Delphi 6
Datenbankprogrammierung

Die Reihe Programmer´s Choice

Von Profis für Profis
Folgende Titel sind bereits erschienen:

Bjarne Stroustrup
Die C++-Programmiersprache
1072 Seiten, ISBN 3-8273-1660-X

Elmar Warken
Kylix - Delphi für Linux
1018 Seiten, ISBN 3-8273-1686-3

Don Box, Aaron Skonnard, John Lam
Essential XML
320 Seiten, ISBN 3-8273-1769-X

Elmar Warken
Delphi 6
1334 Seiten, ISBN 3-8273-1773-8

Bruno Schienmann
Kontinuierliches Anforderungsmanagement
392 Seiten, ISBN 3-8273-1787-8

Damian Conway
Objektorientiert Programmieren mit Perl
632 Seiten, ISBN 3-8273-1812-2

Ken Arnold, James Gosling, David Holmes
Die Programmiersprache Java
628 Seiten, ISBN 3-8273-1821-1

Kent Beck, Martin Fowler
Extreme Programming planen
152 Seiten, ISBN 3-8273-1832-7

Jens Hartwig
PostgreSQL - professionell und praxisnah
456 Seiten, ISBN 3-8273-1860-2

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Entwurfsmuster
480 Seiten, ISBN 3-8273-1862-9

Heinz-Gerd Raymans
MySQL im Einsatz
618 Seiten, ISBN 3-8273-1887-4

Dusan Petkovic, Markus Brüderl
Java in Datenbanksystemen
424 Seiten, ISBN 3-8273-1889-0

Joshua Bloch
Effektiv Java programmieren
250 Seiten, ISBN 3-8273-1933-1

PEARSON Education

ADDISON-WESLEY

Michael Ebner

Delphi 6 Datenbankprogrammierung

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Franzisco • Harlow, England
Don Mills, Ontario • Sydney
Mexico City • Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist
bei der Deutschen Bibliothek erhältlich

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.

Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig. Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis: Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt. Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

5 4 3 2 1

05 04 03 02

ISBN 3-8273-1785-1

© 2002 by Addison-Wesley Verlag, ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Str. 10-12, D-81829 München/ Germany
Alle Rechte vorbehalten
Einbandgestaltung: Christine Rechl, München
Titelbild Phacelia congesta, Phazelie.
© Karl Blossfeldt Archiv – Ann u. Jürgen Wilde, Zülpich / VG Bild-Kunst Bonn, 2002
Lektorat: Martin Asbach, masbach@pearson.de
Korrektorat: Anne Spranger, Freising
Herstellung: Monika Weiher, mweiher@pearson.de
Satz: Michael Ebner, Berlin
Druck und Verarbeitung: Bercker Graphischer Betrieb, Kevelaer
Printed in Germany

Inhaltsverzeichnis

Einführung 17

1 Die ersten Schritte 21

1.1 Inhalt einer Tabelle anzeigen 21
1.1.1 Erstellen einer Eingabemaske 26
1.1.2 Verwendung von TDBCtrlGrid 29
1.1.3 Drucken der Tabelle 31

1.2 Einen BDE-Alias einrichten 36

1.3 Abfragen 40
1.3.1 Anzeige der Kunden 42
1.3.2 Anzeige der Bestellungen 44
1.3.3 Die Posten der Bestellung 47
1.3.4 Gesamtumsatz einer Bestellung 48
1.3.5 Diagramm der Umsätze 50

2 Datenbanken 53

2.1 Was sind Datenbanken? 53
2.1.1 Historisches 54
2.1.2 Desktop- und Client-Server-Datenbanken 56
2.1.3 Multi-Tier-Systeme 58

2.2 Relationale Datenbanken 60
2.2.1 Begriffe 60
2.2.2 Keys (Schlüssel) 62
2.2.3 Weitere Elemente relationaler Datenbanksysteme 67
2.2.4 Normalisierung 69
2.2.5 Das erweiterte Entity-Relationship-Modell 73

2.3 Datenbanken und Delphi 79

3 TDataSet, TField und TDataSource 81

3.1 TDataSet 81

3.2 TField 99
3.2.1 Referenz von TField 103
3.2.2 Die TField-Nachkommen 113

3.3 TDataSource 116

4 Die BDE 119

4.1 Die BDE-Verwaltung 121

4.2 Die Datenbankoberfläche 123
4.2.1 Öffnen von Tabellen 124
4.2.2 Anlegen von Tabellen mit der DBO 125
4.2.3 Stapel-Operationen 133
4.2.4 SQL 133

4.3 Der Datenbank-Explorer 134
4.3.1 Eine Datenbank untersuchen 135
4.3.2 Tabellen 137
4.3.3 Das Dictionary 139
4.3.4 Sonstiges 141

4.4 Die Komponente TSession 142
4.4.1 Passwort 142
4.4.2 Alias- und Tabellennamen 143
4.4.3 Verzeichnisse 145
4.4.4 Adressensuche im Hintergrund 146

4.5 TDatabase 151
4.5.1 Transaktionen 151
4.5.2 Execute 155
4.5.3 Referenz TDatabase 163

5 TTable 169

5.1 Erstellen von Testdaten 169
5.1.1 Erstellen der Tabellen 169

5.2 Master-Detail-Verknüpfungen ... 175
5.2.1 Anzeige der Bestellungen ... 176
5.2.2 Anzeige der Posten ... 178
5.2.3 Messung der Ausführungsgeschwindigkeit ... 178
5.3 Tabellen zur Laufzeit erstellen ... 180
5.4 Suchen und Filtern ... 183
5.4.1 Einen Bereich setzen ... 183
5.4.2 Datensätze suchen ... 185
5.4.3 Datensätze filtern ... 187
5.5 Referenz TDBDataSet ... 190
5.6 TTable ... 196
5.7 Die Komponente TBatchMove ... 203
5.7.1 Ein Datentransferprogramm ... 203
5.7.2 Referenz TBatchMove ... 206

6 TQuery ... 209

6.1 LocalSQL ... 209
6.1.1 SELECT ... 210
6.1.2 JOINS ... 211
6.1.3 Filtern mit WHERE ... 213
6.1.4 Sortieren mit ORDER BY ... 216
6.1.5 Aggregatfunktionen ... 217
6.1.6 Gruppieren mit GROUP BY und HAVING ... 217
6.1.7 Operatoren und Funktionen ... 218
6.1.8 Ergebnismengen verbinden mit UNION ... 220
6.1.9 Unterabfragen ... 220
6.1.10 Reihenfolge der Klauseln ... 221
6.1.11 Datenänderung ... 221
6.1.12 Definition von Metadaten ... 223
6.2 Arbeiten mit TQuery ... 224
6.2.1 Parameter ... 225
6.2.2 Datenmengen filtern ... 227
6.2.3 Heterogene Joins ... 230
6.2.4 Die Eigenschaft Filter bei SQL-Servern ... 230

6.3 Referenz TQuery ... 231
6.4 Die Komponente TUpdateSQL ... 235
6.4.1 0/1/n : 0/1/m-Master-Detail-Beziehungen ... 238
6.4.2 Referenz TUpdateObject ... 242
6.5 Adressensuche ... 243
6.5.1 Generierung von Such-Strings ... 247
6.5.2 Entfernen doppelter Adressen ... 254

7 MyBase ... 267

7.1 Eine Tabelle erstellen ... 267
7.1.1 Eine Tabelle zur Entwurfszeit erstellen ... 268
7.1.2 Eine Tabelle zur Laufzeit erstellen ... 268
7.2 Daten speichern ... 269
7.3 Sortieren von Datenmengen ... 276
7.4 Suchen und Filtern ... 277
7.5 Master-Detail-Verknüpfungen ... 280
7.6 Gewartete Aggregate ... 280
7.7 Referenz TClientDataSet ... 281
7.8 TBClientDataSet ... 291
7.9 Die TClientDataSet-Datenbankoberfläche ... 293
7.9.1 Anlegen neuer Tabellen ... 293
7.9.2 Tabellen ansehen ... 300
7.9.3 Importieren von BDE-Daten ... 304

8 Datensteuerung ... 309

8.1 TDBGrid ... 309
8.1.1 Referenz TDBGrid ... 314
8.1.2 Referenz TColumn ... 318
8.1.3 TBDBGrid ... 319
8.2 TDBNavigator ... 322
8.3 TDBText ... 323
8.4 TDBEdit ... 324

8.5 TDBMemo 328
8.6 TDBImage 330
8.7 TDBListBox 331
8.8 TDBComboBox 334
8.9 TDBCheckBox 336
8.10 TDBRadioGroup 337
8.11 TDBLookupListBox 338
8.12 TDBLookupComboBox 339
8.13 TDBRichEdit 340
8.14 TDBCtrlGrid 346

9 Datenanalyse 349

9.1 TChart und TDBChart 349
9.1.1 Das Beispielprogramm 349
9.1.2 Die aktuelle Wahl 355
9.1.3 Gewinn- und Verlustrechnung 357
9.2 Referenz TChart 361
9.2.1 Die Achsen 362
9.2.2 Legende 369
9.2.3 Darstellung 372
9.3 TChartSeries 376
9.3.1 Grafische Darstellung 377
9.3.2 Anzeigewerte 380
9.3.3 Werteberechnungen 384
9.4 Die Reihenkomponenten 387
9.4.1 TLineSeries 387
9.4.2 TBarSeries und THorizBarSeries 388
9.4.3 TAreaSeries 390
9.4.4 TPointSeries 391
9.4.5 TPieSeries 392
9.4.6 TFastLineSeries 393
9.4.7 Die Komponente TGanttSeries 394
9.4.8 TBubbleSeries 396

9.4.9 TArrowSeries ... 397
9.4.10 TChartShape ... 399
9.5 TDBChart ... 400
9.6 TDecisionGraph ... 401
9.7 TQRChart ... 402
9.8 Die Komponenten zur Datenanalyse ... 403
9.8.1 Ein Beispielprojekt ... 403
9.8.2 TDecisionQuery ... 405
9.8.3 TDecisionCube ... 406
9.8.4 TDecisionSource ... 408
9.8.5 TDecisionPivot ... 411
9.8.6 TDecisionGrid ... 413
9.8.7 TDecisionGraph ... 417
9.8.8 Datenanalyse im Thread ... 417

10 Die QuickReport-Komponenten ... 421

10.1 Eine Adressenliste ... 423
10.2 Gruppieren der Adressen ... 425
10.3 Ausdruck von Etiketten ... 427
10.3.1 Eigenes Vorschauformular ... 428
10.4 Reports mit Bildern und Memos ... 432
10.5 Master-Detail-Reports ... 433
10.6 Referenz der QuickReport-Komponenten ... 435
10.6.1 Die Komponente TQuickRep ... 436
10.6.2 Die Komponente TQRBand ... 439
10.6.3 Die Komponente TQRSubDetail ... 441
10.6.4 Die Komponente TQRGroup ... 443
10.6.5 TQRChildBand ... 443
10.6.6 TStringsBand ... 444
10.6.7 TQRLabel, TQRMemo und TQRDBText ... 445
10.6.8 Die Komponente TQRExpr ... 446
10.6.9 Die Komponente TQRSysData ... 447
10.6.10 TQRRichText und TQRDBRichText ... 448

10.6.11 TQRImage und TQRDBImage 448
10.6.12 Die Komponente TQRShape 449
10.6.13 Die Filterkomponenten 450
10.6.14 Die Komponente TQRCompositeReport 450

11 InterBase 6 453

11.1 Starten des Servers 453
11.2 IBConsole 455
11.2.1 Registrieren einer Server-Verbindung 456
11.2.2 Einsehen der Metadaten 461
11.2.3 Backups 465
11.3 Interactive SQL 467
11.4 Die Multi-Generationen-Architektur 469
11.4.1 Transaktionen beim Lesen 469
11.4.2 Multi-Generationen-Architektur 470

12 InterBase SQL 473

12.1 SELECT 473
12.1.1 WHERE 473
12.1.2 ORDER BY 474
12.2 DOMAINS 477
12.2.1 Datentypen 478
12.2.2 DEFAULT-Werte 481
12.2.3 Eingabe erzwingen 483
12.2.4 Gültigkeitsprüfungen 483
12.2.5 Domains ändern 484
12.2.6 Domains löschen 485
12.2.7 Generatoren 485
12.3 Tabellen 488
12.3.1 CREATE TABLE 488
12.3.2 ALTER TABLE 498
12.3.3 Drop Table 503
12.3.4 Indizes 503

12.4 VIEWS ... 504
12.4.1 Eine VIEW erstellen ... 505
12.4.2 Zugriffsmodus ... 506
12.4.3 Eine VIEW löschen ... 509
12.5 STORED PROCEDURES ... 509
12.5.1 Ein Beispiel ... 510
12.5.2 Übersicht über die Prozedur-Sprache ... 513
12.5.3 Einsatz von STORED PROCEDURES ... 518
12.5.4 Ändern und Löschen von STORED PROCEDURES ... 519
12.6 TRIGGER ... 519
12.6.1 Ein Beispiel ... 519
12.6.2 Vergleich alter und neuer Spaltenwerte ... 520
12.6.3 TRIGGER ändern und löschen ... 521
12.7 Zugriffsberechtigungen ... 523
12.8 Sonstiges ... 525

13 Client-Server mit der BDE ... 527

13.1 TTable und TQuery ... 527
13.1.1 Zugriff auf eine Tabelle ... 527
13.2 Transaktionen ... 534
13.2.1 Abschottung von Transaktionen ... 535
13.2.2 Automatische Transaktionskontrolle ... 538
13.3 Zugriff auf STORED PROCEDURES ... 541
13.3.1 Zugriff mit TQuery ... 541
13.3.2 Zugriff mit TStoredProc ... 543
13.4 Selbstinkrementierende Felder ... 543
13.5 Optimierung ... 547
13.5.1 Erstellen von Testdaten ... 547
13.5.2 Datensätze filtern ... 553
13.5.3 Indizes verwenden ... 555
13.5.4 Vermeidung von LIKE-Operatoren ... 557
13.5.5 Datensätze zählen ... 558

14 Die Interbase-Komponenten 563

14.1 Ein erstes Beispiel 563

14.2 Die Komponente TIBDataSet 565

14.3 Referenz der InterBase-Komponenten 569

14.3.1 TIBDatabase 570

14.3.2 TIBTransaction 572

14.3.3 TIBCustomDataSet 574

14.3.4 TIBTable 576

14.3.5 TIBQuery 579

14.3.6 TIBStoredProc 580

14.3.7 TIBUpdateSQL 581

14.3.8 TIBDataSet 582

14.3.9 TIBSQL 583

14.3.10 TIBDatabaseInfo 584

14.3.11 TIBMonitor 585

14.3.12 TIBEvents 585

15 dbExpress 587

15.1 Erste Schritte mit dbExpress 588

15.1.1 Zugriff auf eine Tabelle 588

15.1.2 Belauschen der Datenbankverbindung 591

15.1.3 TSQLClientDataSet 596

15.1.4 Stored Procedures 600

15.2 Referenz dbExpress 602

15.2.1 TSQLConnection 602

15.2.2 TCustomSQLDataSet 608

15.2.3 TSQLDataSet 614

15.2.4 TSQLQuery 616

15.2.5 TSQLTable 616

15.2.6 TSQLStoredProc 617

15.2.7 TSQLClientDataSet 617

15.2.8 TSQLMonitor 618

16 Multi-Tier-Anwendungen 619

16.1 Multi-Tier mit DCOM 619
16.1.1 Erstellung des Middle-Tiers 619
16.1.2 Erstellung des Clients 622
16.1.3 Referenz TCustomRemoteServer 626
16.1.4 Referenz TProvider 630

16.2 Corba 631
16.2.1 Der Broker 632
16.2.2 Einen Corba-Anwendungs-Server erstellen 632
16.2.3 Einen Corba-Client erstellen 633

17 Die To-Do-Liste 635

17.1 Was ist die To-Do-Liste? 635

17.2 Die ersten Schritte mit der To-Do-Liste 637
17.2.1 Erstellen der Datenbank 637
17.2.2 Anmelden 638
17.2.3 Anlegen von Benutzern und Funktionen 639
17.2.4 Verwaltung der Projekte 643
17.2.5 Eine Abfrage generieren 647
17.2.6 Einen Eintrag erstellen 649
17.2.7 Eintrag ansehen 651
17.2.8 Drucken 654
17.2.9 Abfragen mit dem Assistenten erstellen 655

17.3 Das SQL-Script 657
17.3.1 Die DOMAINS 657
17.3.2 Die Tabelle t_benutzer 657
17.3.3 Die Tabelle t_funktion 659
17.3.4 Die Tabelle t_benutzer_funktion 659
17.3.5 Die Tabelle t_projekt 660
17.3.6 Die Tabelle t_recht 661
17.3.7 Die Tabelle t_prioritaet 662
17.3.8 Die Tabelle t_eintrag 663
17.3.9 Die Tabelle t_status 663

17.3.10 Die Tabelle tn_status ... 665
17.3.11 Die Tabelle t_datei ... 665
17.3.12 Die Tabelle t_anzeige ... 666
17.3.13 Die STORED PROCEDURE p_eintragaktuell ... 667
17.3.14 Die STORED PROCEDURE p_eintraegeoffen ... 669

18 To-Do-Liste mit der BDE ... 673

18.1 Wie man größere Projekte realisiert ... 673
18.1.1 Das Datenmodul ... 676

18.2 Die Verwaltung ... 680
18.2.1 Die Benutzerverwaltung ... 682
18.2.2 Verwaltung der Projekte ... 694

18.3 Einträge erstellen und bearbeiten ... 720
18.3.1 Einen neuen Eintrag erstellen ... 721
18.3.2 Einen Eintrag ansehen und bearbeiten ... 729
18.3.3 Einträge drucken ... 745

18.4 Abfragen erstellen ... 752
18.4.1 Die Unit u_anzeigen ... 753
18.4.2 Abfragen erstellen ... 757

18.5 Das Hauptprojekt ... 769
18.5.1 Das Hauptformular ... 769
18.5.2 Liste drucken ... 774

19 To-Do-Liste mit IBX ... 777

19.1 Die Verwaltung ... 777
19.1.1 Das Datenmodul ... 777
19.1.2 Die Benutzerverwaltung ... 778
19.1.3 Die Verwaltung der Projekte ... 783
19.1.4 Der Passwortdialog ... 786

19.2 Die Einträge ... 786
19.2.1 Verwendung von TIBSQL ... 789

19.3 Die Anzeige ... 790

19.4 Das Gesamtprojekt ... 792

20 To-Do-Liste mit dbExpress 795

20.1 Das Datenmodul 795

20.2 Die Verwaltung 797

20.2.1 Die Benutzerverwaltung 797

20.2.2 Der Passwortdialog 802

20.2.3 Die Projektverwaltung 803

20.3 Die Einträge 811

20.4 Die Anzeige 812

20.5 Das Gesamtprojekt 817

20.6 Fazit 818

Index 821

21 ADO und Access (auf der beiliegenden CD)

21.1 Access-BDE-Treiber 837

21.2 Einrichten einer ODBC-Verbindung 838

21.3 Einrichten einer ADO-Verbindung 839

21.4 Aufbau einer Master-Detail-Verknüpfung 842

21.5 Referenz der ADO-Komponenten 845

21.5.1 TADOConnection 845

21.5.2 TADOCommand 846

21.5.3 TCustomADODataSet 847

21.5.4 TADODataSet 851

21.5.5 TADOTable 852

21.5.6 TADOQuery 853

21.5.7 TADOStoredProc 854

Einführung

Zu Zeiten von Delphi 1 war alles noch schön übersichtlich: Für den Datenbankzugriff gab es die BDE und noch ein paar Lösungen von Drittherstellern. Unter Delphi 6 haben Sie deutlich mehr Möglichkeiten:

- Nach wie vor gibt es die BDE und die Möglichkeit, darüber auch auf ODBC-Datenbanken zuzugreifen.
- Mit *TClientDataSet* können Sie schnelle Einbenutzersysteme erstellen.
- Für den Zugriff auf Client-Server-Systeme gibt es die plattformübergreifende Lösung *dbExpress*.
- Für den Zugriff auf InterBase können Sie IBX verwenden.
- Und natürlich gibt es auch Komponenten, die auf ADO basieren.

Als Entwickler haben Sie nun die Qual der Wahl. In diesem Buch soll versucht werden, ein wenig Übersicht in diese Vielfalt zu bringen.

Achtung

Im Bestreben um eine möglichst knappe Darstellung wurden zwei Sachverhalte hin und wieder nicht ganz korrekt dargestellt.

- Voreingestellte Werte werden hinter dem Schlüsselwort *default* angegeben.

  ```
  property Alignment: (paLeft, paRight, paCenter) default paLeft;
  ```

  Nicht in allen Fällen handelt es sich dabei tatsächlich um einen *default*-Wert. Trotzdem wurde – wegen der Prägnanz der Darstellung – diese Schreibweise verwendet.

- Bei der Darstellung von Ereignissen wurde die Parameterliste an den Ereignisbezeichner angehängt. Statt also ausführlich – und ganz korrekt – zu schreiben

  ```
  type TNotifyEvent = procedure (Sender: TObject) of object;
  property OnChange: TNotifyEvent;
  ```

  heißt es in diesem Buch kurz und prägnant:

  ```
  property OnChange(Sender: TObject);
  ```

Ich nehme an, dass Sie auch so daraus schlau werden.

Überblick über den Inhalt

- Im ersten Kapitel wollen wir über ein paar kleine Projekte den Einstieg in die Thematik finden – dieses Kapitel richtet sich vor allem an Einsteiger. In Kapitel 2 wollen wir uns ein wenig der grauen Theorie annehmen, und in Kapitel 3 geht es um die Komponenten *TDataSet*, *TField* und *TDataSource*, die das Rückgrat einer jeden Datenbankanwendung bilden.
- In den Kapiteln 4 bis 6 geht es um Datenbankprogrammierung mit der BDE. Die Grundlagen nebst den Komponenten *TDatabase* und *TSession* schauen wir uns in Kapitel 4 an. In Kapitel 5 beschäftigen wir uns mit *TTable* und *TBatchMove*, während *TQuery* und *TUpdateSQL* das Thema von Kapitel 6 bilden.
- Einplatz-Datenbanksysteme mit *TClientDataSet* (»MyBase«) besprechen wir in Kapitel 7.
- In der Kapiteln 8 bis 10 geht es um die Anzeige der Daten. Kapitel 8 behandelt die Datensteuerungskomponenten, Kapitel 9 die Komponenten zur Datenanalyse einschließlich *TChart* und Kapitel 10 die *QuickReport*-Komponenten.
- In Kapitel 11 wollen wir uns die Bedienung von *InterBase* ansehen, in Kapitel 12 geht es um *InterBase SQL*.
- In den Kapiteln 13 bis 15 beschäftigen wir uns mit den drei Wegen, auf Client-Server-Systeme, insbesondere auf InterBase, zuzugreifen. In Kapitel 13 nehmen wir dazu die BDE, in Kapitel 14 die InterBase-Komponenten und in Kapitel 15 *dbExpress*.
- Das Thema Multi-Tier-Datenbankanwendungen werden wir ganz kurz in Kapitel 16 streifen.
- In den Kapiteln 17 bis 20 werden wir uns dem Beispielprogramm To-Do-Liste widmen. Kapitel 17 führt in die Thematik ein und erläutert das Datenbankdesign. In Kapitel 18 werden wir das Programm mit der BDE realisieren, in Kapitel 19 mit IBX und in Kapitel 20 mit dbExpress.
- Zum Schluss – also in Kapitel 21 – wollen wir noch einen kurzen Blick auf ADO werfen. Aus Platzgründen musste dieses Kapitel auf die CD »verbannt« werden.

CD

Auf der Buch-CD finden Sie die Beispielprogramme sowie eine Trial-Version von Delphi 6. Des Weiteren finden Sie ein Bonus-Kapitel über ADO und Access im pdf-Format.

Kopieren Sie die Dateien vor der Verwendung auf Ihre Festplatte und löschen Sie das *Schreibgeschützt*-Bit.

Andere Bücher

Es soll nicht versäumt werden, hier einige andere Bücher zu empfehlen:

- Für Einsteiger *Delphi 6 lernen* und *Goto Delphi 6* (beide von Lang/Bohne), erschienen bei Addison-Wesley;
- Als Grundlagenbuch *Delphi 6* (Elmar Warken), erschienen bei Addison-Wesley;
- Als Nachschlagewerk für verschiedene Themen *Delphi 6 Kochbuch* (Doberenz/Kowalski), erschienen bei Hanser, und *Delphi 6 im Team* (Redaktion Toolbox), erschienen bei C&L;
- Für die Datenbankprogrammierung *Client/Server* (Andreas Kosch), erschienen bei Der Entwickler;
- Und wenn Sie sich COM antun wollen/müssen *COM/DCOM* (Andreas Kosch) erschienen bei Der Entwickler.

Fehlerberichtigungen und Ergänzungen

Erfahrungsgemäß schleichen sich trotz penibler Korrektur immer einige Fehler ein. Deswegen werden im Internet unter *www.tabu-datentechnik.de* Fehlerberichtigungen und Ergänzungen veröffentlicht.

Schulungen

Der Autor hält auch Delphi-Schulungen. Näheres erfahren Sie bei *TABU-Datentechnik* (www.tabu-datentechnik.de).

Kontakt zum Autor

Anregungen und Kritik, Hinweise auf (leider) übersehene Fehler bitte an unten angegebene E-Mail-Adresse. Beachten Sie bitte, dass der Autor keine Delphi-Hotline betreibt.

Berlin, Februar 2002

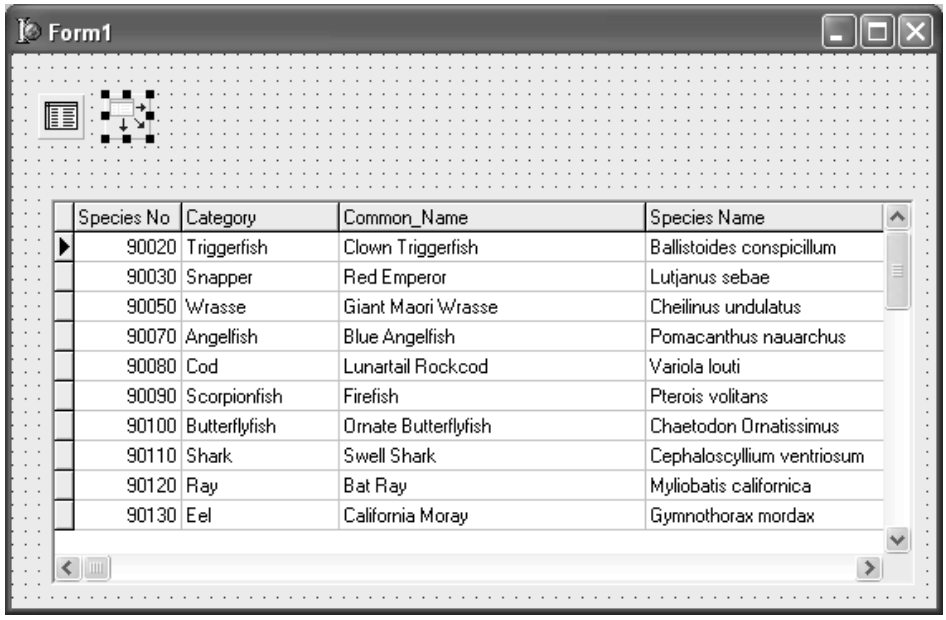
Michael Ebner
info@tabu-datentechnik.de
www.tabu-datentechnik.de

1 Die ersten Schritte

Für den Fall, dass Sie unter Delphi noch nie mit Datenbanken gearbeitet haben, erhalten Sie hier eine kurze Einführung. Der Umgang mit der Delphi-IDE sollte Ihnen jedoch geläufig sein – gegebenenfalls sollten Sie dieses Buch zunächst beiseite legen und ein Grundlagenbuch zu Delphi lesen.

1.1 Inhalt einer Tabelle anzeigen

Zunächst soll ein ganz einfaches Projekt erstellt werden, das lediglich den Inhalt einer Tabelle anzeigt.



Legen Sie ein neues Projekt an und legen Sie die benötigten Komponenten auf das Hauptformular und ändern Sie die Eigenschaften entsprechend der folgenden Liste:

- Von der Palettenseite *BDE* (in den Versionen bis einschließlich Delphi 5 von der Palettenseite *Datenzugriff*) die Komponente *TTable*:

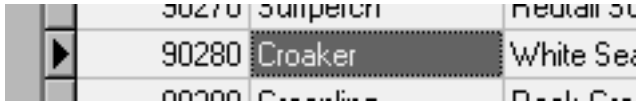
- Die Eigenschaft *DatabaseName* setzen Sie auf *DBDEMOS*.
- *TableName* erhält den Wert *biolife.db*.
- Und zuletzt wird die Eigenschaft *Active* auf *true* gesetzt.

- Von der Palettenseite *Datenzugriff* wird die Komponente *TDataSource* eingefügt:
  - Die Eigenschaft *DataSet* setzen Sie auf *Table1*.
- Zuletzt noch *TDBGrid* von der Palettenseite *Datensteuerung*:
  - Hier setzen Sie *DataSource* auf *DataSource1*.

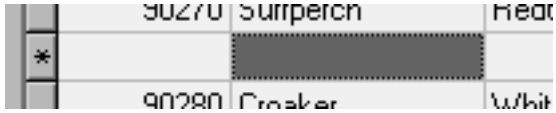
Wie Sie sehen, werden schon zur Entwurfszeit die Daten angezeigt, sie können auch jetzt schon im Daten-Grid scrollen – das Ändern der Daten funktioniert jedoch noch nicht.

Starten Sie nun das Projekt mit F9.

- Mit den Cursor-Tasten können Sie in eine andere Zelle wechseln, Sie können jedoch die gewünschte Zelle auch direkt mit der Maus anklicken. Sie befinden sich im so genannten Browse-Modus, erkennbar am schwarzen Dreieck vor dem aktuellen Datensatz.



- Sobald Sie in der untersten Zeile sind und die Cursor-Taste DOWN betätigen oder halten, wird ein neuer Datensatz eingefügt. Dies ist am Stern vor der betreffenden Zeile zu erkennen.

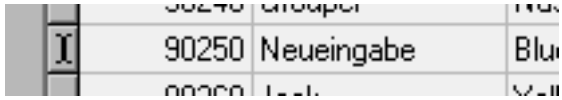


Alternativ können Sie auch mit der INSERT-Taste einen neuen Datensatz einfügen. Füllen Sie nun einige oder alle Felder des neuen Datensatzes und wechseln dann auf einen anderen Datensatz – mit der Tastatur oder mit der Maus. Damit sind die Eingaben übernommen und der aktuelle Datensatz ist wieder mit einem Dreieck gekennzeichnet.

Sollten Sie es sich anders überlegt haben und die Neueingabe verwerfen wollen, dann verwenden Sie die Taste ESC.

- Möchten Sie den Inhalt eines Feldes ändern, so markieren Sie dieses und beginnen mit der Änderung. Die Datenmenge wird dadurch automatisch in den Edit-Modus versetzt, den Sie an der Eingabemarke erkennen. Um die Ände-

rungen zu übernehmen, wechseln Sie wieder auf einen anderen Datensatz, um sie zu verwerfen, verwenden Sie die Taste Esc.

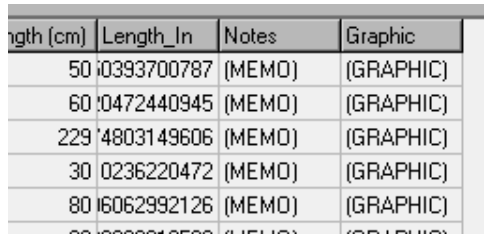


- Möchten Sie einen Datensatz löschen, dann verwenden Sie die Tastenkombination Ctrl+Del (auf deutschen Tastaturen Strg+Entf).

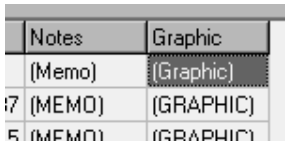
Wenn Sie die Sicherheitsabfrage mit Ok bestätigen, wird der Datensatz gelöscht.

Probleme bei der Darstellung

Aufgrund von zwei kleinen Problemen ist die Darstellung im Moment noch nicht befriedigend:



- Die Angabe der Länge in *Inch* erfolgt auf 15 Stellen genau, also mit sämtlichen 13 oder 14 Nachkommastellen – bei der Längenangabe von Fischen ist diese Genauigkeit sicherlich übertrieben.
- Die Komponente *TDBGrid* kann weder mehrzeiligen Text noch Bilder richtig anzeigen.



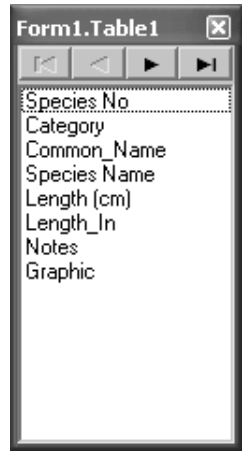
Anhand der Ersatzbeschriftung kann man erkennen, ob das entsprechende Feld gefüllt ist oder nicht: Steht dort *(Graphic)*, dann liegt kein Bild vor, mit *(GRAPHIC)* wird angezeigt, dass ein Bild vorhanden ist.

Wir werden später eine eigene Komponente programmieren, welche die Anzeige von Bildern erlaubt, bis dahin sollen die Spalten jedoch einfach ausgeblendet werden.

Formatieren der Zahlendarstellung

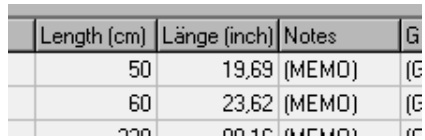
Um die Zahlendarstellung zu optimieren, müssen wir zunächst so genannte *persistente TField-Instanzen* anlegen. Was das genau ist, werden wir später klären, hier soll nur interessieren, wie man dabei vorgeht:

- Führen Sie einen Doppelklick auf *Table1* aus. Es wird das Fenster des Felder-Editors geöffnet.
- Wählen Sie aus dem Kontextmenü (rechte Maustaste) den Eintrag ALLE FELDER HINZUFÜGEN. Sie erhalten nun eine Liste aller in der Datenbank vorhandenen Spalten.



- Markieren Sie nun die Spalte *Length_In*, so dass deren Eigenschaften im Objektinspektor angezeigt werden.
- Setzen Sie nun die Eigenschaft *DisplayFormat* auf *0.00* – zwei Nachkommastellen sind hier mehr als genug. (Bitte beachten: Da in den USA der Punkt das Dezimaltrennzeichen ist, muss auch hier ein Punkt verwendet werden. Was dann tatsächlich angezeigt wird, entscheidet die Windows-Systemeinstellung.)
- Wenn wir schon gerade dabei sind, dann ändern wir auch noch *DisplayLabel* auf *Länge (inch)*.

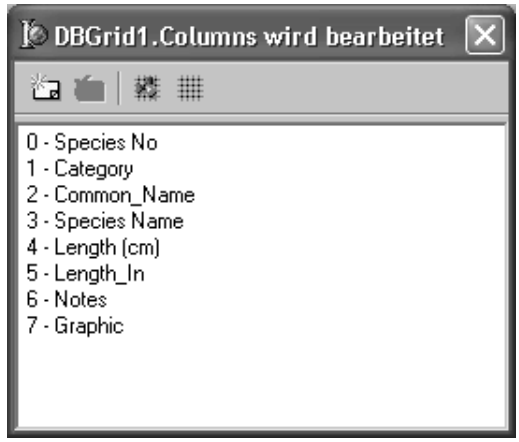
Die Werte dieser Spalte werden nun mit zwei Nachkommastellen ausgegeben, außerdem wurde die Spaltenüberschrift geändert.



Ausblenden von Spalten

Sie könnten nun die Spalten *Notes* und *Graphic* aus der Liste im Felder-Editor löschen – damit wären Sie jedoch auch dann nicht vorhanden, wenn Sie mit anderen Dialogkomponenten die Daten anzeigen wollen. Sinnvoller ist es hier, Spalten-Instanzen anzulegen und die beiden unerwünschten dann zu löschen. Dazu gehen Sie wie folgt vor:

- Führen Sie einen Doppelklick auf *DBGrid1* durch, dadurch wird der Spalten-Editor geöffnet.
- Klicken Sie auf das dritte Icon von links (*Alle Felder hinzufügen*). Nun erhalten Sie eine Liste mit allen angelegten Spalten.

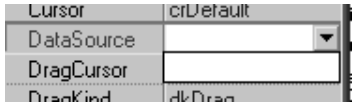


- Markieren Sie *Notes* und *Graphic* und löschen diese aus der Liste. Daraufhin werden diese Spalten auch nicht mehr angezeigt.
- Dann wählen Sie noch *Species No* und geben in *Title.Caption* den Wert *Nummer* ein. Damit ist die Spaltenüberschrift entsprechend abgeändert. Den Unterschied zum Ändern von *DisplayLabel* der *TField*-Instanz werden wir im nächsten Abschnitt sehen.

1.1.1 Erstellen einer Eingabemaske

Wir wollen nun eine Eingabemaske für die Tabelle erstellen. Erstellen Sie mit DATEI|NEU|FORMULAR ein neues Formular und fügen folgende Komponenten hinzu, die Sie alle auf der Palettenseite *Datensteuerung* finden:

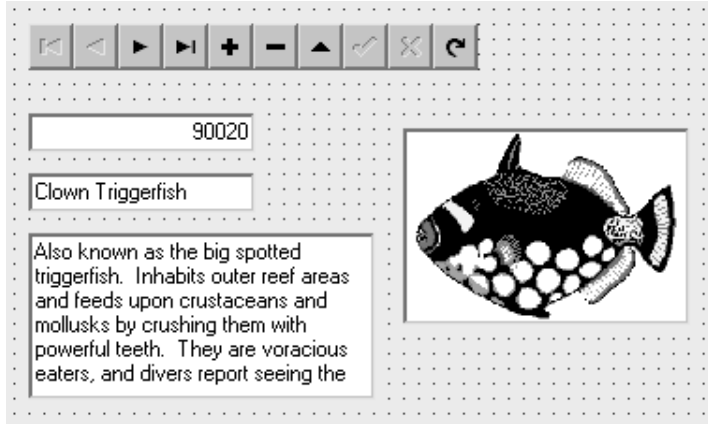
- Zunächst TDBNavigator, um in der Datenmenge navigieren zu können, also einen anderen Datensatz wählen, Datensätze einfügen, löschen oder ändern zu können.
  - Setzen Sie die Eigenschaft *DataSource* auf *Form1.DataSource1*. Dies wird zunächst nicht gelingen, weil *unit1* noch nicht eingebunden ist.



  Gehen Sie in den *implementation*-Teil von *unit2* und ergänzen Sie:

```
uses unit1;
```

  Anschließend sollten Sie im Objektinspektor *Form1.DataSource1* auswählen können.



- Nun fügen Sie zwei *TDBEdit*-Instanzen ein.
  - Bei beiden wird *DataSource* auf *Form1.DataSource1* gesetzt.
  - Die Eigenschaft *DataField* erhält die Werte *Species No* und *Common_Name*.
- Zur Anzeige der Bemerkungen wird *TDBMemo* ergänzt.
  - *DataSource* erhält wieder den Wert *Form1.DataSource1*.
  - Die Eigenschaft *DataField* wird auf *Notes* gesetzt.

- Zuletzt fügen wir noch eine *TDBImage* ein.
  - Als *DataSource* wählen wir *Form1.DataSource1*.
  - Die Eigenschaft *DataField* wird auf *Graphic* gesetzt.
  - Ziehen Sie die Komponente etwas breiter und setzen Sie dann *Stretch* auf *true*.

Erstellen der Eingabemaske mit Drag&Drop

Gerade dann, wenn viele Spalten vorhanden sind, ist es etwas mühselig, die Eingabemaske auf diese Weise zu erstellen. Es geht jedoch auch einfacher:

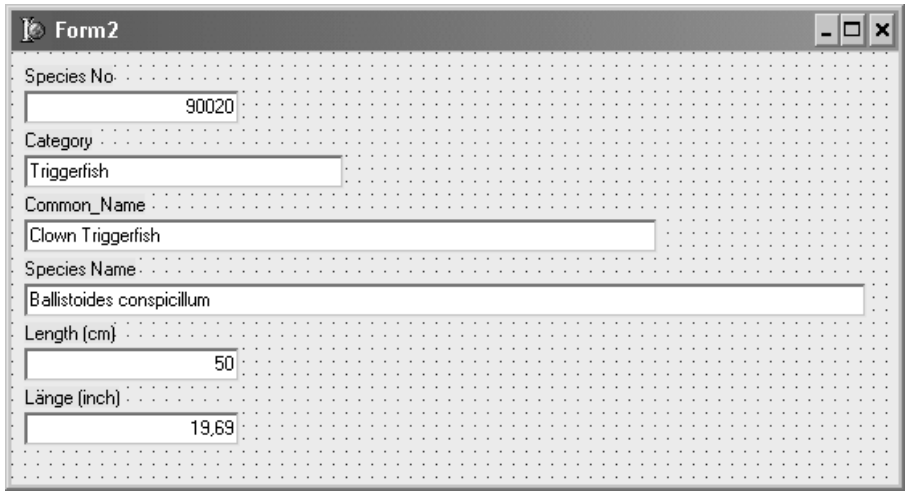
- Löschen Sie zunächst die bisherigen Komponenten.
- Rufen Sie den Felder-Editor von *Table1* auf und markieren Sie alle Felder mit Ausnahme von *Notes* und *Graphic*.

(Zu diesem Zweck klicken Sie auf den ersten Eintrag, halten dann die Shift-Taste und klicken auf den Eintrag *Length_In*.)

- Ziehen Sie nun diese Einträge mit *Drag&Drop* auf *Form2*. (Zu diesem Zweck klicken Sie auf die markierten Einträge in der Liste, lassen die [linke] Maustaste jedoch nicht los, sondern ziehen den Mauszeiger in die linke obere Ecke von *Form2*. Erst dort geben Sie die Maustaste frei.)

  Für die gewählten Spalten ist nun jeweils eine *TDBEdit*-Instanz angelegt, diese wird außerdem mit einem Label beschriftet. Die Beschriftung richtet sich nach der Eigenschaft *DisplayLabel* der entsprechenden *TField*-Instanz, für die Spalte *Length_In* ist sie demnach *Länge (inch)*. Auch hier funktioniert die Formatierung der Länge mit zwei Nachkommastellen.

Die Beschriftung der Spalte *Species No* haben wir jedoch nicht in der *TField*-Instanz, sondern in der dazugehörenden Spalte von *DBGrid1* vorgenommen. Folgerichtig ist sie auch nur dort wirksam.



- Die Breite der *TDBGrid*-Komponenten richtet sich nach der Länge der damit verbundenen Datenbankspalte und ist hier leider sehr ungleichmäßig. Das wollen wir sogleich ändern.

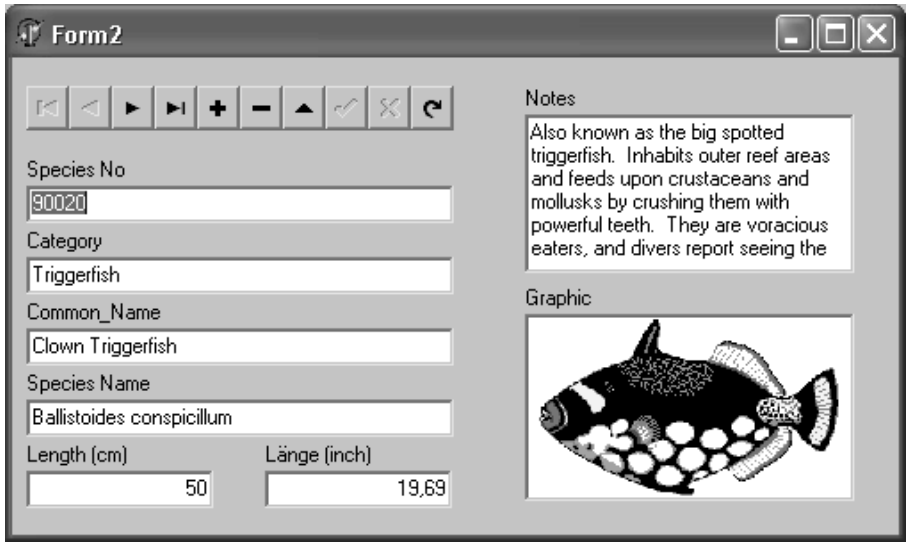
  Zu diesem Zweck wäre es nützlich zu wissen, wie lang die Feldinhalte bei den bereits vorhandenen Datensätzen tatsächlich sind. Um durch die Datenmenge zu scrollen, gibt es beim Felder-Editor einen kleinen Navigator.

  Mit einer Komponentenbreite von 200 Pixeln sollte man überall hinkommen, da wir aber gleich noch einen Navigator einfügen wollen, wählen wir 240 Pixel. Da diese Breite für die Spalten *Length (cm)* und *Length_In* viel zu breit ist, verkleinern wir diese beiden Eingabefelder und setzen sie nebeneinander.

- Nun ergänzen wir eine *TDBNavigator*-Instanz und setzen deren Eigenschaft *DataSource* auf *Form1.DataSource1*.

- Zuletzt werden noch die beiden Spalten *Notes* und *Graphic* aus dem Felder-Editor von *Table1* auf *Form2* gezogen.

Hier wird von Delphi automatisch eine TDBMemo- beziehungsweise eine *TDBImage*-Instanz angelegt. Ändern Sie von *DBImage1* die Proportionen und setzen Sie *Stretch* auf *true*.

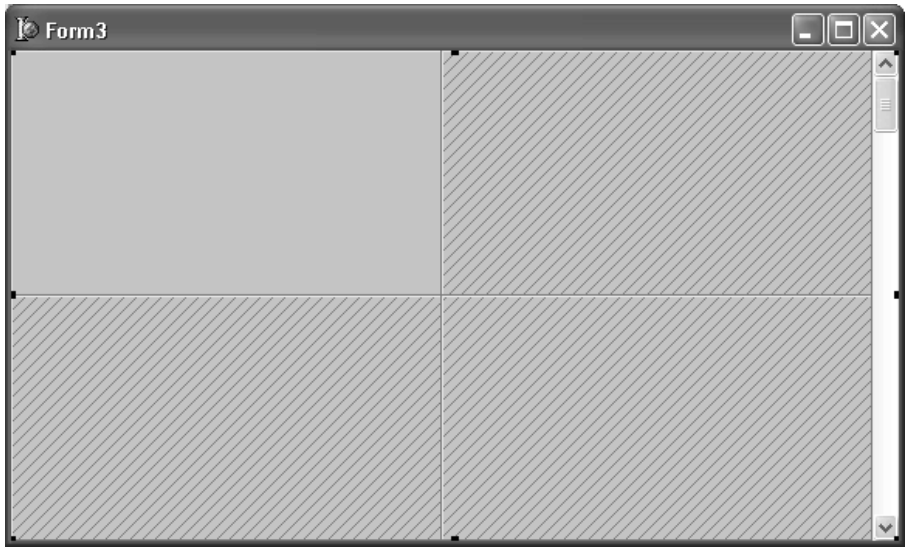


1.1.2 Verwendung von TDBCtrlGrid

Bei der Verwendung von *TDBGrid* können Sie recht viele Datensätze anzeigen, aber die Möglichkeiten, die Anzeige individuell zu gestalten, sind limitiert. Zudem kann *TDBGrid* weder Memos noch Bilder anzeigen.

Gestalten Sie eine individuelle Eingabemaske, so umgehen Sie diese Nachteile. Sie sind dann jedoch auf die Anzeige eines einzigen Datensatzes beschränkt. Mit *TDBCtrlGrid* bietet sich ein Mittelweg zwischen diesen beiden Lösungen.

- Fügen Sie dem Projekt ein neues Formular hinzu und von der Palettenseite *Datensteuerung* eine *TDBCtrlGrid*-Komponente ein.
- Setzen Sie zunächst die Eigenschaften *ColCount* und *RowCount* jeweils auf zwei, anschließend die Eigenschaft *Align* auf *alClient*.
- Binden Sie in der dazugehörenden Unit im *implementation*-Teil die *unit1* ein. Setzen Sie anschließend die Eigenschaft *DataSource* von *DBCtrlGrid1* auf *Form1.Table1*.
- *DBCtrlGrid1* ist in vier Bereiche geteilt. Der Bereich links oben dient zum Design der Maske, die übrigen Bereiche sind schraffiert.



Ziehen Sie die ersten vier Tabellenspalten aus dem Felder-Editor auf den Bereich links oben. Die *TDBEdit*-Komponenten für *Species No* und *Category* erhalten eine Breite von 150 Pixeln, *Common_Name* und *Species Name* eine Breite von 300 Pixeln.

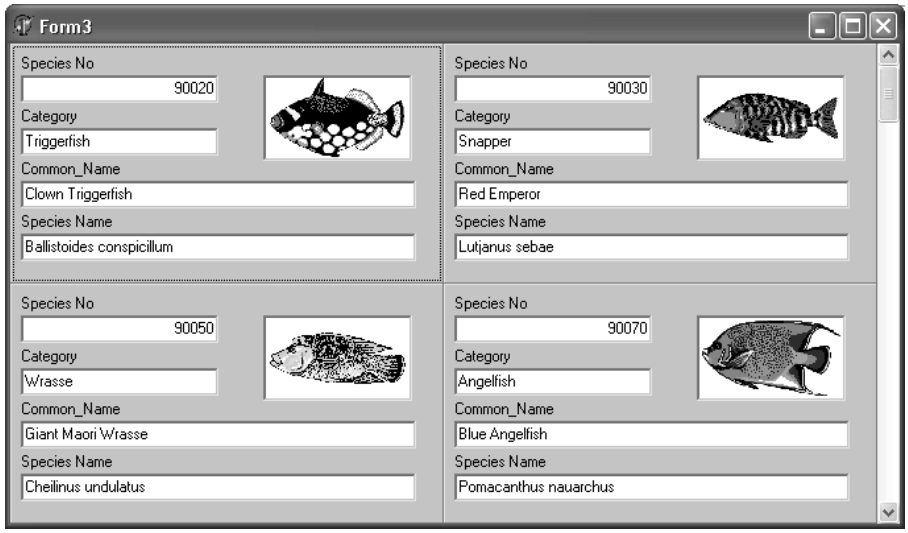
- Anschließend fügen Sie eine *TDBImage*-Komponente ein – diesmal nicht über den Felder-Editor, sondern über die Komponentenpalette. Vielleicht fällt Ihnen dabei auf, dass die Eigenschaft *DataSource* bereits beim Einfügen gesetzt wird – man könnte sie auch nicht abändern, weil sie mit dem Wert der Eigenschaft *DataSource* von *DBCtrlGrid1* übereinstimmen muss.

  Setzen Sie *DataField* auf *Graphic*, passen Sie die Größe an und ändern Sie *Stretch* auf *true*.

- Um das Formular zur Laufzeit überhaupt anzeigen zu können, fügen Sie in *Form1* einen Button ein und erstellen eine entsprechende Ereignisbehandlungsroutine.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Form3.Show;
end;
```

- Bei der Betrachtung des Formulars zur Laufzeit fällt auf, dass der aktuelle Datensatz zwar durch einen punktierten Rahmen gekennzeichnet wird, dass dieser jedoch nicht so richtig auffällig ist. Wollen Sie hier weniger subtil vorgehen, dann können Sie die *TDBCtrlGrid*-Eigenschaft *SelectedColor* auf einen anderen Wert setzen.



- Mit Hilfe von *SelectedColor* lässt sich aber auch der Status der Datenmenge visualisieren: Im Normalfall wird Hellgrau verwendet, wird ein Datensatz geändert, Rot/Braun, wird ein neuer Datensatz eingefügt, Grün.

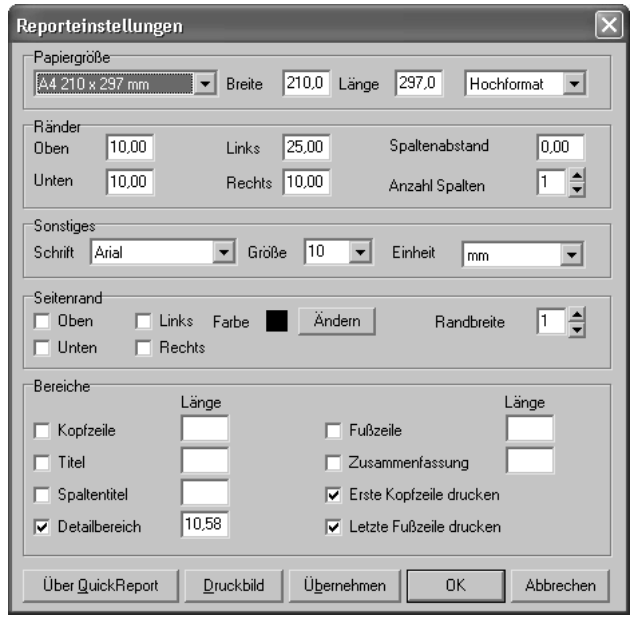
```
procedure TForm1.DataSource1StateChange(Sender: TObject);
begin
  if Assigned(Form3) and Form3.Visible then
  begin
    if Table1.State = dsEdit
      then Form3.DBCtrlGrid1.SelectedColor := clMaroon;
    if Table1.State = dsInsert
      then Form3.DBCtrlGrid1.SelectedColor := clGreen;
    if Table1.State = dsBrowse
      then Form3.DBCtrlGrid1.SelectedColor := clBtnHighlight;
  end;
end;
```

1.1.3 Drucken der Tabelle

Nun sollen die Daten auch noch gedruckt werden. Dafür wollen wir die QuickReport-Komponenten verwenden.

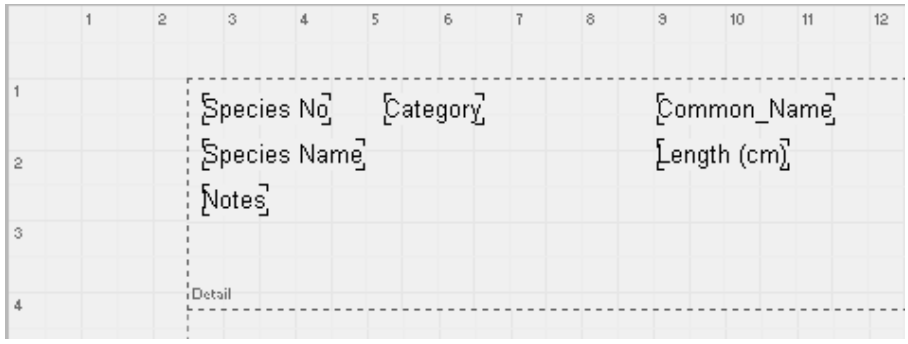
- Fügen Sie dem Projekt ein neues Formular hinzu und dort von der Palettenseite *QReport* die Komponente *TQuickRep* ein. Wie Sie diese Komponente auf dem Formular ausrichten, ist völlig belanglos.

- Mit einem Doppelklick auf *QuickRep1* rufen Sie den Dialog zur Einstellung der Reporteigenschaften auf.



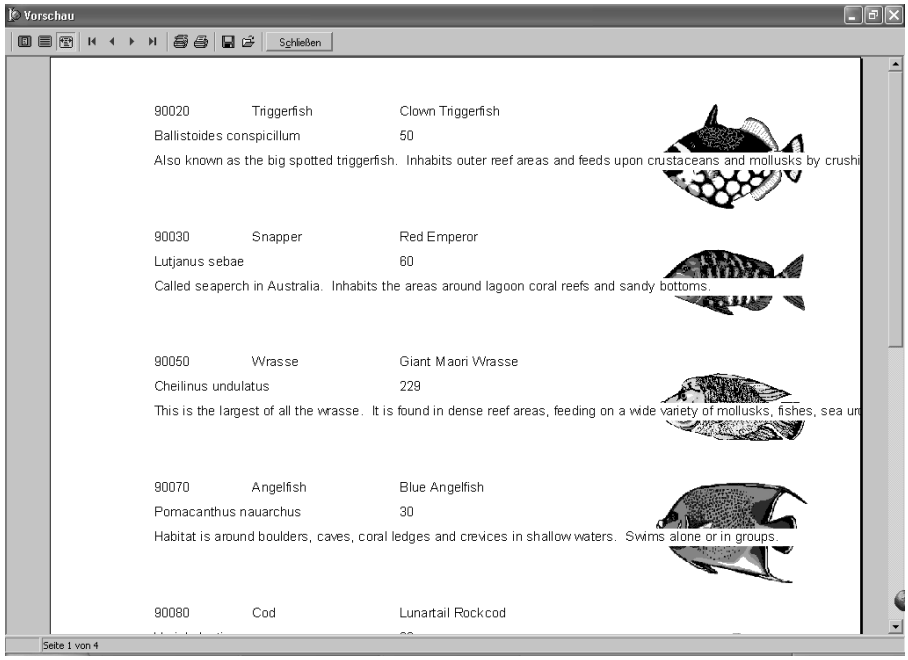
Setzen Sie hier den linken Rand auf 25 mm und markieren Sie die Checkbox *Detailbereich*. Die Länge des Detailbandes spielt keine Rolle, da wir es ohnehin visuell setzen werden.

- Setzen Sie die Eigenschaft *DataSet* von *QuickRep1* auf *Form1.Table1*. Damit das funktioniert, muss *unit1* eingebunden werden.
- Fügen Sie sechs *TQRDBText*-Komponenten ein und richten Sie diese so aus, wie dies in der folgenden Abbildung gezeigt wird:



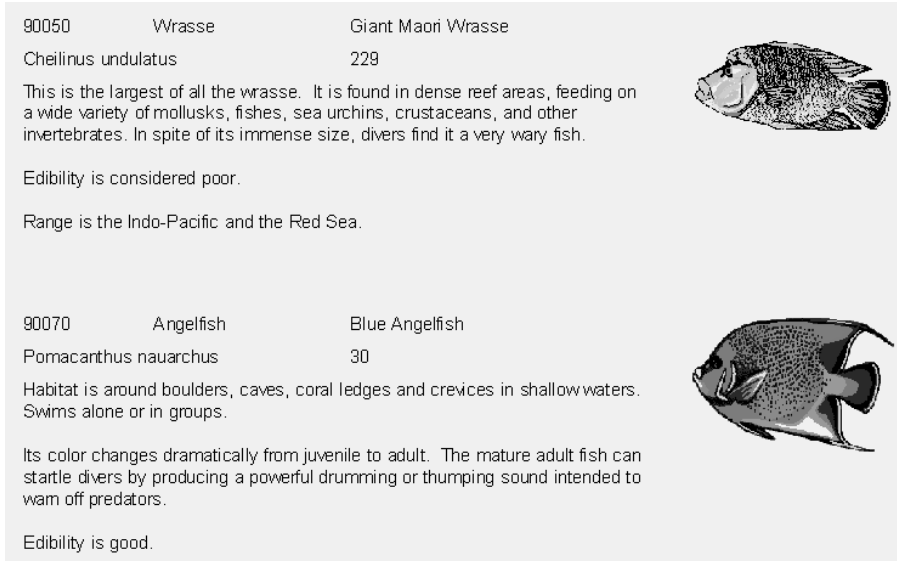
Um diese Komponenten mit dem dazugehörenden Datenbankfeld zu verbinden, setzen Sie die Eigenschaft *DataSet* auf *Form1.Table1* sowie die Eigenschaft *DataField* auf den entsprechenden Wert.

- Auf der rechten Seite ergänzen Sie nun *TQRDBImage* und setzen *DataSet* auf *Form1.Table1* sowie *DataField* auf *Graphic*. Passen Sie die Proportionen an und setzen Sie *Stretch* auf *true*.
- Nun lassen Sie sich die Vorschau anzeigen, indem Sie aus dem Kontextmenü von *QuickRep1* (nicht über dem Detailband!) den Menüpunkt Druckbild wählen:



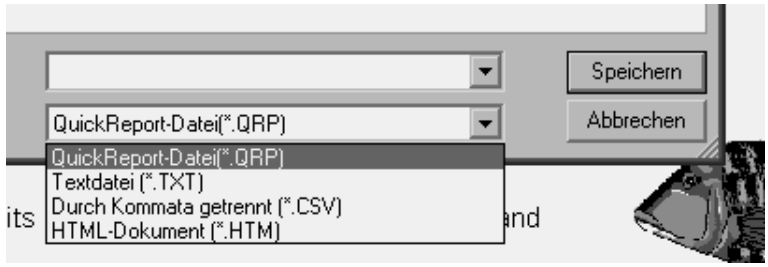
- Teilweise sieht die Darstellung schon ganz brauchbar aus, nur das Feld *Notes* macht Probleme, die wir jedoch sogleich beheben werden:
  - Die Eigenschaft *AutoSize* wird auf *false* gesetzt. Damit verhindern wir, dass die Komponente breiter wird als im Designer eingestellt.
  - Anschließend wird die Breite so gesetzt, dass der Text bis kurz vor das Bild geht. (Gegebenenfalls ändern Sie zunächst die Eigenschaft *Width* im Objektinspektor.)
  - Dann setzen Sie die Eigenschaft *AutoStretch* auf *true*. Ist nun in den Bemerkungen mehr Text, als in einer Zeile angezeigt werden kann, dann werden automatisch mehrere Zeilen eingefügt, gegebenenfalls wird die Höhe des Bandes vergrößert.

- Nun lassen wir uns noch mal die Seitenvorschau anzeigen:



- Aus der Vorschau heraus könnten Sie den Report nun drucken. Es besteht auch die Möglichkeit, den Report abzuspeichern, momentan jedoch nur im Quickreport-eigenen *qrp*-Format. Dies wollen wir gleich ändern: Fügen Sie die Komponenten *TQRTextFilter*, *TQRCSVFilter* und *TQRHTMLFilter* ein.

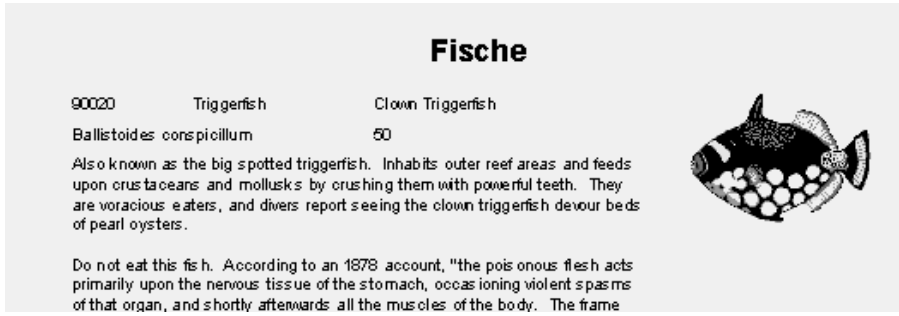
Eigenschaften dieser Komponenten können und müssen nicht gesetzt werden, allenfalls bei *TQRCSVFilter* können Sie das Trennzeichen spezifizieren. Nun können Sie den Report als ASCII-Text, als durch Kommata getrennte Werteliste und als HTML-Datei abspeichern, allerdings werden bei keinem der Formate Bilder mitgespeichert.



Einfügen eines Titels

Wir wollen den Report optisch noch ein wenig aufbessern, indem wir einen Reporttitel erstellen.

- Setzen Sie die Eigenschaft *Bands.HasTitle* auf *true*. Es wird automatisch ein neues Band eingefügt, dessen Eigenschaft *BandType* auch schon auf *rbTitle* gesetzt ist.
- Setzen Sie auf dieses Band eine *TQRLabel*-Komponente:
  - *AlignToBand* erhält den Wert *true*,
  - *Alignment* den Wert *taCenter*,
  - als *Caption* geben wir *Fische* ein
  - und der *Font* soll 18 Punkt groß und fett werden.



Starten des Reports zur Laufzeit

Bislang haben wir uns den Report lediglich zur Entwurfszeit angesehen. Soll aus dem laufenden Programm heraus der Report aufgerufen werden, dann ist die Methode *Print* oder *Preview* von *TQuickRep* aufzurufen. Mit *Preview* wird die Vorschau angezeigt, aus der heraus dann gedruckt werden kann. Mit *Print* wird der Report ohne Vorschau gleich gedruckt.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  Form4.QuickRep1.Preview;
end;
```

Startet man den Druck des Reports ohne Vorschau, dann besteht die Gefahr, dass das Ergebnis nicht so aussieht, wie der Anwender es sich vorstellt, oder die Anzahl der Seiten ist höher als erwartet. Von daher empfehle ich, grundsätzlich über die Vorschau zu gehen.

1.2 Einen BDE-Alias einrichten

In Kapitel 1.3 wollen wir ein wenig mit *TQuery* arbeiten. Dafür verwenden wir eine Datenbank, die auf der Buch-CD zu finden ist. Diese muss auf die Festplatte kopiert werden, anschließend richten wir darauf einen BDE-Alias ein.

Kopieren der Datenbank

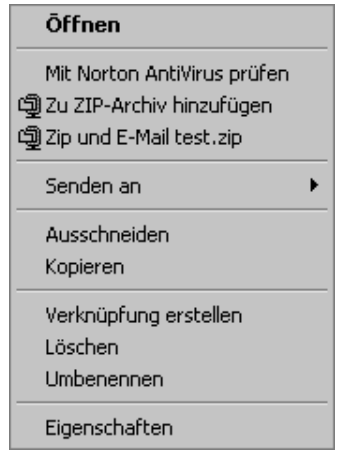
Mit Hilfe des Windows-Explorers kopieren wir das Verzeichnis *Tabellen\Test* von der beiliegenden CD irgendwohin auf die lokale Festplatte. Die genaue Position ist dabei nebensächlich, solange Sie sich den Pfad bis zum Einrichten des Aliases merken können.

Löschen des Schreibgeschützt-Flag

Damit die BDE mit den Tabellen arbeiten kann, muss darauf ein Schreibzugriff möglich sein – aus diesem Grund funktioniert es auch nicht, den Alias auf dem entsprechendem Verzeichnis auf der CD einzurichten. Wenn Sie jedoch Dateien von der CD auf die Festplatte kopieren, dann ist noch das Schreibgeschützt-Flag gesetzt. Das müssen wir ändern.

Wählen Sie im Windows-Explorer eine beliebige Datei aus der Datenbank auf der Festplatte und betätigen die Tastenkombination CTRL+A (auf deutschen Tastaturen STRG+A). Nun sind alle Dateien in diesem Verzeichnis selektiert.

Mit der rechten Maustaste rufen Sie nun das Kontextmenü dieser Dateien auf:



Wählen Sie hieraus den Menüpunkt EIGENSCHAFTEN, es ist der letzte Menüpunkt.

Es öffnet sich der Eigenschaftsdialog, die darin enthaltene Checkbox *Schreibgeschützt* ist markiert, und das müssen Sie ändern.

Nun können Sie den Dialog mit Ok wieder schließen.

Was ist ein BDE-Alias?

Eine dBase- oder eine Paradox-Datenbank ist ein Verzeichnis, in dem die dazugehörenden Tabellen- und Indexdateien liegen. Würde man den Pfad zu diesem Verzeichnis im Programm hart kodieren, dann müsste bei allen Rechnern, die dieses Programm verwenden, die Datenbank denselben Pfad haben. Würde die Datenbank in einem anderen Verzeichnis installiert oder dorthin verschoben, dann würde das Programm nicht mehr laufen.

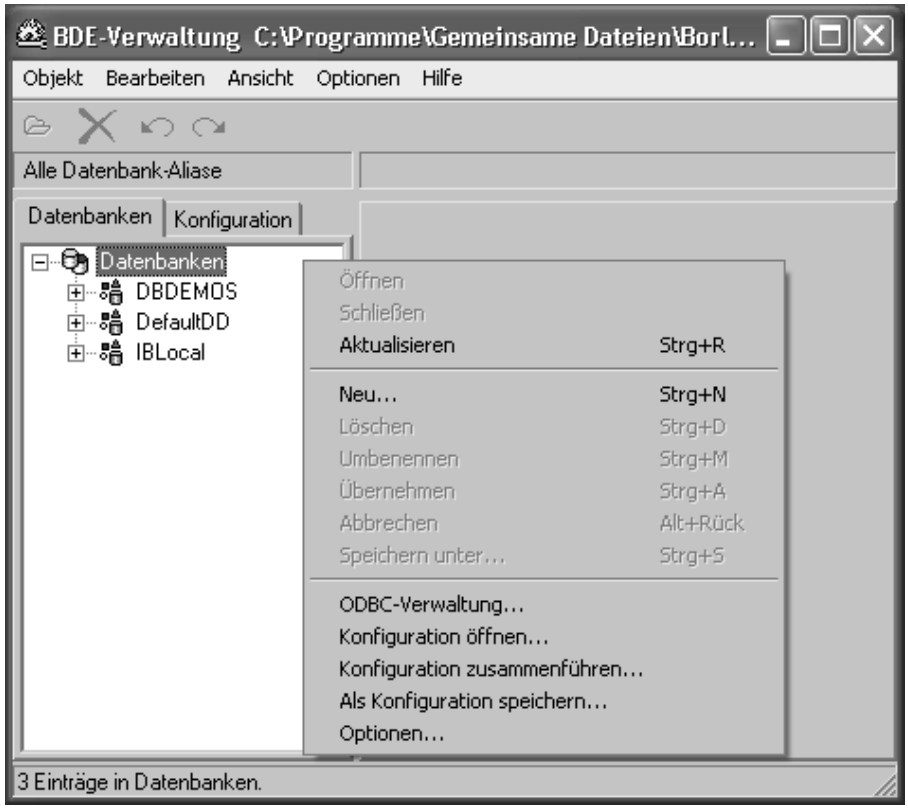
Um das zu vermeiden, weist man jeder Datenbank einen Namen, einen so genannten Alias zu. In der BDE-Verwaltung (oder im Datenbank-Explorer) findet man nun eine Liste der vergebenen Aliase sowie die dazugehörenden Pfade. (Bei Aliasen, die auf einen SQL-Server eingerichtet sind, kann man darüber hinaus noch einige Parameter einstellen.)

Möchte nun ein Anwender seine Datenbank in ein anderes Verzeichnis verschieben, dann kann er dies tun, muss aber aus der Systemsteuerung die BDE-Verwaltung aufrufen und den Pfad des betreffenden Aliases entsprechend abändern.

Den Alias einrichten

- Wählen Sie aus der Windows-Systemsteuerung die BDE-Verwaltung auf.
- Auf der linken Seite finden Sie eine Liste der bereits eingerichteten BDE-Aliase. Vielleicht benutzen Programme, die sich schon auf Ihrem Rechner befinden, auch die BDE, so dass Sie hier einige Einträge mehr finden als hier auf dem Bild. Sollten ODBC-Aliase vorhanden sein, dann sind auch diese in der Liste zu finden.

  Öffnen Sie das Kontextmenü (rechte Maustaste) und wählen Sie den Eintrag NEU...



- Im dann erscheinenden Fenster wählen Sie den Eintrag *Standard* aus.

- Nun geben Sie dem neuen Alias den Namen *Test*.

- In der Zeile *PATH* wird nun der Pfad des Datenbank-Verzeichnisses eingegeben (oder ausgewählt). Die anderen Parameter brauchen nicht verändert zu werden.

Typ	STANDARD
DEFAULT DRIVER	PARADOX
ENABLE BCD	FALSE
PATH	D:\D6_data\test ...

- Anschließend wählen Sie aus dem Kontextmenü den Eintrag ÜBERNEHMEN. Die erscheinende Sicherheitsabfrage beantworten Sie mit OK.

Damit ist der BDE-Alias eingerichtet.

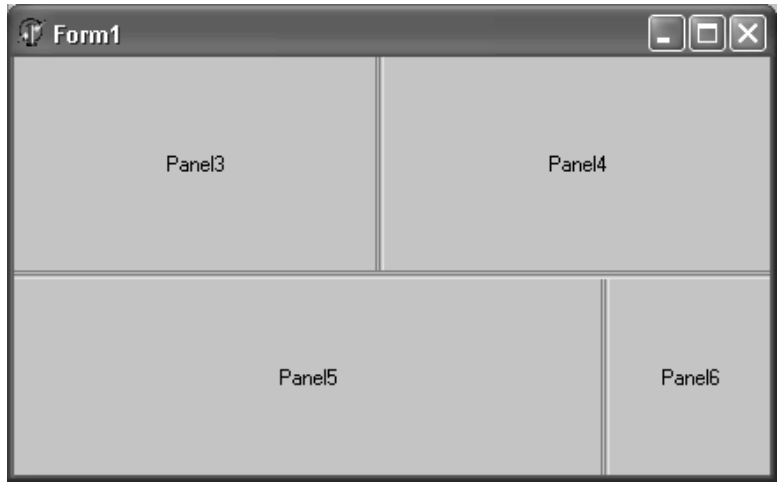
1.3 Abfragen

In der Test-Datenbank finden Sie die Auftragsdatenbank einer fiktiven Firma, die mit Hardware handelt. Wir wollen jetzt ein Formular erstellen, in dem ein Kunde dieser Firma angezeigt wird, zu diesem Kunden die Bestellungen, die er bislang getätigt hat, und zu diesen Bestellungen die einzelnen Posten. Es handelt sich dabei also um eine zweifache Master-Detail-Verknüpfung. Zudem wollen wir den Verlauf der Auftragssumme in einem Diagramm anzeigen.

Bei diesem Projekt wollen wir die Komponente *TQuery* einsetzen, welche mit der Datenbanksprache SQL arbeitet. SQL wird ausführlicher in Kapitel 6 beschrieben.

Das Anzeigeformular

Zunächst wollen wir das Anzeigeformular vorbereiten:



- Fügen Sie ein Panel ein:
  - Setzen Sie *Align* auf *alTop*,
  - ziehen Sie das Panel auf etwa halbe Formularhöhe und
  - ändern Sie *BevelOuter* auf *bvNone*.
- Ergänzen Sie *TSplitter* (Palettenseite *Zusätzlich*)
  - Setzen Sie *Align* auf *alTop*,
  - *Height* auf 5 und
  - *Beveled* auf *true*.

- Es folgt ein weiteres Panel:
  - *Align* auf *alClient* und
  - *BevelOuter* auf *bvNone*.

Das Formular ist nun in einen oberen und einen unteren Bereich getrennt. Die Größe dieser Bereich können durch den Splitter variiert werden. Beide Bereiche werden wiederum in zwei Bereiche getrennt, diesmal aber vertikal. Auch zwischen den dafür erforderlichen Panels wird jeweils ein Splitter eingefügt. Bei allen vier Panels wird die Eigenschaft *BorderWidth* auf 8 gesetzt, außerdem die *Caption* gelöscht (auf einen leeren String gesetzt).

Das Datenmodul

In Kapitel 1.1 haben wir *Table1* und *DataSource1* direkt auf das Formular gelegt. Dies war weiter kein Problem, schließlich werden die Icons zur Laufzeit nicht angezeigt.

In diesem Beispiel werden wir aber neun solcher nicht visuellen Komponenten benötigen, und dann wird es langsam unübersichtlich. Deshalb wollen wir ein Datenmodul verwenden. Dabei handelt es sich um ein Formular, das zur Laufzeit nicht angezeigt wird und das auch nur nicht visuelle Komponenten aufnimmt.

Dazu wählen Sie den Menüpunkt DATEI|NEU|DATA MODULE. (Zumindest in der mir vorliegenden Delphi-Version ist dieser Menüpunkt nicht übersetzt worden.) Es wird nun ein weiteres Fenster angezeigt, auf das Sie nicht visuelle Komponenten legen können.

Erschrecken Sie nicht, wenn es in der Komponentenpalette etwas »übersichtlicher« wird. Alle Komponenten, die nicht in ein Datenmodul eingefügt werden können, werden nun ausgeblendet. Keine Sorge, das wird sich ändern, sobald ein normales Formular wieder den Fokus hat.

Das Datenmodul heißt zwar Datenmodul und ist für diesen Zweck in Delphi 2 eingeführt worden, es lassen sich aber auch andere nicht visuelle Komponenten dort unterbringen als nur datensensitive.

Wie ein normales Formular hat ein Datenmodul auch eine dazugehörende Unit, die in alle Units eingebunden werden muss, die auf Komponenten des Datenmoduls zugreifen wollen.

1.3.1 Anzeige der Kunden

In das Datenmodul fügen Sie jetzt eine *TQuery*- und eine *TDataSource*-Instanz ein. *TQuery* finden Sie auf der Palettenseite *BDE*. Im ersten Beispiel haben wir den Namen der Komponenten so belassen, wie er von Delphi vergeben wurde – gerade bei kleinen Beispiel- und Testprojekten lohnt es sich meist nicht, über eine besonders anschauliche Benennung von Komponenten nachzudenken. Da Im Datenmodul jedoch alle Komponenten beschriftet dargestellt werden, wollen wir diese beiden Komponenten mit *qrKunde* und *dsKunde* benennen.

Die Eigenschaft *DataSet* von *dsKunde* setzen wir auf *qrKunde* und die Eigenschaft *DatabaseName* von *qrKunde* auf *test*. In der Eigenschaft *SQL* von *qrKunde* wird eine SQL-Anweisung eingegeben, genauer gesagt, eine SELECT-Anweisung.

Den einfachsten Zugriff auf die Tabelle *t_kunde* würde man wie folgt formulieren:

```
SELECT *
  FROM t_kunde
```

Damit würde die Anweisung alle Spalten (und ohnehin alle Reihen) der Tabelle *t_kunde* liefern. Wir werden hier jedoch eine etwas kompliziertere Anweisung verfassen:

```
SELECT nummer,
    vorname || "   " || nachname AS name,
    strasse,
    plz || "   " || ort AS ort,
    tel
  FROM t_kunde
```

Hier werden nun die Zeilen einzeln aufgelistet, welche uns interessieren. Zudem werden für den Namen und die Adresse jeweils zwei Spalten zusammengefasst: Mit den zwei senkrechten Strichen (man spricht hier von einer Pipe) fügt man zwei Strings zusammen, in diesem Fall beispielsweise *vorname* und *nachname*. Zwischen Vor- und Nachname sollte man mindestens ein Leerzeichen setzen, hier verwenden wir gleich drei.

Dieser Kombinationsspalte geben wir den Bezeichner *name*. Wenn wir die Datenmenge in einem Grid anzeigen würden, dann würde die Spaltenüberschrift genauso lauten, und auch wenn wir im Objektinspektor die Eigenschaft *DataField* setzen, wird dieser Bezeichner verwendet.

Setzen wir nun die Eigenschaft *Active* auf *true*. (Dazu ein Hinweis: Wenn Sie die SQL-Anweisung ändern, wird die Eigenschaft *Active* automatisch auf *false* gesetzt. Sie müssen die Datenmenge dann explizit wieder öffnen.)

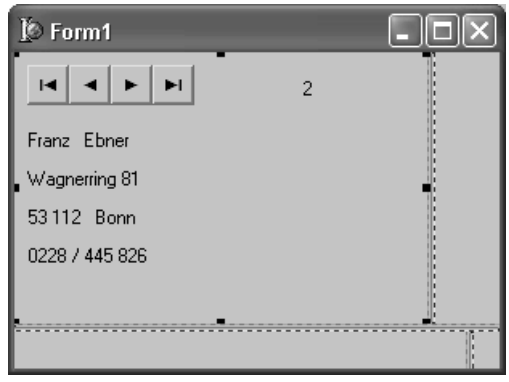
Gehen Sie nun nach *Form1*, fügen Sie links oben einen *TDBNavigator* ein und setzen Sie dessen Eigenschaft *DataSource* auf *DataModule2.dsKunde*. Dabei fällt auf, dass die Buttons zum Ändern eines Datensatzes oder zum Einfügen eines neuen Datensatzes nicht verfügbar sind:

Per Voreinstellung (*RequestLive* hat den Wert *false*) liefert *TQuery* keine aktualisierbare Datenmengen, das Ergebnis von Abfragen kann dann nur zur Anzeige verwendet werden.

Bei manchen Abfragen kann man das ändern, indem man (bevor *Active* auf *true* gesetzt wird) die Eigenschaft *RequestLive* auf *true* setzt. Bei der hier verwendeten Abfrage würde selbst dies nicht helfen – woher sollte die BDE beispielsweise wissen, wie sie einen neuen Wert für *name* auf die Spalten *vorname* und *nachname* verteilen sollte.

Gestaltung der Anzeige

Da wir ohnehin keine Eingaben machen können, verwenden wir zur Anzeige der einzelnen Felder nicht *TDBEdit*, sondern *TDBText*. Um die nicht benötigten Buttons von *DBNavigator1* auszublenden, verwenden Sie dessen Eigenschaft *VisibleButtons*.



1.3.2 Anzeige der Bestellungen

Wir wollen nun im Feld links unten die Bestellungen anzeigen. Dabei sollen jedoch nicht alle Bestellungen angezeigt werden, sondern nur diejenigen des betreffenden Kunden. Wir müssen also eine Master-Detail-Verknüpfung erstellen.

- Fügen Sie in das Datenmodul eine neue *TQuery*-Instanz ein und ändern deren Namen auf *qrBestellung*.
  - Die Eigenschaft *DatabaseName* erhält wieder den Wert *test*.
  - *DataSource* wird auf *dsKunde* gesetzt. Dies ist der erste Schritt zur Erstellung der Master-Detail-Verknüpfung.
  - Formulieren Sie folgende *SQL*-Anweisung:

    ```
    SELECT *
     FROM t_bestellung
     WHERE kunde = :nummer
    ```

    Mit dieser Anweisung holen wir alle Spalten der Tabelle *t_bestellung*, aber nur diejenigen Zeilen, deren Feld *kunde* den gleichen Wert hat wie der Parameter *:nummer*. Der Parameter wird über die Master-Datenmenge bezogen, also über *qrKunde*, und ist der Wert des jeweils aktiven Feldes *nummer*. Wird dort ein anderer Kunde angezeigt, dann wird auch diese Abfrage aktualisiert und somit eine neue Liste der Bestellungen angezeigt.
  - Setzen Sie *Active* auf *true*.
- Fügen Sie eine *TDataSource*-Instanz in das Datenmodul ein, benennen Sie diese in *dsBestellung* um und wählen Sie als *Dataset* die Komponente *qrBestellung*.
- Nun positionieren Sie die Komponente *TDBGrid* auf dem linken unteren Feld des Formulars.
  - Setzen Sie *Align* auf *alClient*,
  - *DataSource* bekommt den Wert *DataModule2.dsBestellung*.

Sie bekommen so zu jedem Kunden die Liste seiner Bestellungen angezeigt:

NUMMER	KUNDE	DATUM	BEARBEITER
6763	1	04.07.1996 21:16:25	9
11858	1	16.03.1997 23:54:40	27
41921	1	19.05.1996 18:38:22	10
46950	1	23.09.1997 15:41:39	5
49658	1	02.02.1998 18:11:43	14
51172	1	28.04.1997 19:18:43	10

Das Ergebnis ist jedoch an einigen Stellen noch verbesserungsfähig:

- Die Anzeige der Spaltentitel in Großbuchstaben ist optisch zu aufdringlich.
- Die Spalte *Kunde* könnte ganz entfallen, die Kundennummer bekommt man ohnehin bei der Adresse angezeigt.
- Die Anzeige der Uhrzeit beim Datum ist dann doch etwas übertrieben.
- Statt der Mitarbeiternummer des Bearbeiters sollte lieber dessen Name angezeigt werden.

Beginnen wir mit dem letzten Punkt:

Verbinden zweier Tabellen mittels eines JOINS

Brav nach den Regeln normalisierter Datenbanken wird in der Tabelle *t_bestellung* nicht der Name des Bearbeiters gespeichert, sondern dessen Mitarbeiternummer. Der Name des Mitarbeiters steht in der Tabelle *t_mitarbeiter* und muss jeweils mit Hilfe der Mitarbeiternummer ermittelt werden.

Keine Sorge, dies bedingt keinen Programmieraufwand, dafür müssen wir lediglich mittels einer bestimmten SQL-Anweisung, nämlich anhand eines JOINS, zwei Tabellen kombinieren. Ändern Sie die Anweisung *SQL* von *qrBestellung* wie folgt:

```
SELECT b.nummer,
     b.datum,
     m.nachname || ", " || m.vorname AS bearbeiter
  FROM t_bestellung b
     INNER JOIN t_mitarbeiter m
       ON b.bearbeiter = m.nummer
  WHERE kunde = :nummer
```

Was genau ein INNER JOIN ist und worin die Unterschiede zum OUTER JOIN liegen, werden wir in Kapitel 6.1 noch ausführlich besprechen. Im Moment nur so viel: Es werden die beiden Tabellen *t_bestellung* und *t_mitarbeiter* verknüpft, und zwar so, dass der Wert der Spalte *bearbeiter* (Tabelle *t_bestellung*) mit dem Wert der Spalte *nummer* (Tabelle *t_mitarbeiter*) übereinstimmt.

Sobald eine Abfrage mehr als eine Tabelle verwendet, müsste man eigentlich vor jeden Spaltenbezeichner den Tabellenbezeichner setzen. Die ON-Klausel hätte also wie folgt formuliert werden müssen:

```
ON t_bestellung.bearbeiter = t_mitarbeiter.nummer
```

Da auf diese Weise die Abfragen schnell unübersichtlich würden (und eine Abfrage kann schon mal mehrere hundert Zeilen lang sein), hat man die Tabellen-Aliase eingeführt: Nach Nennung des Tabellennamens in der FROM-Klausel wird ein Alias genannt, in diesem Fall der Buchstabe *b* beziehungsweise *m*.

Dieser Tabellen-Alias ersetzt an allen anderen Stellen den Tabellen-Bezeichner, nicht nur in der ON-Klausel, sondern auch in der Spaltenliste, in der WHERE-Klausel und was es sonst noch so gibt.

Aufmerksamen Lesern wird nicht entgangen sein, dass wir bei dieser SQL-Anweisung keinen Alias in der WHERE-Klausel verwenden. In der Regel kommen SQL-Systeme auch ganz gut ohne solche Tabellen-Aliase zurecht, solange der Spaltenbezeichner nur in einer der beteiligten Tabellen verwendet wird.

Wird aber ein und derselbe Bezeichner in mehreren Tabellen als Spaltenname verwendet, dann ist das Verhalten der einzelnen SQL-Systeme unterschiedlich. Bei manchen Systemen ist es zufällig, welche Spalte verwendet wird, andere wiederum verwenden die zuerst genannte Tabelle, bisweilen wird eine Fehlermeldung ausgegeben. Um hier potenzielle Fehler auszuschließen, sollten Sie grundsätzlich Tabellen-Bezeichner (oder -Aliase) voranstellen.

Es gibt zwei Mitarbeiter mit dem Namen *Müller*. Selbst wenn dem nicht so wäre, müsste man stets damit rechnen, dass durch eine Neueinstellung ein Name öfters vorkommt. Deswegen verwenden wir Vor- und Nachnamen, die wir wieder zu einer Spalte zusammenfügen. (Lassen Sie mal versuchsweise die Spalten-Umbenennung mit AS weg ...)

Optimieren der Darstellung

Von der Spalte *Kunde* haben wir uns schon beim Formulieren der Spaltenliste getrennt. Jetzt beheben wir noch die anderen optischen Mängel:

- Legen Sie für *qrBestellung* persistente *TField*-Instanzen an:
  - Doppelklick auf das Komponenten-Icon.
  - Im Felder-Editor aus dem Kontextmenü (rechte Maustaste) den Eintrag ALLE FELDER HINZUFÜGEN wählen.
- Da wir in der SQL-Anweisung die Spaltennamen klein geschrieben haben, werden sie auch durchgehend mit Kleinbuchstaben dargestellt. Ändern wir das jeweils mit der Eigenschaft *DisplayLabe*l.
- Als *DisplayFormat* von *qrBestellDatum* setzen wir auf *dd.mm.yyyy*, *DisplayWidth* kann darauf auf 12 reduziert werden.
- Die Breite der Spalte Bearbeiter richtet sich nach der maximal möglichen Länge von Vorname plus Nachname (plus Komma und Leerzeichen). Dies wird nicht mal annäherend ausgenutzt, setzen Sie dehalb *DisplayWidth* auf 25.

1.3.3 Die Posten der Bestellung

Für die jeweils selektierte Bestellung sollen rechts unten die einzelnen Posten angezeigt werden.

- Fügen Sie in das Datenmodul eine neue *TQuery*-Instanz ein und ändern Sie deren Namen auf *qrPosten*.
  - Die Eigenschaft *DatabaseName* erhält wieder den Wert *test*.
  - *DataSource* wird zwecks Erstellung einer Master-Detail-Verknüpfung auf *dsBestellung* gesetzt.
  - Formulieren Sie folgende *SQL*-Anweisung:

```
SELECT *
  FROM t_posten
  WHERE bestellung = :nummer
```

    Wie schon bei *qrBestellung* wird die Ergebnismenge über die Master-Detail-Verknüpfung gefiltert.

  - Setzen Sie *Active* auf *true*.
- Fügen Sie eine *TDataSource*-Instanz in das Datenmodul ein, benennen Sie diese in *dsPosten* um und wählen Sie als *Dataset* die Komponente *qrPosten*.
- Nun positionieren Sie die Komponente *TDBGrid* auf dem rechten unteren Feld des Formulars. *Align* wird wieder auf *alClient* gesetzt, *DataSource* bekommt den Wert *DataModule2.dsBestellung*.

Sie bekommen jetzt zu jeder Bestellung die einzelnen Posten angezeigt:

NUMMER	BESTELLUNG	STUECKZAHL	ARTIKEL	PREIS
187182	46950	1	31	639
187183	46950	0	71	1179
187184	46950	0	30	499

Aber auch hier entspricht das Ergebnis noch nicht ganz unseren Vorstellungen:

- Die Anzeige der Spaltentitel in Großbuchstaben ist optisch zu aufdringlich.
- Die Spalte *Bestellung* könnte ganz entfallen, diese Nummer bekommt man ohnehin bei der Liste der Bestellungen angezeigt. Die Angabe der Postennummer ist auch ohne besonderen Informationsgehalt.
- Statt der Artikelnummer wäre die Artikelbezeichnung aussagekräftiger.
- Der Preis sollte mit zwei Nachkommastellen angezeigt werden, außerdem wäre neben dem Einzelpreis auch der Gesamtpreis von Interesse.

Die meisten Probleme können wir mit einer Umformulierung der SQL-Anweisung beheben:

```
SELECT
     p.stueckzahl AS Stueckzahl,
     a.bezeichnung AS Artikel,
     p.preis AS Einzelpreis,
     p.stueckzahl * p.preis AS Gesamtpreis
  FROM t_posten p
     INNER JOIN t_artikel a
       ON a.nummer = p.artikel
  WHERE bestellung = :nummer
```

Des Weiteren setzen wir die Eigenschaft *Currency* von Einzel- und Gesamtpreis auf *true* – nun werden diese Zahlen auf zwei Nachkommastellen genau ausgegeben, allerdings mit der Währungsangabe EUR. Die Preise in der Datenbank – ohnehin längst überholt – sind jedoch DM-Preise. Also setzen wir Currency wieder auf *false* und verwenden für die Eigenschaft *DisplayFormat* dieser beiden TField-Instanzen den Wert *0.00 DM*.

Stueckzahl	Artikel	Einzelpreis	Gesamtpreis
9	Thunder + 3940W	749,00 DM	6741,00 DM
1	PS/2 16 MB	34,00 DM	34,00 DM

1.3.4 Gesamtumsatz einer Bestellung

Den Gesamtumsatz der Bestellung könnten wir mit einem Join ermitteln, dazu wird die SQL-Anweisung von *qrBestellung* geändert:

```
SELECT
     b.nummer,
     b.datum,
     m.nachname,
     m.vorname,
     SUM (p.stueckzahl * p.preis)
  FROM t_bestellung b
     INNER JOIN t_mitarbeiter m
       ON b.bearbeiter = m.nummer
  INNER JOIN t_posten p
       ON p.bestellung = b.nummer
  WHERE kunde = :nummer
  GROUP BY b.nummer, b.datum, m.nachname, m.vorname
```

Nun ist allerdings die BDE-Engine von Paradox kein System, das JOINS besonders schnell ausführt, das Scrollen bei den Kunden wird also ziemlich träge. Zudem erlaubt die GROUP BY-Klausel keine zusammengesetzen Spalten, Vorname und Nachname sind demnach als einzelne Spalten abzufragen.

Umsatz als Nachschlagefeld

Je nach Rechnerkonfiguration kann ein berechnetes Feld ein wenig schneller sein. Zu diesem Zweck fügen wir die Komponente *qrRechnung* ein, welche die folgende SQL-Anweisung verwendet:

```
SELECT
     SUM(p.preis * p.stueckzahl) AS rechnungssumme
  FROM t_posten p
     INNER JOIN t_bestellung b
       ON p.bestellung = b.nummer
  WHERE b.nummer  = :nummer
```

Wir führen für *qrBestellung* ein berechnetes Feld ein, das wir aus *qrRechnung* füllen:

```
procedure TDataModule2.qrBestellungCalcFields(DataSet: TDataSet);
begin
  qrRechnung.Close;
  qrRechnung.Params[0].AsInteger
    := qrBestellungnummer.AsInteger;
  qrRechnung.Open;
  qrBestellungUmsatz.AsCurrency
    := qrRechnungrechnungssumme.AsCurrency;
end;
```

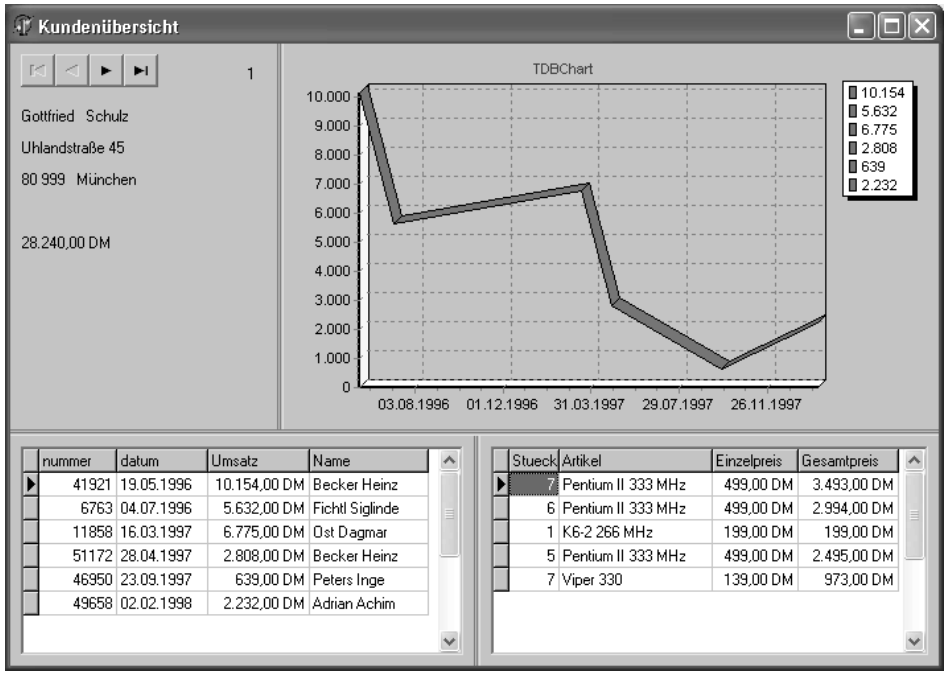
Gesamtumsatz eines Kunden

Auf ähnliche Weise ermitteln wir den Gesamtumsatz einer Bestellung. Da aber die Kundenadressen nicht in einem DBGrid angezeigt werden, benötigen wir kein berechnetes Feld, sondern hängen mittels einer DataSource eine *TDBText*-Instanz an.

```
SELECT
     SUM(p.preis * p.stueckzahl) AS Umsatz
  FROM t_posten p
     INNER JOIN t_bestellung b
       ON p.bestellung = b.nummer
  WHERE b.kunde = :nummer
```

1.3.5 Diagramm der Umsätze

Wenn Sie nicht schon wegen der Trägheit des Systems die Lust verloren haben – als kleiner Trost: Mit InterBase würde es deutlich schneller gehen –, dann wollen wir nun noch die Umsatzentwicklung des Kunden grafisch darstellen:

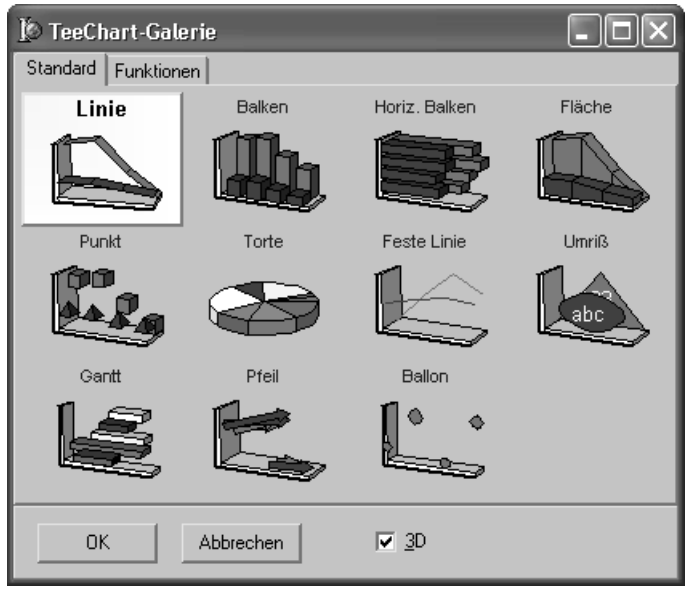


Für diesen Zweck setzen wir die Komponente *TDBChart* ein. Um dieser Komponente etwas »das Leben zu erleichtern«, sollen die einzelnen Datensätze in *qrBestellung* nach dem Datum sortiert werden:

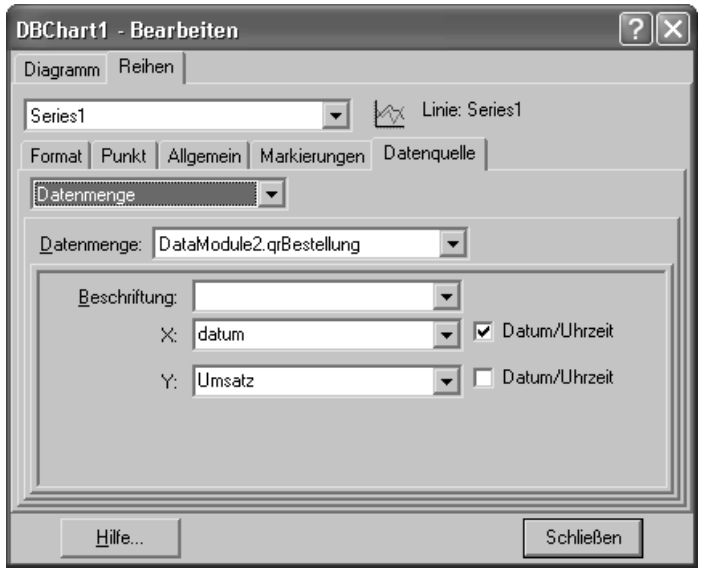
```
SELECT
    b.nummer,
    b.datum,
    m.nachname || " " || m.vorname AS Name
  FROM t_bestellung b
    INNER JOIN t_mitarbeiter m
      ON b.bearbeiter = m.nummer
  WHERE kunde = :nummer
  ORDER BY b.datum
```

Anschließend setzen wir auf das Panel rechts oben eine *TDBChart*-Instanz.

Mit einem Doppelklick gelangen wir in den Komponenteneditor und wählen *Hinzufügen*. Nun können wir auswählen, welche Art von Diagramm wir haben möchten:

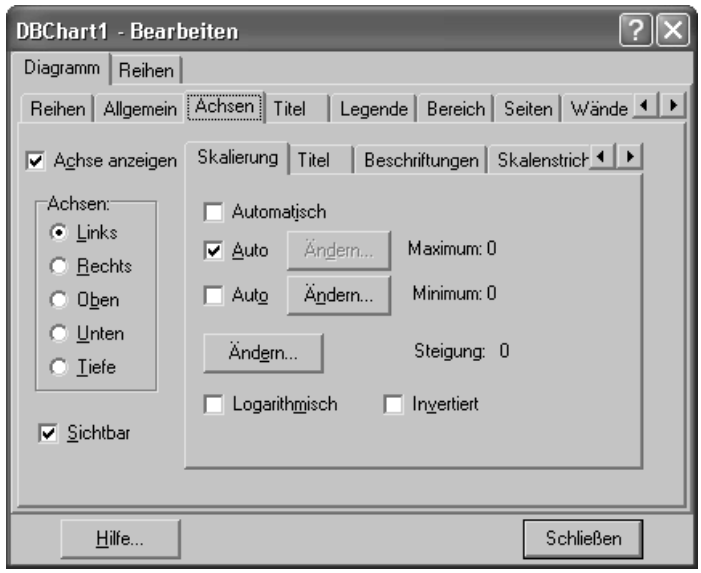


Der Einfachheit halber wählen wir ein Linien-Diagramm. Anschließend gehen wir zur Registerseite *Reihen* und dort auf *Datenquelle*:



Unsere Daten sollen aus einer Datenquelle kommen, genauer gesagt aus *qrBestellung*. Für die x-Achse verwenden wir das Datum, die y-Achse soll den Umsatz anzeigen.

Nun wollen wir noch die Achsenskalierung so abändern, dass die y-Achse immer bei 0 beginnt:



(Drei ineinander geschachtelte Register, ein Highlight übersichtlicher Benutzerführung ...)

Nun muss nur noch dafür gesorgt werden, dass die Anzeige beim Wählen eines neuen Kunden aktualisiert wird. Dazu muss lediglich *RefreshDataSet* aufgerufen werden:

```
procedure TDataModule2.qrKundeAfterScroll(DataSet: TDataSet);
begin
  qrUmsatz.Close;
  qrUmsatz.Params[0].AsInteger := qrKundenummer.AsInteger;
  qrUmsatz.Open;
  Form1.DBChart1.RefreshDataSet(qrBestellung, Form1.Series1);
end;
```

2 Datenbanken

Wir wollen uns nun ein wenig mit der Theorie von Datenbanken – vor allem von relationalen Datenbanken – beschäftigen und einige Begriffe klären.

2.1 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 ernst zu nehmendes) 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 Bilddateien. 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. Multiuser-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. Reden wir

- von einem Datenbestand, wenn wir die Daten meinen, und
- von einem Datenbanksystem, wenn wir von der Software sprechen.

2.1.1 Historisches

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

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 so genannter 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 weniger) direkt zugreifen. Bei der Suche nach bestimmten Kriterien war man dann aber immer noch auf die sequenzielle 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 enthält häufig Angaben über Bestelldatum, Anzahl, Bestellnummer, Bezeichnung, Einzel- und Gesamtpreis der gelieferten Waren und natürlich über den Kunden. Hier gibt es nun 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 sieben Mal gibt. Ohne Kundennummer hat man nun ein Problem.

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

  Wir haben hier das Problem, dass wir eine Referenz auf einen Datensatz haben, der nicht mehr existiert. Hier ist die *Integrität* der Datenbank verletzt.

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 jedes Mal von neuem die erforderlichen Sicherungen einzufügen ist unökonomisch (und dazu 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 Delphi werden Sie nicht darauf zugreifen können. Eine Beschreibung dieser Datenbanken soll aus diesem Grund hier unterbleiben.

2.1.2 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.

Für viele Fälle ist die Stand-Alone-Datenbank die sinnvollste Lösung, weil das Datenbanksystem durch die fehlende Netzwerk-Funktionalität schnell und schlank ist. Mit Hilfe der Komponente *TClientDataSet* können Sie solche Stand-Alone-Datenbanken realisieren.

File-Share-Datenbank

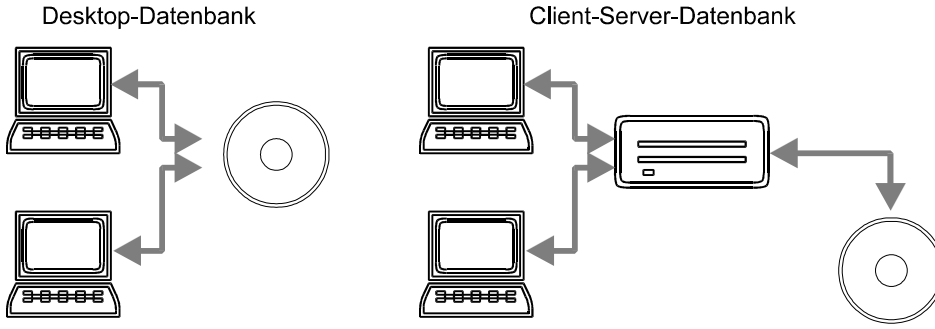
Moderne Netzwerke bieten die Möglichkeit, dass mehrere Anwender auf ein und dieselbe Datei zugreifen. Auf diese Weise ist es auch möglich, dass mit zwei Datenbankanwendungen auf dieselbe Datenbankdatei zugegriffen wird. Diese Version der *Desktop*-Datenbank nennt man *File-Share*-Datenbank, und damit ist schon ein echter Multiuser-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 entsprechend niedrig die Performance) des Netzwerks.

Zudem muss verhindert werden, dass zwei Benutzer gleichzeitig in dieselbe Datei schreiben, man realisiert dies durch eine so genannte Locking-Tabelle. Solange ein Benutzer in eine Datei schreibt, ist diese für alle anderen Benutzer gesperrt. Verständlicherweise ist diese Vorgehensweise nicht besonders effektiv und auch für eine hohe Zahl von Benutzern nicht geeignet.

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 Hardware-mäßig 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 5.) 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.

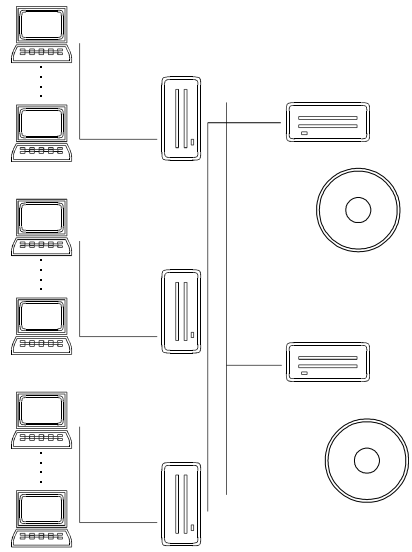
Bei einem *Client-Server*-System würde der Server die Anfrage selbst bearbeiten und dann rund 10 kbyte zum Arbeitsplatzrechner übertragen. Dies würde einer Beschleunigung um den Faktor 2 000 entsprechen und bei manchen Abfragen sind die Verhältnisse noch viel krasser.

Hinzu kommt, dass *Client-Server*-Systeme meist viel besser auf die Erfordernisse des Umgangs 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 über eine Millionen Euro an Lohn- und Lohnnebenkosten an. Daran gemessen sind die Mehrkosten für C-S-Systeme wirklich »Peanuts«.

2.1.3 Multi-Tier-Systeme

Ab Version 3 bietet Delphi auch die Möglichkeit, Multi-Tier-Systeme – also mehrschichtige Datenbankanwendungen – zu entwickeln. Bei diesen mehrschichtigen Systemen sind zwischen den Clients und den Datenbank-Servern die *Application-Server* installiert. Die nachfolgende Abbildung zeigt den grundsätzlichen Aufbau eines solchen Systems:



Warum Multi-Tier?

Nach einer Untersuchung der *Gartner Group* sind im Durchschnitt pro Client-Server-Anwendung 700 Clients im Einsatz. Auch wenn nicht alle Clients gleichzeitig auf den Server zugreifen, sind dies Größenordnungen, die den Server zum Nadelöhr des gesamten Systems machen.

Der Server wird durch Multi-Tier-Systeme schon allein dadurch entlastet, dass er weniger Clients und somit auch weniger Transaktionen verwalten muss. Zudem können viele der *Business-Rules* auf dem Application-Server implementiert werden, so dass fehlerhafte Anweisungen erst gar nicht zum Server gelangen. In manchen Fällen führen auch mehrere Anwender der gleichen Arbeitsgruppe dieselbe Abfrage aus, so dass die Daten nur einmal vom Server bezogen werden müssen. Mit Hilfe von *Cached Updates* können auch die Daten vom Server geladen, in der Arbeitsgruppe bearbeitet und nach einiger Zeit wieder auf dem Server aktualisiert werden.

Des Weiteren ist die Installation neuer Programme oder das Updaten bestehender Programme sehr aufwändig, wenn dies auf einer so großen Anzahl von Systemen erfolgen muss. Mit Hilfe der Multi-Tier-Technologie können *Thin Clients* erstellt werden, deren Größe in der Größenordnung von 100 bis 500 kbyte liegen (bei der Verwendung von Packages). Solche Clients können mit vertretbarem Zeitaufwand bei jedem Systemstart vom Application-Server heruntergeladen werden und müssen somit gar nicht auf allen Clients installiert werden.

Selbst eine Verbreitung der Clients über Modem wäre denkbar. Bei einer Übertragungsgeschwindigkeit von gemäßigten 56 800 kbit/s wäre eine Anwendung von 200 kbyte in einer halben Minute heruntergeladen. Das würde man dann wohl nicht bei jedem Systemstart tun, sondern immer nur dann, wenn eine neue Version verfügbar ist.

Wann Multi-Tier?

Mehrschichtige Anwendungen sind anspruchsvoller und auch aufwändiger zu programmieren als die klassischen, zweischichtigen Client-Server-Anwendungen. Für ein kleines Firmennetzwerk mit vielleicht fünf Arbeitsplätzen wäre ein solches Konzept um mehrere Größenordnungen überdimensioniert. In folgenden Fällen sollte man jedoch über die Erstellung einer mehrschichtigen Anwendung nachdenken:

- Die Zahl der Clients steigt über 100 oder mehrere Server arbeiten parallel, um einen ununterbrochenen Betrieb zu gewährleisten.
- Mehrere dezentrale, kleinere Netzwerke sind über ein WAN (*wide area network*) miteinander verbunden (Filialbetrieb). Bei solchen Systemen gilt es, Anzahl und Dauer der Serverzugriffe zu minimieren, weil dadurch Übertragungs-

kosten entstehen. In diesem Fall würde jedes der kleineren Netzwerke von einem Application-Server (Anwendungs-Server) versorgt.

- Viele Clients führen einen Fernzugriff (oder auch einen Offline-Zugriff) auf die Datenbank durch. So könnte beispielsweise eine Versicherungszentrale mit einigen hundert Vertretern verbunden sein. Diese benötigen meist nur einen bestimmten Satz an Informationen, die ihnen der Anwendungs-Server zur Verfügung stellt.

2.2 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.

2.2.1 Begriffe

Man kann nicht über relationale Datenbanken sprechen, ohne zuvor 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 Pascal 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 (Integer-Zahlen). 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 weiter kein Problem, es werden alle Nachnamen-*Attributes* als VARCHAR(20) definiert.

Später wird eine Änderung nötig; beispielsweise kommen Doppelnamen in Mode und die Nachnamen lauten 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 13.1 (es behandelt *InterBase SQL*) 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 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.

2.2.2 Keys (Schlüssel)

Im Zuge der so genannten »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, 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 sequenziell organisiert (intern 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 Kunden-

nummer oder eine Bestellnummer. Zwingend notwendig ist das (zumindest theoretisch) nicht. Eine Person könnte man auch durch die Kombination mehrerer Personendaten eindeutig identifizieren – es wären allerdings viele Personendaten.

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 natürlich 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«.

Zudem 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 mindestens zehnmal so groß.

Außerdem dürfen solche *Candidate Keys* nicht geändert werden. Wenn die betreffende Person mit dem Familienstand auch ihren Nachnamen wechselt, 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.

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 Firmenfusion in einer Tabelle die Kundennummern von Firma A den Kundennummern der Firma B zugeordnet würden. Jeder Kunde lässt sich sowohl anhand der einen als auch der anderen Kundennummer zweifelsfrei identifizieren, aber *Primary Key* kann nur eine sein.

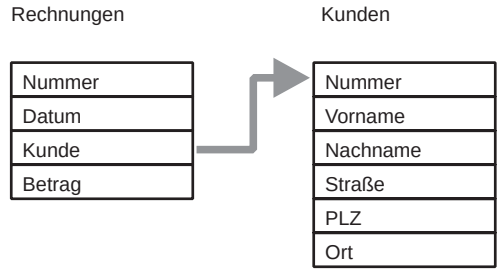
Es gilt auch zwischen Schlüsseln zu unterscheiden, die momentan (noch) *Candidate Key* sind (Vor- und Nachnamen, Geburtsdatum und Geburtsort), und solchen, die garantiert immer *Candidate Key* sein werden (also die durchlaufenden Nummern beispielsweise). Als *Primary Key* eignen sich nur letztere.

Alternate Key

Zusätzlich zu den Primärschlüsseln lassen sich auch *Alternate Keys*, auch *Secondary Keys* (Sekundärschlüssel), definieren. Bei einer Tabelle, in der Projekte gespeichert werden, wird jedes Projekt mit einer eindeutigen, meist durchlaufenden Nummer gekennzeichnet. Darüber hinaus könnte man jedoch mit einem Sekundärschlüssel sicherstellen, dass der Projektname ebenfalls nur ein einziges Mal verwendet wird.

Foreign Key

Gerade bei relationalen Datenbanken werden sehr viele Master-Detail-Verknüpfungen erstellt. Ein Beispiel dafür wäre die Rechnungstabelle und die Kundentabelle einer Firma:



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.

Für die Spalte *Kunde* würde hier ein *Foreign Key*, also ein Fremdschlüssel, eingerichtet. Dieser verweist auf eine Spalte einer anderen Tabelle (in Ausnahmefällen auch auf eine Spalte der gleichen Tabelle) und stellt über ihn die Verbindung zu dieser her. Der SQL-Befehl für die Tabelle *Rechnungen* lautet folgendermaßen:

```
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))
```

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

Referenzielle Integrität

Eine wesentliche Aufgabe der RDBMS ist die Sicherstellung der *referenziellen 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, dass 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* in der 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 nahe liegendste, kann aber zu einem Konflikt mit dem Datenschutzgesetz führen. Wie man sich daran halbwegs vorbeimogelt, werden wir uns gleich ansehen.
  - Als zweite Möglichkeit bleibt, gleich auch noch alle damit verbundenen Datensätze entsprechend zu ändern beziehungsweise zu löschen. Die referenzielle Integrität wäre damit sichergestellt.

    Aber bei den meisten Datenbanken würde eine solche kaskadierte Aktion das ganze System durcheinander bringen – beispielsweise dann, wenn über die Bestellungstabelle auch die Bilanz erstellt wird.

Referenzielle Integrität und der Datenschutz

Bestellt ein Kunde etwas bei einer Firma, dann willigt er in der Regel über die allgemeinen Geschäftsbedingungen darin ein, dass seine Daten gespeichert werden. Nun könnte der Kunde auf den Gedanken kommen, diese Einwilligung zu widerufen und um die Löschung der über ihn gespeicherten Daten ersuchen. (Das Problem stellt sich nicht nur bei Firmen, sondern beispielsweise auch bei Parteien, Vereinen und Gewerkschaften, nur dass man dann nicht von *Kunden* spricht.)

Nun werden für den Kundendatensatz in aller Regel Detaildatensätze vorhanden sein, die man so ohne weiteres nicht aus der Tabelle entfernen kann. Deshalb würde das Datenbanksystem die Löschung des Kunden verhindern. Was tun?

- In den meisten Fällen sind die Leute schon zufrieden, wenn sie keine Post mehr von der betreffenden Firma erhalten. Also ergänzt man die Tabelle um eine Spalte »gelöscht« und ändert alle Abfragen so, dass nur noch die »ungelöschten« Daten angezeigt werden. Die Sachbearbeiter sehen somit diesen Datensatz nicht mehr, an ihn werden keine Schreiben mehr verschickt, und dass der Datensatz tatsächlich noch gespeichert ist, wissen nur ein paar Leute aus der EDV.
- Man kann den Datensatz »leer« machen: Für Name, Straße und Ort werden leere Strings eingesetzt, lediglich die Kundennummer bleibt bestehen. Verständlicherweise hat man dann jedoch ein Problem, wenn man die Adresse doch mal wieder benötigt, beispielsweise weil die letzte Rechnung vom Konto des Kunden eingezogen wurde und dieser nun plötzlich die Sache zurückbuchen lässt. (Tipp: Backup-Bänder!)
- Was man nie und unter gar keinen Umständen jemals auch nur in Erwägung ziehen sollte, ist, eine Kundennummer irgendwann an einen anderen Kunden zu vergeben.

Indizes

Indizes und Schlüssel werden immer mal wieder durcheinander gebracht, schließlich wird für jeden Schlüssel ein Index aufgebaut – aber nicht alle Indizes sind Schlüssel. Ein Index ist ein Suchbaum, mit dessen Hilfe sich Datensätze schneller finden lassen. 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 sequenzielle 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 worden ist. Außerdem muss die Kundennummer ja nicht eine fortlaufende Nummer sein; das kann von Karteikarten übertragen und daher völlig durcheinander geraten sein. Es bleibt zunächst nichts anderes als die sequenzielle 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 100000):

- Ist die Kundennummer größer oder kleiner als 50000? Sie ist kleiner.
- Ist die Kundennummer größer oder kleiner als 25000? Sie ist kleiner.
- Ist die Kundennummer größer oder kleiner als 12500? 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 Datensatzes 12345, sucht anhand der Adresse diesen auf der Platte und gibt ihn aus. Dies alles erledigt das RDBMS selbstständig, für den Anwender spielt sich der Vorgang also »unsichtbar« ab. Ob eine Suche sequenziell oder mit einem Index erfolgt, lässt sich nur daraus erahnen, wie schnell das Ergebnis bereitsteht.

Um nach anderen Attributmengen (also Spalten oder Zusammenfassungen von Spalten) beschleunigt zu suchen, lassen sich auch für diese Attributmengen solche Indizes erstellen. Diese belegen zusätzlichen Platz auf der Festplatte und beschleunigen das Zustandekommen des Abfrageergebnisses.

Allerdings müssen solche Indizes auch gepflegt werden: Für jeden neuen Datensatz ist auch ein Eintrag in jeweils jedem Index erforderlich. Werden größere Mengen von Daten in eine Tabelle geschrieben, dann kann es sinnvoll sein, alle Indizes vorher zu löschen und anschließend neu aufzubauen.

2.2.3 Weitere Elemente relationaler Datenbanksysteme

Relationale Datenbanksysteme realisieren noch einige weitere Elemente, beispielsweise VIEWS, STORED PROCEDURES, Transaktionen oder detailliert einstellbare Zugriffsrechte.

Views

Bei allen (vernünftigen) Client-Server-Datenbanken lassen sich *Views*, also Ansichten definieren. Dabei handelt es sich um eine eingeschränkte oder zusammenfassende Sicht auf Tabellen. Meist handelt es sich dabei um umfangreiche JOINS, die man nicht immer wieder formulieren möchte.

Manchmal wird man auch VIEWS erstellen, damit von einer Tabelle nicht alle Daten sichtbar sind. 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*.

STORED PROCEDURES

STORED PROCEDURES bieten die Möglichkeit, komplexere Aktionen auf dem Datenbank-Server auszuführen. So sind innerhalb solcher Prozeduren Schleifen und Verzweigungen möglich. Manche STORED PROCEDURES liefern Werte und ähneln darin einer Abfrage oder einer VIEW, manche fügen Daten ein, oder ändern oder löschen Daten. Wie Pascal-Prozeduren können auch STORED PROCEDURES Parameter entgegennehmen.

Bei manchen Datenbanksystemen besteht die Möglichkeit von TRIGGERN: Wird in eine Tabelle ein Datensatz eingefügt, geändert oder gelöscht, dann wird davor und/oder danach eine Art STORED PROCEDURE aufgerufen, die hier zusätzliche Funktionalität bereitstellt – beispielsweise weitere Daten einfügt.

Transaktionen

Mit Hilfe von Transaktionen kann man eine Reihe von Aktionen zu einer Gruppe zusammenfassen, die entweder gemeinsam bestätigt oder gemeinsam verworfen werden. Beliebtes Beispiel dafür ist die doppelte Buchhaltung: Wird ein Konto belastet, muss derselbe Betrag einem anderen gutgeschrieben werden. Mit Hilfe von Transaktionen kann man nun gewährleisten, dass diese beiden Aktionen immer nur gemeinsam ausgeführt werden.

Transaktionen werden auch verwendet, um eine konsistente Datensicht zu garantieren: Müssen beispielsweise einige statistische Auswertungen durchgeführt werden, dann ist es sehr ungünstig, wenn sich dabei der Datenbestand ändert. Mit Hilfe von Transaktionen können auch solche Abfragen im laufenden Betrieb durchgeführt werden.

Zugriffsberechtigungen

Bei größeren Client-Server-Systemen kann für jede Tabelle detailliert eingestellt werden, welcher Benutzer Daten lesen, einfügen, ändern oder löschen darf. Ebenso kann festgelegt werden, wer zum Ausführen welcher STORED PROCEDURE berechtigt ist.

2.2.4 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 Tuplereihenfolge 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 Schlüsselattribut 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 interessiert, dem sei das Buch *Relationale Datenbanken* von *Hermann Sauer* empfohlen (Addison-Wesley, ISBN 3-8273-1381-3).

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 mal das folgende Bild:

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

Der Datensatzaufbau könnte der Mitarbeiterdatei 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-Name*). 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 Kundennummern vor, dann sind es streng genommen 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 zu-

künftig 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

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 alle übrigen Spalten nicht nur 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.

Momentan haben wir aber noch solche Abhängigkeiten:

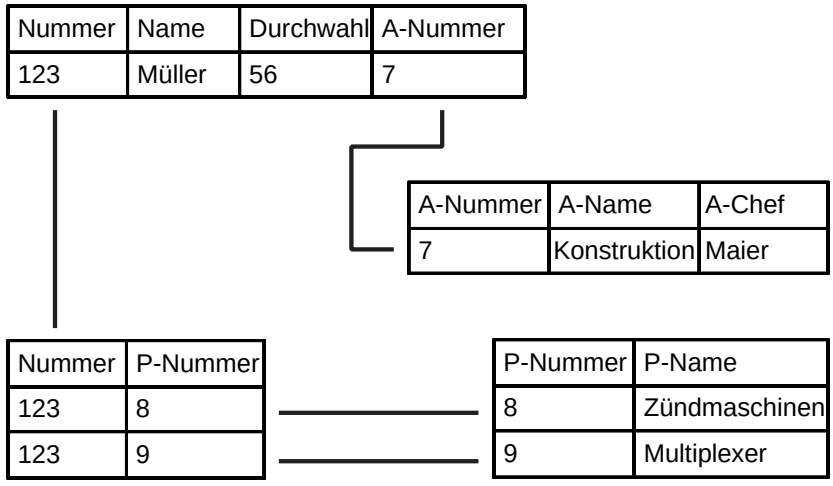
- Abteilungsnummer, Abteilungsname und Abteilungsleiter sind voneinander funktional abhängig. Abteilung *sieben* ist immer *Konstruktion* und hat immer den Chef *Maier* (umgekehrt ist das nicht zwingend; es könnte eine weitere Konstruktionsabteilung geben, welche dann auch eine andere Nummer hätte).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

Warum stören diese Abhängigkeiten? Normalerweise sind in einer solchen Tabelle mehrere Datensätze, beispielsweise 80 aus der Abteilung *sieben*, und jedes Mal 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 (und zeigen Sie mir mal die Sekretärin, welche das mittels einer SQL-Anweisung anstatt mit 80 manuellen Eingaben ändert ...).

Schwieriger wird es 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.

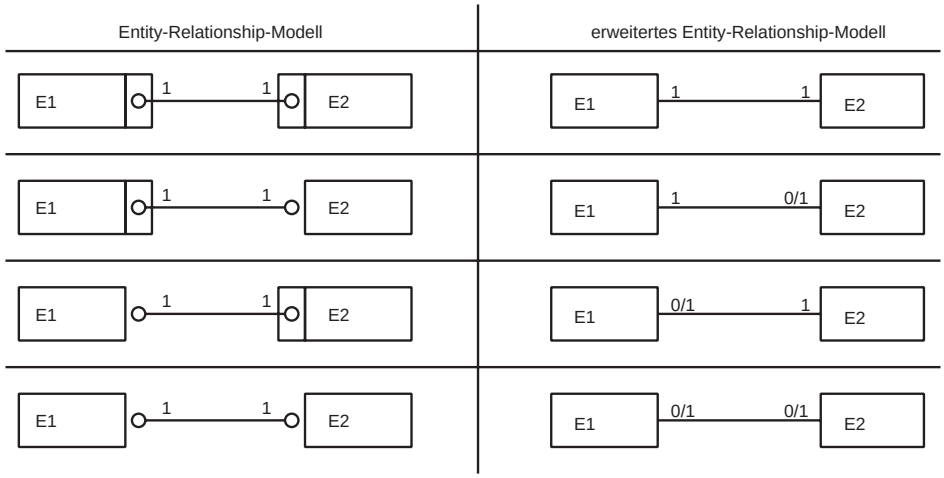
Die dritte Normalform

Um diese funktionalen Abhängigkeiten zu reduzieren, wird die ursprüngliche Tabelle einfach in mehrere kleinere Tabellen aufgeteilt:



2.2.5 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. Dies werden wir das erweiterte *Entity-Relationship-Modell* nennen.

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 die Projekte derzeit ruhen.)

Das erweiterte ER-Modell

Ob eine obligatorische Mitgliedschaft besteht, wurde nun beim herkömmlichen ER-Modell in Blockdiagrammen dargestellt: Ist der Kreis im Kästchen, ist die Mitgliedschaft obligatorisch, ist er außerhalb des Kästchens, ist die Mitgliedschaft optional.

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 Gefahr, dass man die Darstellungsformen von obligatorischer und nicht obligatorischer Mitgliedschaft durcheinander bringt. Zudem lässt sich diese Information auch außerhalb von Blockdiagrammen darstellen.

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 die *1 : N*-Beziehung, 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,
  - 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. (Gäbe es Wagen in der Tabelle, die keinem Mitarbeiter fest zugeordnet sind, dann hätten wir eine 0/1 : 0/1-Beziehung. Und die Möglichkeit, dass ein Mitarbeiter mehrere Dienstwagen hat, ignorieren wir mal lieber.)

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

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

0 / 1

1

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

- 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 überwiegen. 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 (die Namen in der folgenden Abbildung 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	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

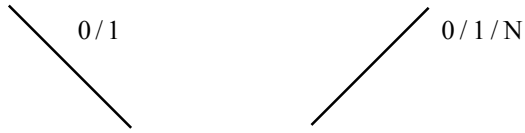
Dozent	Preis
7	3
3	5

An diesem Beispiel wird auch noch einmal deutlich, welchen Vorteil die Verwendung von (durchlaufenden) Nummern als Primärindex hat: Prinzipiell hätte man für die Preisträgertabelle auch Jahreszahl und Fachbereich verwenden können (so lange 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. Auch hier wird dann die Verknüpfung über eine eigene Relation hergestellt:

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



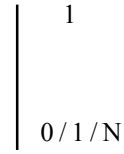
Dozent	Buch
2	3
2	4
3	5
5	27

Es wäre auch nicht weltfremd, dass mehrere Professoren gemeinsam ein Buch geschrieben haben, das würde dann eine *0/1/N : 0/1/N*-Beziehung, die man auch mit einer Verknüpfungstabelle realisieren würde.

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.

- Als Nächstes Beispiel sei die Auftrags-Kunden-Beziehung einer Firma genannt, wobei es sich streng genommen 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 potenzielle 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:

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



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

- 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.

2.3 Datenbanken und Delphi

Zu Zeiten von Delphi 1 war alles noch schön übersichtlich: Der Datenbankzugriff erfolgte mit der BDE und die Reports gestaltete man mit ReportSmith. Alternativen wurden allenfalls von Drittanbietern zur Verfügung gestellt. Im Laufe der bislang erschienenen Delphi-Versionen wurden dem Entwickler immer mehr Möglichkeiten zur Verfügung gestellt – damit wurde die Qual der Wahl aber auch immer größer:

- Nach wie vor gibt es die BDE (Borland Database Engine), das »aufgebohrte« Paradox-Datenbanksystem. Die BDE wird inwischen von Borland nicht mehr weiterentwickelt.

  Ich gehe jedoch davon aus, dass die BDE noch in vielen weiteren Delphi-Versionen vorhanden sein wird, die Komponenten von der Palettenseite *Beispiel* begleiten uns ja auch seit Delphi 1. Borland wird sich hüten, einen Großteil der Delphi-Entwickler zu verärgern.

  Paradox ist auch ein sehr schönes Datenbank-System – vorausgesetzt, man vermeidet den Multiuser-Betrieb. Und auch beim Zugriff auf Client-Server-Systeme schlägt sich die BDE sehr gut, wenn man die gängigen Fehler vermeidet. Über die BDE gelangt man auch an alle eingerichteten ODBC-Datenbanken – man braucht allerdings nicht glauben, dass ODBC dadurch schneller wird. Die Installation der BDE bleibt allerdings ein Problem, insbesondere, wenn die Erstinstallation über die Corel WordPerfect Suite erfolgt ist.

- Der von Borland derzeit favorisierte Weg geht über *TClientDataSet*. Diese Komponente wird nicht nur für mehrschichtige Datenbankanwendungen verwendet, man kann damit auch sehr schnelle Stand-Alone-Datenbanksysteme aufbauen (»MyBase«).

- Die InterBase-Komponenten (»IBX«) nutzen sehr schön die Möglichkeit von InterBase.

- Mit dbExpress hat man einen recht schlanken Zugriff auf InterBase und MySQL, in der Enterprise-Version zusätzlich auf Oracle und DB/2. Allerdings kann man hier mit einem leicht erhöhten Programmieraufwand rechnen.

- Und schließlich gibt es noch die ADO-Komponenten.

3 TDataSet, TField und TDataSource

Wir wollen uns nun mit drei ganz essenziellen Datenbankkomponenten beschäftigen:

- *TDataSet* finden Sie nicht in der Komponentenpalette, es handelt sich dabei um eine abstrakte Klasse, die Sie auch nicht direkt instantisieren können. *TDataSet* ist jedoch der Urahne (fast) aller Datenzugriffskomponenten.
- *TField* ist der Vorfahre aller Spaltenkomponenten. Für jede Spalte wird ein *TField*-Nachfahre instantisiert, entweder automatisch beim Öffnen der Datenmenge, oder Sie legen zur Entwurfszeit persistente *TField*-Instanzen an.
- *TDataSource* ist das Bindeglied zwischen *TDataSet* und den Datensteuerungskomponenten.

3.1 TDataSet

Die Komponente *TDataSet* ist der gemeinsame Vorfahre aller Datenmengenkomponenten. *TDataSet* ist von *TComponent* abgeleitet, alle Datenmengenkomponenten sind somit nicht visuell.

Auch wenn alle Datenmengenkomponenten von *TDataSet* abgeleitet sind, heißt das noch lange nicht, dass alle hier beschriebenen Eigenschaften und Ereignisse auch veröffentlicht sind und dass alle Methoden funktionieren. Gerade die *dbExpress*-Komponenten stellen diese Elemente nur sehr eingeschränkt zur Verfügung.

Navigieren in der Datenmenge

- First, Prior, Next, Last (Methoden)

```
procedure First;
procedure Prior;
procedure Next;
procedure Last;
```

Mit Hilfe dieser Methoden kann der Datenzeiger zum ersten (*First*), vorhergehenden (*Prior*), nächsten (*Next*) oder letzten (*Last*) Datensatz verschoben werden. Unidirektionale Datenmengen (dbExpress) unterstützen nur *First* und *Next*, der Aufruf von *Prior* oder *Last* würde eine Exception auslösen.

- BOF, EOF (Eigenschaft, öffentlich, nur Lesen)

```
property BOF: Boolean;
property EOF: Boolean;
```

Die Eigenschaft *BOF* (*EOF*) ist dann gleich *true*, wenn

- die Methode *First* (*Last*) aufgerufen wird,
- das Verschieben des Datenzeigers fehlschlägt, weil er schon auf dem ersten (letzten) Datensatz steht.

Diese beiden Eigenschaften werden vor allem für die Abbruchbedingungen von Schleifen benötigt.

```
while not Table1.EOF {BOF} do
begin
    ...
  Table1.Next {Prior};
end;
```

Vergessen Sie nicht, die Methode *Next* oder *Prior* aufzurufen (oder eine andere Methode, welche den Datenzeiger verschiebt), anderenfalls hätten Sie eine Endlosschleife. Das Gleiche würde passieren, wenn die Methode den Datenzeiger zum entgegengesetzten Ende der Datenmenge verschiebt.

- MoveBy (Methode)

```
function MoveBy(Distance: Integer): Integer;
```

Mit der Methode *MoveBy* kann man um die im Parameter angegebene Zahl von Datensätzen nach hinten navigieren.

Soll zum Dateianfang hin navigiert werden, dann muss ein negativer Parameter übergeben werden. Trifft diese Funktion auf Dateiende oder Dateianfang, dann wird an dieser Stelle gestoppt und die Eigenschaft *EOF* beziehungsweise *BOF* auf *true* gesetzt. Die Zahl der Datensätze, um die der Datenzeiger wirklich bewegt worden ist, wird als Funktionsergebnis zurückgegeben.

Bei unidirektionalen Datenmengen funktioniert *MoveBy* nur mit positiven *Distance*-Werten.

- FindFirst, FindPrior, FindNext, FindLast (Methoden)

```
function FindFirst: Boolean;
function FindPrior: Boolean;
function FindNext: Boolean;
function FindLast: Boolean;
```

Mit Hilfe dieser Methoden kann man in mit *OnFilterRecord* gefilterten Datenmengen navigiert, wenn bei jedem Aufruf einer dieser Methoden der gesamte Datenbestand neu durchgefiltert werden soll.

Greifen wir mit *TTable* auf die Tabelle *t_art* unserer Beispieldatenbank zu, die in der Spalte *nummer* die Werte von 1 bis 14 enthält. Diese Datenmenge wird mit der folgenden Ereignisbehandlungsroutine auf die Datensätze 11 und 12 beschränkt:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  Accept := ((Table1NUMMER.AsInteger > 10)
    and (Table1NUMMER.AsInteger < 13));
end;
```

Mit Hilfe des Debuggers schauen wir uns den Inhalt der Spalte *nummer* an. Bei der Verwendung der Methode *First* wird *OnFilterRecord* exakt 14-mal aufgerufen, für jeden Datensatz ein einziges Mal. Danach werden die Datensätze 11 und 12 angezeigt. Beim darauf folgenden Aufruf von *Next* wird *OnFilterRecord* nicht aufgerufen, bei noch einem Aufruf von *Next* wird *OnFilterRecord* für die Datensätze 13 oder 14 aufgerufen – es wird sicherheitshalber nochmals geschaut, ob inzwischen weitere Datensätze der Filterbedingung – die sich ja geändert haben kann – entsprechen.

Nun die ganze Geschichte mit *FindFirst*: Das Ereignis *OnFilterRecord* wird nacheinander in der folgenden Reihenfolge aufgerufen:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
12, 13, 14,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Um 14 Datensätze zu filtern, wird *OnFilterRecord* 35-mal aufgerufen. Wir stehen nun auf Datensatz 11 und rufen *FindNext* auf. Nun wird *OnFilterRecord* für die folgenden Datensätze aufgerufen:

```
12, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
13, 14,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Ein weiterer Aufruf von *FindNext* beschäftigt *OnFilterRecord* dann nur mit den Datensätzen 13 und 14.

Aus welchen Gründen mit diesen Methoden der Datenbestand manchmal mehrfach durch die *OnFilterRecord*-Ereignisbehandlungsroutine geschickt wird, ist mir zugegebenermaßen völlig rätselhaft.

- Locate (Methode)

```
function Locate(const KeyFields: string;
  const KeyValues: Variant; Options: TLocateOptions): Boolean;
```

Die Methode *Locate* wird verwendet, um einen Datensatz anhand seiner Feldwerte zu finden. Das folgende Beispiel setzt den Datenzeiger auf den ersten Datensatz, dessen Feld *Vorname* den Wert *Bert* und dessen *Nachname* den Wert *Heller* hat.

```
Table1.Locate('Vorname; Nachname', VarArrayOf(['Bert', 'Heller']), []);
```

Mit Hilfe des Parameters *Options* kann dafür gesorgt werden, dass die Groß- und Kleinschreibung bei der Suche nicht beachtet wird und dass ein Datensatz auch dann gefunden wird, wenn er nur teilweise mit den angegebenen Werten übereinstimmt.

```
TLocateOptions = set of (loCaseInsensitive, loPartialKey);
```

Hätten wir *loPartialKey* gesetzt, dann hätte *Locate* auch einen *Bert Hellermann* gefunden. Mit dem Funktionsergebnis erfahren wir, ob die Suche erfolgreich war.

Soll nur in einer einzelnen Spalte gesucht werden, dann muss keine Typenumwandlung erfolgen:

```
Table1.Locate('bezeichnung', 'Monitor', []);
```

- GetBookmark, GotoBookmark, FreeBookmark, CompareBookmarks (Methoden)

```
TBookmark = Pointer;
function GetBookmark: TBookmark;
procedure GotoBookmark(Bookmark: TBookmark);
procedure FreeBookmark(Bookmark: TBookmark);
function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;
```

Mit Hilfe der erwähnten Methoden lassen sich in der Datenmenge Lesezeichen setzen, die später wieder aufgerufen werden können.

```
private
  BM1: Pointer;
  BM2: Pointer;
```

Deklarieren sie vor der Verwendung entsprechend viele *Pointer*-Variablen.

```
procedure TForm1.Setzen2Click(Sender: TObject);
begin
  BM1 := Table1.GetBookmark;
end;
```

Mit *GetBookmark* wird die aktuelle Position des Satzzeigers in der Variablen gespeichert.

```
procedure TForm1.Aufrufen2Click(Sender: TObject);
begin
  Table1.GotoBookmark(BM1);
end;
```

Mit *GotoBookmark* kann man an die gespeicherte Stelle zurückspringen. *FreeBookmark* löscht ein Lesezeichen, mit *CompareBookmark* kann festgestellt werden, ob zwei Lesezeichen auf denselben Datensatz zeigen.

Unidirektionale Datenmengen unterstützen keine Lesezeichen.

- Bookmark (Eigenschaft)

```
property Bookmark: string;
```

Mittels der Eigenschaft *Bookmark* kann man die aktuelle Position des Satzzeigers in einem String speichern. Wenn man diesen String wieder der Eigenschaft *Bookmark* zuweist, bewegt man den Satzzeiger wieder an die betreffende Stelle.

Der String beginnt gewöhnlich mit einem (ASCII-)Null-Zeichen, sie können ihn somit nicht mit beispielsweise *TEdit* anzeigen.

Unidirektionale Datenmengen unterstützen keine Lesezeichen.

- AfterScroll, BeforeScroll (Ereignisse)

```
property AfterScroll(DataSet: TDataSet);
property BeforeScroll(DataSet: TDataSet);
```

Diese Ereignisse treten vor beziehungsweise nach dem Verschieben des Datenzeigers auf. Beachten Sie in diesem Zusammenhang, dass bei einigen *TDataSet*-Ereignissen das Präfix *On* entfällt.

Ändern der Daten

- Append, AppendRecord (Methoden)

```
procedure Append;
procedure AppendRecord(const Values: array of const);
```

Mit *Append* wird hinter den letzten Eintrag der Tabelle gesprungen und diese in den Eingabe-Modus versetzt; die Eingabe muss mit *Post* (oder Methoden, welche *Post* aufrufen) bestätigt oder mit *Cancel* abgebrochen werden. Wird für die Tabelle ein Index verwendet, dann wird der Datensatz nach dem Aufruf von *Post* an der richtigen Position eingefügt.

Um neue Werte einzugeben, werden der Eigenschaft *Fields* die entsprechenden Werte zugewiesen; werden keine Werte zugewiesen, dann behalten diese den Wert NULL.

```
with Table1 do
begin
  Append;
  Fields[0].AsString := 'Karo';
  Fields[1].AsString := 'Ass';
  Post;
end;
```

Der Methode *AppendRecord* werden die einzugebenden Werte als Parameter übergeben, die Bestätigung mittels *Post* kann unterbleiben. Der neue Datensatz wird an die Tabelle angehängt.

```
Table1.AppendRecord('Karo', 'Ass');
```

- Insert, InsertRecord (Methoden)

```
procedure Insert;
procedure InsertRecord(const Values: array of const);
```

Die Methode *Insert* ähnelt *Append*, springt aber im Gegensatz dazu nicht an das Ende der Tabelle, sondern fügt den neuen Datensatz an der aktuellen Datenzeigerposition ein.

Wird für die Tabelle ein Index verwendet, dann wird der neue Datensatz an der richtigen Position eingefügt. Mit *InsertRecord* werden die übergebenen Parameter als neuer Datensatz an aktueller Datenzeigerposition eingefügt.

- Edit (Methode)

```
procedure Edit;
```

Die Methode *Edit* versetzt die Datenmenge in den Status *dsEdit*. Anschließend können die Daten des aktuellen Datensatzes geändert werden, abschließend muss die Änderung mit *Post* bestätigt oder mit *Cancel* verworfen werden.

```
with Table1 do
begin
  Edit;
  FieldByName('Feld1').AsString := 'Wert 1';
  FieldByName('Feld2').AsString := 'Wert 2';
  Post;
end;
```

- Delete (Methode)

```
procedure Delete;
```

Die Methode *Delete* löscht den aktuellen Datensatz. Dies muss nicht mit *Post* bestätigt werden.

- Post, Cancel (Methoden)

```
procedure Post;
procedure Cancel;
```

Mit der Methode *Post* werden die vorgenommenen Änderungen bestätigt, mit *Cancel* werden sie verworfen. Die Datenmenge hat danach den Status *dsBrowse*.

Beachten Sie bitte, dass bei Desktop-Datenbanken beim Aufruf der Methode *Post* die Änderungen nicht auf die Festplatte, sondern nur in den Speicher geschrieben werden. Möchten Sie sicherstellen, dass die Änderungen auch nach einem Absturz der Rechners nicht verloren sind, müssen Sie jeweils die *TBDEDataSet*-Methode *FlushBuffers* aufrufen, was beispielsweise in der *AfterPost*-Ereignisbehandlungsroutine erfolgen kann.

```
procedure TForm1.Table1AfterPost(DataSet: TDataSet);
begin
  Table1.FlushBuffers;
end;
```

- ClearFields (Methode)

```
procedure ClearFields;
```

Mit *ClearFields* werden alle Einträge des aktuellen Datensatzes auf NULL gesetzt, ohne jedoch den Datensatz selbst zu löschen.

- Diverse Ereignisse (Ereignisse)

```
property AfterCancel(DataSet: TDataSet)
property AfterDelete(DataSet: TDataSet)
property AfterEdit(DataSet: TDataSet)
property AfterInsert(DataSet: TDataSet)
property AfterPost(DataSet: TDataSet)

property BeforeCancel(DataSet: TDataSet)
property BeforeDelete(DataSet: TDataSet)
property BeforeEdit(DataSet: TDataSet)
property BeforeInsert(DataSet: TDataSet)
property BeforePost(DataSet: TDataSet)

property OnNewRecord(DataSet: TDataSet)
```

Die erwähnten Ereignisse treten vor beziehungsweise nach den entsprechenden Methoden auf. Beachten Sie in diesem Zusammenhang, dass bei einigen *TDataSet*-Ereignissen das Präfix *On* entfällt.

In den entsprechenden Ereignisbehandlungsmethoden kann man nun beispielsweise verhindern, dass die Datenmenge in den Zustand *dsEdit* versetzt wird, indem man eine Exception auslöst.

```
procedure TForm1.Table1BeforeEdit(DataSet: TDataset);
begin
  Raise EDatabaseError.Create
    ('Datenmenge darf nicht geändert werden');
end;
```

Ein für den Anwender ähnliches Verhalten erreicht man, indem man dem Ereignis *OnAfterEdit* folgende Anweisungen zuweist:

```
procedure TForm1.Table1AfterEdit(DataSet: TDataset);
begin
  ShowMessage('Datenmenge kann nicht editiert werden');
  Table1.Cancel;
end;
```

Das Ereignis *OnNewRecord* tritt sowohl bei *Append*- als auch bei *Insert*-Aktionen auf, und zwar nach *BeforeInsert* und vor *AfterInsert*.

- OnDeleteError, OnEditError, OnPostError (Ereignisse)

```
property OnDeleteError(DataSet: TDataSet; E: EDatabaseError;
  var Action: TDataAction);
property OnEditError(DataSet: TDataSet; E: EDatabaseError;
  var Action: TDataAction);
property OnPostError(DataSet: TDataSet; E: EDatabaseError;
  var Action: TDataAction);
```

Tritt bei den dazugehörenden Aktionen ein Fehler auf, dann werden die erwähnten Ereignisse aufgerufen. Mittels des Variablenparameters *Action* kann bestimmt werden, wie auf den Fehler reagiert werden soll.

- *daFail:* Die Aktion wird abgebrochen, eine Fehlermeldung wird angezeigt.
- *daAbort:* Die Aktion wird ohne Fehlermeldung abgebrochen.
- *daRetry:* Die Aktion wird wiederholt – vorher sollte jedoch versucht werden, den Fehler zu beheben.

- UpdateStatus (Eigenschaft)

```
function UpdateStatus: (usUnmodified, usModified,
  usInserted, usDeleted);
```

Mit der Eigenschaft UpdateStatus, die nicht von allen Datenmengenkomponenten unterstützt wird, kann ermittelt werden, ob – und wenn ja, wie – der aktuelle Datensatz verändert worden ist.

- Constraints (Eigenschaft, veröffentlicht)

```
property Constraints: TCheckConstraints;
```

SQL-Server erlauben umfangreiche Gültigkeitsprüfungen der zu speichernden Daten. Werden Dateneingaben jedoch nicht gleich zum Server geschickt, weil *Cached Updates* oder *TClientDataSet* verwendet wird, dann sollten die Gültigkeitsprüfungen schon vom Client durchgeführt werden, um hier spätere Probleme zu vermeiden.

Gültigkeitsprüfungen, die nur eine Spalte betreffen – beispielsweise ein beschränkter Wertebereich bei Zahlen –, können mit den Eigenschaften des jeweiligen *TField*-Nachfolgers realisiert werden. Mit Hilfe der Eigenschaft *Constraints* können Gültigkeitsprüfungen vorgenommen werden, die mehrere Spalten umfassen.

TCheckConstrains ist eine Kollektion von *TCheckConstraint*-Elementen. *TCheckConstraint* implementiert unter anderem die folgenden Eigenschaften:

- In *CustomConstraint* wird die Beschränkung in SQL-Syntax formuliert, beispielsweise:

```
(nummer < 10) OR (nummer2 > 10)
```

- Beschränkungen, die von einem Datenbank-Server oder einem Data-Dictionary importiert werden, findet man in der Eigenschaft *ImportedConstraint*.
- Wird gegen eine Beschränkung verstoßen, dann wird eine *EDBEngineError*-Exception ausgelöst, welcher der in *ErrorMessage* gespeicherte Text zugewiesen wird.

Status der Datenmenge

- Active (Ereignis, veröffentlicht)

```
property Active: boolean;
```

Mit der Eigenschaft *Active* kann die Datenmenge geöffnet und geschlossen werden.

- Open, Close (Methoden, TDataSet)

```
procedure Open;
procedure Close;
```

  Mit *Open* wird die Datenmenge geöffnet, mit *Close* wird sie geschlossen.

- AfterClose, AfterOpen, BeforeClose, BeforeOpen (Ereignisse)

```
property AfterClose(DataSet: TDataSet)
property AfterOpen(DataSet: TDataSet)
property BeforeClose(DataSet: TDataSet)
property BeforeOpen(DataSet: TDataSet)
```

  Diese Ereignisse treten vor beziehungsweise nach dem Schließen beziehungsweise Öffnen der Datenmenge auf.

- State (Eigenschaft, öffentlich, nur Lesen)

```
property State: TDataSetState;
```

  Die Eigenschaft *State* zeigt den Status der Datenmenge an und kann folgende Werte annehmen:

  - *dsInactive*: Die Datenmenge ist nicht aktiv, auf ihre Daten kann nicht zugegriffen werden.
  - *dsBrowse*: Die Daten können angezeigt, jedoch nicht geändert werden. Solange keine anderen Operationen durchgeführt werden, hat eine Datenmenge diesen Status.
  - *dsEdit*: Der aktuelle Datensatz kann geändert werden. Diesen Status erhält man mit dem direkten oder indirekten Aufruf von *Edit*.
  - *dsInsert*: Nach dem direkten oder indirekten Aufruf von *Insert* oder *Append* kann ein neuer Datensatz ein- oder angefügt werden.
  - *dsSetKey*: Nur bei *TTable*. Die Datensatzsuche ist aktiviert oder eine *SetRange*-Operation wird durchgeführt. Es können keine Datensätze geändert oder eingefügt werden.
  - *dsCalcFields*: Ein *OnCalcFields*-Ereignis wurde ausgelöst, es gibt somit Felder, deren Inhalt nicht verändert werden kann.
  - *dsFilter*: Es wird eine gefilterte Datenmenge angezeigt. Im Gegensatz zu den Angaben in der Online-Hilfe können Änderungen vorgenommen und neue Datensätze eingefügt werden, es kann jedoch sein, dass diese nach der *Post*-Anweisung nicht mehr angezeigt werden.

Im Zusammenhang mit *CachedUpdates* sind folgende Stati möglich:

- *dsNewValue*: Temporäre, interne Statuszuweisung, die anzeigt, dass gerade ein Zugriff auf die Eigenschaft *TField.NewValue* erfolgt.
- *dsOldValue*: Temporäre, interne Statuszuweisung, die anzeigt, dass gerade ein Zugriff auf die Eigenschaft *TField.OldValue* erfolgt.
- *dsCurValue*: Temporäre, interne Statuszuweisung, die anzeigt, dass gerade ein Zugriff auf die Eigenschaft *TField.CurValue* erfolgt.

- DisableControls, EnableControls, ControlsDisabled (Methoden)

```
procedure DisableControls;
procedure EnableControls;
function ControlsDisabled: Boolean;
```

Werden die Daten einer Datenmenge geändert, so wird im Regelfall laufend die Anzeige aktualisiert. Werden viele Daten in einer Schleife eingefügt, verlangsamt dies den Vorgang deutlich. Mit *DisableControls* kann deshalb diese Aktualisierung abgeschaltet und mit *EnableControls* anschließend wieder angeschaltet werden.

Mit *ControlsDisabled* kann ermittelt werden, ob die Aktualisierung abgeschaltet ist.

Zugriff auf die Daten

- Fields (Eigenschaft, öffentlich)

```
property Fields[Index: Integer]: TField ;
```

Mit der Array-Eigenschaft *Fields* kann auf jedes Feld der Datenmenge zugegriffen werden.

Bei einem Verändern der Tabellenstruktur kann sich jedoch die Reihenfolge der Felder verändern. Verwenden Sie deshalb besser *FieldByName*.

Das Objekt *TField* wird in Kapitel 3.2 besprochen.

- FieldByName, FindField (Methoden)

```
function FieldByName(const FieldName: string): TField;
function FindField(const FieldName: string): TField;
```

Die Methode *FieldByName* greift über den Feldnamen auf das Feld zu – solange das betreffende Feld noch vorhanden ist, kommt es bei einer Änderung der Tabellenstruktur zu keinen Schwierigkeiten. Ist der als Parameter übergebene Spaltenname nicht vorhanden, dann löst *FieldByName* eine Exception aus, während *FindField* den Wert *nil* zurückgibt.

```
Table1.FieldByName('Vorname').AsString := 'Patty';
```

Das Objekt *TField* wird in Kapitel 3.2 besprochen.

- FieldValues (Eigenschaft, öffentlich)

```
property FieldValues[const FieldName: string]: Variant;
```

Mit besonders wenig Schreibaufwand kann mittels der Array-Eigenschaft *FieldValues* auf die Feldinhalte zugegriffen werden: Zum einen hat diese Eigenschaft den Datentyp *Variant*, so dass ohne weitere Konvertierung die Werte zugewiesen werden können. Zum anderen ist diese Eigenschaft die Default-Eigenschaft von *TDataSet*, sie muss also gar nicht erwähnt werden. Um dem Feld *Vorname* einen Wert zuzuweisen, formuliert man einfach:

```
Table1['Vorname'] := 'Anja';
```

- Lookup (Methode)

```
function Lookup(const KeyFields: string; const KeyValues: Variant;
  const ResultFields: string): Variant;
```

Mit *Lookup* kann der Inhalt eines bestimmten Datensatzes ermittelt werden, ohne den Satzzeiger zu bewegen. Der gesuchte Datensatz wird mit *KeyFields* und *KeyValues* spezifiziert (siehe *Locate*). Mit *ResultFields* gibt man an, welche Felder in die Ergebnismenge geschrieben werden sollen – mehrere Felder sind dabei durch Semikola zu trennen.

Das folgende Beispiel zeigt, wie ein Varianten-Record zerlegt und in einen String geschrieben wird (würde nur eine Spalte zurückgegeben, dann wäre das Ergebnis direkt mit *string* zuweisungskompatibel).

```
var
  v: Variant;
begin
  v := Query1.Lookup('Nummer', '30', 'Preis;Hersteller');
  Caption := string(VarArrayGet(v, [0])) + ';'
    + string(VarArrayGet(v, [1]));
```

- FieldDefs (öffentlich)

```
property FieldDefs: TFieldDefs;
```

Mit Hilfe von *FieldDefs* können die Metadaten einer Tabelle ermittelt und verändert werden. Diese Eigenschaft wird vor allem dazu benötigt, um zur Laufzeit Tabellen zu erstellen. *TFieldDefs* wird in einem eigenen Abschnitt besprochen.

- DefaultFields (Eigenschaft, öffentlich, nur Lesen)

```
property DefaultFields Boolean;
```

Mit *DefaultFields* kann ermittelt werden, ob *TField*-Instanzen beim Öffnen der Datenmenge angelegt werden müssen (*true*), oder ob es persistente *TField*-Instanzen gibt (*false*).

- CanModify, Modified (Eigenschaften, öffentlich, nur Lesen)

```
property CanModify: Boolean;
property Modified: Boolean;
```

Mit Hilfe der Eigenschaft *CanModify* kann festgestellt werden, ob die Datenmenge geändert werden kann. Sie wird beispielsweise auch dann den Wert *false* haben, wenn bei SQL-JOINS die *TQuery*-Eigenschaft *RequestLive* auf *true* gesetzt wird. Vorsicht: Diese Eigenschaft ermittelt nicht, ob Sie die Schreibberechtigung auf dem betreffenden Datenbank-Server haben.

Die Eigenschaft *Modified* ist gleich *true*, wenn der aktuelle Datensatz geändert, aber *Post* oder *Cancel* noch nicht aufgerufen wurden.

- FieldCount, RecordCount, RecNo, IsSequenced
  (Eigenschaften, öffentlich, nur Lesen)

```
property FieldCount: Integer;
property RecordCount: Integer;
property RecNo: Integer;
function IsSequenced: Boolean;
```

Mit Hilfe dieser Eigenschaften können die Zahl der Spalten (*FieldCount*), die Zahl der Datensätze (*RecordCount*) und die Nummer des aktuellen Datensatzes (*RecNo*) ermittelt werden.

```
Label1.Caption := 'Datensatz ' + IntToStr(Table1.RecNo)
  + ' von ' + IntToStr(Table1.RecordCount);
```

Durch die Zuweisung von *RecNo* kann der Satzzeiger auf den betreffenden Datensatz gesetzt werden. *RecNo* wird nur dann unterstützt, wenn *IsSequenced* den Wert *true* hat.

- IsEmpty (Methode)

```
function IsEmpty: Boolean;
```

IsEmpty gibt *true* zurück, wenn die Datenmenge leer ist.

- GetFieldNames (Methode, TDataSet)

```
procedure GetFieldNames(List: TStrings);
```

Die Methode *GetFieldNames* ermittelt die Spaltennamen einer Tabelle und schreibt sie in die vorgegebene String-Liste.

```
Table1.GetFieldNames(ListBox1.Items);
```

- AutoCalcFields (Eigenschaft, veröffentlicht), OnCalcFields (Ereignis)

```
property AutoCalcFields: Boolean;
property OnCalcFields(DataSet: TDataSet)
```

Um einem berechneten Feld einen Wert zuzuweisen, wird das Ereignis *OnCalcFields* verwendet. Mit *AutoCalcFields* kann die Aufrufhäufigkeit des Ereignisses verändert werden, siehe Online-Hilfe.

Filtern der Datenmenge

- Filter, Filtered, FilterOptions (veröffentlicht)

```
property Filter: string;
property Filtered: Boolean;
property FilterOptions:
  set of (foCaseInsensitive, foNoPartialCompare);;
```

Eine Datenmenge kann mit der Eigenschaft *Filter* und/oder mit dem Ereignis *OnFilterRecord* gefiltert werden. Damit diese Filter aktiv sind, muss die Eigenschaft *Filtered* auf *true* gesetzt werden.

Um mit der Eigenschaft *Filter* eine Datenmenge zu filtern, muss dieser Eigenschaft eine Filteranweisung zugewiesen werden, welche an die SQL-Syntax angelehnt ist. So sind dabei auch logische Verknüpfungen und/oder Jokerzeichen möglich:

```
(name='v*') OR (name='d*')
```

Mit dieser Anweisung würde man die angezeigten Datensätze auf solche beschränken, deren Werte im Feld *name* mit *d* oder *v* beginnen. Leider müssen String-Konstanten hier mit einfachen Anführungszeichen eingegeben werden, so dass eine Zuweisung zur Laufzeit mitunter ein wenig problematisch ist:

```
Table1.Filter := '(name=' + #39 + 'v*' + #39 + ') OR (Name='
  + #39 + 'd*' + #39 + ')';
```

Mit Hilfe von *FilterOptions* kann spezifiziert werden, ob Groß- und Kleinschreibung bei Strings unterbleiben soll (*foCaseInsensitive*), und/oder ob *-Zeichen

als literale Zeichen statt als Jokerzeichen verwendet werden sollen (*foNoPartialCompare*).

Im Regelfall wird die Datenmenge lokal von der betreffenden Datenmengenkomponente gefiltert. Eine Ausnahme macht *TTable* im Zusammenhang mit SQL-Datenbank-Servern: Weil es in der Regel sehr viel Zeit benötigen würde, den gesamten Datenbestand auf den Client herunterzuladen, wird hier die Filter-Anweisung in eine SQL-Anweisung übersetzt und vom Server ausgeführt.

Im Gegensatz dazu wird bei *TQuery* der durch die SQL-Anweisung spezifizierte Datenbestand auf den Client heruntergeladen, damit *TQuery* die *Filter*-Anweisung dann lokal ausführen kann. Aus Performance-Gründen sollte der Einsatz der Eigenschaft *Filter* im Zusammenhang mit SQL-Datenbank-Servern und *TQuery* besser unterbleiben.

- OnFilterRecord (Ereignis)

```
property OnFilterRecord(DataSet: TDataSet; var Accept: Boolean);
```

Das Ereignis *OnFilterRecord* wird für jeden Datensatz aufgerufen. Wird der Variablenparameter *Accept* auf *true* gesetzt, dann wird der Datensatz aufgenommen, wird er auf *false* gesetzt, unterbleibt dies.

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
  var Accept:Boolean);
begin
  if DataSet.FieldByName('Nachnamen').AsString = 'Rasemann'
    then Accept := true
    else Accept := false;
end;
```

Aktualisieren der Datenmenge

- Refresh (Methode), BeforeRefresh, AfterRefresh (Ereignisse)

```
procedure Refresh;
property BeforeRefresh(DataSet: TDataSet);
property AfterRefresh(DataSet: TDataSet);
```

Mit *Refesh* aktualisieren Sie die Datenmenge. Davor wird das Ereignis *BeforeRefresh* aufgerufen, anschließend das Ereignis *AfterRefresh*.

Bei Komponenten wie *TQuery* führt *Refresh* nicht zum gewünschten Erfolg, weil hier die Ergebnismenge auf dem Datenbank-Server neu aufgebaut werden muss. Dazu muss *TQuery* geschlossen und neu geöffnet werden.

Bei den unidirektionalen Datenmengen sorgt der Aufruf von *Refresh* dafür, dass die Datenmenge geschlossen und neu geöffnet wird:

```
procedure TCustomSQLDataSet.InternalRefresh;
begin
  SetState(dsInactive);
  CloseCursor;
  OpenCursor(False);
end;
```

- IsUnidirectional (Eigenschaft, öffentlich, nur Lesen)

```
property IsUnidirectional: Boolean;
```

IsUnidirectional gibt *true* zurück, wenn es sich um eine unidirektionale Datenmenge handelt. Viele Elemente von *TDataSet* können dann nicht verwendet werden.

TFieldDefs

Das Objekt *TFieldDefs* stellt eine Liste von *TFieldDef*-Objekten zur Verfügung, welche die Felder einer Datenmenge repräsentieren. *TFieldDefs* wird vor allem benötigt, um zur Laufzeit Tabellen zu erstellen. Das folgende Beispiel zeigt die prinzipielle Vorgehensweise:

```
procedure TForm1.N1Tabelleerstellen1Click(Sender: TObject);
begin
  with Table1 do
  begin
    Active := false;
    DataBaseName := 'test';
    TableName := 'test_1.db';
    TableType := ttParadox;
    with FieldDefs do
    begin
      Clear;
      Add('nummer', ftInteger, 0, false);
      Add('name', ftString, 20, false);
    end;
    CreateTable;
    Open;
  end; {with table1 do}
end; {procedure TForm1.N1Tabelleerstellen1Click}
```

TFieldDefs ist von *TObject* abgeleitet.

- Items (Eigenschaft, öffentlich)

```
property Items[Index: Integer]: TFieldDef; default;
```

Mit der Array-Eigenschaft *Items* kann auf die einzelnen *TFieldDef*-Einträge zugegriffen werden. Das Objekt *TFieldDef* ist im nächsten Abschnitt beschrieben.

- Count (Eigenschaft, öffentlich, nur Lesen)

```
property Count: Integer;
```

Mit *Count* kann die Anzahl der Einträge ermittelt werden.

- Add, AddFieldDef (Methoden)

```
procedure Add(const Name: string; DataType: TFieldType;
  Size: Word; Required: Boolean);
function AddFieldDef: TFieldDef;
```

Mit diesen beiden Methoden fügen Sie jeweils einen neuen Eintrag in die Liste ein. Während bei *Add* die *TFieldDef*-Eigenschaften als Parameter übergeben werden, liefert *AddFieldDef* einen Zeiger auf das neu erstellte *TFieldDef*-Objekt zurück, mit dessen Hilfe dann die einzelnen Eigenschaften gesetzt werden können.

```
Table1.FieldDefs.Add('name', ftString, 20, false);
```

```
var
  aFieldDef: TFieldDef;
begin
  aFieldDef := Table1.FieldDefs.AddFieldDef;
  aFieldDef.Name := 'name';
  aFieldDef.DataType := ftString;
  aFieldDef.Size := 20;
```

- Clear (Methode)

```
procedure Clear;
```

Mit *Clear* werden alle Einträge aus der Liste gelöscht.

- Find, IndexOf (Methoden)

```
function Find(const Name: string): TFieldDef;
function IndexOf(const AName: string): Integer;
```

Mittels dieser Methoden kann anhand des Feldnamens ein Zeiger auf den *TFieldDef*-Eintrag oder dessen Index ermittelt werden.

TFieldDef

Das Objekt *TFieldDef* repräsentiert die Definition eines Feldes einer Datenmenge. *TFieldDef* ist von *TObject* abgeleitet.

- Name (Eigenschaft, öffentlich)

```
property Name: string;
```

  Die Eigenschaft *Name* repräsentiert den Feldnamen.

- DataType, Size, Precision (Eigenschaften, öffentlich)

```
property DataType: (ftUnknown, ftString, ftSmallint, ftInteger,
  ftWord, ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime,
  ftDateTime, ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo,
  ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, ftTypedBinary,
  ftCursor, ftFixedChar, ftWideString, ftLargeint, ftADT, ftArray,
  ftReference, ftDataSet);
property Size: Integer;
property Precision: Integer;
```

  Mit *DataType* wird der Feldtyp festgelegt, mit *Size* – so benötigt – die Größe des Feldes. Beachten Sie bitte beim Erstellen neuer Tabellen, dass nicht alle Datenbanksysteme alle Feldtypen unterstützen.

  Beim Typ *ftBCD* werden mit *Precision* die Zahl der Stellen und mit *Size* die Nachkommastellen festgelegt.

- Required, Attributes, InternalCalcField (Eigenschaften, öffentlich)

```
property Required: Boolean;
property Attributes:
  set of (faHiddenCol, faReadonly, faRequired, faLink, faUnNamed);
property InternalCalcField: Boolean;
```

  Bei einem NOT NULL-Feld hat *Required* den Wert *true* und die Mengeneigenschaft *Attributes* enthält das Element *faRequired*. Das Element *faReadOnly* zeigt an, dass der betreffende Feldwert nicht geändert werden kann – beispielsweise bei berechneten Feldern.

  Bei berechneten Feldern hat *InternalCalcField* den Wert *true*.

- Attributes (Eigenschaft, veröffentlicht)

```
property Attributes: set of (faHiddenCol, faReadonly,
  faRequired, faLink, faUnNamed, faFixed);
```

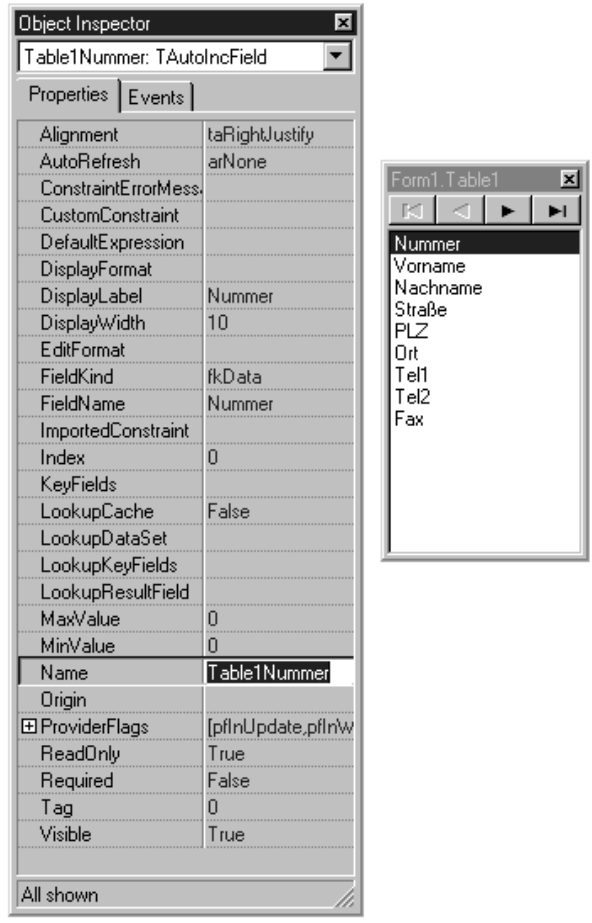
  Mit dem Attribut *faRequired* können Felder definiert werden, deren Eingabe erforderlich ist, mit *faReadonly* Nur-Lese-Felder.

3.2 TField

Für die Spalten von Datenmengen werden Instanzen von *TField*-Nachkommen eingerichtet. Entweder werden schon zur Entwurfszeit so genannte *persistente TField-Instanzen* angelegt – deren Eigenschaften können dann mit dem Objektinspektor bearbeitet werden –, oder beim Öffnen der Datenmenge werden automatisch *TField*-Instanzen angelegt.

Erzeugen statischer TField-Instanzen

Um statische Feldkomponenten zu erzeugen, wird ein Doppelklick auf die Datenmengenkomponente ausgeführt. Delphi öffnet dann den Feldeditor (siehe Abbildung, rechts). Mit dem Kontextmenü kann man hier *TField*-Instanzen hinzufügen:



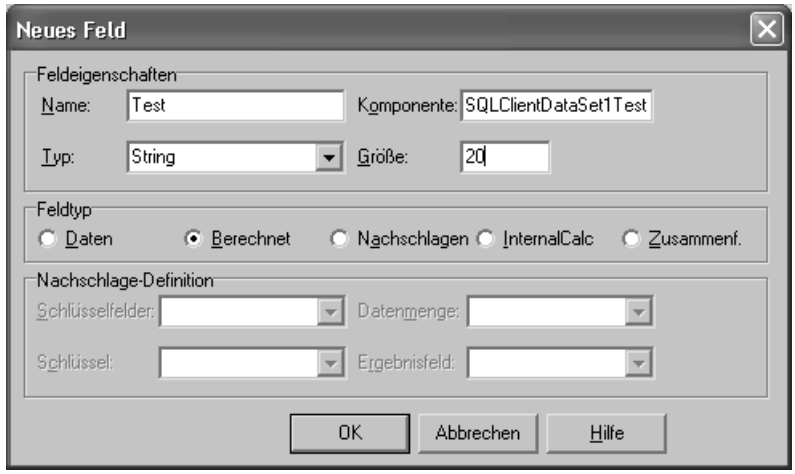
Welche Vorteile hat das Anlegen persistenter *TField*-Instanzen?

- Die Eigenschaften können mit dem Objektinspektor bearbeitet werden. In Kapitel 1 haben wir beispielsweise die Eigenschaft *DisplayFormat* auf diese Weise gesetzt.
- Mit Drag&Drop können die Felder auf ein Formular gezogen werden, Delphi legt dann die passenden Datensteuerungskomponenten an.
- Möchte man berechnete Felder oder Nachschlagefelder verwenden, dann müssen ohnehin persistente *TField*-Instanzen eingerichtet werden.

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, sondern bei der Abfrage berechnet. Dazu wird eine neue *TField*-Instanz eingefügt, die ein berechnetes Feld erstellt.

Wählen Sie dazu aus dem Kontextmenü des Feldereditors den Eintrag NEUES FELD. Es erscheint der folgende Dialog:



Wählen Sie hier als Feld-Typ *Berechnet*, geben Sie der *TField*-Instanz einen Namen (*Test*) und wählen Sie einen Typ (*String*). Bei manchen Typen muss dazu auch noch die Größe eingegeben werden (20).

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

```
procedure TForm1.Table1CalcFields(Sender: TObject)
begin
  Table1.FieldByName('Test').AsString
    := Table1.FieldByName('Vorname').AsString + ' '
    + Table1.FieldByName('Nachname').AsString;
end;
```

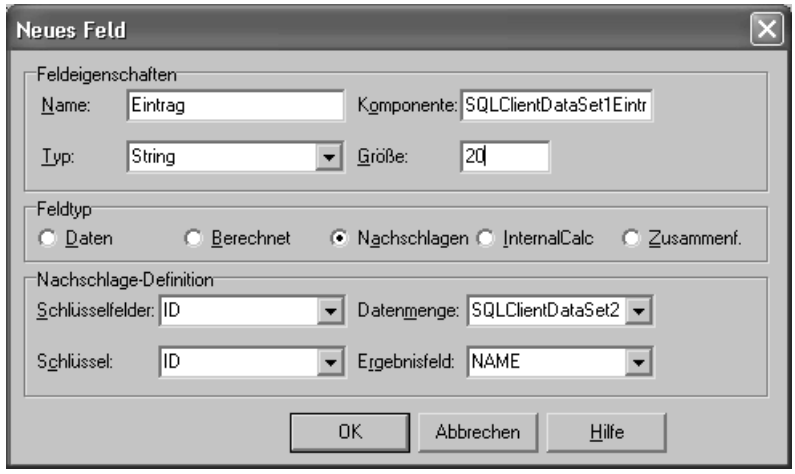
Hier im Beispiel wird dem Inhalt des berechneten Feldes eine Zusammenfügung aus Vor- und Nachname zugewiesen. Dies ist beispielsweise dann sinnvoll, wenn Adressetiketten gedruckt werden sollen.

Wird die Datenmenge mittels eines SQL-Statements generiert, dann könnte man das Feld auch vom Server (beziehungsweise der SQL-Engine der BDE) berechnen lassen:

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

Lookup-Felder

Eine Abwandlung der berechneten Felder sind die Lookup-Felder. Hier wird der Spaltenwert nicht über eine *OnCalcFields*-Ereignisbehandlungsroutine zugewiesen, sondern aus einer anderen Tabelle bezogen (»Nachschlagetabelle«).



Wie bei der Definition eines berechneten Feldes müssen auch hier der Spaltenname und der Spaltentyp eingegeben werden, des Weiteren wird der Radiobutton *Nachschlagen* gewählt.

Mit den Anklicken des Radiobuttons *Nachschlagen* werden die *Nachschlage-Definitionen* verfügbar geschaltet. Als *Schlüsselfeld* wählen Sie die Spalte, welche die Nachschlage-Datenmenge steuern soll:

Nehmen wir an, in unserer Datenmenge sind nur die Mitarbeiternummern gespeichert, der Mitarbeitername soll aus einer Nachschlagetabelle kommen. Dann wird die Spalte, in der die Mitarbeiternummer steht, zum Schlüsselfeld. Als *Datenmenge* wählen Sie die Datenmengenkomponente, aus welcher die einzufügenden Datensätze kommen. Dies könnte eine Abfrage der Tabelle *t_mitarbeiter* sein.

Schlüssel ist die Spalte, die mit *Schlüsselfeld* korrespondiert, in der Regel also der Primärschlüssel der Nachschlagetabelle. Das *Ergenisfeld* ist nun das Feld, das in unsere Datenmenge aufgenommen werden soll, also der Name des Mitarbeiters.

Nachschlagefelder sind etwas für die BDE-Datenbanksysteme dBase und Paradox oder auch für MyBase. Greift man auf einen SQL-Server zu, dann arbeitet man besser mit JOINS.

Datenbankfelder hinzufügen

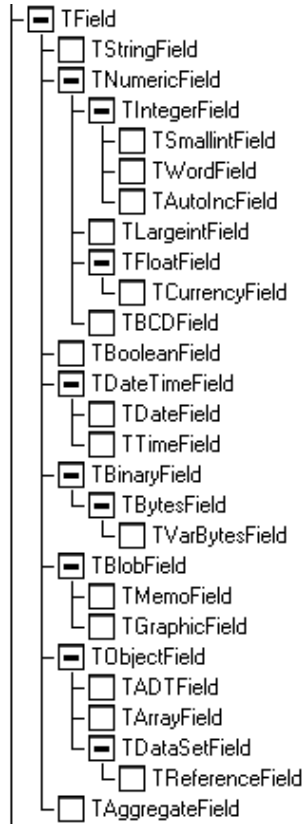
Wenn Sie im Feldereditor eine TField-Instanz löschen, dann ist diese nicht mehr in der Datenmenge. Auch Spalten, die beispielsweise bei einer Tabelle neu hinzugekommen sind oder in eine Abfrage neu aufgenommen wurden, werden erst einmal nicht angezeigt.

Um solche Felder zu ergänzen, wählen Sie im Feldereditor aus dem Kontextmenü den Eintrag FELDER HINZUFÜGEN.

Wählen Sie hier die Spalte oder die Spalten, die Sie hinzufügen möchten.

3.2.1 Referenz von TField

In Delphi wird jedes Feld einer Datenbank durch ein Objekt repräsentiert, das von *TField* abgeleitet worden ist (*TStringField*, *TCurrencyField* usw). Die nachfolgende Abbildung zeigt die Hierarchie der verschiedenen *TField*-Nachkommen.



Wir wollen nun zunächst *TField* besprechen, die *TField*-Nachkommen sind in Kapitel 3.2.2 beschrieben.

Beachten Sie bitte, dass nicht alle Eigenschaften in *TField* auch implementiert sind. So ist beispielsweise die Eigenschaft *AsBoolean* bereits in *TField* definiert, aber erst in *TBooleanField* implementiert.

Zugriff auf die Daten

- AsXxx (Eigenschaften, öffentlich)

```
property AsBCD: TBcd;
property AsBoolean: boolean;
property AsCurrency: Currency;
property AsDateTime: TDateTime;
property AsFloat: double;
property AsInteger: longint;
property AsSQLTimeStamp: TSQLTimeStamp;
property AsString: string;
property AsVariant: Variant;
```

  Mittels dieser Eigenschaften kann im entsprechenden Format sowohl lesend als auch schreibend auf den Inhalt des Feldes zugegriffen werden.

```
Table1.FieldByName('Name').AsString := 'Maier';
i := Table1.FieldByName('Nummer').AsInteger;
```

  TBcd und *TSQLTimeStamp* sind Records. Ihr Aufbau kann in der Online-Hilfe nachgeschlagen werden.

- Assign (*TField*)

```
procedure Assign(Source: TPersistent);
```

  Mit Hilfe von *Assign* kann einem Feld der Inhalt eines anderen Feldes oder eines anderen *TPersistent*-Nachfahren zugewiesen werden. Bei den Nachfahren von *TBlobField* ist dies in der Regel der einzige Weg, um auf den Feldinhalt zugreifen zu können.

