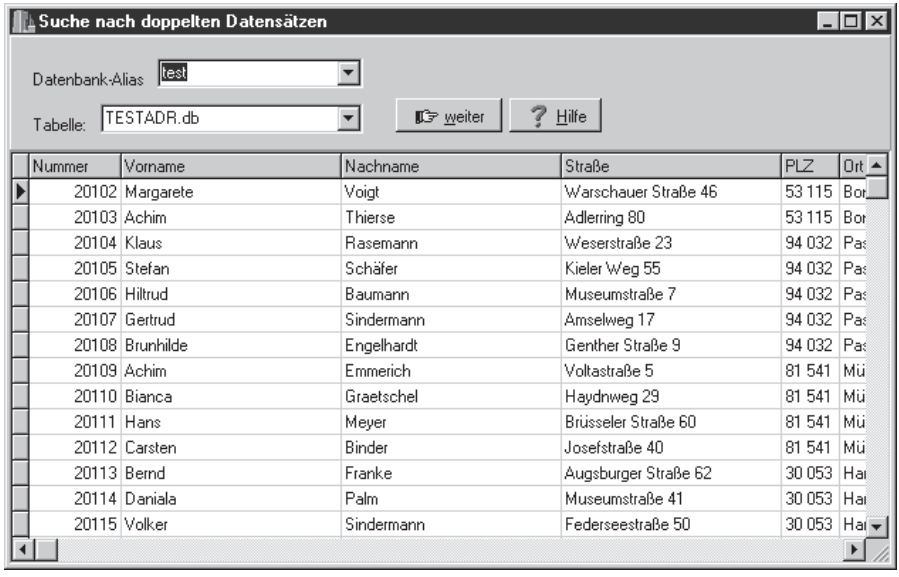
Das prinzipielle Vorgehen beim Suchen nach Nummern wurde schon in Kapitel 3 vorgestellt – es wird einfach zwischen alle Zahlen eine %-Zeichen eingefügt.

```
AnsiString __fastcall likeqt(AnsiString rein)
// Wandelt Zahlen wie Telefonnummern oder
// Postleitzahlen in SQL-Suchstrings um
{
  int i;
  AnsiString s;
  for(i=0; i<rein.Length(); i++)
  {
    char c = rein[i+1];
    if(('0' <= c)*(c <= '9'))
      s = s + c + '%';
  }
  return s;
} // likeqt
```

12.2 Entfernen doppelter Adressen

Nun soll – wie versprochen – ein Programm erstellt werden, mit dem sich doppelte Datensätze in einer Tabelle auffinden lassen und der überflüssige sich entfernen läßt. Es muß sich bei den Datensätzen nicht zwangsläufig um Adressen handeln, das Programm wurde jedoch auf diese Anwendung hin optimiert.

Wir wollen das Programm als eigenständige Anwendung konzipieren. Dies bedeutet, daß das Programm in der Lage sein soll, mit verschiedenen Tabellen zu arbeiten, ohne daß es dafür verändert werden muß. Es müssen also insbesondere Möglichkeiten geschaffen werden, Aliase und Tabellen auszuwählen. Zudem muß das Programm darauf vorbereitet sein, daß sich die Spaltennamen ändern.



Nummer	Vorname	Nachname	Straße	PLZ	Ort
20102	Margarete	Voigt	Warschauer Straße 46	53 115	Bor
20103	Achim	Thierse	Adlerring 80	53 115	Bor
20104	Klaus	Rasemann	Weserstraße 23	94 032	Pas
20105	Stefan	Schäfer	Kieler Weg 55	94 032	Pas
20106	Hiltrud	Baumann	Museumstraße 7	94 032	Pas
20107	Gertrud	Sindermann	Amselweg 17	94 032	Pas
20108	Brunhilde	Engelhardt	Genther Straße 9	94 032	Pas
20109	Achim	Emmerich	Voltastraße 5	81 541	Mü
20110	Bianca	Graetschel	Haydnweg 29	81 541	Mü
20111	Hans	Meyer	Brüsseler Straße 60	81 541	Mü
20112	Carsten	Binder	Josefstraße 40	81 541	Mü
20113	Bernd	Franke	Augsburger Straße 62	30 053	Han
20114	Daniala	Palm	Museumstraße 41	30 053	Han
20115	Volker	Sindermann	Federseestraße 50	30 053	Han

Bild 12.3: Auswahl der Datenbank und der Tabelle

12.2.1 Auswählen von Alias und Tabelle

Das ganze Programm ist in einem einzigen Formular untergebracht, das von einer *TNotebook*-Komponente ausgefüllt wird. Diese hat die Seiten *Verbinden*, *Definieren*, *SQL* und *Suchen*. Auf der Seite *Verbinden* – sie wird in Bild 12.3 gezeigt – werden der Alias der Datenbank und der Tabellenname ausgewählt.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Notebook1->ActivePage = "Verbinden";
  Session->GetAliasNames(ComboBox1->Items);
}
```

Beim Starten des Programms muß zunächst die Liste der verfügbaren BDE-Aliase in *ComboBox1* geschrieben werden. Des weiteren wird die *Notebook*-Seite *Verbinden* ausgewählt – diese Anweisung sorgt dafür, daß stets die korrekte *Notebook*-Seite angezeigt wird, ohne daß dies jeweils vor dem Starten des Programms im Objektinspektor eingestellt werden muß.

```
void __fastcall TForm1::ComboBox1Change(TObject *Sender)
// Datenbank-Alias auswählen
{
  Table1->Close();
```

```
  Session->GetTableNames(ComboBox1->Text, "", true,
    false, ComboBox2->Items);
  ComboBox2->Text = "";
}
```

Sobald ein BDE-Alias ausgewählt wurde, werden alle dort vorhandenen Tabellennamen nach *ComboBox2* geschrieben. Damit keine Diskrepanzen zwischen den Anzeigen in *ComboBox1* ,*ComboBox2* und *DBGrid1* auftreten, wird *Table1* geschlossen und der Text in *ComboBox2* gelöscht.

```
void __fastcall TForm1::ComboBox2Change(TObject *Sender)
// Tabelle auswählen
{
  DBText1->DataField = "";
  DBText2->DataField = "";
    ...
  DBText9->DataField = "";

  Table1->Close();
  Table1->DatabaseName = ComboBox1->Text;
  Table1->TableName = ComboBox2->Text;
  try
  {
    Table1->Open();
    Table2->Close();
    Table2->DatabaseName = ComboBox1->Text;
    Table2->TableName = ComboBox2->Text;
    Table2->Open();
    Query1->Close();
    Query1->DatabaseName = ComboBox1->Text;

    ComboBox3->Items->Clear();
    ComboBox4->Items->Clear();
      ...
    ComboBox37->Items->Clear();

    ComboBox3->Text = "";
    ComboBox4->Text = "";
      ...
    ComboBox37->Text = "";

    for(int i = 0; i < Table1->FieldCount; i++)
    {
```

```
      ComboBox3->Items->Add(Table1->Fields[i]->FieldName);
      ComboBox4->Items->Add(Table1->Fields[i]->FieldName);
        ...
      ComboBox37->Items->Add(Table1->Fields[i]->FieldName);
    }
  } // try
  catch(...)
  {
    ShowMessage("Tabelle läßt sich nicht öffnen");
  } // catch
} // ComboBox2Change
```

Auf einer anderen Seite von *Notebook1* werden wir später einige *TDBText*-Komponenten einfügen. Deren Eigenschaft *DataField* muß zur Laufzeit zugewiesen werden, da ja zur Entwurfszeit noch nicht feststeht, wie die Spaltennamen lauten werden. Wird eine andere Tabelle ausgewählt, dann werden sich mit hoher Wahrscheinlichkeit auch die Spaltennamen ändern. Damit hier keine Exception ausgelöst wird, wird die Eigenschaft *DataField* bei allen *DBText*-Komponenten zurückgesetzt.

Sobald nun der Tabellenname ausgewählt wurde, können die *TTable*-Datenzugriffskomponenten geöffnet werden. Der Komponente *Query1* kann jetzt schon der Datenbank-Alias als *DatabaseName* zugewiesen werden, zuvor muß die Ab-

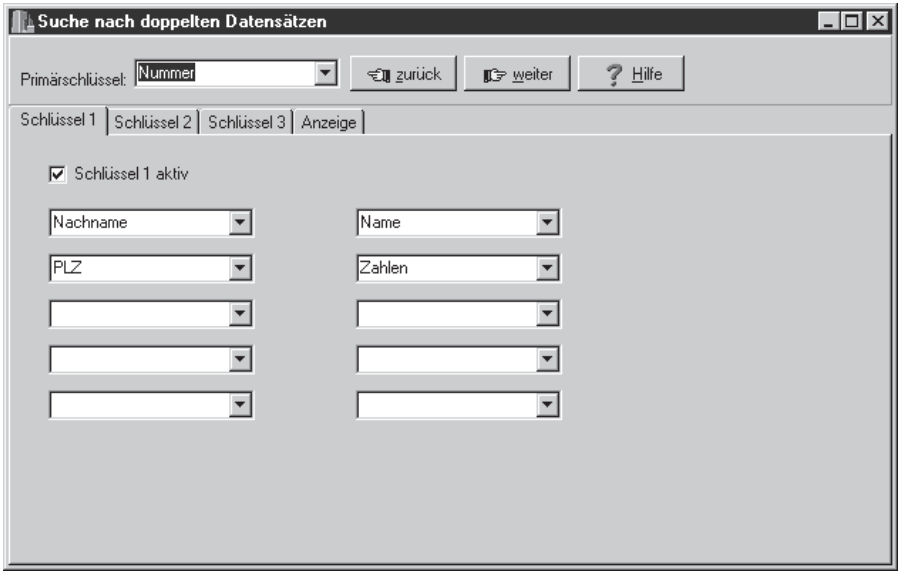


Bild 12.4: Definition der Schlüssel

frage verständlicherweise geschlossen werden. Des weiteren müssen alle Combo-Boxen, mit deren Hilfe sich ein Spaltenname auswählen läßt, auf die neue Tabelle umgestellt werden.

12.2.2 Definieren der Suchkriterien

Auf der nächsten Notebook-Seite werden die Suchkriterien definiert. Es muß also ausgewählt werden, in welcher Kombination von Feldern eine Übereinstimmung gefunden werden muß, damit das Programm einen doppelten Datensatz meldet. Das Programm erlaubt dabei die Verwendung von bis zu drei Schlüsseln (*Key1* bis *Key3*); dabei reicht es aus, daß eine der drei Schlüsselbedingungen erfüllt wird.

In Bild 12.4 wird der Schlüssel *Key1* über die Felder *Nachname* und *PLZ* definiert. Die zweite Reihe der ComboBoxen gibt an, mit welcher Funktion (siehe Kapitel 12.1) der Suchstring aufbereitet wird.

Die *TabbedNotebook*-Seite *Anzeige* behandeln wir später. In Kopf-Panel kann zudem eingegeben werden, welches der Felder den Primärschlüssel bildet. Für die Suche nach den doppelten Adressen ist dies nicht erforderlich, sondern nur für die schnelle Suche nach einem Datensatz in der Tabelle.

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
  Notebook1->ActivePage = "Definieren";
}
```

Da alle nötigen Vorbereitungen schon bei der Wahl der Tabelle getroffen wurden, muß mit dem *weiter*-Button lediglich die *Notebook*-Seite gewechselt werden.

12.2.3 Zuweisung der Feldnamen

Auf der *Notebook*-Seite *Suchen* soll die jeweils aktuelle Adresse mit Hilfe einer Reihe von *TDBText*-Komponenten angezeigt werden. Da diese Komponenten jeweils nur ein Datenfeld anzeigen können, muß die Eigenschaft *DataField* auf den entsprechenden Wert gesetzt werden. Dies kann jedoch nicht zur Entwurfszeit geschehen, da das Programm mit verschiedenen Tabellen zusammenarbeiten soll, deren Feldnamen unterschiedlich lauten werden. Deshalb muß man dem Anwender eine Möglichkeit geben, den einzelnen *TDBText*-Komponenten eines der vorhandenen Datenbankfelder zuzuweisen. Auch dies soll wieder mit Combo-Boxen geschehen.

Bild 12.5 zeigt die entsprechende *TabbedNotebook*-Seite. Der Anwender kann über die einzelnen Spalten frei verfügen und sie den jeweiligen Plätzen zuordnen, er kann sich aber auch auf die Anzeige einiger Spalten beschränken.

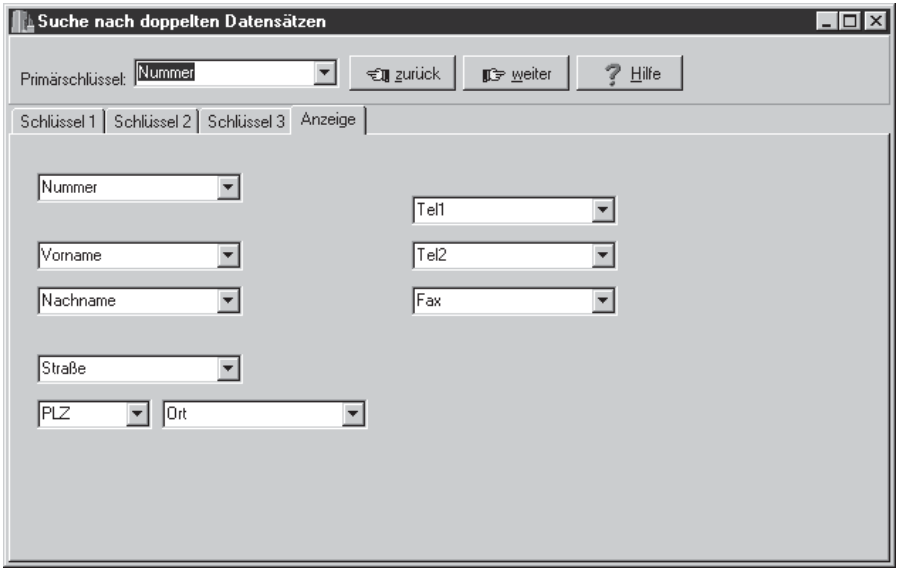


Bild 12.5: Auswahl der Tabellenspalten für die Anzeige des Datensatzes

12.2.4 Generierung der SQL-Anweisung

Wird nun der Button *weiter* betätigt, dann wird die SQL-Anweisung generiert und in *Memo1* angezeigt. Die Anzeige der SQL-Anweisung erlaubt dem Anwender den schnellen Überblick über die Wirkungsweise der Programms. Außerdem wird ihm auf diese Weise schnell auffallen, wenn alle drei CheckBoxen nicht gesetzt sind, und somit keiner der drei Schlüssel aktiv ist.

```
void __fastcall TForm1::BitBtn4Click(TObject *Sender)
// Button "Weiter" auf der Seite "Definieren"
{
  Notebook1->ActivePage = "SQL";
  Memo1->Clear();
  Memo1->Lines->Add("SELECT DISTINCT * FROM '"
    + ComboBox2->Text.LowerCase() + "'");
  Memo1->Lines->Add("  WHERE");

  if((CheckBox1->Checked)*(ComboBox13->Text != ""))
  {
    Memo1->Lines->Add("    (");
    Memo1->Lines->Add("     (LOWER("
```

```
      + ComboBox13->Text.LowerCase()
      + ") LIKE LOWER(:para11))");
    if(ComboBox14->Text != "")
      Memo1->Lines->Add("      AND(LOWER("
        + ComboBox14->Text.LowerCase()
        + ") LIKE LOWER(:para12))");
      ...
    if(ComboBox17->Text != "")
      Memo1->Lines->Add("      AND(LOWER("
        + ComboBox17->Text.LowerCase()
        + ") LIKE LOWER(:para15))");
    Memo1->Lines->Add("    )");
  } // if(CheckBox1->Checked)*(ComboBox13->Text != ""))

  if((CheckBox2->Checked)*(ComboBox23->Text != ""))
  {
    if((CheckBox1->Checked)*(ComboBox13->Text != ""))
      Memo1->Lines->Add("    OR");
      Memo1->Lines->Add("    (");
        ...
  } // if(CheckBox2->Checked)*(ComboBox23->Text != ""))

  if((CheckBox3->Checked)*(ComboBox33->Text != ""))
  {
    if(((CheckBox1->Checked)*(ComboBox13->Text != ""))
        +((CheckBox2->Checked)*(ComboBox23->Text != "")))
      Memo1->Lines->Add("    OR");
      Memo1->Lines->Add("    (");
        ...
  } // if(CheckBox3->Checked)*(ComboBox33->Text != ""))
```

Wie Bild 12.6 zeigt, ist die generierte SQL-Anweisung nicht sonderlich spektakulär. Die ausgewählten Felder werden mit Hilfe des LIKE-Operators mit dem dazugehörenden Parameter verglichen. Die Bedingungen innerhalb eines Schlüssels werden mit einer AND-Verknüpfung, die einzelnen Schlüssel mit einer OR-Verknüpfung verbunden. Des weiteren muß der Tabellenname korrekt eingefügt und die Klammern müssen an der richtigen Stellen gesetzt werden.

```
  // Anzeige einrichten
  if(ComboBox4->Text == "")
    DBText1->Visible = false;
```

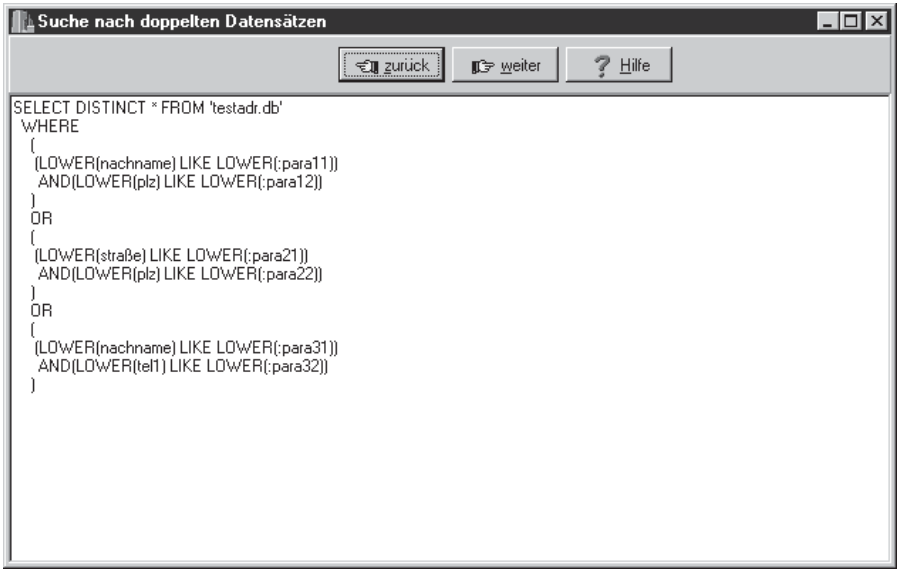


Bild 12.6: Die SQL-Anweisung

```
  else
  {
    DBText1->Visible = true;
    DBText1->DataField = ComboBox4->Text;
  }
    ...
  if(ComboBox12->Text == "")
    DBText9->Visible = false;
  else
  {
    DBText9->Visible = true;
    DBText9->DataField = ComboBox12->Text;
  }
} // TForm1::BitBtn4Click
```

Außerdem muß noch bei den *TDBText*-Komponenten die Eigenschaft *DataField* gesetzt werden. Ist in der ComboBox kein Feld ausgewählt, dann soll die dazugehörende *TDBText*-Komponente nicht angezeigt werden.

Auf der *Notebook*-Seite *SQL* wird lediglich die SQL-Anweisung angezeigt. Die drei Buttons benötigen keine anderen Anweisungen als die zur Weiterschaltung auf die nächste bzw. letzte *Notebook*-Seite bzw. zur Anzeige der Online-Hilfe benötigten.

12.2.5 Suchen nach doppelten Adressen

Bei der Suche nach doppelten Adressen werden nacheinander die Felder aller Adressen in die gewünschten Suchstrings aufbereitet und als Parameter der SQL-Anweisung zugewiesen.

```
void __fastcall TForm1::BitBtn12Click(TObject *Sender)
// weitersuchen
{
  try
  {
    FAbbruch = false;
    int i = 0;
    AnsiString s;
    DataSource1->Enabled = false;
    Label4->Visible = true;
    Label4->Caption = Table1->FieldByName(ComboBox3->Text)-
>AsString;
    Screen->Cursor = crHourGlass;
```

Zunächst werden einige Variablen auf sinnvolle Ausgangswerte gesetzt. Um den Programmablauf zu beschleunigen, soll die Anzeige nicht laufend aktualisiert werden, die *DataSource*-Eigenschaft *Enabled* wird deshalb auf *false* gesetzt. Außerdem wird der Cursor in eine Sanduhr verwandelt.

```
    while((!Table1->Eof) * (FAbbruch == false))
    {
      if(FErster)
        FErster = false;
      else
        Table1->Next();
      i++;
      if(i==10)
      {
        Label4->Caption
          = Table1->FieldByName(ComboBox3->Text)->AsString;
        Application->ProcessMessages();
        i = 0;
      }
      if(FAbbruch == true)
      {
        DataSource1->Enabled = true;
        MessageBeep(-1);
```

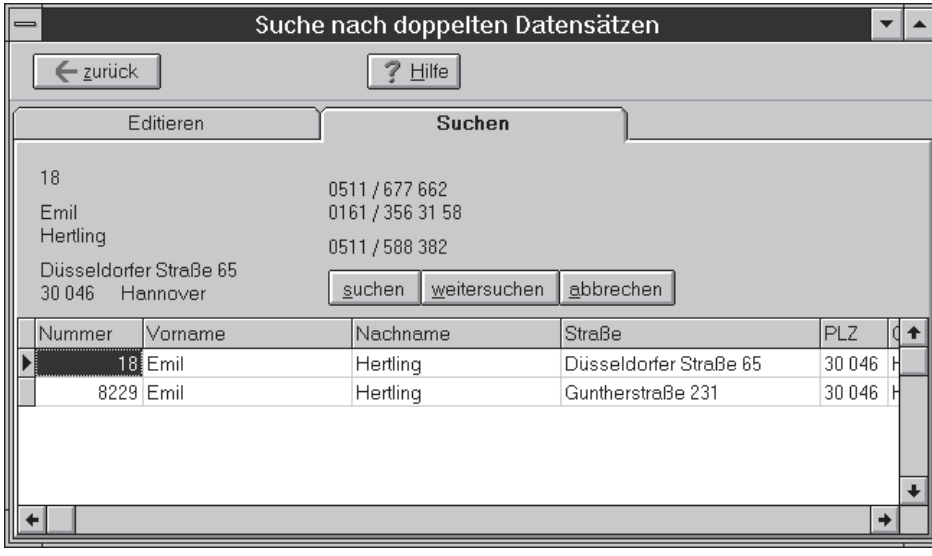


Bild 12.7: Teilweise Übereinstimmung in einem Datensatz

```
    Label4->Visible = false;
    Screen->Cursor = crDefault;
    ShowMessage("Suche abgebrochen");
  }
  Query1->Close();
```

Mit einer *while*-Schleife werden alle Datensätze von *Table1* durchlaufen und daraufhin überprüft, ob es Adressen mit ähnlichem Inhalt gibt. Die *if (FErster)...*-Konstruktion sorgt dafür, daß beim ersten Durchlauf noch keine *Next*-Anweisung erfolgt und somit auch nach Datensätzen gesucht wird, die dem ersten Datensatz ähneln. Man könnte nun einwenden, daß, wenn der Datensatz x (x > 1) dem Datensatz 1 ähnelt, auch der Datensatz 1 dem Datensatz x ähnelt und daß diese Übereinstimmung dann gefunden werden würde, wenn der Datensatz x überprüft wird.

Jedoch trifft die Annahme, die diesem Einwand zugrundeliegt, nicht immer zu: Lautet beispielsweise bei Datensatz x der Vorname *Hans-Jürgen* und bei Datensatz 1 *Hans*, dann würde man mit der Anweisung *LIKE %Hans%* den Vornamen *Hans-Jürgen* finden, mit der Anweisung *LIKE %Hans%Jürgen%* aber nicht den Vornamen *Hans*.

Der Programmablauf gestaltet sich auch bei schnellen Rechnern relativ zäh. Damit der Anwender erkennen kann, daß das Programm nicht abgesürzt ist, soll deshalb hin und wieder die gerade aktuelle Datensatznummer angezeigt wer-

den. Zudem soll die Möglichkeit vorgesehen werden, durch Setzen der Variablen *FAbbruch* die Schleife zu verlassen.

```
if((CheckBox1->Checked)*(ComboBox13->Text != ""))
{
  s = Table1->FieldByName(ComboBox13->Text)->AsString;
  if(ComboBox18->ItemIndex == 0)
    s = likeqa(s);
    ...
  if(ComboBox18->ItemIndex == 6)
    s = likeqp(s);
  Query1->ParamByName("para11")->AsString = s.LowerCase();

  if(ComboBox14->Text != "")
  {
    s = Table1->FieldByName(ComboBox14->Text)->AsString;
    if(ComboBox19->ItemIndex == 0)
      s = likeqa(s);
      ...
    if(ComboBox19->ItemIndex == 6)
      s = likeqp(s);
    Query1->ParamByName("para12")->AsString = s.LowerCase();
  } // if(ComboBox14->Text != "")
....
```

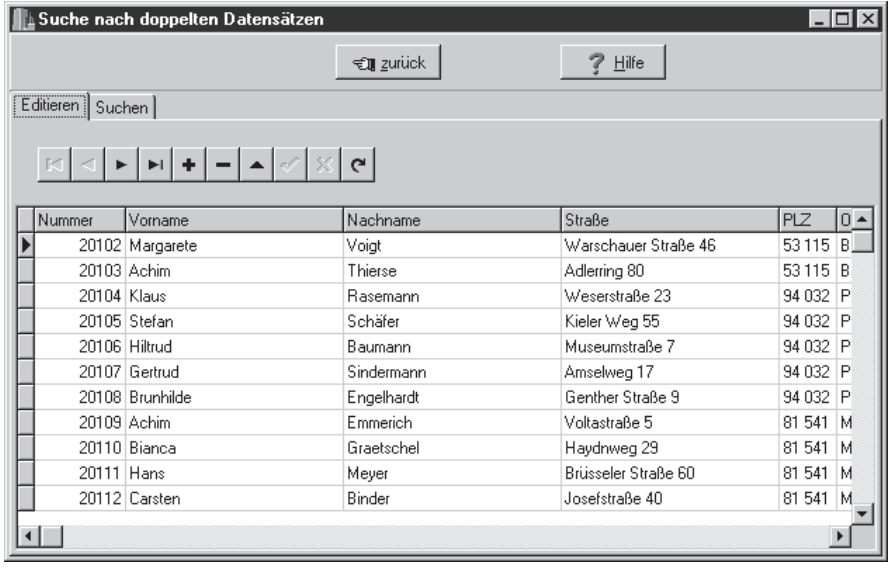
Nun werden die Feldinhalte in den String *s* eingelesen. Je nach dem, welche Funktion in der *ComboBox* ausgewählt worden ist, wird der String *s* mit einer anderen Funktion aufbereitet. Danach wird er dem entsprechenden Parameter zugewiesen. Insgesamt umfaßt das Programm 15 solcher Blöcke, die hier jedoch nicht alle abgedruckt werden sollen.

```
  Query1->Open();
  if(Query1->RecordCount > 1)
  {
    DataSource1->Enabled = true;
    Label4->Visible = false;
    Screen->Cursor = crDefault;
    FAbbruch = true;
  } // if(Query1->RecordCount > 1)
} // while((!Table1->Eof) * (FAbbruch == false))
DataSource1->Enabled = true;
Label4->Visible = false;
```

```
    Screen->Cursor = crDefault;
    if(Table1->Eof)
      ShowMessage("Gesamter Datenbestand durchsucht");
  } // try
  catch(...)
  {
    ShowMessage("Fehler aufgetreten");
    DataSource1->Enabled = true;
    Label4->Visible = false;
    Screen->Cursor = crDefault;
    MessageBeep(-1);
  } // catch
} // TForm1::BitBtn12Click
```

Nun kann die Abfrage mit *Open* gestartet werden. Dem aktuellen Datensatz ist mindestens ein Datensatz ähnlich, und das ist dieser Datensatz selbst. Sind ihm noch weitere Datensätze ähnlich, dann wird die Schleife an dieser Stelle abgebrochen, und die gefundenen Datensätze werden angezeigt.

Da in dieser Prozedur keine *First*-Anweisung auftritt, kann später an der Stelle mit der Suche fortgefahren werden, an der sie unterbrochen wurde. Soll bei der Suche mit dem ersten Datensatz begonnen werden, dann muß der Button *suchen* verwendet werden.



Nummer	Vorname	Nachname	Straße	PLZ	O
20102	Margarete	Voigt	Warschauer Straße 46	53 115	B
20103	Achim	Thierse	Adlerring 80	53 115	B
20104	Klaus	Rasemann	Weserstraße 23	94 032	P
20105	Stefan	Schäfer	Kieler Weg 55	94 032	P
20106	Hiltrud	Baumann	Museumstraße 7	94 032	P
20107	Gertrud	Sindermann	Amselweg 17	94 032	P
20108	Brunhilde	Engelhardt	Genther Straße 9	94 032	P
20109	Achim	Emmerich	Voltastraße 5	81 541	M
20110	Bianca	Graetschel	Haydnweg 29	81 541	M
20111	Hans	Meyer	Brüsseler Straße 60	81 541	M
20112	Carsten	Binder	Josefstraße 40	81 541	M

Bild 12.8: Editieren der Datensätze

```
void __fastcall TForm1::BitBtn11Click(TObject *Sender)
// suchen (von Anfang an)
{
  Table1->First();
  FErster = true;
  BitBtn12Click(Sender);
}
```

12.2.6 Adressen editieren

Das Programm findet lediglich Adressen, die dem jeweiligen Datensatz ähnlich sind. Die Entscheidung, ob hier ein doppelter Datensatz vorliegt, muß der Anwender treffen. Ist er der Ansicht, daß ein doppelter Datensatz vorliegt, dann sollte er ihn gleich aus diesem Programm heraus löschen können. Mit einem Doppelklick auf das *DBGrid* soll auf die entsprechende *TabbedNotebook*-Seite gewechselt und der entsprechende Datensatz gleich ausgewählt werden können.

```
void __fastcall TForm1::DBGrid2DblClick(TObject *Sender)
{
  TVarRec q(int(Query1->FieldByName(ComboBox3->Text)->AsInteger));
  Table2->FindKey(&q, 0);
  TabbedNotebook2->ActivePage = "Editieren";
}
```

Beachten Sie bitte, daß zum Editieren nicht *Table1* verwendet werden darf, wäre doch dann nicht gewährleistet, daß an derselben Stelle weitergesucht wird. Fügen Sie deshalb dafür eine weitere *TTable*-Komponente ein.

13 Der Local InterBase Server

Der *Local InterBase Server* ist die Einplatz-Version des *InterBase Workgroup Servers*, eines Client-Server-Systems. Vom C++Builder aus greift die BDE auf den *Local InterBase Server* (*LIBS*) zu, der dann seinerseits auf die Datenbankdatei zugreift. Der Einsatz des LIBS bringt folgende Vorteile und Nachteile:

- Die Anwendung wird gleich als Client-Server-Anwendung erstellt. Soll später – wenn beispielsweise die Firma expandiert – auf ein Mehplatzsystem aufgestockt werden (*Upsizing*), dann kann der *InterBase Workgroup Server* verwendet werden, und die Anwendung läuft dann (theoretisch) ohne Änderung.
- Der LIBS bietet Funktionen, welche beispielsweise bei Paradox-Datenbanken nicht oder zumindest nicht in diesem Umfang verfügbar sind (Zugangskontrolle, Backup im laufenden Betrieb, *Stored Procedures*, usw.).
- Die Weitergabe des LIBS ist mit der Entrichtung von Lizenzgebühren verbunden. Wenn Sie eine Datenbankanwendung für einen Kunden maßschneidern, dann sind diese Kosten vernachlässigbar, für Share- und Freeware-Applikationen ist dies allerdings ein Problem.

Der *Local InterBase Server* ist in der *Professional*- und der *Client-Server*-Version des C++Builders vorhanden, jedoch nicht in dessen Standard-Version. Den Client-Server-Versionen liegt eine Lizenz des *InterBase Workgroup Servers* für zwei Plätze bei, so daß hier eine echte Mehrplatzdatenbank aufgebaut werden kann (nur dann läßt sich beispielsweise eine Aussage über die Performance der Datenbank machen, wenn sich der Datenaustausch über ein Netzwerk abwickelt.

13.1 Starten und Beenden des Servers

Der LIBS wird beim Starten von Windows gestartet, er ist dann als Icon in der Task-Bar zu sehen.

Bild 13.1: Taskbar-Icon des Local-InterBase-Servers

Mit einem Doppelklick auf dieses Icon rufen Sie ein Dialogfenster auf, welches die Eigenschaften des Servers anzeigt.

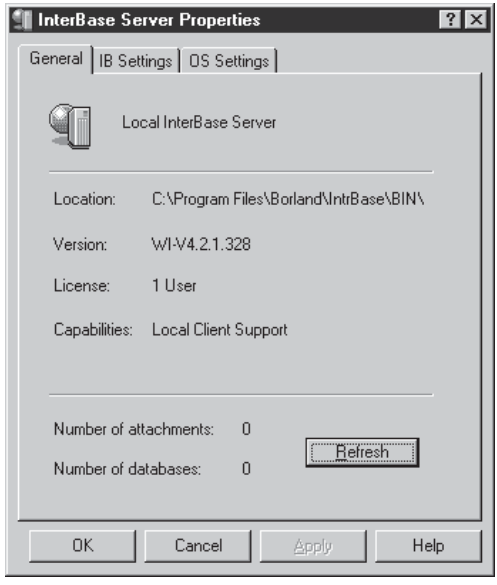


Bild 13.2: Das Icon des InterBase Servers

13.2 Der InterBase Server Manager

Mit dem *InterBase Server Manager* können einige Server-Funktionen gestartet werden, beispielsweise das Erstellen von Backups. Dieses Programm kann gleichzeitig mit dem *Local InterBase Server* und dem *InterBase Server for Workgroups* arbeiten, das erklärt einige Funktionen, die allein mit dem LIBS nicht viel Sinn machen würden.

13.2.1 Anmelden

Bevor mit dem *InterBase Server Manager* gearbeitet werden kann, muß sich der Anwender zunächst anmelden. Dazu wählt man FILE | SERVER LOGIN, worauf sich ein Fenster nach Bild 13.3 öffnet. Hier kann man wählen, ob auf dem LIBS (*Local Engine*) oder einem Netzwerk-Server gearbeitet werden soll.

Des weiteren werden ein Benutzername und das dazugehörende Paßwort eingegeben. Nach der Installation ist der Benutzer *SYSDBA* installiert, dessen Paßwort *masterkey* (Paßwort in Kleinbuchstaben) lautet. Sobald damit zu rechnen ist, daß auch andere Personen am Rechner arbeiten, spätestens dann also, wenn die

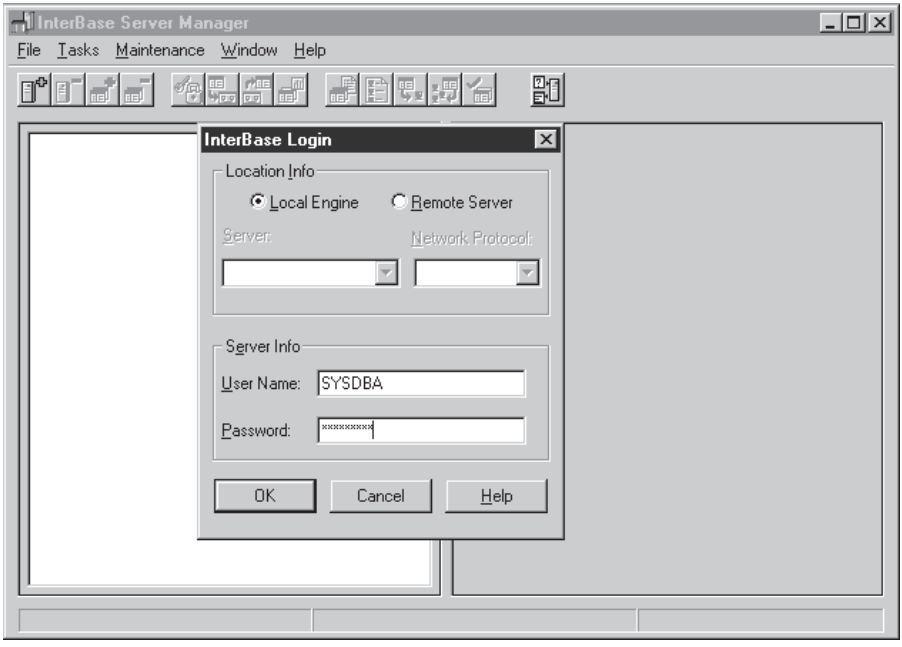


Bild 13.3: Der InterBase Server Manager

Applikation ausgeliefert wird, sollte das Paßwort des *SYSDBA* geändert werden, denn der *SYSDBA* kann auf alle Tabellen zugreifen, ohne daß ihm dies mit GRANT ausdrücklich erlaubt werden müßte.

Nach dem Anmelden wird nun auf der linken Seite angezeigt, daß man mit dem *Local Server* verbunden ist. Solange noch keine Verbindung zu einer Datenbank besteht, sind noch nicht alle Funktionen verfügbar. Mit FILE | DATABASE CONNECT kann die Verbindung nun hergestellt werden. (Anmerkung: Viele der Menübefehle sind auch als SpeedButton verfügbar.) Hier muß lediglich der Name der Datenbank in einer ComboBox eingegeben oder ausgewählt werden. Mit dem Button *Browse* kann man die entsprechende Datenbankdatei suchen.

Im Gegensatz zu den Desktop-Datenbanken (dBase, Paradox) sind bei den meisten Client-Server-Datenbanken (somit auch hier) alle Tabellen (und andere Daten) in einer einzigen Datei untergebracht, die hier mit **.gdb* endet. Bei den **.gbk*-Dateien handelt es sich um Backup-Dateien.

Die Dateiendungen sind übrigens frei wählbar: Sie können beispielsweise die Datei *employee.gdb* in *murks.dat* umbenennen und danach ganz normal eine Verbindung herstellen (ein Filter **.dat* existiert zwar nicht, aber Sie können immer noch *All Files* *.* wählen).

13.2.2 Benutzer anmelden

Mit TASKS | USER SECURITY können Sie neue Benutzer anmelden oder bei bekannten Benutzern das Paßwort ändern. Um diese Funktion nutzen zu können, müssen Sie sich allerdings als *SYSDBA* angemeldet haben; ist dies nicht geschehen, können Sie noch nicht einmal Ihr eigenes Paßwort ändern.

Die Definition eines Benutzers geschieht mittels der Eingabe seines Namens, gefolgt von der zweimaligen Eingabe seines Paßworts, das dabei durch *-Zeichen dargestellt wird. Des weiteren können Sie noch den vollständigen Namen des Benutzers eingeben, besondere Vorteile hat dies allerdings nicht.

Der Systemadministrator

Der Benutzer mit dem Namen *SYSDBA* wird als System-Administrator behandelt und besitzt somit alles, was man sich so an Zugriffsrechten wünschen kann. Das Anmelden neuer Benutzer ist nur dem SYSDBA möglich. Diese relativ grobe Trennung in zwei Gruppen von Zugriffsrechten (*SYSDBA* und andere Benutzer) ist zwar schön übersichtlich, allerdings nicht immer ganz zweckmäßig.

Sind beispielsweise neue Mitarbeiter anzumelden, dann kann dies nur der Systemadministrator tun, der aber auch mal Urlaub machen möchte. Hier gibt es dann prinzipiell drei Möglichkeiten:

- Der Systemadministrator gibt sein Paßwort an andere Mitabeiter weiter, die dann ihrerseits Mitarbeiter anmelden können. Dieses Vorgehen ist nun aber nicht ganz ungefährlich, zumal man kaum noch die Übersicht behalten kann, wer denn nun im Besitz des Paßwortes ist.
- Mitarbeiter geben neuen Mitarbeiter zunächst den eigenen Namen und das eigene Paßwort weiter. Dies ist auch nicht ganz unproblematisch, weil diese damit auf die gleichen Tabellen dieselben Zugriffsrechte haben wie der eigentliche Inhaber des Benutzernamens.
- Der Systemadministrator legt einige Benutzer auf Vorrat an.

Datensicherheit

In diesem Zusammenhang noch ein Wort zum Thema Datensicherheit. Man könnte der Ansicht sein, daß die Daten auf dem *Local InterBase Server* vor unbefugtem Zugriff geschützt sind. Dies ist aber gerade nicht der Fall, bestehen doch sogar zwei Möglichkeiten, den vollständigen Zugriff auf alle Daten zu erhalten:

- Das simpelste ist, die entsprechende *.*gdb*-Datei auf Diskette zu kopieren, auf den eigenen Rechner zu laden und dann beispielsweise mit *InterBase Interactive SQL* zu betrachten. Angemeldet wird hier mit *SYSDBA* und *masterkey* (oder welches Paßwort auf dem eigenen Rechner auch immer vorhanden ist).

- Mit dem *InterBase Server Manager* wird ein Backup auf Disketten gezogen und dieses dann auf den eigenen Rechner gespielt. Wählt man die Option *Transportable Format*, dann wird auch gleich noch die Dateigröße verringert.

Geht man mit sensiblen Daten um, dann ergeben sich daraus folgende Konsequenzen:

- Der Zugriff auf die **.gdb*-Datei muß gesperrt werden, beispielsweise, indem man Windows NT verwendet und das ganze Verzeichnis sperrt.
- Zweitens muß der Zugriff auf den *InterBase Server Manager* unterbunden werden. Dies könnte auch so geschehen, daß nur der SYSDBA angemeldet ist, und die Applikation sich sozusagen unsichtbar einloggt (wir besprechen nachher, wie das funktioniert).

  Die Anwender loggen sich dann mit anderen Namen bei der Applikation ein und erhalten somit mit ihren Paßwörtern nie Zugriff auf den *Server Manager*. Zum Ziehen eines Backups kann dann die Komponente *TBatchMove* verwendet werden.

13.2.3 Backups

Die EDV und insbesondere Datenbanken sind häufig das organisatorische Rückgrat einer Firma; fällt die EDV aus, beginnt das große Chaos. Von daher wäre ein Verlust des Datenbestandes gleich doppelt problematisch: Zum einen würde die manuelle Neueingabe Kosten von mindestens einigen Tausend DM verursachen, zum anderen würden die »Produktionsausfallkosten« noch einmal mindestens die gleiche Summe ausmachen.

Von daher ist es selbstverständlich, daß von der Datenbank laufend Backups gezogen werden. Bei Client-Server-Datenbanken geht das nicht mehr mit der Methode *einer bleibt Freitags eine halbe Stunde länger und zieht das Backup*, vor allem deshalb nicht, weil dieses Intervall viel zu grobmaschig gewählt wäre: Nehmen wir einmal an, daß fünf Personen mit der Eingabe von Daten beschäftigt sind, und in der Mitte der Woche das System aussteigt. Für die Neueingabe einer halben Woche könnte man Kosten von rund 3000,– DM veranschlagen, die Produktionsausfallskosten sind dabei noch nicht einmal eingerechnet.

Es wird sich somit nicht vermeiden zu lassen, je nach Anzahl der Transaktionen bis zu mehrmals täglich Backups zu erstellen. Hier sind dann die üblichen Backup-Programme relativ unbrauchbar, weil diese im laufenden Betrieb nicht auf die Dateien zugreifen können. Andererseits ist es nicht zumutbar, mehrmals täglich den Betrieb lahmzulegen, weil ein Backup gezogen werden soll. Das Backup muß folglich vom Datenbank-Server während des laufenden Betriebs erstellt werden.

Hier sind wir gleich beim nächsten Problem: Die Backups müssen bei InterBase manuell erstellt werden, der Systemadministrator möchte aber auch mal Urlaub

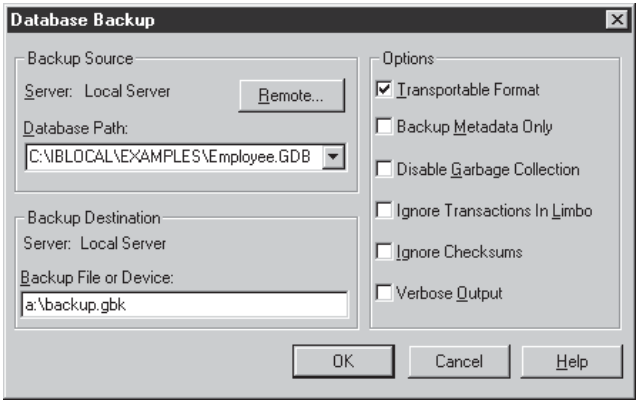


Bild 13.4: InterBase Backup

machen oder ist sonstwie verhindert. Auf der anderen Seite muß die Zahl der Personen, die auf den *InterBase Server Manager* zugreifen können, minimiert werden, weil diese Personen an den vollständigen Datenbestand gelangen können.