```
Table1.FieldByName('Grafik').Assign(Image1.Picture);
```

- OnChange (*TField*)

```
property OnChange(Sender: TField);
```

  Das Ereignis *OnChange* tritt nach dem Speichern der Felddaten auf.

- Clear (Methode), IsNull (Eigenschaft, öffentlich, nur Lesen)

```
procedure Clear;
property IsNull: boolean;
```

  Die Methode *Clear* löscht den Feldinhalt und weist somit dem Feld den Wert NULL zu. Bei einem leeren Datenfeld ist die Eigenschaft *IsNull* gleich *true*.

- NewValue (Eigenschaft, öffentlich)

```
property NewValue: Variant;
```

  Die Werte von *NewValue* und *Value* sind identisch, nur während des Eintragens von zwischengespeicherten Aktualisierungen können sie sich unterscheiden. 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 zugrunde liegenden Datenbanktabelle eingetragen sind.

- OldValue, CurValue (Eigenschaft, öffentlich, nur Lesen)

```
property OldValue: Variant;
property CurValue: Variant;
```

  Durch *OldValue* kann der ursprüngliche Wert des Feldes ermittelt werden, den es vor dem Eintrag zwischengespeicherter Aktualisierungen hatte.

  Tritt beim Eintragen über einen Provider ein Fehler auf, dann kann man mit *CurValue* ermitteln, welchen Wert das Feld derzeit auf dem Server hat.

- Text (Eigenschaft, öffentlich), OnGetText, OnSetText (Ereignis)

```
property Text: string;
property OnGetText(Sender: TField; var Text: String;
  DisplayText: Boolean);
property OnSetText(Sender: TField; const Text: String);
```

  Für gewöhnlich entspricht der Inhalt der Eigenschaft *Text* dem der Eigenschaft *AsString*. Abweichungen können mit Hilfe der *OnGetText*-Ereignisbehandlungsroutine implementiert werden.

  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.

  Das Ereignis *OnSetText* wird ausgelöst, wenn der Eigenschaft *Text* ein Wert zugewiesen wird.

- DisplayText (Eigenschaft, öffentlich, nur Lesen)

```
  property DisplayText: string;
```

  Der Inhalt von *DisplayText* entspricht normalerweise dem von *AsString*. Mit Hilfe der *OnGetText*-Ereignisbehandlungsroutine kann davon Abweichendes programmiert werden.

- GetData, SetData (Methoden)

```
function GetData(Buffer: Pointer): Boolean;
procedure SetData(Buffer: Pointer);
```

  Enthält das Feld Daten, dann werden diese unformatiert in den übergebenen Puffer geschrieben, anderenfalls wird *false* zurückgegeben.

  Die Methode *SetData* weist dem Feld die unformatierten Daten im Puffer zu.

- Value (Eigenschaft, öffentlich)

```
property Value: Variant;
```

  Die Eigenschaft *Value* enthält den Inhalt des Feldes. Die Nachfahren von *TField* implementieren diese Eigenschaft mit verschiedenen Datentypen.

- AssignValue (Methode)

```
procedure AssignValue(const Value: TVarRec);
```

  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.

- OnValidate (Ereignis)

```
property OnValidate(Sender: TField);
```

  Vor dem Speichern der Daten wird das Ereignis *OnValidate* ausgelöst. Dadurch ist eine Überprüfung der Eingabe möglich.

- IsValidChar (Methode), ValidChars (Eigenschaft, öffentlich)

```
property ValidChars: set of Char;
function IsValidChar(InputChar: Char): Boolean;
```

  Mit der Methode *IsValidChar* kann geprüft werden, ob ein bestimmtes Zeichen in einem Feld verwendet werden kann. *TIntegerField* beispielsweise erlaubt nur Ziffern sowie das Plus- und das Minuszeichen, jedoch keine Buchstaben.

  Mit *ValidChars* kann die Menge der erlaubten Zeichen bearbeitet werden. Beispielsweise wäre es möglich, die deutschen Sonderzeichen zu entfernen.

- EditMask (Eigenschaft, veröffentlicht)

```
property EditMask: string;
```

  Mit der Eigenschaft *EditMask* kann eine Eingabemaske für ein Feld erstellt werden. Damit kann die Menge der zulässigen Zeichen für jede Stelle der Eingabe einzeln festgelegt werden.

Anzeige

- Alignment (Eigenschaft, veröffentlicht)

  ```
  property Alignment: (taLeftJustify, taRightJustify, taCenter);
  ```

  Alignment spezifiziert die Ausrichtung des Strings (oder wessen auch immer) im jeweiligen Dialogelement. Die Voreinstellung bei String-Feldern ist linksbündig, bei Zahlenfeldern rechtsbündig.

- DisplayLabel (Eigenschaft, veröffentlicht)

  ```
  property DisplayLabel: string;
  ```

  Normalerweise wird als Titel einer *DBGrid*-Spalte der Name des Feldes angezeigt. Soll ihr ein anderer Titel gegeben werden, kann dies mit der Eigenschaft *DisplayLabel* geschehen. Eine Änderung ist darüber hinaus im Spalteneditor des Datengitters möglich.

- DisplayName (Eigenschaft, öffentlich, nur Lesen)

  ```
  property DisplayName: string;
  ```

  Die Eigenschaft *DisplayName* liefert den aktuellen Feldnamen zu Anzeigezwecken. Ist *DisplayLabel* definiert, dann ist *DisplayName* gleich *DisplayLabel*, andernfalls ist *DisplayName* gleich *FieldName*.

- DisplayWidth (Eigenschaft, veröffentlicht)

  ```
  property DisplayWidth: integer;
  ```

  Die Breite einer *DBGrid*-Spalte richtet sich nach der Länge des dazugehörenden Feldes; diese kann mit der Eigenschaft *DisplayWidth* vermindert oder erweitert werden. Die Breite einer *TDBGrid*-Spalte kann jedoch auch mit dem Spalteneditor von *TDBGrid* geändert werden.

- FocusControl (Methode)

  ```
  function FocusControl;
  ```

  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.

Verbindung zur Datenbank

- FieldName (Eigenschaft, veröffentlicht)

  ```
  property FieldName: string;
  ```

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

- Index (Eigenschaft, veröffentlicht)

  ```
  property Index: integer;
  ```

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

- DataSet (Eigenschaft, öffentlich)

  ```
  property DataSet: TDataSet;
  ```

  Die Komponenteneigenschaft *DataSet* verweist auf die Datenmengenkomponente, aus der das Feld stammt.

- FieldNo (Eigenschaft, öffentlich, nur Lesen)

  ```
  property FieldNo: integer;
  ```

  Feldnummer der eigentlichen Datenbanktabelle. Wird nur für Funktionen der BDE-API benötigt.

Sonstiges

- ReadOnly (Eigenschaft, veröffentlicht)

  ```
  property ReadOnly: boolean;
  ```

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

- CanModify (Eigenschaft, öffentlich, nur Lesen)

  ```
  property CanModify: boolean;
  ```

  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.

- Required (Eigenschaft, veröffentlicht)

  ```
  property Required: boolean;
  ```

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

- AutoGenerateValue (Eigenschaft, veröffentlicht)

  ```
  property AutoGenerateValue: (arNone, arAutoInc, arDefault);
  ```

  Mit dieser Eigenschaft kann einer Datenmenge mitgeteilt werden, dass für dieses Feld vom Server automatisch Daten generiert werden – entweder in Form eines selbstinkrementierenden Feldes oder in Form eines Vorgabewertes. Die Datenmenge muss somit nach dem Einfügen eines Datensatzes solche Felder wieder auslesen, um deren aktuelle Werte darzustellen.

- Calculated (Eigenschaft, veröffentlicht)

  ```
  property Calculated: boolean;
  ```

  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*.

- Visible (Eigenschaft, veröffentlicht)

  ```
  property Visible: boolean;
  ```

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

- CustomConstraint, ImportedConstraint, ConstraintErrorMessage (Eigenschaften, veröffentlicht), HasConstraints (Eigenschaft, veröffentlicht, nur Lesen)

  ```
  property CustomConstraint: String;
  property ImportedConstraint: String;
  property ConstraintErrorMessage: String;
  property HasConstraints: Boolean;
  ```

  Mit *CustomConstraint* kann eine Gültigkeitsprüfung in SQL-Syntax formuliert werden. Dabei wird der Spaltenwert durch einen einheitlichen Bezeichner repräsentiert, der nicht zu den SQL-Schlüsselwörtern gehört:

  ```
  (x < 10) OR (x > 100)
  ```

  Wird gegen *CustomContraint* verstoßen, dann wird eine Exception ausgelöst, welcher der in *ConstraintErrorMessage* vorgegebenen Fehlertext hinzugefügt wird.

  Eine *ImportedConstraint* ist eine Beschränkung, die vom Server importiert wurde. Mit *HasConstraints* kann ermittelt werden, ob eigene oder vom Server importierte Beschränkungen vorliegen.

- DefaultExpression (Eigenschaft, veröffentlicht)

  ```
  property DefaultExpression: String;
  ```

  Mit *DefaultExpression* kann ein Wert vorgegeben werden, mit dem die Spalte beim Einfügen eines neuen Datensatzes vorbelegt wird. Dabei wird SQL-Syntax verwendet, Strings sind somit in Anführungszeichen zu setzen.

- AttributeSet (Eigenschaft, öffentlich)

  ```
  property AttributeSet: string;
  ```

  Die Eigenschaft *AttributeSet* ist der Name der Attributmenge im Daten-Dictionary, die auf diese Feldkomponente angewendet wird.

- DataSize (Eigenschaft, öffentlich, nur Lesen)

  ```
  property DataSize: Word;
  ```

  Größe 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 (Eigenschaft, öffentlich, nur Lesen)

  ```
  property DataType: TFieldType;
  ```

  Die Eigenschaft *DataType* gibt den Datentyp des Feldes an. String-Felder haben beispielsweise den Typ *ftString*.

- FieldKind (Eigenschaft, öffentlich)

  ```
  property FieldKind: (fkData, fkCalculated, fkLookup,
    fkInternalCalc, fkAggregate);
  ```

  Die Eigenschaft *FieldKind* kann folgende Werte annehmen:

  - *fkData*: Feld aus einer Datenbank
  - *fkCalculated*: in einer *OnCalcField*-Ereignisbehandlungsroutine berechnetes Feld
  - *fkLookup*: Lookup-Feld
  - *fkInternalCalc*: von der BDE berechnetes Feld
  - *fkAggregate*: Aggregatfeld einer Client-Datenmenge

- IsIndexField (Eigenschaft, öffentlich, nur Lesen)

  ```
  property IsIndexField: boolean;
  ```

  Die Eigenschaft *IsIndexField* ist gleich *true*, wenn das betreffende Feld indiziert ist.

- Size (Eigenschaft, öffentlich)

  ```
  property Size: word;
  ```

  Definiert die Größe des Feldes in Zeichen (*TStringField*, *TMemoField*) oder Bytes (*TBlobField* und Nachkommen).

- ProviderFlags (Eigenschaft, veröffentlicht)

  ```
  property ProviderFlags: set of (pfInUpdate, pfInWhere,
    pfInKey, pfHidden) default [pfInUpdate, pfInWhere,];
  ```

  Wird in einer Client-Datenmenge die Methode *ApplyUpdates* aufgerufen, dann generiert der Provider SQL-Statements, um die Änderungen zum Server zu übertragen. Solange nur auf einzelne Tabellen zugegriffen wird, ist dies weiter kein Problem. Wenn jedoch JOINS verwendet werden, dann muss dem Provider mitgeteilt werden können, für welche Felder Daten in die Tabelle geschrieben werden sollen und wie diese gefunden werden. Beachten Sie dabei bitte, dass der Provider nur das aktualisieren kann, was sich mittels einer einzigen INSERT-, UPDATE- oder DELETE-Anweisung aktualisieren lässt.

  - Damit die Spalte – so ihr Inhalt geändert wurde – über das INSERT- oder UPDATE-Statement geändert wird, muss das Flag *pfInUpdate* gesetzt sein.
  - Hat *UpdateMode* den Wert *upWhereAll* oder *upWhereChanged*, dann wird die Spalte nur dann in die WHERE-Klausel aufgenommen, wenn das Flag *pfInWhere* gesetzt ist.
  - Mittels des Flags *pfInKey* ermittelt der Provider, welche Spalten zum Schlüssel gehören, wenn *UpdateMode* den Wert *pfWhereKeyOnly* hat.
  - Mit *pfHidden* werden Spalten gekennzeichnet, die nicht angezeigt werden, sondern lediglich zu dem Zweck abgerufen werden, um später die Änderungen wieder eintragen zu können.

Lookup-Felder

- KeyFields (Eigenschaft, veröffentlicht)

  ```
  property KeyFields: string;
  ```

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

- Lookup (Eigenschaft, veröffentlicht)

  ```
  property Lookup: boolean;
  ```

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

- LookupDataSet (Eigenschaft, veröffentlicht)

  ```
  property LookupDataSet: TDataSet;
  ```

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

- LookupKeyFields (Eigenschaft, veröffentlicht)

  ```
  property LookupKeyFields: string;
  ```

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

- LookupResultField (Eigenschaft, veröffentlicht)

  ```
  property LookupResultField: string;
  ```

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

- LookupCache (Eigenschaft, veröffentlicht), RefreshLookupList (Methode)

  ```
  property LookupCache: Boolean;
  procedure RefreshLookupList;
  ```

  Wird *LookupChache* auf *true* gesetzt, dann werden die Inhalte von Nachschlagefeldern zwischengespeichert und müssen nicht immer neu aus der Nachschlagedatenmenge geholt werden. Eine solche Zwischenspeicherung hat dann Vorteile, wenn die Nachschlagedatenmenge vergleichsweise klein ist und somit immer wieder dieselben Daten angezeigt werden.

  Mit *RefreshLookupList* wird der Cache neu gefüllt.

3.2.2 Die TField-Nachkommen

Die folgenden Elemente sind nur bei einer Teilmenge der *TField*-Nachkommen implementiert.

- BlobType (Eigenschaft, veröffentlicht, *TBlobField*)

  ```
  property BlobType: TBlobType;
  ```

  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

- Currency (Eigenschaft, veröffentlicht,*TFloatField, TBCDField*)

  ```
  property Currency: boolean;
  ```

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

- DisplayFormat (Eigenschaft, veröffentlicht,*TNumericField, TDateTimeField*)

  ```
  property DisplayFormat: string;
  ```

  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 EUR* zuweisen. Näheres dazu in der Online-Hilfe.

- DisplayValues (Eigenschaft, veröffentlicht,*TBooleanField*)

  ```
  property DisplayValues: string;
  ```

  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.

- EditFormat (Eigenschaft, veröffentlicht, *TNumericField*)

  ```
  property EditFormat: string;
  ```

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

- EditMask (Eigenschaft, veröffentlicht, *TStringField*)

  ```
  property EditMask: string;
  ```

  Bisweilen kann es sinnvoll sein, eine Eingabe in ein bestimmtes Schema zu zwingen (beispielsweise eine Postleitzahl in die Form *12 345*), oft geschieht dies zur Überprüfung der Eingabe. Diesem Zweck dient die Eigenschaft *EditMask*. Die Syntax der Eingabemaske gleicht der von *TMaskEdit* (siehe dort in der Online-Hilfe).

- MaxValue, MinValue (Eigenschaften, veröffentlicht, *TIntegerField, TFloatField, TBCDField*)

  ```
  property MaxValue: longint;
  property MinValue: longint;
  ```

  Die Eigenschaft *MaxValue* bestimmt den maximalen Wert eines Zahlenfeldes, die Eigenschaft *MinValue* den minimalen Wert.

- Precision (Eigenschaft, veröffentlicht, *TFloatField*)

  ```
  property Precision: integer;
  ```

  Legt die Genauigkeit fest, mit der Gleitkommazahlen angezeigt werden.

- BlobSize (Eigenschaft, veröffentlicht, *TStringField*, *TBlobField*)

  ```
  property BlobSize: integer;
  ```

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

- Transliterate (Eigenschaft, veröffentlicht, *TStringField*, *TMemoField*)

  ```
  property Transliterate: boolean;
  ```

  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.

- SetFieldType (Methode *TField*)

  ```
  procedure SetFieldType(Value: TFieldType);
  ```

  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

Speichern in Dateien und Streams

- LoadFromFile (Methode, *TBlobField*)

  ```
  procedure LoadFromFile(const FileName: string);
  ```

  Die Methode *LoadFromFile* lädt den Wert aus einer Datei in das Feld.

- LoadFromStream (Methode, *TBlobField*)

  ```
  procedure LoadFromStream(Stream: TStream);
  ```

  Die Methode *LoadFromStream* lädt den Wert aus einem Stream in das Feld.

- SaveToFile (Methode, *TBlobField*)

  ```
  procedure SaveToFile(const FileName: string);
  ```

  Die Methode *SaveToFile* speichert den Feldinhalt in eine Datei.

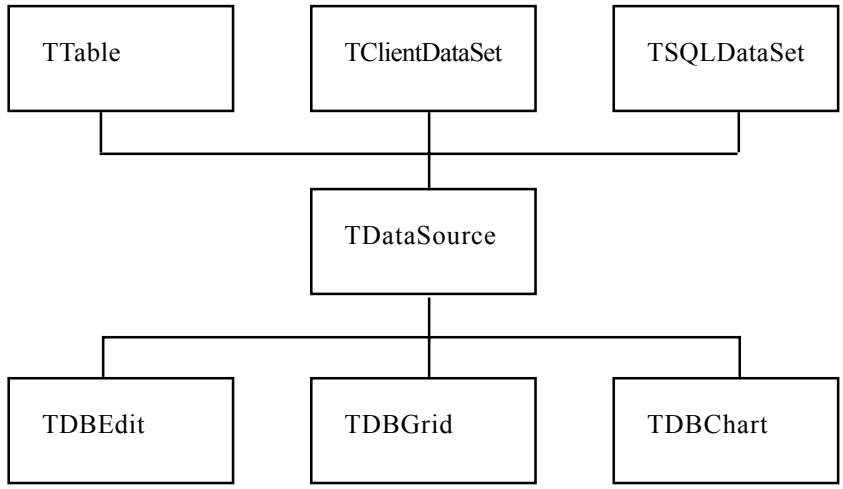
- SaveToStream (Methode, *TBlobField*)

  ```
  procedure SaveToStream(Stream: TStream);
  ```

  Die Methode *SaveToStream* speichert den Feldinhalt in einen Stream.

3.3 TDataSource

Die Komponente *TDataSource* verbindet *TDataSet*-Komponenten (*TClientDataSet*, *TTable*, *TQuery*, *TStoredProcedure* ...) mit den entsprechenden datensensitiven Dialogelementen.



Zu was braucht es diese Verbindungskomponente, warum werden nicht alle Datensteuerungskomponenten direkt an die Datenmengenkomponenten gehängt?

- Manche Tabellen und manche Abfragen sind recht groß, da können schnell einmal 30 datensensitive Dialogelemente zusammenkommen. Sollen die nun einmal an *TTable*, einmal an *TQuery* gehängt werden, dann müsste man von allen 30 Komponenten die Eigenschaft *DataSet* ändern. Bei der Verwendung von *TDataSource* muss nur eine einzige Eigenschaft geändert werden.
- Wenn Anwendungen umgestellt werden, beispielsweise von der BDE auf IBX, dann müssen nur die *TDataSource*-Instanzen auf die neuen Komponenten eingestellt werden. Wer schon einmal *QuickReport*-Formulare umgestellt hat – dort hängen die Komponenten direkt an TDataSet –, der weiß, wie unerfreulich und fehleranfällig eine solche Umstellung ohne *TDataSource* wäre.
- DataSet (Eigenschaft, veröffentlicht)

  ```
  property DataSet: TDataSet;
  ```

  Die wohl wichtigste Eigenschaft von *TDataSource* ist die Eigenschaft *DataSet*. Hier wird festgelegt, mit welcher *TDataSet*-Komponente die Datenquelle verbunden ist. Diese Eigenschaft kann auch zur Laufzeit verändert werden.

- AutoEdit (Eigenschaft, veröffentlicht)

```
property AutoEdit: Boolean default true;
```

Ist die Eigenschaft *AutoEdit* gleich *true*, 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 vor einer Änderung die Methode *Edit* aufgerufen werden.

- Enabled (Eigenschaft, veröffentlicht)

```
property Enabled: Boolean default true;
```

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.

- State (Eigenschaft, öffentlich, nur Lesen)

```
property State: (dsInactive, dsBrowse, dsEdit, dsInsert,
  dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue,
  dsCurValue, dsBlockRead, dsInternalCalc);
```

Die Eigenschaft *State* gleicht der gleichnamigen Eigenschaft der verbundenen Datenmenge, es sei denn, die Eigenschaft *Enabled* ist gleich *false* – dann ist *State* gleich *dsInactive*.

- OnDataChange (Ereignis)

```
property OnDataChange(Sender: TObject; Field: TField);
```

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 wurde früher, als es das Ereignis *AfterScroll* noch nicht gab, dazu verwendet, zwei *DataSet*-Komponenten zu synchronisieren. Hier im Beispiel ist *DataSource2* mit *TQuery1* und einem *DBGrid* verbunden. Immer wenn in diesem *DBGrid* zu einem anderen Datensatz gewechselt wird, soll dieser in der aus *TEdit*-Komponenten zusammengesetzten Eingabemaske angezeigt werden:

```
procedure TForm4.DataSource2DataChange(Sender: TObject;
  Field: TField);
begin
  Table1.FindKey([Query1.FieldByName('Nummer').AsInteger]);
end;
```

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

- OnStateChange (Ereignis)

```
property OnStateChange(Sender: TObject);
```

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, einen Bit- oder Speed-Button 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 *TDataSet*-Methoden über Buttons oder Menüpunkte aufrufen.

Sollen – analog zum *DBNavigator* – nur die Menüpunkte 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 (Ereignis)

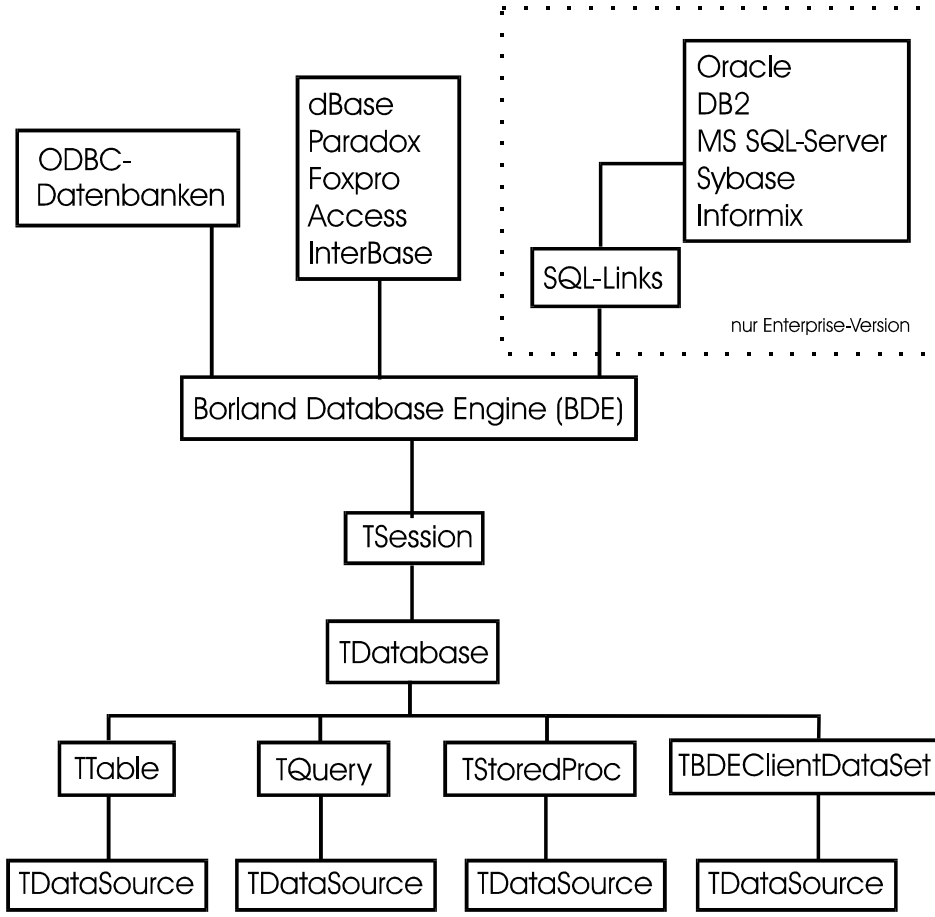
```
property OnUpdateData(Sender: TObject);
```

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

4 Die BDE

Lange Jahre war die BDE (Borland Database Engine) der einzig »offizielle« Weg des Datenbankzugriffs unter Delphi. Inzwischen hat Borland andere Wege in Delphi implementiert (ADO, IBX, dbExpress) und erklärt, die BDE nicht mehr weiterentwickeln zu wollen. Die BDE ist damit ein »Auslaufmodell«, aber immer noch ein sehr bequemer und – wenn man es richtig angeht – effektiver Weg des Datenbankzugriffs.

Die folgende Abbildung zeigt, welche Komponenten beim Einsatz der BDE zusammenarbeiten:



Die BDE greift auf drei Gruppen von Datenbanken zu:

- Lokale Datenbanken wie beispielsweise dBase und Paradox, aber auch Access und Foxpro. Auf diese Datenbanken kann direkt zugegriffen werden. Des Weiteren kann die BDE direkt auf den InterBase-Client zugrifen, so dass hier das Programm SQL Links nicht benötigt wird.
- Die BDE kann auf alle eingerichteten ODBC-Datenbanken zugreifen, dafür werden automatisch entsprechende Aliase eingerichtet. Auf diese Weise kann auf fast jedes Datenbanksystem zugegriffen werden.
- Über das Programm *SQL Links* kann auf verschiedene SQL-Server zugegriffen werden, beispielsweise auf Oracle, MS SQL Server oder DB2. Das Programm *SQL Links* wird jedoch nur mit der Enterprise-Version geliefert.

Auf alle diese Datenbanken kann über die BDE mit einer ziemlich einheitlichen Schnittstelle zugegriffen werden. In der Theorie sieht das so aus, dass man eine Datenbankanwendung mit Paradox beginnen könnte. Steigt irgendwann die Zahl der Anwender auf ein Maß, das eine Client-Server-Anwendung ratsam erscheinen lässt – und sei es, dass eine Firma ganz unerwartet floriert –, dann spielt man die Datenbank auf einen SQL-Server, setzt den BDE-Alias um und arbeitet ohne Redesign der Anwendung weiter.

Wie gesagt, diese Möglichkeit besteht nur theoretisch. Zwar würde – zumindest in den meisten Fällen – eine Anwendung auch nach einer solchen Umstellung noch laufen, aber doch in der Regel nicht mehr besonders performant. Auf Client-Server-Systeme muss eben ganz anders zugegriffen werden, als man das von lokalen Datenbanken her gewohnt ist.

Leider wurde das in den Anfangstagen von Delphi nicht deutlich genug betont, so dass sich hartnäckig das Gerücht hält, man könne mit der BDE keine performanten Client-Server-Anwendungen schreiben. Richtig ist: Die BDE biegt viel »Unsinn« so hin, dass er lauffähig ist – es wäre zu viel verlangt, dass es dann auch noch performant sein soll.

In einer Delphi-Anwendung stellt eine *TSession*-Komponente die Verbindung zur BDE her. Eine solche *TSession*-Instanz wird automatisch erzeugt. Für den Fall jedoch, dass in einer Multithread-Anwendung mehrere Threads auf die BDE zugreifen, muss für jeden Thread eine eigene *TSession*-Instanz verwendet werden. Einzig zu diesem Zweck findet man diese Komponente auf der Komponentenpalette.

An jede Session wird eine Database gehängt. Solange wir eine Singlethread-Anwendung haben und somit nur eine *TSession*-Instanz, kann man auch darauf verzichten, *TDatabase* manuell zu instantisieren – auch dies wird dann automatisch erledigt. Allerdings gibt es insbesondere bei Client-Server-Anwendungen gute Gründe, auf die Eigenschaften von *TDatabase* zuzugreifen, und damit man dies

mit dem Objektinspektor tun kann, sollte man eine solche Komponente manuell einfügen.

Zum Zugriff auf die Datenbank stehen uns vier Komponenten zur Verfügung:

- *TTable* greift auf eine Tabelle oder eine VIEW zu. Der Zugriff auf Client-Server-Systeme ist von *TTable* aus zwar grundätzlich möglich, jedoch nicht in allen Fällen performant – in einzelnen Fällen sind die Ausführungszeiten sogar indiskutabel hoch. *TTable* ist für den Zugriff auf lokale Datenbanken hin optimiert und sollte nach Möglichkeit auch so eingesetzt werden.
- *TQuery* ist eine Komponente, die mittels SQL auf eine Datenbank zugreift. Das beschränkt sich nicht auf Lesezugriffe mittels SELECT, sondern umfasst auch Datenänderungen mittels INSERT, UPDATE und DELETE sowie die Änderung der Metadaten.

  TQuery ist nicht auf den Einsatz mit Client-Server-Systemen beschränkt. Die BDE besitzt eine eigene SQL-Engine, welche beim Zugriff auf lokale Datenbanken (und bei heterogenen JOINS) zum Einsatz kommt. Allerdings ist die Performance dieser SQL-Engine limitiert.

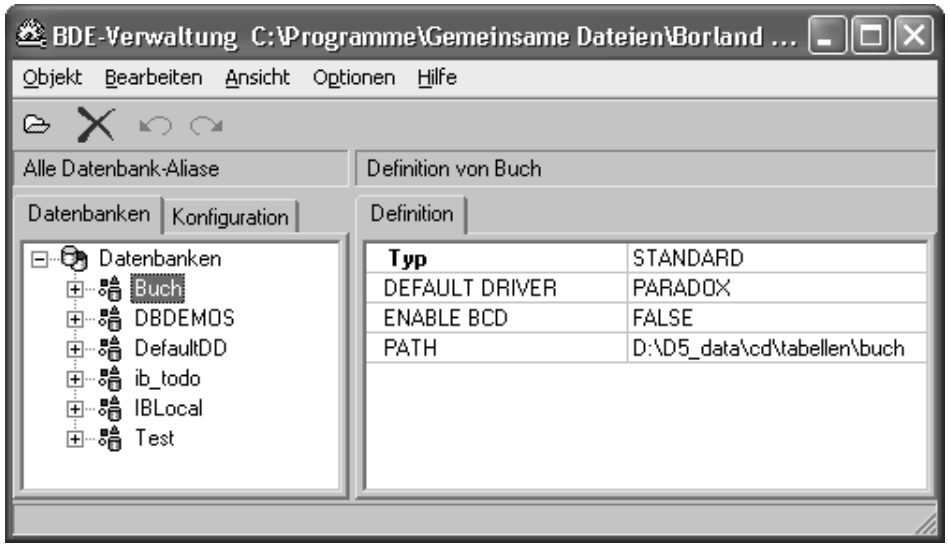
  Mit *TQuery* können so genannte *heterogene JOINS* erstellt werden, also Datenbankabfragen, die ihre Daten aus verschiedenen Datenbanken beziehen. So könnte man beispielsweise Kundendaten, die auf einem SQL-Server liegen, mit lokal gespeicherten Daten kombinieren – nicht nur in derselben Anwendung, sondern sogar in derselben Abfrage,
- *TStoredProc* wird bei manchen SQL-Servern zum Zugriff auf STORED PROCEDURES benötigt. Bei anderen Systemen – beispielsweise Interbase – verwendet man auch dafür *TQuery*.
- Die in Delphi 6 neu hinzugekommene Komponente *TBDEClientDataSet* kombiniert *TClientDataSet* mit dem Zugriff auf die BDE. Dafür ist zwingend nötig, eine *TDatabase*-Instanz manuell einzufügen.

4.1 Die BDE-Verwaltung

Mit der BDE-Verwaltung können Aliase erstellt, Treiber eingebunden und Systemeinstellungen verändert werden. Bei der Entwicklung von Datenbankanwendungen wird man eher selten mit der BDE-Verwaltung arbeiten und stattdessen den Datenbank-Explorer (in der Enterprise-Version *SQL-Explorer*) verwenden.

Allerdings besteht die Möglichkeit, die Installation einer Delphi-Anwendung so zu gestalten, dass die BDE-Verwaltung auf dem Anwenderrechner installiert wird – diese Möglichkeit besteht mit dem Datenbank-Explorer nicht.

Die BDE-Verwaltung ist in der Systemsteuerung zu finden. Sie finden auf der linken Seite zwei Registerseiten, *Datenbanken* und *Konfiguration*. Auf *Datenbanken* finden Sie eine Liste der eingerichteten BDE-Aliase (sowie – falls vorhanden – der ODBC-Aliase).

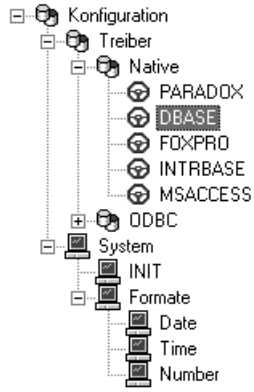


Einen Alias einrichten

Wie ein neuer BDE-Alias angelegt wird, ist in Kapitel 1.2 ausführlich beschrieben.

Konfiguration

Auf der Registerseite Konfiguration können Sie die BDE konfigurieren.



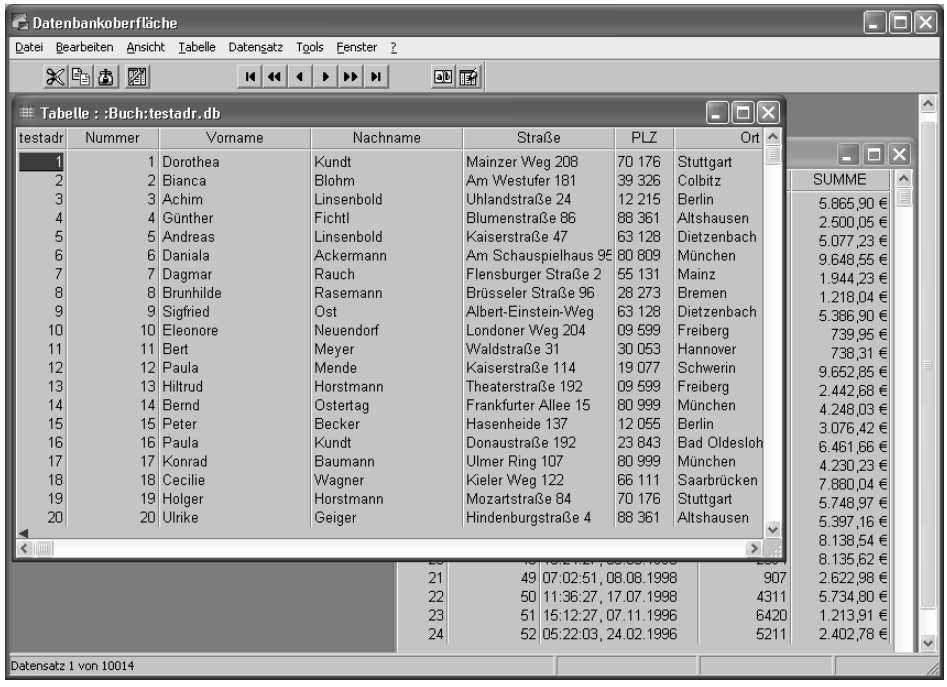
Hier finden Sie beispielsweise eine Liste aller installierten BDE-Treiber (in der Abbildung von der Professional-Version, also ohne die SQL Links-Treiber) und können hier Eigenschaften setzen, die als Vorgabewerte für eine Alias-Neuanlage verwendet werden.

Da der Sprachtreiber sich bei dBase und Paradox nicht auf Alias-Ebene einstellen lässt, ist dies auch die einzige Möglichkeit, einen anderen Sprachtreiber für solche lokalen Datenbanken zu verwenden.

4.2 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.

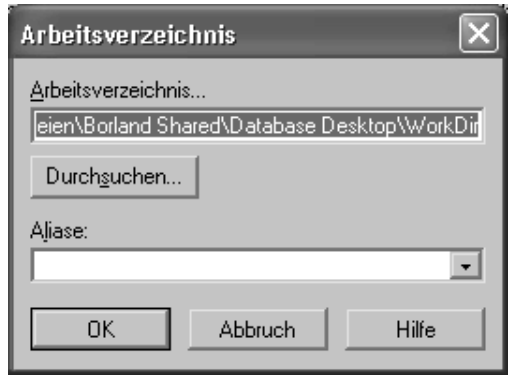
Mit der Datenbankoberfläche kann man auch auf Client-Server-Systeme zugreifen. In der Regel stellen diese jedoch eigene und damit besser geeignete Tools zur Verfügung, so dass die Datenbankoberfläche meist nur für dBase- und Paradox-Datenbanken zum Einsatz kommt.



Die Datenbankoberfläche finden Sie als eigenständige Anwendung in der Delphi-Programmgruppe. Sie können sie aber auch mit TOOLS|DATENBANKOBERFLÄCHE aus Delphi heraus aufrufen.

Einstellen der Arbeitsverzeichnisses

Wenn Sie Tabellen erstellen oder öffnen möchten, dann ist das voreingestellte Verzeichnis das so genannte Arbeitsverzeichnis. Sie können sich die Arbeit mit der Datenbankoberfläche deutlich erleichtern, wenn Sie gleich am Anfang das Arbeitsverzeichnis korrekt einstellen. Wählen Sie dazu DATEI|ARBEITSVERZEICHNIS.



Für das Arbeitsverzeichnis muss nicht zwingend ein Alias eingerichtet sein. Ist jedoch ein solcher eingerichtet, dann ist damit das Verzeichnis schneller gewählt als mit dem Dateidialog.

4.2.1 Öffnen von Tabellen

Mit dem Menüpunkt DATEI|ÖFFNEN|TABELLE können Sie eine bereits vorhandene Tabelle öffnen. Bei dBase und Paradox wird jede Tabelle in einer eigenen Datei gespeichert, somit wird an dieser Stelle auch ein Dateiauswahldialog geöffnet.

Dieser Dateiauswahldialog ist auf das Arbeitsverzeichnis voreingestellt, es kann aber auch ein anderes Verzeichnis gewählt werden, ebenso ein BDE-Alias.

Mit TABELLE|DATEN EDITIEREN oder der Funktionstaste F9 können Sie in den Bearbeitungsmodus schalten. Nun können Sie die Daten ändern, nicht jedoch in BLOB-Feldern (also auch Bilder-Feldern, Memo-Feldern ...).

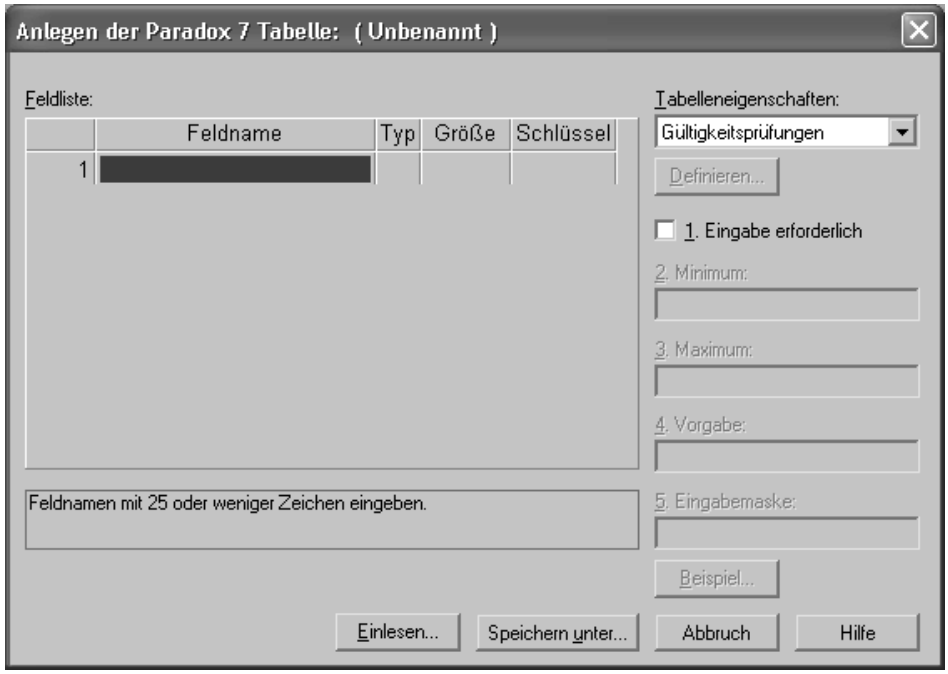
Mit TABELLE|UMSTRUKTURIEREN gelangen Sie in einen Dialog zum Ändern der Tabellendefinition. Hier können Sie all das tun, was wir uns gleich zum Thema *Anlegen von Tabellen* ansehen werden.

4.2.2 Anlegen 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.

Das aktuellste hier unterstützte Format ist *Paradox 7*. Vorhergehende Versionen sollten Sie nur dann verwenden, wenn dies aus Gründen der Abwärtskompatibilität mit DOS- oder Windows 3.1-Programmen erforderlich ist.

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



Um eine Tabelle anzulegen, haben Sie prinzipiell zwei Möglichkeiten:

- Sie können die Definition von einer anderen Tabelle übernehmen und dann gegebenenfalls abändern. Nehmen Sie dazu den Button *Einlesen*.

  Dabei lässt sich detailliert einstellen, welche Elemente neben den Spaltendefinitionen übernommen werden sollen.

- Sie können die Spaltendefinitionen manuell erstellen. Hier geben Sie zunächst die Namen der einzelnen Spalten ein. Verwenden Sie in den Feldnamen auf keinen Fall das Punktzeichen – hier wird dies akzeptiert, aber es wird keine SQL-Abfrage mehr brauchbar mit dieser Spalte funktionieren. Ebenso sollten Sie Leer- und Sonderzeichen vermeiden. Am besten Sie halten sich an das, was auch Delphi als Bezeichner akzeptieren würde.

  Anschließend wählen Sie aus dem Kontextmenü den Spaltentyp (oder Sie verwenden den Buchstaben, der im nächsten Abschnitt angegeben ist).

  Bei manchen Spalten ist die Angabe eine Größe erforderlich. Bei alphanumerischen Spalten ist dies die Größe der Spalte, bei Memos beispielsweise die Größe des Anteils, der in der Tabelle selbst gespeichert wird – der Rest wandert in die *.MB*-Datei.

In der Spalte *Schlüssel* wird angegeben, welche Spalten den Primärschlüssel bilden – diese werden mit einem Stern gekennzeichnet. Diesen Stern erhalten Sie entweder mit der Leertaste oder mit einem Doppelklick.

Achten Sie bitte darauf, dass bei Paradox-Tabellen die erste Spalte oder eine zusammenhängende Menge der ersten Spalte den Primärschlüssel bildet. Überlegen Sie also zunächst, welche Spalten den Primärschlüssel bilden und setzen Sie diesen an den Anfang der Tabelle.

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.
- *EUR(oder $, Währung):* Diesem Zahlenfeld wird automatisch ein String mit der jeweiligen Währung hinzugefügt (in Deutschland EUR). Für gewöhnlich werden zwei Nachkommastellen angezeigt, intern wird mit sechs Nachkommastellen gerechnet.
- *S (Short):* Hierbei handelt es sich um eine 16-Bit-Integerzahl (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):* Enthält Zahlen im BCD-Format. Dieser Feldtyp ist vor allem der Kompatibilität mit anderen Systemen wegen vorhanden und sollte normalerweise nicht verwendet werden.
- *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 in der BDE-Verwaltung 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 in der BDE-Verwaltung 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 Durch-Scrollen einer Tabelle beschleunigt. Eine Eingabe von Memos ist mit der Datenbankoberfläche nicht möglich; verwenden Sie dazu die Delphi-Komponente *TDBMemo*.
- *F (formatiertes Memo):* Hier können auch Informationen zur Schrift u.ä. aufgenommen werden.
- *G (Grafik):* In diese Felder können Bilder gespeichert werden. Eine Eingabe von Bildern ist mit der Datenbankoberfläche nicht möglich; verwenden Sie dazu die Delphi-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 Delphi-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 dann eine Kombination von mehreren Feldern als Primärschlüssel nötig. 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 die 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 benötigt (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. Sollen beispielsweise die Postleitzahlen in der Form *12 345* eingegeben werden, kann dies durch eine entsprechende Eingabemaske sichergestellt werden.

Nachschlagetabelle

Die Nachschlagetabelle gewährleistet die »halbe referenzielle 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 Delphi leider nicht unterstützt).

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

Sekundärindizes

Viele Methoden der Delphi-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 so genannte 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.

Referenzielle Integrität

Was man unter dem Begriff *referenzielle Integrität* versteht, wurde in Kapitel 2.2.2 erläutert. Um eine solche für Paradox-Tabellen zu erreichen, muss man sowohl das Feld, das auf eine andere Tabelle verweist, als auch die Tabelle, auf die verwiesen werden soll, auswählen. Das Referenzfeld 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 Fremdschlüsselfelder von *12345* auf *54321* geändert – die Integrität der Datenbank bleibt gewährleistet.

Beachten Sie bitte, dass bei *weitergeben* gegebenenfalls viele Datensätze aus einer Detailtabelle gelöscht werden können, wenn der dazugehörige Hauptdatensatz entfernt wird.

Passwortschutz

In Datenbanken werden oft Daten abgelegt, 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.

Die folgende Abbildung zeigt, wie eine sehr einfache, kleine Datenbank im Text Editor 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.)

```
1            xnÅ¿)w          À   À¶ 9}Å3}Å ö   /               ·      öö    p¦f2 ù   ÁØ
¶I}Åý}Åƒ}ÅÑ}ÅRESTTEMP.DB
Nummer Namen Hund  intl         ô
  Ç  Maier                                    Ç  Schulze
 Ç  Schulze                                  Ç  Müller
```

Die nachfolgende Abbildung zeigt, wie der Anfang derselben Tabelle mit Passwortschutz aussieht.

```
1            (sw /¸          À    Ö_¸ ô_¸  ö     ô‘¸
‘¸          ·      öö-Ì½¾p¦f2 ù   ÁØ          ¶®_¸ °_¸  _¸ û‘¸ RESTTEMP.DB
 Nummer Namen Hund  intl   Í
-ÂC:q¦]!.ú¸¦ãY>C!±T+¦êÃÔKZÛqñ[ß‘L_¾¦-üqÊMô)1@+y       Ç=¦MzéÏðx+~Ú¤e-
$¦uö¦Ï¹$ÇVÂ|Bý+·Dí:·²½^Ç§+ç¦¶ïPÆr   pçµ{ùY4#+ý.¦(^¶éùîlXÊlE-©Í¹iEù-´6Ìiæèä1â¶8¶Àap
y,ó¤Ýõ¦µ¦ZTö¦ÀY ,oZzÒF“+ZêÆöH\ÐuOÍâ{ù¸(Ë-ü-{û¦Çóú5w#0tV  bÜ[%BWLÝr<²+î+:QÎuÖHl¦-Zhù4;-»\çA
=Ï!  I/¦,’+¦a+Fx+ƒY¦+ØQ<öhÅ¦Á×MÛ$+õCÔ«¦|NÙ+5Ä Æ7èytõƒÁi©c/>dd~O-è-¦ÈÙI:]—FæHDciÌÁôµU¾ú¾
f1.G»|°WíÜÍ+Üm|<ddh ÖñD  ¦B0jù ¦¦L1|ÇZm

öâqp°¦Ðƒ!  zªÈ-xÄPû¥[+Y¦}ÓöU®G¦É?+Í=©

¹ç¦+p¶ËõäÃ×÷¥ë;n¦°QÕýmÄé&rï>+n+¦öJ$ÉÓX „G^GkÃ k Å}+Ié¾l-û      ëØO4£èk1' J¯+ÛA-M¸®-
_°sÌ+¶¼Ú,+×çÄÖÞ@äy¸òËwÄ¦/¦Ê-[`,ÈÙúð^¬ÙË+b»  }¡r|¼Ï¦§Óø-JSÇtï+2-¤+GDxó+¦!;¥äÈ_-Ë=É
±J´¬ü-mÕöCìZ?+æÊº â2As  ¶dFwù¦ì-t(ñÿ70P¥-³é!¦ÉR3¦^¦Áö iWºÄ9r´E<»ç&è+åx©ðLÅ[+te©~¥GR+D-
ñÌ]G&8Ë[¾  -£A’-FWûwóúÐƒ¶5þl^¦Á\9+Ým3:x_zí¦{ÿÈ´±+¨=rÈ`b@ÿA¨-öó^kÇò¦=  ƒÀ-¦=ÚÕÛ·¬¡_¾tj
gþ_Oo’u±~´W ÿ±ÃÜ  ¦ü5vª_äB0lZìDXLüÞ·ÂUwñ~¦¨|+9ä’AÉü0zÃ-JÞ8ÎWä¯QÌ+$×Luö0ô¯Vµ+}-¦_Éa§ª2"ÀF-´I-
OV
```

Wie Sie sehen, sehen Sie nichts – zumindest nichts, was Sie ohne weiteres lesen könnten. Im Übrigen vermittelt die Abbildung 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 Delphi-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 das Zugriffsrecht um auch nur eine einzige Spalte eingeschränkt, dann wird das Tabellenzugriffsrecht 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 Delphi basteln kann, ist das Fehlen von Referenzen sehr ärgerlich: Das Sicherstellen der referenziellen Integrität ist Aufgabe des DBMSs und nicht der Anwendung.
- Es lassen sich keine Passwörter vergeben.

4.2.3 Stapel-Operationen

Mit der Datenbankoberfläche sind auch Stapel-Operationen möglich, also Aktionen, die in Delphi 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 Menütitel *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.

4.2.4 SQL

Auch mit der Datenbankoberfläche können Sie SQL verwenden. Dazu wählen Sie DATEI|NEU|SQL-DATEI und geben eine SQL-Anweisung ein, die Sie dann mit SQL|SQL STARTEN oder der Funktionstaste F8 ausführen können. Dabei sind nicht nur SELECT-Anweisungen erlaubt, sondern Sie können den gesamten Sprachumfang von *LocalSQL* verwenden.

4.3 Der Datenbank-Explorer

Der Datenbank-Explorer (der in der Enterprise-Version *SQL-Explorer* genannt wird) ist ein universelles Tool für die Arbeit mit Datenbanken.

Aliase einrichten

Wie man mit dem Datenbank-Explorer BDE-Aliase einrichtet, wurde bereits in Kapitel 1.2 ausführlich beschrieben.

Interne Aliase

Wird über die Komponente *TDatabase* ein interner Alias eingerichtet, dann erscheint auch dieser im Datenbank-Explorer in der Alias-Liste.

Ist ein Alias-Symbol grün umrandet, dann kennzeichnet dies eine geöffnete Datenbankverbindung. Sie können eine solche Datenbankverbindung über den Datenbank-Explorer schließen, die Eigenschaft *Connected* der zuständigen *TDatabase*-Instanz wird dadurch auf *false* gesetzt.

ODBC-Aliase

In der Liste der Aliase findet man auch alle eingerichteten ODBC-Datenbanken. Als Kennzeichen, dass es sich um eine ODBC-Verbindung handelt, wird eine dunkelblaue, liegende Acht über dem Symbol angezeigt, im folgenden Beispiel bei MQIS:

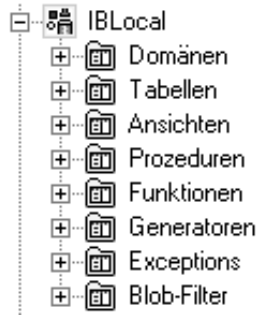
4.3.1 Eine Datenbank untersuchen

Ein *Explorer* wäre in der wörtlichen Übersetzung ein *Forscher*. Wir wollen nun den Datenbank-Explorer verwenden, um eine Datenbank zu erforschen. Damit die Sache nicht zu »billig« wird, verwenden wir die Interbase-Beispieldatenbank *IBLocal*.

Nach einem Mausklick auf das Plus neben den entsprechenden Alias-Eintrag werden wir zunächst mit einem Login-Dialog konfrontiert – so gehört sich das für ein »richtiges« Datenbanksystem.

Als Benutzernamen (beziehungsweise dem, was bei einer festen Label-Größe davon übrig geblieben ist) verwenden wir *SYSDBA*, als Passwort *masterkey*.

Wie sehen nun eine Liste der verschiedenen Elemente, die hier vorhanden sein können:



Domänen

Schauen wir uns zunächst die Domänen (DOMAIN) an, die Typendefinitionen der Spalten. Hier sehen wir zunächst einige selbst definierte Domänen, anschließend eine Menge Domänen, deren Namen mit RDB$ beginnt. Das Präfix RDB$ ist ein Kennzeichen dafür, dass dieses Element von Interbase selbst angegelegt wurde.

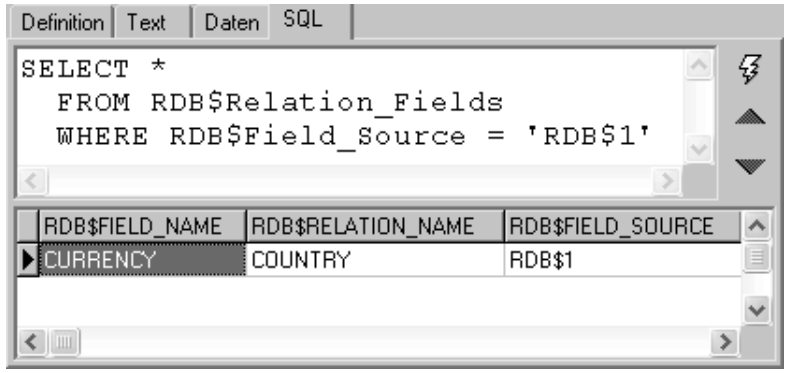
Warum werden nun vom System massenhaft Domänen angelegt und für welche Tabellen? Lassen Sie uns der Sache nachgehen.

Zunächst benötigen wir die Anzeige der Systemtabellen: Stellen Sie sicher, dass der Menüpunkt ANSICHT|SYSTEMDATEN markiert ist.

Wir wollen nun untersuchen, in welcher Tabelle die Domäne *RDB$1* verwendet wird. Alle Spaltendefinitionen sind in der Tabelle *RDB$Relation_Fields* zu finden. Anstatt lange zu suchen, formulieren wir eine SQL-Anweisung:

```
SELECT *
  FROM RDB$Relation_Fields
  WHERE RDB$Field_Source = 'RDB$1'
```

Auf der rechten Seite finden Sie eine Registerseite *SQL*, auf der SQL-Anweisungen eingegeben werden können. Mit dem »Blitz« am rechten Rande können diese ausgeführt werden, und wenn die Anweisung ein SELECT-Statement ist, dann wird darunter die Ergebnismenge angezeigt.



Es handelt sich also um die Spalte *Currency* in der Tabelle *Country*. Expandieren wir nun den Eintrag *Tabellen*, wählen *Country* und auf der rechten Seite den Register-Reiter *Text*: Dort wird nun die Definition der Tabelle angezeigt:

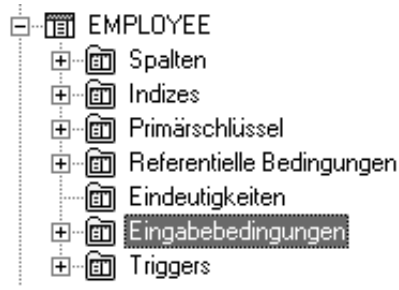
```
CREATE TABLE COUNTRY (
  COUNTRY COUNTRYNAME NOT NULL
  CURRENCY VARCHAR(10) NOT NULL
)
```

(Leider wird dieses SQL-Statement nicht ganz so hübsch formatiert, wie ich das gerne hätte ...)

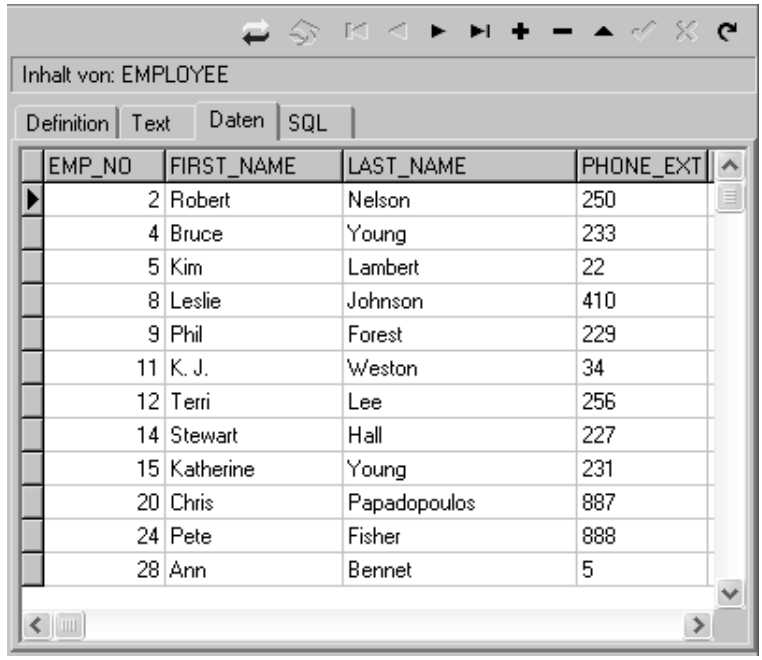
Warum nun die automatisch angelegte Domäne? Für alle Spalten, bei denen nicht eine selbst definierte Domäne verwendet wird, wird automatisch eine angelegt, und wenn das System hunderte *VARCHAR(20)*-Domänen erstellen muss.

4.3.2 Tabellen

Nun wollen wir uns die Tabelle *Employee* näher ansehen – diese ist interessanter als *Country*.



Zunächst besteht die Möglichkeit, sich auf der Registerseite *Daten* (rechts) den Inhalt der Tabelle anzusehen. Diese Registerseite ist jedoch nur dann sichtbar, wenn die Tabelle selbst gewählt ist.



EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT
2	Robert	Nelson	250
4	Bruce	Young	233
5	Kim	Lambert	22
8	Leslie	Johnson	410
9	Phil	Forest	229
11	K. J.	Weston	34
12	Terri	Lee	256
14	Stewart	Hall	227
15	Katherine	Young	231
20	Chris	Papadopoulos	887
24	Pete	Fisher	888
28	Ann	Bennet	5

Der DBNavigator oberhalb der Tabelle lässt vermuten, dass die Datenmenge hier auch editiert werden kann (was eine durchaus zutreffende Vermutung ist ...). Die beiden Buttons links daneben dienen der Transaktionskontrolle. Mit dem linken Button kann man Transaktionen starten und bestätigen, mit dem rechten Button kann man Transaktionen verwerfen.

Spalten

Unter Spalten finden wir eine Auflistung aller Spaltendefinitionen. Dabei kann auf der Registerseite *Text* auch gleich das Statement zur Definition dieser Spalte erfragt werden, hier im Beispiel von *Hire_Date*.

```
HIRE_DATE TIMESTAMP DEFAULT 'NOW' NOT NULL
```

Indizes

Unter Indizes findet man nicht nur solche, die man selbst definiert hat, sondern auch vom Datenbanksystem angelegte. Diese erkennt man am Präfix *RDB$*.

Das Datenbanksystem legt automatisch Indizes für alle Primär-, Sekundär- und Fremdschlüssel an.

Die Primärschlüssel, die referenziellen Bedingungen und die Eindeutigkeiten sollen jetzt keine besondere Beachtung finden.

Eingabebedingungen

In »richtigen« SQL-Systemen umfassen Gültigkeitsprüfungen nicht nur einfache Wertebereiche, sondern man kann alles prüfen, was man mittels einer SQL-Anweisung formulieren kann. Hier wird beispielsweise mittels zwei Unterabfragen geprüft, ob das eingegebene Gehalt »im Rahmen« liegt.

```
/*  CheckConstraint INTEG_30  */
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT INTEG_30
    CHECK ( salary >= (SELECT min_salary FROM job WHERE
      job.job_code = employee.job_code AND
      job.job_grade = employee.job_grade AND
      job.job_country = employee.job_country) AND
    salary <= (SELECT max_salary FROM job WHERE
      job.job_code = employee.job_code AND
      job.job_grade = employee.job_grade AND
      job.job_country = employee.job_country))
```

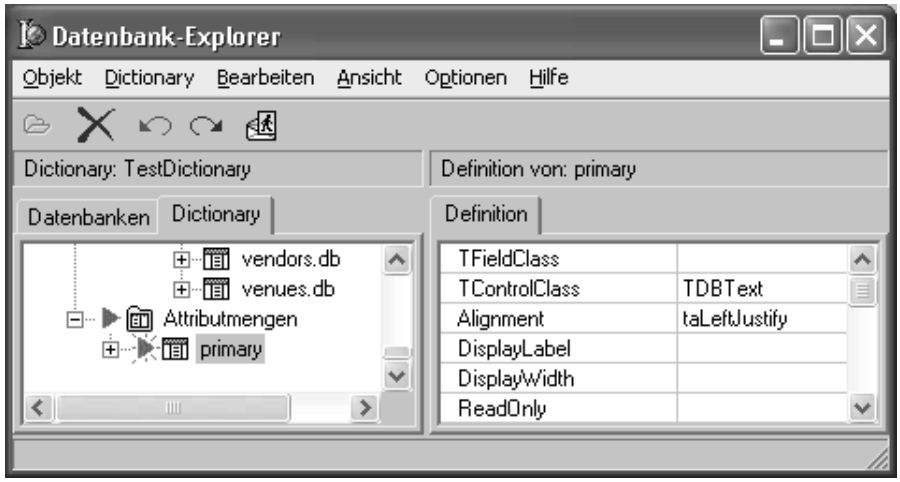
4.3.3 Das Dictionary

Wenn man immer wieder mit derselben Datenbank arbeitet, dann ist es reichlich lästig, immer wieder dieselben *TField*-Definitionen einzugeben. Hier hat man die Möglichkeit, sich ein so genanntes *Dictionary* anzulegen und sich diese Festlegungen darin zu speichern.

Wir wollen ein solches Dictionary für DBDEMOS anlegen. Nehmen wir an, das Dictionary ist noch leer. Dann wählen Sie DICTIONARY|AUS DATENBANK IMPORTIEREN und wählen den gewünschten Alias:

Nun wollen wir zwei Attributmengen anlegen. Dazu wählen wir unter *Attributmengen* aus dem Kontextmenü den Eintrag NEU und geben der neuen Attributmenge den Namen *primary*.

Wir wollen hier so tun, als ob die Werte für den Primärschlüssel aus einem selbstinkrementierenden Feld stammen. In diesem Fall würden Eingaben keinen Sinn machen, weshalb dieses Feld nicht editierbar sein soll.



Anstatt nun einfach *ReadOnly* auf *true* zu setzen und dadurch vielleicht zur Verwirrung des Anwenders beizutragen, wollen wir als *TControlClass* die Komponente *TDBText* wählen – dort kann nichts editiert werden. Als kleine Ergänzung soll *Alignment* auf *taLeftJustify* gesetzt werden. Wählen Sie dann ÜBERNEHMEN aus dem Kontextmenü.

Und weil es so schön ist, wollen wir nun noch die Attributmenge *datum* anlegen und *DisplayFormat* auf *dd.mm.yyyy* setzen.

Eine Attributmenge anwenden

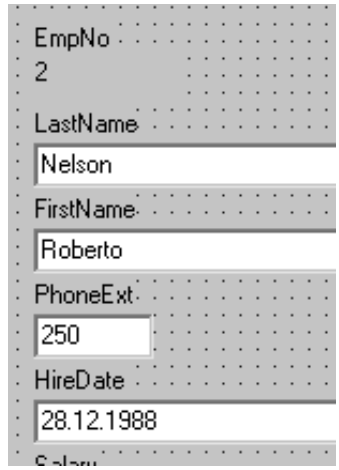
Nun wechseln wir zur Tabelle *employee.db* und weisen hier die Attributmenge *primary* dem Feld *EmpNo* zu.

Die Spalte *HireDate* bekommt die Attributmenge *datum*. Auch diese Änderungen sind jeweils zu übernehmen.

Nun legen wir ein neues Delphi-Projekt an und legen eine *TTable*-Komponente aufs Hauptformular:

- *DatabaseName* erhält den Wert *DBDEMOS* und *TableName* den Wert *employee.db*. Anschließend wird *Active* auf *true* gesetzt.
- Legen Sie nun persistente *TField*-Instanzen an.

- Wählen Sie alle Einträge im Felder-Editor und ziehen Sie diese auf das Formular, um die entsprechenden Datensteuerungskomponenten anzulegen.



Wie wir sehen, ist für *EmpNo* die gewünschte Komponente erstellt worden, und auch die anderen beiden Eigenschaften sind übernommen worden.

Das Dictionary ist eigentlich eine sehr nette Idee, deren praktischen Nutzen dadurch limitiert wird, dass selten mehrere mit Delphi entwickelte Anwendungen auf ein und dieselbe Datenbank zugreifen.

4.3.4 Sonstiges

Noch ein paar Informationen zum Datenbank-Explorer:

- Unter ANSICHT|SCHRIFTART können Sie die Schriftgröße der SQL-Statements einstellen. Die voreingestellte Schriftgröße ist zu Schulungszwecken meist zu klein.
- Unter OPTIONEN|TRANSAKTIONS-ISOLATION können Sie einstellen, welchen Isolationsstatus die (automatisch oder manuell) begonnenen Transaktionen erhalten.
- In der Datenansicht einer Tabelle können Sie mit einem Doppelklick auf Memo-Felder den Inhalt in einem eigenen Fenster anzeigen lassen. Mit dem SQL-Explorer funktioniert das auch bei Graphik-Feldern.
- Aus dem Menü OBJEKT heraus können Sie sowohl die BDE- als auch die ODBC-Verwaltung öffnen.

4.4 Die Komponente TSession

Sobald das Formular um eine Datenzugriffskomponente ergänzt wird, wird automatisch die Komponente *TSession* eingefügt. Die dabei von *TSession* erzeugte Instanz heißt *Session*. Wenn Sie eine Multithread-Anwendung entwickeln, bei der mehrere Threads gleichzeitig auf Datenbanken zugreifen, dann müssen Sie weitere Instanzen von *TSession* einfügen. Wenn Sie keine Multithread-Anwendung programmieren, dann sollten Sie auch nie eine *TSession*-Komponente in ein Formular oder ein Datenmodul einfügen.

4.4.1 Passwort

Tabellen mit sensiblen Daten wird man in der Regel mit einem Passwort sichern. Dieses muss beim Start der Anwendung eingegeben werden. Nun kann eine Datenbank – insbesondere dann, wenn sie normalisiert ist – ohne weiteres über 20 Tabellen enthalten (auch Datenbanken mit über 2000 Tabellen sind nicht selten). 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 dieses nur einmal eingegeben werden muss – 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, für die Anwendung ein anderes Passwort zu vergeben, mit dem der Anwender zwar auf das Programm, nicht aber direkt auf die Tabellen zugreifen kann.

Es ist 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*, 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.

- AddPassword, RemovePassword, RemoveAllPasswords (Methoden)

```
procedure AddPassword(const Password: string);
procedure RemovePassword(const Password: string);
procedure RemoveAllPasswords;
```

Damit die Passwortabfrage 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 der Passwortliste gelöscht.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.AddPassword('chriss');
  Table1.Open;
end;
```

- OnPassword, OnStartup (Ereignisse)

```
property OnPassword(Sender: TObject; var Continue: Boolean);
property OnStartup(Sender: TObject);
```

Wird versucht, eine Paradox-Tabelle zu öffnen, ohne dass dafür das Passwort vorliegt, dann wird ein Standard-Passwort-Dialog angezeigt. Dies kann dadurch vermieden werden, dass dem Ereignis *OnPassword* eine Ereignisbehandlungsroutine zugewiesen wird, in welcher mit *AddPassword* das Passwort angegeben wird.

Das Ereignis *OnStartup* wird ausgelöst, wenn die Datenbanksitzung geöffnet wird, beispielsweise durch die Aktivierung einer ihrer Datenmengenkomponenten.

4.4.2 Alias- und Tabellennamen

Die Komponente *TSession* enthält einige Methoden, mit deren Hilfe man Informationen über die Datenbank erhalten kann.

- Alias (Methoden, TSession)

```
procedure GetAliasNames(List: TStrings);
procedure AddAlias(const Name, Driver: string; List: TStrings);
procedure DeleteAlias(const Name: string);
function GetAliasDriverName(const AliasName: string): string;
procedure GetAliasParams(const AliasName: string; List: TStrings);
procedure ModifyAlias(Name: string; List: TStrings);
```

Mit *GetAliasNames* erhält man eine Liste aller verfügbaren BDE-Aliase. Dies ist beispielsweise dann erforderlich, wenn die Datenbankverbindung zur Laufzeit erstellt werden soll. Im Beispielprogramm zu *TBatchMove* wurde so vorgegangen. Mit Hilfe von *AddAlias* kann ein Alias zur Laufzeit eingerichtet werden.

```
procedure TForm1.Button1Click(Sender: TObject)
var
  Para: TStringList;
begin
  Para := TStringList.Create;
  Para.Add('SERVER NAME=IB_SERVER:/PATH/DATABASE.GDB');
  Para.Add('USER NAME=Anwender');
  Session1.AddAlias('TestIBAlias', 'INTRBASE', Para);
  Para.Free;
end;
```

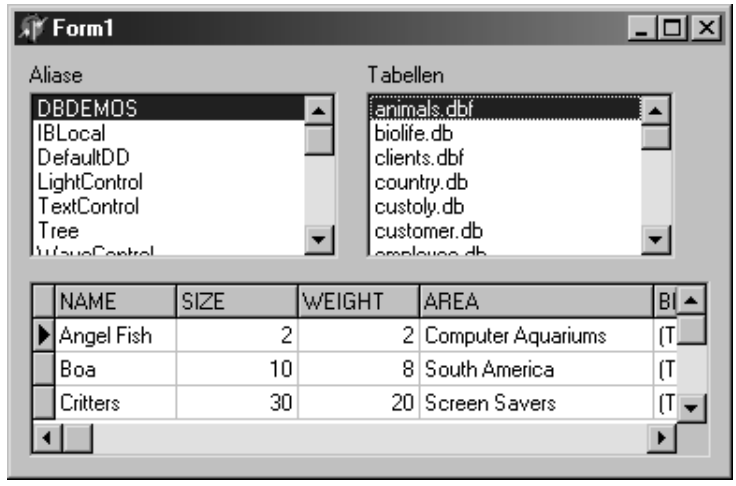
- GetTableNames, GetStoredProcNames (Methoden)

```
procedure GetTableNames(const DatabaseName, Pattern: string;
  Extensions, SystemTables: Boolean; List: TStrings);
procedure GetStoredProcNames(const DatabaseName: string;
  List: TStrings);
```

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.

Kleiner Tabellenbetrachter

Mit den eben vorgestellten Routinen können wir uns einen kleinen Tabellenbetrachter bauen.



Beim Starten der Anwendung werden alle Aliase ermittelt und in die erste Listbox geschrieben.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.GetAliasNames(ListBox1.Items);
end;
```

Wird dann ein Alias gewählt, werden dazu alle Tabellen angezeigt.

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  Session.GetTableNames(ListBox1.Items[ListBox1.ItemIndex],
    '', true, false, ListBox2.Items);
end;
```

Wird nun eine der Tabellen angeklickt, dann wird diese im *DBGrid* dargestellt.

```
procedure TForm1.ListBox2Click(Sender: TObject);
begin
  with Table1 do
  begin
    Close;
    DatabaseName := ListBox1.Items[ListBox1.ItemIndex];
    TableName := ListBox2.Items[ListBox2.ItemIndex];
    Open;
  end;
end;
```

4.4.3 Verzeichnisse

- NetFileDir, PrivateDir (Eigenschaften, veröffentlicht)

  ```
  property NetFileDir: String;
  property PrivateDir: String;
  ```

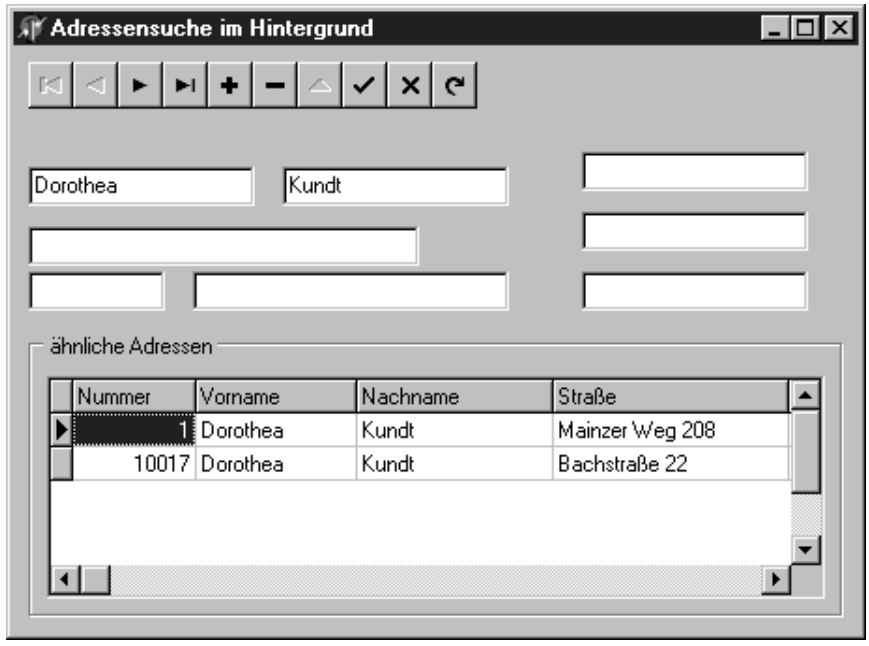
  Soll Paradox im Netzwerk eingesetzt werden, dann greifen mehrere Instanzen der BDE gleichzeitig auf dieselben Dateien zu. Damit dies nicht in einem großen Chaos endet, müssen sich diese BDE-Instanzen »absprechen«, und dazu dient die Datei PDOXUSRS.NET. Diese Datei wird im *NetFileDir* gespeichert. Alle Anwendungen, die auf dieselbe Datenbank zugreifen, sollten tunlichst denselben Wert für *NetFileDir* verwenden.

  Das *PrivateDir* dient zum Speichern temporärer Dateien.

4.4.4 Adressensuche im Hintergrund

Als kleines Beispiel zu *TSession* soll die folgende Multithread-Anwendung dienen: In einer Datenbanktabelle möchte man jeden Datensatz in der Regel nur einmal haben. In einer Kundentabelle beispielsweise sollte jede Kundenadresse nur einmal vorhanden sein. Nun kommt es aber häufig vor, dass ein Kunde, der bereits in der Tabelle ist, sich wieder bei der Firma meldet, seine Kundennummer nicht angibt und somit wieder in die Tabelle aufgenommen wird. Dies führt in der Regel auch dazu, dass dieser Kunde ab sofort alle Kataloge zweimal bekommt.

Hier soll nun ein Formular zur Adresseneingabe so programmiert werden, dass während der Eingabe von Adressen geprüft wird, ob ein entsprechender Datensatz bereits in der Tabelle ist.



Sicherheitshalber wird dabei nicht auf Gleichheit, sondern auf Ähnlichkeit geprüft. Theoretisch wären zwei sonst identische Adressen, bei denen die eine Straße *Bismarckstraße* und die andere *Bismarckstr.* lauten würde, ungleiche Datensätze. In der Praxis möchte man entsprechende Adressen auch bei solchen kleinen Abweichungen finden. Deshalb werden hier relativ grobe Filter erstellt, so dass zwar im Zweifelsfall mehrere Adressen angezeigt werden, worunter sich jedoch die betreffende Adresse mit hoher Wahrscheinlichkeit befindet.

Suche nach Vorname und Nachname

Nachdem Vorname und Nachname eingegeben wurden, soll geprüft werden, ob sich entsprechende Kombinationen bereits in der Tabelle befinden. Der Start der entsprechenden Suche wird an das *OnEnter*-Ereignis von *DBEdit3* (Straße) gekoppelt.

```
procedure TForm1.DBEdit3Enter(Sender: TObject);
begin
  if Table1.State in [dsInsert, dsEdit] then
  begin
    DBGrid1.DataSource := nil;
    with Query1.SQL do
    begin
      Clear;
      Add('SELECT * FROM testadr');
      Add('WHERE UPPER(vorname) LIKE UPPER(:vorname)');
      Add('AND UPPER(nachname) LIKE UPPER(:nachname)');
    end; {with Query1.SQL do}
    Query1.ParamByName('vorname').AsString
      := Table1.FieldByName('vorname').AsString + '%';
    Query1.ParamByName('nachname').AsString
      := Table1.FieldByName('nachname').AsString + '%';
    FSearchThread := TSearchThread.Create(false);
    FSearchThread.FreeOnTerminate := true;
  end; {if Table1.State in [dsInsert, dsEdit]}
end; {procedure TForm1.DBEdit3Enter}
```

Wird nach einem Adressenwechsel ein Datensatz abgeändert, dann kann er auf einen Wert geändert werden, der bereits in der Tabelle vorhanden ist. Damit auch dies festgestellt werden kann, wird die Suche nicht nur bei der Eingabe neuer Datensätze, sondern auch beim Ändern bestehender durchgeführt.

Zunächst wird eine SQL-SELECT-Anweisung formuliert, welche die zu suchenden Werte als Parameter entgegennimmt. Diesem Parameter werden dann die eben getätigten Eingaben zugewiesen. Nun ist es möglich, dass ein und dieselbe Person ihren Namen immer verschieden angibt. Um hier wenigstens in einigen Fällen einen entsprechenden Datensatz zu finden, wird dem eingegebenen Wert das SQL-Jokerzeichen *%* angehängt. Einen Datensatz *Heinz-Udo Müller* würde man auch dann noch finden, wenn die Eingabe *Heinz Müller* wäre. Umgekehrt geht das leider nicht.

Anschließend wird der Thread erzeugt. Damit er wieder freigegeben wird, wenn seine Arbeit erledigt ist, wird *FreeOnTerminate* auf *true* gesetzt.

TSearchThread

Nun wollen wir einen Blick auf unseren Thread werfen.

```
type
  TSearchThread = class(TThread)
  protected
    procedure Execute; override;
    procedure anzeigen;
  end;
```

TSearchThread ist direkt von *TThread* abgeleitet. Neben der obligatorischen Prozedur *Execute* findet sich hier nur noch die Prozedur *anzeigen*.

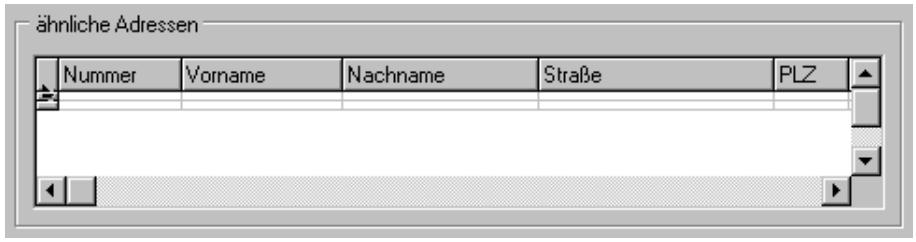
```
procedure TSearchThread.Execute;
begin
  Form1.Query1.Open;
  Synchronize(anzeigen);
end;
```

Die Prozedur *Execute* öffnet *Query1*. Weil die Suche nach entsprechenden Datensätzen in einem anderen Thread abläuft, kann man mit der Eingabe der Adresse fortfahren.

```
procedure TSearchThread.anzeigen;
begin
  Form1.DBGrid1.DataSource := Form1.DataSource2;
end;
```

Mit der Prozedur *anzeigen* wird *DBGrid1* an *DataSource2* gehängt. Da hier auf visuelle Komponenten zugegriffen wird, muss *anzeigen* über *Synchronize* aufgerufen werden.

Würde man *anzeigen* nicht über *Synchronize* aufrufen, dann würden – zumindest auf einem Dual-Prozessor-System – die Spaltenhöhen auf viel zu kleine Werte gesetzt.



Suche nach Straße und Postleitzahl

Für den Fall, dass eine Person ihren Namen ändert oder anders schreibt, wäre die Suche nach Vorname und Nachname nicht zielführend. Deshalb soll, sobald die Straße und die Postleitzahl eingegeben wurden, auch nach dieser Kombination gesucht werden.

```
procedure TForm1.DBEdit5Enter(Sender: TObject);
begin
  if Table1.State in [dsInsert, dsEdit] then
  begin
    DBGrid1.DataSource := nil;
    with Query1.SQL do
    begin
      Clear;
      Add('SELECT * FROM testadr');
      Add('WHERE UPPER(straße) LIKE UPPER(:straße)');
      Add('AND plz LIKE :plz');
    end; {with Query1.SQL do}
    Query1.ParamByName('straße').AsString
      := copy(Table1.FieldByName('straße').AsString, 1, 5) + '%';
    Query1.ParamByName('plz').AsString
      := zahlstring(Table1.FieldByName('plz').AsString);
    FSearchThread := TSearchThread.Create(false);
    FSearchThread.FreeOnTerminate := true;
  end; {if Table1.State in [dsInsert, dsEdit]}
end; {procedure TForm1.DBEdit5Enter}
```

Weitgehend gleicht diese Prozedur der Prozedur *TForm1.Edit3Enter*. Bei der Straße soll es keinen Unterschied machen, ob man beispielsweise *Bismarckstraße* oder *Bismarckstr.*, ob man *Hammer Landstraße* oder *Hammerlandstraße* schreibt. Für die Suche nach Straßennamen kann man ausgefeilte Algorithmen schreiben. Hier im Beispiel wird jedoch die »Holzhammermethode« verwendet und die Auswertung auf die ersten fünf Zeichen beschränkt.

Auch Postleitzahlen sind nicht ganz unproblematisch. So könnte man beispielsweise *10 317*, *10317*, *103 17* oder wie auch immer schreiben. Hier sorgt die Funktion *zahlstring* dafür, dass ein Suchstring *1%0%3%1%7%* alle denkbaren Formatierungsvarianten findet.

Die Funktion *zahlstring* wird im nächsten Abschnitt beschrieben.

Die Funktion zahlstring

Die Funktion *zahlstring* wird verwendet, um Zahlenkombinationen, wie beispielsweise Postleitzahlen oder Telefonnummern, auch bei unterschiedlichen Formatierungen zu finden.

```
function TForm1.zahlstring(rein: string): string;
var
  i: integer;
begin
  result := '';
  for i := 1 to Length(rein) do
  begin
    if rein[i] in ['0'..'9']
      then result := result + rein[i] + '%';
  end; {for i := 1 to Length(rein) do}
end; {function TForm1.zahlstring}
```

Die Funktion geht den als Parameter übergebenen String Zeichen für Zeichen durch. Handelt es sich um ein Ziffernzeichen, dann wird dieses dem Funktionsergebnis hinzugefügt, des Weiteren wird jeweils ein %-Zeichen angehängt.

Suche nach der Telefonnummer

Als letzte Chance, eine Adresse zu finden, wird hier die Telefonnummer genutzt.

```
procedure TForm1.DBEdit7Enter(Sender: TObject);
begin
  if Table1.State in [dsInsert, dsEdit] then
  begin
    DBGrid1.DataSource := nil;
    with Query1.SQL do
    begin
      Clear;
      Add('SELECT * FROM testadr');
      Add('WHERE UPPER(tel1) LIKE UPPER(:tel1)');
    end; {with Query1.SQL do}
    Query1.ParamByName('tel1').AsString
      := zahlstring(Table1.FieldByName('tel1').AsString);
    FSearchThread := TSearchThread.Create(false);
    FSearchThread.FreeOnTerminate := true;
  end; {if Table1.State in [dsInsert, dsEdit]}
end; {procedure TForm1.DBEdit7Enter}
```

Auch hier kommt wieder die Funktion *zahlstring* zum Einsatz.

4.5 TDatabase

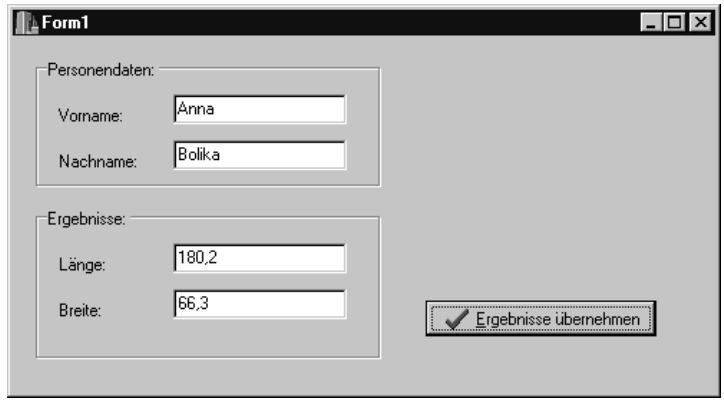
Die Komponente *TDatabase* steuert die Verbindung zu einer Datenbank. Für alle Datenbankverbindungen, welche nicht über eine explizit eingefügte *TDatabase*-Instanz laufen, werden von Delphi automatisch *TDatabase*-Instanzen erstellt. Daneben übernimmt *TDatabase* auch die Transaktionskontrolle.

4.5.1 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 ist 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 sinnvoll, die Nummern von der mit der Eingabe betrauten Person frei nach Lust und Laune auswählen zu lassen. Die Wahrscheinlichkeit ist zu hoch, dass sich eine 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 (so lange nicht der höchst unwahrscheinliche Fall eintritt, dass es wieder zu einer Primärschlüsselverletzung kommt, und diesmal exakt bei der anderen Tabelle).

Bei Einplatzsystemen kann man die beiden Nummern vorher abfragen, damit ist zumindest dieses Problem umgangen. Bei Client-Server-Systemen jedoch ist 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 Transaktionsmanagements ist klar: Es sollen die beiden Datensätze geschrieben werden; geht dabei alles klar, dann wird der Vorgang bestätigt, tritt irgendwo ein Fehler auf, werden beide Einträge zurückgenommen.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  randomize;
end;
```

Vor der Verwendung des Zufallszahlengenerators muss dieser mit *randomize* initialisiert werden.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  try
    Database1.StartTransaction;
    with Table1 do
    begin
      Append;
      FieldByName('vornamen').AsString := Edit1.Text;
```

```
      FieldByName('nachnamen').AsString := Edit2.Text;
      FieldByName('nummer').AsInteger := random(65000);
      Post;
    end;
    with Table2 do
    begin
      Append;
      FieldByName('laenge').AsString := Edit3.Text;
      FieldByName('breite').AsString := Edit4.Text;
      FieldByName('nummer').AsInteger := random(65000);
      Post;
    end;
    Database1.Commit;
    Edit1.Text := '';
    Edit2.Text := '';
    Edit3.Text := '';
    Edit4.Text := '';
  except
    Database1.Rollback;
    ShowMessage('Datensatz nicht geschrieben!');
  end; {try}
end;
```

Die Anweisungen zum Schreiben des Datensatzes werden in einen *try...except*-Rahmen gesteckt. Tritt dort während der Ausführung eine *Exception* auf, dann springt Delphi zum *except*-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. Die beiden Einfügungen werden mit *Commit* bestätigt. Anschließend werden die Edit-Felder gelöscht, damit der Anwender sieht, dass die Datenbank die Eingabe akzeptiert hat und nicht versehentlich alte Werte stehen bleiben.

Tritt während der Ausführung eine *Exception* auf, springt Delphi zu den Anweisungen nach *except*. Zunächst werden die Änderungen widerrufen. Des Weiteren wird eine Fehlermeldung ausgegeben, denn sonst wäre der Fehler für den Anwender nicht mehr erkennbar, sobald das Programm nicht mehr unter dem Debugger läuft.

Rücknahme von falschen Eingaben

Transaktionen können auch dazu verwendet werden, falsche Eingaben und Änderungen zu widerrufen. Angenommen, es soll 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 Anweisungen, die sie wieder rückgängig machen wollen. Sieht beispielsweise das Programm die Möglichkeit vor, SQL-Anweisungen in einem Editor zu erstellen und dann auszuführen, dann kommt es 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 hätten alle Personen in *t_adressen* den Namen *Müller*. Bei der DELETE-Anweisung wollte der Anwender alle Datensätze mit einer Nummer größer als 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, ist es ratsam, die COMMIT-Aktion mit einem Passwort zu koppeln.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DataBase1.StartTransaction;
end;

procedure TForm1.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  DataBase1.Commit;
end;
```

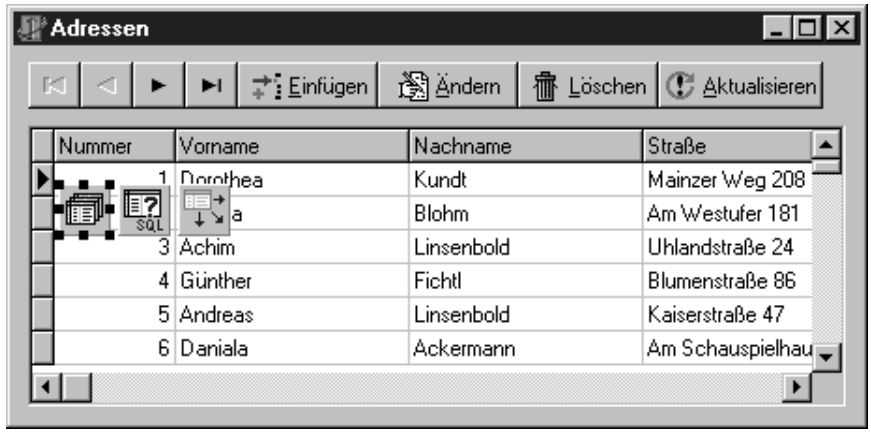
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DataBase1.Commit;
  DataBase1.StartTransaction;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  DataBase1.RollBack;
  Table1.Refresh;
  DataBase1.StartTransaction;
end;
```

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 bisher Eingegebene 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.

4.5.2 Execute

Ab Delphi 5 besteht die Möglichkeit, mit Hilfe der Methode *Execute* SQL-Anweisungen direkt von *TDatabase* aus auszuführen – eine *TQuery*-Instanz ist dann nicht mehr erforderlich. Wir wollen nun anhand eines kleinen Projektes untersuchen, ob der Verzicht auf *TQuery* zu einer messbaren Geschwindigkeitssteigerung führt.

Die folgende Abbildung zeigt das Formular, in dem Datensätze in einem Grid angezeigt werden – dafür wird in beiden Varianten des Programms *TQuery* verwendet.



Um gleiche Ausgangsbedingungen zu schaffen, wird in beiden Fällen keine aktualisierbare Datenmenge erzeugt, sondern die Datenänderungen werden mittels SQL-Anweisungen in die Datenbank geschrieben. (Übrigens: Bei manchen SQL-Servern ist dies die einzig sinnvolle Vorgehensweise.)

Aktualisieren

Wie bereits an anderer Stelle vermerkt wurde, funktioniert die Anweisung *Refresh* bei *TQuery* nicht – es ist erforderlich, die Datenmenge mit *Close* und *Open* neu zu öffnen. Dies hat allerdings den Nachteil, dass hinterher der Cursor auf dem ersten Datensatz steht. Deshalb wurde hier die Prozedur zur Aktualisierung etwas aufwändiger gestaltet: Zunächst wird ermittelt, auf welchem Datensatz der Cursor steht. Dessen Datensatznummer wird in *FNummer* zwischengespeichert. Nach dem Öffnen wird der Cursor wieder auf diesen Datensatz gesetzt.

```
procedure TForm1.BitBtn4Click(Sender: TObject);
// Aktualisieren
begin
  Screen.Cursor := crSQLWait;
  try
    with Query1 do
    begin
      if (Sender = BitBtn4) or (Sender = BitBtn1)
        then FNummer := FieldByName('Nummer').AsInteger;
      Close;
      Open;
      if FNummer > 0
        then Locate('Nummer', Variant(FNummer), []);
    end; {with Query1 do}
  finally
    Screen.Cursor := crDefault;
  end; {try}
  FNummer := 0;
end; {procedure TForm1.BitBtn4Click}
```

Löschen

Zum Löschen des Datensatzes wird die folgende Prozedur verwendet. Beachten Sie, dass der Aufwand für die Zeitmessung in der Regel entfallen dürfte.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
// Löschen
var
  c, t1, t2: int64;
```

```
begin
  if MessageDlg('Datensatz '
       + Query1.FieldByName('Nummer').AsString
       + ' (' + Query1.FieldByName('Vorname').AsString
       + ' ' + Query1.FieldByName('Nachname').AsString + ') '
       + 'wirklich löschen ?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes then
  begin
    QueryPerformanceFrequency(c);
    QueryPerformanceCounter(t1);
    Database1.Execute('DELETE FROM testadr '
      + 'WHERE (nummer='
      + Query1.FieldByName('nummer').AsString + ')');
    QueryPerformanceCounter(t2);
    Label1.Caption := IntToStr(1000 * (t2 - t1) div c) + ' ms';
    Query1.Prior;
    BitBtn4Click(Sender);
  end; {if MessageDlg('Datensatz '...}
end; {procedure TForm1.BitBtn1Click}
```

Vor dem Löschen des Datensatzes erfolgt eine Sicherheitsabfrage. Wird diese mit JA geschlossen, erfolgt die Löschung des Datensatzes. Bevor die Datenmenge aktualisiert wird, wird der Cursor auf den vorangehenden Datensatz gesetzt.

Wird statt *TDatabase.Execute* die Komponente *TQuery* verwendet, so wird der im vorherigen Listing unterstrichene Teil durch folgende Passage ersetzt:

```
  Query2.SQL.Text := 'DELETE FROM testadr '
    + 'WHERE (nummer='
    + Query1.FieldByName('nummer').AsString + ')';
  Query2.ExecSQL;
```

Wie Sie sehen, ist der Aufwand bei *TDatabase.Execute* hier etwas geringer.

Einfügen

Zum Einfügen von Datensätzen wird ein eigenes Formular geöffnet, das im folgenden Abschnitt besprochen wird.

```
procedure TForm1.BitBtn2Click(Sender: TObject);
// Einfügen
var
  s: string;
  Para: TParams;
  c, t1, t2: int64;
```

```
begin
  Form2.Einfuegen;
  if Form2.ShowModal = mrOK then
  begin
    QueryPerformanceFrequency(c);
    QueryPerformanceCounter(t1);
    s := 'INSERT INTO testadr (vorname, nachname, '
      + 'straße, plz, ort, tel1, tel2, fax) '
      + 'VALUES (:vorname, :nachname, '
      + ':strasse, :plz, :ort, :tel1, :tel2, :fax)';
    Para := TParams.Create;
    try
      Para.CreateParam(ftString, 'Vorname', ptInput).AsString
        := Form2.Vorname;
      Para.CreateParam(ftString, 'Nachname', ptInput).AsString
        := Form2.Nachname;
          ...
      Para.CreateParam(ftString, 'Fax', ptInput).AsString
        := Form2.Fax;
      Database1.Execute(s, Para);
    finally
      Para.Free;
      QueryPerformanceCounter(t2);
      Label1.Caption := IntToStr(1000 * (t2 - t1) div c) + ' ms';
    end; {try}
    BitBtn4Click(nil);
    Query1.Last;
  end; {if Form2.ShowModal = mrOK}
end; {procedure TForm1.BitBtn2Click}
```

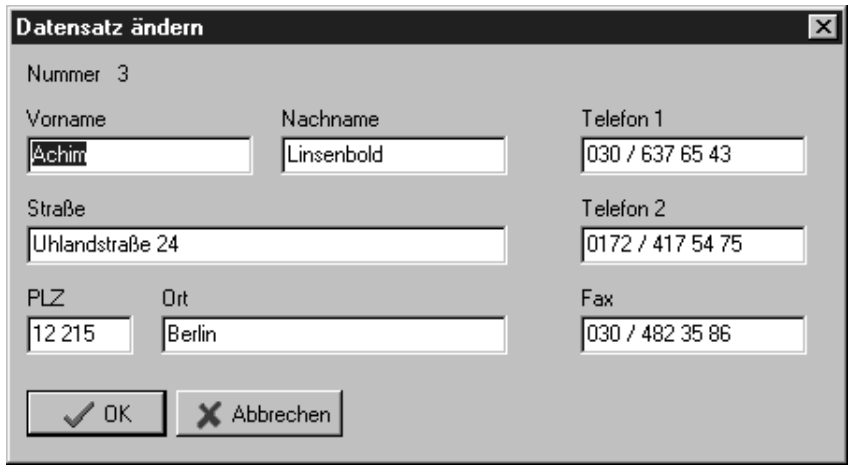
Die *TForm2*-Methode *Einfuegen* wird im nächsten Abschnitt besprochen. Für die Parameter der SQL-Anweisung muss hier eine *TParams*-Instanz erzeugt werden; dieser Aufwand entfällt bei der Verwendung von *TQuery*:

```
    ...
  Query2.SQL.Text := 'INSERT INTO testadr (vorname, nachname, '
    + 'straße, plz, ort, tel1, tel2, fax) '
    + 'VALUES (:vorname, :nachname, '
    + ':strasse, :plz, :ort, :tel1, :tel2, :fax)';
  Query2.ParamByName('Vorname').AsString
    := Form2.Vorname;
```

```
  Query2.ParamByName('Nachname').AsString
    := Form2.Nachname;
      ...
  Query2.ParamByName('Fax').AsString
    := Form2.Fax;
  Query2.ExecSQL;
    ...
```

Das Formular zur Dateneingabe

Für die Dateneingabe wurde ein eigenes Formular erstellt:



Dieses Formular wird sowohl bei der Neueingabe als auch beim Ändern von Daten verwendet.

```
procedure TFrmAdressenAendern.Einfuegen;
var
  i: integer;
begin
  for i := 1 to 8
    do TEdit(FindComponent('Edit' + IntToStr(i))).Text := '';
  BitBtn1.Enabled := false;
  Caption := 'Datensatz einfügen';
  Label1.Caption := '';
  Label2.Caption := '';
end;
```

Wird ein Datensatz neu eingegeben, dann werden zunächst alle *TEdit*-Instanzen geleert. Da zu diesem Zeitpunkt noch keine Datensatznummer vorhanden ist, wird *Label1* und *Label2* jeweils ein leerer Text zugewiesen, außerdem wird die Titelzeile des Formulars entsprechend beschriftet.

```
procedure TFrmAdressenAendern.Aendern(Nummer: integer);
begin
  Caption := 'Datensatz ändern';
  BitBtn2.Enabled := true;
  if Nummer >= 0 then
  begin
    Label1.Caption := 'Nummer';
    Label2.Caption := IntToStr(Nummer);
  end
  else
  begin
    Label1.Caption := '';
    Label2.Caption := '';
  end;
end;
```

Beim Ändern der Datensätze wird die Nummer als Parameter der Prozedur *Aendern* übergeben und dann angezeigt.

```
procedure TFrmAdressenAendern.Edit2Change(Sender: TObject);
begin
  BitBtn1.Enabled := (Length(Edit2.Text) > 1);
end;
```

Damit der Ok-Button verwendet werden kann, muss ein Nachname von mindestens zwei Zeichen vorhanden sein.

```
type
  TFrmAdressenAendern = class(TForm)
      ...
  private
    function GetVorname: string;
    procedure SetVorname(const Value: string);
      ...
  public
    procedure Einfuegen;
    procedure Aendern(Nummer: integer);
    property Vorname: string read GetVorname write SetVorname;
```

```
    property Nachname: string read GetNachname write SetNachname;
      ...
    property Fax: string read GetFax write SetFax;
  end;
```

Für jede Datenbankspalte, die angezeigt werden soll, implementiert *TForm2* eine Eigenschaft, mit deren Hilfe die Werte angezeigt und ausgelesen werden können. Dies ermöglicht es, das Formular später komplett anders aufzubauen, ohne dass andere Units davon betroffen sind.

```
function TFrmAdressenAendern.GetVorname: string;
begin
  result := Edit1.Text;
end;

procedure TFrmAdressenAendern.SetVorname(const Value: string);
begin
  Edit1.Text := Value;
end;
```

Dank der Klassenvervollständigung (SHIFT+CTRL+C) sind die Lese- und Schreibmethoden schnell erstellt.

Ändern

Das Ändern der Datensätze gleicht weitgehend dem Einfügen – mit dem Unterschied, dass die bisherigen Feldwerte vorher in die Eingabemaske geschrieben werden.

```
procedure TForm1.BitBtn3Click(Sender: TObject);
// Ändern
var
  s: string;
  Para: TParams;
  c, t1, t2: int64;
begin
  Form2.Vorname
    := Query1.FieldByName('Vorname').AsString;
      ...
  Form2.Fax
    := Query1.FieldByName('Fax').AsString;
  FNummer := Query1.FieldByName('Nummer').AsInteger;
  Form2.Aendern(FNummer);
```

```
  if Form2.ShowModal = mrOK then
  begin
    QueryPerformanceFrequency(c);
    QueryPerformanceCounter(t1);
    s := 'Update testadr '
      + 'SET vorname = :vorname, '
      + 'nachname = :nachname, '
        ...
      + 'fax = :fax '
      + 'WHERE nummer = :nummer';
    Para := TParams.Create;
    try
      Para.CreateParam(ftString, 'Vorname', ptInput).AsString
        := Form2.Vorname;
          ...
      Para.CreateParam(ftString, 'Fax', ptInput).AsString
        := Form2.Fax;
      Para.CreateParam(ftInteger, 'Nummer', ptInput).AsInteger
        := FNummer;
      Database1.Execute(s, Para);
    finally
      Para.Free;
      QueryPerformanceCounter(t2);
      Label1.Caption := IntToStr(1000 * (t2 - t1) div c) + ' ms';
    end; {try}
    BitBtn4Click(nil);
  end; {if Form2.ShowModal = mrOK}
end; {procedure TForm1.BitBtn3Click}
```

Die Nummer des Datensatzes wird in *FNummer* geschrieben, so dass nach dem Aktualisieren der Cursor wieder auf dem geänderten Datensatz steht.

Bei der Verwendung von *TQuery* kann wieder auf das Erzeugen einer *TParams*-Instanz verzichtet werden:

```
    ...
  Query2.SQL.Text := 'Update testadr '
    + 'SET vorname = :vorname, '
    + 'nachname = :nachname, '
      ...
    + 'fax = :fax '
    + 'WHERE nummer = :nummer';
```

```
Query2.ParamByName('Vorname').AsString
  := Form2.Vorname;
    ...
Query2.ParamByName('Fax').AsString
  := Form2.Fax;
Query2.ParamByName('Nummer').AsInteger
  := FNummer;
Query2.ExecSQL;
  ...
```

Fazit

Verwendet eine SQL-Anweisung keine Parameter, dann ist die Programmierung mittels *TDatabase.Execute* etwas prägnanter als die mit *TQuery*. Muss jedoch erst eine *TParams*-Instanz erstellt werden, dann geht es mit *TQuery* einfacher.

Die Unterschiede in den Ausführungszeiten liegen deutlich unterhalb der Streuung der Messwerte. Es ist zwar zu vermuten, dass *TDatabase.Execute* etwas schneller ist, doch dieser Unterschied ist vollkommen vernachlässigbar.

4.5.3 Referenz TDatabase

Verbindung zur Datenbank

- AliasName, DatabaseName (Eigenschaften, veröffentlicht)

  ```
  property AliasName: string;
  property DatabaseName: string;
  ```

  Setzen Sie die Eigenschaft *AliasName* auf den Namen des BDE-Aliases, mit dem die *TDatabase*-Komponente verbunden werden soll.

  Als *DatabaseName* geben Sie den »internen Alias« an. Auf diesen Wert ist die Eigenschaft *DatabaseName* der verbundenen Datenmengenkomponenten zu setzen. (Wenn Sie einen internen Alias verwenden, der bei alphabetischer Sortierung weit oben steht – beispielsweise AA_INTERN –, dann ist er in der Auswahlliste leicht zu finden.)

  Ändern Sie diese Eigenschaften nur, wenn *Connected* den Wert *false* hat.

- Connected (Eigenschaft, veröffentlicht), Open, Close (Methoden)

  ```
  property Connected: Boolean;
  procedure Open;
  procedure Close;
  ```

Um eine Verbindung zur Datenbank aufzubauen, wird die Eigenschaft Connected auf *true* gesetzt.

Sie können auch die Methode *Open* verwenden, um die Verbindung aufzubauen, und *Close*, um sie zu schließen.

- AfterConnect, AfterDisconnect, BeforeConnect, BeforeDisconnect

```
property AfterConnect(Sender: TObject);
property AfterDisconnect(Sender: TObject);
property BeforeConnect(Sender: TObject);
property BeforeDisconnect(Sender: TObject);
```

Diese Ereignisse treten vor beziehungsweise nach dem Erstellen oder Trennen der Datenbankverbindung auf.

- OnLogin (Ereignis)

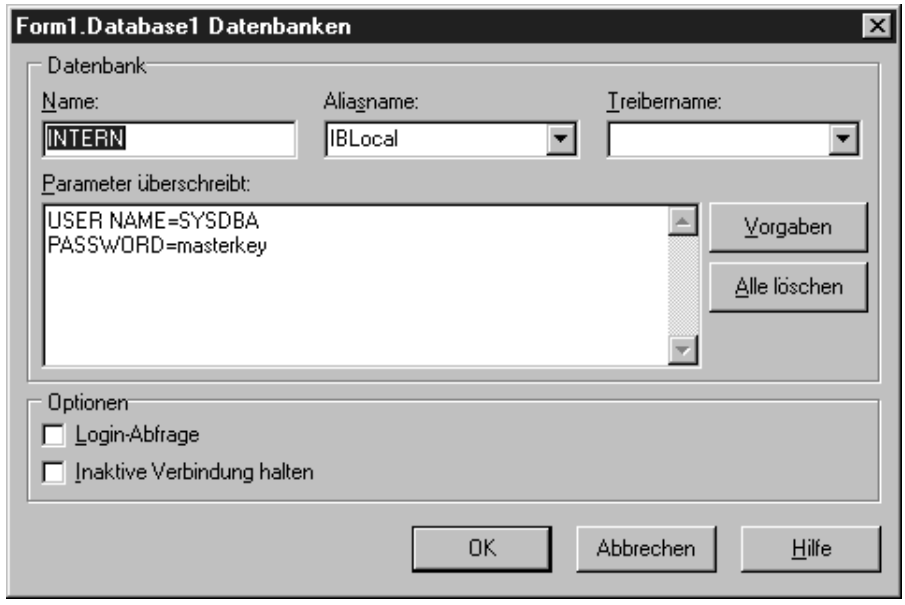
```
property OnLogin(Database: TDatabase; LoginParams: TStrings);
```

Das Ereignis *OnLogin* tritt beim Verbinden mit einer Datenbank auf.

- LoginPrompt, Params (Eigenschaften, veröffentlicht)

```
property LoginPrompt: Boolean;
property Params: TStrings;
```

Beide Eigenschaften können Sie auch mittels des Komponenteneditors setzen:



Mit *LoginPrompt* können Sie einstellen, ob ein Anmeldedialog angezeigt werden soll. Möchten Sie einen solchen Dialog vermeiden, obwohl der Datenbank-Server ein Passwort verlangt, müssen Sie das Passwort mit Hilfe der Eigenschaft *Params* eingeben.

Mit Hilfe der Eigenschaft *Params* können Sie Parameter eingeben, welche die Verbindung zu einem SQL-Server steuern und die entsprechenden BDE-Einstellungen ersetzen. Meist wird *Params* dazu verwendet, ein Passwort anzugeben, wenn *LoginPrompt* auf *false* gesetzt ist.

- KeepConnection (Eigenschaft, veröffentlicht)

```
property KeepConnection: Boolean;
```

Hat *KeepConnection* den Wert *true*, dann bleibt eine Datenbankverbindung auch dann geöffnet, wenn alle beteiligten Datenmengenkomponenten geschlossen werden. Dies stellt sicher, dass beim Öffnen einer dieser Datenmengenkomponenten nicht erst eine Datenbankverbindung geöffnet werden muss.

- SessionName (Eigenschaft, veröffentlicht),
  Session (Eigenschaft, öffentlich, nur Lesen)

```
property SessionName: string;
property Session: TSession;
```

Verwenden die mit der *TDatabase*-Instanz verbundenen Datenmengenkomponenten eine andere Session als die Standard-Session, so muss *SessionName* auf den Namen dieser Session gesetzt werden.

Beachten Sie, dass alle mit einer bestimmten *TDatabase*-Komponente verbundenen Datenmengenkomponenten dieselbe Session haben müssen wie diese *TDatabase*-Instanz. Gegebenenfalls müssen Sie zusätzliche *TDatabase*-Komponenten einfügen. Mit der Eigenschaft *Session* erhalten Sie einen Zeiger auf die *TSession*-Instanz, mit der die *TDatabase*-Instanz verbunden ist.

Verbindung zu den Datenmengenkomponenten

- DataSets, DataSetCount (Eigenschaften, öffentlich, nur Lesen)

```
property DataSets[Index: Integer]: TDBDataSet;
property DataSetCount: Integer;
```

Mit der Array-Eigenschaft *DataSets* erhält man Zeiger auf alle verbundenen Datenmengenkomponenten. Deren Anzahl kann mit *DataSetCount* ermittelt werden.

Die folgenden Anweisungen aktualisieren alle Datenmengenkomponenten nach einem Rollback. (Die Transaktionssteuerungsanweisungen lösen leider keine entsprechenden Ereignisse aus.)

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: integer;
begin
  with Database1 do
  begin
    Rollback;
    for i := 0 to DataSetCount - 1 do
    begin
      if DataSets[i] is TTable
        then TTable(DataSets[i]).Refresh;
      if DataSets[i] is TQuery then
      begin
        TQuery(DataSets[i]).Close;
        TQuery(DataSets[i]).Open;
      end;
    end; {for i := 0 to DataSetCount - 1 do}
  end; {with Database1 do}
end; {procedure TForm1.Button1Click}
```

- CloseDatasets (Methode)

  ```
  procedure CloseDatasets;
  ```

  Mit *CloseDatasets* werden alle Datenmengenkomponenten geschlossen, die mit der *TDatabase*-Instanz verbunden sind.

Transaktionen

Alle Datenbankaktionen auf SQL-Servern werden im Kontext von so genannten Transaktionen ausgeführt. Alle Anweisungen einer Transaktion werden entweder gemeinsam ausgeführt oder gemeinsam verworfen. Auf diese Weise kann die Konsistenz der Daten sichergestellt werden.

Wird nicht explizit eine Transaktion gestartet, dann erstellt Delphi für jede einzelne Datenbankaktion eine Transaktion. Der damit verbundene Verwaltungsaufwand kann vermindert werden, wenn explizit Transaktionen gestartet werden.

- StartTransaction, Commit, Rollback (Methoden)

```
procedure StartTransaction;
procedure Commit;
procedure Rollback;
```

Mit *StartTransaction* wird eine Transaktion begonnen, mit *Commit* wird sie bestätigt – es werden dann alle Aktionen ausgeführt. Sollen alle Aktionen verworfen werden, ist die Transaktion mit *Rollback* zu beenden.

- TransIsolation (Eigenschaft, veröffentlicht)

```
property TransIsolation: (tiDirtyRead, tiReadCommitted,
  tiRepeatableRead);
```

Mit *TransIsolation* kann bestimmt werden, inwieweit die Datenänderungen verschiedener Transaktionen voneinander isoliert werden. Bei *tiDirtyRead* können auch die Änderungen von anderen, noch nicht abgeschlossenen Transaktionen gelesen werden.

Hat *TransIsolation* den Wert *tiReadCommited*, dann können die Änderungen, die andere Transaktionen vorgenommen haben, nur dann gelesen werden, wenn diese Transaktionen mit COMMIT bestätigt wurden.

Am meisten sind die Transaktionen bei *tiRepeatableRead* voneinander isoliert. Bei dieser Einstellung können Änderungen anderer Transaktionen nicht gelesen werden, auch wenn diese mit COMMIT bestätigt wurden. Die Sicht des Datenbestandes bleibt exakt so, wie er zu Beginn der Transaktion gewesen ist.

- InTransaction (Eigenschaft, öffentlich, nur Lesen)

```
property InTransaction: Boolean;
```

Die Eigenschaft *InTransaction* hat den Wert *true*, wenn gerade eine explizite Transaktion gestartet und noch nicht beendet ist.

Execute

- Execute (Methode)

```
function Execute(const SQL: string; Params: TParams = nil;
  Cache: Boolean = False; Cursor: phDBICur = nil): Integer;
```

Mit der Methode *Execute* können von *TDatabase* aus SQL-Anweisungen abgesetzt werden. Dies lohnt sich jedoch nur bei Anweisungen ohne Parameter, die zudem keine Ergebnismenge liefern. Ein Beispiel dazu finden Sie in Kapitel 4.5.2.

5 TTable

Die Komponente *TTable* dient dem Zugriff auf eine einzelne Tabelle (oder – bei Client-Server-Systemen – auf eine einzelne VIEW). *TTable* ist für den Zugriff auf lokale Datenbanken – insbesondere dBase und Paradox – hin optimiert. Sie kann zwar auch mit Client-Server-Systemen eingesetzt werden, aber bei manchen Datenbanksystemen führt dies zu einer miserablen Performance.

5.1 Erstellen von Testdaten

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. Der Versuch, solch eine Datenbank selbst einzugeben, dürfte sich allein aus Zeitgründen verbieten.

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

5.1.1 Erstellen der Tabellen

Zunächst benötigen wir drei Tabellen. Aus den Tabellen *Namen.DB* und *Orte.DB* beziehen wir die Daten, in *Testadr.DB* fügen wir die generierten Testdatensätze ein.

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 durcheinander gemischt:

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

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

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

Die Tabelle *Testadr.DB* benötigt keine Einträge, denn diese wollen wir ja generieren:

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	

Ein Programm zur Erstellung von Testdatensätzen geht normalerweise nicht an die Kunden, von daher können die Ansprüche an die Benutzeroberfläche bescheidener ausfallen. Wir brauchen hier die Darstellung der drei beteiligten Tabellen, wobei es eigentlich egal ist, ob wir dafür *TDBGrid* oder *TDBEdit* verwenden:

Mit der folgenden Prozedur wird eine einzelne Testadresse erstellt. Ob ihrer Länge wollen wir alle lokalen Prozeduren einzeln besprechen:

```
procedure TForm1.btnAdresseClick(Sender: TObject);
var
  s: string;

  ...

begin
  Table3.Append;
  DoTable1;
  DoTable2;
  Table3Tel1.AsString := GetNummer;
  if random(3) = 1
    then Table3Fax.AsString := GetNummer;
  DoHandy;
  Table3.Post;
end; {procedure TForm1.btnAdresseClick}
```

Mit *Table3.Append* wird ein neuer Datensatz hinzugefügt, der mit *Post* in die Datenbank geschrieben wird. Aus *Table1* kommen die Daten für Vor- und Nachname sowie die Straße, aus *Table2* die Postleitzahl, der Ort und die Vorwahl.

Mit *GetNummer* wird eine Telefonnummer ermittelt, in manchen Fällen wird mit derselben Routine auch noch eine Faxnummer ermittelt. In einige Datensätze wird auch noch eine Handynummer geschrieben.

```
procedure DoRandom(ATable: TTable);
begin
  with ATable
    do FindKey([random(RecordCount) + 1]);
end;

procedure DoTable1;
{Vornamen, Nachnamen, Straße und Hausnummer}
begin
  DoRandom(Table1);
  Table3Vorname.AsString := Table1Vorname.AsString;
  DoRandom(Table1);
  Table3Nachname.AsString := Table1Nachname.AsString;
  DoRandom(Table1);
  s := Table1Strae.AsString;
  s := s + ' ' + IntToStr(random(500) + 1);
  Table3Strae.AsString := s;
end; {procedure DoTable1}
```

Mit der lokalen Prozedur *DoRandom* wird ein zufälliger Datensatz aus der angegebenen Datenmenge gewählt. Damit diese Prozedur zu korrekten Ergebnissen kommt, müssen in der Tabelle *Namen.DB* immer alle drei Datenspalten gefüllt sein.

In *DoTable1* wird somit ein zufällig ausgewählter Vorname mit einem ebenso zufällig ausgewählten Nachnamen sowie einer Straße kombiniert. Der Straße wird eine ebenfalls per Zufallsgenerator erstellte Hausnummer angehängt.

```
procedure DoTable2;
{PLZ, Ort, Vorwahl}
begin
  DoRandom(Table2);
  Table3PLZ.AsString := Table2PLZ.AsString;
  Table3Ort.AsString := Table2Ort.AsString;
  s := Table2Vorwahl.AsString;
end; {procedure DoTable2}
```

Die Postleitzahl, der Ort sowie die Vorwahl müssen zusammenpassen, deswegen wird hier *DoRandom* nur ein einziges Mal aufgerufen.

```
function GetNummer: string;
var
  i, l: integer;
  Leerzeichen: set of Byte;
begin
  l := length(s);
  case l of
    3: Leerzeichen := [4, 7];
    4: Leerzeichen := [4];
    5: Leerzeichen := [3];
  end;
  result := '';
  for i := 1 to (11 - l + integer(l = 3)) do
  begin
    if i > 1 then
    begin
    if i in Leerzeichen
      then result := result + ' '
      else result := result + IntToStr(Random(10))
    end
    else result := result + IntToStr(Random(9) + 1)
  end; {for i := 1 to (13 - l - integer(l = 3)) do}
  result := s + ' / ' + result;
end; {function GetNummer}
```

Für die Telefonnummer müssen »eigentlich« nur ein paar Ziffern zufällig generiert und zusammengefügt werden. Die Funktion *GetNummer* fällt dann doch ein wenig komplizierter aus:

- Ja länger die Vorwahl, deso kürzer die zu erstellende Nummer.
- Ja nach Länge der zu erstellenden Nummer wird diese ein- oder zweimal durch ein Leerzeichen unterbrochen, die Leerzeichen stehen auch nicht immer an derselben Stelle.
- Als erste Ziffer darf keine Null verwendet werden, bei den folgenden Ziffern schon.

Die Stellen, an denen die Nummer mit einem Leerzeichen unterbrochen werden soll, werden in der Menge *Leerzeichen* gespeichert. Diese Mengenvariable wird in einer *case*-Verzweigung gefüllt.

Dem Einsteiger mag die Berechnung der Schleifendurchlaufzahl etwas verwirrend vorkommen: Hat die Vorwahl vier Stellen, dann hat die Durchwahl sechs Stellen, eine fünfstellige Vorwahl ergänzen wir mit einer ebenfalls fünfstelligen Durchwahl – hinzu kommt jeweils ein Leerzeichen. Bei einer dreistelligen Vorwahl haben wir jedoch nicht nur eine siebenstellige Durchwahl, sondern auch noch ein zweites Leerzeichen. Bei einer dreistelligen Vorwahl hat der Ausdruck *l = 3* den Wert *true*, die Umwandlung nach *integer* führt zur Zahl *1*.

```
procedure DoHandy;
begin
  case random(9) of
    5: s := '161';
    6: s := '171';
    7: s := '172';
    8: s := '177';
    else s := '';
  end;
  if s <> ''
    then Table3Tel2.AsString := '0' + GetNummer;
end; {procedure DoHandy}
```

Wollen wir für die Erstellung der Handynummer auch die Funktion *GetNummer* verwenden, dann müssen wir ein wenig »in die Trickkiste greifen«. Handyvorwahlen sind vierstellig, damit würde *GetNummer* eine sechsstellige Durchwahl generieren, wir benötigen jedoch eine siebenstellige.

Deshalb entfernen wir zunächst die führende Null der Vorwahl und fügen diese beim Schreiben in die Datenbank wieder hinzu.

Mehrere Datensätze einfügen

Nun möchte man zum Erstellen der Testdaten auch nicht einige tausend Mal auf *btnAdresse* klicken. Deshalb verwenden wir hier eine Schleife, die gleich 10000 Datensätze auf einen Schlag einfügt.

```
procedure TForm1.btnZehntausendClick(Sender: TObject);
var
  i: integer;
begin
  if not Quelldatenbankenanzeigen1.Checked then
  begin
    Table1.DisableControls;
    Table2.DisableControls;
    Table3.DisableControls;
  end;
```

```
  try
    for i := 1 to 100 do
    begin
      btnAdresseClick(Sender);
      Application.Processmessages;
    end; {for i := 1 to 100 do}
  finally
    Table1.EnableControls;
    Table2.EnableControls;
    Table3.EnableControls;
  end;
end; {procedure TForm1.BitBtn2Click}
```

Zur Beschleunigung des Ablaufs lässt sich die Anzeige per *DisableControls* vorübergehend abschalten, dazu dient der Menüpunkt *Quelldatenbankenanzeigen1*.

5.2 Master-Detail-Verknüpfungen

Sie erinnern sich vielleicht noch an Kapitel 1.3, in dem wir eine zweistufige Master-Detail-Verknüpfung mittels *TQuery* erstellt hatten. Wir wollen nun dieses Projekt mit *TTable* nachbauen.

Anzeige der Kunden

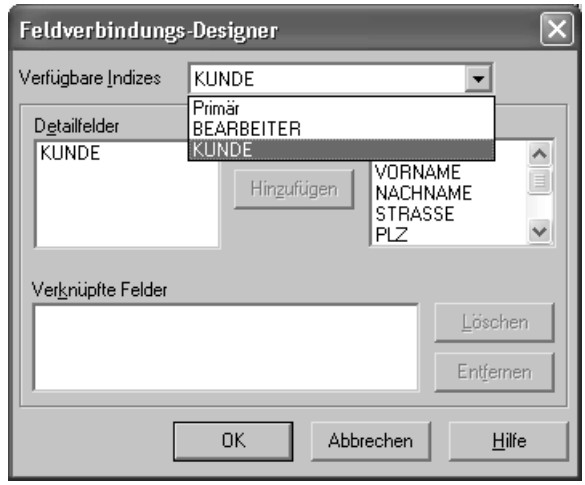
Zur Anzeige der Kunden legen Sie eine *TTable*- und eine *TDataSource*-Komponente auf das Datenmodul und geben diesen die Namen *tblKunde* und *dsKunde*. Als *DatabaseName* verwenden Sie *Buch*, als *TableName t_kunde.db*. Hängen Sie an *tblKunde* die Komponente *dsKunde* und daran wiederum geeignete Datensteuerungskomponenten:

Zu den Buttons *Open* und *Next* kommen wir später.

5.2.1 Anzeige der Bestellungen

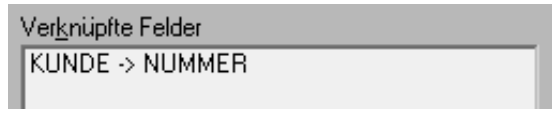
Für die Anzeige der Bestellungen brauchen wir eine weitere *TTable-TDataSource*-Kombination (*tblBestellung* / *dsBestellung*). Als *TableName* verwenden wir *t_bestellung.db*.

Um eine Master-Detail-Verknüpfung zu erstellen, setzen wir die Eigenschaft *MasterSource* auf *dsKunde*, anschließend rufen wir den *Feldverbindungs-Designer* auf, der als Eigenschaftseditor der Eigenschaft *MasterFields* fungiert.



Solange wir mit dBase oder Paradox arbeiten, sind die verfügbaren Felder diejenigen Felder, aus denen sich der aktuell gewählte Index zusammensetzt. Folglich müssen wir den Index *Kunde* wählen.

Markieren Sie anschließend bei den Detailfeldern *Kunde* und bei den Hauptfeldern *Nummer*, und klicken Sie auf den Button *Hinzufügen*. In der Liste der verknüpften Felder wird diese Kombination angezeigt:

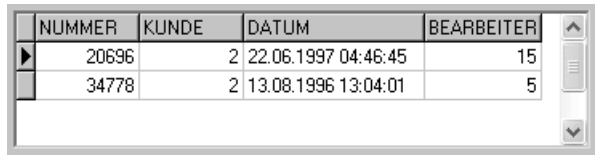


Wird nun der Feldverbindungs-Designer mit OK geschlossen, dann ist die Master-Detail-Verknüpfung erstellt.

Noch eine kleine Anmerkung: Bei Client-Server-Systemen ist kein Index für die Detailspalte erforderlich, er kann dort auch gar nicht gewählt werden.

Erstellen einer Nachschlagespalte

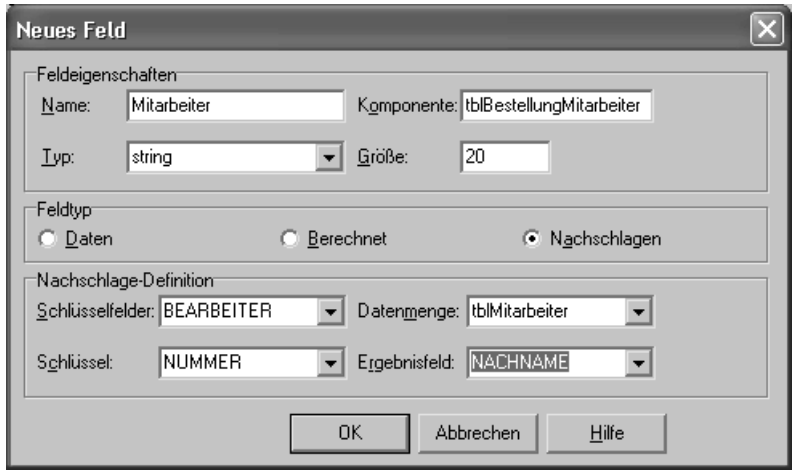
Es werden nun korrekt die zum jeweiligen Kunden gehörenden Bestellungen angezeigt, aber die Darstellung dieser Datensätze lässt noch etwas zu wünschen übrig – wir kennen das ja aus Kapitel 1.3:



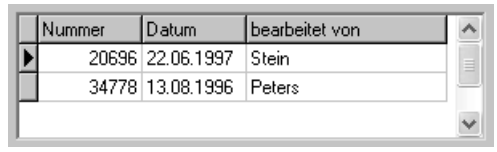
NUMMER	KUNDE	DATUM	BEARBEITER
20696	2	22.06.1997 04:46:45	15
34778	2	13.08.1996 13:04:01	5

Legen wir zunächst persistene *TField*-Instanzen an, sorgen für vernünftige Groß- und Kleinschreibung bei den Spaltenüberschriften und verwenden *dd.mm.yyyy* als *DisplayFormat* der Spalte *Datum*, die dann auch etwas schmaler werden darf. Dadurch, dass wir *Visible* auf *false* setzen, blenden wir die Spalte *Kunde* aus, die Spalte *Bearbeiter* werden wir nachher auf dieselbe Weise los.

In Kapitel 1.3 haben wir den Mitarbeiternamen mittels eines JOINS in die Ergebnismenge aufgenommen. Diese Möglichkeit steht uns hier nicht zur Verfügung, stattdessen verwenden wir ein Nachschlage-Feld:



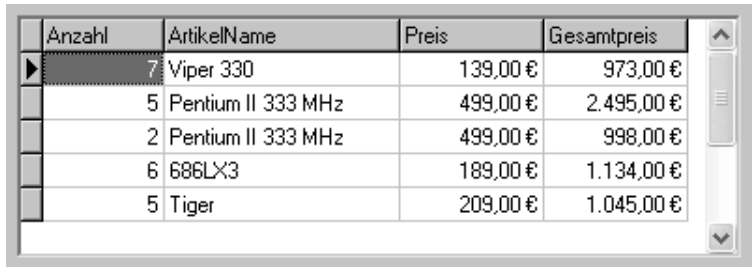
Zu diesem Zweck benötigen wir eine neue TTable-Komponente (*tblMitarbeiter*), diesmal ohne *TDataSource*. *tblMitarbeiter* greift auf die Tabelle *t_mitarbeiter.db* zu. Die Eigenschaft *DisplayLabel* der neuen Spalte setzen wir auf *bearbeitet von*:



Nummer	Datum	bearbeitet von
20696	22.06.1997	Stein
34778	13.08.1996	Peters

5.2.2 Anzeige der Posten

Für die Anzeige der Posten verwenden wir die Komponenten *tblPosten* und *dsPosten*, die zugrunde liegende Tabelle ist *t_posten.db*. Als Nachschlagetabelle für den Artikelnamen verwenden wir *tblArtikel*, die auf *t_artikel.db* zugreift.



Anzahl	ArtikelName	Preis	Gesamtpreis
7	Viper 330	139,00 €	973,00 €
5	Pentium II 333 MHz	499,00 €	2.495,00 €
2	Pentium II 333 MHz	499,00 €	998,00 €
6	686LX3	189,00 €	1.134,00 €
5	Tiger	209,00 €	1.045,00 €

Für den Gesamtpreis wird ein berechnetes Feld verwendet, dementsprechend muss auf das *tblPosten*-Ereignis *OnCalcFields* reagiert werden:

```
procedure TDataModule2.tblPostenCalcFields(DataSet: TDataSet);
begin
  tblPostenGESAMTPREIS.AsCurrency
    := tblPostenSTUECKZAHL.AsInteger * tblPostenPREIS.AsCurrency;
end;
```

5.2.3 Messung der Ausführungsgeschwindigkeit

Um *TTable* und *TQuery* vergleichen zu können, wollen wir das Öffnen der Tabellen erst zur Laufzeit vornehmen und dabei die benötigte Zeit messen:

```
procedure TForm1.btnOpenClick(Sender: TObject);
var
  c, t1, t2: int64;
begin
  QueryPerformanceFrequency(c);
  QueryPerformanceCounter(t1);
  with DataModule2 do
  begin
    tblKunde.Open;
    tblMitarbeiter.Open;
    tblBestellung.Open;
    tblArtikel.Open;
```

```
    tblPosten.Open;
  end;
  QueryPerformanceCounter(t2);
  Label1.Caption := IntToStr(1000 * (t2 - t1) div c) + ' ms';
end;
```

Mit entsprechenden Anweisungen messen wir die Zeit zur Erledigung von *tblKunde.Next*. Die dabei gemessenen Ergebnisse sind absolut gesehen natürlich stark von der Geschwindigkeit des Rechners abhängig – sie sind nur im Vergleich miteinander aussagekräftig. Zudem schwanken die Werte etwas und wurden für die nachfolgende Übersicht gemittelt:

- Das Öffnen der Tabellen ist in 67 ms erledigt, für das Scrollen werden etwa 3 ms gebraucht.
- Würde man auf die Nachschlagetabellen verzichten, wären die Tabellen in etwa 42 ms geöffnet, das Scrollen würde etwa 2 ms in Anspruch nehmen.
- Beim Umstellen dieses Projektes auf *TQuery* wurde zunächst auf die Nachschlagetabellen verzichtet. Hat hier *RequestLive* (gemäß der Voreinstellung) den Wert *false*, dann dauert das Öffnen 282 ms, das Scrollen 140 ms. Setzt man *RequestLive* auf *true*, dann ist das Öffnen schon in 73 ms erledigt, das Scrollen dauert dann nur noch 30 ms.
- Werden – wie bei *TTable* – Nachschlagetabellen verwendet, dann dauert (*RequestLive=false*) das Öffnen 379 ms, das Scrollen 140 ms. Setzt man *RequestLive* auf *true*, dann braucht das Öffnen 82 ms, das Scrollen nur noch 17 ms.

  Es mag verwundert, dass der Einsatz von Nachschlagetabellen das Scrollen beschleunigt, die Zahlen wurden jedoch mehrmals nachgemessen und haben sich als »belastbar« erwiesen.
- Gar nicht mehr gut sieht *TQuery* aus, wenn man JOINS statt Nachschlagetabellen einsetzt: Das Öffnen dauert dann 906 ms beziehungsweise 749 ms, das Scrollen 762 ms beziehungsweise 682 ms – die kürzeren Werte jeweils dann, wenn *RequestLive* den Wert *true* hat.

Fazit

- Bei lokalen Datenbanken arbeitet *TTable* grundsätzlich schneller – schließlich ist diese Komponente auch dafür entwickelt worden. Bei Client-Server-Systemen ist *TQuery* schneller.
- Bei lokalen Datenbanken sollte man die *TQuery*-Eigenschaft *RequestLive* auf *true* setzen und lieber Nachschlagetabellen als JOINS verwenden – auch diese Empfehlungen sind auf Client-Server-Systeme nicht zu übertragen.

5.3 Tabellen zur Laufzeit erstellen

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.

Außerdem benötigen wir ein Menü mit den in der folgenden Abbildung zu sehenden Menüpunkten:

Erstellen von Tabellen

Zum Erstellen der Tabelle sind folgende Schritte nötig:

```
procedure TForm1.N1Tabelleerstellen1Click(Sender: TObject);
begin
  with Table1 do
  begin
    Active := false;
    DataBaseName := 'test';
    TableName := 'test_1.db';
    TableType := ttParadox;
```

Bevor Tabellenname und Alias vorgegeben werden, muss die Eigenschaft *Active false* sein, ansonsten würden sich die Eigenschaften *DataBaseName* und *TableName* nicht verändern lassen. Die Eigenschaft *TableType* ist hier optional (auch wenn die Online-Hilfe etwas anderes behauptet), die BDE kann auch aus der Dateiendung erkennen, um welchen Tabellentyp es sich handelt.

```
    with FieldDefs do
    begin
      Clear;
      Add('nummer', ftAutoInc, 0, false);
      Add('vornamen', ftString, 20, false);
      Add('nachnamen', ftString, 20, false);
      Add('straße', ftString, 20, false);
      Add('plz', ftString, 6, false);
      Add('ort', ftString, 20, false);
      Add('tel1', ftString, 20, false);
      Add('tel2', ftString, 20, false);
      Add('fax', ftString, 20, false);
    end;
```

Im nächsten Schritt werden die einzelnen Felder definiert. Die Definition eines selbstinkrementierenden Feldes erfolgt mit dem Feldtyp *ftAutoInc*, alle anderen Felder sind vom Typ *ftString*.

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.

```
    with IndexDefs do
    begin
      Clear;
      Add('Primary', 'nummer', [ixPrimary, ixUnique]);
    end;
    CreateTable;
    Open;
  end; {with table1 do}
end; {procedure 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:

```
procedure TForm1.N2Datensatzeinfgen1Click(Sender: TObject);
begin
  with table1 do
  begin
    Active := false;
    DataBaseName := 'test';
    TableName := 'test_1.db';
    Active := true;
    Append;
    Fields[1].AsString := 'Ludwig';
    Fields[2].AsString := 'Meier';
    Fields[3].AsString := 'Mozartstraße 32';
    ...
    Post;
  end;
end;
```

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 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 (zu den Problemen kommen wir gleich). Anschließend werden die einzelnen Daten der Eigenschaft *Fields* zugewiesen, abschließend wird mit *Post* bestätigt.

Sie können nun beliebig viele Datensätze einfügen. Erfolgt dies jedoch über das Menü, dann werden die sich lediglich in der Spalte *nummer* unterscheiden.

Tabelle entfernen

Das Entfernen der Tabelle ist dann vergleichsweise simpel:

```
procedure TForm1.N3Tabelleentfernen1Click(Sender: TObject);
begin
  with Table1 do
  begin
    Active := false;
    DataBaseName := 'test';
    TableName := 'test_1.db';
    DeleteTable;
  end;
end;
```

5.4 Suchen und Filtern

Um die Zahl der Datensätze zu beschränken oder einen bestimmten Datensatz zu suchen, stehen bei *TTable* einige Methoden zur Verfügung.

5.4.1 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:

```
procedure TForm1.Bereichsetzen1Click(Sender: TObject);
begin
  with Table1 do
  begin
    SetRangeStart;
    FieldByName('nummer').AsInteger := 100;
    SetRangeEnd;
    FieldByName('nummer').AsInteger := 200;
    ApplyRange;
  end;
  N2KeyExclusive1.Checked := false;
end;
```

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. Die folgende Prozedur schaltet zwischen den beiden *KeyExclusive*-Varianten um:

```
procedure TForm1.N2KeyExclusive1Click(Sender: TObject);
begin
  if N2KeyExclusive1.Checked = true then
  begin
  ... {Setzt KeyExclusive auf false, ggf. siehe CD)
  end {if N2KeyExclusive1.Checked = true then}
  else
  begin
    N2KeyExclusive1.Checked := true;
    with Table1 do
    begin
      EditRangeStart;
      FieldByName('nummer').AsInteger
        := FieldByName('nummer').AsInteger;
      KeyExclusive := true;
      EditRangeEnd;
      FieldByName('nummer').AsInteger
        := FieldByName('nummer').AsInteger;
      KeyExclusive := true;
      ApplyRange;
    end;
  end; {else N2KeyExclusive1.Checked = true then}
end; {procedure TForm1.N2KeyExclusive1Click}
```

Etwas kurios mögen hier die beiden Anweisungen erscheinen, welche einer Eigenschaft ihren eigenen Wert zuweisen (den sie somit schon hat); dies ist jedoch nötig, damit der Wert nicht gelöscht wird. Die Eigenschaft *KeyExclusive* muss für jede Bereichsgrenze einzeln gesetzt werden.

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*:

```
procedure TForm1.N3MethodeSetRange1Click(Sender: TObject);
begin
  table1.SetRange([1200], [1300]);
  N2KeyExclusive1.Checked := false;
end;
```

Bereich aufheben

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

5.4.2 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. Die folgende Prozedur findet einen Datensatz in der Mitte des Datenbestandes in etwa 3 ms.

```
procedure TForm1.N1nachNummerindiziert1Click(Sender: TObject);
var
  s: string;
begin
  s := InputBox('Datensatz suchen', 'Nummer:', '5120');
  QueryPerformanceCounter(t1);
  table1.FindKey([s]);
  DoZeit;
end;
```

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.

Der Vollständigkeit halber noch die Routine *DoZeit*:

```
procedure TForm1.DoZeit;
begin
  QueryPerformanceCounter(t2);
  Label1.Caption := IntToStr(1000 * (t2 - t1) div c) + ' ms';
end;
```

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, hier im Beispiel 6500 ms.

```
procedure TForm1.N2nachNachnamennichtindiziert1Click(Sender: TObject);
var
  s: string;
begin
  s := InputBox('Datensatz suchen', 'Nummer:', '5120');
  QueryPerformanceCounter(t1);
  with table1 do
  begin
    First;
    repeat
      Next;
    until FieldByName('Nummer').AsString = s;
  end;
  DoZeit;
end;
```

Zunächst wird zum ersten Datensatz gesprungen, danach wird ein Datensatz nach dem anderen daraufhin geprüft, ob er der Suchbedingung 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. Mit einer leichten Modifikation kann man diese unterbinden und somit die Ausführungszeit auf etwa 55 ms verkürzen.

```
with table1 do
begin
  DataSource1.Enabled := false;
  First;
  repeat
    Next;
  until FieldByName('Nummer').AsString = s;
  DataSource1.Enabled := true;
end;
```

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.

Locate

Seit Delphi 2 implementiert *TDataSet* die Möglichkeit, mit der Funktion *Locate* nach Datensätzen über ihre Felderinhalte zu suchen. Damit erübrigt sich in den meisten Fällen die inkrementale Suche.

Das folgende Beispiel sucht die Adresse von *Bert Heller*. Dank der Option *loCaseInsensitive* wird hier die falsche Groß- und Kleinschreibung ignoriert.

```
procedure TForm1.Locate1Click(Sender: TObject);
begin
  if Table1.Locate('Vorname; Nachname',
       VarArrayOf(['bert', 'Heller']), [loCaseInsensitive]) = false
    then ShowMessage('Datensatz nicht gefunden');
end;
```

5.4.3 Datensätze filtern

Seit Delphi 2 implementiert *TDataSet* die Möglichkeit, Datensätze zu filtern. Dazu stehen die Eigenschaft *Filter* und das Ereignis *OnFilterRecord* zur Verfügung.

Die Eigenschaft Filter

Mittels der Eigenschaft *Filter* kann einer Datenmengenkomponente eine Anweisung zum Filtern der Datensätze zugewiesen werden. Die Syntax der Filteranweisung lehnt sich teilweise an SQL an. Das folgende Beispiel sorgt dafür, dass nur diejenigen Datensätze angezeigt werden, deren Feld *Nummer* einen Wert kleiner sieben hat.

```
procedure TForm1.Nummer1Click(Sender: TObject);
begin
  Table1.Filter := '(nummer < 7)';
  if not Table1.Filtered then Filtern1Click(Sender);
end;

procedure TForm1.Filtern1Click(Sender: TObject);
begin
  Filtern1.Checked := not Filtern1.Checked;
  Table1.Filtered := Filtern1.Checked;
end;
```

Mit Hilfe des Menüpunktes *Filtern* kann zwischen gefilterter und ungefilterter Datenmenge umgeschaltet werden.

String-Felder filtern

Etwas problematischer wird die Geschichte, wenn in String-Feldern gesucht werden soll. Soll beispielsweise die Anzeige auf all diejenigen Datensätze beschränkt werden, deren Spalte *Vorname* den Wert *Bert* hat, dann muss der Eigenschaft Filter folgende Anweisung hinzugefügt werden:

```
Vorname = 'Bert'
```

Im Objektinspektor ist diese Zuweisung nicht weiter problematisch. Soll die Zuweisung jedoch zur Laufzeit geändert werden, dann werden einfache Anführungszeichen als String-Begrenzungszeichen interpretiert.

```
procedure TForm1.Name1Click(Sender: TObject);
var
  s: string;
begin
  s := chr(39);
  Table1.Filter := 'Vorname = ' + s + 'Bert' + s;
  Caption := Table1.Filter;
  if not Table1.Filtered then Filtern1Click(Sender);
end;
```

Um dieses Problem zu umgehen, kann einem String das ASCII-Zeichen 39 zugewiesen und auf diese Weise am Compiler vorbeigemogelt werden.

Platzhalter

Erfreulicher dagegen ist, dass die Verwendung von Platzhaltern erlaubt ist – auch wenn man hier von der SQL-Syntax abgewichen ist.

```
procedure TForm1.Platzhalter1Click(Sender: TObject);
var
  s: string;
begin
  s := chr(39);
  Table1.Filter := 'Vorname = ' + s + 'B*' + s;
  Caption := Table1.Filter;
  if not Table1.Filtered then Filtern1Click(Sender);
end;
```

Anwenderdefinierte Filteranweisungen

Mit Hilfe der Funktion *InputBox* lassen sich Filter-Strings vom Anwender abfragen und der Eigenschaft *Filter* zuweisen. Hier sind auch die Anführungszeichen kein Problem mehr.

```
procedure TForm1.Eingabe1Click(Sender: TObject);
var
  s: string;
begin
  s := InputBox('Datensätze filtern', 'Filter-String:', '');
  Table1.Filter := s;
  if not Table1.Filtered then Filtern1Click(Sender);
end;
```

Logische Verknüpfungen

Die Filteranweisungen erlauben auch logische Verknüpfungen – die folgenden beiden Beispiele müssen wohl nicht näher erläutert werden:

```
(Vorname = 'Bert') OR (Nachname = 'Müller')
(Vorname = 'Bert') AND (Nachname = 'Müller')
```

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 Variablenparameter *Accept* übergeben; wird dieser auf *false* gesetzt, dann wird der betreffende Datensatz nicht aufgenommen.

Um damit verschiedene Filterkriterien zu verwenden, womöglich auch noch mit frei wählbaren Parametern, sind einige Verrenkungen nötig. Das folgende Beispiel zeigt, wie man die Filtermöglichkeiten *Nummern 100 bis 200* und *Nachname gleich Rasemann* implementiert. Bei *Filterwert* handelt es sich dabei um eine globale Variable, welche je nach Filterwunsch gesetzt wird (beispielsweise über das Menü).

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
  var Accept:Boolean);
var
  i: Integer;
  s: string;
begin
```

```
  case filterwert of
  1:
  begin
    i := DataSet.FieldByName('Nummer').AsInteger;
    if (i >= 100) and (i <= 200)
      then Accept := true
      else Accept := false;
  end;
  2:
  begin
    s := DataSet.FieldByName('Nachnamen').AsString;
    if s = 'Rasemann'
      then Accept := true
      else Accept := false;
  end;
  end; {case filtertext of}
end;
```

5.5 Referenz TDBDataSet

Von der Komponente *TDBDataSet* sind die Komponenten *TTable*, *TQuery* und *TStoredProc* abgeleitet. Um unnötige Wiederholungen zu vermeiden, soll hier *TDBDataSet* eigens besprochen werden.

TDBDataSet ist ihrerseits wiederum von *TBDEDataSet* abgeleitet, in der beispielsweise die Funktionalität der *CachedUpdates* implementiert ist. In der Praxis macht es keinen Unterschied, ob ein Element von *TDBDataSet* oder von *TBDEDataSet* implementiert wird, wir wollen deshalb zwischen diesen beiden Komponenten nicht unterscheiden. *TBDEDataSet* wiederum ist von der in Kapitel 3.1 beschriebenen Komponente *TDataSet* abgeleitet.

Verbindung zur Datenbank

- DatabaseName (Eigenschaft, veröffentlicht)

  ```
  property DatabaseName: string;
  ```

  Mit der Eigenschaft *DatabaseName* wird vorgegeben, wo die Tabelle zu suchen ist. Dabei werden folgende Werte akzeptiert:

  - ein definierter BDE-Alias
  - ein Verzeichnis für Desktop-Datenbankdateien

- ein Dateiname (inklusive Verzeichnispfad) auf einem SQL-Server
- ein anwendungsspezifischer Alias, der über eine Komponente *TDatabase* definiert wurde

- SessionName (Eigenschaft, veröffentlicht)

```
property SessionName: string;
```

Diese Eigenschaft können Sie in der Regel leer lassen, die Datenmenge wird dann mit der Default-Session verbunden. Bei Multithread-Anwendungen muss für jeden zusätzlichen Thread, der eine Datenbankverbindung nutzt, eine weitere *TSession*-Komponente eingefügt werden. Die in diesem Thread verwendeten Datenmengenkomponenten müssen mit dieser *TSession*-Komponente verbunden werden. Dazu muss aus der Drop-Down-Liste der Eigenschaft *SessionName* die entsprechende Session gewählt werden.

- Database, DBSession (Eigenschaft, öffentlich, nur Lesen)

```
property Database: TDatabase;
property DBSession: TSession
```

Mit Hilfe dieser Eigenschaften lässt sich ermitteln, mit welcher *TDatabase*- und welcher *TSession*-Komponente die jeweilige Datenmengenkomponente verbunden ist.

- FetchAll (Methode)

```
procedure FetchAll;
```

Normalerweise würde die Datenmengenkomponente nur diejenigen Daten aus der Datenbank anfordern, die momentan zur Anzeige benötigt werden. Scrollt der Anwender beispielsweise durch eine Datenmenge, dann würden laufend neue Datensätze angefordert, was die Netzwerkauslastung nicht gerade minimiert.

Mit der Methode *FetchAll* kann dafür gesorgt werden, dass alle Datensätze von der Cursor-Position ausgehend bis zum Dateiende von der Datenbank gelesen und lokal zwischengespeichert werden.

- FlushBuffers (Methode)

```
procedure FlushBuffers;
```

Bei Desktop-Datenbanken werden Änderungen beim Aufruf der Methode *Post* nicht auf die Festplatte, sondern nur in den Speicher geschrieben. Möchten Sie sicherstellen, dass die Änderungen auch nach einem Absturz des Rechners nicht verloren sind, müssen Sie jeweils die Methode *FlushBuffers* aufrufen, was beispielsweise in der *AfterPost*-Ereignisbehandlungsroutine erfolgen kann.

FlushBuffers ist auch zwingend erforderlich, wenn eine dBase-Datei von einer anderen Anwendung (und nicht über die BDE) geöffnet werden soll.

Der Aufruf von *FlushBuffers* ist bei SQL-Servern nicht erforderlich, er würde zu einer Fehlermeldung führen.

```
procedure TForm1.Table1AfterPost(DataSet: TDataSet);
begin
  Table1.FlushBuffers;
end;
```

Cached Updates

Normalerweise werden Änderungen in der Datenmenge sofort in die betreffende Datenbank geschrieben. Dieses Verhalten kann dann problematisch sein, wenn der Datenbankbestand auf einem Datenbank-Server gespeichert ist. Hier müsste entweder bei jedem neuen Datensatz eine neue Transaktion gestartet und verwaltet werden, oder der Datenbank-Server wäre mit dem *Versioning* (InterBase) beschäftigt – andere Datenbank-Server haben andere Methoden, um eine konstante Datensicht zu erhalten. Außerdem müsste für jeden Datensatz der ganze Overhead über das Netzwerk geschickt werden. Solange nur wenige Clients vorhanden sind, ist das ja alles noch kein Problem. Was ist jedoch, wenn einige hundert Clients auf den Server zugreifen? Hier kann es helfen, die Änderungen zunächst zwischenzuspeichern und dann mehrere Änderungen gemeinsam zum Server zu schicken.

Nehmen wir beispielsweise die Auftragseingabe eines Versandhauses. Egal, ob die Bestellungen nun telefonisch oder schriftlich eintreffen, würde das System – konservativ programmiert – folgendermaßen arbeiten: Bei jeder Bestellung würde man eine Transaktion starten. Zunächst würde die Kundennummer oder die Adresse erfasst. Hier müsste geklärt werden, ob man diesen Kunden überhaupt noch beliefern möchte. Anschließend werden die bestellten Artikel eingegeben, wobei festgestellt werden muss, ob der Artikel noch lieferbar ist. Ist die Bestellung komplett erfasst, kann die Transaktion beendet werden, in der Versandabteilung werden dann Lieferschein und Rechnung gedruckt.

Grob geschätzt, kann eine Adresse in einer Minute eingegeben werden, Anzahl und Bestellnummer eines Artikels jeweils in zehn Sekunden. Sie können sich sicher vorstellen, wie das Netzwerk belastet ist, wenn hier an vielen Rechnern Bestellungen eingegeben werden.

Mit *CachedUpdates* würde man nach jeder abgeschlossenen Bestellung eine Transaktion starten und die Datensätze auf einmal zum Server schicken. Statt alle zehn Sekunden würde dann beispielsweise nur alle zwei Minuten eine Transaktion gestartet, und diese würde dann auch nicht zwei Minuten, sondern weniger als eine Sekunde dauern.

Tritt bei der Aktualisierung ein Fehler auf, ist beispielsweise ein Artikel nicht mehr lieferbar, dann wird die ganze Bestellung mit ROLLBACK zurückgenommen. Der Sachbearbeiter oder die Sachbearbeiterin können dann die Bestellung entsprechend modifizieren – beispielsweise kann auf die Lieferung des dazugehörenden Netzteiles verzichtet werden, wenn das Gerät nicht mehr lieferbar ist.

In diesem Beispiel fallen die Nachteile von *CachedUpdates* so gut wie überhaupt nicht ins Gewicht: Zum einen erhält man bei Abfragen nicht ganz die aktuelle Sicht der Dinge. Aber ob der Einkauf zwei Minuten früher oder später davon erfährt, dass ein Artikel nachbestellt werden muss, dürfte nun wirklich unerheblich sein. Außerdem erfährt der Anwender nicht sofort, dass eine Datensatzänderung nicht möglich ist, beispielsweise weil der betreffende Artikel ausverkauft ist.

- CachedUpdates (Eigenschaft, veröffentlicht)

  ```
  property CachedUpdates: Boolean default false;
  ```

  Mit der Eigenschaft *CachedUpdates* legt man fest, ob zwischengespeicherte Aktualisierungen verwendet werden sollen oder nicht.

- ApplyUpdates, CommitUpdates (Methoden)

  ```
  procedure ApplyUpdates;
  procedure CommitUpdates;
  ```

  Mit *ApplyUpdates* werden die Änderungen in die Datenbank geschrieben, anschließend kann der Zwischenspeicher mit *CommitUpdates* gelöscht werden.

  ```
  procedure Button1Click(Sender: TObject);
  begin
    Database1.StartTransaction;
    try
      Table1.ApplyUpdates;
      Database1.Commit;
      Table1.CommitUpdates;
    except
      Database1.Rollback;
      ShowMessage('Änderungen konnten nicht geschrieben werden');
    end;
  end;
  ```

  Vor dem Schreiben der Datensätze wird zunächst eine Transaktion gestartet, anschließend eine *try..except..end*-Konstruktion. Nun werden mit *ApplyUpdates* die zwischengespeicherten Änderungen in die Datenbank geschrieben, anschließend wird die Transaktion mit *Commit* abgeschlossen, und die Änderungen werden mit *CommitUpdates* aus dem Zwischenspeicher entfernt.

Tritt beim Schreiben der Änderungen ein Fehler auf, dann wird eine Exception ausgelöst, welche die Ausführungen der folgenden Anweisungen verhindert. Stattdessen wird mit *Rollback* die Transaktion zurückgenommen und eine Fehlermeldung ausgegeben.

- UpdatesPending (Eigenschaft, öffentlich, nur Lesen)

```
property UpdatesPending: Boolean;
```

Die Eigenschaft *UpdatesPending* ist gleich *true*, wenn im Zwischenspeicher Änderungen enthalten sind, die noch nicht ausgeführt wurden.

- UpdateRecordTypes (Eigenschaft, öffentlich)

```
property UpdateRecordTypes:
  set of (rtModified, rtInserted, rtDeleted, rtUnmodified);
```

Um zu steuern, welche Datensätze in der Datenmenge angezeigt werden sollen, wird die Eigenschaft *UpdateRecordTypes* verwendet.

- CancelUpdates, RevertRecord (Methoden)

```
  procedure CancelUpdates;
  procedure RevertRecord;
```

Mit *CancelUpdates* werden alle zwischengespeicherten Änderungen aus dem Speicher gelöscht und die vorherige Sicht der Datenbank wird wiederhergestellt. Mit *RevertRecord* geschieht dies nur mit dem aktuellen Datensatz.

```
procedure Button2Click(Sender: TObject);
begin
  Table1.UpdateRecordTypes := [rtDeleted];
  try
    Table1.First;
    while not Table1.EOF do
    begin
      Table1.RevertRecord;
      Table1.Next;
    end;
  finally
    Table1.UpdateRecordTypes
      := [rtModified, rtInserted, rtUnmodified];
  end;
end;
```

Mit dieser Methode werden alle gelöschten Datensätze wiederhergestellt. Zunächst wird dafür gesorgt, dass nur die gelöschten Datensätze angezeigt werden. Mit der Schleife werden alle diese Datensätze durchlaufen, mit *RevertRecord* wird das Löschen rückgängig gemacht. Abschließend wird die ursprüngliche Sicht der Datenmenge wiederhergestellt – dies sorgt dafür, dass die Datenmenge so aussieht, wie sie nach *ApplyUpdates* (voraussichtlich) in der Datenmenge steht.

- UpdateObject (veröffentlicht)

```
property UpdateObject: TDataSetUpdateObject;
```

Mit Hilfe dieser Eigenschaft kann die Datenmenge mit einer *TUpdateObject*-Komponente verbunden werden. Mit dieser Komponente können beispielsweise SQL-JOINS bearbeitet werden. Mehr dazu in Kapitel 6.4.

- OnUpdateRecord, OnUpdateError (Ereignisse)

```
property OnUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
property OnUpdateError(DataSet: TDataSet; E: EDatabaseError;
    UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
```

Das erstgenannte Ereignis wird bei der Aktualisierung eines Datensatzes ausgelöst, das zweite immer dann, wenn es bei der Aktualisierung zu einem Fehler gekommen ist. Der Parameter *UpdateKind* informiert darüber, ob das Ereignis beim Ändern (*ukModify*), Einfügen (*ukInsert*) oder Löschen (*ukDelete*) aufgetreten ist.

Mit Hilfe des Variablenparameters *UpdateAction* kann bestimmt werden, welche Aktion ausgeführt werden soll – beispielsweise, auf welche Art und Weise der Fehler behandelt werden soll. Der Parameter *UpdateAction* kann folgende Werte annehmen:

- *uaApplied:* Die Änderung wird eingetragen und aus dem Zwischenspeicher gelöscht.
- *uaAbort:* Die Aktualisierung wird ohne Fehlermeldung abgebrochen.
- *uaFail:* Die Aktualisierung wird mit Fehlermeldung abgebrochen.
- *uaRetry:* Es wird erneut versucht, die Änderung einzutragen. Zuvor sollte der Fehler möglichst behoben werden. (Tritt nur bei *OnUpdateError* auf.)
- *uaSkip:* Die Aktualisierung wird übergangen, die entsprechenden Daten bleiben im Zwischenspeicher.

5.6 TTable

Die Komponente *TTable* kapselt den Zugriff auf eine Datenbanktabelle oder (nur bei SQL-Servern) eine VIEW. *TTable* ist von *TDBDataSet* (Kapitel 5.5) abgeleitet.

Verbindung zur Tabelle

- TableName, TableType (Eigenschaften, veröffentlicht)

```
property TableName: TFileName;
property TableType: TTableType
```

  Mit der Eigenschaft *TableName* wird der Name derjenigen Tabelle oder VIEW angegeben, die *TTable* öffnen soll. Für gewöhnlich entscheidet *TTable* anhand der Dateiendung, welcher Datenbanktyp der verwendeten Tabelle zugrunde liegt (*TableType* = *ttDefault*). Sollen davon abweichende Dateiendungen verwendet werden, dann ist mit *TableType* der Datenbanktyp (*ttParadox*, *ttDBase*, *ttASCII*, *ttFoxPro*) anzugeben.

- RenameTable (Methode)

```
procedure RenameTable(const NewTableName: string);
```

  Mit dieser Methode können *Paradox*- und *dBase*-Tabellen umbenannt werden.

- CreateTable, DeleteTable (Methoden)

```
procedure CreateTable;
procedure DeleteTable;
```

  Mit diesen Methoden können zur Laufzeit Tabellen erstellt und gelöscht werden. Ein Beispiel zum Erstellen von Tabellen finden Sie in Kapitel 3.1 bei der Beschreibung von *TFieldDefs*.

- ReadOnly (Eigenschaft, veröffentlicht)

```
property ReadOnly: Boolean default false;
```

  Wird *ReadOnly* auf *true* gesetzt, dann kann die *TTable*-Instanz keine Änderungen am Datenbestand vornehmen.

- Exclusive (Eigenschaft, veröffentlicht)

```
property Exclusive: Boolean default false;
```

  Wird eine Tabelle *Exclusive* geöffnet, dann kann sie von keiner weiteren Tabelle geöffnet werden. Ist die Tabelle bereits von einer anderen Tabelle geöffnet, dann wird eine Exception ausgelöst. Diese Funktion wird nur von *Paradox*- und *dBase*-Tabellen unterstützt. Soll diese Eigenschaft auf *true* gesetzt werden, dann muss währenddessen die Datenmenge geschlossen sein.

Vorsicht: Wenn Sie *Exclusive* und *Active* einer Tabelle schon zur Entwurfszeit auf *true* setzen, können Sie die Anwendung nicht mehr starten, weil sich dann die betreffende Tabelle nicht noch einmal öffnen lässt.

- Lock, Unlock (Methoden)

```
procedure LockTable(LockType: TLockType);
procedure UnlockTable(LockType: TLockType);
```

Mit der Prozedur *LockTable* kann eine Lesesperre (*ltReadLock*) oder eine Schreibsperre (*ltWriteLock*) für *dBase*- und *Paradox*-Tabellen erstellt werden. Soll sowohl der Lese- als auch der Schreibzugriff durch andere Tabellen verhindert werden, dann ist *LockTable* zweimal aufzurufen. Mit *UnlockTable* kann eine Lese- oder Schreibsperre aufgehoben werden.

Lese- und Schreibsperren sind besonders dann hilfreich, wenn beim Einsatz von Desktop-Datenbanksystemen im Netzwerk verhindert werden muss, dass durch den gleichzeitigen Zugriff mehrerer Benutzer der Datenbestand inkonsistent wird.

Ändern der Daten

- EmptyTable (Methode)

```
procedure EmptyTable;
```

Mit dieser Methode werden alle Datensätze aus der Tabelle gelöscht, die Tabellenstruktur wird jedoch beibehalten.

- BatchMove (Methode)

```
function BatchMove(ASource: TBDEDataSet;
  AMode: TBatchMode): Longint;
```

Mit der Methode *BatchMove* können Stapeloperationen durchgeführt werden. Über den Parameter *ASource* muss die Datenmenge angegeben werden, aus der die zu bearbeitenden Datensätze stammen, *AMode* gibt die Art der Stapeloperation an:

- *batCopy* legt eine exakte Kopie der im Parameter angegebenen Tabelle an.
- *batDelete* löscht alle Datensätze, die 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, die (über den Index gesucht) schon in der Zieltabelle vorhanden sind, werden geändert, alle anderen angehängt.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Table2: TTable;
begin
  Screen.Cursor := crHourGlass;
  Table2 := TTable.Create(Form1);
  with Table2 do
  begin
    DatabaseName := 'Test';
    TableName := 'testadr3.txt';
    TableType := ttAscii;
    BatchMove(Table1, batCopy);
  end;
  Table2.Free;
  Screen.Cursor := crDefault
end;
```

Dieses Beispiel erzeugt zur Laufzeit eine *TTable*-Komponente und legt in ihr eine exakte Kopie der *Table1*-Daten im ASCII-Format an.

Stapeloperationen können Sie auch mit der Komponente *TBatchMove* durchführen, die in Kapitel 5.7 beschrieben ist.

Bereiche

- SetRangeStart, SetRangeEnd (Methoden)

```
procedure SetRangeStart;
procedure SetRangeEnd;
```

Mit den Methoden *SetRangeStart* und *SetRangeEnd* werden die oberen und unteren Grenzen eines Bereichs gesetzt. Die Werte für die Grenzen werden über die Eigenschaft *Fields* oder die Methode *FieldByName* gesetzt.

```
with Table1 do
begin
  SetRangeStart;
  FieldByName('nummer').AsInteger := 100;
  SetRangeEnd;
  FieldByName('nummer').AsInteger := 200;
  ApplyRange;
end;
```

- ApplyRange, CancelRange (Methoden)

```
procedure ApplyRange;
procedure CancelRange;
```

  Diese Prozeduren wenden einen gesetzten Bereich an (*ApplyRange*) oder heben ihn auf (*CancelRange*).

- KeyExclusive (öffentlich, TTable)

```
property KeyExclusive: Boolean;
```

  Mit Hilfe dieser Eigenschaft wird festgelegt, ob die Bereichsgrenzen zum Bereich gehören (*false*) oder nicht (*true*).

- SetRange (Methode)

```
procedure SetRange(const StartValues, EndValues: array of const);
```

  Werden Bereichsgrenzen mit *SetRangeStart* und *SetRangeEnd* gesetzt, dann ist dies ein wenig aufwändig. Mit Hilfe von *SetRange* kann der Bereich durch eine einzige Anweisung gesetzt werden.

```
Table1.SetRange([200], [300]);
```

- EditRangeStart, EditRangeEnd (Methoden)

```
procedure EditRangeStart;
procedure EditRangeEnd;
```

  Mit diesen Methoden können die Grenzen bestehender Bereiche verändert werden.

Schlüssel

Mit den folgenden Anweisungen kann man über die Schlüsselfelder einer Tabelle einen Datensatz finden.

- SetKey, EditKey (Methoden)

```
procedure SetKey;
procedure EditKey;
```

  Mit *SetKey* wird das Setzen, mit *EditKey* das Ändern des Suchschlüssels eingeleitet.

- GotoKey, GotoNearest (Methoden)

```
function GotoKey: Boolean;
procedure GotoNearest;
```

Die Funktion *GotoKey* versucht, einen Datensatz anhand des gesetzten Schlüssels zu finden. Gelingt ihr das, wird der Datenzeiger auf diesen Datensatz gesetzt und als Funktionsergebnis *true* zurückgegeben.

Die Prozedur *GotoNearest* positioniert den Datenzeiger auf den Datensatz, dessen Feldwerte dem Suchschlüssel am meisten gleichen.

```
procedure Button1Click(Sender: TObject);
begin
  with Table 1 do
  begin
    SetKey;
    FieldByName('Nummer').AsInteger := 2345;
    if GotoKey = false
      then ShowMessage('Datensatz nicht gefunden');
  end;
end;
```

- FindKey, FindNearest (Methoden)

```
function FindKey(const KeyValues: array of const): Boolean;
procedure FindNearest(const KeyValues: array of const);
```

Diese Methoden fassen das Suchen nach einem Datensatz zu einer einzigen Anweisung zusammen, die Schlüsselwerte werden als Parameter übergeben. Das gerade gegebene Beispiel würde man mit *FindKey* folgendermaßen formulieren:

```
procedure Button1Click(Sender: TObject);
begin
  if Table1.FindKey([2345]) = false
    then ShowMessage('Datensatz nicht gefunden');
end;
```

- GotoCurrent (Methode)

```
procedure GotoCurrent(Table: TTable);
```

Die Methode *GotoCurrent* setzt den Datenzeiger auf denselben Datensatz wie den Datenzeiger der als Parameter übergebenen Tabelle. Auf diese Weise lassen sich sehr einfach zwei *TTable*-Instanzen synchronisieren.

Indizes

Viele Operationen bei *TTable* (Bereiche, Suche nach Schlüsseln) sind nur dann möglich, wenn die verwendeten Felder indiziert sind.

- IndexName (Eigenschaft, veröffentlicht)

  ```
  property IndexName: string;
  ```

  Die Eigenschaft *IndexName* gibt den Namen des Sekundärindexes an, nach dem die Datenmenge sortiert wird. Bleibt die Eigenschaft leer, dann wird nach dem Primärindex sortiert.

- IndexFieldNames (Eigenschaft, veröffentlicht),
  IndexFields (Eigenschaft, öffentlich)

  ```
  property IndexFieldNames: string;
  property IndexFields: [Index: Integer]: TField;
  ```

  Alternativ zu *IndexName* können mit *IndexFieldNames* die einzelnen Spalten angegeben werden, aus denen ein Index gebildet werden soll. Bei Desktop-Datenbanken ist es dabei erforderlich, dass die einzelnen Spalten bereits selbst indiziert sind.

  Die an einem Index beteiligten Spalten können mit *IndexFields* ermittelt werden.

- AddIndex, DeleteIndex (Methoden)

  ```
  procedure AddIndex(const Name, Fields: string;
    Options: TIndexOptions);
  procedure DeleteIndex(const Name: string);
  ```

  Um zur Laufzeit neue Indizes zu erstellen, wird die Methode *AddIndex* verwendet. Näheres dazu in der Online-Hilfe. Mit *DeleteIndex* können Sie Indizes löschen.

- GetIndexNames (Methoden)

  ```
  procedure GetIndexNames(List: TStrings);
  ```

  Mit *GetIndexNames* können die Namen aller vorhandenen Indizes abgefragt und in eine Liste geschrieben werden.

- IndexDefs (Eigenschaft, veröffentlicht)

  ```
  property IndexDefs: TIndexDefs;
  ```

  Um Informationen über die Indizes zu erhalten, wird die Eigenschaft *IndexDefs* verwendet. Im folgenden Beispiel wird die Eigenschaft *IndexDefs* beim Erstellen einer Tabelle zur Laufzeit verwendet.

```
procedure TForm1.N1Tabelleerstellen1Click(Sender: TObject);
begin
  with Table1 do
  begin
    Active := false;
    DataBaseName := 'test';
    TableName := 'test_1.db';
    TableType := ttParadox;
    with FieldDefs do
    begin
      Clear;
      Add('nummer', ftInteger, 0, false);
      Add('name', ftString, 20, false);
    end;
    with IndexDefs do
    begin
      Clear;
      Add('Primary', 'nummer', [ixPrimary, ixUnique]);
    end;
    CreateTable;
    Open;
  end; {with table1 do}
end; {procedure TForm1.N1Tabelleerstellen1Click}
```

Master-Detail-Verknüpfungen

Master-Detail-Verknüpfungen wurden ausführlich in Kapitel 5.2 besprochen.

- MasterSource (Eigenschaft, veröffentlicht)

  ```
  property MasterSource: TDataSource;
  ```

  Mit der Eigenschaft *MasterSource* wird angegeben, welche Datenquelle diese Datenmenge als Master-Tabelle steuert.

- MasterFields (Eigenschaft, veröffentlicht)

  ```
  property MasterFields: string;
  ```

  Der Eigenschaft *MasterFields* werden diejenigen Feldnamen zugewiesen, über die bei der Master-Tabelle die Verknüpfung gesteuert wird. Mehrere Feldnamen werden dabei mit Semikola getrennt. In Anzahl und Feldart müssen diese Felder dem momentan verwendeten Schlüssel der Detailtabelle entsprechen.

5.7 Die Komponente TBatchMove

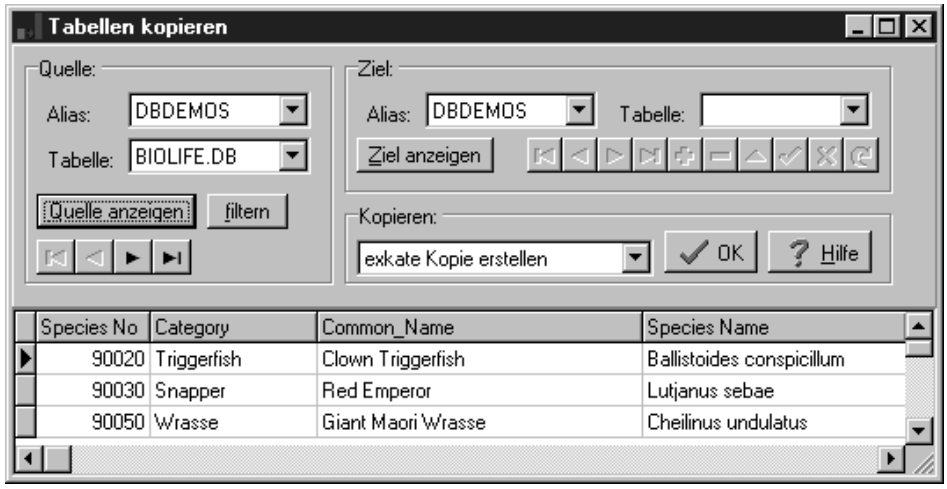
Es gibt Situationen, in denen große Datenmengen von einer Tabelle in eine andere kopiert werden müssen.

- 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 auf das Notebook eines Außendienstmitarbeiters.
- 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.

5.7.1 Ein Datentransferprogramm

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 welche Daten befinden, kann mit *DBGrid1* wahlweise die Quell- oder die Zieltabelle angezeigt werden. Außerdem besteht die Möglichkeit, die Quelldatenbank mit Hilfe einer SQL-Anweisung zu filtern.

Auswählen der Datenbanken und der Tabellen

Mit Hilfe von vier *TComboBox*-Komponenten soll 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.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.GetAliasNames(ComboBox1.Items);
  Session.GetAliasNames(ComboBox3.Items);
end;
```

Beim Starten des Programms werden alle vorhandenen Datenbank-Aliase den entsprechenden *ComboBoxen* als Einträge zugewiesen.

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  Query1.Active := false;
  Query1.DatabaseName := ComboBox1.Text;
  Session.GetTablenames(ComboBox1.Text, '', True, False,
    ComboBox2.Items);
  ComboBox2.Text := '';
end;
```

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 abgeändert werden.

```
procedure TForm1.ComboBox2Change(Sender: TObject);
begin
  with Query1 do
  begin
    Active := false;
    SQL.Clear;
    SQL.Add('SELECT * FROM "' + LowerCase(ComboBox2.Text) + '"');
    Active := true;
  end; {with Query1 do}
end; {procedure TForm1.ComboBox2Change}
```

Wird eine Quelltabelle ausgewählt, dann wird eine entsprechende SQL-Anweisung generiert und die Abfrage geöffnet. Gemäß den SQL-Konventionen werden alle Schlüsselwörter in Großbuchstaben und alle Bezeichner in Kleinbuchstaben geschrieben – entsprechend wird der Tabellenname mit der Funktion *LowerCase* in Kleinbuchstaben umgewandelt. Da der ausgewählte Tabellenname meist die Endung *.DB* oder *.DBF* aufweist, muss dieser Bezeichner in Anführungszeichen gesetzt werden.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if form5.ShowModal = mrOk
    then Form5.Memo1.Lines := Query1.SQL;
end;
```

Wird der Button *Filtern* betätigt, dann wird ein Eingabe-Dialog für eine SQL-Anweisung aufgerufen.

Bei der Prozedur *ComboBox4Change* muss der ausgewählte Tabellenname der Eigenschaft *TableName* von *Table1* zugewiesen werden.

```
procedure TForm1.ComboBox5Change(Sender: TObject);
begin
  with BatchMove1 do
  begin
    case ComboBox5.ItemIndex of
      0: Mode := batAppend;
      1: Mode := batUpdate;
      2: Mode := batAppendUpdate;
      3: Mode := batDelete;
      4: Mode := batCopy;
    end;
  end; {with BatchMove1 do}
end; {procedure TForm1.ComboBox5Change}
```

Mit *ComboBox5* kann man auswählen, welche Operation die Komponente *BatchMove1* durchführen soll, entsprechend wird die Eigenschaft *Mode* gesetzt.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  if ComboBox5.ItemIndex = 4 then
  begin
    Table1.Close;
    Table1.TableName := InputBox('Tabellennamen', 'Geben Sie den' +
      Tabellennamen ein, den die neue Tabelle haben soll',
      'Test.DB');
  end;
  BatchMove1.Execute;
end;
```

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. Allerdings kann man in *ComboBox4* keine nicht vorhandenen Tabellennamen eingeben – und wenn man es könnte, dann würde der Versuch, diese Tabelle zu öffnen, zu einer Exception führen.

Deshalb wird bei der Operation *batCopy* vom Anwender ein Tabellenname abgefragt – dies kann ein bereits vorhandener oder ein neuer Dateiname sein.

5.7.2 Referenz TBatchMove

Im Folgenden soll eine kurze Übersicht über die wesentlichen Eigenschaften und Methoden von *TBatchMove* gegeben werden:

- Source (veröffentlicht, TBatchMove)

  ```
  property Source: TBDEDataSet;
  ```

  Die Eigenschaft *Source* spezifiziert die *DataSet*-Komponente (*TTable* oder *TQuery*), von welcher aus die Daten übernommen werden.

- Destination (veröffentlicht, TBatchMove)

  ```
  property Destination: TTable;
  ```

  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.

- Mode (veröffentlicht, TBatchMove)

  ```
  property Mode: TBatchMode;
  ```

  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.

- Execute (Methode, TBatchMove)

  ```
  procedure Execute;
  ```

  Mit *Execute* wird die Stapeloperation gestartet.

- Fehlerbehandlung (veröffentlicht, TBatchMove)

  ```
  property AbortOnKeyViol: Boolean;
  property AbortOnProblem: Boolean;
  property KeyViolTableName: TTable;
  property ProblemTableName: TTable;
  ```

  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 nachbearbeitet werden können.

- Mappings (veröffentlicht, TBatchMove)

```
property Mappings: TStrings;
```

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.

```
procedure Zuordnungsliste_erstellen;
var
  Maps: TStringList;
begin
  Maps := TStringList.Create;
  with Maps do
  begin
    Clear;
    Add('Nummer=Ku_Nr');
    Add('Namen=Namen_1');
    ...
  end;
  BatchMove1.Mappings := Maps;
end;
```

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.

6 TQuery

Die Komponente *TQuery* arbeitet mit SQL-Anweisungen:

- Wird ein SELECT-Statement verwendet, dann wird die Ergebnismenge als Datenmenge zur Verfügung gestellt. Ein SELECT-Statement kann eine einzelne Tabelle umfassen, es lassen sich damit aber auch Tabellen zusammenfügen (JOIN).
- Mit INSERT-, UPDATE- und DELETE-Anweisungen lassen sich Änderungen am Datenbestand vornehmen.
- Verwendet man die Befehle der DDL (*database definition language*), dann kann man Tabellen erstellen, ändern und löschen, Indizes erstellen und vieles andere mehr.
- Über so genannte *heterogene JOINS* können Daten aus verschiedenen Datenmengen zu einer einzelnen Ergebnismenge zusammengefügt werden.

TQuery ist die geeignete Komponente zum Zugriff auf Client-Server-Systeme. Hier wird die SQL-Anweisung direkt zum Server durchgeleitet und nicht von der BDE bearbeitet. Somit steht auch der Befehlsumfang des Servers vollständig zur Verfügung.

Mit *TQuery* kann man aber auch auf lokale Datenbanken zugreifen. Hier wird die Abfrage dann von der BDE bearbeitet. Es ist jedoch damit zu rechnen, dass die Ausführungsgeschwindigkeit solcher Abfragen nicht das Niveau von *TTable* erreicht.

6.1 LocalSQL

Um mit *TQuery* arbeiten zu können, ist die Beherrschung der Datenbanksprache SQL nötig. SQL (*structured query language*) ist zwar von der amerikanischen Standardisierungsbehörde ASCII genormt, doch implementieren Client-Server-Systeme für gewöhnlich verschiedene »Dialekte« dieser Sprache.

Wir wollen uns daher bei unserer Einführung auf *LocalSQL* beschränken. *LocalSQL* ist die Menge von SQL-Anweisungen, welche von der BDE implementiert werden und die somit bei allen Datenbankanwendungen (welche die BDE verwenden) benutzt werden können. Beim Zugriff auf SQL-Server kann deren SQL-Sprachschatz verwendet werden, welcher meist deutlich über den von LocalSQL hinausgeht.

6.1.1 SELECT

Das SELECT-Statement wird verwendet, um Daten aus der Datenbank abzufragen.

Das folgende Beispiel ermittelt die Spalten *vorname* und *nachname* der Tabelle *kunden*.

```
SELECT vorname, nachname
  FROM kunden
```

Bei SQL ist die Groß- und Kleinschreibung belanglos, gemäß den Konventionen werden jedoch alle SQL-Schlüsselwörter durchgehend mit Großbuchstaben und alle Bezeichner durchgehend mit Kleinbuchstaben geschrieben.

Jokerzeichen

Um alle Spalten einer Tabelle abzufragen, kann das Jokerzeichen * verwendet werden.

```
SELECT *
  FROM kunden
```

Konstanten und zusammengesetzte Strings

Mittels Konstanten können Sie jedem Datensatz ein zusätzliches Feld mit einem konstanten Wert hinzufügen.

```
SELECT "Kunde", vorname, nachname
  FROM kunden
```

Konstanten machen als eigene Spalte meist nicht viel Sinn, sind jedoch wichtig, wenn zwei Spaltenwerte zusammengefügt werden, um dann ein Leerzeichen einzufügen.

```
SELECT vorname || " " || nachname
  FROM kunden
```

Um einen senkrechten Strich (»Pipe«) einzufügen, verwenden Sie Alt Gr + <. Zum Zusammenfügen von Strings benötigen Sie zwei Pipes.

DISTINCT

Mit der folgenden Anweisung würde man alle Nachnamen der Tabelle *kunde* ermitteln:

```
SELECT nachname
  FROM kunde
```

Bei umfangreicheren Tabellen ist es sehr wahrscheinlich, dass manche Nachnamen mehrmals vorkommen. Man hätte dann beispielsweise drei Mal *Maier* und fünf Mal *Schmidt*. Möchte man doppelte Datensätze vermeiden, dann verwendet man das Schlüsselwort DISTINCT.

```
SELECT DISTINCT nachname
  FROM kunde
```

6.1.2 JOINS

Bei normalisierten Datenbanken sind die Daten meistens auf sehr viele Tabellen verteilt. Um vernünftige Ergebnisse zu erhalten, müssen dann die Daten mehrerer Tabellen zusammengefügt werden. Dies geschieht mit einem JOIN.

INNER JOIN mit WHERE-Klausel

Die folgende Anweisung verbindet die Tabelle *auftrag* mit der Tabelle *kunde*.

```
SELECT
     auftrag.nummer,
     auftrag.datum,
     kunde.vorname || " " || kunde.nachname
  FROM auftrag, kunde
  WHERE auftrag.kunde = kunde.nummer
```

Mittels einer WHERE-Klausel wird hier spezifiziert, zu welchem Auftrag welcher Kunde gehört.

Tabellen-Aliase

Sobald sich Spaltennamen in mehreren beteiligten Tabellen gleichen, kann SQL nicht mehr erkennen, aus welcher Tabelle die Spalte nun bezogen werden muss. Deshalb muss jeder Spalte der Tabellenbezeichner vorangestellt werden, getrennt durch einen Punkt.

Allerdings werden SQL-Anweisungen dadurch nicht gerade übersichtlicher. Deshalb besteht die Möglichkeit, Tabellen-Aliase zu verwenden.

```
SELECT
     a.nummer,
     a.datum,
     k.vorname || " " || k.nachname
  FROM auftrag a, kunde k
  WHERE a.kunde = k.nummer
```

Tabellen-Aliase werden in der FROM-Klausel definiert und können in der übrigen SQL-Anweisung verwendet werden. Tabellen-Aliase können auch mehrere Zeichen umfassen, meist wünscht man sich jedoch möglichst kurze Bezeichner.

INNER JOIN mit der ON-Klausel

Ein INNER JOIN kann auch mit der ON-Klausel erstellt werden und ist dann der Syntax eines OUTER JOIN angelehnt.

```
SELECT
    a.nummer,
    a.datum,
    k.vorname || " " || k.nachname
  FROM auftrag a
    INNER JOIN kunde k
      ON a.kunde = k.nummer
```

Es ist eine Geschmacksfrage, aber ich bevorzuge die Schreibweise mit der ON-Klausel, weil über die WHERE-Klausel die Datenmenge auch gefiltert wird.

OUTER JOIN

Ein INNER JOIN ist eine »strenge Verknüpfung«. Ein Datensatz wird nur dann in die Ergebnismenge aufgenommen, wenn von allen beteiligten Tabellen ein Datensatz zur Verfügung gestellt wird. Im folgenden Beispiel müsste ein Mitarbeiter über mindestens einen Telefonanschluss verfügen, damit er in die Ergebnismenge aufgenommen wird.

```
SELECT
    m.vorname || " " || m.nachname,
    t.telefon
  FROM mitarbeiter m
    INNER JOIN telefon t
      ON t.mitarbeiter = m.nummer
```

In manchen Fällen ist ein solches Verhalten erwünscht. Es kann aber auch sein, dass man eine Mitarbeiterliste mit allen Mitarbeitern erstellen möchte und dass bei denjenigen Mitarbeitern, zu denen Telefonnummern gespeichert sind, diese auch angezeigt werden. Hier müsste man nun einen OUTER JOIN erstellen:

```
SELECT
    m.vorname || " " || m.nachname,
    t.telefon
  FROM mitarbeiter m
    LEFT OUTER JOIN telefon t
      ON t.mitarbeiter = m.nummer
```

Mit dem Schlüsselwort LEFT OUTER JOIN wird ein Join erstellt, der alle Datensätze der linken Seite – im Beispiel der Tabelle *mitarbeiter* – umfasst. Dort, wo es auf der rechten Seite passende Datensätze gibt, werden diese dazugefügt.

Nun könnten in der Tabelle *telefon* auch Datensätze sein, die keinem Mitarbeiter zugeordnet sind. Möchte man nun eine Liste aller Telefonnummern erstellen, welche gegebenenfalls die Angabe des Mitarbeiters enthält, müsste man entweder die beiden Tabellenbezeichner vertauschen oder einen RIGHT OUTER JOIN erstellen:

```
SELECT
    m.vorname || " " || m.nachname,
    t.telefon
  FROM mitarbeiter m
    RIGHT OUTER JOIN telefon t
      ON t.mitarbeiter = m.nummer
```

Nun gäbe es auch noch die Möglichkeit, dass man eine Liste mit allen Mitarbeitern und allen Telefonnummern möchte, bei der, wo vorhanden, eine entsprechende Zuordnung besteht. Dies wäre die Aufgabe eines FULL OUTER JOIN.

```
SELECT
    m.vorname || " " || m.nachname,
    t.telefon
  FROM mitarbeiter m
    FULL OUTER JOIN telefon t
      ON t.mitarbeiter = m.nummer
```

Sie können mit einem JOIN mehr als zwei Tabellen verbinden. Setzen Sie gegebenenfalls Klammern, um verschiedene JOIN-Arten korrekt zu mischen.

6.1.3 Filtern mit WHERE

Um eine Datenmenge zu filtern, verwendet man die WHERE-Klausel. Mit der folgenden Anweisung würde man alle Kunden aus Berlin ermitteln:

```
SELECT *
  FROM kunde
  WHERE ort = "Berlin"
```

Beachten Sie, dass Sie Strings in Anführungszeichen setzen müssen, nicht jedoch Zahlen.

```
SELECT *
  FROM kunde
  WHERE nummer = 100
```

Logische Verknüpfungen

Sie können mehrere Filterbedingungen logisch verknüpfen. Die folgende Anweisung sucht nach allen Kunden namens *Peter Maier*.

```
SELECT *
  FROM kunde
  WHERE (vorname = "Peter")
    AND (nachname = "Maier")
```

Sie können auch eine Oder-Verknüpfung erstellen, beispielsweise, um die Kunden aus Berlin und Hamburg zu ermitteln:

```
SELECT *
  FROM kunde
  WHERE (ort = "Berlin")
    OR (ort = "Hamburg")
```

Oder Sie suchen alle Kunden außer denen, die aus München kommen:

```
SELECT *
  FROM kunde
  WHERE NOT (ort = "München")
```

In vielen Fällen sind die Klammern überflüssig. Wenn Sie sich jedoch angewöhnen, stets Klammern zu setzen, dann werden sie auch dann vorhanden sein, wenn sie gebraucht werden.

```
SELECT *
  FROM kunde
  WHERE ((vorname = "Peter")
       OR (vorname = "Petra"))
    AND ((nachname = "Maier")
      OR (nachname = "Meier")
      OR (nachname = "Meyer")
      OR (nachname = "Meyr"))
```

Vergleichs-Operatoren

Sie können nicht nur auf Gleichheit, sondern auch auf Ungleichheit prüfen.

```
SELECT *
  FROM kunde
  WHERE nummer < 200
```

Das funktioniert natürlich auch bei Strings, beispielsweise, um nach einem bestimmten Anfangsbuchstaben zu suchen. Man könnte das auch mit dem LIKE-Operator tun, in der Regel sind Vergleiche bezüglich größer und kleiner jedoch schneller.

```
SELECT *
  FROM kunde
  WHERE (nachname >= "A")
    AND (nachname < "B")
```

Der LIKE-Operator

Mit dem LIKE-Operator können Jokerzeichen verwendet werden. Das folgende Beispiel ermittelt alle Kunden, deren Nachname mit *Ma* beginnt:

```
SELECT *
  FROM kunde
  WHERE nachname LIKE "Ma%"
```

Das Joker-Zeichen % steht für beliebig viele (null, eins, mehrere) Zeichen. Daneben gibt es auch den Unterstrich, der exakt ein Zeichen ersetzt, um hier im Beispiel alle Namen zu suchen, deren zweiter Buchstabe ein *e* ist.

```
SELECT *
  FROM kunde
  WHERE nachname LIKE "_e%"
```

Mit der Funktion UPPER können alle Buchstaben in Majuskeln gewandelt werden, mit LOWER in Minuskeln. Das kann dazu verwendet werden, einen Datensatz auch dann zu finden, wenn die Groß- und Kleinschreibung nicht beachtet wurde:

```
SELECT *
  FROM kunde
  WHERE UPPER(nachname) LIKE ("MA%")
```

Der IN-Operator

Mit dem IN-Operator kann ermittelt werden, ob ein Wert sich in einer bestimmten Menge befindet.

```
SELECT *
  FROM kunde
  WHERE ort IN ("Berlin", "Hamburg", "München")
```

IS NULL

Der Wert NULL steht bei relationalen Datenbanken für einen nicht vorhandenen Wert. Die folgende Anweisung sucht alle Kunden ohne Telefon:

```
SELECT *
  FROM kunde
  WHERE tel IS NULL
```

Für die Kunden mit Telefon sind die folgenden beiden Schreibweisen zulässig:

```
SELECT *
  FROM kunde
  WHERE NOT (tel IS NULL)
```

```
SELECT *
  FROM kunde
  WHERE tel IS NOT NULL
```

6.1.4 Sortieren mit ORDER BY

Solange nicht explizit eine Sortierung verlangt wird, können die Daten in beliebiger Reihenfolge zurückgegeben werden. Ist ein Primärschlüssel vorhanden, so wird in der Regel nach diesem sortiert.

Um eine Datenmenge zu sortieren, verwenden Sie die ORDER BY-Klausel.

```
SELECT *
  FROM kunde
  ORDER BY nachname
```

Es ist auch möglich, nach mehreren Spalten zu sortieren, im folgenden Beispiel zuerst nach dem Nachnamen und dann nach dem Vornamen:

```
SELECT *
  FROM kunde
  ORDER BY nachname, vorname
```

Solange nichts anderes verlangt wird, wird bei Verwendung der ORDER BY-Klausel aufsteigend sortiert (zuerst *A*, zuletzt *Z*). Soll absteigend sortiert werden, ist dem jeweiligen Spaltenbezeichner das Schlüsselwort DESC anzuhängen.

```
SELECT *
  FROM kunde
  ORDER BY nachname DESC, vorname
```

6.1.5 Aggregatfunktionen

SQL implementiert 5 Aggregatfunktionen zur Berechnung statistischer Werte:

- COUNT ermittelt die Anzahl der Datensätze,
- SUM die Summe,
- AVG den Durchschnitt,
- MIN das Minimum und
- MAX das Maximum.

Die folgende Anweisung ermittelt die Zahl aller Datensätze in der Tabelle *kunde*:

```
SELECT COUNT(*)
  FROM kunde
```

Mit der folgenden Anweisung wird die Zahl aller Berliner Kunden mit Telefon ermittelt:

```
SELECT COUNT(tel)
  FROM kunde
  WHERE ort = "Berlin"
```

Und auf diese Weise erfährt man das Datum der ersten und der letzten Bestellung:

```
SELECT MIN(datum), MAX(datum)
  FROM bestellung
```

Wenn Sie Aggregat-Funktionen in einer SELECT-Anweisung verwenden, dürfen Sie nur dann auch noch andere Spalten in die Ergebnismenge aufnehmen, wenn Sie nach diesen Spalten gruppieren. Dazu verwendet man die GROUP BY-Klausel.

6.1.6 Gruppieren mit GROUP BY und HAVING

Mit der GROUP BY-Klausel können Sie detaillierte Statistiken erstellen. Mit der folgenden Anweisung ermitteln Sie beispielsweise die Orte, in denen die Kunden wohnen, und dazu die Zahl der Kunden des jeweiligen Ortes.

```
SELECT COUNT(*), ort
  FROM kunde
  GROUP BY ort
```

Nun sind das in der Regel sehr viele Orte. Um hier etwas mehr Klarheit zu schaffen, sollen nun nur noch diejenigen Orte in die Datenmenge aufgenommen werden, in denen mehr als 50 Kunden wohnen.

```
SELECT COUNT(*), ort
  FROM kunde
  GROUP BY ort
  HAVING COUNT(*) > 50
```

Die HAVING-Klausel ähnelt der WHERE-Klausel, lässt sich aber auch auf Aggregatfunktion-Spalten anwenden.

6.1.7 Operatoren und Funktionen

Local SQL stellt einige Operatoren und Funktionen zum Bearbeiten von Werten zur Verfügung. Einige dieser Elemente wurden bereits besprochen.

Die mathematischen Operatoren sowie die String-Funktionen können Sie sowohl in der Spaltenliste als auch in der WHERE- beziehungsweise HAVING-Klausel verwenden.

Mathematische Operatoren

SQL beherrscht die vier Grundrechenarten und verwendet dafür die Zeichen +, -, * und /. Im Folgenden wird der Netto- und der Brutto-Preis der Artikel bestimmt.

```
SELECT preis,
    preis * 1.16
  FROM artikel
```

String-Funktionen

Mit zwei Pipes können Strings zusammengefügt werden.

```
SELECT vorname || " " || nachname,
    strasse,
    plz || " " || ort
  FROM kunde
```

Um Spaltenwerte in Groß- oder in Kleinbuchstaben umzuwandeln, verwenden Sie die Funktionen UPPER beziehungsweise LOWER.

```
SELECT UPPER(nachname)
  FROM kunde
```

Gebräuchlicher sind diese Funktionen in der WHERE-Klausel, um Datensätze trotz inkonsistenter Groß- und Kleinschreibung zu finden.

```
SELECT *
  FROM kunde
  WHERE LOWER(nachname) LIKE ("b%")
```

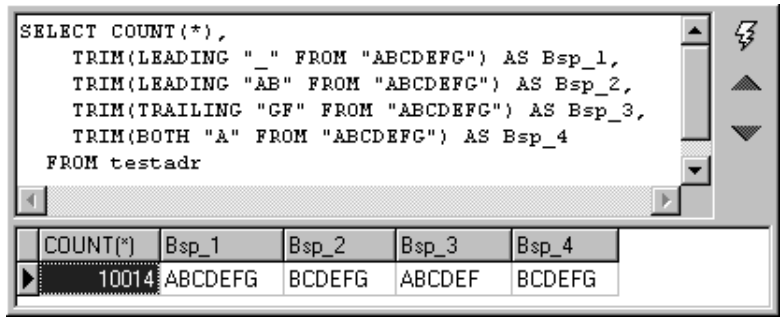
Mit SUBSTRING kann man Teile aus einem String extrahieren. Die folgende Anweisung ermittelt aus der Spalte *ident* sieben Zeichen, beginnend vom zweiten Zeichen.

```
SELECT SUBSTRING(ident FROM 2 FOR 7)
  FROM kunde
```

Bei der Funktion SUBSTRING ist die FOR-Klausel optional. Entfällt sie, wird das Ergebnis von bezeichneter Position bis zum Ende des Strings gebildet. Das folgende Beispiel würde also in der Ergebnismenge das erste Zeichen der Spalte *ident* entfernen.

```
SELECT SUBSTRING(ident FROM 2)
  FROM kunde
```

Mit der Funktion TRIM entfernt man Zeichen am Anfang, am Ende oder an beiden Seiten eines Strings. Als Beispiel siehe die folgende Abbildung.



Typenumwandlung

Mit der Funktion CAST können Datentypen umgewandelt werden. Im folgenden Beispiel wird der Betrag auf zwei Stellen gerundet, indem er zunächst mit 100 multipliziert wird. Durch eine Umwandlung in den Typen INTEGER verliert er alle Nachkommastellen, anschließend wird er mit 0.01 multipliziert.

```
SELECT nummer,
    datum,
    CAST((betrag * 100 + 0.49) AS INTEGER) * 0.01
  FROM bestellung
```

Mit der Funktion EXTRACT können Datumsbestandteile aus einem Datum extrahiert werden.

```
SELECT nummer,
     EXTRACT(YEAR FROM datum)
  FROM auftrag
```

Dabei können YEAR, MONTH, DAY, HOUR, MINUTE und SECOND ermittelt werden.

6.1.8 Ergebnismengen verbinden mit UNION

Um die Ergebnismengen von zwei SELECT-Abfragen zu verbinden, wird die UNION-Klausel verwendet.

```
SELECT vorname, nachname
  FROM kunden
UNION
SELECT vorname, nachname
  FROM mitarbeiter
```

Beachten Sie dabei, dass die Spaltentypen und Spaltengrößen übereinstimmen müssen.

6.1.9 Unterabfragen

Werte für die Spaltenliste oder für die WHERE-Klausel lassen sich auch aus einer anderen Abfrage, nämlich einer Unterabfrage, ermitteln.

Im folgenden Beispiel sollen alle Mitarbeiter ermittelt werden, welche dem Chef direkt unterstellt sind, deren Spalte *vorgesetzter* somit den Wert der Spalte *nummer* des Chefs hat. Um nun zu ermitteln, welche Nummer der Chef hat, wird eine Unterabfrage gestartet, welche den Datensatz sucht, bei dem *vorgesetzter* und *nummer* den gleichen Wert haben.

```
SELECT vorname, nachname
  FROM mitarbeiter
  WHERE (vorgesetzter <> nummer)
    AND (vorgesetzter = (SELECT nummer
      FROM mitarbeiter
      WHERE vorgesetzter = nummer))
```

Wenn Sie Unterabfragen einsetzen, um Werte für die Spaltenliste zu ermitteln, dann achten Sie darauf, dass solche Unterabfragen nicht mehr als einen Wert ermitteln dürfen. In der Regel lässt sich das gestellte Problem einfacher und –

von der Reaktionszeit des Datenbanksystems her – schneller mit einem JOIN lösen.

Des Weiteren gibt es die Funktionen ALL, ANY, SOME und EXISTS für die Verwendung bei Unterabfragen. Die Probleme, die damit gelöst werden können, lassen sich meist auch anders (und in der Regel auch einfacher) lösen.

6.1.10 Reihenfolge der Klauseln

Wenn Sie mehrere Klauseln kombinieren, dann halten Sie sich an folgende Reihenfolge:

```
SELECT [DISTINCT] * | column_list
  FROM table_reference
  [WHERE predicates]
  [GROUP BY group_list]
  [HAVING having_condition]
  [ORDER BY order_list]
```

6.1.11 Datenänderung

Mit den Anweisungen INSERT, UPDATE und DELETE werden die Werte in einer Tabelle geändert.

INSERT

Mit INSERT wird ein neuer Datensatz in die Tabelle eingefügt. Benennen Sie zunächst die Spalten, in denen Werte eingefügt werden sollen, und anschließend die einzufügenden Werte in der gleichen Reihenfolge.

```
SELECT INTO t_art
     (nummer, bezeichnung)
  VALUES
     (8, "Telefon bei Partner(in)")
```

Bei einer INSERT-Anweisung kann auch eine Unterabfrage verwendet werden. Die folgende Anweisung kopiert alle Kunden mit der Kundennummer größer 120 in die Tabelle *alte_kunden*.

```
INSERT INTO alte_kunden
     (vorname, nachname)
  SELECT vorname, nachname
     FROM kunde
     WHERE nummer > 120
```

UPDATE

Mit der Anweisung UPDATE können Datensätze geändert werden.

```
UPDATE t_art
  SET nummer = 9,
    bezeichnung = "Telefon bei PartnerIn"
  WHERE nummer = 8
```

Wenn Sie eine UPDATE-Anweisung ohne WHERE-Klausel erstellen, dann werden alle Datensätze der Tabelle entsprechend geändert. Auf diese Weise können beispielsweise Spalten kopiert werden.

```
UPDATE kunde
  SET name = vorname || " " || nachname
```

Auch bei UPDATE-Anweisungen sind Unterabfragen erlaubt:

```
UPDATE mitarbeiter m
  SET m.bemerkung = "Über Handy erreichbar"
  WHERE m.nummer IN (SELECT t.mitarbeiter
    FROM tele t
    WHERE t.art = 4)
```

DELETE

Um einen oder mehrere Datensätze zu löschen verwenden Sie DELETE.

```
DELETE FROM mitarbeiter
  WHERE nummer = 8
```

Beachten Sie, dass das Löschen eines Datensatzes von einer Referenz verhindert werden kann. Um mehrere Datensätze zu löschen müssen Sie die WHERE-Klausel entsprechend gestalten:

```
DELETE FROM kunde
  WHERE ort = "Berlin"
```

Wenn Sie auf die WHERE-Klausel verzichten, werden alle Datensätze der betreffenden Tabelle gelöscht.

Auch bei einer DELETE-Anweisung kann mit Unterabfragen gearbeitet werden:

```
DELETE FROM t_tele
  WHERE art = (SELECT nummer
    FROM t_art
    WHERE bezeichnung = "Handy")
```

6.1.12 Definition von Metadaten

Mit Hilfe von SQL-Anweisungen können auch Tabellen und Indizes erstellt und gelöscht werden.

CREATE TABLE

Mit CREATE TABLE wird eine neue Tabelle erstellt. In Klammern werden die Spaltenbezeichner mit dem jeweiligen Spaltentyp aufgeführt, außerdem kann ein Primärschlüssel – im Beispiel für die Spalte *nummer* – erstellt werden.

```
CREATE TABLE t_test
  (nummer AUTOINC,
  name CHAR(20),
  wert INTEGER,
  datum DATE,
  betrag FLOAT,
  PRIMARY KEY (nummer))
```

LocalSQL kennt folgende Feldtypen:

- Die Ganzzahlentypen SMALLINT, INTEGER und AUTOINC. Letzterer ist ein selbstinkrementierender Typ.
- Die Festkommatypen DECIMAL[(s[, p])], NUMERIC[(s[, p])] sowie – als Sonderfall – den Währungstyp MONEY.
- Den Gleitkommatyp FLOAT(s, p).
- Die Stringtypen CHARACTER(length) und VARCHAR(length).
- Den booleschen Typ BOOLEAN.
- DATE für ein Datum, TIME für eine Uhrzeit und TIMESTAMP für eine Kombination aus Datum und Uhrzeit.
- BYTES(length) für untypisierte Binärdaten und BLOB(length, type) für typisierte Binärdaten.

Bei manchen Feldtypen müssen Sie in Klammern Angaben über die Größe, die Präzision der Darstellung oder den BLOB-Typ machen.

ALTER TABLE

Um eine Tabelle zu ändern, verwenden Sie ALTER TABLE. Mittels DROP können Sie Spalten löschen, mittels ADD Spalten hinzufügen. Sie können DROP und ADD in derselben ALTER TABLE-Anweisung verwenden.

```
ALTER TABLE t_test
  DROP datum, betrag,
  ADD neu CHAR(20)
```

Beachten Sie, dass die Existenz von Fremdschlüsseln das Löschen von Spalten verhindern kann.

DROP TABLE

Mit DROP TABLE wird eine Tabelle gelöscht.

```
DROP TABLE t_test
```

Beachten Sie, dass die Existenz von Fremdschlüsseln das Löschen von Tabellen verhindern kann.

CREATE INDEX

Um einen Index zu erstellen, verwenden Sie die Anweisung CREATE INDEX. Sie müssen den Index benennen und die Tabelle sowie die Spalte(n) angeben, für welche der Index gebildet werden soll.

```
CREATE INDEX i_nachname
  ON kunde (nachname)
```

Um einen absteigenden Index zu erstellen, verwenden Sie die DESC-Klausel, um einen Schlüssel (jeder Wert darf nur einmal vorkommen) zu erstellen, die UNIQUE-Klausel. Ein Index kann auch über mehrere Spalten gebildet werden. Im folgenden Beispiel werden alle drei Optionen gemeinsam genutzt.

```
CREATE UNIQUE DESC INDEX i_name
  ON kunde (nachname, vorname)
```

DROP INDEX

Einen Index können Sie nicht ändern, sondern nur löschen und anschließend neu erstellen. Löschen Sie einen Index mit DROP INDEX.

```
DROP INDEX i_name
```

6.2 Arbeiten mit TQuery

In Kapitel 1 haben wir schon gesehen, wie man auf Tabellen zugreift und wie man mit Hilfe eines Parameters eine Master-Detail-Verknüpfung erstellt. *TQuery* kann aber noch einiges mehr:

6.2.1 Parameter

Mit Hilfe von Parametern kann man in manchen Fällen die Erstellung von Abfragen deutlich erleichtern. Insbesondere beim Einrichten von Master-Detail-Beziehungen sind sie recht hilfreich. Parameter sind – wie bei Pascal – Werte, die einer Abfrage übergeben werden. Wir wollen uns das gleich mal in der Praxis ansehen.

Nehmen wir einmal an, dass alle Datensätze gesucht werden sollen, deren Spalte *vorname* gleich dem ist, was der Anwender in *edtVorname* eingibt, und deren Spalte *nachname* gleich dem ist, was in *edtNachname* angegeben wurde. Die entsprechende Anweisung könnte man folgendermaßen konstruieren:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Query1 do
  begin
    SQL.Clear;
    SQL.Add('SELECT * FROM testadr');
    SQL.Add('  WHERE (vorname = "' + edtVorname.text + '")');
    SQL.Add('    AND (nachname = "' + edtNachname.text + '")');
    Open;
  end;
end;
```

Man kann das Problem aber auch mittels der Verwendung von Parametern lösen:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Query1 do
  begin
    SQL.Clear;
    SQL.Add('SELECT * FROM testadr');
    SQL.Add('  WHERE (vorname = :vorname');
    SQL.Add('    AND (nachname = :nachname');
    Params[0].AsString := edtVorname.Text;
    Params[1].AsString := edtNachname.Text;
    Open;
  end;
end;
```

In der SQL-Anweisung werden die Parameter *vorname* und *nachname* durch Voranstellung eines Doppelpunktes definiert, diesen werden dann über die Eigen-

schaft *Params* die Texte von *edtVorname* bzw. *edtNachname* zugewiesen. In der Eigenschaft *Params* müssen die Parameter in derselben Reihenfolge aufgeführt sein, wie sie in der SQL-Anweisung definiert sind. Da dies etwas fehleranfällig ist, lässt sich auch die Methode *ParamByName* verwenden:

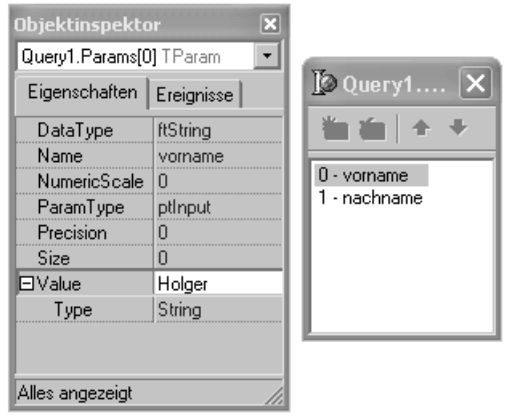
```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ...
    ParamByName('vorname').AsString := edtVorname.Text;
    ParamByName('nachname').AsString := edtNachname.Text;
    ...
end;
```

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?).

Parameter zur Entwurfszeit

Der Einsatz von Parametern bedingt natürlich nicht, der *TQuery*-Komponente die SQL-Anweisung erst zur Laufzeit zuzuweisen. Meist wird man die SQL-Anweisung zur Entwurfszeit setzen und die Werte für die Parameter dann zur Laufzeit zuweisen.

Es besteht jedoch auch die Möglichkeit – und sei es, um die SQL-Anweisung zu testen –, den Parametern bereits zur Entwurfszeit Werte zuzuweisen. Dazu wird der Eigenschaftseditor der Eigenschaft *Params* aufgerufen:



In den einfacheren Fällen reicht es, der Eigenschaft *Value* den korrekten Wert zuzuweisen. Manchmal muss auch *DataType* und/oder *ParamType* gesetzt werden.

Parameter bei INSERT und UPDATE

Um Daten in einer *TQuery*-Datenmenge zu ändern, kann man in manchen Fällen *RequestLive* auf *true* setzen. Dies wird beispielsweise bei JOINS keine aktualisierbare Datenmenge bewirken, und bei manchen SQL-Servern ist es auch keine gute Idee, *RequestLive* auf *true* zu setzen.

In solchen Fällen wird man dann mit INSERT und UPDATE arbeiten müssen. Die einzufügenden Werte werden dabei für gewöhnlich mit Parametern übergeben. Die folgende Prozedur stammt aus dem Beispielprogramm in Kapitel 18:

```
procedure DoEintrag;
begin
  with qrWork do
  begin
    SQL.Text := 'INSERT INTO t_eintrag';
    SQL.Add('VALUES (:ID, :Name, :Memo)');
    ParamByName('ID').AsInteger := iEin;
    ParamByName('Name').AsString := edtName.Text;
    ParamByName('Memo').Assign(Memo1.Lines);
    ExecSQL;
  end; {with qrWork do}
end; {procedure DoEintrag}
```

Um eine SQL-Anweisung auszuführen, die keine Datenmenge liefert, wird nicht *Open*, sondern *ExecSQL* verwendet.

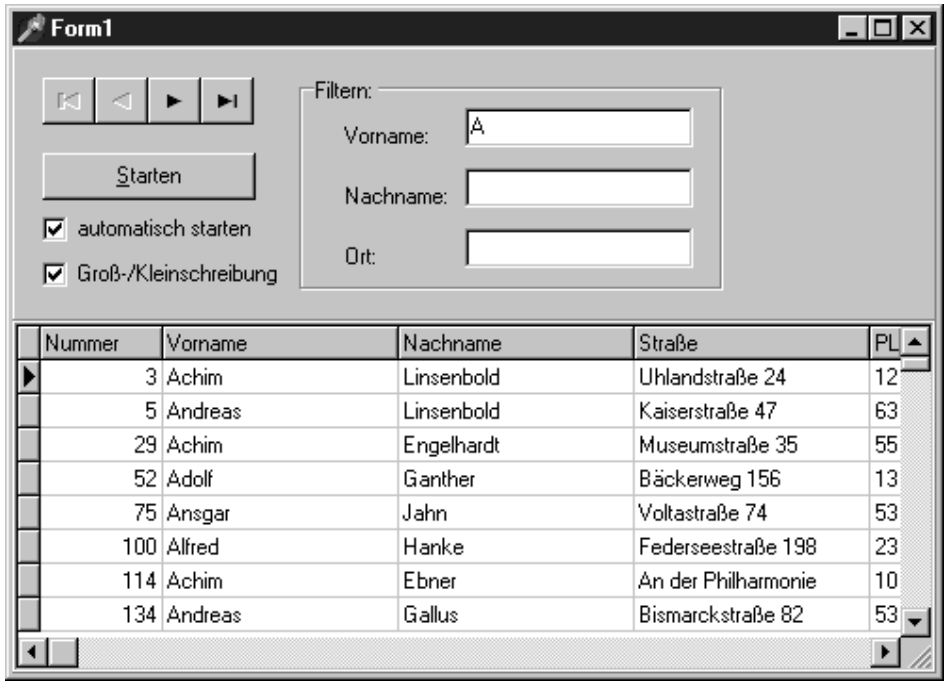
6.2.2 Datenmengen filtern

Der Einsatz von Parametern erleichtert auch das Filtern von Datenmengen. Bei großen Tabellen ist es wenig effektiv, einzelne Datensätze von Hand zu suchen, zumal es gerade bei Client-Server-Systemen auch nicht möglich sein dürfte, alle Datensätze zum Client zu übertragen. Hier muss die Datenmenge gefiltert werden.

In unserem Beispiel soll die Tabelle *testadr* nach den Spalten *Vorname*, *Nachname* und *Ort* gefiltert werden. Da Sie die grundsätzliche Vorgehensweise beim Erstellen der entsprechenden SQL-Anweisungen bereits kennen, wollen wir uns gleich mit den Details beschäftigen.

Groß- und Kleinschreibung ignorieren

In der Regel macht es keinen Sinn, bei der Suche nach Namen die Groß- und Kleinschreibung zu beachten. Je nach Zustand von *chkGross* soll deshalb diese bei der Abfrage berücksichtigt werden oder nicht.




```
procedure TForm1.chkGrossClick(Sender: TObject);
begin
  if chkGross.Checked then
  begin
    with Query1.SQL do
    begin
      Clear;
      Add('SELECT * FROM testadr');
      Add('  WHERE vorname LIKE :vor');
      Add('    AND nachname LIKE :nach');
      Add('    AND ort LIKE :ort');
    end;
  end {if chkGross.Checked}
  else
  begin
    with Query1.SQL do
    begin
      Clear;
      Add('SELECT * FROM testadr');
```

```
      Add('  WHERE UPPER(vorname) LIKE UPPER(:vor)');
      Add('    AND UPPER(nachname) LIKE UPPER(:nach)');
      Add('    AND UPPER(ort) LIKE UPPER(:ort)');
    end;
  end; {else chkGross.Checked}
end; {procedure TForm1.chkGrossClick}
```

Mit Hilfe der SQL-Funktion UPPER werden dabei sowohl alle Zeichen der betreffenden Spalte als auch diejenigen des Such-Strings in Großbuchstaben umgewandelt.

Automatisch Starten

Des Weiteren soll die Möglichkeit bestehen, dass die Abfrage dann startet, wenn der Anwender den kompletten Such-String eingegeben hat. Auch hier soll das Kriterium wieder das Einhalten einer kleinen Pause (500 ms) sein, damit nicht nach der Eingabe eines jeden Buchstabens eine neue Abfrage gestartet wird.

```
procedure TForm1.EditChange(Sender: TObject);
begin
  Query1.Close;
  Timer1.Enabled := false;
  Timer1.Enabled := true;
end;
```

Alle drei *TEdit*-Instanzen werden mit *EditChange* verknüpft. Wird irgendwo ein Buchstabe eingegeben, dann wird die Abfrage geschlossen und der Timer gestartet.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Query1 do
  begin
    ParamByName('vor').AsString := edtVor.Text + '%';
    ParamByName('nach').AsString := edtNach.Text + '%';
    ParamByName('ort').AsString := edtOrt.Text + '%';
    Open;
  end;
end;
```

Wird *Button1* betätigt, werden die Texte der *TEdit*-Instanzen – zuzüglich eines *%*-Zeichens – den Parametern von *Query1* zugewiesen. Anschließend wird die Abfrage gestartet.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if chkAuto.Checked = true
    then Button1Click(Sender);
end;
```

Wird ein automatisches Starten der Abfrage gewünscht, dann wird nach Ablauf der Timer-Zeit *Button1Click* aufgerufen.

6.2.3 Heterogene Joins

Bei einem normalen Join werden die Daten mehrerer Tabellen aus einer Datenbank zusammengefügt.

Wenn Sie die Daten aus mehreren Datenbanken zusammenfügen möchten, dann müssen Sie einen heterogenen Join erstellen. Dazu stellen Sie dem Tabellenbezeichner den Bezeichner des BDE-Alias der entsprechenden Tabelle voran.

```
SELECT
  e.lastname, p.proj_id
  FROM ":dbdemos:employee" e,
    INNER JOIN ":iblocal:employee_project" p
      ON e.empno = p.emp_no
```

Mit heterogenen JOINS kann man beispielsweise Kundendaten, die sich auf einem SQL-Server befinden, mit Zusatzinformationen ergänzen, die in einer lokalen Datenbank gespeichert sind. Man erspart sich dadurch, in der lokalen Datenbank einen eigenen Kundenstamm zu pflegen.

6.2.4 Die Eigenschaft Filter bei SQL-Servern

Bei *TQuery* können Sie eine Beschränkung der Datensätze entweder mit einer WHERE-Klausel oder mit der *TDataSet*-Eigenschaft *Filter* durchführen. Wenn Sie die Eigenschaft *Filter* verwenden, dann wird der komplette Datenbestand von der *TQuery*-Instanz gefiltert. Bei SQL-Servern muss dazu der komplette Datenbestand auf den Client geladen werden – das kann dauern.

Verwenden Sie deshalb nach Möglichlichkeit die Eigenschaft *Filter* in Zusammenhang mit *TQuery* und SQL-Servern allenfalls bei Tabellen mit wenigen Datensätzen.

6.3 Referenz TQuery

TQuery ist von *TDBDataSet* (Kapitel 5.1) sowie von TDataSet (Kapitel 3.1) abgeleitet und erbt somit alle dort eingeführten Eigenschaften, Methoden und Ereignisse.

Die SQL-Anweisung

- SQL (Eigenschaft, veröffentlicht)

  ```
  property SQL: TStrings;
  ```

  Die Eigenschaft *SQL* erhält die SQL-Anweisung für die Abfrage oder die Datenmanipulation. Eine Abfrage wird mit *Open* geöffnet, eine Datenmanipulation mit *ExecSQL* ausgeführt. Die Datenbanksprache *SQL* ist in Kapitel 6.1 beschrieben worden.

- ExecSQL (Methode)

  ```
  procedure ExecSQL;
  ```

  Wird der Eigenschaft *SQL* eine Anweisung zur Datenmanipulation zugewiesen (Erstellen von Tabellen, Ändern von Daten), dann kann *TQuery* keine Datenmenge öffnen, und deshalb darf weder die Methode *Open* verwendet noch die Eigenschaft *Active* auf *true* gesetzt werden. Um eine solche SQL-Anweisung auszuführen, muss die Methode *ExecSQL* eingesetzt werden.

- RequestLive (Eigenschaft, veröffentlicht)

  ```
  property RequestLive: Boolean default false;
  ```

  Wird die Eigenschaft *RequestLive* auf *true* gestellt, dann wird versucht, eine Datenmenge zu öffnen, die sich bearbeiten lässt, bei der also wie bei *TTable* auch Datenmanipulationen – Datensätze einfügen, ändern und löschen – vorgenommen werden können. Ist *RequestLive* gleich *false*, dann können die Datensätze nur gelesen werden, was bei Abfragen in der Regel völlig ausreicht.

  Bei vielen Abfragen, beispielsweise bei Aggregat-Funktionen, Gruppierungen und JOINS kann keine Datenmenge geöffnet werden, die sich manipulieren lässt. Hier hilft gegebenenfalls die Verwendung der Komponente *TUpdateObject*. Ist es möglich, die Datenmenge zu bearbeiten, dann ist die Eigenschaft *CanModify* gleich *true*.

  Bei Client-Server-Systemen ist der Einsatz von *RequestLive* nicht unbedingt zu empfehlen, siehe auch Kapitel 13. Dagegen beschleunigt *RequestLive* bei lokalen Datenbanken fast immer die Ausführung einer Abfrage – siehe auch Kapitel 5.2.3.

- Text (Eigenschaft, öffentlich, nur Lesen)

  ```
  property Text: PChar;
  ```

  Die Eigenschaft *Text* gibt an, welche SQL-Anweisung zuletzt an die BDE oder an den Server geschickt wurde. Die entsprechenden Werte für die Parameter sind dabei schon eingefügt, so dass diese Eigenschaft eine Hilfe bei der Fehlersuche darstellt, wenn bei der Verwendung von Parametern nicht alles wie geplant läuft.

- Refresh (Methode)

  ```
  procedure Refresh;
  ```

  Die Methode *Refresh* können Sie bei *TQuery* vergessen – schließen Sie die Abfrage mit *Close* und führen Sie mit *Open* die Abfrage erneut durch.

- Constrained (Eigenschaft, veröffentlicht)

  ```
  property Constrained: Boolean default false;
  ```

  Das Setzen der Eigenschaft *Constrained* auf *true* verhindert, dass Abfragedatensätze durch eine UPDATE-Anweisung so geändert werden, dass sie nicht mehr der Abfragebedingung entsprechen. Lautet beispielsweise die Abfrage

  ```
  SELECT * FROM testadr
    WHERE nummer < 100,
  ```

  dann kann die Datensatznummer nicht auf einen Wert größer/gleich hundert gesetzt werden (wenn *Constrained* gleich *true* ist). *Constrained* ist nur im Zusammenhang mit dBase und Paradox von Bedeutung.

- Unidirectional (Eigenschaft, veröffentlicht)

  ```
  property UniDirectional: Boolean default false;
  ```

  Zum Thema *UniDirectional* erzählt die Online-Hilfe ziemlichen Unsinn – zumindest was die Datenbanksysteme *InterBase* und *Paradox* betrifft. Das Setzen von *UniDirectional* auf *true* soll eine Ergebnismenge liefert, in der nur vorwärts gescrollt werden kann. Durch den Verzicht auf die Zwischenspeicherung soll die Leistung verbessert werden.

  Zumindest bei den beiden von mir getesteten Datenbanksystemen kann – im Gegensatz zu den wirklich unidirektionalen Komponenten von dbExpress – ganz problemlos in beide Richtungen gescrollt werden, die Verwendung von *TDBGrid* ist überhaupt kein Problem und eine Differenz bei den Ausführungszeiten konnte auch nicht nachgewiesen werden (die Unterschiede waren geringer als die Unterschiede zwischen den einzelnen Messungen).

Parameter

- Params (Eigenschaft, veröffentlicht)

  ```
  property Params[Index: Word]: TParams;
  ```

  Mit der Eigenschaft *Params* kann auf die Parameter einer Abfrage zugegriffen werden. Um Verwechslungen zu vermeiden, sollten Sie dafür jedoch die Methode *ParamByName* bevorzugen.

- ParamByName (Eigenschaft, Methode)

  ```
  function ParamByName(const Value: string): TParam;
  ```

  Statt über den Index – wie bei der Eigenschaft Params – können Parameter auch über den Parameternamen gesetzt werden. Dazu dient die Methode *ParamByName*.

- DataSource (Eigenschaft, veröffentlicht)

  ```
  property DataSource: TDataSource;
  ```

  Die Eigenschaft *DataSource* wird benötigt, um Master-Detail-Beziehungen mit Hilfe von *TQuery*-Komponenten herzustellen. Näheres siehe nächster Abschnitt.

- ParamCount (Eigenschaft, öffentlich, nur Lesen)

  ```
  property ParamCount: Word;
  ```

  Mit *ParamsCount* kann die Anzahl der vorhandenen Parameter abgefragt werden.

- Prepare, Unprepare (Methoden), Prepared (Eigenschaft, öffentlich)

  ```
  procedure Prepare;
  procedure Unprepare;
  property Prepared: Boolean;
  ```

  Mit der Methode *Prepare* können Sie eine Abfrage vorbereiten, mit *Unprepare* die Vorbereitung wieder aufheben. Alternativ dazu kann die Eigenschaft *Prepared* verwendet werden – mit ihrer Hilfe kann man auch ermitteln, ob die Abfrage bereits vorbereitet wurde.

  Zu welchem Zweck können Abfragen überhaupt vorbereitet werden? Wenn ein SQL-Server eine Anweisung erhält, dann muss er diese interpretieren und daraus einen Query-Plan erstellen (welche Daten werden in welcher Reihenfolge ermittelt). Die Client-DLL des Client-Server-Systems stellt dafür eine eigene Prozedur zur Verfügung, die von der BDE normalerweise automatisch aufgerufen wird.

Wird nun ein und dieselbe SQL-Anweisung wiederholt aufgerufen, dann muss sie eigentlich nicht neu interpretiert werden, auch wenn sich die Parameter geändert haben sollten. Um zu verhindern, dass die SQL-Anweisung automatisch vorbereitet wird, rufen Sie vorher *Prepare* auf – bevor Sie die SQL-Anweisung ändern, müssen Sie dann jedoch mit *Unprepare* die Vorbereitung wieder aufheben.

Wie stark die Ausführung einer SQL-Anweisung durch die Vorbereitung beschleunigt wird, hängt natürlich davon ab, wie lange die Vorbereitung dauert. Komplizierte Anweisungen profitieren dabei mehr als einfache.

Wir wollen das an der einfachsten parametrisierten Abfrage testen:

```
SELECT nachname
  FROM t_kunde
  WHERE nachname LIKE :name
```

Die folgende Tabelle vergleicht die Ausführungszeiten von explizit vorbereiteten und nicht explizit vorbereiteten Abfragen:

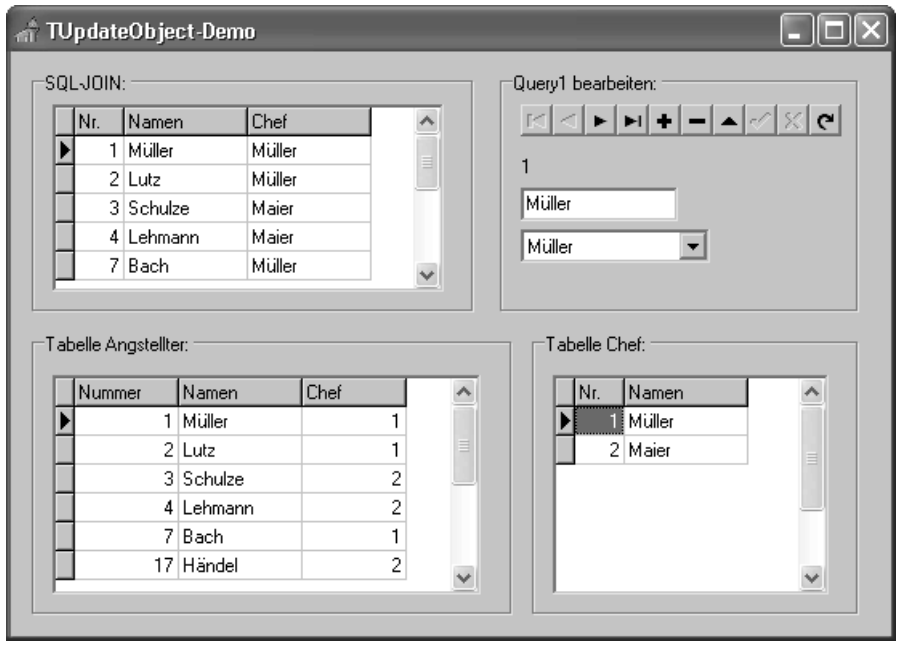
	Prepared	Unprepared
Prepare	15	
A	284	293
B	283	295
C	283	293
D	283	295
E	283	294
F	284	295
G	283	294
H	283	295
I	283	294
J	283	294
Unprepare	1	

Im Schnitt sind die explizit vorbereiteten Abfragen etwas 11 bis 12 Millisekunden schneller – das ist knapp die Zeit, die *Prepare* zur Ausführung benötigt.

Komplizierte Anweisungen, die häufiger ausgeführt werden, sollten also vorbereitet werden – alternativ kann man auch eine VIEW oder eine STORED PROCEDURE erstellen.

6.4 Die Komponente TUpdateSQL

Bei normalisierten Datenbanken muss man in der Regel auf JOINS zurückgreifen, um aussagekräftige Daten zurückzuerhalten. Als Beispiel dienen hier die beiden Tabellen *Angestellter* und *Chef* aus der folgenden Abbildung:



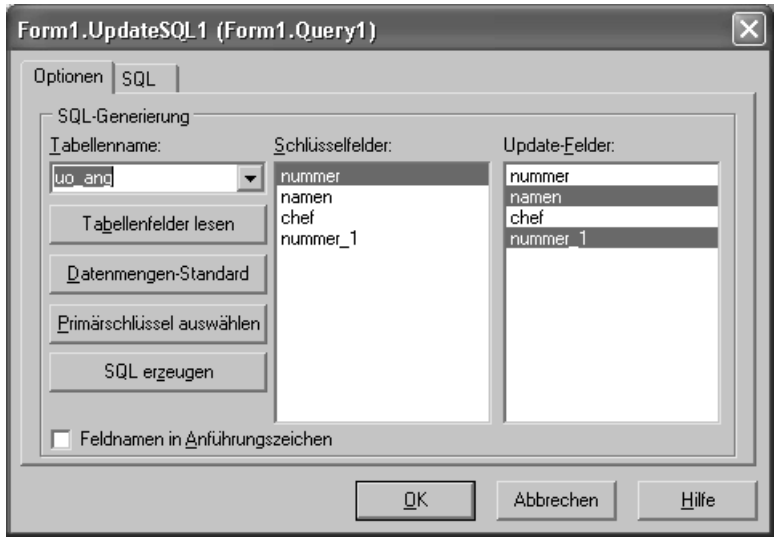
Um hier im *DBGrid* links oben aussagekräftige Daten zu erhalten, wurde die folgende SQL-Anweisung formuliert.

```
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. Man könnte nun *TQuery*- und *TTable*-Instanzen so verknüpfen, dass die Anzeige über einen JOIN und die Eingabe auf Tabellenebene erfolgt. Einfacher geht es jedoch mit der Komponente *TUpdateSQL*.

Zunächst einmal wird eine Instanz der Komponente *TUpdateSQL* in das Projekt eingefügt, danach werden zwei Eigenschaften der dazugehörenden *TQuery*-Komponente geändert: Die Eigenschaft *CachedUpdates* 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.



Wählen Sie unter *Schlüsselfelder* das Feld oder die Felder, die den Primärschlüssel der zu ändernden Tabelle bilden. Anhand dieser Schlüsselfelder werden die zu ändernden oder zu löschenden Datensätze identifiziert.

Die Update-Felder sind diejenigen Felder, die im INSERT- oder UPDATE-Fall geändert werden sollen. Da wir für die Angestelltennummer ein selbstinkrementierendes Feld haben, darf diese nicht in die Menge der Update-Felder.

Nun klicken wir auf *SQL erzeugen* und betrachten zunächst die Anweisungen, die hier automatisch erstellt werden:

```
update uo_ang
set
  namen = :namen,
  nummer_1 = :nummer_1
where
  nummer = :OLD_nummer
```

```
insert into uo_ang
  (namen, nummer_1)
values
  (:namen, :nummer_1)

delete from uo_ang
where
  nummer = :OLD_nummer
```

Die Tabelle *uo_ang* hat nun jedoch keine Spalte *nummer_1*, dementsprechend sind die Anweisungen umzuformulieren:

```
insert into uo_ang
  (namen, chef)
values
  (:namen, :nummer_1)

update uo_ang
set
  namen = :namen,
  chef = :nummer_1
where
  nummer = :OLD_nummer
```

Alles, was für dieses Programm an Quelltext benötigt wird, ist die Prozedur *Query1AfterPost*. Da bei der Verwendung von *TUpdateSQL* zwingend die Eigenschaft *CachedUpdates* auf *true* gesetzt werden muss, muss die Methode *ApplyUpdates* aufgerufen werden, damit die Änderungen auch übernommen werden. Des Weiteren müssen die Abfrage und die *Table2* aktualisiert werden. Weisen Sie diese Prozedur auch dem Ereignis *OnAfterDelete* zu.

```
procedure TForm1.Query1AfterPost(DataSet: TDataSet);
begin
  with Query1 do
  begin
    ApplyUpdates;
    Close;
    Open;
  end;
  Table2.Refresh;
end;
```

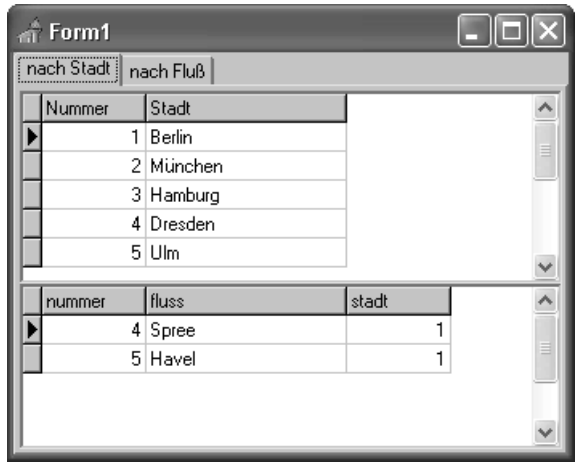
6.4.1 0/1/n : 0/1/m-Master-Detail-Beziehungen

Wie wir in Kapitel 2.2 gelernt haben, verwendet man bei *0/1/n : 0/1/m*-Beziehungen in der Regel eine eigene Beziehungstabelle. Wir wollen nun betrachten, wie man mit *TQuery* hier eine Master-Detail-Beziehung aufbaut und diese mit *TUpdateSQL* aktualisierbar macht.

Für unser Beispiel nehmen wir eine Tabelle mit Flüssen und eine Tabelle mit Städten. Des Weiteren soll es eine Tabelle geben, die anzeigt, welche Flüsse durch welche Städte fließen. Mit Hilfe unserer Master-Detail-Beziehung wollen wir einmal die Städte anzeigen, durch die ein bestimmter Fluss fließt, und zum anderen die Flüsse, die durch eine bestimmte Stadt fließen. Die folgende Abbildung zeigt den Aufbau der drei Tabellen:

stadt.db		
nummer	+	primär
stadt	A 20	
fluss.db		
nummer	+	primär
fluss	A 20	
suf.db		
stadt	I	Referenz auf stadt.db
fluss	I	Referenz auf fluss.db

Der Aufbau des Formulars (siehe Bild 3.8) ist nicht unter Schönheitsaspekten erstellt worden, dafür sehen Sie sehr deutlich die zugrunde liegenden Tabellen.



Die SQL-Anweisung der Detailabfrage lautet wie folgt:

```
SELECT
     f.nummer,
     f.fluss,
     u.stadt
  FROM fluss f, suf u
  WHERE (f.nummer = u.fluss)
     AND (u.stadt = :nummer)
```

Wie Sie sehen, wird zwischen der Verknüpfungstabelle *suf* und der Detailtabelle *fluss* ein fester INNER JOIN hergestellt, während zwischen *suf* und der Master-Tabelle *stadt* eine Master-Detail-Beziehung aufgebaut wird.

Nun könnte man auf die Idee kommen, die Tabellen aus diesem Formular heraus bearbeiten zu wollen, beispielsweise, um neue Flüsse und/oder neue Städte hinzuzufügen. Während man bei den Master-Abfragen lediglich die Eigenschaft *RequestLive* auf *true* setzt (oder gleich *TTable*-Komponenten verwendet), ist die Bearbeitung der Tabelle *suf* deutlich aufwändiger und erfordert auch die Komponete *TUpdateSQL:*

Zunächst muss auch hier diejenige *TQuery*-Komponente, welche die Detailtabelle anzeigt, über die Eigenschaft *UpdateObject* mit einer *TUpdateSQL*-Komponente verknüpft werden. Auch muss die Eigenschaft *CachedUpdates* dieser *TQuery*-Komponente auf *true* gestellt werden.

Datensätze ändern

Vergleichsweise einfach ist das Ändern von Datensätzen, was beispielsweise dann erforderlich wird, wenn ein Stadt- oder ein Flussname falsch geschrieben wurde.

```
UPDATE fluss
SET
  Fluss = :Fluss
WHERE
  Nummer = :OLD_Nummer
```

Anschließend müssen die Änderungen in die Tabelle geschrieben werden:

```
procedure TForm1.Query2AfterPost(DataSet: TDataSet);
begin
  with Query2 do
  begin
    if FPostAbbruch = false
      then ApplyUpdates
      else CancelUpdates;
```

```
    Close;
    Open;
    FPostAbbruch := false;
  end;
end;
```

Was es mit dem Feld *FPostAbbruch* auf sich hat, werden wir später noch sehen. Momentan können wir davon ausgehen, dass *FPostAbbruch* den Wert *false* hat, und somit mit *ApplyUpdates* die Änderungen in die Tabelle geschrieben werden. Um die Anzeige zu aktualisieren, ist die wohl bekannte *Close*-*Open*-Kombination erforderlich.

Datensätze löschen

Ähnlich einfach geht das Löschen von Datensätzen: Hier wird jedoch davon ausgegangen, dass nicht der Fluss gelöscht werden soll, sondern nur die Verknüpfung der aktuellen Stadt mit dem jeweiligen Fluss. Es muss also nur in der Tabelle *suf* gelöscht werden.

```
DELETE FROM suf
  WHERE (fluss = :OLD_Nummer)
    AND (stadt = :OLD_stadt)
```

Nach dem Löschen eines Datensatzes wird das Ereignis *OnAfterPost* nicht aufgerufen. Somit ist es erforderlich, die entsprechenden Anweisungen in eine *OnAfterDelete*-Ereignisbehandlungsroutine zu kopieren.

```
procedure TForm1.Query2AfterDelete(DataSet: TDataSet);
begin
  with Query2 do
  begin
    ApplyUpdates;
    Close;
    Open;
  end;
end;
```

Datensätze einfügen

Kompliziert wird es dagegen beim Einfügen neuer Datensätze. Hier gibt es prinzipiell zwei Möglichkeiten:

- Der Fluss existiert schon, es muss also nur noch eine Referenz in die Tabelle *suf* geschrieben werden.

- Der Fluss existiert noch nicht. Er muss also erst in der Tabelle *fluss* angelegt werden, anschließend ist ein entsprechender Eintrag in *suf* nötig.

```
procedure TForm1.Query2BeforePost(DataSet: TDataSet);
begin
  if Query2.FieldByName('fluss').AsString = '' then
  begin
    Query2.Cancel;
    FPostAbbruch := true;
    Exit;
  end;
```

Zunächst wird geprüft, ob überhaupt ein neuer Flussname eingegeben wurde, oder ob nicht beim Scrollen über das Ende der Tabelle hinaus ein *OnBeforePost*-Ereignis ausgelöst wurde. Ist dies der Fall, muss die unbeabsichtigte Bearbeitung mit *Cancel* abgebrochen werden. Durch das Setzen des Feldes *FPostAbbruch* wird darüber hinaus sichergestellt, dass in der *OnAfterPost*-Ereignisbehandlungsroutine *CancelUpdates* anstatt *ApplyUpdates* aufgerufen wird.

```
  with Query5 do
  begin
    Close;
    SQL.Clear;
    SQL.Add('SELECT * FROM fluss');
    SQL.Add('  WHERE fluss = :fluss');
    ParamByName('fluss').AsString
      := Query2.FieldByName('fluss').AsString;
    Open;
    if Query5.RecordCount > 0
      then Query2.FieldByName('nummer').AsString
        := Query5.FieldByName('nummer').AsString
    else
    begin
      Close;
      SQL.Clear;
      SQL.Add('INSERT INTO fluss (fluss) VALUES (:fluss)');
      ParamByName('fluss').AsString
        := Query2.FieldByName('fluss').AsString;
      ExecSQL;
      Query2BeforePost(Query2);
    end;
  end; {with Query5 do}
end; {procedure TForm1.Query2BeforePost}
```

Mit *Query5* – einer neu einzufügenden *TQuery*-Komponente – wird zunächst nach dem Flussnamen in der Tabelle *fluss* gesucht. Ist dieser vorhanden, also die Eigenschaft *RecordCount* größer null, dann wird der Spalte *nummer* von *Query2* dessen Datensatznummer zugewiesen.

Ist der Flussname jedoch noch nicht vorhanden, wird er mit einer entsprechenden SQL-Anweisung in die Tabelle eingefügt. Es sei hier darauf hingewiesen, dass Parameter auch in Verbindung mit *ExecSQL* funktionieren. Wenn der Flussname eingefügt ist, ruft sich diese Prozedur selbst auf, damit die Nummer des neuen Datensatzes der *Query2*-Spalte *nummer* zugewiesen wird.

Der Komponente *UpdateSQL1* obliegt es dann, den Verknüpfungsdatensatz in die Tabelle *suf* zu schreiben:

```
INSERT INTO suf
  (stadt, fluss)
VALUES
  (:stadt, :Nummer)
```

6.4.2 Referenz TUpdateObject

- DeleteSQL, InsertSQL, ModifySQL (veröffentlicht, TUpdateObject)

  ```
  property DeleteSQL: TStrings;
  property InsertSQL: TStrings;
  property ModifySQL: TStrings;
  ```

  Mit diesen drei Eigenschaften werden die SQL-Anweisungen vorgegeben, welche von der Komponente verwendet werden.

- Query (öffentlich, nur Lesen, TUpdateObject)

  ```
  property Query[UpdateKind: TUpdateKind]: TQuery;
  ```

  Die Komponente *TUpdateObject* verwaltet drei *TQuery*-Komponenten, welche die Aufgaben des Löschens, des Einfügens und des Änderns von Datensätzen zwischen sich aufteilen. Mit Hilfe der Eigenschaften *DeleteSQL*, *InsertSQL* und *ModifySQL* kann auf die Eigenschaft *SQL* dieser drei *TQuery*-Instanzen zugegriffen werden. Sollen andere Eigenschaften, Methoden oder Ereignisse verwendet werden, dann ist dazu die Array-Eigenschaft *Query* zu verwenden.

- DataSet (öffentlich, TUpdateObject)

```
property DataSet: TDataSet;
```

  Die Eigenschaft *DataSet* gibt die Datenmenge an, mit der die Komponente verbunden ist. Normalerweise wird die Eigenschaft zur Entwurfszeit automatisch gesetzt.

- SetParams, ExecSQL, Apply (Methoden, TUpdateObject)

```
procedure SetParams(UpdateKind: TUpdateKind);
procedure ExecSQL(UpdateKind: TUpdateKind);
procedure Apply(UpdateKind: TUpdateKind);
```

  Für gewöhnlich löst die über *DataSet* verbundene Datenmengenkomponente automatisch die Aktualisierung aus. Muss dies manuell geschehen, dann werden mit *SetParams* die Parameter gesetzt, mit *ExecSQL* wird die Aktualisierung vorgenommen. Mit *Apply* werden diese beiden Methoden hintereinander abgearbeitet.

6.5 Adressensuche

Zum Schluss wollen wir uns noch an ein größeres Demo-Programm wagen: 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 möglichst bei der Eingabe schon geprüft werden, ob die Adresse nicht vielleicht bereits 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 zehn Prozent. 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 Adressetiketten, 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*) sind so lang, dass sie in fast kein Formular passen. Hier wird je nach dem zur Verfügung stehenden Platz 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.
- 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 manchmal korrekt zugestellt.

- Des Weiteren müssen bisweilen zusätzliche Informationen angehängt werden. *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 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.

6.5.1 Generierung von Such-Strings

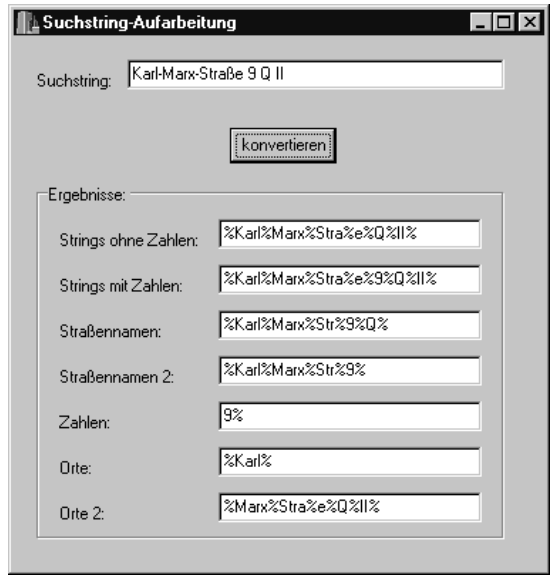
Um bei der Suche von Adressen eine hohe Trefferquote zu erzielen, muss die SELECT-Anweisung entsprechend gestaltet werden:

- 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 Teil-String *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.

Wir wollen uns nun einige Möglichkeiten ansehen, aus einer Adresse geeignete Such-Strings zu generieren. Im nächsten Unterkapitel soll dann ein Programm erstellt werden, das eine Tabelle auf doppelte Einträge hin untersucht.

Die nachfolgende Abbildung 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.

```
function likeqa(rein: string): string;
{Wandelt Strings mit Zahlen in SQL-Such-Strings um}
var
  i, j: integer;
begin
  result := rein;
  for i := 0 to length(rein) do
  begin
    if (('A' <= result[i]) and (result[i] <= 'Z'))
      or (('a' <= result[i]) and (result[i] <= 'z'))
      or (('0' <= result[i]) and (result[i] <= '9')) then
    else result[i] := '%'
  end; {for i := 0 to length(rein) do}
  result := '%' + Copy(result, 1, length(rein)) + '%';
  while pos('%%', result) > 0
    do delete (result, pos('%%', result), 1);
end; {function likeqa}
```

Zunächst wird der Variablen *result* (deren Inhalt als Ergebnis der Funktion zurückgegeben wird) der übergebene Parameter *rein* zugewiesen. Würden keine weiteren Anweisungen folgen, dann würde diese Funktion einen String nicht verändern.

Nun werden alle Stellen des Strings einzeln daraufhin überprüft, ob es sich um einen Buchstaben oder um eine Zahl handelt. Ist dies nicht der Fall, dann werden diese Zeichen durch ein *%*-Zeichen ersetzt. Soll die Funktion auch keine Zahlen zurückgeben, dann ist die dritte Zeile der *if*-Anweisung zu streichen.

Nun wird an den Anfang und an das Ende des Strings ein *%*-Zeichen gesetzt. Abschließend werden alle doppelten *%*-Zeichen eliminiert.

Suche nach Straßennamen

Mit der vorherigen Funktion können Straßennamen nicht optimal aufgearbeitet werden, deshalb soll hier eine neue Funktion erstellt werden.

```
function likeqs(rein: string): string;
{Wandelt Straßennamen in SQL-Such-Strings um}
var
  i, j, k, l: integer;
  zahla, zahln: boolean;
begin
  k := 0;
  i := pos('straße', rein);
  l := length(rein);
  if i = 0 then i := pos('Straße', rein);
  if i <> 0 then
  begin
    rein[i+3] := '%';
    rein[i+4] := '%';
    rein[i+5] := '%';
  end;
  i := pos('strasse', rein);
  l := length(rein);
  if i = 0 then i := pos('Strasse', rein);
  if i <> 0 then
  begin
    rein[i+3] := '%';
    rein[i+4] := '%';
    rein[i+5] := '%';
    rein[i+6] := '%';
  end;
```

Zunächst soll das Wort *Straße* durch *Str%%%* ersetzt werden. Da die Funktion *pos*, welche die Position eines Teil-Strings innerhalb eines Strings sucht, auf Groß- und Kleinschreibung achtet, muss sowohl nach *Straße* als auch nach *straße* gesucht werden. Außerdem wird nach der Schreibweise mit *ss* gesucht.

```
  result := '';
  for i := 1 to length(rein) do
  begin
    if (('A' <= rein[i]) and (rein[i] <= 'Z'))
      or (('a' <= rein[i]) and (rein[i] <= 'z'))
      or (('0' <= rein[i]) and (rein[i] <= '9')) then
```

```
    begin
      if ('0' <= rein[i]) and (rein[i] <= '9')
        then zahln := true
        else zahln := false;
      if zahln <> zahla then
      begin
        result := result + '%';
        k := k + 1;
        result := result + rein[i];
      end
      else result := result + rein[i];
      zahla := zahln;
    end {if (('A' <= rein[i])...}
    else result := result + '%';
  end; {for i := 0 to length(rein) do}
```

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.

```
  for i := round(0.75*(l+k)) - 3 to l+k do
  begin
    if (result[i] = 'I') or (result[i] = 'V')
      then result[i] := '%';
  end;
```

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.

```
  result := '%' + Copy(result, 1, l+k) + '%';
  while pos('%%', result) > 0
    do delete (result, pos('%%', result), 1);
end; {function likeqs}
```

Schließlich werden noch *%*-Zeichen am Anfang und am Ende eingefügt sowie alle doppelt vorkommenden *%*-Zeichen entfernt.

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.

```
function likeqh(rein: string): string;
{Wandelt Straßennamen in SQL-Such-Strings um,
  eliminiert nach der Hausnummer alle Zeichen}
var
  i, j, k, l: integer;
  zahla, zahln, ende: boolean;
begin
  ende := false;
  zahla := false;
  zahln := false;
  k := 0;
  i := pos('straße', rein);
  l := length(rein);

  {...Straße durch Str%%% ersetzen}
```

Ist die erste Verwendung einer Variablen eine *if*-Anweisung, dann muss diese Variable vorher entsprechend definiert werden, ansonsten sind die Ergebnisse nicht immer vorhersehbar. Das Ersetzen von *Straße* durch *Str%%%* wurde schon vorhin besprochen.

```
  result := '';
  for i := 1 to length(rein) do
  begin
    if ('0' <= rein[i]) and (rein[i] <= '9')
      then zahln := true
      else zahln := false;
    if (zahln = false) and (zahla = true) then ende := true;
    if ((('A' <= rein[i]) and (rein[i] <= 'Z'))
      or (('a' <= rein[i]) and (rein[i] <= 'z'))
      or (('0' <= rein[i]) and (rein[i] <= '9')))
      and (ende = false) then
    begin
      {...kopieren der Zeichen nach result}
    end {if (('A' <= rein[i])...}
```

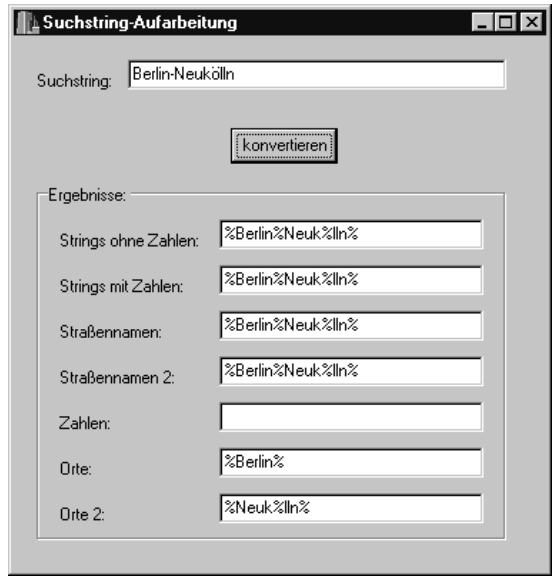
```
    else result := result + '%';
  end; {for i := 0 to length(rein) do}
  {Schlussbearbeitung}
end; {function likeqh}
```

Die Schleife gleicht in weiten Teilen der vorherigen, viele Anweisungen wurden hier nicht erneut aufgeführt. Die ersten beiden *if*-Anweisungen untersuchen den String auf die Stelle hin, an der die erste Ziffernfolge endet, also auf diejenige, an der das erste Mal von Zahlen auf andere Zeichen gewechselt wird. Hier wird dann die Variable *ende* auf *true* gesetzt, so dass künftig alle Zeichen durch ein *%*-Zeichen ersetzt werden.

Alle Angaben über Stockwerksnummern, Vorder- oder Hinterhaus bzw. Seitenflügel usw. werden hier ignoriert – eben alle Zeichen, welche je nach Laune der Anwender mal in die Datenbank eingegeben werden oder nicht.

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 Teil-String aufgeteilt wird:



```
function likeqo(rein: string): string;
{Wandelt Ortsnamen in SQL-Such-Strings um,
  eliminiert nach dem ersten Teil-String alle Zeichen}
var
  i: integer;
  ende: boolean;
begin
  ende := false;
  result := '';
  for i := 1 to length(rein) do
  begin
    if ((('A' <= rein[i]) and (rein[i] <= 'Z'))
      or (('a' <= rein[i]) and (rein[i] <= 'z'))
      or (rein[i] in ['ä','ö','ü','Ä','ö','ü','ß']))
      and (ende = false) then
    begin
      if ((('A' <= rein[i]) and (rein[i] <= 'Z'))
           or (('a' <= rein[i]) and (rein[i] <= 'z')))
        then result := result + rein[i]
        else result := result + '%';
    end {if (('A' <= rein[i])...}
    else
    begin
      result := result + '%';
      ende := true;
    end; {else (('A' <= rein[i])...}
  end; {for i := 0 to length(rein) do}
  result := '%' + result + '%';
  while pos('%%', result) > 0
    do delete (result, pos('%%', result), 1);
  if length(result) = 1 then result := '';
end; {function likeqo}
```

Diese Funktion geht davon aus, dass beim Auftreten des ersten Zeichens, welches kein Buchstabe ist, der erste Teil-String 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 Such-String *%K%,* sondern *%Köln%* generiert wird, müssen hier auch die deutschen Umlaute in die *if*-Anweisung aufgenommen werden, was hier mit einer Mengenoperation geschieht.

Die Funktion *likeqp* generiert aus einem Ortsnamen die zweite Hälfte als Such-String. 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 anderenfalls den Such-String *%* generieren. Somit würden bei der Suche alle Datensätze zurückgegeben, was wenig sinnvoll wäre.

Suche nach Nummern

Das prinzipielle Vorgehen beim Suchen nach Nummern wurde schon in Kapitel 3 vorgestellt – es wird einfach zwischen alle Zahlen ein *%*-Zeichen eingefügt.

```
function likeqz(rein: string): string;
{Wandelt Zahlenkette wie Telefonnummern
oder Postleitzahlen in SQL-Such-Strings}
var
  s: string;
  i: integer;
begin
  result := '';
  for i := 1 to length(rein) do
  begin
    s := Copy(rein, i, 1);
    if ('0' <= s) and (s <= '9') then result := result + s + '%';
  end; {for i := 1 to length(rein)}
end; {function likeqz}
```

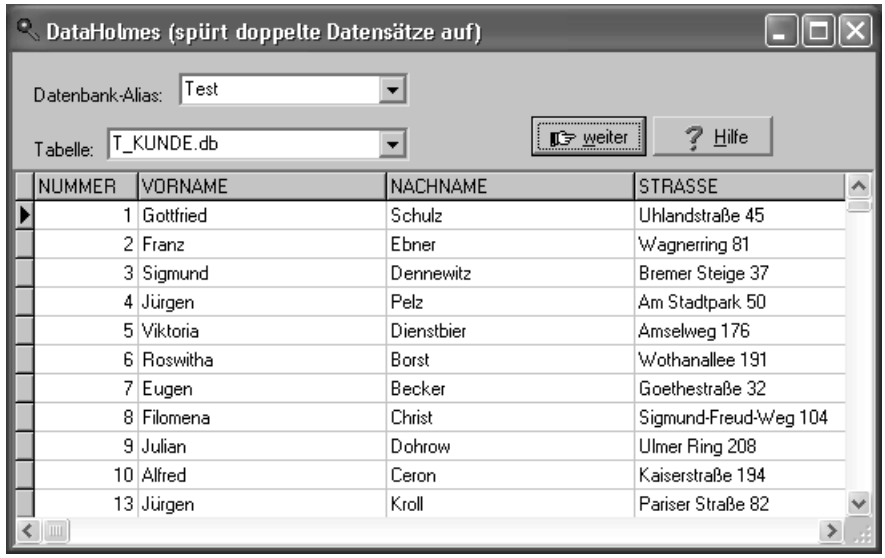
6.5.2 Entfernen doppelter Adressen

Nun soll – wie versprochen – ein Programm erstellt werden, mit dem sich doppelte Datensätze in einer Tabelle auffinden lassen und überflüssige entfernt werden können. 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.

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* werden der Alias der Datenbank und der Tabellenname ausgewählt



NUMMER	VORNAME	NACHNAME	STRASSE
1	Gottfried	Schulz	Uhlandstraße 45
2	Franz	Ebner	Wagnerring 81
3	Sigmund	Dennewitz	Bremer Steige 37
4	Jürgen	Pelz	Am Stadtpark 50
5	Viktoria	Dienstbier	Amselweg 176
6	Roswitha	Borst	Wothanallee 191
7	Eugen	Becker	Goethestraße 32
8	Filomena	Christ	Sigmund-Freud-Weg 104
9	Julian	Dohrow	Ulmer Ring 208
10	Alfred	Ceron	Kaiserstraße 194
13	Jürgen	Kroll	Pariser Straße 82

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.GetAliasNames(ComboBox1.Items);
  Notebook1.ActivePage := 'Verbinden';
end;
```

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.

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  Table1.Active := false;
  Session.GetTableNames(ComboBox1.Text, '', True, False,
    ComboBox2.Items);
  ComboBox2.Text := '';
end;
```

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.

```
procedure TForm1.ComboBox2Change(Sender: TObject);
begin
  DBText1.DataField := '';
  DBText2.DataField := '';
  DBText3.DataField := '';
  DBText4.DataField := '';
  DBText5.DataField := '';
  DBText6.DataField := '';
  DBText7.DataField := '';
  DBText8.DataField := '';
  DBText9.DataField := '';

  with Table1 do
  begin
    Active := false;
    DatabaseName := ComboBox1.Text;
    TableName := ComboBox2.Text;
    try
      Open;
    except
      ShowMessage('Tabelle lässt sich nicht öffnen');
      exit;
    end;
  end; {with Table1 do}

  with Table2 do
  begin
    Active := false;
    DatabaseName := ComboBox1.Text;
    TableName := ComboBox2.Text;
    Open;
  end; {with Table2 do}

  with Query1 do
  begin
    Active := false;
    DatabaseName := ComboBox1.Text;
  end;
end; {procedure TForm1.ComboBox2Change}
```

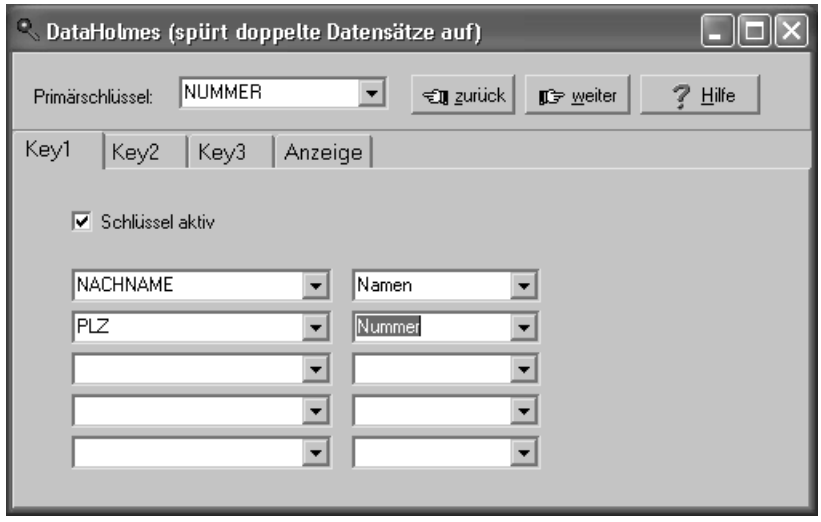
Auf einer anderen Seite von *Notebook1* werden wir später einige *TDBText*-Komponenten einfügen. Deren Eigenschaft *DataField* muss zur Laufzeit zugewiesen werden, da ja zur Entwurfszeit noch nicht feststeht, wie die Spaltennamen lauten werden. Wird eine andere Tabelle ausgewählt, dann werden sich mit 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 Abfrage verständlicherweise geschlossen werden.

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 der folgenden Abbildung wird der Schlüssel *Key1* über die Felder *Nachname*, *Straße* und *PLZ* definiert:



Die zweite Reihe der ComboBoxen gibt an, mit welcher Funktion (siehe Kapitel 6.5.1) der Such-String 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.

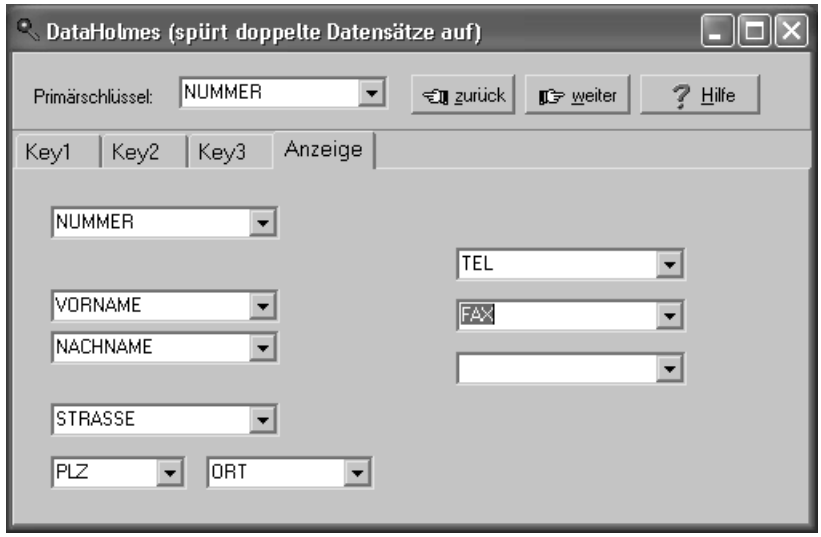
```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  i: integer;
begin
  Notebook1.ActivePage := 'Definieren';
  ComboBox3.Items.Clear;
  ComboBox5.Items.Clear;
    ...
  ComboBox42.Items.Clear;
  for i := 0 to Table1.FieldCount - 1 do
  begin
    ComboBox3.Items.Add(Table1.Fields[i].FieldName);
    ComboBox5.Items.Add(Table1.Fields[i].FieldName);
      ...
    ComboBox42.Items.Add(Table1.Fields[i].FieldName);
  end; { for i := 0 to Table1.FieldCount - 1 do}
end; {procedure TForm1.BitBtn1Click}
```

Wird die *Notebook*-Seite *Definieren* aufgerufen, dann muss vielen ComboBoxen die Liste der vorhandenen Feldnamen zugewiesen werden – die Prozedur wurde an dieser Stelle etwas gekürzt. Weitere Anweisungen sind an dieser Stelle nicht notwendig.

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 ComboBoxen geschehen.

Die folgende Abbildung 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.

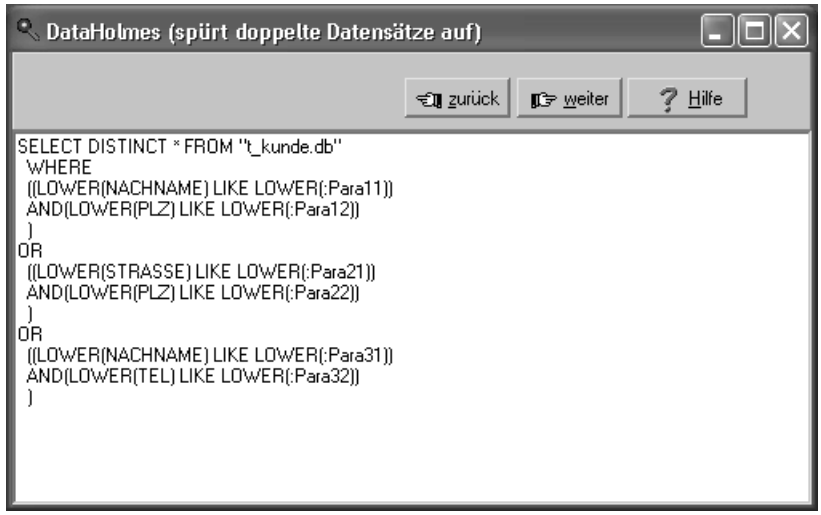
Generierung der SQL-Anweisung

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 des Programms. Außerdem wird ihm auf diese Weise schnell auffallen, wenn alle drei CheckBoxen nicht gesetzt sind und somit keiner der drei Schlüssel aktiv ist.

```
procedure TForm1.BitBtn4Click(Sender: TObject);
begin
  Notebook1.ActivePage := 'SQL';
  Memo1.Clear;
  with Memo1.Lines do
  begin
    Add('SELECT DISTINCT * FROM "'
      + LowerCase(ComboBox2.Text) + '"');
    Add('  WHERE');
```

```
    if CheckBox1.Checked = true then
    begin
      Add('  ((LOWER(' + ComboBox3.text + ') LIKE LOWER(:Para11))');
      if ComboBox5.Text <> '' then
        Add('  AND(LOWER(' + ComboBox5.text
          + ') LIKE LOWER(:Para12))');
      if ComboBox7.Text <> '' then
        Add('  AND(LOWER(' + ComboBox7.text
          + ') LIKE LOWER(:Para13))');
        ...
      Add('  )');
    end; {if CheckBox1.Checked = true then}

    if CheckBox2.Checked = true then
    begin
      if CheckBox1.Checked = true then Add('OR');
      Add('  ((LOWER(' + ComboBox14.text
          + ') LIKE LOWER(:Para21))');
      if ComboBox16.Text <> '' then
        Add('  AND(LOWER(' + ComboBox16.text
          + ') LIKE LOWER(:Para22))');
        ...
      Add('  )');
    end; {if CheckBox1.Checked = true then}

    if CheckBox3.Checked = true then
    begin
      if (CheckBox1.Checked = true)
        or (CheckBox2.Checked = true) then Add('OR');
      Add('  ((LOWER(' + ComboBox24.text
        + ') LIKE LOWER(:Para31))');
      if ComboBox26.Text <> '' then
        Add('  AND(LOWER(' + ComboBox26.text
          + ') LIKE LOWER(:Para32))');
        ...
      Add('  )');
    end; {if CheckBox1.Checked = true then}
  end; {with Memo1.Lines do}
```

Wie die folgende Abbildung 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.

```
  if ComboBox34.Text = '' then DBText1.Visible := false else
  begin
    DBText1.Visible := true;
    DBText1.DataField := ComboBox34.Text;
  end;
    ...
  if ComboBox42.Text = '' then DBText9.Visible := false else
  begin
    DBText9.Visible := true;
    DBText9.DataField := ComboBox42.Text;
  end;
end; {procedure 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.

Suchen nach doppelten Adressen

Bei der Suche nach doppelten Adressen werden nacheinander die Felder aller Adressen in die gewünschten Such-Strings aufbereitet und als Parameter der SQL-Anweisung zugewiesen.

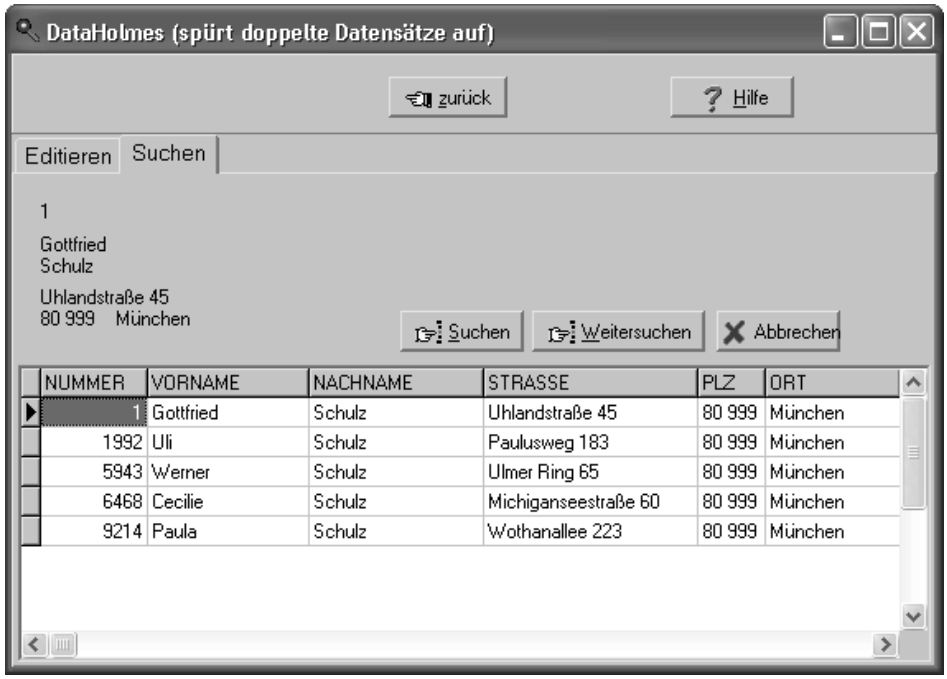
```
procedure TForm1.Button2Click(Sender: TObject);
var
  s: string;
  i: integer;
begin
  abbruch := false;
  i := 0;
  Label1.Visible := true;
  DataSource2.Enabled := false;
  DataSource1.Enabled := false;
  Screen.Cursor := crHourGlass;
```

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.

```
  with Table1 do
  begin
    while not EOF do
    begin
      if erster = false then Next else erster := false;
      i := i + 1;
      if i mod 10  = 0 then
      begin
        Label1.Caption := FieldByName(ComboBox13.Text).AsString;
      end;
      Application.ProcessMessages;
      if abbruch = true then
      begin
        DataSource2.Enabled := true;
        DataSource1.Enabled := true;
        Label1.Visible := false;
        Screen.Cursor := crDefault;
        exit;
      end;
```

Mit einer *while*-Schleife werden alle Datensätze von *Table1* durchlaufen und daraufhin überprüft, ob es Adressen mit ähnlichem Inhalt gibt. Die *if erster...*-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 zugrunde liegt, 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 werden. Außerdem soll die Möglichkeit vorgesehen werden, durch Setzen der Variablen *abbruch* die Schleife zu verlassen.

```
Query1.Close;
if CheckBox1.Checked = true then
begin
  s := Table1.FieldByName(ComboBox3.Text).AsString;
  case ComboBox4.ItemIndex of
    0: s := likeqb(s);
    1: s := likeqs(s);
    2: s := likeqh(s);
    3: s := likeqo(s);
    4: s := likeqp(s);
    5: s := likeqz(s);
  end; {case}
  Query1.ParamByName('Para11').AsString := s;

  if ComboBox5.Text <> '' then
  begin
    s := Table1.FieldByName(ComboBox5.Text).AsString;
    case ComboBox6.ItemIndex of
      0: s := likeqb(s);
      1: s := likeqs(s);
      2: s := likeqh(s);
      3: s := likeqo(s);
      4: s := likeqp(s);
      5: s := likeqz(s);
    end; {case}
    Query1.ParamByName('Para12').AsString := s;
  end; {if ComboBox5.Text <> '' then}
....
```

Nun werden die Feldinhalte in den String *s* eingelesen. Je nachdem, 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 then
begin
  DataSource2.Enabled := true;
  DataSource1.Enabled := true;
  Label1.Visible := false;
```

```
          Screen.Cursor := crDefault;
          exit;
        end;
      end; {while not EOF do}
      DataSource2.Enabled := true;
      DataSource1.Enabled := true;
      Label1.Visible := false;
      Screen.Cursor := crDefault;
    end; {with Table1 do}
end; {procedure TForm1.Button2Click}
```

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.

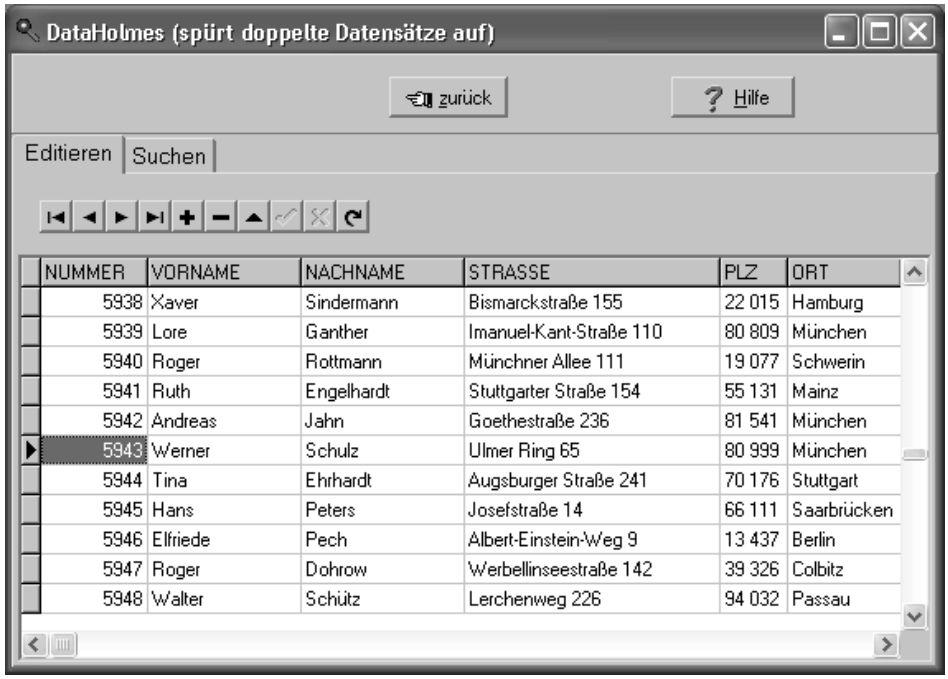
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.First;
  erster := true;
  Button2Click(Sender);
end;
```

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.

```
procedure TForm1.DBGrid2DblClick(Sender: TObject);
begin
  Table2.FindKey([Query1.FieldByName(ComboBox13.Text).AsString]);
  TabbedNotebook1.ActivePage := 'Editieren';
end;
```

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.



NUMMER	VORNAME	NACHNAME	STRASSE	PLZ	ORT
5938	Xaver	Sindermann	Bismarckstraße 155	22 015	Hamburg
5939	Lore	Ganther	Imanuel-Kant-Straße 110	80 809	München
5940	Roger	Rottmann	Münchner Allee 111	19 077	Schwerin
5941	Ruth	Engelhardt	Stuttgarter Straße 154	55 131	Mainz
5942	Andreas	Jahn	Goethestraße 236	81 541	München
5943	Werner	Schulz	Ulmer Ring 65	80 999	München
5944	Tina	Ehrhardt	Augsburger Straße 241	70 176	Stuttgart
5945	Hans	Peters	Josefstraße 14	66 111	Saarbrücken
5946	Elfriede	Pech	Albert-Einstein-Weg 9	13 437	Berlin
5947	Roger	Dohrow	Werbellinseestraße 142	39 326	Colbitz
5948	Walter	Schütz	Lerchenweg 226	94 032	Passau

7 MyBase

Die Komponente *TClientDataSet* wird zunehmend zum »Dreh- und Angelpunkt« der Datenbankprogrammierung unter Delphi:

- Mit *TClientDataSet* lassen sich hoch effiziente speicherbasierte Stand-Alone-Datenbankanwendungen erstellen.
- Als Einheit mit Datenzugriffskomponente und Provider gibt es die Komponenten *TBDEClientDataSet*, *TIBClientDataSet* und *TSQLClientDataSet*.
- Bei mehrschichtigen Datenbankanwendungen kommt man an *TClientDataSet* erst recht nicht vorbei.

Zunehmend interessanter wird die Möglichkeit, Einplatz-Datenbankanwendungen ausschließlich mit *TClientDataSet* zu erstellen. Die Ausstattung heutiger Rechner mit Arbeitsspeicher erlaubt es vielfach, den Datenbestand einer Anwendung komplett im Speicher zu halten – ein solches System ist natürlich sehr performant.

Auch die Features von *MyBase* – so der »Marketing-Name« dieser Technologie – lassen wenige Wünsche offen: Filtern der Datensätze (auch mit dem LIKE-Operator), sortieren, Nachschlagespalten, Master-Detail-Verknüpfung, gewartete Aggregate.

7.1 Eine Tabelle erstellen

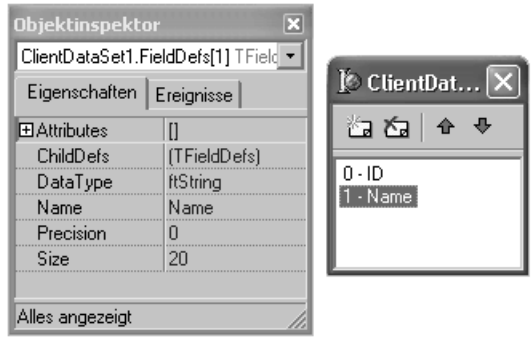
Solange *TClientDataSet* über einen Provider mit einer anderen Datenmenge verbunden ist, stellt sich diese Frage nicht. Für ein Einplatz-System müssen die Tabellen jedoch erst einmal erstellt werden. Dazu gibt es prinzipiell drei Möglichkeiten:

- Sie können ein externes Programm verwenden, welches die Tabellen erstellt und in einer Datei speichert. Ein solches Programm ist in Delphi (derzeit) nicht dabei, wir werden aber eine solche »TClientDataSet-Datenbankoberfläche« nachher noch erstellen.
- Eine Tabelle kann per Quelltext zur Laufzeit erstellt werden.
- Und mit der Eigenschaft *FieldDefs* und dem Menüpunkt DataSet erstellen aus dem Kontextmenü kann man eine Tabelle auch zur Entwurfszeit erstellen.

7.1.1 Eine Tabelle zur Entwurfszeit erstellen

Um eine Tabelle zur Entwurfszeit zu erstellen, gehen Sie wie folgt vor:

- Öffnen Sie den Eigenschaftseditor der Eigenschaft *FieldDefs*.
- Fügen Sie eine neue Felddefinition hinzu. Deren Daten können Sie dann mit dem Objektinspektor bearbeiten.



- Zunächst sollten Sie für die Spalte einen »sprechenden« Namen vergeben.
- Wählen Sie dann einen passenden *DataType*. Je nachdem, was Sie gewählt haben, müssen Sie auch noch *Size* vielleicht auch *Precision* setzen.
- Mit *Attributes* können Sie unter anderem festlegen, ob für eine Spalte eine Eingabe erforderlich ist (*faRequired*).
- Bei einem Array- oder ADT-Feld gibt es untergeordnete Spalten, die mit *ChildDefs* definiert werden.
- Mit DATASET ERSTELLEN aus dem Kontextmenü von *TClientDataSet* kann nun die Tabelle erstellt werden.
- Anschließend kann die Tabelle in eine Datei gespeichert werden – entweder im binären Format oder in einem der beiden XML-Formate.

7.1.2 Eine Tabelle zur Laufzeit erstellen

Um eine Tabelle zur Laufzeit zu erstellen, muss auch hier mit der Eigenschaft *FieldDefs* eine Tabellendefinition erstellt werden. Anschließend wird *CreateDataSet* aufgerufen.

```
with ClientDataSet1 do
begin
  Active := false;
```

```
  with FieldDefs do
  begin
    Clear;
    Add('ID', ftAutoInc, 0, false);
    Add('Name', ftString, 20, true);
  end;
  CreateDataSet;
  Open;
end; {with ClientDataSet1 do}
```

7.2 Daten speichern

An die Komponente *TClientDataSet* kann eine *TDataSource*-Instanz gehängt werden, mit den Datensteuerungskomponenten können dann Daten eingegeben, geändert oder gelöscht werden. Da diese Daten jedoch ausschließlich im Speicher gehalten werden, müssen sie vor Programmende auf die Festplatte. Dafür gibt es prinzipiell drei Möglichkeiten:

- Die Eigenschaft *FileName* wird auf einen Dateiname gesetzt, aus dieser Datei werden beim Erzeugen der *TClientDataSet*-Instanz – also in der Regel beim Programmstart – die Datensätze geladen und beim Freigeben der Komponente – meist beim Programmende – werden die Daten wieder dorthin gespeichert.

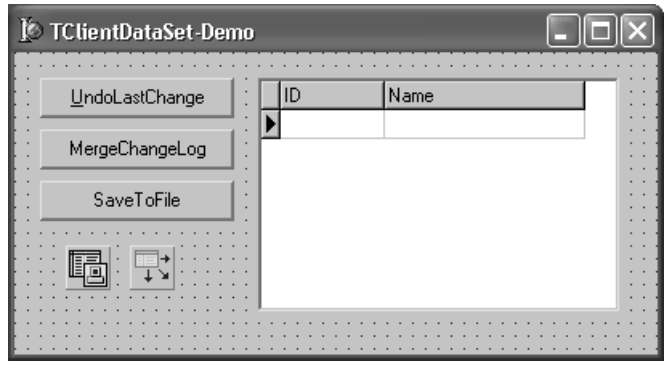
  Diese Möglichkeit ist die bequemste, macht aber in Zusammenarbeit mit Methoden wie *UndoLastChange* manchmal Probleme.

- Man kann manuell die Methode *SaveToFile* und *LoadFromFile* aufrufen.

- Mit *SaveToStream* und *LoadFromStream* kann man die Daten in einen Stream speichern und auch daraus laden. Allerdings muss man ein wenig basteln, um mehrere Tabellen über einen Stream in eine einzige Datei zu bekommen – wir werden uns auch das noch genauer ansehen.

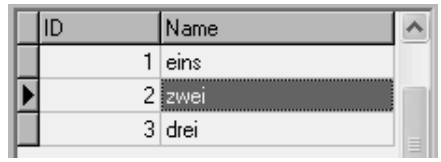
Wir wollen uns nun ansehen, was in der Datei gespeichert wird. Aus Gründen der Anschaulichkeit verwenden wir dafür das XML-Format. Im binären Format werden keine anderen Daten gespeichert, sie sind dort aber etwas »kompakter« und somit auch weniger gut lesbar.

Für die folgenden Versuche verwenden wir eine *TClientDataSet*-Komponente mit der Felddefinition aus Kapitel 7.1. Die Eigenschaft *FileName* ist entsprechend gesetzt und die drei Buttons rufen die Methode auf, die ihrer Beschriftung entspricht:



Die ersten Datensätze

Geben wir nun drei Datensätze ein:



Sobald wir das Programm schließen, werden die Daten auf die Festplatte geschrieben. Schauen wir uns diese Datei an:

```
  <?xml version="1.0" standalone="yes" ?>
- <DATAPACKET Version="2.0">
  - <METADATA>
    - <FIELDS>
      <FIELD attrname="ID" fieldtype="i4" readonly="true"
        SUBTYPE="Autoinc" />
      <FIELD attrname="Name" fieldtype="string" WIDTH="20" />
    </FIELDS>
    <PARAMS CHANGE_LOG="1 0 4 2 0 4 3 0 4" AUTOINCVALUE="4" />
  </METADATA>
  - <ROWDATA>
    <ROW RowState="4" ID="1" Name="eins" />
    <ROW RowState="4" ID="2" Name="zwei" />
    <ROW RowState="4" ID="3" Name="drei" />
  </ROWDATA>
</DATAPACKET>
```

In der ersten Zeile finden wir den XML-Header, danach folgt das Datenpaket, das aufgeteilt ist in die Metadaten und in die Datensätze (*Rowdata*).

Bei den Metadaten finden wir zunächst einmal die Felddefinitionen. Das Feld ID ist ein selbstinkrementierendes, *i4* deutet an, dass es mit 32 Bit Breite gespeichert wird, und beschrieben werden kann dieses Feld auch nicht – das erledigt bei einem selbstinkrementierenden Feld die Datenbank. Des Weiteren haben wir ein 20 Zeichen langes String-Feld.

In PARAMS CHANGE_LOG finden wir das Änderungsprotokoll. Mit *0 4* wird ein neu eingefügter Datensatz abgeschlossen, man kann also daraus ersehen, dass die Datensätze in der Reihenfolge ihrer ID-Werte in die Tabelle geschrieben wurden – bei einem selbstinkrementierenden Feld auch nicht weiter verwunderlich. Des Weiteren finden wir hier den nächsten Wert für die selbstinkrementierende Spalte *ID*.

Anschließend finden wir die drei Datensätze. Alle haben als *RowState* den Wert vier (vergleichen Sie das mit PARAMS CHANGE_LOG).

Einen Datensatz ändern

Nun nehmen wir den dritten Datensatz und geben dort als *Name* zunächst den Wert *vier* ein, dann *fünf* und schließlich *sechs*.

ID	Name
1	eins
2	zwei
3	sechs

Schauen wir uns auch dies wieder in der XML-Datei an (da sich die Felddefinition nicht geändert hat, habe ich sie gleich weggelassen):

```
      <PARAMS CHANGE_LOG="1 0 4 2 0 4 3 0 4 4 3 8 5 4 8 6 5 8"
        AUTOINCVALUE="4" />
    </METADATA>
  - <ROWDATA>
    <ROW RowState="4" ID="1" Name="eins" />
    <ROW RowState="4" ID="2" Name="zwei" />
    <ROW RowState="5" ID="3" Name="drei" />
    <ROW RowState="13" ID="3" Name="vier" />
    <ROW RowState="13" ID="3" Name="fünf" />
    <ROW RowState="12" ID="3" Name="sechs" />
  </ROWDATA>
</DATAPACKET>
```

Schauen wir uns erst die Datensätze an: Für jede Änderung wurde hier ein neuer Datensatz angelegt. Der ursprüngliche Datensatz erhält den Wert 5 als *Rowstate*, die Änderungen, die ihrerseits wieder überschrieben wurden, den Wert 13, die aktuelle Fassung den Wert 12.

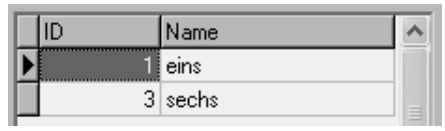
PARAMS CHANGE_LOG ist inzwischen ein wenig unübersichtlich, von daher wollen wir diese Reihe in die einzelnen Werte aufspalten:

```
1 0 4
2 0 4
3 0 4
4 3 8
5 4 8
6 5 8
```

Im Bereich der Änderungen: Zunächst ersetzt der vierte Datensatz den dritten, dann der fünfte den vierten, zuletzt der sechste den fünften.

Einen Datensatz löschen

Nun wollen wir den zweiten Datensatz löschen.



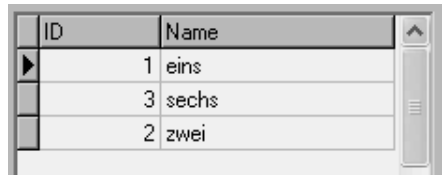
Hier wird nun der zweite Datensatz mit *RowState* 6 gekennzeichnet und drei weitere Zeichen werden an die Änderungsliste angehängt.

```
    <PARAMS CHANGE_LOG="1 0 4 2 0 4 3 0 4 4 3 8 5 4 8 6 5 8 2 0 2"
      AUTOINCVALUE="4" />
  </METADATA>
  - <ROWDATA>
    <ROW RowState="4" ID="1" Name="eins" />
    <ROW RowState="6" ID="2" Name="zwei" />
    <ROW RowState="5" ID="3" Name="drei" />
    <ROW RowState="13" ID="3" Name="vier" />
    <ROW RowState="13" ID="3" Name="fünf" />
    <ROW RowState="12" ID="3" Name="sechs" />
  </ROWDATA>
</DATAPACKET>
```

Änderungen zurücknehmen

(Im Folgenden wird vorausgesetzt, dass Sie die Anwendung neu geöffnet haben.)

Nun ist es nicht gänzlich ausgeschlossen, dass der Anwender eine Änderung etwas voreilig vorgenommen hat. Da hier alle Änderungen mitprotokolliert werden, ist es nicht weiter schwierig, diese zurückzunehmen; dazu muss lediglich *UndoLastChange* aufgerufen werden. Mittels des Parameters *FollowChange* kann man steuern, ob der Cursor auf wiederhergestellt Datensätze positioniert wird oder seine Position beibehält.



Beim erstmaligen Aufruf von *UndoLastChange* wird der gelöschte Datensatz wiederhergestellt, allerdings nicht an seiner ursprünglichen Position, sondern am Ende der Tabelle. Dies liegt daran, dass wir derzeit keinen Index verwenden. Würden wie beispielsweise nach der Spalte ID sortieren, dann wäre der neue Datensatz selbstverständlich an seiner vorherigen Position.

Durch wiederholtes Aufrufen von *UndoLastChange* können alle Änderungen zurückgenommen werden – am Schluss ist die Tabelle wieder leer. Wenn Sie jetzt die Anwendung schließen und neu öffnen, haben Sie den Zustand vor *UndoLastChange* und können nun wieder anfangen, die Änderungen zurückzunehmen.

Tun Sie dies und fügen in die leere Tabelle einen neuen Datensatz ein, die Spalte *Name* erhält den Wert *test*. Zum einen fällt auf, dass in der Spalte ID nun wieder die Zahl 1 steht. Wenn Sie jetzt die Anwendung schließen und erneut öffnen, dann sehen Sie lediglich den *test*-Datensatz. Mit dem Aufruf von *UndoLastChange* können Sie diesen wieder entfernen, unsere vorherigen Datensätze können Sie damit jedoch nicht mehr wiederherstellen.

Warum wurde jedoch vorhin die Rücknahme der Änderungen nicht dauerhaft gespeichert? Werden mittels der Eigenschaft *FileName* die Datensätze gespeichert, wird intern nur dann *SaveToFile* aufgerufen, wenn eine Änderung erfolgt ist. Der Aufruf von *UndoLastChange* wird jedoch nicht als Änderung registriert. Durch das Einfügen des *test*-Datensatzes im zweiten Versuch haben wir dann eine Änderung vorgenommen, die tatsächlich registriert wurde, und so haben die Rücknahmen der Änderungen auch ihren Weg auf die Festplatte gefunden.

Alternativ hätte man auch *SaveToFile* aufrufen können:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  with ClientDataSet1
    do SaveToFile(FileName);
end;
```

Das Protokoll leeren

Im Laufe der Zeit wird das Änderungsprotokoll immer länger – verständlicherweise sorgt das weder für Performance noch für geringen Speicherplatzbedarf. Mit *MergeChangeLog* kann das Änderungsprotokoll geleert werden:

```
  <?xml version="1.0" standalone="yes" ?>
- <DATAPACKET Version="2.0">
  - <METADATA>
    - <FIELDS>
      <FIELD attrname="ID" fieldtype="i4" readonly="true"
        SUBTYPE="Autoinc" />
      <FIELD attrname="Name" fieldtype="string" WIDTH="20" />
    </FIELDS>
    <PARAMS AUTOINCVALUE="3" />
  </METADATA>
  - <ROWDATA>
    <ROW ID="1" Name="test" />
    <ROW ID="2" Name="sechs" />
  </ROWDATA>
</DATAPACKET
```

Zum einen fehlt nun die »Spalte« *RowStates*, zum anderen das gesamte Änderungsprotokoll. Verständlicherweise werden Sie nun mit einem Aufruf von *UndoLastChange* nichts mehr erreichen.

Ebenso wie *UndoLastChange* gilt *MergeChangeLog* nicht als Änderung – Sie müssen gegebenenfalls *SaveToFile* manuell aufrufen.

Fügen Sie nun einen weiteren Datensatz ein, wird wieder ein Änderungsprotokoll begonnen, ebenso findet sich dann die Spalte *RowState* wieder. Dasselbe würde passieren, wenn Sie Datensätze ändern oder löschen.

```
    <PARAMS CHANGE_LOG="3 0 4" AUTOINCVALUE="4" />
  </METADATA>
  - <ROWDATA>
    <ROW ID="1" Name="test" />
```

```
    <ROW ID="2" Name="sechs" />
    <ROW RowState="4" ID="3" Name="sieben" />
  </ROWDATA>
</DATAPACKET>
```

Auf das Änderungsprotokoll verzichten

Die Komponente *TClientDataSet* wurde für mehrschichtige Datenbankanwendungen konzipiert. Sollen Änderungen zu einem späteren Zeitpunkt auf einen Datenbankserver geschrieben werden, dann ist es essenziell, alle Änderungen präzise mitzuprotokollieren.

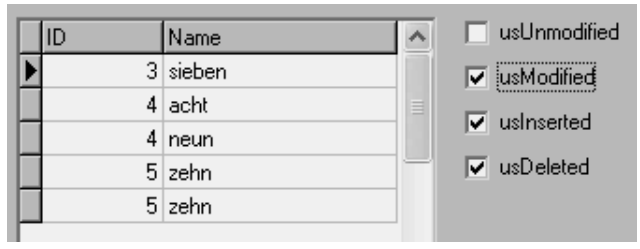
Bei einschichtigen Datenbankanwendungen ist dieses Protokoll dagegen Luxus – es ermöglicht die Rücknahme der Änderungen mit *UndoLastChange*, kostet aber auch Speicherplatz und Performance. Dies lässt sich mit dem gelegentlichen Aufruf von *MergeChangeLog* reduzieren, man kann aber auch das Änderungsprotokoll ganz abschalten, indem man *LogChanges* auf *false* setzt.

Änderungen an einem Datensatz zurücknehmen

Bisweilen möchte man alle Änderungen an einem einzelnen Datensatz zurücknehmen. Wenn zwischen den Änderungen am betreffenden Datensatz auch andere Datensätze geändert wurden, dann kann dafür nicht *UndoLastChange* verwendet werden. Mit *RevertRecord* lässt sich ein Datensatz in seinen ursprünglichen Zustand zurückversetzen.

Die Status-Filter

Mit Hilfe der Eigenschaft *StatusFilter* besteht die Möglichkeit, sich nur bestimmte Datensätze anzeigen zu lassen. *StatusFilter* ist vom Typ *TUpdateStatus*, der die Elemente *usUnmodified*, *usModified*, *usInserted* und *usDeleted* beinhaltet.



Beim Einsatz von *UpdateFilter* ist damit zu rechnen, dass einzelne Datensätze mehrfach in der Ansicht erscheinen.

Einen SavePoint verwenden

Die Eigenschaft *SavePoint* erlaubt es, einen beliebigen Zustand der Datenmenge zwischenzuspeichern. Dadurch wird der Inhalt von *SavePoint* in eine Integer-Variable geschrieben:

```
FSavePoint := ClientDataSet1.SavePoint;
```

Sollen mehrere Zwischenstände »archiviert« werden, dann können dafür verschiedene Integer-Variablen verwendet werden. Um den gespeicherten Zustand wiederherzustellen, wird einfach der Inhalt der Variablen wieder der Eigenschaft *SavePoint* zugewiesen:

```
ClientDataSet1.SavePoint := FSavePoint;
```

Intern wird auch hier mit *UndoLastChange* gearbeitet – Änderungen, die mit *UndoLastChange* zurückgenommen wurden, können nicht wiederhergestellt werden, und den Einsatz von *MergeChangeLog* sollten Sie sich auch verkneifen.

7.3 Sortieren von Datenmengen

Zum Sortieren von Client-Datenmengen gibt es drei Möglichkeiten. Bei keiner dieser Möglichkeiten werden die Indexdaten mit auf die Festplatte gespeichert, sie werden immer zur Laufzeit neu aufgebaut.

Am einfachsten geht die Sortierung mit *IndexFieldNames*: Weisen Sie dieser Eigenschaft einfach die Spalte oder – durch Semikola getrennt – die Spalten zu, nach denen sortiert werden soll:

```
ClientDataSet1.IndexFieldNames := 'Nummer;Wert';
```

Dabei sind jedoch nur aufsteigende Indizes möglich.

Sind absteigende Indizes gewünscht oder soll die Groß- und Kleinschreibung ignoriert werden, dann muss mit *AddIndex* oder *IndexDefs* gearbeitet werden.

```
with ClientDataSet1.IndexDefs.AddIndexDef do
begin
  Name := 'ixName';
  Fields := 'Nachname;Vorname;ID';
  DescFields := 'ID';
  CaseInsFields := 'Nachname;Vorname';
  Options := [ixDescending, ixCaseInsensitive]
end;
```

Beide Wege bieten im Prinzip die gleichen Möglichkeiten, wobei bei der Methode *AddIndex* die Werte als Parameter übergeben und bei der Objekteigenschaft *IndexDefs* an Eigenschaften zugewiesen werden. Vor dem Erstellen der Datenmenge muss mit *IndexDefs* gearbeitet werden, danach mit *AddIndex*.

In *Fields* werden die Spalten, die den Index bilden, der Reihe nach aufgeführt. In *DescFields* werden nochmals diejenigen Spalten aufgeführt, die absteigend zu sortieren sind, in *CaseInsFields* diejenigen Spalten, bei denen die Groß- und Kleinschreibung ignoriert werden soll. Als weitere Optionen lassen sich angeben, dass der Index ein Schlüssel sein soll oder nicht gewartet werden soll.

Mit *IndexName* kann dann aus der Menge der definierten Indizes gewählt werden. Achten Sie dabei bitte auf die Groß- und Kleinschreibung des Indexnamens.

7.4 Suchen und Filtern

Bei *TClientDataSet* können Sie zum Suchen und Filtern alles das verwenden, was wir für *TTable* in Kapitel 5.4 besprochen haben.

Natürlich gibt es hier auch die Eigenschaft *Filter*, schließlich wird diese von *TDataSet* geerbt. Der Funktionsumfang geht aber deutlich über das hinaus, was *TDataSet* implementiert und erreicht das Niveau einfacher SQL-Systeme:

- Vergleichsoperatoren:

=	gleich
>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
<>	ungleich
IS NULL	kein Wert
IS NOT NULL	irgendein Wert

  Das Schlüsselwort BLANK, das in der Online-Hilfe vorgestellt wird, führt zumindest bei mir auf dem System zu einer Fehlermeldung. Die folgende Anweisung würde auf diejenigen Datensätze filtern, die in der Spalte Text einen Wert haben:

  ```
  Text IS NOT NULL
  ```

- Logische Operatoren

AND	Und-Verknüpfung
OR	Oder-Verknüpfung
NOT	Negierung

  Mehrere Bedingungen dürfen mittels logischer Operatoren verknüpft werden. Der Einsatz von Klammern ist dabei seltener erforderlich als in Delphi, im folgenden Beispiel sind sie überflüssig.

  ```
  (Nummer > 10) OR (Wert > 50)
  ```

- Arithmetische Operatoren

+	Addition
-	Subtraktion
*	Multiplikation
/	Division

- Groß- und Kleinschreibung

UPPER	Umwandlung in Großbuchstaben
LOWER	Umwandlung in Kleinbuchstaben

- *Substring* gibt einen Teil eines Strings zurück. Diese Funktion wird mit zwei oder drei Parametern aufgerufen

  SUBSTRING (text, beginn) gibt alle Zeichen ab *beginn* zurück

  SUBSTRING (text, beginn, anzahl) gibt *anzahl* Zeichen ab Beginn zurück.

- *Trim* enfernt Zeichen am Anfang und/oder am Ende des Strings. Werden diese Funktionen ohne zweites Argument verwendet, dann werden Leerzeichen entfernt.

TRIM(text, zeichen)	entfernt an beiden Seiten des Strings
TRIMLEFT(text, zeichen)	entfernt am Anfang des Strings
TRIMRIGHT(text, zeichen)	entfernt am Ende des Strings

- Datums- und Zeitfunktionen

DATE	Datumsanteil
YEAR	Jahr
MONTH	Monat
DAY	Tag

TIME	Zeitanteil
HOUR	Stunde
MINUTE	Minute
SECOND	Sekunde
GETDATE	Jetzt (aktuelles Datum und aktuelle Uhrzeit)

Das folgende Beispiel ermittelt alle Datensätze, die in den letzten 14 Tagen geändert wurden (Das Feld *Changed* muss natürlich in der Datenmenge und entsprechend gefüllt sein):

```
GETDATE - Changed > 14
```

- Weitere Operatoren

  LIKE — arbeitet wie der entsprechende SQL-Operator. Das Jokerzeichen _ steht für ein einzelnes Zeichen, % für beliebig viele Zeichen. Die folgende Filteranweisung ermittelt alle Zeilen, deren Feld *Text* als zweites Zeichen ein kleingeschriebenes *e* enthält.

```
Text LIKE '_e%'
```

  Um Groß- und Kleinschreibung zu ignorieren, wird der Spalteninhalt in Groß- oder Kleinbuchstaben umgewandelt und der Vergleichs-String entsprechend angepasst (siehe auch *FilterOption*)

```
UPPER(text) LIKE 'DELTA'
```

  IN — Wert in der Datenmenge, erspart umfangreiche OR-Verknüpfungen

```
Nummer IN (1, 2, 5, 7)
```

  * — Jokerzeichen in Kombination mit dem Gleichheitszeichen: Die folgende Anweisung würde alle Text-Felder ermitteln, die mit einem großen *W* beginnen.

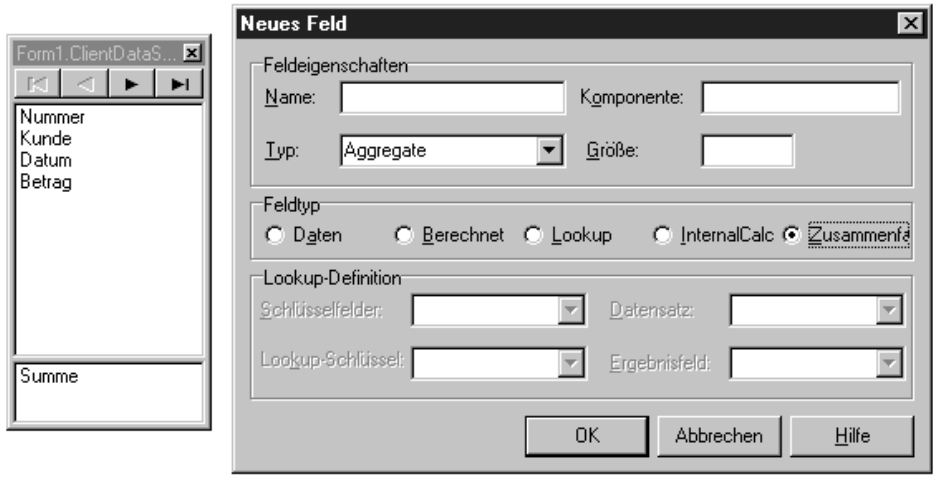
```
Text = 'W*'
```

7.5 Master-Detail-Verknüpfungen

Auch bei Master-Detail-Verknüpfungen kann auf *TTable* verwiesen werden, genauer gesagt auf Kapitel 5.2. Im Gegensatz zu dBase- und Paradox-Tabellen wird hier für die Detailfelder kein Index benötigt.

7.6 Gewartete Aggregate

Mit Hilfe von Aggregatfunktionen können Sie statistische Funktionen wie Summe oder Durchschnitt einer Spalte berechnen. Gehen Sie zum Erstellen gewarteter Aggregate wie folgt vor:



Erstellen Sie dann persistente *TField*-Komponenten für alle Datenbankfelder und rufen dann *Neues Feld* aus dem Kontextmenü auf. Es erscheint ein Dialogfenster entsprechend oben stehender Abbildung. Hier wählen Sie in der Rubrik *Feldtyp* den Eintrag *Zusammenf.* aus. Als Typ der Feldeigenschaft wird nun automatisch *Aggregate* eingestellt. Geben Sie jetzt einen Namen ein und beenden Sie den Dialog mit OK.

Für diese Art von persistenten *TField*-Instanzen wird eine eigene Rubrik eingerichtet; diese Felder werden auch nicht in Datengittern angezeigt.

Mit dem Objektinspektor setzen Sie nun die Eigenschaft *Expression* auf einen geeigneten Wert. Um beispielsweise die Summe der Spalte *Betrag* zu bilden, wäre die Eigenschaft *Expression* auf *SUM(betrag)* zu setzen. Die Syntax lehnt sich auch hier dem SQL-Standard an. Diese Summe kann man dann beispielsweise mit einer *TDBText*-Komponente anzeigen. Vergessen Sie nicht, die *TClientDataSet*-Eigenschaft *AggregatesActive* auf *true* zu setzen.

7.7 Referenz TClientDataSet

TClientDataSet ist von der in Kapitel 3.1 beschriebenen Komponente *TDataSet* abgeleitet und erbt somit alles, was dort beschrieben ist.

Verbindung zur Datenbank oder zur Datenbasis

- ProviderName (Eigenschaft, veröffentlicht)

  ```
  property ProviderName: string;
  ```

  Eine *TClientDataSet*-Instanz kann Daten von einem Provider beziehen, der sie wiederum aus einer anderen Datenmenge bezieht. So kann beispielsweise eine Tabelle oder eine Abfrage von einem Datenbank-Server auf den Client heruntergeladen werden.

- FileName (Eigenschaft, veröffentlicht)

  ```
  property FileName: string;
  ```

  Wird *FileName* ein Wert zugewiesen, dann lädt *TClientDataSet* die Daten bei Programmstart aus dieser Datei und speichert sie beim Beenden des Programms dort ab – die bisherige Datei wird dabei ohne Rückfrage überschrieben.

- SaveToFile, SaveToStream (Methoden)

  ```
  procedure SaveToFile(const FileName: string);
  procedure SaveToStream(Stream: TStream);
  ```

  Die Daten können in eine Datei oder in einen Stream geschrieben werden. Letzteres wäre besonders dann nützlich, wenn man bei vielen *TClientDataSet*-Instanzen nicht ebenso viele Dateien anlegen möchte.

  Der Konjunktiv hat an dieser Stelle durchaus seine Berechtigung: *TClientDataSet* speichert die Daten ohne Längenangabe in den Stream, so dass beim Auslesen bis zum Stream-Ende gelesen wird. Dieses Verhalten vermeidet die Komponente *TBClientDataSet*, welche in Kapitel 7.8 beschrieben wird.

- LoadFromFile, LoadFromStream (Methoden)

  ```
  procedure LoadFromFile(const FileName: string);
  procedure LoadFromStream(Stream: TStream);
  ```

  Mit Hilfe dieser Methoden werden die Daten aus Dateien oder Streams geladen.

- CreateDataSet (Methode)

```
procedure CreateDataSet;
```

Mit *CreateDataSet* wird eine neue Datenmenge erzeugt. Vorher muss mit *FieldDefs* die Felddefinition erstellt werden.

```
with ClientDataSet1 do
begin
  Active := false;
  with FieldDefs do
  begin
    Clear;
    Add('Nummer', ftAutoInc, 0, false);
    Add('Vornamen', ftString, 30, true);
    Add('Nachnamen', ftString, 30, true);
  end;
  with IndexDefs do
  begin
    Clear;
    Add('Primary', 'Nummer', [ixPrimary, ixUnique]);
  end;
  CreateDataSet;
  Open;
end; {with ClientDataSet1 do}
```

Beachten Sie, dass Indizes nur dann dauerhaft zur Verfügung stehen, wenn Sie mit *IndexDefs* vor dem Aufruf von *CreateDataSet* erzeugt worden sind. Indizes, die mit *AddIndex* erzeugt wurden, »verfallen« beim Freigeben der *TClientDataSet*-Instanz, normalerweise also beim Schließen der Anwendung.

- RemoteServer, ConnectionBroker (Eigenschaften, veröffentlicht)

```
property RemoteServer: TCustomRemoteServer;
property ConnectionBroker: TConnectionBroker;
```

Befindet sich der Provider in einem anderen Programm, dann muss die Verbindung mit einem *RemoteServer* hergestellt werden. Alternativ dazu kann mit einem *ConnectionBroker* die Verbindung von mehreren *TClientDataSet*-Instanzen zentral verwaltet werden.

- ReadOnly (Eigenschaft, veröffentlicht)

```
property ReadOnly: Boolean;
```

Wird *ReadOnly* auf *true* gesetzt, dann kann die *TClientDataSet*-Instanz keine Änderungen am Datenbestand vornehmen.

- CloneCursor (Methode) CloneSource (Eigenschaft, öffentlich, nur Lesen)

```
procedure CloneCursor(Source :TCustomClientDataSet;
  Reset: Boolean; KeepSettings: Boolean = False);
property CloneSource: TCustomClientDataSet;
```

CloneCursor wird verwendet, um auf den Datenbestand eines anderen *ClientDataSets* zuzugreifen. Einen Zeiger auf diese Datenmenge kann man mit *CloneSource* erhalten.

Aktualisieren der Server-Daten oder der Datenbasis

- ApplyUpdates, CancelUpdates (Methoden)

```
function ApplyUpdates(MaxErrors: Integer); Integer;
procedure CancelUpdates;
```

Ist die Komponente *TClientDataSet* mit einem Server verbunden, dann aktualisiert sie dort nicht automatisch die Datenmenge. Damit dies geschieht, muss *ApplyUpdates* aufgerufen werden. Dieses Verhalten ist beispielsweise mit *TTable* vergleichbar, wenn dort die Eigenschaft *CachedUpdates* auf *true* gesetzt wird. Dementsprechend werden mit *CancelUpdates* alle Änderungen verworfen.

- RevertRecord, UndoLastChange (Methoden)

```
procedure RevertRecord;
function UndoLastChange(FollowChange: Boolean): Boolean;
```

Mit *UndoLastChange* wird die letzte Änderung widerrufen. Wird dabei der Parameter *FollowChange* auf *true* gesetzt, dann wird der Datenzeiger auf den betreffenden Datensatz gesetzt. Um alle Änderungen am aktuellen Datensatz rückgängig zu machen, wird *RevertRecord* aufgerufen.

- OnReconcileError (Ereignis)

```
property OnReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
```

Tritt beim Aktualisieren der Daten auf dem Server ein Fehler auf, dann wird das Ereignis *OnReconcileError* aufgerufen. Mit Hilfe des Variablenparameters *Action* kann die Reaktion auf den Fehler bestimmt werden. Dieses Ereignis ähnelt dem Ereignis *OnUpdateError*.

- LogChanges (Eigenschaft, öffentlich), MergeChangeLog (Methode)

```
property LogChanges: Boolean default true;
procedure MergeChangeLog;
```

Hat *LogChanges* den Wert *true*, dann werden die Änderungen nicht in die Datenbasis geschrieben, sondern in ein Änderungsprotokoll (»Delta«). Aus diesem Delta und der ursprünglichen Datenbasis berechnet dann *TClientDataSet* die anzuzeigende Datenmenge. Dieses Delta erlaubt es dann auch, Daten auf dem Server zu aktualisieren.

Wird *TClientDataSet* ohne Verbindung mit einem Server betrieben, dann kostet dieses Delta nur Platz im Arbeitspeicher und auf der Festplatte. Um diese Änderungen in die Datenbasis einzutragen, kann man *MergeChangeLog* aufrufen oder man setzt gleich *LogChanges* auf *false*.

- CommandText (Eigenschaft, veröffentlicht), Execute (Methode)

```
property CommandText: string;
procedure Execute;
```

Ist *TClientDataSet* über einen Provider mit einer Komponente verbunden, die SQL-Anweisungen interpretieren kann (*TQuery*, *TStoredProc*...), dann kann mit *CommandText* eine solche SQL-Anweisung zugewiesen werden.

Mit *Execute* kann man Anweisungen ausführen, die keine Ergebnismenge zurückliefern (CREATE TABLE beispielsweise).

- FetchOnDemand, PacketRecords (Eigenschaften, veröffentlicht), GetNextPacket (Methode)

```
property FetchOnDemand: Boolean default true;
function GetNextPacket: Integer;
property PacketRecords: Integer;
```

Ist *TClientDataSet* mit einem Provider verbunden und hat *FetchOnDemand* den Wert *true*, dann holt sich die Komponente automatisch diejenigen Datensätze, die sie benötigt.

Anderenfalls müssen weitere Daten bei Bedarf manuell mit *GetNextPacket* angefordert werden. Wie viele Datensätze in einem »Packet« enthalten sind, wird mit *PacketRecords* spezifiziert.

Zugriff auf die Daten

- RecordCount (Eigenschaft, öffentlich, nur Lesen)

```
property RecordCount: Longint;
```

RecordCount liefert die Zahl der vorhandenen Datensätze.

- RecNo (Eigenschaft, öffentlich)

```
property RecNo: Integer;
```

Die Nummer des aktuellen Datensatzes kann man mit *RecNo* ermitteln (wobei die Zählung mit eins beginnt). Um den Datenzeiger auf einen bestimmten Datensatz zu setzen, kann man *RecNo* auch einen Wert zuweisen.

- EmptyDataSet (Methode)

```
procedure EmptyDataSet;
```

Löscht alle Datensätze.

- RevertRecord, UndoLastChange (Methode)

```
procedure RevertRecord;
function UndoLastChange(FollowChange: Boolean): Boolean;
```

Vorausgesetzt, die Änderungen befinden sich noch im Aktualisierungsspeicher, können mit *RevertRecord* alle Änderungen rückgängig gemacht werden.

Mit *UndoLastChange* wird die letzte Änderung rückgängig gemacht. Wird das Löschen eines Datensatzes rückgängig gemacht, dann wird der Datenzeiger auf diesen Datensatz positioniert, wenn *FollowChange* den Wert *true* hat.

Suchen und Filtern

- SetRangeStart, SetRangeEnd (Methoden)

```
procedure SetRangeStart;
procedure SetRangeEnd;
```

Mit den Methoden *SetRangeStart* und *SetRangeEnd* werden die oberen und unteren Grenzen eines Bereichs gesetzt. Die Werte für die Grenzen werden über die Eigenschaft *Fields* oder die Methode *FieldByName* gesetzt. Beachten Sie bitte, dass für das Setzen eines Bereichs ein Index für alle am Bereich beteiligten Spalten benötigt wird.

```
with ClientDataSet1 do
begin
  IndexFieldNames := 'Nummer';
  SetRangeStart;
  FieldByName('nummer').AsInteger := 100;
  SetRangeEnd;
  FieldByName('nummer').AsInteger := 200;
  ApplyRange;
end;
```

- ApplyRange, CancelRange (Methoden)

```
procedure ApplyRange;
procedure CancelRange;
```

  Diese Prozeduren dienen zum Anwenden und Aufheben eines Bereiches.

- KeyExclusive (Eigenschaft, öffentlich)

```
property KeyExclusive: Boolean;
```

  Mit Hilfe dieser Eigenschaft wird festgelegt, ob die Bereichsgrenzen mit zum Bereich gehören oder nicht.

- SetRange (Methode)

```
procedure SetRange(const StartValues, EndValues: array of const);
```

  Mit Hilfe von *SetRange* kann der Bereich mit Hilfe einer einzigen Anweisung gesetzt werden. Das bei *SetRangeStart* aufgeführte Beispiel könnte man also auch wie folgt realisieren:

```
with ClientDataSet1 do
begin
  IndexFieldNames := 'Nummer';
  SetRange([100], [200]);
end;
```

- EditRangeStart, EditRangeEnd (Methoden)

```
procedure EditRangeStart;
procedure EditRangeEnd;
```

  Mit diesen Methoden können die Grenzen bestehender Bereiche verändert werden.

- SetKey, EditKey (Methoden)

```
procedure SetKey;
procedure EditKey;
```

  Mit *SetKey* wird das Setzen, mit *EditKey* wird das Ändern des Suchschlüssels eingeleitet. Beachten Sie bitte, dass für die beteiligten Felder zwingend ein Index gesetzt werden muss.

```
with ClientDataSet1 do
begin
  IndexFieldNames := 'Nummer';
  SetKey;
  FieldByName('Nummer').AsInteger := 2;
  if not GotoKey
    then ShowMessage('Wert nicht gefunden');
end;
```

- GotoKey, GotoNearest (Methoden)

```
function GotoKey: Boolean;
procedure GotoNearest;
```

Die Funktion *GotoKey* versucht, einen Datensatz anhand des gesetzten Schlüssels zu finden. Gelingt ihr das, wird der Datenzeiger auf diesen Datensatz gesetzt und *true* als Funktionsergebnis zurückgegeben.

Die Prozedur *GotoNearest* positioniert den Datenzeiger auf den Datensatz, dessen Feldwerte dem Suchschlüssel am meisten gleichen.

Im Gegensatz zum Setzen von Bereichen wird bei *GotoKey* und *GotoNearest* der angezeigte Datenbestand nicht eingeschränkt, es wird lediglich der Datenzeiger auf den entsprechenden Wert gesetzt.

- FindKey, FindNearest (Methoden)

```
function FindKey(const KeyValues: array of const): Boolean;
procedure FindNearest(const KeyValues: array of const);
```

Diese Methoden fassen die Suche nach einem Datensatz zu einer einzigen Anweisung zusammen, die Schlüsselwerte werden als Parameter übergeben. Ein Beispiel dazu finden Sie bei der Referenz von *TTable*.

```
with ClientDataSet1 do
begin
  IndexFieldNames := 'Nummer';
  if not FindKey([2])
    then ShowMessage('Wert nicht gefunden');
end;
```

- GotoCurrent (Methode)

```
procedure GotoCurrent(Table: TClientDataSet);
```

Die Methode *GotoCurrent* setzt den Datenzeiger auf denselben Datensatz wie den Datenzeiger der als Parameter übergebenen Datenmenge. Auf diese Weise lassen sich sehr einfach zwei *TClientDataSet*-Instanzen synchronisieren.

Indizes

Viele Operationen bei *TClientDataSet* (Bereiche, Suche nach Schlüsseln) sind nur dann möglich, wenn die verwendeten Felder indiziert sind. Durch die Auswahl eines Indexes wird die Datenmenge entsprechend sortiert.

Prinzipiell gibt es beim Erstellen von Indizes zwei Möglichkeiten:

- Mit *IndexDefs* oder *AddIndex* wird ein permanenter Index erstellt.
- Mit *IndexFieldNames* wird ein temporärer Index erstellt.

Auch dann, wenn ein permanenter Index erstellt wurde, wird dieser nicht in der Datei gespeichert, sondern für die erste Verwendung erstellt. Das dauert beim ersten Mal ein wenig länger, dann ist aber der Index nahezu verzögerungslos einsetzbar. Bei *IndexFieldNames* dagegen wird der Index jedes Mal neu erstellt, der Aufbau des Indexes nimmt also jedes Mal Zeit in Anspruch.

Wir sollten jedoch nicht vergessen, dass wir hier von Zeiten im Millisekundenbereich sprechen – die Indizierung einer Tabelle von 10 000 Datensätzen hat bei mir auf dem Notebook (500 MHz) 20 bis 40 ms gedauert.

- IndexFieldNames, IndexFields (Eigenschaften veröffentlicht/öffentlich)

  ```
  property IndexFieldNames: string;
  property IndexFields[Index: Integer]: TField;
  ```

  Um einen temporären Index zu erstellen, werden einfach die beteiligten Spalten durch Semikola getrennt aufgeführt.

  ```
  ClientDataSet1.IndexFieldNames := 'Nummer;Wert';
  ```

  Die an einem Index beteiligten Spalten können mit *IndexFields* ermittelt werden.

- IndexName (Eigenschaft, veröffentlicht)

  ```
  property IndexName: string;
  ```

  Die Eigenschaft *IndexName* gibt den Namen des Indexes an, der verwendet werden soll. Dieser Index muss davor mit *AddIndex* oder *IndexDefs* erstellt worden sein.

- AddIndex, DeleteIndex (Methoden)

  ```
  procedure AddIndex(const Name, Fields: string;
    Options: TIndexOptions; const DescFields: string = '';
    const CaseInsFields: string = '';
    const GroupingLevel: Integer = 0 );
  procedure DeleteIndex(const Name: string);
  ```

```
TIndexOption = (ixPrimary, ixUnique, ixDescending,
  ixCaseInsensitive, ixExpression, ixNonMaintained);
```

Um zur Laufzeit neue Indizes zu erstellen, wird die Methode *AddIndex* verwendet. *Name* ist dabei der Name des Indexes, in *Fields* zählen Sie die beteiligten Felder durch Semikola getrennt auf.

Von den *Options* interessiert im Prinzip nur *ixUnique* – damit machen Sie aus dem Index einen Schlüssel: Ein Wert (oder bei zusammengesetzten Schlüsseln eine Wertekombination) kann damit nur ein einziges Mal vorkommen.

Mit *DescFields* wird angegeben, welche der Spalten abwärts sortiert werden sollen, mit *CaseInsFields*, bei welchen Spalten nicht auf Groß- und Kleinschreibung geachtet werden soll. *GroupingLevel* ist allenfalls bei gewarteten Aggregaten interessant.

Mit *DeleteIndex* können Sie Indizes löschen.

- GetIndexNames (Methode)

```
procedure GetIndexNames(List: TStrings);
```

Mit *GetIndexNames* können die Namen aller vorhandenen Indizes abgefragt und in eine Liste geschrieben werden.

- IndexDefs (Eigenschaft, veröffentlicht)

```
property IndexDefs: TIndexDefs;
```

Um Informationen über die Indizes zu erhalten, wird die Eigenschaft *IndexDefs* verwendet. Sie wird vor allem verwendet, um Indizes zur Entwurfszeit zu erstellen oder bevor mit *CreateDataSet* eine neue Datenmenge erzeugt wird.

```
with ClientDataSet1.IndexDefs do
begin
  Clear;
  Add('Primary', 'Nummer', [ixPrimary, ixUnique]);
end;
```

Master-Detail-Verknüpfungen

Eine Master-Detail-Verknüpfung erlaubt es, dass zu einem Hauptdatensatz mehrere Detaildatensätze zugeordnet werden. So sind einem Datensatz in der Tabelle *t_kunde* mehrere Datensätze in der Tabelle *t_bestellung* zugeordnet.

Der Aufbau von Master-Detail-Tabellen funktioniert genauso wie bei *TTable* und ist dort in Kapitel 5.2 beschrieben. Beachten Sie bitte, das im Gegensatz zu dBase- und Paradox-Tabellen keine Beschränkung der Detailspalten auf den aktuellen Index erfolgt.

- MasterSource (Eigenschaft, veröffentlicht)

  ```
  property MasterSource: TDataSource;
  ```

  Mit der Eigenschaft *MasterSource* wird angegeben, welche Datenquelle diese Datenmenge als Master-Tabelle steuert.

- MasterFields (Eigenschaft, veröffentlicht)

  ```
  property MasterFields: string;
  ```

  Der Eigenschaft *MasterFields* werden diejenigen Feldnamen zugewiesen, über die bei der Master-Tabelle die Verknüpfung gesteuert wird. Mehrere Feldnamen werden dabei mit Semikola gestrennt. In Anzahl und Feldart müssen diese Felder dem momentan verwendeten Schlüssel der Detailtabelle entsprechen.

  Zur Entwurfszeit steht Ihnen zum Setzen dieser Eigenschaft der Feldverbindungs-Designer zur Verfügung.

Gewartete Aggregate

Gewartete Aggregate sind in Kapitel 7.6 beschrieben.

- Aggregates (Eigenschaft, veröffentlicht)

  ```
  property Aggregates: TAggregates;
  ```

  Mit dieser Eigenschaft kann auf sämtliche gewartete Aggregate einer Client-Datenmenge zugegriffen werden. Dies ist beispielsweise erforderlich, um diese Aggregate einzeln zu aktivieren beziehungsweise zu deaktivieren.

  Normalerweise werden Sie über diese Eigenschaft keine neuen Aggregate erzeugen. Stattdessen wird Delphi hier selbst Einträge anlegen, wenn Sie in der dazugehörenden Datenmenge Aggregatspalten erzeugen.

- AggregatesActive (Eigenschaft, veröffentlicht)

  ```
  property AggregatesActive: Boolean;
  ```

  Mit dieser Eigenschaft können sämtliche gewarteten Aggregate einer Datenmenge deaktiviert beziehungsweise reaktiviert werden. Das Deaktivieren der gewarteten Aggregate vermindert die erforderliche Rechenleistung und sollte immer dann erfolgen, wenn die Aggregate ohnehin nicht angezeigt werden.

7.8 TBClientDataSet

Die Komponente *TBClientDataSet* hat leider einen kleinen Nachteil: Beim Speichern in einen Stream wird nicht die Anzahl der Bytes mitgespeichert, so dass beim Laden aus dem Stream alle Daten bis zum Stream-Ende ausgelesen werden. Wenn man mehrere Datenmengen in einen Stream speichern möchte, ist dieses Verhalten natürlich nicht zu gebrauchen.

Erfreulicherweise kann man ganz einfach etwas tun: Die Routinen *LoadFromStream* und *SaveFromStream* rufen intern die Routinen *ReadDataPacket* beziehungsweise *WriteDataPacket* auf. In diesen Methoden gibt es dann die Parameter *ReadSize* beziehungsweise *WriteSize* und diese müssen lediglich auf *true* gesetzt werden.

```
type
  TBClientDataSet = class(TClientDataSet)
  public
    procedure SaveToStream(Stream: TStream;
      Format: TDataPacketFormat = dfBinary);
    procedure LoadFromStream(Stream: TStream);
  end;

procedure TBClientDataSet.LoadFromStream(Stream: TStream);
begin
  Close;
  ReadDataPacket(Stream, true);
  Open;
end;

procedure TBClientDataSet.SaveToStream(Stream: TStream;
  Format: TDataPacketFormat);
begin
  WriteDataPacket(Stream, true, Format);
end;
```

Das nachfolgende Beispiel zeigt, wie man nun mehrere Datenmengen über einen Stream in eine einzelne Datei schreibt:

```
procedure Tfrm_data.SaveFile(FileName: string);
{Schreibt die Datenbanktabellen und den Text in eine Datei}
var
  stream: TFileStream;
  Format: TDataPacketFormat;
```

```
begin
  if ExtractFileExt(FileName) = 'xml'
    then Format := dfXML
    else Format := dfBinary;
  stream := TFileStream.Create(FileName, fmCreate);
  try
    cds_Kanal.SaveToStream(stream, Format);
    cds_gruppe.SaveToStream(stream, Format);
    cds_seite.SaveToStream(stream, Format);
    cds_fader.SaveToStream(stream, Format);
    frm_text.RichEdit1.Lines.SaveToStream(stream);
    FFileName := FileName;
  finally
    stream.Free;
  end;
end; {procedure Tfrm_data.SaveFile}
```

Und so bekommt man die Daten wieder zurück:

```
procedure Tfrm_data.OpenFile(FileName: string);
var
  stream: TFileStream;
  Format: TDataPacketFormat;
begin
  if ExtractFileExt(FileName) = 'xml'
    then Format := dfXML
    else Format := dfBinary;
  FWhileCreating := true;
  stream := TFileStream.Create(FileName, fmOpenRead);
  try
    cds_Kanal.LoadFromStream(stream);
    cds_gruppe.LoadFromStream(stream);
    cds_seite.LoadFromStream(stream);
    cds_fader.LoadFromStream(stream);
    frm_text.RichEdit1.Lines.LoadFromStream(stream);
    FFileName := FileName;
  finally
    FWhileCreating := false;
    stream.Free;
  end;
end; {procedure Tfrm_data.OpenFile}
```

7.9 Die TClientDataSet-Datenbankoberfläche

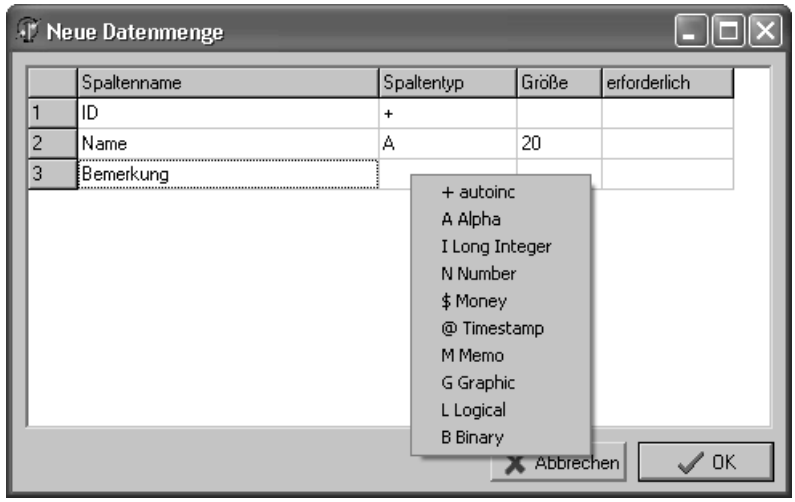
Die Erstellung neuer Tabellen mittels *TFieldDefs* ist zwar nicht weiter schwierig, aber den Begriff »komfortabel« würde man eher nicht dafür verwenden. Wir wollen uns deshalb ein kleines Tool für die wichtigsten Aufgaben basteln:

- Anlegen neuer Tabellen
- Editieren der Daten
- Importieren von BDE-Tabellen

Die Anwendung arbeitet mit einer MDI-Oberfläche, so dass man stets mehrere Fenster geöffnet halten kann.

7.9.1 Anlegen neuer Tabellen

Das Fenster zum Anlegen neuer Tabellen ist der Datenbankoberfläche von Delphi nachempfunden:



Zunächst geben Sie den Spaltennamen ein. Mit der ENTER-Taste öffnen Sie automatisch ein Kontextmenü zur Auswahl des Spaltentyps. Je nachdem, was Sie hier auswählen, gelangen Sie in die *Spalte*, *Größe*, *erforderlich* oder gleich wieder in *Spaltenname*.

Mit einem Klick auf *Ok* legen Sie dann die neue Tabelle an.

Der Menüpunkt NEU ist mit der Action *act_new* verbunden, deren *OnExecute*-Ereignisbehandlungsroutine eine Instanz von *TFrm_new* erzeugt und diese anzeigt.

```
procedure Tfrm_main.act_newExecute(Sender: TObject);
var
  fo_new: TFrm_new;
begin
  fo_new := TFrm_new.Create(Application);
  fo_new.Show;
end;
```

Zuerst werden die Spaltenüberschriften gesetzt:

```
procedure Tfrm_new.FormCreate(Sender: TObject);
begin
  with StringGrid1 do
  begin
    cells[1, 0] := 'Spaltenname';
    cells[2, 0] := 'Spaltentyp';
    cells[3, 0] := 'Größe';
    cells[4, 0] := 'erforderlich';
    cells[0, 1] := '1';
  end;
end; {procedure Tfrm_new.FormCreate}
```

Bei der nachfolgenden Prozedur wollen wir alle lokalen Prozeduren im Anschluss besprechen:

```
procedure Tfrm_new.StringGrid1KeyDown(Sender: TObject;
  var Key: Word;  Shift: TShiftState);
  ...
begin
  with StringGrid1 do
  begin
    if Key = VK_RETURN then
    begin
      case Col of
        1: DoEins;
        2: DoZwei;
        3: DoDrei;
        4: DoVier;
      end;
    end; {if Key = VK_RETURN}
```

```
    if Key = VK_INSERT
      then DoInsert;
    if (Key = VK_SPACE) and (Col = 4)
      then Cells[Col, Row] := 'x';
  end; {with StringGrid1 do}
end; {procedure Tfrm_new.StringGrid1KeyDown}
```

Wird die ENTER-Taste betätigt, dann wird in Abhängigkeit zur aktuellen Spalte eine der lokalen Prozeduren aufgerufen. Mit der INSERT-Taste ruft man *DoInsert* auf und in der vierten Spalte fügt man mit der Leertaste ein *x* ein.

```
  procedure DoEins;
  var
    p: TPoint;
  begin
    with StringGrid1 do
    begin
      if Length(Cells[Col, Row]) > 0 then
      begin
        Col := 2;
        p := ClientToScreen(Point(Left, Top));
        PopupMenu1.PopUp(p.x, p.y);
      end;
    end; {with StringGrid1 do}
  end; {procedure DoEins}
```

In der ersten Spalte wird mit der ENTER-Taste – sofern schon die Eingabe eines Spaltennamens erfolgt ist – in die zweite Spalte gesprungen und das Kontextmenü angezeigt, damit der Spaltentyp gewählt werden kann.

```
  procedure DoVier;
  begin
    with StringGrid1 do
    begin
      if Row = RowCount - 1 then
      begin
        RowCount := RowCount + 1;
        Cells[0, RowCount - 1] := IntToStr(RowCount - 1);
      end;
      Row := Row + 1;
      Col := 1;
    end; {with StringGrid1 do}
  end; {procedure DoVier}
```

Von der vierten Spalten aus springt man einfach nur in die erste Spalte der darunter liegenden Zeile. Befindet man sich jedoch schon in der letzten Zeile, dann muss eine neue eingefügt werden.

```
procedure DoDrei;
begin
  with StringGrid1 do
  begin
    if Length(Cells[Col, Row]) > 0
      then StringGrid1.Col := 4;
  end; {with StringGrid1 do}
end; {procedure DoDrei}
```

Aus der dritten Spalten springt man in die vierte.

```
procedure DoZwei;
var
  s: string;
begin
  with StringGrid1
    do s := Cells[Col, Row];
  if Length(s) > 0 then
  begin
    case s[1] of
      'a', 'A', 'm', 'M', 'g', 'G', 'b', 'B':
        StringGrid1.Col := 3;
      '+': DoVier;
      else DoDrei;
    end;
  end; {if Length(s) > 0 then}
end; {procedure DoZwei}
```

Handelt es sich beim eingegebenen Spaltentyp um ein alphanumerisches Feld, um ein Memo, eine Grafik oder ein binäres Feld, dann muss eine Länge eingegeben werden, dementsprechend wird nach Spalte drei gesprungen. Wird keine Länge benötigt, dann wird in die vierte Spalte gesprungen, um zu kennzeichnen, ob eine Eingabe erforderlich ist. Bei selbstinkrementierenden Feldern ist auch das nicht sinnvoll, mit *DoVier* wird gleich an den Anfang der nächsten Zeile gesprungen.

```
procedure DoInsert;
var
  i: integer;
```

```
  begin
    with StringGrid1 do
    begin
      if Row < RowCount - 1 then
      begin
        RowCount := RowCount + 1;
        Cells[0, RowCount - 1] := IntToStr(RowCount - 1);
        for i := RowCount - 1 downto Row + 1 do
        begin
          Cells[1, i] := Cells[1, i - 1];
          Cells[2, i] := Cells[2, i - 1];
          Cells[3, i] := Cells[3, i - 1];
          Cells[4, i] := Cells[4, i - 1];
        end; do}
        Cells[1, Row] := '';
        Cells[2, Row] := '';
        Cells[3, Row] := '';
        Cells[4, Row] := '';
      end {if Row < RowCount - 1 then}
      else DoDrei;
    end; {with StringGrid1 do}
  end; {procedure DoInsert}
```

Mit *DoInsert* wird die Anzahl der Zeilen um eine erhöht und alle Zeilen, die sich unter der einzufügenden befinden, werden um jeweils eins nach unten verschoben.

Klick auf den Ok-Button

Wird auf den *Ok*-Button geklickt, dann wird ein neues Kindfenster erzeugt und mit der dortigen Komponente *ClientDataSet1* eine neue Tabelle generiert:

```
procedure Tfrm_new.btnOkClick(Sender: TObject);
var
  frm: Tfrm_child;
  i: integer;
  s: string;
  required: boolean;

  function GetDataType(s: string): TFieldType;
  begin
    case s[1] of
      '+': result := ftAutoInc;
```

```
      'A', 'a': result := ftString;
      'I', 'i': result := ftInteger;
      'N', 'n': result := ftFloat;
      '$': result := ftCurrency;
      '@': result := ftDateTime;
      'M', 'm': result := ftMemo;
      'G', 'g': result := ftGraphic;
      'L', 'l': result := ftBoolean;
      'B', 'b': result := ftBlob;
    end;
  end; {function GetDataType}

begin
  frm := Tfrm_child.Create(Application);
  with StringGrid1 do
  begin
    for i := 1 to RowCount - 1 do
    begin
      if (Length(Cells[1, i]) > 0)
        and (Length(Cells[2, i]) > 0) then
      begin
        s := Cells[4, i];
        required := (Length(s) > 0)
          and ( s[1] in ['x', 'j', '*', 'y']);
        frm.ClientDataSet1.FieldDefs.Add(Cells[1, i],
          GetDataType(cells[2, i]), StrToIntDef(cells[3, i], 0),
          required);
      end;
    end; {for i := 1 to RowCount - 1 do}
  end; {with StringGrid1 do}
  frm.ClientDataSet1.CreateDataSet;
  frm.Show;
  Close;
end; {procedure Tfrm_new.btnOkClick}
```

Die Prozedur *btnOkClick* geht durch *StringGrid1* und legt für alle Spalten, die einen Namen und einen Feldtyp haben, einen *FieldDefs*-Eintrag an. Als erforderlich wird eine Spalte gekennzeichnet, wenn in der Spalte ein Zeichen wie *, *x* oder *j* steht. Den Datentyp ermittelt die lokale Funktion *GetDataType*.

Anschließend wird *CreateDataSet* aufgerufen, das Anzeigefenster angezeigt und das eigene Fenster geschlossen.

Das Nötigste wäre damit getan, aber wir wollen auch noch dafür sorgen, dass der Anwender nicht allzu viel »Mist« machen kann:

```
procedure Tfrm_new.StringGrid1KeyPress(Sender: TObject;
  var Key: Char);
  var
    s: string;

  procedure DoBezeichner;
  begin
    if Length(s) = 0 then
    begin
      if not (Key in ['a'..'z', 'A'..'Z', '_', #8])
        then Key := #0;
    end {if Length(s) = 0 then}
    else
    begin
      if not (Key in ['a'..'z', 'A'..'Z', '_', '0'..'9', #8])
        then Key := #0;
    end; {else Length(s) = 0 then}
  end; {procedure DoBezeichner}

begin
  with StringGrid1
    do s := Cells[Col, Row];
  case StringGrid1.Col of
    1: DoBezeichner;
    2: if (not (Key in ['+', 'a', 'A', 'i', 'I', 'n', 'N', '$', '@',
              'm', 'M', 'g', 'G', 'l', 'L', 'b', 'B', #8]))
           or (Length(s) > 0)
         then Key := #0;
    3: if not (Key in ['0'..'9', #8])
         then Key := #0;
    4: if (not (Key in ['x', 'j', '*', 'y', #8]))
           or (Length(s) > 0)
         then Key := #0;
  end; {case StringGrid1.Cols of}
end; {procedure Tfrm_new.StringGrid1KeyPress}
```

In der ersten Spalte wird sichergestellt, dass für die Bezeichner keine Sonderzeichen mit Ausnahme des Unterstrichs verwendet werden. Als erstes Zeichen sollen außerdem keine Ziffern zulässig sein.

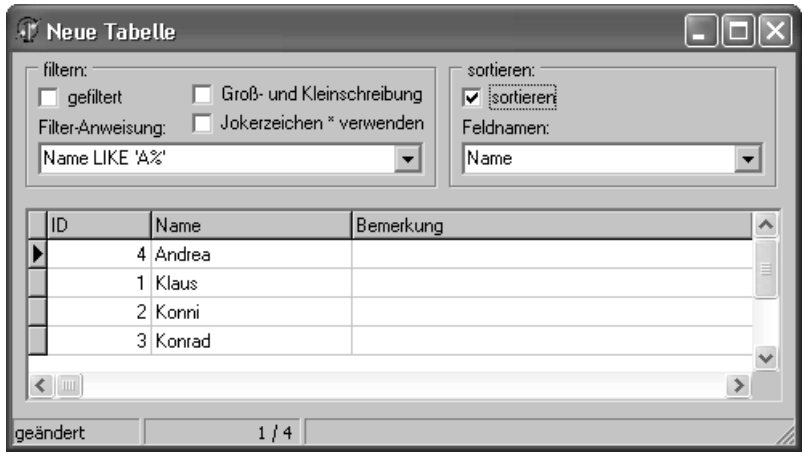
In der zweiten Spalte möchten wir nur die Abkürzungen für die Spaltentypen haben, in der dritten (Länge der Spalte) nur Ziffern und in der vierten Spalte (*erforderlich*) nur Zeichen, die sich irgendwie als Zustimmung interpretieren lassen.

```
procedure Tfrm_new.btnAbortClick(Sender: TObject);
begin
  if FChanged then
  begin
    if MessageDlg('Änderungen gehen verloren. Weiter?',
mtConfirmation,
        [mbYes, mbNo], 0) = mrYes
      then Close;
  end {if FChanged then}
  else Close;
end; {procedure Tfrm_new.btnAbortClick}
```

Des Weiteren noch eine kleine Sicherung gegen unbeabsichtigtes Schließen des Formulars.

7.9.2 Tabellen ansehen

Zum Ansehen und Ändern der Daten wird dasselbe Fenster verwendet, unabhängig davon, ob nun eine neue Tabelle erstellt oder eine bestehende geöffnet wurde:



Zur Eingabe der Daten verwendet man ein gewöhnliches DBGrid, des Weiteren gibt es die Möglichkeit, den Datenbestand zu filtern und zu sortieren.

Speichern der Daten

Wurde eine bestehende Tabelle geöffnet, dann steht deren Dateiname in der Titelleiste des Fensters. Mit ihm wird dann *SaveFile* aufgerufen. Steht *Neue Tabelle* in der Titelleiste, wird die Routine von SPEICHERN UNTER verwendet.

```
procedure Tfrm_child.Save;
begin
  if Caption <> 'Neue Tabelle'
    then SaveFile(Caption)
    else SaveAs;
end; {procedure Tfrm_child.Save}
```

Die Routine *SaveAs* besorgt über einen Dateidialog einen Dateinamen und prüft ab, ob damit vielleicht eine bestehende Datei überschrieben würde. Wenn ja, erfolgt eine Sicherheitsabfrage. Letztlich wird auch hier wieder *SaveFile* aufgerufen.

```
procedure Tfrm_child.SaveAs;
var
  s: string;
begin
  if SaveDialog1.Execute then
  begin
    s := SaveDialog1.FileName;
    if FileExists(s) then
    begin
      if MessageDlg('Datei ' + s
          + ' existiert bereit. Überschreiben?', mtConfirmation,
          [mbYes, mbNo], 0) = mrYes
        then SaveFile(s);
    end
    else SaveFile(s);
  end; {if SaveDialog1.Execute then}
end; {procedure Tfrm_child.SaveAs}
```

Aus der Dateiendung erkennt die Prozedur *SaveFile*, in welchem Format gespeichert werden soll. Der Dateiname wird dann auch in die Titelzeile des Fensters geschrieben.

Mittels *Changed* wird festgestellt, ob nach der letzten Speicherung noch Änderungen erfolgt sind.

```
procedure Tfrm_child.SaveFile(FileName: string);
var
  Format: TDataPacketFormat;
begin
  if ExtractFileExt(FileName) = '.cds'
    then Format := dfBinary
    else Format := dfXML;
  ClientDataSet1.SaveToFile(FileName, Format);
  Caption := FileName;
  Changed := false;
end; {procedure Tfrm_child.SaveFile}
```

Wird versucht, das Formular zu schließen, solange Änderungen noch nicht gespeichert wurden, erfolgt eine Sicherheitsabfrage.

```
procedure Tfrm_child.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
var
  mr: TModalResult;
begin
  if Changed then
  begin
    mr := MessageDlg('Tabelle wurde geändert. Speichern?',
      mtConfirmation, [mbYes, mbNo, mbAbort], 0);
    if mr = mrAbort
      then CanClose := false;
    if mr = mrYes
      then Save;
  end; {if Changed then}
end; {procedure Tfrm_child.FormCloseQuery}
```

Bei Änderungen am Datenbestand muss natürlich *Changed* auf *true* gesetzt werden:

```
procedure Tfrm_child.ClientDataSet1AfterPost(DataSet: TDataSet);
// für AfterPost und AfterDelete
begin
  Changed := true;
end;
```

Sortieren

Soll die Datenmenge sortiert werden, dann werden die Spalten in der gewünschten Reihenfolge in *ComboSort* eingegeben. Dieser Text wird anschließend der *TClientDataSet*-Eigenschaft *IndexFieldNames* zugewiesen:

```
procedure Tfrm_child.cbSortClick(Sender: TObject);
begin
  if cbSort.Checked then
  begin
    ClientDataSet1.IndexFieldNames := comboSort.Text;
    AbInDieCombo(comboSort);
  end;
end;
```

Damit man nicht immer dasselbe eingeben muss, werden die Sortieranweisungen in die Nachschlageliste der ComboBox geschrieben. Ist die betreffende Kombination dort bereits vorhanden, wird sie lediglich an die erste Stelle gesetzt.

```
procedure AbInDieCombo(Box: TComboBox);
begin
  if Box.Text = '' then exit;
  if Box.Items.IndexOf(Box.Text) >= 0
    then Box.Items.Move(Box.Items.IndexOf(Box.Text), 0)
    else Box.Items.Insert(0, Box.Text);
end;
```

Filtern

Des Weiteren soll auch noch die Möglichkeit geschaffen werden, den Datenbestand zu filtern:

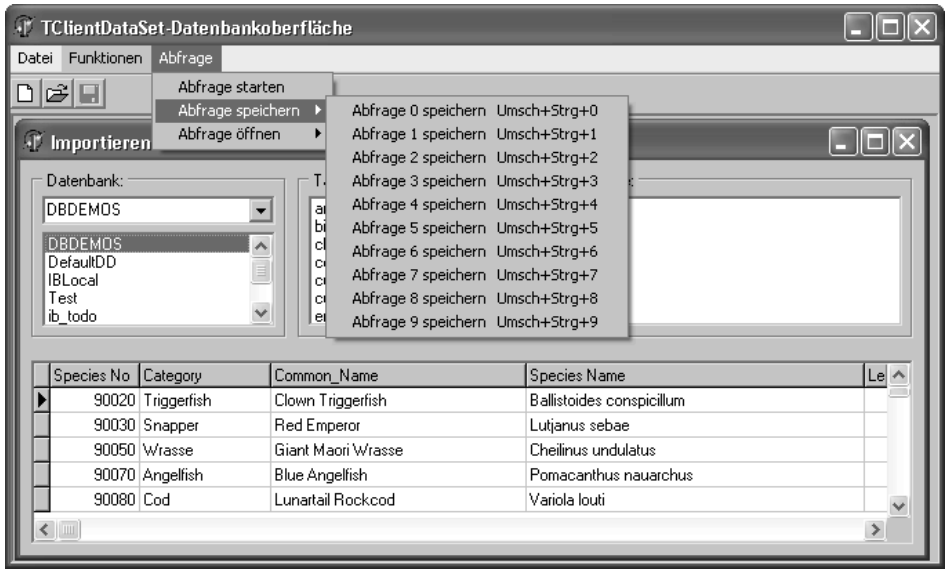
```
procedure Tfrm_child.chkFilteredClick(Sender: TObject);
var
  fo: TFilterOptions;
begin
  chkCase.Enabled := not chkFiltered.Checked;
  chkJoker.Enabled := not chkFiltered.Checked;
  comboFilter.Enabled := not chkFiltered.Checked;
  ClientDataSet1.Filter := comboFilter.Text;
  AbInDieCombo(comboFilter);
  fo := [];
  if not chkCase.Checked
    then include(fo, foCaseInsensitive);
```

```
  if not chkJoker.Checked
    then include(fo, foNoPartialCompare);
  ClientDataSet1.FilterOptions := fo;
  try
    ClientDataSet1.Filtered := chkFiltered.Checked;
  except
    chkFiltered.Checked := false;
  end;
end; {procedure Tfrm_child.CheckBox1Click}
```

Solange gefiltert wird, dürfen weder die Filteranweisung noch die Optionen geändert werden. Die Filteranweisung kommt dann wieder in die Nachschlageliste. Je nach Zustand der CheckBoxen werden die Filteroptionen gesetzt. Führt die Filteranweisung zu einer Exception, wird die Filterung automatisch zurückgenommen.

7.9.3 Importieren von BDE-Daten

Um Daten aus BDE-Tabellen und -Abfragen zu extrahieren, importieren Sie diese und speichern sie auf die Festplatte.



Auch hier gibt es wieder ein wenig »Luxus«: Bis zu zehn SQL-Anweisungen lassen sich abspeichern und später über das Menü oder das dazugehörende Tastenkürzel wieder aufrufen.

Beim Erzeugen des Formulars wird eine Liste aller BDE-Aliase nach *lbAlias* geschrieben:

```
procedure Tfrm_import.FormCreate(Sender: TObject);
begin
  Session.GetAliasNames(lbAlias.Items);
  AbfrageSpeichernMenuErstellen;
end;
```

Die BDE arbeitet nicht nur mit Aliasen, sondern auch mit Dateien und Verzeichnissen. Diese können in *cmbAlias* eingegeben werden. Mit einem Doppelklick auf einen Alias in der ListBox wird dieser in die ComboBox geschrieben.

```
procedure Tfrm_import.lbAliasDblClick(Sender: TObject);
begin
  cmbAlias.Text := lbAlias.Items[lbAlias.ItemIndex];
  NeuerAlias;
end;
```

Mit dem neuen Alias werden die Tabellennamen nach *lbTable* geschrieben.

```
procedure Tfrm_import.NeuerAlias;
begin
  AbInDieCombo(cmbAlias);
  Session.GetTableNames(cmbAlias.Text, '', true, false,
lbTable.Items);
end;
```

Mit einem Doppelklick auf den Tabellennamen sollen die Daten aus der Tabelle geladen werden:

```
procedure Tfrm_import.lbTableDblClick(Sender: TObject);
begin
  Screen.Cursor := crHourGlass;
  try
    gbTable.Font.Color := clGreen;
    gbQuery.Font.Color := clBlack;
    ClientDataSet1.Close;
    with Table1 do
    begin
      Close;
      DatabaseName := cmbAlias.Text;
      TableName := lbTable.Items[lbTable.ItemIndex];
      Open;
    end; {with Table1 do}
```

```
    DataSetProvider1.DataSet := Table1;
    ClientDataSet1.Open;
  finally
    Screen.Cursor := crDefault;
  end;
end; {procedure Tfrm_import.lbTableDblClick}
```

Zunächst wird als Zeichen dafür, dass nun die Daten aus einer Tabelle kommen, der Titel der GroupBox *gbTable* grün dargestellt. Die Client-Datenmenge und die Table werden geschlossen, *DatabaseName* und *TableName* eingestellt, die Tabelle wieder geöffnet, der Provider angehängt und mit dem Öffnen der Client-Datenmenge die Datensätze importiert.

Ähnlich funktioniert das Starten der Abfrage:

```
procedure Tfrm_import.AbfrageStarten;
begin
  Screen.Cursor := crHourGlass;
  try
    gbQuery.Font.Color := clGreen;
    gbTable.Font.Color := clBlack;
    ClientDataSet1.Close;
    with Query1 do
    begin
      Close;
      DatabaseName := cmbAlias.Text;
      SQL := Memo1.Lines;
      Open;
    end; {with Query1 do}
    DataSetProvider1.DataSet := Query1;
    ClientDataSet1.Open;
  finally
    Screen.Cursor := crDefault;
  end;
end; {procedure Tfrm_import.AbfrageStarten}
```

Statt eines Tabellennamens wird hier der Inhalt von *Memo1* als SQL-Statement zugewiesen. Zudem wird hier der Titel von *cbQuery* grün und der von *cbTable* schwarz angezeigt.

Speichern der Abfragen

Es müssen zehn Menüpunkte erstellt werden, um die Abfragen zu speichern, weitere zehn Menüpunkte, um die gespeicherten Abfragen aufzurufen, sowie zehn String-Listen, um die SQL-Statements zu speichern. Für alle Menüpunkte werden ShortCuts eingerichtet. Solange noch keine Abfragetexte gespeichert sind, sind die Menüpunkte zum Aufrufen derselben auch nicht verfügbar.

```
procedure Tfrm_import.AbfrageSpeichernMenuErstellen;
var
  i: integer;
  s: string;
  MenuItem: TMenuItem;

  procedure MakeMenuSpeichern;
  begin
    MenuItem := TMenuItem.Create(Self);
    with MenuItem do
    begin
      Caption := 'Abfrage &' + IntToStr(i) + ' speichern';
      Tag := i;
      ShortCut := TextToShortCut('Umsch+Strg+' + IntToStr(i));
      OnClick := AbfrageSpeichernClick;
    end;
    Abfragespeichern1.Add(MenuItem);
  end; {procedure MakeMenuItem}

  procedure MakeMenuFfnen;
  begin
    MenuItem := TMenuItem.Create(Self);
    with MenuItem do
    begin
      Caption := 'Abfrage &' + IntToStr(i) + ' speichern';
      Tag := i;
      ShortCut := TextToShortCut('Strg+' + IntToStr(i));
      Enabled := false;
      OnClick := AbfrageFfnenClick;
    end;
    Abfrageffnen1.Add(MenuItem);
  end; {procedure MakeMenuFfnen}
```

```
begin
  for i := 0 to 9 do
  begin
    MakeMenuSpeichern;
    MakeMenuFfnen;
    FAbfrage[i] := TStringList.Create;
  end; {for i := 0 to 9 do}
end; {procedure Tfrm_import.AbfrageSpeichernMenuErstellen}
```

Zum Speichern muss lediglich der Inhalt von *Memo1* in die dafür vorgesehene String-Liste gespeichert und der Menüpunkt zum Aufrufen dieser Abfrage verfügbar geschaltet werden.

```
procedure Tfrm_import.AbfrageSpeichernClick(Sender: TObject);
var
  i: integer;
begin
  i := TMenuItem(Sender).Tag;
  FAbfrage[i].Assign(Memo1.Lines);
  Abfrageffnen1.Items[i].Enabled := true;
end;
```

Zum Öffnen der Abfrage wird der Text nach *Memo1* zurückgeschrieben:

```
procedure Tfrm_import.AbfrageFfnenClick(Sender: TObject);
begin
  Memo1.Lines.Assign(FAbfrage[TMenuItem(Sender).Tag]);
end;
```

Beim Freigeben des Formulars müssen dann die zehn String-Listen weggeräumt werden.

```
procedure Tfrm_import.FormDestroy(Sender: TObject);
var
  i: integer;
begin
  for i := 0 to 9
    do FAbfrage[i].Free;
end;
```

Des Weiteren gibt es die Routine *SaveAs* zum Speichern der Datensätze auf die Festplatte, aber das ist im Wesentlichen eine Zusammenfassung von dem, was wir aus Kapitel 7.9.2 als *SaveAs* und *SaveFile* kennen.

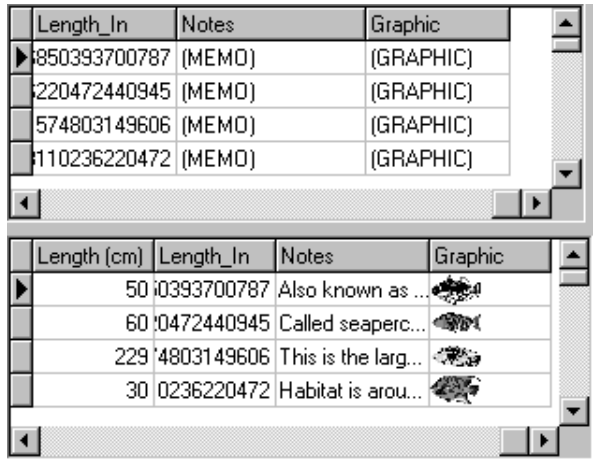
8 Datensteuerung

Nach meinem Dafürhalten ist die Bezeichnung *Datensteuerung* etwas unglücklich gewählt, denn es werden keine Daten gesteuert, sondern allenfalls – über *TDBNavigator* – Datenmengenkomponenten. Alle anderen Komponenten dieser Palettenseite dienen dem Anzeigen und Ändern von Daten.

Die Komponente *TDBChart* wird in Kapitel 9 besprochen.

8.1 TDBGrid

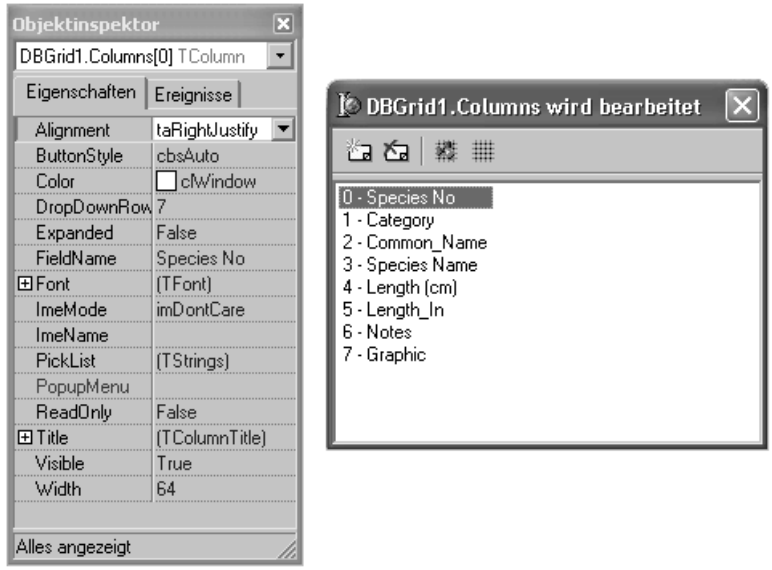
Die Komponente *TDBGrid* dient zur Anzeige von Daten in Tabellenform. *TDBGrid* ist von *TWinControl* abgeleitet.



Der eine oder andere Leser wird sich vermutlich über die Anzeige der Bilder im Grid wundern und auch die Bemerkungen werden normalerweise nicht angezeigt. Wie man das trotzdem hinbekommt, wollen wir uns gleich ansehen.

Anlegen der Spaltendefinitionen

Solange nichts anderes eingestellt wird, werden im DBGrid alle Spalten einer Datenmenge angezeigt. Möchte man nun einige Spalten ausschließen oder Spalteneigenschaften ändern, dann wird mit einem Doppelklick auf das DBGrid (alternativ kann man aus dem Kontextmenü SPALTENEDITOR wählen) der Spaltenditor aufgerufen:



Anfangs wird die Liste leer sein. Man könnte nun jede Spalte einzeln hinzufügen, meist ist es jedoch deutlich effektiver, erst einmal alle Spalten hinzuzufügen und die nicht benötigten dann zu löschen (oder man setzt deren Eigenschaft *Visible* auf *false*). Persistente Spaltendefinitionen sind die Voraussetzung dafür, dass zur Entwurfszeit vorgenommene Änderungen der Spaltenbreiten zur Laufzeit wirksam werden. Jede Spaltendefinition ist vom Typ *TColumn*, die Klasse wird im Referenzteil beschrieben.

Anzeige von Bildern und Memos

Wie die Abbildung am Anfang des Kapitels zeigt, werden in Bilder- und Memo-Spalten für gewöhnlich die Texte *(GRAPHIC)* und *(MEMO)* ausgegeben. Soll der Anfang eines Memo-Textes oder eine Abbildung als Miniatur angezeigt werden, dann ist eine entsprechende *OnDrawColumnCell*-Ereignisbehandlungsroutine zu erstellen.

Um nicht beim Zeichnen einer jeden Bild- oder Memo-Zelle die entsprechende *TPicture*- oder *TMemo*-Instanz erstellen zu müssen, erledigen wir das beim Erstellen des Formulars.

```
type
  TForm1 = class(TForm)
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    procedure DBGrid1DrawColumnCell(Sender: TObject;
      const ARect: TRect;
      DataCol: Integer; Column: TColumn; State: TGridDrawState);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    FPic: TPicture;
    FSl: TStringList;
  end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  FPic := TPicture.Create;
  FSl := TStringList.Create;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  FPic.Free;
  FSl.Free;
end;

procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const ARect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
  ...
begin
  if Column.Field is TGraphicField
    then DrawImage;
  if Column.Field is TMemoField
    then DrawMemo;
end; {procedure TForm1.DBGrid1DrawColumnCell}
```

Mit dem Parameter *Field* wird geprüft, ob der zugrunde liegende Feldtyp ein Memo-Feld oder ein Grafik-Feld ist und gegebenenfalls die entsprechende lokale Prozedur aufgerufen.

```
procedure DrawImage;
var
  i: integer;
  NRect: TRect;
begin
  FPic.Assign(Column.Field as TGraphicField);
  i := round((ARect.Bottom - ARect.Top)
    * (FPic.Width / (FPic.height + 0.001)));
  NRect := Rect(ARect.Left, ARect.Top, ARect.Left + i,
    ARect.Bottom);
  with DBGrid1.Canvas do
  begin
    FillRect(ARect);
    StretchDraw(NRect, FPic.Bitmap);
  end;
end; {procedure DrawImage}
```

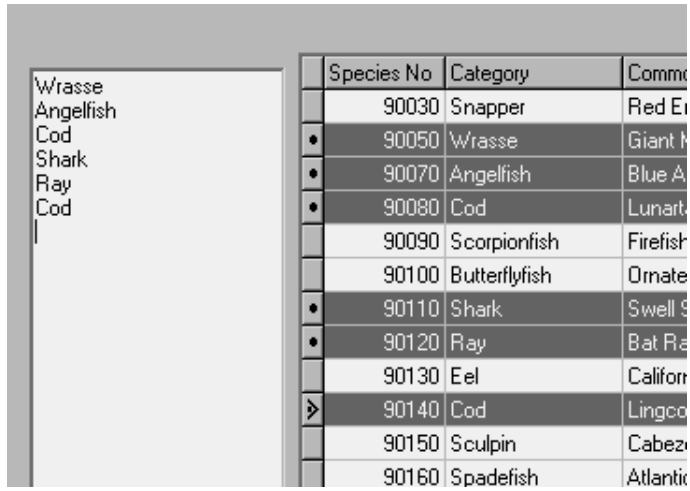
Zur Ausgabe einer Grafik wird zunächst eine *TRect*-Struktur berechnet, die in die Zelle passt und die dem Höhen-Breiten-Verhältnis der ursprünglichen Grafik entspricht. In dieses Rechteck wird das Bild dann mit der *TCanvas*-Methode *StretchDraw* kopiert, wobei es in der Regel verkleinert wird.

```
procedure DrawMemo;
var
  i: integer;
  s: string;
begin
  FSl.Assign(Column.Field as TMemoField);
  i := ARect.Right - ARect.Left - Canvas.TextWidth('...');
  if Fsl.count > 0
    then s := FSl.Strings[0]
    else s := '';
  with DBGrid1.Canvas do
  begin
    repeat
      s := copy(s, 0, (length(s) - 1));
    until TextWidth(s) < i;
    FillRect(ARect);
    TextRect(ARect, ARect.Left, ARect.Top, s + '...');
  end;
end; {procedure DrawMemo}
```

Bei einem Memo-Feld wird geprüft, wie viel Text in den zur Verfügung stehenden Platz passt. Dieser wird dann zusammen mit drei Punkten (als Zeichen dafür, dass der Text noch nicht zu Ende ist) von der *TCanvas*-Eigenschaft *TextRect* ausgegeben.

Mehrere selektierte Reihen

Wird die Option *dgMultiSelect* gesetzt, dann können mit Hilfe der STRG-Taste mehrere Datensätze markiert werden. Wir wollen im folgenden Beispiel von den markierten Datensätzen die Spalte *Category* in ein Memo schreiben:



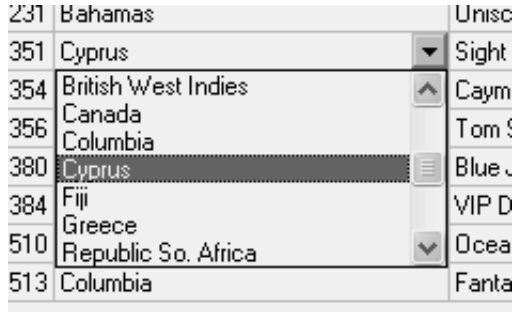

```
procedure TForm1.Memo1DblClick(Sender: TObject);
var
  i: integer;
  p: Pointer;
begin
  Memo1.Lines.Clear;
  p := Table1.GetBookmark;
  for i := 0 to DBGrid1.SelectedRows.Count - 1 do
  begin
    Table1.Bookmark := DBGrid1.SelectedRows[i];
    Memo1.Lines.Add(Table1.FieldByName('Category').AsString);
  end;
  Table1.GotoBookmark(p);
end; {procedure TForm1.Memo1DblClick}
```

Zunächst löschen wir den bisherigen Inhalt des Memos, dann speichern wir die aktuelle Position des Satzzeigers, um sie nachher wiederherstellen zu können. Die *TDBGrid*-Eigenschaft *SelectedRows* ist vom Typ *TBookmarklist*, also eine Liste von Lesezeichen. Mit deren Hilfe wird der Satzzeiger von *Table1* auf die markierten Datensätze gesetzt, anschließend kann der Inhalt von *Category* ausgelesen werden. Zuletzt wird die alte Position des Satzzeigers wiederhergestellt.

Bitte beachten Sie, dass es zwei Arten von Lesezeichen gibt, die wir hier beide verwenden: Die eine ist vom Typ *Pointer*, die andere vom Typ *string*.

Nachschlagelisten

Mit Hilfe der *TColumn*-Eigenschaft *PickList* lässt sich eine Nachschlageliste definieren, aus der im Grid mittels einer ComboBox gewählt werden kann:



Man könnte mit einer solchen Nachschlageliste beispielsweise die Liste aller Länder hinterlegen. Die Abbildung hier ist aber anders entstanden: Für Lookup-Felder füllt *TDBGrid* ganz automatisch eine solche Nachschlageliste, ohne dass man auch nur irgendeinde Eigenschaft setzen müsste.

8.1.1 Referenz TDBGrid

- DataSource (Eigenschaft, veröffentlicht)

  ```
  property DataSource: TDataSource;
  ```

  Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden.

- OnCellClick, OnTitleClick (Ereignisse)

  ```
  property OnCellClick(Column: TColumn);
  property OnTitleClick(Column: TColumn);
  ```

  Das Ereignis *OnCellClick* wird nach dem Anklicken einer Zelle im Datenbereich ausgelöst, das Ereignis *OnTitleClick* nach dem Anklicken einer Titelzeile.

- OnEditButtonClick (Ereignis)

```
property OnEditButtonClick(Sender: TObject);
```

Wenn Sie die *TColumn*-Eigenschaft *ButtonStyle* auf den Wert *cbsEllipsis* setzen, dann wird, sobald die Zelle ausgewählt wird, ein Button mit drei Punkten angezeigt. Das Ereignis *OnEditButtonClick* wird angezeigt, wenn dieser Button angeklickt wird.

- SelectedField, SelectedIndex (Eigenschaften, öffentlich)

```
property SelectedField: TField;
property SelectedIndex: Integer;
```

Mit diesen beiden Eigenschaften kann die selektierte Zelle bestimmt werden. *SelectedField* verwendet dabei die Feldinstanz, *SelectedIndex* den Index der Spalte.

- OnColEnter, OnColExit (Ereignisse),

```
property OnColEnter(Sender: TObject);
property OnColExit(Sender: TObject);
```

Das Ereignis *OnColEnter* tritt auf, wenn die Zelle einer anderen Spalte den Fokus erhält, *OnColExit*, wenn sie ihn verliert. Das bloße Wechseln der Reihe löst diese Ereignisse nicht aus.

Bei *OnColExit* referenzieren *SelectedField* und *SelectedIndex* noch die alte Zelle, bei *OnColEnter* die neue.

- OnColumnMoved (Ereignis)

```
property OnColumnMoved(Sender: TObject;
  FromIndex, ToIndex: Integer);
```

Wenn die Option *dgColumnResize* gesetzt ist, können die Spalten zur Laufzeit verschoben werden. In diesem Fall wird das Ereignis *OnColumnMoved* ausgelöst.

- SelectedRows (Eigenschaft, öffentlich, nur Lesen)

```
property SelectedRows: TBookmarkList;
```

Ist die Option *dgMultiSelect* gesetzt, dann können mehrere Reihen markiert werden. Welche das sind, erfahren Sie über *SelectedRows*. Mehr dazu im Abschnit *mehrere selektierte Reihen*.

TBookmarkList ist von *TObejct* und nicht von *TList* abgeleitet und implementiert (unter anderem) die folgenden Eigenschaften und Methoden:

- Mit der booleschen Eigenschaft *CurrentRowSelected* kann man die aktuelle Reihe in die Liste aufnehmen oder daraus entfernen. (Dies ist auch der einzige Weg, um neue Elemente in die Liste zu bekommen.)
- Die Anzahl der Listenelemente ermittelt man mit *Count*, ein Zugriff auf diese erfolgt mit *Items*. (*Items* ist vom Typ String, beginnt aber in der Regel mit einer terminierenden Null, so dass eine Anzeige nicht möglich ist.)
- Mit *Clear* löscht man die Liste. Mit *Delete* (aufpassen!) löscht man alle markierten Werte aus der Datenmenge.
- Mit *Find* und *IndexOf* kann man ermitteln, ob ein bestimmtes Lesezeichen in der Liste ist und wenn ja, an welcher Position man es dort findet.

- Options (Eigenschaft, veröffentlicht)

```
property Options: set of (dgEditing, dgAlwaysShowEditor, dgTitles,
  dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs,
  dgRowSelect,dgAlwaysShowSelection, dgConfirmDelete,
  dgCancelOnExit, dgMultiSelect) default [dgEditing, dgTitles,
  dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs,
  dgConfirmDelete, dgCancelOnExit];
```

Mit der Eigenschaft *Options* können Sie einige Optionen setzen. Eine Beschreibung aller Optionen finden Sie in der Online-Hilfe. Hier sollen die wichtigsten besprochen werden:

- *dgEditing* ist die Voraussetzung dafür, dass im Grid Daten geändert werden können.
- Sollen die Spaltenbreiten verändert werden können, dann ist *dgColumnResize* zu setzen.
- Die Auswahl mehrerer Datensätze erfordert *dgMultiSelect*, siehe auch den Abschnitt *mehrere selektierte Reihen*.
- Ist die Option *dgTabs* gesetzt, dann gelangt man mit der Tab-Taste von einer Spalte zur nächsten.
- Die Spaltentitel lassen sich ausblenden, indem man *dgTitles* entfernt.
- Verliert die Komponente den Fokus, dann wird auch die selektierte Zelle nicht mehr markiert. Soll sie auch nach dem Verlust des Fokus weiter markiert werden, dann ist die Option *dgAlwaysShowSelection* zu setzen.

Darstellung

- Columns (Eigenschaft, veröffentlicht)

```
property Columns: TDBGridColumns;
```

  Mit der Eigenschaft *Columns* können Einstellungen für die einzelnen Spalten vorgenommen werden. Der *TCollection*-Nachfolger *TDBGridColumns* implementiert eine *Items*-Eigenschaft vom Typ *TColumn*, welche im nächsten Abschnitt beschrieben ist.

- Font, TitleFont (Eigenschaften, veröffentlicht)

```
property Font: TFont;
property TitleFont: TFont;
```

  Mit *Font* spezifiziert man die Schrift, mit der die Daten angezeigt werden, mit *TitleFont* die Schrift, mit der die Spaltentitel geschrieben werden.

- DefaultDrawing (Eigenschaft, veröffentlicht), OnDrawDataCell (Ereignis)

```
property DefaultDrawing: Boolean default true;
property OnDrawDataCell(Sender: TObject; const Rect: TRect;
  Field: TField; State: TGridDrawState);
TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
```

  Wenn Sie alle Zelleninhalte selbst darstellen möchten, dann setzen Sie *DefaultDrawing* auf *false* und schreiben eine entsprechende Ereignisbehandlungsroutine für *OnDrawDataCell*.

  Wenn Sie nur einen Teil der Zellen selbst zeichnen möchten, dann lassen Sie *DefaultDrawing* auf *true*, die *OnDrawDataCell*-Ereignisbehandlungsroutine überschreibt dann die Ausgaben, welche das Datengitter selbst vorgenommen hat.

  Ein Beispiel für die Verwendung von *OnDrawDataCell* finden Sie im Abschnitt *Anzeige von Bildern und Memos*.

- OnDrawColumnCell (Ereignis)

```
property OnDrawColumnCell(Sender: TObject; const Rect: TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
```

  Auch das Ereignis *OnDrawColumnCell* kann zum Selbstzeichnen von Zelleninhalten verwendet werden. Statt der Feldinstanz referenziert das Ereignis *OnDrawColumnCell* die Spalteninstanz. Sie sollten die beiden Ereignisse nicht gleichzeitig verwenden.

8.1.2 Referenz TColumn

Der Typ *TColumn* repräsentiert die Einstellungen einer einzelnen Spalte eines Datengitters. Alle *TColumn*-Instanzen werden von der *TDBGrid*-Eigenschaft *Columns* zusammengefasst. *TColumn* ist von *TComponent* abgeleitet.

- FieldName (Eigenschaft, veröffentlicht), Field (Eigenschaft, öffentlich)

  ```
  property FieldName: string;
  property Field: TField;
  ```

  Mit *FieldName* wird spezifiziert, von welcher *TField*-Instanz die Daten bezogen werden. Eine Referenz auf das *TField*-Objekt erhält man mit *Field*.

- ReadOnly (Eigenschaft, veröffentlicht)

  ```
  property ReadOnly: Boolean default false;
  ```

  Wenn Sie verhindern möchten, dass in einem Datengitter die Werte der betreffenden Spalte geändert werden können, dann setzen Sie *ReadOnly* auf *true*.

- Visible (Eigenschaft, veröffentlicht)

  ```
  property Visible: Boolean default true;
  ```

  Wenn Sie (vorübergehend) verhindern möchten, dass die betreffende Spalte angezeigt wird, dann setzen Sie *Visible* auf *false*.

- PopupMenu (Eigenschaft, veröffentlicht)

  ```
  property PopupMenu: TPopupMenu;
  ```

  Mit *PopupMenu* können Sie der Spalte ein spezielles Kontextmenü zuweisen.

Darstellung

- Width (Eigenschaft, veröffentlicht)

  ```
  property Width: Integer;
  ```

  Mit *Width* wird die Spaltenbreite eingestellt.

- Title (Eigenschaft, veröffentlicht)

  ```
  property Title: TColumnTitle;
  ```

  Mit *Title* wird spezifiziert, wie die Spaltenüberschrift dargestellt werden soll. *TColumnTitle* implementiert unter anderem die folgenden Eigenschaften:

  - Alignment (Ausrichtung)
  - Caption (Text)

- Color (Zellenfarbe)
- Font (Schriftart)

- Alignment (Eigenschaft, veröffentlicht)

```
property Alignment: (taLeftJustify, taRightJustify, taCenter);
```

Mit *Alignment* wird die Ausrichtung des Textes in der Spalte eingestellt. Der Vorgabewert richtet sich danach, ob es sich um eine Text- (linksbündig) oder eine Zahlenspalte (rechtsbündig) handelt.

- Color, Font (Eigenschaften, veröffentlicht)

```
property Color: TColor;
property Font: TFont;
```

Mit *Color* wird die Hintergrundfarbe, mit *Font* die Schriftart eingestellt.

- RestoreDefaults (Methode)

```
procedure RestoreDefaults;
```

Mit *RestoreDefaults* werden die Vorgabewerte wiederhergestellt.

Nachschlageliste

- ButtonStyle, PickListe (Eigenschaften, veröffentlicht)

```
property ButtonStyle: (cbsAuto, cbsEllipsis, cbsNone)
  default cbsAuto;
property PickList: TStrings;
```

Mit der Eigenschaft *PickList* können Sie für die betreffende Spalte eine Nachschlageliste definieren. Näheres in der Online-Hilfe.

8.1.3 TBDBGrid

Wir haben vorhin auf das Ereignis *OnDrawColumnCell* reagiert, um Bilder und den Anfang von Memos anzuzeigen. Immer wieder den gleichen Code zu erstellen ist natürlich wenig effektiv, deshalb wollen wir uns dafür nun eine eigene Komponente erstellen. Statt der Nutzung des Ereignisses werden wir nun die Methode *DrawColumnCell* überschreiben.

Nun beschleunigt die Anzeige von Memos und Bildern den Ablauf naturgemäß nicht. Deshalb soll mit den Eigenschaften *AutoDisp_Image* und *AutoDisp_Memo* eingestellt werden können, ob Bilder und Memos angezeigt werden sollen oder nicht.

```
type
  TDBBGrid = class(TDBGrid)
  private
    FAutoDisp_Image: boolean;
    FAutoDisp_Memo: boolean;
    Pic: TPicture;
    Zeilen: TStringList;
  public
    constructor create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure DrawColumnCell(const Rect: TRect; DataCol: Integer;
      Column: TColumn; State: TGridDrawState);override;
  published
    property AutoDisp_Image: boolean read FAutoDisp_Image
      write FAutoDisp_Image default true;
    property AutoDisp_Memo: boolean read FAutoDisp_Memo
      write FAutoDisp_Memo default true;
  end;
```

Im Konstruktor werden *AutoDisp_Image* und *AutoDisp_Memo* auf *true* gesetzt, außerdem werden die Instanzen *Pic* und *Zeilen* erzeugt, die dann im Destruktor wieder freizugeben sind.

```
constructor TDBBGrid.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FAutoDisp_Image := true;
  FAutoDisp_Memo := true;
  Pic := TPicture.Create;
  Zeilen := TStringList.Create;
end;

destructor TDBBGrid.Destroy;
begin
  Pic.Free;
  Zeilen.Free;
  inherited Destroy;
end;
```

In der Prozedur *DrawColumnCell* werden dann die Bilder als Miniaturen und der Anfang von Memos in das Grid geschrieben.

```
procedure TDBBGrid.DrawColumnCell(const Rect: TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
var
  NRect: TRect;
  i: integer;
  weite: integer;
  s: string;
begin
  inherited;
  if Column.Field is TGraphicField then
  begin
    if FAutoDisp_Image = true then
    begin
      Pic.Assign(Column.Field as TGraphicField);
      i := round((Rect.Bottom - Rect.Top)
        *(Pic.Width / (Pic.height + 0.001)));
      NRect := Classes.Rect(Rect.Left, Rect.Top, Rect.Left + i,
        Rect.Bottom);
      with Canvas do
      begin
        FillRect(Rect);
        StretchDraw(NRect, Pic.Bitmap);
      end;
    end; {if FAutoDisp_Image = true then}
  end; {if Field ist TGraphicField then}

  if Column.Field is TMemoField then
  begin
    if FAutoDisp_Memo = true then
    begin
      Zeilen.Assign (Column.Field as TMemoField);
      weite := Rect.Right - Rect.Left - Canvas.TextWidth('...');
      s := Zeilen.Text;
      repeat
        s := copy(s, 0, (length(s) - 1));
      until Canvas.TextWidth(s) < weite;
      Casnvas.TextRect(Rect, Rect.Left, Rect.Top, s + '...');
    end; {if FAutoDisp_Image = true then}
  end; {if Field ist TMemoField then}
end; {procedure TDBBGrid.DrawColumnCell}
```

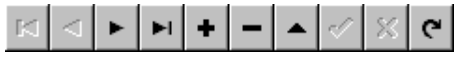
Mit *Columns.Field* wird geprüft, ob der zugrunde liegende Feldtyp ein Memo-Feld oder ein Grafik-Feld ist. Zur Ausgabe einer Grafik wird zunächst eine *TRect*-Struktur berechnet, die in die Zelle passt und die dem Höhen-Breiten-Verhältnis der ursprünglichen Grafik entspricht. In dieses Rechteck wird das Bild dann mit der *TCanvas*-Methode *StretchDraw* kopiert, wobei es in der Regel verkleinert wird.

Zur Verwendung der Funktion *Rect* muss der Unit-Bezeichner (*Classes*) vorangestellt werden, damit der Compiler diese Funktion vom Parameter *Rect* unterscheiden kann.

Bei einem Memo-Feld wird geprüft, wie viel Text in den zur Verfügung stehenden Platz passt. Dieser wird dann zusammen mit drei Punkten (als Zeichen dafür, dass der Text noch nicht zu Ende ist) von der *TCanvas*-Eigenschaft *TextRect* ausgegeben.

8.2 TDBNavigator

Die Komponente *TDBNavigator* dient zum Steuern einer Datenmenge. *TDBNavigator* ist von *TWinControl* abgeleitet.



- DataSource (Eigenschaft, veröffentlicht)

  ```
  property DataSource: TDataSource;
  ```

  Mit der Eigenschaft *DataSource* wird spezifiziert, welche Datenmenge gesteuert werden soll.

- VisibleButtons (Eigenschaft, veröffentlicht)

  ```
  property VisibleButtons: set of (nbFirst, nbPrior, nbNext, nbLast,
    nbInsert, nbDelete, nbEdit, nbPost, nbCancel, nbRefresh);
  ```

  Mit *VisibleButtons* können einzelne Buttons von der Anzeige ausgeschlossen werden. Per Voreinstellung werden alle Buttons angezeigt.

- ConfirmDelete (Eigenschaft, veröffentlicht)

  ```
  property ConfirmDelete: Boolean default true;
  ```

  Hat *ConfirmDelete* den Wert *true*, dann wird beim Löschen jedes Datensatzes über die Steuerelemente ein Dialogfenster zur Bestätigung aufgerufen. Beim Löschen über die *TDataSet*-Methode *Delete* entfällt diese Bestätigung.

- Hints (Eigenschaft, veröffentlicht)

  ```
  property Hints: TStrings;
  ```

  Um die Hinweistexte zu ändern, beispielsweise in eine andere Sprache zu übersetzen, ändern Sie die entsprechenden Einträge der Eigenschaft *Hints*.

- BeforeAction, OnClick (Eigenschaften)

  ```
  property BeforeAction(Sender: TObject; Button: TNavigateBtn);
  property OnClick(Sender: TObject; Button: TNavigateBtn);
  ```

  Wird ein Button angeklickt, wird erst das Ereignis *BeforeAction* ausgelöst, dann die betreffende Aktion durchgeführt und danach *OnClick* aufgerufen.

- BtnClick (Methode)

  ```
  procedure BtnClick(Index: TNavigateBtn);
  ```

  Mit der Methode *BtnClick* kann ein Mausklick auf einen der Buttons simuliert werden. Normalerweise würde man eher direkt die entsprechende *TDataSet*-Routine aufrufen, soll aber beispielsweise eine *BeforeAction*-Ereignisbehandlungsroutine mit ausgelöst werden, dann macht auch die Verwendung dieser Methode Sinn.

8.3 TDBText

Die Komponente *TDBText* dient zur Anzeige von Werten, die nicht verändert werden sollen – beispielsweise automatisch vergebene Nummern oder berechnete Werte. *TDBText* ähnelt in der Darstellung *TLabel*. *TDBText* ist von *TControl* abgeleitet.

Statt *TDBText* können Sie auch *TDBEdit* verwenden, *ReadOnly* und *ParentColor* auf *true* sowie *BorderStyte* auf *bsNone* setzen. Die Komponente würde nun genauso aussehen wie *TDBText*, hätte aber den Vorteil, dass man ihren Text markieren und über die Zwischenablage kopieren könnte.

- DataSource, DataField (Eigenschaft, veröffentlicht)

  ```
  property DataSource: TDataSource;
  property DataField: string;
  ```

  Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld.

- AutoSize (Eigenschaft, veröffentlicht)

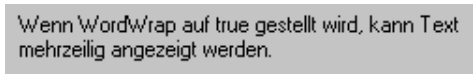
```
property AutoSize: boolean default true;
```

Wird die Eigenschaft *AutoSize* auf *true* gesetzt, dann passt sich die *TDBText*-Komponente dem darzustellenden Text an. Anderenfalls kann es passieren, dass der Text nicht vollständig angezeigt werden kann:

- WordWrap (Eigenschaft, veröffentlicht)

```
property WordWrap: boolean default false;
```

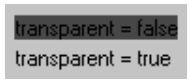
Um Text mehrzeilig anzuzeigen, muss die Eigenschaft *WordWrap* auf *true* gesetzt werden. Den Umbruch nimmt die Komponente selbst vor:



- Transparent (Eigenschaft, veröffentlicht)

```
property Transparent: boolean default false;
```

Im Regelfall wird der Hintergrund einer *TDBText*-Komponente mit der Farbe ausgefüllt, welche in der Eigenschaft *Color* angegeben ist. Soll jedoch als Hintergrund die hinter der *TDBText-Instanz* liegende Komponente angezeigt werden (beispielsweise eine *TImage*), dann muss die Eigenschaft *Transparent* auf *true* gesetzt werden.



8.4 TDBEdit

Die Komponente *TDBEdit* dient zur Anzeige und zur Eingabe von Daten. Sie ähnelt der Komponente

- DataSource, DataField (Eigenschaften, veröffentlicht)

```
property DataSource: TDataSource;
property DataField: string;
```

Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld.

- OnChange (Ereignis)

  ```
  property OnChange(Sender: TObject);
  ```

  Das Ereignis *OnChange* wird ausgelöst, wenn der Text einer *TDBEdit*-Instanz geändert wird. *OnChange* tritt auch dann auf, wenn der Text durch Zugriff auf die Eigenschaft *Text* geändert oder wenn der Eigenschaft *PasswordChar* ein anderer Wert zugewiesen wird. *OnChange* tritt nicht auf, wenn die Eigenschaft *CharCase* geändert wird.

- CharCase (Eigenschaft, veröffentlicht)

  ```
  property CharCase: (ecNormal, ecUpperCase, ecLowerCase);
  ```

  Mit der Eigenschaft *CharCase* kann bestimmt werden, ob der Text in Groß- und Kleinschreibung (*ecNormal*), nur in Klein- (*ecLowerCase*) oder nur in Großbuchstaben (*ecUpperCase*) gespeichert wird. *CharCase* wirkt sich nicht nur auf die Anzeige, sondern auch auf den Inhalt der Eigenschaft *Text* aus.

- PasswordChar (Eigenschaft, veröffentlicht)

  ```
  property PasswordChar: Char default #0;
  ```

  Soll mit einer *TDBEdit*-Komponente ein Passwort abgefragt werden, dann soll dies in der Regel nicht auf dem Bildschirm dargestellt werden. Dies kann dadurch erreicht werden, dass der Eigenschaft *PasswordChar* der darzustellende Buchstabe zugewiesen wird. *PasswordChar* wirkt sich nicht auf die Eigenschaft *Text* aus.

- MaxLength (Eigenschaft, veröffentlicht)

  ```
  property MaxLength: Integer default 0;
  ```

  Mit der Eigenschaft *MaxLength* wird die Zahl der Zeichen begrenzt, die in *TDBEdit* eingegeben werden können. *MaxLength* begrenzt nur die Eingabe über die Tastatur – der Eigenschaft *Text* können längere Zeichenfolgen zugewiesen werden, die dann auch angezeigt werden.

  Soll die Eigenschaft *Text* unter Berücksichtigung von *MaxLength* gesetzt werden, dann kann die Eigenschaft *SelText* verwendet werden, siehe Beispiel dort.

- AutoSize (Eigenschaft, veröffentlicht)

  ```
  property AutoSize: Boolean default true;
  ```

  Wird *AutoSize* auf *true* belassen, dann wird die Höhe einer *TDBEdit*-Instanz automatisch der Schriftgröße und den Rahmeneinstellungen angepasst.

- SelText (Eigenschaft, öffentlich)

```
property SelText: string;
```

Mit Hilfe der Eigenschaft *SelText* kann der selektierte Text einer *TDBEdit*-Instanz gelesen oder geändert werden. Ist kein Text selektiert, dann gibt ein Lesezugriff auf *SelText* einen leeren String zurück, während ein Schreibzugriff den Text an der Cursor-Position einfügt.

- SelStart, SelLength (Eigenschaften, öffentlich)

```
property SelStart: Integer;
property SelLength: Integer;
```

Mit Hilfe der Eigenschaften können der Beginn und die Länge des selektierten Textes bestimmt werden.

Mit Hilfe der folgenden Routine würde man einen Text vor den bereits enthaltenen Text setzen:

```
with DBEdit1 do
begin
  SelStart := 0;
  SelLength := 0;
  SelText := '1234567';
end;
```

- CutToClipboard, CopyToClipboard, PasteFromClipboard (Methoden)

```
procedure CutToClipboard;
procedure CopyToClipboard;
procedure PasteFromClipboard;
```

Mit Hilfe dieser Methoden kann man den selektierten Text in die Zwischenablage ausschneiden (*CutToClipboard*) beziehungsweise kopieren (*CopyToClipboard*) oder den selektierten Text durch den Inhalt der Zwischenablage ersetzen (*PasteFromClipboard*). Ist kein Text selektiert, dann wird der Inhalt der Zwischenablage an der Cursor-Position eingefügt.

Sind die Daten, die in der Zwischenablage enthalten sind, in keinem Textformat, dann wird der selektierte Text durch einen leeren String ersetzt. Bei Einfügen über die Zwischenablage wird die Eigenschaft *MaxLength* berücksichtigt.

Die Tastenkombinationen Strg+X, Strg+C und Strg+V lösen automatisch die entsprechenden Zwischenablage-Aktionen aus.

- ReadOnly (Eigenschaft, veröffentlicht)

```
property ReadOnly: Boolean default false;
```

  Wenn Sie (vorübergehend) verhindern wollen, dass der Benutzer den Inhalt von *TDBEdit* ändert, dann setzen Sie *ReadOnly* auf *true*.

- Clear, ClearSelection (Methode)

```
procedure Clear;
procedure ClearSelection;
```

  Die Methode *Clear* löscht den gesamten Inhalt der Eigenschaft *Text*, während *ClearSelection* nur den selektierten Text löscht.

  Mit der folgenden Routine wird das jeweils zweite Zeichen des Textes von *DBEdit1* entfernt.

```
with DBEdit1 do
begin
  SelStart := 1;
  SelLength := 1;
  ClearSelection;
end;
```

- SelectAll (Methode)

```
procedure SelectAll;
```

  Mit *SelectAll* wird der ganze Text markiert.

- CanUndo (Eigenschaft, öffentlich, nur Lesen), Undo, ClearUndo (Methoden)

```
property CanUndo: Boolean;
procedure Undo;
procedure ClearUndo;
```

  Mit *Undo* kann eine Eingabe rückgängig gemacht werden. Mit *CanUndo* kann festgestellt werden, ob es Eingaben gibt, die rückgängig gemacht werden können, und ob *Undo* somit eine Funktion hat.

  Die Methode *ClearUndo* nimmt laut Online-Hilfe alle im Rückgängig-Puffer gespeicherten Änderungen zurück. Ein solches Verhalten konnte jedoch – zumindest auf meinem Rechner – nicht festgestellt werden.

8.5 TDBMemo

Die Komponente *TDBMemo* dient zur Eingabe und Anzeige mehrzeiliger Texte, sie ähnelt der Komponente *TMemo*. *TDBMemo* ist von *TWinControl* abgeleitet.

Für die Darstellung von formatiertem Text verwenden Sie die Komponente *TDBRichEdit*.

- DataSource, DataField (Eigenschaften, veröffentlicht)

  ```
  property DataSource: TDataSource;
  property DataField: string;
  ```

  Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld. Dabei muss der Feldtyp nicht zwingend *TMemoField* sein, er kann beispielsweise auch *TStringField* oder *TIntegerField* sein.

- AutoDisplay (Eigenschaft, veröffentlicht), LoadMemo (Methode)

  ```
  property AutoDisplay: Boolean default true;
  procedure LoadMemo;
  ```

  Per Voreinstellung werden die Inhalte von Memo-Feldern in der Komponente automatisch angezeigt, dies verhindert aber ein schnellstmögliches Scrollen durch die Datenmenge. Wenn es auf Geschwindigkeit ankommt, dann setzen Sie *AutoDisplay* auf *false* und laden die Memos, die angezeigt werden sollen, mit *LoadMemo* manuell.

- Lines (Eigenschaft, veröffentlicht)

  ```
  property Lines: TStrings;
  ```

  Mit Hilfe der Eigenschaft *Lines* kann zeilenweise auf den Inhalt einer *TDBMemo*-Instanz zugegriffen werden.

  ```
  with DBMemo1.Lines do
  begin
    Clear;
    Add('Dies');
    Add('ist');
    Add('ein Dreizeiler');
  end;
  ```

- Text (Eigenschaft, öffentlich)

```
property Text: string;
```

Mit Hilfe der Eigenschaft *Text* kann auf den kompletten Inhalt einer *TDBMemo*-Instanz zugegriffen werden.

```
DBMemo1.Text := 'Dies' + chr(13) + chr(10)
  + 'ist' + chr(13) + chr(10)
  + 'ein Dreizeiler';
```

- ScrollBars (Eigenschaft, veröffentlicht)

```
property ScrollBars: (ssNone, ssHorizontal, ssVertical, ssBoth)
  default ssNone;
```

Mit der Eigenschaft *ScrollBar* wird spezifiziert, ob die *TDBMemo*-Instanz mit Scroll-Balken ausgestattet wird. *ScrollBar* kann vier verschiedene Werte annehmen.

- WantReturns (Eigenschaft, veröffentlicht)

```
property WantReturns: Boolean default true;
```

Hat *WantReturns* den Wert *true*, dann werden mit der EINGABE-Taste Zeilenumbrüche in den *TDBMemo*-Text eingefügt. Soll die EINGABE-Taste auf das Formular wirken – beispielsweise um einen Dialog mit OK abzuschließen –, dann muss *WantReturns* auf *false* gesetzt werden; Zeilenumbrüche können dann mit STRG+EINGABE eingefügt werden.

- WantTabs (Eigenschaft, veröffentlicht)

```
property WantTabs: Boolean default false;
```

Per Voreinstellung (*WantTabs = false*) wird mit der TAB-Taste der Fokus zwischen den einzelnen *TWinControl*-Nachfolgern weitergeschaltet. Sollen jedoch in einer *TDBMemo*-Instanz Tabulatorzeichen eingefügt werden, dann ist *WantTabs* auf *true* zu setzen; eine Möglichkeit, den Fokus über die Tastatur weiterzuschalten, besteht dann aber nicht mehr.

- WordWrap (Eigenschaft, veröffentlicht)

```
property WordWrap: Boolean default true;
```

Hat *WordWrap* (gemäß Voreinstellung) den Wert *true*, dann werden Zeichen, die über das Zeilenende hinausgehen, in die nächste Zeile umgebrochen. *TDBMemo* führt den Umbruch – wenn möglich – nach Leerzeichen durch, nicht jedoch nach Satzzeichen. Soll kein automatischer Umbruch durchgeführt werden, ist *WordWrap* auf *false* zu setzen.

TDBMemo ist wie *TDBEdit* von *TCustomEdit* abgeleitet und implementiert einige Eigenschaften und Ereignisse, die bereits bei *TDBEdit* (Kapitel 4.4) beschrieben wurden. Im Einzelnen handelt es sich dabei um:

- MaxLength (Eigenschaft, veröffentlicht)
- ReadOnly (Eigenschaft, veröffentlicht)
- SelLength, SelStart, SelText (Eigenschaften, öffentlich)
- CutToClipboard, CopyToClipboard, PasteFromClipboard (Methoden)
- Clear, ClearSelection (Methode)
- SelectAll (Methode)
- CanUndo (Eigenschaft, öffentlich, nur Lesen), Undo, ClearUndo (Methoden)
- OnChange (Ereignis)

8.6 TDBImage

Die Komponente *TDBImage* dient zur Eingabe und Anzeige mehrzeiliger Texte, sie ähnelt der Komponente *TImage*. *TDBImage* ist von *TControl* abgeleitet.

- DataSource, DataField (Eigenschaften, veröffentlicht)

```
property DataSource: TDataSource;
property DataField: string;
```

  Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld.

- AutoDisplay (Eigenschaft, veröffentlicht), LoadPicture (Methode)

```
property AutoDisplay: Boolean default true;
procedure LoadPicture;
```

  Per Voreinstellung werden die Inhalte von Grafik-Feldern in der Komponente automatisch angezeigt, dies verhindert aber ein schnellstmögliches Scrollen durch die Datenmenge. Wenn es auf Geschwindigkeit ankommt, dann setzen Sie *AutoDisplay* auf *false* und laden die anzuzeigenden Bilder mit *LoadPicture* manuell.

- Center, Stretch (Eigenschaften, veröffentlicht)

```
property Center: Boolean default true;
property Stretch: Boolean default false;
```

Hat *Center* den Wert *true*, dann wird das Bild in der Komponentenmitte zentriert, anderenfalls wird es in die linke obere Komponentenecke geschoben.

Wird *Stretch* auf *true* gesetzt, dann wird das Bild auf die Komponentenmaße vergrößert oder verkleinert. Dabei verändert sich meist das Höhen-Breiten-Verhältnis.

- Picture (Eigenschaft, öffentlich)

```
property Picture: TPicture;
```

Das von der Komponente angezeigte Bild kann mit *Picture* referenziert werden. Verwenden Sie die *TPicture*-Eigenschaft *Canvas*, um das Bild zu ändern.

```
with DBImage1.Picture.Canvas do
begin
  ...
end;
```

8.7 TDBListBox

Die Komponente *TDBListBox* ist eine datensensitive Listbox, sie ähnelt der Komponente *TListBox*. *TDBListBox* ist von *TWinControl* abgeleitet.

- DataSource, DataField (Eigenschaften, veröffentlicht)

```
property DataSource: TDataSource;
property DataField: string;
```

Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld.

- Items (Eigenschaft, veröffentlicht)

```
property Items: TStrings;
```

Die Listeneinträge der Komponente werden der Eigenschaft *Items* zugewiesen. Sollen die Listeneinträge aus einer Datenmengenkomponente bezogen werden, so ist die Komponente *TDBLookupListBox* zu verwenden.

Gleicht beim Browsen durch die Datenmenge der betreffende Feldinhalt einem der Listeneinträge, so wird dieser Eintrag markiert dargestellt. Ist der Feldwert in der Liste nicht vorhanden, so wird kein Eintrag markiert.

- ItemIndex (Eigenschaft, öffentlich)

```
property ItemIndex: Integer;
```

Mit *ItemIndex* wird bestimmt, welcher Eintrag gewählt ist, wobei die Zählung bei null beginnt. Hat *ItemIndex* den Wert -1, so ist kein Eintrag ausgewählt.

Den Text des aktuell ausgewählten Eintrags ermittelt man wie folgt:

```
if DBListBox1.ItemIndex >= 0
  then s := DBListBox1.Items[DBListBox1.ItemIndex];
```

- Selected (Eigenschaft, öffentlich)

```
property Selected[Index: Integer]: Boolean;
```

Mit Hilfe der Eigenschaft *Selected* wird abgefragt, ob ein bestimmter Eintrag in der Liste markiert ist. Die Zählung der Einträge beginnt dabei bei null.

- IntegralHeight (Eigenschaft, veröffentlicht)

```
property IntegralHeight:boolean default false;
```

Je nach Höhe der List-Box und der Größe der verwendeten Schrift wird der letzte Eintrag einer ListBox meist nicht vollständig angezeigt, was etwas unschön aussieht. Indem man die Eigenschaft *IntegralHeight* auf *true* setzt, kann man die Höhe der List-Box so reduzieren, dass sie nach dem letzten noch vollständig angezeigten Eintrag endet.

- TopIndex (Eigenschaft, öffentlich)

```
property TopIndex: Integer;
```

Hat die Eigenschaft *Items* mehr Einträge als die List-Box Zeilen, dann kann nur eine Teilmenge davon angezeigt werden – es wird dann automatisch ein Scroll-Balken eingeblendet. Mit *TopIndex* kann der Index des Eintrags bestimmt werden, der in der obersten Zeile angezeigt wird.

Selbst gezeichnete List-Boxen

Im Regelfall wird die Anzeige der Einträge von der List-Box automatisch vorgenommen. Es besteht jedoch auch die Möglichkeit, die Darstellung selbst zu übernehmen – das ist beispielsweise dann erforderlich, wenn zusätzlich zu den Einträgen Bilder angezeigt werden sollen.

- Style (Eigenschaft, veröffentlicht)

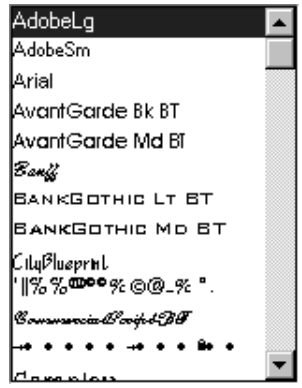
```
property Style: (lbStandard, lbOwnerDrawFixed, lbOwnerDrawVariable);
  default lbStandard;
```

Wenn Sie die Darstellung der Einträge in der List-Box selbst übernehmen möchten, müssen Sie die Eigenschaft *Style* auf den Wert *lbOwnerDrawFixed* oder *lbOwnerDrawVariable* setzen. Während bei *lbOwnerDrawFixed* die Höhe aller Einträge gleich hoch ist, kann sie bei *lbOwnerDrawVariable* von Eintrag zu Eintrag variieren.

- OnDrawItem (Ereignis)

```
property OnDrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
```

Das Ereignis *OnDrawItem* tritt immer dann auf, wenn ein Eintrag neu gezeichnet werden muss. Als Parameter werden der Ereignisbehandlungsroutine der Index des Eintrages und dessen Position in der List-Box übergeben. Die Änderung der Darstellung beim Markieren der Einträge führt *TListBox* automatisch durch, so dass der Parameter *State* nur in Ausnahmefällen benötigt wird.