Wie dies nun firmenintern gelöst werden kann, soll hier nicht weiter erörtert werden. Es sollte jedoch nicht versäumt werden, auf diese Probleme hinzuweisen (Hans-Joachim von Ziethen, preußischer General: *Eine Gefahr, die man kennt, ist keine Gefahr mehr.*)

Des weiteren besteht bei *InterBase* die Möglichkeit, einen SHADOW zu definieren. Dies ist im Prinzip eine zweite Datenbankdatei, welche jederzeit eine exakte Kopie der eigentlichen Datei ist. Dieser SHADOW wird zweckmäßigerweise auf einer anderem Festplatte betrieben. Im Schadensfall kann man dann mit dem SHADOW weiterarbeiten.

Ein Backup ziehen

Um ein Backup zu erstellen, wählt man TASKS | BACKUP und öffnet damit ein Fenster nach Bild 13.4. Unter *Backup Source* wählt man die Datenbank, von der ein Backup gezogen werden soll, bei *Backup Destination* wird angegeben, in welche Datei das Backup geschrieben werden soll.

Bei größeren Datenbanken wird man wohl kaum ein Diskettenlaufwerk verwenden wollen, dies würde einfach zu lange dauern. Auf der anderen Seite macht es keinen Sinn, die Backups laufend auf die eigene Festplatte zu spielen: Im Falle eines Defektes der Festplatte sind auch diese Daten meist nicht mehr verfügbar. (Es gibt zwar Spezialisten, die auch hier die Daten meist noch retten können, aber ein Backup, auf das man drei Wochen warten muß, ist eine sehr schlechte Lösung.)

Aus Gründen der Arbeitsvereinfachung kann es eine Lösung sein, die Backups auf die Festplatte eines anderen Rechners zu spielen und diese Dateien dann auf einem anderen Medium (Streamer-Tape, CD, usw.) zu speichern. Bedenken Sie auch, daß im Falle von Wasser, Feuer oder sonstiger Katastrophen auch noch Kopien an anderer Stelle lagern sollten (beispielsweise beim Chef daheim im Tresor). Das muß nicht die aktuellste Version sein, »Antiquitäten« haben hier aber auch nichts zu suchen.

Beim Erstellen des Backups können einige Optionen gewählt werden:

- *Transportable Format:* Erstellt das Backup in einem vom Betriebssystem unabhängigen Format. *InterBase* läuft auf mehreren Plattformen, so daß auf diese Weise Daten ausgetauscht werden können.
- *Backup Metadata Only:* Bisweilen soll die Struktur einer Datenbank (Tabellendefinition, VIEWs, STORED PROCEDUREs, usw.) weitergegeben werden, nicht jedoch die Daten selbst (Anwendungsbeispiel: eine neue Filiale wird eröffnet); dies wird mit dieser Option bewerkstelligt.
- *Disable Garbage Collection:* Für gewöhnlich hat sich in einer Datenbank auch noch etwas »Datenmüll« angesammelt (von abgebrochenen Transaktionen beispielsweise). Diese Daten werden beim Erstellen eines Backups automatisch gelöscht. Soll dies verhindert werden, muß diese Option eingesetzt werden.
- *Ignore Transaktion in Limbo:* Tritt bei einer Transaktion ein Fehler auf, dann kann es vorkommen, daß einige Datensätze geschrieben, aber noch nicht mit *Commit* bestätigt worden sind. Diese Datensätze lassen sich mit MAINTENANCE | TRANSACTION RECOVERY manuell behandeln (*Commit* oder *Rollback*) und werden normalerweise auch bei einem Backup gesichert. Die Option *Ignore Transaktion in Limbo* vermeidet dies.
- *Ignore Checksums:* InterBase bildet von jeder »Seite« auf einer Festplatte eine Prüfsumme. Sobald diese nicht mehr stimmt, ist auf dieser Seite ein Fehler aufgetreten (System-Crash, ein »Spezialist« hat die Datenbank-Datei mit dem Texteditor bearbeitet, usw.). Normalerweise wird verhindert, daß von einer solchen Datenbank ein Backup gezogen wird. Mit dieser Option läßt sich das umgehen.
- Mit *Verbose Output* werden alle Backup-Schritte in ein Textfenster geschrieben. Auf diese Weise läßt sich feststellen, welche Schritte durchgeführt werden und wo ggf. ein Fehler aufgetreten ist. (Sie werden dabei sehr gründlich informiert: Bei der Datenbank *employee.gdb* werden dabei 546 Zeilen ausgegeben.)

Ein Backup zurückspielen

Backups nützen herzlich wenig, wenn sie sich nicht zurückspielen lassen. Es wäre denkbar, daß man dies auch im laufenden Betrieb machen könnte, dies würde

aber einige Probleme mit sich bringen. Um ein Backup zurückspielen zu können, müssen schließlich alle Verbindungen zu dieser Datenbank geschlossen werden, auch die des *InterBase Server Managers*.

Was beim *Local InterBase Server* recht einfach ist, kann beim *InterBase Workgroup Server* zum ernsten Problem werden, schließlich hat der Systemadministrator in diesem Augenblick wirklich anderes zu tun als durch die Abteilungen zu marschieren und die Clients abzumelden. Diesem Zweck dient eine extra Funktion, auf die wir später zurückkommen werden.

Um ein Backup zurückzuspielen, wählt man TASKS | RESTORE, gibt das Backup-File sowie das Datenbank-File ein (leider hat man gerade hier von Browse-Buttons nichts gehalten) und wählt die Optionen:

- Bei InterBase ist es möglich, die Datenbank in mehrere Dateien aufzuteilen, was allerdings nur bei sehr großen Datenbanken Sinn macht. Hier wäre es dann beispielsweise möglich, diese Dateien auf mehreren Festplatten unterzubringen, was den Zugriff beschleunigen kann. Näheres dazu in der Online-Hilfe.
- Der Zugriff kann auch beschleunigt werden, wenn der Parameter *Page Size* optimal gewählt wird. Auch hierzu sei auf Handbücher und Online-Hilfe verwiesen.
- *Replace Existing Database:* Normalerweise überschreibt der Server-Manager keine existierende Datenbank. Der Systemadministrator sollte also die vorhandene Datenbank umbenennen oder löschen und erst dann das Backup durchführen. Mit der Option *Replace Existing Database* kann man diese Sicherung umgehen (beispielsweise wenn es extrem eilt).
- *Commit after Each Table:* Für gewöhnlich werden zunächst alle Metadaten und erst dann die Daten geschrieben. Ist nun auch das Backup fehlerhaft, dann kann der Fall eintreten, daß nur ein Teil der Metadaten wiederhergestellt werden kann.

  Mit der Option *Restore after Each Table* würde dann eine komplette Tabelle nach der anderen wiederhergestellt, bis der Fehler auftritt. Danach gilt auch hier: »Rien ne va plus«.
- *Deactivate Indexes:* Sind Indizes vorübergehend inaktiv, dann können regelwidrige Datensätze in die Datenbank gelangen (beispielsweise doppelte Index-Einträge). Da bei Wiederherstellen zunächst die Metadaten geschrieben werden, würde erst beim Schreiben der Daten ein Fehler gemeldet und der *Restore*-Vorgang abgebrochen.

  Nun ist es im Ernstfall meist wichtiger, die Daten überhaupt wiederherzustellen, so daß mit dieser Option auf sämtliche Indizes verzichtet werden kann. Diese müssen dann manuell erstellt werden, danach sollte man tunlichst nach (alten und neuen) Regelverletzungen suchen und diese beheben.

- *Do not restore Validity Conditions:* Ein ähnliches Problem kann bei Gültigkeitsprüfungen aller Art auftreten. Dazu gleich ein anschauliches Beispiel: die Mitgliederverwaltung eines Vereins. Diese wurde auf der Grundlage der alten Adressendatei aufgebaut, welche nur die Felder für Adresse kannte. Die daraus erstellte Mitgliederdatenbank kennt nun noch einige weitere Felder, beispielsweise das Eintrittsdatum, und damit das nicht aus lauter Schlamperei unterschlagen wird, wurde dieses Feld als NOT NULL deklariert.

  Alle neuen Mitglieder werden nun mit Eintrittsdatum erfaßt, bei den alten wird dieses nach und nach getan (wenn man Zeit hat, meist hat man keine...). Versucht man nun ein Backup zurückzuspielen, dann werden die alten Daten nicht mehr den geänderten Gültigkeitsprüfungen genügen, was Probleme verursachen könnte (der Konjunktiv ist hier angebracht: Bei einem entsprechenden Versuch sind auch alle alten Datensätze kommentarlos akzeptiert worden).

  Auf jeden Fall werden mit der Option *Do not restore Validity Conditions* alle Gültigkeitsprüfungen nicht übernommen.

- Schließlich gibt es auch hier wieder die schon beschriebene Funktion *Verbose Output*.

13.2.4 Statistiken

Unter dem Menütitel *Tasks* finden sich die Menüpunkte TASKS | DATABASE STATISTIKS und TASKS | LOCK MANAGER STATISTICS. Diese Funktionen sind beide beim *Local Interbase Server* nicht verfügbar und deshalb grau dargestellt. Beim *InterBase Workgroup Server* rufen sie Statistiken über den Zustand der Datenbank auf, damit diese optimiert werden kann.

Mit TASKS | INTERACTIVE SQL kann das gleichnamige Programm gestartet werden, das sozusagen die »Datenbankoberfläche für InterBase« ist (allerdings längst nicht so komfortabel).

13.2.5 Das Maintenance-Menü

Unter dem Menü-Titel *Maintenance* sind einige weitere Funktionen verfügbar, welche allerdings nur zum Teil für den LIBS verfügbar sind.

Die Verbindungen

Mit MAINTENANCE | DATABASE CONNECTION öffnet man ein Fenster, in dem alle bestehenden Verbindungen zur entsprechenden Datenbank angezeigt werden (dazu muß eine Datenbank explizit ausgewählt werden, es reicht nicht aus, mit dem Server verbunden zu sein).

Wenn Sie beispielsweise ein Backup zurückspielen wollen, dann müssen alle bestehenden Verbindungen beendet werden. Hier kann diese Funktion hilfreich sein, beispielsweise wenn Sie dadurch feststellen, daß eine C++Builder-Applikation meist zweimal verbunden ist (einmal C++Builder für die Live-Daten, zum zweiten die Applikation, sobald sie gestartet wird).

Beachten Sie bitte auch in diesem Zusammenhang, daß die maximale Zahl der Verbindungen inzwischen auf zwei beschränkt wurde. Wenn Sie beispielsweise zusätzlich noch *Interactive SQL* laufen haben, dann kann eine C++Builder-Anwendung nicht mehr gestartet werden.

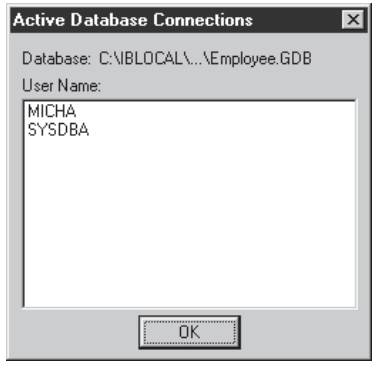


Bild 13.5: Aktive Verbindungen zu einer Datenbank

Eine Datenbank »ausfegen«

Die Übersetzung von MAINTENANCE | DATABASE SWEEP klingt eher komisch, aber genau das tut diese Funktion. Um zu verstehen, was das ganze soll, muß ein bischen tiefer in die Thematik eingestiegen werden:

Stellen Sie sich eine Datenbanktabelle als eine große Tabelle vor. Wird hier ein Datensatz gelöscht, dann entsteht an dieser Stelle eine Lücke. Man könnte nun alle nachfolgenden Datensätze um jeweils einen Platz nach vorne rücken, aber das ist zeitaufwendig, also läßt man es bleiben. Dasselbe passiert beim Ändern eines Datensatzes: Hier wird zunächst der neue Datensatz hinten angehängt (man weiß ja nicht, ob der Anwender doch noch ein ROLLBACK-Kommando absetzt), sobald mit COMMIT bestätigt wird, wird der ursprüngliche Datensatz gelöscht. (Es spielt dabei keine Rolle, ob die Datensätze wirklich gelöscht werden oder ob sie bloß nicht mehr verwendet werden).

Im Laufe der Zeit wird die Datenbank immer länger und immer lückenhafter, dann wird es Zeit für einen »Hausputz«. Mit dem Befehl MAINTENANCE | DATABASE SWEEP werden nun alle Tabellen überarbeitet und die entsprechenden Lücken gefüllt; die Tabellen werden folglich kürzer.

Bei den Datenbankeigenschaften (besprechen wir gleich) kann ein Sweeping-Intervall eingestellt werden. Alle 20 000 Transaktionen (oder wie der Wert auch immer eingestellt sein mag) erscheint ein Fenster auf dem Bildschirm, das zu dieser Aktion anregt. Der Anwender kann dies auch ablehnen, beispielsweise deshalb, weil der Server im Moment ohnehin genug belastet ist.

Transaktionen abschließen

Damit Sie den nächsten Punkt verstehen, muß ein wenig auf das Thema *Transaktionen* vorgegriffen werden: Es gibt Situationen, in denen zwei Aktionen entweder gemeinsam durchgeführt oder gemeinsam abgebrochen werden müssen. Nehmen wir als Beispiel eine Überweisung: Entweder soll das eine Konto belastet und dem anderen eine entsprechende Gutschrift erteilt werden, oder es soll bei beiden Konten unterbleiben. Auf jeden Fall sollte nicht die Situation auftreten, daß dem einen Konto eine Gutschrift erteilt wird, dem anderen mangels Dekkung oder Überschreitung der Kreditlinie jedoch nichts abgezogen wird.

Die Applikationen behandeln dies in der Regel als zwei Aktionen, welche einzeln durchgeführt werden. Sobald das Geld auf dem Konto ist, könnte man nur noch eine Storno-Buchung machen (welche vielleicht ebenfalls mangels Deckung nicht ausgeführt werden kann).

Das Problem löst man mit einer Transaktion. Vor den beiden Update-Befehlen (oder Insert-Befehlen, je nach dem) wird mit START TRANSAKTION eine sogenannte Transaktion gestartet. Erfolgt bei allen (es können auch mehrere Befehle zusammengefaßt werden) Befehlen kein Fehler, dann wird die Änderung mit COMMIT bestätigt, andernfalls werden beide Befehle mit ROLLBACK zurückgenommen.

Nun kann der unschöne Fall eintreten, daß eine Transaktion begonnen, aber weder mit COMMIT noch mit ROLLBACK abgeschlossen wird (die Appliaktion hängt sich auf, der Chef stolpert über das Netzkabel, usw.). Auf diese Weise sammeln sich nach und nach die Transaktionen, welche auf ihren Abschluß warten. Solange diese Transaktion lediglich auf eine Datenbank zugreift, ist dies nicht weiter problematisch, InterBase macht in diesem Fall selbst die Transaktion rückgängig.

Richtig kompliziert wird es, wenn eine Transaktion mehrere Datenbanken umfasst: In diesem Fall wird die Bestätigung mit COMMIT in zwei Phasen durchgeführt. Wenn hier nun ein Fehler auftritt, entsteht eine *limbo transaction*, hier weiß nun InterBase nicht mehr weiter. (So etwas ist recht selten: Ich habe zwei Stunden lang versucht, so etwas zu provozieren, um Ihnen den entsprechenden ScreenShot präsentieren zu können – es ist beim Versuch geblieben).

Auf jeden Fall können Sie mit MAINTENANCE | TRANSACTION RECOVERY ein Fenster öffnen (leider hier ohne Bild...), mit dem Sie die entsprechenden Transaktionen einzelnen bestätigen oder verwerfen sowie auch einzelne Sub-Transaktionen lö-

schen können. Normalerweise werden Sie beim Aufruf dieses Fensters nur die Mitteilung *No pending transactions were found* erhalten – für Sie ein Grund zur Freude.

13.2.6 Eine Datenbank herunterfahren

Nehmen wir einmal an, irgendwo würde ein Fehler auftreten, und es müßte ein Backup auf die Datenbank gespielt werden; wie vorher bereits erwähnt, müssen dafür alle Verbindungen zu dieser Datenbank unterbrochen sein. Bei größeren Systemen, bei denen dann die Clients über mehrere Räume verteilt sind, kann dies recht aufwendig werden, und schließlich hat der Systemadministrator auch etwas anderes zu tun als persönlich alle Clients abzumelden (und gemäß dem *Gesetz der größten Gemeinheit* haben sich dann zwischenzeitlich wieder drei Benutzer angemeldet...). Es ist also nötig, alle Benutzer zentral vom Server aus abzumelden. Hier wird MAINTENANCE | DATABASE SHUTDOWN gewählt, es stehen dann drei Möglichkeiten zur Verfügung, siehe Bild 13.5:

- *Deny New Connections while waiting:* In der angegebenen Zeit können keine neuen Verbindungen zur Datenbank mehr hergestellt werden, alle bestehenden Verbindungen bleiben bestehen.
- *Deny New Transactions while waiting:* In der angegebenen Zeit können keine neuen Transaktionen mehr durchgeführt (und keine neuen Verbindungen hergestellt) werden, bestehende Transaktionen werden nicht beeinträchtigt.

Beiden Optionen ist gemeinsam, daß die Datenbank geschlossen wird, sobald keine Verbindungen bzw. Transaktionen mehr bestehen. Nach Ablauf der Zeit unterbleibt des Versuch, die Datenbank herunterzufahren, wenn noch Verbindungen bzw. Transaktionen bestehen.

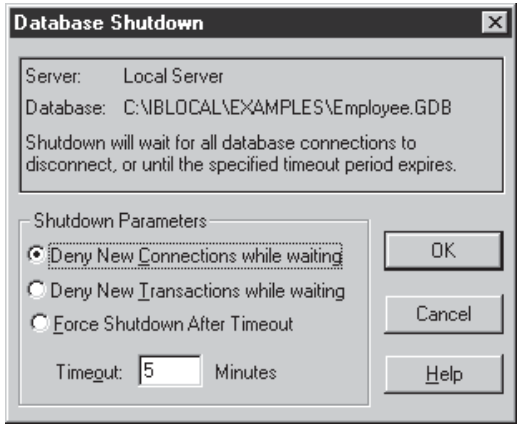


Bild 13.6: Server herunterfahren

- *Force Shutdown After Timeout*, die »Holzhammermethode«*:* Die Clients haben innerhalb der angegebenen Zeit die Chance, Ihre Arbeit zu beenden. Sobald alle Transaktionen beendet sind, wird die Datenbank heruntergefahren. Sind in der angegebenen Zeit nicht alle Transaktionen beendet, dann sendet der Server-Manager den ROLLBACK-Befehl und beendet alle Verbindungen.

Mit MAINTENANCE | DATABASE RESTART kann die Datenbank anschließend wieder gestartet werden.

13.2.7 Eine Datenbank reparieren

Im laufenden Betrieb kommt es immer mal wieder vor, daß Teile der Datenbank Schaden nehmen. Mit MAINTENANCE | DATABASE VALIDATION wird ein Fenster geöffnet, das einige Funktionen zum Reparieren der Datenbank bietet. Beachten Sie, daß dafür alle anderen Verbindungen zur Datenbank geschlossen sein müssen.

Haben Sie den Verdacht, daß mit der Datenbank etwas nicht stimmen könnte, dann rufen Sie diese Funktion auf, und wählen Sie zunächst die Option *Read-only validation* – damit wird sichergestellt, daß keine Änderungen durchgeführt werden. Stellt diese Funktion Fehler in der Datenbank fest, dann werden diese aufgelistet, andernfalls erscheint die Meldung *No Database validation errors were detectet.*

13.2.8 Eigenschaften anzeigen

Mit MAINTENANCE | DATABASE PROPERTIES wird ein Fenster nach Bild 13.7 geöffnet, in dem einige Eigenschaften der Datenbank angezeigt werden. Manche lassen sich auch verändern:

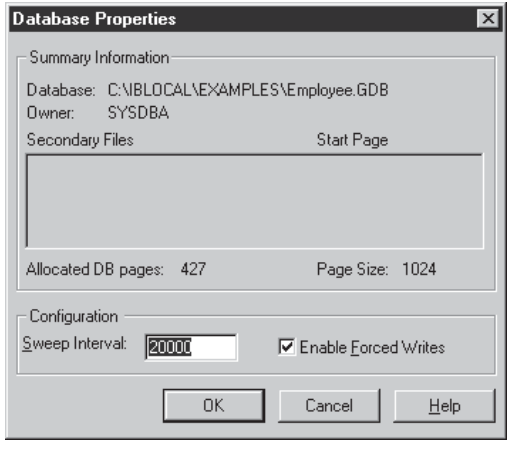


Bild 13.7: Die Eigenschaften der Datenbank

- Zunächst werden die Datei und der Benutzer angezeigt, welcher die Datenbank angelegt hat.
- Ist die Datenbank auf mehrere Dateien verteilt, dann werden diese ebenfalls aufgelistet.
- Des weiteren wird angezeigt, wieviel Seiten die Datenbank belegt und wie groß die einzelne Seite ist. In Bild 13.7 sind 427 Seiten von je 1024 Bytes belegt, das ergibt eine Dateigröße von 427 KByte (was auch der Dateimanager bestätigt).
- Ein weiteres Anzeigefeld legt fest, daß alle 20 000 Transaktionen (läßt sich ändern) eine MessageBox erscheint, die zum »Ausfegen« der Datenbank animiert (siehe Kapitel 13.2.5).
- Das Thema *Forced Writes* ist nur bei einigen Servern von Belang, es soll deshalb hier nicht näher darauf eingegangen werden.

13.3 Interactive SQL

Das Programm *Interactive SQL* (ISQL) ist ein Tool zum Zugriff auf *InterBase*-Datenbanken per SQL-Anweisung. Das Programm ist recht spartanisch ausgelegt, fast alle Befehle müssen manuell als SQL-Anweisung geschrieben werden. Dies zwingt dazu, sich intensiv mit SQL auseinanderzusetzen, was wiederum auch seine Vorteile hat.

Bild 13.8 zeigt die Oberfläche von ISQL. Im Memo *SQL-Statement* wird ein SQL-Text eingegeben, dann wird der Button *Run* betätigt, und im Memo *ISQL Output* wird dann das Ergebnis angezeigt (wenn es sich um Abfragen handelt) oder die SQL-Anweisung wiederholt (bei anderen SQL-Befehlen). Oft wird man auch mit einer Fehlermeldung konfrontiert, wenn man sich beispielsweise nicht an die Syntax der SQL-Anweisungen gehalten hat.

Vielfach wird man SQL-Anweisungen stufenweise entwickeln oder die gleichen Befehle mehrmals verwenden. ISQL speichert alle Anweisungen, die im Laufe einer Sitzung gegeben wurden. Mit den Buttons *Previous* und *Next* kann in diesen Anweisungen geblättert werden.

Die Texte, die im Memo *ISQL Output* angezeigt werden, können mit dem Button *Save Results* als Text-Datei gespeichert werden. Dies kann beispielsweise dann ganz nützlich sein, wenn die Daten in eine Textverarbeitung importiert werden sollen und in der eigentlichen Datenbankapplikation eine entsprechende Funktion nicht zur Verfügung steht. (Noch ein Tip: Vergessen Sie die Endung *txt* nicht, da sie im Gegensatz zu den C++Builder-Dialogen nicht automatisch angehängt wird.)

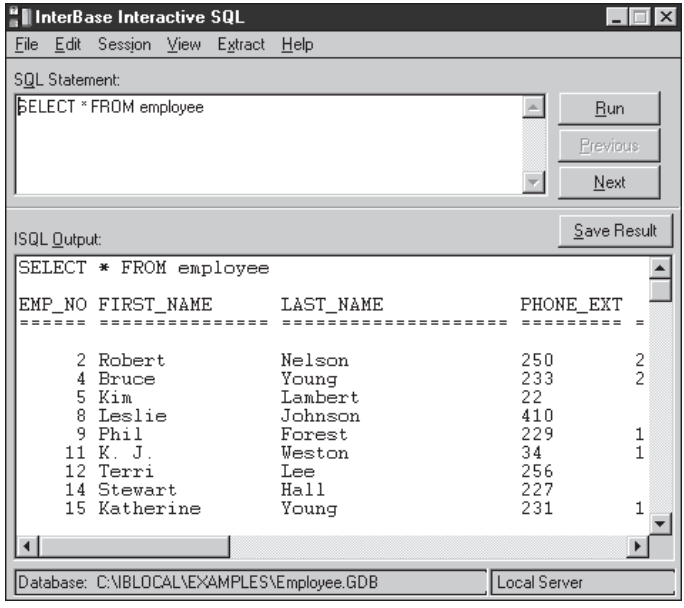


Bild 13.8: Interactive SQL

13.3.1 Mit einer Datenbank verbinden

Bevor mit ISQL gearbeitet werden kann, muß zuerst die Verbindung mit einer Datenbank hergestellt werden. Dazu wählt man FILE | CONNECT TO DATABASE, worauf sich ein Fenster nach Bild 13.3 öffnet. Hier wählt man dann die Datenbank und gibt Usernamen und Paßwort ein.

Besteht beim Öffnen dieses Fensters bereits eine Verbindung zu einer Datenbank, dann erscheint ein Message-Dialog *Commit Work for Database?*.

Verbindung zu einer Datenbank lösen

Mit FILE | DISCONNECT FROM DATABASE können Sie die Verbindung zu einer Datenbank lösen. In der Regel werden Sie den Befehl nicht benötigen, da die Verbindung beim Verlassen des Programms oder beim Öffnen des Fenster *Database Connect* gelöst wird.

Bei manchen Aktionen des Server-Managers ist es aber nötig, daß keine weiteren Verbindungen zur Datenbank bestehen. Soll hier nicht gleich das ganze Programm beendet werden, muß die Verbindung manuell gelöst werden.

Eine neue Datenbank erstellen

Um eine neue Datenbank zu erstellen, kann der SQL-Befehl CREATE DATABASE verwendet werden. Hier bietet ISQL ein wenig Unterstützung, kann doch mit FILE | CREATE DATABASE ein Fenster nach Bild 13.9 geöffnet werden.

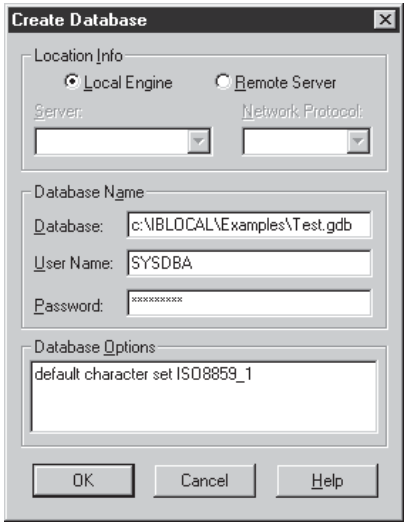


Bild 13.9: Eine Datenbank erstellen

Zunächst muß angegeben werden, auf welchem Server die Datenbank erstellt werden soll, in diesem Fall auf dem LIBS. Des weiteren werden der Name der Datenbank (Pfad!), ihr Besitzer und dessen Paßwort abgefragt. Das Paßwort wird dabei nur einmal abgefragt, Sie sollten sich deshalb merken, was Sie hier eingeben (als SYSDBA kommt man aber ohnehin in jede Datenbank auf dem Server).

Des weiteren können einige *Database Options* eingegeben werden; in Bild 13.8 wird der Zeichensatz ISO 8859_1 eingestellt (dies ist weniger zu empfehlen, weil ISQL dann die Eingabe von Umlauten verweigert). Zu den weiteren Details bei der Erstellung von Datenbanken kommen wir später.

Wenn Sie die gerade erstellte Datei einmal mit dem Dateimanager betrachten, dann bekommen Sie keinen Schrecken ob der Dateigröße: Ohne eine einzige Tabelle und ohne einen Datensatz benötigt diese Datenbank schon einmal 202 kByte. Zu Ihrer Beruhigung: Beim Erstellen von Tabellen und bei der Eingabe von Daten wächst die Dateigröße dann normal an (also in dem Umfang, indem es die verwendeten Datentypen nun einmal erzwingen).

Würden Sie jetzt von der Datenbank ein Backup ziehen, dann würde dessen Dateigröße auch nur 1 kByte betragen. Sie können sich ja einmal interessehalber die Dateien der Datenbank und des Backups mit dem Texteditor ansehen.

Eine Datenbank löschen

Mit File | Drop Database können Sie die bestehende Datenbank löschen. Passen Sie ja auf, daß Sie diesen Befehl nicht versehentlich geben, es wird zwar noch eine Sicherheitsabfrage getätigt, aber die sieht so harmlos aus wie die beim Lösen der Verbindung zu einer Datenbank (das ist der Befehl direkt darüber).

13.3.2 Ein ISQL-Script verwenden

Bisweilen sollen umfangreichere Änderungen auf mehreren Servern gleich durchgeführt werden, zum Beispiel dann, wenn die Datenstruktur bei allen Filialen einer Firma geändert werden soll. Die Datenbank-Applikation sieht für solche Fälle sicher keine passende Funktion vor, und alles manuell mit ISQL zu ändern, ist ein recht mühseliges Unterfangen.

Hier besteht nun die Möglichkeit, ein ISQL-Script zu erstellen, also eine Reihe von SQL-Anweisungen, welche in einer externen Textdatei aufgelistet sind. Mit File | Run an ISQL Script können diese Befehle zur Ausführung gebracht werden (in unserem Beispiel würde der Systembeauftragte mit einer entsprechenden Diskette umherziehen und dem Server gelassen bei der Arbeit zusehen).

Ein ISQL-Script beinhaltet eine abgeschlossene Transaktion. Bevor es gestartet wird, fragt ISQL an, ob die vorherige Transaktion mit COMMIT abgeschlossen oder mit ROLLBACK zurückgenommen werden soll. Jedes ISQL-Script muß dann mit einem CONNECT-Befehl (oder einem CREATE DATABASE-Befehl) beginnen. Auf dem LIBS könnte dies wie folgt aussehen (Achtung: Das Paßwort steht in der Datei, somit muß diese entsprechend geschützt werden):

```
CONNECT "C:\IBLOCAL\EXAMPLES\EMPLOYEE.GDB"
USER "SYSDBA" PASSWORD "masterkey";
```

Mit dem Befehl File | Save Session to a file kann eine ISQL-Sitzung protokolliert werden, so daß sie sich später automatisch (und exakt) reproduzieren läßt. Mit diesem Befehl werden alle Befehle seit der letzten Aufzeichnung abgespeichert. Vor dem Beginn der eigentlichen Aktionen sollte man daher die bisherige Sitzung speichern, um diese Anweisungen damit aus der Liste zu löschen (alternativ ISQL neu starten).

Bild 13.9 zeigt ein solches (wenig umfangreiches) ISQL-Script: Zunächst wurde eine neuen Datenbank erstellt (mit den Parametern gemäß Bild 13.8). Mit dem Befehl *Extract Database* wurde nachher das Ergebnis betrachtet; dies macht in einem ISQL-Script keinen Sinn, deshalb hat InterBase diese Anweisung gleich zu einem Kommentar umgewandelt (mit den /* */-Zeichen). Des weiteren wurde eine Tabelle mit zwei Spalten definiert und in diese Tabelle ein Datensatz eingefügt. Anschließend wurde der Inhalt der Tabelle betrachtet.

Noch ein Hinweis: Eigentlich muß jede SQL-Anweisung mit einem Semikolon (Strichpunkt) abgeschlossen werden. Bei einzelnen Anweisungen bleibt das Weglassen ohne Konsequenzen, aber ein daraus gewonnenes ISQL-Script wird nicht mehr wie gewünscht funktionieren, bis man die Semikola nach jeder Anweisung von Hand einfügt.

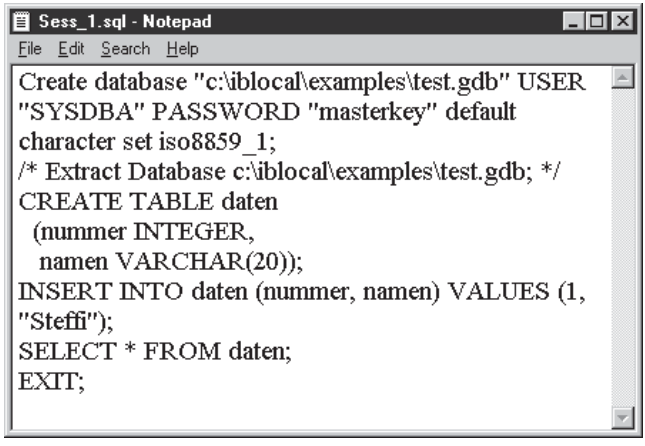


Bild 13.9: Ein ISQL-Script

13.3.3 Explizite Transaktionskontrolle

Bei jeder neuen Verbindung zu einer Datenbank wird eine neue Transaktion begonnen, die für gewöhnlich immer dann mit COMMIT abgeschlossen wird, wenn die Verbindung zu dieser Datenbank gelöst wird (explizit oder beispielsweise auch beim Verlassen des Programms).

Bisweilen möchte man schon während der Sitzung die bisherige Arbeit bestätigen oder zurücknehmen. Dazu dienen die Menüpunkt FILE | COMMIT WORK und FILE | ROLLBACK WORK.

Befehle der DDL (Datendefinitionssprache, also beispielsweise CREATE TABLE) werden sofort mit COMMIT bestätigt, falls die Option SESSION | BASIC SETTINGS | AUTO COMMIT DDL gesetzt ist.

13.3.4 Metadaten anzeigen

Unter den Metadaten einer Datenbank versteht man die Definition von Tabellen, DOMAINs, VIEWs, usw. Wenn Sie mit einer unbekannten Datenbank konfrontiert werden, dann müssen Sie zunächst herausfinden, wie die Datenstruktur dieser Datenbank aussieht. (Solange Sie beispielsweise keine Tabellen- und Viewnamen kennen, können Sie noch nicht einmal einen SELECT-Befehl richtig formulieren.)

Unter den Menü-Titeln VIEW und EXTRACT stellt ISQL einige Funktionen zum Ermitteln der Metadaten zur Verfügung:

Mit EXTRACT | SQL METADATA FOR DATABASE werden alle Anweisungen zur Definition dieser Datenbank zurückgegeben. Das Ergebnis läßt sich wahlweise im Fenster *ISQL Output* anzeigen oder in eine Datei speichern. Diese Datei läßt sich dann als ISQL-Script verwenden, um die Metadaten dieser Datenbank auf einen anderen Server zu spielen (die Alternative wäre, ein Backup der Metadaten zu ziehen, siehe Kapitel 13.2.3).

Nachfolgend wollte ich eigentlich die Metadaten der Datenbank *Employee.gdb* aufführen, das hätte aber (bei einer Schriftgröße von 6 Punkt) mehr als acht Seiten belegt. Damit Sie trotzdem eine Ahnung davon bekommen, was in so einer Datenbank an Metadaten steckt, wird diese Datei stark gekürzt wiedergegeben:

```
/* Extract Database C:\IBLOCAL\EXAMPLES\Employee.GDB */
CREATE DATABASE "C:\IBLOCAL\EXAMPLES\Employee.GDB" PAGE_SIZE 1024;

/* Domain definitions */
CREATE DOMAIN FIRSTNAME AS VARCHAR(15);

CREATE DOMAIN DEPTNO AS CHAR(3)
     CHECK (VALUE = '000' OR (VALUE > '0' AND VALUE <= '999')
       OR VALUE IS NULL);

/* Table: CUSTOMER, Owner: SYSDBA */
CREATE TABLE CUSTOMER
     (CUST_NO CUSTNO NOT NULL,
     CUSTOMER VARCHAR(25) NOT NULL,
     CONTACT_FIRST FIRSTNAME,
     CONTACT_LAST LASTNAME,
     PHONE_NO PHONENUMBER,
     ADDRESS_LINE1 ADDRESSLINE,
     ADDRESS_LINE2 ADDRESSLINE,
     CITY VARCHAR(25),
     STATE_PROVINCE VARCHAR(15),
     COUNTRY COUNTRYNAME,
     POSTAL_CODE VARCHAR(12),
     ON_HOLD CHAR(1) DEFAULT NULL,
   PRIMARY KEY (CUST_NO));

ALTER TABLE JOB ADD FOREIGN KEY (JOB_COUNTRY)
   REFERENCES COUNTRY(COUNTRY);

CREATE GENERATOR EMP_NO_GEN;
```

```
/* View: PHONE_LIST, Owner: SYSDBA */
CREATE VIEW PHONE_LIST (EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT,
  LOCATION, PHONE_NO) AS SELECT
    emp_no, first_name, last_name, phone_ext, location, phone_no
  FROM employee, department
  WHERE employee.dept_no = department.dept_no;

/*  Exceptions */
CREATE EXCEPTION UNKNOWN_EMP_ID
  "Invalid employee number or project id.";

/* Stored procedures */
CREATE PROCEDURE GET_EMP_PROJ AS BEGIN EXIT; END ^

ALTER PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT proj_id
  FROM employee_project
  WHERE emp_no = :emp_no
  INTO :proj_id
  DO
  SUSPEND;
END
 ^

CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
  ACTIVE AFTER UPDATE POSITION 0
AS
BEGIN
  IF (old.salary <> new.salary) THEN
    INSERT INTO salary_history
      (emp_no, change_date, updater_id, old_salary, percent_change)
    VALUES (old.emp_no, 'now', user, old.salary,
      (new.salary - old.salary) * 100 / old.salary);
END
 ^

/* Grant permissions for this database */
GRANT DELETE, INSERT, SELECT, UPDATE ON COUNTRY TO PUBLIC
  WITH GRANT OPTION;
```

Am Beginn steht (nicht ganz unerwartet) die Erstellung der Datenbank, danach werden insgesamt 15 DOMAINS erstellt, von denen hier zwei vorgestellt werden. Die DOMAIN *Firstname* legt einfach nur einen String von 15 Zeichen fest, während *deptno* noch einige Bereichsüberprüfungen durchführt. Zu den Details (nicht nur von DOMAINS) kommen wir im nächsten Kapitel.

Als nächstes werden elf Tabellen erstellt. Dabei ist es eine sinnvolle Vorgehensweise, zunächst alle Tabellen zu erstellen und dann die Fremdschlüssel zu definieren – auf diese Weise kommt man nicht in Versuchung, einen Fremdschlüssel auf eine Tabelle zu definieren, die noch gar nicht erstellt worden ist.

Eine VIEW ist eine festdefinierte Abfrage auf eine oder mehrere Tabellen, auf welche wie auf eine real existierende Tabelle zugegriffen werden kann; verständlicherweise müssen die Tabellen vor den VIEWS erstellt werden.

Eine EXCEPTION ist hier schlicht eine Fehlermeldung, welche von InterBase ausgegeben oder an die Datenbank-Anwendung weitergeleitet wird. Des weiteren können noch Prozeduren und TRIGGER definiert werden, doch dazu später mehr.

Zum Schluß werden noch die Zugriffsberechtigungen vergeben. Beachten Sie, daß der Zugriff auf jede Tabelle, VIEW, STORED PROCEDURE explizit vergeben werden muß. Hier im Beispiel werden die Rechte an jeden (*Public*) vergeben; was dann noch die Option *with grant option* soll, bleibt Geheimnis des Programmierers.

Die Metadaten von Tabellen und VIEWS

Wie eben festgestellt wurde, bringt die Anzeige aller Metadaten einer Datenbank eher unübersichtliche Ergebnisse. Von daher kann man mit EXTRACT | SQL METADATA FOR TABLE bzw. EXTRACT | SQL METADATA FOR VIEW die Metadaten einer einzigen Tabelle oder View auflisten. Leider beschränkt sich die Anzeige auf die Domains und die Tabellendefinition; zumindest die Zugriffsrechte würden sachlich hier noch dazugehören.

```
/* Extract Table SALES */
/* Domain definitions */
CREATE DOMAIN EMPNO AS SMALLINT;
CREATE DOMAIN CUSTNO AS INTEGER
  CHECK (VALUE > 1000);
CREATE DOMAIN PRODTYPE AS VARCHAR(12)
  DEFAULT 'software'
  CHECK (VALUE IN ('software', 'hardware', 'other', 'N/A')) NOT NULL;
CREATE DOMAIN PONUMBER AS CHAR(8)
  CHECK (VALUE STARTING WITH 'V');
```

```
/* Table: SALES, Owner: SYSDBA */
CREATE TABLE SALES (PO_NUMBER PONUMBER NOT NULL,
     CUST_NO CUSTNO NOT NULL,
     SALES_REP EMPNO,
     ORDER_STATUS VARCHAR(7) DEFAULT 'new' NOT NULL,
     ORDER_DATE DATE DEFAULT 'now' NOT NULL,
     SHIP_DATE DATE,
     DATE_NEEDED DATE,
     PAID CHAR(1) DEFAULT 'n',
     QTY_ORDERED INTEGER DEFAULT 1 NOT NULL,
     TOTAL_VALUE NUMERIC(9, 2) NOT NULL,
     DISCOUNT FLOAT DEFAULT 0 NOT NULL,
     ITEM_TYPE PRODTYPE,
     AGED COMPUTED BY (ship_date - order_date),
   PRIMARY KEY (PO_NUMBER));
```

Einzelne Metadaten

Mit VIEW | METADATA INFORMATION lassen sich Metadaten gezielter auffinden. Nach Aufruf dieses Menüpunktes wird zunächst ein Fenster nach Bild 13.11 geöffnet.

Wählen Sie zunächst in der ComboBox *View Information on*, welche Art von Metadaten gesucht wird. Hier stehen Ihnen folgende Einträge zur Auswahl:

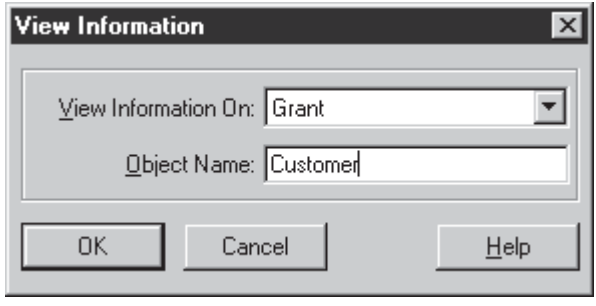


Bild 13.11: Metadaten anzeigen

- Check
- Database
- Domain
- Exception

- Filter
- Function
- Generator
- Grant
- Index
- Procedure
- System
- Table
- Trigger

Manchmal muß im Edit-Feld *Object Name* eine Eingabe gemacht werden, bei manchen Arten kann dieses Feld auch leer bleiben. Wird beispielsweise in der ComboBox *Table* gewählt und das Edit-Feld leer gelassen, dann wird angezeigt, welche Tabellen definiert sind. Wird dann der Tabellenname eingegeben, dann werden die Metadaten dieser Tabelle angezeigt.

13.4 Upsizing

In der Mehrzahl der Fälle, bei denen eine Datenbankanwendung entwickelt wird, gibt es schon eine Datenbank, welche jedoch den Anforderungen nicht mehr genügt. Während es in manchen Fällen ausreicht, eine neue Oberfläche und neue Funktionen zu generieren, muß in vielen Fällen das Datenbanksystem gewechselt werden. In den meisten Fällen wird dabei von einem Desktop-Datenbanksystem – wie beispielsweise dBase oder Paradox – auf ein Client-Server-System gewechselt. Diesen Vorgang nennt man *Upsizing*.

Meist wird man beim Upsizing das Datenmodell neu erstellen müssen, was entsprechend aufwendig ist. Da jedoch in den meisten Fällen sich das alte Datenmodell als nicht optimal herausgestellt hat, wird dieses Vorgehen unumgänglich. Auch werden Datenbanken auf Client-Server-Systemen meist in viel mehr einzelne Tabellen aufgeteilt, welche dann über JOINs zusammengefügt werden (in Form von VIEWs oder STORED PROCEDUREs).