Die Abbildung zeigt eine List-Box, welche die im System installierten Fonts in entsprechender Darstellung anzeigt. Die dazugehörende *OnDrawItem*-Ereignisbehandlungsroutine lautet wie folgt:

```
procedure TForm1.DBListBox2DrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
begin
  DBListBox2.Canvas.Font.Name := DBListBox2.Items[Index];
  DBListBox2.Canvas.TextRect
    (Rect, Rect.Left, Rect.Top, DBListBox2.Items[Index]);
end;
```

Um die im System installierten Schriftarten der Eigenschaft *Items* zuzuweisen, geht man wie folgt vor:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DBListBox2.Items.Assign(Screen.Fonts);
  DBListBox2.ItemIndex := 0;
end;
```

- OnMeasureItem (Ereignis)

```
property OnMeasureItem(Control: TWinControl; Index: Integer;
  var Height: Integer);
```

Ist die Eigenschaft *Style* auf den Wert *lbOwnerDrawVariable* gesetzt, dann wird vor dem Ereignis *OnDrawItem* das Ereignis *OnMeasureItem* aufgerufen, dessen Variablen-Parameter *Height* die Höhe des folgenden Eintrags zugewiesen wird.

8.8 TDBComboBox

Die Komponente *TDBComboBox* ist eine datensensitive List-Box, sie ähnelt der Komponente *TComboBox*. *TDBComboBox* ist von *TWinControl* abgeleitet.

- DataSource, DataField (Eigenschaften, veröffentlicht)

```
property DataSource: TDataSource;
property DataField: string;
```

Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld.

- Items (Eigenschaft, veröffentlicht)

```
property Items: TStrings;
```

Die Listeneinträge der Komponente werden der Eigenschaft *Items* zugewiesen. Sollen die Listeneinträge aus einer Datenmengenkomponente bezogen werden, so ist die Komponente *TDBLookupComboBox* zu verwenden.

Gleicht beim Browsen durch die Datenmenge der betreffende Feldinhalt einem der Listeneinträge, so wird dieser Eintrag dargestellt. Ist der Feldwert in der Liste nicht vorhanden, so wird entweder der Feldwert (*Style* = *csDropDown*) oder ein leeres Feld (*Style* = *csDropDownList*) angezeigt.

- Style (Eigenschaft, veröffentlicht)

```
property Style: (csDropDown, csSimple, csDropDownList,
  csOwnerDrawFixed, csOwnerDrawVariable); default csDropDown;
```

  Die Eigenschaft *Style* bestimmt das Aussehen der ComboBox. *Style* kann die folgenden Werte annehmen:

  - *csDropDown*: Der angezeigte Text kann frei editiert werden, es kann aber auch ein Eintrag aus der Liste ausgewählt werden.
  - *csDropDownList*: Es können ausschließlich Einträge aus der Liste ausgewählt werden.
  - *csSimple*: Die Komponente verhält sich wie eine *TEdit*-Instanz, Einträge aus der Liste können somit nicht gewählt werden.
  - *csOwnerDrawFixed*: Die Einträge in der Liste werden vom Anwender selbst gezeichnet, die Höhe der Einträge ist konstant. Siehe auch *TDBListBox*.
  - *csOwnerDrawVariable*: Die Einträge in der Liste werden vom Anwender selbst gezeichnet, die Höhe der Einträge kann für jeden Eintrag einzeln festgelegt werden. Siehe auch *TDBListBox*.

- DropDownCount (Eigenschaft, veröffentlicht)

```
property DropDownCount: integer default 8;
```

  Mit der Eigenschaft *DropDownCount* wird bestimmt, wie viele Einträge in der heruntergeklappten Liste maximal angezeigt werden.

- OnChange, OnDropDown (Ereignisse)

```
property OnChange(Sender: TObject);
property OnDropDown(Sender: TObject);
```

  Das Ereignis *OnChange* tritt auf, wenn sich der angezeigte Text ändert – sowohl bei der manuellen Eingabe eines Textes als auch bei der Auswahl eines Eintrages. Das Ereignis *OnDropDown* tritt auf, wenn die Auswahlliste nach unten geklappt wird.

- OnDrawItem, OnMeasureItem (Ereignisse)

```
property OnDrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
property OnMeasureItem(Control: TWinControl; Index: Integer;
  var Height: Integer);
```

  Eine ausführliche Beschreibung dieser Ereignisse ist bei *TListBox* zu finden.

8.9 TDBCheckBox

Die Komponente *TDBCheckBox* implementiert eine datensensitive CheckBox. TDBCheckBox ist von *TWinControl* abgeleitet.

- DataSource, DataField (Eigenschaften, veröffentlicht)

  ```
  property DataSource: TDataSource;
  property DataField: string;
  ```

  Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld.

- ValueChecked, ValueUnchecked (Eigenschaften, veröffentlicht)

  ```
  property ValueChecked: string default 'Wahr';
  property ValueUnchecked: string default 'Falsch';
  ```

  Für den Fall, dass eine *TDBCheckBox*-Instanz nicht mit einer *TBooleanField*-Instanz verbunden ist, kann hier eingegeben werden, bei welchen Werten die CheckBox markiert und bei welchen Werten sie nicht markiert dargestellt werden soll. Bei allen anderen Werten wird sie »grau« dargestellt.

  Sollen diesen Eigenschaften mehrere Werte zugewiesen werden, so sind diese mit Semikola zu trennen:

  ```
  DBCheckBox1.ValueChecked := 'Wahr;Ja;Ein';
  ```

- AllowGrayed (Eigenschaft, veröffentlicht)

  ```
  property AllowGrayed: Boolean default False;
  ```

  Wird *AllowGrayed* auf true gesetzt, dann kann mit der Maus oder der Tastatur (LEERTASTE) die Eigenschaft *State* auf *cbGrayed* gesetzt werden. Hat *AllowGrayed* den Wert *false*, kann dies nur per Zugriff auf die Eigenschaft *State* oder bei Feldwerten ungleich *ValueChecked* und *ValueUnchecked* erfolgen.

- Checked, State (Eigenschaften, öffentlich)

  ```
  property Checked: Boolean;
  property State: (cbUnchecked, cbChecked, cbGrayed);
  ```

  Die Eigenschaft *Checked* hat den Wert *true*, wenn *State* den Wert *cbChecked* hat. Wird *Checked* auf *false* gesetzt, hat *State* anschließend auch dann den Wert *cbUnchecked*, wenn *State* davor den Wert *cbGrayed* gehabt hat.

8.10 TDBRadioGroup

Die Komponente *TDBRadioGroup* ähnelt *TDBListBox*, stellt die Einträge aber als Radio-Button statt als Liste dar. *TDBRadioGroup* ist von *TWinControl* abgeleitet.

- DataSource, DataField (Eigenschaften, veröffentlicht)

  ```
  property DataSource: TDataSource;
  property DataField: string;
  ```

  Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld.

- Items, Values (Eigenschaften, veröffentlicht)

  ```
  property Items: TStrings;
  property Values: TStrings;
  ```

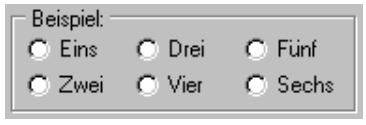
  Für gewöhnlich werden die Einträge der Eigenschaft *Items* als Wert für die Anzeige und als Wert für die Datenbank verwendet.

  Möchte man andere Werte anzeigen, als in der Datenbank gespeichert werden, dann weist man die Werte, die in der Datenbank gespeichert werden sollen, der Eigenschaft *Values* zu. Angezeigt wird dann jeweils der Wert von *Items*, der denselben Index hat wie der *Values*-Wert, dessen Wert der Datenbank-Feldwert hat.

- Columns (Eigenschaft, veröffentlicht)

  ```
  property Columns: Integer default 1;
  ```

  Die Einträge der Radio-Button-Gruppe können in mehreren Spalten angezeigt werden. Die Anzahl der Spalten wird mit *Columns* spezifiziert.

- ItemIndex (Eigenschaft, öffentlich)

  ```
  property ItemIndex: Integer;
  ```

  Mit der Eigenschaft *ItemIndex* kann ermittelt werden, welcher der Einträge momentan selektiert ist. Es ist auch möglich, den gewählten Eintrag mit Hilfe von *ItemIndex* zu ändern.

8.11 TDBLookupListBox

Die Komponente *TDBLookupListBox* dient zur Auswahl von Werten aus einer Liste. Im Gegensatz zu *TDBListBox* wird diese Liste nicht einer Eigenschaft *Items* zugewiesen, sondern von einer Datenquellenkomponente bezogen.

- DataSource, DataField (Eigenschaften, veröffentlicht)

```
property DataSource: TDataSource;
property DataField: string;
```

  Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld.

- ListSource, KeyField, ListField (Eigenschaften, veröffentlicht)

```
property ListSource: TDataSource;
property KeyField: string;
property ListField: string;
```

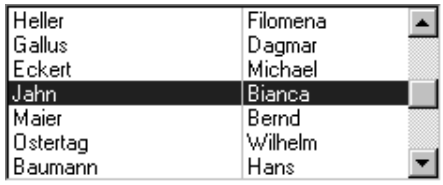
  Mit *ListSource* wird angegeben, von welcher Datenquellenkomponente die Listeneinträge übernommen werden. Mit *KeyFields* wird spezifiziert, welche Felder der Lookup-Datenmenge in die über *DataSource* verbundene Datenmenge eingefügt werden, wenn ein Listeneintrag ausgewählt wird.

  Nehmen wir an, mit der *TDBLookupListBox*-Instanz sollen Kunden für eine andere Tabelle ausgewählt werden. In diesem Fall würde man in der anderen Tabelle nur die Kundennummer speichern, als *KeyField* würde man dann diese Kundennummer verwenden.

  Soll auch nur die Kundennummer angezeigt werden, dann würde man die Eigenschaft *ListField* leer lassen. Man kann aber dort auch den Spaltenbezeichner der Kundennamen eingeben. Es werden dann die Kundennamen angezeigt, aber die Kundennummern in die Tabelle eingefügt.

  Es ist auch möglich, der Eigenschaft *ListField* mehrere Spaltenbezeichner zuzuweisen – es werden dann auch mehrere Spalten angezeigt. Die Spaltenbezeichner sind durch Semikola zu trennen.

```
DBLookupListBox1.ListField := 'Nachname;Vorname';
```



8.12 TDBLookupComboBox

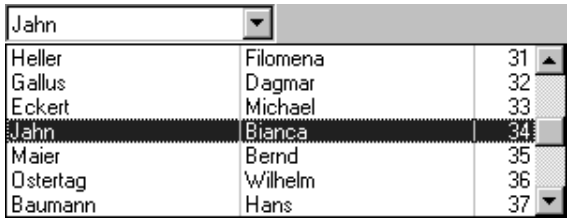
Die Komponente *TDBLookupComboBox* tut dasselbe wie *TDBLookupListBox* (Kapitel 8.11), mit dem Unterschied, dass anstelle einer List-Box eine ComboBox zur Anzeige verwendet wird. *TDBLookupComboBox* ist von *TWinControl* abgeleitet.

Die folgenden Eigenschaften wurden bereits bei *TDBLookupListBox* beschrieben:

- DataSource, DataField (Eigenschaften, veröffentlicht)
- ListSource, KeyField, ListField (Eigenschaften, veröffentlicht)
- DropDownWidth (Eigenschaft, veröffentlicht)

```
property DropDownWidth: Integer default 0;
```

Hat *DropDownWidth* einen Wert ungleich null, dann wird damit die Breite der herunterklappbaren Liste eingestellt. Dies ist insbesondere dann hilfreich, wenn mehrere Spalten angezeigt werden sollen.



- DropDownRows (Eigenschaft, veröffentlicht)

```
property DropDownRows: Integer default 7;
```

Mit *DropDownRows* wird spezifiziert, wie viele Zeilen die heruntergeklappte Liste anzeigt.

- DropDownAlign (Eigenschaft, veröffentlicht)

```
property DropDownAlign: (daLeft, daRight, daCenter)
  default daLeft;
```

Mit *DropDownAlign* wird eingestellt, ob die heruntergeklappte Liste links-, rechtsbündig oder zentriert zur eigentlichen Komponente dargestellt wird. Diese Eigenschaft ist nur dann relevant, wenn *DropDownWidth* einen Wert null hat.

8.13 TDBRichEdit

Die Komponente *TDBRichEdit* dient zur Anzeige von formatiertem Text. Sie ähnelt der Komponente *TRichEdit*. *TDBRichEdit* ist von *TWinControl* abgeleitet.

- DataSource, DataField (Eigenschaften, veröffentlicht)

  ```
  property DataSource: TDataSource;
  property DataField: string;
  ```

  Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden, *DataField* spezifiziert das entsprechende Feld.

- AutoDisplay (Eigenschaft, veröffentlicht), LoadMemo (Methode)

  ```
  property AutoDisplay: Boolean default true;
  procedure LoadMemo;
  ```

  Per Voreinstellung werden die Inhalte von Memo-Feldern in der Komponente automatisch angezeigt, dies verhindert aber ein schnellstmögliches Scrollen durch die Datenmenge. Wenn es auf Geschwindigkeit ankommt, dann setzen Sie *AutoDisplay* auf *false* und laden die Memos, die angezeigt werden sollen, mit *LoadMemo* manuell.

- SelAttributes, Paragraph (Eigenschaften, öffentlich)

  ```
  property SelAttributes: TTextAttributes;
  property Paragraph: TParaAttributes;
  ```

  Mit *SelAttributes* werden die Formatierungen des momentan markierten Textes, mit *Paragraph* die Formatierungen des aktuellen Abschnitts ermittelt oder geändert. Die Typen *TTextAttributes* sowie *TParaAttributes* werden in eigenen Abschnitten besprochen.

- PlainText (Eigenschaft, veröffentlicht)

  ```
  property PlainText: Boolean default false;
  ```

  Wird *PlainText* auf *true* gesetzt, dann wird der Text als ANSI-Text geöffnet und gespeichert. Beim Speichern gehen vorhandene Formatierungsinformationen verloren. Wird eine RTF-Datei mit *PlainText = true* geöffnet, dann werden die Formatierungsinformationen angezeigt und nicht interpretiert.

- Print (Methode)

```
procedure Print(const Caption: string);
```

Mit der Methode *Print* wird der gesamte Text der *TDBRichEdit*-Instanz ausgedruckt. Dabei werden die Randeinstellungen des Druckertreibers verwendet. Der mit *Caption* übergebene Parameter wird im Druckmanager angezeigt.

- OnSelectionChange (Ereignis)

```
property OnSelectionChange(Sender: TObject);
```

Das Ereignis *OnSelectionChange* tritt auf, wenn ein anderer Text markiert wird, aber auch dann, wenn der Eingabe-Cursor verschoben wird. Verwenden Sie die *OnSelectionChange*-Ereignisbehandlungsroutine, um die aktuellen Formatierungsinformationen anzuzeigen.

- FindText (Methode)

```
TSearchTypes = set of (stWholeWord, stMatchCase);

function FindText(const SearchStr: string;
  StartPos, Length: Integer; Options: TSearchTypes): Integer;
```

Wegen der Formatierungsanweisungen im Text würden normale Suchroutinen Positionen zurückgeben, die nicht mit den *SelStart*-/*SelLength*-Angaben übereinstimmen. Um brauchbare Ergebnisse zu erhalten, müssen Sie zum Suchen von Text die Funktion *FindText* verwenden.

Das folgende Listing zeigt, wie man mittels einer *TFindDialog*-Komponente Text in einer *TDBRichEdit*-Instanz findet.

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
  i: integer;
  SearchTypes: TSearchTypes;
begin
  SearchTypes := [];
  if frMatchCase in FindDialog1.Options
    then SearchTypes := SearchTypes + [stMatchCase];
  if frWholeWord in FindDialog1.Options
    then SearchTypes := SearchTypes + [stWholeWord];
  i := DBRichEdit1.FindText(FindDialog1.FindText,
    RichEdit1.SelStart + 1, Length(RichEdit1.Lines.Text),
    SearchTypes);
```

```
  if i > -1 then
  begin
    DBRichEdit1.SelStart := i;
    DBRichEdit1.SelLength := Length(FindDialog1.FindText);
  end
  else ShowMessage('Nichts gefunden');
end; {procedure TForm1.FindDialog1Find}
```

TTextAttributes

Mit *TTextAttributes* werden die zeichenbasierten Formatierungsinformationen zusammengefasst.

- Name, Color (Eigenschaften, öffentlich)

  ```
  property Name: string;
  property Color: TColor;
  ```

  Mit *Name* und *Color* werden der Zeichensatz und dessen Farbe eingestellt.

- Height, Size (Eigenschaften, öffentlich)

  ```
  property Height: Integer;
  property Size: Integer;
  ```

  Mit *Height* wird die Höhe in Pixel, mit *Size* in Punkt angegeben. Das Setzen einer Eigenschaft überschreibt die jeweils andere.

- Style (Eigenschaft, öffentlich)

  ```
  property Style:
    set of (fsBold, fsItalic, fsUnderline, fsStrikeOut);
  ```

  In der Mengeneigenschaft *Style* sind die Formatierungsattribute fett (*fsBold*), kursiv (*fsItalic*), unterstrichen (*fsUnderline*) und durchgestrichen (*fsStrikeOut*) zusammengefasst.

  Wie diese Informationen auf einzelne SpeedButtons aufgeteilt werden, zeigen die folgenden beiden Prozeduren:

  ```
  procedure TForm1.DBRichEdit1SelectionChange(Sender: TObject);
  begin
    with RichEdit1.SelAttributes do
    begin
      SpeedButton1.Down := (fsBold in Style);
      SpeedButton2.Down := (fsItalic in Style);
      SpeedButton3.Down := (fsUnderline in Style);
  ```

```
    SpeedButton4.Down := (fsStrikeOut in Style);
  end; {with RichEdit1.SelAttributes do}
end; {procedure TForm1.DBRichEdit1SelectionChange}

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  with DBRichEdit1.SelAttributes do
  begin
    Style := [];
    if SpeedButton1.Down then Style := Style + [fsBold];
    if SpeedButton2.Down then Style := Style + [fsItalic];
    if SpeedButton3.Down then Style := Style + [fsUnderline];
    if SpeedButton4.Down then Style := Style + [fsStrikeOut];
  end; {with Form2.DBRichEdit1.SelAttributes do}
end; {procedure TForm1.SpeedButton1Click}
```

Die Prozedur *SpeedButton1Click* muss allen vier SpeedButtons als *OnClick*-Ereignisbehandlungsroutine zugewiesen werden.

Etwas kürzer, dafür aber auch unübersichtlicher, folgende Anweisung:

```
DBRichEdit1.SelAttributes.Style := TFontStyles(Byte(
  (integer(SpeedButton1.Down) + 2*integer(SpeedButton2.Down) +
  4*integer(SpeedButton3.Down) + 8*integer(SpeedButton4.Down))));
```

- ConsistentAttributes (Eigenschaft, öffentlich, nur Lesen)

```
property ConsistentAttributes: set of (caBold, caColor, caFace,
  caItalic, caSize, caStrikeout, caUnderline, caProtected);
```

Mit *ConsistentAttributes* kann ermittelt werden, welche Formatierungsattribute innerhalb des markierten Textes konsistent sind.

Nehmen wir an, Sie haben den Text *eins zwei*. Davon ist das Wort *zwei* unterstrichen, das Wort *eins* nicht. Nun markieren Sie den gesamten Text. Dann ist das Formatierungsattribut *Unterstrichen* innerhalb des markierten Textes nicht konsistent, weil ein Teil des Textes unterstrichen ist, der andere nicht. Das Element *caUnderline* wäre somit nicht in der Menge *ConsistentAttributes*.

TParaAttributes

Mit *TParaAttributes* werden die absatzbasierten Formatierungsinformationen zusammengefasst.

- FirstIndent, LeftIndent, RichtIndent (Eigenschaften, öffentlich)

```
property FirstIndent: Longint;
property LeftIndent: Longint;
property RightIndent: Longint;
```

Mit *RightIndent* wird der rechte Rand eingestellt. *FirstIndent* ist der linke Rand der ersten Zeile. *LeftIndent* ist der linke Rand aller weiteren Zeilen relativ zur ersten Zeile.

Die hier verwendete Funktionalität für *FirstIndent* und *RightIndent* ist eher ungebräuchlich. Häufiger kommt es vor, dass man den linken Rand direkt einstellt und zusätzlich einen positiven oder negativen Einzug der ersten Zeile relativ zum linken Rand festlegt. Die folgenden Listings zeigen, wie man eine entsprechende Umrechnung vornehmen würde:

```
procedure RichEdit1SelectionChange(Sender: TObject);
begin
  with RichEdit1.Paragraph do
  begin
    ComboBox6.Text := IntToStr(FirstIndent + LeftIndent);
    ComboBox7.Text := IntToStr(- LeftIndent);
    ComboBox8.Text := IntToStr(RightIndent);
  end;
end; {procedure TForm1.RichEdit1SelectionChange}

procedure TForm1.ComboBox6Change(Sender: TObject);
var
  links, erster, rechts, f, f1, f2: integer;
begin
  val(ComboBox6.Text, links, f1);
  val(ComboBox7.Text, erster, f2);
  val(ComboBox8.Text, rechts, f);
  if (f1 = 0) and (f2 = 0) then
  begin
    RichEdit1.Paragraph.LeftIndent := - erster;
    RichEdit1.Paragraph.FirstIndent := links + erster;
  end;
  if f = 0
    then  RichEdit1.Paragraph.RightIndent := rechts;
end; {procedure TForm1.ComboBox6Change}
```

Die Verwendung von *val* statt *StrToInt* vermeidet das Anhalten bei Exceptions, wenn versehentlich ein falsches Zeichen oder (häufiger) ein leerer String eingegeben wird.

- Alignment (Eigenschaft, öffentlich)

  ```
  property Alignment = (taLeftJustify, taRightJustify, taCenter);
  ```

  Mit Alignment wird festgelegt, ob der Text im Absatz linksbündig, rechtsbündig oder zentriert ausgerichtet wird.

- Numbering (Eigenschaft, öffentlich)

  ```
  property Numbering: (nsNone, nsBullet);
  ```

  Um kleine Punkte am Anfang jeden Absatzes anzuzeigen, setzen Sie *Numbering* auf *nsBullet*.

- Tab (Eigenschaft, öffentlich), TabCount (Eigenschaft, öffentlich, nur Lesen)

  ```
  property Tab[Index: Byte]: Longint;
  property TabCount: Integer;
  ```

  Mit der Array-Eigenschaft *Tab* können Tabulatoren gesetzt werden, die Maßeinheit ist Pixel. Mit *TabCount* kann die Anzahl der Tabulatoren ermittelt werden.

  Im Moment wirken die mit *Tab* eingefügten Tabulatoren zusätzlich zu denen, die bereits definiert sind, somit ist die Sache nicht so richtig verwendbar.

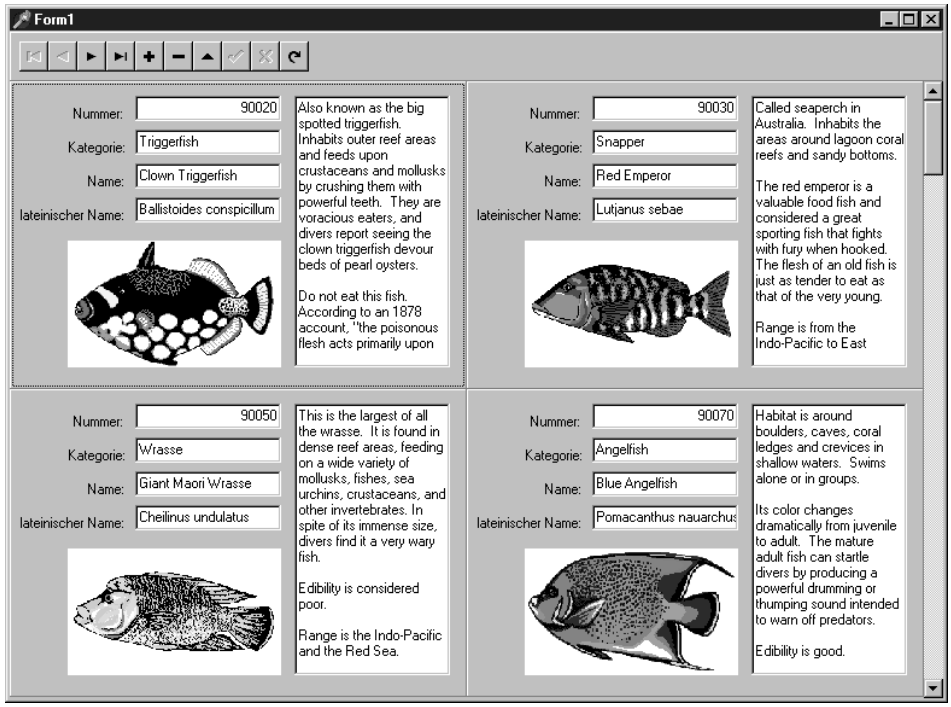
TDBRichEdit implementiert einige Eigenschaften, Methoden und Ereignisse, die bereits bei *TDBEdit* oder *TDBMemo* beschrieben wurden:

- Lines (Eigenschaft, TMemo)
- SelLength, SelStart, SelText (Eigenschaften, TEdit)
- ScrollBars (Eigenschaft, TMemo)
- WantReturns, WantTabs (Eigenschaften, TMemo)
- WordWrap (Eigenschaft, TMemo)
- MaxLength, ReadOnly (Eigenschaften, TEdit)
- SelectAll, Clear, ClearSelection (Methoden, TEdit)
- Undo, ClearUndo (Methoden, TEdit)
- CopyToClipboard, CutToClipboard, PasteFromClipboard (Methoden, TEdit)
- OnChange (Ereignis, TEdit)

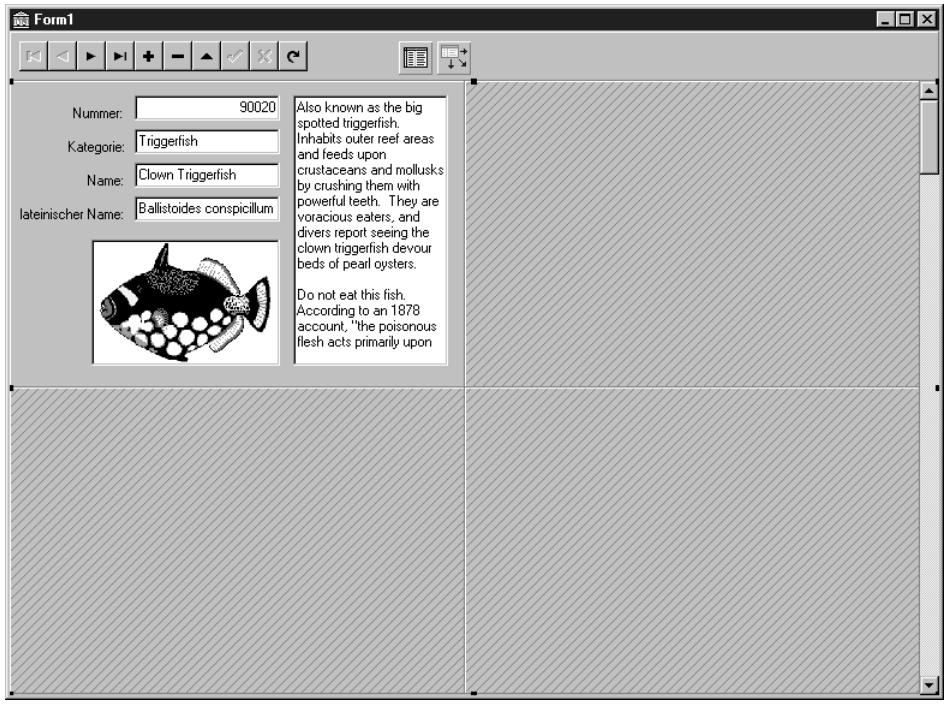
8.14 TDBCtrlGrid

Wird eine Datenmenge mit Komponenten wie *TDBEdit* oder *TDBImage* dargestellt, so ist jeweils immer nur ein einziger Datensatz sichtbar. Um mehrere Datensätze gleichzeitig zu betrachten, kann die Komponente *TDBGrid* eingesetzt werden. Allerdings ist der zur Verfügung stehende Platz zur Anzeige eines Datensatzes recht limitiert – das macht sich insbesondere bei der Anzeige von Bildern und Memos bemerkbar.

Einen Kompromiss bietet die Komponente *TDBCtrlGrid*. Hier werden auf verschiedenen Panels die Daten jeweils eines Datensatzes angezeigt. *TDBCtrlGrid* ist von *TWinControl* abgeleitet.



Zur Entwurfszeit werden datensensitive Dialogkomponenten auf das Design-Panel (links oben) gelegt. Beachten Sie dabei, dass Sie die Eigenschaft *DataSource* dieser Komponenten nicht ändern dürfen. Vielmehr muss die Eigenschaft *DataSource* der *TDBCtrlGrid*-Instanz gesetzt werden. Dieser Wert wird dann allen eingefügten datensensitiven Dialogkomponenten zugewiesen.



- DataSource (Eigenschaft, veröffentlicht)

  ```
  property DataSource: TDataSource;
  ```

  Mit der Eigenschaft *DataSource* wird spezifiziert, von welcher Datenquellenkomponente die Daten bezogen werden. Die Eigenschaft *DataSource* der eingefügten datensensitiven Dialogkomponenten wird automatisch auf diesen Wert gesetzt.

- ColCount, RowCount (Eigenschaften, veröffentlicht)

  ```
  property ColCount: Integer default 1;
  property RowCount: Integer default 3;
  ```

  Die Anzahl der Spalten wird mit *ColCount*, die der Reihen mit *RowCount* spezifiziert. Beachten Sie bitte, dass weder automatisch Scroll-Balken angezeigt werden, wenn der zur Verfügung stehende Platz für die Darstellung aller Komponenten nicht ausreicht, noch die Verwendung einer Scroll-Box erlaubt ist.

- AllowDelete, AllowInsert (Eigenschaften, veröffentlicht)

```
property AllowDelete: Boolean default true;
property AllowInsert: Boolean default true;
```

Hat *AllowDelete* den Wert *true*, dann kann mit Ctrl+Entf der aktuelle Datensatz gelöscht werden. Ebenso kann mit Ctrl+Einfg ein neuer Datensatz eingefügt werden, wenn *AllowInsert* den Wert *true* hat.

- DoKey (Methode)

```
procedure DoKey(Key: TDBCtrlGridKey);
```

Soll programmgesteuert ein anderes Panel fokussiert werden, dann kann dazu die Methode *DoKey* verwendet werden. Die möglichen Werte für den Typ *TDBCtrlGridKey* sind in der Online-Hilfe erläutert.

- ShowFocus, SelectedColor (Eigenschaften, veröffentlicht)

```
property ShowFocus: Boolean default true;
property SelectedColor: TColor default clBtnFace;
```

Hat *ShowFocus* den Wert *true*, dann wird das Panel des aktuellen Datensatzes mit einer punktierten Linie umgeben. Mit *SelectedColor* wird die Farbe des Panels vorgegeben, das den aktuellen Datensatz anzeigt.

- Orientation (Eigenschaft, veröffentlicht)

```
property Orientation: (goVertical, goHorizontal)
  default goVertical;
```

Mit *Orientation* wird spezifiziert, ob die Panels reihenweise (*goVertical*) oder spaltenweise (*goHorizontal*) angeordnet werden.

- OnPaintPanel (Ereignis)

```
property OnPaintPanel(DBCtrlGrid: TDBCtrlGrid; Index: Integer);
```

Das Ereignis *OnPaintPanel* wird aufgerufen, wenn eines der Panels neu gezeichnet werden muss.

In Delphi 2 war es noch nicht möglich, die Komponente *TDBImage* auf ein Control-Grid zu legen. Hier konnte man sich dann mit einer normalen *TImage*-Komponente behelfen, der ein Bild in einer *OnPaintPanel*-Ereignisbehandlungsroutine zugewiesen wurde.

```
procedure TForm1.DBCtrlGrid1PaintPanel(DBCtrlGrid: TDBCtrlGrid;
  Index: Integer);
begin
  Image1.Picture.Assign(Table1.FieldByName('Graphic'));
end;
```

9 Datenanalyse

In vielen Fällen möchte man die in der Datenbank enthaltenen Daten nicht nur anzeigen und ausdrucken, oft müssen die Daten auch mathematisch aufbereitet, zusammengefasst und in Beziehung zueinander gesetzt werden, um relevante Aussagen treffen zu können. Zu diesem Zweck gibt es in Delphi folgende Komponenten:

- Zur Anzeige von Daten als Geschäftsgrafik gibt es in der Professional- und der Enterprise-Version die Komponente *TChart* mit »Geschwistern« (*TDBChart*, *TQRChart*, *TDecisionGraph*).
- In der Enterprise-Version gibt es die Komponenten zur Datenanalyse.

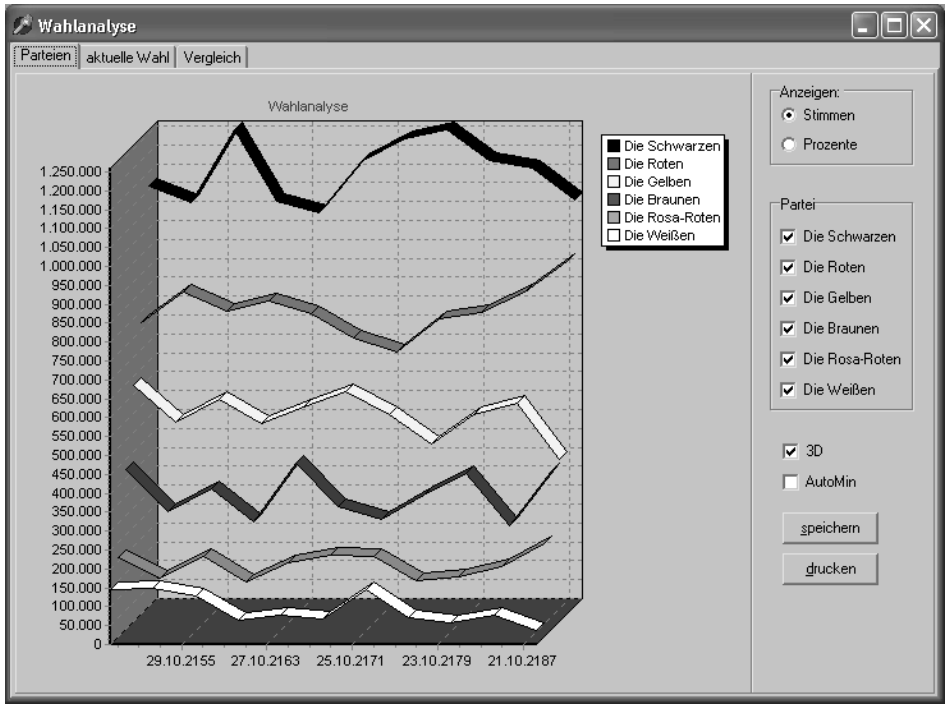
9.1 TChart und TDBChart

Die Komponenten *TChart* und *TDBChart* unterscheiden sich dadurch voneinander, dass bei *TDBChart* die Fähigkeit hinzugekommen ist, Daten direkt aus einer Datenquelle zu übernehmen. Da jedoch oftmals die Daten vorher speziell aufbereitet werden müssen, insbesondere dann, wenn sie nicht von einem Datenbank-Server stammen, der die Berechnung als STORED PROCEDURE implementieren kann, lässt sich diese spezielle Fähigkeit von *TDBChart* in der Regel nicht einsetzen.

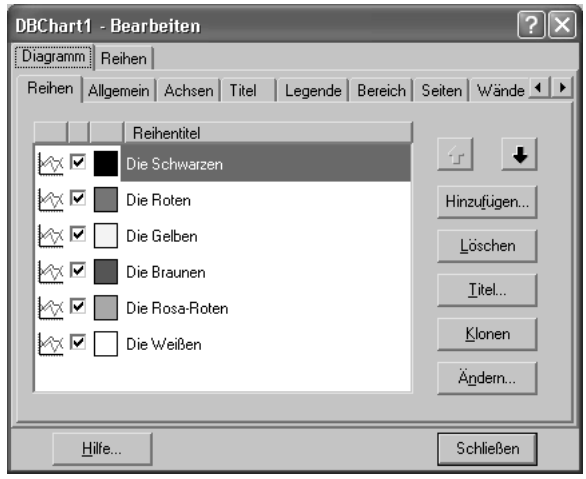
9.1.1 Das Beispielprogramm

Analysiert werden sollen in unserem Beispielprogramm die Wahlergebnisse von fiktiven Wahlen irgendwann in den nächsten Jahrzehnten, in denen ebenso fiktive Parteien gegeneinander antreten. Prinzipiell wäre es auch möglich gewesen, die Daten von Bundes- oder Landtagswahlen zu besorgen, da jedoch auch Fachbuchautoren manchmal zur Bequemlichkeit neigen, habe ich einfach mit einem kleinen Programm kurz die benötigten Datensätze generiert. Dazu wurde einfach ein kleiner Zufallsgenerator »angeworfen«, der die Stimmenzahlen ermittelt. Die »Wahlbeteiligung«, also die Summe aller Stimmen, schwankt natürlich immer ein wenig, aber da erst aus diesen Zahlen die prozentualen Ergebnisse der einzelnen Parteien berechnet werden, haben diese in der Summe nie mehr als 100%.

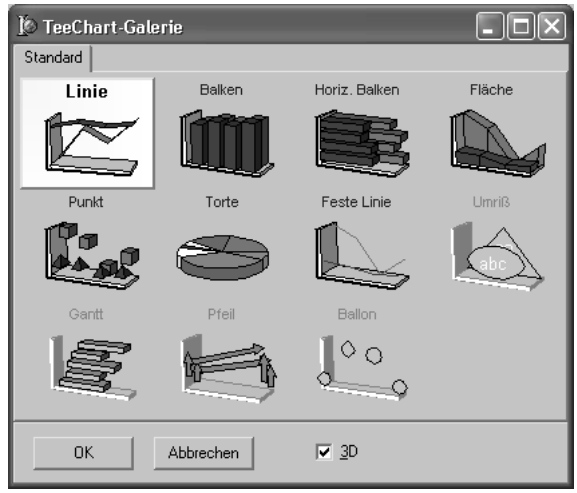
Wie diese elf Wahlen ausgefallen sind, zeigt die folgende Abbildung:



Auf der ersten Seite des *TabbedNotebooks* werden die Ergebnisse aller elf Wahlen angezeigt, mit Hilfe von *TCheckBox*-Komponenten lässt sich bestimmen, welche Parteien mit ihrem Ergebnis angezeigt werden sollen und welche nicht. Zunächst wird einmal eine TDBChart-Komponente eingefügt, mit einem Doppelklick auf dieselbe wird der Komponenteneditor geöffnet:



Nach Betätigung des Buttons HINZUFÜGEN wird zunächst ein Dialog zur Auswahl eines Diagrammtyps geöffnet. Je nachdem, welcher Diagrammtyp gewählt wird, wird eine andere Reihenkomponente erstellt – eine spätere Umwandlung ist nicht immer möglich.



Die Wahlergebnisse sollen hier als Liniendiagramm erscheinen – für jede Partei wird nun eine entsprechende Reihenkomponente erstellt und ihr als Titel der Parteiname zugewiesen, siehe Bild.

Solange keine Werte oder keine Datenquelle zugewiesen worden ist, werden zufällig generierte Werte angezeigt. Wir wollen zunächst daran gehen, die Komponente mit Daten zu versorgen. Da für jede Partei eine eigene Reihenkomponente erstellt wird, wäre dafür auch jeweils eine Datenmengenkomponente erforderlich, welche die zu der jeweiligen Partei gehörenden Daten herausfiltert. Sollen dann nicht die Stimmen, sondern die Stimmanteile dargestellt werden, wird diese Vorgehensweise unverhältnismäßig aufwändig.

Deshalb wird hier die Möglichkeit der Datenbankanbindung nicht genutzt, sondern die Werte werden einzeln aus *Table1* ausgelesen und den jeweiligen Reihenkomponenten zugewiesen.

```
procedure TForm1.FormShow(Sender: TObject);
var
  i: integer;
  ergebnisse: array[1..6] of integer;
  summe: integer;
  datum: TDateTime;
```

```
begin
  DBChart1.Visible := false;
  Series1.Clear;
    ...
  Series6.Clear;
  Table1.First;
  summe := 0;
  i := 1;
```

Zunächst wird die Eigenschaft *Visible* von *DBChart1* auf *false* gesetzt, damit auf jeden Fall ein Bildneuaufbau während der Zuweisung der Werte unterbleibt. Hier im Beispiel ist das Löschen alter Werte in den Reihenkomponenten eigentlich überflüssig.

```
  while(Table1.EOF = false) do
  begin
    ergebnisse[i] := Table1.FieldByName('Stimmen').AsInteger;
    summe := summe + ergebnisse[i];
    i := i + 1;
    if i = 7 then
    begin
      Datum := Table1.FieldByName('Datum').AsDateTime;
      if RadioGroup1.ItemIndex = 0 then
      begin
        Series1.AddXY(Datum, ergebnisse[1], '', clBlack);
          ...
        Series6.AddXY(Datum, ergebnisse[6], '', clWhite);
      end {if RadioGroup1.ItemIndex = 0 then}
```

Nacheinander werden alle Werte der Tabellen ausgelesen und – sobald eine Wahl komplett eingelesen worden ist – den Reihenkomponenten zugewiesen. Beachten Sie, dass Sie den jeweiligen Werten eine eigene Farbe zuweisen können – Sie könnten somit für jeden Wert einer Reihe eine andere Farbe verwenden.

```
      else
      begin
        Series1.AddXY(Datum, (ergebnisse[1]*100/summe), '',
          clBlack);
            ...
        Series6.AddXY(Datum, (ergebnisse[6]*100/summe), '',
          clWhite);
      end; {else RadioGroup1.ItemIndex = 0 then}
      i := 1;
```

```
      summe := 0;
    end; {if i = 7 then}
    Table1.Next;
  end; {Table1.EOF = false}
  DBChart1.Visible := true;
end; {procedure TForm1.FormShow}
```

Sollen nicht die Stimmen, sondern die Anteile dargestellt werden, dann muss die Stimmenzahl der jeweiligen Partei durch die Gesamtstimmenzahl geteilt werden. Je nach »Wahlbeteiligung« haben die Diagramme von Stimmenzahl und Anteilen ein etwas voneinander differierendes Aussehen.

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  Series1.Active := CheckBox1.Checked;
    ...
  Series6.Active := CheckBox6.Checked;
end;
```

Nicht immer sollen die Ergebnisse aller Parteien dargestellt werden. Mit Hilfe der Komponenten *CheckBox1* bis *CheckBox6* kann für jede Partei entschieden werden, ob sie in das Diagramm aufgenommen wird oder nicht. Allen diesen *TCheckBox*-Komponenten wird die Ereignisbehandlungsroutine *CheckBox1Click* zugewiesen.

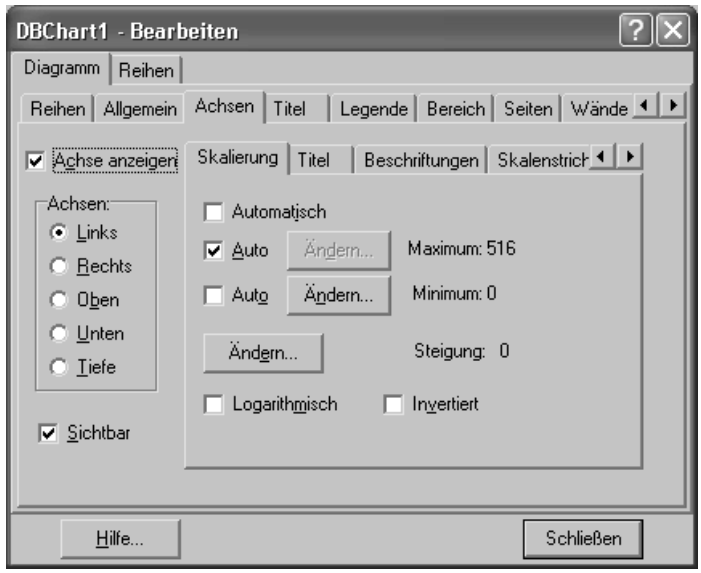
```
procedure TForm1.CheckBox7Click(Sender: TObject);
begin
  DBChart1.View3D := CheckBox7.Checked;
end;
```

Die 3-D-Darstellung macht zwar optisch keinen schlechten Eindruck, es ist aber nahezu unmöglich, auch nur halbwegs genaue Werte aus der Grafik abzulesen. Deshalb wurde die Möglichkeit implementiert, auf zweidimensionale Darstellung umzuschalten.

```
procedure TForm1.CheckBox8Click(Sender: TObject);
begin
  DBChart1.LeftAxis.AutomaticMinimum := CheckBox8.Checked;
  if CheckBox8.Checked = false
    then DBChart1.LeftAxis.Minimum := 0;
end;
```

Per Voreinstellung werden die Achsen automatisch skaliert. Diese Lösung erspart viel Arbeit und Fehlersuche, ist hier aber nicht ganz optimal, weil der Betrachter über die tatsächlichen Werte getäuscht wird.

Zunächst einmal soll das Minimum der linken Achse fest auf den Wert *null* gesetzt werden, wie dies die folgende Abbildung zeigt:



Beim Vergleich von zwei Parteien, deren Stimmergebnisse eher näher beieinander liegen, kann es jedoch sinnvoll sein, das Minimum automatisch zu wählen, weil dann der zur Verfügung stehende Platz besser genutzt wird. Mit Hilfe von *CheckBox8* kann zwischen den beiden Darstellungsarten umgeschaltet werden. Wird auf die Darstellung umgeschaltet, bei der die linke Achse mit *null* beginnt, dann muss dieser Wert explizit zugewiesen werden.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
  begin
    if FileExists(SaveDialog1.FileName) then
    begin
      if MessageDlg('Datei ' + SaveDialog1.FileName +
           ' existiert. Überschreiben?', mtWarning, mbYesNoCancel,
           0) = mrYes
        then DBChart1.SaveToBitmapFile(SaveDialog1.FileName);
    end {if FileExists(SaveDialog1.FileName)}
    else DBChart1.SaveToBitmapFile(SaveDialog1.FileName);
  end; {if SaveDialog1.Execute then}
end; {procedure TForm1.Button1Click}
```

Eine Chart-Grafik kann auch als Bitmap (oder als Metafile) gespeichert werden, die Anweisung lautet hier schlicht *SaveToBitmapFile* (und nicht *SaveToBitmap*, wie in der Online-Hilfe von Delphi 3 behauptet wird – solche Widersprüche kann man mit Hilfe der Symbolanzeige aufspüren). Der zu verwendende Dateiname wird mit Hilfe von *SaveDialog1* abgefragt. Des Weiteren wird festgestellt, ob unter dem fraglichen Namen bereits eine Datei existiert – ist dies der Fall, dann wird eine Sicherheitsabfrage durchgeführt.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if MessageDlg('Soll die Grafik gedruckt werden?', mtConfirmation,
       mbOkCancel, 0) = mrOK
    then DBChart1.PrintLandscape;
end;
```

Ebenfalls nicht weiter aufwändig ist der Ausdruck der Grafik. Da die Methode *Print* jedoch die aktuelle Seiteneinstellung übernimmt – was bei unserem Beispiel nicht gerade zu optimalen Ergebnissen führt – wird hier mit der Methode *PrintLandscape* die Grafik auf alle Fälle im Querformat gedruckt.

9.1.2 Die aktuelle Wahl

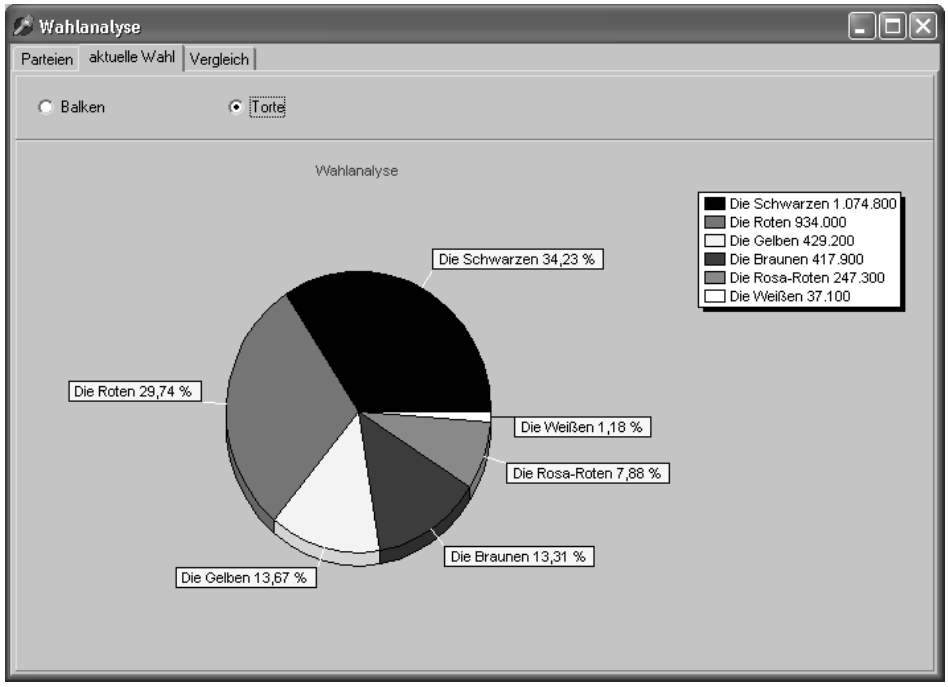
Auf der nächsten Notebook-Seite soll das Ergebnis der aktuellen Wahl dargestellt werden, wahlweise als Balken- oder als Tortengrafik. Wir wollen hier nun auch die Möglichkeit zur Datenbankanbindung von *TDBChart* nutzen. In das Formular wird deshalb die Komponente *Query1* eingefügt, der folgende SQL-Anweisung zugewiesen wird:

```
SELECT
    e.stimmen,
    p.namen
  FROM ergebnisse e, parteien p
  WHERE (e.partei = p.nummer)
    AND (e.datum = (SELECT MAX(datum) FROM ergebnisse))
```

Damit auch die Parteiennamen zugewiesen werden können, muss ein JOIN die Tabellen *ergebnisse* und *parteien* verbinden. Das Datum der jüngsten Wahl wird mit Hilfe einer Unterabfrage ermittelt.

In *DBChart1* werden nun zwei Reihenkomponenten erstellt: eine für die Balken- und eine für die Tortengrafik. Um die Datenquelle auszuwählen, wird der Komponenteneditor geöffnet und die jeweilige Reihenkomponente gewählt. Auf der Registerseite REIHEN|DATENQUELLE wird dann *Query1* als *Datenmenge* gewählt, die *Beschriftung* bezieht die Komponente aus der Spalte *namen*, *Pie* beziehungsweise *Bar* aus der Spalte *stimmen*.

Wie in der folgenden Abbildung zu sehen ist, sind in der Legende die Stimmergebnisse, bei der Beschriftung der Grafik die Anteile angegeben. Auch dies macht *TDBChart* automatisch, wenn es denn entsprechend eingestellt wird.



Um die Beschriftung der Grafik einzustellen, wählt man im Komponenteneditor REIHEN|MARKIERUNGEN und wählt dort den *Stil Beschriftung & Prozent*. Zur Gestaltung der Legende wählt man DIAGRAMM|LEGENDE, als *Textstil* wird hier *Rechter Wert* verwendet.

Momentan würden beide Reihenkomponenten gleichzeitig angezeigt – dies ist ohne weiteres möglich, aber nicht unbedingt übersichtlich. Die Wahl zwischen den beiden Reihenkomponenten erfolgt wie gehabt mit Hilfe der Eigenschaft *Active*.

Für die einzelnen Parteien werden zunächst willkürliche Farben verwendet, was gerade hier nicht so beibehalten werden kann. Prinzipiell wäre es möglich, in die Datenbanktabelle eine zusätzliche Spalte aufzunehmen, welche die Farbe der jeweiligen Partei als RGB-Wert enthält (*0* als *schwarz*, *255* als *rot* usw). Als Eigenschaft *ColorSource* der betreffenden Reihenkomponenten würde dann diese Spalte ausgewählt (die Abfrage müsste entsprechend ergänzt werden).

Nun gehören nach meiner Auffassung in eine Datenbank Daten und keine Informationen darüber, wie diese darzustellen sind. Die Farben für die einzelnen Werte können auch im Programm selbst zugewiesen werden.

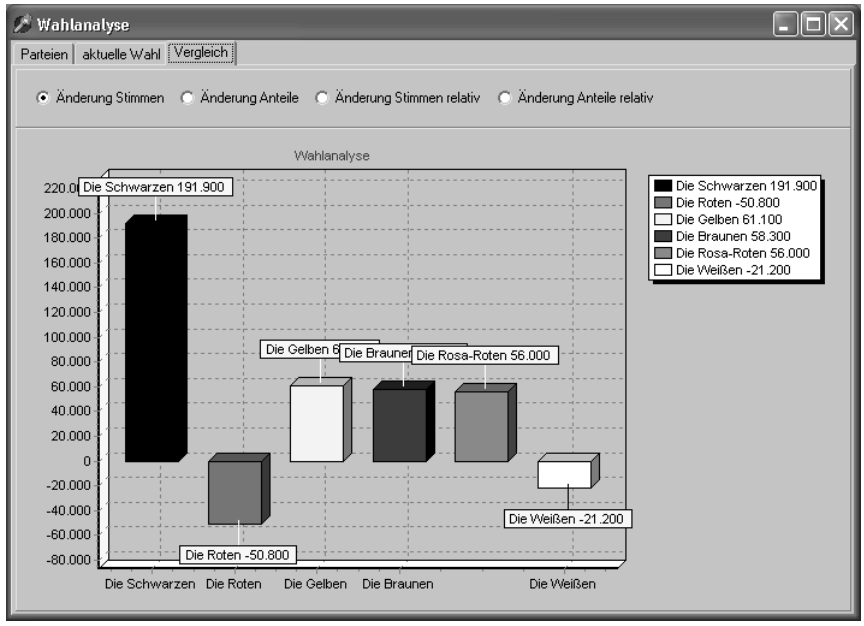
```
Series7.ValueColor[0] := clBlack;
Series7.ValueColor[1] := clRed;
Series7.ValueColor[2] := clYellow;
Series7.ValueColor[3] := clMaroon;
Series7.ValueColor[4] := clFuchsia;
Series7.ValueColor[5] := clWhite;

Series8.ValueColor[0] := clBlack;
  ...
Series8.ValueColor[5] := clWhite;
```

Die jeweilige Farbe muss den Werten in beiden Reihenkomponenten zugewiesen werden. Beachten Sie auch, dass die Zählung der Werte mit dem Indexwert *null* beginnt.

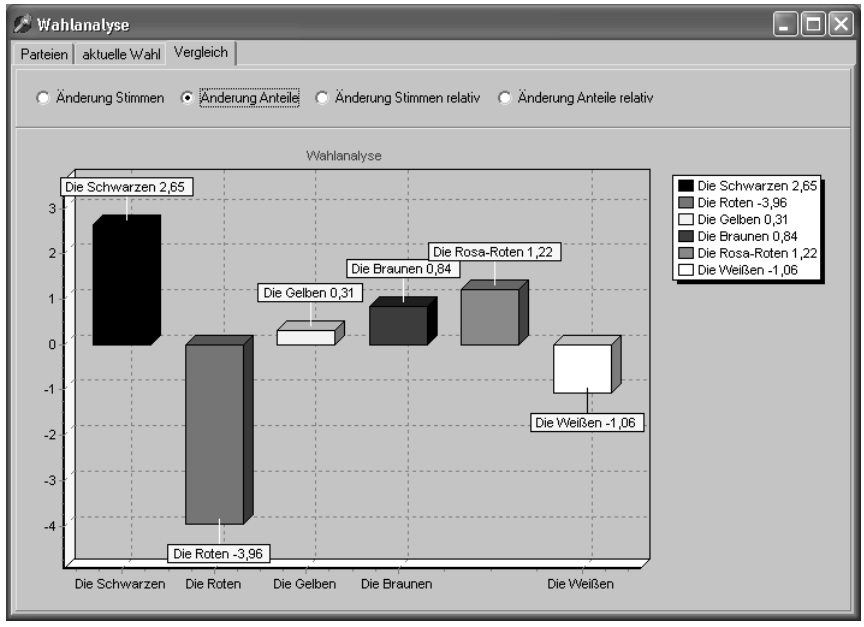
9.1.3 Gewinn- und Verlustrechnung

Auf der letzten Seite sollen die Gewinne und Verluste der einzelnen Parteien dargestellt werden. Wenn Sie sich die Grafiken auf den nächsten Seiten ansehen, dann werden Sie feststellen, dass man ein und denselben Sachverhalt (nämlich die Stimmenverschiebung zwischen der Wahl vom 20.1.2153 und der Wahl am 19.1.2157) auf ganz unterschiedliche Art und Weise darstellen kann. Vielleicht liegt es ja daran, dass sich hinterher fast jede Partei für den eigentlichen Wahlsieger hält.

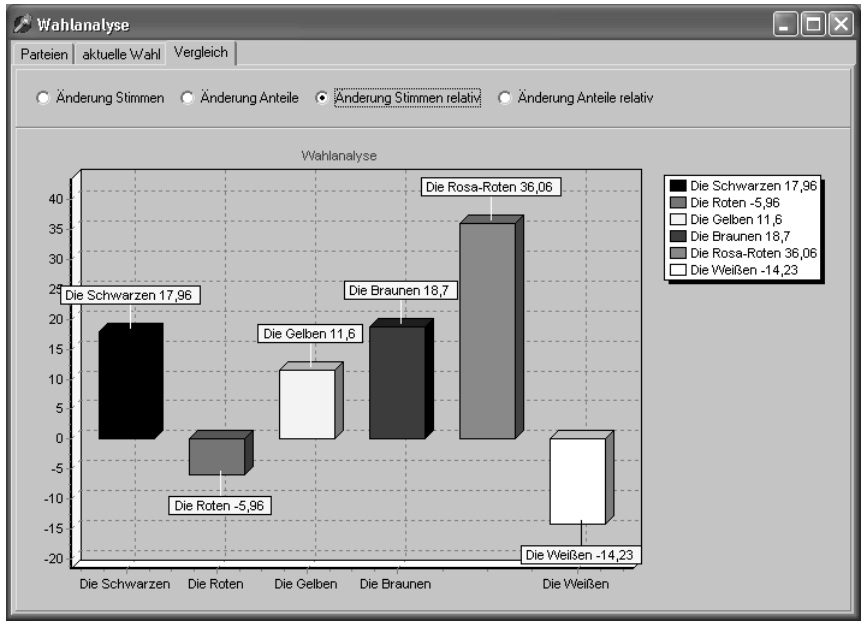
```
procedure TForm1.RadioButton3Click(Sender: TObject);
var
  neu: array[1..6] of integer;
  alt: array[1..6] of integer;
  sumalt, sumneu: integer;
  i: integer;
begin
  DBChart3.Visible := false;
  Series9.Clear;
  Table1.FindKey([205]);
  sumalt := 0;
  sumneu := 0;
```



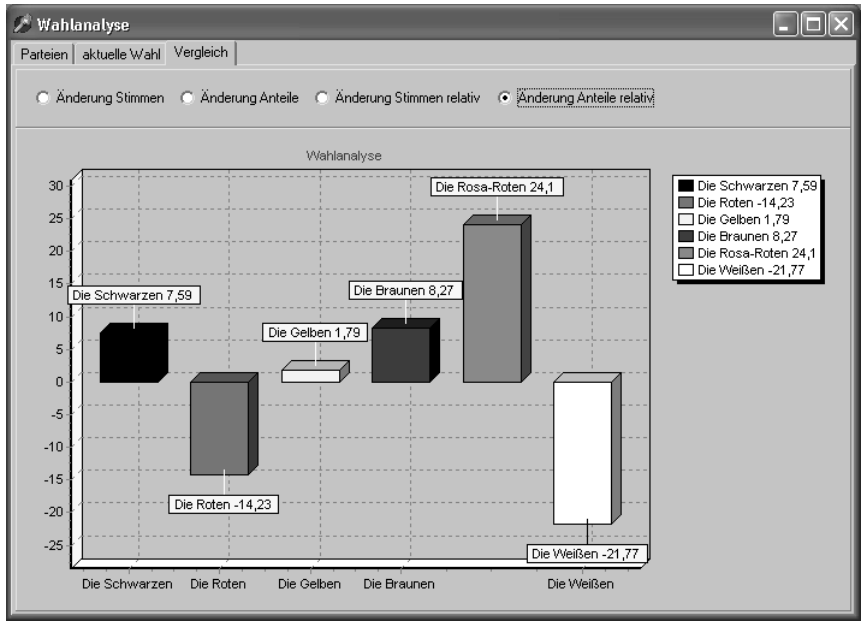
Die Veränderung der Stimmen absolut



Die Veränderung der Stimmen absolut



Die Veränderung der Stimmen absolut



Die Veränderung der Stimmen absolut

```
for i := 1 to 6 do
begin
  alt[i] := Table1.FieldByName('Stimmen').AsInteger;
  sumalt := sumalt + alt[i];
  Table1.Next;
end;
for i := 1 to 6 do
begin
  neu[i] := Table1.FieldByName('Stimmen').AsInteger;
  sumneu := sumneu + neu[i];
  Table1.Next;
end;
```

Zunächst werden die Ergebnisse der beiden Wahlen eingelesen und zu den Gesamtstimmenzahlen aufsummiert.

```
if RadioButton3.Checked = true then
begin
  Series9.AddXY(1, neu[1] - alt[1], 'Die Schwarzen', clBlack);
    ...
  Series9.AddXY(6, neu[6] - alt[6], 'Die Weißen', clWhite);
end;
```

Um die Änderungen der Stimmenzahlen zu erhalten, muss lediglich die Differenz zwischen den jeweiligen Wahlen gebildet werden.

```
if RadioButton4.Checked = true then
begin
  sumneu := sumneu div 100;
  sumalt := sumalt div 100;
  Series9.AddXY(1, neu[1]/sumneu - alt[1]/sumalt, 'Die Schwarzen',
    clBlack);
    ...
  Series9.AddXY(6, neu[6]/sumneu - alt[6]/sumalt, 'Die Weißen',
    clWhite);
end;
```

Für die Veränderungen der Anteile muss die Differenz zwischen den Anteilen gebildet werden. Diese soll jeweils als Prozentzahl ausgegeben werden, die jeweilig berechneten Anteile wären also mit 100 zu multiplizieren. Damit diese Rechnung jedoch nicht bei allen sechs Parteien durchgeführt werden muss, wurden einfach anfangs die Gesamtstimmenzahlen durch 100 dividiert.

```
  if RadioButton5.Checked = true then
  begin
    Series9.AddXY(1, (neu[1]/alt[1] - 1)*100, 'Die Schwarzen', clBlack);
      ...
    Series9.AddXY(6, (neu[6]/alt[6] - 1)*100, 'Die Weißen', clWhite);
  end;
```

Um die relative Änderung der Stimmenzahlen zu ermitteln, wird der Quotient der Wahlergebnisse gebildet, daraus wird dann der Prozentsatz der Änderung berechnet.

```
  if RadioButton6.Checked = true then
  begin
    Series9.AddXY(1, (((neu[1]/sumneu) / (alt[1]/sumalt)) - 1)*100,
      'Die Schwarzen', clBlack);
      ...
    Series9.AddXY(6, (((neu[6]/sumneu) / (alt[6]/sumalt)) - 1)*100,
      'Die Weißen', clWhite);
  end;
  DBChart3.Visible := true;
end; {procedure TForm1.RadioButton3Click}
```

Zuletzt wird berechnet, um wie viel sich relativ der Stimmenanteil verändert hat, also Prozent von Prozent. Abschließend wird die Eigenschaft *Visible* wieder auf *true* gesetzt. Die Ereignisbehandlungsroutine *RadioButton3Click* wird den *OnClick*-Ereignissen der Komponenten *RadioButton3* bis *RadioButton6* zugewiesen.

9.2 Referenz TChart

Die Komponente *TChart* dient zur Aufnahme von Reihenkomponenten. Die Reihenkomponenten werden jeweils in eigenen Kapiteln besprochen.

TChart ist von *TWinControl* abgeleitet.

- SeriesList (Eigenschaft, veröffentlicht)

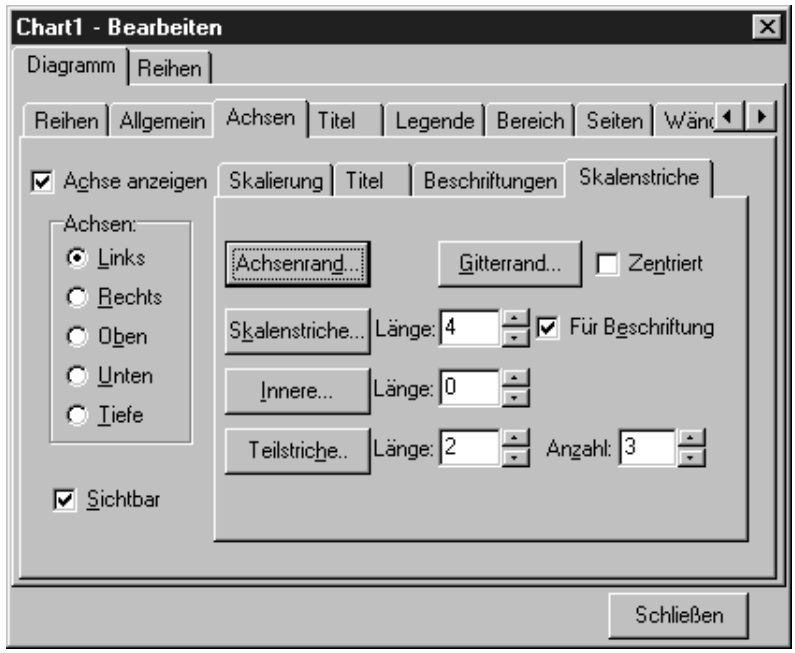
  ```
  property SeriesList: TChartSeriesList;
  ```

  Die Eigenschaft *SeriesList* verwaltet die Liste der Reihenkomponenten. Die Reihenkomponenten kapseln die Anzeige der Daten.

  Der abstrakte Vorfahre aller Reihenkomponenten – *TChartSeries* – ist in Kapitel 9.3 beschrieben, die instantisierbaren Reihenkomponenten in Kapitel 9.4.

9.2.1 Die Achsen

Die folgenden Eigenschaften benötigen Sie lediglich, wenn Sie zur Laufzeit die Achseneinstellungen ändern müssen. Zur Entwurfszeit verwenden Sie lieber den Komponenteneditor:



- BottomAxis, LeftAxis, RightAxis, TopAxis, DepthAxis
  (Eigenschaften, veröffentlicht)

```
property BottomAxis : TChartAxis;
property LeftAxis : TChartAxis;
property RightAxis : TChartAxis;
property TopAxis : TChartAxis;
property DepthAxis: TChartAxis;
```

Die Komponente *TChart* implementiert Achsen an allen vier Rändern der Komponente. Damit diese gezeichnet werden, muss nicht nur die Eigenschaft *Visible* jeweils *true* sein, die Achse muss auch bei wenigstens einer Reihenkomponente als *HorizAxis* oder *VertAxis* ausgewählt sein. Da jede Reihenkomponente nur jeweils eine horizontale und vertikale Achse wählen kann, müssen mindestens zwei Reihenkomponenten verwendet werden, damit sowohl die obere wie die untere (beziehungsweise sowohl die linke als auch rechte) Achse angezeigt werden kann.

Eine weitere Achse für die Tiefe eines dreidimensionalen Diagramms wird als *DepthAxis* implementiert.

Die Eigenschaften und Methoden von *TChartAxis* werden im kommenden Abschnitt besprochen.

- AxisVisible (Eigenschaft, veröffentlicht)

```
property AxisVisible : Boolean default true;
```

Um alle Achsen gemeinsam von der Anzeige auszuschließen, wird die Eigenschaft *AxisVisible* auf *false* gesetzt.

- GetAxisSeries (Methode)

```
function GetAxisSeries(Axis: TChartAxis): TChartSeries ;
```

Die Funktion *GetAxisSeries* ermittelt die erste Reihenkomponente, welche die als Parameter übergebene Achse *Axis* als horizontale beziehungsweise vertikale Achse gewählt hat. Hat keine Reihenkomponente diese Achse gewählt, gibt die Funktion den Wert *nil* zurück.

- MaxXValue, MaxYValue, MinXValue, MinYValue (Methoden)

```
function MaxXValue (AAxis: TChartAxis): Double ;
function MaxYValue (AAxis: TChartAxis): Double ;
function MinXValue (AAxis: TChartAxis): Double ;
function MinYValue (AAxis: TChartAxis): Double ;
```

Diese Funktionen geben die Minima und Maxima der mit der jeweiligen Achse verbundenen Wertelisten an. Diese Funktionen sind beispielsweise dann sehr nützlich, wenn man die Achsen manuell skaliert, die Skalierung jedoch zur Laufzeit den Werten anpassen möchte.

Die Komponente TChartAxis

- Visible (Eigenschaft, veröffentlicht)

```
property Visible : Boolean default true;
```

Zum Zeigen und Verbergen der jeweiligen Achse wird die Eigenschaft *Visible* verwendet. Sollen alle Achsen gemeinsam verborgen werden, kann dazu die *TChart*-Eigenschaft *AxisVisible* verwendet werden.

- Automatic, AutomaticMaximum, AutomaticMinimum (Eigenschaften, veröffentlicht)

```
property Automatic: Boolean default true;
property AutomaticMaximum: Boolean default true;
property AutomaticMinimum: Boolean default true;
```

Wird die Eigenschaft *Automatic* auf *true* gesetzt, dann wird die Achse automatisch skaliert, anderenfalls kann immer noch das Maximum oder das Minimum automatisch skaliert werden.

Wird die automatische Skalierung abgestellt, dann haben die Eigenschaften *Maximum* und *Minimum* den Wert, der von der Automatik berechnet wurde. Sollen die Werte verwendet werden, die vor dem Anstellen der Automatik verwendet wurden, dann sind diese explizit zu setzen.

- Maximum, Minimum, Inverted (Eigenschaften, veröffentlicht)

```
property Maximum: Double default 0;
property Minimum: Double default 0;
property Inverted: Boolean default false;
```

Um die Achsen manuell zu skalieren, werden die Eigenschaften *Maximum* und *Minimum* entsprechend gesetzt – die Automatik muss dafür abgeschaltet sein. In jedem Fall, auch bei der Verwendung der Automatik, lassen sich die Werte auslesen.

Um für die Achsenskalierung Minimum und Maximum zu vertauschen, wird die Eigenschaft *Inverted* auf *true* gesetzt.

- Logarithmic (Eigenschaft, veröffentlicht)

```
property Logarithmic: Boolean default false;
```

Um eine Achse logarithmisch zu skalieren, wird die Eigenschaft *Logarithmic* auf *true* gesetzt. Dabei müssen die Eigenschaften *Maximum* und *Minimum* Werte größer *null* haben und dürfen nicht als Datumswerte angezeigt werden.

- RoundFirstLabel (Eigenschaft, veröffentlicht)

```
property RoundFirstLabel: Boolean default true;;
```

Wenn das Minimum einer Achse bei *37,25* liegen würde, dann würde die Beschriftung der Achse (je nach Werteumfang und Skalenteilung) beispielsweise bei *47,25* beginnen. Setzt man nun die Eigenschaft *RoundFirstLabel* auf *true*, dann beginnt die Skalierung mit einem »runderen« Wert, beispielsweise mit *40*.

- AxisValuesFormat (Eigenschaft, veröffentlicht)

```
property AxisValuesFormat: String default #,##0.###;
```

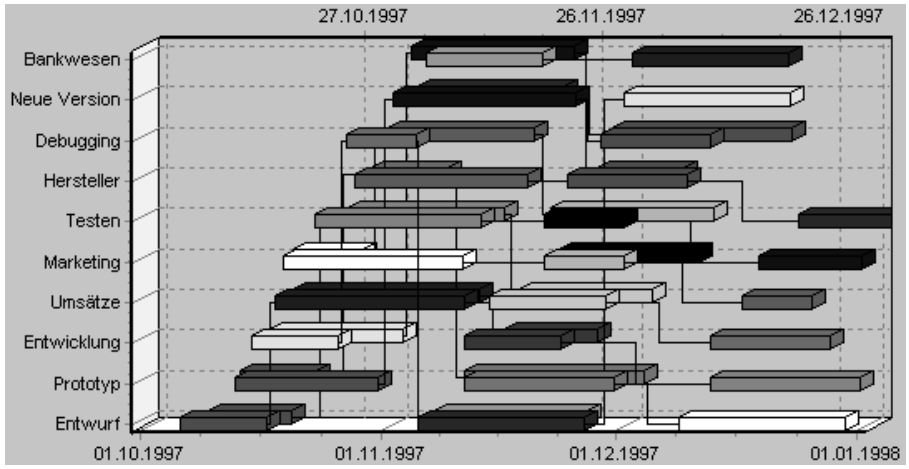
Mit Hilfe der Eigenschaft *AxisValueFormat* wird die Achsenbeschriftung formatiert. Der Vorgabewert *#,##0.###* formatiert die Beschriftung auf maximal drei Stellen genau und fügt ein Dezimal- und ein Tausendertrennzeichen ein – bei deutscher Systemeinstellung ein Dezimalkomma und einen Tausenderpunkt.

- DateTimeFormat, ExactDateTime (Eigenschaften, veröffentlicht)

```
property DateTimeFormat: String;
property ExactDateTime: Boolean;
```

Zum Formatieren von Datums- und Zeitangaben wird die Eigenschaft *DateTimeFormat* verwendet. Sollen beispielsweise die Jahreszahlen vierstellig ausgegeben werden, dann wird der Wert *dd.mm.yyyy* verwendet.

Die Funktion der Eigenschaft *ExactDateTime* lässt sich in der folgenden Abbildung betrachten:



Bei sonst identischen Einstellungen hat bei der unteren Achse die Eigenschaft *ExactDateTime* den Wert *true*, bei der oberen Achse den Wert *false*.

- Increment (Eigenschaft, veröffentlicht)

```
property Increment: Double;
```

Um den Abstand zwischen zwei Skalen(grob-)teilungen zu bestimmen, wird die Eigenschaft *Increment* verwendet. Ist dieser Wert für eine vernünftige Anzeige zu klein, berechnet die Komponente selbst einen akzeptablen Wert.

Da *TDateTime*-Werte *Double*-Zahlen sind, können auf diese Weise auch Zeitabstände festgelegt werden. Mit der folgenden Anweisung würde eine Einteilung in Monate festgelegt (siehe dazu auch die Eigenschaft *ExactDateTime*):

```
Chart1.BottomAxis.Increment := 30;
```

Was man bei Datumswerten noch problemlos mit Kopfrechnen bewältigen kann, wird bei Zeitwerten doch problematisch. Aus diesem Grund wurde das

Konstanten-Array *DateTimeStep* definiert, welches alle gebräuchlichen Datums- und Zeitintervalle implementiert.

```
TDateTimeStep = (dtOneSecond, dtFiveSeconds, dtTenSeconds,
  dtFifteenSeconds, dtThirtySeconds, dtOneMinute, dtFiveMinutes,
  dtTenMinutes, dtFifteenMinutes, dtThirtyMinutes, dtOneHour,
  dtTwoHours, dtSixHours, dtTwelveHours, dtOneDay, dtTwoDays,
  dtThreeDays, dtOneWeek, dtHalfMonth, dtOneMonth, dtTwoMonths,
  dtSixMonths, dtOneYear);

Chart1.BottomAxis.Increment := DateTimeStep[dtOneMonth};
```

- Ticks, TickLength, TickInnerLength, TicksInner (Eigenschaften, veröffentlicht)

```
property Ticks: TChartPen;
property TickLength: Integer default 4;
property TickInnerLength: Integer default 0;
property TicksInner: TChartPen;
```

Mit Hilfe der Eigenschaften *Ticks* und *TickLength* wird das Aussehen der Skalenstriche eingestellt. Die Skalenstriche, die innerhalb des Diagramms angezeigt werden (nicht zu verwechseln mit dem Gitter), werden mit den Eigenschaften *TickInnerLength* und *TicksInner* formatiert.

- MinorTickCount, MinorTicks, MinorTickLength (Eigenschaften, veröffentlicht)

```
property MinorTickCount: Integer default 3;
property MinorTicks: TChartPen;
property MinorTickLength: Ipasster default 2;
```

Zwischen den Skalenstrichen wird noch eine Feinteilung angezeigt, die nicht beschriftet wird. Die Anzahl der Feinteilungsstriche wird mit *MinorTickCount* eingestellt, *MinorTicks* und *MinorTickLength* bestimmen das Aussehen.

- Grid (Eigenschaft, veröffentlicht)

```
property Grid : TChartPen;
```

Zum leichteren Ablesen der Werte kann innerhalb des Diagramms ein Skalengitter angezeigt werden. Die dabei verwendeten Linien werden mit Hilfe der Eigenschaft *Grid* eingestellt.

- Labels und andere (Eigenschaften, veröffentlicht)

```
property Labels: Boolean default true;
property LabelsFont: TFont;
property LabelsAngle: Integer default 0;
property LabelsSeparation: Integer;
```

```
property LabelsSize: Integer default 0;
property LabelStyle: TAxisLabelStyle default talAuto;
```

Mit den aufgeführten Eigenschaften wird das Aussehen der Skalenbeschriftung bestimmt. Mit Hilfe der Eigenschaft *Labels* wird bestimmt, ob die Skalenbeschriftung überhaupt angezeigt wird, mit *LabelsFont* wird die dabei verwendete Schrift festgelegt. Um die Beschriftung in 90°-Schritten zu drehen, wird die Eigenschaft Labels *Angle* verwendet. Mit *LabelsSeparation* wird der Mindestabstand zwischend den Beschriftungen vorgegeben.

Der Abstand zwischen der unteren Komponentenkante und der Beschriftung wird automatisch berechnet, wenn *LabelsSize* gleich *null* ist – anderenfalls wird er mit dieser Eigenschaft eingestellt. Was denn nun genau als Beschriftungstext angezeigt wird, kann mit der Eigenschaft *LabelStyle* festgelegt werden. Wenn Sie die Skalenbeschriftung selbst erstellen möchten, dann können Sie dafür das *TChart*-Ereignis *OnGetAxisLabel* verwenden.

- Title (Eigenschaft, veröffentlicht)

```
property Title: string;
```

Für zusätzliche Informationen, beispielsweise, welche Dimension für die Achse verwendet wird, kann die Eigenschaft *Title* verwendet werden.

- Berechnungsmethoden

```
function CalcPosPoint(Value: Integer): Double;
function CalcXPosValue(const Value: Double): Longint;
function CalcXSizeValue(const Value: Double): Longint;
function CalcYPosValue(const Value: Double): Longint;
function CalcYSizeValue(const Value: Double): Longint;
```

Um Komponentenkoordinaten in Realwerte umzurechnen und umgekehrt, werden die aufgeführten Methoden verwendet.

Zoomen und Verschieben

Um bei komplexen Diagrammen alle Details genau betrachten zu können, unterstützt die Komponente *TChart* das Zoomen und das Verschieben. Um einen Ausschnitt zu vergrößern, ziehen Sie mit der linken Maustaste von links oben nach rechts unten ein Rechteck über diesen Ausschnitt auf. Dieser Ausschnitt wird dann auf Komponentengröße vergrößert. Um wieder den ursprünglichen Vergrößerungsfaktor einzustellen, ziehen Sie mit der linken Maustaste wieder ein Rechteck auf, diesmal jedoch nach oben oder von rechts oben nach links unten.

Um den Bildausschnitt zu verschieben, halten Sie die rechte Maustaste gedrückt und verschieben die Maus in die gewünschte Richtung.

- AllowPanning (Eigenschaft, veröffentlicht)

```
property AllowPanning: (pmNone, pmHorizontal, pmVertical, pmBoth)
  default pmBoth;
```

  Mit *AllowPanning* legt man fest, ob und in welche Richtungen der Diagrammausschnitt verschoben werden darf.

- OnScroll (Ereignis)

```
property OnScroll(Sender: TObject);
```

  Beim Verschieben des Diagrammausschnittes tritt das Ereignis *OnScroll* auf.

- AllowZoom (Eigenschaft, veröffentlicht)

```
property AllowZoom: Boolean default true;
```

  Diese Eigenschaft bestimmt, ob Zoomen möglich ist oder nicht.

- AnimatedZoom, AnimatedZoomSteps (Eigenschaften, veröffentlicht)

```
property AnimatedZoom: Boolean default false;
property AnimatedZoomSteps: Integer default 8;
```

  Wird *AnimatedZoom* auf *true* gesetzt, dann wird in mehreren Stufen auf den endgültigen Ausschnitt gezoomt. Die Anzahl der Zwischenschritte wird mit *AnimatedZoomSteps* eingestellt.

- ZoomRect, ZoomPercent, UndoZoom (Methoden)

```
procedure ZoomRect(const Rect: TRect);
procedure ZoomPercent(const  PercentZoom: Double);
procedure UndoZoom ;
```

  Um vom Quelltext aus den Bildausschnitt zu verändern, werden die hier aufgeführten Methoden verwendet. Während bei *ZoomRect* der Bildausschnitt in Pixeln angegeben wird, erwartet die Prozedur *ZoomPercent* die Angabe, um wie viel Prozent der Ausschnitt vergößert oder verkleinert werden soll. Mit *UndoZoom* wird der Standardbildausschnitt wiederhergestellt.

- OnZoom, OnUndoZoom (Ereignisse)

```
property OnZoom(Sender: TObject);
property OnUndoZoom(Sender: TObject);
```

  Beim Zoomen oder bei der Rückstellung auf den Standardbildausschnitt treten diese beiden Ereignisse auf und ermöglichen es beispielsweise, das Hintergrundbild (welches nicht vergrößert wird) zu entfernen.

9.2.2 Legende

- Legend (Eigenschaft, veröffentlicht)

```
property Legend: TChartLegend;
```

Per Voreinstellung an der rechten Seite des Diagramms wird eine Legende angezeigt, deren Aussehen mit der Eigenschaft *Legend* eingestellt wird. *TChartLegend* wird im folgenden Abschnitt behandelt.

- OnGetLegendRect (Ereignis)

```
property OnGetLegendRect(Sender: TCustomChart; var Rect: TRect);
```

Das Ereignis *OnGetLegendRect* tritt auf, wenn die Legende als Ganzes platziert wird. Um sie beispielsweise um 50 Pixel nach unten zu verschieben, würde man folgende Ereignisbehandlungsroutine implementieren:

```
procedure TForm1.DBChart3GetLegendRect(Sender: TCustomChart;
  var Rect: TRect);
begin
  Rect.Top := Rect.Top + 50;
  Rect.Bottom := Rect.Bottom + 50;
end;
```

- OnGetLegendPos (Ereignis)

```
property OnGetLegendPScrollbalkenustomChart; Index: Integer;
  var X, Y, XColor: Integer);
```

Das Ereignis *OnGetLegendPos* tritt vor dem Zeichnen jedes Legendeneintrags auf. Mit der folgenden Prozedur würde man den vierten Eintrag (die Zählung beginnt bei *null*) um zehn Pixel nach rechts verschieben. (Schaffen Sie gegebenfalls mit *OnGetLegendRect* ausreichend Platz dafür.)

```
procedure TForm1.DBChart3GetLegendPos(Sender: TCustomChart;
  Index: Integer; var X, Y, XColor: Integer);
begin
  if Index = 3
    then X := x + 10;
end;
```

- OnGetLegendText (Ereignis)

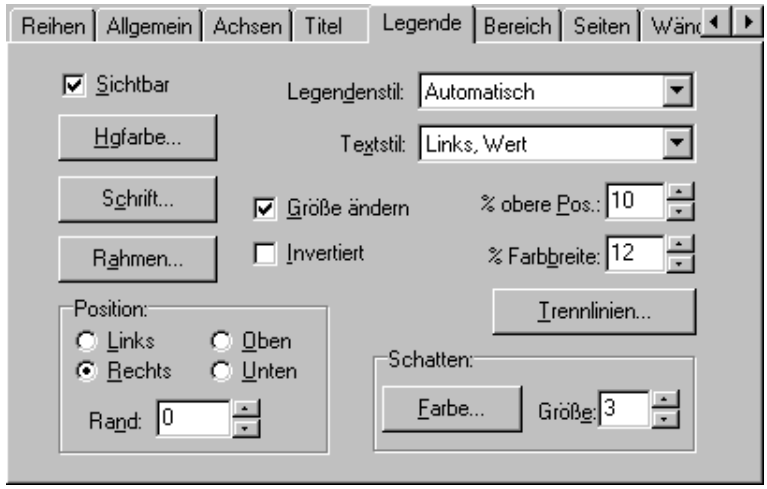
```
property OnGetLegendText(Sender: TCustomAxisPanel;
  LegendStyle: TLegendStyle; Index: Integer;
  var LegendText: String);
```

Auch der Text, der in der Legende angezeigt wird, kann modifiziert werden. Zu diesem Zweck wird die Ereignisbehandlungsroutine des Ereignisses *OnGetLegendText* verwendet. Der neue Text wird dem Variablen-Parameter *LegendText* zugewiesen. Hier ist es nicht erforderlich, das Legendenrechteck zu verbreitern, da dies von der Komponente automatisch erledigt wird.

```
procedure TForm1.DBChart3GetLegendText(Sender: TCustomAxisPanel;
  LegendStyle: TLegendStyle; Index: Integer;
  var LegendText: string);
begin
if Index = 3
  then LegendText := LegendText + ' (vorläufiges Ergebnis)';
end;
```

TChartLegend

Die *TChart*-Eigenschaft *Legend* ist vom Typ *TChartLegend*. Zur Entwurfszeit können diese Eigenschaften auch per Dialog gesetzt werden:



- Visible (Eigenschaft, veröffentlicht)

  ```
  property Visible: Boolean;
  ```

  Legt fest, ob eine Legende angezeigt wird oder nicht.

- Alignment (Eigenschaft, veröffentlicht)

  ```
  property Alignment: (laLeft, laRight, laTop, laBottom)
    default laLeft;
  ```

Mit Hilfe der Eigenschaft *Alignment* wird bestimmt, an welcher Position die Legende angezeigt wird. Die Werte *laLeft* und *laRight* erzwingen eine einspaltige, die Werte *laTop* und *laBottom* eine einzeilige Legende.

- LegendStyle (Eigenschaft, veröffentlicht)

```
property LegendStyle: TLegendStyle default lsAuto;
```

Mit Hilfe der Eigenschaft *LegendStyle* kann angegeben werden, welche Informationen in der Legende angezeigt werden sollen. Hierbei sind folgende Werte möglich:

- *lsAuto:* Wie *lsValues* bei nur einer aktiven Reihenkomponente, ansonsten wie bei *lsSeries*.
- *lsSeries:* Zeigt den oder die Reihentitel an.
- *lsValues:* Zeigt die Werte der ersten aktiven Reihe an.
- *lsLastValues:* Zeigt den oder (bei mehreren Reihenkomponenten) die jeweils letzten Werte an.

- TextStyle (Eigenschaft, veröffentlicht)

```
property TextStyle: TLegendTextStyle;
```

Mit der Eigenschaft *TextStyle* wird bestimmt, wie die Legendeneinträge formatiert werden. Soll nur die Wertbeschriftung angezeigt werden, dann erhält *TextStyle* den Wert *ltsPlain*. Es ist auch möglich, die Beschriftung und den Wert gemeinsam anzuzeigen, wobei die Beschriftung links (*ltsRightValue*) oder rechts (*ltsLeftValue*) steht. Des Weiteren ist es möglich, die Beschriftung und den von der Komponente berechneten Prozentwert anzuzeigen (*ltsLeftPercent* und *ltsRightPercent*).

- FirstValue, Inverted (Eigenschaften, veröffentlicht),
  TotalLegendItems (Methode)

```
property FirstValue: LongInt;
property Inverted: Boolean;
function TotalLegendItems: Longint;
```

Mit Hilfe der Eigenschaft *Inverted* wird die Reihenfolge der Legendeneinträge invertiert. Die Eigenschaft *FirstValue* legt fest, ab welchem Platz in der Reihenfolge die Legendeneinträge angezeigt werden. Auf diese Weise ist es möglich, mit Hilfe entsprechender Anweisungen die Legende »durchzuscrollen«, wenn der zur Verfügung stehende Platz nicht für alle Legendeneinträge ausreicht. Zur Bestimmung der Gesamtzahl aller Legendeneinträge wird die Funktion *TotalLegendItems* verwendet.

- ColorWidth (Eigenschaft, veröffentlicht)

  ```
  property ColorWidth : Integer;
  ```

  Vor dem Legendeneintrag wird ein kleines Rechteck in der Farbe des jeweiligen Diagrammsymbols angezeigt, dessen Breite sich mit *ColorWidth* verändern lässt. Dies ist besonders dann nützlich, wenn – beispielsweise bei Balken- oder Tortendiagrammen – die Symbole gemustert werden.

9.2.3 Darstellung

Die folgenden Eigenschaften regeln weitere Details der Darstellung:

3-D-Darstellung

- View3D (Eigenschaft, veröffentlicht)

  ```
  property View3d: Boolean default true;
  ```

  Mit *View3D* wird die (pseudo-)dreidimensionale Darstellung an- und abgeschaltet.

- Chart3dPercent (Eigenschaft, veröffentlicht)

  ```
  property Chart3dPercent: Integer default 15;
  ```

  Diese Eigenschaft regelt das Größenverhältnis zwischen Diagrammhöhe und -breite einerseits und Diagrammtiefe andererseits. Der Wert wird in Prozent angegeben und darf zwischen 1 und 100 liegen.

- View3dWalls, LeftWall, BottomWall (Eigenschaften, veröffentlicht)

  ```
  property View3dWalls: Boolean default true;
  property LeftWall: TChartWall;
  property BottomWall: TChartWall;
  ```

  Sollen bei 3-D-Darstellung links und unten Wände gezeichnet werden, dann muss die Eigenschaft *View3dWalls* auf *true* gesetzt werden. Das Aussehen der Wände kann mit den Eigenschaften *LeftWall* und *BottomWall* bestimmt werden. Eine Beschreibung von *TChartWall* ist in der Online-Hilfe zu finden.

Hintergrund

- BackColor (Eigenschaft, veröffentlicht)

```
property BackColor: TColor default clScrollBar;
```

  Mit *BackColor* wird die Grafikgrundfarbe spezifiziert. Lassen Sie diese Eigenschaft auf *clScrollBar*, wenn Sie einen Farbverlauf oder ein Hintergrundbild verwenden.

- Gradient (Eigenschaft, veröffentlicht)

```
property Gradient: TChartGradient;
```

  Um einen Farbverlauf als Hintergrund zu setzen, wird die Eigenschaft *Gradient* verwendet. Mit *Gradient.StartColor* und *Gradient.EndColor* werden die Farben vorgegeben, mit *Gradient.Direction* die Richtung des Verlaufs. Des Weiteren muss die Eigenschaft *Visible* auf *true* gesetzt werden.

- BackImage, BackImageMode, BackImageInside (Eigenschaften, veröffentlicht)

```
property BackImagePicture;
property BackImageMode: (pbmStretch, pbmTile, pbmCenter)
  default pbmStretch;
property BackImageInside: Boolean default false;
```

  Um ein Hintergrundbild zu verwenden, weist man der Eigenschaft *BackImage* ein Bild zu.

  In den meisten Fällen wird das Hintergrundbild nicht exakt in das Diagramm passen. Man kann es entweder zentrieren (*pbmCenter*), auf Diagrammabmessungen vergrößern oder verkleinern (*cbmStretch*) oder kacheln (*pbmTile*).

  Normalerweise bildet das Bild den Hintergrund für die ganze Komponente. Soll es nur hinter dem eigentlichen Diagramm angezeigt werden, setzt man die Eigenschaft *BackImageInside* auf *true*.

Titel und Ränder

- MarginBottom, MarginLeft, MarginRight, MarginTop (Eigenschaften, veröffentlicht)

```
property MarginLeft: Integer default 4;
property MarginTop: Integer default 3;
property MarginRight: Integer default 3;
property MarginBottom: Integer default 4;
```

  Die oben genannten Eigenschaften bestimmen die Ausdehnung der Ränder, von denen das Diagramm umgeben wird.

- Title, Foot (Eigenschaften, veröffentlicht)

```
property Title: TChartTitle;
property Foot: TChartTitle;
```

Am oberen und unteren Rand der Komponente lässt sich mit Hilfe dieser Eigenschaften ein Text ausgeben. *TChartTitle* implementiert unter anderem folgende Eigenschaften:

- *Text (TStrings):* der anzuzeigende Text
- *Visible (boolean):* Titel anzeigen oder nicht anzeigen
- *Font (TFont):* Schriftart des Textes
- *Alignment (TAlignment):* Ausrichtung links, Mitte oder rechts
- *Frame (TChartHiddenPen):* Rahmen um den Text

Seiten

Sollen in einem Diagramm sehr viele Punkte angezeigt werden, dann kann darunter die Übersichtlichkeit leiden. Um dem vorzubeugen, kann man das Diagramm auf mehrere Seiten verteilen.

- MaxPointsPerPage (Eigenschaft, veröffentlicht)

```
property MaxPointsPerPage: LongInt default 0;
```

Per Voreinstellung hat diese Eigenschaft den Wert *null*. Dies bewirkt, dass alle Punkte auf der ersten Seite dargestellt werden. Wird ein Wert größer *null* eingegeben, dann gibt dieser Wert vor, wie viele Punkte maximal auf einer Seite dargestellt werden. Sind mehr Punkte vorhanden, dann werden diese auf mehrere Seiten aufgeteilt.

- Page (Eigenschaft, veröffentlicht), NumPages (Methode)

```
property Page: Longint default 1;
function NumPages: Longint;
```

Mit Hilfe der Eigenschaft *Page* kann sowohl bestimmt werden, welche Seite gerade angezeigt wird, als auch die Seite gewechselt werden. Die Zählung der Seiten beginnt bei 1.

Die Funktion *NumPages* ermittelt die Zahl der zurzeit vorhandenen Seiten.

- NextPage, PreviousPage (Methoden)

```
procedure NextPage;
procedure PreviousPage;
```

  Mit diesen Methoden wird auf die nächste beziehungsweise auf die vorangehende Seite gewechselt.

- OnPageChange (Ereignis)

```
property OnPageChange(Sender: TObject);
```

  Beim Wechseln von einer Seite zur anderen tritt das Ereignis *OnPageChange* auf. Mit Hilfe dieses Ereignisses kann beispielsweise die Anzeige, auf welcher Seite man sich gerade befindet, aktualisiert werden.

Exportieren und Drucken

TChart implementiert für das Exportieren und das Drucken der Diagramme wichtige Funktionen.

- SaveToBitmap, SaveToMetafile, SaveToMetafileEnh (Methoden)

```
procedure SaveToBitmap(const FileName: string ) ;
procedure SaveToMetafile(const FileName: string);
procedure SaveToMetafileEnh(const FileName: string);
```

  Mit Hilfe der genannten Methoden kann das Diagramm in einem Grafikformat abgespeichert werden. Den Metafiles wird ein so genannter Aldus-Header vorangestellt, der in vielen Fällen für das Einfügen der Grafiken erforderlich ist. *Enhanced Metafiles* sind ein 32-Bit-Format, das zusätzliche Funktionen unterstützt.

- CopyToClipboardBitmap, CopyToClipboardMetafile (Methoden)

```
procedure CopyToClipboardBitmap;
procedure CopyToClipboardMetafile;
```

  Mit diesen Methoden lässt sich die Grafik in die Zwischenablage kopieren.

- Print, PrintRect (Methoden)

```
procedure Print ;
procedure PrintRect (const R: TRect);
```

  Mit der Methode *Print* wird das Diagramm gedruckt. Dabei wird die aktuelle Seitenausrichtung verwendet.

  Soll das Diagramm nicht die ganze Seite ausfüllen, dann kann mit *PrintRect* bestimmt werden, welche Position auf dem Papier das Diagramm einnehmen soll.

- PrintLandscape, PrintOrientation, PrintPortrait (Methoden)

```
procedure PrintLandscape;
procedure PrintOrientation (AOrientation: TPrinterOrientation);
procedure PrintPortrait ;
```

  Auch diese Methoden drucken das Diagramm aus, dabei kann jedoch die Seitenlage des Druckers explizit angegeben werden.

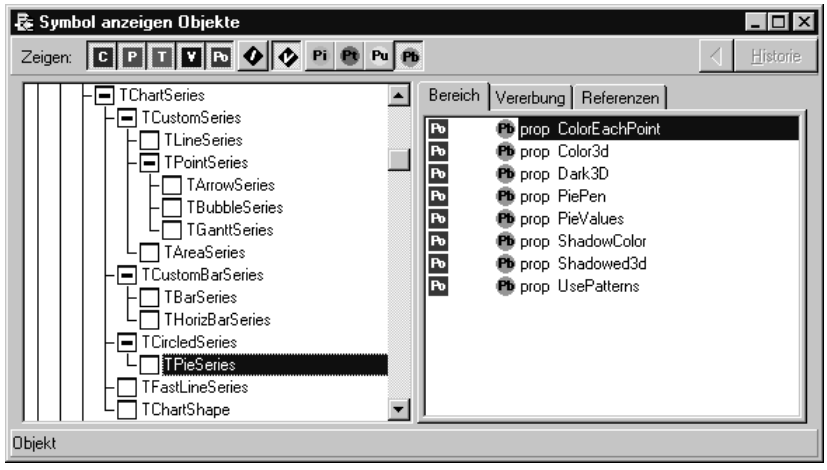
- PrintPartialCanvas (Methode)

```
procedure PrintPartialCanvas(PrintCanvas: TCanvas;
  const PrinterRect: TRect);
```

  Soll ein Diagramm mit anderen Elementen ausgedruckt oder sollen mehrere Diagramme auf eine Seite gedruckt werden, dann wird dazu die Methode *PrintPartialCanvas* verwendet. Dabei wird der Druckauftrag weder gestartet noch beendet, die dafür benötigten Anweisungen müssen explizit aufgerufen werden. Diese Methode gibt nur das Diagramm zum Drucker, ähnlich wie *Printer.TextOut* eine Textzeile ausgibt.

9.3 TChartSeries

Die Komponente *TChartSeries* ist der abstrakte Vorfahre aller Reihenkomponenten. *TChartSeries* ist von *TComponent* abgeleitet.



Die einzelnen Reihenkomponenten sind in Kapitel 9.4 beschrieben.

Allgemeine Eigenschaften

- Active (Eigenschaft, veröffentlicht)

  ```
  property Active: Boolean default true;
  ```

  Die Eigenschaft *Active* legt fest, ob eine Reihenkomponente angezeigt wird oder nicht. Das folgende Beispiel zeigt, wie man mit Hilfe einer *TCheckBox*-Komponente die Anzeige einer Reihenkomponente steuert.

  ```
  Series1.Active := CheckBox1.Checked;
  ```

- ParentChart (Eigenschaft, veröffentlicht)

  ```
  property ParentChart : TCustomChart
  ```

  Verweist auf die übergeordnete Chart-Komponente.

- Title (Eigenschaft, veröffentlicht)

  ```
  property Title : String;
  ```

  Titel der Reihenkomponente, nicht zu verwechseln mit dem Komponentennamen.

9.3.1 Grafische Darstellung

- ValueColor (Eigenschaft, öffentlich)

  ```
  property ValueColor[Index:Longint]:TColor
  ```

  Mit Hilfe der Eigenschaft *ValueColor* kann jedem einzelnen Wert im Diagramm eine eigene Farbe zugewiesen werden. Die Eigenschaft *ColorEachPoint* hat dabei manchmal Einfluss auf die Darstellung.

- ColorSource (Eigenschaft, veröffentlicht)

  ```
  property ColorSource : String;
  ```

  Informationen über die Farbdarstellung kann die Reihenkomponente auch aus der Datenmenge beziehen, welche die übrigen Daten liefert. Dazu wird die Eigenschaft *ColorSource* auf die entsprechende Spalte gesetzt. In der Datenmenge müssen die Farben dann als RGB-Werte vorliegen.

- SeriesColor (Eigenschaft, veröffentlicht)

  ```
  property SeriesColor : TColor
  ```

  Standardfarbe der Reihenkomponente.

- Cursor (Eigenschaft, veröffentlicht)

```
property Cursor : TCursor;
```

  Die Eigenschaft *Cursor* legt fest, welches Aussehen der Mauszeiger haben soll, wenn er sich exakt über der jeweiligen Reihenkomponente befindet.

- HorizAxis, VertAxis (Eigenschaften, veröffentlicht)

```
property HorizAxis: (aTopAxis,aBottomAxis);
property VertAxis: (aLeftAxis,aRightAxis);
```

  Die Eigenschaft *HorizAxis* bestimmt, ob die obere oder die untere horizontale Achse verwendet wird.

  Die Eigenschaft *VertAxis* bestimmt, ob die linke oder die rechte vertikale Achse verwendet wird.

- Marks (Eigenschaft, veröffentlicht), OnGetMarkText (Ereignis)

```
property Marks : TSeriesMarks;
property OnGetMarkText(Sender: TChartSeries; ValueIndex: Integer;
  var MarkText: string);
```

  Mit Hilfe der Objekteigenschaft *Marks* lässt sich die Art und Weise der Wertebeschriftung einstellen. Der genaue Aufbau dieser Eigenschaft lässt sich mit dem Objektinspektor ergründen, wesentlich ist vor allem die Eigenschaft *Marks.Style*.

  Vor der Ausgabe von Beschriftungen wird das Ereignis *OnGetMarkText* aufgerufen, dabei wird der String, der ausgegeben werden soll, als Parameter *MarkText* übergeben. Durch Änderung dieses Parameters lässt sich der auszugebende Text ändern.

- PercentFormat (Eigenschaft, veröffentlicht)

```
property PercentFormat : String;
```

  Stellt ein, wie Prozentangaben bei der Reihenbeschriftung formatiert werden. Im folgenden Beispiel werden diese Angaben mit einer Nachkommastelle formatiert:

```
Series7.PercentFormat := '##0.# %';
```

- ValueFormat (Eigenschaft, veröffentlicht)

```
property ValueFormat : String;
```

  Mit Hilfe der Eigenschaft *ValueFormat* legt man fest, wie die angezeigten Zahlen formatiert werden sollen. Nach meiner Erfahrung muss ein wenig experimentiert werden, bis die Reihenkomponente wirklich das tut, was man möchte.

- ColorRange (Methode)

```
procedure ColorRange(AValueList: TChartValueList;
  const FromValue, ToValue: Double; AColor: TColor);
```

Bisweilen sollen Teile einer Grafik farblich anders gekennzeichnet werden – beispielsweise, wenn das Wahlergebnis unter 5% liegt. Mit Hilfe der Prozedur *ColorRange* kann einem Wertebereich eine andere Farbe zugewiesen werden.

```
Series5.ColorRange(Series5.YValues, 0, 5, clred);
```

- VisibleCount (Methode)

```
function VisibleCount: Longint;
```

Wenn die Reihenkomponenten nicht automatisch skaliert werden, kann es vorkommen, dass nicht alle Punkte angezeigt werden. Die Funktion *VisibleCount* ermittelt die Anzahl der angezeigten Punkte.

- XValueToText, YValueToText (Methoden)

```
function XValueToText(Const AValue: Double): string;
function XValueToText(Const AValue: Double): string;
```

Diese Funktionen wandeln einen Wert in einen String um und verwenden dabei dieselben Formatierungsregeln, wie sie bei der Reihenkomponente verwendet werden.

- OnAfterDrawValues, OnBeforeDrawValues (Ereignisse)

```
property OnAfterDrawValues(Sender: TObject);
property OnAfterDrawValues(Sender: TObject);
```

Diese beiden Ereignisse treten vor und nach dem Zeichnen der Reihenpunkte auf und können beispielsweise dafür genutzt werden, eigene Grafikausgaben auf *Chart.Canvas* vorzunehmen. Achten Sie darauf, dass Sie dabei keine Eigenschaften der Komponente ändern oder eine ihrer Prozeduren aufrufen, da dies ein erneutes Zeichnen der Werte auslösen könnte – eine Rekursion wäre die Folge.

9.3.2 Anzeigewerte

- XValues, YValues (Eigenschaften, veröffentlicht)

```
property XValues:TChartValueList
property YValues:TChartValueList
```

  Die Objekteigenschaften *XValues* und *YValues* regeln, woher die Reiheneigenschaft ihre Daten bezieht sowie einige damit zusammenhängende Details. *TChartValueList* ist im folgenden Abschnitt beschrieben.

- xValue, yValue (Eigenschaften, öffentlich, nur Lesen)

```
property xValue[Index:Longint] : Double;
property yValue[Index:Longint] : Double;
```

  Mit Hilfe der Leseeigenschaften *xValue* und *yValue* kann man zur Laufzeit auf die angezeigten Werte zugreifen.

- DataSource (Eigenschaft, veröffentlicht), CheckDataSource (Methode)

```
property DataSource : TComponent;
procedure CheckDataSource(Series: TChartSeries);
```

  Die Eigenschaft *DataSource* wird benötigt, um Daten aus einer Datenmengenkomponente zu beziehen.

  Werden die Reihenpunktwerte aus einer Datenmenge oder aus einer anderen Reihenkomponente bezogen, dann bewirkt *CheckDataSource*, dass diese Werte erneut eingelesen werden.

- FirstValueIndex, LastValueIndex (Eigenschaften, öffentlich, nur Lesen)

```
property FirstValueIndex : LongInt;
property LastValueIndex : LongInt;
```

  Wenn die Achsen nicht automatisch skaliert werden, dann kann es – gewollt oder ungewollt – vorkommen, dass nicht die komplette Funktion dargestellt wird. *FirstValueIndex* ist dann der Index des ersten Wertes, der dargestellt wird, *LastValueIndex* der Index des letzten Wertes, der noch dargestellt wird. Werden die Achsen automatisch skaliert, dann nimmt die erste Eigenschaft stets den Wert *null*, die zweite den Wert der Anzahl der verfügbaren Werte -1 an.

- FunctionType (Eigenschaft, öffentlich)

```
property FunctionType : TTeeFunction
```

  Reihenkomponenten können ihre Daten auch aus Funktionen beziehen. Im Beispielprojekt haben die Komponenten *Series7* und *Series8* ihre Daten aus

Query1 bezogen. Genauso wäre es möglich, dass *Series8* ihre Daten von *Series7* kopiert:

- AddXY, AddY (Methoden)

```
function AddXY(const AXValue, AYValue: Double;
  Const AXLabel: string; AColor: TColor) : Longint;
function AddY(const AYValue: Double;
  Const AXLabel: string; AColor: TColor): Longint;
```

Die Funktionen *AddXY* und *AddY* fügen der Werteliste neue Werte hinzu. Als Funktionsergebnis liefern sie den Index des neuen Wertes zurück. Die Beschriftung *AXLabel* und der Farbwert *AColor* sind optional. Bei der Funktion *AddY* wird der X-Wert – sofern er benötigt wird – von der Reihenkomponente berechnet.

- OnAfterAdd, OnBeforeAdd (Ereignisse)

```
property OnAfterAdd(Sender: TChartSeries; ValueIndex: Integer);
property OnBeforeAdd(Sender: TChartSeries): Boolean;
```

Die Ereignisse *OnAfterAdd* und *OnBeforeAdd* treten nach beziehungsweise vor dem Einfügen neuer Werte in die Werteliste auf.

Das Ereignis *OnBeforeAdd* ist kein Zeiger auf eine Prozedur, sondern ein Zeiger auf eine Funktion. Diese erlaubt es, Werte von der Aufnahme in die Werteliste auszuschließen. Um dies zu erreichen, ist das Funktionsergebnis auf *false* zu setzen. Das folgende Beispiel schließt pauschal alle Werte von der Aufnahme aus.

```
function TForm1.Series9BeforeAdd(Sender: TChartSeries): Boolean;
begin
  result := false;
end;
```

- AssignValues (Methode)

  ```
  procedure AssignValues(Source : TChartSeries);
  ```

  Mit der Funktion *AssignValues* wird die Werteliste einer anderen *TChartSeries*-Komponente kopiert.

- Clear, Delete (Methoden), OnClearValues (Ereignis)

  ```
  procedure Clear;
  procedure Delete(ValueIndex: LongInt);
  property OnClearValues(Sender: TChartSeries);
  ```

  Die Prozedur *Clear* löscht die Werteliste der Reihenkomponente. Wenn für eine neue Werteliste Werte mit *AddY* oder *AddXY* hinzugefügt werden, sollte vorher *Clear* aufgerufen werden.

  Mit *Delete* wird der angegebene Reihenpunktwert gelöscht.

  Das Ereignis *OnClearValues* teilt Ihnen das Löschen der Punktwerte mit – verhindern lässt es sich allerdings nicht.

- Count (Methode)

  ```
  function Count : Longint;
  ```

  Die Funktion *Count* ermittelt die Gesamtzahl aller Reihenpunktwerte.

- XLabel (Eigenschaft, öffentlich), XLabelsSource (veröffentlicht)

  ```
  property XLabel[Index:Longint] : String;
  property XLabelsSource : String;
  ```

  XLabel ist das Array der horizontalen Reihenbeschriftungen.

  Sollen die horizontalen Achsenbeschriftungen aus einer Datenmenge übernommen werden, dann müssen in der Eigenschaft *XLabelsSource* deren Spaltennamen angegeben werden.

TChartValueList

Die Werte für die Darstellung in der Reihenkomponente werden in Wertelisten gehalten, die vom Typ *TChartValueList* sind. Dabei wurde eine etwas ungewöhnliche, aber geradezu geniale Konstruktion verwendet: Die Wertelisten sind veröffentlicht, enthalten aber eine öffentliche Default-Array-Eigenschaft. Im Quelltext

sieht es dann so aus, als ob diese Wertelisten (öffentliche) Array-Eigenschaften sind, bestimmte Eigenschaften dieser Wertelisten lassen sich aber trotzdem mit dem Objektinspektor einstellen.

- Value (Eigenschaft, öffentlich)

  ```
  property Value[Index: Longint]: Double;
  ```

  Mit der Eigenschaft *Value* können Werte in die Werteliste geschrieben und von dort ausgelesen werden. *Value* ist die Default-Eigenschaft von *TChartValueList*, die beiden folgenden Zeilen bewirken somit dasselbe:

  ```
  LineSeries1.YValues.Value[3] := 17;
  LineSeries1.YValues[3] := 17;
  ```

- Count (Methode)

  ```
  function Count: Longint;
  ```

  Die Funktion *Count* ermittelt die Anzahl der Werte in der Werteliste.

- ModifyValue (Methode)

  ```
  procedure ModifyValue(ValueIndex: Longint; const AValue:Double);
  ```

  Wird ein Wert in der Werteliste geändert, so wird daraufhin die Darstellung aktualisiert. Soll diese Aktualisierung unterbleiben – beispielsweise, weil aus Geschwindigkeitsgründen für mehrere Änderungen nur eine Aktualisierung erfolgen soll –, dann kann dafür die Methode *ModifyValue* verwendet werden.

- DateTime (Eigenschaft, veröffentlicht)

  ```
  property DateTime : Boolean;
  ```

  Wenn die Eigenschaft *DateTime* auf *true* gesetzt wird, dann werden die Werte der Werteliste als *TDateTime*-Werte interpretiert und bei der Ausgabe als Text entsprechend formatiert.

- Multiplier (Eigenschaft, veröffentlicht)

  ```
  property Multiplier : Double;
  ```

  Die Achsen werden von der *TChart*-Komponente und für alle Reihenkomponenten einheitlich gesetzt. Dies erzwingt es manchmal, die Werte der verschiedenen Reihenkomponenten an einander anzugleichen. Die Eigenschaft *Multiplier* enthält einen Gleitkommawert, mit dem alle Werte der Werteliste verrechnet werden – auch die in der Legende angezeigten Werte werden entsprechend angepasst.

- Order (Eigenschaft, veröffentlicht)

  ```
  property Order: (loNone, loAscending, loDescending);
  ```

  Die Einträge in der Werteliste können aufsteigend (*loAscending*), absteigend (*loDescending*) oder überhaupt nicht sortiert werden (*loNone*). Sorgen Sie dafür, dass die Einträge nicht sortiert werden, wenn Sie bei *TGanttSeries* die Balken miteinander verbinden wollen.

- Name (Eigenschaft, veröffentlicht)

  ```
  property Name : String;
  ```

  Diese Eigenschaft beinhaltet den Namen der Werteliste als String und wird vor allem für *ValueSource*-Zuweisungen benötigt.

- ValueSource (Eigenschaft, veröffentlicht)

  ```
  property ValueSource : String;
  ```

  Bezieht die Reihenkomponente ihre Werte aus einer anderen Reihenkomponente, dann wird hier der Name der Werteliste eingetragen, von welcher die Daten kopiert (oder was auch immer) werden.

  ```
  Series2.DataSource := Series2;
  Series2.YValues.ValueSource := 'Y':
  Series2.XValues.ValueSource := 'X';
  ```

  Die Komponente *TDBChart* kann ihre Daten auch aus einer Datenmenge beziehen:

  ```
  Series1.DataSource := Query1;
  Series2.YValues.ValueSource := 'Umsatz':
  Series2.XValues.ValueSource := 'Datum';
  ```

9.3.3 Werteberechnungen

- CalcXPos, CalcYPos (Methoden)

  ```
  function CalcXPos(ValueIndex: Longint): LongInt;
  function CalcYPos(ValueIndex: Longint): LongInt;
  ```

  Berechnet, an welcher horizontalen beziehungsweise vertikalen Position der *TChart*-Komponente der jeweilige Punkt der Werteliste angezeigt wird.

- CalcXPosValue, CalcYPosValue (Methoden)

```
function CalcXPosValue(const Value: Double): Longint;
function CalcYPosValue(const Value: Double): Longint;
```

Diese Funktionen rechnen einen Skalenwert der X-Achse beziehungsweise der Y-Achse in die entsprechende horizontale beziehungsweise vertikale Position der *TChart*-Komponente um.

- CalcXSizeValue, CalcYSizeValue (Methoden)

```
function CalcXSizeValue(const Value: Double): Longint ;
function CalcYSizeValue(const Value: Double): Longint;
```

Die Methoden *CalcXSizeValue* und *CalcYSizeValue* rechnen eine Gleitkomma-Größe in die entsprechenden Bildschirmpixel um.

- OnClick, OnDblClick (Ereignisse)

```
property OnClick(Sender: TChartSeries; ValueIndex: Integer;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
property OnDblClick(Sender: TChartSeries; ValueIndex: Integer;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
```

Wird ein Mausklick auf einer *TChart*-Komponente ausgeführt, dann wird ermittelt, ob sich an dieser Stelle eine Reihenkomponente befindet. Ist dies der Fall, dann wird das Ereignis *OnClick* ausgelöst. Werden an der betreffenden Stelle mehrere Reihenkomponenten angezeigt, so tritt das Ereignis nur bei der zuoberst angezeigten Reihenkomponente ein.

Für das Ereignis *OnDblClick* gelten diese Ausführungen entsprechend.

- Clicked (Methode)

```
function Clicked(x, y: Integer): Longint;
```

Diese Funktion ermittelt, welcher Reihenpunktwert an der angegebenen Position liegt, und gibt dessen Index zurück. Liegt an der fraglichen Stelle kein Reihenpunktwert, dann gibt die Funktion *-1* zurück.

- GetCursorValueIndex (Methode)

```
function GetCursorValueIndex: LongInt;
```

Die Funktion *GetCursorValueIndex* ermittelt, welcher Reihenpunktwert an der aktuellen Cursor-Position liegt, und gibt dessen Index zurück. Liegt an der fraglichen Stelle kein Reihenpunktwert, gibt die Funktion *-1* zurück.

- GetCursorValues (Methode)

```
procedure GetCursorValues(var  x , y: Double);
```

  Die Prozedur rechnet die aktuelle Cursor-Position in entsprechende Realwerte gemäß den Achsenskalierungen um.

- MaxXValue, MaxYValue, MinXValue, MinYValue (Methoden)

```
function MaxXValue : Double;
function MaxYValue : Double;
function MinXValue : Double;
function MinYValue : Double;
```

  Die aufgezählten Funktionen ermitteln die Maxima und die Minima der beiden Wertelisten.

- RefreshSeries (Methode)

```
procedure RefreshSeries;
```

  Die Methode *RefreshSeries* bewirkt, dass alle abhängigen Reihen – also alle Reihenkomponenten, die ihre Daten von dieser Reihenkomponente beziehen – ihre Punkte neu berechnen.

- SwapValueIndex (Methode)

```
procedure SwapValueIndex(a, b: Longint );
```

  Um zwei Reihenpunktwerte gegeneinander zu vertauschen, wird die Prozedur *SwapValueIndex* verwendet. Als Parameter sind die Indizes der Reihenpunktwerte zu übergeben, die vertauscht werden sollen.

- ValuesListCount (Methode)

```
function ValuesListCount: Longint;
```

  Die Funktion *ValuesListCount* ermittelt die Zahl der Reihenpunktlisten. Für gewöhnlich hat eine Reihenkomponente zwei Reihenpunktlisten (x und y), Reihenkomponenten des Typs *TBubbleSeries* (x, y, Radius) und *TGanttSeries* (x, Start, End) haben deren drei.

- XScreenToValue, YScreenToValue (Methoden)

```
function XScreenToValue(ScreenPos:Longint):Double;
function YScreenToValue(ScreenPos:Longint):Double;
```

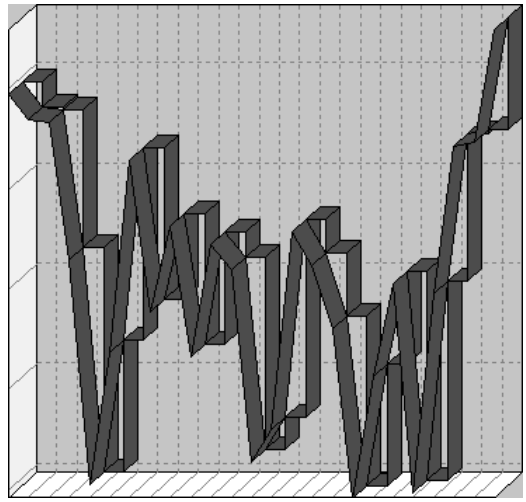
  Diese Funktionen rechnen einen Punkt auf der Komponente in den entsprechenden Realwert um.

9.4 Die Reihenkomponenten

Die folgenden Reihenkomponenten können instantisiert werden. Sie sind alle (direkt oder indirekt) von der in Kapitel 9.3 beschriebenen abstrakten Reihenkomponente *TChartSeries* abgeleitet.

9.4.1 TLineSeries

Die Komponente *TLineSeries* stellt die Punktwerte in einem Achsenkreuz dar und verbindet die einzelnen Punkte mit Linien.



- Stairs, InvertedStairs (Eigenschaften, veröffentlicht)

  ```
  property Stairs: boolean;
  property InvertedStairs: boolean;
  ```

  Damit die Funktion treppenförmig dargestellt wird, muss die Eigenschaft *Stairs* auf *true* gesetzt werden. Die Treppenstufe wird nach (*InvertedStairs* = *false*) oder vor (*InvertedStairs* = *true*) dem jeweiligen Wert angezeigt.

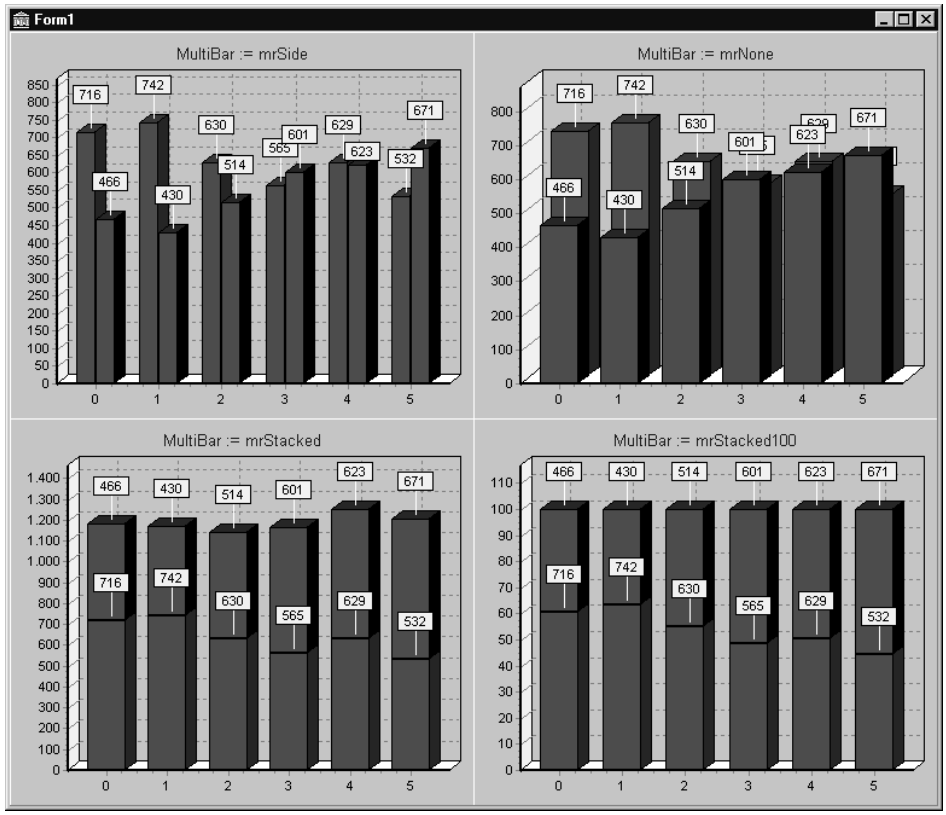
- LineBrush, LinePen (Eigenschaften, veröffentlicht)

  ```
  property TLineBrush: TBrush;
  property TLinePen: TPen;
  ```

  Mit Hilfe dieser Eigenschaften kann das Aussehen der Linien beeinflusst werden. Die Eigenschaft *LineBrush* ist nur bei 3-D-Darstellung wirksam.

9.4.2 TBarSeries und THorizBarSeries

Soll ein Balkendiagramm mit senkrechten Balken gezeichnet werden, muss man die Komponente *TBarSeries* verwenden – für waagerechte Balken verwendet man *THorizBarSeries*.



- MultiBar (Eigenschaft, veröffentlicht)

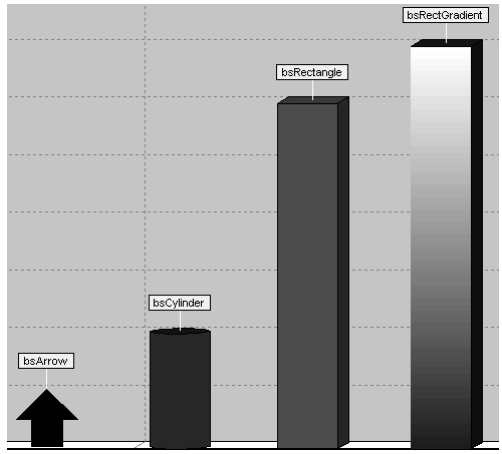
  ```
  property MultiBar : (mbNone, mbSide, mbStacked, mbStacked100);
  ```

  Werden mehrere *TBarSeries*-Reihenkomponenten in einer *TChart*-Komponente angezeigt, dann kann mit Hilfe der Eigenschaft *MultiBar* bestimmt werden, wie diese angeordnet werden sollen. Die vier möglichen Werte von *MultiBar* sind in oben stehender Abbildung mitsamt entsprechenden Beispielen zu sehen.

- BarStyle (Eigenschaft, veröffentlicht)

  ```
  property BarStyle : TBarStyle;
  ```

  Mit Hilfe der Eigenschaft *BarStyle* lässt sich das Aussehen des Balkens verändern. Einige zulässige Werte von *BarStyle* (*bsEllipse*, *bsInvPyramid*, *bsPyramid*) werden jedoch wie bei *bsRectangle* angezeigt.

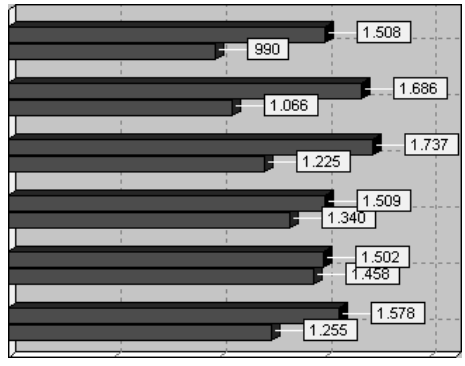


- BarBrush, BarPen (Eigenschaft, veröffentlicht)

  ```
  property BarBrush: TBrush;
  property BarPen: TPen;
  ```

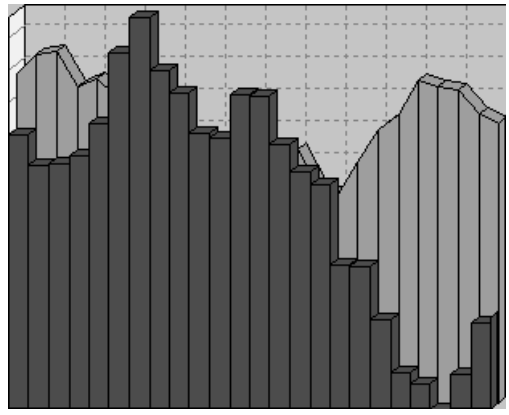
  Mit Hilfe dieser Eigenschaften kann das Aussehen der Balken bestimmt werden.

Hier noch eine Abbildung zu *THorizBarSeries*:



9.4.3 TAreaSeries

Die Komponente *TAreaSeries* ähnelt *TLineSeries*, die beiden Komponenten unterscheiden sich jedoch dadurch, dass die Fläche unterhalb der Linie bei *TAreaSeries* gefüllt wird.



- Stairs, InvertedStairs (Eigenschaften, veröffentlicht)

  ```
  property Stairs: boolean;
  property InvertedStairs: boolean;
  ```

  Damit die Funktion treppenförmig dargestellt wird, muss die Eigenschaft *Stairs* auf *true* gesetzt werden. Die Treppenstufe wird nach (*InvertedStairs* = *false*) oder vor (*InvertedStairs* = *true*) dem jeweiligen Wert angezeigt.

- MultiArea (Eigenschaft, veröffentlicht)

  ```
  property MultiArea: (maNone, maStacked, maStacked100);
  ```

  Auch bei *TAreaSeries* lassen sich die einzelnen Reihenkomponenten übereinandersetzen, *MultiArea* kann dabei die Werte *maNone*, *maStacked* und *maStacked100* annehmen.

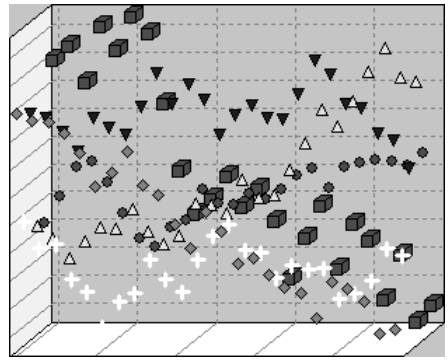
- AreaLinesPen (Eigenschaft, veröffentlicht)

  ```
  property AreaLinesPen : TChartPen;
  ```

  Wie in Bild 7.16 zu sehen, werden die einzelnen Werte durch senkrechte Linien voneinander getrennt. Mit Hilfe der Eigenschaft *AreaLinesPen* lassen sich die Linien detailliert gestalten. Sollen diese Linien nicht angezeigt werden, muss die Eigenschaft *AreaLinesPen.Visible* auf *false* gesetzt werden.

9.4.4 TPointSeries

Die Komponente *TPointSeries* bewirkt die Darstellung als Einzelpunkte.



- Pointer (Eigenschaft, veröffentlicht)

  ```
  property Pointer: TSeriesPointer;
  ```

  Alle relevanten Eigenschaften sind in der Objekteigenschaft *Pointer* zusammengefasst – einige *TSeriesPointer*-Eigenschaften sollen hier nun aufgeführt werden:

  - Style, Draw3D (Eigenschaften, veröffentlicht)

    ```
    property Style: (psRectangle, psCircle, psTriangle,
        psDownTriangle, psCross, psDiagCross, psStar, psDiamond,
        psSmallDot) default psRectangle;
    ```

    Die Eigenschaft *Style* bestimmt das Aussehen der einzelnen Punkte. Bei manchen Style-Werten ist eine 3-D-Darstellung möglich – andere, wie beispielsweise *psCross*, ignorieren diese Eigenschaft.

    Vergessen Sie nicht, dass die Eigenschaft Style eine Untereigenschaft von *Point* ist.

    ```
    Series1.Pointer.Style := psTriangle;
    ```

  - HorizSize, VertSize (Eigenschaften, veröffentlicht)

    ```
    property HorizSize : Integer;
    property VertSize : Integer;
    ```

    Wie die Abbildung zeigt, haben die verschiedenen Punktformen zwar formal dieselbe Größe, scheinen aber unterschiedlich gewichtig. Mit Hilfe der Eigenschaften *HorizSize* und *VertSize* lässt sich deren Größe genau einstellen.

- Brush, Pen (Eigenschaften, veröffentlicht)

```
property Brush: TBrush;
property BarPen: TPen;
```

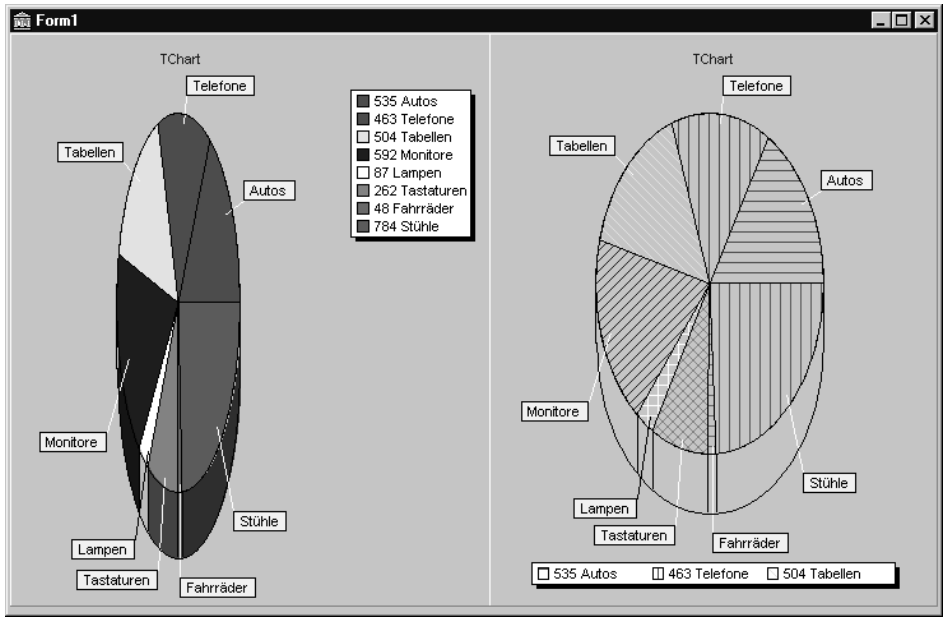
  Mit Hilfe dieser Eigenschaften kann man das Aussehen der Punkte beeinflussen.

9.4.5 TPieSeries

Zur Erstellung von Torten-Grafiken wird die Komponente *TPieSeries* verwendet. Wie die Abbildung zeigt, gibt es dabei ein kleines Problem: Wenn die *TChart*-Komponente eher hoch ist, sieht die »Torte« etwas seltsam aus. Setzt man die Eigenschaft *Circled* auf *true*, dann wird zwar ein Kreis angezeigt, dieser gerät jedoch dann unter Umständen sehr klein.

Prinzipiell könnte man die Darstellung dadurch verbessern, dass man die Legende nicht rechts, sondern unten platziert. Dann aber werden möglicherweise nicht alle Werte in der Legende aufgeführt, weil an dieser Stelle eine mehrzeilige Legende nicht möglich ist.

Deshalb sollte man dafür sorgen, dass die *TChart*-Komponente über ein sinnvolles Höhe-zu-Breite-Verhältnis verfügt. Bei Formularen, die sich in der Größe ändern lassen, sind entsprechende Vorkehrungen zu treffen, um Verzerrungen zu vermeiden.



- UsePatterns (Eigenschaft, veröffentlicht)

  ```
  property UsePatterns: Boolean
  ```

  Für Ausdrucke steht nicht immer ein Farbdrucker zur Verfügung. Um die Tortensegmente zu mustern, wie dies in der Abbildung rechts gezeigt wird, muss die Eigenschaft *UsePatterns* auf *true* gesetzt werden.

- RotationAngle (Eigenschaften, veröffentlicht), Rotate (Methode)

  ```
  property RotationAngle: Integer;
  procedure Rotate(Angle: Integer);
  ```

  Mit Hilfe der Eigenschaft *RotationAngle* lässt sich die Torte in die gewünschte Stellung drehen. Soll die Torte zur Laufzeit rotieren, dann wird die Methode *Rotate* vom *OnTimer*-Ereignis der Komponente *TTimer* aufgerufen. Ist eine Drehung im Uhrzeigersinn, also im mathematisch negativen Drehsinn erwünscht, dann muss als Parameter ein negativer Wert übergeben werden.

- Circled (Eigenschaft, veröffentlicht)

  ```
  property Circled : Boolean;
  ```

  Um eine Darstellung als Kreis zu erzwingen, wird die Eigenschaft *Circled* auf *true* gesetzt.

- CustomXRadius, CustomYRadius (Eigenschaften, veröffentlicht)

  ```
  property CustomXRadius : LongInt;
  property CustomYRadius : LongInt;
  ```

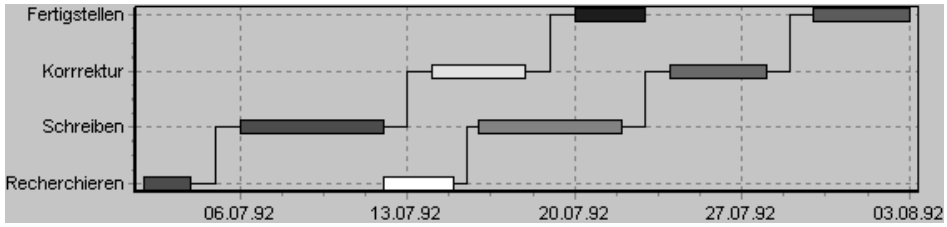
  Per Voreinstellung haben diese beiden Eigenschaften den Wert *null*, die Größe der Ellipse wird somit von der Reihenkomponente berechnet. Um die Größe der Ellipse explizit zu setzen, werden diesen beiden Eigenschaften die gewünschten Werte zugewiesen.

9.4.6 TFastLineSeries

Die Komponente *TFastLineSeries* gibt die Werte als Funktionskurven aus. Sie unterscheidet sich insofern von *TLineSeries*, als viele Features – wie beispielsweise die 3-D-Darstellung – nicht implementiert sind. Dafür arbeitet diese Komponente jedoch sehr schnell.

9.4.7 Die Komponente TGanttSeries

Speziell für Projektverwaltungsprogramme wurde die Komponente *TGanttSeries* entwickelt. Hier lassen sich Termine anhand ihrer Start- und Enddaten darstellen und durch Linien verbinden.



- AddGantt (Methode)

```
function AddGantt(const AStart, AEnd, AY: double;
  const AXLabel: string): LongInt;
```

Mit Hilfe der Funktion *AddGantt* werden neue Balken eingefügt. Als Parameter werden der Start- und der Endtermin (oder andere Gleitkommazahlen), der Y-Wert und die Beschriftung übergeben. Als Funktionsergebnis erhält man den Index des neuen Balkens.

- AddGanttColor (Methode)

```
function AddGanttColor(const AStart, AEnd, AY: Double;
  const AXLabel: string; AColor: TColor): LongInt;
```

In den meisten Fällen möchte man Balken, die zusammengehören, mit einer einheitlichen Farbe versehen. Man könnte dazu für jeden Balken der Eigenschaft *ValueColor* einen entsprechenden Wert zuweisen. Einfacher geht es jedoch, wenn statt *AddGantt* die Methode *AddGanttColor* verwendet wird, der man gleich die gewünschte Farbe als Parameter übergeben kann.

- NextTask (Eigenschaft, veröffentlicht)

```
property NextTask : TChartValueList;
```

Mit Hilfe der Eigenschaft *NextTask* lassen sich die einzelnen Balken miteinander verbinden. Ignorieren Sie dabei die Deklaration und den Objektinspektor und gehen Sie folgendermaßen vor:

```
var
  t1, t2: integer;
begin
  t1 := Series1.AddGantt(StrToDate('2.7.92'), StrToDate('4.7.92'),
    0, 'Recherchieren');
  t2 := Series1.AddGantt(StrToDate('6.7.92'),
StrToDate('12.7.92'),
    1, 'Schreiben');
  Series1.NextTask[t1] := t2;
```

Auf diese Weise wird der zweite Balken mit dem ersten verbunden. Dazu noch ein Hinweis: Wenn Sie mehrere Balken miteinander verbinden möchten, dann kann es vorkommen, dass alle möglichen Balken miteinander verbunden sind, nur nicht die, welche Sie eigentlich verbinden wollten. Ursache dafür ist in der Regel, dass die X-Werte sortiert werden, so dass beim Einfügen neuer Balken der Indexwert einiger Balken geändert wird. Um das Sortieren zu vermeiden, setzen Sie die Eigenschaft *XValues.Order* auf *loNone*.

- ConnectingPen (Eigenschaft, veröffentlicht)

  ```
  property ConnectingPen : TChartPen;
  ```

  Zur Gestaltung der Verbindungslinien verwenden Sie die Objekteigenschaft *ConnectingPen*.

- EndValues, StartValues (Eigenschaften, veröffentlicht)

  ```
  property EndValues : TChartValueList;
  property StartValues : TChartValueList;
  ```

  In den Listen *StartValues* und *EndValues* werden die Start- und Endwerte der Balken gespeichert.

- Pointer (Eigenschaft, veröffentlicht)

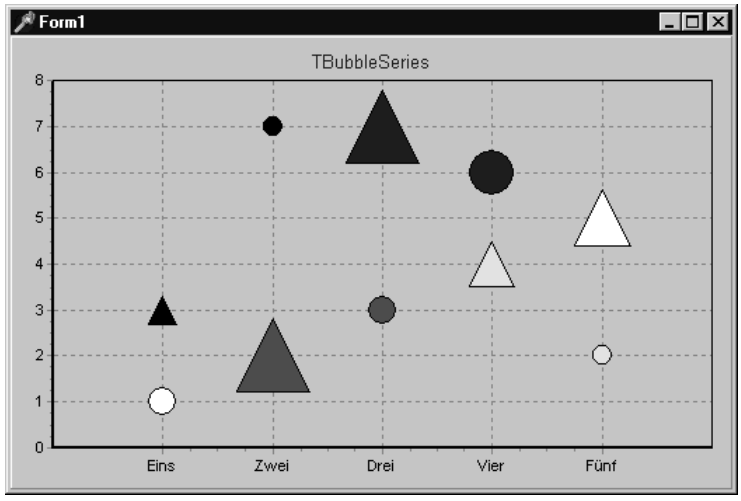
  ```
  property Pointer : TSeriesPointer;
  ```

  Zur Gestaltung der Balken wird die Objekteigenschaft *Pointer* verwendet, über deren Aufbau der Objektinspektor informiert. Interessant ist vor allem die Eigenschaft *VertSize*, mit der sich die Höhe der Balken einstellen lässt – die Eigenschaft *HorizSize* zeigt dagegen keine Auswirkung.

9.4.8 TBubbleSeries

Hin und wieder steht man vor dem Problem, Objekte in mehreren Größen gleichzeitig darstellen zu müssen. So könnte man sich beispielsweise vorstellen, dass in einer Quartalsübersicht nicht nur die Umsätze der einzelnen Filialen, sondern auch deren Gewinnspannen angezeigt werden sollen. Prinzipiell könnte man die zusätzliche Größe über eine farbliche Information visualisieren, diese Information geht aber beim Ausdruck, spätestens jedoch beim Vervielfältigen verloren.

Eine Alternative wäre hier die Komponente *TBubbleSeries*: Hier könnte man den Umsatz beispielsweise über die Y-Koordinate, die Gewinnspanne über den Durchmesser der einzelnen Funktionswerte darstellen.



- AddBubble (Methode)

```
function AddBubble(const AX, AY, ARadius: Double;
  const AXLabel: string; AColor: TColor): Longint;
```

Um die einzelnen Symbole hinzuzufügen, wird die Methode *AddBubble* verwendet. Als Parameter werden ihr die X- und Y-Koordinate sowie der Radius des Symbols übergeben, des Weiteren die Beschriftung und die zu verwendende Farbe.

Beachten Sie bitte, dass bei automatischer Achsenskalierung die Symbole an den Rändern beschnitten werden. Hier ist es dann gegebenenfalls erforderlich, die Grenzen manuell zu setzen.

- RadiusValues (Eigenschaft, veröffentlicht)

  ```
  property RadiusValues : TChartValueList;
  ```

  In der Werteliste *RadiusValues* werden die Radien der einzelnen Symbole gespeichert.

- Squared (Eigenschaft, veröffentlicht)

  ```
  property Squared : Boolean
  ```

  Für gewöhnlich werden X- und Y-Achsen verschieden skaliert. Würde nun die Radiusangabe exakt umgesetzt (*Squared = false*), dann würden keine Kreise, sondern Ellipsen angezeigt werden. Unter Umständen würden dabei inakzeptable Verzerrungen auftreten.

  Wird die Eigenschaft *Squared* auf *true* gesetzt, dann wird anhand der Y-Achsen-Skalierung der Radius berechnet und dann für beide Koordinaten verwendet.

- Pointer (Eigenschaft, veröffentlicht)

  ```
  property Pointer : TSeriesPointer;
  ```

  Zur Gestaltung der Symbole dient die Objekteigenschaft *Pointer*. Mit Hilfe der Eigenschaft *Pointer.Style* kann das verwendete Symbol gewählt werden, die möglichen Werte sind bei *TPointSeries* aufgezählt.

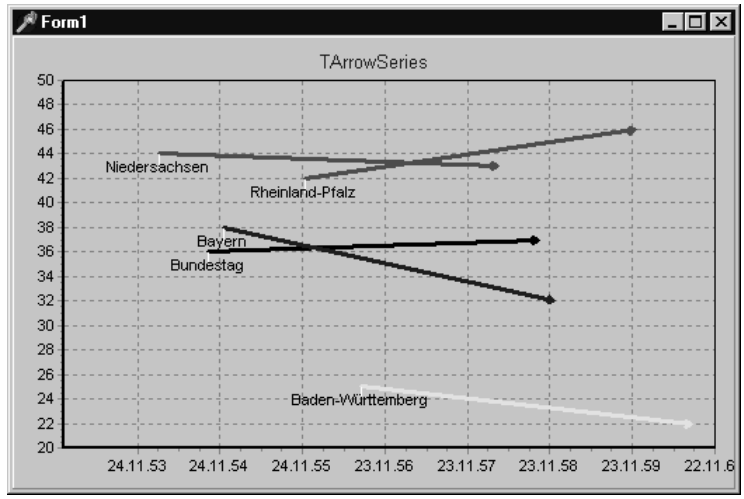
9.4.9 TArrowSeries

Zur Darstellung von Tendenzen eignet sich die Komponente *TArrowSeries*. Die Abbildung zeigt die Wahlergebnisse einer (wie immer fiktiven) Partei in einigen Bundesländern. Prinzipiell könnte man ein ähnliches Ergebnis auch mit entsprechend vielen *TLineSeries*-Komponenten erreichen. Bei *TArrowSeries* ist jedoch diese Darstellung mit einer einzelnen Komponente möglich, die zudem nicht nur durch Linien, sondern auch durch Pfeile dargestellt werden kann.

- AddArrow (Methode)

  ```
  function AddArrow(const X0, Y0, X1, Y1: Double;
    const ALabel: string; AColor: TColor): Longint;
  ```

  Als Parameter werden die Start- und die Endkoordinaten übergeben, des Weiteren die Beschriftung und die Farbe. Damit die Beschriftung vollständig angezeigt wird, müssen in der Regel die Achsen manuell skaliert werden.



- ArrowHeight, ArrowWidth (Eigenschaften, veröffentlicht)

```
property ArrowHeight : Integer;
property ArrowWidth : Integer;
```

Mit Hilfe der Eigenschaften *ArrowHeight* und *ArrowWidth* wird die Größe der Pfeilspitze in Pixeln eingestellt.

- EndXValues, EndYValues, StartXValues, StartYValues (Eigenschaften, veröffentlicht)

```
property EndXValues : TChartValueList;
property EndYValues : TChartValueList;
property StartXValues : TChartValueList;
property StartYValues : TChartValueList;
```

Diese Eigenschaften sind die Wertelisten der Anfangs- und Endkoordinaten.

- Pointer (veröffentlicht)

```
property Pointer : TSeriesPointer;
```

Zur Gestaltung der Pfeile wird die Objekteigenschaft *Pointer* verwendet. Mit Hilfe der Eigenschaft *Pointer.Pen.Width* kann die Dicke der Pfeillinie eingestellt werden.

9.4.10 TChartShape

Die Komponente *TChartShape* dient dazu, geometrische Formen auf der *TChart*-Komponente zu platzieren. Da *TChartShape* von *TChartSeries* abgeleitet ist, sind Methoden wie *AddXY* und Eigenschaften wie *XValues* vorhanden, sie haben jedoch keine Auswirkung auf das Erscheinungsbild der Formen.

- Style, RoundRectangle (Eigenschaften, veröffentlicht)

```
property Style: chasRectangle, chasCircle, chasVertLine,
  chasHorizLine, chasTriangle, chasInvertTriangle, chasLine,
  chasDiamond);
property RoundRectangle : Boolean;
```

  Mit Hilfe der Eigenschaft *Style* wird das Aussehen der Komponente definiert.

  Soll ein Rechteck mit abgerundeten Ecken angezeigt werden, dann muss die Eigenschaft *RoundRectangle* auf *true* gesetzt werden.

- X0, X1, Y0, Y1, XYStyle (Eigenschaften, veröffentlicht)

```
property X0 : Double;
property Y0 : Double;
property X1 : Double;
property Y1 : Double;
property XYStyle : TChartShapeXYStyle;
```

  Die Eigenschaften *X0* und *Y0* sowie *X1* und *Y1* enthalten die Koordinaten von gegenüberliegenden Ecken der Komponente. Mit Hilfe der Eigenschaft *XYStyle* bestimmt man, ob die Koordinaten in Pixel (*xysPixels*) oder in Skalenwerten (*xysAxis*) angegeben werden.

- Text, Font, Alignment (Eigenschaften, veröffentlicht)

```
property Text : TStrings;
property Font : TFont;
property Alignment : TAlignment;
```

  Innerhalb der Komponente kann mit der Eigenschaft *Text* mehrzeiliger Text ausgegeben werden. Auf diese Weise kann mit Hilfe von *TChartShape* das Diagramm beschriftet werden. Die dabei verwendete Schriftart wird mit der Eigenschaft *Font* eingestellt. Mit der Eigenschaft *Alignment* wird die Ausrichtung des Textes innerhalb der Komponente festgelegt.

9.5 TDBChart

Die Komponente *TDBChart* kann die Daten aus einer Datenbank beziehen, die Eigenschaft *DataSource* der enthaltenen Reihenkomponenten kann somit verwendet werden.

TDBChart erbt die eben beschriebenen Eigenschaften, Methoden und Ereignisse von *TChart*.

- AutoRefresh (Eigenschaft, veröffentlicht)

  ```
  property AutoRefresh: Booleann default true;
  ```

  Hat *AutoSize* den Wert *true*, dann werden automatisch die Daten eingelesen, wenn eine Datenmenge geöffnet wird. Hat *AutoSize* den Wert *false*, dann sind dafür die Methoden *RefreshDataSet* und *RefreshData* zu verwenden.

- RefreshDataSet, RefreshData (Methoden)

  ```
  procedure RefreshDataSet(ADataSet: TDataSet;
    ASeries: TChartSeries);
  procedure RefreshData;
  ```

  Mit *RefreshDataSet* werden die Daten für die angegebene Reihenkomponente eingelesen. Mit *RefreshData* werden die Daten für alle Reihenkomponenten eingelesen. Diese beiden Methoden müssen nur dann verwendet werden, wenn *AutoRefresh* den Wert *false* hat.

  Eine Ausnahme bilden hier Charts, die mit den Detailtabellen von Master-Detail-Verknüpfungen verbunden sind. Hier muss im *OnScroll*-Ereignis der Master-Datenquelle die Aktualisierung manuell aufgerufen werden.

- RefreshInterval (Eigenschaft, veröffentlicht), CheckDataSource (Methode)

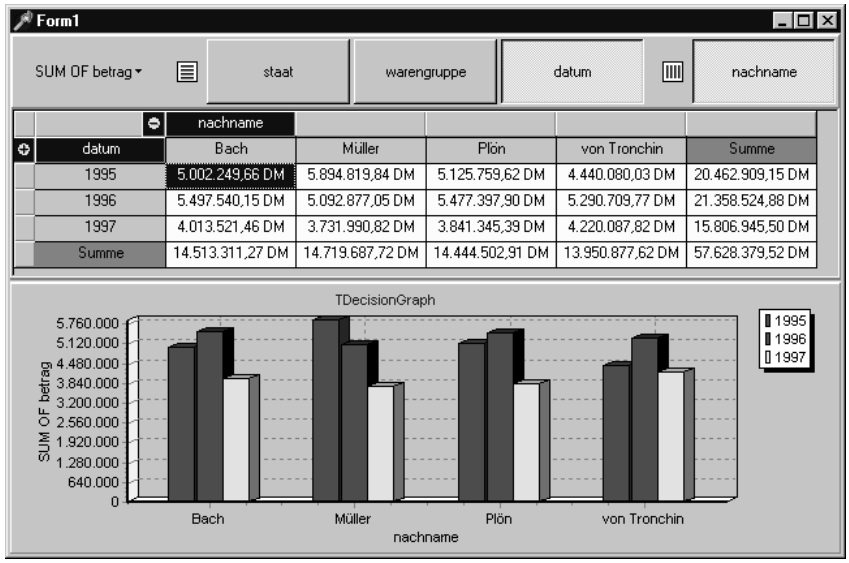
  ```
  property RefreshInterval: LongInt default 0;
  procedure CheckDataSource(ASeries: TChartSeries);
  ```

  Hat *RefreshInterval* einen Wert größer null, so wird der angegebene Wert einem Timer zugewiesen, der – sofern *AutoRefresh* den Wert *true* hat – die Daten neu einliest und die Anzeige aktualisiert. Damit kann man so etwas wie eine Echtzeitanzeige basteln.

  Mit *CheckDataSource* können Sie das Einlesen der Daten und das Aktualisieren der Anzeige manuell auslösen.

9.6 TDecisionGraph

Die Komponente *TDecisionGraph* dient zur grafischen Darstellung der Daten, die von Datenanalysekomponenten bereitgestellt werden. Diese Komponenten sind in Kapitel 9.8 beschrieben.



TDecisionGraph erbt die vorhin beschriebenen Eigenschaften, Methoden und Ereignisse von *TChart*.

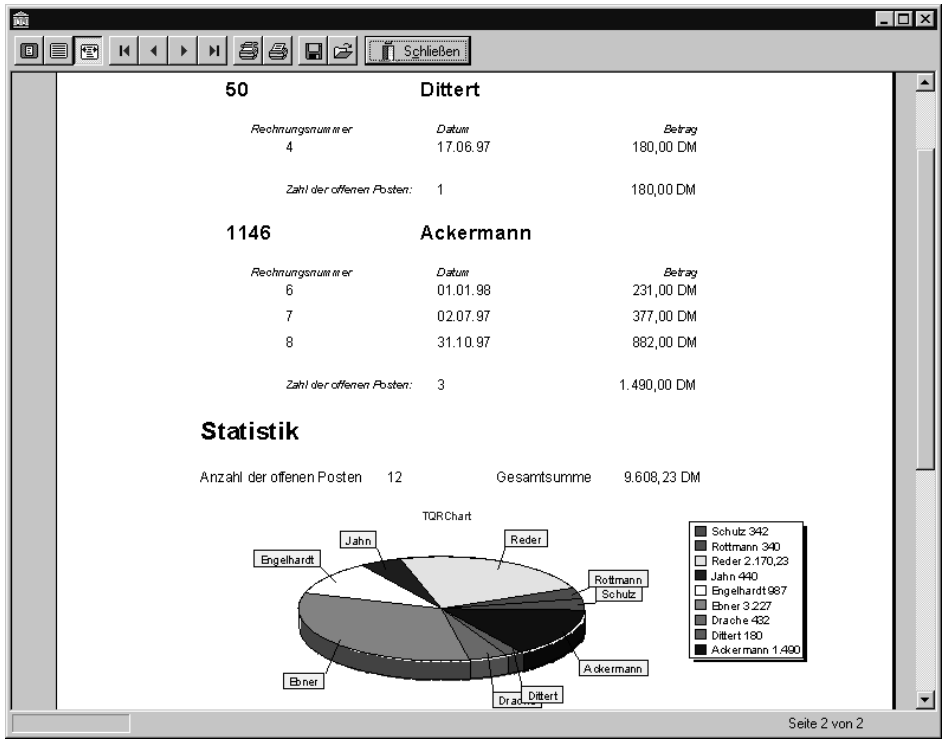
- DecisionSource (Eigenschaft, veröffentlicht)

  ```
  property DecisionSource: TDecisionSource;
  ```

  Über die Eigenschaft *DecisionSource* wird die Komponente mit einer *TDecisionSource*-Instanz verbunden, von der sie die Daten bezieht.

9.7 TQRChart

Zum Darstellen von Diagrammen in einem QuickReport dient die Komponente *TQRChart*.



- Chart (Eigenschaft, veröffentlicht)

  ```
  property Chart: TQRDBChart;
  ```

  Über die Eigenschaft *Chart* wird eine Diagrammkomponente integriert. *TQRDBChart* erbt die vorhin beschriebenen Eigenschaften, Methoden und Ereignisse von *TChart*.

9.8 Die Komponenten zur Datenanalyse

Die Komponenten zur Datenanalyse dienen zur Darstellung von Daten in Tabellenform oder als Diagramm. Während bei herkömmlicher Darstellung die Daten jedoch nur von ein oder zwei Dimensionen dargestellt werden konnten, ermöglichen die Komponenten zur Datenanalyse die Darstellung von mehreren Dimensionen. So ist es beispielsweise möglich, Umsätze nach dem Zeitraum, dem Verkäufer, dem Kunden und/oder dem Produkt darzustellen.

Die Komponenten zur Datenanalyse sind nur in der Enterprise-Version (bei Delphi 3 und Delphi 4 nur in der Client/Server-Suite) vorhanden.

9.8.1 Ein Beispielprojekt

Da die Komponenten zur Datenanalyse fast immer im Verbund eingesetzt werden, soll hier mit einem Beispielprojekt begonnen werden, welches das Zusammenwirken der einzelnen Komponenten illustriert:

Warum die Komponenten zur Datenanalyse?

Reale Vorgänge umfassen in der Regel viele Dimensionen. Nehmen wir als Beispiel die Umsatzübersicht einer Firma: Zunächst einmal interessiert es, wie viel Umsatz pro Jahr, pro Verkäufer, pro Warengruppe oder pro Land gemacht wurde. Diese Daten kann man mit Hilfe einfacher SQL-Abfragen ermitteln.

Sollen nun mehrere Dimensionen gleichzeitig dargestellt werden, dann benötigt man eine oder mehrere Master-Detail-Verknüpfungen. Auch dafür hat es in diesem Buch bereits Beispiele gegeben. Werden jedoch solche Verknüpfungen mit den herkömmlichen Komponenten erstellt, dann sind diese Gebilde reichlich unflexibel: Zur Entwurfszeit wird bereits festgelegt, welche Dimension die Hauptdimension ist und welche Dimensionen die Detail- oder Subdetaildimensionen sind. Zudem ist die gleichzeitige Anzeige mehrerer Dimensionen mit viel Programmieraufwand oder dem Einsatz der QuickReport-Komponenten verbunden.

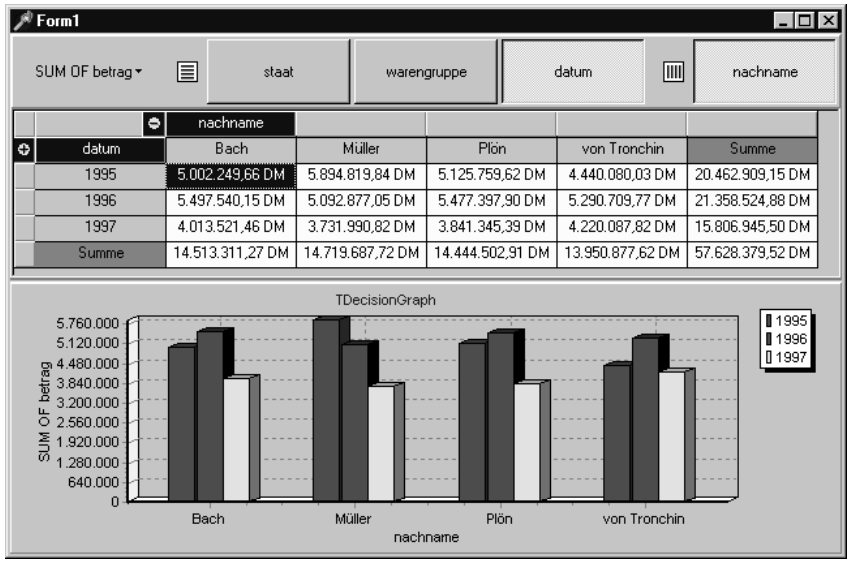
Wesentlich einfacher gestaltet sich ein solches Unterfangen mit den Komponenten zur Datenanalyse: Mit Hilfe von *TDecisionPivot* wird bestimmt, welche Dimensionen an welcher Stelle angezeigt werden. Um die Anzeige umzukonfigurieren, reichen zur Laufzeit wenige Mausklicks oder ein Verschieben mit Drag&Drop – die Anzeige wird dann nahezu verzögerungsfrei umgestellt.

Wo Licht ist, ist bekanntlich auch Schatten: Damit die Anzeige schnell umgestellt werden kann, müssen viele Daten im Speicher gehalten werden. Daraus resultiert ein nicht unerheblicher Speichermehrbedarf – bei unserem Beispielprojekt

etwa fünf MByte, wenn das Projekt direkt gestartet wird, und etwa zwölf Mbyte, wenn es unter Delphi läuft. Außerdem muss dieser Datenbestand beim Öffnen der Datenmenge erst einmal berechnet werden, also auch beim Programmstart.

Das Beispielprojekt

Für das Beispielprojekt wurde eine Tabelle mit den Umsatzzahlen einer fiktiven Firma generiert. Damit wenigstens halbwegs realistische Bedingungen gegeben sind, wurden dafür 5 000 Verkaufsdatensätze generiert.



Für das Beispielprojekt fügen Sie alle Komponenten aus der Palettenseite *Datenanalyse* je einmal in das Formular ein. Die Eigenschaft *DecisionSource* von *DecisionPivot1*, *DecisionGrid1* und *DecisionGraph1* wird auf *DecisionSource1* gesetzt, deren Eigenschaft *DecisionCube* den Wert *DecisionCube1* erhält.

Die Eigenschaft *DataSet* von *DecisionCube1* erhält den Wert *DecisionQuery1*. Dieser wiederum wird folgende SQL-Anweisung zugewiesen:

```
SELECT
     n.nachname,
     s.staat,
     w.warengruppe,
     v.datum,
     SUM(betrag),
     COUNT(betrag)
```

```
FROM f_name n, f_staat s, f_verkauf v, f_waren w
WHERE (n.nummer = v.verkäufer)
  AND (s.nummer = v.land)
  AND (w.nummer = v.warengruppe)
GROUP BY
  n.nachname,
  s.staat,
  w.warengruppe,
  v.datum
```

Die Komponente *TDecisionQuery* verfügt über einen Komponenteneditor zur Generierung der SQL-Anweisung, dieser ist jedoch nicht gerade auf die Erstellung von JOINs hin optimiert. Wenn Sie die Eigenschaft *Active* von *TDecisionQuery* auf *true* gesetzt haben, werden zunächst die Daten in den Speicher gelesen, was einige Zeit in Anspruch nehmen kann. Danach können Sie schon zur Entwurfszeit mit fast voller Funktionalität arbeiten.

9.8.2 TDecisionQuery

Die Komponente *TDecisionQuery* ist direkt von *TQuery* abgeleitet und implementiert keine weiteren Eigenschaften, Methoden oder Ereignisse. Lediglich ein Komponenteneditor zur Erzeugung der SQL-Anweisung ist hinzugekommen:

Mit einem Doppelklick auf *TDecisionQuery* wird der Komponenteneditor gestartet. Wählen Sie die Datenbank und die Tabelle und Ihnen werden daraufhin alle vorhandenen Spalten angezeigt. Mit Hilfe der Pfeil-Buttons werden einzelne Spalten als Dimensionen oder als Zusammenfassungen deklariert.

Als Zusammenfassungsfelder eignen sich insbesondere Umsatzzahlen und ähnliche Geldbeträge. Beim Einfügen in diese List-Box wählen Sie per Popupmenü aus, ob Sie davon die Summe, die Zahl oder den Mittelwert anzeigen lassen möchten. Sie können hier mehrere Zusammenfassungen gleichzeitig bilden. Mit Hilfe der Schaltfläche ganz links im *DecisionPivot* kann man zur Entwurfs- und zur Laufzeit zwischen den einzelnen Zusammenfassungen umschalten.

Zu Dimensionen erklären Sie alle Spalten, nach denen Sie die Daten gruppieren möchten. Sie sollten dabei einerseits möglichst viele Dimensionen verwenden, denn die Komponenten zur Datenanalyse werden erst so richtig interessant, wenn zwischen drei oder vier Dimensionen umgeschaltet werden kann. Andererseits steigt mit der Zahl der Dimensionen auch der Speicherbedarf und die Bearbeitungsdauer. Eine gewisse Aussagekraft sollten die verwendeten Spalten schon haben.

Wenn Sie die Registerseite wechseln, dann sehen Sie, dass automatisch eine SQL-Anweisung generiert wurde.

9.8.3 TDecisionCube

Die Komponente *TDecisionCube* speichert die von der Datenmenge erhaltenen Daten und gibt sie an *TDecisionSource* weiter. *TDecisionCube* ist von *TComponent* abgeleitet.

- DataSet (Eigenschaft, veröffentlicht)

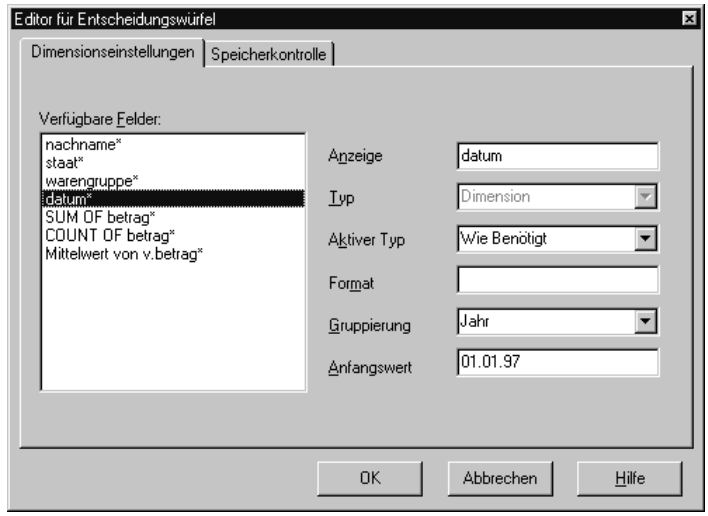
  ```
  property DataSet: TDataSet;
  ```

  Mit Hilfe von *DataSet* geben Sie die Datenmengenkomponente an, von welcher die Daten bezogen werden sollen. In der Regel wird dies eine *TDecisionQuery*-Instanz sein.

- DimensionMap (Eigenschaft, veröffentlicht), OnRefresh (Ereignis)

  ```
  property DimensionMap: TCubeDims;
  property OnRefresh(DataCube: TCustomDataStore; DimMap: TCubeDims);
  ```

  Mit Hilfe der Eigenschaft *DimensionMap* kann bestimmt werden, wie die einzelnen Dimensionen angezeigt werden. Die folgende Abbildung zeigt den dafür vorhandenen Eigenschaftseditor:



Wenn eine Ihrer Dimensionen ein Datum ist, dann möchten Sie dieses vielleicht nach dem Monat, dem Quartal oder dem Jahr gruppieren. Wählen Sie die entsprechende Einheit aus der ComboBox *Gruppierung*. Mit *Anzeige* können Sie die Bezeichnung der Dimension umbenennen, Spaltenbezeichnungen wie *SUM OF betrag* erfreut meist nur SQL-Kundige.

Bevor die neuen Einstellungen zugewiesen werden, wird das Ereignis *OnRefresh* ausgelöst.

- ShowProgressDialog (Eigenschaft, veröffentlicht)

```
property ShowProgressDialog: Boolean default true;
```

Das Einlesen der Daten benötigt meist einige Zeit, während der gemäß Voreinstellung eine Fortschrittsanzeige den Anwender darüber informiert, dass die Anwendung noch nicht abgestürzt ist. Soll das Einlesen der Daten im Hintergrund erfolgen, dann stört diese Fortschrittsanzeige und *ShowProgressDialog* ist auf *false* zu setzen.

- OnAfterClose, OnAfterOpen, OnBeforeClose, OnBeforeOpen (Ereignisse)

```
property OnAfterClose(DataCube: TCustomDataStore);
property OnAfterOpen(DataCube: TCustomDataStore);
property OnBeforeClose(DataCube: TCustomDataStore);
property OnBeforeOpen(DataCube: TCustomDataStore);
```

Mit diesen vier Ereignissen kann auf das Öffnen und Schließen des Kreuztabellenspeichers reagiert werden.

Speichernutzung

Die Daten, welche eine *TDecisionCube*-Instanz verwaltet, werden alle im Arbeitsspeicher gehalten. Der Bedarf an Speicher steigt mit der Potenz der verwendeten Dimensionen, von daher sollte man sich um eine effektive Speichernutzung Gedanken machen.

- MaxCells, MaxDimensions, MaxSummaries (Eigenschaften, veröffentlicht)

  ```
  property MaxCells: Integer default 0;
  property MaxDimensions: Integer default 5;
  property MaxSummaries: Integer default 10;
  ```

  Mit diesen drei Eigenschaften kann man die Zahl der Zellen, der Dimensionen und der Zusammenfassungen beschränken, die gleichzeitig im Speicher gehalten werden. Sofern die betreffende Eigenschaft den Wert null hat, findet keine Beschränkung statt.

- Capacity (Eigenschaft, öffentlich), OnLowCapacity (Ereignis)

  ```
  property Capacity: Integer;
  property OnLowCapacity(var EAction: TErrorAction);
  ```

  Mit *Capacity* wird vorgegeben, wie viele Bytes eine *TDecisionCube*-Instanz maximal an Speicher belegt. Beim Versuch, diese Grenze zu überschreiten, wird das Ereignis *OnLowCapacity* ausgelöst.

9.8.4 TDecisionSource

Die Komponente *TDecisionSource* leitet die Daten an *TDecisionGrid* und *TDecisionGraph* und die Steuerungsbefehle von *TDecisionPivot* weiter. *TDecisionSource* ist von *TComponent* abgeleitet.

Im Regelfall wird *TDecisionSource* lediglich eine Mittlerfunktion wahrnehmen, so dass ausschließlich die Eigenschaft *DecisionCube* gesetzt werden muss. Wenn Sie jedoch vom Programm aus auf die Daten zugreifen möchten, dann helfen die vielfältigen Methoden von *TDecisionSource* weiter, von denen nur ein geringer Teil hier besprochen wird.

- DecisionCube (Eigenschaft, veröffentlicht)

  ```
  property DecisionCube: TDecisionCube;
  ```

  Mit der Eigenschaft *DecisionCube* wird bestimmt, von welchem Kreuztabellenspeicher die Daten bezogen werden.

- nDims, nColDims, nRowDims, nOpenColDims, nOpenRowDims (Eigenschaften, öffentlich, nur Lesen)

```
property nDims: Integer;
property nColDims: Integer;
property nRowDims: Integer;
property nOpenColDims: Integer;
property nOpenRowDims: Integer;
```

  Mit *nDims* kann die Zahl der aktiven Dimensionen bestimmt werden. Wenn es von Interesse sein sollte, wie viele davon als Spalten beziehungsweise Reihen angezeigt werden, sind die Eigenschaften *nColDims* beziehungsweise *nRowDims* zu verwenden.

  Mit *nOpenColDims* kann die Anzahl der momentan »aufgeschlüsselten« Spaltendimensionen festgestellt werden, mit *nOpenRowDims* die Anzahl offener Reihendimensionen.

- nDataCols, nDataRows (Eigenschaften, öffentlich, nur Lesen)

```
property nDataCols: Integer;
property nDataRows: Integer;
```

  Mit *nDataCols* kann die Zahl der Spalten im Datenbereich ermittelt werden, mit *nDataRows* die Zahl der Reihen im Datenbereich. Eventuell angezeigte Summenspalten werden dabei mitgezählt.

- nSums, CurrentSum (Eigenschaften, öffentlich, nur Lesen)

```
property nSums: Integer;
property CurrentSum: Integer;
```

  Mit *nSums* kann ermittelt werden, wie viele Aggregatfunktionen zur Anzeige zur Verfügung stehen.

  Der Index der momentan angezeigten Aggregatfunktion kann mit *CurrentSum* ermittelt werden, die Zählung beginnt dabei bei null.

- GetDataAsString, GetDataAsVariant (Methoden)

```
function GetDataAsString(ARow, ACol:Integer;
  var SubLevel: Integer): string;
function GetDataAsVariant(ARow, ACol: Integer;
  var SubLevel: Integer): Variant;
```

  Die Daten eines jeden Datenfeldes lassen sich mit diesen beiden Funktionen als String oder als Variante ermitteln.

Mittels des Variablen-Parameters *SubLevel* kann dabei auch ermittelt werden, ob es sich beim Inhalt des Feldes um Daten (null) oder um Summen von Daten (eins) oder um Summen von Summen (zwei oder höher) handelt.

- GetDimensionName, GetMemberAsString, GetMemberAsVariant (Methoden)

```
function GetDimensionName(iDim: Integer): String;
function GetMemberAsString(iDim: Integer;
  ValueIndex: Integer): string;
function GetMemberAsVariant(iDim: Integer;
  ValueIndex: Integer): Variant;
```

Mit *GetDimensionName* kann die Bezeichnung einer Dimension (beispielsweise *Warengruppe*) ermittelt werden. Der Parameter *iDim* erwartet – wie auch bei den anderen beiden Funktionen – einen absoluten Index.

Mit *GetMemberAsString* und *GetMemberAsVariant* können die Spalten- oder Reihenüberschriften innerhalb einer Dimension (beispielsweise *Festplatten*) ermittelt werden. Der Index der gesuchten Spalte beziehungsweise Reihe wird als Parameter *ValueIndex* übergeben, die Zählung beginnt auch hier bei null.

- GetActiveDim, GetGroup, GetIndex (Methoden)

```
function GetActiveDim(dimGroup: TDimGroup; Index: Integer;
  bOpen: Boolean): Integer;
function GetGroup(iDim: Integer): TDimGroup;
function GetIndex(iDim: Integer; bOpen: Boolean): Integer;
```

TDecisionSource arbeitet mit zwei Index-Systemen, welche sich mit diesen Methoden konvertieren lassen. Das eine System ist das absolute Index-System, bei dem die einzelnen Dimensionen so durchgezählt werden, wie sie von *TDecisionCube* zur Verfügung gestellt werden. Das relative Index-System zählt die angezeigten Spalten und Reihen durch, wobei man noch unterscheiden kann, ob nur »aufgeschlüsselte« Dimensionen mitgezählt werden.

Mit *GetActiveDim* lässt sich der absolute Index auf einen relativen Index berechnen. Der Parameter *dimGroup* gibt dabei an, ob der Parameter *Index* sich auf eine Reihe (*dgRow*) oder eine Spalte (*dgCol*) bezieht. Hat *bOpen* den Wert *true*, dann werden bei der Ermittlung von *Index* nur »aufgeschlüsselte« Dimensionen mitgezählt.

Zur Berechnung eines relativen Index werden die Funktionen *GetGroup* und *GetIndex* verwendet. Mit *GetGroup* kann ermittelt werden, ob der als Parameter *iDim* übergebene absolute Index als Reihe (*dgRow*) oder als Spalte (*dgCol*) angezeigt wird. Zur Berechnung der Position in der Reihe oder Spalte wird dann *GetIndex* verwendet, wobei auch hier mittels *bOpen* spezifiziert werden kann, ob nur »aufgeschlüsselte« Dimensionen mitgezählt werden.

Ereignisse

- OnAfterPivot, OnBeforePivot, OnLayoutChange (Ereignisse)

```
property OnAfterPivot(Sender: TObject);
property OnBeforePivot(Sender: TObject);
property OnLayoutChange(Sender: TObject);
```

  Vor dem Aufschlüsseln oder Zusammenfassen einer Dimension tritt *OnBeforePivot* auf, danach *OnAfterPivot*.

  Das Ereignis *OnLayoutChange* tritt auf, nachdem die Darstellung geändert wurde, beispielsweise auch dann, wenn Dimensionen aufgeschlüsselt oder zusammengefasst wurden. *OnLayoutChange* tritt dabei vor *OnAfterPivot* auf.

- OnSummaryChange (Ereignis)

```
property OnSummaryChange(Sender: TObject);
```

  Wird eine andere Aggregatfunktion zur Anzeige ausgewählt, dann tritt *OnSummaryChange* auf.

9.8.5 TDecisionPivot

Mit der Komponente *TDecisionPivot* wird ausgewählt, welche Dimensionen und welche Zusammenfassungen angezeigt werden. *TDecisionPivot* ist von *TWinControl* abgeleitet.

Die Elemente einer *TDecisionPivot*-Instanz gliedern sich in drei Gruppen. Mit der ComboBox links können Sie wählen, welche Aggregatfunktion angezeigt wird – hier in der Abbildung die Summe.

Es folgt die Gruppe der Dimensionen, die untereinander angezeigt werden, ganz rechts die Gruppe der Dimensionen, die nebeneinander angezeigt werden. Die einzelnen Dimensionen werden aufgeschlüsselt, wenn der betreffende Button »gedrückt« dargestellt wird.

Mit Drag&Drop können die einzelnen Dimensionen von der Darstellung untereinander zur Darstellung nebeneinander sowie umgekehrt verschoben werden.

- DecisionSource (Eigenschaft, veröffentlicht)

```
property DecisionSource: TDecisionSource;
```

  Verwenden Sie diese Eigenschaft, um die Verbindung zu der *TDecisionSource*-Instanz herzustellen, welche die Komponente steuern soll.

- Groups (Eigenschaft, veröffentlicht)

```
property Groups: set of (xtRows, xtColumns, xtSummaries)
  default [xtRows, xtColumns, xtSummaries];
```

Mit der Eigenschaft *Groups* kann spezifiziert werden, welche Buttons auf der *TDecisionPivot*-Instanz angezeigt werden sollen. Wenn Sie bestimmte Gruppen von Buttons ausschließen, dann müssen diese meist von anderen *TDecisionPivot*-Instanzen angezeigt werden.

- ButtonAutoSize, ButtonHeight, ButtonWidth (Eigenschaften, veröffentlicht)

```
property ButtonAutoSize: Boolean default true;
property ButtonHeight: Integer default 24;
property ButtonWidth: Integer default 64;
```

Hat *ButtonAutoSize* den Wert *true*, dann wird die Größe der Buttons automatisch an die Größe der Komponente angepasst, anderenfalls wird sie mit *ButtonHeight* und *ButtonWidth* eingestellt.

Hat *GroupLayout* den Wert *xtLeftTop*, so wird die Einstellung von *ButtonAutoSize* ignoriert.

- GroupLayout (Eigenschaft, veröffentlicht)

```
property GroupLayout: (xtHorizontal, xtVertical, xtLeftTop)
  default xtHorizontal;
```

Mit *GroupLayout* können Sie festlegen, ob die einzelnen Elemente nebeneinander (*xtHorizontal*), übereinander (*xtVertical*) oder – wie in der folgenden Abbildung – über Eck angeordnet werden.

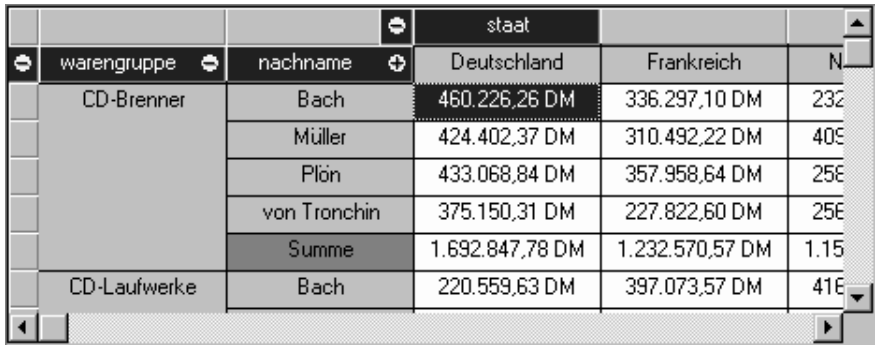
- ButtonSpacing, GroupSpacing (Eigenschaften, veröffentlicht)

```
property ButtonSpacing: Integer default 0;
property GroupSpacing: Integer default 10;
```

Mit *ButtonSpacing* wird der Abstand zwischen den einzelnen Buttons eingestellt, mit *GroupSpacing* der zwischen den drei Gruppen.

9.8.6 TDecisionGrid

Mit der Komponente *TDecisionGrid* werden die Daten in einem Datengitter angezeigt. *TDecisionGrid* ist von *TWinControl* abgeleitet.



		staat		
warengruppe	nachname	Deutschland	Frankreich	N
CD-Brenner	Bach	460.226,26 DM	336.297,10 DM	232
	Müller	424.402,37 DM	310.492,22 DM	409
	Plön	433.068,84 DM	357.958,64 DM	258
	von Tronchin	375.150,31 DM	227.822,60 DM	256
	Summe	1.692.847,78 DM	1.232.570,57 DM	1.15
CD-Laufwerke	Bach	220.559,63 DM	397.073,57 DM	416

- DecisionSource (Eigenschaft, veröffentlicht)

```
property DecisionSource: TDecisionSource;
```

Über die Eigenschaft *DecisionSource* wird die Komponente mit einer *TDecisionSource*-Instanz verbunden, von der sie die Daten bezieht.

- ColCount, RowCount, FixedCols, FixedRows
  (Eigenschaften, öffentlich, nur Lesen)

```
property ColCount: Integer;
property RowCount: Integer;
property FixedCols: Integer;
property FixedRows: Integer;
```

Mit *ColCount* und *RowCount* ermittelt man die Anzahl der Spalten und der Reihen des Gitters. Die Anzahl der Überschriftspalten und -Reihen wird mit *FixedCols* und *FixedRows* ermittelt.

- Cells (Eigenschaft, öffentlich, nur Lesen), MouseCoord (Methode)

```
property Cells[ACol, ARow: Integer]: string;
function MouseCoord(X, Y: Integer): TGridCoord;
```

Mit der Eigenschaft *Cells* können die einzelnen Zelleninhalte ermittelt werden. Die Indizierung dieser Array-Eigenschaft beginnt jedoch nicht in der linken oberen Zelle des Gitters, sondern in der linken oberen Zelle des Datenbereichs, was in der folgenden Beispielprozedur eine Umrechnung erforderlich macht.

Mit *MouseCoord* lassen sich die Reihen- und Spaltenindizes einer bestimmten Bildschirmposition ermitteln. Die folgende Beispielprozedur zeigt den Inhalt der gerade angeklickten Zelle in der Titelzeile des Formulars an.

```
procedure TForm1.DecisionGrid1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  Coord: TGridCoord;
begin
  with DecisionGrid1 do
  begin
    Coord := MouseCoord(x, y);
    Form1.Caption := Cells[Coord.x - FixedCols,
      Coord.y - FixedRows];
  end;
end; {procedure TForm1.DecisionGrid1MouseDown}
```

- Totals (Eigenschaft, öffentlich)

```
property Totals: Boolean;
```

Mittels der Eigenschaft *Totals* wird festgelegt, ob für die einzelnen Spalten und Reihen zusätzlich die Summe der Werte angezeigt wird (*true*) oder nicht (*false*).

Soll die Anzeige von Summen für einzelne Dimensionen unterschiedlich festgelegt werden, so ist die *TDisplayDim*-Eigenschaft *Subtotals* zu verwenden.

- Dimensions (Eigenschaft, veröffentlicht)

```
property Dimensions: TDisplayDims;
```

Mit Hilfe der Eigenschaft *Dimensions* kann man Vorgaben für die Darstellung der einzelnen Dimensionen machen.

Der Typ *TDisplayDims* ist ein Nachfahre von *TCollection*, dessen Eigenschaft *Item* vom Typ *TDisplayDim* ist. *TDisplayDim* implementiert die folgenden Eigenschaften:

- Mit *Alignment* wird festgelegt, ob die Daten und die Beschriftung linksbündig, rechtsbündig oder zentriert ausgegeben werden.
- *Color* legt die Hintergrundfarbe des Dimensionsbeschriftung fest.
- Mit *FieldName* kann festgestellt werden, auf welcher Spalte oder Aggregatfunktion die Dimension beruht. Diese wird auch als Bezeichnung der Dimension verwendet, solange sie nicht mit einem Wert in der Eigenschaft *DisplayName* geändert wird.

- Sollen die Daten nach bestimmten Kriterien formatiert werden, dann kann der Eigenschaft *Format* ein entsprechender Formatierungs-String zugewiesen werden. Dessen Syntax ist an die RTL-Funktionen *FormatFloat* und *FormatDateTime* angelehnt.
- Mit der booleschen Eigenschaft *Subtotals* kann bestimmt werden, ob zu einer Dimension Summen angezeigt werden oder nicht.

Darstellung

- Diverse veröffentlichte Eigenschaften

```
property CaptionColor: TColor default clActiveCaption;
property CaptionFont: TFont;
property DataColor: TColor default clInfoBk;
property DataFont: TFont;
property DataSumColor: TColor default clNone;
property LabelColor: TColor default clBtnFace;
property LabelFont: TFont;
property LabelSumColor: TColor default clInactiveCaption;
```

  Mit diesen Eigenschaften lassen sich die Hintergrundfarbe (*Color*) und die Schrift (*Font*) der einzelnen Zellenarten vorgeben. Die Typen *TColor* und *TFont* sind in Kapitel 4.2.1 beschrieben.

  Im Einzelnen handelt es sich dabei um folgende Zellenarten:

  - Die *Caption*-Zellen zeigen die Dimensionsbezeichnungen an (*Warengruppe, Nachname, Staat*).
  - Die *Data*-Zellen geben die gruppierten Werte an, während die *DataSum*-Felder die Summen der einzelnen Reihen und Spalten anzeigen.
  - Die Beschriftung der Spalten und Reihen (*Bach, Müller ...*) erfolgt durch die *Label*-Zellen, die Beschriftung der Summenspalten und -reihen (*Summe*) durch die *LabelSum*-Zellen.

- Options, GridLineColor, GridLineWidth (Eigenschaft, veröffentlicht)

```
property Options: set of (cgGridLines, cgOutliner, cgPivotable)
  default [cgGridLines, cgOutliner, cgPivotable];
property GridLineColor: TColor default clWindowText;
property GridLineWidth: Integer default 1;
```

  Sollen Linien zwischen den einzelnen Zellen des Gitters angezeigt werden, dann ist die Option *cgGridLines* zu setzen. Die Farbe der Linien kann mit *GridLineColor*, deren Breite mit *GridLineWidth* spezifiziert werden.

Sollen die Symbole + und - angezeigt werden, mit deren Hilfe sich die einzelnen Dimensionen aufschlüsseln oder zusammenfassen lassen, dann muss die Option *cgOutliner* gesetzt werden.

Sollen sich die Dimensionen mit Drag&Drop von der Anzeige als Spalte zur Anzeige als Reihe und umgekehrt verschieben lassen, dann ist die Option *cgPivotable* zu setzen.

- DefaultColWidth, DefaultRowHeight (Eigenschaft, veröffentlicht)

```
property DefaultColWidth: Integer default 100;
property DefaultRowHeight: Integer default 20;
```

Die Ausgangswerte für die Spaltenbreiten und Reihenhöhen werden mit diesen beiden Eigenschaften eingestellt.

- OnDecisionDrawCell (Ereignis)

```
property OnDecisionDrawCell(Sender: TObject; Col, Row: Integer;
  var Value: String; var aFont: TFont; var aColor: TColor;
  AState: TGridDrawState; aDrawState: DecisionDrawState);
```

Vor jedem Zeichnen einer Zelle wird das Ereignis *OnDecisionDrawCell* aufgerufen. Hier besteht dann die Möglichkeit, den anzuzeigenden String, die Schrift oder die Hintergrundfarbe zu ändern. Nähere Informationen zu den Parametern *AState* und *aDrawState* finden Sie in der Online-Hilfe.

Das folgende Beispiel stellt alle Zahlen größer 300000 mit roter Schrift dar:

```
procedure TForm1.DecisionGrid1DecisionDrawCell(Sender: TObject;
   Col, Row: Integer; var Value: String; var aFont: TFont;
  var aColor: TColor; AState: TGridDrawState;
  aDrawState: TDecisionDrawState);
var
  f: double;
  i: integer;
begin
  val(Value, f, i);
  if (f > 300000)
    then aFont.Color := clRed;
end;
```

9.8.7 TDecisionGraph

Die Komponente *TDecisionGraph* dient zur grafischen Darstellung der Daten. *TDecisionGraph* ist von *TChart* abgeleitet, welche in Kapitel 9.2 beschrieben ist.

Beachten Sie dabei, dass immer nur die Reihenkomponenten der gerade aktuellen Dimension zur Verfügung stehen. Für andere Dimensionen können Sie jedoch die *Vorlage* ändern. Diese wird – wie der Name ja bereits andeutet – als Vorlage zur Erzeugung der neuen Reihenkomponente verwendet. Dabei wird für jede Dimension eine eigene Vorlagen-Reihenkomponente erstellt:

- DecisionSource (Eigenschaft, veröffentlicht)

  ```
  property DecisionSource: TDecisionSource;
  ```

  Über die Eigenschaft *DecisionSource* wird die Komponente mit einer *TDecisionSource*-Instanz verbunden, von der sie die Daten bezieht.

9.8.8 Datenanalyse im Thread

So, wie wir das Beispiel in Kapitel 3.12.1 programmiert haben, wird das Öffnen der Abfrage und das Füllen des Datenwürfels beim Programmstart durchgeführt. Dies würde bedeuten, dass zwischen dem Doppelklick auf das Programm-Icon und der Anzeige des Formulars der Anwender erst einmal warten muss und sich in der Zwischenzeit überlegen kann, ob nicht vielleicht das Programm abgestürzt ist (21 Sekunden auf einem 100 MHz Pentium bei ausreichend Speicher – okay, 100 Mhz ist nicht mehr Stand der Dinge, aber viele Anwender arbeiten noch auf älteren Maschinen).

Man könnten nun die Datenmenge beim Programmstart geschlossen lassen und dann beispielsweise beim Aufruf des entsprechenden Menüpunktes öffnen. Dann wartet der Anwender zwar immer noch, aber nur, wenn er wirklich Datenanalyse betreiben will. Will er andere Programmfunktionen nützen, dann bleibt ihm die Warterei erspart.

Nun sollte ein gutes Programm den Anwender möglichst überhaupt nicht warten lassen. Dies könnte man beispielsweise dadurch erreichen, dass man zunächst das Programm startet und dann die Abfrage für die Datenanalyse im Hintergrund, also in einem zweiten Thread startet. Der Anwender kann dann zwar die erste halbe Minute keine Datenanalyse durchführen, diese Zeit aber anderweitig (mehr oder weniger) sinnvoll nutzen.

Die Besonderheiten der Thread-Programmierung werden in Kapitel 5.1 ausführlich beschrieben, das hier Erforderliche deshalb nur im »Telegrammstil«:

Das nun erstellte Beispielprogramm umfasst zwei Formulare: Auf *Form1* wird in einem *DBGrid* eine Abfrage über die Tabelle *f_verkauf* angezeigt (als JOIN mit den dazugehörenden Klartexttabellen). Während der Anwender hier die Zahlen betrachten kann, wird in einem zweiten Thread die Abfrage für die Datenanalyse in *Form2* geöffnet. Der Aufbau von *Form2* gleicht unserem Beispielprogramm von vorhin, auf die wenigen Unterschiede kommen wir gleich.

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Button1.Enabled := false;
  StatusBar1.SimpleText
    := 'Abfrage wird durchgeführt. Bitte warten';
  Application.ProcessMessages;
  Query1.Active := true;
  StatusBar1.SimpleText
    := 'Datenanalyse wird vorbereitet. Bitte warten';
  Application.ProcessMessages;
  with TAbfrage.Create(false)
    doFreeOnTerminate := true;
end;
```

Damit die Anwendung so schnell wie möglich startet, wird auch die Abfrage für *Form1* erst dann geöffnet, wenn das Formular erstellt ist. Damit der Anwender darüber informiert ist, was nun gerade passiert, wird ein entsprechender Hinweis in der Statuszeile ausgegeben. Nun wird *Query1* geöffnet. Da die Abfrage die Datensätze nach Datum absteigend sortiert, werden gleich die aktuellsten Datensätze angezeigt – diese dürften den Anwender auch am meisten interessieren. Anschließend wird das Thread-Objekt erzeugt.

Die Typendeklaration des Thread-Objektes muss vom Programmierer selbst eingefügt werden. Hier im Beispiel reicht schon ein Minimal-Objekt:

```
TAbfrage = class(TThread)
protected
  procedure Execute; override;
end;
```

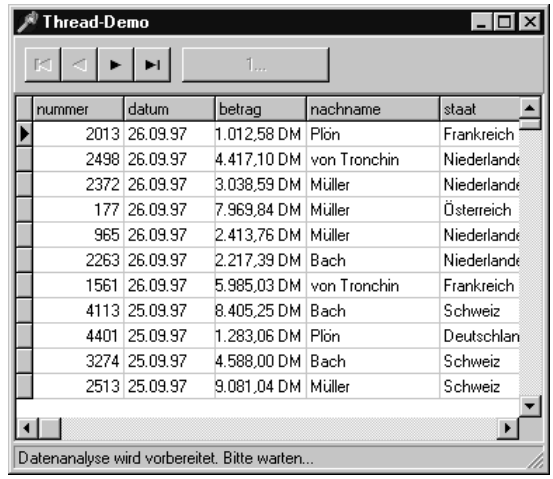
In die Methode *Execute* werden die Befehle geschrieben, die der Thread ausführen soll. Beachten Sie bitte, dass diese Prozedur in *TThread* abstrakt definiert ist und deshalb zwingend überschrieben werden muss.

```
procedure TAbfrage.Execute;
begin
  Form1.Button1.Caption := '1...';
  Form2.DecisionQuery1.Active := true;
  Form1.Button1.Caption := '2...';
  Form2.DecisionCube1.DataSet := Form2.DecisionQuery1;
  Form1.Button1.Caption := 'Datenanalyse';
  Form1.Button1.Enabled := true;
  StatusBar1.SimpleText := '';
  MessageBeep(MB_OK);
end;
```

Wenn die Vorbereitung für die Datenanalyse im Hintergrund ablaufen soll, dann kann die Fortschrittsanzeige von *TDecisionCube* nicht verwendet werden – die Eigenschaft *ShowProgressDialog* ist somit auf *false* zu setzen. Wir wollen uns jedoch einen bescheidenen Ersatz dafür basteln. Zunächst wird der Button, mit dem später *Form2* aufgerufen wird, mit *1* beschriftet, anschließend wird die Abfrage für die Datenanalyse geöffnet.

Im nächsten Schritt wird die Beschriftung auf *2* geändert, anschließend wird der Datenwürfel gefüllt, indem ihm die Datenmenge zugewiesen wird – auch dies darf dann zur Entwurfszeit nicht geschehen. Jetzt wird der Button mit *Datenanalyse* beschriftet und seine Betätigung ermöglicht. Um den Anwender darüber zu informieren, dass er nun mit der Datenanalyse beginnen kann, wird ein *MessageBeep* ausgegeben.

Bevor die Anwendung gestartet werden kann, muss für den Thread noch eine zweite Session eingefügt werden. Die erste Session generiert Delphi automatisch – sobald eine zweite benötigt wird, muss explizit eine *TSession*-Komponente eingefügt werden. Deren Eigenschaft *SessionName* wird ein beliebiger Name zugewiesen (beispielsweise *Abfrage*), anschließend wird die Eigenschaft *Active* auf *true* gesetzt. Nun muss nur noch die Eigenschaft *SessionName* von *DecisionQuery1* entsprechend gesetzt werden.



Wie eingangs erwähnt, dauerte bei mir das Starten der Datenanalyse ohne Thread 21 Sekunden. Mit Thread liegt die Zeit zwischen 24 und 27 Sekunden, je nachdem, ob man währenddessen in der Abfrage von *Form1* navigiert oder nicht. Wie Sie sehen, beschleunigt der Thread den Vorgang nicht (wie sollte er auch), aber er nutzt die Zeit aus, in der das System ohnehin unbeschäftigt ist.

10 Die 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 prinzipiell folgende Möglichkeiten:

- Daten lassen sich mittels *TPrinter* und dessen *Canvas*-Methoden ausdrucken. Dies ist zwar alles andere als komfortabel, bietet aber die beste Kontrolle über das Endergebnis.
- Einfache Reports kann man mit den *QuickReport*-Komponenten erstellen, die wir in diesem Kapitel besprechen wollen. Diese Komponenten haben zwei Vorteile: Sie sind schnell erstellt und sie fügen sich nahtlos in Delphi ein; es besteht somit nicht die Notwendigkeit, einen externen Reportgenerator zu installieren und anzusteuern. Für anspruchsvolle Reports sind die QuickReport-Komponenten jedoch nicht die »erste Wahl«.
- Externe Reportgeneratoren wie beipielsweise *CrystalReport* stellen sehr viele Features bereit, die jedoch die Bedienung nicht immer vereinfachen. Externe Reportgeneratoren bedeuten zudem immer zusätzlichen Installationsaufwand und fügen sich nicht nahtlos in die Anwendung ein.
- Eigenständige Textverarbeitungssoftware wie beispielsweise *Microsoft Word* kann von außen gesteuert werden. Auf diese Weise lassen sich auch Reports aufbauen, die dann den Vorteil haben, dass sie der Anwender gleich mit der gewohnten Software weiterverarbeiten kann.

Die Odyssee der QuickReport-Komponenten

Die *QuickReport*-Komponenten haben unter Delphi-Programmierern nicht den besten Ruf, weil Sie am Anfang ihrer Entstehungsgeschichte durch zahlreiche Macken und insbesondere durch Inkompatibilitäten zwischen den Versionen ganze Heerscharen von Entwicklern zur Verzweiflung brachten.

QuickReport wird von der Firma QuSoft entwickelt und lag in der Version 1 der Delphi-Version 2 bei. (Unter Delphi 1 arbeitete man noch mit *ReportSmith*, dieser externe Reportgenerator gehört nun schon seit vielen Delphi-Generationen der Vergangenheit an.) Mit QuickReport 1 konnte man recht schnell einfache Reports erstellen, außerdem musste kein externer Report-Generator mehr installiert werden, und so stürzten sich viele Delphi-Entwickler auf die neuen Komponenten. Diese hatten zwar so manche Macke, aber die meisten Klippen konnte man irgendwie umschiffen.

Nun haben schnell zusammengestrickte Geschichten meist den Nachteil, dass die ganzen Grundlagen nicht stimmen. In so einer Situation ist es besser, komplett von vorne zu beginnen, als mit einem vermurksten Code weiterzuarbeiten. QuickReport 2, das in Delphi 3 beilag, war das Ergebnis eines solchen Neuanfangs. Allerdings waren nun alle bisherigen Reports nicht mehr zu gebrauchen.

Was einem als *teilweise Konvertierung* angeboten wurde, hat in der Regel zur Folge, dass das Ergebnis unter keiner der beiden Delphi-Versionen mehr gelaufen ist. Eine weitere Folge dieser Neuprogrammierung war, dass zwar an einigen Stellen altbekannte Fehler behoben – genauer gesagt, nicht wiederholt – wurden, dafür waren jede Menge neue Probleme vorhanden. Und die Online-Hilfe habe ich in meinem damaligen Datenbankbuch als *geradezu grausam* bezeichnet.

Die Hoffnung, dass unter Delphi 4 (QuickReport 3) alles besser wird, hatte sich leider nicht bewahrheitet, ganz im Gegenteil: Auf der Entwicklerkonferenz 1998 hatte Imprise (wie Borland sich damals genannt hatte) empfohlen, sich für Delphi 4 QuickReport 2 zu besorgen (wegen des geänderten Package-Formats haben sich die Dateien von Delphi 3 nicht weiterverwenden lassen).

Über die Homepage (*www.qusoft.no*) von Qusoft war dann eine aktualisierte Version von QuickReport 3 erhältlich – welche dann jedoch nicht mehr mit den *TTeeChart*-Komponenten kompatibel war. Erst ab UpdatePack 2 von Delphi 4 funktionierte die Sache dann vernünftig.

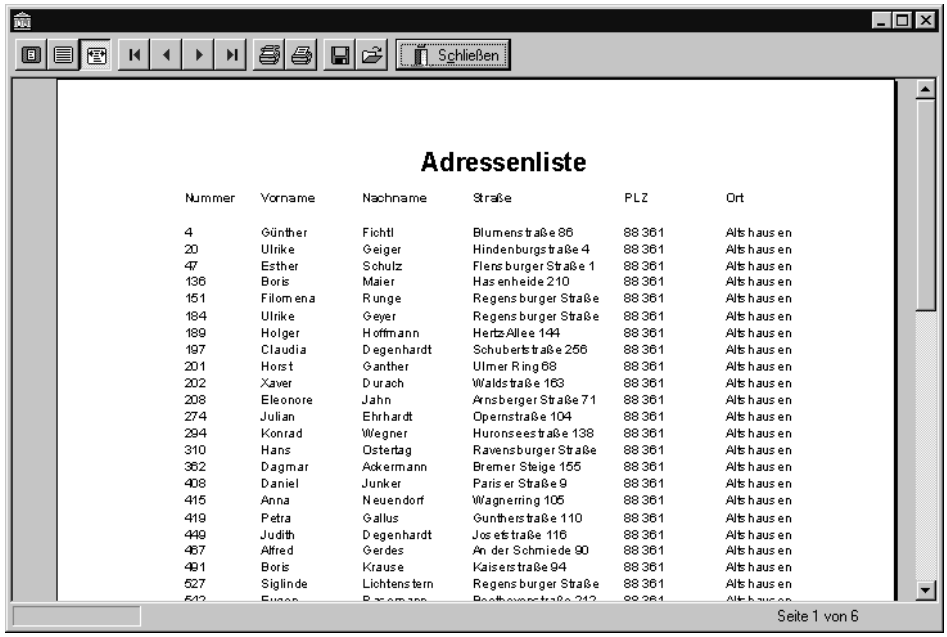
Nun, da jeder Kredit verspielt war, begann die Sache dann vernünftig zu funktionieren. Nach der Devise *Never touch a running system* wurden die QuickReport-Komponenten für Delphi 5 (QuickReport 3.05) nur noch ganz unwesentlich überarbeitet, so dass keine größeren Macken zu beobachten sind. Und auch in Delphi 6 (QuickReport 3.09) hat sich erfreulich wenig getan.

Das hat nicht nur den Vorteil (manche mögen es eine Selbstverständlichkeit nennen), dass sich die bislang erstellten Reports weiterverwenden lassen, sondern auch die Erfahrungen, die man mit der vorhergehenden Version gesammelt hat, werden nicht einfach wertlos. Vorteile, die man erst dann richtig zu schätzen weiß, wenn man es ein paar Mal anders erlebt hat.

Zudem sollte man nicht vergessen, dass auch andere Reportgeneratoren ihre Macken haben. Es gibt so manche Entwicklungsabteilung, die nach diversen anderen Versuchen inzwischen bei QuickReport gelandet ist. Und gerade die Möglichkeit, die Reports einfach an die ohnehin vorhandenen Datenmengenkomponenten anzuhängen, weiß man dann erst recht zu schätzen, wenn man mal eine kompliziertere Abfrage auf einen externen Reportgenerator übertragen musste.

10.1 Eine Adressenliste

Zunächst wollen wir als Einstieg eine einfache Adressenliste ausdrucken:



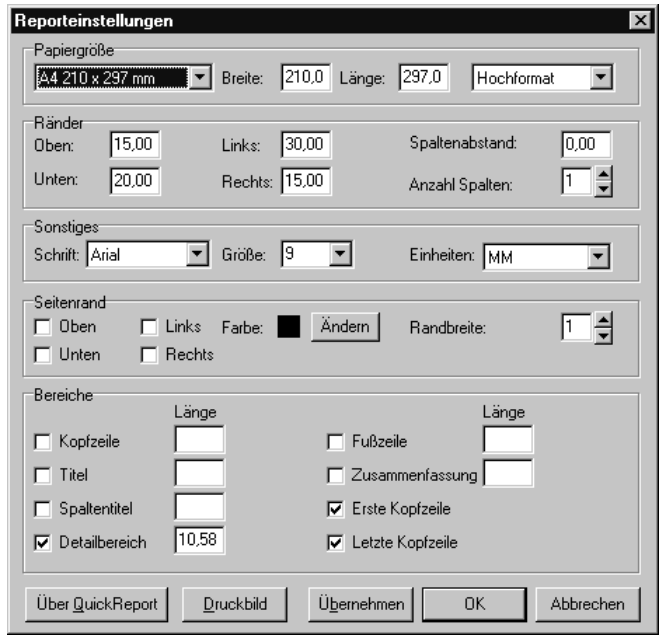
Bei QuickReport wird für jeden einzelnen Report dem Projekt ein neues Formular hinzugefügt, dem eine *TQuickRep*-Instanz hinzugefügt wird. Diese Formulare werden nie mit *Show* geöffnet, stattdessen werden die Methoden *Preview* oder *Print* dieser *TQuickRep*-Komponente aufgerufen.

Zum Starten der einzelnen Reports fügen wir deshalb *Form1* einige Buttons hinzu, mit denen die Vorschau der Reports gestartet wird.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.QuickRep1.Preview;
end;
```

Form2 wird nun die Komponente *QuickRep1* hinzugefügt. Es ist dabei unerheblich, wie diese Komponente im Formular ausgerichtet ist.

Mit einem Doppelklick auf eine *TQuickRep*-Komponente wird ein Dialog geöffnet, der einige Seiteneinstellungen wie Papiergröße und Seitenränder gestattet:



Hier sollen zunächst einmal die Ränder vergrößert werden – die Voreinstellung von jeweils 1cm ist nicht besonders praxisnah, insbesondere dann nicht, wenn die Listen abgeheftet werden sollen.

Als Nächstes wird eine *TQuery*-Instanz hinzugefügt, die aus der Datenbank *Buch* die Tabelle *testadr* bezieht. Damit das Erstellen des Reports nicht zum Geduldspiel wird – 10000 Datensätze machen sich auch bei QuickReport bemerkbar – wird der Datenbestand gefiltert.

```
SELECT * FROM adressen
  WHERE ort = "Altshausen"
```

Die Eigenschaft *DataSet* von *QuickRep1* wird auf *Query1* gesetzt, eine Datenquellenkomponente ist hier nicht erforderlich. Nun werden aus der Mengeneigenschaft *Bands* von *QuickRep1* die Eigenschaften *HasTitle*, *HasColumnHeader* und *HasDetail* auf *true* gesetzt. Es werden dann automatisch drei Bänder eingefügt.

In das Titelband soll lediglich der Reporttitel eingefügt werden. Dazu wird in dieses Band eine *TQRLabel*-Instanz eingefügt, deren Eigenschaften *Caption* und *Font* entsprechend gesetzt werden.

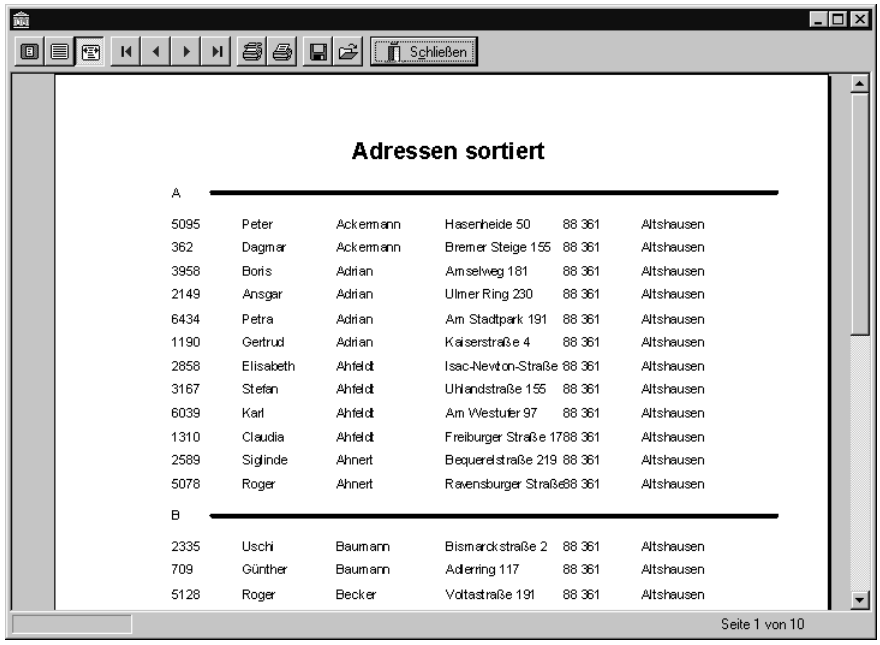
Auf den Detailbändern werden die Datensätze ausgedruckt. Diese Detailbänder lassen sich in mehrere Spalten teilen, was beispielweise zum Ausdruck von Etiketten erforderlich ist – wir kommen später dazu. Das Column-Header-Band wird oberhalb jeder Spalte gedruckt und dient uns hier dazu, die einzelnen Tabellen-Spalten (nicht verwechseln mit Detailband-Spalten) zu beschriften. Auch dafür wird wieder die Komponente *TQRLabel* verwendet.

Für jeden Datensatz der Datenmenge *QuickRep1.DataSet* wird ein Detailband ausgedruckt. Auf dieses Detailband werden nun entsprechend viele *TQRDBText*-Komponenten gesetzt. Mit den Eigenschaften *DataSet* und *DataField* werden ihnen die Datenmenge und die Felder zugewiesen, aus denen sie ihre Daten beziehen.

Damit ist der erste Report fertig. Zu den Vorzügen von QuickReport gehört es, dass die Anwendung nicht gestartet werden muss –, der Report lässt sich auch ansehen, indem man aus dem Kontextmenü von *QuickRep1* den Menüpunkt DRUCKBILD wählt.

10.2 Gruppieren der Adressen

Die Übersichtlichkeit von Listen kann oft durch die Gruppierung der Datensätze gesteigert werden. Wir wollen hier nun bei der Adressenliste die Datensätze nach dem Anfangsbuchstaben des Nachnamens gruppieren:



Zunächst wird eine TQRGroup-Komponente in das Formular eingefügt. Deren Eigenschaft *Master* wird auf *QuickRep1* gestellt, somit werden die Detail- und nicht die (hier nicht vorhandenen) Sub-Detail-Datensätze gruppiert.

Mit Hilfe der Eigenschaft *Expression* wird festgelegt, wann ein Gruppenband eingefügt wird:

Für diese Eigenschaft gibt es einen Eigenschaftseditor, der allerdings ein wenig ungewohnt zu bedienen ist. Halten Sie sich streng an die folgende Reihenfolge:

- Zunächst wird in der GroupBox *Funktion* die Funktion COPY gewählt und mit HINZUFÜGEN eingefügt.
- In der GroupBox *Datenfeld* wird das Feld *Nachname* gewählt, anschließend betätigen Sie HINZUFÜGEN und OK.
- Nun geben Sie in der GroupBox *Konstanten* den Wert 1 ein und wählen die Option *Numerisch*. Anschließend betätigen Sie HINZUFÜGEN und OK. Wiederholen Sie die Schritte dieses Punktes.
- Als Ausdruck erhalten Sie somit COPY(Query1.Nachname, 1, 1), damit wird der erste Buchstabe des Nachnamens extrahiert. Sobald ein neuer Anfangsbuchstaben erscheint, wird somit ein neues Band eingefügt. Mit OK schließen Sie den Dialog.

Nun muss die Abfrage noch nach dem Nachnamen sortiert werden.

```
SELECT * FROM testadr
  WHERE plz LIKE "88%"
  ORDER BY nachname
```

Es ist nicht nötig, weitere Elemente auf dem *TQRGroup*-Band zu platzieren – die Datensätze können auch schlicht durch einen größeren Abstand voneinander getrennt werden. Wir wollen jedoch auf diesem Band die Initiale und eine horizontale Linie darstellen.

Für die Initiale wird eine *TQRExpr*-Instanz eingefügt. Wie Sie hier die Eigenschaft *Expression* setzen, wurde eben schon beschrieben. Des Weiteren darf die Schrift ein wenig größer und fett formatiert werden.

Für die Linie wird eine *TQRShape*-Komponente eingefügt, deren Eigenschaft *Shape* auf *qrsHorLine* gesetzt wird. Damit die Linie etwas kräftiger wird, wird die Eigenschaft *Pen.Width* auf *fünf* gesetzt.

10.3 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 die gleichen wie beim ersten Report, jedoch soll hier folgende SQL-Anweisung verwendet werden:

```
SELECT
     Vorname || " " || Nachname AS Name,
     Straße,
     PLZ || " " || Ort AS Wohnort
  FROM testadr
  WHERE PLZ LIKE "8%"
```

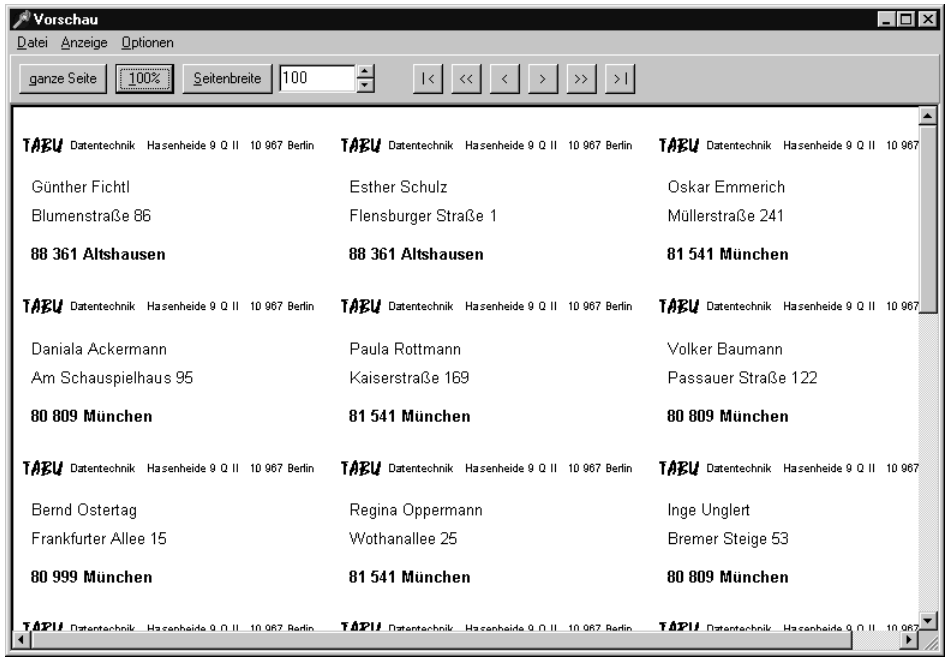
Neu ist hier die Zusammenfassung von Vor- und Nachnamen sowie von Postleitzahl und Ort. Dadurch wird jeweils nur ein *QRDBText* erforderlich und die beiden Wörter schließen direkt aneinander an (bei zwei Feldern müssten Sie zwei *QRDBText*-Komponenten verwenden; da die Vornamen unterschiedlich lang sind, würde zwischen Vor- und Nachname eine unterschiedlich lange Lücke entstehen).

Die Eigenschaft *Page.Columns* von *QuickRep1* ist auf *3* zu setzen; die Ausgabe wird dann in drei Spalten formatiert (so einfach geht das mit den Etiketten). Das *DetailBand1* muss nun in der Höhe den Etiketten angepasst werden. Dazu dient die Eigenschaft *Size.Height*, mit welcher die Höhe in Millimeter angegeben werden kann.

Auf jedem 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.

10.3.1 Eigenes Vorschauformular

Wem das QuickReport-Vorschauformular nicht gefällt, der kann relativ problemlos ein eigenes entwickeln. Dazu wird ein neues Formular erstellt, welches zunächst mit einem Menü, einem Panel und der Komponente *TQRPreview* versehen wird.



Damit das eigene Vorschauformular verwendet werden kann, muss der Report der *QRPrinter*-Eigenschaft der *TQRPreview*-Komponente zugewiesen werden. Prinzipiell wäre es möglich, für jeden Report ein eigenes Vorschauformular zu entwicklen.

```
procedure TForm3.QuickRep1Preview(Sender: TObject);
begin
  Form4.QRPreview1.QRPrinter := (Sender as TQRPrinter);
  Form4.Show;
end;
```

Zoom-Faktor

Wenn das Formular gestartet wird, dann wird zunächst der Zoom-Faktor auf 100 % gestellt.

```
procedure TForm4.FormShow(Sender: TObject);
begin
  QRPreview1.Zoom := 100;
  Edit1.Text := '100';
end;
```

Der Zoom-Faktor kann dadurch geändert werden, dass in *Edit1* ein anderer Wert eingegeben wird, was auch mit Hilfe von *UpDown1* geschehen kann. Der Text von *Edit1* wird in eine Integer-Zahl gewandelt und der Eigenschaft *Zoom* von *QRPreview1* zugewiesen.

```
procedure TForm4.Edit1Change(Sender: TObject);
begin
  QRPreview1.Zoom := StrToInt(Edit1.Text);
end;
```

Mit Hilfe von *Button1* bis *Button3* kann der Report auf Originalgröße, Seitenhöhe oder Seitenbreite gezoomt werden. Die Anzeige in *Edit1* wird dementsprechend aktualisiert.

```
procedure TForm4.Button1Click(Sender: TObject);
begin
  {Zoom 100 %}
  QRPreview1.Zoom := 100;
  Edit1.Text := IntToStr(QRPreview1.Zoom);
end;

procedure TForm4.Button2Click(Sender: TObject);
begin
  {Zoom Seitenhöhe}
  QRPreview1.ZoomToFit;
  Edit1.Text := IntToStr(QRPreview1.Zoom);
end;

procedure TForm4.Button3Click(Sender: TObject);
begin
  {Zoom Seitenbreite}
  QRPreview1.ZoomToWidth;
  Edit1.Text := IntToStr(QRPreview1.Zoom);
end;
```

Wechseln der Seite

Das Zusammenstellen der Seiten geschieht im Hintergrund. Während die erste Seite schon betrachtet werden kann, werden weitere Seiten zusammengestellt. Somit kann am Anfang noch nicht festgestellt werden, wie viele Seiten der Report umfasst. Ist eine Seite zusammengestellt, dann wird das Ereignis *OnPageAvailable* ausgelöst. Mit Hilfe des Parameters *PageNum* wird dann das Feld *FPageCount* aktualisiert, außerdem wird die Gesamtzahl der Seiten mit *Label2* angezeigt.

```
procedure TForm4.QRPreview1PageAvailable(Sender: TObject;
  PageNum: Integer);
begin
  Label2.Caption := IntToStr(PageNum);
  FPageCount := PageNum;
end;
```

Um die Seite zu wechseln, lösen *Button4* bis *Button9* folgende Anweisungen aus:

```
procedure TForm4.Button4Click(Sender: TObject);
begin
  {Erste Seite}
  QRPreview1.PageNumber := 1;
  Label3.Caption := 'Seite ' + IntToStr(QRPreview1.PageNumber);
end;

procedure TForm4.Button6Click(Sender: TObject);
begin
  {vorangehende Seite}
  QRPreview1.PageNumber := QRPreview1.PageNumber - 1;
  Label3.Caption := 'Seite ' + IntToStr(QRPreview1.PageNumber);
end;

procedure TForm4.Button7Click(Sender: TObject);
begin
  {nächste Seite}
  QRPreview1.PageNumber := QRPreview1.PageNumber + 1;
  Label3.Caption := 'Seite ' + IntToStr(QRPreview1.PageNumber);
end;
```

Bei umfangreicheren Reports ist es sehr hilfreich, die Seite auch in Zehnerschritten wechseln zu können. Mit *Label3* wird die aktuelle Seitenzahl angezeigt.

```
procedure TForm4.Button5Click(Sender: TObject);
begin
  {10 Seiten nach vorne}
  QRPreview1.PageNumber := QRPreview1.PageNumber - 10;
  Label3.Caption := 'Seite ' + IntToStr(QRPreview1.PageNumber);
end;

procedure TForm4.Button8Click(Sender: TObject);
begin
  {10 Seiten nach hinten}
  QRPreview1.PageNumber := QRPreview1.PageNumber + 10;
  Label3.Caption := 'Seite ' + IntToStr(QRPreview1.PageNumber);
end;

procedure TForm4.Button9Click(Sender: TObject);
begin
  {letzte Seite}
  QRPreview1.PageNumber := FPageCount;
  Label3.Caption := 'Seite ' + IntToStr(QRPreview1.PageNumber);
end;
```

Speichern und Drucken

Zum Speichern des Reports muss lediglich die Methode *Save* aufgerufen werden.

```
procedure TForm4.Speichern1Click(Sender: TObject);
begin
  if SaveDialog1.Execute
    then QRPreview1.QRPrinter.Save(SaveDialog1.FileName);
end;
```

Zum Öffnen eines Reports wird dementsprechend die Methode *Load* verwendet. Anschließend muss ein Seitenwechsel durchgeführt werden, damit der neue Report angezeigt wird, außerdem müssen die Anzeigen der Gesamtseitenzahl und der aktuellen Seitenzahl modifiziert werden.

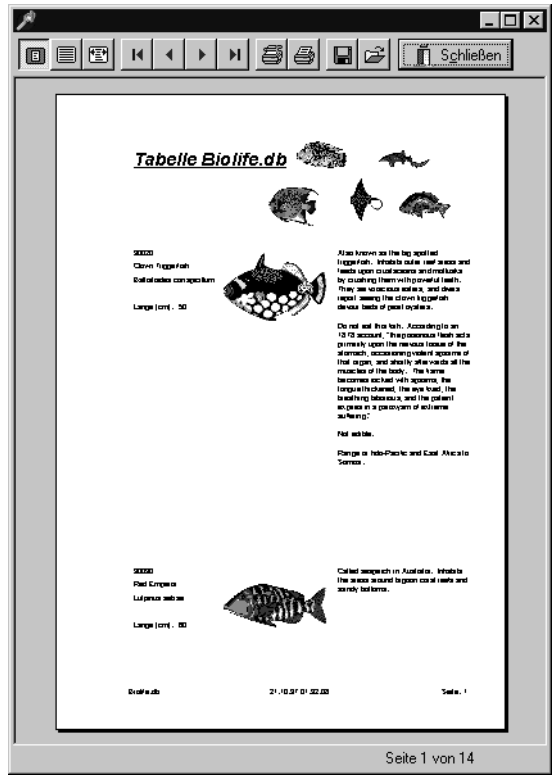
```
procedure TForm4.ffnen1Click(Sender: TObject);
begin
  if OpenDialog1.Execute
    then QRPreview1.QRPrinter.Load(OpenDialog1.FileName);
  QRPreview1.PageNumber := 2;
  QRPreview1.PageNumber := 1;
```

```
  Label3.Caption := 'Seite ' + IntToStr(QRPreview1.PageNumber);
  Label2.Caption := IntToStr(QRPreview1.QRPrinter.PageCount);
end;
```

Zum Drucken des Reports wird die Methode *Print* aufgerufen, zum Einstellen der auszudruckenden Seiten, der Anzahl der Kopien und des Druckers wird die Methode *PrintSetup* verwendet. Das Exportieren – beispielsweise als ASCII-Text – hat bei der mir vorliegenden Version (noch) nicht funktioniert.

10.4 Reports mit Bildern und Memos

Mit den *QuickReport*-Komponenten können auch Reports erstellt werden, die Bilder und Memos anzeigen. Für die Anzeige von Bildern werden lediglich entsprechende *TQRImage*- und *TQRDBImage*-Komponenten auf die Bänder gesetzt. Die Fähigkeit zur Anzeige von Memos ist bereits in den *TQRDBText*-Komponenten enthalten.



Hier sind nun drei verschiedene Bänder vorgesehen: Beim obersten Band ist die Eigenschaft *BandType* gleich *rbTitle*, dieses Band soll zu Beginn des Reports einmalig auf die erste Seite gedruckt werden. In dieses Band ist eine *TQRImage*-Komponente eingefügt. (Die Grafik mit den verschiedenen Fischen wurde per Bildbearbeitungsprogramm erstellt.)

Das nächste Band ist das wohl bekannte *rbDetail*-Band, welches einmal pro (Haupt-) Datensatz verwendet wird. Hier wird eine *TQRDBImage*-Komponente eingefügt.

Für das Memo wird die Komponente *TQRDBText* verwendet. *QuickReport* fügt automatisch weitere Zeilen an, wenn sich mehr Text im Memo-Feld befindet. Die Höhe der Komponente wird dem zur Verfügung stehenden Platz angepasst. Die Eigenschaft *AutoStretch* muss auf *true*, die Eigenschaft *AutoSize* auf *false* gesetzt werden.

Als Nächstes wird eine Fußzeile für alle Seiten erstellt: Hier sollen nun der Reporttitel, Datum und Zeitpunkt der Reporterstellung sowie die Seitenzahl erscheinen. Für die Darstellung solcher Informationen gibt es 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 *TQuickRep* zugewiesen.

Sie werden sich vielleicht wundern, dass bislang überhaupt kein Quelltext geschrieben wurde; dies ist für die Definition eines Reports meist überhaupt nicht erforderlich.

10.5 Master-Detail-Reports

Mit den *QuickReport*-Komponenten ist es auch möglich, Master-Detail-Reports zu erstellen. 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 aufwändiger und fehleranfälliger als einfache Reports; halten Sie sich deshalb genau an die vorgegebenen Anweisungen und deren Reihenfolge.

Sinnvollerweise beginnt man hier, für den Master einen ganz gewöhnlichen Report zu erstellen. Die Master-Tabelle ist hier die Adressenliste, es wäre jedoch unsinnig, hier alle Datensätze aufzunehmen. Mit Hilfe eines OUTER JOINs werden nur diejenigen Datensätze aufgenommen, die auch in der Tabelle offen_2 vorkommen.

```
SELECT DISTINCT
     t.nummer,
     t.nachname,
     SUM(o.betrag)
  FROM offen_2 o LEFT OUTER JOIN testadr t
     ON(o.kundennummer = t.nummer)
  GROUP BY t.nummer, t.nachname
```

Wie Sie die Bänder einrichten und bestücken, wissen Sie inzwischen selbst. Der Report kann nun probeweise gestartet werden.

Die Details

Der Report soll nun um das Detailband erweitert werden. Zunächst wird eine weitere *TQuery*-Instanz eingefügt. Damit diese mit *Query1* in Form einer Master-Detail-Beziehung verknüpft werden kann, muss für *Query1* die Komponente *DataSource1* hinzugefügt werden. Die Eigenschaft *DataSource* von *Query2* wird dann auf *DataSource1* gesetzt. Für *Query2* wird folgende SQL-Anweisung verwendet:

```
SELECT * FROM offen_2
  WHERE kundennummer = :nummer
```

Detaildatensätze werden auf *TQRSubDetail*-Komponenten angezeigt. Fügen Sie eine solche Komponente in den Report ein und setzen Sie deren Eigenschaft auf *DataSet2*. Die Eigenschaft *Master* wird auf *QuickRep1* gesetzt. Es wäre durchaus möglich, Detaildatensätze zu Detaildatensätzen anzuzeigen – die Eigenschaft *Master* der Detail-Detail-Datensätze würde man dann auf *QRSubDetail1* setzen.

Des Weiteren sollen in die Detaildatensätze eine Kopf- und eine Fußzeile eingefügt werden. Setzen Sie also die Eigenschaften *Bands.HasFooter* und *Bands.HasHeader* auf *true*. Datensatzwerte und Beschriftungen werden auch hier mit Hilfe von *TQRLabel*- und *TQRDBText*-Komponenten angezeigt.

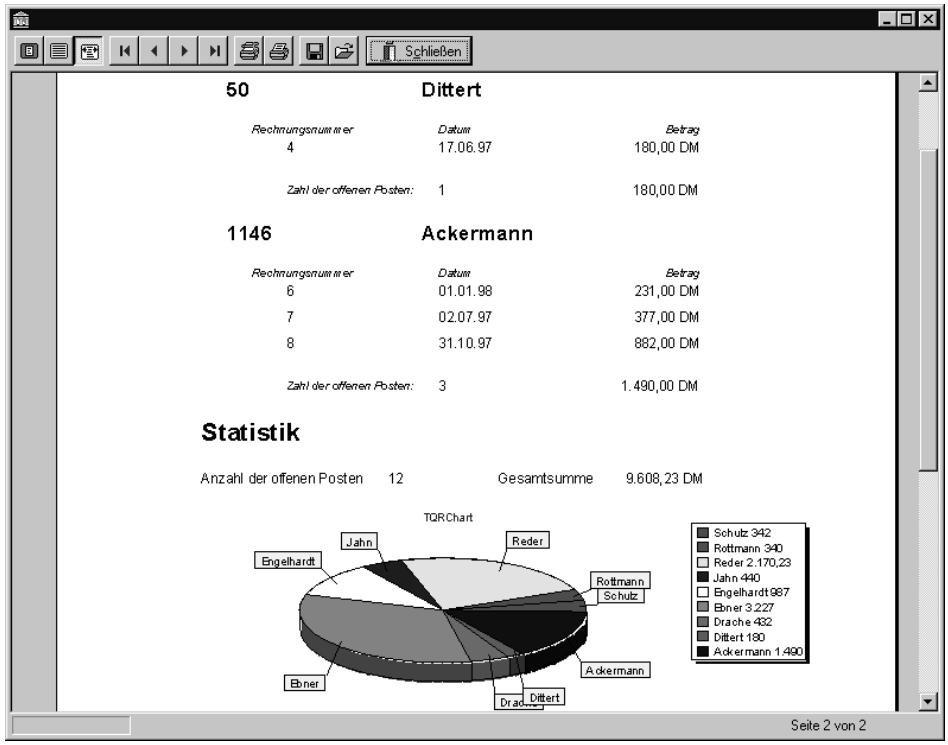
Auf den Detailfußzeilen sollen die Zahl und die Gesamtsumme der Außenstände angezeigt werden. Zu diesem Zweck werden zwei *TQRExpression*-Komponenten eingefügt.

Zur Anzeige der Gesamtzahl der Außenstände des jeweiligen Kunden wird die Funktion COUNT verwendet, welche bei jedem Detailband – also bei jedem Detaildatensatz – um eins erhöht wird. Damit bei jedem neuen Kunden neu gezählt wird, muss die Eigenschaft *ResetAfterPrint* auf *true* gesetzt werden.

Zur Anzeige der Gesamtsumme wird der Eigenschaft *Expression* die Funktion *SUM(Query2.Betrag)* zugewiesen.

Die Statistik

Zum Schluss wollen wir noch eine Statistik der offenen Posten erstellen. Die Verwendung von *TQRChart* soll dabei jedoch nicht besprochen werden, hier möchte ich Sie auf das Kapitel 9 verweisen. Zur Anzeige der Gesamtzahl und der Gesamtsumme der Außenstände werden hier wieder zwei TQRExpression-Instanzen verwendet. Da diese nur einmal am Schluss der Reports verwendet werden, ist hier der Wert der Eigenschaft *ResetAfterPrint* unerheblich.



10.6 Referenz der QuickReport-Komponenten

In diesem Unterkapitel soll eine kurze Referenz einiger *QuickReport*-Komponenten gegeben werden.

10.6.1 Die Komponente TQuickRep

Die Komponente *TQuickRep* macht aus einem Delphi-Formular ein Reportformular. Sie ist für jedes Reportformular zwingend erforderlich, jedoch nicht für eigene Vorschauformulare.

- DataSet (Eigenschaft, veröffentlicht)

  ```
  property DataSet: TDataSet;
  ```

  Mit der Eigenschaft *DataSet* wird festgelegt, welche Datenmenge den Report steuert. Für jeden Datensatz dieser Datenmenge wird ein Detailband in den Report eingefügt. Bei Master-Detail-Reports muss *DataSet* auf die Master-Datenmenge gesetzt werden.

  Sie können Datenmengenkomponenten ausschließlich für die Benutzung durch QuickReport einfügen. Sinnvoller ist es jedoch meist, QuickReport an vorhandene Datenmengenkomponenten anzuhängen, so dass der Anwender das ausdrucken kann, was er ohnehin im DBGrid angezeigt bekommt.

  Bei QuickReport 3.05 ist es unerheblich, ob sich die Datenmengenkomponente auf dem gleichen Formular wie der Report, auf einem anderen Formular oder in einem Datenmodul befindet. Bei älteren Versionen von QuickReport treten manchmal Probleme auf, wenn sich die Datenmengenkomponente nicht auf dem Reportformular befindet.

- Page (Eigenschaft, veröffentlicht)

  ```
  property Page: TQRPage
  ```

  Die Seiteneinstellungen, beispielsweise die Einstellung der Blattränder, werden mit der Objekt-Eigenschaft *Page* vorgenommen. *TQRPage* wiederum verfügt über die folgenden Eigenschaften:

  - Die Blattgröße wird mit *PaperSize* oder mit *Length* und *Width* eingestellt.
  - Die Blattränder werden mit den Eigenschaften *LeftMargin*, *RightMargin*, *TopMargin* und *BottomMargin* eingestellt.
  - Die Zahl der Spalten wird mit Hilfe der Eigenschaft *Columns* angegeben. Der Steg zwischen den Spalten wird mit *ColumnSpace* spezifiziert.
  - Über die Ausrichtung des Blattes entscheidet die Eigenschaft *Orientation*.

- Sollen während des Entwurfs Lineale und ein Gitter zum Ausrichten angezeigt werden, dann ist *Ruler* auf *true* zu setzen.

- Units (Eigenschaft, veröffentlicht)

```
property Units: (MM, Inches, Pixels, Characters, Native);
```

Alle Maße werden in der Einheit angegeben, die in *Units* vorgegeben wird. Dabei sind folgende Werte möglich: *Inches*, *MM* (Millimeter), *Pixels*, *Native* (1/10 mm) und *Characters* (arbeitet in der aktuellen Version noch nicht korrekt).

- Bands (Eigenschaft, veröffentlicht)

```
property Bands: TQuickRepBands;
```

Die Eigenschaft *Bands* wird verwendet, um die Bänder (mit Ausnahme der Detailbänder) in den Report einzufügen. Folgende Bänder können eingefügt werden:

- Title wird einmal am Anfang des Reports gedruckt.
- Summary wird einmal am Ende des Reports gedruckt.
- PageHeader wird oben auf jeder Seite gedruckt. (Wenn *Options.FirstPageHeader* gleich *false* ist, dann wird das Header-Band nicht auf der ersten Seite gedruckt.)
- PageFooter wird unten auf jeder Seite gedruckt. (Wenn *Options.LastPageFooter* gleich *false* ist, dann wird das Footer-Band nicht auf der letzten Seite gedruckt.)
- ColumnHeader wird oberhalb jeder Spalte gedruckt.
- Das Detailband wird einmal pro (Master-)Datensatz eingefügt.

- Zoom (Eigenschaft, veröffentlicht)

```
property Zoom: integer;
```

Mit der Eigenschaft *Zoom* lassen sich Vergrößerungsfaktoren für den Entwurf einstellen.

- Preview, Print; PrinterSetup (Methoden)

```
procedure Preview;
procedure Print;
procedure PrinterSetup;
```

Mit der Methode *Preview* startet man die Anzeige des Berichts im QuickReport- oder im selbst erstellten Vorschaufenster. Mit *Print* wird der Ausdruck gestartet. Um den Drucker, die zu druckenden Seiten oder die Zahl der Kopien einzustellen, kann mit *PrinterSetup* ein Dialog geöffnet werden.

- NewPage, NewColumn (Methoden)

```
procedure NewPage;
procedure NewColumn;
```

  Mit *NewPage* wird ein Seitenumbruch, mit *NewColumn* wird ein Spaltenumbruch erzwungen.

- Printer (Eigenschaft, öffentlich)

```
property Printer: TQRPrinter;
```

  Die Eigenschaft *Printer* verweist auf das dazugehörige *TQRPrinter*-Objekt.

- ResetPageFooterSize (Methode)

```
procedure ResetPageFooterSize;
```

  QuickReport berechnet nicht für jedes Band den auf dieser Seite zur Verfügung stehenden Platz. Sollten Sie während eines Reports das Footer-Band an- oder abschalten, dann müssen Sie mit dieser Methode QuickReport davon informieren.

- OnBeforePrint, OnAfterPrint (Ereignisse)

```
property OnBeforePrint(Sender: TQuickRep;
  var PrintReport: Boolean);
property OnAfterPrint(Sender: TObject);
```

  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.

- OnAfterPreview (Ereignis)

```
property OnAfterPreview(Sender: TObject);
```

  Das Ereignis *OnAfterPreview* wird ausgelöst, wenn das Vorschaufenster geschlossen wird.

- OnStartPage, OnEndPage (Ereignisse)

```
property OnStartPage(Sender: TQuickRep);
property OnEndPage(Sender: TQuickRep);
```

  Das Ereignis *OnStartPage* wird aufgerufen, wenn eine neue Seite begonnen, das Ereignis *OnEndPage*, wenn eine Seite beendet wird.

- OnNeedData (Ereignis)

```
property OnNeedData(Sender: TObject; var MoreData: Boolean);
```

Stammen die Daten nicht aus einer Datenquelle, dann bleibt die Eigenschaft *DataSource* leer. Stattdessen wird eine Ereignisbehandlungsprozedur für das *OnNeedData*-Ereignis erstellt.

Die Daten werden dann direkt den einzelnen Komponenten wie beispielsweise *TQRLabel* zugewiesen. Mit dem Variablen-Parameter *MoreData* wird die Anzahl der einzufügenden Detailbänder gesteuert: Hat *MoreData* den Wert *true*, dann wird ein weiteres Band eingefügt. *MoreData* ist beim ersten Aufruf mit *true* und anschließend mit *false* vorbelegt – ohne eine entsprechende Ereignisbehandlungsroutine würde also exakt ein Detailband veröffentlicht.

10.6.2 Die Komponente TQRBand

Die einzelnen Elemente werden bei *QuickReport* auf Bändern, also auf *TQRBand*-Komponenten platziert. Für gewöhnlich werden Sie Bänder über die *TQuickRep*-Eigenschaft *Bands* einfügen.

- BandType (Eigenschaft, veröffentlicht)

```
property BandType: (rbTitle, rbPageHeader, rbDetail, rbPageFooter,
  rbSummary, rbGroupHeader, rbGroupFooter, rbSubDetail,
  rbColumnHeader, rbOverlay, rbChild);
```

Nach 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 Reportdaten platziert.
- Wenn ein Master-Detail-Report mit Hilfe der Komponente *QRDetailLink* entworfen wird, so stellt das Band *rbSubDetail* das Detailband 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* ist aus Gründen der Abwärtskompatibilität vorhanden und sollte nicht genutzt werden.
- Mit *rbChildBand* kann ein Band erstellt werden, das immer dem verbundenen Band folgt. Im Beispielprojekt in Kapitel 18 ist dazu eine Anwendung zu sehen.

- Enabled (Eigenschaft, veröffentlicht)

```
property Enabled: boolean;
```

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.

- ForceNewPage, ForceNewColumn (Eigenschaften, veröffentlicht)

```
property ForceNewPage: boolean;
property ForceNewColumn: boolean;
```

Wird die Eigenschaft *ForceNewPage* auf *true* gestellt, so wird für dieses Band immer eine neue Seite begonnen. Mit *ForceNewColumn* kann eine neue Spalte erzwungen werden.

- Frame, Color, Font (Eigenschaften, veröffentlicht)

```
property Frame: TQRFrame;
property Color: TColor;
property Font: TFont;
```

Mit der Eigenschaft *Frame* lässt sich ein Rahmen um das Band ziehen. (Sie müssen gegebenenfalls die Strichstärke auf 2 oder höher setzen.) Zur Gestaltung dienen außerdem die Eigenschaften *Color* und *Font*.

- LinkBand (Eigenschaft, veröffentlicht)

```
property LinkBand : TQRCustomBand
```

Mit *LinkBand* kann ein Band referenziert werden, das auf derselben Seite gedruckt werden soll wie das jeweilige Band.

- HasChild (Eigenschaft, veröffentlicht)

  ```
  property HasChild: boolean;
  ```

  Mit Hilfe der Eigenschaft *HasChild* kann ein untergeordnetes Band erzeugt werden.

- Size (Eigenschaft, veröffentlicht)

  ```
  property Size: TQRBandSize;
  ```

  Die Größe des Bandes in der aktuellen Einheit kann mit Hilfe der Eigenschaft *Size* eingestellt werden.

- OnBeforePrint, OnAfterPrint (Ereignisse)

  ```
  property OnBeforePrint(Sender: TQRCustomBand;
    var PrintBand: Boolean);
  property OnAfterPrint(Sender: TQRCustomBand; BandPrinted: Boolean);
  ```

  Die Ereignisse *OnBeforePrint* und *OnAfterPrint* werden erwartungsgemäß vor bzw. nach dem Druck des jeweiligen Bandes ausgelöst. Mit Hilfe des Variablen-Parameters *PrintBand* kann das Drucken des Bandes verhindert werden.

- AddPrintable (Methode)

  ```
  function AddPrintable
    (PrintableClass: TQRNewComponentClass): TQRPrintable;
  ```

  Mit *AddPrintable* werden zur Laufzeit weitere Komponenten hinzugefügt.

10.6.3 Die Komponente TQRSubDetail

Bei Master-Detail-Reports werden die Detaildatensätze auf *TQRSubDetail*-Bändern ausgegeben. Ein Beispiel für einen Master-Detail-Report finden Sie in Kapitel 10.5. *TQRSubDetail* ist wie *TQRBand* von *TQRCustomBand* abgeleitet. Viele Eigenschaften von *TQRSubDetail* sind deshalb auch bei *TQRBand* vorhanden. Ein Verweis auf *TQRBand* soll deshalb hier genügen.

- DataSet (Eigenschaft, veröffentlicht)

  ```
  property DataSet: TDataSet;
  ```

  Die Eigenschaft *DataSet* verweist auf die Datenquelle, welche das Detailband steuert.

  Für jeden Datensatz der Detaildatenmenge wird ein Detailband eingefügt. Achten Sie darauf, dass Sie eine funktionierende Master-Detail-Verbindung der Datenmengenkomponente selbst erstellen müssen.

- Master (Eigenschaft, veröffentlicht)

  ```
  property Master: TQRCustomBand;
  ```

  Die Eigenschaft *Master* wird auf diejenige Komponente gesetzt, welche der jeweiligen Detailtabelle übergeordnet ist.

  Bei einfachen Master-Detail-Reports wird dies die Komponente *QuickRep1* sein, werden mehrere Detailebenen verwendet, so kann dies auch eine (andere) *TQRSubDetail*-Komponente sein.

- Bands (Eigenschaft, veröffentlicht)

  ```
  property Bands : TQRSubDetailGroupBands
  ```

  Die Detailbänder können um ein Kopf- und ein Fußband ergänzt werden. Um ein Header-Band einzufügen, wird *Bands.HasHeader* auf *true* gesetzt, zum Einfügen eines Footer-Bandes wird *Bands.HasFooter* verwendet. Die Eigenschaften *HeaderBand* und *FooterBand* werden beim Einfügen dieser Bänder automatisch richtig gesetzt.

- Enabled (veröffentlicht, TQRBand)
- ForceNewPage, (veröffentlicht, TQRBand)
- ForceNewColumn (veröffentlicht, TQRBand)
- Frame (veröffentlicht, TQRBand)
- Color (veröffentlicht, TQRBand)
- Font (veröffentlicht, TQRBand)
- HasChild (veröffentlicht, TQRBand)
- Size (veröffentlicht, TQRBand)
- OnBeforePrint (Ereignis, TQRBand)
- OnAfterPrint (Ereignis, TQRBand)
- OnNeedData (Ereignis, TQRSubDetail)

  ```
  property OnNeedData(Sender: TObject; var MoreData: Boolean);
  ```

  Stammen die Daten nicht aus einer Datenquelle, dann bleibt die Eigenschaft *DataSource* leer. Stattdessen wird eine Ereignisbehandlungsprozedur für das *OnNeedData*-Ereignis erstellt.

  Näheres dazu in Kapitel 10.6.1 (*TQuickRep*).

10.6.4 Die Komponente TQRGroup

Die Komponente *TQRGroup* ist ein Band, welches dazu dient, angezeigte Datensätze voneinander zu trennen und auf diese Weise zu gruppieren. Ein Beispiel dazu ist in Kapitel 10.2 zu finden.

- Expression (Eigenschaft, veröffentlicht)

  ```
  property Expression: string;
  ```

  Mit Hilfe der Eigenschaft *Expression* wird spezifiziert, nach welchem Ausdruck die Datensätze gruppiert werden sollen. Für diese Eigenschaft ist ein Eigenschaftseditor implementiert. Dort werden auch alle vorhandenen Funktionen kurz erläutert, so dass dies hier unterbleiben kann.

- HasChild (Eigenschaft, veröffentlicht)

  ```
  property HasChild: boolean;
  ```

  Mit Hilfe der Eigenschaft *HasChild* kann ein untergeordnetes Band erzeugt werden.

10.6.5 TQRChildBand

Wenn Sie die Eigenschaft *HasChild* eines Bandes auf *true* setzen, dann wird automatisch eine *TQRChildBand*-Instanz mit dem jeweiligen Band verknüpft. Sie können eine solche Komponente jedoch auch über die Komponentenpalette einfügen. Das Kind-Band wird direkt im Anschluss an das Eltern-Band gedruckt. Sinn macht ein Kind-Band beispielsweise, wenn ein Eltern-Band Komponenten enthält, deren Wert *AutoStretch* den Wert *true* hat. In diesem Fall passen sich die Abmessungen des Eltern-Bandes denen der betreffenden Komponenten an. Sollen nach diesen Komponenten weitere gedruckt werden, so sind diese auf ein Kind-Band zu platzieren.

- ParentBand (Eigenschaft, veröffentlicht)

  ```
  property ParentBand : TQRCustomBand
  ```

  Mit *ParentBand* wird das Eltern-Band eingestellt.

- LinkBand (Eigenschaft, veröffentlicht)

  ```
  property LinkBand : TQRCustomBand
  ```

  Ein Kind-Band kann ein weiteres Kind-Band haben. Soll sichergestellt werden, dass dieses weitere Kind-Band auf derselben Seite gedruckt wird, dann ist die Eigenschaft *LinkBand* (des dann als Eltern-Band fungierenden Bandes) auf das Kind-Band zu setzen.

10.6.6 TStringsBand

TStringsBand dient dazu, mehrzeiligen Text in einen Report aufzunehmen.

- Items (Eigenschaft veröffentlicht)

```
property Items: TStrings;
```

Für jede Zeile in der *TStrings*-Eigenschaft *Items* wird eine Kopie von *TStringsBand* in den Report eingefügt. Ist der Report nicht mit einer Datenmenge verknüpft, so werden diese Bänder nur einmal eingefügt. Ist der Report mit einer Datenmenge verknüpft, so wird für jeden Datensatz ein Satz dieser Bänder eingefügt.

Auf die Zeilen in der Eigenschaft *Items* kann mittels der Komponente *TQRExpr* zugegriffen werden. Öffnen Sie dazu den Eigenschaftseditor der Eigenschaft *Expression* und wählen dann den Button *Variable*. Wählen Sie dann den Eintrag mit dem Komponentennamen des Bandes:

- Index, Item (Eigenschaften, öffentlich, nur Lesen)

```
property Index: Integer;
property Item: string;
```

Auf den Index des aktuellen Eintrages kann mit *Index*, auf den Eintrag selbst mit *Item* zugegriffen werden.

10.6.7 TQRLabel, TQRMemo und TQRDBText

Die Komponenten *TQRLabel* und *TQRMemo* dienen zur Ausgabe von statischem Text. Während *TQRLabel* der auszugebende Text als String zugewiesen wird, nimmt *TQRMemo* eine String-Liste entgegen. Mit *TQRDBText* werden Felder aus einer Datenbank ausgegeben – hier ist sowohl einzeiliger als auch mehrzeiliger Text möglich.

Der auszugebende Text

- Caption (Eigenschaft, veröffentlicht, TQRLabel)

  ```
  property Caption: string;
  ```

  Der Eigenschaft *Caption* wird der zu druckende Text zugewiesen.

- Lines (Eigenschaft, veröffentlicht, TQRMemo)

  ```
  property Lines: TStrings;
  ```

  Der Eigenschaft *Lines* wird der auszugebende Text zugewiesen.

- DataSet, DataField (Eigenschaften, veröffentlicht, TQRDBText)

  ```
  property DataSet: TDataSet;
  property DataField: string;
  ```

  Mit Hilfe dieser Eigenschaften wird spezifiziert, von welcher Datenmenge und welcher Spalte der auszugebende Text bezogen wird.

Formatierung des Textes

- Alignment, AlignToBand (Eigenschaften, veröffentlicht)

  ```
  property Alignment: TAlignment;
  property AlignToBand: boolean;
  ```

  Mit Hilfe der Eigenschaft *Alignment* kann bestimmt werden, ob der Text links, rechts oder zentriert in der Komponente ausgegeben wird. Wird *AlignToBand* auf *true* gesetzt, dann wird der Text an den Bandgrenzen ausgerichtet. Dies ist besonders dann hilfreich, wenn damit zu rechnen ist, dass der Report von Hoch- auf Querformat (oder umgekehrt) umgestellt wird.

- Transparent (Eigenschaft, veröffentlicht)

  ```
  property Transparent: boolean;
  ```

  Wird *Transparent* auf *true* gesetzt, dann scheint der Hintergrund durch den Text, anderenfalls wird der Text mit weißem Hintergrund hinterlegt. Wenn Sie dem Band eine Farbe zuweisen, dann sollte *Transparent* auf *true* gesetzt werden.

- AutoSize, AutoStretch, WordWrap (Eigenschaften, veröffentlicht)

```
property AutoSize: boolean;
property AutoStretch: boolean;
property WordWrap: boolean;
```

  Wird *AutoSize* auf *true* gesetzt, dann wird die Breite der Komponente des auszugebenden Textes angepasst. Soll die Höhe der Komponente dem auszugebenden Text angepasst werden, dann ist *AutoSize* auf *false* und *AutoStretch* sowie *WordWrap* auf *true* zu setzen. *WordWrap* sorgt dafür, dass der Text umgebrochen wird.

- Frame, Color, Font (Eigenschaften, veröffentlicht)

```
property Frame: TQRFrame;
property Color: TColor;
property Font: TFont;
```

  Mit der Eigenschaft *Frame* lässt sich ein Rahmen um den Text ziehen. Zur Gestaltung dienen außerdem die Eigenschaften *Color* und *Font*.

- Mask (Eigenschaft, veröffentlicht, TQRDBText)

```
property Mask: string;
```

  Für gewöhnlich werden die Formatierungsanweisungen der *TField*-Komponenten übernommen. Bisweilen kann es jedoch erforderlich sein, davon abweichende Formatierungen zu verwenden. Diese werden mit der Eigenschaft *Mask* eingestellt. Um den Text beispielsweise mit zwei Nachkommastellen und dem Währungssymbol *DM* zu formatieren, weist man der Eigenschaft *Mask* den String *0.00 DM* zu.

- OnPrint (Ereignis, TQRLabel, TQRDBText)

```
property OnPrint(sender: TObject; var Value: String);
```

  Das Ereignis *OnPrint* wird ausgelöst, wenn der Text in die Vorschau oder zum Drucker ausgegeben wird.

10.6.8 Die Komponente TQRExpr

Bisweilen müssen in einem Report einfache Berechnungen durchgeführt werden, beispielsweise die Summe aller Abteilungsergebnisse. Prinzipiell könnte man dazu eine neue *TQuery*-Komponente verwenden und die Rechenanweisung als SQL-Befehl formulieren. Mit der Komponente *TQRExpr* geht es aber in der Regel einfacher. Viele Eigenschaften zur Formatierung des Textes sind in derselben Form bei *TQRLabel* vorhanden und sollen hier nicht erneut erläutert werden.

- Expression (Eigenschaft, veröffentlicht)

  ```
  property Expression: string;
  ```

  Mit Hilfe der Eigenschaft *Expression* wird spezifiziert, welchen Ausdruck diese Komponente auswerten und ausgeben soll. Für diese Eigenschaft ist ein Eigenschaftseditor implementiert. Dort werden auch alle vorhandenen Funktionen kurz erläutert, so dass dies hier unterbleiben kann. Beispiele für den Einsatz der Komponente *TQRExpr* und der Eigenschaft *Expression* finden sich in Kapitel 10.5.

- ResetAfterPrint (Eigenschaft, veröffentlicht)

  ```
  property ResetAfterPrint: boolean;
  ```

  Wird *ResetAfterPrint* auf *true* gesetzt, dann wird die Komponente nach der Ausgabe zurückgesetzt. Sollen beispielsweise die Datensätze durchnummeriert werden, würde man dafür die Eigenschaft COUNT verwenden, welche eine interne Variable hochzählt. Hier muss *ResetAfterPrint* auf *false* gesetzt werden. Soll dagegen die Anzahl der Sub-Detailbänder im Sub-Detail-Footer angezeigt werden, dann muss *ResetAfterPrint* auf true gesetzt werden, damit bei jedem Master-Band erneut mit der Zählung begonnen wird.

10.6.9 Die Komponente TQRSysData

Die Komponente *TQRSysData* dient zur Ausgabe von *QuickReport*-internen Daten. Auch diese Komponente gleicht weitgehend der Komponente *TQRLabel*. Zusätzlich sind hier die folgenden beiden Eigenschaften vorhanden:

- Data (Eigenschaft, veröffentlicht)

  ```
  property Data: TQRSysDataType;
  ```

  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.
- *qrsColumnNo* beinhaltet die aktuelle Spaltennummer und ist bei einspaltigen Reports immer gleich 1.

- Text (Eigenschaft, veröffentlicht)

  ```
  property Text: string;
  ```

  Die Eigenschaft *Text* ermöglicht es, dem eigentlichen Wert noch einen (erläuternden) Text voranzustellen. Die Ausgabe lautet dann beispielsweise nicht *7*, sondern *Seite 7*.

10.6.10 TQRRichText und TQRDBRichText

Die Komponenten *TQRRichText* und *TQRDBRichText* dienen zur Anzeige von formatiertem Text. Während *TQRRichText* den anzuzeigenden Text zur Entwurfszeit zugewiesen bekommt oder aus einer anderen Komponente bezieht, zeigt *TQRDBRichText* in Datenbanken gespeicherte Texte an.

- Lines, ParentRichEdit (Eigenschaften, veröffentlicht)

  ```
  property Lines: TStrings;
  property ParentRichEdit: TRichEdit;
  ```

  Der Eigenschaft *Lines* wird der anzuzeigende Text zugewiesen. Alternativ dazu kann mittels *ParentRichEdit* eine Referenz auf eine *TRichEdit*-Instanz erstellt werden, aus der dann der anzuzeigende Text bezogen wird.

- DataSet, DataField (Eigenschaften, veröffentlicht)

  ```
  property DataSet: TDataSet;
  property DataField: string;
  ```

  Mit Hilfe der Eigenschaften *DataSet* und *DataField* wird spezifiziert, aus welcher Datenmenge und welcher Spalte dort der Text bezogen wird.

10.6.11 TQRImage und TQRDBImage

Zur Anzeige von Bildern werden die Komponenten *TQRImage* und *TQRDBImage* verwendet. Während *TQRImage* das anzuzeigende Bild zur Entwurfszeit zugewiesen bekommt oder aus einer Datei bezieht, zeigt *TQRDBImage* in Datenbanken gespeicherte Bilder an.

- Picture (Eigenschaft, veröffentlicht, TQRImage)

  ```
  property Picture: TPicture;
  ```

  Der Eigenschaft *Picture* wird das anzuzeigende Bild zugewiesen.

- DataSet, DataField (Eigenschaften, veröffentlicht, TQRDBImage)

```
property DataSet: TDataSet;
property DataField: string;
```

  Mit Hilfe der Eigenschaften *DataSet* und *DataField* wird spezifiziert, aus welcher Datenmenge und welcher Spalte dort die Bilder bezogen werden.

- Center, Stretch (Eigenschaften, veröffentlicht)

```
property Center: boolean;
property Stretch: boolean;
```

  Normalerweise wird das Bild an der linken oberen Ecke der Komponente ausgerichtet. Wird die Eigenschaft *Center* auf *true* gesetzt, dann wird das Bild in der Komponentenmitte ausgerichtet. Wird die Eigenschaft *Stretch* auf *true* gesetzt, dann wird das Bild auf Komponentenmaße gedehnt oder gestaucht.

- Autosize (Eigenschaft, veröffentlicht, TQRImage)

```
property Autosize: boolean;
```

  Ist *Autosize* gleich *true*, dann passen sich die Komponentenmaße dem anzuzeigenden Bild an.

10.6.12 Die Komponente TQRShape

Um Linien und geometrische Formen in den Report zu zeichnen, wird die Komponente TQRShape verwendet.

- Shape (Eigenschaft, veröffentlicht)

```
property Shape: TQRShapeType;
```

  Die zu zeichnende Form wird mit Hilfe der Eigenschaft *Shape* spezifiziert. Hierbei sind folgende Werte möglich:

  - *qrsCircle:* Kreis
  - *qrsRectangle:* Rechteck
  - *qrsHorLine:* horizontale Linie in der Mitte der Komponente
  - *qrsVertLine:* vertikale Linie in der Mitte der Komponente
  - *qrsRightAndLeft:* vertikale Linien am linken und rechten Rand
  - *qrsTopAndBottom:* horizontale Linien am oberen und unteren Rand

- Pen, Brush (Eigenschaften, veröffentlicht)

```
property Brush: TBrush;
property Pen: TPen;
```

  Mit Hilfe der Eigenschaft *Pen* wird die Umrisslinie gestaltet. Die Eigenschaft *Brush* legt fest, wie die Fläche innerhalb der Form dargestellt wird.

10.6.13 Die Filterkomponenten

Standardmäßig kann QuickReport Reports im eigenen Format speichern (und auch wieder öffnen). Mit Hilfe der Filterkomponenten besteht die Möglichkeit, einen Report auch in anderen Formaten zu speichern. Dazu muss lediglich die entsprechende Komponente in das Projekt eingefügt werden. Momentan stehen folgende Filter zur Verfügung:

- Die Komponente *TQRTextFilter* speichert im ANSI-Format.
- Die Komponente *TQRCSVFilter* speichert auch im ANSI-Format; das Trennzeichen zwischen den Spalten kann hier mit der Eigenschaft *Seperator* eingestellt werden.
- Die Komponente *TQRHTMLFilter* speichert den Text (und leider nicht die Bilder) eines Reports im HTML-Format.

10.6.14 Die Komponente TQRCompositeReport

Bisweilen steht man vor dem Problem, dass man mehrere Reports zu einem einzigen zusammenfassen möchte. Zu diesem Zweck verwendet man die Komponente *TQRCompositeReport*. Dabei wird auf der letzten Seite eines Reports gleich mit dem nächsten begonnen.

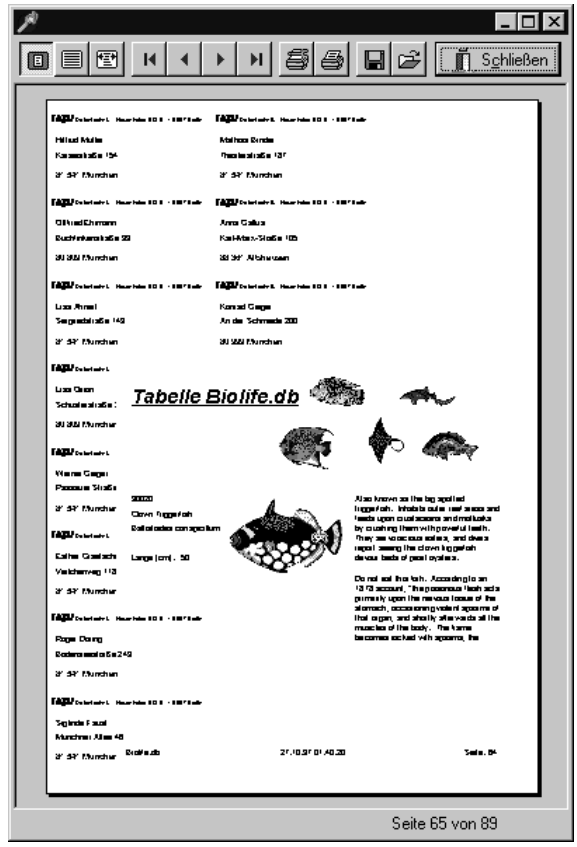
Dies mag in der Regel so erwünscht sein – ansonsten könnte man ja auch mehrere Reports erstellen und drucken –, es führt jedoch bisweilen zu sehr kuriosen Ergebnissen. Man kann hier den Beginn einer neuen Seite auch dadurch nicht erzwingen, dass die Eigenschaft *ForceNewPage* des Titel-Bandes auf *true* gesetzt wird.

Die Komponente *TQRCompositeReport* wird nicht in ein Reportformular, sondern in ein Formular der Anwendung eingefügt. Um die Vorschau zu starten, ruft man die Methode *Preview* von *TQRCompositeReport* auf.

```
procedure TForm1.Button6Click(Sender: TObject);
begin
  QRCompositeReport1.Preview;
end;
```

Damit die Komponente weiß, welche Reports zusammengefasst werden sollen, muss auf das Ereignis *OnAddReports* entsprechend reagiert werden.

```
procedure TForm1.QRCompositeReport1AddReports(Sender: TObject);
begin
  with QRCompositeReport1 do
  begin
    Reports.Add(Form2.QuickRep1);
      ...
    Reports.Add(Form7.QuickRep1);
  end;
end;
```



11 InterBase 6

Zum Entwickeln »richtiger« Datenbankanwendungen liegt der InterBase-Server den Delphi-Versionen *Professional* und *Enterprise* bei.

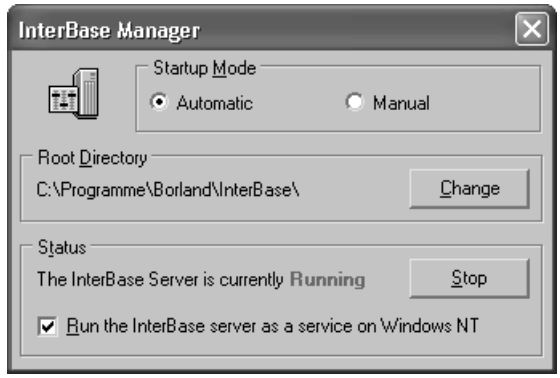
- Der Local-Server, der diesen beiden Versionen beiliegt, ist auf zwei Datenbankverbindungen beschränkt.
- Eine 5er-Version des InterBase-Servers liegt lediglich der Enterprise-Version bei.

11.1 Starten des Servers

Per Voreinstellung wird unter NT der InterBase Server und der *Guardian* als Service betrieben. Der *Guardian* ist der Wächter, der auf den Server aufpasst. Läuft aus irgendwelchen Gründen der Server nicht mehr, dann wird er vom Guardian neu gestartet.

(Das mag nun den Verdacht erregen, dass es mit der Stabilität des Servers nicht zum Besten bestellt ist. Dieser Verdacht ist zumindest so lange unbegründet, solange der InterBase-Server alleine läuft. Nun hat man aber die Möglichkeit, für so genannte *user defined functions* eigene DLLs einzubinden. Zumindest während der Entwicklungsphase solcher UDFs häufen sich nach meiner Erfahrung die Server-Abstürze.)

Mit dem InterBase-Manager können Sie einstellen, ob der Server automatisch gestartet werden soll oder manuell und ob er als Service läuft oder als Anwendung.



InterBase als Anwendung

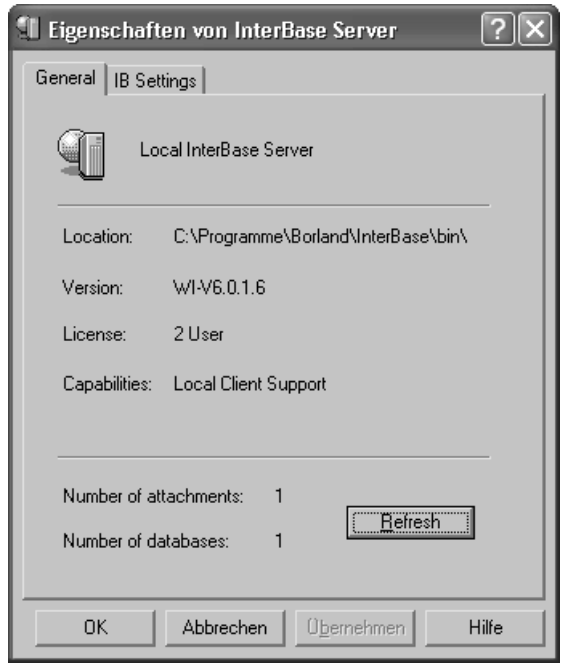
Normalerweise läuft InterBase als Service und erledigt brav im Hintergrund seine Arbeit. Sie können InterBase jedoch auch als Anwendung betreiben – wir werden uns gleich ansehen, welchen Vorteil das hat. Vorher sollten sie jedoch den Service beenden.

Der InterBase-Server wird nicht direkt gestartet, sondern man startet den Guardian. Dieser stellt fest, dass der Server nicht läuft und startet diesen. Soll InterBase als Anwendung betrieben werden, dann ist beim Start des Guardians ein *-a* anzuhängen.

```
bin\ibguard.exe -a
```

In der Taskbar-Notofication-Area (also rechts unten) wird der Guardian nun durch ein kleines Icon repräsentiert:

Über das Kontextmenü können Sie nun nicht nur den Server herunterfahren, sondern sich auch die aktuellen Eigenschaftswerte ansehen. Der Guardian informiert dabei lediglich über die Versionsnummer und die Uhrzeit, zu der er gestartet wurde – beides ist nur mäßig interessant.



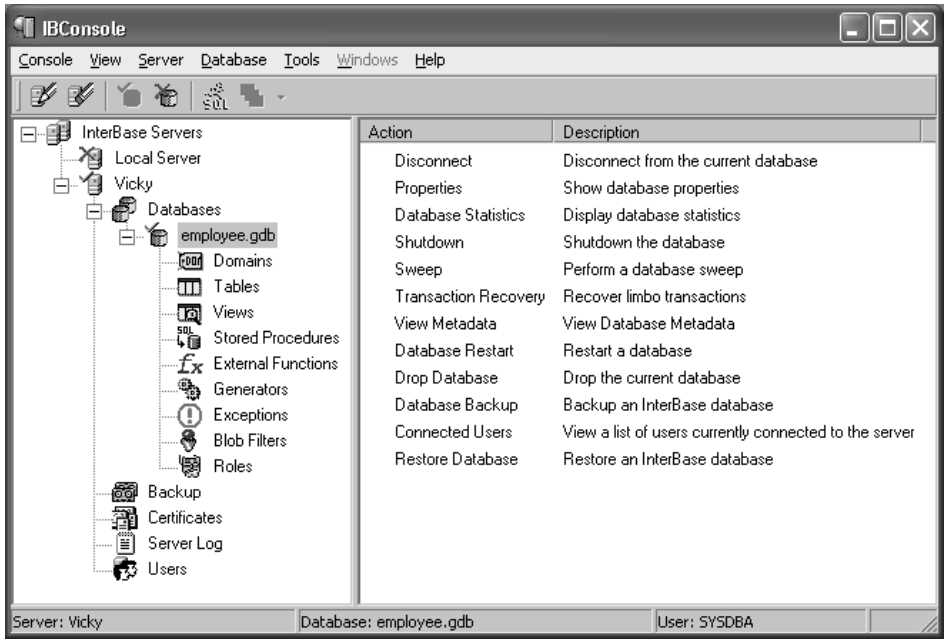
In der Eigenschaftsseite des Servers sehen Sie jedoch, wie viele Verbindungen gerade bestehen und auf wie viele Datenbanken dabei zugegriffen wird, ohne dafür *IBConsole* starten zu müssen.

Die Registerseite *IBSettings* können Sie dagegen getrost ignorieren.

11.2 IBConsole

Zur Verwaltung von InterBase-Servern gibt es das Programm IBConsole. Von dort aus können Sie Benutzer anlegen, Backups ziehen (und natürlich auch wieder einspielen) oder *Interactive SQL* aufrufen.

Für jede registrierte Datenbank können Sie die Daten und die Metadaten ansehen – Letztere sowohl als SQL-Script als auch in Tabellenform.



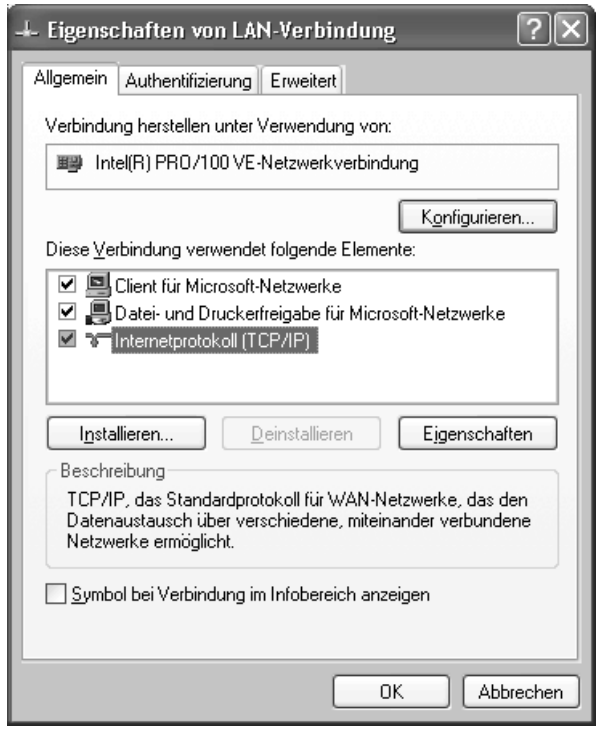
Per Voreinstellung ist lediglich der lokale Server registriert, also der Server, der auf demselben Rechner läuft. Sie können sich jedoch mit beliebigen anderen Servern verbinden.

11.2.1 Registrieren einer Server-Verbindung

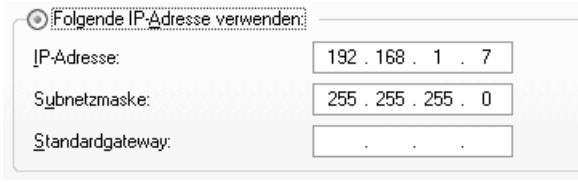
Wir wollen nun auf eine Datenbank zugreifen, die auf einem anderen Rechner liegt. Dieser muss natürlich über ein funktionierendes Netzwerk verbunden sein. Hier im Beispiel soll die Verbindung über eine TCP/IP-Verbindung eingerichtet werden.

Ermitteln der IP-Adresse

Zunächst benötigen wir die IP-Adresse des Servers. Dazu rufen wir in der *Systemsteuerung* die *Netzwerkverbindungen* auf. Hier interessiert uns das TCP/IP-Protokoll. Ist dies nicht vorhanden, dann muss es nachinstalliert werden.



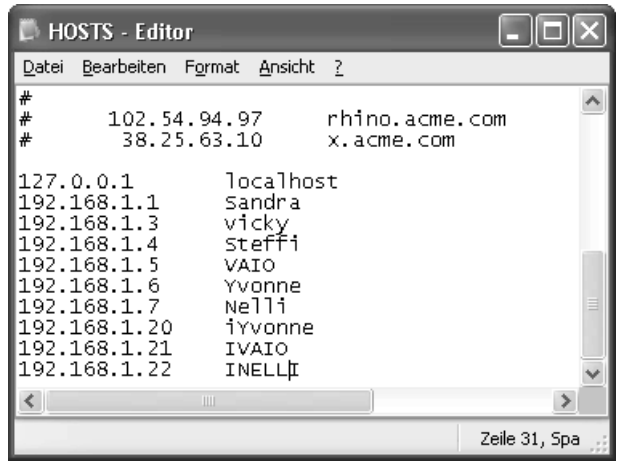
Nun klicken wir auf *Eigenschaften*:



Je nach Windows-Version sieht das immer ein wenig anders aus, aber irgendwo ist die IP-Adresse zu finden, die wir uns notieren.

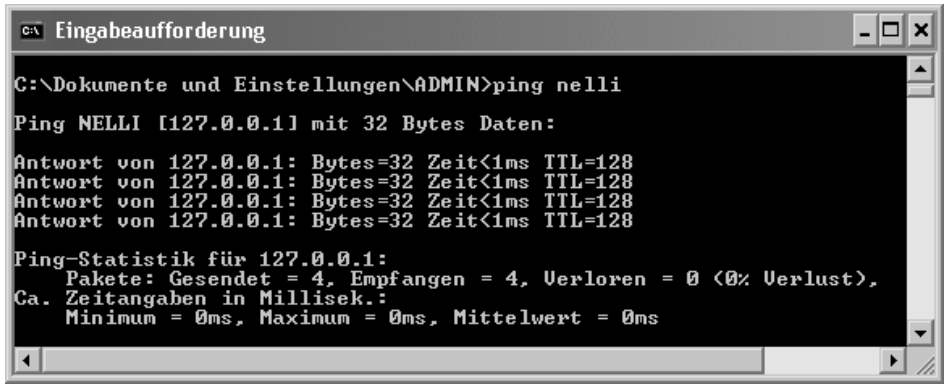
Einrichten eines Host-Namens

Die folgenden Schritte erfolgen nun auf dem Client. Dort öffnen wir die Datei *Hosts*, die irgendwo im Windows-Verzeichnis liegt und mit dem (Text-)Editor geöffnet werden kann. Dort werden für die IP-Adressen Host-Namen vergeben.



Hier gibt es zwei Möglichkeiten: Entweder ist für die IP-Adresse des Servers schon ein Host-Namen vergeben, dann muss dieser lediglich notiert werden. Anderenfalls muss hier ein neuer Eintrag vorgenommen werden.

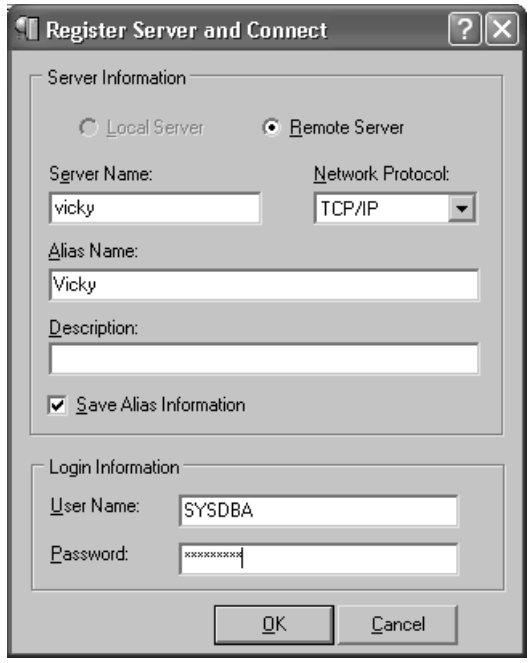
Um sicherzugehen, dass die Verbindung steht, wechseln wir in die Eingabeaufforderung und setzen einen Ping ab (Das Kommando *ping,* ein Leerzeichen und dann der Host-Name).



Wenn die Verbindung steht, dann sollte der Server antworten. Wenn er das nicht tut, dann ist irgendwo ein Fehler (Netzwerkkabel steckt nicht, Netzwerk ist nicht oder falsch eingerichtet, oder – wird zunehmend häufiger – eine Firewall verhindert die Verbindungsaufnahme).

Server registrieren

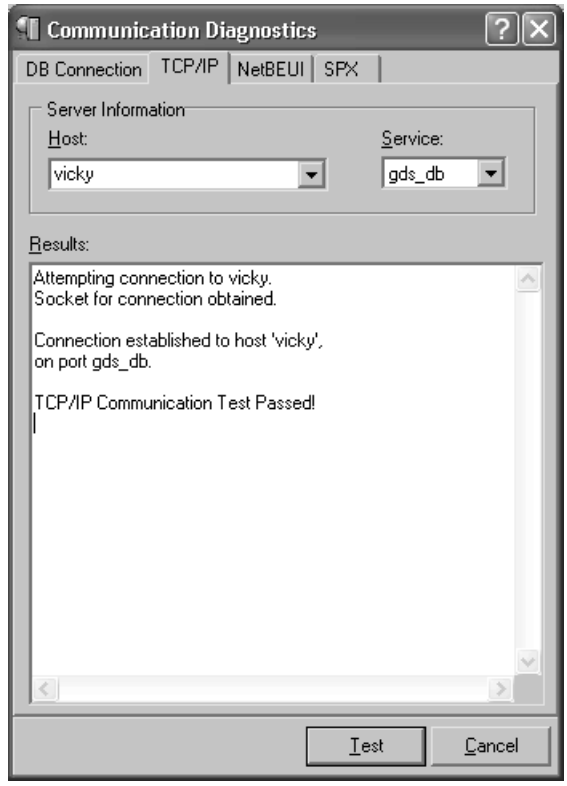
Starten Sie nun *IBConsole* und rufen Sie dort aus dem Kontextmenü des Root-Knotens den Menüpunkt *Register* auf.



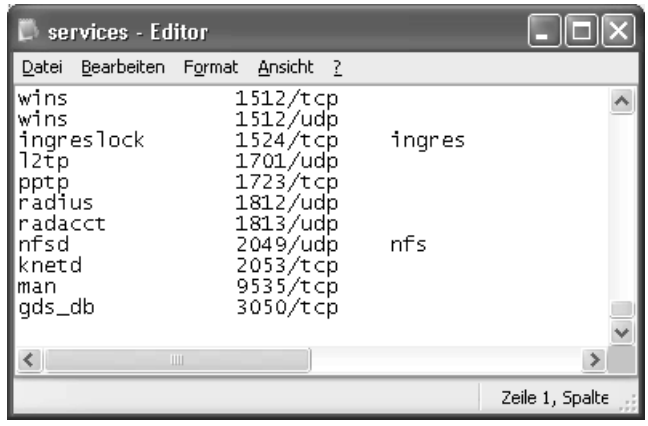
Als *ServerName* müssen Sie den Host-Namen eingeben, auf dem der Server läuft, als *Network Protocol* wählen wir TCP/IP. Der Alias-Name kann frei vergeben werden. *User Name* ist – so lange nichts anderes eingestellt wurde – *SYSDBA*, *Password* ist *masterkey*.

Verbindung testen

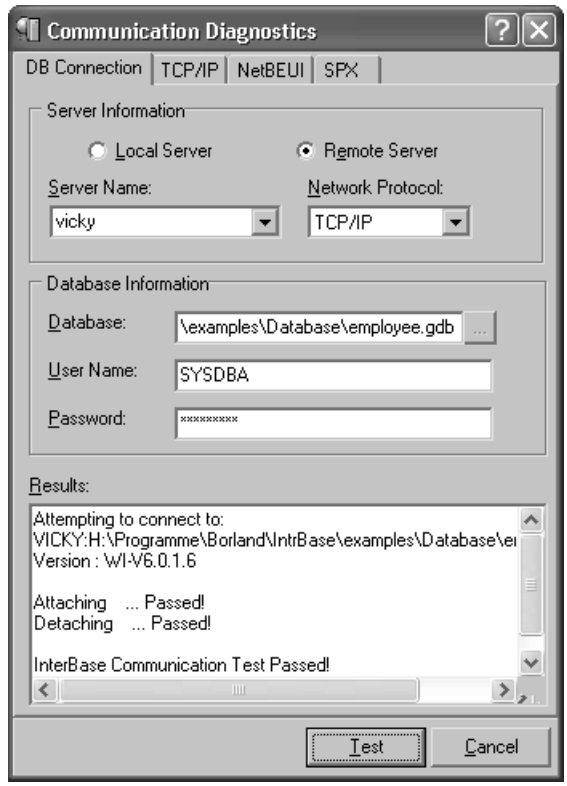
Nun rufen wir aus dem Kontextmenü *Communication Diagnostics* auf und wechseln zunächst auf die Registerseite *TCP/IP*. Hier wählen wir den Host-Namen und als *Service* den Wert *gds_db*. Schlägt die Verbindung fehl, dann versuchen Sie statt *gds_db* mal die Nummer 3050.



Funktioniert 3050, nicht jedoch *gds_db*, dann fehlt der Eintrag in der Datei *Services*, die auch irgendwo im Windows-Verzeichnis liegt. Hier wird der Servicename (*gds_db*) der Port-Nummer (3050) zugeordnet.



Geht auch 3050 nicht, aber der Ping, dann dürfte irgendwo eine Firewall sitzen, die den Port 3050 sperrt – dies wäre dann zu ändern.

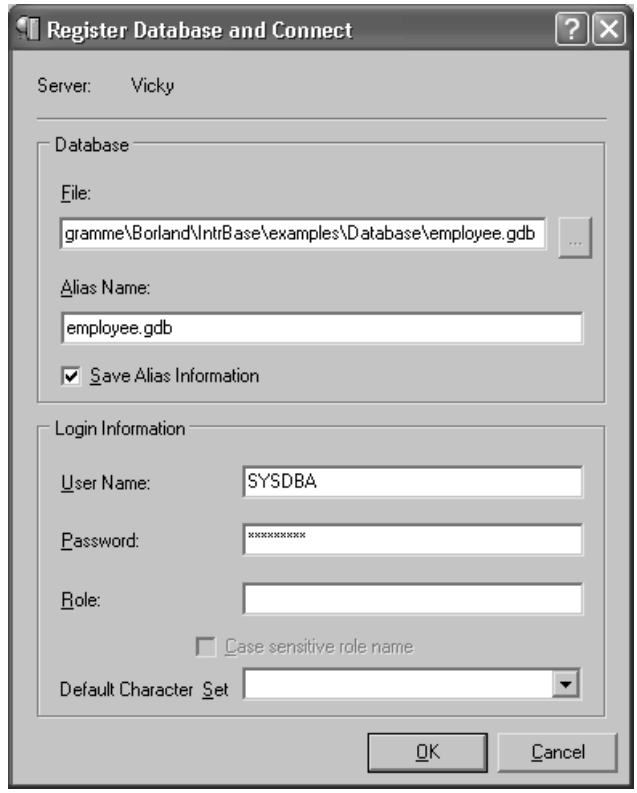


Nun gehen wir auf die Registerseite *DB Connection*. Hier testen wir nicht die Verbindung zu einem Server, sondern gleich zu einer Datenbank – also brauchen wir zusätzlich den Dateinamen der Datenbank sowie die Login-Informationen.

Registrieren der Datenbank

Ist die Verbindung erfolgreich getestet, dann ist es an der Zeit, die Datenbank zu registrieren. Dazu wird aus dem Kontextmenü des betreffenden Servers der Menüpunkt *Register* aufgerufen. Hier geben Sie zunächst den Dateinamen ein. Da ein solcher Dateiname ziemlich lang werden kann, können Sie einen Alias-Namen vergeben, der stattdessen in der Baumansicht angezeigt wird.

Die Login-Informationen werden gespeichert, so dass sie später nicht noch mal eingegeben werden müssen. Des Weiteren sollte der Zeichensatz angegeben werden, der in der Datenbank verwendet wird.

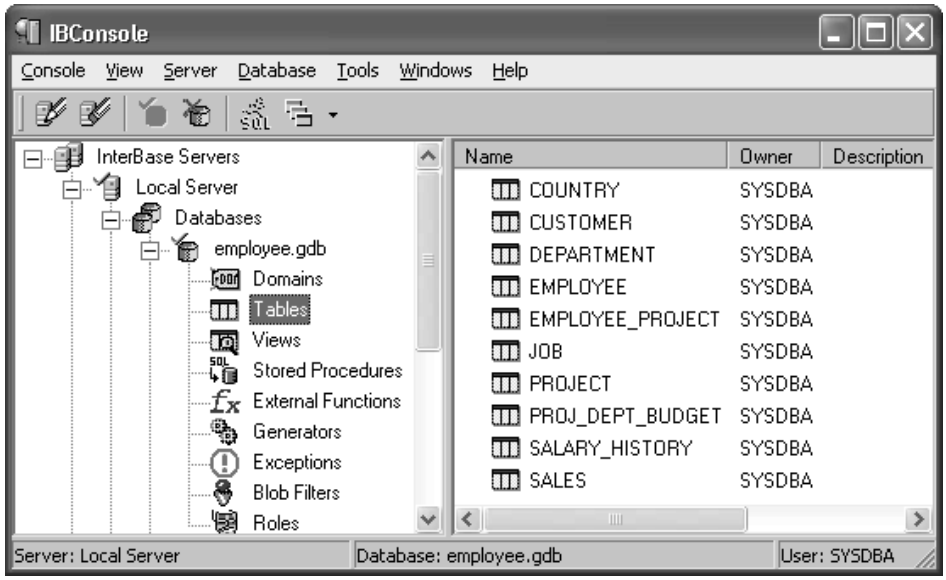


11.2.2 Einsehen der Metadaten

Während der Entwicklungsphase dürfte IBConsole vor allem dafür verwendet werden, die Metadaten einzusehen. Unter der jeweiligen Datenbank finden Sie die Einträge für Tabellen, STORED PROCEDURES, VIEWS und so weiter. Wählen Sie einen dieser Einträge, dann wird auf der rechten Seite angezeigt, was in dieser Kategorie so alles in der Datenbank steckt.

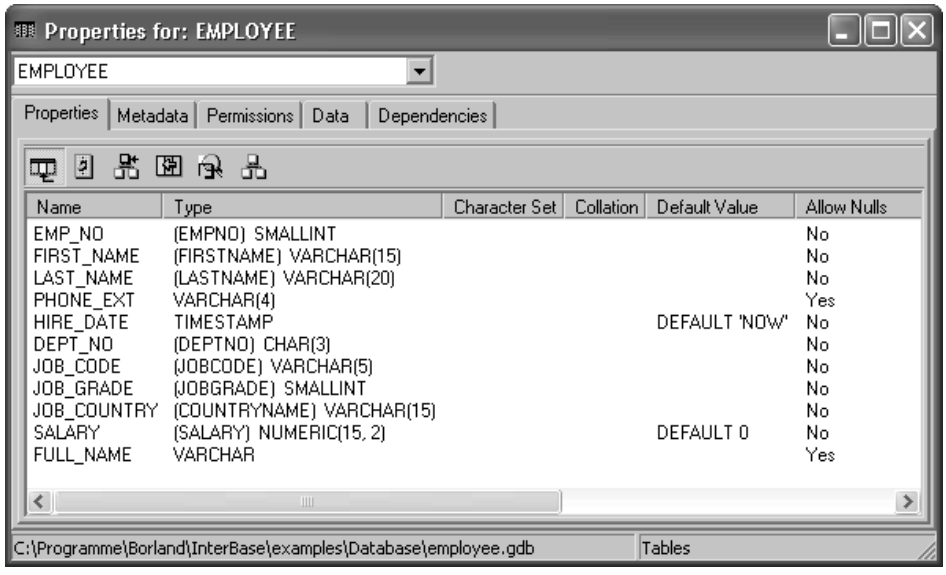
Über das Kontextmenü (EDIT DESCRIPTION) können Sie einen Editor aufrufen, um eine Beschreibung zu dem jeweiligen Element einzugeben – diese Beschreibung wird in der Liste mit angezeigt.

Mit EXTRACT aus dem Kontextmenü können Sie die Metadaten als SQL-Script erhalten. Bei Tabellen sind hier gleich die DOMAIN-Definitionen und die Definitionen der TRIGGER dabei.



Mit einem Doppelklick auf einen Eintrag öffnen Sie ein Fenster, in dem Sie detailliertere Informationen finden. Wir wollen uns das am Beispiel der Tabellen einmal ansehen.

Auf der Registerseite *Properties* finden Sie Informationen in tabellenartiger Darstellung:

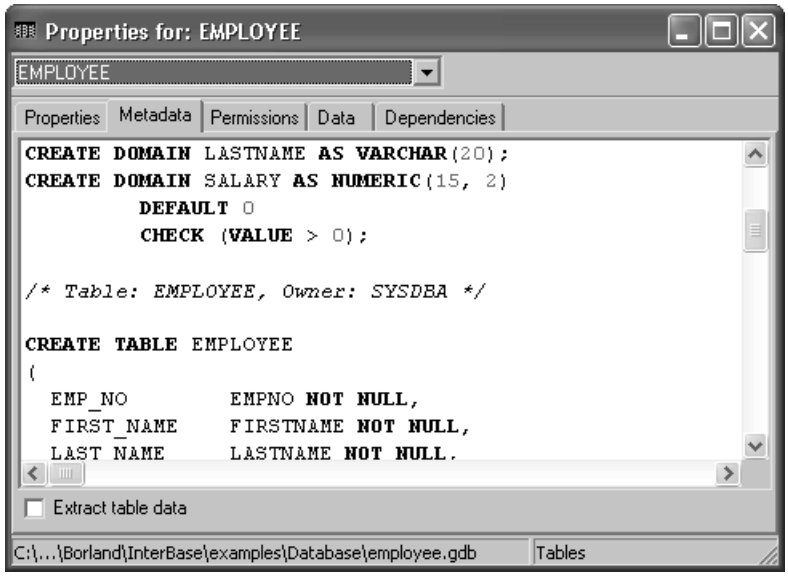


Mit Hilfe der Icons links oben können Sie einstellen, wozu Sie Formationen benötigen. Sie finden hier:

- Eine Liste der Spalten. Beim Spaltentyp sehen sie wowohl die DOMAIN (wenn eine definiert wurde) als auch den zugrunde liegenden Typ. Zeichensätze und Sortierreihenfolgen werden hier ebenso dargestellt wie Vorgabewerte oder die Erforderlichkeit einer Eingabe.
- Die Liste der TRIGGER, die dann auch gleich im SQL-Code dargestellt werden.
- Alle CONSTRAINTS, also Gültigkeitsprüfungen. Auch hier wird der SQL-Code mit angezeigt.
- Eine Liste alle Indizes, egal, ob sie selbst definiert wurden oder vom System für Primär-, Sekundär- oder Fremdschlüssel angelegt wurden.
- Ein Liste aller Sekundärschlüssel
- Und natürlich eine Liste aller Fremdschlüssel inklusive der Tabellen, die sie referenzieren, der Update- und der Delete-Regel und einiger weiterer Informationen.

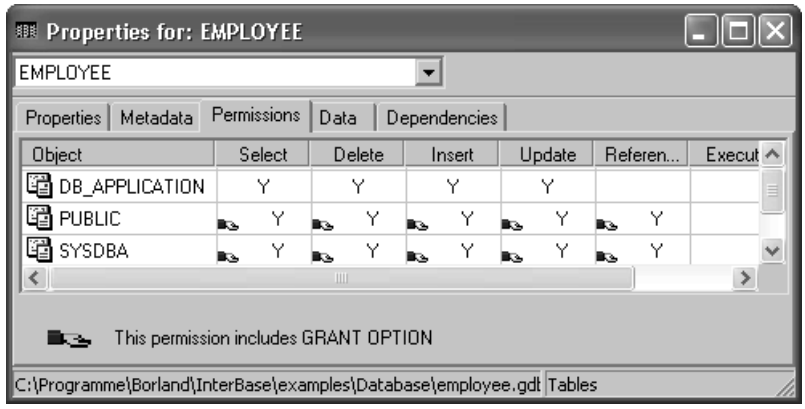
Die Metadaten

Die Metadaten zeigen nicht nur die Definition der Tabelle, sondern auch die dabei verwendeten DOMAINS sowie die TRIGGER. Das SQL-Script kann man über die Zwischenablage kopieren und in anderen Projekten verwenden.



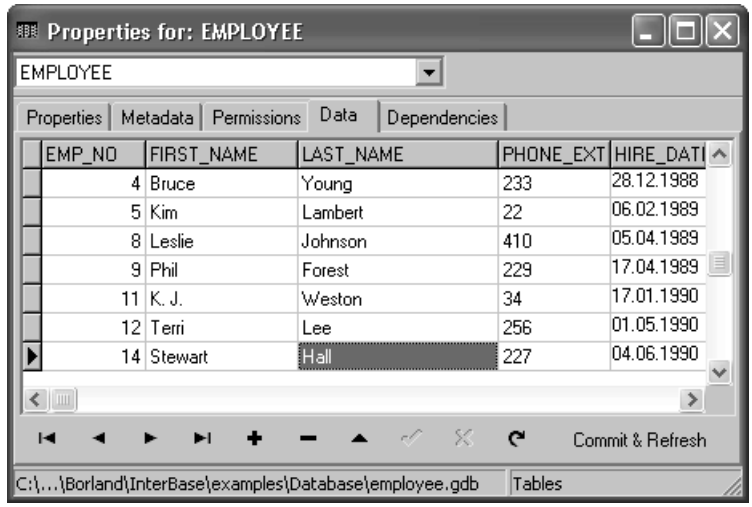
Die Rechte

Auf der Registerseite *Permissions* findet man die Rechte, die für die Tabelle vergeben sind.



Die Daten

Auf der Registerseite *Data* findet man die Daten, die in dieser Tabelle enthalten sind, und kann diese auch editieren.

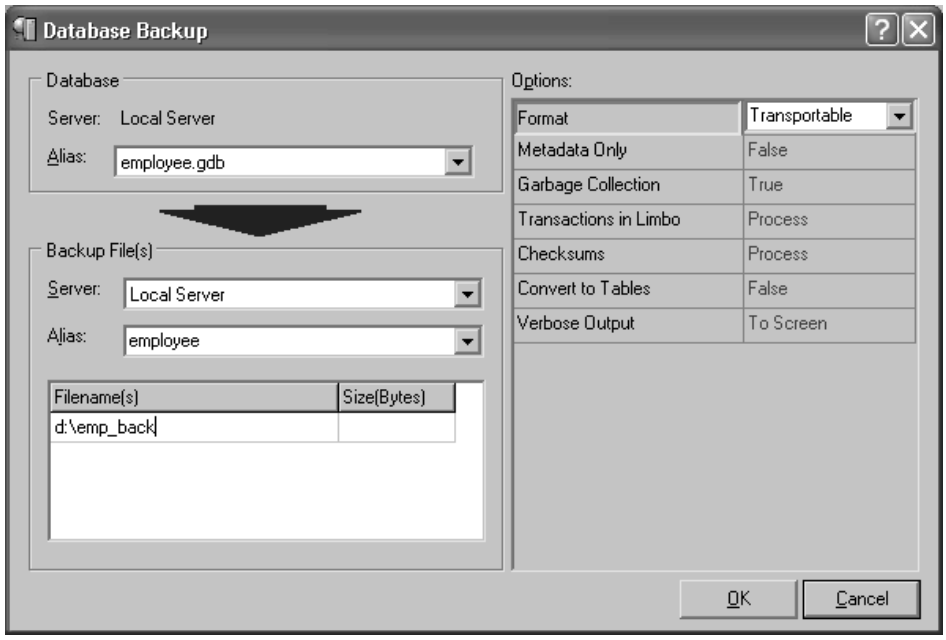


Hier besteht zwar die Möglichkeit, die (automatisch gestartete) Transaktion mit *Commit* zu bestätigen, den *Rollback*-Button sucht man jedoch vergebens.

11.2.3 Backups

Wenn Datenbanken im industriellen Umfeld produktiv arbeiten, dann hat der Datenbestand oft einen Wert, der den für Hard- und Software um ein Vielfaches übersteigt. Ein Totalverlust ihrer Daten dürfte die meisten Firmen ruinieren. Deshalb muss von der Datenbank regelmäßig eine Sicherungskopie gezogen werden.

Wählen Sie zu diesem Zweck aus dem Kontextmenü BACKUP/RESTORE|BACKUP.

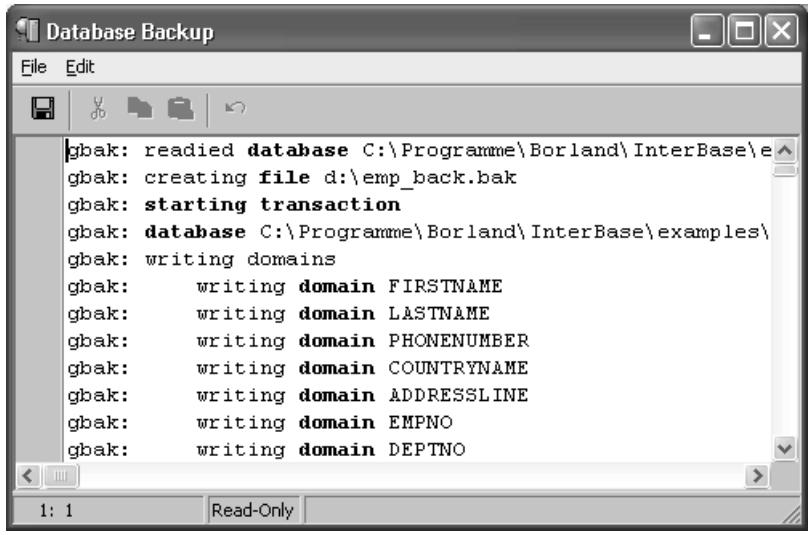


Zunächst einmal wählen Sie die Datenbank, die gesichert werden soll. Das Backup kann auf denselben Rechner oder auf einen anderen Rechner gespielt werden. Des Weiteren haben Sie die Möglichkeit, das Backup auf mehrere Dateien aufzuteilen – das kann dann Sinn machen, wenn sie Wechselmedien verwenden.

Wenn Sie das transportable Format verwenden, dann kann das Backup auch auf einem Server eingespielt werden, der unter einem anderen Betriebssystem läuft – dies ist auch die Methode, um die Daten von einem Server zu einem anderen zu übertragen.

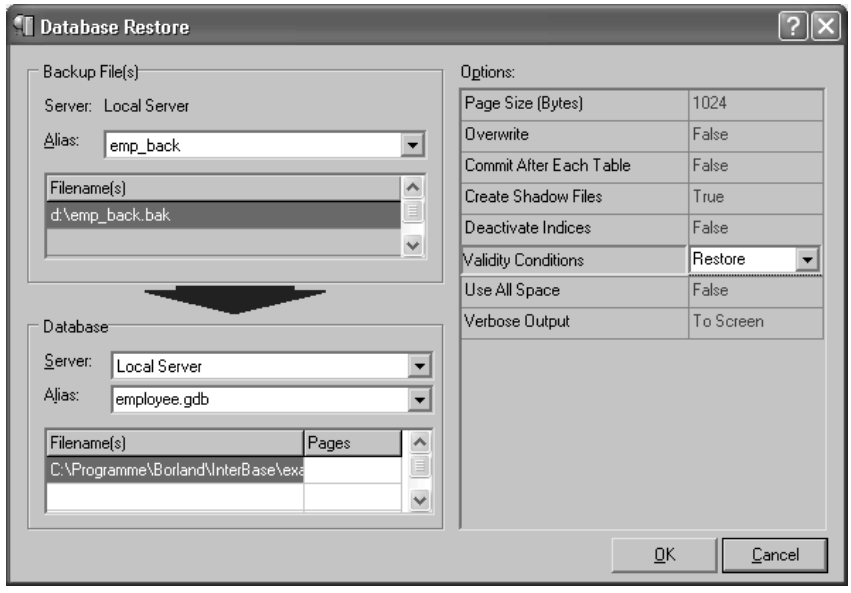
Weiterhin können Sie einige Details einstellen, beispielsweise ob eine Prüfsumme erstellt werden soll oder wie mit nicht abgeschlossenen Transaktionen zu verfahren ist.

Wenn *VerboseOutput* den Wert *To Screen* hat, dann erhalten Sie ein detailliertes Protokoll des Backups.



Restore

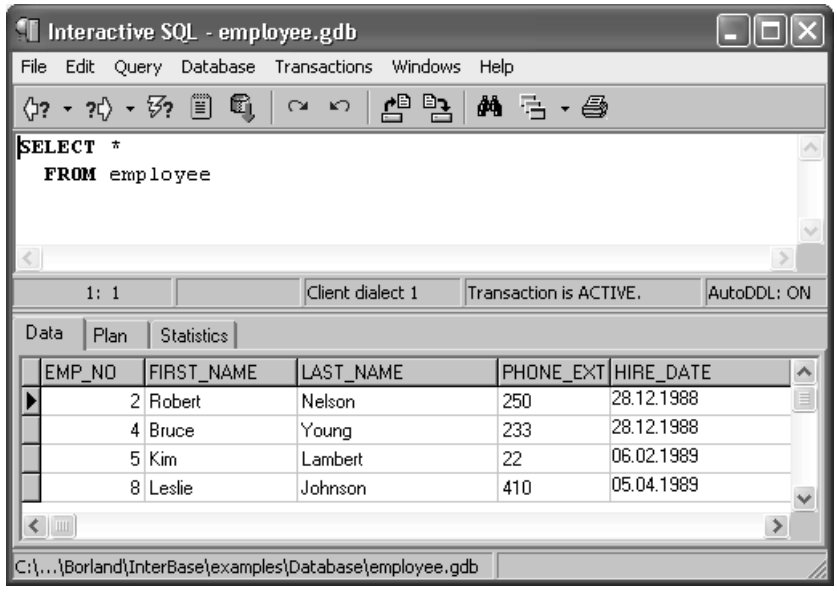
Zum Wiederherstellen verwenden Sie BACKUP/RESTORES|RESTORE aus dem Kontextmenü.



Hier haben Sie die Möglichkeit, die Datenbank auf mehrere Dateien aufzuteilen. Liegen diese Dateien auf unterschiedlichen Festplatten, dann können Sie damit die Performance verbessern. Ebenfalls können Sie hier die Seitengröße einstellen oder einen Shadow (eine dauernd gepflegte Kopie) einrichten.

11.3 Interactive SQL

Mit dem Programm *Interactive SQL* können Sie SQL-Anweisungen zum Server schicken. Handelt es sich um SELECT-Statements, dann wird die Ergebnismenge in einem DBGrid angezeigt:

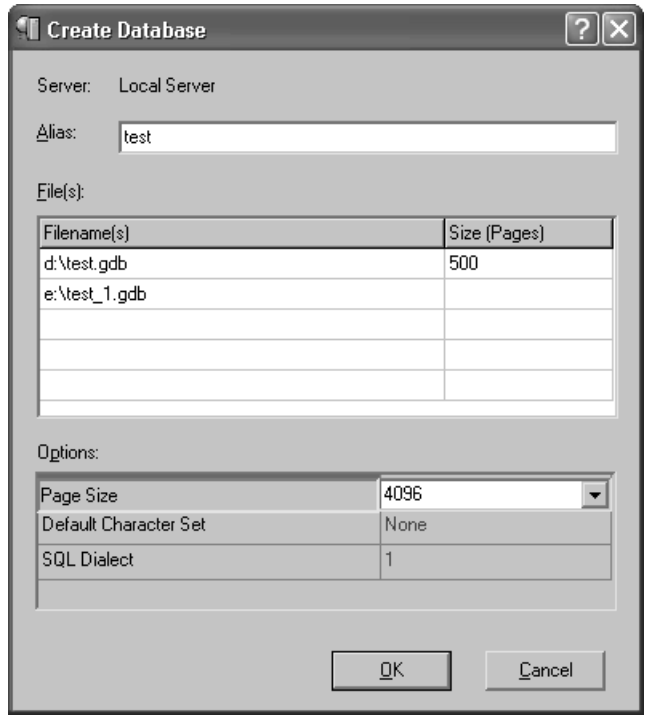


Interactive SQL wird von *IBConsole* aus für die aktuelle Tabelle aufgerufen.

Eine Datenbank erstellen

Bevor eine neue Datenbank erstellt werden kann, muss die Verbindung zur bestehenden beendet werden. Wählen Sie dazu DATABASE|DISCONNECT.

Anschließend wird dann DATABASE|CREATE DATABASE gewählt. Hier können Sie wählen, auf welche Dateien die Datenbank aufgeteilt werden soll und wie groß die Seitengröße ist.



Das Aufteilen einer Datenbank auf mehrere Dateien macht vor allem dann Sinn, wenn diese auf verschiedenen Festplatten liegen, weil dann die Datenbank in zwei Festplatten gleichzeitig schreiben kann (und natürlich auch daraus lesen).

Datenbank mit einem Script erstellen

Sie können eine Datenbank auch per SQL-Script erstellen. Das Skript können Sie mit QUERY|LOADSCRIPT laden. Die folgenden Zeilen zeigen den Anfang des SQL-Scripts zur Erstellung der To-Do-Datenbank:

```
SET SQL DIALECT 1;

CREATE DATABASE 'D:\TODO.GDB'
  USER 'SYSDBA'
  PASSWORD 'masterkey'
  PAGE_SIZE 4096;

/* Domain definitions */
CREATE DOMAIN D_DATEI AS BLOB SUB_TYPE 0 SEGMENT SIZE 80;
CREATE DOMAIN D_ID AS INTEGER NOT NULL;
```

11.4 Die Multi-Generationen-Architektur

Auf eine Besonderheit »hinter den Kulissen« von *InterBase* soll hier noch kurz eingegangen werden.

Transaktionen

Alle Datenbankaktionen finden im Rahmen so genannter *Transaktionen* statt. Eine solche Transaktion ist eine Zusammenfassung von mehreren Datenbankaktionen, die entweder alle gemeinsam bestätigt oder alle gemeinsam verworfen werden.

Gehen wir dies am Beispiel einer Banküberweisung durch: Zunächst wird die Transaktion mit START TRANSACTION begonnen. Dann wird vom Konto 1 der Betrag x abgebucht. Anschließend wird dem Konto 2 der Betrag x gutgeschrieben. Zuletzt wird die Transaktion mit COMMIT bestätigt.

Tritt nun ein Fehler auf, beispielsweise dass Konto 1 nicht die erforderliche Deckung aufweist, oder dass der Client während der Transaktion abstürzt, dann wird die Transaktion mit ROLLBACK in ihrer Gesamtheit rückgängig gemacht.

11.4.1 Transaktionen beim Lesen

Auch beim Lesen von Daten werden Transaktionen gestartet. Der Zweck hierbei ist eine konsistente Datensicht. Nehmen wir einmal an, ein Anwender lässt sich die Mitgliederdatei eines Vereins zuerst nach den Wohnorten und dann nach der Altersstruktur anzeigen, dann erwartet er, dass in beiden Fällen die gleiche Gesamtanzahl von Mitgliedern vorhanden ist. Nun ist aber in einer Client-Server-Umgebung damit zu rechnen, dass zwischen diesen beiden Abfragen ein anderer Anwender Änderungen am Datenbestand ausführt, so dass sich die Gesamtzahl von Mitgliedern ändert. Um hier Inkonsistenzen zu vermeiden, gibt es mehrere Möglichkeiten, die im Folgenden aufgeführt sind:

Replikationen

Für statistische Auswertungen möchte man immer auf denselben Datenbestand zugreifen. Dabei ist es meist zweitrangig, ob dieser ganz aktuell oder ein paar Stunden alt ist. Hier kann man dann eine Replikation erstellen, man kopiert also die komplette Datenbank in eine zweite, die dann nur der statistischen Auswertung dient. Weil hier nur Lesezugriffe stattfinden, braucht man sich um die Datenkonsistenz auch keine Sorgen zu machen.

Das Problem an den Replikationen ist ihre Erstellung. Um eine komplette Datenbank zu kopieren, braucht man meist eine Weile, und damit hier die Konsistenz gewahrt bleibt, bleibt wenig anderes übrig, als den Zugriff auf die komplette Datenbank zu sperren.

Die Erstellung einer Replikation findet deshalb sinnvollerweise automatisiert außerhalb der Arbeitszeit statt. Bei dieser Gelegenheit kann man auch gleich das Backup ziehen, weil hier die gleiche Problematik hinsichtlich der Konsistenz auftritt. Genau genommen ist die Replikation ein Backup, das man aber sicherheitshalber auf ein anderes Medium kopieren sollte.

Sperren von Datensätzen, Seiten und Tabellen

Nicht immer ist es möglich, mit Replikationen zu arbeiten. In manchen Fällen müssen es einfach die aktuellsten Daten sein oder es fehlt die Zeit zum Erstellen einer Replikation im Rund-um-die-Uhr-Betrieb. Hier kann man nun beginnen, die beteiligten Datensätze, Seiten oder Tabellen zu sperren.

Zeigt ein Client beispielsweise einen Datensatz an, kann ein anderer Client ihn nicht ändern. Umgekehrt kann ein Datensatz auch nicht angezeigt werden, solange eine Transaktion läuft, welche diesen Datensatz geändert hat. Die Folge dieser Vorgehensweise sind aufwendige Locking-Mechanismen. Damit die Geschwindigkeit der Datenbank nicht zu sehr darunter leidet, sperrt man dann meist auch ganze *Seiten*, das sind die Blöcke auf der Festplatte, in welche die Datenbank eingeteilt ist.

Während das Sperren von Datensätzen und/oder Seiten noch einigermaßen erträglich ist, wird das Sperren von ganzen Tabellen recht problematisch. Hier wäre es durchaus möglich, dass beispielsweise 20 Clients in der Buchhaltung nicht arbeiten können, weil gerade eine Bilanz erstellt und somit eine konsistente Datensicht benötigt wird.

11.4.2 Multi-Generationen-Architektur

Die eben genannten Nachteile von Sperrungen und Replikationen vermeidet *InterBase* mit der Multi-Generationen-Architektur. Dazu wird jede Transaktion und jede Datenbankänderung mit einem »Zeitstempel« versehen, also mit der Information, wann die Transaktion gestartet oder wann die Änderung durchgeführt worden ist.

Ändern von Datensätzen

Wir beginnen die erste Transaktion, welche den Zeitstempel 9.00 Uhr und den Isolationsgrad SNAPSHOT haben soll. (Der Zeitstempel wird natürlich mit Datum und auf die Millisekunde genau gespeichert, hier im Beispiel soll es aber etwas vereinfacht dargestellt werden.)

Um 9.05 Uhr beginnen wir eine zweite Transaktion. Die zweite Transaktion ändert nun einen Datensatz in der Tabelle *test* und wir schließen die Transaktion um 9.06 Uhr mit COMMIT ab. *InterBase* erkennt nun, dass im Moment noch eine

andere Transaktion läuft, und schreibt die Änderungen nicht direkt in die Datenbank. Stattdessen wird eine Änderungsmeldung hinzugefügt, welche die Daten der Änderung und den Zeitstempel von 9.06 Uhr speichert.

Nun führen wir im Kontext der ersten Transaktion eine Abfrage auf die Tabelle *test* durch. *InterBase* erkennt nun, dass die abfragende Transaktion älter ist als die Änderung und gibt den Stand der Tabelle *test* von 9.00 Uhr zurück.

Um 9.10 Uhr starten wir eine dritte Transaktion, welche ebenfalls eine Abfrage auf *test* durchführt. *InterBase* erkennt nun anhand des Zeitstempels, dass die Transaktion jünger ist als die Änderung und gibt die Tabelle mit der Änderung zurück.

Wie Sie sehen, müssen hier weder eine Tabelle, noch ein Datensatz oder eine Seite gesperrt werden. Infolgedessen brauchen Sie erstens keine Replikationen für Ihre Abfragen und können zweitens ein Backup im laufenden Betrieb erstellen.

Konkurrierender Zugriff

Unsere erste Transaktion ändert nun den Datensatz 12345 in der Tabelle *test* und wird mit COMMIT um 9.15 Uhr abgeschlossen. Wieder werden nur die Art der Änderung und der Zeitstempel in die Tabelle geschrieben.

Nun ändert die dritte Transaktion ebenfalls Datensatz 12345 und versucht, um 9.17 Uhr die Transaktion abzuschließen. Hier erkennt nun *InterBase* einen konkurrierenden Zugriff und weist die Transaktion ab. An dieser Stelle muss nun der Anwender eine neue Transaktion starten. Erst dann kann er den Datensatz ändern (wobei er hoffentlich erst prüft, welche Änderungen gerade erfolgt sind). Auch hier ist ein Sperren von Tabellen nicht erforderlich.

12 InterBase SQL

In Kapitel 6.1 haben wir uns LocalSQL angesehen, den Sprachumfang, den die BDE implementiert. InterBase implementiert aber deutlich mehr Sprachelemente, insbesondere, was DOMAINS, Zugriffsberechtigungen sowie TRIGGER und STORED PORCEDURES betrifft. Diese Elemente wollen wir uns in diesem Kapitel ansehen.

12.1 SELECT

Beim SELECT-Statement kann InterBase nicht wesentlich mehr als LocalSQL, aber da das SELECT-Statement wohl am häufigsten verwendet wird, wollen wir unsere Betrachtung damit beginnen.

12.1.1 WHERE

Für die Filterungen mit WHERE stehen ein paar weitere Operatoren zur Verfügung.

Der BETWEEN-Operator

Wenn wir alle Kunden mit einer Kundennummer zwischen 500 und 600 ermitteln wollen, dann können wir mit AND eine Größer-gleich- und eine Kleiner-gleich-Bedingung verknüpfen. Einfacher und übersichtlicher geht es mit dem Operator BETWEEN.

```
SELECT *
  FROM t_kunde
  WHERE (nummer BETWEEN 500 AND 600)
```

Beim Operator BETWEEN zählen die beiden Grenzen immer noch mit zum Bereich.

Der STARTING WITH-Operator

Bei vielen WHERE-Klauseln wird nach dem Anfang eines Strings gesucht. Dies ist mit Hilfe des LIKE-Operators problemlos möglich. Allerdings fordert die Flexibilität des LIKE-Operators ihren Preis in Form längerer Ausführungszeiten. Schneller arbeitet der Operator STARTING WITH.

Wie groß die Unterschiede sind, hängt vom Datenbestand und vom DMS ab (beispielsweise 1,7 s statt 2,6 s):

```
SELECT *
  FROM t_kunde
  WHERE nachname STARTING WITH "Ma"
```

Diese Anweisung ermittelt alle Kunden, deren Nachnamen mit *Ma* beginnen. Beachten Sie, dass auch STARTING WITH zwischen Groß- und Kleinschreibung unterscheidet.

Der CONTAINING-Operator

Mit Hilfe des CONTAINING-Operators kann ein String auf das beliebige Vorkommen eines anderen Strings untersucht werden.

```
SELECT *
  FROM t_kunde
  WHERE (nachname CONTAINING "ä")
    AND (ort = "Berlin")
```

Diese Anweisung ermittelt alle Berliner Kunden, deren Nachname den Buchstaben *ä* enthält. Die Länge des zu suchenden Strings ist (nahezu) beliebig.

12.1.2 ORDER BY

Auch bei der ORDER BY-Klausel haben wir ein paar Möglichkeiten mehr:

Sortieren nach Spalten

Es besteht die Möglichkeit, die Spalte, nach der sortiert werden soll, mit ihrem Index anzugeben. Nehmen wir an, wir machen eine Statistik, wie viele Kunden aus den einzelnen Orten kommen:

```
SELECT COUNT(*), ort
  FROM t_kunde
  GROUP BY ort
```

Nun wollen wir die Sache nach der Anzahl sortieren, und zwar absteigend. Die folgende Anweisung funktioniert nicht, und zwar unabhängig davon, ob *Count* – weil es ein SQL-Schlüsselwort ist – in Anführungszeichen gesetzt wird oder nicht.

```
SELECT COUNT(*), ort
  FROM t_kunde
  GROUP BY ort
  ORDER BY "Count"
```

Dagegen funktioniert die Sache problemlos mit der folgenden Anweisung:

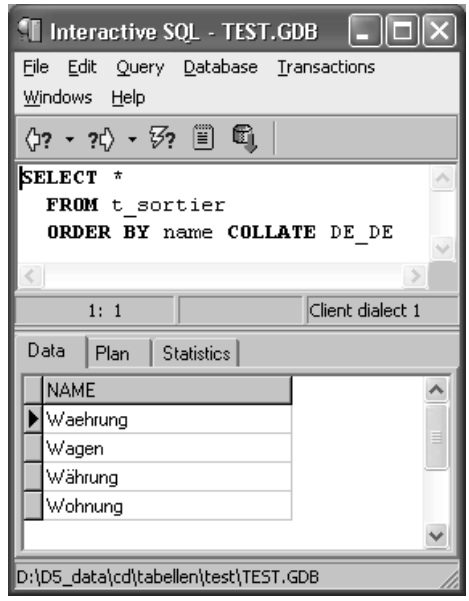
```
SELECT COUNT(*), ort
  FROM t_kunde
  GROUP BY ort
  ORDER BY 1 DESC
```

ORDER BY 1 bedeutet dabei, dass nach der ersten Spalte sortiert wird.

Angabe der Sortierreihenfolge

Immer wieder interessant ist die Frage, wie bei einer alphabetischen Sortierung genau sortiert wird, beispielsweise, wie die deutschen Umlaute einsortiert werden. Erstellt man eine Datenbank für den deutschen Sprachraum, so ist beispielsweise der Zeichensatz *ISO8859_1* nicht verkehrt, der sich mit der Sortierreihenfolge DE_DE kombinieren lässt.

Die Sortierreihenfolge lässt sich dabei nicht nur bei der DOMAIN-Definition spezifizieren, sondern auch bei der ORDER-Klausel eines SELECT-Statements:

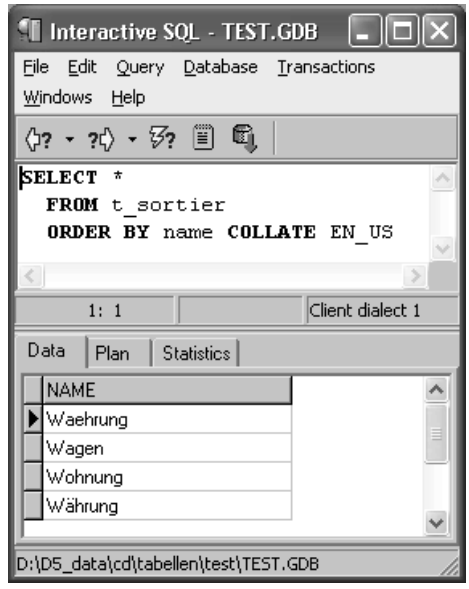


Der Zeichensatz *ISO8859_1* unterstützt die folgenden Sortierreihenfolgen:

- DA_DA (Dänisch)
- DE_DE (Deutsch)
- DU_NL (Holländisch)

- EN_UK (englisches Englisch)
- EN_US (amerikanisches Englisch)
- ES_ES (spanisch)
- FI_FI (Finnisch)
- FR_CA (kanadisches Französisch)
- FR_FR (französiches Französisch)
- IS_IS (Isländisch)
- IT_IT (Italienisch)
- NO_NO (Norwegisch)
- PT_PT (Portugiesisch)
- SV_SV (Schwedisch)

Würde man dieselbe Abfrage in amerikanischem Englisch sortieren, wären die Umlaute nicht korrekt eingeordnet.



Kommen Sie nun nicht auf die Idee, nach dem Motto *das können wir ja bei der ORDER-Klausel einstellen* auf Zeichensatz- und Sortierreihenfolgendefinitionen bei der Erstellung der Datenbank zu verzichten: Der Standardsprachtreiber unterstützt überhaupt keine anderen Sortierreihenfolgen.

12.2 DOMAINS

DOMAINS sind Wertebereiche, auf denen die einzelnen Tabellenspalten beruhen; sie entsprechen den Typen bei Pascal. 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 Standardtyp 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 Integertypen 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, und so wird dann auch nicht so genau nach Referenzen auf diese Spalte gesucht. Momentan geht noch alles gut, muss aber eine Referenz auf einen überlangen Nachnamen gebildet werden, 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).

Automatische Erstellung von DOMAINS

Wenn Sie sich im SQL-Explorer die Liste der definierten DOMAINS ansehen, dann werden Sie dort vermutlich viele Einträge finden, die Sie nie angelegt haben. *InterBase* erstellt nämlich für jede Tabellenspalte, die nicht explizit auf einer DOMAIN beruht, jeweils eine eigene DOMAIN. Diese beginnen mit *RDB$* und werden mit durchlaufenden Nummern versehen.

Mit Hilfe dieser DOMAINS kann man nachträglich Gültigkeitsprüfungen einfügen, selbst dann, wenn sich in der darauf aufbauenden Tabelle schon Daten befinden, und selbst dann, wenn es sich um eine automatisch erstellte DOMAIN handelt:

```
ALTER DOMAIN RDB$46
  ADD CHECK (VALUE < 100)
```

Und so lässt sich die Gültigkeitsprüfung wieder ausschalten:

```
ALTER DOMAIN RDB$46
  DROP CONSTRAINT
```

12.2.1 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:

- VARCHAR (alternativ CHAR VARYING oder CHARACTER VARYING).

  ```
  CREATE DOMAIN d_name AS VARCHAR(30);
  ```

Spalten dieser DOMAIN beinhalten Strings, die maximal die bei der Definition angegebene Länge haben können.

- CHAR (alternativ CHARACTER)

```
CREATE DOMAIN d_name_l AS CHAR(30) NOT NULL;
```

Hier haben die Spalten exakt die angegebene Länge. Ist der darin gespeicherte String kürzer, wird der Rest mit Leerzeichen aufgefüllt.

Für die Programmierung in Delphi verwenden wir fast immer VARCHAR.

Beachten Sie bitte auch, dass Sie es *InterBase* explizit mitteilen müssen, wenn Sie eine deutsche Sortierreihenfolge haben möchten (damit die Umlaute richtig einsortiert werden):

```
CREATE DOMAIN namen AS VARCHAR(20)
  CHARACTER SET ISO8859_1 COLLATE DE_DE;
```

Zahlen

Bei InterBase sind folgende Zahlentypen verfügbar:

- SMALLINT: Ein 16-Bit-Integertyp mit einem Wertebereich von -32768 bis 32767 (alle Angaben laut *InterBase Data Definition Guide*)
- INTEGER: Ein 32-Bit-Integertyp mit einem Wertebereich von -2147483648 bis 2147483648.
- FLOAT: Ein Gleitkommatyp mit 32 Bit, ähnlich dem Delphi-Typ *single*, 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 etwa dem Delphi-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 in Delphi stets zu einem *TIntegerField* werden und dann ohne Nachkommastellen angezeigt werden. Wenn Sie Delphi-Applikationen erstellen, dann sollten Sie also diese beiden DOMAINS meiden.

- AUTOINC: Gibt es bei InterBase nicht, bei Paradox-Tabellen können Sie damit selbstinkrementierende Felder (also Felder, die automatisch durchnummeriert werden) deklarieren. Für eine solche Funktion wird ein INTEGER-Typ verwendet und dafür ein GENERATOR definiert. Näheres dazu erfahren Sie in Kapitel 17.2.4.

Datum und Zeit

Der Datentyp DATE speichert – dem Delphi-Typ *TDateTime* entsprechend – immer Datum und Zeit. Um Datum und Zeit einzeln zu erhalten, werden folgende Anweisungen benötigt:

```
procedure TForm1.Button1Click(Sender: TObject)
var
  dt: TDateTime;
begin
  dt := Table1.FieldByName('Hire_Date').AsDateTime;
  Edit1.Text := TimeToStr(dt);
  Edit2.Text := DateToStr(dt);
end;
```

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 Tabelle, sondern getrennt davon gespeichert werden.

Bei Delphi 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, bedenken Sie bitte, dass das Feld ein *TBlobField* und nicht ein *TMemoField* oder ein *TGraphikField* ist.

Sollen Tools wie beispielsweise *IBConsole* den Inhalt von BLOBS darstellen können, dann müssen diese SUB_TYPE TEXT definiert werden.

```
CREATE DOMAIN d_memo AS BLOB SUB_TYPE TEXT;
```

Wenn Sie BLOB-Spalten als SUB_TYPE TEXT definiert haben, erlaubt InterBase die Suche darin mit denselben Operatoren, die Sie auch auf String-Spalten anwenden können:

```
SELECT *
  FROM t_benutzer
  WHERE memo LIKE '%Ebner%'
```

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 im Normalfall 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.)

12.2.2 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".

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 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 wäre dann die Möglichkeit notwendig, 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 Byte, bei angenommenen 500 Datensätzen sind dies rund 7,5 kbyte. Ein Referenzfeld kommt mit Byte 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).

12.2.3 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. In diesem Fall kann der Befehl NOT NULL gegeben werden:

```
CREATE DOMAIN d_test AS VARCHAR(15) NOT NULL
```

12.2.4 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 aufwändig wäre, während des Betriebs der Datenbank diese Menge beispielsweise um den Eintrag *Reinigungskraft* zu ergänzen.

12.2.5 DOMAINS ändern

Auf der DOMAIN-Definition beruhen die Tabellenspalten. Daher muss man nicht erwarten, dass sich der zugrunde liegende Datentyp ändern lässt – hier bliebe nur die Möglichkeit, 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 verwenden Sie die entsprechenden Möglichkeiten von *IBConsole*, die in Kapitel 11.2.2 beschrieben wurden. Um die Prüfung zu löschen, wählen Sie DROP CONSTRAINT, den Vorgabewert löschen Sie mit DROP DEFAULT.

12.2.6 DOMAINS löschen

Sehr einfach ist die Anweisung zum Löschen einer DOMAIN (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
```

12.2.7 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 eine fortlaufende Nummer vergeben werden. Um einen Generator zu erstellen, muss einfach nur der Befehl CREATE GENERATOR gegeben werden, dem der Name 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 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 des 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, 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).

Datenaustausch zwischen Transaktionen

Datenbankaktionen auf SQL-Servern finden im Kontext so genannter Transaktionen statt. Diese Transaktionen sind voneinander isoliert, so dass eine Transaktion nichts davon merkt, ob andere Transaktionen zur selben Zeit auf den Server zugreifen. Wir werden uns nachher noch intensiver mit Transaktionen beschäftigen.

Wenn jedoch Transaktionen voneinander isoliert sind, ist auch ein Datenaustausch zwischen ihnen – wenn er denn ausnahmsweise einmal nötig sein sollte – nicht möglich. Hier kann man dann Generatoren zweckentfremden, weil diese nicht im Kontext von Transaktionen stattfinden (das würde auch ihrem Zweck zuwiderlaufen).

Man kann einen Generator auch als ganz simple 32-Bit-Integer-Variable betrachten. Mit SET GENERATOR wird diese Variable gesetzt, mit GEN_ID wird der Inhalt ausgelesen.

12.3 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.

12.3.1 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))
```

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 als »Parameter« hinzugefügt. Eine Spaltendefinition besteht stets aus einem Spaltennamen sowie einem Typ (oder einer DOMAIN), auf dem (der) sie beruht. Des Weiteren sind alle Möglichkeiten gegeben, die wir schon beim Thema DOMAIN kennen gelernt 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 eine 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. Die Spalte(n) der beiden Tabellen sollte(n) streng genommen 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 aufwändig sind, sollte man ein vernünftiges Konzept erarbeiten, bevor man beginnt:

- Zuallererst wird die Datenbank richtig entworfen und normalisiert. Die entsprechenden Schritte wurden in Kapitel 2 beschrieben.
- Als Nächstes werden zweckmäßige Datentypen festgelegt und entsprechend die DOMAINS erstellt.
- Alle Tabellen werden ohne Fremdschlüssel erstellt. Diese werden nachher mit ALTER TABLE hinzugefügt; 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 || " EUR"))
```

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 9,- Euro angenommen. In der Tabelle *t_aushilfen* soll nun auch von einem Fehler ausgegangen werden, wenn der Stundenlohn 32,- Euro übersteigt.

Bitte beachten Sie, dass Ihnen mit dem Menübefehl 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, 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 2 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 das noch lange nicht, dass dies sinnvoll ist.

Um einen Primärschlüssel zu definieren, wird die Anweisung PRIMARY KEY verwendet. Die entsprechende(n) 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, die 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 separaten 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 am Ende der Tabelle mit dem Befehl UNIQUE ein Sekundärschlüssel erstellt werden.

Wird in einer Adressendatenbank die Nummer zum Primärschlüssel erhoben, kann es leicht vorkommen, dass sich doppelte Datensätze in die Tabelle einschleichen. Um dies ein wenig einzudämmen, wurde hier ein Sekundärschlüssel über die Felder *nachname, strasse* und *plz* gebildet.

Der Verzicht auf den Vornamen kann zwei Gründe haben: Erstens 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 streng genommen nicht mehr in der dritten Normalform; dies sollte jedoch kein Grund dafür sein, die Tabelle in zwei Tabellen zu zerlegen.

InterBase erstellt für jeden Primär-, Zweit- und Fremdschlüssel automatisch einen Index. Dies beschleunigt die Selektion und Sortierung von Datensätzen. Sie können jedoch auch Indizes für andere Spalten erstellen. Es ist dabei nicht erforderlich, dass diese Spalten mit eindeutigen Werten belegt sind.

Gültigkeitsprüfungen mit CONSTRAINT benennen

Mit *IBConsole* lassen wir uns die Metadaten von *t_adressen* anzeigen:

```
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, 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 Tabelle 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. Deshalb erstellt man erst alle Tabellen ohne Referenzen und fügt sie dann später mit ALTER TABLE hinzu.
- Der Anwender kann keinen Datensatz aus der Tabelle löschen, auf den eine Referenz besteht, wenn dadurch Zuordnungen verloren gehen würden; genauso wenig dürfen Änderungen an der oder den Spalte(n), die den Primärschlüssel bildet(n), vorgenommen werden, wenn dadurch die referenzielle Integrität nicht mehr gewährleistet wäre.

  (Mit CASCADE könnte man auch einstellen, dass beim Löschen des Hauptdatensatzes auch alle referenzierenden Datensätze gelöscht werden, aber diese Vorgehensweise ist in der Praxis meist nicht zu empfehlen.)
- 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 Fremdschlüsselspalten 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 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 wird 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 auf 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.

12.3.2 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 Tabellenspalte umdefiniert. Nehmen wir einmal an, dass wir eine Tabelle mit Mitarbeiterdaten hätten, in der die Spalte Nachname 25 Zeichen breit ist:

```
CREATE TABLE t_mitarbeiter
  (nummer INTEGER NOT NULL,
  vorname VARCHAR(25),
  nachname VARCHAR(25),
  abteilung INTEGER,
  PRIMARY KEY (nummer),
  FOREIGN KEY (abteilung) REFERENCES t_abteilung (nummer))
```

Irgendwann tritt einer derjenigen Fälle auf, in denen ein Doppelname 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 (so lange 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 (so lange 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 verweigert.

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 eingegeben 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 verwendet man das Programm *IBConsole*.

```
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 10000 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)
```

Fremdschlüssel hinzufügen

Mit der folgenden Anweisung wird ein Fremdschlüssel erstellt:

```
ALTER TABLE t_benutzer_funktion
  ADD FOREIGN KEY (benutzer)
  REFERENCES t_benutzer (id);
```

12.3.3 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 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.

12.3.4 Indizes

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 sequenzielle 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.

```
CREATE INDEX testadr_tel1 ON testadr (tel1)

CREATE INDEX mitarbeiter_namen ON mitarbeiter (vornamen, nachnamen)

DROP INDEX testadr_1
```

Mit CREATE INDEX wird ein Index erstellt. Dieser kann sich auch über mehrere Spalten einer Tabelle erstrecken. Mit DROP INDEX wird der Index wieder gelöscht.

Dass ein Index eine Abfrage ganz erheblich beschleunigen kann, erfahren Sie in Kapitel 13.

12.4 VIEWS

Eine VIEW ist im Prinzip eine vordefinierte Abfrage (mit SELECT), auf die wie auf eine Tabelle zugegriffen 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 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 (inwieweit 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 durchführen, 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 (*IBConsole* steht ihm 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.
- InterBase kann den QueryPlan speichern und muss die Abfrage nicht erst interpretieren – das beschleunigt die Ausführung.

12.4.1 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
```

12.4.2 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, sich mit der zweiten Anweisung 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 zu verhindern, dass über die Durchwahlnummerntabelle plötzlich Mitarbeiterdaten verschwinden oder abgeändert werden (und sei es aus Unkenntnis oder aus Versehen), 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 Benutzer 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 Bedingung (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 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.

12.4.3 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
```

12.5 STORED PROCEDURES

Es gehört zur Philosopie von Client-Server-Systemen, dass Anfragen auf dem Server bearbeitet 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 Philosophie 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 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 eigenen Zahlen die richtigen sind.

  Wären die Kosten des Projektes von einer STORED PROCEDURE ermittelt 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 aufwändig, 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.
- Eine STORED PROCEDURE wird vom Server fertig interpretiert gespeichert – das beschleunigt die Ausführung.

12.5.1 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 beiseite 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 die direkt, sondern auch die indirekt unterstellten Abteilungen interessieren, und dies ist mit einer SELECT-Abfrage nun ein wenig aufwändig 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 die 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. 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.

12.5.2 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.

Hier haben Sie die Möglichkeit, Schleifen zu bilden oder Verzweigungen einzufügen, es lassen sich auch Variablen deklarieren.

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, 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 Eingangsparameter, die nach dem Prozedurnamen aufgelistet sind.
- Die Ausgangs- oder Rückgabeparameter, 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.

12.5.3 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 dargestellt, 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
```

12.5.4 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.

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.

12.6 TRIGGER

Ein TRIGGER ist im Prinzip das, was bei Delphi 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.

12.6.1 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 aufwändiger 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 Kontextvariablen 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
```

12.6.2 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 Kontextvariablen 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
```

12.6.3 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 mit einer Fehlermeldung 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).

Einen TRIGGER löschen

Wahrscheinlich wissen Sie schon aus Erfahrung, wie bei SQL Objekte gelöscht werden:

```
DROP TRIGGER e_gehalt
```

12.7 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 darunter liegende 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
```

12.8 Sonstiges

Der *InterBase Server* ist ein sehr umfangreiches Thema, welches in diesen beiden Kapiteln nicht vollständig abgehandelt werden konnte.

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:

- Das Thema *Transaktionen* (SET TRANSACTION, COMMIT, ROLLBACK) wurde bereits in Kapitel 4.5.1 bei der Komponente *TDatabase* erläutert.
- Um den Datenbestand gegen Hardware-Fehler 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. Das Programmieren von solchen UDFS (User Defined Functions) werden wir in Kapitel 13 besprechen.
- Für die einfachere Handhabung der Benutzerrechte können ROLES erstellt werden. Diesen können – wie einem Benutzer – Rechte zugewiesen werden. Die Benutzer erhalten dann ihrerseits Rechte auf diese Rollen.

13 Client-Server mit der BDE

Wir wollen uns in diesem Kapitel mit der Frage beschäftigen, was zu beachten ist, wenn mit der BDE (und dann natürlich auch SQL Links) Client-Server-Datenbankanwendungen erstellt werden.

SQL Links erlaubt den Zugriff auf recht viele Datenbanksysteme. Unsere Beispiele beziehen sich hier auf InterBase, vieles lässt sich jedoch auf andere Datenbanksysteme übertragen.

13.1 TTable und TQuery

Die Frage, ob *TTable* oder *TQuery* verwendet werden soll, stellt sich in vielen Fällen überhaupt nicht: Große Datenbanken sind in der Regel stark normalisiert, um die Daten wieder zusammenzuführen, benötigt man JOINS, die durchaus auch mal ein paar Dutzend Tabellen umfassen können. Mit *TTable* kommen Sie hier allenfalls dann weiter, wenn Sie eine VIEW verwenden.

Es gibt aber auch Fälle, bei denen beide Komponenten eingesetzt werden können. Wie wollen uns nun ansehen, wie unterschiedlich sich diese beiden Komponenten verhalten.

13.1.1 Zugriff auf eine Tabelle

Beginnen wir mit dem einfachsten Fall: mit dem Zugriff auf eine einzelne Tabelle. Die hier gemessenen Werte beziehen sich auf die Tabelle *employee* aus der InterBase-Beispiel-Datenbank *employee.gdb*. Da die Zahlen ohnehin von Rechner zu Rechner stark differieren, spielt dies jedoch keine größere Rolle.

Uns interessiert nun die Zeit, die benötigt wird, um die Datenmenge zu öffnen:

- Beim Zugriff über *TTable* dauert der Vorgang 24 ms.
- Wenn wir stattdessen *TQuery* und die Anweisung *SELECT * FROM employee* verwendet, dauert das Öffnen nur 7 ms.
- *TQuery* liefert normalerweise keine aktualisierbare Datenmenge – es sei denn, wir stellen *RequestLive* auf *true*. Dann würde das Öffnen jedoch 32 ms dauern.

Wir wollen nun diesem Unterschied nachgehen und spionieren deshalb ein wenig mit dem SQL-Monitor. (Die Messungen wurden natürlich ohne dieses Tool gemacht.)

TTable

Zur besseren Lesbarkeit habe ich die Statements manuell formatiert. Und da Sie vermutlich auch keine Lust haben, seitenweise das Protokoll von Datenabrufen zu lesen, habe ich auch an dieser Stelle etwas gekürzt – Sie merken es an den Zeilennummern. Des Weiteren wurden alle Uhrzeiten entfernt.

```
1    SQL Prepare: INTRBASE -
     select
          RDB$OWNER_NAME,
          RDB$RELATION_NAME,
          RDB$SYSTEM_FLAG,
          RDB$VIEW_BLR,
          RDB$RELATION_ID
       from RDB$RELATIONS
       where RDB$RELATION_NAME = 'EMPLOYEE'
2    SQL Vendor: INTRBASE - isc_dsql_allocate_statement
3    SQL Vendor: INTRBASE - isc_start_transaction
4    SQL Vendor: INTRBASE - isc_dsql_prepare
5    SQL Execute: INTRBASE - select ...
6    SQL Vendor: INTRBASE - isc_dsql_execute
7    SQL Stmt: INTRBASE - Fetch
8    SQL Vendor: INTRBASE - isc_dsql_fetch
9    SQL Stmt: INTRBASE - Fetch
10   SQL Vendor: INTRBASE - isc_dsql_fetch
11   SQL Stmt: INTRBASE - EOF
12   SQL Transact: INTRBASE - XACT Commit
13   SQL Vendor: INTRBASE - isc_commit_transaction
14   SQL Stmt: INTRBASE - Close
15   SQL Vendor: INTRBASE - isc_dsql_free_statement
```

Zunächst werden einige Informationen aus der Tabelle *RDB$RELATIONS* ausgelesen. Dafür wird zunächst Speicherplatz alloziert, eine Transaktion gestartet, die Anweisung vorbereitet (also interpretiert) und dann ausgeführt.

Danach wird der erste Datensatz ausgelesen. Da diese Anweisung nur einen Datensatz liefert, kommt beim zweiten Aufruf von *Fetch* die EOF-Meldung. Danach wird die Transaktion abgeschlossen und der Speicher wieder freigegeben.

```
16   SQL Prepare: INTRBASE -
     select
          R.RDB$FIELD_NAME,
          F.RDB$FIELD_TYPE,
          F.RDB$FIELD_SUB_TYPE,
```

```
        F.RDB$DIMENSIONS,
        F.RDB$FIELD_LENGTH,
        F.RDB$FIELD_SCALE,
        F.RDB$VALIDATION_BLR,
        F.RDB$COMPUTED_BLR,
        R.RDB$DEFAULT_VALUE,
        F.RDB$DEFAULT_VALUE,
        R.RDB$NULL_FLAG
      from RDB$RELATION_FIELDS R, RDB$FIELDS F
      where R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
        and R.RDB$RELATION_NAME = 'EMPLOYEE'
      order by R.RDB$FIELD_POSITION ASC
17  SQL Vendor: INTRBASE - isc_dsql_allocate_statement
18  SQL Vendor: INTRBASE - isc_start_transaction
19  SQL Vendor: INTRBASE - isc_dsql_prepare
20  SQL Execute: INTRBASE - select ...
21  SQL Vendor: INTRBASE - isc_dsql_execute
22  SQL Stmt: INTRBASE - Fetch
23  SQL Vendor: INTRBASE - isc_dsql_fetch
    ...
32  SQL Stmt: INTRBASE - Fetch
33  SQL Vendor: INTRBASE - isc_dsql_fetch
34  SQL Blob IO: INTRBASE - Get Len (SCHEMA)
35  SQL Vendor: INTRBASE - isc_dsql_allocate_statement
36  SQL Vendor: INTRBASE - isc_open_blob
37  SQL Vendor: INTRBASE - isc_blob_info
38  SQL Vendor: INTRBASE - isc_close_blob
39  SQL Vendor: INTRBASE - isc_dsql_free_statement
    ...
68  SQL Stmt: INTRBASE - Fetch
69  SQL Vendor: INTRBASE - isc_dsql_fetch
70  SQL Stmt: INTRBASE - EOF
71  SQL Transact: INTRBASE - XACT Commit
72  SQL Vendor: INTRBASE - isc_commit_transaction
73  SQL Stmt: INTRBASE - Close
74  SQL Vendor: INTRBASE - isc_dsql_free_statement
```

Welche Schritte auszuführen sind, um eine Tabelle auszulesen, haben wir vorhin schon besprochen. Mit dem JOIN von *RDB$RELATION_FIELDS* und *RDB$FIELDS* werden hier Informationen über die beteiligten Felder besorgt.

Bei der Gelegenheit sehen Sie auch, welche Anweisungen nötig sind, um einen BLOB auszulesen.

```
75  SQL Prepare: INTRBASE -
    select
        I.RDB$INDEX_NAME,
        I.RDB$UNIQUE_FLAG,
        I.RDB$INDEX_TYPE,
        F.RDB$FIELD_NAME
      from RDB$INDICES I, RDB$INDEX_SEGMENTS F
      where I.RDB$RELATION_NAME = 'EMPLOYEE'
        and  I.RDB$INDEX_NAME = F.RDB$INDEX_NAME
      order by I.RDB$INDEX_ID, F.RDB$FIELD_POSITION ASC
```

Als Nächstes interessiert sich *TTable* für die vorhandenen Indizes und die daran beteiligten Spalten. Alle weiteren Anweisungen dazu habe ich mir nun erspart, achten Sie einfach auf die Zeilennummer.

```
102 SQL Prepare: INTRBASE -
    select
        R.RDB$FIELD_NAME,
        F.RDB$VALIDATION_BLR,
        F.RDB$COMPUTED_BLR,
        R.RDB$DEFAULT_VALUE,
        F.RDB$DEFAULT_VALUE,
        R.RDB$NULL_FLAG
      from RDB$RELATION_FIELDS R, RDB$FIELDS F
      where R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
        and R.RDB$RELATION_NAME = 'EMPLOYEE'
      order by R.RDB$FIELD_POSITION ASC
```

Wenn Sie genau aufgepasst haben, dann werden Sie sich erinnern, dass dieser JOIN mit denselben WHERE- und ORDER-Klauseln bereits einmal abgefragt wurde. Vielleicht kein Optimierungs-Highlight.

```
137 SQL Prepare: INTRBASE -
    SELECT
        EMP_NO ,
        FIRST_NAME ,
        LAST_NAME ,
        PHONE_EXT ,
        HIRE_DATE ,
        DEPT_NO ,
        JOB_CODE ,
        JOB_GRADE ,
        JOB_COUNTRY ,
```

```
        SALARY ,
        FULL_NAME
      FROM EMPLOYEE
      ORDER BY  EMP_NO ASC
138 SQL Vendor: INTRBASE - isc_dsql_allocate_statement
139 SQL Vendor: INTRBASE - isc_start_transaction
140 SQL Vendor: INTRBASE - isc_dsql_prepare
141 SQL Execute: INTRBASE - SELECT ...
142 SQL Vendor: INTRBASE - isc_dsql_execute
143 SQL Stmt: INTRBASE - Fetch
144 SQL Vendor: INTRBASE - isc_dsql_fetch
145 SQL Data Out: INTRBASE - Column = 1, Name = EMP_NO,
      Type = fldINT16, Precision = 1, Scale = 0, Data = 2
      ...
155 SQL Data Out: INTRBASE - Column = 11, Name = FULL_NAME,
      Type = fldZSTRING, Precision = 37, Scale = 0,
      Data = Nelson, Robert
```

Zuletzt kommen wir zu den Daten, die wir eigentlich haben wollen. Beachten Sie, in welcher Ausführlichkeit hier die ausgegebenen Daten mitprotokolliert werden.

TQuery

Im Vergleich dazu die Aufrufe, wenn *TQuery* verwendet wird:

```
1   SQL Prepare: INTRBASE -
    SELECT *
      FROM employee
2   SQL Vendor: INTRBASE - isc_dsql_allocate_statement
3   SQL Vendor: INTRBASE - isc_start_transaction
4   SQL Vendor: INTRBASE - isc_dsql_prepare
5   SQL Vendor: INTRBASE - isc_dsql_sql_info
6   SQL Vendor: INTRBASE - isc_vax_integer
7   SQL Transact: INTRBASE - XACT (UNKNOWN)
8   SQL Vendor: INTRBASE - isc_commit_retaining
9   SQL Execute: INTRBASE - SELECT *  FROM employee
10  SQL Vendor: INTRBASE - isc_dsql_execute
11  SQL Stmt: INTRBASE - Fetch
12  SQL Vendor: INTRBASE - isc_dsql_fetch
13  SQL Data Out: INTRBASE - Column = 1, Name = EMP_NO,
      Type = fldINT16, Precision = 1, Scale = 0, Data = 2
    ...
```

```
23  SQL Data Out: INTRBASE - Column = 11, Name = FULL_NAME,
      Type = fldZSTRING, Precision = 37, Scale = 0,
      Data = Nelson, Robert
```

Im Gegensatz zu *TTable* ist *TQuery* deutlich weniger »neugierig«: Es interessieren die Daten der Tabelle, nicht, wie diese aufgebaut ist. Dass so die Datenmenge schneller geöffnet ist, dürfte wohl verständlich sein.

RequestLive = true

Nun verwenden wir abermals *TQuery*, setzen *RequestLive* jedoch auf den Wert *true*.

```
1   SQL Prepare: INTRBASE -
    select
        RDB$OWNER_NAME,
        RDB$RELATION_NAME,
        RDB$SYSTEM_FLAG,
        RDB$VIEW_BLR,
        RDB$RELATION_ID
      from RDB$RELATIONS
      where RDB$RELATION_NAME = 'EMPLOYEE'
    ...
16  SQL Prepare: INTRBASE -
    select
        R.RDB$FIELD_NAME,
        F.RDB$FIELD_TYPE,
        F.RDB$FIELD_SUB_TYPE,
        F.RDB$DIMENSIONS,
        F.RDB$FIELD_LENGTH,
        F.RDB$FIELD_SCALE,
        F.RDB$VALIDATION_BLR,
        F.RDB$COMPUTED_BLR,
        R.RDB$DEFAULT_VALUE,
        F.RDB$DEFAULT_VALUE,
        R.RDB$NULL_FLAG
      from RDB$RELATION_FIELDS R, RDB$FIELDS F
      where R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
        and R.RDB$RELATION_NAME = 'EMPLOYEE'
      order by R.RDB$FIELD_POSITION ASC
    ...
```

```
75  SQL Prepare: INTRBASE -
    select
        I.RDB$INDEX_NAME,
        I.RDB$UNIQUE_FLAG,
        I.RDB$INDEX_TYPE,
        F.RDB$FIELD_NAME
      from RDB$INDICES I, RDB$INDEX_SEGMENTS F
      where I.RDB$RELATION_NAME = 'EMPLOYEE'
        and  I.RDB$INDEX_NAME = F.RDB$INDEX_NAME
      order by I.RDB$INDEX_ID, F.RDB$FIELD_POSITION ASC
    ...
102 SQL Prepare: INTRBASE -
    select
        R.RDB$FIELD_NAME,
        F.RDB$VALIDATION_BLR,
        F.RDB$COMPUTED_BLR,
        R.RDB$DEFAULT_VALUE,
        F.RDB$DEFAULT_VALUE,
        R.RDB$NULL_FLAG
      from RDB$RELATION_FIELDS R, RDB$FIELDS F
      where R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
        and R.RDB$RELATION_NAME = 'EMPLOYEE'
      order by R.RDB$FIELD_POSITION ASC
    ...
137 SQL Prepare: INTRBASE -
    SELECT
        EMP_NO,
        FIRST_NAME,
        LAST_NAME,
        PHONE_EXT,
        HIRE_DATE,
        DEPT_NO,
        JOB_CODE,
        JOB_GRADE,
        JOB_COUNTRY,
        SALARY,
        FULL_NAME
      FROM employee
138 SQL Vendor: INTRBASE - isc_dsql_allocate_statement
139 SQL Vendor: INTRBASE - isc_start_transaction
140 SQL Vendor: INTRBASE - isc_dsql_prepare
```

```
141 SQL Vendor: INTRBASE - isc_dsql_sql_info
142 SQL Vendor: INTRBASE - isc_vax_integer
143 SQL Transact: INTRBASE - XACT (UNKNOWN)
144 SQL Vendor: INTRBASE - isc_commit_retaining
145 SQL Execute: INTRBASE - SELECT ...
146 SQL Vendor: INTRBASE - isc_dsql_execute
147 SQL Stmt: INTRBASE - Fetch
148 SQL Vendor: INTRBASE - isc_dsql_fetch
149 SQL Data Out: INTRBASE - Column = 1, Name = EMP_NO,
      Type = fldINT16, Precision = 1, Scale = 0, Data = 2
159 SQL Data Out: INTRBASE - Column = 11, Name = FULL_NAME,
      Type = fldZSTRING, Precision = 37, Scale = 0,
      Data = Nelson, Robert
```

Auf den ersten Blick scheinen sich diese Aufrufe nicht von *TTable* zu unterscheiden. Wenn man genauer hinsieht, dann erkennt man die zusätzlichen Anweisungen 141 bis 144, mit denen – hier überflüssigerweise – Parameterinformationen geholt werden. Des Weiteren wird für die Abfrage der Daten keine ORDER-Klausel verwendet.

Fazit

Eine Datenmengenkomponente, die Änderungen in die Tabelle zurückschreiben soll, braucht Informationen über die Tabelle – sollen nur Daten gelesen werden, dann braucht man diese Daten auch nur entgegenzunehmen.

Es wäre jedoch ein Trugschluss zu glauben, dass *TTable* weniger »neugierig« wäre, würde man *ReadOnly* auf *true* setzen. Im Gegensatz zu den *dbExpress*-Komponenten hilft hier auch das Anlegen von persistenten *TField*-Instanzen nichts.

Daher gilt: Alle Abfragen, die häufig ausgeführt werden oder bei denen es auf hohe Performance ankommt, werden mit *TQuery* realisiert und *RequestLive* bleibt dabei auf *false*. Etwas anderes sind Tabellen, die nur gelegentlich aktualisiert werden müssen – bei unserem Beispielprojekt, der To-Do-Liste, beispielsweise die Benutzerverwaltung. Hier steht der Aufwand, alle Änderungen per SQL-Statement durchzuführen, oft in keinem vertretbaren Verhältnis zum Nutzen.

13.2 Transaktionen

Alle Zugriffe auf einen Datenbank-Server erfolgen im Kontext einer Transaktion. (Es gibt nur wenige Client-Server-Datenbanksysteme, die keine Transaktionen unterstützen, beispielsweise MySQL.) Mit diesen Transaktionen garantiert man einen stabilen Mehrbenutzer-Betrieb.

13.2.1 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 2 344 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 Delphi-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 mit der Professional-Version von Delphi arbeiten, dann hat ihr InterBase-Server lediglich zwei Benutzer; diese wären dadurch belegt, das Delphi selbst und die gestartete Anwendung eine Datenbankverbindung halten. Umgehen können Sie dieses Problem, wenn Sie erst zur Laufzeit die *TDatabase*-Eigenschaft *Connected* auf *true* setzen.)

Lost Updates

Von *Lost Updates* (verlorenen Änderungen) spricht man, wenn eine geänderte Reihe von einer parallel verlaufenden Transaktion ebenfalls geändert werden soll. 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 Paradox-Datenbanken durchgeführt werden. Hier muss dann zwingend der Abschottungsgrad *DirtyRead* verwendet werden.

Non-reproducible 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 nicht reproduzierbaren Lesevorgänge.
- Mit SNAPSHOT (in Delphi *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 dauern. Würde beispielsweise die Transaktion beim Starten des Programms

gestartet und deren Beendigung mit COMMIT bestätigt, dann arbeitet der Anwender am Ende eines Arbeitstages schon mit ziemlich veralteten 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 wurde.

Informationsaustausch zwischen Transaktionen

Transaktionen sind bei Client-Server-Systemen voneinander isoliert; eine Transaktion bekommt nichts davon mit, was eine andere Transaktion tut, es sei denn, beide versuchen, denselben Datensatz zu ändern.

Nun ist es aber hin und wieder erforderlich, Informationen zwischen Transaktionen auszutauschen. Die einzige Möglichkeit dazu sind Generatoren, denn deren Zweck ist es ja gerade, auch bei vielen laufenden Transaktionen eindeutige Werte zu erzeugen.

Intern sind Generatoren nichts anderes als 4-Byte-Speicherzellen, denen mit SET GENERATOR Werte zugewiesen werden, die dann mit GEN_ID wieder ausgelesen werden können. Sollen jedoch andere Datenformate übertragen werden (Text, Gleitkommazahlen, Zeitstempel), dann kann man beispielsweise diese Werte mittels UDFs in Integer-Zahlen wandeln und über einen oder mehrere Generatoren übertragen.

13.2.2 Automatische Transaktionskontrolle

SQL-Datenbanken wickeln alle Datenzugriffe grundsätzlich im Kontext von Transaktionen ab. Nun haben wir aber auf *InterBase*-Tabellen zugegriffen, ohne eine Transaktion gestartet zu haben. Dies funktioniert nur dadurch, dass die BDE im Hintergrund für jeden Zugriff eine eigene Transaktion durchführt.

Blättern wir noch einmal zurück zu Kapitel 13.1:

```
2    SQL Vendor: INTRBASE - isc_dsql_allocate_statement
3    SQL Vendor: INTRBASE - isc_start_transaction
     ...
13   SQL Vendor: INTRBASE - isc_commit_transaction
14   SQL Stmt: INTRBASE - Close
15   SQL Vendor: INTRBASE - isc_dsql_free_statement
```

Selbst für die Ausführung eines Statements, das nur Daten abfragt, wird automatisch eine Transaktion gestartet, die hinterher bestätigt wird. Wenn sehr viele Aktionen hintereinander ausgeführt werden, dann mindert das deutlich die Ausführungsgeschwindigkeit; wir werden uns das am Ende des Kapitels genauer ansehen. In diesem Fall startet man explizit eigene Transaktionen.

SQLPassThruMode

In diesem Zusammenhang wollen wir eine Einstellung beim Einrichten eines Aliases besprechen, nämlich den Wert der Eigenschaft *SQLPassThruMode*. Damit Sie verstehen können, worin der Unterschied der möglichen Einstellungen liegt, müssen wir jedoch zunächst den Begriff der durchgereichten *SQL-Anweisungen* klären:

Der Zugriff auf *InterBase*-Tabellen kann mit *TTable* oder *TQuery* erfolgen. Erfolgt er mit *TTable*, dann erstellt die BDE dafür die entsprechenden SQL-Anweisungen. Bei der Verwendung von *TQuery* werden jedoch – sofern SQLQRYMODE nicht auf LOCAL gesetzt wird – die SQL-Anweisungen ohne Bearbeitung an den Server durchgereicht; hier handelt es sich dann um eine durchgereichte SQL-Anweisung.

Eine Ausnahme bilden diejenigen *TQuery*-Komponenten, bei denen die Eigenschaft *RequestLive* auf *true* gesetzt wurde. Da SQL-Server mit dieser Eigenschaft nichts anfangen können, erfolgt eine Bearbeitung durch die BDE.

Für die Eigenschaft *SQLPassThruMode* stehen drei Werte zur Auswahl:

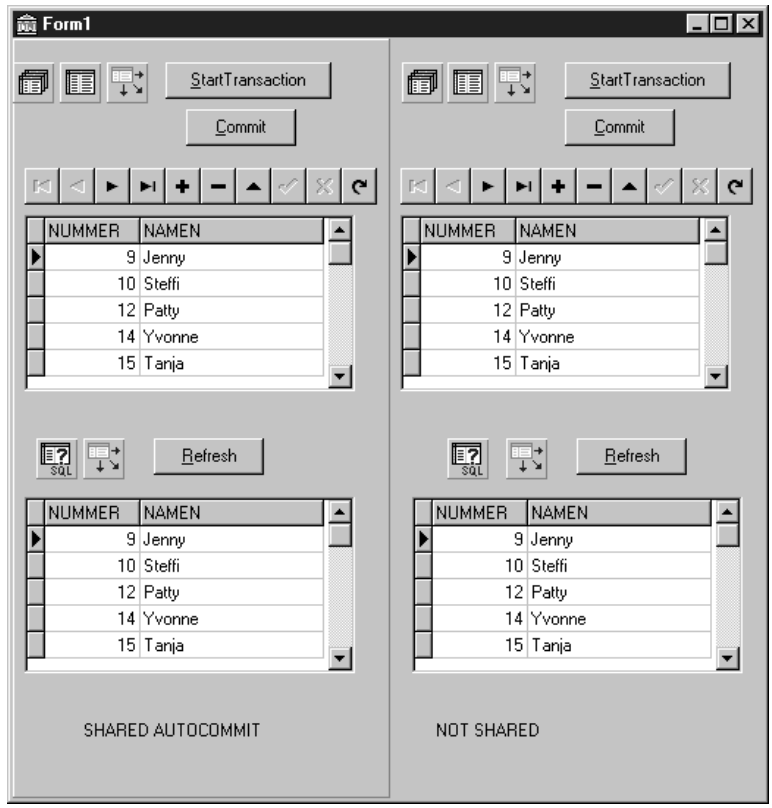
- Bei der Einstellung SHARED AUTOCOMMIT teilen sich die durchgereichten und die lokal bearbeiteten Aktionen eine Datenbankverbindung. Wenn keine explizite Transaktion gestartet wird, wird für jede Aktion eine Transaktion gestartet.

Die Einstellung SHARED AUTOCOMMIT ist die Voreinstellung für alle Server-Verbindungen. Sie sollten nur dann einen anderen Wert verwenden, wenn Sie sich wirklich sicher sind, dass Sie etwas anderes wollen.

- Auch bei der Einstellung SHARED NOAUTOCOMMIT teilen sich die durchgereichten und die lokal bearbeiteten Aktionen eine Datenbankverbindung. Im Unterschied zu SHARED AUTOCOMMIT werden jedoch bei den durchgereichten Verbindungen keine automatischen Transaktionen erstellt. Bei den lokal bearbeiteten Anweisungen werden auch hier automatisch Transaktionen erstellt.

  Bei der Einstellung SHARED NOAUTOCOMMIT brauchen Sie nicht zu versuchen, die Eigenschaft *RequestLive* einer beteiligten *TQuery*-Komponente auf *true* zu setzen – das wird mit einer Fehlermeldung quittiert.

- Bei der Einstellung NOT SHARED werden für lokal bearbeitete und durchgereichte Verbindungen zwei getrennte Verbindungen zur Datenbank erstellt (Sie können das überprüfen, indem Sie mit der *InterBase*-Eigenschaftsseite die Zahl der Datenbankverbindungen ermitteln). Welche Auswirkung diese Einstellung hat, wollen wir gleich anhand eines Beispielprogramms ermitteln.



In einem Formular haben wir für jeweils einen SHARED AUTOCOMMIT- und einen NOT SHARED-Alias eine *TTable*- und eine *TQuery*-Instanz nebst Datenquellen- und Datengitter-Komponenten. Mit Hilfe der altbekannten Anweisungen

```
procedure TForm1.Table1AfterDelete(DataSet: TDataSet);
begin
  with Query1 do
  begin
    Close;
    Open;
  end;
end;
```

für das *OnAfterDelete*- und das *OnAfterPost*-Ereignis sorgen wir dafür, dass mit der *TTable*-Instanz vorgenommene Änderungen auch im Datengitter sichtbar werden.

Fügen wir nun einen Datensatz ein, dann wird er im *TQuery*-Grid der gleichen Datenbankverbindung sofort sichtbar, bei der anderen Datenbankverbindung muss die Aktualisierung manuell ausgelöst werden. Bisher erfolgt also alles noch wie erwartet.

Nun fügen wir jeweils zwei Buttons ein, mit denen wir Transaktionen starten und abschließen können.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DataBase1.StartTransaction;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  DataBase1.Commit;
end;
```

Nun starten wir bei der SHARED AUTOCOMMIT-Verbindung eine Transaktion und fügen einen neuen Datensatz ein. Es wird jedoch noch kein *Commit* durchgeführt. Der neue Datensatz wird im dazugehörenden *TQuery*-Grid sofort sichtbar. Bei der anderen Datenbankverbindung können Sie so oft aktualisieren, wie Sie wollen – da die Transaktion noch nicht abgeschlossen ist, können die Daten von anderen Transaktionen noch nicht gelesen werden. Führen Sie jetzt einen *Commit* durch und aktualisieren Sie dann wieder; nun ist der neue Datensatz vorhanden.

Als Nächstes führen wir diesen Versuch mit der anderen Datenbankverbindung durch: Starten Sie eine Transaktion auf der NOT SHARED-Seite und fügen Sie einen neuen Datensatz ein. Dieser wird im dazugehörenden *TQuery*-Grid nicht angezeigt. Die Anzeige im *TQuery*-Grid läuft nämlich über eine andere Transaktion. Erst wenn Sie die Transaktion mit *Commit* abschließen, können Sie den Datensatz in allen anderen Datenbankverbindungen auslesen.

Zum Thema *Transaktionen* sollten Sie sich auch Kapitel 11.4 ansehen.

13.3 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.

13.3.1 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 setzen, dies würde die Fehlermeldung *Tabelle nicht vorhanden* nach sich ziehen. (Schließlich kann man ja wohl nicht ernsthaft erwarten, dass so eine Abfrageprozedur 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.

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  with Query2 do
  begin
    SQL.Clear;
    SQL.Add('EXECUTE PROCEDURE p_test ("Ramona")');
    ExecSQL;
  end;
end;
```

Da eine solche STORED PROCEDURE keine Ergebnismenge liefert, darf hier die Datenmenge nicht mit *Open* oder *Active := true* geöffnet werden.

13.3.2 Zugriff mit TStoredProc

Bei manchen Servern wie beispielsweise bei 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 vermutlich die Methode *GetResults* benötigen; Näheres in der Online-Hilfe.

13.4 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), 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 (eine bessere Lösung wird im Kapitel *Lösung mit STORED PROCEDURES* vorgestellt):

```
procedure TForm1.Table1AfterInsert(DataSet: TDataset);
begin
  table1.FieldByName('nummer').AsInteger := 1;
end;

procedure TForm1.Table1AfterPost(DataSet: TDataset);
begin
  Table1.Refresh;
end;
```

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* vorübergehend auf *false* gesetzt werden kann.

Beim Aufruf der Methode *Post* wird dieser Datensatz in die Datenbank geschrieben und die Spalte *nummer* vom TRIGGER korrekt gesetzt. Damit dieser Wert nun auch in der Anwendung angezeigt wird, muss die Methode *Table1.Refresh* aufgerufen werden.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  with table1 do
  begin
    Append;
    FieldByName('namen').AsString := Edit1.Text;
    FieldByName('nummer').AsInteger := 2000000000;
```

```
    Post;
    Table1.Refresh;
  end;
end;
```

Werden keine datensensitiven Dialogelemente verwendet, 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, anderenfalls beschwert sich die BDE über einen NULL-Wert.

Wird eine *TQuery*-Komponente verwendet, 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 leider die Methode *Refresh* nicht mehr (damit kann der Server nichts anfangen), so dass man diese mit den folgenden beiden Anweisungen simulieren muss:

```
procedure TForm1.Query1AfterPost(DataSet: TDataset);
begin
  with Query1 do
  begin
    Close;
    Open;
  end;
end;
```

Soll das Ganze ohne datensensitive Dialogelemente bewerkstelligt werden, wird dafür folgende Prozedur verwendet (man könnte hier auch Parameter benutzen, siehe Kapitel 5):

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  with query1 do
  begin
    SQL.Clear;
    SQL.Add('INSERT INTO t_namen (namen)');
    SQL.Add('VALUES ("' + Edit1.Text + '")');
    ExecSQL;
    Table1.Refresh;
  end;
end;
```

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*.

Lösung mit STORED PROCEDURES

Deutlich eleganter, aber eben auch aufwändiger als die eben genannte Lösung ist die Verwendung einer STORED PROCEDURE. Diese könnte man beispielsweise wie folgt definieren:

```
CREATE GENERATOR g_testadr;

CREATE PROCEDURE p_testadr RETURNS (gen INTEGER)
AS
BEGIN
  gen = GEN_ID(g_testadr, 1);
  SUSPEND;
END

CREATE TRIGGER trig_testadr FOR testadr
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF(NEW.nummer IS NULL)
    THEN NEW.nummer = GEN_ID(g_testadr, 1);
END
```

Der TRIGGER wird so abgeändert, dass er nur dann einen Wert für das Feld *nummer* einfügt, wenn keiner vorhanden ist. Die STORED PROCEDURE tut nichts anderes, als einen neuen Generatorwert zu ermitteln. Die Anweisung SUSPEND ist erforderlich, um auf diese Prozedur mit einer *TQuery*-Instanz zugreifen zu können (ansonsten müsste man *TStoredProc* verwenden).

```
SELECT * FROM p_testadr
```

Die SQL-Anweisung der *TQuery*-Instanz besteht aus einer simplen Abfrage.

```
procedure TForm1.Table1AfterInsert(DataSet: TDataSet);
begin
  with Query1 do
  begin
    Open;
```

```
    Table1.FieldByName('nummer').AsInteger
      := FieldByName('gen').AsInteger;
    Close;
  end;
end;
```

Sobald ein neuer Datensatz eingefügt wird, wird ein Generatorwert angefordert und in das Feld *nummer* geschrieben; somit wird gleich mit den korrekten Daten gearbeitet.

Der eine oder andere Leser mag sich fragen, aus welchen Gründen dieser Aufwand getrieben wird und warum der Wert nicht gleich mit

```
SELECT MAX(nummer) FROM testadr
```

ermittelt wird. Dies wäre jedoch bei einem Client-Server-System sehr gefährlich: Zwischen dem Zeitpunkt der Abfrage und dem Einfügen des Datensatzes kann der entsprechende Wert schon von einer anderen Transaktion gesetzt worden sein. Die Verwendung eines Generators garantiert dagegen eindeutige Werte.

13.5 Optimierung

Der Umstieg von Desktop-Datenbanken zu einem Client-Server-System wird von Delphi sehr einfach gemacht, eigentlich muss lediglich ein Alias umgestellt werden. Das böse Erwachen kommt meist dann, wenn man bemerkt, dass die Reaktionszeit der Datenbank indiskutabel ist. Ganz so einfach ist das mit den Client-Server-Systemen eben doch nicht.

13.5.1 Erstellen von Testdaten

Zunächst wollen wir uns ein paar Tabellen mit Testdatensätzen füllen. Die prinzipielle Vorgehensweise kennen Sie ja bereits aus Kapitel 5.1, das dort vorgestellte Programm werden wir auch entsprechend abwandeln. Zunächst einmal werden die Tabellen *namen* und *orte* auf dem Server eingerichtet:

```
CREATE TABLE namen
  (nummer INTEGER NOT NULL,
  vorname VARCHAR(20) CHARACTER SET ISO8859_1 COLLATE DE_DE,
  nachname VARCHAR(20) CHARACTER SET ISO8859_1 COLLATE DE_DE,
  strasse VARCHAR(20) CHARACTER SET ISO8859_1 COLLATE DE_DE,
  PRIMARY KEY(nummer));
```

Die Tabelle *orte* ist entsprechend aufgebaut. Nun wollen wir die Daten aus der Desktop-Datenbank auf den Server transferieren. Weil wir bequem sind, verwenden wir dazu die Datenbankoberfläche (TOOLS|TABELLENOPERATIONEN|KOPIEREN). Würden wir jedoch direkt in die Tabelle *namen* kopieren, dann würde eine exakte Kopie der Desktop-Datenbanktabelle angelegt werden, was die Sortierreihenfolge löschen würde. Wir kopieren also in eine temporäre Tabelle namens *namen_t*.

```
CREATE PROCEDURE uber
AS
  DECLARE VARIABLE nummer INTEGER;
  DECLARE VARIABLE vor VARCHAR(20);
  DECLARE VARIABLE nach VARCHAR(20);
  DECLARE VARIABLE stra VARCHAR(20);
BEGIN
  FOR SELECT nummer, vorname, nachname, stra_e FROM namen_t
      INTO :nummer, :vor, :nach, :stra
    DO INSERT INTO namen (nummer, vorname, nachname, strasse)
      VALUES (:nummer, :vor, :nach, :stra);‘
END;
```

Als Nächstes wird eine STORED PROCEDURE erstellt, welche alle Datensätze von einer Tabelle in die andere kopiert. Beachten Sie dabei, dass InterBase kein *ß*-Zeichen in den Bezeichnern zulässt und aus dem Spaltennamen *straße* den Namen *stra_e* gemacht hat.

```
EXECUTE PROCEDURE uber
```

Mit EXECUTE PROCEDURE wird die Prozedur ausgeführt. Analog dazu wird mit der Tabelle *orte* verfahren, außerdem muss die Tabelle *testadr* erstellt werden.

Daten generieren

Als Nächstes wollen wir das Programm zum Erstellen der Testdatensätze entsprechend anpassen, das wir schon in Kapitel 5.1 verwendet haben.

Zunächst einmal fügen wir eine *TDataBase*-Komponente ein, über welche die Datenbankverbindung der drei *TTable*-Instanzen läuft. Mit den Eigenschaften *Params* und *LoginPrompt* vermeiden wir den lästigen Login-Dialog (siehe Kapitel 4.5.3).

Um ein autoinkrementierendes Feld zu erzeugen, müssen zunächst ein Generator und die folgende Prozedur erzeugt werden:

```
CREATE PROCEDURE P_testadr
RETURNS (gen INTEGER)
AS
```

```
BEGIN
  en = GEN_ID(g_testadr, 1);
  SUSPEND;
END;
```

Dank der Anweisung SUSPEND können wir den Wert mit einer *TQuery*-Instanz ermitteln:

```
SELECT * FROM p_testadr
```

Die Abfrage wird nicht geöffnet, das erfolgt jeweils beim Einfügen eines neuen Datensatzes.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  i, j, l: integer;
  s, t: string;
begin
  Table3.Append;
  with Query1 do
  begin
    Open;
    Table3.FieldByName('nummer').AsInteger
      := FieldByName('gen').AsInteger;
    Close;
  end;

  {Vornamen, Nachnamen, Straße und Hausnummer}
    ...
end; {procedure TForm1.BitBtn1Click}
```

Des Weiteren soll der Anwender wählen können, ob die Anzeige während des Einfügens aktualisiert wird oder nicht.

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  DataSource1.Enabled := CheckBox1.Checked;
  DataSource2.Enabled := CheckBox1.Checked;
  DataSource3.Enabled := CheckBox1.Checked;
end;
```

Außerdem soll die Möglichkeit bestehen, alle Datensätze im Kontext einer expliziten Transaktion durchzuführen:

```
procedure TForm1.BitBtn2Click(Sender: TObject);
var
  i, z: integer;
  c, t1, t2: int64;
begin
  QueryPerformanceFrequency(c);
  QueryPerformanceCounter(t1);
  if CheckBox2.Checked
    then Database1.StartTransaction;
  for i := 1 to 1000 do
  begin
    BitBtn1Click(Sender);
    if Quelldatenbankenanzeigen1.Checked = true
      then Application.Processmessages;
  end; {for i := 1 to 100 do}
  if CheckBox2.Checked
    then Database1.Commit;
  QueryPerformanceCounter(t2);
  z := 1000 * (t2 - t1) div c;
  Label12.Caption := 'Zeit: ' + IntToStr(z) + ' ms';
end; {procedure TForm1.BitBtn2Click}
```

Bei dieser Gelegenheit wollen wir auch gleich die Ausführungszeit messen, und zwar zunächst die Zeiten für den Fall, dass sich Client und Server auf demselben Rechner befinden (100 MHz-Pentium, SCSI-Platte).

	Client und Server auf einem Rechner		Netzwerkverbindung	
Transaktion	mit	ohne	mit	ohne
mit Anzeige	96 s	318 s	199 s	435 s
ohne Anzeige	57 s	273 s	161 s	391 s

Wie Sie sehen, kann man das Einfügen von Datensätzen ganz erheblich beschleunigen, wenn man eine explizite Transaktion startet, so dass nicht für jeden Datensatz eine eigene Transaktion gestartet und beendet werden muss. Immer noch sehr deutlich sind die Unterschiede, wenn der Client auf einem anderen Rechner läuft, und somit alle Daten über ein Netzwerk müssen (Server 100 MHz-Pentium, SCSI-Platte, Client 200 MHz-MMX, Notebook, 10 MBit-Netzwerk).

Fazit: Wenn Sie mehrere DML-Operationen hintereinander durchführen, dann sollten Sie eine explizite Transaktion starten.

Verkaufsdaten generieren

Damit wir uns auch mit JOINS beschäftigen können, wollen wir auch noch eine weitere Tabelle generieren. Wir tun dabei so, als ob die Adressen in *testadr* Kunden einer Firma seien, und erstellen eine Tabelle *verkauf*, in der wir speichern, wann wer für wie viel bei dieser Firma eingekauft hat.

```
CREATE TABLE verkauf
  (nummer INTEGER NOT NULL PRIMARY KEY,
  datum DATE,
  kunde INTEGER NOT NULL,
  summe FLOAT,
  FOREIGN KEY (kunde) REFERENCES testadr (nummer))
```

Auch hier wird aus dem Feld *nummer* mittels eines TRIGGERS und einer STORED PROCEDURE ein selbstinkrementierendes Feld gemacht, aber das kennen Sie ja bereits.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i, z: integer;
  c, t1, t2: int64;
begin
  QueryPerformanceFrequency(c);
  QueryPerformanceCounter(t1);
  if CheckBox2.Checked
    then Database1.StartTransaction;
```

An dieser Stelle wollen wir wieder die Ausführungszeit messen (etwa 369 Sekunden bei 10000 Datensätzen bei expliziter Transaktion), außerdem soll die Möglichkeit bestehen, eine explizite Transaktion zu starten.

```
  for i := 1 to 10000 do
  begin
    with Table4 do
    begin
      Append;
      // Datensatznummer
      Query2.Open;
      FieldByName('Nummer').AsInteger
        := Query2.FieldByName('gen').AsInteger;
      Query2.Close;
      // Datum
      FieldByName('Datum').AsDateTime
        := Now - (random(100000) / 100);
```

Hier sollen gleich 10000 Datensätze auf einmal erstellt werden. Für das Feld *nummer* wird – wie gehabt – per STORED PROCEDURE der Generatorwert abgefragt. Als Datum wird ein Zufallswert generiert, der etwa die letzten drei Jahre vor dem aktuellen Datum liegt. Mit Hilfe der Division sorgen wir dafür, dass auch die Uhrzeiten zufällig sind.

```
      // Kunde
      repeat
        Query3.Close;
        Query3.ParamByName('Nummer').AsInteger := random(10000);
        Query3.Open;
      until Query3.RecordCount = 1;
      FieldByName('Kunde').AsInteger
        := Query3.FieldByName('Nummer').AsInteger;
```

Wenn wir für die Tabelle *testadr* 10 000 Testdatensätze erstellt haben, dann müssten eigentlich Nummern von eins bis 10 000 durchgehend vergeben sein. Nicht immer werden Sie jedoch mit einer so übersichtlichen Ausgangslage konfrontiert sein. Im realen Betrieb wäre durchaus damit zu rechnen, dass die Nummern nicht durchgehend vergeben sind oder dass ein Teil von Ihnen bereits wieder gelöscht wurde.

Verwenden Sie allerdings eine Nummer, für die es keinen Datensatz in *testadr* gibt, dann verursachen Sie eine Fremdschlüsselverletzung. Wir prüfen deshalb bei jeder generierten Nummer, ob ein entsprechender Datensatz in *testadr* existiert.

```
      // Summe
      FieldByName('Summe').AsCurrency := random(1000000) / 100;
      Post;
    end; {with Table4 do}
  end; {for i := 1 to 100 do}
  if CheckBox2.Checked
    then Database1.Commit;
  QueryPerformanceCounter(t2);
  z := 1000 * (t2 - t1) div c;
  Label12.Caption := 'Zeit: ' + IntToStr(z) + ' ms';
end;
```

Zum Schluss wird auch noch ein Wert für die Spalte *Summe* generiert, danach wird die Operation mit *Post* abgeschlossen. Nach Beendigung der Schleife wird gegebenenfalls die Transaktion mit *Commit* bestätigt und noch die Ausführungszeit angezeigt.

13.5.2 Datensätze filtern

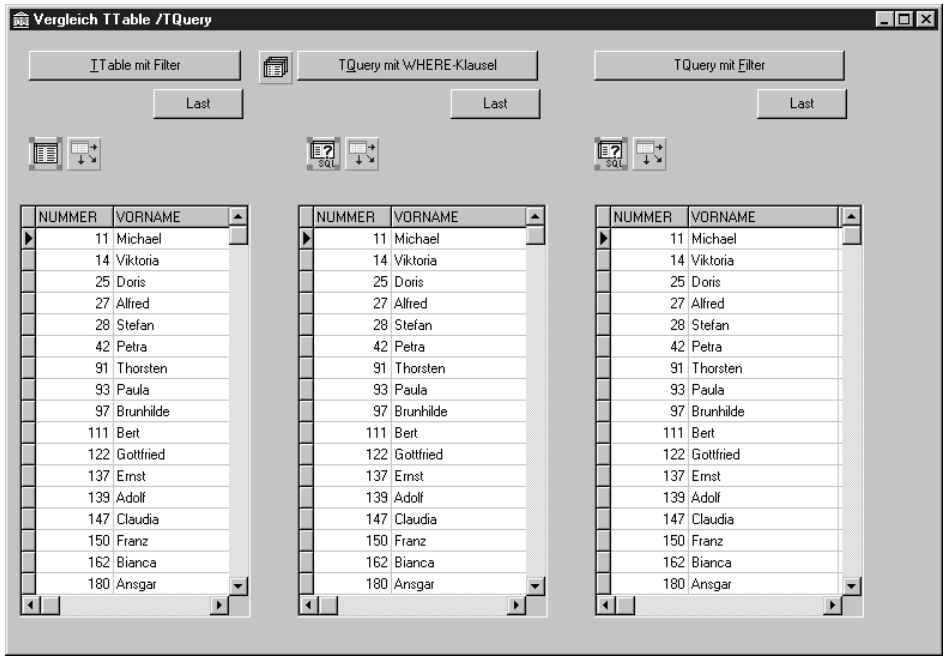
Nach Möglichkeit sollte man keine Abfrage des gesamten Datenbestandes durchführen, weil dann alle Datensätze über das Netzwerk »geschaufelt« werden müssen. Welche Besonderheiten beim Filtern zu beachten sind, soll in den kommenden Abschnitten geklärt werden.

TTable und TQuery

Um einen Datenbestand zu filtern, gibt es drei Möglichkeiten (VIEWS und STORED PROCEDURES einmal außer Acht gelassen):

- Es wird eine *TTable* verwendet und die Eigenschaft *Filter* entsprechend gesetzt.
- Es wird eine *TQuery* verwendet und eine entsprechende WHERE-Klausel formuliert.
- Schließlich kann die Eigenschaft *Filter* auch noch im Zusammenhang mit *TQuery* verwendet werden.

Wir werden diese drei Möglichkeiten miteinander vergleichen. Dazu bestücken wir ein Formular gemäß der folgenden Abbildung:



Bei der *TTable*- und einer *TQuery*-Instanz wird der Eigenschaft *Filter* der folgende String zugewiesen:

```
Nachname='B*'
```

Des Weiteren wird die Eigenschaft *Filtered* auf *true* gesetzt. Der gefilterten *TQuery*-Instanz wird folgende SQL-Anweisung zugewiesen:

```
SELECT * FROM testadr
```

Die andere *TQuery*-Instanz erhält folgende Anweisung:

```
SELECT * FROM testadr
  WHERE nachname LIKE "B%"
```

Mit Hilfe der Buttons werden die Datenmengen geöffnet beziehungsweise der jeweils letzte Datensatz wird aufgerufen. Dabei erfolgt eine Messung der Ausführungszeit.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  c, t1, t2: int64;
begin
  QueryPerformanceFrequency(c);
  QueryPerformanceCounter(t1);
  Table1.Open;
  {Table1.Last;}
  QueryPerformanceCounter(t2);
  Label1.Caption := IntToStr(Round(1000*(t2 - t1)/c)) + ' ms';
end;
```

Die folgende Tabelle zeigt die Messwerte, die hier über ein Netzwerk gemessen wurden. Befinden sich Client und Server auf demselben Rechner, ergeben sich aber fast identische Werte.

	Open	Last
TTable gefiltert	722 ms	2268 ms
TQuery mit WHERE-Klausel	112 ms	2198 ms
TQuery gefiltert	9124 ms	7 ms

Woher kommen also die extremen Werte bei einer gefilterten *TQuery*?

Während bei einer WHERE-Klausel die Abfrage vom Server bearbeitet und nur ein Teil der Ergebnismenge zum Client geschickt wird, werden beim Setzen der Eigenschaft *Filter* alle Datensätze zum Client übertragen, der dann lokal die Filterung übernimmt. Letzteres hat natürlich den Vorteil, dass sehr schnell durch die Datenmenge navigiert werden kann, weil der komplette Datenbestand lokal im Speicher liegt.

Wird jedoch die Abfrage vom Server bearbeitet, werden nur diejenigen Datensätze zum Client übertragen, die auch tatsächlich angezeigt werden. Erst beim Aufruf von *Last* werden alle Datensätze übermittelt, was dann entsprechend viel Zeit braucht.

Die Frage ist nun, warum *TTable* beim Setzen eines Filters so anders reagiert als *TQuery*. Ursprünglich bestand zwischen diesen beiden Komponenten in dieser Hinsicht kein Unterschied, *TTable* hat sich also beim Filtern so verhalten wie *TQuery*.

Das hatte zur Folge, dass sich viele Entwickler ganz energisch bei Borland beschwerten, dass das Client-Server-System viel zu langsam sei (dabei hätten sie es nur richtig machen müssen). Um den Ruf der Firma zu retten, hat man die Sache dann so geändert, dass bei *TTable* die Filteranweisung automatisch in eine WHERE-Klausel umgewandelt wird. Wird jedoch bei *TQuery* ein Filter gesetzt, geht man davon aus, dass der Entwickler weiß, was er tut.

Nun besteht jedoch die Möglichkeit, dass das Öffnen einer Tabelle oder Abfrage ruhig etwas länger dauern darf, dass aber das Navigieren sehr schnell gehen soll. Hier muss man dann nicht den Umweg über einen Filter gehen – es reicht, die Methode *FetchAll* aufzurufen, welche alle Datensätze anfordert.

Die Durchführung von *FetchAll* dauert hier im Beispiel etwas über 2000 ms, also etwa so lange wie der Aufruf von *Last*. Der anschließende Aufruf von *Last* erfolgt extrem schnell (*TTable* 150 ms, *TQuery* 5 ms). Wird bei einer lokal gefilterten *TQuery*-Instanz die Anweisung *FetchAll* aufgerufen, wird diese gar nicht ausgeführt, weil ohnehin schon alle Daten lokal vorhanden sind.

13.5.3 Indizes verwenden

Eine sehr effektive Möglichkeit, Abfragen zu beschleunigen, ist die Verwendung von Indizes. Wir wollen auch das testen. Dazu erstellen wir eine Abfrage mit einer WHERE-Klausel, eine weitere mit einer ORDER-Klausel und eine dritte mit sowohl einer WHERE- als auch einer ORDER-Klausel. Zunächst messen wir die Ausführungszeiten ohne Indizes.

Wir stellen dabei fest, dass das Sortieren einer Abfrage deutlich länger dauert als das Filtern (was auch sehr einsichtig ist). Nun wollen wir zunächst einen Index auf die Spalte *nachname* erstellen.

```
CREATE INDEX testadr_nachname ON testadr(nachname)
```

Sie müssen jedem Index einen Namen geben. Das ist sehr sinnvoll, denn manchmal soll ein Index auch wieder gelöscht werden, und dazu braucht man dessen Namen.

Wenn wir nun dieselben Abfragen wiederholen, stellen wir fest, dass insbesondere das Sortieren einer indizierten Spalte extrem beschleunigt wird.

	ohne Index	Index auf *nachname*	Index auf *vorname* und *nachname*
SELECT * FROM testadr WHERE nachname LIKE "B%"	194 ms	94 ms	93 ms
SELECT * FROM testadr WHERE nachname LIKE "B%" ORDER BY vorname	2364 ms	485 ms	137 ms
SELECT * FROM testadr WHERE nachname LIKE "B%" ORDER BY nachname	2911 ms	123 ms	125 ms

Erstellen Sie jetzt einen Index auf die Spalte *vorname*.

```
CREATE INDEX testadr_vorname ON testadr(vorname)
```

Nun wollen wir in der Abfrage auch entgegengesetzt sortieren:

```
SELECT * FROM testadr
  WHERE nachname LIKE "B%"
  ORDER BY vorname DESC
```

Hier stellen wir fest, dass ein Index bei umgekehrter Sortierung keine Beschleunigung bewirkt. Erstellen Sie den Index deshalb in der beabsichtigten Suchrichtung, falls nötig, erstellen Sie für eine Spalte zwei Indizes.

```
CREATE DESC INDEX testadr_vorname_desc ON testadr(vorname)
```

Beachten Sie hier die Anweisung DESC hinter dem Schlüsselwort CREATE.

Datensätze in indizierte Spalten einfügen

Wo Licht ist, findet man meist auch Schatten: Werden Datensätze eingefügt oder gelöscht oder werden Datensätze in den Indexspalten geändert, dann müssen die Indizes entsprechend angepasst werden, was die Operationen naturgemäß nicht beschleunigt.

Um festzustellen, wie groß hier die Auswirkungen sind, wollen wir nochmals einige 1000 Datensätze in die Tabelle *testadr* einfügen. Das eine Mal mit den drei eben erstellten Indizes (plus natürlich dem Primärschlüssel-Index auf der Spalte *nummer*), das andere Mal nach dem Löschen dieser drei Indizes. Das Löschen von Indizes erfolgt mit DROP INDEX.

```
DROP INDEX testadr_vorname
```

Wie wir sehen, verzögern Indizes DML-Operationen merklich, aber nicht dramatisch.

	mit Index	ohne Index
mit Transaktion	81 ms	57 ms
ohne Transaktion	280 ms	263 ms

Bei sehr umfangreichen Transaktionen kann es sinnvoll sein, vorher alle Indizes zu löschen, um sie hinterher wieder aufzubauen.

Ergänzend sei daran erinnert, dass für sämtliche Primär-, Sekundär- oder Fremdschlüssel automatisch Indizes erstellt werden.

13.5.4 Vermeidung von LIKE-Operatoren

Eine weitere Möglichkeit zur Optimierung von Abfragen ist die Wahl des geeignetsten Operators. Wie das folgende Beispiel zeigt, sind die Unterschiede deutlich, aber nicht weltbewegend.

Anweisung	Zeit
SELECT * FROM testadr WHERE (nachname LIKE "B%") AND (vorname LIKE "B%")	2587 ms
SELECT * FROM testadr WHERE (nachname > "B") AND (nachname <"C") AND (vorname > "B") AND (vorname <"C")	2431 ms
SELECT * FROM testadr WHERE (nachname BETWEEN "B" AND "C") AND (vorname BETWEEN "B" AND "C")	2415 ms
SELECT * FROM testadr WHERE (nachname STARTING WITH "B") AND (vorname STARTING WITH "B")	1742 ms

13.5.5 Datensätze zählen

Einer der Schwachpunkte von *InterBase* ist die mangelnde Geschwindigkeit beim Zählen von Datensätzen.

RecordCount

Die *TBDEDataSet*-Eigenschaft *RecordCount* liefert die Anzahl der Datensätze in einer Tabelle oder Abfrage. Bei Desktop-Datenbanken wird dieser Wert von der BDE erfragt und ist so gut wie verzögerungsfrei vorhanden.

Anders verhält es sich bei *InterBase*-Datenbanken. Sobald hier auf die Eigenschaft *RecordCount* zugegriffen wird, fordert der Client alle Datensätze beim Server an, es wird sozusagen eine *FetchAll*-Operation durchgeführt. (Ein anschließender Aufruf von *Last* erfolgt dann auch verzögerungsfrei.)

Es ist somit keine gute Idee, eine größere Menge von Datensätzen mit *RecordCount* zu zählen.

Die Alias-Eigenschaft MAX ROWS

Beim Einrichten eines Aliases lässt sich mit der Eigenschaft MAX ROWS bestimmen, wie viele Datensätze pro Tabelle oder Abfrage maximal vom Server zum Client übertragen werden sollen. Ein in der Praxis brauchbarer Wert liegt bei 100, weil man hier in einem DBGrid noch ein wenig scrollen kann, die Geschwindigkeit aber im akzeptablen Bereich liegt.

Gibt es mehr Datensätze in der Datenmenge, dann werden beim Zugriff auf die Eigenschaft *RecordCount* auch nur so viele Datensätze, wie durch MAX ROWS bestimmt wurden, zum Client übertragen und mit *RecordCount* gezählt, so dass hier in der Regel kein nennenswerter Zeitverlust entsteht. Man sollte den Anwender jedoch darauf aufmerksam machen, dass nicht alle Datensätze angezeigt werden.

```
  ...
if Query1.RecordCount = 100
  then ShowMessage ('Es werden nicht alle Datensätze angezeigt.'
    + #13#10 + 'Setzen Sie weitere Filter!');
  ...
```

Mit dem Ausdruck *#13#10* bewirken Sie einen Zeilenumbruch.

SELECT COUNT(*)

Dem SQL-Kundigen wird sicher die Idee gekommen sein, direkt den Server nach der Zahl der Datensätze zu fragen.