Es gibt jedoch auch Fälle, in denen das alte Datenmodell ganz oder wenigstens teilweise brauchbar ist. Auch wird man die Daten wohl in den seltensten Fällen neu von Hand eingeben, das wäre viel zu aufwendig und damit zu teuer. Auf jeden Fall muß die Datenbank ganz oder teilweise auf den Server transferiert werden. Zu diesem Zweck eignet sich beispielsweise die Komponente *TBatchMove*. Einfacher geht es jedoch mit einem kleinen Tool, das mit C++Builder geliefert wird: dem Datenmigrations-Experten.

13.4.1 Der Datenmigrations-Experte

Der Datenmigrations-Experte ist ein Tool, welches automatisch Tabellenstrukturen und Daten kopiert. Wir wollen dieses Tool gleich dazu verwenden, um zwei Paradox-Tabellen, die mit einer Referenz verbunden sind, in eine InterBase-Datenbank zu kopieren.

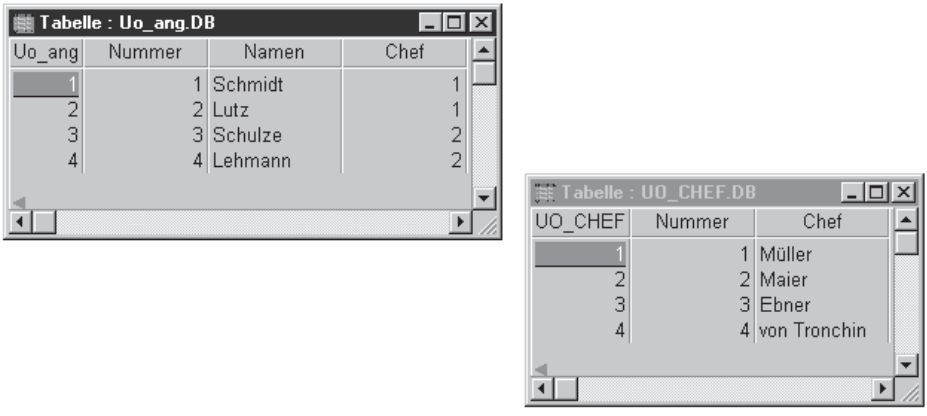


Bild 13.12: Die Beispieltabellen

Bild 13.12 zeigt die Datenstruktur und den Inhalt der beiden Tabellen. Bei der Tabelle *uo_ang.db* muß das Feld *Namen* zwingend eingegeben werden, das Feld

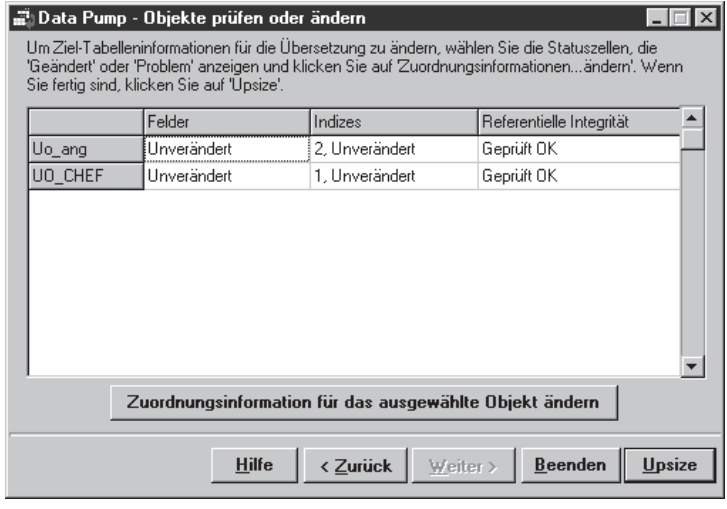


Bild 13.13: Das Analyseergebnis des Datenmigrationsexperten

Nummer ist ein selbstinkrementierendes Feld. Nach dem Start des Datenmigrations-Experten werden zunächst die Aliase der Quelle- und der Zieldatenbank sowie die zu kopierenden Felder ausgewählt. Anschließend analysiert das Tool diese Tabellen und zeigt eventuell auftretende Probleme an. Bei unseren beiden Tabellen gibt es – wie Bild 13.13 zeigt – keine weiteren Probleme. Mit Hilfe des Buttons ZUORDNUNGSINFORMATION FÜR DAS AUSGEWÄHLTE OBJEKT ÄNDERN öffnet man ein Fenster gemäß Bild 13.14.

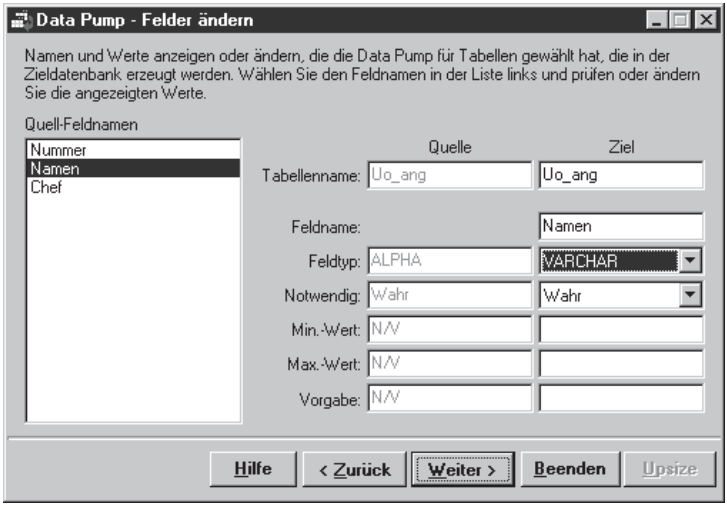


Bild 13.14: Information über die Umandlungsvorhaben

Hier erhält der Anwender Informationen darüber, was der Datenmigrationsexperte mit den einzelnen Feldern zu tun gedenkt. Beim Feld *Namen* ist beispielsweise eine Eingabe nötig. Das Tool würde dies genauso auf den Server transferieren – es sei denn, der Anwender gibt hier etwas anderes vor. Auch Minimal- und Maximal- sowie Vorgabewerte lassen sich hier ändern.

Das Feld Nummer übrigens ist vom Typ AUTOINC (selbstinkrementierend). Da InterBase diesen Datentyp nicht kennt, wird hier der Typ INTEGER verwendet.

Nun kann man mit UPSIZE den Vorgang starten. Der Datenmigrationsexperte gibt anschließend umfangreiche Informationen über das, was er getan hat. Wir wollen nun mit Hilfe des Datenbank-Explorers überprüfen, ob das Tool auch korrekt gearbeitet hat.

Zunächst werden die Tabellen und ihr Inhalt überprüft – hier ist alles korrekt, auch das Feld *Namen* ist vom Typ NOT NULL. Bei den Schlüsseln tritt dann eine Überraschung auf: Zwar werden der Fremdschlüssel und der Primärschlüssel der Chef-Tabelle korrekt übertragen, doch der Primärschlüssel der Angestellten-

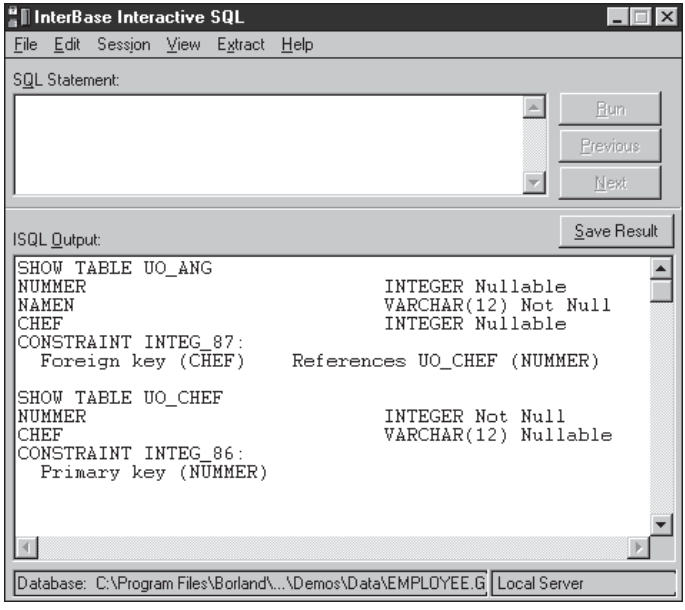


Bild 13.15: Der Aufbau der Tabellen

Tabelle wurde schlicht unterschlagen. Eine Überprüfung mit ISQL bestätigt diesen Sachverhalt, siehe Bild 13.15.

Zunächst könnte man meinen, der Datenmigrationsexperte würde hier »Mist« machen, doch die Sache klärt sich, wenn man sich die in der Datenbank vorhandenen Indizes anzeigen läßt. Hier findet man die folgende Zeile:

```
UO_ANG0 UNIQUE INDEX ON UO_ANG(NUMMER)
```

Aus dem Primärschlüssel wird hier also ein eindeutiger Index derselben Spalte. Ein (Primär- oder Sekundär-)schlüssel wird bei InterBase-Tabellen lediglich dann benötigt, wenn darauf eine Referenz gebildet wird. Nun könnte es allerdings sein, daß die Datenbank nachträglich erweitert werden soll, und daß gerade auf diese Tabelle eine Referenz gebildet werden soll.

```
CREATE TABLE uo_test
  (nummer INTEGER NOT NULL,
  ang INTEGER,
  PRIMARY KEY (nummer),
  FOREIGN KEY (ang) REFERENCES uo_ang(nummer))
```

Diese Anweisung zur Erstellung einer Tabelle würde an einem fehlenden Schlüssel in der Spalte *nummer* der Tabelle *uo_ang* scheitern. Leider läßt sich dieser nicht nachträglich erstellen, solange sich Daten in der Tabellen befinden. Auch der Trick,

alle Daten in eine andere, neu erstellte Spalte zu kopieren, die alte Spalte zu löschen und mit Primärschlüssel neu zu erstellen, funktioniert leider nicht: Zum Zeitpunkt der Spaltenerstellung wären dort – aus einsichtigen Gründen – noch keine Daten. Alle Felder hätten somit den Wert NULL, und somit wäre diese Spalte nich UNIQUE.

Man kann also nichts anderes tun, als eine neue Tabelle zu erstellen, welche über die gewünschten Metadaten verfügt.

```
CREATE TABLE uo_ang2
  (nummer INTEGER NOT NULL,
  namen VARCHAR(12),
  chef INTEGER,
  PRIMARY KEY(nummer),
  FOREIGN KEY(chef) REFERENCES uo_chef(nummer))
```

Zum Übertragen der Daten muß man eine STORED PROCEDURE erstellen, welche anschließend einmal ausgeführt wird.

```
CREATE PROCEDURE p_loschen
AS
  DECLARE VARIABLE v_nummer INTEGER;
  DECLARE VARIABLE v_namen VARCHAR(12);
  DECLARE VARIABLE v_chef INTEGER;
BEGIN
  FOR SELECT nummer, namen, chef
    FROM uo_ang
    INTO :v_nummer, :v_namen, :v_chef
  DO
  BEGIN
    INSERT INTO uo_ang2 (nummer, namen, chef)
      VALUES (:v_nummer, :v_namen, :v_chef);
  END
END


EXECUTE PROCEDURE p_loschen
```

Anschließend kann man die Daten aus der alten Tabelle löschen – das Löschen der ganzen Tabelle war zumindest auf meinem System nicht möglich.

```
DELETE FROM uo_ang
```

Um sicherzustellen, daß für alle Tabellen auch ein Primärschlüssel erstellt wird, kann man eine Tabelle generieren, welche alle anderen Tabellen referenziert. Diese Tabelle wird mit allen anderen auf den Server transferiert und anschließend gelöscht.

14 InterBase SQL

Die SQL-Anweisungen, welche die BDE bei Paradox- und dBase-Tabellen erlaubt, sind nur eine Teilmenge des SQL-Standards, insbesondere bei der DDL (Data Definition Language) gibt es viele Einschränkungen. Für vieles sind eigene SQL-Anweisungen auch nicht erforderlich, weil sich diese Dinge viel einfacher über die Datenbankoberfläche regeln lassen.

Die InterBase-Server lassen sich jedoch vollständig (und ausschließlich) über SQL-Anweisungen ansprechen, schon von daher ergibt sich die Notwendigkeit, den SQL-Sprachumfang vollständig zu implementieren (dafür läßt sich längst nicht alles über die Datenbankoberfläche regeln).

Nachfolgend soll eine Einführung in die Datenbanksprache SQL gegeben werden, wobei der Schwerpunkt auf der Datendefinition (DDL) liegt; die Anweisung SELECT wurde schon recht umfassend in Kapitel 3 besprochen. Die einzelnen Themen sind in der Reihenfolge aufgenommen, in der sie bei einer (vernünftig geplanten) Datenbankerstellung auftreten:

- DOMAINS
- Tabellen & VIEWs
- Generatoren, Prozeduren, TRIGGER und EXCEPTIONS
- Zugriffsberechtigungen

14.1 DOMAINS

DOMAINS sind Wertebereiche, auf denen dann die einzelnen Tabellenspalten beruhen; sie entsprechen den Typen bei C++. Hier wie dort sind einige Typen / DOMAINS vordefiniert, andere lassen sich vom Programmierer definieren. Neben der Angabe eines Wertebereichs ist es auch möglich, Gültigkeitsprüfungen durchführen zu lassen.

Für den Einsteiger ist es oft nicht ersichtlich, warum in diesem oder jenem Fall eine DOMAIN definiert wird, die lediglich auf einen Standard-Typ verweist. Für eine Liste mit Namen könnte man einfach formulieren:

```
CREATE TABLE t_namen
   (vornamen VARCHAR(20),
    nachnamen VARCHAR(30)
    PRIMARY KEY (nachnamen))
```

Mit dem Einsatz von DOMAINS wird alles zunächst komplizierter:

```
CREATE DOMAIN d_vornamen AS VARCHAR(20);
CREATE DOMAIN d_nachnamen AS VARCHAR(30);
CREATE TABLE t_namen
   (vornamen d_vornamen,
    nachnamen d_nachnamen,
    PRIMARY KEY (nachnamen))
```

Was soll nun in diesem Fall der Vorteil von DOMAINS sein? Bei der Erstellung von Fremdschlüsseln ist es eigentlich erforderlich, daß beide Spalten der gleichen DOMAIN unterliegen. Dies ist zunächst kein Problem, bei der Definition der Tabellen muß halt entsprechend aufgepaßt werden, zumal InterBase hier recht pragmatisch reagiert und aus verschieden langen Strings oder unterschiedlichen Integer-Typen nicht gleich ein Problem macht.

Nehmen wir aber einmal den Fall, daß wir eine Tabelle mit der Spalte *Nachnamen* hätten, welche auf 30 Zeichen ausgelegt ist und den Primärschlüssel bildet, sowie eine zweite Tabelle, welche auf eben diesen Primärschlüssel eine Referenz bildet. Nun stellt man während des Betriebs fest, daß die 30 Zeichen nicht ausreichen und erhöht sie »mal schnell« auf 35; daß dies gar nicht so einfach ist, werden wir nachher noch feststellen.

(Man kann nämlich nur Spalten hinzufügen oder löschen, nicht aber diese umbenennen oder umdefinieren. In diesem Fall müßte also eine neue Spalte eingefügt werden, alle Werte von der einen in die andere Spalte kopiert werden, die alte Spalte gelöscht, dann entsprechend neu erstellt sowie die Daten zurückkopiert werden. Schließlich kann man die Hilfsspalte löschen. Leider kann man dieses Procedere nicht dadurch umgehen, daß man bei der Verwendung von DOMAINS dieselben entsprechend umdefiniert.)

Auf jeden Fall ist die Motivation gering, dies alles auch bei anderen Tabellen zu tun, so wird dann auch nicht so genau nach Referenzen auf diese Spalte gesucht. Momentan geht noch alles gut, dann aber muß eine Referenz auf einen überlangen Nachnamen gebildet werden, und dann ist »Schluß mit lustig«.

Ein weiterer Grund für den Einsatz von DOMAINS ist eher psychologischer Natur: Man macht sich dabei zwangsläufig Gedanken über die notwendigen Datentypen und definiert nicht alles bei CREATE TABLE als VARCHAR (20).

14.1.1 Datentypen

Im Vergleich zu Paradox-Tabellen ist die Vielfalt der Datentypen bei InterBase ein wenig eingeschränkt, was aber nicht unbedingt ein Problem sein muß, ganz im Gegenteil.

Strings

InterBase bietet (wenn wir einmal NCHAR vernachlässigen wollen) zwei Gruppen von Zeichentypen:

- CHAR (alternativ CHARACTER)
- VARCHAR (alternativ CHAR VARYING oder CHARACTER VARYING).

Laut Handbuch werden bei der VARCHAR-Gruppe nur die vorhandenen Zeichen gespeichert, während bei der CHAR-Gruppe fehlende Länge mit Leerzeichen aufgefüllt wird; dies verbrauche zwar mehr Speicherplatz, biete dafür aber schnelleren Zugriff.

Um dies zu überprüfen, wurde eine neue Datenbank erstellt und folgende Anweisungen eingegeben:

```
CREATE TABLE ttest
  (linie1 CHAR(30),
   linie2 VARCHAR(30));
INSERT INTO ttest VALUES ("Linie1", "eins");
INSERT INTO ttest VALUES ("Linie1", "zwei");
INSERT INTO ttest VALUES ("Linie1", "drei");
INSERT INTO ttest VALUES ("Linie1", "vier");
```

Wie dies auf der Festplatte aussieht, zeigt Bild 14.1 auf der nächsten Seite. Als erstes fällt auf, daß es zwischen CHAR und VARCHAR keine gravierenden Unterschiede gibt, zumindest gibt es hier bei CHAR "Linie1" keine 24 Leerzeichen pro Datensatz. (»Und die Moral von der Geschicht': Traue Deinem Handbuch nicht!« Zumindest nicht beim LIBS.)

Des weiteren fällt auf, daß die Datensätze in umgekehrter Reihenfolge in der Datei gespeichert werden; der Datensatz mit dem Wert *eins* in Spalte *linie2* wurde zuerst eingegeben und steht doch ganz hinten in der Datei.

Nach dem Typenbezeichner CHAR oder VARCHAR wird in Klammer die maximale Größe (als Zahl der Zeichen) angegeben. InterBase erlaubt Zeichentypen mit bis zu 32 767 Bytes, bei allen 1-Byte-Datensätzen sind dies 32 767 Zeichen (das sollte ausreichen). Es gibt aber auch Zeichensätze mit zwei oder gar drei Byte, hier ist die maximale Stringlänge dann entsprechend geringer. Im Ernstfall können immer noch Memos über BLOB-Felder in den Tabellen gespeichert werden.

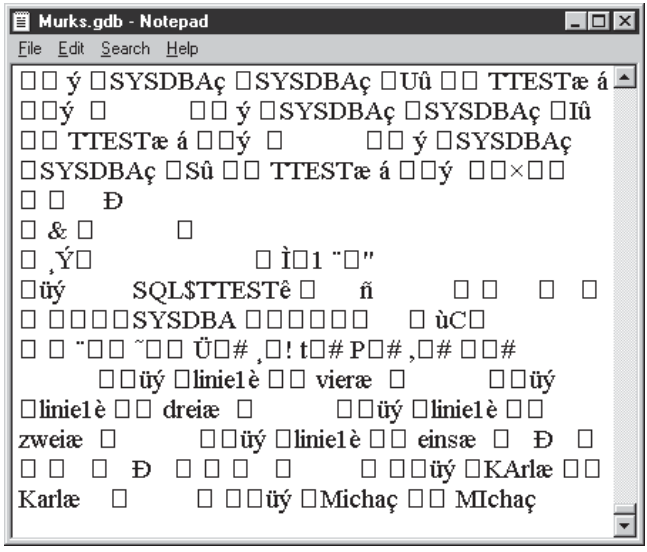


Bild 14.1: Der fehlende Unterschied von CHAR und VARCHAR

Zahlen

Bei InterBase sind folgende Zahlentypen verfügbar:

- SMALLINT: Ein 16-Bit-Integertyp mit einem Wertebereich von -32 768 bis 32 767 (alle Angaben laut *InterBase Data Definition Guide*)
- INTEGER: Ein 32-Bit-Integertyp mit einem Wertebereich von -2 147 483 648 bis 2 147 483 648.
- FLOAT: Ein Gleitkommatyp mit 32 Bit, ähnlich dem C++-Typ *float*, umfaßt den Wertebereich von +/- 3.4 * 10^{-38} bis +/- 3.4 * 10^{38};
- DOUBLE PRECISION: Ein Gleitkommatyp mit 64 Bit, umfaßt den Wertebereich von +/- 1.7 * 10^{-308} bis +/- 1.7 * 10^{308}; entspricht in etwas dem C++-Typ *double*.
- DECIMAL (precision, scale) und NUMERIC (precision, scale): Festkommatypen, deren Anzahl von Stellen und Nachkommastellen definiert werden können. Die Definition

  ```
  ... NUMERIC(8,2)
  ```

  würde eine Zahl mit insgesamt acht Stellen deklarieren, davon sechs Vorkomma- und zwei Nachkommastellen (*123 456,78*, beispielsweise für Geldbeträge).

Bei NUMERIC wird exakt mit der angegebenen *precision* gespeichert, während bei DECIMAL mindestens mit der angegebenen *precision* gespeichert wird – zur Not können auch größere Zahlen untergebracht werden. Versuchen Sie nicht, *Scale* größer als *Precision* zu machen. Bitte beachten Sie auch, daß NUMERIC und DECIMAL im C++Builder stets zu einem *TIntegerField* werden und dann ohne Nachkommastellen angezeigt werden.

- AUTOINC: Gibt es bei InterBase nicht, bei Paradox-Tabellen können Sie damit selbstinkrementierende Felder (also Felder, die automatisch durchnumeriert werden) deklarieren. Für eine solche Funktion wird ein INTEGER-Typ verwendet und dafür ein GENERATOR definiert.

Datum und Zeit

Datum und Zeit werden bei InterBase stets gemeinsam im Datentyp DATE gespeichert, der in C++Builder vom Typ *TDateTime* ist. Um Datum und Zeit einzeln zu erhalten, werden folgende Anweisungen benötigt:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TDateTime dt = Table1->FieldByName("Hire_Date")->AsDateTime;
  Edit1->Text = TimeToStr(dt);
  Edit2->Text = DateToStr(dt);
}
```

BLOBs

Die Abkürzung BLOB steht für *binary large Object*. Darunter versteht man große Spaltenwerte von nicht unbedingt weiter spezifizierten Daten wie Memos, Bilder, usw., welche auch nicht direkt in der eigentlichen Tabellen, sondern getrennt davon gespeichert werden.

Beim C++Builder müssen Sie sich nicht groß darum kümmern, die Komponenten *TDBMemo* und *TDBImage* arbeiten problemlos mit diesem Datentyp zusammen. Wenn Sie direkt auf den Inhalt der Felder zugreifen möchten, dann bedenken Sie bitte, daß das Feld ein *TBlobField* und nicht ein *TMemoField* oder ein *TGraphikField* ist.

Datenkonvertierung

InterBase verhält sich bei »Daten-Mischmasch« ähnlich gutmütig wie die 16-Bit-BDE bei Paradox-Tabellen; die folgende Anweisung bereitet somit keine Probleme:

```
SELECT *  FROM employee
  WHERE emp_no BETWEEN 1 and "30"
```

Sollte die Umwandlung einmal nicht automatisch passieren, kann die Funktion CAST verwendet werden, welche die Typen DATE, CHARACTER und NUMERIC ineinander umwandelt:

```
... WHERE hire_date = CAST(interview_date AS DATE)
```

Arrays

Zu den Forderungen der ersten Normalform (siehe Kapitel 1) gehört, daß alle Spaltenwerte *atomar* sind, sich also nicht sinnvoll weiter unterteilen lassen – Spalten, die auf Arrays beruhen, erfüllen diese Bedingungen nicht.

Auch wenn InterBase die Möglichkeit bietet, sogar mehrdimensionale Arrays zu definieren, sollte man lieber keinen Gebrauch davon machen. Vielmehr ist der Wunsch nach einem Array oft ein Zeichen für eine nicht optimal entworfene Datenbank. (Selbstverständlich bestätigen auch hier Ausnahmen die Regel.)

14.1.2 DEFAULT-Werte

Bei der Definition von DOMAINS haben Sie die Möglichkeit, Werte vorzugeben, die immer dann eingefügt werden, wenn der Anwender keine Eingabe macht. Hier gibt es grundsätzlich vier Möglichkeiten:

- einen explizit angegebenen Wert
- den Wert NULL
- die Variable USER
- das heutige Datum mit "NOW", siehe auch Kapitel 14.2.2

Den Wert NULL als DEFAULT-Wert zu definieren, macht nicht viel Sinn, schließlich fügt die Datenbank immer NULL-Werte ein, wenn der Anwender keine Eingabe macht. Die Variable USER beinhaltet den Wert des Benutzernamens, mit dem der Anwender sich beim Server eingeloggt hat.

Mit Hilfe von DEFAULT-Werten ist es sogar möglich, Spaltenwerte ganz automatisch einzufügen. Als Beispiel sei hier die Möglichkeit genannt, bei jedem Datensatz mitzuspeichern, wer ihn denn eingegeben hat:

```
CREATE DOMAIN d_username AS VARCHAR(15) DEFAULT USER

CREATE TABLE namen
  (vornamen VARCHAR(20),
   nachnamen VARCHAR(30),
   eingegeben d_username)

INSERT INTO namen (vornamen, nachnamen)
  VALUES ("Michael", "Mustermann")
```

```
SELECT * FROM namen

Michael     Mustermann       SYSDBA
```

Dagegen macht die automatische Eingabe von Festwerten meist nicht viel Sinn – hier würde man dann die Möglichkeit benötigen, den DEFAULT-Wert zu überschreiben. Nehmen wir hier einmal die Adressendatenbank einer Schule. Für gewöhnlich werden die eingegebenen Adressen die von Schülern sein, nur manchmal muß ein Lehrer, ein Hausmeister, eine Reinigungskraft eingegeben werden:

```
CREATE DOMAIN dberuf AS VARCHAR(15) DEFAULT "Schüler"

CREATE TABLE tadressen
  (vornamen VARCHAR(20),
    nachnamen VARCHAR(30),
    beruf dberuf)

INSERT INTO tadressen (vornamen, nachnamen)
  VALUES ("Adam", "Amsel")

INSERT INTO tadressen (vornamen, nachnamen)
  VALUES ("Berta", "Borst")

INSERT INTO tadressen (vornamen, nachnamen, beruf)
  VALUES ("Cäsar", "Conradi", "Rektor")

SELECT * FROM tadressen

VORNAMEN             NACHNAMEN                      BERUF
==================== ============================== ===============

Adam                 Amsel                          Schüler
Berta                Borst                          Schüler
Cäsar                Conradi                        Rektor
```

Ich möchte noch darauf hinweisen, daß ich diese Tabelle nur als Beispiel für DEFAULT-Werte entworfen habe. In der Praxis würde man selbstverständlich eine zweite Tabelle mit den verschiedenen Berufen anlegen und eine Referenz darauf bilden (das Feld *beruf* benötigt jeweils 15 Bytes, bei angenommenen 500 Datensätzen sind dies rund 7,5 kByte. Ein Referenzfeld kommt mit zwei Bytes aus, das wären hier 1 kByte, und außerdem wäre die Änderung einfacher, wenn es statt *Schüler* nun »politisch korrekt« *SchülerIn* oder *Schüler(in)* heißen müßte).

14.1.3 Eingabe erzwingen

In manchen Feldern dürfen keine NULL-Werte vorhanden sein, weil beispielsweise darauf eine Referenz gebildet wird oder eine Eingabe ohne diese Angabe nicht sinnvoll wäre. Hier kann dann der Befehl NOT NULL gegeben werden:

```
CREATE DOMAIN d_test AS VARCHAR(15) NOT NULL
```

14.1.4 Gültigkeitsprüfungen

Es kommt immer mal wieder vor, daß Personen eine Datenbankanwendung bedienen, die nicht genau wissen, was sie tun müssen, oder die sich bei der Eingabe vertippen. Auf diese Weise wäre es möglich, daß Daten in die Datenbank gelangen, welche nicht richtig sind. Vielfach läßt sich das nicht vermeiden, doch für einige Fälle kann man Gültigkeitsprüfungen implementieren, welche dafür sorgen, daß die Aufnahme unsinniger Daten mit einer Fehlermeldung verweigert wird.

Das Gehalt eines Angestellten kann beispielsweise nie negativ sein:

```
CREATE DOMAIN dgehalt AS NUMERIC(7,2)
  CHECK (VALUE > 0)
```

An dieser Stelle gleich eine Warnung: Bedenken Sie immer, daß die Datenbank sich weigern wird, einen Datensatz anzunehmen, welcher die CHECK-Bedingung(en) nicht erfüllt. Von daher ist es nicht sinnvoll, hier beispielsweise den tariflichen Mindestlohn einzutragen – sobald eine Aushilfskraft für ein paar Stunden eingestellt wird, müßte diese sehr großzügig bezahlt werden, damit das Gehalt eingegeben werden kann.

Bei der Gültigkeitsprüfung sind auch AND- und OR-Verknüpfungen erlaubt:

```
CREATE DOMAIN dgehalt AS NUMERIC(7,2)
  CHECK (VALUE > 0) AND (VALUE < 12000)
```

Des weiteren gibt es die Möglichkeit, über die CHECK-Bedingung Mengentypen zu definieren:

```
CREATE DOMAIN dberuf AS VARCHAR(15)
  DEFAULT "Schüler"
  CHECK (VALUE IN ("Schüler", "Lehrer", "Rektor", "Hausmeister"))
```

Was ich davon halte, hier keine extra Tabelle zu erstellen und eine Referenz darauf zu bilden, habe ich vorhin bereits erwähnt. Hier kommt noch erschwerend hinzu, daß es extrem aufwendig wäre, während des Betriebs der Datenbank diese Menge beispielsweise um den Eintrag *Reinigungskraft* zu ergänzen.

14.1.6 DOMAINS ändern

Auf der DOMAIN-Definition beruhen die Tabellenspalten, von daher muß man nicht erwarten, daß sich der zugrundeliegende Datentyp ändern läßt – hier würde nur die Möglichkeit bleiben, eine neue DOMAIN zu erstellen und die alte zu löschen; und selbst das geht nicht, solange noch eine Tabellenspalte auf dieser DOMAIN beruht.

Was sich dagegen zur Laufzeit ändern läßt, das sind die Gültigkeitsprüfung und der DEFAULT-Wert:

```
CREATE DOMAIN dtest AS VARCHAR(15)

ALTER DOMAIN dtest SET DEFAULT "Beruf Schüler"

ALTER DOMAIN dtest ADD CHECK (VALUE LIKE "Beruf %")

SHOW DOMAIN dtest
DTEST                VARCHAR(15) Nullable
                     DEFAULT "Beruf Schüler"
                     CHECK (VALUE LIKE "Beruf %")

ALTER DOMAIN dtest DROP CONSTRAINT

SHOW DOMAIN dtest
DTEST                VARCHAR(15) Nullable
                     DEFAULT "Beruf Schüler"

ALTER DOMAIN dtest DROP DEFAULT

SHOW DOMAIN dtest
DTEST                VARCHAR(15) Nullable
```

Zunächst wird wie gewohnt eine DOMAIN erstellt, dann wird mit ALTER DOMAIN zunächst ein Vorgabewert und dann eine Gültigkeitsprüfung definiert. Beachten Sie, daß der Vorgabewert mit SET, die Prüfung jedoch mit ADD hinzugefügt wird. (Sie können aber keine weiteren Gültigkeitsprüfungen hinzufügen – löschen Sie gegebenenfalls die alte Prüfung und fügen Sie eine neue hinzu, vielleicht mit den entsprechenden AND- und OR-Operatoren).

Versuchen Sie nicht, den Befehl SHOW einzugeben (das zieht eine Fehlermeldung nach sich), sondern wählen Sie VIEW | METADATE INFORMATION. Um die Prüfung zu löschen, wählen Sie DROP CONSTRAINT, den Vorgabewert löschen Sie mit DROP DEFAULT.

14.1.7 DOMAINS löschen

Sehr einfach ist dann die Anweisung, um eine DOMAIN zu löschen (beispielsweise um sie danach mit einem anderen Datentyp neu zu erstellen). Beachten Sie bitte, daß die DOMAIN nicht in irgendeiner Spaltendefinition verwendet werden darf, um gelöscht werden zu können.

```
DROP DOMAIN dtest
```

14.1.8 Generatoren

Eigentlich gehört das Thema GENERATOR nicht zum Thema DOMAIN, da es aber in Paradox die selbstinkrementierenden Felder gibt, werden vielleicht einige Leser das Thema hier suchen.

Mit einem GENERATOR ist es möglich, in Spalten automatisch fortlaufende Nummern einzufügen. Dies ist insbesondere dann vorteilhaft, wenn man einen eindeutigen Primärschlüssel benötigt, beispielsweise eine Personalnummer, eine Kundennummer, eine Rechnungsnummer. Sie können in einer Datenbank mehrere Generatoren erstellen, welche allerdings unterschiedliche Namen haben müssen.

Wie man hier vorgeht, soll gleich anhand eines Beispiels erläutert werden:

```
CREATE TABLE test_1
   (nummer INTEGER,
    namen VARCHAR(15))

CREATE GENERATOR nummer_gen

SET GENERATOR nummer_gen TO 234
```

Zunächst wird eine kleine Tabelle für unser Beispiel erstellt, für die Spalte *Nummer* soll dann eine fortlaufende Nummer vergeben werden. Um einen Generator zu erstellen, muß einfach nur der Befehl CREATE GENERATOR gegeben werden, dem der Namen des Generators folgt. Weitere Optionen gibt es hier nicht. Ein GENERATOR ist nicht an eine bestimmte Tabelle oder Spalte gebunden.

Ein Generator wird mit dem Zahlenwert Null initialisiert. Mit der Anweisung SET GENERATOR kann der GENERATOR auf einen anderen Wert gestellt werden. Dies ist beispielsweise dann nötig, wenn auf ein anderes System umgestellt wird und die neuen Rechnungsnummern (oder was auch immer) nahtlos an die alten anschließen sollen.

Bisweilen möchte eine Firma auch kaschieren, daß sie neu auf dem Markt ist und beginnt bei einer willkürlich gewählten Rechnungsnummer. Ob das sinnvoll ist oder eher peinlich wirkt, soll an dieser Stelle nicht diskutiert werden, auf jeden Fall ist es möglich.

```
INSERT INTO test_1 (nummer, namen)
  VALUES (GEN_ID(nummer_gen, 3), "Eins")

INSERT INTO test_1 (nummer, namen)
  VALUES (GEN_ID(nummer_gen, 3), "Zwei")

INSERT INTO test_1 (nummer, namen)
  VALUES (GEN_ID(nummer_gen, 3), "Drei")
```

```
SET GENERATOR nummer_gen TO 100

INSERT INTO test_1 (nummer, namen)
  VALUES (GEN_ID(nummer_gen, -1), "Vier")

INSERT INTO test_1 (nummer, namen)
  VALUES (GEN_ID(nummer_gen, -1), "Fünf")

SELECT * FROM test_1

     NUMMER NAMEN
=========== ===============

        237 Eins
        240 Zwei
        243 Drei
         99 Vier
         98 Fünf
```

Um einen Generatorwert einzufügen, wird in einer INSERT- oder UPDATE-Anweisung die Funktion GEN_ID verwendet, als Parameter werden der Name der Generators sowie der Wert, um den er erhöht werden soll, übergeben.

Wie das Ergebnis der Abfrage zeigt, wird zuerst die Generatorzahl verändert, erst dann wird die (geänderte) Zahl in die Tabellenspalte eingefügt. Wollte man, daß der erste Datensatz den Generatorwert Null erhält, dann müßte man vorher den Generatorwert mit SET GENERATOR entsprechend negativ setzen.

Wie das Beispiel zeigt, muß der Generator nicht zwangsweise immer um eins erhöht werden, hier im Beispiel wird einmal *drei*, ein andermal *minus eins* verwendet, bei letzterem wird also rückwärts gezählt. Auf diese Weise wäre es beispielsweise möglich, den Mitgliederstand eines Vereins etwas freundlicher aussehen zu lassen, als er es tatsächlich ist.

Des weiteren ist es erlaubt, während des Betriebs mit SET GENERATOR einen neuen Wert zuzuweisen – es sollte dann aber sichergestellt sein, daß nicht plötzlich doppelte Werte ausgegeben werden, womöglich noch in einem Primärschlüssel (deren Annahme die Datenbank verweigern würde).

14.2 Tabellen

Tabellen sind die Grundlage einer jeden Datenbank. Nicht nur die eigentlichen Daten werden in Tabellen gespeichert, sondern auch alle Metadaten wie Referenzen, Gültigkeitsprüfungen usw.

14.2.1 CREATE TABLE

Um eine Tabelle zu erstellen, wird der CREATE TABLE-Befehl verwendet. Statt einer abstrakten Referenz der Syntax lieber gleich ein anschauliches Beispiel:

```
CREATE TABLE t_mitarbeiter
  (nummer INTEGER NOT NULL,
  vornamen d_vornamen,
  nachnamen d_nachnamen,
  arbeitsplatz INTEGER NOT NULL
    CHECK ((arbeitsplatz > 99) AND (arbeitsplatz < 1000)),
  PRIMARY KEY (nummer),
  FOREIGN KEY (arbeitsplatz) REFERENCES t_arbeitsplatz (nummer))
```

Zwingend erforderlich für einen CREATE TABLE-Befehl ist die Vergabe eines Tabellennamens sowie die Definition mindestens einer Spalte (alles andere würde ohnehin nicht viel Sinn machen). Gleich noch ein Wort zum Tabellennamen: Um in großen Datenbanken die Übersicht zu behalten, kann man allen Bezeichnern einen Buchstaben voranstellen, aus dem zu ersehen ist, um was für ein Objekt es sich handelt (Tabelle *t_test*, View *v_test*, Domain *d_test*, usw.) Des weiteren empfehle ich, alle SQL-Schlüsselwörter in Großbuchstaben und alle Bezeichner in Kleinbuchstaben zu schreiben. Auf diese Weise ist der »grammatische Aufbau« einer Anweisung leichter zu erfassen.

Alle Spaltendefinitonen werden dann als »Parameter« hinzugefügt. Eine Spalten-Definition besteht stets aus einem Spaltennamen sowie einem Typ (oder einer DOMAIN), auf den (der) sie beruht. Des weiteren sind alle Möglichkeiten gegeben, die wir schon beim Thema DOMAIN kennengelernt haben. Man kann also

- mit NOT NULL eine Eingabe erzwingen,
- mit DEFAULT einen Wert vorgeben und
- mit CHECK Gültigkeitsprüfungen durchführen. Beachten Sie bei CHECK-Anweisungen, daß Sie hier nicht mehr VALUE verwenden dürfen, sondern den Namen der Spalte angeben müssen.

Eine Spalte oder eine Kombination von Spalten kann als Primärschlüssel gewählt werden. Im Gegensatz zu Paradox-Tabellen muß der Primärschlüssel nicht aus dem ersten Feld oder den ersten Feldern bestehen.

Um eine Referenz auf eine andere Tabelle zu bilden, wird ein Anweisung FOREIGN KEY verwendet; in Klammern schließt sich die Spalte (oder die Spalten) der »eigenen« Tabelle an, über welche die Referenz verbunden wird. Nach dem Schlüsselwort REFERENCES schließt sich der Name der Tabelle an, auf welche die Referenz gebildet wird, in Klammern dahinter die Spalte(n), über welche die Tabelle verbunden wird (werden). Die Spalte(n) der beiden Tabellen sollte(n) strenggenommen derselben DOMAIN unterliegen und in der Praxis zumindest halbwegs kompatibel sein. Die Spalte(n) der Fremdtabelle muß (müssen) als Primärschlüssel oder als UNIQUE definiert worden sein.

Bevor Sie Tabellen erstellen...

... sollten Sie ein wenig nachdenken. Wer bislang mit Paradox-Tabellen und der Datenbankoberfläche gearbeitet hat, der wird leicht ein wenig die Übersicht verlieren. Es ist zwar prinzipiell auch möglich, mit der Datenbankoberfläche InterBase-Tabellen zu erstellen, viele Funktionen sind aber hier nicht vorhanden.

Da nachträgliche Änderungen mit *Interactive SQL* meist reichlich aufwendig sind, sollte man ein vernünftiges Konzept erarbeiten, bevor man beginnt:

- Zuallererst wird die Datenbank richtig entworfen und normalisiert. Die entsprechenden Schritte wurden in Kapitel 1 beschrieben.
- Als nächstes werden zweckmäßige Datentypen festgelegt und entsprechend die DOMAINs erstellt.
- Meist ist es zweckmäßig, alle Tabellen ohne Fremdschlüssel zu erstellen und diesen nachher mit ALTER TABLE hinzuzufügen; auf diese Weise muß man sich keine Gedanken darum machen, in welcher Reihenfolge die Tabellen zu erstellen sind.

Berechnete Spalten

Bei der Definition von Tabellen kann man Spalten definieren, deren Werte nicht eingegeben, sondern vom DBMS berechnet werden; diese Spalten werden mit dem Schlüsselwort COMPUTED BY definiert. Als Rechenoperationen sind dabei die Operationen erlaubt, welche man auch bei einer Abfrage mit SELECT durchführen könnte. Das folgende Beispiel berechnet den Gesamtpreis aus Stückzahl und Einzelpreis:

```
CREATE TABLE t_test
  (anzahl INTEGER,
  bestnr INTEGER,
  ep NUMERIC (8,2),
  gp COMPUTED BY (anzahl * ep))

INSERT INTO t_test (anzahl, bestnr, ep) VALUES (1, 1234, 10.00)
INSERT INTO t_test (anzahl, bestnr, ep) VALUES (7, 1234, 2.00)
INSERT INTO t_test (anzahl, bestnr, ep) VALUES (2, 14534, 15.00)

SELECT * FROM t_test

     ANZAHL      BESTNR           EP           GP
========== ========== ========== ==========
          1       1234      10.00       10.00
          7       1234       2.00       14.00
          2      14534      15.00       30.00
```

Vernünftigerweise werden die berechneten Spaltenwerte nicht in der Tabelle gespeichert, sondern bei Abfragen anhand der Formel neu berechnet. Es ist auch möglich, auf diese Weise Strings zusammenzufügen:

```
CREATE TABLE t_test
  (vornamen VARCHAR(20),
  nachnamen VARCHAR(30),
  namen COMPUTED BY (vornamen || " " || nachnamen))
```

Das nächste Beispiel sorgt dafür, daß den Beträgen immer die Währungsbezeichnung beigefügt wird, die Typenumwandlung funktioniert hier automatisch:

```
CREATE TABLE t_money
  (zahl NUMERIC (8,2),
  betrag COMPUTED BY (zahl || " DM"))
```

Ergänzen von DOMAINS

Bei der Definition von DOMAINS können diverse Gültigkeitsprüfungen definiert werden; diese können bei der Definition von Tabellen ergänzt werden:

```
CREATE DOMAIN d_namen AS VARCHAR (30)
  CHECK ((VALUE LIKE "Herr%") OR (VALUE LIKE "Frau%"))
```

```
CREATE DOMAIN d_stundenlohn AS NUMERIC (5,2)
  CHECK (VALUE > 9.00)

CREATE TABLE t_aushilfen
  (namen d_namen NOT NULL,
  stundenlohn d_stundenlohn CHECK (stundenlohn < 32.00))
```

Im ersten Fall wird die Anweisung NOT NULL hinzugefügt; während in anderen Tabellen, die auf dieser DOMAIN beruhen, NULL-Werte erlaubt sind, ist dies in der Tabelle *t_aushilfen* nicht erlaubt. Bei der DOMAIN *d_stundenlohn* wird ein Mindestwert von neun DM angenommen. In der Tabelle *t_aushilfen* soll nun auch von einem Fehler ausgegangen werden, wenn der Stundenlohn 32,- DM übersteigt.

Bitte beachten Sie, daß Ihnen mit dem Menü-Befehl EXTRACT | SQL METADATA FOR TABLE die zusätzliche Gültigkeitsprüfung nicht angezeigt wird:

```
/* Extract Table T_AUSHILFEN */
/* Domain definitions */
CREATE DOMAIN D_NAMEN AS VARCHAR(30)
     CHECK ((VALUE LIKE "Herr%") OR (VALUE LIKE "Frau%"));
CREATE DOMAIN D_STUNDENLOHN AS NUMERIC(9, 2)
     CHECK (VALUE > 9.00);

/* Table: T_AUSHILFEN, Owner: SYSDBA */
CREATE TABLE T_AUSHILFEN (NAMEN D_NAMEN NOT NULL,
     STUNDENLOHN D_STUNDENLOHN);
```

Wenn die Annahme von Werten verweigert wird, dann wählen Sie stets VIEW | METADATA INFORMATION | TABLE, dort wird diese zusätzliche Gültigkeitsprüfung dann unter dem Begriff CONSTRAINT aufgeführt.

```
SHOW TABLE t_aushilfen
NAMEN                          (D_NAMEN) VARCHAR(30) Not Null
                               CHECK ((VALUE LIKE "Herr%")
                               OR (VALUE LIKE "Frau%"))

STUNDENLOHN                    (D_STUNDENLOHN) NUMERIC(9, 2) Nullable
                               CHECK (VALUE > 9.00)

CONSTRAINT INTEG_2:
  CHECK (stundenlohn < 32.00)
```

Primär- und Sekundärschlüssel

Wie in Kapitel 1 ausführlich erläutert wurde, erhalten Tabellen in der Regel eine durchlaufende Nummer als Primärschlüssel. Eine Ausnahme bilden Tabellen, die eine Verknüpfung zwischen zwei Tabellen herstellen und nur aus zwei INTEGER-Spalten bestehen, die jeweils eine Referenz auf eine andere Tabelle bilden – hier bildet man den Primärschlüssel aus der Kombination dieser beiden Spalten. Auch wenn alle SQL-Datenbanken es erlauben, andere Felder zu Primärschlüsseln zu erheben, dann heißt dies noch lange nicht, daß das sinnvoll ist.

Um einen Primärschlüssel zu definieren, wird die Anweisung PRIMARY KEY verwendet. Die entsprechende Spalte(n) muß (müssen) dabei als NOT NULL definiert sein. Liegt dem Primärschlüssel nur eine Spalte zugrunde, dann wird diese durch die Definition des Primärschlüssels implizit als UNIQUE definiert, liegen dem Primärschlüssel mehrere Spalten zugrunde, dann muß die Kombination der Spaltenwerte in dieser Tabelle einzigartig sein.

Ein Sekundärschlüssel ist eine weitere Spalte oder eine weitere Kombination von Spalten, der nur einzigartige Werte oder Kombinationen enthalten darf. Im folgenden Beispiel einer Artikeldatenbank wird die Artikelbezeichnung als Sekundärschlüssel definiert. Dadurch wird vermieden, daß dieselbe Artikelbezeichnung ein zweites Mal in der Tabelle auftaucht (also der Artikel schon in das Sortiment aufgenommen worden ist). Die zweite INSERT-Anweisung wird deshalb vom DBMS abgelehnt:

```
CREATE TABLE t_waren
  (nummer INTEGER NOT NULL,
  artikel VARCHAR(15) NOT NULL UNIQUE,
  preis NUMERIC (8,2) NOT NULL,
  PRIMARY KEY (nummer))

INSERT INTO t_waren VALUES (1, "BC 238 C", 0.25)
INSERT INTO t_waren VALUES (2, "BC 238 C", 0.25)
```

Liegt dem Primärschlüssel nur eine einzige Spalte zugrunde, dann kann der Spaltendefinition auch die Anweisung PRIMARY KEY beigefügt werden, wie das nachfolgende Beispiel zeigt. Läßt man sich die Metadaten einer Tabelle anzeigen, dann findet man den Primärschlüssel schneller, wenn er in einer extra Anweisung definiert ist, so daß ich die vorherige Vorgehensweise bevorzuge.

```
CREATE TABLE t_waren2
  (nummer INTEGER NOT NULL PRIMARY KEY,
  artikel VARCHAR(15) NOT NULL UNIQUE,
  preis NUMERIC (8,2) NOT NULL)
```

Soll eine Kombination von Feldern zum Sekundärschlüssel erhoben werden, dann kann nicht einfach jeder Spalte eine UNIQUE-Anweisung beigefügt werden, würde doch die Datenbank jedes dieser Felder zu einem Sekundärschlüssel erheben. Hier muß dann am Ende der Tabelle mit dem Befehl UNIQUE ein Sekundärschlüssel erstellt werden.

Wird in einer Adressendatenbank die Nummer zum Primärschlüssel erhoben, dann kann es leicht vorkommen, daß sich doppelte Datensätze in die Tabelle einschleichen. Um dies ein wenig einzudämmen, wurde hier ein Sekundärschlüssel über die Felder *nachnamen, strasse* und *plz* gebildet. Der Verzicht auf den Vornamen kann zwei Gründe haben: Zum einen wird dadurch vermieden, daß doppelte Datensätze durch unterschiedliche Schreibweise des Vornnamens entstehen (*Karl Mustermann, K. Mustermann, Karl-Heinz Mustermann, Karl Heinz Mustermann*, usw.). Zweitens werden dadurch auch mehrere Lieferungen an dieselbe Familie vermieden (wenn dies nicht gewünscht ist, muß der Vorname mit in die Definition des Sekundärschlüssels aufgenommen werden). In diesem Beispiel verursachen die letzten beiden INSERT-Anweisungen eine Fehlermeldung:

```
CREATE TABLE t_adressen
(nummer INTEGER NOT NULL,
vornamen VARCHAR(20),
nachnamen VARCHAR(30) NOT NULL,
strasse VARCHAR(25) NOT NULL,
plz VARCHAR(6) NOT NULL,
ort VARCHAR(20) NOT NULL,
PRIMARY KEY (nummer),
UNIQUE (nachnamen, strasse, plz))

INSERT INTO t_adressen VALUES (1, "Michael", "Mustermann",
"Bismarckstraße 22", "12 345", "Berlin")

INSERT INTO t_adressen VALUES (2, "Michael", "Mustermann",
"Bismarckstraße 22", "12 345", "Berlin")

INSERT INTO t_adressen VALUES (3, "Gabriele", "Mustermann",
"Bismarckstraße 22", "12 345", "Berlin")
```

Noch zwei Anmerkungen: Aufmerksamen Lesern wird es nicht entgangen sein, daß die Spalte *Straße* hier als *Strasse* geschrieben wurde. Ich habe keinen Zeichensatz gefunden, der bei der Definition ein *ß* erlauben würde. Bei manchen Zeichensätzen sind die deutschen Umlaute und das *ß* noch nicht einmal bei den Datensätzen erlaubt.

Durch die Einführung eines Sekundärschlüssels ist die Tabelle strenggenommen nicht mehr in der dritten Normalform; dies sollte jedoch kein Grund dafür sein, die Tabelle in zwei Tabellen zu zerlegen.

Gültigkeitsprüfungen mit CONSTRAINT benennen

Mit dem Menüpunkt VIEW | METADATA INFORMATION | TABLE lassen wir uns die Metadaten von *t_adressen* anzeigen:

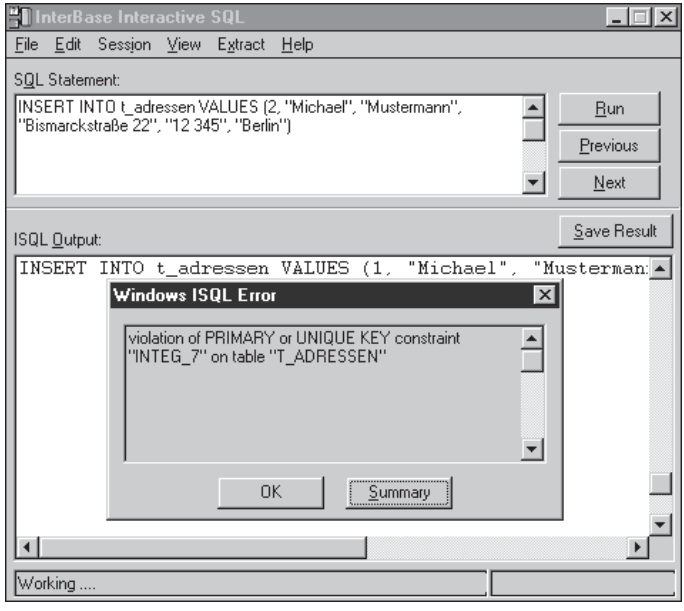


Bild 14.2: Namen des CONSTRAINTs bei einer Regelverletzung

```
SHOW TABLE t_adressen
NUMMER                          INTEGER Not Null
VORNAMEN                        VARCHAR(20) Nullable
NACHNAMEN                       VARCHAR(30) Not Null
STRASSE                         VARCHAR(25) Not Null
PLZ                             VARCHAR(6) Not Null
ORT                             VARCHAR(20) Not Null
CONSTRAINT INTEG_32:
  Primary key (NUMMER)
CONSTRAINT INTEG_33:
  Unique key (NACHNAMEN, STRASSE, PLZ)
```

Wie wir sehen, werden die (Primär- und Sekundär-) Schlüssel zu CONSTRAINTs, die von InterBase automatisch mit einem Namen versehen werden, in diesem Fall *integ_32* und *integ_33*; auch Referenzen auf andere Tabellen (FOREIGN KEY) und Prüfbedingungen (CHECK) werden auf diese Weise zu CONSTRAINTs.

Der Name des CONSTRAINTs (zu deutsch *Zwang*) wird bei Fehlermeldungen angezeigt (siehe Bild 14.2), aber auch dazu benötigt, um solche CONSTRAINTs wieder zu löschen.

```
ALTER TABLE t_adressen DROP CONSTRAINT integ_33
```

Das Thema ALTER TABLE werden wir gleich noch ausführlicher behandeln. Der Name *integ_33* ist nun nicht besonders anschaulich, insbesondere nicht bei Fehlermeldungen. Von daher besteht die Möglichkeit, mit dem Schlüsselwort CONSTRAINT einen eigenen Namen zu vergeben:

```
CREATE TABLE t_personen
  (nummer INTEGER NOT NULL,
  panr INTEGER NOT NULL,
  PRIMARY KEY (nummer),
  CONSTRAINT Personalausweisnummer UNIQUE (panr))
```

Wenn das Wort *Personalausweisnummer* dann in einer Fehlermeldung angezeigt wird, dann weiß man schon eher, was denn nun falsch sein könnte.

Fremdschlüssel

Mit Fremdschlüsseln wird eine Referenz auf eine andere Tabelle erstellt. Für gewöhnlich wird man auf den Primärschlüssel einer anderen Tabellen verweisen, und dieser wird normalerweise eine fortlaufende Nummer sein:

```
CREATE TABLE t_autor
  (nummer INTEGER NOT NULL,
  namen VARCHAR(30),
  PRIMARY KEY (nummer))

CREATE TABLE t_buch
  (nummer INTEGER NOT NULL,
  autor INTEGER,
  isbn VARCHAR (15),
  titel VARCHAR(30),
  PRIMARY KEY (nummer))
  FOREIGN KEY (autor) REFERENCES t_autor (nummer))
```

Hier im Beispiel gibt es eine Tabelle mit Autoren und eine Tabelle mit Büchern. Jeder Eintrag in der Tabelle *t_buch* hat einen Autor, dessen Name in der Tabelle *t_autor* steht. Die *autor*-Nummer in der Tabelle *t_buch* verweist auf einen Eintrag in der Tabelle *t_autor*, der anhand der Spalte *nummer* gesucht wird.

Sobald ein FOREIGN KEY definiert worden ist, stellt die Datenbank sicher, daß zu jeder *autor*-Nummer in *t_buch* ein Eintrag in *t_autor* existiert:

- Bevor ein Fremdschlüssel definiert wird, muß die betreffende Tabelle mit der Spalte (oder den Spalten) bereits existieren. In der Praxis ist es deshalb meist zweckmäßig, zunächst alle Tabellen ohne Referenzen zu erstellen und diese erst nachträglich einzufügen.
- Der Anwender kann keinen Datensatz aus der Tabelle löschen, auf den eine Referenz besteht, wenn dadurch Zuordnungen verloren gehen würden; genausowenig dürfen Änderungen an der oder den Spalte(n), die den Primärschlüssel bildet(n), vorgenommen werden, wenn dadurch die referentielle Integrität nicht mehr gewährleistet wäre.
- Es dürfen nur Werte in die Fremdschlüssel-Spalte(n) eingefügt werden (im Beispiel *autor*), die auch in der damit verbundenen Tabelle existieren. Gibt es dort den Autor *12345* nicht, dann kann auch kein Buch die *autor*-Nummer *12345* haben.

Es ist auch möglich, auf einen Sekundärschlüssel (UNIQUE) einer Tabelle eine Referenz zu erstellen. Ist der (Primär- und Sekundär-)Schlüssel einer Tabelle aus mehreren Spalten zusammengesetzt, dann müssen diese alle in den Fremdschlüssel aufgenommen werden:

```
CREATE TABLE t_abteilung
  (nummer INTEGER NOT NULL,
  namen VARCHAR(20),
  chef_vornamen VARCHAR(20),
  chef_nachnamen VARCHAR(30),
  chef_geburtsdatum DATE,
  chef_geburtsort VARCHAR(30),
  PRIMARY KEY (nummer),
  FOREIGN KEY (chef_vornamen, chef_nachnamen, chef_geburtsdatum,
      chef_geburtsort)
    REFERENCES t_mitarbeiter (vornamen, nachnamen, geburtsdatum,
      geburtsort))
```

Ich nehme an, das ist nun abschreckend genug gewesen – vergeben Sie lieber eine durchlaufende Nummer als Primärschlüssel. Meist gibt es dann auch überhaupt keinen Grund, einen Sekundärschlüssel für eine Referenz zu verwenden.

Zirkuläre Referenzen

Nicht immer ist es klug, die Fremdschlüssel-Spalten als NOT NULL zu definieren, insbesondere nicht, wenn zirkuläre Referenzen vorhanden sind. Dazu ein Beispiel:

```
CREATE TABLE t_mitarbeiter
  (nummer INTEGER NOT NULL,
  namen VARCHAR (30),
```

```
  abteilung INTEGER,
  PRIMARY KEY (nummer))

CREATE TABLE t_abteilung
  (nummer INTEGER NOT NULL,
  namen VARCHAR(20),
  chef INTEGER,
  PRIMARY KEY (nummer),
  FOREIGN KEY (chef) REFERENCES t_mitarbeiter (nummer))

ALTER TABLE t_mitarbeiter
  ADD FOREIGN KEY (abteilung) REFERENCES t_abteilung (nummer)
```

Nun versuchen Sie einmal, einen Datensatz einzugeben: Wenn Sie bei *t_mitarbeiter* eine Abteilungsnummer eingeben, dann beschwert sich die Datenbank, daß es die entsprechende Abteilung noch nicht gibt, soll in *t_abteilung* ein *chef* eingegeben werden, dann existiert die passende Mitarbeiternummer noch nicht. (In der englischsprachigen Literatur nennt man dies eine *deadlock situation*.)

Um hier trotzdem weiterzukommen, geht man hier folgendermaßen vor:

- Da es vermutlich mehr Mitarbeiter als Abteilungen gibt, werden zunächst alle Abteilungen eingegeben; in die Spalte *chef* trägt man NULL ein.
- Danach werden alle Mitarbeiter eingegeben; hier existieren die Abteilungen schon, so daß ein Eintrag in die Spalte *abteilung* kein Problem verursacht.
- Zuletzt werden bei allen Abteilungen die Spalte *chef* auf eine Mitarbeiternummer abgeändert.

Wenn nun beide Fremdschlüsselspalten auch noch als NOT NULL definiert worden sind, dann funktioniert dieser Trick nicht mehr (und es gilt »rien ne va plus«).

Tabellen-CHECKS

Hin und wieder möchte man Prüfbedingungen definieren, die nicht nur eine einzelne Spalte umfassen. Im nachfolgenden Beispiel soll sichergestellt werden, daß für eine Firmenadresse Straße oder Postfach angegeben wird, auch die Angabe von beiden Informationen soll erlaubt sein. Hier können dann nicht einzelne Spalten NOT NULL gesetzt werden.

```
CREATE TABLE t_firmen
  (nummer INTEGER NOT NULL,
  namen VARCHAR(30),
  strasse VARCHAR(30),
  s_plz  VARCHAR(6),
  postfach SMALLINT,
```

```
  p_plz VARCHAR(6),
  ort VARCHAR(20),
  PRIMARY KEY (nummer),
  CONSTRAINT adresse CHECK
    (((strasse IS NOT NULL) AND (s_plz IS NOT NULL))
    OR (postfach IS NOT NULL) AND (p_plz IS NOT NULL)))
```

Die nachfolgende Anweisung wäre beispielsweise nicht erlaubt, da hier Straße und Postfach fehlen:

```
INSERT INTO t_firmen (nummer, namen)
  VALUES (1, "TABU Licht- und Tontechnik")
```

Dagegen würde diese Anweisung akzeptiert:

```
INSERT INTO t_firmen (nummer, namen, postfach, p_plz, ort )
  VALUES (1, "TABU Licht- und Tontechnik", 31, "10 001", "Berlin")
```

Mit Tabellen-CHECKs können Sie fast alles überprüfen: Ob das Einstellungsdatum früher als das Kündigungsdatum ist, ob den verschiedenen Mitarbeitergruppen ein angemessenes Gehalt gezahlt wird, usw.

Noch eine Bemerkung zum vorigem Beispiel: Der Kürze und Übersichtlichkeit wegen wurde nur eine Spalte *ort* definiert; in der Praxis können Sie sich nie sicher sein, ob sich das Postfach nicht an einem anderen Ort befindet als das Werksgelände – verwenden Sie hier deshalb lieber zwei Spalten.

14.2.2 ALTER TABLE

Prinzipiell gibt es vier Möglichkeiten, um die Metadaten einer Tabelle zu ändern:

- Es werden neue Spalten hinzugefügt.
- Es werden Spalten gelöscht.
- Neue Gültigkeitsprüfungen werden hinzugefügt.
- Gültigkeitsprüfungen werden gelöscht.

Wie Sie dieser Aufstellung entnehmen, gibt es keine Möglichkeit, die Definition von Spalten oder CONSTRAINTs zu verändern, es bleibt Ihnen gegebenenfalls nur die Möglichkeit, die alte Tabelle oder Gültigkeitsprüfung zu löschen und geändert wieder zu erstellen.

Spalten löschen und hinzufügen

Da die Anweisungen zum Löschen und Hinzufügen von Spalten sehr einfach sind, soll gleich ein Beispiel verwendet werden, das eine bestehende Tabellen-

spalte umdefiniert. Nehmen wir einmal an, daß wir eine Tabelle mit Mitarbeiterdaten hätten, in der die Spalte Nachnamen 25 Zeichen breit ist:

```
CREATE TABLE t_mitarbeiter
  (nummer INTEGER NOT NULL,
  vornamen VARCHAR(25),
  nachnamen VARCHAR(25),
  abteilung INTEGER,
  PRIMARY KEY (nummer),
  FOREIGN KEY (abteilung) REFERENCES t_abteilung (nummer))
```

Irgendwann tritt einer derjenigen Fälle auf, in denen ein Doppelnamen nicht in die Tabelle paßt (und es sich der Namensträger auch energisch verbittet, seinen Namen in irgendeiner Weise abzukürzen). Die Spalte muß also verbreitert werden. Da beim Löschen einer Spalte auch alle Inhalte verloren gehen, müssen diese zunächst irgendwo zwischengespeichert werden:

```
ALTER TABLE t_mitarbeiter ADD temp_nachnamen VARCHAR(25);


UPDATE t_mitarbeiter SET temp_nachnamen = nachnamen;
```

Nun kann die alte Spalte gelöscht und die neue erstellt werden. Danach können die Daten wieder zurückkopiert und die Hilfsspalte kann gelöscht werden:

```
ALTER TABLE t_mitarbeiter DROP nachnamen;
ALTER TABLE t_mitarbeiter ADD nachnamen VARCHAR(35);
UPDATE t_mitarbeiter SET nachnamen = temp_nachnamen;
ALTER TABLE t_mitarbeiter DROP temp_nachnamen;
```

Während das Hinzufügen von Spalten eigentlich immer möglich ist (solange der Benutzer die entsprechenden Zugriffsrechte hat), gibt es Situationen, in denen das Löschen von Spalten nicht gestattet wird:

- Die Spalte ist Teil eines zusammengesetzten Schlüssels (UNIQUE; PRIMARY KEY oder FOREIGN KEY); in diesem Fall wäre zuerst der Schlüssel zu löschen.
- Die Spalte ist Teil einer zusammengesetzten Gültigkeitsprüfung (CHECK), hier müßte zunächst die Gültigkeitsprüfung gelöscht werden.
- Es können zwar Spalten gelöscht werden, welche den Primär- oder Sekundärschlüssel einer Tabelle bilden (solange es sich nicht um zusammengesetzte Schlüssel handelt), nicht aber dann, wenn diese Schlüssel für einen Fremdschlüssel benötigt werden. Auch dazu ein Beispiel:

```
CREATE TABLE t_namen
  (nummer INTEGER NOT NULL,
```

```
  namen VARCHAR(22),
  PRIMARY KEY (nummer))

INSERT INTO t_namen VALUES (1, "Ardt Adamson")

INSERT INTO t_namen VALUES (2, "Britta Borst")

CREATE TABLE t_rechnungen
  (nummer INTEGER NOT NULL,
  person INTEGER,
  datum DATE,
  betrag NUMERIC (8,2),
  PRIMARY KEY (nummer),
  FOREIGN KEY (person) REFERENCES t_namen (nummer))

INSERT INTO t_rechnungen VALUES (1, 2, "NOW", 12.34)

SELECT r.nummer, n.namen, r.datum, r.betrag
  FROM t_rechnungen r, t_namen n
  WHERE r.person = n.nummer

     NUMMER NAMEN                          DATUM       BETRAG
========== ====================== =========== ===========
          1 Britta Borst             5-DEC-1996       12.34

ALTER TABLE t_namen DROP nummer
```

Die letzte Anweisung (*DROP nummer*) würde vom DBMS mit der Fehlermeldung nach Bild 14.3 verweigert.

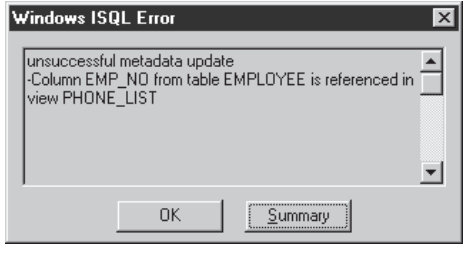


Bild 14.3: Fehlermeldung beim Versuch, eine Primärschlüssel-Spalte zu löschen, die für einen Fremdschlüssel benötigt wird

Beachten Sie auch, daß das Datum mit *"NOW"* eingegeben wurde, mit dieser Anweisung wird das jeweils aktuelle Datum eingegeben.

In einer ALTER TABLE-Anweisung können mehrere ADD- und DROP-Befehle gegeben werden, wie das nachfolgende Beispiel zeigt; dabei ist es auch möglich, Gültigkeitsprüfungen und Spalten gemeinsam zu löschen oder hinzuzufügen.

```
ALTER TABLE t_rechnungen
  DROP person,
  DROP datum,
  ADD bemerkung VARCHAR(30)
```

Suchen Sie bitte keinen besonderen Sinn in dieser Anweisung.

CONSTRAINTS löschen und hinzufügen

Im folgenden Beispiel soll ein Primärschlüssel gelöscht und wieder hinzugefügt werden. Damit es nicht gar zu langweilig wird, werden wir für eine kleine Komplikation sorgen.

```
CREATE TABLE t_ort
  (nummer INTEGER NOT NULL,
  namen VARCHAR(30),
  PRIMARY KEY(nummer))

INSERT INTO t_ort VALUES (1,"Altshausen")
```

Bevor mit ALTER TABLE der Primärschlüssel gelöscht wird, muß zunächst dessen (automatisch vergebener) Name ermittelt werden. Dazu wählt man VIEW | METADATA INFORMATION | TABLE.

```
SHOW TABLE t_ort
NUMMER                          INTEGER Not Null
NAMEN                           VARCHAR(30) CHARACTER SET ISO8859_1
Nullable
CONSTRAINT INTEG_12:
  Primary key (NUMMER)

ALTER TABLE t_ort DROP CONSTRAINT integ_12
```

Nun wollen wir die Gelegenheit nutzen, einen zweiten Datensatz mit der Nummer eins einzugeben. Beim nachfolgenden Versuch, wieder einen Primärschlüssel für diese Spalte zu definieren, reagiert das System mit einer Fehlermeldung.

```
INSERT INTO t_ort VALUES (1,"Bad Buchau")

ALTER TABLE t_ort ADD PRIMARY KEY(nummer)
```

So etwas kann übrigens böse Folgen haben: Nehmen wir einmal an, daß aus irgendwelchen Gründen der Primärschlüssel vorübergehend außer Kraft gesetzt wurde und daß sich in dieser Zeit doppelte Werte in die Datenbank eingeschmuggelt haben. Da nun der Primärindex nicht wieder in Kraft gesetzt werden kann, folgen laufend neue Datensätze deren Nummer eigentlich schon anderweitig vergeben wurde.

Irgendwann gibt es dann den Zustand, daß von 10 000 Datensätzen einer Tabelle 100 doppelte »Primärindex«-Werte haben, welche nun vom SYSDBA von Hand abgeändert werden müssen – dazu müssen Sie allerdings erst einmal gefunden werden. Nehmen wir an, es liegt nun folgende Tabelle vor (der Kürze halber mit der undramatischen Anzahl von sieben Datensätzen):

```
SELECT * FROM t_ort

     NUMMER   NAMEN
==========   ==============================
           1 Altshausen
           2 Bad Buchau
           3 Ebenweiler
           4 Fleischwangen
           4 Horgenzell
           5 Luegen
           5 Ebersbach
```

Nun muß man herausbekommen, welche Datensätze in der Tabelle doppelt sind. Dazu verwendet man folgende SELECT-Konstruktion:

```
SELECT nummer, namen FROM t_ort
  WHERE nummer IN (
    SELECT nummer FROM t_ort
      GROUP BY nummer
      HAVING COUNT(nummer) > 1)

     NUMMER   NAMEN
==========   ==============================
           4 Fleischwangen
           4 Horgenzell
           5 Luegen
           5 Ebersbach
```

Bei der SELECT-Anweisung wird mit einer Unterabfrage gearbeitet, welche die Nummern der Datensätze ermittelt, die mehr als einmal vorkommen. Weil dabei

mehrere Werte zurückgegeben werden, darf nicht *WHERE nummer = ( )* formuliert werden, sondern es muß der Mengenoperator IN verwendet werden.

Wenn Sie noch keine Datensätze in eine Tabelle eingefügt haben, dann läßt sich mit einer einzigen ALTER TABLE-Anweisung eine komplette Tabelle umdefinieren. Dies ist Beispielsweise dann nützlich, wenn Sie festgestellt haben, daß die verwendete Tabellendefinition nicht optimal ist, Sie aber nicht gleich die ganze Tabelle löschen möchten.

```
CREATE TABLE t_personen
  (vornamen VARCHAR(20) NOT NULL,
  nachnamen VARCHAR(20) NOT NULL,
  tel VARCHAR(15),
  CONSTRAINT Personenschlüssel PRIMARY KEY (vornamen, nachnamen))

ALTER TABLE t_personen
  DROP CONSTRAINT personenschlüssel,
  DROP nachnamen,
  ADD nachnamen VARCHAR(30) NOT NULL,
  ADD CONSTRAINT personenschlüssel PRIMARY KEY(vornamen, nachnamen)
```

14.2.3 DROP TABLE

Der Befehl zum Löschen einer Tabelle ist denkbar einfach:

```
DROP TABLE t_personen
```

In folgenden Fällen verhindert InterBase jedoch das Löschen von Tabellen:

- Wenn Transaktionen mit dieser Tabelle noch nicht abgeschlossen sind. Wenn Sie also seit der Erstellung der Tabelle beispielsweise eine INSERT-Anweisung durchgeführt haben, dann müssen Sie erst mit FILE | COMMIT WORK oder FILE | ROLLBACK WORK die Transaktion abschließen.
- Die Tabelle ist Ziel einer Referenz.
- Die Tabelle wird in einer VIEW, einem TRIGGER oder einer STORED PROCEDURE verwendet.
- Der Benutzer ist weder Besitzer der Tabelle noch SYSDBA. In diesem Fall ist sogar eine Änderung der Tabelle unmöglich.

14.2.4 Indizes

Wenn Sie bislang mit Paradox-Tabellen gearbeitet haben, dann werden Sie unter einem Index und einem Schlüssel ein und dasselbe verstehen. Bei InterBase ist dies nicht ganz der Fall.

Ein (Primär- oder Sekundär-)Schlüssel ist hier ein einzigartiger Wert (also ein Wert, der nicht ein zweites Mal in der Spalte auftaucht). Anhand des Schlüssels kann eine Referenz auf die Tabelle erstellt werden.

Ein Index dagegen ist eine Art Suchbaum, der es ermöglicht, einen Wert nicht nur über eine sequentielle Suche zu finden (also jeden Wert auslesen und vergleichen, ob er zutrifft), sondern anhand weniger Suchschritte feststellt, an welcher Adresse sich der Wert befindet.

Wird ein (Primär-, Sekundär- oder Fremd-)Schlüssel erstellt, so erstellt InterBase automatisch auch gleich einen entsprechenden Index. Es gibt allerdings auch die Möglichkeit, Indizes zu erstellen, die keine Schlüssel sind. Dies ist beispielsweise dann nötig, wenn die Werte in dieser Spalte nicht einzigartig sind.

Die Verwendung von Indizes soll die Suche nach Datensätzen beschleunigen. Wie schon bei Paradox-Tabellen habe ich allerdings keine Beschleunigung feststellen können.

Der Vollständigkeit halber sollen die entsprechenden Befehle hier trotzdem kurz vorgestellt werden; wie Sie sehen, müssen Indizes benannt werden und können auch über mehrere Spalten erstellt werden.

```
CREATE INDEX testadr_tel1 ON testadr (tel1)

CREATE INDEX mitarbeiter_namen ON mitarbeiter (vornamen, nachnamen)

DROP INDEX testadr_1
```

14.3 VIEWs

Eine VIEW ist im Prinzip eine vordefinierte Abfrage (mit SELECT), auf die wie auf eine Tabelle zuggriffen wird. Die Verwendung von VIEWs kann mehrere Gründe haben.

Zugriffsbeschränkung

Nehmen wir beispielsweise die Mitarbeitertabelle einer Firma:

```
CREATE TABLE t_mitarbeiter
  (nummer INTEGER NOT NULL,
  vornamen VARCHAR(20),
  nachnamen VARCHAR(30),
  durchwahl VARCHAR(3),
  gehalt DECIMAL (8,2),
  PRIMARY KEY(nummer))
```

Nun sollen zwar alle Mitarbeiter erfahren dürfen, welche Durchwahl denn der Kollege hat, dessen Gehalt geht sie aber überhaupt nichts an. Während man bei Desktop-Datenbanken für gewöhnlich zwei Tabellen definieren würde und den Zugriff auf eine einschränkt, definiert man bei Client-Server-Systemen eine VIEW:

```
CREATE VIEW v_mitarbeiter AS
  SELECT nummer, vornamen, nachnamen, durchwahl FROM t_mitarbeiter
```

Mit Hilfe von Zugriffsberechtigungen stellt man nun sicher, daß nur diejenigen Benutzer an die Tabelle *t_mitarbeiter* herankommen, welche auch wirklich auf das Gehalt zugreifen müssen; alle anderen erhalten nur Zugriff auf die VIEW.

Die Beschränkung auf einige Spalten nennt man auch *vertikale Teilmenge* (*vertical subset*) der betreffenden Tabelle. Ebenso ist es auch möglich, eine *horizonale Teilmenge* (*horizontal subset*) einer Tabelle zu bilden. Beispielsweise soll der Auszubildende aus der Lohnbuchhaltung nicht die Gehälter des Managements einsehen können:

```
CREATE VIEW v_mitarbeiter AS
  SELECT * FROM t_mitarbeiter
  WHERE gehalt < 10000
```

Selbstverständlich kann auch eine Kombination von horizontaler und vertikaler Teilmenge gebildet werden, beispielsweise dann, wenn das Management nicht für jeden Mitarbeiter telefonisch erreichbar sein möchte (in wieweit das mit modernen Management-Methoden in Einklang zu bringen ist, soll uns hier nicht interessieren):

```
CREATE VIEW v_mitarbeiter AS
  SELECT nummer, vornamen, nachnamen, durchwahl FROM t_mitarbeiter
  WHERE gehalt < 10000
```

JOINS von Tabellen

Ein weiterer Grund für die Verwendung von VIEWs könnte sein, daß man damit unauffällig zwei Tabellen zusammenfügen möchte:

```
CREATE VIEW v_offenpo (namen, straße, ort, nummer, datum, betrag) AS
  SELECT a.vornamen || " " || a.nachnamen, a.straße,
    a.plz || " " || a.ort, o.nummer, o.datum, o.betrag
  FROM adresse a, offenpo o
  WHERE a.nummer = o.kunde
```

In diesem Beispiel wird die Tabelle der offenen Posten mit der Kundendatei verknüpft. Prinzipiell könnte man dies auch in der entsprechenden Abfrage, doch es gibt Gründe, auf die hier beschriebene Weise vorzugehen:

- Bei komplizierteren Abfragen wird der Anwender den SQL-Text selbst erstellen müssen. Hier kann ihm viel Schreibarbeit (mit der entsprechenden Gefahr eines Tippfehlers) abgenommen werden, wenn ein Großteil der Abfrage quasi schon durch die VIEW vorgegeben ist. Vor allem besteht nicht die Gefahr, daß der Anwender von der Buchhaltung sich nicht mehr genau an die Spaltennamen erinnert und (ISQL steht im vielleicht nicht zur Verfügung) kurzerhand die Anweisung *SELECT * FROM adresse* eingibt, worauf ihm mal schnell alle Kundenadressen über das Netzwerk geschickt werden.
- In diesen Beispiel vielleicht weniger wahrscheinlich, aber immerhin vorstellbar wäre es, daß man bestimmten Personen innerhalb des Betriebs keinen Einblick in die komplette Kundentabelle gewähren möchte.

14.3.1 Eine VIEW erstellen

Um eine VIEW zu erstellen, wird der Befehl *CREATE VIEW v_namen AS SELECT...* verwendet. Die SELECT-Anweisung wird genauso erstellt wie eine übliche SELECT-Anweisung; das einzige, was nicht funktioniert, ist die Spaltenumbenennung mit AS – dieses Schlüsselwort hat nun einmal hier eine andere Funktion. Die folgende Anweisung wird also eine Fehlermeldung verursachen:

```
CREATE VIEW v_adressen AS
  SELECT a.vornamen || " " || a.nachnamen AS namen
  FROM adresse a
```

Hier muß schon im Kopf der VIEW-Anweisung eine Aufzählung der Spalten erfolgen (ein ausführlicheres Beispiel auf der vorigen Seite):

```
CREATE VIEW v_adressen (namen) AS
  SELECT a.vornamen || " " || a.nachnamen
  FROM adresse a
```

14.3.2 Zugriffsmodus

Auf eine VIEW kann nicht nur mit einer SELECT-Abfrage zugegriffen werden, es funktionieren auch die Befehle INSERT, UPDATE und DELETE, wenn auch manchmal nur eingeschränkt. Dazu ein Beispiel:

```
CREATE TABLE t_test2
  (nummer INTEGER NOT NULL,
  vornamen VARCHAR(20),
  nachnamen VARCHAR(30) NOT NULL,
  PRIMARY KEY (nummer))
```

```
INSERT INTO t_test2 VALUES (1, "Michael", "Mustermann")

INSERT INTO t_test2 VALUES (2, "Gabi", "Mustermann")

INSERT INTO t_test2 VALUES (3, "Horst", "Mustermann")

CREATE VIEW v_test2 AS
  SELECT nummer, vornamen FROM t_test2

INSERT INTO v_test2 VALUES (4, "Claudia")
INSERT INTO v_test2 (nummer, vornamen, nachnamen)
  VALUES (4, "Claudia", "Maier")
```

Diese beiden INSERT-Anweisungen funktionieren natürlich nicht. Bei einer INSERT-Operation in eine VIEW werden die entsprechenden Spalten der Tabelle, welche nicht in der VIEW vorhanden sind, mit NULL-Werten versehen. Da hier allerdings die Spalte *nachnamen* als NOT NULL definiert worden ist, wird die INSERT-Anweisung mit der entsprechenden Fehlermeldung abgelehnt.

Der Versuch, mit der zweiten Anweisung sich um dieses Problem herumzumogeln scheitert ebenfalls, da die ausgeschlossenen Spalten nicht etwa bloß versteckt, sondern tatsächlich nicht vorhanden sind.

```
DELETE FROM v_test2 WHERE nummer = 2

UPDATE v_test2 SET vornamen = "Yvonne" WHERE nummer = 3
```

Dagegen funktionieren die DELETE- und UPDATE-Anweisungen problemlos. Um nun zu verhindern, daß über die Durchwahlnummerntabelle plötzlich Mitarbeiterdaten verschwinden oder abgeändert werden (und sei es aus Unkenntnis oder aus Versehen), dann sollten Sie dafür nur SELECT-Rechte vergeben. (Sonst könnte es vorkommen, daß man sich im Mahnwesen sagt, daß man die Leute von der Konstruktion sowieso nie anruft und die entsprechenden Nummern aus der Liste löscht. Darauf bekommen zwanzig Ingenieure kommentarlos kein Gehalt mehr und vermuten die Firma in erheblichen Liquiditätsschwierigkeiten – bis das Problem erkannt wird, hat sich die Hälfte davon nach einer neuen Stelle umgesehen.)

INSERT-, UPDATE- und DELETE-Anweisungen sind nicht bei allen VIEWs möglich. So sind sie (laut Handbuch) nur dann erlaubt, wenn auf eine einzige Tabelle (oder eine VIEW, die ihrerseits wiederum auf eine einzige Tabelle zugreift) zugegriffen wird.

Auch sind sie beispielsweise dann nicht erlaubt, wenn in der Abfrage eine Aggregatfunktion verwendet wird (was sollte InterBase auch tun, wenn der Be-

nutzer zum Beispiel die Zahl der offenen Posten von 14 auf 27 erhöhen möchte?). Auch wenn der Ersteller der VIEW nur SELECT-Privilegien für eine Tabelle hat, wird man mit der VIEW keine Daten ändern können.

Die CHECK OPTION

Das nächste Problem schildere ich gleich anhand eines Beispiels:

```
CREATE VIEW v_test2 AS
  SELECT nummer, vorname, nachname FROM t_test2
  WHERE nummer < 10

INSERT INTO v_test2 VALUES (11, "Eugen", "Endres")

SELECT * FROM v_test2

     NUMMER VORNAME              NACHNAME
========== ==================== ==============================
          1 Michael               Mustermann
          3 Yvonne                Mustermann
```

Sie erstellen eine VIEW mit irgendeiner Bedingungen (in unserem Beispiel muß die Nummer kleiner zehn sein) und fügen einen Wert ein, der diese Regel verletzt. InterBase akzeptiert dies ohne weiteres, doch in der VIEW ist dieser Wert dann nicht vorhanden.

Dadurch können solche Situationen entstehen, die durchschnittlich informierte Anwender bisweilen zur Verzweiflung treiben: Irgendeine *TTable*- oder *TQuery*-Komponente ist mit einer VIEW verbunden, für den Anwender sieht dies nach einer gewöhnlichen Tabelle aus, und in die gibt er nun einen Datensatz ein – der dann aber nicht angezeigt wird. Da sich inzwischen die Erkenntnis durchgesetzt hat, daß »Computer auch bloß Menschen« sind, wiederholt er die Eingabe, und das DBMS beschwert sich, daß es unter diesem Primärschlüssel schon einen Eintrag gibt. Nun versucht der Anwender, diesen Eintrag zu finden: Vielleicht ist er ja falsch einsortiert, vielleicht muß man in diesem speziellen Fall die Applikation beenden und neu starten, vielleicht sollte man etwas am Netzwerkkabel rütteln oder mit der flachen Hand gegen das Rechnergehäuse schlagen; ein Kollege will schon immer gewußt haben, daß zu solch' einer Anlage eine unterbrechungsfreie Spannungsversorgung gehört ...

```
SELECT * FROM t_test2

     NUMMER   VORNAME              NACHCHNAME
==========   ==============       ==============================

          1  Michael              Mustermann
         11  Eugen                Endres
          3  Yvonne               Mustermann

SELECT * FROM t_test2
  ORDER BY nummer
     NUMMER   VORNAME              NACHNAME
==========   =================    ==============================

          1  Michael              Mustermann
          3  Yvonne               Mustermann
         11  Eugen                Endres
```

Wie diese Abfragen beweisen, wurde der Datensatz tatsächlich eingefügt, aber wegen der WHERE-Klausel eben nicht angezeigt. Wie sich hier außerdem zeigt, ist es übehaupt nicht sichergestellt, daß die Datensätze in einer bestimmten Reihenfolge aufgelistet werden, es sei denn, dies würde mit ORDER BY explizit so gewünscht.

```
CREATE VIEW v_test2 AS
  SELECT nummer, vornamen, nachnamen FROM t_test2
  WHERE nummer < 10
  WITH CHECK OPTION

INSERT INTO v_test2 VALUES (12, "Emelie", "Endres")
```

Um Irritationen des Anwenders zu vermeiden, sollte man die Anweisung WITH CHECK OPTION geben, dadurch wird das Einfügen von Datensätzen unterbunden, welche nicht den Bedingungen entsprechen. Die INSERT-Anweisung im Beispiel wird dadurch abgelehnt.

14.3.3 Eine VIEW löschen

Um eine VIEW zu löschen, verwendet man den Befehl DROP VIEW. Sind vorher DML-Anweisungen auf diese VIEW erfolgt, müssen sie vor dem Löschen mit COMMIT oder ROLLBACK abgeschlossen werden.