```
SELECT COUNT(*) FROM testadr
  WHERE nachname STARTING WITH "B"
```

Diese Anweisung zieht erfreulich wenig Kommunikation zwischen Client und Server nach sich, was aber die Sache auch nicht beschleunigt, weil nun der Server entsprechend lange braucht (2,6 s bei 1104 Datensätzen).

Datensätze im Thread zählen

In den Fällen, in denen man die Zahl der Datensätze benötigt, kann man diese in einem seperaten Thread ermitteln. Auf diese Weise ist das Ergebnis der Abfrage schnell vorhanden und die Datensatzzahl wird vom Thread wenig später »nachgeliefert«.

Die Vorgehensweise ist eigentlich sehr einfach, um jedoch die möglichen Fehlerquellen auszuschließen, sollten Sie sich an folgende Reihenfolge halten:

- Zunächst einmal wird eine *TSession*-Komponente eingefügt. Bei dieser muss nur die Eigenschaft *SessionName* auf einen frei zu wählenden Wert gesetzt werden; wir wollen hier *Thread* verwenden. Anschließend wird die Eigenschaft *Active* auf *true* gesetzt.
- Als Nächstes fügen wir eine weitere *TDataBase*-Komponente ein. Als *AliasName* wählen wir *IB_TEST* aus, als *DatabaseName* geben wir *IN_2 ein*. Mit Hilfe der Eigenschaften *Params* und *LoginPrompt* vermeiden wir das manuelle Einloggen in die Datenbank, Sie kennen das ja schon.

  Wichtig ist, dass Sie die Eigenschaft *SessionName* auf *Thread* abändern. Setzen Sie nun die Eigenschaft *Connected* auf *true* und prüfen somit, ob alles funktioniert.
- Die nächste Komponente ist eine *TQuery*. Setzen Sie hier die Eigenschaften *DatabaseName* auf *IN_2* und *SessionName* auf *Thread*.
- Sicherheitshalber wollen wir die Datenbankverbindung testen. Hängen Sie dazu eine *TDataSource*- und eine *TDBGrid*-Instanz an *Query2*. Die Eigenschaft *SQL* von *Query2* erhält die Anweisung *SELECT COUNT(*) FROM testadr*, anschließend wird *Active* auf *true* gesetzt. Die Zahl der Datensätze sollte nun angezeigt werden.

  Funktioniert alles, können Sie *Query2.Active* wieder auf *false* setzen und die *TDataSource*- sowie die *TDBGrid*-Instanz löschen.
- Ein Thread muss nicht gleich in eine eigene Unit gesteckt werden. Meist sind hier nur so wenige Anweisungen erforderlich, dass man diese einfach in die Unit das jeweiligen Formulars einbaut. Zunächst benötigen wir eine Typendefinition und eine Variable dieses Typs.

```
type
  TQueryThread = class(TThread)
  protected
    procedure Execute; override;
  end;

implementation

var
  Form1: TForm1;
  QueryThread: TQueryThread;
```

- Nach dem Starten der eigentlichen Abfrage soll unser Thread gestartet werden. Dazu wird die Prozedur *BitBtn1Click* wie folgt ergänzt:

```
    ...
  if CheckBox2.Checked = true then
  begin
    QueryThread := TQueryThread.Create(false);
    QueryThread.FreeOnTerminate := true;
    QueryThread.Priority := tpLowest;
  end;
end; {procedure TForm1.BitBtn1Click}
```

  Zunächst wird der Thread mit *Create* erzeugt. Der Parameter *CreateSuspended* wird dabei auf *false* gesetzt, schließlich soll der Thread gleich mit der Arbeit beginnen. Die Eigenschaft *FreeOnTerminate* wird auf *true* gesetzt, damit der Thread nach getaner Arbeit entfernt wird. Die Eigenschaft *Priority* wird auf *tpLowest* gesetzt, damit der Thread nicht viel Prozessorzeit anfordert und mit dem Vordergrund-Thread zügig gearbeitet werden kann.

- Nun muss noch die Methode *Execute* überschrieben werden, damit der Thread weiß, was er überhaupt tun soll.

```
procedure TQueryThread.Execute;
var
  s: string;
  i: integer;
begin
  Form1.Query2.Close;
  Form1.Query2.SQL.Assign(Form1.Query1.SQL);
  s := Form1.Query2.SQL.Strings[0];
  i := Pos('*', s);
  if i = 0 then exit;
```

```
    Delete(s, i, 1);
    Insert('COUNT(*)', s, i);
    Form1.Query2.SQL.Strings[0] := s;
    Form1.Query2.Open;
  end;
```

Zunächst einmal wird die SQL-Anweisung von *Query1* nach *Query2* kopiert. Weil der Zugriff auf *RecordCount* bei sehr großen Datenmengen doch erheblich länger dauert als eine entsprechende Abfrage, wollen wir versuchen, aus der jeweiligen SQL-Anweisung eine Abfrage zu generieren, welche die Zahl der Datensätze ermittelt. Dabei gehen wir davon aus, dass die eigentliche Abfrage folgendermaßen beginnt:

```
SELECT * FROM tabelle
  ...
```

Hier ermitteln wir dann das *-Zeichen, löschen es und fügen an seiner Stelle die Zeichenfolge *COUNT(*)* ein. Somit beginnt die zweite SQL-Anweisung dann folgendermaßen:

```
SELECT COUNT(*) FROM tabelle
  ...
```

Bei Abfragen, die gezielt nach Spalten fragen, versagt verständlicherweise diese Methode. Wenn Sie aber nicht gerade einen SQL-Editor erstellen, werden Sie in der Regel wissen, welche SQL-Anweisungen vorkommen werden, und können entsprechende Anweisungen zum Zählen der Datensätze formulieren.

14 Die Interbase-Komponenten

Seit Delphi 5 gibt es eine Sammlung von Komponenten, mit denen direkt auf InterBase zugegriffen werden kann – die BDE ist hier nicht mehr nötig. Da diese Komponenten auch der Professional-Version von Delphi 5 beiliegen, kann man damit Client-Server-Anwendungen schreiben, ohne in die Enterprise-Version investieren zu müssen.

Die Hoffnung, den Zugriff auf InterBase beschleunigen zu können, hatte sich anfangs nicht erfüllt. Bei Messungen, die ich für das Buch *Delphi 5 Datenbankprogrammierung* durchgeführt hatte, war ein Zugriff über die BDE meist ein wenig schneller. Inzwischen sind die IBX-Komponenten noch ein wenig optimiert worden und haben nun ihrerseits leicht »die Nase vorn« – wobei in manchen Fällen durchaus auch die BDE schneller sein kann.

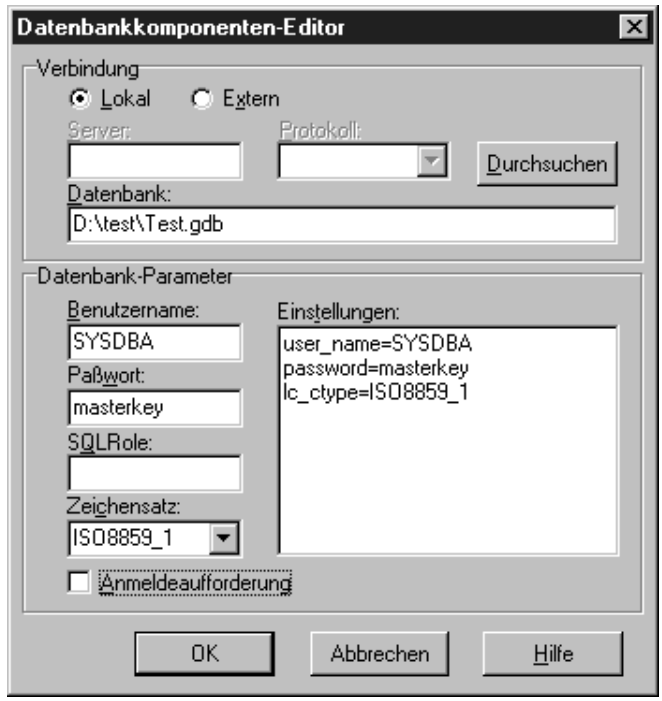
Zudem gibt es einige weitere Gründe, die für die IBX-Komponenten sprechen:

- Der Installationsaufwand der BDE entfällt.
- Die IBX-Komponenten nutzen besser die Möglichkeiten von Interbase – beispielsweise durch die Eigenschaft *GeneratorField* von *TIBDataSet*.
- Die Komponente *IBSQLMonitor* ist auch in der Professional-Version dabei, so dass hier detaillierte Untersuchungen der Kommunikation auch ohne die Enterprise-Version durchgeführt werden können.
- Viele Informationen über die Datenbank liefert auch die Komponente *TIBDatabaseInfo*.

14.1 Ein erstes Beispiel

Wir wollen hier wieder die dreistufige Master-Detail-Verknüpfung aufbauen, mit der wir in Kapitel 1 in die Thematik *TQuery* eingestiegen sind.

Zunächst muss in das Datenmodul eine *TIBDatabase*-Komponente eingefügt werden. Für diese Komponente gibt es einen Komponenteneditor. Zunächst wird hier der Pfad zur Datenbank eingegeben. Befindet sich der Server auf einem anderen Computer, so ist der RadioButton *Extern* zu wählen, des Weiteren sind dann Computer-Name und Netzwerkprotokoll anzugeben. Als Benutzername wird *SYSDBA* und als Passwort wird *masterkey* verwendet. Um das Anmelden beim Verbinden zur Datenbank zu vermeiden, wird die entsprechende Checkbox deaktiviert:



Des Weiteren ist der Sprachtreiber anzugeben, der bei der Definition der Datenbank verwendet wurde. Anschließend wird der Komponentennditor mit *OK* geschlossen und die Eigenschaft *Connected* auf *true* gesetzt.

Zudem wird eine *TIBTransaction*-Komponente benötigt, die mittels der Eigenschaft *DefaultDatabase* mit der *TIBDatabase*-Komponente verknüpft wird. Die Eigenschaft *Active* wird danach auf *true* gesetzt.

Nun werden vier *TIBQuery*-Instanzen eingefügt, die mit den bereits in Kapitel 1 verwendeten SQL-Statements gefüllt werden. Statt – wie bei *TQuery* – für die Eigenschaft *DatabaseName* einen internen oder externen Alias auszuwählen, wird hier mit der Eigenschaft *Database* die *TIBDatabase*-Instanz referenziert. Dabei wird auch die Eigenschaft *Transaction* automatisch gesetzt. Für *IBQuery1* müssen die Parameter entsprechend gesetzt werden, bei *IBQuery2* und *IBQuery3* werden sie der Master-Detail-Verknüpfung entnommen.

Das Öffnen der Datenmenge dauerte etwa 40 ms, das Scrollen etwa 20 ms. Dagegen benötigte das Öffnen der Datenmenge bei der Verwendung von *TQuery* und der BDE etwa 42 ms und das Scrollen etwa 33 ms. Der Geschwindigkeitsunterschied ist also alles andere als »gewaltig«, die IBX-Komponenten sind aber inzwischen messbar schneller.

14.2 Die Komponente TIBDataSet

Bei den InterBase-Komponenten gibt es zwar Entsprechungen zu den BDE-Komponenten (*TIBTable*, *TIBQuery*, *TIBUpdateSQL*), der effektivste Weg geht jedoch in fast allen Fällen über *TIBDataSet*. Deshalb wollen wir diese Komponente nun einmal etwas näher betrachten:

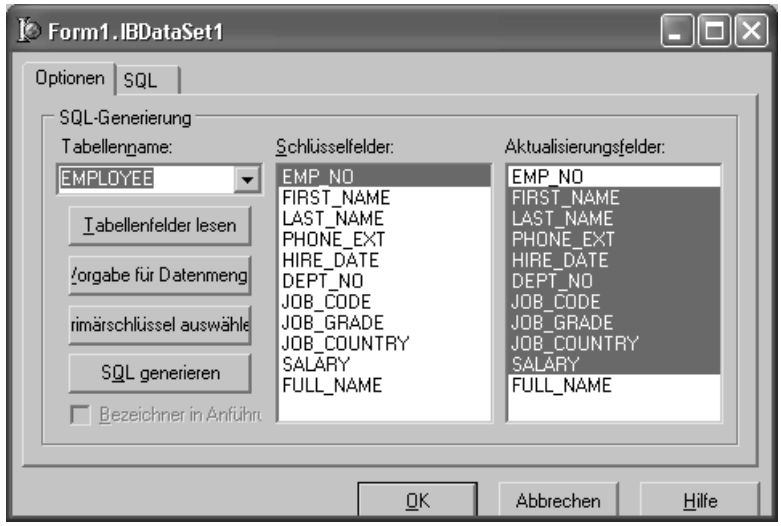
In Kapitel 13 haben wir im Zugriff auf *employee* die Komponenten *TTable* und *TQuery* verglichen. Setzen wir dafür nun auch *TIBDataSet* ein. Hier dauert das Öffnen der Datenmenge 9 ms (im Übrigen auch mit *TIBQuery*). Eigentlich sollte man annehmen, dass durch den Wegfall des »Umwegs BDE« ein schnellerer Zugriff möglich ist. Um der Sache auf den Grund zu gehen, schauen wir uns das Verbindungsprotokoll an (bitte beachten Sie, dass der *SQL Monitor* anders arbeitet und so die Protokolle nicht direkt vergleichbar sind):

```
IBDataSet2: [Prepare] select * from EMPLOYEE
  Plan: PLAN (EMPLOYEE NATURAL)
: [Start transaction]
: [Prepare]
  Select
       F.RDB$COMPUTED_BLR,
       F.RDB$DEFAULT_VALUE,
       R.RDB$FIELD_NAME
    from RDB$RELATION_FIELDS R, RDB$FIELDS F
    where R.RDB$RELATION_NAME = :RELATION
      and R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
      and ((not F.RDB$COMPUTED_BLR is NULL)
        or      (not F.RDB$DEFAULT_VALUE is NULL))
  Plan: PLAN JOIN (R INDEX (RDB$INDEX_4),F INDEX (RDB$INDEX_2))
: [Fetch] Select ...
: [Execute] Select ...
RELATION = EMPLOYEE
: [Fetch] Select ...
: [Fetch] Select ...
SEOFReached
: [Commit (Hard commit)]
IBDataSet2: [Execute] select * from EMPLOYEE
IBDataSet2: [Fetch] select * from EMPLOYEE
```

Die Formatierung der SQL-Anweisung wurde manuell vorgenommen. Vor der Ausführung des eigentlichen SELECT-Statements wird hier eine Liste derjenigen Spalten erstellt, die berechnet werden oder einen Vorgabewert haben. Diese zusätzliche Abfrage erklärt die längere Ausführungszeit.

Daten aktualisieren

Die Komponente *TIBDataSet* kann auch Daten aktualisieren. Um dies einzurichten, rufen wir den Menüpunkt DATASET-EDITOR aus dem Kontextmenü auf.



Dieser Dialog ist gestalterisch etwas »verunglückt«, sehen wir gnädig darüber hinweg ... Mit Hilfe der Schlüsselfelder wählen wir diejenigen Spalten, die den Primärschlüssel bilden. Mit Hilfe dieser Spalten wird der Datensatz eindeutig identifiziert, wenn er gelöscht, geändert oder aktualisiert werden muss.

Die Aktualisierungsfelder sind diejenigen Spalten, in die im INSERT- oder UPDATE-Fall Werte geschrieben werden. Für *emp_no* gibt es einen Generator und *full_name* ist ein berechnetes Feld, somit sollten Sie nicht versuchen, in diese Spalten etwas zu schreiben.

Klicken Sie nun auf *SQL generieren* und erstellen Sie somit vier SQL-Anweisungen:

```
update EMPLOYEE
set
  FIRST_NAME = :FIRST_NAME,
  LAST_NAME = :LAST_NAME,
    ...
  SALARY = :SALARY
where
  EMP_NO = :OLD_EMP_NO
```

Mit der UPDATE-Anweisung wird in sämtliche Aktualisierungsfelder geschrieben, der Datensatz wird dabei mit dem Schlüsselfeld identifiziert.

```
insert into EMPLOYEE
  (FIRST_NAME, LAST_NAME, ... SALARY)
values
  (:FIRST_NAME, :LAST_NAME, ... :SALARY)
```

Die INSERT-Anweisung schreibt in alle gewählten Aktualisierungsfelder.

```
delete from EMPLOYEE
where
  EMP_NO = :OLD_EMP_NO
```

Die DELETE-Anweisung löscht den Datensatz und identifiziert ihn dabei mit Hilfe der Schlüsselfelder.

```
Select
  EMP_NO,
  FIRST_NAME,
    ...
  FULL_NAME
from EMPLOYEE
where
  EMP_NO = :EMP_NO
```

Muss ein Datensatz aktualisiert werden, dann kann er mit der Eigenschaft *RefreshSQL* gezielt vom Server abgerufen werden.

Das Öffnen der Datenmenge dauert statt 9 nun 15 ms. Ein Blick in das Protokoll zeigt auch, warum das so ist:

```
IBDataSet1: [Prepare]
  select * from EMPLOYEE
  Plan: PLAN (EMPLOYEE NATURAL)
IBDataSet1: [Prepare]
  delete from EMPLOYEE
  where
    EMP_NO = :OLD_EMP_NO
    Plan: PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
IBDataSet1: [Prepare]
  insert into EMPLOYEE
    (FIRST_NAME, LAST_NAME, PHONE_EXT, HIRE_DATE, DEPT_NO, JOB_CODE,
      JOB_GRADE, JOB_COUNTRY, SALARY)
```

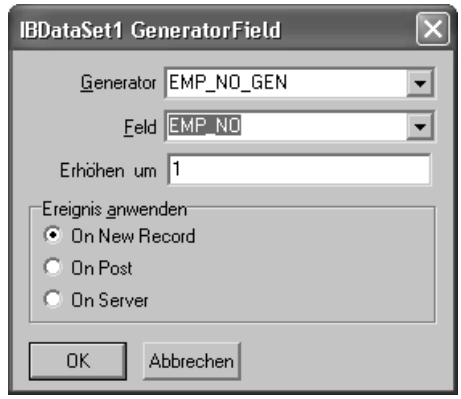
```
  values
    (:FIRST_NAME, :LAST_NAME, :PHONE_EXT, :HIRE_DATE, :DEPT_NO,
      :JOB_CODE, :JOB_GRADE, :JOB_COUNTRY, :SALARY)
  Plan:
IBDataSet1: [Prepare]
  update EMPLOYEE
  set
    FIRST_NAME = :FIRST_NAME
    LAST_NAME = :LAST_NAME,
      ...
    SALARY = :SALARY
  where
  EMP_NO = :OLD_EMP_NO
  Plan: PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
IBDataSet1: [Prepare]
  Select
    EMP_NO,
    FIRST_NAME,
    LAST_NAME,
      ...
    SALARY,
    FULL_NAME
  from EMPLOYEE
  where
    EMP_NO = :EMP_NO
  Plan: PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
: [Start transaction]
: [Prepare]
  Select
      F.RDB$COMPUTED_BLR,
      F.RDB$DEFAULT_VALUE,
      R.RDB$FIELD_NAME
    from RDB$RELATION_FIELDS R, RDB$FIELDS F
    where R.RDB$RELATION_NAME = :RELATION
      and R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
      and ((not F.RDB$COMPUTED_BLR is NULL)
        or      (not F.RDB$DEFAULT_VALUE is NULL))
  Plan: PLAN JOIN (R INDEX (RDB$INDEX_4),F INDEX (RDB$INDEX_2))
: [Fetch] Select ...
: [Execute] Select ...
RELATION = EMPLOYEE
```

```
: [Fetch] Select ...
: [Fetch] Select ...
SEOFReached
: [Commit (Hard commit)]
IBDataSet1: [Execute] select * from EMPLOYEE
IBDataSet1: [Fetch] select * from EMPLOYEE
```

Beim Öffnen der Datenmenge wird nicht nur das SELECT-Statement vorbereitet (also vom Server interpretiert), sondern auch die UPDATE-, INSERT-, DELETE- und Refresh-Anweisung. Verständlicherweise beschleunigt das den Vorgang nicht.

Generatoren

Selbstinkrementierende Felder, die bei InterBase mittels eines Generators erstellt werden, sorgen bei BDE-Client-Server-Anwendungen für vergleichsweise hohen Mehraufwand. Nicht so bei *TIBDataSet*: Mittels des Eigenschaftseditors der Eigenschaft *GeneratorField* wird einfach gewählt, welcher Generator welches Feld setzen soll, und um wie viel er dabei erhöht wird:



14.3 Referenz der InterBase-Komponenten

Die InterBase-Komponenten sind erfreulicherweise stark an die anderen Datenzugriffskomponenten angelehnt. In einigen Fällen erfolgt deshalb nur ein entsprechender Verweis auf diese Komponenten.

14.3.1 TIBDatabase

TIBDatabase stellt die Verbindung zum InterBase-Client her. *TIBDatabase* ähnelt in vielem *TDatabase*.

Verbindung zur Datenbank

- DatabaseName (Eigenschaft, veröffentlicht)

  ```
  property DatabaseName: String;
  ```

  Mittels *DatabaseName* wird die Datenbank angegeben, mit der die Komponente verbunden werden soll.

- Connected (Eigenschaft, veröffentlicht), Open, Close (Methoden)

  ```
  property Connected: Boolean;
  procedure Open;
  procedure Close;
  ```

  Um die Verbindung zur Datenbank herzustellen, setzen Sie *Connected* auf *true* oder rufen *Open* auf. Um die Verbindung zu beenden, wird *Close* aufgerufen oder *Connected* auf *false* gesetzt.

- Params, LoginPrompt (Eigenschaften, veröffentlicht)

  ```
  property Params: TStrings;
  property LoginPrompt: Boolean;
  ```

  Über die Eigenschaft *Params* können Parameter wie der Benutzername oder das Passwort eingestellt werden. Dies ist besonders dann wichtig, wenn kein Anmelde-Dialog angezeigt werden soll und somit die Eigenschaft *LoginPrompt* auf *false* gesetzt wird.

  Beachten Sie bitte die im Vergleich zu *TDatabase* leicht abweichende Schreibweise des Benutzernamens:

  ```
  user_name=SYSDBA
  password=masterkey
  lc_ctype=ISO8859_1
  ```

- Diverse Ereignisse

  ```
  property OnAfterConnect(Sender: TObject);
  property OnAfterDisconnect(Sender: TObject);
  property OnBeforeConnect(Sender: TObject);
  property OnBeforeDisconnect(Sender: TObject);
  ```

  Diese Ereignisse treten vor beziehungsweise nach dem Erstellen oder Trennen einer Datenbankverbindung auf.

- IdleTimer (Eigenschaft, veröffentlicht), OnIdleTimer (Ereignis)

```
property IdleTimer: Integer;
property OnIdleTimer(Sender: TObject);
```

  Mit *IdleTimer* wird die Zeit in Sekunden eingestellt, die auf einen Verbindungsaufbau gewartet wird. Verstreicht diese Zeit, ohne dass eine Verbindung zustande kommt, dann tritt das Ereignis *OnIdleTimer* auf.

- IsReadOnly (Eigenschaft, öffentlich, nur Lesen)

```
property IsReadOnly: Boolean;
```

  Ist die Datenbank schreibgeschützt, dann hat *IsReadOnly* den Wert *true*.

Transaktionen

Im Gegensatz zu *TDatabase* erfolgt mit *TIBDatabase* keine Transaktionssteuerung – dafür ist die Komponente *TIBTransaction* zuständig. Mit einer *TIBDatabase* können mehrere *TIBTransaction*-Komponenten verbunden werden.

- DefaultTransaction (Eigenschaft, veröffentlicht)

```
property DefaultTransaction: TIBTransaction;
```

  Wird bei einer IB-Datenmengenkomponente die Eigenschaft *Database* gesetzt, wird die Eigenschaft *Transaction* automatisch auf die mit *DefaultTransaction* eingestellte *TIBTransaction*-Instanz gesetzt.

- Transactions, TransactionCount (Eigenschaften, öffentlich, nur Lesen)

```
property Transactions [Index: Integer]: TIBTransaction;
property TransactionCount: Integer;
```

  Die Array-Eigenschaft *Transactions* referenziert alle mit der *TIBDatabase*-Komponente verbundenen Transaktionskomponenten. Mit *TransactionCount* kann deren Zahl festgestellt werden.

- AddTransaction, RemoveTransaction, RemoveTransactions, FindTransaction (Methoden)

```
function AddTransaction(TR: TIBTransaction): Integer;
procedure RemoveTransaction(Idx: Integer);
procedure RemoveTransactions;
function FindTransaction (TR: TIBTransaction): Integer;
```

  Mit der Methode *AddTransaction* wird der Datenbankkomponente eine neue Transaktionskomponente hinzugefügt. Als Rückgabewert liefert diese Funktion den Index der neuen Transaktionskomponente in *Transactions*.

Um eine Transaktionskomponente aus der Liste zu entfernen, wird *RemoveTransaction* verwendet – um alle Transaktionskomponenten zu entfernen, *RemoveTransactions*. Um den Index einer Transaktionskomponente zu ermitteln, wird *FindTransaction* verwendet.

Informationen über die Datenbank

- GetTableNames, GetFieldNames (Methoden)

```
procedure GetTableNames(List: TStrings;
  SystemTables: Boolean = False);
procedure GetFieldNames(const TableName: string; List: TStrings);
```

GetTableNames ermittelt die Namen aller Tabellen in der Datenbank. Im Gegensatz zu *TDatabase* fehlen hier die Namen der VIEWS. Wird *SystemTables* auf *true* gesetzt, dann werden auch die Systemtabellen mit angegeben.

```
IBDatabase1.GetTableNames(Form1.ListBox1.Items);
```

Mit *GetFieldNames* werden die Spaltennamen der angegebenen Tabelle ermittelt.

- TraceFlags (Eigenschaft)

```
property TraceFlags: TTraceFlags;
```

Mittels *TIBSQLMonitor* können die SQL-Anweisungen protokolliert werden, die zum Server geschickt werden. Über die Eigenschaft *TraceFlags* wird dabei spezifiziert, welche Anweisungen dabei beachtet werden sollen.

Die Bedeutung der einzelnen Flags entnehmen Sie bitte der Online-Hilfe.

14.3.2 TIBTransaction

Die Komponente *TIBTransaction* implementiert die Transaktionsverwaltung. Für den Zugriff auf eine InterBase-Datenbank benötigen Sie mindestens eine *TIBDatabase*- und eine *TIBTransaction*-Komponente.

Transaktionsverwaltung

- StartTransaction, Commit, CommitRetaining, Rollback, RollbackRetaining (Methoden)

```
procedure StartTransaction;
procedure Commit;
procedure CommitRetaining;
procedure Rollback;
procedure RollbackRetaining;
```

Mit *StartTransaction* wird eine Transaktion neu gestartet. Mit *Commit* werden alle Änderungen bestätigt und die Transaktion wird abgeschlossen, während *Rollback* alle Änderungen verwirft, bevor die Transaktion abgeschlossen wird.

Mit *CommitRetaining* werden Änderungen bestätigt, die Transaktion aber nicht abgeschlossen. Ebenso werden mit *RollbackRetaining* Änderungen verworfen, ohne die Transaktion zu beenden.

- Active (Eigenschaft, veröffentlicht)

```
property Active: Boolean;
```

Mit *Active* kann zur Entwurfszeit eine Transaktion begonnen und somit mit Live-Daten gearbeitet werden. Wird *Active* auf *false* gesetzt, dann wird *Rollback* aufgerufen.

- InTransaction (Eigenschaft, öffentlich, nur Lesen)

```
property InTransaction: Boolean;
```

Mit *InTransaction* kann geprüft werden, ob momentan eine Transaktion gestartet ist.

Datenbanken

- DefaultDatabase (Eigenschaft, veröffentlicht)

```
property DefaultDatabase: TIBDatabase;
```

Mit *DefaultDatabase* wird die »hauptsächlich eingesetzte« (in den meisten Fällen ausschließlich verwendete) Datenbank angegeben.

- Databases, DatabaseCount (Eigenschaften, nur Lesen)

```
property Databases[Index: Integer]: TIBDatabase;
property DatabaseCount: Integer;
```

Die mit der Transaktionskomponente verbundenen Datenbankkomponenten können mit der Array-Eigenschaft *Databases* referenziert werden. Die Anzahl dieser Datenbankkomponenten kann mit *DatabaseCount* ermittelt werden.

- AddDatabase, RemoveDatabase, RemoveDatabases, FindDatabase (Methoden)

```
function AddDatabase(db: TIBDatabase): Integer;
procedure RemoveDatabase(Idx: Integer);
procedure RemoveDatabases;
function FindDatabase (db: TIBDatabase): Integer;
```

Um der Liste der Datenbanken eine neue hinzuzufügen, wird *AddDatabase* verwendet, um eine aus der Liste zu entfernen, *RemoveDatabase*. *RemoveDatabases* entfernt alle Datenbanken aus der Liste und *FindDatabase* ermittelt den Index einer Datenbankkomponente.

14.3.3 TIBCustomDataSet

Die Komponente *TIBCustomDataSet* ist der gemeinsame Vorfahre von *TIBTable*, *TIBQuery*, *TIBStoredProc* und *TIBDataSet*.

TIBCustomDataSet ist von der in Kapitel 3.1 beschriebenen Komponente *TDataSet* abgeleitet und erbt eine Reihe von Eigenschaften, Methoden und Ereignissen, die hier nicht noch einmal erläutert werden sollen.

Verbindung zur Datenbank

- Database, Transaction (Eigenschaften, veröffentlicht)

```
property Database: TIBDatabase;
property Transaction: TIBTransaction;
```

Mit *Database* wird die *TIBDatabase*-Komponente ausgewählt, über welche die Verbindung zur Datenbank hergestellt wird. Mit *Transaction* wird die zuständige Transaktionskomponente eingestellt.

Cached Updates

- CachedUpdates, UpdateObject (Eigenschaften, veröffentlicht)

```
property CachedUpdates: Boolean;
property UpdateObject: TIBDataSetUpdateObject;
```

Wird *CachedUpdates* auf *true* gestellt, dann werden zwischengespeicherte Aktualisierungen verwendet. Soll eine *TIBUpdateObject*-Komponente verwendet werden, so ist diese mit der Eigenschaft *UpdateObject* einzustellen.

- ApplyUpdates, CancelUpdates, RevertRecord (Methoden)

```
procedure ApplyUpdates;
procedure CancelUpdates;
procedure RevertRecord;
```

Mit *ApplyUpdates* werden die Änderungen in die Datenbank geschrieben, mit *CancelUpdates* verworfen. Beachten Sie, dass die Daten nur dann wirklich in die Datenbank übernommen werden, wenn die Transaktion mit COMMIT abgeschlossen wird.

Mit *RevertRecord* werden die Änderungen am aktuellen Datensatz rückgängig gemacht. Das ist möglich, solange zwischengespeicherte Aktualisierungen verwendet werden und der Datensatz noch nicht zum Server geschickt wurde.

- UpdatesPending (Eigenschaft, öffentlich, nur Lesen)

```
property UpdatesPending: Boolean;
```

Gibt es zwischengespeicherte Aktualisierungen, die noch nicht zum Server übertragen wurden, dann hat *UpdatesPending* den Wert *true*.

- UpdateRecordTypes (Eigenschaft, öffentlich)

```
property UpdateRecordTypes: TIBUpdateRecordTypes;
```

Mit *UpdateRecordTypes* wird spezifiziert, welche Datensätze momentan sichtbar sind. Funktioniert nur bei der Verwendung von zwischengespeicherten Aktualisierungen.

- OnUpdateRecord, OnUpdateError (Ereignisse)

```
property OnUpdateRecord(DataSet: TDataSet; UpdateKind:
TUpdateKind;     var UpdateAction: TIBUpdateAction);
property OnUpdateError(DataSet: TDataSet; E: EDatabaseError;
    UpdateKind: TUpdateKind; var UpdateAction: TIBUpdateAction);
```

Das Ereignis *OnUpdateRecord* tritt auf, wenn ein Datensatz von den zwischengespeicherten Aktualisierungen auf den Server übertragen wird, *OnUpdateError*, wenn dabei ein Fehler auftritt.

Der Parameter *UpdateKind* informiert darüber, ob das Ereignis beim Ändern (*ukModify*), Einfügen (*ukInsert*) oder Löschen (*ukDelete*) aufgetreten ist.

Mit Hilfe des Variablenparameters *UpdateAction* kann bestimmt werden, welche Aktion ausgeführt werden soll – beispielsweise, auf welche Art und Weise der Fehler behandelt werden soll. Der Parameter *UpdateAction* kann folgende Werte annehmen:

- *uaApplied:* Die Änderung wird eingetragen und aus dem Zwischenspeicher gelöscht.
- *uaAbort:* Die Aktualisierung wird ohne Fehlermeldung abgebrochen.
- *uaFail:* Die Aktualisierung wird mit Fehlermeldung abgebrochen.
- *uaRetry:* Es wird erneut versucht, die Änderung einzutragen. Zuvor sollte der Fehler möglichst behoben werden. (Tritt nur bei *OnUpdateError* auf.)
- *uaSkip:* Die Aktualisierung wird übergangen, die entsprechenden Daten bleiben im Zwischenspeicher.

- FetchAll (Methode, TBDEDataSet)

  ```
  procedure FetchAll;
  ```

  Normalerweise fordert die Datenmengenkomponente nur diejenigen Daten aus der Datenbank an, welche momentan zur Anzeige benötigt werden. Scrollt der Anwender beispielsweise durch eine Datenmenge, dann werden laufend neue Datensätze angefordert, was die Netzwerkbelastung erhöht.

  Mit der Methode *FetchAll* kann man alle Datensätze von der Cursor-Position ausgehend bis zum Dateiende der Datenbank lesen und lokal zwischenspeichern.

14.3.4 TIBTable

Die Komponente *TIBTable* dient zum Zugriff auf eine einzelne Tabelle. Die Komponente ist von *TIBCustomDataSet* abgeleitet und ähnelt *TTable*.

Verbindung zur Tabelle

- TableName, TableTypes (Eigenschaften, veröffentlicht)

  ```
  property TableName: string;
  property TableTypes: set of (ttSystem, ttView);
  ```

  Mit der Eigenschaft *TableName* wird der Name derjenigen Tabelle oder VIEW angegeben, die *TTable* öffnen soll.

  Im Objektinspektor existiert für die Eigenschaft *TableName* eine ComboBox, mit der aus den vorhandenen Tabellen ausgewählt werden kann. Sollen dabei auch die VIEWS angezeigt werden, dann ist *ttView* in die Menge *TableTypes* aufzunehmen. Ebenso ist mit *ttSystem* zu verfahren, wenn Systemtabellen angezeigt werden sollen.

- CreateTable, DeleteTable (Methoden)

  ```
  procedure CreateTable;
  procedure DeleteTable;
  ```

  Mit diesen Methoden können zur Laufzeit Tabellen erstellt und gelöscht werden. Ein Beispiel hierzu finden Sie in Kapitel 2.3.

- Exists (Eigenschaft, öffentlich, nur Lesen)

  ```
  property Exists: Boolean;
  ```

  Wenn die angegebene Tabelle nicht (mehr) existiert, hat *Exists* den Wert *false*.

- ReadOnly (Eigenschaft, veröffentlicht)

  ```
  property ReadOnly: Boolean;
  ```

  Wird *ReadOnly* auf *true* gesetzt, kann die *TTable*-Instanz keine Änderungen am Datenbestand vornehmen.

- EmptyTable (Methode)

  ```
  procedure EmptyTable;
  ```

  Mit dieser Methode werden alle Datensätze aus der Tabelle gelöscht, die Tabellenstruktur wird jedoch beibehalten.

- GotoCurrent (Methode)

  ```
  procedure GotoCurrent(Table: TIBTable);
  ```

  Die Methode *GotoCurrent* setzt den Datenzeiger auf denselben Datensatz wie den Datenzeiger der als Parameter übergebenen Tabelle. Auf diese Weise lassen sich sehr einfach zwei *TIBTable*-Instanzen synchronisieren.

Indizes

Viele Operationen bei *TTable* (Bereiche, Suche nach Schlüsseln) sind nur dann möglich, wenn die verwendeten Felder indiziert sind.

- DefaultIndex (Eigenschaft, veröffentlicht)

  ```
  property DefaultIndex: Boolean;
  ```

  Hat *DefaultIndex* den Wert *true*, dann werden die Daten entsprechend dem Primärschlüssel oder – falls ein solcher nicht existiert – nach dem Index sortiert, den die Komponente für den ersten hält.

- IndexName (Eigenschaft, veröffentlicht)

  ```
  property IndexName: string;
  ```

  Die Eigenschaft *IndexName* gibt den Namen des Sekundärindex an, nach dem die Datenmenge sortiert wird. Bleibt die Eigenschaft leer, dann wird nach dem Primärindex sortiert.

- IndexFieldNames (Eigenschaft, veröffentlicht)

  ```
  property IndexFieldNames: string;
  ```

  Alternativ zu *IndexName* können mit *IndexFieldNames* die einzelnen Spalten angegeben werden, aus denen ein Index gebildet werden soll.

- IndexFields (Eigenschaften, öffentlich)

```
property IndexFields: [Index: Integer]: TField;
property IndexFieldCount: Integer;
```

  Die an einem Index beteiligten Spalten werden mit *IndexFields* ermittelt, die Anzahl dieser Spalten mit *IndexFieldCount*.

- GetIndexNames (Methoden)

```
procedure GetIndexNames(List: TStrings);
```

  Mit *GetIndexNames* werden die Namen aller vorhandenen Indizes abgefragt und in eine Liste geschrieben.

- IndexDefs (Eigenschaft, öffentlich)

```
property IndexDefs: TIndexDefs;
```

  Um Informationen über die Indizes zu erhalten, wird die Eigenschaft *IndexDefs* verwendet. Ein Beispiel dafür, wie mit Hilfe dieser Eigenschaft beim Erstellen einer Tabelle ein Index erstellt wird, ist in Kapitel 2.3 zu finden. Entgegen den Angaben in der Online-Hilfe ist diese Eigenschaft keine Read-only-Eigenschaft.

Master-Detail-Verknüpfungen

- MasterSource (Eigenschaft, veröffentlicht)

```
property MasterSource: TDataSource;
```

  Mit der Eigenschaft *MasterSource* wird angegeben, welche Datenquelle diese Datenmenge als Master-Tabelle steuert.

- MasterFields (Eigenschaft, veröffentlicht)

```
property MasterFields: string;
```

  Der Eigenschaft *MasterFields* werden diejenigen Feldnamen zugewiesen, über die bei der Master-Tabelle die Verknüpfung gesteuert wird. Mehrere Feldnamen werden dabei mit Semikola gestrennt. Im Gegensatz zu Desktop-Datenbanken müssen diese Felder in Anzahl und Feldart dem momentan verwendeten Schlüssel der Detailtabelle nicht (!) entsprechen.

  Zur Entwurfszeit steht Ihnen zum Setzen dieser Eigenschaft der Feldverbindungs-Designer zur Verfügung. Mehr zum Thema Master-Detail-Verknüpfung finden Sie in Kapitel 2.2.

14.3.5 TIBQuery

Die Komponente *TIBQuery* dient dem Zugriff auf die Datenbank mittels einer SQL-Anweisung. Dabei sind sowohl Abfragen als auch Datenänderungen oder die Definition von Metadaten möglich. Die Komponente ähnelt *TQuery*.

Die SQL-Anweisung

- SQL (Eigenschaft, veröffentlicht)

  ```
  property SQL: TStrings;
  ```

  Die Eigenschaft *SQL* erhält die SQL-Anweisung für die Abfrage oder die Datenmanipulation. Eine Abfrage wird mit *Open* geöffnet, eine Datenmanipulation mit *ExecSQL* ausgeführt.

  Die Datenbanksprache *SQL* wird in Kapitel 3.2 beschrieben. Den vollen InterBase-Sprachumfang finden Sie in der InterBase-Dokumentation oder im Buch *SQL lernen* (Michael Ebner, Addison-Wesley).

- ExecSQL (Methode)

  ```
  procedure ExecSQL;
  ```

  Wird der Eigenschaft *SQL* eine Anweisung zur Datenmanipulation zugewiesen (Erstellen von Tabellen, Ändern von Daten), dann kann *TQuery* keine Datenmenge öffnen, und deshalb darf weder die Methode *Open* verwendet noch die Eigenschaft *Active* auf *true* gesetzt werden. Um eine solche SQL-Anweisung auszuführen, muss die Methode *ExecSQL* eingesetzt werden.

- RowsAffected (Eigenschaft, öffentlich, nur Lesen)

  ```
  property RowsAffected: Integer;
  ```

  Mit *RowsAffected* kann ermittelt werden, wie viele Datensätze bei der letzten Anweisung geändert oder gelöscht wurden. Bei Anweisungen, die keine Daten ändern können (SELECT beispielsweise) wird -1 zurückgegeben.

- Text (Eigenschaft, öffentlich, nur Lesen)

  ```
  property Text: PChar;
  ```

  Die Eigenschaft *Text* gibt an, welche SQL-Anweisung zuletzt an die BDE oder an den Server geschickt wurde. Die entsprechenden Werte für die Parameter sind dabei schon eingefügt, so dass diese Eigenschaft eine Hilfe bei der Fehlersuche darstellt, wenn bei der Verwendung von Parametern nicht alles wie geplant läuft.

Parameter

- Params (Eigenschaft, veröffentlicht)

  ```
  property Params[Index: Word]: TParams;
  ```

  Mit der Eigenschaft *Params* kann auf die Parameter einer Abfrage zugegriffen werden. Um Verwechslungen zu vermeiden, sollten Sie dafür jedoch die Methode *ParamByName* bevorzugen.

- ParamByName (Methode)

  ```
  function ParamByName(const Value: string): TParam;
  ```

  Statt über den Index – wie bei der Eigenschaft Params – können Parameter auch über den Parameternamen gesetzt werden. Dazu dient die Methode *ParamByName*.

- DataSource (Eigenschaft, veröffentlicht)

  ```
  property DataSource: TDataSource;
  ```

  Die Eigenschaft *DataSource* wird benötigt, um Master-Detail-Beziehungen mit Hilfe von *TQuery*-Komponenten herzustellen. Näheres siehe Beispiel.

- ParamCount (Eigenschaft, öffentlich, nur Lesen)

  ```
  property ParamCount: Word;
  ```

  Mit *ParamCount* kann die Anzahl der vorhandenen Parameter abgefragt werden.

14.3.6 TIBStoredProc

Die Komponente *TIBStoredProc* dient zum Zugriff auf STORED PROCEDURES und ähnelt *TStoredProc*. Sie sollten nur auf auszuführende Prozeduren mit dieser Komponente zugreifen. Für Prozeduren, die eine Ergebnismenge liefern, verwenden Sie *TIBQuery* oder *TIBDataSet*.

- StoredProcName (Eigenschaft, veröffentlicht)

  ```
  property StoredProcName: String;
  ```

  Mit dieser Eigenschaft wird die auszuführende Prozedur ausgewählt.

- ExecProc (Methode)

  ```
  procedure ExecProc;
  ```

  Mit *ExecProc* wird die Prozedur ausgeführt.

- Params (Eigenschaft, veröffentlicht), ParamByName (Methode)

```
property Params[Index: Word]: TParams;
function ParamByName(const Value: string): TParam;
```

  Mit der Eigenschaft *Params* kann auf die Parameter einer Abfrage zugegriffen werden. Um Verwechslungen zu vermeiden, sollten Sie dafür jedoch die Methode *ParamByName* bevorzugen.

- ParamCount (Eigenschaft, öffentlich, nur Lesen)

```
property ParamCount: Word;
```

  Mit *ParamCount* kann die Anzahl der vorhandenen Parameter abgefragt werden.

14.3.7 TIBUpdateSQL

Die Komponente *TIBUpdateSQL* dient dazu, Abfragen aktualisierbar zu machen, die für gewöhnlich keine aktualisierbare Ergebnismenge liefern – beispielsweise JOINS. Diese Komponente ähnelt der in Kapitel 6.4 beschriebenen Komponente *TUpdateSQL*.

- DeleteSQL, InsertSQL, ModifySQL, RefreshSQL (Eigenschaften, veröffentlicht)

```
property DeleteSQL: TStrings;
property InsertSQL: TStrings;
property ModifySQL: TStrings;
property RefreshSQL: TStrings;
```

  Mit diesen vier Eigenschaften werden die SQL-Anweisungen vorgegeben, die beim Löschen, Einfügen oder Ändern von Datensätzen beziehungsweise beim Aktualisieren der Ergebnismenge ausgeführt werden.

- Query (Eigenschaft, öffentlich, nur Lesen)

```
property Query[UpdateKind: TUpdateKind]: TQuery;
```

  Die Komponente *TUpdateObject* verwaltet drei *TQuery*-Komponenten, welche die Aufgaben des Löschens, des Einfügens und des Änderns von Datensätzen zwischen sich aufteilen. Mit Hilfe der Eigenschaften *DeleteSQL*, *InsertSQL* und *ModifySQL* kann auf die Eigenschaft *SQL* dieser drei *TQuery*-Instanzen zugegriffen werden. Sollen andere Eigenschaften, Methoden oder Ereignisse verwendet werden, dann ist die Array-Eigenschaft *Query* zu verwenden.

- DataSet (Eigenschaft, öffentlich, nur Lesen)

  ```
  property DataSet: TDataSet;
  ```

  Die Eigenschaft *DataSet* gibt die Datenmenge an, mit der die Komponente verbunden ist. Normalerweise wird die Eigenschaft zur Entwurfszeit automatisch gesetzt.

- SetParams, ExecSQL, Apply (Methoden)

  ```
  procedure SetParams(UpdateKind: TUpdateKind);
  procedure ExecSQL(UpdateKind: TUpdateKind);
  procedure Apply(UpdateKind: TUpdateKind);
  ```

  Für gewöhnlich löst die über *DataSet* verbundene Datenmengenkomponente automatisch die Aktualisierung aus. Muss dies – aus was für Gründen auch immer – manuell geschehen, dann werden mit *SetParams* die Parameter gesetzt, mit *ExecSQL* wird anschließend die Aktualisierung vorgenommen. Mit *Apply* werden diese beiden Methoden hintereinander abgearbeitet.

14.3.8 TIBDataSet

Die Komponente *TIBDataSet* erlaubt es, für alle möglichen Situationen eigene SQL-Anweisungen zu verfassen. Diese Komponente verhält sich in etwa wie eine Kombination aus *TIBQuery* und *TIBUpdateSQL*, ohne jedoch die Änderungen zwischenzuspeichern.

- Database, Transaction (Eigenschaften, veröffentlicht)

  ```
  property Database: TIBDatabase;
  property Transaction: TIBTransaction;
  ```

  Mittels dieser beiden Eigenschaften wird die Komponente mit einer Datenbank- und einer Transaktionskomponente verbunden.

- SelectSQL, RefreshSQL, InsertSQL, UpdateSQL, DeleteSQL (Eigenschaften, veröffentlicht)

  ```
  property SelectSQL: TStrings;
  property RefreshSQL: TStrings;
  property InsertSQL: TStrings;
  property UpdateSQL: TStrings;
  property DeleteSQL: TStrings;
  ```

  Mittels dieser Eigenschaften werden die einzelnen SQL-Statements zugewiesen. Mehr dazu im Beispiel.

- QSelect, QRefresh, QInsert, QUpdate, QDelete
  (Eigenschaften, öffentlich, nur Lesen)

```
property QSelect: TIBSQL;
property QRefresh: TIBSQL;
property QInsert: TIBSQL;
property QUpdate: TIBSQL;
property QDelete: TIBSQL;
```

  Die Statements werden mittels einzelner *TIBSQL*-Komponenten abgesetzt (siehe Kapitel 14.3.9). Mittels dieser Eigenschaften erhält man einen Zeiger auf diese Komponenten.

14.3.9 TIBSQL

Die Komponente TIBSQL ist die spartanischte Möglichkeit, auf InterBase zuzugreifen. In der Regel dürfte man diese Komponente für DML- und DDL-Anweisungen nutzen, es können darüber aber auch SELECT-Statements gefahren werden.

Interessant ist diese Komponente vor allem dann, wenn kleine Tools für Spezialaufgaben in der Dateigröße minimiert werden müssen.

- Database, Transaction (Eigenschaften, veröffentlicht)

```
property Database: TIBDatabase;
property Transaction: TIBTransaction;
```

  Mittels dieser beiden Eigenschaften wird die Komponente mit einer Datenbank- und einer Transaktionskomponente verbunden.

- SQL (Eigenschaft, veröffentlicht), ExecQuery, Close (Methoden)

```
property SQL: TStrings;
procedure ExecQuery;
procedure Close;
```

  Mit der Eigenschaft SQL wird das auszuführende Statement zugewiesen, mit *ExecSQL* wird es ausgeführt. Liefert das Statement eine Ergebnismenge, dann kann diese mit *Close* geschlossen werden.

- Params (Eigenschaft, öffentlich, nur Lesen)

```
property Params: TIBXSQLDA;
```

  Einen Zeiger auf die *TIBXSQLDA*-Instanz liefert die Eigenschaft *Params*. Mehr dazu in der Online-Hilfe.

- BOF, EOF (Eigenschaften, öffentlich, nur Lesen), Next (Methode)

```
property BOF: Boolean;
property EOF: Boolean;
procedure Next;
```

TIBSQL stellt nur eindirektionale Cursor zur Verfügung, es gibt somit auch nur eine Richtung, in der Datenmenge zu navigieren. Mit *EOF* kann man ermitteln, ob man das Ende der Datenmenge erreicht hat.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with IBSQL1 do
  begin
    ExecQuery;
    while not EOF do
    begin
      Memo1.Lines.Add(FieldByName('Nachname').AsString);
      Next;
    end;
    Close;
  end; {with IBSQL1 do}
end; {procedure TForm1.Button1Click}
```

- Fields (Eigenschaft, öffentlich, nur Lesen), FieldByName (Methode)

```
property Fields[const Idx: Integer]: TIBXSQLVAR;
function FieldByName[FieldName: String]: TIBXSQLVAR;
```

Mit *Fields* beziehungsweise *FieldByName* kann man auf die einzelnen Spalten einer Ergebnismenge zugreifen. Der Typ *TIBXSQLVAR* ähnelt weitgehend *TField*, so finden Sie beispielsweise auch hier Eigenschaften wie *AsString* oder *AsInteger*.

14.3.10 TIBDatabaseInfo

Mittels *TIBDatabaseInfo* erhalten Sie Informationen über die Datenbank. Diese Informationen werden Ihnen von öffentlichen Nur-Lesen-Eigenschaften zur Verfügung gestellt, die hier nicht weiter beschrieben werden sollen – ziehen Sie hierzu die Online-Hilfe zu Rate.

Die Namen der momentan angemeldeten Benutzer erhalten Sie mit *UserNames*. Interessant ist auch der Vergleich zwischen den Lesevorgängen insgesamt (*Reads*) und den Lesevorgängen, die über einen Index laufen (*ReadIdxCount*).

- Database (Eigenschaft, veröffentlicht)

  ```
  property Database: TIBDatabase;
  ```

  Über diese Eigenschaft verbinden Sie die Komponente mit der Datenbankkomponente. Eine Transaktionskomponente ist hier nicht erforderlich.

14.3.11 TIBMonitor

Mit der Komponente *TIBMonitor* können Sie mitverfolgen, welche Informationen mit dem InterBase-Client ausgetauscht werden. Vergessen Sie nicht, dazu die Eigenschaft *TraceFlags* von *TIBDatabase* entsprechend zu setzen.

- OnSQL (Ereignis)

  ```
  property OnSQL(EventText: String);
  ```

  Das Ereignis *OnSQL* tritt dann auf, wenn eine SQL-Anweisung zum InterBase-Client geschickt wird.

  ```
  procedure TForm1.IBSQLMonitor1SQL(EventText: String);
  begin
    Memo1.Lines.Add(EventText);
  end;
  ```

14.3.12 TIBEvents

Die Komponente *TIBEvents* setzt *InterBase*-Events in Delphi-Ereignisse um.

- Database (Eigenschaft, veröffentlicht)

  ```
  property Database: TIBDatabase;
  ```

  Über diese Eigenschaft verbinden Sie die Komponente mit der Datenbank-Komponente. Eine Transaktionskomponente ist hier nicht erforderlich.

- Events (Eigenschaft, veröffentlicht)

  ```
  property Events: TStrings;
  ```

  In der Eigenschaft *Events* zählen Sie die Events auf, auf welche die Komponente reagieren soll. Eine einzelne Komponente kann auf bis zu 15 Events reagieren.

- OnEventAlert (Ereignis)

```
property OnEventAlert(Sender: TObject; EventName: String;
  EventCount: Integer; var CancelAlerts: Boolean);
```

  Das Ereignis *OnEventAlert* tritt ein, wenn InterBase ein Event auslöst.

- Registered (Eigenschaft, veröffentlicht)

```
property Registered: Boolean;
```

  Die Komponente reagiert nur dann auf Events, wenn *Registered* den Wert *true* hat.

15 dbExpress

Mit den Komponenten von *dbExpress* können Sie auf verschiedene SQL-Server zugreifen:

- In der Professional-Version haben Sie die DLLs für den Zugriff auf *InterBase* und *MySQL*.
- In der Enterprise-Version können Sie zusätzlich noch auf *Oracle* und *DB2* zugreifen.

Im Gegensatz zur BDE ist *dbExpress* für Cross-Plattform-Entwicklung geeignet – die Komponenten gibt es nicht nur bei der VCL, sondern auch bei CLX, und somit können Sie aus demselben Code Windows- und Linux-Anwendungen erstellen. Bei der Programmierung der Treiber hat sich Borland folglich auch auf die Datenbanksysteme beschränkt, die sowohl unter Windows als auch unter Linux verfügbar sind.

Einfache Installation

dbExpress zeichnet sich durch eine einfache Installation aus:

- Wenn im Betrieb zwischen verschiedenen Datenbanksystemen hin- und hergeschaltet werden muss oder während der Entwicklung noch nicht feststeht, welches Datenbanksystem verwendet werden soll, dann müssen DLLs für alle in Frage kommenden Datenbanksysteme installiert werden, darüber hinaus noch zwei Ini-Dateien.
- In der Regel wird eine Client-Server-Datenbankanwendung nur mit einem Datenbanksystem zusammenarbeiten und dies ist auch schon während der Entwicklung bekannt. In diesem Fall muss nur eine DLL an eine Stelle kopiert werden, an der sie vom Programm gefunden wird – am einfachsten in das Verzeichnis, in dem auch die *exe*-Datei liegt.
- In jedem Fall muss der Client der verwendeten Datenbanksysteme auf den Rechnern installiert sein.
- Die Möglichkeit, auf die externe DLL zu verzichten und stattdessen eine Unit einzubinden, ist vorgesehen, führt aber bei mir auf dem Rechner zu einem *Internen Fehler* bei der Kompilierung.

Der langen Rede kurzer Sinn: Die Installation beschränkt sich in der Regel darauf, eine DLL mitzukopieren.

15.1 Erste Schritte mit dbExpress

dbExpress stellt unidirektionale, nicht editierbare Datenmengen zur Verfügung – für Entwickler, die bislang mit der BDE gearbeitet haben, ist dies eine ziemliche Umstellung. Wir werden uns nun ansehen, was man mit dbExpress tun darf und was nicht.

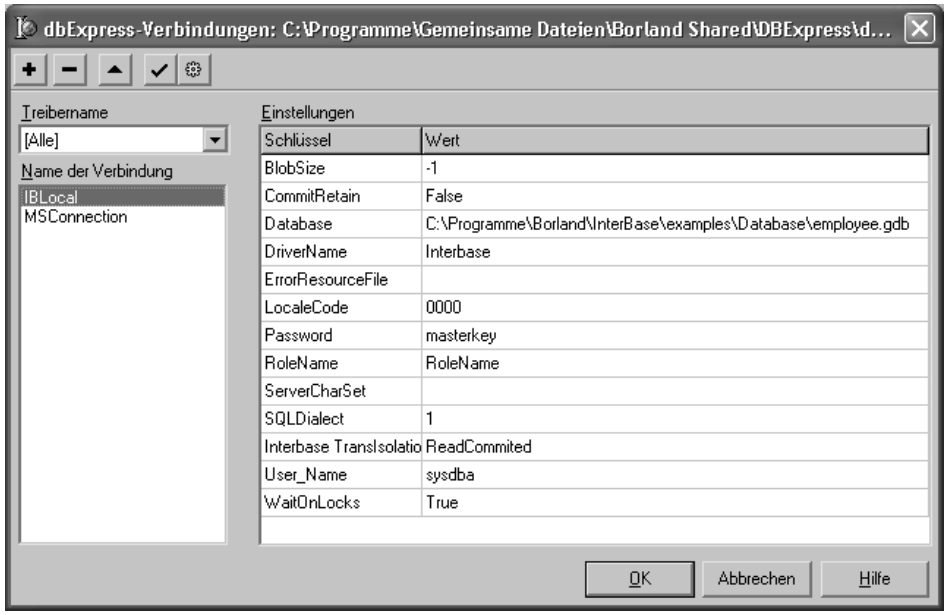
Damit alle Leser dieses Beispiel nachvollziehen können, arbeiten wir mit der Professional-Version und InterBase.

15.1.1 Zugriff auf eine Tabelle

Für unser erstes Beispiel wollen wir auf die Tabelle *projects* aus der Datenbank *employee.gdb* zugreifen.

Einrichten der Datenbankverbindung

Zunächst stellen wir eine Verbindung zur Datenbank her. Dazu legen wir die Komponente *TSQLConnection* auf ein Formular und rufen den Komponenteneditor auf:

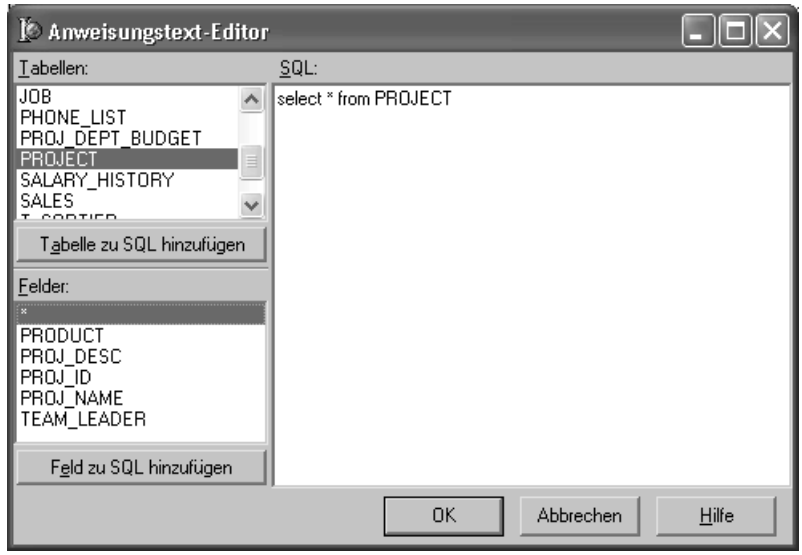


Hier stellen wir unter *Database* den Dateinamen der Datenbank inklusive Pfad ein. Des Weiteren soll der Benutzername und das Password angegeben werden.

Nun schließen wir den Komponenteneditor. Um den lästigen Login-Dialogen zu entgehen, stellen wir *LoginPrompt* auf *false*. Anschließend stellen wir die Verbindung dadurch her, dass *Connected* auf *true* gesetzt wird.

Zugriff auf die Tabelle

Für den Zugriff auf die Tabelle nehmen wir *TSQLDataSet*. Zunächst setzen wir *SQLConnection* auf *SQLConnection1*. Die Eigenschaft *CommandType* belassen wir auf *ctQuery* und öffnen dann den Eigenschaftseditor von *CommandText*:



Hier greifen wir nun zunächst auf die komplette Tabelle *project* zu.

An die Komponente *SQLDataSet1* hängen wir eine *TDataSource*-Instanz, daran wieder eine *TDBNavigator*- und zwei *TDBText*-Komponenten – auf *TDBGrid* soll zunächst verzichtet werden.

Messung der Ausführungsgeschwindigkeit

Nun wollen wir wissen, wie schnell die Abfrage ausgeführt wird, und natürlich auch, wie schnell zum nächsten Datensatz gescrollt wird. Mit diesen Messungen wollen wir *TSQLDataSet* und *TSQLQuery* vergleichen, des Weiteren die BDE-Komponente *TQuery*.

```
procedure TForm1.btnOpenClick(Sender: TObject);
var
  c, t1, t2: int64;
```

```
begin
  QueryPerformanceFrequency(c);
  QueryPerformanceCounter(t1);
  SQLDataSet1.Open;
  QueryPerformanceCounter(t2);
  Label2.Caption := IntToStr(1000000 * (t2 - t1) div c) + ' µs';
end;
```

Mit der Windows-API-Funktion *QueryPerformanceCounter* kann man sehr genaue Zeitmessungen durchführen, so dass wir hier das Ergebnis auf die µs (eine millionstel Sekunde) genau ausgeben können. Das Zeichen µ können Sie in Delphi nicht eingeben, zumindest nicht bei der Standard-Tastaturbelegung. Erzeugen Sie es in einem anderen Programm und kopieren Sie es dann über die Zwischenablage.

Um den Einfluss externer Faktoren zu minimieren, wurden Client und Server auf demselben Rechner ausgeführt. Damit die Daten bereits im Cache liegen – schließlich wollen wir die Verbindung prüfen und nicht den Server –, wurde zunächst eine Anwendung einmal gestartet, die Abfrage geöffnet und ein paar Scroll-Vorgänge durchgeführt. Anschließend wurde die Anwendung geschlossen, erneut gestartet und die dann gemessenen Werte sind die hier veröffentlichten.

Der erste Scroll-Vorgang nach dem Aufruf von *Open* dauert vergleichsweise lange (beispielsweise 757 µs), die dann folgenden Aufrufe von *Next* werden in etwa alle gleich schnell ausgeführt (bei diesem Beispiel etwa 115 µs). Der erste Scroll-Vorgang, der nach dem letzten Datensatz erfolgt, dauert dann wieder lange (780 µs), anschließend weiß das System, dass es nichts mehr zu Scrollen gibt, der Aufruf von Next ist dann in 8 µs erledigt. Die hier veröffentlichten Zahlen sind ein Mittelwert des zweiten, dritten und vierten Scroll-Vorgangs (115 µs).

Bei der Abfrage über alle vorhandenen Spalten dauerte das Öffnen 27 168 µs, das Scrollen 115 µs. Würde *TSQLQuery* anstelle von *TSQLDataSet* verwendet, dann würde das Öffnen bei 27 397 µs liegen und das Scrollen ebenfalls bei 115 µs – die Unterschiede sind geringer als die Streuung der Messwerte, ich habe jedoch den Eindruck gewonnen, dass *TSQLQuery* stets einen Hauch länger braucht.

Würden wir nun mit *TQuery* vergleichen, dann dauert dort das Öffnen 4 323 µs und das Scrollen 349 µs. Das heißt im Klartext, dass *TQuery* das Ergebnis der Abfrage etwa sechsmal so schnell liefert wie *TSQLDataSet* und *TSQLQuery* – dafür geht bei Letzteren das Scrollen etwa dreimal schneller.

Der erste Gedanke könnte nun sein, dass durch das Jokerzeichen * in der SQL-Anweisung die Memo-Spalte mit in die Abfrage aufgenommen wird – dies ist zwar nicht der Fall, aber wir wollen es sicherheitshalber ausschließen:

```
SELECT proj_id, proj_name
  FROM project
```

Einen Hauch schneller wird das Öffnen der Abfrage dadurch (26 862 statt 27 168 µs), und auch das Scrollen geht etwas schneller (110 statt 115 µs), aber ein großer Fortschritt ist auch das nicht.

Persistente TField-Instanzen anlegen

Nun legen wir persistente TField-Instanzen an. Dabei ist es egal, ob wir *TSQLDataSet* oder *TSQLQuery* verwenden, und ob wir die benötigten Spalten einzeln aufzählen oder wir im SELECT-Statement das Jokerzeichen verwenden – der Beschleunigungseffekt ist in etwa derselbe: Statt 26 862 µs dauert das Öffnen der Abfrage nun nur noch 2 884 µs.

Bei *TQuery* fällt die Beschleunigung deutlich geringer aus, die Abfrage ist in 4 273 µs geöffnet.

Wir wollen nun der Frage nachgehen, warum das Anlegen persistenter TField-Instanzen das Öffnen der Datenmenge um etwa den Faktor neun beschleunigt.

15.1.2 Belauschen der Datenbankverbindung

Wir wollen nun protokollieren, welche Anweisungen der Client-DLL von InterBase aufgerufen werden. In der Enterprise-Version von Delphi gibt es dafür die Komponente *TSQLMonitor*. Damit aber auch die Leser, welche die Professional-Version verwenden, dieses Beispiel nachvollziehen können, werden wir stattdessen eine Callback-Funktion verwenden.

```
function TraceCallback(CallType: TRACECat;
  CBInfo: Pointer): CBRType; stdcall;
var
  s: string;
  rec: pSQLTRACEDesc;
begin
  rec := CBInfo;
  s := rec.pszTrace;
  Form1.Memo1.Lines.Add(s);
  result := cbrUSEDEF;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  SQLConnection1.SetTraceCallbackEvent(TraceCallback, 1);
  SQLConnection1.Connected := true;
end;
```

Diese Callback-Funktion belegt ihrerseits eine Verbindung zur Datenbank. Wenn Sie *SQLConnection1* schon zur Entwurfszeit verbunden haben, dann sind nach dem Programmstart drei Datenbankverbindungen offen – oder auch nicht: Denn wenn Sie als Besitzer der Professional-Version nur eine 2-Benutzer-Lizenz haben, lässt sich das Programm nicht mehr starten. Deshalb setzen wir die Eigenschaft *Connected* von *SQLConnection1* auf *false* und verbinden erst zur Laufzeit.

Die Details der Callback-Verbindung sollen uns hier nicht interessieren – wenn Sie so etwas für eigene Projekte benötigen, dann schreiben Sie hier einfach den Code ab.

```
procedure TForm1.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  SQLConnection1.SetTraceCallbackEvent(nil, 0);
end;
```

Beim Schließen des Programms wird die Protokollfunktion wieder aufgehoben. (Normalerweise würde man alles, was man mit *FormCreate* erzeugt hat, in *FormDestroy* wieder freigeben. In diesem Fall soll die Callback-Funktion jedoch so früh wie möglich »abgeklemmt« werden, so dass wir hier *FormClose* verwenden.)

Für Vergleiche wollen wir uns jetzt noch die Zeilen in *Memo1* zählen lassen:

```
procedure TForm1.Memo1Change(Sender: TObject);
var
  i: integer;
begin
  i := Memo1.Lines.Count;
  if i = 1
    then Label1.Caption := '1 Zeile'
    else Label1.Caption := IntToStr(i) + ' Zeilen';
end;
```

Nach dem Programmstart sehen wir nun, wie die Verbindung zur Datenbank hergestellt wurde:

```
INTERBASE - isc_attach_database
```

Nun Öffnen wir *SQLDataSet1*:

```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
SELECT proj_id, proj_name
  FROM project
```

```
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
```

Zunächst wird Speicher für die Anweisung alloziert und dann eine Transaktion gestartet. Anschließend wird das SQL-Statement zum Server übertragen und vorbereitet. Mit *isc_dsql_describe_bind* werden die Parameter übertragen – in diesem Fall haben wir keine –, danach wird die Anweisung ausgeführt. Mit *isc_dsql_fetch* wird ein (in diesem Fall der erste) Datensatz geholt.

Wird nun gescrollt, dann werden weitere Aufrufe von *isc_dsql_fetch* abgesetzt, so lange, bis der Server meldet, dass keine weiteren Datensätze mehr vorhanden sind. Dies wird von der Komponente registriert, so dass weitere Aufrufe von *Next* keine Aktion mehr auslösen und damit sehr schnell ausgeführt werden.

Ohne persistente TField-Instanzen

Nun wollen wir die Sache ohne persistente TField-Instanzen wiederholen:

```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
select * from PROJECT
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_allocate_statement
SELECT 0, '', '', A.RDB$RELATION_NAME, A.RDB$INDEX_NAME,
    B.RDB$FIELD_NAME, B.RDB$FIELD_POSITION, '', 0, A.RDB$INDEX_TYPE,
    '', A.RDB$UNIQUE_FLAG, C.RDB$CONSTRAINT_NAME,
    C.RDB$CONSTRAINT_TYPE
  FROM RDB$INDICES A, RDB$INDEX_SEGMENTS B
    FULL OUTER JOIN RDB$RELATION_CONSTRAINTS C
      ON A.RDB$RELATION_NAME = C.RDB$RELATION_NAME
        AND C.RDB$CONSTRAINT_TYPE = 'PRIMARY KEY'
  WHERE (A.RDB$SYSTEM_FLAG <> 1 OR A.RDB$SYSTEM_FLAG IS NULL)
    AND (A.RDB$INDEX_NAME = B.RDB$INDEX_NAME)
    AND (A.RDB$RELATION_NAME = UPPER('PROJECT'))
  ORDER BY RDB$INDEX_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
```

```
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_fetch
```

Zunächst geht es genauso los wie mit den persistenten *TField*-Instanzen. Nach dem Aufruf von *isc_dsql_execute* wird jedoch eine weitere Abfrage ausgeführt, die den JOIN einiger Systemtabellen beinhaltet (die Formatierung hier stammt von mir ...). Mit diesem JOIN werden die Indexinformationen der Abfrage ermittelt. Diese würden benötigt, wenn an unserer Datenmenge über einen Provider eine *TClientDataSet*-Komponente hängen würde.

Den Verzicht, diese Indexinformationen zu ermitteln, könnte man auch dadurch bewirken, dass man *NoMetadata* auf *true* setzt.

BLOB-Felder

Die Tabelle *project* – aus diesem Grunde habe ich Sie hier verwendet – hat eine BLOB-Spalte namens *proj_desc*, die wir bislang nicht angezeigt haben. Dies wollen wir nun tun. Fügen Sie dafür eine *TDBMemo*-Instanz ein und setzen die Eigenschaften entsprechend. Für die Abfrage verwenden wir wieder das Joker-Zeichen.

Zunächst wollen wir eine Zeitmessung durchführen. Dazu legen wir die Zuweisung der Callback-Funktion vorübergehend still. Das Öffnen der Abfrage dauert nun 11 091 µs (statt 3 007 µs), das Scrollen 845 µs (statt 115 µs). Beim Öffnen und bei jedem Scrollen werden zusätzlich die folgenden Anweisungen ausgeführt:

```
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_open_blob
INTERBASE - isc_blob_info
INTERBASE - isc_close_blob
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_open_blob
INTERBASE - isc_blob_info
INTERBASE - isc_close_blob
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_allocate_statement
```

```
INTERBASE - isc_open_blob
INTERBASE - isc_blob_info
INTERBASE - isc_close_blob
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_open_blob
INTERBASE - isc_blob_info
INTERBASE - isc_get_segment
INTERBASE - isc_get_segment
INTERBASE - isc_close_blob
INTERBASE - isc_dsql_free_statement
```

Dadurch wird der Vorgang natürlich alles andere als beschleunigt.

Aufruf von First

Fügen Sie einen *TDBNavigator* hinzu, verbinden Sie diesen mit *DataSource1* und starten das Programm. Nach dem Öffnen der Datenmenge und einem Aufruf von *Next* rufen Sie (über den DBNavigator) *First* auf (den einen Aufruf von *Next* benötigen wir, damit der Button *First* überhaupt verfügbar geschalten wird). Hier erfolgen dann die folgenden DLL-Aufrufe:

```
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
```

Die Transaktion wird also bestätigt, dann wird die Anweisung erneut aufgerufen. Der erneute Aufruf von *isc_dsql_execute* legt den Verdacht nahe, dass zwischenzeitlich eingefügte Datensätze dann auch angezeigt werden. Wir werden das gleich experimentell ermitteln und fügen dafür eine neue *TSQLQuery*-Instanz ein, der wir folgendes SQL-Statement zuweisen:

```
INSERT INTO project (proj_id, proj_name)
  VALUES (:id, :name)
```

Mit den folgenden Anweisungen wird ein Datensatz eingefügt:

```
with SQLQuery2 do
begin
  ParamByName('id').AsString := 'TEST';
  ParamByName('name').AsString := 'Dies ist ein TEST';
  ExecSQL;
end;
```

Nach einem Aufruf von *First* wird dieser Datensatz dann mit angezeigt. Im Gegensatz zu *TQuery* ist also ein erneutes Öffnen der Datenmenge nicht erforderlich – der Aufruf von *First* wird deutlich schneller ausgeführt als der von *Open*, weil die Anweisung nicht mehr interpretiert werden muss. (Dasselbe könnte man erreichen, wenn man die Abfrage explizit vorbereiten würde.)

Unidirektionale Datenmenge

Wenn wir schon einen DBNavigator auf dem Formular liegen haben, dann wollen wir auch mal den Aufruf von *Prior* und *Last* testen – allerdings mit mäßigem Erfolg:

Die Komponenten von dbExpress liefern eine unidirektionale Datenmenge, und diese heißt so, weil man nur in eine Richtung navigieren kann: nach hinten. Genau genommen ist noch nicht einmal der Aufruf von *First* möglich, die Entwickler von Borland haben diese Anweisung – wie wir gerade gesehen haben – aber so »hingebogen«, dass die Abfrage neu ausgeführt wird und somit der Cursor auf dem ersten Datensatz steht.

Auch der Versuch, an *DataSource1* ein *DBGrid* zu hängen, wird scheitern. Aufrufe von *Locate* oder *Filtered* ohnehin. Die Schnelligkeit und »Schlankheit« von dbExpress haben halt ihren Preis: geringen Komfort.

15.1.3 TSQLClientDataSet

Um die Datensätze in einem DBGrid anzeigen zu können, könnte man nun an *SQLDataSet1* einen *TDataSetProvider* und daran ein *TClientDataSet* hängen. Borland hat uns aber hier ein wenig Arbeit abgenommen und stellt uns die Komponente *TSQLClientDataSet* zur Verfügung.

Statt *SQLConnection* heißt die Verbindung zu *SQLConnection1* nun *DBConnection* – wir sind ja flexibel. Ansonsten setzen Sie *CommandText* wieder auf *select * from PROJECT*.

Das Öffnen dauert nun wesentlich länger (49 636 µs), das Scrollen auch (1 861 µs). Das Scrollen liegt am DBGrid, bei *TQuery* kämen Sie auf ähnliche Zeiten.

Beim Öffnen der Datenmenge gibt es zwei »Bremsen«:

- Es werden die Systemtabellen abgefragt – da helfen hier auch keine persistenten TField-Instanzen und *NoMetadata* gibt es hier nicht.
- Es werden alle Datensätze auf den Client übertragen, und zwar inklusive aller BLOBS, obwohl diese gar nicht angezeigt werden. (Da hilft es auch nichts, die Spalte aus der Spaltenliste von *DBGrid1* und/oder aus der Liste der persistenten TField-Instanzen zu löschen).

Für das erste Problem habe ich noch keine Lösung gefunden, gegen das zweite kann man jedoch etwas tun:

- Um das Übertragen der Memos zu vermeiden, ändert man entweder das SQL-Statement entsprechend, oder man setzt die Option *poFetchBlobsOnDemand* – Letzteres hat den Vorteil, dass man sie problemlos nachladen kann.
- Mit der Eigenschaft *PacketRecords* kann man einstellen, wie viele Datensätze jeweils übertragen werden. Per Voreinstellung steht diese Eigenschaft auf *-1*, somit werden alle Datensätze übertragen. Würde man *PacketRecords* auf 3 setzen, dann würden jeweils nur drei Datensätze übertragen: Beim Öffnen der Datenmenge die ersten drei, und wenn auf einen weiteren Datensatz gescrollt wird, die nächsten drei. Gerade bei großen Datenmengen macht es viel Sinn, *PacketRecords* auf einen »handlichen« Wert zu setzen.

Datensätze ändern

TSQLClientDataSet hat nicht nur den Vorteil, ein DBGrid verwenden zu können, wir können auch Datensätze einfügen, ändern und löschen. Die folgenden Ausführungen beziehen sich auf alle drei Möglichkeiten der Bearbeitung, deswegen wollen wir uns hier nur den Update-Fall ansehen.

Damit die Änderungen auf den Server übertragen werden, verwenden wir eine *AfterPost*-Ereignisbehandlungsroutine:

```
procedure TForm1.SQLClientDataSet1AfterPost(DataSet: TDataSet);
begin
  SQLClientDataSet1.ApplyUpdates(0);
end;
```

Wir ändern nun einen Datensatz in der Spalte *proj_name*. Mit dem Aufruf von *ApplyUpdates* werden die folgenden Routinen der Client-DLL aufgerufen:

```
INTERBASE - isc_start_transaction
INTERBASE - isc_dsql_allocate_statement
```

```
update PROJECT  set
 PROJ_NAME = ?
where
 PROJ_ID = ? and
 PROJ_NAME = ? and
 TEAM_LEADER is null and
 PRODUCT = ?

INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_commit_transaction
```

Zunächst wird eine Transaktion gestartet und Speicher für die Anweisung alloziert. Die UPDATE-Anweisung beinhaltet das, was verwendet wird, wenn die Spalte *proj_name* geändert wird und *UpdateMode* den Wert *upWhereAll* hat. Anschließend wird das Statement vorbereitet.

Mit *isc_dsql_sql_info* können Informationen über das Statement, insbesondere über Art und Anzahl der Parameter, erhalten werden. InterBase verwendet für Integer-Zahlen ein anderes Format als Windows, somit muss hier mit *isc_vax_integer* eine Umwandlung erfolgen. Mit *isc_dsql_describe_bind* werden Parameterinformationen besorgt, anschließend wird das Statement ausgeführt, der Speicher freigegeben und die Transaktion bestätigt.

So weit, so schön. Nun steht man bei Client-Server-Datenbanksystemen häufig vor dem Problem, dass man eine Datenmenge aktualisieren möchte, die aus einem JOIN resultiert. Dies sollte eigentlich dank der Eigenschaften *UpdateMode* und der *TField*-Eigenschaft *ProviderFlags* kein Problem sein. Stellen wir also *UpdateMode* auf *upWhereKeyOnly* und schauen zu, was passiert (es interessiert hier nur das SQL-Statement):

```
update PROJECT  set
  PROJ_NAME = ?
where
  PROJ_NAME = ? and
  PRODUCT = ?
```

Zunächst sind wir etwas verwundert, weil normalerweise die Spalte *id* (in diesem Fall *proj_id*) den Primärschlüssel bildet.

Sicherheitshalber schauen wir uns die Definition von *project* an, und es ist tatsächlich so, wie wir vermuten:

```
CREATE TABLE project
  (proj_id projno NOT NULL,
  proj_name VARCHAR(20) NOT NULL,
  proj_desc BLOB SUB_TYPE TEXT SEGMENT SIZE 800,
  team_leader empno,
  product prodtype,
  UNIQUE (proj_name),
  PRIMARY KEY (proj_id));
```

Obwohl *TSQLClientDataSet* beim Öffnen Indexinformationen aus den Systemtabellen liest, erkennt sie nicht den korrekten Schlüssel. Nun gut, gibt es noch die Eigenschaft *ProviderFlags*, wo wir es mit *pfInWhere* und *pfInKey* versuchen wollen. Allerdings – Auswirkungen: keine.

Die Komponente *TSQLClientDataSet* ist nicht von *TSQLDataSet*, sondern von *TClientDataSet* (genauer *TCustomClientDataSet*) abgeleitet. Erstellen wir persistente TField-Instanzen, so gehören diese zum ClientDataSet und nicht zum SQLDataSet. Aus diesem Grunde werden nicht nur die Indexinformationen abgefragt, wenn wir die Datenmenge öffnen, wir können auch nicht die Eigenschaft *ProviderFlags* setzen, die den Provider interessieren würde. Fazit: Da muss Borland noch mal nachbessern!

Einzelne Komponenten

In vielen Fällen werden Sie sich die Komponenten *TSQLDataSet*, *TDataSetProvider* und *TClientDataSet* einzeln auf das Formular (oder das Datenmodul) legen müssen. Dann legen Sie bei *TSQLDataSet* persistente TField-Instanzen an und setzen dort entsprechend die Eigenschaft *ProviderFlags*. Und siehe da, das Statement ist so, wie wir es brauchen:

```
update PROJECT  set
  PROJ_NAME = ?
where
  PROJ_ID = ?
```

Schließen des Handles

Am Rande: Wenn Sie mit *TSQLClientDataSet* oder mit *TDataSetProvider* alle Datensätze abrufen, wird hinterher die Transaktion beendet und das Handle freigegeben:

```
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
```

15.1.4 Stored Procedures

Zum Zugriff auf STORED PROCUEDURES hat man in der Regel vier Möglichkeiten:

- Mit *TSQLDataSet* über eine SQL-Anweisung.
- Man kann die *TSQLDataSet*-Eigenschaft *CommandType* auf *ctStoredProc* setzen und dann eine Prozedur auswählen.
- Man kann *TSQLQuery* verwenden.
- Und schließlich gibt es noch *TSQLStoredProc*.

Wir wollen diese vier Möglichkeiten beim Zugriff auf die folgende STORED PROCEDURE vergleichen, die dem Beispielprojekt To-Do-Liste entnommen wurde:

```
ALTER PROCEDURE P_BENUTZER_ID
RETURNS
  (GEN INTEGER)
AS
BEGIN
  gen=GEN_ID(g_benutzer, 1);
  SUSPEND;
END^
```

Bei allen vier Komponenten wurde die Eigenschaft *NoMetadata* auf *true* gesetzt.

TSQLDataSet und SQL-Anweisung

Zunächst verwenden wir *TSQLDataSet* und die folgende SQL-Anweisung:

```
SELECT *
  FROM p_benutzer_id
```

Es werden hier die folgenden Anweisungen verwendet:

```
SQLDataSet2.Open;
s := SQLDataSet2.FieldByName('GEN').AsString;
SQLDataSet2.Close;
```

Dabei werden folgende Client-Aufrufe registriert:

```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
SELECT *
  FROM p_benutzer_id
```

```
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
```

Also ganz das, was wir von anderen Abfragen her kennen.

TSQLDataSet und ctStoredProc

Bei der nächsten *TSQLDataSet*-Instanz setzen wir *CommandType* auf *ctStoredProc* und *CommandText* auf *p_benutzer_id*. Statt *Open* müssen wir hier *ExecSQL* verwenden, den Wert erhalten wir über die Eigenschaft *Params*.

```
SQLDataSet3.ExecSQL;
s := SQLDataSet3.Params[0].AsString;
```

Obwohl wir die Eigenschaft *NoMetadata* auf *true* gesetzt haben, erhalten wir das folgende Protokoll:

```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
SELECT 0, '', '', '', A.RDB$PARAMETER_NAME, A.RDB$PARAMETER_NUMBER,
    A.RDB$PARAMETER_TYPE, B.RDB$FIELD_TYPE, B.RDB$FIELD_SUB_TYPE,
    B.RDB$FIELD_SUB_TYPE, B.RDB$FIELD_LENGTH, 0, B.RDB$FIELD_SCALE,
    B.RDB$NULL_FLAG, A.RDB$SYSTEM_FLAG
  FROM RDB$PROCEDURE_PARAMETERS A, RDB$FIELDS B
  WHERE (A.RDB$FIELD_SOURCE = B.RDB$FIELD_NAME)
    AND (A.RDB$PROCEDURE_NAME = 'P_BENUTZER_ID')
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
EXECUTE PROCEDURE P_BENUTZER_ID
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_sql_info
```

```
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_execute2
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
```

TSQLQuery und TSQLStoredProc

Bei *TSQLQuery* erhält man dasselbe Protokoll wie bei einer SQL-Anweisung von *TSQLDataSet*. *TSQLStoredProc* gleicht *TSQLDataSet*, wenn *ctStoredProc* verwendet wird. Zur Abwechslung wollen wir die Möglichkeit nutzen, den Parameter über den Namen zu erhalten:

```
s := SQLStoredProc1.ParamByName('GEN').AsString;
```

Fazit

Wenn möglich, sollten Sie auf STORED PORCEDURES per SQL-Statement zugreifen, sei es mit *TSQLDataSet*, sei es mit *TQuery*.

15.2 Referenz dbExpress

Im Folgenden sollen die Eigenschaften, Methoden und Ereignisse der *dbExpress*-Komponenten beschrieben werden. Die Komponenten *TSQLDataSet*, *TSQLQuery*, *TSQLTable* und *TSQLStoredProc* sind zwar alle von *TDataSet* abgeleitet, implementieren aber nur einen geringen Teil der *TDataSet*-Elemente. Manche von dort geerbten Elemente bleiben wirkungslos, andere lösen bei ihrer Verwendung eine Exception aus.

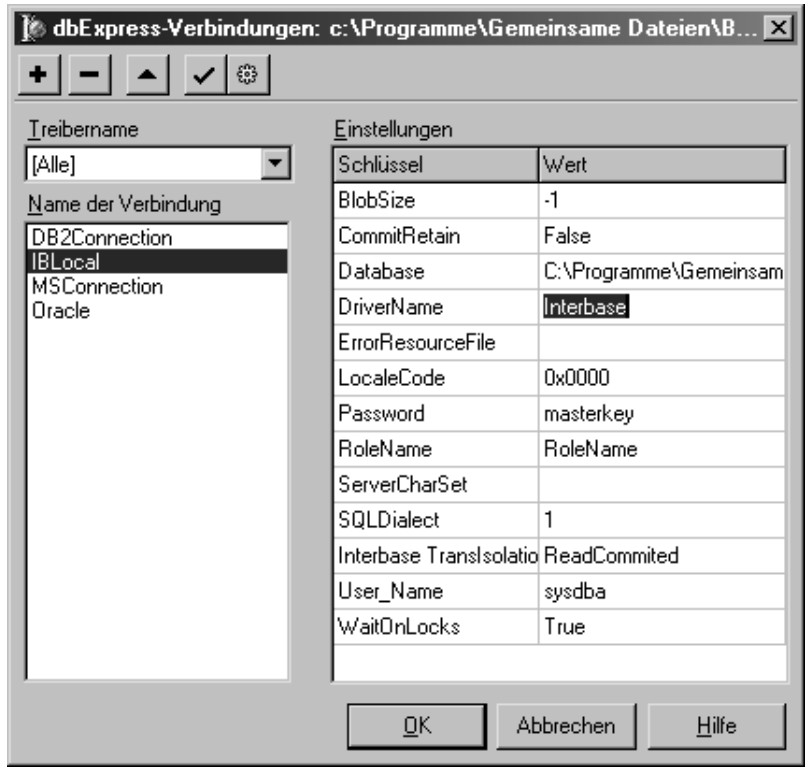
Da alle vier Komponenten von *TCustomSQLDataSet* abgeleitet sind, wollen wir bei der Beschreibung von *TCustomSQLDataSet* auch alle Elemente von *TDataSet* besprechen, die sich hier (sinnvoll) verwenden lassen.

15.2.1 TSQLConnection

Mit der Komponente *TSQLConnection* wird die Verbindung zur Client-DLL des Datenbanksystems hergestellt. Des Weiteren wird hier die Transaktionssteuerung vorgenommen.

Verbindung zur Datenbank

Um die Verbindung zur Datenbank herzustellen, verwendet man am einfachsten den Komponenteneditor von *TSQLConnection*:



Die Einstellungen, die Sie hier vornehmen müssen, hängen stark vom verwendeten Datenbanksystem ab. Bei InterBase beispielsweise benötigen wir unter *Database* den Namen der **.gdb*-Datei. Wenn auf den Login-Dialog verzichtet werden soll, dann müssen auch noch *User_Name* und *Password* gesetzt werden.

- ConnectionName (Eigenschaft, veröffentlicht)

  ```
  property ConnectionName: string;
  ```

  Mit *ConnectionName* wird der Name der Verbindung angegeben. Wenn Sie die Eigenschaften einer Verbindung ändern oder eine neue Verbindung anlegen, dann werden deren Parameter in der Datei *Dbxconnections.ini* gespeichert. Möchten Sie in anderen Anwendungen nun dieselben Parameter verwenden, dann müssen Sie lediglich *ConnectionName* entsprechend wählen.

- Connected (Eigenschaft, veröffentlicht)

  ```
  property Connected: Boolean;
  ```

  Wird *Connected* auf *true* gesetzt, dann wird eine Verbindung zur Datenbank hergestellt.

- Params (Eigenschaft, veröffentlicht)

  ```
  property Params: TStrings;
  ```

  Beim Setzen von *ConnectionName* werden die Parameter aus der Datei *Dbxconnections.ini* nach *Params* kopiert, die Verbindung arbeitet dann mit dem Inhalt von *Params*.

- LoadParamsOnConnect (Eigenschaft, veröffentlicht), LoadParamsFromIniFile (Methode)

  ```
  property LoadParamsOnConnect: Boolean default false;
  procedure LoadParamsFromIniFile(AFileName: String = '' );
  ```

  Per Voreinstellung ist der Inhalt von *Params* für die Datenbankverbindung relevant, nicht der von *Dbxconnections.ini*. Sollen aber beim Herstellen der Verbindung – gewöhnlich also beim Programmstart – die Parameter aus *Dbxconnections.ini* geladen werden, dann ist *LoadParamsOnConnect* auf *true* zu setzen. Alternativ kann man auch manuell *LoadParamsFromIniFile* aufrufen.

- DriverName, VendorLib, LibraryName (Eigenschaften, veröffentlicht)

  ```
  property DriverName: string;
  property LibraryName: string;
  property VendorLib: string;
  ```

  Beim Setzen von *ConnectionName* werden für diese drei Eigenschaften passende Werte gesetzt, die im Normalfall nicht abgeändert werden brauchen. *DriverName* ist der Name des Datenbanktreibers, also beispielsweise *InterBase*. *LibraryName* ist der Name der DLL, mit der *dbExpress* gerade arbeitet. Im Falle von InterBase wäre dies *dbexpint.dll*. *VendorLib* ist der Name des Datenbank-Clients, im Falle von InterBase also *GDS32.DLL*.

- KeepConnection (Eigenschaft, veröffentlicht)

  ```
  property KeepConnection: Boolean default true;
  ```

  Hat *KeepConnection* den Wert *false*, dann wird die Verbindung zur Datenbank abgebaut, wenn die letzte offene Datenmenge geschlossen wird.

- AfterConnect, AfterDisconnect, BeforeConnect, BeforeDisconnect

  ```
  property AfterConnect(Sender: TObject);
  property AfterDisconnect(Sender: TObject);
  property BeforeConnect(Sender: TObject);
  property BeforeDisconnect(Sender: TObject);
  ```

  Diese Ereignisse treten vor beziehungsweise nach dem Erstellen oder Trennen der Datenbankverbindung auf.

Login

- LoginPrompt (Eigenschaft, veröffentlicht)

```
property LoginPrompt: Boolean default true;
```

Um zu vermeiden, dass sich der Anwender beim Server anmelden muss, setzt man *LoginPrompt* auf *false*. Der Benutzername und das Passwort müssen dann bei den Parametern angegeben sein.

- OnLogin (Ereignis)

```
property OnLogin(Database: TSQLConnection; LoginParams: TStrings)
```

Hat *LoginPrompt* den Wert *true*, dann wird das Ereignis *OnLogin* aufgerufen, bevor der Anmelde-Dialog angezeigt wird. Hier können Sie in *LoginParams* Benutzernamen und Passwort setzen, um die Anzeige des Login-Dialogs zu vermeiden.

Transaktionssteuerung

- StartTransaction, Commit, Rollback (Methoden)

```
procedure StartTransaction(TransDesc: TTransactionDesc);
procedure Commit(TransDesc: TTransactionDesc);
procedure Rollback(TransDesc: TTransactionDesc);
```

Mit *StartTransaction* wird eine Transaktion gestartet, mit *Commit* bestätigt, mit *Rollback* zurückgenommen. Allen drei Methoden muss ein Parameter vom Typ *TTransactionDesc* übergeben werden:

```
TTransactionDesc = packed record
  TransactionID: LongWord;
  GlobalID: LongWord;
  IsolationLevel: (xilDIRTYREAD, xilREADCOMMITTED,
    xilREPEATABLEREAD, xilCUSTOM);
  CustomIsolation: LongWord;
end;
```

Dabei muss *TransactionID* auf einen beliebigen, aber eindeutigen und innerhalb der Transaktion einheitlichen Wert sowie *IsolationLevel* auf einen der ersten drei Werte gesetzt werden.

Wird nicht explizit eine Transaktion gestartet, so geschieht dies automatisch. Diese wird dann jedoch nach jeder »Anweisung« beendet.

- InTransaction, TransactionsSupported (Eigenschaften, öffentlich, nur Lesen)

```
property InTransaction: Boolean;
property TransactionsSupported: LongBool;
```

Mit *InTransaction* kann man prüfen, ob derzeit eine Transaktion gestartet ist. Mit *TransactionsSupported* wird geprüft, ob das Datenbanksystem Transaktionen unterstützt – bei MySQL ist dies derzeit nicht der Fall.

Informationen über die Datenbank

- GetTableNames, GetFieldNames, GetIndexNames (Methoden)

```
procedure GetTableNames(List: TStrings;
  SystemTables: Boolean = False);
procedure GetFieldNames(const TableName: string; List: TStrings);
procedure GetIndexNames(const TableName: string; List: TStrings);
```

Mit *GetTableNames* kann man die Liste aller Tabellen in die String-Liste *List* schreiben. Normalerweise werden diejenigen Tabellen nach *List* geschrieben, die den Kriterien von *TableScope* genügen. Setzt man *SystemTables* auf *true*, dann liefert *GetTableNames* ausschließlich Systemtabellen.

Mit *GetFieldNames* erhält man die Liste aller Spalten in der angegebenen Tabelle, mit *GetIndexNames* die Liste aller Indizes.

- TableScope (Eigenschaft, veröffentlicht)

```
property TableScope: set of (tsSynonym, tsSysTable, tsTable, tsView)
  default [tsTable,tsView];
```

Mit *TableScope* spezifiziert man, welche Tabellen von *GetTableNames* ermittelt werden sollen.

- GetProcedureNames, GetProcedureParams (Methoden)

```
procedure GetProcedureNames(List: TStrings);
procedure GetProcedureParams(ProcedureName: String; List: TList);
```

Mit *GetProcedureNames* erhält man eine Liste aller STORED PROCEDURES, mit *GetProcedureParams* die Liste aller Parameter in der angegebenen Prozedur.

- SetTraceCallbackEvent (Methode),
  TraceCallbackEvent (Eigenschaft, öffentlich, nur Lesen)

```
procedure SetTraceCallbackEvent(Event: TSQLCallbackEvent;
  IClientInfo: Integer);
property TraceCallbackEvent: TSQLCallbackEvent;
```

Mit *TraceCallbackEvent* kann man die Kommunikation zum Datenbank-Server »belauschen«. Gehen Sie hier wie folgt vor:

```
function TraceCallback(CallType: TRACECat;
  CBInfo: Pointer): CBRType; stdcall;
var
  s: string;
  rec: pSQLTRACEDesc;
begin
  rec := CBInfo;
  s := rec.pszTrace;
  Form1.Memo1.Lines.Add(s);
  result := cbrUSEDEF;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  SQLConnection1.SetTraceCallbackEvent(TraceCallback, 1);
  SQLConnection1.Connected := true;
end;
```

Verbundene Datenmengen

- DataSets, DataSetCount (Eigenschaften, öffentlich, nur Lesen)

```
property DataSets[Index: Integer]: TCustomSQLDataSet;
property DataSetCount: Integer;
```

Über die Array-Eigenschaft *DataSets* erhält man Referenzen auf alle verbundenen Datenmengen. Deren Anzahl ermittelt man mit *DataSetCount*.

- CloseDataSets (Methode)

```
procedure CloseDataSets;
```

Mit *CloseDataSets* schließt man alle verbundenen Datenmengen.

Sonstiges

- SQLHourGlass (Eigenschaft, öffentlich)

```
property SQLHourGlass: Boolean default true;
```

Den Sanduhr-Cursor beim Ausführen von SQL-Anweisungen kann man abschalten, indem man *SQLHourGlass* auf *false* setzt.

- Execute, ExecuteDirect (Methoden)

```
function Execute(const SQL: string; Params: TParams;
  ResultSet: Pointer = nil): Integer;
function ExecuteDirect(const SQL: string): LongWord;
```

Mit *Execute* kann die übergebene SQL-Anweisung ausgeführt werden. Parameter werden mit *Params* übergeben. Liefert die Anweisung eine Ergebnismenge, dann wird dafür eine *TCustomSQLDataSet*-Instanz erzeugt und der Zeiger darauf *ResultSet* zugewiesen.

Mit *ExecuteDirect* kann man eine Anweisung ohne Parameter ausführen. Sollte diese eine Ergebnismenge liefern, wird diese verworfen.

Für die Ausführung von SQL-Anweisungen kann man auch *TSQLDataSet* und *TSQLQuery* verwenden.

15.2.2 TCustomSQLDataSet

Die Komponente TCustomSQLDataSet ist der gemeinsame Vorfahre von *TSQLDataSet*, *TSQLQuery*, *TSQLTable* und *TSQLStoredProc*. Da diese vier Komponenten die Elemente von *TDataSet* nur sehr unvollständig implementieren, wollen wir hier diejenigen Elemente von *TDataSet* besprechen, die verwendet werden können.

Verbindung zur Datenbank

- SQLConnection (Eigenschaft, veröffentlicht)

```
property SQLConnection: TSQLConnection;
```

Mit der Komponenteneigenschaft *SQLConnection* wird die Verbindungskomponente gewählt, über die eine Verbindung zur Datenbank hergestellt wird.

- TransactionLevel (Eigenschaft, öffentlich)

```
property TransactionLevel: SmallInt default 0;
```

Mit *TransactionLevel* wird spezifiziert, im Kontext welcher Transaktion das Öffnen der Datenmenge erfolgt. Dieser Wert muss mit demjenigen übereinstimmen, der bei *StartTransaction* verwendet wurde. Hat *TransactionLevel* den Vorgabewert Null, dann wird die zuletzt gestartete Transaktion verwendet.

Vorsicht: Nicht alle Datenbanksysteme unterstützen das Öffnen mehrerer Transaktionen.

- NoMetadata (Eigenschaft, veröffentlicht)

```
property NoMetadata: Boolean default false;
```

Wird *NoMetadata* auf *true* gesetzt, dann werden keine Indexinformationen vom Server abgerufen, was das Öffnen der Datenmenge beschleunigt. In manchen Fällen gibt es aber dann Probleme, wenn über einen Provider Datensätze eingetragen werden sollen.

Status der Datenmenge

- Active (Ereignis, veröffentlicht)

```
property Active: boolean;
```

Mit der Eigenschaft *Active* kann die Datenmenge geöffnet und geschlossen werden.

- Open, Close (Methoden, TDataSet)

```
procedure Open;
procedure Close;
```

Mit *Open* wird die Datenmenge geöffnet, mit *Close* wird sie geschlossen.

- AfterClose, AfterOpen, BeforeClose, BeforeOpen (Ereignisse)

```
property AfterClose(DataSet: TDataSet)
property AfterOpen(DataSet: TDataSet)
property BeforeClose(DataSet: TDataSet)
property BeforeOpen(DataSet: TDataSet)
```

Diese Ereignisse treten vor beziehungsweise nach dem Schließen beziehungsweise Öffnen der Datenmenge auf.

- State (Eigenschaft, öffentlich, nur Lesen)

```
property State: TDataSetState;
```

Die Eigenschaft *State* zeigt den Status der Datenmenge an und kann folgende Werte annehmen:

  - *dsInactive*: Die Datenmenge ist nicht aktiv, auf ihre Daten kann nicht zugegriffen werden.
  - *dsBrowse*: Die Daten können angezeigt, jedoch nicht geändert werden. Solange keine anderen Operationen durchgeführt werden, hat eine Datenmenge diesen Status.
  - *dsCalcFields*: Ein *OnCalcFields*-Ereignis wurde ausgelöst, es werden also gerade Felder berechnet.

- DisableControls, EnableControls, ControlsDisabled (Methoden)

```
procedure DisableControls;
procedure EnableControls;
function ControlsDisabled: Boolean;
```

Werden die Daten einer Datenmenge geändert, so wird im Regelfall laufend die Anzeige aktualisiert. Werden viele Daten in einer Schleife eingefügt, verlangsamt dies den Vorgang deutlich. Mit *DisableControls* kann deshalb diese Aktualisierung abgeschaltet und mit *EnableControls* anschließend wieder angeschaltet werden.

Mit *ControlsDisabled* kann ermittelt werden, ob die Aktualisierung abgeschaltet ist.

Navigieren in der Datenmenge

- First, Next (Methoden)

```
procedure First;
procedure Next;
```

Eigentlich unterstützen unidirektionale Datenmengen nur den Aufruf von *Next*. Ruft man *First* auf, dann wird die Abfrage neu ausgeführt.

- BOF, EOF (Eigenschaften, öffentlich, nur Lesen)

```
property BOF: Boolean;
property EOF: Boolean;
```

Die Eigenschaft *BOF* ist dann gleich *true*, wenn die Methode *First* aufgerufen wurde. Schlägt das Verschieben des Datenzeigers fehl, weil er schon auf dem letzten Datensatz steht, wird *EOF* auf *true* gesetzt.

Diese beiden Eigenschaften werden vor allem für die Abbruchbedingungen von Schleifen benötigt.

```
while not Table1.EOF do
begin
    ...
  Table1.Next;
end;
```

- MoveBy (Methode)

```
function MoveBy(Distance: Integer): Integer;
```

Mit der Methode *MoveBy* kann man um die im Parameter angegebene Zahl von Datensätzen nach hinten navigieren. Bei unidirektionalen Datenmengen funktioniert *MoveBy* nur mit positiven *Distance*-Werten.

- AfterScroll, BeforeScroll (Ereignisse)

```
property AfterScroll(DataSet: TDataSet);
property BeforeScroll(DataSet: TDataSet);
```

Diese Ereignisse treten vor beziehungsweise nach dem Verschieben des Datenzeigers auf.

Zugriff auf die Daten

- Fields (Eigenschaft, öffentlich)

```
property Fields[Index: Integer]: TField ;
```

Mit der Array-Eigenschaft *Fields* kann auf jedes Feld der Datenmenge zugegriffen werden.

Bei einem Verändern der Tabellenstruktur kann sich jedoch die Reihenfolge der Felder verändern. Verwenden Sie deshalb besser *FieldByName*.

Das Objekt *TField* wird in Kapitel 3.2 besprochen.

- FieldByName, FindField (Methoden)

```
function FieldByName(const FieldName: string): TField;
function FindField(const FieldName: string): TField;
```

Die Methode *FieldByName* greift über den Feldnamen auf das Feld zu – solange das betreffende Feld noch vorhanden ist, kommt es bei einer Änderung der Tabellenstruktur zu keinen Schwierigkeiten. Ist der als Parameter übergebene Spaltenname nicht vorhanden, dann löst *FieldByName* eine Exception aus, während *FindField* den Wert *nil* zurückgibt.

```
Table1.FieldByName('Vorname').AsString := 'Patty';
```

Das Objekt *TField* wird in Kapitel 3.2 besprochen.

- FieldValues (Eigenschaft, öffentlich)

```
property FieldValues[const FieldName: string]: Variant;
```

Mit besonders wenig Schreibaufwand kann mittels der Array-Eigenschaft *FieldValues* auf die Feldinhalte zugegriffen werden: Zum einen hat diese Eigenschaft den Datentyp *Variant*, so dass ohne weitere Konvertierung die Werte zugewiesen werden können. Zum anderen ist diese Eigenschaft die Default-Eigenschaft von *TDataSet*, sie muss also gar nicht erwähnt werden. Um den Inhalt des Feldes *Vorname* der Variablen *s* zuzuweisen, formuliert man einfach:

```
s := SQLDataSet1['Vorname'];
```

- DefaultFields (Eigenschaft, öffentlich, nur Lesen)

```
property DefaultFields Boolean;
```

  Mit *DefaultFields* kann ermittelt werden, ob *TField*-Instanzen beim Öffnen der Datenmenge angelegt werden müssen (*true*), oder ob es persistente *TField*-Instanzen gibt (*false*).

- CanModify (Eigenschaften, öffentlich, nur Lesen)

```
property CanModify: Boolean;
```

  Mit Hilfe der Eigenschaft *CanModify* kann festgestellt werden, ob die Datenmenge geändert werden kann. Sie gibt bei unidirektionalen Datenmengen immer *false* zurück.

- FieldCount, RecordCount (Eigenschaften, öffentlich, nur Lesen)

```
property FieldCount: Integer;
property RecordCount: Integer;
```

  Mit *FieldCount* kann die Zahl der Spalten (*FieldCount*) ermittelt werden. *RecordCount* zählt die Datensätze und holt dabei alle noch nicht abgerufenen Datensätze vom Server.

- IsEmpty (Methode)

```
function IsEmpty: Boolean;
```

  IsEmpty gibt *true* zurück, wenn die Datenmenge leer ist.

- GetFieldNames (Methode, TDataSet)

```
procedure GetFieldNames(List: TStrings);
```

  Die Methode *GetFieldNames* ermittelt die Spaltennamen einer Tabelle und schreibt sie in die vorgegebene String-Liste.

```
Table1.GetFieldNames(ListBox1.Items);
```

- OnCalcFields (Ereignis)

```
property OnCalcFields(DataSet: TDataSet)
```

  Um einem berechneten Feld einen Wert zuzuweisen, wird das Ereignis *OnCalcFields* verwendet.

Aktualisieren der Datenmenge

- Refresh (Methode), BeforeRefresh, AfterRefresh (Ereignisse)

```
procedure Refresh;
property BeforeRefresh(DataSet: TDataSet);
property AfterRefresh(DataSet: TDataSet);
```

Mit *Refresh* aktualisieren Sie die Datenmenge. Davor wird das Ereignis *BeforeRefresh* aufgerufen, anschließend das Ereignis *AfterRefresh*. Bei den unidirektionalen Datenmengen sorgt der Aufruf von *Refresh* dafür, dass die Datenmenge geschlossen und neu geöffnet wird:

```
procedure TCustomSQLDataSet.InternalRefresh;
begin
  SetState(dsInactive);
  CloseCursor;
  OpenCursor(False);
end;
```

- IsUnidirectional (Eigenschaft, öffentlich, nur Lesen)

```
property IsUnidirectional: Boolean;
```

IsUnidirectional gibt *true* zurück, wenn es sich um eine unidirektionale Datenmenge handelt.

Parameter

Die folgenden Elemente sind in *TSQLTable* nicht veröffentlicht:

- Params (Eigenschaft, veröffentlicht), ParamByName (Methode)

```
property Params: TParams;
function ParamByName(const Value: string): TParam;
```

Mit Hilfe von *Params* und *ParamByName* kann man auf die Parameter zugreifen – entweder über deren Index oder über deren Namen.

- DataSource (Eigenschaft, veröffentlicht)

```
property DataSource: TDataSource;
```

Über die Eigenschaft *DataSource* können Master-Detail-Verbindungen hergestellt werden. Aus der angegebenen Datenquelle bezieht die Abfrage diejenigen Parameter, die ihr nicht explizit zugewiesen werden. Des Weiteren wird nach dem Scrollen der mit *DataSource* verbundenen Datenmenge die Client-Datenmenge mit aktualisierten Parametern neu ausgeführt.

- Prepared (Eigenschaft, öffentlich)

```
property Prepared: Boolean;
```

Vor der Ausführung einer SQL-Anweisung muss diese auf dem Server interpretiert werden. Wird dieselbe Anweisung mehrmals hintereinander ausgeführt, dann braucht sie nur ein einziges Mal interpretiert werden, selbst dann, wenn sich die Werte der Parameter geändert haben.

Um zu verhindern, dass die Anweisung in einem solchen Fall nochmals interpretiert wird, setzt man *Prepared* auf true.

15.2.3 TSQLDataSet

Die Komponente *TSQLDataSet* ist der »vorgesehene« Weg, dbExpress einzusetzen. Die anderen Datenmengenkomponenten (*TSQLQuery*, *TSQLTable* und *TSQLStoredProc* dienen lediglich dazu, dem Programmierer die Umstellung zu erleichtern.)

- CommandType, CommandText (Eigenschaften, veröffentlicht)

```
property CommandType: (ctQuery, ctTable, ctStoredProc);
property CommandText: string;
```

Mit *CommandType* wird eingestellt, wie *CommandText* zu interpretieren ist:

- Hat *CommandType* den Wert *ctQuery*, dann wird mit *CommandText* eine SQL-Anweisung eingegeben. Handelt es sich um ein SELECT-Statement, kann anschließend die Datenmenge mit *Open* geöffnet werden, anderenfalls kann die Anweisung mit *ExecSQL* ausgeführt werden.

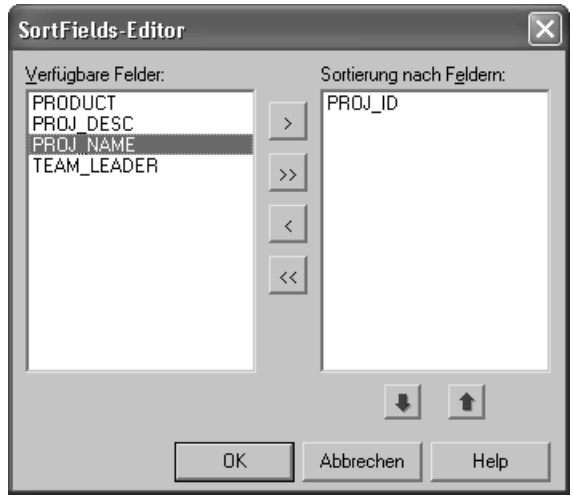
  Für *CommandText* steht dann ein Eigenschaftseditor zur Verfügung, der das Erstellen von Abfragen erleichtert:

- Hat *CommandType* den Wert *ctTable*, dann wird mit *CommandText* ein Tabellenname ausgewählt; der Objektinspektor stellt dafür eine Nachschlageliste zur Verfügung. Die Komponenten generiert automatisch eine SQL-Abfrage, welche alle Spalten der betreffenden Tabelle holt (SELECT * FROM tabelle).
- Bei *ctStoredProc* stellt der Objektinspektor eine Liste aller STORED PROCEDURES zur Verfügung.

- SortFieldNames (Eigenschaft, veröffentlicht)

```
property SortFieldNames: string;
```

Mit dem Eigenschaftseditor kann eine Spaltenliste erstellt werden, aus der – wenn *CommandType* den Wert *ctTable* hat – dann eine ORDER-Klausel gebildet wird.



- ExecSQL (Methode)

```
function ExecSQL(ExecDirect: Boolean = False): Integer;
```

Mit *ExecSQL* kann eine SQL-Anweisung oder eine STORED PROCEDURE ausgeführt werden. Da mit *ExecSQL* keine Datenmenge geöffnet wird, eignet sie sich nicht für SELECT-Statements und Prozeduren, die mehrere Datensätze zurückgeben.

Um die Anweisung ohne Vorbereitung direkt auszuführen, setzen Sie *ExecDirect* auf *true*. Der Rückgabewert der Funktion beinhaltet die Anzahl der bearbeiteten Datensätze.

15.2.4 TSQLQuery

Die Komponente *TSQLQuery* tut nichts anderes als *TSQLDataSet*, wenn *CommandType* auf *ctQuery* gesetzt ist. Die SQL-Anweisung wird jedoch nicht der Eigenschaft *CommandText*, sondern der Eigenschaft *SQL* zugewiesen.

15.2.5 TSQLTable

Die Komponente *TSQLTable* dient dem Zugriff auf eine einzelne Tabelle. Die Verwendung ist an *TTable* (Kapitel 5) angelehnt.

- TableNames (Eigenschaft, veröffentlicht)

  ```
  property TableName: string;
  ```

  Mit *TableName* wird der Name der Tabelle spezifiziert, für die eine Abfrage erstellt werden soll.

- MasterSource, MasterFields (Eigenschaften, veröffentlicht)

  ```
  property MasterSource: TDataSource;
  property MasterFields: string;
  ```

  Zum Aufbau einer Master-Detail-Verknüpfung setzt man von der Detailtabelle *MasterSource* auf die Datenquelle der Master-Datenmenge und erstellt mit dem Eigenschaftseditor von *MasterFields* eine Liste der Spalten, über welche die Verknüpfung hergestellt werden soll.

- IndexFieldNames (Eigenschaft, veröffentlicht)

  ```
  property IndexFieldNames: string;
  ```

  Aus den hier aufgeführten Spalten wird eine ORDER-Klausel generiert. Mehrere Spalten sind durch Semikola zu trennen.

- IndexName (Eigenschaft, veröffentlicht)

  ```
  property IndexName: string;
  ```

  IndexName sollten Sie lieber nicht verwenden: Die Komponente muss hier nämlich beim Datenbanksystem nachfragen, aus welchen Spalten sich der betreffende Index zusammensetzt, und generiert daraus dann die ORDER-Klausel. Setzen Sie lieber die Eigenschaft *IndexFieldNames*.

- DeleteRecords (Methode)

  ```
  procedure DeleteRecords;
  ```

  Mit *DeleteRecords* entfernt man alle Datensätze aus der aktuellen Tabelle.

15.2.6 TSQLStoredProc

Mit *TSQLStoredProc* können Sie STORED PROCEDURES ausführen. Die Verwendung ist an die BDE-Komponente *TStoredProc* angelehnt: Der Prozeduren-Name wird mit *StoredProcName* angegeben, die Prozedur mit *ExecProc* ausgeführt.

Sollte die Prozedur mehrere Ergebnismengen zurückgeben, dann schauen Sie sich in der Online-Hilfe die Methode *NextRecordSet* an – dort gibt es dann auch ein Beispiel.

15.2.7 TSQLClientDataSet

Die Komponente *TSQLClientDataSet* ist wie *TClientDataSet* von der Komponente *TCustomClientDataSet* abgeleitet und weist nahezu dieselbe Funktionalität auf. *TClientDataSet* ist in Kapitel 7 ausführlich beschrieben, eine Wiederholung soll hier nicht erfolgen.

- SQLConnection (Eigenschaft, veröffentlicht)

  ```
  property SQLConnection: TSQLConnection;
  ```

  Mit der Komponenteneigenschaft *SQLConnection* wird die Verbindungskomponente gewählt, über die eine Verbindung zur Datenbank hergestellt wird.

- CommandType, CommandText (Eigenschaften, veröffentlicht)

  ```
  property CommandType: (ctQuery, ctTable, ctStoredProc);
  property CommandText: string;
  ```

  Mit *CommandType* wird eingestellt, wie *CommandText* zu interpretieren ist:

  - Hat *CommandType* den Wert *ctQuery*, dann wird mit *CommandText* eine SQL-Anweisung eingegeben. Dabei solle es sich um ein SELECT-Statement handelt, da *TSQLClientDataSet* die Methode *ExecSQL* nicht kennt.

    Für *CommandText* steht ein Eigenschaftseditor zur Verfügung, der das Erstellen von Abfragen erleichtert.

  - Hat *CommandType* den Wert *ctTable*, dann wird mit *CommandText* ein Tabellenname ausgewählt; der Objektinspektor stellt dafür eine Nachschlageliste zur Verfügung. Die Komponente generiert automatisch eine SQL-Abfrage, welche alle Spalten der betreffenden Tabelle holt (SELECT * FROM tabelle).

  - Bei *ctStoredProc* stellt der Objektinspektor eine Liste aller STORED PROCEDURES zur Verfügung. Verwenden Sie hier nur Prozeduren, die eine Ergebnismenge liefern.

15.2.8 TSQLMonitor

Mit der Komponente *TSQLMonitor* können Sie den Verkehr zum Datenbank-Server »belauschen«.

TSQLMonitor liegt nur in der Enterprise-Version auf der Komponentenpalette. (Wenn Sie im Besitz der Professional-Version sind, dann schauen Sie sich doch mal genau die Datei *SqlExpr.pas* an ...)

Die Komponente *TSQLMonitor* setzt ihrerseits eine Call-Back-Routine ein – den gleichzeitigen Aufruf der *TSQLConnection*-Methode *SetTraceCallbackEvent* sollten Sie sich dann verkneifen.

- SQLConnection, Active (Eigenschaften, veröffentlicht)

  ```
  property SQLConnection: TSQLConnection;
  property Active: Boolean;
  ```

  Mit *SQLConnection* wird die Verbindungskomponente gewählt, deren DLL-Aufrufe protokolliert werden sollen. Mit *Active* kann die Protokollierung ein- und ausgeschaltet werden.

- OnTrace, OnLogTrace (Ereignisse)

  ```
  property OnTrace: TTraceEvent(Sender: TObject;
    CBInfo: pSQLTRACEDesc; var LogTrace: Boolean);
  property OnLogTrace: (Sender: TObject; CBInfo: pSQLTRACEDesc);
  ```

  Das Ereignis *OnTrace* tritt auf, bevor ein DLL-Aufruf in die Log-Liste geschrieben wird – man kann dies verhindern, indem man *LogTrace* auf *false* setzt.

  OnLogTrace tritt auf, nachdem der Aufruf in die Liste geschrieben wurde.

- TraceList (Eigenschaft, veröffentlicht)

  ```
  property TraceList: TStrings;
  ```

  Die DLL-Aufrufe werden in diese String-Liste geschrieben.

- AutoSave, FileName (Eigenschaften, veröffentlicht)

  ```
  property AutoSave: Boolean;
  property FileName: string;
  ```

  Mit diesen Eigenschaften kann man die Log-Liste automatisch in eine Datei speichern. (Alternativ kann man auch *LoadFromFile* und *SaveToFile* aufrufen.)

16 Multi-Tier-Anwendungen

Ab Version 3 ist es mit Delphi möglich, mehrschichtige Anwendungen zu erstellen. Mit den Vorzügen und Nachteilen von solchen so genannten Multi-Tier-Anwendungen haben wir uns schon in Kapitel 2 beschäftigt. Hier soll nun eine kleine Beispielanwendung erstellt werden. Da mehrschichtige Anwendungen nicht für alle Leser von Interesse sind und zum Nachvollziehen der Beispiele auch die Client-Server-Suite erforderlich ist, wollen wir dieses Thema nur streifen.

16.1 Multi-Tier mit DCOM

Zunächst werden wir eine Multi-Tier-Anwendung auf der Basis von DCOM aufbauen; dazu reicht auch Delphi 3. Anschließend werden wir einen CORBA-Server und einen CORBA-Client entwickeln, dafür ist dann Delphi 4 erforderlich.

16.1.1 Erstellung des Middle-Tiers

Die Erstellung von Multi-Tier-Anwendungen geschieht zweckmäßigerweise in folgender Reihenfolge:

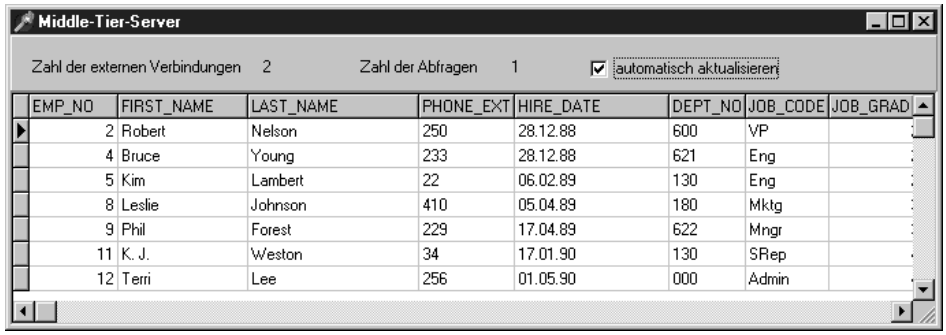
- Zunächst wird die Datenbank auf dem Datenbank-Server erstellt. In vielen Fällen ist sie bereits vorhanden.
- Anschließend wird der Anwendungs-Server, also die mittlere Schicht erstellt. Hier kann bereits die Verbindung zur Datenbank hergestellt und mit Live-Daten gearbeitet werden. Dabei wird der Anwendungs-Server als OLE-Automatisierungsobjekt registriert.
- Zuletzt wird der Client erstellt. Sobald mit der Komponente *TRemoteServer* der Versuch eines Zugriffs auf den Anwendungs-Server unternommen wird, wird dieser automatisch gestartet. Somit kann auch hier schon während des Entwurfs mit Live-Daten gearbeitet werden.

Als Datenbank soll hier die *InterBase*-Testdatenbank *Employee.gdb* verwendet werden, auf welche der Alias *IBLOCAL* eingerichtet ist. Somit kann hier gleich mit der Erstellung des Anwendungs-Servers begonnen werden. Zu diesem Zweck wird ein neues Projekt begonnen.

Der Anwendungs-Server soll unauffällig im Hintergrund arbeiten, ein Formular wäre somit nicht zwingend erforderlich. Bei der Fehlersuche hat es sich jedoch

als zweckmäßig erwiesen, wenn der Anwendungs-Server gewisse Informationen zur Verfügung stellt. Für gewöhnlich wird man die Zahl der aktuellen Verbindungen und die Zahl der bislang erfolgten Abfragen anzeigen. Hier im Beispiel soll auch noch die Tabelle angezeigt werden, deren Daten der Server weitervermittelt. (Mit Server ist hier in Kapitel 16 stets der Anwendungs-Server gemeint.)

Zunächst wird also ein Formular gemäß dem folgenden eingerichtet, außerdem wird dem Projekt ein Datenmodul und diesem werden die Komponenten *Database1*, *Query1* und *DataSource1* hinzugefügt:



Aufgabe der *TDatabase*-Komponente ist es, die Anwendung automatisch bei *InterBase* anzumelden, so dass kein Passwort-Dialog erscheint. Verknüpfen Sie nun die einzelnen Komponenten so, dass Sie in *DBGrid1* die Tabelle betrachten können.

Externes Datenmodul hinzufügen

Nun muss ein *externes Datenmodul* hinzugefügt werden. Wählen Sie dazu DATEI|NEU|EXTERNES DATENMODUL. Prinzipiell könnte man in dieses Datenmodul nun eine *TProvider*-Komponente einfügen, welche die Daten von *DataModule2.Query1* besorgt und weitervertreibt. Dies würde jedoch später einige Probleme bereiten.

Fügen Sie deshalb in das externe Datenmodul eine *TQuery*-Komponente ein und verbinden Sie diese über die *DataModule2.Database1*-Komponente mit der Datenbank und mit der Tabelle *employee*. Bei Ihren eigenen Projekten sollten Sie darauf achten, dass der Einsatz von *TDatabase*-Komponenten innerhalb von externen Datenmodulen zu Konflikten führen kann. Verwenden Sie deshalb für *TDatabase*-Komponenten stets ein normales Datenmodul.

Damit die *TQuery*-Komponente registriert wird, muss die Funktion *Get_Query1* erstellt werden. Damit dies Delphi automatisch tut, wählen Sie aus dem Kontextmenü von *Query1* den Eintrag QUERY1 AUS DATENMODUL EXPORTIEREN.

Damit die Zahl der Verbindungen in *Form1.Label2* angezeigt wird, werden die *OnCreate*- und *OnDestroy*-Ereignisse des externen Datenmoduls mit folgenden Anweisungen verbunden:

```
procedure TMyMT.MyMTCreate(Sender: TObject);
begin
  Form1.Label2.Caption
    := IntToStr(StrToInt(Form1.Label2.Caption) + 1);
  Application.Title := 'Server ' + Form1.Label2.Caption + ' V';
end;

procedure TMyMT.MyMTDestroy(Sender: TObject);
begin
  Form1.Label2.Caption
    := IntToStr(StrToInt(Form1.Label2.Caption) - 1);
  Application.Title := 'Server ' + Form1.Label2.Caption + ' V';
end;
```

Für gewöhnlich wird der Anwendungs-Server lediglich in der Task-Leiste angezeigt. Damit hier ebenfalls die Zahl der vorhandenen Verbindungen angezeigt wird, wird auch der Eigenschaft *Application.Title* diese Information zugewiesen.

Des Weiteren wird die Zahl der insgesamt durchgeführten Abfragen angezeigt.

```
procedure TMyMT.Query1AfterOpen(DataSet: TDataSet);
begin
  Form1.Label4.Caption
    := IntToStr(StrToInt(Form1.Label4.Caption) + 1);
end;
```

Beachten Sie bitte, dass diese *TQuery*-Instanz nun nicht mehr ganz so wie gewohnt reagieren wird. Wir werden beispielsweise mit Hilfe des Clients Datensätze ändern, ohne dass die Eigenschaft *RequestLive* den Wert *true* hat. Auch das Ereignis *OnAfterPost* wird dabei nicht ausgelöst. Um die im Server angezeigte Abfrage zu aktualisieren, muss die Funktion *Get_Query1* entsprechend ergänzt werden.

```
function TMyMT.Get_Query1: IProvider;
begin
  Query1.Close;
  Query1.Open;
```

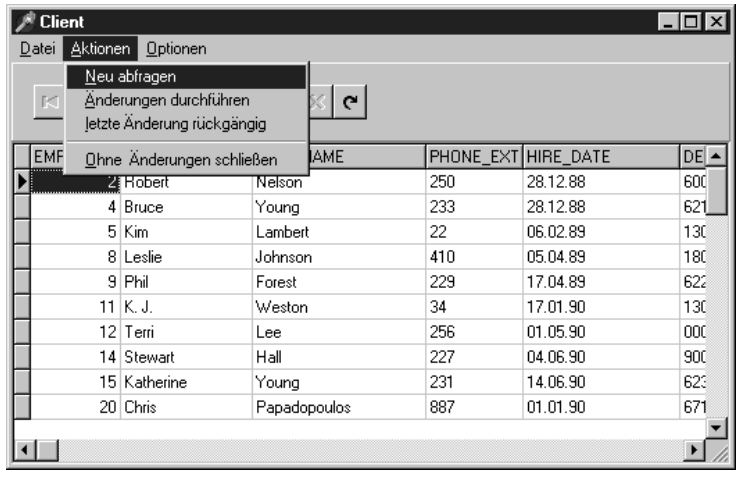
```
  if Form1.CheckBox1.Checked = true then
  begin
    DataModule2.Query1.Close;
    DataModule2.Query1.Open;
  end;
  Result := Query1.Provider;
end;
```

Für den Fall, dass eine automatische Aktualisierung nicht gewählt wird, soll diese auch mit einem Mausklick auf *Panel1* ausgelöst werden können.

```
procedure TForm1.Panel1Click(Sender: TObject);
begin
  with DataModule2.Query1 do
  begin
    Close;
    Open;
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnActivate := Panel1Click;
end;
```

16.1.2 Erstellung des Clients

Für den Client wird ein Formular gemäß der folgenden Abbildung bestückt:



Zusätzlich sind hier die Komponenten *RemoteServer1* (bei Delphi 4 *TDCOMConnection*), *ClientDataSet1* und *DataSource1* hinzuzufügen. Wählen Sie aus der Drop-Down-Liste der Eigenschaft *ServerName* den Namen Ihres Servers (den Namen des externen Datenmoduls) und setzen Sie die Eigenschaft *Connected* auf *true*. Nun wird automatisch der Anwendungs-Server gestartet.

Bei der Komponente *ClientDataSet1* setzen Sie die Eigenschaft *RemoteServer* auf *RemoteServer1* und die Eigenschaft *ProviderName* auf *Query1* (der Name der DataSet-Komponente im externen Datenmodul). Nun können Sie die Datenmenge öffnen, indem Sie die Eigenschaft *Active* auf *true* setzen. Nun müssen Sie nur noch *ClientDataSet1* über *DataSource1* mit dem Datengitter und dem Navigator verbinden, dann können Sie die Anwendung starten.

Mit einem Doppelklick auf das InterBase-Icon können Sie den Eigenschaftsdialog des InterBase-Servers öffnen. Hier werden Sie feststellen, dass hier nur eine Verbindung geführt wird (die zum Anwendungs-Server). Dass zum Anwendungs-Server zwei Verbindungen hergestellt sind (Delphi-IDE und laufende Anwendung), registriert der Datenbank-Server nicht. Sie können sogar über den Explorer weitere Instanzen der Anwendung starten, ohne dass InterBase sich darüber beschwert, dass die Zahl der Lizenzen nicht ausreicht.

Datenbestand ändern

Sie können jetzt schon nach Belieben neue Datensätze einfügen sowie bestehende ändern und löschen – diese Änderungen werden jedoch nicht in die Datenbank geschrieben, denn dazu ist Anweisung *ApplyUpdates* erforderlich.

```
procedure TForm1.nderungendurchfhren1Click(Sender: TObject);
begin
  ClientDataSet1.ApplyUpdates(-1);
  ClientDataSet1.Close;
  ClientDataSet1.Open;
end;
```

Des Weiteren soll dafür gesorgt werden, dass beim Schließen des Programms alle Änderungen zum Server geschickt werden. Können manche Änderungen nicht geschrieben werden, wird das Schließen des Programms verhindert. Der Anwender kann dann die Änderungen stornieren oder in eine Datei speichern.

```
procedure TForm1.FormCloseQuery(Sender: TObject;
var CanClose: Boolean);
begin
  if FAbbruch = false then
    if ClientDataSet1.ApplyUpdates(-1) <> 0
      then CanClose := false;
end;
```

Nun könnte es vorkommen, dass die Anwendung so »hängt«, dass der Variablenparameter *CanClose* immer auf *false* gesetzt wird. Um hier die Anwendung nicht mit dem Task-Manager beenden zu müssen, besteht die Möglichkeit, *FAbbruch* auf *true* zu setzen.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FAbbruch := false;
end;

procedure TForm1.Ohnenderungenschlieen1Click(Sender: TObject);
begin
  FAbbruch := true;
  Close;
end;
```

Mit dem Menüpunkt Neuabfragen werden die Daten neu gelesen, mit letzte Änderung rückgängig können die Änderungen verworfen werden.

```
procedure TForm1.Neuabfragen1Click(Sender: TObject);
begin
  ClientDataSet1.Close;
  ClientDataSet1.Open;
end;

procedure TForm1.letztenderungrckgngig1Click(Sender: TObject);
begin
  ClientDataSet1.UndoLastChange(false);
end;
```

Treten beim Aktualisieren der Daten Fehler auf, dann wird das Ereignis *OnReconcileError* ausgelöst. zur Anzeige der Fehler bietet Delphi einen fertigen Dialog in der Objektablage, wählen Sie dazu NEU|DIALOGE|FEHLER-DIALOG. Der Aufruf dieses Dialoges gestaltet sich dann recht einfach:

```
procedure TForm1.ClientDataSet1ReconcileError
  (DataSet: TClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Laden und Speichern der Daten

Sehr hilfreich ist die Fähigkeit von *TClientDataSet*, die Daten in einer Datei speichern zu können. Auf diese Weise kann ohne direkte Verbindung zum Datenbank-Server gearbeitet werden. Dies ist nicht nur dann nötig, wenn sich Datenbank-Server und/oder Netzwerk als wenig stabil erweisen. Es ist so auch möglich, die Daten auf ein Notebook zu spielen und unterwegs zu arbeiten.

Zum Laden und Speichern müssen lediglich die Methoden *LoadFromFile* beziehunsgweise *SaveToFile* aufgerufen werden. Nachdem die Daten geladen worden sind, können die Änderungen mit *ApplyUpdates* zum Server übertragen werden.

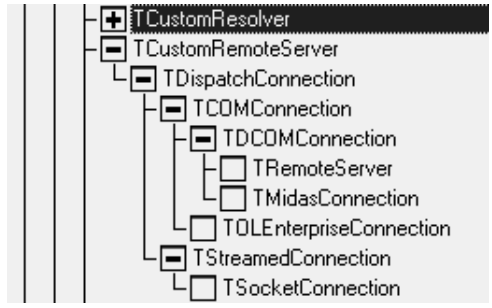
```
procedure TForm1.ffnen1Click(Sender: TObject);
begin
  if OpenDialog1.Execute
    then ClientDataSet1.LoadFromFile(OpenDialog1.FileName);
end;

procedure TForm1.Speichern1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
  begin
    if FileExists(SaveDialog1.FileName) then
    begin
      if MessageDlg('Datei existiert bereits. Überschreiben?',
          mtWarning, mbYesNoCancel, 0) = mrYes
        then ClientDataSet1.SaveToFile(SaveDialog1.FileName);
```

```
    end
    else ClientDataSet1.SaveToFile(SaveDialog1.FileName);
  end; {if SaveDialog1.Execute then}
end; {procedure TForm1.Speichern1Click}
```

16.1.3 Referenz TCustomRemoteServer

Zur Verbindung mit Anwendungs-Servern stellt Delphi insgesamt sechs Komponenten bereit:

- TDCOMConnection zur Verbindung über DCOM
- TCorbaConnection zur Verbindung über CORBA
- TSocketConnection zur Verbindung über TCP/IP
- TOLEEnterpriseConnection zur Verbindung über OLE Enterprise
- TMidasConnection zur Abwärtskompatibilität mit Delphi 3.01, Verbindung wahlweise über DCOM, TCP/IP oder OLE Enterprise
- TRemoteServer zur Abwärtskompatibilität mit Delphi 3, Verbindung über DCOM



Im Folgenden sollten die wichigsten Eigenschaften und Ereignisse dieser sechs Komponenten aufgeführt werden. Eine vollständige Referenz finden Sie in der Online-Hilfe.

Gemeinsame Eigenschaften und Ereignisse

- Connected (veröffentlicht, TCustomRemoteServer)

  ```
  property Connected: boolean;
  ```

  Um die Komponente *TRemoteServer* mit dem Anwendungs-Server zu verbinden, wird die Eigenschaft *Connected* auf *true* gesetzt. Ist der Anwendungs-Server noch nicht gestartet, dann wird dies nun getan.

- Diverse Ereignisse (Ereignisse, TCustomRemoteServer)

```
property OnAfterConnect(Sender: TObject);
property OnAfterDisconnect(Sender: TObject);
property OnBeforeConnect(Sender: TObject);
property OnBeforeDisconnect(Sender: TObject);
```

  Diese Ereignisse treten vor beziehungsweise nach dem Verbinden oder dem Lösen der Verbindung zum Anwendungs-Server auf.

TDCOMConnection

Von der Komponente *TDCOMConnection* werden die Komponenten *TDCOMConnection*, *TMidasConnection* und *TRemoteServer* abgeleitet.

- ServerName (veröffentlicht, TDispatchConnection)

```
property ServerName: string;
```

  Dieser Eigenschaft muss der Name des Anwendungs-Servers zugewiesen werden (Name des externen Datenmoduls). Unter den registrierten Servern kann der Name zur Entwurfszeit mit Hilfe einer Drop-Down-Liste ausgewählt werden. Wird kein Server-Name angegeben, muss sich der Server-Name aus den DCOM- oder OLE Enterprise-Konfigurationseinstellungen der Registrierdatenbank ableiten lassen.

- ServerGUID (veröffentlicht, TDispatchConnection)

```
property ServerGUID: string;
```

  Statt mit der Eigenschaft *ServerName* kann der Server auch mit der ID-Nummer *ServerGUID* spezifiziert werden. Die Eigenschaften *ServerName* und *ServerGUID* überschreiben sich gegenseitig.

- ComputerName (veröffentlicht, TDCOMConnection)

```
property ComputerName: string;
```

  Dieser Eigenschaft muss der Name des Rechners zugewiesen werden, auf dem der Anwendungs-Server läuft. Wird kein Server-Name angegeben, dann muss sich der Server-Name aus den DCOM- oder OLE Enterprise-Konfigurationseinstellungen der Registrierdatenbank ableiten lassen, oder der Anwendungs-Server wird auf demselben Rechner wie der Client gesucht.

- ObjectBroker (veröffentlicht, TDispatchConnection)

  ```
  property ObjectBroker: TCustomObjectBroker;
  ```

  Mit der Eigenschaft *ObjectBroker* kann ein Broker angegeben werden, mit dessen Hilfe die Verbindungskomponente zwischen mehreren Anwendungs-Servern wählen kann.

- LoginPrompt (veröffentlicht, TDispatchConnection)

  ```
  property LoginPrompt: Boolean;
  ```

  Legt fest, ob ein Login-Dialog angezeigt wird oder nicht.

- Diverse Ereignisse (Ereignisse, TDispatchConnection)

  ```
  property OnGetUsername(Sender: TObject; var Username: string);
  property OnLogin(Sender: TObject; Username, Password: string)
  ```

  Das Ereignis *OnGetUsername* tritt unmittelbar vor der Anzeige des Remote-Anmeldedialogfelds ein. Das Ereignis *OnLogin* tritt ein, nachdem eine Verbindung zu einem Remote-Anwendungs-Server aufgebaut wurde.

TCorbaConnection

Damit eine Verbindung über CORBA aufgebaut werden kann, muss irgendwo im lokalen Netzwerk (mindestens) ein Corba-Broker aktiviert sein. Mit Delphi 4 wird der *Visibroker Smart Agent* mitgeliefert, der diese Aufgabe übernehmen kann.

- RepositoryID (veröffentlicht, TCorbaConnection)

  ```
  property RepositoryId: string;
  ```

  Mit Hilfe der Eigenschaft *RepositoryID* werden der Anwendungs-Server und das von ihm zur Verfügung gestellte Objekt spezifiziert. Eine gültige Eingabe lautet beispielsweise *Server/TMeinCorbaServer*, wobei *Server* der Anwendungs-Server ist und *TMeinCorbaServer* das von diesem veröffentlichte Objekt. Um die momentan zur Verfügung stehenden Objekte zu ermitteln, können Sie das Programm *Visibroker Smart Finder* verwenden.

- HostName (veröffentlicht, TCorbaConnection)

  ```
  property HostName: string;
  ```

  Im Regelfall verwendet *TCorbaConnection* den ersten Server, der die gewünschte Schnittstelle besitzt. Soll jedoch ein bestimmter Server verwendet werden, dann ist *HostName* entsprechend zu setzen.

TSocketConnection

Um eine Verbindung über TCP/IP aufzubauen, muss auf dem Server entweder *SCKTSRVR.EXE* oder *SCKTSRVC.EXE* (Windows NT-Service) betrieben werden.

- ServerName (veröffentlicht, TDispatchConnection)

  ```
  property ServerName: string;
  ```

  Dieser Eigenschaft muss der Name des Anwendungs-Servers zugewiesen werden (Name des externen Datenmoduls). Unter den registrierten Servern kann der Name zur Entwurfszeit mit Hilfe einer Drop-Down-Liste ausgewählt werden. Wird kein Server-Name angegeben, muss sich der Server-Name aus den DCOM- oder OLE Enterprise-Konfigurationseinstellungen der Registrierdatenbank ableiten lassen.

- ServerGUID (veröffentlicht, TDispatchConnection)

  ```
  property ServerGUID: string;
  ```

  Statt mit der Eigenschaft *ServerName* kann der Server auch mit der ID-Nummer *ServerGUID* spezifiziert werden. Die Eigenschaften *ServerName* und *ServerGUID* überschreiben sich gegenseitig.

- Host (veröffentlicht, TSocketConnection),

  Adress (veröffentlicht, TSocketConnection)

  ```
  property Host: String;
  property Adress: String;
  ```

  Der Server muss entweder über die IP-Adresse oder über den Host-Namen spezifiziert werden.

- Port (veröffentlicht, TSocketConnection)

  ```
  property Port: Integer;
  ```

  Mit der Eigenschaft *Port* wird der Port festgelegt, über den der Socket-Dispatcher Verbindungsanforderungen des Clients empfängt. Per Voreinstellung hat *Port* den Wert 211.

- ObjectBroker (veröffentlicht, TDispatchConnection)

  ```
  property ObjectBroker: TCustomObjectBroker;
  ```

  Mit der Eigenschaft *ObjectBroker* kann ein Broker angegeben werden, mit dessen Hilfe die Verbindungskomponente zwischen mehreren Anwendungs-Servern wählen kann.

- LoginPrompt (veröffentlicht, TDispatchConnection)

  ```
  property LoginPrompt: Boolean;
  ```

  Legt fest, ob ein Login-Dialog angezeigt wird oder nicht.

- Diverse Ereignisse (Ereignisse, TDispatchConnection)

  ```
  property OnGetUsername(Sender: TObject; var Username: string);
  property OnLogin(Sender: TObject; Username, Password: string)
  ```

  Das Ereignis *OnGetUsername* tritt unmittelbar vor der Anzeige des Remote-Anmeldedialogfelds ein. Das Ereignis *OnLogin* tritt ein, nachdem eine Verbindung zu einem Remote-Anwendungs-Server aufgebaut wurde.

TOLEEnterpriseConnection

Die Eigenschaften und Ereignisse von *TOLEEnterpriseConnection* gleichen weitgehend *TDCOMConnection*. Beachten Sie bitte, dass Sie bei der Verwendung von OLE Enterprise einen entsprechenden Broker installieren müssen.

- BrokerName (veröffentlicht, TOLEEnterpriseConnection)

  ```
  property BrokerName: string;
  ```

  Diese Eigenschaft spezifiziert den Business Object Broker, auf dem der Anwendungs-Server läuft.

16.1.4 Referenz TProvider

Mit Hilfe der Komponente *TProvider* können Daten von anderen Datenmengenkomponenten weiterverbreitet werden. Wird nicht explizit eine *TProvider*-Instanz erzeugt, dann wird automatisch eine erstellt.

Die Komponente *TProvider* ist auf *TTable*, *TQuery* und *TStoredProc* beschränkt, während *TDataSetProvider* auch *TClientDataSet*-Datenmengen referenziert. *TProvider* ist von *TDataSetProvider* abgeleitet.

Auch hier soll nur eine Teilmenge der vorhandenen Eigenschaften, Methoden und Ereignisse besprochen werden. Der Rest kann in der Online-Hilfe nachgeschlagen werden.

- DataSet (veröffentlicht, TDataSetProvider)

  ```
  property DataSet: TDBDataSet;
  ```

  Mit der Eigenschaft *TDataSet* wird gewählt, von welcher Datenmengenkomponente die Daten bezogen werden.

- Constraints (veröffentlicht, TDataSetProvider)

```
property Constraints: WordBool;
```

  Hat *Constraints* den Wert *true*, dann gelten die Beschränkungen seitens der Datenbank (beispielsweise NOT NULL) auch für die Client-Datenmenge.

- ApplyUpdates (Methode, TProvider)

```
function ApplyUpdates(Delta: OleVarant; MaxErrors: Integer;
    out ErrorCount: Integer): OleVariant;
```

  Mit *ApplyUpdates* werden die Aktualisierungen in die Datenmengenkomponente geschrieben.

- OnUpdateError (Ereignis, TProvider)

```
property OnUpdateError(DataSet: TClientDataSet; E: EUpdateError;
    UpdateKind: TUpdateKind; var Response: TResolverResponse);
```

  Tritt beim Eintragen der Daten ein Fehler auf, wird das Ereignis *OnUpdateError* ausgelöst.

16.2 Corba

Corba (*Common Object Request Broker Architecture*) ist eine Spezifikation, um einfacher verteilte Anwendungen schreiben zu können. Corba wird von der OMG (Object Management Group) entwickelt, einem Zusammenschluss fast aller großen Softwarefirmen. Welche Vorteile hat nun Corba?

- Corba ist plattformunabhängig. Wenn Sie ein Netzwerk haben, an dem ein paar Unix-Server, viele Windows-PCs und einige paar Macs hängen, dann können Sie in diesem Netz verteilte Anwendungen auf Corba-Basis betreiben.
- Corba ist einfach zu verwalten. Sie betreiben irgendwo im Netz einen Client, an einer beliebigen anderen Stelle einen Server und an dritter Stelle den Broker. Konfigurieren müssen Sie nichts. Wenn beispielsweise der Server-Rechner ausfällt, auf dem der Anwendungs-Server installiert ist, dann betreiben Sie den Anwendungs-Server einfach auf einem anderen Rechner, ohne auch nur einen Client umkonfigurieren zu müssen.
- Corba ist der wichtigste Middleware-Standard. Hunderte an der OMG beteiligte Firmen stellen sicher, dass Corba nicht so schnell vom Markt verschwindet, auch wenn *Microsoft* versucht, DCOM dagegen zu positionieren.

- Über Corba können alle Arten von Objekten miteinander kommunizieren. Diese Technologie ist nicht auf mehrschichtige Datenbankanwendungen beschränkt, auch wenn wir andere Anwendungsmöglichkeiten in diesem Buch nicht besprechen wollen.

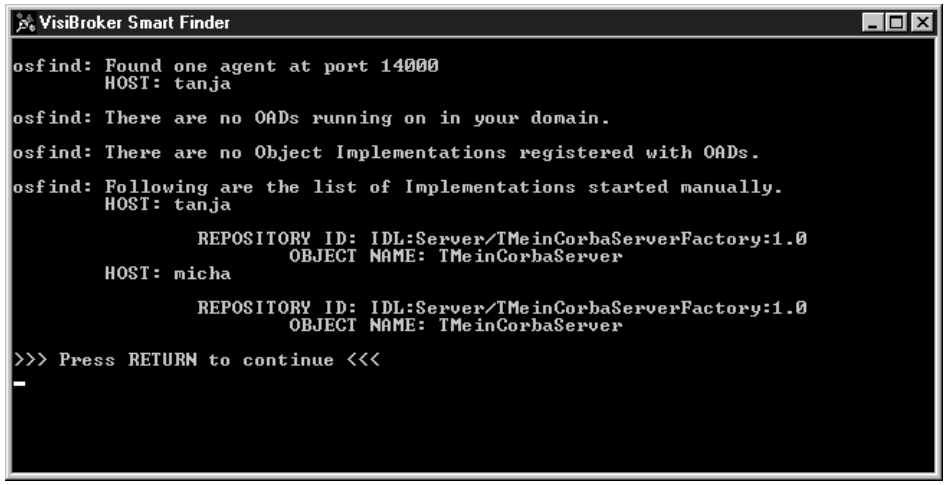
16.2.1 Der Broker

Der Broker (zu deutsch *Makler*) sorgt dafür, dass Client- und Server-Objekte miteinander kommunizieren können, auch wenn sie (anfangs) jeweils nicht wissen, auf welchem Rechner die jeweils andere Seite liegt.

Die Enterprise-Version von Delphi wird zusammen mit dem *Visibroker Smart Agent* ausgeliefert, den Sie starten sollten, bevor Sie sich an die Entwicklung von Corba-Anwendungen machen.

Der Broker kann dabei irgendwo im lokalen Netzwerk betrieben werden. Um zu überprüfen, ob auch tatsächlich eine Verbindung zum Broker besteht, können Sie den *Visibroker Smart Finder* verwenden.

Im Beispiel der folgenden Abbildung würde der Broker auf dem Host *Tanja* laufen, während jeweils eine Server-Anwendung auf den Hosts *Tanja* und *Micha* läuft.

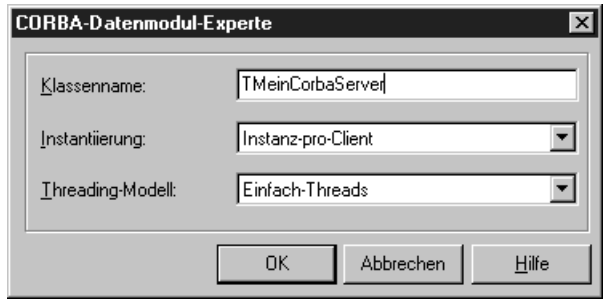


16.2.2 Einen Corba-Anwendungs-Server erstellen

Aus Gründen der Bequemlichkeit wollen wir unseren Corba-Server auf der Grundlage des DCOM-Servers aus Kapitel 16.1 erstellen.

Dazu kopieren wir alle Dateien in ein anderes Verzeichnis, öffnen das Projekt und entfernen das Remote-Datenmodul. Anschließend können die Unit DM_2 und die Typenbibliothek aus dem Verzeichnis gelöscht werden.

Um ein neues Datenmodul einzufügen, wählen Sie DATEI|NEU|MULTI-TIER|CORBA-DATENMODUL.



Hier müssen Sie einen Klassennamen eingeben, der bei jedem neuen Projekt anders sein sollte. Die Optionen *Instantiierung* und *Threading-Modell* lassen Sie auf den voreingestellten Werten. Mit OK schließen Sie den Vorgang ab.

In das neue Remote-Datenmodul fügen Sie nun eine *TQuery*-Komponente ein und ergänzen dieselben Anweisungen wie in Kapitel 16.1.1. Anschließend können Sie den Server starten. (Wenn lange Zeit kein Formular auf dem Bildschirm erscheint, läuft vermutlich der Broker nicht.)

16.2.3 Einen Corba-Client erstellen

Auch den Corba-Client wollen wir auf Grundlage unseres DCOM-Clients erstellen. Kopieren Sie wieder die Dateien in ein anderes Verzeichnis, öffnen Sie das Projekt und entfernen Sie die *TDCOMConnection*-Instanz (beziehungsweise *RemoteServer1*, wenn Sie das Projekt mit Delphi 3 erstellt hatten). Als Verbindungskomponente muss nun allerdings *TCorbaConnection* verwendet werden.

Stellen Sie sicher, dass der Server und der Broker irgendwo im Netzwerk laufen (es darf auch auf demselben Rechner sein), bevor Sie fortfahren.

Zunächst wird die Eigenschaft *RepositoryID* der Komponente *CorbaConnection1* auf *Server/TMeinCorbaServer* gesetzt. Damit spezifizieren Sie den Namen des Servers (hier *Server.Exe*, der Name sollte von Mal zu Mal variieren) und des Objektes (hier des Datenmoduls *TMeinCorbaServer*).

Setzen Sie nun die Eigenschaft *Connected* auf *true*. Treten dabei keine Fehlermeldungen auf, dann steht die Corba-Verbindung. Anschließend setzen Sie noch die Eigenschaft *RemoteServer* von *ClientDataSet1* auf *CorbaConnection1* und öffnen Sie dann die Datenmenge. Nun ist der Corba-Client fertig.

17 Die To-Do-Liste

Leser haben mich immer mal wieder gebeten, als Beispiel nicht irgendwelche trivialen Projekte vorzustellen, sondern ein »richtig großes« Projekt. Leider kann ich diesem Wunsch nicht so ganz nachkommen: Zum einen müsste man »richtig große« Projekte mehrbändig beschreiben – wer sollte denn das kaufen, geschweige denn lesen. Zum anderen stecken in solchen Projekten mehre Mannjahre Arbeit. Mal so als Beispiel kann ich solche Projekte nicht erstellen, und die großen Projekte, an denen ich bislang gearbeitet habe, waren alle keine Open-Source-Geschichten.

So fällt auch die To-Do-Liste in den Bereich der kleinen Projekte, hat mit 25 beteiligten Units aber die »Trivialitätsschwelle« bereits überschritten. Als »kleines Schmankerl« wollen wir dieses Projekt gleich dreimal erstellen:

- Mit der BDE, beschrieben in Kapitel 18.
- Mit den InterBase-Komponenten, beschrieben in Kapitel 19.
- Und mit dbExpress, beschrieben in Kapitel 20.

Dabei soll das Projekt mit der BDE entwickelt und dort auch ausführlich beschrieben werden. Für die anderen beiden Projekte kopieren wir dann den bestehenden Code und ändern ihn entsprechend ab.

17.1 Was ist die To-Do-Liste?

Software wird für gewöhnlich nicht im »luftleeren Raum« entwickelt: Neben den Leuten, welche den Code schreiben, gibt es Fachabteilungen, die Vorgaben machen, ausgewählte Anwender, welche die Programme testen, und über alledem wachen die Vorgesetzten, die vor allem wissen möchte, wann dieses und jenes voraussichtlich fertig wird.

Wird die Kommunikation zwischen den Beteiligten per E-Mail abgewickelt, dann werden im Laufe der Zeit ziemlich viele E-Mails verschickt, erfahrungsgemäß an lieber zu viele als an zu wenige Beteiligte, und die Beteiligten müssen schon recht viel Disziplin aufbringen, damit der Überblick nicht verloren geht.

Wesentlich einfacher geht die Sache mit der To-Do-Liste. Die Idee für dieses Programm stammt von Thomas Plamper, der ein ähnliches Programm für die Mannheimer Versicherung erstellt hat. Die hier vorliegende Version wurde auf ein

Client-Server-System umgestellt und die Datenbank sowie die Benutzeroberfläche wurden neu (vor allem universeller) konzipiert.

Arbeitsablauf mit der To-Do-Liste

Die To-Do-Liste sieht den folgenden Arbeitsablauf vor:

- Für eine Fehlermeldung oder eine neue Anforderung wird ein Eintrag in der To-Do-Liste erstellt. Dieser Eintrag ist gleich einem Projekt zugeordnet. Am Anfang hat der Eintrag den Status E (wie *erstellt*).
- Normalerweise werden Einträge von Personen erstellt, die mit der Arbeitsteilung zwischen den einzelnen Programmierern nicht vertraut sind. Deshalb kommt als nächster Schritt die Zuweisung: Eine damit beauftragte Person legt fest, welcher Eintrag von welcher Funktion erledigt wird. Eine Funktion kann ein einzelner Programmierer sein, es kann sich aber auch um eine Gruppe oder eine Abteilung handeln.

  Bei dieser Gelegenheit kann gleich noch die Priorität des Eintrags festgelegt werden. Der Eintrag hat nun den Status Z (wie *zugewiesen*).
- Die Entwickler schauen sich nun die Einträge an, die ihrer Funktion zugewiesen sind. Sind sie einer Funktion zugewiesen, welche mit dieser Aufgabe eigentlich nicht betraut ist, dann werden sie – entsprechende Benutzerrechte vorausgesetzt – der zuständigen Funktion zugewiesen, oder es wird Rücksprache mit der Person gehalten, welche den Eintrag zugewiesen hat.

  Korrekt zugewiesene Aufträge werden nun zwischen den Mitgliedern einer Funktion aufgeteilt, indem die Person, die den Eintrag bearbeiten möchte/muss, diesen auf den Status A (wie *akzeptiert*) setzt.
- Sobald mit der Bearbeitung des Eintrags begonnen wird, setzt man diesen auf den Status B (wie *in Bearbeitung*).
- Sollte es erforderlich sein, die Bearbeitung zu unterbrechen, beispielsweise weil ein Auftrag höherer Priorität zu erledigen ist, dann bekommt der Eintrag den Status U (wie *unterbrochen*).
- Ist die Bearbeitung des Eintrags abgeschlossen, erhält dieser den Status F (wie *fertig*). Je nachdem, wie die Rechte vergeben sind, setzen die Entwickler diesen Status selbst, oder er wird von der Qualitätsprüfung gesetzt, wenn diese der Ansicht ist, dass die Sache wirklich erledigt ist.

Weitere Features der To-Do-Liste

Um die Arbeit produktiver zu gestalten, bietet die To-Do-Liste die folgenden Möglichkeiten:

- An jeden Eintrag können mehrere Dateien angehängt werden. Das können die Daten sein, mit denen gearbeitet wurde, irgendwelche zu Debug-Zwecken gespeicherte Dateien, Ausdrucke im pdf-Format, Protokolle von irgendwelchen Besprechungen, aus denen sich die betreffende Anforderung ergibt oder was auch immer.
- Jede Statusänderung kann kommentiert werden. So kann beispielsweise begründet werden, warum ein Eintrag gerade dieser Funktion zugewiesen oder warum eine Unterbrechung der Arbeit vorgenommen wurde. Es gibt auch »Statusänderungen«, die den Status nicht ändern und lediglich den Zweck haben, weitere Informationen hinzuzufügen.
- Jeder Anwender mit Lese-Recht kann sich die Liste der Statusänderungen anschauen und so mitverfolgen, welchen Fortschritt die Erledigung des Eintrages macht.
- Die Liste der Einträge kann gefiltert werden. So wird man meist auf die bereits fertig gestellten Einträge verzichten wollen. Die gesetzten Filter werden gespeichert, sie sind entweder öffentlich, oder dem Benutzer zugeordnet, der sie erstellt hat. Auf diese Weise kann jeder Benutzer aus einer Menge von Abfragen wählen, die auf ihn zugeschnitten ist. Zur Erstellung dieser Abfragen gibt es einen Assistenten.
- Und außerdem kann sowohl jede Liste als auch der einzelne Eintrag als Zusammenfassung oder Detailansicht ausgedruckt werden.

17.2 Die ersten Schritte mit der To-Do-Liste

Wir wollen uns die Handhabung der To-Do-Liste nun ein wenig genauer ansehen.

17.2.1 Erstellen der Datenbank

Die Datenbank wird mittels eines SQL-Scripts (*To-Do_script.sql*) erstellt. Rufen Sie zu diesem Zweck das Programm *IBConsole* auf und loggen Sie sich in den gewünschten Server ein.

Mit dem Menüpunkt TOOLS|INTERACTIVE SQL rufen Sie das entsprechende Programm auf. Dort wählen Sie nun *Query|Load Script* und laden die Datei *To-Do_script.sql*.

Mit *Query|Execute* führen Sie diese Anweisungen nun aus. Das SQL-Script werden wir uns in Kapitel 17.3 noch näher ansehen. Momentan soll es uns genügen, wenn es fehlerfrei durchläuft und wir eine funktionierende Datenbank haben.

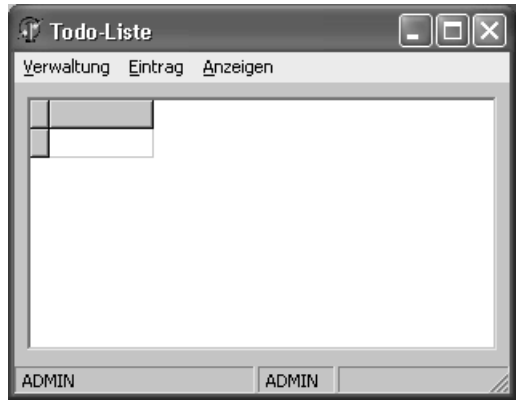
17.2.2 Anmelden

Starten wir nun das Programm *p_To-Do.exe*. Es ist dabei egal, ob Sie die BDE-, die IBX- oder die dbExpress-Version verwenden.

Zunächst werden Sie mit einem Passwort-Dialog konfrontiert:

Der erste Benutzer, der Administrator, wurde schon vom SQL-Script angelegt und verwendet als Passwort einen leeren String. Sobald die To-Do-Liste eingerichtet ist, sollte dieses Passwort geändert werden.

Nun wird die Hauptmaske angezeigt. Oben haben wir das Menü, in der Mitte die Liste der Einträge und unten die Statusleiste.



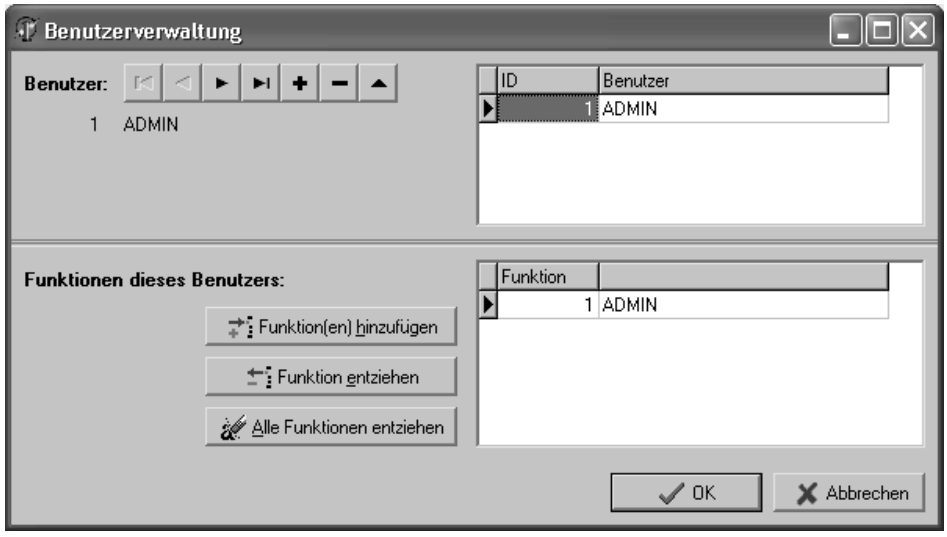
In der Liste der Einträge ist noch nicht viel zu sehen. Dies liegt nicht nur daran, dass es noch keine Einträge gibt, sondern vor allem daran, dass noch keine Abfragen gespeichert sind, die ausgeführt werden können. Der Anwender kann eine Abfrage wählen, die beim Programmstart ausgewählt werden soll, aber dazu muss erst einmal eine vorhanden sein.

In der Statusleiste wird links der angemeldete Benutzer angezeigt, in diesem Fall *ADMIN*. Daneben wird dann, wenn der Benutzer Administratorrechte hat, *ADMIN* angezeigt – in der Regel wird der Benutzer *ADMIN* nach dem Einrichten des Programms nicht mehr verwendet, sondern die Benutzer mit Administratorrechten melden sich mit ihrem eigenen Namen an.

Rechts wird der Name der zuletzt ausgeführten Abfrage angezeigt. Nun haben wir allerdings noch keine Abfrage, also kann auch keine ausgeführt werden – folglich bleibt dieses Feld erst einmal leer.

17.2.3 Anlegen von Benutzern und Funktionen

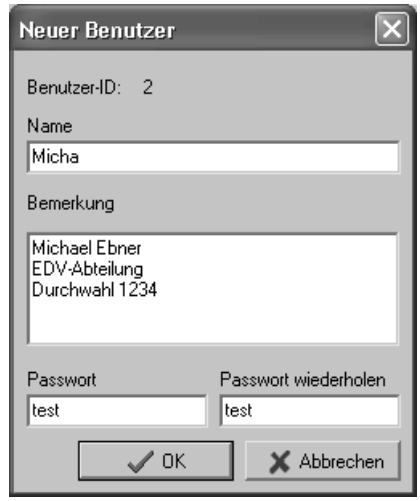
Wir wollen nun einige Benutzer und Funktionen anlegen. Wählen Sie dazu VERWALTUNG|BENUTZER.



Oben sehen wir die eingerichteten Benutzer, unten die Liste der Funktionen, die der jeweilige Benutzer wahrnimmt. Momentan gibt es hier nur den *ADMIN*.

Auch wenn es nicht danach aussieht: Die Daten des Benutzers links oben werden nicht mit *TDBText*, sondern mit *TDBEdit* und *TDBMemo* angezeigt. Die Bearbeitung über diese Komponenten wurde unterbunden, Sie können aber die Daten über die Zwischenablage kopieren.

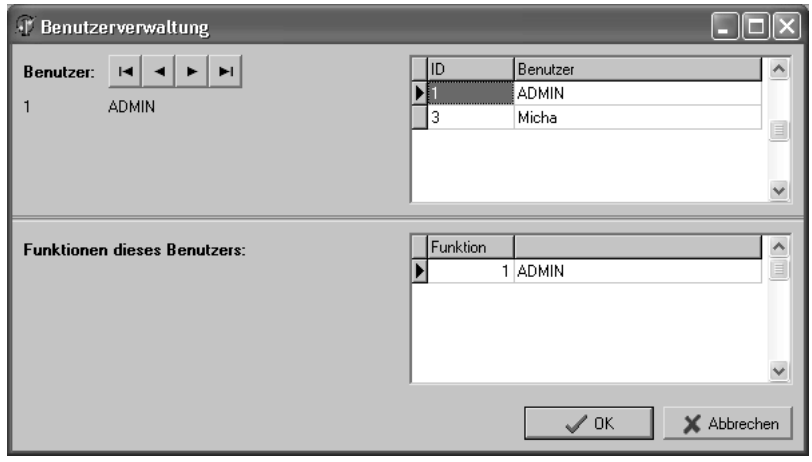
Klicken wir nun auf den Button + im Navigator und öffnen einen Dialog zum Einrichten neuer Benutzer:



Geben Sie hier nun den Benutzernamen und die Passwörter ein. Zu jedem Benutzer kann noch eine Bemerkung eingegeben werden. Dort können Daten wie die Abteilung oder die Telefonnummer gespeichert werden.

Solange ein Benutzer mit Administratorrechten angemeldet ist, wird das Passwort im Klartext angezeigt. Ändert dagegen ein Benutzer sein Passwort ab, wird es mit *-Zeichen dargestellt. Wir wollen das gleich mal testen und melden uns mit dem gerade angelegten Benutzer an.

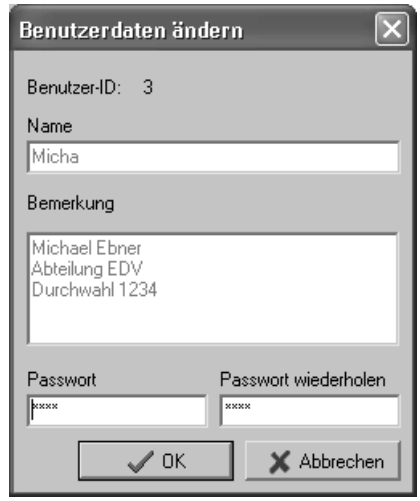
Zunächst fällt uns auf, dass wir nun an den Benutzern gar keine Änderungen durchführen können. Im Navigator links oben fehlen die »interessanten« Buttons und auch die Buttons zum Vergeben oder Entziehen von Funktionen sind ausgeblendet.



Eigentlich würden die Designregeln von Windows vorsehen, diese Buttons als nicht verfügbar anzuzeigen. Dies führt jedoch erfahrungsgemäß dazu, dass die Benutzer nach einem Weg suchen, diese Buttons zu aktivieren. Nicht, dass es einen solchen Weg geben würde, aber man muss seine Anwender ja nicht zu unnötiger Arbeit verleiten.

Wird jedoch der Benutzer angezeigt, der gerade angemeldet ist, dann wird ihm der Button *Passwort ändern* angeboten.

Wird nun dieser Button gewählt, dann erscheint der bereits bekannte Dialog:



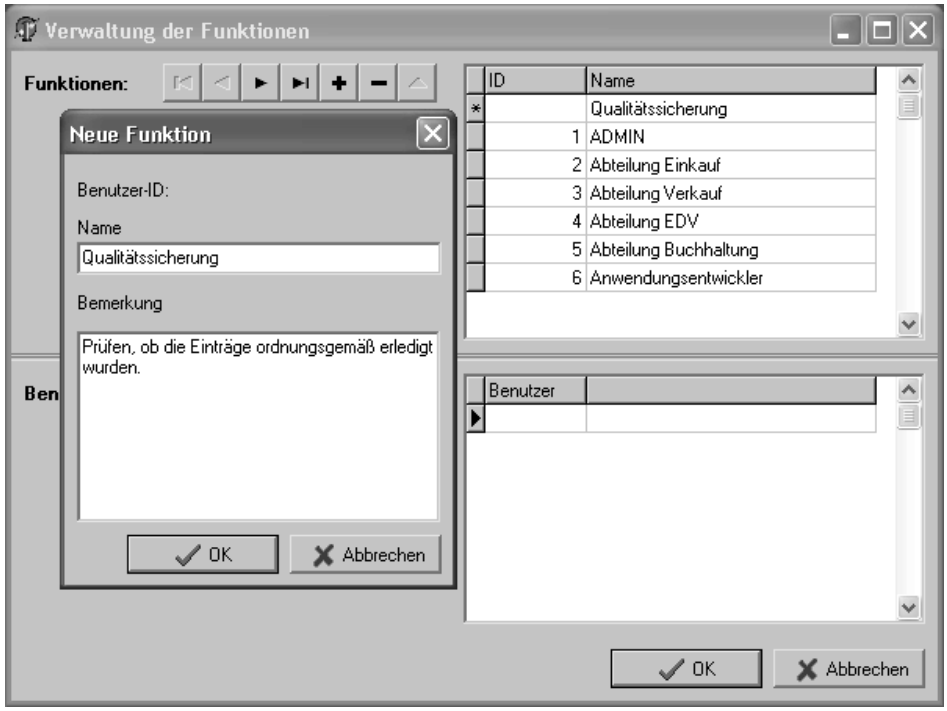
Nun wird allerdings die Benutzer-ID angezeigt, Name und Bemerkung können nicht geändert werden, und das Passwort (und natürlich auch dessen Wiederholung) wird nicht im Klartext angezeigt.

(Das Programm kann leicht so abgeändert werden, dass die Benutzer auch ihre eigenen Bemerkungen ändern können. Wird ein Benutzer beispielsweise versetzt, dann kann er selbst seine neue Abteilung und Durchwahl eingeben und muss dazu nicht einen Anwender mit Administratorrechten bemühen.)

Bevor wir uns den Funktionen zuwenden, geben wir erst noch ein paar Anwender ein. Achten Sie darauf, am Schluss den Dialog mit *Ok* und nicht mit *Abbrechen* zu schließen, weil sonst alle Änderungen verworfen werden.

Funktionen anlegen

Zum Anlegen von Funktionen wählen wir den Menüpunkt VERWALTUNG|FUNKTIONEN. Dieser Dialog ist genauso aufgebaut wie der zur Eingabe der Benutzer, und der Dialog zum Einfügen oder Ändern der Daten ist derselbe wie bei dem des Benutzers, lediglich die Passworteingabe wurde ausgeblendet. Geben Sie hier ein paar der abgebildeten Funktionen ein:



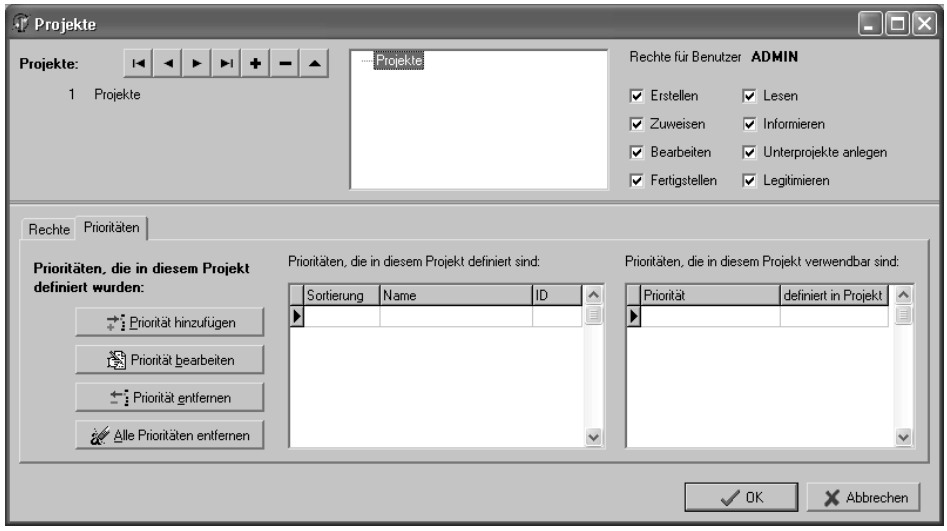
Nun wollen wir einer Funktion ein paar Benutzer zuweisen. Zu diesem Zweck rufen wir den Button *Benutzer hinzufügen* auf:

Es erscheint eine Liste aller eingetragenen Benutzer. Mit Hilfe der STRG-Taste können mehrere Benutzer gleichzeitig ausgewählt werden, so dass diese auch gleichzeitig der entsprechenden Funktion zugewiesen werden.

Ebenso können wir aus der Benutzerverwaltung heraus einem Benutzer mehrere Funktionen auf einmal zuweisen:

17.2.4 Verwaltung der Projekte

Mit VERWALTUNG|PROJEKTE öffent man die Projektverwaltung:



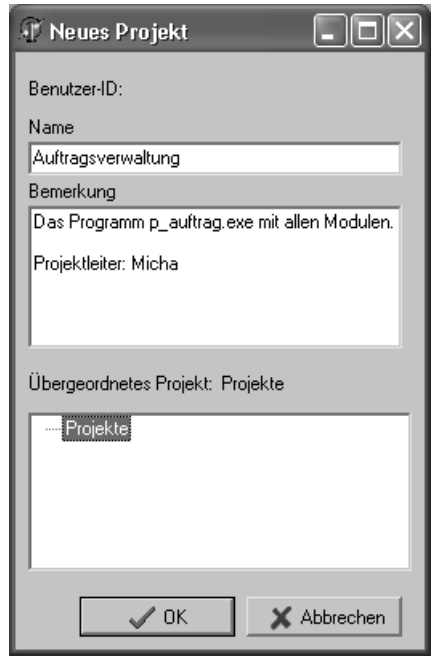
Damit die Übersicht nicht verloren geht, ist jeder Eintrag einem Projekt zugeordnet. Die Liste der Projekte ist dabei hierarchisch aufgebaut, ein Projekt kann somit Unterprojekte haben, die ihrerseits wieder Unterprojekte haben können, die ihrerseits ... Der »Stammvater« aller Projekte ist das Projekt *Projekte*.

An diese Projekte gekoppelt sind die Rechte der einzelnen Funktionen sowie die Prioritäten. Dabei sind auch die Rechte hierarchisch aufgebaut: Ein Benutzer, der ein bestimmtes Recht bezüglich eines Projektes hat, hat dieses Recht auch für alle dessen Unterprojekte.

Da die Funktion *ADMIN* sämtliche Rechte für *Projekte* hat, haben Administratoren sämtliche Rechte in allen Projekten.

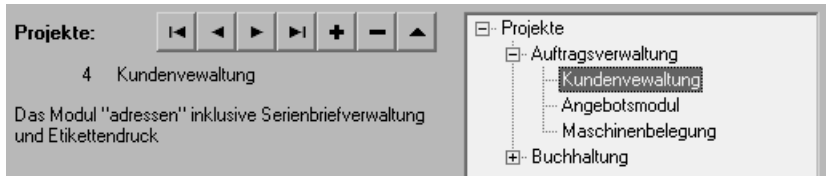
Anlegen von Projekten

Wir wollen nun einige Projekte anlegen und bedienen uns dabei des Buttons +.



Ein Projekt benötigt einen Namen sowie ein übergeordnetes Projekt. Das übergeordnete Projekt kann aus einer Baumansicht ausgewählt werden – wobei es natürlich Voraussetzung für einen Baum ist, dass schon Projekte angelegt sind. Optional kann dann noch eine Bemerkung eingegeben werden.

Legen Sie nun ein paar Projekte an.



Wählen Sie nun das Projekt *Auftragsverwaltung* und editieren es. Sie können nicht nur den Namen und die Bemerkung ändern, sondern auch das übergeordnete Projekt. Auch hier wird die Baumansicht der Projekte angezeigt, allerdings sind hier das Projekt selbst und alle untergeordneten Projekte ausgenommen, weil eine solche Zuweisung den korrekten Aufbau des Baumes unterbinden würde.

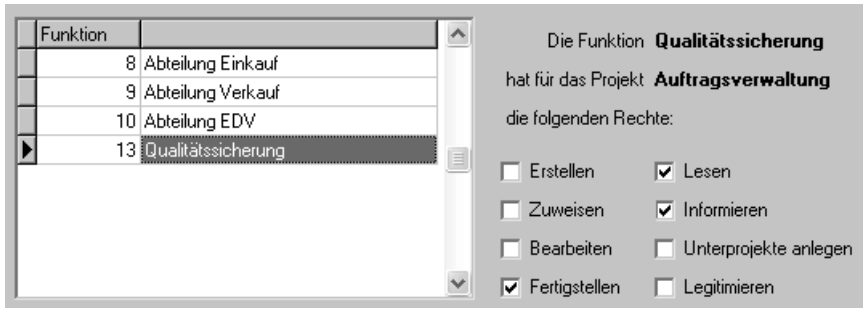
Rechte vergeben

Wählen Sie jetzt das Projekt *Auftragsverwaltung* und klicken Sie dann auf den Button *Funktion(en) hinzufügen*; dieser befindet sich auf der Registerseite *Rechte*.

Aus der Liste können Sie nun mehrere Funktionen wählen (wieder mit der STRG-Taste), mit den CheckBoxen werden die Rechte gewählt, die den ausgewählten Funktionen zugewiesen werden.

Dadurch, dass die Rechte den Funktionen und nicht den Benutzern zugewiesen werden, lassen sich Ereignisse wie die Fluktuation im Bereich der Mitarbeiter besonders unbürokratisch verwalten.

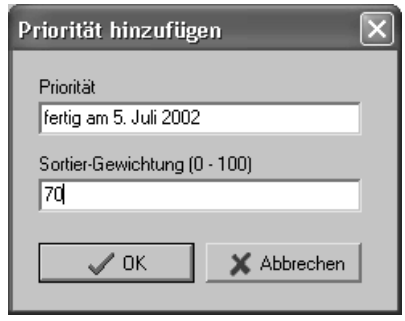
Auf dem Grid werden nun die einzelnen Funktionen, an die Rechte vergeben wurden, sowie deren Rechte angezeigt.



Prioritäten erstellen

Den Einträgen in der Liste können verschiedene Prioritäten zugewiesen werden, beispielsweise *niedrig*, *mittel* oder *hoch*. Diese Prioritäten sind *Projekten* zugeordnet, genauso wie *nicht zugewiesen* für alle Einträge gilt, bei denen die Priorität erst noch festgelegt werden soll.

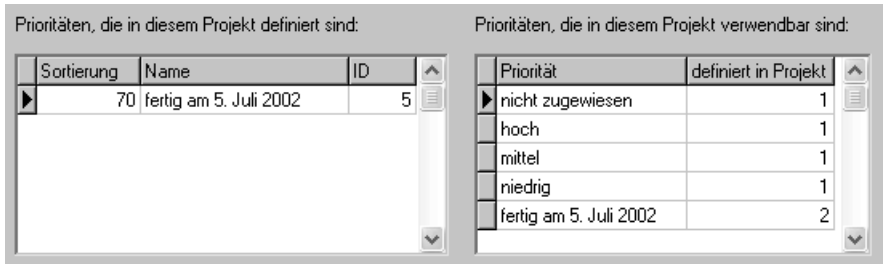
Nun besteht die Möglichkeit, auf der Ebene eines einzelnen Projektes (und somit auch für alle dessen Unterprojekte) eigene Prioritäten zu vergeben, wobei dabei auch Fertigstellungstermine möglich sind. Dabei wird das betreffende Projekt ausgewählt und der Button *Priorität hinzufügen* angeklickt.



Eine Priorität hat neben einem Namen auch einen ganzzahligen Wert, nach dem die Einträge sortiert werden können – je höher dieser Wert, desto höher auch die Priorität und desto weiter vorne in der Liste erscheint der Eintrag.

Prioritäten lassen sich auch ändern. Insbesondere bei Fertigstellungsterminen macht es Sinn, diesen beim Näherrücken einen immer höheren Sortierwert zuzuweisen.

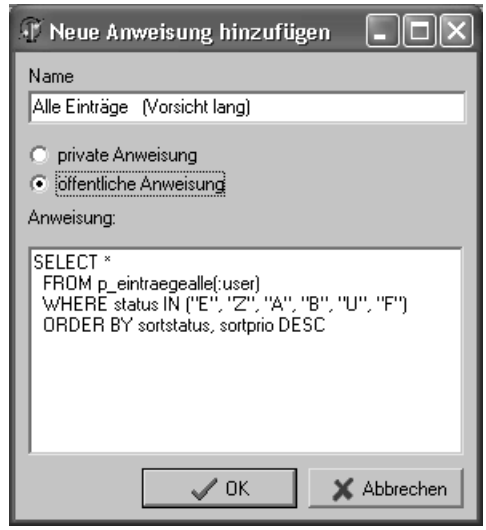
Zum jeweiligen Projekt werden nicht nur die Prioritäten angezeigt, die dort definiert wurden, sondern auch alle aus übergeordneten Projekten, sprich alle, die sich dort verwenden lassen.



17.2.5 Eine Abfrage generieren

Eigentlich wäre es jetzt an der Zeit, einen Eintrag zu erstellen. Allerdings könnten wir uns noch keine Liste der Einträge erstellen lassen, so dass wir keine Erfolgskontrolle über unser Tun hätten. Also wollen wir uns erst mit den Abfragen und dann mit den Einträgen beschäftigen.

Wählen Sie zu diesem Zweck den Menüpunkt ANZEIGEN|ABFRAGE ERZEUGEN. Dort sehen wir eine (bislang noch leere) Liste der Abfragen. Zum Erstellen einer neuen Abfrage wählen wir *Anweisung hinzufügen*.



Eine Abfrage benötigt vor allem ein SQL-Statement als Anweisung. Mit dem Namen kann die Abfrage dann aus dem Menü gewählt werden, zudem kann gewählt werden, ob es sich um eine öffentliche oder private Anweisung handelt. Öffentliche Anweisungen sind für alle Benutzer verfügbar, private nur für den, der sie erstellt hat.

```
SELECT *
  FROM p_eintraegealle(:user)
  ORDER BY sortstatus, sortprio DESC
```

Mit der oben stehenden SQL-Anweisung generieren wir eine Liste aller Einträge. Wird die To-Do-Liste länger verwendet, dann sammeln sich viele erledigte Einträge an, was zu einer entsprechend großen Ergebnismenge führt. Daher sollte im Titel auf dieses Problem hingewiesen werden.

Mit Hilfe des Parameters *user* prüft die STORED PROCEDURE *p_eintraegealle*, ob der Benutzer für den jeweiligen Eintrag auch das Leserecht hat – nur solche Einträge werden in die Ergebnismenge aufgenommen.

Als zweite öffentliche Anweisung erstellen wir eine Liste der noch nicht abgeschlossenen Einträge, wofür die STORED PROCEDURE *p_eintraegeunfertig* erstellt worden ist.

```
SELECT *
  FROM p_eintraegeunfertig(:user)
  ORDER BY sortstatus, sortprio DESC
```

Zuletzt noch die zu erledigenden Einträge als Liste derjenigen Einträge, die auf Funktionen zugewiesen sind, denen man selbst angehört:

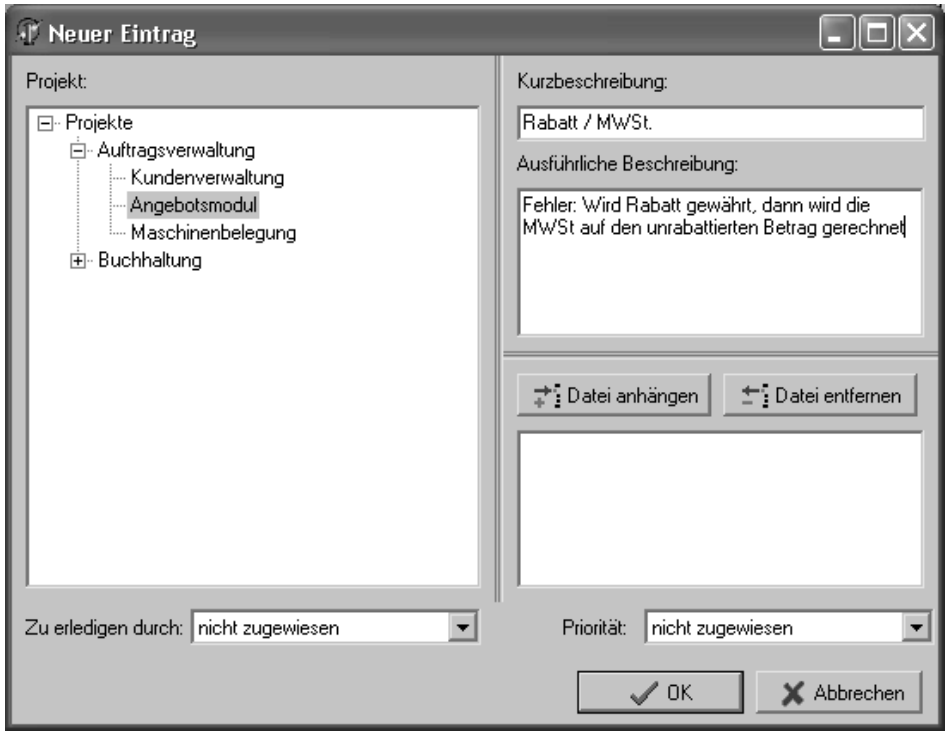
```
SELECT *
  FROM p_eintraegeunfertig(:user)
  WHERE funktion
    IN (SELECT name
      FROM t_funktion f
        INNER JOIN t_benutzer_funktion b
          ON b.funktion = f.ID
      WHERE b.benutzer = :user)
  ORDER BY sortstatus, sortprio DESC
```

Ein kleiner Tipp am Rande: Solche Anweisungen lässt man sich zunächst einmal vom Assistenten generieren und passt sie dann entsprechend an. Den Assistenten besprechen wir später.

In der Liste der Abfragen findet man die selbst erstellten privaten und die öffentliche Anweisungen, Letztere erkennt man am fehlenden Benutzer.

17.2.6 Einen Eintrag erstellen

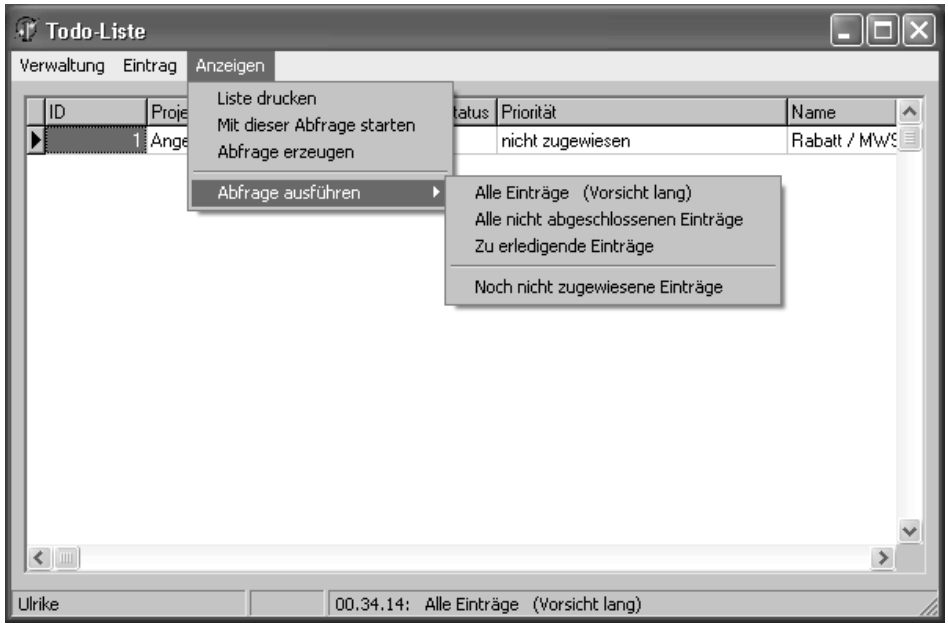
Nun wählen wir EINTRAG|NEUER EINTRAG.



Der Eintrag benötigt erst einmal ein Projekt, dem er zugeordnet ist – vorher wird der Ok-Button erst gar nicht freigegeben. Auch bei einem Projekt, für das kein E-Recht (*eintragen*) besteht, wird der Ok-Button gesperrt.

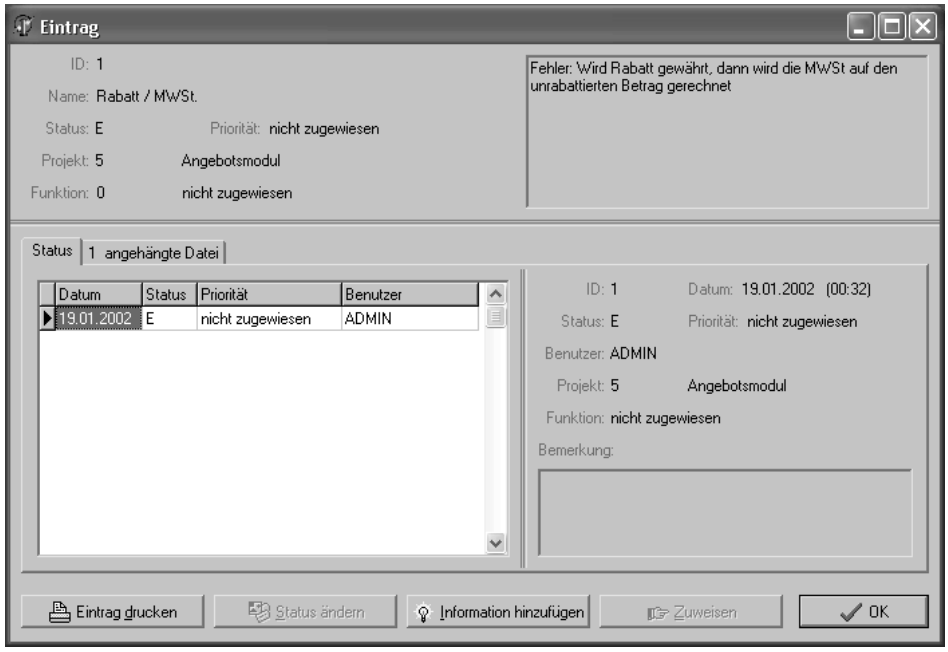
Des Weiteren sollte eine Kurz- und eine ausführliche Beschreibung eingegeben werden. Eine Zuweisung der Funktion und der Priorität kann erfolgen, muss aber nicht. Zusätzlich können Dateien angehängt werden:

Nun schließen wir beide Dialoge mit Ok und wählen dann als Anzeige die Liste aller Einträge:

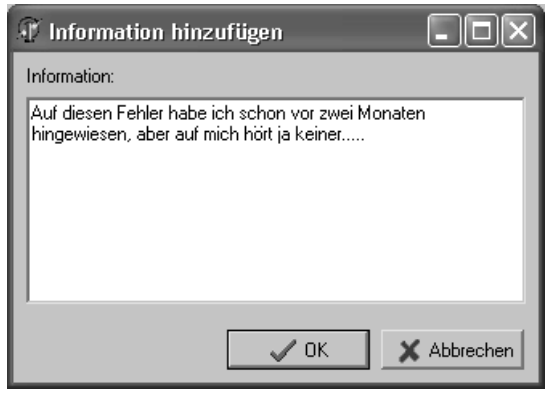


17.2.7 Eintrag ansehen

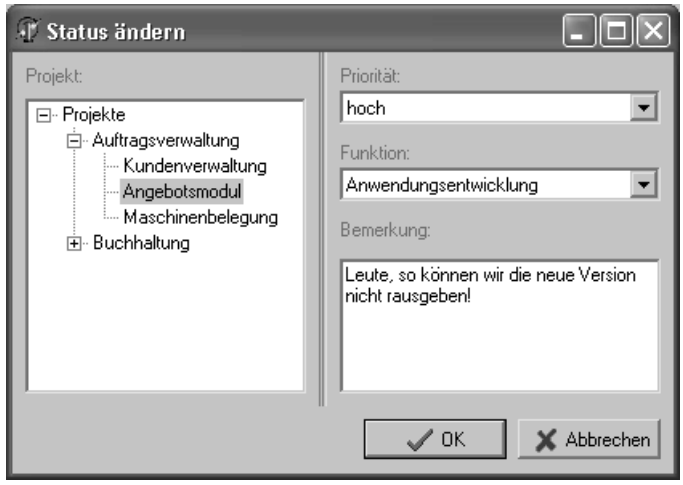
Mit dem Menüpunkt EINTRAG|DETAILS ANSEHEN / BEARBEITEN (oder mit einem Doppelklick auf den Eintrag) kann nun ein Dialog geöffnet werden, in dem man sich nicht nur den Eintrag selbst, sondern auch alle Statusänderungen und alle angehängten Dateien ansehen kann.



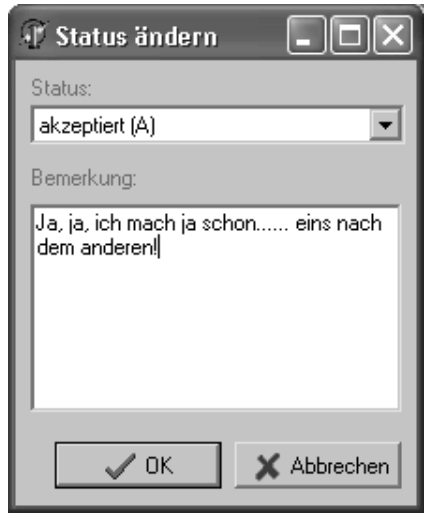
Je nachdem, welche Rechte der angemeldete Benutzer bezüglich des aktuellen Projektes hat, sind die Buttons am unteren Rand verfügbar oder nicht. In diesem Fall können wir lediglich eine Information hinzufügen.



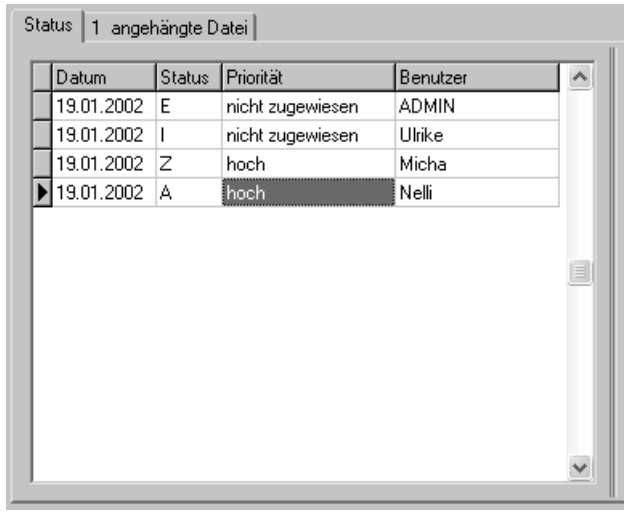
Der nächste vom Geschäftsprozess vorgesehene Schritt ist die Zuweisung von Priorität und der Funktion, die den Eintrag zu erledigen hat. Bei der Gelegenheit könnte man auch das Projekt ändern und außerdem könnte man wieder einen Kommentar hinzufügen.