```
DROP VIEW v_adressen
```

14.4 STORED PROCEDUREs

Es gehört zur Philosopie von Client-Server-Systemen, daß Anfragen auf dem Server bearbeitet werden und nur die Ergebnisse zum Client geschickt werden. Nun gibt es allerdings Situationen, in denen die normalen SQL-Befehle nicht ausreichen, in denen beispielsweise Schleifen und Verzweigungen benötigt werden.

Prinzipiell würde die Möglichkeit bestehen, daß man diese Anweisungen in der Programmiersprache der Datenbankanwendung verfaßt (in unserem Beispiel in Pascal) und auf dem Client durchführen läßt. Dies würde aber der Philisophie von Client-Server-Systemen zuwiderlaufen (und daß man dies bei Client-Server-Systemen so und nicht anders macht, hat durchaus handfeste Gründe, wir werden gleich darauf kommen).

Von daher bieten Client-Server-Systeme die Möglichkeit der STORED PROCEDURES. Das sind Prozeduren, die als Metadaten auf dem Server gespeichert werden und vom DBMS ausgeführt werden. STORED PROCEDUREs können von der Datenbankanwendung sowohl Parameter übernehmen als auch solche an sie übergeben.

Diese Vorgehensweise bietet gegenüber von Prozeduren, die in der Datenbankanwendung vorhanden sind, folgende Vorteile:

- Über das Netzwerk werden nur die benötigten Daten geschickt.
- Ähnliche Anfragen werden einheitlich behandelt. Zum Verständnis ein kleines Beispiel: Nehmen wir einmal an, eine Prozedur berechnet die Kosten für ein Projekt. In den verschiedenen Anwendungsprogrammen gibt es jedoch verschiedene Prozeduren, die einige Teilaspekte jeweils ganz anders behandeln.

  Nun soll in einer Konferenz beschlossen werden, ob das Projekt durchgeführt werden soll oder nicht, und in dieser Konferenz kursieren dann drei verschiedene Zahlen die Kosten des Projekts betreffend. Das erleichtert nicht gerade die Diskussion, zumal jeder der Überzeugung ist, daß die eigenenen Zahlen die richtigen sind.

  Wären die Kosten des Projektes von einer STORED PROCEDURE ermittelt worden und die Zahlen dann an die jeweiligen Anwendungen weitergeleitet worden, wären zwangsläufig auch dieselben Ergebnisse vorhanden.
- Eine STORED PROCEDURE läßt sich einfacher ändern. Angenommen, die Regierung verändert (erhöht, alles andere ist höchst unwahrscheinlich) mal wieder die Mehrwertsteuer, die in einer Prozedur als Konstante verwendet wird, dann müßte die Datenbankanwendung überarbeitet und neu kompiliert werden. Das ist aufwendig, und wenn die Datenbankanwendung von einem Fremdanbieter stammt, wird es meist auch noch teuer. Eine STORED PROCEDURE läßt sich dagegen mit ein paar SQL-Anweisungen ändern.

14.4.1 Ein Beispiel

Bevor wir uns die Details ansehen, wollen wir uns ein Beispiel einer STORED PROCEDURE ansehen, damit verständlich wird, wo die Vorteile einer solchen Konstruktion liegen. Ausgangsbasis ist die Tabelle *department*. Die Tabelle finden Sie auch in der Datenbank *employee.gdb*, die mit Delphi ausgeliefert wird – hier im Beispiel wurden aus Gründen der Übersichtlichkeit einige DOMAINs beseite gelassen:

```
CREATE DOMAIN d_deptno AS CHAR(3)
  CHECK (VALUE = '000' OR (VALUE > '0' AND VALUE <= '999')
    OR VALUE IS NULL);

CREATE TABLE department
  (dept_no d_deptno NOT NULL,
  department VARCHAR(25) NOT NULL,
  head_dept d_deptno,
  mngr_no SMALLINT,
  budget NUMERIC(15,2),
  location VARCHAR(15),
  UNIQUE (department),
  PRIMARY KEY (dept_no));
```

Die relevanten Spalten finden Sie in der folgenden Abfrage (gekürzt):

```
SELECT dept_no, department, head_dept, budget
  FROM department

DEPT_NO DEPARTMENT               HEAD_DEPT                  BUDGET
======= ====================     ========= ======================
000  Corporate Headquarters      <null>                1000000.00
100  Sales and Marketing         000                   2000000.00
600  Engineering                 000                   1100000.00
900  Finance                     000                    400000.00
180  Marketing                   100                   1500000.00
620  Software Products Div.      600                   1200000.00
621  Software Development        620                    400000.00
622  Quality Assurance           620                    300000.00
623  Customer Support            620                    650000.00
670  Consumer Electronics Div.   600                   1150000.00
671  Research and Development    670                    460000.00
672  Customer Services           670                    850000.00
```

```
130  Field Office: East Coast     100                    500000.00
140  Field Office: Canada         100                    500000.00
110  Pacific Rim Headquarters     100                    600000.00
```

Um das Budget einer Abteilung zu ermitteln, langt eine simple Abfrage. Nun könnte aber interessieren, wie hoch die Summe der Budgets dieser Abteilung inklusive aller ihr direkt unterstellten Unterabteilungen ist – auch dies ist weiter kein Problem:

```
SELECT SUM( budget) FROM department
  WHERE (head_dept = 100) OR (dept_no = 100)
```

Der Abteilung *100* untersteht unter anderem die Abteilung *130*, der wiederum unterstehen die Abteilungen *115* und *116* – und diese Ebene wird nun nicht mehr von dieser Abfrage erfaßt. Nun werden bei so einer Summe nicht nur direkt, sondern auch die indirekt unterstellten Abteilungen interessieren, und dies ist mit einer SELECT-Abfrage nun ein wenig aufwendig zu realisieren (um es einmal vorsichtig zu formulieren).

Deshalb wird dafür die folgende STORED PROCEDURE eingesetzt, welche rekursiv ermittelt, welche unterstellten Abteilungen es jeweils gibt, und deren Budgets aufaddiert:

```
CREATE PROCEDURE p_budget (dno CHAR(3))
RETURNS (tot NUMERIC (15,2))
AS
  DECLARE VARIABLE sumb DECIMAL(12, 2);
  DECLARE VARIABLE rdno CHAR(3);
  DECLARE VARIABLE cnt INTEGER;
BEGIN
  SELECT budget FROM department
    WHERE dept_no = :dno INTO :tot;

  SELECT count(budget) FROM department
    WHERE head_dept = :dno INTO :cnt;
  IF (cnt = 0) THEN
    SUSPEND;

  FOR
    SELECT dept_no
      FROM department
      WHERE head_dept = :dno
      INTO :rdno
  DO
```

```
  BEGIN
    EXECUTE PROCEDURE dept_budget :rdno RETURNING_VALUES :sumb;
    tot = tot + sumb;
  END

  SUSPEND;
END
```

Wie Sie sicher nicht anders erwartet haben, beginnt die Definition mit CREATE PROCEDURE und einem Namen. Des weiteren findet man im Prozedurkopf die Parameter, die an die PROCEDURE übergeben werden (hier *dno*, »department-number«) sowie die Parameter, welche die PROCEDURE zurückgibt (hier *tot*, »Total- oder Gesamtbudget«). Es ist hier möglich, sowohl mehrere Parameter zu übergeben als auch zurückzunehmen.

Mit dem Schlüsselwort AS beginnt der Anweisungsteil. Zunächst werden einige Variablen deklariert, die eigentlichen Anweisungen befinden sich zwischen einem BEGIN und einem END. Im Gegensatz zu C, aber ähnlich wie in Pascal müssen die Variablen am Anfang der Prozedur deklariert werden.

Eine SELECT-Anweisung sieht unter diesen Umständen ein wenig anders aus: Zunächst werden wie gewohnt Spalte und Tabelle angegeben, gegebenenfalls ergänzt durch beispielsweise eine WHERE-Klausel. In der ersten SELECT-Anweisung wird das Budget der Abteilung ermittelt, deren Abteilungsnummer (*dept_no*) der STORED PROCEDURE als Parameter übergeben wurde. Das Ergebnis wird mit INTO der Variablen *tot*, also dem Rückgabeparameter zugewiesen.

Mit der nächsten SELECT-Anweisung wird ermittelt, wie viele Abteilungen der aktuell gewählten Abteilung unterstehen. Unterstehen dieser Abteilung keine weiteren Abteilungen, dann muß deren Budget auch nicht ermittelt werden, die STORED PROCEDURE kann an dieser Stelle mit SUSPEND beendet werden.

Für den Fall, daß es unterstellte Abteilungen gibt, wird eine FOR..SELECT..DO-Schleife erstellt, welche die einzelnen Unterabteilungen durchläuft. Hier wird dann die STORED PROCEDURE rekursiv aufgerufen und ermittelt das Budget der jeweiligen Unterabteilung sowie die Budgets all der Abteilungen, die wiederum dieser unterstellt sind – die jeweiligen Ergebnisse werden im Rückgabeparameter *tot* aufaddiert. Die Anweisung SUSPEND wird zum Übertragen der Ergebnisse benötigt, wir klären das noch.

14.4.2 Übersicht über die Prozedur-Sprache

Neben den originären SQL-Anweisungen (DDL und DML) gibt es noch eine Reihe von Befehlen, die den STORED PROCEDUREs und den TRIGGERs vorbehalten sind.

FOR SELECT...DO

Mit FOR SELECT...DO läßt sich eine Schleife bilden, welche alle Datensätze einer Tabelle durchläuft. Damit lassen sich nicht nur, wie im vorigen Beispiel, Werte aus Tabellen auslesen, es lassen sich auch in der Schleife Änderungen in die Tabelle schreiben:

```
CREATE PROCEDURE p_adr2
AS
  DECLARE VARIABLE lang VARCHAR(15);
  DECLARE VARIABLE dat INTEGER;
BEGIN
  FOR
    SELECT nummer, tel
    FROM t_adr2
    INTO :dat, :lang
  DO
  BEGIN
    lang = "030 / " || LANG;
    UPDATE t_adr2 SET tel = :lang WHERE nummer = :dat;
  END
END
```

Diese STORED PROCEDURE ist für den Fall erstellt, daß die Berliner Filiale hin und wieder eine Diskette mit Adressen schickt, bei denen konsequent die Telefonnummern ohne Vorwahl aufgeführt sind. Die Prozedur durchläuft nun in einer FOR SELECT...DO-Schleife sämtliche Tabellenreihen und fügt vor jede Nummer den String *030 /* ein.

Diese STORED PROCEDURE ist ein hübsches Beispiel dafür, wie man etwas einfaches auch kompliziert machen kann: Das gleiche Ergebnis erhält man nämlich auch mit folgender Anweisung:

```
UPDATE t_adr SET tel = "030 / " || tel
```

Die SELECT-Anweisung

Eine SELECT-Anweisung muß natürlich nicht in einer Schleife verwendet werden – insbesondere dann, wenn Aggregatfunktionen ermittelt werden, macht dies meist auch nicht viel Sinn. Die Syntax bleibt in diesem Fall dieselbe:

```
SELECT SUM(budget), AVG(budget)
  FROM department
  WHERE head_dept = :head_dept
  INTO :sum_budget, :avg_budget
```

Die Variablen hinter INTO müssen in derselben Reihe aufgeführt werden wie die Spalten oder Aggregatfunktionen hinter SELECT.

Die WHILE...DO-Anweisung

Für herkömmliche Schleifen verwendet man die WHILE...DO-Anweisung (um eine FOR...TO...DO-Schleife zu simulieren, müßte man zunächst eine Tabelle mit durchlaufenden Nummern erstellen; dies wäre durchaus möglich, aber doch ein wenig umständlich).

```
CREATE PROCEDURE p_fakult (x INTEGER)
  RETURNS (f INTEGER)
AS
BEGIN
  f = 1;
  WHILE (x > 0) DO
  BEGIN
    f = f * x;
    x = x - 1;
  END
  SUSPEND;
END
```

Dieses Beispiel berechnet mit einer Schleife die Fakultät einer Zahl. Alternativ wäre es möglich, zu diesem Zweck eine rekursive Prozedur zu verwenden, wie das Beispiel im nächsten Abschnitt zeigt:

Verzweigungen mit IF...THEN...ELSE

Verzweigungen werden mit IF...THEN ...ELSE-Anweisungen realisiert:

```
ALTER PROCEDURE p_fakult (x INTEGER)
  RETURNS (f INTEGER)
AS
  DECLARE VARIABLE num INTEGER;
BEGIN
  IF (x=1) THEN   f = 1;
  ELSE
  BEGIN
    num = :x - 1;
    EXECUTE PROCEDURE p_fakult :num RETURNING_VALUES :f;
    f = :f * x;
  END
  SUSPEND;
END
```

Auch hier wird wieder die Fakultät einer Zahl berechnet. Über eine Rekursion berechnet die Prozedur die Fakultät einer Zahl als Produkt der Zahl multipliziert mit der Fakultät der nächstkleineren Zahl. Vergessen Sie nicht die Abbruchbedingung!

EXIT und SUSPEND

Was Einsteigern immer wieder Probleme bereitet, ist der Unterschied zwischen EXIT und SUSPEND; dazu die folgenden Beispiele:

```
CREATE PROCEDURE p_liste RETURNS (r INTEGER)
AS
BEGIN
  r = 0;
  WHILE (r < 5) DO
  BEGIN
    r = r + 1;
    SUSPEND;
  END
END

SELECT * FROM p_liste

           R
============
           1
           2
           3
           4
           5
```

Mit dem Befehl SUSPEND wird das aktuelle Ergebnis an die SELECT-Anweisung übergeben und mit dem nächsten Schleifendurchlauf begonnen.

```
ALTER PROCEDURE p_liste RETURNS (r INTEGER)
AS
BEGIN
  r = 0;
  WHILE (r < 5) DO
  BEGIN
    r = r + 1;
    SUSPEND;
    IF (r = 3) THEN EXIT;
  END
END
```

```
SELECT * FROM p_liste

           R
===========

           1
           2
           3
```

Wird dagegen eine EXIT-Anweisung eingefügt, dann werden die Schleife und die Prozedur an dieser Stelle abgebrochen.

```
EXECUTE PROCEDURE p_liste

           R
===========
           1
```

Die Anweisung EXECUTE PROCEDURE ist nicht dafür ausgelegt, mehrere Datensätze zurückzugeben. Hier werden dann mit der SUSPEND-Anweisung praktisch die Schleife und die Prozedur abgebrochen.

Interessant ist auch, was passiert, wenn aus der Prozedur die SUSPEND-Anweisung entfernt wird: Während bei EXECUTE PROCEDURE kein Unterschied auftritt (es wird nach wie vor nur der erste Wert zurückgegeben), wird bei einer SELECT-Anweisung überhaupt kein Wert zurückgegeben. (Fazit: Wenn bei einer SELECT-Anweisung auf eine Prozedur kein Ergebnis zurückgegeben wird, aber auch keine Fehlermeldung ausgegeben wird, dann fehlt vermutlich eine SUSPEND-Anweisung.)

Variablen und Parameter

In einer Prozedur gibt es folgende Variablen:

- Die lokalen Variablen, welche nur innerhalb der Prozedur gültig sind. Sie werden zu Beginn der Prozedur zwischen AS und BEGIN deklariert.
- Die Eingangs-Parameter, die nach dem Prozedur-Namen aufgelistet sind.
- Die Ausgangs- oder Rückgabe-Parameter, die nach dem Schlüsselwort RETURNS aufgeführt werden.

```
CREATE PROCEDURE p_test
  (eingangsparameter1 INTEGER, eingangsparameter2 VARCHAR(20))
  RETURNS (ausgangsparameter1 INTEGER,
    ausgangsparameter2 VARCHAR(20))
AS
  DECLARE VARIABLE lokalevariable1 INTEGER;
  DECLARE VARIABLE lokalevariable2 VARCHAR(20);
```

```
BEGIN
  ...
END
```

Innerhalb des Anweisungsteils werden alle Variablen und Parameter gleich behandelt.

14.4.3 Einsatz von STORED PROCEDUREs

Um eine STORED PROCEDURE aufzurufen, gibt es prinzipiell zwei Möglichkeiten:

- Soll die STORED PROCEDURE Datenänderungen durchführen (INSERT; UPDATE, DELETE), dann wird Sie mit EXECUTE PROCEDURE ausgeführt.
- Soll die Prozedur mehrere Werte zurückgeben, dann muß Sie mit einer SELECT-Anweisung ausgeführt werden. Soll nur ein Wert zurückgegeben werden, dann funktionieren beide Möglichkeiten.

Bisweilen sollen die Reihen einer Tabelle und die Ergebnisse einer STORED PROCEDURE gemeinsam angezeigt werden. Hier muß dann häufig der Umweg über eine zweite Prozedur gegangen werden, wie im folgenden Beispiel gezeigt, das einzig und allein der Anschauung und nichts anderem dienen soll:

```
CREATE PROCEDURE p_faktab
  RETURNS (nummer INTEGER, fakultaet INTEGER,
    vornamen VARCHAR(20), nachnamen VARCHAR(30))
AS
BEGIN
  FOR
  SELECT nummer, vornamen, nachnamen
    FROM t_test2
    INTO :nummer, :vornamen, :nachnamen
  DO
  BEGIN
    EXECUTE PROCEDURE p_fakult :nummer
      RETURNING_VALUES :fakultaet;
    SUSPEND;
  END
END

SELECT * FROM p_faktab
```

```
     NUMMER    FAKULTAET     VORNAMEN              NACHNAMEN
========== ==========     ==================== ====================

          1            1    Michael              Mustermann
         11     39916800    Eugen                Endres
          3            6    Yvonne               Mustermann
```

14.4.4 Ändern und Löschen von STORED PROCEDUREs

Um eine STORED PROCEDURE zu löschen, wird die Anweisung DROP PROCEDURE verwendet.

```
DROP PROCEDURE p_faktab
```

Eine Prozedur kann nur gelöscht werden, wenn sie nicht von anderen Prozeduren, von TRIGGERn oder von VIEWs verwendet wird. Wenn bei einer Prozedur nur eine leichte Änderung notwendig ist (beispielsweise, weil sich herausgestellt hat, daß sie unter bestimmten Bedingungen nicht stabil arbeitet), dann ist es recht uneffektiv, diese zu löschen und neu zu erstellen, wenn dabei erst einige andere Prozeduren oder ähnliches gelöscht werden müßten.

Von daher besteht die Möglichkeit, eine Prozedur mit ALTER PROCEDURE zu ändern; dabei handelt es sich im Prinzip um eine CREATE PROCEDURE-Anweisung, mit deren Hilfe eine Prozedur desselben Namens überschrieben wird.

14.5 TRIGGER

Ein TRIGGER ist im Prinzip das, was bei C++Builder eine Ereignisbehandlungsroutine ist: Eine Folge von Anweisungen – aus Standard-SQL-Anweisungen und den STORED PROCEDURE-Befehlen –, welche dann aufgerufen wird, wenn ein bestimmtes Ereignis eintritt.

14.5.1 Ein Beispiel

Wie schon mehrmals erläutert wurde, gibt es bei InterBase keine selbstinkrementierenden Felder. Mit der Hilfe von TRIGGERn besteht jedoch die Möglichkeit, sich selbst so etwas zurechtzubasteln (was allerdings etwas aufwendiger ist, als bei der Tabellendefinition einfach AUTOINC anzugeben).

```
CREATE TABLE t_namen
  (nummer INTEGER NOT NULL,
  namen VARCHAR(20),
  PRIMARY KEY (nummer))
```

```
CREATE GENERATOR g_namen

CREATE TRIGGER e_namen FOR t_namen
BEFORE INSERT
AS
BEGIN
  NEW.nummer = GEN_ID(g_namen, 1);
END
```

Einem TRIGGER muß zunächst einmal ein Name gegeben werden, anhand dessen man ihn identifizieren, ändern und löschen kann. Nachdem der Präfix *t_* schon für Tabellen verwendet wird, habe ich hier *e_* (wie Ereignis) verwendet.

Nach dem Schlüsselwort FOR wird bestimmt, bei welcher Tabelle der TRIGGER ausgelöst werden soll, BEFORE INSERT ist das Ereignis, auf das der TRIGGER reagiert. (Es ist hier übrigens zwingend, BEFORE zu verwenden, weil sich InterBase ansonsten über einen NULL-Wert beschwert.)

Mit den Kontext-Variablen OLD und NEW können Sie auf die Spalten der Tabelle zugreifen. Während bei einer INSERT-Anweisung nur NEW-Variablen von Belang sind, kann mit OLD-Variablen bei UPDATE-Anweisungen beispielsweise daraufhin geprüft werden, ob sich der Inhalt einer Spalte geändert hat.

In die Spalte *nummer* wird mit dieser Anweisung der um eins erhöhte Generatorwert eingefügt. Bei künftigen INSERT-Anweisungen braucht man sich dann keine Gedanken mehr um die Spalte *nummer* zu machen.

```
INSERT INTO t_namen (namen) VALUES ("Caesar")

INSERT INTO t_namen (namen) VALUES ("Daniel")

SELECT * FROM t_namen

     NUMMER NAMEN
========== ====================
         1 Caesar
         2 Daniel
```

14.5.2 Vergleich alter und neuer Spaltenwerte

Die Anweisungen, welche bei TRIGGERn verwendet werden können, gleichen im wesentlichen denen, die auch bei STORED PROCEDUREs verwendet werden. Eine Ausnahme bilden die Kontext-Variablen OLD und NEW, mit denen auf die bisherigen und zukünftigen Spaltenwerte zugegriffen werden kann.

Dies wollen wir im nächsten Beispiel dazu verwenden, um bei einer Liste mit den Gehältern der Mitarbeiter eine separate Spalte einzuführen, welche anzeigt, um welchen Prozentsatz das Gehalt bei der letzten Anpassung verändert wurde:

```
CREATE TABLE t_gehalt
  (angestellter INTEGER NOT NULL,
  gehalt NUMERIC (7,2),
  letzte_aenderung NUMERIC (4,2))

INSERT INTO t_gehalt VALUES (1, 3400, 100)
INSERT INTO t_gehalt VALUES (2, 2800, 100)

CREATE TRIGGER e_gehalt FOR t_gehalt
BEFORE UPDATE
AS
BEGIN
  IF (OLD.gehalt <> NEW.gehalt)
    THEN NEW.letzte_aenderung = ((NEW.gehalt - OLD.gehalt) * 100 /
    OLD.gehalt);
END

UPDATE t_gehalt SET gehalt = 5000 WHERE angestellter = 1

SELECT * FROM t_gehalt

ANGESTELLTER      GEHALT LETZTE_AENDERUNG
============ =========== ================
           1     5000.00            56.25
           2     2800.00           100.00
```

14.5.3 TRIGGER ändern und löschen

Ein TRIGGER kann wie eine STORED PROCEDURE als Ganzes überschrieben werden, indem man die Anweisung ALTER TRIGGER verwendet. Darüber hinaus – und dies ist bei der STORED PROCEDURE nicht erlaubt – können Änderungen im Prozedurkopf durchgeführt werden.

Zum einen ist es dabei möglich, den TRIGGER zu deaktivieren (und danach wieder zu aktivieren), wobei die folgenden beiden Anweisungen verwendet werden:

```
ALTER TRIGGER e_gehalt INACTIVE
ALTER TRIGGER e_gehalt ACTIVE
```

Zum anderen ist es möglich, den Zeitpunkt des Auslösens zu spezifizieren:

```
ALTER TRIGGER e_gehalt AFTER INSERT
```

Auslösen des TRIGGERs

Das Auslösen eines TRIGGERS kann an sechs verschiedene Ereignisse gekoppelt werden:

- BEFORE INSERT
- AFTER INSERT
- BEFORE UPDATE
- AFTER UPDATE
- BEFORE DELETE
- AFTER DELETE

Wird ein TRIGGER an das Ereignis BEFORE DELETE geknüpft, so kann damit das Löschen eines Datensatzes unterbunden werden:

```
CREATE EXCEPTION  x_gehalt "Datensatz darf nicht gelöscht werden"

CREATE TRIGGER e_gehalt2 FOR t_gehalt
BEFORE DELETE
AS
BEGIN
  EXCEPTION x_gehalt;
END

DELETE FROM t_gehalt WHERE angestellter = 2
```

Beim Versuch, einen Datensatz zu löschen, wird nun ein Fenster gemäß Bild 14.4 angezeigt

Wie eine SELECT-Abfrage zeigen würde, wird durch das Auslösen einer EXCEPTION das Löschen des Datensatzes verhindert. In der Praxis macht es wenig Sinn, das Löschen von Datensätzen pauschal zu unterbinden. Hier wird man eher mit IF...THEN-Verzweigungen dafür sorgen, daß besondere Datensätze nur unter bestimmten Umständen gelöscht werden können (Führungskräfte können nur vom SYSDBA »entlassen« werden).

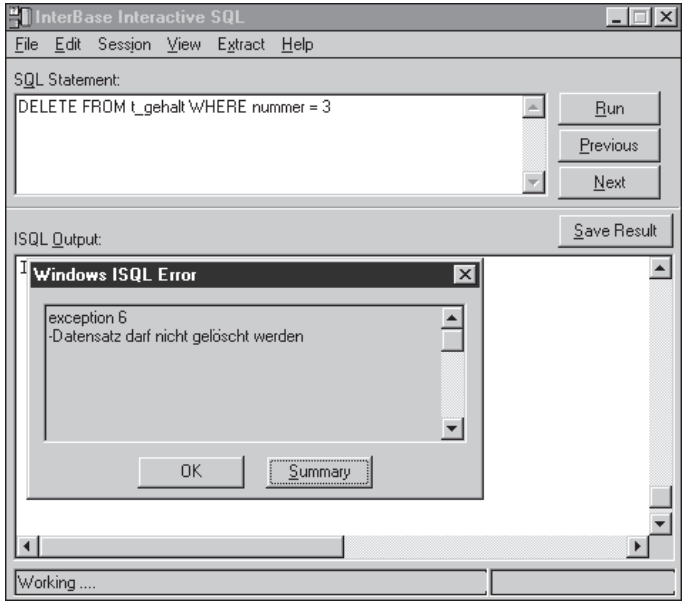


Bild 14.4: Fenster einer selbstdefinierten EXCEPTION

Einen TRIGGER löschen

Wahrscheinlich wissen Sie schon aus Erfahrung, wie bei SQL Objekte gelöscht werden:

```
DROP TRIGGER e_gehalt
```

14.6 Zugriffsberechtigungen

Standardmäßig hat nur der Ersteller einer Tabelle, VIEW oder STORED PROCEDURE oder der SYSDBA Zugriff auf dieses Objekt. Anderen Benutzern muß der Zugriff auf diese Objekte explizit gewährt werden. Versucht ein Benutzer einen Zugriff auf ein Objekt und ist nicht autorisiert, dann wird dies mit einer Fehlermeldung verweigert.

```
GRANT SELECT ON t_gehalt TO emil
```

Mit dieser Anweisung wird der Benutzer *emil* autorisiert, Daten von der Tabelle *t_gehalt* abzufragen – Datensätze ändern, einfügen oder löschen darf er jedoch nicht.

```
GRANT SELECT, DELETE, INSERT, UPDATE ON t_gehalt TO emil
GRANT ALL ON t_gehalt TO emil
```

Mit einer GRANT-Anweisung können gleichzeitig mehrere Rechte vergeben werden, wie die erste Anweisung zeigt. Mit ALL können alle Rechte vergeben werden; beide Anweisungen führen also zu demselben Resultat.

```
GRANT SELECT ON t_gehalt TO emil, frieda, gustav, hans
GRANT SELECT ON t_gehalt TO PUBLIC
```

Mit einer Anweisung können auch dieselben Rechte an mehrere Benutzer vergeben werden. Mit dem Schlüsselwort PUBLIC können Rechte an alle Benutzer vergeben werden.

```
GRANT SELECT (nummer, vornamen, nachnamen) ON t_adressen TO emil
```

Dieses Beispiel zeigt nun, wie Zugriffsrechte auf bestimmte Spalten eingeschränkt werden können; sind keine Spalten angegeben, dann erhält der Benutzer (sinnvollerweise) Zugriff auf alle Spalten einer Tabelle.

```
GRANT UPDATE ON t_gehalt TO emil WITH GRANT OPTION
```

Mit dem Zusatz WITH GRANT OPTION erhält der Benutzer außerdem die Gelegenheit, die an ihn übertragenen Rechte an andere Benutzer weiterzugeben. Dieser Zusatz sollte vorsichtig eingesetzt werden.

Zugriffsberechtigungen widerrufen

Um Zugriffsberechtigungen zu widerrufen, verwenden Sie die Anweisung REVOKE.

```
REVOKE UPDATE ON t_gehalt FROM emil
REVOKE ALL ON t_gehalt FROM emil
```

Der Kürze halber können Sie mit ALL alle Rechte widerrufen, auch wenn dem Benutzer gar nicht alle Rechte erteilt worden sind.

```
REVOKE DELETE ON t_gehalt FROM PUBLIC
REVOKE GRANT OPTION FOR SELECT ON t_gehalt FROM emil
```

Das erste Beispiel widerruft eine Zugriffsberechtigung aller Benutzer. Das zweite Beispiel widerruft die GRANT OPTION, also die Erlaubnis, Zugriffsberechtigungen weiterzugeben.

Beim Einsatz von REVOKE sind folgende Punkte zu beachten:

- Eine Zugriffsberechtigung kann nur von dem Anwender widerrufen werden, der sie vergeben hat.

- Ist einem Anwender von zwei verschiedenen Benutzern das gleiche Recht verliehen worden, dann müssen beide Benutzer es widerrufen, damit es diesem entzogen ist.
- Wird einem Anwender ein Recht entzogen, der dafür auch die GRANT OPTION hatte, dann wird es allen Anwendern, an die er es weitergegeben hat, ebenfalls entzogen.
- Ein Recht, das PUBLIC vergeben worden ist, kann auch nur so widerrufen werden. Es besteht nicht die Möglichkeit, an alle ein Recht zu vergeben und es dann für *emil* zu widerrufen.

Zugriffsberechtigungen für VIEWs und Prozeduren

Bei den Zugriffsrechten können VIEWs behandelt werden wie gewöhnliche Tabellen. Meist ist es zu empfehlen, nur das SELECT-Recht an einer VIEW weiterzugeben, damit nicht ein Benutzer durch Änderungen der VIEW die für ihn nicht sichtbare darunterliegende Datenmenge in Mitleidenschaft zieht.

Um eine VIEW zu erstellen, muß der Anwender, der sie erstellt, die SELECT-Rechte für die beteiligten Tabellen besitzen.

Zugriffsberechtigungen können / müssen auch an STORED PROCEDUREs vergeben werden:

```
GRANT INSERT ON t_adressen TO PROCEDURE p_adressen, p_test
```

Diese Anweisung würde nun das INSERT-Recht an die beiden Prozeduren *p_adressen* und *p_test* vergeben.

Zudem muß auch das Recht gewährt werden, eine Prozedur ausführen zu dürfen; hierbei handelt es sich um ein EXECUTE-Recht:

```
GRANT EXECUTE ON PROCEDURE p_adressen TO PUBLIC
```

14.7 Sonstiges

Der *Local InterBase Server* ist ein sehr umfangreiches Thema, welches in diesen beiden Kapiteln nicht vollständig abgehandelt werden konnte (allein das Handbuch weist – bei Verzicht auf viele Themen – einen Umfang von mehr als 200 Seiten auf).

Es ist versucht worden, hier das Wichtigste darzustellen. Diejenigen Leser, die sich intensiver mit der Materie beschäftigen wollen, mögen die folgenden Themen in der Online-Hilfe und / oder im Handbuch nachschlagen:

- Nur am Rande erwähnt wurden die Befehle der DML (Data Manipulation Language), also die Befehle SELECT, INSERT, UPDATE und DELETE. Über den SELECT-Befehl finden Sie einiges in Kapitel 3 bei der Komponente *TQuery*. Mit INSERT-, UPDATE- und DELETE-Anweisungen werden Sie in der Regel nicht viel zu tun haben, dies werden die C++Builder-Komponenten und die BDE selbstständig regeln.
- Das Thema *Transaktionen* (SET TRANSACTION, COMMIT, ROLLBACK) wird im nächsten Kapitel bei der Komponente *TDatabase* erläutert.
- Um den Datenbestand gegen Hardwarefehler zu sichern, besteht die Möglichkeit, einen SHADOW zu definieren, also eine Datei, welche jederzeit eine exakte Kopie der Datenbank ist.
- Der Programmierer kann eigene EXCEPTIONs erstellen, die er in TRIGGERn und STORED PROCEDUREs auslösen und behandeln kann. Auch auf Fehlermeldungen der Datenbank kann reagiert werden.
- Es können Funktionen von externen Programmiersprachen importiert und Filter für BLOBs erstellt werden.

15 C++Builder und Client-Server

Werden Datenbank-Applikationen für Client-Server-Systeme erstellt, dann sind einige Dinge zu beachten.

Zunächst einmal muß *Borland SQL Links* installiert sein, welches eine Art BDE-Treibersammlung zum Zugriff auf Datenbankserver ist. *Borland SQL Links* ist in der Client-Server-Version von C++Builder enthalten und in einem kleinen Handbuch beschrieben, deshalb soll hier nicht näher darauf eingegangen werden.

Zum Zugriff auf den *Local InterBase Server* ist dieses Programm nicht erforderlich, hier sind die entsprechenden Treiber schon in der BDE enthalten. Damit die folgenden Beispiele auch von Lesern nachvollzogen werden können, die nicht über die Client-Server-Suite und ggf. zusätzlich über ein Datenbankserver-Programm verfügen, werden sich die Beispiele auf den LIBS beschränken.

15.1 Feldtypen

Zunächst die gute Nachricht: Alle C++Builder-Komponenten arbeiten relativ problemlos mit InterBase-Datenbanken zusammen. Eine Ausnahme bildet die Komponente *TDBBGrid*, das in Kapitel 4 entwickelte Datenbankgitter, welches auch Memos und Graphiken anzeigt.

Die Anzeige der Memos funktioniert auch weiterhin, nur bei den Graphiken gibt es ein Problem – diese sind vom Typ *TBlobField* und nicht vom Typ *TGraphicField*, denn InterBase kennt keine Graphik-Felder (für Memos gibt es einen speziellen BLOB-Subtyp, so daß C++Builder diese korrekt verarbeitet).

15.1.1 Anzeige von Graphiken

Während die Komponente *TDBImage* problemlos auf diese Felder zugreifen kann, gibt es bei *TDBBGrid* ein Problem in der folgenden Programmzeile:

```
if(Field->ClassNameIs("TGraphicField"))
```

Da es sich hier dann um ein *TBlobField* und nicht um ein *TGraphicField* handelt, wird das Bild gar nicht erst angezeigt. Um in einem *DBGrid* (oder *DBBGrid*) dennoch Graphiken anzeigen zu können, muß einfach nur die Ereignisbehandlungsroutine *OnDrawDataCell* entsprechend programmiert werden. Auf diese Weise bekommen Sie auch Graphiken in die Gitter, wenn Sie die Komponente *TDBBGrid* nicht einsetzen können oder wollen (bei dBase- und Paradox-Tabellen dann wieder *TGraphicField* !). Die *if*-Verzweigung ist nun folgendermaßen zu programmieren:

```
if(Field->ClassNameIs("TBlobField"))
```

15.1.2 Anzeige von Zahlen und Geldbeträgen

Wer mit InterBase Spalten vom Typ NUMERIC oder DECIMAL definiert hat, wird unter C++Builder eine hübsche Überraschung erleben – diese werden nämlich zu Feldern vom Typ *TIntegerField*, etwaige Nachkommastellen werden schlicht unterschlagen.

Spalten, die Nachkommastellen benötigen, gilt es deshalb als FLOAT zu definieren. Soll die Zahl der angezeigten Nachkommastellen beschränkt werden (dies ist sehr zu empfehlen), dann kann dafür die Eigenschaft *DisplayFormat* von *TFloatField* verwendet werden.

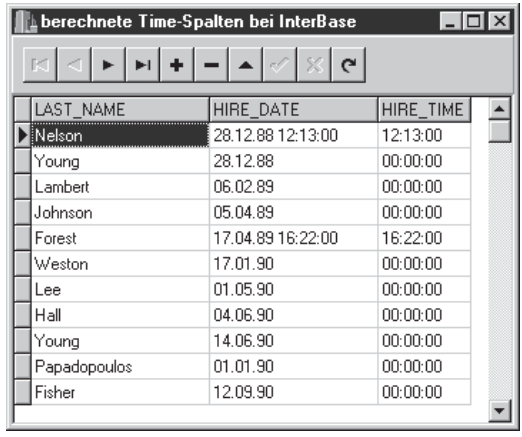
Eigene Währungsfelder gibt es bei InterBase nicht, deklarieren Sie solche Spalten als FLOAT. Um bei der Anzeige die Nachkommastellen auf zwei zu beschränken und den String *DM* anzuhängen, setzen Sie die Eigenschaft *Currency* auf *true*. Alternativ kann die Eigenschaft *DisplayFormat* auf *0.00 DM* gesetzt werden (oder auf beispielsweise *0.00 öS* bei anderen Währungen).

15.1.3 Anzeige von Datums- und Zeitwerten

InterBase kennt nur den Datentyp DATE, der allerdings Zeit- und Datumswerte enthält; in C++Builder wird daraus dann folgerichtig ein *TDateTimeField*.

Werden in eine DATE-Spalte nur Datumswerte eingegeben, dann wird die Uhrzeit automatisch auf *0:00 Uhr* gesetzt. In den datensensitiven Dialogelementen wird dann davon ausgegangen, daß eine nicht eingegebene Zeit auch nicht angegeben werden soll und zeigt in disem Fall die Zeit nicht an, auch dann nicht, wenn diese mit *0:00 Uhr* explizit so angegeben wurde. (In manchen Fällen kommt es eher auf die Anzeige als auf eine Sekunde an, dann kann man *0:00:01 Uhr* angeben).

Es besteht dann auch die Möglichkeit, wie Bild 15.1, Datum und Uhrzeit getrennt anzuzeigen, dazu sind allerdings ein paar Zeilen Quelltext erforderlich:



LAST_NAME	HIRE_DATE	HIRE_TIME
Nelson	28.12.88 12:13:00	12:13:00
Young	28.12.88	00:00:00
Lambert	06.02.89	00:00:00
Johnson	05.04.89	00:00:00
Forest	17.04.89 16:22:00	16:22:00
Weston	17.01.90	00:00:00
Lee	01.05.90	00:00:00
Hall	04.06.90	00:00:00
Young	14.06.90	00:00:00
Papadopoulos	01.01.90	00:00:00
Fisher	12.09.90	00:00:00

Bild 15.1: Anzeige von TDateTimeField

```
void __fastcall TForm1::Table1CalcFields(TDataSet *DataSet)
{
  Table1->FieldByName("HIRE_TIME")->AsString
    = TimeToStr(Table1->FieldByName("HIRE_DATE")->AsDateTime);
}
```

Zunächst wird im *DBGrid* die Zeit mit Hilfe eines berechneten Feldes (Typ *TStringField*) angezeigt. Die Funktion *TimeToStr* sorgt dafür, daß nur die Zeit angezeigt wird.

Beachten Sie bitte, daß berechnete Felder keine Eingaben entgegennehmen. Dafür wäre es erforderlich, zwei *TEdit*-Komponenten einzufügen (alternativ kann man natürlich auch Datum und Zeit in der Spalte HIRE_DATE eingeben).

```
void __fastcall TForm1::DataSource1DataChange(TObject *Sender,
TField *Field)
{
  Edit1->Text
    = DateToStr(Table1->FieldByName("HIRE_DATE")->AsDateTime);
  Edit2->Text
    = TimeToStr(Table1->FieldByName("HIRE_DATE")->AsDateTime);
}
```

Mit ähnlichen Anweisungen zeigt man hier Datum und Uhrzeit voneinander getrennt an.

```
void __fastcall TForm1::DataSource1UpdateData(TObject *Sender)
{
  AnsiString s;
  s = Edit1->Text + " " + Edit2->Text;
  Table1->FieldByName("HIRE_DATE")->AsString = s;
}
```

Wenn ein Datensatz aktualisiert werden soll, dann müssen die Texte von *Edit1* und *Edit2* ausgelesen und dem Feld HIRE_DATE zugewiesen werden. Die *TField*-Komponente wandelt hier selbstständig den String in einen *TDateTime*-Wert um.

```
void __fastcall TForm1::Edit1Enter(TObject *Sender)
{
  DataSource1->Edit();
}
```

Datensensitive Dialogelemente versetzen die Datenmenge automatisch in den Modus *dsEdit*, wenn sie den Fokus erhalten. Bei nichtdatensensitiven Dialogelementen kann man durch Aufruf der Methode *Edit* diese Funktion implementieren. Die Ereignisbehandlungsroutine *TForm1::Edit1Enter* wird hier auch der Komponente *Edit2* zugewiesen.

Im übrigen sollte man es vermeiden, daß mehrere Komponenten auf dasselbe Feld derselben Datenquelle zugreifen – im harmlosesten Fall stehen widersprüchliche Feldinhalte auf dem Bildschirm.

15.1.4 Selbstinkrementierende Felder

Im letzten Kapitel haben wir besprochen, wie man mit Hilfe von Generatoren selbstinkrementierende Felder erzeugt:

```
CREATE TABLE t_namen
  (nummer INTEGER NOT NULL,
  namen VARCHAR(20),
  PRIMARY KEY (nummer))

CREATE GENERATOR g_namen

CREATE TRIGGER e_namen FOR t_namen
BEFORE INSERT
AS
BEGIN
  NEW.nummer = GEN_ID(g_namen, 1);
END
```

Wird nun auf diese Tabelle mit einer *TTable*-Komponente zugegriffen, dann wird man nicht viel Freude damit haben: Sobald versucht wird, einen Datensatz ohne das Feld *nummer* einzufügen (dieses soll ja die Datenbank selbst generieren), dann erfolgt eine Fehlermeldung, die besagt, daß NULL-Werte in der Spalte *nummer* nicht erlaubt sind.

Die Komponente *TTable* ist nämlich nicht dafür ausgelegt, nur auf einige Spalten zuzugreifen. Wird ein neuer Datensatz eingefügt, dann wird in alle Felder, für die es keine Eingabe gibt, ein NULL-Wert geschrieben. Hier löst nun die BDE einen Fehler aus, da für die Spalte *nummer* keine NULL-Werte gestattet sind.

Dieses Problem kann man umgehen, auch wenn es bei der Verwendung von einem *DBGrid* etwas seltsam aussieht:

```
void __fastcall TForm1::Table1AfterInsert(TDataSet *DataSet)
{
  Table1->FieldByName("Nummer")->AsInteger = 1;
}

void __fastcall TForm1::Table1AfterPost(TDataSet *DataSet)
{
  Table1->Refresh();
}
```

Sobald die Datenmenge in den Status *dsEdit* versetzt wird, fügt die Methode *Table1AfterInsert* in die Spalte *nummer* eine beliebig große Zahl ein. In einem *DBGrid* wird diese Zahl nun angezeigt, was nicht schön aussieht, sich aber nicht vermeiden läßt (mit *OnBeforePost* funktioniert die Sache leider nicht). Wird eine Eingabemaske aus *TDBEdit*-Komponenten zusammengesetzt, dann wird man für die Spalte *nummer* eine *TDBText*-Komponente verwenden, deren Eigenschaft *visible* dann vorübergehend auf *false* gesetzt werden kann.

Wird nun die Methode *Post* aufgerufen, dann wird dieser Datensatz in die Datenbank geschrieben und die Spalte *nummer* vom TRIGGER nun korrekt gesetzt. Damit dieser Wert nun auch in der Anwendung angezeigt wird, muß die Methode *Table1.Refresh* aufgerufen werden.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Table1->Append();
  Table1->FieldByName("Namen")->AsString = Edit1->Text;
  Table1->FieldByName("Nummer")->AsInteger = 1;
  Table1->Post();
  Table1->Refresh();
}
```

Werden keine datensensitiven Dialogelemente verwendet, dann können die entsprechenden Anweisungen gleich in die Prozedur geschrieben werden, welche den Datensatz aktualisiert. Beachten Sie bitte, daß auch hier die Spalte *nummer* gesetzt werden muß, andernfalls beschwert sich die BDE über einen NULL-Wert.

Wird eine *TQuery*-Komponente verwendet, dann wird die SQL-Anweisung nicht von der BDE, sondern vom Server bearbeitet (sofern dies bei der BDE-Konfiguration so eingestellt worden ist). Soll eine *TQuery* eine *TTable* ersetzen, dann wird folgende SQL-Anweisung verwendet:

```
SELECT * FROM t_namen
```

Des weiteren muß, damit Datensätze eingefügt (geändert, gelöscht) werden können, die Eigenschaft *RequestLive* auf *true* gesetzt werden. Hier funktioniert dann leider die Methode *Refresh* nicht mehr (damit kann der Server nichts anfangen), so daß man diese mit den folgenden beiden Anweisungen simulieren muß:

```
void __fastcall TForm1::Query1AfterPost(TDataSet *DataSet)
{
  Query1->Close();
  Query1->Open();
}
```

Soll das ganze ohne datensensitive Dialogelemente bewerkstelligt werden, dann wird dafür folgende Prozedur verwendet (man könnte hier auch Parameter benutzen, siehe Kapitel 3):

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Query1->SQL->Clear();
  Query1->SQL->Add("INSERT INTO t_namen (namen)");
  Query1->SQL->Add("  VALUES ('" + Edit1->Text + "')");
  Query1->ExecSQL();
  Table1->Refresh();
}
```

Im Beispiel ist ein *DBGrid* mit *Table1* verbunden gewesen. Damit hier der neue Datensatz auch angezeigt wird, muß die Anweisung *Table1.Refresh* in die Prozedur aufgenommen werden. Im übrigen sollten Sie die Eigenschaft *RequestLive* von *Query1* auf *false* stellen, ansonsten erhalten Sie bei der Methode *ExecSQL* die Fehlermeldung *Tabelle ist schreibgeschützt*.

15.2 Transaktionen

Bisweilen kommt es vor, daß eine Gruppe von Datenbankaktionen unbedingt gemeinsam ausgeführt werden muß – wenn nicht alle Aktionen gemeinsam ausgeführt werden können, dann sollen sie gemeinsam verworfen werden.

Nehmen wir einmal an, die Daten einer wissenschaftlichen Erhebung sollten anonymisiert gespeichert werden. Aus was für Gründen auch immer sollen dabei sowohl die Daten als auch die Namen der dazugehörenden Personen gespeichert werden, allerdings auf eine Art und Weise, daß sich die Daten nicht mehr den Personen zuordnen lassen.

Prinzipiell ist dies gar kein sonderlich großes Problem: Es werden zwei Tabellen erstellt, die nicht weiter miteinander verknüpft sind. Eine der Tabellen nimmt die Personendaten, die andere die Erhebungsdaten auf.

Aus Gründen, die in diesem Buch schon hinreichend beschrieben worden sind, sollten Tabellen als Primärschlüssel eine (normalerweise durchlaufende) Nummer haben. Dies würde allerdings hier den Datenschutzbemühungen völlig zuwiderlaufen. Es wäre hier keine Lösung, aus der Nummer der einen Tabelle die der anderen durch eine mathematische Formel abzuleiten, denn dann könnte man die Zusammenhänge rekonstruieren. Auch ist es nicht unbedingt sinnvoll, die Nummern von der mit der Eingabe betrauten Person frei nach Lust und Laune wählen zu lassen, auch hier ist die Wahrscheinlichkeit viel zu hoch, daß sich irgendeine ungewollte Systematik einschleicht.

Es bleibt also gar nichts anderes übrig, als die Schlüsselnummern mit Hilfe des Zufallszahlengenerators (die Pascal-Funktion *random*) generieren zu lassen. Nun ist es allerdings nicht auszuschließen, daß im Laufe der Eingabe einer der Werte vom Generator doppelt erzeugt wird, und dies zieht bei der Datenbank eine Fehlermeldung nach sich – als Primärschlüssel sind auch in diesem Fall nur einmalige Werte gestattet.

Das Problem bei dieser Geschichte ist, daß vermutlich diese Primärschlüsselverletzung nur bei einer der beiden Tabellen auftritt – entweder ist der Personendatensatz oder der Erhebungsdatensatz bereits geschrieben. Wird nun das Datensatzpaar erneut eingegeben, dann ist dieser Wert doppelt in der Tabelle enthalten (solange nicht der höchst unwahrscheinliche Fall eintritt, daß es wieder zu einer Primärschlüsselverletzung kommt, und diesmal exakt bei der anderen Tabelle).

Bei Einplatzsystemen könnte man die beiden Nummern vorher abfragen, damit wäre zumindest dieses Problem umgangen. Nun sind wir aber beim Thema Client-Server-Systeme, und hier wäre nicht auszuschließen, daß die gerade geprüfte Nummer zwischenzeitlich von einem anderen System verwendet worden ist. Zudem sind Primärschlüsselverletzungen nicht das einzige, was dem Schreiben eines Datensatzes im Wege stehen kann.

Die Aufgabe des Transaktions-Managements ist klar: Es sollen die beiden Datensätze geschrieben werden; geht dabei alles klar, dann wird der Vorgang bestätigt, tritt irgendwo ein Fehler auf, dann werden beide Einträge zurückgenommen.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  randomize();
}
```

Vor der Verwendung des Zufallszahlengenerators muß dieser mit *randomize* initialisiert werden.