Und als Nächstes würde dann irgendjemand von der Entwicklung den Status auf A setzen (und dann auf B und schließlich auf F).



Auf der Registerseite *Status* kann man alle diese Änderungen mitverfolgen und sich die Bemerkungen ansehen.



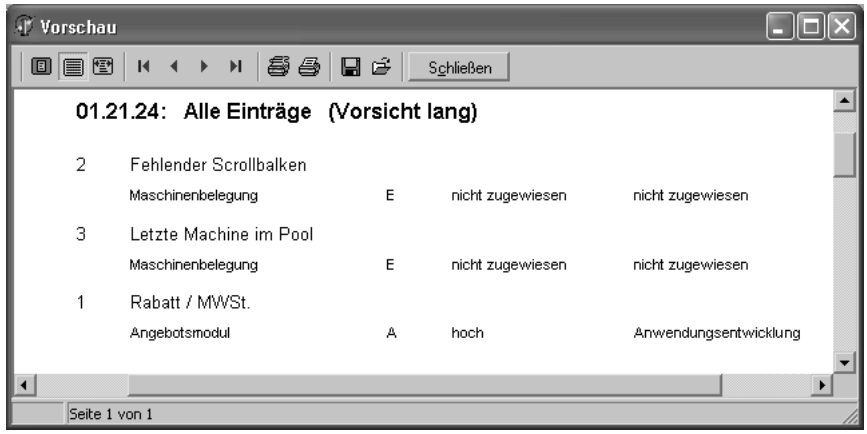
Die angehängten Dateien

Auf der Registerseite *x angehängte Dateien* findet man die Liste der angehängten Dateien, kann weitere Dateien hinzufügen, aber auch angehängte Dateien herunterladen.

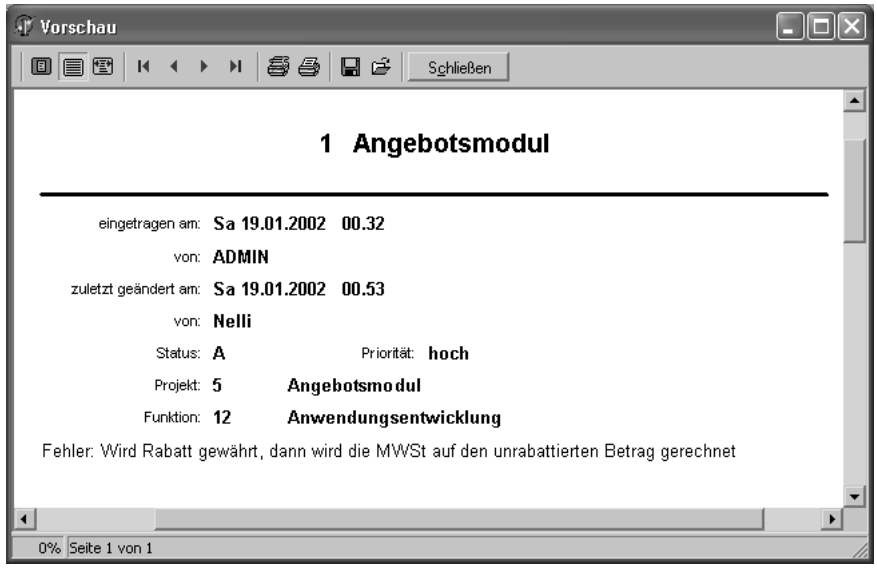
Auf dem Register-Reiter wird die Zahl der angehängten Dateien angegeben, damit man nicht immer nachschauen muss, ob überhaupt Dateien vorhanden sind.

17.2.8 Drucken

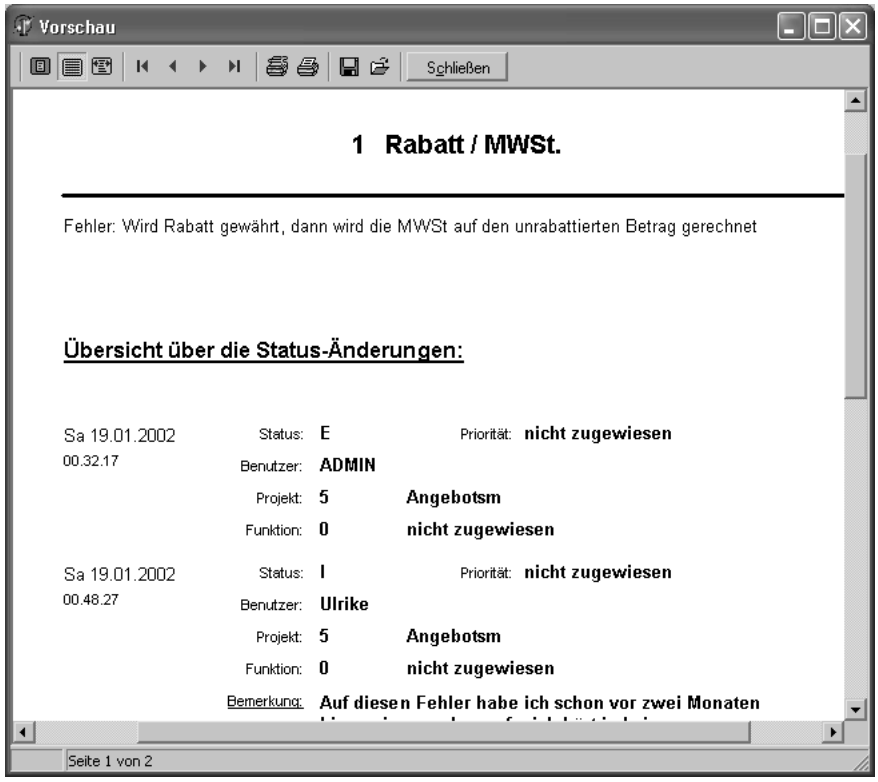
Das Projekt beinhaltet drei Reports, über die sich die Daten auch im ASCII-, CSV- und HTML-Format exportieren lassen. Zunächst gibt es einmal die Liste der Einträge, die sich an die Abfrage anhängt, die im Hauptformular angezeigt wird – die dort verwendeten Filter wirken also auch auf den Ausdruck:



Des Weiteren kann ein Eintrag in der »Papier sparenden« Variante ausgedruckt werden. Hier interessieren Informationen wie der aktuelle Status oder die Funktion, die sich des Eintrags annehmen soll, nicht jedoch alle Details.



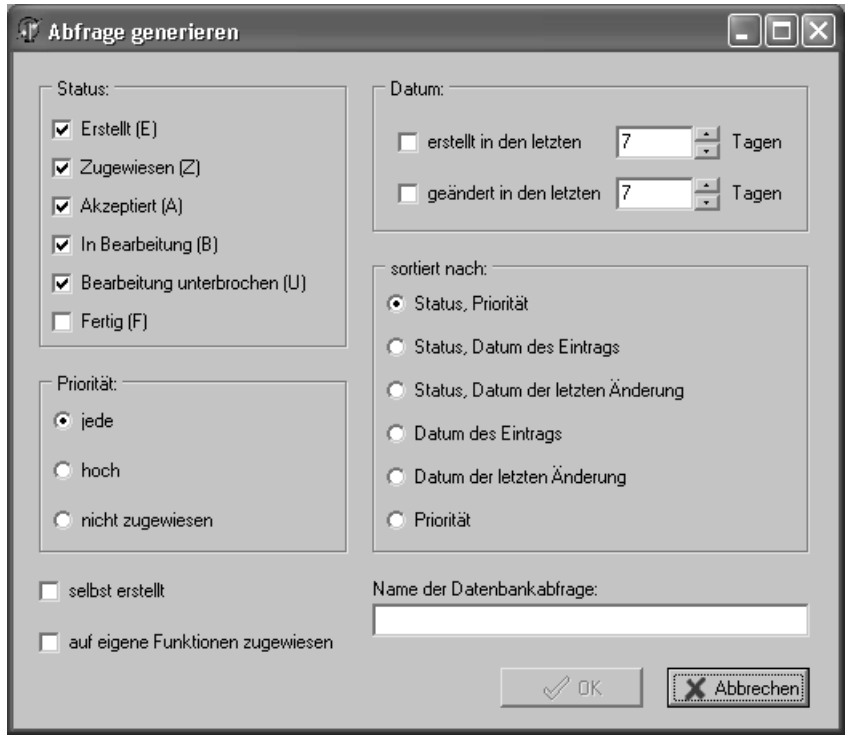
Wird ein Eintrag in der Detail-Variante ausgedruckt, dann beinhaltet er die vollständige Liste aller Statusänderungen (inklusive der dazugehörenden Kommentare) sowie die Liste aller angehängten Dateien.



17.2.9 Abfragen mit dem Assistenten erstellen

Wenn wir einen Assistenten zur Generierung von SQL-Abfragen bereitstellen, dann geschieht dies aus zwei Gründen:

- Zum einen kann man nicht von jedem Benutzer ausreichende SQL-Kenntnisse erwarten, um so eine Abfrage selbst zu formulieren.
- Ein Benutzer soll sich nur diejenigen Einträge ansehen können, für deren Projekte eine seiner Funktionen das Leserecht hat. Dafür muss der Zugriff über die dafür vorgesehenen STORED PROCEDURES erfolgen, was natürlich nicht zu gewährleisten wäre, wenn der Benutzer seine Abfragen selbst formuliert. Lediglich die Administratoren können Abfragen frei formulieren – diese haben ohnehin für alle Projekte das Leserecht, und man geht davon aus, dass sie wissen, was sie tun, wenn sie öffentliche Abfragen erstellen.



Die verschiedenen Filtermodule werden hier mit einer UND-Operation verknüpft, innerhalb der einzelnen Module wird eine ODER-Verknüpfung eingesetzt. Im oben abgebildeten Beispiel würden die Einträge angezeigt, die den Status E oder Z oder A oder B oder U haben – oder umgekehrt formuliert: die noch nicht fertig gestellt sind.

Bei der Priorität wird über die Sortierreihenfolge gefiltert. Somit können auch Einträge mit selbst erstellter Priorität den Weg in die Ergebnismenge finden, wenn nur der Sortierwert groß genug ist.

Da die Abfragen gespeichert werden, erfolgt die Filterung auf das Erstellungs- oder Änderungsdatum (Datum der letzten Statusänderung) nicht absolut, sondern in Abhängigkeit zum aktuellen Datum.

Des Weiteren besteht die Filtermöglichkeit auf die Einträge, die man selbst erstellt hat, und auf die Einträge, die Funktionen zugewiesen sind, denen man selbst angehört.

Zusätzlich kann nach sechs verschiedenen Kriterien sortiert werden. Damit der OK-Button freigegeben wird, muss die Abfrage einen Namen erhalten (der dann im Menü angezeigt wird).

17.3 Das SQL-Script

Mit dem SQL-Script wird die Datenbank definiert und erste Daten werden dort hingeschrieben. Wir wollen nun nicht dieses Script von vorne nach hinten durchgehen, sondern uns immer alle Elemente ansehen, die zu einer Tabelle gehören. Davor schauen wir uns jedoch die Definitionen der DOMAINS an:

17.3.1 Die DOMAINS

Ist der Typ einer Spalte keine selbst definiert DOMAIN, dann legt InterBase automatisch eine DOMAIN an, und zwar für jede Spalte eine eigene. Es bringt also keinen Vorteil, auf selbst definierte DOMAINS zu verzichten und die vordefinierten Typen zu verwenden:

```
CREATE DOMAIN d_datei AS BLOB SUB_TYPE 0 SEGMENT SIZE 80;
CREATE DOMAIN d_id AS INTEGER NOT NULL;
CREATE DOMAIN d_idchar AS CHAR(1) NOT NULL;
CREATE DOMAIN d_memo AS BLOB SUB_TYPE TEXT SEGMENT SIZE 80;
CREATE DOMAIN d_name AS VARCHAR(30) NOT NULL;
CREATE DOMAIN d_namelang AS VARCHAR(50) NOT NULL;
CREATE DOMAIN d_passwort AS BLOB SUB_TYPE 0 SEGMENT SIZE 80;
CREATE DOMAIN d_sortier AS INTEGER
   CHECK ((VALUE >= 0) AND (VALUE <= 100));
```

Alle DOMAINS erhalten als Prefix *d_*. Die DOMAIN *d_id* werden wir für alle Schlüsselspalten verwenden, *d_name* für die »normalen« Textspalten, *d_namelang*, wenn es etwas länger wird, *d_memo* für mehrzeiligen Text.

Alle anderen DOMAINS sind für jeweils einen einzigen Zweck: *d_datei* für angehängte Dateien, *d_idchar* als Schlüssel für den Status, *d_passwort* für Passwörter und *d_sortier* als Sortierreihenfolge für die Priorität.

17.3.2 Die Tabelle t_benutzer

Alle eingerichteten Benutzer werden in *t_benutzer* gespeichert:

```
CREATE TABLE t_benutzer
  (id D_ID,
  name D_NAME,
  passwort D_PASSWORT,
  memo D_MEMO,
  PRIMARY KEY (id),
  UNIQUE (name));
```

Ein Benutzer hat natürlich einen Namen und ohnehin hat jeder Datensatz eine eindeutige Nummer als Primärschlüssel. Des Weiteren kann zu jedem Benutzer eine Bemerkung eingegeben werden. Das Passwort wird binär gespeichert, somit ist es mit gängigen Datenbank-Tools nichts so ohne weiteres einsehbar.

Neben dem Primärschlüssel gibt es einen Schlüssel auf die Spalte *name*, welche doppelte Benutzernamen verhindert.

```
CREATE GENERATOR g_benutzer;
```

Für die Benutzernummer richten wir uns einen Generator ein.

```
CREATE PROCEDURE p_benutzer_id
RETURNS
  (gen INTEGER)
AS
BEGIN
  gen=GEN_ID(g_benutzer, 1);
  SUSPEND;
END^
```

Im SQL-Script werden zunächst alle STORED PROCEDURES »leer« erzeugt, die Anweisungen werden dann in einem zweiten Schritt mit ALTER PROCEDURE hinzugefügt. Dies hat den Vorteil, dass es zu keinen Konflikten kommt, wenn sich STORED PROCEDURES gegenseitig aufrufen. Aus Gründen der kürzeren Darstellung werden hier beide Anweisungen zu einer einzelnen zusammengefasst.

Die STORED PROCEDURE *p_benutzer_id* zählt den Generator um 1 hoch und gibt den neuen Wert zurück. Auf diese Weise können wir vom Programm aus recht einfach einen neuen Generatorwert abfragen. Beachten Sie, dass bei den STORED PROCEDURES (wie auch bei den TRIGGERN) ein anderes Terminierungszeichen verwendet wird – dieses ist vorher mit SET TERM eingestellt worden.

```
CREATE TRIGGER tr_benutzer FOR t_benutzer
ACTIVE BEFORE INSERT POSITION 0
AS BEGIN
  IF (NEW.id IS NULL)
    THEN NEW.id = GEN_ID(g_benutzer, 1);
END ^
```

Außerdem setzen wir einen TRIGGER ein, der verhindert, dass Datensätze ohne *id* in die Tabelle gelangen.

```
INSERT INTO t_benutzer (id, name)
  VALUES (1, 'ADMIN');
```

Zuletzt legen wir noch einen ersten Benutzer an, nämlich den Administrator. Da hier kein Passwort eingefügt wird – um es zu verschlüsseln, müssten wir erst eine UDF schreiben –, kommt man beim ersten Anmelden ohne Passwort in das Programm.

17.3.3 Die Tabelle t_funktion

Die Tabelle *t_funktion* gleicht *t_benutzer*, jedoch mit dem Unterschied, dass es kein Passwort gibt.

```
CREATE TABLE t_funktion
  (id D_ID,
  name D_NAME,
  memo D_MEMO,
  PRIMARY KEY (id),
  UNIQUE (name));
```

Auch hier gibt es wieder einen GENERATOR (*g_funktion*), eine STORED PROCEDURE (*p_funktion_id*) und einen Trigger (*tr_funktion*).

```
INSERT INTO t_funktion (id, name)
  VALUES (0, 'nicht zugewiesen');
INSERT INTO t_funktion (id, name)
  VALUES (1, 'ADMIN');
```

Neben der Administrator-Funktion gibt es hier auch die Funktion *nicht zugewiesen* für alle Einträge, die noch keiner Funktion zur Bearbeitung zugewiesen sind.

17.3.4 Die Tabelle t_benutzer_funktion

Mit der Tabelle *t_benutzer_funktion* werden die Benutzer den Funktionen zugeordnet.

```
CREATE TABLE t_benutzer_funktion
  (benutzer D_ID,
  funktion D_ID,
  UNIQUE (benutzer, funktion));
```

Ein Schlüssel über der Spaltenkombination sorgt dafür, dass eine solche Zuordnung nur einmal in der Tabelle gespeichert wird.

```
ALTER TABLE t_benutzer_funktion
  ADD FOREIGN KEY (benutzer)
  REFERENCES t_benutzer (id);
ALTER TABLE t_benutzer_funktion
  ADD FOREIGN KEY (funktion)
  REFERENCES t_funktion(id);
```

Damit sichergestellt wird, dass es die Datensätze, die da verknüpft werden, auch tatsächlich gibt, wird je ein Fremdschlüssel auf die Tabelle *t_benutzer* und *t_funktion* eingerichtet.

```
INSERT INTO t_benutzer_funktion (benutzer, funktion)
  VALUES (1, 1);
```

Zuletzt wird noch der Benutzer *ADMIN* der Funktion *ADMIN* zugeordnet.

17.3.5 Die Tabelle t_projekt

In der Tabelle *t_projekt* werden die Projekte gespeichert. Neben *id*, *name* und *memo* haben wir die Spalte *master*, welche das übergeordnete Projekt spezifiziert. Mit Hilfe diese Spalte wird dann die Baumansicht aufgebaut.

```
CREATE TABLE t_projekt
  (id D_ID,
  name D_NAME,
  memo D_MEMO,
  master INTEGER,
  PRIMARY KEY (id));
```

Um zu verhindern, dass nicht existierende Projekte als übergeordnetes Projekt gewählt werden oder dass übergeordnete Projekte gelöscht werden, wird hier ein Fremdschlüssel auf die eigene Tabelle eingerichtet.

```
ALTER TABLE t_projekt
  ADD FOREIGN KEY(master)
  REFERENCES t_projekt(id);
```

Auch hier wird für die Vergabe der Projekt-ID ein GENERATOR, eine STORED PROCEDURE sowie ein TRIGGER eingerichtet. Des Weiteren wird der erste Datensatz in die Tabelle eingefügt, der »Stammvater« aller noch zu definierenden Projekte.

```
INSERT INTO t_projekt (ID, name)
  VALUES (1, 'Projekte');
```

17.3.6 Die Tabelle t_recht

In der Tabelle *t_recht* wird gespeichert, welche Funktion für welches Projekt welche Rechte hat. Es handelt sich dabei um eine Verknüpfungstabelle zu *t_funktion* und *t_projekt*, in der es weitere Spalten gibt.

```
CREATE TABLE t_recht
  (funktion D_ID,
  projekt D_ID,
  erstellen D_IDCHAR,
  zuweisen D_IDCHAR,
  bearbeiten D_IDCHAR,
  fertigstellen D_IDCHAR,
  lesen D_IDCHAR,
  informieren D_IDCHAR,
  legitimieren D_IDCHAR,
  unterprojekte D_IDCHAR);
```

Auf die verbundenen Tabellen werden natürlich Fremdschlüssel gesetzt.

```
ALTER TABLE  t_recht
  ADD FOREIGN KEY (funktion)
  REFERENCES t_funktion (id);
ALTER TABLE  t_recht
  ADD FOREIGN KEY (projekt)
  REFERENCES t_projekt (id);
```

Wird einer Funktion irgendein Recht auf ein Projekt eingeräumt, dann wird ihr automatisch auch das Leserecht gewährt.

```
CREATE TRIGGER tr_recht_insert FOR t_recht
ACTIVE BEFORE INSERT POSITION 0
AS BEGIN
  IF ((NEW.erstellen = 'J')
       OR (NEW.erstellen = 'J')
       OR (NEW.zuweisen = 'J')
       OR (NEW.bearbeiten = 'J')
       OR (NEW.fertigstellen = 'J')
       OR (NEW.informieren = 'J')
       OR (NEW.legitimieren = 'J'))
    THEN NEW.lesen = 'J';
END ^
```

```
CREATE TRIGGER tr_recht_update FOR r_recht
ACTIVE BEFORE UPDATE POSITION 0
AS BEGIN
  IF ((NEW.erstellen = 'J')
      OR (NEW.erstellen = 'J')
      OR (NEW.zuweisen = 'J')
      OR (NEW.bearbeiten = 'J')
      OR (NEW.fertigstellen = 'J')
      OR (NEW.informieren = 'J')
      OR (NEW.legitimieren = 'J'))
    THEN NEW.lesen = 'J';
END ^
```

(Natürlich hätte man das auch über eine berechnete Spalte lösen können, aber da Rechte vergleichsweise selten vergeben, jedoch sehr häufig ermittelt werden müssen, ist diese Vorgehensweise die schnellere, zumal der Speicherplatzbedarf für eine Rechtespalte bei exakt einem Byte pro Datensatz liegt.)

```
INSERT INTO t_recht (funktion, projekt, erstellen, zuweisen,
    bearbeiten, fertigstellen, lesen, informieren, legitimieren,
    unterprojekte)
  VALUES (1, 1, 'J', 'J', 'J', 'J', 'J', 'J', 'J', 'J');
```

Nun erhält die Funktion ADMIN noch sämtliche Rechte für das Projekt 1, und da dem Projekt 1 alle anderen Projekte untergeordnet sind, erhalten damit die Administratoren sämtliche Rechte für alle Projekte.

17.3.7 Die Tabelle t_prioritaet

Jedem Eintrag kann eine Priorität zugewiesen werden. Diese wird in der Tabelle *t_prioritaet* gespeichert. In der Spalte *sortier* wird ein Wert zwischen 0 und 100 gespeichert, mit dessen Hilfe die Reihenfolge der Prioritäten festgelegt werden kann.

```
CREATE TABLE t_prioritaet
  (id D_ID,
  projekt D_ID,
  sortier D_SORTIER,
  name D_NAME,
  PRIMARY KEY (id));
```

Da die Priorität einem Projekt zugeordnet ist, wird auf die Tabelle *t_projekt* ein Fremdschlüssel erstellt.

```
ALTER TABLE t_prioritaet
  ADD FOREIGN KEY (projekt)
  REFERENCES t_projekt (id);
```

Die Prioritäten *hoch*, *mittel*, *niedrig* sowie *nicht zugewiesen* werden dem Projekt 1 zugewiesen und sind somit in allen Projekten verfügbar.

```
INSERT INTO t_prioritaet (projekt, sortier, name)
  VALUES (1, 99, 'nicht zugewiesen');
INSERT INTO t_prioritaet (projekt, sortier, name)
  VALUES (1, 80, 'hoch');
INSERT INTO t_prioritaet (projekt, sortier, name)
  VALUES (1, 50, 'mittel');
INSERT INTO t_prioritaet (projekt, sortier, name)
  VALUES (1, 20, 'niedrig');
```

Für die Vergabe der ID gibt es einen GENERATOR sowie einen TRIGGER.

Zur Ermittlung der Prioritäten, die in einem Projekt zur Verfügung stehen, gibt es die STORED PROCEDURE *p_prioritaeten*, welche in Kapitel 18.2.2 beschrieben ist.

17.3.8 Die Tabelle t_eintrag

Die Tabelle *t_eintrag* ist erstaunlich kurz – die Statusänderungen und die angehängten Dateien kommen schließlich in verbundene Tabellen.

```
CREATE TABLE t_eintrag
  (id D_ID,
  name D_NAMELANG,
  memo D_MEMO,
  PRIMARY KEY (id));
```

Auch hier wird die Eintrags-ID wieder mit GENERATOR, STORED PROCEDURE und TRIGGER bereitgestellt.

17.3.9 Die Tabelle t_status

In die Tabelle *t_status* kommen dann die Detail-Informationen zu einem Eintrag: Wer hat wann welchen Status gesetzt, ein Projekt oder eine Funktion zugewiesen oder eine Bemerkung hinzugefügt.

```
CREATE TABLE t_status
  (id D_ID,
  eintrag D_ID,
  datum TIMESTAMP,
  benutzer D_ID,
  status D_IDCHAR,
  funktion INTEGER,
  bemerkung D_MEMO,
  projekt D_ID,
  prioritaet D_ID,
  PRIMARY KEY (id));
```

Die Tabelle *t_status* verknüpft eine ganze Reihe anderer Tabellen mit dem Eintrag, entsprechend werden Fremdschlüssel eingerichtet.

```
ALTER TABLE t_status
  ADD FOREIGN KEY (eintrag)
  REFERENCES t_eintrag (id);
ALTER TABLE t_status
  ADD FOREIGN KEY (benutzer)
  REFERENCES t_benutzer (id);
ALTER TABLE t_status
  ADD FOREIGN KEY (projekt)
  REFERENCES t_projekt (id);
ALTER TABLE t_status
  ADD FOREIGN KEY (prioritaet)
  REFERENCES t_prioritaet (id);
```

Zur Vergabe einer ID gibt es dann wieder einen GENERATOR, eine STORED PROCEDURE und einen TRIGGER. Mit dem TRIGGER wird auch gleich das aktuelle Datum gespeichert, sofern dieses nicht bereits zugewiesen ist:

```
CREATE TRIGGER tr_status FOR t_status
ACTIVE BEFORE INSERT POSITION 0
AS BEGIN
  IF (NEW.id IS NULL)
    THEN NEW.id = GEN_ID(g_STATUS, 1);
  IF (NEW.datum IS NULL)
    THEN NEW.datum = "NOW";
END ^
```

17.3.10 Die Tabelle tn_status

Die Tabelle *tn_status* definiert die einzelnen Stati, die einem Eintrag zugewiesen werden können. In der Spalte *sortier* wird ein ganzzahliger Wert vergeben, anhand dessen die Stati in der Bearbeitungsreihenfolge sortiert werden können.

```
CREATE TABLE tn_status
  (id D_IDCHAR,
  name D_NAME,
  sortier INTEGER,
  PRIMARY KEY (id));
```

Diese Tabelle wird gleich mit den benötigten Werten gefüllt. Im Programm selbst brauchen wir dann nicht die Möglichkeit, hier Daten einzugeben.

```
INSERT INTO tn_status (ID, name, sortier)
  VALUES('A', 'akzeptiert', 3);
INSERT INTO tn_status (ID, name, sortier)
  VALUES('B', 'in Bearbeitung', 4);
INSERT INTO tn_status (ID, name, sortier)
  VALUES('E', 'erstellt', 1);
INSERT INTO tn_status (ID, name, sortier)
  VALUES('F', 'fertig', 6);
INSERT INTO tn_status (ID, name, sortier)
  VALUES('I', 'informieren', NULL);
INSERT INTO tn_status (ID, name, sortier)
  VALUES('U', 'Bearbeitung unterbrochen', 5);
INSERT INTO tn_status (ID, name, sortier)
  VALUES('Z', 'zugewiesen', 2);
```

17.3.11 Die Tabelle t_datei

In der Tabelle *t_datei* werden die angehängten Dateien gespeichert.

```
CREATE TABLE t_datei
  (id D_ID,
  eintrag D_ID,
  name D_NAME,
  datum TIMESTAMP,
  benutzer D_ID,
  memo D_MEMO,
  datei D_DATEI,
  PRIMARY KEY (id));
```

Die Datei erhält einen Namen, unter dem sie gespeichert wird, wenn sie heruntergeladen wird. Das Datum und der Benutzer werden gespeichert, ebenso kann eine Bemerkung angehängt werden.

```
ALTER TABLE t_datei
  ADD FOREIGN KEY (eintrag)
  REFERENCES t_eintrag (id);
ALTER TABLE t_datei
  ADD FOREIGN KEY (benutzer)
  REFERENCES t_benutzer (id);
```

Für die verbundenen Tabellen werden wieder Fremdschlüssel eingerichtet. Die Vergabe der ID erfolgt mit GENERATOR und TRIGGER. Auf *tr_datei* wird das aktuelle Datum gespeichert:

```
CREATE TRIGGER tr_datei FOR t_datei
ACTIVE BEFORE INSERT POSITION 0
AS BEGIN
  IF (NEW.id IS NULL)
    THEN NEW.id = GEN_ID(g_DATEI, 1);
  IF (NEW.datum IS NULL)
    THEN NEW.datum = "NOW";
END ^
```

17.3.12 Die Tabelle t_anzeige

In der Tabelle *t_anzeige* werden die SQL-Statements gespeichert, mit denen die Liste der Einträge gefiltert wird.

```
CREATE TABLE t_anzeige
  (id D_ID,
  benutzer INTEGER,
  anweisung D_MEMO,
  name D_NAMELANG);
```

Ist die Anweisung einem Benutzer zugewiesen, dann handelt es sich um eine private Anweisung – öffentliche Anweisungen haben den Wert NULL in der Spalte *benutzer*. Mit dem Inhalt der Spalte *name* wird für die Anweisung ein Menüpunkt erstellt.

Ach ja, und mit einem GENERATOR und einem TRIGGER wird die ID generiert, wer hätte es gedacht ...

17.3.13 Die STORED PROCEDURE p_eintragaktuell

Die STORED PROCEDURE *p_eintragaktuell* liefert aktuelle Informationen bezüglich des Eintrags, dessen ID als Parameter übergeben wird. Aufgrund ihrer Länge wollen wir diese Prozedur abschnittsweise besprechen:

```
CREATE PROCEDURE p_eintragaktuell
  (para_eintrag INTEGER)
RETURNS
  (id INTEGER,
  name VARCHAR(50),
  memo BLOB,
  datum_eintrag TIMESTAMP,
  datum_aender TIMESTAMP,
  benutzer_eintrag INTEGER,
  benutzer_aender INTEGER,
  status VARCHAR(1),
  funktion INTEGER,
  projekt INTEGER,
  prioritaet INTEGER,
  bemerkung_aender BLOB)
AS
  DECLARE VARIABLE v_min INTEGER;
  DECLARE VARIABLE v_max INTEGER;
```

Als Parameter wird – wie bereits gesagt – die ID des Eintrags übergeben. Die Ausgabeparameter *id*, *name* und *memo* beinhalten die gleichnamigen Felder des Eintrags. Des Weiteren erhält man sowohl das Datum als auch den ausführenden Benutzer sowohl des Eintrags als auch der letzten Änderung. Die Bemerkung des letzten Eintrags wird nach *bemerkung_aender* geschrieben.

Zudem werden der aktuelle Status, die Funktion, welcher der Eintrag aktuell zugewiesen ist, das aktuelle Projekt sowie die aktuelle Priorität zurückgegeben. Anschließend werden die Variablen *v_min* und *v_max* deklariert.

```
BEGIN
  SELECT
      id, name, memo
    FROM t_eintrag
    WHERE ID = :para_eintrag
    INTO :id, :name, :memo;
```

Zuerst werden die Daten aus *t_eintrag* geholt.

```
SELECT
    MIN(id), MAX(id)
  FROM t_status
  WHERE EINTRAG = :para_eintrag
  INTO :v_min, :v_max;
```

Nun wird der minimale und der maximale Wert für *id* aus der Tabelle *t_status* ermittelt, der dem gesuchten Eintrag zugewiesen ist. Da die Nummern fortlaufend vergeben werden, erhalten wir so die *id* des Eintrags und der letzten Änderung.

```
SELECT
    datum, benutzer
  FROM t_status
  WHERE ID = :v_min
  INTO :datum_eintrag, :benutzer_eintrag;
```

Mit *v_min* besorgen wir uns nun das Datum und den Benutzer des Eintrags.

```
SELECT
    datum, benutzer, funktion, projekt, prioritaet, bemerkung
  FROM t_status
  WHERE ID = :v_max
  INTO :DATUM_aender, :benutzer_aender,
    :funktion, :projekt, :prioritaet, :bemerkung_aender;
```

Mit *v_max* erhalten wir neben Datum und Benutzer der letzten Änderung auch deren Bemerkung sowie die aktuelle Funktion, das Projekt und die Priorität.

```
SELECT
    MAX(id)
  FROM t_status
  WHERE EINTRAG = :para_eintrag
    AND status IN ('E', 'A', 'B', 'U', 'F')
  INTO :v_max;
```

Was wir nicht mit der vorherigen Abfrage erhalten, ist der Status der letzten Änderung, denn schließlich kann die letzte Änderung auch eine hinzugefügte Information sein, und *I* ist kein Status. Es muss also der letzte Datensatz ermittelt werden, der sich auf eine Statusänderung bezieht. Mit dessen Hilfe kann dann der aktuelle Status ermittelt werden.

Mit SUSPEND werden die Daten übertragen.

```
  SELECT
      status
    FROM t_status
    WHERE ID = :v_max
    INTO :status;
  SUSPEND;
END ^
```

17.3.14 Die STORED PROCEDURE p_eintraegeoffen

Die STORED PROCEDURE *p_eintraegeoffen* ermittelt zu einem Benutzer alle »offenen« Einträge, also alle Einträge, die vom betreffenden Benutzer akzeptiert wurden und den Status *A*, *B* oder *U* haben. Diese Abfrage eignet sich besonders als Startabfrage, damit beim Programmstart gleich sichtbar wird, was noch zu tun ist.

```
CREATE PROCEDURE p_eintraegeoffen
  (para_benutzer INTEGER)
RETURNS
  (id INTEGER,
  projekt VARCHAR(30),
  status VARCHAR(1),
  prioritaet VARCHAR(30),
  name VARCHAR(50),
  datum_eintrag TIMESTAMP,
  datum_aender TIMESTAMP,
  benutzer_eintrag VARCHAR(30),
  funktion VARCHAR(30),
  sortprio INTEGER,
  sortstatus INTEGER)
AS
  DECLARE VARIABLE v_id INTEGER;
  DECLARE VARIABLE v_benutzer_aender INTEGER;
BEGIN
  FOR SELECT id
    FROM t_eintrag
    INTO :v_id
  DO
  BEGIN
    SELECT
        e.id,
```

```
        e.name,
        e.datum_eintrag,
        e.datum_aender,
        e.status,
        e.benutzer_aender,
        b.name,
        f.name,
        p.name,
        i.name,
        i.sortier,
        n.sortier
      FROM p_eintragAktuell(:v_id) e
        INNER JOIN t_benutzer b
          ON e.benutzer_eintrag = b.ID
        LEFT OUTER JOIN t_funktion f
          ON e.funktion = f.ID
        INNER JOIN t_projekt p
          ON e.projekt = p.ID
        INNER JOIN t_prioritaet i
          ON e.prioritaet = i.ID
        LEFT OUTER JOIN tn_status n
          ON e.status = n.id
      INTO
        :id,
        :name,
        :datum_eintrag,
        :datum_aender,
        :status,
        :v_benutzer_aender,
        :benutzer_eintrag,
        :funktion,
        :projekt,
        :prioritaet,
        :sortprio,
        :sortstatus,
    IF ((v_benutzer_aender = para_benutzer)
        AND (status IN ('A', 'B', 'U')))
       THEN SUSPEND;
  END
END;
```

Die eigentliche »Arbeit« erledigt hier die eben besprochene STORED PROCEDURE *p_eintragaktuell*. Da das Ergebnis von *p_eintraegeoffen* direkt angezeigt werden soll, werden hier mittels JOIN für Benutzer, Funktion, Projekt und Priorität die Klartextwerte eingefügt. Der JOIN auf *tn_status* ermittelt die Sortierreihenfolge, wenn nach dem Status sortiert wird.

Die STORED PROCEDURE ermittelt diese Werte für alle Einträge, ruft jedoch nur dann SUSPEND auf, wenn der Status *A*, *B* oder *U* und der Benutzer der letzten Änderung der als Parameter übergebene Benutzer ist.

18 To-Do-Liste mit der BDE

Die BDE ist immer noch der bequemste Weg, um ein Datenbankprojekt zu realisieren, deshalb wollen wir damit beginnen. Die dabei erstellten Masken wollen wir dann auch für die anderen Realisierungswege nutzen.

Die To-Do-Liste ist eigentlich ein kleineres Projekt. Es soll jedoch die Gelegenheit genutzt werden, die Vorgehensweise beim Erstellen größerer Projekte aufzuzeigen.

18.1 Wie man größere Projekte realisiert

Kleinere Projekte kann man mit Delphi ziemlich »geradeaus« programmieren: Alle Formulare werden beim Programmstart erstellt, alle Bezeichner blieben so, wie sie Delphi mal benannt hat, jeder hat Zugriff auf alles, alle beteiligten Units werden in das Projekt aufgenommen usw.

Wenn man jedoch größere Projekte (über 100 selbst geschriebene Units) auf diese Weise erstellt, dann bekommt man einige Probleme:

- Das Erstellen der Formulare beim Programmstart dauert zu lange, insbesondere dann, wenn dabei noch Datenbankverbindungen hergestellt werden sollen.
- Die Projektdatei wird reichlich unübersichtlich, da helfen irgendwann auch »sprechende« Bezeichner nichts mehr.
- Vor allem aber: Durch Datenaustausch zwischen den Formularen geht sämtliche Übersicht verloren. Wenn dann noch im Team programmiert wird, potenziert sich dieses Problem.

Das Problem der Kreuz- und Querverbindungen

Nehmen wir einmal an, wir haben ein Projekt mit etwa hundert Formularen. Diese werden zwar nicht alle beim Programmstart erstellt – Delphi sieht das ja durchaus vor –, aber alle Formular-Variablen befinden sich im *interface*-Teil der betreffenden Unit.

Nun hindert einen nichts daran, dass man in *unit37* (oder wie ein sprechender Bezeichner auch immer lauten würde) folgende Anweisung erstellt:

```
Form84.Label27.Text := 'Suche fehlgeschlagen';
```

Das ist zunächst einmal eine ganz normale Anweisung, die auch problemlos funktioniert – wo liegt nun das Problem?

Wenn nach ein paar Monaten eine ganz andere Person Änderungen an diesem Projekt vornehmen soll, dann wird sie »viel Freude« damit haben.

Nehmen wir einmal an, *TForm84* wird geändert und dabei wird *Label27* für einen anderen Zweck verwendet. Das Projekt lässt sich nun wie gewohnt kompilieren und starten, aber irgendwann wird *Form37* aufgerufen und ändert die Beschriftung von diesem Label – selbstverständlich fällt das erst auf, nachdem das Programm beim Kunden ist, denn dem Tester fällt es wirklich nicht ein, eine Sache neu zu testen, die bei den letzten 20 Versionen einwandfrei funktioniert hat.

Nun ist dieses Beispiel ja auch noch schön einfach und es geht auch nur um die Anzeige einer Fehlerinformation. Wenn solche Kreuz- und Querverbindungen jedoch bei der Datenbearbeitung passieren – und da treten sie meist viel häufiger auf als bei der Anzeige –, dann fallen sie oft erst später auf und sind schwerer reproduzierbar und lassen sich nur mit großem Aufwand beheben.

Spätestens dann, wenn sich kein Programmierer mehr so richtig traut, Änderungen am Quellcode vorzunehmen, weil er immer Angst haben muss, dass ihm das Projekt »um die Ohren fliegt«, sollte man komplett neu und von vorne beginnen. Leider fehlt oft der Mut zu einem solchen harten Schnitt. Stattdessen »fliehen« die Entwickler in andere Projekte, und wenn sich neue Leute in den Code einarbeiten müssen, mildert das keinesfalls dieses Problem.

Projekte zerlegen

Erfahrungsgemäß ist es keine Lösung, allen beteiligten Entwicklern ganz streng solche Kreuz- und Querverbindungen zu untersagen – das funktioniert in der Praxis einfach nicht. Stattdessen muss man eine Vorgehensweise finden, die das Problem gleich von Anfang an ausschließt: Dazu muss das Projekt in Teilprojekte zerlegt werden.

Damit ist jetzt nicht gemeint, verschiedene DLLs zu erstellen. Es gehört zu den Stärken von Delphi, das »alles« in einer Exe ist und man sich deshalb um die Installation keine großen Gedanken machen muss. Außerdem müsste in jede DLL die Klassenbibliothek gelinkt werden (es sei denn, man verwendet Runtime-Packages), so dass die Größe der Dateien unnötig wächst.

Vielmehr sollen in einer Projektgruppe Test-Projekte zu den einzelnen Bausteinen angelegt werden. Auf diese Weise steckt in einem Projekt immer nur ein einzelner Baustein – zum Beispiel die Benutzerverwaltung. Das Hauptformular dieser Projekte wird auch gleich darauf ausgelegt, nur diesen einen Baustein zu testen – es muss ja nicht sein, dass erst viele Daten erfasst werden müssen, bevor man zur eigentlichen Aufgabe kommt.

Für jedes dieser Baustein-Projekte wird nun eine eigene Schnittstellen-Unit erstellt. Mit dieser Schnittstellen-Unit wird einmal der Datenaustausch zwischen den einzelnen Bausteinen erledigt. Es ist also nicht mehr möglich, auf eine Komponente aus *Form84* zuzugreifen, weil man den dazugehörenden Zeiger gar nicht mehr bekommt. Vielmehr würde für einen solchen Fehler-Hinweis wie im vorigen Beispiel eine Eigenschaft in der Schnittstellen-Unit erstellt. Solange man nun diese Schnittstellen-Unit nicht ändert, kann man in diesem Teilprojekt alles tun und lassen, was man möchte (naja, nicht ganz ...), ohne dass dies Auswirkungen auf andere Teilprojekte hat.

Durch ein solches Zerlegen in Teilprojekte kann man ein Projekt auch viel leichter auf mehrere Entwickler aufteilen.

Das Hauptprojekt hat dann nur noch die Unit des Hauptformulars, meist ein Datenmodul für die *TDatabase*-Instanz, und ansonsten die Schnittstellen-Units. Wenn ein Projekt von beispielsweise 100 Formularen in zehn Teilprojekte à etwa zehn Formulare aufgeteilt wird, dann werden sowohl das Hauptprojekt als auch die Teilprojekte überschaubar.

Doch nun genug der grauen Theorie, lassen Sie uns ansehen, wie das in der Praxis funktioniert.

Die Projektgruppe anlegen

Zunächst legen wir mit DATEI|NEU|WEITERE|NEU|PROJEKTGRUPPE eine neue Projektgruppe an und speichern diese in ein neues Verzeichnis unter dem Namen *To-DoGroup*.

Legen Sie dann darin ein neues Projekt an, das Sie *p_testverw* nennen. Es ist das Projekt zum Testen des Bausteins *Verwaltung*. In diesem Baustein findet man die Verwaltung der Benutzer, der Funktionen und der Projekte, also der Rechte und Prioritäten.

Das Hauptformular bekommt den Namen *frm_VerwMain*, die Unit heißt dementsprechend *u_VerwMain*.

Datenmodul oder kein Datenmodul

Eigentlich ist so ein Datenmodul eine feine Sache: Man bekommt seine Formulare schön übersichtlich, hat alle Datenzugriffskomponenten beisammen, und wenn man mal ein Projekt hätte, das dieselben Datenzugriffskomponenten brauchen würde ...

In der Praxis unterläuft so ein Datenmodul ganz prima die Trennung in einzelne Bausteine. Die Idee, nur diejenigen Datenbankverbindungen offen zu halten, die auch benötigt werden, lässt sich mit einem Datenmodul meist nur sehr schwer umsetzen.

Und außerdem: Schon in einem Teilprojekt kommen schnell ein paar dutzend Datenzugriffskomponenten zusammen – können Sie dort den Überblick behalten? (Ja, ja, ich weiß: *Nur der Kleingeist hält Ordnung, das Genie durchblickt das Chaos*. Wenn ich in Erinnerung so manches Projekt Revue passieren lasse, dann wünsche ich mir weniger Genies in der Programmierung ...)

Ich höre schon den Einwand: Immer noch besser als ein paar dutzend Datenzugriffskomponenten auf einem Formular. Das ist zwar nicht falsch, aber wer so viel auf ein Formular packen müsste, der hat entweder ein unübersichtliches Formular oder ein verbesserungsbedürftiges Datenbankdesign (beispielsweise in Form fehlender STORED PROCEDURES), erfahrungsgemäß meist beides ...

Der langen Rede kurzer Sinn: Die Datenzugriffskomponenten kommen auf die Formulare. Dennoch erstellen wir ein Datenmodul: Für die *TDatabase*-Instanz, für ein paar globale Variablen und sonst noch ein paar Kleinigkeiten.

18.1.1 Das Datenmodul

Erstellen wir also ein neues Datenmodul und geben diesem den Namen *frm_data*; der Name der Unit lautet somit *u_data*. In dieses Datenmodul kommt zunächst nur eine Komponente, nämlich *Database1*. Solange Sie nur eine solche Komponente haben, spricht überhaupt nichts dagegen, den Namen so zu lassen, wie ihn Delphi vergeben hat.

Als *DatabaseName* verwenden wir *_INTERN*, das steht im Alphabet ganz vorne, und wenn wir diese Eigenschaft einer Datenzugriffskomponente setzen wollen, dann reicht ein Doppelklick im Objektinspektor. *LoginPromt* setzen wir auf *false*, die Parameter werden wie folgt gesetzt:

```
USER NAME=SYSDBA
PASSWORD=masterkey
```

Anschließend setzen wir *Connected* auf *true*.

Nun ergänzen wir zunächst einmal ein paar globale Variablen:

```
var
  frm_data: Tfrm_data;
  gvUserID: integer;
  gvUserName: string;
  gvIsAdmin: boolean = true;
  gvChanged: boolean;
```

Alle selbst angelegten globalen Variablen erhalten als Präfix *gv*. In *gvUserID* steckt die Nummer des Benutzers, der sich angemeldet hat. Diese Nummer wird benötigt, um abzuprüfen, welche Rechte der Benutzer an einem bestimmten Projekt

hat. Um den Benutzer mit dem Namen anreden zu können, wird der Name in *gvUserName* gespeichert.

Manche Aktionen können nur von einem Benutzer mit Administratorfunktion ausgeführt werden. Um hier nicht jedes Mal prüfen zu müssen, ob der aktuell angemeldete Benutzer dazu gehört, wird dies in *gvIsAdmin* gespeichert. In *gvChanged* wird gespeichert, ob seit dem Start einer Transaktion eine Datenänderung vorgenommen wurde.

Des Weiteren werden einige Routinen und Methoden erstellt:

```
procedure RefreshQuery(Query: TQuery);
begin
  Query.Close;
  Query.Open;
end;
```

Bei *TQuery* zeigt der Aufruf von *Refresh* nicht den erwünschten Erfolg, die Abfrage muss zur Aktualisierung geschlossen und neu geöffnet werden. Passiert das nur bedingt, dann muss gegebenenfalls nur für diese beiden Anweisungen ein eigener *begin..then*-Block erstellt werden. Und auch sonst kommt diese Kombination so häufig vor, dass sich eine eigene Routine lohnt.

```
procedure UpdateRefreshQuery(Query: TQuery; IndexField: string);
var
  i: integer;
begin
  try
    gvChanged := true;
    if IndexField <> ''
      then i := Query.FieldByName(IndexField).AsInteger;
    if Query.UpdatesPending then
    begin
      Query.ApplyUpdates;
      Query.CommitUpdates;
    end;
  finally
    Query.Close;
    Query.Open;
    if IndexField <> ''
      then Query.Locate(IndexField, Variant(i), []);
  end;
end; {procedure UpdateRefreshQuery}
```

Des Öfteren müssen zwischengespeicherte Aktualisierungen eingetragen werden, anschließend muss die Abfrage aktualisiert werden und häufig soll der Satzzeiger dabei seine Position nicht verändern. Dies alles wird in die Routine *UpdatesRefreshQuery* gepackt.

Wird dem Parameter *IndexField* etwas anderes als ein leerer String übergeben, dann wird dies als Spaltenbezeichner der Schlüsselspalte verstanden. Der dort enthaltene Wert wird in *i* zwischengespeichert, damit am Ende mit *Locate* wieder positioniert werden kann.

Alsdann wird geprüft, ob überhaupt zwischengespeicherte Aktualisierungen vorliegen – ein Aufruf von *ApplyUpdates* würde anderenfalls eine Exception auslösen. Mit *CommitUpdates* werden die zwischengespeicherten Werte dann gelöscht.

Normalerweise werden diese Werte erst gelöscht, wenn die Transaktion erfolgreich beendet wurde. Da diese Routine jedoch für Administrationsaufgaben eingesetzt wird und konkurrierender Zugriff somit unwahrscheinlich sein sollte, kann hier vereinfacht verfahren werden.

```
procedure Tfrm_data.BeginTransaction;
begin
  if Database1.InTransaction
    then Database1.Commit;
  Database1.StartTransaction;
end;
```

Normalerweise sollte ausgeschlossen sein, dass *BeginTransaction* eine Transaktion startet, solange noch eine läuft, sicherheitshalber wird das jedoch geprüft und eine gegebenenfalls laufende Transaktion mit *Commit* beendet.

```
function CheckCanClose(Form: TForm): boolean;
begin
  if Form.ModalResult <> mrOK then
  begin
    if gvChanged
      then result := (MessageDlg('Alle Änderungen verwerfen?',
        mtConfirmation, [mbYes, mbNo], 0) = mrYes)
      else result := true;
  end {if Form.ModalResult <> mrOK then}
  else result := true;
end; {function Tfrm_data.CheckCanClose}
```

Fast alle Formulare werden wir modal anzeigen, sie lassen sich somit mit *Ok* oder *Abbrechen* schließen. Dementsprechend wird die laufende Transaktion mit *Commit* oder *Rollback* beendet. Wird das Formular mit dem *Abbrechen*-Button ge-

schlossen, dann wird geprüft, ob Änderungen vorgenommen wurden. Ist dies der Fall, wird der Anwender gefragt, ob diese wirklich verworfen werden sollen.

```
procedure Tfrm_data.ModalAnzeigen(FormClass: TComponentClass;
  StartTransaction: boolean; FormTag: integer = 0);
var
  Form: TForm;
begin
  Form := FormClass.Create(nil) as TForm;
  try
    if StartTransaction then
    begin
      BeginTransaction;
      Form.Tag := FormTag;
      if Form.ShowModal = mrOK then
      begin
        try
          Database1.Commit;
        except
          Database1.Rollback;
          ShowMessage('Die Änderungen konnten nicht '
            + 'geschrieben werden!');
        end;
      end {if Form.ShowModal = mrOK then}
      else Database1.Rollback;
      gvChanged := false;
    end {if StartTransaction then}
    else Form.ShowModal;
  finally
    Form.Free;
  end;
end; {procedure ModalAnzeigen}
```

Die Vorgehensweise beim Anzeigen von Formularen ist meist die gleiche: Erst wird das Formular erzeugt, dann kommt eine *try..finally..end*-Konstruktion, in deren *finally*-Teil das Formular wieder freigegeben wird. Der Typ des Formulars wird dabei als Parameter übergeben.

Um auf das Formular noch gewissen Einfluss nehmen zu können, wird der Inhalt des Parameters *FormTag* in die Eigenschaft *Tag* des Formulars geschrieben. Dieser Parameter hat einen Vorgabewert, er muss somit beim Aufruf nicht zwingend gesetzt werden.

Hat der Parameter *StartTransaction* den Wert *false*, dann wird das Formular einfach nur *modal* angezeigt. Anderenfalls wird vor der Anzeige eine Transaktion gestartet, die danach mit *Commit* beendet wird. Tritt dabei eine Exception auf, dann wird eine Fehlermeldung ausgegeben und die Transaktion mit *Rollback* zurückgenommen. Zuletzt wird *gvChanged* zurückgesetzt.

Da ohnehin nur Tabellen aktualisiert wurden, die mit einem modal geöffneten Formular angezeigt werden, das gerade geschlossen wurde, ist nach einem *Rollback* auch keine Aktualisierung erforderlich.

18.2 Die Verwaltung

In der Verwaltung werden wir Benutzer, Funktionen und Projekte verwalten. Den Projekten zugeordnet sind dabei die Prioritäten und die Rechte. Um die einzelnen Formulare aufzurufen, ergänzen wir das Hauptformular unseres Testprojekts mit einem Menü, das jedoch nur den Menütitel *Verwaltung* erhält.

Die Schnittstellen-Unit u_verwaltung

Nun erstellen wir eine neue Unit namens *u_verwaltung*, welche die Schnittstelle zum Baustein *Verwaltung* bilden soll. Darin erstellen wir die Klasse *TVerwaltung*:

```
interface

type
  TVerwaltung = class(TObject)
  private
    procedure BenutzerClick(Sender: TObject);
    procedure FunktionenClick(Sender: TObject);
    procedure ProjekteClick(Sender: TObject);
  public
    class procedure SetMenuItem(Master: TMenuItem);
    class function PasswortDialog: boolean;
  end;

implementation

var
  Verwaltung: TVerwaltung;
```

Die einzelnen Methoden sollen uns momentan weniger interessieren als der Umstand, dass die Zeigervariable im *implementation*-Teil liegt – Zugriffe von außen sind demnach nicht vorgesehen.

```
class procedure TVerwaltung.SetMenuItem(Master: TMenuItem);

  procedure NewMenuItem(ACaption: string; AOnClick: TNotifyEvent);
  var
    Item: TMenuItem;
  begin
    Item := TMenuItem.Create(Master.Owner);
    Item.Caption := ACaption;
    Item.OnClick := AOnClick;
    Master.Add(Item);
  end;

begin
  if not Assigned(Verwaltung)
    then Verwaltung := TVerwaltung.Create;
  NewMenuItem('Benutzer', Verwaltung.BenutzerClick);
  NewMenuItem('Funktionen', Verwaltung.FunktionenClick);
  NewMenuItem('Projekte', Verwaltung.ProjekteClick);
end; {class procedure TVerwaltung.SetMenuItem}
```

Eine Instanz dieser Klasse wird beim erstmaligen Aufruf von *SetMenuItem* erstellt. Dieser Klassenmethode wird ein Zeiger auf einen Menüeintrag übergeben, unter dem drei neue Einträge erstellt werden, nämlich *Benutzer*, *Funktionen* und *Projekte*.

Warum nicht die konventionelle Vorgehensweise mit dem Menü-Designer? Weil erstens dann das Hauptprojekt schrecklich unübersichtlich wird ob der vielen *OnClick*-Ereignisbehandlungsroutinen. Außerdem müsste man Änderungen immer sowohl im Haupt- als auch im jeweiligen Testprojekt vornehmen; da schleichen sich immer mal Fehler ein.

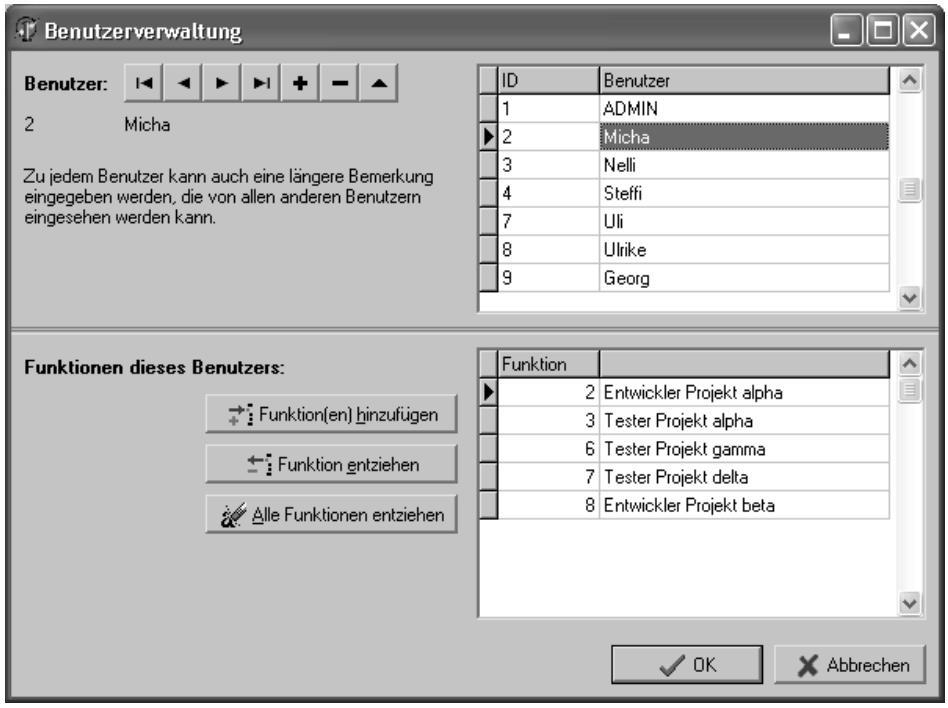
Und wenn im Team programmiert wird, dann muss das Hauptformular bei jeder Änderung aus der Versionsverwaltung ausgecheckt werden, das »freut alle Beteiligten«. Die zwei Minuten, die hier in zusätzlichen Quelltext gesteckt werden, holt man durch weniger Ärger mehrfach wieder herein.

```
procedure TVerwaltung.BenutzerClick(Sender: TObject);
begin
  frm_data.ModalAnzeigen(Tfrm_benutzer, true);
end;
```

Die *OnClick*-Ereignisbehandlungsroutinen der erstellten Menüpunkte sind alle prinzipiell gleich aufgebaut: Das dazugehörende Formular wird modal angezeigt, dabei wird eine Transaktion gestartet. Die Methode *ModalAnzeigen* macht uns hier die Arbeit sehr einfach.

18.2.1 Die Benutzerverwaltung

In der Benutzerverwaltung können neue Benutzer hinzugefügt werden, können einem Benutzer Funktionen zugewiesen oder entzogen sowie die Passwörter geändert werden. Für alle Aktionen mit Ausnahme der Passwortänderung benötigt man Administrationsrechte.



Benutzer, die kein Administrationsrecht haben, bekommen im *DBNavigator1* nur den Button zum Browsen angezeigt, auch die Buttons zum Hinzufügen und Entziehen von Funktionen fehlen.

Lediglich dann, wenn der Benutzer auf dem eigenen Datensatz steht, erscheint ein Button *Passwort ändern*.

Aufbau des Formulars

Die Formulare für die Verwaltung von Benutzern und Funktionen sehen ziemlich ähnlich aus. Es wäre auch möglich gewesen, hier ein gemeinsames Fenster zu verwenden, durch die Logik mit der Passwortänderung von Nicht-Administratoren wäre der Aufwand jedoch unnötig hoch geworden. Also doch getrennte Formulare.

- Das Formular erhält den Namen *frm_benutzer*, die Unit wird folglich unter *u_benutzer* gespeichert.
- Da keine Notwendigkeit besteht, die Formularbreite zu verändern, setzen wir *Constraints.MaxWidth* und *Constraints.MinWidth* jeweils auf 600, außerdem wird *Scaled* auf *false* gesetzt.
- Formulare, die modal angezeigt werden, zeigt man normalerweise in der Bildschirmmitte an. Nun ist es jedoch möglich, dass man während der Bearbeitung das Fenster eines anderen Programms geöffnet hält, um hier Daten zu übernehmen. Wird *Position* auf *poMainFormCenter* gesetzt, dann wird das Fenster innerhalb des Hauptformulars der Anwendung positioniert.

  Das Problem ist hier der Betrieb mit zwei Monitoren, wenn der zweite Monitor oberhalb des ersten steht (was im Bürobereich ziemlich unüblich ist). Die Werte für *Left* und *Top* sind nämlich auf den positiven Bereich beschränkt, so dass diese Fenster gegebenenfalls auf dem falschen Monitor angezeigt werden. Angesichts dessen, dass diese Konfiguration unüblich ist, nehmen wir diesen Nachteil in Kauf.
- Wir fügen zwei Panels ein und setzen *Align* auf *alTop* und *alBottom*.
- Nun ergänzen wir einen Splitter mit den Eigenschaften *Align* = *alTop*, *Height* = *5* und *Beveled* = *true*.
- In den freien Bereich wird ein weiteres Panel eingefügt, dessen Eigenschaft *Align* auf *alClient* gesetzt wird. Nun werden alle drei Panels markiert, die Beschriftung gelöscht und *BevelOuter* auf *bvNone* gesetzt.
- In das unterste Panel werden zwei Buttons gesetzt, deren Eigenschaft *Kind* auf *bkOK* und *bkAbort* gesetzt wird. Das Panel kann nun in der Höhe den Buttons angepasst werden.

- Das obere und das mittlere Panel werden nun ihrerseits durch zwei Panels geteilt, bei denen jeweils die Beschriftung und der Rand entfernt werden sowie die Eigenschaft *BorderWidth* auf 8 gesetzt wird.
- In die beiden rechten Panels wird eine *TDBGrid*-Instanz gesetzt und mit *alClient* ausgerichtet.

Die Liste der Benutzer

Zur Anzeige der Benutzer verwenden wir eine *TQuery/TDataSource*-Kombination mit den Namen *qrBenutzer* und *dsBenutzer*. *DatabaseName* setzen wir auf *_INTERN*, *RequestLive* auf *true* und *SQL* auf folgende Anweisung:

```
SELECT *
  FROM t_benutzer
  ORDER BY ID
```

Nachdem die Abfrage geöffnet wurde, erstellen wir persistente *TField*-Instanzen und hängen über *dsBenutzer* das Grid und einen DBNavigator an. Zur weiteren Anzeige verwenden wir zwei *DBEdit* und ein *DBMemo*. Bei einer Änderung der Daten soll nicht dieses Formular verwendet werden, deshalb setzen wir *BorderStyle* auf *bsNone* und *ParentColor* auf *true*. Die Anzeige sieht nun nach *TDBText* aus, die Daten lassen sich jedoch markieren und somit über die Zwischenablage kopieren.

Des Weiteren ergänzen wir noch den Button *btnEdit* (Beschriftung *Passwort ändern*) und ein Label mit der Beschriftung *Benutzer:*.

Um ein direktes Bearbeiten der Daten auszuschließen, setzen wir *AutoEdit* von *dsBenutzer* auf *false*. Außerdem ergänzen wir *qrBenutzer* um folgende Ereignisbehandlungsroutinen:

```
procedure Tfrm_benutzer.qrBenutzerAfterPost(DataSet: TDataSet);
// für AfterPost und After Delete
begin
  RefreshQuery(qrBenutzer);
  gvChanged := true;
end;

procedure Tfrm_benutzer.qrBenutzerBeforeDelete(DataSet: TDataSet);
begin
  if qrBenutzerID.AsInteger = 1
    then raise Exception.Create('ADMIN kann nicht gelöscht werden');
end;
```

Die *AfterPost*-Ereignisbehandlungsroutine dient zur Aktualisierung der Anzeige, *qrBenutzerBeforeDelete* verhindert, dass der Administrator gelöscht wird.

```
procedure Tfrm_benutzer.qrBenutzerAfterScroll(DataSet: TDataSet);
begin
  if not gvIsAdmin
    then btnEdit.Visible
      := gvUserID = qrBenutzerID.AsInteger;
end;
```

Hat der angemeldete Benutzer keine Administratorrechte, dann bekommt er *btnEdit* nur zu sehen, wenn er auf dem eigenen Datensatz steht.

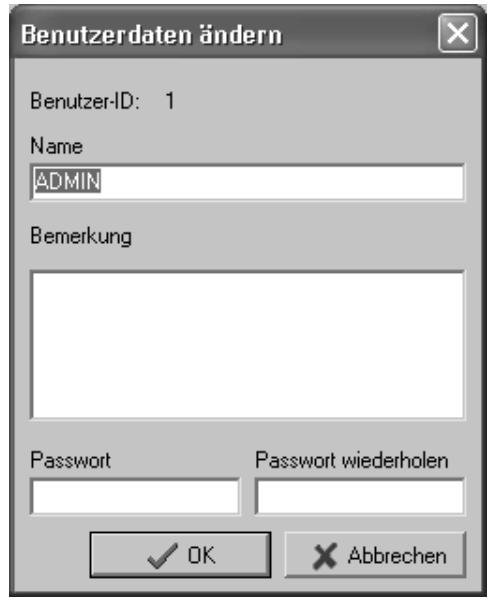
```
procedure Tfrm_benutzer.FormShow(Sender: TObject);
begin
  qrBenutzer.Open;
  qrBenFunk.Open;
  if gvIsAdmin then
  begin
    DBNavigator1.VisibleButtons
      := [nbFirst,nbPrior,nbNext,nbLast,nbInsert,nbDelete,nbEdit];
    DBNavigator1.Width := 175;
  end {if IsAdmin then}
  else
  begin
    DBNavigator1.VisibleButtons
      := [nbFirst,nbPrior,nbNext,nbLast];
    DBNavigator1.Width := 100;
  end; {else IsAdmin then}
  btnDelete.Visible := gvIsAdmin;
  btnClear.Visible := gvIsAdmin;
  btnInsert.Visible := gvIsAdmin;
  if DBGrid1.CanFocus
    then DBGrid1.SetFocus;
end; {procedure Tfrm_benutzer.FormShow}
```

Während der Laufzeit des Programms ändert sich der Benutzer nicht, und wenn der Benutzer sich mit Administratorrechten anmeldet, dann behält er diese bis zum Programmende, auch wenn er sich diese zwischenzeitlich entzieht. Das Setzen der *VisibleButtons* von *DBNavigator1* braucht somit nur beim Anzeigen des Formulars erfolgen. Bei dieser Gelegenheit wird gleich die Sichtbarkeit von *btnDelete*, *btnClear* und *btnInsert* gesteuert (welche wir im Moment noch nicht eingefügt haben).

Um während der Entwicklung nicht unnötig viele Abfragen offen zu haben, bleibt *Active* bei den einzelnen Datenzugriffskomponenten auf *false* – das bedingt allerdings, dass diese Abfragen beim Anzeigen des Formulars geöffnet werden.

Ändern der Benutzerdaten

Für das Anlegen neuer Benutzer und das Ändern bestehender Datensätze verwenden wir ein eigenes Formular.



Das Anzeigen des Formulars erfolgt auf die Ereignisse *AfterInsert* und *AfterEdit*:

```
procedure Tfrm_benutzer.qrBenutzerAfterInsert(DataSet: TDataSet);
// für AfterInsert und AfterEdit
var
  Form: Tfrm_benutzereingabe;
begin
  Form := Tfrm_benutzereingabe.Create(nil);
  try
    if qrBenutzer.State = dsEdit then
    begin
      Form.Edit1.Text
        := Tfrm_passwort.GetPasswort(qrBenutzerPASSWORT);
      Form.Edit2.Text := Form.Edit1.Text;
    end;
    Form.SetWerte(true, dsBenutzer);
```

Zunächst wird das Formular erzeugt. Im Edit-Fall wird das vorhandene Passwort angezeigt, *SetWerte* werden wir uns nachher noch anschauen.

```
    if Form.ShowModal = mrOK then
    begin
      Tfrm_passwort.SetPasswort(qrBenutzerPASSWORT,
        Form.Edit1.Text);
      qrBenutzer.Post;
    end
    else qrBenutzer.Cancel;
  finally
    Form.Free;
  end;
end; {procedure Tfrm_benutzer.qrBenutzerAfterInsert}
```

Wird das Formular mit *OK* geschlossen, dann werden die Änderungen mit *Post* bestätigt, anderenfalls mit *Cancel* verworfen.

Das Formular für die Eingabe (Name *frm_benutzereingabe*, Unit *u_benutzereingabe*) ist mit vier Panels segmentiert, wobei das zweite (von oben) auf *alCient* gesetzt ist, so dass für die Eingabe längerer Bemerkungen einfach nur das Formular vergrößert werden muss. Auf dem dritten Panel (von oben) liegen die Edit-Felder für die Passworteingabe.

Damit dabei die Buttons brav an der rechten Seite bleiben, setzt man deren Eigenschaft *Anchors* auf *[akTop,akRight]*.

Da dieses Formular sowohl für neue Benutzer als auch für neue Funktionen verwendet werden soll, müssen mit *SetWerte* einige Sachen »gerade gebogen« werden:

```
procedure Tfrm_benutzereingabe.SetWerte(const Benutzer: boolean;
  const DataSource: TDataSource);
begin
  pnlPasswort.Visible := Benutzer;
  if Benutzer then
  begin
    if Insert
      then Caption := 'Neuer Benutzer'
      else Caption := 'Benutzerdaten ändern';
  end {if Value then}
  else
  begin
    if Insert
      then Caption := 'Neue Funktion'
      else Caption := 'Funktion ändern';
  end; {else Value then}
```

```
  DBText1.DataSource := DataSource;
  DBEdit1.DataSource := DataSource;
  DBMemo1.DataSource := DataSource;
end; {procedure Tfrm_benutzereingabe.SetIsBenutzer}
```

Je nachdem, ob ein Benutzer oder eine Funktion geändert oder neu angelegt wird, erhält das Formular eine andere Beschriftung. Das Panel für die Passworteingabe wird nur dann angezeigt, wenn ein Benutzerdatensatz geändert oder neu angelegt wird – bei Funktionen macht dies keinen Sinn. Des Weiteren muss für alle datensensitiven Dialogelemente die Eigenschaft *DataSource* gesetzt werden.

```
procedure Tfrm_benutzereingabe.FormShow(Sender: TObject);
begin
  DBEdit1.Enabled := gvIsAdmin;
  DBMemo1.Enabled := gvIsAdmin;
  if DBEdit1.CanFocus
    then DBEdit1.SetFocus
    else if Edit1.CanFocus
      then Edit1.SetFocus;
  if not gvIsAdmin
    then Edit1.PasswordChar := '*';
  Edit2.PasswordChar := Edit1.PasswordChar;
end;
```

Ist der Benutzer nicht der Administrator, dann kann er hier nur sein Passwort ändern, alle anderen Felder sind für die Eingabe gesperrt. Außerdem soll sein Passwort nicht im Klartext angezeigt werden. (Dem Administrator soll damit nicht das Leben schwer gemacht werden, schließlich basteln wir keine »Hochsicherheitsanwendung«.)

```
procedure Tfrm_benutzereingabe.Edit1Change(Sender: TObject);
begin
  Edit2.Text := '';
end;
```

Wird das Passwort in *Edit1* überschrieben, dann wird es für die erneute Eingabe gleich gelöscht.

```
procedure Tfrm_benutzereingabe.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if (ModalResult = mrOK)
    and (Edit1.Text <> Edit2.Text) then
```

```
  begin
    CanClose := false;
    ShowMessage('Die eingegebenen Passwörter differieren!');
  end;
end;
```

Der Versuch, das Formular bei differierenden Passworteingaben mit OK zu schließen, wird mit einer entsprechenden Fehlermeldung abgewiesen.

Die Routinen zum Verschlüsseln und Entschlüsseln des Passworts sind Bestandteil von *Tfrm_passwort* und werden auch dort beschrieben.

Die Funktionen eines Benutzers

Zur Anzeige der Funktionen, die der einzelne Benutzer wahrnimmt, wird folgender JOIN erstellt, der als Detail-Datenmenge zu *qrBenutzer* agiert. Die Eigenschaft *DataSource* von *qrBenFunk* wird somit auf *dsBenutzer* gesetzt.

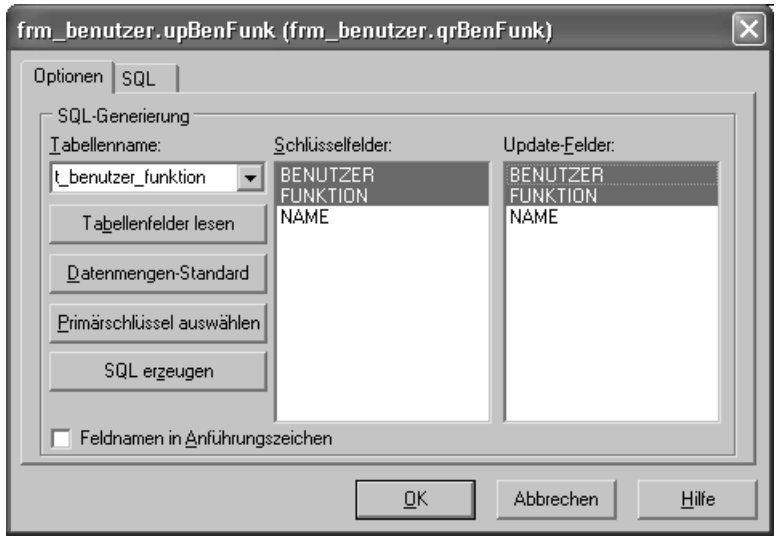
```
SELECT
    m.benutzer,
    m.funktion,
    f.name
  FROM t_benutzer_funktion m
    INNER JOIN t_funktion f
      ON f.id = m.funktion
  WHERE m.benutzer = :ID
  ORDER BY m.funktion
```

Um über diesen JOIN Daten ändern zu können, ist eine *TUpdateSQL*-Komponente nötig. Die Eigenschaft *CachedUpdates* von *qrBenFunk* wird auf *true* und *UpdateObject* auf *upBenFunk* gesetzt.

Mit einem Doppelklick auf *upBenFunk* kommen wir in den SQL-Editor. Hier wählen wir sowohl als Schlüssel- als auch als Update-Felder *Benutzer* und *Funktion*. Mit einem Mausklick auf *SQL erzeugen* erstellen wir dann die folgenden Anweisungen:

```
update t_benutzer_funktion
set
  BENUTZER = :BENUTZER,
  FUNKTION = :FUNKTION
where
  BENUTZER = :OLD_BENUTZER and
  FUNKTION = :OLD_FUNKTION
```

```
insert into t_benutzer_funktion
  (BENUTZER, FUNKTION)
values
  (:BENUTZER, :FUNKTION)

delete from t_benutzer_funktion
where
  BENUTZER = :OLD_BENUTZER and
  FUNKTION = :OLD_FUNKTION
```



Leider hält sich Delphi nicht an die Konventionen, SQL-Schlüsselwörter in Großbuchstaben und alles andere in Kleinbuchstaben zu schreiben.

```
procedure Tfrm_benutzer.btnDeleteClick(Sender: TObject);
// gemeinsam für btnDelete und btnClear
var
  s: string;
begin
  if Sender = btnDelete
    then s := 'Soll dem Benutzer "' +  qrBenutzerNAME.AsString + '"'
      + #13#10 + 'die Funktion "' + qrBenFunkNAME.AsString
      + '" entzogen werden?'
    else s := 'Sollen dem Benutzer "' +  qrBenutzerNAME.AsString
      + '" alle Funktionen entzogen werden?';
```

```
  if MessageDlg(s, mtConfirmation, [mbYes, mbNo], 0) = mrYes then
  begin
    if Sender = btnClear then
    begin
      while qrBenFunk.RecordCount > 0
        do qrBenFunk.Delete;
    end
    else qrBenFunk.Delete;
    qrBenFunk.ApplyUpdates;
    qrBenFunk.CommitUpdates;
    RefreshQuery(qrBenFunk);
  end; {if MessageDlg...}
end; {procedure Tfrm_benutzer.btnDeleteClick}
```

Zum Entfernen der aktuellen Funktion (*btnDelete*) oder aller Funktionen (*btnClear*) wird dieselbe Ereignisbehandlungsroutine verwendet: Zunächst erfolgt eine Sicherheitsabfrage, dann werden die Datensätze gelöscht und zuletzt die zwischengespeicherten Aktualisierungen eingetragen.

Neue Funktionen hinzufügen

Für das Hinzufügen neuer Benutzer wird ein weiteres Formular verwendet, auf dem in einem DBGrid alle definierten Funktionen angezeigt werden. Da die Option *dsMultiSelect* des Grids gesetzt ist, lassen sich mehrere Funktionen markieren und so auf einmal zuweisen.

Dasselbe Formular wird auch dafür verwendet, um einer Funktion neue Benutzer hinzuzufügen. Die darauf verwendete *TQuery*-Instanz hat zwar die folgende SQL-Anweisung

```
SELECT *
  FROM t_funktion
  ORDER BY ID
```

je nach Verwendungszweck wird jedoch die zweite Zeile dieser SQL-Anweisung abgeändert. Darüber hinaus werden Fenster- und Spaltenüberschrift der Verwendung angepasst.

```
procedure Tfrm_FunktionVergeben.SetWerte(const Benutzer: boolean);
begin
  if Benutzer then
  begin
    Caption := 'Benutzer hinzufügen';
    DBGrid1.Columns[1].Title.Caption := 'Benutzer';
    Query1.SQL[1] := '  FROM t_funktion';
  end
  else
  begin
    Caption := 'Funktion vergeben';
    DBGrid1.Columns[1].Title.Caption := 'Funktion';
    Query1.SQL[1] := '  FROM t_benutzer';
  end; {if Benutzer then}
  Query1.Open;
end; {procedure Tfrm_FunktionVergeben.SetWerte}
```

Beim Mausklick auf den Button *Funktion(en) hinzufügen* wird zunächst eine Instanz von *Tfrm_FunktionVergeben* erzeugt und diese modal angezeigt. Wird dieser Dialog mit OK geschlossen, dann wird durch die markierten Einträge iteriert und dafür die lokale Prozedur *FunktionEintragen* aufgerufen.

Dort wird zunächst geprüft, ob es einen entsprechenden Eintrag bereits gibt, anderenfalls wird dieser erstellt. Zuletzt werden die zwischengespeicherten Aktualisierungen eingetragen und die Abfrage wird neu geöffnet.

```
procedure Tfrm_benutzer.btnInsertClick(Sender: TObject);
var
  i: integer;
  funktion: integer;
  Form: Tfrm_FunktionVergeben;
```

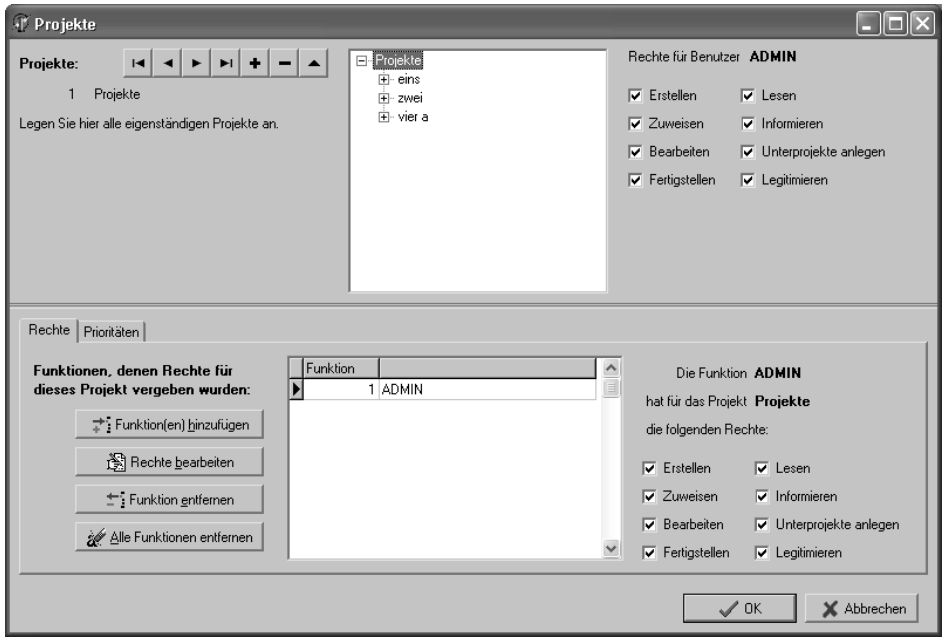
```
  procedure FunktionEintragen(Funktion: integer);
  begin
    if not qrBenFunk.Locate('Funktion', Variant(funktion), []) then
    begin
      qrBenFunk.Insert;
      qrBenFunkBENUTZER.AsInteger := qrBenutzerID.AsInteger;
      qrBenFunkFUNKTION.AsInteger := funktion;
      qrBenFunk.Post;
    end;
  end;

begin
  Form := Tfrm_FunktionVergeben.Create(nil);
  try
    Form.SetWerte(true);
    if Form.ShowModal = mrOK then
    begin
      with Form.DBGrid1, Form.Query1 do
      begin
        for i := 0 to SelectedRows.Count - 1 do
        begin
          GotoBookmark(Pointer(SelectedRows.Items[i]));
          FunktionEintragen(FieldByName('ID').AsInteger);
        end; {for i := 0 to SelectedRows.Count - 1 do}
        qrBenFunk.ApplyUpdates;
        qrBenFunk.CommitUpdates;
        RefreshQuery(qrBenFunk);
      end; {with frm_data, frm_FunktionVergeben.DBGrid1 do}
    end; {if ShowModal = mrOK then}
  finally
    Form.Free;
  end;
end; {procedure Tfrm_benutzer.btnInsertClick}
```

Die Verwaltung der Funktionen

Die Verwaltung der Funktionen ist fast identisch aufgebaut (es sind keine Passwörter einzugeben), es werden lediglich andere Tabellen verwendet. Deshalb sollen diese Units nicht eigens besprochen werden.

18.2.2 Verwaltung der Projekte

Jeder Eintrag ist einem Projekt zugeordnet. Einem Projekt zugeordnet sind Rechte für die einzelnen Funktionen und Prioritäten. Die Projekte können dabei hierarchisch untergliedert werden – hat eine Gruppe Rechte für ein Projekt, dann hat sie dieselben Rechte auch für dessen Unterprojekte.



Das Formular (*frm_projekte*, Name der Unit somit *u_projekte*) wird wieder mit drei Panels in drei Bereiche geteilt, wobei zwischen dem ersten und dem zweiten (von oben) ein Splitter eingefügt wird. Auf dem untersten finden sich dann die Buttons *btnOk* und *btnAbbruch*. Die Breite der Formulars wird über die Eigenschaften *Constraints.MinWidth* und *Constraints.MaxWidth* auf 800 Pixel fixiert.

Anzeige der Projekte

Mit Hilfe von *qrProjekt* erhält man eine Liste aller Projekte. Die dazugehörende SQL-Anweisung ist ziemlich simpel:

```
SELECT *
  FROM t_projekt
  ORDER BY ID
```

Um den hierarchischen Aufbau zu visualisieren, wird *TreeView1* verwendet. In der *FormShow*-Ereignisbehandlungsroutine wird zu diesem Zweck die Routine *RefreshTree(TreeView1)* aufgerufen.

```
procedure Tfrm_projekte.RefreshTree(Tree: TTreeView);
var
  id: integer;

  procedure DatenEinlesen;
  begin
    with qrProjekt do
    begin
      id := qrProjektID.AsInteger;
      Close;
      Open;
      while not EOF do
      begin
        if qrProjektMASTER.IsNull
          then Tree.Items.AddObject(nil,
            qrProjektNAME.AsString, Pointer(qrProjektID.AsInteger))
          else Tree.Items.AddChildObject(
            GetNodeByID(Tree, qrProjektMASTER.AsInteger),
            qrProjektNAME.AsString, Pointer(qrProjektID.AsInteger));
        Next;
      end; {while not EOF do}
      Locate('ID', Variant(id), []);
    end; {with qrProjekt do}
  end; {procedure DatenEinlesen}

begin
  CanSync := false;
  Tree.Items.BeginUpdate;
  Tree.Items.Clear;
  try
    try
      qrProjekt.DisableControls;
      qrRecht.Close;
      DatenEinlesen;
    finally
      qrProjekt.EnableControls;
      qrRecht.Open;
    end;
```

```
  finally
    Tree.Items.EndUpdate;
    CanSync := true;
  end;
end; {procedure Tfrm_projekte.RefreshTree}
```

RefreshTree wird nicht nur beim Anzeigen des Formulars aufgerufen, sondern auch dann, wenn ein Projekt hinzugefügt wird. Deswegen werden zunächst die bisherigen Einträge des Baums gelöscht. Um die Ausführung dieser Prozedur nicht unnötig zu hemmen, wird die Aktualisierung der Anzeige und das Ermitteln der jeweiligen Rechte vorübergehend ausgesetzt.

Das eigentliche Aufbauen des Baums wird dann von der lokalen Prozedur *DatenEinlesen* vorgenommen. Zunächst wird die Position des Datenzeigers zwischengespeichert, dann mit *Close* und *Open* die Anzeige aktualisiert. In einer Schleife wird nun durch alle Einträge iteriert.

Hat der jeweilige Datensatz als Master den Wert NULL – das dürfte nur beim Wurzeleintrag *Projekte* der Fall sein –, dann wird der Eintrag auf der obersten Ebene eingefügt. Hat ein Datensatz einen Master, dann wird der neue Eintrag unter dessen Eintrag erstellt. In die Eigenschaft *Data* eines jeden Eintrags wird dessen ID gespeichert, dazu muss eine Typenumwandlung nach *Pointer* erfolgen.

Mit dieser ID erhält man dann mittels der Routine *GetNodeByID* einen Zeiger auf den Eintrag:

```
function Tfrm_projekte.GetNodeByID(Tree: TTreeView;
  ID: integer): TTreeNode;
var
  i: integer;
begin
  result := nil;
  for i := 0 to Tree.Items.Count - 1 do
  begin
    if integer(Tree.Items[i].Data) = ID then
    begin
      result := Tree.Items[i];
      exit;
    end;
  end; {for i := 0 to TreeView1.Items.Count - 1 do}
end; {function Tfrm_projekte.GetNodeByID}
```

Zu diesem Zweck wird durch den Baum iteriert und der Wert in *Data* mit dem Parameter *ID* verglichen. Im Erfolgsfall wird ein Zeiger auf den entsprechenden Eintrag zurückgegeben.

Mit einem Mausklick auf den entsprechenden Eintrag wird das dazugehörende Projekt gewählt, zur Anzeige der Bemerkung muss der Satzzeiger von *qrProjekt* entsprechend positioniert werden.

```
procedure Tfrm_projekte.TreeView1Change(Sender: TObject;
  Node: TTreeNode);
begin
  if Assigned(Node)
    then qrProjekt.Locate('ID', Variant(integer(Node.Data)), []);
end;
```

Umgekehrt soll nach dem Scrollen der aktuelle Datensatz im Tree markiert werden.

```
procedure Tfrm_projekte.qrProjektAfterScroll(DataSet: TDataSet);
begin
  if CanSync then
  begin
    FProjektID := qrProjektID.AsInteger;
    TreeView1.Selected
      := GetNodeByID(TreeView1, FProjektID);
    ...
  end; {if CanSync then}
end; {procedure Tfrm_projekte.qrProjektAfterScroll}
```

Dazu wird ein Zeiger auf den Eintrag mit *GetNodeByID* ermittelt und dieser der Eigenschaft *Selected* zugewiesen. Anschließend werden die zum jeweiligen Recht gehörenden Rechte ermittelt, was wir später besprechen werden.

Projekte hinzufügen und ändern

Das Recht, Projekte hinzuzufügen, zu ändern und zu löschen bleibt den Benutzern mit Administratorrechten vorbehalten. Anderen Benutzern kann jedoch das Recht eingeräumt werden, zu einem Projekt Unterprojekte anzulegen – dafür wird ihnen dann ein eigener Button eingeblendet.

```
procedure Tfrm_projekte.FormShow(Sender: TObject);
begin
  qrProjekt.Open;
  qrRecht.Open;
  qrBenRechte.Open;
  qrPrios.Open;
  qrPriosAll.Open;
  RefreshTree(TreeView1);
  TreeView1.Items[0].Expand(false);
```

```
  if gvIsAdmin then
  begin
    DBNavigator1.VisibleButtons
      := [nbFirst,nbPrior,nbNext,nbLast,nbInsert,nbDelete,nbEdit];
    DBNavigator1.Width := 175;
  end {if IsAdmin then}
  else
  begin
    DBNavigator1.VisibleButtons
      := [nbFirst,nbPrior,nbNext,nbLast];
    DBNavigator1.Width := 100;
  end; {else IsAdmin then}
  RechteVergebenVisible(gvIsAdmin);
  PriosVergebenVisible(gvIsAdmin);
  lblBenutzer.Caption := gvUserName;
  if TreeView1.CanFocus
    then TreeView1.SetFocus;
end; {procedure Tfrm_projekte.FormShow}
```

In *FormShow* werden zunächst die Abfragen geöffnet und dann wird der Baum aufgebaut. Der Eintrag *Projekte* wird expandiert, so dass alle Einträge der ersten Ebene sichtbar sind. Danach wird die Menge der sichtbaren Navigator-Buttons gesetzt, ebenso die Buttons zur Vergabe von Rechten und zum Erstellen von Prioritäten.

Für das Bearbeiten bestehender und das Einfügen neuer Einträge wird das Formular *frm_projektbearbeiten* (Unit *u_projektbearbeiten*) verwendet.

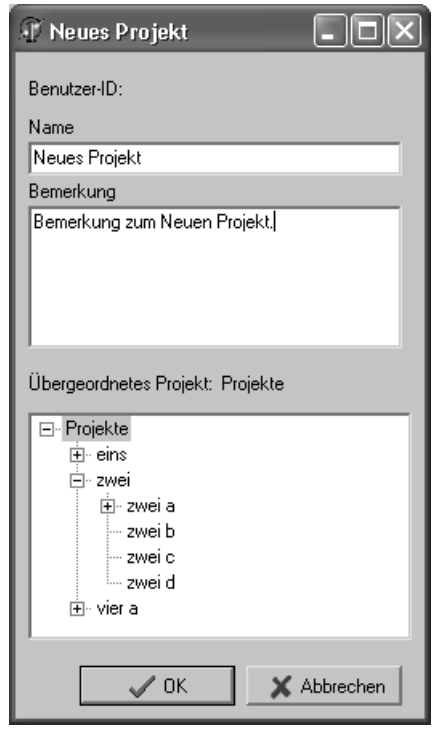
```
procedure Tfrm_projekte.qrProjektAfterInsert(DataSet: TDataSet);
var
  Form: Tfrm_projektbearbeiten;
begin
  Form := Tfrm_projektbearbeiten.Create(nil);
  try
    Form.Caption := 'Neues Projekt';
    Form.SetWerte(gvIsAdmin, dsProjekt);
    Form.TreeView1.Items.Assign(TreeView1.Items);
    Form.TreeView1.Selected
      := GetNodeByID(Form.TreeView1, FProjektID);
    if Form.ShowModal = mrOK then
    begin
      if Assigned(Form.TreeView1.Selected)
        then qrProjektMASTER.AsInteger
```

```
          := integer(Form.TreeView1.Selected.Data)
        else raise Exception.Create('Kein Master');
      qrProjekt.Post;
    end
    else qrProjekt.Cancel;
  finally
    Form.Free;
  end;
end; {procedure Tfrm_projekte.qrProjektAfterInsert}
```

Beim Erstellen neuer Projekte wird nach dem Erzeugen des Formulars zunächst einmal dessen Projektbaum aufgebaut und das aktuelle Projekt selektiert; dies würde dann zum Master des neuen Projektes.

Wird das Formular mit *Ok* geschlossen, dann wird geprüft, ob ein Master-Projekt gewählt wurde. Ist dies der Fall, dann wird dessen Nummer in das Feld *qrProjektMASTER* geschrieben, anderenfalls wird eine Exception ausgelöst.

Wird der Button *Abbrechen* gewählt, dann wird die Änderung mit *Cancel* verworfen. Zuletzt wird das Formular wieder freigegeben. Beachten Sie bitte, dass diese Änderungen immer noch über die Transaktion rückgängig gemacht werden können.



Schauen wir uns kurz die drei Routinen von *Tfrm_projektbearbeiten* an:

```
procedure Tfrm_projektbearbeiten.FormShow(Sender: TObject);
begin
  if DBEdit1.CanFocus
    then DBEdit1.SetFocus;
end;
```

DBEdit1 soll zunächst den Focus erhalten.

```
procedure Tfrm_projektbearbeiten.SetWerte(PanelVisible: boolean;
  DataSource: TDataSource);
begin
  pnlUeber.Visible := PanelVisible;
  if PanelVisible
    then Height := 480
    else Height := 300;
  DBEdit1.DataSource := DataSource;
  DBMemo1.DataSource := DataSource;
end;
```

Hat der Anwender keine Administratorrechte, dann kann er das übergeordnete Projekt nicht verändern, *pnlUeber* wird somit ausgeblendet, des Weiteren wird dem Formular eine entsprechende Höhe zugewiesen. Außerdem wird den beiden datensensitiven Dialogelementen ein Wert für *DataSource* zugewiesen.

```
procedure Tfrm_projektbearbeiten.TreeView1Change(Sender: TObject;
  Node: TTreeNode);
begin
  if TreeView1.Items.Count > 0
    then lblUeber.Caption := Node.Text
    else lblUeber.Caption := '';
end;
```

Der gewählte übergeordnete Eintrag wird in *lblUeber* angezeigt, und zwar auch dann, wenn *pnlUeber* ausgeblendet ist.

Ähnlich wie die *AfterInsert*- sieht die *AfterEdit*-Ereignisbehandlungsroutine aus. Es gibt jedoch einige Unterschiede:

```
procedure Tfrm_projekte.qrProjektAfterEdit(DataSet: TDataSet);
var
  Form: Tfrm_projektbearbeiten;
begin
  Form := Tfrm_projektbearbeiten.Create(nil);
```

```
  try
    Form.SetWerte(true, dsProjekt);
    Form.TreeView1.Items.Assign(TreeView1.Items);
    GetNodeByID(Form.TreeView1, qrProjektID.AsInteger).Delete;
    Form.TreeView1.Selected
      := GetNodeByID(Form.TreeView1, qrProjektMASTER.AsInteger);
    if Form.ShowModal = mrOK then
    begin
      if Assigned(Form.TreeView1.Selected)
        then qrProjektMASTER.AsInteger
          := integer(Form.TreeView1.Selected.Data);
      qrProjekt.Post;
    end
    else qrProjekt.Cancel;
  finally
    Form.Free;
  end;
end; {procedure Tfrm_projekte.qrProjektAfterEdit}
```

Da nur die Administratoren überhaupt den Button zum Ändern eines Projektes angezeigt bekommen, kann man in *SetWerte* den Parameter *PanelVisible* auf *true* setzen.

Beim Ändern eines Eintrags kann man auch das übergeordnete Projekt abändern, das Projekt also mit allen seinen Unterprojekten verschieben – allerdings nicht unter einen Eintrag, der dem zu verschiebenden untergeordnet ist. Deswegen wird der aktuelle Eintrag (mit allen seinen Unterprojekten) aus dem Baum gelöscht.

```
procedure Tfrm_projekte.qrProjektAfterPost(DataSet: TDataSet);
// für AfterPost und AfterDelete
var
  i: integer;
begin
  i := FProjektID;
  RefreshQuery(qrProjekt);
  gvChanged := true;
  RefreshTree(TreeView1);
  qrProjekt.Locate('ID', Variant(i), []);
end;
```

Nach *Post* und *Delete* muss die Abfrage aktualisiert und der Baum neu aufgebaut werden.

Die Rechte des angemeldeten Benutzers

Zum aktuellen Projekt sollen die Rechte des angemeldeten Benutzers ermittelt werden. Darüber wird dann gesteuert, ob beispielsweise die Register-Reiter für *Rechte* und *Prioritäten* überhaupt sichtbar sind.

Die Verwendung einer STORED PROCEDURE macht hier die SQL-Anweisung von *qrBenRechte* sehr einfach:

```
SELECT *
  FROM p_checkrechte(:projekt, :benutzer)
```

Naturgemäß fällt dann die STORED PROCEDURE etwas umfangreicher aus, zumal die Rechte an der Funktion und nicht am Benutzer hängen. Ob ihrer Länge wollen wir diese Prozedur in Abschnitten besprechen.

```
CREATE PROCEDURE p_checkrechte
  (para_projekt INTEGER,
  para_benutzer INTEGER)
RETURNS
  (projekt INTEGER,
  funktion INTEGER,
  erstellen CHAR(1),
  zuweisen CHAR(1),
  bearbeiten CHAR(1),
  fertigstellen CHAR(1),
  lesen CHAR(1),
  informieren CHAR(1),
  legitimieren CHAR(1),
  unterprojekte CHAR(1))
AS
  DECLARE VARIABLE v_projekt INTEGER;
  DECLARE VARIABLE v_erstellen CHAR(1);
  DECLARE VARIABLE v_zuweisen CHAR(1);
  DECLARE VARIABLE v_bearbeiten CHAR(1);
  DECLARE VARIABLE v_fertigstellen CHAR(1);
  DECLARE VARIABLE v_lesen CHAR(1);
  DECLARE VARIABLE v_informieren CHAR(1);
  DECLARE VARIABLE v_legitimieren CHAR(1);
  DECLARE VARIABLE v_unterprojekte CHAR(1);
```

Für die einzelnen Rechte gibt es jeweils sowohl einen Rückgabe-Parameter als auch eine Variable.

```
BEGIN
  erstellen = 'N';
  zuweisen = 'N';
  bearbeiten = 'N';
  fertigstellen = 'N';
  lesen = 'N';
  informieren = 'N';
  legitimieren = 'N';
  unterprojekte = 'N';
  IF ((:para_projekt = 0)
    OR (:para_benutzer = 0)) THEN
  BEGIN
    SUSPEND;
    EXIT;
  END
```

Zunächst werden alle Rechte auf *N* gesetzt. Ist das Projekt oder der Benutzer nicht gesetzt, dann wird die Ausführung der Prozedur an dieser Stelle abgebrochen.

```
  v_projekt = para_projekt;
  WHILE (v_projekt IS NOT NULL) DO
  BEGIN
```

In der Variablen *v_projekt* steht die Nummer des Projektes, für das jeweils die Rechte ermittelt werden sollen. In der WHILE-Schleife wird dann *v_projekt* solange auf das jeweils direkt übergeordnete Projekt gesetzt, bis man beim Wurzel-Projekt angelangt ist.

```
    FOR SELECT
      r.funktion,
      r.erstellen,
      r.zuweisen,
      r.bearbeiten,
      r.fertigstellen,
      r.lesen,
      r.informieren,
      r.legitimieren,
      r.unterprojekte
    FROM t_benutzer_funktion b
      INNER JOIN t_recht r
        ON r.funktion = b.funktion
```

```
WHERE b.benutzer = :para_benutzer
  AND r.projekt = :v_projekt
INTO
  :funktion,
  :v_erstellen,
  :v_zuweisen,
  :v_bearbeiten,
  :v_fertigstellen,
  :v_lesen,
  :v_informieren,
  :v_legitimieren,
  :v_unterprojekte
DO
BEGIN
  IF (v_erstellen = 'J')
    THEN erstellen = 'J';
  IF (v_zuweisen = 'J')
    THEN zuweisen = 'J';
  IF (v_bearbeiten = 'J')
    THEN bearbeiten = 'J';
  IF (v_fertigstellen = 'J')
    THEN fertigstellen = 'J';
  IF (v_lesen = 'J')
    THEN lesen = 'J';
  IF (v_informieren = 'J')
    THEN informieren = 'J';
  IF (v_legitimieren = 'J')
    THEN legitimieren = 'J';
  IF (v_unterprojekte = 'J')
    THEN unterprojekte = 'J';
  PROJEKT = :v_projekt;
END /* FOR SELECT */
```

Mit einem JOIN von *t_benutzer_funktion* und *t_recht*, gefiltert auf den jeweiligen Benutzer und das aktuelle Projekt, erhält man die Rechte, die der betreffende Benutzer auf das aktuelle Projekt hat. Hat er eins oder mehrere dieser Rechte, dann wird der dazugehörende Ausgabeparameter auf *J* gesetzt.

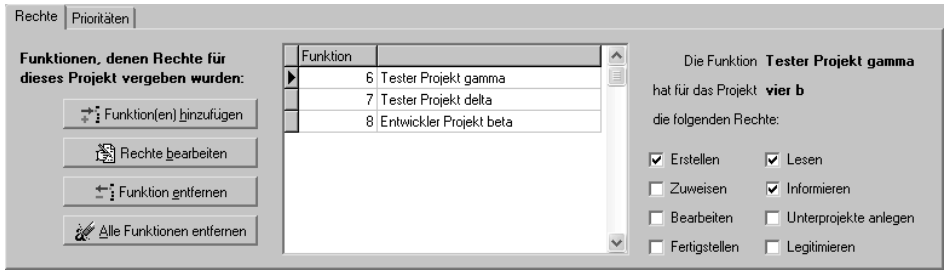
```
SELECT
    master
  FROM t_projekt
  WHERE ID = :v_projekt
```

```
      INTO :v_projekt;
  END /* WHILE (v_projekt IS NOT NULL) DO */
  SUSPEND;
END
```

Nun wird noch das übergeordnete Projekt ermittelt und dessen Nummer in *v_projekt* geschrieben. Mit SUSPEND werden schließlich die Werte übergeben.

Vergabe von Rechten

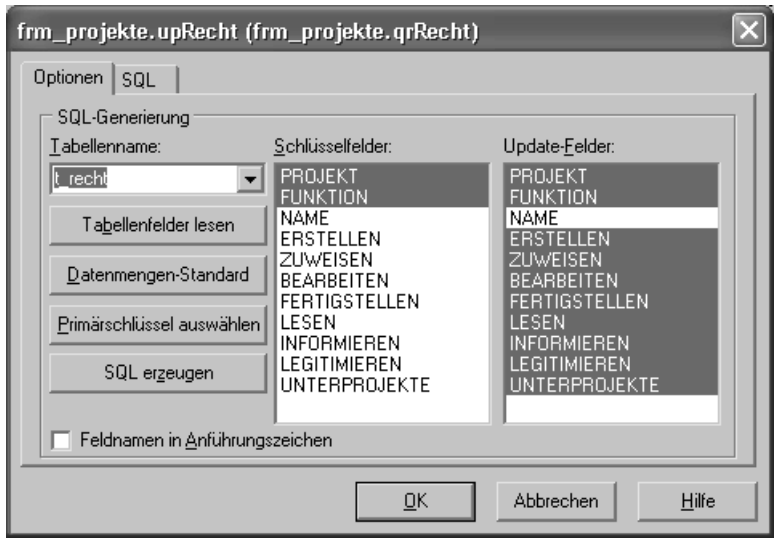
In das Panel unter dem Splitter setzen wir eine *TPageControl*-Instanz mit den Registerseiten *tbsRechte* und *tbsPrios*. Mittels Panels werden die beiden Registerseiten in je drei Abschnitte geteilt.



Die Rechte eines Benutzers hängen an seiner Funktion und am Projekt. Die Abfrage in *qrRecht* ist eine Detail-Datenmenge zu qrProjekt. Damit auch die dazugehörende Funktion dargestellt werden kann, erfolgt ein JOIN auf *t_funktion*.

```
SELECT
    r.projekt,
    r.funktion,
    f.name,
    r.erstellen,
    r.zuweisen,
    r.bearbeiten,
    r.fertigstellen,
    r.lesen,
    r.informieren,
    r.legitimieren,
    r.unterprojekte
  FROM t_recht r
    INNER JOIN t_funktion f
      ON r.funktion = f.ID
  WHERE r.projekt = :id
```

Um visualisieren zu können, welche Rechte einer Funktion zugewiesen sind, werden datensensitive Checkboxen verwendet. Zur Aktualisierung von *qrRechte* wird die *TUpdateSQL*-Instanz *upRechte* verwendet.



Das es sich bei *t_recht* um eine Verknüpfungstabelle zwischen *t_projekt* und *t_funktion* handelt, müssen sowohl die Spalte *Projekt* als auch die Spalte *Funktion* als Schlüsselfelder gewählt werden. Da beim Erstellen einer Verknüpfung beide Spalten gesetzt werden müssen, gehören sie auch zu den Update-Feldern – lediglich der Name der Funktion kann hier außen vor bleiben.

```
update t_recht
set
  PROJEKT = :PROJEKT,
  FUNKTION = :FUNKTION,
  ERSTELLEN = :ERSTELLEN,
    ...
  UNTERPROJEKTE = :UNTERPROJEKTE
where
  PROJEKT = :OLD_PROJEKT and
  FUNKTION = :OLD_FUNKTION

insert into t_recht
  (PROJEKT, FUNKTION, ERSTELLEN ... UNTERPROJEKTE)
values
  (:PROJEKT, :FUNKTION, :ERSTELLEN ... UNTERPROJEKTE)
```

```
delete from t_recht
where
  PROJEKT = :OLD_PROJEKT and
  FUNKTION = :OLD_FUNKTION
```

(Die verkürzte Darstellung mit den drei Pünktchen ist natürlich im Original so nicht zu finden.)

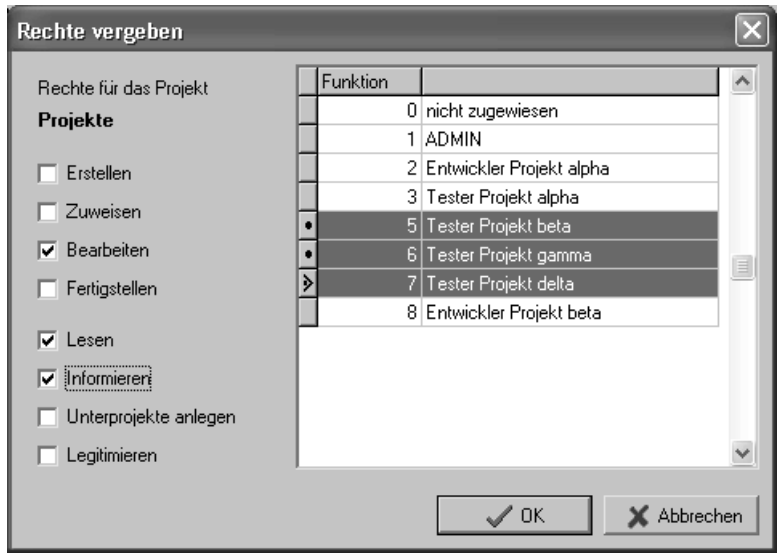
Bei den Fenstern zu Benutzern und Funktionen haben sich *btnDelete* und *btnClear* jeweils eine *OnClick*-Ereignisbehandlungsroutine geteilt. Hier wollen wir das zur Abwechslung einmal anders lösen:

```
procedure Tfrm_projekte.btnDeleteReClick(Sender: TObject);
begin
  if MessageDlg('Sollen die Rechte der Funktion "'
       +  qrRechtNAME.AsString + '"' + #13#10
       + 'dem Projekt "' + qrProjektNAME.AsString
       + '" entzogen werden?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes then
  begin
    qrRecht.Delete;
    UpdateRefreshQuery(qrRecht, '');
  end;
end;

procedure Tfrm_projekte.btnClearReClick(Sender: TObject);
begin
  if MessageDlg('Sollen die Rechte aller Funktionen' + #13#10
       + 'dem Projekt "' + qrProjektNAME.AsString
       + '" entzogen werden?', mtConfirmation,
   [mbYes, mbNo], 0) = mrYes then
  begin
    with qrRecht do
    begin
      while RecordCount > 0
        do Delete;
    end; {with frm_DataVerwVerw.qrRecht do}
    UpdateRefreshQuery(qrRecht, '');
  end; {if MessageDlg...}
end; {procedure Tfrm_projekte.btnClearClick}
```

Das Hinzufügen neuer Datensätze zeigt drei Detailprobleme, welche die Verwendung von datensensitiven Checkboxen erschweren:

- Es soll möglich sein, Rechte gleich mehreren Funktionen zuzuweisen.
- Den Status *cdGrayed* können wir nach dem Einfügen eines neuen Datensatzes nicht gebrauchen.
- Die Vergabe eines neuen Rechtes zieht das Leserecht für dieses Projekt nach sich. Dafür sorgt zwar schon ein entsprechender Trigger in der Datenbank, dennoch wollen wir dieses Verhalten auch visualisieren.

Aus diesem Grund wollen wir normale Checkboxen verwenden, daneben noch ein DBGrid zur Anzeige und Auswahl der Funktionen, für welche die Rechte vergeben werden sollen.



Das Formular *frm_rechte* (Unit *u_rechte*) enthält lediglich zwei kleine Routinen, von denen die eine die Abfrage öffnet und die andere dafür sorgt, dass immer auch das Leserecht markiert ist, wenn ein anderes Recht vergeben wird.

```
procedure Tfrm_rechte.FormCreate(Sender: TObject);
begin
  Query1.Open;
end;

procedure Tfrm_rechte.cbFertigstellenClick(Sender: TObject);
begin
  if (Sender as TCheckBox).Checked
    then cbLesen.Checked := true;
end;
```

Query1 liefert eine Liste aller Funktionen. Vergessen Sie nicht, die Option *dgMultiSelect* von *DBGrid1* auf *true* zu setzen.

```
SELECT *
  FROM t_funktion
  ORDER BY ID
```

Für das Einfügen neuer und das Ändern bestehender Rechte wird die gleiche Routine verwendet, die wir ob ihrer Länge in Abschnitten besprechen wollen, und zwar zunächst den Hauptteil und dann die lokalen Prozeduren.

```
procedure Tfrm_projekte.btnInsertReClick(Sender: TObject);
// für btnInsertRe und btnEditRe
var
  i: integer;
  Form: Tfrm_rechte;

  ...

begin
  Form := Tfrm_rechte.Create(nil);
  try
    CheckBoxenSetzen;
    Form.DBGrid1.Visible := (Sender = btnInsertRe);
    Form.Query1.Locate('ID', Variant(qrRechtFUNKTION.AsInteger), []);
    if Form.ShowModal = mrOK then
    begin
      if Sender = btnInsertRe then
      begin
        with Form.DBGrid1 do
        begin
          for i := 0 to SelectedRows.Count - 1 do
          begin
            DataSource.DataSet.GotoBookmark
              (Pointer(SelectedRows.Items[i]));
            RechteEintragen(Form.Query1ID.AsInteger);
          end; {for i := 0 to SelectedRows.Count - 1 do}
        end; {with frm_rechte.DBGrid1 do}
      end; {if Sender = btnInsert then}
      if Sender = btnEditRe
        then RechteEintragen(qrRechtFUNKTION.AsInteger);
      UpdateRefreshQuery(qrRecht, 'Funktion');
    end; {if frm_rechte.ShowModal = mrOK then}
```

```
  finally
    Form.Free;
  end;
end; {procedure Tfrm_projekte.btnInsertClick}
```

Zunächst wird das Formular erzeugt und anschließend mittels der lokalen Prozedur *CheckBoxenSetzen* der aktuelle Rechte-Zustand abgebildet. Das Grid zur Auswahl mehrerer Funktionen steht nur im Insert-Fall zur Verfügung, für den Edit-Fall wird der Satzzeiger von *Query1* entsprechend positioniert.

Wird der Dialog mit *Ok* geschlossen, dann werden die Rechte mit der lokalen Prozedur *RechteEintragen* in die Datenbank geschrieben – im Edit-Fall für den aktuellen Datensatz, im Insert-Fall für alle Funktionen, die in *Form.DBGrid1* markiert sind.

Schauen wir uns die lokalen Prozeduren zum Setzen der CheckBoxen an:

```
  procedure CheckSetzen(Field: TField; Check: TCheckBox);
  begin
    if Sender = btnEditRe
      then Check.Checked := (Field.AsString = 'J');
    if Sender = btnInsertRe
      then Check.Checked := false;
  end;

  procedure CheckBoxenSetzen;
  begin
    with Form do
    begin
      CheckSetzen(qrRechtERSTELLEN, cbErstellen);
      CheckSetzen(qrRechtZUWEISEN, cbZuweisen);
      CheckSetzen(qrRechtBEARBEITEN, cbBearbeiten);
      CheckSetzen(qrRechtFERTIGSTELLEN, cbFertigstellen);
      CheckSetzen(qrRechtLESEN, cbLesen);
      CheckSetzen(qrRechtINFORMIEREN, cbInformieren);
      CheckSetzen(qrRechtLEGITIMIEREN, cbLegitimieren);
      CheckSetzen(qrRechtUNTERPROJEKTE, cbUnterprojekte);
    end;
  end; {procedure CheckBoxenSetzen}
```

Das Setzen des Status übernimmt wiederum eine lokale Routine, nämlich *CheckSetzen*. Bei einem neuen Datensatz wird die CheckBox immer auf nicht markiert gesetzt, bei einem bestehenden hängt dies davon ab, ob das Recht vergeben wurde oder nicht.

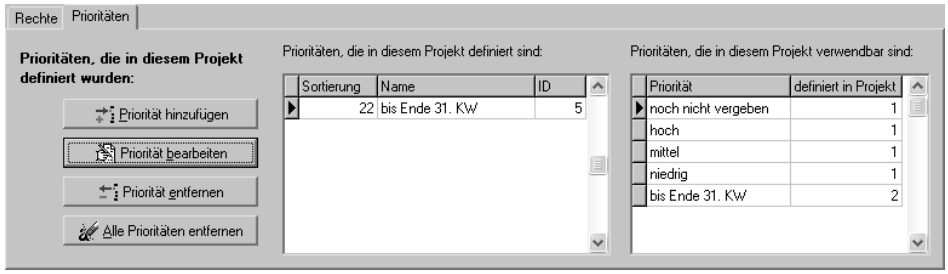
```
procedure Feldwert(Field: TField; Check: TCheckBox);
begin
  if Check.Checked
    then Field.AsString := 'J'
    else Field.AsString := 'N'
end;

procedure RechteEintragen(Funktion: integer);
begin
  with Form, qrRecht do
  begin
    if not Locate('Funktion', Variant(Funktion), []) then
    begin
      Insert;
      qrRechtFUNKTION.AsInteger := Funktion;
      qrRechtPROJEKT.AsInteger := qrProjektID.AsInteger;
    end
    else Edit;
    FeldWert(qrRechtERSTELLEN, cbErstellen);
    FeldWert(qrRechtZUWEISEN, cbZuweisen);
    FeldWert(qrRechtBEARBEITEN, cbBearbeiten);
    FeldWert(qrRechtFERTIGSTELLEN, cbFertigstellen);
    FeldWert(qrRechtLESEN, cbLesen);
    FeldWert(qrRechtINFORMIEREN, cbInformieren);
    FeldWert(qrRechtLEGITIMIEREN, cbLegitimieren);
    FeldWert(qrRechtUNTERPROJEKTE, cbUnterprojekte);
    Post;
  end; {with Form, qrRecht do}
end; {procedure RechteEintragen}
```

Umgekehrt beim Zurückschreiben des Datensatzes: Hier wird je nach dem Status der CheckBox von *Feldwert* entweder ein *J* oder ein *N* zurückgegeben. Vor dem Schreiben in die Datenbank wird geprüft, ob es eine entsprechende Kombination von Recht und Funktion schon gibt. Ist dies der Fall, dann wird diese geändert, andernfalls wird ein neuer Datensatz angelegt.

Die Prioritäten

Jedem Eintrag kann eine Priorität zugeordnet werden. Die Prioritätsstufen werden dabei einem Projekt zugeordnet und sind dann auch in allen dessen Unterprojekten verügbar. Die Priorität kann eine solche im strengen Sinne sein (*hoch*, *mittel*, *niedrig*), es können aber auch Stichtage (*bis Ende 31. KW*) oder sonstige beliebige Texte eingegeben werden.

Eine Liste der offenen Einträge kann nach Prioritäten sortiert werden. Damit das Programm weiß, ob nun *hoch* oder *bis Ende 31. KW* zuerst in der Liste erscheinen soll, wird jeder Priorität noch eine Zahl zwischen 0 und 100 mitgegeben, welche die Sortierreihenfolge bestimmt.



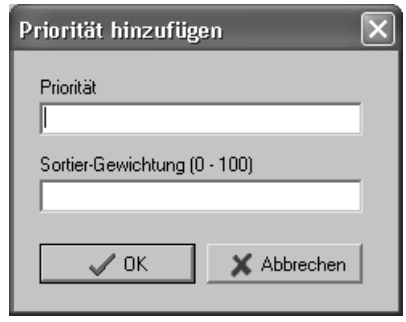
Die Registerseite für die Prioritäten wird mittels Panels wieder in drei Teile geteilt. In der Mitte werden die Prioritäten angezeigt, welche in diesem Projekt vergeben wurden, rechts die Prioritäten, die in diesem Projekt verfügbar sind (also auch solche aus übergeordneten Projekten).

Schauen wir uns zunächst die im jeweiligen Projekt vergebenen Prioritäten an. Die Komponente *qrPrios* verwendet die folgende SQL-Anweisung:

```
SELECT *
  FROM t_prioritaet
  WHERE projekt = :ID
  ORDER BY sortier
```

Routinen zum Löschen von einem oder allen Datensätzen mit vorheriger Sicherheitsabfrage waren auf den vergangenen Seiten schon häufiger beschrieben, wir wollen es dabei belassen.

Für das Einfügen neuer und das Ändern bestehender Prioritäten wurde wieder ein eigenes Formular erstellt:



Für dieses Formular gibt es nur zwei Methoden: Die erste verhindert die Eingabe von Buchstaben bei den Prioritäten, die andere setzt den Focus beim Aufruf:

```
procedure Tfrm_prios.Edit2KeyPress(Sender: TObject; var Key: Char);
begin
  if not (Key in ['0'..'9', #8])
    then Key := #0;
end;

procedure Tfrm_prios.FormShow(Sender: TObject);
begin
  if DBEdit1.CanFocus
    then DBEdit1.SetFocus;
end;
```

Für das Einfügen neuer und das Ändern bestehender Datensätze wird dann wieder die gleiche Routine verwendet:

```
procedure Tfrm_projekte.btnInsertPriClick(Sender: TObject);
// für btnInsert und btnEdit
var
  i: integer;
  Form: Tfrm_prios;
begin
  Form := Tfrm_prios.Create(nil);
  try
    if Sender = btnInsertPri then
    begin
      qrPrios.Insert;
      qrPriosPROJEKT.AsInteger := qrProjektID.AsInteger;
      Form.Caption := 'Priorität hinzufügen';
    end
    else qrPrios.Edit;
    Form.DBEdit1.DataSource := dsPrios;
    Form.DBEdit2.DataSource := dsPrios;
    if Form.ShowModal = mrOK then
    begin
      qrPrios.Post;
      UpdateRefreshQuery(qrPrios, 'ID');
      RefreshQuery(qrPriosAll);
    end {if frm_prios.ShowModal = mrOK then}
    else qrPrios.Cancel;
```

```
  finally
    Form.Free;
  end;
end; {procedure Tfrm_projekte.btnInsertPriClick}
```

Je nachdem, ob ein neuer Datensatz eingefügt oder ein bestehender geändert werden soll, wird *Insert* oder *Edit* aufgerufen. Vor der Anzeige des Formulars müssen die beiden datensensitiven Edit-Felder die *DataSource* zugewiesen bekommen. Je nachdem, wie der Dialog geschlossen wird, wird nun *Post* oder *Cancel* aufgerufen und die entsprechenden Abfragen werden gegebenenfalls aktualisiert.

Die Anzeige aller Prioritäten

Um alle Prioritäten anzuzeigen, die in einem Projekt verfügbar sind, müssen die Prioritäten aus diesem Projekt und allen übergeordneten Projekten zusammengefasst werden. Dies delegieren wir wieder an eine STORED PROCEDURE:

```
CREATE PROCEDURE P_PRIORITAETEN
  (para_projekt INTEGER)
RETURNS
  (ID INTEGER,
  projekt INTEGER,
  sortier INTEGER,
  name CHAR(30))
AS
  DECLARE VARIABLE v_projekt INTEGER;
BEGIN
  IF (:para_projekt = 0) THEN
  BEGIN
    SUSPEND;
    EXIT;
  END
  v_projekt = para_projekt;
  WHILE (:v_projekt IS NOT NULL) DO
  BEGIN
    FOR SELECT
        id,
        projekt,
        sortier,
        name
      FROM T_prioritaet
      WHERE projekt = :v_projekt
```

```
      INTO
        :id,
        :projekt,
        :sortier,
        :name
    DO SUSPEND;
    SELECT
        master
      FROM t_projekt
      WHERE ID = :v_projekt
      INTO :v_projekt;
  END /* WHILE (v_projekt IS NOT NULL) DO */
END
```

In einer Schleife wird hier so lange der jeweilige Master des aktuellen Projektes ermittelt, bis man beim Wurzeleintrag angelangt ist und die Spalte *master* den Wert NULL hat. Zu jedem aktuellen Projekt werden dann die Prioritäten ermittelt.

Zur Anzeige sollen die Einträge nach der Projektnummer sortiert werden:

```
SELECT
    ID,
    name,
    projekt
  FROM p_prioritaeten(:ID)
  ORDER BY projekt, sortier DESC
```

Die Anzeigensteuerung der Formularteile

Je nachdem, welcher Benutzer angemeldet ist und welches Projekt gerade aktuell ist, sind Teile des Formulars sichtbar oder nicht. Schauen wir uns noch mal *qrProjektAfterScroll* an, diesmal vollständig:

```
procedure Tfrm_projekte.qrProjektAfterScroll(DataSet: TDataSet);
begin
  if CanSync then
  begin
    FProjektID := qrProjektID.AsInteger;
    TreeView1.Selected
      := GetNodeByID(TreeView1, FProjektID);
    if qrRecht.Active
      then qrRecht.First;
```

```
    with qrBenRechte do
    begin
      Close;
      ParamByName('benutzer').AsInteger := gvUserID;
      ParamByName('projekt').AsInteger := FProjektID;
      Open;
    end; {with qrBenRechte do}
    pnlFunktionen.Visible := qrBenRechteLESEN.AsString = 'J';
    RechteVergebenVisible(qrBenRechteLEGITIMIEREN.AsString = 'J');
    btnUnterprojekt.Visible := (not gvIsAdmin)
      and (qrBenRechteUNTERPROJEKTE.AsString = 'J');
  end; {if CanSync then}
end; {procedure Tfrm_projekte.qrProjektAfterScroll}
```

Nach dem Scrollen wird die Anzeige in Tree und die Datenmenge *qrBenRechte* aktualisiert. Auf *pnlFunktionen* liegen die beiden TabSheets. Damit diese sichtbar sind, sind Leserechte für das betreffende Projekt erforderlich.

Hat der angemeldete Benutzer das Recht, andere Benutzer zu legitimieren, dann werden die dafür erforderlichen Buttons eingeblendet. Das Recht zur Vergabe von Prioritäten haben nur Administratoren. Da dieses Recht nicht am Projekt hängt, wird die Routine *PriosVergebenVisible* nur beim Öffnen des Formulars aufgerufen.

Hat der angemeldete Benutzer Administratorfunktion, dann werden im DBNavigator auch die Buttons zum Einfügen, Ändern und Löschen von Datensätzen angezeigt. Fehlt dem Benutzer die Administratorfunktion, dann wird ihm der Button zum Anlegen von Unterprojekten angezeigt, vorausgesetzt, er hat im aktuellen Projekt das erforderliche Recht.

Der Passwortdialog

Beim Programmstart soll sich der Benutzer mit seinem Passwort anmelden.

Zu diesem Zweck gibt es in *TVerwaltung* die Klassenfunktion *PasswortDialog*, welche den dafür vorgesehenen Dialog aufruft und je nach Richtigkeit der Eingabe *true* oder *false* zurückgibt:

```
class function TVerwaltung.PasswortDialog: boolean;
var
  frm_passwort: Tfrm_passwort;
begin
  frm_passwort := Tfrm_passwort.Create(nil);
  with frm_passwort do
  begin
    try
      result := false;
      edBenutzer.Text := gvUserName;
      if ShowModal = mrOK
        then result
          := CheckPasswort(edBenutzer.Text, edPasswort.Text);
    finally
      Free;
    end;
  end; {with frm_passwort, frm_DataVerw do}
end; {class function TVerwaltung.PasswortDialog}
```

Die eigentliche Prüfung wird in der Routine *CheckPasswort* vorgenommen, welche eine Methode von *Tfrm_passwort* ist. Auf diesem Formular liegen zwei *TQuery*-Instanzen. *qrBenutzer* liefert eine Liste aller Benutzer nach ID sortiert, während *qrBenFunk* als dazugehörende Detail-Datenmenge die vom jeweiligen Benutzer wahrgenommenen Funktionen liefert.

```
SELECT
    m.benutzer,
    m.funktion
  FROM t_benutzer_funktion m
  WHERE m.benutzer = :ID
  ORDER BY m.funktion
```

Der Funktion *CheckPasswort* werden die eingegebenen Daten übergeben.

```
function Tfrm_passwort.CheckPasswort(UserName,
  Passwort: string): boolean;
begin
  result := false;
  qrBenutzer.Open;
  qrBenFunk.Open;
```

```
  if qrBenutzer.Locate('NAME', Variant(UserName), []) then
  begin
    if GetPasswort(qrBenutzerPASSWORT) = Passwort then
    begin
      gvUserID := qrBenutzerID.AsInteger;
      gvUserName := qrBenutzerNAME.AsString;
      qrBenFunk.First; // Ist nach Nummern sortiert
      gvIsAdmin := (qrBenFunkFUNKTION.AsInteger = 1);
      result := true;
    end
    else ShowMessage('Passwort falsch');
  end; {if frm_data.qrBenutzer.Locate...}
  qrBenutzer.Close;
  qrBenFunk.Close;
end; {function Tfrm_passwort.CheckPasswort}
```

Zunächst werden beide Datenmengen geöffnet, dann wird nach dem eingegebenen Benutzernamen gesucht, schlägt die Suche fehl, wird eine entsprechende Fehlermeldung ausgegeben. Im Erfolgsfall wird anschließend das Passwort geprüft – die dazugehörende Routine werden wir uns gleich noch ansehen. Stimmt auch das Passwort, wird das Funktionsergebnis auf *true* gesetzt. Des Weiteren werden die globalen Variablen *gvUserID*, *gvUserName* und *gvIsAdmin* gesetzt.

Passwortverschlüsselung

Die To-Do-Liste ist kein »Hochsicherheitssystem«, aber die Passwörter im Klartext speichern wollen wir trotzdem nicht. Als erste kleine Hürde werden die Passwörter in einem BLOB-Feld gespeichert, so dass man den Inhalt mit den InterBase-Tools nicht ansehen kann. (Schon der Datenbank-Explorer würde jedoch eine Betrachtung ermöglichen.)

Des Weiteren werden alle Passwörter mit einer XOR-Funktion verschlüsselt. Da für alle Passwörter derselbe Schlüssel verwendet wird, ist der Schutz limitiert, aber wie schon gesagt, ein Hochsicherheitssystem war damit nie beabsichtigt.

```
const
  C_KEY = 'Wi8&jfe,fs';   // 10 Stellen !
```

Im *implementation*-Teil der Unit findet sich als Schlüssel die Konstante C_KEY. Diese sollten Sie für andere Projekte abändern, achten Sie aber darauf, dass Sie eine Länge von mindestens zehn Stellen haben muss.

```
class function Tfrm_passwort.GetPasswort(Field: TBlobField): string;
var
  Stream: TMemoryStream;
```

```
  i: integer;
  a: array[0..9] of char;
begin
  if not Field.IsNull then
  begin
    Stream := TMemoryStream.Create;
    try
      Field.SaveToStream(Stream);
      Stream.Position := 0;
      Stream.Read(a, 10);
      for i := 0 to 9
        do a[i] := chr(byte(a[i]) xor byte(C_KEY[i + 1]));
      result := a;
    finally
      Stream.Free;
    end;
  end {if not Field.IsNull then}
  else result := '';
end; {class function Tfrm_passwort.GetPasswort}
```

Der Inhalt des Blob-Felder wird mittels eines Streams in ein Array eingelesen und dort einer XOR-Operation mit C_KEY unterzogen. Das Ergebnis wird dem Funktionsergebnis zugewiesen.

Mit der Prozedur *SetPasswort* wird ein neues Passwort in die Datenbanktabelle geschrieben:

```
class procedure Tfrm_passwort.SetPasswort(Field: TBlobField;
  passwort: string);
var
  Stream: TMemoryStream;
  i: integer;
  a: array[0..9] of char;
begin
  if passwort <> '' then
  begin
    Stream := TMemoryStream.Create;
    try
      StrPCopy(a, passwort);
      for i := 0 to 9
        do a[i] := chr(byte(a[i]) xor byte(C_KEY[i + 1]));
      Stream.Write(a, 10);
```

```
      Field.LoadFromStream(Stream);
    finally
      Stream.Free;
    end;
  end {if Value <> '' then}
  else Field.Clear;
end; {class procedure Tfrm_passwort.SetPasswort}
```

Hier wird zunächst der Parameter *passwort* in ein Array geschrieben, dann verschlüsselt und dann über einen Stream in eine Datenbank geschrieben.

Da die beiden Routinen nicht nur beim Passwortdialog, sondern auch beim Anlegen neuer Benutzer und Ändern des Passworts benötigt werden, sind sie hier als Klassenmethoden realisiert.

Verwendung des Passwortdialogs

Die Verwendung des Passwortdialogs ist ziemlich simpel:

```
procedure Tfrm_VerwMain.FormShow(Sender: TObject);
begin
  gvUserName := 'ADMIN';
  if not TVerwaltung.PasswortDialog
    then Application.Terminate;
end;
```

Wenn gewünscht, kann *gvUserName* entsprechend initialisiert werden. Während der Entwicklungsphase macht man dies sinnvollerweise mit *ADMIN*, später wollen wir den zuletzt angemeldeten Benutzer über eine Ini-Datei hier vorbelegen.

Mit der Methode *PasswortDialog* erfolgt dann die Anmeldung. Gibt diese Funktion *false* zurück, wird die Anwendung beendet.

18.3 Einträge erstellen und bearbeiten

Für das Erstellen und Bearbeiten von Einträgen beginnen wir ein neues Projekt (*p_testEin.dpr*) mit einem neuen Hauptformular (*Tfrm_EinMain* in der Unit *u_EinMain*). Das Datenmodul *frm_data* nehmen wir auch in dieses Projekt auf. Als Schnittstellen-Unit dient *u_eintrag*, darin wird *TEintrag* definiert:

```
type
  TEintrag = class(TObject)
  private
    procedure NeuClick(Sender: TObject);
```

```
  public
    class procedure SetMenuItem(Master: TMenuItem);
    class procedure Anzeigen(Eintrag: integer);
    class procedure Drucken(Eintrag: integer);
    class procedure DruckenDetail(Eintrag: integer);
  end;
```

Hier werden Klassenroutinen bereitgestellt, um einem Menü einen Menüpunkt zur Erstellung eines neuen Eintrags hinzuzufügen, um einen bestehenden Eintrag anzuzeigen oder ihn zu drucken.

Den prinzipiellen Aufbau von *SetMenuItem* kennen wir schon aus der Unit *u_verwaltung*, er soll hier nicht noch einmal besprochen werden.

Klickt der Anwender auf den dadurch neu eingefügten Menüpunkt, wird die Routine *NeuClick* ausgeführt.

```
procedure TEintrag.NeuClick(Sender: TObject);
begin
  frm_data.ModalAnzeigen(Tfrm_NeuEintrag, true);
end;
```

Dank der Methode *ModalAnzeigen* aus dem Datenmodul ist dies hier auch sehr übersichtlich.

18.3.1 Einen neuen Eintrag erstellen

Zum Erstellen eines neuen Eintrags wird das Formular *Tfrm_NeuEintrag* (Unit *u_NeuEintrag*) verwendet. Hier wird ein Projekt ausgewählt sowie eine Kurzbeschreibung und eine ausführliche Beschreibung eingegeben. Des Weiteren können die Funktion und die Priorität gesetzt sowie Dateien angehängt werden.

Hat der Benutzer für das ausgewählte Projekt nicht die Berechtigung, neue Einträge zu erstellen, dann wird der *OK*-Button gesperrt; fehlen ihm die Zuweisungsrechte, dann bleiben die ComboBoxen für Funktion und Priorität gesperrt.

Die Projekte werden wieder in einer Baumansicht dargestellt. Wie eine solche aufgebaut ist, wurde auf den letzten Seiten schon beschrieben. An *qrProjekt* hängt dann (über *dsProjekt*) die Query *qrPriosAll*, um die in diesem Projekt vergebbaren Prioritäten zu ermitteln.

Die Query *qrBenRechte* wird in der *AfterScroll*-Ereignisbehandlungsroutine von *qrProjekt* aktualisiert – auch dies hatten wir schon einmal bei der Projektverwaltung so gelöst. Mit *qrFunktion* wird eine Liste der Funktionen ermittelt – einer dieser Funktionen wird dann die Erledigung des Eintrags zugewiesen.

Die ComboBoxen

Mit der Liste der Funktionen und der Prioritäten werden die ComboBoxen *cmbFunktion* und *cmbPrios* gefüllt. Dabei wird die ID in einen Pointer umgewandelt und an den jeweiligen Eintrag angehängt.

```
procedure Tfrm_NeuEintrag.qrPriosAllAfterOpen(DataSet: TDataSet);
var
  i: integer;
begin
  i := Prioritaet;
  cmbPrios.Items.Clear;
  while not qrPriosAll.EOF do
  begin
    cmbPrios.Items.AddObject(qrPriosAllNAME.AsString,
      Pointer(qrPriosAllID.AsInteger));
    qrPriosAll.Next;
  end; {while not qrPriosAllEOF do}
  if i >= 0
    then Prioritaet := i;
end; {procedure Tfrm_NeuEintrag.qrPriosAllAfterOpen}
```

Mit der Funktion *GetPrioritaet* kann nun diese ID wieder ermittelt werden. Ist kein Eintrag ausgewählt, dann wird als Ergebnis -1 zurückgegeben. Dass kein Eintrag ausgewählt ist, kommt immer dann vor, wenn eine Priorität vergeben wurde und anschließend in ein Projekt gewechselt wird, in der diese nicht vorhanden ist.

```
function Tfrm_NeuEintrag.GetPrioritaet: integer;
begin
  if cmbPrios.ItemIndex >= 0
    then result
      := integer(cmbPrios.Items.Objects[cmbPrios.ItemIndex])
    else result := -1;
end;
```

Die Eigenschaft Priorität hat auch eine Schreibroutine, welche versucht, in der ComboBox den passenden Eintrag auszuwählen:

```
procedure Tfrm_NeuEintrag.SetPrioritaet(const Value: integer);
var
  i: integer;
begin
  cmbPrios.ItemIndex := -1;
  for i := 0 to cmbPrios.Items.Count - 1 do
  begin
    if integer(cmbPrios.Items.Objects[i]) = Value
      then cmbPrios.ItemIndex := i;
  end; {for i ...}
end; {procedure Tfrm_NeuEintrag.SetPrioritaet}
```

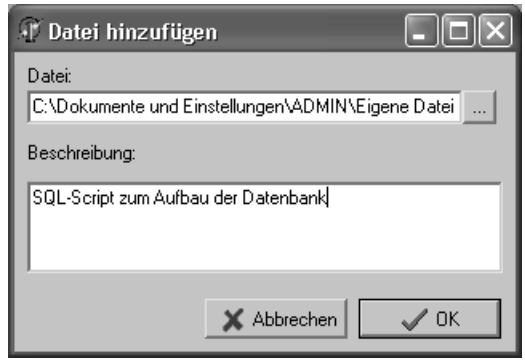
So wird auch klar, warum in *qrPriosAllAfterOpen* die ID der vergebenen Priorität in *i* zwischengespeichert wird: Nach dem erneuten Füllen der ComboBox wäre anderenfalls erst einmal keine Priorität ausgewählt.

Die prinzipielle Vorgehensweise zum Speichern der ID wird auch bei *cmbFunktion* verwendet.

Dateien hinzufügen

An jeden Eintrag können mehrere Dateien angehängt werden, die in der Datenbank gespeichert werden. Im Vergleich zur Ablage von Dateien in Verzeichnissen hat dies den Vorteil, dass in der Datenbank gespeicherte Dateien nicht einfach gelöscht werden können und so kein Problem mit Referenzen besteht, zu denen die Datei bereits gelöscht wurde.

Zum Anhängen von Dateien öffnen wir wieder ein neues Fenster:



Zunächst einmal kann hier der Dateiname der anzuhängenden Datei eingegeben oder mit einem entsprechenden Dialog ausgewählt weren. Des Weiteren kann hier eine Beschreibung ergänzt werden – der Anwender möchte schließlich vorher wissen, was er sich auf seine Festplatte herunterlädt.

Der Dateiauswahl-Dialog erhält zwei Methoden (im Moment, die später hinzukommende *FormClose*-Methode entfaltet hier noch keine Wirkung).

```
procedure Tfrm_AddFile.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute
    then edtFileName.Text := OpenDialog1.FileName;
end;

procedure Tfrm_AddFile.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if (ModalResult = mrOK)
    and not FileExists(edtFileName.Text) then
  begin
    ShowMessage('angegebene Datei existiert nicht!');
    CanClose := false;
  end;
end;
```

Wird auf den Button mit den drei Pünktchen geklickt, dann wird ein Dateiauswahl-Dialog geöffnet. Vor dem Schließen des Dialogs mit *OK* wird geprüft, ob es die angegebene Datei überhaupt gibt, anderenfalls wird eine Fehlermeldung ausgegeben.

```
procedure Tfrm_NeuEintrag.btnAddClick(Sender: TObject);
var
  Form: Tfrm_AddFile;
  sl: TStringList;
begin
  Form := Tfrm_AddFile.Create(nil);
  try
    if Form.ShowModal = mrOK then
    begin
      sl := TStringList.Create;
      sl.Assign(Form.Memo1.Lines);
      lbFiles.Items.AddObject(Form.edtFileName.Text, sl);
    end; {if Form.ShowModal then}
  finally
    Form.Free;
  end;
end; {procedure Tfrm_NeuEintrag.btnAddClick}
```

Wird in *Tfrm_NeuEintrag* der Button *Datei anhängen* betätigt, dann wird der Dateiauswahl-Dialog erzeugt und geöffnet und – sofern er mit *OK* geschlossen wurde – der Dateiname in die ListBox *lbFiles* geschrieben. Für die Bemerkung wird eine *TStringList*-Instanz erstellt, der Zeiger darauf wird an den Listbox-Eintrag angehängt.

```
procedure Tfrm_NeuEintrag.btnDeleteClick(Sender: TObject);
begin
  if lbFiles.ItemIndex = -1
    then exit;
  if MessageDlg('Soll die Datei "' + lbFiles.Items[lbFiles.ItemIndex]
      + '" entfernt werden?', mtConfirmation, [mbYes, mbNo], 0)
    = mrYes then
  begin
    lbFiles.Items.Objects[lbFiles.ItemIndex].Free;
    lbFiles.Items.Delete(lbFiles.ItemIndex);
  end;
end; {procedure Tfrm_NeuEintrag.btnDeleteClick}
```

Soll ein Eintrag aus der Liste der anzuhängenden Dateien gelöscht werden, dann wird nach einer Sicherheitsabfrage zuerst der angehängte Text und dann der Eintrag selbst gelöscht.

Vor dem Schließen des Formulars müssen dann noch alle angehängten Texte entfernt werden.

```
procedure Tfrm_NeuEintrag.FormDestroy(Sender: TObject);
var
  i: integer;
begin
  for i := 0 to lbFiles.Items.Count - 1
    do lbFiles.Items.Objects[i].Free;
end;
```

Speichern des Eintrags

Dank einiger lokaler Routinen ist der Hauptteil von *btnOKClick* recht übersichtlich:

```
procedure Tfrm_NeuEintrag.btnOKClick(Sender: TObject);
var
  iEin: integer;

  ...

begin
  ModalResult := mrAbort;
  IdHolen;
  DoEintrag;
  DoStatus;
  DoFiles;
  ModalResult := mrOK;
end; {procedure Tfrm_NeuEintrag.btnOKClick}
```

Zunächst wird *ModalResult* auf *mrAbort* gesetzt. Tritt bei der Abarbeitung dieser Prozedur eine Exception auf, dann wird die Transaktion mit *Rollback* zurückgenommen. Erst am Ende wird *ModalResult* auf *mrOK* gesetzt, somit kann die Transaktion mit *Commit* abgeschlossen werden.

Die anderen Anweisungen sind alle lokale Prozeduren, die wir uns nun anschauen wollen:

```
  procedure IdHolen;
  begin
    qrWork.SQL.Text := 'SELECT * FROM p_eintrag_ID';
    qrWork.Open;
    iEin := qrWork.FieldByName('GEN').AsInteger;
  end;
```

Schreiben wir einen neuen Datensatz in *t_eintrag*, dann würde ein Trigger dieser Tabelle vom Generator eine neue ID besorgen. Nun brauchen wir aber diese ID auch für die Einträge in *t_status* und *t_datei*.

Prinzipiell könnte man einen neuen Eintrag vornehmen, die Datenmenge aktualisieren, auf den letzten Datensatz positionieren und dort die ID auslesen. Im Mehrbenutzer-Betrieb kann diese Vorgehensweise jedoch auch schon mal danebengehen, solange *TransIsolation* von *Database1* den Wert *tiReadCommitted* hat.

Also fordern wir uns einen neuen Generatorwert an und speichern diesen in *iEin*. Die Komponente *qrWork* ist übrigens eine *TQuery*-Instanz, der die jeweilige SQL-Anweisung vor ihrer Ausführung zuzuweisen ist.

```
procedure DoEintrag;
begin
  with qrWork do
  begin
    SQL.Text := 'INSERT INTO t_eintrag';
    SQL.Add('VALUES (:ID, :Name, :Memo)');
    ParamByName('ID').AsInteger := iEin;
    ParamByName('Name').AsString := edtName.Text;
    ParamByName('Memo').Assign(Memo1.Lines);
    ExecSQL;
  end; {with qrWork do}
end; {procedure DoEintrag}
```

Mit *DoEintrag* werden dann die vorgesehenen Werte in die Tabelle *t_eintrag* geschrieben.

```
procedure DoStatus;
begin
  with qrWork do
  begin
    SQL.Text := 'INSERT INTO t_status (eintrag, benutzer,';
    SQL.Add('status, Funktion, Projekt, Prioritaet)');
    SQL.Add('VALUES (:eintrag, :benutzer, :status, :Funktion, ');
    SQL.Add(':Projekt, :Prioritaet)');
    ParamByName('eintrag').AsInteger := iEin;
    ParamByName('benutzer').AsInteger := gvUserID;
    if (cmbFunktion.ItemIndex = 0)
        and (cmbPrios.ItemIndex = 0)
      then ParamByName('status').AsString := 'E'
      else ParamByName('status').AsString := 'Z';
```

```
      ParamByName('Funktion').AsInteger := integer(cmbFunktion.
        Items.Objects[cmbFunktion.ItemIndex]);
      ParamByName('Projekt').AsInteger := FProjektID;
      ParamByName('Prioritaet').AsInteger
        := integer(cmbPrios.Items.Objects[cmbPrios.ItemIndex]);
      ExecSQL;
    end; {with qrWork do}
  end; {procedure DoStatus}
```

In die Tabelle *t_status* kommt gleich der erste Eintrag. Die ID wird per Generator erstellt, die Referenz auf den Eintrag und die UserID holen wir aber aus den entsprechenden Variablen. Sind sowohl Funktion wie auch Priorität nicht zugewiesen, dann erhält der Eintrag den Status *E*, anderenfalls den Status *Z*. Des Weiteren werden die Nummern für Funktion, Priorität und Projekt hinzugefügt.

```
function GetFileName(Text: string): string;
var
  s: string;
begin
  s := ExtractFileName(Text);
  if Length(s) > 30 then
  begin
    result := ExtractFileExt(s);
    result := copy(s, 1, 30 - Length(result)) + result;
  end
  else result := s;
end; {function GetFileName}

procedure DoFiles;
var
  i: integer;
  s: string;
begin
  for i := 0 to lbFiles.Items.Count - 1 do
  begin
    with qrFiles do
    begin
      s := lbFiles.Items[i];
      ParamByName('Eintrag').AsInteger := iEin;
      ParamByName('Name').AsString := GetFileName(s);
      ParamByName('Benutzer').AsInteger := gvUserID;
      ParamByName('Memo').Assign
        (lbFiles.Items.Objects[i] as TStrings);
```

```
        ParamByName('Datei').LoadFromFile(s, ftBlob);
        ExecSQL;
      end; {with qrFiles do}
    end; {for i := 0 to lbFiles.Items.Count - 1 do}
  end; {procedure DoFiles}
```

Die Routine *DoFiles* iteriert durch alle Einträge in *lbFiles*, fügt die entsprechenden Werte der Tabelle *t_datei* hinzu und lädt die zugewiesenen Dateien von der Festplatte in die Datenbank.

Der Dateiname wird dabei mit in die Datenbank gespeichert, insbesondere wegen der Extension. Das entsprechende Feld in der Datenbank hat eine Länge von 30 Zeichen, manche Dateinamen sind länger. Deshalb prüft die Funktion *GetFileName* die Länge und bildet gegebenenfalls eine Kombination aus dem Anfang des Dateinamens und der Extension.

Prüfen der Rechte

Von verschiedener Stelle aus wird die Prozedur *CheckRechte* aufgerufen.

```
procedure Tfrm_NeuEintrag.CheckRechte;
begin
  btnOK.Enabled := (cmbPrios.ItemIndex >= 0)
    and (qrBenRechteERSTELLEN.AsString = 'J');
  pnlFunktion.Enabled := qrBenRechteZUWEISEN.AsString = 'J';
end;
```

Der OK-Button wird freigegeben, wenn der Benutzer das Recht zum Erstellen von Einträgen im ausgewählten Projekt hat und wenn eine Priorität zugewiesen ist. (Hat der Benutzer nicht das Zuweisungsrecht, bleibt der erste Eintrag gewählt und *ItemIndex* hat somit den Wert Null.) Die beiden ComboBoxen liegen auf *pnlFunktion* und werden mit dem Zuweisungsrecht aktiviert.

18.3.2 Einen Eintrag ansehen und bearbeiten

Wir werden später Formulare erstellen, die in Tabellenform eine Reihe von Einträgen anzeigen, beispielsweise die Liste aller offenen Einträge oder alle Einträge, die noch zugewiesen werden müssen.

Aus diesen Listen heraus sollen einzelne Einträge angezeigt oder gedruckt werden. Während in der Liste nur einige Informationen zu sehen sind, werden in dem dann geöffneten Fenster alle Statusänderungen und alle angehängten Dateien zu sehen sein.

Für die Anzeige ist die *TEintrag*-Methode *Anzeigen* zuständig, der als Parameter die ID des anzuzeigenden Eintrags übergeben wird.

```
class procedure TEintrag.Anzeigen(Eintrag: integer);
var
  Form: Tfrm_EintragDetail;
begin
  Form := Tfrm_EintragDetail.Create(nil);
  with Form do
  begin
    try
      qrEintrag.ParamByName('ID').AsInteger := Eintrag;
      qrEintrag.Open;
      qrStatus.Open;
      qrDatei.Open;
      ShowModal;
    finally
      Free;
    end;
  end; {with Form do}
end; {class procedure TEintrag.Anzeigen}
```

In dieser Klassenmethode wird zunächst eine Instanz von *Tfrm_EintragDetail* erstellt. Bevor dieses Formular modal angezeigt wird, werden ein paar Querys geöffnet, *qrEintrag* wird davor der Eintrag als Parameter zugewiesen.

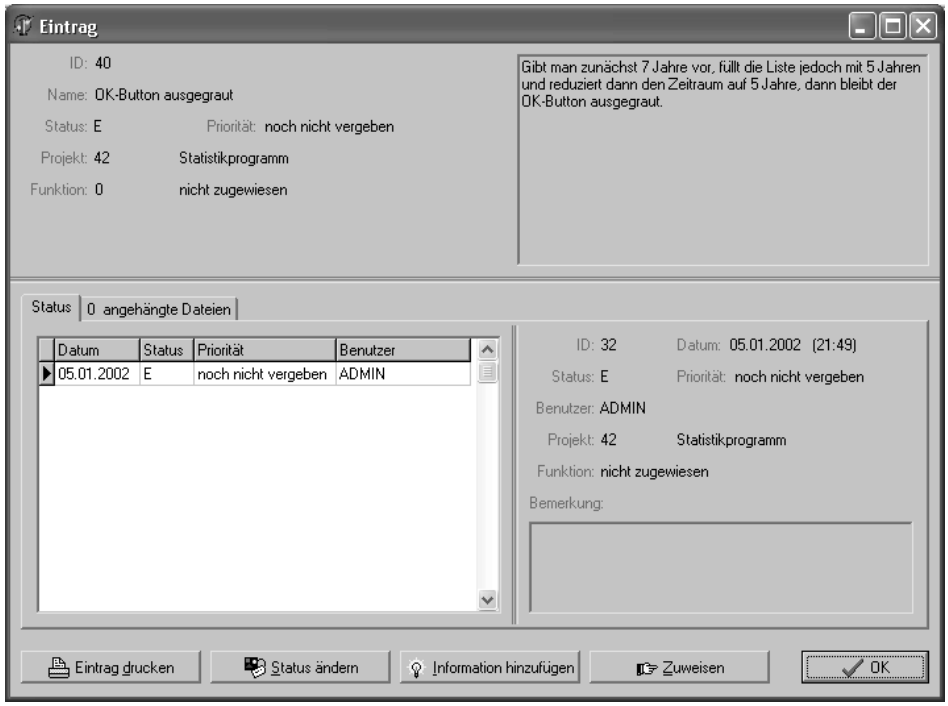
Das Formular hat wieder unseren »klassischen« Aufbau mit einem Panel unten, auf dem der OK-Button liegt (und diesmal ein wenig mehr), und zwei übereinander liegenden Panels, deren Größenverhältnis mittels eines Splitters verändert werden kann.

Oben findet man die wesentlichen Daten zum jeweiligen Eintrag, insbesondere den Kurz- und den Langtext, aber auch den aktuellen Status und die aktuelle Priorität, das Projekt und die Funktion, deren Benutzer sich um die Erledigung kümmern müssen.

Auf dem mittleren Panel finden sich zwei Registerseiten. Das eine davon zeigt die Datensätze in den Tabelle *t_status* an. Dabei kann es sich um Statusänderungen im eigentlichen Sinne handeln, also wenn beispielsweise die Bearbeitung aufgenommen wird, oder es handelt sich um zusätzliche Informationen oder um die Zuweisung eines Projektes oder an eine Funktion.

Auf dem anderen Panel findet man eine Liste der angehängten Dateien. Deren Anzahl wird im Register-Reiter vermerkt, so dass man nicht jedes Mal nachschauen muss, ob überhaupt etwas angehängt wurde.

Auf dem unteren Panel finden sich die Buttons zum Ausrucken, zur Statusänderung, zur Zuweisung an ein Projekt oder an eine Funktion und zum Hinzufügen von Dateien und Informationen.



Die Daten des Eintrags

Zu den Daten des Eintrags gehören zunächst einmal der Kurz- und der Langtext sowie die ID. Aus Gründen der besseren Lesbarkeit sind hier weitere Informationen hinzugefügt worden, beispielsweise der aktuelle Status und die aktuelle Priorität. Diese könnte man auch aus dem Grid auf der Registerseite *Status* ersehen, aber man kann sich ja auch mal auf der anderen Registerseite befinden.

Diese Daten werden von *qrEintrag* bereitgestellt, welche die folgende SQL-Anweisung verwendet:

```
SELECT
    e.ID,
    e.NAME,
    e.MEMO,
    e.DATUM_EINTRAG,
    e.DATUM_AENDER,
    e.STATUS,
    e.BENUTZER_AENDER,
    b.name,
```

```
    f.ID AS FUnktionsnummer,
    f.name AS Funktion,
    p.ID AS Projektnummer,
    p.name AS Projekt,
    i.name AS Prioritaet,
    i.ID AS Prioritaetnummer,
    i.sortier
  FROM p_eintragAktuell(:id) e
    INNER JOIN t_benutzer b
      ON e.benutzer_eintrag = b.ID
    LEFT OUTER JOIN t_funktion f
      ON e.funktion = f.ID
    INNER JOIN t_projekt p
      ON e.projekt = p.ID
    INNER JOIN t_prioritaet i
      ON e.prioritaet = i.ID
```

Die eigentlichen Informationen werden dabei von der STORED PROCEDURE *p_EintragAktuell* geliefert. Mit Hilfe von JOINS wird für Benutzer, Funktion, Projekt und Priorität der Name zur ID gestellt. Schauen wir uns die STORED PROCEDURE an:

```
CREATE PROCEDURE p_eintragaktuell
  (para_eintrag INTEGER)
RETURNS
  (id INTEGER,
  name VARCHAR(50),
  memo BLOB SUB_TYPE 1 SEGMENT SIZE 80,
  datum_eintrag TIMESTAMP,
  datum_aender TIMESTAMP,
  benutzer_eintrag INTEGER,
  benutzer_aender INTEGER,
  status VARCHAR(1),
  funktion INTEGER,
  projekt INTEGER,
  prioritaet INTEGER,
  bemerkung_aender BLOB SUB_TYPE 1 SEGMENT SIZE 80)
AS
  DECLARE VARIABLE v_min INTEGER;
  DECLARE VARIABLE v_max INTEGER;
```

Achten Sie darauf, dass die Blob-Parameter vom SUB_TYPE 1 sind, damit Delphi korrekt ein *TMemoField* anlegt.

Die Segmentgröße entspricht der der DOMAIN, ob das hier aber irgendeine Wirkung hat, habe ich noch nicht herausgefunden.

```
BEGIN
  SELECT
      id, name, memo
    FROM t_eintrag
    WHERE ID = :para_eintrag
    INTO :id, :name, :memo;
```

Zunächst werden die Daten aus *t_eintrag* geholt.

```
  SELECT
      MIN(id), MAX(id)
    FROM t_status
    WHERE EINTRAG = :para_eintrag
    INTO :v_min, :v_max;
```

Anschließend werden die IDs des ersten und des letzten Eintrags erfasst. Da die IDs über einen Generator vergeben werden, kann man hier eine chronologische Reihenfolge unterstellen. Die Ergebnisse werden in die Variablen *v_min* und *v_max* geschrieben.

```
  SELECT
      datum, benutzer
    FROM t_status
    WHERE ID = :v_min
    INTO :datum_eintrag, :benutzer_eintrag;
```

Alsdann wird das Datum der Erstellung des Eintrags und der eintragende Benutzer ermittelt. Diese Daten befinden sich im ersten Eintrag von *t_status*, somit wird mit *v_min* gefiltert.

```
  SELECT
      datum, benutzer, funktion, projekt, prioritaet, bemerkung
    FROM t_status
    WHERE ID = :v_max
    INTO :DATUM_aender, :benutzer_aender,
      :funktion, :projekt, :prioritaet, :bemerkung_aender;
```

Nun wird das Datum der letzten Änderung in *t_status* ermittelt (das muss keine Statusänderung sein, sondern kann auch eine hinzugefügte Information betreffen) und der dazugehörende Benutzer, weiterhin die Funktion, das Projekt und die Priorität.

```
  SELECT
      MAX(ID)
    FROM T_STATUS
    WHERE EINTRAG = :para_eintrag
      AND status IN ('E', 'A', 'B', 'U', 'F')
    INTO :v_max;
  SELECT
      status
    FROM t_status
    WHERE ID = :v_max
    INTO :status;
  SUSPEND;
END
```

Zuletzt wird noch die ID der letzten tatsächlichen Statusänderung ermittelt und mit dieser der aktuelle Status abgefragt. Mit SUSPEND werden dann die Daten übertragen.

Der Status

Als Detail-Datenmenge von *qrEintrag* fungiert *qrStatus*. Hier werden alle Einträge der Tabelle *t_status* ermittelt, die zum angezeigten Eintrag gehören, ergänzt um die Klartextnamen von Benutzer, Projekt, Priorität und Funktion.

```
SELECT
    s.ID,
    s.Eintrag,
    s.Datum,
    s.Datum AS DatumEinzeln,
    b.name AS Benutzer,
    s.Status,
    f.name AS Funktion,
    s.Bemerkung,
    s.Projekt AS Projektnummer,
    p.name AS Projekt,
    i.name AS Prioritaet
  FROM t_status s
    INNER JOIN t_benutzer b
      ON s.benutzer = b.ID
    INNER JOIN t_projekt p
      ON s.projekt = p.ID
    INNER JOIN t_prioritaet i
      ON s.prioritaet = i.ID
```

```
    LEFT OUTER JOIN t_funktion f
      ON s.funktion = f.ID
  WHERE s.eintrag = :ID
  ORDER BY s.DATUM
```

Im Detailbereich soll das Datum vollständig angezeigt werden, im Grid dagegen ohne Uhrzeit. Zu diesem Zweck wird ein berechnetes Feld angelegt (*qrStatusDatumEinzeln*), bei dem über die Eigenschaft *DisplayFormat* die Anzeige der Uhrzeit unterbleibt.

```
procedure Tfrm_EintragDetail.qrStatusCalcFields(DataSet: TDataSet);
begin
  qrStatusDatumEinzeln.AsDateTime := qrStatusDATUM.AsDateTime;
end;
```

Da es sich um ein berechnetes Feld handelt, muss auf das Ereignis *OnCalcFields* reagiert werden. Anstelle der Lösung mit dem berechneten Feld könnten wir auch das betreffende Feld ein zweites Mal mit der SQL-Anweisung abfragen – diese Variante werden wir beim Druck einsetzen.

Den Status ändern

Wird auch der Button *Status ändern* betätigt, dann erscheint ein Dialog, mit dessen Hilfe man zwischen den einzelnen Bearbeiter-Stati (*akzeptieren*, *in Bearbeitung*, *Bearbeitung unterbrochen* und *fertig*) wählen und eine Bemerkung ergänzen kann.

Die Stati *Zuweisen* und *Information* werden automatisch gesetzt, wenn der Eintrag zugewiesen oder eine Information hinzugefügt wird.

```
function Tfrm_EditStatus.GetStatus: Char;
begin
  case cmbStatus.ItemIndex of
    0: result := 'A';
    1: result := 'B';
    2: result := 'U';
    3: result := 'F';
  end;
end;
```

Der Dialog enthält nur eine eigene Methode, welche je nach gewähltem Eintrag in der ComboBox den dazugehörenden Buchstaben zurückgibt.

Beim Anzeigen des Formulars *Tfrm_EintragDetail* wird keine Transaktion geöffnet, schließlich geht es in erster Linie um das Anzeigen der Einträge. Wird der Status geändert, muss erst eine Transaktion gestartet werden.

```
procedure Tfrm_EintragDetail.btnStatusClick(Sender: TObject);
var
  Form: Tfrm_EditStatus;

  ...

begin
  Form := Tfrm_EditStatus.Create(nil);
  try
    if Form.ShowModal = mrOK then
    begin
      frm_data.BeginTransaction;
      try
        WerteEintragen;
        frm_data.Database1.Commit;
      except
        frm_data.Database1.Rollback;
      end;
    end; {if Form.ShowModal = mrOK then}
  finally
    Form.Free;
  end;
  RefreshQuery(qrEintrag);
  RefreshQuery(qrStatus);
end; {procedure Tfrm_EintragDetail.btnStatusClick}
```

Von der Query *qrEintrag* wird auch der aktuelle Status ermittelt, deshalb muss hier ebenfalls diese Abfrage aktualisiert werden.

Die eigentliche Arbeit übernimmt dann die lokale Prozedur *WerteEintragen*:

```
procedure WerteEintragen;
begin
  with qrAddStatus do
  begin
    ParamByName('Eintrag').AsInteger
      := qrEintragID.AsInteger;
    ParamByName('Benutzer').AsInteger := gvUserID;
    ParamByName('Status').AsString := Form.GetStatus;
    ParamByName('Funktion').AsInteger
      := qrEintragFUNKTIONSNUMMER.AsInteger;
    ParamByName('Bemerkung').Assign(Form.Memo1.Lines);
    ParamByName('Projekt').AsInteger
      := qrEintragPROJEKTNUMMER.AsInteger;
    ParamByName('Prioritaet').AsInteger
      := qrEintragPRIORITAETNUMMER.AsInteger;
    ExecSQL;
  end; {with qrAddStatus do}
end; {procedure WerteEintragen}
```

Die Query *qrAddStatus* dient dem Eintragen neuer Datensätze in *t_status* und verwendet folgende SQL-Anweisung:

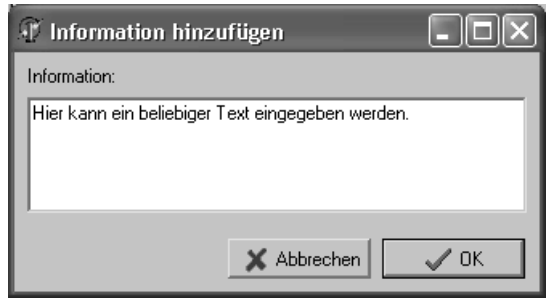
```
INSERT INTO t_status (Eintrag, Benutzer, Status,
    Funktion, Bemerkung, Projekt, Prioritaet)
  VALUES (:Eintrag, :Benutzer, :Status,
    :Funktion, :Bemerkung, :Projekt, :Prioritaet)
```

Informationen hinzufügen

Auch zum Hinzufügen von Informationen gibt es ein eigenes Formular; in einem Memo kann die Information eingegeben werden, ansonsten findet man nur noch *btnOK* und *btnAbort*.

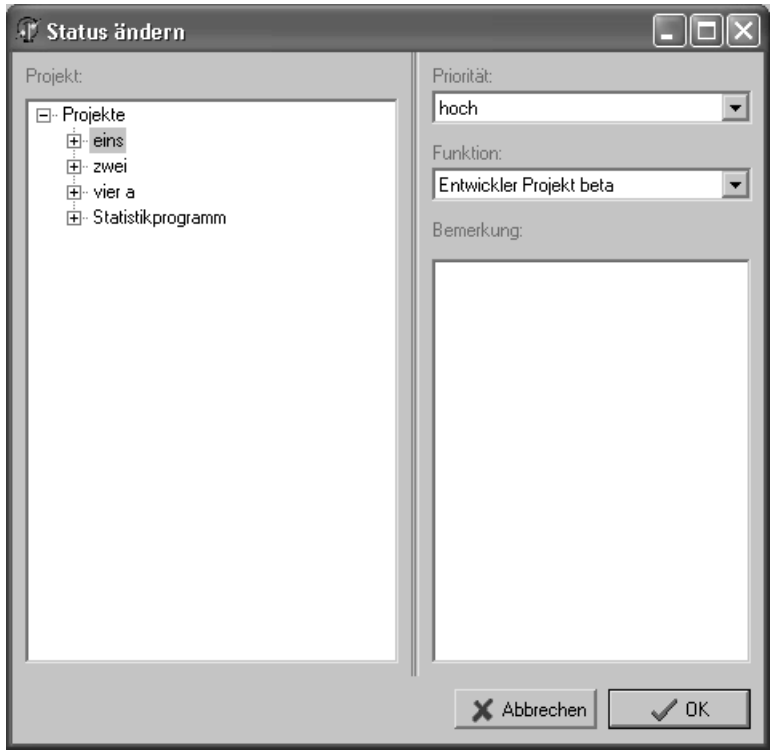
Die Routine zum Anzeigen dieses Dialogs und zum Einfügen der Daten gleicht fast völlig *btnStatusClick* und soll hier nicht abgedruckt werden. Die beiden Abweichungen von *btnStatusClick*:

- Als Status wird immer *I* geschrieben.
- *qrEintrag* muss nicht aktualisiert werden.



Zuweisen

Soll ein Eintrag einer anderen Funktion zur Bearbeitung zugewiesen werden oder soll die Priorität geändert werden, dann wird *frm_Zuweis* geöffnet.



Schauen wir uns zunächst den Aufruf dieses Formulars an:

```
procedure Tfrm_EintragDetail.btnZuweisClick(Sender: TObject);
var
  Form: Tfrm_Zuweis;

  ...

begin
  Form := Tfrm_Zuweis.Create(nil);
  try
    Form.Projekt := qrEintragPROJEKTNUMMER.AsInteger;
    Form.Prioritaet := qrEintragPRIORITAETNUMMER.AsInteger;
    Form.Funktion := qrEintragFUNKTIONSNUMMER.AsInteger;
    if Form.ShowModal = mrOK then
    begin
      frm_data.BeginTransaction;
      try
        WerteEintragen;
        frm_data.Database1.Commit;
      except
        frm_data.Database1.Rollback;
      end;
    end; {if Form.ShowModal = mrOK then}
  finally
    Form.Free;
  end;
  RefreshQuery(qrEintrag);
  RefreshQuery(qrStatus);
end; {procedure Tfrm_EintragDetail.btnZuweisClick}
```

Die lokale Routine *WerteEintragen* gleicht weitgehend der von *btnStatusClick*, lediglich als Status wird *Z* geschrieben. Vor dem Anzeigen des Formulars werden jedoch das aktuelle Projekt, die Priorität und die Funktion in die gleichnamigen Eigenschaften von *Tfrm_Zuweis* geschrieben.

```
type
  Tfrm_Zuweis = class(TForm)
    ...
  private
    CanSync: boolean;
    procedure RefreshTree(Tree: TTreeView);
    function GetNodeByID(Tree: TTreeView; ID: integer): TTreeNode;
    procedure SetFunktion(const Value: integer);
```

```
    function GetFunktion: integer;
    procedure SetProjekt(const Value: integer);
    procedure SetPrioritaet(const Value: integer);
    function GetPrioritaet: integer;
    procedure CheckOkBtn;
  public
    { Public-Deklarationen }
    FProjektID: integer;
    property Funktion: integer read GetFunktion write SetFunktion;
    property Projekt: integer read FProjektID write SetProjekt;
    property Prioritaet: integer
      read GetPrioritaet write SetPrioritaet;
  end;
```

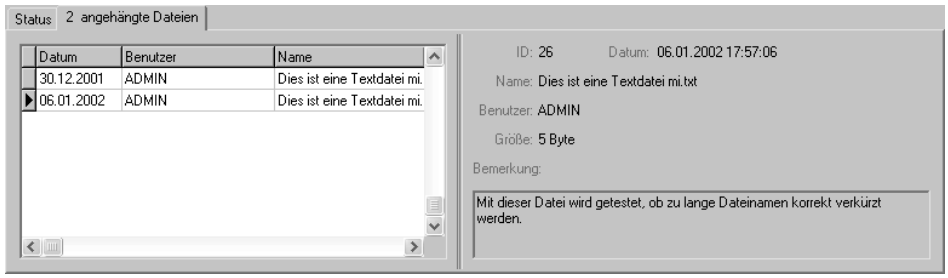
Neben »alten Bekannten« wie *RefreshTree* und *GetNodeByID* finden wir vor allem die Lese- und Schreiberoutinen dieser Eigenschaften. Bei den ComboBoxen und beim Tree wird wieder die ID als »Zeiger« angehängt, *GetPrioritaet* und *GetFunktion* lesen lediglich diesen Wert aus, während die Schreibroutinen alle Einträge durchsuchen, ob der gesuchte Wert denn vorkommt. Die Vorgehensweise gleicht der von *Tfrm_NeuEintrag*.

Auch werden wieder die Rechte ermittelt und dementsprechend wird der OK-Button gesteuert:

```
procedure Tfrm_Zuweis.CheckOkBtn;
begin
  btnOK.Enabled := (qrBenRechteZUWEISEN.AsString = 'J')
    and (cmbPrios.ItemIndex >= 0);
end;
```

Anzeige der angehängten Dateien

Auf der Registerseite *tabDateien* findet man eine Liste der Dateien, die dem aktuellen Eintrag angehängt sind.



Zu diesem Zweck erfolgt über *qrDatei* ein Zugriff auf die Tabelle *t_datei*, über einen JOIN wird der Benutzername hinzugefügt:

```
SELECT
    d.ID,
    d.eintrag,
    d.name,
    d.datum,
    d.datum AS DatumEinzeln,
    b.name,
    d.memo,
    d.datei
  FROM t_datei d
    INNER JOIN t_benutzer b
      ON d.benutzer = b.id
  WHERE eintrag = :ID
  ORDER BY DATUM
```

Auf dem Register-Reiter wird dabei die Anzahl der angehängten Dateien angezeigt:

```
procedure Tfrm_EintragDetail.qrDateiAfterOpen(DataSet: TDataSet);
begin
  if qrDatei.RecordCount = 1
    then tabDateien.Caption := '1  angehängte Datei'
    else tabDateien.Caption := IntToStr(qrDatei.RecordCount)
      + '  angehängte Dateien';
end;
```

Des Weiteren wird zu jeder angehängten Datei die Größe angezeigt:

```
procedure Tfrm_EintragDetail.qrDateiAfterScroll(DataSet: TDataSet);
var
  i: integer;
begin
  i := qrDateiDATEI.BlobSize;
  case i of
    0..1024: edtSize.Text := IntToStr(i) + ' Byte';
    1025..1048576: edtSize.Text
      := FormatFloat('0.00', i / 1024) + ' kB';
    else edtSize.Text := FormatFloat('0.00', i / 1048576) + ' MB';
  end;
end; {procedure Tfrm_EintragDetail.qrDateiAfterScroll}
```

Dateien hinzufügen

Den Dialog zum Hinzufügen von Dateien haben wir uns bereits angesehen, den Eintrag in die Datenbank hat dabei die aufrufende Routine übernommen, schließlich sollte dieser nur dann erfolgen, wenn der betreffende Dialog mit *OK* geschlossen wurde. Hier wird nun anders vorgegangen:

```
procedure Tfrm_EintragDetail.btnDateiClick(Sender: TObject);
begin
  frm_data.Modalanzeigen(Tfrm_AddFile, true, qrEintragID.AsInteger);
  RefreshQuery(qrDatei);
  qrDatei.Last;
end;
```

Der Routine *ModalAnzeigen* wird die ID des Eintrags übergeben, welche diese Prozedur dann in die Eigenschaft *Tag* des angezeigten Formulars schreibt. Nach dem Hinzufügen wird *qrDatei* aktualisiert und auf den letzten Datensatz gescrollt – da nach dem Datum sortiert wird, sollte dies der soeben eingefügte sein.

Das Eintragen in die Datenbank übernimmt die Routine *FormClose*:

```
procedure Tfrm_AddFile.FormClose(Sender: TObject;
  var Action: TCloseAction);

  function GetFileName: string;
  var
    s: string;
  begin
    s := ExtractFileName(edtFileName.Text);
    if Length(s) > 30 then
    begin
      result := ExtractFileExt(s);
      result := copy(s, 1, 30 - Length(result)) + result;
    end
    else result := s;
  end; {function GetFileName}

begin
  if (ModalResult = mrOK) and (Tag > 0) then
  begin
    with Query1 do
    begin
      ParamByName('Eintrag').AsInteger := Self.Tag;
```

```
        ParamByName('Name').AsString := GetFileName;
        ParamByName('Benutzer').AsInteger := gvUserID;
        ParamByName('Memo').Assign(Memo1.Lines);
        ParamByName('Datei').LoadFromFile(edtFileName.Text, ftBlob);
        ExecSQL;
      end; {with Query1 do}
    end; {if (ModalResult = mrOK)...}
end; {procedure Tfrm_AddFile.FormClose}
```

Erwartungsgemäß wird die Datei nur dann in die Datenbank geschrieben, wenn der Dialog mit *OK* geschlossen wird. Des Weiteren wird geprüft, ob eine ID des dazugehörenden Eintrags in die Eigenschaft *Tag* geschrieben wurde – dies ist nicht der Fall, wenn dieser Dialog von *Tfrm_NeuEintrag* aus aufgerufen wird.

Die Komponente *Query1* enthält ein »klassisches« Insert-Statement, die Werte werden über Parameter »reingereicht«. Auf hier muss wieder dafür gesorgt werden, dass Dateinamen, die länger als 30 Zeichen lang sind, entsprechend gekürzt werden.

Tritt bei der Abarbeitung dieser Prozedur eine Exception auf, dann macht *Modal-Anzeigen* die Transaktion rückgängig.

Eine Datei lokal speichern

Es macht natürlich nur dann Sinn, Dateien in eine Datenbank zu speichern, wenn man sie da auch wieder herausbekommt. Zu diesem Zweck gibt es den Button *Datei herunterladen*. Eigentlich müsste man nur das entsprechende Feld mit *SaveToFile* abspeichern, »anstandshalber« sollte man eine Sicherheitsabfrage vor dem Überschreiben vorhandener Dateien einfügen.

```
procedure Tfrm_EintragDetail.btnDownClick(Sender: TObject);
var
  s: string;

  procedure Speichern(FileName: string);
  begin
    qrDateiDATEI.SaveToFile(FileName);
  end;

begin
  SaveDialog1.FileName := qrDateiNAME.AsString;
  if SaveDialog1.Execute then
  begin
    s := SaveDialog1.FileName;
```

```
    if FileExists(s) then
    begin
      if MessageDlg('Die Datei "' + s + '" existiert bereits.'
          + #13#10 + 'Überschreiben?', mtConfirmation,
          [mbYes, mbNo], 0) = mrYes
        then Speichern(s);
    end {if FileExists(s) then}
    else Speichern(s);
  end; {if SaveDialog1.Execute then}
end; {procedure Tfrm_EintragDetail.btnDownClick}
```

Sonstiges

Je nachdem, welche Registerseite ausgewählt ist, werden andere Buttons angezeigt:

```
procedure Tfrm_EintragDetail.PageControl1Change(Sender: TObject);
begin
  btnDatei.Visible := PageControl1.ActivePageIndex = 1;
  btnDown.Visible := PageControl1.ActivePageIndex = 1;
  btnStatus.Visible := PageControl1.ActivePageIndex = 0;
  btnInfo.Visible := PageControl1.ActivePageIndex = 0;
  btnZuweis.Visible := PageControl1.ActivePageIndex = 0;
end;
```

Für das betreffende Projekt werden die Benutzerrechte ermittelt. Dafür verwendet *qrRechte* die bereits beschriebene STORED PROCEDURE *p_checkrechte*.

```
procedure Tfrm_EintragDetail.qrEintragAfterOpen(DataSet: TDataSet);
begin
  with qrRechte do
  begin
    Close;
    ParamByName('Projekt').AsInteger
      := qrEintragPROJEKTNUMMER.AsInteger;
    ParamByName('Benutzer').AsInteger := gvUserID;
    Open;
  end; {with qrRechte do}
  btnDatei.Enabled := (qrRechteERSTELLEN.AsString = 'J')
    or (qrRechteBEARBEITEN.AsString = 'J')
    or (qrRechteINFORMIEREN.AsString = 'J');
  btnDown.Enabled := (qrRechteLESEN.AsString = 'J');
  btnStatus.Enabled := (qrRechteBEARBEITEN.AsString = 'J');
```

```
  btnInfo.Enabled := (qrRechteINFORMIEREN.AsString = 'J');
  btnZuweis.Enabled := (qrRechteZUWEISEN.AsString = 'J');
end; {procedure Tfrm_EintragDetail.qrEintragAfterOpen}
```

Um den Status ändern zu können, benötigt man Bearbeiterrechte, für Infos und Zuweisung gibt es eigene Rechte. Um Dateien anhängen zu können, braucht man Erstellungs-, Bearbeitungs- oder Informationsrechte, um sie herunterladen zu können, reicht das Leserecht.

```
procedure Tfrm_EintragDetail.btnPrintClick(Sender: TObject);
begin
  TEintrag.DruckenDetail(qrEintragID.AsInteger);
end;
```

Zum Drucken braucht lediglich die entsprechende Routine aufgerufen zu werden.

18.3.3 Einträge drucken

Die Klasse *TEintrag* implementiert zwei Methoden zum Drucken eines Eintrags: *Drucken* für den Druck einer Zusammenfassung und *DruckenDetail* für den Druck des Eintrags mit allen Statusänderungen und mit einer Liste der angehängten Dateien.

Zusammenfassung drucken

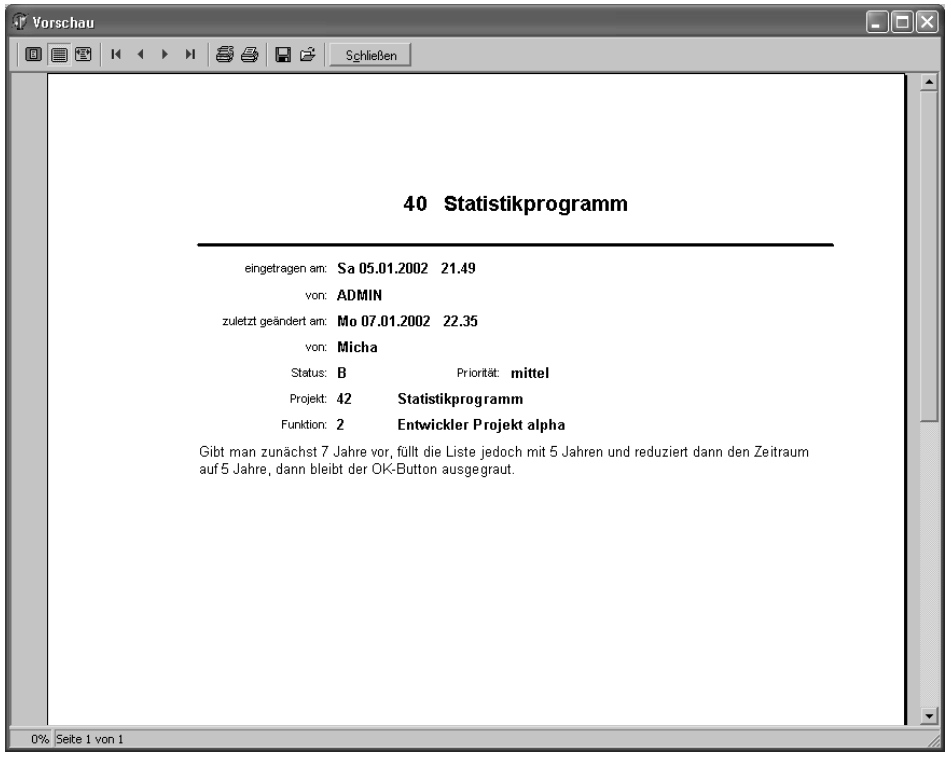
Für die Zusammenfassung wird die Unit *u_EintragPrint* verwendet. Es wird hier lediglich eine Datenzugriffskomponente benötigt, die auf die STORED PROCEDURE *p_EintragAktuell* zugreift.

```
SELECT
    e.id || "   " || p.name AS name,
    e.memo,
    e.datum_eintrag,
    e.datum_aender,
    be.name AS benutzer_eintrag,
    ba.name AS benutzer_aender,
    e.status,
    f.id AS funktionsnummer,
    f.name as funktionsname,
    p.id AS projektnummer,
    p.name AS projektname,
    i.name AS prioritaet
  FROM p_EintragAktuell(:ID) e
```

```
INNER JOIN t_benutzer be
  ON e.benutzer_eintrag = be.id
INNER JOIN t_benutzer ba
  ON e.benutzer_aender = ba.id
INNER JOIN t_funktion f
  ON e.funktion = f.id
INNER JOIN t_projekt p
  ON e.projekt = p.id
INNER JOIN t_prioritaet i
  ON e.prioritaet = i.id
```

Die Klartextnamen von Benutzer, Funktion, Projekt und Priorität werden wieder per JOIN hinzugefügt. Da hier zwei verschiedene Benutzernamen benötigt werden – derjenige des Eintrags und derjenige der letzten Statusänderung – wird hier zweimal ein JOIN auf die Tabelle *t_benutzer* durchgeführt.

Eintragsnummer und -name werden zu einem Feld zusammengefasst, damit man sie gemeinsam zentrieren kann.



Die Datumsangaben werden mit Wochentag, die Uhrzeit aber ohne Sekunden dargestellt. Den beiden Spalten DATUM_EINTRAG und DATUM_AENDER ist als *DisplayFormat* der Wert *ddd dd.mm.yyyy hh.mm* zuzuweisen.

Beim Report werden zunächst einmal »bürotaugliche« Randeinstellungen getroffen, also beispielsweise links 35 mm (zum Abheften), oben 25 mm, rechts und unten 20 mm.

Der Report umfasst nur ein Band, wobei es egal ist, ob dieses vom Typ *rbTitle* oder *rbDetail* ist, da ohnehin nur ein Datensatz angezeigt wird. Mit *QRLabel*- und *QRDBText*-Komponenten werden die Daten und deren Beschriftung angezeigt. Die Komponente *qrDBMemo* zeigt dabei den Langtext des Eintrags an. Hier wird die Eigenschaft *AutoSize* auf *false* sowie *AutoStretch* und *WorfWrap* auf *true* gesetzt, damit der Text mehrzeilig angezeigt werden kann und dafür auch die Bandhöhe vergrößert wird.

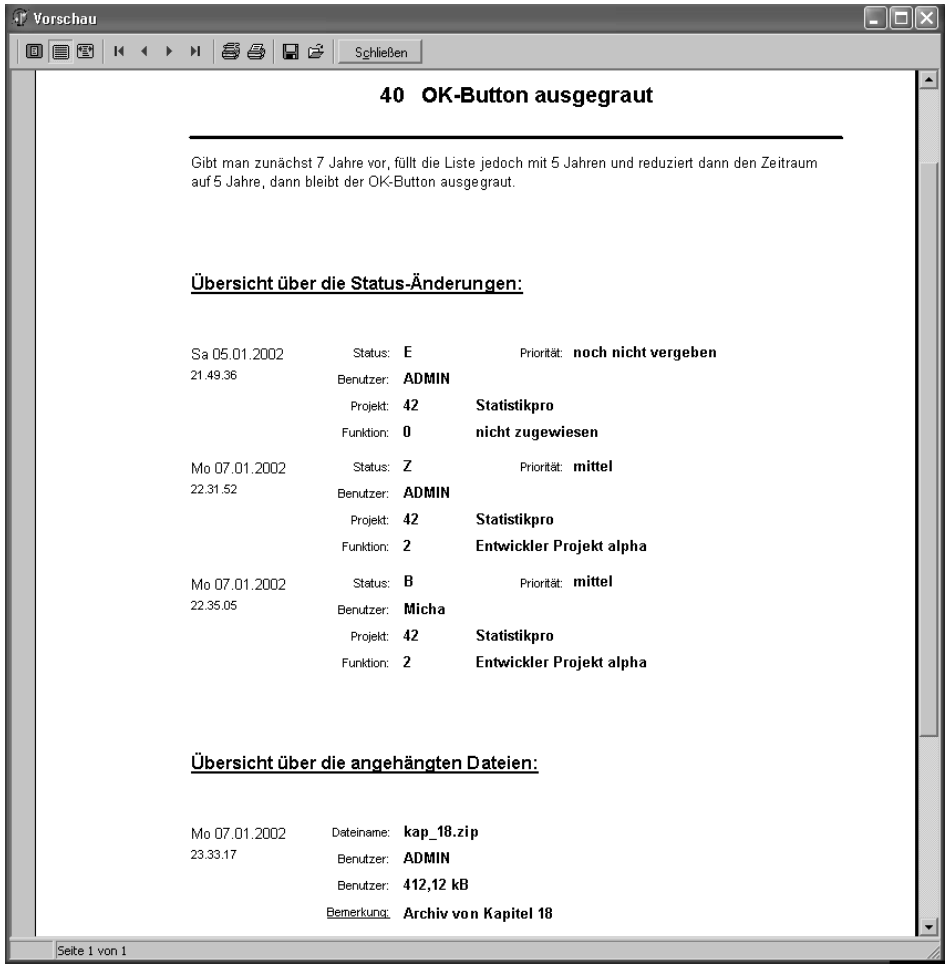
Mit der folgenden Routine wird dieser Report zur Voransicht geöffnet:

```
class procedure TEintrag.Drucken(Eintrag: integer);
var
  Form: Tfrm_EintragPrint;
begin
  Form := Tfrm_EintragPrint.Create(nil);
  with Form do
  begin
    try
      qrEintrag.Close;
      qrEintrag.ParamByName('ID').AsInteger := Eintrag;
      qrEintrag.Open;
      QuickRep1.Preview;
    finally
      Form.Free;
    end;
  end; {with Form do}
end; {class procedure TEintrag.Drucken}
```

Im Prinzip das gewohnte Vorgehen: Formular erzeugen, Parameter setzen, Datenmenge öffnen und mit *QuickRep1.Preview* die Voransicht aufrufen. Etwas verwundern mag die Anweisung *qrEintrag.Close*, wird doch das Formular direkt davor erzeugt. Wurde jedoch beim Entwickeln zur Entwurfszeit die *qrEintrag*-Eigenschaft *Active* auf *true* belassen, dann würde schon beim Create die Abfrage mit dem Parameter geöffnet, der zur Entwurfszeit im Objektinspektor eingestellt wurde. Die neuerliche Zuweisung des Parameters hätte dann keinen Einfluss mehr. Mit *qrEintrag.Close* kann man dieses Problem vermeiden.

Detailreport drucken

Mit dem Report auf *frm_EintragDetailPrint* (Unit *u_EintragDetailPrint*) wird ein Eintrag mitsamt allen Statusänderungen und der Liste der angehängten Dateien gedruckt.



Auch hier werden zunächst die Randeinstellungen geändert, da hier aber eine Fußzeile verwendet werden soll, werden wir 10 mm als unteren Rand verwenden.

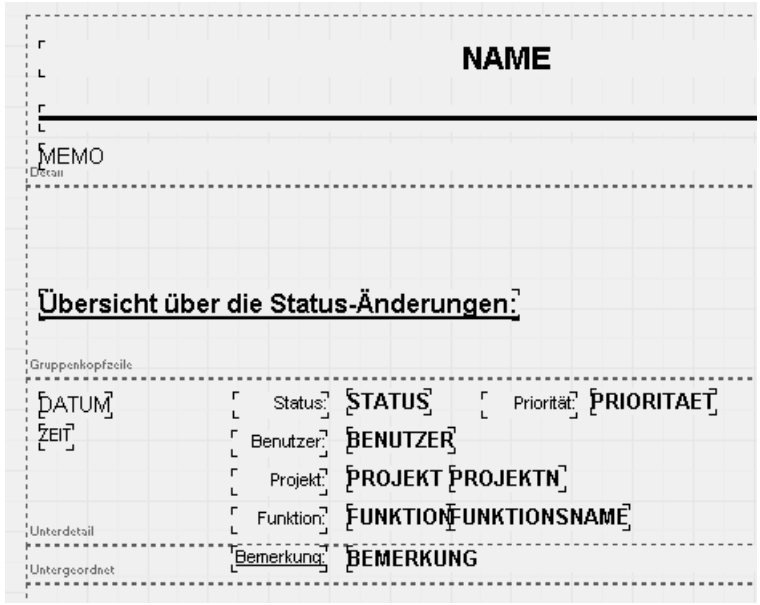
Die Hauptdatenmenge dieses Reports wird *qrEintrag*, die *QuickRep1*-Eigenschaft *DataSet* ist entsprechend zu setzen. Die Query *qrEintrag* greift direkt auf *t_eintrag* zu, für die Titelzeile werden wieder ID und Name verbunden:

```
SELECT
     ID,
     ID || "   " || Name AS Name,
     MEMO
  FROM t_eintrag
  WHERE ID = :ID
```

Die Daten werden dann wieder mittels *QRDBText*-Komponenten auf einem Detailband angezeigt.

Anzeige der Statusänderungen

Für die Anzeige der Statusänderungen verwenden wir drei Bänder:



Die Daten werden dabei auf dem *TQRSubDetail*band *QRSubStatus* angezeigt. Als verbundene Datenmenge dient hier *qrStatus*, als Master *QuickRep1*. Auf diesem Band liegen die Komponenten zur Darstellung der Daten mit Ausnahme der Bemerkungen.

Durch Setzen der *QRSubStatus*-Eigenschaft *HasChild* auf *true* wird ein Band vom Typ *TQRChildBand* hinzugefügt, wo nun die Komponenten zur Darstellung der Bemerkung liegen. Warum diese auf einem eigenen Band liegen, klären wir später.

Nun wird noch ein weiteres Band eingefügt, dem wir dem Namen *QRStatusHeader* und den *BandType rbGroupHeader* geben. Auf dieses Band kommt die Überschrift *Übersicht über die Status-Änderungen*. Die *QRSubStatus*-Eigenschaft *HeaderBand* setzen wir auf *QRStatusHeader*.

Als SQL-Anweisung für *qrStatus* verwenden wir den folgenden JOIN:

```
SELECT
    s.ID,
    s.Eintrag,
    s.Datum,
    s.Datum AS Zeit,
    b.name AS Benutzer,
    s.Status,
    f.ID  AS Funktion,
    f.name AS Funktionsname,
    s.Bemerkung,
    s.Projekt AS Projekt,
    p.name AS Projektname,
    i.name AS Prioritaet
  FROM t_status s
    INNER JOIN t_benutzer b
      ON s.benutzer = b.ID
    INNER JOIN t_projekt p
      ON s.projekt = p.ID
    INNER JOIN t_prioritaet i
      ON s.prioritaet = i.ID
    LEFT OUTER JOIN t_funktion f
      ON s.funktion = f.ID
  WHERE s.eintrag = :ID
  ORDER BY s.DATUM
```

Es mag auffallen, dass wir die Spalte *Datum* zweimal abfragen. Über die jeweiligen TField-Eigenschaften *DisplayFormat* wird daraus einmal das Datum und einmal die Uhrzeit extrahiert; beides wird angezeigt, aber in verschiedenen Zeilen.

Sie erinnern sich vielleicht, dass dasselbe Problem vorhin mit einem berechneten Feld gelöst wurde. Vorteil der Lösung per SQL-Anweisung: Schon zur Entwurfszeit werden die Daten korrekt dargestellt. Nachteil: Es müssen mehr Daten übers Netzwerk, aber auf die paar Bytes kommt es nun wirklich nicht an.

Warum wurde nun die Anzeige der Bemerkung auf ein eigenes Band gelegt? Damit dieses Band ausgeblendet werden kann, wenn keine Bemerkung vorliegt. So hat man ohne größeren Aufwand auch die Beschriftung des Textes und den dafür vorgesehenen Platz beseitigt.

```
procedure Tfrm_EintragDetailPrint.ChildStatusBeforePrint(
  Sender: TQRCustomBand; var PrintBand: Boolean);
var
  s: string;
begin
  s := qrStatusBEMERKUNG.AsString;
  PrintBand := (Length(s) > 1);
end;
```

Die angehängten Dateien

Die Liste der angehängten Dateien arbeitet nach demselben Muster wie die Statusänderungen, auch hier werden wieder drei Bänder verwendet. Die Query *qrDatei* verwendet den folgenden JOIN:

```
SELECT
    d.name,
    d.datum AS Datum,
    d.datum AS Zeit,
    d.memo,
    b.name AS Benutzer,
    d.datei
  FROM t_datei d
    INNER JOIN t_benutzer b
      ON b.ID = d.benutzer
  WHERE eintrag = :ID
```

Auch hier wird das Datum zweimal abgefragt, es wurde ja vorhin beschrieben, was wir damit machen.

Es mag Sie verwundern, dass wir auch die Datei hier aufnehmen, da diese doch nicht angezeigt werden kann. Sie dient uns allerdings zur Ermittlung der Dateigröße:

```
procedure Tfrm_EintragDetailPrint.qrLblFileSizePrint
  (sender: TObject; var Value: String);
var
  i: integer;
begin
  i := qrDateiDATEI.BlobSize;
  case i of
    0..1024: Value := IntToStr(i) + ' Byte';
    1025..1048576: Value
      := FormatFloat('0.00', i / 1024) + ' kB';
```

```
    else Value := FormatFloat('0.00', i / 1048576) + ' MB';
  end;
end;
```

Ein Nachteil dieser Lösung soll jedoch nicht verschwiegen werden: Allein für die Ermittlung der Dateigröße muss die ganze angehängte Datei heruntergeladen werden. Nun ist das Anhängen von Dateien nicht dazu gedacht, mehrere große Dateien an einen Eintrag anzuhängen, und das Herunterladen von – sagen wir einmal – 100 kbyte sollte bei lokalen Netzwerken kein größeres Problem darstellen.

Wird die To-Do-Liste erfahrungsgemäß jedoch dazu benutzt, mehrere größere Dateien anzuhängen, dann muss entweder auf die Größenangabe ganz verzichtet werden, oder man ermittelt die Größe beim Schreiben in die Datenbank und speichert sie in einem eigenen Feld – natürlich mit der Gefahr, dass dieser Wert dann mal nicht stimmt.