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
  try
  {
    Database1->StartTransaction();

    Table1->Append();
    Table1->FieldByName("Nummer")->AsInteger = 3; //random(65000);
    Table1->FieldByName("Vorname")->AsString = Edit1->Text;
    Table1->FieldByName("Nachname")->AsString = Edit2->Text;
    Table1->Post();

    Table2->Append();
    Table2->FieldByName("Nummer")->AsInteger = 4; //random(65000);
    Table2->FieldByName("Laenge")->AsString = Edit3->Text;
    Table2->FieldByName("Breite")->AsString = Edit4->Text;
    Table2->Post();

    Database1->Commit();
    Edit1->Text = "";
    Edit2->Text = "";
    Edit3->Text = "";
    Edit4->Text = "";
  } // try
  catch (...)
  {
    Database1->Rollback();
    ShowMessage("Datensatz nicht geschrieben");
  }
} // TForm1::BitBtn1Click
```

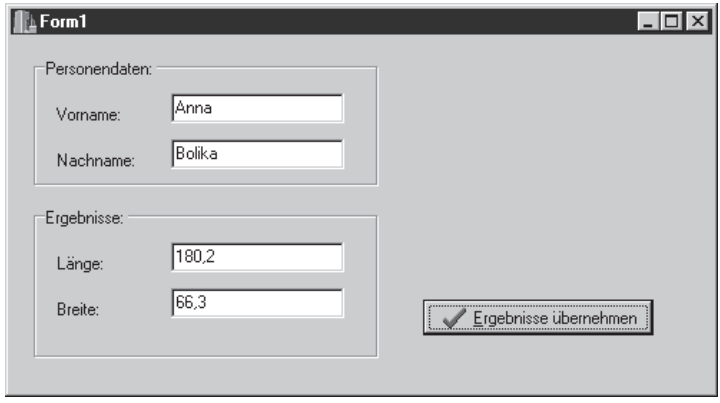


Bild 15.2: Anonymisierte Speicherung personenbezogener Daten

Die Anweisungen zum Schreiben des Datensatzes werden in einen *try...catch*-Rahmen gesteckt. Tritt dort während der Ausführung irgendeine *Exception* auf, dann springt C++Builder zum *catch*-Teil und nimmt dort die Änderungen mit *Database1.Rollback* zurück.

Zu den Eigenschaften und Methoden von *TDatabase* kommen wir gleich. Mit der Methode *StartTransaction* wird die Transaktion gestartet. Es wird dann an die beiden Tabellen je ein Datensatz angehängt, und die entsprechenden Felder werden mit Werten gefüllt. Beachten Sie auch, daß hier im Beispiel die Verwendung datensensitiver Dialogelemente nicht nötig gewesen ist. Geht alles klar, dann werden die beiden Einfügungen mit *Commit* bestätigt. Anschließend werden die Edit-Felder gelöscht, damit der Anwender auch sieht, daß die Datenbank die Eingabe akzeptiert hat und nicht versehentlich alte Werte stehenbleiben.

Tritt während der Ausführung eine *Exception* auf, dann springt C++Builder sofort zu den Anweisungen nach *catch*. Zunächst werden die Änderungen widerrufen. Des weiteren wird eine Fehlermeldung ausgegeben, denn sonst würde der Fehler für den Anwender nicht mehr erkennbar sein, sobald das Programm nicht mehr unter dem Debugger läuft.

15.2.1 Rücknahme von falschen Eingaben

Transaktionen können auch dazu verwendet werden, um falsche Eingaben und Änderungen zu widerrufen. Angenommen, es sollte eine Person im Umgang mit der Anwendung angelernt werden. Damit nun diese Person ohne Beaufsichtigung mit dem System experimentieren kann, wird zu Beginn eine Transaktion gestartet, welche zum Schluß mit ROLLBACK widerrufen wird. Nun sind alle Tabellen – unabhängig davon, was eingegeben, geändert oder gelöscht wurde –

wieder im alten Zustand. Durch die Abschottung der einzelnen Transaktionen voneinander – wir werden das gleich besprechen – sind noch nicht einmal die Anwender, welche die Datenbank zur gleichen Zeit benutzen, von diesen Experimenten betroffen.

Auch erfahrenen Anwendern unterlaufen manchmal Anweisungen, die sie später lieber wieder rückgängig machen wollen. Sieht beispielsweise das Programm die Möglichkeit vor, SQL-Anweisungen in einem Editor zu erstellen und dann auszuführen, dann kommt es schnell vor, daß bei UPDATE- oder DELETE-Anweisungen die WHERE-Klausel vergessen wird und somit alle Datensätze geändert bzw. gelöscht werden.

```
UPDATE t_adressen SET nachnamen = "Müller"
DELETE FROM t_adressen WHERE nummer < 12345
```

Nach der ersten Anweisung würde alle Personen in *t_adressen* den Namen *Müller* haben. Bei der DELETE-Anweisung wollte der Anwender alle Datensätze mit einer Nummer größer 12345 löschen (also die fünf, die er gerade falsch eingegeben hat) und hat versehentlich das falsche Vergleichszeichen verwendet – schon ist seine Tabelle ziemlich handlich.

Um hier die Möglichkeit zu bieten, das Eingegebene zurückzunehmen oder zu bestätigen, können entsprechende Buttons oder Menüpunkte vorgesehen werden. Soll vermieden werden, daß Anzulernende versehentlich ihre Experimentierergebnisse bestätigen, dann ist es ratsam, die COMMIT-Aktion mit einem Paßwort zu koppeln.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Database1->StartTransaction();
}

void __fastcall TForm1::FormClose(TObject *Sender,
  TCloseAction &Action)
{
  Database1->Commit();
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Database1->Commit();
  Database1->StartTransaction();
}
```

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  Database1->Rollback();
  Table1->Refresh();
  Database1->StartTransaction();
}
```

Hier im Beispiel wird beim Starten der Anwendung automatisch eine Transaktion erzeugt, die beim Verlassen der Anwendung bestätigt wird. Wird *Button1* betätigt, dann wird das bisherige bestätigt und eine neue Transaktion begonnen. Mit *Button2* werden die bisherigen Aktionen zurückgenommen, und es wird ebenfalls eine neue Transaktion gestartet. Damit die Rücknahme auch im *DBGrid* zu sehen ist, wird noch *Table1->Refresh* aufgerufen.

15.2.2 Abschottung von Transaktionen

Ein Kennzeichen von Client-Server-Systemen ist es, daß mehrere Benutzer gleichzeitig auf die Datenbank zugreifen können. Durch die Verwendung von Transaktionen können hier zusätzliche Probleme auftreten. Nehmen wir einmal an, ein Anwender würde die Mitgliederstatistik einer Partei erstellen. Zunächst wird die Zahl der Mitglieder nach Bundesländern aufgeschlüsselt, hier sind es insgesamt 2345 Mitglieder. Danach wird die Altersstruktur ermittelt; weil inzwischen ein neues Mitglied von einem anderen Anwender eingegeben wurde, beträgt nun die Summe 2346 Mitglieder.

Weil das Erstellen der Reports automatisch abläuft, können diese Zahlen nicht manuell korrigiert werden. Andererseits sollen die Zahlen dem Vorstand präsentiert werden, und der Schatzmeister ist ein sehr penibler Mensch – eine Abweichung würde hier eine Diskussion von mindestens einer halben Stunde bedeuten. Also wird nochmals die Zahl der Mitglieder nach Bundesländern aufgeschlüsselt. Da inzwischen die Austrittserklärung von Famile Mustermann eingegeben wurde, beträgt die Summe nun 2344 Mitglieder. An dieser Stelle möchte ich die Geschichte abbrechen, obwohl man noch seitenweise weiterfabulieren könnte.

Sicher ist auf jeden Fall, daß man die einzelnen Clients voneinander isolieren oder abschotten muß. In diesem Zusammenhang gibt es drei klassische Probleme, die hier kurz besprochen werden sollen. Wenn Sie die Beispiele selbst nachvollziehen möchten, dann starten Sie eine C++Builder-Anwendung und das Programm *Interactive SQL* und greifen Sie auf dieselbe Datenbank zu; auf diese Weise können Sie Probleme, die bei mehreren Anwendern auftreten, auf einem Rechner simulieren. (Achtung: Wenn Sie den *Local InterBase Server* verwenden, dann müssen Sie die C++Builder-Anwendung vom Explorer aus starten, weil maximal zwei Verbindungen gehalten werden.)

Lost Updates

Von *Lost Updates* (verlorenen Änderungen) spricht man, wenn eine geänderte Reihe von einer parallel verlaufenden Transaktion ebenfalls geändert werden soll. Ein Beispiel dazu zeigt Bild 15.3: Zunächst wurde in der C++Builder-Anwendung das Feld *namen* in Reihe *35* auf *Yvonne* gesetzt. Darauf wurde mit ISQL versucht, dasselbe Feld auf *Carsten* zu setzen, worauf eine Fehlermeldung ausgegeben wird. (Die Beschreibung lautet in diesem Fall kurz *deadlock*, dieser Begriff beschreibt aber in der Literatur ein etwas anderes Problem.)

Aktionen, welche *Lost Updates* verursachen würden, werden von jedem mir bekannten System mit einer Fehlermeldung beantwortet, unabhängig davon, welcher Isolationsgrad gewählt wurde.

Dirty Reads

Unter »schmutzigen Lesevorgängen«, wie eine wörtliche Übersetzung lauten würde, versteht man das Lesen von Daten, deren Transaktion noch nicht mit COMMIT abgeschlossen wurde. Soll beispielsweise der aktuelle Lagerbestand eines Artikels abgerufen werden, dann erhält man mit dem Lesen unbestätigter Aktionen den aktuelleren Stand.

Es wäre dann aber folgende Situation denkbar: Lagerbestand an 21"-Monitoren beträgt zehn Stück. In der Auftragsbearbeitung wird eine Bestellung eingegeben, der Kunde hat zehn Mouse-Pads bestellt, der Sachbearbeiter vertippt sich und gibt anstelle der Bestellnummer eines Mouse-Pads die des Monitors ein.

Kurz danach fragt der Einkauf die Lagerbestände ab und stellt fest, daß bei den 21"-Monitoren der Bestand innerhalb einer Woche von zehn auf null zurückgegangen ist. Das Gerät ist anscheinend ein Renner, kurzerhand werden 20 Stück nachbestellt. Inzwischen hat sich der Sachbearbeiter in der Auftragsbearbeitung die Gesamtsumme der Bestellung angesehen und seinen Fehler bemerkt, mit einer ROLLBACK-Anweisung werden die Eingaben rückgängig gemacht. Zwei Wochen später muß der Einkauf dann erklären, wieso von diesen völlig unverkäuflichen 21"-Monitoren 20 Stück nachbestellt wurden.

Wie dieses Beispiel zeigt, sind *dirty reads* eine nicht ganz ungefährliche Sache. Sie werden auch nur vom Informix-Server unterstützt (und dort läßt sich der Isolationsgrad auch so einstellen, daß nur mit COMMIT bestätigte Transaktionen gelesen werden können).

Mit der Komponente *TDatabase* können Transaktionen auch bei Desktop-Datenbanken durchgeführt werden. Hier muß dann zwingend der Abschottungsgrad *DirtyRead* verwendet werden.

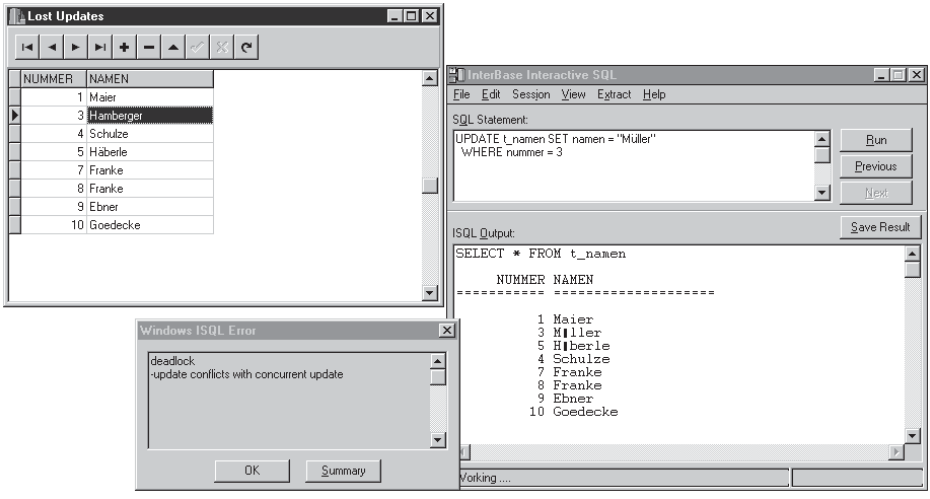


Bild 15.3: Lost Updates

Non-reproducable Reads

Das Problem der nicht-reproduzierbaren Lesevorgänge tritt dann auf, wenn mehrmals dieselbe oder eine ähnliche SELECT-Anweisung durchgeführt wird, während eine andere Transaktion INSERT-, UPDATE- oder DELETE-Aktionen durchführt. Ein Beispiel dazu wurde eingangs gegeben.

Dieses Problem führte zur Einführung von zwei verschiedenen Abschottungsgraden:

- Mit READ COMMITTED können alle Datenänderungen gelesen werden, welche mit COMMIT abgeschlossen worden sind. Dieser Abschottungsgrad vermeidet keine nichtreproduzierbaren Lesevorgänge.
- Mit SNAPSHOT (in C++Builder *tiRepeatableRead*) wird zu Beginn der Transaktion die Tabelle sozusagen »eingefroren«, die Daten ändern sich für diese Transaktion nicht, auch wenn andere Transaktionen ihre Änderungen mit COMMIT bestätigt haben.

  Hier sollte man genau darauf achten, daß die Transaktionen nicht zu lange währen. Würde beispielweise die Transaktion beim Starten des Programms gestartet und dessen Beendigung mit COMMIT bestätigt, dann arbeitet der Anwender am Ende eines Arbeitstages schon mit ziemlich veralteteten Daten. Zudem steigt der Aufwand des DBMS, welches bei jeder Datenbankaktion anderer Transaktionen eine zusätzliche Version speichern muß.

Von der Komponente *TDatabase* nicht unterstützt wird der InterBase-Abschottungsgrad SNAPSHOT TABLE STABILITY, auf den daher hier nicht näher eingegangen werden soll.

Deadlock

Bei der InterBase-Anweisung SET TRANSACTION kann angegeben werden, ob bei einem *lock conflict* eine Fehlermeldung ausgegeben wird oder ob die Anwendung so lange wartet, bis die Sperre aufgehoben ist, und die Aktion durchgeführt werden kann.

Soll die Transaktion warten, dann können Situationen auftreten, bei denen Transaktion 1 darauf wartet, daß die Transaktion 2 (mit COMMIT oder ROLLBACK) abgeschlossen wird, während diese wiederum darauf wartet, daß Transaktion 1 abgeschlossen wird. In diesem Fall ist dann eine sog. *deadlock situation* aufgetreten, die beiden Transaktionen lähmen sich gegenseitig.

Um eine solche Situation zu vermeiden, sollten Transaktionen lieber gleich eine Fehlermeldung ausgeben, anstatt zu warten. Ausnahmen können dann sinnvoll sein, wenn Transaktionen automatisch ablaufen müssen, wenn also kein Anwender den Vorgang erneut starten könnte, wenn mit einer Fehlermeldung abgebrochen worden ist.

15.3 Die Komponente TDatabase

Bei jeder Datenbank-Anwendung wird normalerweise automatisch eine Instanz der *TDatabase*-Komponente mit dem Namen *Database* erzeugt, auf die vom Programm aus zugegriffen werden kann. Sollen Eigenschaften oder Methoden von *TDatabase* verwendet werden, dann ist es meist sinnvoller, explizit eine *TDatabase*-Komponente in das Projekt einzufügen; dies erlaubt es dann auch, Eigenschaften zur Entwurfszeit mit dem Objektinspektor zu setzen.

Entgegen der sonstigen Gewohnheit in diesem Buch sollen hier nicht der Reihe nach die Eigenschaften, Methoden und Ereignisse von *TDatabase* abgehandelt werden. Statt dessen sollen hier kurz die drei sinnvollen Einsatzgebiete von *TDatabase* erläutert werden.

15.3.1 Lokaler Alias

In den Komponenten *TTable*, *TQuery* und *TStoredProc* wird der Eigenschaft *DatabaseName* für gewöhnlich der Namen eines BDE-Aliases zugewiesen. Alternativ kann auch das Verzeichnis (und bei InterBase-Datenbanken der Dateiname) angegeben werden.

Wird eine *TDatabase*-Komponente eingefügt, dann wird für gewöhnlich ein lokaler Alias verwendet. Hier wird ein frei gewählter Alias-Name (beispielsweise INTERN) der Eigenschaft *DatabaseName* von *TDatabase* zugewiesen. In der LookUp-Liste *DatabaseName* von *TTable*, *TQuery* und *TStoredProc* ist nun dieser lokale Alias zusammen mit den BDE-Aliasen aufgeführt – wird er gewählt, dann

bezieht die jeweilige *DataSet*-Komponente ihre Daten über diese *TDatabase*-Komponente.

Der Eigenschaft *AliasName* von *TDatabase* wird nun auf den BDE-Alias gesetzt, aus dessen Datenbank die Daten bezogen werden sollen. Anschließend wird die Eigenschaft *Connected* auf *true* gesetzt – nach dem Einloggen (wenn es sich um eine Client-Server-Datenbank handelt) sind dann auch zur Entwurfszeit wieder Live-Daten verfügbar.

Die Verwendung eines lokalen Aliases kann beim Upsizing von Datenbanken viel Arbeit ersparen, weil der Name des BDE-Aliases nur bei einer *TDatabase*-Komponente und nicht bei vielen *DataSet*-Komponenten geändert werden zu braucht. (Alternativ läßt sich selbstverständlich auch der BDE-Alias auf eine andere Datenbank setzen).

15.3.2 Automatisches Einloggen

Wird eine C++Builder-Anwendung gestartet, die auf eine Client-Server-Datenbank zugreift, dann wird vom Anwender zunächst einmal die Eingabe von Benutzername und Paßwort verlangt. Zumindest beim Programmieren ist dies auf Dauer äußerst lästig. Auch im späteren Betrieb gibt es Gründe, dies zu vermeiden, denn auf diese Weise kommen alle Anwender in den Besitz eines gültigen Paßwortes und können dann auch mit Programmen wie *Interactive SQL* oder dem *Server Manager* auf die Datenbank zugreifen, was meist weniger erwünscht ist.

Hier besteht nun die Möglichkeit, daß in der Datenbank eine eigene Tabelle mit Benutzernamen und dazugehörigen Paßwörtern erstellt wird. Die Anwendung loggt sich dann bei der Datenbank mit einem Paßwort ein, das keinem Anwender bekannt ist, und die Anwender loggen sich bei der Anwendung mit einem Paßwort ein, mit dem sie nicht direkt auf die Datenbank zugreifen können.

Um das Anzeigen des Paßwort-Dialoges zu vermeiden, gibt es zwei Möglichkeiten: Zum einen kann die Eigenschaft *LoginPrompt* auf *false* gesetzt werden, zum anderen kann dem Ereignis *OnLogin* eine Ereignisbehandlungsroutine zugewiesen werden (beispielsweise eine, die das Öffnen eines eigenen Paßwort-Dialoges veranlaßt).

Beiden Möglichkeiten ist gemeinsam, daß Benutzernamen und Paßwort der Eigenschaft *Params* zugewiesen werden müssen. Die entsprechenden Zeilen könnten dann folgendermaßen aussehen:

```
USER NAME=SYSDBA
PASSWORD=masterkey
```

Beachten Sie bitte, daß Paßwort und Benutzernamen ohne Anführungszeichen angegeben werden, daß beim Paßwort die Groß- und Kleinschreibung zu beachten ist und daß vor und nach dem Gleichheitszeichen keine Leerzeichen stehen dürfen.

15.3.3 Transaktionen

Normalerweise wird nach jedem *Post*-Befehl eine Transaktion abgeschlossen und eine neue begonnen. Sollen mehrere Anweisungen zu einer Transaktion zusammengefaßt werden, dann muß explizit mit *StartTransaction* eine Transaktion begonnen werden. Mit den Methoden *Commit* und *Rollback* wird diese Transaktion bestätigt oder zurückgenommen, in jedem Fall aber beendet. Im Gegensatz zu den gleichnamigen SQL-Anweisungen sind hier keine Parameter erlaubt.

Den Abschottungsgrad gibt man in der Eigenschaft *TransIsolation* an. Hier stehen die Möglichkeiten *tiDirtyRead* (nur beim Informix-Server und bei Desktop-Datenbanken), *tiReadCommitted* und *tiRepeatableRead* (entspricht SNAPSHOT) zur Verfügung. Voreinstellung ist hier *tiReadCommitted*.

15.4 Zugriff auf STORED PROCEDUREs

Der Zugriff auf STORED PROCEDUREs hängt stark vom verwendeten Server ab. Während Sie bei InterBase alles mit der Komponente *TQuery* erledigen können, müssen sie bei anderen Servern bisweilen auch die Komponente *TStoredProc* verwenden.

15.4.1 Zugriff mit TQuery

Je nachdem, ob eine STORED PROCEDURE eine Ergebnismenge zurückgeben oder Datensätze ändern soll, müssen die SQL-Anweisungen SELECT oder EXECUTE PROCEDURE verwendet werden.

Zugriff mit SELECT

Prozeduren, welche Ergebnismengen zurückgeben, erkennt man daran, daß in einer FOR...SELECT-Schleife eine SUSPEND-Anweisung aufgerufen wird. Das folgende Beispiel dafür wurde schon besprochen:

```
CREATE PROCEDURE p_faktab
RETURNS (nummer INTEGER, fakultaet INTEGER,
  vornamen VARCHAR(20), nachnamen VARCHAR(20))
AS
BEGIN
  FOR SELECT nummer, vornamen, nachnamen FROM t_test2
    INTO :nummer, :vornamen, :nachnamen
  DO
  BEGIN
```

```
    EXECUTE PROCEDURE p_fakult :nummer
      RETURNING_VALUES :fakultaet;
    SUSPEND;
  END
END
```

Auf eine solche STORED PROCEDURE würde man mit einer SELECT-Anweisung zugreifen

```
SELECT * FROM p_faktab
```

Selbstverständlich sind hier auch Optionen erlaubt, die bei einer SELECT-Anweisung vorgesehen sind. (WHERE, ORDER BY...). Die *TQuery*-Komponente kann schon zur Laufzeit geöffnet werden, auf sie wird wie gewohnt mit einer *TDataSource* zugegriffen, schon zur Entwurfszeit können die Daten betrachtet werden.

Versuchen Sie bitte nicht, die Eigenschaft *RequestLive* auf *true* zu stellen, dies würde die Fehlermeldung *Tabelle nicht vorhanden* nach sich ziehen. (Schließlich kann man ja wohl nicht ernsthaft erwarten, daß so eine Abfrage-Prozedur irgendwo Daten einfügt, ändert oder löscht.)

Zugriff mit EXECUTE PROCEDURE

Das folgende Beispiel zeigt eine einfache Prozedur, die Änderungen am Datenbestand ausführt. Kennzeichnend dafür sind INSERT-, UPDATE- oder DELETE-Anweisungen.

```
CREATE PROCEDURE p_test (neu VARCHAR(20))
AS
BEGIN
  UPDATE t_test2 SET vornamen = :neu
    WHERE nummer = 11;
END
```

Das folgende Listing zeigt, wie man diese Prozedur mit *ExecSQL* ausführt

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Query1->SQL->Clear();
  Query1->SQL->Add("EXECUTE PROCEDURE p_test ('Ramona')");
  Query1->ExecSQL();
}
```

15.4.2 Zugriff mit TStoredProc

Bei manchen Servern wie beispielsweise InterBase gibt es keinen vernünftigen Grund, die Komponente *TStoredProc* einzusetzen, zumal sie auch nur bei Prozeduren funktioniert, die eine Datenänderung durchführen. Bei Prozeduren, die eine Ergebnismenge zurückgeben, scheitert ihr Einsatz daran, daß kein Cursor-Handle erstellt werden kann und deshalb eine entsprechende Fehlermeldung ausgegeben wird.

Kann der Server eine Ergebnismenge mit einem Cursor liefern, dann kann die *TStoredProc*-Eigenschaft *Active* auf *true* gesetzt werden und über eine *TDataSource*-Komponente wie auf eine Abfrage zugegriffen werden. *TStoredProcedure* ist von *TDBDataSet* abgeleitet und verfügt somit über viele Eigenschaften, die wir von *TTable* und *TQuery* her kennen.

Bei Prozeduren, die eine Datenänderung bewirken, wird die Methode *ExecProc* aufgerufen, die Eingabeparameter können zur Entwurfszeit der Eigenschaft *Params* oder zur Laufzeit mit der Methode *ParamByName* zugewiesen werden.

Wenn Sie auf *Oracle*-Server zugreifen, dann sollten Sie sich die Eigenschaft *Overload* ansehen, bei *Sybase*-Servern werden Sie vermutliche die Methode *GetResults* benötigen; näheres in der Online-Hilfe.

15.5 Events

Einen besonderen Vorzug von InterBase habe ich Ihnen bislang vorenthalten: die Events. Hier wurde die Möglichkeit implementiert, daß die Datenbank von sich aus auf besondere Ereignisse aufmerksam macht. Dazu gleich ein Beispiel:

In manchen Firmen oder Verbänden ist es üblich, den hundertsten Mitarbeiter oder das tausendste Mitglied besonders zu feiern. Unabhängig davon, ob man dies für sinnvoll hält oder nicht: Die Aufgabe, diese besondere Person zweifelsfrei festzustellen, ist Aufgabe der datenverarbeitenden Abteilung.

Bei herkömmlicher Vorgehensweise würde man immer mal wieder die Anzahl der Mitarbeiter, der Kunden, der Mitglieder oder was auch immer feststellen (SELECT COUNT(nummer) FROM ...), um dann – hoffentlich – den gewünschten Datensatz zu ermitteln. Nach dem »Gesetz der größten Gemeinheit« wird nämlich zwischen zwei Abfragen nicht nur die fragliche Zahl überschritten, sondern es finden eine Reihe Zu- und Abgängen in der Reihenfolge statt, daß die Ermittlung des gesuchten Datensatzes in eine Denksportaufgabe ausartet.

Sinnvoller wäre es auf jeden Fall, wenn die Datenbank dieses Ereignis von sich aus feststellen und dem Anwender melden würde. Exakt zu diesem Zweck dienen Events. Das gerade angesprochene Beispiel wollen wir nun gleich in die Pra-

xis umsetzen – damit zuvor nicht ganz so viele Datensätze einzugeben sind, soll bei einer Datensatzzahl von zehn der Event ausgelöst werden. Dazu wird zunächst folgender Trigger erstellt:

```
CREATE TRIGGER e_namen FOR t_namen
AFTER INSERT
AS
DECLARE VARIABLE num INTEGER;
BEGIN
  SELECT COUNT(nummer) FROM t_namen INTO :num;
  IF(num = 10)
  THEN POST_EVENT "Zehntes Mitglied";
END
```

Bei einer Datensatzzahl von zehn wird der Befehl POST_EVENT aufgerufen, der das Event *Zehntes Mitglied* auslöst. Damit C++Builder auf dieses Event reagieren kann, muß die Komponente *TIBEventAlerter* eingefügt werden, die sich auf der Seite *Beispiele* befindet. Ist diese Komponente noch nicht registriert, dann muß die Datei *$(BCB)\examples\controls\sampreg.cpp* in die Komponentenbibliothek eingebunden werden. Dem *TIBEventAlerter*-Ereignis *OnEventAlert* wird dabei die folgende Ereignisbehandlungsroutine zugewiesen:

```
void __fastcall TForm1::IBEventAlerter1EventAlert(TObject *Sender,
  AnsiString EventName, long EventCount, bool &CancelAlerts)
{
  ShowMessage("Zehntes Mitglied");
}
```

Bild 15.4: Ein Event wird ausgelöst

Des weiteren müssen zwei Eigenschaften dieser Komponente gesetzt werden. Zunächst einmal muß die Eigenschaft *Database* auf eine *TDatabase*-Komponente gesetzt werden – eine solche muß zwingend eingefügt werden (das empfiehlt sich bei Client-Server-Datenbanken ohnehin). Zweitens muß das Event *Zehntes Mitglied* in die String-Liste *Events* aufgenommen werden.

Selbstverständlich lassen sich Events auch für »ernsthafte« Zwecke einsetzen. So kann ein Buchhaltungsprogramm beispielsweise über fällige Mahntermine informieren, das Mitgliederverwaltungsprogramm einer Partei über geänderte Deligiertenschlüssel. Bei längerem Nachdenken werden Ihnen sicher noch viele Möglichkeiten einfallen.

16 Vorarbeiten zur Erstellung eines Datenmodells

Wie in diesem Buch schon mehrmals erwähnt wurde, gliedert sich das Programmieren einer Datenbankapplikation in zwei Teile: Zum einen muß ein Datenmodell erstellt und die entsprechenden Tabellen müssen definiert werden (mit der Datenbankoberfläche oder ISQL), zum anderen muß die eigentliche Anwendung programmiert werden (dazu wird der C++Builder verwendet).

Durch die C++Builder-Komponenten ist das Erstellen der Datenbankanwendung ziemlich simpel geworden. Für einen Großteil des Funktionsumfangs benötigt man nicht eine Zeile Quelltext. Auch die Probleme, die nicht per Komponenten und Objektinspektor vollständig gelöst werden können, sind harmlos im Vergleich zu anderen Programmieraufgaben (CAD-Programme, Spiele, usw.).

Dagegen ist bei Datenbankanwendungen die Erstellung eines zweckmäßigen Datenmodells ziemlich schwierig. Vor allem die »letzten zehn Prozent« zwischen einer weitgehend brauchbaren Datenbank und einer uneingeschränkt brauchbaren Datenbank bereiten meist die größten Schwierigkeiten.

Im Gegensatz zu Karteikarten gibt es bei Datenbanken keine Möglichkeit, etwas »an den Rand« zu schreiben. Wenn eine bestimmte Funktion nicht vorgesehen ist, dann ist sie nun einmal nicht vorhanden.

Beim zweiten großen Beispielprojekt wollen wir uns hauptsächlich auf die Projektierung des Datenmodells beschränken. Die Erstellung der eigentlichen Datenbankanwendung wird dagegen nicht behandelt, entsprechende Lösungsansätze sind in den vorhergehenden Kapiteln zu finden.

16.1 Mitgliederverwaltung der NDW

Wir wollen in diesem Beispielprojekt die Mitgliederverwaltung für eine politische Partei erstellen. Damit bei dieser Thematik keine Sym- oder Antipathien von der Sache ablenken, soll dazu eine fiktive Partei herangezogen werden, welche wir Neue Deutsche Welle (NDW, »Wir wollen Spaß«) nennen wollen.

Für den Fall, daß eine Datenbankanwendung für einen fremden Auftraggeber erstellt werden soll, ist es entscheidend, zusammen mit ihm möglichst genau die Geschäftsabläufe zu erfassen (und gegebenenfalls zu verbessern), damit dann daraus das Datenmodell entwickelt werden kann.

Damit auch diese Phase eines solchen Projektes erläutert werden kann, wollen wir davon ausgehen, daß wir von der NDW den Auftrag erhalten hätten, eine Datenbankanwendung für die Mitgliederverwaltung zu erstellen.

16.1.1 Die Anfrage

In der Praxis würde es vermutlich so aussehen, daß die NDW bei verschiedenen kleineren Softwareherstellern um ein Angebot nachsucht. Zu diesem Zeitpunkt wird über den Funktionsumfang noch nicht besonders viel Klarheit herrschen. Auf der anderen Seite möchte man einen relativ präzisen Kostenvoranschlag, denn die Angebote der einzelnen Firmen sollen verglichen werden, und außerdem möchte der Schatzmeister gerne wissen, welche Summe er einplanen muß. Aus diesem Grund hat man auch eine Abneigung gegen eine Honorierung nach benötigter Arbeitszeit.

Dies Situation ist nun wahrlich kein Spezifikum von politischen Parteien, sondern würde sich bei jedem Betrieb wohl ähnlich abspielen. Wie Sie auf dieser Grundlage ein Angebot kalkulieren sollen, kann ich Ihnen beim besten Willen nicht erklären.

16.1.2 Informationsbeschaffung

Um den Umfang des Auftrags besser einschätzen zu können, sollte man sich nun jede verfügbare Information beschaffen. Insbesondere folgende Fragen bedürfen einer Klärung:

- Besteht bereits eine Datenbank? Wenn ja, wie sieht das Datenmodell aus? Soll vielleicht nur eine andere Benutzeroberfläche geschaffen werden, gegebenenfalls im Zuge eines Betriebsystemwechsels? Welche Funktionen sollen eventuell hinzugefügt werden?

  Wenn noch keine EDV existiert, wie werden die Daten dann momentan verwaltet? Welche Strukturen sind hier vorhanden? Wie sollen die Daten in den Rechner eingegeben werden?

- Wie sieht die Hardware-Struktur aus? Existiert vielleicht ein funktionierendes Netzwerk? Ist es erforderlich, von mehreren Arbeitsplätzen aus auf den Datenbestand zuzugreifen?

- Wie sehen die Betriebsabläufe aus? Welche Funktionen werden häufig benötigt, welche weniger häufig? Welche Informationen müssen aus der Datenbank abrufbar sein? Welcher Umfang des Datenbestandes ist zu erwarten?

Ausgangssituation

In unserem Beispiel sei folgende Ausgangssituation gegeben:

- Es existiert bereits eine dBase-Datenbank, welche den Anforderungen eher schlecht als recht entspricht. Um die Integrität der Daten zu erhalten, werden alle Eingaben von einem Rechner aus gemacht. Die Datenbankdateien werden hin und wieder auf die anderen Rechner gespielt, damit auch von dort die Daten abgefragt werden können.

  Ein Netzwerk hatte vor einigen Jahren zuviel Ärger verursacht und wurde deshalb wieder stillgelegt. Der eine Arbeitsplatz, auf dem die Dateneingabe erfolgt, entwickelt sich regelmäßig zum Engpass der ganzen Geschäftsstelle.

- Die Partei hat rund 2000 Mitglieder und etwa 10 000 Interessenten, deren Adressen zwecks Versendung von Rundschreiben auch in der Datenbank gespeichert werden sollen. Die Partei geht von einem Mitgliederwachstum in der nächsten Zeit aus, das System soll auf rund 10 000 Mitglieder und dementsprechend rund 50 000 Interessenten ausgelegt werden.

- Bei einer ersten Befragung kann man noch nicht erwarten, daß dem Kunden gleich alle Anforderungen an die Datenbankanwendung einfallen. Folgende Informationen werden jedoch häufiger benötigt und sollten sich möglichst einfach beschaffen lassen:

  – Alle zwei Monate wird eine Mitgliederzeitschrift (PS, »Politik und Spaß«) an alle Mitglieder und einige Interessenten verschickt. Der Postvertrieb wird von einer Firma abgewickelt, der alle Adressen im dBase-Format zur Verfügung gestellt werden.

  – Damit die Untergliederungen (Landesverbände, Kreisverbände) die Mitglieder anschreiben können, müssen an die Untergliederungen Adressendateien mit den jeweiligen Mitgliedern in diesem Verband weitergegeben werden. Diese Listen müssen sich auch ausdrucken lassen, weil nicht alle Untergliederungen mit EDV ausgestattet sind.

  – Für Einladungen zu Parteitagen und Vorstandssitzungen müssen die Unterlagen gezielt an die jeweiligen Delegierten und Vorstandsmitglieder versandt werden.

  – Um Mitgliedsbeiträge anzumahnen, sollten die Adressen säumiger Mitglieder automatisch ausgeben werden.

16.2 Anforderungen aus Satzung und Nebenordnungen

Als nächstes wird man daran gehen, die Satzung und die Nebenordnungen zu studieren und mit einem Textmarker die Stellen hervorzuheben, die Auswirkungen auf das Programm haben könnten. Da zu Beginn die Brisanz aller Textstellen noch längst nicht klar sein dürfte, muß diese Arbeit während der gesamten Projektierungsphase einige Male wiederholt werden.

Wenn Sie so etwas üben wollen, dann fordern Sie einfach bei irgendeiner Partei diese Papiere an – da wesentliche Teile schon vom Parteiengesetz vorgegeben sind, fallen die Unterschiede zwischen den Satzungen der verschiedenen Parteien nicht allzu gravierend aus.

16.2.1 Anforderungen aus der Satzung

Wir werden nun so vorgehen, daß einige Passagen der Satzung zitiert werden. Dazu wird jeweils erklärt, welche Auswirkungen sie auf das Programm haben. Um hier nicht die Seiten mit Irrelevantem zu füllen, sollen lediglich die Passagen zitiert werden, die für die Mitgliederverwaltung von Bedeutung sind.

Mitgliedschaft

Mitglied kann jede Person werden, welche die deutsche Staatsbürgerschaft oder ihren ständigen Wohnsitz in der Bundesrepublik hat, mindestens 16 Jahre alt ist und die Satzung sowie das Grundsatzprogramm anerkennt.

Mitglied können also auch Personen werden, welche nicht in Deutschland leben. Deshalb müssen bei der Adressentabelle Felder für das Land und das Landeskürzel, welches der Postleitzahl vorangestellt wird, vorgesehen werden. Des weiteren sollte das Programm anhand des Geburtsdatums automatisch prüfen, ob die Person, welche die Mitgliedschaft beantragt, mindestens 16 Jahre alt ist.

Die Mitgliedschaft endet durch Austritt, Ausschluß, Streichung oder Tod.

Der Austritt ist jederzeit möglich. Er muß schriftlich erklärt werden und bedarf keiner Begründung. Er ist sofort wirksam und entbindet von weiterer Beitragszahlung. Ein bei Beendigung der Mitgliedschaft bereits entrichteter Betrag wird nicht zurückgezahlt.

Die Streichung erfolgt durch den zuständigen Landesverband, wenn das Mitglied nach mindestens einjährigem Zahlungsrückstand trotz zweimaliger Aufforderung mit Fristsetzung und Hinweis auf eine mögliche Streichung den fälligen Beitrag nicht vollständig bezahlt hat.

Zunächst muß hier geklärt werden, was weitere Beitragszahlungen sind – dieser Passus läßt sich nämlich auf zwei verschiedene Weisen interpretieren:

- Der Austritt entbindet nur von Beitragszahlungen, die für die Mitgliedschaft nach dem Austrittsdatum fällig wären.
- Der Austritt entbindet zusätzlich von den Beitragszahlungen, welche für die Zeit der Mitgliedschaft fällig gewesen wären, aber noch nicht geleistet worden sind.

In solchen und ähnlichen Fällen brauchen Sie gar keine großartigen Überlegungen darüber anzustellen, was denn nun eigentlich richtig wäre – bei Interpretationsmöglichkeiten ist grundsätzliche eine Entscheidung des Auftraggebers herbeizuführen.

Des weiteren muß im Programm eine Option vorgesehen werden, welche überfällige Beiträge weitgehend automatisch anmahnt.

Rechte und Pflichten der Mitglieder

Jedes Mitglied hat das Recht, an der politischen Willensbildung der Partei mitzuwirken durch Beteiligung an Beratungen, Wahlen und Abstimmungen, durch Anträge im Rahmen der Gesetze und dieser Satzung in den Versammlungen der Partei, durch Beteiligung an der Aufstellung von Kandidatinnen und durch Bewerbung um eine Kandidatur, wie es die Wahlgesetze vorschreiben.

Jedes Mitglied hat die Pflicht ... den Beitrag pünktlich zu entrichten. Der Beitrag ist eine Bringschuld. Höhe und Zahlungsweise bestimmt der Bundesparteitag in der Finanzordnung.

Die Antrags-, Stimm- und Wahlrechte ruhen, wenn mit Ablauf des 30. Juni der Jahresbeitrag des laufenden Jahres nicht in voller Höhe bezahlt ist. Dies gilt auch für die Ausübung der Delegiertenrechte. Mit der Zahlung des Beitrags leben die genannten Rechte wieder auf.

Aus diesem Passus ergibt sich, daß den Gebietsverbänden Mitgliederlisten zur Verfügung zu stellen sind, aus denen sich ergibt, bei welchen Mitgliedern die Mitgliedsrechte ruhen. Den Termin mit dem 30. Juni behalten wir im Gedächtnis; das ergibt zusammen mit der Regelung aus der Finanzordnung, welche wir später durchsehen werden, eine etwas eigenartige Konstruktion.

Gliederungen

Die Partei gliedert sich in Kreis- und Landesverbände, zusammengeschlossen im Bundesverband. Orts-, Regional- und Bezirksverbände können mit Zustimmung des nächsthöheren Verbandes gebildet werden.

Der räumliche Geltungsbereich dieser Verbände deckt sich mit dem der entsprechenden politischen Gliederungen (Landkreise, Bundesländer...d. Verf). Ausnahmen bedürfen der Zustimmung des nächsthöheren Verbandes.

Für die einzelnen Gliederungen sind Mitgliederlisten (Papier oder Diskette) zu erstellen, von daher muß sich präzise erfassen lassen, welche Mitglieder zu welchem Verband gehören. Dabei schließt die Satzung nicht aus, daß es überlappende Verbände derselben hierarchischen Stufe gibt, daß also beispielsweise der Bezirk *Schwaben* sowohl zum Landesverband *Bayern* als auch zum Landesverband *Baden-Würtemberg* gehört (letzteres würde nicht der politischen Gliederung entsprechen und würde der Zustimmung des Bundesverbandes bedürfen).

Des weiteren wäre es auch denkbar, daß ein Kreisverband ein Gebiet umfaßt, das teils zum einen, teils zu einem anderen Landesverband gehört. Solche Konstruktionen sind natürlich nicht sinnvoll, und in diesen Fällen kann man durchaus von der Regel abweichen, daß die Datenbank-Anwendung alles ermöglichen muß, was auch in der Realität vorkommen könnte.

Bundesparteitag

Die stimmberechtigten Mitglieder des Bundesparteitages sind die Delegierten der Landesverbände und die Bundesvorstandsmitglieder.

Mit beratender Stimme teilnahmeberechtigt sind die Landesvorsitzenden, die Sprecherinnen der Bundeskommissionen und die Sprecherinnen der Bundesarbeitskreise im Rahmen der Beratung von Anträgen ihrer Arbeitskreise.

Die Landesverbände werden je angefangene 15 Mitglieder durch eine Delegierte vertreten. Im Verhinderungsfall muß sich eine Delegierte durch eine gewählte Ersatzdelegierte vertreten lassen.

Für die Delegiertenberechnung sind die Mitgliederzahlen der Landesverbände nach dem Stand von vier Monaten vor dem Bundesparteitag maßgebend. Von diesen Mitgliederzahlen ist die Zahl der Mitglieder abzuziehen, die an diesem Stichtag den Jahresbeitrag des Vorjahres nicht in voller Höhe bezahlt haben.

Die Delegierten und Ersatzdelegierten werden entweder auf den Landesparteitagen oder auf den Parteitagen bzw. Hauptversammlungen der zuständigen Untergliederungen für höchstens zwei Jahre gewählt. Näheres regeln die Satzungen der Landesverbände.

Nun wird es richtig kompliziert. Zunächst einmal müssen für den Bundesparteitag Einladungen, Anträge und Änderungsanträge verschickt werden, und zwar an die stimmberechtigten und mit beratender Stimme teilnahmeberechtigten Personen. Dabei sollten keine Fehler unterlaufen, weil sonst die auf dem Parteitag gefällten Beschlüsse wegen dieses Formfehlers angefochten werden könnten.

Für den Parteitag selbst muß sich dann auch zweifelsfrei feststellen lassen, welche der Teilnehmer stimmberechtigt und welche nur redeberechtigt sind. In der Praxis geschieht dies mit Stimmkarten unterschiedlicher Farbe, auf welche Etiketten mit dem jeweiligen Name des Teilnehmers geklebt werden. Diese Etiketten sollten sich automatisch mit dem Programm ausdrucken lassen. Dabei sollte

auf dem Etikett auch vermerkt, auf welcher Grundlage das Stimm- beziehungsweise Rederecht erteilt worden ist.

Des weiteren muß sich eindeutig feststellen lassen, wie viele Delegierte einem Landesverband zustehen. Auf den ersten Blick ist die getroffene Regelung vielleicht ein wenig kompliziert formuliert, aber doch eindeutig. Bei genauerem Hinsehen stellt sich die Frage, wann ein Beitrag vollständig bezahlt worden ist. Dazu ein Beispiel: Frau Mayer tritt 1993 in die NDW ein, bezahlt in diesem Jahr keinen Beitrag und 1994 den Jahresbeitrag in voller Höhe. Gilt nun der Jahresbeitrag 1993 bezahlt und der von 1994 nicht, dann zählt sie nicht zur Bemessungsgrundlage. Gilt dagegen 1993 als nicht bezahlt, jedoch 1994 als bezahlt, dann zählt sie zur Bemessungsgrundlage.

Als letztes Problem muß sich auch noch feststellen lassen, wie lange die letzte Delegiertenwahl zurückliegt, und ob die jeweiligen Delegierten überhaupt noch ordnungsgemäß im Amt sind.

Der Bundeshauptausschuß

Der Bundeshauptausschuß ist das Beschlußorgan zwischen den Bundesparteitagen (»kleiner Parteitag«).

Die stimmberechtigten Mitglieder des Bundeshauptausschusses sind die Delegierten der Landesverbände (wobei jeder Landesverband je angefangene 100 Mitglieder eine Delegierte stellt), die Bundesvorsitzende und deren beiden Stellvertreterinnen.

Mit beratender Stimme nehmen die Landesvorsitzenden, die Sprecherinnen der Bundesprogrammkommission und die restlichen Bundesvorstandsmitglieder teil.

Prinzipiell stellen sich hier die gleichen Aufgaben wie beim Bundesparteitag, lediglich die Zahl der teilnehmenden Personen ist geringer. Es ist zu beachten, daß hier bei der Delegiertenberechnung kein Unterschied zwischen beitragszahlenden und säumigen Mitgliedern gemacht wird, doch ist damit zu rechnen, daß dies bei der nächsten Satzungsänderung eingeführt wird. Außerdem ist es nicht auszuschließen, daß irgendwann die Delegiertenschlüssel geändert werden.

Wenn der Auftraggeber vorausschauend handelt, dann wird er fordern, daß sich die Delegiertenschlüssel im Programm ändern lassen und somit kein teures Update gekauft werden muß.

Sonstiges

Auf Bundesebene gibt es die Programm- und die Satzungskommission, deren Mitglieder gespeichert werden müssen, um bei Bedarf Schreiben an dieselben zu versenden, dasselbe gilt für die Bundesarbeitskreise.

Das Schiedsgericht kann das Ruhen von Mitgliedsrechten beschließen, auch dies sollte entsprechend berücksichtigt werden.

16.2.2 Anforderungen aus der Finanzordnung

In der Finanzordnung sind die finanziellen Angelegenheiten der Partei geregelt.

Mitgliedsbeiträge

Die Mitgliedsbeiträge werden durch den Bundesverband erhoben, und zwar auch für die Gebietsverbände.

Die Höhe der Mitgliedsbeiträge (soziale Staffelung) wird vom Bundesparteitag festgelegt. Zur Zeit betragen die Mitgliedsbeiträge jährlich 120,- DM (Regelbeitrag), 150,- DM (Familienbeitrag) sowie 24,- DM (Schülerinnen, Studentinnen, Wehr- oder Ersatzdienstleistende sowie Personen ohne oder mit geringem Einkommen).

Die Mitgliedsbeiträge sind fällig am 31. Januar für das erste Kalenderhalbjahr und am 31. Juli für das zweite Kalenderhalbjahr. Beim Beitritt in der ersten Jahreshälfte ist der volle, bei späterem Beitritt der halbe Jahresbeitrag zu entrichten.

Zunächst muß bei jedem Mitglied gespeichert werden können, welchen Beitrag es zu zahlen hat. Des weiteren müssen sich die geleisteten Zahlungen erfassen lassen, damit gegebenenfalls die ausstehenden Zahlungen angemahnt werden können und sich auch die Delegiertenzahlen korrekt berechnen lassen.

Ein etwas größeres Problem sind die Familien-Mitgliedschaften: Hier sind mehrere Personen Mitglied, es wird aber nur ein Beitrag gezahlt. Auch die Parteizeitung wird nur ein einziges Mal versandt – auf deren Etikett müssen dann aber alle Beteiligten der Familienmitgliedschaft aufgeführt werden, weil es großen Ärger geben kann, wenn hier Personen einfach unterschlagen werden.

Dem aufmerksamen Leser wird darüber hinaus aufgefallen sein, daß es in der Satzung eine Regelung gibt, die mit den Bestimmungen über die Fälligkeit der Mitgliedsbeiträge nicht ganz harmoniert. Die Mitgliedsrechte derjenigen Personen, die pünktlich am 31. Juli die zweite Rate des Jahresbeitrags bezahlen, würde eigentlich vom 30. Juni bis zum 31. Juli ruhen. Bei solchen Ungereimtheiten sollte man Rücksprache mit dem Auftraggeber nehmen – hier wäre es tatsächlich sinnvoll, die Satzung zu modifizieren.

Aufteilung der Beitragsanteile

Die eingehenden Mitgliedsbeiträge stehen zu 50% dem Bundesverband und zu 50% dem zuständigen Landesverband zu.

Die Hälfte der dem zuständigen Landesverband zufließenden Beitragsanteile sind an den zuständigen Kreisverband als Zuschuß weiterzuleiten. Für Mitglieder, die noch keinem Kreisverband angehören, ist dieser Zuschuß in nachfolgender Reihenfolge an den zuständigen Regional-, Bezirks- oder Landesverband weiterzuleiten.

Die Bundesgeschäftsstelle fertigt zu den Stichtagen 31. Dezember, 28. Februar und 31. August Aufstellungen über die Beitragseingänge, unter Nennung der Mitgliedsnamen, geordnet nach den Landes- und Kreisverbänden. Sie leitet diese Aufstellung bis zum 31. Januar, 31. März beziehungsweise 30. September an die zuständigen Landesverbände weiter.

Die Aufstellung vom 31. Dezember ist als Unterlage für die Landesrechenschaftsberichte vorgesehen. Aufgrund der Aufstellungen vom 28. Februar und 31. August überweist die Bundesgeschäftsstelle in den Monaten März und September die Hälfte der jeweils eingegangenen Mitgliedsbeiträge an die zuständigen Landesverbände, die ihrerseits die Zuschüsse unverzüglich an die zuständigen Gebietsverbände weiterleiten.

Bei der Mahnung beitragssäumiger Mitglieder werden die Kreis- und Landesverbände durch die Bundesgeschäftsstelle unterstützt.

Die Bundesgeschäftsstelle – und somit die Software – muß ermitteln können, welches Mitglied wann und in welcher Höhe Beiträge entrichtet hat. Auf dieser Grundlage werden dann die Beitragsanteile berechnet, welche an die Landesverbände weitergeleitet werden. Des weiteren muß berechnet werden, wie hoch die Zuschüsse sein müssen, welche die Landesverbände an die Kreisverbände weiterzuleiten haben.

So weit ist alles kein Problem, wenngleich auch reichlich aufwendig. Nun werden aber eine ganze Reihe von Mitgliedern den Kreis- und Landesverband wechseln, und sie werden das mit Sicherheit nicht gerade an den Stichtagen tun. Hier gibt es mehrere Möglichkeiten der Aufteilung:

- Der Beitragsanteil geht an den Landesverband, in dem das betreffende Mitglied am Stichtag seinen Erstwohnsitz hat.
- Der Beitragsanteil geht an den Landesverband, in dem das Mitglied die längere Zeit in dem halben Jahr vor dem Stichtag den Erstwohnsitz hatte.
- Die Beitragsanteile werden gemäß der Mitgliedsdauer zwischen den betreffenden Landesverbänden aufgeteilt.

Auch hier wird man wieder Rücksprache nehmen müssen, wie das nun in der Praxis gehandhabt wird. Auch muß geklärt werden, wie die Unterstützung der Kreis- und Landesverbände bei der Mahnung beitragssäumiger Mitglieder aussieht.

Spenden

Kreis-, Regional-, Bezirks- und Landesverbände sowie der Bundesverband sind zur Entgegennahme von Spenden und zur Ausstellung von Spendenbescheinigungen berechtigt.

Spenden gehen an den tatsächlichen Empfänger. Hat ein Spender einen anderen als Empfänger genannt, so ist der Spendenbetrag umgehend an diesen weiterzuleiten.

Erhält ein Ortsverband eine Spende, so hat er den vollen Betrag unverzüglich an den Kreisverband weiterzuleiten, der dem Spender eine Spendenbescheinigung ausstellt und den vollen Betrag ausschließlich für Zwecke der betreffenden Ortsverbandes zu verwenden hat.

Die Gebietsverbände haben gemäß §24 Abs 1 Parteiengesetz die Pflicht, Listen über alle Spendeneingänge zu führen, in denen Name, Vorname und Adresse des Spenders sowie Eingangsdatum und Betrag jeder Einzelspende aufgelistet ist. Spenden von nicht feststellbaren Personen sind gesondert auszuweisen.

Insbesondere der letzte Absatz wird für Probleme sorgen. Der Grund dafür sind die Modalitäten der Parteienfinanzierung, welche den Parteien staatliche Zuschüsse für jede Spende einer Privatperson gewähren. Die Berechnung dieser Zuschüsse ist reichlich kompliziert und soll hier nicht erörtert werden.

Die Regelung der Parteienfinanzierung zwingen jedoch die Parteien dazu, Listen mit sämtlichen Spendern und der Höhe deren Einzelspenden sowie deren Gesamtspendensumme vorzulegen. Das ist vor allem deswegen ein Problem, weil diese Personen möglicherweise mehreren Verbänden gespendet haben, und davon muß nun die Gesamtsumme gebildet werden. Das wäre weiter kein größeres Problem, wenn sichergestellt wäre, daß die Adresse stets gleich geschrieben würde, aber auch davon darf man nicht ausgehen. Zur Freude aller Beteiligten wird der Rechenschaftsbericht, dessen Bestandteil diese Liste ist, von einem vereidigten Rechnungsprüfer geprüft und dann bei der Bundestagsverwaltung eingereicht – Fehler sollten deshalb nicht auftreten.

Das Problem läßt sich wesentlich entschärfen, wenn alle Spendeneingänge mit Hilfe der Mitgliederverwaltung verbucht werden. Dort werden dann nicht Name und Adresse, sondern die Mitgliedsnummer angegeben. So ist zumindest bei den Mitgliedern gewährleistet, daß sich die korrekten Summen bilden lassen, und meist treten nur bei den Mitgliedern Spenden an unterschiedliche Gebietsverbände auf.

Parteifremde Personen, welche eine Spende machen, könnte man auch in die Mitgliederdatenbank aufnehmen, und zwar mit einem besonderen Status, beispielsweise mit *I* wie Interessent. Hier ergibt sich nun das Problem, daß diese immer dann in die Datenbank aufgenommen, wenn sie eine Spende tätigen. Soll also ein neuer Spender in die Mitgliederverwaltung aufgenommen werden, dann muß geprüft werden, ob er nicht bereits vertreten ist, vielleicht auch unter einem anders geschriebenen Namen oder mit einer anderen Adresse. Mit dem Problem der geringfügigen Abweichungen beim Schreiben der Adresse haben wir uns bereits in Kapitel 13 beschäftigt.

Des weiteren sollte das Programm automatisch eine Spendenbescheinigung erstellen, wenn eine Spende verbucht wird.

16.3 Anforderungen aus dem Arbeitsablauf

Im nächsten Schritt sollte der Arbeitsablauf in der Partei und besonders in der Geschäftsstelle genau beobachtet werden. Häufig wird man erleben, daß bei ein und demselben Vorgang gänzlich verschieden verfahren wird. Meist werden weder genaue Durchführungsvorschriften existieren, noch zwingt eine EDV die Beteiligten, sich an gewisse Spielregeln zu halten – die Erfahrung zeigt, daß diese Freiheiten meist auch gründlich genutzt werden.

Hier wird man die undankbare Aufgabe übernehmen müssen, die entsprechende Kreativität zu beschneiden. Selbstverständlich wäre es auch möglich, die Anwendung so zu programmieren, daß ein und derselbe Vorgang auf grundverschiedene Arten und Weisen erledigt werden kann. Der damit verbundene Mehraufwand – insbesondere dabei, die Abfragen entsprechend zu gestalten – wird in der Regel in indiskutable Dimensionen steigen und von keinem Kunden mehr bezahlt werden.

Wir wollen nun anhand einiger Beispiele betrachten, wie man verschiedene Varianten eines Arbeitsganges unter einen Hut bringt.

16.3.1 Neueingabe eines Mitgliedes

Als erstes wollen wir betrachten, wie Neumitglieder in die Datenbank gelangen.

In der Satzung ist es so geregelt, daß die Aufnahme beim jeweiligen Kreisverband beantragt wird – wo dieser nicht existiert, beim Landesverband. Der Kreis- respektive Landesverband befindet über die Aufnahme und meldet dann das Ergebnis an die Bundesgeschäftsstelle.

In der Praxis läuft das so, daß die meisten Beitrittsanträge bei der Bundesgeschäftsstelle eingehen, schließlich haben die wenigsten Interessenten eine Satzung vorliegen, und wenn doch, dann haben sie bestimmt nicht besonders intensiv darin gelesen.

Die Bundesgeschäftsstelle schickt den Aufnahmeantrag nun an den zuständigen Landes- oder Kreisverband, dessen Vorstand bei der nächten Sitzung über die Aufnahme entscheidet. Nun gibt es eine Reihe von Kreis- bzw. Landesvorständen, die eher selten tagen, und hin und wieder geht der Antrag auch im Papierkram des Schriftführers unter, so daß er verzögert behandelt wird oder die Meldung über den Beschluß nicht sofort – wenn überhaupt – an die Bundesgeschäftsstelle geht.

Oft genug weiß auch die linke Hand nicht, was die rechte tut. Der Landesgeschäftsführer verzichtet auf eine entsprechende Meldung, schließlich steht ja alles im Protokoll, der Schriftführer wiederum erspart es sich, die Protokolle nach

Berlin zu schicken, um die ohnehin knapp besetzte Bundesgeschäftsstelle nicht unnötig mit Papierkram zu belasten. Alternativ werden die Protokolle das ganze Jahr über gesammelt und nach jedem Landesparteitag zusammen mit dessen Protokoll an die Bundespartei geschickt.

Der Bundesgeschäftsstelle sind diese Schlampereien bekannt, und da man die Neumitglieder nicht gleich vergraulen will – schließlich möchte man ja größer werden –, werden sie einfach sofort in die Datenbank aufgenommen und bekommen somit auch die Parteizeitung zugeschickt. Das bringt es dann mit sich, daß von manchen Personen schon der Beitrag eingezogen worden ist, obwohl über deren Aufnahme noch gar nicht beschlossen wurde.

Liebe Leserinnen und Leser, wenn Sie sich nun fragen, ob Sie überhaupt noch im richtigen Buch sind: dies soll jetzt kein Kapitel über die Zustände bei kleinen politischen Parteien werden. Vielmehr wollte ich mit dieser Darstellung aufzeigen, daß in vielen Fällen vollständig anders verfahren wird, als es nach der Satzung (oder was auch immer Ihnen Schriftliches in die Hand gegeben wird) vorgesehen ist.

Als Außenstehender können Sie so etwas nicht wissen, Sie müssen es aber bei der Programmierung beachten, sonst werden Ihre Kunden mit der Anwendung nicht besonders glücklich werden. (Ein gutes Programm ist kein Programm, daß der Programmierer für gelungen hält, sondern eins, daß der Anwender für ein gutes Programm hält.) Die einzige Chance, den Anspüchen zu genügen, ist das Stellen von Fragen. Fragen Sie einfach alles, und fragen Sie vor allem mehrere Personen – die Ergebnisse werden wahrscheinlich »interessant« sein.

Der Status B

Zurück zu unserem Beispiel: Nehmen wir an, Sie hätten nun das Personal der Geschäftsstelle befragt und herausgefunden, daß Herr Maier die Adressen von Beitrittsanträgen immer gleich in die Datenbank eingibt, während Frau Müller immer erst auf den Aufnahmebeschluß des zuständigen Gremiums wartet. Frau Schulze verfährt dagegen nach der Methode, daß bei eher schlampig arbeitenden Verbänden (»ich kenne meine Pappenheimer«) sofort eingegeben wird, bei zuverlässigen Verbänden dagegen auf den Aufnahmebeschluß gewartet wird.

Ihre Aufgabe wäre es nun, hier ein sinnvolles (in der Praxis durchführbares und in der Datenbank praktikables) Verfahren vorzuschlagen und alle Beteiligten davon zu überzeugen, daß es nur so und nicht anders geht. Hier sieht das dann so aus, daß in jeden Personendatensatz ein String-Feld mit der Länge von einem Zeichen eingefügt wird. Der dort aufgeführte Buchstabe zeigt an, welchen Status die Adresse hat.

- Mit einem *M* werden alle regulären Mitglieder gekennzeichnet.
- Personen, die einen Aufnahmeantrag gestellt haben, über deren Aufnahme aber noch nicht entschieden worden ist, werden mit einem *B* (wie *beantragt*) gekennzeichnet.
- Mitglieder, deren Mitgliedsrechte aufgrund von Schiedsgerichtsurteilen ruhen, werden mit einem *R* versehen.
- Laut Satzung (und Parteiengesetz) können erst Personen ab 16 Jahren Mitglieder von Parteien werden. Nun kommt es aber hin und wieder vor, daß Jugendliche schon vorher die Mitgliedschaft beantragen. Hier muß zunächst dafür gesorgt werden, daß dies vom Programm automatisch erkannt wird, indem das auf dem Beitrittsformular anzugebende Geburtsdatum mit dem aktuellen Datum verglichen und gegebenenfalls eine Meldung ausgegeben wird.

  Traditionell wurde in solchen Fällen bisher so verfahren, daß die betreffenden Jugendlichen sofort in den Verteiler der Parteizeitung *PS* aufgenommen wurden, daß der Eintritt aber erst mit dem 16. Geburtstag wirksam wurde und auch erst ab diesem Zeitpunkt der Beitrag eingezogen worden ist. Dieser Status soll in der Mitgliedertabelle mit einem *J* gekennzeichnet werden.

16.3.2 Löschen eines Mitgliedes

Die Mitgliedschaft in der Partei endet durch Austritt, Ausschluß, Streichung oder Tod. Dann kann der Datensatz aber noch lange nicht aus der Datenbank gelöscht werden, denn schließlich handelt es sich um eine relationale Datenbank, und deshalb werden sehr wahrscheinlich Referenzen auf Mitgliederdatensätze bestehen.

Nehmen wir einmal an, in der Datenbank würde eine Tabelle der parteiinternen Gremien (Vorstände, Kommissionen, Schiedsgerichte, usw) existieren, welche Referenzen auf die Mitgliedertabelle bildet. Würde nun bei einem dieser Mitglieder die Mitgliedschaft enden, dann könnte man diesen Datensatz gar nicht löschen.

Prinzipiell könnte man auf die Idee kommen, daß man dann halt auch alle Datensätze löscht, welche eine Referenz auf diesen Datensatz bilden. Dann würden aber schnell Situationen auftreten, in der die Zahl der in der Datenbank aufgeführten Vorstandsmitglieder nicht mit der Zahl der satzungsmäßig vorgeschriebenen Vorstandsmitglieder übereinstimmt, und solche Situationen sollten tunlichst vermieden werden.

Es bleibt also nur die Möglichkeit, die Datensätze in der Datenbank zu belassen und für diese Fälle neue Stati einzuführen:

- Mit *A* werden Mitglieder gekennzeichnet, welche ihren Austritt aus der Partei erklärt haben.
- Mit *U* werden die ausgeschlossenen und mit *S* die gestrichenen Mitglieder gekennzeichnet (ausgeschlossen wird ein Mitglied wegen parteischädigenden Verhaltens, gestrichen, wenn jahrelang kein Beitrag gezahlt worden ist oder wenn sich die Adresse nicht mehr ermitteln läßt).
- Verstorbene Mitglieder erhalten den Status *T*.

Die Vorgehensweise mit dem Status-Feld hat noch einen weiteren Vorteil: Wenn die Datensätze nicht wirklich gelöscht werden, dann ist es einfacher, eine solche Aktion rückgängig zu machen, wenn fälschlicherweise ein Ende der Mitgliedschaft eingegeben wurde. Das kann nicht nur durch Fehler in der Geschäftsstelle passieren – in meiner Zeit als Berliner Landesgeschäftsführer kam bei drei Mitgliedern die Post mit dem Vermerk *Empfänger verstorben* zurück; zwei der Adressaten leben noch heute.

16.4 Ausgabe von Informationen

Wie in diesem Buch bereits erwähnt, ist es kaum ein Problem, selbst größere Datenmengen irgendwo zu speichern. Deutlich schwieriger ist es meistens, daraus die gewünschte Information zu gewinnen. Deshalb sollte man feststellen, welche Informationen aus der Datenbank benötigt werden und in welcher Form sie vorliegen sollte, bevor man das Datenmodell erstellt.

16.4.1 Mitgliederlisten

Alle Verbände, vom Ortsverband bis zum Bundesverband, benötigen Mitgliederlisten (in gedruckter Form oder auf Diskette). Je nachdem, was mit diesen Listen getan werden soll, ist Mitgliederliste nicht gleich Mitgliederliste.

Grundlage für Mitgliederrundschreiben

Am häufigsten werden Adressenlisten wohl für Mitgliederrundschreiben benötigt. Hier gibt es auch wieder zwei Varianten: Die eine sind die satzungsgemäßen Versammlungen (Kreishauptversammlung, Landesparteitag), für die alle ordentlichen Mitglieder geladen werden müssen. Auf diesen Versammlungen werden in der Regel Wahlen durchgeführt (für den Vorstand oder auch Kandidaten für öffentliche Wahlen). Hin und wieder paßt einigen Mitgliedern das Ergebnis nicht, und sie fechten die Wahl an, was am einfachsten zu machen ist, wenn Formfehler nachgewiesen werden können.

Ein solcher Formfehler würde vorliegen, wenn nicht alle ordentlichen Mitglieder eingeladen worden wären. Das Programm muß also eine Liste aller ordentlichen Mitglieder erstellen können. Dazu gehört auch, daß bei Familienmitgliedschaften alle Einzelmitglieder auch eine eigene Einladung erhalten (sofern sie ordentliche Mitglieder sind). Keine ordentlichen Mitglieder sind diejenigen, die nur einen Zweitwohnsitz im jeweiligen Verband haben, die noch keine 16 Jahre alt sind oder deren Mitgliedsrechte ruhen. Ob auch diese Personen als Gäste eingeladen werden, hängt von den Gepflogenheiten des jeweiligen Verbandes ab.

Die anderen Mitgliederrundschreiben sind die Einladungen zu Veranstaltungen, in denen Vorträge gehalten, Spendenaufrufe getätigt oder Weihnachtsgrüße verschickt werden. In diesen Fällen genügt ein Schreiben pro Familie. Andererseits wäre es allerdings günstig, wenn auch nichtordentlichen Mitgliedern (z.B. Studenten, die am Studienort ihren Zweitwohnsitz haben) ein entsprechendes Schreiben zukäme. Es sind jedoch auch Fälle möglich, in denen ein Mitglied beispielsweise Erst- und Zweitwohnsitz im selben Landesverband hat – hier wäre ein zusätzliches Schreiben an den Zweitwohnsitz weniger sinnvoll.

Grundlage für Ausübung der Mitgliedsrechte

In einigen Fällen muß zudem festgestellt werden können, ob eine Person überhaupt Mitgliedsrechte ausüben darf, also z.B. an Wahlen teilnehmen oder Anträge stellen kann. Hier ist derselbe Personenkreis zu ermitteln wie bei den Einladungen zu satzungsgemäßen Versammlungen, jedoch ist hier die Listenform meist praktikabler als die Etikettenform.

Versand der Parteizeitschrift

Beim Versand des Parteiorgans handelt es sich im Prinzip um ein informelles Rundschreiben auf Bundesebene an alle Parteimitglieder, inklusive aller Anwärter, Jugendlicher und Personen, deren Mitgliedsrechte ruhen.

Aktualisierung lokaler Mitgliederdatenbanken

An dieser Stelle sei noch nicht entschieden, welches DBMS verwendet werden soll, und ob eine spezielle Version der Applikation an alle Untergliederungen verteilt werden soll. Ganz gleich wie dies nun auch gehandhabt wird, wird es immer Gebietsverbände geben, welche lieber mit ihrem eigenen System arbeiten. Dabei kann es sich um den altbewährten Zettelkasten oder um einen Rechner mit einem inkompatiblen Beriebssystem handeln, oder die zuständige Person verweigert konsequent den Umgang mit allen Programmen, die sich nicht exakt so wie ihre Textverarbeitung bedienen lassen.

Wenn in diesen Fällen die lokalen Datenbanken mit der Mitgliederliste abgeglichen werden müssen, dann schleichen sich oft Fehler ein. Hier wäre es dann sinnvoll, spezielle Aktualisierungslisten zu drucken, in denen lediglich alle Veränderungen aufgelistet sind.

16.4.2 Delegiertenlisten

Die Landesverbände schicken zum Bundesparteitag und zum Bundeshauptausschuß Delegierte. Deren Anzahl berechnet sich aus der Mitgliederzahl des jeweiligen Verbandes. Das Programm muß also feststellen können, wie viele Delegierte den einzelnen Verbänden zustehen. Dies würde man sinnvollerweise in Form einer kleinen Mitgliederstatistik ausgeben. Geordnet nach Landesverbänden wird hier folgendes aufgelistet:

- Wie viele Mitglieder zum Stichtag im jeweiligen Verband gemeldet sind.
- Wie viele haben davon ihren Beitrag vom Vorjahr vollständig bezahlt.
- Welche Zahl von Mitgliedern für den Bundesparteitag berechnet sich daraus.
- Wie viele Delegierte für den Bundeshauptausschuß stehen dem Verband zu.

Des weiteren soll in der Datenbank gespeichert werden, welche der Mitglieder als Delegierte oder Ersatzdelegierte gewählt worden sind. Aus dieser Information müssen sich dann Adreß-Etiketten mit allen Delegierten drucken lassen, damit sich die Parteitags- oder Bundeshauptausschuß-Unterlagen versenden lassen.

16.4.3 Vorstandslisten

Des weiteren sollen sich Listen der Vorstandsmitglieder ausgeben lassen. Diese Listen können einen Verband (alle Vorstandsmitglieder vom Kreisverband Ravensburg) oder eine Funktion (alle Schatzmeister) sowie deren Kombination (alle Vorsitzenden aus Bayern) umfassen.

Ähnliche Listen müssen sich von den einzelnen Parteigremien (Satzungs- und Programmkommissionen, Arbeitskreise, Schiedsgerichte, Kassenprüfer) anfertigen lassen. Generell sollte es immer möglich sein, solche Listen auch auf Etiketten auszudrucken.

16.5.4 Mahnlisten

Auch zur Anmahnung ausstehender Mitgliedsbeiträge werden Listen beziehungsweise Etiketten benötigt. In den Listen sollte dann vermerkt sein, wie hoch der ausstehende Beitrag ist und wann das letzte Mal gezahlt wurde.

Da Mahnschreiben in der Regel nur ein Blatt umfassen, ist es meist zweckmäßig, die Adressen gleich auf das Schreiben zu drucken und dann Fensterumschläge zu verwenden. Nun wird der Text des Mahnschreibens vielleicht zur Entwurfszeit noch nicht feststehen, und auf ein nachträgliches Ändern sind sowohl *ReportSmith* als auch die *QuickReport*-Komponenten nicht gerade ausgelegt. Hier wäre es dann sinnvoll, eine Textverarbeitung in das Programm zum implementieren.

16.5.5 Postverteiler

Es wird immer mal wieder Schreiben geben, die an einen heterogenen Verteiler gerichtet sind. Für diesen Fall sollen sich Postverteiler erstellen lassen. Dies sind frei definierbare Listen von Personen, an welche die Schreiben gerichtet werden sollen.

16.5 Systementscheidungen

Bevor wir nun daran gehen, das Datenmodell zu erstellen, sollen noch einige Systementscheidungen getroffen werden.

InterBase

Als Datenbank-Management-System soll das Programm *InterBase for Workgroups Server* verwendet werden. Aus der Anforderungsliste ergibt sich, daß die zu erstellende Anwendung mehrplatzfähig sein muß. Zudem ist davon auszugehen, daß häufig gleichzeitig von verschiedenen Arbeitsplatzrechnern aus auf den Datenbestand zugegriffen werden soll. Unter diesen Umständen wird sich die Verwendung eines Client-Server-Systems nicht vermeiden lassen. Da das System das Rückgrat der Bundesgeschäftsstelle bildet, sind die Ausgaben für ein Client-Server-System auch durchaus gerechtfertigt.

Für die Verwendung von InterBase spricht dessen gute Anbindung an C++Builder.

C++Builder

Aus dem Thema dieses Buches ergibt sich, daß der C++Builder verwendet werden soll (bei der Original-Ausgabe dieses Buches wurde analog dazu *Delphi* empfohlen). Da eine Clint-Server-Datenbank-Anwendung erstellt werden soll, wird folglich auch die Client-Server-Suite verwendet.

17 Entwicklung des Datenmodells

Aus den Vorgaben soll nun das Datenmodell entwickelt werden. Dabei möchte ich Ihnen keine fertige Lösung präsentieren, sondern das Datenmodell soll Schritt für Schritt erarbeitet werden. Auf diese Weise soll nicht nur erklärt werden, wie etwas gemacht wird, sondern auch, warum die Lösung so und nicht anders aussieht.

17.1 Die Mitgliederadressen

Zum Speichern der Mitgliederadressen wird man zunächst wohl die folgende Tabelle vorsehen:

Mitglieder	
Nummer	+
Vorname	A 30
Nachname	A 30
Strasse	A 50
PLZ	A 6
Ort	A 43
Land	A 50
Tel_1	A 20
Fax	A 20
Geburtstag	D
Beruf	A 20
Verband	I
Status	A 1

Bild 17.1: Der erste Entwurf der Mitgliedertabelle

Dabei handelt es sich um die Daten, die man in der Regel so vom Beitrittsantrag übernehmen kann. In der Regel wird man darüber hinausgehende Daten auch kaum benötigen.

Nun ist es erfahrungsgemäß so, daß die Ausnahmefälle den größten Ärger bereiten und die meiste Arbeit machen. Professionelle und weniger professionelle Programme unterscheiden sich eben auch dadurch, inwieweit sich damit kompliziertere Probleme lösen lassen. Die folgenden Beispiele beschreiben solche eher seltenen, aber anspruchsvollen Probleme:

- Der Landesvorsitzende von Schleswig-Holstein ruft in der Bundesgeschäftsstelle an und erfragt die Adresse von *Udo Maier*, der ihm vor zwei Jahren versprochen hat, Plakatentwürfe für die nächste Wahl zu erstellen. Nun hat Herr Maier neulich geheiratet und heißt nun *Udo Müller* – ein Mitglied namens *Udo Maier* gibt es nun nicht mehr. Mitglieder mit dem Vornamen *Udo* oder dem Nachnamen *Maier* gibt es dann wieder eine ganze Reihe. Um hier weiterzukommen, müssen alle alten Namen in der Datenbank gespeichert werden.

- Je nachdem, ob der Name für eine Adresse, als Anrede, für eine Mitgliederliste oder für eine Suche benötigt wird, muß ein akademischer Grad oder ein Adelsprädikat mit dem Vornamen oder mit dem Nachnamen kombiniert werden. Da die Partei nach Höherem strebt, sollen die Namen auch mit Anhängen wie *MdB*, *MdL* oder *MdE* kombinierbar sein.

- Frau *Jutta König* studiert in *Hamburg* und hat ihren Zweitwohnsitz bei Ihren Eltern in *Greifswald*. Da die Partei in Ostdeutschland noch nicht so stark vertreten ist, sollten Einladungen zu Veranstaltungen auch an alle Mitglieder gehen, die nur ihren Zweitwohnsitz im entsprechenden Verband haben und deshalb nicht Mitglied des entsprechenden Gebietsverbandes sind. Also sollte auch die Adresse der Zweitwohnung irgendwie in die Datenbank aufgenommen werden können.

- Ende des übernächsten Monats möchte Frau *König* von der *Bismarckstraße* in die *Hindenburgstraße* umziehen, und teilt dies der Bundesgeschäftsstelle jetzt schon mit. Würde die Adresse jetzt schon in der Datenbank geändert, dann würde die nächste Ausgabe der Parteizeitung *PS* schon an die neue Adresse geschickt – dort würde sie zu diesem Zeitpunkt noch gar nicht wohnen. Man könnte nun die Meldung auf Wiedervorlage legen und erst zum Umzugszeitpunkt eingeben.

  Nun wird aber jeden Monat der Datenbestand an die Untergliederungen weitergegeben, und wenn man hier mit der größtmöglichen Aktualität arbeiten möchte, dann müssen sich beide Adressen und das Umzugsdatum speichern lassen. Dies ist auch deshalb erforderlich, weil manchmal Mitglieder gesucht werden müssen, von denen nur eine veraltete Adresse bekannt ist.

- Das Mitglied *Holger Seitz* ist Organisationsbeauftragter der Partei und sollte deshalb immer telefonisch erreichbar sein. Zu diesem Zweck hat er ein Handy, das allerdings nicht immer funktioniert, sein Telefon zu Hause, zwei Nummern bei der Firma und eine bei seiner Freundin. Des weiteren hätte man auch gerne noch die Fax-Nummern von daheim und von der Firma sowie seine e-mail-Adresse gespeichert.

  Auch stellt sich hin und wieder das Problem, daß man Mitglieder anhand einer Telefonnummer identifizieren muß. Aus diesem Grund sollen auch alte Telefonnummern gespeichert werden. Das Datenmodell muß somit die Möglichkeit bieten, sowohl mehrere Nummern als auch alte Nummern zu speichern.

- Auch die Berufe und die Stati sind alles andere als konstant. Wenn man einen *Diplom-Geologen* zwecks Beantwortung einer Sachfrage sucht, dann möchte man, daß das Programm auch einen *Rentner* anzeigt, welcher früher in diesem Beruf gearbeitet hat.

17.1.1 Verknüpfungstabellen

Man könnte nun damit beginnen, bei jedem Namen, bei jeder Adresse und bei jeder Telefonnummer die Information hinzuzufügen, ab wann und bis wann diese gültig sein soll. Bei den Namen werden wir auch so verfahren, bei den anderen Informationen wäre dies aber aus zwei Gründen weniger zweckmäßig, wie hier am Beispiel der Adressen erläutert werden soll.

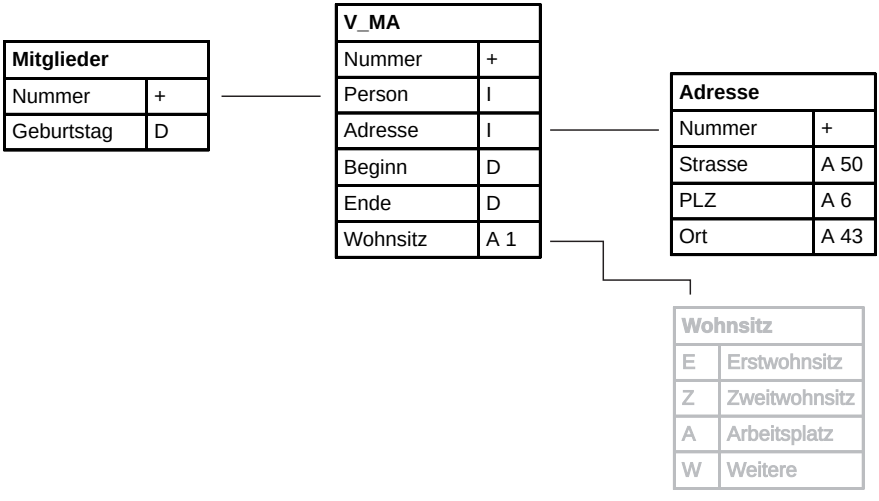


Bild 17.2: Verknüpfung zwischen Person und Adresse

- Zum einen wäre jeder Datensatz der Adressen-Tabelle fest einer Person zugeordnet. Bei einer Familienmitgliedschaft haben nun mehrere Personen dieselbe Adresse, so daß bei dieser Vorgehensweise Redundanzen entstehen würden – deren Problematik haben wir bereits in Kapitel 1 besprochen.
- Häufig wird bei einem Umzug die alte Adresse zum Zweitwohnsitz. Hier würde dann dieselbe Adresse erneut eingegeben, lediglich mit etwas geänderten Daten und der Information, daß es sich nun um den Zweitwohnsitz handelt.

Um diese Nachteile zu vermeiden, wird eine Verknüpfungstabelle zwischen den Tabellen *Person* und *Adresse* verwendet, so daß auf diese Weise jede Adresse nur einmal in der Tabelle *Adresse* enthalten sein muß. Bild 17.2 zeigt, wie diese Verknüpfung aufgebaut ist.

Nun wird man in der Regel dann wieder die derzeit gültige Erstwohnsitz-Adresse abfragen wollen. Damit hier die SQL-Anweisung nicht zu kompliziert wird, könnte man auf dem Server eine entsprechende VIEW erstellen.

```
CREATE VIEW adressen_2 (mitgliedsnummer, adressennummer,   strasse,
  plz, ort, wohnsitz) AS
SELECT p.nummer, a.nummer, a.strasse, a.plz, a.ort, w.namen
  FROM person p, adresse a, v_ma v, wohnsitz w
    WHERE (v.person = p.nummer)
      AND (v.adresse = a.nummer)
      AND (v.wohnsitz = wohnsitz.nummer)
      AND (v.ende IS NULL)
      AND (v.wohnsitz = "E")
```

Diese VIEW ist schon relativ aufwendig, dabei kommen Namen, Telefonnummern, Stati und Berufe erst noch hinzu. Es ist leicht einzusehen, daß der Anwender solche Abfragen dann nicht mehr selbst erstellen kann, sondern daß hier eine entsprechende VIEW oder eine STORED PROCEDURE definiert werden muß. Dann jedoch gestaltet sich die Abfrage wieder ganz einfach.

```
SELECT * FROM adressen_2
```

Bei der Erstellung der VIEW wurde davon ausgegangen, daß die jeweils aktuelle Adresse noch keinen Eintrag in der Spalte *v.ende* aufweist. Dies ist jedoch immer dann nicht richtig, wenn Adressenänderungen schon vor ihrem Eintreten eingegeben werden. Man müßte die VIEW dann so umformulieren, daß daraufhin geprüft wird, ob das heutige Datum zwischen den Daten *v.beginn* und *v.ende* liegt oder ob das heutige Datum später als *v.beginn* liegt und das Feld *v.ende* den Wert NULL aufweist.

Nach meiner Ansicht ist es jedoch sinnvoller, für diesen Zweck eine STORED PROCEDURE zu verwenden, der das jeweilige Datum übergeben wird.

```
CREATE PROCEDURE p_adressen_2
  (datum DATE, wsn VARCHAR(1))
RETURNS
  (mitgliedsnummer INTEGER,
  adressennummer INTEGER,
  strasse VARCHAR(50),
  plz VARCHAR(6),
  ort VARCHAR(43),
  wohnsitz VARCHAR(20))
AS
BEGIN
  FOR SELECT p.nummer, a.nummer, a.strasse, a.plz, a.ort, w.namen
    FROM person p, adresse a, v_ma v, wohnsitz w
    WHERE (v.person = p.nummer)
      AND (v.adresse = a.nummer)
      AND (v.wohnsitz = wohnsitz.nummer)
      AND (v.beginn < :datum)
      AND ((:datum < v.ende) OR (v.ende IS NULL))
      AND (v.wohnsitz LIKE :wsn)
    INTO :mitgliedsnummer, :adressennummer, :strasse, :plz, :ort,
      :wohnsitz
  DO SUSPEND;
END
```

Bei dieser STORED PROCEDURE wird als Parameter nicht nur das Datum, sondern auch die Wohnsitznummer übergeben. (Bei allen Nachschlagetabellen heißen die Spalten *nummer* und *namen*, wobei die Spalte *nummer* oft vom Typ VARCHAR(1) ist. Dies hat den Vorteil, daß die »Nummern« ein wenig anschaulicher sind, begrenzt aber die Zahl der Tabelleneinträge auf die Zahl der verwendbaren Zeichen.) Damit die STORED PROCEDURE alle aktuellen Erstwohnsitzadressen zurückgibt, wird die folgende Abfrage gestartet.

```
SELECT * FROM p_adressen_2("NOW","E")
```

Beachten Sie bitte, daß Sie das Schlüsselwort NOW in Anführungszeichen setzen müssen, da *InterBase* ansonsten nach einer Spalte dieses Namens suchen würde. Die Verwendung des LIKE-Operators ins Zusammenhang mit der Saplte *v.wohnsitz* erlaubt die Rückgabe der Adressen aller Wohnsitz-Typen.

```
SELECT * FROM p_adressen_2("NOW","%")
  ORDER BY wohnsitz
```

Man könnte nun auf die Idee kommen, eine VIEW mit dieser Abfrage zu definieren, doch hier stößt *InterBase* an seine Grenzen: Die Definition der VIEW funktio-

niert noch problemlos, doch beim Zugriff auf dieselbe meldet die Komponente *TTable* einen ungültigen Tabellennamen, bei der Verwendung von *TQuery* oder ISQL tritt eine allgemeine Schutzverletzung auf.

17.1.2 Der vollständige Tabellensatz

Bild 17.3 zeigt den vollständigen Tabellensatz für das Speichern der Mitglieder. Was man auf Desktop-Datenbanken mit einer Tabelle bewerkstelligt hätte – vergleiche Bild 17.1 – benötigt nun 14 Tabellen. Es ist wohl leicht zu verstehen, daß man für solche Gebilde keine einzelnen Abfragen erstellt. Wir werden nachher eine STORED PROCEDURE erstellen, welche diese Arbeit erledigt. Dabei tritt ein größeres Problem auf – Sie können sich ja bei der Besprechung der Tabellen schon einmal überlegen, wo dieses begründet liegt.