Fußzeile

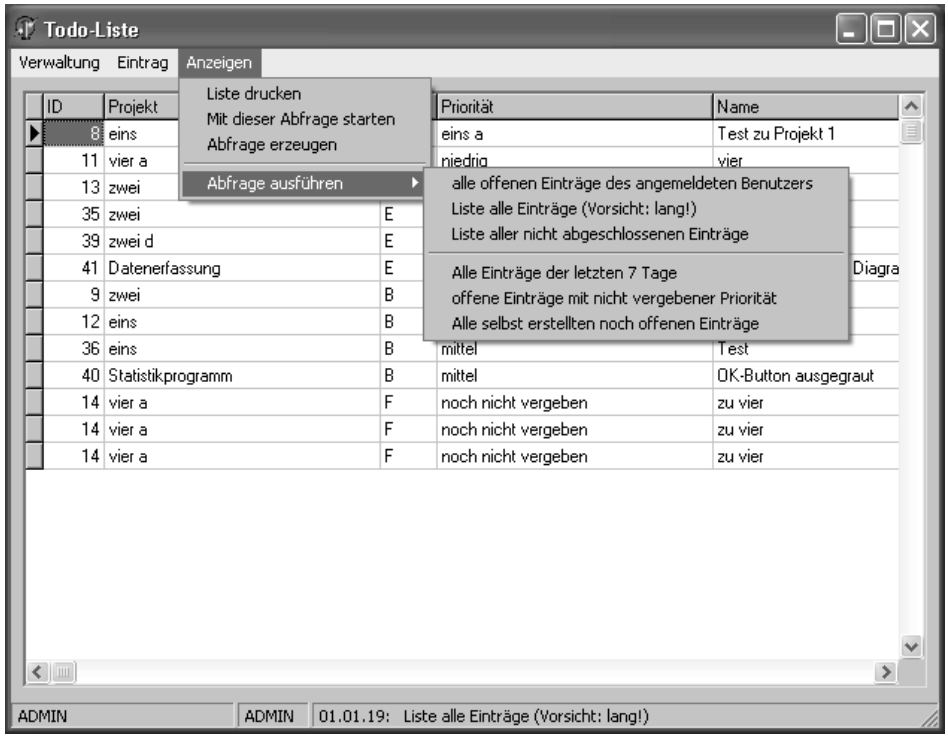
Damit man bei mehrseitigen Ausdrucken noch zuordnen kann, welche Seite zu welchem Eintrag gehört, soll über eine Fußzeile die Bezeichnung des Eintrags (inklusive ID) sowie Datum und Seitenzahl ausgedruckt werden. Das dafür vorgesehene Band erhält als *BandType* den Wert *rbPageFooter*.

```
procedure Tfrm_EintragDetailPrint.QRLabel1Print(sender: TObject;
  var Value: String);
begin
  Value := FormatDateTime('dd.mm.yyyy   hh.mm', now) + '         '
    + 'Seite ' + IntToStr(QuickRep1.PageNumber);
end;
```

18.4 Abfragen erstellen

Für gewöhnlich liegen die Schwierigkeiten nicht darin, Daten irgendwo abzulegen, sondern sie in einer brauchbaren Form wiederzubekommen. Für unser Projekt heißt dies: Wir benötigen brauchbare Listen der Einträge – und zwar nicht von allen Einträgen, sondern von denen, die interessieren.

Nun interessieren jeden Anwender etwas andere Einträge: Den Programmierer interessiert eher, was er noch zu erledigen hat, während der Gruppenleiter sich vermutlich mehr für die neuen Einträge interessiert, die er noch unter seinen Leuten zu verteilen hat. Die Qualitätskontrolle hat noch mal eine andere Sicht der Dinge.



Um das Programm individuell auf die jeweiligen Bedürfnisse einrichten zu können, sind die einzelnen Abfragen in einer Datenbank hinterlegt und können von den Anwendern selbst erstellt werden – entweder als selbst formuliertes SQL-Statement oder über einen Assistenten.

Dabei ist zu unterscheiden zwischen globalen Abfragen, die allen Anwendern zur Verfügung stehen, und Abfragen, die dem einzelnen Benutzer zur Verfügung stehen.

18.4.1 Die Unit u_anzeigen

Auch für diesen Teil wird wieder ein eigenes Projekt angelegt (*p_testAnz.dpr*). Die Schnittstellen-Unit ist *u_anzeigen.pas*. Auch hier gibt es wieder eine Klasse, nämlich *TAnzeigen*.

```
type
  TAnzeigen = class
  private
    FOnAbfrage: TNotifyEvent;
```

```
    FMaster: TMenuItem;
    procedure AbfrageErzeugen(Sender: TObject);
    procedure AbfrageAusfuehren(Sender: TObject);
    procedure MenuErstellen;
  public
    qrAbfragen: TQuery;
    constructor Create;
    class procedure SetMenuItem(Master: TMenuItem);
    class procedure SetEventHandler(EventHandler: TNotifyEvent);
    class procedure SetAbfrage(ID: integer);
  end;
```

Auch hier gibt es wieder eine Klassenmethode *SetMenuItem*, welche neben den Menüeinträgen auch eine Instanz von *TAnzeigen* erstellt.

```
class procedure TAnzeigen.SetMenuItem(Master: TMenuItem);

  function NewMenuItem(ACaption: string; AOnClick: TNotifyEvent;
    ATag: integer): TMenuItem;
  var
    Item: TMenuItem;
  begin
    Item := TMenuItem.Create(Master.Owner);
    Item.Caption := ACaption;
    Item.Tag := ATag;
    Item.OnClick := AOnClick;
    Master.Add(Item);
    result := Item;
  end;

begin
  if not Assigned(Anzeigen)
    then Anzeigen := TAnzeigen.Create;
  NewMenuItem('Abfrage erzeugen', Anzeigen.AbfrageErzeugen, 0);
  NewMenuItem('-', nil, 0);
  Anzeigen.FMaster := NewMenuItem('Abfrage ausführen', nil, 0);
  Anzeigen.MenuErstellen;
end; {class procedure TAnzeigen.SetMenuItem}
```

Es fällt auf, dass diesmal *NewMenuItem* eine Funktion ist, welche einen Zeiger auf den neuen Menüpunkt zurückgibt. Dieser wird im Fall von ABFRAGE AUSFÜHREN dem Feld *FMaster* zugewiesen, anschließend wird *MenuErstellen* aufgerufen.

MenuErstellen wird nicht nur von *SetMenuItem* aufgerufen, sondern immer dann, wenn eine Abfrage erstellt, geändert oder gelöscht wurde. Aus diesem Grund wurden diese Anweisungen in eine eigene Prozedur ausgelagert.

```
procedure TAnzeigen.MenuErstellen;

  procedure NewMenuItem(ACaption: string; AOnClick: TNotifyEvent;
    ATag: integer);
  var
    Item: TMenuItem;
  begin
    Item := TMenuItem.Create(FMaster.Owner);
    Item.Caption := ACaption;
    Item.Tag := ATag;
    Item.OnClick := AOnClick;
    FMaster.Add(Item);
  end;

  procedure AddItems(FilterText: string);
  begin
    with Anzeigen.qrAbfragen do
    begin
      SQL.Clear;
      SQL.Add('SELECT id, name FROM t_anzeige');
      SQL.Add(Filtertext);
      Open;
      while not EOF do
      begin
        NewMenuItem(FieldByName('name').AsString,
          Anzeigen.AbfrageAusfuehren,
          FieldByName('ID').AsInteger);
        Next;
      end;
    end; {with Anzeigen.qrAbfragen do}
  end; {procedure AddItems}

begin
  FMaster.Clear;
  AddItems('WHERE benutzer IS NULL');
  NewMenuItem('-', nil, 0);
  AddItems('WHERE benutzer = ' + IntToStr(gvUserID));
```

```
  Anzeigen.qrAbfragen.SQL.Text
    := 'SELECT ID, name, anweisung FROM t_anzeige '
      + 'WHERE ID = :ID';
end; {procedure TAnzeigen.MenuErstellen}
```

Mit *AddItems* erfolgt zweimal eine Abfrage von *t_anzeige*. Beim ersten Mal werden über *benutzer IS NULL* die globalen Abfragen in Menüpunkte umgewandelt. Die lokale Prozedur *NewMenuItem* haben wir ja schon einige Male verwendet, die hier verwendete unterscheidet sich lediglich dadurch, dass die ID des jeweiligen Datensatzes in die Eigenschaft *Tag* des Menüpunktes geschrieben wird.

Nach einem Trennstrich folgen die personalisierten Einträge, hier wird dann ein Filter auf *gvUserID* gesetzt. Anschließend wird die SQL-Abfrage gesetzt, die wir für *AbfrageAusfuehren* benötigen.

Der Vollständigkeit halber: *qrAbfragen* muss natürlich erst einmal erzeugt werden, auch *DatabaseName* will irgendwann gesetzt werden.

```
constructor TAnzeigen.Create;
begin
  inherited;
  qrAbfragen := TQuery.Create(Application);
  qrAbfragen.DatabaseName := '_INTERN';
end; {constructor TAnzeigen.Create}
```

Wird dann auf einen dieser Menüpunkte geklickt, die eine Abfrage repräsentieren, dann wird *AbfrageAusfuehren* ausgeführt:

```
procedure TAnzeigen.AbfrageAusfuehren(Sender: TObject);
var
  i: integer;
begin
  with qrAbfragen do
  begin
    Close;
    i := (Sender as TComponent).Tag;
    ParamByName('ID').AsInteger := i;
    Open;
  end; {with qrAbfragen do}
  if Assigned(FOnAbfrage)
    then FOnAbfrage(qrAbfragen);
end; {procedure TAnzeigen.AbfrageAusfuehren}
```

Im *Tag* des jeweiligen Menüpunktes steht die ID des dazugehörenden Datensatzes, auf den *qrAbfragen* gefiltert wird.

Die SQL-Anweisung wird später dann von einer *TQuery*-Instanz des Hauptformulars ausgeführt, diese wird über das Ereignis *OnAbfrage* benachrichtig. Als *Sender* wird dabei ein Zeiger auf *qrAbfragen* übergeben, so dass auf die dazugehörenden Daten zugegriffen werden kann.

Da die Variable *Anzeigen* im *implementation*-Teil der Unit liegt und somit ein Zugriff von anderen Units aus nicht möglich ist, benötigen wir eine Möglichkeit, das Ereignis mit einer Ereignisbehandlungsroutine zu verbinden.

```
class procedure TAnzeigen.SetEventHandler
  (EventHandler: TNotifyEvent);
begin
  Anzeigen.FOnAbfrage := EventHandler;
end;
```

Des Weiteren soll der Anwender einstellen können, welche Abfrage ihm beim Programmstart angezeigt wird – traditionell eine Liste der zu bearbeitenden Einträge, aber das ist nicht zwingend. Die ID dieser Abfrage wird in eine Ini-Datei gespeichert, wir benötigen nun nur noch eine Möglichkeit, die betreffende Abfrage auszuführen:

```
class procedure TAnzeigen.SetAbfrage(ID: integer);
begin
  with Anzeigen.qrAbfragen, Anzeigen do
  begin
    Close;
    ParamByName('ID').AsInteger := ID;
    Open;
    if Assigned(FOnAbfrage)
      then FOnAbfrage(qrAbfragen);
  end; {with qrAbfragen do}
end; {class procedure TAnzeigen.SetAbfrage}
```

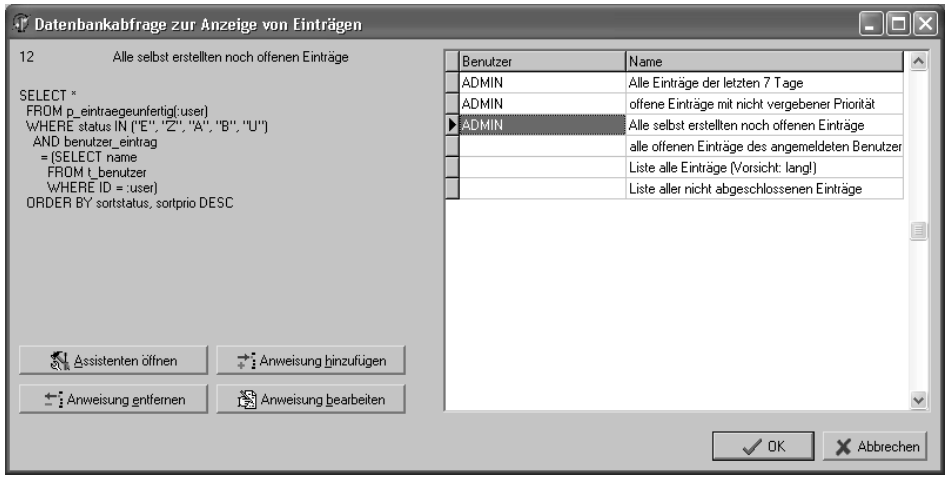
18.4.2 Abfragen erstellen

Wird der Menüpunkt ABFRAGE ERZEUGEN aufgerufen, wird mit *AbfrageErzeugen* das Formular *Tfrm_abfrage* aufgerufen, anschließend wird das Menü neu erstellt.

```
procedure TAnzeigen.AbfrageErzeugen(Sender: TObject);
begin
  frm_data.ModalAnzeigen(Tfrm_abfrage, true);
  MenuErstellen;
end;
```

Das Formular Tfrm_abfrage

Das Formular *TFrm_abfrage* dient zur Anzeige der globalen und der jeweiligen personalisierten Abfragen. Nur Benutzer mit Administratorrechten können Anweisungen hinzufügen oder bearbeiten, bei anderen Benutzern ist dieser Button gesperrt. Sie können Abfragen lediglich über den Assistenten erstellen. Auf diese Weise wird vermieden, dass Anwender über eine selbst formulierte SQL-Anweisung an Daten herankommen, für die sie keine Leseberechtigung haben. Der Button *Anweisung entfernen* wird bei Anwendern ohne Administratorrecht für alle globalen Abfragen gesperrt.



Die SQL-Anweisung von *qrAnzeige* filtert auf den aktuellen Benutzer und IS NULL, so dass auch die globalen Abfragen angezeigt werden. Über einen JOIN wird der Benutzer im Klartext angezeigt.

```
SELECT
    a.id,
    a.name,
    a.anweisung,
    a.benutzer,
    b.name AS Benutzername
  FROM t_anzeige a
    LEFT OUTER JOIN t_benutzer b
      ON a.benutzer = b.id
  WHERE (benutzer = :benutzer)
    OR benutzer IS NULL
  ORDER BY a.benutzer, a.id
```

Für jeden gewählten Datensatz muss der *Enabled*-Status von *btnDelete* so gesetzt werden, dass Nicht-Administratoren keine globalen Abfragen löschen können:

```
procedure Tfrm_abfrage.qrAnzeigeAfterScroll(DataSet: TDataSet);
// für AfterOpen und AfterScroll
begin
  btnDelete.Enabled := (qrAnzeigeBENUTZER.IsNull = false)
    or gvIsAdmin;
end;
```

Der Enabled-Status von *btnEdit* und *btnInsert* ist nicht vom Datensatz abhängig und wird deshalb beim Anzeigen des Formulars gesetzt.

```
procedure Tfrm_abfrage.FormShow(Sender: TObject);
begin
  btnEdit.Enabled := gvIsAdmin;
  btnInsert.Enabled := gvIsAdmin;
end;
```

Zum Bearbeiten von *qrAnzeige* wird dann wieder eine *TUpdateSQL*-Instanz verwendet. *btnInsert* ruft lediglich die Methode *Insert* auf, *btnEdit* nur *Edit*. Für beide Fälle wird die gleiche Ereignisbehandlungsroutine verwendet:

```
procedure Tfrm_abfrage.qrAnzeigeAfterInsert(DataSet: TDataSet);
// AfterEdit und AfterInsert
var
  Form: Tfrm_EditAnweisung;
begin
  Form := Tfrm_EditAnweisung.Create(nil);
  try
    Form.SetWerte(dsAnzeige);
    if Form.ShowModal = mrOK then
    begin
      if Form.rbPrivat.Checked
        then qrAnzeigeBENUTZER.AsInteger := gvUserID;
      qrAnzeige.Post;
    end {if Form.ShowModdal = mrOK then}
    else qrAnzeige.Cancel;
  finally
    Form.Free;
  end;
end; {procedure Tfrm_abfrage.qrAnzeigeAfterInsert}
```

Das Formular *Tfrm_EditAnweisung*, das dabei aufgerufen wird, arbeitet mit datensensitiven Dialogelementen, so dass dann nur noch *Post* oder *Cancel* aufgerufen werden muss.

Erstellt ein Administrator einen neuen Eintrag, dann kann er wählen, ob dieser ein globaler oder ein personalisierter Eintrag werden soll. Wird ein Eintrag geändert oder von einem Nicht-Administrator erstellt, dann sind diese RadioButtons gesperrt. Je nachdem, ob eine Anweisung erstellt oder geändert wird, bekommt das Formular eine andere Überschrift, und außerdem muss die Eigenschaft *DataSource* von *DBMemo1* und *DBEdit1* gesetzt werden.

```
procedure Tfrm_EditAnweisung.SetWerte(DataSource: TDataSource);
begin
  DBMemo1.DataSource := DataSource;
  DBEdit1.DataSource := DataSource;
  if DataSource.DataSet.State = dsInsert then
  begin
    Caption := 'Neue Anweisung hinzufügen';
    rbPrivat.Enabled := gvIsAdmin;
    rbPublic.Enabled := gvIsAdmin;
  end {if DataSource.DataSet.State = dsInsert then}
  else
  begin
    Caption := 'Anweisung bearbeiten';
    rbPrivat.Enabled := false;
```

```
    rbPublic.Enabled := false;
    rbPublic.Checked
      := DataSource.DataSet.FieldByName('Benutzer').IsNull;
  end; {else DataSource.DataSet.State = dsInsert then}
end; {procedure Tfrm_EditAnweisung.SetWerte}
```

Für das Löschen ist dann wieder eine Sicherheitsabfrage vorgesehen:

```
procedure Tfrm_abfrage.btnDeleteClick(Sender: TObject);
begin
  if MessageDlg('Soll die Anweisung "' + qrAnzeigeNAME.AsString
        + '" gelöscht werden?', mtConfirmation,
        [mbYes, mbNo], 0) = mrYes
    then qrAnzeige.Delete;
end; {procedure Tfrm_abfrage.btnDeleteClick}
```

Und außerdem muss nach dem Erstellen oder Löschen einer Abfrage *qrAnzeige* aktualisiert werden.

```
procedure Tfrm_abfrage.qrAnzeigeAfterPost(DataSet: TDataSet);
// für AfterPost und AfterEdit
begin
  UpdateRefreshQuery(qrAnzeige, 'ID');
end;
```

Der Assistent

Wird der Button *Assistenten öffnen* geklickt, dann wird das Formular *Tfrm_AssiAbfrage* modal geöffnet.

```
procedure Tfrm_abfrage.btnAssiClick(Sender: TObject);
var
  Form: Tfrm_AssiAbfrage;
  sl: TStringList;
begin
  Form := Tfrm_AssiAbfrage.Create(nil);
  sl := TStringList.Create;
  try
    if Form.ShowModal = mrOK then
    begin
      with upAnzeige.Query[ukInsert] do
      begin
        ParamByName('Name').AsString := Form.edtName.Text;
        sl.Text := Form.GetText;
```

```
          ParamByName('Anweisung').Assign(sl);
          ParamByName('Benutzer').AsInteger := gvUserID;
          ExecSQL;
          UpdateRefreshQuery(qrAnzeige, 'ID');
        end; {with upAnzeige.Query[ukInsert] do}
      end; {if Form.ShowModal = mrOK then}
    finally
      Form.Free;
      sl.Free;
    end;
end; {procedure Tfrm_abfrage.btnAssiClick}
```

Die Anweisungen zum Erstellen des Formulars, seiner modalen Anzeige, der Aktion im Falle, dass es mit *Ok* geschlossen wird, und der Aktualisierung von *qrAnzeige* haben wir so oder so ähnlich schon ein paar Mal gehabt – schauen wir uns das an, was neu ist:

Würden wir hier mit dem Befehl *Insert* arbeiten, dann würde *qrAnzeigeAfterInsert* den Dialog *Tfrm_EditAnweisung* anzeigen. Entweder wir klemmen diese Ereignisbehandlungsroutine vorübergehend ab, oder wir wählen einen anderen Weg: Die *TUpdateSQL*-Komponente *upAnzeige* beinhaltet nämlich drei *TQuery*-Instanzen, darunter auch eine für den Insert-Fall:

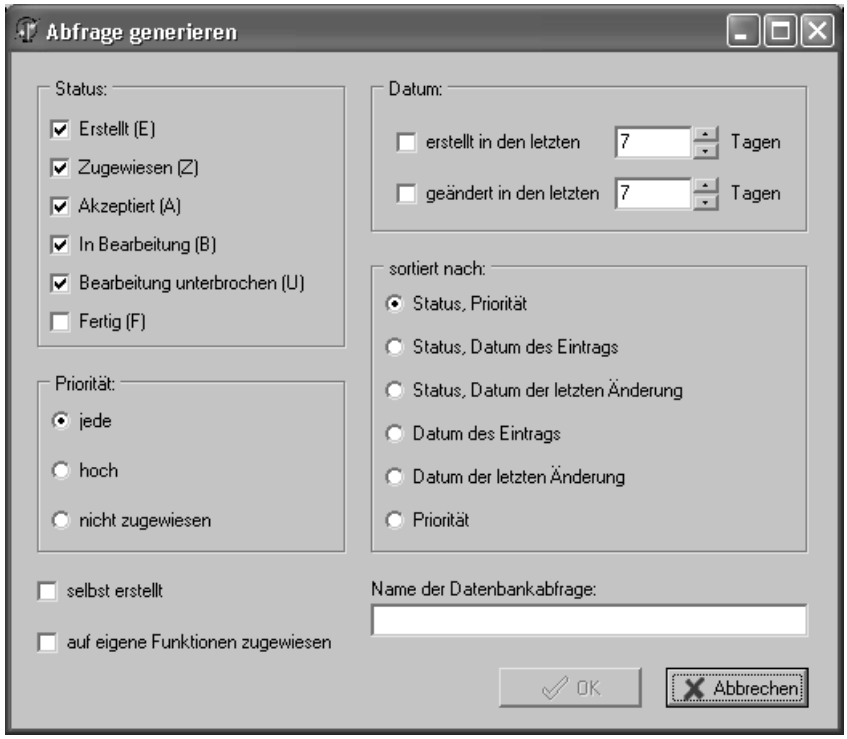
```
insert into t_anzeige
  (NAME, ANWEISUNG, BENUTZER)
values
  (:NAME, :ANWEISUNG, :BENUTZER)
```

Wir brauchen hier nur noch die entsprechenden Parameter zuzuweisen und ExecSQL aufzurufen. Die SQL-Anweisung, die der Assistent generiert, erhalten wir mit der *Tfrm_AssiAbfrage*-Methode *GetText*.

In *Tfrm_AssiAbfrage* besteht die Möglichkeit, sich aus den gebräuchlichsten Filter-Optionen eine SQL-Anweisung generieren zu lassen:

- Auf den Status, um beispielsweise alle noch nicht abgeschlossenen oder alle gerade erst erstellten Einträge zu erhalten.
- Auf das Datum, um die Einträge zu erhalten, die in einem definierten Zeitraum erstellt worden sind oder bei denen sich der Status geändert hat.
- Einträge mit hoher Priorität oder deren Priorität noch nicht zugewiesen wurde.
- Selbst erstellte Einträge: Diese Option benötigen vor allem die Tester, um zu sehen, welchen »Fortschritt« die eigenen Meldungen machen. (Die Programmierer haben für die Liste ihrer offenen Einträge ohnehin eine STORED PROCEDURE.)

- Einträge, die einer Funktion zur Bearbeitung zugewiesen wurden, deren Mitglied der Anwender ist. Einer der Anwender dieser Funktion muss sich nun dafür entscheiden, den betreffenden Eintrag zu bearbeiten und den Status auf *Akzeptiert* setzen.



Die zu erstellende Abfrage würde keine Datensätze liefern, wenn nicht wenigstens einer der Stati gewählt wäre. Als gemeinsame *OnClick*-Ereignisbehandlungsroutine aller Status-CheckBoxen wird deshalb *cbErstelltClick* verwendet.

```
procedure Tfrm_AssiAbfrage.cbErstelltClick(Sender: TObject);
begin
  btnOK.Enabled := (cbErstellt.Checked
      or cbZugewiesen.Checked
      or cbAkzeptiert.Checked
      or cbBearbeitung.Checked
      or cbUnterbrochen.Checked
      or cbFertig.Checked)
    and (Length(edtName.Text) > 2);
end;
```

Zudem wird noch geprüft, ob der eingegebene Name wenigstens drei Zeichen umfasst – *cbErstelltClick* wird somit auch der Komponente *edtName* als *OnChange*-Ereignisbehandlungsroutine zugewiesen.

Mit der Funktion *GetText* wird die SQL-Anweisung generiert:

```
function Tfrm_AssiAbfrage.GetText: string;

  procedure AZ(Text: string);
  begin
    result := result + Text + #13#10;
  end;

  ...

begin
  AZ('SELECT *');
  if cbFertig.Checked
    then AZ('  FROM p_eintraegealle(:user)')
    else AZ('  FROM p_eintraegeunfertig(:user)');
  DoStatus;
  DoDatum;
  DoPrio;
  if cbEigen.Checked
    then DoEigen;
  if cbFunktion.Checked
    then DoFunktion;
  DoSort;
end; {function Tfrm_AssiAbfrage.GetText}
```

Wegen der Länge dieser Methode wollen wir alle lokalen Prozeduren (bis auf *AZ*) getrennt besprechen. *AZ* steht übrigens für *AddZeile* und fügt dem Funktionsergebnis den Text neben einem Zeilenumbruch hinzu. Mit dieser Routine bekommt man den Quelltext ein wenig übersichtlicher.

Für den Fall, dass *cbFertig* gewählt ist (und somit auch schon abgeschlossene Einträge in die Ergebnismenge aufgenommen werden sollen), erfolgt eine Abfrage mit der STORED PROCEDURE *p_eintraegealle*:

```
CREATE PROCEDURE p_eintraegealle
  (para_benutzer INTEGER)
 RETURNS
  (id INTEGER,
  projekt VARCHAR(30),
```

```
  status VARCHAR(1),
  prioritaet VARCHAR(30),
  name VARCHAR(50),
  datum_eintrag TIMESTAMP,
  datum_aender TIMESTAMP,
  benutzer_eintrag VARCHAR(30),
  funktion VARCHAR(30),
  sortprio INTEGER,
  sortstatus INTEGER)
AS
  DECLARE VARIABLE v_id INTEGER;
  DECLARE VARIABLE v_idprojekt INTEGER;
  DECLARE VARIABLE v_benutzer_aender INTEGER;
  DECLARE VARIABLE v_lesen CHAR(1);
BEGIN
  FOR SELECT id
    FROM t_eintrag
    INTO :v_id
  DO
  BEGIN
    SELECT
        e.id,
        e.name,
        e.datum_eintrag,
        e.datum_aender,
        e.status,
        e.benutzer_aender,
        b.name,
        f.name,
        p.name,
        p.id,
        i.name,
        i.sortier,
        n.sortier
    FROM p_eintragAktuell(:v_id) e
      INNER JOIN t_benutzer b
        ON e.benutzer_eintrag = b.id
      LEFT OUTER JOIN t_funktion f
        ON e.funktion = f.id
      INNER JOIN t_projekt p
        ON e.projekt = p.id
```

```
      INNER JOIN t_prioritaet i
        ON e.prioritaet = i.id
      LEFT OUTER JOIN tn_status n
        ON e.status = n.id
    INTO
        :id,
        :name,
        :datum_eintrag,
        :datum_aender,
        :status,
        :v_benutzer_aender,
        :benutzer_eintrag,
        :funktion,
        :projekt,
        :v_idprojekt,
        :prioritaet,
        :sortprio,
        :sortstatus;
    SELECT
        lesen
      FROM p_checkrechte(:v_idprojekt, :para_benutzer)
      INTO :v_lesen;
    IF (v_lesen = 'J')
      THEN SUSPEND;
  END
END
```

Die Prozedur geht durch alle Einträge, ermittelt mittels *p_eintragAktuell* den aktuellen Status des Eintrags und fügt für Benutzer, Funktion, Projekt und Priorität die Klartextnamen hinzu. Über *tn_status* wird die Sortierreihenfolge für den Status ermittelt.

Je nachdem, welche Stati nun im Assistenten gewählt wurden, wird auf den Status mit einer WHERE-Klausel und dem Operator IN gefiltert. Ist die To-Do-Liste einige Jahre im Einsatz, dann sammeln sich eine Menge fertiger Einträge an. Es beschleunigt nun nicht die Ausführung der Abfrage, diese erst in die Ergebnismenge der STORED PROCEDURE aufzunehmen und anschließend herauszufiltern. Aus diesem Grund wird für den Fall, dass der Status *F* im Assistenten nicht ausgewählt wird, eine Abfrage auf *p_eintragUnfertig* durchgeführt. Diese STORED PROCEDURE gleicht weitgehend *p_eintragAktuell*, in der IF-Anweisung am Ende erfolgt jedoch eine Filterung auf alle Einträge, die noch nicht den Status *F* haben:

```
IF ((v_lesen = 'J')
     AND NOT (STATUS = 'F'))
  THEN SUSPEND;
```

Schauen wir uns nun die einzelnen lokalen Prozeduren an, welche Stück für Stück die Filteranweisung zusammenstellen:

```
procedure DoStatus;
var
  s: string;
begin
  if cbErstellt.Checked then s := ', "E"';
  if cbZugewiesen.Checked then s := s + ', "Z"';
  if cbAkzeptiert.Checked then s := s + ', "A"';
  if cbBearbeitung.Checked then s := s + ', "B"';
  if cbUnterbrochen.Checked then s := s + ', "U"';
  if cbFertig.Checked then s := s + ', "F"';
  AZ('  WHERE status IN (' + copy(s, 3, MaxInt) + ')');
end; {procedure DoStatus}
```

Für die Filterung auf den Status wird mit dem Mengenoperator IN gearbeitet. Um die einzelnen Buchstaben aneinander fügen zu können, wird grundsätzlich immer ein Komma und ein Leerzeichen vorangestellt, das somit am Anfang auch immer gelöscht werden kann.

```
procedure DoDatum;
begin
  if cbDatErst.Checked and cbDatAend.Checked then
  begin
    AZ('    AND (datum_eintrag > ("now" - '
      + edtDatErst.Text + '))');
    AZ('      OR (datum_aender > ("now" - '
      + edtDatAend.Text + '))');
  end
  else
  begin
    if cbDatErst.Checked
      then AZ('    AND datum_eintrag > ("now" - '
        + edtDatErst.Text + ')');
    if cbDatAend.Checked
      then AZ('    AND datum_aender > ("now" - '
        + edtDatAend.Text + ')');
  end; {else cbDatErst.Checked and cbDatAend.Checked}
end; {procedure DoDatum}
```

Die Filterung auf das Datum kann sowohl den Eintrag als auch die letzte Statusänderung betreffen und bezieht sich immer auf eine Menge zurückliegender Tage. Bei täglicher Benutzung könnte man beispielsweise auf die Einträge des letzten und des heutigen Tages filtern, während man eine Abfrage für »nach dem Urlaub« wohl etwas großzügiger fassen würde.

Wird sowohl auf Eintrags- als auch auf Änderungsdatum gefiltert, dann wird dies mit einer OR-Verknüpfung verbunden

```
procedure DoPrio;
begin
  case rgPrio.ItemIndex of
    1: AZ('    AND sortprio > 80');
    2: AZ('    AND sortprio >= 99');
  end;
end; {procedure DoPrio}
```

Die Filterung auf die Priorität erfolgt mit Hilfe der Prioritäts-Sortierung, so dass auch selbst definierte Prioritäten – sofern sie hohe Relevanz haben – berücksichtigt werden.

```
procedure DoEigen;
begin
  AZ('    AND benutzer_eintrag ');
  AZ('      = (SELECT name');
  AZ('        FROM t_benutzer');
  AZ('        WHERE ID = :user)');
end; {procedure DoEigen}
```

Sollen nur Einträge angezeigt werden, die vom angemeldeten Benutzer selbst verfasst wurden, dann besteht die Schwierigkeit darin, dass zwar eine Filterung auf den Parameter *:user* erfolgen kann, die STORED PROCEDURE den Namen jedoch im Klartext zurückgibt. Mittels einer Unterabfrage wird deshalb der Name aus der ID ermittelt.

```
procedure DoFunktion;
begin
  AZ('    AND funktion ');
  AZ('      IN (SELECT name');
  AZ('        FROM t_funktion f');
  AZ('          INNER JOIN t_benutzer_funktion b');
  AZ('            ON b.funktion = f.ID');
  AZ('        WHERE b.benutzer = :user)');
end; {procedure DoFunktion}
```

Sollen nur die Einträge angezeigt werden, die einer der Funktionen zur Bearbeitung zugewiesen sind, welcher der angemeldete Benutzer zugeordnet ist, dann muss mittels des Parameters *:user* ermittelt werden, welche Funktionen dem angemeldeten Benutzer zugeordnet sind. Die Filterung erfolgt dann mit dem Mengenoperator IN.

```
procedure DoSort;
begin
  case rgSort.ItemIndex of
    0: AZ('  ORDER BY sortstatus, sortprio DESC');
    1: AZ('  ORDER BY sortstatus, datum_eintrag DESC');
    2: AZ('  ORDER BY sortstatus, datum_aender DESC');
    3: AZ('  ORDER BY datum_eintrag DESC');
    4: AZ('  ORDER BY datum_aender DESC');
    5: AZ('  ORDER BY sortprio DESC');
  end; {case rgSort.ItemIndex of}
end; {procedure DoSort}
```

Zuletzt gibt es noch sechs Varianten, die Ergebnismenge zu sortieren, wobei die aktuelleren Einträge oder die Einträge mit höherer Priorität zuerst erscheinen sollen. Dementsprechend sind diese Spalten absteigend zu sortieren.

18.5 Das Hauptprojekt

Nun wollen wir unsere drei Testprojekte zu einem Hauptprojekt (*p_To-Do.dpr*) zusammenfassen. Neben einem Hauptformular (*frm_main* in *u_main*) und dem Datenmodul fügen wir die drei Schnittstellen-Units *u_anzeigen.pas*, *u_eintrag.pas* und *u_verwaltung.pas* hinzu. Später werden wir noch eine Unit für den Listendruck (*u_PrintListe.pas*) ergänzen.

18.5.1 Das Hauptformular

Das Hauptformular besteht lediglich aus einem Menü, einer Statusleiste und einem Grid, in dem das Ergebnis der jeweiligen Abfrage angezeigt wird. Da viele der Menüpunkte auch nicht in *u_main* implementiert sind, ist diese Unit erstaunlich kurz (222 Zeilen).

```
procedure Tfrm_main.FormCreate(Sender: TObject);
begin
  TVerwaltung.SetMenuItem(Verwaltung1);
  TEintrag.SetMenuItem(Eintrag1);
  TAnzeigen.SetMenuItem(Anzeigen1);
```

```
  TAnzeigen.SetEventHandler(Anzeigen);
end;
```

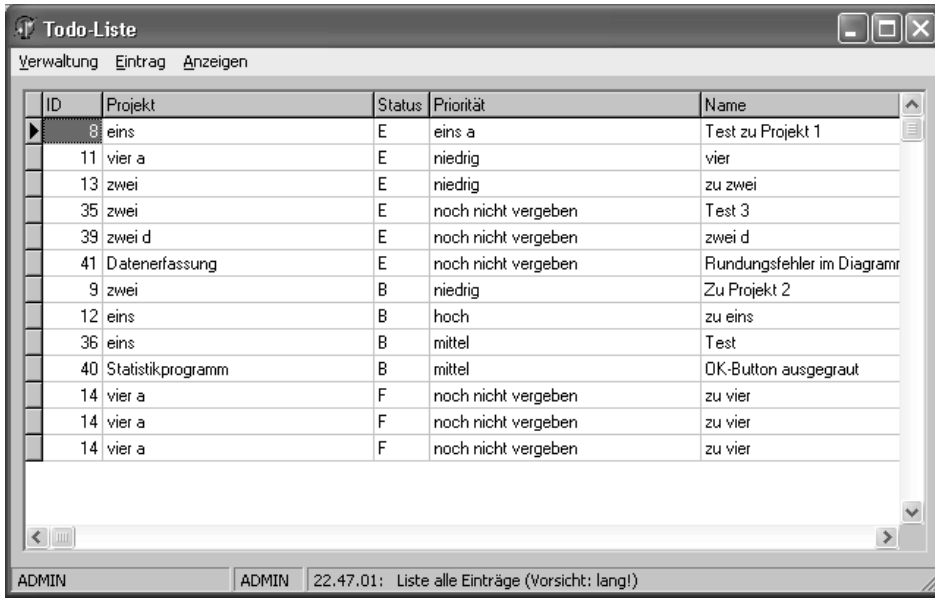
Beim Erstellen des Formulars werden die *SetMenuItem*-Klassenmethoden der einzelnen Modul-Klassen aufgerufen. Auf diese Weise werden die benötigten Instanzen erstellt und die Menüeinträge hinzugefügt. Außerdem muss die Ereignisbehandlungsroutine für die Wahl einer neuen Abfrage hinzugefügt werden.

```
procedure Tfrm_main.FormShow(Sender: TObject);

  procedure DoIni;
  var
    Ini: TIniFile;
    s: string;
  begin
    s := ExtractFilePath(Application.Exename);
    Ini := TIniFile.Create(s + 'To-Do.ini');
    try
      Left := Ini.ReadInteger(Name, 'Left', 0);
      Top := Ini.ReadInteger(Name, 'Top', 0);
      Width := Ini.ReadInteger(Name, 'Width', 800);
      Height := Ini.ReadInteger(Name, 'Height', 600);
      gvUserName := Ini.ReadString(Name, 'User', 'ADMIN');
      FAbfrageStart := Ini.ReadInteger(Name, 'Abfrage', 1);
    finally
      Ini.Free;
    end;
  end; {procedure DoIni}

begin
  DoIni;
  if not TVerwaltung.PasswortDialog
    then Application.Terminate;
  StatusBar1.Panels[0].Text := ' ' + gvUserName;
  if gvIsAdmin
    then StatusBar1.Panels[1].Text := ' ADMIN';
  TAnzeigen.SetAbfrage(FAbfrageStart);
end; {procedure Tfrm_main.FormShow}
```

Vor dem Anzeigen des Formulars wird zuerst die Ini-Datei eingelesen und somit werden die Position und Abmessungen gesetzt. Des Weiteren wird der Benutzername in der Ini-Datei gespeichert sowie die ID der Abfrage, die als Erstes angezeigt werden soll.

Anschließend wird der Passwortdialog aufgerufen. Erfolgt keine ordnungsgemäße Anmeldung (Abbruch, unbekannter Benutzer, falsches Passwort), wird die Anwendung beendet.

Anderenfalls wird der Name des angemeldeten Benutzers in die Statusleiste geschrieben, des Weiteren, ob er Administratorrechte hat. Zuletzt wird die Start-Abfrage ausgeführt.



Erwartungsgemäß werden beim Beenden der Anwendung (genauer beim Schließen des Hauptformulars) einige Daten in die Ini-Datei geschrieben:

```
procedure Tfrm_main.FormClose(Sender: TObject;
  var Action: TCloseAction);
var
  Ini: TIniFile;
  s: string;
begin
  s := ExtractFilePath(Application.Exename);
  Ini := TIniFile.Create(s + 'To-Do.ini');
  try
    Ini.WriteInteger(Name, 'Left', Left);
    Ini.WriteInteger(Name, 'Top', Top);
    Ini.WriteInteger(Name, 'Width', Width);
    Ini.WriteInteger(Name, 'Height', Height);
```

```
    Ini.WriteInteger(Name, 'Abfrage', FAbfrageStart);
    Ini.WriteString(Name, 'User', gvUserName);
  finally
    Ini.Free;
  end;
end;
```

Ausführen einer Anweisung

```
procedure Tfrm_main.Anzeigen(Sender: TObject);
var
  i: integer;
begin
  with Query1 do
  begin
    SQL.Text
      := (Sender as TQuery).FieldByName('anweisung').AsString;
    for i := 0 to ParamCount - 1
      do Params[i].AsInteger := gvUserID;
    if Length(SQL.Text) > 10 then
    begin
      Open;
      SpaltenUeberschriften;
      StatusBar1.Panels[2].Text
        := ' ' + FormatDateTime('hh.mm.ss:   ', now)
          + (Sender as TQuery).FieldByName('name').AsString;
      FAbfrageAktuell := (Sender as
        TQuery).FieldByName('ID').AsInteger;
    end;
  end; {with Query1 do}
end; {procedure Tfrm_main.Anzeigen}
```

Die Routine *Anzeigen* ist die Ereignisbehandlungsroutine für das Ausführen einer Abfrage. Dem Sender wird dabei der Zeiger auf die *TQuery*-Instanz in *u_anzeigen* übergeben, so dass damit ID, Name und Anweisung erfragt werden können.

Zunächst wird die Anweisung *Query1* zugewiesen. Allen darin enthaltenen Parametern wird die User-ID zugewiesen. Bei Anweisungen, die mit dem Assistenten erstellt worden sind, sollte es nur ein Parameter sein, Administratoren haben jedoch die Möglichkeit, SQL-Anweisungen frei zu erstellen.

Für den Fall, dass überhaupt eine SQL-Anweisung gespeichert ist (wenn nicht mindestens zehn Zeichen vorhanden sind, wird unterstellt, dass keine SQL-Anweisung vorliegt), wird diese dann ausgeführt. Die Prozedur *SpaltenUeberschriften* schauen wir uns gleich an. In die StatusBar wird der Name der Anweisung geschrieben, so dass man später noch erkennen kann, was da eigentlich angezeigt werden soll. Anhand der Uhrzeit soll man erkennen können, wie aktuell die angezeigte Abfrage ist.

Die ID der Abfrage wird in *FAbfrageAktuell* gespeichert. Wird nun der Menüpunkt MIT DIESER ABFRAGE STARTEN gewählt, dann wird diese wiederum *FAbfrageStart* zugewiesen und somit in die Ini-Datei geschrieben, damit mit dieser Abfrage das Programm begonnen wird.

SQL-Anweisungen haben meist Spaltenüberschriften in Großbuchstaben und auch ansonsten lässt sich hier noch ein wenig optische Kleinarbeit leisten:

```
procedure Tfrm_main.SpaltenUeberschriften;

  function Ueberschrift(Text: string): string;
  begin
    Text := AnsiUpperCase(Text);
    result := Text;
    if Text = 'PROJEKT'
      then result := 'Projekt';
    if Text = 'STATUS'
      then result := 'Status ';
    if Text = 'PRIORITAET'
      then result := 'Priorität';
    if Text = 'NAME'
      then result := 'Name';
    if Text = 'DATUM_EINTRAG'
      then result := 'Eintragsdatum ';
    if Text = 'DATUM_AENDER'
      then result := 'geändert ';
    if Text = 'BENUTZER_EINTRAG'
      then result := 'eingetragen von';
    if Text = 'FUNKTION'
      then result := 'zu erledigen von';
    if Text = 'SORTPRIO'
      then result := 'Prio';
    if Text = 'SORTSTATUS'
      then result := 'Stat';
  end; {function Ueberschrift}
```

```
var
  i: integer;
begin
  with Query1 do
  begin
    for i := 0 to FieldCount - 1 do
    begin
      Fields[i].DisplayLabel
        := Ueberschrift(Fields[i].DisplayLabel);
      if Fields[i] is TDateTimeField then
      begin
        TDateTimeField(Fields[i]).DisplayFormat := 'ddd  dd.mmm.';
        Fields[i].DisplayWidth := 12;
      end;
      if Fields[i] is TIntegerField
        then Fields[i].DisplayWidth := 6;
    end; {for i := 0 to FieldCount do}
  end; {with Query1 do}
end; {procedure Tfrm_main.SpaltenUeberschriften}
```

In einer Schleife werden alle vorhandenen Spalten durchlaufen. Mit der lokalen Funktion *Ueberschrift* wird die Eigenschaft *DisplayLabel* »gerade gezogen«; alle Spaltenüberschriften, die von den STORED PROCEDURES *p_eintraegealle* und *p_eintraegeunfertig* geliefert werden, werden hier in Kleinbuchstaben (nebst einigen anderen Änderungen) umgewandelt.

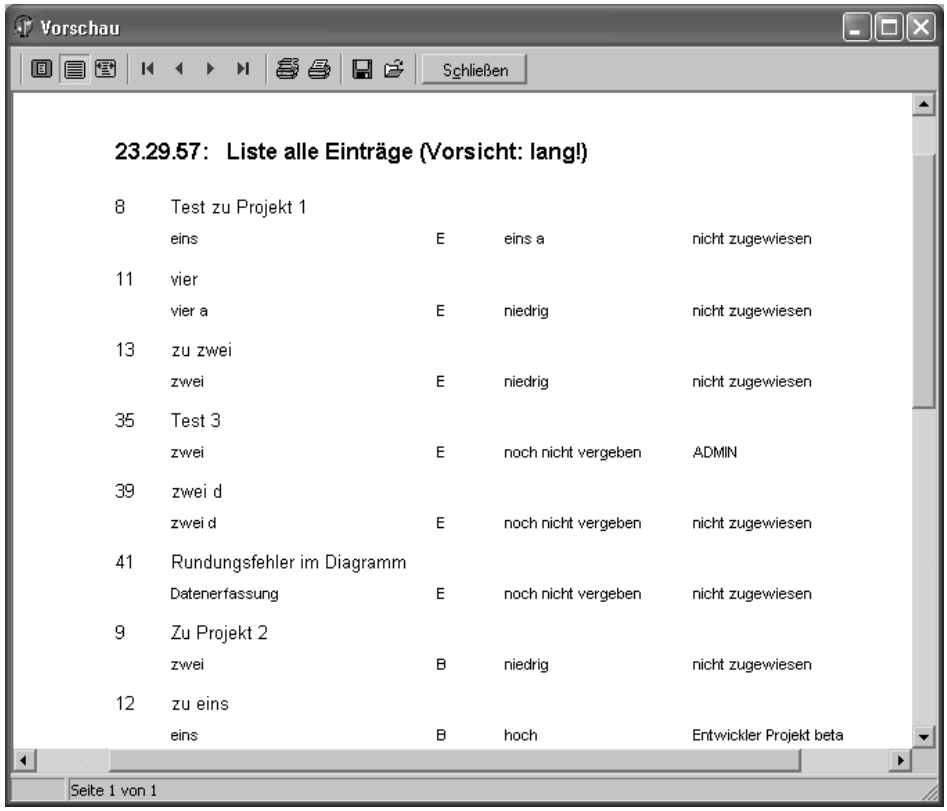
Datumsangaben werden mit Wochentag, aber ohne Jahr und Uhrzeit dargestellt, die Spalte kann dann auch etwas schmaler ausfallen. Auch die hinteren Integer-Spalten (für Prioritäts- und Statussortierung) können etwas schmaler gemacht werden.

18.5.2 Liste drucken

Die Ergebnisse der Abfragen sollen natürlich auch ausgedruckt werden können, dazu dient der Menüpunkt LISTE DRUCKEN.

```
procedure Tfrm_main.Listedrucken1Click(Sender: TObject);
begin
  frm_PrintListe.QuickRep1.Preview;
end;
```

Hier soll das Formular mal nicht vorher erzeugt und hinterher freigegeben, sondern beim erstmaligen Bedarf erzeugt und anschließend beibehalten werden.



Dazu wird die Unit wie folgt umgebaut:

```
unit u_PrintListe;

interface

  ...

function frm_PrintListe: Tfrm_PrintListe;

implementation

  ...

var
  var_PrintListe: Tfrm_PrintListe;
```

```
function frm_PrintListe: Tfrm_PrintListe;
begin
  if var_PrintListe = nil
    then var_PrintListe := Tfrm_PrintListe.Create(Application);
  result := var_PrintListe;
end;
```

Aus der globalen Variablen *frm_PrintListe* wird eine Funktion gemacht. Eine Variable namens *var_PrintListe* wird im Implementation-Teil angelegt und entzieht sich somit dem Zugriff von außen. Bei jedem Zugriff wird nun geprüft, ob eine Instanz vorhanden ist, gegebenenfalls wird diese erzeugt. Anschließend wird der Zeiger darauf als Funktionsergebnis zurückgegeben.

Die hier gezeigte Vorgehensweise hat den Vorteil, dass keine Änderung in anderen Formularen erforderlich ist und dass nie ein Zugriff erfolgen kann, bevor die Instanz erzeugt ist – eine Zugriffsverletzung ist demnach ausgeschlossen.

19 To-Do-Liste mit IBX

Nun wollen wir unser Projekt umstellen auf die Komponenten aus der Palette *InterBase*. Dazu kopieren wir das komplette bisherige Projekt in ein neues Verzeichnis.

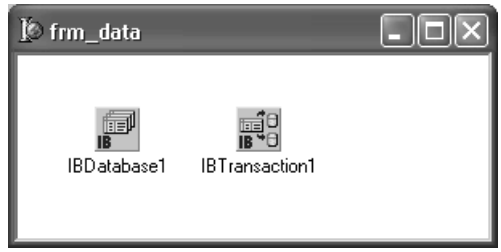
Die Aufspaltung unseres Projektes in mehrere Teilprojekte kommt uns hier in der Hinsicht entgegen, dass wir Teilprojekt für Teilprojekt umstellen können und so schneller einen Stand haben, der sich kompilieren lässt.

19.1 Die Verwaltung

Wir wollen auch hier wieder mit der Verwaltung anfangen. Öffnen Sie das Projekt *p_testverw.exe* und dort zunächst das Datenmodul.

19.1.1 Das Datenmodul

Die BDE-Komponente *TDatebase* hat eine Entsprechung bei den InterBase-Komponenten, nämlich *TIBDatabase*. Diese realisiert zwar den Zugriff auf die Datenbank, nicht aber die Transaktionsverwaltung. Dafür gibt es eine weitere Komponente, nämlich *TIBTransaction*.



Dementsprechend ist die Methode *BeginTransaction* umzustellen:

```
procedure Tfrm_data.BeginTransaction;
begin
  if IBTransaction1.InTransaction
    then IBTransaction1.Commit;
  IBTransaction1.StartTransaction;
end;
```

Des Weiteren können die Routinen von *RefreshQuery* und *UpdateRefreshQuery* entfernt werden, sie werden nicht mehr benötigt.

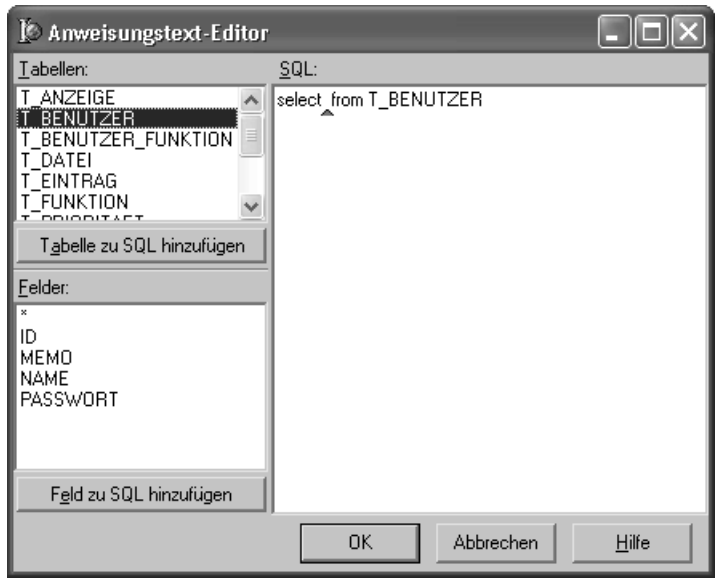
19.1.2 Die Benutzerverwaltung

Auf der Palettenseite *InterBase* gibt es einige Komponenten, die von ihrer Funktionalität an die »BDE-Kollegen« angelehnt sind, beispielsweise *TIBTable* oder *TIBQuery*.

Wir könnten nun der Versuchung nachgeben, alles auf *TIBQuery* umzustellen, wofür in vielen Fällen die Suchen- und Ersetzen-Funktion reichen würde. *TIBQuery* hat jedoch einen kleinen, aber sehr entscheidenden Nachteil: Die Eigenschaft *RequestLive* gibt es dort nicht. Nun könnte man entsprechende Datenmengen auf *TTable* umstellen, dort macht uns jedoch Probleme, da wir für die Spalte ID einen Wert benötigen.

Natürlich sind das alles Probleme, die man – mit entsprechendem Aufwand – schon irgendwie »hinbiegen« kann. Deutlich einfacher geht es, wenn wir gleich alles auf *TIBDataSet* umstellen. Entfernen wir also *qrBenutzer* und fügen stattdessen eine *TIBDataSet*-Komponente ein, die wir *dataBenutzer* nennen. Für die Eigenschaft *Database* wählen wir *frm_data.IBDatabase1*, dabei wird automatisch *frm_data.IBTransaction1* der Eigenschaft *Transaction* zugewiesen.

Öffnen wir den Eigenschaftseditor für die Eigenschaft *SelectSQL*:



Nach einem Doppelklick auf *T_BENUTZER* wird der Rumpf eines SELECT-Statements generiert, das wir durch einen Doppelklick auf den Stern vervollständigen können. Geben wir dann noch manuell eine ORDER-Klausel ein:

```
select *
  from T_BENUTZER
  order by ID
```

Nun rufen wir über das Kontextmenü den Dataset-Editor auf (und sehen gnädig über die etwas verunglückte Optik des Formulars hinweg ...).

Das Schlüsselfeld wird die ID, die anderen Felder werden Aktualisierungsfelder. Mit dem Button *SQL generieren* erzeugen wir dann die entsprechenden Statements.

```
update T_BENUTZER
set
  NAME = :NAME,
  PASSWORT = :PASSWORT,
  MEMO = :MEMO
where
  ID = :OLD_ID

insert into T_BENUTZER
  (NAME, PASSWORT, MEMO)
values
  (:NAME, :PASSWORT, :MEMO)
```

```
delete from T_BENUTZER
where
  ID = :OLD_ID
```

Diese Statements sehen so aus, als wenn wir *TUpdateSQL* verwenden würden. Neu ist die Anweisung für *RefreshSQL*:

```
Select
  ID,
  NAME,
  PASSWORT,
  MEMO
from T_BENUTZER
where
  ID = :ID
```

Bei der BDE-Variante haben wir im Refresh-Fall die Datenmenge geschlossen und neu geöffnet, der Server hat somit die Abfrage komplett neu aufgebaut. Hier können wir nun die Abfrage auf den einen Datensatz beschränken, der geändert oder neu eingefügt wurde.

Legen wir nun noch persistente *TField*-Eigenschaften an. Dann hängen wir die Ereignisbehandlungsroutinen an *dataBenutzer* an, für die Änderung der Bezeichner verwenden wir die Suchen- und Ersetzen-Funktion.

```
procedure Tfrm_benutzer.dataBenutzer_AfterPost(DataSet: TDataSet);
// für AfterPost und After Delete
begin
  //  dataBenutzer.Refresh;
  gvChanged := true;
end;
```

In der Methode *dataBenutzer_AfterPost* benötigen wir noch nicht einmal den Aufruf von *Refresh* (auch die Eigenschaft *ForceRefresh* kann den Wert *false* behalten), denn die Komponente erkennt selbst, dass die Ausführung von *RefreshSQL* erforderlich ist. (Entfernen Sie mal vorübergehend die WHERE-Klausel aus *RefreshSQL* und betrachten Sie, wie dann aktualisiert wird.)

Würden wir nun beim derzeitigen Stand versuchen, neue Benutzer einzutragen, dann beschwert sich die Komponente über eine fehlende ID – auch wenn diese vom INSERT-Statement gar nicht verwendet wird. Anhand des Ergebnisses der Abfrage wird hier eine NOT NULL-Spalte erkannt und somit muss diese mit einem Wert gefüllt werden. Erfreulicherweise geht das ohne größeren Aufwand. Wir öffnen den Eigenschaftseditor von *GeneratorField*:

Unser Generator lautet *G_BENUTZER* und er besorgt das Feld für die Spalte *ID*. Einen neuen Generatorwert brauchen wir für jeden neuen Datensatz.

Nun haben wir allerdings ein Problem mit unserem INSERT-Statement: Da hier die ID nicht mit eingefügt wird, würden für jeden neuen Datensatz nun zwei IDs besorgt – die eine von *TIBDataSet*, die andere vom Trigger der Tabelle. Um dies zu vermeiden, müssen wir das INSERT-Statement um die ID ergänzen:

```
insert into T_BENUTZER
  (ID, NAME, PASSWORT, MEMO)
values
  (:ID, :NAME, :PASSWORT, :MEMO)
```

Die Benutzerrechte

Auch für die Benutzerrechte setzen wir wieder *TIBDataSet* ein, in diesem Fall ersetzt diese Komponente sowohl *qrBenFunk* als auch *upBenFunk*, aus denen vorher noch die benötigten SQL-Statements entnommen werden.

Für RefreshSQL formulieren wir:

```
SELECT
    m.benutzer,
    m.funktion,
    f.name
  FROM t_benutzer_funktion m
    INNER JOIN t_funktion f
      ON f.id = m.funktion
  WHERE m.benutzer = :benutzer
  AND m.funktion = :funktion
```

Verwenden Sie statt *:benutzer* nicht *:ID*, sonst werden die Funktionsnamen nicht angezeigt (aus welchen Gründen auch immer).

```
procedure Tfrm_benutzer.dataBenFunkAfterPost(DataSet: TDataSet);
// für AfterPost und AfterQuery
begin
  gvChanged := true;
end;
```

Eine explizite Aktualisierung nach *Post* oder *Delete* kann unterbleiben, lediglich *gvChanged* wird auf *true* gesetzt.

```
procedure Tfrm_benutzer.btnInsertClick(Sender: TObject);

  ...

  procedure FunktionEintragen(Funktion: integer);
  begin
    if not dataBenFunk.Locate('Funktion',
      Variant(funktion), []) then
    begin
      dataBenFunk.Insert;
      dataBenFunkBENUTZER.AsInteger := dataBenutzerID.AsInteger;
      dataBenFunkFUNKTION.AsInteger := funktion;
      dataBenFunkNAME.AsString := ' '; // dreifacher Kotau
      // vor der Vollständigkeitsprüfung
      dataBenFunk.Post;
    end;
  end;

begin
  ...
end; {procedure Tfrm_benutzer.btnInsertClick}
```

Zwei Änderungen sind in der Methode *btnInsertClick* zu finden: Zum einen brauchen wir weder *ApplyUpdates* noch *UpdateRefreshQuery*. Zum anderen müssen wir einen Wert in die Spalte *Name* eintragen, auch wenn diese vom INSERT-Statement gar nicht berücksichtigt wird. Die Komponente legt für eine NOT NULL-Spalte eine *TField*-Instanz an, bei der *Required* den Wert *true* hat. Entweder setzt man *Required* auf *false*, oder man weist irgendeinen Wert zu, der dann keine Berücksichtigung findet.

Die Verwaltung der Funktionen

Die Verwaltung der Funktionen wird in der gleichen Art und Weise umgestellt wie die Verwaltung der Benutzer.

19.1.3 Die Verwaltung der Projekte

Die Komponente *qrProjekt* wird ersetzt durch *dataProjekt*, das SQL-Statement wird in die Eigenschaft *SelectSQL* übernommen, die anderen Anweisungen werden von der Komponente selbst (im Kontextmenü: DATASET-EDITOR) generiert. Auch hier muss die Eigenschaft *GeneratorField* entsprechend gesetzt werden.

Die Anweisungen zum Aufbau des Baumes bleiben dieselben. Allerdings gibt es bei der Anweisung *Locate* einen gewichtigen Unterschied: Während bei *TQuery* der gesuchte Datensatz direkt angesprungen wird und somit das Ereignis *AfterScroll* nur ein einziges Mal auftritt, wird bei *TIBDataSet* so lange durch die Datenmenge gescrollt, bis der gesuchte Datensatz erreicht ist. Dabei wird entsprechend oft das Ereignis *AfterScroll* ausgelöst, was die dazugehörenden Aktualisierungsauswirkungen auf dem Bildschirm nach sich zieht.

Um dieses »Geflacker« zu vermeiden, wird vor der Ausführung von *Locate* die *AfterScroll*-Ereignisbehandlungsroutine »abgeklemmt« und hinterher wieder zugewiesen. Anschließend wird sie einmal manuell aufgerufen, damit die entsprechenden Aktualisierungen ausgeführt werden.

```
procedure Tfrm_projekte.TreeView1Change(Sender: TObject;
  Node: TTreeNode);
begin
  try
    dataProjekt.AfterScroll := nil;
    if Assigned(Node)
      then dataProjekt.Locate('ID', Variant(integer(Node.Data)), []);
  finally
    dataProjekt.AfterScroll := dataProjekt_AfterScroll;
    dataProjekt_AfterScroll(dataProjekt);
  end;
end;
```

Des Weiteren würde beim Einfügen neuer Projekte versucht, schon vor dem *Post* die Detaildatenmenge *qrPriosAll* zu aktualisieren. Da das neue Projekt – dank Generator – schon eine ID hat, aber noch keinen Master, würde die abgefragte STORED PROCEDURE in eine Endlosschleife laufen. Folglich wird *qrPriosAll* im Ereignis *BeforeInsert* deaktiviert:

```
procedure Tfrm_projekte.dataProjektBeforeInsert(DataSet: TDataSet);
begin
  qrPriosAll.Close;
end;
```

Des Weiteren – wer hätte es gedacht – muss diese Datenmengenkomponente im Ereignis *AfterInsert* wieder aktiviert werden:

```
procedure Tfrm_projekte.dataProjekt_AfterInsert(DataSet: TDataSet);
var
  Form: Tfrm_projektbearbeiten;
begin
  Form := Tfrm_projektbearbeiten.Create(nil);
  try
    ...
  finally
    Form.Free;
    qrPriosAll.Open;
  end;
end; {procedure Tfrm_projekte.dataProjektAfterInsert}
```

Die Rechte

Der Einfachheit halber wird *qrBenRechte* durch eine *TIBQuery*-Komponente ersetzt, die den gleichen Bezeichner haben soll. Kopieren Sie dazu erst die SQL-Anweisung, dann den Namen der Komponente. Bevor Sie den Namen der *TIBQuery*-Instanz zuweisen, müssen Sie die alte Komponente löschen, sonst »meckert« das System wegen doppelter Bezeichner. Vergessen Sie dann nicht, persistente *TField*-Instanzen anzulegen. Da sich der Bezeichner der Komponente nicht geändert hat, ist eine Anpassung des Quelltextes nicht erforderlich.

Die Komponente *qrRecht* wird dann wieder durch *TIBDataSet* ersetzt, *upRecht* wird dadurch wieder überflüssig. Die bisherige SQL-Anweisung wird wieder in die Eigenschaft *SelectSQL* übernommen, die anderen SQL-Anweisungen werden zunächst automatisch generiert.

In *ModifySQL* wird dann die Aktualisierung von Projekt und Funktion unterbunden:

```
update t_recht
set
  ERSTELLEN = :ERSTELLEN,
  ZUWEISEN = :ZUWEISEN,
  BEARBEITEN = :BEARBEITEN,
```

```
  FERTIGSTELLEN = :FERTIGSTELLEN,
  LESEN = :LESEN,
  INFORMIEREN = :INFORMIEREN,
  LEGITIMIEREN = :LEGITIMIEREN,
  UNTERPROJEKTE = :UNTERPROJEKTE
where
  PROJEKT = :OLD_PROJEKT and
  FUNKTION = :OLD_FUNKTION
```

RefreshSQL wird dann von *SelectSQL* übernommen und in der WHERE-Klausel angepasst:

```
SELECT
    r.projekt,
    r.funktion,
    f.name,
    r.erstellen,
    r.zuweisen,
    r.bearbeiten,
    r.fertigstellen,
    r.lesen,
    r.informieren,
    r.legitimieren,
    r.unterprojekte
  FROM t_recht r
    INNER JOIN t_funktion f
      ON r.funktion = f.ID
  WHERE
  PROJEKT = :PROJEKT and
  FUNKTION = :FUNKTION
```

Im Formular *Tfrm_rechte* muss *Query1* ersetzt werden. Aus Gründen der Arbeitserleichterung nennen wir die neue Komponente wieder *Query1*, somit muss kein Quelltext geändert werden.

Die Prioritäten

Die Komponente *dataPrios* ersetzt *qrPrios* und übernimmt deren SELECT-Statement; die anderen Anweisungen können automatisch generiert werden. Vergessen Sie nicht, *GeneratorField* zu setzen.

Die Komponente *qrPriosAll* wird lediglich durch eine *TIBQuery* ersetzt, die SQL-Anweisung kann beibehalten werden.

19.1.4 Der Passwortdialog

Auch im Passwortdialog gibt es noch zwei *TQuery*-Komponenten, die durch *TIBQuery* zu ersetzen sind. Auch hier behalten wir die Bezeichner (*qrBenutzer* und *qrBenFunk*) bei.

Nun sollte sich das Projekt kompilieren lassen.

19.2 Die Einträge

Die Formulare von *p_testEin.dpr* verzichten auf *TUpdateSQL* und *RequestLive*, alle Datenänderungen werden mit INSERT- und UPDATE-Statements sowie *ExecSQL* ausgeführt. Somit sollte man meinen, dass es ausreichen würde, alle *TQuery*-Instanzen nach *TIBQuery* umzusetzen.

Nun gut, das Projekt lässt sich nach einer solchen Umsetzung kompilieren, zur Laufzeit treten dann jedoch einige Exceptions auf – der Teufel liegt also mal wieder im Detail.

StartTransaction

Beim Start einer Transaktion werden alle Datenmengen geschlossen und müssen dann neu geöffnet werden – oder man öffnet sie gleich erst danach. Schauen wir uns noch mal die Methode *ModalAnzeigen* an:

```
procedure Tfrm_data.ModalAnzeigen(FormClass: TComponentClass;
  StartTransaction: boolean; FormTag: integer = 0);
var
  Form: TForm;
begin
  Form := FormClass.Create(nil) as TForm;
  try
        ...
      BeginTransaction;
      if Form.ShowModal = mrOK then
      begin
        try
          IBTransaction1.Commit;
        except
          IBTransaction1.Rollback;
            ...
end; {procedure ModalAnzeigen}
```

Zunächst wird ein Formular erzeugt, dann wird eine Transaktion gestartet, anschließend wird das Formular modal angezeigt. Wenn wir also – beispielsweise in *frm_NeuEintrag* – die Datenmengenkomponenten in der *FormCreate*-Routine öffnen, dann werden sie mit dem Start der Transaktion wieder geschlossen. Verlegen wir jedoch diese Anweisung in die Routine *FormShow*, dann treten diesbezüglich keine Probleme auf.

Dieses Problem tritt auch dann auf, wenn in *frm_eintragDetail* eine Statusänderung vorgenommen wird, eine Information hinzugefügt oder ein Eintrag zugewiesen wird. Nach dem Aufruf von *BeginTransaction* wird (unter anderem) *qrEintrag* geschlossen. Wird danach erst *WerteEintragen* aufgerufen, dann wird dem Parameter *Eintrag* die Zahl Null zugewiesen, was natürlich zu einer Fremdschlüsselverletzung führt.

```
procedure Tfrm_EintragDetail.btnStatusClick(Sender: TObject);
var
  Form: Tfrm_EditStatus;

  procedure WerteEintragen;
  begin
    with qrAddStatus do
    begin
      ParamByName('Eintrag').AsInteger
        := qrEintragID.AsInteger;
      // ExecSQL
    end; {with qrAddStatus do}
  end; {procedure WerteEintragen}

begin
  Form := Tfrm_EditStatus.Create(nil);
  try
    if Form.ShowModal = mrOK then
    begin
      WerteEintragen;
      frm_data.BeginTransaction;
      try
        qrAddStatus.ExecSQL;
        frm_data.IBTransaction1.Commit;
      except
        frm_data.IBTransaction1.Rollback;
      end;
    end; {if Form.ShowModal = mrOK then}
  finally
```

```
    Form.Free;
  end;
  RefreshAll;
end; {procedure Tfrm_EintragDetail.btnStatusClick}
```

Deshalb wird nun *WerteEintragen* vor *BeginTransaction* aufgerufen. Dies bedingt allerdings, dass *ExecSQL* nicht mehr von *WerteEintragen* aufgerufen wird, sondern dass diese Anweisung nach *BeginTransaction* ausgeführt wird.

Zudem müssen nun bei allen Änderungen die Datenmengen *qrEintrag*, *qrStatus* und *qrDatei* neu geöffnet werden, was zur Anweisung *RefreshAll* zusammengefasst wird.

```
procedure Tfrm_EintragDetail.RefreshAll;
begin
  RefreshQuery(qrEintrag);
  RefreshQuery(qrStatus);
  RefreshQuery(qrDatei);
end;
```

In der gleichen Art und Weise wie *btnStatusClick* werden auch die Routinen *btnInfoClick* und *btnZuweisClick* geändert.

Zugriff auf einen Generator

Im Formular *frm_NeuEintrag* hatten wir bislang über eine SELECT-Anweisung aus der STORED PROCEDURE *p_eintrag* einen neuen Generatorwert geholt. Zur Erinnerung nochmals diese lokale Routine:

```
  procedure IdHolen;
  begin
    qrWork.SQL.Text := 'SELECT * FROM p_eintrag_ID';
    qrWork.Open;
    iEin := qrWork.FieldByName('GEN').AsInteger;
  end;
```

Wird dies nun so nach *TIBQuery* umgesetzt, dann tritt an dieser Stelle eine Exception auf, weil InterBase den Query-Plan nicht findet. Warum das nun genau scheitert, habe ich nicht herausgefunden, zumal diese Anweisung zur Entwurfszeit problemlos arbeitet.

Das Problem lässt sich aber beheben, indem man die Komponente *TIBStoredProc* verwendet. Hier setzen wir neben *Database* und *Name* (*spID*) auch die Eigenschaft *StoredProcName* (*p_eintrag_id*). Die lokale Routine *IdHolen* wird nun wie folgt abgeändert:

```
procedure IdHolen;
begin
  spID.ExecProc;
  iEin := spID.Params[0].Value;
end;
```

19.2.1 Verwendung von TIBSQL

Die Komponente *TIBQuery* verbraucht mehr Ressourcen, als für manche Aufgabe unbedingt erforderlich ist. Werden beispielsweise die Daten einer Abfrage nur einmal benötigt, um sie sequenziell auszulesen und damit beispielsweise eine ComboBox zu füllen, dann ist es nicht nötig, die Daten in der Komponente zu speichern.

Mit *TIBSQL* wird eine vergleichsweise minimalistische Komponente zur Verfügung gestellt, welche jedoch für manche Aufgabe völlig ausreicht. Wir wollen nun an zwei Stellen – in einem SELECT-Statement und in einem INSERT-Statement – *TQuery* durch *TIBSQL* ersetzen.

sqlFunktionen

Zunächst ersetzen wir *qrFunktionen* durch *sqlFunktionen*, neben *Database* und *Name* muss die Eigenschaft *SQL* gesetzt werden:

```
SELECT id, name
  FROM t_funktion
  ORDER BY ID
```

Beim Programmstart werden nun die Daten in *cmbFunktion* geschrieben.

```
with sqlFunktion do
begin
  ExecQuery;
  while not EOF do
  begin
    cmbFunktion.Items.AddObject(FieldByName('Name').AsString,
      Pointer(FieldByName('ID').AsInteger));
    Next;
  end;
end; {with qrFunktion do}
```

Da hier die Daten ohnehin sequenziell ausgelesen werden, stört es überhaupt nicht, dass *TIBSQL* keine andere Browse-Methode außer *Next* zur Verfügung stellt.

sqlFiles

Zum Schreiben neuer Dateien in die Datenbank wird die folgende Anweisung verwendet:

```
INSERT INTO t_datei (EINTRAG, NAME, BENUTZER, MEMO, DATEI)
  VALUES (:EINTRAG, :NAME, :BENUTZER, :MEMO, :DATEI)
```

Der Text des Memos wird statt mit *Assign* mit *AsString* zugewiesen, die Datei selbst wird dann wieder mit *LoadFromFile* eingelesen, wobei hier als Parameter nur der Dateiname erwartet wird. Statt mit *ExecSQL* wird die Anweisung mit *ExecQuery* ausgeführt.

```
  procedure DoFiles;
  var
    i: integer;
    s: string;
  begin
    for i := 0 to lbFiles.Items.Count - 1 do
    begin
      with sqlFiles do
      begin
        s := lbFiles.Items[i];
        ParamByName('Eintrag').AsInteger := iEin;
        ParamByName('Name').AsString := GetFileName(s);
        ParamByName('Benutzer').AsInteger := gvUserID;
        ParamByName('Memo').AsString
          := (lbFiles.Items.Objects[i] as TStrings).Text;
        ParamByName('Datei').LoadFromFile(s);
        ExecQuery;
      end; {with qrFiles do}
    end; {for i := 0 to lbFiles.Items.Count - 1 do}
  end; {procedure DoFiles}
```

19.3 Die Anzeige

Im Projekt *p_testAnz* muss lediglich das Formular *frm_abfrage* angepasst werden. Die Komponenten *qrAnzeige* und *upAnzeige* werden durch *dataAnzeige* ersetzt. Die SQL-Anweisung aus *qrAnzeige* wird der Eigenschaft *SelectSQL* zugewiesen, die Anweisungen für Insert, Update und Delete werden automatisch generiert.

Das Statement für *RefreshSQL* entnehmen wir *SelectSQL* und passen die WHERE-Klausel entsprechend an:

```
SELECT
    a.id,
    a.name,
    a.anweisung,
    a.benutzer,
    b.name AS Benutzername
  FROM t_anzeige a
    LEFT OUTER JOIN t_benutzer b
      ON a.benutzer = b.id
  WHERE a.ID = :ID
```

Vergessen Sie nicht, die Eigenschaft *GeneratorField* entsprechend zu setzen.

Für den Assistenten hatten wir bislang das INSERT-Statement der *TUpdateSQL*-Komponente *upAnzeige* verwendet, *TIBDataSet* veröffentlich jedoch *QInsert* nicht, so dass wir diesen Weg nicht gehen können. Deshalb fügen wir die *TIBSQL*-Komponente *sqlAssi* ein, der wir folgendes INSERT-Statement zuweisen:

```
insert into t_anzeige
  (NAME, ANWEISUNG, BENUTZER)
values
  (:NAME, :ANWEISUNG, :BENUTZER)
```

Darüber hinaus muss auch die Routine *btnAssiClick* ein wenig angepasst werden:

```
procedure Tfrm_abfrage.btnAssiClick(Sender: TObject);
var
  Form: Tfrm_AssiAbfrage;
begin
  Form := Tfrm_AssiAbfrage.Create(nil);
  try
    if Form.ShowModal = mrOK then
    begin
      with sqlAssi do
      begin
        ParamByName('Name').AsString := Form.edtName.Text;
        ParamByName('Anweisung').AsString := Form.GetText;
        ParamByName('Benutzer').AsInteger := gvUserID;
        ExecQuery;
      end; {with upAnzeige.Query[ukInsert] do}
```

```
      dataAnzeige.Close;
      dataAnzeige.Open;
    end; {if Form.ShowModal = mrOK then}
  finally
    Form.Free;
  end;
end; {procedure Tfrm_abfrage.btnAssiClick}
```

Die Zuweisung der Anweisung kann nun über *AsString* erfolgen, ein Umweg über eine String-Liste ist hier nicht erforderlich. Statt *ExecSQL* muss mal wieder *ExecQuery* eingesetzt werden und die Datenmenge *dataAnzeige* muss aktualisiert werden – da die Änderung nicht den aktuellen Datensatz betrifft, hilft hier *Refresh* nicht weiter.

19.4 Das Gesamtprojekt

Im Formular *frm_main* muss zunächst einmal die *TQuery*-Komponente durch eine *TIBQuery*-Komponente ersetzt werden. Da wir diese wieder *Query1* nennen, muss hier kein Quelltext angepasst werden.

Das Transaktionsproblem

Allerdings haben wir wieder das Problem, dass nach einem *StartTransaction Query1* geschlossen wird. Leider implementiert *TIBTransaction* zwar ein Ereignis *OnIdleTimer*, nicht aber *OnStartTransaction*. Jedoch macht uns hier nur das *StartTransaction* Probleme, das über *ModalAnzeigen* aufgerufen wird, und das bekommt man auch so noch hin:

```
type
  Tfrm_data = class(TDataModule)
    ...
  private
    Fsl: TStringList;
    FAfterTransaction: TNotifyEvent;
    procedure DoAfterTransaction;
  public
    procedure ModalAnzeigen(FormClass: TComponentClass;
      StartTransaction: boolean; FormTag: integer = 0);
    procedure BeginTransaction;
    property AfterTransaction: TNotifyEvent
      read FAfterTransaction write FAfterTransaction;
  end;
```

Zunächst wird das Datenmodul über ein Ereignis *AfterTransaction* erweitert, der Typ *TNotifyEvent* reicht hier völlig.

```
procedure Tfrm_data.DoAfterTransaction;
begin
  if Assigned(FAfterTransaction)
    then FAfterTransaction(IBTransaction1);
end;
```

Mit der Prozedur *DoAfterTransaction* wird dieses Ereignis ausgelöst.

```
procedure Tfrm_main.FormShow(Sender: TObject);
begin
    ...
  frm_data.AfterTransaction := DataAfterTransaction;
end; {procedure Tfrm_main.FormShow}
```

Im Hauptformular wird diesem Ereignis die Methode *DataAfterTransaction* als Ereignisbehandlungsroutine zugewiesen.

```
procedure Tfrm_main.DataAfterTransaction(Sender: TObject);
begin
  Query1.Close;
  Query1.Open;
  SpaltenUeberschriften;
end;
```

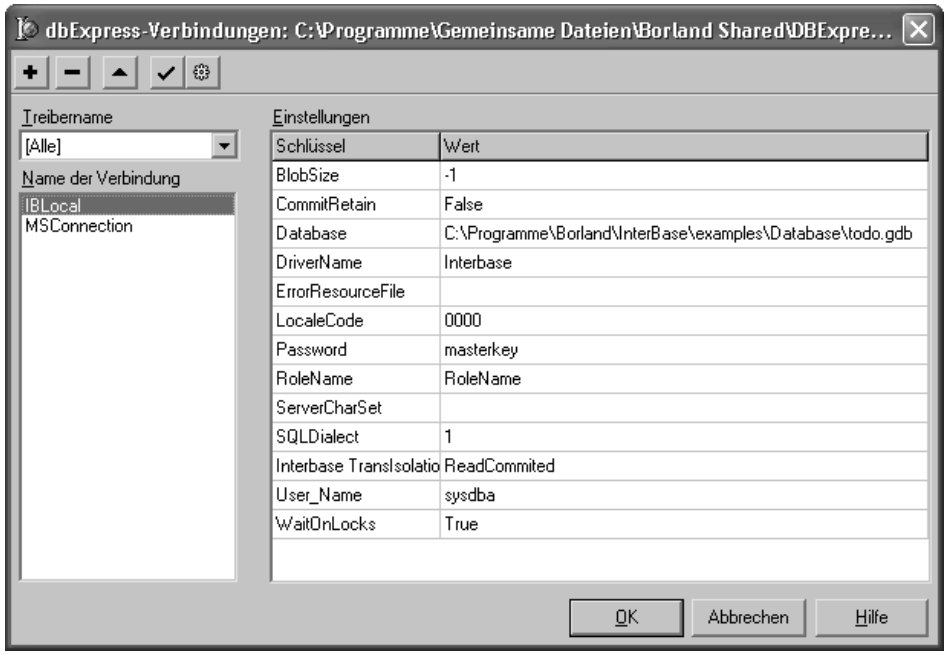
In *DataAfterTransaction* wird dann *Query1* neu geöffnet, anschließend werden wieder die Spaltenüberschriften »gerade gezogen«.

20 To-Do-Liste mit dbExpress

Zuletzt wollen wir das Projekt auch noch nach *dbExpress* umstellen. Der »schlanke« Zugriff durch diese Komponenten hat zur Folge, dass wir einige Dinge ganz anders lösen müssen – aber sehen Sie selbst.

20.1 Das Datenmodul

Anstelle einer *TDatabase*- benötigen wir nun eine *TSQLConnection*-Instanz. Mit einem Doppelklick auf diese Komponente öffnen wir den Verbindungsassistenten:



Der Assistent ist leider etwas »spartanischer« geraten als nötig, ein Dateiauswahldialog für *Database* fehlt, aber vielleicht bessert Borland ja in der nächsten Version nach. *Password* und *User_Name* dürften schon automatisch auf die korrekten Werte gesetzt sein.

Die Routine *RefreshQuery* stellen wir ganz pragmatisch von *TQuery* auf *TDataSet* um, somit wird sie auch mit den jetzt verwendeten Datenzugriffskomponenten funktionieren.

```
procedure RefreshQuery(Query: TDataSet);
begin
  Query.Close;
  Query.Open;
end;
```

Die Transaktionsverwaltung

TSQLConnection implementiert keine Eigenschaft zum Isolationsmodus, stattdessen ist beim Start der Transaktion ein Record zu übergeben, bei dem die ID der Transaktion und der Isolationsmodus gesetzt sind.

```
procedure Tfrm_data.DataModuleCreate(Sender: TObject);
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
end;
```

Das Feld *TransactionID* kann willkürlich vergeben werden, der Wert bei *Commit* oder *Rollback* muss lediglich derselbe sein. Als *IsolationLevel* verwenden wir – wie auch in den anderen beiden Versionen – *Read Commited*.

```
procedure Tfrm_data.BeginTransaction;
begin
  if SQLConnection1.InTransaction
    then SQLConnection1.Commit(TD);
  SQLConnection1.StartTransaction(TD);
end;
```

Der Record *TD* wird dann den für die Transaktionen zuständigen Methoden als Parameter übergeben.

```
procedure Tfrm_data.ModalAnzeigen(FormClass: TComponentClass;
  StartTransaction: boolean; FormTag: integer = 0);
var
  Form: TForm;
begin
  Form := FormClass.Create(nil) as TForm;
  try
    Form.Tag := FormTag;
```

```
    if StartTransaction then
    begin
      BeginTransaction;
      if Form.ShowModal = mrOK then
      begin
        try
          SQLConnection1.Commit(TD);
        except
          SQLConnection1.Rollback(TD);
          ShowMessage
            ('Die Änderungen konnten nicht geschrieben werden!');
        end;
      end {if Form.ShowModal = mrOK then}
      else SQLConnection1.Rollback(TD);
      gvChanged := false;
    end {if StartTransaction then}
    else Form.ShowModal;
  finally
    Form.Free;
  end;
end; {procedure ModalAnzeigen}
```

Die Zusammenfassung zahlreicher Formularaufrufe in *ModalAnzeigen* nimmt uns hier viel Arbeit ab.

20.2 Die Verwaltung

Und auch hier wollen wir die Umstellung wieder mit der Verwaltung beginnen, und zwar mit der Benutzerverwaltung.

20.2.1 Die Benutzerverwaltung

Die Komponente *qrBenutzer* ersetzen wir durch eine *TSQLClientDataSet*-Instanz namens *dataBenutzer*. Die Eigenschaft *DBConnection* erhält den Wert *frm_data.SQLConnection1* und für *CommandText* verwenden wir die folgende Anweisung:

```
SELECT *
  FROM t_benutzer
  ORDER BY id
```

Diese Datenmenge ist aktualisierungsfähig, es muss lediglich dafür gesorgt werden, dass die Änderungen auch auf den Server übertragen werden:

```
procedure Tfrm_benutzer.dataBenutzerAfterPost(DataSet: TDataSet);
// für AfterPost und After Delete
begin
  dataBenutzer.ApplyUpdates(0);
  gvChanged := true;
end;
```

Allerdings benötigen wir im Insert-Fall einen Wert für die Spalte ID und so eine praktische Eigenschaft wie *GeneratorField* gibt es bei den *dbExpress*-Komponenten leider nicht.

```
procedure Tfrm_benutzer.dataBenutzerAfterInsert(DataSet: TDataSet);
// für AfterInsert und AfterEdit
var
  Form: Tfrm_benutzereingabe;
begin
  Form := Tfrm_benutzereingabe.Create(nil);
  try
    if dataBenutzer.State = dsEdit then
    begin
      Form.Edit1.Text
        := Tfrm_passwort.GetPasswort(dataBenutzerPASSWORT);
      Form.Edit2.Text := Form.Edit1.Text;
    end;
    if dataBenutzer.State = dsInsert then
    begin
      sqlBenutzerID.Open;
      dataBenutzerID.AsInteger := sqlBenutzerIDGEN.AsInteger;
      sqlBenutzerID.Close;
    end;
    Form.SetWerte(true, dsBenutzer);

    if Form.ShowModal = mrOK then
    begin
      Tfrm_passwort.
        SetPasswort(dataBenutzerPASSWORT, Form.Edit1.Text);
      dataBenutzer.Post;
    end
    else dataBenutzer.Cancel;
```

```
  finally
    Form.Free;
  end;
end; {procedure Tfrm_benutzer.dataBenutzerAfterInsert}
```

Den Generatorwert ermittelt uns die STORED PROCEDURE *p_benutzer_id*, für den Zugriff darauf wird die *TSQLQuery*-Komponente *sqlBenutzerID* benutzt, welche das folgende Statement verwendet:

```
SELECT *
  FROM p_benutzer_id
```

In diesem Zusammenhang noch eine Bemerkung: Der Gedanke, dass mit der Komponente *TSQLStoredProc* der Zugriff auf eine STORED PROCEDURE besonder effektiv ist, ist nahe liegend, aber zumindest hier falsch. Im Fall des Zugriffs mit *TSQLQuery* oder *TSQLDataSet* werden, das lässt sich über den Monitor nachvollziehen, deutlich weniger Server-Zugriffe erforderlich, insbesondere dann, wenn persistente *TField*-Instanzen verwendet werden.

```
procedure Tfrm_benutzer.dataBenutzerAfterScroll(DataSet: TDataSet);
begin
  if not gvIsAdmin
    then btnEdit.Visible
      := gvUserID = dataBenutzerID.AsInteger;
  with dataBenFunk do
  begin
    Close;
    Params[0].AsInteger := dataBenutzerID.AsInteger;
    Open;
  end;
end;
```

Die Master-Detail-Verknüpfung mit *dataBenFunk* müssen wir manuell herstellen, deshalb ist die Methode *dataBenutzerAfterScroll* entsprechend zu erweitern.

Die Liste der Funktionen

Ist der Umstieg von *qrBenutzer* zu *dataBenutzer* noch vergleichsweise harmlos gewesen, so stehen nun größere »Umbauten« an. Für den Datenzugriff verwenden wir wieder *TSQLClientDataSet*. Es kann dabei auch die gleiche SQL-Anweisung verwendet werden, doch für die Aktualisierung der Daten steht uns kein »bequemer« Weg zur Verfügung.

Deshalb verwenden wir eine *TSQLQuery*-Komponente namens *sqlWork* und führen die Datenänderungen mit INSERT- und DELETE-Statements durch.

```
procedure Tfrm_benutzer.btnDeleteClick(Sender: TObject);
// gemeinsam für btnDelete und btnClear
var
  s: string;
begin
  if Sender = btnDelete
    then s := 'Soll dem Benutzer "' +  dataBenutzerNAME.AsString
      + '"' + #13#10  + 'die Funktion "' + dataBenFunkNAME.AsString
      + '" entzogen werden?'
    else s := 'Sollen dem Benutzer "' +  dataBenutzerNAME.AsString
      + '" alle Funktionen entzogen werden?';
  if MessageDlg(s, mtConfirmation, [mbYes, mbNo], 0) = mrYes then
  begin
    with SQLWork do
    begin
      SQL.Text := 'DELETE FROM t_benutzer_funktion';
      SQL.Add('WHERE Benutzer = :benutzer');
      if Sender = btnDelete then
      begin
        SQL.Add('AND Funktion = :funktion');
        ParamByName('funktion').AsInteger
          := dataBenFunkFUNKTION.AsInteger;
      end;
      ParamByName('benutzer').AsInteger
        := dataBenutzerID.AsInteger;
      ExecSQL(false);
      RefreshQuery(dataBenFunk);
      gvChanged := true;
    end; {with SQLWork do}
  end; {if MessageDlg...}
end; {procedure Tfrm_benutzer.btnDeleteClick}
```

Die Komponenten *btnDelete* und *btnClear* verwenden wieder dieselbe Ereignisbehandlungsroutine, im Falle von *btnDelete* muss lediglich die WHERE-Klausel ein wenig enger gefasst und ein weiterer Parameter zugewiesen werden. Da der Zugriff an *dataBenFunk* vorbei geschieht, muss diese Datenmenge aktualisiert werden. Und da hier auch kein *AfterPost*-Ereignis ausgelöst wird, muss *gvChanged* in dieser Routine auf *true* gesetzt werden.

Im Prinzip dieselbe Vorgehensweise verwenden wir im Insert-Fall: SQL-Statement setzen, Parameter setzen, ausführen.

```
procedure Tfrm_benutzer.btnInsertClick(Sender: TObject);
var
  i: integer;
  Form: Tfrm_FunktionVergeben;

  procedure FunktionEintragen(Funktion: integer);
  begin
    if not dataBenFunk.Locate('Funktion', Variant(funktion), []) then
    begin
      sqlWork.ParamByName('Benutzer').AsInteger
        := dataBenutzerID.AsInteger;
      sqlWork.ParamByName('Funktion').AsInteger := funktion;
      sqlWork.ExecSQL(false);
    end;
  end;

begin
  Form := Tfrm_FunktionVergeben.Create(nil);
  try
    Form.SetWerte(true);
    if Form.ShowModal = mrOK then
    begin
      sqlWork.SQL.Text := 'INSERT INTO t_benutzer_funktion';
      sqlWork.SQL.Add('(benutzer, funktion)');
      sqlWork.SQL.Add('VALUES (:benutzer, :funktion)');
      with Form.DBGrid1, Form.SQLClientDataSet1 do
      begin
        for i := 0 to SelectedRows.Count - 1 do
        begin
          GotoBookmark(Pointer(SelectedRows.Items[i]));
          FunktionEintragen(FieldByName('ID').AsInteger);
        end; {for i := 0 to SelectedRows.Count - 1 do}
        RefreshQuery(dataBenFunk);
        gvChanged := true;
      end; {with frm_data, frm_FunktionVergeben.DBGrid1 do}
    end; {if ShowModal = mrOK then}
  finally
    Form.Free;
  end;
end; {procedure Tfrm_benutzer.btnInsertClick}
```

Auch das Formular *Tfrm_FunktionVergeben* bedarf einer kleinen Anpassung: Durch die Umstellung von *TQuery* auf *TSQLClientDataSet* gibt es keine Eigenschaft *SQL* vom Typ *TStrings*, sondern eine Eigenschaft *CommandText* vom Typ *string* – infolgedessen können wir nicht mehr so einfach eine Zeile aus dem Statement auswechseln. Deshalb wird nun jeweils die komplette Anweisung zugewiesen:

```
procedure Tfrm_FunktionVergeben.SetWerte(const Benutzer: boolean);
begin
  if Benutzer then
  begin
    Caption := 'Benutzer hinzufügen';
    DBGrid1.Columns[1].Title.Caption := 'Benutzer';
    SQLClientDataSet1.CommandText := 'SELECT * FROM t_funktion '
      + 'WHERE ID > 0 ORDER BY ID';
  end
  else
  begin
    Caption := 'Funktion vergeben';
    DBGrid1.Columns[1].Title.Caption := 'Funktion';
    SQLClientDataSet1.CommandText := 'SELECT * FROM t_benutzer '
      + 'ORDER BY ID';
  end; {if Benutzer then}
  SQLClientDataSet1.Open;
end; {procedure Tfrm_FunktionVergeben.SetWerte}
```

Die Verwaltung der Funktionen

Die Verwaltung der Funktionen ist so gut wie identisch mit der Verwaltung der Benutzer aufgebaut, so dass wir sie nicht eigens besprechen wollen.

20.2.2 Der Passwortdialog

Im Passwortdialog stellen wir von *TQuery* auf *TSQLQuery* um. Allerdings haben wir bei der BDE-Version ein wenig »gepfuscht«, nämlich alle Datensätze von *t_benutzer* geholt und den Satzzeiger dann mit *Locate* positioniert. *Locate* wird jedoch hier nicht mehr unterstützt.

```
SELECT *
  FROM t_benutzer
  WHERE name = :name
```

Machen wir es hier also »anständig« und filtern die Datenmenge mittels einer WHERE-Klausel.

```
function Tfrm_passwort.CheckPasswort(UserName,
  Passwort: string): boolean;
begin
  result := false;
  qrBenutzer.ParamByName('Name').AsString := UserName;
  qrBenutzer.Open;
  if qrBenutzerID.IsNull = false then
  begin
    if GetPasswort(qrBenutzerPASSWORT) = Passwort then
    begin
      gvUserID := qrBenutzerID.AsInteger;
      gvUserName := qrBenutzerNAME.AsString;
      qrBenFunk.Open;
      gvIsAdmin := (qrBenFunkFUNKTION.AsInteger = 1);
      result := true;
    end
    else ShowMessage('Passwort falsch');
  end {if qrBenutzerID.IsNull = false}
  else ShowMessage('Benutzer nicht bekannt');
  qrBenutzer.Close;
  qrBenFunk.Close;
end; {function Tfrm_passwort.CheckPasswort}
```

Jetzt müssen wir lediglich noch prüfen, ob *qrBenutzer* einen Datensatz oder eine leere Datenmenge als Ergebnis liefert. Das Einfachste wäre hier die Prüfung mittels *RecordCount*, die jedoch auch nicht unterstützt wird. Also prüfen wir ein Feld – es wäre egal, ob *ID* oder *Name* – mit *IsNull* auf das Vorliegen eines Datensatzes. Anschließend wird das Passwort geprüft und gegebenenfalls der Administratorstatus gesetzt.

20.2.3 Die Projektverwaltung

Die Anzeige der Projekte stellen wir von *TQuery* auf *TSQLClientDataSet* um. Die Änderungen werden mit *ApplyUpdates* zum Server übertragen:

```
procedure Tfrm_projekte.dataProjektAfterPost(DataSet: TDataSet);
// für AfterPost und AfterDelete
var
  i: integer;
begin
  i := FProjektID;
  dataProjekt.ApplyUpdates(0);
  gvChanged := true;
```

```
  RefreshTree(TreeView1);
  dataProjekt.Locate('ID', Variant(i), []);
end;
```

Die Master-Detail-Verknüpfungen werden per Quelltext erstellt. Da hier mehrere Detail-Datenmengen aktualisiert werden müssen, lagern wir den dazugehörenden Quelltext in eine lokale Prozedur aus:

```
procedure Tfrm_projekte.dataProjektAfterScroll(DataSet: TDataSet);

  procedure ProjektAktualisieren(Data: TSQLCLientDataSet);
  begin
    with Data do
    begin
      Close;
      Params[0].AsInteger := FProjektID;
      Open;
    end;
  end; {procedure ProjektAktualisieren}

begin
  if CanSync then
  begin
    FProjektID := dataProjektID.AsInteger;
    TreeView1.Selected
      := GetNodeByID(TreeView1, FProjektID);
    if dataRecht.Active
      then dataRecht.First;
    with qrBenRechte do
    begin
      Close;
      ParamByName('benutzer').AsInteger := gvUserID;
      ParamByName('projekt').AsInteger := FProjektID;
      Open;
    end; {with qrBenRechte do}
    ProjektAktualisieren(dataPriosAll);
    ProjektAktualisieren(dataPrios);
    ProjektAktualisieren(dataRecht);
    pnlFunktionen.Visible := qrBenRechteLESEN.AsString = 'J';
    RechteVergebenVisible(qrBenRechteLEGITIMIEREN.AsString = 'J');
    btnUnterprojekt.Visible := (not gvIsAdmin)
```

```
      and (qrBenRechteUNTERPROJEKTE.AsString = 'J');
  end; {if CanSync then}
end; {procedure Tfrm_projekte.dataProjektAfterScroll}
```

Wie gehabt bleibt die Aktualisierung der Baumansicht und die Geschichte mit den Rechten.

Hat ein Projekt Unterprojekte, dann sollte man es nicht löschen, sonst lässt sich der Baum nicht mehr korrekt aufbauen:

```
procedure Tfrm_projekte.dataProjektBeforeDelete(DataSet: TDataSet);
begin
  if TreeView1.Selected.HasChildren
    then raise Exception.Create('Projekt hat Unterprojekte'
      + ' und kann deshalb nicht gelöscht werden!');
end;
```

Die *AfterEdit*-Ereignisbehandlungsroutine bedarf keiner Änderung, für *AfterInsert* müssen wir uns jedoch vom Generator eine ID besorgen:

```
procedure Tfrm_projekte.dataProjektAfterInsert(DataSet: TDataSet);
var
  Form: Tfrm_projektbearbeiten;
begin
        ...
      with qrProjektID do
      begin
        Open;
        dataProjektID.AsInteger := qrProjektIDGEN.AsInteger;
        Close;
      end;
      dataProjekt.Post;
        ...
end; {procedure Tfrm_projekte.dataProjektAfterInsert}
```

Das Statement von *qrProjektID* lautet:

```
SELECT *
  FROM p_projekt_id
```

Für *qrBenRechte* wird *TSQLQuery* eingesetzt. Da hier nur ein einziger Datensatz angezeigt wird, können wir problemlos eine *DataSource* an diese Komponente hängen.

Die Verwaltung der Rechte

Etwas komplizierter wird es bei den Rechten, für welche die *TSQLClientDataSet*-Komponente *dataRechte* eingesetzt wird. Da hier ein JOIN auf *t_funktion* erfolgt, lässt sich diese nicht mit *ApplyUpdates* aktualisieren, zumindest nicht bei der mir vorliegenden Delphi-Version, allen Versuchen, die Sache mit *UpdateMode* und *ProviderFlags* hinzubiegen, zum Trotz.

Also verwenden wir eine *TSQLQuery*-Komponente (*qrWork*) und realisieren die Änderungen mittels SQL-Statement:

```
procedure Tfrm_projekte.btnDeleteReClick(Sender: TObject);
begin
  if MessageDlg('Sollen die Rechte der Funktion „'
      +  dataRechtNAME.AsString + '„' + #13#10
      + 'dem Projekt „' + dataProjektNAME.AsString
      + '„ entzogen werden?', mtConfirmation, [mbYes, mbNo], 0)
    = mrYes then
  begin
    with qrWork do
    begin
      SQL.Text := 'DELETE FROM t_recht';
      SQL.Add('  WHERE projekt = :projekt');
      SQL.Add('    AND funktion = :funktion');
      ParamByName('projekt').AsInteger := dataProjektID.AsInteger;
      ParamByName('funktion').AsInteger
        := dataRechtFUNKTION.AsInteger;
      ExecSQL(false);
      RefreshQuery(dataRecht);
      RefreshQuery(qrBenRechte);
      gvChanged := true;
    end;
  end; {if MessageDlg...}
end; {procedure Tfrm_projekte.btnDeleteReClick}

procedure Tfrm_projekte.btnClearReClick(Sender: TObject);
begin
  if MessageDlg('Sollen die Rechte aller Funktionen' + #13#10
      + 'dem Projekt „' + dataProjektNAME.AsString
      + '„ entzogen werden?', mtConfirmation, [mbYes, mbNo], 0)
    = mrYes then
  begin
```

```
    with qrWork do
    begin
      SQL.Text := 'DELETE FROM t_recht';
      SQL.Add('  WHERE projekt = :projekt');
      ParamByName('projekt').AsInteger := dataProjektID.AsInteger;
      ExecSQL(false);
      RefreshQuery(dataRecht);
      RefreshQuery(qrBenRechte);
      gvChanged := true;
    end;
  end; {if MessageDlg...}
end; {procedure Tfrm_projekte.btnClearClick}
```

Je nachdem, ob nur einer Funktion oder allen Funktionen Rechte entzogen werden sollen, muss man die WHERE-Klausel strenger oder weiter fassen. Soll nur einer Funktion die Rechte entzogen werden, dann muss auch noch der Parameter *funktion* gesetzt werden. Anschließend wird die Datenmenge *dataRecht* – an der vorbei ja die Änderungen durchgeführt werden – aktualisiert und *gvChanged* auf *true* gesetzt. Da die Änderung der Rechte auch den angemeldeten Benutzer betreffen kann, muss auch *qrBenRechte* aktualisiert werden.

Für den Insert- und den Update-Fall wird dann wieder eine gemeinsame Routine verwendet. Die lokalen Prozeduren, die dabei nicht verändert wurden, wollen wir uns dabei nicht näher ansehen.

```
procedure Tfrm_projekte.btnInsertReClick(Sender: TObject);
// für btnInsertRe und btnEditRe
var
  i: integer;
  Form: Tfrm_rechte;

  procedure CheckSetzen(Field: TField; Check: TCheckBox);
  begin
    ...
  end;

  procedure CheckBoxenSetzen;
  begin
    ...
  end; {procedure CheckBoxenSetzen}
```

```
procedure Feldwert(Param: string; Check: TCheckBox);
begin
  if Check.Checked
    then qrWork.ParamByName(Param).AsString := 'J'
    else qrWork.ParamByName(Param).AsString := 'N'
end;
```

Statt einem Feld ist der Wert nun einem Parameter zuzuweisen.

```
procedure SQLAnweisungInsert;
begin
  with qrWork.SQL do
  begin
    Text := 'INSERT INTO t_recht';
    Add('(projekt, funktion, erstellen, zuweisen, ');
    Add('bearbeiten, fertigstellen, lesen, ');
    Add('informieren, legitimieren, unterprojekte)');
    Add('VALUES');
    Add('(:projekt, :funktion, :erstellen, :zuweisen, ');
    Add(':bearbeiten, :fertigstellen, :lesen, ');
    Add(':informieren, :legitimieren, :unterprojekte)');
  end; {with qrWork.SQL do}
end; {procedure SQLAnweisung}

procedure SQLAnweisungUpdate;
begin
  with qrWork.SQL do
  begin
    Text := 'UPDATE t_recht SET';
    Add('    erstellen = :erstellen,');
    Add('    zuweisen = :zuweisen,');
    Add('    bearbeiten = :bearbeiten,');
    Add('    fertigstellen = :fertigstellen,');
    Add('    lesen = :lesen,');
    Add('    informieren = :informieren,');
    Add('    legitimieren = :legitimieren,');
    Add('    unterprojekte = :unterprojekte');
    Add('  WHERE projekt = :projekt');
    Add('    AND funktion = :funktion');
  end; {with qrWork.SQL do}
end; {procedure SQLAnweisungUpdate}
```

Je nachdem, ob ein Datensatz eingefügt oder geändert wird, muss eine andere SQL-Anweisung verwendet werden.

```
procedure RechteEintragen(Funktion: integer);
begin
  with Form do
  begin
    if not dataRecht.Locate('Funktion', Variant(Funktion), [])
      then SQLAnweisungInsert
      else SQLAnweisungUpdate;
    qrWork.ParamByName('funktion').AsInteger := Funktion;
    qrWork.ParamByName('projekt').AsInteger
      := dataProjektID.AsInteger;
    FeldWert('erstellen', cbErstellen);
    FeldWert('zuweisen', cbZuweisen);
    FeldWert('bearbeiten', cbBearbeiten);
    FeldWert('fertigstellen', cbFertigstellen);
    FeldWert('lesen', cbLesen);
    FeldWert('informieren', cbInformieren);
    FeldWert('legitimieren', cbLegitimieren);
    FeldWert('unterprojekte', cbUnterprojekte);
    qrWork.ExecSQL(false);
  end; {with Form, dataRecht do}
end; {procedure RechteEintragen}
```

In der lokalen Prozedur *RechteEintragen* wird zunächst die zuständige SQL-Anweisung gewählt und anschließend werden die Werte über Parameter zugewiesen. Mit *ExecSQL* wird die Anweisung dann ausgeführt.

```
begin
  Form := Tfrm_rechte.Create(nil);
  try
    CheckBoxenSetzen;
    Form.DBGrid1.Visible := (Sender = btnInsertRe);
    Form.dataFunktionen.Locate('ID',
      Variant(dataRechtFUNKTION.AsInteger), []);
    if Form.ShowModal = mrOK then
    begin
      if Sender = btnInsertRe then
      begin
        with Form.DBGrid1 do
        begin
```

```
            for i := 0 to SelectedRows.Count - 1 do
            begin
              DataSource.DataSet.GotoBookmark
                (Pointer(SelectedRows.Items[i]));
              RechteEintragen(Form.dataFunktionenID.AsInteger);
            end; {for i := 0 to SelectedRows.Count - 1 do}
          end; {with frm_rechte.DBGrid1 do}
        end; {if Sender = btnInsert then}
        if Sender = btnEditRe
          then RechteEintragen(dataRechtFUNKTION.AsInteger);
        RefreshQuery(dataRecht);
        RefreshQuery(qrBenRechte);
        gvChanged := true;
      end; {if frm_rechte.ShowModal = mrOK then}
    finally
      Form.Free;
    end;
end; {procedure Tfrm_projekte.btnInsertClick}
```

Vor dem Anzeigen von *Tfrm_rechte* werden dort die Checkboxen entsprechend gesetzt, so dass im Update-Fall der bisherige Status voreingestellt wird. Nach dem Anzeigen wird wieder zwischen Insert- und Update-Fall unterschieden. Im Update-Fall muss nur ein einziger Datensatz geändert werden, während im Insert-Fall für alle in *Form.DBGrid1* markierten Funktionen ein Datensatz zu erstellen ist. Neben *dataRecht* ist dann auch wieder *qrBenRechte* zu aktualisieren und *gvChanged* auf *true* zu setzen.

Die Verwaltung der Prioritäten

Die Verwaltung der Prioritäten ist wiederum wenig aufregend, hier wird von *TQuery* nach *TSQLClientDataSet* umgestellt, die Statements bleiben dieselben. Für die Namensänderung (von *qrPrios* auf *dataPrios* und von *qrPriosAll* auf *dataPriosAll*) wird die Suchen- und Ersetzen-Funktion verwendet.

Mit *ApplyUpdates* werden dann wieder die Daten zum Server übertragen. Anschließend werden *dataPrios* und *dataPriosAll* aktualisiert. Bei *dataPriosAll* mag das ja noch einleuchten, aber warum bei *dataPrios*? Nun, für die Prioritäten gibt es wieder eine ID, die über einen Generator vergeben wird. Ausnahmsweise ist es hier einmal nicht nötig, über eine STORED PROCEDURE einen Generatorwert zu holen. Damit die *TSQLClientDataSet*-Instanz jedoch den durch den TRIGGER eingefügten ID-Wert erhält, muss hier auch *dataPrios* aktualisiert werden.

```
procedure Tfrm_projekte.btnInsertPriClick(Sender: TObject);
// für btnInsert und btnEdit
      ...
    if Form.ShowModal = mrOK then
    begin
      dataPrios.Post;
      dataPrios.ApplyUpdates(0);
      refreshQuery(dataPrios);
      RefreshQuery(dataPriosAll);
    end {if frm_prios.ShowModal = mrOK then}
      ...
end; {procedure Tfrm_projekte.btnInsertPriClick}
```

20.3 Die Einträge

Bei den Einträgen wurde auf *RequestLive* und *TUpdateSQL* verzichtet, somit macht die Umstellung keine größeren Probleme.

Die Unit u_EintragDetail

Da *qrEintrag* nur einen Datensatz zurückgibt, können wir hier *TSQLQuery* verwenden. Das Gleiche gilt für *qrRechte*. Auch für *qrAddStatus* kann *TSQLQuery* verwendet werden, da hier nur ein INSERT-Statement abgesetzt wird. Für Status und Datei müssen jedoch *TSQLClientDataSet*-Instanzen eingesetzt werden, da hier die Anzeige mit *TDBGrid* erfolgen soll.

Die Aufrufe zur Statusänderung, zum Hinzufügen von Informationen und zum Zuweisen von Projekt, Funktion oder Priorität bedürfen zwar keiner qualitativen Änderung, allerdings muss im Bereich der Transaktionssteuerung ein wenig Hand angelegt werden:

```
procedure Tfrm_EintragDetail.btnStatusClick(Sender: TObject);
  ...
      frm_data.BeginTransaction;
      try
        WerteEintragen;
        frm_data.SQLConnection1.Commit(TD);
      except
        frm_data.SQLConnection1.Rollback(TD);
      end;
  ...
end; {procedure Tfrm_EintragDetail.btnStatusClick}
```

Für die untergeordneten *TSQLClientDataSet*-Instanzen muss die Master-Detail-Verknüpfung manuell erstellt werden:

```
procedure Tfrm_EintragDetail.qrEintragAfterScroll(DataSet:
TDataSet);

  procedure DoDetail(Data: TSQLClientDataSet);
  begin
    with Data do
    begin
      Close;
      Params[0].AsInteger := qrEintragID.AsInteger;
      Open;
    end;
  end; {procedure DoDetail}

begin
  DoDetail(dataStatus);
  DoDetail(dataDatei);
end; {procedure Tfrm_EintragDetail.qrEintragAfterScroll}
```

Die Units u_zuweis und u_NeuEintrag

Da beim Scrollen im Baum *Locate* aufgerufen wird, muss die Datenmenge für das Projekt vom Typ *TSQLClientDataSet* sein, die anderen Komponenten können nach *TSQLQuery* umgestellt werden. In den ComboBoxen werden zwar Daten aus mehreren Datensätzen angezeigt, die Datenmengen werden jedoch sequenziell ausgelesen, so dass es kein Problem ist, dass sie unidirektional sind.

Drucken

Eigentlich werden bei QuickReport die Daten sequenziell ausgelesen und für die Vorschau zur Entwurfszeit ist die Verwendung von *TSQLQuery* auch völlig unproblematisch. Zur Laufzeit wird jedoch die Eigenschaft *RecordCount* abgefragt, und das unterstützen unidirektionale Datenmengen nicht, so dass wir hier doch *TSQLClientDataSet* verwenden müssen.

20.4 Die Anzeige

Bei der Anzeige »rächt« sich dann wieder der Einsatz von *TUpdateSQL*. Während wir *qrAnzeige* unter Beibehaltung der SQL-Anweisung in *dataAnzeige* umwandeln können, müssen wir die Datenänderung gänzlich neu aufbauen und nutzen dafür wieder *sqlWork*.

Schauen wir uns zunächst den einfachsten Fall an, das Löschen eines Datensatzes:

```
procedure Tfrm_abfrage.btnDeleteClick(Sender: TObject);
begin
  if MessageDlg('Soll die Anweisung „' + dataAnzeigeNAME.AsString
      + '„ gelöscht werden?', mtConfirmation, [mbYes, mbNo], 0)
    = mrYes then
  begin
    with sqlWork do
    begin
      SQL.Text := 'DELETE FROM t_anzeige';
      SQL.Add('  WHERE id = :id');
      ParamByName('id').AsInteger := dataAnzeigeID.AsInteger;
      ExecSQL(false);
    end;
    RefreshQuery(dataAnzeige);
  end; {if MessageDlg...}
end; {procedure Tfrm_abfrage.btnDeleteClick}
```

Die Grundstruktur ist dieselbe, wie wir sie auch beim Einfügen und Ändern von Datensätzen verwenden werden:

- Der Komponente *sqlWork* das passende Statement zuweisen.
- Setzen der Parameter.
- Die Anweisung mit *ExecSQL* ausführen.
- Die Datenmenge *dataAnzeige* aktualisieren.

Für den Insert- und den Update-Fall verwenden wir dann wieder dieselbe Ereignisbehandlungsroutine. Wir werden erst den Hauptteil dieser Prozedur besprechen und dann die beiden lokalen Prozeduren *WerteSetzenInsert* und *WerteSetzenUpdate*:

```
procedure Tfrm_abfrage.btnEditClick(Sender: TObject);
// für btnEdit und btnInsert
var
  Form: Tfrm_EditAnweisung;

  ...

begin
  Form := Tfrm_EditAnweisung.Create(nil);
  try
```

```
    if Sender = btnInsert
      then WerteSetzenInsert
      else WerteSetzenUpdate;
    sqlWork.ParamByName('benutzer').DataType := ftInteger;
    if Form.ShowModal = mrOK then
    begin
      if Form.rbPrivat.Checked
        then sqlWork.ParamByName('benutzer').AsInteger := gvUserID;
      sqlWork.ParamByName('name').AsString
        := Form.Edit1.Text;
      sqlWork.ParamByName('anweisung').Assign(Form.Memo1.Lines);
      sqlWork.ExecSQL(false);
      RefreshQuery(dataAnzeige);
    end; {if Form.ShowModdal = mrOK then}
  finally
    Form.Free;
  end;
end; {procedure Tfrm_abfrage.btnEditClick}
```

Zunächst einmal wird eine Instanz von *Tfrm_EditAnweisung* erstellt, deren Routine *SetWerte* bei der nun folgenden Vorgehensweise überflüssig wird. Vom betätigten Button hängt ab, welche der beiden lokalen Prozeduren aufgerufen wird.

Bisweilen behält die Spalte *benutzer* den Wert NULL, somit wird auch dem entsprechenden Parameter nichts zugewiesen. Durch den Aufruf einer Eigenschaft wie *AsInteger* oder *AsString* wird jedoch die Eigenschaft *DataType* des Parameters gesetzt. Unterbleibt ein solcher Aufruf, dann behält der Parameter in der Eigenschaft *DataType* den Wert *ftUnknown*, was zu einer Exception führen würde – folglich setzen wir *DataType* explizit auf *ftInteger*.

Nun wird das Formular mit *ShowModal* angezeigt. Wird es mit *Ok* geschlossen, dann werden die Parameter *name* und *anweisung* gesetzt; wenn es sich um eine private Anweisung handelt auch *benutzer*. Anschließend wird die Anweisung mit *ExecSQL* ausgeführt und dann die Datenmenge *dataAnzeige* aktualisiert.

```
  procedure WerteSetzenInsert;
  begin
    with Form do
    begin
      Caption := 'Neue Anweisung hinzufügen';
      rbPrivat.Enabled := gvIsAdmin;
      rbPublic.Enabled := gvIsAdmin;
    end; {with Form do}
```

```
    with sqlWork.SQL do
    begin
      Text := 'INSERT INTO t_anzeige';
      Add('(name, anweisung, benutzer)');
      Add('VALUES');
      Add('(:name, :anweisung, :benutzer)');
    end; {with sqlWork.SQL do}
  end; {procedure WerteSetzenInsert}
```

Im Insert-Fall hängt der *Enabled*-Status der beiden Radiobuttons davon ab, ob der Benutzer ADMIN-Rechte hat. Des Weiteren muss *sqlWork* das passende Statement zugewiesen werden.

```
  procedure WerteSetzenUpdate;
  begin
    with Form do
    begin
      Caption := 'Anweisung bearbeiten';
      Edit1.Text := dataAnzeigeNAME.AsString;
      Memo1.Lines.Assign(dataAnzeigeANWEISUNG);
      rbPrivat.Enabled := false;
      rbPublic.Enabled := false;
      rbPublic.Checked
        := dataAnzeigeBENUTZER.IsNull;
    end; {with Form do}
    with sqlWork.SQL do
    begin
      Text := 'UPDATE t_anzeige SET';
      Add('name = :name,');
      Add('anweisung = :anweisung,');
      Add('benutzer = :benutzer');
      Add('WHERE id = :id');
    end; {with sqlWork.SQL do}
    sqlWork.ParamByName('id').AsInteger := dataAnzeigeID.AsInteger;
  end; {procedure WerteSetzenUpdate}
```

Bei der hier verwendeten Vorgehensweise können wir in *Tfrm_EditAnweisung* keine datensensitiven Komponenten mehr gebrauchen. Diese Komponenten werden also durch *Edit1* und *Memo1* ersetzt und diesen müssen zunächst einmal die passenden Werte zugewiesen werden. Die Radiobuttons bleiben stets nicht verfügbar, allerdings muss *rbPublic* aktiviert werden, wenn eine öffentliche Anweisung verändert wird.

Anschließend wird wieder die SQL-Anweisung zugewiesen. Den Wert für den Parameter *id* beziehen wir aus dem gerade aktuellen Datensatz.

Der Aufruf des Assistenten

Die vom Assistenten generierten Daten haben wir schon immer per SQL-Statement eingefügt, auch wenn wir dieses bei der *TUpdateSQL*-Komponente »entlehnt« haben. So müssen nun zwar einige Anpassungen, jedoch keine substanziellen Änderungen erfolgen:

```
procedure Tfrm_abfrage.btnAssiClick(Sender: TObject);
var
  Form: Tfrm_AssiAbfrage;
  sl: TStringList;
begin
  Form := Tfrm_AssiAbfrage.Create(nil);
  sl := TStringList.Create;
  try
    if Form.ShowModal = mrOK then
    begin
      with sqlWork.SQL do
      begin
        Text := 'INSERT INTO t_anzeige';
        Add('(name, anweisung, benutzer)');
        Add('VALUES');
        Add('(:name, :anweisung, :benutzer)');
      end; {with sqlWork.SQL do}
      with sqlWork do
      begin
        ParamByName('Name').AsString := Form.edtName.Text;
        sl.Text := Form.GetText;
        ParamByName('Anweisung').Assign(sl);
        ParamByName('Benutzer').AsInteger := gvUserID;
        ExecSQL;
      end; {with upAnzeige.Query[ukInsert] do}
      RefreshQuery(dataAnzeige);
    end; {if Form.ShowModal = mrOK then}
  finally
    Form.Free;
    sl.Free;
  end;
end; {procedure Tfrm_abfrage.btnAssiClick}
```

20.5 Das Gesamtprojekt

Nun – so könnte man meinen – muss lediglich noch *Query1* von *u_main* gegen eine *TSQLClientDataSet*-Instanz gewechselt werden und das Gesamtprojekt ist umgestellt. Aber hier spielt uns *TSQLClientDataSet* wieder einen kleinen Streich: Nach dem Zuweisen vom *CommandText* wird die Eigenschaft *Params* nicht aktualisiert, *Params.Count* behält den Wert null und das Zuweisen eines Parameters endet in einer Exception. Lässt man jedoch den Parameter weg, dann wird dessen Fehlen beim Öffnen der Datenmenge bemängelt. In der mir vorliegenden Delphi-Version half auch der Einsatz von *FetchParams* nicht weiter.

Also machen wir es umständlich:

- *Query1* wird durch *SQLQuery1* ersetzt, als Komponente im Designer sowie mit *Suchen und Ersetzen* im Quelltext. Eine SQL-Anweisung braucht nicht zugewiesen werden, *SQLConnection* sollte aber nicht vergessen werden.
- An *SQLQuery1* hängen wir die Komponente *DataSetProvider1*. Deren Eigenschaft *DataSet* wird auf *SQLQuery1* gesetzt.
- Nun wird noch eine *TClientDataSet*-Instanz eingefügt, deren Eigenschaft *Provider* auf *DataSetProvider1* gesetzt wird.

Und schließlich muss auch noch die Routine *Anzeigen* abgeändert werden: Es müssen nun sowohl *SQLQuery1* als auch *ClientDataSet1* aktualisiert werden.

```
procedure Tfrm_main.Anzeigen(Sender: TObject);
var
  i: integer;
  SQ: TSQLQuery;
begin
  SQ := Sender as TSQLQuery;
  with SQLQuery1 do
  begin
    SQL.Text := SQ.FieldByName('anweisung').AsString;
    for i := 0 to Params.Count - 1
      do Params[i].AsInteger := gvUserID;
    if Length(SQL.Text) > 10 then
    begin
      RefreshQuery(SQLQuery1);
      RefreshQuery(ClientDataSet1);
      SpaltenUeberschriften;
      StatusBar1.Panels[2].Text
        := ' ' + FormatDateTime('hh.mm.ss:   ', now)
          + SQ.FieldByName('name').AsString;
```

```
        FAbfrageAktuell := SQ.FieldByName('ID').AsInteger;
      end;
    end; {with Query1 do}
end; {procedure Tfrm_main.Anzeigen}
```

Am Rande: Der Sender wird hier einmal einer Typenumwandlung unterzogen, das Ergebnis wird in *SQ* (wie *SenderQuery*) gespeichert – das macht den Quelltext ein wenig übersichtlicher.

Vergessen Sie nicht, in *frm_PrintListe* auf *ClientDataSet1* umzustellen, genauso bei der Anzeige oder dem Druck von Details:

```
procedure Tfrm_main.Detailberichtdrucken1Click(Sender: TObject);
var
  i: integer;
begin
  i := ClientDataSet1.FieldByName('ID').AsInteger;
  case (Sender as TMenuItem).Tag of
    1: TEintrag.Anzeigen(i);
    2: TEintrag.Drucken(i);
    3: TEintrag.DruckenDetail(i);
  end;
end;
```

20.6 Fazit

Drei Mal dasselbe Projekt, aber mit jeweils einer anderen Komponentengruppe.

- In der Client-Server-Programmierung gab es immer schon die »Puristen«, die alles mit *TQuery* gelöst haben, dabei nie *RequestLive* einsetzten und auch *TUpdateSQL* außen vor ließen. Eine solche Vorgehensweise überzeugt durch hohe Performance bei der Programmausführung, benötigt aber bei der Programmierung ein wenig länger.

  (Bei der BDE-Variante wurde hier gemischt vorgegangen: Bei den »kleinen« Tabellen wurde die Bequemlichkeit von *RequestLive* und *TUpdateSQL* eingesetzt, bei den Einträgen jedoch alles per INSERT-Statement gelöst.)

  Projekte, die »puristisch« erstellt worden sind, lassen sich mit minimalem Aufwand umstellen, sowohl nach *IBX* als auch nach *dbExpress*. Eigentlich müssen hier nur Eigenschaftswerte übernommen und Bezeichner angepasst werden.

- Der Unterschied bei der Neuprogrammierung war deutlich geringer als zunächst angenommen. Die *InterBase*-Komponenten haben ganz leicht »die Nase vorne«, insbesondere dank *GeneratorField*. Lösungen mit der BDE sind nur sehr unwesentlich aufwändiger.

  Erstaunlich gut schlagen sich die Komponenten von *dbExpress*, wenn man bedenkt, was diese alles nicht können – passt man sich jedoch von Anfang an diesen Beschränkungen an, dann ist der Mehraufwand im Vergleich zum Gesamtaufwand des Projektes fast schon vernachlässigbar. Auf der anderen Seite entfällt der Nachteil, die BDE installieren zu müssen. Trotzdem kann man dasselbe Projekt auf verschiedenen Datenbank-Servern ausführen.

- Ein paar »Ungereimtheiten« bei den *dbExpress*-Komponenten lassen sich vergleichsweise einfach umgehen. Sobald diese ausgeräumt sind, dürfte der Aufwand für die Programmierung – vorausgesetzt man weiß, was man tut – in der Größenordnung von BDE und IBX liegen.

- Und zuletzt noch ein Blick zu den Dateigrößen:

  - BDE: 1485 kbyte
  - IBX: 1780 kbyte
  - dbExpress: 1849 kbyte.

  Fairerweise sollte man nicht unerwähnt lassen, dass die BDE selbst noch mal ziemlich groß ist.

Index

Symbole

0/1/n : 0/1/m-Master-Detail-Beziehungen 238
3-D-Darstellung 353

A

AbortOnKeyViol 207
AbortOnProblem 207
Abschottung 535
Accept 189
ACTIVE 522
Active 89, 377, 573, 609, 618
ActiveX 837
ADD 97, 223, 499
AddAlias 143
AddArrow 398
AddBubble 396
AddDatabase 573
AddFieldDef 97
AddGantt 394
AddGanttColor 394
AddIndex 201, 288
AddPassword 142
AddTransaction 571
AddXY 352, 381
AddY 381
Adress 629
Adressenliste 423
Adressensuche 146, 243
AFTER DELETE 522
AFTER INSERT 522
AFTER UPDATE 522
AfterCancel 87
AfterClose 90, 609
AfterDelete 87
AfterEdit 87
AfterInsert 87
AfterOpen 90, 609
AfterPost 87
AfterRefresh 95, 613
AfterScroll 85, 611
Aggregate 280
Aggregates 290
AggregatesActive 280, 290
Aggregatfunktionen 217, 290
Alias 211, 255
AliasName 163
Alignment *107*, 318, 319, 345, 371, 399, 414
ALL 221, 523
AllowDelete 348
AllowGrayed 336
AllowInsert 348
AllowPanning 368
AllowZoom 368
ALTER DOMAIN 484
ALTER PROCEDURE 519
ALTER TABLE 223, 498
ALTER TRIGGER 522
Alternate Key 64
AND 214
AnimatedZoom 368
AnimatedZoomSteps 368
Anwendungsserver 619, 632
ANY 221
Append 85
AppendRecord 85, 86
Application-Server 59
Apply 243, 582
ApplyRange 184, 199, 286
ApplyUpdates 193, 237, 283, 574, 623, 631
AreaLinesPen 390
Array 481
ArrowHeight 398
ArrowWidth 398
AS 512
AsBCD 104
AsBoolean 104
AsCurrency 104
AsDateTime 104

AsFloat 104
AsInteger 104
Assign *104*
AssignValue *106*
AssignValues 382
AsSQLTimeStamp 104
AsString 104
AsVariant 104
atomar 60
Attributes 60, 98
AttributeSet *110*
Ausdehnungsgrad 62
Ausführungsgeschwindigkeit 589
AutoCalcFields 94
AutoDisplay 328, 330, 340
AutoEdit 117
AUTOINC 223, 480
autoinkrementierendes Feld 548
Automatic 364
AutomaticMaximum 364
AutomaticMinimum 364
Automatische Transaktionskontrolle 538
AutoRefresh 400
AutoSave 618
AutoSize 324, 325
AVG 217
AxisValuesFormat 365
AxisVisible 363

B

BackColor 373
BackImage 373
BackImageInside 373
BackImageMode 373
Backup 465
BandType 439
BarBrush 389
BarPen 389
BarStyle 389
batAppend 197, 207
batAppendUpdate 198, 207
BatchMove 197
batCopy 197, 207
batDelete 197, 207
batUpdate 197, 207
BDE 209
BEFORE DELETE 522
BEFORE INSERT 520, 522
BEFORE UPDATE 522
BeforeAction 323
BeforeCancel 87
BeforeClose 90, 609
BeforeDelete 87
BeforeEdit 87
BeforeInsert 87
BeforeOpen 90, 609
BeforePost 87
BeforeRefresh 95, 613
BeforeScroll 85, 611
BEGIN 512
BeginTrans 846
Beispielprojekt Adressensuche 243
Berechnete Felder *100*
Bereich 183
BETWEEN 473
Bild 480
binär 62
binary large Object 480
BLOB 223, 480, 594
BlobType *113*
BOF 82, 584, 610
Bookmark 85
BookmarkValid 850
BOOLEAN 223
BottomAxis 362
BottomWall 373
Broker 632
BrokerName 630
Brush 391
BtnClick 323
Business-Rules 59
ButtonAutoSize 412
ButtonHeight 412
ButtonSpacing 412

ButtonStyle 319
ButtonWidth 412
BYTES 223

C

Cached Updates 59, 192
CachedUpdates 91, 193, 237, 574
CalcPosPoint 369
Calculated *109*
CalcXPos 384
CalcXPosValue 385
CalcXSizeValue 385
CalcYPos 384
CalcYPosValue 385
CalcYSizeValue 370, 385
Cancel 87, 845, 847
CancelRange 185, 199, 286
CancelUpdates 194, 283, 574
CanClose 624
Candidate Key 62
CanModify 93, *108*, 612, 849
CanUndo 327, 330
Capacity 408
Caption 318
CaptionColor 415
CaptionFont 415
CAST 219, 481
Cells 413
Center 330
CHANGE_LOG 271
CHAR 479
CHAR VARYING 478
CHARACTER 223, 479
CHARACTER VARYING 478
CharCase 325
Chart3dPercent 372
CHECK 483, 488, 497
CHECK OPTION 507
CheckDataSource 380, 400
Checked 336
Circled 393
Clear 97, *104*, 316, 327, 330, 382
ClearFields 87
ClearSelection 327, 330
ClearUndo 327, 330
Clicked 385
Client 619, 622, 633
Client-Server 509
Client-Server-Datenbank 56
Clone 849
CloneCursor 283
CloneSource 283
Close 90, 163, 570, 583, 609, 845
CloseDataSets 607
CloseDatasets 166
Codd 60
ColCount 347, 413
COLLATE DE_DE 479
Color 319, 342, 414
ColorRange 379
ColorSource 377
Columns 317, 337, 427
CommandText
284, 589, 614, 617, 846, 851
CommandType 589, 614, 617, 851
Commit 153, 167, 572, 605
CommitRetaining 572
CommitTrans 846
CommitUpdates 193
Communication Diagnostics 458
CompareBookmark 850
CompareBookmarks 84
COMPUTED BY 489
ComputerName 627
ConfirmDelete 322
Connected
163, 570, 603, 623, 626, 840, 845
ConnectingPen 395
Connection 846, 847
ConnectionBroker 282
ConnectionName 603
ConnectionString 845, 846, 847
ConnectionTimeout 845
ConsistentAttributes 343
Constrained 232
CONSTRAINT 494, 500

ConstraintErrorMessage 109
Constraints 89, 631
CONTAINING 474
ControlsDisabled 91, 610
CopyToClipboard 326, 330
CopyToClipboardBitmap 375
CopyToClipboardMetafile 375
CORBA 628
Corba 631
COUNT 217
Count 97, 316, 382, 383
CREATE DOMAIN 477
CREATE EXCEPTION 522
CREATE GENERATOR 486
CREATE INDEX 224, 503
CREATE PROCEDURE 512
CREATE TABLE 223, 488
CREATE TRIGGER 520
CREATE VIEW 505
CreateDataSet 269, 282
CreateTable 196, 576
Currency *113*
CurrentRowSelected 316
CurrentSum 409
Cursor 378
Cursor-Handle 543
CurValue 105
CustomConstraint 89, 109
CustomXRadius 393
CustomYRadius 393
CutToClipboard 326, 330

D

daAbort 88
daFail 88
daRetry 88
Data-Pump-Experte 842
Database 191, 574, 583
DatabaseCount 573
DatabaseName 163, 190, 570
Databases 573
DataColor 415
DataFont 415
DataSet *108*, 116, 243, 582, 630
DataSetCount 165, 607
DataSets 165, 607
DataSize *110*
DataSource 233, 380, 580, 613, 853
DataSumColor 415
DataType 98, *110*
DATE 223, 480
Datenanalyse 349, 403
Datenanalyse im Thread 417
Datenänderung 221
Datenbank
  Client-Server 56
  Definition 53
  hierarchisch 55
Datenbank-Management-Systeme 53
Datenbank-Explorer 134
Datenbank-Oberfläche 123, 293, 548
Datenbankserver 619
Datenbanksysteme 53
Datenbestände 53
Datenkonvertierung 481
Datensätze zählen 558
Datensteuerung 309
Datentransfer 203
DateTime 383
DateTimeFormat 365
Datum 480
DB2 587
dBase 192
dBase-Tabellen 133
dbExpress 587, 602
DBO 123
DBSession 191
DCOM 631
Deadlock 537
deadlock 496
DECIMAL 223, 479
DecisionCube 408
DecisionSource 411, 413, 417
DECLARE VARIABLE 512
DEFAULT 488
DefaultColWidth 416

DefaultDatabase 573, 845
DefaultDrawing 317
DefaultExpression 110
DefaultFields 93, 612
DefaultIndex 577
DefaultRowHeight 416
DefaultTransaction 571
Degree 62
Delete 87, 316, 382
DeleteAlias 143
DeleteIndex 201, 288
DeleteRecords 616, 849
DeleteSQL 236, 242, 581, 582
DeleteTable 196, 576
DepthAxis 362
DESC 216, 224
Destination 206
Dictionary 139
DimensionMap 406
Dimensions 414
Dirty Reads 535
DisableControls 91, 610
DisplayFormat *113*
DisplayLabel *107*
DisplayName *107*
DisplayText *105*
DisplayValues *113*
DisplayWidth *107*
DISTINCT 210
DoKey 348
Domain 60
DOMAINS 478
DOUBLE PRECISION 479
Draw3D 391
DrawColumnCell 319
DriverName 604
DROP 223, 500
DROP CONSTRAINT 494
DROP DOMAIN 485
DROP INDEX 224, 503, 557
DROP PROCEDURE 519
DROP TABLE 224, 503
DROP TRIGGER 523
DROP VIEW 509
DropDownAlign 339
DropDownCount 335
DropDownRows 339
DropDownWidth 339
dsBrowse 90, 609
dsCalcFields 90, 609
dsCurValue 91
dsEdit 90
dsFilter 90
dsInactive 90, 609
dsInsert 90
dsNewValue 91
dsOldValue 91
dsSetKey 90

E

Edit 86
EditFormat *114*
editierbar 588
EditKey 199, 286
EditMask 106, *114*
EditRangeEnd 199, 286
EditRangeStart 199, 286
EmptyDataSet 285
EmptyTable 197, 577
EnableControls 91, 610
Enabled 117
enabled 440
END 512
EndValues 395
EndXValues 398
EndYValues 398
Entity-Relationship-Modell 73
EOF 82, 584, 610
ErrorMessage 89
Events 585
ExactDateTime 365
EXCEPTION 522
Exclusive 196
ExecProc 580
ExecQuery 583

ExecSQL
231, 243, 542, 579, 582, 615, 854
Execute 207, 284, 419, 608, 847
EXECUTE PROCEDURE 517, 542
ExecuteDirect 608
EXISTS 221
Exists 576
EXIT 516
explizite Transaktion 550
Externes Datenmodul 620
EXTRACT 220

F

Feld
selbstinkrementierend 128
Feldeditor *99*
Felder
berechnete *100*
Lookup *101*
Feldtypen 127
FetchAll 191, 558, 576
FetchOnDemand 284
Field 318
FieldByName 91, 584, 611
FieldCount 93, 612
FieldDefs 92, 267, 269, 282
FieldKind *110*
FieldName *108*, 318, 414
FieldNo *108*
Fields 91, 584, 611
FieldValues 92, 611
File-Systeme 54
FileName 281, 618
Filialbetrieb 59
Filter 94, 187, 553
Filtered 94
FilterGroup 850
Filtern 183, 213
filtern 227, 553
FilterOnBookmarks 850
FilterOptions 94
Find 97, 316
FindDatabase 573
FindField 91, 611
FindFirst 82
FindKey 185, 200, 287
FindLast 82
FindNearest 200, 287
FindNext 82
FindPrior 82
FindText 341
FindTransaction 571
Firewall 460
First 81, 595, 610
FirstIndent 344
FirstValue 371
FirstValueIndex 380
FixedCols 413
FixedRows 413
fkCalculated 110
fkData 110
fkLookup 110
FLOAT 223, 479
FlushBuffers 191
FocusControl *107*
Font 317, 319, 399
Foot 374
FOR SELECT...DO 513
FOR...TO...DO 514
ForceNewPage 440
FOREIGN KEY 495
Foreign Key 64
Format 415
Frame 440, 446
FreeBookmark 84
FreeOnTerminate 147, 418
Fremdschlüssel 65, 494
Fremdschlüsselverletzung 552
FULL OUTER JOIN 213
FunctionType 380
Funktionen 218

G

GEN_ID 487, 520
GENERATOR 485, 520
GetActiveDim 410

GetAliasDriverName 143
GetAliasNames 143, 204
GetAliasParams 143
GetAxisSeries 363
GetBookmark 84
GetCursorValueIndex 385
GetCursorValues 386
GetData *106*
GetDataAsString 409
GetDataAsVariant 409
GetDimensionName 410
GetFieldNames 94, 572, 606, 612
GetGroup 410
GetIndex 410
GetIndexNames
201, 289, 578, 606, 852, 853
GetMemberAsString 410
GetMemberAsVariant 410
GetNextPacket 284
GetProcedureNames 606
GetProcedureParams 606
GetStoredProcNames 144
GetTableNames 144, 572, 606
GetTablenames 204
Gewartete Aggregate 290
GotoBookmark 84
GotoCurrent 200, 287, 577
GotoKey 200, 287
GotoNearest 200, 287
Gradient 373
GRANT 523
GRANT OPTION 523
Grid 367
GridLineColor 415
GridLineWidth 415
Groß- und Kleinschreibung 227
GROUP BY 217
GroupLayout 412
Groups 412
GroupSpacing 412
Gruppieren 217
Guardian 453
Gültigkeitsprüfung 129

H

HAVING 217
Height 342
Heterogene Joins 230
hierarchische Datenbank 55
Hints 323
HorizAxis 378
horizonale Teilmenge 504
HorizSize 391
Host 457, 629
HostName 628

I

IBConsole 455
IdleTimer 571
IF...THEN...ELSE 515
ImportedConstraint 89
IN 215
INACTIVE 522
Increment 365
Index *108*, 493, 503
IndexDefs 201, 282, 289, 578
IndexFieldCount 850
IndexFieldNames
201, 276, 288, 577, 616, 853
IndexFields 201, 288, 578, 850
IndexName 201, 288, 577, 616
IndexOf 97, 316
Indizes 555
Inkonsistenz 54
Inkrementale Suche 186
INNER JOIN 211, 212
InputBox 185
INSERT 221
Insert 86
InsertRecord 86
InsertSQL 236, 242, 581, 582
Installation 587
INTEGER 223, 479
IntegralHeight 332
Integrität 54
Interactive SQL 467

InterBase 453, 469, 587
InternalCalcField 98
InTransaction 167, 573, 606, 846
Inverted 364, 371
InvertedStairs 387, 390
IP-Adresse 457
IS NULL 216
IsEmpty 93, 612
IsIndexField *110*
IsNull *104*
IsReadOnly 571
IsSequenced 93, 848
IsUnidirectional 96, 613
IsValidChar 106
ItemIndex 332, 337
Items 97, 316, 331, 334, 337

J

Jet-Engine 839
JOIN 211, 505
Jokerzeichen 210, 215

K

Kardinalität 62
KeepConnection 165, 604
KeyExclusive 199, 286
KeyExlusive 184
KeyField 338, 339
KeyFields *111*
KeyViolTableName 207
Konkurrierender Zugriff 471
Konsistenz 469
Konstanten 210
Kopieren 548

L

LabelColor 415
LabelFont 415
Labels 367
LabelsAngle 367
LabelsFont 367
LabelsSeparation 367
LabelsSize 367
LabelStyle 367
LabelSumColor 415
Last 81
LastValueIndex 380
lbOwnerDrawFixed 333
LEFT OUTER JOIN 212
LeftAxis 362
LeftIndent 344
LeftWall 373
Legend 369, 370
LegendStyle 371
Lesezeichen 84
LibraryName 604
LIKE 215, 247, 279, 557
LineBrush 387
LinePen 387
Lines 328
ListField 338, 339
ListSource 338, 339
LoadFromFile
115, 269, 281, 625, 848
LoadFromStream *115*, 269, 281
LoadMemo 328, 340
LoadParamsFromIniFile 604
LoadParamsOnConnect 604
LoadPicture 330
LocalSQL 209
loCaseInsensitive 187
Locate 84, 187
Lock 197
LockType 849
Logarithmic 364
LogChanges 275, 284
LoginPrompt
164, 570, 605, 628, 630, 840
Logische Verknüpfungen 214
Lookup 92, *112*
Lookup-Felder *101*
LookupCache 112
LookupDataSet *112*
LookupKeyFields *112*
LookupResultField *112*
Lost Updates 535
LOWER 215, 218

M

Makler 632
Mappings 208
MarginBottom 374
MarginLeft 374
MarginRight 374
MarginTop 374
Marks 378
Master-Detail 202, 225
Master-Detail-Beziehungen 238
Master-Detail-Verknüpfung 64, 842
MasterFields 202, 290, 578, 616, 852
MasterSource
202, 290, 578, 616, 852
Mathematische Operatoren 218
MAX 217
MAX ROWS 558
MaxCells 408
MaxDimensions 408
Maximum 364
MaxLength 325
MaxPointsPerPage 374
MaxRecords 848
MaxSummaries 408
MaxValue *114*
MaxXValue 363, 386
MaxYValue 363, 386
mehrschichtige
Anwendungen 58, 619
Memo 480
MergeChangeLog 275, 284
Metadaten 223, 461
Microsoft 837
Microsoft Jet 839
Middleware 631
MIN 217
Minimum 364
MinorTickCount 366
MinorTickLength 366
MinorTicks 366
MinValue *114*
MinXValue 363, 386
MinYValue 363, 386
Mode 207
Modem 59
Modified 93
ModifyAlias 143
ModifySQL 236, 242, 581
ModifyValue 361
MONEY 223
MouseCoord 413
MoveBy 82, 610
Multi-Generationen-Architektur 469
Multi-Tier 619
Multi-Tier-Systeme 58
Multi-User-Betrieb 56
MultiArea 390
MultiBar 388
Multiplier 383
MyBase 267
MySQL 587

N

Nachschlagetabelle 130
Name 98, 342, 384
nColDims 409
nDataCols 409
nDataRows 409
nDims 409
Netzwerk 510
Netzwerk-Datenbank 55
NEW 520
NewPage 438
NewValue *105*
Next 81, 584, 610
NextPage 375
NextTask 395
NoMetadata 609
Non-reproducible Reads 536
nOpenColDims 409
nOpenRowDims 409
Normalform 69
NOT NULL 483, 488
NOT SHARED 539
NOW 481, 500
nRowDims 409

nSums 409
NULL 61, 481, 483
Numbering 345
NUMERIC 223, 479

O

Object Management Group 631
ObjectBroker 628, 629
Objektablage 624
ODBC 837
Oder 214
OLD 520
OldValue *105*
OLE-Automatisierungsobjekt 619
OMG 631
ON 523
ON-Klausel 212
OnAfterAdd 381
OnAfterClose 407
OnAfterConnect 164, 604, 627
OnAfterDisconnect 164, 604, 627
OnAfterDrawValues 379
OnAfterOpen 407
OnAfterPivot 411
OnAfterPrint 438, 440
OnBeforeAdd 381
OnBeforeClose 407
OnBeforeConnect 164, 604, 627
OnBeforeDisconnect 164, 604, 627
OnBeforeDrawValues 379
OnBeforeEdit 88
OnBeforeOpen 407
OnBeforePivot 411
OnBeforePrint 438, 440
OnCalcFields 90, 94, 609, 612
OnCellClick 314
OnChange 325, 335
OnChangeField *104*
OnClearValues 382
OnClick 323, 385
OnColEnter 315
OnColExit 315
OnColumnMoved 315
OnDataChange 117
OnDblClick 385
OnDecisionDrawCell 416
OnDeleteError 88
OnDrawColumnsCell 317
OnDrawItem 333, 335
OnDropDown 335
OnEditButtonClick 315
OnEditError 88
OnEndPage 438
OnEventAlert 586
OnFilterRecord 95, 189
OnGetLegendPos 369
OnGetLegendRect 369
OnGetLegendText 370
OnGetMarkText 378
OnGetText *105*
OnGetUsername 628, 630
OnIdleTimer 571
OnLayoutChange 411
OnLogin 164, 605, 628, 630
OnLogTrace 618
OnLowCapacity 408
OnMeasureItem 334, 335
OnNeedData 439, 442
OnNewRecord 87
OnPageChange 375
OnPaintPanel 348
OnPassword 143
OnPostError 88
OnReconcileError 283, 624
OnScroll 368
OnSelectionChange 341
OnSQL 585
OnStartPage 438
OnStartup 143
OnStateChange 118
OnSummaryChange 411
OnTitleClick 314
OnTrace 618
OnUpdateData 118
OnUpdateDate 118
OnUpdateError 195, 575, 631

OnUpdateRecord 195, 575
OnValidate *106*
Open 90, 163, 570, 609, 845
Operatoren 218
Optimierung 547
Options 316, 415
OR 214
Oracle 543, 587
ORDER 555
Order 384
ORDER BY 216, 474
Orientation 348
OUTER JOIN 212

P

PacketRecords 284
Page 374
PainText 340
Paragraph 340
ParamByName
226, 233, 580, 581, 613
ParamCount 233, 581
Parameter 225, 233, 517
Parameters 847, 852, 853, 854
Params
164, 226, 233, 570, 580, 581, 604, 613
PARAMS CHANGE_LOG 271
ParamsCount 233, 580, 581
Parans 583
ParentChart 377
PasswordChar 325
Paßwort 131
PasteFromClipboard 326, 330
Pen 391
PercentFormat 378
Persistente TField-Instanzen 591
PickListe 319
Picture 331
Ping 457
Pipe 218
plattformunabhängig 631
Pointer 396, 397, 399
PopupMenu 318
Port 629
Post 87
Precision 98, *114*
Prepared 614
PreviousPage 375
Primärindex 64
Primärschlüssel 64, 128, 223, 491
PRIMARY KEY 488, 492
Primary Key 64
Print 341, 376
PrintLandscape 376
PrintOrientation 376
PrintPartialCanvas 376
PrintPortrait 376
PrintRect 376
PrintReport 438
Prior 81
ProblemTableName 207
ProcedureName 854
Provider 267, 845
ProviderName 281
PUBLIC 523

Q

QDelete 583
QInsert 583
QRDetailLink 439, 440
QRefresh 583
QRGroup 440
qrsColumnNo 448
qrsDate 447
qrsDateTime 447
qrsDetailCount 447
qrsDetailNo 448
qrsPageNumber 447
qrsReportTitle 447
qrsTime 447
QSelect 583
Query 242, 581
QUpdate 583

R

RadiusValues 397
random 153

randomize 152
rbChildBand 440
rbColumnHeader 440
rbDetail 439
rbOverlay 440
rbPageFooter 439
rbPageHeader 439
rbSubDetail 439
rbTitle 439
RDB$ 478
READ COMMITTED 536
ReadDataPacket 291
ReadOnly
108, 196, 282, 318, 327, 577, 852
ReadSize 291
RecNo 93
RecordCount
93, 284, 285, 558, 612, 848
RecordNo 848
Redundanz 54, 243
REFERENCES 495
Referenzielle Integrität 65, 130
Refresh 95, 232, 613
RefreshData 400
RefreshDataSet 400
RefreshInterval 400
RefreshLookupList 112
RefreshSeries 386
RefreshSQL 581, 582
Registered 586
Reihe 60
Reihenfolge 221
Relation 60
RemoteServer 282, 623
RemoveAllPasswords 143
RemoveDatabase 573
RemoveDatabases 573
RemovePassword 143
RemoveTransaction 571
RemoveTransactions 571
RenameTable 196
Replikationen 469
Report 421
RepositoryID 628
RequestLive 231
Required 98, *108*
Restore 466
RestoreDefaults 319
RETURNS 512
RevertRecord
194, 275, 283, 285, 574
REVOKE 524
RichtIndent 344
RIGHT OUTER JOIN 213
RightAxis 362
Rollback 153, 167, 572, 605
RollbackRetaining 572
RollbackTrans 846
Rotate 393
RotationsAngle 393
RoundFirstLabel 364
RoundRectangle 399
RowCount 347, 413
RowsAffected 579, 854

S

SavePoint 276
SaveToBitmap 375
SaveToBitmapFile 355
SaveToFile *115*, 269, 281, 625, 848
SaveToMetafile 375
SaveToMetafileEnh 375
SaveToStream *115*, 269, 281
Schlüssel 62, 503
SCKTSRVC.EXE 629
SCKTSRVR.EXE 629
Script 468
ScrollBars 329
Secondary Key 64
Seek 849
Sekundärindizes 130
Sekundärschlüssel 64, 491
SelAttributes 340
selbstinkrementierend 128, 543
SELECT 147, 210, 473, 514, 541

SelectAll 327, 330
Selected 332
SelectedColor 348
SelectedField 315
SelectedIndex 315
SelectedRows 315
SelectSQL 582
SelLength 326
SelStart 326
SelText 326
sequenzielle Suche 66
SeriesColor 377
Server 453
ServerGUID 627, 629
ServerName 627, 629
Service 458
SessionName 165, 191
SET 222
SET GENERATOR 486
SetFieldType *115*
SetKey 199, 286
SetParams 243, 582
SetRange 184, 199, 286
SetRangeEnd 183, 198, 285
SetRangeStart 183, 198, 285
SetTraceCallbackEvent 606
SHADOW 525
SHARED AUTOCOMMIT 538
SHARED NOAUTOCOMMIT 539
ShowFocus 348
ShowProgressDialog 407, 419
Size 98, *111*, 342
SMALLINT 223, 479
Smart Agent 632
SNAPSHOT 536
SNAPSHOT TABLE STABILITY 537
SOME 221
Sort 851
SortFieldNames 615
Sortieren 216
Sortierreihenfolge 479
Source 206
Spalte 60
Sperren 470
SQL 147, 231, 473, 579, 583, 853
SQL-Script 468
SQLConnection 608, 617, 618
SQLHourGlass 607
SQLPassThruMode 538
Squared 397
Stairs 387, 390
Stapel-Operationen 133
STARTING WITH 473
StartTransaction 152, 167, 572, 605
StartValues 395
StartXValues 398
StartYValues 398
State 90, 117, 336, 609
StatusFilter 275
STORED PROCEDURE
509, 541, 546, 548
STORED PROCUEDURES 600
StoredProcName 580
Stretch 330
Strings 478
Style 332, 335, 342, 391, 399
SUBSTRING 219
Subtotals 415
Suche 243
nach Nummern 254
sequenziell 66
Suchkriterien 257
Suchstring 247
SUM 217
Supports 848
SUSPEND 516, 546
SwapValueIndex 386
Sybase 543

T

Tab 345
TabCount 345
Tabelle 60
Entfernen zur Laufzeit 182
Tabellen-Alias 211

TableName 196, 576, 852
TableNames 616
TableScope 606
TableType 181, 196, 576
TADOCommand 846
TADOConnection 839, 845
TADODataSet 851
TADOQuery 840, 853
TADOStoredProc 854
TADOTable 842, 852
TAreaSeries 390
TArrowSeries 397
Taskbar-Notofication-Area 454
TBarSeries 388
TBatchMove 203, 206
TBClientDataSet 291
TBDEClientDataSet 267
TBookmark 84
TBubbleSeries 396
TChart 349
TChartAxis 364
TChartShape 399
TChartValueList 383
TClientDataSet 267
TColumn 318
TCorbaConnection 626, 628
TCP/IP 456, 629
TCustomADODataSet 847
TCustomRemoteServer 626
TCustomSQLDataSet 608
TDatabase 163
TDataSource 116
TDateTime 480
TDBChart 349
TDBCheckBox 336
TDBComboBox 334
TDBCtrlGrid 346
TDBEdit 324
TDBGrid 309
TDBImage 330
TDBListBox 331
TDBLookupComboBox 339
TDBLookupListBox 338
TDBMemo 328
TDBNavigator 322
TDBRadioGroup 337
TDBRichEdit 340
TDBText 323
TDCOMConnection 626, 627
TDecisionQuery 406
TDecisionSource 408, 411
Testdaten 547
Text *105*, 232, 329, 399, 448, 579
TextStyle 371
TFastLineSeries 394
TField 842
TFieldDef 98
TFieldDefs 96
TGanttSeries 394
THorizBarSeries 388
Thread 417, 559
TIBClientDataSet 267
TIBCustomDataSet 574
TIBDatabase 570
TIBDatabaseInfo 584
TIBDataSet 582
TIBEvents 585
TIBMonitor 585
TIBQuery 579
TIBSQL 583
TIBStoredProc 580
TIBTable 576
TIBTransaction 572
TIBUpdateSQL 581
TickInner 366
TickInnerLength 366
TickLength 366
Ticks 366
TIME 223
Timer 229
TIMESTAMP 223
Title 318, 367, 374, 377
TitleFont 317
TLineSeries 387
TMidasConnection 626, 627
TO 523

TO PUBLIC 523
TOLEEnterpriseConnection 626, 630
TopAxis 362
TopIndex 332
TotalLegendItems 371
Totals 414
TParaAttributes 343
TPointSeries 391
TProvider 630
TQRBand 439
TQRDBCalc 446
TQRDetailLink 441
TQRLabel 428
TQRSysData 447
TQuery 541, 553
TQuickReport 436
TraceCallbackEvent 606
TraceFlags 572
TraceList 618
Transaction 574, 583
TransactionCount 571
TransactionLevel 608
Transactions 571
TransactionsSupported 606
Transaktion 151, 166, 571
Transaktionen 469
  beim Lesen 469
TransIsolation 167
Transliterate *114*
Transparent 324
TRemoteServer 626, 627
TRIGGER 519, 546
TRIM 219
try...except 153
TSession 559
TSocketConnection 626, 629
TSQLClientDataSet 267, 596, 617
TSQLConnection 588, 602
TSQLDataSet 589, 614
TSQLMonitor 618
TSQLQuery 590, 616
TSQLStoredProc 617
TSQLTable 616
TStoredProc 543
TTable 196, 553
TTextAttributes 342
TThread 419
TUpdateSQL 235
Tuples 60, 62
Typenumwandlung 219

U

uaAbort 195, 575
uaApplied 195, 575
uaFail 195, 575
uaRetry 195, 575
uaSkip 195, 575
UDF 453
Umlaute 479
unär 62
Und 214
Undo 327, 330
UndoLastChange 269, 275, 283, 285
Unidirektional 596
unidirektional 588
UNION 220
UNIQUE 224, 492
Unix 631
Unlock 197
Unterabfragen 220
UpdateObject 195, 574
UpdateRecordTypes 194, 575
UpdatesPending 194, 575
UpdateSQL 582
UpdateStatus 89
UPPER 215, 218, 229
Upsizing 203
UsePatterns 393
USER 481
user defined function 453

V

ValidChars 106
Value *106*
ValueChecked 336
ValueColor 357, 377

ValueFormat 378
ValueListCount 386
VALUES 221
Values 337
ValueSource 384
ValueUnchecked 336
VARCHAR 223, 478
Variable 517
VendorLib 604
Vergleichs-Operatoren 214
Verknüpfungstabelle 239
VertAxis 378
vertikale Teilmenge 504
VertSize 391
VIEW 504
View 68
View3D 372
View3dWalls 373
Visible *109*, 318, 364, 371
VisibleButtons 322
VisibleCount 379
Visibroker Smart Agent 628, 632
Visibroker Smart Finder 632
Vorschauformular 428

W

WantReturns 329
WantTabs 329
Wertebereich 477
WHERE 213, 473, 555
WHERE-Klausel 211
WHILE...DO 514
Width 318
WITH GRANT OPTION 523
WordWrap 324, 329
WriteDataPacket 291
WriteSize 291

X

X0 399
X1 399
XLabel 382
XLabelSource 382
XML-Format 269
XScreenToValue 386
xValue 380
XValues 380
XValueToText 379
XYStyle 399

Y

Y0 399
Y1 399
YScreenToValue 386
yValue 380
YValues 380
YValueToText 379

Z

Zahlen 479
Zeichensatz 475
Zeile 60
Zeit 480
Zeitstempel 470
Zirkuläre Referenzen 496
ZoomPercent 368
ZoomRect 368
Zugriffsberechtigung 523
Zugriffsbeschränkung 504
Zugriffsmodus 506

21 ADO und Access

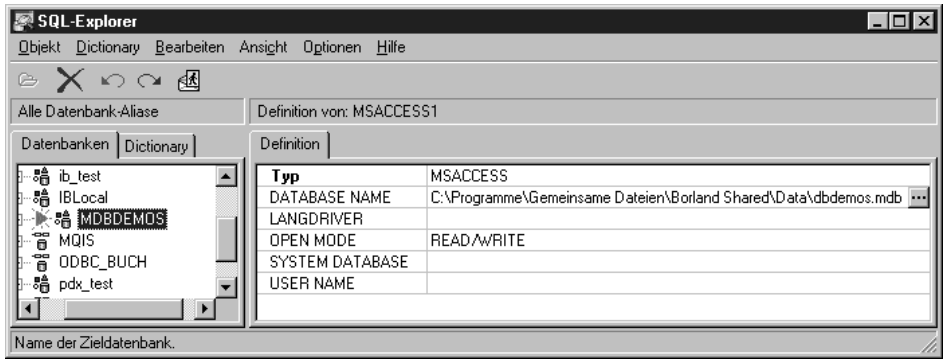
In Delphi 5 ist die Möglichkeit hinzugekommen, mit ADO (*ActiveX Database Objects*) ohne die BDE auf Datenbanken zuzugreifen. Momentan gibt es noch nicht allzu viele Datenbanken, die dafür Treiber bereitstellen, aber diese Technologie wird in den nächsten Jahren vermutlich eine ähnliche Bedeutung erhalten, wie sie die ODBC im Moment schon hat.

Apropos ODBC: Auf alle Datenbanken, für die ein ODBC-Treiber zur Verfügung steht, können Sie mit ADO bereits jetzt schon zugreifen. Allerdings steht ODBC nicht im Verdacht einer hohen Performance (um es einmal so zu formulieren).

Da ADO aus dem Hause Microsoft kommt, ist es nicht verwunderlich, wenn für MS Access bereits Treiber vorliegen. Da bei einer Delphi-Installation auch eine Access-Datenbank auf den Rechner gespielt wird, haben wir die Möglichkeit, mit dieser Technologie erste Erfahrungen zu sammeln. Vorher sollen aber zwei andere Möglichkeiten gezeigt werden, auf eine Access-Datenbank zuzugreifen.

21.1 Access-BDE-Treiber

Seit Dephi 4 gibt es einen nativen BDE-Treiber für Access-Datenbanken. Um hier einen Alias einzurichten, wählen Sie im Datenbank-Explorer OBJECT|NEU und als Treiber *MSACCESS*.



Als *Database Name* wählen Sie nun *C:\Programme\Gemeinsame Dateien\Borland Shared\Data\dbdemos.mdb* aus. (Wenn Sie Delphi in einem anderen Verzeichnis installiert haben, dann ändert sich der Pfad entsprechend.)

Sie können nun die Änderungen übernehmen und die Datenbank öffnen. Wenn Sie nach einem Benutzernamen und einem Passwort gefragt werden, schließen Sie den Dialog ohne Eingabe mit OK.

21.2 Einrichten einer ODBC-Verbindung

Die BDE erstellt automatisch für jede ODBC-Verbindung einen entsprechenden Alias. Auf diese Weise können Sie auch auf Datenbanken zugreifen, für die Sie keinen BDE- oder SQL-Links-Treiber erhalten, jedoch einen ODBC-Treiber.

Die ODBC-Verwaltung können Sie direkt aus dem SQL-Explorer mit OBJECT|ODBC-VERWALTUNG aufrufen. Alternativ gibt es den Weg über die Systemsteuerung.

Gehen Sie auf die Registerseite *Benutzer-DSN* und wählen den Button HINZUFÜGEN. Sie werden dann mit folgendem Dialog konfrontiert:

Geben Sie hier als Datenquellennamen den Namen ein, der später als BDE-Alias verwendet werden soll. Im Beispiel habe ich *ODBC_DBDEMOS* verwendet. Mit AUSWÄHLEN öffnen Sie einen Dateiauswahl-Dialog, mit dessen Hilfe die Datenbankdatei gewählt werden kann. Hier können Sie dann auch gleich einstellen, ob Sie die Datenbank exklusiv oder im Read-only-Modus öffnen möchten.

Nun schließen Sie die Fenster jeweils mit OK. Gegebenenfalls muss der SQL-Explorer neu gestartet werden, damit der neue ODBC-Alias erkannt wird. Sie können nun die Datenbank öffnen, den Anmeldedialog schließen Sie ohne Eingabe von Benutzername und Passwort.

Wenn Sie den ODBC-Alias in einem Delphi-Projekt verwenden, dann ist es eine vernünftige Idee, eine *TDatabase*-Instanz einzufügen und deren Eigenschaft *LoginPrompt* auf *false* zu setzen.

21.3 Einrichten einer ADO-Verbindung

Im Gegensatz zu den eben gezeigten Wegen gehen Sie bei einer ADO-Verbindung nicht über die BDE und somit auch nicht über die Komponenten *TTable* und *TQuery*. Versuchen Sie nicht, in der BDE-Verwaltung etwas für eine ADO-Verbindung zu konfigurieren.

Um eine ADO-Verbindung aufzubauen, benötigen Sie zunächst die Komponente *TADOConnection* von der Komponentenseite ADO. Rufen Sie mit einem Doppelklick auf diese Komponente den Komponenteneditor auf.

Zunächst werden Sie gefragt, ob Sie einen Verbindungs-String eingeben oder eine Datenverknüpfungsdatei verwenden möchten. Wir nehmen den Verbindungs-String und betätigen dann den Button ERSTELLEN.

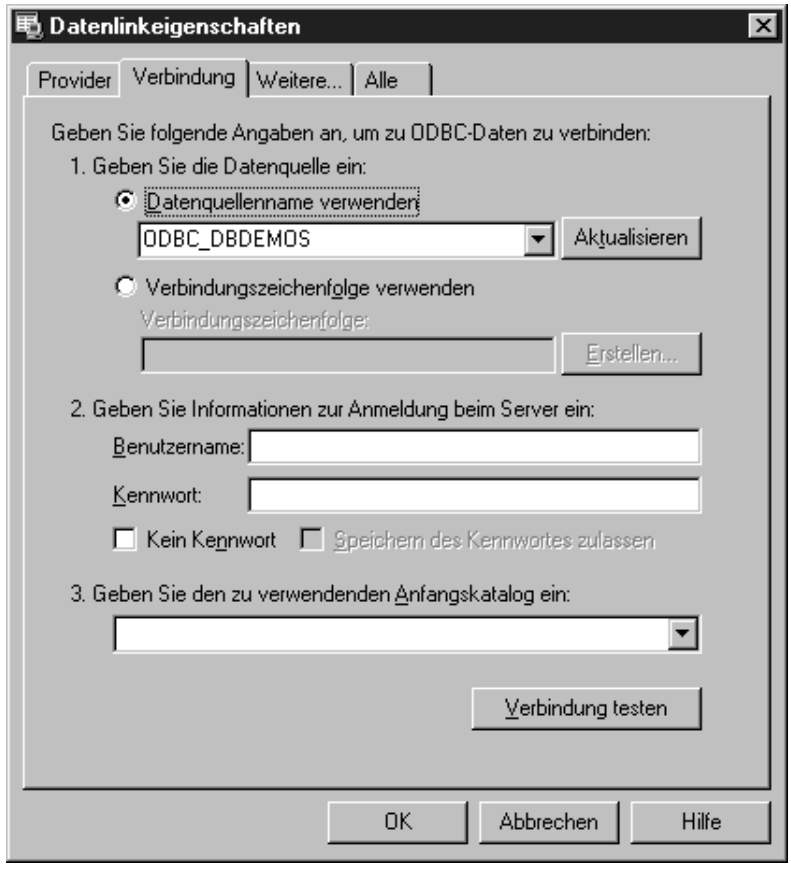
Nun öffnet sich ein mehrseitiger Registerdialog, zunächst mit der Registerseite *Provider*. Hier haben wir prinzipiell zwei Möglichkeiten, wenn wir auf Access-Datenbanken zugreifen möchten: *Microsoft Jet 4.0 OLE DB Provider* oder *Microsoft OLE DB Provider for ODBC Drivers*. Es sollen beide Wege kurz besprochen werden.

ADO über die Jet-Engine

Wenn Sie *Microsoft Jet 4.0 OLE DB Provider* wählen, dann können Sie auf der Registerseite *Verbindung* den Datenbanknamen auswählen. Anschließend sollten Sie den Button VERBINDUNG TESTEN betätigen.

ADO über ODBC

Wenn Sie *Microsoft OLE DB Provider for ODBC Drivers* wählen, dann können Sie auf der Registerseite *Verbindung* den Datenquellennamen wählen.



Anschließend sollten Sie auch hier die Verbindung testen.

Ist alles in Ordnung, dann schließen Sie in beiden Fällen den Dialog *Datenlinkeigenschaften* und auch den Komponenteneditor von *ADOConnection1* mit OK. Des Weiteren wird die Eigenschaft *LoginPrompt* auf *false* und anschließend *Connected* auf *true* gesetzt.

Zugriff auf eine Tabelle

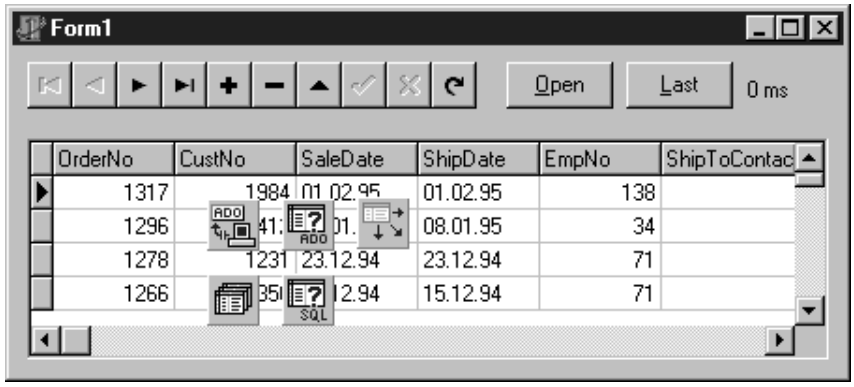
Wir wollen nun mit einer ADO-Datenmengenkomponente auf eine Tabelle in der Datenbank zugreifen. Für eine vergleichende Geschwindigkeitsmessung sind in den Tabellen von DBDEMOS leider nicht ausreichend viele Datensätze. Ersatzweise nehmen wir eine SQL-Anweisung.

Fügen Sie also eine *TADOQuery*-Komponente hinzu und setzen Sie deren Eigenschaft *Connection* auf *ADOConnection1*.

Der Eigenschaft *SQL* weisen wir folgendes Statement zu:

```
SELECT *
  FROM ORDERS
  WHERE shipvia = 'DHL'
  ORDER BY shipdate DESC
```

Anschließend kann *Active* auf *true* gesetzt werden. An *ADOQuery1* wird nun eine *TDataSource*-Instanz und an diese *DBGrid1* und *DBNavigator1* gehängt.



Um die beiden Möglichkeiten der Verbindung vergleichen zu können, wurde für das Öffnen und für das Aufrufen des letzten Datensatzes jeweils eine Zeitmessung durchgeführt.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  c, t1, t2: int64;
begin
  QueryPerformanceFrequency(c);
  QueryPerformanceCounter(t1);
  ADOQuery1.Open;
  // ADOQuery1.Last;
  QueryPerformanceCounter(t2);
  Label1.Caption := IntToStr(1000 * (t2 - t1) div c) + ' ms';
end;
```

Deren Ergebnisse verbessern sich natürlich, wenn die Daten bereits im Cache liegen. Es wurde das Öffnen der Abfrage (mit anschließendem Beenden des Programms) so lange wiederholt, bis die Werte ziemlich stabil waren.

- Bei der ODBC-Verbindung dauerte das Öffnen etwa 630 ms und der Aufruf des letzten Datensatzes etwa 12 ms.
- Über die Jet-Engine dauerte *Open* etwa 840 ms und *Last* etwa 11 ms.
- Außer Konkurenz: die Paradox-Tabelle aus der Datenbank DBDEMOS über die BDE und *TQuery*: *Open* etwa 400 ms, *Last* etwa 8 ms. Allerdings wurde dabei keine aktualisierbare Datenmenge zur Verfügung gestellt – auch dann nicht, wenn *RequestLive* auf *true* gestellt wurde.

21.4 Aufbau einer Master-Detail-Verknüpfung

Nachdem erste Versuche neugierig gemacht hatten, habe ich den Datenbestand der Testdatenbank in eine Access-Datenbank portiert (mit dem Data-Pump-Experten etwa eine Stunde Zeitaufwand auf meinem Notebook) und die dreistufige Master-Detail-Verbindung aus Kapitel 1 nachgebaut. Die Erfahrungen dabei waren recht unterschiedlich.

Master-Detail-Verknüpfung mit TADOTable

Der Aufbau einer Master-Detail-Verknüpfung erfolgt ähnlich wie bei *TTable*. Es wird dabei – ähnlich wie bei *TTable* im Zugriff auf Client-Server-Systeme – kein Index benötigt, im Feldverbindungs-Designer stehen somit alle Spalten zur Auswahl.

Im Zeitvergleich zu *TTable* und *BDE* schnitt ADO beim Öffnen ein wenig schlechter ab (etwa 280 ms statt 240 ms), beim Aufruf des nächsten Datensatzes lagen die Zeiten gleich (etwa 11 ms).

Was jedoch deutlich länger dauert, ist das Öffnen von Datenmengen zur Entwurfszeit. Auch das Erzeugen von persistenten *TField*-Instanzen für alle Tabellenspalten wird zur Geduldsprobe.

Master-Detail-Verknüpfung mit TADOQuery

Der eigentliche Beweggrund, die dreistufige Master-Detail-Verknüpfung nachzubauen, war natürlich *TADOQuery*. Nachdem die Sache mit *TQuery* und der *BDE* quälend langsam war, wollte ich sehen, ob es mit ADO hier schneller geht. Dies ist im Scrollen tatsächlich der Fall – vorausgesetzt, es steht genügend Arbeitsspeicher zur Verfügung.

Leider ist jedoch bei *TADOQuery* der SQL-Sprachumfang stark limitiert (noch viel stärker als bei der BDE). Vor allem sind die Fehlermeldungen weit weniger zielführend als bei der BDE.

Schauen wir uns die einzelnen SQL-Anweisungen doch einmal genauer an:

```
SELECT *
  FROM t_kunde
  WHERE vorname LIKE :vorname
    AND nachname LIKE :nachname
    AND strasse LIKE :strasse
    AND plz LIKE :plz
    AND ort LIKE :ort
```

Eine Funktion UPPER kennt man hier nicht, LOWER genauso wenig und UPPERCASE auch nicht.

Des Weiteren gibt es bei *TADOQuery* die Methode *ParamByName* nicht und auf die Parameter greift man dann anstatt über die Eigenschaft *Params* über die Eigenschaft *Parameters* zu.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  if Form3.ShowModal = mrOK then
  begin
    with DataModule2.ADOQuery1 do
    begin
      Close;
      Parameters[0].Value
        := Form3.Edit1.Text + '%';
          ...
      Parameters[4].Value
        := Form3.Edit5.Text + '%';
      Open;
    end; {with DataModule2.Query1 do}
  end; {if Form3.ShowModal = mrOK then}
end; {procedure TForm1.BitBtn1Click}
```

Sie müssen also gut aufpassen, wenn Sie die SQL-Anweisung umstellen und dabei die Parameter – oder auch nur deren Reihenfolge – verändern.

Nun zu der Anweisung der ersten Detailtabelle:

```
SELECT
    b.nummer,
    b.datum,
    m.nachname
  FROM t_bestellung b
    INNER JOIN t_mitarbeiter m
```

```
        ON b.bearbeiter = m.nummer
    WHERE b.kunde = :nummer
```

Einen JOIN kennt man hier, er wird sogar – im Gegensatz zur BDE – mit akzeptabler Geschwindigeit ausgeführt. Auf die Idee, eine Unterabfrage oder eine GROUP BY-Klausel zu verwenden, brauchen Sie hier allerdings nicht zu kommen.

Beachten Sie weiter, dass – im Gegensatz zu *TQuery* – bei *TADOQuery* auch bei einer Master-Detail-Verknüpfung dem Parameter ein Typ und ein Wert zuzuweisen ist. Nehmen Sie hier als Wert die Zahl eins, zur Laufzeit wird das dann von *DataSource1* übernommen.

Nun zur zweiten Detailtabelle:

```
SELECT
    p.nummer,
    p.stueckzahl,
    a.bezeichnung,
    p.preis
  FROM t_posten p
    INNER JOIN t_artikel a
      ON p.artikel = a.nummer
  WHERE p.bestellung = :nummer
```

Berechnete Felder funktionieren ebenfalls nicht, so dass auch hier ein einfacher JOIN verwendet wird. Dem Parameter sind wieder Typ und Wert zuzuweisen.

Vergleich der Ausführungszeiten

Nun zu der spannenden Frage, wie schnell die Sache denn läuft:

- Bei ADO dauert das Öffnen der Datenmenge 4020 ms, dafür das Scrollen nur etwa 250 ms.
- Das Öffnen geht mit Paradox-Tabellen etwas schneller, nämlich 2520 ms. Dafür wird aber das Scrollen – dank der Verwendung von zwei JOINS – quälend langsam, nämlich etwa 1900 ms.
- Die Frage, ob ADO so schnell oder die BDE so langsam ist, lässt sich recht eindeutig beantworten, wenn man den Vergleich mit InterBase durchführt: Öffnen der Datenmenge in 960 ms, Scrollen in etwa 60 ms.

  ADO ist etwa um den Faktor vier langsamer und über das Thema JOIN und BDE schweigen wir mal lieber ...

21.5 Referenz der ADO-Komponenten

21.5.1 TADOConnection

Die Komponente *TADOConnection* stellt die Verbindung zu einem Provider her. Die einzelnen ADO-Datenmengenkomponenten können diese Verbindungen dann nutzen. Es besteht jedoch auch die Möglichkeit, eine ADO-Datenmengenkomponente direkt mit einem Provider zu verbinden. (*TADOConnection* ist das Gegenstück zu *TDatabase*.)

- ConnectionString, (Eigenschaft, veröffentlicht)

  ```
  property ConnectionString: WideString;
  ```

  Mit der Eigenschaft *ConnectionString* kann eine Verbindung zu einem Provider hergestellt werden. Für diese Eigenschaft existiert ein Eigenschaftseditor.

- Provider, DefaultDatabase, ConnectionTimeout (Eigenschaften, veröffentlicht)

  ```
  property Provider: WideString;
  property DefaultDatabase: WideString;
  property ConnectionTimeout: Integer;
  ```

  Diesen drei Eigenschaften ist gemeinsam, dass ihr Wert aus dem *ConnectionString* übernommen wird, solange ihnen nicht explizit ein Wert zugewiesen wird. Mit *Provider* wird der ADO-Provider eingestellt, von dem die Daten bezogen werden, mit *DefaultDatabase* die dabei verwendete Datenbank.

  Die Eigenschaft *ConnectionTimeout* gibt (in Sekunden) an, wie lange das Öffnen einer Verbindung dauern kann. Ist nach dieser Zeit noch keine Verbindung zustande gekommen (oder *Cancel* aufgerufen worden), wird eine Exception ausgelöst.

- Connected (Eigenschaft, veröffentlicht), Open, Close, Cancel (Methoden)

  ```
  property Connected: Boolean;
  procedure Open; overload;
  procedure Open(const UserID: WideString;
    const Password: WideString); overload;
  procedure Close;
  procedure Cancel;
  ```

  Sie können die Verbindung zu einem ADO-Provider herstellen, indem Sie *Connected* auf *true* setzen oder *Open* aufrufen. Von *Open* existiert auch eine überladene Version, der Sie Benutzernamen und Passwort als Parameter über-

geben können – alternativ dazu können diese Angaben auch im *ConnectionString* erfolgen.

Um die Verbindung zu trennen, setzen Sie *Connected* auf *false* oder rufen *Close* auf. Den Versuch eines Verbindungsaufbaus können Sie mit *Cancel* abbrechen.

- BeginTrans, CommitTrans, RollbackTrans (Methoden),
  InTransaction (Eigenschaft, öffentlich, nur Lesen)

```
function BeginTrans: Integer;
procedure CommitTrans;
procedure RollbackTrans;
property InTransaction: Boolean
```

Mit *BeginTrans* wird eine neue Transaktion begonnen, mit *CommitTrans* diese bestätigt, mit *RollbackTrans* verworfen. Mit der Eigenschaft *InTransaction* kann ermittelt werden, ob momentan eine Transaktion aktiv ist.

Des Weiteren implementiert *TADOConnection* eine Reihe von Ereignissen, die vor beziehungsweise nach einer Aktion ausgelöst werden. Außerdem stehen die Array-Eigenschaften *Commands* und *DataSets* zur Verfügung, die Referenzen auf die verbundenen Befehls- und Datenmengenkomponenten liefern. Mit Hilfe von Eigenschaften wie *IsolationLevel*, *Mode*, *Attributes* oder *CursorLocation* lässt sich das genaue Verhalten der Verbindung spezifizieren.

21.5.2 TADOCommand

Mit *TADOCommand* können DML- oder DDL-Anweisungen an den Provider geschickt werden. *TADOCommand* implementiert keine Ereignisse, Sie können jedoch die Ereignisse *OnWillExecute* und *OnExecuteComplete* von *TADOConnection* verwenden.

- Connection, ConnectionString (Eigenschaften, veröffentlicht)

```
property Connection: TADOConnection;
property ConnectionString: WideString;
```

Die Komponente *TADOCommand* wird über die mit *Connection* spezifizierte Komponente mit einem Provider verbunden. Alternativ dazu kann auch ein eigener *ConnectionString* verfasst werden.

- CommandText (Eigenschaften, veröffentlicht)

```
property CommandText: WideString;
```

Mittels der Eigenschaft *CommandText* wird die auszuführende Anweisung zugewiesen.

```
ADOCommand1.CommandText := 'DROP TABLE t_test';
```

- Execute, Cancel (Methoden), Parameters (Eigenschaft, veröffentlicht)

```
function Execute: _RecordSet; overload;
function Execute(const Parameters: OleVariant): _Recordset;
  overload;
function Execute(var RecordsAffected: Integer;
  var Parameters: OleVariant;
  ExecuteOptions: TExecuteOptions = []): _RecordSet; overload;
procedure Cancel;
property Parameters: TParameters;
```

  Mit *Execute* wird die Anweisung ausgeführt. Es existieren überladene Versionen, welche die Übergabe von Parametern und Optionen erlauben. Man kann auch mit einem Variablenparameter die Anzahl der betroffenen Datensätze ermitteln.

  Als Funktionsergebnis wird ein Zeiger auf eine Ergebnismenge zurückgegeben – für Anweisungen, die Ergebnismengen zurückliefern, nehmen Sie jedoch besser andere Komponenten.

  Um die Ausführung einer Anweisung abzubrechen, verwenden Sie *Cancel*.

  Zur Übergabe von Parametern verwenden Sie die Eigenschaft *Parameters*.

21.5.3 TCustomADODataSet

Von der Vorfahrenkomponente *TCustomADODataSet* sind die Komponenten *TADODataSet*, *TADOTable*, *TADOQuery* und *TADOStoredProc* abgeleitet. Um unnötige Wiederholungen zu vermeiden, sollen hier *TCustomADODataSet* und somit die gemeinsamen Eigenschaften, Methoden und Ereignisse dieser vier ADO-Datenmengenkomponenten besprochen werden.

TCustomADODataSet ist wiederum von der in Kapitel 3.9 besprochenen Komponente *TDataSet* abgeleitet.

Verbindung zur Datenbank

- Connection, ConnectionString (Eigenschaften, veröffentlicht)

```
property Connection: TADOConnection;
property ConnectionString: WideString;
```

  Die ADO-Datenmengenkomponente wird über die mit *Connection* spezifizierte Komponente mit einem Provider verbunden. Alternativ dazu kann auch ein eigener *ConnectionString* verfasst werden.

- SaveToFile, LoadFromFile (Methoden)

```
procedure SaveToFile(const FileName: String = '';
  Format: TPersistFormat = pfADTG);
procedure LoadFromFile(const FileName: WideString);
```

Mit diesen beiden Methoden kann eine Datenmenge in eine Datei gespeichert oder daraus geladen werden.

Navigieren in der Datenmenge

- RecordCount (Eigenschaft, öffentlich, nur Lesen),

```
property RecordCount: Integer;
```

Mit *RecordCount* kann die Zahl der Datensätze ermittelt werden.

- RecNo (Eigenschaft, öffentlich), IsSequenced (Methode)

```
property RecNo: Integer;
function IsSequenced: Boolean;
```

Mit *RecNo* können Sie ermitteln, auf welchem Datensatz der Datenzeiger steht. Sie können aber auch den Datenzeiger bewegen, indem Sie *RecordNo* einen Wert zuweisen. Das funktioniert allerdings nur dann, wenn die Datenmenge die Datensatznummern verwendet und *IsSequenced* somit den Wert *true* hat.

- MaxRecords (Eigenschaft, veröffentlicht)

```
property MaxRecords: Integer
```

Mit *MaxRecords* kann eingestellt werden, wie viele Datensätze maximal in die Ergebnismenge aufgenommen werden sollen. Auf diese Weise kann verhindert werden, dass bei unzureichend gesetzten Filtern Unmengen von Datensätzen vom Server zum Client übertragen werden.

Ändern der Daten

- Supports (Methode)

```
function Supports(CursorOptions: TCursorOptions): Boolean;
TCursorOptions = set of (coHoldRecords, coMovePrevious, coAddNew,
  coDelete, coUpdate, coBookmark, coApproxPosition, coUpdateBatch,
  coResync, coNotify, coFind , coSeek, coIndex);
```

Mit *Supports* kann ermittelt werden, ob die Datenmenge eine bestimmte Operation – beispielsweise das Löschen eines Datensatzes – überhaupt unterstützt.

- DeleteRecords (Methode)

```
procedure DeleteRecords(AffectRecords: (arCurrent, arFiltered,
  arAll, arAllChapters) = arAll);
```

Mit *DeleteRecords* können mehrere Datensätze gleichzeitig gelöscht werden – beispielsweise alle im aktuellen Filter (*arFiltered*) oder alle in der aktuellen Tabelle (*arAll*).

- LockType (Eigenschaft, veröffentlicht)

```
property LockType: (ltUnspecified, ltReadOnly, ltPessimistic,
  ltOptimistic, ltBatchOptimistic);
```

Mit *LockType* kann spezifiziert werden, wie der konkurrierende Zugriff auf Daten geregelt wird. Hat *LockType* den Wert *ltReadOnly*, dann können die Daten nur gelesen werden.

Bei einer pessimistischen Datensatzsperre wird ein Datensatz gesperrt, wenn seine Felder bearbeitet werden – diese können dann auch nicht mehr gelesen werden. Bei einer optimistischen Sperre werden die Datensätze auch dann nicht gesperrt, wenn sie geändert werden. Wenn aber zwei (oder mehr) Clients am gleichen Datensatz Änderungen eintragen wollen, dann führt nur die erste Änderung zum Erfolg, alle anderen Änderungen lösen eine Exception aus.

- CanModify (Eigenschaft, öffentlich, nur Lesen)

```
property CanModify: Boolean;
```

Mit *CanModify* kann ermittelt werden, ob die Daten der Datenmenge geändert werden können.

- Clone (Methode)

```
procedure Clone(Source: TCustomADODataSet;
  LockType: TLockType = ltUnspecified);
```

Mit *Clone* kann die Ergebnismenge aus einer anderen ADO-Datenmenge kopiert werden.

Suchen und Filtern, Lesezeichen

- Seek (Methode)

```
function Seek(const KeyValues: Variant;
  SeekOption: TSeekOption = soFirstEQ): Boolean;
```

Mit *Seek* wird ein Datensatz gesucht, wobei der aktuelle Index verwendet wird. Mit *SeekOptions* wird spezifiziert, wo denn der Datenzeiger positioniert wer-

den soll – beispielsweise auf dem ersten (*soFirstEQ*) oder dem letzten (*soLastEQ*) gefundenen Vorkommen.

- FilterGroup (Eigenschaft, öffentlich)

```
property FilterGroup: (fgUnassigned, fgNone, fgPendingRecords,
  fgAffectedRecords, fgFetchedRecords, fgPredicate,
  fgConflictingRecords);
```

Mit *FilterGroup* können bestimmte Gruppen von Datensätzen angezeigt werden. So werden beispielsweise mit *fgPredicate* die gelöschten Datensätze angezeigt.

- BookmarkValid (Methode)

```
function BookmarkValid(Bookmark: TBookmark): Boolean;
```

Mit *BookmarkValid* kann geprüft werden, ob ein Lesezeichen auf einen gültigen Datensatz zeigt.

- CompareBookmarks (Methode)

```
function CompareBookmarks
  (Bookmark1, Bookmark2: TBookmark): Integer;
```

Mit *CompareBookmarks* können zwei Lesezeichen miteinander verglichen werden.

- FilterOnBookmarks (Methode)

```
procedure FilterOnBookmarks(Bookmarks: array of const);
```

Mit *FilterOnBookmarks* kann eine Datenmenge auf die übergebenen Lesezeichen gefiltert werden.

Indizes

- IndexFields (Eigenschaft, öffentlich)

```
property IndexFields[Index: Integer]: TField;
```

Mit *IndexFields* kann man Referenzen auf die *TField*-Instanzen der im aktuellen Index enthaltenen Felder bekommen.

- IndexFieldCount (Eigenschaft, öffentlich, nur Lesen)

```
property IndexFieldCount: Integer;
```

Mit *IndexFieldCount* kann die Anzahl der im aktuellen Index enthaltenen Felder ermittelt werden.

- Sort (Eigenschaft, veröffentlicht)

```
property Sort: WideString;
```

Mit *Sort* wird die Sortierung der Datenmenge spezifiziert. Es ist nicht erforderlich, dass die betreffenden Spalten bereits über einen Index verfügen – gegebenenfalls wird ein solcher temporär erstellt. Wenn Sie einer Spalte das Schlüsselwort *DESC* anhängen, dann wird diese absteigend sortiert.

```
ADOQuery1.Sort := 'Nachname, Datum DESC'
```

21.5.4 TADODataSet

Mittels *TADODataSet* können Sie die Verbindung zu einer Datenmenge zur Verfügung stellen. An *TADODataSet*-Instanzen können Sie Datenquellenkomponenten hängen, um so datensensitive Dialogkomponenten damit zu verknüpfen.

- CommandText, CommandType (Eigenschaften, veröffentlicht)

```
property CommandText: WideString;
property CommandType: TCommandType;
```

Mit *CommandText* wird die Komponente angewiesen, wo sie sich die Daten holen soll. Dabei wird mit *CommandType* spezifiziert, wie der in *CommandText* enthaltene Text zu interpretieren ist:

- *cmdText*: als SQL-Anweisung
- *cmdTable*: als Name der zu verwendenden Tabelle
- *cmdStoredProc*: als Name der zu verwendenden Prozedur
- *cmdFile*: als Dateiname einer gespeicherten Ergebnismenge

```
with ADODataSet1 do
begin
  CommandType := cmdText;
  CommandText := 'SELECT * FROM t_test';
  Open;
end;
```

Wenn Sie *CommandType* auf *cmdTable* setzen, kann im Objektinspektor mit einer ComboBox die Eigenschaft *CommandText* aus der Liste der vorhandenen Tabellen ausgewählt werden.

- GetIndexNames (Methode)

  ```
  procedure GetIndexNames(List: TStrings);
  ```

  Mit *GetIndexNames* können die Namen der verfügbaren Indizes ermittelt und in eine String-Liste geschrieben werden.

- Parameters (Eigenschaft, veröffentlicht)

  ```
  property Parameters: TParameters;
  ```

  Mit der *TCollection*-Eigenschaft *Parameters* können Werte für Parameter, SQL-Anweisungen und Prozeduren zugewiesen werden.

21.5.5 TADOTable

TADOTable dient dem Zugriff auf eine einzelne Tabelle und ähnelt der Komponente *TTable*.

- TableName (Eigenschaft, veröffentlicht)

  ```
  property TableName: WideString;
  ```

  Mit *TableName* wird der Name der zu verwendenden Tabelle angegeben.

- MasterSource, MasterFields (Eigenschaften, veröffentlicht)

  ```
  property MasterSource: TDataSource;
  property MasterFields: String;
  ```

  Um eine Master-Detail-Verknüpfung zu erstellen, wird mit *MasterSource* die Datenquellenkomponente der Master-Datenmenge spezifiziert.

  Um anzugeben, über welche Spalten die Verknüpfung erstellt werden soll, wird die Eigenschaft *MasterFields* verwendet. Für die Eigenschaft *MasterFields* existiert ein Eigenschaftseditor, nämlich der in Kapitel 2.2 bereits beschriebene Feldverbindungs-Designer.

  Die für die Master-Detail-Verknüpfung verwendeten Spalten brauchen nicht indiziert zu sein, der Feldverbindungs-Designer sieht auch gar nicht vor, einen Index zu wählen.

- ReadOnly (Eigenschaft, veröffentlicht)

  ```
  property ReadOnly: Boolean
  ```

  Soll die Tabelle nur zum Lesen geöffnet werden, dann ist *ReadOnly* auf *true* zu setzen.

- IndexFieldNames (Eigenschaft, veröffentlicht)

  ```
  property IndexFieldNames: String;
  ```

  Mit *IndexFieldNames* kann die Datenmenge sortiert werden. Geben Sie die Spalten ein, nach welchen sortiert werden soll, und trennen Sie die einzelnen Spaltennamen mit Semikola. Es ist nicht erforderlich, dass die beteiligten Spalten indiziert sind.

- GetIndexNames (Methode)

  ```
  procedure GetIndexNames(List: TStrings);
  ```

  Mit *GetIndexNames* können die Namen der verfügbaren Indizes ermittelt und in eine String-Liste geschrieben werden.

21.5.6 TADOQuery

Um die Ergebnismenge zu spezifizieren, verwendet *TADOQuery* eine SQL-Anweisung. Diese Komponente ähnelt *TQuery*.

- SQL (Eigenschaft, veröffentlicht)

  ```
  property SQL: TStrings;
  ```

  Mit SQL geben Sie die Anweisung ein, mit der die Ergebnismenge erzeugt wird. Verwenden Sie dabei nur die SQL-Anweisungen, die vom jeweiligen Provider auch verstanden werden.

- Parameters (Eigenschaft, veröffentlicht)

  ```
  property Parameters: TParameters;
  ```

  Mit der *TCollection*-Eigenschaft *Parameters* können den SQL-Anweisungen Werte für Parameter zugewiesen werden. Beachten Sie dabei, dass es – im Gegensatz zu TQuery – eine Methode *ParamByName* nicht gibt. Seien Sie von daher besonders beim Umformulieren der SQL-Anweisung sehr vorsichtig.

- DataSource (Eigenschaft, veröffentlicht)

  ```
  property DataSource: TDataSource;
  ```

  Mit *DataSource* können Sie eine Datenquellenkomponente benennen, aus der die Werte für die Parameter bezogen werden. Das wird in der Regel dazu verwendet, um Master-Detail-Verknüpfungen zu erstellen. Ein Beispiel dafür finden Sie in Kapitel 21.4.

  Beachten Sie bitte, dass zur Entwurfszeit – im Gegensatz zu *TQuery* – der Typ und ein Wert für jeden Parameter anzugeben sind, auch wenn dieser aus der Datenquellenkomponente bezogen wird.

- ExecSQL (Methode), RowsAffected (Eigenschaft, öffentlich, nur Lesen)

```
function ExecSQL: Integer;
property RowsAffected: Integer;
```

  Es gibt SQL-Anweisungen, die keine Ergebnismenge zurückliefern (INSERT, UPDATE, DELETE, CREATE TABLE ...). Solche Anweisungen dürfen nicht mit *Open* gestartet werden, es muss dazu *ExecSQL* verwendet werden.

  Mit *RowsAffected* kann ermittelt werden, wie viele Datensätze bei der letzten *ExecSQL*-Anweisung bearbeitet wurden.

21.5.7 TADOStoredProc

Mit *TADOStoredProc* kann man auf gespeicherte Prozeduren zugreifen. Die Komponente ähnelt *TStoredProc*. Beachten Sie bitte, dass Prozeduren nicht von allen Datenbanksystemen unterstützt werden.

- ProcedureName (Eigenschaft, veröffentlicht)

```
property ProcedureName: WideString;
```

  Mit *ProcedureName* wird der Name der zu verwendenden Prozedur spezifiziert.

- Parameters (Eigenschaft, veröffentlicht)

```
property Parameters: TParameters;
```

  Mit der *TCollection*-Eigenschaft *Parameters* können der Prozedur Werte für Parameter zugewiesen werden. Beachten Sie dabei, dass es – im Gegensatz zu *TQuery* – eine Methode *ParamByName* nicht gibt. Informieren Sie sich genau über die Reihenfolge der Parameter.

Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt. Dieses eBook stellen wir lediglich als persönliche Einzelplatz-Lizenz zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschliesslich

- der Reproduktion,
- der Weitergabe,
- des Weitervertriebs,
- der Platzierung im Internet, in Intranets, in Extranets,
- der Veränderung,
- des Weiterverkaufs
- und der Veröffentlichung

bedarf der schriftlichen Genehmigung des Verlags.

Insbesondere ist die Entfernung oder Änderung des vom Verlag vergebenen Passwortschutzes ausdrücklich untersagt!

Bei Fragen zu diesem Thema wenden Sie sich bitte an: info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf unseren Websites ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und viele weitere eBooks können Sie rund um die Uhr und legal auf unserer Website

http://www.informit.de

herunterladen