Beginnen wir mit der Tabelle *Name*. Im Gegensatz zu den Adressen, den Telefonnummern und den Stati soll hier keine eigene Verknüpfungstabelle erstellt werden, denn erstens wechseln Personen ihren Namen eher selten, und zweitens haben sie jeweils nur einen Namen. Der Mehraufwand für eine eigene Verknüpfungstabelle ist deshalb nicht gerechtfertigt, und somit sind die Felder *Beginn* und *Ende* direkt in dieser Tabelle enthalten.

Damit der Name stets korrekt zusammengesetzt werden kann, gibt es hier die Felder *Titel, Adel* und *Anhang*, für die jeweils eine Nachschlagetabelle angelegt wird. Dies sorgt für eine einheitliche Schreibweise, hat aber hier noch einen zweiten Grund: Beim Zusammensetzen des Namens wird man in der Regel SQL-Anweisungen verwenden, diese würden – ausschnittsweise betrachtet – folgendermaßen lauten

```
SELECT n.titel || " " || n.vorname...
```

Nun tritt dabei das Problem auf, daß bei Personen ohne Titel der Vorname jeweils mit einem Leerzeichen beginnt. Das sieht ähnlich merkwürdig aus, als wenn auf das Leerzeichen generell verzichtet würde. Die Lösung des Problems besteht darin, daß man in der SQL-Anweisung kein Leerzeichen einfügt, sondern jeweils dem Feld *Titel* eines beigibt. Nun wird der Anwender bei der Eingabe einer Adresse an alles mögliche denken, nur nicht daran, dem Titel ein Leerzeichen anzuhängen. Dies wird man dann nachträglich verbessern können, auf der anderen Seite wird es aber immer wieder falsch eingegeben. Auf diese Weise hat man eine Menge Arbeit und doch nie ein vernünftiges Ergebnis.

Mit der Verwendung einer Nachschlagetabelle kann man dies vermeiden. Zwar kann der Fehler auch hier auftreten, nämlich dann, wenn man diese Nachschlagetabellen um weitere Einträge ergänzt. Dies wird aber eher selten erfolgen, und wenn man es nachträglich korrigieren muß, dann nur an einer einzigen Stelle – nämlich in der Nachschlagetabelle.

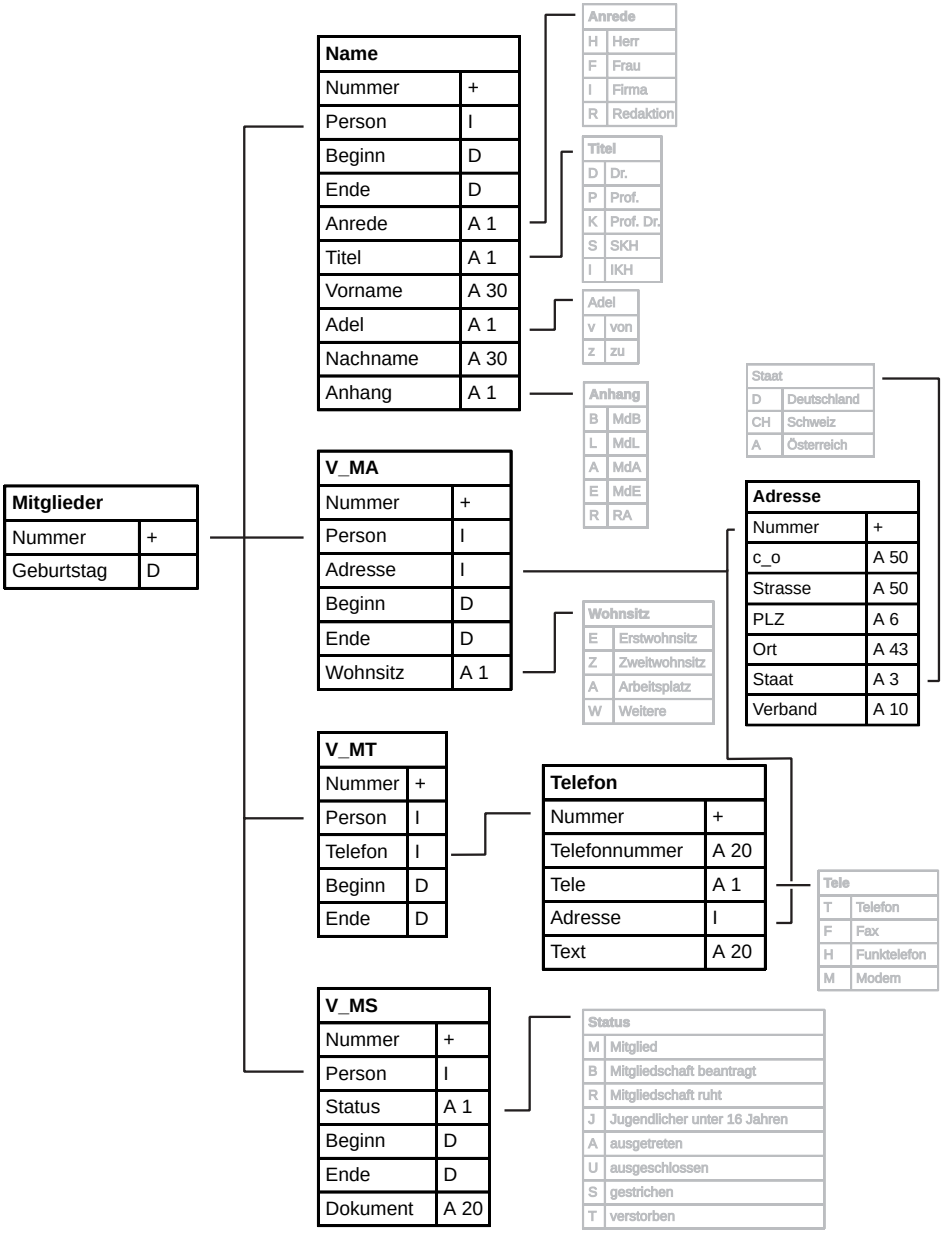


Bild 17.3: Alle Tabellen zum Speichern von Mitgliedern

Die Tabellen *Adresse*, *V_MA* und *Wohnsitz* haben wir bereits besprochen, ähnlich geht man bei den Tabellen *Telefon, V_MT* und *Tele* vor. Im Feld *Telefon.Adresse* kann eine Telefonnummer an eine Adresse gekoppelt werden, damit für den Fall des Ausdrucks einer Adressenliste die Telefonnummer ermittelt wird, welche zur jeweiligen Adresse gehört. Im Feld *Telefon.Text* kann eine kurze Bemerkung eingegeben werden. Beispielsweise könnte ein Mitglied irgendwo zur Untermiete wohnen, und am Telefon meldet sich dann *Schröder*. Mit einem Hinweis könnte der Anrufer darauf vorbereitet werden.

Wie schon in Kapitel 17 erläutert, gibt es verschiedene Stati der Mitgliedschaft. Beispielsweise könnten die Mitgliedsrechte nach einem Urteil des Schiedsgerichtes für sechs Monate ruhen. Damit nach Ablauf dieser Frist der Status nicht manuell abgeändert werden muß, wird auch hier eine Verknüpfungstabelle verwendet. In der Spalte *Dokument* könnte man dann beispielsweise das Aktenzeichen des Schiedsgerichtsurteils speichern.

17.1.3 Abfrage mit einer STORED PROCEDURE

Wir wollen nun erneut eine STORED PROCEDURE erstellen, welche aus einigen übergebenen Parametern (Datum, Wohnsitznummer, Mitgliederstatus) eine kompletten Adressensatz ermittelt. Im Prinzip handelt es sich dabei um die bereits vorgestellte Prozedur, welche entsprechend erweitert werden muß.

Dabei tritt allerdings – wie bereits erwähnt – ein größeres Problem auf: Manche der Mitglieder haben kein Telefon oder haben ihre Telefonnummer nicht angegeben. Würde man nun hier einen EQUI-JOIN verwenden, dann würden in der Mitgliederliste nur diejenigen Mitglieder vorkommen, von denen auch eine Telefonnummer bekannt ist. Außerdem könnte der Fall eintreten, daß ein Mitglied an ein und derselben Adresse mehrere Telefonnummern hat – in diesem Fall würde dann die komplette Adresse mehrmals aufgenommen.

Zurück zu den Mitgliedern ohne Telefon: Prinzipiell könnte man hier einen OUTER JOIN verwenden, eine entsprechende Anweisung würde etwas wie folgt aussehen:

```
SELECT DISTINCT
     p.nummer,
     n.nummer,
     ...
  FROM person p
     LEFT OUTER JOIN  v_mt ON (p.nummer = v_mt.person)
     LEFT OUTER JOIN telefon t ON (v_mt.telefon = t.nummer),
     namen n, anrede, titel, adel, anhang, v_ma, adresse a, staat,
       status s, v_ms
```

```
  WHERE (p.nummer = n.person)
    AND (n.anrede = anrede.nummer)
    ...
```

So weit funktioniert dies ganz brauchbar, doch nun würde man weitere Einschränkungen in die WHERE-Klausel aufnehmen müssen, damit keine alten Telefonnummern angezeigt werden, damit nur Telefon- und keine Faxnummern ausgegeben werden, usw. Hier gilt es, einen anderen Weg einzuschlagen.

```
CREATE PROCEDURE p_adressen_6
  (datum DATE,
  wsn VARCHAR(1),
  sn VARCHAR(1))
RETURNS
  (personennummer INTEGER,
  namennummer INTEGER,
  v_ma_nummer INTEGER,
  adressennummer INTEGER,
  v_mt_nummer INTEGER,
  telefonnummer INTEGER,
  v_ms_nummer INTEGER,
  anrede VARCHAR(20),
  titel VARCHAR(20),
  vorname VARCHAR(30),
  adel VARCHAR(20),
  nachname VARCHAR(30),
  anhang VARCHAR(20),
  c_o VARCHAR(50),
  strasse VARCHAR(50),
  plz VARCHAR(6),
  ort VARCHAR (43),
  staatkuerzel VARCHAR(3),
  staat VARCHAR(50),
  telefon VARCHAR(20),
  status VARCHAR(20)) AS
BEGIN
  FOR SELECT DISTINCT
    p.nummer,
    n.nummer,
    v_ma.nummer,
    a.nummer,
    v_ms.nummer,
```

```
    anrede.name,
    titel.name,
    n.vorname,
    adel.name,
    n.nachname,
    anhang.name,
    a.c_o,
    a.strasse,
    a.plz,
    a.ort,
    a.staat,
    staat.name,
    s.namen
  FROM person p, name n, anrede, titel, adel, anhang,
    v_ma, adresse a, staat, status s, v_ms
  WHERE (p.nummer = n.person)
    AND (n.anrede = anrede.nummer)
    AND (n.titel = titel.nummer)
    AND (n.adel = adel.nummer)
    AND (n.anhang = anhang.nummer)
    AND (n.beginn < :datum)
    AND ((:datum < n.ende) OR (n.ende IS NULL))
      AND (v_ma.person = person.nummer)
    AND (v_ma.beginn < :datum)
      AND ((:datum < v_ma.ende) OR (v_ma.ende IS NULL))
    AND (v_ma.wohnsitz LIKE :wsn)
    AND (v_ma.adresse = a.nummer)
    AND (a.staat = staat.nummer)
    AND (v_ms.person = person.nummer)
    AND (v_ms.beginn < :datum)
      AND ((:datum < v_ms.ende) OR (v_ms.ende IS NULL))
    AND (v_ms.status = s.nummer)
    AND (v_ms.status LIKE :sn)
  INTO
    :personennummer,
    :namennummer,
    :v_ma_nummer,
    :adressennummer,
    :v_ms_nummer,
    :anrede,
    :titel,
```

```
    :vorname,
    :adel,
    :nachname,
    :anhang,
    :c_o,
    :strasse,
    :plz,
    :ort,
    :staatkuerzel,
    :staat,
    :status
  DO
  BEGIN
    SELECT *
      FROM p_telefon_2(:datum, :personennummer, :adressennummer)
      INTO
        :telefon,
        :telefonnummer,
        :v_mt_nummer;
    SUSPEND;
  END
END
```

Erschrecken Sie nicht gleich über diese Prozedur, sie ist zwar lang, aber nicht weiter kompliziert. (Ein Tip am Rande: Verwenden sie ruhig für jeden Parameter und jede Spalte eine eigene Zeile. Solche Prozeduren werden Sie wohl kaum auf Anhieb fehlerfrei erstellen können, und bei der Fehlersuche werden Sie für das erhöhte Maß an Übersichtlichkeit dankbar sein.)

Nun zu unserem Problem mit den Telefonnummern: Die Ermittlung der Telefonnummern wurde aus der Abfrage ausgegliedert und in die Prozedur *p_telefon_2* gesteckt. Von der Struktur her sieht die Prozedur *p_adressen_6* folgendermaßen aus:

```
FOR SELECT
  ... {Equi-Join}
DO
BEGIN
  SELECT * FROM p_telefon_2 ... {Telefonnummer};
  SUSPEND;
END
```

Bei jedem Durchlauf durch die FOR SELECT-Schleife wird die Prozedur *p_telefon_2* aufgerufen. Dieser werden als Parameter das Datum, die Personennummer und die Adressennummer übergeben. Kann diese Prozedur eine Telefonnummer ermitteln, dann wird sie den entsprechenden Spalten zugewiesen, anderfalls wird in diesen Spalten der Wert NULL ausgegeben.

Die Prozedur *p_telefon_2* steht nun vor der Aufgabe, diejenige Telefonnummer auszugeben, die:

- am fraglichen Datum aktuell ist
- zu einem Telefon gehört
- zur fraglichen Person gehört
- zur entsprechenden Adresse gehört

Nun ist es nicht ausgeschlossen, daß mehrere Telefonnummern diesen Forderungen entsprechen. Hier wird nun davon ausgegangen, daß die zuletzt der jeweiligen Person zugewiesene Telefonnummer die aktuellste ist und somit angezeigt werden soll. Die zuletzt zugewiesene Telefonnummer hat – da hier die Nummern fortlaufend vergeben werden – den höchsten Wert in der Spalte *V_MT. Nummer*.

```
CREATE PROCEDURE p_telefon_2
  (datum DATE,
  person INTEGER,
  adresse INTEGER)
RETURNS
  (telefon VARCHAR(20),
  telefonnummer INTEGER,
  v_mt_nummer INTEGER)
AS
BEGIN
  SELECT t.telefonnummer, t.nummer, v_mt.nummer
    FROM v_mt, telefon t
    WHERE (v_mt.telefon = t.nummer)
      AND (v_mt.nummer =
        (SELECT MAX( v_mt.nummer)
          FROM v_mt, telefon t
          WHERE (v_mt.telefon = t.nummer)
          AND (v_mt.person = :person)
          AND (t.tele = "T")
          AND (t.adresse = :adresse)
          AND (v_mt.beginn < :datum)
          AND ((:datum < v_mt.ende) OR (v_mt.ende IS NULL))))
```

```
    INTO
      : telefon,
      : telefonnummer,
      : v_mt_nummer;
  SUSPEND;
END
```

Zunächst wird in einer Unterabfrage ermittelt, welches der höchste Wert in der Spalte *V_MT.Nummer* ist, der die oben genannten Anforderungen erfüllt. Mit diesem Wert werden nun die eigentliche Telefonnummer sowie die Primärindizes des Tabellen *V_MT* und *Telefon* ermittelt und als Ergebnis der STORED PROCEDURE zurückgegeben.

17.2 Die Mitgliedschaft

Die Verfasser der Finanzordnung haben mit der Einführung der Familienmitgliedschaft ein Problem geschaffen, das uns erhebliche Schwierigkeiten bereiten wird. Familienmitgliedschaft bedeutet, daß mehrere Personen Mitglied der Partei sind, aber nur einmal (pro Halbjahr) einen Beitrag entrichten und auch nur einmal die Parteizeitung *PS* bekommen.

Eine Familie umfaßt in der Mehrzahl der Fälle ein Ehepaar und somit zwei Personen. In einigen Fällen sind jedoch auch noch einige Kinder (über 16 Jahre) beteiligt, manchmal auch noch der Opa, die Tante, uns was sonst noch zusammen in einem Haushalt lebt. Studieren die Kinder, dann ist noch nicht einmal sichergestellt, daß alle Beteiligten der Familienmitgliedschaft denselben Erstwohnsitz haben. Leider definieren weder Satzung noch Finanzordnung, was denn nun genau unter einer Familie zu verstehen sei. Somit wird alles als Familienmitgliedschaft geführt, was als solche beantragt wird.

Es kommt noch ein weiteres Problem hinzu: Es wäre möglich, daß eine Mitgliedschaft als ganz normale Mitgliedschaft beginnt und dann durch den Beitritt eines weiteren Mitgliedes zur Familienmitgliedschaft erweitert wird. Im umgekehrten Fall könnte eine Familienmitgliedschaft durch Austritt eines Mitgliedes zur Einzelmitgliedschaft werden.

Nächstes Problem: Bei Postsendungen wird im Regelfall pro Familienmitgliedschaft nur ein Schreiben versandt. Hier wird jedoch erwartet, daß alle Mitglieder der Familienmitgliedschaft genannt werden – sofern das aus Platzgründen möglich ist. Solange eine Familienmitgliedschaft aus zwei Mitgliedern besteht, läßt sich das noch aus den Namen dieser beiden Mitglieder generieren. Bei steigender Zahl der beteiligten Personen stößt diese Methode jedoch schnell an ihre Grenzen.

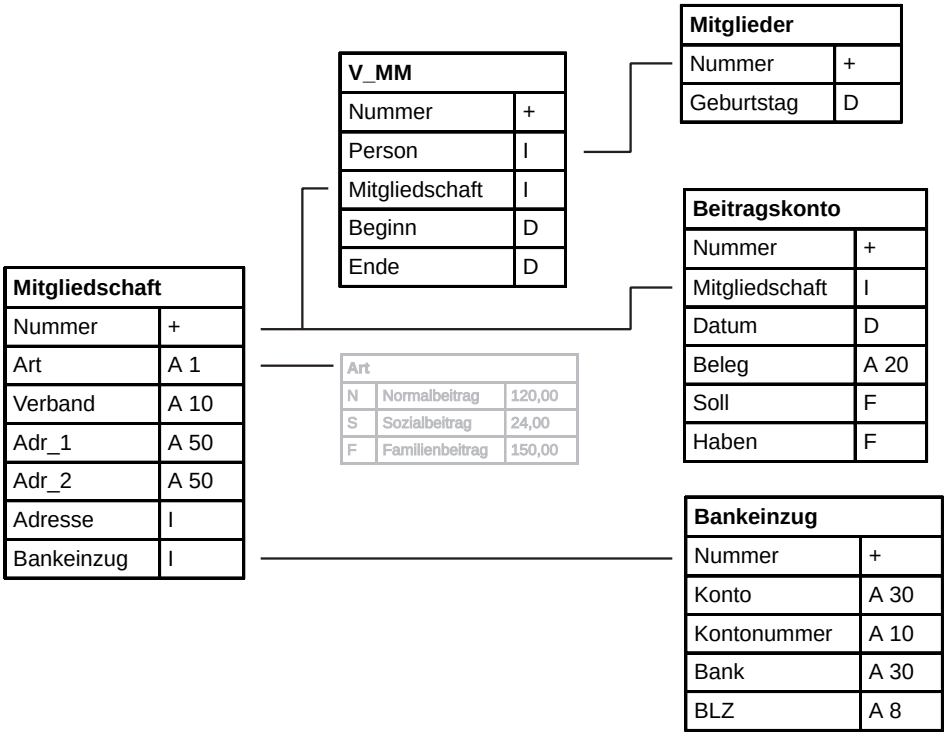


Bild 17.4: Die Mitgliedschaftstabellen

Sie werden inzwischen sicher einsehen, daß man die Mitgliedschaftsdaten nicht an die Personendaten koppeln kann. Deshalb soll hier ein neues Tabellensystem eingeführt werden – die Mitgliedschaft. Dieses Tabellensystem ist in Bild 17.4 dargestellt. Die schon oft verwendetete Verknüpfungstabelle – hier heißt Sie *V_MM* – stellt in unserem Fall den Bezug zwischen den Mitgliedern und den Mitgliedschaften her.

In der Tabelle *Mitgliedschaft* wird zunächst einmal gespeichert, um welche Art von Mitgliedschaft es sich handelt, daraus resultiert auch der Beitrag. Aus dem Feld *Verband* läßt sich ersehen, zu welchen Verbänden (Landesverband, Kreisverband, ...) die Mitgliedschaft gehört – dorthin wird auch der Beitragsanteil abgeführt. Die Struktur dieses Feldes werden wir uns noch ansehen. Es folgen zwei Felder, welche die ersten beiden Zeilen der Anschrift beinhalten. Bei Einzelmitgliedschaften werden Sie aus dem Namen und dem Feld *c_o* der Adresse und der Anrede generiert, bei Familienmitgliedschaften werden diese Felder manuell gesetzt. Den Rest der Adresse ermittelt man aus der Tabelle *Adressen*, auf die hier eine Referenz gebildet wird.

Liegt eine Einwilligung vor, die Mitgliedsbeiträge einzuziehen, dann wird im Feld *Bankeinzug* eine Referenz auf die gleichnamige Tabelle gebildet, andernfalls erhält dieses Feld den Wert NULL. Die Tabelle *Bankeinzug* speichert die Daten der Kontoverbindung.

Für jede Mitgliedschaft wird ein Beitragskonto geführt. Dort werden die fälligen Mitgliedsbeiträge in der Spalte *Soll* gebucht, eingehende Überweisungen oder eingezogene Beiträge werden unter *Haben* vermerkt. Mit einer STORED PROCEDURE kann jederzeit der Saldo des Kontos festgestellt werden.

```
CREATE PROCEDURE saldo_1
  (mitgliedschaft INTEGER,
  datum DATE)
RETURNS
  (saldo FLOAT)
AS
  DECLARE VARIABLE s_soll FLOAT;
  DECLARE VARIABLE s_haben FLOAT;
BEGIN
  SELECT SUM(soll)
    FROM beitragskonto
    WHERE (mitgliedschaft = :mitgliedschaft)
      AND (datum < :datum)
    INTO :s_soll;
  SELECT SUM(haben)
    FROM beitragskonto
    WHERE (mitgliedschaft = :mitgliedschaft)
      AND (datum < :datum)
    INTO :s_haben;
  IF (:s_soll IS NULL) THEN s_soll = 0;
  IF (:s_haben IS NULL) THEN s_haben = 0;
  saldo = s_haben - s_soll;
  SUSPEND;
END
```

In zwei Abfragen wird die Summe der *Soll*- und der *Haben*-Spalte von den Einträgen gebildet, die bei dem fraglichen Mitglied bis zum angegebenen Datum vorgelegen haben. Nun ist es möglich, daß es in der *Soll*- oder in der *Haben*-Spalte keine Einträge gibt und somit der entsprechenden Variablen NULL zugewiesen wird. Wird jedoch eine mathematische Operation mit NULL durchgeführt, dann lautet das Ergebnis ebenfalls NULL, und dies würde hier zu falschen Ergebnissen führen. Deshalb werden vor Bildung der Differenz beide Variablen auf ihren Wert hin geprüft, gegebenenfalls werden sie dann auf Null gesetzt.

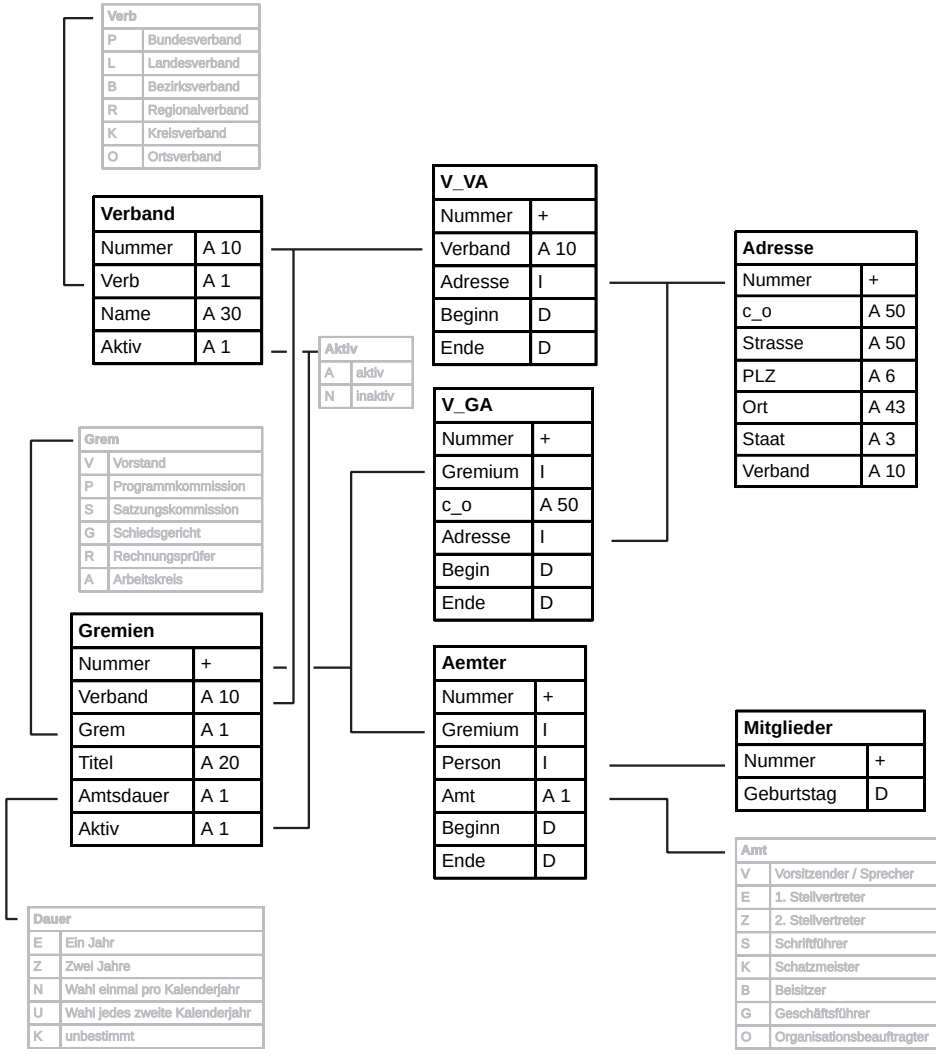


Bild 17.5: Verbände und Gremien

17.3 Verbände und Gremien

Als nächster Komplex sollen die Verbände und deren Gremien in der Datenbank erfaßt werden, dieser Teil der Datenmodells ist in Bild 17.5 dargestellt. In der Tabelle *Verband* sind alle Verbände der Partei aufgelistet. Als Primärschlüssel wird hier nicht eine durchlaufende Nummer, sondern eine zehnstellige Buchstaben-

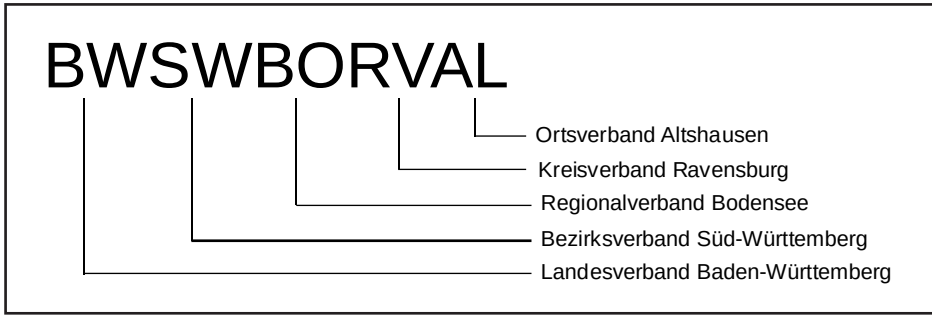


Bild 17.6: Zusammensetzung des Schlüssels der Tabelle VERBAND

kombination verwendet – das hat den Vorteil, daß der Verband so gut wie in Klartext aus dem Schlüssel zu erkennen ist.

Bild 17.6 zeigt, wie dieser Schlüssel aufgebaut ist. Die ersten beiden Buchstaben bezeichnen den Landesverband. Ist der Verband ein Landesverband, dann werden keine weiteren Zeichen angehängt. Der Schlüssel des Landesverbandes Baden-Württemberg lautet somit *BW* und der von Berlin *BE*. Die nächsten beiden Zeichen bezeichnen den Bezirksverband, das dritte Zeichenpaar den Regionalverband.

Das vierte Zeichenpaar bezeichnet den Kreisverband. Ist ein Kreisverband keinem Bezirks- und keinem Regionalverband untergeordnet, dann werden für diese beiden Buchstabenpaare Leerzeichen eingefügt. Der Schlüssel des Kreisverbandes Charlottenburg, der keinem Bezirks- oder Regionalverband untergeordnet ist, lautet somit *BE CH*. Mit dem fünften Zeichenpaar wird dann der Ortsverband bezeichnet.

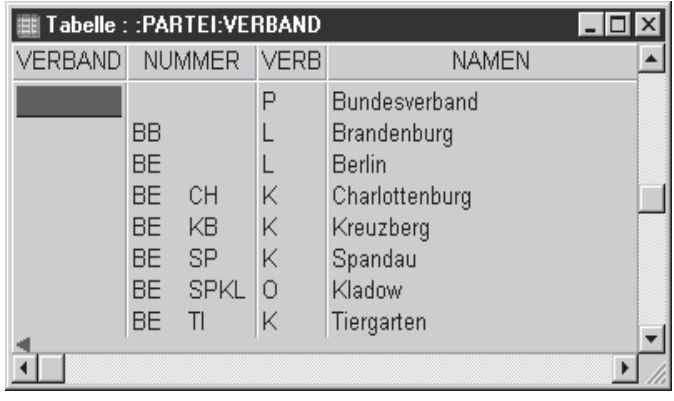


Tabelle : :PARTEI:VERBAND

VERBAND	NUMMER	VERB	NAMEN
		P	Bundesverband
	BB	L	Brandenburg
	BE	L	Berlin
	BE CH	K	Charlottenburg
	BE KB	K	Kreuzberg
	BE SP	K	Spandau
	BE SPKL	O	Kladow
	BE TI	K	Tiergarten

Bild 17.7: Ausschnitt aus der Tabelle VERBAND

Dieser Aufbau des Schlüssels hat den Vorteil, daß die Liste der Verbände automatisch in korrekter Struktur angezeigt wird, wenn diese Tabelle nach dem Primärschlüsel sortiert wird – Bild 17.7 zeigt einen Ausschnitt aus der Tabelle *Verband*, in dem dieses Prinzip erkennbar ist.

Nachteilig ist diese Vorgehensweise dann, wenn ein neuer Verband gegeründet wird, denn dann müssen nicht nur die Schlüssel aller untergeordneten Verbände geändert werden, sondern auch alle Datensätze, die einen dieser Schlüssel referenzieren. Die oben geschilderte Vorgehensweise ist deshalb nur dann sinnvoll, wenn die Struktur der Partei einigermaßen entwickelt ist, was wir hier einmal annehmen möchten.

Prinzipiell könnte man aus dem Primärschlüssel erkennen, um welche Art Verband (Landesverband, Kreisverband ...) es sich handelt. Es ist jedoch ein wenig aufwendig, diese Erkennung in eine SQL-Anweisung einzufügen; deshalb soll hier – auch wenn dadurch eine Redundanz entsteht – die Art des Verbandes im Feld *Verb* gespeichert werden, das über eine Referenz mit der gleichnamigen Tabelle verbunden ist; diese stellt die Information dann im Klartext bereit.

Wird ein Verband durch Beschluß aufgelöst oder findet eine satzungsgemäße Neuwahl des Vorstands nicht statt, dann existiert der Verband – formal gesehen – nicht mehr. Nun ist es nicht sinnvoll, hier gleich wieder die ganze Stuktur abzuändern, es reicht, wenn das Feld *Aktiv* auf *N* gesetzt wird. Jeder Verband hat eine Adresse, die sich allerdings im Laufe der Zeit auch ändern kann. Die Adresse wird – ebenso wie die Adressen der Mitglieder – in der Tabelle *Adresse* gespeichert, die Verbindung wird über die Tabelle *V_VA* hergestellt.

Die Gremien, die einem Verband zugeordnet sind, werden in der Tabelle *Gremien* gespeichert. Für die Art des Gremiums wird hier eine Nachschlagetabelle verwendet (*Grem*), genau wie für die Dauer, für die das jeweilige Gremium laut Satzung gewählt wird (*Dauer*). Über die Tabelle *Aemter* werden die Mitglieder dieser Gremien (Tabelle *Mitglieder*) mit den Gremien verbunden. Auch hier handelt es sich wieder um eine Zuordnungstabelle, in welcher Beginn und Ende der Amtsdauer vermerkt sind. Des weiteren haben sämtliche Gremien auch eine Anschrift, sie werden deshalb über die Tabelle *V_GA* mit der Tabelle *Adresse* verbunden. Da es sich hierbei meist um die Privatadresse eines Mitgliedes handeln wird, wird hier die Spalte *c_o* eingefügt, damit die Post auch darüber informiert wird, welcher der dreißig Mieter nun der *Arbeitskreis Innenpolitik* ist.

Aufmerksame Leser werden sich nun vielleicht fragen, warum in der Tabelle *Grem* keine Datensätze für die Delegierten zu Parteitagen und Hauptausschüssen existieren. Diese wurden hier nicht vergessen, vielmehr ist die Struktur der Tabelle *Aemter* nicht dafür geeignet, die Delegierten zu speichern – wir werden gleich sehen, warum.

17.4 Delegierte

Die Zahl der Delegierten hängt von der Mitgliederzahl ab und ist somit – wie diese – Schwankungen unterworfen. Da bei der Wahl der Delegierten folglich nicht exakt feststehen kann, wie viele Delegierte nun tatsächlich benötigt werden, wird rund die 2,5-fache Menge der aktuellen Delegiertenzahl gewählt. Nehmen wir an, dem Verband würden nach aktuellem Mitgliederstand zehn Delegierte zustehen, dann werden etwa 25 Delegierte gewählt. Bei der Wahl der Delegierten ergibt sich aus der Zahl der Stimmen, die auf die einzelnen Kandidaten entfallen, eine Reihenfolge.

Werden nun zum nächsten Parteitag zwölf Delegierte benötigt, dann sind die ersten zwölf Personen der Liste Delegierte und der Rest Ersatzdelegierte. Sind nun drei Delegierte verhindert, dann werden die Personen 13 bis 15 zu Delegierten für den jeweiligen Parteitag oder Hauptausschuß.

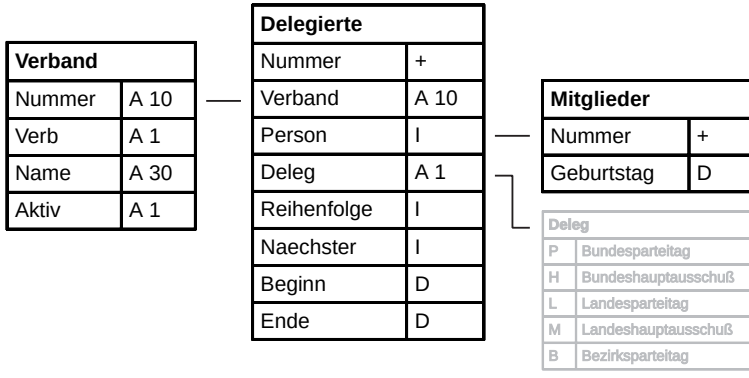


Bild 17.8: Speichern der Delegierten

Wie Bild 17.8 zeigt, gleicht die Struktur der Tabelle *Delegierte* weitgehend der Struktur der Tabelle *Aemter*, es wurden jedoch die Spalten *Reihenfolge* und *Naechster* eingefügt. In der Spalte Reihenfolge wird vermerkt, in welcher Reihenfolge die Delegierten zu laden sind. Nun kommt es – wie gesagt – manchmal vor, daß Delegierte verhindert sind und Ersatzdelegierte bemüht werden müssen. Meist wird das intern in den Landesverbänden geregelt, auch die Delegiertenunterlagen werden dann ohne Mitwirkung der Bundesgeschäftsstelle weitergereicht.

Manchmal kommt es aber auch vor, daß ein Delegierter der Bundesgeschäftsstelle schreibt, er sei am nächsten Parteitag verhindert, und man möge doch die Unterlagen bitte seinem Ersatzdelegierten zuschicken. Diesem Zweck dient die Spalte *Naechster*. Dort wird die Reihenfolge der Delegierten für die nächste Tagung festgehalten. Diese gleicht im Normalfall der Reihenfolge in der Spalte *Reihenfolge*.

Sind nun ein oder mehrere Delegierte verhindert, dann werden deren Zahlen auf den höchsten Wert gesetzt, und die Ersatzdelegierten rücken entsprechend eins nach vorn. Auf diese Weise läßt sich mit einer einfacher SQL-Anweisung die Liste (oder die Etiketten) der aktuellen Delegierten ausgeben.

```
SELECT * FROM ...
  WHERE delegierte.naechster < :anzahl
```

Des weiteren ist die Tabelle *Delegierte* mit der Tabelle *Verband* und nicht mit der Tabelle *Gremien* verbunden. In der Spalte *Deleg*, die mit der gleichnamigen Nachschlagetabelle verbunden ist, wird festgehalten, für welches Gremium das jeweilige Mitglied nun delegiert ist.

17.5 Postverteiler und Spenden

Von der Bundesgeschäftsstelle aus werden häufig wiederholt Informationen an denselben Personenkreis herausgegeben. Dies können Mitteilungen an die Presse oder Schulungsunterlagen an alle aktiven Mitglieder sein.

Solange diese Schreiben an eine fest definierte Personengruppe gehen, kann diese per SQL-Anweisung definiert werden. Sollen beispielsweise Richtlinien zur Ausstellung von Spendenquittungen an alle Kreis-, Regional- und Bezirksschatzmeister verschickt werden, dann würde man die folgende Abfrage definieren.

```
SELECT
    anrede,
    titel || vornamen || " " || adel || nachnamen,
    c_o,
    strasse,
    staatkuerzel || " " || plz || " " || ort
  FROM
    p_adressen_6("NOW", "E", "M") p, aemter a, gremien g, verband v
  WHERE (p.personennummer = a.person)
    AND (a.amt = "K")
    AND (a.beginn < "NOW")
    AND ((a.ende > "NOW") OR (a.ende IS NULL))
    AND (a.gremium = g.nummer)
    AND (g.verband = v.nummer)
    AND (v.verb IN ("B", "R", "K"))
```

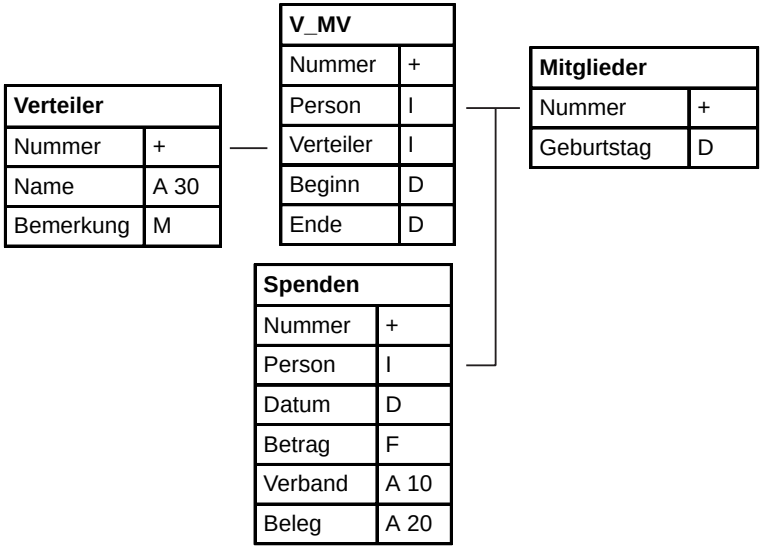


Bild 17.9: Die Tabellen für den Postverteiler und die Spenden

Wie Sie sehen, wird diese Abfrage durch die Einbindung der STORED PROCEDURE stark vereinfacht, denn die JOINS für die Zusammensetzung des Namens und der Adresse entfallen.

Nun gibt es allerdings auch Adressatengruppen, die nicht so einfach zu definieren sind. Es ist beispielsweise vorstellbar, daß die Partei einen Pressespiegel herausgibt, der an diejenigen Mitglieder versandt wird, welche ein Spende in Höhe der Herstellungs- und Versandkosten tätigen. Hier muß also eine Tabelle angelegt werden, welche die betreffenden Personen speichert. Selbstverständlich erstellt man keine Tabelle eigens für den Pressespiegel, sondern man bietet die Möglichkeit, beliebig viele Verteiler zu definieren, denen dann über die Tabelle *V_MV* die jeweiligen Personen zugeordnet werden.

Neben dem Namen der Verteiler lassen sich im Feld *Bemerkung* auch noch weitere Hinweise abspeichern, beispielsweise darüber, wie sich der Verteiler zusammensetzt.

Spenden

Ziemlich unspektakulär ist auch die Tabelle *Spenden*, in der alle an die Partei gerichteten Spenden erfaßt werden. Nun werden jedoch diese Spenden nicht zentral erfaßt, sondern über den jeweiligen Verband abgewickelt. Dieser selbst oder einer der übergeordneten Verbände erfaßt dann die Spenden per EDV. Damit eine Gesamtübersicht über alle Spenden erstellt werden kann, werden die Ein-

zeltabellen in die Tabelle *Spenden* kopiert – dazu kann die Komponente *TBatchMove* verwendet werden.

Bei einer Spende wird eine Spendenquittung ausgestellt, die sich eindeutig dem jeweiligen Datensatz zuordnen lassen muß. Bei der Ausstellung der Spendenquittung steht allerdings noch nicht fest, welchen Wert das Feld *Nummer* erhalten wird, wenn die Daten schließlich auf den Server kopiert werden. Jeder Verband vergibt deshalb einen Schlüsselwert, der sich aus der Verbandskennung, dem Jahr und einer fortlaufenden Nummer zusammensetzt; dieser Schlüsselwert wird dann in der Spalte *Beleg* gespeichert.

Für die geforderte Übersicht über alle Spenden wird dann lediglich noch ein Master-Detail-Report benötigt. Das Masterband listet die Adressen aller Spender sowie die Gesamtsumme deren Spenden auf, die Detailbänder geben Information über Datum und Höhe der Einzelspenden.

17.6 Weitere Tabellen

Im Prinzip ist das Datenmodell nun so weit komplett. Im folgenden sollen noch einige Ideen geäußert werden, wie sich dieses Datenmodell noch etwas ausbauen läßt.

Postleitzahlen und Verbände

Wenn eine Person einen Aufnahmeantrag oder eine Umzugsmeldung schickt, dann wird sie in der Regel nicht konkret darüber informiert sein, welche Untergliederungen für den Wohnsitz zuständig sind. Die Zuordnung wird deshalb die Partei, im Regelfall also die Bundesgeschäftsstelle vornehmen müssen.

Hier wäre es sinnvoll, wenn aus der Postleitzahl automatisch die zuständigen Verbände ermittelt werden könnten. Dafür würde man eine Tabelle erstellen, welche den Postleitzahlen die Verbandsnummern zuordnet. Es ist dabei zu beachten, daß in einzelnen Fällen bei ein und derselben Postleitzahl mehrere Gebietsverbände derselben Stufe in Frage kommen – die Postleitzahlen richten sich nur manchmal nach den politischen Grenzen.

Berufe und Hobbies

Gerade kleine Parteien sind stets knapp bei Kasse, viele Aufgaben müssen deshalb von den Mitgliedern in Eigenarbeit erledigt werden. Dabei wäre es sehr zweckmäßig, die einzelnen Mitglieder optimal nach ihren Fähigkeiten einzusetzen. Von daher mag der Wunsch an die Datenbank bestehen, daß sich die Berufe und besonderen Fähigkeiten (kann photographieren oder Karrikaturen zeichnen) speichern lassen.

Wenn ein solcher Wunsch an Sie herangetragen wird, dann sollten Sie stets darauf hinweisen, daß dabei eventuell Kollisionen mit dem Datenschutz zu befürchten sind. Es sind jedoch Ihre Auftraggeber letztlich dafür verantwortlich, was sie in den von Ihnen geschaffenen Strukturen speichern.

Um solche Informationen zu speichern, sollten Sie eine Tabelle der Berufe und eine dazugehörende Verknüpfungstabelle mit den Referenzen auf die Berufe und die Personen erstellen. Damit man die Berufe auch wiederfindet (Photograph, Pressephotograph, Photojournalist ...), sollten sie irgendwie gruppiert werden.

Stichwortverzeichnis

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Copyright

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website

(http://www.informit.de)

herunterladen